MicroEJ Documentation

MicroEJ Corp.

Mar 13, 2026

Copyright 2008-2024, MicroEJ Corp. Content in this space is free for read and redistribute. Except if otherwise stated, modification is subject to MicroEJ Corp prior approval. MicroEJ is a trademark of MicroEJ Corp. All other trademarks and copyrights are the property of their respective owners.

CONTENTS

1 Navigation **2**

2 Useful Links **3**

3 Useful Resources **4**

3.1 MicroEJ Glossary . . . 4
3.2 Overview . . . 6
3.2.1 What is MicroEJ? . . . 6
3.2.2 MICROEJ VEE . . . 6
3.2.3 MICROEJ SDK . . . 7
3.2.4 Build Workflow . . . 8
3.3 Getting Started . . . 10
3.3.1 VEE Ports for Evaluation . . . 11
3.3.2 VEE Port Examples . . . 50
3.4 SDK 6 User Guide . . . 50
3.4.1 Installation . . . 51
3.4.2 Licenses . . . 59
3.4.3 Sentinel License Management . . . 71
3.4.4 Scope and Limitations . . . 78
3.4.5 Create a Project . . . 79
3.4.6 Import a Project . . . 95
3.4.7 Select a VEE Port . . . 98
3.4.8 Run on Simulator . . . 102
3.4.9 Build an Executable . . . 110
3.4.10 Run on Device . . . 112
3.4.11 Select a Kernel . . . 114
3.4.12 Build a Feature file . . . 116
3.4.13 Build a Virtual Device . . . 120
3.4.14 Add a Dependency . . . 124
3.4.15 Test a Project . . . 126
3.4.16 Publish a Project . . . 148
3.4.17 Development Tools . . . 149
3.4.18 VEE Port . . . 200
3.4.19 Module Repository . . . 219
3.4.20 Manage Versioning . . . 231
3.4.21 Manage Resolution Conflicts . . . 231
3.4.22 Migrate an SDK 5 Project . . . 233
3.4.23 Module Natures . . . 248
3.4.24 Troubleshooting . . . 264
3.4.25 Tutorials . . . 277

3.4.26 How-to Guides . . . 286
3.4.27 Appendices . . . 312
3.4.28 Changelog . . . 328
3.4.29 Migration Notes . . . 346
3.5 Application Developer Guide . . . 351
3.5.1 MicroEJ Runtime . . . 353
3.5.2 SOAR . . . 405
3.5.3 SOAR Output Files . . . 408
3.5.4 Virtual Device . . . 413
3.5.5 MicroEJ Classpath . . . 413
3.5.6 Application Resources . . . 420
3.5.7 Standalone Application . . . 421
3.5.8 Sandboxed Application . . . 455
3.5.9 Character Encoding . . . 466
3.5.10 Limitations . . . 468
3.5.11 GitHub Repositories . . . 469
3.5.12 Module Repositories . . . 474
3.5.13 Libraries . . . 476
3.5.14 Development Tools . . . 743
3.6 VEE Porting Guide . . . 788
3.6.1 Introduction . . . 788
3.6.2 MicroEJ Architecture . . . 793
3.6.3 MicroEJ Packs . . . 836
3.6.4 BSP Connection . . . 838
3.6.5 VEE Port Qualification . . . 845
3.6.6 Core Engine . . . 851
3.6.7 Event Tracing . . . 866
3.6.8 Advanced Event Tracing . . . 867
3.6.9 Multi-Sandbox . . . 868
3.6.10 Tiny-Sandbox . . . 879
3.6.11 Foundation Libraries . . . 880
3.6.12 Native Interface Mechanisms . . . 886
3.6.13 External Resources Loader . . . 891
3.6.14 Serial Communications . . . 894
3.6.15 Graphical User Interface . . . 905
3.6.16 Vector Graphics . . . 1221
3.6.17 Networking . . . 1272
3.6.18 Bluetooth . . . 1278
3.6.19 Audio . . . 1279
3.6.20 MicroAI . . . 1279
3.6.21 Event Queue . . . 1280
3.6.22 File System . . . 1281
3.6.23 GNSS . . . 1285
3.6.24 Hardware Abstraction Layer . . . 1286
3.6.25 Device Information . . . 1289
3.6.26 Security . . . 1290
3.6.27 Watchdog Timer . . . 1291
3.6.28 SystemView . . . 1299
3.6.29 Simulation . . . 1313
3.6.30 Appendices . . . 1346
3.7 Kernel Developer Guide . . . 1456
3.7.1 Overview . . . 1456
3.7.2 Kernel Creation . . . 1459
3.7.3 Kernel APIs . . . 1465

3.7.4 Runtime Environment . . . 1467
3.7.5 Kernel UID . . . 1473
3.7.6 Sandboxed Application Lifecycle . . . 1474
3.7.7 Define a Security Policy . . . 1474
3.7.8 Resource Manager . . . 1480
3.7.9 Kernel and Features Communication . . . 1483
3.7.10 Multi-Sandbox Enabled Libraries . . . 1489
3.7.11 Setup a KF Testsuite . . . 1494
3.7.12 Kernel Linking . . . 1500
3.7.13 Application Linking . . . 1503
3.8 VEE Wear User Guide . . . 1509
3.8.1 VEE Wear Framework . . . 1511
3.8.2 Android Compatibility Kit . . . 1516
3.8.3 iOS Compatibility Kit . . . 1528
3.8.4 Offloading . . . 1530
3.8.5 VEE Script . . . 1531
3.9 VEE Energy User Guide . . . 1533
3.10 Training Courses . . . 1534
3.10.1 For Beginners . . . 1534
3.10.2 For Application Developers . . . 1543
3.10.3 For VEE Port Developers . . . 1587
3.10.4 For Kernel Developers . . . 1635
3.10.5 Graphical User Interface . . . 1643
3.10.6 Connectivity . . . 1661
3.10.7 For DevOps . . . 1662
3.11 SDK 5 User Guide . . . 1674
3.11.1 Installation . . . 1676
3.11.2 Licenses . . . 1698
3.11.3 Standalone Application . . . 1711
3.11.4 Sandboxed Application . . . 1724
3.11.5 Module Repository . . . 1734
3.11.6 Select a VEE Port . . . 1739
3.11.7 Module Natures . . . 1740
3.11.8 Debug an Application . . . 1752
3.11.9 Development Tools . . . 1761
3.11.10 IDE . . . 1811
3.11.11 SDK Version . . . 1816
3.11.12 MicroEJ Module Manager . . . 1818
3.11.13 VEE Port . . . 1839
3.11.14 How-to Guides . . . 1859
3.11.15 Release Notes . . . 1860
3.11.16 SDK Distribution Changelog . . . 1861
3.11.17 SDK Changelog . . . 1864
3.11.18 Build Types per SDK . . . 1884
3.11.19 Migration Notes . . . 1891
3.12 Get Support . . . 1893
3.13 About MicroEJ . . . 1893

Welcome to the MicroEJ developer documentation.

MicroEJ (pronounced “micro-EDGE”) turns any electronic product or ‘things’ into smart things, MicroEJ tools and libraries enable you to easily create efficient and feature complete applications for embedded devices. MicroEJ leverages a runtime environment called VEE (Virtual Execution Environment) to simplify and secure embedded development.

CHAPTER

ONE

NAVIGATION

Browse the following chapters to familiarize yourself and understand the principles of development with MicroEJ Technology.

- The *MicroEJ Glossary* chapter describes MicroEJ terminology.
- The *Overview* chapter introduces MicroEJ products and technology.
- The *Getting Started* is the best way to start your journey with MicroEJ's technology. Set up a development environment and start coding!
- The *SDK 6 User Guide* chapter presents MICROEJ SDK 6, the Software Development Kit for developing new projects.
- The *Application Developer Guide* presents how to develop a Java or JavaScript application on MICROEJ VEE (Virtual Execution Environment).
- The *VEE Porting Guide* teaches you how to integrate MICROEJ VEE into a C Board Support Package as well as simulation configurations.
- The *Kernel Developer Guide* introduces you to advanced concepts, such as partial updates and dynamic app life cycle workflows.
- The *VEE Wear User Guide* addresses the development of smartwatch applications using VEE Wear, specifically designed for low-power MCU and MPU.
- The *Training Courses* chapter offers online courses to help you learn the essentials of Firmware development using MICROEJ SDK.

CHAPTER

TWO

USEFUL LINKS

- Resources Hub: gathers all useful resources for developers on a daily basis.
- Forum: provides announcements to keep up to date in MicroEJ technology releases. It also provides many topics that could answer your questions when developing on MicroEJ technology.
- Browse MicroEJ Module Repositories to look for dependencies:
  - Central Repository Modules.
  - Developer Repository Modules.
  - See *Module Repositories* for more information.
- Licenses Server

CHAPTER

THREE

USEFUL RESOURCES

- *SDK6 Cheat Sheet*: a must-have when working with MICROEJ SDK6.
- MicroEJ GitHub: provides several pinned repositories, a must-have to get started.

3.1 MicroEJ Glossary

Add-On Library
: An Add-On Library is a pure Managed Code (Java code, C code or Javascript code) library. It runs over one or more Foundation Libraries.

Abstraction Layer
: An Abstraction Layer is the native code (C code, Assembly code, etc.) that implements a Foundation Library's low-level APIs over a board support package (BSP) or a C library.

Application
: An Application is a software program that runs on a MICROEJ VEE.

  Standalone Application
  : A Standalone Application is the main application that is executed by MICROEJ VEE. It is linked statically to produce a Mono-Sandbox Executable.

  Sandboxed Application
  : A Sandboxed Application is an Application that can run over a Multi-Sandbox Executable. It is linked dynamically.

  Kernel Application
  : A Kernel Application is a Standalone Application that implements the ability to be extended to produce a Multi-Sandbox Executable.

Architecture
: An Architecture is a software package that includes the Core Engine port to a target instruction set and a C compiler, core Foundation Libraries (*[EDC], [BON], [SNI], [KF]*) and the Simulator. Architectures are distributed either as evaluation or production version.

BSP
: A Board Support Package (BSP) is a collection of native C code and configuration files that exposes the capabilities of a specific hardware board through low-level software APIs.

C Module
: A C Module is a collection of C source files packaged together with its own versioning and delivery lifecycle. A C Module represents an Abstraction Layer, a third-party library integration, or a set of reusable generic C code intended to be shared across different VEE Ports.

Core Engine
: The Core Engine, also named MEJ32, is a scalable 32-bit core for resource-constrained embedded devices. It is delivered in various flavors, mostly as a binary software package. The Core Engine allows applications written in various languages to run in a safe container.

Executable
: An Executable is the result of the binary link of a Standalone Application with a VEE Port. It can be programmed into the flash memory of a device.

  Mono-Sandbox Executable
  : A Mono-Sandbox Executable is an Executable that implements an unmodifiable set of functions.

  Multi-Sandbox Executable
  : A Multi-Sandbox Executable is an Executable that implements the ability to be extended, by exposing a Runtime Environment and a memory space to link Sandboxed Applications.

Foundation Library
: A Foundation Library is a library that provides core or hardware-dependent functionalities. A Foundation Library combines Managed Code (Java code, C code or Javascript code) and low-level APIs (C code, Assembly code, etc.) implemented by one or more Abstraction Layers through a native interface (*SNI*).

Managed Code
: Managed Code is code that is executed under the supervision of a virtual processor, rather than directly by the hardware processor. In the context of MicroEJ, the runtime responsible for managing code execution is called MICROEJ VEE. This approach ensures consistent execution behavior, portability across different hardware platforms, and secure memory isolation (sandboxing).

MICROEJ SDK
: MICROEJ SDK is a comprehensive tools suite for developers to build VEE Ports for their devices, create Applications, build Executable, and run Virtual Devices.

MICROEJ VEE
: MICROEJ VEE is an application container. VEE stands for Virtual Execution Environment, and refers to the first implementation that embeds a virtual 32-bit processor, hence the term “Virtual”. MICROEJ VEE runs on any OS/RTOS commonly used in embedded systems (FreeRTOS, QP/C, uc/OS, ThreadX, embOS, Mbed OS, Zephyr OS, VxWorks, PikeOS, Integrity, Linux, QNX, …) and can also run without RTOS (bare-metal) or proprietary RTOS. MICROEJ VEE includes the small MEJ32, along with a wide range of libraries (Add-On Libraries and Foundation Libraries).

Mock
: A Mock is a mockup of a board support package (BSP) capability that mimics a hardware functionality for the Simulator.

Runtime Environment
: A Runtime Environment is a custom set of APIs exposed by a Multi-Sandbox Executable, designed to support the development and execution of compatible Sandboxed Applications. It typically serves as the base component for building an ecosystem of applications within a specific domain.

Simulator
: The Simulator allows running Applications on a target hardware simulator on the developer’s desktop computer. The Simulator runs one or more Mock that mimics the hardware functionality. It enables developers to develop their Applications without the need of hardware.

VEE Port
: A VEE Port is an implementation of MICROEJ VEE tailored to run on a particular device (hardware board including both the processor and the peripherals). It integrates an Architecture, one or more Foundation Libraries with their respective Abstraction Layers, and the board support package (BSP). It also includes associated Mocks for the Simulator.

Virtual Device
: A Virtual Device is a software package that includes the simulation part of an Executable: runtime, libraries and application(s). It can be run on any desktop computer without the need of the SDK.

3.2 Overview

3.2.1 What is MicroEJ?

MicroEJ (pronounced “micro-EDGE”) is a software vendor of cost-driven solutions for embedded and IoT devices.

The MicroEJ solution is made up of two products:

- MICROEJ VEE (Virtual Execution Environment) is an application container for resource-constrained embedded devices running on microcontrollers or microprocessors. It allows devices to run multiple and mixed managed code (Java code, C code or JavaScript code).
- MICROEJ SDK allows developers to develop applications in Managed Code and deploy them to resource-constrained devices, such as microcontrollers.

The following video demonstrates the typical development process using MICROEJ SDK 6 Simulator:

3.2.2 MICROEJ VEE

MICROEJ VEE provides a fully configurable set of services that can be expanded, including but not limited to:

- a secure multi-application framework,
- a GUI framework (includes widgets),
- a network framework with security (SSL/TLS, HTTPS, REST, MQTT, …),
- a storage framework (file system),
- a Java Cryptography Architecture (JCA) implementation.

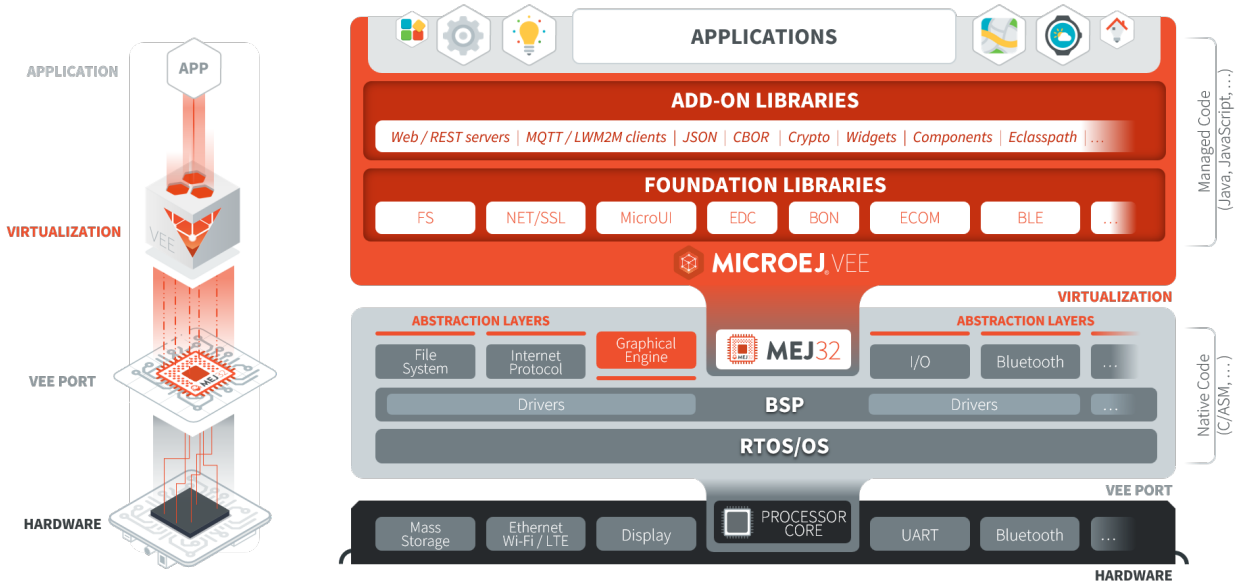


Fig. 1: MICROEJ VEE Overview

For more information about our Supported Processor Architectures, visit the Supported Hardware page.

3.2.3 MICROEJ SDK

MICROEJ SDK offers a comprehensive toolset to build the software for embedded devices.

The SDK covers two levels in device software development:

- Device integration and porting (adapting MICROEJ VEE to run on the target device).
- Application development.

Using the SDK, an application developer is able to:

- Develop and test applications on the Simulator.
- Deploy the application locally on the device.
- Package and publish the application on a repository or an application store, enabling remote end users to install it on their devices.

The SDK produces two versions of the application build:

- An Executable binary to be deployed on the device. It includes MICROEJ VEE, all device drivers and a specific set of functionalities useful for application developers targeting this device.
- A Virtual Device which is used as a device simulator by application developers, acting as a bridge to other stakeholders, including marketing teams, translators, and project managers.

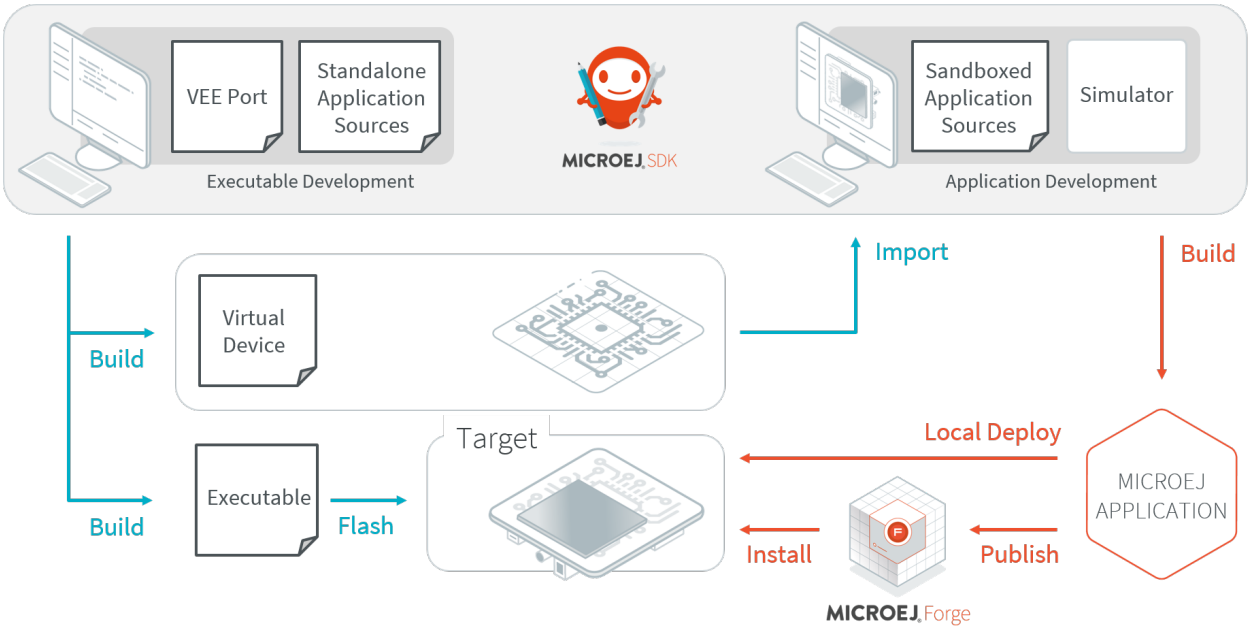


Fig. 2: SDK Workflow Overview

The following diagram outlines the SDK content. Please refer to the *SDK 6 User Guide* chapter for more details on the SDK and its usage.

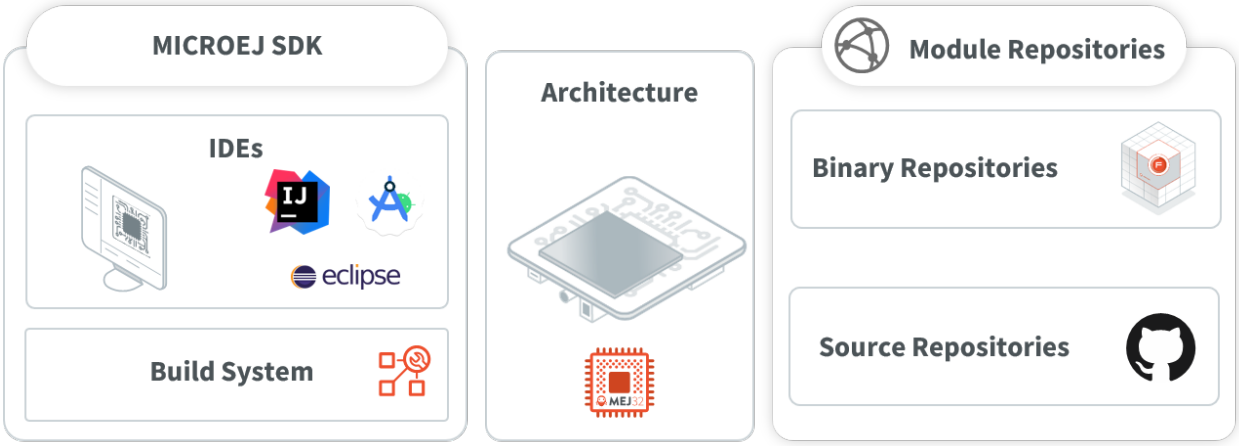


Fig. 3: SDK Ecosystem Overview

3.2.4 Build Workflow

An Executable is built from several input resources and tools. Each component has dependencies and requirements that must be carefully respected in order to build an Executable.

Resources and Dependencies Flow

Let's take a closer look at the resources MicroEJ has to offer and where to find them.

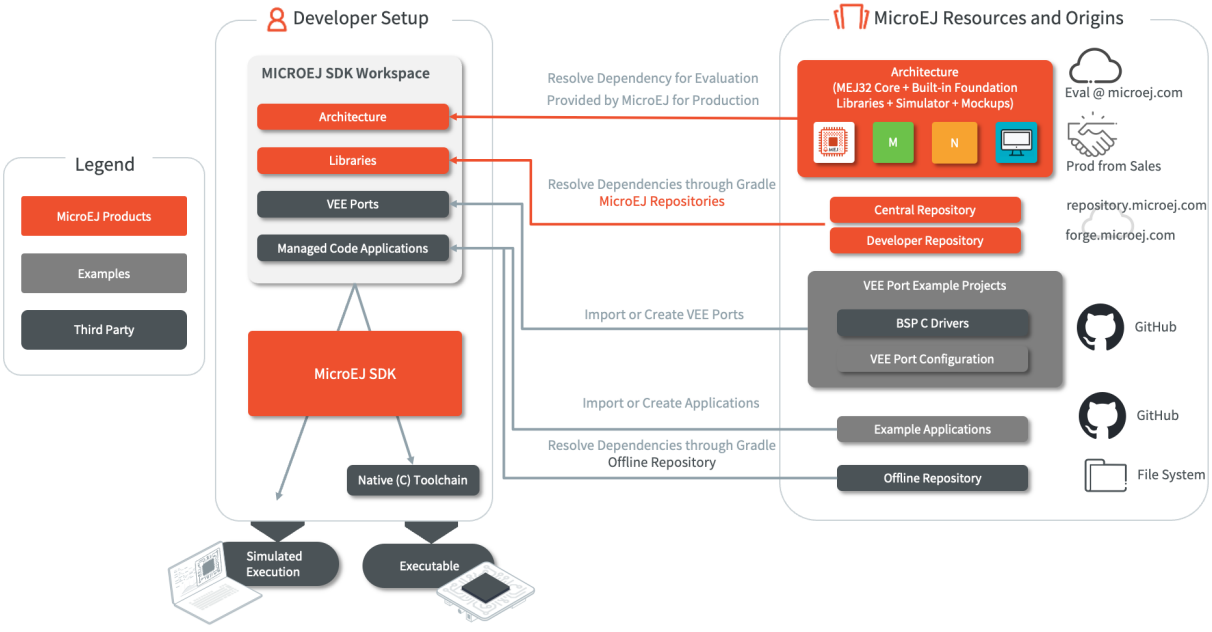


Fig. 4: MicroEJ Resources to Build Applications and their Origins

VEE Port Build Flow

To build an application, a VEE Port is required, you can see below how a VEE port is built from a MICROEJ Architecture.

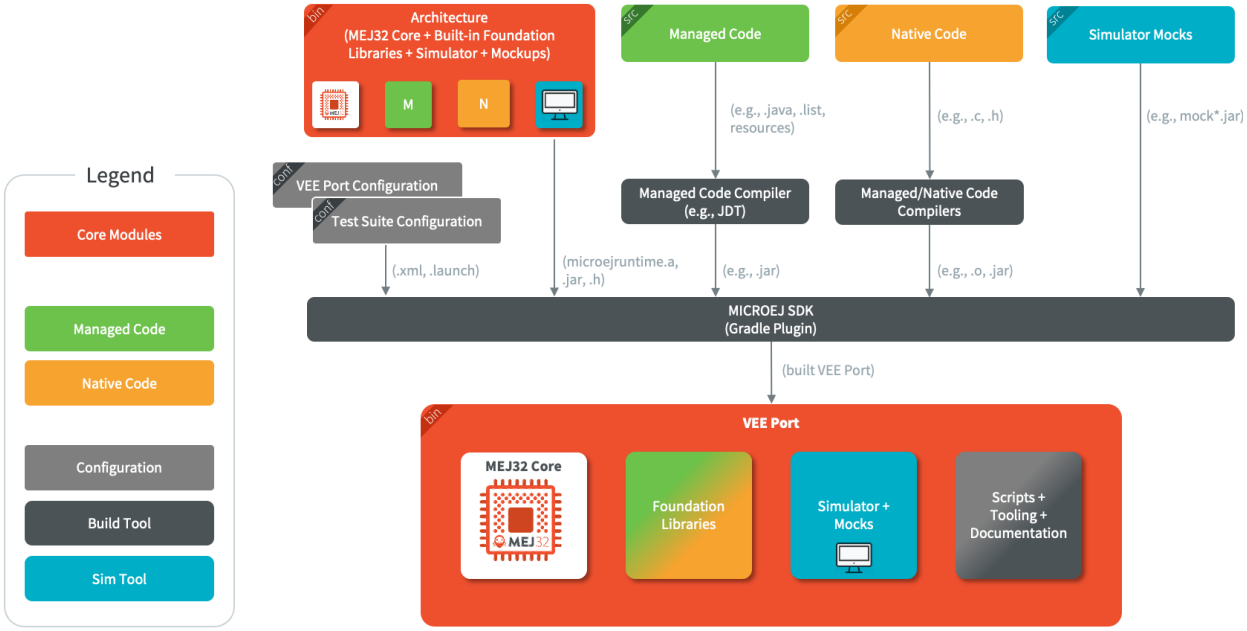


Fig. 5: VEE Port Build Flow

Executable Build Flow

Using a VEE Port here is how we build the executable for embedded devices.

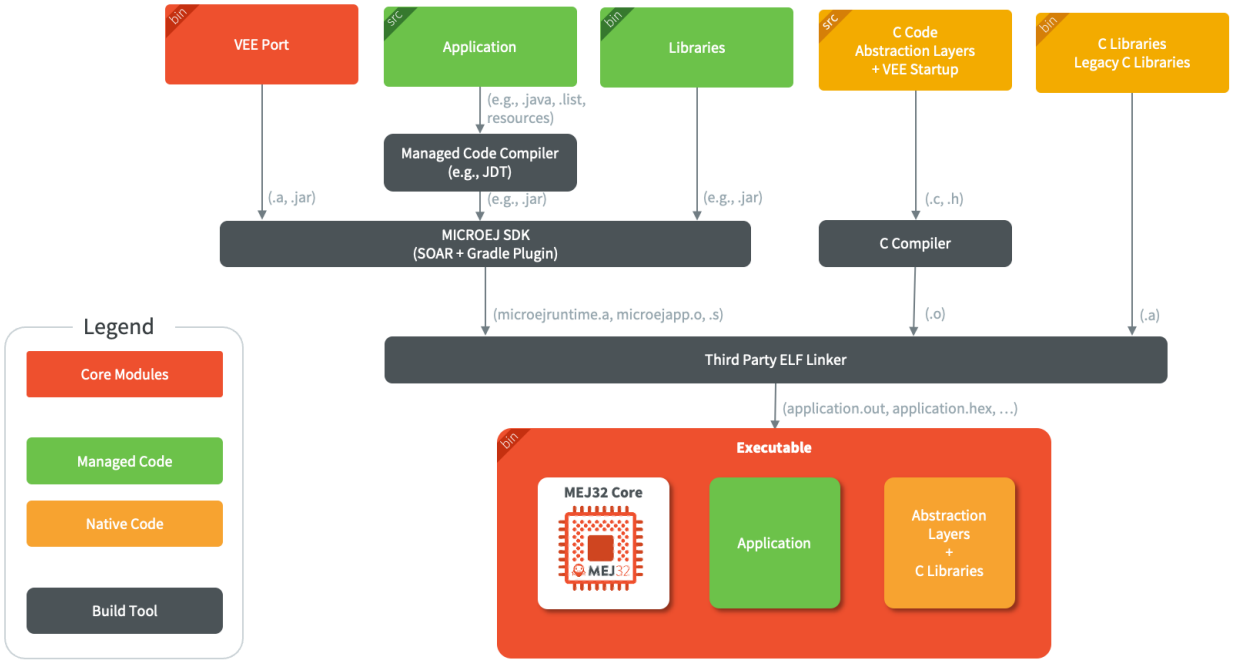
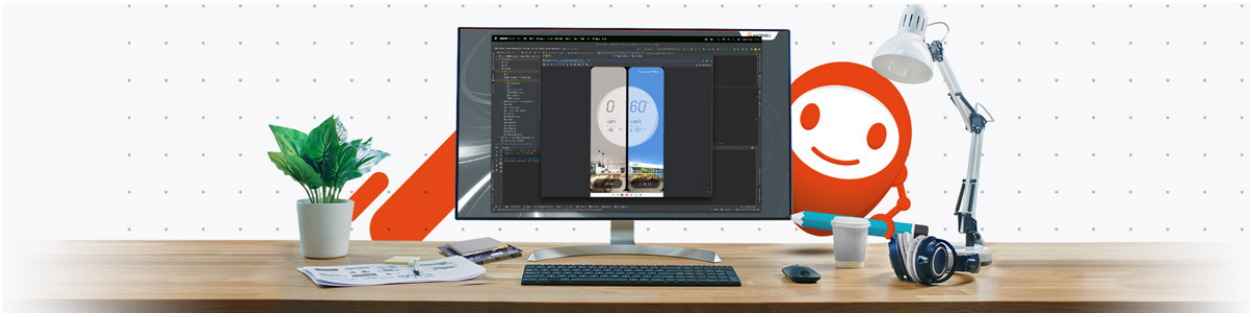


Fig. 6: Executable Build Flow

3.3 Getting Started



Welcome! Follow this step-by-step instructional guide to get started with MICROEJ SDK (Software Development Kit) and MICROEJ VEE (Virtual Execution Environment). Learn how to create apps and build a VEE Port by taking advantage of interactive tutorials and extensive technical documentation.

Before you get started, be sure to understand the fundamentals about MICROEJ VEE, and take a look at our glossary page to familiarize yourselves with MicroEJ key terminology.

What you will do:

- Choose a starter kit in the tables below
- Install MICROEJ SDK
- Watch the Get Started Tutorial
- Get Started with MICROEJ VEE

What you will learn:

- How to run an app using a Virtual Device.
- How to run an app on a real development board.
- Discover more examples and guided tutorials to go further with MicroEJ technology.

3.3.1 VEE Ports for Evaluation

These VEE ports are ideal to evaluate MicroEJ's technology. They include the latest versions of MicroEJ software components as much as possible. In case you cannot get hands on your desired starter kit, please contact us, we may help to find hardware for you.

NXP	*i.MX RT1170 Evaluation Kit*
NXP	*i.MX 93 FRDM & EVK Boards*
ST	*STM32F7508-DK Discovery Kit*

i.MX RT1170 Evaluation Kit

During this Getting Started, you will learn to:

- Run an Application on the i.MX RT1170 Evaluation Kit Virtual Device.
- Run the same Application on your i.MX RT1170 Evaluation Kit.

In case you are not familiar with MicroEJ, please visit Discover MicroEJ to understand the principles of our technology.

Prerequisites

Note: This Getting Started has been tested on Windows 10 & 11. Also note that examples used in this Getting Started could depend on older tools and libraries. Most notably our dependency manager plugin (using Gradle) could be an older version.

This Getting Started is separated in two main parts.

The first part consists of running a demo application on the Virtual Device. All you need is:

- An Internet connection to access Github repositories & *Module Repositories*.
- MICROEJ SDK 6 (installed during *Environment Setup*).

The second part consists of running the same demo application on your device. For that, you will need:

- An i.MX RT1170 EVKB Evaluation Kit, available here.
- An RK055HDMIPI4MA0 display panel, available here.
- The MCUXPresso SDK to build the BSP and flash the board. You will be guided on how to install this tool later.
- The West command line tool from the Zephyr project. You will be guided on how to install this tool later.

Environment Setup

To follow this Getting Started, you need to:

- Install MICROEJ SDK 6.
- Get the VEE Port sources from Github.

Install MICROEJ SDK 6

Install MICROEJ SDK 6 by following *Installation* instructions.

IntelliJ IDEA is used in this Getting Started, but feel free to use your favorite IDE.

Install West

West is a Zephyr tool for multiple repository management systems. It will be used to fetch the code and its dependencies.

- Install West by following Installing west instructions (tested with west 1.2.0).
- Check that the tool has been properly installed by running the command: `west --version`, it should print the west version you installed.

Get the i.MX RT1170 VEE Port project

For this Getting Started, the NXP i.MX RT1170 Evaluation Kit VEE Port will be used.

It contains the sources of the VEE Port which will be necessary to build and run applications on the simulator and on the device. The VEE Port also includes some applications samples which will be used in this Getting Started.

You can download the i.MX RT1170 VEE Port using the following commands:

```
mkdir nxpvee-mimxrt1170-prj
cd nxpvee-mimxrt1170-prj
west init -m https://github.com/MicroEJ/nxp-vee-imxrt1170-evk .
west update
```

Warning: The toolchain requires the path to the VEE Port sources to be as short as possible and contain **no whitespace** or **non-ASCII** character. It is recommended to clone the repository as close as possible to your file system root.

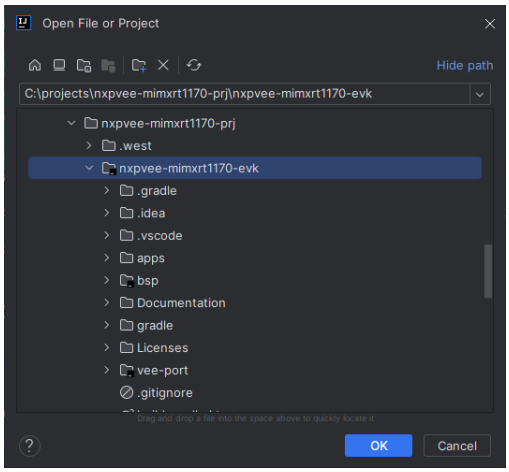
Set up the i.MX RT1170 VEE Port project on your IDE

Import the Project

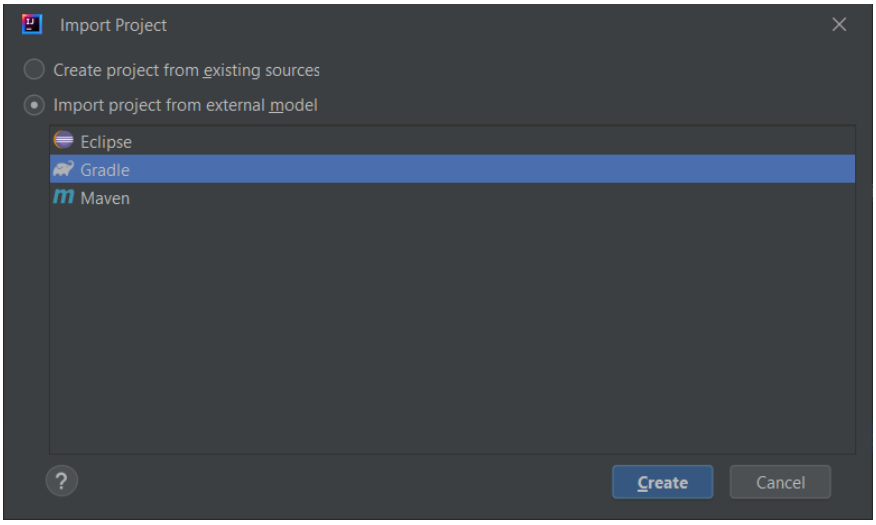
The first step is to import the i.MX RT1170 VEE Port into your IDE:

Note: If you are using another IDE than IntelliJ IDEA, please have a look at *Import a Project* section.

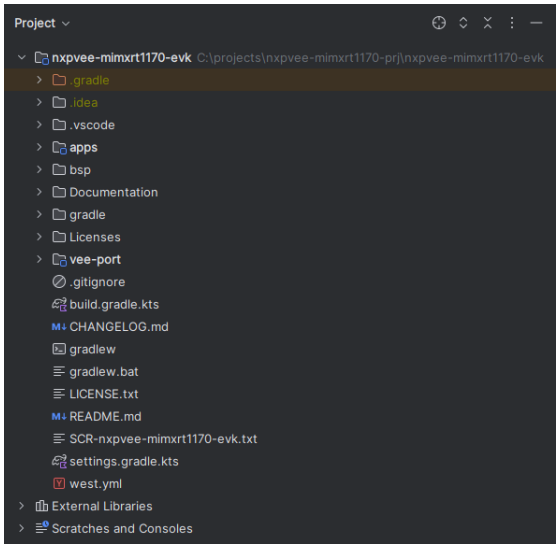
- If you are on the Welcome Screen, click on the Open button. Otherwise click either on File > Open… or on File > New > Project From Existing Sources… .
- Select the nxp-vee-imxrt1170-evk directory located where you downloaded it and click on the OK button.



- If you are asked to choose a project model, select Gradle and click on the Create button.



The Gradle project should now be imported in IntelliJ IDEA, your workspace contains the following projects:



Accept the MICROEJ SDK EULA

You may have to accept the SDK EULA if you haven't already done it, please have a look at *SDK EULA Acceptation*.

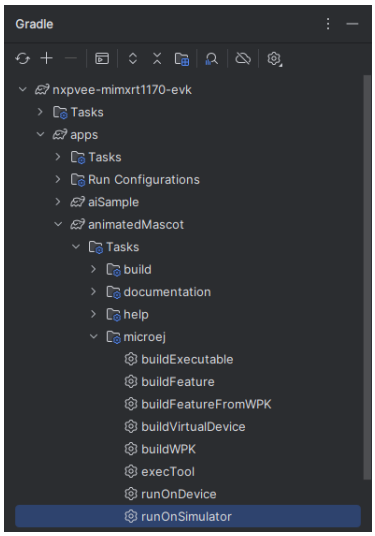
Run an Application on the Virtual Device

In this example, we will run the `animatedMascot` Application, which is available as an example sample in the i.MX RT1170 VEE Port.

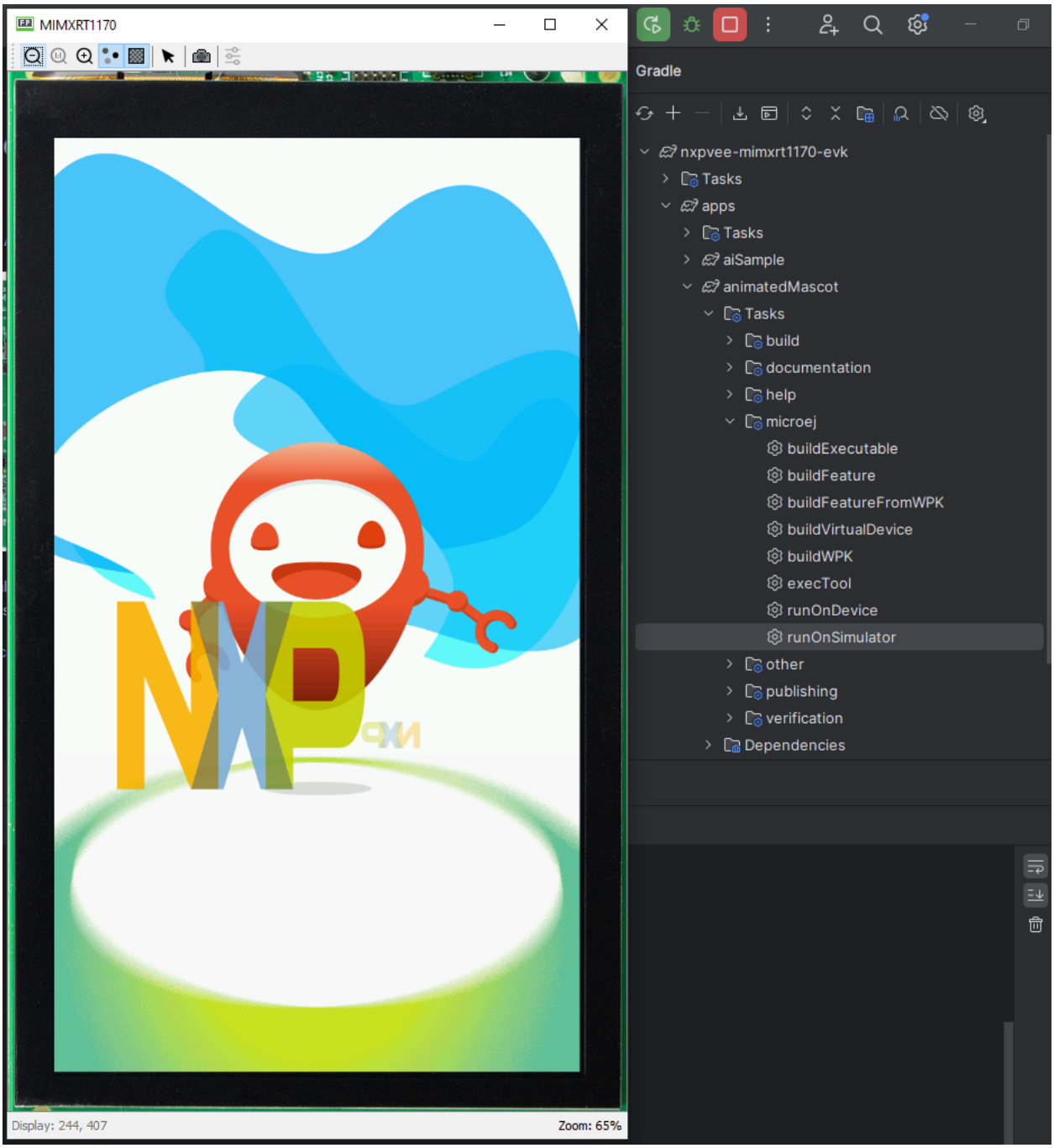
In order to execute the `animatedMascot` Application on the Virtual Device, the SDK provides the `runOnSimulator` Gradle task.

Note: If you are using another IDE than IntelliJ IDEA, please have a look at the *Run on Simulator* section.

- In the Gradle view, navigate to `nxpvee-mimxrt1170-evk > apps > animatedMascot > Tasks > microej` . Double-click on the `runOnSimulator` task.



The Virtual Device starts and executes the `animatedMascot` application.


MIMXRT1170
Display: 244, 407
Zoom: 65%
Gradle
nxpvee-mimxrt1170-evk
Tasks
apps
Tasks
aiSample
animatedMascot
Tasks
build
documentation
help
microej
buildExecutable
buildFeature
buildFeatureFromWPK
buildVirtualDevice
buildWPK
execTool
runOnDevice
runOnSimulator
other
publishing
verification
Dependencies

Well Done!

Now you know how to run an application on a Virtual Device.

If you want to learn how to run an application on your i.MX RT1170 Evaluation Kit, you can continue this Getting Started: *Run an Application on the i.MX RT1170 Evaluation Kit*.

Otherwise, learn how to *Modify the Java Application*.

Run an Application on the i.MX RT1170 Evaluation Kit

To deploy the `animatedMascot` application on your board, you will have to:

- Setup your Environment (Toolchain, flashing tool, hardware setup).
- Request a 30 days *Evaluation License* and install an activation key.
- Build the Executable and Flash it on the board.

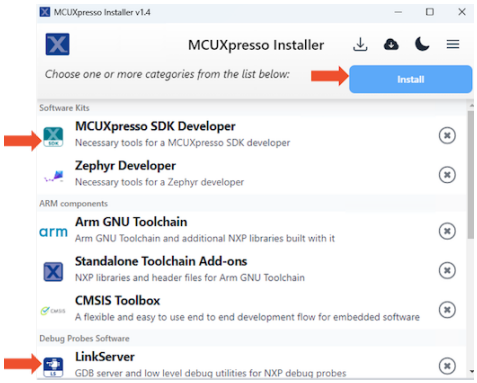
Environment Setup

This chapter takes approximately one hour and will take you through the steps to set up your board and build the BSP.

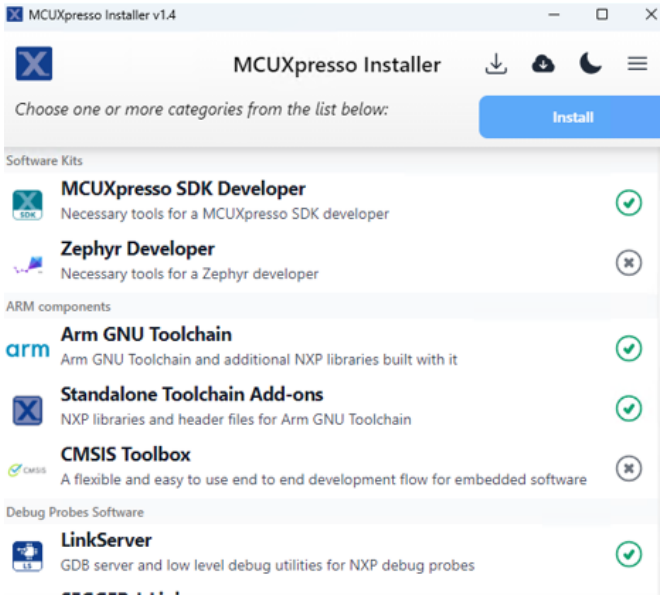
MCUXPresso SDK Setup

Install MCUXPresso SDK

- Download and install MCUXpresso Installer.
- Once installed, open it.
- Select `MCUXpresso SDK Developer` and `LinkSever` and click Install:



- Once done, a green tick appears next to the installed packages:



Add GNU ARM Embedded Toolchain Environment variable

MCUXpresso Installer installs a GNU ARM Embedded Toolchain in the `$user/.mcuxpressotools` folder.

`ARMGCC_DIR` must be set as an environment variable and point to the toolchain directory. To do so:

- Open the `Edit the system environment variables` application on Windows.
- Click on the `Environment Variables…` button.
- Click on the `New…` button under the `User variables` section.
- Set `Variable Name` to `ARMGCC_DIR`.
- Set `Variable Value` to the toolchain directory (e.g. `C:\Users\MicroEJ\.mcuxpressotools\arm-gnu-toolchain-13.2.Rel1-mingw-w64-i686-arm-none-eabi`).
- Click on the `Ok` button until it closes `Edit the system environment variables` application.

Install Make

Make is the tool that will generate the Executable based on the files generated by CMake. It will also be used to flash the board. Under the Download section, you can select the Setup program for the complete package, except sources.

By default, it will automatically add Make to your path. If not, you can manually add the GnuWin32\bin folder to your path.

Check that the tool has been properly installed:

```
C:\Users\[redacted]>make --version
GNU Make 3.81
Copyright (C) 2006  Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.

This program built for i386-pc-mingw32
```

Add the Flashing Tool Environment variable

- Open the Edit the system environment variables application on Windows.
- Click on the Environment Variables… button.
- Select Path variable under the User variables section and edit it.
- Click on New and point to the LinkServer_{version} folder located where you installed LinkServer (e.g. C:\nxp\LinkServer_1.6.133).

Hardware Setup

Set up the NXP i.MX RT1170 EVKB:

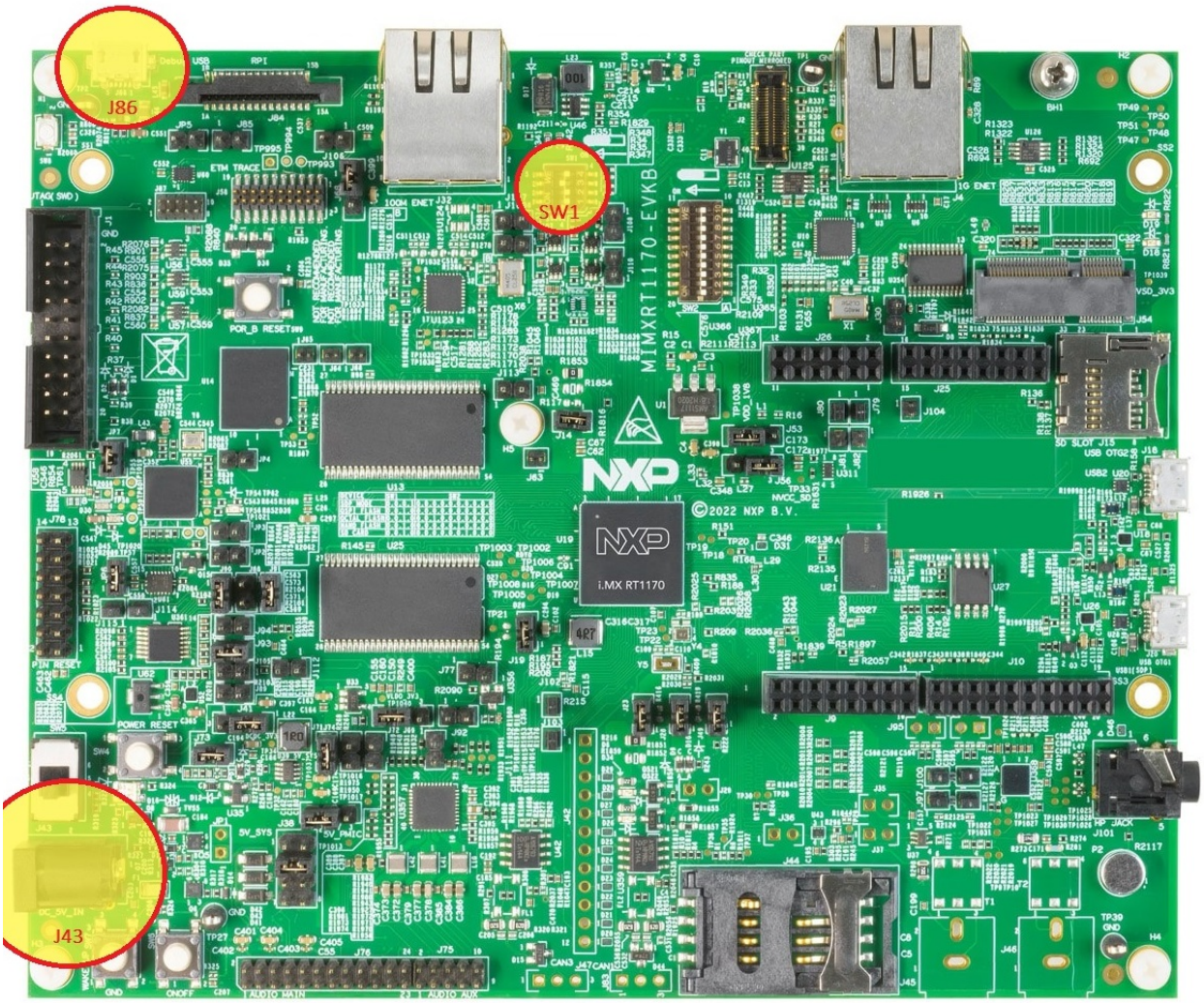


Fig. 7: NXP i.MX RT1170 EVKB Hardware Setup

- Check that the dip switches (SW1) are set to OFF, OFF, ON, and OFF.
- Connect a micro-USB cable to J86 to power the board.
- You can connect a 5V power supply to J43 if you need to use the display

The USB connection is used as a serial console for the SoC, as a CMSIS-DAP debugger, and as a power input for the board.

The VEE Port uses the virtual UART from the i.MX RT1170 EVKB USB port. A COM port is automatically mounted when the board is plugged into a computer using a USB cable. All board logs are available through this COM port.

The COM port uses the following parameters:

Baudrate	Data bits	Parity bits	Stop bits	Flow control
115200	8	None	1	None

You can have a look at your application logs with an RS232 Terminal (e.g. Termite).

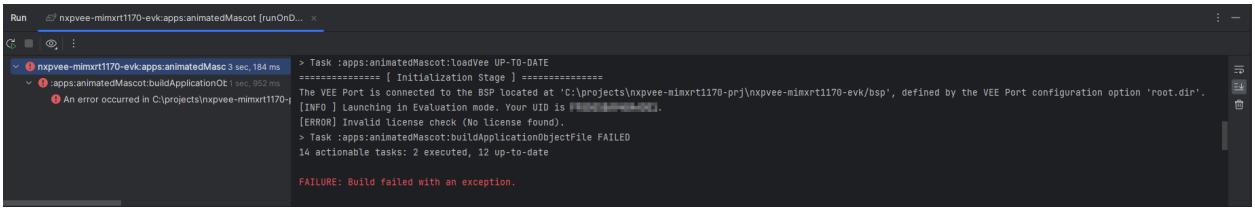
Congratulations! You have finished setting up your environment. You are now ready to discover how to build and flash a MicroEJ application.

Build the Executable for i.MX RT1170 Evaluation Kit

In order to build the Executable of the `animatedMascot` Application, the SDK provides the `buildExecutable` Gradle task.

Note: If you are using another IDE than IntelliJ IDEA, please have a look at *Build an Executable* section. Come back to this page if you need to activate an Evaluation License.

- Double-click on the `buildExecutable` task in the Gradle tasks view.
- The build stops with a failure.
- Go to the top project in the console view and scroll up to get the following error message:



- Copy the UID. It will be required to activate your Evaluation license.

Request your Evaluation License:

- Request your Evaluation license by following the *Request your Activation Key* instructions. You will be asked to fill the machine UID field with the UID you copied before.
- When you have received your activation key by email, drop it in the license directory by following the *Install the License Key* instructions (drop the license key zip file to the $\sim/.\text{microej}/\text{licenses}/$ directory).

Now your Evaluation license is installed, you can relaunch your application build by double-clicking on the `buildExecutable` task in the Gradle tasks view. It may take some time.

The Gradle task deploys the Application in the BSP and then builds the BSP using Make.

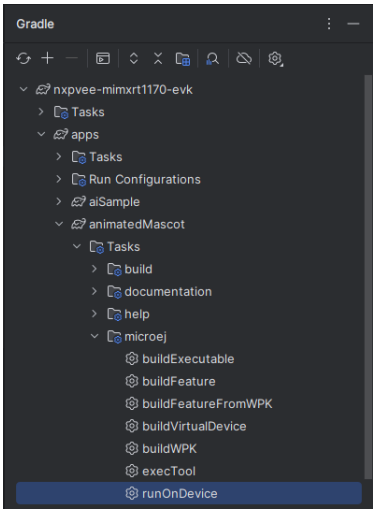
The `animatedMascot` application is built and ready to be flashed on i.MX RT1170 Evaluation Kit once the hardware setup is completed.

Flash the Application on the i.MX RT1170 Evaluation Kit

In order to flash the `animatedMascot` Application on i.MX RT1170 Evaluation Kit, the application provides the `runOnDevice` Gradle task.

Note: If you are using another IDE than IntelliJ IDEA, please have a look at *Run on Device* section.

- Double-click on the `runOnDevice` task in the Gradle tasks view. It may take some time.



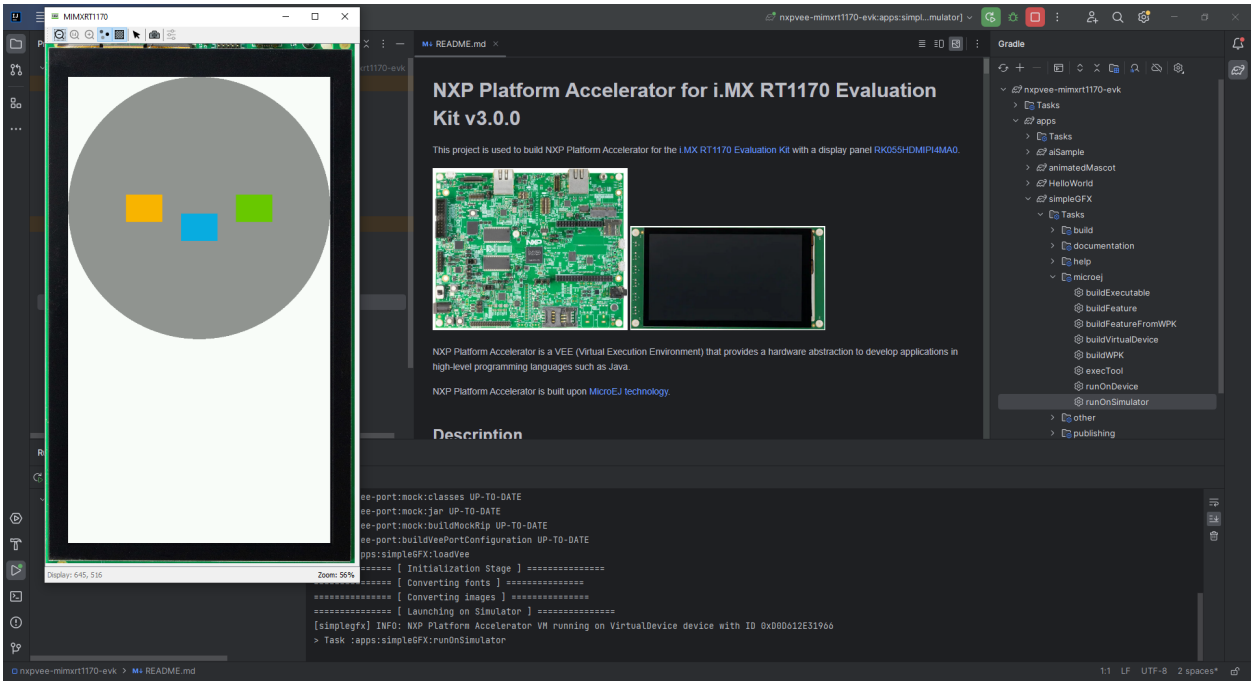
Once the firmware is flashed, you should see the `animatedMascot` running on your board.

Modify the Java Application

With MicroEJ, it is easy to modify and test your Java application on the Virtual Device.

We will modify the `simpleGFX` Application which is available as an example sample in the i.MX RT1170 VEE Port.

The `simpleGFX` Application can be executed on the Virtual Device with the he `runOnSimulator` Gradle task:



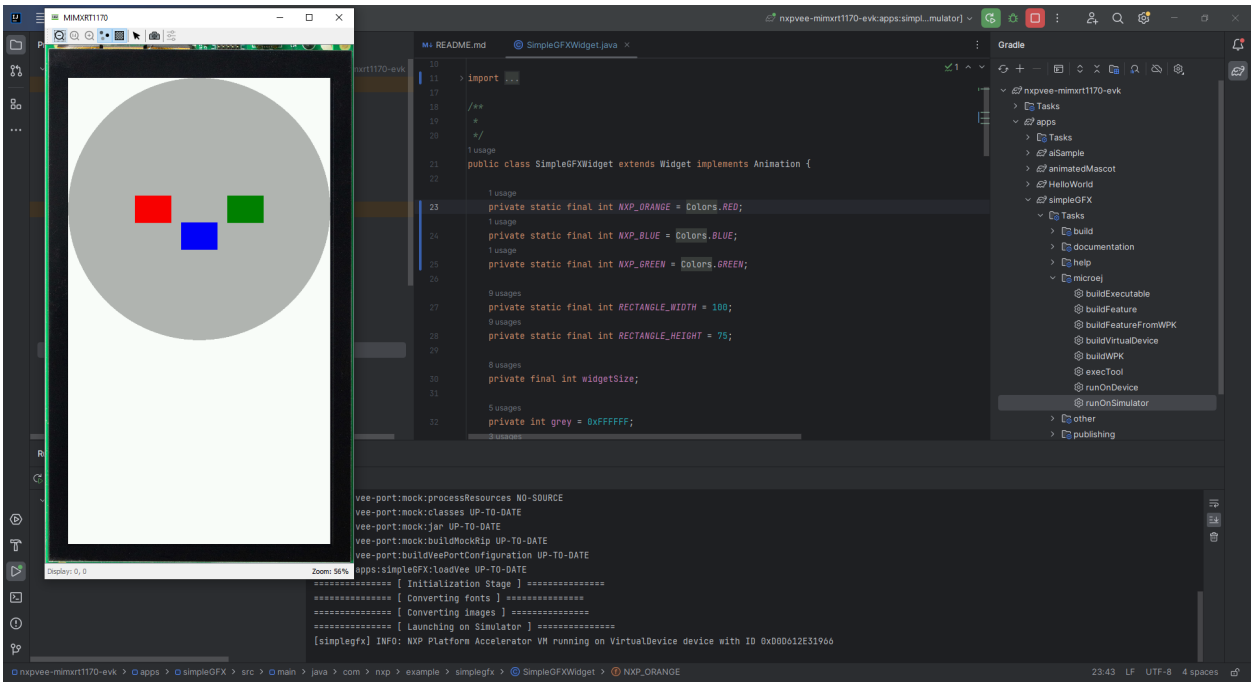
- Open `SimpleGFXWidget.java` file located in the `src/main/java/com/nxp/example/simplegfx` folder.
- The rectangles colors are set line 22, replace the following lines:

```
private static final int NXP_ORANGE = 0xF9B500;
private static final int NXP_BLUE = 0x0EAFE0;
private static final int NXP_GREEN = 0x69CA00;
```

by

```
private static final int NXP_ORANGE = Colors.RED;
private static final int NXP_BLUE = Colors.BLUE;
private static final int NXP_GREEN = Colors.GREEN;
```

- Relaunch the application on the Virtual Device:



Going Further

You have now successfully executed a MicroEJ application on an embedded device, so what's next?

If you are an application developer, you can continue to explore MicroEJ's APIs and functionalities by running and studying our samples at GitHub:

Foundation Libraries	Eclasspath	IoT
This project gathers all the basic examples of the foundation libraries.	This project gathers all the examples of eclasspath.	This project gathers simple applications using net libraries.
https://github.com/MicroEJ/Example-Foundation-Libraries	https://github.com/MicroEJ/Example-Eclasspath	https://github.com/MicroEJ/Example-IOT

You can also learn how to build bigger and better applications by reading our *Application Developer Guide*.

If you are an embedded engineer, you could look at our VEE port examples at GitHub. And to learn how to create custom VEE Ports, you can read our *VEE Porting Guide*.

You can also follow the *Kernel Developer Guide* for more information on our multi-application framework or read about our powerful wearable solution called *VEE Wear*.

Last but not least, you can choose to learn about specific topics by following one of our many *Training Courses* ranging from how to easily debug applications to setting up a Continuous Integration process and a lot of things in between.

i.MX 93 FRDM & EVK Boards

During this Getting Started, you will learn to:

- Run an Application on the i.MX93 Virtual Device.
- Run the same Application on your FRDM i.MX93 Development Board or i.MX93 Evaluation Kit.

In case you are not familiar with MicroEJ, please visit Discover MicroEJ to understand the principles of our technology.

Prerequisites

Note: This Getting Started has been tested on Windows 10 & 11 with a WSL distribution Ubuntu 22.04. Also note that the examples used in this Getting Started guide may depend on older tools and libraries. Notably, our dependency manager plugin (using Gradle) may be an older version.

This Getting Started is separated into two main parts.

The first part consists of running a demo application on the Virtual Device. All you need is:

- An Internet connection to access GitHub repositories & *Module Repositories*.
- MICROEJ SDK 6 (installed during *Environment Setup*).

The second part consists of running the same demo application on your device. For that, you will need:

- One of these two Development Boards:
  - FRDM i.MX93 Development Board, available here.
  - i.MX93 Evaluation Kit, available here.
    - With an IMX-MIPI-HDMI adapter to connect the HDMI display.
- An HDMI display with a touchscreen.
  - This getting started has been tested with a 1280x720 HDMI display: MageDok T080A.
- A prebuilt Yocto Linux image, with all necessary Linux packages preinstalled.
- A Yocto SDK, to cross-compile a sample application.

Environment Setup

To follow this Getting Started, you need to:

- Install MICROEJ SDK 6.
- Get the Example-Java-Widget from github.

Install MICROEJ SDK 6

Install MICROEJ SDK 6 by following the *Installation* instructions.

IntelliJ IDEA is used in this Getting Started, but feel free to use your favorite IDE.

Get Example-Java-Widget

For this Getting Started, the `Example-Java-Widget` Application will be used. You can download it using the following command:

```
git clone -b 8.1.1 https://github.com/MicroEJ/Example-Java-Widget.git
```

Note: If you don't have Git installed, you can download the source code directly from our GitHub repository. Then you can click on: `Code > Download ZIP` .

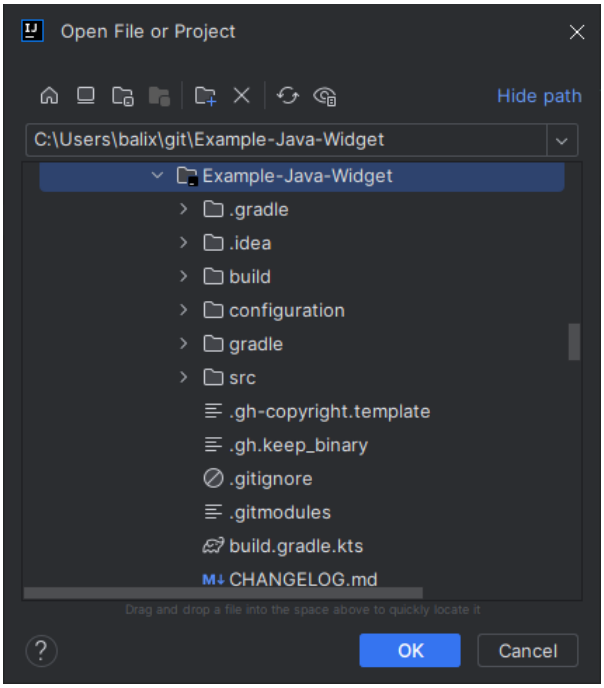
Set up the Application on your IDE

Import the Project

The first step is to import the `Example-Java-Widget` Application into your IDE:

Note: If you are using another IDE than IntelliJ IDEA, please have a look at *Import a Project* section.

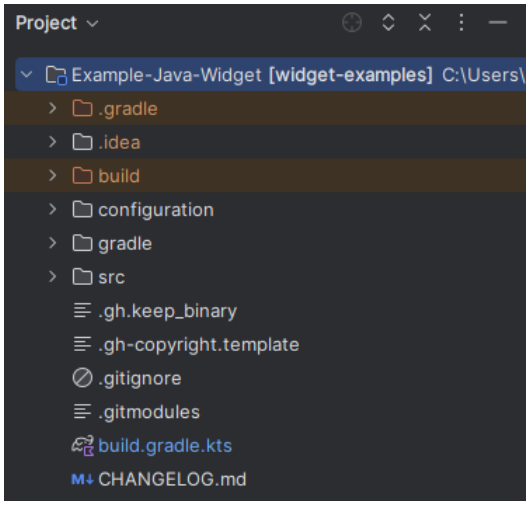
- If you are in the Welcome Screen, click on the `Open` button. Otherwise, click either on `File` > `Open…` or on `File` > `New` > `Project From Existing Sources…` .
- Select the `Example-Java-Widget` directory located where you downloaded it and click on the `OK` button.



- If you are asked to choose a project model, select `Gradle` .

- Click on the `Create` button.

The Gradle project should now be imported into IntelliJ IDEA. Your workspace contains the following projects:



Select the VEE Port

For this getting started, we will use the i.MX93 VEE Port demo module that is available on the *Developer Repository*.

In the Gradle build file `build.gradle.kts`, replace the VEE dependency `microejVEE` with the i.MX93, like this:

```
dependencies {
  microejVee("com.microej.veeport.imx93:vee-port:3.1.0")
}
```

For more information about how to select a VEE Port please refer to the following section: *Select a VEE Port*.

Accept the MICROEJ SDK EULA

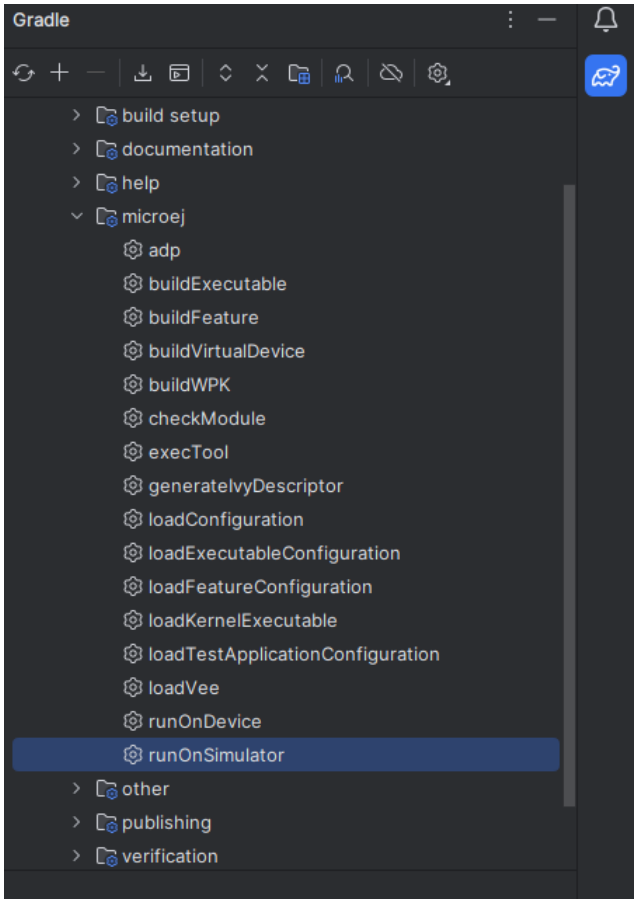
You may have to accept the SDK EULA if you haven't already done so, please have a look at *SDK EULA Acceptation*.

Run an Application on the Virtual Device

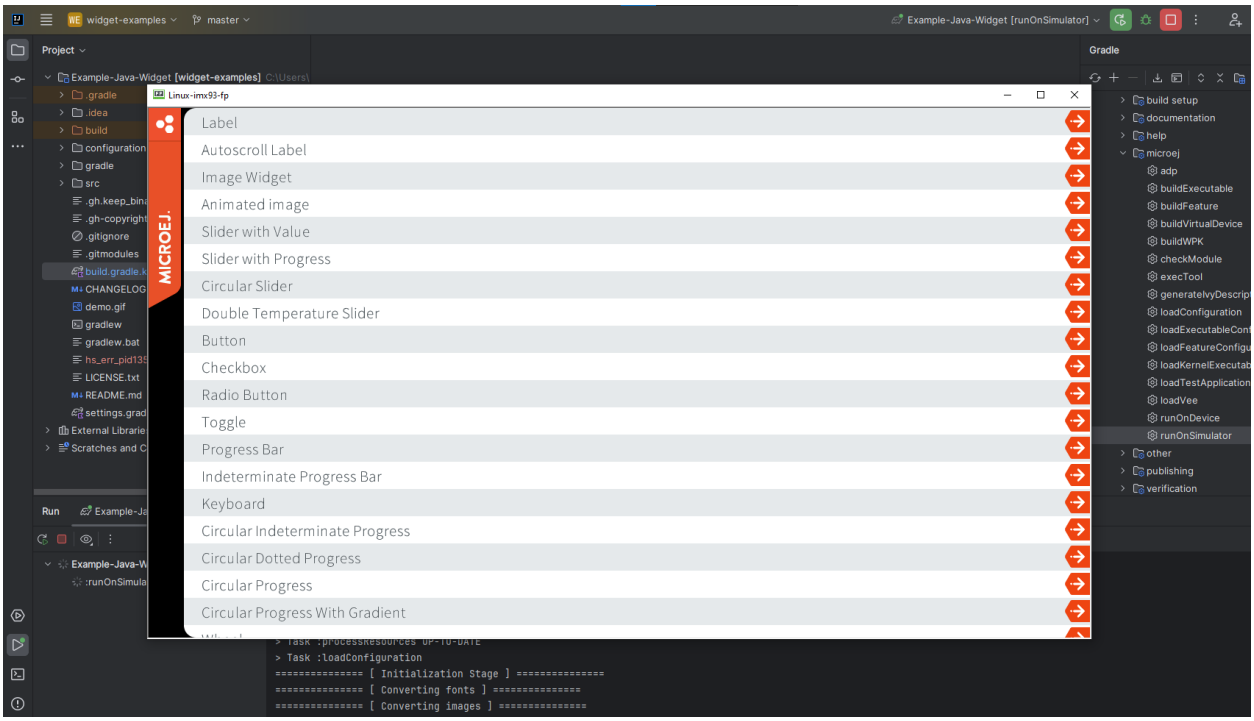
To execute the `Example-Java-Widget` Application on the Virtual Device, the SDK provides the Gradle `runOnSimulator` task.

Note: If you are using another IDE than IntelliJ IDEA, please have a look at the *Run on Simulator* section.

- Double-click on the `runOnSimulator` task in the Gradle tasks view. It may take a few seconds.



The Virtual Device starts and executes the `Example-Java-Widget` application.



Well done!

Now you know how to run an application on a Virtual Device.

If you want to learn how to run an application on your i.MX93, you can continue this Getting Started: *Run an Application on i.MX93*.

Otherwise, learn how to *Modify the Java Application*.

Run an Application on i.MX93

To deploy `Example-Java-Widget` application on your board, you will have to:

- Set up your environment (toolchain, hardware setup).
- Request a 30 days *Evaluation License* and install an activation key.
- Build the Executable.
- Run the Executable on board.

Environment Setup

This chapter takes approximately one hour and will take you through the steps to build a VEE Executable and set up the device.

Install the Yocto SDK

The Executable is built using a Yocto SDK. It contains the following:

- The C toolchain.
- The cross-compiled libraries.
- All the necessary headers (libc, but also the headers of the kernel and the libraries installed in the firmware rootfs).
- An environment setup script (to set $CC, $LD, $SDKSYSROOT variables).

To install the Yocto SDK, use the following commands in WSL or Linux:

```
$ curl -O https://repository.microej.com/packages/yocto/i.MX93EVK/2024-04-30-IMX93-oecore-x86_64-
↪armv7at2hf-neon-vfpv4-toolchain-nodistro-1.0.0.sh
$ chmod +x 2024-04-30-IMX93-oecore-x86_64-armv7at2hf-neon-vfpv4-toolchain-nodistro-1.0.0.sh
$ ./2024-04-30-IMX93-oecore-x86_64-armv7at2hf-neon-vfpv4-toolchain-nodistro-1.0.0.sh
MicroEJ: 32-bit userspace + 64-bit kernel SDK installer version nodistro.0
==========================================================================================
Enter target directory for SDK (default: /usr/local/oecore-x86_64):
You are about to install the SDK to "/usr/local/oecore-x86_64". Proceed [Y/n]? Y
[sudo] password for xxxxx:
Extracting SDK.....................................................................done
Setting it up...done
SDK has been successfully set up and is ready to be used.
Each time you wish to use the SDK in a new shell session, you need to source the environment setup script e.g.
$ . /usr/local/oecore-x86_64/environment-setup-armv7at2hf-neon-vfpv4-oemllib32-linux-gnueabi
$ . /usr/local/oecore-x86_64/environment-setup-cortexa55-oe-linux
```

The installation path can then be used to build the VEE Executable.

Install Required Packages

Some additionnal packages will be required in order to build an executable. Run the following command (either in Linux or WSL):

```
sudo apt-get update && sudo apt-get install dos2unix make
```

Flash the image on an SD card

The Linux images are available here:

- Yocto WIC Image for FRDM iMX93
- Yocto WIC Image for iMX93 EVK

For this getting started we use a Linux image flashed on an SD card.

Linux

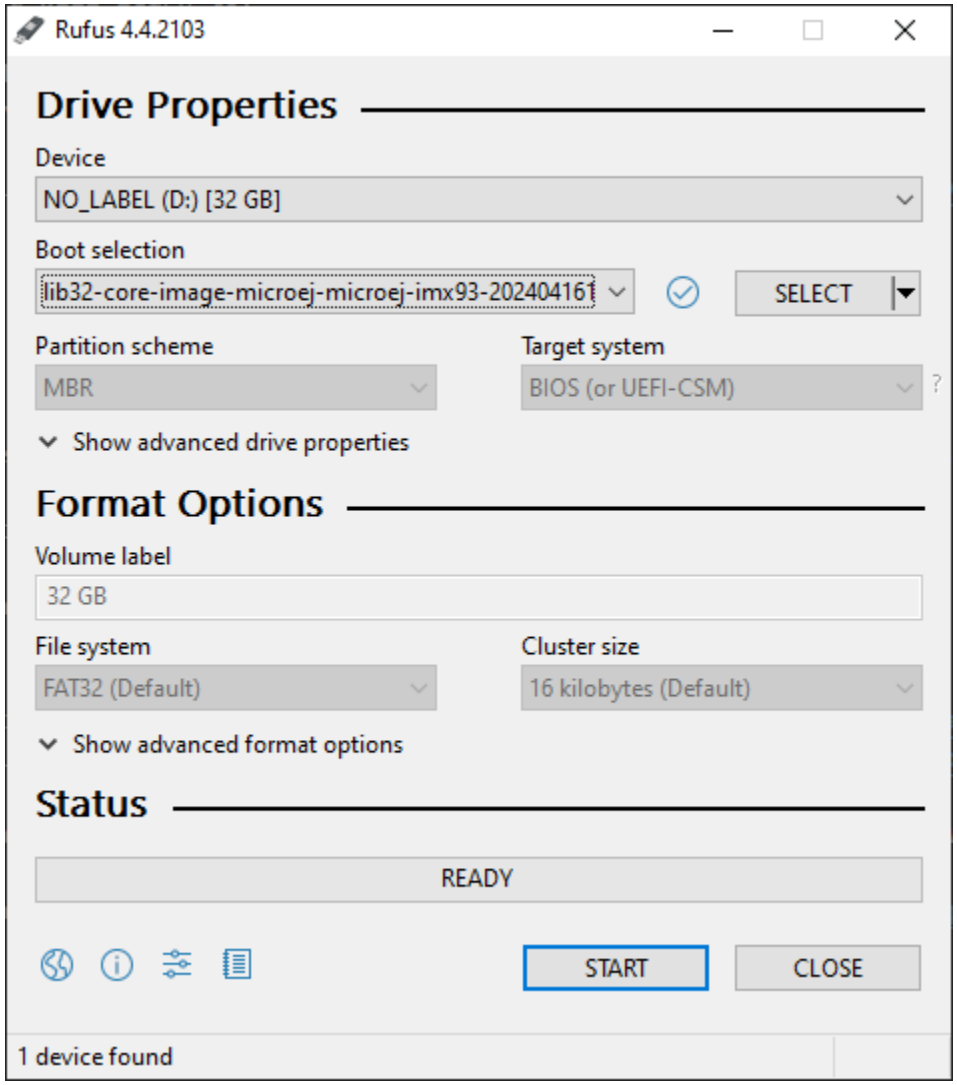
To flash the image on Linux, use the following command (assuming the SDCard device is /dev/mmcblk0):

```
bmaptool copy lib32-core-image-microej-microej-imx93.wic.gz /dev/mmcblk0
```

Windows

To flash the image on Windows, do the following:

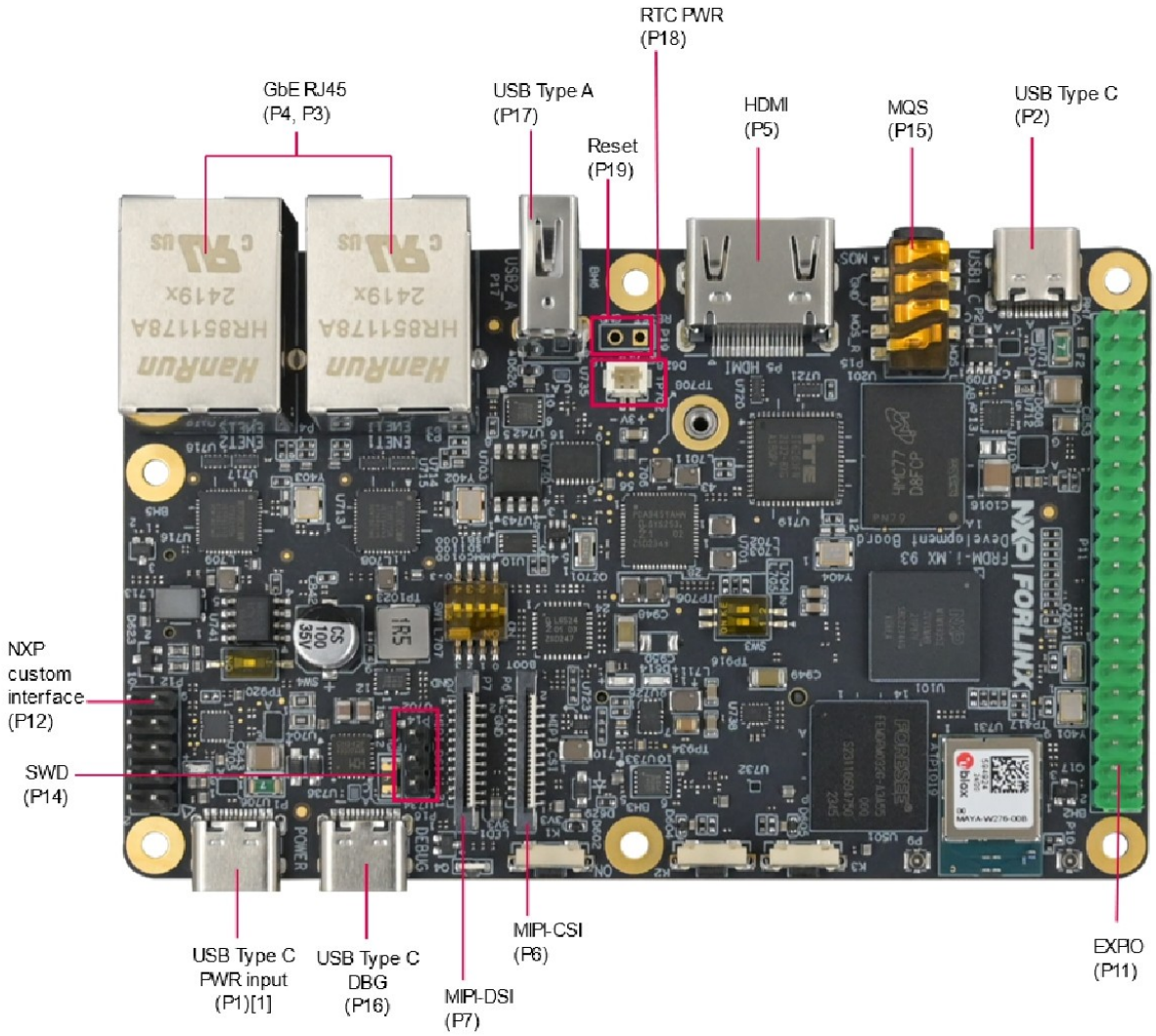
- Install Rufus: https://rufus.ie/en/.
- Extract the `.wic` file from the archive `lib32-core-image-microej-microej-imx93.wic.gz` .
- Select your SD card device in the list at the top (see example of configuration below).
- Select your `.wic` file by clicking on `SELECT`
- Finally click on `START` at the bottom and wait for the process to finish.



Hardware Setup

FRDM i.MX93

i.MX93 EVK



[1] - USB Type C PWR input (P1) shown in the figure is the only power supply port, and must always be supplied for system runnir

To setup the hardware you will need to connect the following on the Development Board:

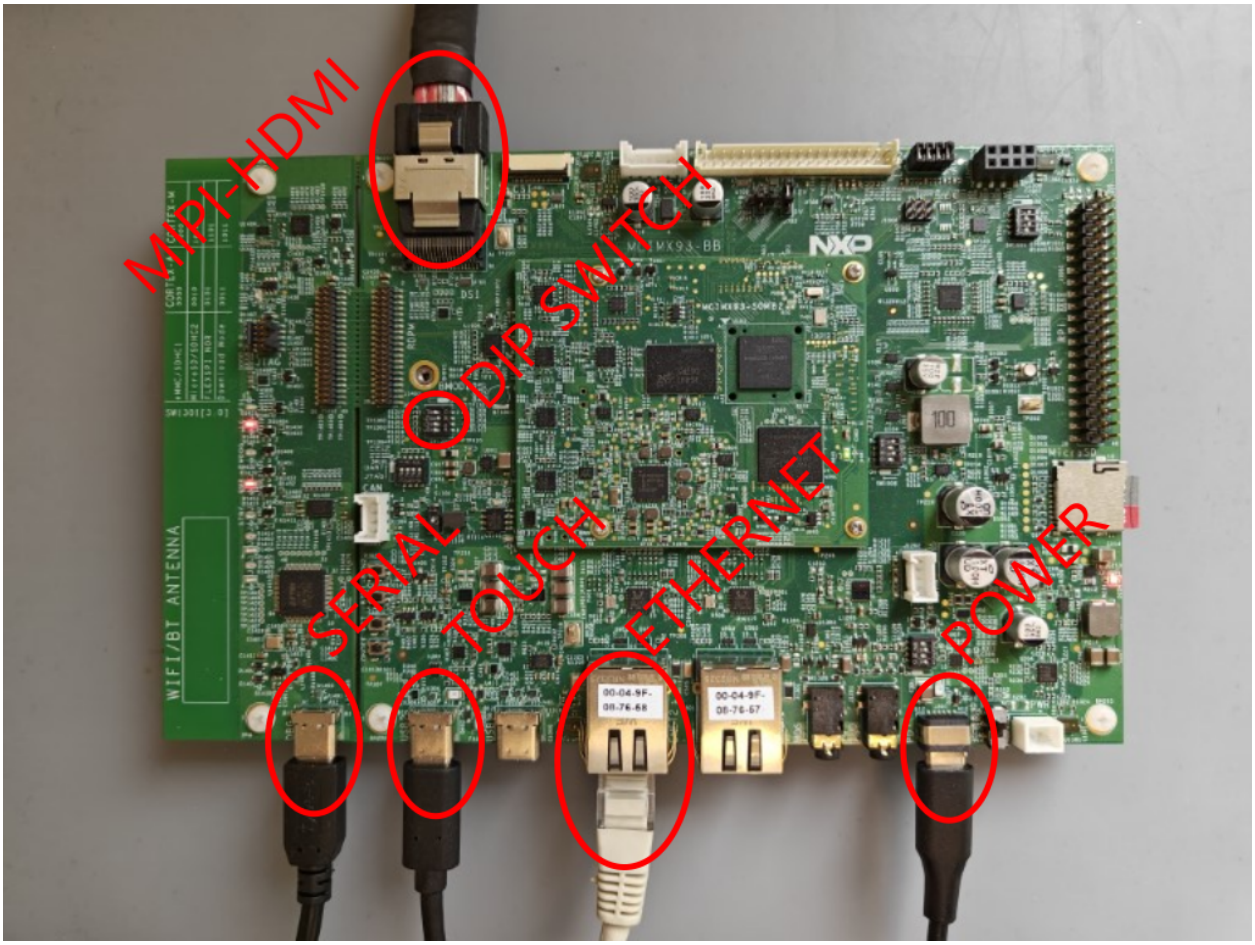
- A USB C cable to the PWR input (P1)
- A USB C cable to the DBG port (P16)
- A USB C, or USB A cable for the touchscreen device (P2 or P17).
- A RJ45 cable to access the network.
- An HDMI cable.

By default, the i.MX93 evaluation kit will boot from the eMMC. To change the boot mode to micro SD, set the DIP switch SW1 [3-0] to 0011 .

The serial port is used to connect to a shell, it uses the following parameters:

Baudrate	Data bits	Parity bits	Stop bits	Flow control
115200	8	None	1	XON/XOFF

To connect to the shell enter the login `root`.



To setup the hardware you will need to connect the following on the EVK:

- A USB C cable for the power (provided with the EVK).
- A USB C cable for the serial port.
- A USB C cable for the touchscreen device.
- A RJ45 cable to access the network.
- An HDMI cable connected to the IMX-HDMI-MIPI adapter.

By default, the FRDM i.MX93 evaluation kit will boot from the eMMC. To change the boot mode to micro SD, set the DIP switch `BMODE` to `0100`, i.e. SW1301-2 set to `1`.

The serial port is used to connect to a shell, it uses the following parameters:

Baudrate	Data bits	Parity bits	Stop bits	Flow control
115200	8	None	1	XON/XOFF

To connect to the shell enter the login `root`.

Touchscreen calibration

The touch screen needs to be calibrated before using the application.

This can be done using the *ts_calibrate* tool provided by tslib on Linux.

Open a console on the target (SSH or serial), start the calibration tool and follow the instructions on your device screen:

```
export TSLIB_TSDEVICE=/dev/input/touchscreen0
export TSLIB_CONFFILE=/etc/ts.conf
export TSLIB_CALIBFILE=/etc/ts.calib
ts_calibrate
```

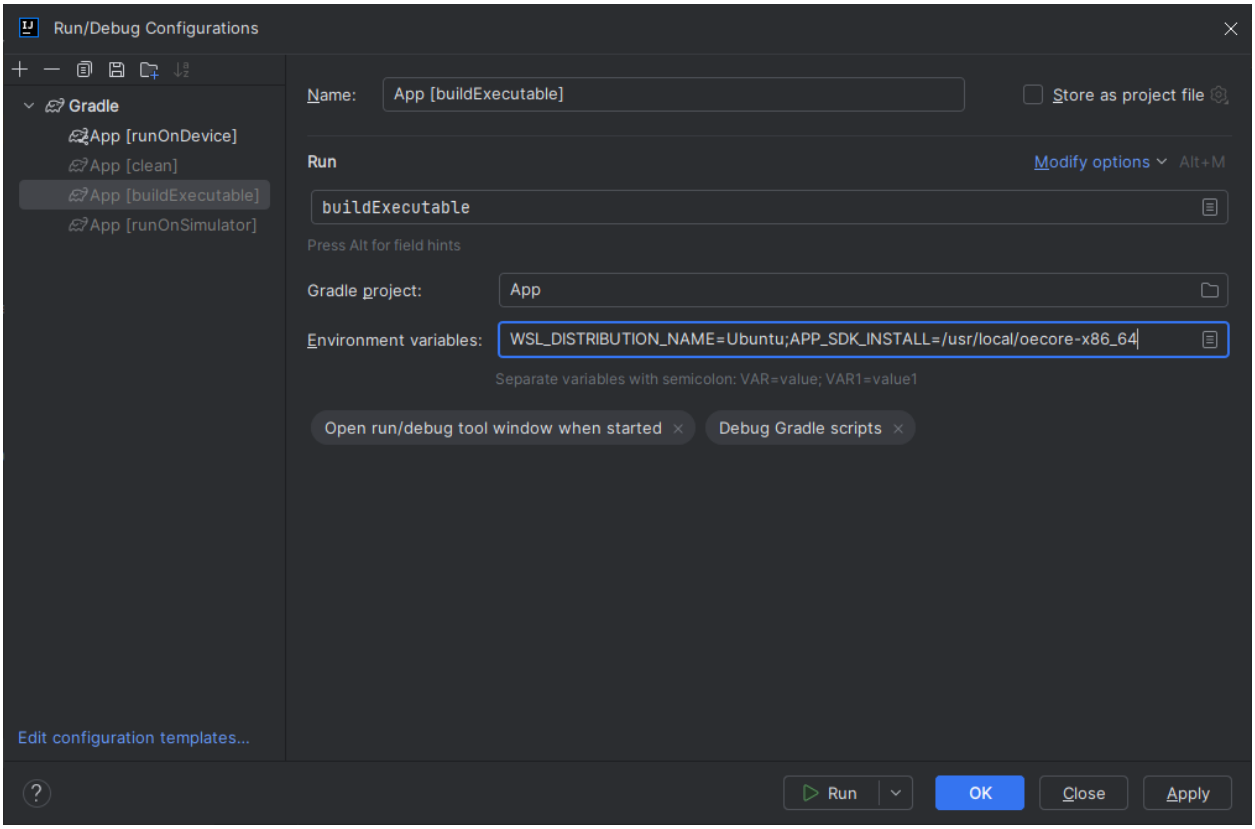
Congratulations! You have finished the setup of your environment. You are now ready to discover how to build and run a MicroEJ application.

Build the Executable for i.MX93

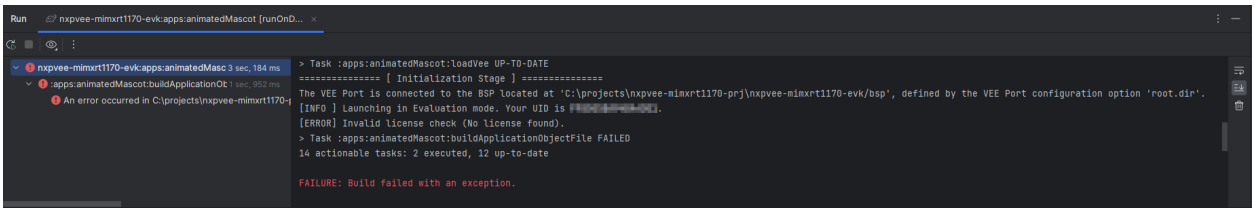
In order to build the Executable of the `Example-Java-Widget` Application, the SDK provides the `buildExecutable` Gradle task.

Note: If you are using another IDE than IntelliJ IDEA, please have a look at *Build an Executable* section. Come back to this page if you need to activate an Evaluation License.

- Before using this task, you will have to configure some environment variables that depend on the OS you are using.
  - To do so, right-click on `buildExecutable` then `Modify Run Configuration...` .
  - Fill in your variables in `Environment variables` , each one separated by a semicolon (see screenshot):
    - If you changed the default Yocto SDK installation path, set the following environment variable
      - `APP_SDK_INSTALL` : Path to the Yocto SDK (by default `/usr/local/oecore-x86_64/`)
    - When using Windows with WSL, set the WSL distribution name, which you can get with the command `wsl –list` in Windows terminal
      - `WSL_DISTRIBUTION_NAME` : Name of the WSL distribution if using Windows (by default `Ubuntu`).



- Double-click on the `buildExecutable` task in the Gradle tasks view.
- The build stops with a failure.
- Go to the top project in the console view and scroll up to get the following error message:



- Copy the UID. It will be required to activate your Evaluation license.

Request your Evaluation License:

- Request your Evaluation license by following the *Request your Activation Key* instructions. You will be asked to fill the machine UID field with the UID you copied before.
- When you have received your activation key by email, drop it in the license directory by following the *Install the License Key* instructions (drop the license key zip file to the `~/.microej/licenses/` directory).

Now your Evaluation license is installed, you can relaunch your application build by double-clicking on the `buildExecutable` task in the Gradle tasks view. It may take some time.

The Gradle task deploys the Application in the BSP and then builds the BSP using Make.

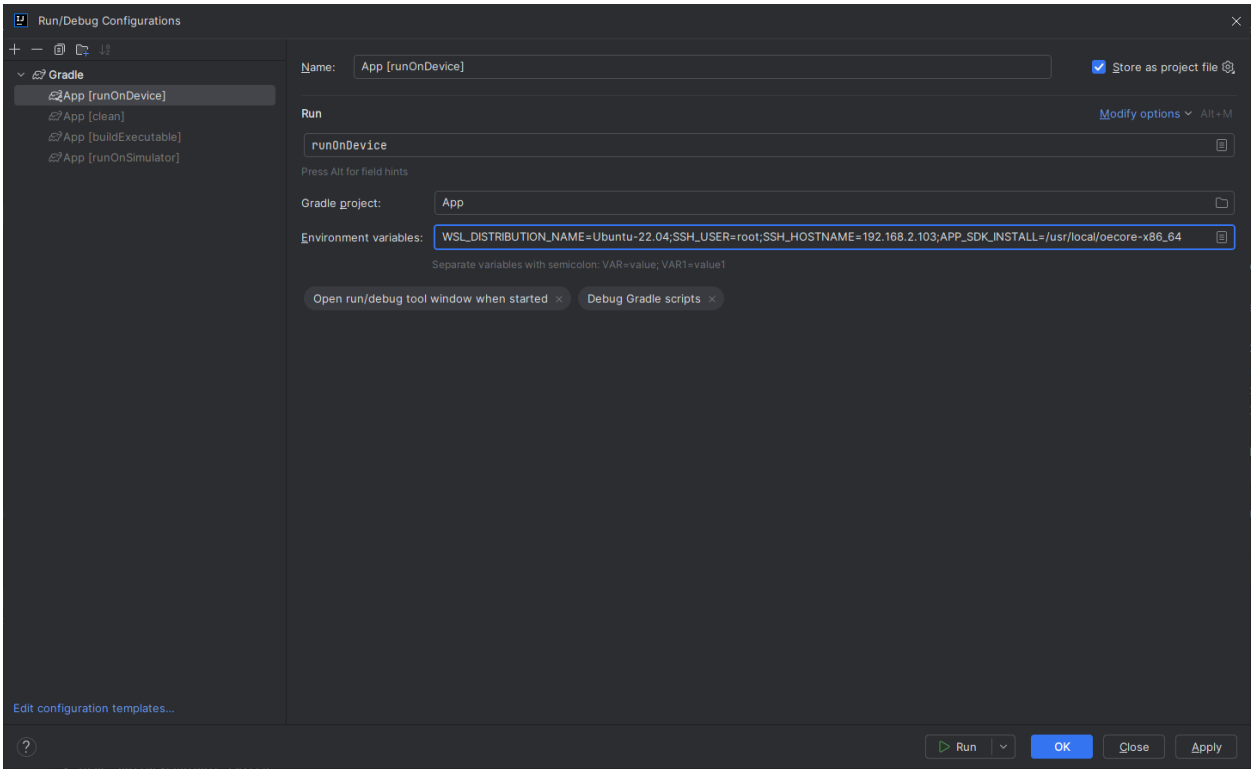
The `Example-Java-Widget` application is built and ready to be flashed on i.MX93 Evaluation Kit once the hardware setup is completed.

Run the Application on the i.MX93

To run the `Example-Java-Widget` Application on i.MX93, the application provides the Gradle `runOnDevice` task.

Note: If you are using another IDE than IntelliJ IDEA, please have a look at the *Run on Device* section.

- Before using this task, you will have to configure the device IP Address in the environment variables.
  - To do so, right-click on `runOnDevice` then `Modify Run Configuration...`.
  - Fill in your variables in `Environment variables`, each one separated by a semicolon:
    - `SSH_USER=root`
    - `SSH_HOSTNAME` : IP address of the i.MX93.
    - `APP_SDK_INSTALL` : Path to the Yocto SDK (by default `/usr/local/oecore-x86_64/`).
    - `WSL_DISTRIBUTION_NAME` : Name of the WSL distribution if using Windows (by default `Ubuntu`).
- Double-click on the `runOnDevice` task in the Gradle tasks view. It may take some time.



Once the application is running, you should see the `Example-Java-Widget` on your board.

Modify the Java Application

With MicroEJ, it is easy to modify and test your Java application on the Virtual Device.

For example, we can modify the color of the items in the main page list.

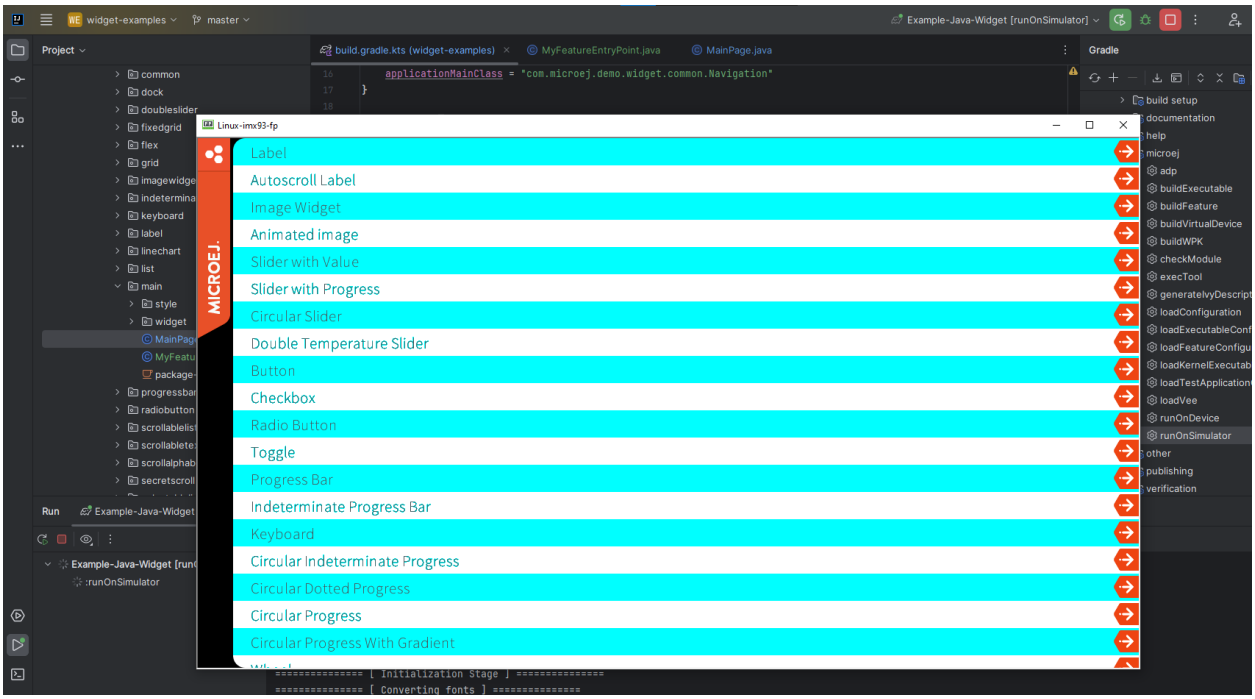
- Open `MainPage.java` located in the `src/main/java/com/microej/demo/widget/main/MainPage.java` folder
- Odd items background color is set line 74, replace the following line:

```
style.setBackground(new GoToBackground(GRAY));
```

with

```
style.setBackground(new GoToBackground(Colors.CYAN));
```

- Follow *Run an Application on the Virtual Device* instructions to launch the modified application on the Virtual Device.



Going Further

You have now successfully executed a MicroEJ application on an embedded device, so what's next?

If you are an application developer, you can continue to explore MicroEJ's APIs and functionalities by running and studying our samples at GitHub:

Foundation Libraries	Eclasspath	IoT
This project gathers all the basic examples of the foundation libraries.	This project gathers all the examples of eclasspath.	This project gathers simple applications using net libraries.
https://github.com/MicroEJ/Example-Foundation-Libraries	https://github.com/MicroEJ/Example-Eclasspath	https://github.com/MicroEJ/Example-IOT

You can also learn how to build bigger and better applications by reading our *Application Developer Guide*.

If you are an embedded engineer, you could look at our VEE port examples at GitHub. And to learn how to create custom VEE Ports, you can read our *VEE Porting Guide*.

You can also follow the *Kernel Developer Guide* for more information on our multi-application framework or read about our powerful wearable solution called *VEE Wear*.

Last but not least, you can choose to learn about specific topics by following one of our many *Training Courses* ranging from how to easily debug applications to setting up a Continuous Integration process and a lot of things in between.

STM32F7508-DK Evaluation Kit

During this Getting Started, you will learn to:

- run an Application on the STM32F7508-DK Evaluation Kit Virtual Device,
- run the same Application on your STM32F7508-DK Evaluation Kit.

In case you are not familiar with MicroEJ, please visit Discover MicroEJ to understand the principles of our technology.

Prerequisites

Note: This Getting Started has been tested on Windows 10. Also note that examples used in this Getting Started could depend on older tools and libraries. Most notably our dependency manager plugin (using Gradle) could be an older version.

This Getting Started is separated in two main parts.

The first part consists of running an MVC demo application on the Virtual Device. All you need is:

- An Internet connection to access Github repositories & *Module Repositories*.
- MICROEJ SDK 6 (installed during *Environment Setup*).
- The Example-Foundation-Libraries samples at GitHub. Download or clone the project here.

The second part consists of running the same demo application on your device. For that you will need:

- STM32F7508-DK Evaluation Kit, available here.
- You will be guided on how to install STM32CubeIDE later.

Environment Setup

To follow this Getting Started, you need to:

- Install MICROEJ SDK 6.
- Get the Example-Foundation-Libraries from GitHub.

Install MICROEJ SDK 6

Install MICROEJ SDK 6 by following *Installation* instructions. Android Studio Hedgehog is used on this Getting Started but feel free to use your favorite IDE.

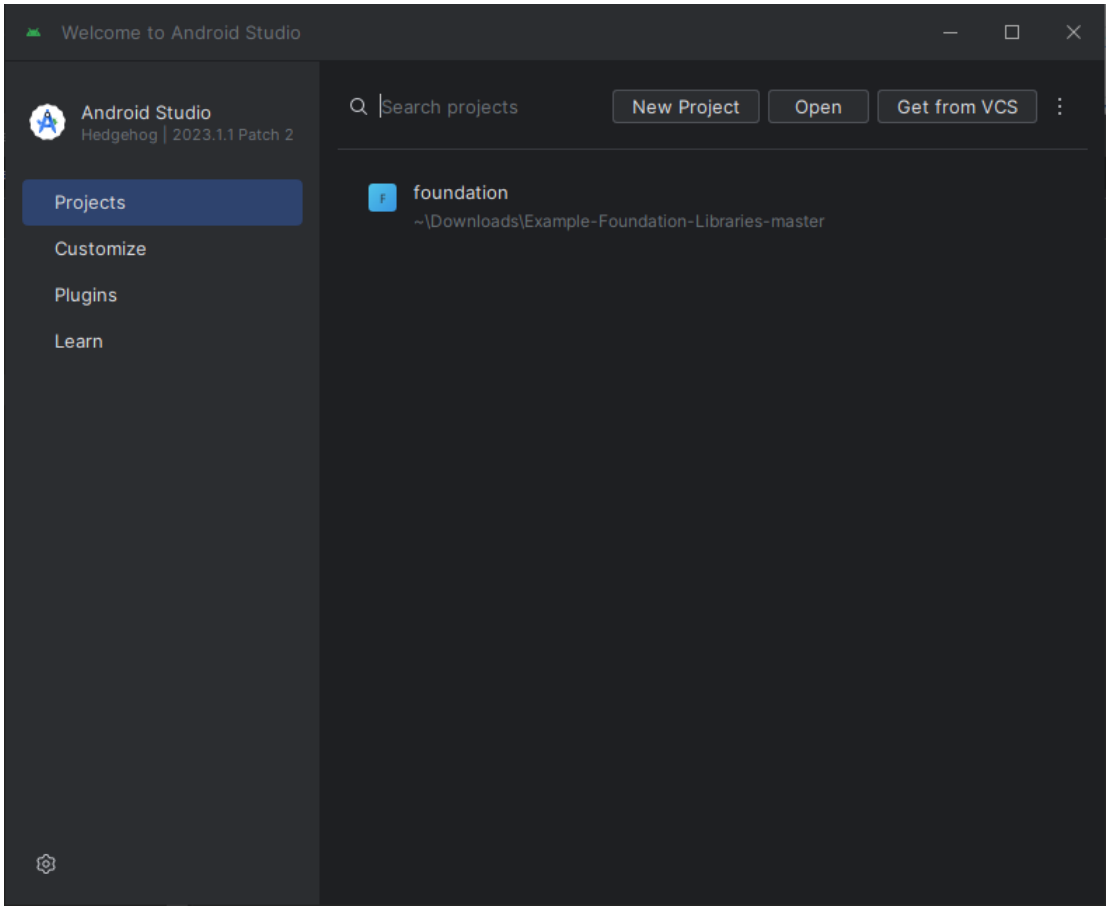
Set up the Application on your IDE

Import the Project

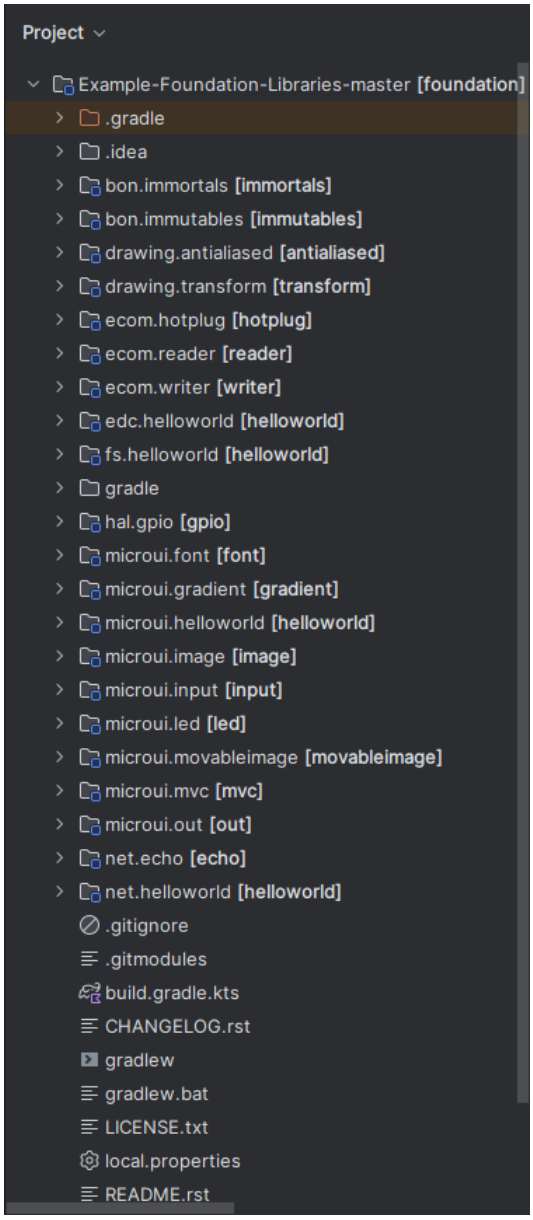
The first step is to import the Application into your IDE:

Note: If you are using an IDE other than Android Studio, please have a look at *Import a Project* section.

- If you are in the Welcome Screen, click on the `Open` button. Otherwise click either on `File` > `Open…` .
- Select the Example-Foundation-Libraries directory located where you downloaded it and click on the `OK` button.



The Gradle project should now be imported in Android Studio, your workspace contains the following project in the Projects view:



Select the VEE Port

For this getting started, we will use the STM32F7508 VEE Port demo module that is available on the *Developer Repository*.

Open the `microui.mvc/build.gradle.kts` Gradle build file, replace the VEE dependency `microejVEE` with the STM32F7508 VEE Port, like this:

```
dependencies {
   microejVee("com.microej.veeport.st.stm32f7508-dk:M5QNX_eval:2.3.0")
}
```

For more information about how to select a VEE Port please refer to the following section: *Select a VEE Port*.

Accept the MICROEJ SDK EULA

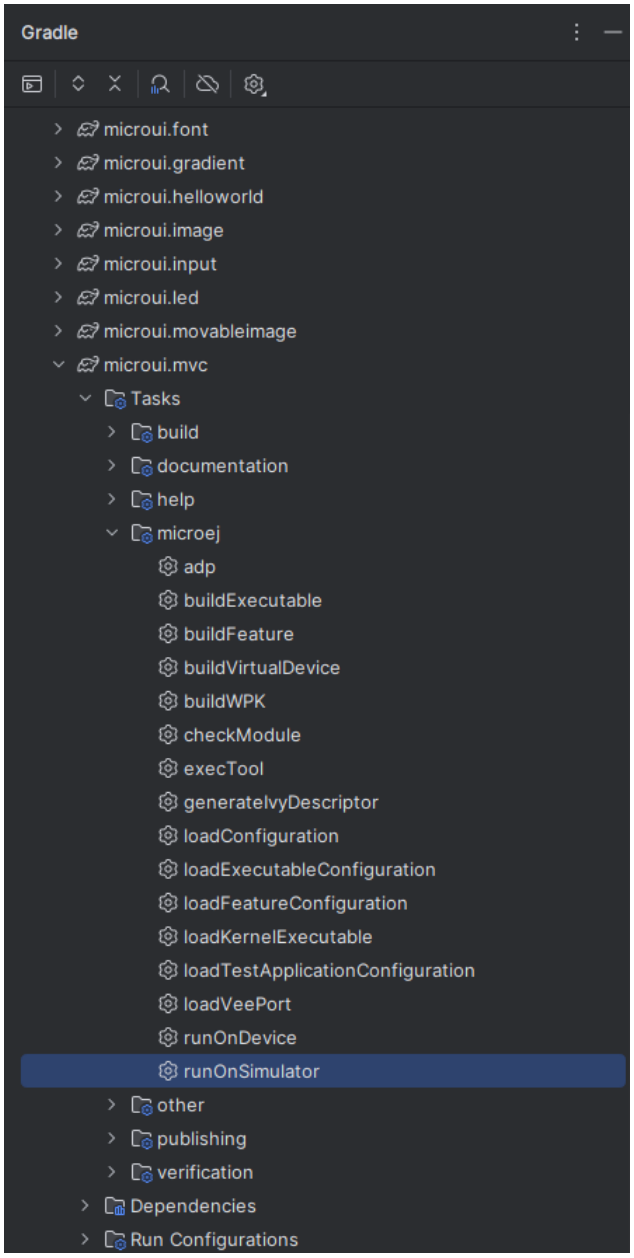
You may have to accept the SDK EULA if you haven't already done it, please have a look at *SDK EULA Acceptation*.

Run an Application on the Virtual Device

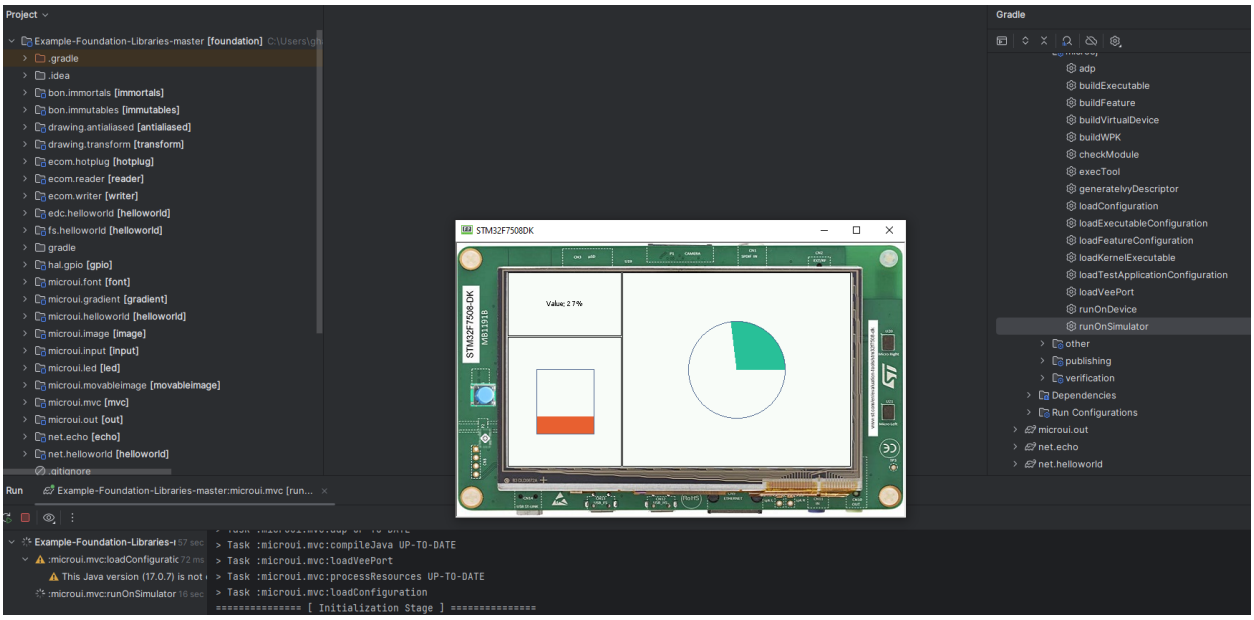
We will be using the `microui.mvc` Application as the sample to test the VEE port simulation execution (you can choose another example it'll work similarly). In order to execute the `microui.mvc` Application on the Virtual Device, the SDK provides the Gradle `runOnSimulator` task.

Note: If you are using an IDE other than Android Studio, please have a look at *Run on Simulator* section.

- Double-click on the `runOnSimulator` task in the Gradle tasks view. It may take few seconds.



The Virtual Device starts and executes the `microui.mvc` application.



Well Done!

Now you know how to run an application on a Virtual Device.

If you want to learn how to run an application on your STM32F7508-DK Evaluation Kit, you can continue this Getting Started: *Run an Application on STM32F7508-DK Evaluation Kit*.

Otherwise, learn how to *Modify the Java Application*.

Run an Application on STM32F7508-DK Evaluation Kit

To deploy `microui.mvc` application on your board, you will have to:

- Setup your Environment (IDE, flashing-tool, hardware setup).
- Request a 30 days *Evaluation License* and install an activation key.
- Build the Executable.
- Flash the board.

Environment Setup

This chapter takes approximately one hour and will take you through the steps to set up your board and build the BSP.

Install the STM32CubeIDE software

Please install the following:

- The STM32CubeIDE version 1.9.0 for STM32F7508-DK, available here.
- The STM32CubeProgrammer utility program, available here.

Be aware that we need the 1.9.0 version of the STM32CubeIDE, also please install the IDE and programmer to the default installation folders, it will simplify future steps.

Hardware Setup

- Check the jumpers configuration on JP1, you only want the `5V link` jumper to be bridged.
- Connect the micro-USB cable to CN14 to power the board.

The USB connection is used as a serial link, a ST-Link probe, and a power input for the board all at once.

The COM port uses the following parameters:

Baudrate	Data bits	Parity bits	Stop bits	Flow control
115200	8	None	1	None

You can have a look at your application logs with an RS232 Terminal (e.g. Termite).

Congratulations! You have finished the setup of your environment. You are now ready to discover how to build and flash a MicroEJ application.

Build the Executable for the STM32F7508-DK Evaluation Kit

In order to build the Executable of the `microui.mvc` Application, the SDK provides the `buildExecutable` Gradle task.

Note: If you are using an IDE other than Android Studio, please have a look at *Build an Executable* section. Come back on this page if you need to activate an Evaluation License.

- Double-click on the `buildExecutable` task in the Gradle tasks view.
- The build stops with a failure.
- Go to the top project in the console view and scroll up to get the following error message:

- Copy the UID. It will be required to activate your Evaluation license.

Request your Evaluation License:

- Request your Evaluation license by following the *Request your Activation Key* instructions. You will be asked to fill the machine UID field with the UID you copied before.
- When you have received your activation key by email, drop it in the license directory by following the *Install the License Key* instructions (drop the license key zip file to the `~/.microej/licenses/` directory).

Now your Evaluation license is installed, you can relaunch your application build by double-clicking on the `buildExecutable` task in the Gradle tasks view. It may take some time.

The Gradle task deploys the Application in the BSP and then builds the BSP using Make.

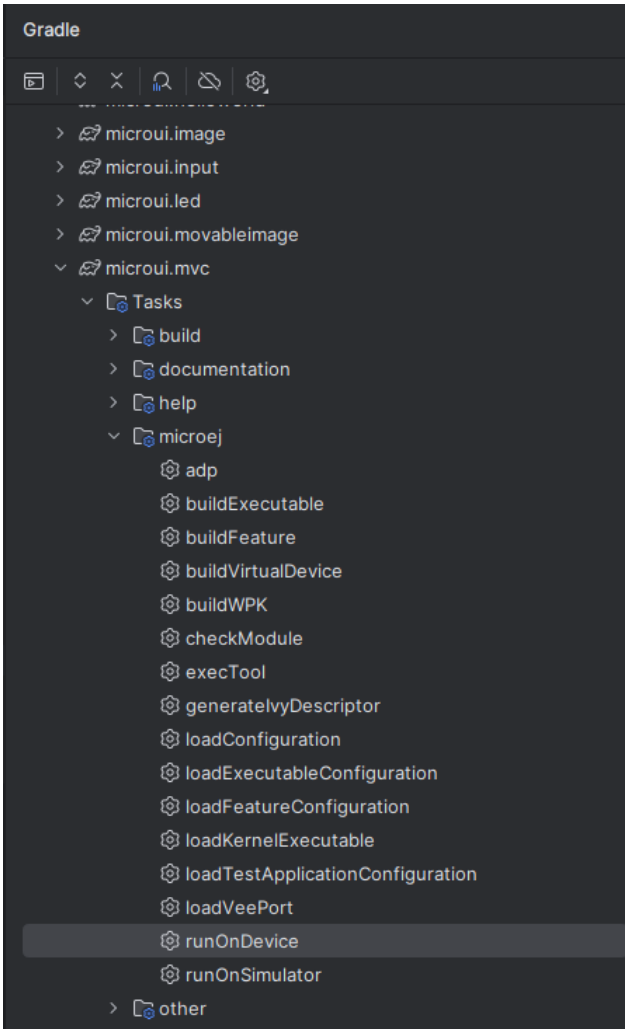
The `microui.mvc` application is built and ready to be flashed on STM32F7508-DK Evaluation Kit once the hardware setup is completed.

Flash the Application on the STM32F7508-DK Evaluation Kit

In order to flash the `microui.mvc` Application on the STM32F7508-DK Evaluation Kit, the application provides the Gradle `runOnDevice` task.

Note: If you are using an IDE other than Android Studio, please have a look at *Run on Device* section.

- Double-click on the `runOnDevice` task in the Gradle tasks view. It may take some time.



Once the firmware is flashed, you should see the `microui.mvc` running on your board.

Modify the Java Application

With MicroEJ, it is easy to modify and test your Java application on the Virtual Device.

For example, we could modify the color used in the pie chart.

- Open the `PieView` file located in the `src/main/java/com/microej/example/foundation/microui/mvc` folder.
- The pie char color is set at line 12, replace the following line:

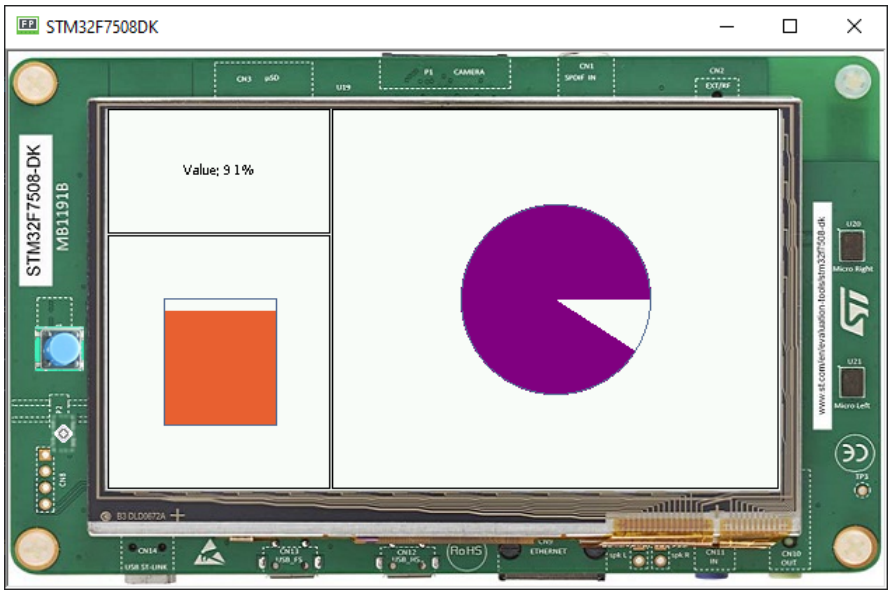
```
public static final int COLOR_CONTENT = 0x2fc19c;    // green
```

by

```
public static final int COLOR_CONTENT = 0x800080;    // purple
```

- Follow *Run an Application on the Virtual Device* instructions to launch the modified application on the Virtual Device.

Here is the modified application running in simulation:



Going Further

You have now successfully executed a MicroEJ application on an embedded device so what's next?

If you are an application developer you can continue to explore MicroEJ's APIs and functionalities by running and studying our samples at GitHub:

Foundation Libraries	Eclasspath	IoT
This project gathers all the basic examples of the foundation libraries.	This project gather all the examples of eclasspath.	This project gathers simple applications using net libraries.
https://github.com/MicroEJ/Example-Foundation-Libraries	https://github.com/MicroEJ/Example-Eclasspath	https://github.com/MicroEJ/Example-IOT

You can also learn how to build bigger and better applications by reading our *Application Developer Guide*.

If you are an embedded engineer you could look at our VEE port examples at GitHub. And to learn how create custom VEE ports you can read our *VEE Porting Guide*.

You can also follow the *Kernel Developer Guide* for more information on our multi-application framework or read about our powerful wearable solution called *VEE Wear*.

Last but not least you can choose to learn about specific topics by following one of our many *Training Courses* ranging from how to easily debug application to setting up a Continuous Integration process and a lot of things in between.

3.3.2 VEE Port Examples

These VEE ports are provided as-is. They can include libraries that are not the latest version. All VEE Port source code examples can be found at GitHub.

3.4 SDK 6 User Guide

MICROEJ SDK 6 provides the tools to write applications for MicroEJ-ready devices and run them on a virtual (simulated) or real device. The capability to execute an application in a simulated environment allows to quickly test changes done in the application code and hence provides a short development feedback loop.

Since the purpose of the SDK is to develop for targeted MCU/MPU computers (IoT, wearable, etc.), it is a cross-development tool. But unlike standard low-level cross-development tools, the SDK offers unique services like hardware simulation. Used with your favorite IDE (Eclipse or IntelliJ IDEA), it provides a complete development environment to create your applications:

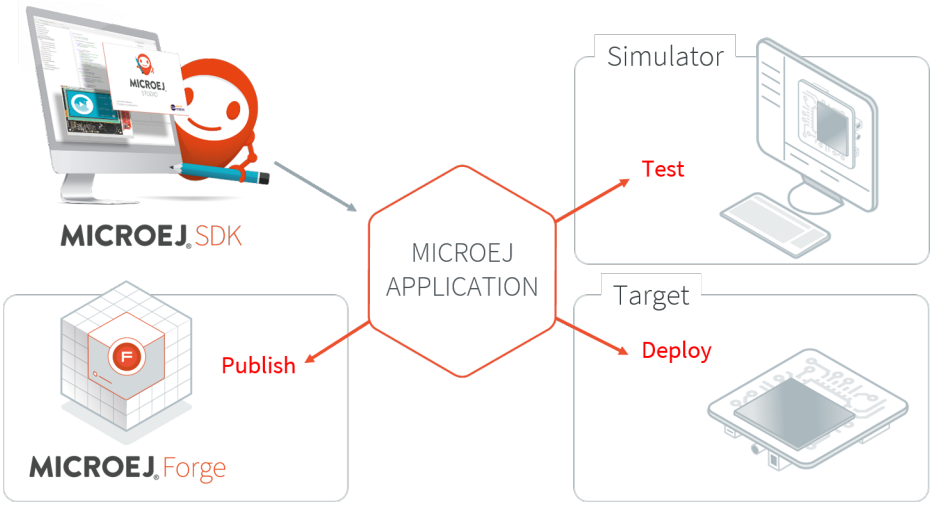


Fig. 8: MicroEJ Application Development Overview

The SDK is composed of the following main elements:

- **Gradle plugins**, the plugins to compile and package MicroEJ modules with Gradle, a popular module and build manager. Gradle provides a Command Line Interface and a complete integration with all the most used IDEs.
- **Architecture**, the software package that includes the MEJ32 port to a target instruction set and a C compiler, SOAR, core libraries and Simulator. See *MicroEJ Architecture* section for more details.

The SDK is licensed under the *SDK End User License Agreement (EULA)*. The following figure shows a detailed view of the elements.

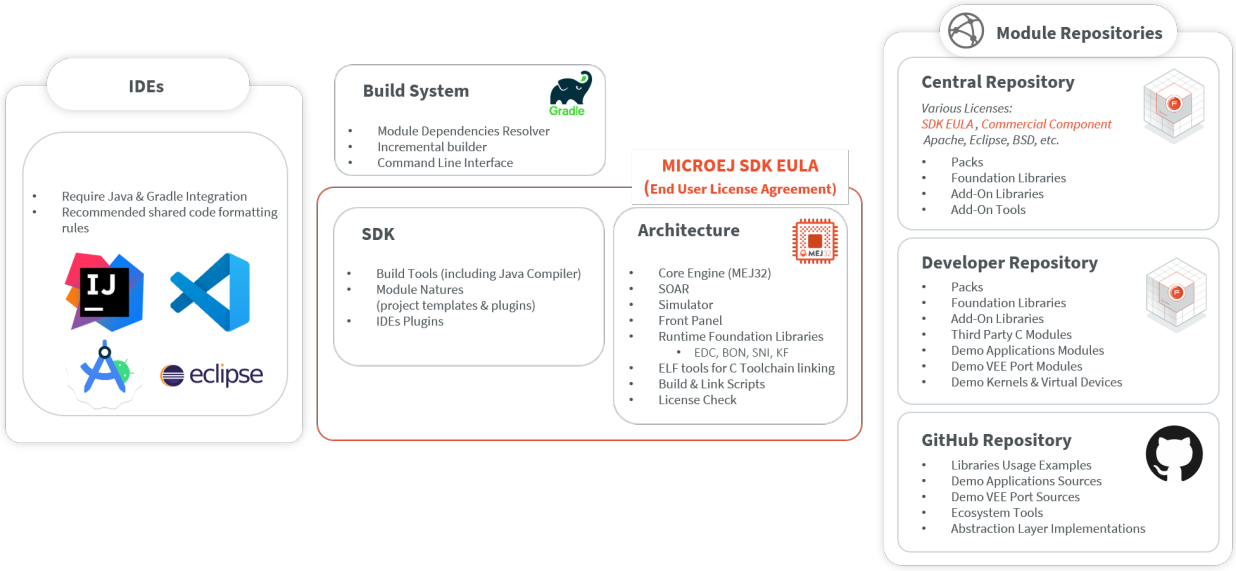


Fig. 9: SDK Detailed View

3.4.1 Installation

This chapter will guide you through the installation process of the SDK on your workstation to start writing applications for MicroEJ-ready devices and running them on a virtual (simulated) or real device. The SDK is mainly composed of Gradle Plugins and IDE Plugins, so the installation process will explain how to perform the following actions:

- Install a JDK (if no compatible version is installed on the workstation).
- Configure the Gradle repositories (to automatically download the Gradle plugins).
- Accept the SDK EULA.
- Install an IDE and the MicroEJ plugin.

First check the System Requirements before proceeding.

System Requirements

- Hardware
  - Intel x64 (Dual-core i5 minimum), ARM or macOS AArch64 (M1) processor
  - 4GB RAM (minimum)
  - 16GB Disk (minimum)
- Operating Systems
  - Windows 11 or Windows 10 with Intel x64 or ARM chip
  - Linux distributions (tested on Ubuntu 20.04, 22.04 and 24.04)
  - macOS x86_64 with Intel chip
  - macOS aarch64 with M1 chip
- Java Runtime Environment

- JDK 11 or higher LTS version (11 , 17 or 21) - Eclipse Temurin Distributions

Check your JDK version

The SDK requires a JDK (not a JRE) 11 or a higher LTS version to be installed and:

- The JAVA_HOME environment variable set to the path of a JDK.

OR

- The java and javac executables of a JDK available in the PATH .

If the JAVA_HOME is set to a JDK, make sure that it is a JDK 11 or a higher LTS version.

If the JAVA_HOME is not set, make sure the JDK executables are available in the PATH environment variable. To check, execute java -version in a terminal:

```
$ java -version
openjdk 11.0.19 2023-04-18
OpenJDK Runtime Environment Temurin-11.0.19+7 (build 11.0.19+7)
OpenJDK 64-Bit Server VM Temurin-11.0.19+7 (build 11.0.19+7, mixed mode)
```

Then execute javac --version :

```
$ javac --version
javac 11.0.19
```

If the java program is not found, it means there is no JDK installed or it is not available in the PATH or JAVA_HOME environment variables. If it is not installed, you can download and install one from Adoptium.

If the java program is found but not the javac program, it means a JRE is installed and used, not a JDK. In this case, install a JDK. You can download and install one from Adoptium.

Configure Repositories

In order to use the SDK Gradle plugins and modules in your project, the *Central* and *Developer* repositories must be configured. There are several ways to declare repositories. To get started, you can declare them globally to make them available in all your projects:

- Create the folder <USER_HOME>/.gradle/init.d if it does not exist.
- Download and copy this file in the previously created folder.

At this stage, you can already build a project from the command line, for example, by executing the command ./gradlew build at the root of the project. But let's continue the installation process to have a complete development environment.

Note: This configuration makes MicroEJ Central and Developer repositories available to every project. If you have several repositories configuration specific to certain projects, you can refer to *multiple repository configuration how-to*

SDK EULA Acceptation

The use of MICROEJ SDK 6 requires to accept the *SDK EULA*.

The acceptance can be done at the project level or system-wide. For a system-wide acceptance, we recommend to define the `accept-microej-sdk-eula-v3-1c` system property in a `gradle.properties` file in your Gradle User Home folder `<USER_HOME>/.gradle/gradle.properties` :

```
systemProp.accept-microej-sdk-eula-v3-1c=YES
```

Refer to *SDK EULA Acceptation* section of *Licenses* to get more information about SDK EULA, and alternative configurations.

Install the IDE

Using an IDE is highly recommended for developing MicroEJ projects, making the development more comfortable and increasing productivity. The following IDEs are supported:

- IntelliJ IDEA (Community or Ultimate edition) - Minimum supported version is `2021.2` .
- Android Studio - Minimum supported version is `Hedgehog - 2023.1.1` .
- Eclipse IDE for Java Developers - Versions from `2022-03` to `2024-06` are supported.
- Visual Studio Code - Minimum supported version is `1.89.0` .

Follow their respective documentation to install one of them.

Install the IDE Plugins

Once your favorite IDE is installed, plugins must be installed to develop MicroEJ Applications.

IntelliJ IDEA

Android Studio

Eclipse

Visual Studio Code

Follow these steps to install the latest stable version of the MicroEJ plugin for IntelliJ IDEA:

- In IntelliJ IDEA, open the Settings window (menu `File` > `Settings…` on Windows and Linux, menu `IntelliJ IDEA` > `Settings…` on macOS).
- Go to `Plugins` menu.
- In the search field, type `MicroEJ` :

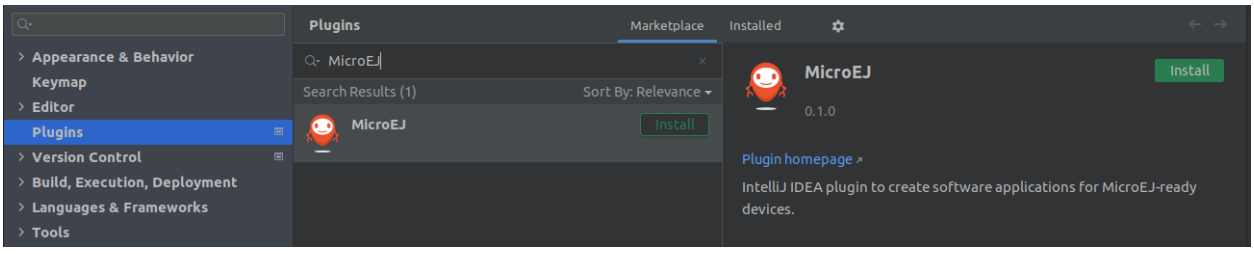


Fig. 10: IntelliJ IDEA Plugin Installation

- Click on the Install button.
- In the upcoming Third-Party Plugins Notice window, click on the Accept button.

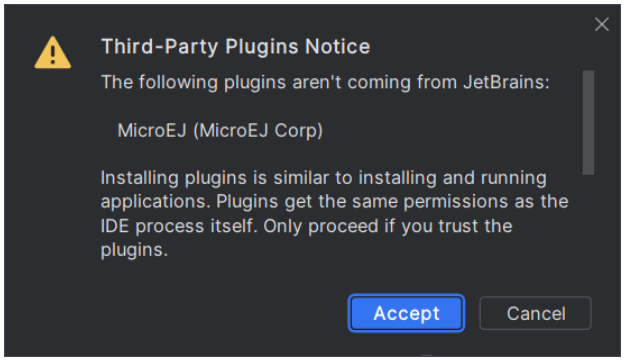


Fig. 11: IntelliJ IDEA Plugin Installation - Third-Party Plugins Notice

- Click on the Restart IDE button.

Warning: Until version 2025.2 , due to an IntelliJ IDEA bug, a lot of errors are raised in Gradle build files (build.gradle.kts). It has been fixed in version 2025.3 , so it is highly recommended to upgrade to this version. If you cannot upgrade to this version, you should follow these steps to remove these errors:

- In IntelliJ IDEA, open the Settings window (menu File > Settings… on Windows and Linux, menu IntelliJ IDEA > Settings… on macOS).
- Go to the Languages & Frameworks > Kotlin menu.
- Uncheck Enable K2 .

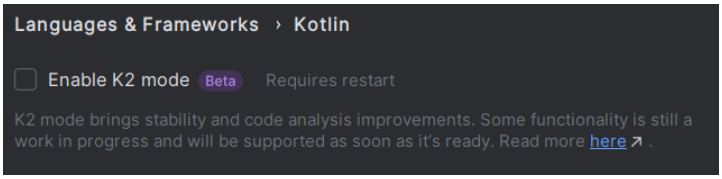


Fig. 12: Disable K2 in IntelliJ IDEA

- Go to Files > Invalidate Caches… , and click on Invalidate and Restart .

Follow these steps to install the latest stable version of the MicroEJ plugin for Android Studio:

- In Android Studio, open the Settings window (menu File > Settings… on Windows and Linux, menu Android Studio > Settings… on macOS).
- Go to Plugins menu.

- In the search field, type MicroEJ for Android Studio :

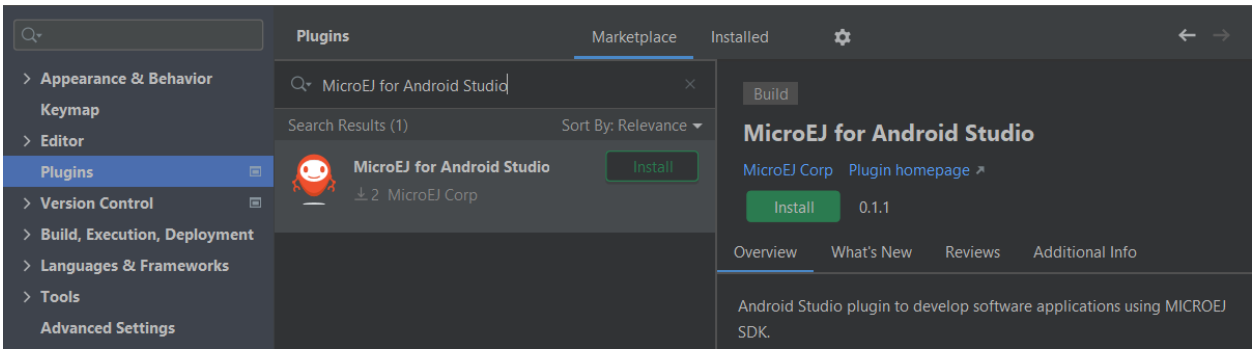


Fig. 13: Android Studio Plugin Installation

- Click on the Install button.
- In the upcoming Third-Party Plugins Notice window, click on the Accept button.

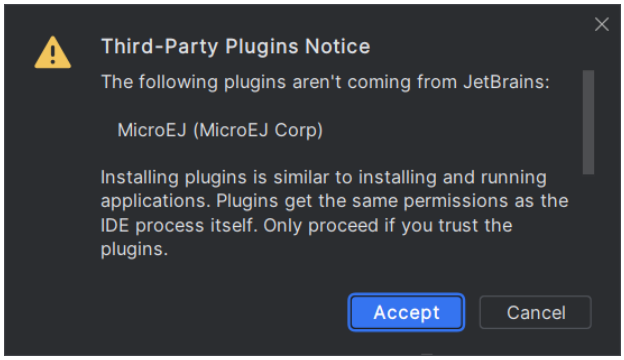


Fig. 14: Android Studio Plugin Installation - Third-Party Plugins Notice

- Click on the Restart IDE button.

Warning: There used to be a unique plugin for both Android Studio and IntelliJ IDEA. Each IDE now has its own dedicated plugin, so if the IntelliJ IDEA MicroEJ plugin has been previously installed, you should uninstall it and install MicroEJ for Android Studio instead.

Follow these steps to install the latest stable version of the MicroEJ plugin for Eclipse:

- In Eclipse, go to Help > Eclipse Marketplace… .
- In the search field, type MicroEJ and press Enter:

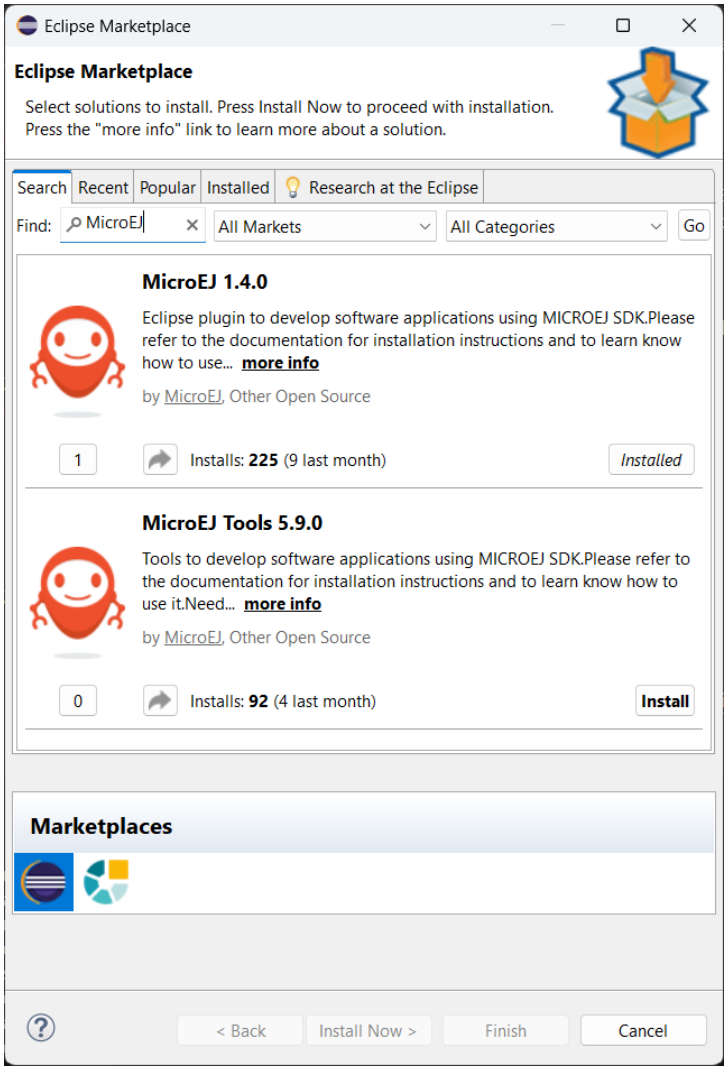


Fig. 15: Eclipse Plugin Installation - Marketplace

- Click on the Install button of the MicroEJ plugin.
- Accept the license agreement and click on the Finish button.
- In the upcoming Trust Authorities window, check the https://repository.microej.com item and click on the Trust Selected button.

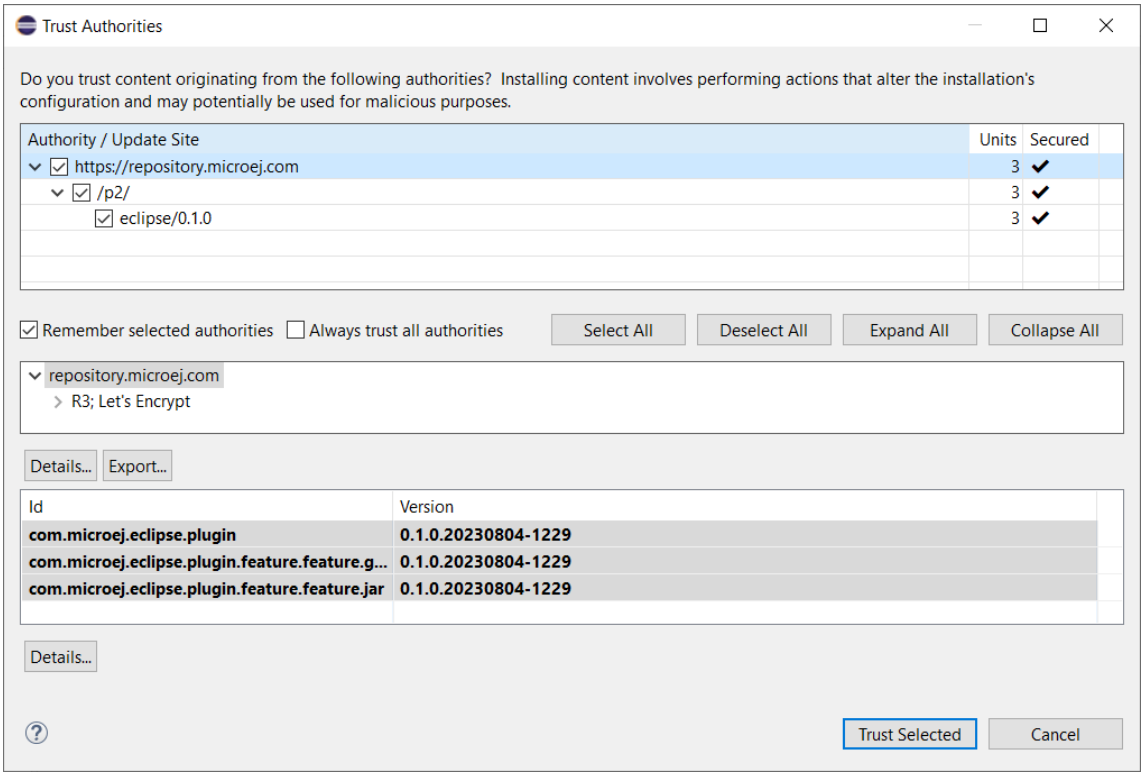


Fig. 16: Eclipse Plugin Installation - Trust Authorities

- In the upcoming Trust Artifacts window, check the Unsigned item and click on Trust Selected button.

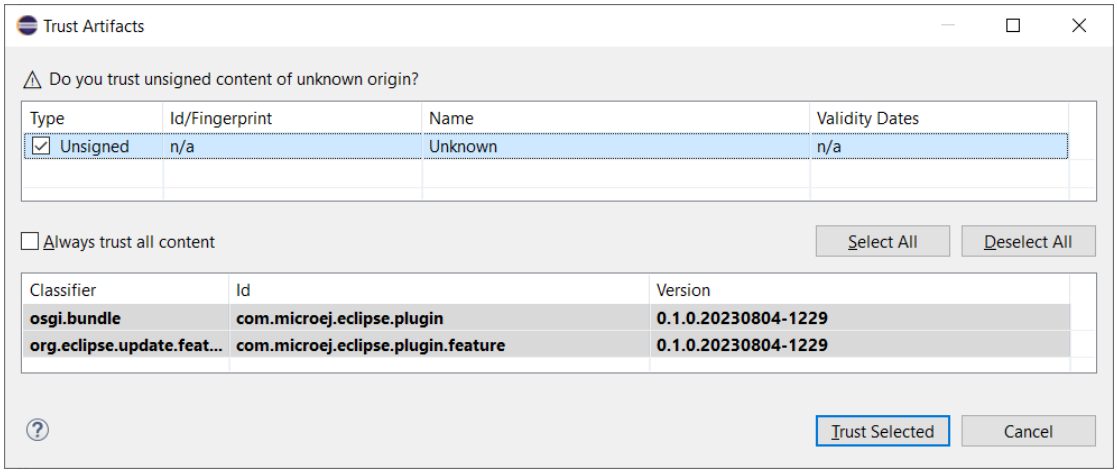


Fig. 17: Eclipse Plugin Installation - Trust Artifacts

- In the upcoming window, click on the Restart Now button.

MicroEJ does not provide a dedicated extension for VS Code, but Microsoft provides a extension that brings a useful collection of extensions for Java called Extension Pack for Java. To install this extension:

- In Visual Studio Code, open the Extensions tab (Ctrl+Shift+X).

- In the search field, type extension pack for Java :

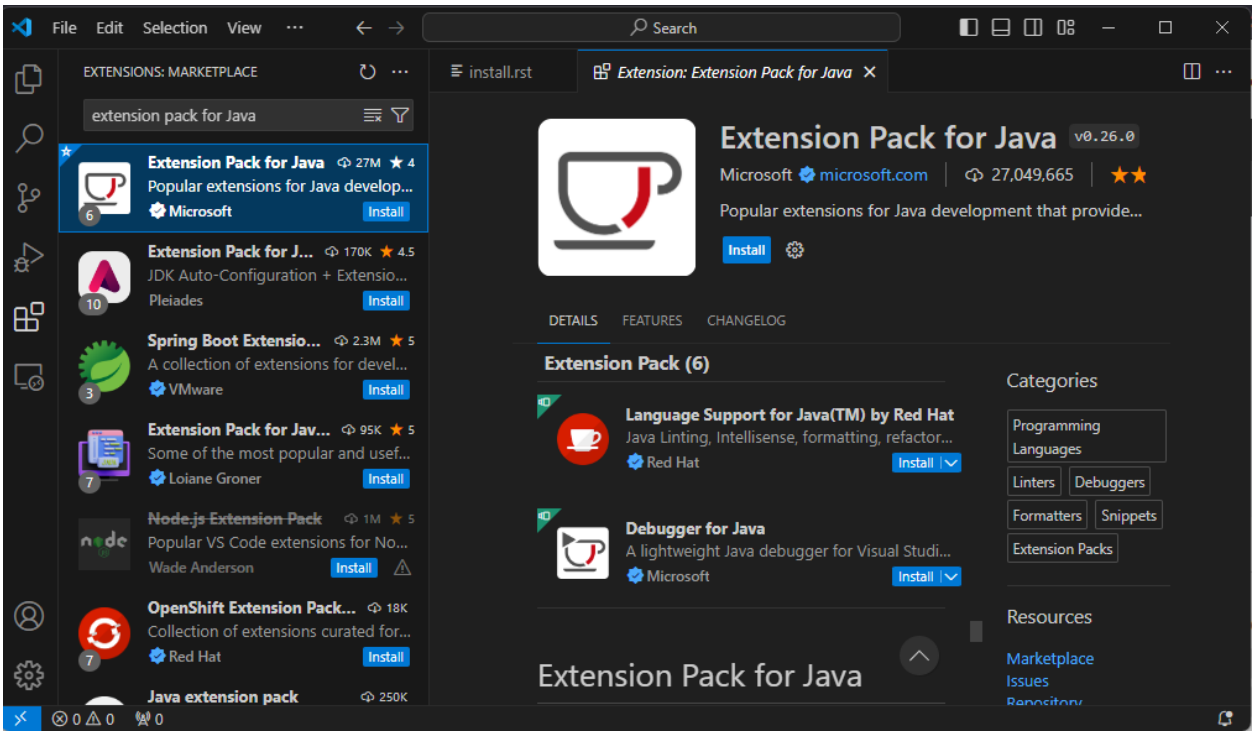


Fig. 18: VS Code Java Extensions Installation

- Click on the Install button of the extension.

Finally, to have an accurate code completion, you need to adjust two settings:

- Go to File > Preferences > Settings .
- In the search bar, type java.gradle.buildServer.enabled and set it to off :

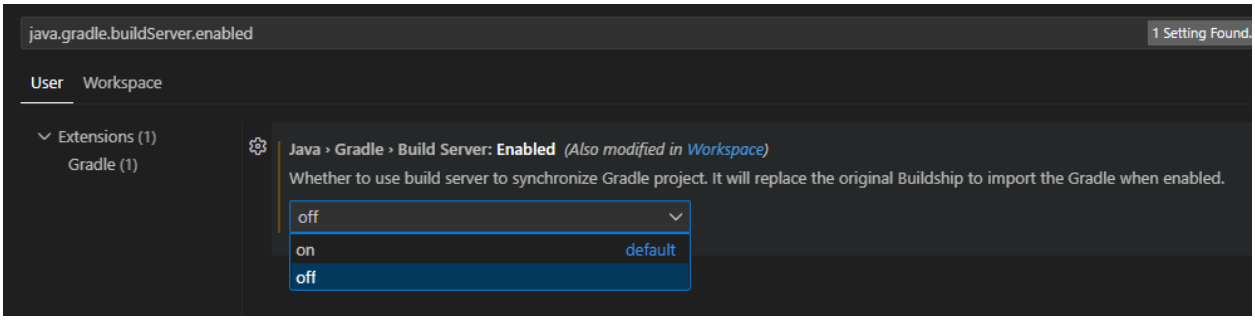


Fig. 19: VS Code Java Gradle Build Server Setting

- Then, in the search bar again, type java.configuration.updateBuildConfiguration and set it to automatic :

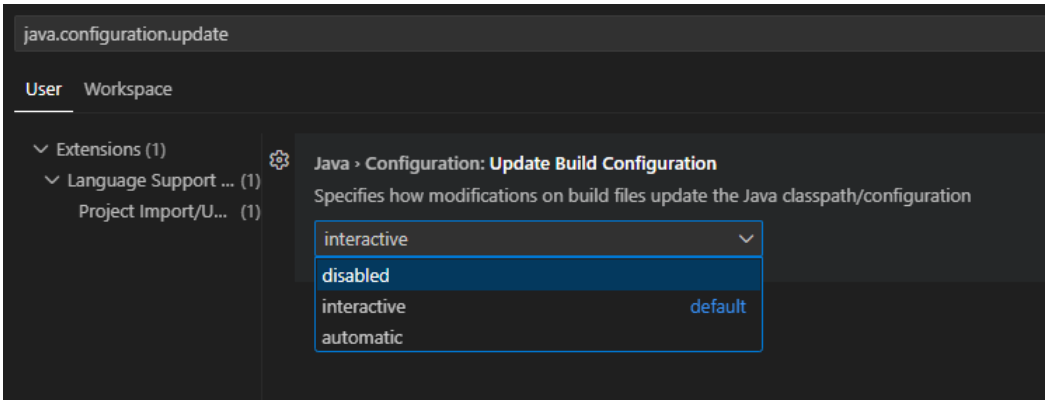


Fig. 20: VS Code Java Update Build Configuration Setting

Note: The extension ships with an embedded JDK, which is used for UI-triggered Gradle builds (for example via the Gradle view or automatic project import). In contrast, builds started from the Terminal (Git Bash, PowerShell, etc.) use the JDK pointed to by `JAVA_HOME` or the one found on the `PATH`. It is recommended to explicitly configure the JDK used by the VS Code Java extension, as this ensures that Gradle builds launched from the UI use the same JDK as the Terminal.

- Go to `File > Preferences > Settings`.
- In the search bar, type `java.import.gradle.java.home` and set it to the path to the correct JDK

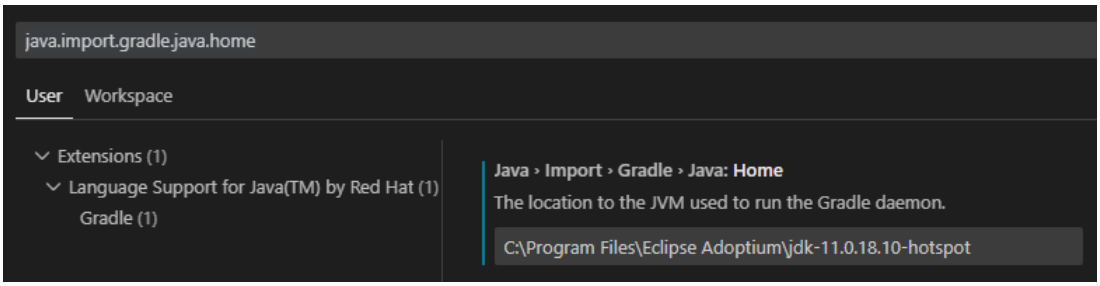


Fig. 21: VS Code Java Update JDK Setting

3.4.2 Licenses

SDK EULA

MICROEJ SDK is licensed under the SDK End User License Agreement (EULA). The following figure shows a detailed view of the elements.

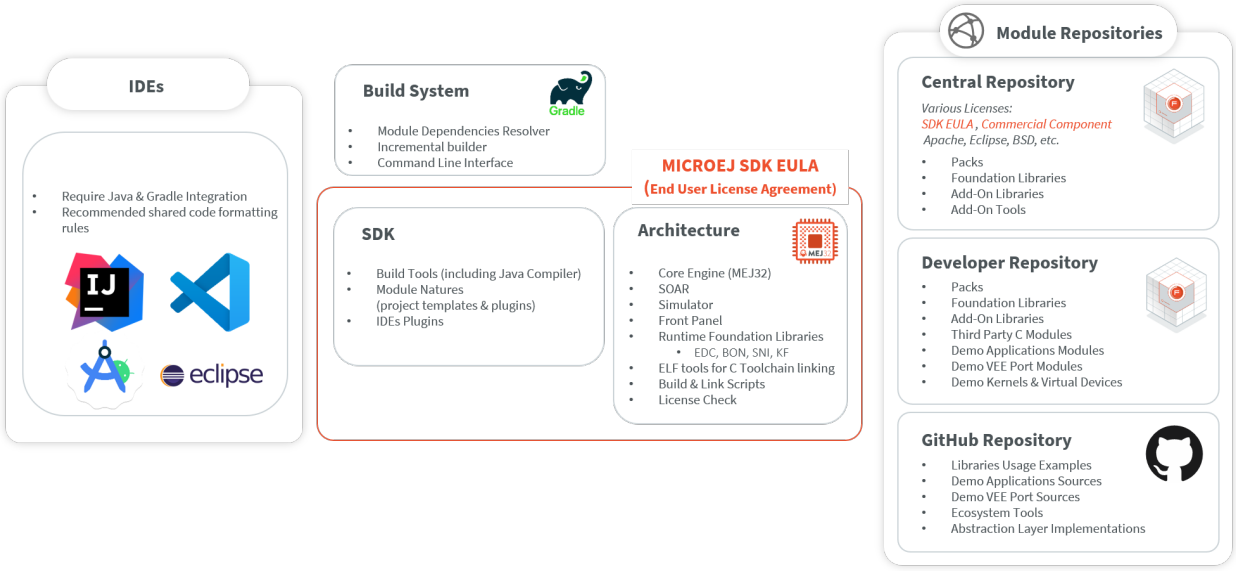


Fig. 22: SDK Detailed View

Commercial Component License

One of the main features of MICROEJ SDK is to download, install and link modules with MICROEJ VEE to produce a binary Executable for a target device. Modules provided by MicroEJ Corp. are hosted in various *repositories*, each distributed under specific license terms detailed in its respective LICENSE.txt file. Some of these modules are licensed under the Commercial Component License. The Commercial Component license extends the SDK EULA terms by adding specific requirements and restrictions related to production deployment.

License Manager Overview

Architectures are distributed in two different versions:

- Evaluation Architectures, associated with a software license key. They can be downloaded at https://repository.microej.com/modules/com/microej/architecture/.
- Production Architectures, associated with a hardware license key stored on a USB dongle. They can be requested to *our support team*.

The license manager is provided with Architectures and then integrated into VEE Ports.

License Check

The table below summarizes where the license is checked.

Application	Run on Simulator (Virtual Device)	Build on Device	Documentation Link
Application containing a Java main class	NO	**YES**	*Run on Device*
Application containing a Feature class	NO	NO	*Application Linking*

SDK EULA Acceptation

The use of MICROEJ SDK 6 requires to accept the *SDK EULA*. If the SDK EULA is not accepted, the following message is displayed when executing a Gradle task:

```
> The MICROEJ SDK End-User License Agreement (EULA) must be accepted before it can start.
  The license terms for this product can be downloaded from
  https://repository.microej.com/licenses/sdk/LAW-0011-LCS-MicroEJ_SDK-EULA-v3.1C.txt
  You can accept the EULA by setting the system property systemProp.accept-microej-sdk-eula-v3-1c=YES in a
↪gradle.properties file
        - in your Gradle User Home folder (<USER_HOME>/.gradle/),
        - or in the root folder of your project,
  or specifying the -Daccept-microej-sdk-eula-v3-1c=YES command line option,
  or setting the ACCEPT_MICROEJ_SDK_EULA_V3_1C=YES environment variable.
```

As mentioned in the message, there are several ways to accept the SDK EULA:

- define the `accept-microej-sdk-eula-v3-1c` system property in a `gradle.properties` file with the `systemProp.` prefix:

  ```
  systemProp.accept-microej-sdk-eula-v3-1c=YES
  ```

  This can be in the `gradle.properties` of your Gradle User Home folder (located by default at `<USER_HOME>/.gradle/gradle.properties`), or in the `gradle.properties` file at the root of your project for example.

- define the `accept-microej-sdk-eula-v3-1c` system property in the command line:

  ```
  ./gradlew build -Daccept-microej-sdk-eula-v3-1c=YES
  ```

- set the `ACCEPT_MICROEJ_SDK_EULA_V3_1C` environment variable to `YES`.

Evaluation Licenses

This section should be considered when using Evaluation Architectures, which use software license keys. A machine UID needs to be provided to activate an Evaluation license on the MicroEJ Licenses Server. The machine UID is a 16 hexadecimal digits number.

Get your Machine UID

If your VEE Port is *defined* in the `build.gradle.kts` of your project, the machine UID will be displayed when building an *Executable*.

```
[INFO ] Launching in Evaluation mode. Your UID is XXXXXXXXXXXXXXXX.
[ERROR] Invalid license check (No license found).
```

Request your Activation Key

- Go to MicroEJ Licenses Server https://license.microej.com.
- Click on Create a new account link.
- Create your account with a valid email address. You will receive a confirmation email a few minutes after. Click on the confirmation link in the email and log in with your new account.
- Click on Activate a License .
- Set Product P/N: to 9PEVNLDBU6IJ .
- Set UID: to the machine UID you copied before.
- Click on Activate .
- The license is being activated. You should receive your activation by email in less than 5 minutes. If not, please contact *our support team*.
- Once received by email, save the attached zip file that contains your activation key.

Install the License Key

The license key zip file must be simply dropped to the ~/.microej/licenses/ directory (create it if it doesn't exist).

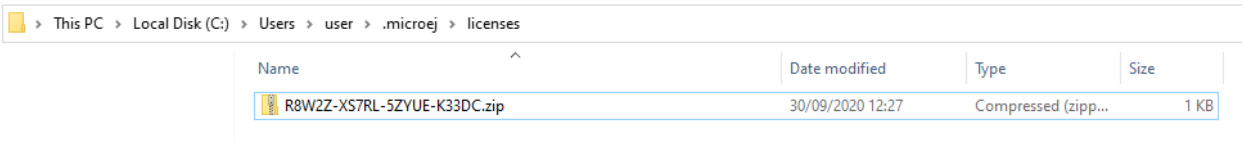


Fig. 23: MicroEJ Shared Licenses Directory

Check Activation

This section contains instructions that will allow you to verify that your Evaluation license has been properly installed.

Note: The command line tool requires *Architecture 8.3.0* or higher.

To get more details on installed Evaluation licenses, proceed with the following steps:

1. Open a terminal.
2. Change directory to an Evaluation VEE Port.
3. Execute the command:

   ```
   java -Djava.library.path=resources/os/[OS_NAME] -jar licenseManager/licenseManagerKeyHardware.jar
   ```

   with OS_NAME set to Windows64 for Windows OS, Linux64 for Linux OS, Mac for macOS x86_64 (Intel chip) or MacA64 for macOS aarch64 (M1 chip).

   You should get something similar to the following output:

```
[DEBUG] ===== MicroEJ Evaluation License Debug Tool =====
[DEBUG] => UID: 39B7C108972A5C36.
[DEBUG] => Please specify a license directory containing 'keysHardware.txt'
```

This step is sufficient if you want to check the machine's UID. For an analysis of the available licenses, proceed with the following steps.

4. *Build your Executable* with verbose mode enabled.
5. Retrieve in the logs the path to the licenses directory by searching for `-Dlicenses.working.dir=[path_to_license_dir]`.
6. Execute the same command than before with the path to the licenses directory as argument:

```
java -Djava.library.path=resources/os/[OS_NAME] -jar licenseManager/licenseManagerKeyHardware.jar␣
↪[path_to_license_dir]
```

You should get something similar to the following output:

```
[DEBUG] ===== MicroEJ Evaluation License Debug Tool =====
[DEBUG] => UID: 39B7C108972A5C36.
[DEBUG] => Detected MicroEJ License HQB48-VCQDQ-I7QDL-IAZUF - valid until YYYY-MM-DD.
[DEBUG] ===== SUCCESS =====
```

Now the list of detected licenses and their validity are dumped.

Expiration Warning Message

When a license is about to expire, a warning message is displayed when building an *Executable*:

```
================================================================================
[WARNING] Your license XXXXX-XXXXX-XXXXX-XXXXX will expire in 20 days.
================================================================================
```

By default the warning is displayed starting 20 days before the expiration date for Evaluation licenses and 30 days for Production licenses. These values can be changed in the `build.gradle.kts` of your project:

```
microej {
  daysBeforeEvaluationLicenseExpirationMessage = 20
  daysBeforeProductionLicenseExpirationMessage = 30
}
```

Troubleshooting

Machine UID has changed

This can occur when the hardware configuration of the machine is changed (especially when the network interfaces have changed).

In this case, you can either request a new activation key for this new UID or go back to the previous hardware configuration.

Production Licenses

This section should be considered when using Production Architectures, which use hardware license keys stored on a USB dongle.

Fig. 24: MicroEJ USB Dongle

Note: If your USB dongle has been provided to you by your sales representative and you don't have received an activation certificate by email, it may be a pre-activated dongle. Then you can skip the activation steps and directly jump to the *Check Activation* section.

Request your Activation Key

- Go to license.microej.com.
- Click on `Create a new account` link.
- Create your account with a valid email address. You will receive a confirmation email a few minutes after. Click on the confirmation link in the email and login with your new account.
- Click on `Activate a License` .
- Set `Product P/N:` to **The P/N on the activation certificate**.
- Enter your UID: serial number printed on the USB dongle label (8 alphanumeric char.).
- Click on `Activate` and check the confirmation message.
- Click on `Confirm your registration` .
- Enter the **Registration Code provided on the activation certificate**.
- Click on `Submit` .
- Your Activation Key will be sent to you by email as soon as it is available (12 business hours max.).

Note: You can check the `My Products` page to verify your product registration status, the Activation Key availability, and download the Activation Key when available.

Once the Activation Key is available, download and save the Activation Key ZIP file to a local directory.

Activate your USB Dongle

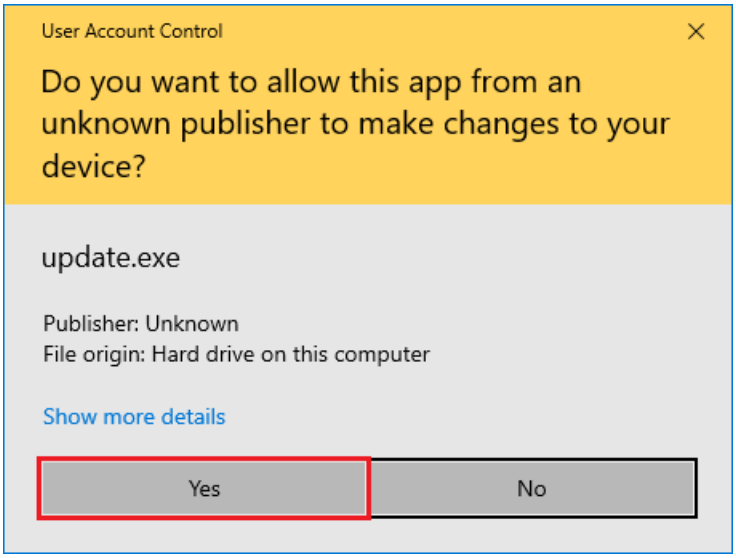
This section contains instructions that will allow you to flash your USB dongle with the proper activation key.

You shall ensure that the following prerequisites are met :

- Your *operating system* is Windows
- The USB dongle is plugged and recognized by your operating system (see *Troubleshooting* section)
- No more than one USB dongle is plugged into the computer while running the update tool
- The update tool is not launched from a network drive or a USB key
- The activation key you downloaded is the one for the dongle UID on the sticker attached to the dongle (each activation key is tied to the unique hardware ID of the dongle).

You can then proceed to the USB dongle update:

- Unzip the Activation Key file to a local directory
- Enter the directory just created by your ZIP extraction tool.
- Launch the executable program.
- Accept running the unsigned software if requested (Windows 10/11)



- Click on the Update button (no password needed)

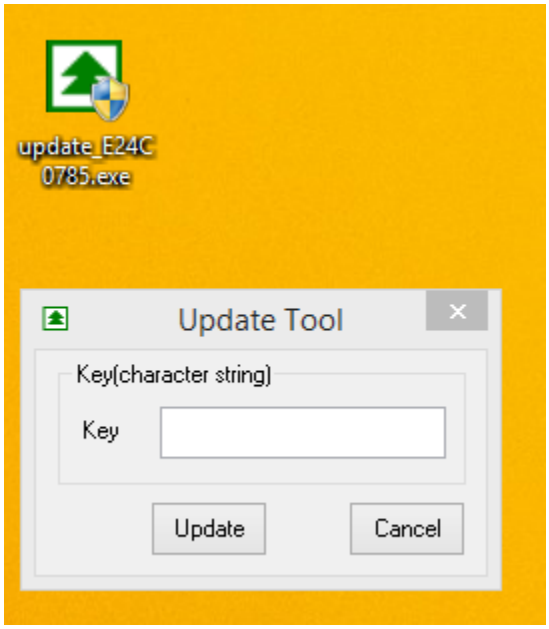


Fig. 25: Dongle Update Tool

- On success, an Update successfully message shall appear. On failure, an Error key or no proper rockey message may appear.

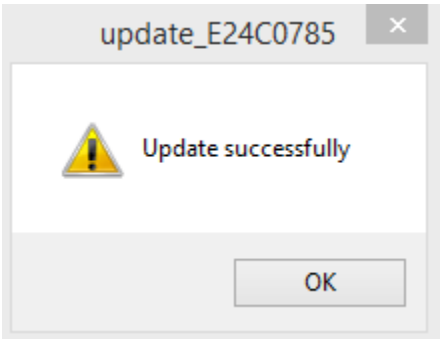


Fig. 26: Successful Dongle Update

Check Activation

This section contains instructions that will allow you to verify that your USB dongle has been properly activated.

To get more details on connected USB dongle(s), run the debug tool as following:

1. Open a terminal.
2. Change directory to a Production VEE Port.
3. Execute the command:

Architecture v8.1.0 or higher

Architecture v8.0.0 or lower

```
java -Djava.library.path=resources/os/[OS_NAME] -jar licenseManager/licenseManagerProduct.jar
```

```
java -Djava.library.path=resources/os/[OS_NAME] -jar licenseManager/licenseManagerUsbDongle.jar
```

with `OS_NAME` set to `Windows64` for Windows OS, `Linux64` for Linux OS, `Mac` for macOS x86_64 (Intel chip) or `MacA64` for macOS aarch64 (Apple Silicon chip).

If your USB dongle has been properly activated, you should get the following output:

```
[DEBUG] ===== MicroEJ Dongle Debug Tool =====
[DEBUG] => Detected dongle UID: XXXXXXXX.
[DEBUG] => Dongle UID has valid MicroEJ data: XXXXXXXX (only the first one is listed).
[DEBUG] => Detected MicroEJ License XXXXX-XXXXX-XXXXX-XXXXX - valid until YYYY-
↪MM-DD.
[DEBUG] ===== SUCCESS =====
```

USB Dongle on GNU/Linux

For GNU/Linux Users (Ubuntu at least), by default, the dongle access has not been granted to the user, you have to modify udev rules. Please create a `/etc/udev/rules.d/91-usbdongle.rules` file with the following contents:

```
ACTION!="add", GOTO="usbdongle_end"
  SUBSYSTEM=="usb", GOTO="usbdongle_start"
  SUBSYSTEMS=="usb", GOTO="usbdongle_start"
  GOTO="usbdongle_end"

  LABEL="usbdongle_start"

  ATTRS{idVendor}=="096e" , ATTRS{idProduct}=="0006" , MODE="0666"

  LABEL="usbdongle_end"
```

Then, restart udev: `sudo /etc/init.d/udev restart`

You can check that the device is recognized by running the `lsusb` command. The output of the command should contain a line similar to the one below for each dongle: `Bus 002 Device 003: ID 096e:0006 Feitian Technologies, Inc.`

USB Dongle with Docker on Linux

If you use the SDK Docker image on a Linux host to build an Executable, the dongle must be mapped to the Docker container. First, it requires to add a symlink on the dongle by following the instructions of the *USB Dongle on GNU/Linux* section but with this `/etc/udev/rules.d/91-usbdongle.rules` file:

```
ACTION!="add", GOTO="usbdongle_end"
  SUBSYSTEM=="usb", GOTO="usbdongle_start"
  SUBSYSTEMS=="usb", GOTO="usbdongle_start"
  GOTO="usbdongle_end"

  LABEL="usbdongle_start"

  ATTRS{idVendor}=="096e" , ATTRS{idProduct}=="0006" , MODE="0666" , SYMLINK+="microej_dongle
↪"
```

(continues on next page)

(continued from previous page)

```
LABEL="usbdongle_end"
```

Then the symlink has to be mapped in the Docker container by adding the following option in the Docker container creation command line:

```
--device /dev/microej_dongle:/dev/bus/usb/999/microej_dongle
```

The `/dev/microej_dongle` symlink can be mapped to any device path as long as it is in `/dev/bus/usb`.

USB Dongle with WSL

Note: The following steps have been tested on WSL2 with Ubuntu 22.04.2 LTS.

To use a USB dongle with WSL, you first need to install *usbipd* following the steps described in Microsoft WSL documentation:

First, check that WSL2 is installed on your system. If not, install it or update it following Microsoft Documentation

Then, you need install usbipd-win v4.0.0 or higher on Windows from usbipd-win Github repository.

And then, install usbipd and update hardware database inside you WSL installation:

```
sudo apt install linux-tools-generic hwdata
sudo update-alternatives --install /usr/local/bin/usbip usbip /usr/lib/linux-tools/*-generic/usbip 20
```

Add the udev rule described in *USB Dongle on GNU/Linux*, and restart udev:

```
sudo /etc/init.d/udev restart
```

Ensure your USB dongle is plugged, then start a PowerShell terminal in administrator mode.

List the connected devices with the following command:

```
usbipd.exe list
```

You should see your USB dongle connected with `VID:PID==096e:0006`:

```
PS C:\Users\user> usbipd list
Connected:
BUSID  VID:PID    DEVICE                                                        STATE
2-6    0c45:674c  Integrated Webcam, Integrated IR Webcam, USB DFU              Not shared
2-8    0a5c:5843  Dell ControlVault w/ Fingerprint Touch Sensor, Microsoft ...  Not shared
2-10   8087:0033  Intel(R) Wireless Bluetooth(R)                                Not shared
3-1    0bda:8153  Realtek USB GbE Family Controller                             Not shared
4-6    413c:c010  Dell DA310                                                    Not shared
6-4    096e:0006  USB Input Device                                              Not shared
6-6    046d:0819  USB Video Device, USB Audio Device                            Not shared
7-1    045e:0084  USB Input Device                                              Not shared
7-2    04d9:1400  USB Input Device                                              Not shared
7-3    10d5:55a2  USB Input Device                                              Not shared
```

Here the `BUSID` is `6-4`.

Bind and attach the dongle to WSL:

```
usbipd.exe bind --busid <BUSID>
usbipd.exe attach --wsl  --busid <BUSID>
```

Open a bash terminal in your WSL instance, and check the USB dongle is successfully mounted with the following command:

```
lsusb
```

You should see your USB dongle connected with `ID 096e:0006` :

```
Bus 002 Device 001: ID 1d6b:0003 Linux Foundation 3.0 root hub
Bus 001 Device 002: ID 096e:0006 Feitian Technologies, Inc. HID Dongle (for OEMs - manufacturer string is
↪"OEM")
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
```

Note: You'll need to follow these steps each time you system is rebooted or the dongle is plugged/unplugged.

Troubleshooting

This section contains instructions to check that your operating system correctly recognizes your USB dongle.

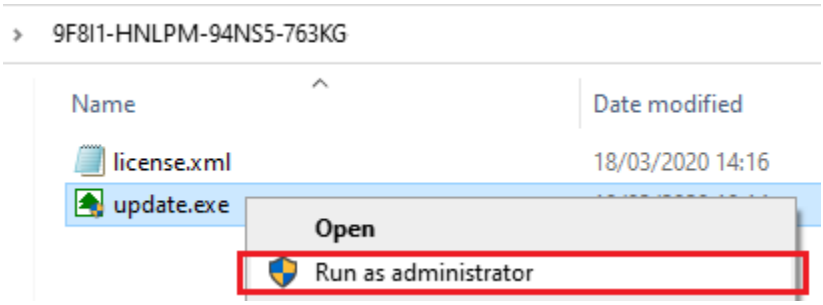
Windows Troubleshooting

- If the *dongle activation* failed with `No rockey` message, check there is one and only one dongle recognized with the following hardware ID :

  ```
  HID\VID_096E&PID_0006&REV_0201
  ```

  Go to the `Device Manager` > `Human Interface Devices` and check among the `USB Input Device` entries that the `Details` > `Hardware Ids` property match the ID mentioned before.

- If the *dongle activation* was successful with `Update successfully` message but the license does not appear in the SDK or is not updated, try to activate again by starting the executable with administrator privileges:



- If the following error message is thrown when building an Executable, either the dongle plugged is a verbatim dongle or it has not been successfully *activated*:

  ```
  Invalid license check (Dongle found is not compatible).
  ```

VirtualBox Troubleshooting

In a VirtualBox virtual machine, USB drives must be enabled to be recognized correctly. Make sure to enable the USB dongle by clicking on it in the VirtualBox menu Devices > USB .

To make this setting persistent, go to Devices > USB > USB Settings… and add the USB dongle in the USB Devices Filters list.

WSL Troubleshooting

Check that your dongle is attached to WSL from PowerShell:

```
usbipd.exe list
```

You should have a line saying Attached - Ubuntu :

```
PS C:\Users\sdkuser> usbipd.exe list
BUSID  VID:PID    DEVICE                                                        STATE
2-1    096e:0006  USB Input Device                                              Attached - Ubuntu
2-6    0c45:6a10  Integrated Webcam                                             Not attached
2-10   8087:0026  Intel(R) Wireless Bluetooth(R)                                Not attached
3-1    045e:0823  USB Input Device                                              Not attached
3-4    046d:c31c  USB Input Device                                              Not attached
```

In you WSL console, the dongle must also be recognized. Ckeck by using lsusb :

```
skduser@host:~/workspaces/docs$ lsusb
Bus 002 Device 001: ID 1d6b:0003 Linux Foundation 3.0 root hub
Bus 001 Device 003: ID 096e:0006 Feitian Technologies, Inc. HID Dongle (for OEMs - manufacturer␣
↪string is "OEM")
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
```

This might not be sufficient. If you're still facing license issues, restart udev, abd attach your dongle to WSL once again.

Note: Hibernation may have unattached your dongle. Reload udev, unplug/plug your dongle and attach it from PowerShell.

Remote USB Dongle Connection

When the dongle cannot be physically plugged to the machine running the SDK (cloud builds, virtualization, missing permissions, …), it can be configured using USB redirection over IP network.

There are many hardware and software solutions available on the market. Among others, this has been tested with https://www.net-usb.com/ and https://www.virtualhere.com/. Please contact *our support team* for more details.

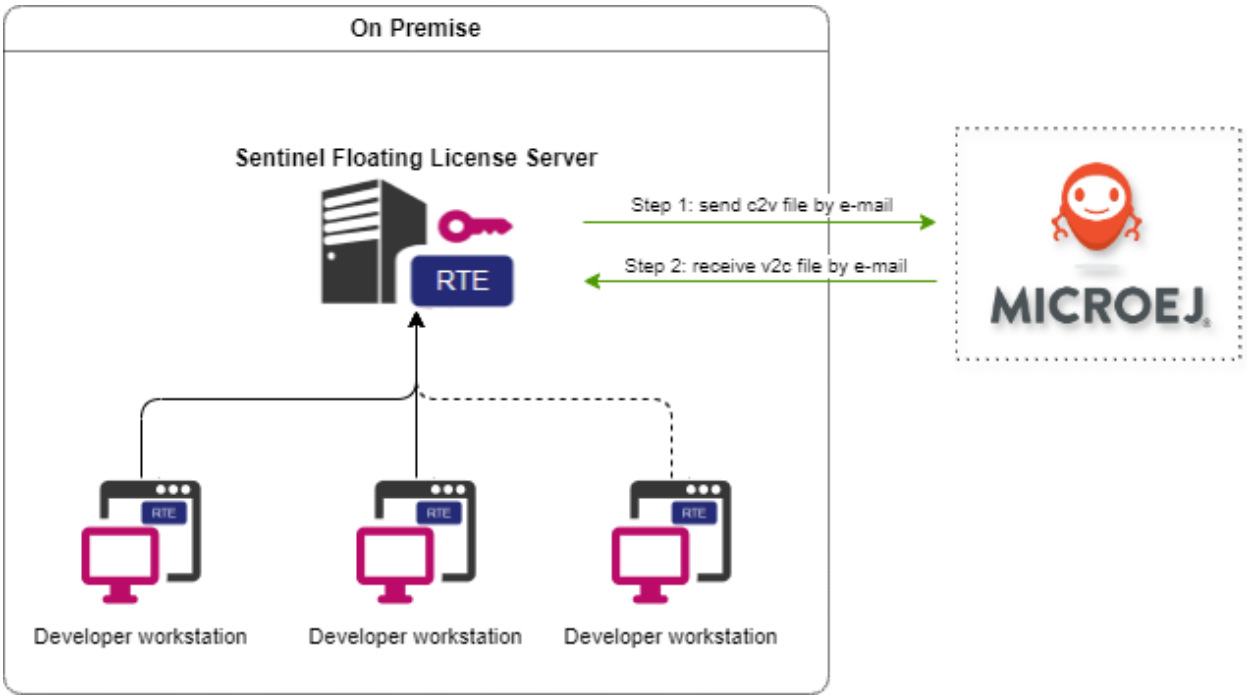
3.4.3 Sentinel License Management

In addition to single-workstation license based on a hardware dongle, MICROEJ SDK supports floating licenses based on Thales Sentinel LDK solution.

This chapter contains instructions that will allow you to setup Sentinel environment and activate your license.

There are two installation flows:

1. If you are a System Administrator, *setup the Sentinel Floating License Server* on a host machine.
2. If you are a Developer, proceed with the *setup of the Sentinel client for the SDK*.



Note: Floating License requires *Architecture 8.1.0* or higher (Production only).

Setup the Sentinel Floating License Server

This section contains instructions for the System Administrator to setup the Sentinel Floating License Server.

- Choose a machine to host the Sentinel Floating License Server. The host machine must be choosen with care, as its fingerprint will be required to generate the license file. Especially, you have to choose a host machine that is accessible through the network to all your developer workstations.

Install the Sentinel LDK Run-time Environment (RTE) for License Server

Sentinel LDK Run-time Environment enables your protected software to run by communicating with Sentinel protection keys.

First, download Sentinel_RTE_Installation-1.1.0.zip file. It contains installer for Windows, macOS and Linux.

Installation for Windows

- Get haspdinst_37102.exe file
- Type haspdinst_37102.exe -i in the command line. The installation or upgrade process is performed automatically. A message is displayed informing you that the Sentinel LDK Run-time Environment was successfully installed

---

Note: To uninstall Sentinel RTE, type haspdinst_37102.exe -r in the command line. A message is displayed informing you that the Sentinel LDK Run-time Environment was successfully removed.

Installation for Linux

Get aksusbd_37102-10.12.1.tar.gz file and extract it. The installation packages are in the directory pkg , as root enter the following command:

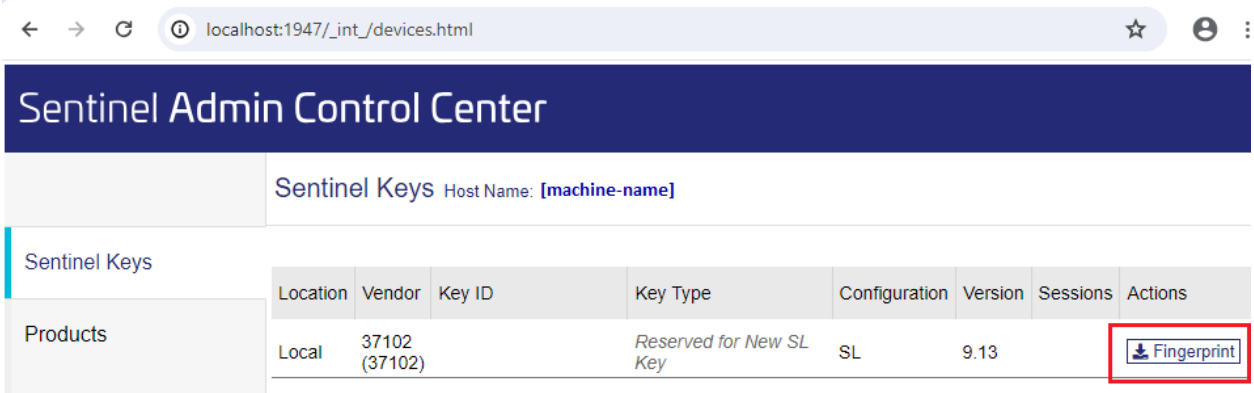
- For RedHat, SUSE, or CentOS 64-bit Intel systems: rpm -i aksusbd-10.12.1.x86_64.rpm
- For Ubuntu or Debian 64-bit Intel systems: dpkg -i aksusbd_10.12-1_amd64.deb
- Copy aksusbd-10.12.1/haspvlib_37102.so and aksusbd-10.12.1/haspvlib_x86_64_37102.so to /var/hasplm directory

---

Note: All install/uninstall commands must be executed with root rights. On Ubuntu, prefix the commands with the sudo command. On other Linux distributions, use the su utility to become root in the terminal window.

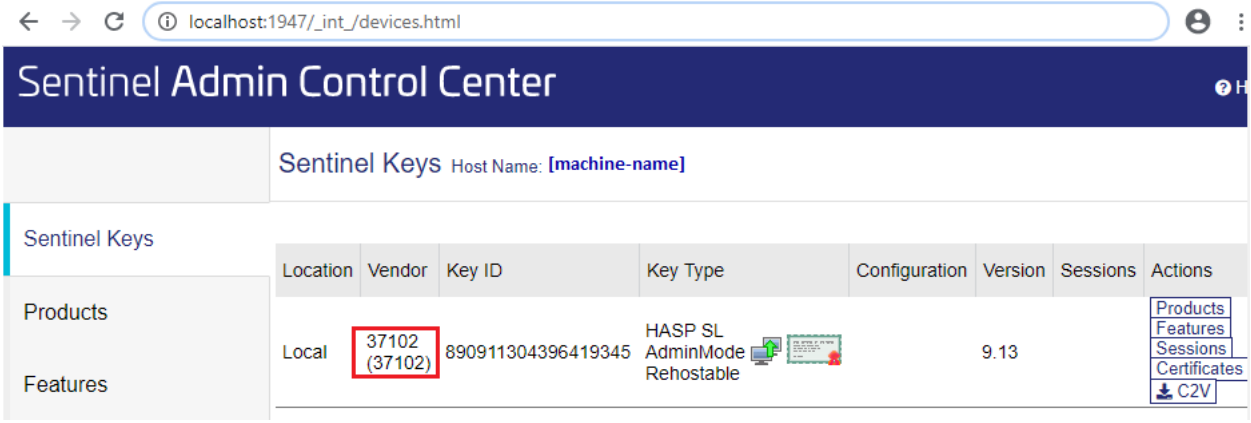
Configure the License Server

- On the host machine, open a web browser.
- Browse http://<server-ip-address>:1947 to open the Sentinel Admin Control Center.
- Go to Sentinel Keys tab and click on Fingerprint button to download the *.c2v file.

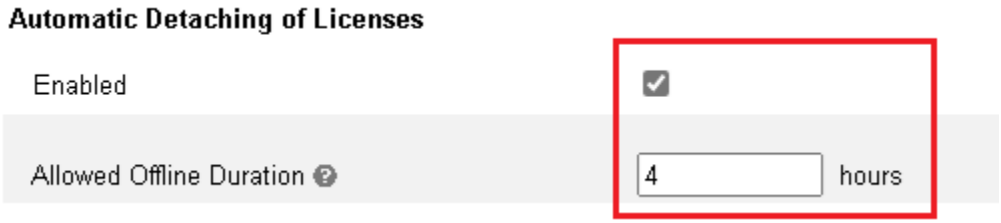


- The *.c2v file stands for *Customer to Vendor*. Send this file to your MicroEJ sales representative.
- Wait until MicroEJ prepares your license key. Then you will receive a .v2c file. The *.v2c file stands for *Vendor to Customer*.
- On the host machine, go back to the Sentinel Admin Control Center.
- Click on Update/Attach tab.
- Click on Select File… button and browse the .v2c file.
- Click on Apply File button.

In Sentinel Keys tab, you should see the successfully installed license key:



- Go to Configuration > Access from Remote Clients and check All licenses are accessible without need of identity. Save your change with the Submit button.
- Go to Configuration > Detachable Licenses > Automatic Detaching of Licenses, check Enabled and set Allowed Offline Duration to 4 hours.
- Click on Submit button.



- Configure the host machine to open the IP port 1947 for TCP, UDP, TCP6, UDP6.
- Optionally, you can set a printable name for the Sentinel Floating License Server that will be displayed on the Developer workstation license list.
  - Go to Configuration > Basic Settings tab.
  - Update the Machine Name text field (e.g. central-sentinel-server).

- Click on Submit button.

Your Sentinel Floating License Server is successfully configured. All you have to do is to share the host machine IP address to your MicroEJ Developers.

Setup the Developer Workstation

This section contains instructions for the MicroEJ Developer to setup its workstation in order to connect a Sentinel Floating License Server.

First, download Sentinel_RTE_Installation-1.1.0.zip file. It contains installer for Windows, macOS and Linux. Sentinel LDK Run-time Environment enables your protected software to run by communicating with Sentinel protection keys.

Installation for Windows

- Get haspdinst_37102.exe file
- Type haspdinst_37102.exe -i in the command line. The installation or upgrade process is performed automatically. A message is displayed informing you that the Sentinel LDK Run-time Environment was successfully installed
- Put MicroEJ_library\hasp_windows_x64_37102.dll file in the system folder (%SystemRoot%\system32) if you have administrator rights on your machine. Otherwise drop the hasp_windows_x64_37102.dll file beside java.exe executable of the Java Development Kit (JDK) used to run the SDK.

Note: To restart RTE, go to Services window and restart Sentinel LDK License Manager service. To uninstall Sentinel RTE, type haspdinst_37102.exe -r in the command line. A message is displayed informing you that the Sentinel LDK Run-time Environment was successfully removed.

Then you can continue with the *Remote Floating License Server* section.

Installation for Linux

Note: All install/uninstall commands must be executed with root rights. On Ubuntu, prefix the commands with the sudo command. On other Linux distributions, use the su utility to become root in the terminal window.

Get aksusbd_37102-10.12.1.tar.gz file and extract it. The installation packages are in the directory pkg , as root enter the following command:

- For RedHat, SUSE, or CentOS 64-bit Intel systems: rpm -i aksusbd-10.12.1.x86_64.rpm
- For Ubuntu or Debian 64-bit Intel systems: dpkg -i aksusbd_10.12-1_amd64.deb
- Copy aksusbd-10.12.1/haspvlib_37102.so and aksusbd-10.12.1/haspvlib_x86_64_37102.so to /var/hasplm directory
- Get MicroEJ_library/libhasp_linux_x86_64_37102.so file and copy it in a directory of your choice
- Set LD_LIBRARY_PATH variable with command export LD_LIBRARY_PATH=<directory_of_libhasp_file>:$LD_LIBRARY_PATH (just the directory, not with the file name). This modification has to be setup at session startup (e.g: using ~/.bashrc or ~/.profile files) to ensure that OS is properly configured before running the SDK.

Note: If you use the Sentinel RTE on WSL (Windows Subsystem for Linux): you can set the service to start on boot by creating the file /etc/wsl.conf and add these lines to it:

```
[boot]
command="service aksusbd start"
```

You can check the service status with the command `sudo service aksusbd status`

Then you can continue with the *Remote Floating License Server* section.

Installation for MacOS

- Get `Sentinel_Runtime_37102.tar` file
- In `SentinelRuntimeInstaller.framework/Versions/A/Resources/` double-click on `Sentinel_Runtime.pkg`
- Double-click the Install Sentinel Runtime Environment disk image icon. The installer wizard is launched
- Follow the instructions of the installer wizard until the installation is complete. The first time that you run Admin Control Center and submit configuration changes, `hasplmd` creates configuration files in `/private/etc/hasplm/`
- Get `MicroEJ_library/hasp_darwin_37102.dylib` file and copy it in a directory of your choice
- Set `DYLD_LIBRARY_PATH` variable with command `export DYLD_LIBRARY_PATH=<your_directory>:$DYLD_LIBRARY_PATH` . This modification has to be setup at session startup (e.g: using `.bashrc` file) to ensure that OS is properly configured before running the SDK.

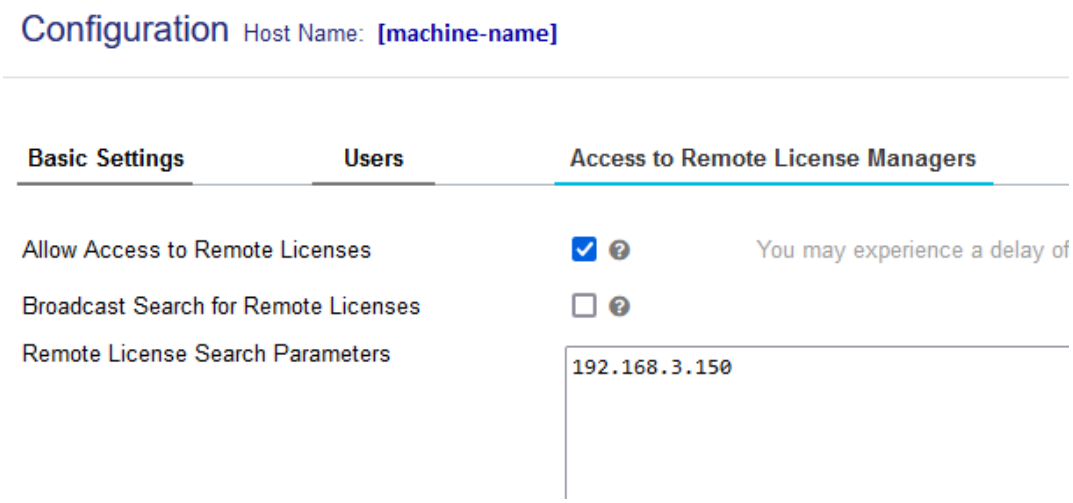
Note: To restart the daemons, on a terminal, go to `/Library/LaunchDaemons/` and launch `sudo launchctl load com.aladdin.aksusbd.plist` and `sudo launchctl load com.aladdin.hasplmd.plist`

Then you can continue with the *Remote Floating License Server* section.

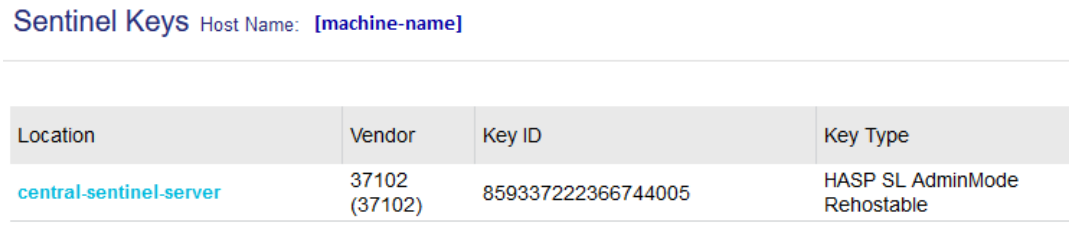
Add Remote Floating License Server

From Sentinel Admin Control Center

- On the developer workstation, open a web browser.
- Browse http://localhost:1947 to open the Sentinel Admin Control Center (if you work with WSL read note below).
- Go to `Configuration` > `Access to Remote License Managers` .
- Check `Allow Access to Remote Licenses` .
- Uncheck `Broadcast Search for Remote Licenses` .
- In `Remote License Search Parameters` , add the Floating License Server IP address that should have been shared by your System Administrator.



- Click on Submit button.
- Your computer should now have access to the licenses configured on the Floating License Server. In Sentinel Keys tab, you should see the license key provided by your Floating License Server (e.g. central-sentinel-server).



Note: If you use WSL (Windows Subsystem for Linux) localhost refers to your Windows localhost and not to your WSL instance. First, on WSL, create /etc/hasplm/hasplm.ini file and add accremote = 1 line to it. Then on WSL terminal, launch hostname -I command, copy the first IP address and use it instead of localhost in your browser (e.g. http://172.30.106.171:1947/).

From Command Line

- On Linux: update or create the file hasplm.ini in /etc/hasplm or hasp_37102.ini in ~/.hasplm (if you have not installed RTE as root).
- On Windows: edit %CommonProgramFiles(x86)%\Aladdin Shared\HASP\hasplm.ini file.

Then add to the .ini file the following lines:

```
[REMOTE]
broadcastsearch = 0
serversearchinterval = 30
serveraddr = <license_server_IP>
```

- Restart the service.

Running in a container

If you want to configure a CI (Continuous integration) runner you can follow one of these two solutions:

- Either create a Docker image with the RTE installed inside, see *Installation for Linux* section.
- Or install and configure the RTE on the host and run the Docker container with these options:

```
-v /var/hasplm:/var/hasplm:ro -v /home/<host_user>/.hasplm:/home/<container_user>/.
↪hasplm:ro -e LD_LIBRARY_PATH=/var/hasplm
```

Runtime Installation Instructions and Troubleshooting

Check Activation with the Command Line Tool

To verify access to the Sentinel license on the workstation where the SDK executes, run the debug tool as following:

1. Open a terminal
2. Change directory to a Production VEE Port
3. Execute the command:

```
java -Djava.library.path=resources/os/[OS_NAME] -jar licenseManager/licenseManagerProduct.
↪jar
```

with `OS_NAME` set to `Windows64` for Windows OS, `Linux64` for Linux OS, `Mac` for macOS x86_64 (Intel chip) or `MacA64` for macOS aarch64 (M1 chip).

Note: On Windows, use `Command Prompt` terminal to run the command, instead of `PowerShell`. Also, specify the absolute path to the required resources to avoid errors, for example:

```
java -Djava.library.path="C:\MY_PATH\my-app\build\vee\resources\os\Windows64" -jar "C:\MY_PATH\my-
↪app\build\vee\licenseManager\licenseManagerKeyHardware.jar"
```

If your Sentinel license has been properly activated, you should get the following output:

```
[DEBUG] ===== MicroEJ Sentinel Debug Tool =====
[DEBUG] => Detected Sentinel License Key ID: XXXXXXXXXXXXXXX.
[DEBUG] => Detected MicroEJ License valid until YYYY-MM-DD.
[DEBUG] ===== SUCCESS =====
```

Troubleshooting

Sentinel API dynamic library not found (code 400)

The following error occurred when the library `hasp_[os]_37102.[ext]` has not been found. Please refer to *Setup the Developer Workstation*.

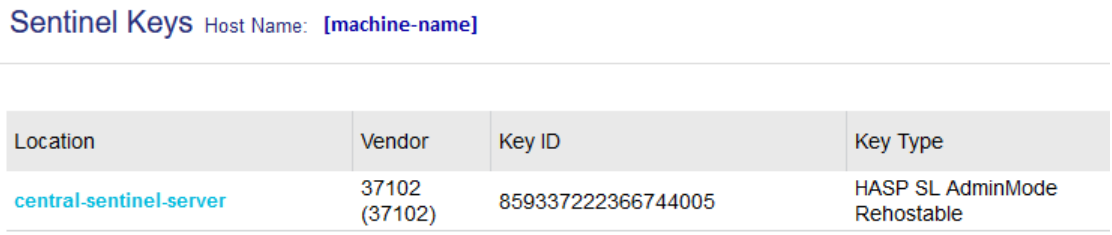
Specifically, if you are on Linux:

- check the library is readable with the command `file libhasp_linux_x86_64_37102.so`.

- check the *LD_LIBRARY_PATH* environment variable is properly set before launching MICROEJ SDK. If MICROEJ SDK is launched from a desktop application, it must have been set in the ~/.profile file.

Sentinel key not found (code 7)

The following error occurred when there is no Sentinel license available. Go to http://localhost:1947/int/devices.html and check your Sentinel licenses. You should see at least one installed license key:



Make sure you correctly configured the access to the Sentinel Floating License Server. Please refer to *Add Remote Floating License Server*.

No Administrator Privileges on Developer Workstation

Sentinel LDK RTE installation requires administrator privileges and facilitates the setup of the network server. However, it is not necessary to be installed on the developer workstation in case of floating licenses. See this documentation for more details. If you are in such situation, please can contact our support team.

Error: Could not find or load main class .library.path=resources.os.Windows64

The following error occurs when running the command line from PowerShell on Windows. Use Command Prompt instead.

3.4.4 Scope and Limitations

The SDK 6 covers all the SDK 5 features, except:

- Build of Add-On Processors.
- Launch of some MicroEJ tools, such as the Kernel Metadata Generator.

If you need these features, you have to use *the SDK 5*.

It is also important to note the following SDK 5 / SDK 6 compatibility rules:

- Libraries produced with SDK 6 can be fetched by SDK 5 projects.
- Using SDK 6 VEE Port in SDK 5 Applications and Libraries is unsupported.
- Architecture 8.1 is recommended to have full support on the SDK 6 features (especially to Debug with IntelliJ IDEA and Android Studio).
- SDK 6 Virtual Device cannot be used by SDK 5 Applications.
- SDK 5 Virtual Device cannot be used to build the Virtual Device of a SDK 6 Application (augmented Virtual Device).

- Using SDK 6 Module Repository in SDK 5 Projects is unsupported.

When *the VEE Port transitive resolution* is enabled, the VEE Ports and Kernels built with SDK 6 prior to 1.3.0 or SDK 5 are not supported, except if they are available locally in a directory or as an archive. It is highly recommended to update your VEE Port or Kernel if necessary, as the feature will be enabled by default in the next SDK 6 major version.

3.4.5 Create a Project

This chapter explains the different ways to create a new Application, Library or Mock project. If you want to create a VEE Port project, refer to the *Create a VEE Port* section.

Note: The different project creation systems do not produce exactly the same project content and structure. Especially, the IntelliJ IDEA wizard produces a simple project whereas the Android Studio, Command Line Interface and Eclipse wizards create multi-projects builds. Both structures (single and multi projects) can be used, the recommended one depends on the context (components, size of the project, …). Refer to the official Gradle documentation for more information.

IntelliJ IDEA

Android Studio

Eclipse

Visual Studio Code

Command Line Interface

The creation of a project with IntelliJ IDEA is done as follows:

- Click on `File` > `New` > `Project…` .
- Select `MicroEJ` in `Generators` list on the left panel.
- Fill the name of the project in the `Name` field.
- Select the location of the project in the `Location` field.
- Select the project type. If there is no button for your project type, click on `Other` button and select it in the drop-down list.
- If you selected `Application` project type, you can check `This is a kernel application` checkbox if your Application is a Kernel.
- Fill the version of the artifact to publish in the `Version` field.
- Fill the group of the artifact to publish in the `Group` field.
- Fill the name of the artifact to publish in the `Artifact` field.
- Select the JVM used by Gradle in the `JDK` combobox.
- Check the `Add sample code` checkbox.
- Click on `Create` button.

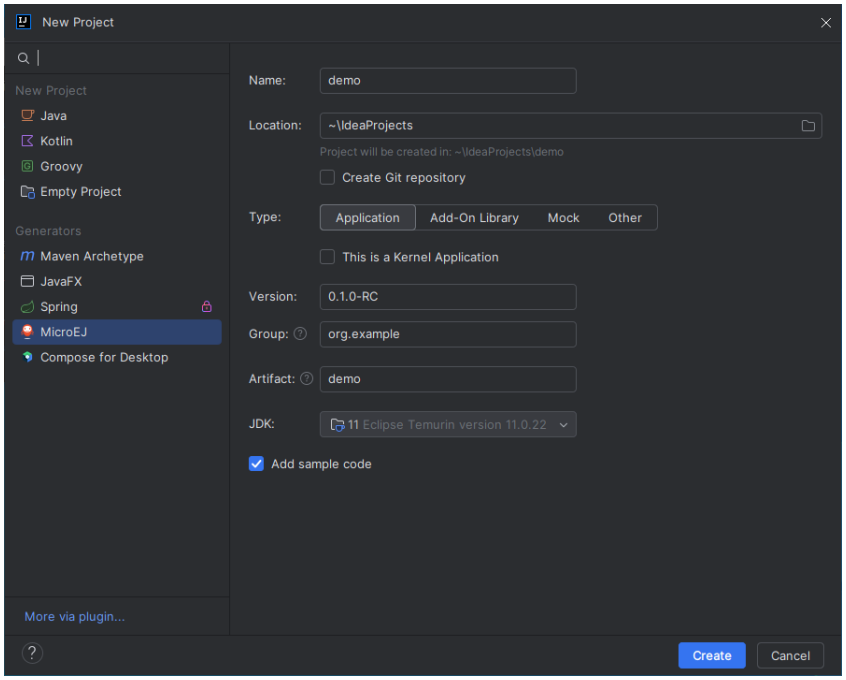


Fig. 27: Project Creation in IntelliJ IDEA

Note: The Gradle project created by the wizard uses Gradle Wrapper with Gradle version 8.10.2 . Refer to the *Gradle Wrapper* section for more information.

Note: By default, IntelliJ IDEA automatically saves any file change, but requires the user to explicitly trigger the reload of a Gradle project when its configuration has changed. Therefore, when the configuration of a Gradle project has been updated, you have to click on the reload icon button which appears on the right of the editor:

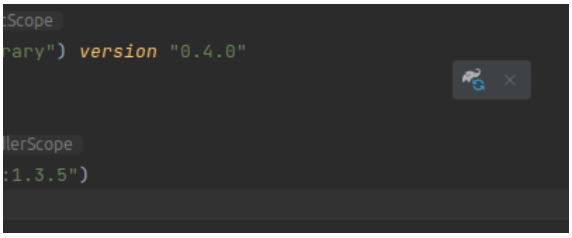


Fig. 28: Gradle Project reload in IntelliJ IDEA

You can also configure IntelliJ IDEA to automatically reload a Gradle project after a change. Refer to the *How To Automatically reload a Gradle project* section for more information.

Warning: When reloading your Gradle project, the build can fail if the SDK EULA has not been accepted. In that case, you must set the ACCEPT_MICROEJ_SDK_EULA_V3_1C environment variable to YES and restart IntelliJ IDEA. For more information about SDK EULA, refer to the *Licenses* chapter.

When the Gradle project is loaded, it should compile successfully, without any error. You can then learn *how to launch the build of the project*, or *how to run it on the Simulator* in the case of an Application.

The creation of a project with Android Studio is done as follows:

- Click on `File` > `New` > `Project…` .
- Select `Generic` > `New MicroEJ project` .

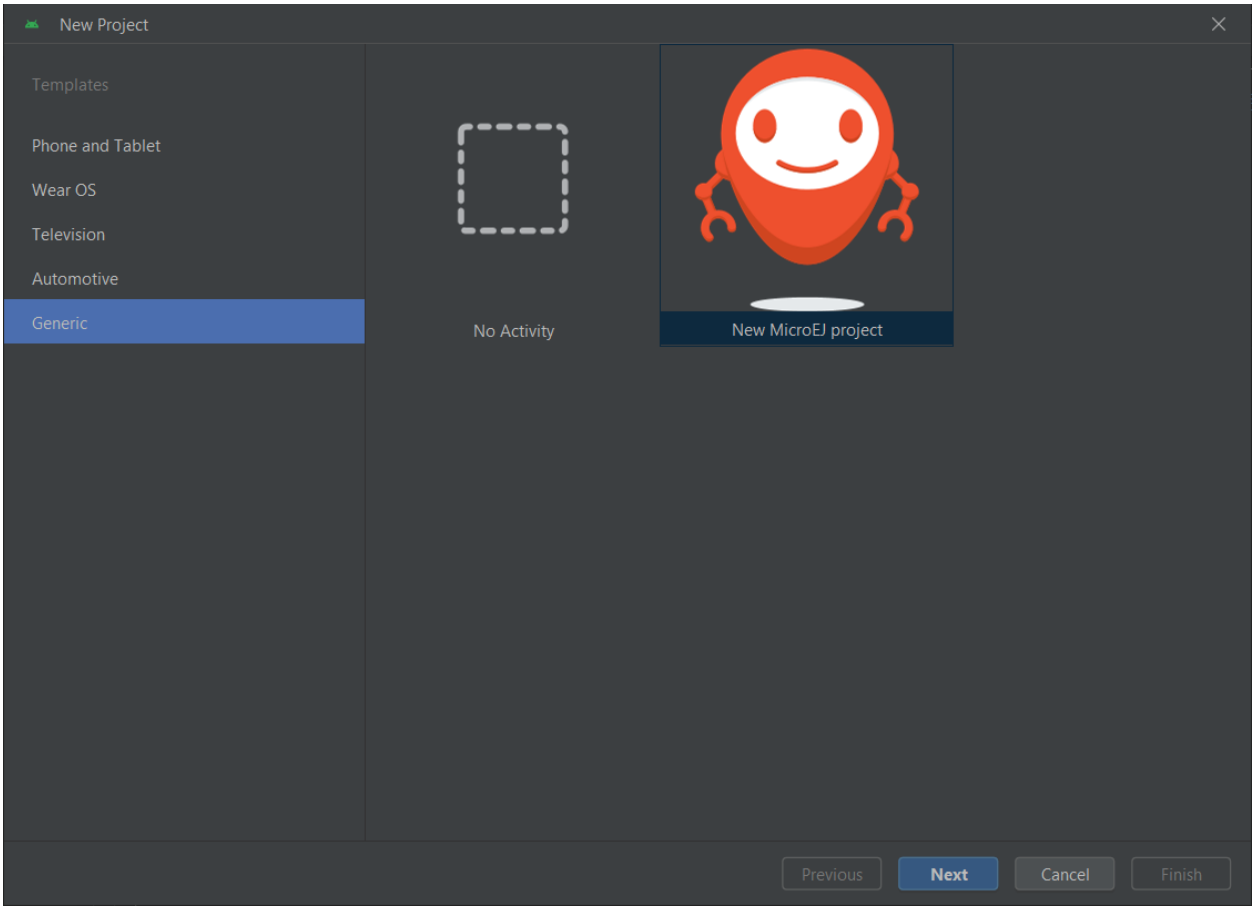


Fig. 29: Project Creation in Android Studio

- Click on the `Next` button.
- Fill the name of the project in the `Name` field.
- Fill the package name of the project in the `Package name` field.
- Select the location of the project in the `Save location` field.
- Keep the default Android SDK in the `Minimum SDK` field.
- Select `Kotlin` for the `Build configuration language` field.

Note: Groovy build script DSL is not officially supported by the SDK, so the project created by the Wizard uses Kotlin regardless of the language selected by the user.

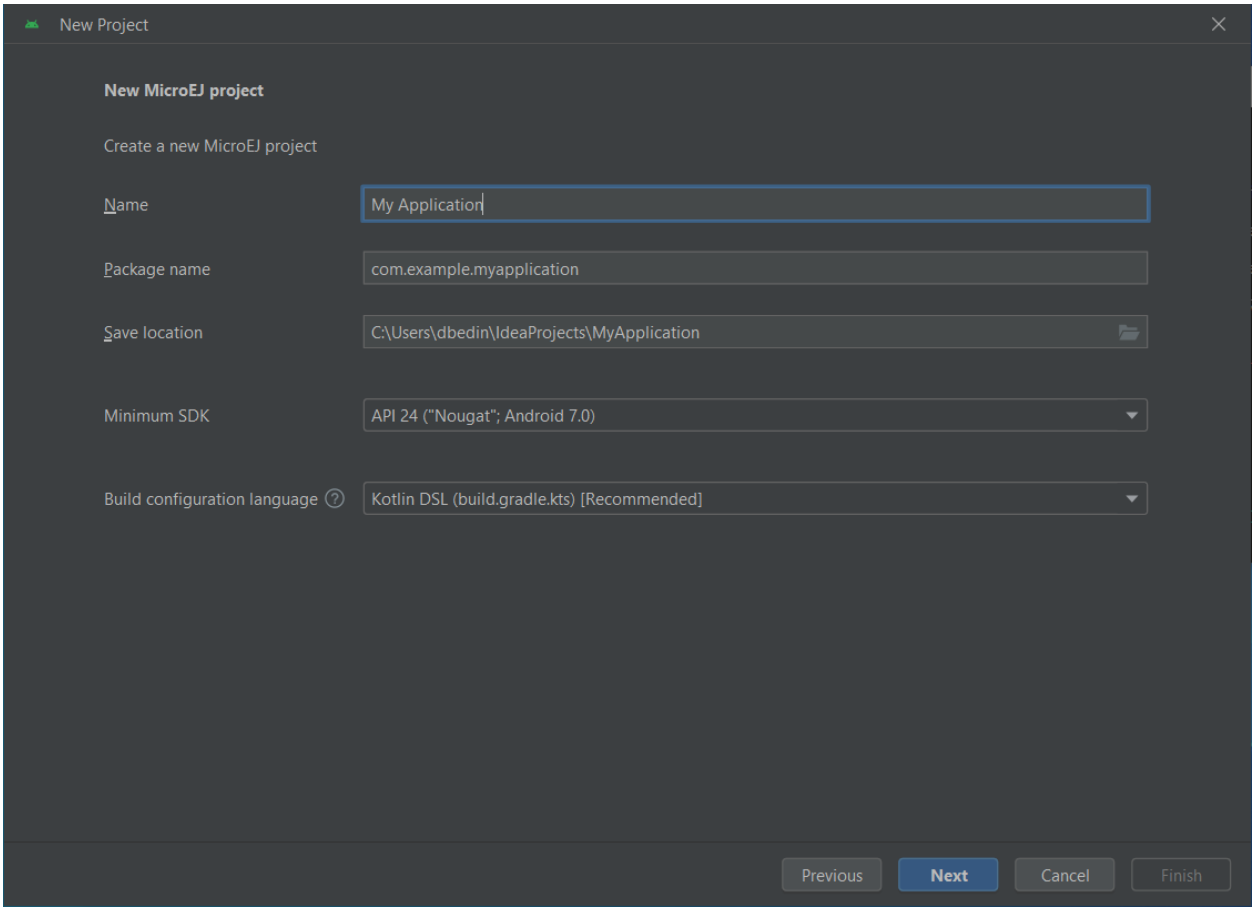


Fig. 30: Project Creation in Android Studio

- Click on `Next` button.
- Fill the group of the artifact to publish in the `Group` field.
- Fill the version of the artifact to publish in the `Version` field.
- Select the project type in the drop-down list.
- If you selected `Application` project type, you can check `This is a kernel application` checkbox if your Application is a Kernel.
- Click on `Finish` button.

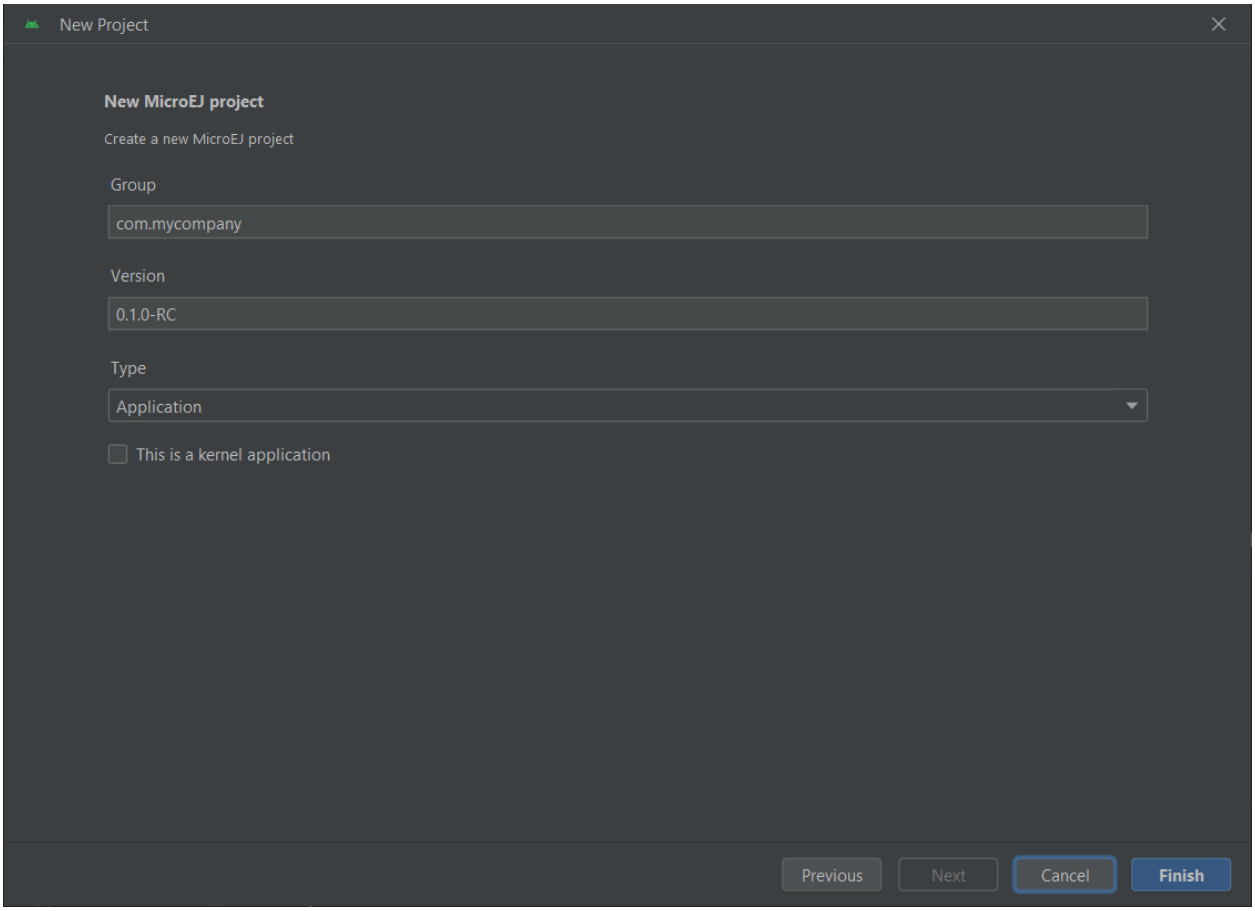


Fig. 31: Project Creation in Android Studio

- Change the view from `Android` to `Project` in the selectbox at the top of the project's files tree:

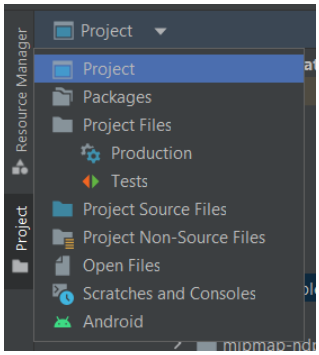


Fig. 32: Project View in Android Studio

Note: If you do not use the last version of Android Studio, make sure that Gradle Wrapper uses Gradle 8 , and at least version 8.6 (the version 9 is not supported yet). Refer to the *Gradle Wrapper* section for more information.

The project created by the wizard is a multi-project with a single subproject (named app). The type of this subproject is the type that has previously been chosen.

Note: By default, Android Studio automatically saves any file change, but requires the user to explicitly trigger the reload of a Gradle project when its configuration has changed. Therefore, when the configuration of a Gradle project has been updated, you have to click on the `Sync Now` action which appears on the top-right of the editor:

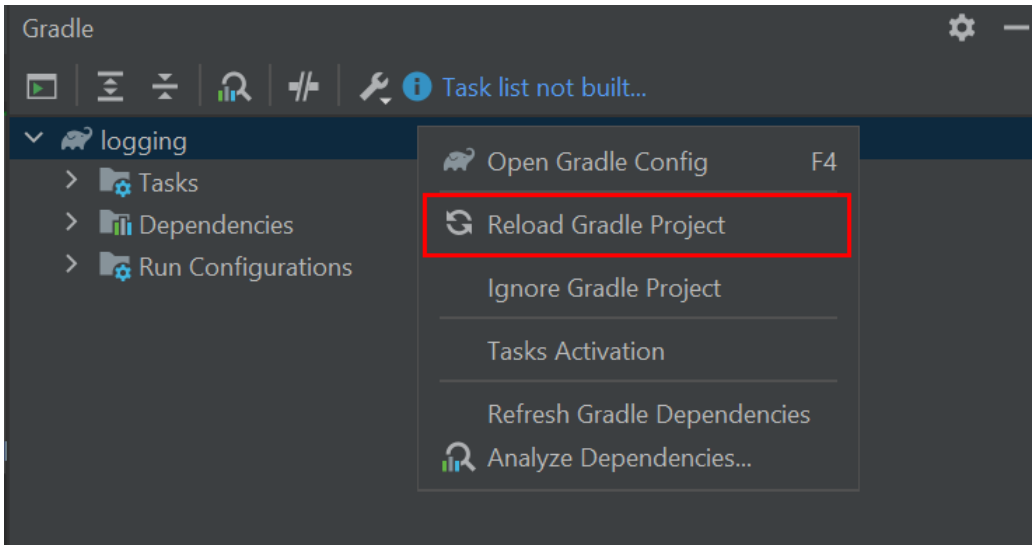


Fig. 33: Gradle Project reload in Android Studio

You can also configure Android Studio to automatically reload a Gradle project after a change. Refer to the *How To Automatically reload a Gradle project* section for more information.

Warning: When reloading your Gradle project, the build can fail if the SDK EULA has not been accepted. In that case, you must set the `ACCEPT_MICROEJ_SDK_EULA_V3_1C` environment variable to `YES` and restart Android Studio. For more information about SDK EULA, refer to the *Licenses* chapter.

When the Gradle project has been reloaded, it should compile successfully, without any error. You can then learn *how to launch the build of the project*, or *how to run it on the Simulator* in the case of an Application.

The creation of a project with Eclipse is done as follows:

- Click on `File` > `New` > `Project…` .
- Select `MicroEJ` > `<Type> project` depending on your project type and click on the `Next` button.

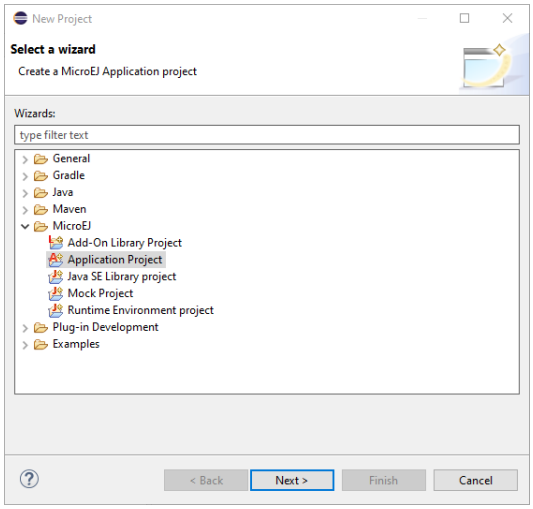


Fig. 34: Project Type Selection in Eclipse

- Fill the name of the project in the `Name` field, for example My Project .
- Fill the group of the artifact to publish in the `Organization` field.
- Fill the name of the artifact to publish in the `Module` field.
- Fill the version of the artifact to publish in the `Revision` field.
- If you selected `Application` module type, you can check `This is a kernel application` checkbox if your Application is a Kernel.
- Click on `Finish` button.

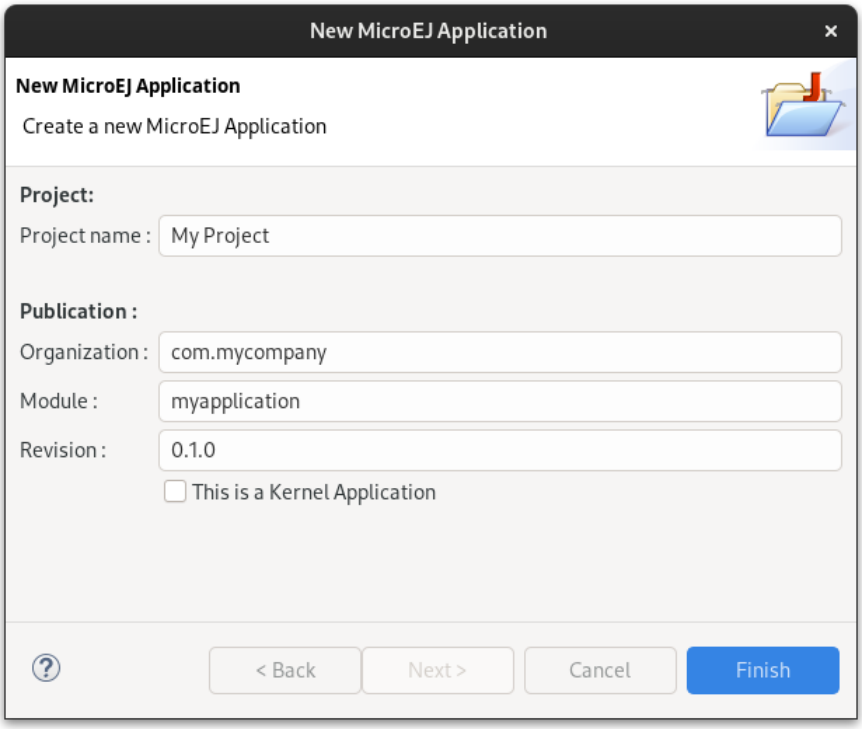


Fig. 35: Application Creation in Eclipse

Note: The Gradle project created by the wizard uses Gradle Wrapper with Gradle version 8.10.2 . Refer to the *Gradle Wrapper* section for more information.

Warning: When reloading your Gradle project, the build can fail if the SDK EULA has not been accepted. In that case, you must set the ACCEPT_MICROEJ_SDK_EULA_V3_1C environment variable to YES and restart Eclipse. For more information about SDK EULA, refer to the *Licenses* chapter.

When the Gradle project is loaded, it should compile successfully, without any error. You can then learn *how to launch the build of the project*, or *how to run it on the Simulator* in the case of an Application.

The creation of a project with Visual Studio Code is done as follows:

- Select View > Command Palette… .
- Run the Git: Clone command in the Command Palette.

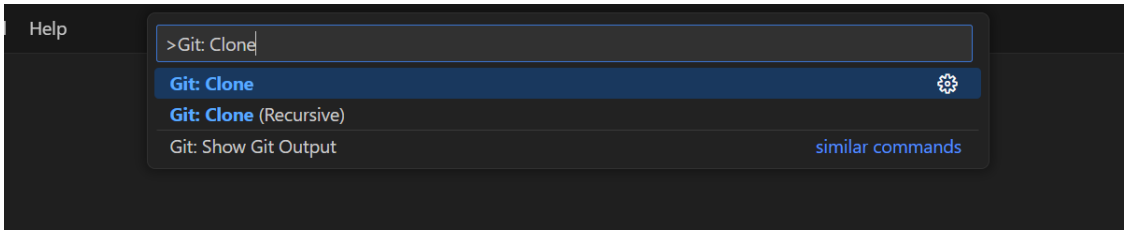


Fig. 36: Command Palette in VS Code

- Depending on the type of your project, fill the URI of the corresponding Github template repository in the Search Bar. The available templates are:
  - Application Project Template
  - Add-On Library Project Template
  - Mock Project Template
  - Runtime Environment Template
  - Java SE Library Template
- Click on Clone from URL .

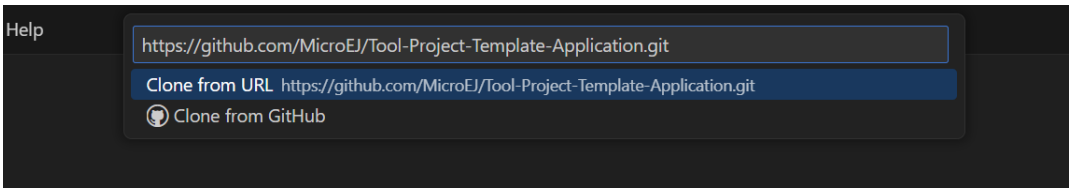


Fig. 37: Search Bar in VS Code

- In the upcoming popup, choose a folder and click on the Select as Repository Destination button.
- When the Gradle project is loaded, select Terminal > New Terminal .
- In the integrated terminal, run the following command at the root of the project to remove the Git Repository:

Windows

Linux/macOS

```
rm -r -Force .git*
```

```
rm -rf .git*
```

- Rename the project and change its group and version in the build.gradle.kts build script.

The creation of a project via Command Line Interface is done as follows:

- Depending on the type of your project, retrieve the URI of the corresponding Github template repository. The available templates are:
  - Application Project Template
  - Add-On Library Project Template
  - Mock Project Template
  - Runtime Environment Template
  - Java SE Library Template
- Clone the repository:

```
git clone <template-repository>
```

- Remove the Git Repository from the project:

Windows

Linux/macOS

```
rm -r -Force .git*
```

```
rm -rf .git*
```

- Rename the project and change its group and version in the `build.gradle.kts` build script.

Configure a Project

The SDK allows to build several types of modules. Each type has its own Gradle plugin and configuration options. Refer to the module type you want to build to configure your project:

- *Application*
- *Library*
- *Mock*
- *Java SE Library*
- *Runtime Environment*

Refer to the page *Module Natures* for a complete list of the available MicroEJ natures and their corresponding plugins.

Application Project

The required configuration elements to define for an Application Project are:

- The Application Entry Point. It is defined by the `applicationEntryPoint` property in the `microej` configuration block of the `build.gradle.kts` file. It must be set to the Full Qualified Name of the Application main class, for example:

  ```
  microej {
    applicationEntryPoint = "com.mycompany.Main"
  }
  ```

  The Project Wizards define a default value, adapt it if you change the main class of your Application.

- The VEE to use to build and run the Application. It is defined using to the `microejVee` configuration in the `dependencies` block of the `build.gradle.kts` file:

  ```
  dependencies {
     microejVee("com.mycompany:vee-port:1.0.0")
  }
  ```

  Refer to the *Select a VEE Port* page to learn more about the different ways to define the VEE.

Refer to the *Application Developer Guide* to learn more on Applications.

Library Project

A Library project does not require any specific configuration. Refer to the *Libraries* and *Foundation Libraries* pages to learn more on Libraries.

Mock

A Mock project does not require any specific configuration. Refer to the *VEE Porting Guide Mock chapter* page to learn more on how to develop a Mock.

Java SE Library Project

A Java SE Library project does not require any specific configuration.

Runtime Environment Project

A Runtime Environment project does not require any specific configuration. Refer to the *Runtime Environment* page to learn more on how to develop a Runtime Environment.

Create a subproject in an existing project

This section explains the different ways to add a module to an existing project.

> **Warning:** If you want to add a MicroEJ module to a non MicroEJ project, for example an Android project, you must *configure the repositories* before creating the module. If the repositories used by your project are centralized in the `settings.gradle.kts` file of the project, the MicroEJ repositories defined in this file must be added to your `settings.gradle.kts` file.

IntelliJ IDEA

Android Studio

Eclipse

Visual Studio Code

The creation of a module with IntelliJ IDEA is done as follows:

- Click on `File` > `New` > `Module...` .
- Select `MicroEJ` in `Generators` list on the left panel.
- Fill the name of the module in the `Name` field.
- Select the location of the module in the `Location` field.
- Select the module type. If there is no button for your module type, click on `Other` button and select it in the drop-down list.
- If you selected `Application` module type, you can check `This is a kernel application` checkbox if your Application is a Kernel.

- Fill the version of the artifact to publish in the `Version` field.
- Fill the group of the artifact to publish in the `Group` field.
- Fill the name of the artifact to publish in the `Artifact` field.
- Select the JVM used by Gradle in the `JDK` combobox.
- Check the `Add sample code` checkbox.
- Click on `Create` button.

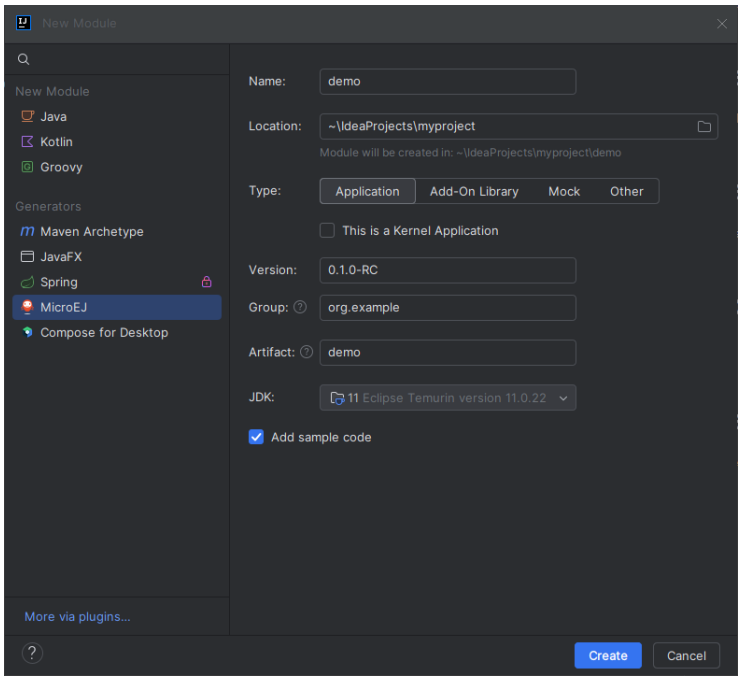


Fig. 38: Module Creation in IntelliJ IDEA

The creation of a module with Android Studio is done as follows:

- Click on `File` > `New` > `New Module...` .
- Select `MicroEJ Module` in `Templates` list on the left panel.
- Fill the name of the module in the `Name` field.
- Fill the group of the artifact to publish in the `Group` field.
- Fill the version of the artifact to publish in the `Version` field.
- Select the module type in the drop-down list.
- If you selected `Application` module type, you can check `This is a kernel application` checkbox if your Application is a Kernel.
- Click on `Finish` button.

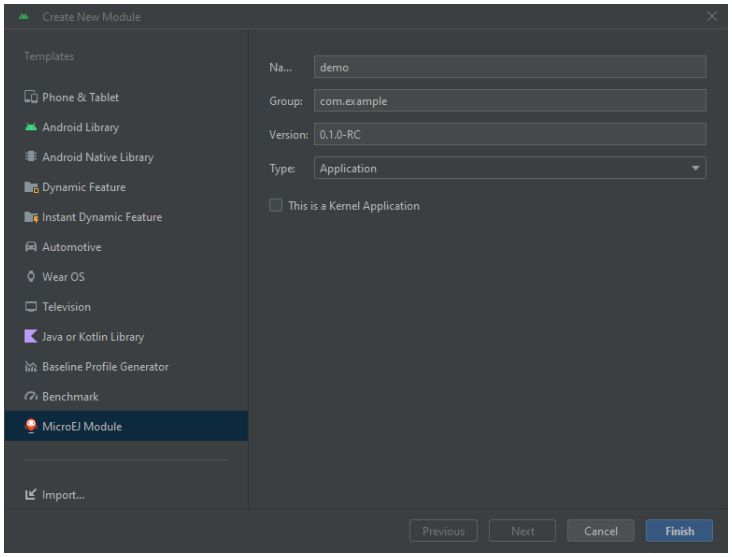


Fig. 39: Module Creation in Android Studio

The creation of a module with Eclipse is done as follows:

- Right-click on your project and click on `New` > `Folder` .
- Select your project as parent folder.
- Fill the name of the module in the `Folder name` field.
- Click on `Finish` button.

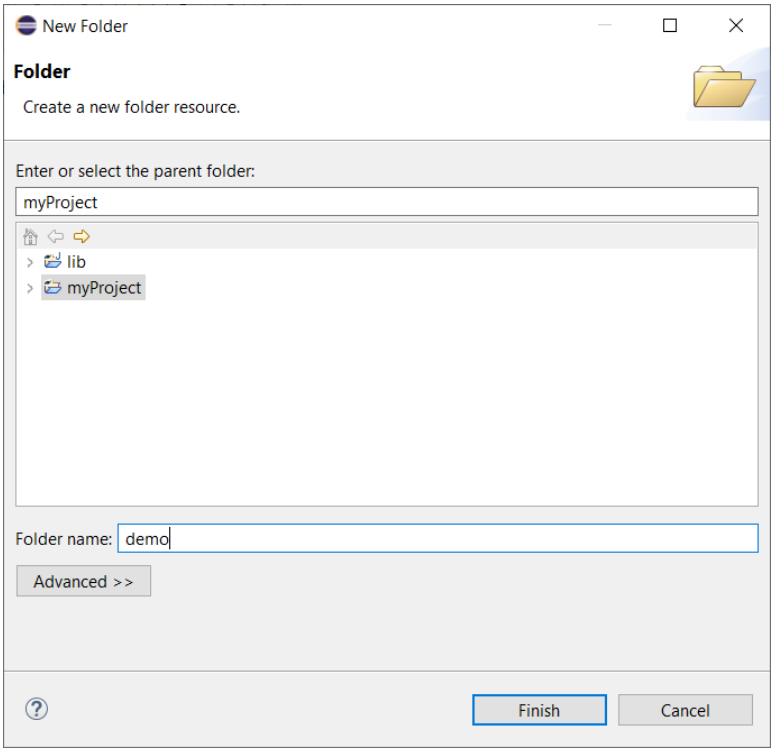


Fig. 40: Module Creation in Eclipse

- Right-click on your newly created folder and click on `New` > `File` .
- Enter `build.gradle.kts` in the `File name` field.
- Click on `Finish` button and open the `build.gradle.kts` file.
- *Configure your module* depending on its type.
- Declare the dependencies required by your project in the `dependencies` block. For example:

```
dependencies {
    implementation("ej.api:edc:1.3.7")
}
```

- Open the `settings.gradle.kts` file of your project and add the following content:

```
include("<module_name>")
```

Note: By default, Eclipse requires the user to explicitly trigger the reload of a Gradle project when its content has changed. Therefore, when the content of a Gradle project has been updated, you have to right-click on the project, then click on `Gradle` and `Refresh Gradle Project` :

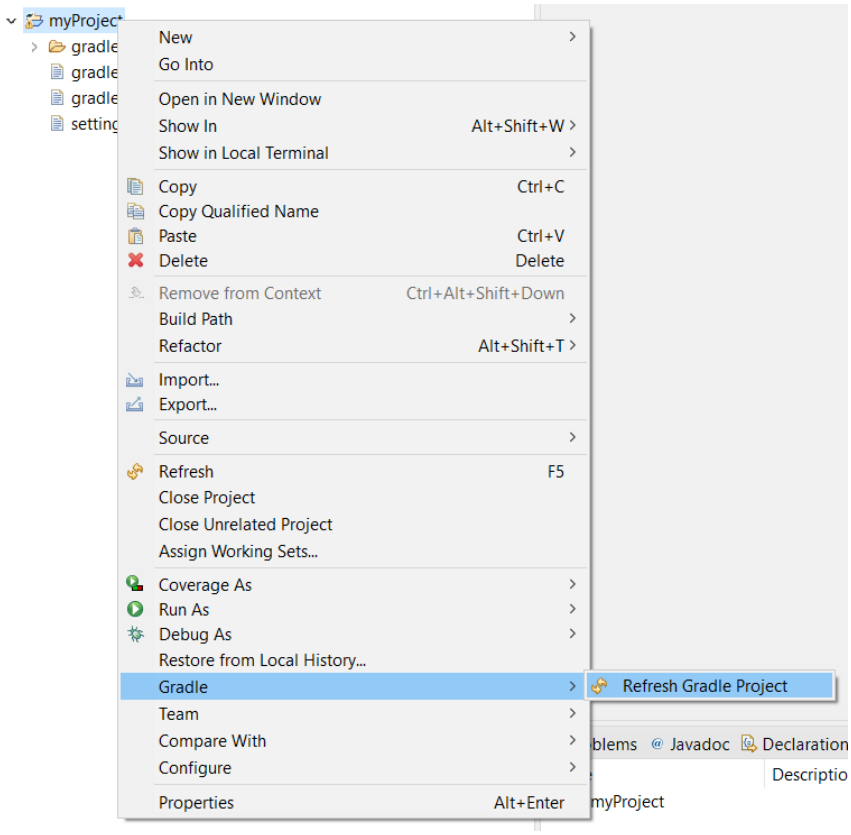


Fig. 41: Gradle Project reload in Eclipse

You can also configure Eclipse to automatically reload a Gradle project after a change. Refer to the *How To Automatically reload a Gradle project* section for more information.

- Right-click on the newly created module and click on `New` > `Source Folder` .
- Enter `src/main/java` in the `Folder name` field.
- Click on `Finish` button.

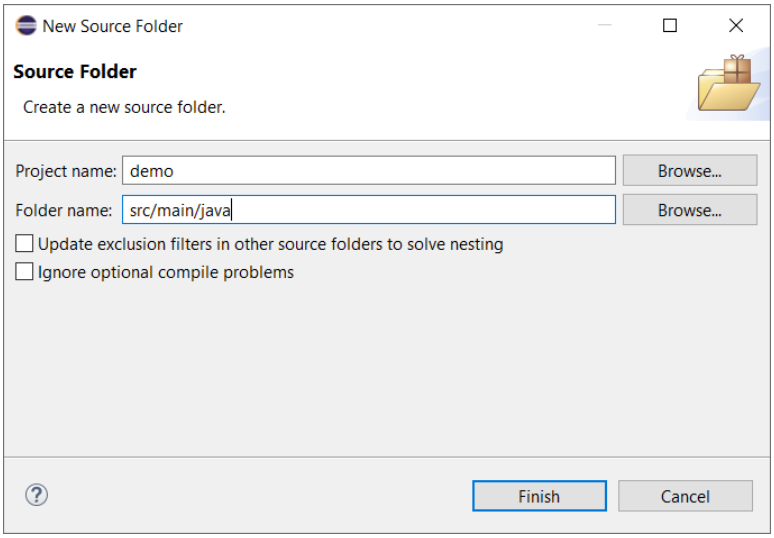


Fig. 42: Source Folder Creation in Eclipse

- Follow the same steps to create the src/main/resources , src/test/java and src/test/resources folders.

The creation of a module with Visual Studio Code is done as follows:

- If the Explorer view is not already opened, open it by selecting View > Open View… > Explorer .
- Right-click in the Explorer view and click on New Folder… .
- Fill the name of the module and press Enter .
- Create a new build.gradle.kts file by right-clicking on your newly created folder and clicking on New File… .
- *Configure your module* depending on its type.
- Declare the dependencies required by your module in the dependencies block. For example:

```
dependencies {
    implementation("ej.api:edc:1.3.7")
}
```

- Open the settings.gradle.kts file of your project and add the following content:

```
include("<module_name>")
```

- Create the src/main/java folder by right-clicking on the newly created module and clicking on New Folder… .
- Follow the same steps to create the src/main/resources , src/test/java and src/test/resources folders.

Gradle Wrapper

It is recommended to use the Gradle Wrapper to execute a build. The Wrapper is a script that ensures that the required version of Gradle is downloaded and used during the build of a project.

When creating a project following one of the project creation systems described in the *Create a Project* section, the Wrapper files are automatically generated in the `gradle/wrapper` folder of the project. It is also possible to add the Wrapper to an existing project. This requires to install the Gradle distribution, then to execute the `wrapper` task with:

```
gradle wrapper
```

The Gradle version used by the project can then be updated in the `gradle/wrapper/gradle-wrapper.properties` file. The SDK requires Gradle `8`, and at least version `8.6` (the version `9` is not supported yet):

```
distributionUrl=https\://services.gradle.org/distributions/gradle-8.6-bin.zip
```

To use the Wrapper during a build, use `gradlew` or `./gradlew` depending on your OS instead of `gradle` in the command line:

Windows

Linux

gradlew build

./gradlew build

In the following chapters of the documentation, the Linux command `./gradlew` is used in all examples to execute a build.

Refer to the official Gradle documentation for more information about the Wrapper.

3.4.6 Import a Project

This chapter explains how to import a project in an IDE.

IntelliJ IDEA

Android Studio

Eclipse

In order to import an existing Gradle project in IntelliJ IDEA, follow the following steps:

- If you are in the Welcome Screen, click on the `Open` button. Otherwise click either on `File` > `Open…` or on `File` > `New` > `Project From Existing Sources…`.
- Select the root directory of the project and click on the `OK` button.

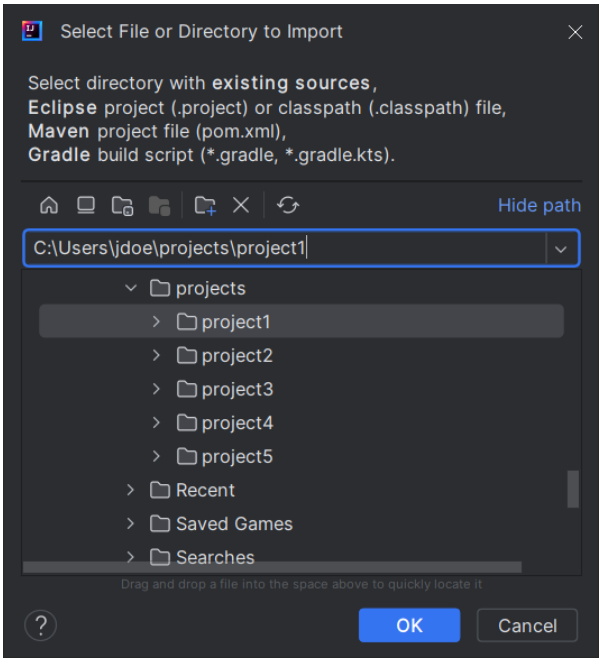


Fig. 43: Project Import in IntelliJ IDEA

- If you are asked to choose a project model, select `Gradle`.
- Click on the `Create` button.

The Gradle project should now be opened in IntelliJ IDEA.

In order to import an existing Gradle project in Android Studio, follow the following steps:

- If you are in the Welcome Screen, click on the `Open` button. Otherwise click either on `File` > `Open…` or on `File` > `Import Project…`.
- Select the root directory of the project and click on the `OK` button.

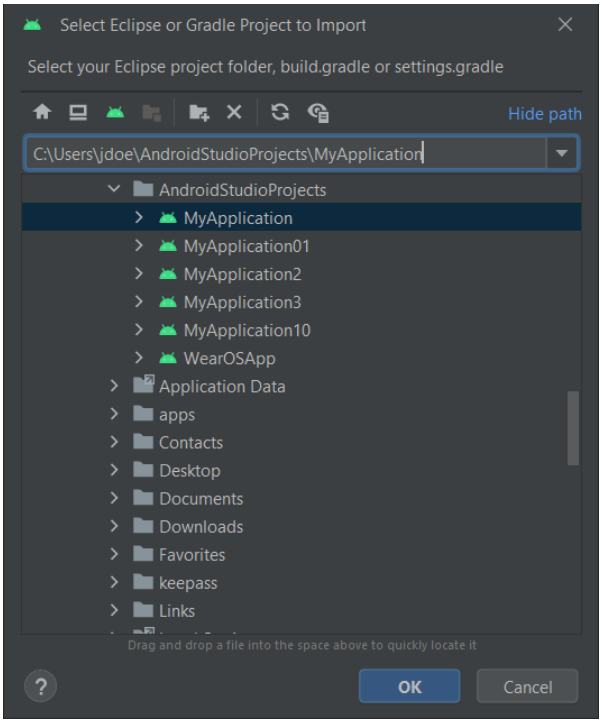


Fig. 44: Project Import in Android Studio

The Gradle project should now be opened in Android Studio.

In order to import an existing Gradle project in Eclipse, follow these steps:

- Click on File > Import… .
- Select the project type Gradle > Existing Gradle Project and click on the Next button.

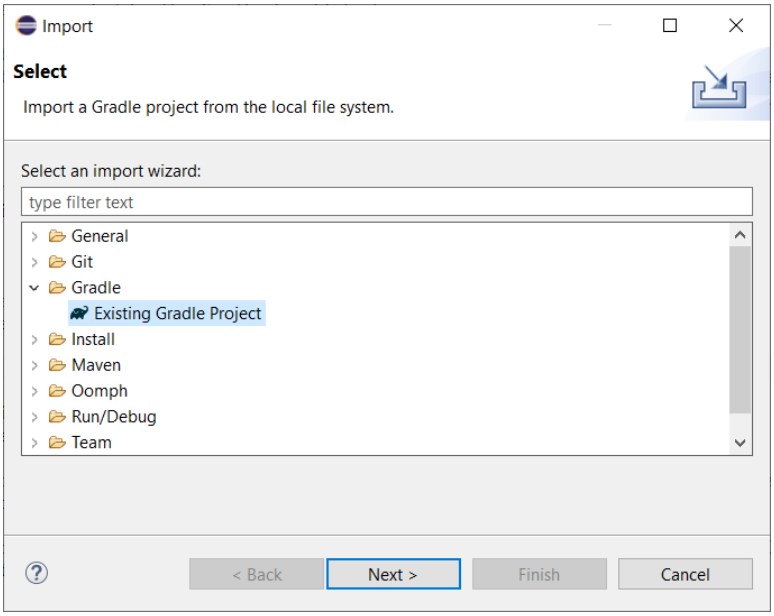


Fig. 45: Project Type Selection in Eclipse

- Select the root directory of the project.

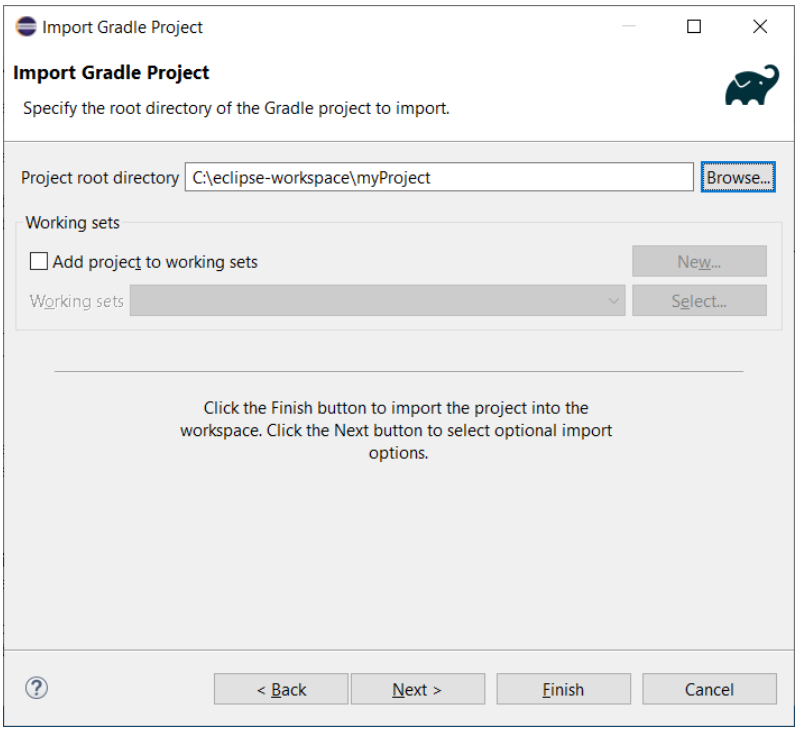


Fig. 46: Project root folder in Eclipse

- Click on the Next button and finally on the Finish button.

The Gradle project should now be opened in Eclipse.

3.4.7 Select a VEE Port

Building or running an Application or a Testsuite with the SDK requires a VEE Port.

Use one of the following available options to provide it to your project.

Note: Declaring a VEE Port in project dependencies only applies to the current project. Especially when configuring the VEE Port to test a library project, application projects depending on this library will not "see" this test VEE Port, they must configure a VEE Port on their own and are free to use a different one.

Using a Module Dependency

When your VEE Port is published in an artifact repository, you can define the VEE Port by declaring a module dependency in the build.gradle.kts file, with the microejVee configuration:

```
dependencies {
   microejVee("com.mycompany:myveeport:1.0.0")
}
```

and by *setting the Architecture Usage* of the VEE Port.

This is generally the case for developers focused on Application and Library development. They don't have (and don't need) the VEE Port project locally, they only need to use it.

Note: For dependencies stored in an Ivy repository, Gradle fetches them with the configuration `default`. If several artifacts are published with this configuration, the build will fail because it doesn't know which artifact to choose. You can select the right artifact by adding information on the one to fetch in the `artifact` block, for example:

```
microejVee("com.mycompany:myveeport:1.0.0") {
  artifact {
    name = "artifact-name"
    type = "zip"
  }
}
```

This will select the artifact with the name `artifact-name` and with the type `zip`.

Refer to the Gradle documentation to learn all the options to select dependencies.

VEE Port project inside a multi-project

When the VEE Port project is in the same multi-project than the component which needs it (an Application for example), the VEE Port project should be declared as a project dependency.

For example if the multi-project contains an Application subproject named `my-app` and a VEE Port subproject called `vee-port`, the VEE Port project must be declared as a dependency in the `build.gradle.kts` file of the `my-app` subproject as follows:

```
dependencies {

    microejVee(project(":vee-port"))

}
```

and the *Architecture Usage* of the VEE Port must be set.

The VEE Port will be automatically built when it is required by the Application. For example when running the Application on the Simulator (with the `runOnSimulator` task) or when building the Application Executable (with the `buildExecutable` task), the VEE Port will be built before executing the requested task.

Local VEE Port project outside a multi-project

When the Application or the Library which needs the VEE Port is not in the same multi-project than the VEE Port, the VEE Port project can be imported thanks to the Gradle Composite Build feature.

This allows to consider the VEE Port project as part of the Application project, so all changes done to the VEE Port are automatically considered when building or running the Application.

This is done by adding the following line in the `settings.gradle.kts` file of the Application project:

```
includeBuild("[vee-port-project-path]")
```

Then declaring the VEE Port as a dependency in the `build.gradle.kts` file of the Application project:

```
dependencies {

    microejVee("com.mycompany:vee-port:1.0.0")

}
```

The dependency must use the module notation ("group:name:version"), where the group and name match with the ones declared in the VEE Port project. The group is defined in the build.gradle.kts file of the vee-port project by the group property. The name is defined in the settings.gradle.kts file when the vee-port project is included. For example, the name of the VEE Port is my-custom-vee-port if the vee-port subproject is included with:

```
include("my-custom-vee-port")
project(":my-custom-vee-port").projectDir = file("vee-port")
```

Otherwise the name of the subproject folder is used, so vee-port in the recommended structure.

Using a Local VEE Port Directory

When your VEE Port is available in a local directory, you can use it by declaring a file dependency in the build.gradle.kts file, with the microejVee configuration:

```
dependencies {
  microejVee(files("C:\\path\\to\\my\\veePort\\source"))
}
```

This is generally the case when the VEE Port has been built locally

- in SDK 6, by executing the buildVeePort Gradle task on the VEE Port project. In this case, the VEE Port directory is located at build/veePort/source in the project.

Warning: It is recommended to *include the VEE Port project* instead of using a file dependency to ensure that any change done to the VEE Port project is considered when building your Application without having to manually rebuild the VEE Port.

- in SDK 5, by executing a Build Module on the VEE Port configuration project. In this case, the VEE Port is a sibling folder of the VEE Port configuration project, named after the VEE Port name.

Note: The build.gradle.kts file, as well as other Gradle configuration files, respects the Java properties file convention: the OS path must use the UNIX path convention (path separator is /). The Windows paths must have been converted manually replacing \ by / or by \\ .

Using a Local VEE Port Archive

When your VEE Port is available locally as an archive file (.zip or .vde), you can use it by declaring a file dependency in the build.gradle.kts file, with the microejVee configuration:

```
dependencies {
  microejVee(files("C:\\path\\to\\my\\veePort\\file.zip"))
}
```

This is generally the case when

- the VEE Port has been built locally in SDK 6, by executing the `buildVeePort` Gradle task on the VEE Port project. In this case, the VEE Port archive is located at `build/veePort.zip` in the project.

Warning: It is recommended to *include the VEE Port project* instead of using a file dependency to ensure that any change done to the VEE Port project is considered when building your Application without having to manually rebuild the VEE Port.

- the VEE Port has been built and published in SDK 5. In this case, the VEE Port archive is available in an artifact repository and can be downloaded manually to be used in your Application or Library project.

Note: The legacy `JPF` format of a VEE Port is not supported anymore in the SDK 6. If you want to use a VEE Port `.jpf` file, you have to use *the SDK 5*.

Architecture Usage Selection

When the selected VEE Port is *fetched from a repository* or is a *subproject of a multi-project*, the default Architecture Usage is `eval` (no matter what is configured in the VEE Port project). It is possible to change the Architecture Usage in the project which needs it (an Application for example) by setting the `architectureUsage` property in the `microej` block in the `build.gradke.kts` file:

```
microej {
  architectureUsage = "prod"
}
```

Set the property to `prod` to use a Production Architecture and to `eval` to use an Evaluation Architecture. If not set, the `eval` value is used.

Warning: When the VEE Port is a local archive or folder (`microejVee(files(...))`), the Architecture Usage is defined when the VEE Port is built and can no longer be changed. In SDK 6 (with the `buildVeePort` task), the Architecture Usage is defined by setting the `architectureUsage` property in the `build.gradke.kts` file of the VEE Port project.

Resolve a VEE Port transitively

Warning: The transitive resolution of a VEE Port is not supported for VEE Ports built with SDK 5 or SDK 6 `1.2.0` or earlier provided as *module* or *project*. While this feature is optional for now, it will be enabled by default in the next SDK 6 major version, so it is highly recommended to update your VEE Port if necessary.

By default, the VEE Port is not fetched transitively by consumer projects, but starting from SDK 6 `1.4.0`, it is possible to enable the transitivity of the VEE Port by:

- Setting the project property `feature.vee.transitivity.enabled` to `true` in command line with the `-P` argument:

```
./gradlew runOnSimulator -Pfeature.vee.transitivity.enabled=true
```

- or by adding it in the `gradle.properties` file of your project:

```
feature.vee.transitivity.enabled=true
```

When the feature is enabled, the transitive dependencies of the VEE Port are fetched to build the compile classpath and runtime classpath of the project. For more information about the transitivity of the VEE Port, refer to *VEE transitivity*.

3.4.8 Run on Simulator

In order to execute an Application on the Simulator, the SDK provides the Gradle `runOnSimulator` task. The prerequisites of this task are:

- The Application EntryPoint must be configured, as described in *Configure a Project*.
- The target VEE must be defined:
  - If your VEE is a VEE Port, refer to the *Select a VEE Port* page to know the different ways to provide a VEE Port for a module project.
  - If your VEE is a Kernel, refer to the *Select a Kernel* page to know the different ways to provide a Kernel for a module project.

Once these prerequisites are fulfilled, the Application can be started with the Simulator:

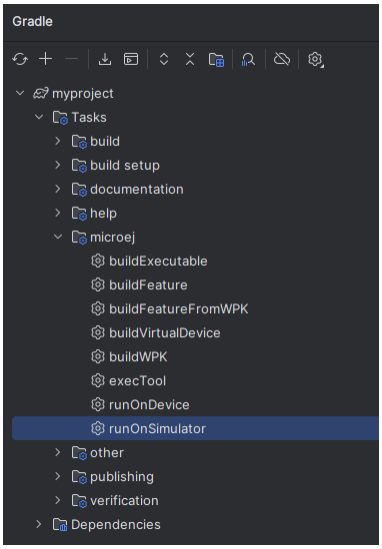
IntelliJ IDEA / Android Studio

Eclipse

Visual Studio Code

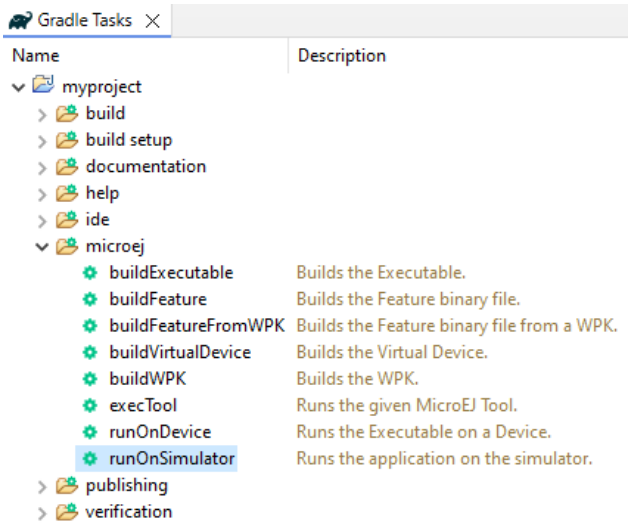
Command Line Interface

By double-clicking on the `runOnSimulator` task in the Gradle tasks view:

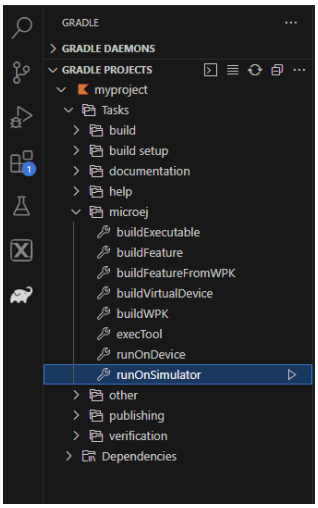


Warning: Android Studio does not allow to run multiple Gradle tasks in parallel. If you still want to execute several Gradle tasks simultaneously, you can launch them from a terminal with the Gradle Command Line Interface (CLI).

By double-clicking on the runOnSimulator task in the Gradle tasks view:



By double-clicking on the runOnSimulator task in the Gradle tasks view:



From the command line interface:

```
$ ./gradlew runOnSimulator
```

With a simple Hello World Application, the output should be:

```
================ [ Initialization Stage ] ================
================ [ Converting fonts ] ================
================ [ Converting images ] ================
================ [ Launching on Simulator ] ================
Hello World!
```

(continues on next page)

(continued from previous page)

```
================ [ Completed Successfully ] ================

SUCCESS
```

Warning: The execution of the runOnSimulator task can fail if the SDK EULA has not been accepted. In that case, you can set the ACCEPT_MICROEJ_SDK_EULA_V3_1C environment variable to YES and restart your IDE or you can define the accept-microej-sdk-eula-v3-1c System property by creating a *custom configuration*. For more information about SDK EULA, refer to the *Licenses* chapter.

Verbose Mode

If you need more information about the execution of the Application with the Simulator, the verbose mode can be enabled by using the --info Gradle option:

```
./gradlew runOnSimulator --info
```

Debug on Simulator

The SDK allows to run an Application with the Simulator in debug mode by setting the project property debug.mode to true when executing the runOnSimulator task:

```
./gradlew runOnSimulator -P"debug.mode"=true
```

The debug mode is activated on the port 12000 by default. The port can be changed by using the project Property debug.port :

```
./gradlew runOnSimulator -P"debug.mode"=true -P"debug.port"=8000
```

Once started, the Simulator waits for the connection of a debugger.

If you want to connect the IDE debugger:

IntelliJ IDEA / Android Studio

Eclipse

Visual Studio Code

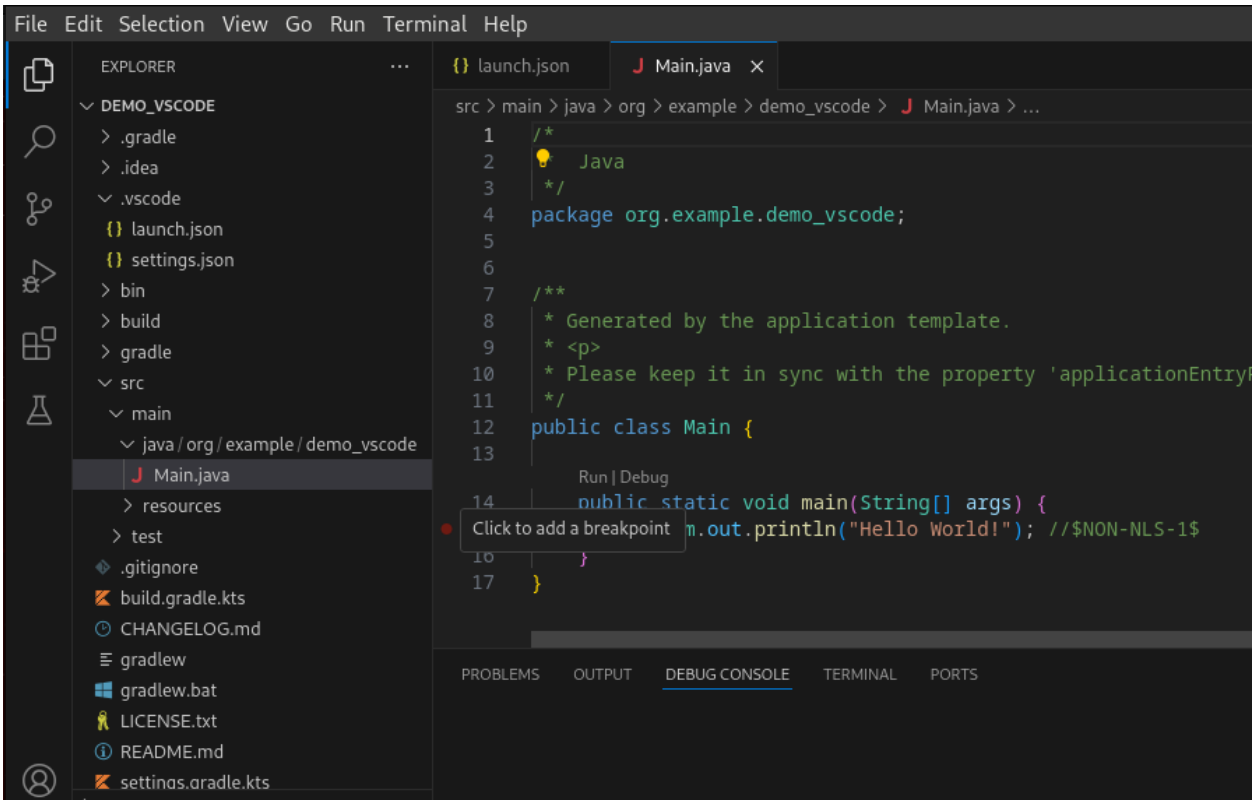
Warning: IntelliJ IDEA and Android Studio need an Architecture 8.1 or higher for debug mode.

- Add a breakpoint in your Application code.
- Click on Run > Edit Configurations… .
- Click on + button (Add New Configuration).
- Select Remote JVM Debug .
- Click on the New launch configuration button.
- Give a name to the launcher in the Name field.

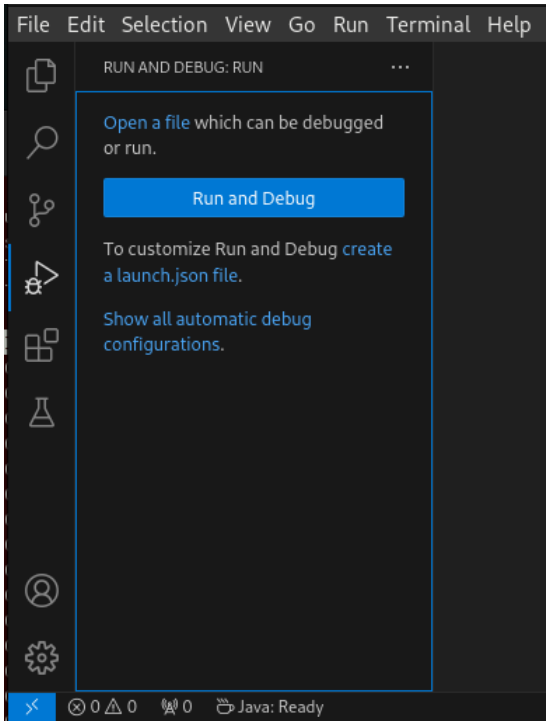
- Set the debug host and port.
- Click on the `Debug` button.
- Add a breakpoint in your Application code.
- Click on `Run` > `Debug Configurations...` .
- Select `Remote Java Application` .
- Click on the `New launch configuration` button.
- Give a name to the launcher in the `Name` field.
- Set the debug host and port.
- Click on the `Debug` button.

Warning: VS Code needs an Architecture 8.1 or higher for debug mode.

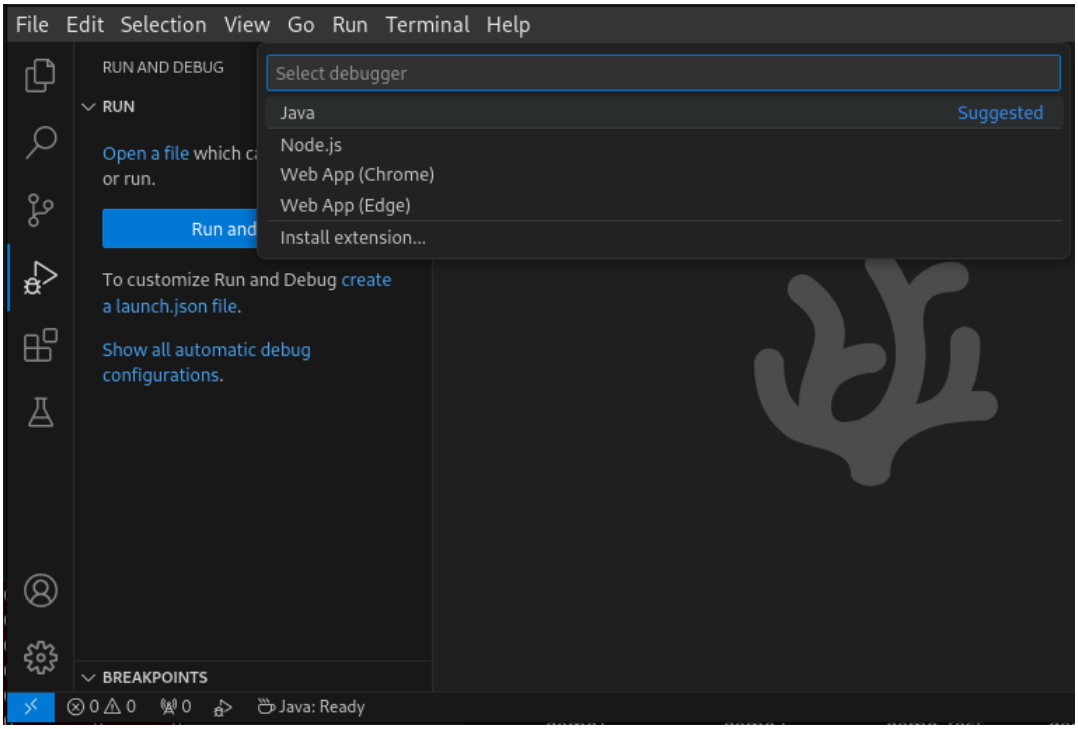
- Add a breakpoint in your Application code.



- Click on the `Run and Debug (Ctrl+Shift+D)` icon on the right panel.
- Click on create a launch.json file in the opened panel.

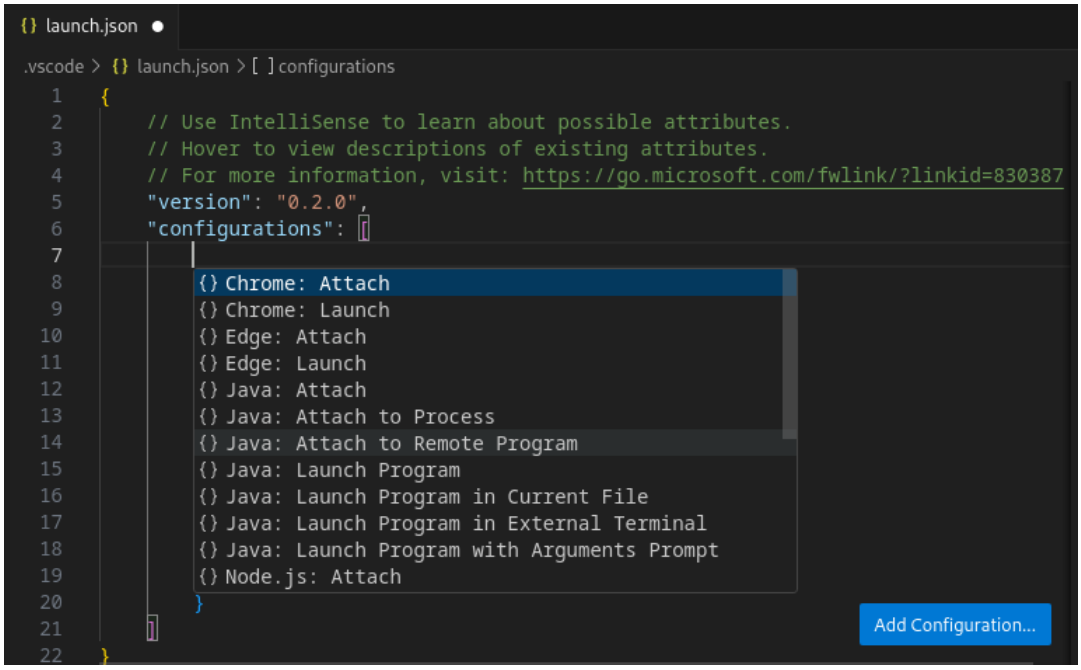


- Click on the Java entry proposed in the search field.

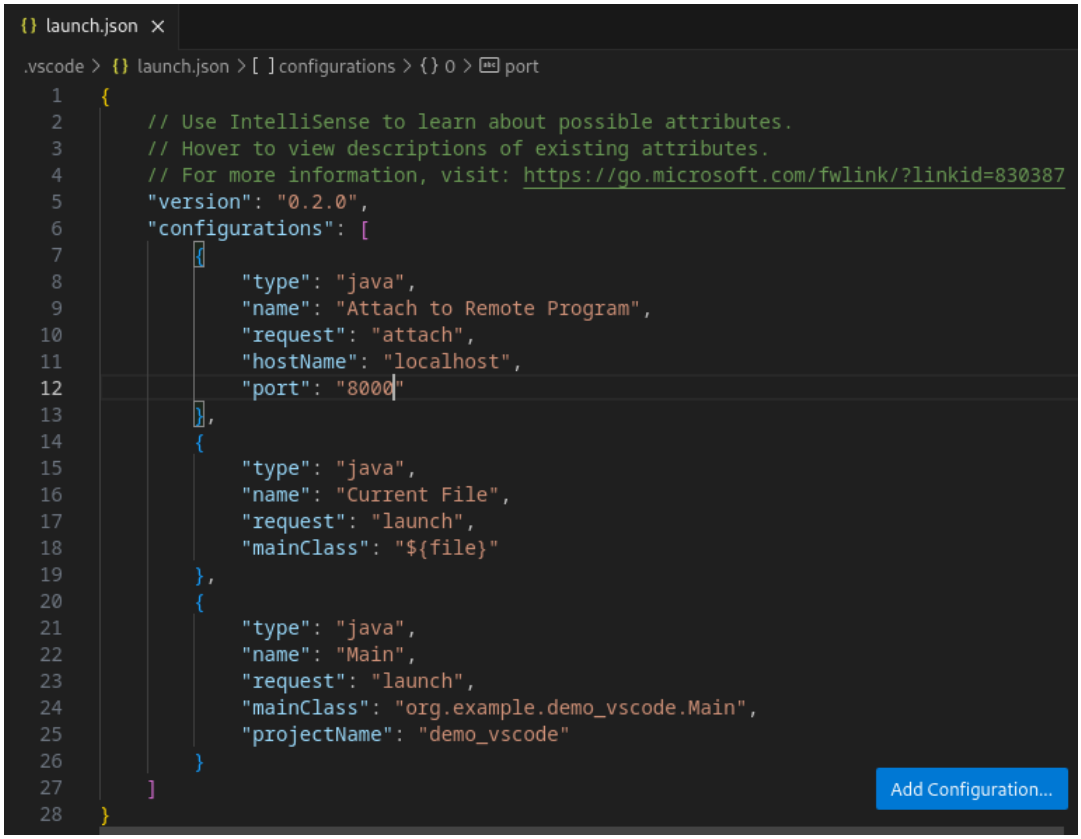


- Click on Add Configuration button

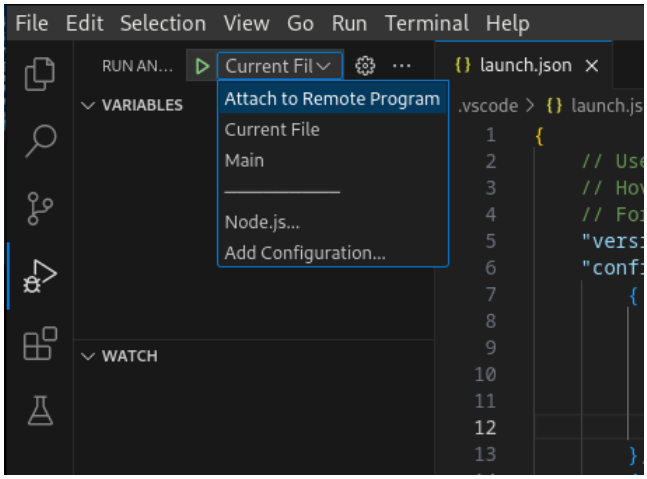
- Select {} Java: Attach to Remote Program entry in the popup list.



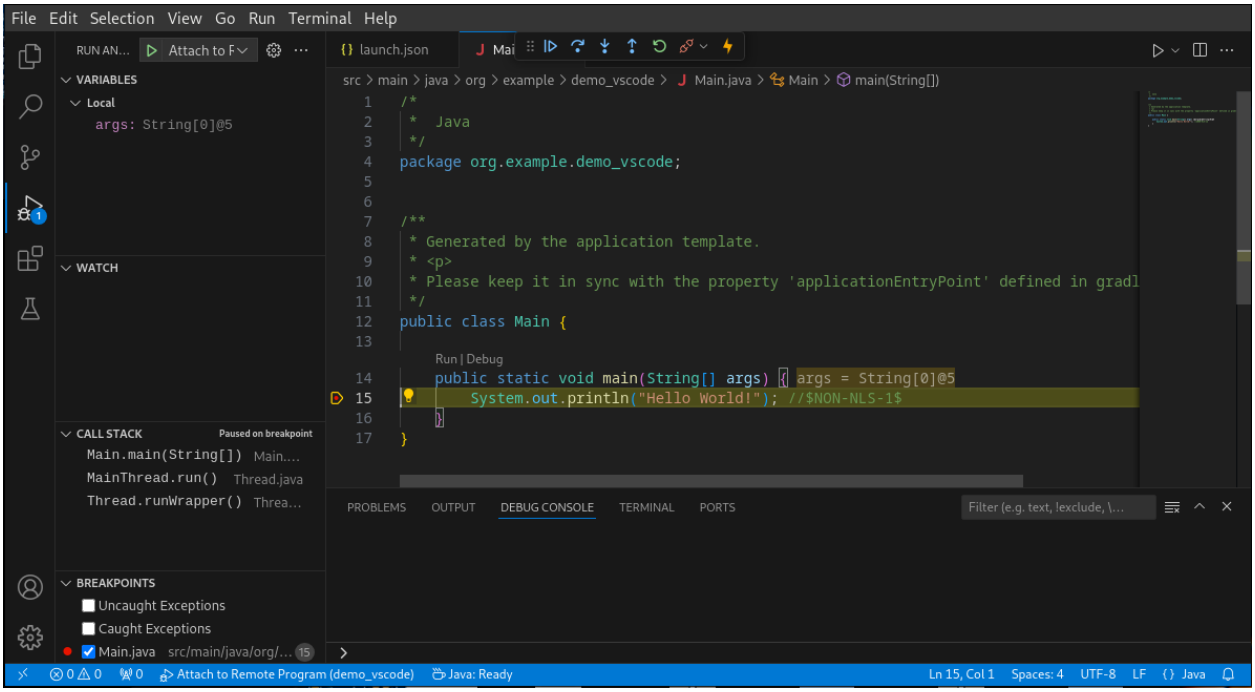
- Set localhost as hostName and the port (default is 1200) in the generated json.



- Select Attach to Remote Program in the selection box of the launcher.



- Click on the Start button



The debugger should connect to the Simulator and you should be able to debug your Application.

Generate Code Coverage

To generate the Code Coverage files (.cc), invoke the :runOnSimulator task as follow:

```
./gradlew :runOnSimulator -D"microej.option.s3.cc.thread.period=15" -D"microej.option.s3.cc.activated=true"
```

Option Name: s3.cc.thread.period

Description:

It specifies the period between the generation of .cc files.

Note: If the application is abruptly ended (for example with Ctrl-C) before the the first period, no .cc files are generated.

Option Name: s3.cc.activated

Description

Set to true to enable the generation of Code Coverage files, don't define the property to disable the generation.

Generate Heap Dump

Option Name: s3.inspect.heap

Description:

Set to true to enable a dump of the heap each time the System.gc() method is called by the MicroEJ Application. The .heap files are generated in build/output/application/heapDump/ .

Use the *Heap Viewer* to visualize the .heap files.

Run several Applications on Simulator

When a Multi-Sandbox Kernel is provided, it is possible to execute your Application on the Simulator along with additional Applications. To run an additional Application on the Simulator, the Application must be declared as a dependency of the project:

- When the Application is published in an artifact repository, you can use it by declaring a Module dependency:

```
dependencies {
   microejApplication("com.mycompany:myapp:1.0.0")
}
```

- When the Application is a subproject of a multi-project, you can use it by declaring a Project dependency in the build.gradle.kts file, with the microejApplication configuration:

```
dependencies {
   microejApplication(project(":myApplication"))
}
```

- You can also use the Application WPK file directly by declaring a File dependency in the build.gradle.kts file, with the microejApplication configuration:

```
dependencies {
   microejApplication(files("C:\\path\\to\\my\\application.wpk"))
}
```

Warning:

- Only modules with the *Application Module Nature* can be declared this way (modules built with the `com.microej.gradle.application` plugin). Declaring a module with another Module Nature would make the build fail.

3.4.9 Build an Executable

In order to build the Executable of an Application, the SDK provides the Gradle `buildExecutable` task. The prerequisites to use this task are:

- The Application EntryPoint must be configured, as described in *Configure a Project*.
- A target VEE Port that uses an Architecture version `7.17` minimum must be defined. Refer to the *Select a VEE Port* page to know the different ways to provide a VEE Port for a module project.

Once these prerequisites are fulfilled, the Executable can be built:

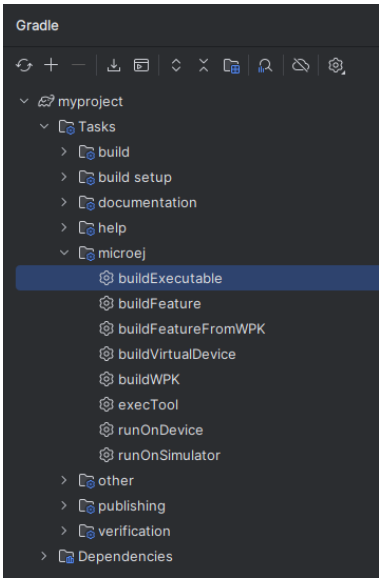
Android Studio / IntelliJ IDEA

Eclipse

Visual Studio Code

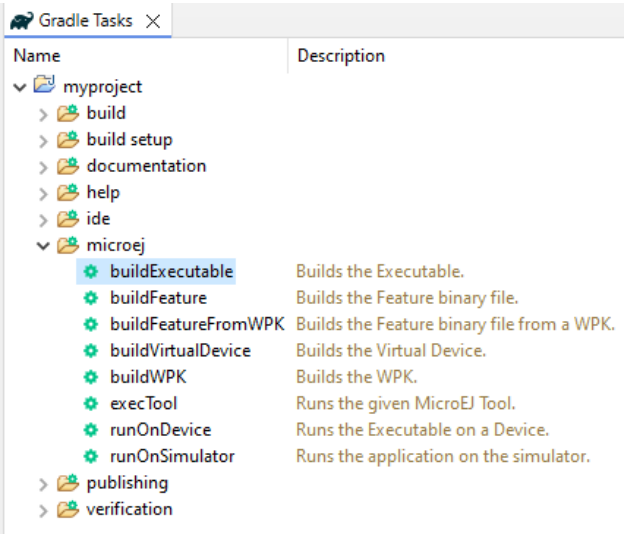
Command Line Interface

By double-clicking on the `buildExecutable` task in the Gradle tasks view:

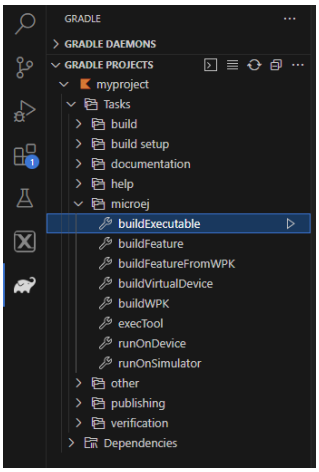


Warning: Android Studio does not allow to run multiple Gradle tasks in parallel. If you still want to execute several Gradle tasks simultaneously, you can launch them from a terminal with the Gradle Command Line Interface (CLI).

By double-clicking on the buildExecutable task in the Gradle tasks view:



By double-clicking on the buildExecutable task in the Gradle tasks view:



From the command line interface:

```
$ ./gradlew buildExecutable
```

In case of *Full BSP Connection*, the Executable file is generated in the build/application/executable folder of the project.

Trigger Executable Build by Default

The Executable of an Application is not built and published by default (when launching a ./gradlew build or a ./gradlew publish for example). This default behavior can be changed by adding the produceExecutableDuringBuild() method in the microej configuration block of the Gradle build file of the project:

```
microej {
  produceExecutableDuringBuild()
}
```

3.4.10 Run on Device

The SDK allows to deploy an Application on a Device thanks to the Gradle `runOnDevice` task. The prerequisites of this task are:

- The Application EntryPoint must be configured, as described in *Configure a Project*.
- The target VEE Port must be defined. Refer to the *Select a VEE Port* page to know the different ways to provide a VEE Port for a module project.
- The Device must be connected to the developer's computer.
- The configuration required by the VEE Port must be set. Refer to the VEE Port documentation to check the required configuration.

Once these prerequisites are fulfilled, the Application can be deployed on the Device:

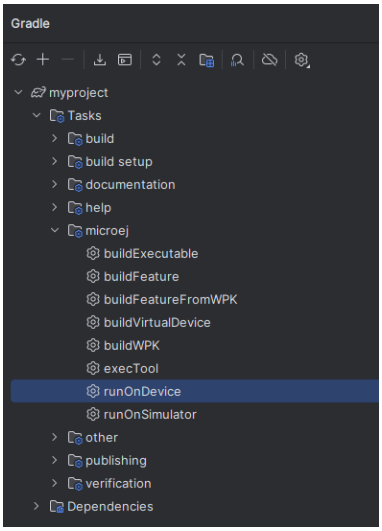
IntelliJ IDEA / Android Studio

Eclipse

Visual Studio Code

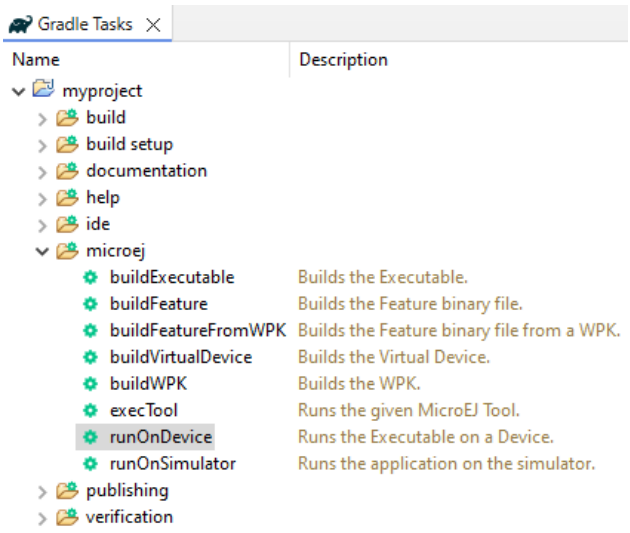
Command Line Interface

By double-clicking on the `runOnDevice` task in the Gradle tasks view:

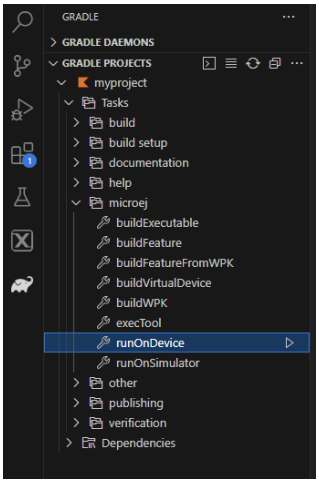


Warning: Android Studio does not allow to run multiple Gradle tasks in parallel. If you still want to execute several Gradle tasks simultaneously, you can launch them from a terminal with the Gradle Command Line Interface (CLI).

By double-clicking on the `runOnDevice` task in the Gradle tasks view:



By double-clicking on the runOnDevice task in the Gradle tasks view:



From the command line interface:

```
$ ./gradlew runOnDevice
```

The build should be successful and the output should end with:

```
Execution of script '<RUN_SCRIPT_PATH>' done.
BUILD SUCCESSFUL
```

where RUN_SCRIPT_PATH is the absolute path to the run.[sh|bat] script of the VEE Port.

The Application Executable is now deployed on the Device.

Deploying the Executable without building it

When executing the runOnDevice task, the Executable is always rebuilt, even if nothing has changed in the project. This ensures that the Executable is always up-to-date, no matter if the BSP has changed or not.

If you want to deploy the Executable on the device without building it (so simply deploying the Executable file already available in the project build folder), you can exclude the buildExecutable task:

```
$ ./gradlew runOnDevice -x buildExecutable
```

You can execute such a command in IDEs by creating custom configurations, as explained in *How To Create a Custom Configuration in the IDE*.

3.4.11 Select a Kernel

Building the Feature file of an Application with the SDK requires a Kernel.

Use one of the following available options to provide it to your project.

Using a Module Dependency

When your Kernel is published in an artifact repository, you can define the Kernel by declaring a module dependency in the build.gradle.kts file, with the microejVee configuration:

```
dependencies {
  microejVee("com.mycompany:mykernel:1.0.0")
}
```

Kernel project inside a multi-project

When the Kernel project is in the same multi-project as the component that depends on it (an Application for example), the Kernel project should be declared as a project dependency.

For example if the multi-project contains an Application subproject named my-app and a Kernel subproject called my-kernel , the Kernel project must be declared as a dependency in the build.gradle.kts file of the my-app subproject as follows:

```
dependencies {

   microejVee(project(":my-kernel"))

}
```

To find the name of the Kernel project, you can execute the projects task on the multi-project to display all its subprojects, for example:

```
Root project 'my-multi-project'
+--- Project ':application'
+--- Project ':library'
\--- Project ':my-kernel'
```

The Kernel Virtual Device and Executable will be automatically built when they are required by the Application. For example the Virtual Device will be built before running the Application on the Simulator (with the runOnSimulator task), and the Executable will be built before building the Application Feature (with the buildFeature task).

Local Kernel project outside a multi-project

When the Application or the Library depending on the Kernel is not in the same multi-project as the Kernel, the Kernel project can be imported using the Gradle Composite Build feature.

This allows the Kernel project to be considered part of the Application project, so all changes done to the Kernel are automatically reflected when building or running the Application.

This is done by adding the following line in the `settings.gradle.kts` file of the Application project:

```
includeBuild("[kernel-project-path]")
```

Then declaring the Kernel as a dependency in the `build.gradle.kts` file of the Application project:

```
dependencies {

    microejVee("com.mycompany:my-kernel:1.0.0")

}
```

The dependency must use the module notation (`"group:name:version"`), where the group and name match with the ones declared in the Kernel project (the version is ignored). The group is defined in the `build.gradle.kts` file of the `my-kernel` project by the `group` property. To find the name of the Kernel project, you can execute the `projects` task on the multi-project to display all its subprojects, for example:

```
Root project 'my-multi-project'
+--- Project ':application'
+--- Project ':library'
\--- Project ':my-kernel'
```

Using a Local Kernel

When your Kernel has been built locally, you can use its Virtual Device and its Executable by declaring a file dependency in the `build.gradle.kts` file, with the `microejVee` configuration:

```
dependencies {
  microejVee(files("C:\\path\\to\\my\\kernel\\virtualDevice\\vee", "C:\\path\\to\\my\\kernel\\executable.out
↪"))
}
```

This is generally the case when the VEE Port has been built locally

- in SDK 6, by executing the `buildVirtualDevice` and `buildExecutable` Gradle tasks on the Kernel project. In this case, the Virtual directory is located in the `build/virtualDevice` of the project and the Executable is located in the `build/application/executable` folder.

Warning: It is recommended to *include the Kernel project* instead of using a file dependency to ensure that any change done to the Kernel project is considered when building your Application without having to manually rebuild the Virtual Device and the Executable.

- in SDK 5, by executing a `Build Module` on the Kernel project. In this case, the Virtual Device archive file and the Executable are located in the `target~/artifacts` folder of the Kernel project.

Note: The `build.gradle.kts` file, as well as other Gradle configuration files, respects the Java properties file convention: the OS path must use the UNIX path convention (path separator is `/`). The Windows paths must have been

converted manually replacing \ by / or by \\ .

Resolve a Kernel transitively

Resolving a Kernel transitively is not supported since it causes issues when building an Application on the Kernel.

When a multi-project is composed of Application projects which depends on a Kernel project, the VEE transitivity should be enabled only in the Kernel project for the VEE Port resolution. Therefore, the `feature.vee.transitivity.enabled` Gradle properties should be set to `true` in the Kernel project only, for example by defining it in the `gradle.properties` file of the project. It should not be set in the Application projects.

3.4.12 Build a Feature file

To build the Feature file (`.fo`) of an Application, the SDK provides the Gradle `buildFeature` task. The prerequisites to use this task are:

- The Application EntryPoint must be configured, as described in *Configure a Project*.
- A Multi-Sandbox Kernel must be defined. Refer to the *Select a Kernel* page to learn how to provide a Kernel for a module project.

Once these prerequisites are fulfilled, the Feature file can be built:

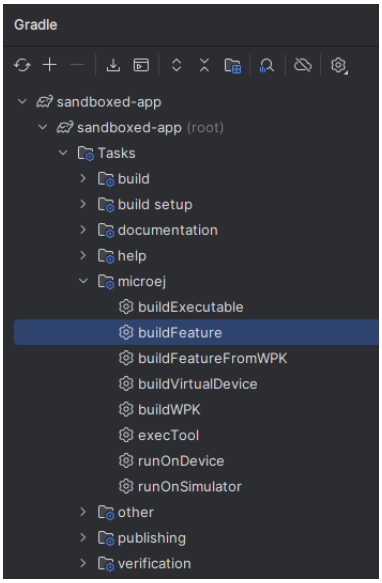
IntelliJ IDEA / Android Studio

Eclipse

Visual Studio Code

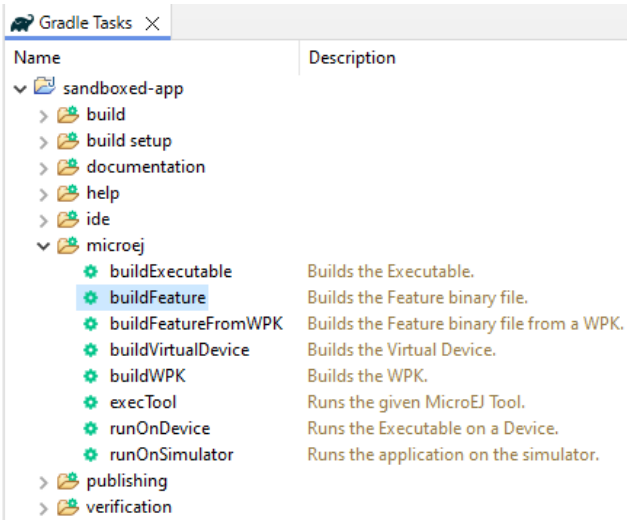
Command Line Interface

By double-clicking on the `buildFeature` task in the Gradle tasks view:

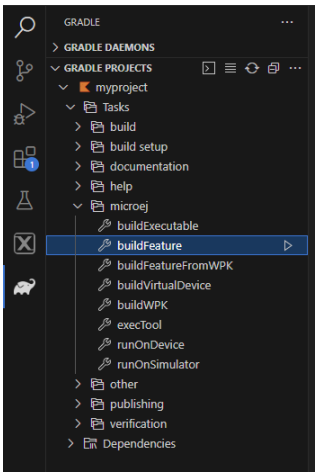


Warning: Android Studio does not allow to run multiple Gradle tasks in parallel. If you still want to execute several Gradle tasks simultaneously, you can launch them from a terminal with the Gradle Command Line Interface (CLI).

By double-clicking on the buildFeature task in the Gradle tasks view:



By double-clicking on the buildFeature task in the Gradle tasks view:



From the command line interface:

```
$ ./gradlew buildFeature
```

The Feature file is generated in the build/application/feature folder of the project.

Trigger Feature Build by Default

The Feature of an Application is not built and published by default (when launching a `./gradlew build` or a `./gradlew publish` for example). This default behavior can be changed by adding the `produceFeatureDuringBuild()` method in the `microej` configuration block of the Gradle build file of the project:

```
microej {
  produceFeatureDuringBuild()
}
```

Build a Feature file from a WPK

The SDK provides the Gradle `buildFeatureFromWPK` task which allows to build a Feature file from the WPK of an Application. The prerequisites to use this task are:

- The WPK must be defined:
  - When the WPK is published in an artifact repository, you can use it by declaring a Module dependency:

    ```
    dependencies {
      microejApplication("com.mycompany:myapp:1.0.0")
    }
    ```

  - When the WPK is built by a subproject of a multi-project, you can use it by declaring a Project dependency in the `build.gradle.kts` file, with the `microejApplication` configuration:

    ```
    dependencies {
      microejApplication(project(":myApplication"))
    }
    ```

  - When the WPK is available locally, you can use it by declaring a File dependency in the `build.gradle.kts` file, with the `microejApplication` configuration:

    ```
    dependencies {
      microejApplication(files("C:\\path\\to\\my\\application.wpk"))
    }
    ```

- A Multi-Sandbox Kernel must be defined. Refer to the *Select a Kernel* page to learn how to provide a Kernel for a module project.

Once these prerequisites are fulfilled, the Feature file can be built:

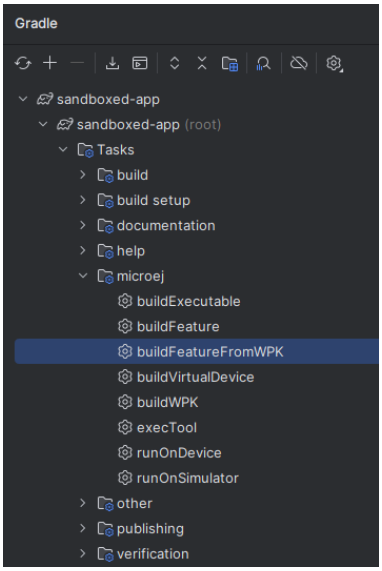
IntelliJ IDEA / Android Studio

Eclipse

Visual Studio Code

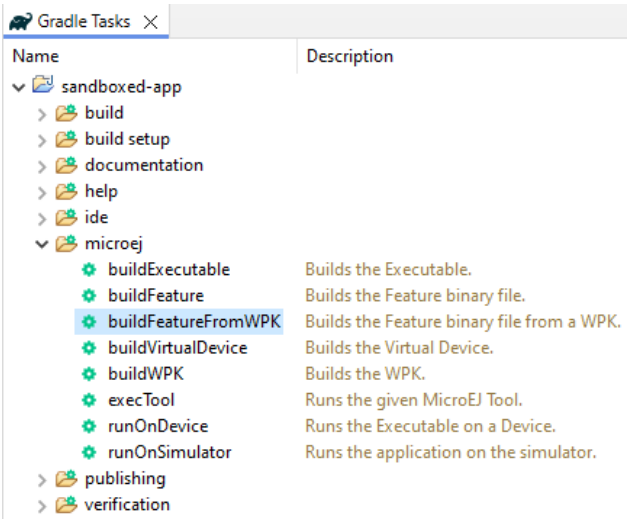
Command Line Interface

By double-clicking on the `buildFeatureFromWPK` task in the Gradle tasks view:

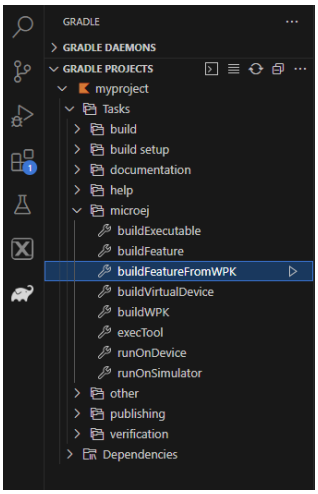


Warning: Android Studio does not allow to run multiple Gradle tasks in parallel. If you still want to execute several Gradle tasks simultaneously, you can launch them from a terminal with the Gradle Command Line Interface (CLI).

By double-clicking on the buildFeatureFromWPK task in the Gradle tasks view:



By double-clicking on the buildFeatureFromWPK task in the Gradle tasks view:



From the command line interface:

```
$ ./gradlew buildFeatureFromWPK
```

The Feature file is generated in the `build/application/wpkFeature` folder of the project.

3.4.13 Build a Virtual Device

In order to build the Virtual Device of an Application, the SDK provides the Gradle `buildVirtualDevice` task. Refer to the *Virtual Device* page for more information about the Virtual Device.

The prerequisites to use the `buildVirtualDevice` task are:

- The Application EntryPoint must be configured, as described in *Configure a Project*.
- A target VEE that uses an Architecture version `7.17` minimum must be defined:
  - If your VEE is a VEE Port, refer to the *Select a VEE Port* page to know the different ways to provide a VEE Port for a module project.
  - If your VEE is a Kernel, refer to the *Select a Kernel* page to know the different ways to provide a Kernel for a module project.

Once these prerequisites are fulfilled, the Virtual Device can be built:

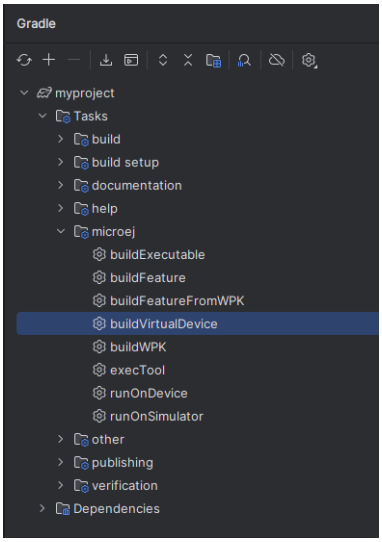
IntelliJ IDEA / Android Studio

Eclipse

Visual Studio Code

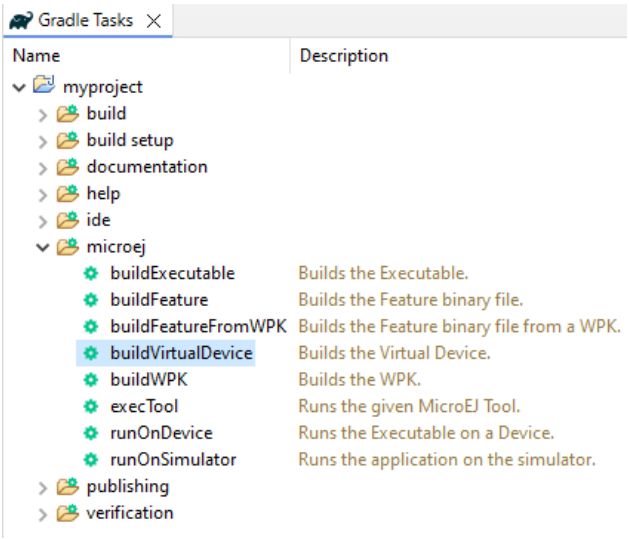
Command Line Interface

By double-clicking on the `buildVirtualDevice` task in the Gradle tasks view:

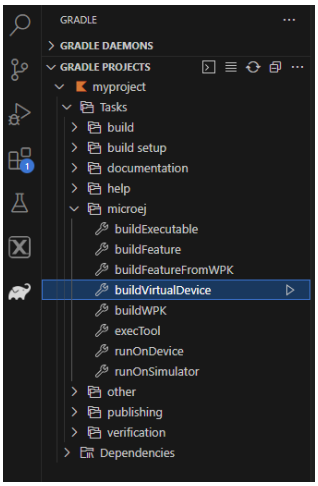


Warning: Android Studio does not allow to run multiple Gradle tasks in parallel. If you still want to execute several Gradle tasks simultaneously, you can launch them from a terminal with the Gradle Command Line Interface (CLI).

By double-clicking on the buildVirtualDevice task in the Gradle tasks view:



By double-clicking on the buildVirtualDevice task in the Gradle tasks view:



From the command line interface:

```
$ ./gradlew buildVirtualDevice
```

When the build is completed, the Virtual Device is available in the `build/virtualDevice` folder of the project.

Note: If the provided VEE is a Kernel, the generated Virtual Device is an augmented version of the Kernel Virtual Device, in which the Application is set as a *Pre-Installed Application*.

The Virtual Device can then be used to *run an Application on the Simulator* for example.

Embed JDKs in a Virtual Device

The Virtual Device can embed its own JDK, so it can run out of the box without requiring a JDK to be installed on the target system. This feature is disabled by default and can be enabled by setting the `microej > virtualDevice > includeJdks` property to `true` in the `build.gradle.kts` file of the project:

```
microej {
  virtualDevice {
    includeJdks = true
    jdkWindowsX64Url = "https://github.com/adoptium/temurin11-binaries/releases/download/jdk-11.0.26
↪%2B4/OpenJDK11U-jdk_x64_windows_hotspot_11.0.26_4.zip"
    jdkLinuxX64Url = "https://github.com/adoptium/temurin11-binaries/releases/download/jdk-11.0.29%2B7/
↪OpenJDK11U-jdk_x64_linux_hotspot_11.0.29_7.tar.gz"
  }
}
```

As shown in the sample above, the URLs of the JDKs to download and embed must be defined for each OS/Architecture you want to support. The JDK URLs can be specified for the 3 OSes `Windows`, `Linux` and `macOs`, and the 2 Architectures `x64` and `Aarch64`, thanks to the following properties:

- `jdkWindowsX64Url`: `Windows` on `x64`
- `jdkWindowsAarch64Url`: `Windows` on `Aarch64`
- `jdkLinuxX64Url`: `Linux` on `x64`
- `jdkLinuxAarch64Url`: `Linux` on `Aarch64`
- `jdkMacOsX64Url`: `macOs` on `x64`

- jdkMacOsAarch64Url : macOs on Aarch64

It's worth mentioning that only .zip and .tar.gz files are supported.

Warning: Embedding JDKs in the Virtual Device makes its usage easier but comes with a cost in term of size. Each JDK adds several dozen of MB, that can lead to a quite big Virtual Device Archive.

Add a Pre-Installed Application in a Virtual Device

When building a Virtual Device for a Kernel, Applications can be pre-installed inside. These Applications can be loaded and started when the Kernel starts for example.

To install an Application in a Virtual Device for a Kernel, you must declare the Application as a dependency of the project:

- When the Application is published in an artifact repository, you can use it by declaring a Module dependency:

```
dependencies {
   microejApplication("com.mycompany:myapp:1.0.0")
}
```

- When the Application is a subproject of a multi-project, you can use it by declaring a Project dependency in the build.gradle.kts file, with the microejApplication configuration:

```
dependencies {
   microejApplication(project(":myApplication"))
}
```

- You can also use the Application WPK file directly by declaring a File dependency in the build.gradle.kts file, with the microejApplication configuration:

```
dependencies {
   microejApplication(files("C:\\path\\to\\my\\application.wpk"))
}
```

Warning:

- Only modules with the *Application Module Nature* can be declared this way (modules built with the com.microej.gradle.application plugin). Declaring a module with another Module Nature would make the build fail.
- The VEE Port used to create the Virtual Device has to be a Multi-Sandbox VEE Port to support the load of these pre-installed Applications.

Add a Kernel API in a Virtual Device

When building a Virtual Device for a Kernel, the Kernel must define the set of classes, methods and static fields all applications are allowed to use. This can be done by declaring *Kernel APIs* as a dependency in the build file:

```
dependencies {
  implementation("com.microej.kernelapi:edc:1.2.0")
}
```

Add a Runtime Environment in a Virtual Device

When building a Virtual Device for a Kernel, the set of classes, methods and static fields allowed to be used by all applications can be defined by declaring a *Runtime Environment* as a dependency in the build file:

```
dependencies {
  microejRuntimeEnvironment("com.mycompany:myruntime-environment:1.0.0")
}
```

The transitive dependencies of the Runtime Environment are then embedded in the Virtual Device.

Add a Tool in a Virtual Device

When building a Virtual Device, it is possible to define additional MicroEJ Tools to install inside, by adding a dependency with the `microejTool` configuration. For example, to install the *Local Deployment Socket* tool, add the following dependency to the build file of the project:

```
dependencies {
  microejTool("com.microej.tool.kernel:localdeploy-extension:1.0.0")
}
```

Trigger Virtual Device Build by Default

The Virtual Device of an Application is not built and published by default (when launching a `./gradlew build` or a `./gradlew publish` for example). This default behavior can be changed by adding the `produceVirtualDeviceDuringBuild()` method in the `microej` configuration block of the Gradle build file of the project:

```
microej {
  produceVirtualDeviceDuringBuild()
}
```

3.4.14 Add a Dependency

A project generally relies on other components such as libraries. These components have to be declared as dependencies of the build to be used by the project. This declaration is done in the `dependencies` block of the `build.gradle.kts` file. For example, to add the EDC library as a dependency:

```
dependencies {
  implementation("ej.api:edc:1.3.5")
}
```

Configurations

Every dependency declared for a Gradle project applies to a specific scope. For example some dependencies should be used for compiling source code whereas others only need to be available at runtime. Gradle represents the scope of a dependency with the help of a configuration. In the above example, the `implementation` configuration is used.

Since the MicroEJ Gradle plugins extend the Gradle Java and Java Library plugins, they inherits from their configurations, but they also adds their own configurations. Let's have a look at the mostly used configurations:

- **implementation** (from Gradle Java plugin) : Dependencies used by the project at compile time and runtime.
- **api** (from Gradle Java Library plugin) : Same as the **implementation** configuration, except that the dependency is also exposed to the consumers of your project.
- **testImplementation** (from Gradle Java plugin) : Dependencies used by the test classes of the project. This configuration extends the **implementation** configuration, so it inherits from all the dependencies declared with the **implementation** configuration.
- **microejVee** : VEE Port, Virtual Device or Kernel used by the project for build and test.
- **testMicroejVee** : VEE Port, Virtual Device or Kernel used by the project for test only.

Here is an example of dependencies declaration for a project:

```
dependencies {
  implementation("ej.library.runtime:basictool:1.7.0")

  testImplementation("ej.library.test:junit:1.12.0")

  microejVee("com.microej.platform.esp32.esp-wrover-kit-v41:HDAHT:1.8.0")
}
```

In this example, the `ej.library.runtime:basictool` module is used at compile time and runtime, the `ej.library.test:junit` module is used for the tests compilation and execution, and the `com.microej.platform.esp32.esp-wrover-kit-v41:HDAHT` module is the VEE Port used for build and test.

For an exhaustive list of the available configurations and more details on how to manage dependencies, refer to the following official documentations:

- Declaring dependencies
- Java plugin
- Java Library plugin

Version

The version declared in the dependencies of a build file are explicit:

- Release version: to depend on a released version of a module, the exact fixed version must be used (e.g., `1.0.0`).
- Snapshot version: to depend on a snapshot version (`-RC`) of a module, the version must be declared explicitly with the `-RC+` suffix (e.g., `1.0.0-RC+`).

Note: This is an important change compared to the SDK 5. In the SDK 5, using a fixed version (e.g., `1.0.0`) fetched the release version (e.g., `1.0.0`) if it existed, or a snapshot version (e.g., `1.0.0-RCxxx`) otherwise. This is not the case anymore in the SDK 6.

Version Check

In order to reduce the risk of mistakes, a check is done during the resolution process on the declared dependencies versions. The dependencies versions must start with digits and be followed by a dot, otherwise the build fails. For example, when declaring a dependency on `edc` with a version `x1.3.5` instead of `1.3.5`:

```
dependencies {
  implementation("ej.api:edc:x1.3.5")
}
```

the following error is raised:

```
* What went wrong:
Execution failed for task ':dependencies'.
> The version of the dependency "ej.api:edc" is not correct: "x1.3.5". It must start with digits, followed by a dot.
```

It is possible to disable this check by setting the `dependenciesVersionsCheckEnabled` property of the `microej` configuration block to `false` in the project build file:

```
microej {
  dependenciesVersionsCheckEnabled = false
}
```

Dependencies Repositories

Gradle needs to know in which repositories the modules must be fetched and published. The *SDK 6 installation process* provides a Gradle Init Script to declare the *MicroEJ public repositories*. You can declare other repositories, either in the same Gradle Init Script and in any other location supported by Gradle. Refer to the official documentation for more information on repositories configuration.

It is important to note that the declaration order of the repositories matters. Gradle requests the repositories in the order they are declared and stops as soon as it finds a matching version.

3.4.15 Test a Project

The SDK provides the capabilities to implement and run tests for a project. It relies on the standard JUnit API.

There are different types of tests:

- Test on the Simulator
- Test on a device
- Test on a Java SE VM

Each type of test is detailed in the next sections.

JUnit Compliance

The SDK relies on JUnit, the most popular Java testing framework, to define and execute the tests. It supports a subset of JUnit 4, namely the annotations: `@After`, `@AfterClass`, `@Before`, `@BeforeClass`, `@Ignore`, `@Test`.

Each test case entry point must be declared using the `org.junit.Test` annotation (`@Test` before a method declaration). Refer to JUnit documentation to get details on the usage of other annotations.

Gradle Integration

Tests are configured and launched by Gradle. Gradle provides 2 ways to configure tests in a project:

- By using the built-in `Test` task, as described in Testing in Java & JVM projects.
- By using the new `JVM Testsuite` plugin, as described in The JVM Testsuite Plugin.

The `JVM Testsuite` plugin provides an enhanced model to declare multiple groups of automated testsuites, and is therefore the recommended way to configure your tests. The next sections use the `JVM Testsuite` plugin to explain how to configure and run tests, but the same results can be achieved with the `Test` task.

Test on Simulator

Tests can be executed on the Simulator. They are run on a target VEE Port and generate a JUnit XML report.

Executing tests on the Simulator allows to check the behavior of the code in an environment similar to the target device but without requiring the board. This solution is therefore less constraining and more portable than testing on the board.

Configure the Testsuite

The configuration of the testsuites of a project must be defined inside the following block in the `build.gradle.kts` file:

```
testing {
  suites { // (1)
    val test by getting(JvmTestSuite::class) { // (2)
      microej.useMicroejTestEngine(this) // (3)

      dependencies { // (4)
        implementation(project())
        implementation("ej.api:edc:1.3.5")
        implementation("ej.library.test:junit:1.12.0")
      }
    }
  }
}
```

This piece of configuration is the minimum configuration required to define a testsuite on the Simulator:

- (1) : configures all the testsuites of the project.
- (2) : configures the built-in `test` suite provided by Gradle. Use this testsuite to configure the tests on the Simulator.
- (3) : declares that this testsuite uses the MicroEJ Testsuite Engine. By default, the MicroEJ Testsuite Engine executes the tests on the Simulator.

- (4) : adds the dependencies required by the tests. The first line declares a dependency to the code of the project. The second line declares a dependency on the edc Library. The third line declares a dependency to the JUnit API used to annotate Java Test classes.

Warning: With SDK 6 prior to 1.1.0 , junit-platform-launcher must be added as a test dependency:

```
implementation("org.junit.platform:junit-platform-launcher:1.8.2")
```

Configure the VEE

The VEE used to execute the tests must be declared in the project dependencies, with the microejVee or the testMicroejVee configuration (refer to *Select a VEE Port* for more details on the selection capabilities). A VEE declared with the microejVee configuration is used to run the Application, as well as to execute the testsuites. The microejVee is generally used in Application projects, since the tests should run on the same Application target VEE:

```
dependencies {
    ...
    microejVee("com.mycompany:vee-port:1.0.0")
}
```

A VEE declared with the testMicroejVee configuration is used only for the testsuites. It is recommended in Library projects, since they don't need a VEE to run, the VEE is scoped for the tests only:

```
dependencies {
    ...
    testMicroejVee("com.mycompany:vee-port:1.0.0")
}
```

As a summary, the rules are:

- Only one VEE must be declared globally.
- If the VEE is declared with microejVee , it is used to run the Application (if it is an Application) and to execute the tests.
- If the VEE is declared with testMicroejVee , it is only used to execute the tests.

Warning: Declaring a VEE in project dependencies only applies to the current project. By default the transitive resolution of the VEE is disabled but you can enable it with the project property feature.vee.transitivity.enabled . For more information, refer to *Resolve a VEE Port transitively*.

Create a Test Class

The SDK provides a JUnit library containing the subset of the supported JUnit API: ej.library.test:junit . Before creating the Test class, make sure this library is declared in the testsuite dependencies:

```
testing {
  suites {
    val test by getting(JvmTestSuite::class) {
      ...
      dependencies {
```

(continues on next page)

(continued from previous page)

```
            implementation("ej.library.test:junit:1.12.0")
        }
        ...
    }
  }
}
```

The test class can now be created in the `src/test/java` folder. This can be done manually or with IDE menu:

IntelliJ IDEA / Android Studio

Eclipse

Visual Studio Code

- right-click on the `src/test/java` folder.
- select `New` > `Java Class`, then press `Alt` + `Insert` and select `Test Method`.
- right-click on the `src/test/java` folder.
- select `New` > `Other...` > `Java` > `JUnit` > `New JUnit Test Case`.
- right-click on the `src/test/java` folder in `JAVA PROJECTS` view.
- select the `+` icon (`New...`) > `Class`, then enter the test class name you want to create.

Note: Gradle allows to define alternative folders for test sources but it would require additional configuration, so it is recommended to stick with the `src/test/java` folder.

Execute the Tests

Once the testsuite is configured, it can be run thanks to the `test` Gradle task. This task is bound to the `check` and the `build` Gradle lifecycle tasks, which means that the tests are also executed when launching one of these tasks.

IntelliJ IDEA / Android Studio

Eclipse

Visual Studio Code

Command Line Interface

In order to execute the testsuite from IntelliJ IDEA or Android Studio, double-click on the task in the Gradle tasks view:

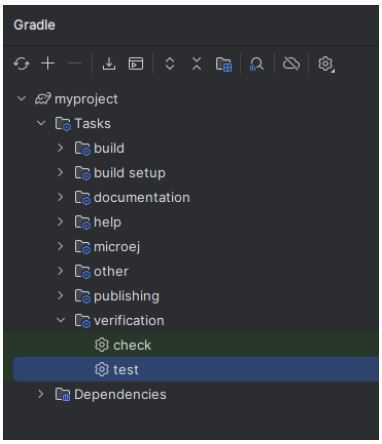


Fig. 47: Run Gradle test task from IntelliJ IDEA / Android Studio

In order to execute the testsuite from Eclipse, double-click on the task in the Gradle tasks view:

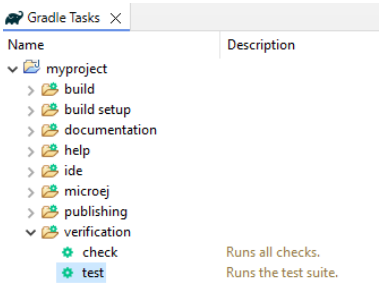


Fig. 48: Run Gradle test task from Eclipse

Warning: By right-clicking on a test class file, the menu proposes `Gradle Test` and `JUnit Test` in the `Run As` entry.

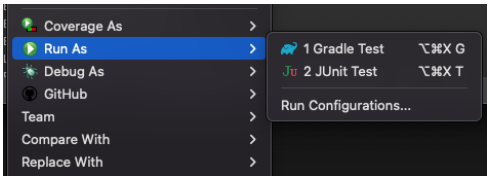


Fig. 49: Run test as Gradle test in a class right-click menu

Always use the `Run` > `Gradle Test` entry.

In order to execute the testsuite from VS Code, double-click on the task in the Gradle tasks view:

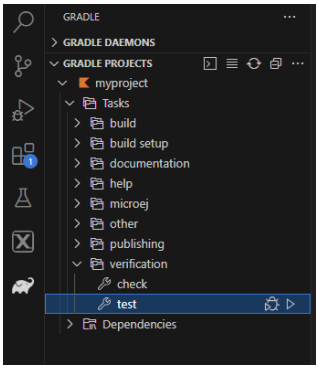


Fig. 50: Run Gradle test task from Visual Studio Code

Warning: Test start buttons (represented as green triangle) may appear on the left side of class and method definitions.

Fig. 51: Green triangles are test start buttons

Running tests from these buttons may fail because they do not use the Gradle Runner by default. To run a test with Gradle, right-click on the green triangle and select `Execute Using Profile…`

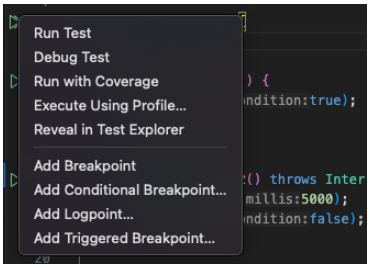


Fig. 52: Right-click menu on test start buttons

and then select `Delegate Test to Gradle` .

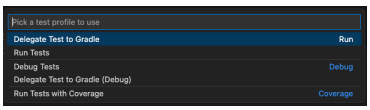


Fig. 53: Run test class or test method with Gradle

In order to execute the testsuite from the Command Line Interface, execute this command:

```
$ ./gradlew test
```

Generate Code Coverage

To generate the Code Coverage files (.cc) for each test, configure the testsuite as follows:

```
testing {
  suites {
    val test by getting(JvmTestSuite::class) {

      ...

      targets {
        all {
          testTask.configure {
            doFirst {
              systemProperties["microej.testsuite.properties.s3.cc.activated"] = "true"
              systemProperties["microej.testsuite.properties.s3.cc.thread.period"] = "15"
            }
          }
        }
      }
    }
  }
}
```

Then, to generate an HTML report, see *Code Coverage Analyzer*.

Filter the Tests

Gradle automatically executes all the tests located in the test source folder. If you want to execute only a subset of these tests, Gradle provides 2 solutions:

- Filtering configuration in the build script file.
- Filtering option in the command line.

The tests filtering configuration must be done in the `filter` block of the test task:

```
testing {
  suites {
    val test by getting(JvmTestSuite::class) {

      ...

      targets {
        all {
          testTask.configure {
            filter {
              includeTestsMatching("com.mycompany.*")
            }
          }
        }
      }
    }
```

(continues on next page)

(continued from previous page)

```
    }
}
```

This example tells Gradle to run the tests located in the `com.mycompany` package only. Other methods are available for test filtering, such as `excludeTestsMatching` to exclude tests. Refer to the TestFilter documentation for the complete list of available filtering methods.

Gradle also allows to filter the tests from the command line directly, thanks to the `--tests` option. For example, to execute only the tests from the class `MyTestClass`, run this command:

```
./gradlew test --tests MyTestClass
```

This can be convenient to quickly execute one test, without requiring a change in the build script file.

Note: The test class referenced by the `--tests` option is executed only if it is not excluded in the test configuration in the `build.gradle.kts` file. Therefore, make sure to adpat your test configuration when using this option.

Refer to the Gradle Test filtering documentation for more details on how to filter the tests and on the available patterns.

> **Warning:** At the moment, only class-level filtering is supported. This means that, for instance, it is not possible to run a single test method within a test class.

Debug the Tests on Simulator

A test can be executed on the Simulator in debug mode by setting the property `debug.mode` to `true` when executing the `test` task:

```
./gradlew test --tests MyTest -P"debug.mode"=true -P"debug.port"=12000
```

Once started, the Simulator waits for the connection of a debugger.

If you want to connect the IDE debugger:

IntelliJ IDEA / Android Studio

Eclipse

Visual Studio Code

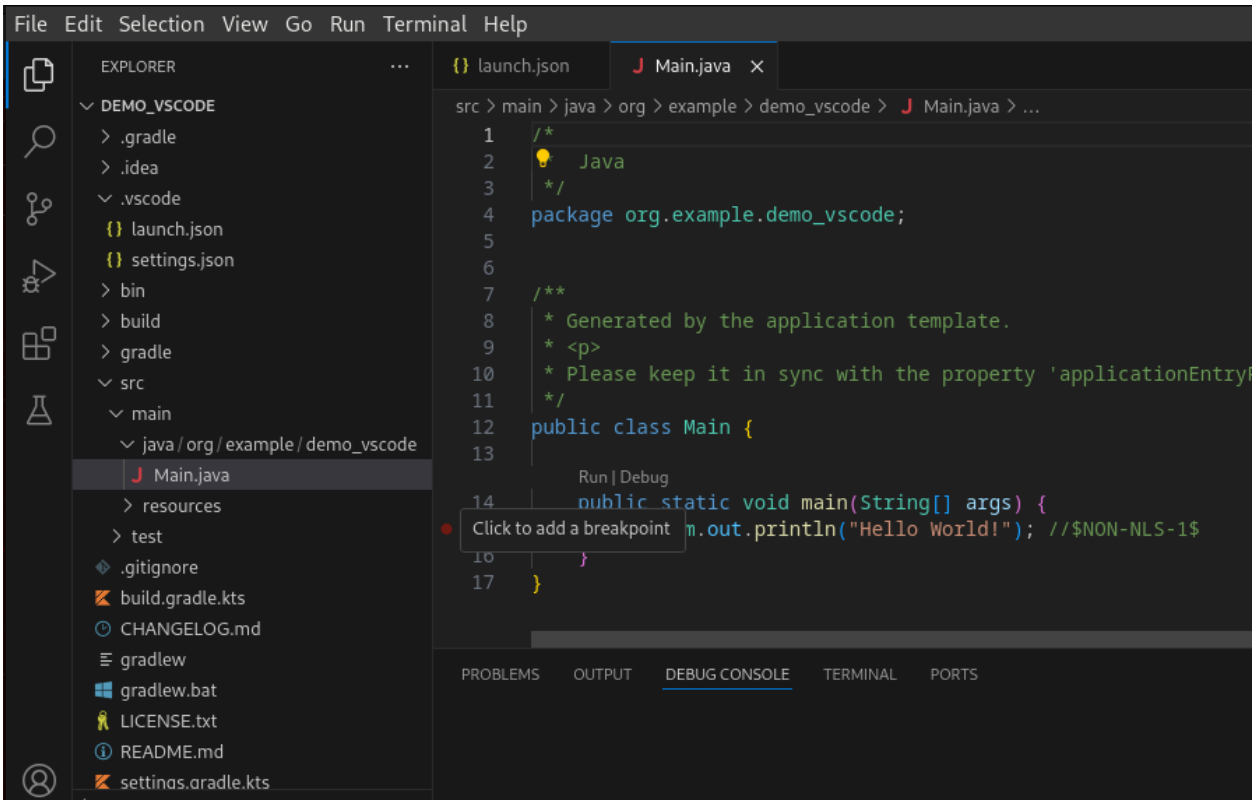
> **Warning:** IntelliJ IDEA and Android Studio need an Architecture 8.1 or higher for debug mode.

- Add a breakpoint in your Application code.
- Click on `Run` > `Edit Configurations...`.
- Click on `+` button (`Add New Configuration`).
- Select `Remote JVM Debug`.
- Click on the `New launch configuration` button.

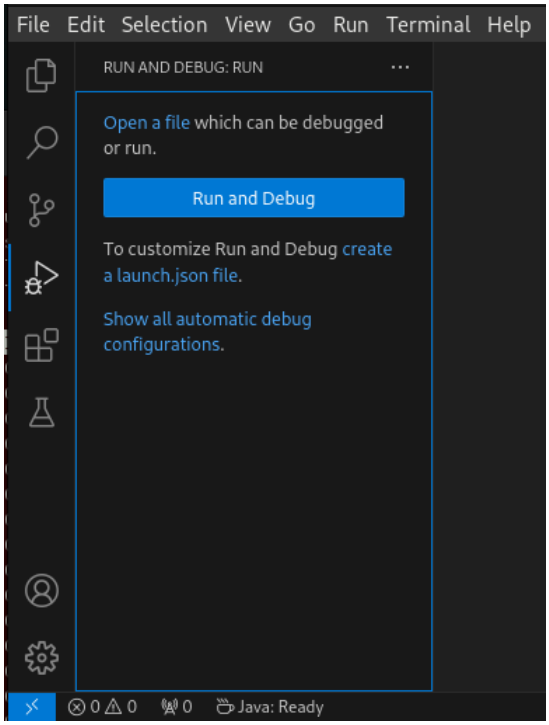
- Give a name to the launcher in the `Name` field.
- Set the debug host and port.
- Click on the `Debug` button.
- Add a breakpoint in your Application code.
- Click on `Run` > `Debug Configurations...` .
- Select `Remote Java Application` .
- Click on the `New launch configuration` button.
- Give a name to the launcher in the `Name` field.
- Set the debug host and port.
- Click on the `Debug` button.

Warning: VS Code needs an Architecture 8.1 or higher for debug mode.

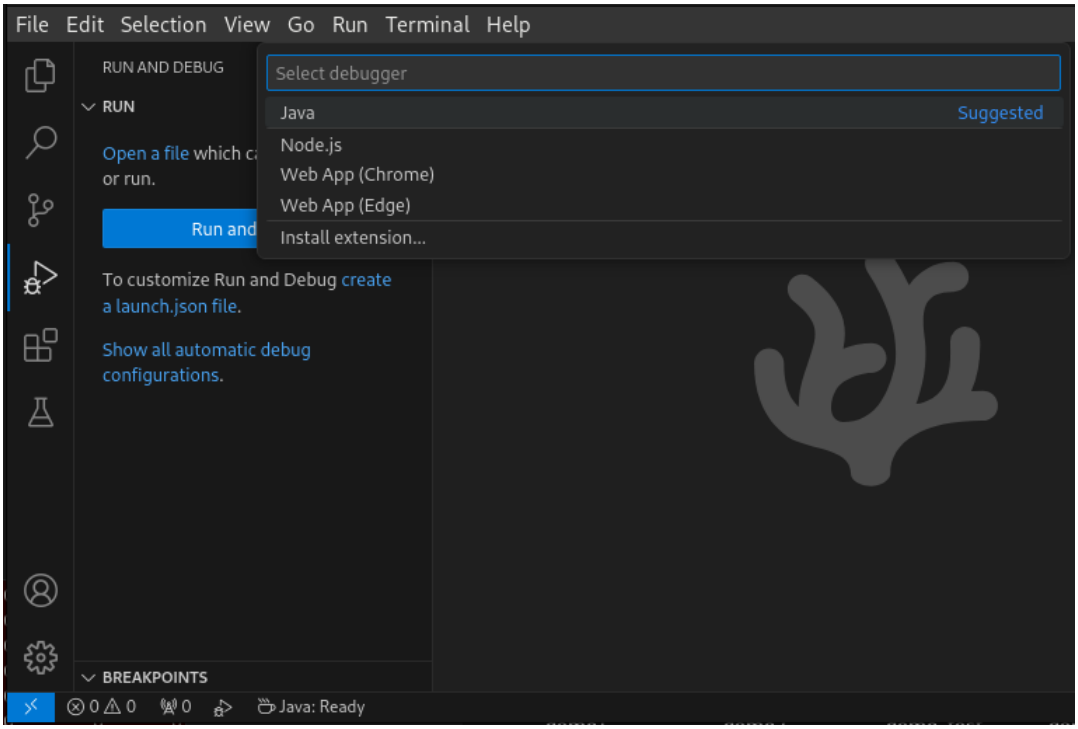
- Add a breakpoint in your Application code.



- Click on the `Run and Debug (Ctrl+Shift+D)` icon on the right panel.
- Click on create a launch.json file in the opened panel.

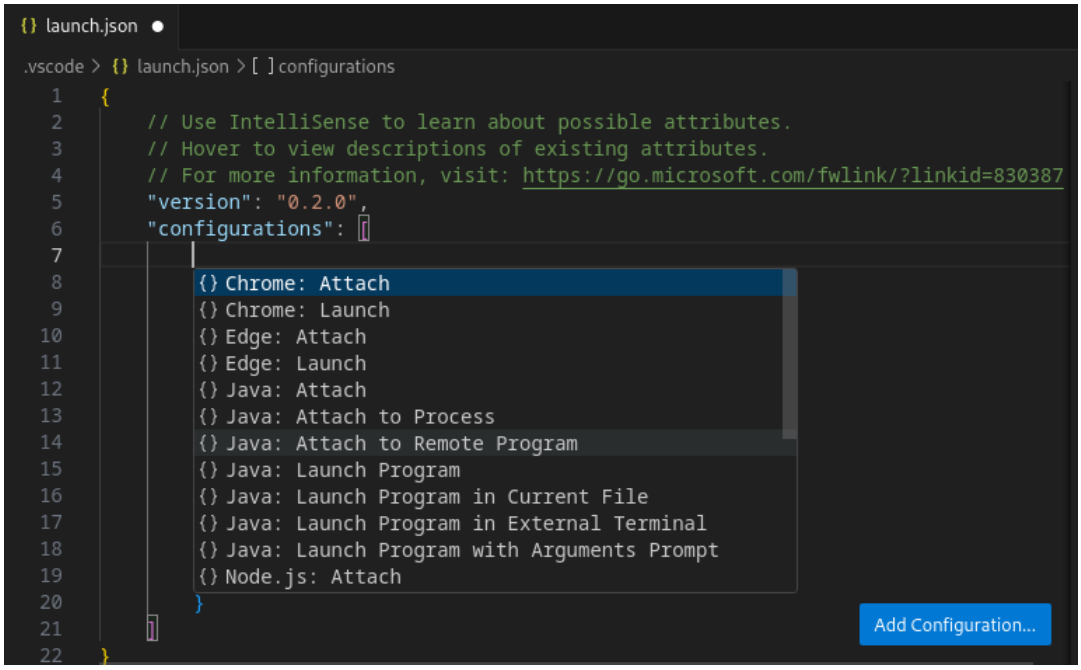


- Click on the Java entry proposed in the search field.

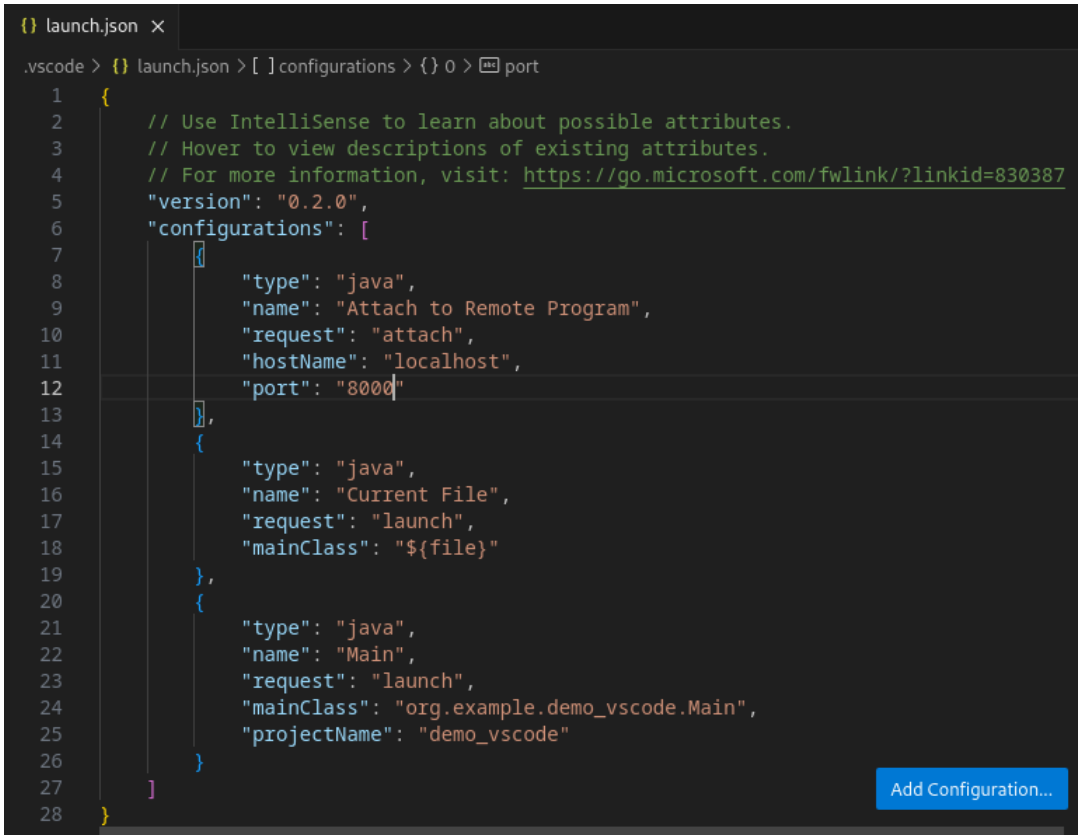


- Click on Add Configuration button

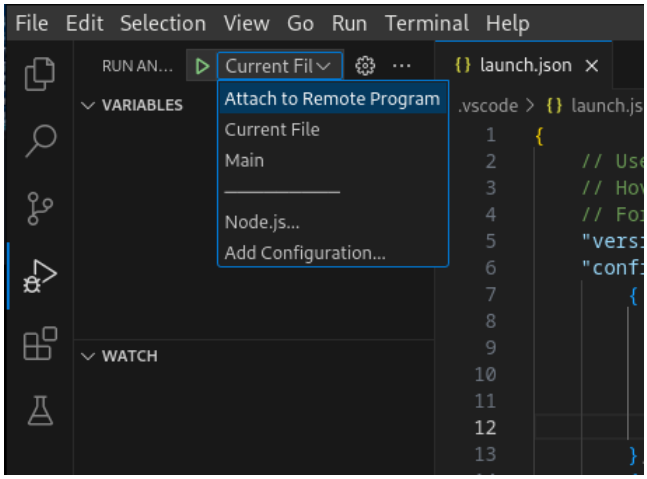
- Select {} Java: Attach to Remote Program entry in the popup list.



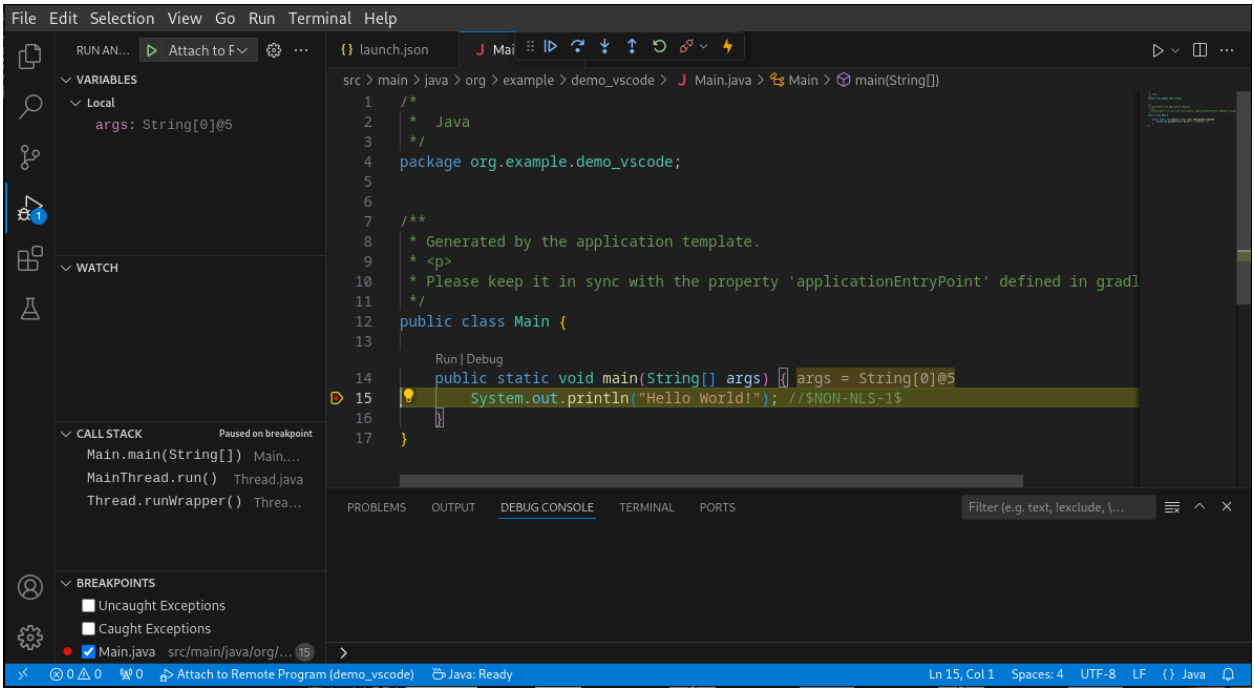
- Set localhost as hostName and the port (default is 1200) in the generated json.



- Select Attach to Remote Program in the selection box of the launcher.



- Click on the Start button



The debugger should connect to the Simulator and you should be able to debug your test.

Test on Device

The SDK allows to execute a testsuite on a device. This requires to:

- Have a VEE Port which implements the *BSP Connection*.
- Have a device connected to your workstation both for programming the Executable and getting the output traces. Consult your VEE Port specific documentation for setup.
- Start the *Serial to Socket Transmitter* tool if the VEE Port does not redirect execution traces.

The configuration is similar to the one used to execute a testsuite on the Simulator.

1. Follow the instructions to *setup a testsuite on the Simulator*.
2. In the build script file, replace the line:

   ```
   microej.useMicroejTestEngine(this)
   ```

   by:

   ```
   microej.useMicroejTestEngine(this, TestTarget.EMB)
   ```

3. Add the `import` statement at the beginning of the file:

   ```
   import com.microej.gradle.plugins.TestTarget
   ```

4. Add the required properties as follows:

   ```
   val test by getting(JvmTestSuite::class) {
     microej.useMicroejTestEngine(this, TestTarget.EMB)

     targets {
       all {
         testTask.configure {
           doFirst {
               systemProperties = mapOf(
                 // Enable the build of the Executable
                 "microej.testsuite.properties.deploy.bsp.microejscript" to "true",
                 "microej.testsuite.properties.microejtool.deploy.name" to "deployToolBSPRun",

                 // Configure the TCP/IP address and port if the VEE Port Run script does not redirect
   ↪execution traces
                 "microej.testsuite.properties.testsuite.trace.ip" to "localhost",
                 "microej.testsuite.properties.testsuite.trace.port" to "5555",
                 // Tell the testsuite engine that the VEE Port Run script redirects execution traces.
                 // Uncomment this line and comment the 2 lines above if the VEE Port supports it.
                 //"microej.testsuite.properties.launch.test.trace.file" to "true"
               )
           }
         }
       }
     }
   }
   ```

The properties are:

- `microej.testsuite.properties.deploy.bsp.microejscript` : enables the build of the Executable. It is required.
- `microej.testsuite.properties.microejtool.deploy.name` : name of the tool used to deploy the Executable to the board. It is required. It is generally set to `deployToolBSPRun` .

- microej.testsuite.properties.launch.test.trace.file : enables the redirection of the traces in file. If the VEE Port does not have this capability, the *Serial to Socket Transmitter* tool must be used to redirect the traces to a socket.
- microej.testsuite.properties.testsuite.trace.ip : TCP/IP address used by the *Serial to Socket Transmitter* tool to redirect traces from the board. This property is only required if the VEE Port does not redirect execution traces.
- microej.testsuite.properties.testsuite.trace.port : TCP/IP port used by the *Serial to Socket Transmitter* tool to redirect traces from the board. This property is only required if the VEE Port does not redirect execution traces.

Any other property can be passed to the Test Engine by prefixing it by `microej.testsuite.properties.`. For example, to set the the Immortals Heap size:

```
systemProperties = mapOf(
  "microej.testsuite.properties.core.memory.immortal.size" to "8192",
  ...
)
```

Test on Java SE VM

The SDK allows to run tests on a Java SE VM. This can be useful, for example, when the usage of mock libraries like Mockito is needed (this kind of library is not supported by the MicroEJ Core Engine).

There is nothing specific related to MicroEJ to run tests on a Java SE VM. Follow the Gradle documentation to setup such tests. As an example, here is a typical configuration to execute the tests located in the `src/test/java` folder:

```
testing {
  suites {
    val test by getting(JvmTestSuite::class) {
      useJUnitJupiter()

      dependencies {
        runtimeOnly("org.junit.platform:junit-platform-launcher:1.8.2")
      }
    }
  }
}
```

If you want to use Mockito, add it in the testsuite dependencies:

```
testing {
  suites {
    val test by getting(JvmTestSuite::class) {
      useJUnitJupiter()

      dependencies {
        implementation("org.mockito:mockito-core:4.11.0")
        runtimeOnly("org.junit.platform:junit-platform-launcher:1.8.2")
      }
    }
  }
}
```

Then you can use it in your test classes:

```
import org.junit.jupiter.api.Test;
import org.mockito.Mockito;

import static org.junit.jupiter.api.Assertions.assertNotNull;

public class MyTest {
   @Test
   public void test() {
      MyClass mock = Mockito.mock(MyClass.class);

      assertNotNull(mock);
   }
}
```

Testsuite Reports

SDK 6 1.1.0 and higher

SDK 6 1.0.0 and below

Once a testsuite is completed, the JUnit HTML report is generated in the module project location `build/reports/tests/<testsuite>/index.html`.

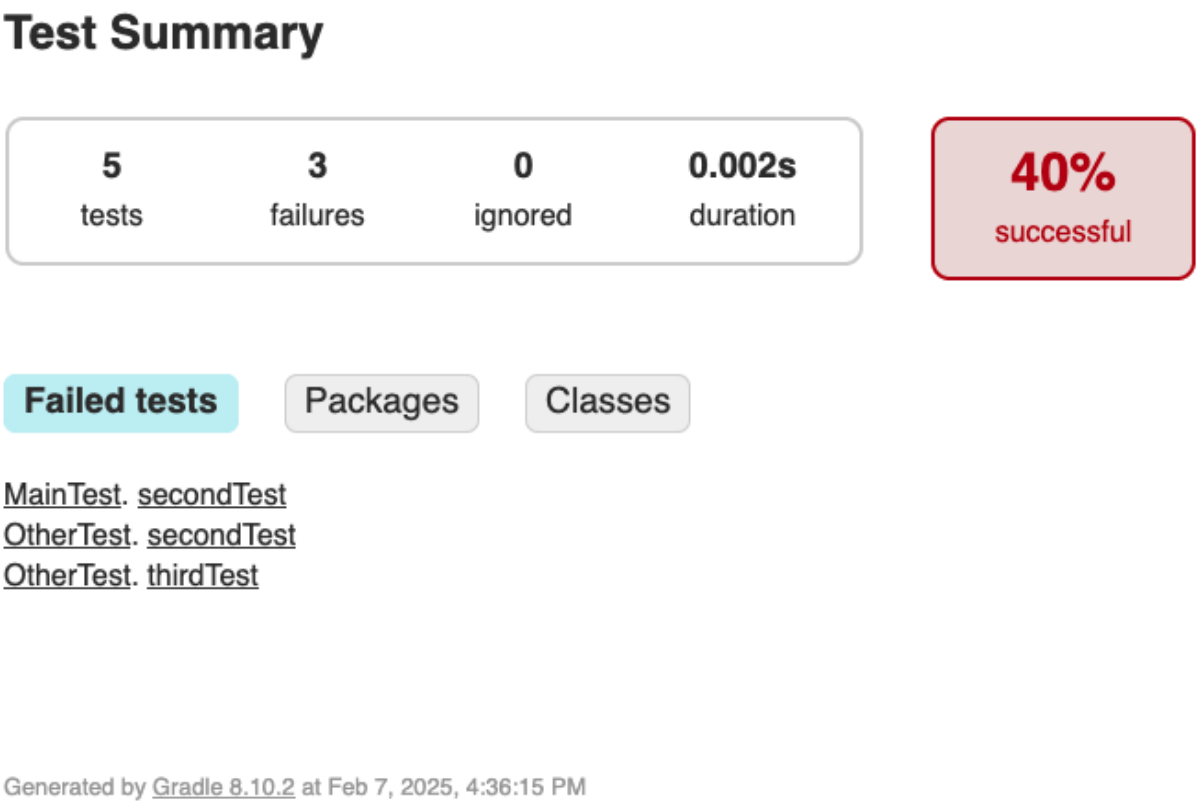


Fig. 54: Example of JUnit HTML Report

Once a testsuite is completed, the JUnit XML report is generated in the module project location `build/testsuite/`

output/<date>/testsuite-report.xml .

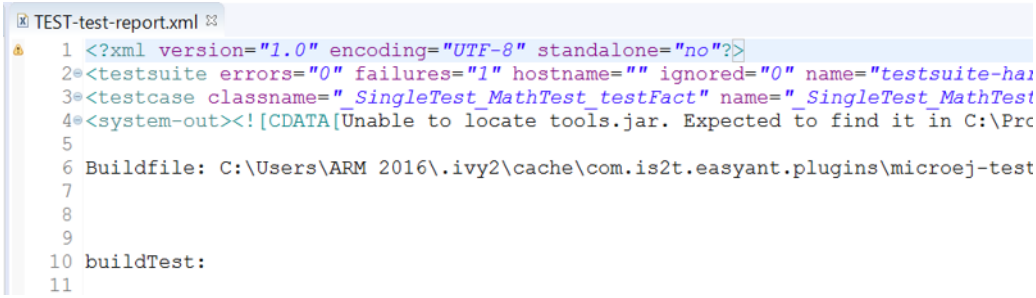


Fig. 55: Example of MicroEJ Testsuite XML Report

XML report file can also be opened In Eclipse in the JUnit View. Right-click on the file > Open With > JUnit View :

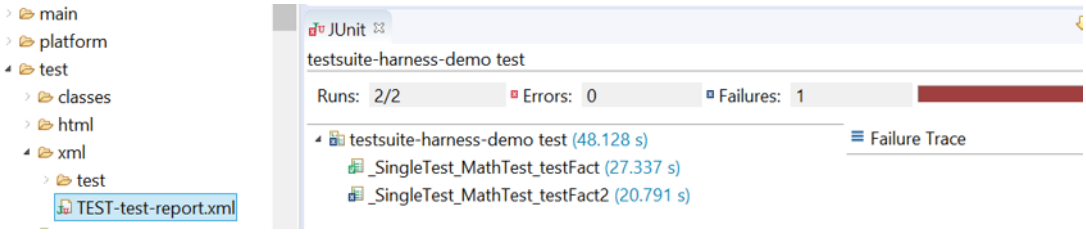


Fig. 56: Example of MicroEJ Testsuite XML Report in JUnit View

Publish Testsuite Reports

Starting from SDK 6 1.2.0 , it is possible to publish an archive file containing all testsuite reports of a project. By default, the tests are not executed when publishing a project, so you must explicitly run your testsuite to publish the reports:

```
./gradlew test publish
```

The published archive file contains the HTML and XML reports of all testsuites that have been executed. If your project contains *multiple testsuites*, you must execute each testsuite whose report must be published:

```
./gradlew testOnSim testOnJavaSE publish
```

You can also bind the check task to all your testsuites in the build.gradle.kts file of your project:

```
tasks.named("check") {
    dependsOn("testOnSim", "testOnJavaSE")
}
```

and execute the check task when publishing the project:

```
./gradlew check publish
```

Mixing tests

The SDK allows to define multiple testsuites on different targets. For example, you can configure a testsuite to run tests on the Simulator and a testsuite to run tests on a device.

Configuring multiple testsuites is almost only a matter of aggregating the testsuite declarations documented in the previous sections, as described in the Gradle documentation.

Mixing tests on the Simulator and on a device

If you need to define a testsuite to run on the Simulator and a testsuite to run on a device, the only point to take care is related to the tests source location, because:

- Gradle automatically uses the testsuite name to know the tests source folder to use. For example, for a testsuite named `test` (the built-in testsuite), the folder `src/test/java` is used, and for a testsuite named `testOnDevice`, the folder `src/testOnDevice/java` is used.
- Tests classes executed by the MicroEJ Test Engine on the Simulator and on device are not directly the tests source classes. The SDK generates new tests classes, based on the original ones, but compliant with the MicroEJ Testsuite mechanism. This process assumes by default that the tests classes are located in the `src/test/java` folder.

Therefore:

- It is recommended to use the built-in `test` testsuite for either the tests on the Simulator or the tests on device. This avoids extra configuration to change the location of the tests source folder.
- The tests source folder of the other testsuite must be changed to use the `src/test/java` folder as well:

```
testing {
  suites {
    val test by getting(JvmTestSuite::class) {
      microej.useMicroejTestEngine(this)

      dependencies {
        implementation(project())
        implementation("ej.library.test:junit:1.12.0")
      }
    }

    val testOnDevice by registering(JvmTestSuite::class) {
      microej.useMicroejTestEngine(this, TestTarget.EMB)

      sources {
        java {
          setSrcDirs(listOf(sourceSets.getByName(SourceSet.TEST_SOURCE_SET_NAME).java))
        }
        resources {
          setSrcDirs(listOf(sourceSets.getByName(SourceSet.TEST_SOURCE_SET_NAME).resources))
        }
      }

      dependencies {
        implementation(project())
        implementation("ej.library.test:junit:1.12.0")
      }

      targets {
```

(continues on next page)

(continued from previous page)

```
            all {
              testTask.configure {
                doFirst {
                  systemProperties = mapOf(
                    "microej.testsuite.properties.deploy.bsp.microejscript" to "true",
                    "microej.testsuite.properties.microejtool.deploy.name" to "deployToolBSPRun",
                    "microej.testsuite.properties.testsuite.trace.ip" to "localhost",
                    "microej.testsuite.properties.testsuite.trace.port" to "5555"
                  )
                }
              }
            }
          }
        }
      }
    }
```

The important part is the `sources` block of the `testOnDevice` testsuite. This allows to use the `src/test/java` and `src/test/resources` folders as the tests source folders.

With this configuration, all tests are executed on both the Simulator and the device. If you want to have different tests for each testsuite, it is recommended to separate the tests in different packages. For example the tests on the Simulator could be in `src/test/java/com/mycompany/sim` and the tests on the device could be in `src/test/java/com/mycompany/emb`. Then you use the test filtering capabilities to configure which package to run in which testsuite:

```
testing {
  suites {
    val test by getting(JvmTestSuite::class) {
      ...

      targets {
        all {
          testTask.configure {
            ...

            filter {
              includeTestsMatching("com.mycompany.sim.*")
            }
          }
        }
      }
    }

    val testOnDevice by registering(JvmTestSuite::class) {
      ...

      targets {
        all {
          testTask.configure {
            ...

            filter {
              includeTestsMatching("com.mycompany.emb.*")
            }
          }
```

(continues on next page)

(continued from previous page)

```
        }
      }
    }
  }
}
```

Mixing tests on the Simulator and on a Java SE VM

Defining tests on the Simulator and on a Java SE VM is only a matter of aggregating the configuration of each test-suite:

```
testing {
  suites {
    val test by getting(JvmTestSuite::class) {
      microej.useMicroejTestEngine(this)
      ...
    }

    val testOnJavaSE by registering(JvmTestSuite::class) {
      useJUnitJupiter()

      dependencies {
        runtimeOnly("org.junit.platform:junit-platform-launcher:1.8.2")
      }

      ...
    }
  }
}
```

As explained in the previous section, it is recommended to use the built-in `test` testsuite for the tests on the Simulator since it avoids adding confguration to change the tests sources folder. With this configuration, tests on the Simulator are located in the `src/test/java` folder, and tests on a Java SE VM are located in the `src/testOnJavaSe/java` folder.

Configure the Testsuite Engine

The engine used to execute the testsuite provides a set of configuration parameters that can be defined with System Properties. For example, to set the timeout of the tests:

- In the command line with `-D` :

```
./gradlew test -Dmicroej.testsuite.timeout=120
```

- In the build script file:

```
testing {
  suites {
    val test by getting(JvmTestSuite::class) {
      ...

      targets {
        all {
```

(continues on next page)

(continued from previous page)

```
            testTask.configure {
               ...

               doFirst {
                  systemProperties = mapOf(
                     "microej.testsuite.timeout" to "120"
                  )
               }
            }
         }
      }
   }
}
```

The following configuration parameters are available:

Name	Description	Default
microej.testsuite.timeout	The time in seconds before a test is considered as failed. Set it to 0 to disable the timeout.	60
microej.testsuite.jvmArgs	The arguments to pass to the Java SE VM started for each test.	Not set
microej.testsuite.lockPort	The localhost port used by the framework to synchronize its execution with other frameworks on same computer. Synchronization is not performed when this port is 0 or negative.	0
microej.testsuite.retry.count	A test execution may not be able to produce the success trace for an external reason, for example an unreliable harness script that may lose some trace characters or crop the end of the trace. For all these unlikely reasons, it is possible to configure the number of retries before a test is considered to have failed.	0
microej.testsuite.retry.if	Regular expression checked against the test output to retry the test. If the regular expression is found in the test output, the test is retried (up to the value of microej.testsuite.retry.count).	Not set
microej.testsuite.retry.unless	Regular expression checked against the test output to retry the test. If the regular expression is not found in the test output, the test is retried (up to the value of microej.testsuite.retry.count).	Not set
microej.testsuite.verbose.level	Verbose level of the testsuite output. Available values are error , warning , info , verbose and debug . Deprecated since version 1.2.0: The testsuite verbose level follows Gradle log level.	info
microej.testsuite.status.pattern	**Requires SDK 6 1.3.0 and ej.library.test:junit:1.12.0 minimum.** Pattern of the test passed (default is .:[|PASSED|]:.) and failed (default is .:[|FAILED|]:.) tags in testsuite logs. These tags are catched by the testsuite engine to determine if a test has passed or failed. The {} placeholder in the pattern will be replaced by PASSED or FAILED respectively in order to discriminate these tags from other test logs.	.:[|{}|]:.

Inject Application Options

Standalone Application Options can be defined to configure the Application or Library being tested. They can be defined globally, to be applied on all tests, or specifically to a test.

Inject Application Options Globally

In order to define an Application Option globally, it must be prefixed by `microej.testsuite.properties.` and passed as a System Property, either in the command line or in the build script file. For example, to inject the property `core.memory.immortal.size`:

- In the command line with `-D`:

```
./gradlew test -Dmicroej.testsuite.properties.core.memory.immortal.size=8192
```

- In the build script file:

```
testing {
  suites {
    val test by getting(JvmTestSuite::class) {
      ...

      targets {
        all {
          testTask.configure {
            ...

            doFirst {
              systemProperties = mapOf(
                "microej.testsuite.properties.core.memory.immortal.size" to "8192"
              )
            }
          }
        }
      }
    }
  }
}
```

Inject Application Options For a Specific Test

In order to define an Application Option for a specific test, it must be set in a file with the same name as the test case file, but with the `.properties` extension instead of the `.java` extension. The file must be put in the `src/test/resources` folder and within the same package than the test file.

For example, to inject an Application Option for the test class `MyTest` located in the `com.mycompany` package, a `MyTest.properties` file must be created. Its path must be: `src/test/resources/com/mycompany/MyTest.properties`.

Application Options defined in this file do not require the `microej.testsuite.properties.` prefix.

Note: If the testsuite is configured to execute main classes (thanks to the parameter `TestMode.MAIN`):

```
microej.useMicroejTestEngine(this, TestTarget.SIM, TestMode.MAIN)
```

the properties file must be named after the main class. If the main class has been generated from a JUnit test class, its class name is prefixed by `__AllTests__`.

Inject Resources For a Specific Test

To embed resources for a specific test (image, model, ...), set the application.resources property in the test properties file (See *Inject Application Options For a Specific Test*). For example:

```
application.resources=/samples/sample0.txt;/samples/sample1.json;/samples/sample2.bin
```

Each path is relative to the resources folder root. Use a semicolon (;) to separate multiple resources.

Testsuite Advanced Configuration

Configure a VEE by Testsuite

The VEE declared in the project dependencies with the microejVee or the testMicroejVee configuration (refer to *Configure the VEE* for more details) is used to execute all testsuites. If your project contains a testsuite other than the built-in test testsuite, it is also possible to run the testsuite on a dedicated VEE. To define a VEE for your custom testsuite, you must:

- Create a new <testsuite_name>MicroejVee configuration depending on your testsuite name in the build.gradle.kts file of your project. For example:

```
configurations.create("testOnDeviceMicroejVee") {
    isCanBeConsumed = false
    isCanBeResolved = false
    isTransitive = false
}
```

- Declare the VEE in the project dependencies with your new configuration:

```
dependencies {
    ...
    "testOnDeviceMicroejVee"("com.mycompany:vee-port:1.0.0")
}
```

3.4.16 Publish a Project

Publishing is the process by which the built artifacts of a module is made available to other modules or any other systems.

The requirements to publish a module are:

- Defining the name of the module. It is set by default to the name of the module folder, and can be changed in the settings.gradle.kts file located at the root of the module, thanks to the property rootProject.name :

```
rootProject.name = "myModule"
```

- Defining the group and version properties. They can be set in the build.gradle.kts file:

```
group = "com.mycompany"
version = "1.0.0"
```

> **Warning:** Refer to *Manage Versioning page* to know how to version your module.

- Declaring a `maven` publication repository. This can be done in the build file for example, with:

```
publishing {
    repositories {
        maven {
            name = "mavenPublish"
            url = uri("https://my.server/repository")
        }
    }
}
```

  Refer to the official documentation for more information on publication repositories.

Then the publication of a module to a repository is achieved by executing the `publish` task:

```
$ ./gradlew publish
```

The following artifacts are automatically published:

- The main artifact, which is the JAR file for Application and Add-On Library natures.
- The README.md file.
- The CHANGELOG.md file.
- The LICENSE.txt file.
- The ASSEMBLY_EXCEPTION.txt file.
- The Gradle module descriptor file.
- The Ivy descriptor file (to allow SDK 5 project to fetch it).
- The WPK file, if the project is an Application.

Note: Ivy descriptor publication can be disabled. You can refer to *disable Ivy Descriptor publication How-to page* if you don't need to publish Ivy descriptor.

3.4.17 Development Tools

MicroEJ provides a number of tools to assist with various aspects of development. These tools are either command line tools or Eclipse IDE plugins.

Command line tools

Command line tools can be executed using the Gradle task `execTool`.

The format of the task is as follow:

```
./gradlew execTool --name=TOOL_NAME --toolProperty=PROPERTY="VALUE" --
↪toolProperty=PROPERTY="VALUE" ...
```

The task required option `--name` is used to describe the name of the tool to execute. The options `--toolProperty` are used to configure the tool's options.

Note: The task options can also be defined in your IDE by creating a new Configuration. For more information, you can refer to *How To Create a Custom Configuration in the IDE*.

In addition, the tool's options can be defined in `configuration/tools/TOOL_NAME.properties`.

The following sections describe the command line tools and their options:

Stack Trace Reader

Principle

Stack Trace Reader is a MicroEJ tool that reads and decodes the MicroEJ stack traces. When an exception occurs, the *Core Engine* prints the stack trace on the standard output `System.out`. The class names, non-required types names(see *Types*), and method names obtained are encoded with a MicroEJ internal format. This internal format prevents embedding all class names and method names in the executable image to save some memory space. The Stack Trace Reader tool allows you to decode the stack traces by replacing the internal class names and method names with their real names. It also retrieves the line numbers in the Application.

Functional Description

The Stack Trace Reader reads the debug information from the fully linked ELF file (the ELF file that contains the Core Engine, the other libraries, the BSP, the OS, and the compiled Application). It prints the decoded stack trace.

When *Multi-Sandbox capability* is enabled, the Stack Trace Reader can simultaneously decode heterogeneous stack traces with lines owned by different Sandboxed Applications and the Kernel. Lines owned by the Kernel can be decoded with the Executable debug information file (optionally made available by your Kernel provider).

Usage

Tip: If you don't have an Application to test yet, you can create an *Application project* with the following example main class:

```
package com.mycompany;

public class MyApp {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
        new Exception().printStackTrace();
    }
}
```

On successful deployment (by executing the `runOnDevice` task), the Application is started on the device and the trace is dumped on the standard output.

```
MicroEJ START
Hello, World!
Exception in thread "main" @C:0x8070c30@
    at @C:0x8070c60@.@M:0x8075850:0x807585a@
    at @C:0x8070c30@.@M:0x80769a4:0x80769ba@
    at @C:0x8070c30@.@M:0x807857c:0x8078596@
    at @C:0x8070c00@.@M:0x8074e04:0x8074e1a@
    at @C:0x8070ce0@.@M:0x807601c:0x807603c@
    at @C:0x806fe10@.@M:0x807779c:0x80777b0@
    at @C:0x8070c00@.@M:0x8077b40:0x8077b4c@
    at @C:0x8070c00@.@M:0x80779b0:0x80779bb@
```

Decoding an Application stack trace depends on the SDK 6 version used in the project.

In an Application project

SDK 6 1.1.0 and higher

SDK 6 1.0.0 and below

If you have an Application project and want to decode a stack trace for this Application, you can execute the tool in the project.

Follow these steps:

- Make sure to build the Executable before executing the tool, by running either the buildExecutable or the runOnDevice task. The Executable file produced by the Application project is automatically used if it exists.
- When the Executable of the Application project is built, execute the stackTraceReader task, either from your IDE, or as a command line:

```
./gradlew stackTraceReader "--console=plain"
```

- When the message [INFO] Paste the MicroEJ core engine stack trace here. is displayed, paste the encoded stacktrace into the console. The Stack Trace Reader immediately displays the decoded stack trace:

```
================ [ MicroEJ Core Engine Trace ] ================
console:
[INFO] Paste the MicroEJ core engine stack trace here.
   MicroEJ START
   Hello, World!
   Exception in thread "main" @C:0x8070c30@
      at @C:0x8070c60@.@M:0x8075850:0x807585a@
      at @C:0x8070c30@.@M:0x80769a4:0x80769ba@
      at @C:0x8070c30@.@M:0x807857c:0x8078596@
      at @C:0x8070c00@.@M:0x8074e04:0x8074e1a@
      at @C:0x8070ce0@.@M:0x807601c:0x807603c@
      at @C:0x806fe10@.@M:0x807779c:0x80777b0@
      at @C:0x8070c00@.@M:0x8077b40:0x8077b4c@
      at @C:0x8070c00@.@M:0x80779b0:0x80779bb@
   MicroEJ START
   Hello, World!
   Exception in thread "main" java.lang.Throwable
      at java.lang.System.getStackTrace(Unknown Source)
      at java.lang.Throwable.fillInStackTrace(Throwable.java:82)
      at java.lang.Throwable.<init>(Throwable.java:32)
      at java.lang.Thread.dumpStack(Thread.java:573)
      at com.microej.Main.main(Main.java:22)
      at java.lang.MainThread.run(Thread.java:855)
      at java.lang.Thread.runWrapper(Thread.java:464)
      at java.lang.Thread.callWrapper(Thread.java:449)
```

The Stack Trace Reader tool interacts with the console by default. See the *Configure* chapter to learn about the other modes and configurations available.

If you have an Application project and want to decode a stack trace for this Application, you can execute the tool in the project.

Follow these steps:

- Make sure to build the Executable before executing the tool, by running either the buildExecutable or the runOnDevice task.
- When the Executable of the Application project is built, execute the execTool task:

Warning: This tool requires to use Gradle **8.8** maximum. If you want to use a higher Gradle version, upgrade the SDK 6 version used in the project to 1.1.0 minimum and use the stackTraceReader task.

```
./gradlew execTool --name=stackTraceDecrypter \
--toolProperty=proxy.connection.connection.type="console" \
--toolProperty=application.file="../../application/executable/application.out" \
--toolProperty=additional.application.files="" \
--console plain
```

Note: It is also possible to create a custom task of type ExecToolTask dedicated to the Stack Trace Reader. Refer to the *How To Create Custom ExecTool Task* chapter for more information.

Paste the previous trace dump into the console. The output of the Stack Trace Reader is the following:

```
================ [ MicroEJ Core Engine Trace ] ================
console:
[INFO] Paste the MicroEJ core engine stack trace here.
    MicroEJ START
    Hello World from Gradle!
    Exception in thread "main" @C:0x8070c30@
        at @C:0x8070c60@.@M:0x8075850:0x807585a@
        at @C:0x8070c30@.@M:0x80769a4:0x80769ba@
        at @C:0x8070c30@.@M:0x807857c:0x8078596@
        at @C:0x8070c00@.@M:0x8074e04:0x8074e1a@
        at @C:0x8070ce0@.@M:0x807601c:0x807603c@
        at @C:0x806fe10@.@M:0x807779c:0x80777b0@
        at @C:0x8070c00@.@M:0x8077b40:0x8077b4c@
        at @C:0x8070c00@.@M:0x80779b0:0x80779bb@
    MicroEJ START
    Hello World from Gradle!
    Exception in thread "main" java.lang.Throwable
        at java.lang.System.getStackTrace(Unknown Source)
        at java.lang.Throwable.fillInStackTrace(Throwable.java:82)
        at java.lang.Throwable.<init>(Throwable.java:32)
        at java.lang.Thread.dumpStack(Thread.java:573)
        at com.microej.Main.main(Main.java:22)
        at java.lang.MainThread.run(Thread.java:855)
        at java.lang.Thread.runWrapper(Thread.java:464)
        at java.lang.Thread.callWrapper(Thread.java:449)
```

Custom Executable File Location

SDK 6 1.1.0 and higher

SDK 6 1.0.0 and below

If you want to decode a stack trace of a different Executable file than the Application project one, you must set the `application.file` System Property to define the Executable file location:

```
./gradlew stackTraceReader -D"application.file"="/path/to/my/application.out" "--console=plain"
```

As shown in the above example, you can define the location of the Executable file with the `application.file` property:

```
./gradlew execTool --name=stackTraceDecrypter \
--toolProperty=proxy.connection.connection.type="console" \
--toolProperty=application.file="../../application/executable/application.out" \
--toolProperty=additional.application.files="" \
--console plain
```

In a Multi-Sandboxed Application project

SDK 6 1.5.0 and higher

SDK 6 1.4.0 and below

Using SDK 6 1.5.0 or higher, the Stack Trace Reader tool can decode stack traces of Multi-Sandboxed Applications. You can still define the additional application files using the property:

```
./gradlew stackTraceReader -D"additional.application.files"="/path/to/my/app1.fodbg,/path/to/my/app2.fodbg"
↪"--console=plain"
```

If you want to decode a stack trace of a Multi-Sandboxed Application, you must set the `additional.application.files` System Property to define the list of the additional application files (`.fodbg` files) separated by commas:

```
./gradlew execTool --name=stackTraceDecrypter \
--toolProperty=proxy.connection.connection.type="console" \
--toolProperty=application.file="./application.out" \
--toolProperty=additional.application.files="/path/to/my/app.fodbg" \
--console plain
```

Configure

SDK 6 1.1.0 and higher

SDK 6 1.0.0 and below

The Stack Trace Reader tool uses by default the console to communicate with the device, but this can be changed by setting the `stackTraceReaderConnectionMode` property in the `microej` block in the `build.gradle.kts` file:

```
microej {
    stackTraceReaderConnectionMode = "file"
    stackTraceReaderFilePath = "/path/to/input/file"
    stackTraceReaderFileResultPath = "/path/to/output/file"
}
```

Each mode has dedicated additional options. The list of the available modes, with their dedicated options, are:

- console (default mode) : use the standard input/output.
- file
  [use files.]
  - stackTraceReaderFilePath : Path to the file containing the encoded stack trace.
  - stackTraceReaderFileResultPath : Path to the output file for the decoded stack trace.
- uart (**Not yet available**) : use Serial communication.
- socket
  [use a socket.]
  - stackTraceReaderSocketAddress : IP address.
  - stackTraceReaderSocketPort : IP port.

For example, here is the configuration to use the socket mode on the address 192.168.1.17 and the port 4000 :

```
microej {
   stackTraceReaderConnectionMode = "socket"
   stackTraceReaderUartPort = "192.168.1.17"
   stackTraceReaderUartBaudRate = "4000"
}
```

The Stack Trace Reader tool uses by default the console to communicate with the device, but this can be changed by setting the proxy.connection.connection.type property:

```
./gradlew execTool --name=stackTraceDecrypter \
--toolProperty=proxy.connection.connection.type="file" \
--toolProperty=application.file="../../application/executable/application.out" \
--toolProperty=additional.application.files="" \
--console plain
```

Each mode has dedicated additional options. The list of the available modes, with their dedicated options, are:

- console (default mode) : use the standard input/output.
- file
  [use files.]
  - pcboardconnection.file.path : Path to the file containing the encoded stack trace.
  - pcboardconnection.result.file : Path to the output file for the decoded stack trace.
- uart (**Not yet available**) : use Serial communication.
- socket
  [use a socket.]
  - pcboardconnection.socket.address : IP address.
  - pcboardconnection.socket.port : IP port.

Code Coverage Analyzer

Principle

The Simulator features an option to output .cc (Code Coverage) files that represent the use rate of functions of an application. It traces how the opcodes are really executed.

Functional Description

The Code Coverage Analyzer scans the output .cc files, and outputs an HTML report to ease the analysis of methods coverage. The HTML report is available in a folder named htmlReport in the same folder as the .cc files.

Dependencies

In order to work properly, the Code Coverage Analyzer should input the .cc files (See *Generate Code Coverage*). The .cc files rely on the classpath used during the execution of the Simulator to the Code Coverage Analyzer. Therefore the classpath is considered to be a dependency of the Code Coverage Analyzer.

Installation

This tool is a built-in Architecture tool.

Use

A MicroEJ tool is available to launch the Code Coverage Analyzer tool. The tool name is Code Coverage Analyzer.

Two levels of code analysis are provided, the Java level and the bytecode level. Also provided is a view of the fully or partially covered classes and methods. From the HTML report index, just use hyperlinks to navigate into the report and source / bytecode level code.

```
./gradlew execTool --name=codeCoverageAnalyzer \
   --toolProperty=cc.dir="/MODULE_PATH/build/output/com.company.Main/cc/" \
   --toolProperty=cc.includes="com.company.*" \
   --toolProperty=cc.excludes="com.company.MainWrapper" \
   --toolProperty=cc.src.folders="/MODULE_PATH/src/main/java" \
   --toolProperty=cc.html.dir="/MODULE_PATH/cc"
```

Adapt the values of the cc.includes and cc.excludes to your project. The cc.excludes should contain at least the wrapper class generated by the SDK, which is named after the following pattern: <applicationEntryPoint>Wrapper .

Note: It is also possible to create a custom task of type ExecToolTask dedicated to the Code Coverage Analyzer. Refer to the *How To Create Custom ExecTool Task* chapter for more information.

Options

Option: *.cc files folder

Option Name: cc.dir

Description:

Specify a folder which contains the cc files to process (*.cc).

Option: Source Folders

Option Name: cc.src.folders

Description:

A list of folders to the source files.

Option: HTML Dir

Option Name: cc.html.dir

Description:

The output directory for the HTML report.

Option: Includes

Option Name: cc.includes

Description:

List packages and classes to include to code coverage report (; separated). If no package/class is specified, all classes found in the project classpath will be analyzed.

Examples:

packageA.packageB.* : includes all classes which are in package packageA.packageB

packageA.packageB.className : includes the class packageA.packageB.className

Option: Excludes

Option Name: cc.excludes

Description:

List packages and classes to exclude to code coverage report (; separated). If no package/class is specified, all classes found in the project classpath will be analyzed.

Examples:

packageA.packageB.* : excludes all classes which are in package packageA.packageB

packageA.packageB.className : excludes the class packageA.packageB.className **IDE tools**

Eclipse IDE tools are graphical tools which are available as Eclipse plugins: Memory Map Analyzer, Heap Analyzer and Font Designer.

Follow these steps to install the latest stable version of these tools:

- Install Eclipse IDE for Java Developers - Minimum supported version is 2022-03 .
- In Eclipse, go to Help > Eclipse Marketplace… .
- In the Find field, type MicroEJ Tools , then press Enter .
- Click on the Install button of the MicroEJ Tools plugin.
- Accept the license, then click on the Finish button.
- In the upcoming Trust Artifacts window, check the Unsigned item and click on Trust Selected button.

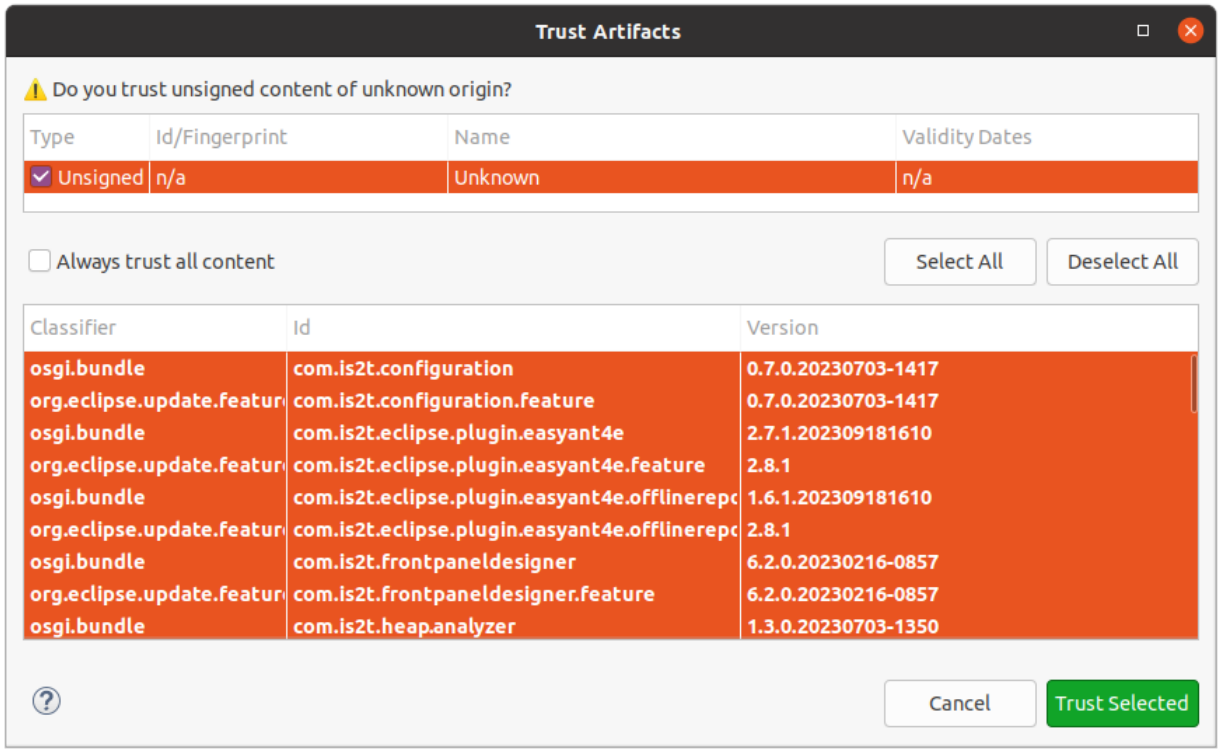


Fig. 57: Eclipse Plugin Installation - Trust Artifacts

- In the upcoming window, click on the Restart Now button.

The following sections describe the IDE tools and their options:

Memory Map Analyzer

Principle

When the Executable of an Application is built, a Memory Map file is generated. This file can be visualized with the Memory Map Analyzer, an Eclipse IDE plug-in. It displays the memory consumption of different features in the RAM and ROM.

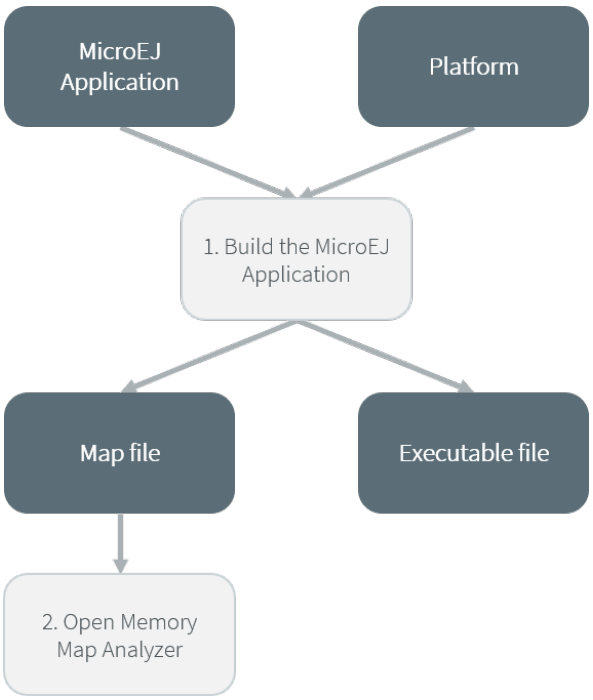


Fig. 58: Memory Map Analyzer Process

Use

When the Executable file of an Application has been produced, the Memory Map file is available at build/application/object/soar/<main-class-full-name>.map .

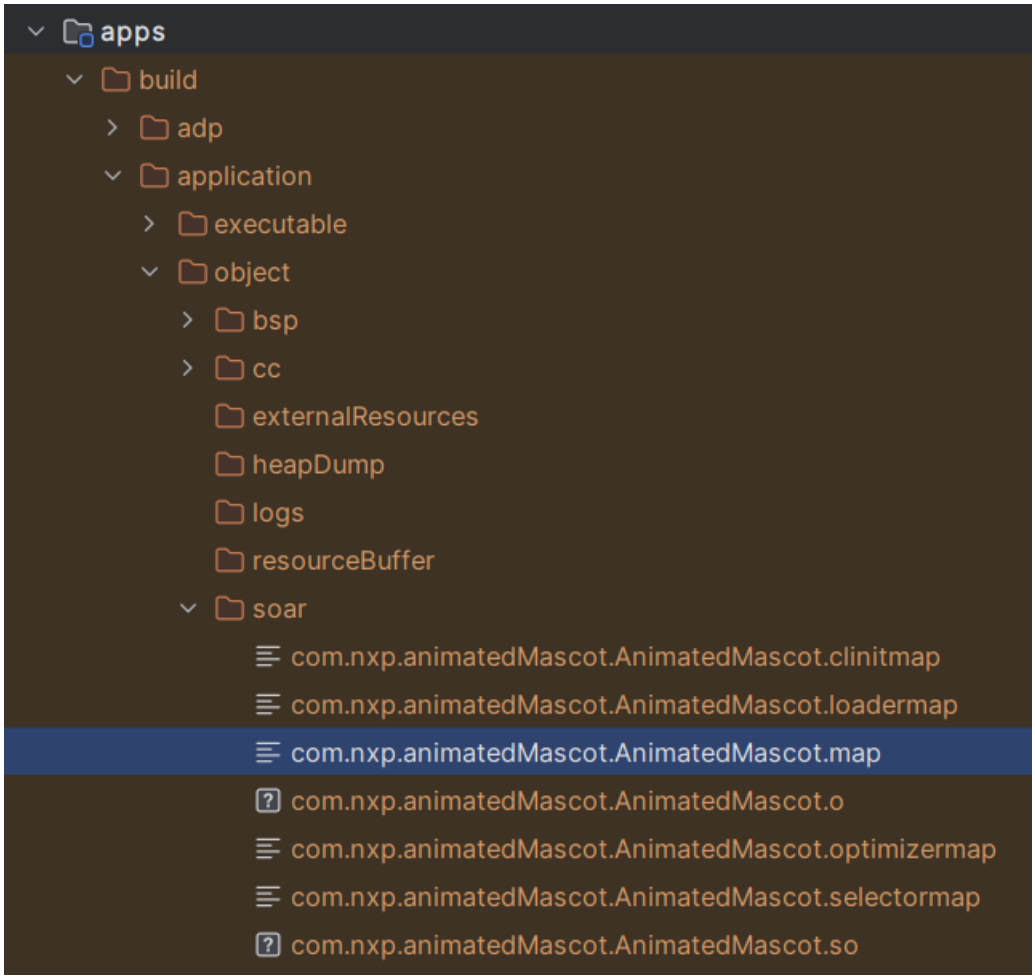


Fig. 59: Memory Map File

You can visualize it by following these steps:

- Make sure *the Eclipse IDE is installed with the required plugin*, then launch it.
- Click on File > Open File… .
- Select the Memory Map file.

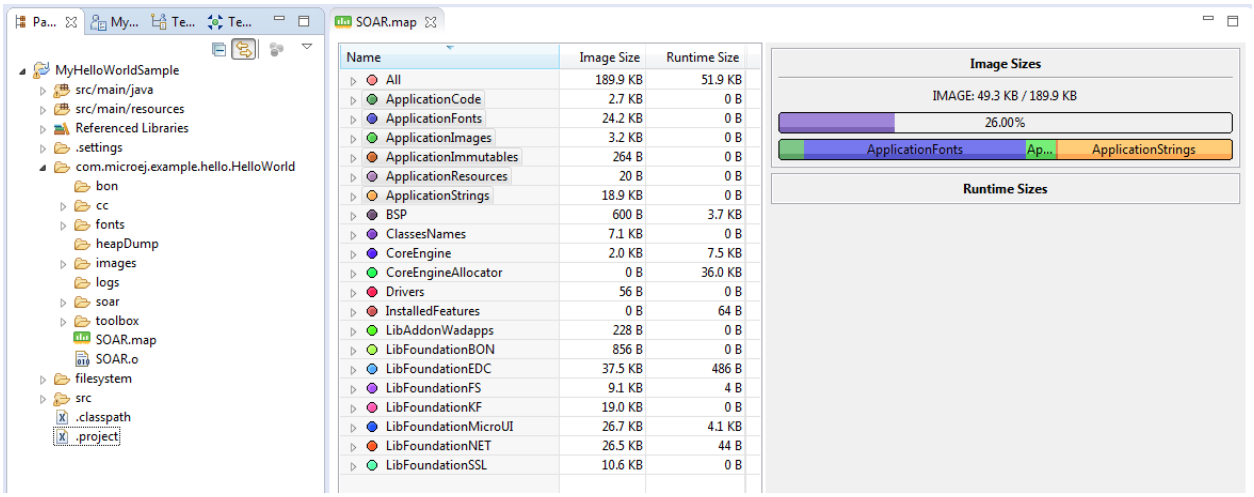


Fig. 60: Consult Full Memory

You can select an item (or several) to show the memory used by this item(s) on the right, or select All to show the memory used by all items. This special item performs the same action as selecting all items in the list.

You can also select an item in the list, and expand it to see all symbols used by the item. This view is useful in understanding why a symbol is embedded.

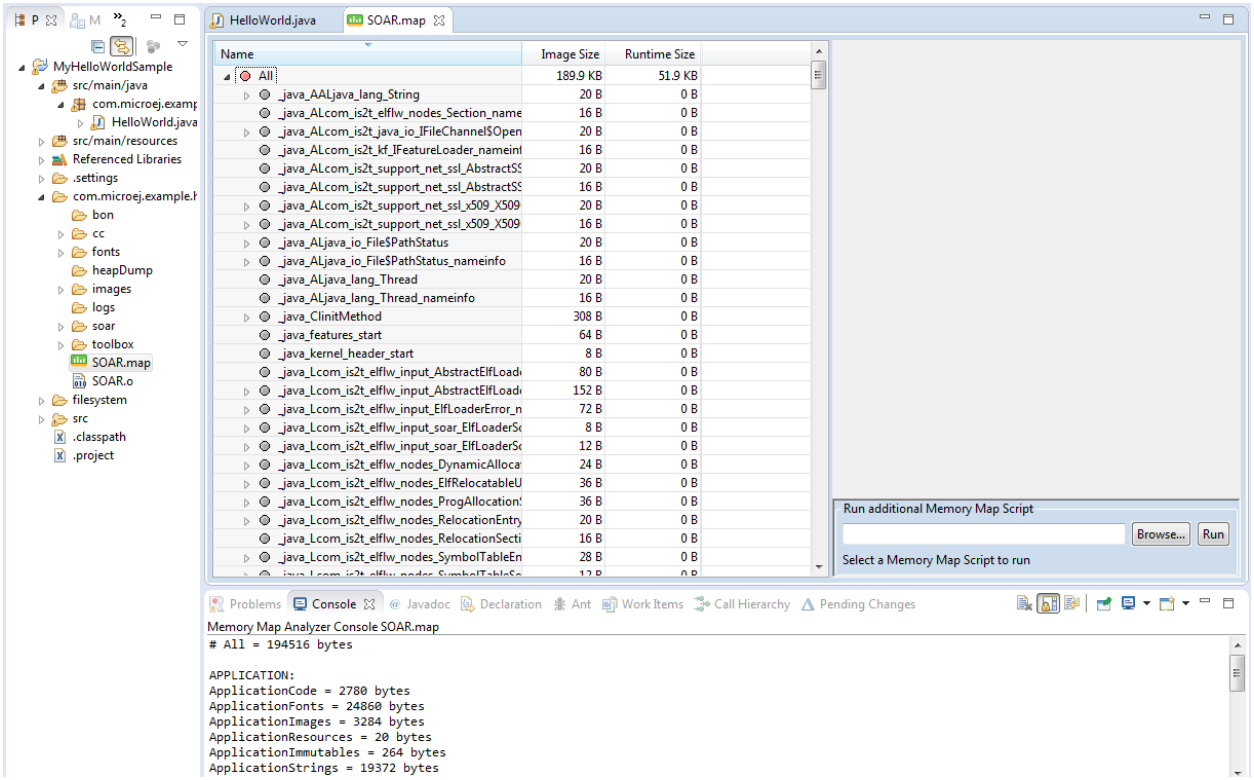


Fig. 61: Detailed View

Troubleshooting

Memory Map Analyzer may fail while opening a Memory Map file saying *Could not read this file*.

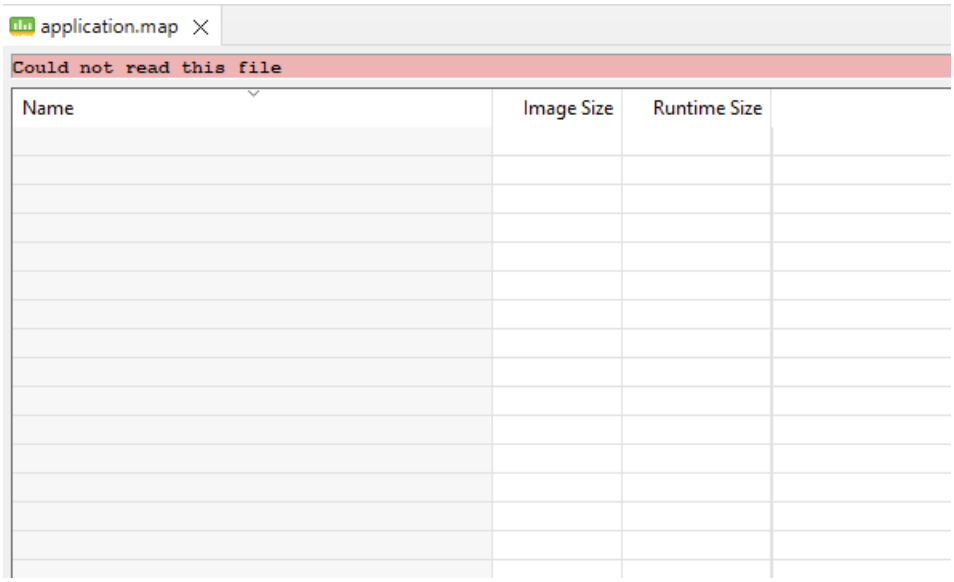


Fig. 62: Error displayed when Memory Map file could not be read

This may happen if the Memory Map Analyzer cannot find a VEE Port to interpret the Memory Map file. To workaround this, you can open your Memory Map file with a text editor and add the following line inside the lscfmap block:

```
<property name="jpf.dir" value="C:\path\to\your\vee"/>
```

For example:

```
<lscmap name="default">
  <property name="jpf.dir" value="C:\path\to\your\vee"/>
  <property name="vendorUrl" value="http://developer.microej.com/4.0/sdk/license"/>
  ...
```

If a line defining the jpf.dir already exists, check if the value corresponds to a valid path.

Note: The path must be the directory of a built VEE Port. If you build a Kernel Application, the VEE Port will be available in the build/vee directory of your Gradle project.

Heap Dumper & Heap Analyzer

Introduction

Heap Dumper is a tool that allows to get a snapshot of the heap of an Application running on the Simulator or on a device.

The Heap Analyzer is a set of tools to help developers understand the contents of the Managed Heap and find problems such as memory leaks. For its part, the Heap Analyzer IDE plugin is able to visualize dump files. It helps you analyze their contents thanks to the following features:

- memory leaks detection
- objects instances browse
- heap usage optimization (using immortal or immutable objects)

The Heap

The heap is a memory area used to hold Java objects created at runtime. Objects persist in the heap until they are garbage collected. An object becomes eligible for garbage collection when there are no longer any references to it from other objects.

Heap Dump

A heap dump is an XML file (with the `.heap` extension) that provides a snapshot of the heap contents at the moment the file is created. It contains a list of all the instances of both class and array types that exist in the heap. For each instance, it records:

- The time at which the instance was created
- The thread that created it
- The method that created it

For instances of class types, it also records:

- The class
- The values in the instance's non-static fields

For instances of array types, it also records:

- The type of the contents of the array
- The contents of the array

For each referenced class type, it records the values in the static fields of the class.

Heap Analyzer Tools

The Heap Analyzer is an Eclipse IDE plugin that adds three tools to the MicroEJ environment.

Tool name	Number of input files	Purpose
Heap Viewer	1	Shows what instances are in the heap, when they were created, and attempts to identify problem areas
Progressive Heap Usage	1 or more	Shows how the number of instances in the heap has changed over time
Compare	2	Compares two heap dumps, showing which objects were created, or garbage collected, or have changed values

Heap Dumper

The Heap Dumper generates `.heap` files. There are two implementations:

- the one integrated to the Simulator: it directly dumps `.heap` files from the Managed Heap.
- the Heap Dumper tool: it generates `.heap` files from `.hex` files that must be manually retrieved from the device.

The heap dump should be performed after a call to System.gc() to exclude discardable objects.

Simulator

In order to generate a Heap dump of an Application running on the Simulator:

- Set the `s3.inspect.heap` Application properties to `true`.
- Update your Application code to call the System.gc() method where you need a Heap dump.
- run the Application on the Simulator.

When the System.gc() method is called:

- if it is called from the Application, the `.heap` file is generated in the `build/output/<fqnMainClass>/heapDump/` folder of the project, where `<fqnMainClass>` is the Fully Qualified Name of the Application Main class, for example (`com.mycompany.Main`).
- if it is called from a test class, the `.heap` file is generated in the `build/testsuite/output/<buildDate>/bin/<fqnMainClass>/heapDump/` folder of the project, where `<fqnMainClass>` is the Fully Qualified Name of the generated Main class and `<buildDate>` is the date of the test execution, for example (`build/testsuite/output/20240625-1605-24/bin/com.mycompany._AllTests_MyTest/heapDump/`).

Device

In order to generate a Heap dump of an Application running on a device:

- Update your Application code to call the System.gc() method where you need a Heap dump.
- Build the Executable and deploy it on the device.
- Start a debug session.
- Add a breakpoint to `LLMJVM_on_Runtime_gc_done` Core Engine hook. This function is called by the Core Engine when System.gc() method is done. Alternatively, if you are experiencing out of memory errors, you can directly add a breakpoint to the `LLMJVM_on_OutOfMemoryError_thrown` Core Engine hook.
- Resume the execution until the breakpoint is reached. You are now ready to dump the memory. Next steps are:
  - *Retrieve the hex file from the device*
  - *Extract the Heap dump from the hex file*

Note: Core Engine hooks may have been inlined by the third-party linker. If the symbol is not accessible to the debugger, you must declare them in your VEE Port:

```
void LLMJVM_on_Runtime_gc_done(){
  //No need to add code to the function
}

void LLMJVM_on_OutOfMemoryError_thrown(){
  //No need to add code to the function
}
```

Retrieve the `.hex` file from the device

If you are in a Mono-Sandbox context and do not use dynamic Heap configuration:

You only have to dump the Core Engine heap section. Here is an example of GDB commands:

```
b LLMJVM_on_Runtime_gc_done
b LLMJVM_on_OutOfMemoryError_thrown
continue
dump ihex memory heap.hex &_java_heap_start &_java_heap_end
```

Otherwise, you must:

- generate the *VEE memory dump script* which will dump all the required sections.
- *Dump the memory of the running Executable*
- Depending on if you are using a Mono-Sandbox or a Multi-Sandbox :
  - In a Mono-Sandbox context and not using *dynamic Heap configuration*, use `{N}_java_heap.hex`.
  - If you are in a Multi-Sandbox context or using *dynamic Heap configuration* you have to merge all `.hex` files with:

Command Prompt

PowerShell

Bash

```
copy /b *.hex memory.hex
```

```
Get-ChildItem *.hex | Get-Content -Raw | Set-Content -NoNewline memory.hex
```

```
cat *.hex > memory.hex
```

You now have a single .hex file and can continue to the next section.

Extract the Heap dump from the .hex file

In order to extract the Heap dump from an .hex file, run the execTool Gradle task with the tool name heapDumperPlatform :

```
./gradlew execTool --name=heapDumperPlatform \
  --toolProperty=output.name="application.heap" \
  --toolProperty=application.filename="../../executable/application/application.out" \
  --toolProperty=heap.filename="/path/to/memory.hex" \
  --toolProperty=additional.application.filenames="" \
  --console plain
```

If you are in a Multi-Sandbox context, you have to include the .fodbg files and additional hex files:

```
./gradlew execTool --name=heapDumperPlatform \
  --toolProperty=output.name="application.heap" \
  --toolProperty=application.filename="../../executable/application/application.out" \
  --toolProperty=heap.filename="/path/to/memory.hex" \
  --toolProperty=additional.application.filenames="/path/to/app1.fodbg;/path/to/app2.fodbg..." \
  --toolProperty=additional.memory.filenames="/path/to/additonal1.hex;/path/to/additional2.hex..." \
  --console plain
```

Note: It is also possible to create a custom task of type ExecToolTask dedicated to the heapDumperPlatform tool. Refer to the *How To Create Custom ExecTool Task* chapter for more information.

You can find the list of available options below:

Name	Description	Default
application.filename	Specify the full path of the Executable file, a full linked ELF file.	Not set
additional.application.filenames	Specify the full path of Feature files with debug information (.fodbg files).	Not set
heap.filename	Specify the full path of heap memory dump, in Intel Hex format.	Not set
additional.memory.filenames	Specify the full path of additional memory files in Intel Hex format (Installed Feature areas, Dynamic Features table, …).	Not set
output.name	Name of the extracted Heap dump file.	application.heap

Heap Viewer

To open the Heap Viewer tool (*Included in MicroEJ Tools*), select a heap dump XML file in the Package Explorer, right-click on it and select Open With > Heap Viewer

Alternatively, right-click on it and select Heap Analyzer > Open heap viewer.

This will open a Heap Viewer tool window for the selected heap dump[1].

The Heap Viewer works in conjunction with two views:

1. The Outline view
2. The Instance Browser view

These views are described below.

The Heap Viewer tool has three tabs, each described below.

Outline View

The Outline view shows a list of all the types in the heap dump, and for each type shows a list of the instances of that type. When an instance is selected it also shows a list of the instances that refer to that instance. The Outline view is opened automatically when an Heap Viewer is opened.

[1] Although this is an Eclipse editor, it is not possible to edit the contents of the heap dump.

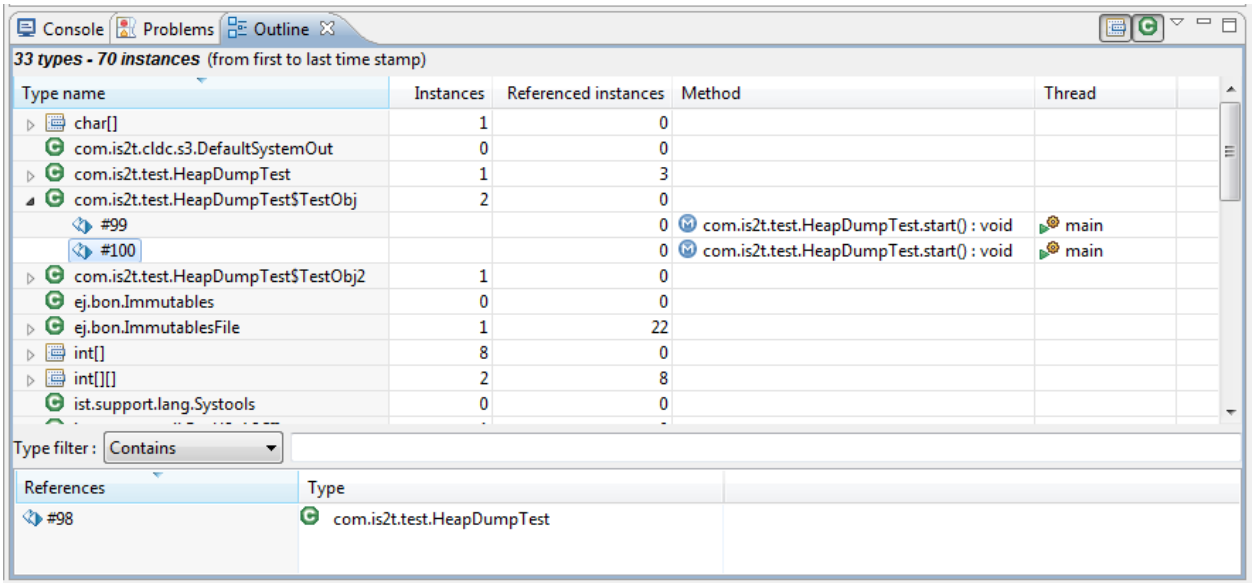


Fig. 63: Outline View

Instance Browser View

The Instance Browser view opens automatically when a type or instance is selected in the Outline view. It has two modes, selected using the buttons in the top right corner of the view. In Fields mode it shows the field values for the selected type or instance, and where those fields hold references it shows the fields of the referenced instance, and so on. In Reference mode it shows the instances that refer to the selected instance, and the instances that refer to them, and so on.

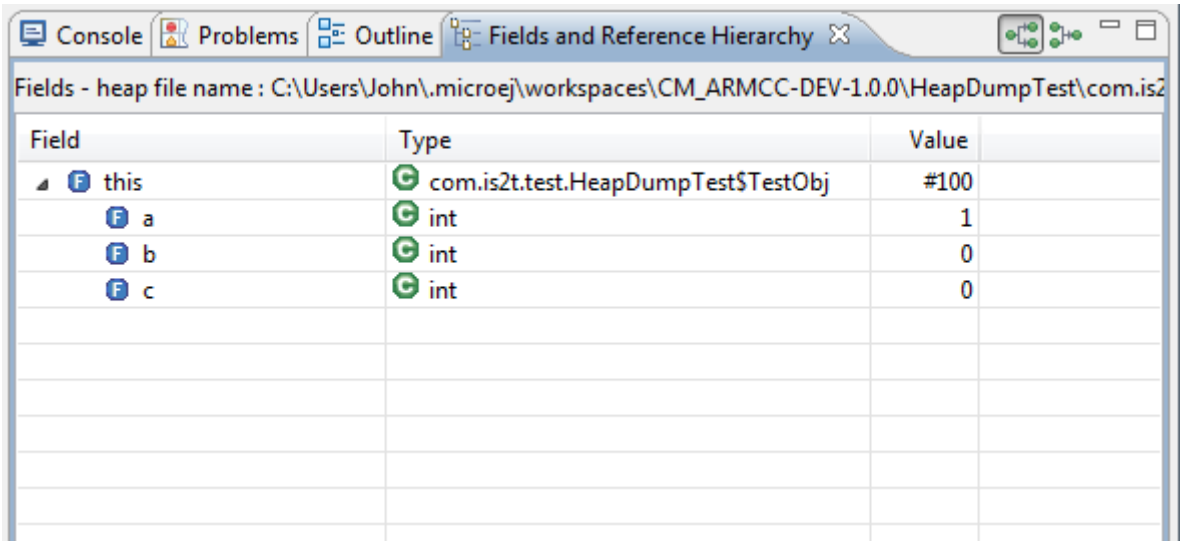


Fig. 64: Instance Browser View - Fields mode

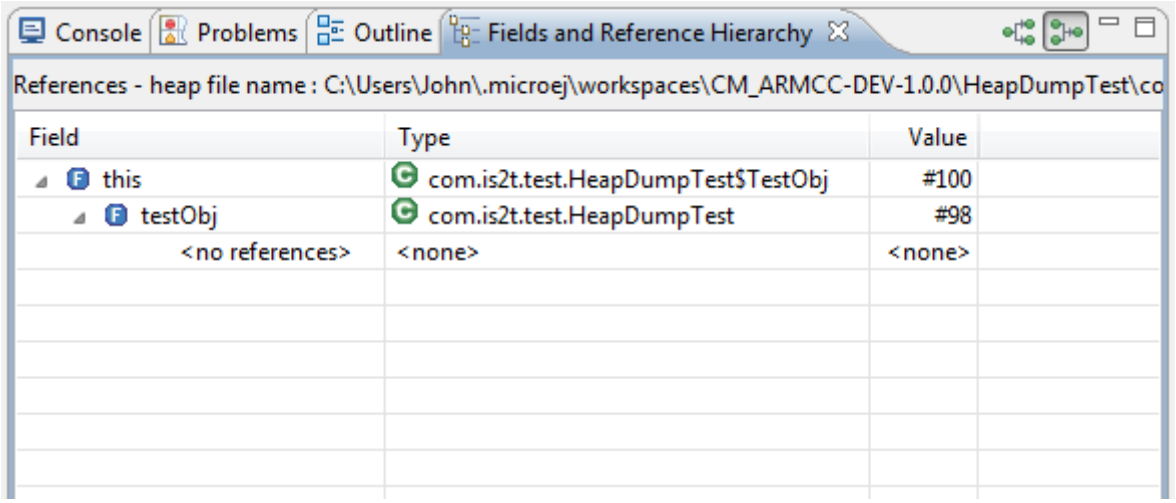


Fig. 65: Instance Browser View - References mode

Heap Usage Tab

The Heap usage page of the Heap Viewer displays four bar charts. Each chart divides the total time span of the heap dump (from the time stamp of the earliest instance creation to the time stamp of the latest instance creation) into a number of periods along the x axis, and shows, by means of a vertical bar, the number of instances created during the period.

- The top-left chart shows the total number of instances created in each period, and is the only chart displayed when the Heap Viewer is first opened.
- When a type or instance is selected in the Outline view the top-right chart is displayed. This chart shows the number of instances of the selected type created in each time period.
- When an instance is selected in the Outline view the bottom-left chart is displayed. This chart shows the number of instances created in each time period by the thread that created the selected instance.
- When an instance is selected in the Outline view the bottom-right chart is displayed. This chart shows the number of instances created in each time period by the method that created the selected instance.

Fig. 66: Heap Viewer - Heap Usage Tab

Clicking on the graph area in a chart restricts the Outline view to just the types and instances that were created during the selected time period. Clicking on a chart but outside of the graph area restores the Outline view to showing all types and instances[2].

The button Generate graphViz file in the top-right corner of the Heap Usage page generates a file compatible with graphviz (www.graphviz.org).

The section *Heap Usage Monitoring* shows how to compute the maximum heap usage.

[2] The Outline can also be restored by selecting the All types and instances option on the drop-down menu at the top of the Outline view.

Dominator Tree Tab

The Dominator tree page of the Heap Viewer allows the user to browse the instance reference tree which contains the greatest number of instances. This can be useful when investigating a memory leak because this tree is likely to contain the instances that should have been garbage collected.

The page contains two tree viewers. The top viewer shows the instances that make up the tree, starting with the root. The left column shows the ids of the instances – initially just the root instance is shown. The Shallow instances column shows the number of instances directly referenced by the instance, and the Referenced instances column shows the total number of instances below this point in the tree (all descendants).

The bottom viewer groups the instances that make up the tree either according to their type, the thread that created them, or the method that created them.

Double-clicking an instance in either viewer opens the Instance Browser view (if not already open) and shows details of the instance in that view.

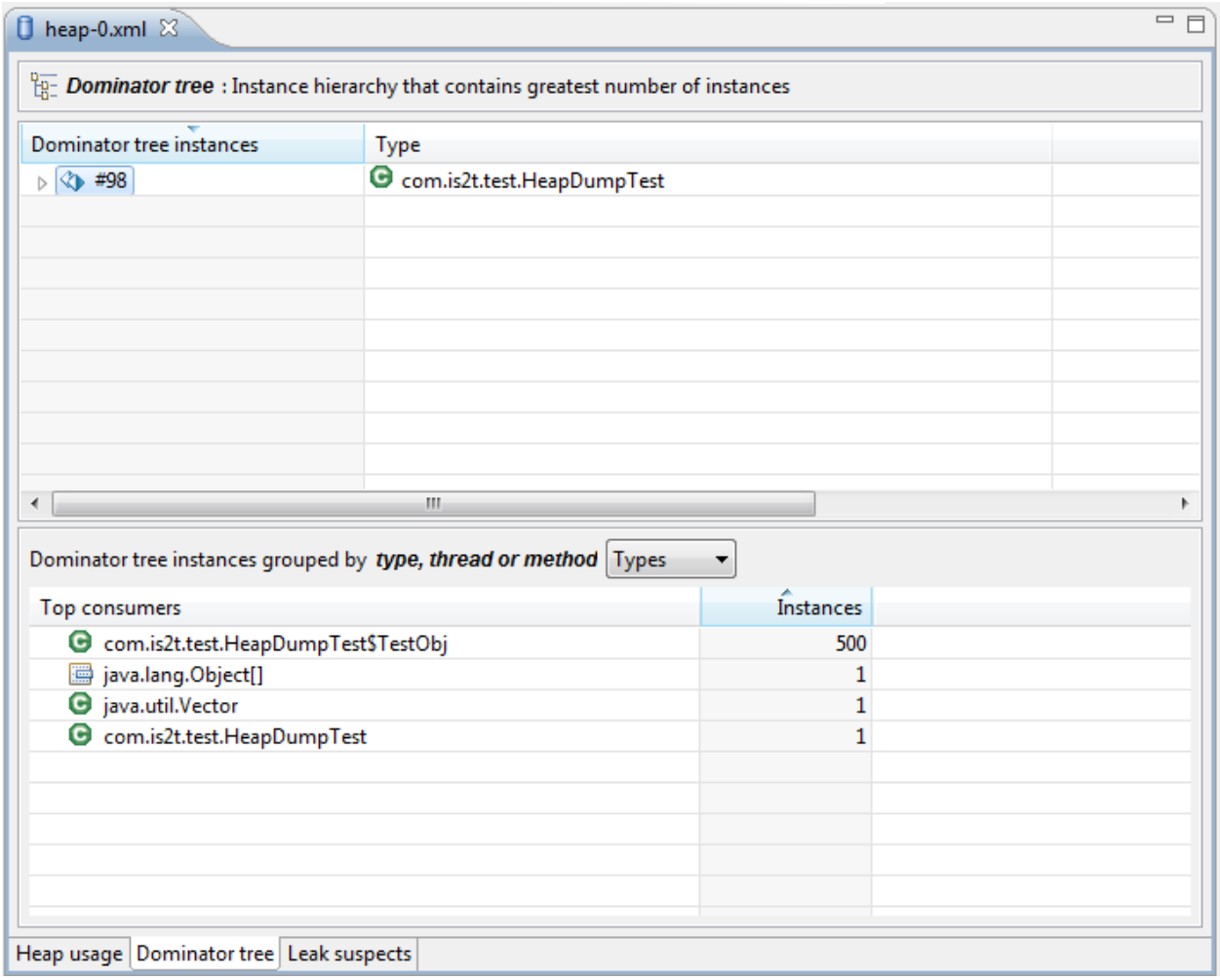


Fig. 67: Heap Viewer - Dominator Tree Tab

Leak Suspects Tab

The Leak suspects page of the Heap Viewer shows the result of applying heuristics to the relationships between instances in the heap to identify possible memory leaks.

The page is in three parts.

- The top part lists the suspected types (classes). Suspected types are classes which, based on numbers of instances and instance creation frequency, may be implicated in a memory leak.
- The middle part lists accumulation points. An accumulation point is an instance that references a high number of instances of a type that may be implicated in a memory leak.
- The bottom part lists the instances accumulated at an accumulation point.

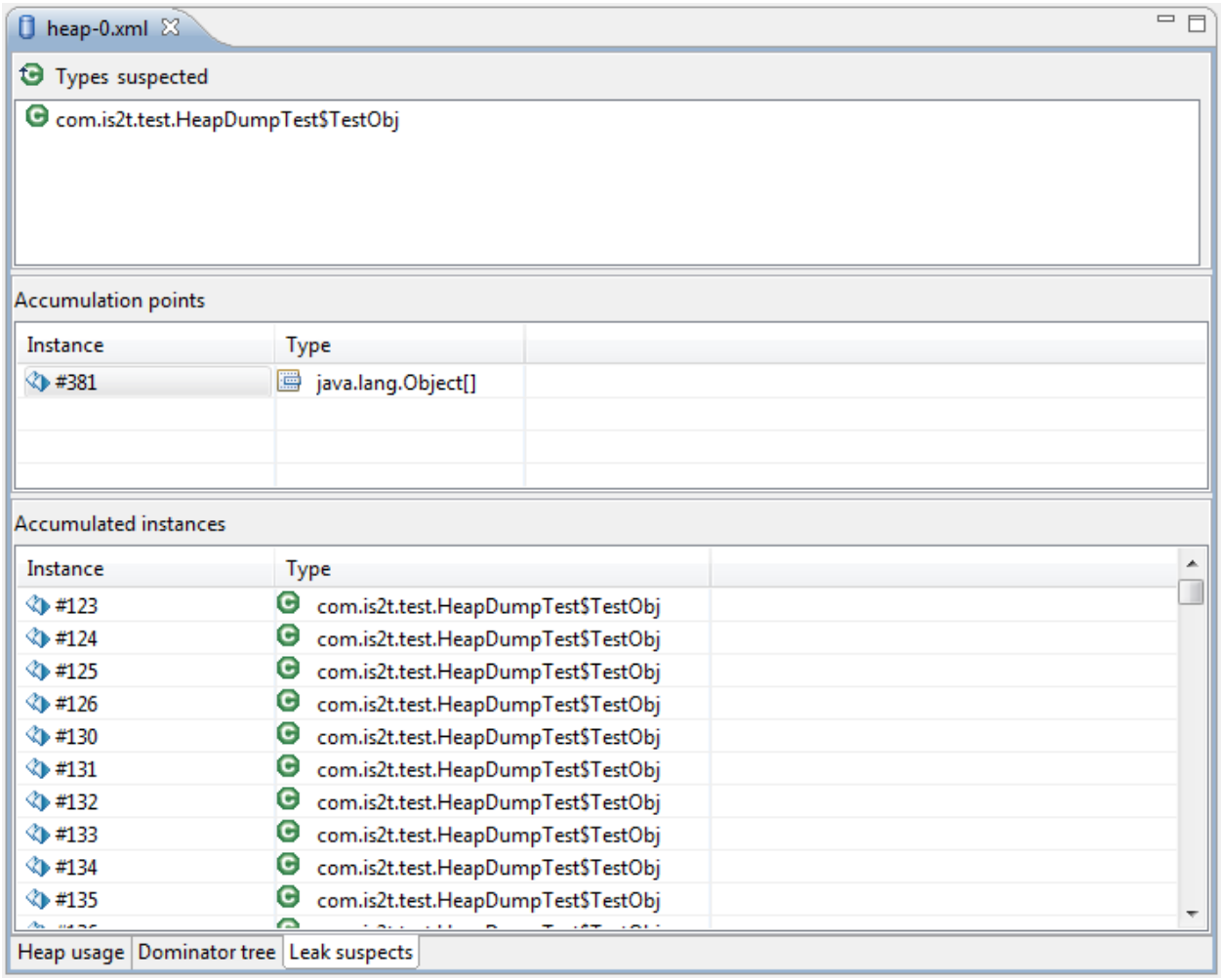


Fig. 68: Heap Viewer - Leak Suspects Tab

Progressive Heap Usage

To open the Progressive Heap Usage tool, select one or more heap dump XML files in the Package Explorer, right-click and select Heap Analyzer > Show progressive heap usage

This tool is much simpler than the Heap Viewer described above. It comprises three parts.

- The top-right part is a line graph showing the total number of instances in the heap over time, based on the creation times of the instances found in the heap dumps.
- The left part is a pane with three tabs, one showing a list of types in the heap dump, another a list of threads that created instances in the heap dump, and the third a list of methods that created instances in the heap dump.
- The bottom-left is a line graph showing the number of instances in the heap over time restricted to those instances that match with the selection in the left pane. If a type is selected, the graph shows only instances of that type; if a thread is selected the graph shows only instances created by that thread; if a method is selected the graph shows only instances created by that method.

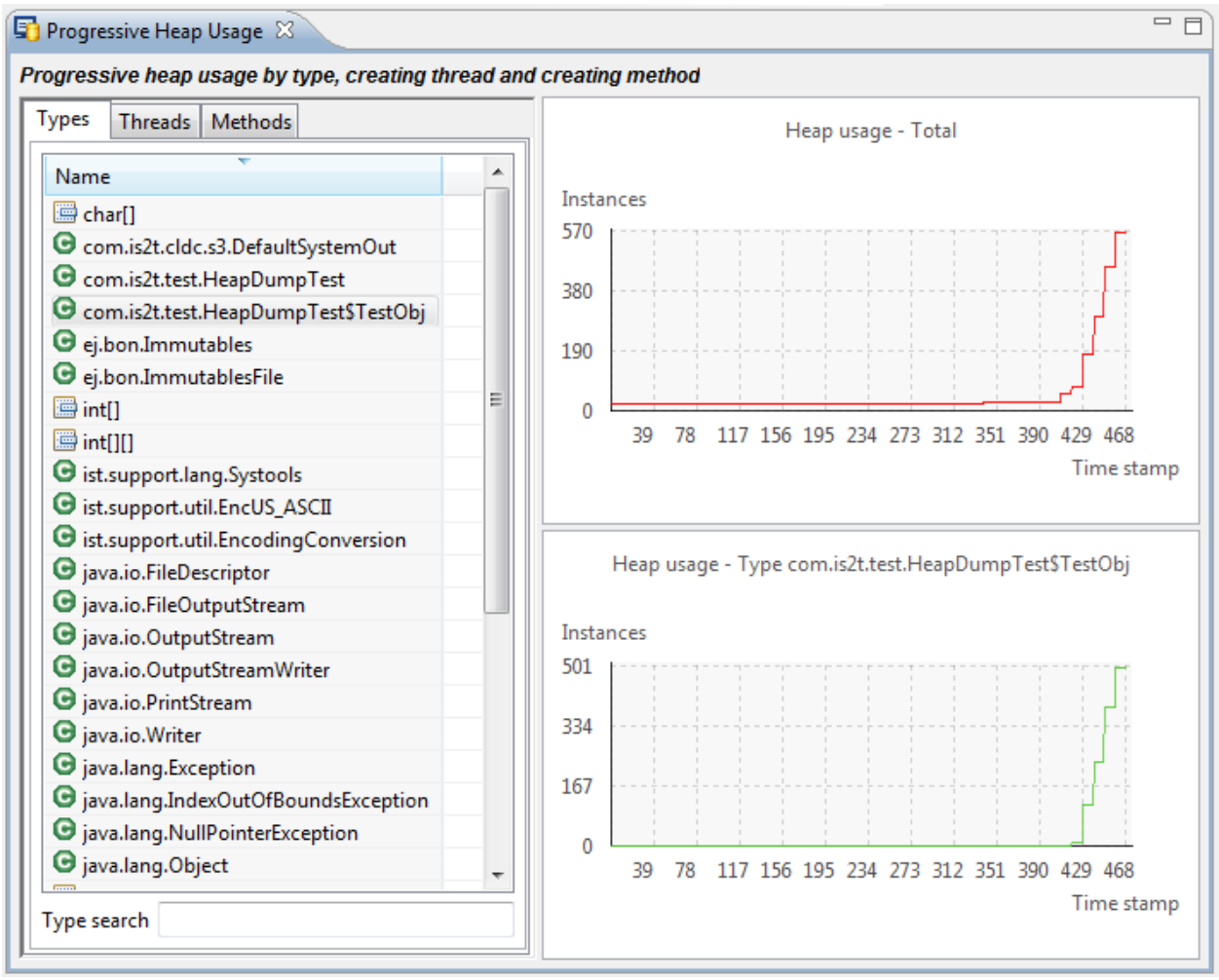


Fig. 69: Progressive Heap Usage

Compare Heap Dumps

The Compare tool compares the contents of two heap dump files. To open the tool select two heap dump XML files in the Package Explorer, right-click and select Heap Analyzer > Compare

The Compare tool shows the types in the old heap on the left-hand side, and the types in the new heap on the right-hand side, and marks the differences between them using different colors.

Types in the old heap dump are colored red if there are one or more instances of this type which are in the old dump but not in the new dump. The missing instances have been garbage collected.

Types in the new heap dump are colored green if there are one or more instances of this type which are in the new dump but not in the old dump. These instances were created after the old heap dump was written.

Clicking to the right of the type name unfolds the list to show the instances of the selected type.

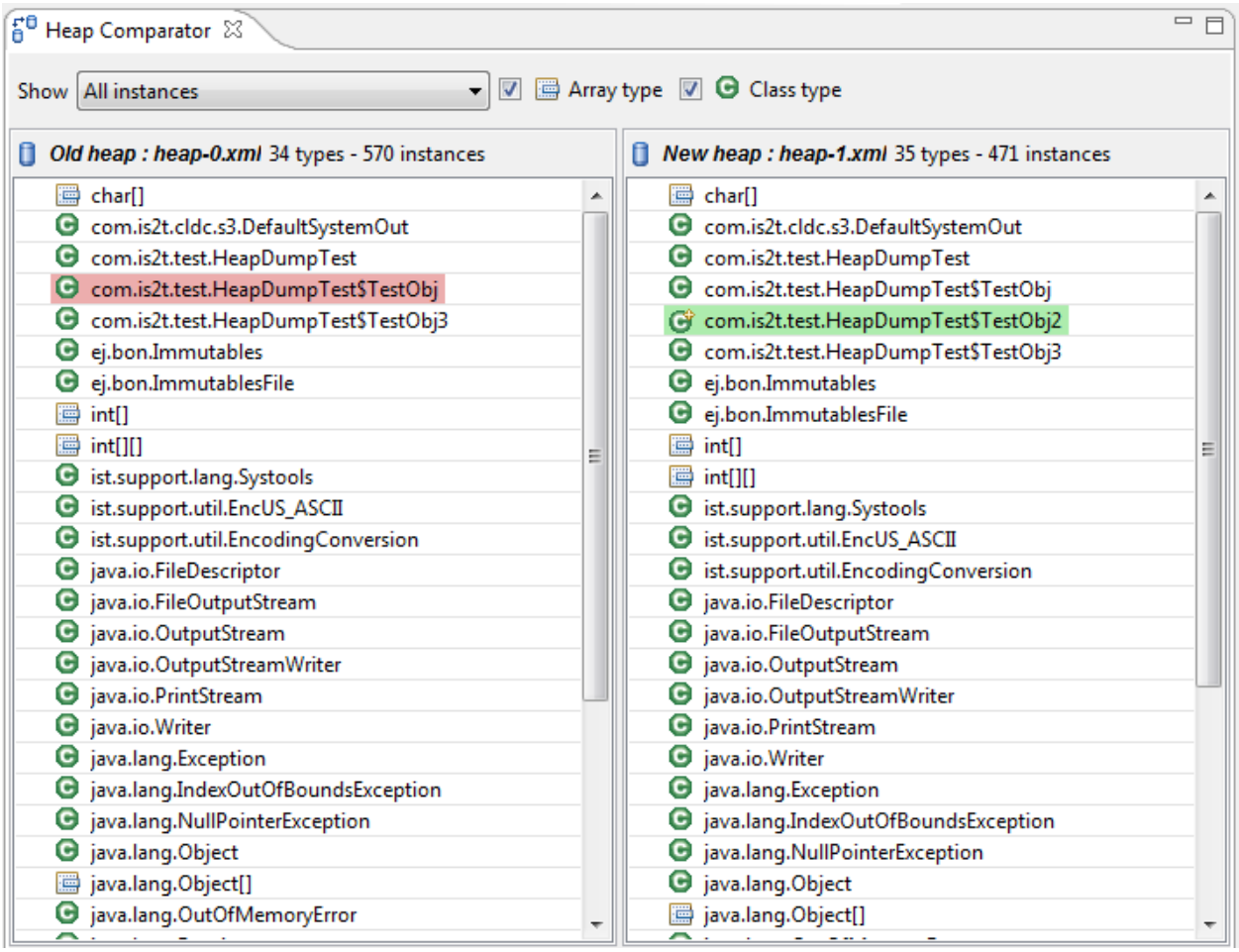


Fig. 70: Compare Heap Dumps

The combo box at the top of the tool allows the list to be restricted in various ways:

- All instances – no restriction.
- Garbage collected and new instances – show only the instances that exist in the old heap dump but not in the new dump, or which exist in the new heap dump but not in the old dump.

- Persistent instances – show only those instances that exist in both the old and new dumps.
- Persistent instances with value changed – show only those instances that exist in both the old and new dumps and have one or more differences in the values of their fields.

Instance Fields Comparison View

The Compare tool works in conjunction with the Instance Fields Comparison view, which opens automatically when an instance is selected in the tool.

The view shows the values of the fields of the instance in both the old and new heap dumps, and highlights any differences between the values.

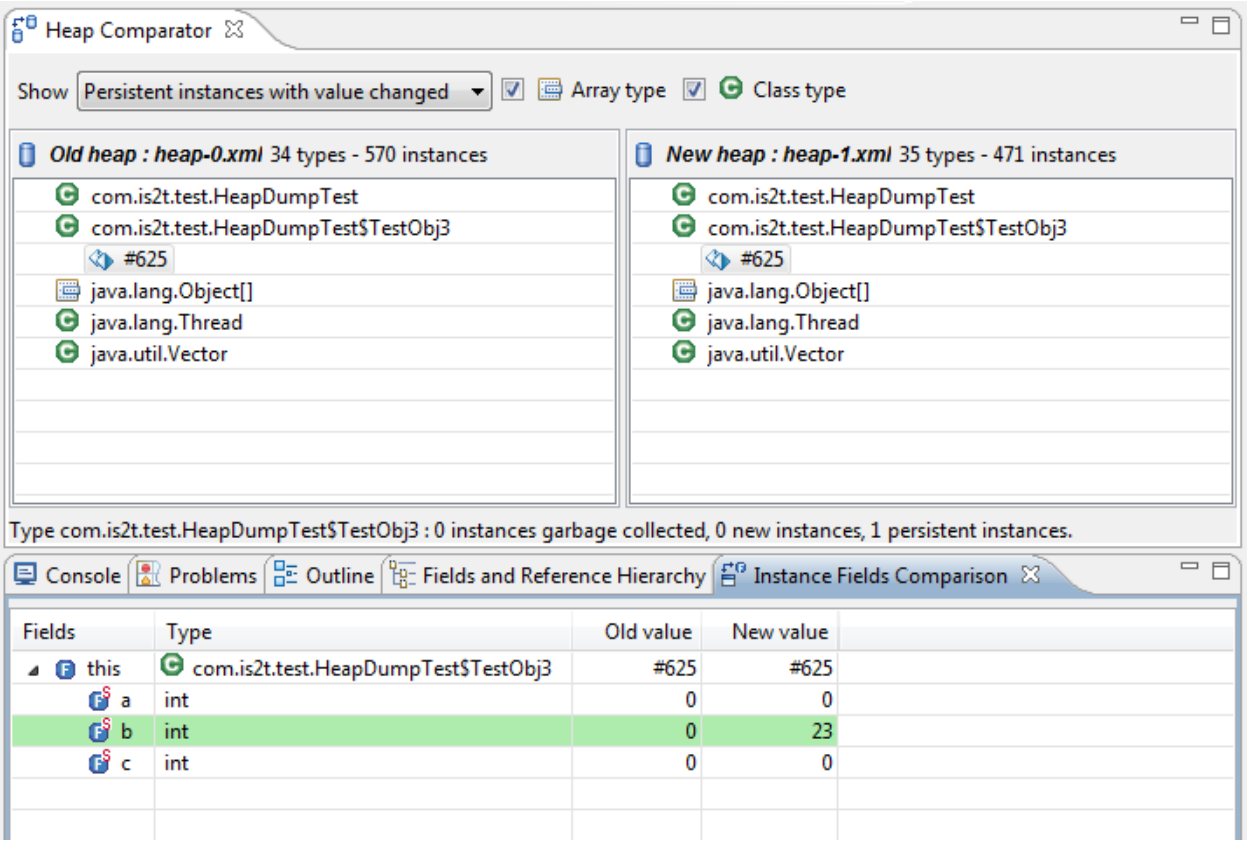


Fig. 71: Instance Fields Comparison view

Font Designer

Principle

The Font Designer module is a graphical tool (Eclipse plugin) that runs within the MicroEJ IDE used to build and edit .ejf fonts. It stores fonts in a VEE Port independent format.

Functional Description

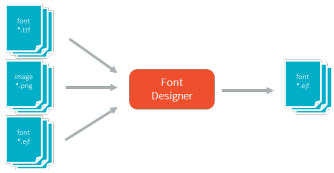


Fig. 72: Font Generation

Create an EJF Font

To create an EJF font, follow the steps below:

1. Open the Eclipse wizard: File > New > Other... > MicroEJ > MicroEJ Font .
2. Select a directory and a name.
3. Click Finish.

Once the font is created, a new editor is opened: the MicroEJ Font Designer.

Edit an EJF Font

You can edit your font with the MicroEJ Font Designer (by double-clicking on a *.ejf file or after running the new MicroEJ Font wizard).

This editor is divided into three main parts:

- The top left part manages the main font properties.
- The top right part manages the character to embed in your font.
- The bottom part allows you to edit a set of characters or an individual character.

Main Properties

The main font properties are:

- font size: height and width (in pixels).
- baseline (in pixels).
- space character size (in pixels).
- styles and filters.
- identifiers.

Refer to the following sections for more information about these properties.

Font Height

A font has a fixed height. This height includes the white pixels at the top and at the bottom of each character simulating line spacing in paragraphs.

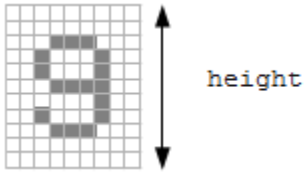


Fig. 73: Font Height

Font Width: Proportional and Monospace Fonts

A monospace font is a font in which all characters have the same width. For example a ! representation will be the same width as a w (they will be in the same size rectangle of pixels). In a proportional font, a w will be wider than a !.

A monospace font usually offers a smaller memory footprint than a proportional font because the Font Designer does not need to store the size of each character. As a result, this option can be useful if the difference between the size of the smallest character and the biggest one is small.

Baseline

Characters have a baseline: an imaginary line on top of which the characters seem to stand. Note that characters can be partly under the line, for example, g or }.

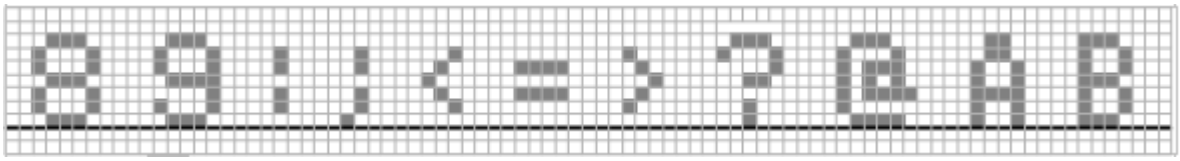


Fig. 74: The Baseline

Space Character

The Space character (0x20) is a specific character because it has no filled pixels. From the Main Properties Menu you can fix the space character size in pixels.

Note: When the font is monospace, the space size is equal to the font width.

Styles

Font Designer allows creating a font file that holds several combinations of built-in styles (styles hardcoded in pixels map) and runtime styles (styles rendered dynamically at runtime). However, since MicroUI 3, a MicroUI font holds only one style: PLAIN , BOLD , ITALIC or BOLD + ITALIC .

Font Designer features three drop-downs, one for each of BOLD , ITALIC , and UNDERLINED . Each drop-down has three options: None , Built-in and Dynamic . The font options must be adjusted to be compatible with MicroUI 3:

- The style option Dynamic (that targets the runtime style) is forbidden; select None instead.
- The syle UNDERLINED is forbidden; select None instead.

The styles options Built-in tag the font as bold, italic, or bold and italic. Adjust the styles options according to the font:

- The font is a *plain* font: select None option for each style.
- The font is a *bold* font: select Built-in for the style *bold* and None for the other styles.
- The font is an *italic* font: select Built-in for the style *italic* and None for the other styles.
- The font is a *bold* and *italic* font: select Built-in for the styles *bold* and *italic* and None for UNDERLINED .

Warning: When a font holds a dynamic style or when the style UNDERLINED is not None , an error at MicroEJ application compile-time is thrown (incompatible font file).

Identifiers

A number of identifiers can be attached to an EJF font. At least one identifier is required to specify the font. Identifiers are a mechanism for specifying the contents of the font – the set or sets of characters it contains. The identifier may be a standard identifier (for example, LATIN) or a user-defined identifier. Identifiers are numbers, but standard identifiers, which are in the range 0 to 80, are typically associated with a handy name. A user-defined identifier is an identifier with a value of 81 or higher.

Character List

The list of characters can be populated through the import button, which allows you to import characters from system fonts, images or another EJF font.

Import from System Font

This page allows you to select the system font to use (left part) and the range of characters. There are predefined ranges of characters below the font selection, as well as a custom selection picker (for example 0x21 to 0xfe for Latin characters).

The right part displays the selected characters with the selected font. If the background color of a displayed character is red, it means that the character is too large for the defined height, or in the case of a monospace font, it means the character is too high or too wide. You can then adjust the font properties (font size and style) to ensure that characters will not be truncated.

When your selection is done, click the Finish button to import this selection into your font.

Import from Images

This page allows the loading of images from a directory. The images must be named as follows: 0x[UTF-8].[extension] .

When your selection is done, click the Finish button to import the images into your font.

Character Editor

When a single character is selected in the list, the character editor is opened.

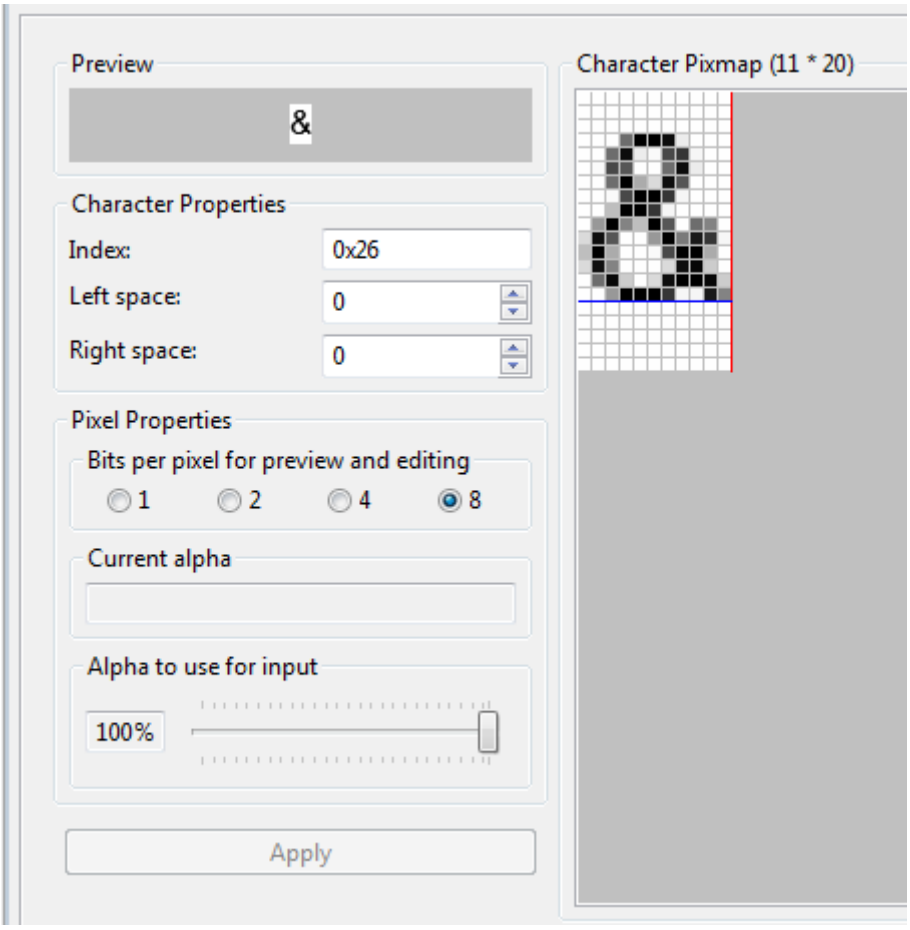


Fig. 75: Character Editor

Properties

You can define specific properties, such as left and right space, or index. You can also draw the character pixel by pixel - a left-click in the grid draws the pixel, a right-click erases it.

The changes are not saved until you click the Apply button. When changes are applied to a character, the editor shows that the font has changed, so you can now save it.

The same part of the editor is also used to edit a set of characters selected in the top right list. You can then edit the common editable properties (left and right space) for all those characters at the same time.

Anti-Aliased Fonts

By default, when characters are imported from a system font, each pixel is either fully opaque or fully transparent. Fully opaque pixels show as black squares in the character grid in the right-hand part of the character editor; fully transparent pixels show as white squares.

However, the pixels stored in an ejf file can take one of 256 grayscale values. A fully-transparent pixel has the value 255 (the RGB value for white), and a fully-opaque pixel has the value 0 (the RGB value for black). These grayscale values are shown in parentheses at the end of the text in the Current alpha field when the mouse cursor hovers over a pixel in the grid. That field also shows the transparency level of the pixel, as a percentage, where 100% means fully opaque.

It is possible to achieve better-looking characters by using a combination of fully-opaque and partially-transparent pixels. This technique is called *anti-aliasing*. Anti-aliased characters can be imported from system fonts by checking the anti aliasing box in the import dialog. The & character shown in the screenshot above was imported using anti aliasing, and you can see the various gray levels of the pixels.

When the Font Generator converts an ejf file into the raw format used at runtime, it can create fonts with characters that have 1, 2, 4 or 8 bits-per-pixel (bpp). If the raw font has 8 bpp, then no conversion is necessary and the characters will render with the same quality as seen in the character editor. However, if the raw font has less than 8 bpp (the default is 1 bpp) any gray pixels in the input file are compressed to fit, and the final rendering will be of lower quality (but less memory will be required to hold the font).

It is useful to be able to see the effects of this compression, so the character editor provides radio buttons that allow the user to preview the character at 1, 2, 4, or 8 bpp. Furthermore, when 2, 4 or 8 bpp is selected, a slider allows the user to select the transparency level of the pixels drawn when the left mouse button is clicked in the grid.

Character Pixmap

This view allows to previewing and editing the character's pixels. The closer the pixel's color is to black, the more opaque the encoded opacity will be.

Left or right click on a pixel to draw or erase it.

Preview

You can preview your font by pressing the Preview... button, which opens the Preview wizard. In the Preview wizard, press the Select File button, and select a text file which contains text that you want to see rendered using your font. Characters that are in the selected text file but not available in the font will be shown as red rectangles.

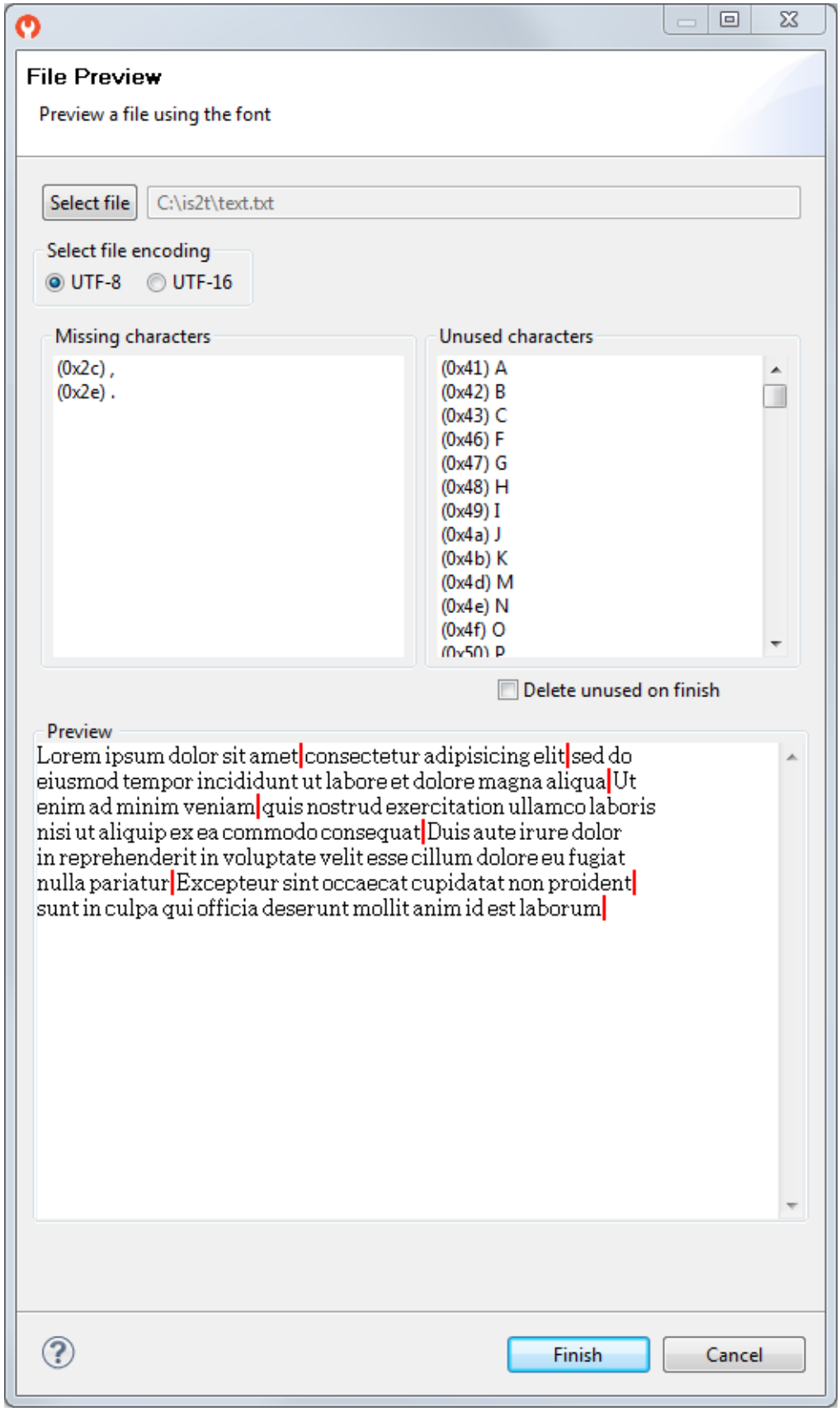


Fig. 76: Font Preview

Removing Unused Characters

In order to reduce the size of a font file, you can reduce the number of characters in your font to be only those characters used by your application. To do this, create a file which contains all the characters used by your application (for example, concatenating all your NLS files is a good starting point). Then open the Preview wizard as described above, selecting that file. If you select the check box Delete unused on finish, then those characters that are in the font but not in the text file will be deleted from the font when you press the Finish button, leaving your font containing the minimum number of characters. As this font will contain only characters used by a specific application, it is best to prepare a "complete" font, and then apply this technique to a copy of that font to produce an application specific cut-down version of the font.

Use an EJF Font

An EJF Font must be converted to a format which is specific to the targeted VEE Port. The Font Generator tool performs this operation for all fonts specified in the list of fonts configured in the application launch.

Dependencies

No dependency.

Installation

The Font Designer module is already installed in the MicroEJ SDK5.

Use

Create a new ejf font file or open an existing one in order to open the Font Designer plugin.

Local Deployment Socket

Principle

The Local Deployment Socket is a tool that allows to transfer an Application on the device over a network connection.

Functional Description

The Local Deployment Socket builds the Application Feature file (.fo) and upload it on the device identified by its IP address. On the device, it is the job of the Kernel Application that receives the Feature file to install and to run the Application.

Use

```
./gradlew execTool --name=localDeploymentSocket \
  --toolProperty=application.main.class="com.mycompany.MyFeature" \
  --toolProperty=kernel.filename="[path_to_application.out]" \
  --toolProperty=board.server.host="10.0.0.171" \
  --toolProperty=board.server.port="4000" \
  --toolProperty=board.timeout="120000" \
  --toolProperty=use.storage="true"
```

Note: It is also possible to create a custom task of type `ExecToolTask` dedicated to the Local Deployment Socket. Refer to the *How To Create Custom ExecTool Task* chapter for more information.

Options

Option: Application Feature Class

Option Name: `application.main.class`

Required?: Yes

Description:

Specify the entry-point as the full qualified name of the Application to deploy.

Option: Kernel Executable

Option Name: `kernel.filename`

Required?: Yes

Description:

Specify the Kernel Executable to use for building the Application Feature file.

Option: Server Host

Option Name: `board.server.host`

Required?: Yes

Description:

The IP of the target device.

Option: Server Port

Option Name: board.server.port

Required?: Yes

Description:

The TCP port on which the Kernel listens (usually 4000).

Option: Timeout

Option Name: board.timeout

Required?: Yes

Description:

If there is no activity within the defined timeout period (in seconds), the tool will disconnect from the device.

Option: Use Storage

Option Name: use.storage

Required?: Yes

Description:

A boolean describing whether to use the storage to store the Application or not. Refer to the Kernel documentation to find out the correct setting.

Serial to Socket Transmitter

Principle

The MicroEJ Serial to Socket Transmitter is a tool which transfers all bytes from a serial port to a TCP client or TCP server.

It is typically used to transfer the output traces of a testsuite executed on a device.

Use

To start transfering data from the COM8 serial port to the port 5555, execute the execTool task as followed:

```
./gradlew execTool --name=serialToSocketTransmitter \
  --toolProperty=serail.to.socket.comm.port="COM8" \
  --toolProperty=serail.to.socket.comm.baudrate="115200" \
  --toolProperty=serail.to.socket.server.port="5555" \
  --console plain
```

Run Serial to Socket Transmitter with Custom Task

To simplify the use of the Serial to Socket Transmitter, it is also possible to create a *custom task* in the `build.gradle.kts` file of your project as follows:

- Import the `ExecToolTask` and `LoadVeeTask` classes:

```
import com.microej.gradle.tasks.ExecToolTask
import com.microej.gradle.tasks.LoadVeeTask
```

- Create a new task of type `ExecToolTask` :

```
val loadVee = tasks.withType(LoadVeeTask::class).named("loadVee")
val mainSourceSet = project.extensions.getByType(SourceSetContainer::class).getByName(SourceSet.
↪MAIN_SOURCE_SET_NAME)

tasks.register<ExecToolTask>("serialToSocketTransmitter") {
    group = "microej"
    veeDir.set(loadVee.flatMap { it.loadedVeeDir })
    resourcesDirectories.from(mainSourceSet.output.resourcesDir, layout.buildDirectory.dir("generated/
↪microej-app-wrapper/resources"))
    classesDirectories.from(mainSourceSet.output.classesDirs, layout.buildDirectory.dir("generated/microej-
↪app-wrapper/classes"))
    classpathFromConfiguration.from(project.configurations.getByName("runtimeClasspath"))

    toolName = "serialToSocketTransmitter"
    toolProperties.putAll(mapOf(
        "serail.to.socket.comm.port" to "COM8",
        "serail.to.socket.comm.baudrate" to "115200",
        "serail.to.socket.server.port" to "5555"
    ))
}
```

- The custom task can then be executed:

```
$ ./gradlew serialToSocketTransmitter
```

Options

Serial Port

Option Name: `serail.to.socket.comm.port`

Default value: `COM0`

Description: Defines the COM port:

Windows - `COM1` , `COM2` , ... , `COM*n*`

Linux - `/dev/ttyS0` , `/dev/ttyUSB0` , ... , `/dev/ttyS*n*` , `/dev/ttyUSB*n*`

Baudrate

Option Name: serail.to.socket.comm.baudrate

Default value: 115200

Available values:

9600

38400

57600

115200

Description: Defines the COM baudrate.

Socket Port

Option Name: serail.to.socket.server.port

Default value: 5555

Description: Defines the TCP server port.

Kernel API Generator

Principle

The Kernel API Generator is a tool that helps to generate a *kernel.api file* based on a Java classpath.

Use

To generate the kernel.api file, make sure to compile all the project Java sources (by running the build task for example), then execute the execTool task as followed:

```
./gradlew execTool --name=kernelAPIGenerator \
  --toolProperty=kernel.api.generator.classpath="C:/my-kernel/build/classes/java/main"  \
  --toolProperty=output.dir="C:/my-kernel/build" \
  --toolProperty=kernel.api.generator.includes.patterns="**/*.class" \
  --toolProperty=kernel.api.generator.excludes.patterns="" \
  --console plain
```

The kernel.api file is generated in the folder defined by the output.dir property.

> **Warning:** The folder specified by the output.dir must exist, it is not automatically created. If the folder does not exist, the kernel.api file is not generated and the execution silently ends (no warning or error message).

Options

Java classpath

Option Name: kernel.api.generator.classpath

Default value: No default value, this option is required.

Description: List of the absolute paths of the JAR files or folders containing Java class files, for example C:/folder1/lib.jar;C:/folder2 .

Destination directory path

Option Name: output.dir

Default value: No default value, this option is required.

Description: Absolute path of the destination directory of the generated kernel.api files. It is important to note that this folder must exist.

Pattern of the classes to include

Option Name: kernel.api.generator.includes.patterns

Default value: **/*.class (which means all the classes).

Description: Ant-style pattern of the classes to include.

Pattern of the classes to exclude

Option Name: kernel.api.generator.excludes.patterns

Default value: Empty (which means no excluded classes).

Description: Ant-style pattern of the classes to exclude.

Null Analysis

NullPointerException thrown at runtime is one of the most common causes for failure of Java programs. All modern IDEs provide a Null Analysis tool which can detect such programming errors (misuse of potential null Java values) at compile-time.

Principle

The Null Analysis tool is based on Java annotations. Each Java field, method parameter and method return value must be marked to indicate whether it can be null or not.

Once the Java code is annotated, the IDE must be configured to enable Null Analysis detection.

Java Code Annotation

MicroEJ defines its own annotations:

- @NonNullByDefault: Indicates that all fields, method return values or parameters can never be null in the annotated package or type. This rule can be overridden on each element by using the @Nullable annotation.
- @Nullable: Indicates that a field, local variable, method return value or parameter can be null.
- @NonNull: Indicates that a field, local variable, method return value or parameter can never be null.

MicroEJ recommends to annotate the Java code as follows:

- In each Java package, create a package-info.java file and annotate the Java package with @NonNullByDefault. This is a common good practice to deal with non null elements by default to avoid undesired NullPointerException. It enforces the behavior which is already widely outlined in Java coding rules.

```
@ej.annotation.NonNullByDefault
package com.mycompany;
```

- In each Java type, annotate all fields, methods return values and parameters that can be null with @Nullable. Usually, this information is already available as textual information in the field or method Javadoc comment. The following example of code shows where annotations must be placed:

```
@Nullable
public Object thisFieldCanBeNull;

@Nullable
public Object thisMethodCanReturnNull() {
  return null;
}

public void thisMethodParameterCanBeNull(@Nullable Object param) {

}
```

IDE Configuration

Requirements

The project must depend at least on the version 1.3.6 of the ej.api:edc module:

```
dependencies {
  implementation("ej.api:edc:1.3.6")
}
```

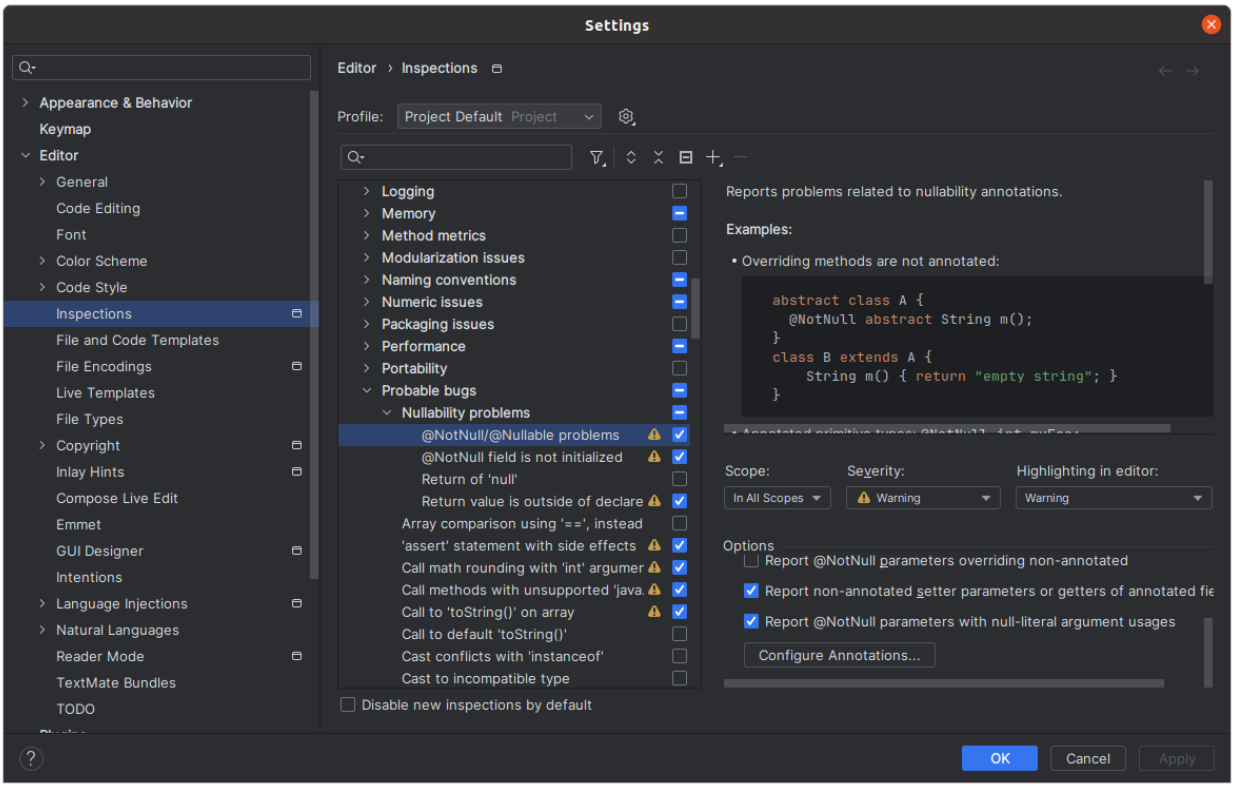
Project configuration

IntelliJ IDEA / Android Studio

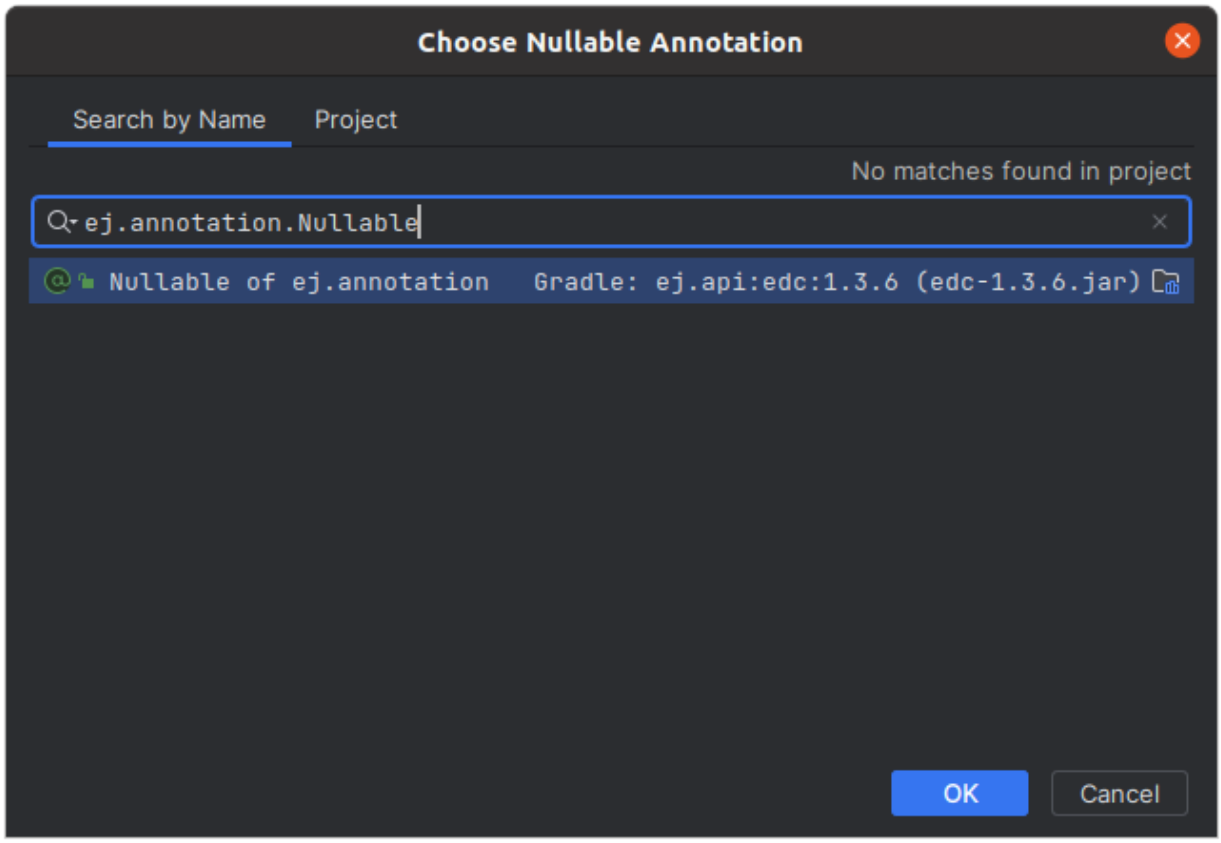
Eclipse

Follow these steps to enable the Null Analysis tool in IntelliJ IDEA and Android Studio:

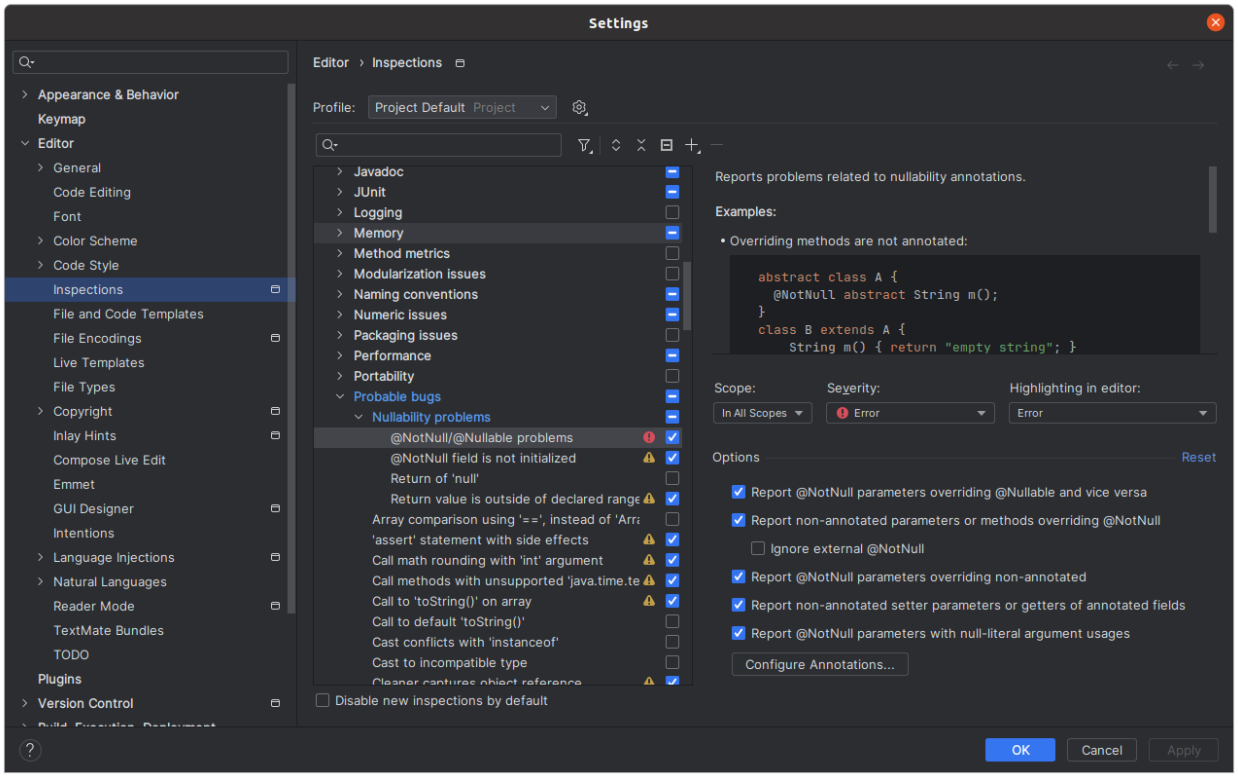
- Go to `File` > `Settings…` .
- Go to `Editor` > `Inspections` .
- Open the category `Java` > `Probable Bugs` > `Nullability problems` > `@NotNull/@Nullable problems` .
- In the `Options` panel, scroll down and click on the `Configure Annotations…` button.



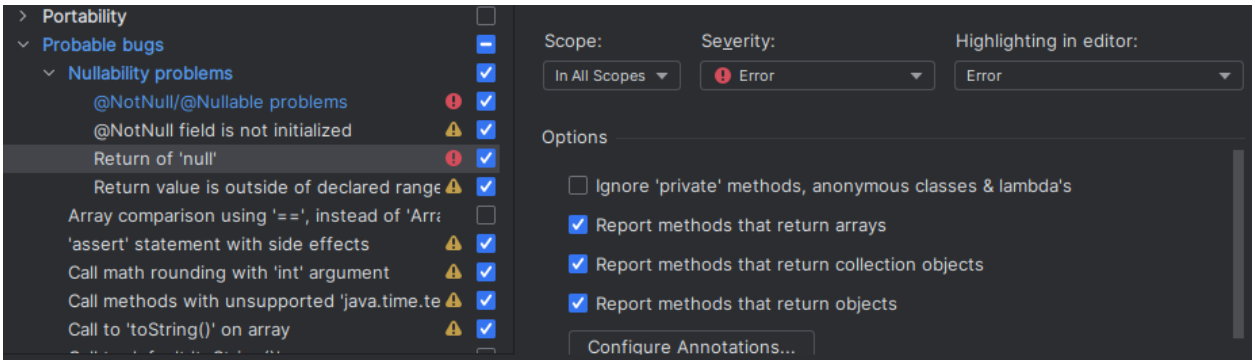
- In the `Nullable` tab, click on the `+` button.
- Type `ej.annotation.Nullable` and select the listed class.



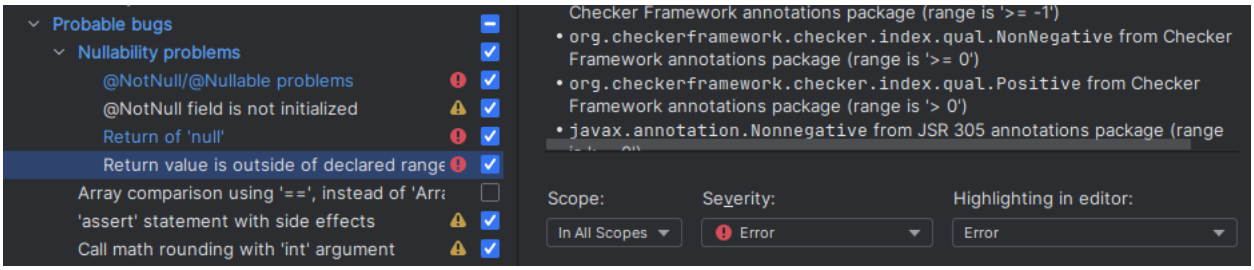
- Select ej.annotation.Nullable in the Nullable annotations list.
- Go to the NotNull tab and repeat the same steps with the ej.annotation.NonNull class.
- Click on the OK button.
- Change the Severity field to Error .
- Check the Report @NotNull parameters overriding non-annotated option.



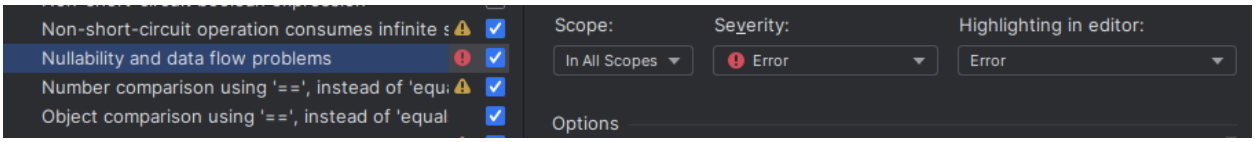
- Check and select the category Java > Probable Bugs > Nullability problems > Return of 'null' .
- Change the Severity field to Error .



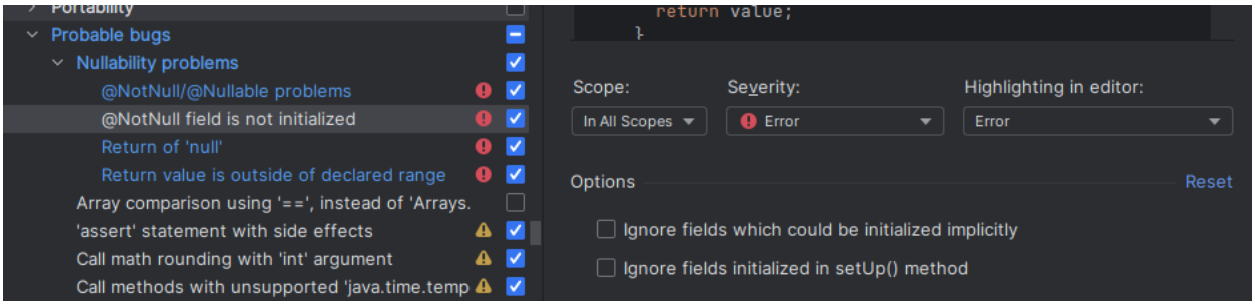
- Select the category Java > Probable Bugs > Nullability problems > Return value is outside of declared range .
- Change the Severity field to Error .



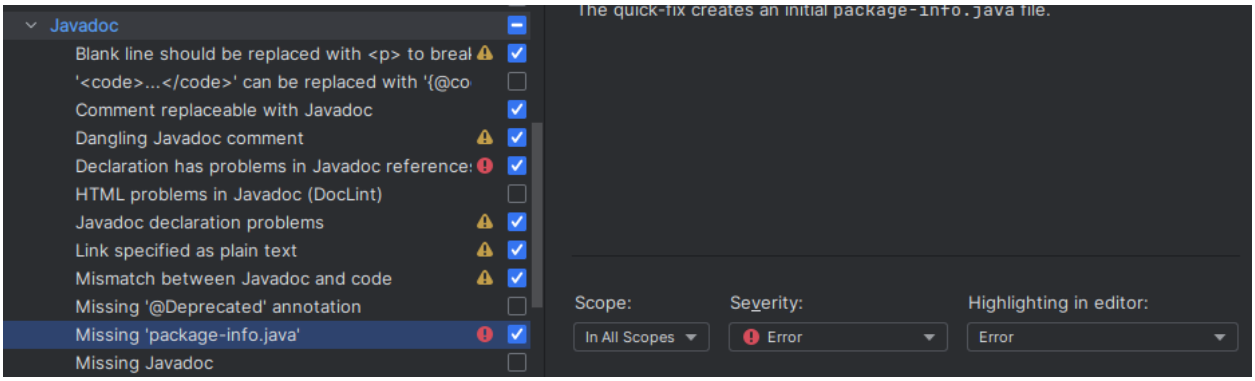
- Select the category `Java` > `Probable Bugs` > `Nullability and data flow problems` .
- Change the `Severity` field to `Error` .



- Check and select the category `Java` > `Probable Bugs` > `Nullability problems` > `@NotNull field is not initialized` .
- Change the `Severity` field to `Error` .
- Uncheck the `Ignore fields which could be initialized implicitly` option.
- Uncheck the `Ignore fields initialized in setUp() method` option.



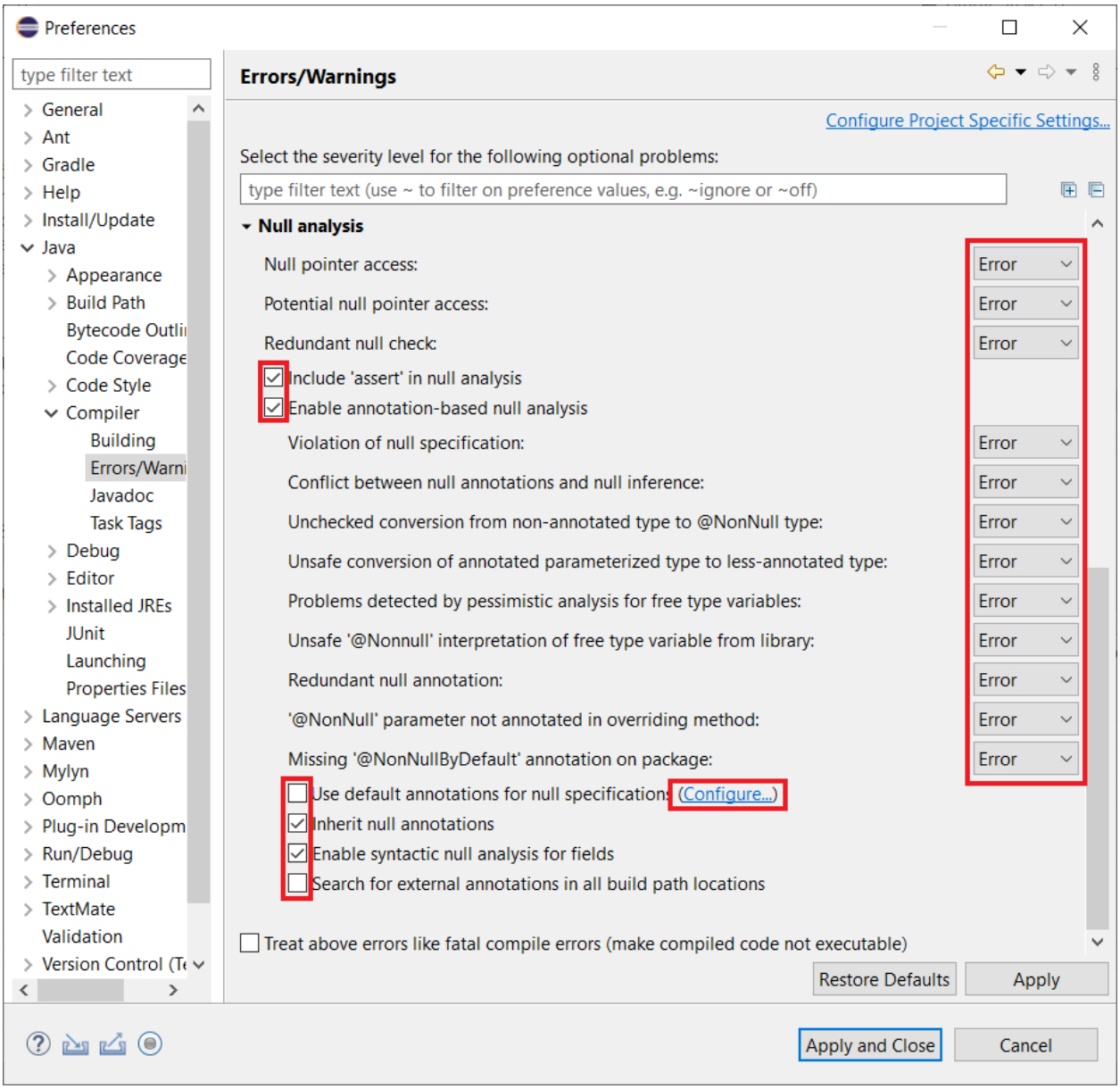
- Check and select the category `Java` > `Javadoc` > `Missing'package-info.java'` .
- Change the `Severity` field to `Error` .



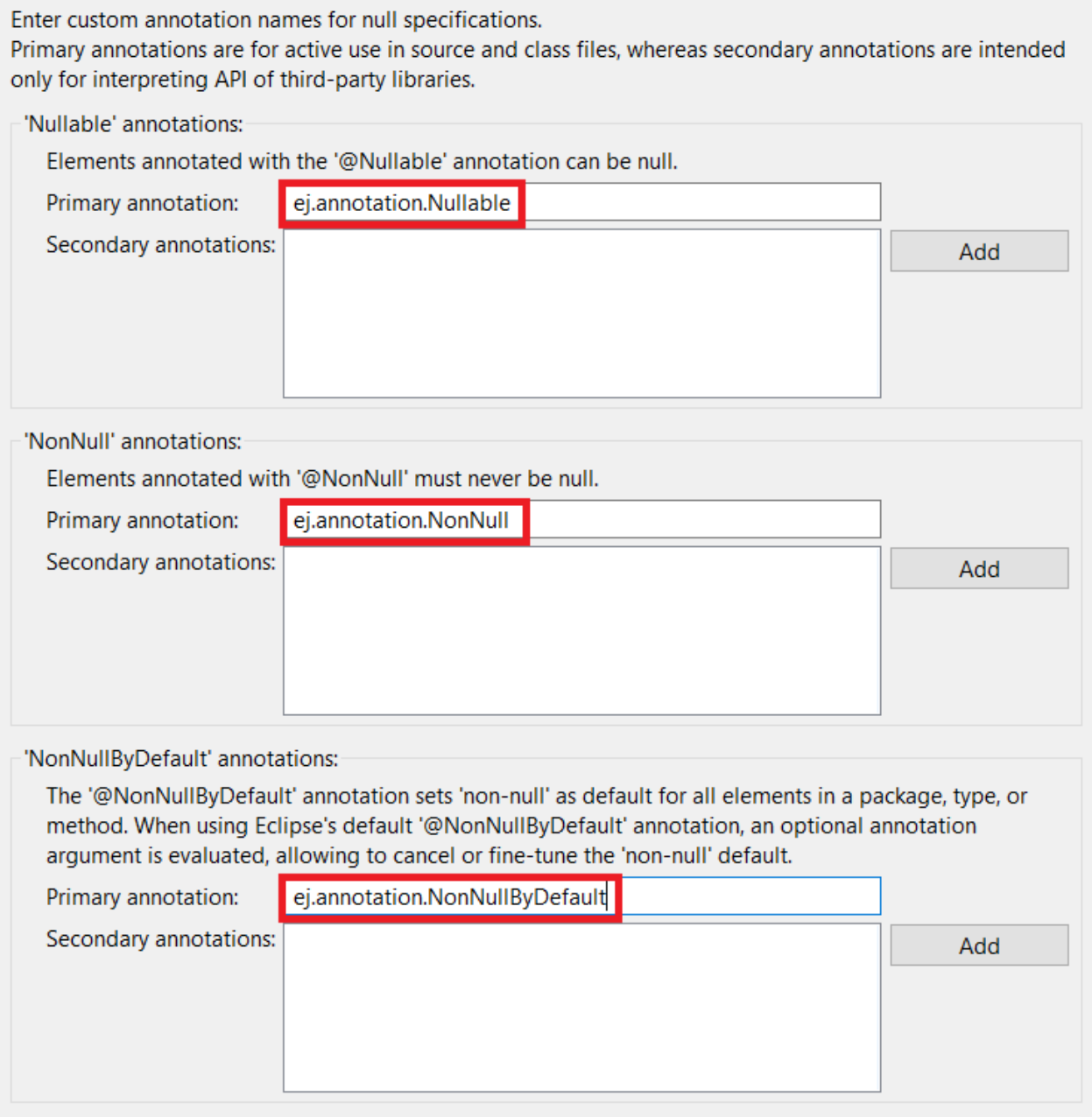
For more details, refer to the official documentation on Configure nullability annotations.

To enable the Null Analysis tool in Eclipse, a project must be configured as follows:

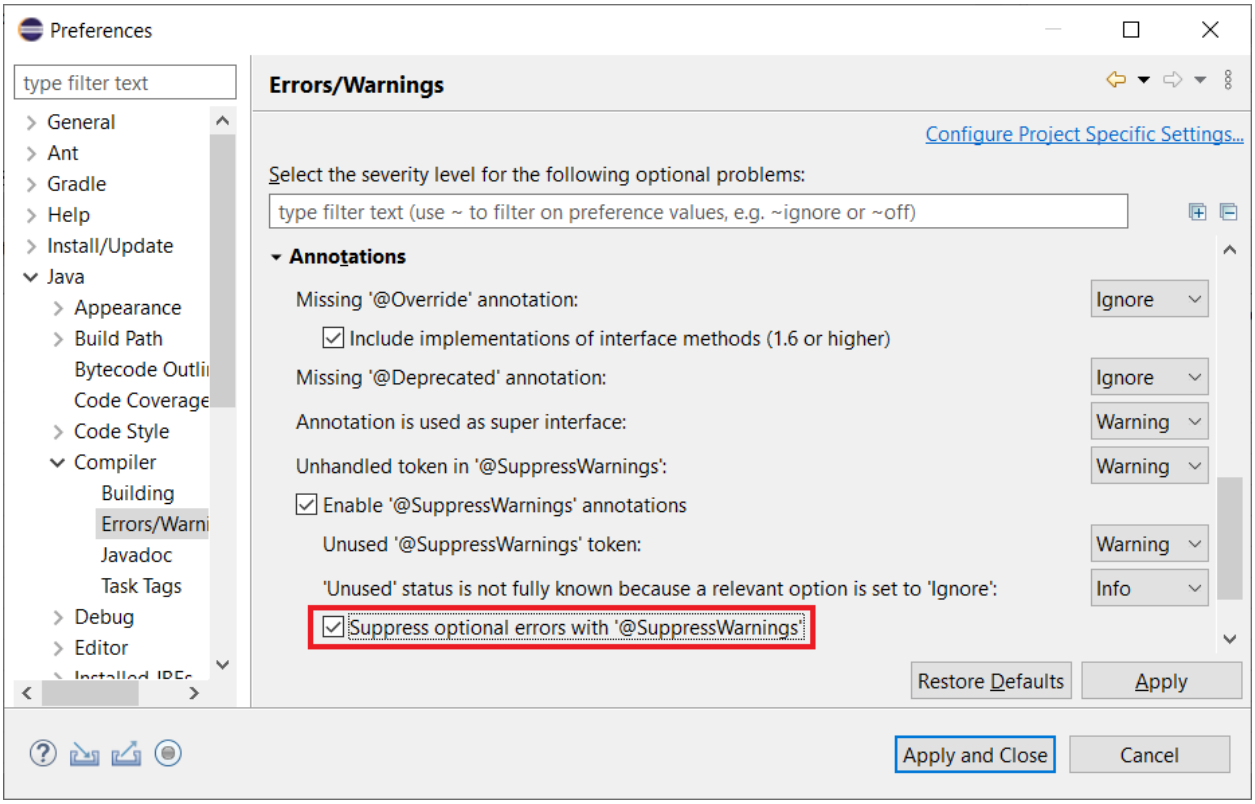
- In the Package Explorer, right-click on the module project and select `Properties` ,
- Navigate to `Java Compiler` > `Errors/Warnings` ,
- In the `Null analysis` section, configure options as follows:



- Click on the Configure… link to configure MicroEJ annotations:
  - ej.annotation.Nullable
  - ej.annotation.NonNull
  - ej.annotation.NonNullByDefault



- In the `Annotations` section, check `Suppress optional errors with '@SuppressWarnings'` option:



This option allows to fully ignore Null Analysis errors in advanced cases using `@SuppressWarnings("null")` annotation.

If you have multiple projects to configure, you can then copy the content of the `.settings` folder to an other module project.

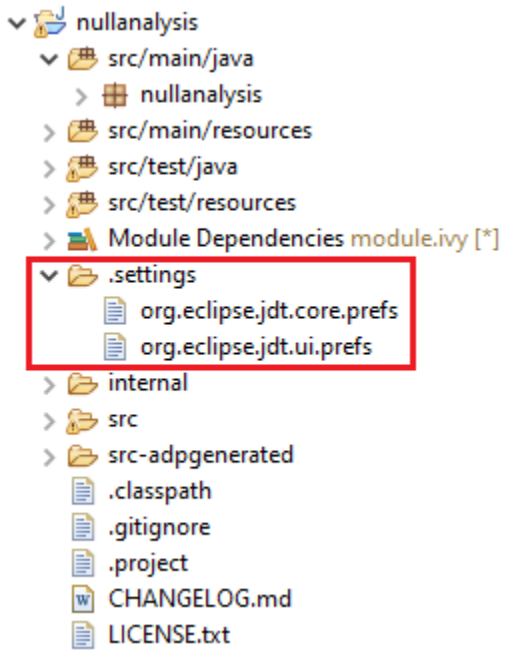


Fig. 77: Null Analysis Settings Folder

Warning: You may lose information if your target module project already has custom parameterization or if it was created with another SDK version. In case of any doubt, please configure the options manually or merge with a text file comparator.

Launching Null Analysis

While Eclipse automatically launches Null Analysis on the whole project and reports all the problems found, IntelliJ IDEA and Android Studio launch the Null Analysis only on the currently open file. In order to launch an Analysis of the full project:

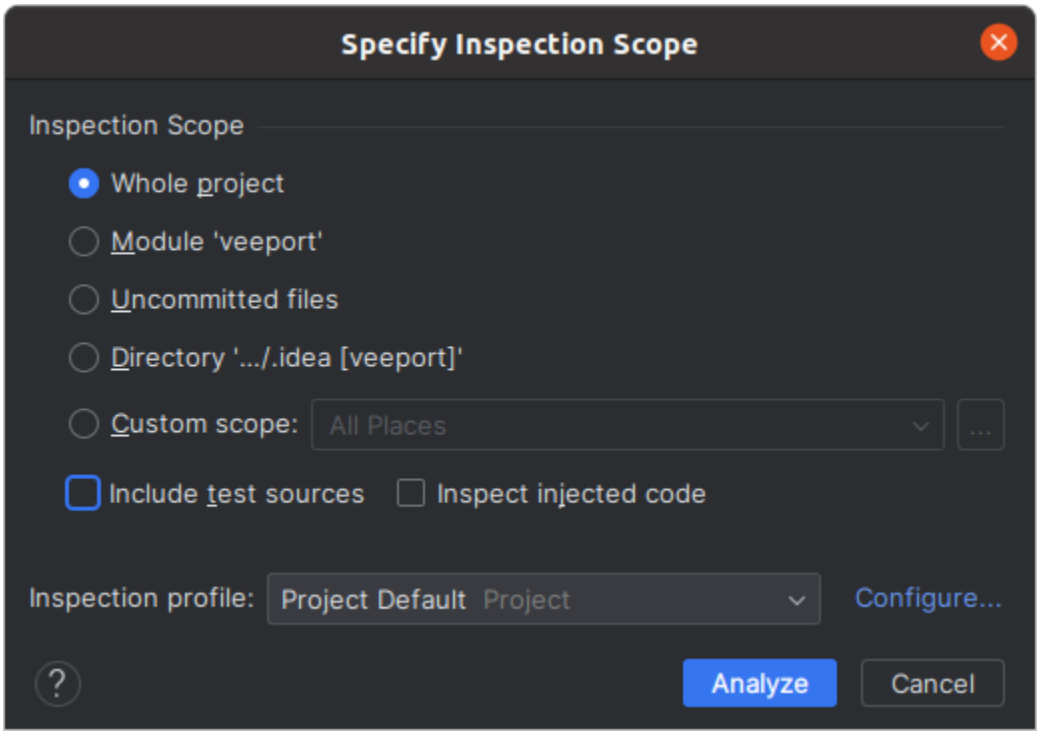
- Go to `Code` > `Inspect Code` .
- Check the `Whole project` option.
- Uncheck the `Include test sources` option.
- Click on `Analyze` .

Disabling Analysis for Test Folder

IntelliJ IDEA / Android Studio

Eclipse

The Analysis of the test folder can be disabled by unchecking the option `Include test sources` when launching a Code Inspection:



The Null Analysis can be automatically disabled in test folder by using the `eclipse-wtp` Gradle plugin:

- In the `build.gradle.kts` plugin of the project, add the `eclipse-wtp` plugin:

```
plugins {
  ...
  `eclipse-wtp`
}
```

- Then add the following snippet of code:

```
import org.gradle.plugins.ide.eclipse.model.AbstractClasspathEntry
import org.gradle.plugins.ide.eclipse.model.Classpath

eclipse.classpath.file {
    whenMerged(Action<Classpath> {
        entries.filter { entry ->
            (entry.kind == "src" && entry is AbstractClasspathEntry && entry.path == "src/test/java")
        }.forEach { entry ->
            (entry as AbstractClasspathEntry).entryAttributes["ignore_optional_problems"] = "true"
        }
```

(continues on next page)

(continued from previous page)

```
    })
}
```

Adapth the path of your test folder accordingly.

Sharing Null Analysis IDE Configuration

IntelliJ IDEA / Android Studio

Eclipse

The configuration related to Null Analysis is located in the `.idea/misc.xml` and `.idea/inspectionProfiles/*` files. In order to share them, they must be committed in your project source reposiory. For Git projects, if you decided to not commit the IDE configuration files, these files can be excluded with the following lines in the `.gitignore` file:

```
/.idea/*
!/.idea/misc.xml
!/.idea/inspectionProfiles/*
```

Warning: IntelliJ IDEA and Android Studio create a `.gitignore` file in the `.idea` folder. You can remove it or adapt it to fit your needs.

The configuration related to Null Analysis is located in the `.settings/org.eclipse.jdt.core.prefs` file. In order to share it, it must be committed in your project source reposiory. For Git projects, if you decided to not commit the IDE configuration files, these files can be excluded with the following lines in the `.gitignore` file:

```
/.settings/*
!/.settings/org.eclipse.jdt.core.prefs
```

MicroEJ Libraries

Many libraries available on *Central Repository* are annotated with Null Analysis. If you are using a library which is not yet annotated, please contact *our support team*.

For the benefit of Null Analysis, some APIs have been slightly constrained compared to the Javadoc description. Here are some examples to illustrate the philosophy:

- System.getProperty(String key, String def) does not accept a `null` default value, which allows to ensure the returned value is always non `null`.
- Collections of the Java Collections Framework that can hold `null` elements (e.g. HashMap) do not accept `null` elements. This allows APIs to return `null` (e.g. HashMap.get(Object)) only when an element is not contained in the collection.

Implementations are left unchanged and still comply with the Javadoc description whether the Null Analysis is enabled or not. So if these additional constraints are not acceptable for your project, please disable Null Analysis.

3.4.18 VEE Port

Create a VEE Port

This section describes the steps to create a new VEE Port with the SDK, and options to connect it to an external Board Support Package (BSP) as well as a third-party C toolchain.

Note: If you own a legacy VEE Port, you can either create your VEE Port again from scratch, or follow the *Former Platform Migration* chapter.

VEE Port Project Creation

The first step is to create a VEE Port configuration project:

IntelliJ IDEA

Android Studio

Eclipse

Visual Studio Code

Command Line Interface

- Click `File` > `New` > `Project from Version Control…` .
- Select `Repository URL` .
- Select `Git` as Version control.
- Fill the URL of the VEE Port Project Template Github Repository.
- Fill the Directory in which the Project Template must be cloned.
- Click on `Clone` button.

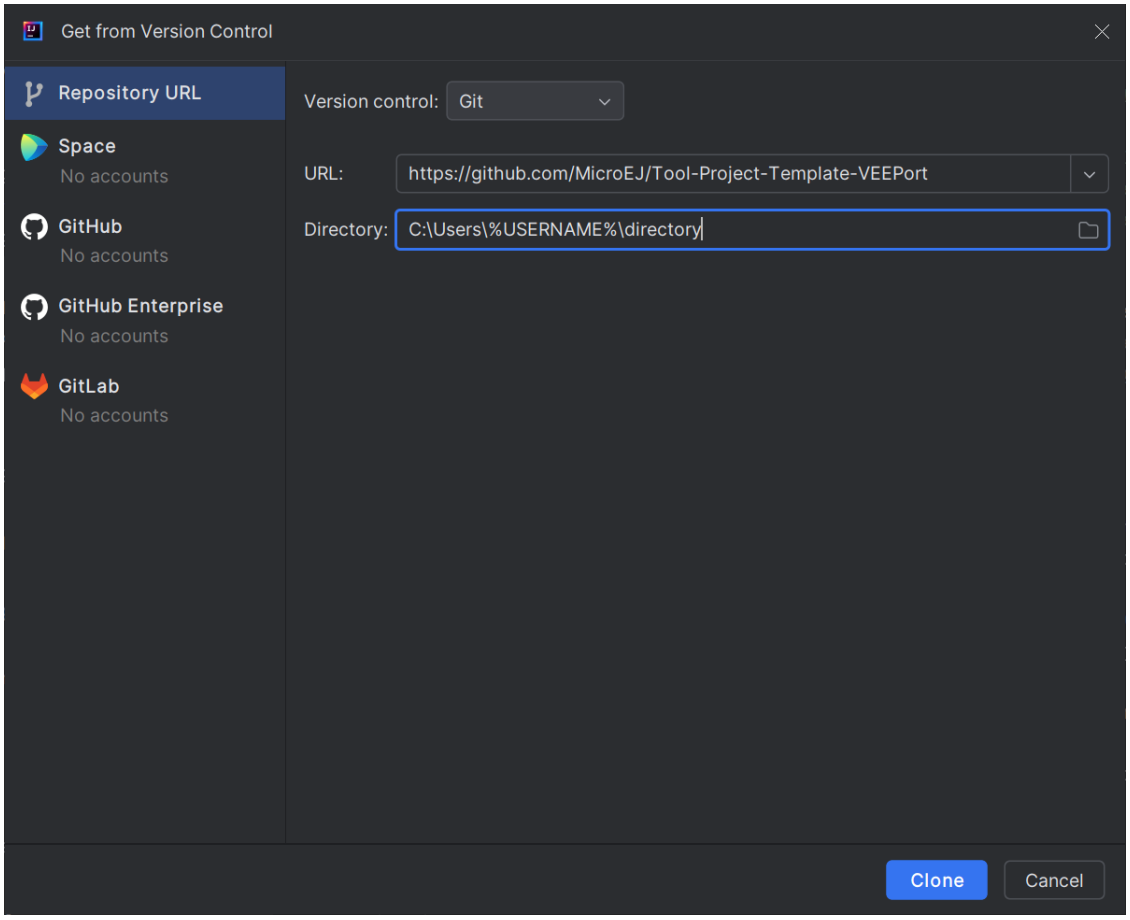


Fig. 78: Version Control Window in IntelliJ IDEA

- When the Gradle project is loaded, rename the project and change its group and version in the build.gradle.kts build script.
- Select View > Tool Windows > Terminal .
- In the integrated terminal, run the following command at the root of the project to remove the Git Repository:

  Windows

  Linux/macOS

```
rm -r -Force .git*
```

```
rm -rf .git*
```

- Click File > New > Project from Version Control... .
- Select Repository URL .
- Select Git as Version control.
- Fill the URL of the VEE Port Project Template Github Repository.
- Fill the Directory in which the Project Template must be cloned.
- Click on Clone button.

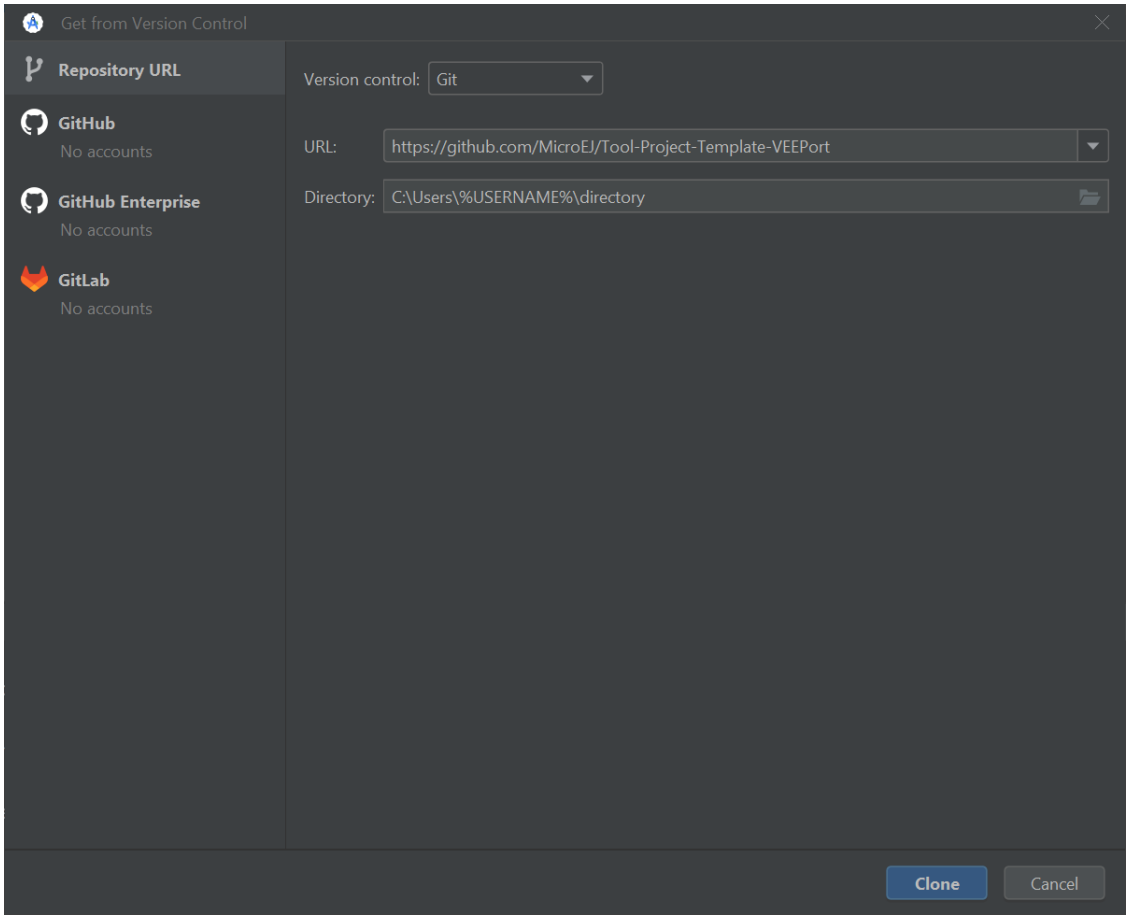


Fig. 79: Version Control Window in Android Studio

- When the Gradle project is loaded, rename the project and change its group and version in the build.gradle.kts build script.
- Select View > Tool Windows > Terminal .
- In the integrated terminal, run the following command at the root of the project to remove the Git Repository:

  Windows

  Linux/macOS

```
rm -r -Force .git*
```

```
rm -rf .git*
```

- Open a new terminal.
- Clone the VEE Port Project Template Github Repository:

```
git clone https://github.com/MicroEJ/Tool-Project-Template-VEEPort.git
```

- Remove the Git Repository from the project:

  Windows

  Linux/macOS

```
rm -r -Force .git*
```

```
rm -rf .git*
```

- Rename the project and change its group and version in the `build.gradle.kts` build script.
- In Eclipse, click on `File` > `Import…` .
- Select the project type `Gradle` > `Existing Gradle Project` and click on the `Next` button.

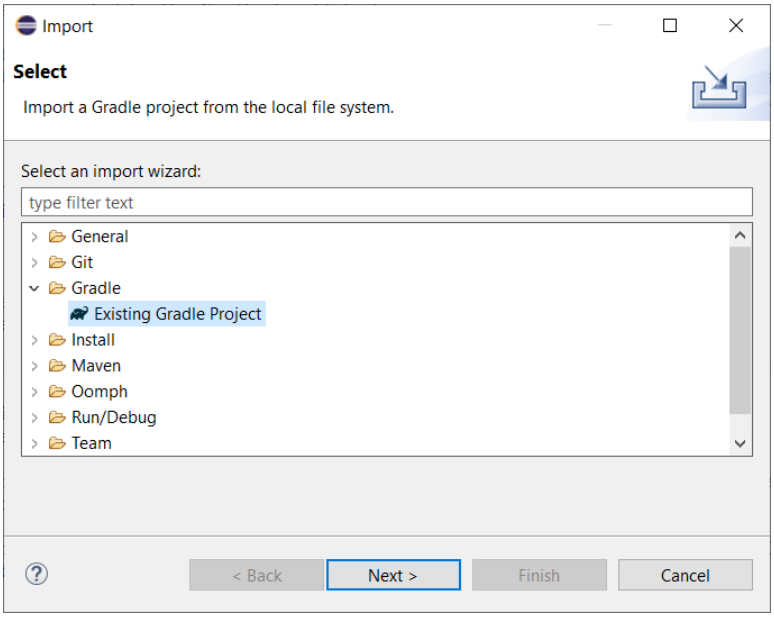


Fig. 80: Project Type Selection in Eclipse

- Select the root directory of the project.

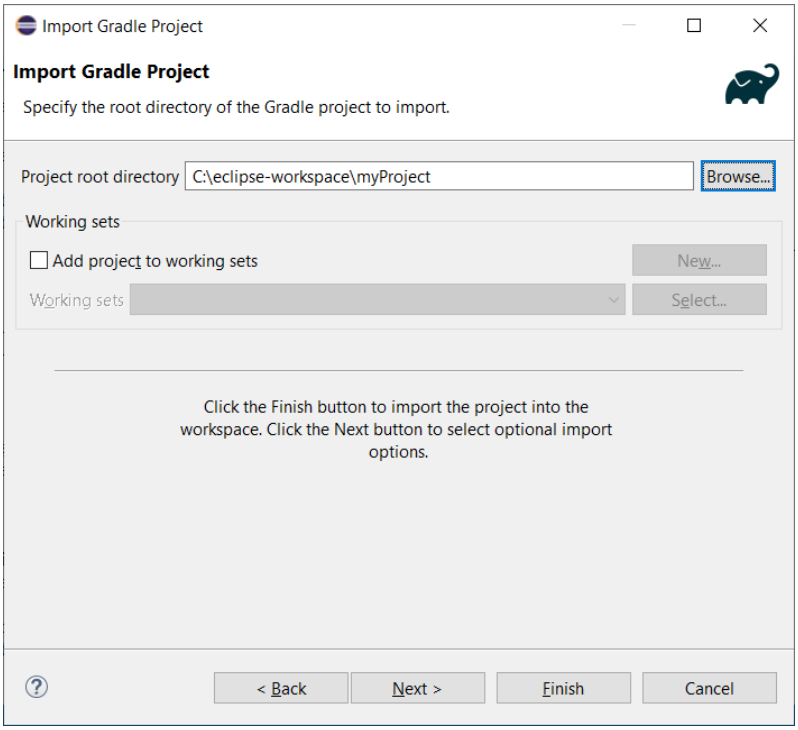


Fig. 81: Project root folder in Eclipse

- Click on the Next button and finally on the Finish button.
- Select View > Command Palette… .
- Run the Git: Clone command in the Command Palette.

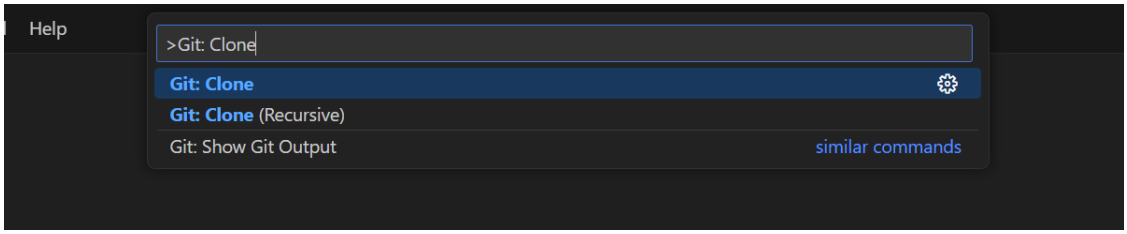


Fig. 82: Command Palette in VS Code

- Fill the URI of the VEE Port Project Template Github Repository in the Search Bar.
- Click on Clone from URL .

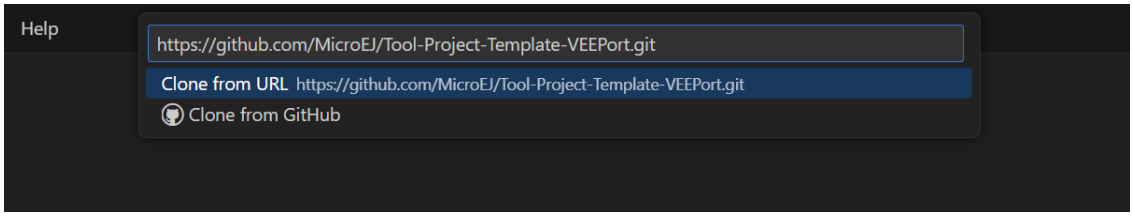


Fig. 83: Search Bar in VS Code

- In the upcoming popup, choose a folder and click on the Select as Repository Destination button.
- When the Gradle project is loaded, rename the project and change its group and version in the build.gradle.kts build script.
- Select Terminal > New Terminal .
- In the integrated terminal, run the following command at the root of the project to remove the Git Repository:

  Windows

  Linux/macOS

```
rm -r -Force .git*
```

```
rm -rf .git*
```

- Clone the VEE Port Project Template Github Repository:

```
git clone https://github.com/MicroEJ/Tool-Project-Template-VEEPort.git
```

- Remove the Git Repository from the project:

  Windows

  Linux/macOS

```
rm -r -Force .git*
```

```
rm -rf .git*
```

- Rename the project and change its group and version in the build.gradle.kts build script.

Architecture Selection

The next step is to select an *Architecture* compatible with your device instructions set and C compiler. A VEE Port requires exactly one Architecture. MicroEJ Corp. provides MicroEJ Evaluation Architectures for most common instructions sets and compilers at https://repository.microej.com/modules/com/microej/architecture. Refer to the chapter *Architectures MCU / Compiler* for the details of ABI and compiler options.

If the requested MicroEJ Architecture is not available for evaluation or to get a MicroEJ Production Architecture, please contact your MicroEJ sales representative or *our support team*.

The VEE Port project template comes with a default Architecture, defined in the Version Catalog file located at gradle/libs.versions.toml :

```
architecture = { group = "com.microej.architecture.CM4.CM4hardfp_GCC48", name = "flopi4G25", version = "8.
↪2.0" }
```

and referenced in the vee-port/build.gradle.kts file:

```
dependencies {

    microejArchitecture(libs.architecture)

}
```

If you want to use another Architecture, update the Version Catalog file accordingly.

The name of the Architecture dependency module needed for your VEE Port can be found in the chapter *Architectures MCU / Compiler*. Check the table of your corresponding Architecture and follow the link in the Module column.

Warning: We recommend to use an Architecture 8.1 minimum to have full support on the SDK features.

For example, to declare the Architecture version 8.3.0 for Arm® Cortex®-M7 microcontrollers compiled with IAR toolchain:

```
architecture = { group = "com.microej.architecture.CM7.CM7hardfp_IAR83", name = "flopi7I36", version = "8.3.
↪0" }
```

Runtime Capability

Depending on the selected Architecture, several Runtime Capabilities are available: mono , multi or tiny . The Capability used for the VEE Port can be defined thanks to the com.microej.runtime.capability property in the vee-port/configuration.properties file of the VEE Port Configuration project:

```
com.microej.runtime.capability=multi
```

Pack Import

MicroEJ Pack provides additional features on top of the MicroEJ Architecture such as Graphical User Interface or Networking. This is the primary mechanism for augmenting the capabilities of a VEE Port.

A MicroEJ Pack is a group of related files (Foundation Libraries, scripts, link files, C libraries, Simulator Mock, tools, etc.) that together provide all or part of a VEE Port capability.

A Pack can extend an Architecture with additional capabilities such as:

- Runtime Capability (e.g. *Multi-Sandbox*, *External Resources Loader*) ,
- Foundation Library Implementation (e.g. *MicroUI*, *NET*),
- Simulator (e.g. *Front Panel Mock*),
- Tool (e.g. *MicroEJ Java H*).

Note: MicroEJ Packs are optional. You can skip this section if you intend to integrate MicroEJ runtime only with custom libraries.

To add a MicroEJ Pack, add it as a dependency of the VEE Port configuration project in the build.gradle.kts file:

```
dependencies {

    // MicroEJ Architecture Specific Pack
    microejPack("com.microej.architecture.[ISA].[TOOLCHAIN]:[UID]-[NAME]-pack:[VERSION]")

    // MicroEJ Generic Pack
    microejPack("com.microej.pack.[NAME]:[NAME]-pack:[VERSION]")
```

(continues on next page)

(continued from previous page)

```
    // Legacy MicroEJ Generic Pack
    microejPack("com.microej.pack:[NAME]:[VERSION]")

}
```

For example, to declare the MicroEJ Architecture Specific Pack UI version 14.0.1 for MicroEJ Architecture flopi4G25 on Arm® Cortex®-M4 microcontrollers compiled with GNU CC toolchain:

```
dependencies {

    // MicroEJ Architecture Specific Pack
    microejPack("com.microej.architecture.CM4.CM4hardfp_GCC48:flopi4G25-ui-pack:14.0.1")

}
```

To declare the MicroEJ Generic Pack Bluetooth version 2.1.0:

```
dependencies {

    // MicroEJ Generic Pack
    microejPack("com.microej.pack.bluetooth:bluetooth-pack:2.1.0")

}
```

And to declare the Legacy MicroEJ Generic Pack Net version 9.2.3:

```
dependencies {

    // Legacy MicroEJ Generic Pack
    microejPack("com.microej.pack:net:9.2.3")

}
```

Enabling/Disabling modules of Legacy and Architecture Specific Packs

Architecture Specific Packs and Legacy Packs may contain multiple modules. They are all automatically installed during the VEE Port build, but can de disabled individually by configuration. This is not the case for the Generic Packs, which always contain only one module. Therefore, if you want to disable the module of a Generic Pack, simply remove the Pack dependency.

The modules of the Architecture Specific Packs and Legacy Packs can be enabled/disabled in the vee-port/configuration.properties file of the VEE Port Configuration project by adding the property com.microej.runtime.<module>.<feature>.enabled=true|false for each module that must be enabled/disabled. The <feature> is optional. When no feature is defined, the whole module is enabled/disabled. Here are some examples:

```
# Disable the "display_decoder_bmpm" feature of the "ui" module of the Pack UI
com.microej.runtime.ui.display_decoder_bmpm.enabled=false

# Disable the whole "ssl" module of the Pack Net
com.microej.runtime.ssl.enabled=false
```

For a complete list of the properties allowing to enable/disable modules, refer to the VEE Port Project Template.

Packs Configuration

Packs can be configured in the `vee-port/configuration.properties` file of the VEE Port Configuration project. Each Pack provides a set of option which follows the pattern name `com.microej.pack.<module>.<option>=<value>`. Here are some examples:

```
# Defines the number of bits per pixels the display device is using to render a pixel
com.microej.pack.display.bpp=argb8888

# Defines the native file system user directory
com.microej.pack.fs.user.dir=/home/microej

# Defines the native stack you want to use
com.microej.pack.net.stack=bsd
```

VEE Port Usage

Depending on your use case, there are different ways to use the VEE Port from an Application or a Library. All of them are described in the *Select a VEE Port* page.

In a nutshell,

- When the VEE Port is intended to be used by developers not having the VEE Port project locally, it must be published in a remote repository and *fetched in dependencies* by the developers. Refer to the *VEE Port Publication* section to learn how to publish a VEE Port.
- When the VEE Port project is available locally on the developers' machine, it can be integrated in the build flow of the Application or the Library. Depending on your project structure, refer to *VEE Port project inside a multi-project* or *Local VEE Port project outside a multi-project* section.

VEE Port Customization

The VEE Port Configuration project can contain an optional `vee-port/dropins` folder. The full content of this folder will be copied in the VEE Port during the build. Files in the dropins folder have the highest priority. If one file has the same path and name as a file already installed in the VEE Port, the file from the dropins folder will be used in the built VEE Port. This feature allows to add or overwrite libraries, tools, etc. into the VEE Port.

The dropins folder organization should respect the VEE Port files and folders organization. For instance, the tools are located in the sub-folder `tools`. In order to see how the VEE Port files and folders are organized, launch a VEE Port build without the dropins folder by executing the Gradle task `buildVeePort`. The built VEE Port is located in the folder `vee-port/build/vee`. Then fill the dropins folder with additional features and build again the VEE Port to get a customized VEE Port.

BSP Connection

In order to build the Executable of an Application, the BSP Connection must be configured. Refer to the *BSP Connection* section for more details.

VEE Port Publication

Publishing a VEE Port in a repository allows to make it easily available to any project.

To be able to publish a VEE Port, you have to make sure that the group and version properties are defined for the VEE Port subproject. If you have created your VEE Port project from the template, they are defined in main build.gradle.kts file:

```
allprojects {
  group = "com.mycompany.myboard"
  version = "0.1.0-RC"
}
```

You can adapt them to your need, or define them directly in the VEE Port subproject, in the vee-port/build.gradle.kts file:

```
group = "com.mycompany.myboard"
version = "1.0.0"
```

Changing the group and version properties in the main build.gradle.kts file will affect all the subprojects, whereas defining them in the vee-port/build.gradle.kts file will only affect the VEE Port subproject.

The name of the artifact can be defined in the settings.gradle.kts file with the rootProject.name property. It is set by default to the folder name of the project.

Once these properties are defined, the publication of a VEE Port is done, as any other Gradle project, by executing the publish task.

An important point to notice is that publishing a VEE Port does not publish the built VEE Port, it publishes all the configuration and dropins of the project, as well as the BSP when it is configured in Full Connection mode. The VEE Port is then built on the fly when it is required (when building the Executable of an Application for example).

Aggregated Javadoc

Starting from SDK 6 1.4.0 , a Javadoc Jar file is automatically built and published when publishing the VEE Port. This file contains the aggregated javadoc of all Libraries provided to the VEE Port with the api configuration.

BSP Publication Filtering

If the VEE Port is configured in Full BSP Connection, the BSP is included in the VEE Port when it is published. Since the BSP can be big and all the files are not necessary to build an Executable, the SDK allows to define properties in the configuration.properties of the VEE Port project to filter the BSP files in publish:

- bsp.publication.includes.pattern : comma-separated list of Ant-style files and folders patterns to include in the published VEE Port. By default all the files and folders are included.
- bsp.publication.excludes.pattern : comma-separated list of Ant-style files and folders patterns to exclude from the published VEE Port. By default no files or folders are excluded.

The paths are relative to the BSP root folder (defined by the bsp.root.dir properties of the configuration.properties file).

Here is an example which excludes 3 folders:

```
bsp.publication.includes.pattern=**/*
bsp.publication.excludes.pattern=**/VEE_PORT_EVAL/,**/WIN32/,**/Utilities/Media/
```

Refer to the Javadoc of the Gradle PatternFilterable class to have more details on the Ant-style patterns.

Filtering the VEE Runtime Library

The SDK forbids the publication of the VEE Runtime Library when publishing a VEE Port. The following error is raised in such a case:

```
SDK EULA does not allow to share MicroEJ Core IP in BSP source files: 'C:\Users\johndoe\my-project\vee-port\
↪build\tmp\publishVeePort\bsp\vee\lib\microejruntime.a'.
```

The BSP Publication Filtering feature allows to filter this file before the publication by defining this property in the configuration.properties file:

```
bsp.publication.excludes.pattern=**/microejruntime.a
```

Link-Time Option

It is possible to define custom *Application options* that can be passed to the BSP through an ELF symbol defined at link-time, hence the term *link-time option*. This allows to provide configuration options to the Application developer without the need to rebuild the BSP source code.

To define a link-time option, first choose an option name with only alphanumeric characters ([a-zA-Z][a-zA-Z0-9]* without spaces).

Proceed with the following steps by replacing [my_option] with your option name everywhere:

- Create a folder inside your *VEE Port Customization* part (e.g: vee-port/dropins/scripts/init-[my_option])
- Create an Ant init script file and put it inside (e.g: vee-port/dropins/scripts/init-[my_option]/init-[my_option].xml file). Here is the Ant init script file template content:

```
<project name="[my_option]-init">
  <target name="init/execution/[my_option]" extensionOf="init/execution" if="onBoard">
    <!-- Set option default value -->
    <property name="[my_option]" value="0"/>

    <!-- Create tmp dir -->
    <local name="link.files.dir"/>
    <microejtempfile deleteonexit="true" prefix="link[my_option]" property="link.files.dir"/>
    <mkdir dir="${link.files.dir}"/>
    <!-- Get tmp link file name -->
    <local name="link.[my_option]"/>
    <property name="link.[my_option]" value="${link.files.dir}/[my_option].lscf" />
    <echoxml file="${link.[my_option]}" append="false">
      <lscFragment>
        <defSymbol name="[my_option]" value="${[my_option]}" rootSymbol="true"/>
      </lscFragment>
    </echoxml>
    <!-- Add link file in linker's link files path -->
    <augment id="partialLink.lscf.path">
      <path location="${link.files.dir}"/>
      <path location="${jpf.dir}/link"/>
    </augment>
  </target>
</project>
```

- In your BSP source code, define an ELF symbol `[my_option]` can then be used inside C files in your BSP with:

```
// Declare the symbol as an extern global
extern int [my_option];

void my_func(void){
  // Get the symbol value
  int [my_option]_value = ((int)(&[my_option]));

  // Get the symbol value
  if([my_option]_value == 1){
    ...
  }
  else{
    ...
  }
}
```

Warning: A Link-time option should avoid to be set to `0`. Some third-party linkers consider such symbols as undefined, even if they are declared.

Test a VEE Port

The SDK provides the capability to test a VEE Port. While you can create your own tests, MicroEJ provides a set of tools and pre-defined projects aimed at simplifying the steps for validating VEE Ports in the form of the VEE Port Qualification Tools.

This page guides you through the steps to add, configure and execute a testsuite on an existing VEE Port project. It takes the example of the testsuite for the *FS* Foundation Library.

To learn more on Foundation Libraries and their Qualification, refer to *VEE Port Qualification*.

Import the Testsuite

It is assumed here that the VEE Port project already exists, with the same structure as the VEE Port Project Template, especially these folders and files:

```
|- vee-port/
|    |- validation/
|    |- build.gradle.kts
|    |- configuration.properties
|- settings.gradle.kts
```

The first step is to import the testsuite in the VEE Port project:

- Clone or download the VEE Port Project Template.
- Copy the `vee-port/validation/fs` folder into the `vee-port/validation` folder of the VEE Port project.
- Add the following line in the `settings.gradle.kts` root file:

```
include("vee-port:validation:fs")
```

At this stage, the VEE Port project structure should look like this:

```
|- vee-port/
|   |- validation/
|   |   |- fs/
|   |       |- validation/
|   |           |- microej-testsuite-common.properties
|   |       |- build.gradle.kts
|   |       |- config.properties.tpl
|   |- build.gradle.kts
|   |- configuration.properties
|- settings.gradle.kts
```

Configure a Testsuite

Select the Testsuite Target

A Testsuite can be executed on the Simulator or on the Device. The target can be configured in the `build.gradle.kts` file of the Testsuite project (so in the `vee-port/validation/fs/build.gradle.kts` file for our example), in the `testing > suites > test` block, thanks to the `microej.useMicroejTestEngine` function. The second argument of this function can be either `TestTarget.SIM` (on the Simulator) or `TestTarget.EMB` (on the Device). For example, to execute the tests on the Device, the configuration is:

```
testing {
  suites {
    val test by getting(JvmTestSuite::class) {
      microej.useMicroejTestEngine(this, TestTarget.EMB)

      ...
    }
  }
}
```

Select the Testsuite Version

For a given Foundation Library version, a specific Testsuite version should be used to validate the Abstraction Layer implementation. Refer to *Testsuite Versioning* to determine the correct Testsuite version to use.

The Testsuite version must be set in the `build.gradle.kts` file of the Testsuite project (so in the `vee-port/validation/fs/build.gradle.kts` file for our example), in the `testing > suites > test > dependencies` block. For example, to use the FS Testsuite `3.0.8`:

```
testing {
  suites {
    val test by getting(JvmTestSuite::class) {
      dependencies {
        implementation("com.microej.pack.fs:fs-testsuite:3.0.8")

        ...
      }
    }
  }
}
```

Adjust the Testsuite Configuration

The SDK provides several properties to adjust the Testsuite behavior, as well as the Tests being executed. These configuration properties can be defined in the `config.properties` file located at the root of the Testsuite project. A configuration properties file template is provided, named `config.properties.tpl`, which contains all the available properties. This file must be renamed to `config.properties`. Then it can be updated, for example to change the timeout and the number of retries:

```
# The testsuite timeout (in seconds)
microej.testsuite.timeout=600

# The number of times we'll retry a test if it fails
microej.testsuite.retry.count=1
```

Also, the Application options used for each test can be defined in the `vee-port/validation/fs/validation/microej-testsuite-common.properties` file located. For example:

```
# Java memory settings
core.memory.immortal.memory=RAM
core.memory.immortal.size=64000
```

Configure the VEE Port BSP Connection

Several configuration properties must be defined depending on the type of BSP Connection used by the VEE Port. These properties are defined in the `config.properties` file located at the root folder of the Testsuite, as described in the section above. They all start by the prefix `microej.testsuite.properties.deploy.*`. See the comments in the `config.properties` file for a description of each property, or refer to *BSP Connection* for a detailed explanation.

Configure the Execution Trace Redirection

When the Testsuite is executed, the Testsuite Engine must read the trace to determine the result of the execution. The SDK provides 2 ways to read these traces: either from a socket, or from a file.

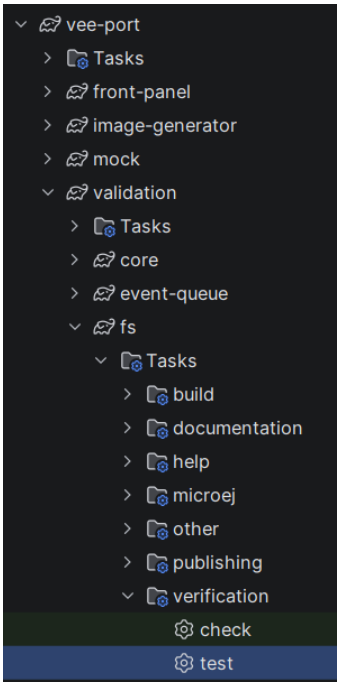
- If the VEE Port does not support the redirection of the traces to a file,
  - the `microej.testsuite.properties.testsuite.trace.ip` and `microej.testsuite.properties.testsuite.trace.port` properties must be configured to listen to the socket, for example:

    ```
    microej.testsuite.properties.testsuite.trace.ip=localhost
    microej.testsuite.properties.testsuite.trace.port=5555
    ```

  - the Serial to Socket tool must be started to redirect from the Serial Port to a socket configured above. Refer to *Serial to Socket Transmitter* to learn how to configure and start it.
- If it does,
  - the `microej.testsuite.properties.launch.test.trace.file` property must be set to `true`.

Run the Testsuite

To run the Testsuite, launch the `test` Gradle task of the Testsuite project:



Or from the command line, you must execute the following command at the root folder of the VEE Port project:

```
./gradlew :vee-port:validation:fs:test
```

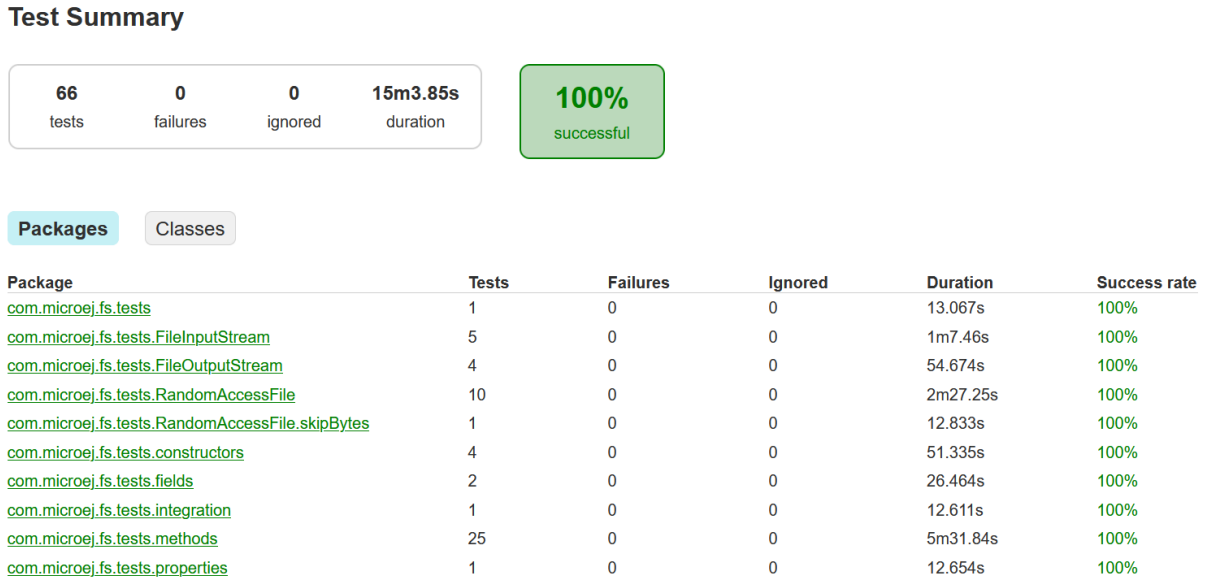
Examine the Testsuite Report

Once the Testsuite is completed, open the HTML Testsuite Report stored at `vee-port/validation/fs/build/reports/tests/test/index.html`.

The first page displays a summary of the test results: number of tests executed, number of tests failed, … Then you can click on the tests links to check the details and the logs of each test.

If necessary, the binaries produced and ran on the device by the Testsuite Engine are available at `vee-port/validation/fs/build/testsuite/output/<timestamp>/bin/<testClassName>/application.out`.

The following image shows the testsuite report fully passed:



Test Summary

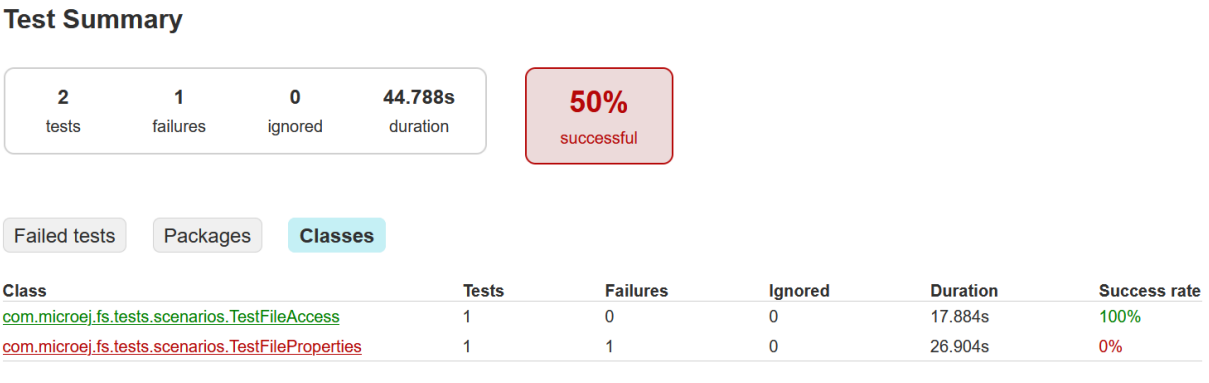
66	0	0	15m3.85s
tests	failures	ignored	duration

100% successful

Packages | Classes

Package	Tests	Failures	Ignored	Duration	Success rate
com.microej.fs.tests	1	0	0	13.067s	100%
com.microej.fs.tests.FileInputStream	5	0	0	1m7.46s	100%
com.microej.fs.tests.FileOutputStream	4	0	0	54.674s	100%
com.microej.fs.tests.RandomAccessFile	10	0	0	2m27.25s	100%
com.microej.fs.tests.RandomAccessFile.skipBytes	1	0	0	12.833s	100%
com.microej.fs.tests.constructors	4	0	0	51.335s	100%
com.microej.fs.tests.fields	2	0	0	26.464s	100%
com.microej.fs.tests.integration	1	0	0	12.611s	100%
com.microej.fs.tests.methods	25	0	0	5m31.84s	100%
com.microej.fs.tests.properties	1	0	0	12.654s	100%
com.microej.fs.tests.scenarios	12	0	0	2m53.66s	100%

If at least one test fails, the build fails and the report displays the list of failed tests:



Test Summary

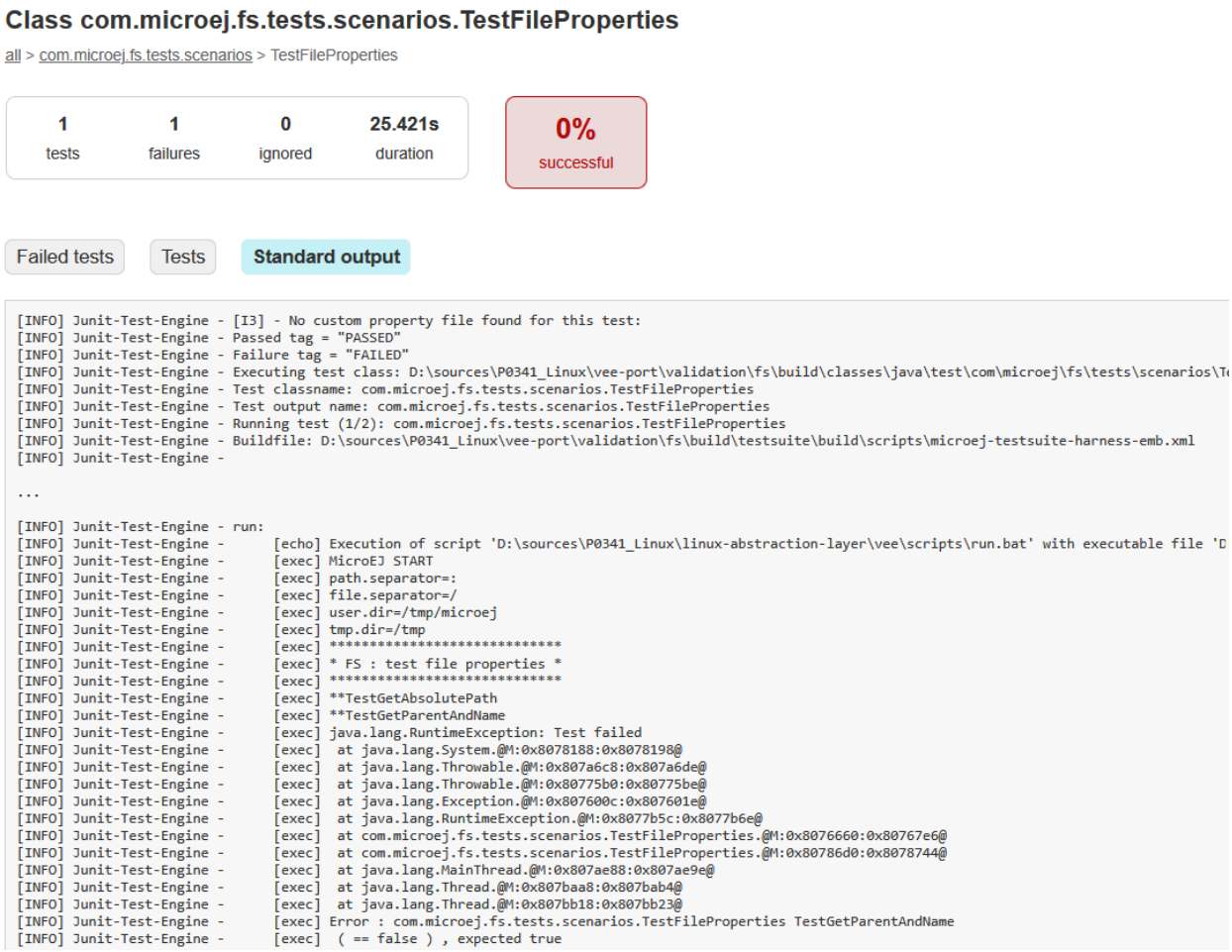
2	1	0	44.788s
tests	failures	ignored	duration

50% successful

Failed tests | Packages | Classes

Class	Tests	Failures	Ignored	Duration	Success rate
com.microej.fs.tests.scenarios.TestFileAccess	1	0	0	17.884s	100%
com.microej.fs.tests.scenarios.TestFileProperties	1	1	0	26.904s	0%

You can click on a failing test, then on the Standard Output tab to get its full logs:



Refer to *this chapter* to decrypt the stack traces.

Rerun Tests

When nothing has changed in the VEE Port or Testsuite project, running again the `test` task does not have any effect since the task is considered `UP-TO-DATE` by Gradle. If you want to force a new execution of the tests in this case, you can use the option `--rerun-tasks`:

```
./gradlew :vee-port:validation:fs:test --rerun-tasks
```

You can also filter the tests to execute, either in the `build.gradle.kts` file or with the `--tests` options, as described in *Filter the Tests*.

Debug Tests on Simulator

Warning: The properties documented in this section (debug.mode and debug.port) are available only from version 1.4.0 of the SDK 6, so make sure to use this version at least.

Tests can be debugged when running on the Simulator by following these steps:

- Make sure the Testsuite is configured to run on the Simulator by checking that the call to the microej.useMicroejTestEngine function in the testsuite configured in the build.gradle.kts . The second argument must be set to TestTarget.SIM , for example:

```
microej.useMicroejTestEngine(this, TestTarget.SIM, TestMode.MAIN)
```

- Copy the Implementation JAR of the Library you want to debug from the VEE Port located in build/testVee (the Testsuite must be executed once before to have this folder) into the root of the Testsuite project. For example, to debug the FS Library, copy the JAR file vee-port/validation/fs/build/testVee/javaLibs/fs-X.Y.jar into the folder vee-port/validation/fs .
- Add this JAR as a dependency of the Testsuite in the build.gradle.kts file. For example, add this line in the testing > suites > test > dependencies block:

```
implementation(files(layout.projectDirectory.file("fs-X.Y.jar")))
```

- Run the Testsuite by executing the test with the debug options:

```
./gradlew :vee-port:validation:fs:test -P"debug.mode"=true -P"debug.port"=8000
```

- Configure your favorite IDE as described in *Debug on Simulator*.

The debugger should connect to the Simulator and you should be able to debug your Testsuite.

Update Test Class of the Testsuite on Device

To help troubleshooting a problem in a Testsuite, it can be convienent to update a Test class. This can be done by extracting the Testsuite sources in the validation project (each step will mention how to apply it for the FS Testsuite):

- In the validation project (vee-port/validation/fs), create the src/test/java and the src/test/resources folders if they do not exist.
- Get the Testsuite RIP file. It is available in the MicroEJ Central Repository. For example, for the FS Testsuite com.microej.pack.fs:fs-testsuite:3.0.8 , the RIP file is located here.
- Extract the RIP file in the temporary folder of your choice.
- Locate the Testsuite JAR from the javaLibs folder to this RIP file. It is named fs-testsuite-x.y.jar for the FS Testsuite.
- Extract the Java sources files into the src/test/java folder (do not copy the .class files) and all other files in the src/test/resources folder.
- Add the missing dependency to make everything compile successfully:
  - Copy all the other JAR files from the javaLibs RIP folder into the validation project folder and add them as file dependencies in the dependencies. The FS Testsuite contains only the checkHelper.jar file. It must be copied into the folder vee-port/validation/fs and the following line must be added: implementation(files(layout.projectDirectory.file("checkHelper.jar")) .

- Add the following snippet of code in the build.gradle.kts file:

```
tasks.withType<com.microej.gradle.tasks.LoadVeeTask> {
  testsuitesClasspath.setFrom(configurations.create(name + "EmptyClasspath"))
}
```

- Then follow the steps related to the testsuite type:

JUnit (All Testsuites except FS, NET and SSL)

Main (FS, NET, SSL)

- Remove all classes starting by __AllTests__ and __SingleTest__ , as well as the class __AllTestClasses .
- In the build.gradle.kts file, make sure that the call to microej.useMicroejTestEngine uses the following arguments:

```
microej.useMicroejTestEngine(this, TestTarget.EMB, TestMode.JUNIT)
```

- In the build.gradle.kts file, make sure that the call to microej.useMicroejTestEngine uses the following arguments:

```
microej.useMicroejTestEngine(this, TestTarget.EMB, TestMode.MAIN)
```

- For the FS testsuite, copy the JAR vee-port/validation/fs/build/testVee/javaLibs/resourcemanager-1.0.jar into the folder vee-port/validation/fs , then add the dependency implementation(files(layout.projectDirectory.file("resourcemanager-1.0.jar")) .

- Update the Test class according to your need.
- Launch the test again with:

```
./gradlew test
```

Decrypt a Stack Trace in a Test

When a stack trace is displayed in the tests logs, it can be decrypted with the Stack Trace Reader tool. This task must be launched with the System Property set to the path of the Executable of the test. The Executable of the tests are located in the build/testsuite/output/<timestamp>/bin/<test-class-fullname>/application.out folder. For example, if a stack trace is displayed when executing the test com.microej.fs.tests.scenarios.TestFileProperties , the Executable is located at build/testsuite/output/20251110-1655-36/bin/com.microej.fs.tests.scenarios.TestFileProperties/application.out , so the stack trace can be decrypted by executing the following command:

```
./gradlew :vee-port:validation:fs:stackTraceReader -D"application.file"="build/testsuite/output/20251110-1655-36/
↪bin/com.microej.fs.tests.scenarios.TestFileProperties/application.out"
```

Then pasting the encrypted stack trace when the [INFO] Paste the MicroEJ core engine stack trace here appears.

3.4.19 Module Repository

A Module Repository is a module that bundles a set of modules in a portable ZIP file. It is a tree structure where modules organizations and names are mapped to folders.

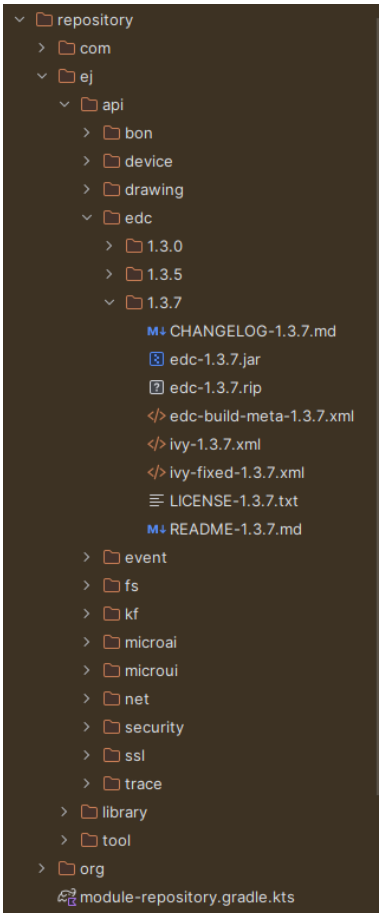


Fig. 84: Example of Module Repository Tree

A Module Repository takes its input modules from other repositories, usually the *MicroEJ Central Repository* which is itself built by MicroEJ Corp. as a Module Repository.

Create a Module Repository Project

A Module Repository project can be created as follows:

IntelliJ IDEA

Android Studio

Eclipse

Visual Studio Code

Command Line Interface

- Click File > New > Project from Version Control… .
- Select Repository URL .
- Select Git as Version control.
- Fill the URL of the Module Repository Project Template Github Repository.
- Fill the Directory in which the Project Template must be cloned.
- Click on Clone button.

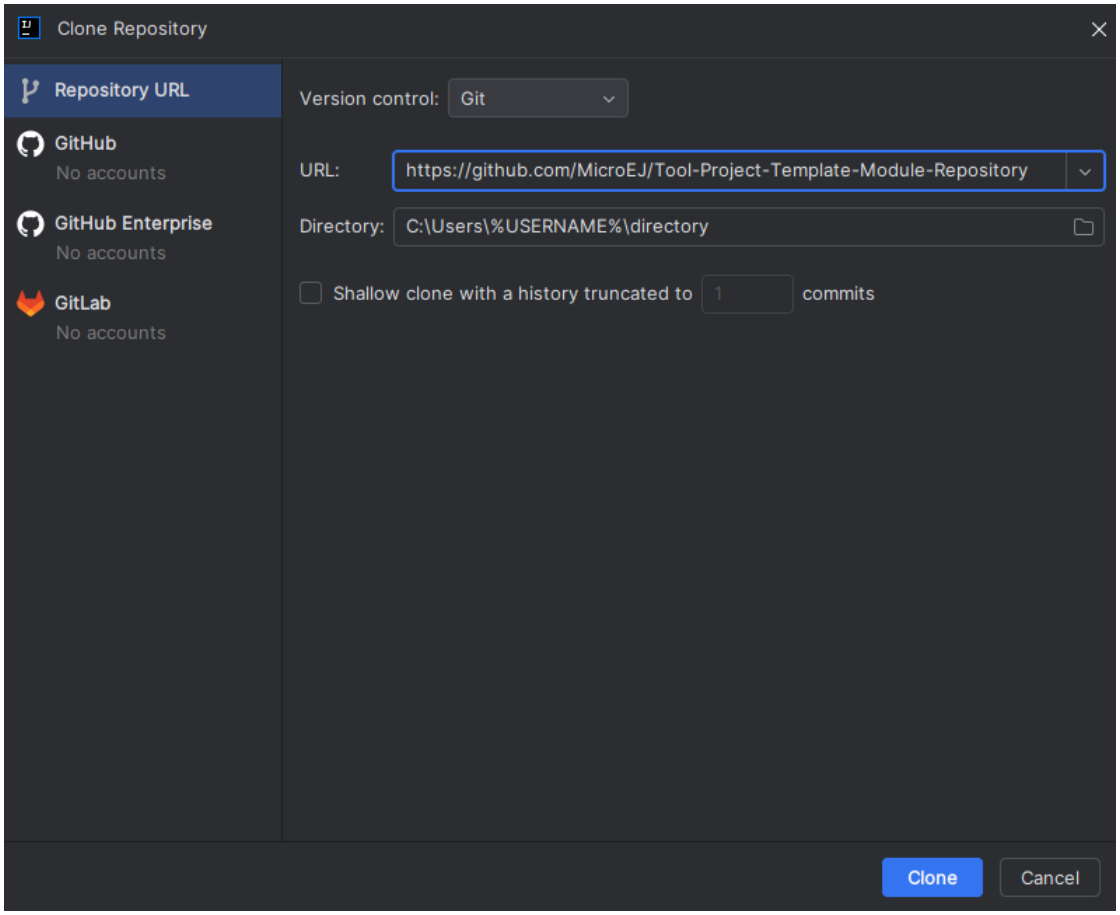


Fig. 85: Version Control Window in IntelliJ IDEA

- When the Gradle project is loaded, rename the project and change its group and version in the build.gradle.kts build script.
- Select View > Tool Windows > Terminal .
- In the integrated terminal, run the following command at the root of the project to remove the Git Repository:

  Windows

  Linux/macOS

```
rm -r -Force .git*
```

```
rm -rf .git*
```

- Click File > New > Project from Version Control… .
- Select Repository URL .
- Select Git as Version control.
- Fill the URL of the Module Repository Project Template Github Repository.
- Fill the Directory in which the Project Template must be cloned.
- Click on Clone button.

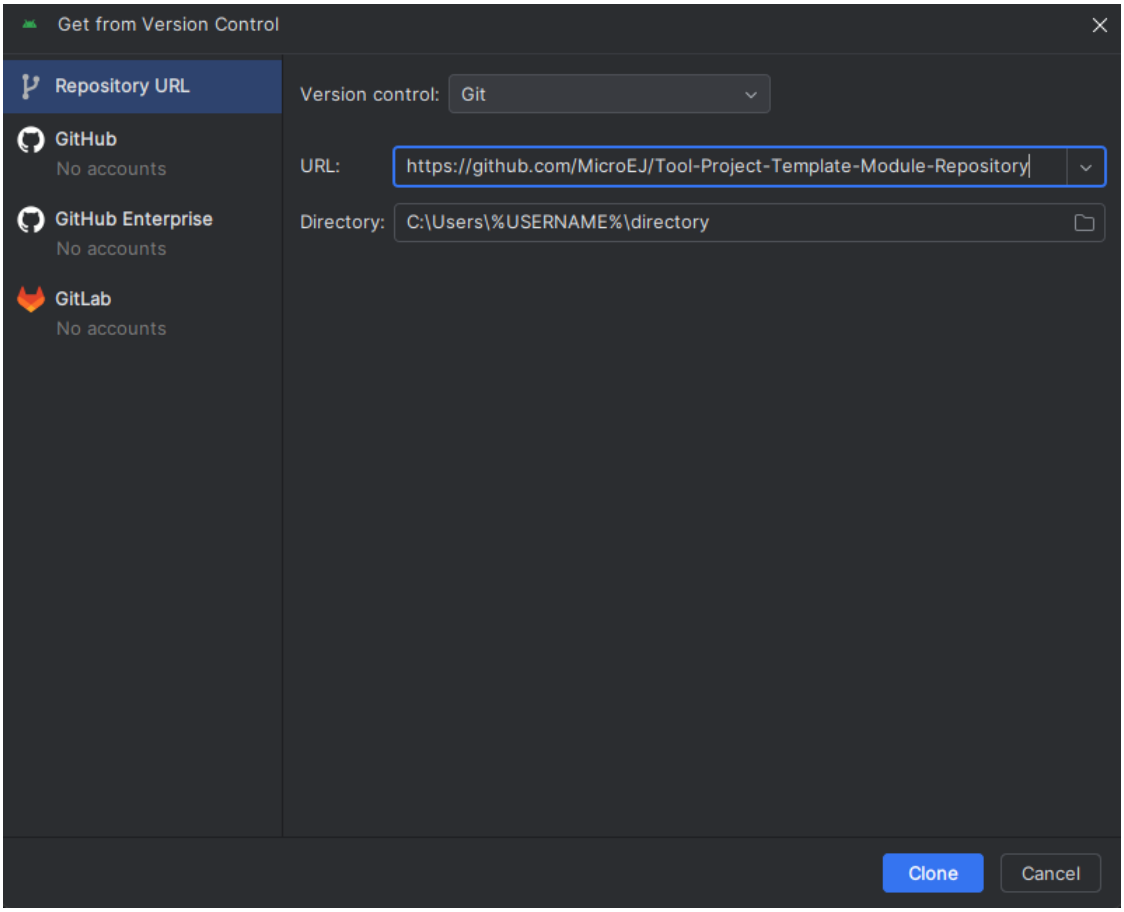


Fig. 86: Version Control Window in Android Studio

- When the Gradle project is loaded, rename the project and change its group and version in the build.gradle.kts build script.
- Select View > Tool Windows > Terminal .
- In the integrated terminal, run the following command at the root of the project to remove the Git Repository:

  Windows

  Linux/macOS

```
rm -r -Force .git*
```

```
rm -rf .git*
```

- Open a new terminal.
- Clone the Module Repository Project Template Github Repository:

```
git clone https://github.com/MicroEJ/Tool-Project-Template-Module-Repository.git
```

- Remove the Git Repository from the project:

  Windows

  Linux/macOS

```
rm -r -Force .git*
```

```
rm -rf .git*
```

- Rename the project and change its group and version in the `build.gradle.kts` build script.
- In Eclipse, click on File > Import… .
- Select the project type Gradle > Existing Gradle Project and click on the Next button.

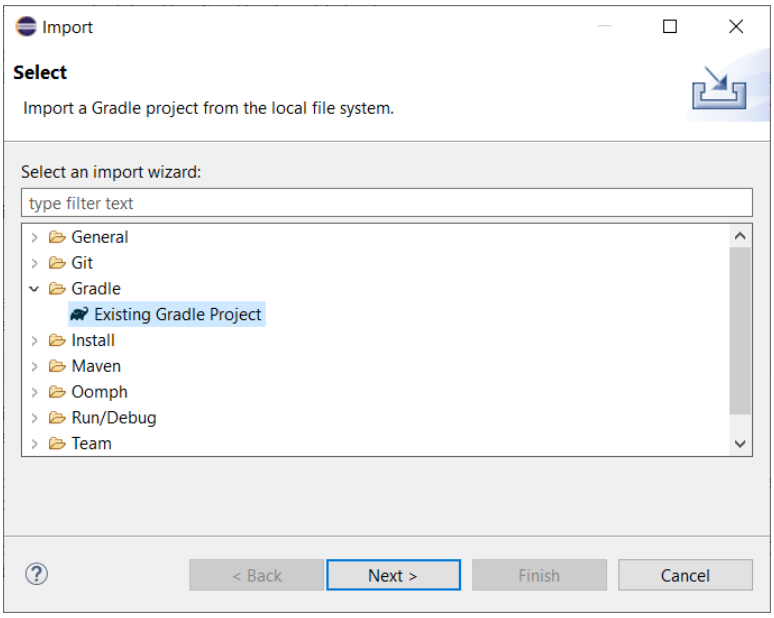


Fig. 87: Project Type Selection in Eclipse

- Select the root directory of the project.

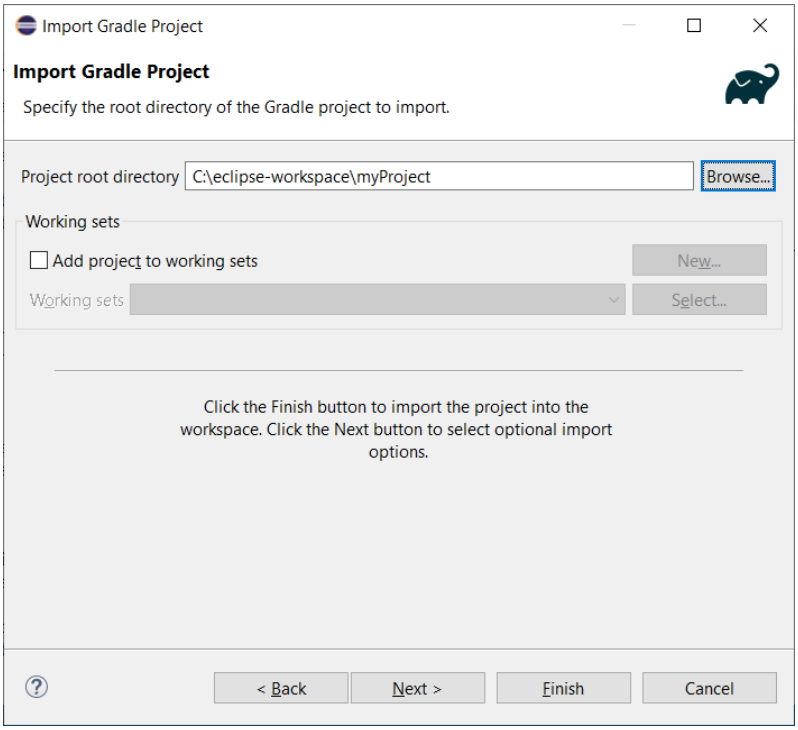


Fig. 88: Project root folder in Eclipse

- Click on the Next button and finally on the Finish button.
- Select View > Command Palette… .
- Run the Git: Clone command in the Command Palette.

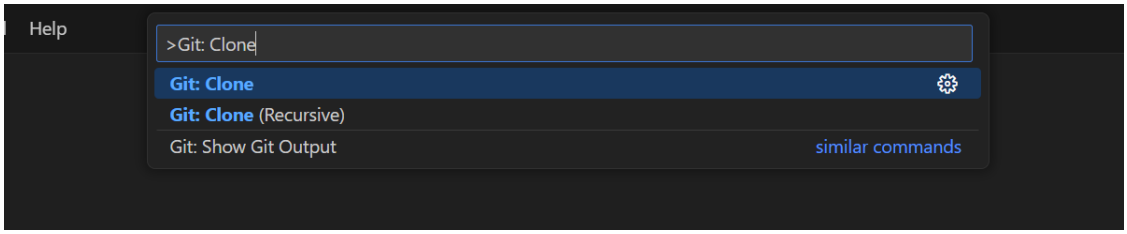


Fig. 89: Command Palette in VS Code

- Fill the URI of the Module Repository Project Template Github Repository in the Search Bar.
- Click on Clone from URL .

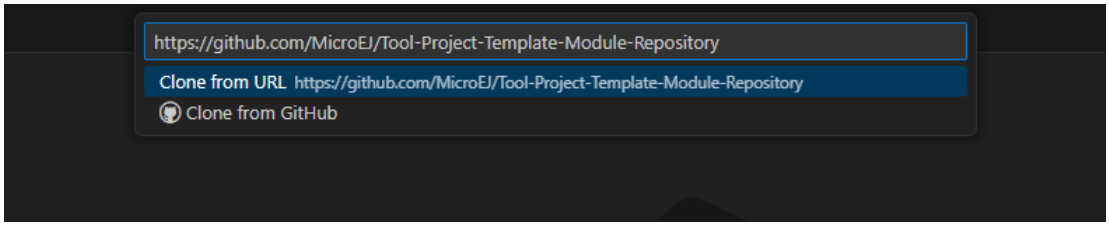


Fig. 90: Search Bar in VS Code

- In the upcoming popup, choose a folder and click on the Select as Repository Destination button.
- When the Gradle project is loaded, rename the project and change its group and version in the build.gradle.kts build script.
- Select Terminal > New Terminal .
- In the integrated terminal, run the following command at the root of the project to remove the Git Repository:

  Windows

  Linux/macOS

```
rm -r -Force .git*
```

```
rm -rf .git*
```

- Clone the Module Repository Project Template Github Repository:

```
git clone https://github.com/MicroEJ/Tool-Project-Template-Module-Repository.git
```

- Remove the Git Repository from the project:

  Windows

  Linux/macOS

```
rm -r -Force .git*
```

```
rm -rf .git*
```

- Rename the project and change its group and version in the build.gradle.kts build script.

Include Modules

Modules bundled into the Module Repository must be declared as dependencies of the project. A dependency is declared with the microejModule configuration:

```
dependencies {
    microejModule("ej.library.runtime:basictool:1.8.0")
}
```

By default, a dependency is resolved transitively but it is possible to not fetch its transitive dependencies by specifying it in the dependency's declaration:

```
dependencies {
    microejModule("ej.library.runtime:basictool:1.8.0") {
        isTransitive = false
    }
}
```

When a dependency is not resolved transitively, you must either explicitly declare its transitive dependencies in your project, or configure the Repository as a *Partial Repository*. To ensure that repository users won't have any resolution issues, a *Module Repository Check* is enabled by default.

Warning: Only Module dependencies are supported in a Module Repository project. To add a module available as project or included build, you must first publish it to be able to add it to the repository.

Configure a Partial Repository

A Partial Repository is a Repository which does not contain all the required modules. It is designed to complete one or several other repositories. For example, if you decide to use the MicroEJ Central Repository in addition to your custom Module Repository, you can include only the modules not available in the MicroEJ Central Repository. So if one module depends on `edc`, you can declare it with:

```
dependencies {
    microejModule("com.mycompany:my-lib:1.0.0") {
        isTransitive = false
    }
}
```

At resolution time, the `my-lib` dependency is resolved from the custom Module Repository and the `edc` library is resolved from the MicroEJ Central Repository.

To configure your Module Repository as a Partial Repository, you must create a file named `module-repository.gradle.kts` at the root of the Module Repository project. This file must contain the list of repositories to use at resolution time. So it should at least contain the custom Module Repository, and any additional Repository. For example if you need to add the MicroEJ Central Repository, the file content should be:

```
dependencyResolutionManagement {
    repositories {
        // Custom Module Repository
        maven {
            name = "moduleRepositoryMaven"
            url = uri(buildscript.sourceFile!!.parentFile)
        }
        ivy {
            name = "moduleRepositoryIvy"
            url = uri(buildscript.sourceFile!!.parentFile)
            patternLayout {
                artifact("[organisation]/[module]/[revision]/[artifact]-[revision](-[classifier])(.[ext])")
                ivy("[organisation]/[module]/[revision]/ivy-[revision].xml")
                setM2compatible(true)
            }
        }

        // MicroEJ Central Repository
        maven {
            name = "microEJCentral"
            url = uri("https://repository.microej.com/modules")
        }
        ivy {
            name = "microEJCentralIvy"
            url = uri("https://repository.microej.com/modules")
            patternLayout {
                artifact("[organisation]/[module]/[revision]/[artifact]-[revision](-[classifier])(.[ext])")
                ivy("[organisation]/[module]/[revision]/ivy-[revision].xml")
                setM2compatible(true)
            }
        }
        // MicroEJ SDK 6 repository for Maven/Gradle modules
        maven {
            name = "microEJForgeSDK6"
            url = uri("https://forge.microej.com/artifactory/microej-sdk6-repository-release/")
        }
```

(continues on next page)

(continued from previous page)

```
        // MicroEJ SDK 6 repository for Ivy modules
        ivy {
            name = "microEJForgeSDK6Ivy"
            url = uri("https://forge.microej.com/artifactory/microej-sdk6-repository-release/")
            patternLayout {
                artifact("[organisation]/[module]/[revision]/[artifact]-[revision](-[classifier])(.[ext])")
                ivy("[organisation]/[module]/[revision]/ivy-[revision].xml")
                setM2compatible(true)
            }
        }
    }
}

pluginManagement {
    repositories {
        // Custom Module Repository
        maven {
            name = "moduleRepositoryMaven"
            url = uri(buildscript.sourceFile!!.parentFile)
        }
        ivy {
            name = "moduleRepositoryIvy"
            url = uri(buildscript.sourceFile!!.parentFile)
            patternLayout {
                artifact("[organisation]/[module]/[revision]/[artifact]-[revision](-[classifier])(.[ext])")
                ivy("[organisation]/[module]/[revision]/ivy-[revision].xml")
                setM2compatible(true)
            }
        }

        // MicroEJ Central Repository
        maven {
            name = "microEJCentral"
            url = uri("https://repository.microej.com/modules")
        }
        ivy {
            name = "microEJCentralIvy"
            url = uri("https://repository.microej.com/modules")
            patternLayout {
                artifact("[organisation]/[module]/[revision]/[artifact]-[revision](-[classifier])(.[ext])")
                ivy("[organisation]/[module]/[revision]/ivy-[revision].xml")
                setM2compatible(true)
            }
        }
        // MicroEJ SDK 6 repository for Maven/Gradle modules
        maven {
            name = "microEJForgeSDK6"
            url = uri("https://forge.microej.com/artifactory/microej-sdk6-repository-release/")
        }
        // MicroEJ SDK 6 repository for Ivy modules
        ivy {
            name = "microEJForgeSDK6Ivy"
            url = uri("https://forge.microej.com/artifactory/microej-sdk6-repository-release/")
            patternLayout {
                artifact("[organisation]/[module]/[revision]/[artifact]-[revision](-[classifier])(.[ext])")
                ivy("[organisation]/[module]/[revision]/ivy-[revision].xml")
```

(continues on next page)

(continued from previous page)

```
            setM2compatible(true)
        }
    }
  }
}
```

This sample also includes the repository for the SDK 6 modules. It is required if you don't have declared it somewhere else (in your Gradle Init Scripts for example).

This file is directly packaged in the Module Repository archive, so it is the one to apply in order to use the custom Module Repository, as explained in *Use an Offline Module Repository*. It is also used to *check the Module Repository consistency*.

Warning: Do not forget to declare the custom Module Repository, otherwise it won't be used at resolution time.

Include a Single Artifact

When a module is added as dependency of a Module Repository project, all its artifacts are fetched and added to the repository. However, it is possible to select one artifact if you don't want to add all of them by specifying its extension in the dependency's declaration. For example, to only fetch the Virtual Device of a Kernel, you must add the `zip` extension in the dependency:

```
dependencies {
  microejModule("org.example:my-kernel:1.0.0@zip")
}
```

You can also specify the name, type, extension or classifier of the artifact if they differ from the module. For example, declare the following dependency to fetch the `imageGenerator` artifact of the UI Pack:

```
dependencies {
  microejModule("com.microej.pack.ui:ui-pack:14.0.1") {
    artifact {
      name = "imageGenerator"
      extension = "jar"
    }
  }
}
```

It is also possible to include several artifacts in the same dependency, for example:

```
dependencies {
  microejModule("com.microej.pack.ui:ui-pack:14.0.1") {
    artifact {
      name = "imageGenerator"
      extension = "jar"
    }
    artifact {
      name = "imageGenerator"
      classifier = "sources"
      extension = "jar"
    }
    artifact {
```

(continues on next page)

(continued from previous page)

```
            name = "frontpanel"
            extension = "jar"
        }
    }
}
```

Include Production Architectures

By default, the Production version of an Architecture is not included in a Module Repository. You can specify to include the Production Architecture in the `microej` block of the `build.gradle.kts` file:

```
microej {
    withArchitectureProductionVersion("com.microej.architecture.CM4.CM4hardfp_GCC48:flopi4G25:8.2.0")
}

dependencies {
    microejModule("com.microej.architecture.CM4.CM4hardfp_GCC48:flopi4G25:8.2.0")
}
```

Exclude Artifacts

When a module is added as dependency of a Module Repository project, it is possible to define a list of its artifacts that must be excluded from the Repository. You can specify the artifacts to exclude in the `microej` block of the `build.gradle.kts` file. For example, the sources and javadoc Jar files of an Application can be excluded as follows:

```
microej {
    excludeArtifacts("org.example:my-app:1.0.0", "sources,javadoc")
}

dependencies {
    microejModule("org.example:my-app:1.0.0")
}
```

The following artifacts can be excluded from the Module Repository using the `excludeArtifacts` method:

Name	Description
`sources`	Exclude the sources Jar file of a Java module.
`javadoc`	Exclude the javadoc Jar file of a Java module.
`application-wpk`	Exclude the WPK of an Application.
`executable`	Exclude the Executable of an Application.
`virtual-device`	Exclude the Virtual Device of an Application.
`runtime-environment-api`	Exclude the Runtime API jar file of a Runtime Environment.
`evaluation-architecture`	Exclude the Evaluation version of an Architecture.

Other types of artifacts cannot be excluded using the `excludeArtifacts` method. If you want to exclude a specific artifact of a dependency, you must explicitly declare all artifacts of the dependency which must be *included in the*

Repository.

Include Multiple Versions of a Module

Each dependency of a Module Repository project are resolved separately from the others, so you can provide multiple versions of the same module by declaring each dependency with the same configuration. For example, the versions of a Library can be provided with:

```
dependencies {
    microejModule("org.example:my-library:1.0.0")
    microejModule("org.example:my-library:1.2.0")
    microejModule("org.example:my-library:2.0.0")
}
```

Include the Ivy Descriptor of a Module

Most SDK 6 modules are published with an Ivy descriptor file to ensure that they can be consumed by SDK 5 projects. By default, this file is not included in a Module Repository, but it is possible to add it by calling the `withDependenciesIvyDescriptor` method in the `microej` block of the `build.gradle.kts` file:

```
microej {
    withDependenciesIvyDescriptor()
}
```

Module Repository Check

The Module Repository check consists in verifying that each declared module can be fetched from the repository. Especially, it ensures that all module's transitive dependencies are also available.

For example, if your project provides the `basictool` library with `isTransitive = false`, the following error may be raised:

```
> Module Repository check failed
  Some dependencies are missing from the Module Repository
    > Could not find ej.api:edc:1.3.3.
        Searched in the following locations:
            - <project_path>/build/tmp/checkModule/repository/ej/api/edc/1.3.3/edc-1.3.3.pom
            - <project_path>/build/tmp/checkModule/repository/ej/api/edc/1.3.3/ivy-1.3.3.xml
        Required by:
            root project : > ej.library.runtime:basictool:1.8.0
    > Could not find ej.api:bon:1.4.0.
        Searched in the following locations:
            - <project_path>/build/tmp/checkModule/repository/ej/api/bon/1.4.0/bon-1.4.0.pom
            - <project_path>/build/tmp/checkModule/repository/ej/api/bon/1.4.0/ivy-1.4.0.xml
        Required by:
            root project : > ej.library.runtime:basictool:1.8.0
```

To fix this issue, you must define the library with the transitivity enabled:

```
dependencies {
    microejModule("ej.library.runtime:basictool:1.8.0")
}
```

Alternatively, you can explicitly declare the transitive dependencies of the module:

```
dependencies {
    microejModule("ej.library.runtime:basictool:1.8.0") {
        isTransitive = false
    }
    microejModule("ej.api:edc:1.3.3")
    microejModule("ej.api:bon:1.4.0")
}
```

The Module Repository check can be run by executing the Gradle `checkModuleRepository` task:

```
./gradlew checkModuleRepository
```

This task is executed by default when *building a Module Repository*. It is also possible to disable the Module Repository for a dependency in the `microej` block of the `build.gradle.kts` file:

```
microej {
    skipModuleRepositoryCheck("ej.library.runtime:basictool:1.8.0")
}

dependencies {
    microejModule("ej.library.runtime:basictool:1.8.0")
}
```

If you don't want to check the Module Repository project, you can also *skip the task*.

The checkers supported by the *checkModule* task are also executed on all dependencies of a Module Repository project by default. The checkers can be enabled or disabled on specific dependencies by calling the following methods when declaring a dependency:

Name	Description
`microej.runCheckers`	This method takes a dependency and a comma-separated list of the names of the checkers to execute as parameters. All checkers are executed by default.
`microej.skipCheckers`	This method takes a dependency and a comma-separated list of the names of the checkers to exclude as parameters. Ignored if the dependency is also used as parameter of the `runCheckers` method.

It is possible to enable or disable the checkers of a dependency in the `microej` block of the `build.gradle.kts` file. For example:

```
microej {
    skipCheckers("ej.library.runtime:basictool:1.8.0", "readme,license,changelog")
}

dependencies {
    microejModule("ej.library.runtime:basictool:1.8.0")
}
```

The properties provided by the *checkModule* task can also be used but they are applied on the project itself, not its dependencies. For example, the Readme checker is skipped on the project but it is executed on the dependencies if the following property is defined:

```
microej {
    skippedCheckers.set("readme")
}
```

Build the Repository

The Module Repository can be built by executing the Gradle `buildModuleRepository` task:

```
./gradlew buildModuleRepository
```

The repository is generated in `build/libs/<project_name>.zip`.

Use an Offline Module Repository

When the Module Repository has been created, you can configure your projects to use it:

- Unzip the Module Repository archive at the location of your choice, for example in the `C:\module-repository` folder.
- Apply the `module-repository.gradle.kts` script found in the Module Repository unzipped folder in the `settings.gradle.kts` file of your project to declare the repository:

```
apply(file("C:\\module-repository\\module-repository.gradle.kts"))
```

> **Warning:** This line must be inserted at the start of the file, before including projects, and before using any plugins that depend on repositories.

3.4.20 Manage Versioning

The SDK 5 used a specific notation for the snapshot versions. Instead of using the `-SNAPSHOT` prefix (e.g., `1.0.0-SNAPSHOT`), it used the `-RCxxx` prefix, where `xxx` is the timestamp (e.g., `1.0.0-RC202212021535`).

In order to be able to transition from SDK 5 to SDK 6 smoothly, it is recommended to continue to publish snapshot versions with the `-RCxxx` prefix. This can be done by setting the Gradle module version with the `-RC` prefix. For example:

```
version = "1.0.0-RC"
```

The SDK will automatically append the timestamp to the version to keep the same notation than MMM. This way, SDK 5 projects will be able to fetch modules published by the SDK 6.

Note: You are free to use any version number notation you want, but you have to be aware that SDK 5 projects will not be able to depend on snapshot modules published without the `-RCxxx` prefix.

3.4.21 Manage Resolution Conflicts

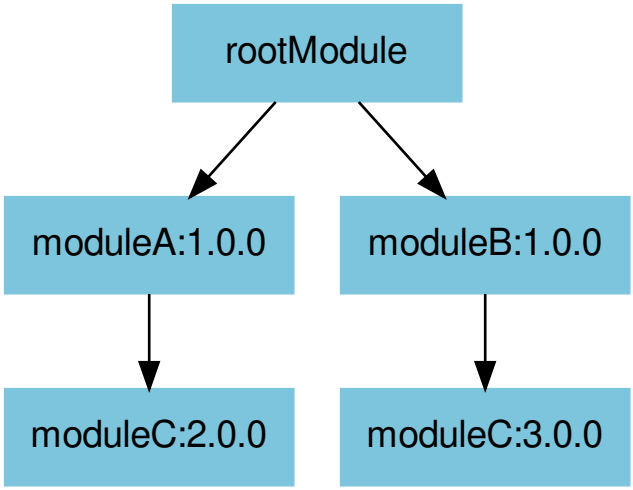
The MicroEJ Gradle plugin adds specific rules for compilation, building, resolving dependencies, versioning, and publishing.

Gradle comes with a powerful dependency manager. One of its job is to resolve the conflicts in the dependency graph, to determine which version should be added to the graph. By default, Gradle selects the highest version amongst all the versions requested for a dependency. There are ways to influence the dependencies resolution, but we believe additional rules should be added to provide a better and safer conflict resolution.

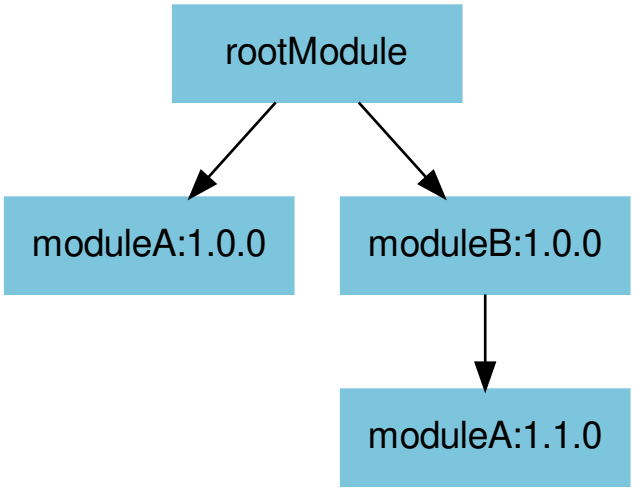
Note: You can learn more on the Gradle conflicts resolution and the way to configure it in the official documentation.

The MicroEJ Gradle plugin adds the 2 following rules:

- The resolution fails when a dependency is requested with 2 incompatible versions in the graph, according to the Semantic Versioning specification. So, it means that if 2 versions do not have the same major version, the build fails. For example, this dependency graph makes the build fail because the `moduleC` dependency is requested in 2 incompatible versions:



- The resolution fails when a transitive dependency is resolved with a higher minor version than the one declared. For example, this dependency graph makes the build fails because the `moduleA` dependency is resolved in version `1.1.0` (the highest one), which is higher than the direct declared version (`1.0.0`):



Note: This rule is not applied if *the VEE Port transitive resolution* is enabled.

If you want to come back to the Gradle default behavior, these 2 rules can be disabled by setting the `microejConflictResolutionRulesEnabled` property of the `microej` configuration block to `false` in the project build file:

```
microej {
  microejConflictResolutionRulesEnabled = false
}
```

3.4.22 Migrate an SDK 5 Project

Migrate an Application/Library Project

This page is the entry point to learn how to migrate an Application or a Library project created with the SDK 5 or lower to the SDK 6. It covers the following items:

- Build descriptor file
- Configuration
- Build scripts

Build Descriptor File

The module.ivy file of the SDK 5 project must be replaced by a build.gradle.kts file and a settings.gradle.kts file. The settings.gradle.kts contains the name of the project, whereas the build.gradle.kts file contains all the other information (module type, group, version, …).

The following chapters describe how to convert the sections of the module.ivy file to the SDK 6 format.

Build Type

The SDK 5 build type defined in the module.ivy file with the ea:build tag is replaced by a plugin in the build.gradle.kts file. For example, here is the block to add at the beginning of the file to migrate a build-microej-javalib SDK 5 module:

```
plugins {
    id("com.microej.gradle.library") version "1.4.0"
}
```

The mapping between the main SDK 5 build types and Gradle plugins is:

MMM Build Type	Gradle Plugin
build-microej-javalib	com.microej.gradle.library
build-application	com.microej.gradle.application
build-firmware-singleapp	com.microej.gradle.application
build-firmware-multiapp	com.microej.gradle.application
build-std-javalib	com.microej.gradle.jse-library
build-microej-mock	com.microej.gradle.mock
build-runtime-api	com.microej.gradle.runtime-environment

Module Information

The module information defined by the info tag in the module.ivy file are split in the 2 following descriptor files:

- settings.gradle.kts
  - The property rootProject.name replaces the module attribute.
- build.gradle.kts
  - The property group replaces the organisation attribute.
  - The property version replaces the revision attribute.

So for example, the following info tag:

```
<info organisation="com.mycompany" module="myProject" status="integration" revision="0.1.0">
```

will be converted to:

Listing 1: settings.gradle.kts

```
rootProject.name = "myProject"
```

Listing 2: build.gradle.kts

```
group = "com.mycompany"
version = "0.1.0"
```

Note: Refer to *Manage Versioning* section for more information on the way to define the module version.

Dependencies

The `dependencies` tag in the `module.ivy` file is replaced by the `dependencies` block in the `build.gradle.kts` file. Each dependency is tight to a Gradle configuration. For example, migrating a dependency used at compile time and runtime should use the `implementation` configuration, so the following dependency:

```
<dependency org="ej.api" name="edc" rev="1.3.5" />
```

will be converted to:

```
implementation("ej.api:edc:1.3.5")
```

wheras a dependency used for the tests only should use the `testIplementation` configuration, so the following dependency:

```
<dependency conf="test->*" org="ej.library.test" name="junit" rev="1.7.1"/>
```

will be converted to:

```
testImplementation("ej.library.test:junit:1.12.0")
```

Also note that this will not resolve snapshot builds since versions are explicit in SDK 6, see *this chapter* for more details. To resolve both snapshot and release versions, use `[1.0.0-RC,1.0.0]` instead of `1.0.0`.

Note: If the dependency relates to another module of the same project, you may use a multi-project structure instead (see Multi-Project Build Basics).

Refer to the *Add a Dependency* page to go further on the Gradle dependencies and configurations.

Tests

The SDK 5 supported only one testsuite by module. The SDK 6 allows to define multiple testsuites, with different configurations. Therefore, the SDK 6 requires to explicitly configure the testsuites to execute.

- If your SDK 5 project executed tests on the Simulator (the most common case),
  - Define the following testsuite in the `build.gradle.kts` file:

```
testing {
  suites {
    val test by getting(JvmTestSuite::class) {
      microej.useMicroejTestEngine(this)

      dependencies {
        implementation(project())
        implementation("ej.api:edc:1.3.5")
        implementation("ej.library.test:junit:1.12.0")
        implementation("org.junit.platform:junit-platform-launcher:1.8.2")
      }
    }
  }
}
```

- Add the additional test dependencies, if any.
- Check that the test sources are located in the `src/<testsuite-name>/java`, so in `src/test/java` with this default testsuite. Since this is the same folder used in SDK 5, there should be not change required in your project.
- Adapt the pattern of the executed tests. It is defined in SDK 5 with the property `test.run.includes.pattern`, and relied on the generated tests classes names, not the original ones. For example if you want to execute only the `MyTest` test class, it was defined like this in SDK 5:

```
<ea:property name="test.run.includes.pattern" value="**/_AllTests_MyTest.class"/>
```

In SDK 6, it is based on the original tests classes and must be defined by the `filter` object of the `test` task in the `build.gradke.kts` file. So it should be configured as follows:

```
testing {
  suites {
    val test by getting(JvmTestSuite::class) {

      ...

      targets {
        all {
          testTask.configure {
            filter {
              includeTestsMatching("MyTest")
            }
          }
        }
      }
    }
  }
}
```

If the tests pattern was the following one (meaning "All test classes"):

```
<ea:property name="test.run.includes.pattern" value="**/_AllTests_*.class"/>
```

there is no need to add a filter, all tests classes are executed by default in SDK 6.

- If your SDK 5 project executed tests on the Device using the KF Testsuite (since it was the only way to execute such tests with the SDK 5),

- If you really want to execute test projects with a Kernel and Sandboxed Applications on Device (formely called KF Testsuite), refer to the *Setup a KF Testsuite* page.
- If you want to execute tests classes (like on the Simulator, but on the Device),
  * Follow the steps to execute the tests on the Simulator.
  * Replace the line `microej.useMicroejTestEngine(this)` by `microej.useMicroejTestEngine(this, TestTarget.EMB)` .
  * Add the `import` statement at the beginning of the file:

```
import com.microej.gradle.plugins.TestTarget
```

Refer to the *Test a Project* page to go further on the SDK 6 testsuites configurations.

Configuration Folder

The `build` folder located at the root of the project and containing the Application configuration properties is replaced by the `configuration` folder. This change is required since Gradle uses the `build` folder to store the generated files and artifacts (equivalent of the MMM `target~` folder).

Specific Configuration

Some configuration options are available in SDK 6 in a different way than in SDK 5. This chapter goes through all this specific configuration options.

You can refer to the *Module Natures* page for a complete list of configurations.

Main class of Standalone Application

The main class of a Standalone Application is defined in SDK 5 with the property `application.main.class` in the `module.ivy` file:

```
<ea:property name="application.main.class" value="com.mycompany.Main"/>
```

It must now be defined in SDK 6 by the `applicationEntryPoint` property of the `microej` block in `build.gradke.kts` file:

```
microej {
  applicationEntryPoint = "com.mycompany.Main"
}
```

Feature Entry Point class of Sandboxed Application

The Feature Entry Point class of a Sandboxed Application is defined in SDK 5 with the property `entryPoint` in the `*.kf` file:

```
entryPoint=com.mycompany.MyFeature
```

It must now be defined in SDK 6 by the `applicationEntryPoint` property of the `microej` block in the `build.gradke.kts` file:

```
microej {
  applicationEntryPoint = "com.mycompany.MyFeature"
}
```

Example

This section gives an example of a migration from a SDK 5 Application project to SDK 6. Here are the projects strucuture side by side:

SDK 5 Project	SDK 6 Project

<pre>|- src
| |- main
| | |- java
| | |- resources
| |- test
| | |- java
| | |- resources
|- build
| |- common.properties
|- module.ivy
|- module.ant</pre>
	<pre>|- src
	| |- main
	| | |- java
	| | |- resources
	| |- test
	| | |- java
	| | |- resources
	|- configuration
	| |- common.properties
	|- build.gradle.kts
	|- settings.gradle.kts</pre>

And here the migration from a `module.ivy` file to a `build.gradle.kts` file and a `settings.gradle.kts` file:

SDK 5 and lower

Listing 3: module.ivy

```
<ivy-module version="2.0" xmlns:ea="http://www.easyant.org" xmlns:m="http://ant.apache.org/ivy/extra"␣
↪xmlns:ej="https://developer.microej.com" ej:version="2.0.0">
  <info organisation="com.mycompany" module="myProject" status="integration" revision="0.1.0">
    <ea:build organisation="com.is2t.easyant.buildtypes" module="build-application" revision="9.2.+">
      <ea:property name="test.run.includes.pattern" value="**/_AllTests_*.class"/>
    </ea:build>
  </info>

  <configurations defaultconfmapping="default->default;provided->provided">
    <conf name="default" visibility="public" description="Runtime dependencies to other artifacts"/>
    <conf name="provided" visibility="public" description="Compile-time dependencies to APIs provided by␣
↪the platform"/>
    <conf name="platform" visibility="private" description="Build-time dependency, specify the platform to use
↪"/>
    <conf name="documentation" visibility="public" description="Documentation related to the artifact␣
↪(javadoc, PDF)"/>
    <conf name="source" visibility="public" description="Source code"/>
    <conf name="dist" visibility="public" description="Contains extra files like README.md, licenses"/>
    <conf name="test" visibility="private" description="Dependencies for test execution. It is not required for␣
↪normal use of the application, and is only available for the test compilation and execution phases."/>
    <conf name="microej.launch.standalone" visibility="private" description="Dependencies for standalone␣
↪application. It is not required for normal use of the application, and is only available when launching the main␣
↪entry point on a standalone MicroEJ launch."/>
  </configurations>
```

(continues on next page)

(continued from previous page)

```
  <publications>
    <!-- keep this empty if no specific artifact to publish -->
    <!-- must be here in order to avoid all configurations for the default artifact -->
  </publications>

  <dependencies>
    <!--
      Put your custom Runtime Environment dependency here. For example:

      <dependency org="com.company" name="my-runtime-api" rev="1.0.0" conf="provided->runtimeapi" />
    -->
    <!--
      Or put direct dependencies to MicroEJ libraries if your Application is not intended to run on a specific␣
↪custom Runtime Environment.
    -->
    <dependency org="ej.api" name="edc" rev="1.3.5" />
    <dependency org="ej.api" name="kf" rev="1.6.1" />

    <dependency conf="test->*" org="ej.library.test" name="junit" rev="1.7.1"/>

    <dependency org="com.microej.platform.esp32.esp-wrover-kit-v41" name="HDAHT" rev="1.8.0" conf=
↪"platform->default" transitive="false"/>
  </dependencies>
</ivy-module>
```

SDK 6

Listing 4: settings.gradle.kts

```
rootProject.name = "myProject"
```

Listing 5: build.gradle.kts

```
plugins {
    id("com.microej.gradle.application") version "1.4.0"
}

group = "com.mycompany"
version = "0.1.0"

dependencies {
  implementation("ej.api:edc:1.3.3")
  implementation("ej.api:kf:1.6.1")

  testImplementation("ej.library.test:junit:1.12.0")

  microejVee("com.microej.platform.esp32.esp-wrover-kit-v41:HDAHT:1.8.2")
}
```

Build Scripts

SDK 5 supports the use of the `module.ant` and `override.module.ant` to customize the build process. These files are not supported anymore with Gradle. Instead, since Gradle build files are code, customizations can be applied directly in the build files.

As an example, defining a property conditionnaly is done as follows in a `module.ant` file:

```
<target name="my-project:define-properties" extensionOf="compile">
  <condition property="myProperty" value="myValue">
    <not><equals arg1="${anotherProperty}" arg2="anotherValue"/></not>
  </condition>
</target>
```

and as follows in a `build.gradle.kts` file:

```
var myProperty = ""
tasks.register("defineProperties") {
  if(project.properties["anotherProperty"] == "anotherValue") {
    myProperty = "myValue"
  }
}

tasks.compileJava {
  dependsOn("defineProperties")
}
```

Migrate a VEE Port Project

This page explains how to migrate an VEE Port project created with the SDK 5 or lower to the SDK 6.

Project Structure

Even if it is not mandatory, a VEE Port project is most of the time structured as a multi-project. Indeed, it generally contains other subprojects than the configuration module, for example a Front Panel or an Image Generator subproject. It can also contains the BSP.

Whereas it was not necessarily structured as a meta-build in SDK 5, it is recommended to structure it as a multi-project in SDK 6.

This brings to the following recommended structure for the VEE Port multi-project:

```
|- my-board-vee-port/
|    |- bsp/
|    |- demo-application/ (optional)
|    |    |- src/main/java/
|    |- vee-port/
|    |    |- extensions/
|    |    |    |- my-front-panel/ (optional)
|    |    |    |- my-image-generator/ (optional)
|    |    |    |- microui/ (optional)
|    |    |    |- ...
|    |    |- validation/
|    |    |    |- core/ (optional)
|    |    |    |- fs/ (optional)
```

(continues on next page)

(continued from previous page)

```
|   |   |   |- ...
|   |   |- dropins/ (optional)
|   |   |- configuration.properties
|   |   |- build.gradle.kts
|   |- settings.gradle.kts
```

This migration guide will go in details in each changes, but as a quick overview, here are the main changes compared to SDK 5:

- All root modules have been moved under a `vee-port` folder, except for the BSP folder.
- The configuration module (`*-configuration`) is now the `vee-port` module, which includes configuration properties, `dropins` folder, as well as the other modules required by the VEE Port.
- All configuration properties file have been gathered in the `vee-port/configuration.properties` (except for `microui`).
- The specific components required by the VEE Port are now in the `vee-port/extensions` folder, such as Front Panel or Image Generator modules.
- The MicroUI extension has been moved in the `vee-port/extensions/microui` folder.

New Development Flow

With the SDK 5, the development flow consisted in 3 main steps:

- Make a change in one the VEE Port projects (for example `*-configuration` or `*-fp`).
- Build the VEE Port configuration project (`*-configuration`), which generates a built VEE Port in the sibling folder.
- Run or build an Application (after defining the path to the VEE Port in the `module.ivy` file) on this VEE Port.

While this development flow is still possible with the SDK 6, the recommended approach is to make the Application depend directly on the VEE Port project (the way to do it is detailed in the following sections). This allows to automatically build the updated projects (VEE Port Configuration, Front Panel, Mock, …) when running or building the Application. There is no need to build the VEE Port anymore before using it, Gradle does it for you. It automatically detects that the VEE Port has changes and triggers the tasks to build the updated components, and only the updated components. Therefore, the development flow in SDK 6 is now:

- Make a change in the VEE Port project.
- Run or build an Application on this VEE Port.

Settings Build File

A multi-project requires a `settings.gradle.kts` file located in the root folder. Its main purposes are to optionaly define a name for the multi-project and to define its subprojects. Create this file and add this line with the name of your choice:

```
rootProject.name = "my-board-vee-port"
```

Configuration Subproject

The Configuration (*-configuration) subproject is still there, but it is recommended to name it vee-port . Follow these steps to migrate it:

- Rename the folder to vee-port .
- Replace the module.ivy file by a build.gradle.kts file.
- Use the com.microej.gradle.veeport plugin in this file:

```
plugins {
  id("com.microej.gradle.veeport") version "0.19.0"
}
```

- Still in the build.gradle.kts file, define the group and version property, which respectively replace the organisation and revision of the module.ivy file. For example:

```
group = "com.mycompany"
version = "0.1.0"
```

Note: Refer to *Manage Versioning* section for more information on the way to define the module version.

- Define the dependencies previously defined in the module.ivy file. In SDK 6, dependencies are defined in a dependencies block in the build.gradle.kts file, and each type of VEE Port component must be defined with a dedicated configuration:
  - Architecture: microejArchitecture
  - Pack: microejPack
  - Mock: microejMock
  - Front Panel: microejFrontPanel
  - Tool: microejTool

  For example the Architecture defined like this in SDK 5:

```
<dependencies>
  <dependency group="com.microej.architecture.CM7.CM7hardfp_GCC48" name ="flopi7G26" rev="8.1.1
↪">
    <artifact name="flopi7G26" m:classifier="eval" ext="xpf" />
  </dependency>
</dependencies>
```

  must be changed to this in build.gradle.kts file:

```
dependencies {
  microejArchitecture("com.microej.architecture.CM7.CM7hardfp_GCC48:flopi7G26:8.1.1")
}
```

  Note: Note that the Architecture usage (eval or prod) is not defined in the VEE Port configuration anymore. It is now set by the project which consumes the VEE Port (for example an Application).

  And the Pack defined like this in SDK 5:

```
<dependency group="com.microej.pack" name ="fs" rev="6.0.4" />
```

must be changed to this in SDK 6:

```
microejPack("com.microej.pack:fs:6.0.4")
```

- Create a `configuration.properties` file at the root of the `vee-port` folder.
- Move the property related to the Runtime Capability from the file `mjvm/mjvm.properties` to the `vee-port/configuration.properties` file, then delete the `mjvm/mjvm.properties` file.
- Move all configuration properties related to the BSP (`bsp/bsp.properties`) into the `vee-port/configuration.properties` file. Each property name must be prefixed by `bsp.<name>.` . For example, the `microejapp.relative.dir` property must be moved as `bsp.microejapp.relative.dir` in the `vee-port/configuration.properties` file.
- Move the `microui` folder (if it exists) into the `vee-port/extensions` folder.
- Move the configuration properties related to all the other Packs (`fs/fs.properties` , ...) into the `vee-portconfiguration.properties` file. Each property name must be prefixed by `com.microej.pack.<name>.` , where `<module>` is the name of the Pack. For example, the `bpp` property defined in the `display/display.properties` file must be moved as `com.microej.pack.display.bpp` in the `vee-portconfiguration.properties` file.
- Delete old configuration properties files ((`bsp/bsp.properties` , `display/display.properties` , `fs/fs.properties` , ...) can be deleted.
- Enable disable Pack modules if required, as described in *Enabling/Disabling modules of Legacy and Architecture Specific Packs*.
- Delete the `.platform` file located at the root of the Configuration subproject, since it is now obsolete.
- Delete all the other files related to SDK 5:
  - `module.ant`
  - `override.module.ant`
  - `module.properties`
  - `configuration.xml`
  - `build` folder
- The `dropins` folder was used as a destination folder for some generated files in SDK 5, so it may contain more than the committed files. So make sure to clean it to keep only committed files. This can be done with git with the following command in the `dropins` folder:

```
git clean -xdf
```

Front Panel Project

The Front Panel is generally a subproject of the VEE Port multi-project. These steps must be followed to migrate a Front Panel subproject:

- Move the Front Panel folder into the `vee-port/extensions` folder.
- Replace the `module.ivy` file by a `build.gradle.kts` file.
- Use the `com.microej.gradle.mock-frontpanel` plugin in this file:

```
plugins {
   id("com.microej.gradle.mock-frontpanel")
}
```

- Define the dependencies previously defined in the `module.ivy` file. A Front Panel generally depends on libraries, so they can be defined with the `implementation` configuration. For example to declare a dependency on the Front Panel framework library, use:

```
implementation("ej.tool.frontpanel:framework:1.1.0")
```

  Also note that the Front Panel library from the UI Pack must be declared with the `name` and `extension`:

```
implementation("com.microej.pack.ui:ui-pack:14.0.1") {
    artifact {
        name = "frontpanel"
        extension = "jar"
    }
}
```

- Include the Front Panel subproject in the multi-project in the `settings.gradle.kts` file:

```
include(":vee-port:front-panel")
project(":vee-port:front-panel").projectDir = file("vee-port/extensions/front-panel")
```

- Make the VEE Port configuration subproject depend on the Front Panel subproject by adding a project dependency in the `vee-port/build.gradle.kts` file:

```
microejFrontPanel(project(":vee-port:front-panel"))
```

Mock

If the VEE Port project contains Mock subprojects, they must be migrated by following these steps:

- Move the Mock folder into the `vee-port/extensions` folder.
- Replace the `module.ivy` file by a `build.gradle.kts` file.
- Use the `com.microej.gradle.mock` plugin in this file:

```
plugins {
   id("com.microej.gradle.mock")
}
```

- Define the dependencies previously defined in the `module.ivy` file. A Mock generally depends on libraries, so they can be defined with the `implementation` configuration.
- Include the Mock subproject in the multi-project in the `settings.gradle.kts` file:

```
include(":vee-port:mock")
project(":vee-port:mock").projectDir = file("vee-port/extensions/mock")
```

- Make the VEE Port configuration subproject depend on the Mock subproject by adding a project dependency in the `vee-port/build.gradle.kts` file:

```
microejMock(project(":vee-port:mock"))
```

- Delete all the files related to SDK 5:

- module.ant
- content/scripts/init-* folder

Tool Subproject

If the VEE Port project contains Tool subprojects (such as an Image Generator), they must be migrated by following these steps:

- Move the Tool folder into the vee-port/extensions folder.
- Replace the module.ivy file by a build.gradle.kts file.
- Use the com.microej.gradle.j2se-library plugin in this file:

```
plugins {
  id("com.microej.gradle.j2se-library")
}
```

- Define the dependencies previously defined in the module.ivy file. A Tool generally depends on libraries, so they can be defined with the implementation configuration. Note that when the Tool is an Image Generator and depends on the Image Generator library from the UI Pack, the dependency must be declared with the name and extension :

```
implementation("com.microej.pack.ui:ui-pack:14.0.1") {
    artifact {
        name = "imageGenerator"
        extension = "jar"
    }
}
```

- Include the Tool subproject in the multi-project in the settings.gradle.kts file:

```
include(":vee-port:image-generator")
project(":vee-port:image-generator").projectDir = file("vee-port/extensions/image-generator")
```

- Make the VEE Port subproject depend on the Tool subproject by adding a project dependency in the vee-port/build.gradle.kts file:

```
microejTool(project(":vee-port:image-generator"))
```

Testsuites Project

These steps must be followed to migrate the Testsuites:

- Move the Testsuite folder into the vee-port/validation folder.

Then for each Testsuite:

- Replace the module.ivy file by a build.gradle.kts file.
- Use the com.microej.gradle.testsuite plugin in this file:

```
plugins {
  id("com.microej.gradle.testsuite")
}
```

- The tested VEE Port was defined in SDK 5 in the config.properties file, with the target.platform.dir property. In SDK 6, it is done by declaring the VEE Port Configuration project as a project dependency:

```
dependencies {
  testMicroejVee(project(":vee-port"))
}
```

  and including the testsuite project in the settings.gradle.kts file of the multi-project:

```
include("java-testsuite-runner-security")
project(":java-testsuite-runner-security").projectDir = file("validation/security/java-testsuite-runner-security
↪")
```

  The second line is required when the testsuite subproject is not directly in a subfolder of the multi-project. The path set as the value is relative to the root folder of the multi-project.

  Once done, you can delete the target.platform.dir property in the config.properties file.

- define the testsuite in the build.gradle.kts file:

```
testing {
  suites {
    val test by getting(JvmTestSuite::class) {
      microej.useMicroejTestEngine(this, TestTarget.EMB, TestMode.MAIN)

      dependencies {
        implementation(project()) // (1)
        implementation("ej.library.test:junit:1.12.0") // (2)
        implementation("org.junit.platform:junit-platform-launcher:1.8.2") // (3)

        implementation("ej.api:security:1.4.0") // (4)
        implementation("com.microej.pack.security:security-1_4-testsuite:1.3.0") // (4)
      }
    }
  }
}
```

  The testsuite dependencies must contain:

  - the project *(1)*
  - the JUnit libraries *(2)*
  - the Foundation Library to test *(3)*
  - the Testsuite related to the Foundation Library *(4)*

- The patterns of the included and excluded test classes was defined with the test.run.includes.pattern and test.run.excludes.pattern properties in the config.properties file. There must be now defined directly in the testsuite configuration in the build.gradle.kts file, by using the standard Gradle filter feature:

```
testing {
  suites {
    val test by getting(JvmTestSuite::class) {
      microej.useMicroejTestEngine(this, TestTarget.EMB, TestMode.MAIN)

      dependencies {
        ...
      }
```

(continues on next page)

(continued from previous page)

```
        targets {
            all {
                testTask.configure {
                    filter {
                        excludeTestsMatching("*AllTestClasses")
                        excludeTestsMatching("*SingleTest*")
                        excludeTestsMatching("*AllTests_MessageDigestTest")
                        excludeTestsMatching("*AllTests_KeyPairGeneratorTest")
                    }
                }
            }
        }
    }
}
```

Refer to the Gradle documentation on test filtering for more details.

Once done, you can delete the `test.run.includes.pattern` and `test.run.excludes.pattern` properties in the `config.properties` file.

- Delete all the files related to SDK 5:
  - `module.ant`
  - `override.module.ant`

BSP

It is recommended to keep the BSP folder at the root of the multi-project. The paths to the BSP can be updated in the VEE Port configuration (`vee-port/configuration.properties`) if necessary.

The `${project.parent.dir}` variable can be used to refer to the project root folder. For example with the recommended structure described at the beginning of this page (the `bsp` is at the root of the project), add this line in the `vee-port/configuration.properties` file:

```
bsp.root.dir=${project.parent.dir}/bsp
```

Refer to the *BSP Connection* documentation for more details.

Wrap up

At the end of the migration, you should have a structure similar to the one presented at the beginning of this page. The `settings.gradle.kts` should look like:

```
// Define the VEE Port mulit-project name
rootProject.name = "my-board-vee-port"

// Include the subprojects
include("vee-port", "vee-port:front-panel", "vee-port:mock", "vee-port:image-generator", "demo-application")
include("vee-port:validation:java-testsuite-runner-core")
include("vee-port:validation:java-testsuite-runner-fs")

// Define the paths of the subprojects
project(":vee-port:front-panel").projectDir = file("vee-port/extensions/front-panel")
```

(continues on next page)

(continued from previous page)

```
project(":vee-port:image-generator").projectDir = file("vee-port/extensions/image-generator")
project(":vee-port:validation:java-testsuite-runner-core").projectDir = file("vee-port/validation/core/java-testsuite-
↪runner-core")
project(":vee-port:validation:java-testsuite-runner-fs").projectDir = file("vee-port/validation/fs/java-testsuite-
↪runner-fs")
```

3.4.23 Module Natures

This page describes the most common module natures as follows:

- **Plugin Name**: the build type name, derived from the module nature name: com.microej.gradle.[NATURE_NAME] .
- **Documentation**: a link to the documentation.
- **Tasks**: tasks available from the module nature, with the graph of their relationships.
- **Configuration**: properties that can be defined to configure the module. Properties are defined inside the microej block of the build.gradle.kts file.

Add-On Library

Plugin Name: com.microej.gradle.library

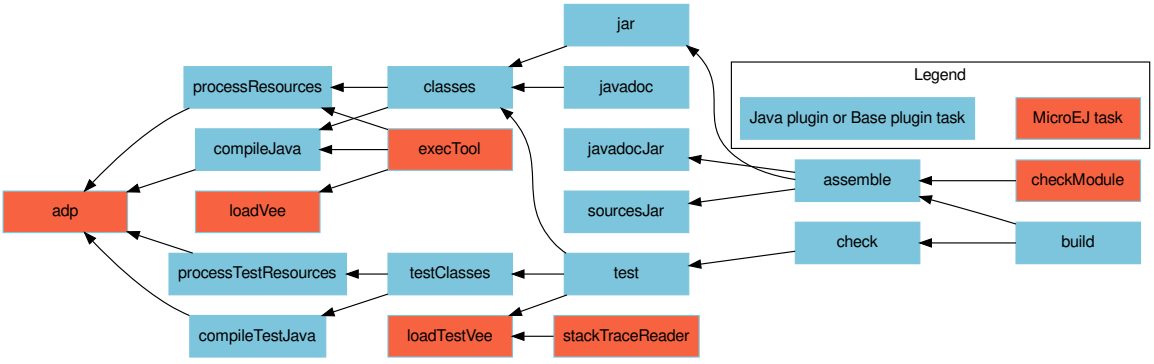
Documentation: *Libraries*

Template: Add-On Library Project Template

Tasks:

This plugin adds the following tasks to your project:

- tasks of the Gradle Java plugin
- *adp*
- *loadVee*
- *checkModule*
- *execTool*



Configuration:

This module nature inherits from the configuration of all its tasks.

Application

Plugin Name: com.microej.gradle.application

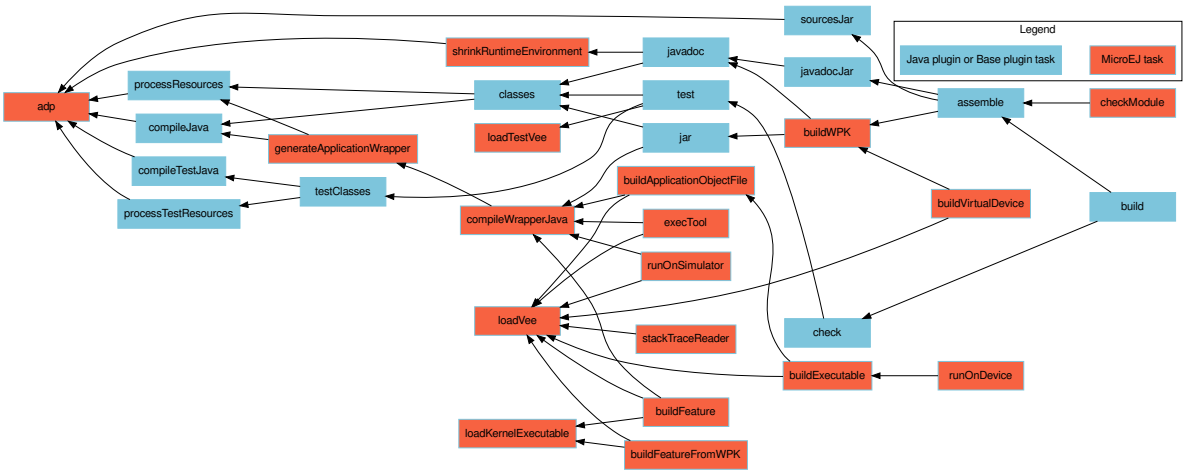
Documentation: *Standalone Application*, *Sandboxed Application*, *Kernel Developer Guide*

Template: Application Project Template

Tasks:

This plugin adds the following tasks to your project:

- tasks of the Gradle Java plugin
- *adp*
- *loadVee*
- *runOnSimulator*
- *checkModule*
- *buildApplicationObjectFile*
- *buildExecutable*
- *buildWPK*
- *buildVirtualDevice*
- *loadKernelExecutable*
- *buildFeature*
- *runOnDevice*
- *execTool*
- *generateApplicationWrapper*
- *compileWrapperJava*
- *buildFeatureFromWPK*
- *shrinkRuntimeEnvironment*



Java SE Library

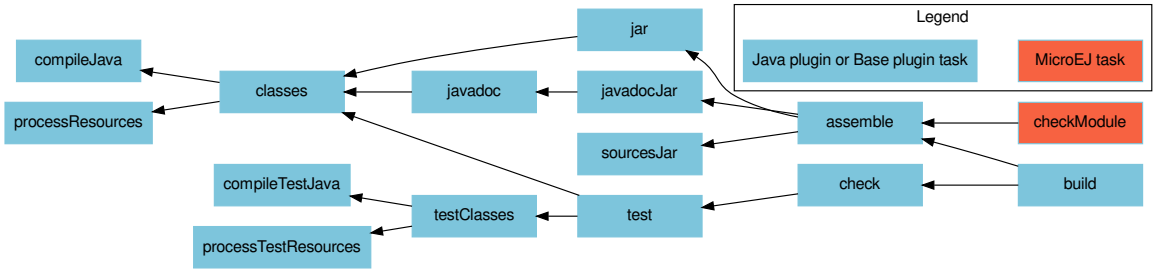
Plugin Name: com.microej.gradle.jse-library

Template: Java SE Library Template

Tasks:

This plugin adds the following tasks to your project:

- tasks of the Gradle Java plugin
- *checkModule*



Configuration:

This module nature inherits from the configuration of all its tasks.

Mock

Plugin Name: com.microej.gradle.mock

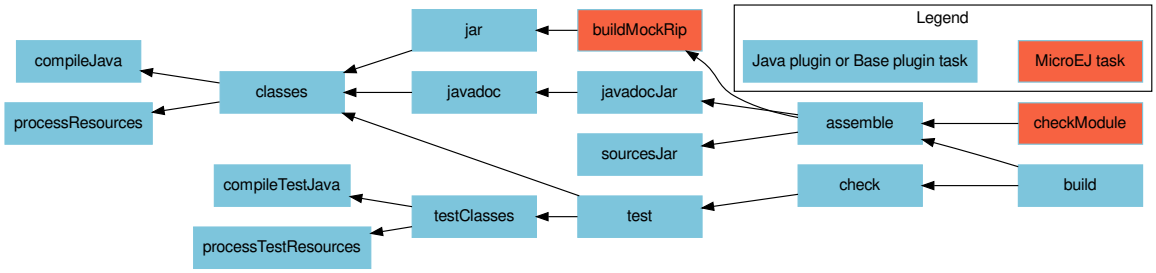
Documentation: *Mock*

Template: Mock Project Template

Tasks:

This plugin adds the following tasks to your project:

- tasks of the Gradle Java plugin
- *checkModule*
- *buildMockRip*



Configuration:

This module nature inherits from the configuration of all its tasks.

Runtime Environment

Plugin Name: com.microej.gradle.runtime-environment

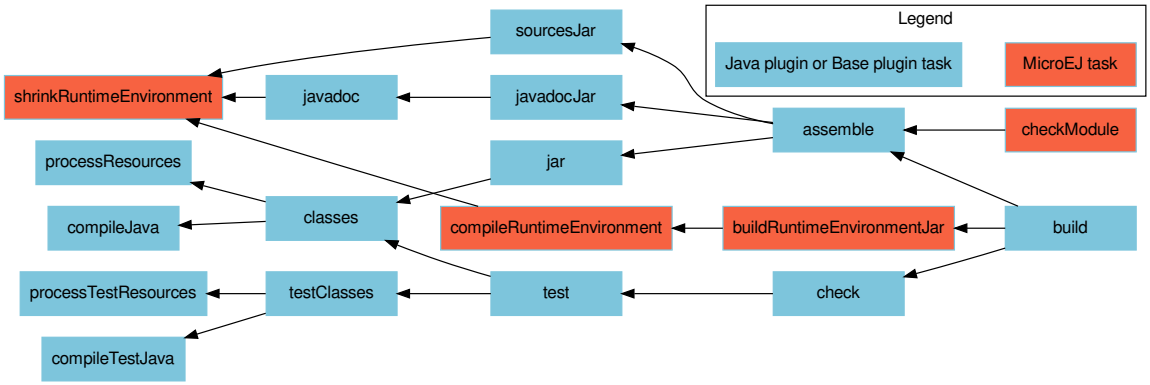
Documentation: *Runtime Environment*

Template: Runtime Environment Template

Tasks:

This plugin adds the following tasks to your project:

- tasks of the Gradle Java plugin
- *checkModule*
- *shrinkRuntimeEnvironment*
- *compileRuntimeEnvironment*
- *buildRuntimeEnvironmentJar*



Configuration:

This module nature inherits from the configuration of all its tasks.

VEE Port

Plugin Name: com.microej.gradle.veeport

Documentation: *VEE Porting Guide*

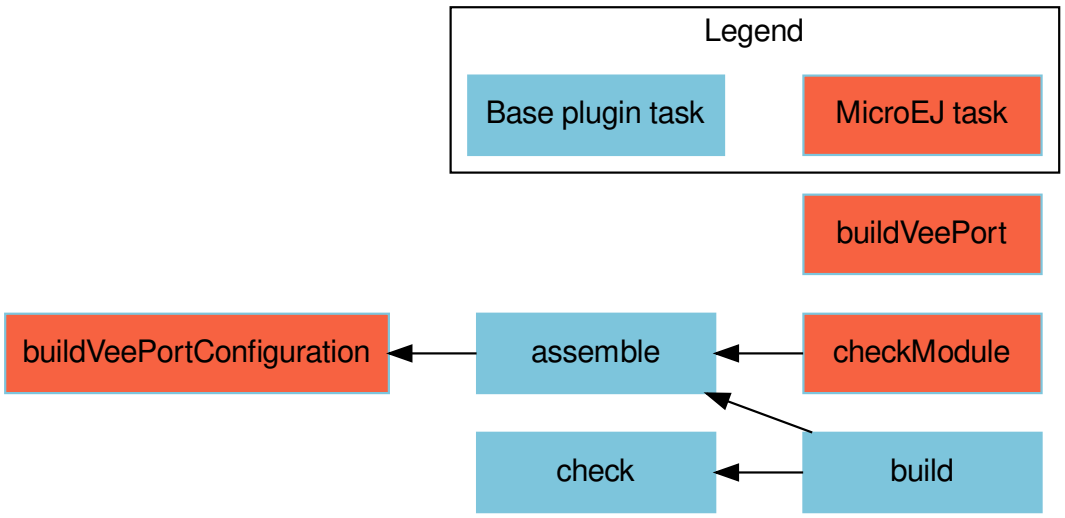
Template: VEE Port Template

Tasks:

This plugin adds the following tasks to your project:

- *checkModule*
- *buildVeePort*

- *buildVeePortConfiguration*
- *extractLibrariesSources*
- *javadoc*
- *javadocJar*



Configuration:

This module nature inherits from the configuration of all its tasks.

Module Repository

Plugin Name: com.microej.gradle.module-repository

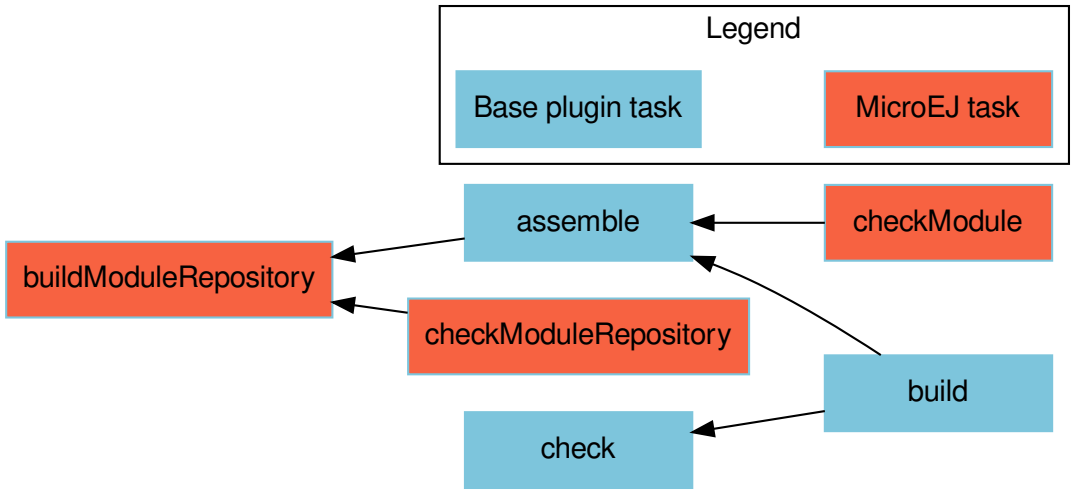
Documentation: *Module Repository*

Template: Module Repository Project Template

Tasks:

This plugin adds the following tasks to your project:

- *checkModule*
- *buildModuleRepository*
- *checkModuleRepository*



Configuration:

This module nature inherits from the configuration of all its tasks.

Tasks

This page describes the module nature tasks as follows:

- **Description**: description and link to the related documentation.
- **Module Natures**: list of *Module Natures* using this task.
- **Configuration**: properties that can be defined to configure the task.

adp

Description: Executes the Addon Processors.

Inputs:

- The project directory

Outputs:

- The directory for each ADP output type (`build/adp/all/main/java` , `build/adp/all/main/resources` , `build/adp/all/test/java` , `build/adp/all/test/resources`)

Module Natures:

This task is used by the following module natures:

- *Add-On Library*
- *Application*

loadVee

Description: Loads the VEE.

Inputs:

- The list of VEE archive files or folders.

Outputs:

- The directory where the VEE is copied/extracted (build/vee)

Module Natures:

This task is used by the following module natures:

- *Add-On Library*
- *Application*

The loadVee task is used internally by the SDK and it is not intended to be executed by the user.

runOnSimulator

Description: Executes the Application with the Simulator.

Inputs:

- The extracted VEE folder
- The project classpath which contains the MicroEJ dependent application classes and resources
- The Full Qualified Name of the Application main class or Feature class
- The folder containing the application configuration (configuration)
- The System properties
- The debug mode
- The debug port

Module Natures:

This task is used by the following module natures:

- *Application*

Configuration:

This task provides the following properties that can be defined in the microej extension:

Name	Description	Default
applicationEntryPoint	Full Qualified Name of the main class or the Feature class of the application. This option is required.	Not set

For example:

```
microej {
  applicationEntryPoint = "com.company.Main"
}
```

checkModule

Description: Checks the compliance of the module.

Inputs:

- The list of the checkers to execute, separated by comas. If not set, all the checkers are executed.
- The list of the checkers to skip, separated by comas.

Module Natures:

This task is used by the following module natures:

- *Add-On Library*
- *Application*
- *Mock*
- *Runtime Environment*
- *VEE Port*
- *Module Repository*

Configuration:

This task is not bound by default on any lifecycle task, which means that it should be called explicitly if it must be executed.

This task provides the following properties that can be defined in the `microej` extension:

Name	Description	Default
`checkers`	Comma-separated list of the names of the checkers to execute. An empty list means that all checkers are executed.	`""`
`skippedCheckers`	Comma-separated list of the names of the checkers to exclude. Only one property of `checkers` and `skippedCheckers` can be defined.	`""`
`checkersRootDir`	Path of the directory to use as root directory for the checkers.	Project directory.

For example:

```
microej {
  checkers = "readme,license"
}
```

buildApplicationObjectFile

Description: Build the object file of the Application.

Inputs:

- The extracted VEE Port folder
- The project classpath which contains the MicroEJ dependent application classes and resources
- The Full Qualified Name of the Application EntryPoint
- The folder containing the application configuration (`configuration`)

Outputs:

- The object file (.o) of the Application and the archive of the build files (build/application/object/bsp/lib/microejapp.o)
- The Zip file containing the generated build files (build/application/applicationObjectBuildFiles.zip)

Module Natures:

This task is used by the following module natures:

- *Application*

The buildApplicationObjectFile task is used internally by the SDK and it is not intended to be executed by the user.

buildExecutable

Description: Builds the Executable of an Application.

Inputs:

- The extracted VEE Port folder
- The folder containing the application configuration (configuration)
- The object file (.o) of the Application

Outputs:

- The directory in which the Executable file and the build files are generated (build/application/executable)

Module Natures:

This task is used by the following module natures:

- *Application*

buildWPK

Description: Builds the WPK of the Application.

Inputs:

- The Application name
- The Application version
- The Full Qualified Name of the Application main class or Feature class
- The Application JAR file
- The Application Javadoc
- The Jar files of the Application classpath
- The folder containing the application configuration (configuration)

Outputs:

- The WPK of the Application (build/libs/<application_name>.wpk)

Module Natures:

This task is used by the following module natures:

- *Application*

buildVirtualDevice

Description: Builds the Virtual Device of an Application.

Inputs:

- The extracted VEE Port folder
- The WPK of the Application
- The project build classpath
- The WPK of the Applications that must be pre-installed in the Virtual Device

Outputs:

- The Zip file of the Virtual Device (build/libs/<application_name>-virtualDevice.zip)

Module Natures:

This task is used by the following module natures:

- *Application*

loadKernelExecutable

Description: Loads the Kernel Executable file.

Inputs:

- The list of Kernel Executable files.

Outputs:

- The loaded Kernel Executable file is copied (build/kernelExecutable/kernel.out)

Module Natures:

This task is used by the following module natures:

- *Application*

The loadKernelExecutable task is used internally by the SDK and it is not intended to be executed by the user.

buildFeature

Description: Builds the Feature file of an Application.

Inputs:

- The Kernel Virtual Device
- The folder containing the Kernel Executable file (build/kernelExecutable)
- The project classpath
- The path of the folder where the Feature file must be generated (build/application/feature)

Outputs:

- The generated Feature file (build/application/feature/application.fo)
- The Zip file containing the generated build files (build/libs/<application_name>-buildFiles.zip)

Module Natures:

This task is used by the following module natures:

- *Application*

runOnDevice

Description: Runs the Executable on a Device.

Inputs:

- The extracted VEE Port folder
- The folder containing the Executable file (build/application/executable)
- The configuration file with all the properties set to launch the build of the Executable (build/properties/target.properties)

Module Natures:

This task is used by the following module natures:

- *Application*

buildMockRip

Description: Builds the Mock RIP.

Inputs:

- The Mock JAR file

Outputs:

- the RIP file of the Mock (build/libs/<project_name>-<project_version>.rip)

Module Natures:

This task is used by the following module natures:

- *Mock*

execTool

Description: Runs the given MicroEJ Tool.

Inputs:

- The extracted VEE Port folder
- The configuration file with all the properties set to launch the application (build/properties/target.properties)
- The folder containing the application configuration (configuration)

Options:

Option	Description
name	Name of the Tool.
toolProperty	Option of the Tool.

Module Natures:

This task is used by the following module natures:

- *Add-On Library*
- *Application*

generateApplicationWrapper

Description: Generates the *Application Wrapper* to be able to run the Application on a VEE Port and a Kernel.

Inputs:

- The Application EntryPoint
- The configuration directory of the project
- The project classpath which contains the MicroEJ dependent application classes and resources

Outputs:

- The directory in which the Wrapper Java class has been generated (build/generated/microej-app-wrapper/java)
- The directory in which the Wrapper Kernel resources have been generated (build/generated/microej-app-wrapper/kernel-resources)
- The directory in which the Wrapper Feature resources have been generated (build/generated/microej-app-wrapper/feature-resources)

Module Natures:

This task is used by the following module natures:

- *Application*

The generateApplicationWrapper task is used internally by the SDK and it is not intended to be executed by the user.

compileWrapperJava

Description: Compiles the *Application Wrapper* to be able to run the Application on a VEE Port and a Kernel.

Inputs:

- The directory containing the Wrapper Java class (build/generated/microej-app-wrapper/java)

Outputs:

- The directory in which the compiled wrapper class is generated (build/generated/microej-app-wrapper/classes)

Module Natures:

This task is used by the following module natures:

- *Application*

The `compileWrapperJava` task is used internally by the SDK and it is not intended to be executed by the user.

shrinkRuntimeEnvironment

Description: Shrinks the Java source files according to the provided *Kernel APIs*.

Inputs:

- Project Kernel API (`src/main/resources/kernel.api`)
- Project Java sources (`src/main/java`)
- The Kernel API files of the Runtime classpath.

Outputs:

- The directory in which shrunk Java sources are generated (`build/runtimeEnvironment/shrunkSources`)

Module Natures:

This task is used by the following module natures:

- *Runtime Environment*
- *Application*

The `shrinkRuntimeEnvironment` task is used internally by the SDK and it is not intended to be executed by the user.

compileRuntimeEnvironment

Description: Compiles the Runtime Environment *Kernel APIs*.

Inputs:

- The directory in which shrunk Java sources are generated (`build/runtimeEnvironment/shrunkSources`)
- The project classpath

Outputs:

- The directory in which shrunk Java classes are generated (`build/runtimeEnvironment/shrunkClasses`)

Module Natures:

This task is used by the following module natures:

- *Runtime Environment*

The `compileRuntimeEnvironment` task is used internally by the SDK and it is not intended to be executed by the user.

buildRuntimeEnvironmentJar

Description: Builds the Runtime Environment Jar file.

Inputs:

- The directory in which shrunk Java classes are generated (build/runtimeEnvironment/shrunkClasses)

Outputs:

- The Jar file of the Runtime Environment (build/libs/<project_name>-<project_version>-runtime-environment.jar)

Module Natures:

This task is used by the following module natures:

- *Runtime Environment*

The buildRuntimeEnvironmentJar task is used internally by the SDK and it is not intended to be executed by the user.

buildFeatureFromWPK

Description: Builds the Feature binary file from a dependent Application.

Inputs:

- The Kernel Virtual Device
- The folder containing the Kernel Executable file (build/kernelExecutable)
- The WPK of the dependent Application
- The path of the folder where the Feature file must be generated (build/application/wpkFeature)

Outputs:

- The generated Feature file (build/application/wpkFeature/application.fo)
- The Zip file containing the generated build files (build/libs/wpkFeature-buildFiles.zip)

Module Natures:

This task is used by the following module natures:

- *Application*

buildVeePort

Description: Builds the VEE Port.

Inputs:

- The project configuration file (configuration.properties)
- The project dropins folder
- The project microui folder (extensions/microui)
- The project classpath which contains the Architecture, Packs, Mocks, Front Panels, Tools and Libraries

Outputs:

- The Zip file of the VEE Port (build/veePort.zip)

Module Natures:

This task is used by the following module natures:

- *VEE Port*

buildVeePortConfiguration

Description: Zips the VEE Port Configuration.

Inputs:

- The project configuration file (configuration.properties)
- The project dropins folder
- The project microui folder (extensions/microui)
- The project classpath which contains the Architecture, Packs, Mocks, Front Panels, Tools and Libraries

Outputs:

- The Zip file of the VEE Port Configuration (build/<project_name>.zip)

Module Natures:

This task is used by the following module natures:

- *VEE Port*

extractLibrariesSources

Description: Extracts all Libraries source Jars to generate the Javadoc of the VEE Port.

Inputs:

- The VEE Port's Libraries exposed to the consumer

Outputs:

- The directory containing the sources of all provided Libraries (build/librariesSources)

Module Natures:

This task is used by the following module natures:

- *VEE Port*

javadoc

Description: Generates VEE Port aggregated javadoc.

Inputs:

- The directory containing the sources of all provided Libraries (build/librariesSources)

Outputs:

- The directory containing the aggregated javadoc (build/docs/javadoc)

Module Natures:

This task is used by the following module natures:

- *VEE Port*

javadocJar

Description: Assembles a jar archive containing the aggregated javadoc.

Inputs:

- The directory containing the aggregated javadoc (build/docs/javadoc)

Outputs:

- The directory containing the Jar file of the aggregated javadoc (build/libs)

Module Natures:

This task is used by the following module natures:

- *VEE Port*

buildModuleRepository

Description: Builds the Module Repository.

Inputs:

- The dependencies of the project.

Outputs:

- The archive file containing the Module Repository (build/libs/<project_name>.zip)

Module Natures:

This task is used by the following module natures:

- *Module Repository*

checkModuleRepository

Description: Checks the consistency of a Module Repository.

Inputs:

- The Module Repository archive file.
- The list of the dependencies to skip. If not set, the check is executed on all dependencies.

Module Natures:

This task is used by the following module natures:

- *Module Repository*

Global Properties

The following properties are available in any module:

Name	Description	Default
microej-ConflictResolution-RulesEnabled	Boolean to enabled or disabled the MicroEJ conflict resolution rules.	true

For example:

```
microej {
  microejConflictResolutionRulesEnabled = false
}
```

3.4.24 Troubleshooting

Java Compiler Version Issue

The SDK requires a JDK 11, so when a JDK 8 is used, the following kind of errors are raised:

- When fetching the MicroEJ Gradle plugin:

```
A problem occurred configuring root project 'myProject'.
> Could not resolve all files for configuration ':classpath'.
> Could not resolve com.microej.gradle.plugins:plugins:0.3.0.
      Required by:
          project : > com.microej.gradle.addon-library:com.microej.gradle.addon-library.gradle.plugin:0.3.
↪0:20221118.151454-1
      > No matching variant of com.microej.gradle.plugins:plugins:0.3.0:20221118.151454-1 was found. The␣
↪consumer was configured to find a runtime of a library compatible with Java 8, packaged as a jar, and its␣
↪dependencies declared externally, as well as attribute 'org.gradle.plugin.api-version' with value '7.4' but:
          - Variant 'apiElements' capability com.microej.gradle.plugins:plugins:0.3.0 declares a library,␣
↪packaged as a jar, and its dependencies declared externally:
              - Incompatible because this component declares an API of a component compatible with␣
↪Java 11 and the consumer needed a runtime of a component compatible with Java 8
              - Other compatible attribute:
                  - Doesn't say anything about org.gradle.plugin.api-version (required '7.4')
          - Variant 'javadocElements' capability com.microej.gradle.plugins:plugins:0.3.0 declares a␣
↪runtime of a component, and its dependencies declared externally:
              - Incompatible because this component declares documentation and the consumer needed␣
↪a library
              - Other compatible attributes:
                  - Doesn't say anything about its target Java version (required compatibility with␣
↪Java 8)
                  - Doesn't say anything about its elements (required them packaged as a jar)
                  - Doesn't say anything about org.gradle.plugin.api-version (required '7.4')
          - Variant 'runtimeElements' capability com.microej.gradle.plugins:plugins:0.3.0 declares a␣
↪runtime of a library, packaged as a jar, and its dependencies declared externally:
              - Incompatible because this component declares a component compatible with Java 11␣
↪and the consumer needed a component compatible with Java 8
```

(continues on next page)

(continued from previous page)

```
                - Other compatible attribute:
                    - Doesn't say anything about org.gradle.plugin.api-version (required '7.4')
            - Variant 'sourcesElements' capability com.microej.gradle.plugins:plugins:0.3.0 declares a␣
↪runtime of a component, and its dependencies declared externally:
                - Incompatible because this component declares documentation and the consumer needed␣
↪a library
                - Other compatible attributes:
                    - Doesn't say anything about its target Java version (required compatibility with␣
↪Java 8)
                    - Doesn't say anything about its elements (required them packaged as a jar)
                    - Doesn't say anything about org.gradle.plugin.api-version (required '7.4')
```

- When using the MicroEJ Gradle plugin:

```
Cause: com/microej/gradle/plugins/MicroejApplicationGradlePlugin has been compiled by a more recent␣
↪version of the Java Runtime (class file version 55.0), this version of the Java Runtime only recognizes␣
↪class file versions up to 52.0
```

The solution is to use a JDK 11 or a higher LTS version (11 , 17 or 21) to fix this error:

- For the command line interface, make sure that a supported JDK version is defined in the PATH environment. To check, run java -version . You should see something like this:

```
$ java -version
openjdk version "11.0.14.1" 2022-02-08
OpenJDK Runtime Environment Temurin-11.0.14.1+1 (build 11.0.14.1+1)
OpenJDK 64-Bit Server VM Temurin-11.0.14.1+1 (build 11.0.14.1+1, mixed mode)
```

  Alternatively, you can set the JAVA_HOME environment variable to point to the installation directory of the JDK.

- For IntelliJ IDEA and Android Studio, go to File > Settings... > Build, Execution, Deployment > Build Tools > Gradle , and make sure the selected Gradle JVM is a supported JDK version:

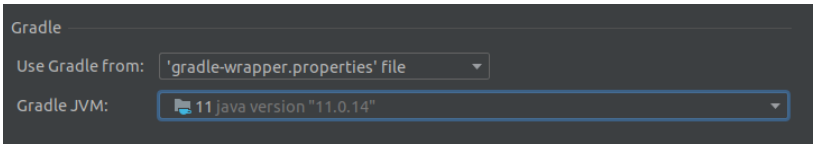


Fig. 91: Project JDK in IntelliJ IDEA and Android Studio

Unresolved Dependency When No Repository

If this kind of message appears when resolving plugins or modules dependencies:

```
* What went wrong:
Plugin [id: 'com.microej.gradle.application', version: '1.4.0'] was not found in any of the following sources:

- Gradle Core Plugins (plugin is not in 'org.gradle' namespace)
- Included Builds (No included builds contain this plugin)
- Plugin Repositories (could not resolve plugin artifact 'com.microej.gradle.application:com.microej.gradle.
```

(continues on next page)

(continued from previous page)

```
↪application.gradle.plugin:1.1.0')
Searched in the following repositories:
        Gradle Central Plugin Repository
```

or this kind:

```
* What went wrong:
Execution failed for task ':compileJava'.
> Could not resolve all files for configuration ':compileClasspath'.
> Cannot resolve external dependency ej.api:edc:1.3.7 because no repositories are defined.
        Required by:
                root project :
```

It means that no module or plugin repository has been defined. Make sure that the repositories have been configured as described in the *Configure Repositories* section. In particular, make sure that the Gradle Init script is in the `.gradle/init.d` folder in the Gradle User Home folder. The Gradle User Home folder is defined by default to the OS User Home folder, and can be changed by setting the `GRADLE_USER_HOME` environment variable or in the IDE settings (see Intellij IDEA documentation).

Unresolved Dependency in Repositories

If this kind of message appears when resolving plugins or modules dependencies:

```
* What went wrong:
Plugin [id: 'com.microej.gradle.application', version: '1.4.0'] was not found in any of the following sources:

- Gradle Core Plugins (plugin is not in 'org.gradle' namespace)
- Included Builds (No included builds contain this plugin)
- Plugin Repositories (could not resolve plugin artifact 'com.microej.gradle.application:com.microej.gradle.
↪application.gradle.plugin:1.4.0')
Searched in the following repositories:
        microEJCentral(https://repository.microej.com/modules)
        microEJForgeCentral(https://forge.microej.com/artifactory/microej-central-repository-release)
        microEJForgeDeveloper(https://forge.microej.com/artifactory/microej-developer-repository-release)
```

or this kind:

```
* What went wrong:
Execution failed for task ':compileJava'.
> Could not resolve all files for configuration ':compileClasspath'.
  > Could not find com.mycompany:mymodule:M.m.p.
    Searched in the following locations:
      - https://my-company-first-repository/com/mycompany/mymodule/M.m.p/kf-M.m.p.pom
      - https://my-company-first-repository/com/mycompany/mymodule/M.m.p/ivy-M.m.p.xml
      - https://my-company-second-repository/com/mycompany/mymodule/M.m.p/kf-M.m.p.pom
      - https://my-company-second-repository/com/mycompany/mymodule/M.m.p/ivy-M.m.p.xml
    Required by:
        project :
```

First, check that either the requested plugin or module exists in the listed repositories.

- If the plugin or module does not exist,
  - if it is declared as a *direct dependency*, the module repository is not compatible with your source code. You can either check if another module version is available in the repository or add the missing module to the repository.

  - otherwise, this is likely a missing transitive module dependency. The module repository is not consistent. Check the module repository and make sure all the transitive dependencies exist.

- If the module exists, this may be due to a missing repository in the configuration. Check that your repository appears in the list of URLs below the error line:

```
Searched in the following locations:
```

  If the URL of your repository is not listed, add it to *the list of the repositories*.

- If the repository is correctly configured, this may be a network connection error. We can check in the debug logs, by adding the `--debug` arguments in the Gradle command line.

Otherwise, if your module repository is an URL, check for an *Invalid SSL Certificate* issue.

Invalid SSL Certificate

If a dependency cannot be retrieved from a remote repository, this may be due to a missing or incorrect SSL certificate. It can be checked in the debug logs, by adding the `--debug` argument in the Gradle command line, for example:

```
./gradlew build --debug
```

If the logs show the SSL handshake start followed immediately by the Connection Shutdown:

```
2026-01-27T18:01:16.708+0100 [DEBUG] [org.apache.http.conn.ssl.SSLConnectionSocketFactory]␣
↪Starting handshake
2026-01-27T18:01:16.749+0100 [DEBUG] [org.apache.http.impl.conn.
↪DefaultManagedHttpClientConnection] http-outgoing-69: Shutdown connection
2026-01-27T18:01:16.749+0100 [DEBUG] [org.apache.http.impl.execchain.MainClientExec]␣
↪Connection discarded
```

it most likely means that the SSL certificate is missing or incorrect.

This can be raised in several cases, such as:

- an artifact repository configured in the MicroEJ Module Manager settings using a self-signed SSL certificate or a SSL certificate not trusted by the JDK.
- the requests to an artifact repository configured in the MicroEJ Module Manager settings are redirected to a proxy server using a SSL certificate not trusted by the JDK.

In all cases, the SSL certificate (used by the artifact repository server or the proxy) must be added to the JDK trust store that is running Gradle.

Before updating the SSL certificate, it is recommended to exit all your IDE projects and stop Gradle daemons (all versions). One simple way is to list all Java processes and kill the ones named `GradleDaemon`:

```
./jps
12768
17792 GradleDaemon
16920
4712 Jps
1820 GradleDaemon
```

Also, if you don't know which JDK is running Gradle, you can first fix the JDK used by Gradle by following *How To Define a Specific Java Home for Gradle*.

Ask your System Administrator, or retrieve the SSL certificate and add it to the JDK trust store:

- on Windows
  1. Install Keystore Explorer.
  2. Start Keystore Explorer, and open file [JRE_HOME]/lib/security/cacerts or [JDK_HOME]/jre/lib/security/cacerts with the password changeit . You may not have the right to modify this file. Edit rights if needed before opening it or open Keystore Explorer with admin rights.
  3. Click on Tools , then Import Trusted Certificate .
  4. Select your certificate.
  5. Save the cacerts file.
- on Linux/macOS
  1. Open a terminal.
  2. Make sure the JDK's bin folder is in the PATH environment variable.
  3. Execute the following command:

```
keytool -importcert -v -noprompt -trustcacerts -alias myAlias -file /path/to/the/certificate.pem -
↪keystore /path/to/the/truststore -storepass changeit
```

If the problem still occurs, there should be a trace which indicates the beggining of the handshake phase of the SSL negotiation:

```
2023-12-15T17:32:47.442+0100 [DEBUG] [org.apache.http.conn.ssl.SSLConnectionSocketFactory] Starting␣
↪handshake
```

The error very probably occurs during this phase. There should be the following trace before the error:

```
Consuming server Certificate handshake message
```

The traces below this one indicates the SSL certificate (or the SSL certificates chain) presented by the server. This certificate or one of the root or intermediate certificates must be added in the JDK truststore as explained previously.

Failing Resolution in adp Task

During the build of a project, the error Cannot locate module version for non-maven layout may be raised:

```
Execution failed for task ':adp'.
> Could not resolve all files for configuration ':addonProcessorClasspath'.
      > Could not download binary-nls-processor-2.4.2.adp (com.microej.tool.addon.runtime:binary-nls-processor:2.
↪4.2)
            > Cannot locate module version for non-maven layout.
      > Could not download js-processor-0.13.0.adp (com.microej.tool.addon.runtime:js-processor:0.13.0)
            > Cannot locate module version for non-maven layout.
      > Could not download junit-processor-1.7.1.adp (ej.tool.addon.test:junit-processor:1.7.1)
            > Cannot locate module version for non-maven layout.
```

This is due to a wrong pattern in the declaration of the Ivy repositories. Check your Ivy repositories and make sure the value of the artifact of the patternLayout block is set to [organisation]/[module]/[revision]/[artifact]-[revision](-[classifier])(.[ext]) . For example:

```
ivy {
	url = uri("https://repository.microej.com/5/artifacts/")
	patternLayout {
		artifact("[organisation]/[module]/[revision]/[artifact]-[revision](-[classifier])(.[ext])")
		ivy("[organisation]/[module]/[revision]/ivy-[revision].xml")
		setM2compatible(true)
	}
}
```

Missing Version for Publication

If the following message is displayed when publishing a module:

```
The project version must be defined.
```

It means the `version` property is missing and should be defined in the module build file. See *Publish a Project* for more information.

Fail to load a VEE Port as dependency

When a VEE Port is defined as a dependency, the build of the project can fail with the following message:

```
    > No 'release.properties' and 'architecture.properties' files found.
The given file <path/to/file> is not a VEE Port archive.
```

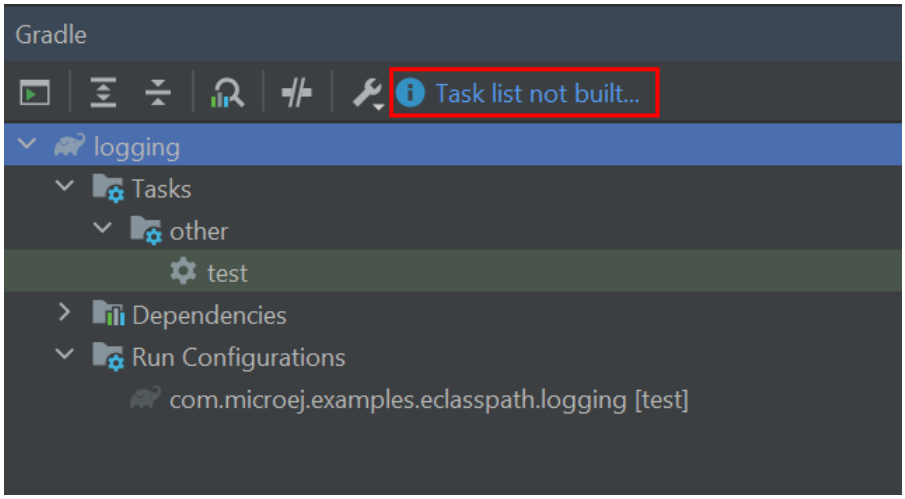
If the dependency is a valid VEE Port, this error probably means that several artifacts of the VEE Port have been published with the `default` Ivy configuration. To fix this issue, you can select the right artifact by adding information on the one to fetch in the `artifact` block, for example:

```
microejVee("com.mycompany:myveeport:1.0.0") {
	artifact {
		name = "artifact-name"
		type = "zip"
	}
}
```

This will select the artifact with the name `artifact-name` and with the type `zip`.

Missing Tasks in the Gradle view of Android Studio

In some cases, Android Studio may not build all the Gradle tasks, the `Task list not built...` message is displayed:

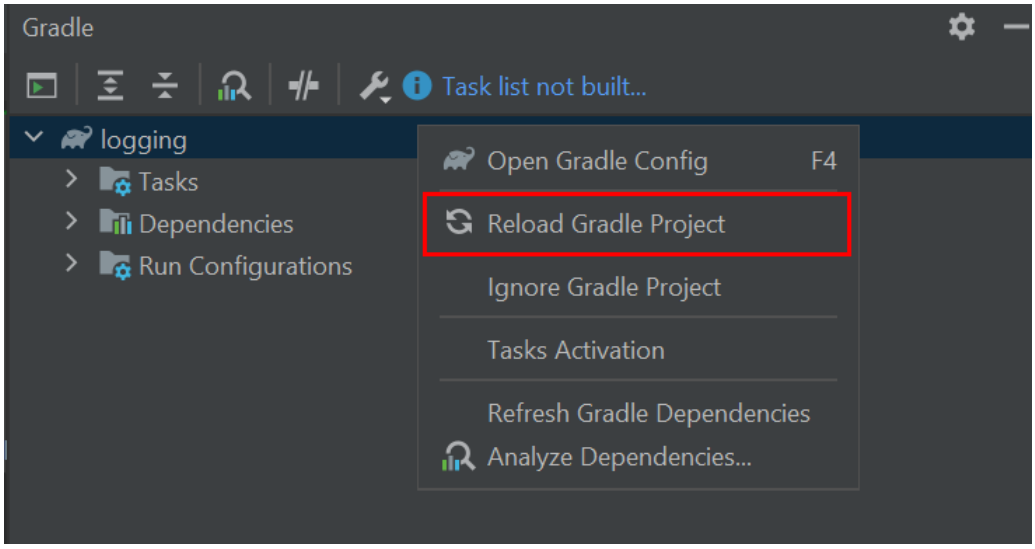


To build all the Gradle tasks in Android Studio:

- Go to `File` > `Settings` > `Experimental` ,
- Enable the option: `Configure all Gradle tasks during Gradle Sync (…)` .

Back in the Gradle task view:

- Right-click on the project name,
- Select `Reload Gradle Project` .



Note: By default, all supported IDEs require the user to explicitly trigger the reload of a Gradle project when its configuration has changed. However you can configure your IDE to automatically reload your project. Refer to the *How To Automatically reload a Gradle project* section for more information.

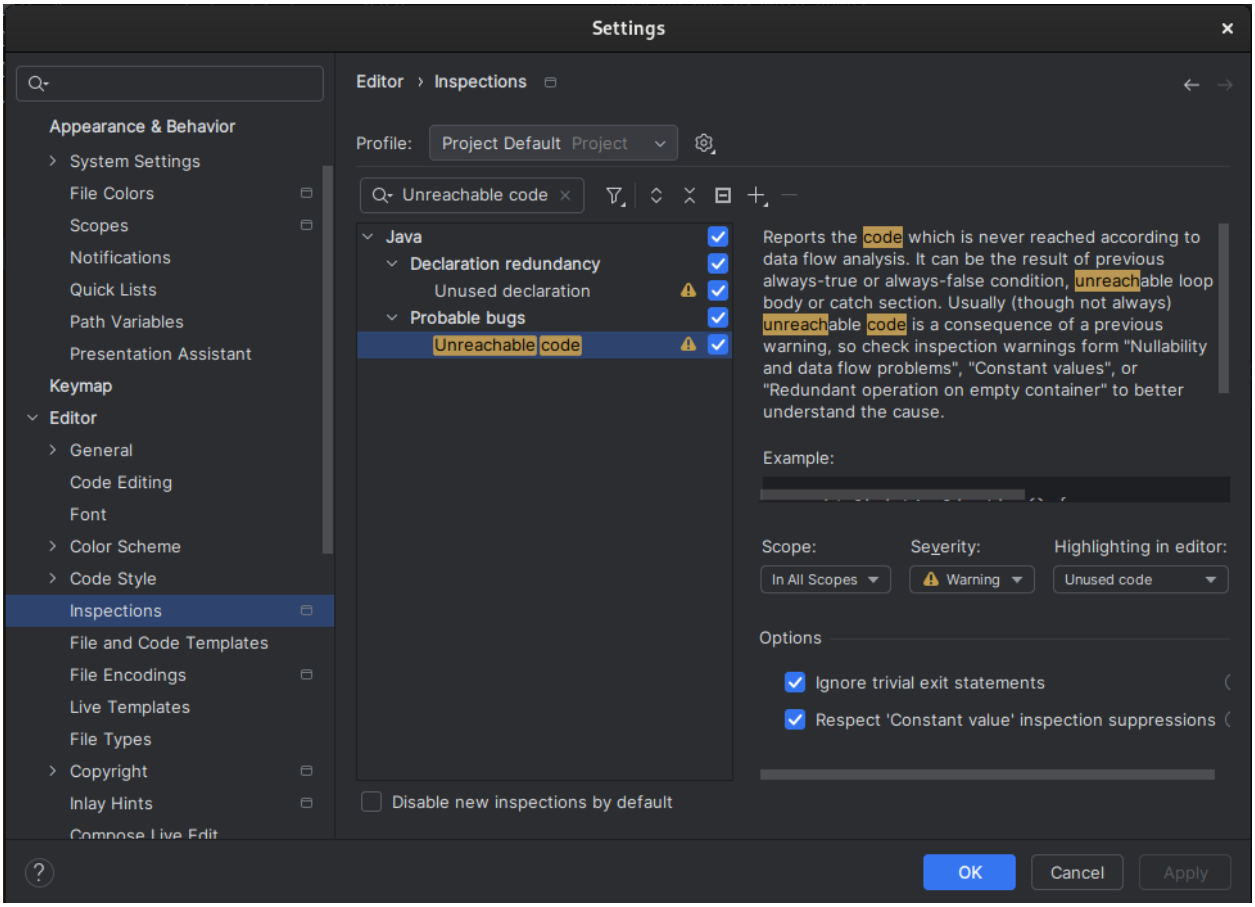
Code Detected as Unreachable in IntelliJ

When opening MicroEJ project with IntelliJ 2024.1 , code is displayed in grey (dead code style) after a call to a MicroEJ Foundation API.

```
public class Main {

    public static void main(String[] args) {
        String value = Constants.getString( name: "my.constant");
        System.out.println("Hello " + value); // detected as unreachable
    }
}
```

This happens because Foundation API dependencies do not include implementation code but only throw RuntimeException . IntelliJ thus infers that the code that comes after is unreachable.

The detection of unreachable code can be disabled by going in Settings... > Editor > Inspections and unchecking the option Unreachable code in Java > Probable bugs .



You can also disable unreachable code detection locally by using @SuppressWarnings("UnreachableCode") on the method or on the class.

Resolution Conflict with Testsuite Dependencies

When a project and its testsuite depend on different versions of the same dependency, the build of the project can fail with the following message:

```
Execution failed for task ':adp'.
> Conflict(s) with a direct dependency for the following module(s):
- ej.api:kf : the resolved version is 1.7.0 whereas the direct dependency version is 1.4.4
```

To fix this issue, you must update the dependency of your testsuite to use the same version as the project dependency:

```
dependencies {
	implementation("ej.api:edc:1.3.7")
	implementation("ej.api:kf:1.7.0")
}

testing {
	suites {
		val test by getting(JvmTestSuite::class) {
			microej.useMicroejTestEngine(this)

			dependencies {
				implementation(project())
				implementation("ej.api:edc:1.3.7")
				implementation("ej.api:kf:1.7.0")
				implementation("ej.library.test:junit:1.12.0")
				implementation("org.junit.platform:junit-platform-launcher:1.8.2")
			}
		}
	}
}
```

Gradle Build Files (**.kts*) Errors in IntelliJ IDEA

Since version 2025.1 , the new Kotlin K2 compiler is enabled by default to compile Gradle build scripts. For now, it requires a JDK in project's classpath in order to interpret the build file successfully. MicroEJ projects do not have JDK in classpath, the Gradle build scripts are thus fully marked in red with the following errors:

```
Cannot access 'Comparable' which is a supertype of 'KotlinBuildScript'. Check your module classpath for missing␣
↪or conflicting dependencies.
Cannot access 'Object' which is a supertype of 'KotlinBuildScript'. Check your module classpath for missing or␣
↪conflicting dependencies.
```

To fix this, you need to

- uncheck Enable K2 in Settings… > Languages & Frameworks > Kotlin

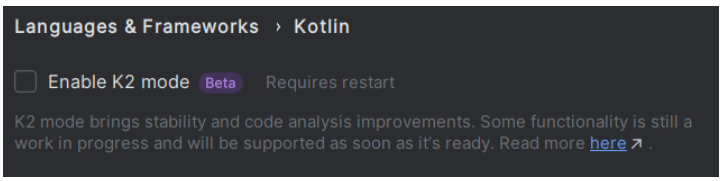


Fig. 92: Disable K2 in IntelliJ IDEA

– Invalidate caches from Files > Invalidate Caches… and check all the checkboxes as shown below, and click on Invalidate and Restart

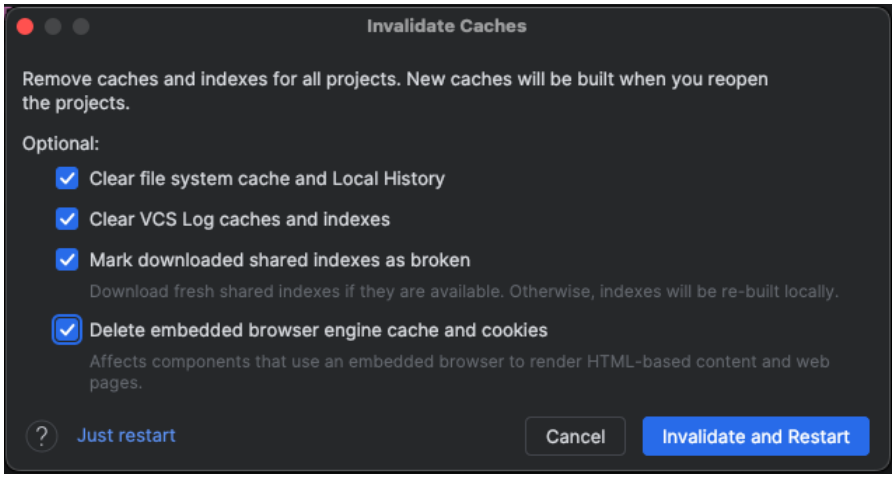


Fig. 93: Invalidate all the caches

Fail to fetch the prod version of an Architecture

Fetching the prod version of an Architecture may raise the following error:

```
> Could not find flopi4G25-8.3.0-prod.xpf (com.microej.architecture.CM4.CM4hardfp_GCC48:flopi4G25:8.4.0).
  Searched in the following locations:
      https://repository.microej.com/modules/com/microej/architecture/CM4/CM4hardfp_GCC48/flopi4G25/8.4.
↪0/flopi4G25-8.4.0-prod.xpf
```

This happens when a module repository containing only the eval version of the Architecture is declared before the module repository containing the prod version. This is typically the case with the MicroEJ Central Repository which contains only the eval versions of the public Architecture. If the prod version is deployed in another repository declared after the MicroEJ Central Repository, the above error is raised.

To fix this, declare the repository containing the prod version before the one containing the eval version. Refer to *How To Add a Repository* for more details on how to declare module repositories.

Missing Compilation Capability

During the build of a project (precisely the Java compilation phase), the following error may be raised:

```
* What went wrong:
Execution failed for task ':compileJava'.
> Error while evaluating property 'javaCompiler' of task ':compileJava'.
      > Failed to calculate the value of task ':compileJava' property 'javaCompiler'.
            > Toolchain installation 'C:\Program Files (x86)\Eclipse Adoptium\jre-11.0.28.6-hotspot' does not␣
↪provide the required capabilities: [JAVA_COMPILER]
```

This means that the Java installation used by Gradle is not a JDK but a JRE, whereas Gradle and the SDK require a JDK. Therefore, the solution is to install and configure a JDK. Refer to *Check your JDK version* for more information.

Build Errors from JVM Memory Misconfiguration

Below are two examples of error messages that can be raised when the JVM used to launch the VEE scripts is not configured properly. The ouput may vary depending on the Gradle task being executed but the errors share the same root cause.

When *building an executable*:

```
================ [ Launching SOAR ] ================
1 : ERROR :
[M2] - OutOfMemory error. Try to increase heap space using JVM option '-Xmx' (e.g. '-Xmx4096M')
Terminated with errors
```

When *running on simulator*:

```
================ [ Launching on Simulator ] ================
"Internal limits reached. Please contact support@microej.com"
```

These errors are typically seen when running `gradlew buildExecutable` or `gradlew runOnSimulator` if the forked JVM is started with insufficient heap.

To resolve this, adjust the JVM memory configuration by setting the JVM options accordingly with the property `microej.launch.jvmargs` , for example:

```
./gradlew buildExecutable -Dmicroej.launch.jvmargs="-Xmx1024m -Xms512m"
```

No Architecture Defined Error

The following error may be raised when building a project:

```
Execution failed for task ':loadVee'.
> No Architecture defined. An Architecture can be provided by declaring adependency with the␣
↪`microejArchitecture` configuration (e.g. `microejArchitecture("com.mycompany:myArchitecture:M.m.p")').
```

This can happen when a VEE Port built with SDK 6 `1.3.0` or higher is provided to a project using an older version of the SDK 6. To resolve this, update your project to use SDK 6 version `1.2.0` minimum.

Build Errors from JDK / Gradle Incompatibilities in VS Code

Gradle must be run with a compatible JDK version. If the Gradle/JDK compatibility matrix is not respected, project sync or builds may fail, and in VS Code the Java/Gradle integration often reports non-explicit errors, making the root cause difficult to identify.

If you experience the following errors, first verify Gradle/JDK compatibility and either update the Gradle Wrapper or use a supported JDK before further troubleshooting.

JDK 25 with Gradle < 9.1.0

```
* What went wrong:
25.0.1
```

JDK 24 and Gradle < 8.14.0

```
* What went wrong:
Could not create task ':outgoingVariants'.
> Could not create task of type 'OutgoingVariantsReportTask'.
      > Could not create an instance of type org.gradle.api.tasks.diagnostics.internal.configurations.
↪ConfigurationReportsImpl.
            > Could not create an instance of type org.gradle.api.reporting.internal.DefaultReportContainer.
                  > Type T not present
```

JDK 23 and Gradle < 8.10

```
* What went wrong:
Could not open cp_init generic class cache for initialization script
BUG! exception in phase 'semantic analysis' in source unit '_BuildScript_' Unsupported class file major version 67
> Unsupported class file major version 67
```

JDK 22 and Gradle < 8.7

```
* What went wrong:
BUG! exception in phase 'semantic analysis' in source unit '_BuildScript_' Unsupported class file major version 66
> Unsupported class file major version 66
```

JDK 11 and Gradle 9+

```
* What went wrong:
[error] [gradle-server] Could not run build action using connection to Gradle distribution 'https://services.gradle.
↪org/distributions/gradle-9.x.x-bin.zip'.
```

Gradle Build Performance

The following settings can improve Gradle build times. They can be configured globally in `<USER_HOME>/.gradle/gradle.properties`, per-project in `<PROJECT_ROOT>/gradle.properties`, or passed via command line with `-D` (system property).

JVM Heap Size

Increases memory allocated to the Gradle Daemon, reducing garbage collection overhead for large projects:

```
org.gradle.jvmargs=-Xmx2048m
```

File System Watching

Gradle tracks file system changes to optimize incremental builds. This feature can significantly slow down builds in certain environments (e.g., unstable network drives). Disable it by passing `--no-watch-fs` in the command line or by setting:

```
org.gradle.vfs.watch=false
```

Refer to the Gradle documentation for more configuration options.

Gradle Build Performance: Windows-Specific Optimizations

Gradle builds can be noticeably slower on Windows 10+ due to real-time antivirus scanning, especially Windows Defender. Gradle and the JVM create, read, and delete thousands of small files during dependency resolution, compilation, packaging, and caching. When Windows Defender scans each of these file operations, it adds significant I/O overhead, which directly increases build times, in particular for large projects.

Step 1: Exclude key folders from Windows Defender

To reduce unnecessary scanning and improve Gradle build performance, configure Windows Defender to exclude the following folders (adapt paths to your environment):

- `C:\Users\<name>\.gradle` : Gradle dependency cache, transforms, metadata.
- `C:\Users\<name>\.microej` : MicroEJ-specifics (e.g., evaluation licences).
- `C:\path\to\project` : project files, or at least all `<project>/build` output directories.
- `C:\Program Files\Eclipse Adoptium\jdk-11.0.18.10-hotspot` : JDK installations used by Gradle/IDE
- JVM temporary directory (default: `%TEMP%` , typically `C:\Users\<name>\AppData\Local\Temp`). Run `java -XshowSettings:properties 2>&1 | findstr java.io.tmpdir` to display the current value.
- `C:\Users\<name>\AppData\Local\JetBrains\IntelliJIdea2025.3` : IDE caches (example: IntelliJ IDEA)

Note: Some IDEs like IntelliJ IDEA automatically display a notification offering to configure exclusions for project directories and IDE caches. It is recommended to accept this suggestion when prompted.

Windows Defender exclusions can be configured either through the graphical interface or via the command line.

In the Windows Security app, go to Virus & threat protection > Manage settings > Exclusions > Add or remove exclusions , and add the relevant folders as exclusion paths.

Exclusions can also be added from an elevated PowerShell prompt using the following command:

```
Add-MpPreference -ExclusionPath "C:\path\to\exclude"
```

You can repeat this command for each directory to exclude, and verify the configuration with:

```
Get-MpPreference | Select -ExpandProperty ExclusionPath
```

Step 2: Disable NTFS features that slow down builds (optional)

Windows NTFS enables legacy features that add overhead for file-heavy workloads like Gradle builds.

To prevent NTFS from generating legacy DOS-style filenames for every new file, disable 8.3 filename generation in an elevated PowerShell:

```
fsutil.exe behavior set disable8dot3 1
```

To prevent NTFS from updating file “last accessed” timestamps on every read, which reduces extra disk writes:

```
fsutil.exe behavior set disableLastAccess 3
```

Warning: These changes may affect other software on the system. Disabling 8.3 filenames can break legacy applications that depend on short DOS-style paths. Disabling last access timestamps may impact backup software using incremental backups based on access time, or file synchronization tools. Refer to the Windows documentation for more details.

3.4.25 Tutorials

Branding an Eclipse IDE

Eclipse IDE allows to create custom versions of its distribution. This can be very convenient if you need to redistribute your own unique Eclise IDE version, customized to your brand.

This tutorial will guide you through the steps to create such a branded Eclipse IDE.

Install Eclipse and the MicroEJ Plugin

- Download the Eclipse IDE for RCP and RAP Developers Edition.
- Install and launch it.
- Install the MicroEJ plugin as described in the Eclipse tab of the *Install the IDE Plugins* chapter.

Create the Project

Once everything is installed, the first step is to create the project:

- Click on File > New > Plug-in Project .
- Fill the Project name field, for example my-branded-eclipse .
- Click on the Next button.
- Change the version, the name and the vendor if necessary.

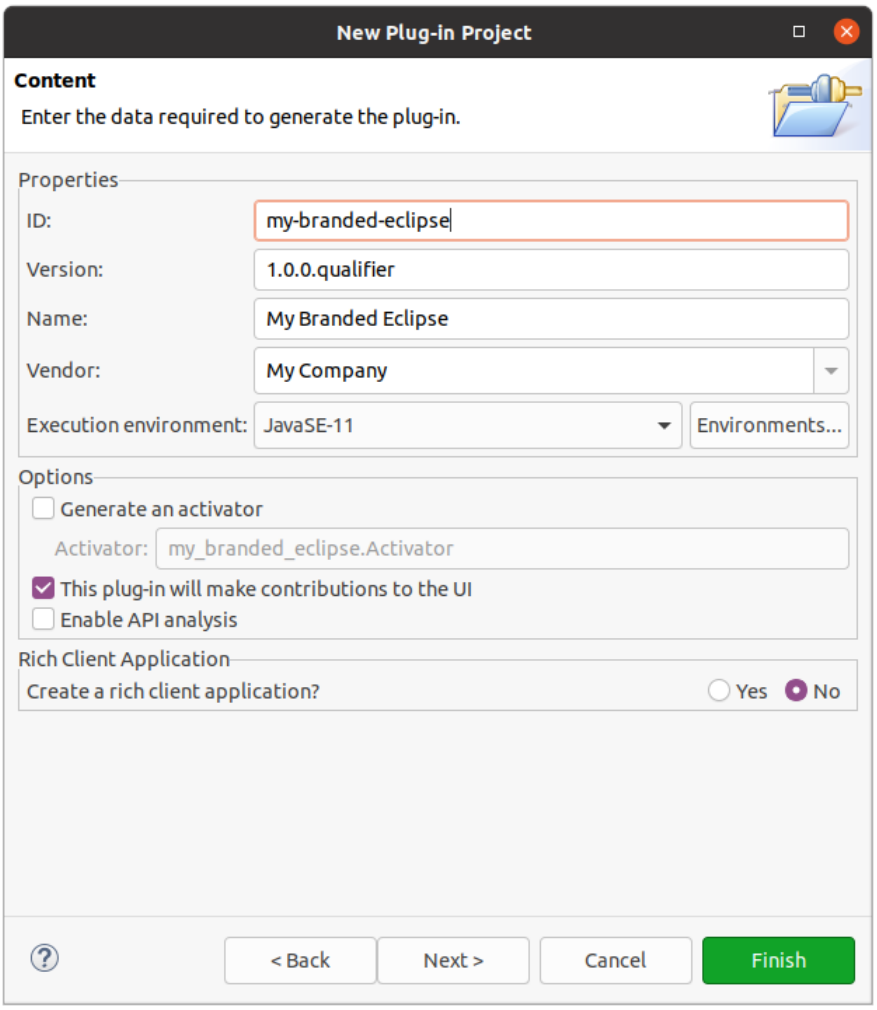


Fig. 94: Creation of a Branding Eclipse Project

- Click on the Finish button.
- Create a folder named images at the root of the project. It will contain the branding resources described later in this tutorial.

Configure the Product

- Right-click on the project, then click on New > Other... .
- Select Plug-in Development > Product Configuration .
- Click on the Next button.
- Set a file name, for example eclipse .
- Click on the Finish button. The Product configuration file should be created and opened now.
- In the Overview tab
  - In the Product field, click on New... .

  - Set a value in the Product Name field, for example My Branded Eclipse.
  - Select the project in the Defining Plug-in field.
  - Set a value in the Product ID field, for example product.
  - In the Application field, select org.eclipse.ui.ide.workbench.
  - Click on the Finish button.
- In the Contents tab
  - For each of the following terms, click on Add… , type the term in the field, then select all the items in the list and click on the Add button:

```
jdt
microej
egit
buildship
mpc
mylyn
org.eclipse.ui.ide.application
<plugin-name> (``my-branded-eclipse`` for the example values used previously)
```

  - Click on Add Required Plug-ins.
- In the Configuration tab
  - Click on the Add Recommended… button.
  - Click on the OK button.
- In the Launching tab
  - If you want to change the default name of the Eclipse launch executable (defaults to eclipse), set the Launcher Name field with the new name.
  - If you want to change the icon files of the Eclipse launch executable file,
    * Copy the image file(s) of the IDE launcher in the images folder. The image format depends on the Operating System:
      · icon.xpm for Linux
      · icon.icns for macOS
      · icon.ico file or icon.bmp files for Windows.

      For Windows, if bmp files are used, it is required to provide one bmp file for each one of the following resolutions: 16x16 (8-bit), 16x16 (32-bit), 32x32 (8-bit), 32x32 (32-bit), 48x48 (8-bit), 48x48 (32-bit), 256x256 (32-bit).
    * Select the icon files for the targeted Operating Systems. Make sure the paths are the relative paths from the project root folder.
- In the Splash tab
  - If you want to change the default splash screen displayed at startup,
    * Copy the image file of the splash screen at the root of the project. The following name and image types are supported:

- splash.png
- splash.jpg
- splash.jpeg
- splash.gif
- splash.bmp

The recommended size for the splash screen is 455x295.

* In the Plug-in field, click on the Browse… button.
* Select the plugin of the project (my-branded-eclipse in our example).

– If you want to change the splash screen behavior, adapt the other options in the Customization section to your need. For example you may want to add a progress bar in the splash screen by checking the option Add a progress bar .

- In the Branding tab (make sure the image paths are the relative paths from the project root folder)
  – If you want to change the default application window icon (visible in the Windows dock for example),
    * Copy the image files associated with the application window in the images folder. The image format must be png , with one png file for each one of the following resolutions: 16x16, 32x32, 48x48, 64x64, 128x128, 256x256.
    * For each field in the Window Images section, select the corresponding image.
  – If you want to content of the About dialog (visible in Help > About…),
    * Copy the About dialog image in the images folder. This image must be in png format and should not exceed 500x330.
    * In the About Dialog section, select the image and fill the text. The text is not shown if the image exceeds 250x330.

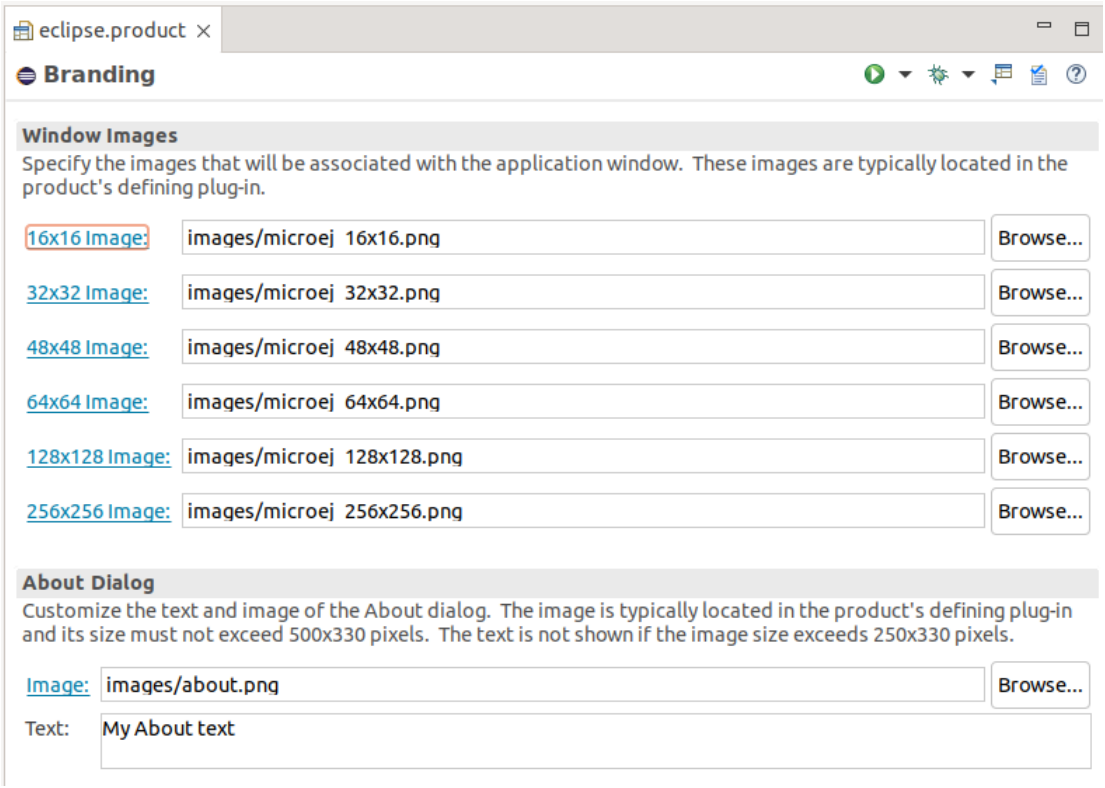


Fig. 95: Branding Tab of a Branding Eclipse Project

- Save the Product file.
- Go back to the `Overview` tab and click on `Synchronize` in the `Testing` section.

Your project should look like this at this stage:

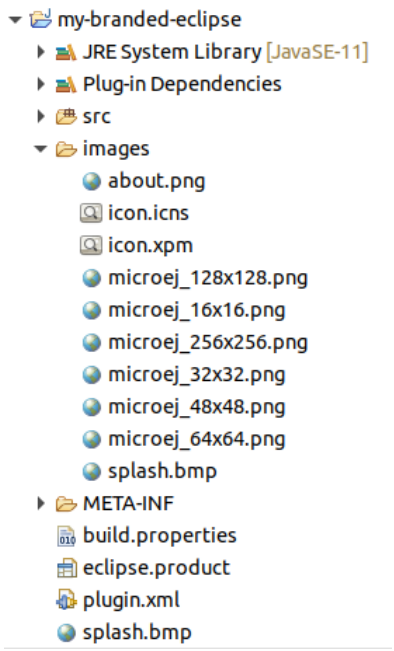


Fig. 96: Structure of a Branding Eclipse Project

Advanced Options

Eclipse provides several other options to customize an Eclipse Product that can be defined in the plugin_customization.ini file located at the root of the project. Create this file if it does not exist in your project.

Then you can define any option, for example to set the default perspective to the Java perspective:

```
org.eclipse.ui/defaultPerspectiveId=org.eclipse.jdt.ui.JavaPerspective
```

Here is a list of interesting options:

Name	Description	Default
org.eclipse.ui/SHOW_PROGRESS_ON_STARTUP	Show progress in the splash screen.	false
org.eclipse.ui/default-PerspectiveId	Perspective that the workbench opens initially.	org.eclipse.ui.resourcePerspective

Export the Product

The final step is to export the project as an Eclipse Product:

- Open the build.properties file, and make sure to select the Build tab.
- In the Binary Build section, select:
  - META-INF folder
  - plugin.xml file
  - splash.bmp file
  - images folder
  - plugin_customization.ini file (if exists)
- Save your changes in the build.properties file.
- Right-click on the project, then click on Export… .
- Select Plug-in Developement > Eclipse product , then click on the Next button.
- In the Configuration field, select the .product file.
- In the Synchronization section, make sure the Synchronize before exporting option is checked.
- In the Directory field of the Destination section, select the destination folder.
- Click on the Finish button.

Once the process is done, you should find the new branded Eclipse IDE in the destination folder.

Creating and Using an Offline Repository

Developing MicroEJ projects requires the Gradle plugins used for the build, as well as the modules (Add-On Libraries, Foundation Libraries, …) used by the project code. All these artifacts must be available in artifact repositories.

MicroEJ provides them as online repositories which can be used directly, thanks to the configuration described in the *Configure Repositories* section. However, it is not always possible to rely on these online repositories. Gradle allows to use repositories packaged as a set of local folders and files, called Offline Module Repositories.

This tutorial explains how to create and use Offline Module Repositories for your MicroEJ project.

Offline Repository for the Gradle Plugins

The first step is to create an Offline Repository for the Gradle plugins. The artifacts of the Gradle plugins are available in the SDK 6 Forge repository.

- Go to the SDK 6 repository.
- Click on the Download button:

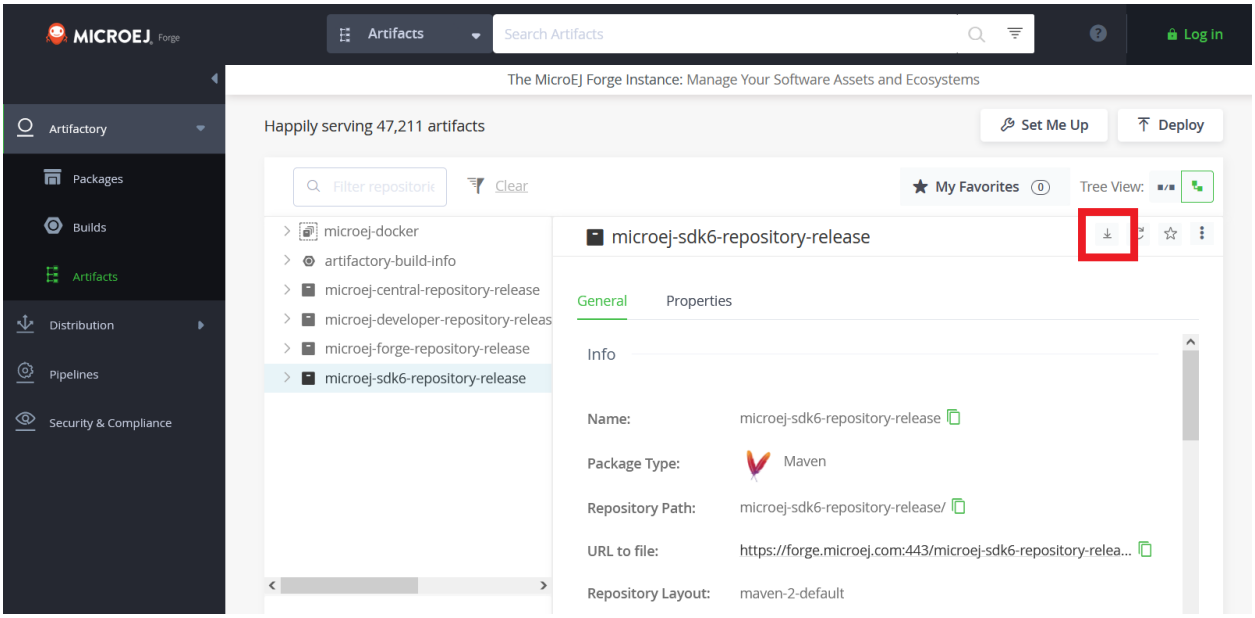


Fig. 97: Download SDK 6 Gradle Plugins Repository

- In the upcoming popup, check the Include Checksum Files checkbox.
- Click on Download .

Now that the Offline Repository of the Gradle plugins has been retrieved, you can configure your projects to use it:

- Unzip the downloaded archive at the location of your choice, for example in the C:\sdk6-repository folder.
- Add the following repository definition at the beginning of *your repositories configuration script*:

```
fun RepositoryHandler.offlineMicroEjSdk() {
  val sdk6Uri = uri("C:\\sdk6-repository")

  /* Offline SDK 6 repository for Maven/Gradle modules */
  maven {
    name = "offlineSDKRepositoryMaven"
    url = sdk6Uri
  }

  /* Offline SDK 6 repository for Ivy modules */
  ivy {
    name = "offlineSDKRepositoryMaven"
    url = sdk6Uri
    patternLayout {
      artifact("[organisation]/[module]/[revision]/[artifact]-[revision](-[classifier])(.[ext])")
      ivy("[organisation]/[module]/[revision]/ivy-[revision].xml")
      setM2compatible(true)
    }
  }
}
```

- Add the previously created repository declaration inside the *repositories* block of both *dependencyResolutionManagement* and *pluginManagement* blocks:

```
dependencyResolutionManagement {
  repositories {
    ...
    offlineMicroEjSdk()
    ...
  }
}

pluginManagement {
  repositories {
    ...
    offlineMicroEjSdk()
    ...
  }
}
```

Offline Module Repository

There are 2 ways to create an Offline Module Repository containing the required modules:

- download an existing online repository.
- create an offline module repository project to create a custom repository.

Download an existing online repository

A quick way to get an Offline Module Repository for the modules is to download an existing online repository. MicroEJ provides several *module repositories*, the main one being the *Central Repository*.

If this online repository, or another one, contains all the module required for your project, download it. For example for the Central Repository, go to its location and click on the Download button.

Now go to *this section* to configure your project to use it.

Custom Offline Module Repository

If you need a custom Offline Module Repository (for example because the available online repositories does not contain all the modules required by your project, or you want to control exactly what contains the repository), you can create your own. Refer to *this page*.

Once done, go to *this section* to configure your project to use it.

Use an Offline Module Repository

When the Offline Module Repository of the modules has been retrieved or created, you can configure your projects to use it:

- Unzip the Offline Module Repository archive at the location of your choice, for example in the `C:\module-repository` folder.
- Depending on the SDK used to build the repository, add it to your project as follows:

SDK 6

SDK 5

Add the following line in the `build.gradle.kts` file of your project to declare the repository:

```
apply(file("C:\\module-repository\\module-repository.gradle.kts"))
```

Add the following repositories declaration in *your repositories configuration script*, inside the `repositories` block:

```
repositories {

  ...

  maven {
      name = "offlineModulesRepositoryMaven"
      url = uri("C:\\modules-repository")
  }
  ivy {
      name = "offlineModulesRepositoryIvy"
      url = uri("C:\\modules-repository")
      patternLayout {
          artifact("[organisation]/[module]/[revision]/[artifact]-[revision](-[classifier])(.[ext])")
          ivy("[organisation]/[module]/[revision]/ivy-[revision].xml")
          setM2compatible(true)
      }
  }

  ...

}
```

3.4.26 How-to Guides

How To Define a Specific Java Home for Gradle

By default, Gradle uses the JDK defined in the `JAVA_HOME` environment variable or in the `PATH`. If you want to use a different JDK without changing the default JDK of your system, you can define the property `org.gradle.java.home` in the Gradle Properties. Gradle Properties can be defined in the following locations, sorted by the highest priority:

- command line, as set using `-D`.
- `gradle.properties` in the `GRADLE_USER_HOME` directory (defaults to `<USER_HOME>/.gradle`).
- `gradle.properties` in the project directory, then its parent project directory up to the build root directory.
- `gradle.properties` in the Gradle installation directory.

If an option is configured in multiple locations, the first one found in any of these locations wins. Therefore, if you want all your Gradle project to use a different JDK than the system default JDK, you can add the following property in the file `<USER_HOME>/.gradle/gradle.properties`:

```
org.gradle.java.home="C:\\path\\to\\the\\jdk"
```

How To Pass a Property to Project Build Script

It is sometimes needed to use properties to pass values to a project build script. This avoids to have hardcoded values in the project sources.

Gradle allows to define System Properties with the command line thanks to the `-D` prefix:

```
$ ./gradlew build -DmyPropertyName="myPropertyValue"
```

and use them with the API `providers.systemProperty("myPropertyName").get()`.

For example to define a local VEE Port directory, the project can be configured with:

```
dependencies {
  microejVee(files(providers.systemProperty("myVeePortPath").get()))
}
```

and built with:

```
$ ./gradlew build -DmyVeePortPath="C:\\path\\to\\my\\veePort\\directory"
```

The `providers.systemProperty("myPropertyName")` API returns a `org.gradle.api.provider.Provider` object, which provides other capabilities like:

- defining a default value if the System Property does not exist: `providers.systemProperty("myPropertyName").getOrElse("myDefaultValue")`
- returning `null` if the value does not exist: `providers.systemProperty("myPropertyName").getOrNull()`.

How To Skip a Gradle Task

When a task is executed, it is possible to skip one or more of the tasks on which the called task depends. For example, you can skip the `test` task if you want to build a project without executing the tests.

If you want to skip a task, one of the following ways can be used :

- Add the `-x` or `--exclude-task` option in the command line:

  ```
  ./gradlew build -x test
  ```

  The task is skipped for this execution only.

- Exclude the task in the build script of the project

  ```
  project.gradle.startParameter.excludedTaskNames.add("test")
  ```

  The task is never executed.

When one of these two ways is used, not only the task but also all the tasks on which it depends are skipped. For example, if you choose to skip the `test` task, all the tasks which are used to produce the test runtime classpath are also skipped.

Skip the task only

It is possible to skip a task but still execute the tasks on which it depends using one of the following ways :

- Disable the task in the build script of the project:

```
tasks.test {
  enabled = false
}
```

  The task is never executed.

- Define a predicate in the build script of the project:

```
tasks.test {
  val skipProvider = providers.gradleProperty("skipTest")
  onlyIf {
    !skipProvider.isPresent()
  }
}
```

  The task is skipped each time the predicate evaluates to `false`. In this example, the `test` task is not executed if the `skipTest` property is added in the command line:

```
./gradlew build -PskipTest
```

How To Automatically reload a Gradle project

By default, regardless of the IDE that you are using (Android Studio, IntelliJ IDEA or Eclipse), the reload of a Gradle project must be explicitly triggered by the user when the configuration of the project has changed. This allows to avoid reloading the project too frequently, but the user must not forget to manually reload the project to apply changes.

It is also possible to configure your IDE to automatically reload your Gradle project:

IntelliJ IDEA / Android Studio

Eclipse

The auto-reload of a Gradle project with IntelliJ IDEA / Android Studio can be enabled as follows:

- Click on `File` > `Settings…` .
- Go to `Build, Execution, Deployment` > `Build Tools` .
- Check the `Reload changes in the build scripts` option and check the `Any changes` option.

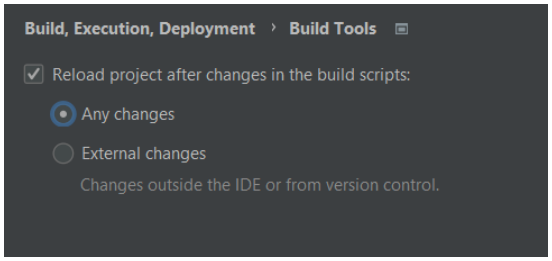


Fig. 98: Auto-reload option in IntelliJ IDEA / Android Studio

- Go to `Languages & Frameworks` > `Kotlin` > `Kotlin Scripting` .
- Check all the `Auto Reload` options.

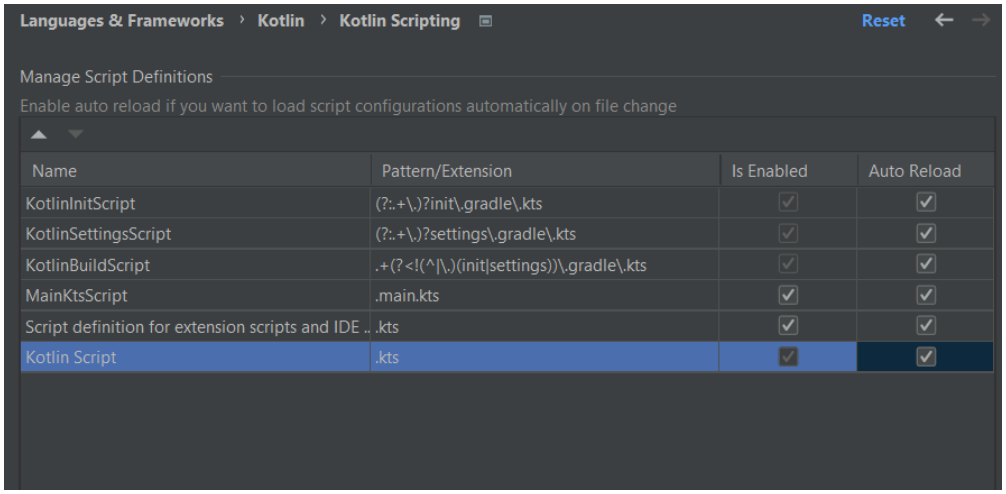


Fig. 99: Auto-reload Kotlin option in IntelliJ IDEA / Android Studio

The auto-reload of a Gradle project with Eclipse can be enabled as follows:

- Click on `Window` > `Preferences` > `Gradle` .
- Check the `Automatic Project Synchronization` option.

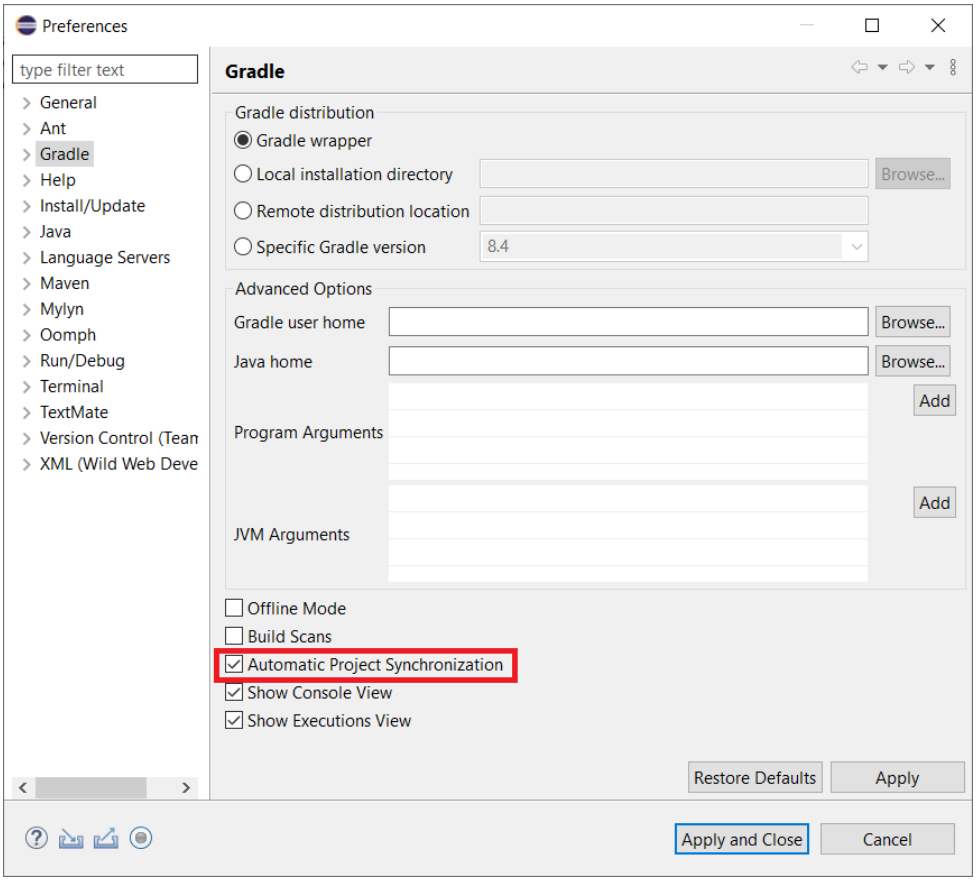


Fig. 100: Auto-reload option in Eclipse

How To Add a Repository

The *SDK 6 installation process* asks to create a Gradle Init Script file to declare modules and plugins repositories. You may need to use additional repositories or replace the default ones, for example to fetch a module only available in your company's repository. This page presents the different options to do that.

If you need more details on this topic, refer to the official Gradle documentation on repository declaration.

Note: In addition to the solutions described in the following sections, it is also possible to directly update the Gradle Init Script to add, replace or delete a repository. The version of this script provided in the *installation process* is a recommended version to be applied to quickly setup an environment. However, it can be modified to adapt it to your need, especially for the list of repositories. The module repositories are defined in the block beforeSettings > dependencyResolutionManagement > repositories and the plugin repositories are defined in the block beforeSettings > pluginManagement > repositories. They are applied to all the Gradle builds executed on the machine.

How To Add a Module Repository ...

... in addition to the Init Script Repositories

If the Module Repositories declared in the Gradle Init Script are declared in a `dependencyResolutionManagement` block, you can declare additional repositories for your project in the `settings.gradle.kts` file with:

```
dependencyResolutionManagement {
  repositories {
    maven {
      name = "myModuleRepository"
      url = uri("https://my.company/my-module-repository")
    }
  }
}
```

With this configuration, Gradle starts by fetching the modules from the repositories declared in the Init Script, then from the ones declared in the `settings.gradle.kts` project file.

... to replace the Init Script Repositories

If you want to declare the list of repositories directly in your project, no matter what is defined in the Gradle Init Script, you must declare them directly in the `build.gradle.kts` file of your project:

```
repositories {
  maven {
    name = "myModuleRepository"
    url = uri("https://my.company/my-module-repository")
  }
}
```

If you want to apply this configuration to all the subprojects of a multi-project, the repositories must be declared in a `build.gradle.kts` file located in the root folder, inside a `subprojects` block (or `allprojects` depending on your needs):

```
subprojects {
  repositories {
    maven {
      name = "myModuleRepository"
      url = uri("https://my.company/my-module-repository")
    }
  }
}
```

How To Add a Plugin Repository

To add a plugin repository, add a `pluginManagement > repositories` block in the `settings.gradle.kts` file of your project:

```
pluginManagement {
  repositories {
    maven {
      name = "myPluginRepository"
      url = uri("https://my.company/my-plugin-repository")
    }
  }
}
```

The repositories defined here are fetched **after** the ones defines in the Gradle Init Script.

Note: To be noted that if no plugin repository is defined (in either the Init Script or the project), Gradle uses the Gradle Plugin Portal.

How To Configure Multiple Gradle Repositories

If you want to make MicroEJ repositories available only to some projects, here is an example of configuration:

- Create a folder `repositories` in `<USER_HOME>/.gradle/init.d` .
- Move `microej.init.gradle.kts` to the `repositories` folder.
- Create a new `repositories.init.gradle.kts` file in `<USER_HOME>/.gradle/init.d` with the following content:

```
val defaultRepository = "myOtherRepo" // can be set to null
val selectedRepository = System.getProperty("gradle.repository") ?: defaultRepository

when (selectedRepository) {
   "microejCentral" ->
      apply {
         from("repositories/microej.init.gradle.kts")
      }
// "myOtherRepo" ->
//    apply {
//       from("repositories/other.gradle.kts")
//    }
}
```

- Add the following property to a `gradle.properties` file at the root of your SDK 6 projects:

```
systemProp.gradle.repository=microejCentral
```

This way, only projects defining the `gradle.repository` system property will use MicroEJ repositories. If you want to activate these repositories by default, you can edit the `defaultRepository` in the `repositories.init.gradle.kts` file.

Note: The name of the property `gradle.repository` is only given as exemple. You can choose the name you want as long as the propety defined in your `gradle.properties` file and in `repositories.init.gradle.kts` is the same.

Warning: If you put a repository configuration file that ends with .gradle.kts at the root of <USER_HOME>/.gradle/init.d , it will be automatically loaded. Contrary to what the official Gradle documentation says, the files does not need to end with .init.gradle.kts . That is the reason why we recommend to put the files in a folder. These files also need to end with .gradle.kts .

How To Resolve Dependencies in the IDE

When contributing to multiple interdependent projects, it is very convenient and more productive to test a change without rebuilding and publishing manually the updated projects.

Gradle allows to consider a local project as a dependency thanks to the Composite Build feature. For example, if you have a project named myApplication , with the coordinates com.mycompany:myApplication:1.0.0 , and a project named myLibrary , with the coordinates com.mycompany:myLibrary:1.0.0 , structured as follows:

```
|- myApplication
|    |- src
|    |- build.gradle.kts
|    |- settings.gradle.kts
|- myLibrary
|    |- src
|    |- build.gradle.kts
|    |- settings.gradle.kts
```

And the build.gradle.kts file of the myApplication project declaring a dependency to the myLibrary module:

```
dependencies {
  implementation("com.mycompany:myLibrary:1.0.0")
}
```

Without any additional configuration, Gradle will try to fetch the com.mycompany:myLibrary:1.0.0 dependency from the declared repositories. This means that when you do a change in the myLibrary project, it would require to build and publish it, then refresh dependencies on the myApplication project to get the update. This is painful and time consuming.

In order to avoid this, Gradle allows to consider the myLibrary build as part of the myApplication build, meaning that when the myApplication project is built, the myLibrary project is also rebuilt if it has been changed, and is used as the dependency. This can be configured by adding the following line in the settings.gradle.kts file of the myApplication project:

```
includeBuild("../myLibrary")
```

The path given to the includeBuild method is the relative path of the project to include.

Warning: The includeBuild method should be used to declare a dependency between two autonomous projects. To declare a dependency between two subprojects of a multi-project, a Project dependency must be used. Refer to the *Dependencies Between Subprojects of a Multi-Project* section for more information.

Refer to the Official Gradle documentation on the Composite Build feature for more details.

Dependencies Between Subprojects of a Multi-Project

Gradle allows to declare dependencies between subprojects of a multi-project build by declaring a Project dependency.

For example, if you have a multi-project named `myProject` composed of two subprojects `myApplication` and `myLibrary` :

```
|- myProject
|   |- myApplication
|   |   |- src
|   |   |- build.gradle.kts
|   |- myLibrary
|   |   |- src
|   |   |- build.gradle.kts
|   |- settings.gradle.kts
```

You can declare a Project dependency in the `build.gradle.kts` file of the `myApplication` subproject to make it depend on the `myLibrary` subproject:

```
dependencies {
  implementation(project(":myLibrary"))
}
```

When building the `myApplication` subproject, the `myLibrary` subproject is also rebuilt if it has been changed, so contrary to a Module dependency (e.g. `implementation("com.mycompany:myLibrary:1.0.0")`), you don't have to manually build and publish it, and then refresh dependencies on the `myApplication` project to get the update.

Refer to the Official Gradle documentation about the different kinds of dependencies for more details.

How To Build a Project

Generally speaking, building a project means compiling the source files, executing the tests and generating the module artifact. Depending on the nature of the project, the build can include other specific phases. Refer to the *Module Natures* page for a complete description of the build phases.

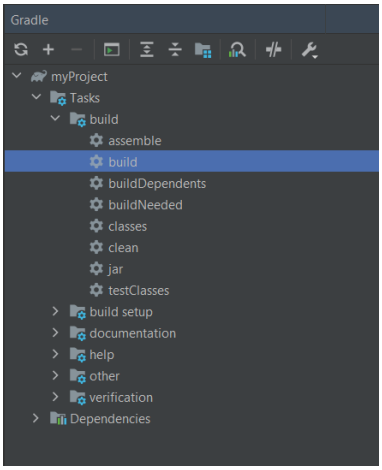
The build of a project is done by executing the Gradle `build` task.

IntelliJ IDEA / Android Studio

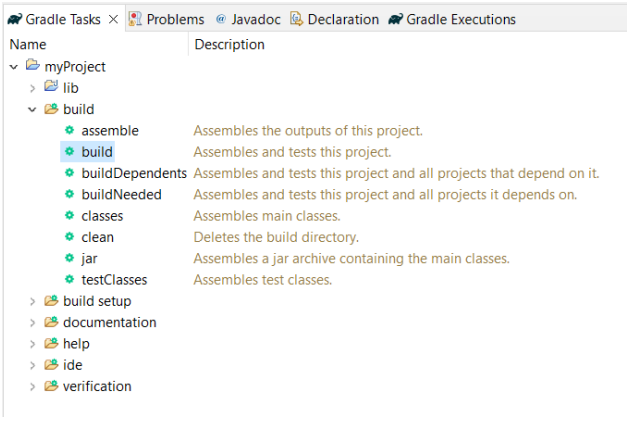
Eclipse

Command Line Interface

It can be executed from IntelliJ IDEA or Android Studio by double-clicking on the `build` task in the Gradle tasks view:



It can be executed from Eclipse by double-clicking on the `build` task in the Gradle tasks view:



It can be executed with the command line interface:

```
$ ./gradlew build
```

Gradle stores the artifacts produced by the build in the `build/libs` folder.

Note: If the build fails with a message related to the `Artifact Checker` such as:

```
The Artifact Checker found the following problems:
```

Fix the listed problems or skip the Artifact Checker by adding the following line in the build script file:

```
project.gradle.startParameter.excludedTaskNames.add("checkModule")
```

How To Build and Deploy Object Files without Building the Executable

If you need to produce the object files of the Application, deploy them to the BSP, but delegate the build of the Executable to another build system, call the buildExecutable task with the *Application option* deploy.bsp.microejscript set to false :

```
./gradlew buildExecutable -Dmicroej.option.deploy.bsp.microejscript=false
```

You should see logs similar to the following ones:

```
================ [ Deployment ] ================
The application output file has been generated here: 'C:\Users\johndoe\my-project\my-app\build\application\
↪object\SOAR.o'
================ [ Completed Successfully ] ================
VEE Port connected to BSP location 'C:\Users\johndoe\my-project\bsp' using platform option 'root.dir' in 'bsp/
↪bsp.properties'.
================ [ Deployment ] ================
MicroEJ application (microejapp.o) has been deployed to: 'C:\Users\johndoe\my-project\bsp\vee\lib'.
MicroEJ library (microejruntime.a) has been deployed to: 'C:\Users\johndoe\my-project\bsp\vee\lib'.
MicroEJ header files (*.h) have been deployed to: 'C:\Users\johndoe\my-project\bsp\vee\inc'.
================ [ Completed Successfully ] ================
```

How To Build an Executable With Multiple VEE Ports

When creating an Application, only one VEE Port must be defined to build an Executable. However, it is possible to build an Executable for a list of VEE Ports by using a Gradle multi-project.

For example, if you want to build an Executable for two VEE Ports, you can create a multi-project composed of two subprojects:

```
|- rootProject
|   |- myApplicationVeePort1
|   |   |- src/main/java
|   |   |- src/main/resources
|   |   |- build.gradle.kts
|   |- myApplicationVeePort2
|   |   |- build.gradle.kts
|   |- settings.gradle.kts
|   |- build.gradle.kts
```

- The myApplicationVeePort1 subproject is the Application project in which the first VEE Port is defined and the applicationEntryPoint property is set to the Fully Qualified Name of your main class.
- The myApplicationVeePort2 subproject is an Application project that only contains a build.gradle.kts file in which the second VEE Port is defined and the applicationEntryPoint property is set to the Fully Qualified Name of your main class. To avoid having to duplicate the Source code of the Application, a SourceSet must be defined to use the Source code of the myApplicationVeePort1 subproject:

```
sourceSets {
        main {
                java {
                        srcDirs(project(":myApplicationVeePort1").layout.projectDirectory.dir("src/main/java"),
                            project(":myApplicationVeePort1").layout.projectDirectory.dir("src/main/resources"))
                }
        }
}
```

- In the build.gradle.kts file of the multi-project, you can define the Gradle configuration that is common to all subprojects to avoid duplicates, for example:

```
plugins {
    id("com.microej.gradle.application") version "1.4.0" apply false
}

subprojects {
    apply(plugin = "com.microej.gradle.application")

    val implementation by configurations

    dependencies {
        implementation("ej.api:edc:1.3.5")
    }
}
```

- For each VEE Port, the Application configuration properties can be defined in the configuration folder of the corresponding Application project.

The Executable of the Application can now be built for a VEE Port by executing the buildExecutable task on the corresponding subproject:

```
./gradlew myApplicationVeePort1:buildExecutable
```

To build the Executable for all VEE Ports, a new task can be created in the build.gradle.kts file of the multi-project:

```
tasks.create("buildAllExecutables")
subprojects {
    rootProject.tasks.getByName("buildAllExecutables").dependsOn("$path:buildExecutable")
}
```

All Executables can now be built by executing the buildAllExecutables task:

```
./gradlew buildAllExecutables
```

For each VEE Port, the Executable is generated in the build/output/application folder of the corresponding sub-project.

How To Create a Custom Configuration in the IDE

This chapter explains how to create a new Configuration in all the supported IDEs.

IntelliJ IDEA / Android Studio

Eclipse

Visual Studio Code

The creation of a new Configuration with IntelliJ IDEA / Android Studio is done as follows:

- Click on Run > Edit Configurations… .
- Click on + and Select Gradle .
- Fill the name of the new Configuration in the Name field.
- Add a task and an option or a property if needed in the Run dialog, for example runOnSimulator -Pdebug.mode=true .

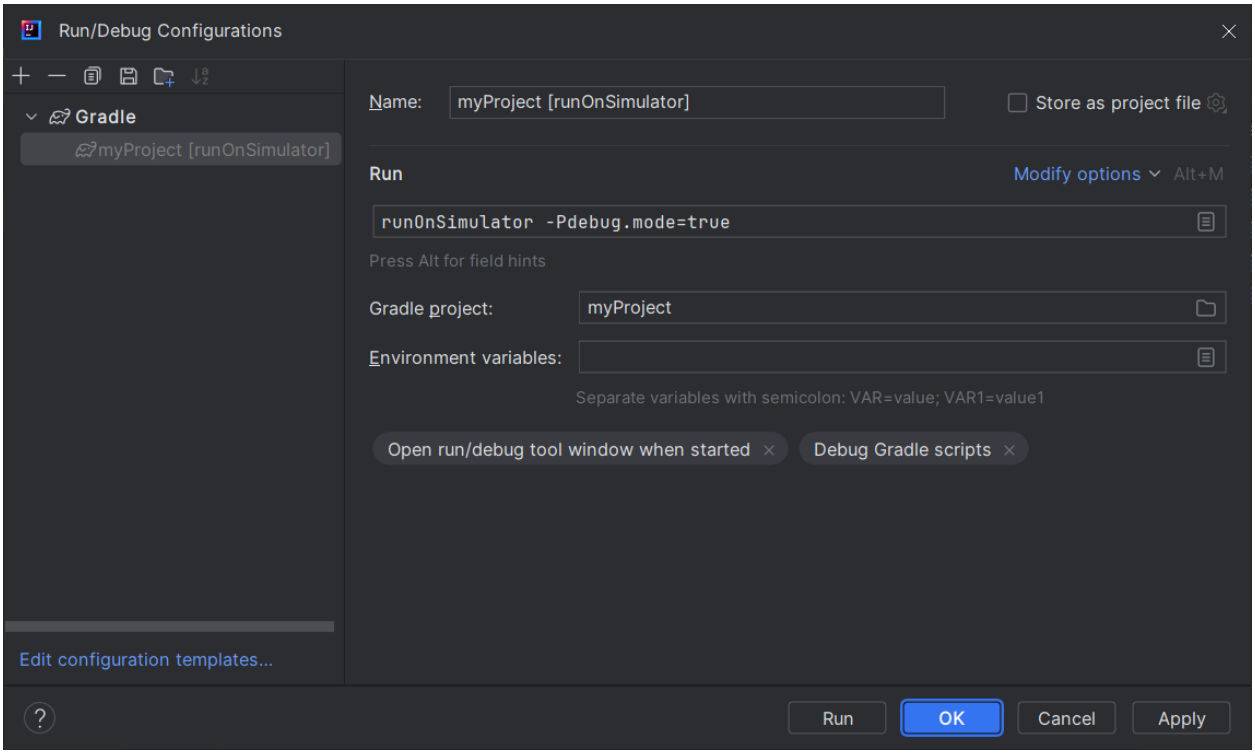


Fig. 101: Configuration Creation in IntelliJ IDEA

- Click on `OK` .
- Select the newly created Configuration in the drop-down list in the Navigation bar and click on the `run` button to launch it.

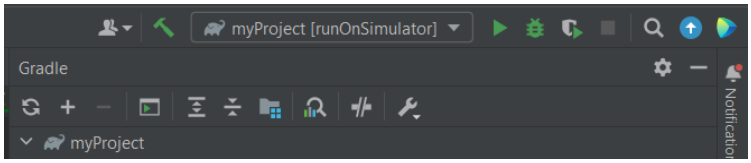


Fig. 102: Navigation bar in IntelliJ IDEA

- The Configuration can also be launched by double-clicking on it in the Gradle tasks view.

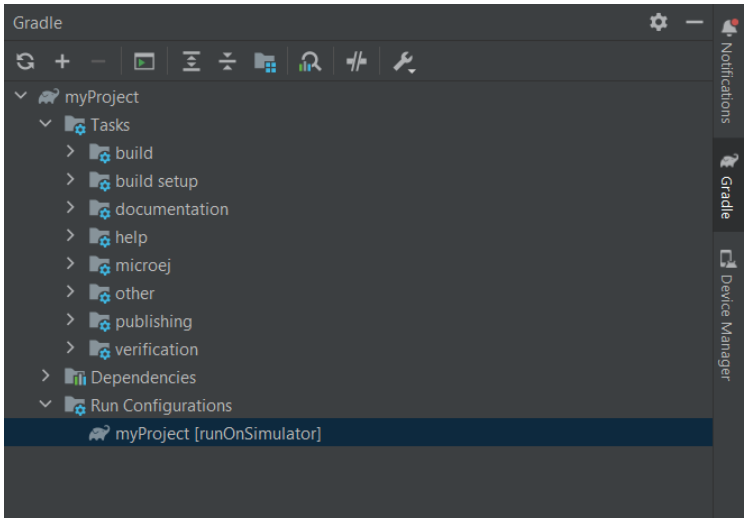


Fig. 103: Gradle view in IntelliJ IDEA

The creation of a new Configuration with Eclipse is done as follows:

- Click on Run > Run Configurations… .
- Right-click on Gradle Task and click on New Configuration .
- Fill the name of the new Configuration in the Name field.
- Add a task's name in the Gradle Tasks tab, for example runOnSimulator .

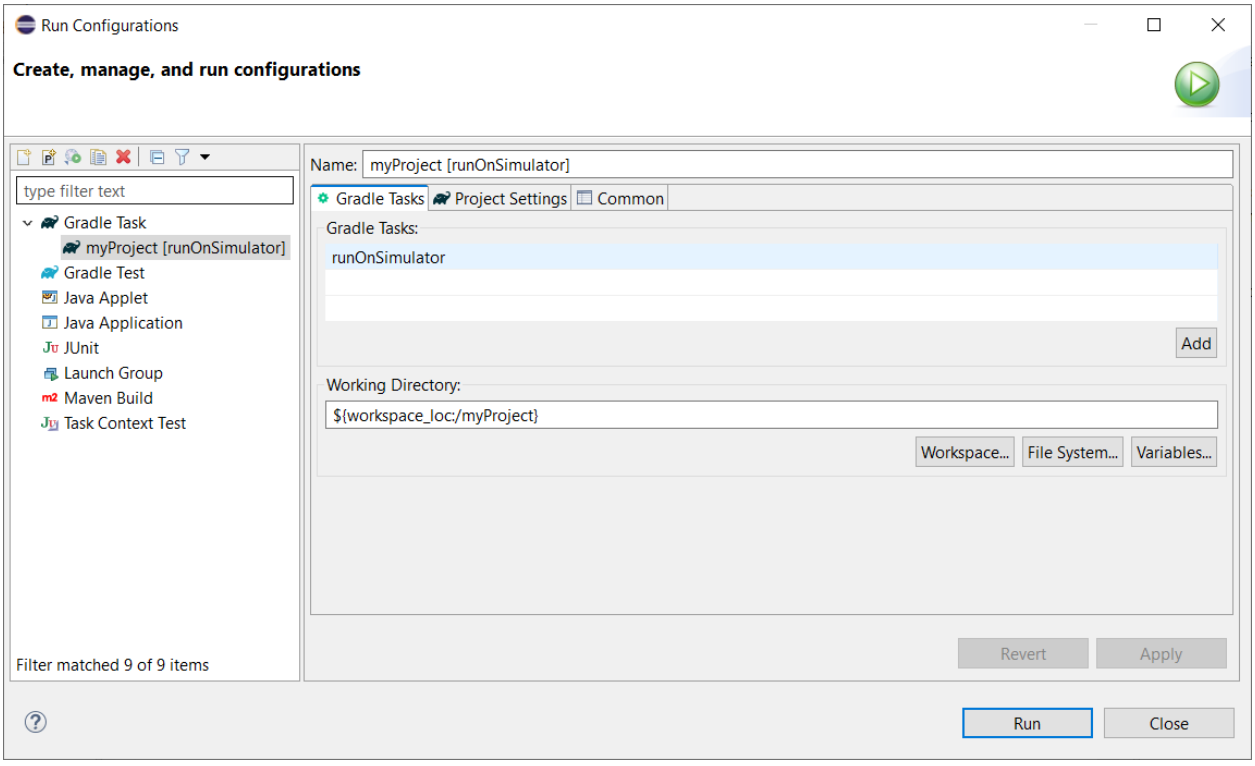


Fig. 104: Configuration Gradle Tasks tab in Eclipse

- Go to the Project Settings tab.
- Check Override project settings.
- Add an option or a property as a Program Argument if needed, for example -Pdebug.mode=true.

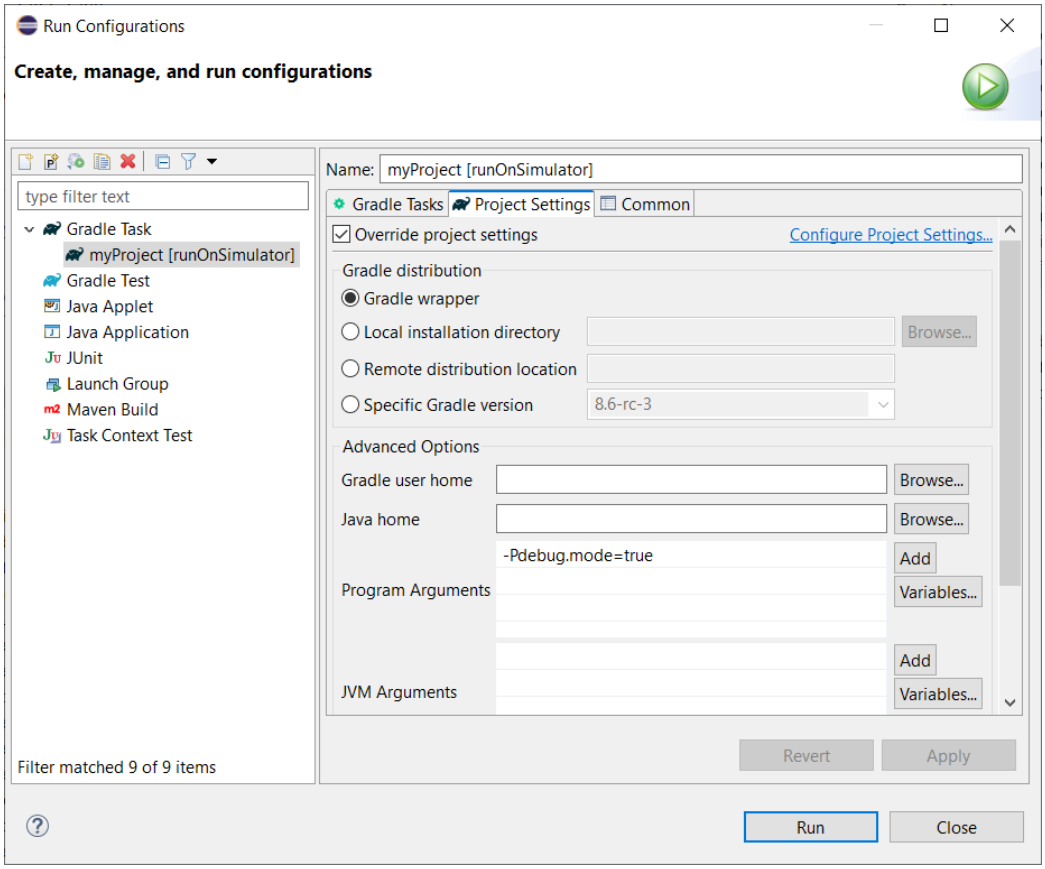


Fig. 105: Configuration Project Settings tab in Eclipse

Warning: Some tasks require to define specific options to be executed. These options must be defined with the task's name in the Gradle Tasks tab. For example, to run the *Local Deployment Socket*, the execTool task and its options must be specified:

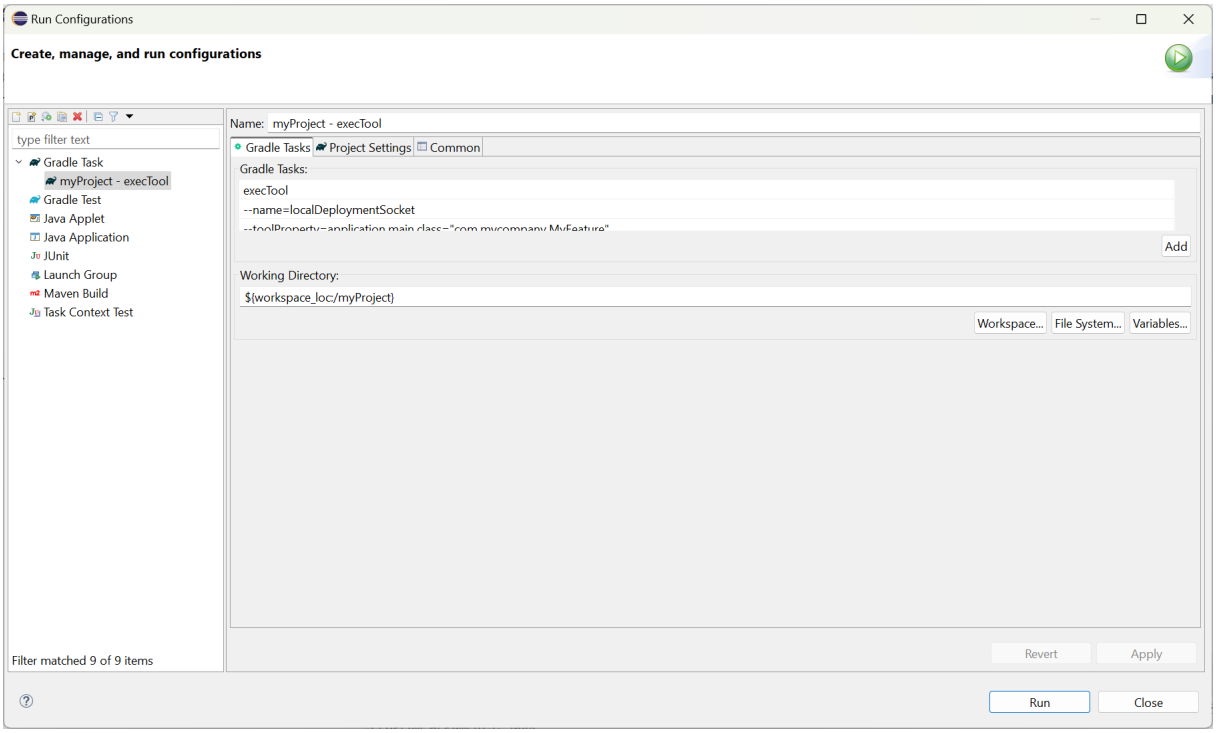


Fig. 106: Configuration execTool Task tab in Eclipse

- Click on Run to launch the Configuration.

The creation of a new Configuration with Visual Studio Code is done as follows:

- In the Gradle tasks view, right-Click on the task for which you want to create a new Configuration.
- Click on Pin Task With Args.

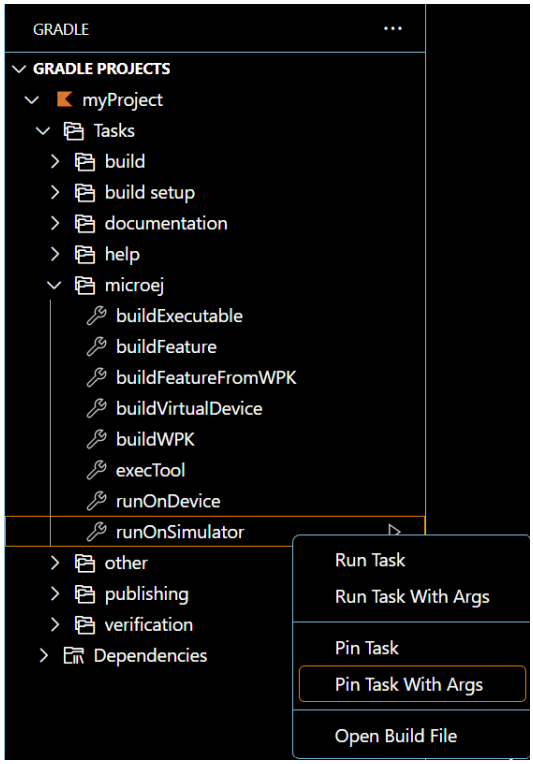


Fig. 107: New Gradle Configuration in Visual Studio Code

- Fill the option or property in the Search Bar and press Enter .

Warning: All task options must be defined **without** quotes in Visual Studio Code. For example, to run the *Local Deployment Socket*, the execTool task must be executed with the --name=localDeploymentSocket option.

If quotes are used (--name="localDeploymentSocket"), Visual Studio Code does not correctly pass the option to Gradle and the build fails with the following error:

```
> MicroEJ Tool '"localDeploymentSocket"' not found in <path\to\project>\build\vee\scripts\
Make sure that the correct MicroEJ VEE is selected.
```

- The newly created Configuration is available in the Gradle tasks view.

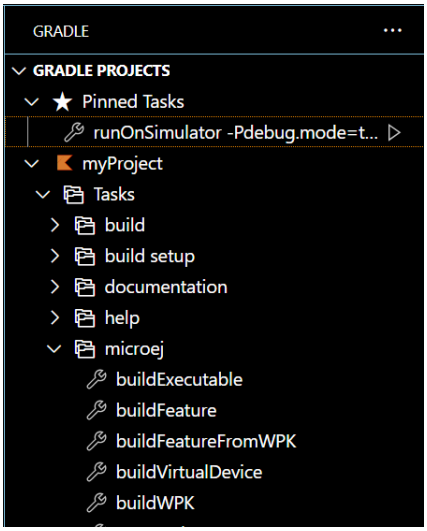


Fig. 108: Pinned Configuration in Visual Studio Code

How To Use a FeatureEntryPoint class as my Application EntryPoint

An Application can require the use of advanced features, for example the FeatureEntryPoint.stop() method, in order to communicate with a Kernel. To use such features, you must create a class implementing the ej.kf.FeatureEntryPoint interface. The creation of a Feature class is done as follows:

- Create a new Application project, as described in *Create a Project*.
- Add the KF module in the dependencies block of the build.gradle.kts file of the project:

```
dependencies {
  implementation("ej.api:kf:1.7.0")
}
```

- Create the Java class of the Feature EntryPoint in the src/main/java folder, for example:

```
package com.mycompany;

import ej.kf.FeatureEntryPoint;

public class MyFeature implements FeatureEntryPoint {
    @Override
    public void start() {
      System.out.println("Feature MyFeature started!");
    }

    @Override
    public void stop() {
      System.out.println("Feature MyFeature stopped!");
    }
}
```

- Define the property applicationEntryPoint in the microej configuration block of the build.gradle.kts file. It must be set to the Full Qualified Name of the Feature class, for example:

```
microej {
  applicationEntryPoint = "com.mycompany.MyFeature"
}
```

How To Disable Ivy Descriptor Publication

By default, SDK 6 adds an Ivy descriptor file to published artifacts in order for SDK 5 projects to be able to consume the module (e.g. an application built with SDK 5 depending on a library published with SDK 6).

Disable Ivy Descriptor Publication by Default

If you don't need SDK 5 backward compatibility, you can disable Ivy descriptor publication by setting the Gradle property `enable.ivy.descriptor.default=false`.

This property can be set in a `gradle.properties` file in `<USER_HOME>/.gradle/` to apply it globally, or at the root of the Gradle project to apply it at project level. It can also be passed to command line with `-Penable.ivy.descriptor.default=false` option.

Force Ivy Descriptor Publication When Disabled by Default

Even when Ivy descriptor publication is globally disabled with the Gradle property, it can be forced in a project by adding `withIvyDescriptor()` to the `microej` block in `build.gradle.kts` file:

```
microej {
  withIvyDescriptor()
}
```

How To Check Dependencies Updates

To check if Gradle dependencies are up-to-date, you can use the Gradle Versions Plugin. This Gradle plugin lists all the dependencies declared in the `build.gradle.kts` file, and tells whether they are up-to-date or if a new version is available.

Configure the Project

Edit your `build.grate.kts` file to declare the Gradle Versions Plugin plugin in addition to the MicroEJ plugin:

```
plugins {
    id("com.microej.gradle.application") version "1.4.0"
    id("com.github.ben-manes.versions") version "0.51.0"
}
```

In order to be able to check dependencies, you must disable *Version Check* option in the `microej` block:

```
microej {
    ...
    dependenciesVersionsCheckEnabled = false
}
```

Some dependencies are automatically included by the MicroEJ Gradle plugin to compile the project with JDT:

```
- org.eclipse.jdt:org.eclipse.jdt.compiler.apt:1.3.1000
- org.eclipse.jdt:org.eclipse.jdt.core:3.22.
```

These dependencies are not relevant for the report. To remove them, you can add the following snippet to the build file:

```
tasks.withType<DependencyUpdatesTask> {
    configurations.all {
        exclude(group = "org.eclipse.jdt")
    }
}
```

If your default Gradle configuration does not fetch plugins from *Gradle Plugin Portal*, you can add this repository to `settings.gradle.kts` :

```
pluginManagement {
    repositories {
        gradlePluginPortal()
    }
}
```

Generate the Report

Once these prerequisites are fulfilled, the report can be generated by calling the `dependencyUpdates` task in `help` group:

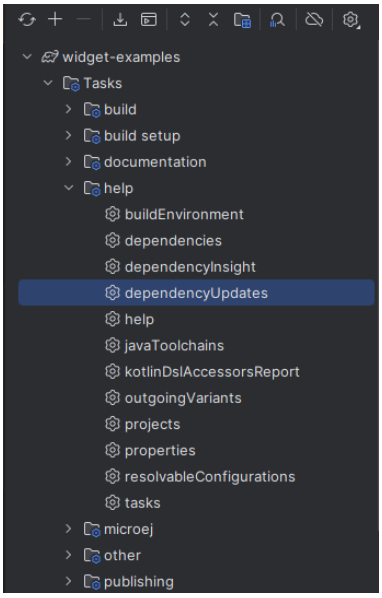
IntelliJ IDEA / Android Studio

Eclipse

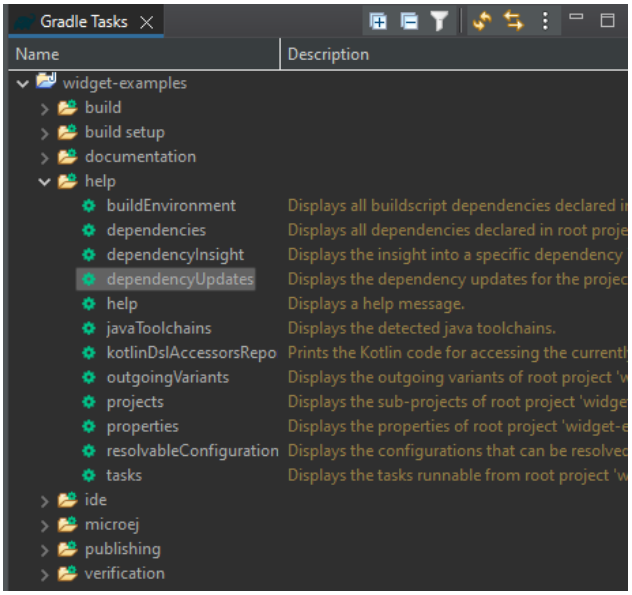
Visual Studio Code

Command Line Interface

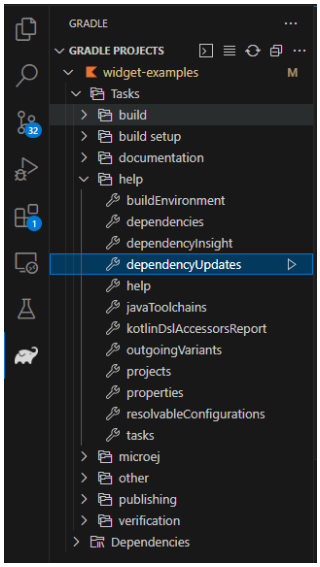
By double-clicking on the `dependencyUpdates` task in the Gradle tasks view:



By double-clicking on the `dependencyUpdates` task in the Gradle tasks view:



By double-clicking on the `dependencyUpdates` task in the Gradle tasks view:



From the command line interface:

```
$ ./gradlew dependencyUpdates
```

A simple text report is printed in the console and generated in `build/dependenciesUpdates/report.txt`. Report is configurable in the plugin. Refer to the Report Format section of the plugin README.

Here is an example of report:

```
------------------------------------------------------------
: Project Dependency Updates (report to plain text file)
------------------------------------------------------------

The following dependencies are using the latest milestone version:
- com.github.ben-manes.versions:com.github.ben-manes.versions.gradle.plugin:0.51.0
- com.is2t.tools:application-repository:2.2.0
```

(continues on next page)

(continued from previous page)

```
- com.microej.veeport.st.stm32f7508-dk:M5QNX_eval:2.2.0
- ej.library.eclasspath:stringtokenizer:1.2.0

The following dependencies exceed the version found at the milestone revision level:
- ej.library.ui:widget [5.2.0 <- 4.2.0]

The following dependencies have later milestone versions:
- com.microej.gradle.application:com.microej.gradle.application.gradle.plugin [0.15.0 -> 1.4.0]
- com.microej.test:junit-test-engine [0.2.2 -> 0.3.0]
- ej.api:drawing [1.0.2 -> 1.0.5]
- ej.api:edc [1.3.5 -> 1.3.7]
- ej.api:microui [3.1.0 -> 3.5.0]
- ej.library.eclasspath:collections [1.4.0 -> 1.4.2]
- ej.library.runtime:basictool [1.5.0 -> 1.7.0]
- ej.library.runtime:service [1.1.1 -> 1.2.0]
- ej.library.test:junit [1.7.1 -> 1.12.0]

Gradle release-candidate updates:
- Gradle: [8.3 -> 8.13]
```

How To Create Custom ExecTool Task

MicroEJ provides the Gradle `execTool` task that allows to execute all *Command Line Tools*. However, it is possible to create a custom task of type `ExecToolTask` specific to a tool. This task can then be used to execute the tool before or after another task when building your project for example.

You can create a new `ExecToolTask` in the build file of your project as follows:

- Import the `ExecToolTask` and `LoadVeeTask` classes:

```
import com.microej.gradle.tasks.ExecToolTask
import com.microej.gradle.tasks.LoadVeeTask
```

- Create a new task of type `ExecToolTask`:

```
val loadVee = tasks.withType(LoadVeeTask::class).named("loadVee")
val mainSourceSet = project.extensions.getByType(SourceSetContainer::class).getByName(SourceSet.
↪MAIN_SOURCE_SET_NAME)

tasks.register<ExecToolTask>("myCustomTask") {
    veeDir.set(loadVee.flatMap { it.loadedVeeDir })
    resourcesDirectories.from(mainSourceSet.output.resourcesDir, layout.buildDirectory.dir("generated/
↪microej-app-wrapper/resources"))
    classesDirectories.from(mainSourceSet.output.classesDirs, layout.buildDirectory.dir("generated/microej-
↪app-wrapper/classes"))
    classpathFromConfiguration.from(project.configurations.getByName("runtimeClasspath"))

    toolName = "microej_tool_name"
    toolProperties.putAll(mapOf(
        "property1" to "value1",
        "property2" to "value2",
        ...
    ))
}
```

Note: Only the `toolName` and `toolProperties` inputs are specific to the tool. The other task inputs are common to all tasks of type `ExecToolTask`.

- The custom task can then be executed:

```
$ ./gradlew myCustomTask
```

As an example, here is how to create a task which calls the *Local Deployment Socket* tool:

```
import com.microej.gradle.tasks.ExecToolTask
import com.microej.gradle.tasks.LoadKernelExecutableTask
import com.microej.gradle.tasks.LoadVeeTask

plugins {
    id("com.microej.gradle.application") version "1.4.0"
}

group="com.mycompany"
version="1.0.0"

microej {
    applicationEntryPoint = "com.mycompany.MyFeature"
}

dependencies {
    ...
    microejVee("com.mycompany:myKernel:1.0.0")
    ...
}

...

val loadVee = tasks.withType(LoadVeeTask::class).named("loadVee")
val loadKernelExecutable = tasks.withType(LoadKernelExecutableTask::class).named("loadKernelExecutable")
val mainSourceSet = project.extensions.getByType(SourceSetContainer::class).getByName(SourceSet.MAIN_
↪SOURCE_SET_NAME)
val buildDirectory = layout.buildDirectory

tasks.register<ExecToolTask>("localDeploy") {
    // These inputs are required for all custom tasks of type "ExecToolTask".
    veeDir.set(loadVee.flatMap { it.loadedVeeDir })
    resourcesDirectories.from(mainSourceSet.output.resourcesDir, buildDirectory.dir("generated/microej-app-
↪wrapper/resources"))
    classesDirectories.from(mainSourceSet.output.classesDirs, buildDirectory.dir("generated/microej-app-wrapper/
↪classes"))
    classpathFromConfiguration.from(project.configurations.getByName("runtimeClasspath"))

    // These inputs are specific to the Local Deployment Socket.
    toolName = "localDeploymentSocket"
    inputs.file(loadKernelExecutable.flatMap { it.loadedKernelExecutableFile })
    toolProperties.putAll(mapOf(
        "application.main.class" to microej.applicationEntryPoint,
        "board.server.host" to "10.0.0.171",
        "board.server.port" to "4000",
        "board.timeout" to "120000",
        "use.storage" to "true"
```

(continues on next page)

(continued from previous page)

```
    ))

    doFirst {
        // Use the Executable of the Kernel provided as dependency to build the Feature file.
        toolProperties["kernel.filename"] = loadKernelExecutable.get().loadedKernelExecutableFile.get().asFile.
↪absolutePath
    }
}
```

You can then execute the task to build and deploy the Feature file of the Application:

```
$ ./gradlew localDeploy
```

How To Pass JVM Arguments to the VEE Launch Scripts

Some tasks (e.g., `runOnSimulator`, `buildExecutable`) start a new JVM process to run the VEE launch scripts. The arguments for this JVM can be configured to customize the execution environment, for example for adjusting memory settings or enabling remote debugging.

The JVM arguments have to be set in a system property named `microej.launch.jvmargs`. This property accepts a whitespace-separated list of options.

The following example shows how to control the maximum and initial heap of the JVM used to launch the simulator:

```
$ ./gradlew runOnSimulator -Dmicroej.launch.jvmargs="-Xmx1024m -Xms512m"
```

Note: The options passed with the property `microej.launch.jvmargs` only apply to the JVM forked during the project build. They do not affect the JVM configuration of the Gradle Daemon. To configure the options of the Gradle Daemon, you can use the property `org.gradle.jvmargs`, as described in the Gradle Properties Reference.

Debugging Options

To debug the JVM used to launch the VEE scripts, set the `microej.launch.jvmargs` system property like the following:

```
-Dmicroej.launch.jvmargs="-agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=*:5005"
```

Then, connect the IDE's debugger to host `localhost` on specified port (`5005` in this example).

This can be used for advanced use cases that require to debug the SDK tools, like the SOAR for example.

How To Use Proguard

ProGuard is a tool that shrinks, optimizes, and obfuscates Java code. It can be executed as a task using the official Proguard Gradle plugin.

The code below registers a Proguard task that is executed whenever the `jar` task runs. To use it, add the following snippet in your `build.gradle.kts` file:

```
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath("com.guardsquare:proguard-gradle:7.8.2")
    }
}

// Registers the Proguard task
tasks.register<proguard.gradle.ProGuardTask>("proguard") {
    dependsOn("jar")
    mustRunAfter("test")
    val jarTask = tasks.named("jar")
    val libsDir = layout.buildDirectory.dir("libs")
    val nonObfuscatedName = "${project.name}-${version}-non-obfuscated.jar"
    val nonObfuscatedJar = layout.buildDirectory.file("libs/$nonObfuscatedName")

    doFirst {
        copy {
            from(jarTask)
            into(libsDir)
            rename { nonObfuscatedName }
        }
    }

    configuration("module.pro")
    injars(nonObfuscatedJar)
    libraryjars(files(configurations.compileClasspath))
    outjars(jarTask)
}

// Makes Proguard automatically run whenever jar runs
tasks.named("jar") {
    finalizedBy("proguard")
}
```

To run Proguard on the project, you can either call `./gradlew proguard` directly or build the module with `./gradlew build` .

In this example, the Proguard configuration is defined in a file named `module.pro` at the root of the project. See the ProGuard manual for all available options.

Javadoc and Sources Management

ProGuard does not automatically update Javadoc or sources JARs, it only obfuscates the compiled `.class` files contained in the main artifact. As a result, the published Javadoc and sources may not correspond to the obfuscated bytecode and can inadvertently expose implementation details or sensitive code that should not be publicly available.

If this default behavior is not acceptable, you have two possible approaches:

- Do not publish Javadoc or sources for obfuscated artifacts.
- Restrict the published Javadoc and sources to classes matching ProGuard `-keep` rules.

Approach 1: Do not Publish Javadoc/Sources for Obfuscated Code

If publishing Javadoc and source artifacts is not required, you can simply disable the `javadocJar` and `sourcesJar` tasks. In this case, the published module will not include the `<module-name>-javadoc.jar` and `<module-name>-sources.jar` artifacts.

Add the following to your `build.gradle.kts` file:

```
tasks.named("javadocJar") {
    enabled = false
}

tasks.named("sourcesJar") {
    enabled = false
}
```

Approach 2: Include Only Classes Matching ProGuard `-keep` Rules

A more selective approach consists of including only the classes that match the ProGuard `-keep` rules.

In the example below, the `javadoc` and `sourcesJar` tasks are configured to include only the classes explicitly listed with `-keep` directives in the `module.pro` file:

```
// Lists the classes matching "-keep" in the "module.pro".
fun getKeptClasses(): List<File> =
    file("module.pro").readLines()
        .map { it.trim() }
        .mapNotNull { line ->
            val regex = Regex("""^-keep\s+.*\bclass\s+([\w.$-]+)""")
            regex.find(line)?.groupValues?.get(1)
        }
        .map { fqn -> file("src/main/java/${fqn.replace('.', '/')}.java") }
        .filter { it.exists() }

// Generates Javadoc only for the classes matching "-keep".
tasks.named<Javadoc>("javadoc") {
    dependsOn("proguard")
    setSource(files(getKeptClasses()))
}

// Packages the sources only with the classes matching "-keep".
tasks.named<Jar>("sourcesJar") {
```

(continues on next page)

(continued from previous page)

```
    dependsOn("proguard")
    getKeptClasses().forEach {
        c -> include(c.relativeTo(file("src/main/java")).path)
    }
}
```

As a result, the published Javadoc and source artifacts will contain only the classes declared as part of the public API.

Note: Depending on your Proguard configuration and the organization of your code, this approach may still include obfuscated methods within public classes. This occurs because the inclusion logic operates at the file (class) level rather than at the method level.

How To Enable Class Loading Verbose Logs

To display verbose logs about classes loaded by the JVM, you can define the `org.gradle.jvmargs` property combined with JVM `-Xlog:class` option using file redirection. Here is a typical usage:

Windows

Linux/MacOS

```
-Dorg.gradle.jvmargs=-Xlog:class+load=info:file=%TEMP%/gradle-daemon-classes.log:time,level
```

```
-Dorg.gradle.jvmargs=-Xlog:class+load=info:file=$TMPDIR/gradle-daemon-classes.log:time,level
```

This redirects the logs in the file `gradle-daemon-classes.log` in the OS temporary folder. It should contain such logs:

```
[2026-01-27T15:27:16.858+0100][info] org.slf4j.Marker source: file:/C:/Users/jdoe/.gradle/wrapper/dists/gradle-8.
↪10.2-bin/a04bxjujx95o3nb99gddekhwo/gradle-8.10.2/lib/slf4j-api-1.7.36.jar
[2026-01-27T15:27:16.858+0100][info] org.slf4j.helpers.BasicMarker source: file:/C:/Users/jdoe/.gradle/wrapper/
↪dists/gradle-8.10.2-bin/a04bxjujx95o3nb99gddekhwo/gradle-8.10.2/lib/slf4j-api-1.7.36.jar
[2026-01-27T15:27:16.859+0100][info] org.slf4j.LoggerFactory source: file:/C:/Users/jdoe/.gradle/wrapper/dists/
↪gradle-8.10.2-bin/a04bxjujx95o3nb99gddekhwo/gradle-8.10.2/lib/slf4j-api-1.7.36.jar
[2026-01-27T15:27:16.859+0100][info] org.slf4j.event.LoggingEvent source: file:/C:/Users/jdoe/.gradle/wrapper/
↪dists/gradle-8.10.2-bin/a04bxjujx95o3nb99gddekhwo/gradle-8.10.2/lib/slf4j-api-1.7.36.jar
```

The JVM provides more options and logs categories, refer to the official documentation to learn more.

3.4.27 Appendices

Virtual Device

This chapter describes the structure of a *Virtual Device*.

Structure

A Virtual Device is structured as follows:

```
|- virtualDevice
|   |- installed-applications
|   |- javaLibs
|   |- MICROJVM
|   |- main-application
|   |- mocks
|   |- options
|   |   |- target.properties
|   |- resources
|   |- S3
|   |- scripts
|   |   |- init-vd
|   |   |   |- vd-init.xml
|   |- tools
|   |- architecture.properties
|   |- release.properties
|   |- veePort.properties
|   |- workbenchExtension*.jar
```

The Virtual Device contains the Simulation part files of the VEE Port used to build it:

- the javaLibs/ folder, that contains the Foundation Libraries which are common to MICROJVM and S3
- the linker/ folder, that contains the Linker jar files. This folder is not embedded in the Virtual Device if an Architecture 8.0.0 is used
- the MICROJVM/ folder, that contains the VEE Port's files required to build a Feature file (.fo)
- the mocks/ folder, that contains the Jar files of the mocks for Foundation Libraries
- the resources/ folder, that contains the OS specific libraries
- the S3/ folder, that contains the Simulator, HIL and Foundation Libraries specific to the Simulator
- the scripts/ folder, that contains launch and initialization scripts
- the tools/ folder
- the workbenchExtension*.jar files

The following elements are also embedded in the Virtual Device:

- a Kernel Application, whose WPK file is extracted in the main-application folder of the Virtual Device
- the WPK files of pre-installed Applications that are extracted in their own folder, in the installed-applications/ folder that is empty by default

---

Note: Applications can only be pre-installed in a Multi-Sandbox Virtual Device. In case of a Mono-Sandbox Virtual Device, the installed-applications/ folder is always empty.

- the options/target.properties , that contains the properties of the VEE Port used to build the Virtual Device
- the scripts/init-vd/vd-init.xml script, that allows to enable or not the Virtual Device. If the Virtual Device is not enabled, the Application main class specified by the user is launched on the VEE Port

You can refer to the *Build a Virtual Device* page to know how to build a Virtual Device.

Dependencies Configurations

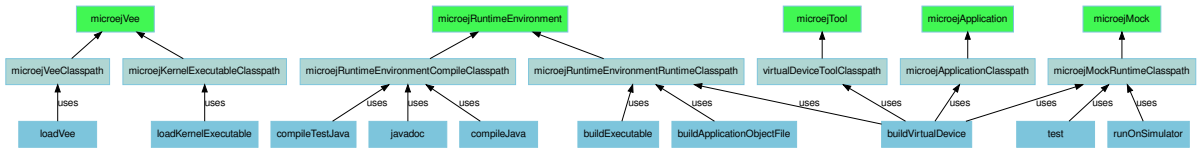
This chapter describes all the dependency configurations added to your project by the MicroEJ Gradle plugins.

Note: The MicroEJ Gradle plugins extend the Gradle Java and Java Library plugins. For more information about the configurations inherited from those plugins, refer to the official documentation :

- Java plugin
- Java Library plugin

The following graph describes the dependency configurations. Use this legend to interpret the colors:

- Green background : Dependencies can be declared against this configuration
- Gray background : This configuration is for consumption by tasks only
- Blue background : A task



The MicroEJ Gradle plugins also define dependency configurations for internal use:

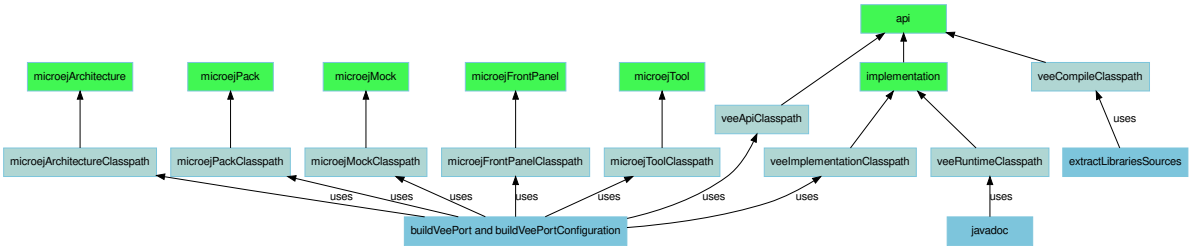
- antScriptLauncherClasspath , used by the *buildVirtualDevice* task
- addonProcessorClasspath , used by the *adp* task
- jdtCompilerClasspath , used by the *Add-On Library* and *Application* plugins
- wrapperClasspath , used by the compileJava task
- microejRuntimeEnvironmentFullRuntimeClasspath , used by the *buildVirtualDevice*, *buildApplicationObjectFile* and *buildExecutable* tasks
- microejFullRuntimeClasspath , used by the *runOnSimulator* and *buildApplicationObjectFile* tasks
- microejVeeFullClasspath , used by the *loadVee* task
- microejVeeRuntimeClasspath , used by the *Application* plugin
- microejMockClasspath , used by the *Add-On Library* plugin

Dependencies Configurations in a VEE Port Project

This section describes all the dependency configurations added to your VEE Port project by the MicroEJ Gradle plugins.

The following graph describes the dependency configurations. Use this legend to interpret the colors:

- Green background: Dependencies can be declared against this configuration
- Gray background: This configuration is for consumption by tasks only
- Blue background: The tasks

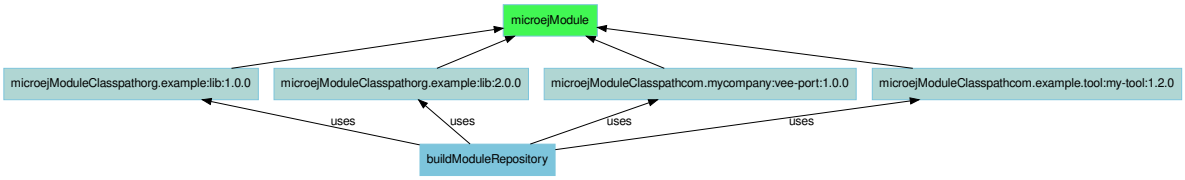


Dependencies Configurations in a Module Repository Project

This section describes all the dependency configurations added to your Module Repository project by the MicroEJ Gradle plugins.

The following graph describes the dependency configurations. Use this legend to interpret the colors:

- Green background: Dependencies can be declared against this configuration
- Gray background: This configuration is for consumption by tasks only
- Blue background: The tasks



For each dependency defined with microejModule , a configuration microejModuleClasspath<group:name:version> , where <group:name:version> is the group, name and version of the dependency, is created to ensure that each dependency has its own resolution graph.

Variants

The MicroEJ Gradle plugins define a list of variants allowing to fetch the right component of a dependency based on the consumer's requirements. Each variant is configured with its own attributes. Refer to *Attributes of a Variant* for more information about attributes.

Add-On Library

The *Add-On Library* plugin defines the following variants:

runtimeAndMockElements

This variant is used to fetch a Library and its dependencies when it is declared with the `implementation` or `api` configuration in your VEE Port project. It is configured with the following attribute:

- the custom `com.microej.veeport.artifact.usage` attribute, set to `default`

additionalElements

This variant is used to publish the README, CHANGELOG and License files of the project. It is configured with the following attribute:

- the custom `com.microej.artifact.element` attribute, set to `additional-files`

Application

The *Application* plugin defines the following variants:

microejWPK

This variant is used to fetch the WPK of an Application when a dependency is declared with the `microejApplication` configuration in your project. It is configured with the following attributes:

- the custom `com.microej.artifact.element` attribute, set to `application-wpk`

microejExecutable

This variant is used to fetch the Executable of an Application when a dependency is declared with the `microejVee` configuration in your project. It is configured with the following attributes:

- the custom `com.microej.artifact.element` attribute, set to `executable`

microejExecutableBuildFiles

This variant is used to publish the files generated when building the Executable of an Application. It is configured with the following attribute:

- the custom `com.microej.artifact.element` attribute, set to `executable-build-files`

microejVirtualDevice

This variant is used to fetch the Virtual Device of an Application when a dependency is declared with the `microejVee` configuration in your project. It is configured with the following attributes:

- the custom `com.microej.artifact.element` attribute, set to `virtual-device`

microejFeatureBuildFiles

This variant is used to publish the files generated when building the Feature of an Application. It is configured with the following attribute:

- the custom com.microej.artifact.element attribute, set to feature-build-files

additionalElements

This variant is used to publish the README, CHANGELOG and License files of the project. It is configured with the following attribute:

- the custom com.microej.artifact.element attribute, set to additional-files

Mock

The *Mock* plugin defines the following variants:

microejMockRip

This variant is used to fetch the RIP of a Mock when a dependency is declared with the microejMock configuration in your project. It is configured with the following attributes:

- the custom com.microej.veeport.artifact.usage attribute, set to default

additionalElements

This variant is used to publish the README, CHANGELOG and License files of the project. It is configured with the following attribute:

- the custom com.microej.artifact.element attribute, set to additional-files

Runtime Environment

The *Runtime Environment* plugin defines the following variants:

runtimeEnvironment

This variant is used to fetch the Runtime Environment Jar when a dependency is declared with the microejRuntimeEnvironment configuration in your project. It is configured with the following attributes:

- the custom com.microej.artifact.element attribute, set to runtime-environment-api

additionalElements

This variant is used to publish the README, CHANGELOG and License files of the project. It is configured with the following attribute:

- the custom `com.microej.artifact.element` attribute, set to `additional-files`

VEE Port

The *VEE Port* plugin defines the following variants:

microejVeePort

This variant is used to fetch the VEE Port when it is declared with the `microejVee` configuration in your project. It is configured with the following attributes:

- the custom `com.microej.veeport.artifact.usage` attribute, set to `default`
- the standard `Usage` attribute, set to `java-runtime`

apiElements

This variant is used to fetch the Libraries provided by the VEE Port when it is declared with the `microejVee` configuration in your project. It is configured with the following attributes:

- the standard `Category` attribute, set to `library`
- the standard `BUNDLING` attribute, set to `external`
- the standard `LIBRARY_ELEMENTS` attribute, set to `jar`
- the standard `Usage` attribute, set to `java-api`

runtimeElements

This variant is used to fetch the Libraries provided by the VEE Port when it is declared with the `microejVee` configuration in your project. It is configured with the following attributes:

- the standard `Category` attribute, set to `library`
- the standard `BUNDLING` attribute, set to `external`
- the standard `LIBRARY_ELEMENTS` attribute, set to `jar`
- the standard `Usage` attribute, set to `java-runtime`

javadocElements

This variant is used to publish the aggregated Javadoc Jar file of all Libraries provided by the VEE Port with the `api` and the `implementation` configurations in your project. It is configured with the following attributes:

- the standard `Category` attribute, set to `documentation`
- the standard `Bundling` attribute, set to `external`
- the standard `DocType` attribute, set to `javadoc`
- the standard `Usage` attribute, set to `java-runtime`

additionalElements

This variant is used to publish the README, CHANGELOG and License files of the project. It is configured with the following attribute:

- the custom `com.microej.artifact.element` attribute, set to `additional-files`

Module Repository

The *Module Repository* plugin defines the following variants:

microejModuleRepository

This variant is used to publish the Module Repository archive file. It is configured with the following attributes:

- the custom `com.microej.veeport.artifact.usage` attribute, set to `module-repository`
- the standard `Usage` attribute, set to `java-runtime`

additionalElements

This variant is used to publish the README, CHANGELOG and License files of the project. It is configured with the following attribute:

- the custom `com.microej.artifact.element` attribute, set to `additional-files`

Attributes of a Variant

An attribute allows Gradle to select the right variant depending on the consumer's requirements. They are two types of attributes: standard attributes and custom attributes. For more information about variants and attributes, refer to the official documentation.

Standard Attributes

The standard attributes are defined by Gradle. Such an attribute is mandatory so the resolution of a dependency will fail if the producer did not define a variant with the attribute set to the same value. All standard attributes are listed in the official documentation.

For example, a VEE Port built with SDK 6 1.2.0 or older is published with a single microejVeePort variant configured with the following attributes:

- the standard Category attribute, set to library
- the standard Bundling attribute, set to embedded
- the standard jvm.version attribute, set to 7
- the standard Libraryelements attribute, set to microej-vee-port
- the standard Usage attribute, set to java-runtime

When the VEE Port is declared as dependency with the implementation configuration, the following attributes are expected at runtime:

- the standard Category attribute, set to library
- the standard Bundling attribute, set to external
- the standard jvm.environment attribute, set to standard-jvm
- the standard jvm.version attribute, set to 11
- the standard Libraryelements attribute, set to jar
- the standard Usage attribute, set to java-runtime

So the resolution fails because the microejVeePort variant of the VEE Port is incompatible:

```
> Could not resolve all files for configuration ':runtimeClasspath'.
> Could not resolve com.mycompany:myVeePort:1.0.0.
  Required by:
      root project :
   > No matching variant of com.mycompany:myVeePort:1.0.0 was found. The consumer was configured to find a␣
↪library for use during runtime, compatible with Java 8, packaged as a jar, preferably optimized for standard␣
↪JVMs, and its dependencies declared externally but:
       - Variant 'microejVeePort' declares a library for use during runtime, compatible with Java 7, and its␣
↪dependencies bundled (fat jar):
           - Incompatible because this component declares a component, with the library elements 'microej-vee-port'␣
↪and the consumer needed a component, packaged as a jar
           - Other compatible attribute:
               - Doesn't say anything about its target Java environment (preferred optimized for standard JVMs)
```

If no variant matches the consumer's requirements, it is possible to make a variant compatible by defining a compatibility rule.

Custom Attributes

The custom attributes are defined by the user and are optional. When such attributes are used, the dependency resolution may differ depending on if the consumer defines them with standard attributes or not:

If the consumer specifies custom attributes only, the variant is selected as follows:

- If a variant is configured with the same custom attributes, Gradle selects it.
- If no variant is compatible with the custom attributes, Gradle selects a runtime variant among the ones configured with standard attributes. For Java modules, the variant selected by default is `runtimeElements`.
- If there is no variant with standard attributes, the resolution fails.

If the consumer specifies custom and standard attributes, the variant is selected as follows:

- If a variant is configured with the same custom and standard attributes, Gradle selects it.
- If no variant is configured with the custom attributes:
  - If a variant is configured with the same standard attributes, Gradle selects it.
  - If there is no variant with same standard attributes, the resolution fails.
- If no variant is configured with the same standard attributes, the resolution fails.
- If multiple variants are compatible, the resolution fails.

For example a Mock is published with the following variants:

```
"variants": [
{
  "name": "apiElements",
  "attributes": {
    "org.gradle.category": "library",
    "org.gradle.dependency.bundling": "external",
    "org.gradle.jvm.version": 11,
    "org.gradle.libraryelements": "jar",
    "org.gradle.usage": "java-api"
  },
},
{
  "name": "runtimeElements",
  "attributes": {
    "org.gradle.category": "library",
    "org.gradle.dependency.bundling": "external",
    "org.gradle.jvm.version": 11,
    "org.gradle.libraryelements": "jar",
    "org.gradle.usage": "java-runtime"
  },
},
{
  "name": "javadocElements",
  "attributes": {
    "org.gradle.category": "documentation",
    "org.gradle.dependency.bundling": "external",
    "org.gradle.docstype": "javadoc",
    "org.gradle.usage": "java-runtime"
  },
},
{
  "name": "sourcesElements",
```

(continues on next page)

(continued from previous page)

```
    "attributes": {
      "org.gradle.category": "documentation",
      "org.gradle.dependency.bundling": "external",
      "org.gradle.docstype": "sources",
      "org.gradle.usage": "java-runtime"
    },
  },
  {
    "name": "additionalElements",
    "attributes": {
      "com.microej.artifact.element": "additional-files"
    },
  {
    "name": "microejMockRip",
    "attributes": {
      "com.microej.veeport.artifact.usage": "default"
    }
  }
]
```

So if it is declared as dependency with a custom attribute not defined by the producer:

```
val myAttribute = Attribute.of("com.example.my-attribute", String::class.java)
dependencies {
    microejModule("org.example:my-mock:1.0.0") {
        attributes {
            attribute(myAttribute, "special-value")
        }
    }
}
```

the standard `runtimeElements` variant is selected and the Jar file of the Mock is fetched. However, if a Mock dependency is configured with a custom attribute and a standard variant:

```
val myAttribute = Attribute.of("com.example.my-attribute", String::class.java)
dependencies {
    microejModule("org.example:my-mock:1.0.0") {
        attributes {
            attribute(myAttribute, "special-value")
            attribute(LibraryElements.LIBRARY_ELEMENTS_ATTRIBUTE, objects.
↪named(LibraryElements::class.java, LibraryElements.OBJECTS))
        }
    }
}
```

the resolution fails because Gradle does not know which variant must be selected:

```
> Could not resolve all dependencies for configuration ':microejMockRuntimeClasspath'.
> Could not resolve org.example:my-mock:1.0.0.
  Required by:
      root project :
   > The consumer was configured to find a component, with the library elements 'objects', as well as attribute
↪'com.example.my-attribute' with value 'special-value'. However we cannot choose between the following variants␣
↪of org.example:my-mock:1.0.0:
       - additionalElements
       - javadocElements
```

(continues on next page)

(continued from previous page)

```
    - microejMockRip
    - sourcesElements
  All of them match the consumer attributes:
    - Variant 'additionalElements' capability 'org.example:my-mock:1.0.0':
      - Unmatched attributes:
        - Doesn't say anything about com.example.my-attribute (required 'special-value')
        - Doesn't say anything about its elements (required them with the library elements 'objects')
        - Provides attribute 'com.microej.artifact.element' with value 'additional-files' but the consumer didn't␣
↪ask for it
        - Provides release status but the consumer didn't ask for it
    - Variant 'javadocElements' capability 'org.example:my-mock:1.0.0':
      - Unmatched attributes:
        - Doesn't say anything about com.example.my-attribute (required 'special-value')
        - Doesn't say anything about its elements (required them with the library elements 'objects')
        - Provides documentation but the consumer didn't ask for it
        - Provides its dependencies declared externally but the consumer didn't ask for it
        - Provides javadocs but the consumer didn't ask for it
        - Provides release status but the consumer didn't ask for it
        - Provides runtime but the consumer didn't ask for it
    - Variant 'microejMockRip' capability 'org.example:my-mock:1.0.0':
      - Unmatched attributes:
        - Doesn't say anything about com.example.my-attribute (required 'special-value')
        - Doesn't say anything about its elements (required them with the library elements 'objects')
        - Provides attribute 'com.microej.veeport.artifact.usage' with value 'default' but the consumer didn't␣
↪ask for it
        - Provides release status but the consumer didn't ask for it
    - Variant 'sourcesElements' capability 'org.example:my-mock:1.0.0':
      - Unmatched attributes:
        - Doesn't say anything about com.example.my-attribute (required 'special-value')
        - Doesn't say anything about its elements (required them with the library elements 'objects')
        - Provides documentation but the consumer didn't ask for it
        - Provides its dependencies declared externally but the consumer didn't ask for it
        - Provides release status but the consumer didn't ask for it
        - Provides runtime but the consumer didn't ask for it
        - Provides sources but the consumer didn't ask for it
```

If no variant matches the consumer's requirements, it is possible to make a variant compatible by defining a compatibility rule.

Application Wrapper

The unification of the Application EntryPoints allows to build and run an Application regardless of the VEE provided by the user (a VEE Port or a Kernel). To ensure that an Application can run on different kinds of VEE, a wrapper class is generated. For example, if your Application EntryPoint is a class containing a `public static void main(String[args])` method:

```
package com.mycompany;

public class MyClass {
   public static void main(String[] args) {
     System.out.println("Hello World!");
   }
}
```

A wrapper class implementing the `ej.kf.FeatureEntryPoint` interface is created:

```
package com.mycompany;

import ej.kf.FeatureEntryPoint;

public class MyClassWrapper implements FeatureEntryPoint {
    @Override
    public void start() {
      com.mycompany.MyClass.main(new String[0]);
    }

    @Override
    public void stop() {
    }
}
```

Then, the execution of the Application depends on the VEE provided in the `build.gradle.kts` file of your project:

- If a VEE Port or a Mono-Sandbox Kernel has been provided, the `MyClass` main class of the Application is executed.
- If a Multi-Sandbox Kernel has been provided, the Application is executed as a Feature of the Kernel.

If your Application requires advanced features, it is also possible to *use a FeatureEntryPoint class as Application EntryPoint*. In that case the created wrapper class is a class containing a `public static void main(String[args])` method that calls the `start()` method of your Application EntryPoint class:

```
package com.mycompany;

public class MyClassWrapper {
    public static void main(String[] args) {
      new com.mycompany.MyClass.start();
    }
}
```

In addition to the wrapper class, the following resource files are generated:

- The Feature Definition Files (`feature.kf` , `feature.cert`), to use your Application as a Sandboxed Application.
- The Kernel Definition Files (`kernel.kf` , `kernel.cert`), to use your Application as a Kernel Application.

SDK 6 1.5.0 and above

SDK 6 1.4.0 and below

Using SDK 6 1.5.0 and above, properties can be modified in the `build.gradle.kts` file of your project, refer to the *Kernel and Features Declaration* section for more information.

Using SDK 6 1.4.0 and below, you can also define your own Feature and Kernel Definition files if needed, refer to the *Feature Definition Files* section for more information.

Cheat Sheet

Download the Cheat Sheet for SDK 6

Java Lambdas

To support *Java lambdas* in the code of an Application, the SDK uses a forked version of RetroLambda.

RetroLambda is not compatible with the Java versions *required by the SDK*, so a Java 8 toolchain is internally used by the SDK to manage Java lambdas.

By default Gradle will use the Java toolchain available locally, but it can be configured to automatically download one by applying the Foojay Toolchains Plugin in the `settings.gradle.kts` file of your project:

```
plugins {
    id("org.gradle.toolchains.foojay-resolver-convention").version("0.9.0")
}
```

This settings plugin downloads the first Java toolchain matching the specified requirements that it finds among the JVM Vendors recognized by Gradle and unzips it in the `<USER_HOME>/.gradle/.jdks` folder, so you do not have to install one manually.

Note: A Java 8 toolchain is only required if the Java lambdas are enabled on your Application project. If it is not the case, you do not need to apply the `Foojay Toolchains Plugin`.

Applied on a MicroEJ project, the plugin downloads the first Java 8 toolchain found, but it is possible to specify a Vendor by configuring the Java toolchain in the `build.gradle.kts` file of your project if needed:

```
tasks.processMainLambdas.configure {
    javaLauncher.set(javaToolchains.launcherFor {
        languageVersion = JavaLanguageVersion.of(8)
        vendor = JvmVendorSpec.ADOPTIUM
    })
}
```

Note: For each source set defined in your project, a `processXXXLambdas` task exists. To modify the Java toolchain, you must configure the task corresponding to the source set containing Java lambdas. For example, if they are used in the `src/main/java` source set, the `processMainLambdas` task must be configured.

VEE transitivity

Starting from SDK 6 `1.4.0`, it is possible to enable the transitive resolution of a *VEE Port*. This allows to use the transitive dependencies of the provided VEE Port to compile and build the project.

VEE Port transitivity

When a VEE Port is resolved transitively, its transitive dependencies are added to the compile classpath and runtime classpath of the project depending on the configuration used to define the Libraries in the VEE Port (microejPack , api , implementation):

- The Libraries defined with microejPack and api and their transitive dependencies are added to the compile classpath and runtime classpath of the project.
- The Libraries defined with implementation and their transitive dependencies are added to the runtime classpath of the project.

For example, if your VEE Port provides the following Libraries:

```
dependencies {
  implementation("org.example:lib1:1.0.0")
  microejPack("org.example:lib2:1.0.0")
  api("org.example:lib3:1.0.0")
}
```

the lib1 library will be added to the runtime classpath of your project, and the lib2 and lib3 libraries will be added to both the compile classpath and the runtime classpath.

Note: In a VEE Port project, the microejPack configuration can be used to define an SDK 5 Pack or an SDK 6 Library. The api and implementation configurations are used to define SDK 6 Libraries only.

If the provided VEE Port is used only for *testsuites*, the Libraries provided by the VEE Port are used to build the runtime classpath of the testsuites, not to compile the project.

Warning: It is important to note that a VEE Port built with SDK 6 1.3.0 minimum is required to use transitivity, the following VEE Ports are not supported:

- VEE Ports built with SDK 6 1.2.0 or older
- VEE Ports built with SDK 5

It is highly recommended to update your VEE Port to SDK 6 1.3.0 minimum if needed, as the feature will be enabled by default in the next SDK 6 major version. This limitation does not concern VEE Ports available locally *in a directory* or *as an archive*.

Foundation Libraries Mapping

When an operation is performed on an Application project (run on Simulator, build an Executable, …), some checks and mappings are done on its Libraries.

Check Resolution Conflicts

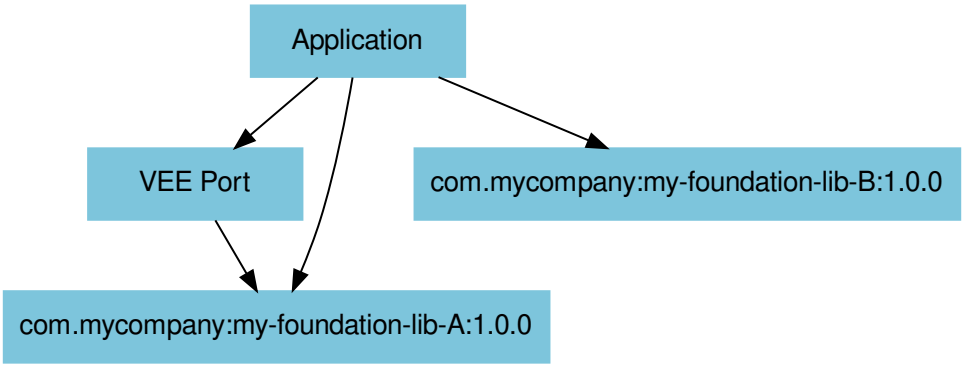
During the resolution of the compile and runtime classpaths, the following conditions are checked:

- no dependency with 2 incompatible versions (different major version).
- no direct dependency has a transitive dependency with a higher minor version.

Refer to *Manage Resolution Conflicts* for more information.

Check Libraries Existence in the VEE Port

Before launching an action on an Application project which uses a VEE Port or a Virtual Device (for example runOnSimulator or buildExecutable), the SDK checks that all the Foundation Libraries found in the calculated runtime classpath exist in the VEE Port or Virtual Device. For example, in the following case:

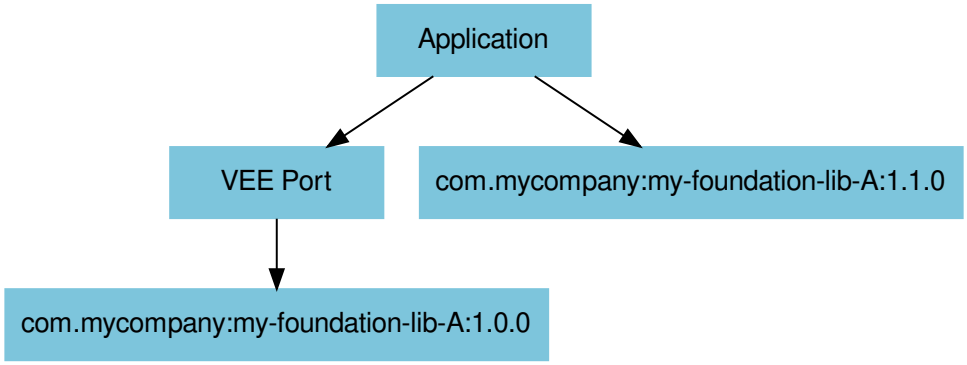


the build fails with the following message:

```
Unable to locate the 'my-foundation-lib-B' Foundation Library in the VEE Port. Make sure that the Library or␣
↪the related Pack is declared as a dependency in the VEE Port project.
```

Check Libraries Compatibility in the VEE Port

The SDK also checks the compatibility between the resolved Foundation Libraries dependencies and the ones available in the VEE Port of the Virtual Device. If the version available in the VEE has a lower minor version, the run fails.



the build fails with the following message:

```
com.mycompany:my-foundation-lib-A : the version 1.1.0 of the Library is not compatible with the version 1.0.0␣
↪provided by the VEE Port
```

3.4.28 Changelog

[1.5.0] - 2025-12-18

Added

- Support the build of Module Repository.
- Add Consistency checker on the dependencies of a Module Repository project.
- Support disabling checkers for a single dependency of a Module Repository project.
- Support excluding artifacts of a dependency in a Module Repository.
- Add `withDependenciesIvyDescriptor` method in `microej` block to include Ivy descriptors in a Module Repository.
- Add a Gradle script `module-repository.gradle.kts` in the archive file of a Module Repository to simplify its configuration in a project.
- Support embedding JDKs in the generated Virtual Device of an Application project.
- Support Stack Trace Reader for Multi-Sandboxed Applications.
- Always build executable when running a testsuite on a device.
- Support new Foundation Libraries format in Artifact Checker.
- Add configuration properties to define Application SSL Certificate information.
- Add `applicationName` and `applicationVersion` properties in `microej` block to configure KF metadata.
- Add a warning message when the user's license is about to expire.
- Consider the Application Options defined in the Application project when building a Feature.
- Add the `microejtool.deploy.name` property to change the deployment script used by the `runOnDevice` task.

- Check the license from the SDK instead of using the `license-checker` tool.

Changed

- Publish `README.md`, `CHANGELOG.md`, `LICENSE.txt`, `ASSEMBLY_EXCEPTION.txt` and `RELEASE_NOTES.md` files as variants to improve their resolution.
- Improve the resolution of dependencies by using custom attribute only instead of `LibraryElements` standard attributes.
- Change the default value of the `name` property in `.kf` files to the module name instead of the entrypoint class name.
- Display a clear error message when building a Virtual Device with a Runtime Environment project with no version.
- Improve error message when a Sandboxed Application is built with a Kernel with a different plugin version in a multi-project.
- Remove dependencies of the published Virtual Device variant of an Application to not fetch them when using transitivity on `microejVee` dependencies.
- Update variants of a VEE Port to improve its resolution.
- Update `retrolambda` dependency to `2.6.2`.

Fixed

- Runtime Environment does not include sources of dependencies included with `includeBuild`.
- Application WPK does not contain all libraries provided as included build when Libraries have the same name.
- Mock RIP does not include runtime dependencies.
- Library projects build fails when Ivy descriptor file generation is disabled.
- Hide javadoc errors of shrunk APIs when building a Kernel or a Runtime Environment.
- Errors caused by the missing annotations when building a Kernel with EDC 1.3.6+.
- Snapshot versions (`-RC`) are not suffixed with timestamp in `.kf` files.
- Report initializationErrors and executionErrors when exception is thrown in `@BeforeClass` or `@AfterClass` in JUnit tests.
- Report test harness execution error as test failure.
- Foundation Library not retrieving its declared mock when running tests on simulator.
- Eclipse and VS Code cannot resolve classes from runtime environment when defined as project dependency.
- Dependency fetching issues because Addon Processors dependencies are not loaded in a dedicated classloader.

[1.4.0] - 2025-09-18

Added

- Support the addition of LSC files in a Library to extend the Linking process.
- Add the plugin `com.microej.gradle.library` to build an Add-On Library or a Foundation Library and deprecate `com.microej.gradle.addon-library` .
- Fetch the transitive dependencies of a `microejVee` dependency (enabled with the project property `feature.vee.transitivity.enabled`).
- Add the support for the system property `microej.launch.jvmargs` to specify JVM arguments to the JVM running Ant scripts.
- Build and publish the aggregated javadoc of a VEE Port (SDK 6 Libraries only).

Changed

- Remove useless usage of `useJUnitJupiter` when configuring a testsuite.
- Replace usage of deprecated Gradle APIs to prepare the support for Gradle 9.
- Update the stackTraceReader to version `3.2.0` to support Managed C stack traces.

Fixed

- Fail to publish to Maven Local repository.
- No publication of the Feature file when no change in the project.
- Virtual Device options are not loaded correctly.
- Test log in case of no failure or success tag encountered.
- Do not rebuild Kernel Executable if not necessary when building a Feature.
- Properties defined in partial KF file in Application projects are not considered.
- `microej.testsuite.retry.count` property not always taken into account.
- Retried tests not correctly reported in HTML report.
- Eclipse reports `java.lang.invoke.MethodHandles` not found in build path when lambdas are enabled.
- `ShrinkRuntimeEnvironmentTask` fails when kernel is in a multiproject and has an `includeBuild` dependency.

[1.3.1] - 2025-07-11

Changed

- Improve the name of the mocks jar files in the Virtual Device (use `[group]-[module]-[version].jar`).

Fixed

- Wrong classpath used to generate Application Wrapper class which could make the build fail when declaring a dependency with `compileOnly`.
- Update `junit-test-engine` to fix testsuite using serial to socket and custom success/failure tag with JUnit.
- Remove duplicated entries in the test classpath.
- Missing white space in error message when no Architecture is provided.
- Pass test engine classpath based on `junit-test-engine` dependencies only to prevent unexpected classpath entries.
- Improve the error message when the `buildVirtualDevice` task fails because two mocks depend on a local jar with the same name.
- Fix installation of dependencies in the Virtual Device when a project defined in the same multi-project depends on it with `includeBuild`.

[1.3.0] - 2025-06-19

Added

- Add license information in published POM file.
- Add a custom attribute to the variants of an Application, a VEE Port, a Mock and a Runtime Environment to improve the resolution of MicroEJ dependencies.
- Add the `stackTraceReader` task to the Library plugin.
- Support the declaration of Libraries with the `microejPack` configuration in a VEE Port.
- Add `microejComponents` task that prints MicroEJ commercial components dependency list in the console.

Changed

- Improve error message when the dependency defined with the `microejApplication` configuration is not an Application.
- Clarify the message when a Library is not found in the VEE Port.
- Add tests on `includeTestsMatching` in tests filter.

Fixed

- Support build of WPK and Virtual Device when an include build is used for a dependent library.
- Failure when running a KF Testsuite defined in a subproject of a multi-project.
- Error message when the Application provided as VEE dependency is not a Kernel.
- Testsuite not executed again after switching test target (Simulator or Device).
- Build of Runtime Environment when it depends on an Addon Library project.
- Load of a VEE Port declared as a project dependency when its project does not use the same version as the Application.

- Application options set with `System.setProperty()` not taken into account.
- Null Analysis check failure in Runtime Environment projects by adding back `package-info.java` files.
- Allow verbose mode on `buildFeature` task by setting system property `microej.option.execution.verbose` to `true`.
- Update `junit-test-engine` to fix test duration and stack traces.
- Compile on JDK instead of on dependencies only when the use of Java Lambdas is enabled.
- VEE Port dropins files not considered when related to RIP packs.
- Compilation errors when building a Runtime Environment with EDC 1.3.6+.

[1.2.0] - 2025-04-09

Added

- Allow to define the JVM used to launch the VD in the `JAVA_HOME` environment variable.
- Add POM and Gradle module files in generated Ivy descriptor file to allow to fetch them from a SDK 5 offline repository.
- Remove the JDK from the classpath in the Eclipse and VS Code (when using RedHat's Java Extension) IDE.
- Support the declaration of Mocks in an Application and in a Library.
- Support Kernel Application, Runtime Environment and VEE Port projects in Ivy descriptor.
- Publish testsuite reports if tests have been executed.
- Support Libraries as dependencies of a VEE Port.

Changed

- Improve the error message when publishing the VEE Runtime Library.
- Improve the error message when project depends on a Kernel whose Virtual Device or Executable is missing.
- Display correctly the error message when the launch of a VEE Port script fails.
- Load the VEE used to build the project and the VEE used for tests in different folders.

Fixed

- Virtual Device launch fails on Windows with a custom Java absolute path.
- Kernel Javadoc contains Kernel sources instead of exposed APIs.
- Add only the JAR file to the variant of a Foundation Library published with MMM, to be compatible with the SPDX Gradle plugin.
- Print test execution logs stdout instead of stderr.
- Mock not installed in the VEE Port built on the fly when running an Application.
- Make lambdas processing tasks fail when lambda processing fails.

[1.1.0] - 2025-02-28

Added

- Allow to execute a KF testsuite.
- Add a dedicated task to run the Stack Trace Reader.
- Allow to define VEE used for tests in the test classpath only.

Removed

- Removed deprecated `com.microej.gradle.runtime-api` and `com.microej.gradle.j2se-library` plugins.

Changed

- Improve testsuite report.
- Always generate `fs.properties` file when building a VEE Port to not make the Pack FS fail.

Fixed

- Execute testsuite when there is no testsuite called `test` declared.
- VEE Port load issue with Gradle greater than `8.10`.
- Tool declared as dependency not installed in the augmented built Virtual Device of the Application project.
- Wrong location of the generated Application object file displayed when executing the `buildApplicationObjectFile` task.
- Error message when a local Kernel whose Executable or Virtual Device does not exist is provided.
- User defined source sets are ignored by ADP.
- Error message when a Virtual Device is used to run tests on a device.
- RC versions not declared with the right version pattern in the generated Ivy descriptor.
- ADP not found when JUnit dependency is defined in custom testsuite.
- Error message when a VEE Project dependency is defined and the Gradle configuration-on-demand feature is enabled.
- VEE Port scripts execution fails when the User Home folder contains a space.
- Failure when running tests defined in the default sources folder of a custom testsuite.
- Testsuite fails after upgrading from 0.15.0 to 1.0.0 without cleaning.

[1.0.0] - 2024-12-18

Added

- Allow to define the Artifact Checker root directory.

Changed

- Improve Gradle version error message to explain how to solve the issue.
- Improve SDK EULA acceptation error message.
- Reorganize Virtual Device structure to reduce the number of files at the root folder.

Fixed

- Fix compilation error during VEE Port build when fetching JDT Core library from Maven Central.
- Exclude `.gitkeep` files from Java sources.
- Fix Application Wrapper class generation when Main class has no package.
- Do not force custom tool tasks to set an Application Entry Point.
- Include local JAR dependencies in classpaths.
- Fix error when building a WPK with a jar in a local repository.
- Support Addon Libraries in Runtime Environments.
- Wrong classpath when running an Application or a testsuite on device.

[0.20.0] - 2024-10-23

Added

- Allow to build a Feature from a WPK and a Kernel.
- Allow to filter the BSP files included in a published VEE Port.
- Check that the MicroEJ Runtime Library is not in the BSP before publishing a VEE Port.
- Support the import of a VEE Port as an included build.

Changed

- Fail with clear error message when trying to build an Executable and provided Runtime API conflicts with other dependency.
- Hide `checkModule` internal task.
- Use the original BSP when depending on a local VEE Port project in BSP Full Connection mode.
- Set Java source to 1.8 to be compatible with new IDEs versions, but keep 1.7 as the target version.
- Upgrade Gradle minimal supported version to `8.6`.
- Enable MicroEJ Java H module by default when building a VEE Port.

- Deploy Application object files in BSP only if their content changed.
- Prevent flaky tests from causing build failure.
- Use “Runtime Environment” term instead of “Runtime API”.
- Use `Java SE` term instead of `J2SE`.
- Use version 2.2.0 of the microej-licenses library to check with the new SDK EULA 3.1-C.
- Do not produce Virtual Device by default during Application build.

Fixed

- Wrong path used for custom test properties files.
- Fix VEE Port build failure when using Architecture with deployment hook for Keil.
- Use the generated kernel.kf file when running an Application on a Multi-Sandbox VEE Port with the simulator.
- Generate again the kernel.kf file when the project version has changed.
- Failure when calling the local deploy tool because of missing generated KF file in the classpath.
- Setting `com.microej.runtime.capability=multi` does generate a Multi-Sandbox VEE Port when using an Architecture 8.1 or higher.

[0.19.0] - 2024-09-13

Added

- Allow to define properties of a testsuite project in the `local.properties` file.
- Display clear error message when trying to build a Feature with a Virtual Device built with a Mono-Sandbox VEE Port.

Changed

- Application options must now add `microej.option.*` prefix to be defined as System properties.
- Load VEE Port MicroUI configuration files from the `extensions/microui` folder instead of the `microui` folder.

Fixed

- Project not configured to build with Java 1.7 when runtimeClasspath configuration has already been resolved.
- Allow to produce feature files during the build in a multi-project with several feature projects.
- Use the provided Runtime API jar if it contains KF to compile the Wrapper class instead of fetching KF to avoid dependency resolution error with Offline repositories.
- Fail with readable error message when building a Runtime API with no Kernel API declared.
- VG Pack 1.6.0+ cannot be used for a VEE Port because it provides JAR artifacts on its default configuration (besides the Pack RIP).
- Use the Runtime API provided by the Kernel to build an augmented Virtual Device.

[0.18.0] - 2024-08-22

Added

- Allow to build a VEE Port.
- Retry tests when they fail to avoid flaky tests (mainly due to license check) to fail the whole build.
- Support build incremental when using a published VEE Port with Full BSP Connection.
- Add the plugin `com.microej.gradle.runtime-api` to build a Runtime API.
- Allow to enable/disable the publication of the Ivy descriptor.
- Allow to run dependent applications on simulator (declared with `microejApplication`).

Fixed

- Fix publication to add dependencies defined with MicroEJ configurations to the `.module` and `.pom` files.
- Comment the `stop` method of the generated Wrapper class to prevent potential Sonar issue.
- Fix MicroEJ Test Engine compatibility with Gradle 8.6 and higher.
- Allow to run a Virtual Device via its launchers on a JDK version higher than 11.
- Fix build failure on multi-project with several applications depending on a kernel as project dependency.
- Some projects are not configured to be built with Java 1.7 when imported in Eclipse.
- Fix feature not found when launching a Virtual Device with installed applications.

[0.17.0] - 2024-05-30

Added

- Allow to add tools to a Virtual Device.
- Allow to fetch Runtime APIs with the `microejRuntimeApi` configuration.

Changed

- Merge the `loadXXXConfiguration` tasks with their matching task.
- Split `buildExecutable` in 2 tasks to support incremental build of the `microejapp.o` file.
- Make FeatureEntryPoint take priority over main method when generating the Application entryPoint wrapper.

Fixed

- Re-Generate the Application entrypoint wrapper if the entrypoint class is modified, if the applicationEntryPoint property is changed or if the resources changed.
- Simplify Ant classpath when executing an Ant script to avoid too long classpath and support multiple Windows drives.
- Append Applications provided with the microejApplication configuration to a Virtual Device.
- Make sure to always generate the Kernel certificate if it does not exist.
- Remove deprecated APIs used to generate Application certificates.
- Log filter in Ant scripts.
- Print last relevant logs as exception message when Ant script execution fails.
- Fix failing Javadoc generation when using EDC 1.3.6 and Null Analysis annotations.
- Make the generated Wrapper Feature class call the main method of the Application with an empty array as parameter instead of null.
- Generate Application entrypoint wrapper if the entrypoint class extends a class implementing the FeatureEntryPoint interface.
- Do not embed generated KF files in Application JAR to avoid switching in KF mode when executing an Application on a VEE Port.
- Fix configurations used to fetch Kernels to avoid NPE during build when a project is provided as dependency.
- Set ADP generated folders in the sourcesets to be detected by the IDEs.
- Make sure to copy the Assembly file in the BSP when it is generated.

[0.16.0] - 2024-03-18

Added

- Unify Application entryPoints.
- Allow to append an Application to a Virtual Device.
- Make execTool task available in library projects.
- Add a check on the dependencies versions format to reduce the risk of mistakes.
- Generate the Feature Definition Files and Kernel Definition Files of an Application.

Changed

- Rename applicationMainClass property to applicationEntryPoint .
- Hide MicroEJ internal tasks.
- Set the microej-testsuite.properties file as output of the loadTestApplicationConfiguration task instead of its parent folder.

Fixed

- Load Kernel and Main Application properties when starting the Application on the Simulator.
- Can run more than one simulator on the same Application on IntelliJ/Windows.
- Fix Addon-Processor not reexecuted when dependencies are updated.
- Do not execute tests when building the Executable of an Application.
- MicroEJ Test Engine compatibility with Gradle 8.6.
- Do not force the compilation of J2SE tests classes in Java 7.
- Generate Jar file when building a project containing all MicroEJ artifacts.
- Clean the working files before creating the WPK file to prevent failure if the task is not UP-TO-DATE.

[0.15.0] - 2024-01-26

Added

- Unify `microejVeePort` and `microejKernel` configurations into `microejVee`.
- Add verification of dependencies checksums during build.
- Add the plugin `com.microej.gradle.mock` to build a Mock.
- Mention the system property to accept SDK EULA in error message.

Changed

- Task `:execTool` looks for a script named after the argument NAME with the following patterns in that order: NAME, NAME.microejTool, NAME.microejLaunch.
- Align the behavior of the `:buildFeature` task with the `localDeploymentSocket.microejLaunch` script.
  - output files are derived after “application” instead of “feature” (for example “application.fo”).
  - the application.main.class is set to the entryPoint defined in the .kf of the application.

Fixed

- Upgrade to junit-test-engine 0.2.2 to fix failing tests using fonts.
- Handle Security Manager removal from JDK 18+ when executing MicroEJ VEE scripts.
- Support all MicroEJ VEE (VEE Ports & Kernel) for the task `:execTool`.

Removed

- Remove support of dropIns folder for MicroEJ VEE (VEE Port or Kernel) selection.

[0.14.0] - 2024-01-03

Added

- Add Jenkinsfile files to build and test with a JDK 17 and a JDK 21 (LTS versions).

Changed

- Do not build/publish an Executable or a Feature by default and add the `produceExecutableDuringBuild()` and `produceFeatureDuringBuild()` methods to build them if needed.
- Set group and version for all projects, including the root project, in order to generate correctly the release tag.
- Use version 2.1.0 of the microej-licenses library to check with the new SDK EULA 3.1-B.

Fixed

- Fix the override behavior of the Application main class that was not consistent when `-Dapplication.main.class` is used.
- Fix the Custom Ant Logger to display build errors without having to enable the verbose mode.
- Follow Gradle recommendation on resolvable and consumable configurations.
- Move the Custom Logger to a dedicated module and use its jar instead of fetching the plugin when executing VEE Port scripts to fix the tests failure during a release.
- Set Java Compiler encoding to UTF-8.
- Set Java Compliance level to 1.7 in JavaPluginExtension to fix the `Cannot find the class file for java.lang.invoke.MethodHandles` error when opening a Gradle project in Eclipse.
- Fix wrong generated Virtual Device of an Application when the VEE Port changed.
- Bump source level for javadoc task to 1.8 to support JDK 21.
- Fix classpaths when using a Virtual Device to remove warnings about kf files not found.
- Make sure to close all opened streams.
- Fix `Wrong java/lang/Object` error when running an Application on the Simulator with a local repository.
- Fix `No .kf file found for this feature classpath` error message in logs when running VD with launcher script.

[0.13.0] - 2023-11-10

Added

- Add a check on EULA acceptation when using the MicroEJ Gradle plugin.
- Automatically publish the ASSEMBLY_EXCEPTION.txt file if it exists at the root of the project.
- Allow to publish the Feature file of an Application.
- Add `:execTool` task to execute Stack Trace Reader and Code Coverage Analyzer Tools provided by the selected VEE Port or Kernel.

Fixed

- Fix warning during compilation because of non-existing file or folder (`incorrect classpath: C:\\Users\\user\\...\myProject\\build\\resources\\main`).
- Fix warning in SOAR when building an Executable with Architecture 8.0.0 (`[M59] - Classpath file [C:\Users\user\...\myProject\build\resources\main] does not exist`).
- Enable Ant verbose mode for VEE Port scripts when Gradle debug log level is enabled.
- Fix the build of a Feature when the provided Virtual Device does not contain the `dynamicFeatureLink.microejLaunch` build script (Virtual Device built with SDK 5).

[0.12.1] - 2023-10-16

Fixed

- Fix the issue with the `microejKernel` configuration that prevented IDEs from loading a project.

[0.12.0] - 2023-10-13

Added

- Allow to publish the Virtual Device of an Application.
- Allow to fetch a Virtual Device and an Executable with the `microejKernel` configuration.

Changed

- Add README, CHANGELOG and License files as publication artifacts in the generated ivy.xml file.
- Publish test report in Jenkins job.
- Set `deploy.bsp.microejscript` property to `true` by default to build the executable.
- Publish the Executable file as a variant.
- Rename the `kernelFile` property to `kernelExecutableFile` .
- Use File dependency instead of the `veePortPath` property to load a local VEE Port.
- Use File dependency instead of the `kernelExecutableFile` property to load a local Kernel Executable.

Fixed

- Fix unexpected fetch of the transitive dependencies of a VEE Port dependency (microejVeePort configuration).
- Fix System properties defined in gradle.properties are ignored.
- Fix VEE Port launcher: temporary configuration file could prevent to launch a second time.
- Remove usage of deprecated API Project.getBuildDir().
- Fix the message when no executable are found by the runOnDevice task.
- Fix Executable not updated after a project change and a call to the buildExecutable task.
- Fix wrong order of tests classes and resources folder in the test classpath.
- Call VEE Port Ant script from a separate temporary directory to satisfy MicroEJ Architecture. This fixes spurious HIL timeouts when calling the runOnSimulator task.
- Fix Java process still running when Simulator is interrupted.
- Fix missing Nashorn dependencies when running a testsuite and when launching the launcher scripts to make it work with JDK 17 and higher.

[0.11.1] - 2023-09-22

Fixed

- Fix usage of a SNAPSHOT version of the junit-test-engine dependency.

[0.11.0] - 2023-09-22

Changed

- Use Gradle standard mechanism to support Multi-VEEPort instead of relying on an in-house feature.

[0.10.0] - 2023-09-13

Added

- Add a task runOnDevice to run the Executable on a Device.
- Support all JDK LTS versions higher or equals to version 11.

Fixed

- Allow to build a Feature file of an Application with a Virtual Device.
- Fix javadoc failure when the project contains a JDK class.

[0.9.0] - 2023-09-01

Added

- Allow to depend on local Application project (dependency with `microejApplication` configuration).
- Implement properties loading chain.
- Add launcher scripts to the Virtual Device.

Changed

- Move the `vd-init.xml` script in the template file instead of hardcoding it in the class.
- Remove the Application properties from `options/application.properties` file and rename file to `target.properties` in Virtual Device.
- Merge `veePortFiles` and `veePortDirs` properties into the `veePortPaths` property.
- Add missing Javadoc and clean the project.

Fixed

- Fix resources generated by Addon Processors of type FolderKind.MainResources not processed.
- Fix root path used for relative VEE port path: use the project root directory.
- Fix the content of a WPK to remove the Foundation Libraries.
- Make sure `.a` and `.o` files of an Application are correct by always executing the `buildExecutable` task.

[0.8.0] - 2023-07-13

Added

- Allow to build the Virtual Device of an Application.
- Add checks to ensure that a Virtual Device can be used or not depending on the called task.
- Allow to build the Feature binary file of an Application.

Fixed

- Add the Application properties defined in the configuration folder to the WPK file.

[0.7.0] - 2023-06-26

Added

- Add Standard Java Library plugin (com.microej.gradle.j2se-library).
- Rename com.microej.gradle.library plugin to com.microej.gradle.addon-library .

Changed

- Unbind the checkModule task from the build task.
- Use version 0.1.1 of the MicroEJ JUnit Test Engine to fix error when test classes are not in a package.
- Use version 2.0.0 of the microej-licenses library to check with the new authorized licenses.

[0.6.0] - 2023-05-30

Added

- Allow to publish WPK file artifact.
- Allow to publish files generated by the buildExecutable task.
- Allow to define multiple testsuites in different environments (sim or J2SE).
- Allow to define a testsuite for tests on device.

Changed

- Use Ivy descriptor content to know if a dependency is a Foundation Library or an Addon Processor Library.
- Optimize the loadVeePort task to reduce the time to load a VEE Port.
- Use a smaller VEE Port as dependency in tests to reduce the time to build.
- Remove JPF support.
- Check that the given file/directory is a VEE Port.
- Move Application properties to configuration folder instead of src/main/resources .
- Clean the Jenkins workspace after a successful build.
- Improve the checker on changelog files to support “-SNAPSHOT” suffix and “Unreleased” label.
- Remove the debugOnSimulator task and use a property to run an Application in debug mode.

Fixed

- Fix multiple VEE Ports error message in `loadConfiguration` task.
- Fix connection to a debugger and debug.port property.
- Fix StackOverflow error when building a project with cyclic dependencies.

[0.5.0] - 2023-03-24

Added

- Add Xlint checking.
- Add verification of using java 11 by user's project.
- Allow to build the Executable file of an Application.
- Allow to build the WPK file of an Application.
- Allow to define multiple VEE Ports.
- Check that the project uses at least Gradle 8.0.
- Add more tests on topological order in the Application classpath.

Changed

- Make the plugin compatible with Gradle `8.0`.

[0.4.0] - 2023-01-27

Added

- Apply the Java Library Plugin to user's project.
- Allow to load a VEE Port by dropping it in the `dropIns` folder.

Changed

- Optimise memory used by project.
- Remove the `runArtifactChecker` property, the Artifact Checker task must be executed explicitly.
- Hide compilation warnings in the adp and compileJava tasks.

Fixed

- Disable the warning on non-compatible version for Maven client.
- Fix loading new dependency when the `build.gradle.kts` file is updated.
- Fix too long classpath error when running the simulator on Windows.

[0.3.0] - 2022-12-09

Added

- Add feature to avoid loading the VEE Port when there is no test.
- Add the auto assembling project for `runOnSimulator` and `debugOnSimulator` tasks.
- Add the opportunity disable custom conflict resolution rules.
- Add the plugin `com.microej.gradle.library` to build an Addon Library.
- Generate and publish the Java sources jar.
- Generate and publish the Javadoc jar.
- Publish `README.md`, `CHANGELOG.md` and `LICENSE.txt` files if they exist in the project.
- Suffix version with timestamp when it ends with "-RC".
- Make the build fail if a direct dependency is resolved with a higher minor version than the one declared.
- Add the `checkModule` task to check compliance of the module with MicroEJ rules.
- Add the execution of tests on the simulator.
- Add support for Mac M1.
- Build the plugin in Java 11.
- Add test to ensure that the dependencies are topologically sorted.

Changed

- Remove automatic version conversion.
- Rename the Application plugin to `com.microej.gradle.application`.
- Change the publication plugin to publish Maven modules instead of Ivy modules.
- Use Ant Java API to launch the simulator to avoid requiring an Ant installation.
- Rename the `runOnSim` and `debugOnSim` tasks to `runOnSimulator` and `debugOnSimulator`.
- Use JDT compiler instead of javac.
- Isolate functional tests to keep a quick build.

Fixed

- VEE Port not reloaded when referenced by `veePortDirPath` and the VEE Port source folder is updated.
- Set Java source and target version to be recognized by IDEs.
- Make `processResources` task implicitly depend on ADP task to fix failures during `runOnSimulator`.

[0.2.0] - 2022-05-17

Changed

- Make the build fails when an ADP raises errors.
- Convert build scripts from `Groovy` to `Kotlin`.

[0.1.0] - 2022-05-03

Added

- Add the capability to load the platform from dependencies.
- Add the task `debugOnSim` to execute the application in debug mode in the simulator.
- Publish the sources jar of the plugin.

Fixed

- Extract ADP classpath JAR files into OS temp dir to avoid error on cleaning because of locks.

3.4.29 Migration Notes

Note: When updating the plugin version, it is recommended to perform a `clean` on your project(s). For multi-projects, run the `clean` command on the root project.

From 1.3.1 to 1.4.0

The plugin `com.microej.gradle.addon-library` has been renamed to `com.microej.gradle.library`. The previous name is still supported but it is deprecated and will be removed in the next major version. It is recommended to update your project now.

From 1.0.0 to 1.1.0

The following plugins have been removed:

- plugin `com.microej.gradle.j2se-library` , replaced by `com.microej.gradle.jse-library` .
- plugin `com.microej.gradle.runtime-api` , replaced by `com.microej.gradle.runtime-environment` .

From 0.19.0 to 0.20.0

This section applies if SDK 6 `0.20.0` is used on a project that was created using SDK 6 `0.19.0` or lower.

Minimum Gradle version

The minimum required version of Gradle is now `8.6.` . Upgrade the Gradle Wrapper version of your project.

New SDK EULA

A new SDK EULA must be approved before using the SDK (version `3.1-C`). Refer to *SDK EULA Acceptation* page to learn how to approve it.

Plugins and Configurations renaming

The following plugins and configurations have been renamed:

- plugin `com.microej.gradle.j2se-library` to `com.microej.gradle.jse-library` .
- plugin `com.microej.gradle.runtime-api` to `com.microej.gradle.runtime-environment` .
- configuration `microejRuntimeApi` to `microejRuntimeEnvironment` .

The previous plugins and configurations are still supported but are deprecated and will be removed in the next major version. It is recommended to update your project now.

Virtual Device Build

The Virtual Device of an Application is not built by default anymore when calling the `build` task. You can call the `buildVirtualDevice` task to build it, or add the following code in your `build.gradle.kts` file to continue to build it when calling the `build` task:

```
microej {
  produceVirtualDeviceDuringBuild()
}
```

From 0.15.0 to 0.16.0

This section applies if SDK 6 `0.16.0` is used on a project that was created using SDK 6 `0.15.0` or lower.

Unification of Application EntryPoint

The creation of a Sandboxed Application and a Standalone Application have been unified. To create an Application, the following steps must be done:

- Create the Java main class in the `src/main/java` folder.
- Define the property `applicationEntryPoint` in the `microej` configuration block of the `build.gradle.kts` file. It must be set to the Full Qualified Name of the Application main class:

```
microej {
   applicationEntryPoint = "com.mycompany.Main"
}
```

- Define a VEE (VEE Port or Kernel) by declaring a dependency in the `build.gradle.kts` file:

```
dependencies {
   microejVee("com.mycompany:myVee:1.0.0")
}
```

If your Application requires the use of advanced features, you must create a Feature class, for example:

```
package com.mycompany;

import ej.kf.FeatureEntryPoint;

public class MyFeature implements FeatureEntryPoint {

  @Override
  public void start() {
    System.out.println("Feature MyFeature started!");
  }

  @Override
  public void stop() {
    System.out.println("Feature MyFeature stopped!");
  }
}
```

and set the property `applicationEntryPoint` to the Full Qualified Name of the Feature class:

```
microej {
   applicationEntryPoint = "com.mycompany.MyFeature"
}
```

Testsuite Execution

When upgrading from `0.15.0` to an higher version (up to `1.0.0`), you may encounter the following error when executing a testsuite:

```
Preparing the execution of tests with the MicroEJ JUnit test engine

org.junit.platform.commons.JUnitException: TestEngine with ID 'microej-junit-test-engine' failed to execute tests
  at org.junit.platform.launcher.core.EngineExecutionOrchestrator.execute(EngineExecutionOrchestrator.java:113)
  at org.junit.platform.launcher.core.EngineExecutionOrchestrator.execute(EngineExecutionOrchestrator.java:88)
  ...
Caused by: com.microej.testengine.TestEngineException: More than one VEE Port have been provided to run the testsuite
  at com.microej.testengine.MicroejTestEngine.execute(MicroejTestEngine.java:203)
  at org.junit.platform.launcher.core.EngineExecutionOrchestrator.execute(EngineExecutionOrchestrator.java:107)
  ... 29 more
```

Execute the `clean` task before executing the testsuite to solve this issue.

From 0.14.0 to 0.15.0

This section applies if SDK 6 `0.15.0` is used on a project that was created using SDK 6 `0.14.0` or lower.

Unification of VEE dependency declaration

The `microejVeePort` configuration, used to define a VEE Port, and the `microejKernel` configuration, used to define a Kernel, have been unified into the `microejVee` configuration.

- To use a VEE Port or a Kernel published in an artifact repository, declare a Module dependency in the `build.gradle.kts` file:

```
dependencies {
  microejVee("com.mycompany:myVee:1.0.0")
}
```

- To use a VEE Port directory available locally, declare a file dependency in the `build.gradle.kts` file:

```
dependencies {
  microejVee(files("C:\\path\\to\\my\\veePort\\source"))
}
```

- To use a VEE Port archive available locally, declare a file dependency in the `build.gradle.kts` file:

```
dependencies {
  microejVee(files("C:\\path\\to\\my\\veePort\\file.zip"))
}
```

- To use a Kernel Virtual Device and Executable available locally, declare a file dependency in the `build.gradle.kts`:

```
dependencies {
  microejVee(files("C:\\path\\to\\my\\kernel\\executable.out", "C:\\path\\to\\my\\kernel\\virtual\\device"))
}
```

From 0.11.1 to 0.12.0

This section applies if SDK 6 0.12.0 is used on a project that was created using SDK 6 0.11.1 or lower.

Use of File Dependencies to Define a Local VEE Port or a Kernel Executable

The veePortPath and the kernelFile properties have been replaced by file dependencies.

- To use a VEE Port archive available locally, declare a file dependency in the build.gradle.kts file, with the microejVeePort configuration:

```
dependencies {
   microejVeePort(files("C:\\path\\to\\my\\veePort\\file.zip"))
}
```

- To use a VEE Port directory available locally, declare a file dependency in the build.gradle.kts file, with the microejVeePort configuration:

```
dependencies {
   microejVeePort(files("C:\\path\\to\\my\\veePort\\source"))
}
```

- To use a kernel Virtual Device and Executable available locally, declare a file dependency in the build.gradle.kts file, with the microejKernel configuration:

```
dependencies {
   microejKernel(files("C:\\path\\to\\my\\kernel\\executable.out", "C:\\path\\to\\my\\kernel\\virtual\\
↪device"))
}
```

From 0.10.0 to 0.11.0

This section applies if SDK 6 0.11.0 is used on a project that was created using SDK 6 0.10.0 or lower.

Gradle mechanism usage for Multiple VEE Ports Support

Using multiple VEE Ports in a project uses Gradle mechanism now instead of relying on in-house feature. This implies: - the veePortPaths property has been renamed to veePortPath and accepts a String value:

```
microej {
   veePortPath = "C:\\path\\to\\my\\veePort\\source"
}
```

- the kernelFiles property has been renamed to kernelFile and accepts a String value:

```
microej {
   kernelFile = "C:\\path\\to\\my\\kernel\\file"
}
```

Refer to the *How To Build an Executable With Multiple VEE Ports* section to learn how to support multiple VEE Ports using the Gradle mechanisms.

From 0.8.0 to 0.9.0

This section applies if SDK 6 `0.9.0` is used on a project that was created using SDK 6 `0.8.0` or lower.

Merge of the veePortDirs and veePortFiles properties

The build properties `veePortDirs` and `veePortFiles` have been merged into a single property `veePortPaths`. To define a local VEE Port, set the build property `veePortPaths` in the `microej` configuration block to the path of the VEE Port file (`.zip` or `.vde`) or to the source folder of the VEE Port:

```
microej {
  veePortPaths = listOf("C:\\path\\to\\my\\veePort\\source")
}
```

The `veePortPaths` property is defined as a list in order to provide multiple VEE Port files or source folders if it is needed:

```
microej {
  veePortPaths = listOf("C:\\path\\to\\my\\veePort1\\source", "C:\\path\\to\\my\\veePort2\\file.zip")
}
```

3.5 Application Developer Guide

The Application Developer Guide serves as a valuable reference for developing applications on MicroEJ, providing essential concepts for MicroEJ application design.

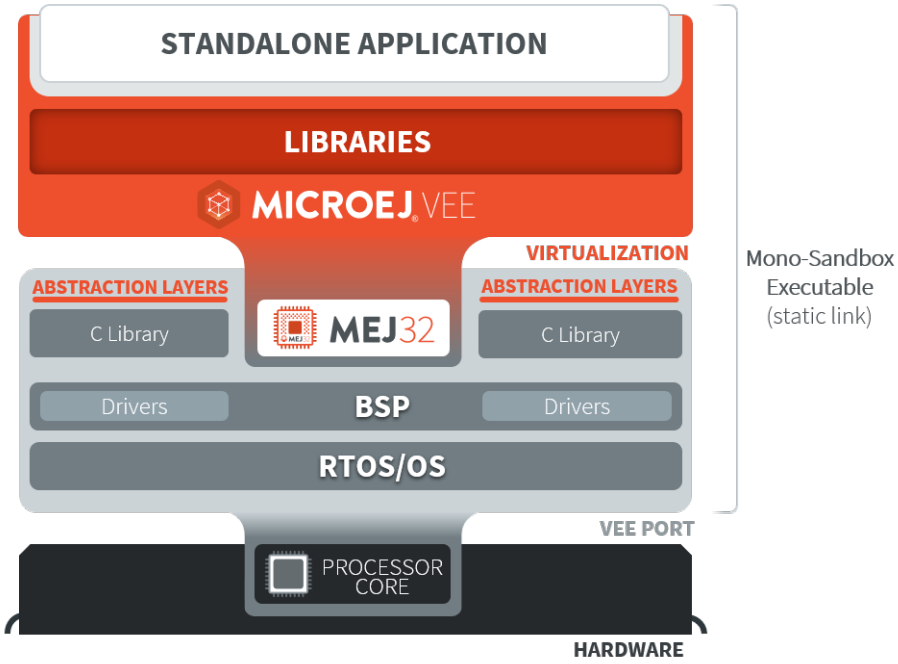
In addition to this document, you can access a number of helpful resources such as:

- Developer resources at https://developer.microej.com/.
- Libraries from the MicroEJ Central Repository.
- Application Examples as source code from MicroEJ Github Repositories.
- APIs Javadoc.
- *Training Courses for Application Developers*.

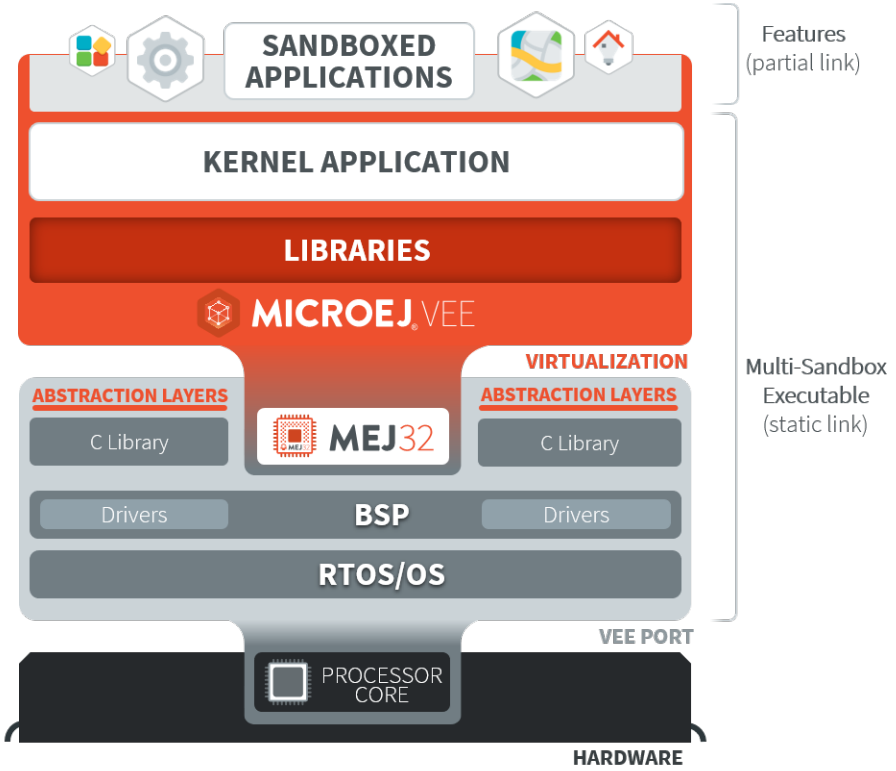
MicroEJ Applications are developed using standard Integrated Development Environments with *MICROEJ SDK*. The SDK allows you to run / debug / deploy Applications on a VEE Port.

Two kinds of applications can be developed: Standalone Applications and Sandboxed Applications.

A *Standalone Application* is the main Application that is directly linked to the C code to produce an Executable.



A *Sandboxed Application* is an Application that is partially linked, then deployed and executed over a Multi-Sandbox Executable.



3.5.1 MicroEJ Runtime

Language

MicroEJ allows to develop Applications in the Java® Language Specification version 7 with *some limitations*, and supports code extensions written in *JavaScript*.

Basically, Java source code is compiled by the Java compiler[1] into the binary format specified in the JVM specification[2]. This binary code is linked by a tool named *SOAR* before execution: `.class` files and some other application-related files (see *Classpath* chapter) are linked to produce the final binary file that the *Core Engine* will execute.

Note: When opened in the SDK 5, make sure that the Compiler Compliance Level of your project is set to 1.7 to ensure the bytecode produced by the Java compiler is compatible with MicroEJ. The Compliance Level can be changed from the menu: Window > Preferences > Java > Compiler .

Java Lambdas

Note: This feature is available for SDK 6 only and requires an Architecture version `8.0.0` or higher.

Starting from SDK 6 `1.0.0`, it is possible to use Java lambdas in the code of your Application. This feature is disabled by default, you can enable it as follows:

- Apply the Foojay Toolchains Plugin in the `settings.gradle.kts` file of your project:

```
plugins {
    id("org.gradle.toolchains.foojay-resolver-convention").version("0.9.0")
}
```

- Set the project property `java.lambdas.enabled` to `true` in command line with the `-P` argument:

```
$ ./gradlew runOnSimulator -Pjava.lambdas.enabled=true
```

or by adding it in the `gradle.properties` file of your project:

```
java.lambdas.enabled=true
```

Warning: Just like anonymous classes, each Java Lambda counts for one concrete class, so it could increase signicantly the number of concrete classes of your Application. See the *Limitations* page for the Architecture Limitations.

[1] The JDT compiler from the Eclipse IDE.
[2] Tim Lindholm & Frank Yellin, The Java Virtual Machine Specification, Second Edition, 1999

Core Libraries

This section describes the core libraries which make up the runtime. Theses Foundation Libraries are tightly coupled with the Core Engine.

Embedded Device Configuration (EDC)

The Embedded Device Configuration specification defines the minimal standard runtime environment for embedded devices.

This module is always required in the build path of an Application project; and all others libraries depend on it. This library provides a set of options. Refer to the chapter *Standalone Application Options* which lists all available options.

Specification Summary:

Java APIs	• java.io • java.lang • java.lang.annotation • java.lang.ref • java.lang.reflect • java.util
Latest Version	1.3

Module Dependency	SDK 6
	SDK 5
	`implementation("ej.api:edc:1.3.7")`
	`<dependency org="ej.api" name="edc" rev="1.3.7" />`

Module Location	https://repository.microej.com/modules/ej/api/edc/

Beyond Profile (BON)

This profile defines a suitable and flexible approach to fully control both memory usage and startup sequences on devices with limited memory resources, while remaining within the boundaries of Java semantics.

More precisely, it allows:

- Controlling the initialization sequence in a deterministic way.
- Defining persistent, immutable, read-only objects (that may be placed into non-volatile memory areas), and which do not require copies to be made in RAM to be manipulated.
- Defining immortal, read-write objects that are always alive.
- Accessing compile-time *constants*.

Read the *Beyond Profile specification* for more details.

Specification Summary:

Java APIs	ej.bon
Latest Version	1.4

Module Dependency	SDK 6
	SDK 5
	`implementation("ej.api:bon:1.4.4")`
	`<dependency org="ej.api" name="bon" rev="1.4.4" />`

Module Location	https://repository.microej.com/modules/ej/api/bon/

Simple Native Interface (SNI)

SNI provides a simple mechanism for implementing native Java methods in the C language.

SNI allows you to:

- Call a C function from a Java method.
- Share memory between C and Java using Immortal arrays (see the *Beyond Profile (BON)* to learn about immortal objects).

SNI also provides some Java APIs to manipulate some data arrays between Java world and the native (C) world.

Read the *Simple Native Interface specification* for more details.

Specification Summary:

Java APIs	ej.sni
Latest Version	1.4

Module Dependency	SDK 6
	SDK 5
	`implementation("ej.api:sni:1.4.3")`
	`<dependency org="ej.api" name="sni" rev="1.4.3" />`

Module Location	https://repository.microej.com/modules/ej/api/sni/

Kernel & Features (KF)

The Kernel & Features semantic (KF) extends the runtime for managing Multi-Sandboxed Applications.

Read the *Kernel & Features Specification (KF)* for more details, the *Multi-Sandbox capability* of the Core Engine and more generally the *Kernel Developer Guide* chapter.

Specification Summary:

Java APIs	ej.kf
Latest Version	1.7
Module Dependency	SDK 6 SDK 5 implementation("ej.api:kf:1.7.0") <dependency org="ej.api" name="kf" rev="1.7.0" />
Module Location	https://repository.microej.com/modules/ej/api/kf/

Specifications

Beyond Profile Specification (BON)

Introduction

This document defines the Beyond Profile Specification (BON), which is designed specifically for devices with limited memory resources. It introduces new concepts tailored to these constrained environments, while still adhering to the boundaries of Java semantics, hence the term "Beyond."

BON mainly defines a suitable and flexible approach to fully control both memory usage and startup sequences. More precisely, it allows:

- Controlling the initialization sequence in a deterministic way.
- Defining persistent immutable read-only objects (that may be placed into non-volatile memory areas), and do not require copies to be made in ram to be manipulated.
- Defining immortal read-write objects that are always alive.

BON also adds a set of useful utilities:

- Timers that allow to schedule one-time or cyclic activities. Such activities are Runnable objects that are automatically scheduled by the timer.
- A platform time from a monotonic clock.
- Read-write ByteArray support according to the underlying processor endianness.
- ResourceBuffer support for random read access in a *resource*.
- Compile-time *constants*.

Specification Summary

Java APIs	ej.bon
Latest Version	1.4
Module Dependency	SDK 6 SDK 5 implementation("ej.api:bon:1.4.4") <dependency org="ej.api" name="bon" rev="1.4.4" />
Module Location	https://repository.microej.com/modules/ej/api/bon/

Comments

Your comments about this specification are welcome. Please contact *our support team* with “BON” as subject.

Why BON ?

Many languages let software engineers define the memory management of their applications. One reason is that most embedded devices have scarce physical memory, while being cost driven. On the other hand, it is well known that memory allocation is one of the most difficult tasks to achieve efficiently as soon as the application uses more than a few objects.

In order to cope with these two contradictory issues, there are two main approaches, each one at the extremity of the possibility spectrum:

- Pre-allocate all of what is needed for the program to run, either statically (at compile-time) or dynamically once and for all at system startup. While running, no extra allocation is done. This approach is often used for Hard Real-Time systems when the memory consumption cannot be defined at compile-time through formal analysis.
- Let the runtime system manage the memory, fully freeing the engineers from that task. This is done through the use of garbage collectors. A huge number of different garbage collection policies are available and each have their own benefits and drawbacks.

The BON specification extends the existing Java specification, which already defines:

- a heap where Java objects are stored. This heap is automatically managed by a garbage collector.
- a semantically immutable set of objects, the pool of intern String,
- how applications are initialized, even though it is quite a loose process where lazy initialization is permitted. Intuitively, classes are initialized before any instance creation or access to its static variables (see *Deterministic Initialization Order*).

One of the newer trends in software involves designing simple solutions that are easy to understand and most importantly easy to manipulate and control. Developers must be able to minimize development time, often dealing with small memory budgets for their application. On microcontrollers, there is much more read-only memory than volatile memory.

First Example

The simple next example illustrates the use of big buffers. They are made immortal in order to recycle them manually while they represent the most critical ram consumption. This example also makes use of an immutable object, an array of values that never changes during the lifetime of the device.

```
package example;

import ej.bon.*

public class Filter {
    public static final int BufferSize = 4096;// 16k (an int is 32-bit)
    public static int[][] Buffers;
    public static int[] ValidValues;
    static {
        ValidValues =(int[])Immutables.get("filter");
    }
```

(continues on next page)

(continued from previous page)

```
    static {
        // Allocate the immortal pool of buffers. Only two Filters may
        // be alive at the same time: ==> 32k of ram for two Filters
        Buffers = new int[2][];
        Buffers[0] = (int[])Immortals.setImmortal(new int[BufferSize]);
        Buffers[1] = (int[])Immortals.setImmortal(new int[BufferSize]);
    }

    public int ptr;
    public int[] buffer;

    public Filter(){
        // grab a buffer from the pool, or throw an exception
        synchronized (Buffers){
            for (int i = Buffers.length; --i >= 0; ){
                if (Buffers[i] != null ){
                    buffer = Buffers[i];
                    ptr=-1;
                    Buffers[i] = null ;
                    break ;
                }
            }
        }
        if (buffer == null ){
            throw new OutOfMemoryError();
        }
    }

    public void close(){
        // recycle manually the immortal buffer: store it in
    // global Buffers array pool
        synchronized (Buffers){
            for (int i = Buffers.length; --i >= 0; ){
                if (Buffers[i] == null ){
                    Buffers[i] = buffer;
                    break;
                }
            }
        }
    }

    public synchronized void insert( int value){
        // only insert permitted values

        if (ptr >= BufferSize) return; // full

        for (int j = ValidValues.length; --j >= 0; ){
            if (value == ValidValues[j]){
                buffer[++ptr] = value;
            }
        }
    }
}
```

Object Natures

The BON specification extends the Java semantic by defining three natures for objects:

- *Immutable objects*: persistent objects that are alive at system startup. Immutable objects are also referred to as read-only objects, since they most probably reside in non-volatile memory. All together they form a pre-existing world that exists on its own, just like the hardware does.
- *Immortal objects*: objects that do not move around in memory: they remain physically located in one memory location forever.
- Reclaimable objects: the regular objects managed by the Garbage Collector.

Although objects get a liveness nature, this is fully transparent for the application developer this is completely transparent to the application developer, except for the restriction that writing to an immutable object is not allowed.

Persistent Immutable Objects

Immutable objects are read-only objects. They are instances of any concrete class. Although they are immutable, they adhere to all the Java object's semantics. In particular, they hold a hash code, have a class and have a monitor that a thread may enter into.

There is no way for an immutable object to directly refer to a non-immutable object. References from immutable objects always refer to other immutable objects. Writing into an immutable object (field write access) results in an unspecified behavior.

Immutable objects are declared at build time by specifying objects in an XML configuration file, as described in the sections immediately below.

The immutable objects will be linked to a particular location in (read-only) memory. Note that immutable objects do not need to be copied in (scarce) ram memory to be manipulated.

Software is made up of several parts, often called libraries, that may come with their own immutable object descriptions. Therefore more than one immutable description may be declared in the classpath.

Object ID and Immutable Object Querying

Immutable objects are semantically organized into one global pool, just like the Java interned String objects.

An immutable object may be attached to a String key, known as its ID. This ID allows an immutable object to be retrieved out of the global pool of immutable objects, thanks to the method Immutables.get(String). The ID of an object is globally unique in the system.

Immutable Objects Descriptions and Creation

Descriptions are based on the structure of objects, that is, they embed structural information such as fully qualified class names and field names. Fields that need to get initialized with some value (base-type or another immutable object) are described using a pair: field-name, value. Only instance fields of objects are involved, i.e., not static fields.

Fields that are not described get initialized with the default Java value (0 for numeric types, null for objects, false for booleans, 0.0 for floating-point numbers). No visibility rule applies, that is, any kind of field may be listed, even private ones.

There is no particular order for the creation of the immutable objects.

XML Grammar

Immutable objects are described according to the following XML syntax (*Annex A: Immutables DTD*).

- <immutables> : the root element of one immutable objects description.
  - attributes:
    - name : an optional attribute that defines the content of the XML description.
  - child elements: <object> , <objectAlias> , <array> , <string> , <class> , <null> , <importObject> .
- <object> : element that defines a new object.
- attributes:
  - id : the ID string that allows the object to be retrieved through the use of Immutables.get(String).
  - type : the name of the class of the object. An alias may be used instead of the fully-qualified class name.
  - private : a boolean that indicates whether the object will be accessible using the Immutables.get(String) method. If false, the objects can only be referenced within the XML immutable objects descriptions.
- child elements: <field> , <refField>
- <objectAlias> : element that defines a new key for an existing object.
  - attributes:
    - id : the ID string that allows the object to be retrieved through the use of Immutables.get(String).
    - object : the existing object ID or alias ID.
    - private : a boolean that indicates whether the object will be accessible using the Immutables.get(String) method. If false, the objects can only be referenced within the XML immutable objects descriptions.
  - child elements: none
- <string> : element that defines an interned string.
  - attributes:
    - id : the ID that allows the object to be retrieved through the use of Immutables.get(String)
    - value : the string literal
    - private : a boolean that indicates whether the object will be accessible using the Immutables.get(String) method. If false, the objects can only be referenced within the XML immutable objects descriptions.
  - child elements: none
- <class> : element that defines an instance of a java.lang.Class . The ID of this object can be used for type attributes.
  - attributes:
    - id : the ID that allows the object to be retrieved through the use of Immutables.get(String)
    - type : the class fully qualified name like java.lang.Object .
    - private : a boolean that indicates whether the object will be accessible using the Immutables.get(String) method. If false, the objects can only be referenced within the XML immutable objects descriptions.
  - child elements: none

- <field> : elements that state a field with its associated literal value.
  - attributes:
    - name : the name of the field as defined in the class that defines it.
    - value : the value of the field. The value is a primitive type (numeric or boolean) or a literal string (see).
    - type : this attribute is optional. It represents the class where the field is defined. A field without its type attribute refers to the first field found while scanning the class hierarchy from the bottom to the top (following the superclass link).
  - child elements: none
- <refField> : elements that state a field that references an immutable object.
  - attributes:
    - name : the name of the field as defined in the class that defines it.
    - ref : the ID of the referenced immutable object.
    - type : this attribute is optional. It represents the class where the field is defined. A field without its type attribute refers to the first field found while scanning the class hierarchy from the bottom to the top (following the superclass link).
  - child elements: none
- <array> : element that defines a new array.
  - attributes:
    - id : the ID that allows the object to be retrieved through the use of Immutables.get(String)
    - type : the array type. An alias may be used instead of the fully qualified class name. Dimensions are given using the Java notation [] .
    - length : this attribute is optional. It represents the number of elements the array has.
    - private : a boolean that indicates whether the object will be accessible using the Immutables.get(String) method. If false, the objects can only be referenced within the XML immutable objects descriptions.
  - child elements: <elem> , <refElem>
- <elem> : element that defines an array element with its literal value.
  - attributes:
    - value : the value of the element. The value is a primitive type (numeric or boolean) or a literal string (see).
  - child elements: none
- <refElem> : element that defines an array element. Such element references an immutable object.
  - attributes:
    - ref : the ID of the referenced immutable object.
  - child elements: none
- <null> : element that defines a null object that can be referenced by an object field or an array element
  - attributes:
    - id : the ID that allows the null object to be retrieved through the use of Immutables.get(String)

        * private : a boolean that indicates whether the object will be accessible using the Immutables.get(String) method. If false, the objects can only be referenced within the XML immutable objects descriptions.
    - child elements: none
- <importObject> : element that import an object that is defined in another immutable file. The referenced object may be private or public.
    - attributes:
        * id : the ID of the imported object
    - child elements: none

Class names use the Java notation (using a '.' as separator): java.lang.Object is an example.

String literals are defined as in XML specification. To allow quotes in XML string data use the apostrophe '' ' separator as XML separator or the escape character " .

To define the next 9 characters String, my"String , as string literal value, use one of following syntax:

```
<field name="f1" value='my"String' />
<field name="f1" value="my&amp;quot;String" />
```

Next table lists the format for the primitive values:

Table 1: Table 1: Immutables Primitive Type Format

Primitive Type	Format	Example
boolean	true or false	<...value="true"/>
byte, short, int, long	Format defined in the Java method Long.decode(String)	<...value="123"/> <...value="0x2A"/> <...value="-561"/>
char	Format defined in the Java method Long.decode(String) or a character value between simple quotes	<...value="123"/ > <...value="'z'"/> <...value="'©'"/>
float	Format defined in the Java method Float.parseFloat (String)	<...value="2.3"/> <...value="4.2e12"/> <...value="-5.671"/>
double	Format defined in the Java method Double.parseDouble (String)	<...value="2.3"/> <...value="4.2e12"/> <...value="-5.671"/>

IDs define one global name space: an ID only refers to only one object. It is an error to have objects sharing ID. As a good practice, it is recommended to define ID using a qualified name, such as *myCorp.myApp.MyID12*.

Immutable XML Description Examples

```
<immutables name="MyCorp objects">

   <array id="corp.immut00" type="boolean[]" length="2">
      <elem value="true"/>
      <elem value="false"/>
   </array>

   <array id="corp.immut01" type="int[]">
      <elem value="3"/>
      <elem value="2"/>
      <elem value="1"/>
   </array>

   <class id="MyClass" type="myCompany.mypackage.MyClass"
   private="true"/>

   <object id="corp.immut02" type="MyClass">
      <field name="a" value="50" />
      <field name="str" value="Hello" />
      <refField name="b" ref="corp.null" />
   </object>

   <object id="corp.immut03" type="myCompany.mypackage.A">
      <refField name="f" ref="corp.immut04" />
      <refField name="s" ref="corp.internalKey"/>
      <refField name="o" ref="corp2.immut"/>
   </object>

   <string id="corp.immut04" value='Hello World!' />

   <string id="corp.internalKey" value="one" private="true" />

   <string id="key1" value="two" />

   <string id="key2" value='thr"ee' />

   <object id="value1" type="java.lang.Object" />

   <null id="corp.null"/>

   <importObject id="corp2.immut"/>

</immutables>
```

Immortal Objects

Non Garbageable Objects

Immortal objects are regular objects that are not managed by the garbage collector. Immortal objects do not move around in memory: they remain physically located in one memory location forever.

Turning Objects Into Immortal Objects

Reclaimable objects may be turned into immortal objects using the Immortals.setImmortal(Object) method. Only the object passed as argument is turned into an immortal object, i.e., none of the objects it refers to through its fields become immortal. This is in contrast with Immortals.deepImmortal(Object) that turns the object passed as the argument and all objects referred to from the argument into immortal objects. Note that weakly reachable objects are not turned into immortal objects; in other words the WeakReference semantic is not affected by this operation.

The total amount of free immortal memory still available is Immortals.freeMemory(). It is system dependent.

The system provides the possibility to create objects directly as immortal objects using the method Immortals.run(Runnable): while the Runnable.run() method executes, all created objects are allocated as immortal objects.

Runtime Phases

BON defines two phases of execution:

- The initialization phase: this is the very first Java code that executes. Its purpose is to let the device "boot", that is, to initialize all necessary resources, like allocating buffers for drivers, performing default sanity checks, scanning hardware, etc. The initialization sequence executes all the static initializer methods (known as the $<\text{clinit}>$ methods).
- The mission phase: once initialized, the device switches to the endless mission phase. The $\text{main}(\text{String}[])$ method of the main class is called and the application runs until the device is switched off.

Util.isInInitialization() and Util.isInMission() methods allow the phase to be tested.

Mono-threaded Phase

During the initialization phase, there is only one thread running: the main thread which will eventually execute the $\text{main}(\text{String}[])$ method once the system enters the mission phase.

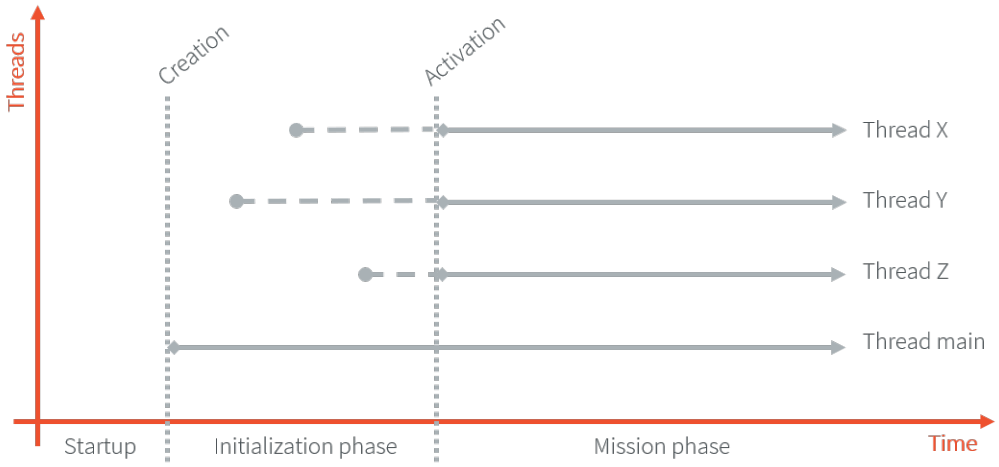


Fig. 109: BON phases and threads activation.

If other threads are created while the class initializations execute (<clinit> methods), those threads will be on hold (i.e., waiting) until the system enters the mission phase, even if those threads have received the start() message and have a higher priority than the main thread.

Deterministic Initialization Order

If a class needs to be initialized, it defines a <clinit> method. These methods are not visible per se at the Java source level. They are generated by compilers: they capture the semantic of the initialization of both static fields and static initializers of classes.

During the initialization phase, all classes which are involved within the application are initialized. It implies calling all <clinit> methods, in sequence.

The order is statically computed from the application code entry points and does not rely on runtime behavior. If the application code does not change, the order remains the same. The order is also compatible with the Java semantic. Intuitively, a class may depend on other classes. Those classes should be initialized first. Here are some of the main dependencies: object creation, superclass, method receiver, argument types, field types, etc.

Dependencies of classes upon themselves define a graph of dependencies. This graph may depict cycles. The graph is linearized in an order which depends only on the graph itself.

The classes dependencies also includes all the classes of pre-configured *immutable objects* objects.

Utilities

Timer & TimerTask

A Timer defines a single thread in charge of scheduling Runnable objects from the TimerTask class. All TimerTask are executed sequentially, according to their schedule. A Timer does its best effort to schedule the TimerTask appropriately, which depends on the TimerTask durations and schedules (there is no real-time guaranties).

A TimerTask may be scheduled repeatedly. In that case, the delay for the next schedule may depend on the end of the previous ending of the TimerTask, and not on some absolute time: if the previously execution of the TimerTask

is delayed for some reason, the next executions are delayed too by the same amount of time. It is also possible to schedule repeatedly a TimerTask at fixed rate, which allows executions to be independent.

If a TimerTask execution terminates unexpectedly, the other tasks are not impacted: the TimerTask is assumed to have terminated its execution regularly, and is not rescheduled event if it was scheduled repeatedly.

The main APIs are:

- Timer.schedule(TimerTask, long) and Timer.schedule(TimerTask, Date) methods allow to schedule one execution after the specified delay.
- Timer.schedule(TimerTask task, long, long) and Timer.schedule(TimerTask, Date, long) methods allow to schedule repeatedly executions, the first one after the specified delay. The waiting time between two executions is relative to the end of the previous execution.
- Timer.scheduleAtFixedRate(TimerTask task, long, long) and Timer.scheduleAtFixedRate(TimerTask, Date, long) methods allow to schedule repeatedly executions, the first one after the specified delay. The waiting time between two executions is independent of the end of the previous execution.

Platform Time

The application time is the user time: it depends on its localization. System.currentTimeMillis() returns the application time expressed in milliseconds since midnight, January 1, 1970 UTC.

BON introduces a platform time that is monotonic. It always moves forward and is not impacted by application time modifications (e.g., Synchronization of time with an NTP server, TimeZone, or Daylight Savings Time updates). It can be implemented by returning the running time since the start of the device.

The Util class defines several methods to handle both application time and platform time:

- Util.platformTimeNanos() and Util.platformTimeMillis() method return the platform time, a `long`, expressed in nanoseconds and in milliseconds.
- Util.setCurrentTimeMillis(long) and Util.setCurrentTimeMillis(Date) methods allow to change the application time in order to match a user localization. This has no effect on the platform time. Util.currentTimeMillis() method is a synonym of System.currentTimeMillis().

Byte Array Accesses

The addresses space is 8-bit oriented even if there are platforms that manipulate quantities that are larger than an 8-bit: 32-bit processors for example do so. The ordering of individual addressable sub-components within the representation of a larger data item is called the endianness. `BigEndian` describes an ordering with the most significant byte first, whereas `LittleEndian` describes an ordering with the least significant byte first.

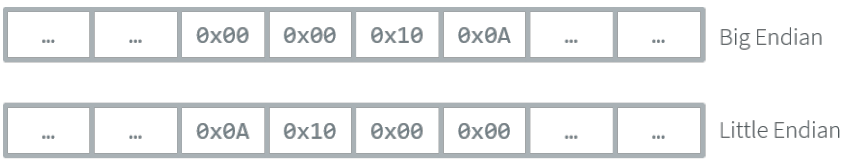


Fig. 110: Representation of the 32-bit quantity 0x0000100A using both BigEndian and in LittleEndian layout.

BON introduces methods to read and write into array of byte (`byte[]`) according to the platform endianness, or according to a specific provided endianness. The ByteArray class provides such APIs:

- ByteArray.getPlatformEndianness() returns the underlying system-dependent endianness, which mostly depends on the target processor(s).
- ByteArray.readInt(byte[], int) and ByteArray.writeInt(byte[], int, int) reads and writes an `int` using the platform specific endianness.
- ByteArray.readInt(byte[], int, int) and ByteArray.writeInt(byte[], int, int, int) reads and writes an `int` using the specified endianness as last argument, which may be either LITTLE_ENDIAN or BIG_ENDIAN.

Similar methods are provided for `short`, `char`, and `long` types.

Annex A: Immutables DTD

```
<!ELEMENT immutables ( object*, objectAlias*, array*, string*, class*, null*, importObject* ) >
<!ATTLIST immutables
   name     CDATA #IMPLIED
>

<!ELEMENT object ( field*, refField* ) >
<!ATTLIST object
   id       ID    #REQUIRED
   private        (true | false) "false"
   type           CDATA #REQUIRED
>

<!ELEMENT objectAlias EMPTY >
<!ATTLIST objectAlias
   id       ID    #REQUIRED
   private        (true | false) "false"
   object   IDREF #REQUIRED
>


<!ELEMENT array ( elem*, refElem* ) >
<!ATTLIST array
   id       ID    #REQUIRED
   private        (true | false) "false"
   type     CDATA #REQUIRED
   length   CDATA #IMPLIED
>

<!ELEMENT elem EMPTY >
<!ATTLIST elem
   value CDATA #REQUIRED
>

<!ELEMENT refElem EMPTY >
<!ATTLIST refElem
   ref IDREF #REQUIRED
>

<!ELEMENT class EMPTY >
<!ATTLIST class
   id       ID    #REQUIRED
   private        (true | false) "false"
   type           CDATA #REQUIRED
>
```

(continues on next page)

(continued from previous page)

```
<!ELEMENT string EMPTY >
<!ATTLIST string
   id      ID    #REQUIRED
   private         (true | false) "false"
   value   CDATA #REQUIRED
>

<!ELEMENT field EMPTY >
<!ATTLIST field
   name  CDATA #REQUIRED
   value CDATA #REQUIRED
   type  CDATA #IMPLIED
>

<!ELEMENT refField EMPTY >
<!ATTLIST refField
   name  CDATA #REQUIRED
   ref     IDREF #REQUIRED
   type  CDATA #IMPLIED
>

<!ELEMENT null EMPTY >
<!ATTLIST null
   id      ID    #REQUIRED
   private         (true | false) "false"
>

<!ELEMENT importObject EMPTY >
<!ATTLIST importObject
   id      ID    #REQUIRED
>
```

Simple Native Interface Specification (SNI)

Introduction

The Simple Native Interface specification (SNI) defines how to cross the barrier between the Managed world and the native world:

- Call a C function from Java code.
- Pass parameters to the C function.
- Return a value from the C world to the Managed world.
- Manipulate (read & write) shared memory both in Java and C code: the immortal space.

Note: In the following explanations, the term *task* refers to native tasks scheduled by the underlying OS or RTOS, while *thread* refers to MicroEJ threads scheduled by the Core Engine.

The sections below also mention some functions from the SNI native API. See here for the complete SNI Native API reference: *SNI Native API*.

Specification Summary

Java APIs	ej.sni
Latest Version	1.4

Module Dependency	SDK 6
	SDK 5
	`implementation("ej.api:sni:1.4.3")`
	`<dependency org="ej.api" name="sni" rev="1.4.3" />`

Module Location	https://repository.microej.com/modules/ej/api/sni/

Comments

Your comments about this specification are welcome. Please contact *our support team* with "SNI" as subject.

First Example

This first example shows how to declare and implement a Java native method using SNI. First the method has to be declared `native` in Java code: this states that the method is written in another language.

```
package example;

import java.io.IOException;

/**
 * Abstract class providing a native method to access sensor value.
 * This method will be executed out of the Core Engine.
 */
public abstract class Sensor {

    public static final int ERROR = -1;

    public int getValue() throws IOException {
        int sensorID = getSensorID();
        int value = getSensorValue(sensorID);
        if (value == ERROR) {
            throw new IOException("Unsupported sensor");
        }
        return value;
    }

    protected abstract int getSensorID();

    public static native int getSensorValue(int sensorID);
}

class Potentiometer extends Sensor {

    protected int getSensorID() {
        return Constants.POTENTIOMETER_ID; // POTENTIOMETER_ID is a static final
```

(continues on next page)

(continued from previous page)

```
    }
}
```

Then, the implementation of the method is written in C language.

```
// File providing an implementation of native method using a C function
#include <sni.h>
#include <potentiometer.h>

#define SENSOR_ERROR (-1)
#define POTENTIOMETER_ID (3)

jint Java_example_Sensor_getSensorValue(jint sensor_id){

    if (sensor_id == POTENTIOMETER_ID)
    {
        return get_potentiometer_value();
    }
    return SENSOR_ERROR;
}
```

Java code and C Execution Sequence

Calling C from Java

When a Java native method executes, it executes its C counterpart function. This is done using the CPU budget of the task that has started the Core Engine. While the C function executes, no other Java methods executes and the Core Engine cannot schedule other threads. The Managed world "waits" for the C function to finish.

The following illustration shows the execution a the Core Engine task. Green thread 3 has called a native method that executes in C. All Java activities is suspended until the C execution has finished.

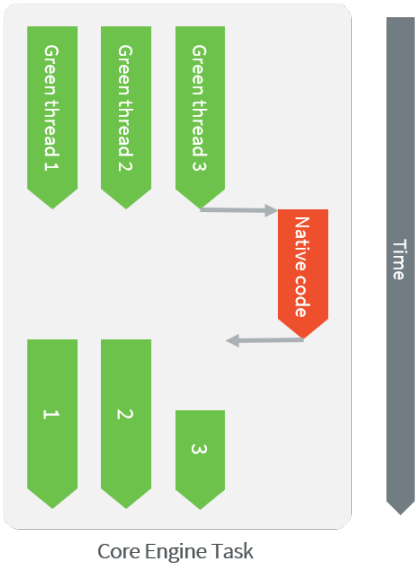


Fig. 111: Execution of Threads by the Core Engine Task

Synchronization

SNI defines C functions that provide controls for the threads activities:

- `int32_t SNI_suspendCurrentJavaThread(int64_t timeout)` : Suspends the execution of the thread that initiated the current C call. This function does not block the C execution. The suspension is effective only at the end of the native method call (when the C call returns). The thread is suspended until either an task calls `SNI_resumeJavaThread` , or the specified number of milliseconds has elapsed.
- `int32_t SNI_getCurrentJavaThreadID(void)` : Permits retrieval of the ID of the current thread within the C function (assuming it is a "native Java to C call"). This ID must be given to the `SNI_resumeJavaThread` function in order to resume execution of the thread.
- `int32_t SNI_resumeJavaThread(int32_t id)` : Resumes the thread with the given ID. If the thread is not suspended, the resume stays pending.

The following illustration shows `Green thread 3` which has called a native method that executes in C. The C code suspends the thread after having provisioned its ID (e.g. 3). Another task may later resume the thread.

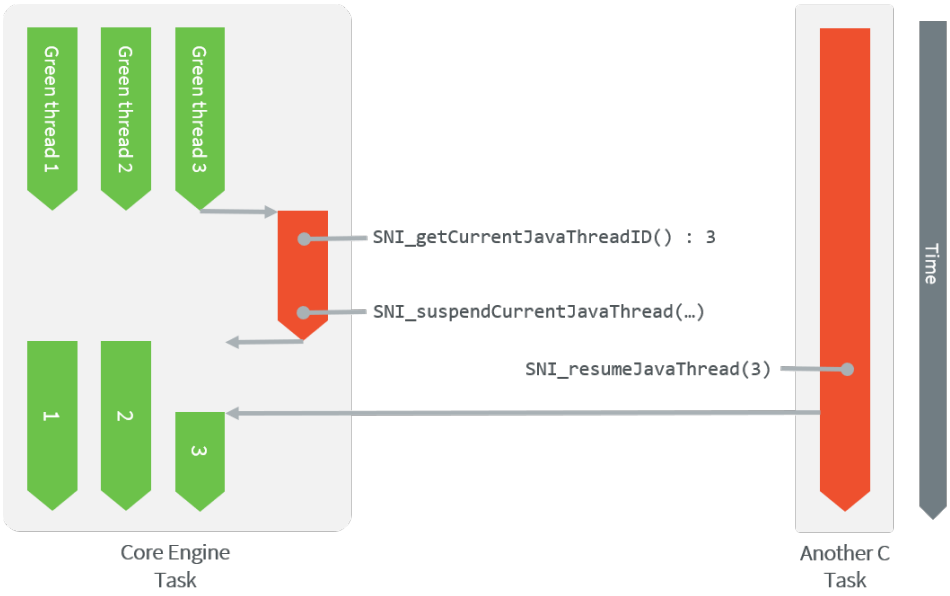


Fig. 112: Green Threads and Task Synchronization

Java And Native Separation

The following illustration shows both Java code and C code accesses to shared objects in the immortal space, while also accessing their respective memory. In C code, non-immortal arrays can only be accessed within the local scope of a native function.

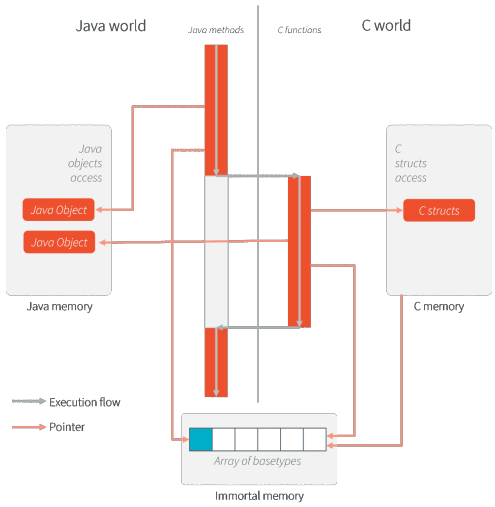


Fig. 113: Java code and C shared objects

Managed World to C World

C Function Call From Managed World

The SNI specification allows the invocation of methods from Java to C: these methods must be declared `static native` methods, and the parameters must be base types or array of base types. These native methods are used in Java code as standard Java methods.

Example:

```
package example;
public class Foo{

   public void bar(){
      int times = 3;
      print(times);
   }

   public static native void print(int times);
}
```

```
#include <sni.h>
#include <stdio.h>

void Java_example_Foo_print(jint times){
   while (--times >= 0){
      printf("Hello world!\n");
   }
}
```

Java Types And C Types

Base Types

Types may have different representations depending on the language. The file `sni.h` defines the C types that represent exactly the Java types.

Table 2: Java types to C types

Java Type	Specification	C type
void	No returned type	`void`
boolean	unsigned 8 bits	`jboolean`
byte	signed 8 bits	`jbyte`
char	unsigned 16 bits	`jchar`
short	signed 16 bits	`jshort`
int	signed 32 bits	`jint`
long	signed 64 bits	`jlong`
float	IEEE 754 single precision 32 bits	`jfloat`
double	IEEE 754 double precision 64 bits	`jdouble`

Java Array

The Java arrays (of base types) are represented in C functions as C arrays: the array is a pointer on the first element of the array, all the elements in line within the memory.

SNI allows to get a Java array length in a C function.

```
int32_t SNI_getArrayLength(void* array);
```

Strings

Strings are typically represented quite differently between C & Java code. In C, strings are represented with C `char` (8-bit) array with a `'\0'` as last character. In Java code, strings are `jchar` (16-bit) array, not terminated by `'\0'`.

To help with the conversion, the SNI Java API provides utility methods to convert between the two representations.

Example:

```
package example;
public class Foo {

   private static final int MAX_STRING_SIZE = 42; // including the '\0' character

   public void pushString(String str) {
      pushString(SNI.toCString(str));
   }
```

(continues on next page)

(continued from previous page)

```
    public String pullString() {
        byte[] buffer = new byte[MAX_STRING_SIZE];
        pullString(buffer);
        return SNI.toJavaString(buffer);
    }

    private static native void pushString(byte[] str);
    private static native void pullString(byte[] buffer);

}
```

```
#include <sni.h>
#include <string.h>

#define MAX_STRING_SIZE 42

static char gStr[MAX_STRING_SIZE];

void Java_example_Foo_pushString(jbyte *str) {
    strncpy(gStr, (char*) str, MAX_STRING_SIZE);
}

void Java_example_Foo_pullString(jbyte *buffer) {
    strncpy((char*) buffer, gStr, MAX_STRING_SIZE);
}
```

Note: The string conversions use the *default platform encoding*. To use a different encoding, refer to the String API.

Naming Convention

SNI uses a naming convention to name-match the Java native method with its C counterpart function.

The C function name is the concatenation of the following components:

- the prefix " Java_ ".
- the package name of the class, each sub packages is separated with " _ ".
- the separator " _ ".
- the class name.
- the separator " _ ".
- the method name.

If the method is overloaded by another method, native or not (the two methods have the same name with different arguments), the function name must be followed by the arguments descriptor, obtained with the following components (except if the method has no arguments):

- the separator " ___ " (two underscores)
- the name of each argument type, without separator, preceded by " _3 " if it is an array.

The following table gives the descriptors of the Java types for arguments.

Table 3: SNI Java types descriptors in arguments

Java type	SNI name
boolean	Z
byte	B
char	C
short	S
int	I
long	J
float	F
double	D

The character underscore (“ __ ”) is used as a separator in the name. If this character is used within the Java name (either in package, class name or method name), it is replaced with “ _1 ”. Because the Java names cannot start with a number, the characters “ _1 ” cannot be confused with separator character.

Examples of Java native methods and their counterpart C functions:

```
package example.sni.impl;

class Hello {

    public static native void nativ01(int i);
    public static native void nativ02(boolean b, int[] i);
    public static native void nativ_03();
    public static native void nativ04();
    public static native void nativ04(long l, double d);
    public static native void nativ04(int[] ia, int ib, char[] ca);

}
```

```
void Java_example_sni_impl_Hello_nativ01(jint i);
void Java_example_sni_impl_Hello_nativ02(jboolean b, jint* i);
void Java_example_sni_impl_Hello_nativ_103();
void Java_example_sni_impl_Hello_nativ04();
void Java_example_sni_impl_Hello_nativ04__JD(jlong l, jdouble d);
void Java_example_sni_impl_Hello_nativ04___3II_3C(jint* ia, jint ib, jchar* ca);
```

Parameters Constraints

There are strong constraints on arguments given by Java methods to native functions:

- Only base types and array of base types are allowed in the parameters. No other objects can be passed: the native functions cannot access Java objects field nor methods.
- When base type arrays are passed in parameters, they must have only one dimension. No multi dimension array are allowed (int[][] is forbidden for example).
- Only base types are allowed as return type.

This constraints are checked at link-time to ensure that they are respected.

Startup

The Core Engine needs first to be initialized, and then started. It is the programmer responsibility to create a task and to start the Core Engine within this task.

SNI defines C functions to create a Managed world, to start it and to free it:

- void SNI_createVM(void) : creates and initializes the Core Engine context.
- int32_t SNI_startVM(void,int32_t,char) : starts the Core Engine. This function returns when the Java application ends.
- int32_t SNI_getExitCode(void vm) : gets the Java application exit code, after SNI_startVM has successfully returned. This is the value passed by the application to System.exit() method.
- void SNI_destroyVM(void vm) : does nothing if the Core Engine is still running. This function must be called in the task that created the Core Engine.

The following illustration shows a typical example of Core Engine startup code.

Listing 6: Example of Core Engine startup code in C

```
void microej_main(int argc, char **argv) {
  void* vm;
  int core_engine_error_code = -1;
  int32_t app_exit_code = 0;

  vm = SNI_createVM();
  if (vm == NULL) {
    printf("MicroEJ initialization error.\n");
  } else {
    core_engine_error_code = (int)SNI_startVM(vm, argc, argv);
    if (core_engine_error_code < 0) {
      printf("MicroEJ execution error (err = %d).\n", (int) core_engine_error_code);
    } else {
      app_exit_code = SNI_getExitCode(vm);
      printf("MicroEJ END (exit code = %d)\n", (int) app_exit_code);
    }
    SNI_destroyVM(vm);
  }
}
```

Resources

SNI also provides a mechanism to register native resources.

This binds the native resource to the calling application so that the native resource is closed if the application is terminated. In the case of a Standalone Application (or if calling from the Kernel context in a Multi-Sandbox setup), this is when the Application (or Kernel) exits, before the Core Engine returns from SNI_startVM() (see *Startup*). In the case of a Sandboxed Application, this is when the Application is being stopped (or when the Kernel exits).

Note: The *Core Engine state dump* lists the registered native resources.

There is nothing to do on the native method declaration side (Managed Code) and this can therefore depend of a specific Abstraction Layer Implementation. In most cases, the library will also provide a "close" API (implemented with a "close" native method) so that the Application can explicitly close the native resource. Typically, the API will provide a class reifying this native resource which would implement AutoCloseable .

It is also possible to bind a native resource to a Managed object (typically a Java object) using NativeResource.closeOnGC(). In this case, the native resource will also be closed when the bound object is Garbage-Collected (useful when the API does not expose any close function).

See the API reference for SNI_registerResource() and SNI_unregisterResource().

Scoped Resources

SNI also provides scoped resources. In this case, the scope of the resource is the native call. It can be used to share a resource between a native function implementation and a callback (see SNI_suspendCurrentJavaThreadWithCallback()). The resource will be closed when the native call returns (after the native function and the potential successive suspend/resume callbacks returns), or after the calling Application is terminated (before the suspended native call is completed).

See the API reference for SNI_registerScopedResource() and SNI_unregisterScopedResource().

Note: Only 1 scoped resource registered at a time is supported per native call.

Typical use cases

A typical SNI use case is for the Managed Code to maintain a reference to a C object (typically a pointer to a struct).

For example, here is a struct:

```
typedef struct point {
    int id;
    char[MAX_NAME_LENGTH] name;
    int[2] coordinates;
} point_t;
```

To keep a reference to an instance of the struct, we typically use a handle: a 32-bit pointer stored as a jint. Then, the handle is passed as an argument to the different native methods that will use the object, either to manipulate directly the struct, or pass it as an argument to another C (library) function.

```
package example.sni;

import ej.bon.Constants;
import ej.sni.SNI;

class Point implements AutoCloseable {

    private final int handle;

    Point() {
        this.handle = PointNatives.createPoint();
    }

    int getId() {
        return PointNatives.getPointId(this.handle);
    }

    void setId(int id) {
        PointNatives.setPointId(this.handle, id);
```

(continues on next page)

(continued from previous page)

```
    }

    String getName() {
        byte[] buffer = new byte[PointNatives.MAX_NAME_LENGTH];
        PointNatives.getPointName(this.handle, buffer);
        return SNI.toJavaString(buffer);
    }

    void setName(String name) {
        if (name.length() > PointNatives.MAX_NAME_LENGTH) {
            throw new IllegalArgumentException();
        }
        PointNatives.setPointName(this.handle, SNI.toCString(name));
    }

    int[] getCoordinates() {
        int[] coordinates = new int[2];
        coordinates[0] = PointNatives.getCoordinate(this.handle, 0);
        coordinates[1] = PointNatives.getCoordinate(this.handle, 1);
        return coordinates;
    }

    void setCoordinates(int[] coordinates) {
        PointNatives.setCoordinate(this.handle, 0, coordinates[0]);
        PointNatives.setCoordinate(this.handle, 1, coordinates[1]);
    }

    @Override
    public void close() {
        PointNatives.deletePoint(this.handle);
    }

}

class PointNatives {

    static final int MAX_NAME_LENGTH = Constants.getInt("example.sni.point.name.max.length");

    static native int createPoint();

    static native void deletePoint(int handle);

    // point->id

    static native int getPointId(int handle);

    static native void setPointId(int handle, int id);

    // point->name

    static native void getPointName(int handle, byte[] buffer); // buffer size: maximum name length,␣
↪implementation must
                                                                // fill buffer with a null-terminated string (or full
                                                                // buffer)

    static native void setPointName(int handle, byte[] name);
```

(continues on next page)

(continued from previous page)

```
    // point->coordinates

    static native int getCoordinate(int handle, int index);

    static native void setCoordinate(int handle, int index, int value);

}
```

```
#define point__create           Java_example_sni_PointNatives_createPoint
#define point__delete           Java_example_sni_PointNatives_deletePoint
#define point__get_id           Java_example_sni_PointNatives_getPointId
#define point__set_id           Java_example_sni_PointNatives_setPointId
#define point__get_name         Java_example_sni_PointNatives_getPointName
#define point__set_name         Java_example_sni_PointNatives_setPointName
#define point__get_coordinate   Java_example_sni_PointNatives_getCoordinate
#define point__set_coordinate   Java_example_sni_PointNatives_setCoordinate

static void point__close(void* point) {
    free(point);
}

jint point__create() {
    point_t* point = malloc(sizeof(point_t));
    SNI_registerResource((void*) point, (SNI_closeFunction) point__close, NULL);
    return (jint) point;
}
void point__delete(jint handle) {
    void* point = (void*) handle;
    SNI_unregisterResource(point, (SNI_closeFunction) point__close);
    point__close(point);
}
jint point__get_id(jint handle) {
    point_t *point = (point_t*) handle;
    return (jint) point->id;
}
void point__set_id(jint handle, jint id) {
    point_t *point = (point_t*) handle;
    point->id = (int) id;
}
void point__get_name(jint handle, jbyte* buffer) {
    point_t *point = (point_t*) handle;
    strncpy((char*) buffer, point->name, SNI_getArrayLength(buffer));
}
void point__set_name(jint handle, jbyte* name) {
    point_t *point = (point_t*) handle;
    if (SNI_getArrayLength(name) > MAX_NAME_LENGTH) {
        SNI_throwNativeException(-1, "given name exceeds maximum length");
    }
    strncpy(point->name, (char*) name, MAX_NAME_LENGTH);
}
jint point__get_coordinate(jint handle, jint index){
    point_t *point = (point_t*) handle;
    return (jint) point->coordinates[(size_t)index];
}
void point__set_coordinate(jint handle, jint index, jint value) {
```

(continues on next page)

(continued from previous page)

```
    point_t *point = (point_t*) handle;
    point->coordinates[(size_t)index] = (int) value;
}
```

Note: For simplicity, this example does not include the usual error handling that real use cases will require.

Kernel & Features Specification (KF)

Introduction

The Kernel & Features (KF) specification defines the concepts for setting up Multi-Sandboxing of Applications, managing their lifecycle, and controlling their execution.

Specification Summary

Java APIs	ej.kf
Latest Version	1.7

Module Dependency	SDK 6
	SDK 5
	`implementation("ej.api:kf:1.7.0")`
	`<dependency org="ej.api" name="kf" rev="1.7.0" />`

Module Location	https://repository.microej.com/modules/ej/api/kf/

Comments

Your comments about this specification are welcome. Please contact *our support team* with "KF" as subject.

Basic Concepts

Kernel & Features semantic (KF) allows an application to be split into multiple parts:

- the main application, called the Kernel.
- zero or more applications, called Features.

The Kernel is mandatory. It is assumed to be reliable, trusted and cannot be modified. If there is only one application (i.e. only one main entry point that the system starts with) then this application is called the Kernel.

A Feature is an application "extension" managed by the Kernel. A Feature is fully controlled by the Kernel: it can be installed, started, stopped and uninstalled at any time independent of the system state (particularly, a Feature never depends on another Feature to be stopped). A Feature is optional, potentially not-trusted, maybe unreliable and can be executed without jeopardizing the safety of the Kernel execution and other Features.

Resources accesses (RAM, hardware peripherals, CPU time, …) are under control of the Kernel.

First Example

This simple example illustrates a log of a message called by a Kernel and a Feature. The KernelExample class is the main Kernel entry point. The FeatureExample class is a Feature entry point. The way these classes are assigned to contexts and how the Feature is installed is not described here. (the Feature is assumed to be installed before the Kernel main method starts).

Kernel class

Listing 7: Illustration 1: Kernel Hello World Example

```
package ej.kf.example.helloworld;

import ej.kf.Feature;
import ej.kf.Kernel;

/**
* Defines a Kernel class. The Kernel entry point is the regular main method.
*/
public class KernelExample {

  public static void main(String[] args) throws Exception {
    log("Hello World !");
    for (Feature f : Kernel.getAllLoadedFeatures()) {
      f.start();
    }
  }

  /**
  * Log a message, prefixed with the name of the caller
  */
  public static void log(String message) {
    String name = Kernel.getContextOwner().getName();
    System.out.println('[' + name + "]: " + message);
  }

}
```

Feature class

Listing 8: Illustration 2: Feature Hello World Example

```
package ej.kf.example.helloworld;

import ej.kf.FeatureEntryPoint;

/**
* Defines a Feature class that implements {@link FeatureEntryPoint} interface.
*/
public class FeatureExample implements FeatureEntryPoint {

  @Override
  public void start() {
```

(continues on next page)

(continued from previous page)

```
    KernelExample.log("Hello World !");
  }

  @Override
  public void stop() {
  }

}
```

Expected Output

```
[KERNEL]: Hello World !
[FEATURE]: Hello World !
```

Ownership Rules

At runtime, each type, object and thread execution context has an owner. This section defines ownership transmission and propagation rules.

Type

The owner of a type is fixed when such type is loaded and that owner cannot be modified after.

The owner of an array-of-type type is the owner of the type. Array of basetypes are lazily loaded. Those that are required by the Kernel are owned by the Kernel. Other arrays are loaded in any Feature that require them.

The owner of a type can be retrieved by calling Kernel.getOwner() with the Class instance.

Object

When an object is created, it is assigned to the owner of the execution context owner.

The owner of an object can be retrieved by calling Kernel.getOwner() with the given object.

Execution Context

When a thread is started, the first execution context is set to the owner of the thread object. When a method is called from *Kernel mode* and its receiver is owned by a Feature, the execution context is set to the owner of the receiver. In all other cases, the execution context of the method called is the execution context of the caller.

The owner of the current execution context can be retrieved by calling Kernel.getContextOwner().

When a method returns, the execution context owner of the caller remains the one it was before the call was done.

The Kernel is the first application to run, and it is triggered by the system when it boots. The Kernel starts in Kernel mode, creating a first thread owned by the Kernel.

The Kernel can execute a dynamic piece of code (Runnable) in a Feature context by calling Kernel.runUnderContext().

Kernel Mode

An execution context is said to be in *Kernel mode* when the current execution context is owned by the Kernel. The method Kernel.enter() sets the current execution context owner to the Kernel. The method Kernel.exit() resets the current execution context owner to the one when the method Kernel.enter() was called.

Execution Rules

Notes: this specification does not force all rules to be checked at runtime. When a rule is checked at runtime, a IllegalAccessError must be thrown at the execution point where the rule is broken.

Type References

A type owned by the Kernel cannot refer to a type owned by a Feature.

A type owned by a Feature can refer to a type owned by the Kernel if and only if it has been exposed as an API type.

A type owned by a Feature cannot refer to a type owned by another Feature.

All the types of the KF library (package $\mathrm{ej.kf.*}$) are owned by the Kernel. A type owned by a Feature cannot access any types of this library except the FeatureEntryPoint interface and the Proxy class.

Method References

A type owned by a Feature can refererence a method of type owned by the Kernel if and only if it has been exposed as an API method.

Field References

Instance Field References

A type owned by a Feature can refer to all instance fields of a type owned by the Kernel, if and only if the type has been exposed as an API type and the field is accessible according to Java access control rules.

Static Field References

A type owned by a Feature can refer to a static field of a type owned by the Kernel if and only if it has been exposed as an API static field.

A static field of a type owned by a Feature cannot refer to an object owned by another Feature.

An object owned by a Feature can be assigned to a static field of a type owned by the Kernel if and only if the current execution context is in *Kernel mode*, otherwise a IllegalAccessError is thrown at runtime.

Context Local Static Field References

By default, a static field holding an object reference is stored in a single memory slot in the context of the owner of the type that defines the field.

The Kernel can declare a static field as a context local storage field in `kernel.intern` file (See section *XML Schema & Format* for full format specification). A memory slot is then allocated for the Kernel and duplicated for each Feature. As it is a static field, it is initialized to `null`.

Listing 9: Illustration 3: Context Local Storage Declaration of a Static Field

```
<kernel>
  <contextLocalStorage name="com.mycompany.MyType.MY_GLOBAL"/>
</kernel>
```

The Kernel can declare an optional initialization method. This method is automatically invoked when the field is being read if its content is `null`. This gives a hook to lazily initialize the static field before its first read access. If the initialization method returns a `null` reference, a NullPointerException is thrown.

Listing 10: Illustration 4: Context Local Storage Declaration of a Static Field with an Initialization Method

```
<kernel>
  <contextLocalStorage
    name="com.mycompany.MyType.MY_GLOBAL"
    initMethod="com.mycompany.MyType.myInit()java.lang.Object"
  />
</kernel>
```

Object References

An object owned by a Feature cannot be assigned to an object owned by another Feature.

An object owned by a Feature can be assigned to an object owned by the Kernel if and only if the current execution context is in Kernel mode.

Note that all possible object assignments are included (field assignment, array assignment and array copies using System.arraycopy()).

Local References

An object owned by a Feature cannot be assigned into a local of an execution context owned by another Feature.

An object owned by a Feature can be assigned into a local of an execution context owned by the Kernel. When leaving Kernel mode explicitly with Kernel.exit(), all locals that refer to an object owned by another Feature are set to `null`.

Monitor Access

A method owned by a Feature cannot synchronize on an object owned by the Kernel.

A monitor entered by a thread while executing in one execution context cannot be re-entered by the same thread when executing in another execution context. For example, a monitor entered by a thread in the Kernel context cannot be re-entered by the same thread when executing in a Feature context.

Native Method Declaration

A class owned by a Feature cannot declare a `native` method.

Reflective Operations

Reflective operations enable dynamic access to Java elements. These operations must adhere to additional rules to maintain isolation semantics, based on the following parameters:

- Context Owner: The current *execution context* at the time the operation is invoked.
- Code Owner: The *owner of the class* that contains the method from which the operation is called.
- Type, Class, or Resource Owner: The owner of the target element being accessed by the operation.

Note: `N/A` indicates that it is not possible to be in Kernel mode within code owned by a Feature.

`Class.forName`

The following table defines the extended rules for Class.forName() to throw a ClassNotFoundException when a type cannot be accessed.

Table 4: Table 1: `Class.forName(...)` access rules

Context Owner	Code Owner	Type Owner	`Class.forName(Type)` allowed
K	K	K	true
K	K	F	false
K	F	K	N/A
K	F	F	N/A
F	K	K	true
Fi	K	Fj	i==j
F	F	K	`true` if the type has been exposed as an *API type*, `false` otherwise.
Fi	Fi	Fj	i==j

Class.newInstance

The following table defines the extended rules for Class.newInstance(). The last column indicates the owner of the newly created instance, if applicable.

Table 5: Table 2: Class.newInstance(...) access rules

Context Owner	Code Owner	Class Owner	New Instance Owner
K	K	K	K
K	K	F	F
K	F	K	N/A
K	F	F	N/A
F	K	K	F
F	K	F	F
F	F	K	F
F	F	F	F

Class.getResourceAsStream

The following table defines the extended rules for Class.getResourceAsStream() to return null when resource is not allowed to be accessed.

Table 6: Table 3: Class.getResourceAsStream(...) access rules

Context owner	Code owner	Resource owner	Class.getResourceAsStream(String) allowed
K	K	K	true
K	K	F	false
K	F	K	N/A
K	F	F	N/A
F	K	K	true
Fi	K	Fj	i==j If the same resource name is declared by both the Kernel and the Feature, the Feature resource takes precedence over the Kernel resource.
F	F	K	false
Fi	Fi	Fj	i==j

Thread.currentThread

Threads and their execution contexts have owners. The Thread.currentThread() method relates to the thread's owner that is executing the current execution context only. There is no obligation that two execution contexts that are in a caller-callee relationship have the same (==) returned java.lang.Thread object when using Thread.currentThread() method.

If the Thread that initiated the execution has the same owner as the current execution context or if execution is in Kernel mode, then the thread that initiates the execution is returned, otherwise, a java.lang.Thread object owned by the Kernel is returned.

Feature Lifecycle

Entry point

Each Feature must define an implementation of the FeatureEntryPoint. FeatureEntryPoint.start() method is called when the Feature is started. It is considered to be the main method of the Feature application. FeatureEntryPoint.stop() method is called when the Feature is stopped. It gives a chance to the Feature to terminate properly.

States

A Feature is in one of the following states:

- **INSTALLED**: Feature has been successfully linked to the Kernel and is not running. There are no references from the Kernel to objects owned by this Feature.
- **STARTED**: Feature has been started and is running.
- **STOPPED**: Feature has been stopped and all its owned threads and execution contexts are terminated. The memory and resources are not yet reclaimed. See section *Stop* for the complete stop sequence.
- **UNINSTALLED**: Feature has been unlinked from the Kernel.

The following illustration describes the Feature state diagram and the methods that changes Feature's state.

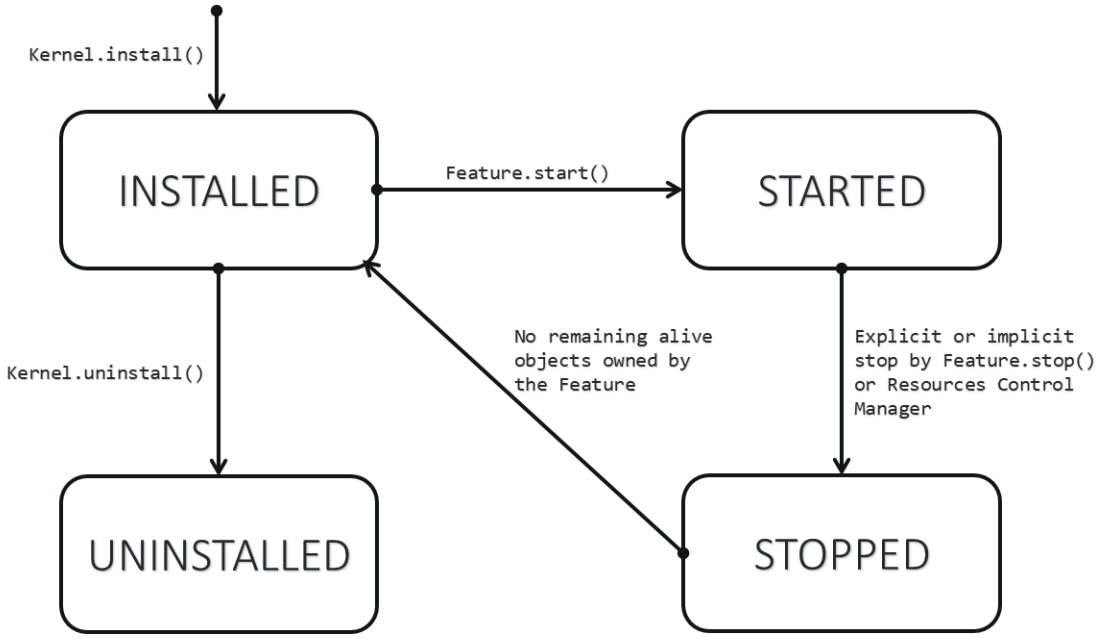


Fig. 114: Illustration 5: Feature State Diagram

Installation

A Feature is installed by the Kernel using Kernel.install(). The content of the Feature data to be loaded is implementation dependent. The Feature data is read and linked to the Kernel. If the Feature cannot be linked to the Kernel, an IncompatibleFeatureException is thrown. Otherwise, the Feature is added to the list of loaded Features and its state is set to the INSTALLED state.

Start

A Feature is started by the Kernel using Feature.start(). The Feature is switched in the STARTED state. A new thread owned by the Feature is created and started. Next steps are executed by the newly created thread:

- Feature clinits are executed.
- Entrypoint is instanciated.
- FeatureEntryPoint.start() is called.

Stop

A Feature is stopped explicitly by the Kernel using Feature.stop(). Features may be stopped implicitly by the Resource Control Manager. Next steps are executed:

- On explicit Feature.stop() call, a new thread owned by the Feature is created and FeatureEntryPoint.stop() is executed within this new thread.
- Wait until this new thread is done, or until a global timeout stop-time occurred[1].

[1] The default timeout stop-time is 2,000ms.

- All execution contexts, from any thread, owned by the Feature are cleared, which implies that a DeadFeatureException is thrown in threads that are running the stopped Feature code or in threads that want to call stopped Feature code.
- Wait until all threads owned by the Feature are terminated.
- Native resources (files, sockets, …) opened by the Feature that remain opened after FeatureEntryPoint.stop() execution are closed abruptly.
- The Feature state is set to the STOPPED state.
- FeatureStateListener.stateChanged() is called for each registered listener.
- Objects owned by the Feature are reclaimed.
- If there are no remaining alive objects[2]:
  - Feature state is set to the INSTALLED state.
  - FeatureStateListener.stateChanged() is called for each registered listener.

The method Feature.stop() can be called several times, until the Feature is set to the INSTALLED state.

Uninstallation

A Feature is uninstalled by the Kernel using Kernel.uninstall(). The Feature code is unlinked from the Kernel and reclaimed. The Feature is removed from the list of loaded Features and its state is set to the UNINSTALLED state. The Feature does not exist anymore in the system.

Class Spaces

[2] If there are any remaining alive Feature objects after the Kernel listeners have been called, the Feature will stay in the STOPPED state indefinitely. The Kernel has an issue. However, it can continue running and orchestrating other applications, but it cannot restart or uninstall the problematic Feature.

Overview

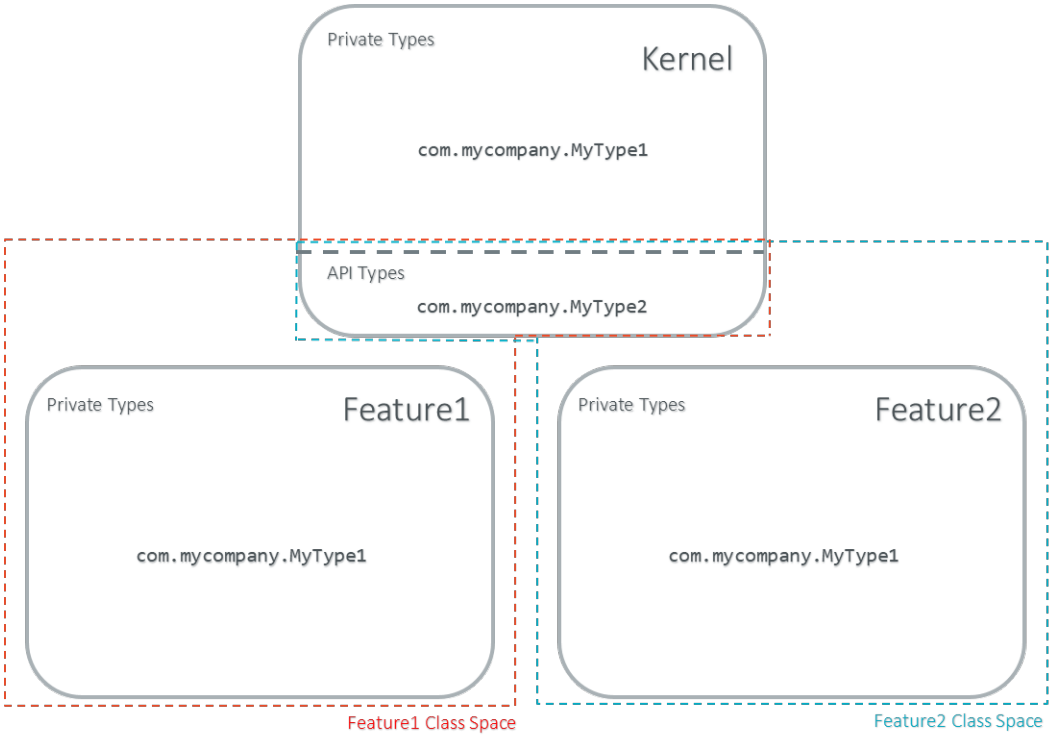


Fig. 115: Illustration 6: Kernel & Features Class Spaces Overview

Private Types

The Kernel and the Features define their own private name space. Internal types are only accessible from within the Kernel or Features that define these types. The Kernel or a Feature can have only one type for a specific fully qualified name, insuring there are not two types in the Kernel or in a Feature sharing the same fully qualified name.

Kernel API Types

The Kernel can expose some of its types, methods and static fields as API to Features. A file describes the list of the types, the methods and the static fields that Features can refer to.

Here is an example for exposing System.out.println(String) to a Feature:

Listing 11: Illustration 7: Kernel API Example for exposing System.out.println

```
<require>
  <field name="java.lang.System.out"/>
  <method name="java.io.PrintStream.println(java.lang.String)void"/>
</require>
```

Section *Kernel API Definition* describes the Kernel API file format.

Precedence Rules

APIs exposed by the Kernel are publicly available for all Features: they form the global name space.

A Kernel API type (from the global name space) always takes precedence over a Feature type with the same fully qualified name when a Feature is loaded. An type exposed by the Kernel cannot be overloaded by a Feature.

Resource Control Manager

CPU Control: Quotas

A Kernel can assign an execution quota to a Feature using Feature.setExecutionQuota(). The quota is expressed in execution units.

Quotas account to the context of the current thread's owner.

When a Feature has reached its execution quota, its execution is suspended until all other Features have reached their execution quota. When there are no threads owned by Features eligible to be scheduled, the execution counter of all Features is reset.

Setting a Feature execution quota to zero causes the Feature to be suspended (the Feature is paused).

If a Feature exceeds its execution quota while holding a monitor (via one of its threads), and another Module (Feature or Kernel) with no execution quota limit attempts to acquire the same monitor (via one of its threads), the thread holding the monitor will continue its execution until it releases the monitor.

RAM Control: Managed Heap Configuration

The Kernel and Features allocate objects in the Managed Heap, whose size is statically configured by the Kernel using the *Option(text): Managed Heap size (in bytes)*. By default, the Kernel and Features can allocate without restriction, as long as sufficient space is available in the Managed Heap. Every allocated object is tracked (see *Object ownership*).

The following APIs allow to configure the Managed Heap usage:

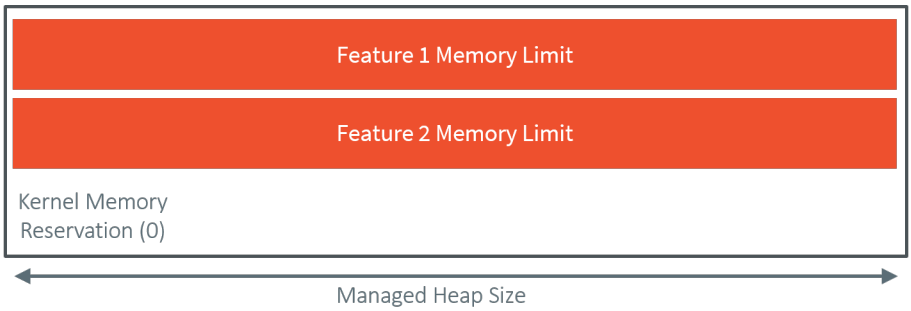
- Kernel.setReservedMemory(): Sets the amount of memory heap *reserved* for the Kernel. This is a lower bound: the Kernel may allocate more memory than this reservation if needed.
- Feature.setMemoryLimit(): Sets the *maximum* amount of memory heap that can be allocated by a Feature. This is an upper bound, not a reservation.

Especially, allocations from Features will fail (throw an OutOfMemoryError):

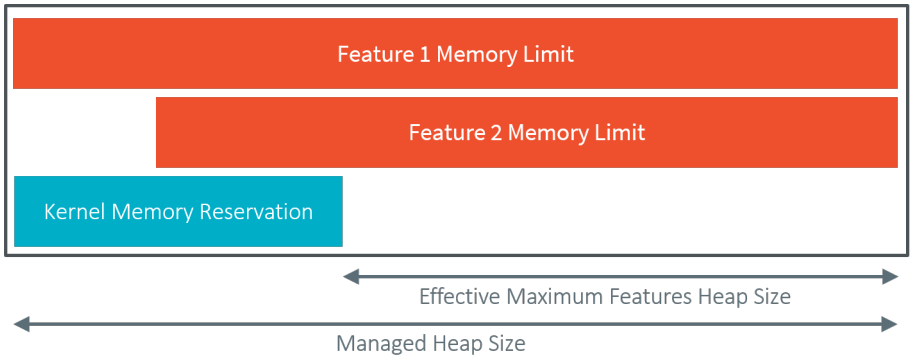
- when the feature-specific limit is reached, before the Managed Heap is full;
- when the Managed Heap is full, before the feature-specific limit is reached;
- when the allocation would exceed the amount of memory available to Features (i.e. Managed Heap size minus memory reserved for the Kernel).

The diagrams below illustrate a Kernel with two Features with various heap configuration scenarios. Kernel reserved memory and Feature memory limits are depicted as contiguous blocks for simplicity. However, in reality, they represent the total size of allocated objects, which are scattered throughout the Managed Heap.

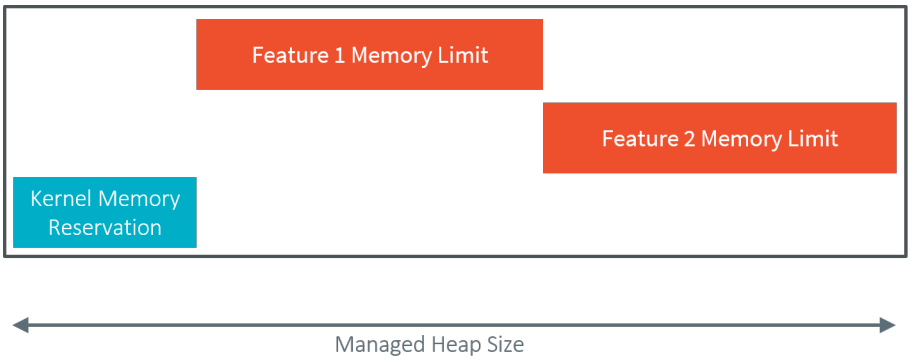
Case 1: No Kernel memory reservation, no Feature limit (Default configuration)



Case 2: Kernel memory reservation, one Feature with no memory limit, one Feature with a memory limit that overlaps the Kernel's reserved memory.



Case 3: Kernel memory reservation, with two Features configured to equally share the remaining heap



RAM Control: Feature Criticality

Each Feature has a criticality level between Feature.MIN_CRITICALITY and Feature.MAX_CRITICALITY. When an execution context cannot allocate new objects because a memory limit has been reached, Features shall be stopped following next semantic:

- Select the Feature with the lowest criticality.
- If the selected Feature has a criticality lower than the current execution context owner criticality, then stop the selected Feature and all the Features with the same criticality.
- If no memory is available, repeat these two previous steps in sequence until there are no more Features to stop.

If no memory is reclaimed, then an OutOfMemoryError is thrown.

See *Core Engine RAM Control implementation* for more details.

Time-out Control: Watchdog

All method calls that are done from a Kernel mode to a Feature mode are automatically executed under the control of a watchdog.

The watchdog timeout is set according to the following rules:

- use the watchdog timeout of the current execution context if it has been set,
- else use the watchdog timeout of the current thread if it has been set,
- else use the global system watchdog timeout.

The global system watchdog timeout value is set to Long.MAX_VALUE at system startup.

When the watchdog timeout occurs the offending Feature is stopped.

Native Resource Control: Security Manager

The Kernel is responsible for holding all the native calls. The Kernel shall provide methods (API) that systematically check, using the standard security manager, that the access to a native call is granted to the specific Feature.

When an object owned by a Feature is not allowed to access a native resource, a specific exception shall be thrown.

Any native resource opened by a Feature must be registered by the Kernel and closed when the Feature is stopped.

Communication Between Features

A Feature can communicate with another Feature using *Shared Interfaces*. This section explains the execution semantics and advanced configuration from the Kernel's perspective.

Method Binding

A Feature can call a method owned by another Feature, provided:

- Both Features own an interface in their class space with the same fully qualified name.
- Both Features have declared such interface as a Shared Interface.
- The source Feature has declared a Proxy class for its Shared Interface.
- The target Feature has registered to the Kernel an instance of a class implementing its Shared Interface.
- The source Feature has requested from the Kernel an instance of a class implementing its interface.
- The Kernel has bound the source interface to the target instance and returned an instance to the source Feature, implementing its Shared Interface.
- The source Feature calls a method declared in the Shared Interface using this instance as receiver.
- A method with the exact descriptor exists in the target Feature interface.
- The arguments given by the source Feature can be transferred to the target Feature.
- The value returned by the target Feature can be transferred to the source Feature (if the method does not return `void`).

Section *Shared Interface Declaration* describes the Shared Interface file format specification.

Object Binding

An object owned by a Feature can be bound to an object owned by another Feature using the method Kernel.bind().

- When the target type is owned by the Kernel, the object is converted using the most accurate *Kernel type converter*.
- When the target type is owned by the Feature, it must be a Shared Interface. In this case, an instance of its *Proxy class* is returned.

Object identity is maintained across Features, so the same proxy instance is returned. If a Proxy is bound to the Feature that owns the reference, the original object is passed instead (Proxy unwrapping).

Note: The Kernel can manually bind an object using the Kernel.bind() method.

Kernel Type Converters

By default, Feature instances of types owned by the Kernel cannot be passed across a Shared Interface method invocation.

The Kernel can register a converter for each allowed type, using Kernel.addConverter(). The converter must implement Converter and can implement one of the following behaviors:

- by wrapper: manually allocating a Proxy reference by calling Kernel.newProxy().
- by copy: with the help of Kernel.clone().

Configuration Files

Kernel and Features Declaration

A Kernel must provide a declaration file named `kernel.kf`. A Feature must provide a declaration file named `[name].kf`.

KF Declaration file is a Properties file. It must appear at the root of any application classpath (directory or JAR file). Keys are described hereafter:

Table 7: Illustration 10: KF Definition File Properties Specification

Key	Usage	Description
entryPoint	Mandatory for Feature only.	The fully qualified name of the class that implements FeatureEntryPoint
name	Optional	`KERNEL` by default for the Kernel, or the name of the file without the `.kf` extension for Features.
version	Mandatory	String version, that can retrieved using Module.getVersion()

SDK 6 1.5.0 and above

SDK 6 1.4.0 and below

SDK 5

Using SDK 6 1.5.0 and above, `.kf` files are automatically generated. The properties can be customized in your `build.gradle.kts` file, for example in a feature:

```
microej {
  applicationName = "myFeature"
  applicationVersion = "1.0.0"
}
```

Using SDK 6 1.4.0 and below, `.kf` files are automatically generated. Their properties can be changed by creating the `.kf` files in the `src/main/resources` folder of the project.

Using SDK 5, `.kf` files are located in the `src/main/resources` folder of the project and can be modified if needed.

Kernel API Definition

By default, when building a Kernel, no types are exposed as API for Features, except FeatureEntryPoint. Kernel types, methods and static fields allowed to be accessed by Features must be declared in one or more `kernel.api` files. They must appear at the root of any application classpath (directory or JAR file). Kernel API file is an XML file, with the following schema:

Listing 12: Illustration 11: Kernel API XML Schema

```
<xs:schema xmlns:xs='http://www.w3.org/2001/XMLSchema'>
   <xs:element name='require'>
      <xs:complexType>
         <xs:choice minOccurs='0' maxOccurs='unbounded'>
            <xs:element ref='type'/>
            <xs:element ref='field'/>
            <xs:element ref='method'/>
         </xs:choice>
```

(continues on next page)

(continued from previous page)

```
    </xs:complexType>
  </xs:element>

  <xs:element name='type'>
    <xs:complexType>
      <xs:attribute name='name' type='xs:string' use='required'/>
    </xs:complexType>
  </xs:element>

  <xs:element name='field'>
    <xs:complexType>
      <xs:attribute name='name' type='xs:string' use='required'/>
    </xs:complexType>
  </xs:element>

  <xs:element name='method'>
    <xs:complexType>
      <xs:attribute name='name' type='xs:string' use='required'/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Table 8: Illustration 12: Kernel API Tags Specification

Tag	Attributes	Description
require		The root element
field		Static field declaration. Declaring a field as a Kernel API automatically declares its type as a Kernel API.
name	Fully qualified name on the form [type].[fieldName]	
method		Method or constructor declaration. Declaring a method or a constructor as a Kernel API automatically declares its type as a Kernel API
name	Fully qualified name on the form [type].[methodName]([typeArg1,..., typeArgN)typeReturned . Types are fully qualified names or one of a base type as described by the Java language (boolean , byte , char , short , int , long , float , double) When declaring a constructor, methodName is the single type name. When declaring a void method or a constructor, typeReturned is void	
type		Type declaration. Declaring a type as Kernel API automatically declares all its super types (classes and interfaces) and the default constructor (if any) as Kernel API.
name	Fully qualified name on the form [package].[package].[typeName]	

Identification

Kernel and Features identification is based on a X509 certificate. The 6 first fields defined by RFC 2253 (CN : commonName, L : localityName, ST : stateOrProvinceName, O : organizationName, OU : organizationalUnitName, C : countryName) can be read by calling ej.kf.Module.getProvider().getValue(...) .

With SDK 6, these fields can be configured in the build.gradle.kts file of the project, in the *microej* > *applicationCertificate* block. The certificate is automatically generated with the given values. Here is an example with all the available fields and their default value:

```
microej {
  applicationCertificate {
    commonName = "myFeature"
    organizationalUnitName = "myCompanyUnit"
    organizationName = "myCompany"
    localityName = "myLocality"
    stateOrProvinceName = "myStateOrProvince"
    countryName = "myCountry"
  }
}
```

The certificate file must be configured as follows:

```
- placed beside the related ``[name].kf`` file.
```

- named [name].cert .
- DER -encoded and may be supplied in binary or printable (Base64) encoding. If the certificate is provided in Base64 encoding, it must be bounded at the beginning by -----BEGIN CERTIFICATE----- , and must be bounded at the end by -----END CERTIFICATE----- .

To be noted that dropping the certificate file in the project sources is the only available solution with SDK 5.

Shared Interface Declaration

A Shared Interface file is an XML file ending with the .si suffix with the following format:

Listing 13: Illustration 13: Shared Interface XML Schema Specification

```
<xs:schema xmlns:xs='http://www.w3.org/2001/XMLSchema'>

<xs:element name='sharedInterfaces'>
  <xs:complexType>
  <xs:choice minOccurs='0' maxOccurs='unbounded'>
    <xs:element ref='sharedInterface'/>
  </xs:choice>
  </xs:complexType>
</xs:element>

<xs:element name='sharedInterface'>
  <xs:complexType>
  <xs:attribute name='name' type='xs:string' use='required'/>
  </xs:complexType>
</xs:element>

</xs:schema>
```

Kernel Advanced Configuration

The kernel.intern files is for Kernel advanced configurations such as declaring *context local storage static fields*. It must appear at the root of any application classpath (directory or JAR file).

Listing 14: Illustration 14: Kernel Intern Root XML Schema Specification

```
<!--
  Root Element
  -->
<xs:element name='kernel'>
  <xs:complexType>
  <xs:choice minOccurs='0' maxOccurs='unbounded'>
    <xs:element ref='contextLocalStorage'/>
    <xs:element ref='property'/>
  </xs:choice>
  </xs:complexType>
</xs:element>
```

Context Local Storage Static Field Configuration

XML Schema & Format

Listing 15: Table 5: Context Local Storage XML Schema Specification

```
<xs:element name='contextLocalStorage'>
  <xs:complexType>
  <!--
    Static Field Simple Name.
  -->
  <xs:attribute name='name' type='xs:string' use='required'/>
  <!--
    Optional Initialization Method descriptor, as specified by Kernel API method descriptor.
    -->
  <xs:attribute name='initMethod' type='xs:string' use='optional'/>
  </xs:complexType>
</xs:element>
```

Typical Example

The following illustration describes the definition of a context local storage static field (I), which is duplicated in each context (Kernel and Features):

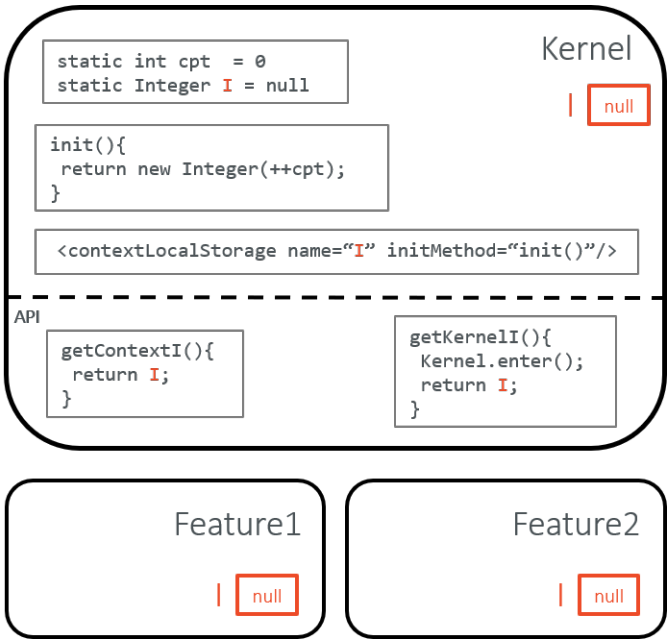


Fig. 116: Illustration 15: Context Local Storage of Static Field Example

The following illustration describes a detailed sequence of method calls with the expected behavior.

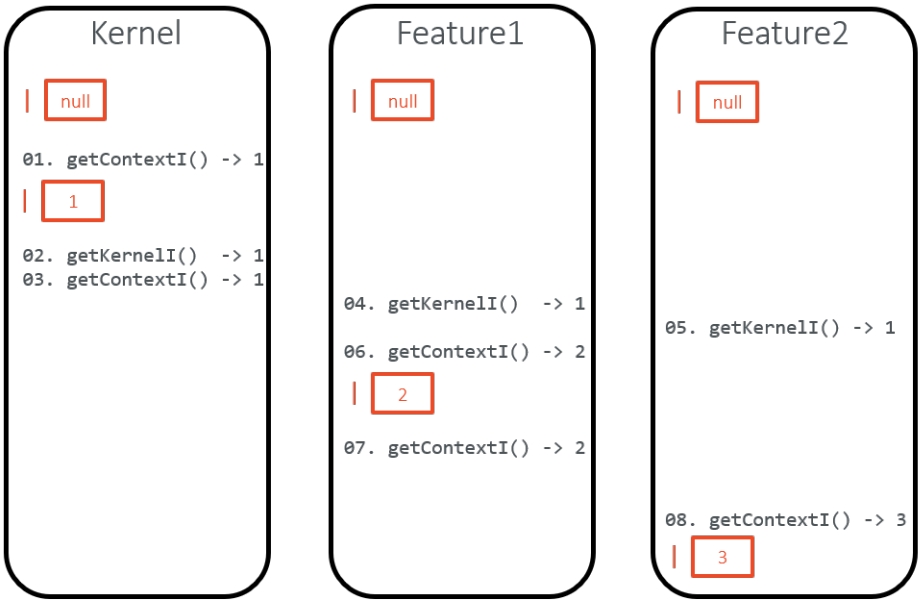


Fig. 117: Illustration 16: Context Local Storage Example of Initialization Sequence

Scheduler

The Core Engine implements a multi-threaded environment without relying on native OS capabilities. It includes a built-in scheduler and adopts a Green Threads model, enabling full control over thread management. This design ensures that the scheduling policy is entirely portable and remains independent of the underlying RTOS or operating system on which the Core Engine runs.

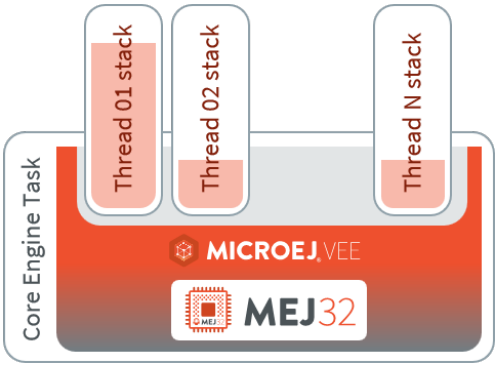


Fig. 118: Green Threads Model Overview

For more details refer to the *Core Engine Theading Integration* section.

Scheduling Policy

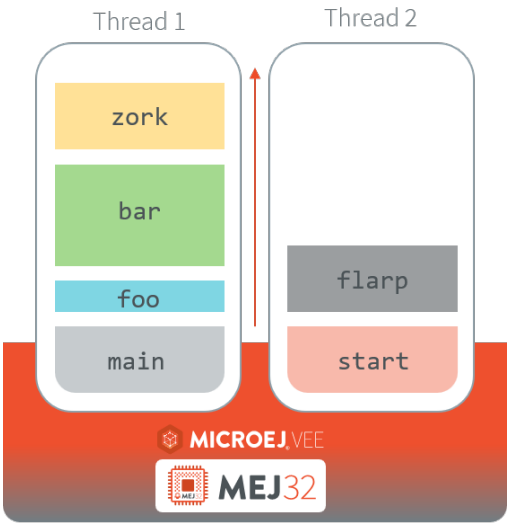
The Core Engine defines the following scheduling semantic:

- preemptive for different priorities,
- round-robin for same priorities,
- "priority inheritance protocol" when priority inversion occurs. This protocol raises the priority of a thread that is holding a monitor needed by a higher-priority thread, to the priority of that higher-priority thread (until exiting the monitor).

Threads and Stacks Allocation

A thread requires a stack to execute. The stack holds an ordered list of call frames (one for each method currently called). Each call frame contains runtime data for the corresponding method invocation, including slots for local variables and parameters. The size of a call frame depends on the method's content and complexity.

The following figure illustrates an example of thread execution, showing each thread with its associated stack of call frames:



Core Engine Memory Allocation

Memory for threads and their stacks is statically defined at link time. The Application developer is responsible for configuring these sizes. This design ensures that thread creation and stack operations remain lightweight and avoid any reliance on dynamic memory allocation or underlying RTOS/OS mechanisms at runtime.

- Thread Allocation: The maximum number of threads is specified via the *maximum number of threads option*.
- Stack Allocation: Each thread stack is composed of fixed-size blocks, which are allocated from a shared pool. This approach ensures efficient memory reuse and eliminates fragmentation during execution. The Application developer defines the *number of blocks in the pool* and the *size of each stack block*.

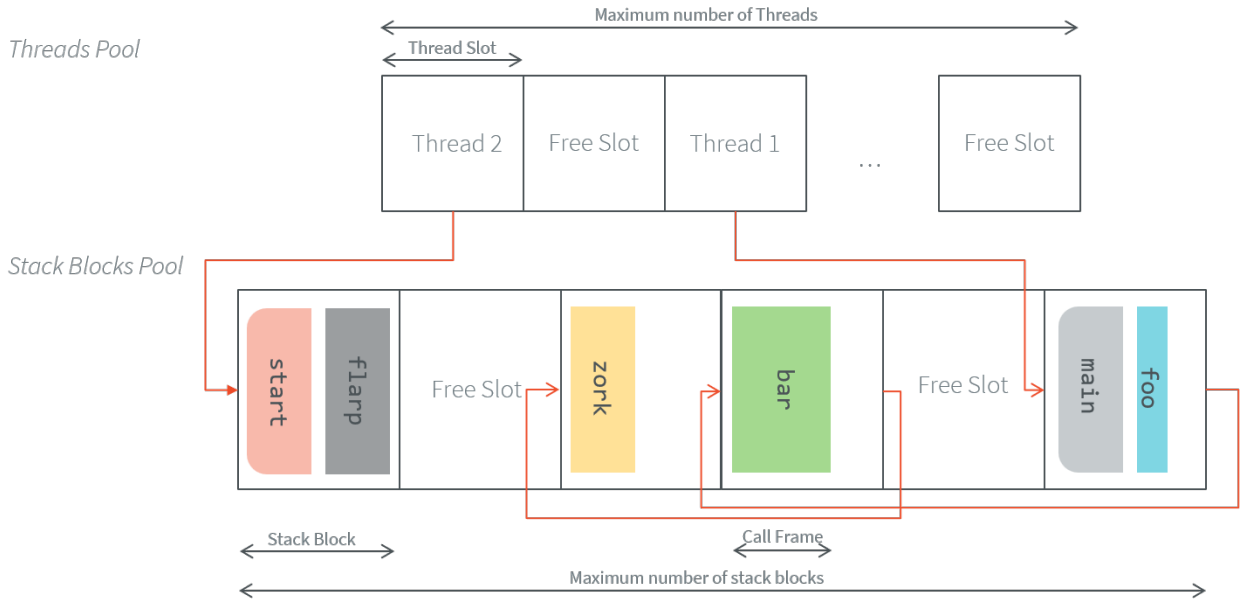
Additionally, the Application developer can configure the *maximum number of stack blocks per thread*, enforcing an upper bound on the stack memory consumption of individual threads. This mechanism prevents a single thread from consuming all available stack blocks, particularly in case of unbounded or deep recursive method calls.

Thread and Call Stack Behavior

A Core Engine thread slot is only assigned when the thread is started by calling the Thread.start() method. At that point, it is assigned one stack block from the stack pool to begin execution. Each method call pushes a call frame onto the stack. If a call frame cannot fit into the currently allocated stack block, the thread acquires an additional block from the pool. Stacks dynamically grow by acquiring more blocks, as needed by the call depth. However, once a block is allocated to a thread, it remains associated with that thread until it is terminated, even if the thread's stack depth decreases.

A thread remains active as long as it is executing. Its status can be checked by calling the Thread.isAlive() method for example. Once a thread is terminated (either by returning from its Runnable.run() method or due to an uncaught exception), its thread slot and all associated stack blocks are released. The released thread slot and stack blocks can then be used by other threads.

The following figure illustrates a possible memory layout for the previously described thread example, showing how call frames can span across multiple fixed-size stack blocks within each thread's stack:



Thread and Stack Memory Errors

If there is insufficient memory to start a thread using the Thread.start() method, the Core Engine may throw one of the following exceptions:

- `InternalLimitsError : too many java alive thread` : The maximum number of alive threads is reached. Check the *maximum number of threads option*.
- `OutOfMemoryError: Stack space` : The stack pool is exhausted and no additional block can be allocated. Check the *number of stack blocks option* or the *stack block size option*.

If a method call requires more stack space than is available, the following exceptions may be thrown:

- `StackOverflowError: EDC-1.3:E=-1` : The stack pool is exhausted and no additional block can be allocated. Check the *number of stack blocks option* or the *stack block size option*.
- `StackOverflowError: EDC-1.3:E=-2` : The next call frame is too large to fit in a single stack block. Check the *stack block size option*.
- `StackOverflowError: EDC-1.3:E=-3` : The thread has reached the maximum number of stack blocks it is allowed to use. Check the *the maximum number of blocks per thread option*.

Managed Heap & Garbage Collector

The Core Engine includes a state-of-the-art memory management system, the Garbage Collector (GC). It manages a bounded piece of RAM memory, devoted to the Managed world. The GC automatically frees dead Java objects, and defragments the memory in order to optimize RAM usage. This is done transparently while the Application keep running.

See also *Garbage Collector options* for more details.

Death Notification

Most objects are reclaimable objects. Sometimes, they interact with the native or system resources using handles. Those handles represent underlying data that needs to be closed/freed/acknowledged/… when the object that holds the handle dies.

The Application can get notified when an object is dead through the use of WeakReference objects. When such objects get their weak reference set to `null` by the system, they are added to the ReferenceQueue they were assigned to at their creation.

Death Notification Actions

Once an object has expired, it cannot be brought to life again. It is the responsibility of the application to make provisions for all actions that have to be taken on an object death. Such provisions are materialized by subclasses of the WeakReference class.

ReferenceQueue.poll() and ReferenceQueue.remove() methods allow the execution of a hook at the death of the object referenced by the weak reference. The first one returns `null` when queue is empty whereas the second one blocks while the queue is empty.

The Application is responsible of the execution of such hook.

Limitations

Primitive Types

Getting a Class instance of a primitive type is not supported:

- `boolean.class`,
- `byte.class`,
- `char.class`,
- `short.class`,
- `int.class`,
- `long.class`,
- `float.class`,
- `double.class`.

On Architecture `8.x`, you will get the following dedicated error message:

```
SOAR-L ERROR :
[M79] - Unsupported access to the Class instance of a primitive type (found 'boolean.class' in method 'com.
↪mycompany.MyClass.myMethod()void')
```

On Architecture `7.x` you will get the following default error message:

```
No such field TYPE at com/mycompany/MyClass.myMethod()V.
```

Architecture Characteristics

The Application can retrieve some characteristics of the Architecture on which it is running. Architecture characteristics are automatically provided as *constants*. Here are the most notable ones:

- com.microej.architecture.capability=[tiny|single|multi] : *Core Engine Capability*
- com.microej.architecture.name=[architecture_uid] : Architecture name.
- com.microej.architecture.level=[eval|prod] : Usage level (Evaluation or Production).
- com.microej.architecture.toolchain=[toolchain_uid] : Toolchain name.
- com.microej.architecture.version=[M.m.p] : Architecture version.

See also *Architecture Naming Convention* for more details.

The following code prints the formatted Architecture characteristics on standard output. You can copy-paste and adapt it to your needs.

```
String name = Constants.getString("com.microej.architecture.name");
String version = Constants.getString("com.microej.architecture.version");
String buildLabel = Constants.getString("com.microej.architecture.buildLabel");

String usage = Constants.getString("com.microej.architecture.level");
String usageStr;
if (usage.equals("prod") || usage.equals("dev")) {
  usageStr = "Production";
} else if (usage.equals("eval")) {
  usageStr = "Evaluation";
} else {
  usageStr = usage;
}

String capability = Constants.getString("com.microej.architecture.capability");
String capabilityStr;
if (capability.equals("multi")) {
  capabilityStr = "Multi";
} else if (capability.equals("tiny")) {
  capabilityStr = "Tiny";
} else if (capability.equals("single") || capability.equals("mono")) {
  capabilityStr = "Mono";
} else {
  capabilityStr = capability;
}

String isaStr = Constants.getString("com.microej.architecture.architecturePrintableName");
String toolchainName = Constants.getString("com.microej.architecture.toolchainPrintableName");
String toolchainFullName = Constants.getString("com.microej.architecture.toolchain");

System.out.println("- Name:                          " + name);
System.out.println("- Version:                       " + version + " (" + buildLabel + ")");
System.out.println("- Usage:                         " + usageStr);
System.out.println("- Core Engine Capability:        " + capabilityStr + "-Sandbox");
System.out.println("- Instruction Set Architecture: " + isaStr);
System.out.println("- Compilation Toolchain:         " + toolchainName + " (" + toolchainFullName + ")");
```

3.5.2 SOAR

This chapter describes SOAR capabilities and optimizations from the Application developer's point of view. To get more details on its internal structure, please refer to *SOAR Build Phases* section.

Java Symbols Encoding

Java symbols are any of package, type, method, field, or local names. In `.class` files, they are encoded in `UTF-8`. However, SOAR only supports Java symbols composed of characters that can be stored on 8 bits (unsigned byte).

This is typically the case of ISO-8859-X encoding family.

If you try to build an Application that includes an unsupported Java symbol you will get the following error:

```
Unsupported Java symbol XXX in file YYY. A character cannot be stored on an unsigned byte.
```

Note: Classpath `*.list` files are standard Java properties files that are encoded in `ISO-8859-1` (Latin-1). If you need to refer to a Java Symbol that contains a character out of this charset, you must declare the character using the `\uHHHH` notation where HHHH is the hexadecimal index of the character in the Unicode character set.

Class Initialization Code

SOAR complies with the deterministic class initialization (`<clinit>`) order specified in *[BON] specification*. The application is statically analyzed from its entry points in order to generate a clinit dependency graph. The computed clinit sequence is the result of the topological sort of the dependency graph. An error is thrown if the clinit dependency graph contains cycles.

A clinit map file (ending with extension `.clinitmap`) is generated beside the SOAR object file. It describes for each clinit dependency:

- the types involved
- the kind of dependency
- the stack calls between the two types

In case of complex clinit dependencies graph, the SOAR may detect static cycles (circular dependencies) and fail with an error. In such case, you have to manually cut-off the cycles, by providing the explicit clinit dependencies.

Explicit clinit dependencies are declared in XML files ending with the `.clinitdesc` extension, at the root of a library or application classpath.

The file has the following format:

```
<?xml version='1.0' encoding='UTF-8'?>
<clinit>
   <type name="T1" depends="T2"/>
</clinit>
```

where `T1` and `T2` are fully qualified names on the form `a.b.C`. This explicitly forces the SOAR to create a dependency from `T1` to `T2`, and therefore cuts a potentially detected dependency from `T2` to `T1`.

Method Devirtualization

Method devirtualization consists of transforming a virtual method call to a direct method call when possible. A virtual method call is a call to a non-private instance method declared either in an interface or in a class. The Core Engine determines the proper method to call at runtime depending on the actual class of the object. A call to a constructor or a private method is already optimized as a direct method call by the Java compiler.

SOAR automatically optimizes a virtual method call to a direct method call if there is one and only one embedded implementation method.

Note: SOAR generates the list of the embedded methods in the *SOAR Information File*.

Method Inlining

Method inlining consists of replacing a direct method call with the content of the method. This avoids the creation of a new call frame, which can be slower than executing the code itself. Method inlining is transitively applied from leaf to root methods.

The following method code patterns are inlined:

- empty code after processing *assertions* and *if code removal*.
- call to a constructor with no parameters.
- call to a private method with no parameters.
- call to a static method with no parameters, if and only if the caller is also a static method.

Note: Method inlining is performed after *method devirtualization*, so a virtual method call will be inlined if there is a unique embedded implementation method that matches one of the inlined method code patterns.

Binary Code Verifier

The Binary Code Verifier is the tool that scrutinizes the bytecode instructions for adherence to strict rules and constraints. This process is crucial for preventing runtime errors, security vulnerabilities, and unexpected behavior. It ensures that code loaded by the SOAR is in a consistent state before being linked. Consequently, this guarantees the safe execution of the code by the Core Engine.

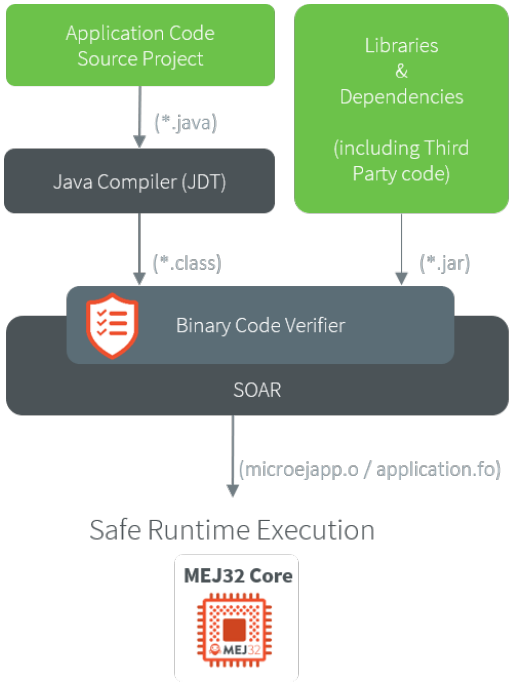


Fig. 119: Application Build Flow with Binary Code Verifier

The Binary Code Verifier performs tasks including:

- Type Checking: Verifying that variables and operands are used in a manner consistent with their declared data types, preventing type-related errors at runtime.
- Bytecode Structure: Ensuring the bytecode is well-formed and follows the structure required by the JVM, which helps prevent memory corruption and crashes.
- Stack Management: Checking that the operand stack used for calculations and evaluations is properly managed to prevent stack overflows or underflows.
- Access Control: Verifying that class accesses and method invocations adhere to Java's access control rules, ensuring data encapsulation and security.
- Exception Handling: Validating that exception handlers are correctly defined and that exceptions are caught and handled appropriately.
- Control Flow: Analyzing the flow of control within bytecode to detect anomalies in loops, branches, and jumps that could lead to program instability.

A default implementation, derived from the Apache BCEL Project, is included in the SOAR. If you wish to integrate an alternative implementation, contact our support team for access to the SOAR interface API and integration instructions.

Note: The Binary Code Verifier is enabled by default when building a Sandboxed Application, and disabled by

default when building a Standalone Application. See *Option(checkbox): Enable Bytecode Verifier* for more details.

Enable SOAR Verbose Logs

SOAR logs can be enabled by setting the `execution.verbose` Application Option to `true`. As for *Application Options*, with SDK 6 the system property must be prefixed with `microej.option`.

For instance, to enable SOAR logs from command line:

```
-Dmicroej.option.execution.verbose=true
```

or from `gradle.properties` file:

```
systemProp.microej.option.execution.verbose=true
```

You should see SOAR logs on tasks `buildExecutable`, `buildFeature` and `buildFeatureFromWPK`. For instance:

```
...
[SOAR-L] Loaded classpath entry /projectpath/build/classes/java/main
[SOAR-L] Loaded Feature Main from file /projectpath/build/generated/microej-app-wrapper/feature-resources/
↪feature.kf
[SOAR-L] Added type org.example.project.MainWrapper to be loaded (type is referenced by the file /projectpath/
↪build/generated/microej-app-wrapper/feature-resources/feature.kf)
[SOAR-L] Added immutables entry com.is2t.microui.fonts.data (type is referenced by the option 'rootImmutables')
[SOAR-L] Try the load of com.is2t.microui.fonts.data from directory /projectpath/build/classes/java/main
...
[SOAR-S] Selected type org.example.project.MainWrapper
[SOAR-S] Selected method org.example.project.MainWrapper.start()void
[SOAR-S] Selected immutable 'ej.microui.fonts.list'
[SOAR-S] Generated map file to /projectpath/build/application/feature/feature/application.selectormap
[SOAR-S] Generated shared object file to /projectpath/build/application/feature/feature/application.so
INFO: Generated Kernel metadata file to /Users/bguedas/tmp/projects/test_junit/build/kernelExecutable/kernel.
↪kdat
[SOAR-O] Generated file to /projectpath/build/application/feature/feature/application.optimizermap
[SOAR-O] Generated file to /projectpath/build/application/feature/feature/application.map
[SOAR-O] Generated object file to /projectpath/build/application/feature/feature/application.o
...
```

3.5.3 SOAR Output Files

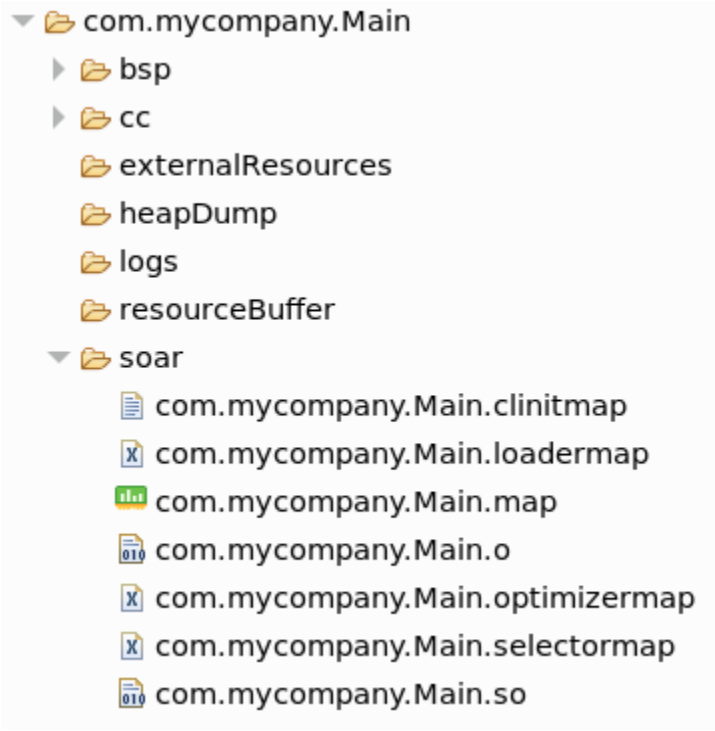
When building a Standalone Application, multiple files are generated next to the ELF executable file.

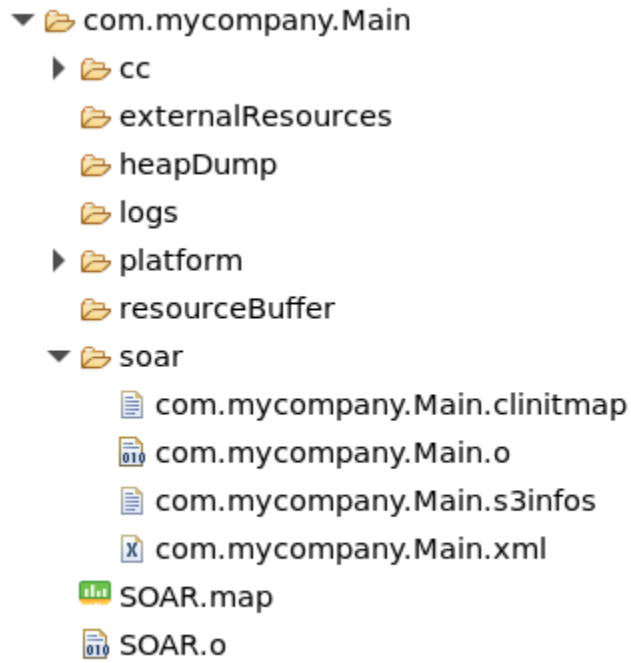
Launch Output Folder

Using a *MicroEJ Application Launch*, the files are generated in a folder which is named like the main type and which is located in the output folder specified in the run configuration.

Build Output Files (Architecture `8.x`)

Build Output Files (Architecture `7.x`)


com.mycompany.Main
bsp
cc
externalResources
heapDump
logs
resourceBuffer
soar
com.mycompany.Main.clinitmap
com.mycompany.Main.loadermap
com.mycompany.Main.map
com.mycompany.Main.o
com.mycompany.Main.optimizermap
com.mycompany.Main.selectormap
com.mycompany.Main.so


com.mycompany.Main
cc
externalResources
heapDump
logs
platform
resourceBuffer
soar
com.mycompany.Main.clinitmap
com.mycompany.Main.o
com.mycompany.Main.s3infos
com.mycompany.Main.xml
SOAR.map
SOAR.o

Published Module Files

After *building* the Standalone Application, the published module contains the following main files:

- [name]-[version].out : Firmware (ELF Executable)
- [name]-[version].zip : Virtual Device
- [name]-[version]-workingEnv.zip : Build intermediate files, including the content of the launch output Folder

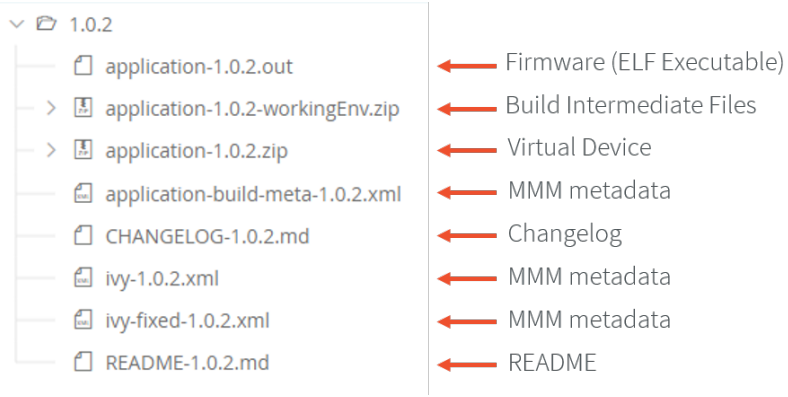


Fig. 120: Published Standalone Application Module Files

The SOAR Map File

The .map file lists every embedded symbol of the application (section, Java class or method, etc.) and its size in ROM or RAM. Since Architecture 8.x , this file is called <main class>.map . It was formerly named SOAR.map for Architecture 7.x . This file can be opened using the *Memory Map Analyzer*.

The embedded symbols are grouped into multiple categories. For example, the Object class and its methods are grouped in the LibFoundationEDC category. For each symbol or each category, you can see its size in ROM (Image Size) and RAM (Runtime Size).

The SOAR groups all the Java strings in the same section, which appears in the ApplicationStrings category. The same applies to the static fields (Statics category), the types (Types category), and the class names (ClassNames category).

The SOAR Information File

The SOAR information file contains details on the embedded elements of an application.

Since Architecture 8.x , information are dispatched in separate files which are related to *SOAR build phases*:

- soar/<main class>.loadermap : generated by the SOAR Resolver. It provides details on files and resources that have been loaded from the *Application Classpath*.
- soar/<main class>.selectormap : generated by the SOAR Resolver. It provides details about the elements that have been included in the application.
- soar/<main class>.optimizermap : generated by the SOAR Optimizer. It provides details about the elements that have been linked in the application.

Each of these files can be opened with an XML editor. The following table describes the information that can be retrieved with their file location.

The SOAR Information File (Architecture 8.x)

The SOAR Information File (Architecture 7.x)

Information	XML Location (tag > subtag [attribute=value])	File Location
Classpath	classpath	soar/<main class>.loadermap
Resources	resources	soar/<main class>.loadermap
External resources	external_resources	soar/<main class>.loadermap
System properties	properties	soar/<main class>.loadermap
Constants	constants	soar/<main class>.loadermap
Immutables	N/A	N/A
Interned strings	strings	soar/<main class>.selectormap
Class initialization order	clinit	soar/<main class>.selectormap
Types	types	soar/<main class>.selectormap
Number of types	types>[nb]	soar/<main class>.selectormap
Number of concrete classes	types[nbConcreteClasses]	soar/<main class>.selectormap
Number of abstract classes	types[nbAbstractClasses]	soar/<main class>.selectormap
Number of interfaces	types[nbInterfaces]	soar/<main class>.selectormap
Number of arrays	types[nbArrays]	soar/<main class>.selectormap
Class instance size (in bytes)	types>type[instanceSize]	soar/<main class>.optimizermap
Type *embeds its name*	types>type[hasRuntimeName = true]	soar/<main class>.selectormap
Type *is exposed as Kernel API*	types>type[api=true]	soar/<main class>.selectormap
Number of reference fields in a class	types>type[nbReferenceFields]	soar/<main class>.optimizermap
Methods	methods	soar/<main class>.selectormap
Method code size (in bytes)	methods>method[codesize]	soar/<main class>.optimizermap
Method *is inlined*	methods>method[inlined=true]	soar/<main class>.optimizermap
Method *is exposed as Kernel API*	methods>method[api=true]	soar/<main class>.selectormap
Statics fields	statics	soar/<main class>.selectormap

Information	XML tag>subtag[attribute=value]	File
Classpath	classpath	soar/<main class>.xml
Resources	selected_resources	soar/<main class>.xml
External resources	external_resources	soar/<main class>.xml
System properties	java_properties	soar/<main class>.xml
Constants	constants	soar/<main class>.xml
Immutables	selected_immutables	soar/<main class>.xml
Interned strings	selected_internStrings	soar/<main class>.xml
Class initialization order	clinit_order	soar/<main class>.xml
Types	selected_types	soar/<main class>.xml
Number of types	selected_types[nb]	soar/<main class>.xml
Number of concrete classes	selected_types[nbConcreteClasses]	soar/<main class>.xml
Number of abstract classes	selected_types[nbAbstractClasses]	soar/<main class>.xml
Number of interfaces	selected_types[nbInterfaces]	soar/<main class>.xml
Number of arrays	selected_types[nbArrays]	soar/<main class>.xml
Class instance size (in bytes)	selected_types>type[instanceSize]	soar/<main class>.xml
Type *embeds its name*	required_types	soar/<main class>.xml
Type *is exposed as Kernel API*	selected_types>type[api=true]	soar/<main class>.xml
Number of reference fields in a class	selected_types>type[nbReferenceFields]	soar/<main class>.xml
Methods	selected_methods	soar/<main class>.xml
Method code size (in bytes)	selected_methods>method[codesize]	soar/<main class>.xml
Method *is inlined*	selected_methods>method[inlined=true]	soar/<main class>.xml
Method *is exposed as Kernel API*	selected_methods>method[api=true]	soar/<main class>.xml
Statics fields	selected_static_fields	soar/<main class>.xml

3.5.4 Virtual Device

The Virtual Device includes the same custom MicroEJ Core, libraries, and pre-installed Applications as the real device. The Virtual Device allows developers to run their applications either on the Simulator, or directly on the real device through local deployment.

The Simulator runs a mockup board support package (BSP Mock) that mimics the hardware functionality. An application on the Simulator is run as a Standalone Application.

Before an application is locally deployed on device, the SDK ensures that it does not depend on any API that is unavailable on the device.

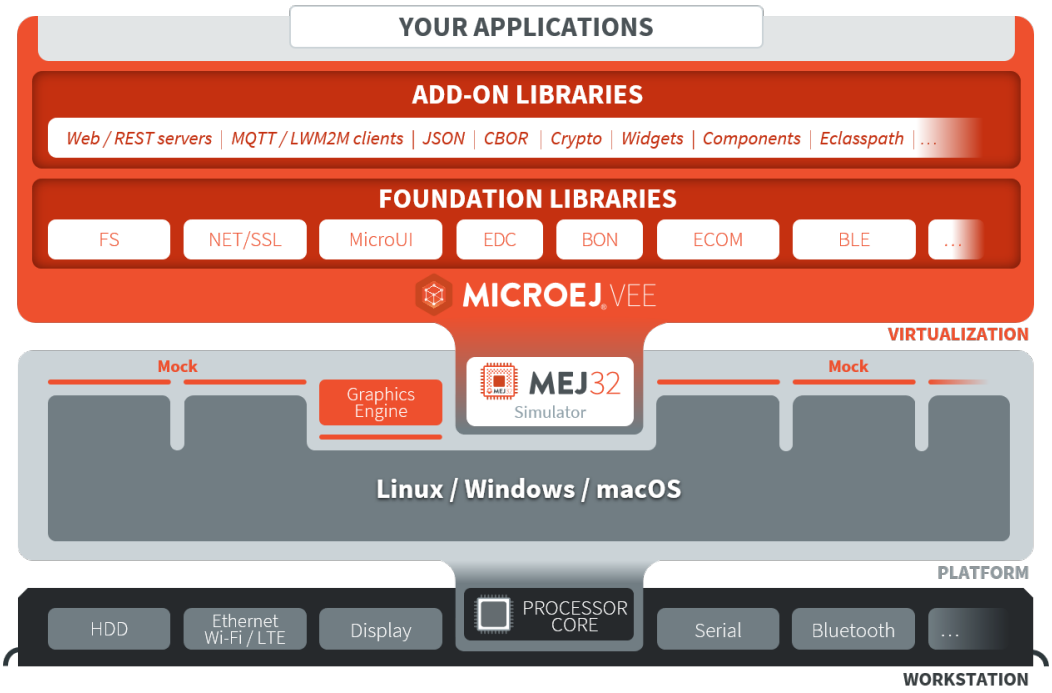


Fig. 121: MicroEJ Virtual Device Architecture

3.5.5 MicroEJ Classpath

MicroEJ Applications run on a target device and their footprint is optimized to fulfill embedded constraints. The final execution context is an embedded device that may not even have a file system. Files required by the application at runtime are not directly copied to the target device, they are compiled to produce the application binary code which will be executed by the Core Engine.

As a part of the compile-time trimming process, all types not required by the embedded application are eliminated from the final binary.

MicroEJ Classpath is a developer defined list of all places containing files to be embedded in the final application binary. MicroEJ Classpath is made up of an ordered list of paths. A path is either a folder or a zip file, called a JAR file (JAR stands for Java ARchive).

- *Application Classpath* explains how the MicroEJ Classpath is built from a MicroEJ Application project.
- *Classpath Load Model* explains how the application contents is loaded from MicroEJ Classpath.
- *Classpath Elements* specifies the different elements that can be declared in MicroEJ Classpath to describe the application contents.

Application Classpath

The following schema shows the classpath mapping from a MicroEJ Application project to the MicroEJ Classpath ordered list of folders and JAR files. The classpath resolution order (left to right) follows the project appearance order (top to bottom).

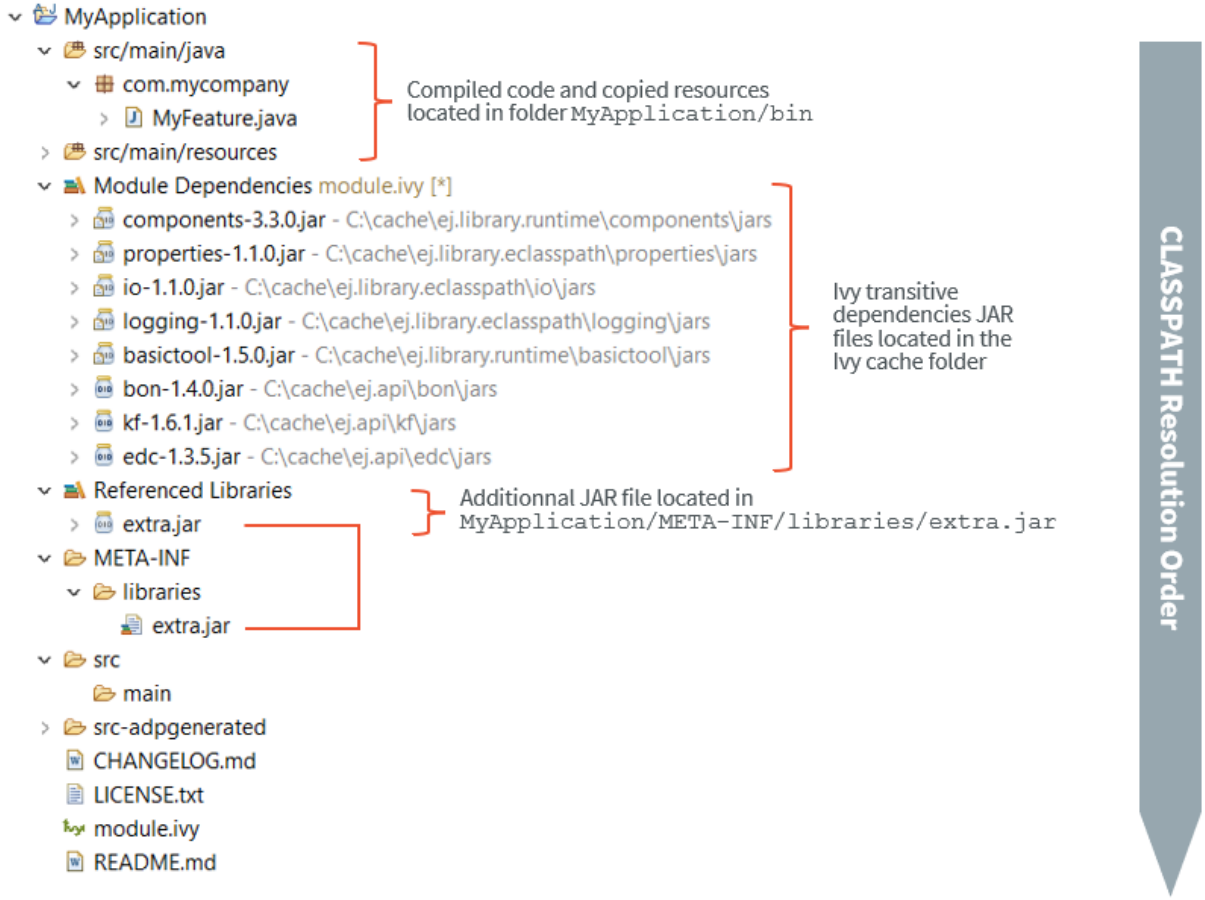


Fig. 122: MicroEJ Application Classpath Mapping

Note: For Sandboxed Applications, when a library cannot be *added as a dependency* (because it is not available in a repository for example), its JAR file can be directly added in the `META-INF/libraries` folder of the Application project. It is then automatically added in the compilation classpath and is available for the Application.

Classpath Load Model

A MicroEJ Application classpath is created via the loading of :

- an entry point type,
- all *.[extension].list files declared in a MicroEJ Classpath.

The different elements that constitute an application are described in *Classpath Elements*. They are searched within MicroEJ Classpath from left to right (the first file found is loaded). Types referenced by previously loaded MicroEJ Classpath elements are loaded transitively.

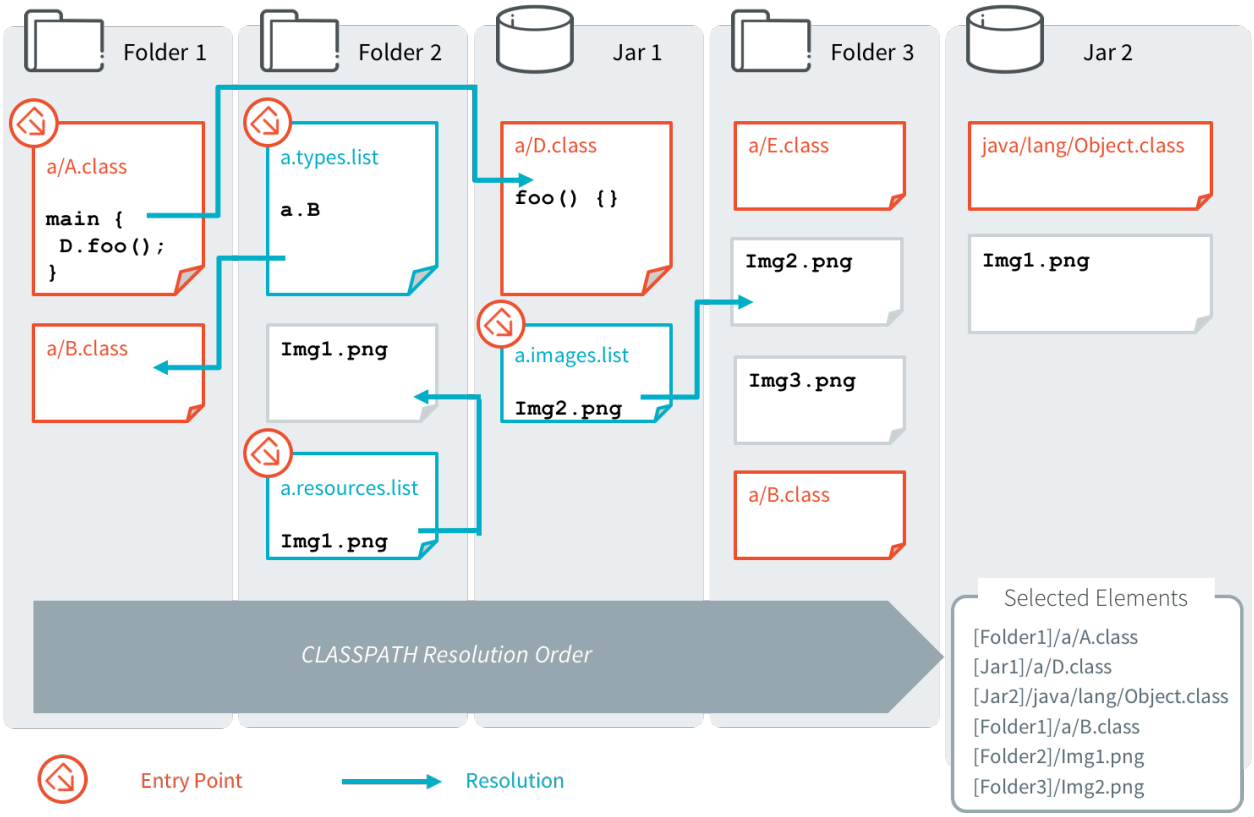


Fig. 123: Classpath Load Principle

Classpath Elements

The MicroEJ Classpath contains the following elements:

- An entrypoint described in section *Application Entry Points*;
- Types in .class files, described in section *Types*;
- Immutables Object data files, described in Section *Immutable Objects*;
- Raw Resources, Images, Fonts and Native Language Support (NLS) described in *Application Resources*;
- *.[extension].list files, declaring contents to load. Supported list file extensions and format is specific to declared application contents and is described in the appropriate section.

At source level, Java types are stored in src/main/java folder of the *module project*, any other kind of resources and list files are stored in the src/main/resources folder.

Application Entry Points

MicroEJ Application entry point declaration differs depending on the application kind:

- In case of a Standalone Application, it is a class that contains a `public static void main(String[])` method, declared using the option `application.main.class`.
- In case of a Sandboxed Application, it is a class that implements ej.kf.FeatureEntryPoint, declared using the `entryPoint` property in the `.kf` file in the `src/main/resources/` folder.

Types

MicroEJ types (classes, interfaces) are compiled from source code (`.java`) to classfiles (`.class`). When a type is loaded, all types dependencies found in the classfile are loaded (transitively).

A type can be declared as a *Required type* in order to enable the following usages:

- to be dynamically loaded from its name (with a call to Class.forName(String));
- to retrieve its fully qualified name (with a call to Class.getName()).
- when *Tiny-Sandbox* capability is enabled, to retrieve its package (with a call to Class.getPackage()).

A type that is not declared as a *Required type* may not have its fully qualified name (FQN) embedded. Its FQN can be retrieved using the stack trace reader tool (see *Stack Trace Reader*).

Required Types are declared in MicroEJ Classpath using `*.types.list` files. The file format is a standard Java properties file, each line listing the fully qualified name of a type. Example:

```
# The following types are marked as MicroEJ Required Types
com.mycompany.MyImplementation
com.mycompany.MyImplementation$MyInnerClass
java.util.Vector
```

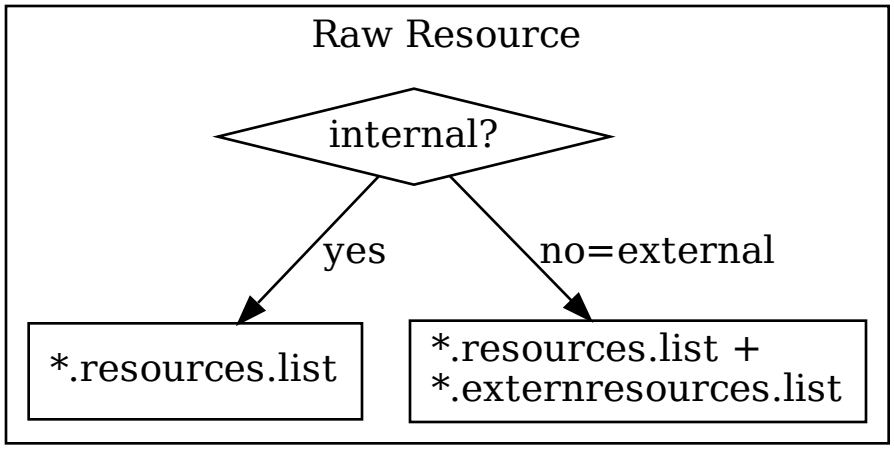
Note: Types that are defined from inner classes should be declared with the outer class fully qualified name followed by `$` and the name of the inner class.

Resources

Resources are binary files that need to be embedded by the application.

Note: For more details on all supported resources types, please refer to *Application Resources* chapter.

Raw resources are resources that can be dynamically retrieved with a call to java.lang.Class.getResourceAsStream(String). Raw Resources are declared in `*.resources.list` files (**and** in `*.externresources.list` for external resources, see *Application Resources*).



The file format is a standard Java properties file, each line is a relative / separated name of a file in MicroEJ Classpath to be embedded as a resource. Example:

```
# The following resource is embedded as a raw resource
com/mycompany/MyResource.txt
```

A resource is 4-bytes aligned in memory by default. Starting from *Architecture 8.0.0*, it is possible to modify the alignment constraint. Example:

```
# The following resource is linked to a 32-bytes aligned address in memory.
com/mycompany/MyResource.txt:32
```

Note: If a Resource is declared multiple times in the classpath, the alignment constraint with the highest value is used. If the alignment constraints are specific to the target, it is recommended to only declare them in the Application project instead of libraries.

If a resource filename contains a delimiter (such as a whitespace, : , or =), the delimiter must be escaped using the backslash \ character in the *.resources.list file. Example:

```
# The resource filename: "resource name contains whitespace.txt"
com/mycompany/resource\ name\ contains\ whitespace.txt
```

Immutable Objects

Immutables objects are regular read-only objects that can be retrieved with a call to ej.bon.Immutables.get(String). Immutables objects are declared in files called *immutable objects data files*, which format is described in the *[BON] specification*. Immutables objects data files are declared in MicroEJ Classpath using `*.immutables.list` files. The file format is a standard Java properties file, each line is a `/` separated name of a relative file in MicroEJ Classpath to be loaded as an Immutable objects data file. Example:

```
# The following file is loaded as an Immutable objects data files
com/mycompany/MyImmutables.data
```

System Properties

System Properties are key/value string pairs that can be accessed with a call to System.getProperty(String).

System Properties are defined when building a *Standalone Application*, by declaring `*.properties.list` files in MicroEJ Classpath.

The file format is a standard Java properties file. Example:

Listing 16: Example of Contents of a MicroEJ Properties File

```
# The following property is embedded as a System property
com.mycompany.key=com.mycompany.value
microedition.encoding=ISO-8859-1
```

System Properties are resolved at runtime, and all declared keys and values are embedded as intern Strings.

System Properties can also be defined using *Standalone Application Options*. This can be done by setting the option with a specific prefix in their name:

- Properties for both the Core Engine and the Simulator : name starts with `microej.java.property.*`
- Properties for the Simulator only: name starts with `sim.java.property.*`
- Properties for the Core Engine only: name starts with `emb.java.property.*`

For example, to define the property `myProp` with the value `theValue`, set the following option :

Listing 17: Example of MicroEJ System Property Definition as Application Option

```
microej.java.property.myProp=theValue
```

Note: A System Property defined as an Application Option takes precedence over a System Property defined in the classpath.

Note: When building a *Sandboxed Application*, `*.properties.list` files found in MicroEJ Classpath are silently skipped.

Constants

Constants are key/value string pairs that can be accessed with a call to ej.bon.Constants.get[Type](String), where Type if one of:

- Boolean,
- Byte,
- Char,
- Class,
- Double,
- Float,
- Int,
- Long,
- Short,
- String.

Constants are declared in MicroEJ Classpath *.constants.list files. The file format is a standard Java properties file. Example:

Listing 18: Example of Contents of a BON constants File

```
# The following property is embedded as a constant
myconstantkey=myconstantvalue
```

Starting from *Architecture 8.3.0*, it is also possible to define a constant using the following *Application Option*:

Listing 19: Example of declaration of a BON constant Application Option

```
microej.constant.myconstantkey=myconstantvalue
```

Note: A Constant defined as an Application Option takes precedence over a constant defined in the classpath.

Constants are resolved at binary level without having to recompile the sources.

At link time, constants are directly inlined at the place of Constants.get[Type] method calls with no cost.

The String key parameter must be resolved as an inlined String:

- either a String literal "myconstantkey"
- or a static final String field resolved as a String constant

The String value is converted to the desired type using conversion rules described by the *[BON]* API. A boolean constant declared in an if statement condition can be used to fully remove portions of code. This feature is similar to C pre-processors #ifdef directive with the difference that this optimization is performed at binary level without having to recompile the sources.

Listing 20: Example of `if` code removal using a BON boolean constant

```
if (Constants.getBoolean("myconstantkey")) {
      System.out.println("this code and the constant string will be fully removed when the constant is resolved to
↪'false'")
}
```

Please mind that `Constants.getXXX` must be inlined in the `if` condition to take effect. The following piece of code will not remove the code:

```
static final boolean MY_CONSTANT = Constants.getBoolean("myconstantkey");

...

if(MY_CONSTANT){
  System.out.println("this code will not be removed when MY_CONSTANT is resolved to 'false'")
}
```

Note: In *Multi-Sandbox* environment, constants are processed locally within each context. In particular, constants defined in the Kernel are not propagated to *Sandboxed Applications*.

3.5.6 Application Resources

An Application resource is the contents of a file identified its relative path from the Application *classpath*.

An Application resource is one of the following type:

- Raw Resource,
- Image,
- Font,
- Internationalized Message (Native Language Support).

The resource may be stored in RAM, flash, or external flash; and it is the responsibility of the Core Engine and/or the BSP to retrieve and load it.

There are two ways to store resources:

- Internal resource: The resource is taken into consideration during the Application build. The SOAR step loads the resource and copies it into the same C library as the Application. Like the Application, the resource is linked into the CPU address space range (internal device memories, external parallel memories, etc.).
- External resource: The resource is not taken into consideration during the Application build. It is the responsibility of the BSP project to manage external resources. The resource is often programmed outside the CPU address space range (storage media like SD card, serial NOR flash, EEPROM, etc.).

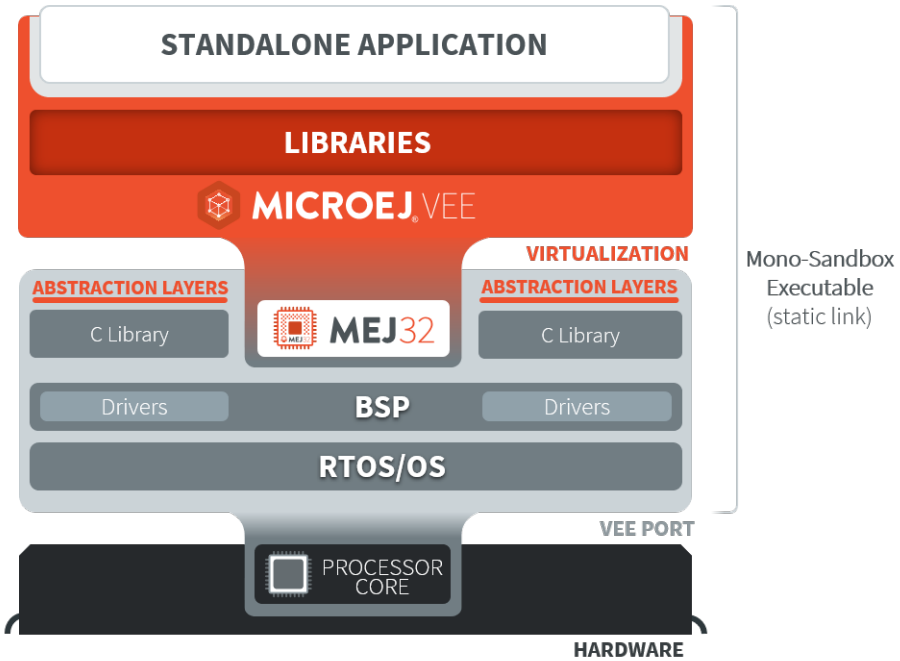
  The BSP must implement the proper Low Level API (LLAPI) C functions: `LLEXT_RES_impl.h`. See *External Resources Loader* for more information on the implementation.

All resources must be added in the project, usually in `src/main/resources/...` folder. All resources must be declared in the appropriate `*.list` files depending on the type (raw, UI image, VG image, UI font, VG font, NLS) and the storage location (internal or external). The following figure summarized how to declare resources:

Table 9: Resources

Resource type	Location	File(s) to update
Raw Resource	internal	*.resources.list
	external	*.resources.list **and** *.externresources.list
MicroUI Image	internal	*.images.list
	external	*.externimages.list **or** *.imagesext.list (deprecated)
MicroVG Image	internal	*.vectorimages.list
	external	*.externvectorimages.list
MicroUI Font	internal	*.fonts.list
	external	*.fontsext.list
MicroVG Font	internal	*.resources.list
	external	*.resources.list **and** *.externresources.list
Internationalized String (Native Language Support)	internal	*.nls.list
	external	*.nls.list **and** *.externresources.list

3.5.7 Standalone Application



Introduction

A Standalone Application is a Java Application directly linked to the C code to produce an Executable. Such an application must define a main entry point (i.e., a class containing a `public static void main(String[])` method).

The following figure shows the general process of building a Standalone Application to an Executable.

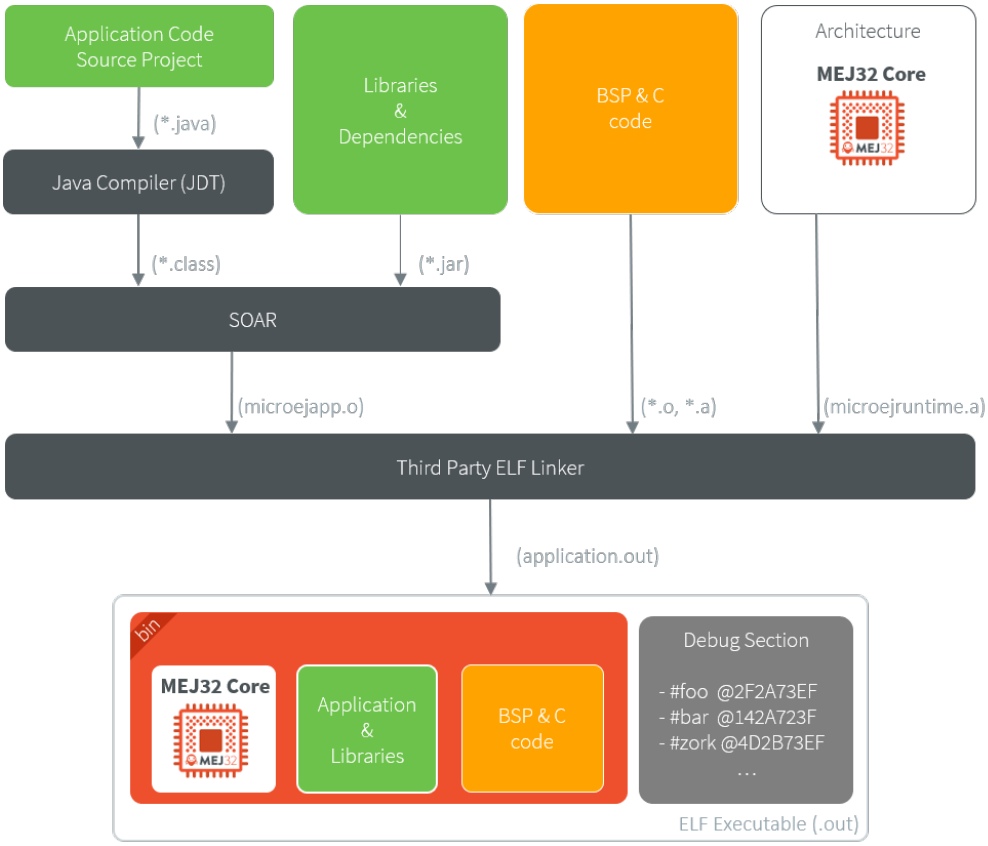


Fig. 124: Standalone Application Link Flow

Standalone Application Options

To run a Standalone Application on a VEE Port, a set of options must be defined. Options can be of different types:

- Memory Allocation options (e.g., set the Managed Heap size). These options are usually called link-time options.
- Simulator & Debug options (e.g., enable periodic Managed Heap dump).
- Deployment options (e.g., copy `microejapp.o` to a suitable BSP location).
- Foundation Library specific options (e.g., embed UTF-8 encoding).

The following section describes options provided by the latest Architecture version. Please consult the appropriate Pack(s) documentation for options related to other Foundation Libraries (MicroUI, NET, SSL, FS, ...) integrated to your VEE Port.

Notice that some options may not be supported by your VEE Port, in the following cases:

- Option is specific to the Core Engine capability (*tiny*/*mono*/*multi*).

- Option is specific to the target (Core Engine or Simulator).
- Option has been introduced in a newer Architecture version.

Defining an Option with SDK 6

Using a Properties File

With the SDK 6, the Applications options can be defined in a properties file located in the `configuration` folder of the project. Usually, the options are defined in a file named `common.properties`, but all properties files located in this folder are loaded, no matter what their name is.

To set an option in a properties file, open the file in a text editor and add a line to set the desired option to the desired value, for example:

```
soar.generate.classnames=true
```

Using System Properties

Application options can be defined by System properties using the `microej.option.*` prefix. System properties can either be defined in command line using `-D` argument, or in `gradle.properties` file using `systemProp.*` prefix. For instance, in command line:

```
-Dmicroej.option.soar.generate.classnames=true
```

or in `gradle.properties` file:

```
systemProp.microej.option.soar.generate.classnames=true
```

See the *How To Pass a Property to Project Build Script* for more information about System properties.

Note: When Application options are also defined in properties from the `configuration` folder of the project, they will be overridden by System properties.

Defining an Option with SDK 5 or lower

With the SDK 5 or lower, a Standalone Application option can be defined either from a launcher or from a properties file. It is also possible to use both together. Each MicroEJ Architecture and MicroEJ Pack option comes with a default value, which is used if the option has not been set by the user.

Using a Launcher

To set an option in a launcher, perform the following steps:

1. In the SDK, select `Run` > `Run Configurations…` menu,
2. Select the launcher of the application under `MicroEJ Application` or create a new one,
3. Select the `Configuration` tab,

4. Find the desired option and set it to the desired value. If the option does not appear in the page, there are two cases: - the option has been introduced in a newer Architecture version, - the option is an advanced option. It is set using a system property in the JRE Tab . See the *JRE Tab* section for more details.

It is recommended to index the launcher configuration to your version control system. To export launcher options to the filesystem, perform the following steps:

1. Select the Common tab,
2. Select the Shared file: option and browse the desired export folder,
3. Press the Apply button. A file named [launcher_configuration_name].launch is generated in the export folder.

Using a Properties File

Options can be also be defined in properties files.

When a Standalone Application is built using the firmware-singleapp *skeleton*, options are loaded from properties files located in the build folder at the root of the project.

The properties files are loaded in the following order:

1. Every file matching build/sim/*.properties , for Simulator options only (Virtual Device build). These files are optional.
2. Every file matching build/emb/*.properties , for Device options only (Firmware build). These files are optional.
3. Every file matching build/*.properties , both for Simulator and Device options. At least one file is required.

Usually, the build folder contains a single file named common.properties .

In case an option is defined in multiple properties files, the option of the first loaded file is taken into account and the same option defined in the other files is ignored (a loaded option cannot be overridden).

The figure below shows the expected tree of the build folder:

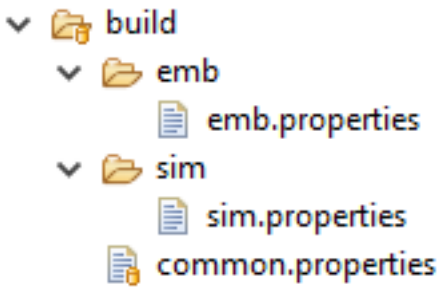


Fig. 125: Build Options Folder

It is recommended to index the properties files to your version control system.

To set an option in a properties file, open the file in a text editor and add a line to set the desired option to the desired value. For example: soar.generate.classnames=true .

To use the options declared in properties files in a launcher, perform the following steps:

1. In the SDK, select Run > Run Configurations… ,

2. Select the launcher of the application,
3. Select the Execution tab,
4. Under Option Files, press the Add… button,
5. Browse the sim.properties file for Simulator or the emb.properties file for Device (if any) and press Open button,
6. Add the common.properties file and press the Open button.

Note: An option set in a properties file can not be modified in the Configuration tab. Options are loaded in the order the properties files are added (you can use Up and Down buttons to change the file order). In Configuration tab, hovering the pointer over an option field will show the location of the properties file that defines the option.

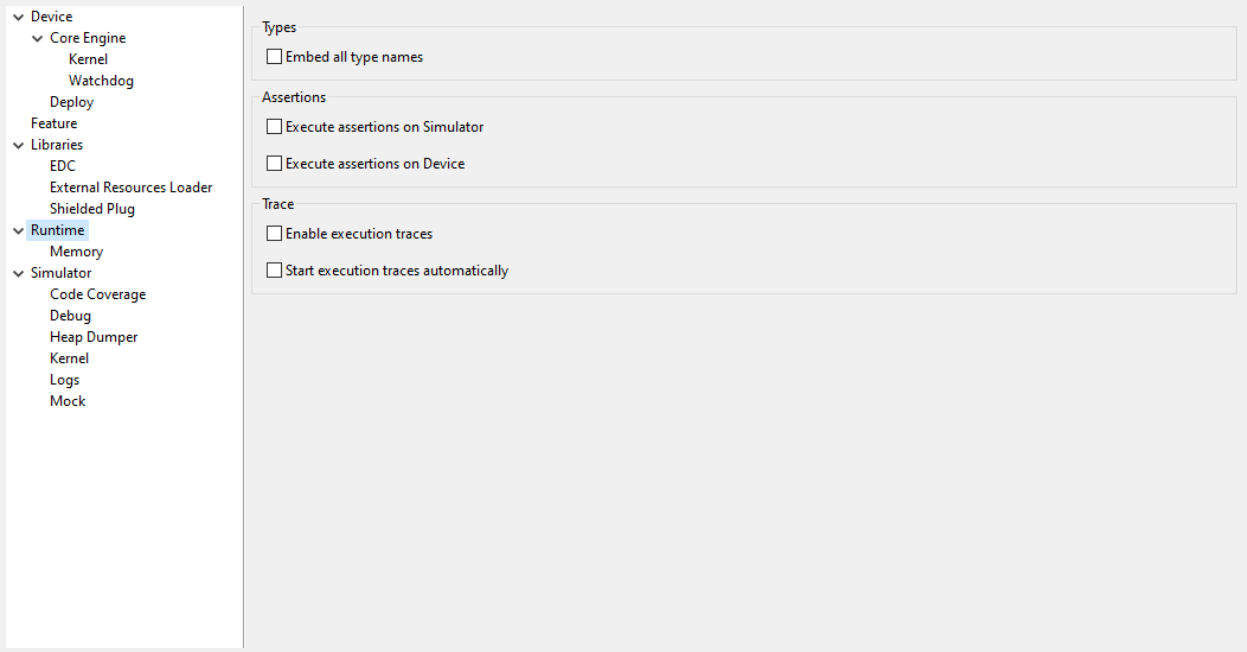
Generating a Properties File

In order to export options defined in a .launch file to a properties file, perform the following steps:

1. Select the [launcher_configuration_name].launch file,
2. Select File > Export > MicroEJ > Launcher as Properties File,
3. Browse the desired output .properties file,
4. Press the Finish button.

> **Warning:** The Simulator uses some system properties to configure internal memory limits. See *Group: Advanced Simulation Options* for more information.

Category: Runtime



Group: Types

Option(checkbox): Embed all type names

Option Name: soar.generate.classnames

Default value: false

Description:

Embed the name of all types. When this option is disabled, only names of declared *required types* are embedded. This option affects code size.

Option(checkbox): Group Methods

Architecture 8

Architecture 7

Option Name: com.microej.soar.groupMethods.enabled

Default value: false

Description:

Group all embedded methods in single ELF section in microejapp.o. This avoids to reach the following SOAR error while building a large application and the maximum number of ELF sections (65536) is reached.

```
1 : ELF BINARY ERROR :
  [M2] - Invalid value for 'Section index range [0,0xFF00[' field. Overflow detected when writing '101635'.
```

Option Name: com.microej.soar.groupMethodsByType.enabled

Default value: false

Description:

Group all embedded methods of a Java type in single ELF section in microejapp.o . This avoids to reach the following SOAR error while building a large application and the maximum number of ELF sections (65536) is reached.

```
1 : ELF ERROR
  [M1] - Invalid value for U2 field. Overflow detected when writing 102092.
```

Warning: This option affects the application code size (especially inline methods are embedded even if they are not used).

Group: Assertions

Option(checkbox): Execute assertions on Simulator

Option Name: core.assertions.sim.enabled

Default value: false

Description:

When this option is enabled, assert statements are executed. Please note that the executed code may produce side effects or throw java.lang.AssertionError .

Option(checkbox): Execute assertions on Device

Option Name: core.assertions.emb.enabled

Default value: false

Description:

When this option is enabled, assert statements are executed. Please note that the executed code may produce side effects or throw java.lang.AssertionError .

Group: Trace

Option(checkbox): Enable execution traces

Option Name: core.trace.enabled

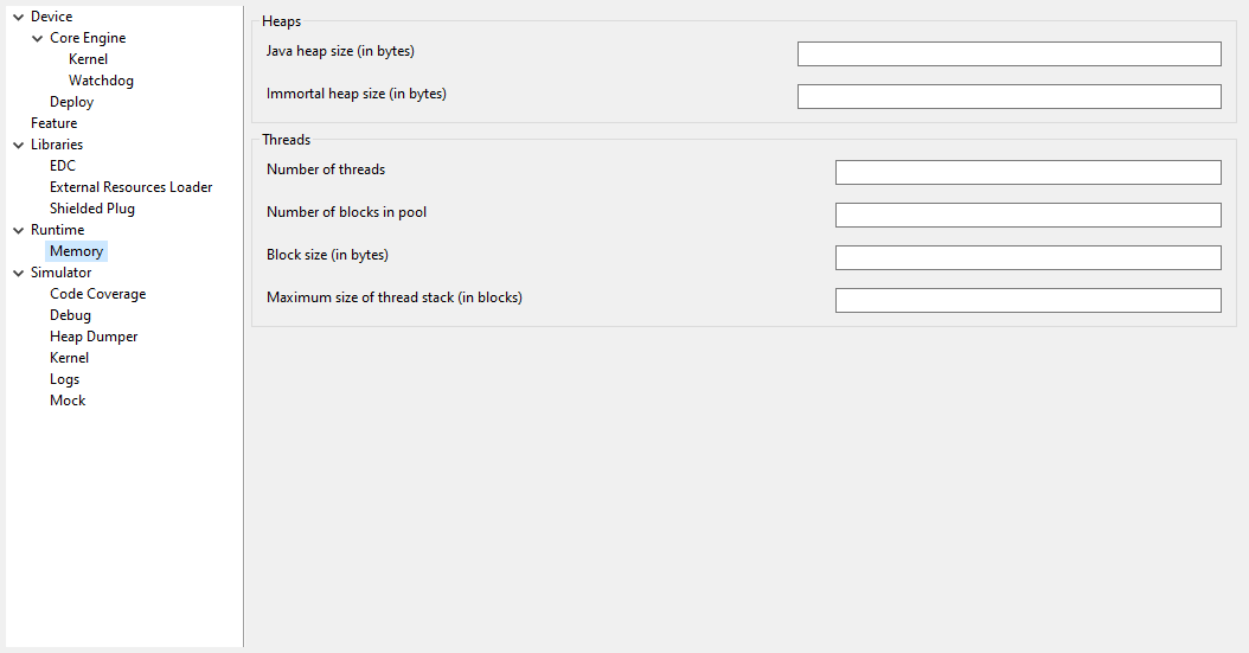
Default value: false

Option(checkbox): Start execution traces automatically

Option Name: core.trace.autostart

Default value: false

Category: Memory



Group: Heaps

Option(text): Managed Heap size (in bytes)

Option Name: core.memory.javaheap.size

Default value: 65536

Maximum value: 67108860 (see *Limitations*)

Description:

Specifies the Managed Heap size in bytes.

A Managed Heap contains live objects. An OutOfMemory error can occur if the heap is too small.

Note: The Managed heap can also be allocated (and sized) dynamically (see *Configuring Memory Dynamically*), in which case this option is ignored.

Option(text): Immortals Heap size (in bytes)

Option Name: core.memory.immortal.size

Default value: 4096

Description:

Specifies the Immortals Heap size in bytes.

The Immortals Heap contains the allocated Immortal objects. An OutOfMemory error can occur if the heap is too small.

Note: The Immortals heap can also be allocated (and sized) dynamically (see *Configuring Memory Dynamically*), in which case this option is ignored.

Group: Threads

Description:

This group allows the configuration of application and library thread(s). See *Threads and Stacks Allocation* section for more details.

Option(text): Number of threads

Option Name: core.memory.threads.size

Default value: 5

Description:

Specifies the number of threads the application will be able to use at the same time.

Option(text): Number of blocks in pool

Option Name: core.memory.threads.pool.size

Default value: 15

Description:

Specifies the number of blocks in the stacks pool.

Option(text): Block size (in bytes)

Option Name: core.memory.thread.block.size

Default value: 512

Description:

Specifies the thread stack block size (in bytes).

Option(text): Maximum size of thread stack (in blocks)

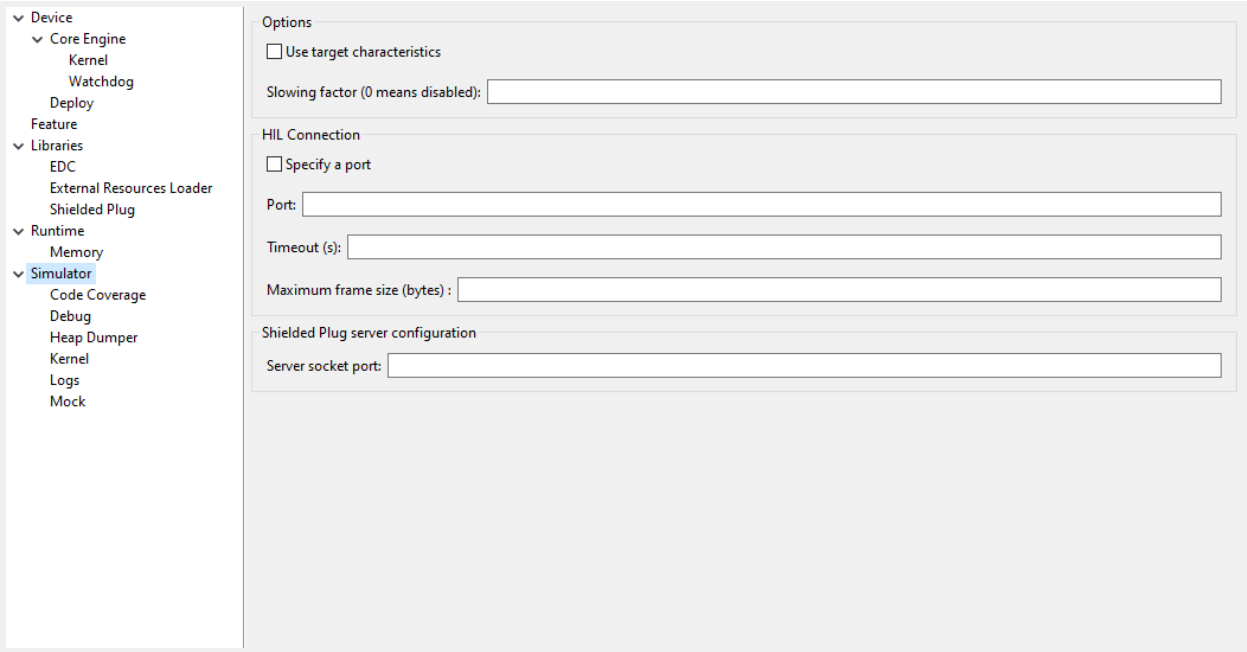
Option Name: core.memory.thread.max.size

Default value: 4

Description:

Specifies the maximum number of blocks a thread can use. If a thread requires more blocks a StackOverflow error will occur.

Category: Simulator



Group: Options

Description:

This group specifies options for MicroEJ Simulator.

Option(checkbox): Use target characteristics

> **Warning:** This option is deprecated and only available in SDK 5. It is not supported in SDK 6.

Option Name: s3.board.compliant

Default value: false

Description:

When selected, this option forces the MicroEJ Simulator to use the MicroEJ Platform exact characteristics. It sets the MicroEJ Simulator scheduling policy according to the MicroEJ Platform one. It forces resources to be explicitly specified. It enables log trace and gives information about the RAM memory size the MicroEJ Platform uses.

Option(text): Slowing factor (0 means disabled)

Option Name: s3.slow

Default value: 0

Description:

Format: Positive integer

This option allows the MicroEJ Simulator to be slowed down in order to match the MicroEJ Platform execution speed. The greater the slowing factor, the slower the MicroEJ Simulator runs.

Group: HIL Connection

Description:

This group enables the control of HIL (Hardware In the Loop) connection parameters (connection between MicroEJ Simulator and the Mocks).

Option(checkbox): Specify a port

Option Name: s3.hil.use.port

Default value: false

Description:

When selected allows the use of a specific HIL connection port, otherwise a random free port is used.

Option(text): Port

Option Name: s3.hil.port

Default value: 8001

Description:

Format: Positive integer

Values: [1024-65535]

It specifies the port used by the MicroEJ Simulator to accept HIL connections.

Option(text): Timeout (s)

Option Name: s3.hil.timeout

Default value: 10

Description:

Format: Positive integer

It specifies the time the MicroEJ Simulator should wait before failing when it invokes native methods.

Option(text): Maximum frame size (bytes)

Option Name: com.microej.simulator.hil.frame.size

Default value: 262144

Description:

Maximum frame size in bytes. Must be increased to transfer large arrays to native side.

Group: Shielded Plug server configuration

Description:

This group allows configuration of the Shielded Plug database.

Option(text): Server socket port

Option Name: sp.server.port

Default value: 10082

Description:

Set the Shielded Plug server socket port.

Group: Advanced Simulation Options

When running large applications, the Simulator can abruptly reach a memory limit with the following trace:

```
[...] An error message [...]
"Internal limits reached. Please contact support@microej.com"
See error log file: /tmp/microej/s3/s3_1616489929186.log
```

Depending on the error message, one of the following options must be set to increase the size of the memory area which is full.

Option: Objects Heap Size

Error Message: java.lang.OutOfMemoryError exception thrown

Option Name: S3.JavaMemory.HeapSize

Default value: 4096 (kilobytes)

Description:

This memory area contains any kind of objects (regular, immortal and immutable objects). If you get a java.lang.OutOfMemoryError exception but your Managed Heap is not full, most likely you should augment this option. It must be greater than the sum of *Managed Heap* and *Immortals Heap*.

Option: System Chars Size

Error Message: Failed to allocate internString.

Option Name: S3.JavaMemory.SystemCharsSize

Default value: 1024 (kilobytes)

Description:

This memory area contains system interned strings. System interned strings are likely allocated by the debugger. If you get a Failed to allocate internString. message while debugging an Application, most likely you should augment this option.

Option: Application Chars Size

Error Message: Failed to allocate internString.

Option Name: S3.JavaMemory.ApplicationCharsSize

Default value: 4096 (kilobytes)

Description:

This memory area contains Application interned strings (String literals). If you get a Failed to allocate internString. message while the Simulator is starting the Application, most likely you should augment this option.

Option: Methods Size

Error Message: Failed to allocate method's code.

Option Name: S3.JavaMemory.MethodsSize

Default value: 10000 (kilobytes)

Description:

This memory area contains loaded methods code.

Option: Thread Stack Size

Error Message: The simulator has encountered a stack overflow error while analyzing method [...]

Option Name: S3.JavaMemory.ThreadStackSize

Default value: 300 (kilobytes)

Description:

This memory area contains all Application threads stacks.

Option: Icetea Heap End

Error Message: S3 internal heap is full.

Option Name: IceteaRuntimeSupport.S3.HeapEnd

Default value: 64000000 (bytes)

Description:

This is the overall Simulator memory limit. It includes fixed sizes internal structures and all memory areas. The value must be greater than the size of the memory areas that can be parameterized above.

Option: Symbol Table Size

Error Message: Symbols table area is full.

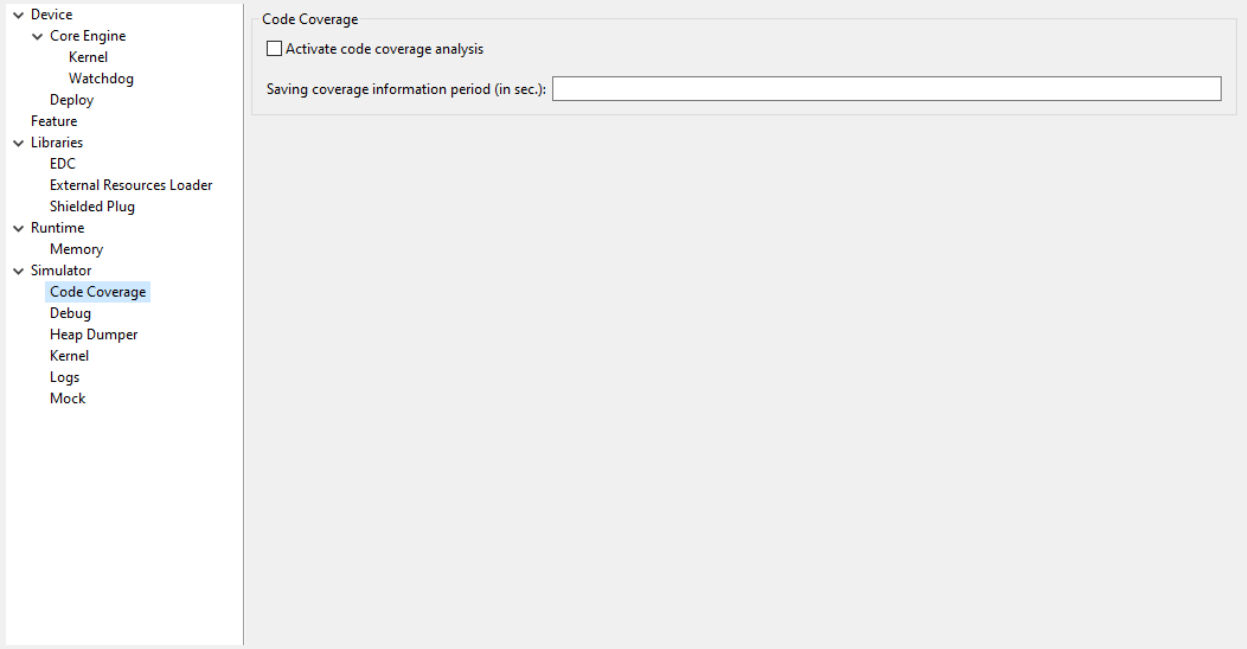
Option Name: S3.SymbolTable.MaxNbState

Default value: 500000

Description:

This is the number of symbols that can be handled by the internal symbol table (any kind of names: class names, method names, …).

Category: Code Coverage



Group: Code Coverage

Description:

This group is used to set parameters of the code coverage analysis tool.

Option(checkbox): Activate code coverage analysis

Option Name: s3.cc.activated

Default value: false

Description:

When selected it enables the code coverage analysis by the MicroEJ Simulator. Resulting files are output in the cc directory inside the output directory. You can process these files to an HTML report afterward with the built-in *Code Coverage Analyzer* .

Option(text): Saving coverage information period (in sec.)

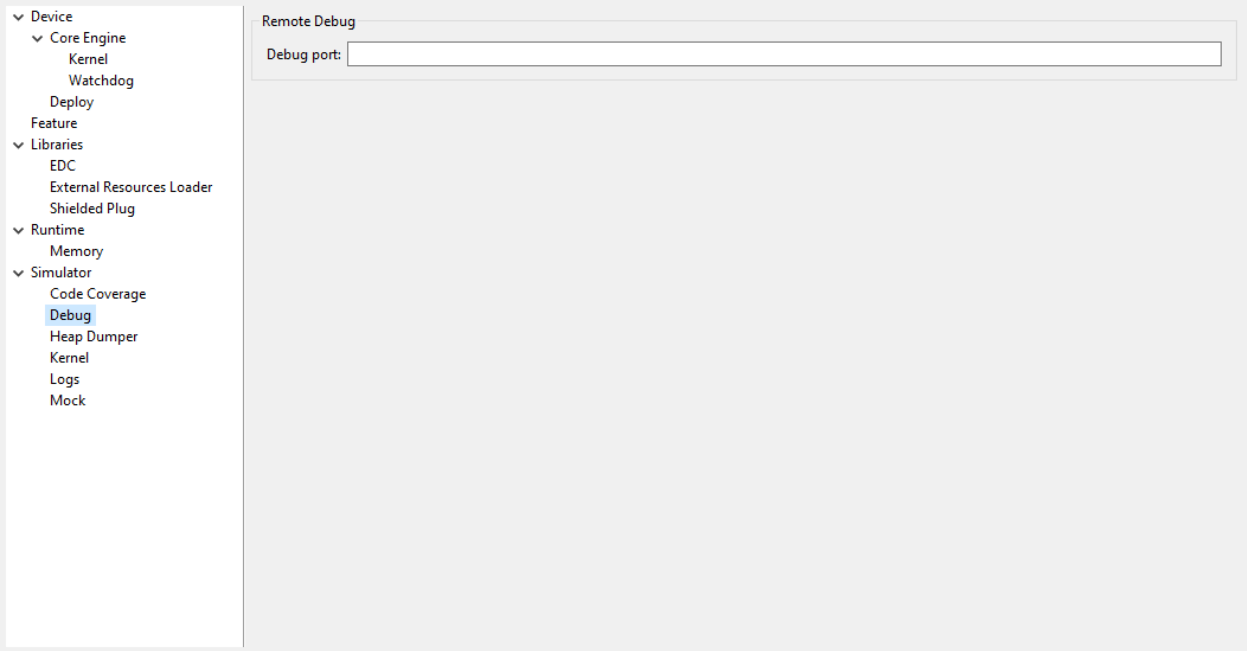
Option Name: s3.cc.thread.period

Default value: 15

Description:

It specifies the period between the generation of .cc files.

Category: Debug



Group: Remote Debug

Option(text): Debug port

Option Name: debug.port

Default value: 12000

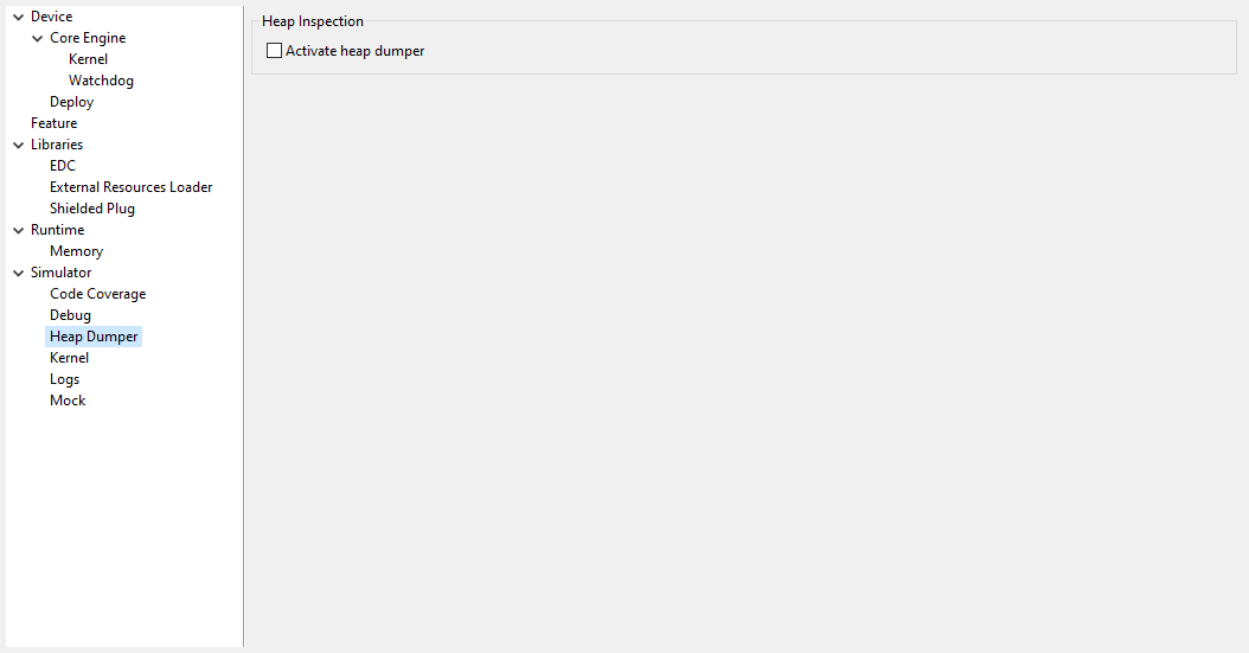
Description:

Configures the JDWP debug port.

Format: Positive integer

Values: [1024-65535]

Category: Heap Dumper



Group: Heap Inspection

Description:

This group is used to specify heap inspection properties.

Option(checkbox): Activate heap dumper

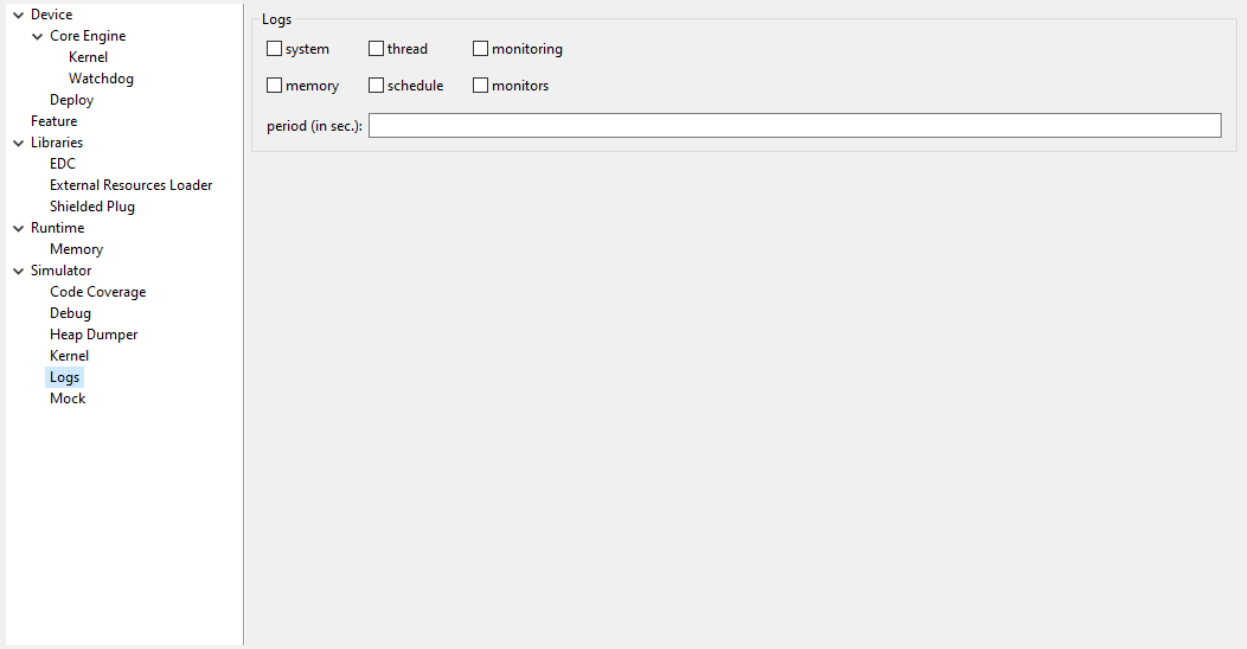
Option Name: s3.inspect.heap

Default value: false

Description:

When selected, this option enables a dump of the heap each time the System.gc() method is called by the MicroEJ Application.

Category: Logs



Group: Logs

Description:

This group defines parameters for MicroEJ Simulator log activity. Note that logs can only be generated if the Simulator > Use target characteristics option is selected.

Some logs are sent when the Simulator executes some specific action (such as start thread, start GC, etc), other logs are sent periodically (according to defined log level and the log periodicity).

Option(checkbox): system

Option Name: console.logs.level.low

Default value: false

Description:

When selected, System logs are sent when the Simulator executes the following actions:

start and terminate a thread

start and terminate a GC

exit

Option(checkbox): thread

Option Name: console.logs.level.thread

Default value: false

Description:

When selected, thread information is sent periodically. It gives information about alive threads (status, memory allocation, stack size).

Option(checkbox): monitoring

Option Name: console.logs.level.monitoring

Default value: false

Description:

When selected, thread monitoring logs are sent periodically. It gives information about time execution of threads.

Option(checkbox): memory

Option Name: console.logs.level.memory

Default value: false

Description:

When selected, memory allocation logs are sent periodically. This level allows to supervise memory allocation.

Option(checkbox): schedule

Option Name: console.logs.level.schedule

Default value: false

Description:

When selected, a log is sent when the Simulator schedules a thread.

Option(checkbox): monitors

Option Name: console.logs.level.monitors

Default value: false

Description:

When selected, monitors information is sent periodically. This level permits tracing of all thread state by tracing monitor operations.

Option(text): period (in sec.)

Option Name: console.logs.period

Default value: 2

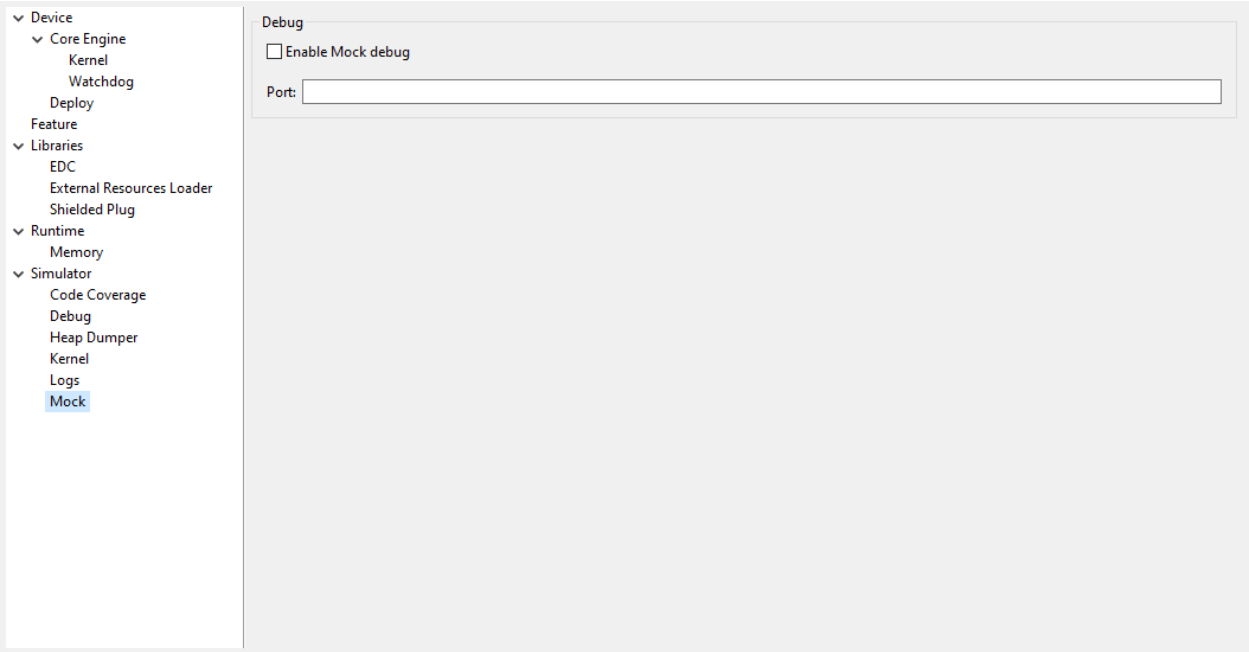
Description:

Format: Positive integer

Values: [0-60]

Defines the periodicity of periodical logs.

Category: Mock



Description:

Specify Hardware In the Loop Mock client options

Group: Debug

Option(checkbox): Enable Mock debug

Option Name: com.microej.simulator.hil.debug.enabled

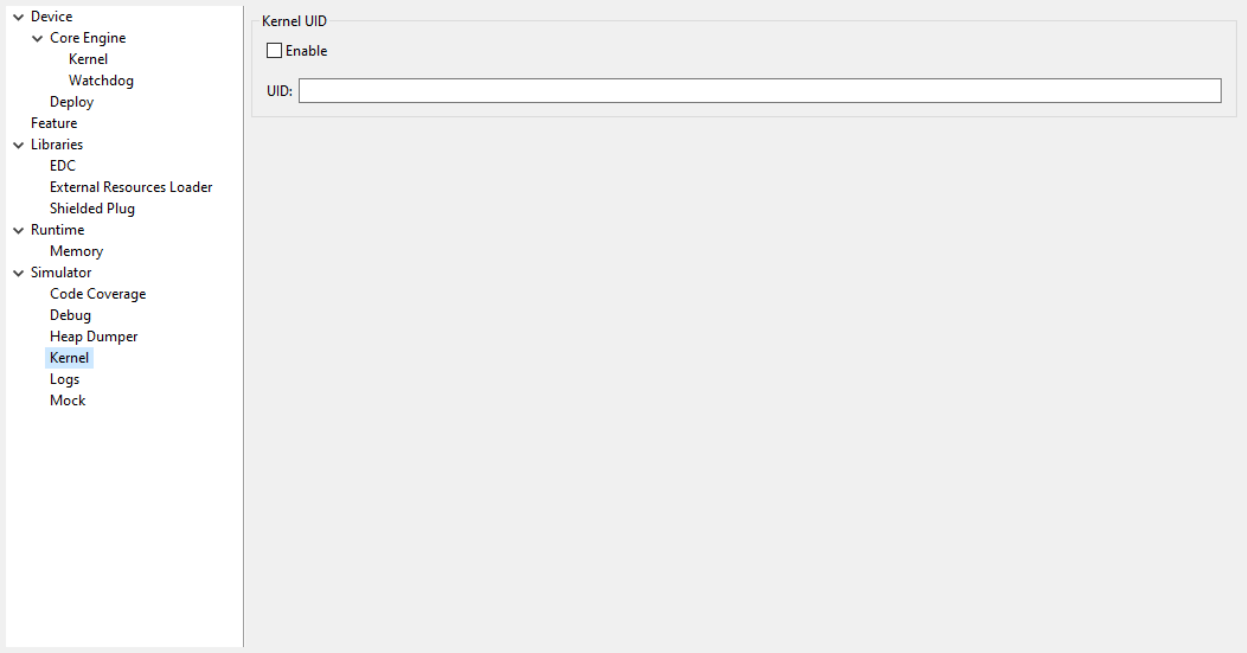
Default value: false

Option(text): Port

Option Name: com.microej.simulator.hil.debug.port

Default value: 8002

Category: Kernel



Group: Kernel UID

Option(checkbox): Enable

Option Name: com.microej.simulator.kf.kernel.uid.enabled

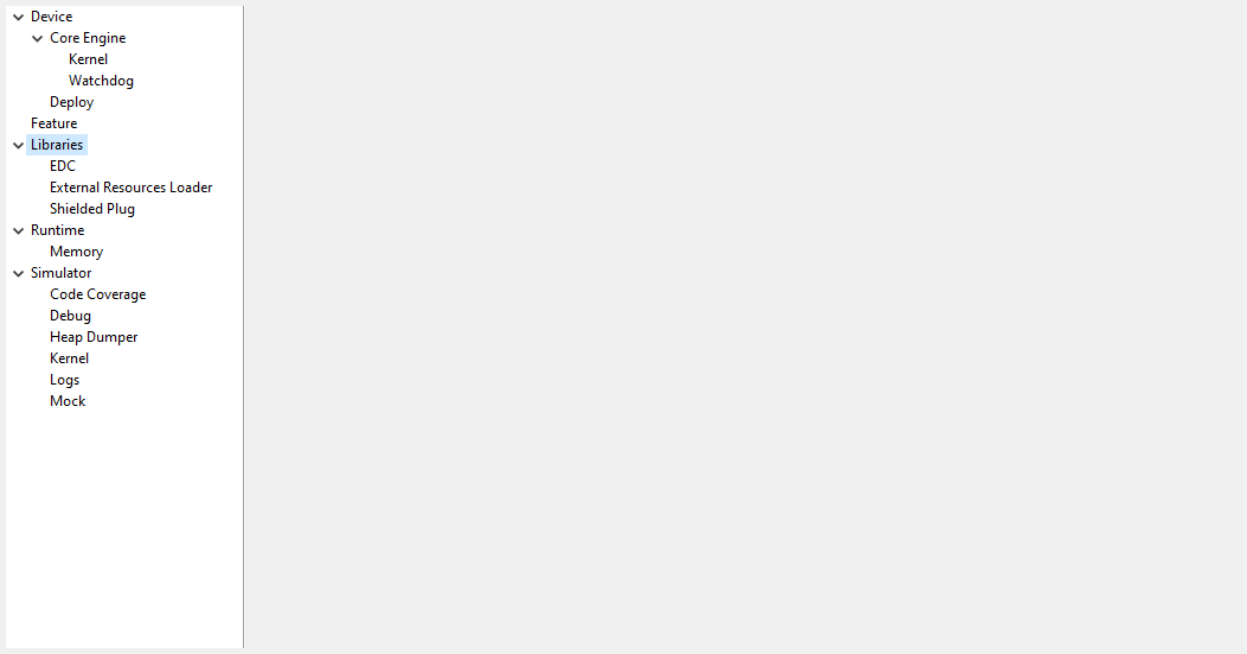
Default value: false

Option(text): UID

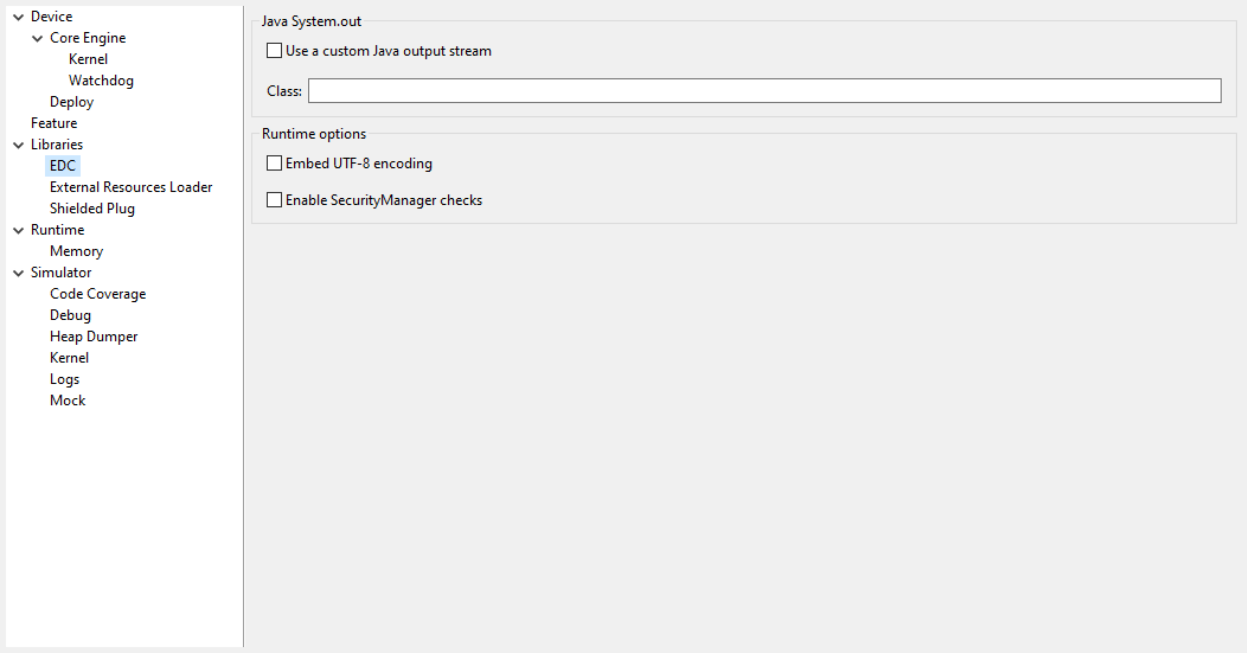
Option Name: com.microej.simulator.kf.kernel.uid

Default value: (empty)

Category: Libraries



Category: EDC



Group: Java System.out

Option(checkbox): Use a custom Java output stream

Option Name: core.outputstream.disable.uart

Default value: false

Description:

Select this option to specify another Java System.out print stream.

If selected, the default Java output stream is not used by the Java application. The Core Engine will not use the default Java output stream at startup.

Option(text): Class

Option Name: core.outputstream.class

Default value: (empty)

Description:

Format: Java class like packageA.packageB.className

Defines the Java class used to manage System.out .

At startup the Core Engine will try to load this class using the Class.forName() method. If the given class is not available, it will use the default Java output stream as usual. The specified class must be available in the application classpath.

Group: Runtime options

Description:

Specifies the additional classes to embed at runtime.

Option(checkbox): Embed UTF-8 encoding

Option Name: cldc.encoding.utf8.included

Default value: true

Description:

Embed UTF-8 encoding.

Option(checkbox): Enable SecurityManager checks

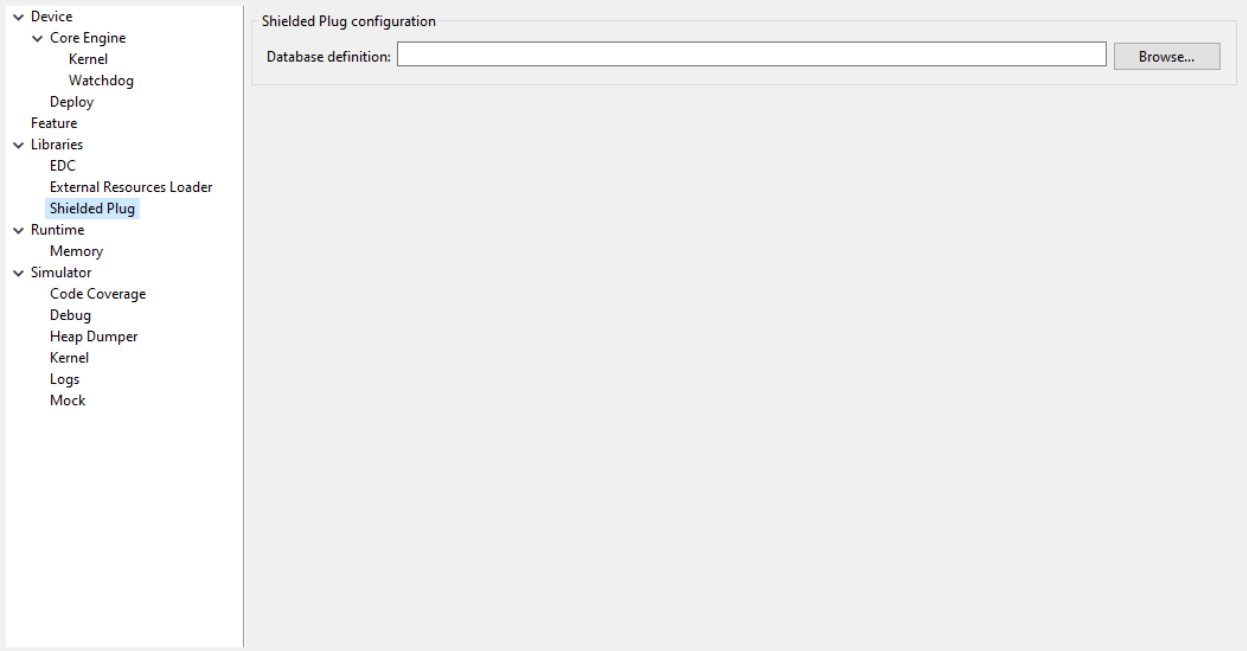
Option Name: com.microej.library.edc.securitymanager.enabled

Default value: false

Description:

Enable the security manager Permission checks.

Category: Shielded Plug



Group: Shielded Plug configuration

Description:

Choose the database XML definition.

Option(browse): Database definition

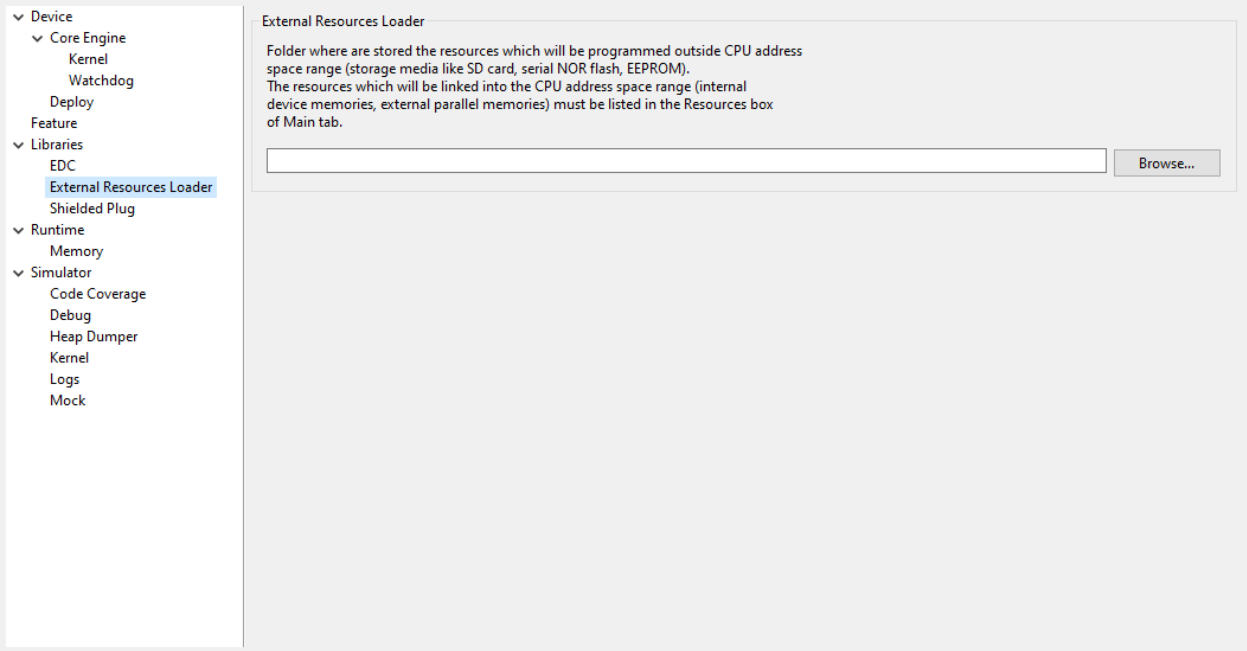
Option Name: sp.database.definition

Default value: (empty)

Description:

Choose the database XML definition.

Category: External Resources Loader



Group: External Resources Loader

Description:

This group allows to specify the external resources input folder. The content of this folder will be copied in an application output folder and used by SOAR and the Simulator. If empty, the default location will be [output folder]/externalResources, where [output folder] is the location defined in Execution tab.

Option(browse):

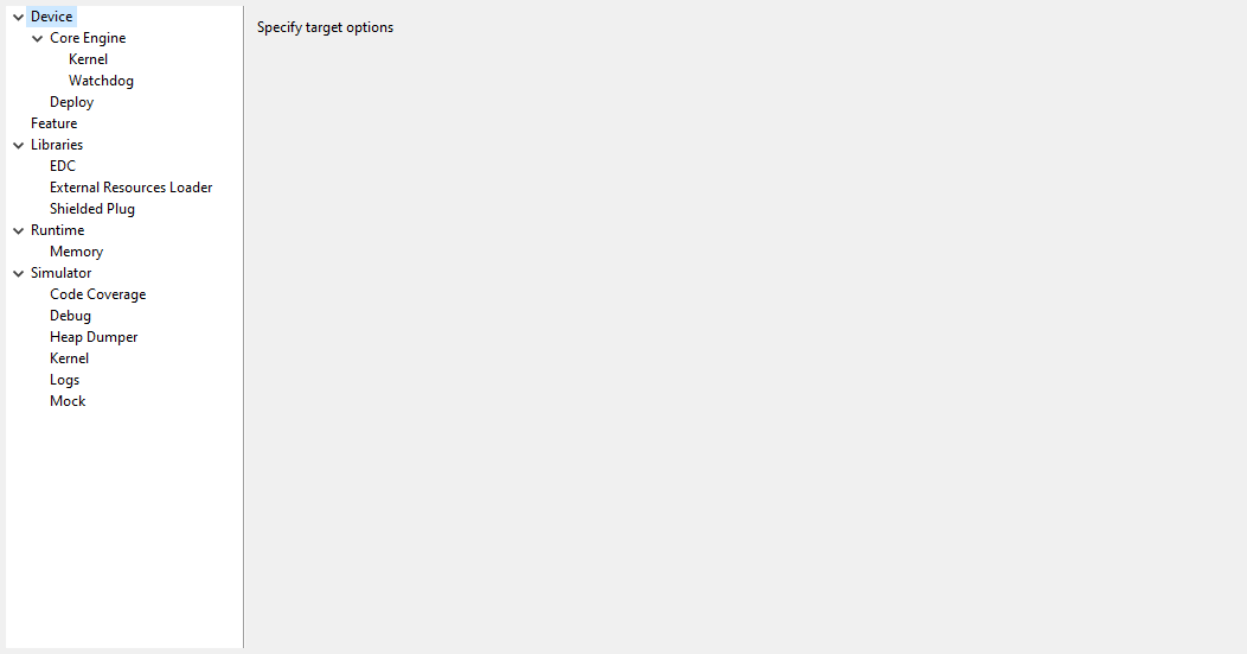
Option Name: ej.externalResources.input.dir

Default value: (empty)

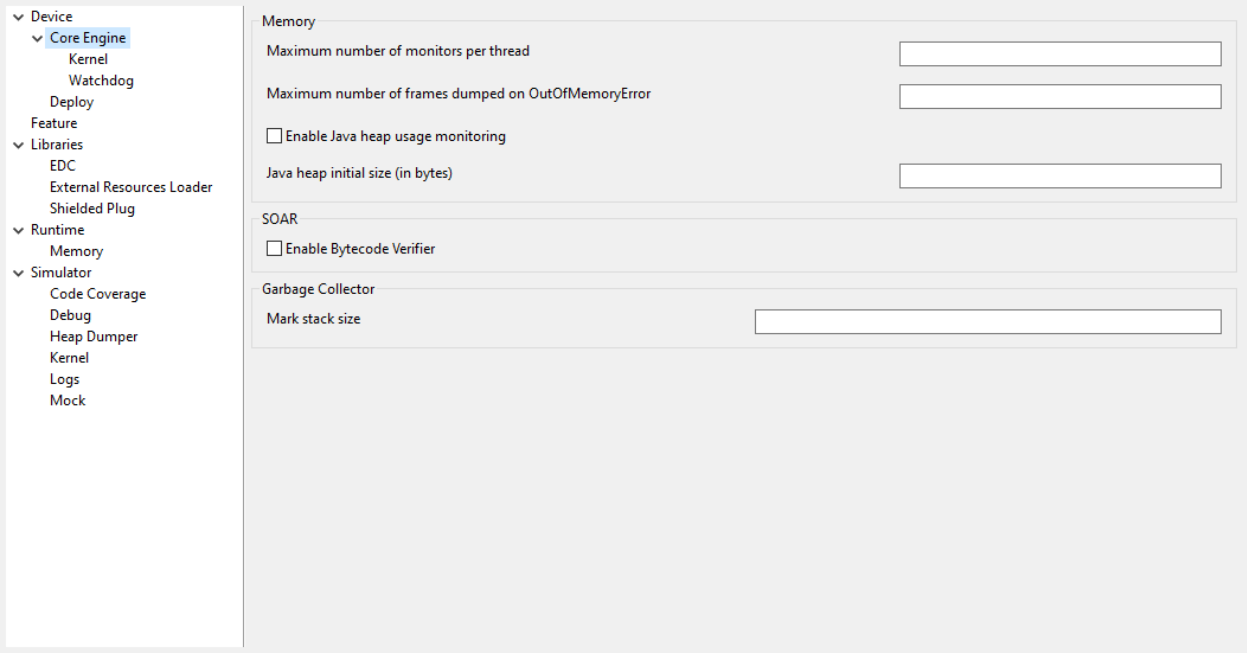
Description:

Browse to specify the external resources folder..

Category: Device



Category: Core Engine



Group: Memory

Option(text): Maximum number of monitors per thread

Option Name: core.memory.thread.max.nb.monitors

Default value: 8

Description:

Specifies the maximum number of monitors a thread can own at the same time.

Option(text): Maximum number of frames dumpers on OutOfMemoryError

Option Name: core.memory.oome.nb.frames

Default value: 1

Description:

Specifies the maximum number of call frames that can be dumped to the standard output when Core Engine throws an OutOfMemoryError.

Option(checkbox): Enable Managed Heap usage monitoring

Option Name: com.microej.runtime.debug.heap.monitoring.enabled

Default value: false

Option(text): Managed Heap initial size

Option Name: com.microej.runtime.debug.heap.monitoring.init.size

Default value: 0

Description:

Specify the initial size (in bytes) of the Managed Heap.

Group: SOAR

Option(checkbox): Enable Bytecode Verifier

Option Name: soar.bytecode.verifier

Default value: Standalone Application: false , Sandboxed Application: true

Description:

Enables *Binary Code Verifier* during application build.

In the context of building a Standalone Application, the bytecode verifier is, by default, disabled to prioritize performance. In this case, the code is considered trusted. Conversely, when building a Sandboxed Application, the bytecode verifier is automatically enabled by default. This is particularly important when dealing with untrusted third-party code.

Group: Garbage Collector

Option(text): GC mark stack size

Option Name: com.microej.runtime.core.gc.markstack.levels.max

Default value: 32

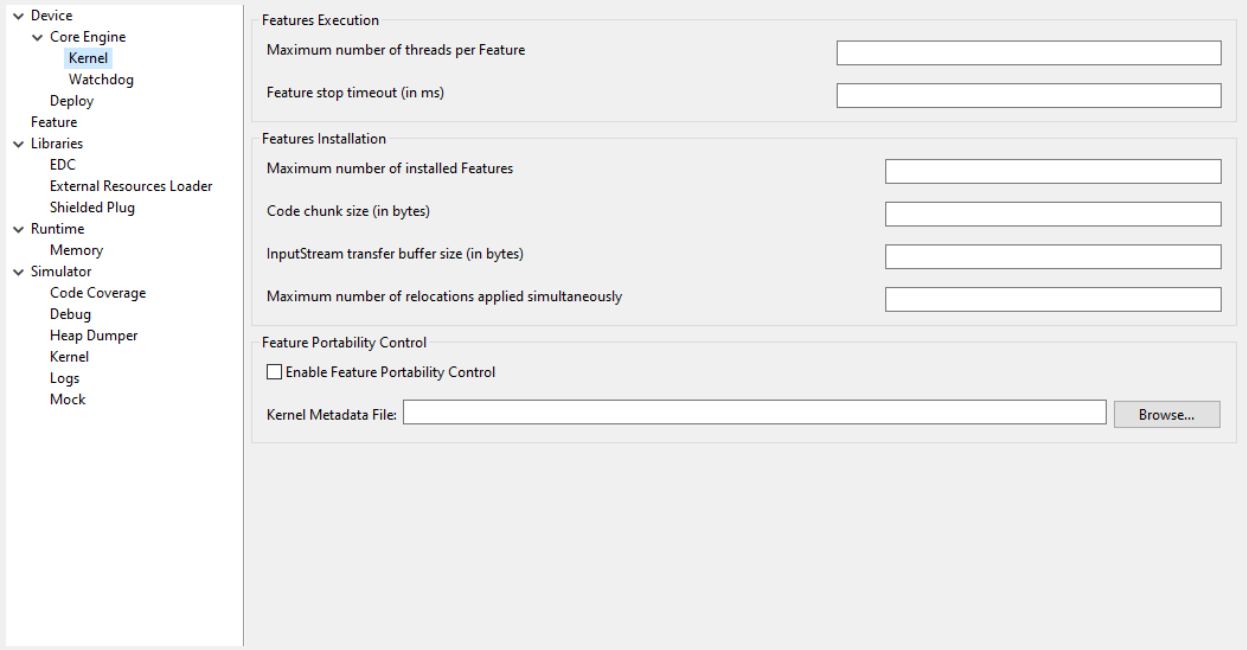
Description:

Indicates the quantity of items in the *Garbage Collector*'s mark stack. The mark stack is used by the Garbage Collector for identifying live objects within the heap through a depth-first search approach. Once the mark stack reaches its capacity, the Garbage Collector proceeds to inspect heap memory, which may slow down garbage collection performance.

You can receive a notification when the mark stack limit is reached by implementing the following hook:

```
void LLMJVM_on_GC_MarkStackOverflow_reached(void) {
  // When entering here, the GC mark stack is undersized, which may affect GC performance.
  // It is recommended to either increase the GC mark stack size or reduce the object graph depth.
}
```

Category: Kernel



Group: Threads

Option(text): Maximum number of threads per Feature

Option Name: core.memory.feature.max.threads

Default value: 5

Description:

Specifies the maximum number of threads a Feature is allowed to use at the same time.

Option(text): Feature stop timeout

Option Name: com.microej.runtime.kf.waitstop.delay

Default value: 2000

Description:

Specifies the maximum time allowed for the FeatureEntryPoint.stop() method to return (value in milliseconds).

Group: Features Installation

Option(text): Maximum number of installed Features

Option Name: com.microej.runtime.kernel.dynamicfeatures.max

Default value: 16

Description:

Specifies the maximum number of Features that can be installed to this Kernel (see Kernel.install() method).

Option(text): Code chunk size

Option Name: com.microej.soar.kernel.featurecodechunk.size

Default value: 65536

Description:

Specifies the size in bytes of the code chunk in RAM. See *Code Chunk Size* section for more details.

Option(text): InputStream transfer buffer size

Option Name: com.microej.runtime.kf.link.transferbuffer.size

Default value: 512

Description:

Specifies the size in bytes of the temporary byte array for reading in the Feature InputStream. See *InputStream Transfer Buffer Size* section for more details.

Option(text): Maximum number of relocations applied simultaneously

Option Name: com.microej.runtime.kf.link.chunk.relocations.count

Default value: 128

Group: Feature Portability Control

Option(checkbox): Enable Feature Portability Control

Option Name: com.microej.soar.kernel.featureportabilitycontrol.enabled

Default value: false

Option(browse): Kernel Metadata File

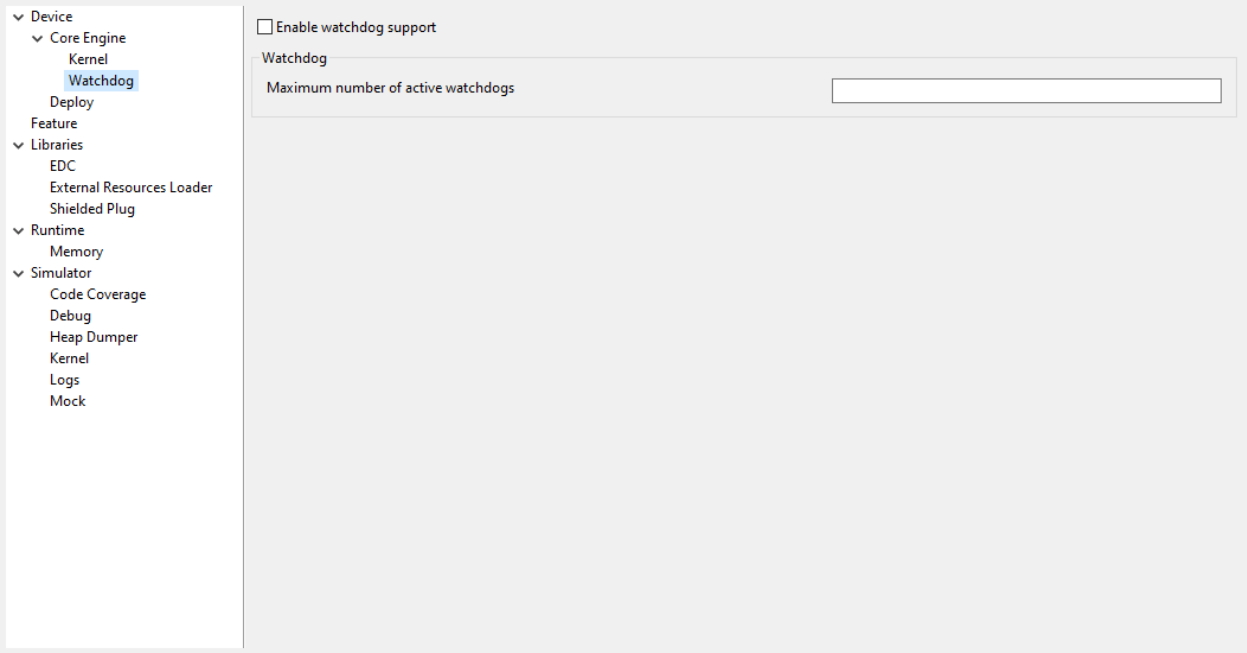
Option Name: com.microej.soar.kernel.featureportabilitycontrol.metadata.path

Default value: (empty)

Description:

Specifies the path to the Kernel metadata file for Feature Portability Control.

Category: Watchdog



Option(checkbox): Enable watchdog support

Option Name: enable.watchdog.support

Default value: true

Group: Watchdog

Option(text):

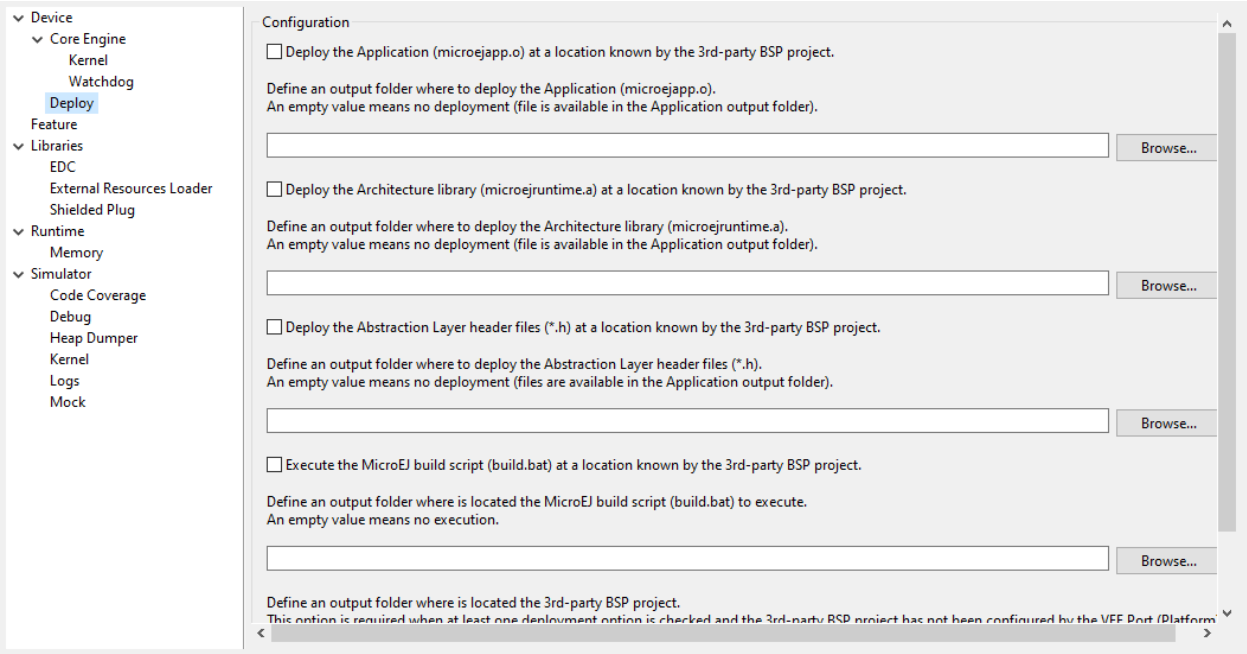
Option Name: maximum.active.watchdogs

Default value: 4

Description:

Specifies the maximum number of active watchdogs at the same time.

Category: Deploy



Description:

Configures the output locations where store the Application, the Architecture libraries and Abstraction Layer header files.

See *Board Support Package (BSP) connection* chapter for more details.

Group: Configuration

Option(checkbox): Deploy the Application (microejapp.o) at a location known by the 3rd-party BSP project.

Option Name: deploy.bsp.microejapp

Default value: true

Description:

Deploy the Application (microejapp.o) at a location known by the 3rd-party BSP project.

Option(browse):

Option Name: deploy.dir.microejapp

Default value: (empty)

Description:

Choose an output folder where to deploy the Application. An empty value means no deployment (file is available in the Application output folder).

Option(checkbox): Deploy the Architecture library (microejruntime.a) at a location known by the 3rd-party BSP project.

Option Name: deploy.bsp.microejlib

Default value: true

Description:

Deploy the Architecture library (microejruntime.a) at a location known by the 3rd-party BSP project.

Option(browse):

Option Name: deploy.dir.microejlib

Default value: (empty)

Description:

Choose an output folder where to deploy the Architecture library. An empty value means no deployment (file is available in the Application output folder).

Option(checkbox): Deploy the Abstraction Layer header files (*.h) at a location known by the 3rd-party BSP project.

Option Name: deploy.bsp.microejinc

Default value: true

Description:

Deploy the Abstraction Layer header files (*.h) at a location known by the 3rd-party BSP project.

Option(browse):

Option Name: deploy.dir.microejinc

Default value: (empty)

Description:

Choose an output folder where to deploy the Architecture library. An empty value means no deployment (file is available in the Application output folder).

Option(checkbox): Execute the MicroEJ build script (build.bat) at a location known by the 3rd-party BSP project.

Option Name: deploy.bsp.microejscript

Default value: false

Description:

Execute the MicroEJ build script (build.bat) at a location known by the 3rd-party BSP project.

Option(browse):

Option Name: deploy.dir.microejscript

Default value: (empty)

Description:

Choose an output folder where is located the MicroEJ build script (build.bat) to execute. An empty value means no execution.

Option(browse):

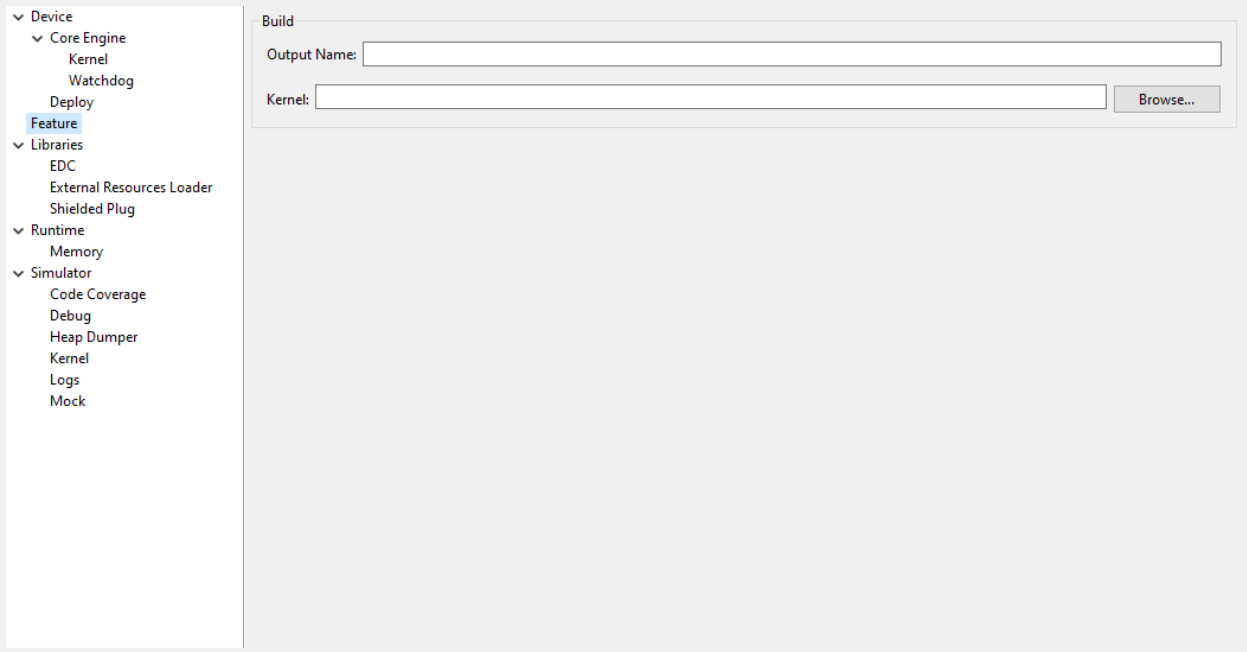
Option Name: deploy.bsp.root.dir

Default value: (empty)

Description:

Choose an output folder where is located the 3rd-party BSP project. An empty value means not set (3rd-party BSP project location may have been configured by the VEE Port).

Category: Feature



Description:

Specify Feature options

Group: Build

Option(text): Output Name

Option Name: feature.output.basename

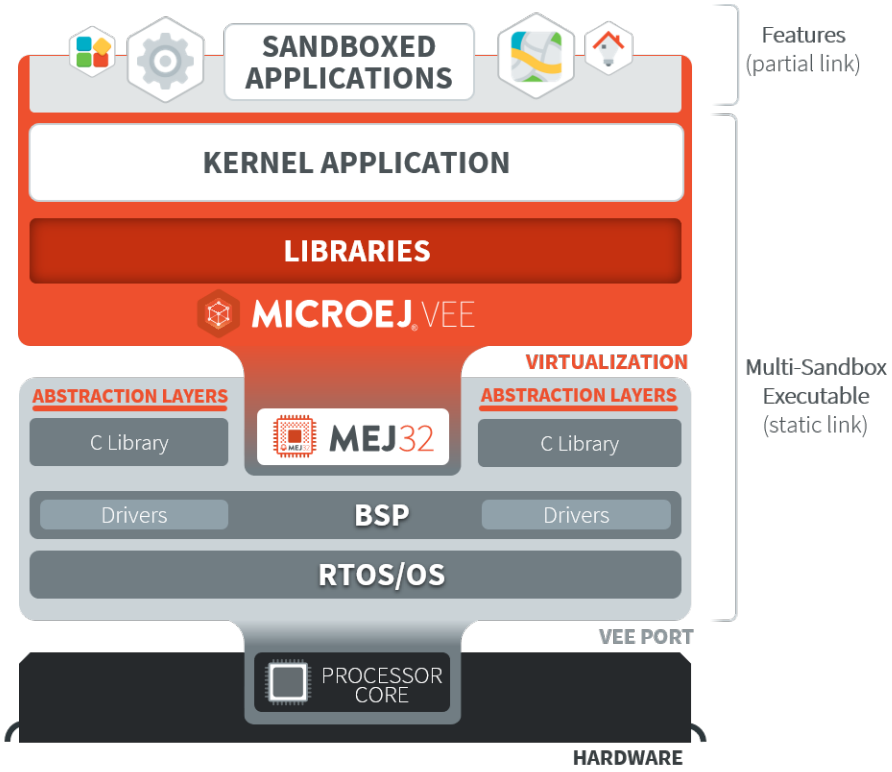
Default value: application

Option(browse): Kernel

Option Name: kernel.filename

Default value: (empty)

3.5.8 Sandboxed Application



Fundamental Concepts

Multi-Sandboxing is based on the *Kernel & Features Specification (KF)*.

It allows an application code to be split between multiples parts:

- the main application, called the Kernel,
- zero or more applications called Features.

Therefore, a Kernel Application relates to the *Kernel* concept and a Sandboxed Application relates to the *Feature* concept.

Some fundamental points:

- The Kernel is mandatory. It is assumed to be reliable, trusted and cannot be modified.
- A Feature is an application “extension” managed by the Kernel.
- A Feature is fully controlled by the Kernel: it can be installed, started, stopped and uninstalled at any time independent of the system state (particularly, a Feature never depends on another Feature to be stopped).
- A Feature is optional, potentially not-trusted, maybe unreliable and can be executed without jeopardizing the safety of the Kernel execution and other Features.
- Resources accesses (RAM, hardware peripherals, CPU time, …) are under control of the Kernel.

Note: You can go further by reading the *Kernel & Features Specification (KF)*.

Shared Interfaces

Principle

The Shared Interface mechanism provided by the Core Engine is an object communication bus based on plain Java interfaces where method calls are allowed to cross Sandboxed Applications boundaries without relying on Kernel APIs.

The Shared Interface mechanism is the cornerstone for designing reliable Service Oriented Architectures. Communication is based on the sharing of interfaces defining APIs (Contract Oriented Programming).

The basic schema:

- A provider application publishes an implementation for a shared interface into a system registry.
- A user application retrieves the implementation from the system registry and directly calls the methods defined by the shared interface.

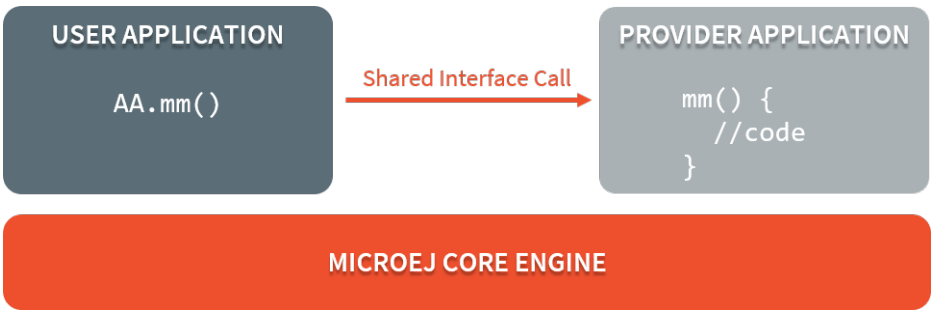


Fig. 126: Shared Interface Call Mechanism

The Shared Interface mechanism is based on automatic proxy objects created by the Core Engine. This offers a reliable way for users to handle broken links in case the provider application has been stopped or uninstalled.

Applications with a Shared Interface must provide a dedicated implementation (called the Proxy class implementation). Its main goal is to perform the remote invocation and provide a reliable implementation regarding the interface contract even if the remote application fails to fulfill its contract (unexpected exceptions, application killed, ...). The Core Engine will allocate instances of this Proxy class when an implementation (of the Shared Interface) owned by another application is being transferred to this application.

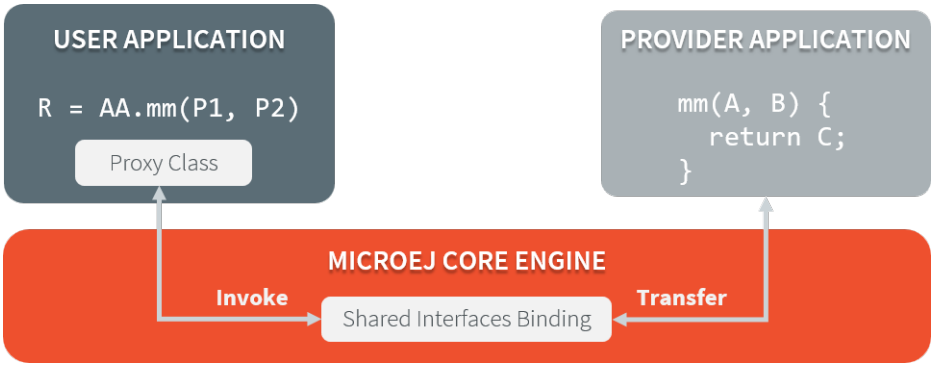


Fig. 127: Shared Interfaces Proxy Overview

This mecanism is formally specified in the *Kernel & Features Specification (KF)*.

Shared Interface Usage

Usage of a Shared Interface follows these steps:

1. Define the Shared Interface:
    1. Define the Java interface
    2. Implement the proxy for the interface
    3. Register the interface as a Shared Interface
2. From the provider application,
    1. Create an instance of this Shared Interface
    2. Register the instance to a KF service registry
3. From the consumer application,
    1. Retrieve a proxy of the instance from the KF service registry
    2. Call methods of the instance proxy.

Define the Shared Interface

Define the Java Interface

The definition of a Shared Interface starts by defining a standard Java interface. For example:

```
package mypackage;
public interface MyInterface {
    void foo();
}
```

A Shared Interface includes all methods it declares, along with those inherited from its super types. It can extend any interface, including Feature interfaces (which may or may not be declared as Shared Interfaces) and Kernel interfaces.

Some restrictions apply to Shared Interfaces compared to standard Java interfaces:

- Types for parameters and return values must be *transferable types*;
- Thrown exceptions must be classes owned by the Kernel.

Implement the Proxy Class

A proxy class is implemented and executed on the client side.

with the following specification:

- its fully qualified name is the shared interface fully qualified name append with `Proxy`.
- it extends the Proxy class.
- it implements the Shared Interface
- it provides an implementation of all interface methods

Each method of the implemented interface must be defined according to the following pattern:

```
package mypackage;

public class MyInterfaceProxy extends Proxy<MyInterface> implements MyInterface {

    // Use the "invoke()" method for "void" returns.
    @Override
    public void foo(){
        try {
            invoke(); // perform remote invocation
        } catch (Throwable e) {
            // Handle any errors thrown during the remote call, including dead Feature.
            // Implement a behavior that complies with the method specification.
            // i.e. return a valid error code or throw a documented exception.
            // Logging traces for debug can also be added here.
        }
    }

   // Use the "invokeRef()" method when returning an object reference.
   @Override
   public String getName() {
      try {
         return (String) invokeRef();
      } catch (Throwable e) {
         e.printStackTrace();
         return MyInterface.UNKNOWN_NAME;
      }
   }

   // Use the "invokeInt()" method when returning an "int" basetype.
   @Override
   public int getValue() {
      try {
         return invokeInt();
      } catch (Throwable e) {
         e.printStackTrace();
         return -1;
      }
   }
}
```

Each implemented method of the proxy class is responsible for performing the remote call and catching all errors from the server side and to provide an appropriate answer to the client application call according to the interface method specification (contract).

The *Proxy class implementation* section documents how to perform the remote invocation.

Register the Shared Interface

To declare an interface as a Shared Interface, it must be registered in a Shared Interfaces identification file. A Shared Interface identification file is an XML file with the `.si` filename extension and the following format:

```
<sharedInterfaces>
   <sharedInterface name="mypackage.MyInterface"/>
</sharedInterfaces>
```

Shared Interface identification files must be placed at the root of the application classpath, typically it is defined in the `src/main/resources` folder.

Use the Shared Interface at Runtime

Projects Structure

Both the consumer and the provider applications must have the Java interface, the proxy class and the identification file on the classpath in order to be able to use the Shared Interface.

Typically, the 3 files can be defined in an Add-On Library that both application projects depend on.

Create and Share an instance of a Shared Interface

The provider application can instantiate the Java interface. For example:

```
MyInterface myInstance = new MyInterface() {
    @Override
    public void foo() {
        System.out.println("Hello world!");
    }
    ...
};
```

In order to share the instance with other applications, the provider application must register the instance with some registry owned by the Kernel (see *Communication between Kernel and Feature* for details) like so:

```
ServiceFactory.register(MyInterface.class, myInstance);
```

Retrieve and Use a Proxy of a Shared Interface Instance

The consumer application can then retrieve the instance from the Kernel registry like so:

```
MyInterface otherAppInstance = ServiceFactory.getService(MyInterface.class);
// otherAppInstance is actually an instance of the proxy class owned by the consumer application
```

Then it can call the interface methods transparently:

```
otherAppInstance.foo(); // remote invocation through the proxy
```

Transferable Types

In the process of a cross-application method call, parameters and return value of methods declared in a Shared Interface must be transferred back and forth between application boundaries.

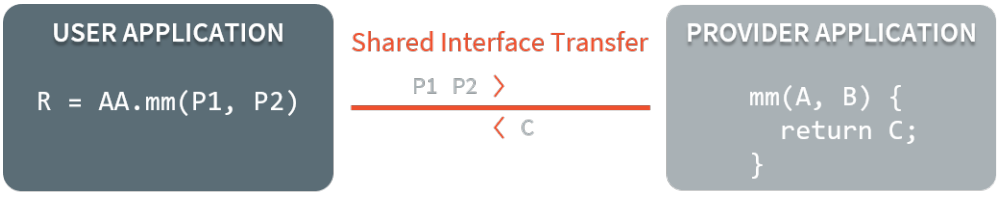


Fig. 128: Shared Interface Parameters Transfer

The following table describes the rules applied depending on the element to be transferred.

Table 10: Shared Interface Types Transfer Rules

Type	Owner	Instance Owner	Rule
Base type	N/A	N/A	Passing by value. (boolean , byte , short , char , int , long , double , float)
Any Class, Array or Interface	Kernel	Kernel	Passing by reference
Any Class, Array or Interface	Kernel	Application	Kernel specific. Converted to a target Feature object if the Kernel has registered a *Kernel type converter*, otherwise Forbidden.
Array of base types	Any	Application	Clone by copy
Arrays of references	Any	Application	Clone and transfer rules applied again on each element (recursively)
Shared Interface	Application	Application	Passing by indirect reference (Proxy creation).
Any Class, Array or Interface	Application	Application	Forbidden

Objects created by an Application which type is owned by the Kernel can be transferred to another Application provided this has been authorized by the Kernel. When an argument transfer is forbidden, the call is abruptly stopped and an java.lang.IllegalAccessError is thrown by the *Core Engine*.

The list of Kernel types that can be transferred is Kernel specific, so you have to consult your Kernel specification. For these types to be transferable, a dedicated *Kernel Type Converter* must have been registered in the Kernel.

The table below lists some well-known types that your Kernel likely can allow to be transferred through a Shared Interface, along with their behaviors:

Table 11: Transfer Rules for well-known Kernel Types

Type	Rule
java.lang.Boolean	Clone by copy
java.lang.Byte	Clone by copy
java.lang.Character	Clone by copy
java.lang.Short	Clone by copy
java.lang.Integer	Clone by copy
java.lang.Float	Clone by copy
java.lang.Long	Clone by copy
java.lang.Double	Clone by copy
java.lang.String	Clone by copy
java.io.InputStream	Create a Proxy reference
java.util.Date	Clone by copy
java.util.List<T>	Clone by copy with recursive element conversion
java.util.Map<K,V>	Clone by copy with recursive keys and values conversion

Implementing the Proxy Class

Remote invocation methods are defined in the super class ej.kf.Proxy and are named invokeXXX() where XXX is the kind of return type.

Table 12: Proxy Remote Invocation Built-in Methods

Invocation Method	Usage
void invoke()	Remote invocation for a proxy method that returns void
Object invokeRef()	Remote invocation for a proxy method that returns a reference
boolean invokeBoolean(), byte invokeByte(), char invokeChar(), short invokeShort(), int invokeInt(), long invokeLong(), double invokeDouble(), float invokeFloat()	Remote invocation for a proxy method that returns a base type

As this class is part of the Application, the developer has the full control on the Proxy implementation and is free to insert additional code such as logging calls and errors for example. It is also possible to have different proxy implementations for the same Shared Interface in different applications.

A Sandboxed Application is an Application that is partially linked, then deployed and executed over a Multi-Sandbox Executable.

Typical use cases for a Sandboxed Application are:

- over the air provisioning: the Application is dynamically installed or updated on a fleet of heterogenous devices.
- modularization: a monolithic application is split into multiple Sandboxed Applications; each of them can be started or stopped separately.

The following figure shows the general process of building a Sandboxed Application.

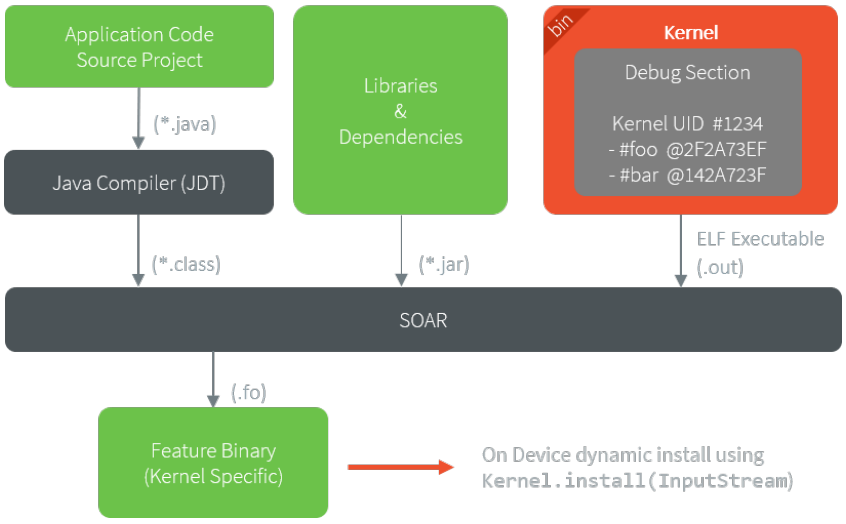


Fig. 129: Sandboxed Application Link Flow

Please refer to the *Kernel Developer Guide* to learn more on writing Kernel Applications and building Multi-Sandbox Executable and Virtual Devices.

AppConnect Application Management

AppConnect is an embedded library designed to simplify the management of on-board Sandboxed Applications directly on the device. It provides services to install, start, stop, and uninstall applications, as well as to retrieve metadata about both deployed applications and the target device. The library includes a full-fledged HTTP server, providing a comprehensive REST API along with a web-based graphical interface.

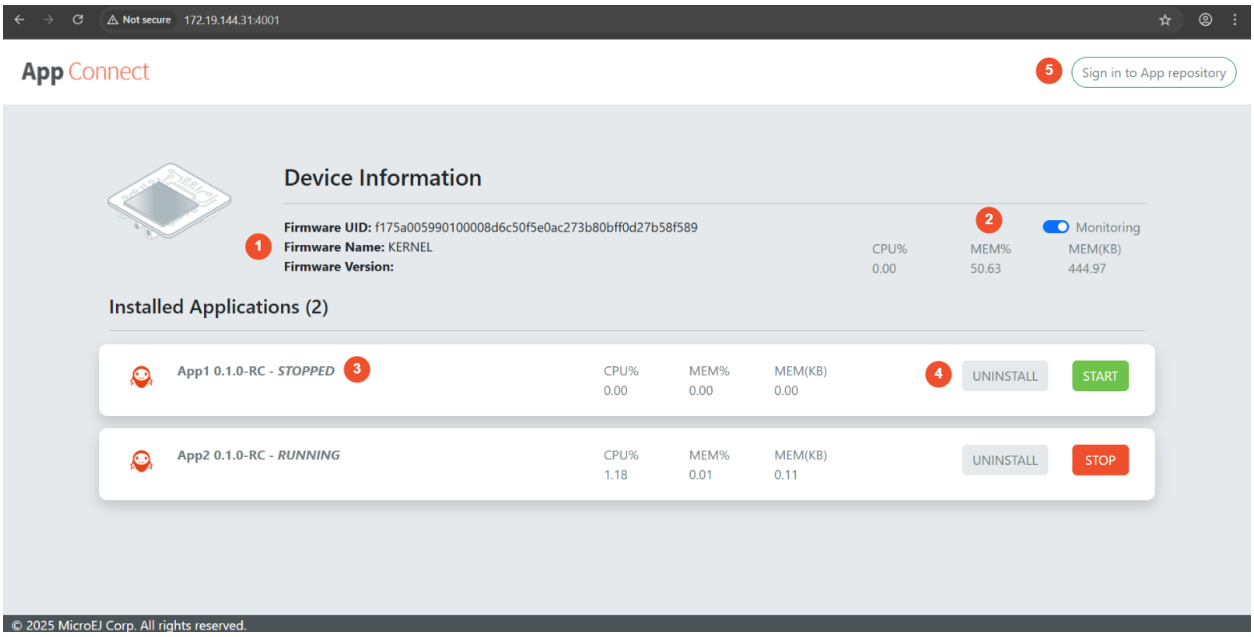
AppConnect supports four modes of interaction:

- AppConnect Web App.
- AppConnect REST API.
- AppConnect HTTP CLI.
- AppConnect Serial Shell.

AppConnect Web App

Installed Applications are listed on the Web UI, and they can be started, stopped, or uninstalled.

The screenshot below highlights the following elements.



1. Device information that displays the firmware UID, name and version.
2. CPU and RAM monitoring when supported by the device and the applications.
3. The current state of each application (RUNNING , STOPPED).
4. Action buttons to start, stop, or uninstall applications.
5. Connection to the MicroEJ App Store.

AppConnect REST API

The REST API exposes the following endpoints to manage the board and its installed applications:

Device Information

Retrieve information about the current device, including forge evaluation settings (URL, username, password) and the runtime type (board, simulator).

Endpoint: `GET /api/device`.

List Installed Applications

Return a list of Applications installed on the device along with their metadata. Metadata includes the Application name, description, and current state.

Endpoint: `GET /api/app`.

Get Application Icon

Return the Application icon (if it is set in the Application's metadata).

Endpoint: `GET /api/app/icon?id=(application identifier)&preferredSize=(preferred size, default 64)`.

Install Applications

Install an application on the device.

Endpoint: `POST /api/app/install?force=(true|false)&start=(true|false)`.

Required parameters: `binary`.

Form data:

- `binary`: The Application binary compatible with the target board or the simulator (e.g., `.fo` or `.fs3`).

Optional query parameters:

- `name`: The name of the Application, used to identify and overwrite an existing application when `force` is set to `true`.
- `force`: If `true`, uninstall any existing application with the same name before installing the new one. Default: `false`.
- `start`: If `true`, automatically start the application after installation. Default: `false`.

Manage Applications

Start, stop, or uninstall an installed Application.

Endpoint: POST /api/app/manage?id=(application identifier)&action=(start|stop|uninstall) .

Required parameters: id , action .

AppConnect HTTP CLI

AppConnect CLI is a Java-based command-line (CLI) tool that serves as a headless interface, offering the full functionality of the web application through its REST API, while enabling seamless automation and scripting directly from the command line.

The CLI is available for download at the following link.

Download the latest version of appconnect-cli-<version>.jar from this page and follow the usage instructions outlined below.

Requirements

AppConnect CLI requires JDK 17 or higher.

Usage

To start AppConnect CLI, run the following command:

```
java -jar path/to/appconnect-cli-*.jar
```

```
                  _______                        __     _______ __     _______
    /\           / _____|                      | |    / _____| |   |_   _|
   /  \   _ ___  _ ___   | |    ____  _ ___  _ ___   ____  ____| |_  | |   | |     | |
  / /\ \ | '_ \| '_ \  | |   / _ \| '_ \| '_ \ / _ \/ ___| ___| | | | |   | |     | |
 / ______ \| |_) | |_) | | |____| (_) | | | | | | |  ___/ (___| |_  | |_____| |_____ _| |_
/_/    \_\ .___/| .___/   \_______\____/|_| |_|_| |_|\____|\____|\___|  \_______|_______|_____|
         | |   | |
         |_|   |_|

Welcome to the appconnect-cli 3.2.0!
Type 'help' to view the list of available commands and their descriptions.
For detailed command usage, type 'help <command>'.
```

After starting the CLI, connect to your device using the command:

```
connect <ip>
```

Once the connection is established, you can execute various commands to manage applications.

```
* connect: Connect to AppConnect server running on board.
* start: Start an app.
* stop: Stop an app.
* install: Install an app.
* uninstall: Uninstall an app.
```

(continues on next page)

(continued from previous page)

```
* device-info: Get information about the device such as firmware ID, name, version, etc.
* list, ls: List information about applications such as PID, name, version, state.
```

For detailed information on any command, use the `--help` option with that command.

AppConnect Serial Shell

The `appconnect-serial` module provides a shell for use over a serial connection (UART). It reads commands from an input stream, and writes execution results to an output stream.

Requirements

AppConnect Serial requires the Serial Pack and API 2.0.0 or higher.

Usage

The `appconnect-serial` can be integrated into a Sandboxed Application as follows:

1. Add the `appconnect-serial` dependency directly to your Application. Alternatively, you can add it to the *Runtime Environment* of your Kernel.
2. If the security manager is enabled and enforcing permission checks, the Application must be granted the following permissions.

```
{
  "permissions": {
    "com.microej.library.appconnect.ApplicationServicePermission": {
      "*": ["list", "start", "stop", "install", "uninstall"]
    },
    "com.microej.library.appconnect.DeviceServicePermission": {
      "*": ["device", "evaluation"]
    }
  }
}
```

Note: For more information about permissions, please refer to this *section*.

3. Create an instance of `AppConnectShell` and provide the input and output streams of your serial connection.

```
AppConnectShell shell = new AppConnectShell(inputStream, outputStream);
try {
  shell.start();
  shell.loop();
} catch (IOException exception) {
  shell.stop();
}
```

4. Deploy the Application.

The AppConnect shell can be accessed using Minicom or a similar tool.

Once the Application is deployed and started, you should see the following output:

```
   _____                                  _    _____                  _    _
  /\                 / ______|                    | |    / ______|          (_)     | |
 /  \    _ ___   _ ___    | |     ____   _ ___   _ ___    ____  _____| |_   | (____    ____ _ _ ___ _   ___ _| |
/ /\ \ | '_ \| '_ \  | |    / _ \| '_ \ / _ \ / ___| ___|  \____ \ / _ \ '___| |/ _` | |
/ ______ \| |_) | |_) | | |____| (_) | | | | | | ___/ (___| |_   _____) |  ___/ |  | | (_| | |
/_/    \_\ .___/| .___/   \_____\___/|_| |_|_| |_|\____|\____|\___| |_____/ \____|_|  |_|\___,_|_|
         | |   | |
         |_|   |_|

Welcome to appconnect-serial 3.2.0!
Type 'help' to view the list of available commands and their descriptions.
For detailed command usage, type 'help --command <command>'.
```

The following commands are available:

```
device-info: Get information about the device such as firmware ID, name, version, etc.
uninstall: Uninstall an application.
exit: Exit the shell.
help: Display information about available commands.
list: List information about the installed applications such as name, PID, version, state, etc.
stop: Stop an application.
start: Start an application.
evaluation-info: Get information about MicroEJ forge URL and credentials.
install: Install an application.
```

3.5.9 Character Encoding

Default Encoding

The default character encoding is ISO-8859-1 . It is thus the encoding used when:

- creating a new string from a byte array without specifying the encoding (String(byte[]) constructor),
- getting the byte array from a string without specifying the encoding (String.getBytes() method),
- printing a string to standard output stream (System.out),
- creating a new PrintStream without specifying the encoding.

UTF-8 Encoding

EDC provides an implementation of the UTF-8 character encoding. It can be embedded using the *Embed UTF-8 encoding option* (otherwise a java.io.UnsupportedEncodingException exception will be thrown).

This implementation also supports Unicode code points as supplementary characters, by setting the *constant* com.microej.library.edc.supplementarycharacter.enabled to true .

Custom Encoding

It is possible to connect additional custom encodings. Please contact *our support team* for more details.

Console Output

By default, the standard output stream (System.out) uses ISO-8859-1 encoding to print strings. If you want to print a string with a different encoding, you can create a new PrintStream:

```
PrintStream outUtf8 = new PrintStream(System.out, true, "UTF-8");
outUtf8.println("  ");
```

Warning: Make sure you embed the UTF-8 encoder (see *UTF-8 Encoding*)

The print methods write the raw byte array with the encoding used by the PrintStream to the console. The console must then be configured with the same encoding to display characters properly.

Set Encoding in MicroEJ SDK Console

The default encoding for Eclipse consoles is UTF-8. If your application prints non-ASCII characters, they may not be displayed properly.

The encoding used by a console for a given application can be set in the application launcher options: Run > Run Configurations… , and then Common tab > Encoding radio buttons.

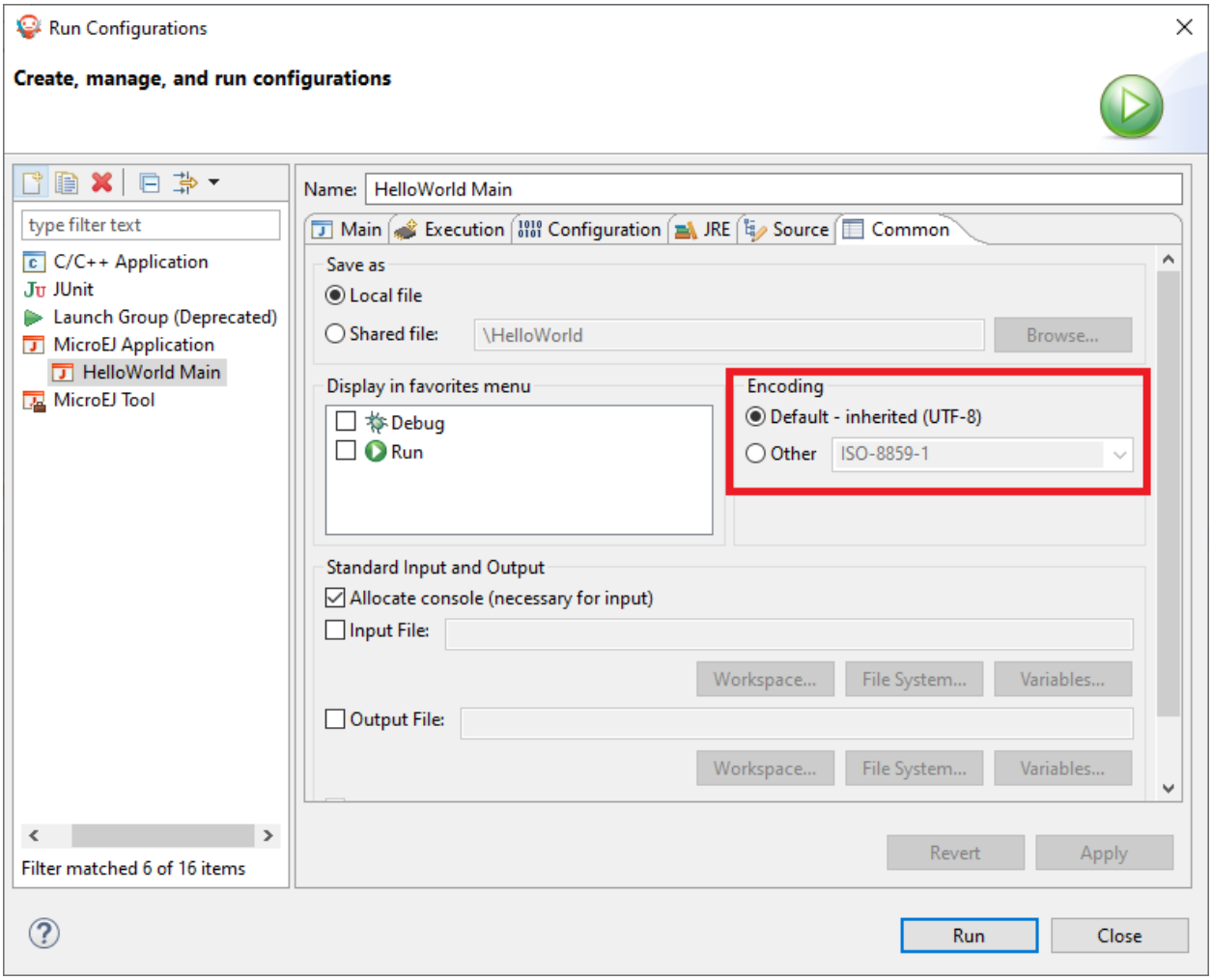


Fig. 130: Eclipse Launcher Console Encoding Options

3.5.10 Limitations

The following table lists the limitations of MicroEJ Architectures version 8.4.0 or higher, for both Evaluation and Production usage. Please consult *Architectures Changelog* for limitations changes on former versions.

Note: The term *unlimited* means there is no Architecture specific limitation. However, there may be limitations driven by device memory layout. Please refer to your VEE Port specific documentation to get the memory mapping of *Core Engine sections*.

Table 13: Architecture Limitations

Item	EVAL	PROD
[Mono-Sandbox] Number of concrete types[1]	8192	8192
[Multi-Sandbox] Number of concrete types per context[Page 469, 1]	4096	4096
Number of abstract classes and interfaces	unlimited	unlimited
Class or Interface hierarchy depth	127	127
Number of methods	unlimited	unlimited
Method size in bytes	65536	65536
Numbers of exception handlers per method	63	63
Number of parameters for an SNI method	15	15
Number of instance fields[2] (Base type)	4096	4096
Number of instance fields[2] (References)	31	31
Number of static fields (boolean + byte)	65536	65536
Number of static fields (short + char)	65536	65536
Number of static fields (int + float)	65536	65536
Number of static fields (long + double)	65536	65536
Number of static fields (References)	65536	65536
[Mono-Sandbox] Number of threads	63	63
[Multi-Sandbox] Number of threads	127	127
[Mono-Sandbox] Number of held monitors[4]	63	63
[Multi-Sandbox] Number of held monitors[4]	127[3]	127[3]
Maximum Managed Heap Memory Size in bytes	67108860	67108860
Time limit	60 minutes	unlimited
Number of methods and constructors calls	500000000	unlimited
Number of Managed Heap Garbage Collection	3000[4]	unlimited
Number of Shielded Plug databases	unlimited	unlimited
Number of blocks per Shielded Plug database	32767	32767

3.5.11 GitHub Repositories

A large number of examples, libraries, demos and tools are shared on MicroEJ GitHub account: https://github.com/MicroEJ.

Most of these GitHub repositories contain projects ready to be imported in MicroEJ SDK.

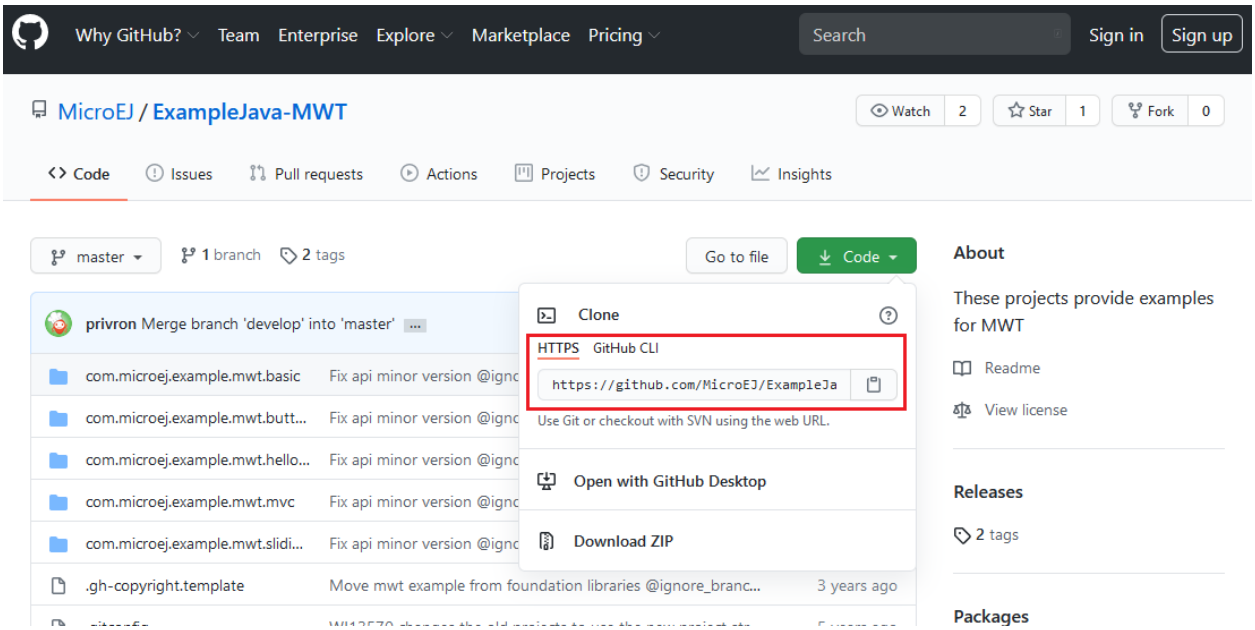
Repository Import

This section explains the steps to import a Github repository in MicroEJ SDK, illustrated with the MWT Examples repository.

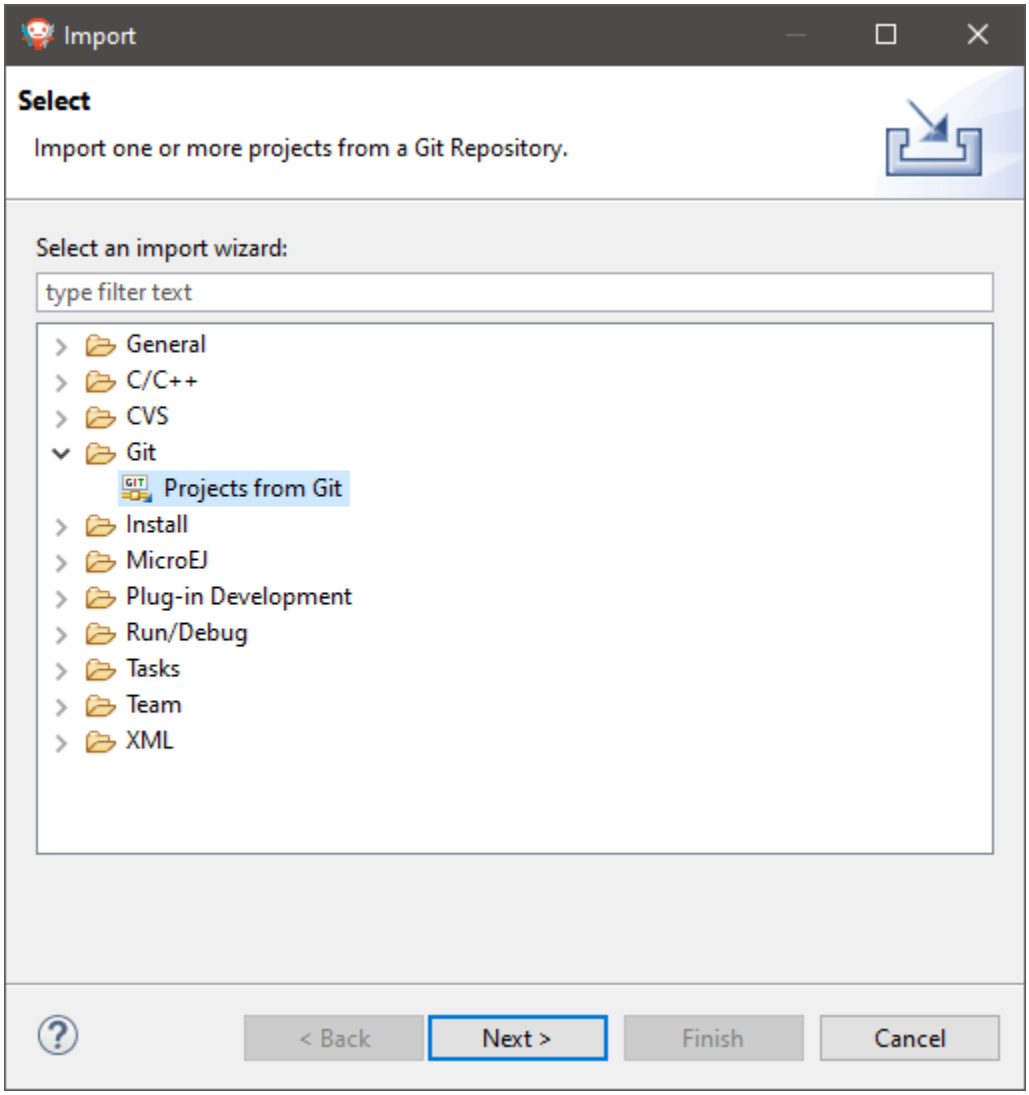
Note: MicroEJ SDK Distribution includes the Eclipse plugin for Git.

First, from the GitHub page, copy the repository URI (HTTP address) from the dedicated field in the right menu (highlighted in red):

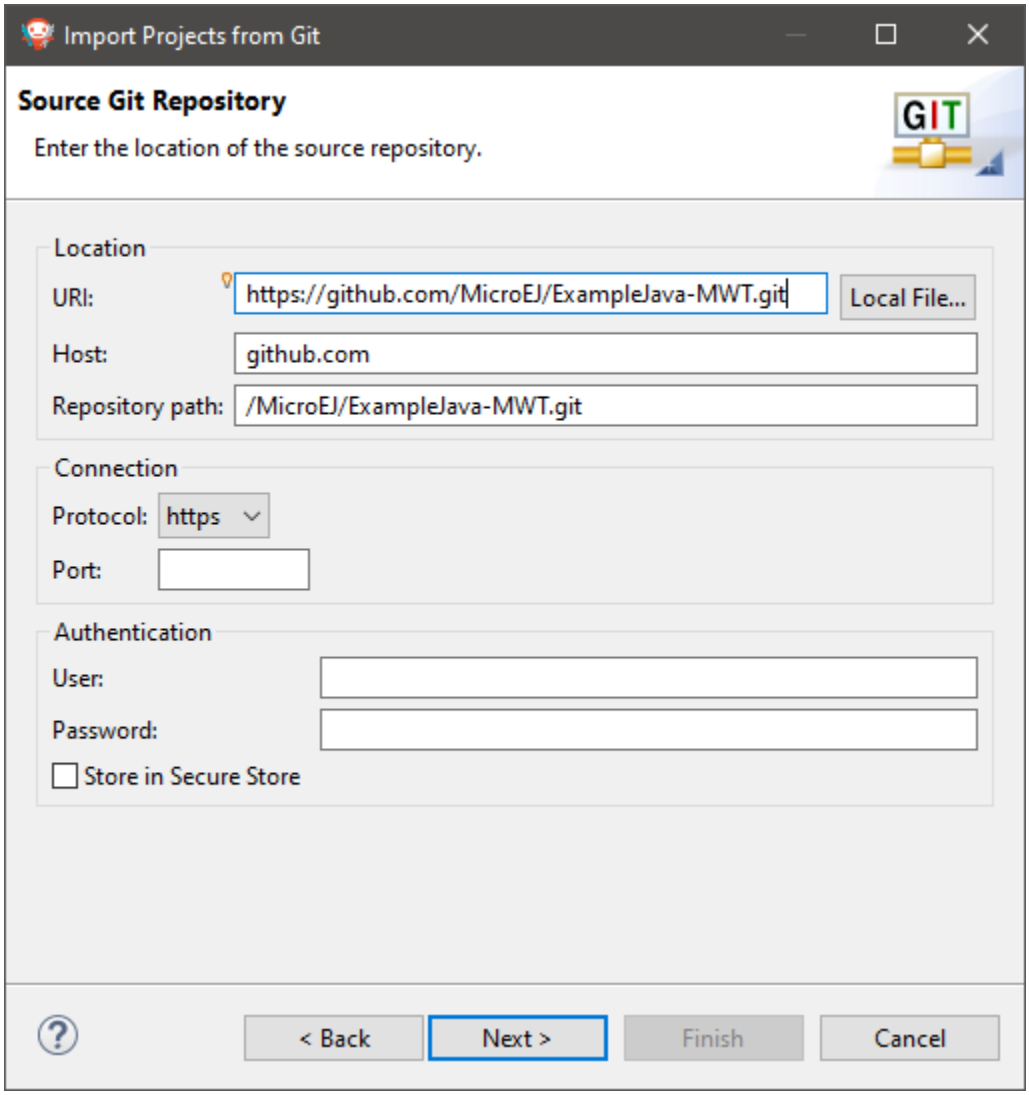
[1] Concrete types are classes and arrays that can be instantiated.
[2] All instance fields declared in the class and its super classes.
[4] The Managed Heap Garbage Collection limit may throw unexpected cascading java.lang.OutOfMemoryError exceptions before the Core Engine exits.
[3] The maximum number of different monitors that can be held by one thread at any time is defined by the maximum number of monitors per thread Application option.



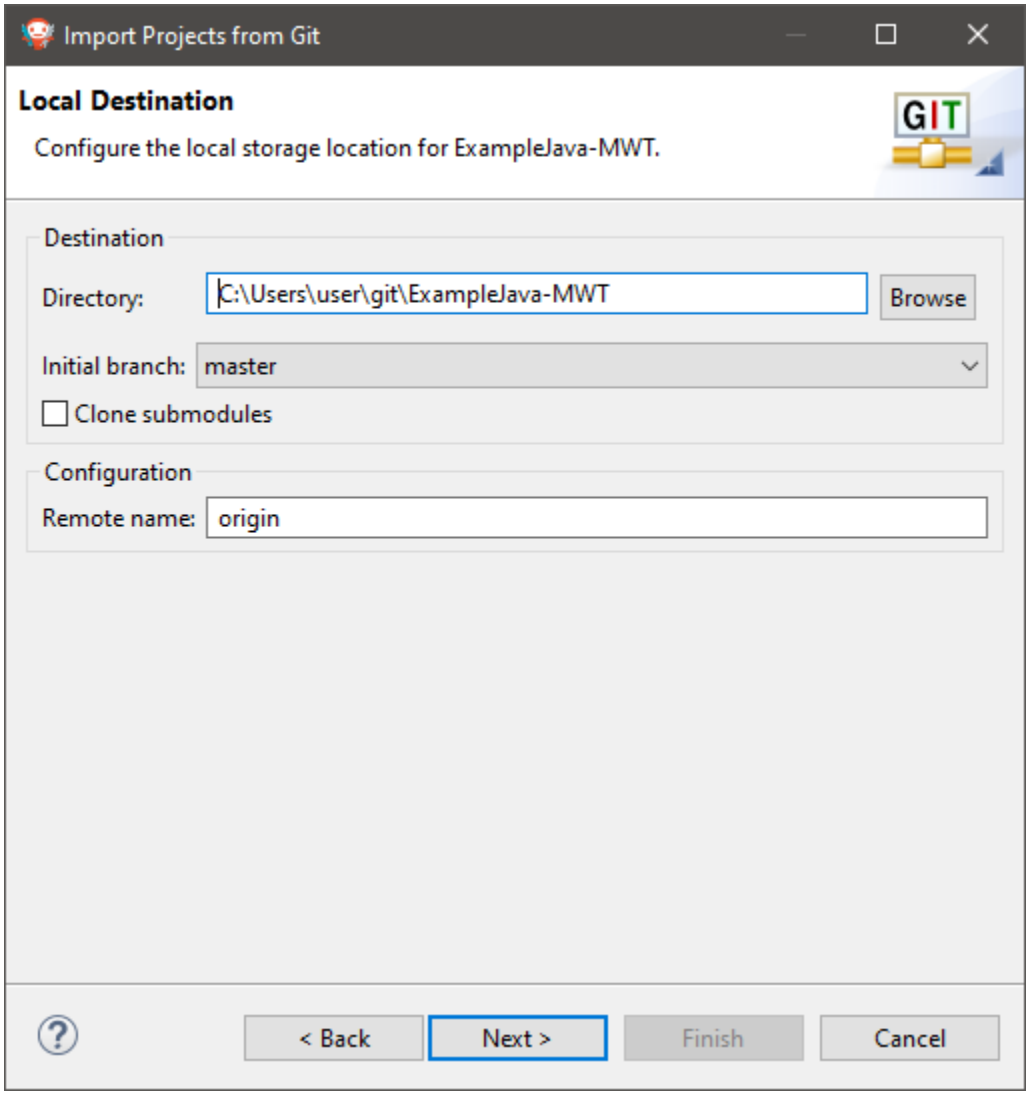
In the SDK, to clone and import the project from the remote Git repository into the MicroEJ workspace, select File > Import > Git > Projects from Git wizard.



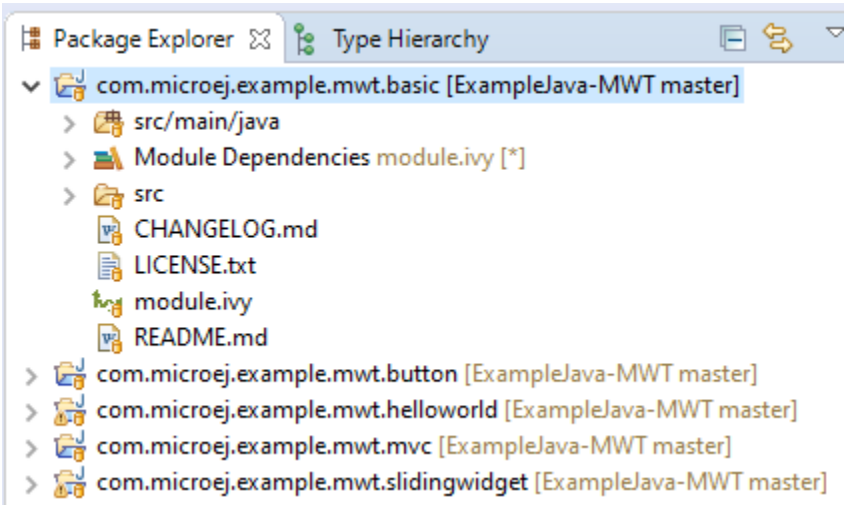
Click Next , select Clone URI , click Next and paste the remote repository address in the URI field. For this repository, the address is https://github.com/MicroEJ/ExampleJava-MWT.git. If the HTTP address is a valid repository, the other fields are filed automatically.



Click Next , select the master branch, click Next and accept the proposed *Local Destination* by clicking Next once again.



Click Next once more and finally Finish . The Package Explorer view now contains the imported projects.



If you want to import projects from another (GitHub) repository, you simply have to do the same procedure using the Git URL of the desired repository.

MicroEJ GitHub Badges

MicroEJ GitHub Badges are badges embedded in a README at the root of the repository. They highlight the compatibilities of the repository at a quick glance with MicroEJ Architecture, MicroEJ SDK and Graphical User Interface versions.

The color of the badge has the following meaning:

- Green means a current supported version:

- Orange means a still supported version that will be deprecated in the future:

- Gray means a deprecated version:

3.5.12 Module Repositories

This chapter describes the *module repositories* provided by MicroEJ Corp.

Central Repository

The Central Repository is the module repository distributed and maintained by MicroEJ Corp. It contains a selection of production-grade modules such as *Libraries* and *Packs*.

Use

By default, the SDK is configured to import modules from the online Central Repository.

You can manually browse the repository at https://repository.microej.com/modules/.

Before starting to develop production code, it is strongly recommended to import the repository to your local environment. Please follow the steps described at https://developer.microej.com/central-repository/.

Licensing

The Central Repository is a set of modules distributed under various software licenses, including the *SDK EULA* and the *Commercial Component License* for some of them. Please consult the LICENSE.txt file attached to each module.

Changelog

The Central Repository content is versioned. The overall changelog is available at https://repository.microej.com/ and describes modules additions or removals. For module content changes, please consult the `CHANGELOG.md` file attached to each module.

Java APIs (Javadoc)

To consult the Java APIs documentation (Javadoc) of all *libraries* available in the repository, please visit https://repository.microej.com/javadoc/microej_5.x/apis/.

Developer Repository

The developer repository is an online repository hosted by MicroEJ Corp., contains community modules provided "as-is". It is similar to what Maven Central Repository are for hosting Java standard modules.

MicroEJ Corp. contributes to the developer repository in the following cases:

- Demos (VEE Ports, Kernel Executables, Virtual Devices, Applications),
- Incubating Libraries,
- Former Central Repository versions,
- Hardware specific modules.

Use

By default, the SDK is configured to import modules from the developer repository[1].

You can also manually browse the repository at https://forge.microej.com/artifactory/microej-developer-repository-release/.

Before starting to develop production code, it is strongly recommended to transfer the desired modules to your local environment by creating your own *module repository* copy.

Licensing

The developer repository is a set of modules distributed under various software licenses. Please consult the `LICENSE.txt` file attached to each module.

Changelog

The developer repository is populated from multiple sources, thus there is no changelog for the whole repository content as it is the case of the Central Repository.

Please consult the `CHANGELOG.md` file attached to each module.

[1] Require SDK version `5.4.0` or higher.

Javadoc

To consult the APIs documentation (Javadoc) of *libraries* available in the developer repository, please consult the javadoc attached to each module.

Community

The developer repository can host modules developed by the community. If your organization plan to develop such module, please contact *our support team* to get dedicated credentials for publication.

Content Organization

The following table describes how are organized the *modules natures* within the repository.

Table 14: Modules Organization

Organization	Module Nature
ej.api , com.microej.api	Foundation Library API
com.microej.architecture	*Architecture*
com.microej.pack	*Pack*
ej.tool , com.microej.tool	Tool or Add-On processor
Any other	Add-On Library

3.5.13 Libraries

Two types of libraries are supported:

- Foundation Libraries are MicroEJ Core libraries that provide core runtime APIs or hardware-dependent functionality.
- Add-On Libraries are MicroEJ libraries that are implemented on top of MicroEJ Foundation Libraries (100% full Managed code).

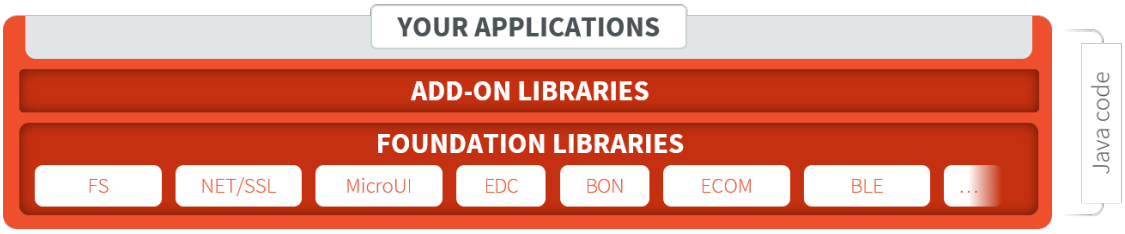


Fig. 131: MicroEJ Foundation Libraries and Add-On Libraries

MicroEJ Corp. provides a large number of libraries through the *MicroEJ Central Repository*. To consult its libraries APIs documentation, please visit https://developer.microej.com/microej-apis/.

Graphical User Interface

This section presents libraries relative to the user interface.

The following schema shows the overall architecture and modules:

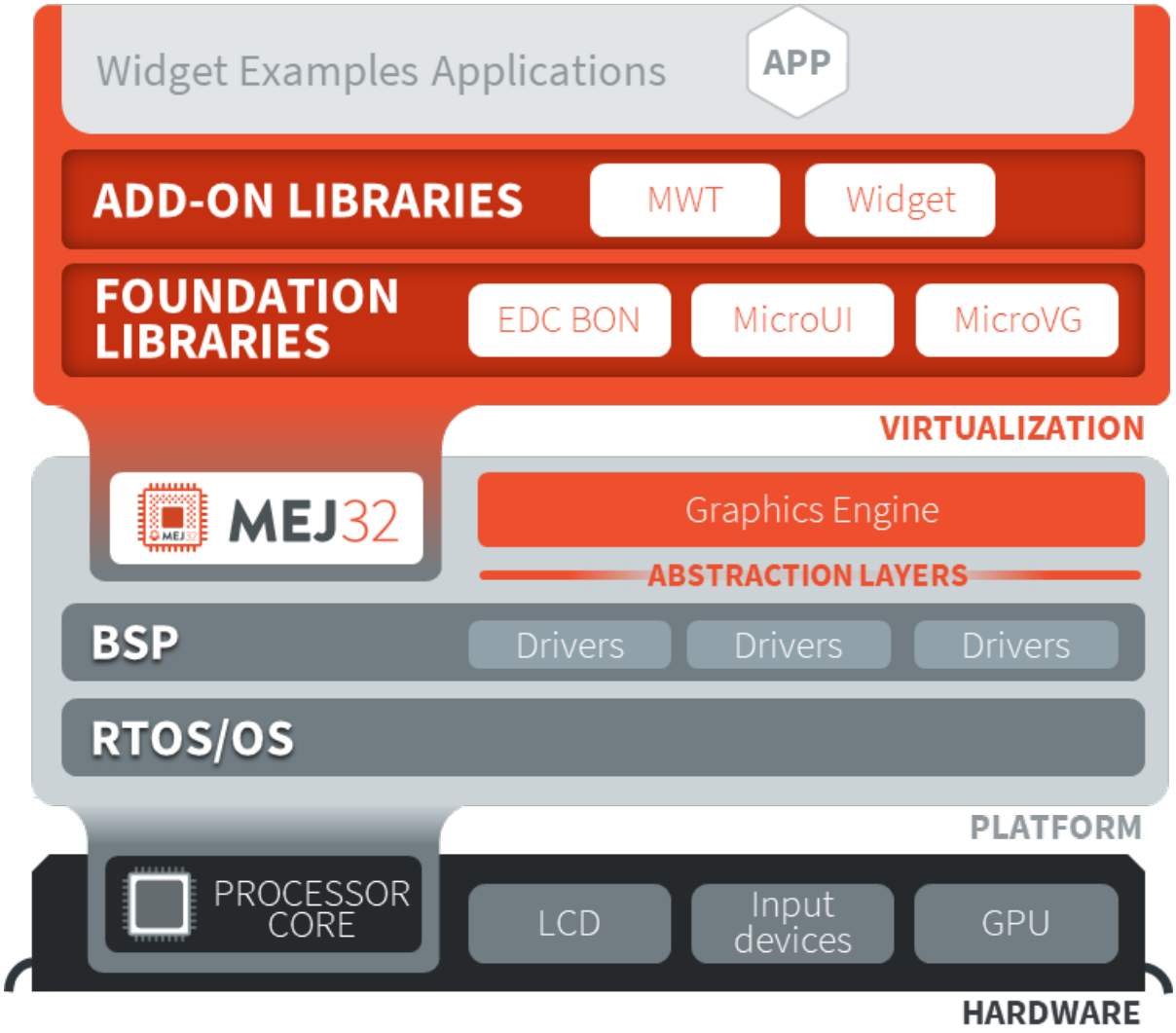


Fig. 132: Graphical User Interface Overview

Note: This chapter describes the current Graphical User Interface version 3 , provided by UI Pack version 14.0.0 or higher.

- If you are using the former Graphical User Interface version 3 provided by MicroEJ UI Pack version 13.x , please refer to this MicroEJ Documentation Archive.
- If you are using the former Graphical User Interface version 2 provided by MicroEJ UI Pack version up to 12.1.x , please refer to this MicroEJ Documentation Archive.

MicroUI

MicroUI Foundation Library provides access to a pixel-based display and inputs.

The aim of this library is to enable the creation of user interface in Java code by reifying hardware capabilities.

Usage

To use the MicroUI Foundation Library, add MicroUI API module to the project build file:

Gradle (build.gradle.kts)

MMM (module.ivy)

```
implementation("ej.api:microui:3.6.0")
```

```
<dependency org="ej.api" name="microui" rev="3.6.0"/>
```

Drawing Foundation Library extends MicroUI drawing APIs[1] with more complex ones such as:

- thick line, arc, circle and ellipse
- polygon
- image deformation and rotation

To use the Drawing Foundation Library, add Drawing API module to the project build file:

Gradle (build.gradle.kts)

MMM (module.ivy)

```
implementation("ej.api:drawing:1.0.4")
```

```
<dependency org="ej.api" name="drawing" rev="1.0.4"/>
```

Rendering

Clipping Region

The clipping region is a rectangle used to restrict the drawing operations. It is usually refered more simply as clip.

The clip can be set on any GraphicsContext with:

- GraphicsContext.setClip(int, int, int, int): sets the clipping area to the given rectangle.
- GraphicsContext.intersectClip(int, int, int, int): sets the clipping area to the intersection between the current clipping area and the given rectangle.
- GraphicsContext.resetClip(): resets the clipping area to the entire bounds of the graphics context.

The clip can be retrieved with:

- GraphicsContext.getClipX(): gets the x coordinate of the clipping area.
- GraphicsContext.getClipY(): gets the y coordinate of the clipping area.
- GraphicsContext.getClipWidth(): gets the width of the clipping area.

[1] These APIs were formerly included in MicroUI 2.x

- GraphicsContext.getClipHeight(): gets the height of the clipping area.

Translation Vector

It is possible to modify the origin of a GraphicsContext by setting a translation vector. This origin will be used by all drawing operations to translate the given coordinates. In other words, the translation x and y coordinates will be added to the x and y coordinates given to the subsequent drawing operations.

The translation coordinates can be negative.

The translation can be set on any GraphicsContext with:

- GraphicsContext.setTranslation(int, int): sets the translation coordinates.
- GraphicsContext.translate(int, int): adds the given coordinates to the current translation coordinates.
- GraphicsContext.resetTranslation(): resets the translation coordinates to (0,0).

The translation can be retrieved with:

- GraphicsContext.getTranslationX(): gets the x coordinate of the translation.
- GraphicsContext.getTranslationY(): gets the y coordinate of the translation.

Rendering Pipeline

The Rendering Pipeline of a MicroUI application consists of two main phases: Render, and Flush.

1. *Render*: During this phase, the application executes its rendering code to perform the necessary drawing operations. The application can either draw the whole screen for each frame or just a part.
2. *Flush*: This phase involves copying the back buffer to the front buffer. The VEE Port performs this operation, and it is the responsibility of the VEE Port developer to optimize this process, for example, by utilizing a GPU.

To reduce the number of pixels drawn, it may be interesting to redraw only subpart(s) of the display. In this case, the *Render* phase consists in setting the clip to the desired region and then draw inside this region. And again for each region to redraw.

Display Buffer Management

The display usually involves several buffers. It may be for one or several of these reasons:

- the display has its own memory that is not mapped by the CPU,
- to avoid seeing the steps of the drawing (if the display reads in the same memory as the application is drawing to),
- to allow the application to draw more quickly after a flush,
- etc.

More information can be found in the *Display* section of the VEE Porting Guide.

When the buffer used to draw is changed after a flush, the new buffer needs to be "refreshed" with the content of the previous buffer to avoid glitches. This copy is managed by the Buffer Refresh Strategy (BRS). To save some copy time, the BRS optimizes the refreshed region. For example, if the full screen is drawn, nothing needs to be restored (the same goes if the very same zone is drawn several times in a row). More information can be found in the *Buffer Refresh Strategy* section of the VEE Porting Guide.

The BRS has two ways to detect the drawn regions. Both are using the clip:

1. The first one does not require any action from the application, the BRS saves the clip when a drawing is done.
2. The second one consists in declaring the list of the regions that will be modified before starting the drawings. This is done by setting the clip and calling the GraphicsContext.notifyDrawingRegion() method for each region. Declaring the regions is advised when redrawing several parts of the display in the frames.

Since the refreshing is based on the clip, it means that all the pixels in the clip need to be drawn since they will not be refreshed. In other words, the pixels left unchanged inside the clip may contain unexpected data that will create glitches in the display.

To detect such pixels, *an option can be set* that draw a colored rectangle when a clip is detected by the BRS. If a pixel with that color is seen on the display, that means that it has not been drawn by the application.

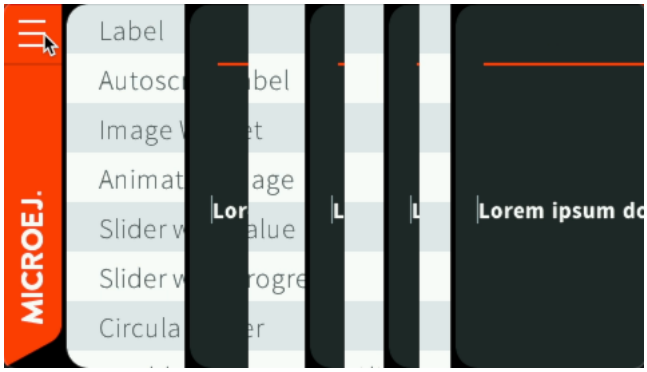


Fig. 133: Example of animation with clip not correctly set.

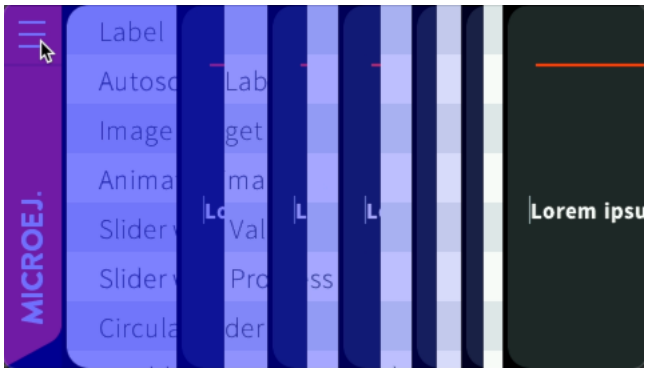


Fig. 134: Example of animation with clip not correctly set with highlight option (ej.fp.brs.dirtyColor=0x200000ff).

Drawing Logs

When performing drawing operations, the VEE port may report incidents that occurred during a drawing to the application. Graphics contexts enable this by holding flags that can be set by the VEE port and read by the application.

Usage Overview

When the VEE port needs to report an incident, it will set *drawing log flags* in the graphics context describing its nature. The application will then be able to read the flag values to know if an incident occurred. This mechanism is meant to help the developer to debug the application if it does not display what is expected. See *Drawing Logs* for more information on setting drawing log flags in the VEE port.

Incidents are split into two categories:

- *Warnings* are *non-critical* incidents that the application developer may ignore. When such an incident is reported, the flags are set in the graphics context so that the application can read them. However, if they are not explicitly read, the incident will be ignored silently.
- *Errors* are *critical* incidents that the application developer should not ignore when developing. As with warnings, *drawing log flags* will be set in the graphics context. Additionally, an exception will be thrown when the display is flushed so that the developer is aware of the incident.

Any incident may be either a warning or an error, depending on how the VEE port reported it. The distinction is made through the value of the flag DRAWING_LOG_ERROR.

Default Behavior

When the VEE port reports an incident, it sets drawing log flags in the graphics context. Additionally, if the incident was an error, it sets the special flag DRAWING_LOG_ERROR.

Every time the display is flushed, the flags contained in its graphics context will be checked. If the flag DRAWING_LOG_ERROR is set — which means an error has been reported — the flush function will throw a MicroUIException with the code DRAWING_ERROR, and the values of the drawing log flags in its message. Afterward, the flags will be reset.

> **Warning:** This behavior can be disabled at build time. In this case, the flags will keep their values after the display is flushed, and no exceptions will be thrown.
>
> Therefore, the developer should **not** rely on the drawing log flags in the application workflow. They are meant to be used as a **debugging hint**.

If an exception is thrown, the application developer should use the flag values to find the cause of the error and fix it accordingly.

Explicit Checks

MicroUI only checks the drawing log flags automatically during a display flush. The developer may want to read the flag values between drawing operations to investigate the cause of an error. Two functions are provided to do so:

- GraphicsContext.getAndClearDrawingLogFlags will return the current values of the flags and reset them.
- GraphicsContext.checkDrawingLogFlags behaves like GraphicsContext.getAndClearDrawingLogFlags. However, it will also throw an exception if the flag DRAWING_LOG_ERROR is set, like it is done when the display is flushed.

For example, if a VEE port with no implementation to draw circles reports an error with the flag DRAWING_LOG_NOT_IMPLEMENTED, the application would behave as below.

```
// The VEE port has not implemented this function.
Painter.drawCircle(gc, 1, 2, 3);

// This throws a MicroUIException with the error code -13 (DRAWING_ERROR).
Display.getDisplay().flush();
```

The application developer could force a check of the drawing log flags:

```
// The VEE port has not implemented this function.
Painter.drawCircle(gc, 1, 2, 3);

// This throws a MicroUIException with the error code -13 (DRAWING_ERROR).
int flags = gc.checkDrawingLogFlags();
```

Or the developer could explicitly retrieve the value of the flags:

```
// The VEE port has not implemented this function.
Painter.drawCircle(gc, 1, 2, 3);

// This retrieves the values of drawing log flags.
int flags = gc.getAndClearDrawingLogFlags();
// This prints "80000001" (DRAWING_LOG_ERROR | DRAWING_LOG_NOT_IMPLEMENTED == 1 <<␣
↪31 | 1 << 0).
System.out.println(Integer.toHexString(flags));
```

Configuration

When releasing an application, the developer should disable the automatic check of drawing log flags performed when the display is flushed. Doing so will prevent exceptions from being thrown, which would cause an unexpected crash. It will also not clear the drawing log flags when the display is flushed.

Disabling this check can be done by setting the *constant* com.microej.library.microui.impl.check-drawing-errors-on-flush to false when building the application. If it is not set, it defaults to true .

Available Constants

MicroUI provides a set of constants to describe reported incidents. They are defined and documented in the class GraphicsContext .

Constant	Value	Description
DRAWING_LOG_NOT_IMPLEMENTED	1 << 0	This function is not implemented.
DRAWING_LOG_FORBIDDEN	1 << 1	This function must not be called in this situation.
DRAWING_LOG_OUT_OF_MEMORY	1 << 2	The system ran out of memory.
DRAWING_LOG_CLIP_MODIFIED	1 << 3	The VEE port modified clip values in the graphics context.
DRAWING_LOG_MISSING_CHARACTER	1 << 4	An undefined character was drawn.
DRAWING_LOG_LIBRARY_INCIDENT	1 << 29	An incident occurred in an underlying library.
DRAWING_LOG_UNKNOWN_INCIDENT	1 << 30	An incident that does not match other flags occurred.
DRAWING_LOG_ERROR	1 << 31	Special flag denoting critical incidents.

The special value DRAWING_SUCCESS (defined as 0) represents a state where no drawing log flags are set, so encountering this value means no incident was reported.

New flag constants may be added in future versions of MicroUI. Also, their actual values may change, and the developer should not rely on them.

Images

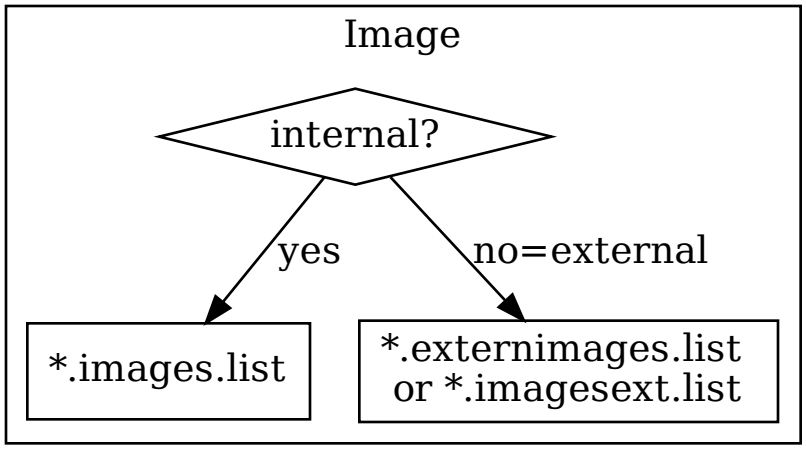
Immutable Images

Overview

Immutable images are graphical resources that can be accessed with a call to ej.microui.display.Image.getImage() or ej.microui.display.ResourceImage.loadImage(). As their name suggests, immutable images can not be modified. Therefore, there is no way to get a Graphics Context to draw into these images. To be displayed, these images have to be converted from their source format to a RAW format. The conversion can either be done:

- At build-time, using the Image Generator.
- At run-time, using the relevant decoder library.

Immutable images are declared in *Classpath* *.images.list files (**or** in *.externimages.list for an external resource, see *External Images*).



The file format is a standard Java properties file. Each line contains a `/` -separated resource path relative to the Classpath root referring to a standard image file (e.g. `.png`, `.jpg`). The resource may be followed by an optional parameter (separated by a `:`) which defines and/or describes the image output file format (RAW format). When no option is specified, the image is embedded as-is and will be decoded at run-time. Example:

```
# The following image is embedded as
# a PNG resource (decoded at run-time)
com/mycompany/MyImage1.png

# The following image is embedded as
# a 16-bit encoding without transparency (decoded at build-time)
com/mycompany/MyImage2.png:RGB565

# The following image is embedded as
# a 16-bit encoding with transparency (decoded at build-time)
com/mycompany/MyImage3.png:ARGB1555
```

Configuration File

Here is the format of the `*.images.list` files.

```
ConfigFile      ::= Line [ 'EOL' Line ]*
Line            ::= ImagePath [ ':' ImageOption ]*
ImagePath       ::= Identifier [ '/' Identifier ]*
ImageOption     ::= [^:]*
Identifier      ::= Letter [ LetterOrDigit ]*
Letter          ::= 'a-zA-Z_$'
LetterOrDigit   ::= 'a-zA-Z_$0-9'
```

Unspecified Output Format

When no output format is set in the image list file, the image is embedded without any conversion / compression. This allows you to embed the resource as-is, in order to keep the source image characteristics (compression, bpp, size, etc.). This option produces the same result as specifying an image as a resource in the MicroEJ launcher (i.e. in a `.resources.list` file).

Refer to the platform specification to retrieve the list of runtime decoders.

Advantages

- Preserves the image characteristics.
- Preserves the original image compression.

Disadvantages

- Requires an image runtime decoder.
- Requires some RAM in which to store the decoded image.
- Requires execution time to decode the image.

```
image1
```

Display Output Format

It encodes the image into the exact display memory representation. If the image to encode contains some transparent pixels, the output file will embed the transparency according to the display's implementation capacity. When all pixels are fully opaque, no extra information will be stored in the output file in order to free up some memory space.

Note: When the display memory representation is standard, the display output format is automatically replaced by a standard format.

Advantages

- Drawing an image is very fast because no pixel conversion is required at runtime.
- Supports alpha encoding when the display pixel format allows it.

Disadvantages

- No compression: the image size in bytes is proportional to the number of pixels.

```
image1:display
```

Standard Output Formats

Some image formats are well known and commonly implemented by GPUs.

Refer to the platform specification to retrieve the list of natively supported formats.

Advantages

- The pixel layout and bit format are standard, so it is easy to manipulate these images on the C-side.
- Drawing an image is very fast when the display driver recognizes the format (with or without transparency).

Disadvantages

- No compression: the image size in bytes is proportional to the number of pixels.
- Slower than `display` format when the display driver does not recognize the format: a pixel conversion is required at runtime.

Here is the list of the standard formats:

- Transparent images:
  - ARGB8888: 32-bit format, 8 bits for transparency, 8 per color,
  - ARGB4444: 16-bit format, 4 bits for transparency, 4 per color,
  - ARGB1555: 16-bit format, 1 bit for transparency, 5 per color.
- Transparent images with premultiplied alpha (RGB and alpha are linked)
  - ARGB8888_PRE: 32-bit format, 8 bits for transparency, 8 per color,
  - ARGB4444_PRE: 16-bit format, 4 bits for transparency, 4 per color,
  - ARGB1555_PRE: 16-bit format, 1 bit for transparency, 5 per color.
- Opaque images:
  - RGB888: 24-bit format, 8 per color,
  - RGB565: 16-bit format, 5 for red, 6 for green, 5 for blue.
- Alpha images, only transparency is encoded (the color applied when drawing the image is the current GraphicsContext color):
  - A8: 8-bit format,
  - A4: 4-bit format,
  - A2: 2-bit format,
  - A1: 1-bit format.

Examples:

```
image1:ARGB8888
image2:RGB565
image3:A4
```

Grayscale Output Formats

Some grayscale formats may be useful on grayscale or black and white displays.

Advantages

- Reduced footprint with less bits per pixels.

Disadvantages

- No compression: the image size in bytes is proportional to the number of pixels.
- Slower: a pixel conversion is required at runtime.

Here is the list of the grayscale formats:

- With transparency:
  - AC44: 4 bits for transparency, 4 bits with grayscale conversion,
  - AC22: 2 bits for transparency, 2 bits with grayscale conversion,
  - AC11: 1 bit for transparency, 1 bit with grayscale conversion.
- Without transparency:
  - C4: 4 bits with grayscale conversion,
  - C2: 2 bits with grayscale conversion,
  - C1: 1 bit with grayscale conversion.

Examples:

```
image1:AC44
image2:C2
```

Compressed Output Formats

Some image formats are compressed using run-length encoding. This compression is lossless. The principle is that identical consecutive pixels are stored as one entry (value and count). The more the consecutive pixels are identical, the more the compression is efficient.

Advantages

- Good compression when there are a lot of identical consecutive pixels.

Disadvantages

- Drawing an image may be slightly slower than using an uncompressed format supported by the GPU.
- Not designed for images with many different pixel colors: in such case, the output file size may be larger than the original image file.

Here is the list of the compressed formats:

- ARGB1565_RLE: 1 bit for transparency, 5 for red, 6 for green, 5 for blue. Colors are encoded as RGB565 and the transparency bit is encoded as part of the RLE block format (Formerly named RLE1 up to UI Pack 13.3.X.)
- A8_RLE: similar to A8.

```
image1:ARGB1565_RLE
image2:RLE1 # Deprecated
image3:A8_RLE
```

Alpha Format

As described above, the formats A8 , A4 , A2 , A1 and A8_RLE (also called *Picto*) only handle the opacity information. The source image can be transparent or not, colored or grayscaled. The alpha format provides two options to interpret the source image's pixels:

- grayscale : The source image is first grayscaled and then drawn over a white background. The black pixels are encoded as fully opaque pixels, the white pixels as fully transparent pixels and gray pixels as transparent pixels (the closer the pixel is to black, the more opaque the encoded opacity is).
- alpha : Only the opacity component is encoded (the R-G-B components are ignored).
- *no option*: same as grayscale (backward compatibility).

```
image1:A8:grayscale
image2:A8_RLE:alpha
image3:A4
```

Note: The MicroUI ResourceImage OutputFormat A8 encodes in the same way as the option alpha .

Expected Result

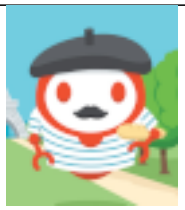
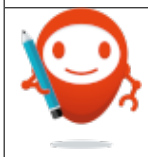
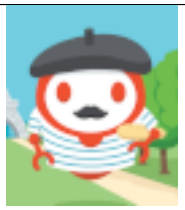
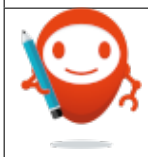
The following table summarizes the usage of the different formats and the actual result on a white background.

Table 15: Image Output Formats Usage

Format	Source	Result
ARGB8888		
ARGB4444		
ARGB1555		
ARGB8888_PRE		
ARGB4444_PRE		

continues on next page

Table 15 – continued from previous page

Format	Source	Result
ARGB1555_PRE		
RGB888		
		
RGB565		
		
A8		
	With 0x0000ff as color and option grayscale	
	With 0x0000ff as color and option alpha	

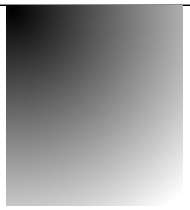
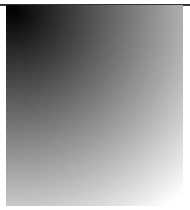
continues on next page

Table 15 – continued from previous page

Format	Source	Result
	With 0x0000ff as color and option grayscale	
	With 0x0000ff as color and option alpha	
A4	option grayscale	
	option alpha	
A2	option grayscale	
	option alpha	
A1	option grayscale	
	option alpha	
C4		

continues on next page

Table 15 – continued from previous page

Format	Source	Result
C2		
C1		
AC44		
AC22		
AC11		
ARGB1565_RLE		
A8_RLE	option grayscale option alpha	

Usage Advice

- When the image is rarely used, or when there is little Flash and enough RAM: embed the image in its original compressed format (PNG or JPG for instance).
- For an opaque image: *RGB565* is usually sufficient.
- For a transparent image: *ARGB4444* is usually sufficient.
- For a transparent image that contains only shape(s) with horizontal or vertical edges:
  - *ARGB1555* may be interesting to have more colors,
  - for a smaller footprint if the image matches the RLE rule, *ARGB1565_RLE* is best.
- For a pictogram to colorize:
  - *A4* is usually sufficient,
  - *A8* may be necessary for pictograms with long gradients,
  - for a smaller footprint if the image matches the RLE rule, *A8_RLE* is best.
- For BSP with a GPU, choose a format compatible with the GPU (all formats may not be available),
  - *ARGB* formats: choose between non-premultiplied formats and premultiplied formats (suffixed with *_PRE*),
  - *Ax* formats (pictogram): all bits-per-pixel values may not be available.
  - be careful about the color components position (*A-R-G-B* versus *R-G-B-A* for instance),
  - avoid formats *Cx*, *ACxx* and *xxx_RLE*, which are not compatible with a GPU.

Caching Generated Images

Images converted using the Image Generator can be cached so that they are not rebuilt every time the application is launched. Doing so can significantly speed up the application build phase.

The cache is enabled by default. It may be disabled by setting the *Application option* `ej.microui.imageConverter.disableCache` to `true`.

The Image Generator obeys several rules when choosing whether an image should be converted.

- If the cache is disabled, all images are generated every time the application is launched.
- All images will be regenerated if the application is launched using another VEE port and the new VEE port uses a different Image Generator or another set of Image Generator plugins.
- If the generated image does not exist, it will be generated.
- If the source image has been modified since the last time it was converted, the image will be regenerated.
- The image will be regenerated if the destination format has been modified in the *images.list* file.

Cached images are stored in `.cache/images`, which is located in the *application output folder*. You may delete this directory to force the generation of all images in your application. An image that was previously generated but is no longer listed in the `*.images.list` files when the application is launched will be deleted from the cache directory.

> **Warning:** When *testing an Image Generator extension project*, the image cache is automatically disabled.

External Images

To fetch immutable images from external memory, the application must pre-register the *external Image resources*. The management of this kind of image may be different than the internal images and may require some allocations in the *Images Heap*. For more details about the external image management, refers to the VEE Port Guide chapter *External Resource*.

Image Generator Error Messages

These errors can occur while preprocessing images.

Table 16: Static Image Generator Error Messages

ID	Type	Description
0	Error	The image generator has encountered an unexpected internal error.
1	Error	The images list file has not been specified.
2	Error	The image generator cannot create the final, raw file.
3	Error	The image generator cannot read the images list file. Make sure the system allows reading of this file.
4	Warning	The image generator has found no image to generate.
5	Error	The image generator cannot load the images list file.
6	Warning	The specified image path is invalid: The image will be not converted.
7	Warning	There are too many or too few options for the desired format.
8	Error	The display format is not generic; a MicroUIRawImageGeneratorExtension implementation is required to generate the MicroUI raw image.
9	Error	The image cannot be read.
10	Error	The image generator has encountered an unexpected internal error (invalid endianness).
11	Error	The image generator has encountered an unexpected internal error (invalid bpp).
12	Error	The image generator has encountered an unexpected internal error (invalid display format).
13	Error	The image generator has encountered an unexpected internal error (invalid pixel layout).
14	Error	The image generator has encountered an unexpected internal error (invalid output folder).
15	Error	The image generator has encountered an unexpected internal error (invalid memory alignment).
16	Error	The input image format and / or the ouput format are not managed by the image generator.
17	Error	The image has been already loaded with another output format.

Mutable Images

Overview

Unlike immutable images, mutable images are graphical resources that can be created and modified at runtime. The application can draw into such images using the Painter classes with the image's Graphics Context as the destination. Mutable images can be created with a call to constructor ej.microui.display.BufferedImage().

```
BufferedImage image = new BufferedImage(320,  240);
GraphicsContext g = image.getGraphicsContext();
g.setColor(Colors.BLACK);
Painter.fillRectangle(g, 0, 0, 320, 240);
g.setColor(Colors.RED);
Painter.drawHorizontalLine(g, 50, 50, 100);
image.close();
```

Display Format

By default, the output format of a BufferedImage matches the display's pixel organization (layout, depth, etc.). The algorithms used to draw in such an image are the same as those used on the display (for footprint purposes). The algorithm cannot draw transparent pixels since the display back buffer is opaque.

In addition, GraphicsContext.setColor() does not consider the alpha channel and only accepts RGB values. The given color value is interpreted as a 24-bit RGB color, where the high-order byte is ignored, and the remaining bytes contain the red, green, and blue channels, respectively.

Other Formats

It is also possible to create a buffered image with another format using the constructor with the format parameter.

The other formats than the display one are not supported by MicroUI. But a VEE port can manage one or more formats (see *Destination Format*).

Depending on the format, the transparency may be supported.

Images Heap

The image heap is used to allocate the pixel data of:

- Mutable images (i.e. BufferedImage instances).
- Immutable images decoded at runtime, typically a PNG: the heap is used to store the decoded image **and** the runtime decoder's temporary buffers, required during the decoding step. After the decoding step, all the temporary buffers are freed. Note that the size of the temporary buffers depends on the decoder **and** on the original image itself (compression level, pixel encoding, etc.).
- Immutable images which are not byte-addressable, such as images opened with an input stream (i.e. ResourceImage instances).
- Immutable images which are byte-addressable but converted to a different output format (i.e. ResourceImage instances).

In other words, every image which cannot be retrieved using ej.microui.display.Image.getImage() is saved on the image heap.

The size of the images heap can be configured with the `ej.microui.memory.imagesheap.size` property.

> **Warning:** A ResourceImage allocated on the images heap must be closed manually by the application (ResourceImage.close()); otherwise, a memory leak will occur on the images heap.

For more details about the images heap implementation, refers to *this chapter* in the VEE Port Guide.

Fonts

Overview

The font system consists in two distinct parts: the built-in part (also known as the *internal font*) and, since MicroUI 3.6, the extended part. The extended part allows the VEE Port to provide one or more additional font systems with their own characteristics. However, once created, all fonts can be used by the application using the Font class. In most cases, the application does not know the type of font and should use all fonts (built-in or extended) in the same way. This makes for portable code (as far as rendering is concerned), as only the code that creates the font at runtime is specific.

Note: The application may need to manipulate certain specific characteristics of an extended font for rendering (text layout, opacity, etc.), in which case it can use the extended font API.

Internal Font

Principle

The internal font format is a simple and small bitmap format.

The built-in font engine used to render this format:

- Does not require any additional support in the VEE Port to be used as-is.
- Has the same rendering whatever the VEE Port capabilities.
- Has a very small memory footprint.
- Is fast.
- Does not need runtime memory allocation.
- Provides some offline tools to generate the font files.
- Allows to tune the footprint of the font files (pixel opacity levels and ranges).
- Can be extended to provide additional features (such as a complex layout manager).

Its limitations are described in the following chapters.

Height

Each font file is encoded for a given font height. To use the same font face (`.ttf` file) with several heights, several font files are required. Consequently, an application that uses a lot of font heights may be penalized (ROM footprint).

Color

The font encoding is similar to the *Alpha Format*. These formats only embed the pixel's opacity information. The strings will be drawn using the current color of the graphics context. No color is embedded, consequently, colored characters as emojis are not supported.

Languages

The language support is limited to the Unicode basic multilingual alphabets, whose characters are encoded on 16 bits, i.e. Unicodes characters ranging from 0x0000 to 0xFFFF (Surrogates characters are allowed). It allows to render left-to-right or right-to-left writing systems: Latin (English, etc.), Arabic, Chinese, etc. are some supported languages. However, the rendering is always performed left-to-right, even if the string is written right-to-left. There is no built-in support for:

- Top-to-bottom writing systems.
- Diacritics, contextual letters, specific character ordering, etc.

The array of characters to render must only contain renderable characters (no escape character) and the very first character in the array is always the character on the left.

Hint: Use the offline tool *Native Language Support* to automatically convert the translation messages in a character array compatible with the built-in font engine.

FNT Font File

Font files which end with the suffix `.fnt` are bitmap fonts specified by AngelCode (without the support of the kerning).

A third-party tool is required to generate a `.fnt` from `.ttf`. To be compatible with the MicroEJ Font Engine, the following settings are mandatory:

- The output file format must be a `.txt` (not XML or binary).
- Images must be PNG files.
- Font and images must be located in the same folder.
- The images files name must end with the file number (0-based): `xxx0.png`, `xxx1.png`, etc.
- The foreground color must be black and the background color white or transparent.

Hint: Open the `.fnt` with a text editor to retrieve the image: `page id=0 file="myfont_0.png"`.

---

fontbm

fontbm is a free cross-platform (Linux / MacOS / Windows) command line bitmap font generator. It is based on FreeType2 and generates exactly the same font on any operating system.

The next command line generates a .fnt from the font file SourceSansPro-Regular.ttf with a size of 24 pixels:

```
./fontbm --font-file SourceSansPro-Regular.ttf --output myfont --color 0,0,0 --font-size 24
```

It generates a .fnt accompanied by its images (one or more):

```
myfont.fnt
myfont_0.png
myfont_1.png
```

bmfont

bmfont is a free Windows UI and command line bitmap font generator, based on FreeType2.

The options to export the font must follow these rules:

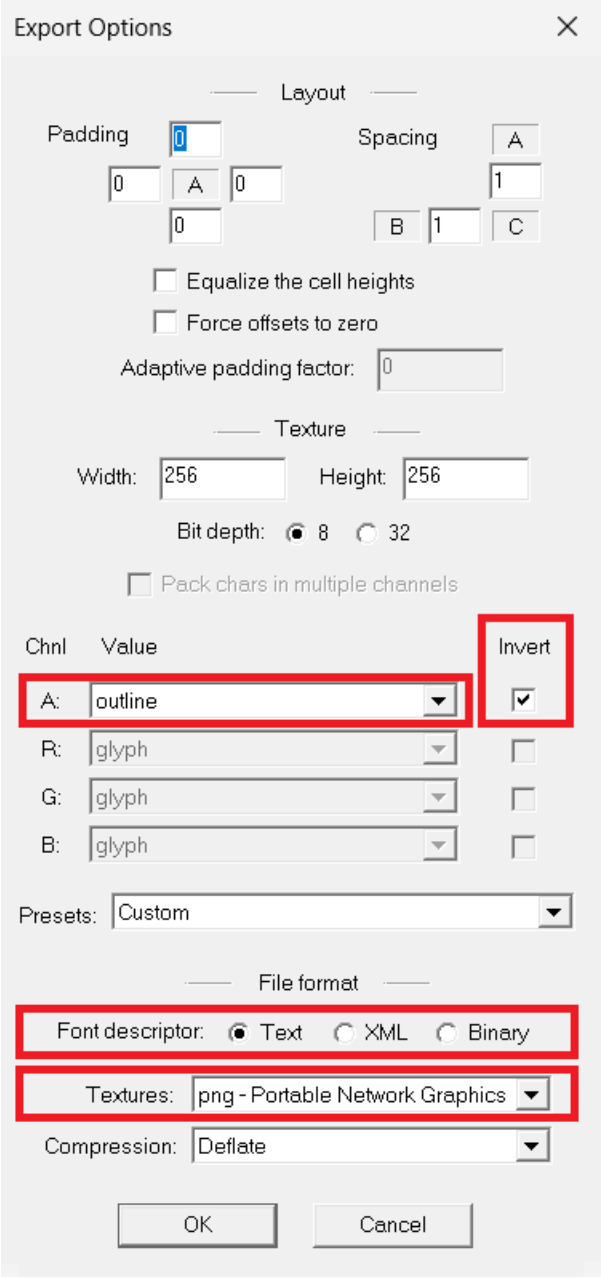


Fig. 135: BMFont Export Options

It generates a .fnt accompanied by its images (one or more):

```
myfont.fnt
myfont_0.png
myfont_1.png
```

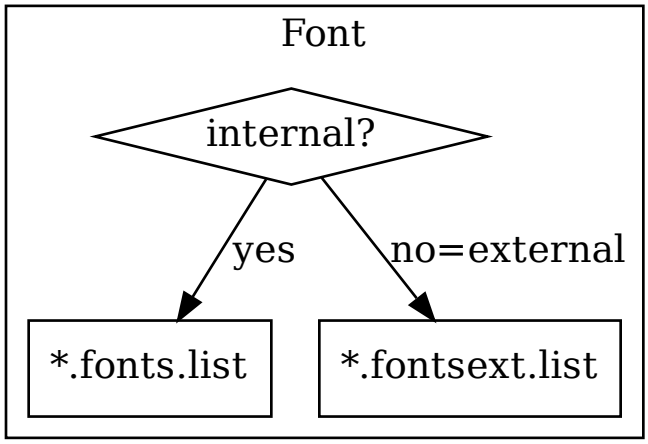
EJF Font File

Font files which end with the `.ejf` suffix are created using the Font Designer (see *Font Designer*).

Usage

A Java application can retrieved a font with a call to Font.getFont() passing its fully qualified name.

Fonts are declared in *Classpath* `*.fonts.list` files (**or** in `*.fontsext.list` for an external resource, see *External Fonts*).



The file format is a standard Java properties file, each line representing a `/` separated resource path relative to the Classpath root referring to a Font file (`fnt` or `.ejf` extension). The resource may be followed by optional parameters which define :

- some ranges of characters to embed in the final raw file;
- the required pixel depth for transparency.

By default, all characters available in the input font file are embedded, and the pixel depth is `1` (i.e 1 bit-per-pixel). Example:

```
# The following font is embedded with all characters
# without transparency
com/mycompany/MyFont1.fnt

# The following font is embedded with only the latin
# unicode range without transparency
com/mycompany/MyFont2.fnt:latin

# The following font is embedded with all characters
# with 2 levels of transparency
com/mycompany/MyFont2.ejf::2
```

Configuration File

Here is the format of the *.fonts.list files.

```
ConfigFile          ::= Line [ 'EOL' Line ]*
Line                ::= FontPath [ ':' [ Ranges ] [ ':' BitsPerPixel ] ]
FontPath            ::= Identifier [ '/' Identifier ]*
Ranges              ::= Range [ ';' Range ]*
Range               ::= CustomRangeList | KnownRange
CustomRangeList     ::= CustomRange [ ',' CustomRange ]*
CustomRange         ::= Number | Number '-' Number
KnownRange          ::= Name [ SubRangeList ]?
SubRangeList        ::= '(' SubRange [ ',' SubRange ]* ')'
SubRange            ::= Number | Number - Number
Identifier          ::= 'a-zA-Z_$' [ 'a-zA-Z_$0-9' ]*
Number              ::= Number16 | Number10
Number16            ::= '0x' [ Digit16 ]+
Number10            ::= [ Digit10 ]+
Digit16             ::= 'a-fA-F0-9'
Digit10             ::= '0-9'
BitsPerPixel        ::= '1' | '2' | '4' | '8'
```

Font Range

The first parameter is for specifying the font ranges to embed. Selecting only a specific set of characters to embed reduces the memory footprint. If unspecified, all characters of the font are embedded.

Several ranges can be specified, separated by ; . There are two ways to specify a character range: the custom range and the known range.

Custom Range

Allows the selection of raw Unicode character ranges.

Examples:

- myfont:0x21-0x49 : Defines one range: embed all characters from 0x21 to 0x49 (included);
- myfont:0x21-0x49,0x55-0x75 : Defines a set of two ranges: embed all characters from 0x21 to 0x49 and from 0x55 to 0x75.
- myfont:0x21-0x49,0x55 : Defines a set of one range and one character: embed all characters from 0x21 to 0x49 and character 0x55.

Known Range

A known range is a range available in the following table.

Examples:

- myfont:basic_latin : Embed all *Basic Latin* characters;
- myfont:basic_latin;arabic : Embed all *Basic Latin* characters, and all *Arabic* characters.

The following table describes the available list of ranges and sub-ranges (processed from the “Unicode Character Database” version 9.0.0 available on the official unicode website https://home.unicode.org/).

Table 17: Ranges

Name	Tag	Start	End
Basic Latin	basic_latin	0x0	0x7f
Latin-1 Supplement	latin-1_supplement	0x80	0xff
Latin Extended-A	latin_extended-a	0x100	0x17f
Latin Extended-B	latin_extended-b	0x180	0x24f
IPA Extensions	ipa_extensions	0x250	0x2af
Spacing Modifier Letters	spacing_modifier_letters	0x2b0	0x2ff
Combining Diacritical Marks	combining_diacritical_marks	0x300	0x36f
Greek and Coptic	greek_and_coptic	0x370	0x3ff
Cyrillic	cyrillic	0x400	0x4ff
Cyrillic Supplement	cyrillic_supplement	0x500	0x52f
Armenian	armenian	0x530	0x58f
Hebrew	hebrew	0x590	0x5ff
Arabic	arabic	0x600	0x6ff
Syriac	syriac	0x700	0x74f
Arabic Supplement	arabic_supplement	0x750	0x77f
Thaana	thaana	0x780	0x7bf
NKo	nko	0x7c0	0x7ff
Samaritan	samaritan	0x800	0x83f
Mandaic	mandaic	0x840	0x85f
Arabic Extended-A	arabic_extended-a	0x8a0	0x8ff
Devanagari	devanagari	0x900	0x97f
Bengali	bengali	0x980	0x9ff
Gurmukhi	gurmukhi	0xa00	0xa7f
Gujarati	gujarati	0xa80	0xaff
Oriya	oriya	0xb00	0xb7f
Tamil	tamil	0xb80	0xbff
Telugu	telugu	0xc00	0xc7f
Kannada	kannada	0xc80	0xcff
Malayalam	malayalam	0xd00	0xd7f
Sinhala	sinhala	0xd80	0xdff
Thai	thai	0xe00	0xe7f
Lao	lao	0xe80	0xeff
Tibetan	tibetan	0xf00	0xfff
Myanmar	myanmar	0x1000	0x109f
Georgian	georgian	0x10a0	0x10ff
Hangul Jamo	hangul_jamo	0x1100	0x11ff
Ethiopic	ethiopic	0x1200	0x137f
Ethiopic Supplement	ethiopic_supplement	0x1380	0x139f
Cherokee	cherokee	0x13a0	0x13ff
Unified Canadian Aboriginal Syllabics	unified_canadian_aboriginal_syllabics	0x1400	0x167f
Ogham	ogham	0x1680	0x169f
Runic	runic	0x16a0	0x16ff
Tagalog	tagalog	0x1700	0x171f
Hanunoo	hanunoo	0x1720	0x173f
Buhid	buhid	0x1740	0x175f
Tagbanwa	tagbanwa	0x1760	0x177f
Khmer	khmer	0x1780	0x17ff
Mongolian	mongolian	0x1800	0x18af

continues on next page

Table 17 – continued from previous page

Name	Tag	Start	End
Unified Canadian Aboriginal Syllabics Extended	unified_canadian_aboriginal_syllabics_extended	0x18b0	0x18ff
Limbu	limbu	0x1900	0x194f
Tai Le	tai_le	0x1950	0x197f
New Tai Lue	new_tai_lue	0x1980	0x19df
Khmer Symbols	khmer_symbols	0x19e0	0x19ff
Buginese	buginese	0x1a00	0x1a1f
Tai Tham	tai_tham	0x1a20	0x1aaf
Combining Diacritical Marks Extended	combining_diacritical_marks_extended	0x1ab0	0x1aff
Balinese	balinese	0x1b00	0x1b7f
Sundanese	sundanese	0x1b80	0x1bbf
Batak	batak	0x1bc0	0x1bff
Lepcha	lepcha	0x1c00	0x1c4f
Ol Chiki	ol_chiki	0x1c50	0x1c7f
Cyrillic Extended-C	cyrillic_extended-c	0x1c80	0x1c8f
Sundanese Supplement	sundanese_supplement	0x1cc0	0x1ccf
Vedic Extensions	vedic_extensions	0x1cd0	0x1cff
Phonetic Extensions	phonetic_extensions	0x1d00	0x1d7f
Phonetic Extensions Supplement	phonetic_extensions_supplement	0x1d80	0x1dbf
Combining Diacritical Marks Supplement	combining_diacritical_marks_supplement	0x1dc0	0x1dff
Latin Extended Additional	latin_extended_additional	0x1e00	0x1eff
Greek Extended	greek_extended	0x1f00	0x1fff
General Punctuation	general_punctuation	0x2000	0x206f
Superscripts and Subscripts	superscripts_and_subscripts	0x2070	0x209f
Currency Symbols	currency_symbols	0x20a0	0x20cf
Combining Diacritical Marks for Symbols	combining_diacritical_marks_for_symbols	0x20d0	0x20ff
Letterlike Symbols	letterlike_symbols	0x2100	0x214f
Number Forms	number_forms	0x2150	0x218f
Arrows	arrows	0x2190	0x21ff
Mathematical Operators	mathematical_operators	0x2200	0x22ff
Miscellaneous Technical	miscellaneous_technical	0x2300	0x23ff
Control Pictures	control_pictures	0x2400	0x243f
Optical Character Recognition	optical_character_recognition	0x2440	0x245f
Enclosed Alphanumerics	enclosed_alphanumerics	0x2460	0x24ff
Box Drawing	box_drawing	0x2500	0x257f
Block Elements	block_elements	0x2580	0x259f
Geometric Shapes	geometric_shapes	0x25a0	0x25ff
Miscellaneous Symbols	miscellaneous_symbols	0x2600	0x26ff
Dingbats	dingbats	0x2700	0x27bf
Miscellaneous Mathematical Symbols-A	miscellaneous_mathematical_symbols-a	0x27c0	0x27ef
Supplemental Arrows-A	supplemental_arrows-a	0x27f0	0x27ff
Braille Patterns	braille_patterns	0x2800	0x28ff
Supplemental Arrows-B	supplemental_arrows-b	0x2900	0x297f
Miscellaneous Mathematical Symbols-B	miscellaneous_mathematical_symbols-b	0x2980	0x29ff

continues on next page

Table 17 – continued from previous page

Name	Tag	Start	End
Supplemental Mathematical Operators	supplemental_mathematical_operators	0x2a00	0x2aff
Miscellaneous Symbols and Arrows	miscellaneous_symbols_and_arrows	0x2b00	0x2bff
Glagolitic	glagolitic	0x2c00	0x2c5f
Latin Extended-C	latin_extended-c	0x2c60	0x2c7f
Coptic	coptic	0x2c80	0x2cff
Georgian Supplement	georgian_supplement	0x2d00	0x2d2f
Tifinagh	tifinagh	0x2d30	0x2d7f
Ethiopic Extended	ethiopic_extended	0x2d80	0x2ddf
Cyrillic Extended-A	cyrillic_extended-a	0x2de0	0x2dff
Supplemental Punctuation	supplemental_punctuation	0x2e00	0x2e7f
CJK Radicals Supplement	cjk_radicals_supplement	0x2e80	0x2eff
Kangxi Radicals	kangxi_radicals	0x2f00	0x2fdf
Ideographic Description Characters	ideographic_description_characters	0x2ff0	0x2fff
CJK Symbols and Punctuation	cjk_symbols_and_punctuation	0x3000	0x303f
Hiragana	hiragana	0x3040	0x309f
Katakana	katakana	0x30a0	0x30ff
Bopomofo	bopomofo	0x3100	0x312f
Hangul Compatibility Jamo	hangul_compatibility_jamo	0x3130	0x318f
Kanbun	kanbun	0x3190	0x319f
Bopomofo Extended	bopomofo_extended	0x31a0	0x31bf
CJK Strokes	cjk_strokes	0x31c0	0x31ef
Katakana Phonetic Extensions	katakana_phonetic_extensions	0x31f0	0x31ff
Enclosed CJK Letters and Months	enclosed_cjk_letters_and_months	0x3200	0x32ff
CJK Compatibility	cjk_compatibility	0x3300	0x33ff
CJK Unified Ideographs Extension A	cjk_unified_ideographs_extension_a	0x3400	0x4dbf
Yijing Hexagram Symbols	yijing_hexagram_symbols	0x4dc0	0x4dff
CJK Unified Ideographs	cjk_unified_ideographs	0x4e00	0x9fff
Yi Syllables	yi_syllables	0xa000	0xa48f
Yi Radicals	yi_radicals	0xa490	0xa4cf
Lisu	lisu	0xa4d0	0xa4ff
Vai	vai	0xa500	0xa63f
Cyrillic Extended-B	cyrillic_extended-b	0xa640	0xa69f
Bamum	bamum	0xa6a0	0xa6ff
Modifier Tone Letters	modifier_tone_letters	0xa700	0xa71f
Latin Extended-D	latin_extended-d	0xa720	0xa7ff
Syloti Nagri	syloti_nagri	0xa800	0xa82f
Common Indic Number Forms	common_indic_number_forms	0xa830	0xa83f
Phags-pa	phags-pa	0xa840	0xa87f
Saurashtra	saurashtra	0xa880	0xa8df
Devanagari Extended	devanagari_extended	0xa8e0	0xa8ff
Kayah Li	kayah_li	0xa900	0xa92f
Rejang	rejang	0xa930	0xa95f
Hangul Jamo Extended-A	hangul_jamo_extended-a	0xa960	0xa97f
Javanese	javanese	0xa980	0xa9df
Myanmar Extended-B	myanmar_extended-b	0xa9e0	0xa9ff
Cham	cham	0xaa00	0xaa5f
Myanmar Extended-A	myanmar_extended-a	0xaa60	0xaa7f
Tai Viet	tai_viet	0xaa80	0xaadf

continues on next page

Table 17 – continued from previous page

Name	Tag	Start	End
Meetei Mayek Extensions	meetei_mayek_extensions	0xaae0	0xaaff
Ethiopic Extended-A	ethiopic_extended-a	0xab00	0xab2f
Latin Extended-E	latin_extended-e	0xab30	0xab6f
Cherokee Supplement	cherokee_supplement	0xab70	0xabbf
Meetei Mayek	meetei_mayek	0xabc0	0xabff
Hangul Syllables	hangul_syllables	0xac00	0xd7af
Hangul Jamo Extended-B	hangul_jamo_extended-b	0xd7b0	0xd7ff
High Surrogates	high_surrogates	0xd800	0xdb7f
High Private Use Surrogates	high_private_use_surrogates	0xdb80	0xdbff
Low Surrogates	low_surrogates	0xdc00	0xdfff
Private Use Area	private_use_area	0xe000	0xf8ff
CJK Compatibility Ideographs	cjk_compatibility_ideographs	0xf900	0xfaff
Alphabetic Presentation Forms	alphabetic_presentation_forms	0xfb00	0xfb4f
Arabic Presentation Forms-A	arabic_presentation_forms-a	0xfb50	0xfdff
Variation Selectors	variation_selectors	0xfe00	0xfe0f
Vertical Forms	vertical_forms	0xfe10	0xfe1f
Combining Half Marks	combining_half_marks	0xfe20	0xfe2f
CJK Compatibility Forms	cjk_compatibility_forms	0xfe30	0xfe4f
Small Form Variants	small_form_variants	0xfe50	0xfe6f
Arabic Presentation Forms-B	arabic_presentation_forms-b	0xfe70	0xfeff
Halfwidth and Fullwidth Forms	halfwidth_and_fullwidth_forms	0xff00	0xffef
Specials	specials	0xfff0	0xffff

Transparency

The second parameter is for specifying the font transparency level (1 , 2 , 4 or 8). If unspecified, the encoded transparency level is 1 (does not depend on the transparency level encoded in the font file).

Examples:

- myfont:latin:4 : Embed all latin characters with 16 levels of transparency
- myfont::2 : Embed all characters with 4 levels of transparency

External Fonts

To fetch fonts from non-byte addressable external memory, the application must pre-register the *external Font resources*. The management of this kind of font may be different than the internal fonts and may require a dedicated heap. For more details about the external font management, refer to the VEE Port Guide chapter *External Resources*.

Font Generator Error Messages

Table 18: Static Font Generator Error Messages

ID	Type	Description
0	Error	The font generator has encountered an unexpected internal error.
1	Error	The Fonts list file has not been specified.
2	Error	The font generator cannot create the final, raw file.
3	Error	The font generator cannot read the fonts list file.
4	Warning	The font generator has found no font to generate.
5	Error	The font generator cannot load the fonts list file.
6	Warning	The specified font path is invalid: The font will be not converted.
7	Warning	There are too many arguments on a line: the current entry is ignored.
8	Error	The font generator has encountered an unexpected internal error (invalid output format).
9	Error	The font generator has encountered an unexpected internal error (invalid endianness).
10	Error	The specified entry is invalid.
11	Error	The specified entry does not contain a list of characters.
12	Error	The specified entry does not contain a list of identifiers.
13	Error	The specified entry is an invalid width.
14	Error	The specified entry is an invalid height.
15	Error	The specified entry does not contain the characters' addresses.
16	Error	The specified entry does not contain the characters' bitmaps.
17	Error	The specified entry bits-per-pixel value is invalid.
18	Error	The specified range is invalid.
19	Error	There are too many identifiers. The output RAW format cannot store all identifiers.
20	Error	The font's name is too long. The output RAW format cannot store all name characters.
21	Error	There are too many ranges. The output RAW format cannot store all ranges.
22	Error	Output list files cannot be created.
23	Error	Dynamic styles are not supported. Only a PLAIN font can be encoded.
24	Error	Underlined style is not supported. Only a BOLD and ITALIC font can be set.

Default Character

The application may request the rendering of a string where some characters are not available in the selected font. In that case, a default character is drawn instead: it is the first available character in the font. For example, the first available character for a font where the range matches the ASCII printable characters ($0x21\text{-}0x7E$) would be the exclamation mark ($0x21$).

The characters of a font are referenced by their Unicode value. For a given *font range*, the default character is the first character of the first range. Consequently, the default character may not be the same for two given fonts of an application: it depends on the specified character range for each font.

To help developers identify quickly why a string is rendered with unexpected characters, it is recommended that the font maker sets a default character that is easy to recognize (a symbol, for example, a rectangle). This character must have the first character index (index 0 is allowed).

Caching Generated Fonts

Fonts converted using the Font Generator can be cached so that they are not rebuilt every time the application is launched. Doing so can significantly speed up the application build phase.

The cache is enabled by default. It may be disabled by setting the *Application option* `ej.microui.fontConverter.disableCache` to `true`.

The Font Generator obeys several rules when choosing whether a font should be converted.

- If the cache is disabled, all fonts are generated every time the application is launched.
- All fonts will be regenerated if the application is launched using another VEE port and the new VEE port uses a different Font Generator.
- If the generated font does not exist, it will be generated.
- If the source font has been modified since the last time it was converted, the font will be regenerated.
- The font will be regenerated if the destination format or the range has been modified in the *fonts.list* file.

Cached fonts are stored in `.cache/fonts`, which is located in the *application output folder*. You may delete this directory to force the generation of all fonts in your application. A font that was previously generated but is no longer listed in the `*.fonts.list` files when the application is launched will be deleted from the cache directory.

Extended Font

A VEE Port can provide one of multiple subclasses of MicroUI Font. The way to open these extended fonts is specific to each subclass. However, each subclass should implement the default MicroUI Painter API to draw and transform the strings. This makes for portable code (as far as rendering is concerned).

- For more information about the way an extended font is added to a VEE Port, see *Custom Font*.
- For more information about the way an extended font is added to the application classpath, opened at runtime, its characteristics, its extended Painter API, etc., refer to the extended font documentation.

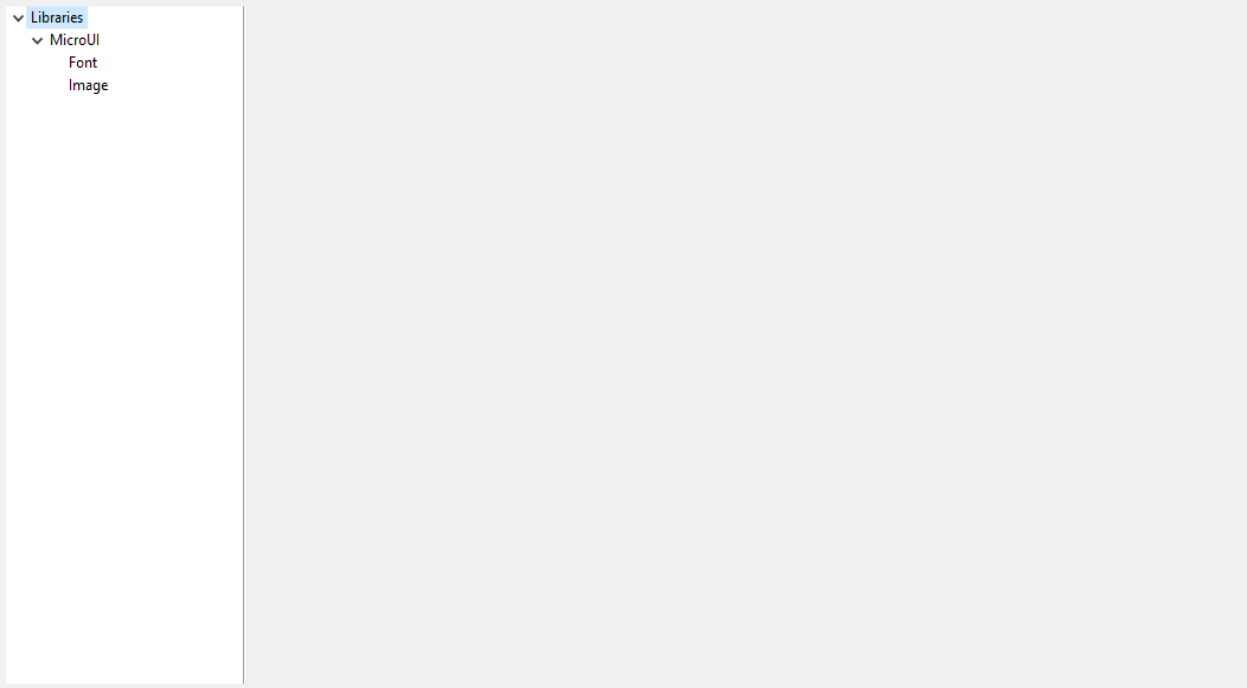
Note: MicroVG's VectorFont offers a way to retrieve an extended font.

Application Options

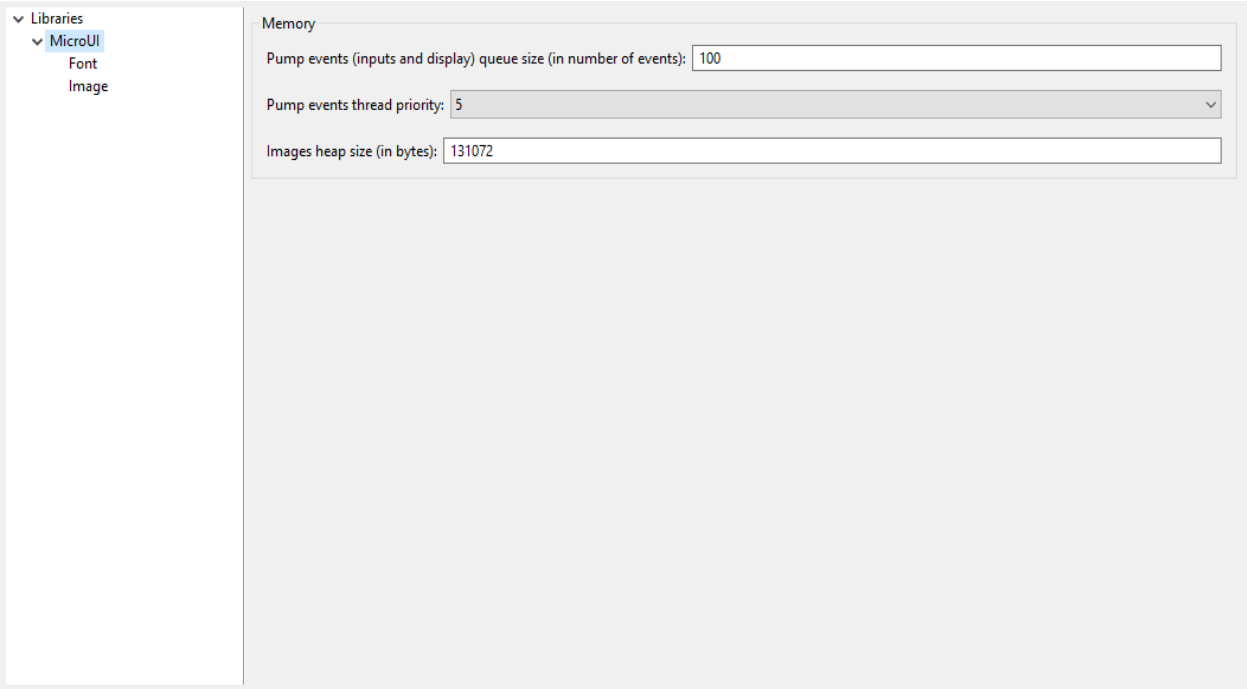
MicroUI libraries and its tools provide a set of options. See *Standalone Application Options* to have more information about the application options.

Note: MicroUI implementation requires one thread (MicroUI Pump) and at least 100 bytes in the *Immortals Heap*.

Category: Libraries



Category: MicroUI



Group: Memory

Option(text): Pump events (inputs and display) queue size (in number of events)

Option Name: ej.microui.memory.queue.size

Default value: 100

Description:

Specifies the size of the pump events queue.

Option(combo): Pump events thread priority

Option Name: com.microej.library.microui.pump.priority

Default value: 5

Available values: 1 to 10

Description:

Specifies the priority of the pump events queue.

Option(text): Images heap size (in bytes)

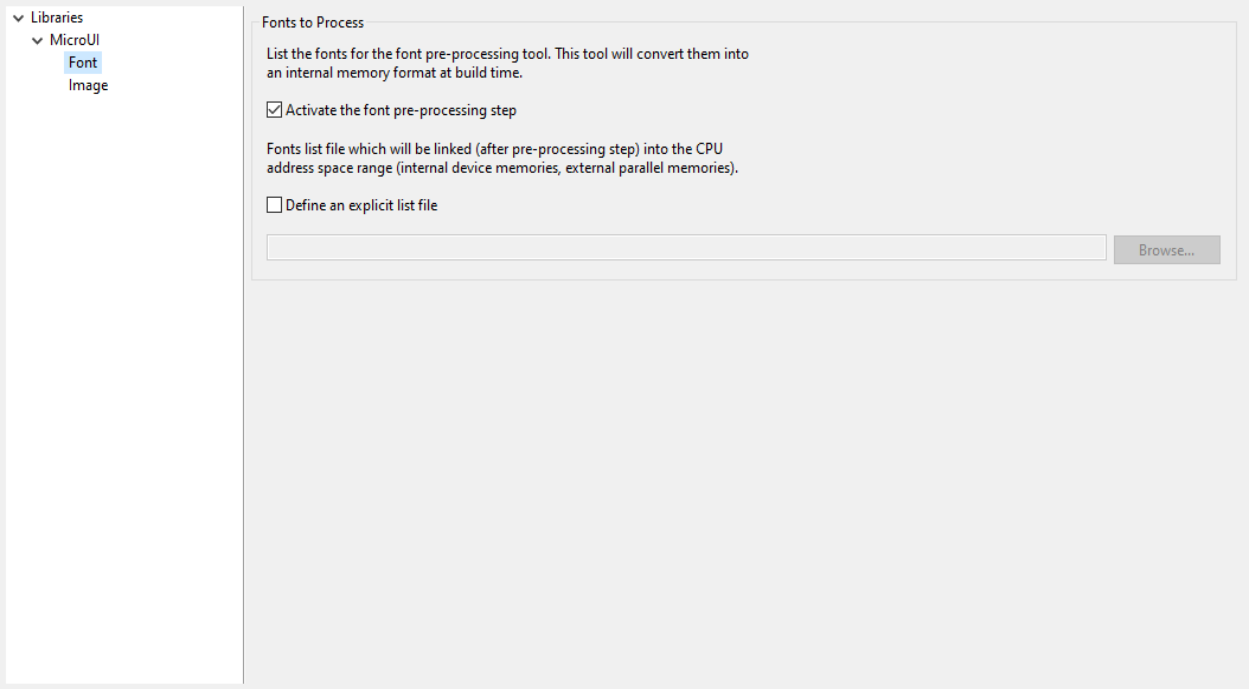
Option Name: ej.microui.memory.imagesheap.size

Default value: 131072

Description:

Specifies the size of the images heap. This heap is used to store the dynamic user images, the decoded images and the working buffers of embedded image decoders (for instance the PNG decoder). A too small value can cause OutOfMemory errors and incomplete drawings.

Category: Font



Group: Fonts to Process

Description:

This group allows to select a file describing the font files which need to be converted into a RAW format. At MicroUI runtime, the pre-generated fonts will be read from the flash memory without any modifications (see MicroUI specification).

Option(checkbox): Activate the font pre-processing step

Option Name: ej.microui.fontConverter.useIt

Default value: true

Description:

When checked, enables the next option Fonts list file. When the next option is disabled, there is no check on the file path validity.

Option(checkbox): Define an explicit list file

Option Name: ej.microui.fontConverter.file.enabled

Default value: false

Description:

By default, list files are loaded from the classpath. When checked, only the next option Fonts list file is processed.

Option(browse):

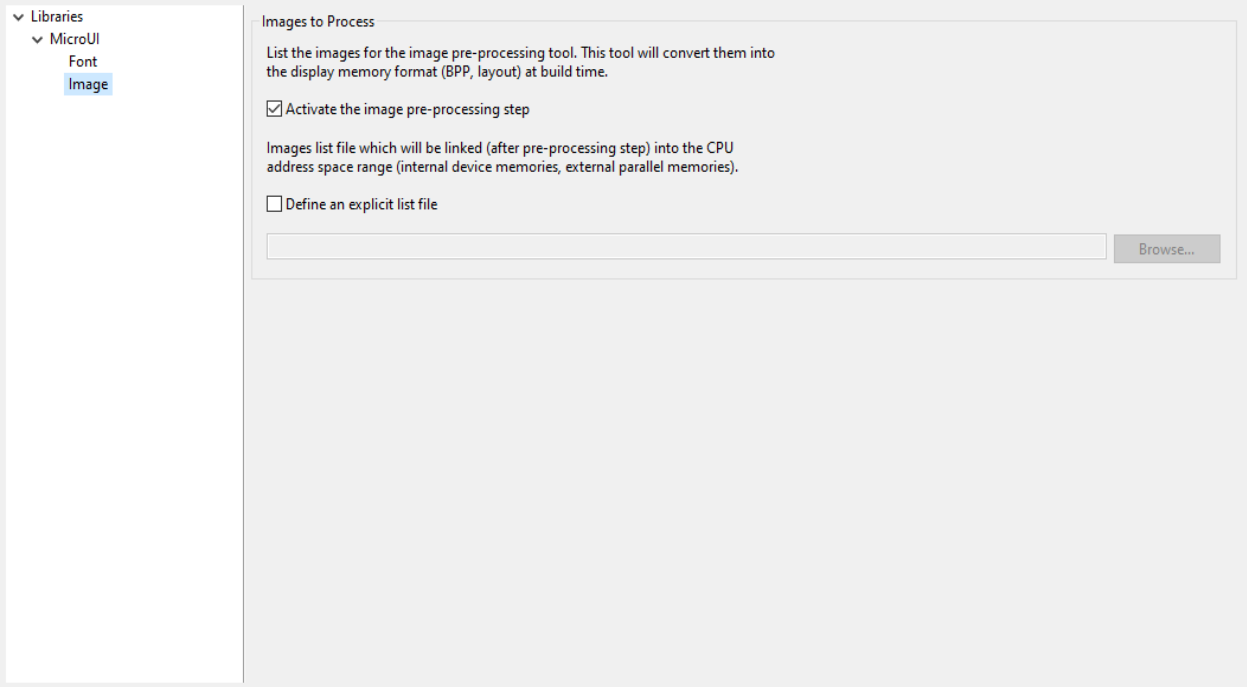
Option Name: ej.microui.fontConverter.file

Default value: (empty)

Description:

Browse to select a font list file. Refer to Font Generator chapter for more information about the font list file format.

Category: Image



Group: Images to Process

Description:

This group allows to select a file describing the image files which need to be converted into a RAW format. At MicroUI runtime, the pre-generated images will be read from the flash memory without any modifications (see MicroUI specification).

Option(checkbox): Activate the image pre-processing step

Option Name: ej.microui.imageConverter.useIt

Default value: true

Description:

When checked, enables the next option Images list file. When the next option is disabled, there is no check on the file path validity.

Option(checkbox): Define an explicit list file

Option Name: ej.microui.imageConverter.file.enabled

Default value: false

Description:

By default, list files are loaded from the classpath. When checked, only the next option Images list file is processed.

Option(browse):

Option Name: ej.microui.imageConverter.file

Default value: (empty)

Description:

Browse to select an image list file. Refer to Image Generator chapter for more information about the image list file format.

Debug Traces

MicroUI traces several actions when traces are enabled (see *Event Tracing*). This chapter explains the trace identifiers.

Trace format

The trace output format is the following:

[TRACE: MicroUI] Event AA(BB[CC],DD[EE])

where:

- AA is the event identifier. See next table.
- BB is the first event data.
- CC is the index of the first event data (0x0).
- DD is the second event data.
- EE is the index of the second event data (0x1).
- etc.

For example, given the following trace output:

[TRACE: MicroUI] Event 0x2(1[0x0],2[0x1],117571586[0x2])

- 0x2 -> Execute native input event
- 1 -> Event “Button” (index 0x0)
- 2 -> Generator Id (index 0x1)
- 117571586 -> event data (index 0x2)

Trace identifiers

The following tables describe some events data.

Warning: These tables are compatible with the UI Packs [12.1.5 - 14.5.2].

Table 19: MicroUI Traces

Event ID	Description	End of event
0x0 (0)	Execute EventGenerator event %0% (see Event Type). Generator id is %1% and data is %2%.	End of %0% (see Event Type).
0x1 (1)	Drop event %0%.	
0x2 (2)	Execute native input event %0% (see Event Type). Generator id is %1% and data is %2%.	End of %0% (see Event Type).
0x3 (3)	Execute display event %0% (see Event Type). Event is %1%.	End of %0% (see Event Type).
0x4 (4)	Execute user event %0%.	End of %0%.
0x5 (5)	Create new image using %0% algorithm (see Create Image).	Image created, image identifier is %0%.
0x6 (6)	New image characteristics: %0% algorithm (see Create Image) %1% (see Image Type), identifier is %2% and size is %3% x %4%.	
0x7 (7)	New image characteristics: %0% algorithm (see Create Image) %1% (see Image Type), identifier is %2%, size is %3% x %4% and format is %5% (see Image Format).	
0xa (10)	Flush start. Region (%0% , %1%) (%2% x %3%).	
0xb (11)	Flush done.	
0xc (12)	Start drawing operation %0% (see Drawing Type).	Drawing status %0% (see Drawing Status)
0xd (13)	Start drawing operation %0% (see Drawing Type).	Drawing status %0% (see Drawing Status)
0xf (15)	Asynchronous drawing operation done.	
0x10 (16)	Flush %0% from back buffer %1% start. Region (%2% , %3%) (%4% x %5%).	
0x11 (17)	Flush %0% done. New back buffer: %1%.	
0x14 (20)	Invalid input event %0%.	
0x15 (21)	Event queue is full, cannot add event %0%.	
0x16 (22)	Add event %0% at index %1%; queue length is %2%.	

continues on next page

Table 19 – continued from previous page

Event ID	Description	End of event
0x17 (23)	Replace event %0% by %1% at index %2% ; queue length is %3% .	
0x18 (24)	Read event %0% at index %1% .	
0x28 (40)	Start drawing operation %0% (see Drawing Type).	Drawing status %0% (see Drawing Status)
0x32 (50)	[BRS] New drawing region (%0% , %1%) to (%2% , %3%)	
0x33 (51)	[BRS] Flush LCD (id = %0% buffer = %1%) with a single region (%2% , %3%) to (%4% , %5%)	
0x34 (52)	[BRS] Flush LCD (id = %0% buffer = %1%) with %2% regions	
0x35 (53)	[BRS] Add a region (%0% , %1%) to (%2% , %3%)	
0x36 (54)	[BRS] Remove a region (%0% , %1%) to (%2% , %3%)	
0x37 (55)	[BRS] Restore a region (%0% , %1%) to (%2% , %3%)	
0x38 (56)	[BRS] Clear the list of regions	
0x39 (57)	[BRS] Restore a region (%0% , %1%) (%2% x %3%)	
0x3c (60)	Get pixel from image %0% (%1% , %2%)	
0x3d (61)	Get ARGB array from image %0% region (%1% , %2%) (%3% x %4%)	
0x3e (62)	Get string width (%0% characters)	
0x3f (63)	Set foreground color: dest %0% color = %1%	
0x40 (64)	Set background color: dest %0% color = %1%	
0x41 (65)	Remove background color: dest %0%	
0x42 (66)	Set clip: dest %0% region (%1% , %2%) (%3% x %4%)	
0x43 (67)	Draw ing out of clip:: dest %0%	
0x44 (68)	Drawn region: dest %0% clip disabled = %1% region (%2% , %3%) (%4% x %5%)	
0x50 (80)	Write pixel: dest %0% (%1% , %2%)	Drawing status %0% (see Drawing Status)
0x51 (81)	Draw line: dest %0% (%1% , %2%) to (%3% , %4%)	Drawing status %0% (see Drawing Status)
0x52 (82)	Draw horizontal line: dest %0% (%1% , %2%) length %3%	Drawing status %0% (see Drawing Status)
0x53 (83)	Draw vertical line: dest %0% (%1% , %2%) length %3%	Drawing status %0% (see Drawing Status)
0x54 (84)	Draw rectangle: dest %0% anchor (%1% , %2%) size (%3% x %4%)	Drawing status %0% (see Drawing Status)

continues on next page

Table 19 – continued from previous page

Event ID	Description	End of event
0x55 (85)	Fill rectangle: dest %0% anchor (%1% , %2%) size (%3% x %4%)	Drawing status %0% (see Drawing Status)
0x56 (86)	Draw rounded rectangle: dest %0% anchor (%1% , %2%) size (%3% x %4%) corner (%5% x %6%)	Drawing status %0% (see Drawing Status)
0x57 (87)	Fill rounded rectangle: dest %0% anchor (%1% , %2%) size (%3% x %4%) corner (%5% x %6%)	Drawing status %0% (see Drawing Status)
0x58 (88)	Draw circle arc: dest %0% anchor (%1% , %2%) diameter %3% start %5% angle %6%	Drawing status %0% (see Drawing Status)
0x59 (89)	Fill circle arc: dest %0% anchor (%1% , %2%) diameter %3% start %5% angle %6%	Drawing status %0% (see Drawing Status)
0x5a (90)	Draw ellipse arc: dest %0% anchor (%1% , %2%) size (%3% x %4%) start %5% angle %6%	Drawing status %0% (see Drawing Status)
0x5b (91)	Fill ellipse arc: dest %0% anchor (%1% , %2%) size (%3% x %4%) start %5% angle %6%	Drawing status %0% (see Drawing Status)
0x5c (92)	Draw ellipse: dest %0% anchor (%1% , %2%) size (%3% x %4%)	Drawing status %0% (see Drawing Status)
0x5d (93)	Fill ellipse: dest %0% anchor (%1% , %2%) size (%3% x %4%)	Drawing status %0% (see Drawing Status)
0x5e (94)	Draw circle: dest %0% anchor (%1% , %2%) diameter %3%	Drawing status %0% (see Drawing Status)
0x5f (95)	Fill circle: dest %0% anchor (%1% , %2%) diameter %3%	Drawing status %0% (see Drawing Status)
0x60 (96)	Draw image: dest %0% image %1% from (%2% , %3%) size (%4% x %5%) anchor (%6% , %7%) alpha %8%	Drawing status %0% (see Drawing Status)
0x61 (97)	Draw string: dest %0% (%1% characters) anchor (%2% , %3%)	Drawing status %0% (see Drawing Status)
0x62 (98)	Draw renderable string: dest %0% (%1% characters) anchor (%2% , %3%)	Drawing status %0% (see Drawing Status)
0x6e (110)	Draw thick faded point: dest %0% (%1% , %2%) thickness %3% fade %4%	Drawing status %0% (see Drawing Status)
0x6f (111)	Draw thick faded line: dest %0% (%1% , %2%) to (%3% , %4%) thickness %5% fade %6%	Drawing status %0% (see Drawing Status)
0x70 (112)	Draw thick faded circle: dest %0% anchor (%1% , %2%) diameter %3% thickness %4% fade %5%	Drawing status %0% (see Drawing Status)
0x71 (113)	Draw thick faded circle arc: dest %0% anchor (%1% , %2%) diameter %3% start %5% angle %6% thickness %7% fade %8%	Drawing status %0% (see Drawing Status)
0x72 (114)	Draw thick faded ellipse: dest %0% anchor (%1% , %2%) size (%3% x %4%) thickness %5% fade %6%	Drawing status %0% (see Drawing Status)
0x73 (115)	Draw thick line: dest %0% (%1% , %2%) to (%3% , %4%) thickness %5%	Drawing status %0% (see Drawing Status)
0x74 (116)	Draw thick circle: dest %0% anchor (%1% , %2%) diameter %3% thickness %4%	Drawing status %0% (see Drawing Status)
0x75 (117)	Draw thick ellipse: dest %0% anchor (%1% , %2%) size (%3% x %4%) thickness %5%	Drawing status %0% (see Drawing Status)
0x76 (118)	Draw thick Circle arc: dest %0% anchor (%1% , %2%) diameter %3% start %4% angle %5% thickness %6% fade %7%	Drawing status %0% (see Drawing Status)
0x77 (119)	Draw polygon: dest %0% lenght %1%	Drawing status %0% (see Drawing Status)

continues on next page

Table 19 – continued from previous page

Event ID	Description	End of event
0x78 (120)	Fill polygon: dest %0% lenght %1%	Drawing status %0% (see Drawing Status)
0x82 (130)	Draw flipped image: dest %0% image %1% from (%2% , %3%) size (%4% x %5%) anchor (%6% , %7%) %8% (see Flip Type) alpha %9%	Drawing status %0% (see Drawing Status)
0x83 (131)	Draw rotated image: dest %0% image %1% anchor (%2% , %3%) center=(%4% , %5%) angle %6% alpha %7% approx %8% (see Approximation)	Drawing status %0% (see Drawing Status)
0x84 (132)	Draw scaled image: dest %0% image %1% anchor (%2% , %3%) factor %4% alpha %5% approx %6% (see Approximation)	Drawing status %0% (see Drawing Status)
0x85 (133)	Draw scaled string: dest %0% (%1% characters) anchor (%2% , %3%) factor %4% approx %5% (see Approximation)	Drawing status %0% (see Drawing Status)
0x86 (134)	Draw scaled renderable string: dest %0% (%1% characters) anchor (%2% , %3%) factor %4% approx %5% (see Approximation)	Drawing status %0% (see Drawing Status)
0x87 (135)	Draw rotated character: dest %0% char %1% anchor (%2% , %3%) center (%4% , %5%) angle %6% alpha %7% approx %8% (see Approximation)	Drawing status %0% (see Drawing Status)
0x88 (136)	Draw image deformed: dest %0% image %1%	Drawing status %0% (see Drawing Status)

Table 20: Event Type

Event ID	Description
0x0 (0)	Event “Command”
0x1 (1)	Event “Button”
0x2 (2)	Event “Pointer”
0x3 (3)	Event “State”
0x4 (4)	Event “Unknwon”
0x5 (5)	Event “Call Serially”
0x6 (6)	Event “MicroUI Stop”
0x7 (7)	Event “Input”
0x8 (8)	Event “Show Displayable”
0x9 (9)	Event “Hide Displayable”
0xb (11)	Event “Pending Flush”
0xc (12)	Event “Force Flush”
0xd (13)	Event “Repaint Displayable”
0xe (14)	Event “Repaint Current Displayable”
0xf (15)	Event “KF Stop Feature”

Table 21: Create Image

Event ID	Description
0x0 (0)	Create BufferedImage
0x1 (1)	Create Image from path
0x2 (2)	Create Image from InputStream

Table 22: Image Type

Event ID	Description
0x0 (0)	New BufferedImage
0x1 (1)	Load MicroEJ Image from RAW file
0x2 (2)	New MicroEJ Image from encoded image
0x3 (3)	New MicroEJ Image from RAW image in external memory
0x4 (4)	New MicroEJ Image from encoded image in external memory
0x5 (5)	New MicroEJ Image from memory InputStream
0x6 (6)	New MicroEJ Image from byte array InputStream
0x7 (7)	New MicroEJ Image from generic InputStream
0x8 (8)	Link Image

Table 23: Image Format

Event ID	Description
0x0 (0)	DISPLAY
0x1 (1)	ARGB1565_RLE
0x2 (2)	ARGB8888
0x3 (3)	RGB888
0x4 (4)	RGB565
0x5 (5)	ARGB1555
0x6 (6)	ARGB4444
0x7 (7)	A4
0x8 (8)	A8
0x9 (9)	LRGB888
0xA (10)	LARGB8888
0xB (11)	A2
0xC (12)	A1
0xD (13)	C4
0xE (14)	C2
0xF (15)	C1
0x10 (16)	AC44
0x11 (17)	AC22
0x12 (18)	AC11
0x13 (19)	F8
0x14 (20)	F4
0x15 (21)	F2
0x16 (22)	F1
0x17 (23)	A8_RLE
0x18 (24)	ARGB8888_PRE
0x19 (25)	ARGB1555_PRE
0x1A (26)	ARGB4444_PRE
0x50 (80)	DEFAULT
0xF8 (248)	CUSTOM7
0xF9 (249)	CUSTOM6
0xFA (250)	CUSTOM5
0xFB (251)	CUSTOM4
0xFC (252)	CUSTOM3
0xFD (253)	CUSTOM2
0xFE (254)	CUSTOM1

continues on next page

Table 23 – continued from previous page

Event ID	Description
0xFF (255)	CUSTOM0

Table 24: Drawing Type

Event ID	Description
0x1 (1)	Write pixel
0x2 (2)	Draw line
0x3 (3)	Draw horizontal line
0x4 (4)	Draw vertical line
0x5 (5)	Draw rectangle
0x6 (6)	Fill rectangle
0x7 (7)	Unknown
0x8 (8)	Draw rounded rectangle
0x9 (9)	Fill rounded rectangle
0xa (10)	Draw circle arc
0xb (11)	Fill circle arc
0xc (12)	Draw ellipse arc
0xd (13)	Fill ellipse arc
0xe (14)	Draw ellipse
0xf (15)	Fill ellipse
0x10 (16)	Draw circle
0x11 (17)	Fill circle
0x12 (18)	Draw ARGB array
0x13 (19)	Draw image
0x14 (20)	Draw string
0x15 (21)	Get string width
0x32 (50)	Draw polygon
0x33 (51)	Fill polygon
0x34 (52)	Get ARGB image data
0x35 (53)	Draw string
0x36 (54)	Draw deformed string
0x37 (55)	Draw deformed image
0x38 (56)	Draw rotated character bilinear
0x39 (57)	Draw rotated character nearest neighbor
0x3a (58)	Get string width
0x3b (59)	Get pixel
0x64 (100)	Draw thick faded point
0x65 (101)	Draw thick faded line
0x66 (102)	Draw thick faded circle
0x67 (103)	Draw thick faded circle arc
0x68 (104)	Draw thick faded ellipse
0x69 (105)	Draw thick line
0x6a (106)	Draw thick circle
0x6b (107)	Draw thick ellipse
0x6c (108)	Draw thick circle arc
0xc8 (200)	Draw image with flip
0xc9 (201)	Draw image with rotation (simple)
0xca (202)	Draw image with rotation (bilinear)
0xcb (203)	Draw image with scalling (simple)

continues on next page

Table 24 – continued from previous page

Event ID	Description
0xcc (204)	Draw image with scalling (bilinear)
0xcd (205)	Draw string with scaling (bilinear)
0xce (206)	Draw character with rotation (bilinear)
0xcf (207)	Draw character with rotation (simple)

Table 25: Drawing Status

Event ID	Description
0x0 (0)	Synchronous drawing done
0x1 (1)	Asynchronous drawing delayed

Table 26: Flip Type

Event ID	Description
0x0 (0)	none
0x1 (1)	mirror 90°
0x2 (2)	270°
0x3 (3)	mirror
0x4 (4)	180°
0x5 (5)	mirror 270°
0x6 (6)	mirror 180°
0x7 (7)	90°

Table 27: Approximation

Event ID	Description
0x0 (0)	bilinear
0x1 (1)	nearest neighbor

SystemView Integration

The traces are *SystemView* compatible.

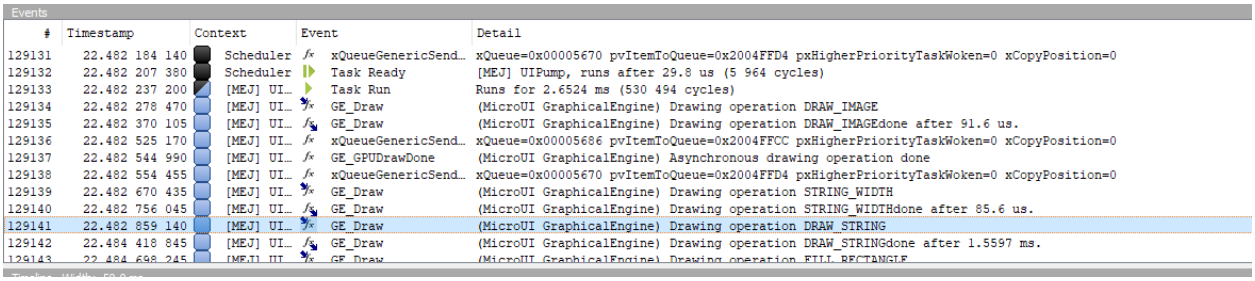


Fig. 136: MicroUI Traces displayed in SystemView

The following text can be copied in a file called SYSVIEW_MicroUI.txt and copied in SystemView installation folder (e.g. SEGGER/SystemView_V252a/Description/). Please note that SystemView requires a newline at the end of the file; otherwise, the events may not be decoded correctly.

Warning: These traces are compatible with the UI Packs [12.1.5 - 14.5.2] (see the known issue in the UI Packs *14.4.1* and *14.4.2*).

```
#
# SystemView Description File
#
#============
# NamedTypes
#============

NamedType Bool 0=false
NamedType Bool 1=true

# ----------
# MicroUI Events
# ----------

NamedType UIEvent 0=COMMAND
NamedType UIEvent 1=BUTTON
NamedType UIEvent 2=POINTER
NamedType UIEvent 3=STATE
NamedType UIEvent 4=UNKNOWN
NamedType UIEvent 5=CALLSERIALLY
NamedType UIEvent 6=STOP
NamedType UIEvent 7=INPUT
NamedType UIEvent 8=SHOW_DISPLAYABLE
NamedType UIEvent 9=HIDE_DISPLAYABLE
NamedType UIEvent 11=PENDING_FLUSH
NamedType UIEvent 12=FORCE_FLUSH
NamedType UIEvent 13=REPAINT_DISPLAYABLE
NamedType UIEvent 14=REPAINT_CURRENT_DISPLAYABLE
NamedType UIEvent 15=KF_STOP_FEATURE
NamedType UIEvent *="%u"

# --------
# Image Events
# --------

NamedType UINewImage 0=MUTABLE_IMAGE
NamedType UINewImage 1=IMAGE_FROM_PATH
NamedType UINewImage 2=IMAGE_FROM_INPUTSTREAM

NamedType UIImageData 0=BUFFER
NamedType UIImageData 1=RAW
NamedType UIImageData 2=ENCODED
NamedType UIImageData 3=RAW_EXTERNAL
NamedType UIImageData 4=ENCODED_EXTERNAL
NamedType UIImageData 5=MEMORY_INPUTSTREAM
NamedType UIImageData 6=BYTEARRAY_INPUTSTREAM
NamedType UIImageData 7=GENERIC_INPUTSTREAM
NamedType UIImageData 8=LINK_IMAGE

NamedType UIImageFormat 0=DISPLAY
NamedType UIImageFormat 1=ARGB1565_RLE
NamedType UIImageFormat 2=ARGB8888
NamedType UIImageFormat 3=RGB888
```

(continues on next page)

(continued from previous page)

```
NamedType UIImageFormat 4=RGB565
NamedType UIImageFormat 5=ARGB1555
NamedType UIImageFormat 6=ARGB4444
NamedType UIImageFormat 7=A4
NamedType UIImageFormat 8=A8
NamedType UIImageFormat 9=LRGB888
NamedType UIImageFormat 10=LARGB8888
NamedType UIImageFormat 11=A2
NamedType UIImageFormat 12=A1
NamedType UIImageFormat 13=C4
NamedType UIImageFormat 14=C2
NamedType UIImageFormat 15=C1
NamedType UIImageFormat 16=AC44
NamedType UIImageFormat 17=AC22
NamedType UIImageFormat 18=AC11
NamedType UIImageFormat 19=F8
NamedType UIImageFormat 20=F4
NamedType UIImageFormat 21=F2
NamedType UIImageFormat 22=F1
NamedType UIImageFormat 23=A8_RLE
NamedType UIImageFormat 24=ARGB8888_PRE
NamedType UIImageFormat 25=ARGB1555_PRE
NamedType UIImageFormat 26=ARGB4444_PRE
NamedType UIImageFormat 80=DEFAULT
NamedType UIImageFormat 248=CUSTOM7
NamedType UIImageFormat 249=CUSTOM6
NamedType UIImageFormat 250=CUSTOM5
NamedType UIImageFormat 251=CUSTOM4
NamedType UIImageFormat 252=CUSTOM3
NamedType UIImageFormat 253=CUSTOM2
NamedType UIImageFormat 254=CUSTOM1
NamedType UIImageFormat 255=CUSTOM0
NamedType UIImageFormat *="%u"

# ----------
# Drawing Events
# ----------

NamedType UIDrawAsync 0="Drawing operation done"
NamedType UIDrawAsync 1="Drawing operation delayed"

NamedType UIDestination  *="dest=0x%x"
NamedType UIFlush  *="flush=%u"
NamedType UISource  *="image=0x%x"
NamedType UIDiameter *="diameter=%u"
NamedType UIAlpha *="alpha=%u"
NamedType UIThickness *="thickness=%u"
NamedType UIFade *="fade=%u"
NamedType UIString *="%u characters"

NamedType UIApprox 0="(bilinear)"
NamedType UIApprox 1="(nearest neighbor)"

NamedType UIFlip 0="flip=none"
NamedType UIFlip 1="flip=mirror90"
NamedType UIFlip 2="flip=270"
```

(continues on next page)

(continued from previous page)

```
NamedType UIFlip 3="flip=mirror"
NamedType UIFlip 4="flip=180"
NamedType UIFlip 5="flip=mirror270"
NamedType UIFlip 6="flip=mirror180"
NamedType UIFlip 7="flip=90"

# -----------
# Old Drawing Events
# -----------

# Deprecated (kept for backward compatibility UI Pack < 14.4.1)

NamedType UIDrawAlgo 1=WRITE_PIXEL
NamedType UIDrawAlgo 2=DRAW_LINE
NamedType UIDrawAlgo 3=DRAW_HORIZONTALLINE
NamedType UIDrawAlgo 4=DRAW_VERTICALLINE
NamedType UIDrawAlgo 5=DRAW_RECTANGLE
NamedType UIDrawAlgo 6=FILL_RECTANGLE
NamedType UIDrawAlgo 7=UNKNOWN
NamedType UIDrawAlgo 8=DRAW_ROUNDEDRECTANGLE
NamedType UIDrawAlgo 9=FILL_ROUNDEDRECTANGLE
NamedType UIDrawAlgo 10=DRAW_CIRCLEARC
NamedType UIDrawAlgo 11=FILL_CIRCLEARC
NamedType UIDrawAlgo 12=DRAW_ELLIPSEARC
NamedType UIDrawAlgo 13=FILL_ELLIPSEARC
NamedType UIDrawAlgo 14=DRAW_ELLIPSE
NamedType UIDrawAlgo 15=FILL_ELLIPSE
NamedType UIDrawAlgo 16=DRAW_CIRCLE
NamedType UIDrawAlgo 17=FILL_CIRCLE
NamedType UIDrawAlgo 18=DRAW_ARGB
NamedType UIDrawAlgo 19=DRAW_IMAGE
NamedType UIDrawAlgo 20=DRAW_STRING
NamedType UIDrawAlgo 21=STRING_WIDTH

NamedType UIDrawAlgo 50=DRAW_POLYGON
NamedType UIDrawAlgo 51=FILL_POLYGON
NamedType UIDrawAlgo 52=GET_ARGB
NamedType UIDrawAlgo 53=DRAW_STRING
NamedType UIDrawAlgo 54=DRAW_DEFORMED_STRING
NamedType UIDrawAlgo 55=DRAW_DEFORMED_IMAGE
NamedType UIDrawAlgo 56=DRAW_CHAR_ROTATION_BILINEAR
NamedType UIDrawAlgo 57=DRAW_CHAR_ROTATION_SIMPLE
NamedType UIDrawAlgo 58=STRING_WIDTH
NamedType UIDrawAlgo 59=READ_PIXEL

NamedType UIDrawAlgo 100=DRAW_THICK_FADED_POINT
NamedType UIDrawAlgo 101=DRAW_THICK_FADED_LINE
NamedType UIDrawAlgo 102=DRAW_THICK_FADED_CIRCLE
NamedType UIDrawAlgo 103=DRAW_THICK_FADED_CIRCLE_ARC
NamedType UIDrawAlgo 104=DRAW_THICK_FADED_ELLIPSE
NamedType UIDrawAlgo 105=DRAW_THICK_LINE
NamedType UIDrawAlgo 106=DRAW_THICK_CIRCLE
NamedType UIDrawAlgo 107=DRAW_THICK_ELLIPSE
NamedType UIDrawAlgo 108=DRAW_THICK_CIRCLEARC

NamedType UIDrawAlgo 200=DRAW_FLIPPED_IMAGE
```

(continues on next page)

(continued from previous page)

```
NamedType UIDrawAlgo 201=DRAW_ROTATED_IMAGE_NEARESTNEIGHBOR
NamedType UIDrawAlgo 202=DRAW_ROTATED_IMAGE_BILINEAR
NamedType UIDrawAlgo 203=DRAW_SCALED_IMAGE_NEARESTNEIGHBOR
NamedType UIDrawAlgo 204=DRAW_SCALED_IMAGE_BILINEAR
NamedType UIDrawAlgo 205=DRAW_SCALED_STRING_BILINEAR
NamedType UIDrawAlgo 206=DRAW_CHAR_WITH_ROTATION_BILINEAR
NamedType UIDrawAlgo 207=DRAW_CHAR_WITH_ROTATION_NEARESTNEIGHBOR

#=======
# Events
#=======

# ----------
# MicroUI Library
# ----------

# [0-9]: 10 events

# Deprecated (kept for backward compatibility UI Pack < 14.5.2)
6       UI_ImageData        type=%UINewImage data=%UIImageData id=0x%x size=%u*%u

0       UI_ExecuteGenEvent  type=%UIEvent (generatorID=%u data=0x%x)   | event=%UIEvent done
1       UI_UnknownEvent     drop=0x%x
2       UI_InputEvent       type=%UIEvent (generatorID=%u, event=0x%x) | event=%UIEvent done
3       UI_DisplayEvent     event=%UIEvent (0x%x)                      | event=%UIEvent done
4       UI_UserEvent        event=0x%x                                 | event=0x%x done
5       UI_NewImage         type=%UINewImage                           | id=0x%x
7       UI_ImageData        type=%UINewImage data=%UIImageData id=0x%x size=%u*%u format=
↪%UIImageFormat

# ----------------
# MicroUI Graphics Engine
# ----------------

# [10-19]: 10 events

# Deprecated (kept for backward compatibility UI Pack < 14.4.1)
10      UI_FlushStart       region=(%u,%u)(%ux%u)
11      UI_FlushDone        flush done
12      UI_DrawInternal     draw=%UIDrawAlgo                   | draw=%UIDrawAlgo done
13      UI_Draw             draw=%UIDrawAlgo                   | draw=%UIDrawAlgo done

15      UI_GPUDrawDone      asynchronous drawing operation done
16      UI_Flush            %UIFlush %UIDestination region=(%u,%u)(%ux%u)
17      UI_FlushDone        %UIFlush %UIDestination

# --------------
# MicroUI Events Engine
# --------------

# [20-29]: 10 events

20      UI_InvalidEvent     invalid=0x%x
21      UI_QueueFull        drop=0x%x
22      UI_AddEvent         add=0x%x (index=%u queueLength=%u)
23      UI_ReplaceEvent     replace=0x%x by 0x%x (index=%u queueLength=%u)
```

(continues on next page)

(continued from previous page)

```
24      UI_ReadEvent        read=0x%x (index=%u)

# --------------
# MicroUI LED Engine
# --------------

# [30-39]: 10 events

# -----------
# MicroUI C Module
# -----------

# [40-139]: 100 events

# Deprecated (kept for backward compatibility UI Pack < 14.4.1)
40      UI_Draw             %UIDrawAlgo  |  %UIDrawAsync
50      BRS_NewRegion       region (%u,%u) to (%u,%u)
53      BRS_AddRegion       (%u,%u) to (%u,%u)
54      BRS_RemoveRegion    (%u,%u) to (%u,%u)
55      BRS_RestoreRegion   (%u,%u) to (%u,%u)
56      BRS_ClearList

51      BRS_FlushSingleRect  %UIFlush %UIDestination (region (%u,%u) to (%u,%u))
52      BRS_FlushMultiRect   %UIFlush %UIDestination (%u regions)
57      BRS_RestoreRegion    region=(%u,%u)(%ux%u)

60      UI_GetPixel          %UISource %d,%d
61      UI_GetImageARGB          %UISource from %d,%d size=%dx%d
62      UI_StringWidth         %UIString
63      UI_SetForegroundColor    %UIDestination color=0x%x
64      UI_SetBackgroundColor    %UIDestination color=0x%x
65      UI_RemoveBackgroundColor  %UIDestination
# // clip seen as a container (use a START event)
66      UI_SetClip           %UIDestination region=(%u,%u)(%ux%u) | (force START event)
67      UI_OutOfClip           %UIDestination
# // region seen as a drawing (use a STANDALONE event)
68      UI_DrawnRegion           %UIDestination clipDisabled=%Bool region=(%u,%u)(%ux%u)

80      UI_WritePixel          %UIDestination %d,%d | %UIDrawAsync
81      UI_DrawLine            %UIDestination %d,%d to %d,%d | %UIDrawAsync
82      UI_DrawHorizontalLine    %UIDestination %d,%d length=%d | %UIDrawAsync
83      UI_DrawVerticalLine      %UIDestination %d,%d length=%d | %UIDrawAsync
84      UI_DrawRectangle         %UIDestination anchor=%d,%d size=%dx%d | %UIDrawAsync
85      UI_FillRectangle        %UIDestination anchor=%d,%d size=%dx%d | %UIDrawAsync
86      UI_DrawRoundedRectangle  %UIDestination anchor=%d,%d size=%dx%d corner=%dx%d |
↪%UIDrawAsync
87      UI_FillRoundedRectangle  %UIDestination anchor=%d,%d size=%dx%d corner=%dx%d | %UIDrawAsync
88      UI_DrawCircleArc         %UIDestination anchor=%d,%d %UIDiameter start=%d angle=%d |
↪%UIDrawAsync
89      UI_FillCircleArc        %UIDestination anchor=%d,%d %UIDiameter start=%d angle=%d |
↪%UIDrawAsync
90      UI_DrawEllipseArc        %UIDestination anchor=%d,%d size=%dx%d start=%d angle=%d |
↪%UIDrawAsync
91      UI_FillEllipseArc       %UIDestination anchor=%d,%d size=%dx%d start=%d angle=%d |
↪%UIDrawAsync
92      UI_DrawEllipse           %UIDestination anchor=%d,%d size=%dx%d | %UIDrawAsync
```

(continues on next page)

(continued from previous page)

```
93      UI_FillEllipse          %UIDestination anchor=%d,%d size=%dx%d | %UIDrawAsync
94      UI_DrawCircle           %UIDestination anchor=%d,%d %UIDiameter | %UIDrawAsync
95      UI_FillCircle           %UIDestination anchor=%d,%d %UIDiameter | %UIDrawAsync
96      UI_DrawImage            %UIDestination %UISource from %d,%d size=%dx%d anchor=%d,%d
↪%UIAlpha | %UIDrawAsync
97      UI_DrawString           %UIDestination %UIString anchor=%d,%d | %UIDrawAsync
98      UI_DrawRenderableString %UIDestination %UIString anchor=%d,%d | %UIDrawAsync

110     UI_DrawThickFadedPoint      %UIDestination %d,%d %UIThickness %UIFade | %UIDrawAsync
111     UI_DrawThickFadedLine       %UIDestination %d,%d to %d,%d %UIThickness %UIFade |
↪%UIDrawAsync
112     UI_DrawThickFadedCircle     %UIDestination anchor=%d,%d %UIDiameter %UIThickness %UIFade |
↪%UIDrawAsync
113     UI_DrawThickFadedCircleArc  %UIDestination anchor=%d,%d %UIDiameter start=%d angle=%d
↪%UIThickness %UIFade | %UIDrawAsync
114     UI_DrawThickFadedEllipse    %UIDestination anchor=%d,%d size=%dx%d %UIThickness %UIFade |
↪%UIDrawAsync
115     UI_DrawThickLine            %UIDestination %d,%d to %d,%d %UIThickness | %UIDrawAsync
116     UI_DrawThickCircle          %UIDestination anchor=%d,%d %UIDiameter %UIThickness |
↪%UIDrawAsync
117     UI_DrawThickEllipse         %UIDestination anchor=%d,%d size=%dx%d %UIThickness | %UIDrawAsync
118     UI_DrawThickCircleArc       %UIDestination anchor=%d,%d %UIDiameter start=%d angle=%d
↪%UIThickness %UIFade | %UIDrawAsync
119     UI_DrawPolygon              %UIDestination lenght=%u | %UIDrawAsync
120     UI_FillPolygon              %UIDestination lenght=%u | %UIDrawAsync

130     UI_DrawFlippedImage         %UIDestination %UISource from %d,%d size=%dx%d anchor=%d,%d
↪%UIFlip %UIAlpha | %UIDrawAsync
131     UI_DrawRotatedImage         %UIDestination %UISource anchor=%d,%d center=%d,%d angle=%d
↪%UIAlpha %UIApprox | %UIDrawAsync
132     UI_DrawScaledImage          %UIDestination %UISource anchor=%d,%d factor=%dx%d %UIAlpha
↪%UIApprox | %UIDrawAsync
133     UI_DrawScaledString         %UIDestination %UIString anchor=%d,%d factor=%dx%d %UIApprox |
↪%UIDrawAsync
134     UI_DrawScaledRenderableString  %UIDestination %UIString anchor=%d,%d factor=%dx%d %UIApprox␣
↪| %UIDrawAsync
135     UI_DrawRotatedCharacter     %UIDestination char=%c anchor=%d,%d center=%d,%d angle=%d
↪%UIAlpha %UIApprox | %UIDrawAsync
136     UI_DrawImageDeformed        %UIDestination %UISource | %UIDrawAsync
```

Hint: Retrieve a compatible file with the VEE Port's UI Pack in the UI Pack (open it with as a zip): in the VEE Port configuration project, open [veeport-configuration]\target~\dependencies\[architecture]-ui-pack.xpfp\build\fragments\microui\content\lib\SYSVIEW_MicroUI.txt .

Error Messages

When an exception is thrown by the implementation of the MicroUI API, the exception MicroUIException with the error message `MicroUI:E=<messageId>` is issued, where the meaning of `<messageId>` is defined in the *Field Detail* of each error code.

Migration Guide

The MicroUI implementation is provided by the UI Pack. According to the UI Pack used to build the VEE Port, the application has to be updated.

- Refer to the *table* that illustrates the implemented MicroUI API for each UI Pack.
- Refer to the latest MicroUI API Changelog.
- Refer to the latest Drawing API Changelog.

The following chapters describe the changes to perform in the application according the UI Pack used to build the VEE Port.

From 13.x to 14.x

Consider the new semantic of the content of the back buffer after a flush: the past is not systematically restored, see *Buffer Refresh Strategy*. The application code may be affected by this update. Ensure that a clip is set before each initial drawing and that it has the same size as what is drawn (at least that every pixels in the clip is drawn).

From 12.x to 13.x

- Update ej.api#microui dependency to the latest available version `3.4.0`.
- Add ej.api#drawing dependency.

Gradle (build.gradle.kts)

MMM (module.ivy)

```
implementation("ej.api:microui:3.4.0")
implementation("ej.api:drawing:1.0.4")
```

```
<dependencies>
  <dependency org="ej.api" name="microui" rev="3.4.0"/>
  <dependency org="ej.api" name="drawing" rev="1.0.4"/>
</dependencies>
```

From 10.x to 12.x

- In MicroEJ application launcher > Configuration tab > *MicroUI*: check Use Flying Images when the application is using the flying images (property com.microej.library.microui.flyingimage.enabled).
- In MicroEJ application launcher, increase the *Managed Heap*: it now contains MicroUI images metadata (size, format, clip etc.). The icetea heap has been automatically decreased.

From 9.x to 10.x

- In MicroEJ application launcher > Configuration tab > *MicroUI*: set the image heap size (property *ej.microui.memory.imagesheap.size*).

MicroVG

MicroVG Foundation Library provides vector drawing capabilities.

Usage

To use the MicroVG Foundation Library, add MicroVG API module to the project build file:

Gradle (build.gradle.kts)

MMM (module.ivy)

```
implementation("ej.api:microvg:1.5.0")
```

```
<dependency org="ej.api" name="microvg" rev="1.5.0"/>
```

The MicroVG Library brings the following features:

- the creation and drawing of paths with color or linear gradient.
- the drawing of texts using vector fonts with color or linear gradient.
- the drawing of vector images.
- the transformation of paths, texts, images with affine transformation matrices.

Note: The MicroVG library natives use different drawing engines, font rendering and layout engines for embedded and simulator implementations.

This can lead to some slightly drawing differences, like for instance in the antialiasing processing of font glyphs.

Path

Path Creation

The MicroVG library enables the creation of vector paths composed of the following commands:

- Move
- Line
- Cubic Bezier Curve
- Quadratic Bezier Curve
- Close

The coordinates of the points associated with these commands can be absolute or relative.

```
Path path = new Path();

path.moveTo(70, 20);
path.cubicTo(0, 0, 10, 50, 80, 90);
path.lineTo(95, 75);
path.quadTo(12, 40, 80, 50);
path.close();
```

```
Path path = new Path();

path.moveTo(70, 20);
path.cubicToRelative(-70, -20, -60, 30, 10, 70);
path.lineToRelative(15, -15);
path.quadToRelative(-83, -35, -15, -25);
path.close();
```

Path Drawing

A path can be drawn with a call to ej.microvg.VectorGraphicsPainter.fillPath().

The drawn path will be filled with the graphic context color or with a linear gradient.

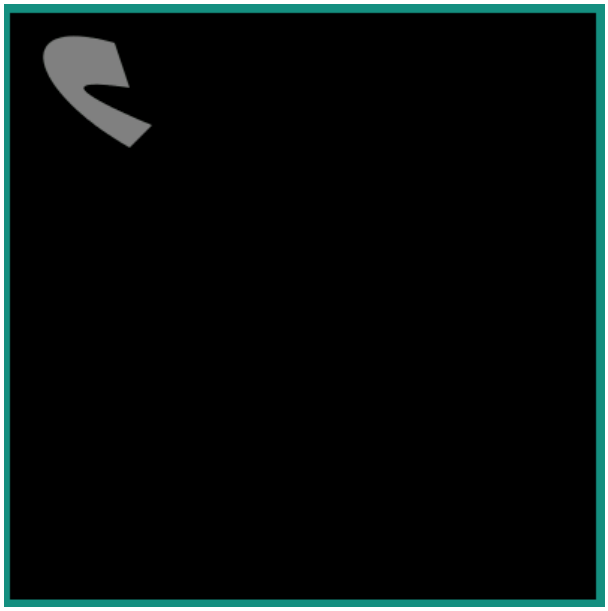
The path can be transformed by a transformation matrix (this concept is explained in *Matrix* section) before drawing.

A *FillType* and an *Alpha Blending Mode* can be applied during the drawing.

Fill Path With Graphics Context Color

The default alpha channel value of the drawing is `0xFF` (opaque opacity).

```
g.setColor(Colors.GRAY);
VectorGraphicsPainter.fillPath(g, path, 0, 0);
```

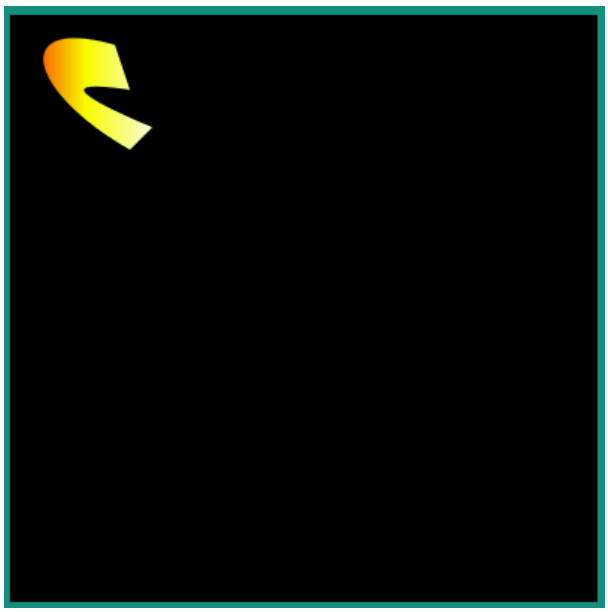


Fill Path With a Linear Gradient

Refer to *Linear Gradient* section for more details about the definition of a linear gradient.

The opacity value of the drawing is defined by the Alpha channel of the ARGB color values of the each linear gradient stop point.

```
LinearGradient gradient = new LinearGradient(0, 0, 100, 0, new int[] { 0xffff0000, 0xffffff00, 0xffffffff });
VectorGraphicsPainter.fillGradientPath(g, path, new Matrix(), gradient);
```



Fill Type

A path can be drawn with a *FillType* argument. This argument defines the way a path will be filled.

The following values are a available:

- FillType.Winding: Specifies that “inside” is computed by a non-zero sum of signed edge crossings.
- FillType.EVEN_ODD: Specifies that “inside” is computed by an odd number of edge crossings.

```
Path path = new Path();

path.moveTo(50, 0);
path.lineTo(21, 90);
path.lineTo(98, 35);
path.lineTo(2, 35);
path.lineTo(79, 90);
path.close();
```

Opacity and Blending Mode

The opacity of the drawing can be provided to the *fillPath* method with a blending mode.

When the drawing is done with graphic context color, the given alpha value replaces the default value (0xFF).

When the drawing is done with a linear gradient, the given alpha is applied above each gradient colors alpha channel values(0x80 alpha value on #80FFFFFF ARGB color leads to #40FFFFFF color).

The supported blending modes are:

- SRC : The source pixels replace the destination pixels.
- SRC_OVER : The source pixels are drawn over the destination pixels.
- DST_OVER : The source pixels are drawn behind the destination pixels.
- SRC_IN : Keeps the source pixels that cover the destination pixels, discards the remaining source and destination pixels.

- DST_IN : Keeps the destination pixels that cover source pixels, discards the remaining source and destination pixels.
- DST_OUT : Keeps the destination pixels that are not covered by source pixels. Discards destination pixels that are covered by source pixels. Discards all source pixels.
- PLUS : Adds the source pixels to the destination pixels and saturates the result.
- SCREEN : Adds the source and destination pixels, then subtracts the source pixels multiplied by the destination.
- MULTIPLY : Multiplies the source and destination pixels.

Fig. 137: SRC

Fig. 138: SRC_OVER

Fig. 139: DST_OVER

Fig. 140: SRC_IN

Fig. 141: DST_IN

Fig. 142: DST_OUT

Fig. 143: PLUS

Fig. 144: SCREEN

g. 145: MUL-IPLY

Matrix

A *Matrix* is composed of an array of numbers with three rows and three columns. It is used to apply an affine transformations to *Path* points. (Refer to https://en.wikipedia.org/wiki/Transformation_matrix#Affine_transformations to get more information about affine transformations).

The available transformations are:

- translation
- rotation
- scaling

Scaling and rotation are always performed around the (0,0) pivot point. In order to rotate or scale a *Path* with a pivot point, the matrix must be translated before and after the rotation/scaling.

A *Matrix* is created as an identity matrix, which means that a *Path* resulting of a transformation with this matrix is identical to the original *Path*.

The *Matrix* can be initialized with a transformation with set methods:

- *setTranslate(translateX, translateY)*
- *setRotate(angle)*
- *setScale(scaleX, scaleY)*

A transformation can be prepended to a *Matrix* with the prepend methods:

- *preTranslate(translateX, translateY)*
- *preRotate(angle)*
- *preScale(scaleX, scaleY)*

A transformation can be appended to a *Matrix* with the append methods:

- *postTranslate(translateX, translateY)*
- *postRotate(angle)*
- *postScale(scaleX, scaleY)*

A *Matrix* can also get transformations from an other *Matrix* with the concatenate and set methods:

- *preConcat(matrix)*
- *postConcat(matrix)*
- *set(matrix)*
- *setConcat(matrix0, matrix1)*

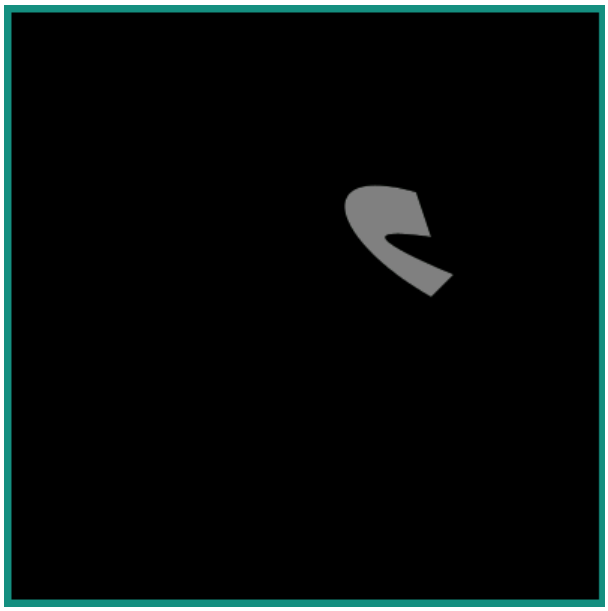
Once a *Matrix* has been computed, it can be used to draw an object (*Path*, *String*, *VectorImage*). All the points of the drawn object will be transformed by the *Matrix*.

When a *Matrix* has been computed with multiple type of transformation, the sequence order of the transformation is important. Chaining the transformations in a different order will not provide the same *Matrix*. The result of the previous transformation is the input to the next transformation.

The following examples use the *Path* created in the section *Path Creation* with different transformations.

Translation

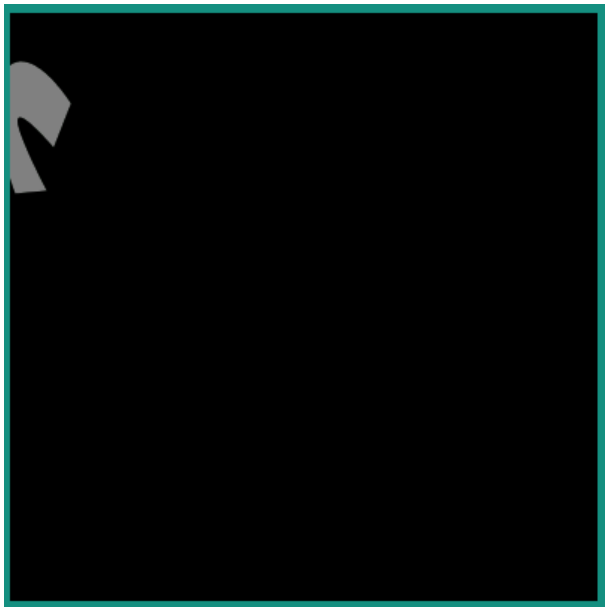
```
Matrix matrix = new Matrix();
matrix.setTranslate(200, 150);
```



Rotation

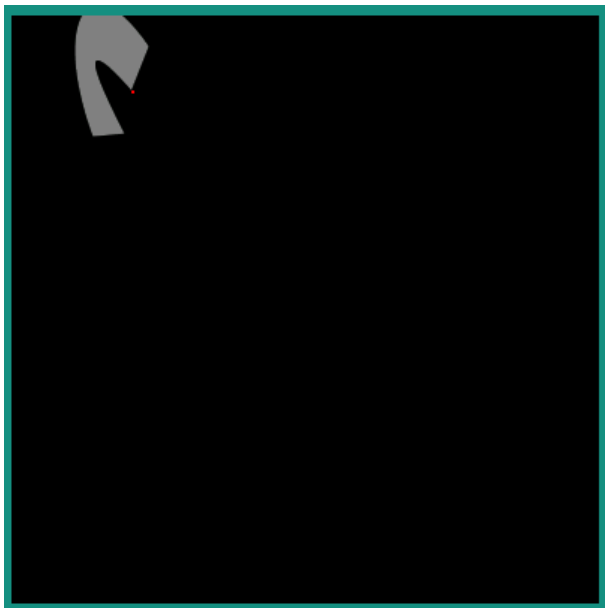
Around point (0,0).

```
Matrix matrix = new Matrix();
matrix.setRotate(40);
```



Around a pivot point (80,50).

```
Matrix matrix = new Matrix();
matrix.setRotate(40);

float pivotX = 80;
float pivotY = 50;
matrix.preTranslate(-pivotX, -pivotY);
matrix.postTranslate(pivotX, pivotY);
```



Scale

From point (0,0).

```
Matrix matrix = new Matrix();
matrix.setScale(2,3);
```

Concatenate Matrixes

Sequence order has an incidence on the rendering.

```
Matrix matrix0 = new Matrix();
matrix0.setScale(2, 3);

Matrix matrix1 = new Matrix();
matrix1.setTranslate(100, 40);

Matrix matrix2 = new Matrix();
matrix2.setConcat(matrix0, matrix1);

g.setColor(Colors.GRAY);
VectorGraphicsPainter.fillPath(g, path, matrix2);

matrix2.setConcat(matrix1, matrix0);

g.setColor(Colors.YELLOW);
VectorGraphicsPainter.fillPath(g, path, matrix2);
```

Linear Gradient

The MicroVG library supports the drawing of shapes with a linear gradient of color.

A linear gradient is specified by a linear segment and a set of ARGB colors associated with points on that segment.

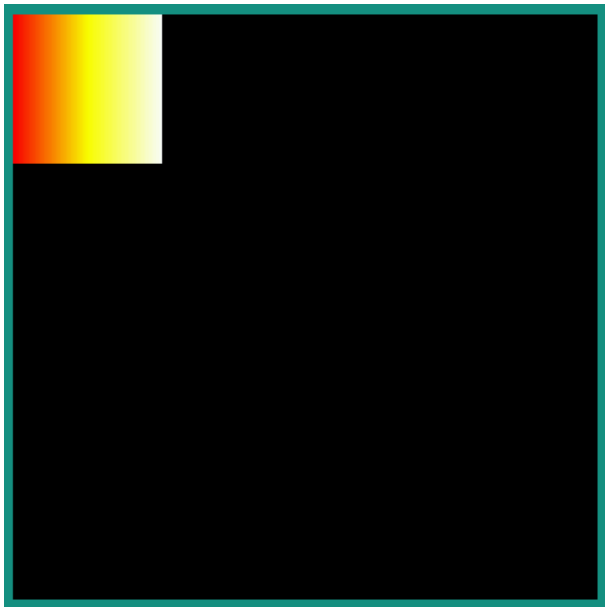
The colors along the segment between those points are calculated using linear interpolation, then extended perpendicular to that line.

The position of the color points on the segment are given from `0.0f` (start of point) to `1.0f` (end of the segment).

There are two ways to create a gradient:

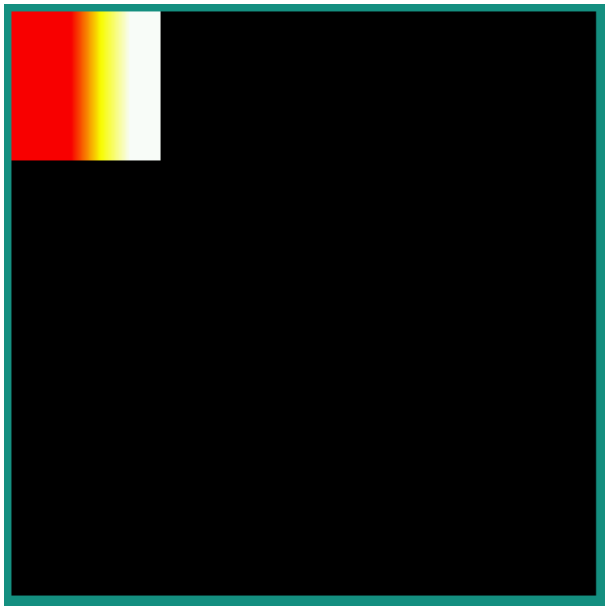
- with a start point, an end point and a color table: the first color will be applied to the start point, the second color to the end point and other colors distributed evenly along the gradient segment.

```
Path path = new Path();
path.moveTo(0, 0);
path.lineTo(100, 0);
path.lineTo(100, 100);
path.lineTo(0, 100);
path.close();

LinearGradient gradient = new LinearGradient(0, 0, 99, 0,
                                  new int[] { 0xffff0000, 0xffffff00, 0xffffffff });

VectorGraphicsPainter.fillGradientPath(g, path, new Matrix(), gradient);
```



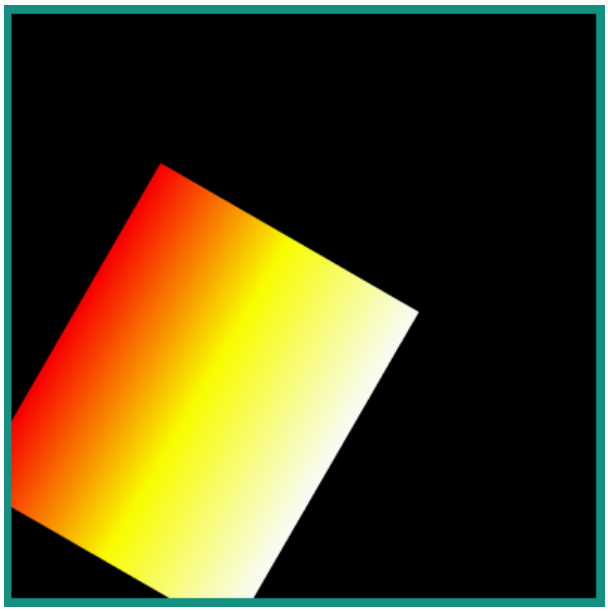
- with a start point, an end point, a color table and a position table: the colors are applied to their corresponding relative positions on the segment. If the first point is not the start point of the segment, then first color is applied from the start of the segment to the first point. If the last point is not the end point of the segment, then last color is applied from the last point to the end of the segment.

```
LinearGradient gradient = new LinearGradient(0, 0, 99, 0,
                                    new int[] { 0xffff0000, 0xffffff00, 0xffffffff },
                                    new float[] { 0.4f, 0.6f, 0.8f });

VectorGraphicsPainter.fillGradientPath(g, path, new Matrix(), gradient);
```



The transformation applied to the object (*Path* or *String*) to draw with a gradient is also applied to that gradient. The *LinearGradient* is not updated after the drawing.

```
LinearGradient gradient = new LinearGradient(0, 0, 99, 0,
                                    new int[] { 0xffff0000, 0xffffff00, 0xffffffff });

Matrix matrix = new Matrix();
matrix.setScale(2, 2.5f);
matrix.postRotate(30);
matrix.postTranslate(100, 100);

VectorGraphicsPainter.fillGradientPath(g, path, matrix, gradient);
```



Vector Fonts

Overview

The MicroVG library enables the usage of Vector Fonts.

Compared to MicroUI *Fonts*, Vector Fonts brings the following features:

- the text strings are scalable and can be transformed using a Matrix object.
- the TTF/OTF font files don't need to be preprocessed.
- the text strings can be drawn with opacity, a color or a linear gradient.

The library also considers the *Kerning* space described in the font file kerning table, and allows a fine adjustement of the inter-letters spacing.

It also provides metrics measurement methods to correctly place the text within the surrounding drawing elements (i.e. in a label).

Loading a Font File

A Vector Font has to be loaded in a VectorFont object with a call to ej.microvg.VectorFont.loadFont(). This VectorFont object can then be used to draw text strings.

The fonts are decoded at runtime. They don't need to be pre-processed by some generator tool like *MicroUI Fonts* Vector Font files must be declared as resources in a `.resources.list` file available in the classpath (*Application Resources*). To declare them as *external resources*, the font files must be declared too in a `.externresources.list` file.

Text String Drawing

A string can be drawn in the graphics context with a call to ej.microvg.VectorGraphicsPainter.drawString().

The text string height is scalable, and multiple font files can be used in parrallel.

```
VectorFont font0 = VectorFont.loadFont("/fonts/Arial.ttf");
VectorFont font1 = VectorFont.loadFont("/fonts/RAVIE.ttf");

int x = 20;
int y = 30;
int yOffset = 150;

g.setColor(Colors.LIME);
VectorGraphicsPainter.drawString(g, "Hello MicroEJ", font0, 20, x, y);
VectorGraphicsPainter.drawString(g, "Hello MicroEJ", font1, 20, x, y + yOffset);

g.setColor(Colors.RED);
y += 20;
VectorGraphicsPainter.drawString(g, "Hello MicroEJ", font0, 30, x, y);
VectorGraphicsPainter.drawString(g, "Hello MicroEJ", font1, 30, x, y + yOffset);

g.setColor(Colors.WHITE);
y += 30;
VectorGraphicsPainter.drawString(g, "Hello MicroEJ", font0, 40, x, y);
VectorGraphicsPainter.drawString(g, "Hello MicroEJ", font1, 40, x, y + yOffset);

g.setColor(Colors.YELLOW);
y += 40;
VectorGraphicsPainter.drawString(g, "Hello MicroEJ", font0, 50, x, y);
VectorGraphicsPainter.drawString(g, "Hello MicroEJ", font1, 50, x, y + yOffset);

display.flush();
```

Text Color

The text string can be colored with the graphics context color or a with a linear gradient(*Linear Gradient*).

FillType and *Alpha Blending Mode* are also managed similarly to *Path* drawing (refer to *Fill Type* and *Opacity and Blending Mode*).

```
g.setColor(Colors.LIME);
VectorGraphicsPainter.drawString(g, "Hello MicroEJ", font, 50, x, y);

LinearGradient gradient = new LinearGradient(0, 0, 250, 50,
                          new int[] { 0xffff0000, 0xffffff00, 0xffffffff });

Matrix matrix = new Matrix();
matrix.setTranslate(x, y + 60);
VectorGraphicsPainter.drawGradientString(g, "Hello MicroEJ", font, 50, matrix, gradient, 0xff,
              BlendMode.SRC_OVER, 0);
```

Text Transformations

The text string can also be transformed with a Matrix to translate, rotate, scale the drawing.

```
Matrix matrix0 = new Matrix();

matrix0.setTranslate(20, 60);
VectorGraphicsPainter.
↪drawString(g, "Hello MicroEJ", font, 50, matrix0, 0xff, BlendMode.SRC_OVER, 0);

matrix0.preRotate(180);
matrix0.postTranslate(300, 120);
VectorGraphicsPainter.
↪drawString(g, "Hello MicroEJ", font, 50, matrix0, 0xff, BlendMode.SRC_OVER, 0);

Matrix matrix1 = new Matrix();
matrix1.setScale(0.5f, 1.2f);
matrix1.postRotate(45);
matrix1.postTranslate(80, 200);

VectorGraphicsPainter.
↪drawString(g, "Hello MicroEJ", font, 50, matrix1, 0xff, BlendMode.SRC_OVER, 0);

matrix1.setScale(0.5f, 1.2f);
matrix1.postRotate(-45);
matrix1.postTranslate(200, 300);
VectorGraphicsPainter.
↪drawString(g, "Hello MicroEJ", font, 50, matrix1, 0xff, BlendMode.SRC_OVER, 0);
```

Letter Spacing

The inter character distance can be adjusted for each string drawing. By default, the inter character distance is computed from the font file metrics, considering kerning, if the font file includes a kerning table. It can be adjusted with the letterSpacing parameter of drawString(). Its default value is 0 pixel, a positive/negative value will increase/reduce the inter space distance by the corresponding pixel value.

```
Matrix matrix = new Matrix();

matrix.setTranslate(20, 60);
VectorGraphicsPainter.
↪drawString(g, "Hello MicroEJ", font, 50, matrix, 0xff, BlendMode.SRC_OVER, 0);

matrix.postTranslate(0, 60);
VectorGraphicsPainter.
↪drawString(g, "Hello MicroEJ", font, 50, matrix, 0xff, BlendMode.SRC_OVER, 5f);

matrix.postTranslate(0, 60);
VectorGraphicsPainter.
↪drawString(g, "Hello MicroEJ", font, 50, matrix, 0xff, BlendMode.SRC_OVER, -2);
```

Colored Emojis

The library supports the drawing of colored multilayer glyphs, but only for the embedded implementation. The simulator implementation draws the full emoji glyph with the color of the graphics context.

Only font files with CPAL/COLR tables are supported.

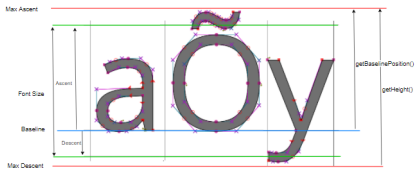
Font files with CBDT/CBLC tables are not supported.

To add colored emojis to a font, see the tutorial *How to Add Emojis to a Vector Font*.

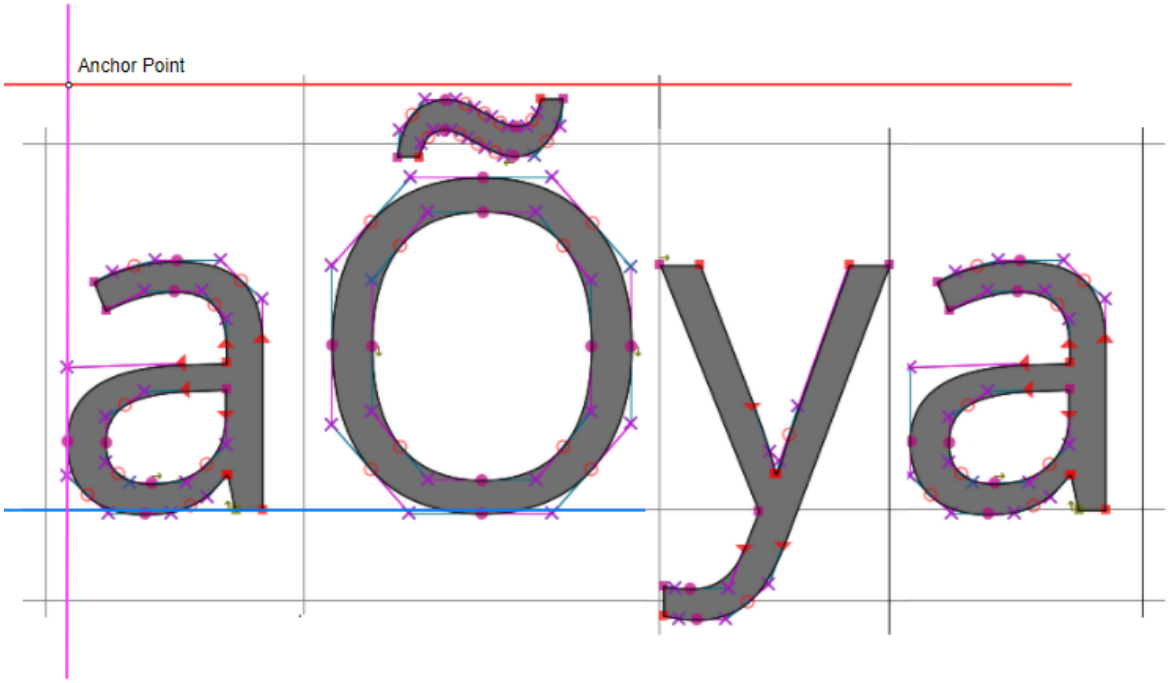
Metrics and Text Positioning

All metrics provided by the ej.microvg.VectorFont class are given for a specific font size. The font size defines the height to which each character bounding box will be scaled.

The following figure presents some concepts of font metrics standarts:



When a string is drawn with a call to ej.microvg.VectorGraphicsPainter.drawString() or ej.microvg.VectorGraphicsPainter.drawGradientString(), the anchor point of the string is the top left corner of the text rendering box. This anchor point is located horizontally on the first pixel of the first drawn glyph and vertically on the max ascent line.



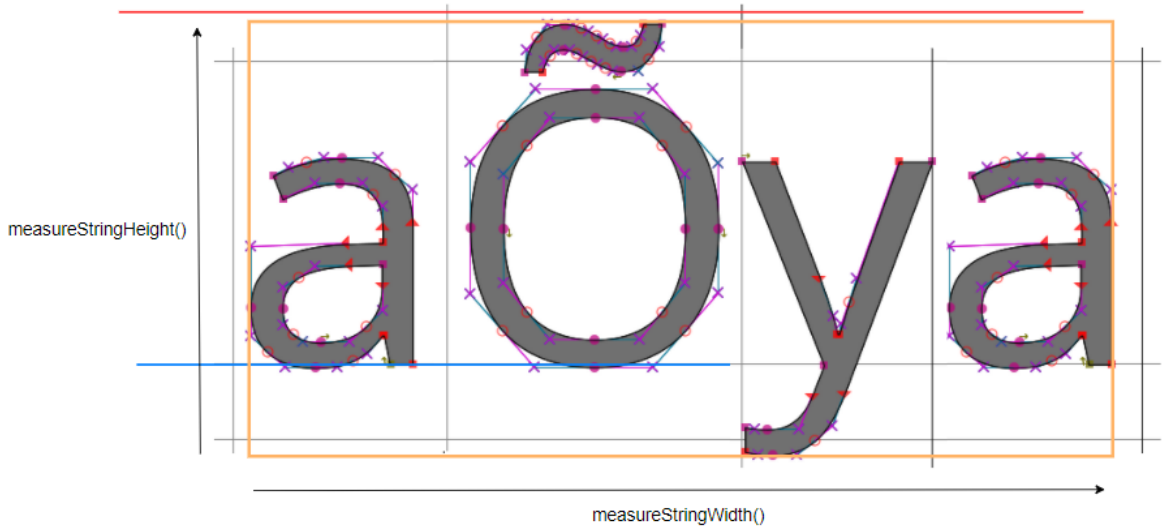
The ej.microvg.VectorFont.getBaselinePosition() method can be used to position the text baseline on a horizontal line.

The ej.microvg.VectorFont.getHeight() method can be used to center a text inside a label, by positionning the anchor point in order to have the same space above and below the text string.

Two other methods are available to position a known text in a label:

- ej.microvg.VectorFont.measureStringHeight()
- ej.microvg.VectorFont.measureStringWidth()

These methods return the width and height of a string drawing. They are computed from the width and height of the glyphs composing the string.



These methods can measure a specific glyph width and height using a one character string.

Note: The metrics are extracted from the character glyph metrics without considering the antialiasing introduced by the glyphs rasterizer.

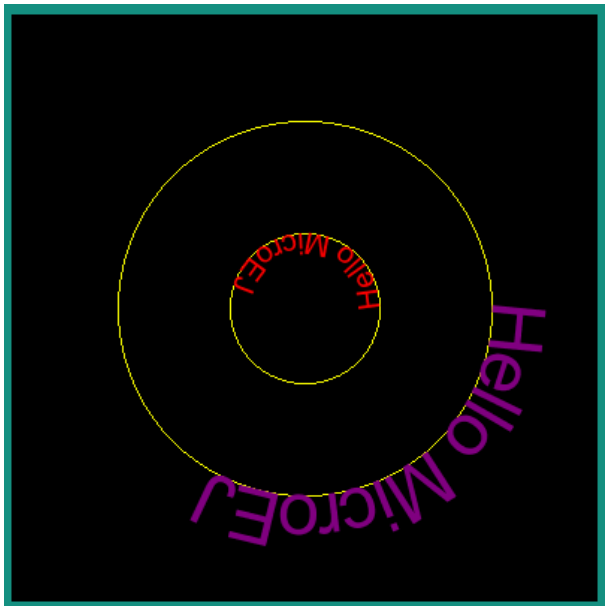
Drawing a Text on a Circle

The library proposes the drawing of a text on a circle by a call to ej.microvg.VectorGraphicsPainter.drawStringOnCircle(). The string is rendered as if the baseline of the string was a circle arc.

The string direction can be either clockwise or counter clockwise.

All the features described above are still available (linear gradient, transformations, letter spacing, kerning, colored emojis).

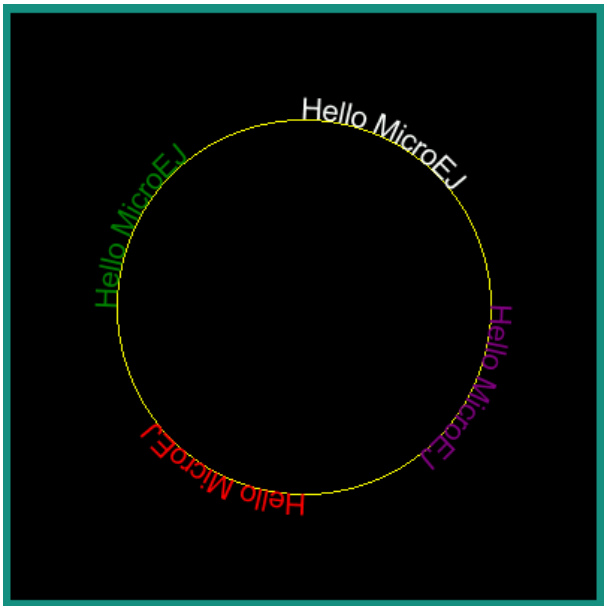
```
int x = 196;
int y = 196;
int diameter = 250;

g.setColor(Colors.YELLOW);

Painter.drawCircle(g, x - diameter / 2, y - diameter / 2, diameter);

g.setColor(Colors.PURPLE);
Matrix matrix = new Matrix();

matrix.setTranslate(x, y);

VectorGraphicsPainter.drawStringOnCircle(g, "Hello MicroEJ", font, 50, matrix, diameter / 2,
        Direction.CLOCKWISE);

diameter = 100;

g.setColor(Colors.YELLOW);
Painter.drawCircle(g, x - diameter / 2, y - diameter / 2, diameter);

g.setColor(Colors.RED);
VectorGraphicsPainter.drawStringOnCircle(g, "Hello MicroEJ", font, 20, matrix, diameter / 2,
        Direction.COUNTER_CLOCKWISE);
```



The anchor point of the drawing is the center of the circle.

The position where the text starts along the circle is the 3 o'clock position (positive X axis). This starting position can be modified by specifying a rotation into the transformation *Matrix*.

```
g.setColor(Colors.PURPLE);
Matrix matrix = new Matrix();

matrix.setTranslate(x, y);

VectorGraphicsPainter.drawStringOnCircle(g, "Hello MicroEJ", font, 20, matrix, diameter / 2,
        Direction.CLOCKWISE);

matrix.preRotate(90);
g.setColor(Colors.RED);
VectorGraphicsPainter.drawStringOnCircle(g, "Hello MicroEJ", font, 20, matrix, diameter / 2,
        Direction.CLOCKWISE);

matrix.preRotate(90);
g.setColor(Colors.GREEN);
VectorGraphicsPainter.drawStringOnCircle(g, "Hello MicroEJ", font, 20, matrix, diameter / 2,
        Direction.CLOCKWISE);

matrix.preRotate(90);
g.setColor(Colors.WHITE);
VectorGraphicsPainter.drawStringOnCircle(g, "Hello MicroEJ", font, 20, matrix, diameter / 2,
        Direction.CLOCKWISE);
```



Complex Text Layout

Some scripts like Arabic or Thai scripts request a specific text layout mode where the shape or positioning of a grapheme depends on its relation to other graphemes (Refer to https://en.wikipedia.org/wiki/Complex_text_layout).

The MicroVG library provides two different layout modes:

- the simple layout mode for latin scripts and other scripts where character unicodes and glyphs are one-to-one associated.
- the complex layout mode for complex text layout scripts like arabic or thai.

The simple layout mode draws the text character as described in the previous sections. It uses the font **Kerning** table and the glyphs advanceX parameter to position the glyphs one after the other.

The complex layout mode uses the **GPOS** and **GSUB** font tables to substitute and position the character glyph.

The complex layout mode can be selected while loading the glyph with *ej.microvg.VectorFont.loadFont* by passing a supplementary boolean argument with value *true*.

Next example shows the same arabic string drawn with the same font but with simple (in white) and complex layout(in RED).

```
VectorFont font0 = VectorFont.loadFont(FONT_NAME, false);
VectorFont font1 = VectorFont.loadFont(FONT_NAME, true);

String s = "      ";

g.setColor(Colors.WHITE);
VectorGraphicsPainter.drawString(g, s, font0, 20, 50, 50);

g.setColor(Colors.RED);
VectorGraphicsPainter.drawString(g, s, font1, 20, 50, 100);
```

Text Measurement and Positioning

The measurement of string in complex layout mode respects the requirements presented in *Metrics and Text Positioning*.

Strings from script where text is read from right to left, like arabic, are still drawn with the anchor point located on the top left of the string.

Bidirectional Text

The complex layout mode does not support bidirectional text. A bidirectional text has to be splitted in multiple strings and each string has to be drawn to the correct location.

Limitations

The simulator rendering of complex layout mode for *Drawing a Text on a Circle* feature is done with many approximations. This rendering can still be used to have an overview of the text positionning on the display.

The letterSpacing feature is not supported by the simulator implementation. Texts will be displayed with a letterspacing value of 0.

External Fonts

To fetch fonts from external memory, the application must pre-register the *external Font resources*. The management of this kind of font may be different than the *internal* images and may require some allocations in the runtime memory. For more details about the external font management, refers to the VEE Port Guide chapter *External Memory*.

Vector Images

Overview

Vector Images are graphical resources that can be accessed with a call to ej.microvg.VectorImage.getImage(). The images are converted at build-time (using the image generator tool) to binary resources.

Images that must be processed by the image generator tool are declared in `*.vectorimages.list` files (**or** in `*.externvectorimages.list` for an external resource, see *External Images*). The file format is a standard Java properties file, each line representing a `/` separated resource path relative to the MicroEJ classpath root referring to a vector image file (e.g. `.svg`, `.xml`). The resource must be followed by a parameter (separated by a `:`) which defines and/or describes the image output file format (raw format).

Currently accepted formats are :

- `:VGF` : MicroVG compatible format with coordinates encoded as float numbers (32 bits).
- `:VG32` : MicroVG compatible format with coordinates encoded as signed int numbers (32 bits).
- `:VG16` : MicroVG compatible format with coordinates encoded as signed short numbers (16 bits).
- `:VG8` : MicroVG compatible format with coordinates encoded as signed char numbers (8 bits).

Note: The ouput format may be adjusted by the MicroVG engine to fit the capabilities of the vectorial GPU.

Example:

```
/com/mycompany/MyImage1.svg:VGF
/com/mycompany/androidVectorDrawable.xml:VG8
```

Warning: In the case where the output format is not specified, the resource is embedded as is, as described in the Image Generator chapter *Unspecified Output Format*. This use case is useful for loading an encoded VG image, as described in the *Android Vector Drawable Loader* chapter.

Supported Input Files

The image generator tool supports the following input file formats:

- Android Vector Drawable
- SVG

Refer to the *Limitations / Supported Features* section for the list of supported features for these file formats.

The vector image objects are extracted and converted to paths made of Move, Line and Curve commands.

Each path is associated with either a fill color or a linear gradient. All object strokes are converted to filled paths at build-time.

Objects group transformations are also extracted from the input file and applied at run-time.

Drawing Images

Drawing and Transforming Images

Once an image has been loaded it can be drawn in the graphic context with a call to ej.microvg.VectorGraphicsPainter.drawImage().

The image is associated with a transformation *Matrix* that will be applied in order to translate, scale and/or rotate the image.

The application can get the width and the height of the image with ej.microvg.VectorImage.getWidth() and ej.microvg.VectorImage.getHeight() to correctly scale and position the image in the application window.

The following example describes how an Android Vector Drawable file can be drawn and positioned on the display.

- Android Vector Drawable file:

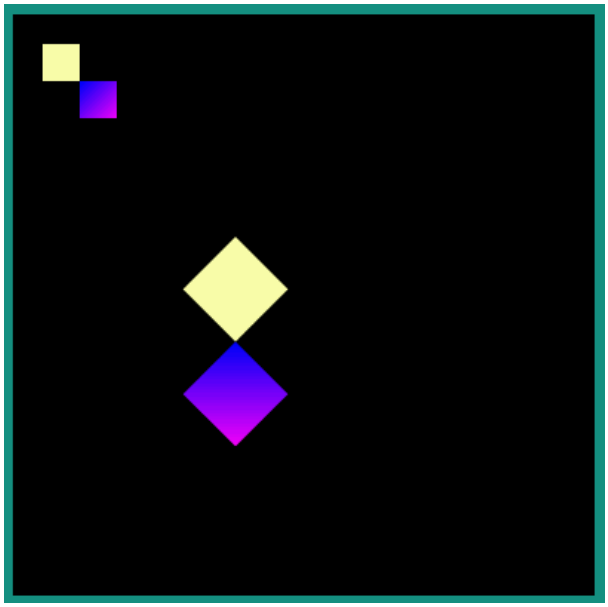
```
<vector xmlns:android="http:/
↪/schemas.android.com/apk/res/android" xmlns:aapt="http://schemas.android.com/aapt"
  android:width="100dp
↪" android:height="100dp" android:viewportWidth="100" android:viewportHeight="100">
  <path android:pathData="M 0 0 h50 v50 h-50 z" android:fillColor="#FFFFAA"/>
  <path android:pathData="M 50 50 h50 v50 h-50 z">
    <aapt:attr name="android:fillColor">
        <gradient
          android:startColor="#0000ff" android:startX="50" android:startY="50"
          android:endColor="#ff00ff" android:endX="100" android:endY="100"
          android:type="linear">
        </gradient>
    </aapt:attr>
  </path>
</vector>
```

```
public static void main(String[] args) {

  MicroUI.start();
```

(continues on next page)

(continued from previous page)

```
    Display display = Display.getDisplay();
    GraphicsContext g = display.getGraphicsContext();

    VectorImage image = VectorImage.getImage("/images/myImage.xml"); //$NON-NLS-1$

    Matrix matrix0 = new Matrix();
    matrix0.setTranslate(20, 20);
    matrix0.preScale(50 / image.getWidth(), 50 / image.getHeight());

    Matrix matrix1 = new Matrix();
    matrix1.setTranslate(150, 150);
    matrix1.preRotate(45);

    VectorGraphicsPainter.drawImage(g, image, matrix0);
    VectorGraphicsPainter.drawImage(g, image, matrix1);

    display.flush();
}
```



Drawing With Opacity

The vector image can be drawn with a global opacity level.

```
VectorImage image = VectorImage.getImage("/images/myImage.xml"); //$NON-NLS-1$

// the global opacity rendering value, between 0 (transparent) and 255 (opaque)
int opacity = 0x80;

VectorGraphicsPainter.drawImage(g, image, new Matrix(), opacity);
```

Warning: As paths are drawn one after the other, images that contain overlapping paths are not correctly colored when a global opacity is applied. The rendering of these images will throw an exception. The images must be reworked to suppress overlapping.

Color Filtering

A VectorImage object can be derived from another VectorImage object, keeping the paths and transformations but updating the colors using a color matrix.

This color matrix is a 4x5 float matrix. It is organized like that:

- Each line is used to compute a component of the resulting color, in this order: red, green, blue, alpha.
- The four first columns are multipliers applied to a component of the initial color, in this order: red, green, blue, alpha.
- The last column is a constant value.

Let A, R, G, B be the components of the initial color and the following array a color matrix:

```
{ rR, rG, rB, rA, rC,  // red
  gR, gG, gB, gA, gC,  // green
  bR, bG, bB, bA, bC,  // blue
  aR, aG, aB, aA, aC } // alpha
```

The resulting color components are computed as:

```
resultRed = rR * R + rG * G + rB * B + rA * A + rC
resultGreen = gR * R + gG * G + gB * B + gA * A + gC
resultBlue = bR * R + bG * G + bB * B + bA * A + bC
resultAlpha = aR * R + aG * G + aB * B + aA * A + aC
```

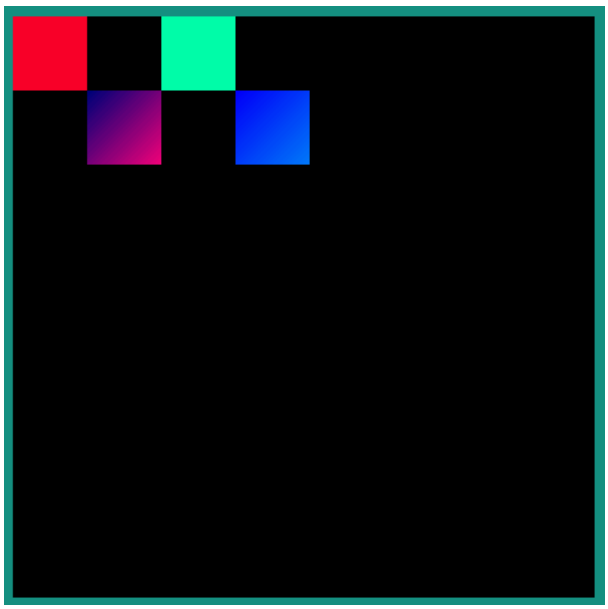
If the resulting component value is below 0 or above 255, the component value is clamped to these limits.

Note: The new image is a ResourceVectorImage. The image buffer is allocated in the MicroUI image heap. The application must manage the image cycle life and close the image to free the image buffer.

A VectorImage object can also be drawn associated to a color matrix by a call to ej.microvg.VectorGraphicsPainter.drawFilteredImage().

The following example illustrates this feature.

```
VectorImage image = VectorImage.getImage("/images/myImage.xml"); //$NON-NLS-1$

// Derive a new VectorImage
float[] colorMatrix0 = new float[] { //
      1f, 0, 0, 0, 0, // red
      0, 0, 0, 0, 0, // green
      0, 0, 1f, 0, 0, // blue
      0, 0, 0, 1f, 0, // alpha
};

VectorImage imageFiltered = image.filterImage(colorMatrix0);
VectorGraphicsPainter.drawImage(g, imageFiltered, new Matrix());

float[] colorMatrix1 = new float[] { //
      0f, 0, 0, 0, 0, // red
      0.5f, 0.5f, 0, 0, 0, // green
      0, 0, 1f, -0.5f, 0, // blue
      0, 0, 0, 1f, 0, // alpha
};
Matrix matrix1 = new Matrix();
matrix1.setTranslate(image.getWidth(), 0);

VectorGraphicsPainter.drawFilteredImage(g, image, matrix1, colorMatrix1);

imageFiltered.close();
```



Animated Vector Images

The Android Vector Drawable format provides the ability to change the properties of vector graphics over time, in order to create animated effects.

The transformations of the objects over the time are embedded in the Vector image file and a call to ej.microvg.VectorGraphicsPainter.drawAnimatedImage() or ej.microvg.VectorGraphicsPainter.drawFilteredAnimatedImage() will draw the image for a specific time frame.

The application can get the duration of the image animation with a call to ej.microvg.VectorImage.getDuration().

Every image object that is animated outside the image viewbox is clipped at the image boundary. In any cases, especially when the image is rotated, the image boundary is the rectangle that contains all the corners of the original image.

The supported file format is an Animated Vector Drawable xml file with animations and vector definition in the same file as described in Android API.

The SVG format also supports the animation of vector graphics objects, but this feature is not yet implemented in the MicroVG library for this file format.

SVG files that need to be animated should be converted to Android Vector Drawable format with the Android Vector Asset tool and then animated manually or with a tool like Shapeshifter.

Warning: A flaw in Eclipse Temurin™ JDK 8 causes animated vector images to render incorrectly on the Simulator. You should upgrade to Eclipse Temurin™ JDK 11 instead.

Supported animations

This section will present the different available animations with an example.

For each example, this simple java code will be used.

```
VectorImage image = VectorImage.getImage("/images/myImage.xml"); //$NON-NLS-1$
Matrix matrix = new Matrix();
matrix.setTranslate(100,100);
matrix.preScale(2,2);

long elapsed = 0;
long step = 10;
while (true) {
  // Clear Screen
  g.setColor(Colors.BLACK);
  Painter.fillRectangle(g, 0, 0, display.getWidth(), display.getHeight());

  VectorGraphicsPainter.drawAnimatedImage(g, image, matrix, elapsed);

  display.flush();

  // Pause the current thread
  try {
    Thread.sleep(step);
  } catch (InterruptedException e) {
    e.printStackTrace();
  }
```

(continues on next page)

(continued from previous page)

```
  // Update current image time
  if (elapsed < image.getDuration()) {
    elapsed += step;
  } else {
    elapsed = 0;
  }
}
```

TranslateX and TranslateY

Any group in the Android Vector Drawable can be translated in X or Y direction with an object animator.

```
<animated-vector xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:aapt="http://schemas.android.com/aapt">
<aapt:attr name="android:drawable">
  <vector android:width="100dp" android:height="100dp"
        android:viewportWidth="100" android:viewportHeight="100">
    <group android:name="yellow_group">
        <path android:pathData="M 0 0 h50 v50 h-50 z" android:fillColor="#FFFFAA"/>
    </group>
    <group android:name="gradient_group">
    <path android:pathData="M 50 50 h50 v50 h-50 z">
        <aapt:attr name="android:fillColor">
          <gradient
            android:startColor="#0000ff" android:startX="50" android:startY="50"
            android:endColor="#ff00ff" android:endX="100" android:endY="100"
            android:type="linear">
          </gradient>
        </aapt:attr>
    </path>
    </group>
  </vector>
</aapt:attr>
<target android:name="yellow_group">
  <aapt:attr name="android:animation">
    <set android:ordering="together">
  ␣
↪      <objectAnimator android:propertyName="translateX" android:valueType="floatType"
          android:duration=
↪"1000" android:startOffset="0" android:valueFrom="0" android:valueTo="50"/>
  ␣
↪      <objectAnimator android:propertyName="translateX" android:valueType="floatType"
          android:duration=
↪"1000" android:startOffset="1500" android:valueFrom="50" android:valueTo="0"/>
    </set>
  </aapt:attr>
</target>
<target android:name="gradient_group">
  <aapt:attr name="android:animation">
    <set android:ordering="together">
  ␣
↪      <objectAnimator android:propertyName="translateX" android:valueType="floatType"
          android:duration=
```

(continues on next page)

(continued from previous page)

```
↪"1000" android:startOffset="0" android:valueFrom="0" android:valueTo="-50"/>
  ␣
↪      <objectAnimator android:propertyName="translateX" android:valueType="floatType"
          android:duration=
↪"1000" android:startOffset="1500" android:valueFrom="-50" android:valueTo="0"/>
  ␣
↪      <objectAnimator android:propertyName="translateY" android:valueType="floatType"
          android:duration=
↪"1000" android:startOffset="0" android:valueFrom="0" android:valueTo="-50"/>
  ␣
↪      <objectAnimator android:propertyName="translateY" android:valueType="floatType"
          android:duration=
↪"1000" android:startOffset="1500" android:valueFrom="-50" android:valueTo="0"/>
    </set>
  </aapt:attr>
</target>
</animated-vector>
```

TranslateXY over a path

Any group in the Android Vector Drawable can be translated over a path.

```
<animated-vector xmlns:android="http://schemas.android.com/apk/res/android"
              xmlns:aapt="http://schemas.android.com/aapt">
<aapt:attr name="android:drawable">
  <vector android:width="100dp" android:height="100dp"
        android:viewportWidth="100" android:viewportHeight="100">
    ... same as previous example
  </vector>
</aapt:attr>
<target android:name="gradient_group">
  <aapt:attr name="android:animation">
    <set android:ordering="together">
        <objectAnimator
            android:propertyName="translateXY" android:duration="5000"
            android:propertyXName="translateX" android:propertyYName="translateY"
            android:pathData="M -0.143 0.479 C -30.355 28.02 -153.405 -111.8 -39.441 -70.818
                      C -48.423 -63.52 70.593 -18.608 -91.09 -15.802 Z"/>
    </set>
  </aapt:attr>
</target>
</animated-vector>
```

ScaleX and ScaleY

A group in the Android Vector Drawable can be scaled on X or Y direction. The scaling pivot point is the one defined in the group attributes. By default, the pivot point is (0,0).

```
<animated-vector xmlns:android="http://schemas.android.com/apk/res/android"
                 xmlns:aapt="http://schemas.android.com/aapt">
<aapt:attr name="android:drawable">
  <vector android:width="100dp" android:height="100dp"
          android:viewportWidth="100" android:viewportHeight="100">
    <group android:name="yellow_group" android:pivotX="25" android:pivotY="25">
          <path android:pathData="M 0 0 h50 v50 h-50 z" android:fillColor="#FFFFAA"/>
    </group>
    <group android:name="gradient_group" >
    <path android:pathData="M 50 50 h50 v50 h-50 z">
          <aapt:attr name="android:fillColor">
            <gradient
              android:startColor="#0000ff" android:startX="50" android:startY="50"
              android:endColor="#ff00ff" android:endX="100" android:endY="100"
              android:type="linear">
            </gradient>
          </aapt:attr>
    </path>
    </group>
  </vector>
</aapt:attr>
<target android:name="yellow_group">
  <aapt:attr name="android:animation">
    <set android:ordering="together">
          <objectAnimator android:propertyName="scaleX" android:valueType="floatType"
            android:duration="1000" android:startOffset="0" android:valueFrom="1"
            android:valueTo="0.5"/>
          <objectAnimator android:propertyName="scaleX" android:valueType="floatType"
            android:duration="1000" android:startOffset="1500" android:valueFrom="0.5"
            android:valueTo="1"/>
    </set>
  </aapt:attr>
</target>
<target android:name="gradient_group">
  <aapt:attr name="android:animation">
    <set android:ordering="together">
          <objectAnimator android:propertyName="scaleX" android:valueType="floatType"
            android:duration="1000" android:startOffset="0"
            android:valueFrom="0.2" android:valueTo="1"/>
          <objectAnimator android:propertyName="scaleX" android:valueType="floatType"
            android:duration="1000" android:startOffset="1500"
            android:valueFrom="1" android:valueTo="0.2"/>
          <objectAnimator android:propertyName="scaleY" android:valueType="floatType"
            android:duration="1000" android:startOffset="0"
            android:valueFrom="0.2" android:valueTo="1"/>
          <objectAnimator android:propertyName="scaleY" android:valueType="floatType"
            android:duration="1000" android:startOffset="1500"
            android:valueFrom="1" android:valueTo="0.2"/>
    </set>
  </aapt:attr>
</target>
</animated-vector>
```

Rotate

A group in the Android Vector Drawable can be rotated around a pivot point. The pivot point is the one defined in the group attributes. By default, the pivot point is (0,0).

```
<animated-vector xmlns:android="http://schemas.android.com/apk/res/android"
                 xmlns:aapt="http://schemas.android.com/aapt">
<aapt:attr name="android:drawable">
  <vector android:width="100dp" android:height="100dp"
          android:viewportWidth="100" android:viewportHeight="100">
    ... same as previous example
  </vector>
</aapt:attr>
<target android:name="yellow_group">
  <aapt:attr name="android:animation">
    <set android:ordering="together">
          <objectAnimator android:propertyName="rotation" android:valueType="floatType"
            android:duration="1000" android:startOffset="0"
            android:valueFrom="0" android:valueTo="720"/>
          <objectAnimator android:propertyName="rotation" android:valueType="floatType"
            android:duration="1000" android:startOffset="1500"
            android:valueFrom="720" android:valueTo="0"/>
    </set>
  </aapt:attr>
</target>
</animated-vector>
```

Morphing

The Android Vector Drawable format supports the animation of the `pathData` attribute of a path. With this type of animation a shape can be transformed to a totally different other shape. The only constraint is that the origin and destination `pathData` must have the same commands format.

Lets take, for instance, the morphing of a rectangle to a circle which have the following commands.

```
Circle: M 11.9 9.8 C 11.9 8.1 13.3␣
↪6.7 14.9 6.7 C 16.6 6.7 18 8.1 18 9.8 C 18 11.6 16.6 13 14.9 13 C 13.3 13 11.9 11.6 11.9 9.8 Z

Rectangle: M 11.9 6.7 H 18 V 13 H 11.9 Z
```

The rectangle path has to be reworked to match with the sequence of commands of the circle path.

The following tools can be used to manipulate the paths to create the wanted animation effect:

- Shapeshifter
- SVGPathEditor

There is an infinity of possibilities to create the new path, and the association of each points of the paths will induce a specific morphing animation. As an example, let's define two rectangles very similar visually but with different definitions:

```
New Rectangle␣
↪path1: M 11.9 9.8 C 11.897 7.735 11.906 7.995 11.906 6.697 C 16.6 6.7 16.601 6.706␣
↪17.995 6.697 C 18 11.6 17.995 11.587 18.004 13.006 C 13.3 13 13.852 13.006 11.897 13.006 Z

New Rectangle path2:␣
↪M 11.906 6.697 C 11.953 6.698 12.993 6.698 17.995 6.697 C 17.999 8.331 17.997 9.93 18.002␣
↪13.004 C 16.239 13.007 16.009 13.001 11.893 13.007 C 13.3 13 13.852 13.006 11.893 13.007 Z
```

```
<animated-vector xmlns:android="http://schemas.android.com/apk/res/android"
                 xmlns:aapt="http://schemas.android.com/aapt">
 <aapt:attr name="android:drawable">
    <vector android:width="20dp" android:height="20dp"
            android:viewportWidth="20" android:viewportHeight="20">
       <path android:fillColor="#FF0000" android:pathData="M 0 0 h40 v40 h-40"/>
       <path android:fillColor="#FF0000" android:pathData="M 0 0 h40 v40 h-40"/>
       <group android:name="group1" android:translateX="-10">
          <path
             android:name="circle1"
             android:pathData="M 11.9 9.8 C 11.9 8.1 13.3 6.7 14.9 6.7
                               C 16.6 6.7 18 8.1 18 9.8
                               C 18 11.6 16.6 13 14.9 13
                               C 13.3 13 11.9 11.6 11.9 9.8 Z"
             android:fillColor="#FFFFAA"/>
       </group>
       <group android:name="group2">
          <path android:name="circle2"
             android:pathData="M 11.9 9.8 C 11.9 8.1 13.3 6.7 14.9 6.7
                               C 16.6 6.7 18 8.1 18 9.8
                               C 18 11.6 16.6 13 14.9 13
                               C 13.3 13 11.9 11.6 11.9 9.8 Z"
             android:fillColor="#00FFAA" />
       </group>
    </vector>
 </aapt:attr>

 <target android:name="circle1">
 <aapt:attr name="android:animation">
    <set>
       <objectAnimator
          android:propertyName="pathData"
          android:duration="2000"
          android:valueFrom="M 11.9 9.8 C 11.9 8.1 13.3 6.7 14.9 6.7
                             C 16.6 6.7 18 8.1 18 9.8
                             C 18 11.6 16.6 13 14.9 13
                             C 13.3 13 11.9 11.6 11.9 9.8 Z"
          android:valueTo="M 11.9 9.8 C 11.897 7.735 11.906 7.995 11.906 6.697
                           C 16.6 6.7 16.601 6.706 17.995 6.697
                           C 18 11.6 17.995 11.587 18.004 13.006
                           C 13.3 13 13.852 13.006 11.897 13.006 Z"
          android:valueType="pathType"/>
    </set>
 </aapt:attr>
 </target>
```

(continues on next page)

(continued from previous page)

```
<target android:name="circle2">
   <aapt:attr name="android:animation">
      <set>
         <objectAnimator
            android:propertyName="pathData"
            android:duration="2000"
            android:valueFrom="M 11.9 9.8 C 11.9 8.1 13.3 6.7 14.9 6.7
                               C 16.6 6.7 18 8.1 18 9.8
                               C 18 11.6 16.6 13 14.9 13
                               C 13.3 13 11.9 11.6 11.9 9.8 Z"
            android:valueTo="M 11.906 6.697 C 11.953 6.698 12.993 6.698 17.995 6.697
                             C 17.999 8.331 17.997 9.93 18.002 13.004
                             C 16.239 13.007 16.009 13.001 11.893 13.007
                             C 13.3 13 13.852 13.006 11.893 13.007 Z"
            android:valueType="pathType"/>
      </set>
   </aapt:attr>
</target>
</animated-vector>
```

> **Warning:** As path strokes are converted at build-time to filled path, the morphing of stroked paths is not supported. Any image with a path morphing animation on a stroked path will be rejected. Path strokes must be manually converted to filled path and the morphing of these new filled paths must be created.

Color and Opacity

Any path fillColor, strokeColor, fillAlpha and strokeAlpha attributes in the Android Vector Drawable can be animated.

```
<animated-vector xmlns:android="http://schemas.android.com/apk/res/android"
             xmlns:aapt="http://schemas.android.com/aapt">
 <aapt:attr name="android:drawable">
    <vector android:width="55dp" android:height="55dp"
          android:viewportWidth="55" android:viewportHeight="55">
       <group android:translateX="5">
        <path android:name="fillColor" android:fillColor="#FF00FF"
             android:pathData="M 0 0 h20 v20 h-20 Z"/>
        <path android:name="fillAlpha" android:fillColor="#FF0000"
             android:pathData="M 25 0 h20 v20 h-20 Z"/>
        <path android:name="strokeColor" android:strokeWidth="5"
             android:strokeColor="#FFFF00" android:pathData="M 0 25 h20 v20 h-20 Z"/>
        <path␣
↪android:name="strokeAlpha" android:strokeWidth="5" android:strokeColor="#00FF00"
             android:pathData="M 25 25 h20 v20 h-20 Z"/>
       </group>
    </vector>
 </aapt:attr>
```

(continues on next page)

(continued from previous page)

```
<target android:name="fillColor">
<aapt:attr name="android:animation">
   <set><objectAnimator
         android:propertyName="fillColor"
         android:duration="3000"
         android:valueFrom="#FF00FF"
         android:valueTo="#FFFF00"/>
   </set>
</aapt:attr>
</target>
<target android:name="strokeColor">
   <aapt:attr name="android:animation">
      <set><objectAnimator
            android:propertyName="strokeColor"
            android:duration="3000"
            android:valueFrom="#FFFF00"
            android:valueTo="#FF00FF"/>
      </set>
   </aapt:attr>
</target>

<target android:name="fillAlpha">
   <aapt:attr name="android:animation">
      <set> <objectAnimator
            android:propertyName="fillAlpha"
            android:duration="3000"
            android:valueFrom="0.2"
            android:valueTo="1"
            android:valueType="floatType"/>
      </set>
   </aapt:attr>
</target>
<target android:name="strokeAlpha">
   <aapt:attr name="android:animation">
      <set> <objectAnimator
            android:propertyName="strokeAlpha"
            android:duration="3000"
            android:valueFrom="1"
            android:valueTo="0.2"
            android:valueType="floatType"/>
      </set>
   </aapt:attr>
</target>
</animated-vector>
```

Warning: The color of paths colored with a linear gradient can not be animated.

Easing Interpolators

Every animation is associated with an easing interpolator. By default, the animation transition is linear, but the rate of change in the animation can be defined by an interpolator. This allows the existing animation effects to be accelerated, decelerated, repeated, bounced, etc.

The supported Android interpolators are:

- accelerate_cubic
- accelerate_decelerate
- accelerate_quad
- anticipate
- anticipate_overshoot
- bounce
- cycle
- decelerate_cubic
- decelerate_quad
- decelerate_quint
- fast_out_extra_slow_in
- fast_out_linear_in
- fast_out_slow_in
- linear
- linear_out_slow_in
- overshoot

  Any other vectorial path can also be used as the interpolator easing function.

  Following examples show the behavior of some of the interpolators for a simple translation animation.

- Image:

```
<animated-vector xmlns:android="http:/
↪/schemas.android.com/apk/res/android" xmlns:aapt="http://schemas.android.com/aapt">
 <aapt:attr name="android:drawable">
    <vector android:width="100dp
↪" android:height="100dp" android:viewportWidth="100" android:viewportHeight="100">
    <path android:pathData=
↪"M 0 0 h100 v20 h-100 Z" android:strokeColor="#FFFFFF" android:strokeWidth="1"/>
       <group android:name="translate">
        <path android:pathData="M 0 0 h20 v20 h-20 Z" android:fillColor="#335566"/>
     </group>
    </vector>
 </aapt:attr>

 <target android:name="translate">
 <aapt:attr name="android:animation">
    <set><objectAnimator
          android:propertyName="translateX"
```

(continues on next page)

(continued from previous page)

```
            android:duration="2000"
            android:valueFrom="0"
            android:valueTo="80"
            android:interpolator = "@android:interpolator/linear" />
    </set>
 </aapt:attr>
 </target>
</animated-vector>
```

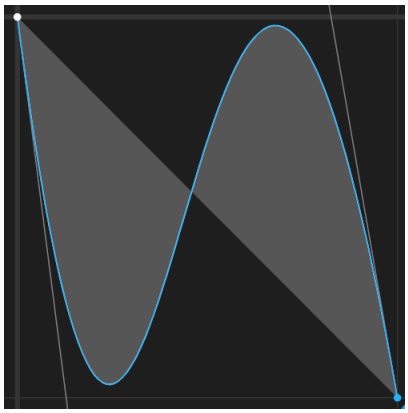
```
android:interpolator = "@android:interpolator/linear"
```

```
android:interpolator = "@android:interpolator/accelerate_cubic"
```

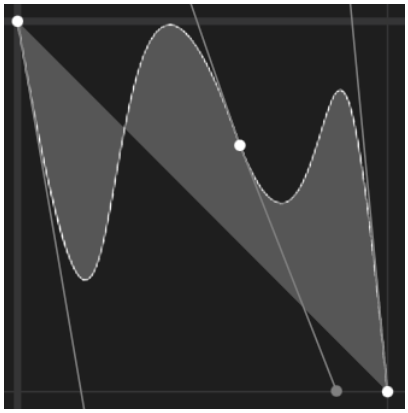
```
android:interpolator = "@android:interpolator/bounce"
```

```
android:interpolator = "@android:interpolator/fast_out_slow_in"
```

```
<aapt:attr name="android:interpolator">
  <pathInterpolator android:pathData="M 0 0 C 0.371 2.888 0.492 -1.91 1 1"/>
</aapt:attr>
```



```
<aapt:attr name="android:interpolator">
  <pathInterpolator android:pathData=
↪"M 0 0 C 0.333 1.939 0.171 -0.906 0.601 0.335 C 0.862 0.998 0.83 -0.771 1 1"/>
</aapt:attr>
```



External Images

To fetch images from external memory, the application must pre-register the *external Image resources*. The management of this kind of image may be different than the compile-time images and may require some allocations in the MicroUI *Images Heap*. For more details about the external image management, refers to the VEE Port Guide chapter *External Memory*.

Caching Generated Images

Images converted using the Image Generator can be cached so that they are not rebuilt every time the application is launched. Doing so can significantly speed up the application build phase.

See *Caching Generated Images* to have more details.

Note: The cache is available from version 13.6 of the UI Pack.

Limitations / Supported Features

Android Vector Drawable

The MicroVG library supports most of the Android Vector Drawable features with the following limitations:

- `clip-path` feature is only supported for static images.
- `trim-path` animation is not supported.
    - morphing animations are not supported for paths with stroke.
    - usage of path opacity is limited
  - `drawImage` with alpha is not supported if the image contains overlapping paths.
    - images with global alpha(`android:alpha` attribute of `vector` element) and overlapping paths are not supported.
    - Beware that using `android:fillColor` and `android:strokeColor` attributes on the same path leads to overlapping paths.

- radial and sweep gradient types are not supported.
- tint , tintMode and autoMirrored features are not supported.
- trimPath feature is not supported.

SVG

The MicroVG library supports a subset of SVGTiny: https://www.w3.org/TR/SVGTiny12/ including:

- Path
- Basic shape
- Painting filling
- Painting stroking
- Painting gradient (only linear gradient with one pattern)
- Painting color formats : #RRGGBB, #RGB, rgb(r,g,b), keywords
- Transforms
- Text
- Fonts (the text fonts used in the SVG file has to be installed on the operating system)

Debug Traces

MicroVG traces several actions when traces are enabled. This chapter explains the trace identifiers.

Note: The traces are only available on the Embedded VEE Port (not on the Simulator).

Trace format

The trace output format is the following:

```
[TRACE: MicroVG] Event AA(BB[CC])
```

where:

- AA is the event identifier. See next table.
- BB is the event data.
- CC is the index of the event data (0x0).

For example, given the following trace output:

```
[TRACE: MicroVG] Event 0x2(2[0x0])
```

- 0x2 -> Execute drawing event
- 2 -> Event “Draw String” (index 0x0)

Trace identifiers

The following tables describe some events data.

Table 28: MicroVG Traces

Event ID	Description	End of Event
0x0 (0)	Image event %0% (see Image Type)	End of %0% (see Image Type).
0x1 (1)	Font event %0% (see Font Type)	End of %0% (see Font Type).
0x2 (2)	Drawing %0% (see Drawing type)	End of %0% (see Drawing Type).
0x3 (3)	New image %0% (see Image Type)	Image created, image identifier is %0% .
0x4 (4)	New image characteristics: %0% (see New Image), identifier is %1% and size is %2% x %3% .	
0xA (10)	Draw Path dest %0% , anchor= %1% , %2%	Drawing status %0% (see Drawing Status)
0xB (11)	Draw Path with Gradient dest %0% , anchor= %1% , %2%	Drawing status %0% (see Drawing Status)
0xC (12)	Draw String dest %0% , %1% characters, anchor= %2% , %3%	Drawing status %0% (see Drawing Status)
0xD (13)	Draw String with Gradient dest %0% , %1% characters, anchor= %2% , %3%	Drawing status %0% (see Drawing Status)
0xE (14)	Draw String On Circle dest %0% , %1% characters, anchor= %2% , %3% , radius= %4% , direction= %5% (see Direction Type)	Drawing status %0% (see Drawing Status)
0xF (15)	Draw String On Circle with Gradient dest %0% , %1% characters, anchor= %2% , %3% , radius= %4% , direction= %5% (see Direction Type)	Drawing status %0% (see Drawing Status)
0x10 (16)	Draw Image dest %0% , image= %1% , anchor= %2% , %3%	Drawing status %0% (see Drawing Status)

Table 29: Image Type

Event ID	Description
0x0 (0)	Get or load image from RAW file
0x1 (1)	Create BufferedVectorImage
0x2 (2)	Close image

Table 30: New Image

Event ID	Description
0x0 (0)	Get image from RAW file
0x1 (1)	Load image from RAW file
0x2 (2)	Create BufferedVectorImage
0x3 (3)	Build image
0x4 (4)	New filtered image from RAW image
0x5 (5)	New filtered image from BufferedVectorImage
0x6 (6)	New filtered image from builder image

Table 31: Font Type

Event ID	Description
0x0 (0)	Load font from TTF / OTF file
0x1 (1)	Retrieve font baseline
0x2 (2)	Retrieve font height
0x3 (3)	Measure string width
0x4 (4)	Measure string height

Table 32: Drawing Type

Event ID	Description
0x0 (0)	Fill path with a color
0x1 (1)	Fill path with a linear gradient
0x2 (2)	Draw string with a color
0x3 (3)	Draw string with a linear gradient
0x4 (4)	Draw string on a circle with a color
0x5 (5)	Draw string on a circle with a gradient
0x6 (6)	Draw image

Table 33: Direction Type

Event ID	Description
0x0 (0)	Clockwise
0x1 (1)	Anticlockwise

Table 34: Drawing Status

Event ID	Description
0x0 (0)	Synchronous drawing done
0x1 (1)	Asynchronous drawing delayed

SystemView Integration

The traces are *SystemView* compatible.

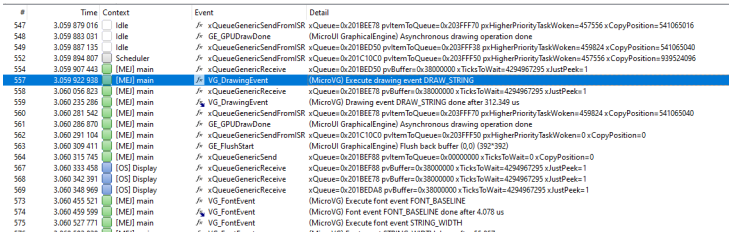


Fig. 146: MicroVG Traces displayed in SystemView

The following text can be copied in a file called SYSVIEW_MicroVG.txt and copied in SystemView installation folder (e.g. SEGGER/SystemView_V252a/Description/).

```
#
# SystemView Description File
#

#============
# NamedTypes
#============

# --------
# Image Events
# --------

# Deprecated (kept for backward compatibility VG Pack < 1.8.1)
NamedType VGImage 0=OPEN_RAW_IMAGE
NamedType VGImage 1=CREATE_IMAGE
NamedType VGImage 2=CLOSE_IMAGE

NamedType VGNewImage 0=GET_RAW_IMAGE
NamedType VGNewImage 1=LOAD_RAW_IMAGE
NamedType VGNewImage 2=MUTABLE_IMAGE
NamedType VGNewImage 3=BUILDER_IMAGE
NamedType VGNewImage 4=FILTERED_IMAGE_FROM_RAW
NamedType VGNewImage 5=FILTERED_IMAGE_FROM_MUTABLE
NamedType VGNewImage 6=FILTERED_IMAGE_FROM_BUILDER

# --------
# Font Events
# --------

NamedType VGFont 0=LOAD_FONT
NamedType VGFont 1=FONT_BASELINE
NamedType VGFont 2=FONT_HEIGHT
NamedType VGFont 3=STRING_WIDTH
NamedType VGFont 4=STRING_HEIGHT

# ----------
# Drawing Events
# ----------

NamedType VGString *="%u characters"
NamedType VGRadius *="radius=%u"
NamedType VGSource *="image=0x%x"

NamedType VGDirection 0="direction=clockwise"
NamedType VGDirection 1="direction=anticlockwise"

# ------------
# Old Drawing Events
# ------------

NamedType VGDraw 0=DRAW_PATH
NamedType VGDraw 1=DRAW_PATH_GRADIENT
NamedType VGDraw 2=DRAW_STRING
NamedType VGDraw 3=DRAW_STRING_GRADIENT
NamedType VGDraw 4=DRAW_STRING_ON_CIRCLE
NamedType VGDraw 5=DRAW_STRING_ON_CIRCLE_GRADIENT
NamedType VGDraw 6=DRAW_IMAGE
```

(continues on next page)

(continued from previous page)

```
# ------------
# UI Drawing Events
# ------------

# Copy of MicroUI events (required by the Front Panel)
NamedType UIDrawAsync 0="Drawing operation done"
NamedType UIDrawAsync 1="Drawing operation delayed"
NamedType UIDestination *="dest=0x%x"

#=======
# Events
#=======

# ----------
# MicroVG Library
# ----------

# [0-9]: 10 events

# Deprecated (kept for backward compatibility VG Pack < 1.8.1)
0       VG_ImageEvent      type=%VGImage | type=%VGImage done
2       VG_DrawingEvent    type=%VGDraw  | type=%VGDraw done

1       VG_FontEvent       type=%VGFont  | type=%VGFont done
3       VG_NewImage        type=%VGNewImage | id=0x%x
4       VG_ImageData       type=%VGNewImage id=0x%x size=%u*%u

# -----------
# MicroVG C Module
# -----------

# [10-39]: 30 events

10      VG_DrawPathColor              %UIDestination anchor=%d,%d | %UIDrawAsync
11      VG_DrawPathGradient           %UIDestination anchor=%d,%d | %UIDrawAsync
12      VG_DrawStringColor            %UIDestination %VGString anchor=%d,%d | %UIDrawAsync
13      VG_DrawStringGradient         %UIDestination %VGString anchor=%d,%d | %UIDrawAsync
14      VG_DrawStringOnCircleColor    %UIDestination %VGString anchor=%d,%d %VGRadius %VGDirection | %UIDrawAsync
15      VG_DrawStringOnCircleGradient %UIDestination %VGString anchor=%d,%d %VGRadius %VGDirection | %UIDrawAsync
16      VG_DrawImage                  %UIDestination %VGSource anchor=%d,%d | %UIDrawAsync
```

Error Messages

When an exception is thrown by the implementation of the MicroVG API, the exception VectorGraphicsException with the error message MicroVG:E=<messageId> is issued, where the meaning of <messageId> is defined in the *Field Detail* of each error code.

Android Vector Drawable Loader

Overview

The AVD Loader is an Add-On Library that can load vector images from Android Vector Drawable XML files. Unlike the vector images that are loaded using a raw output file format (see *Vector Images*), the XML parsing and interpreting is done at runtime. This is useful for loading a vector image as an external resource, especially when the resource has to be loaded dynamically (i.e., not known at build-time).

To use the AVD Loader library, add the following dependency to the project build file:

Gradle (build.gradle.kts)

MMM (module.ivy)

```
implementation("ej.library.ui:vectorimage-loader:1.1.0")
```

```
<dependency org="ej.library.ui" name="vectorimage-loader" rev="1.1.0"/>
```

Note: The AVD Loader library requires the VG Pack 1.2 and above.

Supported Format

The library supports the vector drawables with the following elements (in that order):

<vector>
Used to define a vector drawable

android:viewportWidth The width of the image (must be a positive value).

android:viewportHeight The height of the image (must be a positive value).

<path>
Defines a path.

android:fillColor (optional) The color used to fill the path. Color is specified as a 32-bit ARGB value in hexadecimal format (#AARRGGBB). This attribute is optional when a gradient color is specified (see below).

android:fillType The fillType for the path, can be either evenOdd or nonZero .

android:pathData The path data, using the commands in { M , L , C , Q , Z } (match upper-case).

A linear gradient can also be used as color fill for a <path> . This element is optional if a solid color fill has been specified.

<gradient>
Used to define a linear gradient

android:endX The x-coordinate for the end of the gradient vector.

android:endY The y-coordinate for the end of the gradient vector.

android:startX The x-coordinate for the start of the gradient vector.

android:startY The y-coordinate for the start of the gradient vector.

<item> Defines an item of the gradient (minimum two items for a gradient).

android:color The color of the item. Color is specified as a 32-bit ARGB value in hexadecimal format (#AARRGGBB).

android:offset The position of the item inside the gradient (value in [0..1]).

Here is an example of a Vector Drawable myImage.xml that complies with that format. It defines a 100 x 100 image with two paths: the first one with a solid color fill, the second one with a linear gradient.

```
<vector xmlns:aapt="http://schemas.android.com/aapt" xmlns:android="http://
↪schemas.android.com/apk/res/android" android:height="100.0dp" android:viewportHeight=
↪"100.0" android:viewportWidth="100.0" android:width="100.0dp">
  <path android:fillColor="#FFFFFFAA
↪" android:fillType="nonZero" android:pathData="M0,0L50,0L50,50L0,50Z " />
  <path android:fillType="nonZero" android:pathData="M50,50L100,50L100,100L50,100Z ">
    <aapt:attr name="android:fillColor">
      <gradient android:endX="100.0" android:endY=
↪"100.0" android:startX="50.0" android:startY="50.0" android:type="android:linear">
        <item android:color="#FF0000FF" android:offset="0.0" />
        <item android:color="#FFFF00FF" android:offset="1.0" />
      </gradient>
    </aapt:attr>
  </path>
</vector>
```

If the input Vector Drawable does not comply with this format, the library will throw an exception.

Note: To make a Vector Drawable compatible with the library, use the image generator tool to convert the AVD into a compatible version. See *this section* for more information.

Format Limitations

The library only supports a subset of the Vector Drawable specification.

The AVD Loader is designed to load AVDs at runtime on embedded devices. It minimizes Managed Heap usage and CPU time for XML parsing, Path creation, and adds little code to the final executable. The format is intentionally limited to reduce processing time and complexity while ensuring good performance, knowing that the *pre-processing step* can convert any AVD into the compatible format.

Note that this limitation on the Android Vector Drawable format does not apply to AVDs loaded as *raw vector images*.

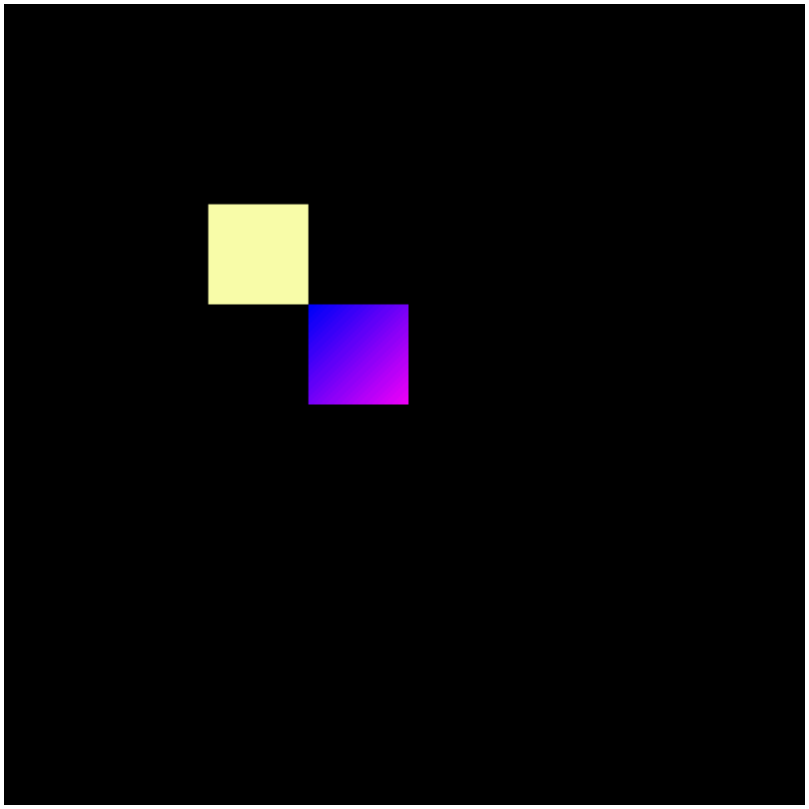
Loading a Vector Drawable

The following code loads the Vector Drawable `myImage.xml` with the `AvdImageLoader.loadImage()` method. This method has one parameter which is the path to the Vector Drawable file, provided as *a raw resource of the application*. The resulting vector image can then be drawn on the display:

```
public static void main(String[] args) {
   MicroUI.start();

   Display display = Display.getDisplay();
   GraphicsContext g = display.getGraphicsContext();

   try (ResourceVectorImage image = AvdImageLoader.loadImage("/images/myImage.xml")) {
      VectorGraphicsPainter.drawImage(g, image, 100, 100);
      display.requestFlush();
   }
}
```

Listing 21: Declaration of the resource in a *.resources.list file.

```
/images/myImage.xml
```



Note: The image must be provided as a raw resource of the application, either *internal or external*. For external resource loading, the BSP must implement the proper Abstraction Layer API (LLAPI), see *External Resources Loader* for more information on the implementation.

Warning: The new image is a ResourceVectorImage. In the current implementation, an image loaded with the AvdImageLoader is allocated in the Managed Heap. To release memory, the application must close the image and remove any references to it.

Limitations

The AVD Loader can only load static images (i.e., no animations). The other limitations are *the same as for vector images*.

Advanced

Make a AVD Compatible with the Library

To ensure that a Vector Drawable can be loaded by the AVD Loader library at runtime, the image generator tool can generate a compatible version of the drawable.

The tool comes with the VG pack installed in the platform, use the following command line to run it:

```
java -cp [path_to_platform]/source/tools/imagegenerator-vectorimage.
↪jar com.microej.converter.vectorimage.Main --input originalImage.xml --avd myImage.xml
```

This processes the input Vector Drawable originalImage.xml and outputs a Vector Drawable myImage.xml which is compliant with the library and optimized for runtime loading.

The processing does the following:

- Normalize the output
- Limit the size of the XML file (e.g., minification)
- Pre-process the resource-consuming operations (e.g., transformations, stroking)

Make a SVG Compatible with the Library

It is possible to convert a SVG into a compatible Vector Drawable using the platform tooling. Use the following command:

```
java -cp [path_to_platform]/source/tools/imagegenerator-vectorimage.
↪jar com.microej.converter.vectorimage.Main --input originalImage.svg --avd myImage.xml
```

This processes the input SVG originalImage.svg and outputs a Vector Drawable myImage.xml .

Memory Usage

The loading of a Vector Drawable at runtime uses Managed Heap:

- for the working buffers and intermediate objects used during the loading phase. The XML parser is optimized to stream the data and uses as few heap as possible.
- for the image data.

Simplify the Path Data

The loading time and heap usage grow linearly with the number of path commands in the Vector Drawable. To achieve optimal performances, it is recommended to reduce the number of path commands, by "simplifying" the paths. The simplification algorithm will determine the optimal amount of anchor points to use in the artwork. Most of the modern Graphic Design Software have an option to simplify a path (check this article for Adobe Illustrator for example).

Monitor the Number of Path Commands

To print the number of paths and path commands declared in a Vector Drawable, set the *constant* ej.vectorimage.loader.debug.enabled to true. This will output the numbers in the console when loading a file.

Output example:

```
avdimageloader INFO: Parsed a path data with a number of 5 commands
avdimageloader INFO: Parsed a path data with a number of 5 commands
avdimageloader INFO: Parsed a path data with a number of 28 commands
avdimageloader INFO: Number of paths in loaded image: 3
```

Troubleshooting

The Image Cannot Be Parsed

A error can be raised when the parsing fails:

```
Exception in␣
↪thread "main" ej.microvg.VectorGraphicsException: MicroVG: The image cannot be parsed.
↪ The image must be a valid AVD image, converted with the platform's image generator.
```

This error indicates that the file is not a compatible Vector Drawable, as specified in *this section*.

How to Add Emojis to a Vector Font

MicroVG supports the drawing of multicolor fonts that use the COLR/CPAL tables to define multi-layered glyphs. Multicolor fonts are mainly used for providing a set of colorful emojis in messaging applications. However, emojis fonts usually do not contain many characters other than emojis, which requires applications to use multiple fonts to handle all use cases.

One solution to minimize the number of fonts used by an application is to add emojis to another font (i.e., combine fonts into one). This article shows how to achieve this using FontLab, a third-party font editor.

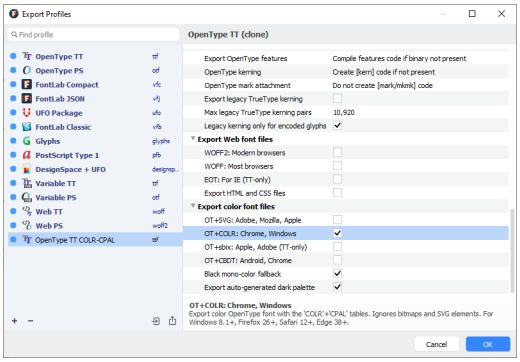
Note: FontLab is not a free software (it has a 30-days trial period). Tests with other tools, including free solutions, were unsuccessful in this very specific task (e.g. FontTools, FontForge which are great tools for font editing).

Prerequisites

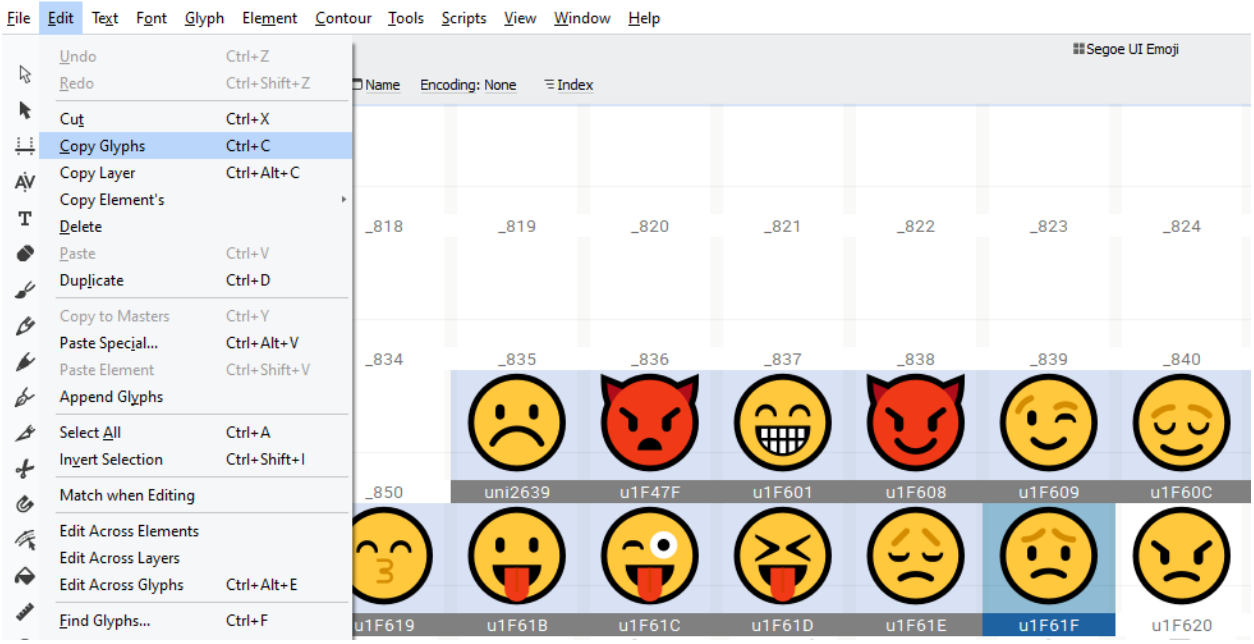
- Windows 10 (and higher) or macOS 10.14 (and higher),
- An COLR/CPAL emoji font (e.g., Segoe UI Emoji),
- A target font (i.e., a TTF/OTF font to append emojis to).

Append the Emoji Glyphs

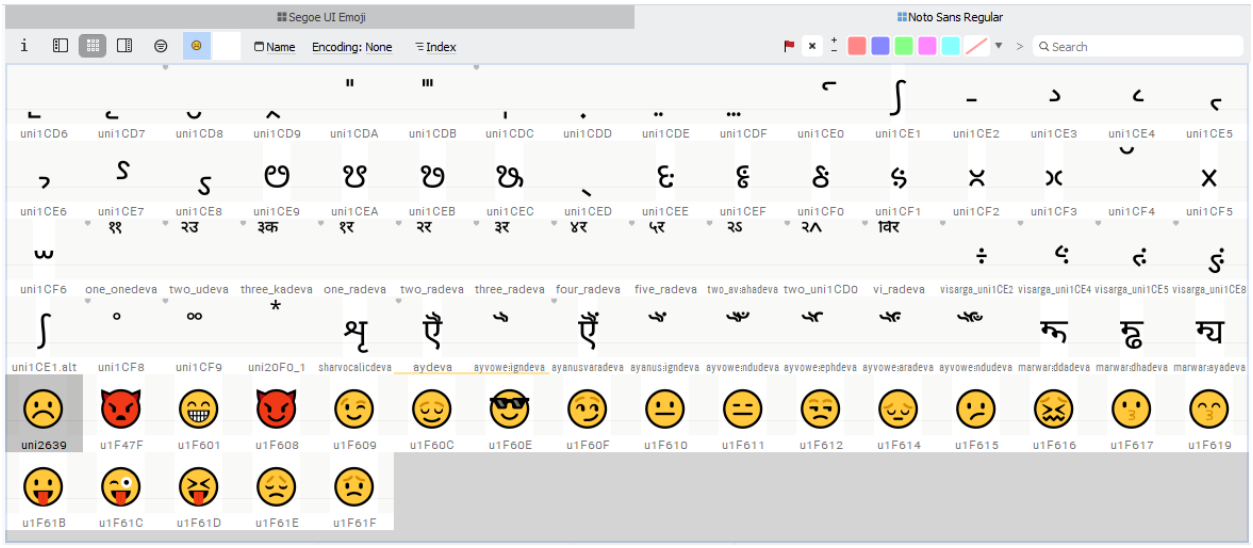
1. Download and install FontLab.
2. In FontLab, go to `File` > `Export Profiles…` .
3. Create a new Export Profile (the `+` button on the bottom-left).
4. Edit the new profile to match the configuration below in menu `Export color font files` :



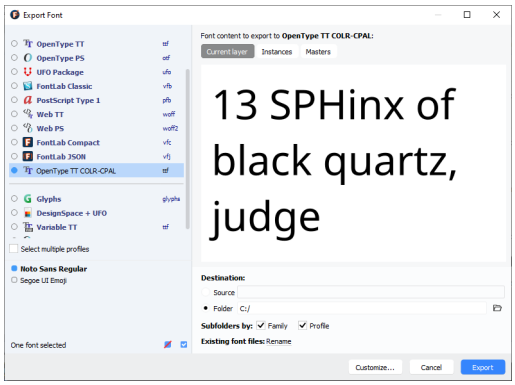
5. Click on `OK` to close the `Export Profiles` window.
6. Open the emoji font: go to `File` > `Open Fonts…` and browse to the font file.
7. Open the target font.
8. Select a range of glyphs in the emoji font and select `Edit` > `Copy Glyphs` :



9. Select the target font and append the copied glyphs: go to `Edit` > `Append Glyphs` .
10. Check that the glyphs have been added to the target font:



11. To save the changes, go to `File` > `Export Font As…` .
12. In the `Export Font` dialog, select the new Export Profile (with COLR/CPAL support) and `Destination` .



13. Click `Export` .

    The exported font can then be used in an application, as described in the *Application Developer Guide*.

    This procedure can also be used to add non-emoji glyphs to a font.

> **Warning:** There are multiple ways of implementing emojis in fonts. The four main formats are COLR/CPAL (Microsoft), CBDT/CBLC (Google), SVG (Adobe/Firefox) and sbix (Apple). Each

format uses custom tables in fonts to describe the emoji glyphs. MicroVG supports COLR/CPAL tables and this article only applies to this case. See *this section* for more details about color emojis support with MicroVG.

Motion

Motion is a library that provides some of the most commonly used easing functions and allows users to define new ones.

Easing functions help to mimic the movement of real objects, which allows for the creation of appealing visual animations. The library can also be useful for other purposes such as defining non-linear gradients for example.

Usage

To use the Motion library, add Motion library module to the project build file:

Gradle (build.gradle.kts)

MMM (module.ivy)

```
implementation("ej.library.ui:motion:4.0.1")
```

```
<dependency org="ej.library.ui" name="motion" rev="4.0.1"/>
```

Concepts

Functions

The Function interface represents a mathematical function. Its purpose is to associate a floating value to any input floating value between `0f` and `1f`. This function specifies the rate of change of a parameter over time (for example the position of an object).

The library provides a set of functions among the most common ones. Each one contains a convenient instance. For example the singleton for the linear function.

Simple functions

The constant function represents a constant motion whose value is always the stop value.

The linear function represents a uniform linear motion: the velocity is constant (no acceleration or deceleration).

Easing functions

The easing functions come in 3 forms:

- Ease-in: the function is eased at the beginning.
- Ease-out: the function is eased at the end.
- Ease-in-out: the function is eased at the beginning and the end.

Here are the easing functions available in the library:

Circ ease-in

Circ ease-out

Circ ease-in-out

Cubic ease-in

Cubic ease-out

Cubic ease-in-out

Expo ease-in

Expo ease-out

Expo ease-in-out

Quad ease-in

Quad ease-out

Quad ease-in-out

Quart ease-in

Quart ease-out

Quart ease-in-out

Quint ease-in

Quint ease-out

Quint ease-in-out

Sine ease-in

Sine ease-out

Sine ease-in-out

Back ease-in

Back ease-out

Back ease-in-out

Bounce ease-in

Bounce ease-out

Bounce ease-in-out

Elastic ease-in

Elastic ease-out

Elastic ease-in-out

Custom Function

It is possible to create custom functions by creating a class implementing the Function interface. The *computeValue(float)* method should return a value between `0f` and `1f`.

Motion

The *Motion* class is used to describe the movement of an element. It is made of a function, a start value, a stop value and a duration.

It proposes a convenient method Motion.getValue(long) to retrieve the position of the element at the specified elapsed time.

MWT (Micro Widget Toolkit)

MWT is a toolkit that simplifies the creation and use of graphical user interface widgets on a pixel-based display.

The aim of this library is to be sufficient to create complex applications with a minimal framework. It provides the main concepts without managing particular needs. Specific needs can be met by a MWT expert by creating new widgets, adding more complex concepts, etc. The flexibility of the MWT open framework allows the selection of only what is necessary for the application in order to guarantee lightweight applications and fast execution.

Usage

To use the MWT library, add MWT library module to the project build file:

Gradle (build.gradle.kts)

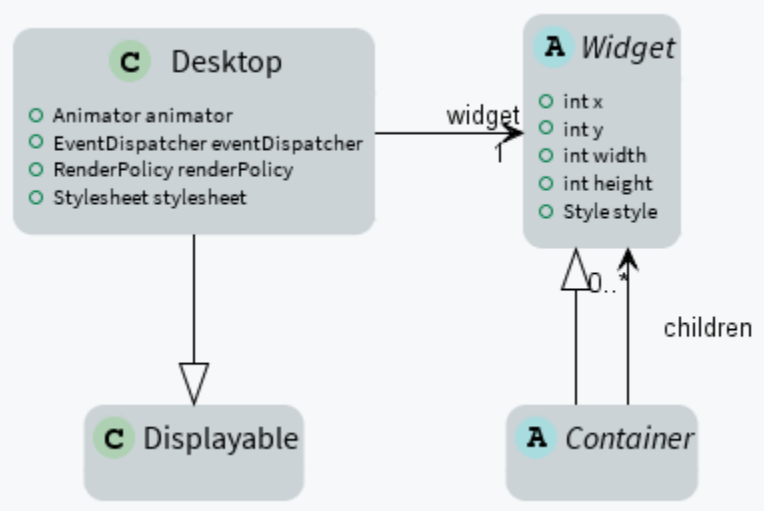
MMM (module.ivy)

```
implementation("ej.library.ui:mwt:3.3.0")
```

```
<dependency org="ej.library.ui" name="mwt" rev="3.3.0"/>
```

Concepts

Graphical Elements



Widget

A widget is an object that is intended to be displayed on a screen. A widget occupies a specific region of the display and holds a state. A user may interact with a widget (using a touch screen or a button for example).

Widgets are arranged on a desktop. A widget can be part of only one desktop hierarchy, and can appear only once on that desktop.

Container

A container follows the composite pattern: it is a widget composed of other widgets. It also defines the layout policy of its children (defining their bounds). The children's positions are relative to the position of their parent. Containers can be nested to design elaborate user interfaces.

By default, the children are rendered in the order in which they have been added in the container. And thus if the container allows overlapping, the widgets added last will be on top of the widgets added first. A container can also modify how its children are rendered.

Desktop

A desktop is a displayable intended to be shown on a display (cf. MicroUI). At any time, only one desktop can be displayed per display.

A desktop contains a widget (or a container). When the desktop is shown, its widget (and all its hierarchy for a container) is drawn on the display.

Rendering

A new rendering of a widget on the display can be requested by calling its requestRender() method. The rendering is done asynchronously in the MicroUI thread.

When a container is rendered, all its children are also rendered.

A widget can be transparent, meaning that it does not draw every pixel within its bounds. In this case, when this widget is asked to be rendered, its parent is asked to be rendered in the area of the widget (recursively if the parent is also transparent). Usually a widget is transparent when its background (from the style) is transparent.

A widget can also be rendered directly in a specific graphics context by calling its render(GraphicsContext) method. It can be useful to render a widget (and its children) in an image for example.

Render Policy

A render policy is a strategy that MWT uses in order to repaint the entire desktop or to repaint a specific widget.

The most naive render policy would be to render the whole hierarchy of the desktop whenever a widget has changed. However DefaultRenderPolicy is smarter than that: it only repaints the widget, and its ancestors if the widget is transparent. The result is correct only if there is no overlapping widget, in which case OverlapRenderPolicy should be used instead. This policy repaints the widget (or its non-transparent ancestor), then it repaints all the widgets that overlap it.

When using a *partial buffer*, these render policies can not be used because they render the entire screen in a single pass. Instead, a custom render policy which renders the screen in multiple passes has to be used. Refer to the partial buffer demo for more information on how to implement this render policy and how to use it.

The render policy can be changed by overridding Desktop.createRenderPolicy().

Widget Lifecycle

Desktops and widgets run through different states. Once created, they can be attached, then they can be laid out, and finally they can be shown.

A desktop is attached automatically as soon as it is shown on the display and detached when hidden. It can also be attached manually by calling Desktop.setAttached() or detached by calling Desktop.setDetached(). It is particularly useful to render the desktop (and its widgets) in a buffered image for example.

A widget is considered as attached when it is contained in a desktop that is attached. In the same way, by default, a widget is shown when its desktop is shown. But for optimization purposes, a container can control when its children are shown or hidden. A typical use case is when the widgets are moved outside the display (in a scroll container for instance).

Hooks

When attached, a widget is notified by a call to its onAttached() method. This notification can be useful to allocate some images or other resources for example. These resources can be used to compute the size of the widget and to render it. In other words, after this call, a widget is ready to be laid out.

After being laid out, a widget is notified by a call to its onLaidOut() method. Being laid out means that its bounds inside its parent are set. During this call, the widget can prepare some resources used later by the rendering. For example, it can split a string into several lines based on its width. Another idea could be to allocate a buffered image and draw the background to avoid repainting everything during the rendering.

Beware that a widget can be laid out several times once attached (typically each time a Desktop.requestLayOut() or Widget.requestLayOut() is done).

When a whole hierarchy is ready to be rendered, all the widgets are notified by a call to their onShown() method. This notification is particularly useful to start a periodic refresh or an animation.

A widget can finally be hidden and detached, in which cases its methods onHidden() and onDetached() will be called respectively. In these methods, anything that has been started or allocated during the previous phases must be stopped or freed correctly to avoid memory leaks.

Lay Out

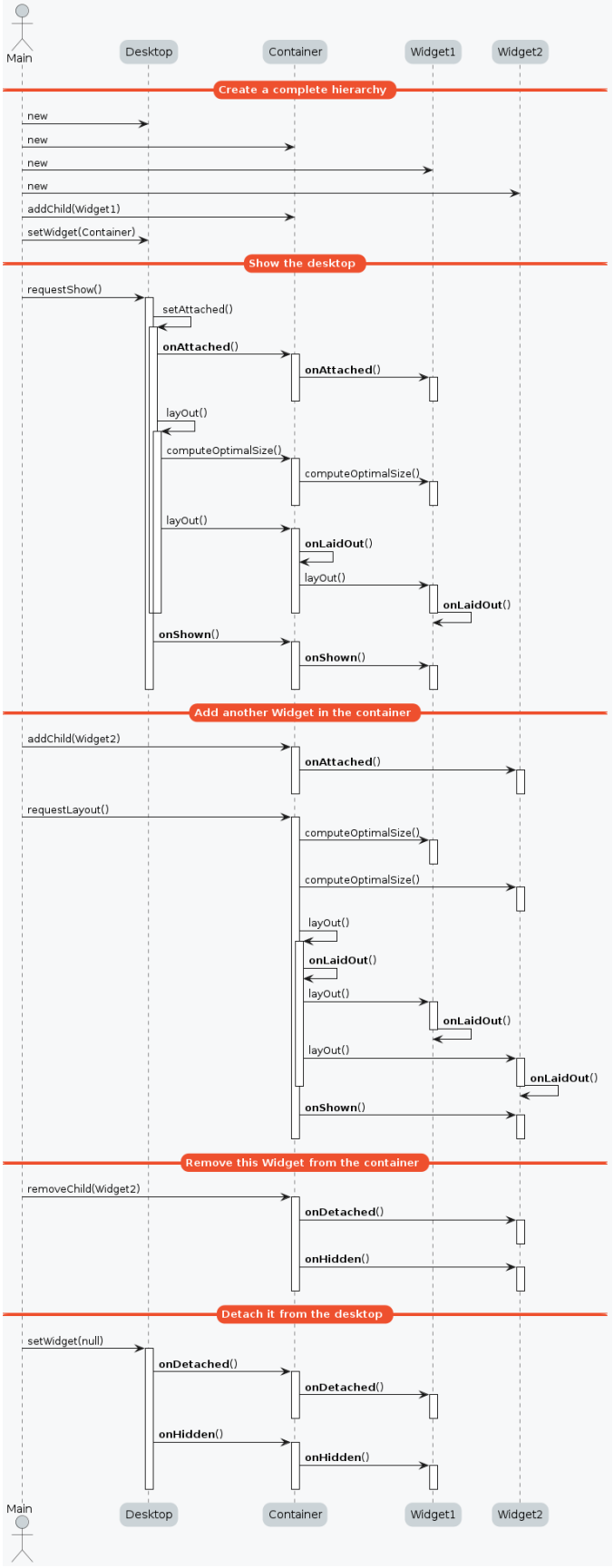
All widgets are laid out at once during the lay out process. This process can be started by Desktop.requestLayOut() or Widget.requestLayOut(). The layout is also automatically done when the desktop is shown (Desktop.onShown()).

This process is composed of two steps. Each step browses the hierarchy of widgets following a depth-first algorithm:

- Compute the optimal size for each widget and container (considering the constraints of the lay out).
- Set the position and size for each widget.

A widget must implement its Widget.computeContentOptimalSize() method. It is explained in detail in this section: *Computing the Optimal Size of the Widget*.

A container is responsible for laying out its children. For that it must implement its own Widget.computeContentOptimalSize() method and call the Container.computeChildOptimalSize() method for each of its children. And it must implement its Container.layOutChildren() method and call the Container.layOutChild() method for each of its children. It is explained in detail in these sections: *Computing the Optimal Size of the Container* and *Laying out the Children of the Container*.


Main
Desktop
Container
Widget1
Widget2
Create a complete hierarchy
new
new
new
new
addChild(Widget1)
setWidget(Container)
Show the desktop
requestShow()
setAttached()
onAttached()
onAttached()
layOut()
computeOptimalSize()
computeOptimalSize()
layOut()
onLaidOut()
layOut()
onLaidOut()
onShown()
onShown()
Add another Widget in the container
addChild(Widget2)
onAttached()
requestLayout()
computeOptimalSize()
computeOptimalSize()
layOut()
onLaidOut()
layOut()
onLaidOut()
layOut()
onLaidOut()
onShown()
Remove this Widget from the container
removeChild(Widget2)
onDetached()
onHidden()
Detach it from the desktop
setWidget(null)
onDetached()
onDetached()
onHidden()
onHidden()
Main
Desktop
Container
Widget1
Widget2

Rendering Pipeline

The Rendering Pipeline of an MWT application consists of three main phases: Layout, Render, and Flush.

1. *Layout*: This phase determines which widgets should be displayed on the screen and the positions of the widgets. It is typically triggered when widgets are added or removed from the widget hierarchy. An application should only modify the widget hierarchy when necessary and avoid doing so during animation to ensure efficiency.
2. *Render*: During this phase, each widget executes its rendering code to perform the necessary drawing operations. The widgets must render only what is needed and minimize overlapping with other widgets to ensure optimal performance.
3. *Flush*: This phase involves copying the back buffer to the front buffer. The VEE Port performs this operation, and it is the responsibility of the VEE Port developer to optimize this process, for example, by utilizing a GPU.

Event Dispatch

Events generated in the hardware (touch, buttons, etc.) are sent to the event dispatcher of the desktop. It is then responsible of sending the event to one or several widgets of the hierarchy. A widget receives the event through its handleEvent(int) method. This method returns a boolean that indicates whether or not the event has been consumed by the widget.

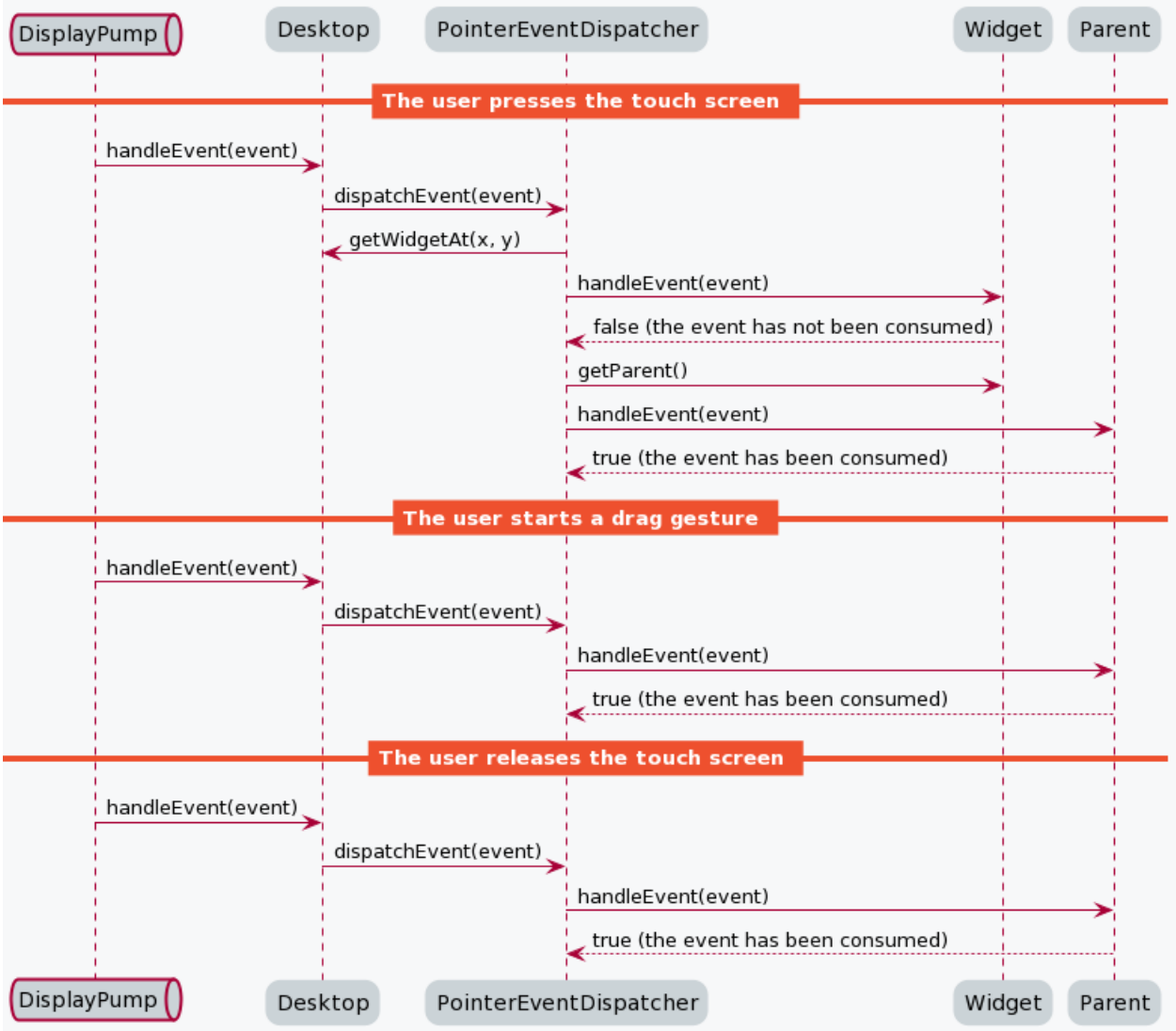
Widgets are disabled by default and don't receive the events.

Pointer Event Dispatcher

By default, the desktop proposes an event dispatcher that handles only pointer events.

Pointer events are grouped in sessions. A session starts when the pointer is pressed, and ends when the pointer is released or when it exits the pressed widget.

While no widget consumes the events, they are sent to the widget that is under the pointer (see Desktop.getWidgetAt(int, int)), then sent to all its parent hierarchy recursively.



Once a widget has consumed an event, it will be the only one to receive the next events during the session.

Use case examples:

- A button-like widget should not consume the press event since the user can drag the pointer after that.
  - In a simple case, the user may release the pointer outside the bounds of the button, thus the button will not be clicked.
  - Or the button may be included in a container that capture the drag events (and/or the release ones) to perform a scroll or a drag-and-drop for instance.

- In the same way, a directed scroll (horizontal or vertical) should only start to consume pointer events once a swipe in the right direction is detected. This way, 2 scrolls with opposite directions may be nested, each one handling one direction.

  By default, the reactive area of a widget is the boundaries of its content, plus its padding, plus its border (does not include the margin).

  A widget can redefine its reactive area by subclassing the contains(int x, int y) method. It is useful when a widget does not fill fully its bounds.

Style

A style describes how widgets must be rendered on screen. The attributes of the style are strongly inspired from CSS.

Dimension

The dimension is used to constrain the size of the widget.

MWT provides multiple implementations of dimensions:

- NoDimension does not constrain the dimension of the widget, so the widget will take all the space granted by its parent container.
- OptimalDimension constrains the dimension of the widget to its optimal size, which is given by the computeContentOptimalSize() method of the widget.
- FixedDimension constrains the dimension of the widget to a fixed absolute size.
- RelativeDimension constrains the dimension of the widget to a percentage of the size of its parent container.

Alignment

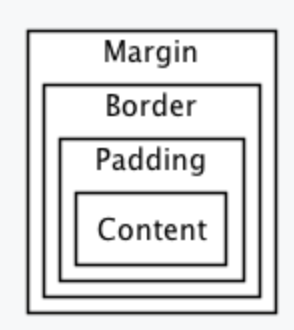
The horizontal and vertical alignments are used to position the content of the widget within its bounds.

The alignment is used by the framework to position the widget within its available space if the size of the widget has been constrained with a Dimension.

The alignment can also be used in the renderContent() method in order to position the drawings of the widget (such as a text or an image) within its content bounds.

Outlines

The margin, border and padding are the 3 outlines which wrap the content of the widget. The widget is wrapped in the following sequence: first the padding, then the border, and finally the margin.



MWT provides multiple implementations of invisible outlines which are usually used for margin and padding:

- NoOutline does not wrap the widget in an outline.
- UniformOutline wraps the widget in an outline which thickness is equal on all sides.
- FlexibleOutline wraps the widget in an outline which thickness can be configured for each side.

MWT also provides multiple implementations of visible outlines which are usually used for border:

- RectangularBorder draws a plain rectangle around the widget.
- RoundedBorder draws a plain rounded rectangle around the widget.

Background

The background is used to render the background of the widget. The background covers the border, the padding and the content of the widget, but not its margin.

MWT provides multiple implementations of backgrounds:

- NoBackground leaves a transparent background behind the widget.
- RectangularBackground draws a plain rectangle behind the widget.
- RoundedBackground draws a plain rounded rectangle behind the widget.
- ImageBackground draws an image behinds the widget.

Color

The color is not used by the framework itself, but it may be used in the `renderContent()` to select the color of the drawings.

Font

The font is not used by framework itself, but it may be used in the `renderContent()` to select the font to use when drawing strings.

Extra Fields

Extra fields are not used by framework itself, but they may be used in the `renderContent()` to customize the behavior and the appearance of the widget.

See chapter *How to Define an Extra Style Field* for more information on extra fields.

Stylesheet

A stylesheet allows to customize the appearance of all the widgets of a desktop without changing the code of the widget subclasses.

MWT provides multiple implementations of stylesheets:

- VoidStylesheet assigns the same default style for every widget.
- CascadingStylesheet assigns styles to widgets using selectors, similarly to CSS.

For example, the following code customizes the style of every Label widget of the desktop:

```
CascadingStylesheet stylesheet = new CascadingStylesheet();

EditableStyle labelStyle = stylesheet.getSelectorStyle(new TypeSelector(Label.class));
labelStyle.setColor(Colors.RED);
labelStyle.setBackground(new RectangularBackground(Colors.WHITE));

desktop.setStylesheet(stylesheet);
```

Widget's Style

At any time, a widget has a style that can be retrieved using Widget.getStyle() method.

When created, the widget's style contains the default value for each field. These default values are defined in the DefaultStyle class.

Once it is attached to a desktop, the widget's style is computed from the stylesheet set in the desktop. This is done using the Stylesheet.getStyle() method. The style can then be used when laying out and rendering the widget.

At any time, the style of the widget can be recomputed by calling Widget.updateStyle(). For example when its state changes:

- When a button is pressed or released.
- When a checkbox is checked or unchecked.
- etc.

Animations

MWT provides a utility class in order to animate widgets: Animator. When a widget is being animated by an animator, the widget is notified each time that the display is flushed. The widget can use this interrupt in order to update its state and request a new rendering.

See chapter *How to Animate a Widget* for more information on animating a widget.

Partial Buffer Considerations

Rendering a widget in *partial buffer mode* may require multiple cycles if the buffer is not big enough to hold all the pixels to update in a single shot. This means that rendering is slower in partial buffer mode, and this may cause performance being significantly affected during animations.

Besides, the whole screen is flushed in multiple times instead of a single one, which means that the user may see the display at a time where every part of the display has not been flushed yet.

Due to these limitations, it is not recommended to repaint big parts of the screen at the same time. For example, a transition on a small part of the screen will look better than a transition affecting the whole screen. A transition will look perfect if the partial buffer can hold all the lines to repaint. Since the buffer holds a group of lines, a horizontal transition may not look the same as a vertical transition.

How to Create a Widget

A widget is the main way to render information on the display. A set of pre-defined widgets is described in the *Widgets* section.

If the needed widget does not already exist, it is possible to create it from scratch (or by derivating another one).

To create a custom widget, a new class should be created, extending the Widget class. Widget subclasses have to implement two methods and may override optional methods, as explained in the following sections.

Implementing the Mandatory Methods

Computing the Optimal Size of the Widget

The computeContentOptimalSize() method is called by the MWT framework in order to know the optimal size of the widget.

The optimal size of the widget is the size of the smallest possible area which would still allow to represent the widget. Unless the widget is using an OptimalDimension in its style, the actual size of the widget will most likely be bigger than the optimal size returned in this method.

The `size` parameter of the computeContentOptimalSize() method initially contains the size available for the widget. An available width or height equal to Widget.NO_CONSTRAINT means that the optimal size should be computed without considering any restriction on the respective axis. Before the method returns, the size object should be set to the optimal size of the widget.

When implementing this method, the getStyle() method may be called in order to retrieve the style of the widget.

For example, the following snippet computes the optimal size of an image widget:

```
@Override
protected void computeContentOptimalSize(Size size) {
	size.setSize(this.image.getWidth(), this.image.getHeight());
}
```

Rendering the Content of the Widget

The renderContent() method is called by the MWT framework in order to render the content of the widget.

The g parameter is used to draw the content of the widget. It is already configured with the translation and clipping area which match the widget's bounds. The contentWidth and contentHeight parameters indicate the actual size of the content of the widget (excluding its outlines). Unless the widget is using an OptimalDimension in its style, the given content size will most likely be bigger than the optimal size returned in computeContentOptimalSize(). If the drawings do not take the complete content area, the position of the drawings should be computed using the horizontal and vertical alignment values set in the widget's style.

When implementing this method, the getStyle() method may be called in order to retrieve the style of the widget.

For example, the following snippet renders the content of an image widget:

```
@Override
protected void renderContent(GraphicsContext g, int contentWidth, int contentHeight) {
	Style style = getStyle();
	int imageX = Alignment.
↪computeLeftX(this.image.getWidth(), 0, contentWidth, style.getHorizontalAlignment());
	int imageY = Alignment.
↪computeTopY(this.image.getHeight(), 0, contentHeight, style.getVerticalAlignment());
	Painter.drawImage(g, this.image, imageX, imageY);
}
```

Handling Events

When a widget is created, it is disabled and it will not receive any event. A widget may be enabled or disabled by calling setEnabled(). A common practice is to enable the widget in its constructor.

Enabled widgets can handle events by overriding handleEvent(). MicroUI event APIs may be used in order to know more information on the event, such as its type. The handleEvent() method should return whether or not the event was consumed by the widget.

For example, the following snippet prints a message when the widget receives an event:

```
@Override
public boolean handleEvent(int event) {
	System.out.println("Event type: " + Event.getType(event));
	return false;
}
```

Consuming Events

To indicate that an event was consumed by a widget, handleEvent() should return `true`. Usually, once an event is consumed, it is not dispatched to other widgets (this behavior is controlled by the event dispatcher). The widget that consumed the event is the last one to receive it.

The following guidelines are recommended to decide when to consume an event and when not to consume an event:

- If the widget triggers an action when receiving the event, it consumes the event.
- If the widget does not trigger an action when receiving the event, it does not consume the event.

Note: If the event is Pointer.PRESSED, do not consume the event unless it is required that the subsequent widgets in the hierarchy do not receive it. The Pointer.PRESSED event is special because pressing a widget is usually not the deciding factor to trigger an action. The user has to release or to drag the widget to trigger an action. If the user presses a widget and then drags the pointer (e.g. their finger or a stylus) out of the widget before releasing it, the action is not triggered.

Listening to the Life-cycle Hooks

Widget subclasses may override the following methods in order to allocate and free the necessary resources:

- onAttached()
- onDetached()
- onLaidOut()
- onShown()
- onHidden()

The `onAttached()` method may be overridden to load an image:

```
@Override
protected void onAttached() {
	this.image = ResourceImage.loadImage(this.imagePath);
}
```

Likewise, the `onDetached()` method may be overridden to close the image:

```
@Override
protected void onDetached() {
	this.image.close();
}
```

The `onLaidOut()` method may be overridden to split a String into several lines based on the widget's width:

```
@Override
protected void onLaidOut() {
	Style style = getStyle();
	Font font = style.getFont();
	this.splittedText = TextHelper.wrap(getText(), font, getContentBounds().getWidth());
}
```

The onShown() method may be overridden to start an animation:

```
@Override
protected void onShown() {
	Animator animator = getDesktop().getAnimator();
	animator.startAnimation(this);
}
```

Likewise, the onHidden() method may be overridden to stop an animation:

```
@Override
protected void onHidden() {
	Animator animator = getDesktop().getAnimator();
	animator.stopAnimation(this);
}
```

How to Create a Container

To create a custom container, a new class should be created, extending the Container class. This new class may define a constructor and setter methods in order to provide a way for the user to configure the container, such as its orientation. Container subclasses have to implement two methods and may override optional methods, as explained in the following sections.

Implementing the Mandatory Methods

This section explains how to implement the two mandatory methods of a container subclass.

Computing the Optimal Size of the Container

The computeContentOptimalSize() method is called by the MWT framework in order to know the optimal size of the container. The optimal size of the container should be big enough so that each child can be laid out with a size at least as big as its own optimal size.

The container is responsible for computing the optimal size of every child. To do so, the computeChildOptimalSize() method should be called for every child. After this method is called, the optimal size of the child can be retrieved by calling getWidth() and getHeight() on the child widget.

The Size parameter of the computeContentOptimalSize() method initially contains the size available for the container. An available width or height equal to Widget.NO_CONSTRAINT means that the optimal size should be computed without considering any restriction on the respective axis. Before the method returns, the size object should be set to the optimal size of the container.

For example, the following snippet computes the optimal size of a simple wrapper:

```
@Override
protected void computeContentOptimalSize(Size size) {
	Widget child = getChild(0);
	computeChildOptimalSize(child, size.getWidth(), size.getHeight());
	size.setSize(child.getWidth(), child.getHeight());
}
```

Laying out the Children of the Container

The layOutChildren() method is called by the MWT framework in order to lay out every child of the container, i.e. to set the position and size of the children. If a child is laid out outside the bounds of the container (partially or fully), only the part of the widget which is within the container bounds will be visible.

The container is responsible for laying out each child. To do so, the layOutChild() method should be called for every child. Before this method is called, the optimal size of the child can be retrieved by calling getWidth() and getHeight() on the child widget.

When laying out a child, its bounds have to be passed as parameter. The position will be interpreted as relative to the position of the container content. This means that the position should not include the outlines of the container. This means that the (0, 0) coordinates represent the top-left pixel of the container content and the (contentWidth-1, contentHeight-1) coordinates represent the bottom-right pixel of the container content.

For example, the following snippet lays out the children of a simple wrapper:

```
@Override
protected void layOutChildren(int contentWidth, int contentHeight) {
        Widget child = getChild(0);
        layOutChild(child, 0, 0, contentWidth, contentHeight);
}
```

Managing the Visibility of the Children of the Container

By default, when a container is shown, each of its children is shown too. This behavior can be changed by overriding the setShownChildren() method of Container. When implementing this method, the setShownChild() method should be called for each child which should be shown when the container is shown.

At any time while the container is visible, children may be shown or hidden by calling setShownChild() or setHiddenChild().

When a container is hidden, each of its children is hidden too (unless it is already hidden). It is not necessary to override setHiddenChildren(), except for optimization.

Providing APIs to Change the Children list of the Container

The Container class introduces protected APIs in order to manipulate the list of children of the container. These methods may be overridden in the container subclass and set as public in order to make these APIs available for the user.

Each of the following methods may be overridden individually:

- addChild()
- removeChild()
- removeAllChildren()
- insertChild()
- replaceChild()
- changeChildIndex()

For example, the following snippet allows the user to call the addChild() method on the container:

```
@Override
public void addChild(Widget child) {
	super.addChild(child);
}
```

How to Animate a Widget

Starting and stopping the animation

To animate a widget, an Animator instance is required. This instance can be retrieved from the desktop of the widget by calling Desktop.getAnimator(). Make sure that your widget subclass implements the Animation interface so that it can be used with an Animator.

An animation can be started at any moment, provided that the widget is shown. For example, the animation can start on a click event. Likewise, an animation can be stopped at any moment, for example a few seconds after the animation has started. Once the widget is hidden, its animation should always be stopped to avoid memory leaks and unnecessary operations.

To start the animation of the widget, call the startAnimation() method of the Animator instance. To stop it, call the stopAnimation() method of the same Animator instance.

For example, the following snippet starts the animation as soon as the widget is shown and stops it once the widget is hidden:

```
public class MyAnimatedWidget extends Widget implements Animation {

	private long startTime;
	private long elapsedTime;

	@Override
	protected void onShown() {
		// start animation
		getDesktop().getAnimator().startAnimation(this);
		// save start time
		this.startTime = Util.platformTimeMillis();
		// set widget initial state
		this.elapsedTime = 0;
	}

	@Override
	protected void onHidden() {
		// stop animation
		getDesktop().getAnimator().stopAnimation(this);
	}
}
```

Performing an animation step

The tick() method is called by the animator in order to update the widget. It is called in the UI thread once the display has been flushed. This method should not render the widget but should update its state and request a new render. The tick() method should return whether or not the animation should continue after this increment.

For example, the following snippet updates the state of the widget when it is ticked, requests a new render and keeps the animation going until 5 seconds have passed:

```
@Override
public boolean tick(long platformTimeMillis) {
	// update widget state
	this.elapsedTime = platformTimeMillis - this.startTime;
	// request new render
	requestRender();
	// return whether to continue or to stop the animation
	return (this.elapsedTime < 5_000);
}
```

The renderContent() method should render the widget by using its current state (saved in the fields of the widget). This method should not call methods such as Util.platformTimeMillis() because the widget could be rendered in multiple passes, for example if a *partial buffer* is used.

For example, the following snippet renders the current state of the widget by displaying the time elapsed since the start of the animation:

```
@Override
protected void renderContent(GraphicsContext g, int contentWidth, int contentHeight) {
	Style style = getStyle();
	g.setColor(style.getColor());
	Painter.drawString(g, Long.toString(this.elapsedTime), style.getFont(), 0, 0);
}
```

Note: Since an animator ticks its animations as often as possible, the animator may take 100% CPU usage if none of its animations requests a render. For more information on how to debug animators, see the *How to Debug Animators* section.

How to Define an Outline or Border

To create a custom outline or border, a new class should be created, extending the Outline class. Outline subclasses have to implement two methods, as explained in the following sections.

Applying the outline on an outlineable object

The apply(Outlineable) method is called by the MWT framework in order to subtract the outline from a Size or Rectangle object.

The Outlineable parameter of the method initially contains the size or bounds of the box, including the outline. Before the method returns, the outlineable object should be modified by subtracting the outline. In order to remove the outline from the object, the removeOutline() method of Outlineable should be used, passing as argument the thickness on each side.

For example, the following snippet applies an outline of 1 pixel on every side:

```
@Override
public void apply(Outlineable outlineable) {
	outlineable.removeOutline(1, 1, 1, 1);
}
```

Applying the outline on a graphics context

The apply(GraphicsContext, Size) method is called by the MWT framework in order to render the outline (only relevant if it is a border) and to update the translation and clip of a graphics context.

The Size parameter of the method initially contains the size of the box, including the outline. Before the method returns, the size object should be modified by subtracting the outline. In order to remove the outline from the object, the removeOutline() method of Outlineable should be used, passing as argument the thickness on each side.

For example, the following snippet applies an outline of 1 pixel on every side:

```
@Override
public void apply(GraphicsContext g, Size size) {
	size.removeOutline(1, 1, 1, 1);
	g.translate(1, 1);
	g.setClip(0, 0, size.getWidth(), size.getHeight());
}
```

How to Define a Background

To create a custom background, a new class should be created, extending the Background class. Background subclasses have to implement two methods, as explained in the following sections.

Informing whether the background is transparent

The isTransparent() method is called by the MWT framework in order to know whether or not the background is transparent. A background is considered as transparent if it does not draw every pixel with maximal opacity when it is applied.

For example, the following snippet informs that the background is completely opaque regardless of its size:

```
@Override
public boolean isTransparent(int width, int height) {
        return false;
}
```

Applying the background on a graphics context

The apply(GraphicsContext g, int width, int height) method is called by the MWT framework in order to render the background and to set or remove the background color of subsequent drawings.

For example, the following snippet applies a white background:

```
@Override
public void apply(GraphicsContext g, int width, int height) {
        g.setColor(Colors.WHITE);
        Painter.fillRectangle(g, 0, 0, width, height);
        g.setBackgroundColor(Colors.WHITE);
}
```

How to Create a Desktop Event Dispatcher

Creating a custom event dispatcher can help you address two use cases:

- [Dispatch] Extending an EventDispatcher is used to dispatch the events. For example, the FocusEventDispatcher will send the events to the widget owning the focus.
- [Handle] Overriding the desktop is used to directly trigger a behavior. For example “BACK” command shows the previous page.

To create a custom event dispatcher, a new class should be created, extending the EventDispatcher class. Event dispatcher subclasses have to implement a method and may override optional methods, as explained in the following sections.

Dispatching the events to the widgets

The dispatchEvent() method is called by the MWT framework in order to dispatch a MicroUI event to the widgets of the desktop. The getDesktop() method may be called in order to retrieve the desktop with which the event dispatcher is associated. This is useful in order to browse the widget hierarchy of the desktop, for example by using the getWidget() and getWidgetAt() methods of Desktop.

In order to send an event to one of the widgets of the hierarchy, the sendEventToWidget() method should be used. The dispatchEvent() method should return whether or not the event was dispatched and consumed by a widget.

For example, the following snippet dispatches every event to the widget of the desktop:

```
@Override
public boolean dispatchEvent(int event) {
	Widget desktopWidget = getDesktop().getWidget();
	if (desktopWidget != null) {
		return sendEventToWidget(desktopWidget, event);
	} else {
		return false;
	}
}
```

In addition to dispatching the provided events, an event dispatcher may generate custom events. This may be done by using a DesktopEventGenerator. Its buildEvent() method allows to build an event which may be sent to a widget using the sendEventToWidget() method.

Initializing and disposing the dispatcher

EventDispatcher subclasses may override the initialize() and dispose() methods in order to allocate and free the necessary resources.

For example, the initialize() method may be overridden to create an event generator and to add it to the system pool of MicroUI:

```
@Override
public void initialize() {
	this.eventGenerator = new DesktopEventGenerator();
	this.eventGenerator.addToSystemPool();
}
```

Likewise, the dispose() method may be overridden to remove the event generator from the system pool of MicroUI:

```
@Override
public void dispose() {
	this.eventGenerator.removeFromSystemPool();
}
```

How to Define an Extra Style Field

Extra style fields allow to customize a widget by configuring graphical elements of the widget from the stylesheet. Extra fields are only relevant to a specific widget type and its subtypes. A widget type can support up to 7 extra fields. The value of an extra field may be represented as an int , a float or any object, and it can not be inherited from parent widgets.

Defining an extra field ID

The recommended practice is to add a public constant for the ID of the new extra field in the widget subtype. This ID should be an integer with a value between 0 and 6 .

Every extra field ID has to be unique within the widget type. However, two unrelated widget types may define an extra field with the same ID.

For example, the following snippet defines an extra field for a secondary color:

```
public static final int SECONDARY_COLOR_FIELD = 0;
```

Setting an extra field in the stylesheet

The value of an extra field may be set in the stylesheet in a similar fashion to built-in style fields, using one of the setExtraXXX() methods of EditableStyle.

For example, the following snippet sets the value of an extra field for all the instances of a widget subtype:

```
EditableStyle style = stylesheet.getSelectorStyle(new TypeSelector(MyWidget.class));
style.setExtraInt(MyWidget.SECONDARY_COLOR_FIELD, Colors.RED);
```

Getting an extra field during rendering

The value of an extra field may be retrieved from the style of a widget in a similar fashion to built-in style fields, using one of the getExtraXXX() methods of Style. When calling one of these methods, a default value has to be given in case the extra field is not set for this widget.

For example, the following snippet gets the value of an extra field of the widget:

```
Style style = getStyle();
int secondaryColor = style.getExtraInt(SECONDARY_COLOR_FIELD, Colors.BLACK);
```

How to Use the Overlap Render Policy

The MWT library implements two *render policies*: the DefaultRenderPolicy and the OverlapRenderPolicy:

- DefaultRenderPolicy: renders the specified widget. If the widget is transparent, it renders its parent (and recursively).
- OverlapRenderPolicy: renders the specified widget but also the other widgets that overlap with it.

While the DefaultRenderPolicy will be fine for most GUIs, it will not handle properly the case where widgets overlap. In this case, the OverlapRenderPolicy will be the best match.

Making Widgets Overlap

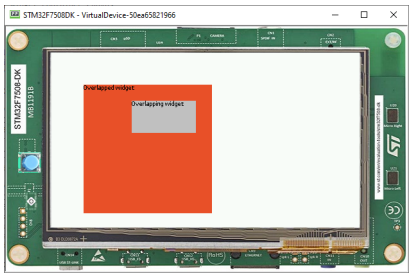
A widget is said to overlap with another when:

- their boundaries intersect
- it comes after in the widget tree (depth-first search)

The following snippet displays two widgets that overlap:

```
public static void main(String[] args) {
    MicroUI.start();

    Desktop desktop = new Desktop();

    // make two widgets overlap in a Canvas container
    Canvas rootWidget = new Canvas();
    final Button overlapped = new Button("Overlapped widget");
    rootWidget.addChild(overlapped, 50, 50, 200, 200);
    final Label overlapping = new Label("Overlapping widget");
    rootWidget.addChild(overlapping, 125, 75, 100, 50);
    desktop.setWidget(rootWidget);

    // the overlapping widget is silver
    CascadingStylesheet stylesheet = new CascadingStylesheet();
    EditableStyle style = stylesheet.getSelectorStyle(new TypeSelector(Label.class));
    style.setBackground(new RectangularBackground(Colors.SILVER));

    // the overlapped widget is orange
    style = stylesheet.getSelectorStyle(new TypeSelector(Button.class));
    style.setBackground(new RectangularBackground(0xee502e));
    desktop.setStylesheet(stylesheet);

    desktop.requestShow();
}
```

As expected from the addChild() sequence, the widget `overlapping` overlaps the widget `overlapped` :



So far, the DefaultRenderPolicy is being used and it seems to look fine: the widgets of the desktop are rendered successively in depth-first order after the call to desktop.requestShow().

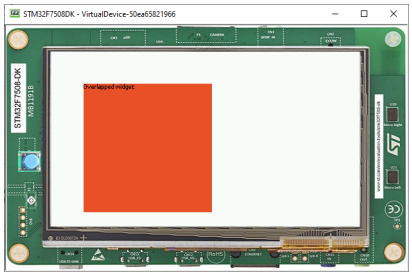
Requesting a New Render

Let's see how the DefaultRenderPolicy performs when the widget `overlapped` is requested to render again. In most cases, a widget is requested to render when its content has been updated (e.g. the value displayed has changed). For demonstration purposes, let's introduce a mean to trigger a new render: each time the user clicks on the widget `overlapped`, it will request the widget to render.

The snippet above shows how to do that, by adding an OnClickListener to the `overlapped` widget:

```
overlapped.setOnClickListener(new OnClickListener() {

    @Override
    public void onClick() {
        overlapped.requestRender();
    }
});
```

When the user clicks on the widget `overlapped`, the widget is rendered again but not the widget `overlapping`. As a consequence, the widget that overlaps is not displayed anymore:



When using the DefaultRenderPolicy, widgets are rendered regardless of their order in the widget hierarchy. However, the OverlapRenderPolicy will take account of the relative order of the other widgets: widgets that come after in the widget tree will be rendered if their boundaries intersect those of the widget.

Using the OverlapRenderPolicy

Overriding the method createRenderPolicy() of the Desktop, as follows, will cause the OverlapRenderPolicy to be used when rendering widgets:

```
Desktop desktop = new Desktop() {
    @Override
    protected RenderPolicy createRenderPolicy() {
        return new OverlapRenderPolicy(this);
    }
};
```

Now, both widgets will be displayed correctly when they are requested to render.

As a conclusion, favor the OverlapRenderPolicy when a GUI uses overlapping elements. Note that this render policy is slightly more time-consuming because it traverses the widget tree to determine which widgets are overlapping with each other.

How to Debug

Highlighting the Bounds of the Widgets

When designing a UI, it can be pretty convenient to highlight the bounds of each widget. Here are some cases where it helps:

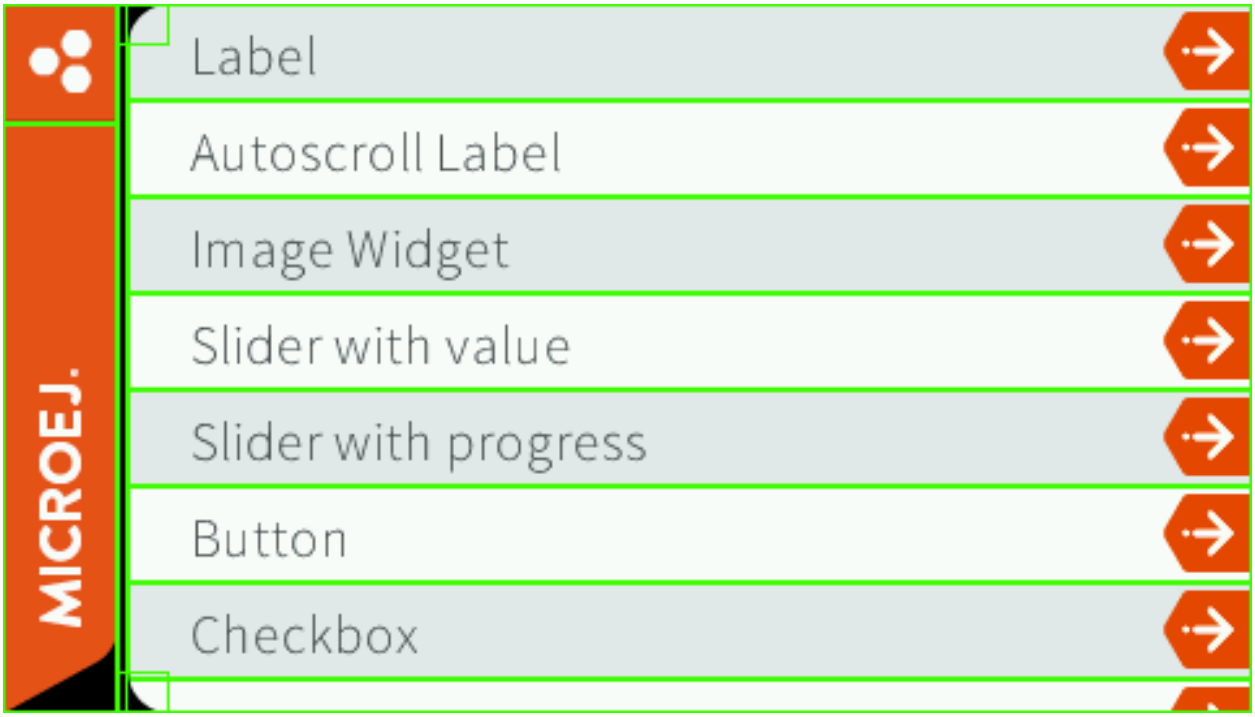
- Verify if the layout fits the expected design.
- Set the outlines (margin, padding, border).
- Check the alignment of the widget content inside its bounds.

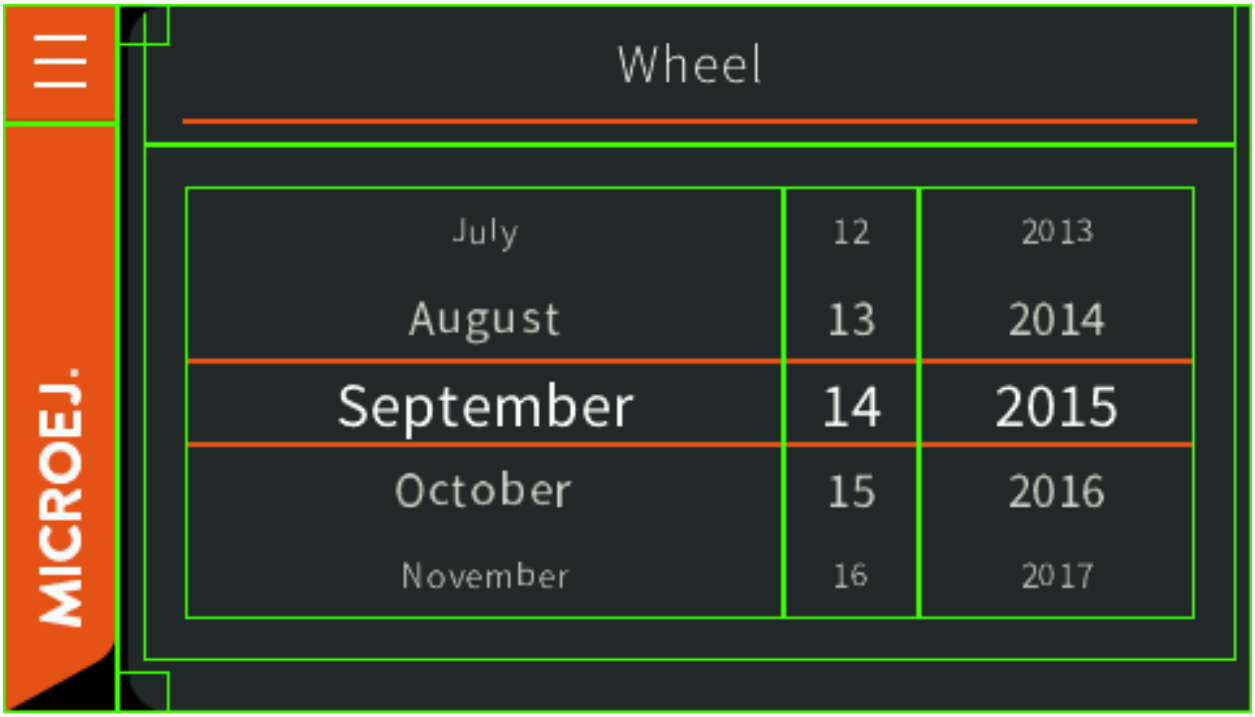
Setting the `ej.mwt.debug.bounds.enabled` constant to `true` will add a rectangle overlay over each widget and container. For more information about constants, see the *Constants* section.

By default, the rectangles around the widgets are magenta. But their color can be adjusted by modifying the `ej.mwt.debug.bounds.color` constant.

Here is an example of a `xxx.constants.list` file with the result in an application:

```
ej.mwt.debug.bounds.enabled=true
ej.mwt.debug.bounds.color=0x00ff00
```





Note: Available since MWT 3.3.0.

Enabling Traces for System View

MWT logs some keypoints when traces are enabled (see *Event Tracing*).

Here is the list of the logs:

- The creation of a widget. A unique ID is associated to the widget at this moment that will be used in the other traces all along its lifetime.
- The creation of a desktop. A unique ID is associated to the desktop at this moment that will be used in the other traces all along its lifetime.
- The request to show a desktop.
- The request to layout a widget.
- The request to layout a desktop.
- The request to render a widget. The region of the rendering request is available in the trace.
- The request to render a desktop.
- The update of the style of a widget. There is another trace for the end of the cascading update of containers.
- The compute of the optimal size of a widget. There is another trace for the end of the cascading compute of containers.
- The lay out of a widget. There is another trace for the end of the cascading lay out of containers.

- The rendering of a widget. There is another trace for the end of the cascading rendering of containers.
- When a widget is shown.
- When a widget is hidden.
- When a desktop is shown.
- When a desktop is hidden.
- When a widget is attached.
- When a widget is detached.

It is possible to print more information about the created widgets and desktops (to associate the ID with an actual object). It can be activated by setting the *Application Option* named `ej.mwt.debug.log.trace.enabled` to `true`.

The traces can be seen in SystemView. For the traces to be more human-readable, the file `SYSVIEW_MWT.txt` must be put in SystemView installation folder (e.g. `SEGGER/SystemView_V252a/Description/`). This file can be found in the `resources` folder of the jar of MWT. Otherwise, it can be created manually with the following content:

```
NamedType UIDestination *="[dest=0x%x]"

0 MWT_CreateWidget (MWT) Create a widget (ID = %u, type = %x)
1 MWT_CreateDesktop (MWT) Create a desktop (ID = %u, type = %x)
2 MWT_RequestShowDesktop (MWT) Request to show a desktop (ID = %u)
3 MWT_RequestLayout (MWT) Request to layout a widget (ID = %u)
4 MWT_RequestLayoutDesktop (MWT) Request to layout a desktop (ID = %u)
5 MWT_
↪RequestRender (MWT) Request to render a widget (ID = %u %u,%u %ux%u)
6 MWT_RequestRenderDesktop (MWT) Request to render a desktop (ID = %u)
7 MWT_UpdateStyle (MWT) Update the style␣
↪of a widget (ID = %u) | (MWT) Widget style updated (ID = %u)
8 MWT_ComputeOptimalSize (MWT) Compute the optimal␣
↪size of a widget (ID = %u %ux%u) | (MWT) Widget size computed (ID = %u)
9 MWT_Layout (MWT) Lay out a widget␣
↪(ID = %u %u,%u %ux%u) | (MWT) Widget layout done (ID = %u)
# // region seen as a container (use a START event)
10 MWT_Render (MWT) %UIDestination Render a␣
↪widget (ID = %u) region=(%u,%u)(%ux%u) | (MWT) Widget render done (ID = %u)
11 MWT_OnShown (MWT) A widget is shown (ID = %u)
12 MWT_OnHidden (MWT) A widget is hidden (ID = %u)
13 MWT_OnShownDesktop (MWT) A desktop is shown (ID = %u)
14 MWT_OnHiddenDesktop (MWT) A desktop is hidden (ID = %u)
15 MWT_OnAttached (MWT) A widget is attached (ID = %u)
16 MWT_OnDetached (MWT) A widget is detached (ID = %u)
```

Note: Available since MWT 3.6.0.

Monitoring the Render Operations

When developing a GUI application, it may not be obvious what/how exactly the UI is rendered. Especially, when a widget can be re-rendered from a distant part of the application code. Or simply because of the `RenderPolicy` used.

MWT provides a way to inject a monitor for the following render operations:

- Render requests done by the Application.
- Successive render executions triggered by the `RenderPolicy` .

Setting the `ej.mwt.debug.render.enabled` constant to `true` will enable the monitoring of above render operations. For more information about the monitoring mechanism, see RenderPolicy Javadoc. For more information about constants, see the *Constants* section.

The Widget library provides a default monitor implementation that prints the operations on the standard output. The logs produced also contain information about what is rendered (widget and area) and what code requested the rendering. For more information about this monitor implementation, see RenderMonitor Javadoc.

To use a different implementation (and if Widget is not in the classpath), set the `ej.mwt.debug.render.monitor` constant to the FQN of the monitor implementation class.

Here is an example of a `xxx.constants.list` file with the result in an application:

```
ej.mwt.debug.render.enabled=true
ej.mwt.debug.render.monitor=ej.widget.debug.RenderMonitor
```

Fig. 147: Screenshot before click

Listing 22: Application logs after click

```
rendermonitor@ INFO: Render requested␣
↪on com.microej.demo.widget.common.PageHelper$2 > SimpleDock > OverlapContainer␣
↪> SimpleDock > List > RadioButton at {0,0 87x25} of {221,116 87x25} by com.microej.
↪demo.widget.radiobutton.widget.RadioButtonGroup.setChecked(RadioButtonGroup.java:47)
rendermonitor@ INFO: Render requested␣
↪on com.microej.demo.widget.common.PageHelper$2 > SimpleDock > OverlapContainer␣
↪> SimpleDock > List > RadioButton at {0,0 87x25} of {221,166 87x25} by com.microej.
↪demo.widget.radiobutton.widget.RadioButtonGroup.setChecked(RadioButtonGroup.java:50)
rendermonitor@ INFO: Render executed␣
↪on  com.microej.demo.widget.common.PageHelper$2 > SimpleDock > OverlapContainer␣
↪> SimpleDock > List > RadioButton at {-221,-116 87x25} of {221,116 87x25}
rendermonitor@ INFO: Render executed␣
↪on  com.microej.demo.widget.common.PageHelper$2 > SimpleDock > OverlapContainer␣
↪> SimpleDock > List > RadioButton at {-221,-141 87x25} of {221,141 87x25}
rendermonitor@ INFO: Render executed␣
↪on  com.microej.demo.widget.common.PageHelper$2 > SimpleDock > OverlapContainer␣
↪> SimpleDock > List > RadioButton at {-221,-166 87x25} of {221,166 87x25}
rendermonitor@ INFO: Render executed on  com.microej.demo.widget.common.PageHelper
```

(continues on next page)

(continued from previous page)

```
↪$2 > SimpleDock > OverlapContainer > ImageWidget at {133,116 87x25} of {44,0 20x16}
rendermonitor@ INFO: Render executed on  com.microej.demo.widget.common.PageHelper$2␣
↪> SimpleDock > OverlapContainer > ImageWidget at {133,-140 87x25} of {44,256 20x16}
rendermonitor@ INFO: Render executed␣
↪on  com.microej.demo.widget.common.PageHelper$2 > SimpleDock > OverlapContainer␣
↪> SimpleDock > List > RadioButton at {-221,-116 87x25} of {221,166 87x25}
rendermonitor@ INFO: Render executed on  com.microej.demo.widget.common.PageHelper
↪$2 > SimpleDock > OverlapContainer > ImageWidget at {133,166 87x25} of {44,0 20x16}
rendermonitor@ INFO: Render executed on  com.microej.demo.widget.common.PageHelper
↪$2 > SimpleDock > OverlapContainer > ImageWidget at {133,-90 87x25} of {44,256 20x16}
```

Fig. 148: Screenshot after click

Note: Available since MWT 3.5.0 & Widget 5.0.0.

Monitoring the Animators

Since an animator ticks its animations as often as possible, the animator may take 100% CPU usage if none of its animations requests a render.

MWT provides a way to inject a monitor to be notified when none of the animations has requested a render during an animator tick.

Setting the `ej.mwt.debug.animator.enabled` constant to `true` will enable animator monitoring. For more information about constants, see the *Constants* section.

The Widget library provides a default monitor implementation which logs warning messages. The logs produced also contain information about the animations running on the animator. The Animation instances are logged using their `toString()` method, so it can be a good idea to override this method in the Animation subclasses to be able to identify them.

To use a different implementation (and if Widget is not in the classpath), set the `ej.mwt.debug.animator.monitor` constant to the FQN of the monitor implementation class.

Here is an example of a `xxx.constants.list` file with the result in an application:

```
ej.mwt.debug.animator.enabled=true
```

Listing 23: Application logs when the watchface update animation is started but it doesn't request a render

```
animatormonitor WARNING:␣
↪No render requested during animator tick. Animations list: [Watchface update animation]
```

Note: Available since MWT 3.5.0 & Widget 5.0.0.

Finding which Rule Originates the Attributes of a Widget's Style

The method CascadingStylesheet.getStyleSources() is able to retrieve the selectors used to create a Style (that has been originated from a CascadingStylesheet). It requires to set the constant ej.mwt.debug.cascadingstyle.enabled to true .

The result of this method is an array containing 16 selectors: one for each parameter of the style. The indices for each style entry are available as constants in the CascadingStylesheet class. For each entry, the selector belongs to the rule selected to fill the matching parameter (thus set by getSelectorStyle()). A null entry means that the parameter is from the default style (thus set by getDefaultStyle()).

This feature is used in the *HierarchyInspector* of the Widget library.

Warning: Beware that enabling that feature may downgrade the performances (more time to compute a style and more Managed Heap used).

Note: Available since MWT 3.6.0.

Detecting Text Overflow

Widgets that display a text may experience text overflow when the strings are too long to fit into the available area. It can be the case, for example, in applications that support multiple languages because widgets have to deal with texts of different lengths.

This document presents a solution to detect such text overflows.

Instrumenting the Widget

The goal is to check whether the text to be displayed is within the content bounds of the widget. A way to test this is to extend or modify the widget. In this article, the widget MyLabel will extend the type Label from the Widget library, which displays a text:

```
import ej.widget.basic.Label;

public class MyLabel extends Label {

    public MyLabel(String text) {
```

(continues on next page)

(continued from previous page)

```
        super(text);
    }
}
```

Overriding the onLaidOut() Method

Once the position and size of a wigdet are set during the lay out process, the onLaidOut() method is called to notify the widget. Overriding onLaidOut() of class MyLabel is a good place to check whether the text overflows or not.

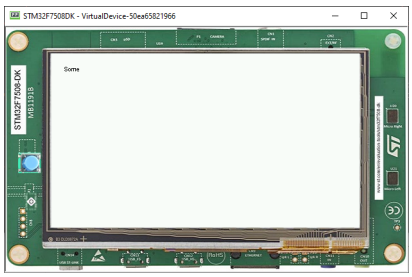
For example, the following snippet compares the text width with the available width: it will print a message if an overflow is detected.

```
@Override
protected void onLaidOut() {
    super.onLaidOut();

    // compute the width of the text with the specified font
    final Font font = getStyle().getFont();
    final String text = getText();
    final int textWidth = font.stringWidth(text);

    // compare to the width available for the content of the widget
    final int contentWidth = getContentBounds().getWidth();
    if (textWidth > contentWidth) {
        System.out.println("Overflow detected:\n > Text: \"
↪" + text + "\"\n > Width = " + textWidth + " px (available: " + contentWidth + " px)");
    }
}
```

Testing

Here is a case where the widget size is set manually to be a little shorter than the text width:

```
public static void main(String[] args) {
    MicroUI.start();
    Desktop desktop = new Desktop();
    Canvas canvas = new Canvas();
    // add a label with an arbitrary fixed width of 25 pixels (which is too short)
    canvas.addChild(new MyLabel("Some text"), 20, 20, 25, 10);
    desktop.setWidget(canvas);
    desktop.requestShow();
}
```



The text is cropped and the console logs that a text overflow has been detected:

```
================ [ Initialization Stage ] ================
================ [ Converting fonts ] ================
================ [ Converting images ] ================
================ [ Launching on Simulator ] ================
Overflow detected:
 > Text: "Some text"
 > Width = 47 px (available: 25 px)
```

Improving the Detection

To ease the correction process, it is best to add some additional debug information to locate the issue. Let's extract the text overflow detection into a helper class, so that it is available for all classes across the application.

The following snippet:

- extracts the text overflow detection into the class `MyTextHelper`.
- prints the part of the text that is displayed.
- prints the path to the widget in the widget tree to help the tester locate the affected widget in the GUI.

```
public class MyLabel extends Label {

    public MyLabel(String text) {
        super(text);
    }

    @Override
    protected void onLaidOut() {
        super.onLaidOut();

        final Font font = getStyle().getFont();
        final String text = getText();
        MyTextHelper.checkTextOverflow(this, text, font);
    }
}

public class MyTextHelper {

    /**
    * Checks whether␣
↪the given text overflows for the specified widget and font. In the case where an overflow is
    * detected, the method prints a message that details the error.
    *
    * @param widget
    *          the widget that displays the text.
    * @param text
    *          the text to display.
    * @param font
    *          the font used for drawing the text.
    */
    ␣
↪public static void checkTextOverflow(final Widget widget, final String text, final Font font) {
```

(continues on next page)

(continued from previous page)

```
        final int textWidth = font.stringWidth(text);
        final int contentWidth = widget.getContentBounds().getWidth();

        if (textWidth > contentWidth) {
            String displayedText = buildDisplayedText(text, font, contentWidth);
            String widgetPath = buildWidgetPath(widget);
            System.out.println(
                    "Overflow␣
↪detected:\n > Text: \"" + text + "\"\n > Width = " + textWidth + " px (available: "
                            + contentWidth␣
↪+ " px) \n > Displayed: \"" + displayedText + "\"\n > Path : " + widgetPath);
        }
    }

    private static String buildDisplayedText(String text, Font font, int width) {
        for (int i = text.length() - 1; i > 0; i--) {
            if (font.substringWidth(text, 0, i) <= width) {
                return text.substring(0, i);
            }
        }

        return "";
    }

    private static String buildWidgetPath(Widget widget) {
        StringBuilder builder = new StringBuilder();

        Widget ancestor = widget;
        do {
            builder.insert(0, " > " + ancestor.getClass().getSimpleName());
            ancestor = ancestor.getParent();
        } while (ancestor != null);
        builder.insert(0, widget.getDesktop().getClass().getSimpleName());

        return builder.toString();
    }
}
```

When the application is launched again, the console shows more information about the text overflow:

```
================ [ Initialization Stage ] ================
================ [ Converting fonts ] ================
================ [ Converting images ] ================
================ [ Launching on Simulator ] ================
Overflow detected:
 > Text: "Some text"
 > Width = 47 px (available: 25 px)
 > Displayed: "Some"
 > Path : Desktop > Canvas > MyLabel
```

To keep control over the extra verbosity and code size, one option is to use *BON constants* to enable/disable this debug code at will. In the following snippet, when the constant `com.mycompany.check.text.overflow` is set to `false`, the debug code will not be embedded in the application.

```
public static void checkTextOverflow(final Widget widget, final String text, final Font font) {
   if (Constants.getBoolean("com.mycompany.check.text.overflow")) {
      final int textWidth = font.stringWidth(text);
      final int contentWidth = widget.getContentBounds().getWidth();

      if (textWidth > contentWidth) {
         String displayedText = buildDisplayedText(text, font, contentWidth);
         String widgetPath = buildWidgetPath(widget);
         System.out.println(
               "Overflow␣
↪detected:\n > Text: \"" + text + "\"\n > Width = " + textWidth + " px (available: "
                     + contentWidth␣
↪+ " px) \n > Displayed: \"" + displayedText + "\"\n > Path : " + widgetPath);
      }
   }
}
```

MWT Examples

The MWT Examples are code samples that show how to implement various use cases with MWT.

Because the MWT toolkit is designed to be compact and customizable, it allows for many possibilities when developing a GUI. Thus, the examples can be used, with or without modifications, to extend and customize the MWT framework for your specific needs. They also help to learn the best practices for the development of graphic interfaces with MWT.

Source

To get the source code of these examples, clone the following GitHub repository: https://github.com/MicroEJ/ExampleJava-MWT.

The repository contains several Gradle projects (one project for each example) that can be imported in your favorite IDE.

For each project, please refer to its `README.md` file for more details about the example and its usage.

Provided Examples

Attribute Selectors

This example shows how to customize the style of widgets using attribute selectors, similar to CSS Attribute Selectors.

It provides several types of attribute selectors, any of which can be used in a stylesheet to select widgets based on custom attributes. In this case, the background color of a label switches depending on the value of an attribute of the label.

Buffered Image Pool

This example shows how to use a pool of BufferedImages to share them across an application.

In this example, there is one image in the pool, which is shared between the histogram widget and the transition container.

Context-Sensitive Container

This example shows a smartwatch application that looks different depending on whether the user is wearing the device on the left arm or on the right arm.

It demonstrates how a container can adapt to the context by changing how its children are laid out: in this case, depending on the wrist mode, the widgets are displayed on either the left or right side. For demonstration purposes, the example displays a virtual watch to simulate the device flip.

Drag'n'Drop

This example shows how to implement drag'n'drop support in a grid.

Focus

This example shows how to introduce focus management in a project when using peripherals like buttons or a joystick.

The virtual joystick (on the right) is used to simulate a hardware joystick. When the joystick directions (up, down, left, right) are pressed, the focus changes on the items in the same way as when using the touch pointer.

Immutable Stylesheet

This example shows how to create and use an immutable stylesheet.

The immutable stylesheet resolves the style for a widget with the same algorithm as the cascading stylesheet. The difference is that the immutable stylesheet is described in an immutable file instead of Java code. Therefore, the style objects are allocated in flash instead of the Managed Heap.

Lazy Stylesheet

This example shows how to create and use a “lazy” stylesheet.

The lazy stylesheet resolves the style for a widget with the same algorithm as the cascading stylesheet. The difference is that the lazy stylesheet associates style factories with selectors (rather than style instances). As a result, the style elements are allocated “on demand” when a rule’s selector applies to a widget.

Masking Grid

This example shows how to mask a widget temporarily.

The grid is a custom container (MaskingGrid) that exposes an API to change the visibility of its children (visible or invisible). When requested to render, the grid only renders the children marked as visible.

MVC

This example shows how to create and use an MVC design pattern (Model, View, Controller).

The value of the model can be changed by clicking on the physical button.

It is also possible to resize all the widgets at once.

Popup

This example shows how to show a popup in an application.

Two types of popups are illustrated. The information popup can be dismissed by clicking outside of its bounds. The action popup needs the user to click on a button to close it.

Remove Widget

This example shows how to add and remove widgets in a widget hierarchy.

The layout adapts automatically to the number of items because requestLayout() is called for each addition/deletion on the container.

Slide Container

This example shows a slide container. This is a container that shows only its last child.

An animation is done when adding/removing a child by translating the widgets from/to the right.

Stack Container

This example shows a stack container. This is a container that stacks its children on top of each other.

An animation is done when adding/removing a child by translating the widget from/to the right.

Stashing Grid

This example shows how to stash a widget temporarily.

The grid is a custom container (StashingGrid) that exposes an API to change the visibility of its children (visible or invisible). When requested to lay out, the grid only lays out the children marked as visible. When requested to render, the grid only renders the children marked as visible.

Theming and Branding

This example shows how to create theming and branding for your project.

The application contains only one page.

There are two different types of theming shown:

1. Changing from normal to condensed by passing a *Theme* when building the stylesheet.
2. Changing the styling (including padding, margin, background, etc.) itself with a *StyleTheme*.

Transition

This example shows a container that triggers effects during page transitions.

The effect applied to the transition container can be changed dynamically. New effects can be developed easily.

Virtual Watch

This example shows how to simulate the skin and inputs of a device with a different device (e.g., an evaluation board).

This can be a convenient option when the target hardware is not yet available.

Here, it simulates a watch with a round screen and 3 buttons. The actual application is shown in a round area of the screen and receives events from the virtual buttons. The virtual buttons send commands when clicked, the same way a target device would have sent events from the native world. The goal is to be able to migrate the application on the target device without modifying the application code.

Widgets

The Widget library provides very common widgets with basic implementations. These simple widgets may not provide every desired feature, but they can easily be forked since their implementation is very simple.

Usage

To use the widgets provided by the widget library, add Widget library module to the project build file:

Gradle (build.gradle.kts)

MMM (module.ivy)

```
implementation("ej.library.ui:widget:4.1.0")
```

```
<dependency org="ej.library.ui" name="widget" rev="4.1.0"/>
```

To fork one of the provided widgets, duplicate the associated Java class from the widget library JAR into the source code of your application. It is recommended to move the duplicated class to an other package and to rename the class in order to avoid confusion between your forked class and the original class.

Provided Widgets

Widgets:

- Label: displays a text.
- ImageWidget: displays an image which is loaded from a resource.
- Button: displays a text and reacts to click events.
- ImageButton: displays an image which is loaded from a resource and reacts to click events.

Containers:

- List: lays out any number of children horizontally or vertically.
- Flow: lays out any number of children horizontally or vertically, using multiple rows if necessary.

- Grid: lays out any number of children in a grid.
- Dock: lays out any number of children by docking each child one by one on a side.
- SimpleDock: lays out three children horizontally or vertically.
- OverlapContainer: lays out any number of children by stacking them.
- Canvas: lays out any number of children freely.

Color Utilities

The widget library offers some color utilities.

The ColorHelper is helpful for decomposing colors into components (alpha, red, blue, green) and building back a color from components.

The GradientHelper can blend two colors and create a gradient from two colors.

The resulting gradient is a list of distinct colors from the start color to the end color. The colors are truncated to the display color depth. As a consequence, for the same start and end colors, a gradient created for a 32-bit display will contain more colors than on a 16-bit display.

The LightHelper proposes several primitives concerning the luminance of the colors. The luminance of a color is computed from the luminance and the quantity of each of its components. The green being the brighter, then the red and finally the blue.

Debug Utilities

A few utilities useful for debugging are available in the package `ej.widget.util.debug` of the widget library.

Print the Hierarchy of Widgets

The method HierarchyInspector.hierarchyToString(Widget) returns a String representing the hierarchy of a widget. In other words, it prints the widget and its children recursively in a tree format.

It may be used to analyse the content of a page and have a quick estimation of the number of widgets and the depth of the hierarchy.

For example:

```
Scroll
+--ScrollableList
|  +--Label
|  +--Dock
|  |  +--ImageWidget
|  |  +--Label
|  +--Label
```

The method HierarchyInspector.hierarchyStyleToString(Widget) returns the same information plus the list of the attributes of the style of each widget in the hierarchy. It only adds the attributes that differ from the default style.

For example:

```
Scroll (background=RectangularBackground, horizontalAlignment=hcenter)
+--ScrollableList (background=RectangularBackground)
|  +--Label (color=white, font=RasterFont[SourceSansPro_
↪19px-300], horizontalAlignment=hcenter, verticalAlignment=vcenter)
|  +--Dock (background=NoBackground)
|  |  +--ImageWidget (horizontalAlignment=hcenter, verticalAlignment=vcenter)
|  |  +--Label (color=0xf8a331, font=RasterFont[SourceSansPro_
↪19px-300], horizontalAlignment=hcenter, verticalAlignment=vcenter)
|  +--Label (color=white, font=RasterFont[SourceSansPro_
↪19px-300], horizontalAlignment=hcenter, verticalAlignment=vcenter)
```

If the constant ej.mwt.debug.cascadingstyle.enabled is set (see *this section*), it will also print the selector that originated each style's attribute.

For example:

```
Scroll (background=RectangularBackground␣
↪from default, horizontalAlignment=hcenter from .class 896560)
+--ScrollableList (background=RectangularBackground from default)
|  +--Label (color=white from .class␣
↪896572, background=NoBackground from .class 896572, font=RasterFont[SourceSansPro_
↪19px-300] from .class 896572, horizontalAlignment=hcenter␣
↪from .class 896572, verticalAlignment=vcenter from .class 896572)
|  +--Dock (background=NoBackground from default)
|  |  +--ImageWidget (horizontalAlignment=hcenter␣
↪from .class 896571, verticalAlignment=vcenter from .class 896571)
|  |  +--Label␣
↪(color=0xf8a331 from .class 896572 and :nth-child(odd), font=RasterFont[SourceSansPro_
↪19px-300] from .class 896572, horizontalAlignment=hcenter␣
↪from .class 896572, verticalAlignment=vcenter from .class 896572)
|  +--Label (color=white from .class 896572, font=RasterFont[SourceSansPro_
↪19px-300] from .class 896572, horizontalAlignment=hcenter␣
↪from .class 896572, verticalAlignment=vcenter from .class 896572)
```

Print the Path to a Widget

The method HierarchyInspector.pathToWidgetToString(Widget) returns a String representing the list of ancestors of the widget. For example: Desktop > Scroll > ScrollableList > Label .

It may be used to identify a widget in a trace.

It is also possible to choose the separator by using HierarchyInspector.pathToWidgetToString(Widget, char) method. For example: Desktop ; Scroll ; ScrollableList ; Label .

Count the Number of Widgets or Containers

The methods HierarchyInspector.countNumberOfWidgets(Widget) and HierarchyInspector.countNumberOfContainers(Widget) respectively count the number of widgets and containers in a hierarchy.

It may be used to evaluate the complexity of a hierarchy of widgets.

Count the Maximum Depth of a Hierarchy

The method HierarchyInspector.countMaxDepth(Widget) counts the maximum depth of a hierarchy. In other words, the depth of the widget with the biggest number of parents recursively.

It may be used to evaluate the complexity of a hierarchy of widgets.

Print the Bounds of a Widget

The method BoundsInspector.boundsToString(Widget) returns a String with the widget type and its bounds. The returned String contains:

- the simple name of the class of the widget,
- its position relative to its parent,
- its size,
- its absolute position.

For example: `Label: 0,0 7x25 (absolute: 75,75)`

Print the bounds of all the widgets in a hierarchy

The method BoundsInspector.boundsRecursiveToString(Widget) returns a String representing the type and bounds of each widget in the hierarchy of a widget.

For example:

```
Scroll: 0,0 480x272 (absolute: 0,0)
+--ScrollableList: 0,0 480x272 (absolute: 0,0)
|  +--Label: 0,0 480x50 (absolute: 0,0)
|  +--Dock: 0,50 480x50 (absolute: 0,50)
|  |  +--ImageWidget: 0,0 70x50 (absolute: 0,50)
|  |  +--Label: 70,0 202x50 (absolute: 70,50)
|  +--Label: 0,100 480x50 (absolute: 0,100)
```

Widget Examples

The Widget Examples provides some widget implementations as well as usage examples for these widgets and for the widgets of the Widget library. The widgets and usage examples are intended to be duplicated by the developers in order to be adapted to their use-case.

Source

To use the widgets provided by the widget examples, clone the following GitHub repository: https://github.com/MicroEJ/Example-Java-Widget.

The repository contains only one Gradle project that can be imported in your favorite IDE.

Each subpackage contains the source code for a specific widget and for a page which showcases the widget. For example, the `com.microej.demo.widget.checkbox` package contains the `Checkbox` widget and the `CheckboxPage` .

Provided Widgets

The showcased widgets are listed in the README of the project.

Simulation

The following sections present the options of the Front Panel and some tooling and tips to help debugging and optimizing an application.

Front Panel Overview

The *Front Panel Mock* is provided by the VEE Ports.

It is especially useful for those exposing a MicroUI display, LEDs and input devices. It then provides an interactive window for the Application simulation.

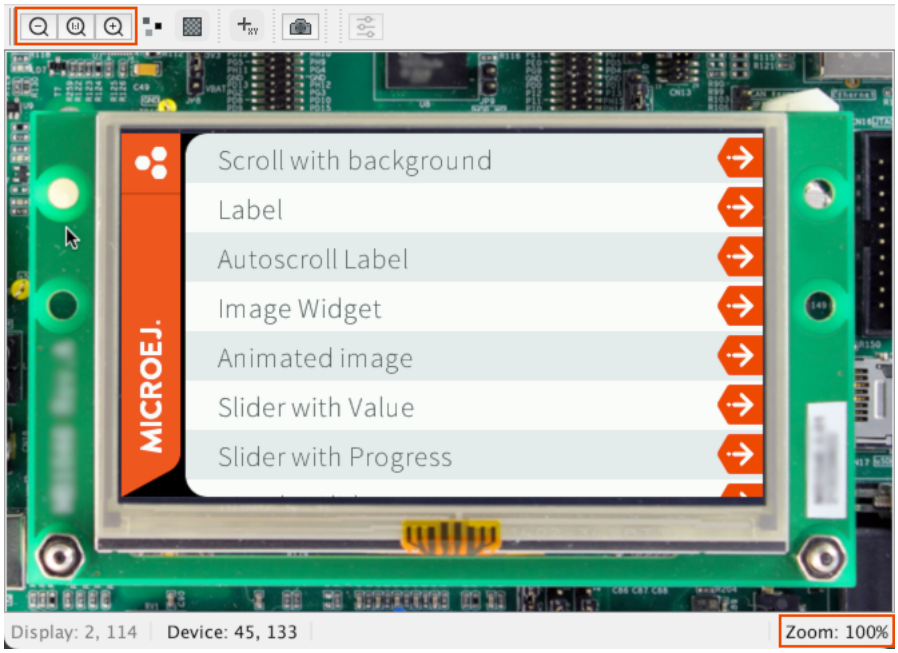
The window contains a toolbar, the device image and a status bar. It is extensible, so VEE ports can add custom controls and informations.

Following is the list of features available as standard in the UI Pack.

Zoom

The Front Panel is able to zoom in or out the represented device.

The current value of the zoom is printed in the status bar.



There are three buttons in the toolbar to change the zoom:

- Zoom out by increment of 10%.
- Reset the zoom to 100%.

- Zoom in by increment of 10%.

  The zoom can also be updated when the window size is changed when the *Fit* option is set.

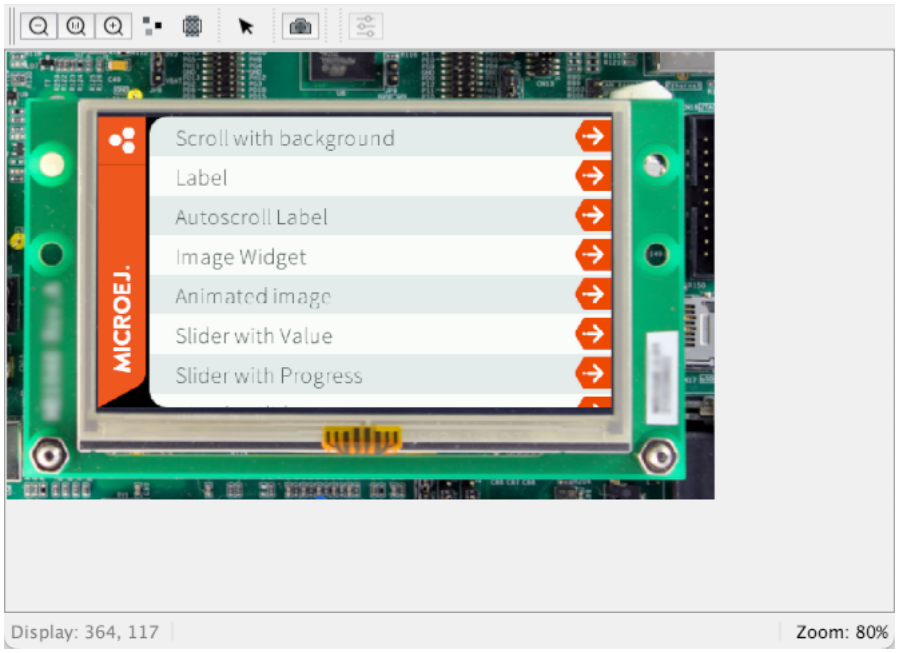
Interpolation

By default, the zoom is done without interpolation to ease the reading of the pixels drawn on the screen. But it could be convenient to enable the interpolation when a great or small zoom is applied, to better read the strings for instance.

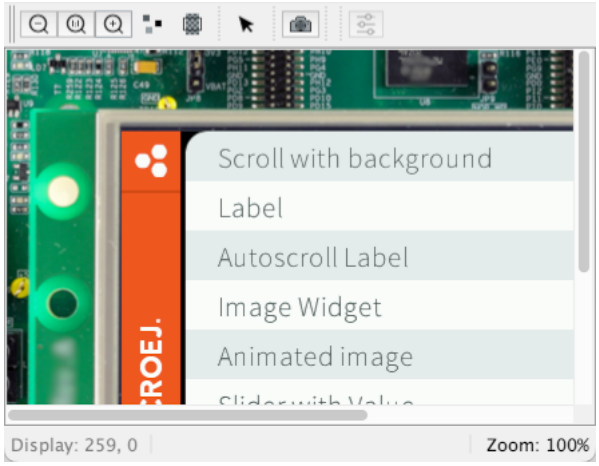
Here is an example of the same content zoomed out with and without interpolation:

Fit

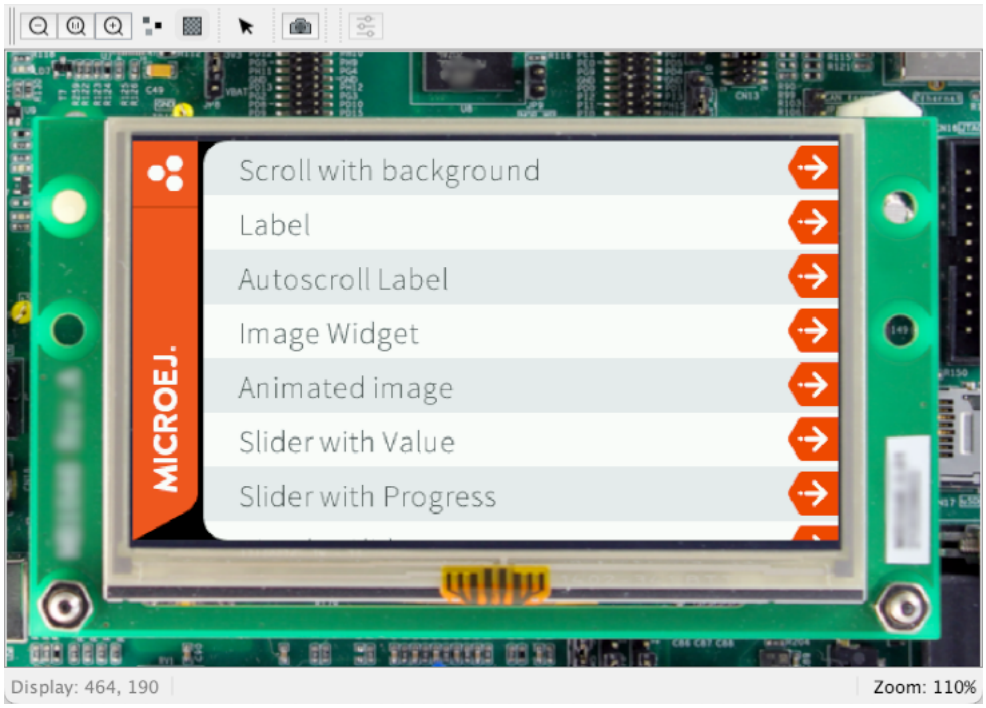
By default, the zoom and the window size are not related . That means that when the zoom is changed the window size does not change and scrollbars may appear to navigate in the device.

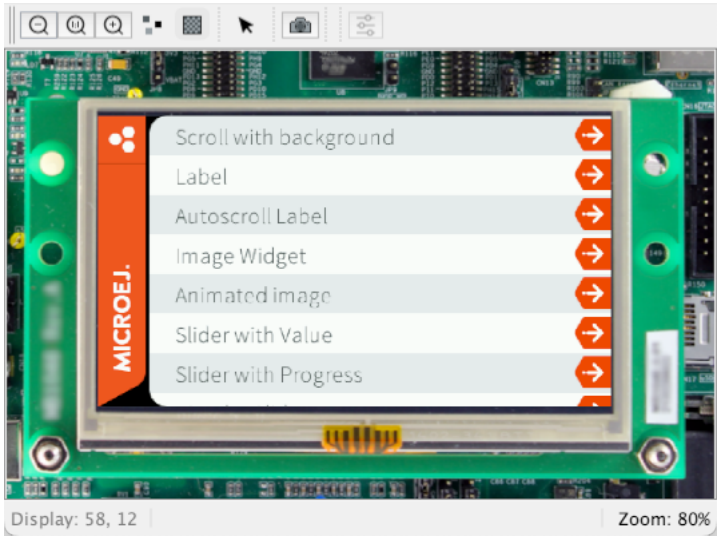


And the other way around, when the window size is changed, the zoom does not change.



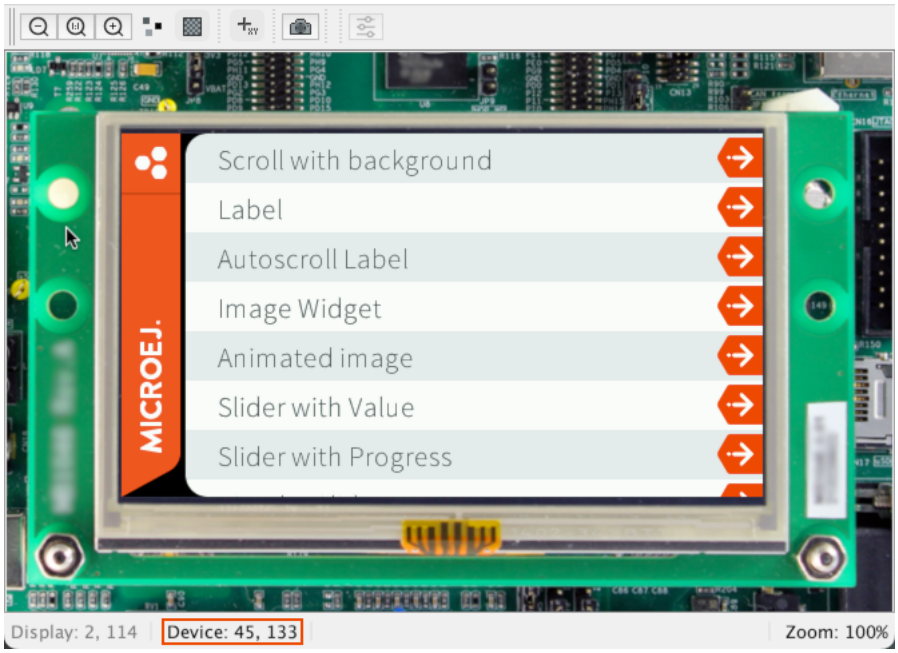
In contrast, the zoom and the window can be linked together . In this case, a modification of the zoom or the window size have an impact on the other. It is important to note that in this case, some zooms may not be possible since the window is bound to the display size.





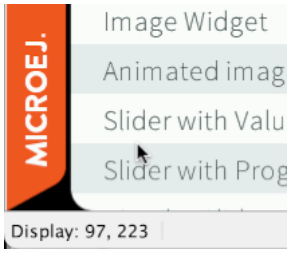
Device Coordinates

It is possible to print the coordinates of the device under the cursor in the status bar. It may be convenient for the development of the Front Panel project of a VEE Port. It is not activated by default .



Display Coordinates

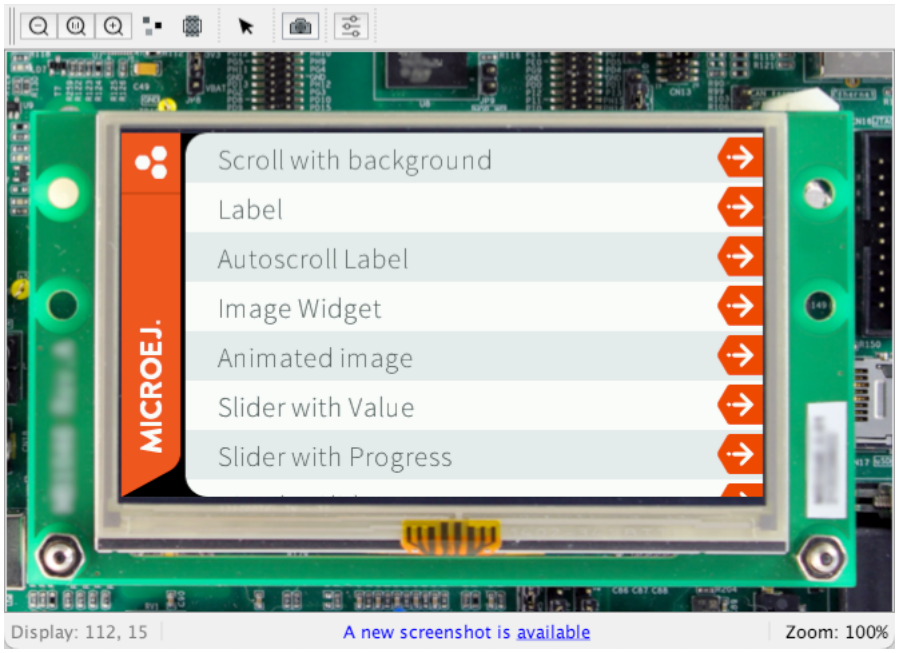
When the device contains a display, the display coordinates under the cursor are printed in the status bar.



Display Screenshot

When the device contains a display, a new button is available in the toolbar that allows to make a screenshot of the current content of the display. The screenshots are saved in the subfolder `screenshots` of the *application output folder*. Their name contains the date and time they are shot at.

When a screenshot is made, a notification appears in the status bar. Clicking on the notification opens the folder containing the screenshots.



Refresh Strategy

Some buttons enable (or disable) some debug information in relation with the *Buffer Refresh Strategy*; see *Refresh Strategy Highlighting*.

Flush Visualizer

Some buttons allow to record and visualize the drawings made between two flushes; see *Flush Visualizer*.

Front Panel Options

Cache

The configuration of the Front Panel is saved and restored for the next launch. This configuration could be reset by deleting the *.cache/frontpanelstate.properties* in the application output folder.

The saved properties are: the *zoom*, the *fit*, the *interpolation*, and the *device coordinates*.

Properties

The configuration of the Front Panel can be configured through properties.

Some of the properties are used by the cache. Defining these properties will overrule the cache. In other words, the cache will not be used for these properties.

However, it is possible to define a default value for the very first launch of the Front Panel (or when the cache has been deleted). It can be done by defining a property with the same name but with the .default suffix.

Please refer to the dedicated sections (*SDK 6* or *SDK 5*) to know how to define options.

Table 35: Front Panel Options

Option	Chapter	Aim	Cached
ej.fp.project	*Installation*	Specify a local Front Panel project to avoid rebuilding VEE Port. Only available before Architecture 8 and before UI Pack 14.3.0.	false
ej.fp.hil	*Classpath*	Run the Front Panel in the same Java SE VM as the standard mocks.	false
ej.fp.brs.drawnColor	*Drawn Region(s)*	Identify the drawn regions for each frame.	false
ej.fp.brs.restoredColor	*Restored Region(s)*	Identify the restored regions for each frame.	false
ej.fp.brs.dirtyColor	*Dirty Region(s)*	Identify the regions not fully filled by the drawings.	false
ej.fp.zoom	*Zoom*	Set the zoom of the device at startup.	true (ej.fp.zoom.default)
ej.fp.zoom.fit	*Fit*	Set whether the device is fitted to the Front Panel bounds at startup. In other words if the zoom changes when the user resizes the frame or the other way around.	true (ej.fp.zoom.fit.default)
ej.fp.zoom.interpolate	*Interpolation*	Set whether the device is drawn with interpolation when zoomed in or out at startup.	true (ej.fp.zoom.interpolate.default)
ej.fp.frame.coordinates	*Device Coordinates*	Set the location of the Front Panel frame at startup.	true (ej.fp.frame.coordinates.default)

Flush Visualizer

Presentation

Building smooth and visually appealing UI applications requires a keen focus on performance. To achieve efficient UI rendering, minimizing unnecessary work that consumes valuable CPU time is essential.

For example, assuming the application targets 60 FPS to perform a transition between two screens, that means the application has 1/60s ~= 16ms in total to execute the rendering and the flush (see *Rendering Pipeline*).

The Flush Visualizer is a tool designed to investigate potential performance bottlenecks in UI applications running on the Simulator. It uses the *Debug Traces* to record the drawings whose target is the display (since UI Pack 14.4.1, see *Installation*). All other traces (drawings in buffered images and third-party traces of the other libraries) are dropped.

The Flush Visualizer provides the following information:

- A timeline with a step for each flush.
- A screenshot of what is shown on the display at flush time.
- The list of what is done before this flush (and after the previous one) organized as a tree.
- A node of the tree can be either a region (the display or a clip) or a drawing operation.
- Each region defines its bounds and can contain one or several other nodes.
- Each region also displays the percentage of their parent region they cover.
- Some of the drawings can compute the percentage of their parent region they cover.
- The others are either negligeable or unknown.
- At the end of each region, there is a summary of the percentage covered by the drawings in this region (recursively).

Consider this part of the tree:

```
|_ UI_SetClip #10 dest=0x0 region=(0,0)(44x46)  [16.91% of UI_SetClip #9]
| |
| |_ MWT_Render #11 (MWT) [dest=0x0]
↪Render a widget (ID = 61) region=(0,0)(44x46)          [100.00% of UI_SetClip #10]
| | |_ UI_SetForegroundColor dest=0x0 color=0xffee502e
| | |
| | |_ UI_FillRectangle dest=0x0 anchor=0,0 size=44x46
| | | |_ UI_DrawnRegion
↪dest=0x0 clipDisabled=true region=(0,0)(44x46) [100.00% of MWT_Render #11]
| | |_ UI_FillRectangle  Drawing operation done
| | |_ UI_SetForegroundColor dest=0x0 color=0xffcf4520
| | |
| | |_ UI_FillRectangle dest=0x0 anchor=0,44 size=44x2
| | | |_ UI_DrawnRegion
↪dest=0x0 clipDisabled=true region=(0,44)(44x2) [4.35% of MWT_Render #11]
| | |_ UI_FillRectangle  Drawing operation done
| | |
```

(continues on next page)

(continued from previous page)

```
| | |_ UI_SetClip #12 dest=0x0 region=(0,0)(44x44) [95.65% of MWT_Render #11]
| | | |
| | | |_ UI_SetClip #13 dest=0x0 region=(10,10)(24x24) [29.75% of UI_SetClip #12]
| | | | |_ UI_SetForegroundColor dest=0x0 color=0xffffffff
| | | | |
| |␣
↪| | |_ UI_DrawImage dest=0x0 image=0x3 from 0,0 size=24x24 anchor=10,10 alpha=255
| | | | | |_ UI_DrawnRegion␣
↪dest=0x0 clipDisabled=true region=(10,10)(24x24) [100.00% of UI_SetClip #13]
| | | | |_ UI_DrawImage Drawing operation done
| | | | [100.00% drawn of UI_SetClip #13]
| | | [29.75% drawn of UI_SetClip #12]
| | [132.81% drawn of MWT_Render #11]
| |_ MWT_Render (MWT) Widget render done (ID = 61)
| [132.81% drawn of UI_SetClip #10]
```

We can see that:

- The MWT_Render #11 is 100% of the clip #10, which is 16.91% of the clip #9.
- It draws 2 rectangles and an image that take respectively 100%, 4.35% and 100% of their parent region.
- The clip #13 is fully covered by the image (100%).
- The clip #10 is covered at 132.81%: 100% for the 1st rectangle, 4.35% for the 2nd, 28.46% for the clip #12 and its inner image (29.75 x 95.65%)

A value of 100% indicates that the area drawn is equivalent to the surface of the region. A value of 200% indicates that the area drawn is equivalent to twice the surface of the region. A perfect application has 100% of its root region drawn but its very unlikely for an application that draws anything else than a rectangle or an image. A total area drawn between 100% to 200% is the norm in practice because widgets often overlap. However, if the total area drawn is bigger than 200%, that means that the total surface of the region was drawn more than twice. Probably meaning that a lot of drawings are done above others. Identifying this drawing (and the ones below) can help reducing the number of drawings done (or their surface).

As always, when conducting a performance study, measure. Use *SystemView* to identify the bottlenecks in your application on the embedded target. A total area drawn over 200% is inefficient, but your application may have bigger bottlenecks. Confirm it by measuring the time spent drawing vs. the time spent elsewhere between flushes.

Installation

The Flush Visualizer option is available for the `Display` widget in frontpanel widget module version 4.+ for UI Pack 14.0.0 or later.

Since UI Pack 14.4.1

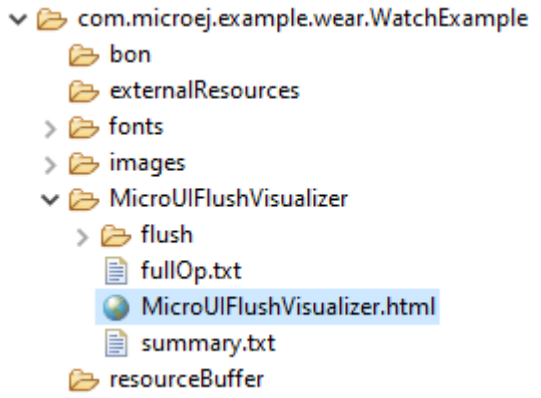
Before UI Pack 14.4.1

- Set the property `core.trace.enabled` to `true` to enable the Flush Visualizer.
- Set the property `core.trace.autostart` to `true` to start the recording on startup.
- Start and pause the recording by clicking on the button ●.
- Set the property `ej.fp.display.flushVisualizer` to `true` to enable the Flush Visualizer.

Refer to the *Standalone Application Options* documentation to set the option.

Usage

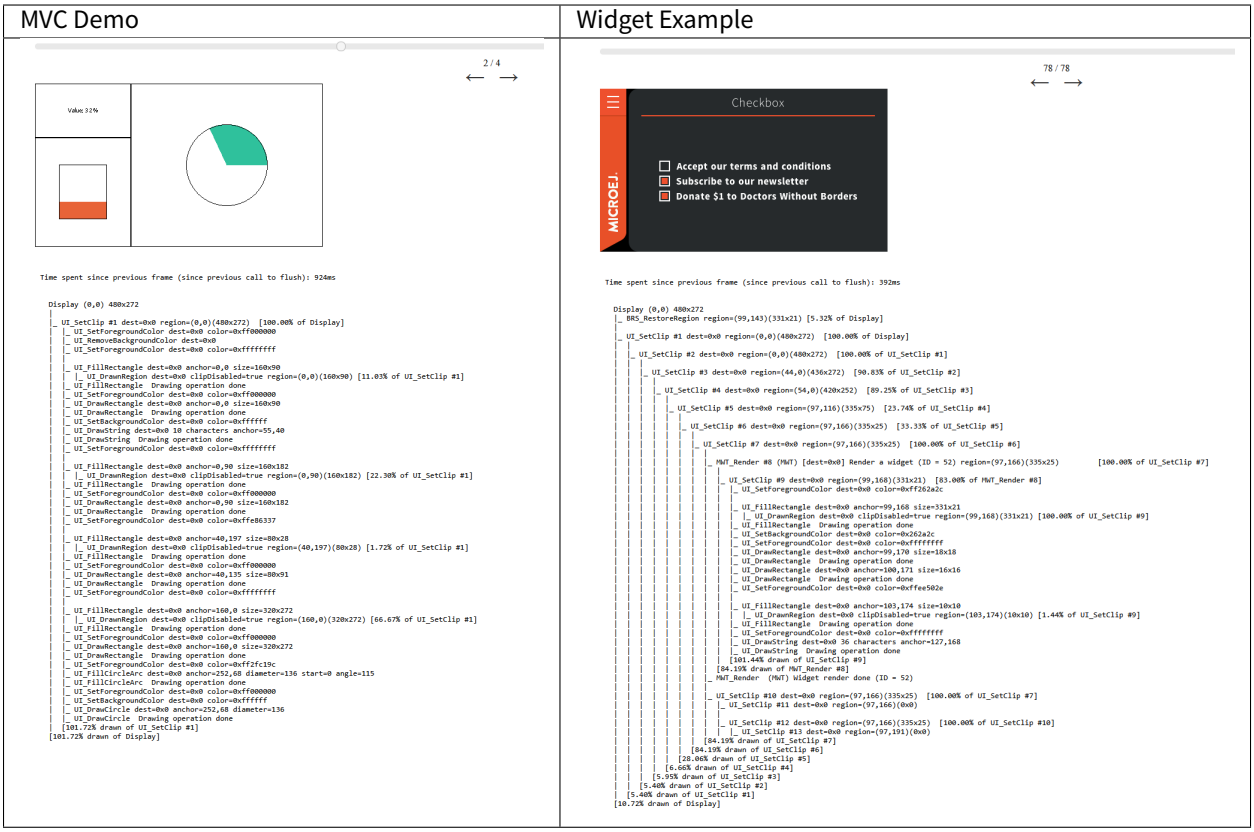
1. Run the application in the Simulator.
2. Since UI Pack 14.3.0, the button in the toolbar of the Front Panel opens the Flush Visualizer.
3. The file `MicroUIFlushVisualizer/MicroUIFlushVisualizer.html` is generated in the *application output folder* and can be opened during or after the execution.



Note: Since MICROEJ SDK 6, the application output folder is located under the `build/output/` folder.

Examples

Here are examples of the Flush Visualizer in action:



Refresh Strategy Highlighting

Presentation

A buffer refresh strategy is responsible of making sure that what is shown on the display contains all the drawings. The ones done since last flush *and* the past. To achieve that it detects the drawn regions and refresh the necessary data in the back buffer.

These information can also be used to understand what happens for each frame in terms of drawings and refreshes. It may be very useful to identify performance issues.

The drawn and restored regions can be very different depending on the selected strategy and the associated options. See *Buffer Refresh Strategy* for more information about the different strategies and their behavior.

Drawn Region(s)

The buffer refresh strategies registers the list of drawn regions between two flushes. These regions can be highlighted during the execution of an application. It can be activated or deactivated with the button in the toolbar.

The color of the rectangles can be configured by setting the `ej.fp.brs.drawnColor` option to any 24-bit RGB color (when set, the feature is activated at startup).

For example with `ej.fp.brs.drawnColor=0x00ff00` :

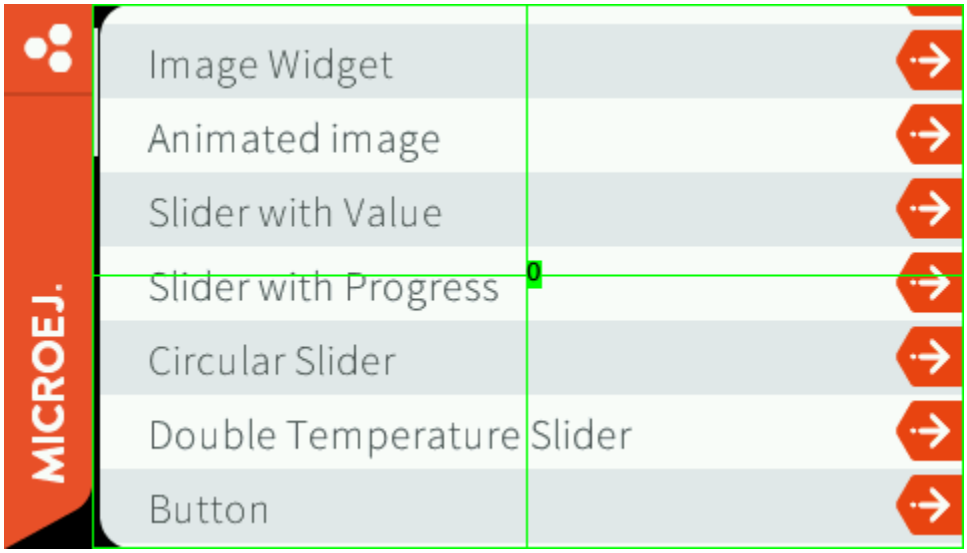


Fig. 149: Drawn region when scrolling.

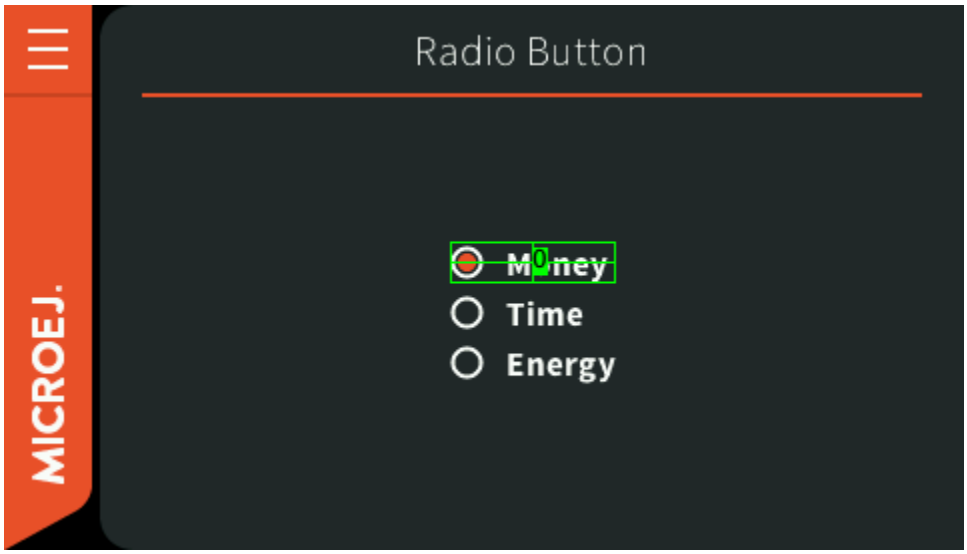


Fig. 150: Drawn region when selecting a radio button.

Restored Region(s)

It is also possible to track the regions restored by the buffer refresh strategies. It can be activated or deactivated with the button in the toolbar.

The color of the rectangles can be configured by setting the ej.fp.brs.restoredColor option can be set to any 24-bit RGB color to highlight these regions (when set, the feature is activated at startup).

For example with ej.fp.brs.restoredColor=0xff00ff :

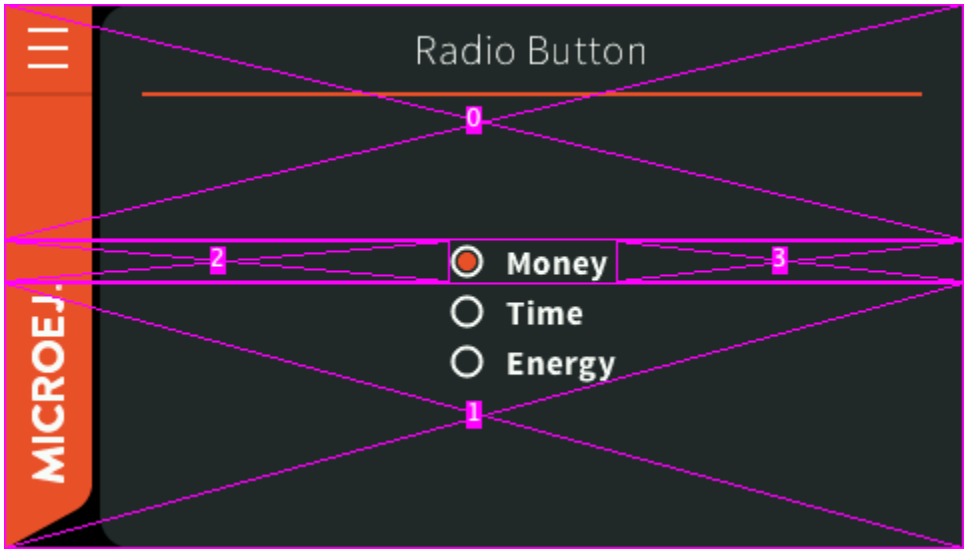


Fig. 151: Restored region when selecting a radio button when entering page.

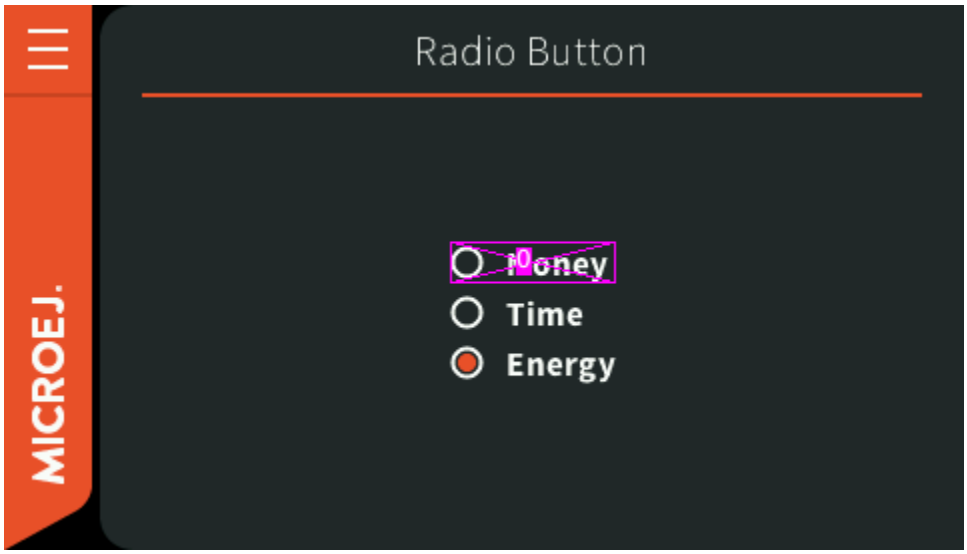


Fig. 152: Restored region when selecting another radio button.

Dirty Region(s)

The buffer refresh strategies use the clip to determine the regions changed between each flush. If a clip has been set but not fully filled by the drawings, the pixels “not drawn” may be flushed to the display as is (without restoration). But the content of these pixels is undefined depending on what this buffer was used for before. It can be a previous frame, one or several flush before depending on the number of buffers. It can also be random pixels if nothing has been drawn on the buffer yet.

These regions are considered as “dirty” since they do not contain the current drawings nor the state of the previous display panel. In other words, it can cause glitches .

To detect easily these regions, a rectangle can be filled with a color for each clip handled by the buffer refresh strategy. It can be activated or deactivated with the button in the toolbar.

The color of the rectangles can be configured by setting the ej.fp.brs.dirtyColor option to any 32-bit ARGB color, opaque or semi-transparent (when set, the feature is activated at startup).

For example: ej.fp.brs.dirtyColor=0x200000ff :

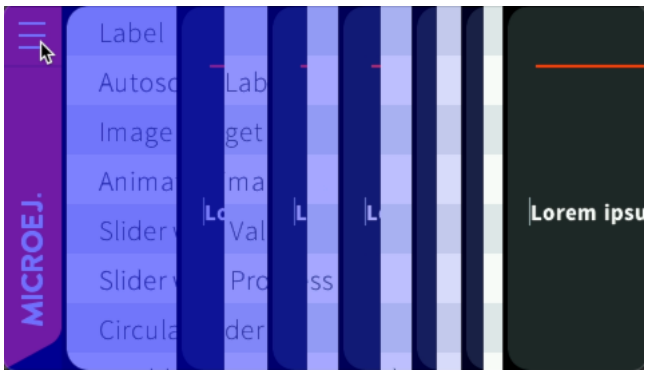


Fig. 153: Dirty regions when animating with the clip not correctly set.

Combining Highlightings

It is possible to use all the highlightings in the same execution. It is particularly convenient to see at the same time the drawn regions and the restored regions.

For example:

```
ej.fp.brs.drawnColor=0x00ff00
ej.fp.brs.restoredColor=0xff00ff
ej.fp.brs.dirtyColor=0x200000ff
```

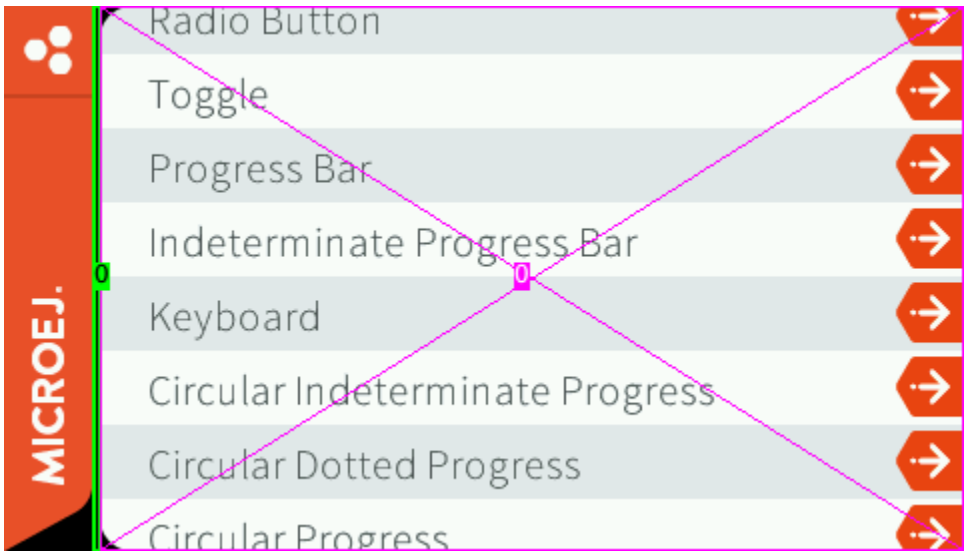


Fig. 154: Drawn and restored regions when the scrollbar is hidden at the end of a scroll.

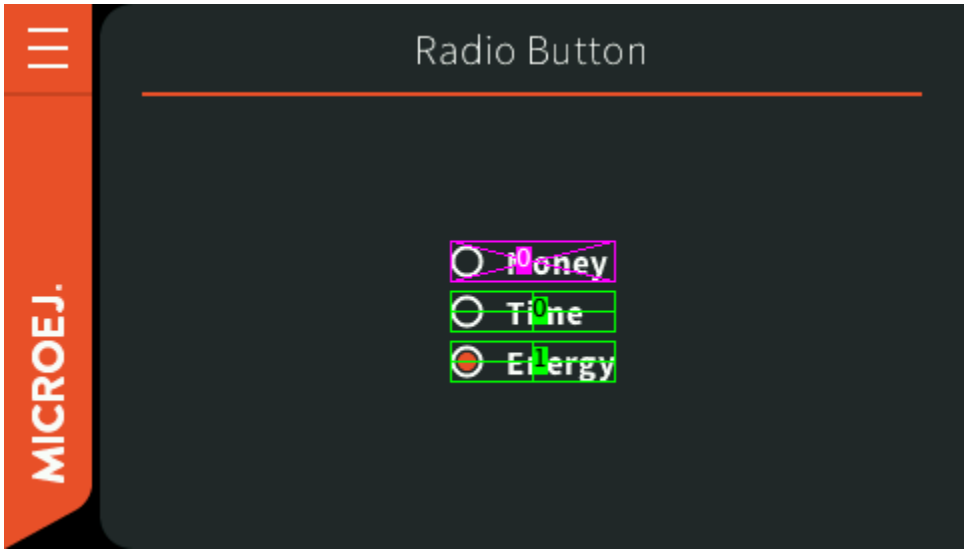


Fig. 155: Drawn and restored regions when selecting another radio button.

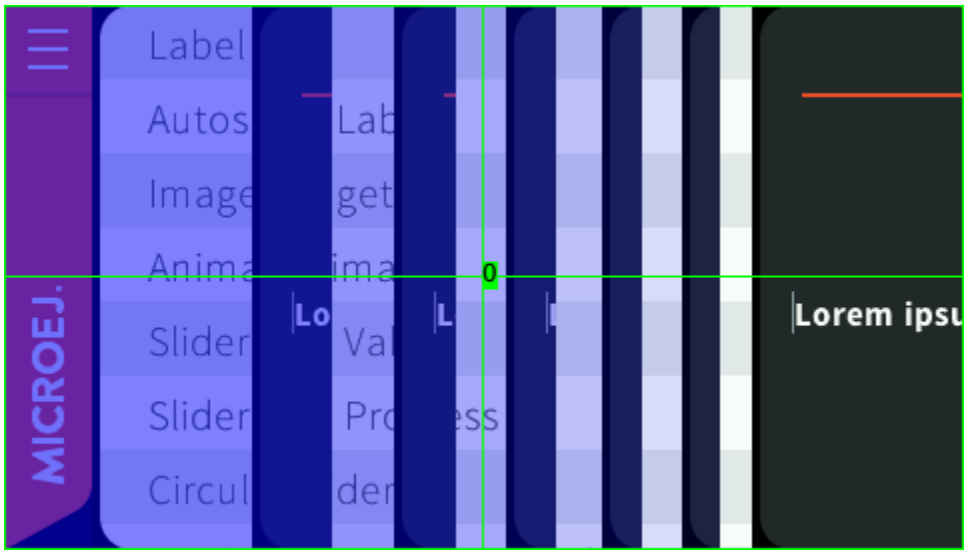


Fig. 156: Drawn and dirty regions when animating with the clip not correctly set.

Front Panel Tips

The following sections provides a non-exhaustive list of tips to make the best use of the Front Panel for Application developers.

Pixel Accurate Display: Window scaling

The Front Panel Display is designed for a pixel-accurate simulation of the GUI application. It uses MicroUI's Graphics Engine, not the system native engine. Therefore, the Front Panel does not support scaling of the window. For accurate graphics, the window must not be resized by the system or the application that launched the simulation (typically the IDE). Here are several distinct solutions:

Scaling of the Front Panel

Disable auto-scaling

Reset display scaling

In Windows settings, if the Display is scaled, the application can be launched with the property `sun.java2d.uiScale` set to `1`. It will force the Front Panel to be displayed at 100% instead of the selected value in the Windows Display settings.

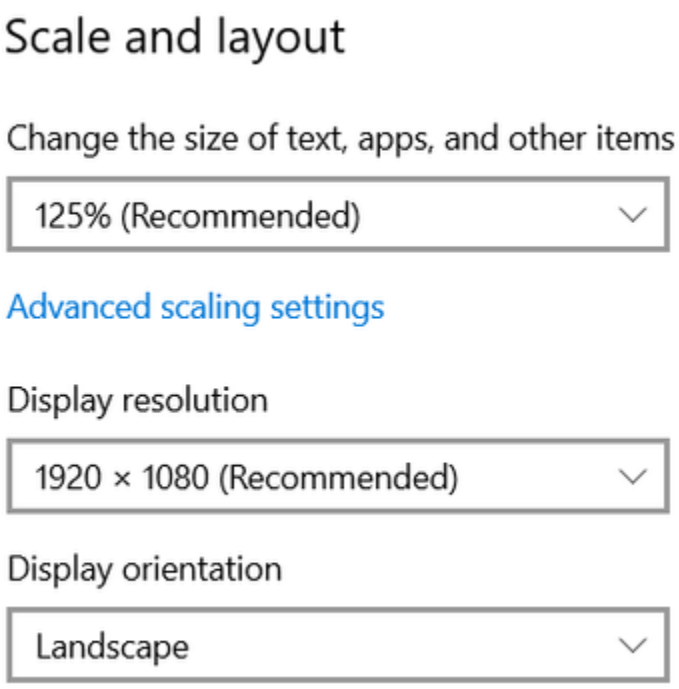


Fig. 157: Windows Display Scale & Layout settings

Override the application auto scaling with the system's in Windows Explorer:

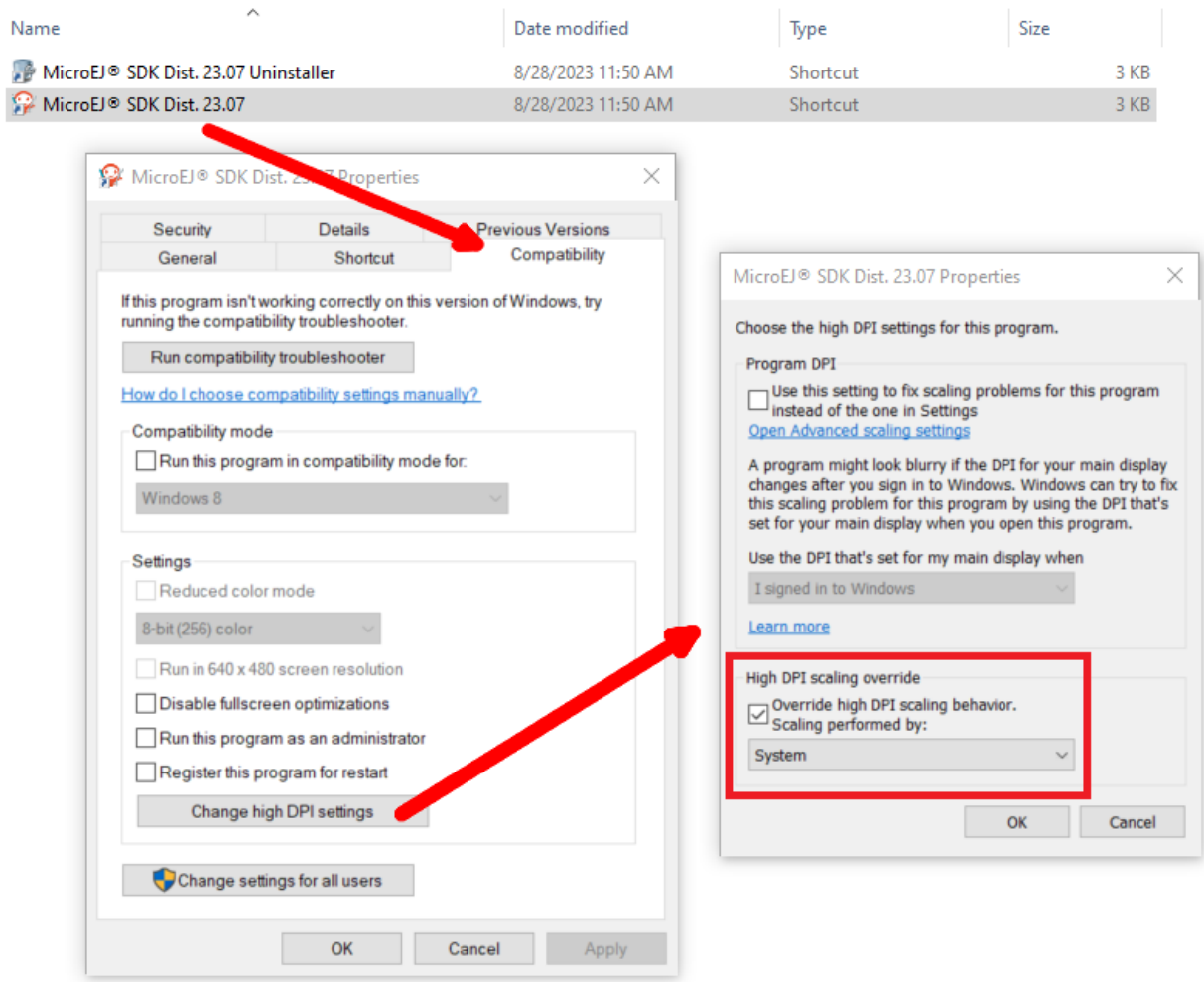


Fig. 158: Windows Application High DPI scaling override setting

On Windows, set your Display settings to 100%:

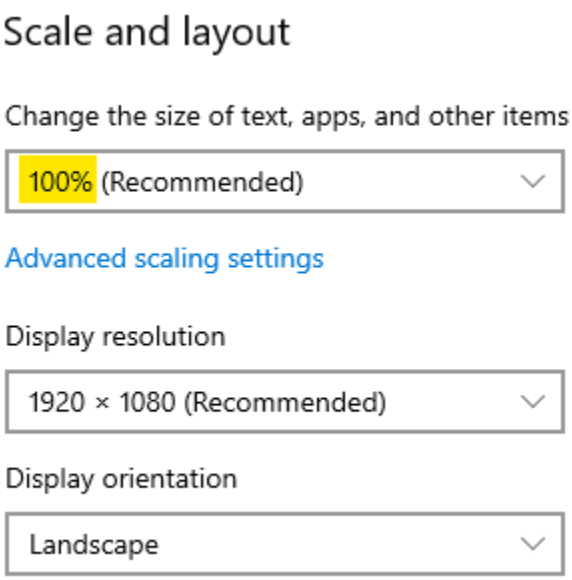


Fig. 159: Windows Display Scale & Layout settings

Zoom on pixelated view for checking custom drawings

Assuming a pixel accurate simulated display (see *Window scaling*), use a screen magnifier tool to zoom on portions of the GUI. It is especially useful to check custom drawings as well as Fonts & Images. Also, make sure the tool does not apply a filter to smooth when scaling.

Windows Magnifier

Windows Magnifier Settings

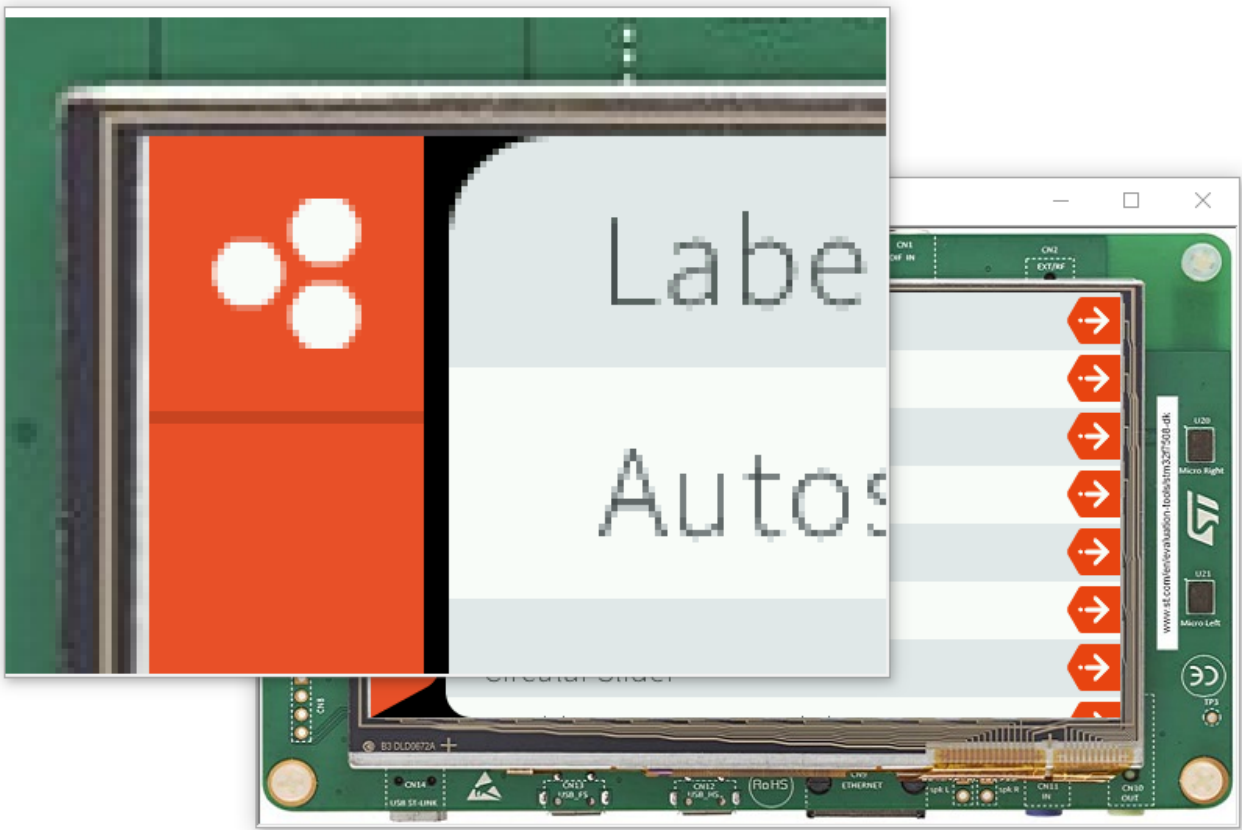


Fig. 160: Windows Magnifier Example

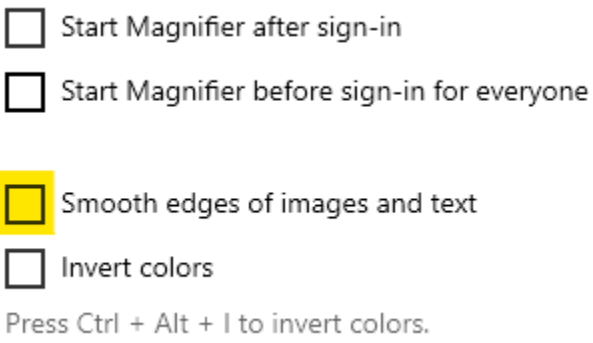


Fig. 161: Windows Magnifier Smooth edges of images and text setting

Take screenshots of the simulated display

An alternative is to make a screenshot and zooming with an image viewer/editor. For that:

- Use *Display Screenshot* button (available since UI Pack 14.3.0).
- Use a screenshot tool.
- Programmatically extend the Front Panel with:

```
// Use ej.fp.widget package to access ej.fp.widget.Display.visibleBuffer
package ej.fp.widget;

import java.awt.Graphics;
import java.awt.Image;
import java.awt.Toolkit;
import java.awt.datatransfer.DataFlavor;
import java.awt.datatransfer.Transferable;
import java.awt.datatransfer.UnsupportedFlavorException;
import java.awt.image.BufferedImage;
import java.io.IOException;

import ej.fp.Device;
import ej.fp.widget.Button.ButtonListener;

public class SceenshotOnClick implements ButtonListener {

   @Override
   public void press(Button widget) {
      
↪  copyImageToClipboard(copyToType(takeScreenshot(), BufferedImage.TYPE_INT_RGB));
      System.out.println("Screenshot copied to clipboard");
   }

   @Override
   public void release(Button widget) {
      // do nothing
   }

   private static BufferedImage takeScreenshot() {
      Display display = Device.getDevice().getWidget(Display.class, null);
      return (BufferedImage) display.visibleBuffer.getRAWImage();
   }

   private static Image copyToType(BufferedImage src, int imageType) {
      BufferedImage dst = new BufferedImage(src.getWidth(), src.getHeight(), imageType);
      Graphics g = dst.createGraphics();
      g.drawImage(src, 0, 0, null);
      g.dispose();
      return dst;
   }

   private static void copyImageToClipboard(Image image) {
      Toolkit.getDefaultToolkit().getSystemClipboard().setContents(new Transferable() {

         @Override
         public boolean isDataFlavorSupported(DataFlavor flavor) {
            return DataFlavor.imageFlavor.equals(flavor);
         }
```

(continues on next page)

(continued from previous page)

```
        @Override
        public DataFlavor[] getTransferDataFlavors() {
            return new DataFlavor[] { DataFlavor.imageFlavor };
        }

        @Override
        public Object␣
↪getTransferData(DataFlavor flavor) throws UnsupportedFlavorException, IOException {
            if (!DataFlavor.imageFlavor.equals(flavor)) {
                throw new UnsupportedFlavorException(flavor);
            }
            return image;
        }

    }, null);
  }

}
```

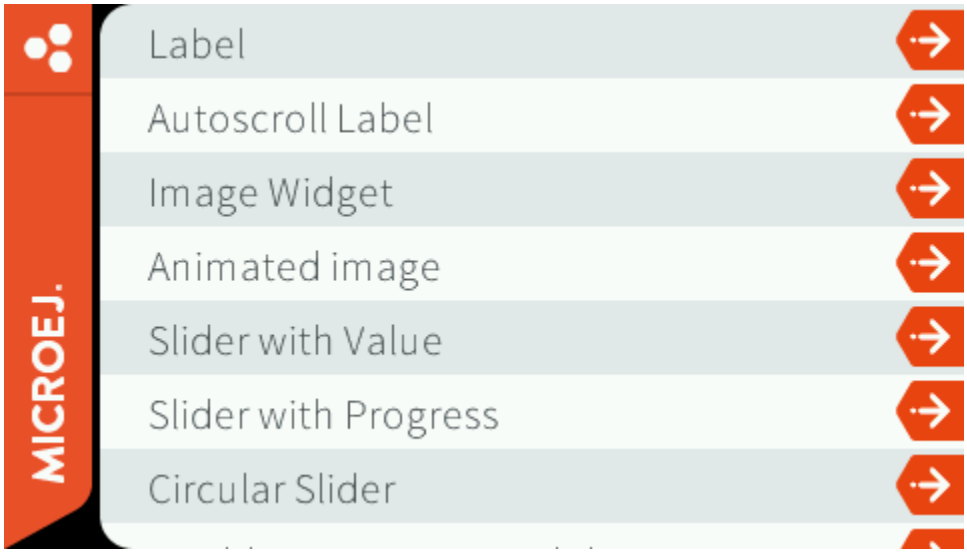
Visual Testing

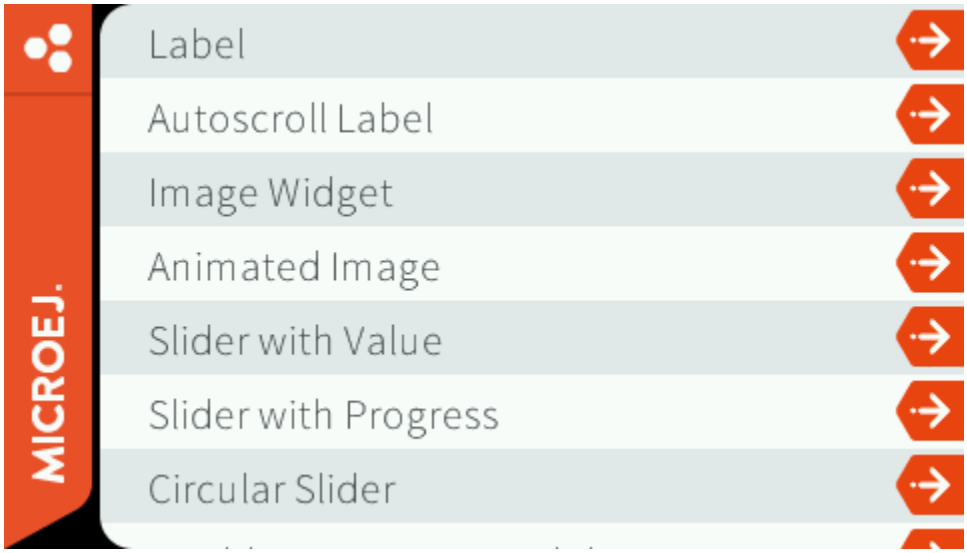
Such screenshots simplifies visual testing. The screenshot can be compared against a made-up image from design specification (typically exported from design tools), or against another screenshot taken from a different version of the application. To go further (and possibly automate such tests), use tools like ImageMagick:

Before

After

Compare

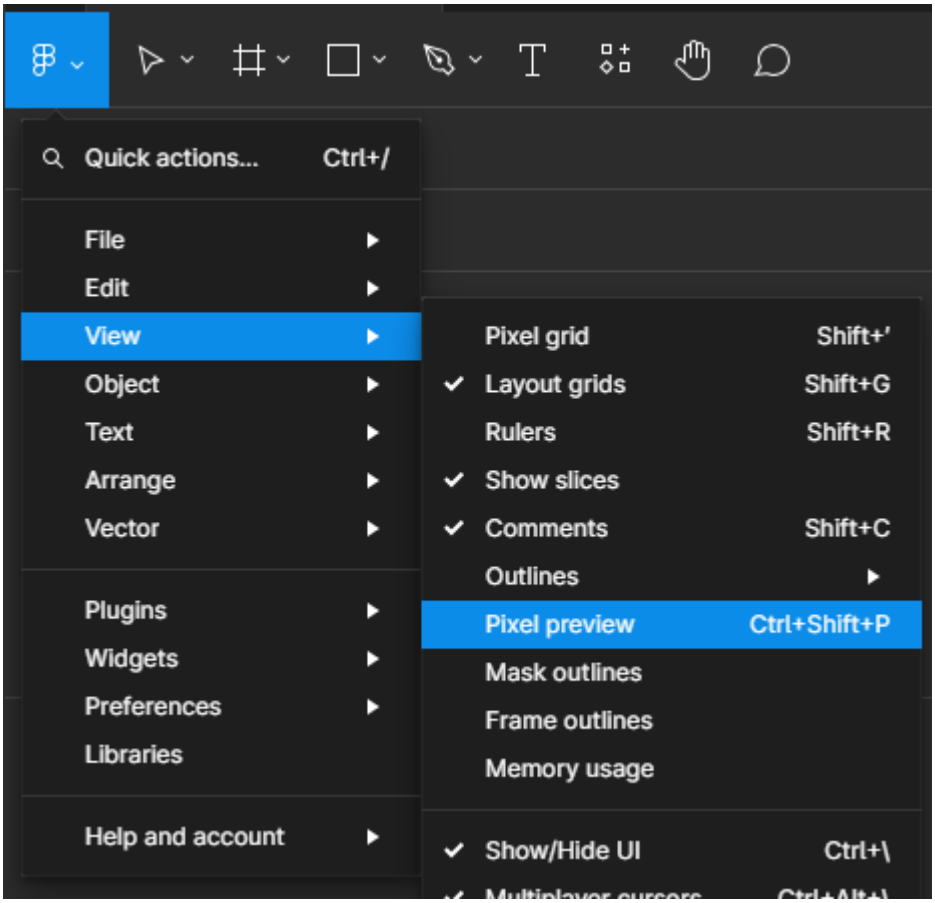





```
$ compare before.png after.png compare.png
```

Compare screenshots with Figma frames

In Figma, frames can be easily exported to PNG images. But it may need more processing before the comparison with the screenshot. First (optional), within Figma, enable Pixel Preview (`View` > `Pixel Preview (Ctrl+Shift+P)`):



Then, if the exported frame does not contain only the display, the image can be cropped with:

```
$ convert figma.png -crop 480x480+45+45 figma-cropped.png
```

Then, if the *MicroUI bpp setting* is not `RGB888` , the image can be filtered to match the supported colors. For example, for `bpp=RGB565` , apply the following filter:

```
$ convert figma.png -channel␣
↪red,blue -evaluate AND 63743 -channel green -evaluate AND 64767 figma-rgb565.png
```

Keep the Front Panel always on top

To keep the Front Panel visible while developing the application, use multiple displays and/or use tools like Microsoft PowerToys' Always on Top utility.

Native Language Support (NLS)

Introduction

Native Language Support (NLS) allows the application to facilitate internationalization. It provides support to manipulate messages and translate them in different languages. Each message to be internationalized is referenced by a key, which can be used in the application code instead of using the message directly.

Principle

NLS is distributed as an add-on library containing a single Java interface: NLS.

In addition to that, the binary-nls library provides a factory for implementations of this interface: it uses an *add-on processor* which processes, offboard, the Localization Source Files into one BON resource buffer file for compactness.

During the *clinit* phase, this resource file is opened and the list of locales is parsed. After that, the resource remains opened for the rest of the Application execution and is directly used to retrieve messages translations for the supported locales.

Localization Source Files

Messages must be defined in localization source files, located in the Classpath of the application (i.e. in the `src/main/resources` folder).

Localization source files can be either PO files or Android String resources.

Here is an example of a PO file:

```
msgid "Label1"
msgstr "My label 1"

msgid "Label2"
msgstr "My label 2"
```

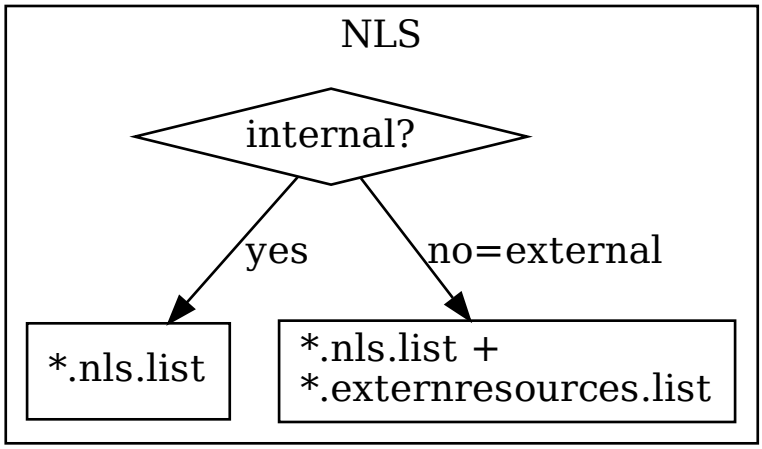
And here is an example of an Android String resource:

```
<resources>
  <string name="Label1">My label 1</string>
  <string name="Label2">My label 2</string>
</resources>
```

Hint: The Android String resources string arrays feature is also supported.

NLS List Files

Localization source files are declared in *Classpath* `*.nls.list` files (**and** to `*.externresources.list` for an external resource, see *Application Resources* and *Loading Translations as an External Resource*).



The file format is a standard Java properties file, each line represents the Full Qualified Name of a Java interface that will be generated and used in the application. Example:

```
com.mycompany.myapp.Labels
com.mycompany.myapp.Messages
```

Usage

The binary-nls module must be added to the Application project build file:

Gradle (build.gradle.kts)

MMM (module.ivy)

```
implementation("com.microej.library.runtime:binary-nls:3.1.0")
```

```
<dependency org="com.microej.library.runtime" name="binary-nls" rev="3.1.0"/>
```

This module includes an Add-On Processor which parses the localization source files. For each interface declared in the NLS list files, all the localization source files whose names start with the interface name are used to generate:

- a Java interface with the given FQN, containing a field for each message of the localization source files
- a NLS binary file containing the translations

So, in the example, the generated interface `com.mycompany.myapp.Labels` will gather all the translations from files named `Labels*` and located in any package of the Classpath. The names of the localization source files should be suffixed by their locale (for example `Labels_en_US.po`).

The generation is triggered when building the application or after a change done in any localization source file or `*.nls.list` files. This allows to always have the Java interfaces up-to-date with the translations and to use them immediately.

Besides the message fields, the generated interface declares an NLS instance which is automatically created in the clinit of the interface.

Once the generation is done, the application can use the Java interfaces to get internationalized messages, for example:

```
String label = Labels.NLS.getMessage(Labels.Label1);
```

Error Codes

The following table lists all the error codes of the library:

Code	Name	Description
1	MISSING_BINARY	The NLS binary resource is missing.
2	WRONG_HEADER	The NLS binary is incompatible: wrong header.
3	WRONG_FORMAT_VERSION	The NLS binary is incompatible: wrong format version.
4	WRONG_CRC	The NLS binary is incompatible: wrong CRC. This typically occurs when at least one localization source file is missing for a language.
5	DEFAULT_IMPLEMENTATION_ERROR	The default NLS implementation cannot be instantiated.
6	REQUIRED_IMPLEMENTATION_ERROR	The required NLS implementation cannot be instantiated.
7	WRONG_LOCALE_INDEX	The provided locale index does not correspond to an available locale.
8	FORM_INDEX_OUT_OF_BOUNDS_ERROR	The plural form index resulting from a plural expression is negative or larger than the number of available forms for the locale.

Locale

For the application to know which language to use among those made available and when, you can set it and change it at any point using the setCurrentLocale(locale) method. If no locale has been set yet when getting a message, the translation for the first locale available in alphabetical order will be used by default. However, you can also pick this locale to default to yourself, by adding a `com.microej.binarynls.defaultLocale` property followed by a locale name in a `.properties.list` file.

Plural Forms

The version 4.0.0 of the NLS module and version 3.0.0 of the binary-nls module introduce the support of GNU gettext's plural forms feature in PO files. The version 3.2.0 of binary-nls adds support of plural forms for Android String resources quantity strings.

This feature allows the localizations to define plural forms and adapt the localized messages based on a numeric value using ej.nls.NLS.getMessage(id, count):

```
int count;
String localizedMessagePlural = MyLabels.NLS.getMessage(MyLabels.MyMessageId, count);
```

The mapping of `count` to a quantity depends on the locale. The binary-nls plural engine is based on GNU gettext's plural forms. Each localization file (PO or XML) that makes use of plural forms must specify the plural rule for the associated locale. The rule is defined in GNU gettext's plural forms format. See the manual (previous link) to get the plural forms for many languages.

GNU gettext (PO)

Android String resources (XML)

First, set the plural rule using the `Plural-Forms` header entry. Then, for each `msgid`, set the `msgstr` for all plural forms.

```
"Plural-Forms: nplurals=3; plural=n%10==1␣
↪&& n%100!=11 ? 0 : n%10>=2 && n%10<=4 && (n%100<10 || n%100>=20) ? 1 : 2;"

msgid "One file removed"
msgid_plural "%d files removed"
msgstr[0] "%d slika je uklonjena"
msgstr[1] "%d datoteke uklonjenih"
msgstr[2] "%d slika uklonjenih"
```

binary-nls maps a quantity to a plural form index and then makes use of a plural rule similar to that found in PO files to map the `count` to the localized message for the associated quantity. This rule must be added to your XML file with a custom `<ej-plural-rule>` element. For example, the plural rule for French could look like this:

```
<resources>
   <ej-plural-rule nplurals="3">
      (n > 1) ? 0 : 1
   </ej-plural-rule>

   <plurals name="Plural forms">
      <item quantity="one">%d fruit</item>
      <item quantity="many">%d fruits</item>
      <item quantity="other">%d fruits</item>
   </plurals>
</resources>
```

The `nplurals` attribute must be set to the number of quantity keys supported by the locale language, as defined in Unicode CLDR Language Plural Rules (for cardinal type). The returned index corresponds to the index of supported quantities, in order from the list `zero`, `one`, `two`, `few`, `many`, and `other`. For example, in French, `one`, `many` and `other` are supported, and they are indexed by, respectively, `0`, `1`, and `2` (and `nplurals` must be set to `3`).

Note: If a locale defines plural forms for a message, all other locales for the same NLS interface must provide the plural forms for this message.

Note: The plural rule can only be set at build time. When updating NLS locale data using *nls_external_resource*, only the localized messages can be updated. It assumes that the plural rule used is the same as when the application was built.

Note: Please note that one significant difference with gettext's implementation is that the expression described in the plural field of the Plural-Forms header must be a valid **Java** expression returning an int, as opposed to a C expression. A usual case in which this makes a difference is for expressions that rely on boolean values being evaluated as zero or one in C, such as in:

```
"Plural-Forms: nplurals=2; plural=n != 1;\n"
```

This expression will not work with our implementation as Java does not interpret booleans as integers. An easy way to convert this expression would be:

```
"Plural-Forms: nplurals=2; plural=n != 1 ? 1 : 0;\n"
```

Also note that the validity of these provided expressions is not entirely checked. Providing an expression that is not valid Java code or that would return an invalid plural form index would cause errors at runtime or even in the Java files generated by the Add-On Processor.

Missing Translations

By default, if a translation is missing for a given msgid in a PO file in a given language, the message returned by the ej.nls.NLS.getMessage() method with the locale set to this language will simply be the msgid itself. In the case of an XML Android String resource, the name attribute of a missing string element will be returned. However if returning this identifier is not a suitable solution, you might want to set a fallback locale parameter for an interface. This parameter corresponds to a language to print the translation for a message in, in case it is not available in the current language.

Starting with version 2.5.0 of the binary-nls module, you can set this fallback locale by specifying a locale name in a .nls.list file, after the name of the interface you want this locale to be the fallback for, separated by a colon : . For example, with the following .nls.list file, if a translation is missing in a language for a message in the Labels and Messages PO/XML files, the message will be translated to en_US instead of just returning its msgid / name .

```
# Missing translations for Labels and Messages will fall back to en_US
com.mycompany.myapp.Labels:en_US
com.mycompany.myapp.Messages:en_US
```

As such, you can specify a different fallback locale for each interface in a .nls.list file. For example, with the following .nls.list file, the messages in Labels will not have a fallback language set and will only return the msgid / name if a translation is missing, while missing translations will default to en_US for the messages in Messages , and to ja_JP for the messages in Content :

```
# Missing translations for Labels will fall back to their msgid/name
com.mycompany.myapp.Labels

# Missing translations for Messages will fall back to en_US
com.mycompany.myapp.Messages:en_US
```

(continues on next page)

(continued from previous page)

```
# Missing translations for Content will fall back to ja_JP
com.mycompany.myapp.Content:ja_JP
```

In the case of a message with plural forms in PO files, this works much the same way, using the messages and forms in the fallback locale if available. If no fallback locale is specified or if the requested message is not specified in it, then the `msgid` will be used for a count value of 1, and the `msgid_plural` will be used for any other value, as gettext would function.

Converter

Problematic

Translated messages can be used directly for the following purposes:

- EDC (in the console): System.out.println().
- MicroUI: ej.microui.display.Painter.drawString().
- MicroVG: ej.microvg.VectorGraphicsPainter.drawString().

When displaying certain languages, such as Arabic, string analysis is necessary for character substitution and right-to-left (RTL) reading direction. *Console encoding* is required for proper display using EDC.

Without Console Encoding

With Console Encoding

```
System.out.print("      ");
```

```
'D91(J)
```

```
System.out.print("      ");
```

In order to render correctly such a message with MicroVG, the *complex layout* must be used. This means that the font must contain substitution tables that the rendering engine can read and apply. If these conditions are not met, the rendering may be incorrect. It is also important to note that using a complex font has a cost in terms of flash storage (due to the increased size of the TTF file and the addition of complex layout algorithms) as well as in run time (due to the time required to apply the substitution tables).

Without Complex Layout

With Complex Layout

ةيبرعلا

العربية

It is not possible to render such a message with MicroUI: the Graphics Engine does not offer substitution table reading or bidirectional string management. The rendering is systematically wrong:

ةيبرعلا

Solution

Since the version 3.1.0, the binary-nls module features an offboard translation conversion. It means that the generated strings can be substituted and rearranged before being embedded in the executable.

This conversion enables MicroUI's Graphics Engine to render complex strings correctly.

Warning: This offboard conversion only concerns PO files.

Without Offboard Conversion

With Offboard Conversion

ةيبرعلا

العربية

Hint: This also avoids embedding substitution tables and the complex layout management when the message is rendered with MicroVG.

Principle

Keep in mind that offboard conversion is only relevant to translated strings. It is important to note that all other fields, such as message identifiers and display names, are not converted as they are not intended to be rendered.

```
msgid "Arabic" // not converted
msgstr "      " // converted
```

Offboard conversion is not a systematic process, so it is necessary to mention it explicitly in the PO file. To do so, add `Language-converter: name_of_converter\n` to the PO file's header, where `name_of_converter` is the name of the converter to be applied (see below for the available list of converters).

```
msgid ""
msgstr ""
"Language: ar_AR\n"
"Language-Team:       \n"
"Language-Converter: Arabic\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=UTF-8\n"
```

(continues on next page)

(continued from previous page)

```
msgid "Arabic"
msgstr "      "
```

List of Converters

Bidi

This converter features details about the bidirectional reordering of text, which is necessary to correctly render Arabic or Hebrew text. These languages are unique in that they are mixed-directional, meaning they order numbers from left to right while ordering most other text from right to left.

- Example of PO file:

```
msgid ""
msgstr ""
"Language: bidi\n"
"Language-Team: Bidirectional\n"
"Language-Converter: Bidi\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=UTF-8\n"

msgid "Hello"
msgstr "Hello"
```

- Result:
  - Unicodes before conversion: U+006f U+006c U+006c U+0065 U+0048
  - After reordering: U+0048 U+0065 U+006c U+006c U+006f

Arabic

This converter is dedicated to the Arabic language, which involves text-based shaping and bidirectional reordering of text. Text-based shaping refers to the process of replacing certain character code points in the text with others depending on the context. The purpose of this process is to transform one type of text into another.

- Example of PO file:

```
msgid ""
msgstr ""
"Language: ar_AR\n"
"Language-Team:      \n"
"Language-Converter: Arabic\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=UTF-8\n"

msgid "Arabic"
msgstr "      "
```

- Result:

- Unicodes before conversion: U+0627 U+0644 U+0639 U+0631 U+0628 U+064a U+0629
- After text shaping: U+fe8d U+fedf U+fecc U+feae U+fe91 U+fef4 U+fe94
- After reordering: U+fe94 U+fef4 U+fe91 U+feae U+fecc U+fedf U+fe8d

Hebrew

This converter is dedicated to the Hebrew language, which involves text-based shaping and bidirectional reordering of text. Not all point-letter combinations match a substituted Unicode character. The following table lists the supported combinations. For all others combinations (Niqqud), the point and the letter are rendered independently.

Point	Representation	Unicode	Letter	Representation	Unicode	Substitution	U
Sheva	◌	U+05B0					
Hataf Segol	◌	U+05B1					
Hataf Patah	◌	U+05B2					
Hataf Qamats	◌	U+05B3					
Hiriq	◌	U+05B4	Yod		U+05B4		U
Tsere	◌	U+05B5					
Segol	◌	U+05B6					
Patah	◌	U+05B7	Alef		U+05D0		U
Qamats	◌	U+05B8	Alef		U+05D0		U
Holam	◌	U+05B9	Vav		U+05D5		U
Holam Haser (for Vav U+05D5)	◌	U+05BA					
Qubuts	◌	U+05BB					
Mapiq	◌	U+05BC	Alef		U+05D0		U
Dagesh	◌	U+05BC	Bet		U+05D1		U
Dagesh	◌	U+05BC	Gimel		U+05D2		U
Dagesh	◌	U+05BC	Dalet		U+05D3		U
Mapiq	◌	U+05BC	He		U+05D4		U
Dagesh	◌	U+05BC	Vav		U+05D5		U
Dagesh	◌	U+05BC	Zayin		U+05D6		U
Dagesh	◌	U+05BC	Tet		U+05D8		U
Dagesh	◌	U+05BC	Yod		U+05D9		U
Dagesh	◌	U+05BC	Final Kaf		U+05DA		U
Dagesh	◌	U+05BC	Kaf		U+05DB		U
Dagesh	◌	U+05BC	Lamed		U+05DC		U
Dagesh	◌	U+05BC	Mem		U+05DE		U
Dagesh	◌	U+05BC	Nun		U+05E0		U
Dagesh	◌	U+05BC	Samekh		U+05E1		U
Dagesh	◌	U+05BC	Final Pe		U+05E3		U
Dagesh	◌	U+05BC	Pe		U+05E4		U
Dagesh	◌	U+05BC	Tsadi		U+05E6		U
Dagesh	◌	U+05BC	Qof		U+05E7		U
Dagesh	◌	U+05BC	Resh		U+05E8		U
Dagesh	◌	U+05BC	Shin		U+05E9		U
Dagesh	◌	U+05BC	Tav		U+05EA		U
Meteg	◌	U+05BD					
Maqaf		U+05BE					

continues on ne

Table 36 – continued from previous page

Point	Representation	Unicode	Letter	Representation	Unicode	Substitution	U
Rafe	◌	U+05BF	Bet		U+05D1		U
Rafe	◌	U+05BF	Kaf		U+05DB		U
Rafe	◌	U+05BF	Pe		U+05E4		U
Paseq		U+05C0					
Shin Dot	◌	U+05C1	Shin		U+05E9		U
Shin Dot	◌	U+05C1	Shin with Dagesh		U+FB49		U
Sin Dot	◌	U+05C2	Sin		U+05E9		U
Sin Dot	◌	U+05C2	Shin with Dagesh		U+FB49		U
Sof Pasuq		U+05C3					
Upper Dot	◌	U+05C4					
Lower Dot	◌	U+05C5					

- Example of PO file:

```
msgid ""
msgstr ""
"Language: he\n"
"Language-Team: Hebrew\n"
"Language-Converter: Hebrew\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=UTF-8\n"

msgid "Man"
msgstr "  "
```

- Result:
  - Unicodes before conversion: U+05D0 U+05BC U+05D9 U+05E9 U+05C1
  - After text shaping: U+FB30 U+05D9 U+FB2A
  - After reordering: U+FB2A U+05D9 U+FB30

Limitations

Conversion is a feature dedicated to graphic display (MicroUI or MicroVG). A message converted and displayed with *EDC* may be shown incorrectly, especially regarding visual orientation.

Without Offboard Conversion

With Offboard Conversion

```
System.out.print("  ");
```

```
System.out.print("  ");
```

Messages are usually displayed using a single type of output, either EDC or UI. When printing the text with EDC, it is correctly rendered without any pre-conversion (the terminal on the PC, that actually prints the text, performs the necessary reordering, substitutions, etc.) To properly render the text on the UI display, the PO file must explicitly specify a converter (see above) to ensure

compatibility. But when printing a pre-converted text with EDC, the application needs to add the character U+202D before the message to force the message orientation, and U+202C after it to restore the previous orientation.

Without U+202D

With U+202D

```
System.out.print("      ");
```

```
System.out.print("\u202D" + "      " + "\u202C");
```

Warning: This tip works on the Simulator but may not work with the MicroVG complex layout manager.

Resource Generation

If the classpath of the Application contains .po / .xml files and .nls.list files, the binary-nls Add-On Processor will generate the following source files for each NLS interface:

- a .resourcebuffer
- a .resourcebuffer.list which references the .resourcebuffer
- a .resources.list which references the resource (this resource does not exist yet but it will be generated later)

When building the Application or running it on Simulator, the Resource Buffer Generator is first executed. Based on the .resourcebuffer and the .resourcebuffer.list , it will generate a resource.

Since the generated resource is referenced by the .resources.list generated by the binary-nls ADP, the SOAR will embed the resource in the Application binary. Unless it is also referenced by an .externresources.list in which case the SOAR will output the resource in the *External Resources Folder* instead.

This resource is loaded as soon as the BinaryNLS instance is created, in the clinit of the generated NLS interface (see *Principle*).

External Resource

When the resource is also referenced by a .nls.externresources.list file (cf. *NLS List Files*), it can be loaded as External Resource in order to be loaded from an external memory (e.g. from a FileSystem).

Note: This mode requires to setup the *External Resources Loader* in the VEE Port.

Usage

The procedure below assumes that the application already has localization source files named HelloWorldMessages*.po referenced as internal resources in a .nls.list file. The localization source files are declared as follows in the .nls.list file: com.microej.example.nls.generated.HelloWorldMessages .

The procedure below explains how to declare those translations as an External Resource:

- Create a .nls.externresources.list file next to the .nls.list file,
- Add the path to the generated External Resource. This path can be deduced from the declaration done in the .nls.list file, for example:

  Content of the .nls.list file:

```
com.microej.example.nls.generated.HelloWorldMessages
```

  Path to add in the .nls.externresources.list file:

```
/com/microej/example/nls/generated/HelloWorldMessages.nls
```

  This path can also be found in the application build folder once the application has been built for the device (e.g. build/adp/src-adpgenerated/binarynls/java/com/microej/exercises/generated/HelloWorldMessages.nls.resources.list).
- Build the application for the device,
- Open the *The SOAR Map File* file to check that the translations are not embedded anymore in the application binary. The xxx_HelloWorldMessages_*.nls lines **must** not appear anymore in the ApplicationResources section.
- The resource containing translations is now located in the *External Resources Folder* (e.g. build/application/object/externalResources/com/microej/exercises/generated/HelloWorldMessages.nls). This resource must be embedded on the target and loaded using the External Resources Loader. For more information, refer to the *External Resources Loader Use section*.

  When using a resource referenced as External Resource, the application is not guaranteed to access it at startup (external memory failure, corruption, …).

  The application can be configured to fallback on a default resource embedded in the Application binary. This resource can be a “lighter” version of the one loaded using the External Resources Loader (e.g. only embed the English language).

Fallback on Default Resource

The procedure below assumes that the application already has localization source files named HelloWorldMessages*.po that are referenced as External Resource.

The procedure below explains how to setup the fallback on a default resource embedding the en_US locale only:

- Create a new localization source file in the src/main/resources folder (e.g. HelloWorldMessagesDefault_en_US.po). This file should contain the same translations as HelloWorldMessages_en_US.po ,
- Declare it in the *.nls.list file (e.g. com.microej.example.nls.generated.HelloWorldMessagesDefault),

- Create a new class that implements the NLS interface (e.g. DefaultNLS),
- Implement every method, wrapping on HelloWorldMessagesDefault :

```
public class DefaultNLS implements NLS {

	@Override
	public String[] getAvailableLocales() {
		return HelloWorldMessagesDefault.NLS.getAvailableLocales();
	}

	@Override
	public String getDisplayName(String locale) {
		return HelloWorldMessagesDefault.NLS.getDisplayName(locale);
	}
	...
```

- Set the DefaultNLS class as the default NLS implementation:
  - Create a *.properties.list file in the src/main/resources folder (if not already created),
  - Add the following property in this file: com.microej.binarynls.defaultImplementation=[FULLY QUALIFIED NAME TO DEFAULT IMPLEMENTATION CLASS] (e.g. com.microej.binarynls.defaultImplementation=com.microej.example.nls.DefaultNLS).
- Declare DefaultNLS as a *Required type*:
  - Create a *.types.list file in the src/main/resources folder (if not already created),
  - Add the fully qualified name of the class (e.g. com.microej.example.nls.DefaultNLS).

    To guarantee the proper application operation, the default translations (HelloWorldMessagesDefault) must be consistent with the translations embedded in External Memory (HelloWorldMessages). In other words, they must contain the exact same set of messages.
- Add the following code in the Main class to perform the consistency check at startup:

```
static {
	if (HelloWorldMessagesDefault.KeysCRC32 != HelloWorldMessages.KeysCRC32) {
		throw new RuntimeException(
␣
↪"CRC check fail between default and fallback translations. Make sure PO files are aligned.");
	}
}
```

> **Warning:** This implementation only checks the consistency of msgid , it does not check the content of msgstr . PO files should be checked carefully to avoid deviation between translations.

The logs below are showing the expected behavior when the resource can be loaded or can't be loaded from External Memory:

Resource Loaded from External Memory

Fallback on Default Resource (External Memory failure)

```
MicroEJ START
Available locales:
```

(continues on next page)

(continued from previous page)

```
- en_US
- es_FR
- fr_FR
Saying:
English (US) (en_US)
- Hello, World
- What's up?
Español (es_FR)
- Hola, Mundo
- ¿ Qué tal ?
Français (fr_FR)
- Bonjour, Le Monde
- Ça va ?
MicroEJ END (exit code = 0)
```

```
MicroEJ START
NLS-PO:I=6
Exception in thread "main" java.io.IOException: NLS-PO:S=1
    at java.lang.System.getStackTrace(Unknown Source)
    at java.lang.Throwable.fillInStackTrace(Throwable.java:82)
    at java.lang.Throwable.<init>(Throwable.java:37)
    at java.lang.Exception.<init>(Exception.java:18)
    at java.io.IOException.<init>(IOException.java:18)
    at com.microej.nls.BinaryNLS.loadBinFile(BinaryNLS.java:385)
    at com.microej.nls.BinaryNLS.<init>(BinaryNLS.java:203)
    at com.microej.nls.BinaryNLS.newBinaryNLSInternal(BinaryNLS.java:161)
    at com.microej.nls.BinaryNLS.newBinaryNLS(BinaryNLS.java:155)
    at com.
↪microej.example.nls.generated.HelloWorldMessages.<clinit>(HelloWorldMessages.java:19)
    at java.lang.Thread.execClinit(Unknown Source)
    at java.lang.Thread.clinitWrapper(Thread.java:483)
    at java.lang.Thread.callWrapper(Thread.java:449)

Available locales:
- en_US
Saying:
English (US) (en_US)
- Hello, World
- What's up?
MicroEJ END (exit code = 0)
```

Limitations

The latest BinaryNLS implementation does not support (even when the resource is external; see *External resource loader*):

- to dynamically add a new locale
- to dynamically modify messages translations

For any addition / modification, the Application must be restarted and, typically, the full resource buffer must be updated (not only the part of the added/modified locale).

Also, there is no API to close the resource buffer. If it is external, the Application must be stopped to close this resource, before it can potentially be modified depending on the external resource loader.

Virtual Device PO Loader Tool

The NLS External Loader tool allows to update the PO files of an application executed on a Virtual Device without rebuilding it. PO files can be dropped in a given location in the Virtual Device folders to dynamically replace the language strings packaged in the application.

This is typically useful when testing or translating an application in order to have a quick feedback when changing the PO files. Once the PO files are updated, a simple restart of the Virtual Device allows to immediately see the result.

Installation

To enable the NLS External Loader in the Virtual Device, add the following dependency to the Firmware project:

Gradle (build.gradle.kts)

MMM (module.ivy)

```
microejTool("com.microej.tool:nls-po-external-loader:3.0.0")
```

```
<dependency␣
↪org="com.microej.tool" name="nls-po-external-loader" rev="3.0.0" transitive="false"/>
```

Then rebuild the Firmware project to produce the Virtual Device.

Usage

Once the project built:

- unzip the Virtual Device and create a folder named `translations` in the root folder.
- copy all the PO files from the project into the `translations` folder. All PO files found in this folder are processed, no matter their folder level.
- start the Virtual Device with the launcher. The following logs should be printed if the NLS External Loader has been executed and has found the PO files:

```
externalPoLoaderInit:init:

externalPoLoaderInit:loadPo:
   [mkdir] Created dir: <PATH>\tmp\microejlaunch1307817858\resourcebuffer
[po-to-nls] *.nls files found in <PATH>\output\<FIRMWARE>\resourceBuffer :
[po-to-nls]   - com.mycompany.Messages1
[po-to-nls]   - com.mycompany.Messages2
[po-to-nls] Loading *.po files for NLS interface com.mycompany.Messages1
[po-to-nls]   => loaded locales : fr_FR,de_DE,ja_JP,en_US
[po-to-nls] Loading *.po files for NLS interface com.mycompany.Messages2
[po-to-nls]   => loaded locales : fr_FR,de_DE,ja_JP,en_US
```

- update the languages strings in the PO files of the Virtual Device (the files in the *translations/* folder).
- restart the Virtual Device and check the changes.

It is important to know the following rules about the NLS External Loader:

- the external PO files names must match with the default PO files names of the application to be processed.
- when PO files with a given name are loaded, the default translations for these PO files are replaced, there is no merge. It means that:
  - if messages are missing in the new PO files, they are not available anymore for the application and may very probably make it crash.
  - if languages are missing (the application has 3 PO files for English, French and Spanish, and only PO files for English and French are available in the translations folder), the messages of the missing languages are not available anymore for the application and may very probably make it crash.
  - if new messages are added in the PO files, it has no impact, they are ignored by the application.
- External PO files are loaded at Virtual Device startup, so any change requires a restart of the Virtual Device to be considered

Troubleshooting

java.io.IOException: NLS-PO:S=4

The following error occurs when at least 1 PO file is missing for a language:

```
[parallel2] NLS-PO:I=6
[parallel2]␣
↪Exception in thread "main" java.io.IOException: NLS-PO:S=4 323463627 -1948548092
[parallel2]     at java.lang.Throwable.fillInStackTrace(Throwable.java:79)
[parallel2]     at java.lang.Throwable.<init>(Throwable.java:30)
[parallel2]     at java.lang.Exception.<init>(Exception.java:10)
[parallel2]     at java.io.IOException.<init>(IOException.java:16)
[parallel2]     at com.microej.nls.BinaryNLS.loadBinFile(BinaryNLS.java:310)
[parallel2]     at com.microej.nls.BinaryNLS.<init>(BinaryNLS.java:157)
[parallel2]     at com.microej.nls.BinaryNLS.newBinaryNLS(BinaryNLS.java:118)
```

Make sure that all PO files are copied in the `translations` folder.

Crowdin

Crowdin is a cloud-based localization platform which allows to manage multilingual content. The NLS External Loader can fetch translations directly from Crowdin to make the translation process even easier. Translators can then contribute and validate their translations in Crowdin and apply them automatically in the Virtual Device.

A new dependency must be added to Firmware project dependencies to enable this integration:

Gradle (build.gradle.kts)

MMM (module.ivy)

```
microejTool("com.microej.tool:nls-po-crowdin:1.0.0")
```

```
<dependency␣
↪org="com.microej.tool" name="nls-po-crowdin" rev="1.0.0" transitive="false"/>
```

Once the module has been built, edit the file `platform/tools/crowdin/crowdin.properties` to configure the Crowdin connection:

- set `crowdin.token` to the Crowdin API token. A token can be generated in the Crowdin in `Settings` > `API` > click on `New Token` .
- set `crowdin.projectsIds` to the id of the Crowdin project. The project id can be found in the `Details` section on a project page. Multiple projects can be set by separating their id with a comma (for example `crowdin.projectsIds=12,586,874`).

  When the configuration is done, the fetch of the Crowdin translations can be done by executing the script `crowdin.bat` or `crowdin.sh` located in the folder `platform/tools/crowdin/` . The PO files retrieved from Crowdin are automatically pasted in the folder `translations` , therefore the new translations are applied after the next Virtual Device restart.

Data Serialization

Overview

MicroEJ has ported a couple Java libraries for some common data serialization formats:

Name	Description	Module Link	API Link	Use
java.io.DataInput & DataOutput	For serialization[1] of Java primitive types and UTF-8 String.	EDC	DataInput / DataInputStream, DataOutput / DataOutputStream	For example, it is used as the serialization format for our ej.rcommand remote command library to transfer data to the remote.
JSON	*JavaScript Object Notation (JSON)* (RFC 8259) encoders/decoders and a JSONPath (RFC 9535) engine.	• *JSON Library for Java ME (org.json.me)*, • ej.json, • JSONPath (from REST client)	• JSONObject (decoder) / JSONWriter (encoder) • JSONReader (decoder) / JSONWriter (encoder) • JSONPathCompiler (JSONPath engine)	org.json.me: *examples*
CBOR	Concise Binary Object Representation (CBOR) encoder and decoder (RFC 7049).	*ej.cbor*	CborEncoder / CborDecoder	*Examples*
XML	kXML 2 eXtensible Markup Language (XML) encoder/decoder and an implementation of the XmlPull API.	kxml2	• kXML API: KXmlParser / KXmlSerializer • XmlPull API: XmlPullParser / XmlSerializer	*Examples*
Protocol Buffers	Protobuf Java Lite Runtime library to support encoders/decoders generated by the proto 3 compiler (protoc).	com.google.protobuf3	CodedInputStream (decoder) / CodedOutputStream (encoder)	Demo
FlatBuffers	FlatBuffers Runtime library to support encoders/decoders generated by the FlatBuffers schema compiler (flatc).	com.google.flatbuffers	• FlatBuffers: FlatBufferBuilder (encoder) • FlexBuffers (schema-less): FlexBuffers (decoder) / FlexBuffersBuilder (encoder)	
ResourceBuffers	*MicroEJ ResourceBuffers* provides a compact format with random read access from an *Applicaton resource*.	BON	ResourceBuffer (decoder)	Used by *NLS* for packaging translation data, and *Java Time* for packaging the TimeZone database data.

[1] MICROEJ VEE does not implement the standard Java Object Serialization. This mechanism has historically introduced numerous compatibility issues and has since been officially deprecated. Serialization of objects should be handled at the application level.

Good Practices

Deciding which format to pick for some use case can depend on many parameters but here are a few things to have in mind:

- When the serialized data is sent to a remote, the format is usually already specified by the API.
- When the serialized data is local (typically for persistence) or you have control over both ends of the communication, binary formats can be used to improve footprint/bandwidth, and speed up serialization/deserialization. Text formats may still be preferred to simplify debugging.
- For configuration files (edited by humans), text formats is preferred and the overhead is, in most cases, insignificant.

  For more properties (schema/schema-less, zero copy support, compatibility between schema versions, …), refer to the formats documentation. When using data serialization for persistence, it is highly recommended to include a version number to indicate the schema version used to serialize the data, allowing a deserializer to verify compatibility.

XML

XML (eXtensible Markup Language) is used to describe data and text. It allows flexible development of user-defined document types. The format is robust, non-proprietary, persistent and is verifiable for storage and transmission. To parse this data format, the XML Pull parser KXmlParser from the Java community has been integrated to MicroEJ Central Repository.

XML Module

To use the XML Module, add the following line to the project build file:

Gradle (build.gradle.kts)

MMM (module.ivy)

```
implementation("org.kxml2:kxml2:2.3.2")
```

```
<dependency org="org.kxml2" name="kxml2" rev="2.3.2"/>
```

Example Of Use

An example is available at https://github.com/MicroEJ/Example-XML. It presents how to use XML data exchange for your Application. It also details how to use the KXmlParser module.

The example parses a short poem written in XML and prints the result on the standard output. The project can run on any VEE Port (no external dependencies).

```
<?xml version="1.0" encoding="UTF-8"?>
<poem xmlns="http://www.megginson.com/ns/exp/poetry">
      <title>Roses are Red</title>
      <l>Roses are red,</l>
      <l>Violets are blue;</l>
      <l>Sugar is sweet,</l>
      <l>And I love you.</l>
</poem>
```

Running the ReadPoem Java application should print the following trace :

```
================ [ Initialization Stage ] ================
================ [ Launching on Simulator ] ================
Roses are Red
_____

Roses are red,
Violets are blue;
Sugar is sweet,
And I love you.

================ [ Completed Successfully ] ================

SUCCESS
```

Using the XML Pull Parser API

kXML also provides an implementation of the XML Pull Parser API.

Running MyXmlPullApp gives more details on the XML parsing and should print this trace :

```
================ [ Initialization Stage ] ================
================ [ Launching on Simulator ] ================
parser implementation class is class org.kxml2.io.KXmlParser
Parsing simple sample XML
Start document
Start element: {http://www.megginson.com/ns/exp/poetry}poem
Characters:    "\n"
Start element: {http://www.megginson.com/ns/exp/poetry}title
Characters:    "Roses are Red"
End element:   {http://www.megginson.com/ns/exp/poetry}title
Characters:    "\n"
Start element: {http://www.megginson.com/ns/exp/poetry}l
Characters:    "Roses are red,"
End element:   {http://www.megginson.com/ns/exp/poetry}l
Characters:    "\n"
Start element: {http://www.megginson.com/ns/exp/poetry}l
Characters:    "Violets are blue;"
End element:   {http://www.megginson.com/ns/exp/poetry}l
Characters:    "\n"
Start element: {http://www.megginson.com/ns/exp/poetry}l
Characters:    "Sugar is sweet,"
End element:   {http://www.megginson.com/ns/exp/poetry}l
Characters:    "\n"
Start element: {http://www.megginson.com/ns/exp/poetry}l
Characters:    "And I love you."
End element:   {http://www.megginson.com/ns/exp/poetry}l
Characters:    "\n"
End element:   {http://www.megginson.com/ns/exp/poetry}poem
================ [ Completed Successfully ] ================

SUCCESS
```

JSON

As described on the JSON official site, JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is widely used in many applications such as:

- as a mean of data serialization for lightweight web services such as REST
- for server interrogation in Ajax to build dynamic webpages
- or even databases.

JSON is easily readable by humans compared to XML. To parse this data format, several JSON parsers are available on the official JSON page, such as *JSON ME*, which has been integrated to MicroEJ Central Repository.

JSON Module

To use the JSON Module, add the following line to the project build file:

Gradle (build.gradle.kts)

MMM (module.ivy)

```
implementation("org.json.me:json:1.4.0")
```

```
<dependency org="org.json.me" name="json" rev="1.4.0"/>
```

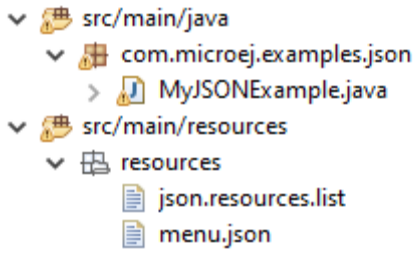
The instantiation and use of the parser is pretty straightforward. First you need to get the JSON content as a `String`, and then create a JSONObject instance with the string. If the string content is a valid JSON content, you should have an workable JSONObject to browse.

Example Of Use

In the following example we will parse this JSON file that represents a simple abstraction of a file menu:

```
{
    "menu": {
        "id": "file",
        "value": "File",
        "popup": {
            "menuitem": [
                {"value": "New", "onclick": "CreateNewDoc()"},
                {"value": "Open", "onclick": "OpenDoc()"},
                {"value": "Close", "onclick": "CloseDoc()"}
            ]
        }
    }
}
```

First, we need to include this file in our project by adding it to the `src/main/resources` folder and creating a `.resources.list` properties file to declare this resource for our application to be able to retrieve it (see *Resources* for more details).



This `.resources.list` file (here named `json.resources.list`) should contain the path to our JSON file as such :

```
resources/menu.json
```

The example below will parse the file, browse the resulting data structure (`org.json.me.JSONObject`) and print the value of the `menuitem` JSON array.

```
package com.microej.examples.json;

import java.io.DataInputStream;
import java.io.IOException;

import org.json.me.JSONArray;
import org.json.me.JSONException;
import org.json.me.JSONObject;

/**
* This example uses the org.json.me parser provided by json.org to parse and
* browse a JSON content.
*
* The JSON content is simple abstraction of a file menu as provided here:
* http://www.json.org/example.html
*
* The example then tries to list all the 'menuitem's available in the popup
* menu. It is assumed the user knows the menu JSON file structure.
*
*/
public class MyJSONExample {

    public static void main(String[] args) {

        // get back an input stream from the resource that represents the JSON
        // content
        DataInputStream dis = new DataInputStream(
↪
                MyJSONExample.class.getResourceAsStream("/resources/menu.json"));

        byte[] bytes = null;

        try {

            // assume the available returns the whole content of the resource
            bytes = new byte[dis.available()];

            dis.readFully(bytes);
```

(continues on next page)

(continued from previous page)

```
            } catch (IOException e1) {
                    // something went wrong
                    e1.printStackTrace();
                    return;
            }

            try {

                    // create the data structure to exploit the content
                    // the string is created assuming default encoding
                    JSONObject jsono = new JSONObject(new String(bytes));

                    // get the JSONObject named "menu" from the root JSONObject
                    JSONObject o = jsono.getJSONObject("menu");

                    o = o.getJSONObject("popup");

                    JSONArray a = o.getJSONArray("menuitem");

                    System.out.println("The menuitem content of popup menu is:");
                    System.out.println(a.toString());

            } catch (JSONException e) {
                    // a getJSONObject() or a getJSONArray() failed
                    // or the parsing failed
                    e.printStackTrace();
            }

    }

}
```

The execution of this example on the Simulator should print the following trace:

```
================ [ Initialization Stage ] ================
================ [ Launching Simulator ] ================
The menuitem content of popup menu is:
[{"value":"New","onclick":"CreateNewDoc()
↪"},{"value":"Open","onclick":"OpenDoc()"},{"value":"Close","onclick":"CloseDoc()"}]
================ [ Completed Successfully ] ================

SUCCESS
```

CBOR

The CBOR (Concise Binary Object Representation) binary data serialization format is a lightweight data-interchange format similar to JSON but with a smaller footprint, making it very practical for embedded applications, though its messages are often less easily readable by humans.

CBOR Module

To use the CBOR Module, add the following line to the project build file:

Gradle (build.gradle.kts)

MMM (module.ivy)

```
implementation("ej.library.iot:cbor:1.2.0")
```

```
<dependency org="ej.library.iot" name="cbor" rev="1.2.0"/>
```

Example Of Use

An example is available at https://github.com/MicroEJ/Example-IOT/tree/master/cbor. It shows how to use the CBOR library in your Application by encoding some data and reading it back, printing it on the standard output both as a raw byte string and in a JSON-like format. You can use tools like cbor.me to convert the byte string output to a JSON format and check that it matches the encoded data. The project can run on any VEE Port (no external dependencies).

The execution of this example on the Simulator should print the following trace:

```
================ [ Initialization Stage ] ================
================ [ Launching on Simulator ] ================
CBOR␣
↪data␣
↪string␣
↪:␣
↪a1646d656e75a36269646466696c656576616c75656446696c6565706f707570a1686d656e756974656d83a26576616c7565634e6577676f6e63
Data content :
{
	"menu" : {
		"id" : "file",
		"value" : "File",
		"popup" : {
			"menuitem" : [ {
					"value" : "New",
					"onclick" : "CreateNewDoc()"
				}, {
					"value" : "Open",
					"onclick" : "OpenDoc()"
				}, {
					"value" : "Close",
					"onclick" : "CloseDoc()"
				} ]
		}
	}
```

(continues on next page)

(continued from previous page)

```
}
================ [ Completed Successfully ] ================
```

Another example showing how to use the *JSON* module along with the *CBOR* module to convert data from JSON to CBOR is available here : https://github.com/MicroEJ/Example-IOT/tree/master/cbor-json.

The execution of this example on the Simulator should print the following trace:

```
Initial data (271 bytes) = {"menu":
↪{"value":"File","id":"file","popup":{"menuitem":[{"value":"New","onclick":"CreateNewDoc()
↪"},{"value":"Open","onclick":"OpenDoc()"},{"value":"Close","onclick":"CloseDoc()"}]}}}
Data serialized (139 bytes)
Data deserialized =␣
↪{menu={value=File, id=file, popup={menuitem=[{value=New, onclick=CreateNewDoc()}
↪, {value=Open, onclick=OpenDoc()}, {value=Close, onclick=CloseDoc()}]}}}
```

Networking

Foundation Libraries

Name	Description	Module Link	API Link	Use
ECOM-Network	Network interfaces management and IP configurations.	ecom-network	NetworkInterfaceManager class	
ECOM-WIFI	Wi-Fi connectivity.	ecom-wifi	WifiManager class	• Wi-Fi setup Example • Wi-Fi utility Library
NET	Client and Server raw TCP/IP sockets.	net	java.net package	• NET Example • NET utility Library
Security	Cryptographic operations.	security	javax.crypto package	
SSL	Client and Server secure sockets layer using Transport Layer Security (TLS) protocols.	ssl	java.net.ssl package	• SSL mutual client Example • SSL mutual server Example • SSL utility Library

Add-On Libraries

IoT Libraries

Name	Description	Module Link	API Link	Use
Android Connectivity	Network connection state and notifications.	android-connectivity	ConnectivityManager class	• Connectivity Example
HTTP Client	OpenJDK HTTP client.	http-client, http-sclient	HttpURLConnection class	• HTTP client README
Web Server (HOKA)	Tiny footprint yet extensible web server.	HOKA	HttpServer class	• *HOKA User Manual* • HOKA Examples
MQTT Client (MicroPaho)	Tiny footprint MQTT 3.1.1 client based on Eclipse Paho Java APIs.	micropaho	MqttClient class	• MicroPaho README • MQTT publish Example • MQTT subscribe Example
REST Client	REpresentational State Tranfer (REST) client.	restclient	Resty class	• REST client README
SNTP Client	Simple Network Time Protocol (SNTP) client, used to retrieve the current time from an NTP server.	sntp-client	SntpClient class	• SNTP client README
WebSocket Client	WebSocket client (RFC 6455).	websocket, websocket-secure	WebSocket class	• WebSocket client README • WebSocket client Example

Cloud Agent Libraries

Name	Description	Module Link	Use
AWS IoT Core	AWS IoT Core client, providing publish/subscribe functionalities.	aws-iot	• AWS IoT Core README • AWS IoT Core Example

HOKA Web Server

HOKA is a tiny extensible Java web server for embedded applications.

It comes with the support of HTTP, HTTPS, Server session, and routing for REST API.

Note: This is the documentation of the latest version of HOKA library 8.X.X

Intended Audience

The intended audience for this document is Java developers who are familiar with socket communication, the HTTP 1.1 protocol, and web server concepts.

Getting Started

Create a new MicroEJ application and add the HOKA library dependency to your MicroEJ application

Gradle (build.gradle.kts)

MMM (module.ivy)

```
implementation("ej.library.iot:hoka:8.4.0")
```

```
<dependency org="ej.library.iot" name="hoka" rev="8.4.0"/>
```

```
public class MyServer {

  public static void main(String[] args) throws IOException {

      HttpServer http = HttpServer.builder().port(8080).build();

      http.get("/hello", new RequestHandler() {

        @Override
        public void process(HttpRequest request, HttpResponse response) {
          response.setData("Hello world!");
        }
      });

      http.start();

  }
}
```

Run the application and check the result at **http://localhost:8080/hello**

Routes Mapping

In HOKA, an HTTP request is a combination of 4 elements:

- **Verb**: The HTTP verbs, GET, POST, PUT or DELETE…
- **Path**: The request path or URI. `/hello/:username`
- **Handler**: The request handler process the request and respond to the client.
- **content type**: (optional) the route supported content type

Note: Paths are matched in the order of their creation. The handler of the first matching path will be invoked. All the paths need to be registered before calling the `start()` method of the `HttpServer` instance. If no path matches the incoming request, the server will return a 404 Not Found response.

```
HttpServer http = HttpServer.builder().port(8080).build();

http.get("/", new RequestHandler() {

  @Override
  public void process(HttpRequest request, HttpResponse response) {
    // read a resource

  }
});

http.post("/", new RequestHandler() {

  @Override
  public void process(HttpRequest request, HttpResponse response) {
    // write a resource
  }
});

http.put("/", new RequestHandler() {

  @Override
  public void process(HttpRequest request, HttpResponse response) {
    // update a resource
  }
});

http.delete("/", new RequestHandler() {

  @Override
  public void process(HttpRequest request, HttpResponse response) {
    // delete a resource
  }
});
```

A path can be registered with **one single** specific content type in different request handlers on the same path.

For example, to map two content types on the same path, do the following:

```
HttpServer http = HttpServer.builder().port(8080).build();

http.get("/", "application/json", new RequestHandler() {

  @Override
  public void process(HttpRequest request, HttpResponse response) {
    // read a resource and return a json formatted response.
  }
});

http.get("/", "application/xml", new RequestHandler() {

  @Override
  public void process(HttpRequest request, HttpResponse response) {
    // read a resource and return a json formatted response.
  }
});
```

Path Parameters

The request path can contain named parameters called path parameters. Those parameters are made available through the `request` instance of the `process()` method of the `RequestHandler`. The path parameter can be accessed by calling `HttpRequest#getPathParam(String param)`

```
HttpServer http = HttpServer.builder().port(8080).build();

http.get("/hello/:name", new RequestHandler() {

  @Override
  public void process(HttpRequest request, HttpResponse response) {
    String name = request.getPathParam("name");
    response.setData("Hello " + name);
  }
});
```

Splat Parameters

The request path also supports splat parameters using wildcard '*'. Those parameters are made available through the `request` instance of the `process()` method of the `RequestHandler`. The splat parameters array can be accessed by calling `HttpRequest#getSplatParams()`

```
HttpServer http = HttpServer.builder().port(8080).build();

http.get("/greet/*/by/*", new RequestHandler() {

  @Override
  public void process(HttpRequest request, HttpResponse response) {
    String name = request.getSplatParams().get(0);
    String greeting = request.getSplatParams().get(1);
    response.setData(greeting + " " + name);
  }
});
```

Request

- HttpRequest#getMethod() : returns the request method (1 for POST , 2 for GET , 3 for PUT and 4 for DELETE . see HttpRequest for the full list).
- HttpRequest#getURI() : returns the requested URI.
- HttpRequest#getQueryParams() : returns the request query parameters map.
- HttpRequest#getQueryParam(String) : returns the query parameter by the given name from the query parameters map.
- HttpRequest#getPathParam(String) : returns the request path parameter by the given name.
- HttpRequest#getSplatParams() : returns the list of splat parameters.
- HttpRequest#setAttribute(String, Object) : set a server-side request attribute. can be used to passe data between handlers.
- HttpRequest#getAttribute(String) : get a server-side request attribute.
- HttpRequest#getVersion() : returns the HTTP protocol version of the request.
- HttpRequest#getHeaders() : returns the request headers, all header field names are converted to lower-case.
- HttpRequest#getHeader(String) : returns the value of the header with the given name.
- HttpRequest#parseBody(BodyParser) : parses the body of the request with the given parser.
- HttpRequest#getRequestBody() : return the request InputStream to be used for any custom request handling.

Body Parsers

The HttpRequest#parseBody(BodyParser) is used to read the body (data) of a request.

HOKA library provides 4 implementations of BodyParser :

- StringBodyParser : returns the full request body as a string.
- MultipartStringsParser : parse a multipart/* request body, each part is returned as a string.
- MultiPartBodyParser : parse a multipart/* body, and parse each part as header fields and an InputStream body.
- ParameterParser : parse an application/x-www-form-urlencoded request body.

Cookies

The cookies are lazily parsed the first time they are accessed.

- HttpRequest#getCookies() : returns the list of cookies.
- HttpRequest#getCookie(String) : returns the value of the cookie by the given name.

Response

Build a HttpResponse based on the request with the following data :

- data : the body of the response as a String , byte[] or as an InputStream .
- status : the status of the response to send. HTTP response code.
- mimeType : the value of the content-type header.
- HttpResponse#addHeader(String name, String value) : adds a header with given name and value.
- HttpResponse#addCookie(Cookie) : adds a cookie to the response. USe ej.hoka.http.Cookie.Builder() to create a cookie instance.

```
// Use the cookie builder to create a cookie instance.
Cookie cookie = Cookie.builder().name("cookieName").value("cookieValue")
                        .expires(expirationDate)
                        .maxAge(900)
                        .domain("www.example.com")
                        .path("/api")
                        .sameSite(SameSite.Strict)
                        .secure()
                        .httpOnly()
                        .build();
```

MIME Types

The Mime class provides constant values for commonly used MIME types and utility methods to return the MIME type of a resource name based on file extensions.

The predefined MIME types are :

- MIME_PLAINTEXT = “text/plain”
- MIME_HTML = “text/html”
- MIME_XML = “text/xml”
- MIME_APP_JSON = “application/json”
- MIME_DEFAULT_BINARY = “application/octet-stream”
- MIME_CSS = “text/css”
- MIME_PNG = “image/png”
- MIME_JPEG = “image/jpeg”
- MIME_GIF = “image/gif”
- MIME_JS = “application/x-javascript”
- MIME_FORM_ENCODED_DATA = “application/x-www-form-urlencoded”
- MIME_MULTIPART_FORM_ENCODED_DATA = “multipart/form-data”

The method Mime#getMIMEType(String URI) returns the MIME type of the given URI, assuming that the file extension in the URI was previously registered with the Mime#mapFileExtensionToMIMEType(String fileExtension, String mimeType) . Only lower case file extensions are recognized.

For example, calling `getMIMEType("/images/logo.png")` will return the string `"image/png"`.

The following table shows the predefined assignments between file extensions and MIME types:

Extension	MIME type
".png"	MIME_PNG
".css"	MIME_CSS
".gif"	MIME_GIF
".jpeg"	MIME_JPEG
".jpg"	MIME_JPEG
".html"	MIME_HTML
".htm"	MIME_HTML
".js"	MIME_JS
".txt"	MIME_PLAINTEXT
".xml"	MIME_XML

Halt Request Processing Chain

to stop a request processing and return immediately. The following static methods form `HttpServer` class should be used.

This will cause the request handler to stop immediately and the response will be returned to the client without executing other filters.

This is useful for error handling for example.

```
halt();  <--- return a 200 OK response.
halt(HTTPConstants.HTTP_STATUS_UNAUTHORIZED);
halt(HTTPConstants.HTTP_STATUS_UNAUTHORIZED, "login required!");
```

Filters

A filter is also a request handler that is executed before or after a registered request.

It needs to be registered before calling the `start()` method on the server instance.

It can be used to pre-process or post-process a request.

Multiple filters can be registered. They will be executed in the order they were added in.

HOKA supports 4 types of filters.

- **before all requests**: runs before any registered path.
- **before a specific path**: runs before a specific registered path.
- **after a specific path**: runs after a specific registered path.
- **after all requests**: runs after any registered path.

Before

Example of adding a filter that will be executed before any registered path.

Multiple before filters can be added by calling `before()` multiple times. They will be executed in their registration order.

```
HttpServer http = HttpServer.builder().port(8080).build();

http.before(new RequestHandler() {

  @Override
  public void process(HttpRequest request, HttpResponse response) {
    boolean authenticated = false;
    // check if authenticated ...
    if (!authenticated) {
      halt(HTTPConstants.
↪HTTP_STATUS_UNAUTHORIZED); // stop the processing and return an error.
    }
  }
});
```

Example of adding a filter that will be executed before a specific registered path.

Unlike global before filters, only one before filter by path can be registered.

```
HttpServer http = HttpServer.builder().port(8080).build();

http.before("/private/*", new RequestHandler() {

  @Override
  public void process(HttpRequest request, HttpResponse response) {
    // check access privilege ...
    halt(HTTPConstants.
↪HTTP_STATUS_FORBIDDEN); // stop the processing and return an error.
  }
});
```

After

Example of adding a filter that will be executed after any registered path.

Multiple global after filters can be added by calling `after()` multiple times. They will be executed in their registration order.

```
HttpServer http = HttpServer.builder().port(8080).build();

http.after(new RequestHandler() {

  @Override
  public void process(HttpRequest request, HttpResponse response) {
    // do some post processing on the request/response
    response.addHeader("common header key", "common header value");
  }
});
```

Example of adding a filter that will be executed after a specific registered path.

Unlike global after filters, only one after filter by path can be registered.

```
HttpServer http = HttpServer.builder().port(8080).build();

http.after("/private/*", new RequestHandler() {

  @Override
  public void process(HttpRequest request, HttpResponse response) {
    // do some post processing on the request/response
    response.addHeader("special header key", "special header value");
  }
});
```

Error Handling

Not Found Error

The 404 not found error can be customized by using the HttpServer#notFoundError() method.

```
HttpServer http = HttpServer.builder().port(8080).build();

// html, The html page can be loaded form a file
http.notFoundError("<html><body><h1>404 Page doesn't exist</h1></body></html>");

// json format
http.notFoundError("{\"message\":\"404 Page doesn't exist\"}", "application/json");
```

Internal Server Error

The 500 Internal Server Error can also be customized.

```
// html, The html page can be loaded form a file
http.internalServerError(
↪"<html><body><h1>505 Something went wrong!</h1></body></html>");

// json format
http.internalServerError("{\"message\":\"505 Something went wrong!\"}", "application/json");
```

Exception Mapping

An exception can be mapped to a custom handler to return specific errors.

```
HttpServer http = HttpServer.builder().port(8080).build();

http.get("/throwerror", new RequestHandler() {

  @Override
  public void process(HttpRequest request, HttpResponse response) {
    throw new MyCustomError();
  }
});
```

(continues on next page)

(continued from previous page)

```
http.exception(MyCustomError.class, new RequestHandler() {

  @Override
  public void process(HttpRequest request, HttpResponse response) {
    // handle the custom error here.
  }
});
```

Static Files

A specific static file handler can be set to serve files from the application classpath by using ClasspathFilesHandler class.

```
HttpServer http = HttpServer.builder() //
    .port(8080) //
    .staticFilesHandler(ClasspathFilesHandler.builder() // set the static file handler
        .rootDirectory("/public") // set the static file folder form src/main/resources
        .build())
    .build();
```

Note that the public directory name is not included in the request URL. to access a file in src/main/resources/public/css/main.css the url is http://localhost:8080/css/main.css

An external file location can be used by providing your own implementation of StaticFilesHandler interface and adding the fs foundation library to work with File* classes from java.io .

Web Server Configuration

HttpServer class builder has the following options :

```
HttpServer http = HttpServer.builder()
    .port(8080) // setup the port number to bind the server socket on. Use 0 for a random port
    .simultaneousConnections(3)␣
↪// setup the max simultaneous connections accepted by the server
    .workerCount(3) // setup the number of threads to handle incoming connections
    .connectionTimeout(60 * 1000) // setup connection timeout
    .encodingRegistry(new␣
↪EncodingRegistry()) // register a custom the content encoding & transfer-coding registry
    .secure(/**SSLContext#getServerSocketFactory()*/) // setup SSL / HTTPS
    .apiBase("/api/v1/") // setup a common␣
↪URI base for all relative registered path. relative means, the path do not starting with a /
    .staticFilesHandler(staticFilesHandler) // setup the static files handler
    .withTrailingSlashSupport() // process route with trailing slash as different routes
    .withStrictAcceptContentEncoding() //
↪ activate strict content acceptance. return 406 Not Acceptable for unknown content-encoding
    .developmentMode()/
↪/ enable development mode, send error stack trace to the client side as in html
    .build();
```

Trailing Slash Matching

By default, the HOKA server ignores the trailing forward slash at the ends of the request URI.

For example:

- GET | host/hello
- GET | host/hello/

Will link to the same request handler.

This behavior can be deactivated by calling the method HttpServer#builder()#withTrailingSlashSupport() on the server builder.

Note that host and host/ will link to the same request handler whatever the Trailing Slash Match is activated or not.

Development Mode

Development mode can be activated by calling HttpServer#builder()#developmentMode() .

This will tell the HOKA server to send the exception stack trace to the client.

The stack trace is sent in a plain text response. This is useful when developing the web application; otherwise, a "500 Internal Error" response is sent.

Note: when development mode is active, internal error page customization is deactivated. The development mode page is returned instead.

Generate Server Self Signed Key and Certificate for HOKA WebServer TLS

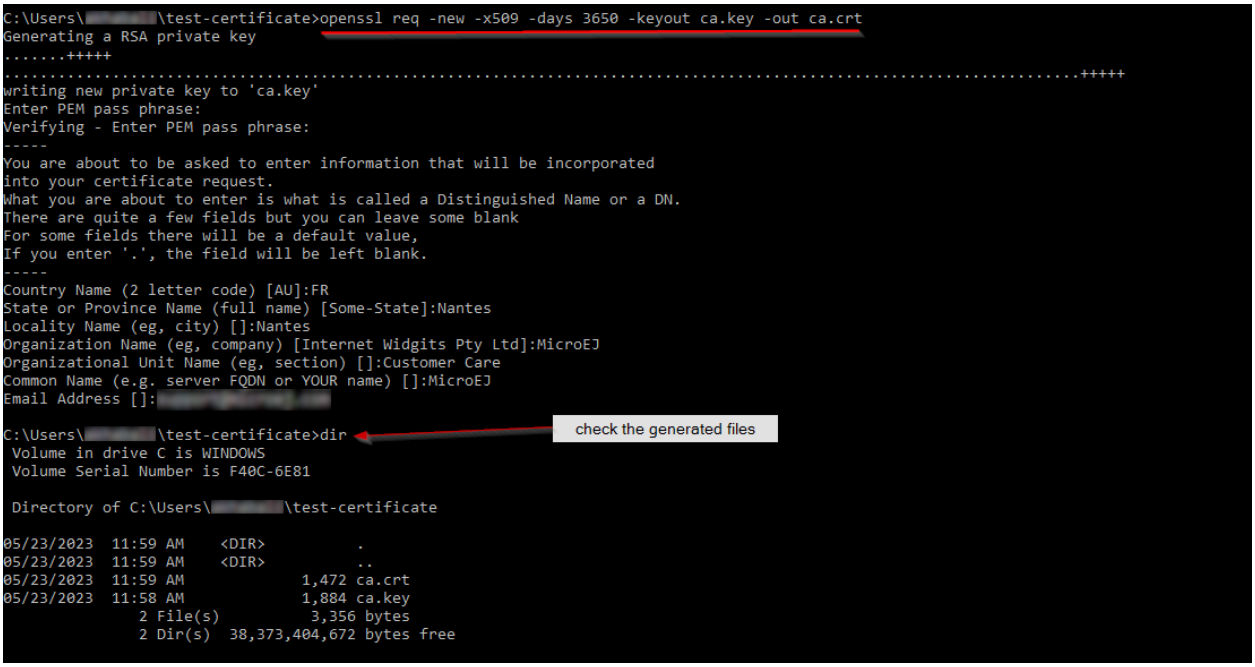
This section details the commands and steps to generate a self signed certificate and a DER formatted key for HOKA server to enable TLS.

Generate Root CA Key & Certificate

To generate a root certificate authority (CA) using openssl, execute the following command and follow the instructions by filling the certificate information:

```
openssl req -new -x509 -days 3650 -keyout ca.key -out ca.crt
```

- ca.key : is the name of the generated root private key in PEM format.
- ca.crt : is the name of the generated root certificate in PEM format.



Generate HOKA Server Private Key

To generate a private key using openssl, execute the following command:

```
openssl genrsa -out hoka.key 4096
```

- `hoka.key` : is the name of the generated private key.
- `4096` : is the length of the private key.

Generate HOKA Server Self Signed Public Key

To generate a Self signed public key:

1. Generate a certificate signing request (CSR) using openssl, for that execute the following command and fill in the information: `openssl req -new -sha256 -key hoka.key -out hoka-csr.pem`

```
C:\Users\[redacted]\test-certificate>openssl req -new -sha256 -key hoka.key -out hoka-csr.pem
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:FR
State or Province Name (full name) [Some-State]:Nantes
Locality Name (eg, city) []:Nantes
Organization Name (eg, company) [Internet Widgits Pty Ltd]:MicroEJ
Organizational Unit Name (eg, section) []:Customer Care
Common Name (e.g. server FQDN or YOUR name) []:MicroEJ
Email Address []:[redacted]

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:MicroEJ

C:\Users\[redacted]\test-certificate>dir
 Volume in drive C is WINDOWS
 Volume Serial Number is F40C-6E81

 Directory of C:\Users\[redacted]\test-certificate

05/23/2023  12:03 PM    <DIR>          .
05/23/2023  12:03 PM    <DIR>          ..
05/23/2023  11:59 AM             1,472 ca.crt
05/23/2023  11:58 AM             1,884 ca.key
05/23/2023  12:03 PM             1,846 hoka-csr.pem
05/23/2023  12:01 PM             3,294 hoka.key
               4 File(s)          8,496 bytes
               2 Dir(s)  38,371,045,376 bytes free
```

2. Use the CSR to generate a self signed certificate using openssl by executing the following command: openssl x509 -req -days 365 -in hoka-csr.pem -CA ca.crt -CAkey ca.key -CAcreateserial -out hoka.crt

Convert HOKA Private Key to DER Format

To convert the private key to DER format using openssl execute the following command:

openssl pkcs8 -inform PEM -in hoka.key -topk8 -outform DER -out hoka.der -v1 PBE-SHA1-3DES -passout pass:changeit

Note: In the HOKA SSL example, hoka.key corresponds to the above hoka.der .

Handle Encoding

Content And Transfer Encoding

The HTTP protocol specifies how to send the request/response payload (the body) with a specific encoding. To guarantee that the receiver can understand the encoded stream, HTTP has specified headers for encoding : content-encoding , transfer-encoding and accept-encoding . The HttpRequest and HttpResponse classes uses encoding handlers stored in the EncodingRegistry to, respectively, decode and encode the payloads with the relevant handler (ContentEncoding or TransferEnCoding). For the response, the accept-encoding the header value is used to determine the available encoding with the highest quality (acceptance value).

By default, the registry contains the “identity” encoding handler and the “chunked” transfer-coding handlers.

Request And Response Encoding

When parsing the request, HttpRequest wraps the body with the appropriate decoder or, if not found, sends a “406 Not Acceptable” response. The body-parser will receive the wrapped (decoded) stream as input to not have to deal with encodings. The same for HttpResponse uses the encoder wrapper to write the response into the encoded stream sent to the socket. Also, when using an input stream with unknown length as the response’s data, the transfer encoding used to send the response is “chunked”; otherwise, it is “identity”. When using a String as the response data, use the HttpResponse#setData(String, String) to specify the encoding of the string (by default, ISO-8859-1 is used).

URL Encoding

The percent-encoded special characters in the URI and in the query (parameters) are automatically decoded at parsing.

Session

HOKA provides tools to enable session management on the HTTP server.

Here is an example of how to use it.

```
// create a new session and store the user data in a session
final SessionHandler sessionHandler = new SessionHandler(new SecureRandom());
final Session session = this.sessionHandler.newSession();

//  for example from a login request handler
// ...  authenticate a user and store it user name into a session attribute
session.setAttribute("username", username);
// add a session cookie to the HttpResponse
response.addCookie("jsessionid", session.getId(), 0, false, true);


// from a protected request handler
// Get the session if from the cookie
String sessionId = request.getCookie("jsessionid");
Session session = this.session.getSession(sessionId); // get the session by it's id
// check if the user exists in the server session.
String␣
↪username = (String) session.getAttribute("username"); // access the username for example.
```

HOKA Configuration

The server can be configured by creating a property file in *src/main/resources* named *hoka.properties*

```
# Copyright 2021 MicroEJ Corp. All rights reserved.
# Use␣
↪of this source code is governed by a BSD-style license that can be found with this software.

# HOKA Server properties

# Use this property to set the logging level of the server.
# TRACE, DEBUG, INFO, WARN, ERROR, NONE
# the lower level activate all the others.
hoka.logger.level=INFO

# use this property␣
↪to set a custom logger. The custom logger must implement the interface ej.hoka.log.Logger
# if not set HOKA use a SimpleLogger implementation that logs to the standard output
# Ensure that your␣
↪logger is kept by the Soar by adding it to *.types.list properties file in the app resources.
#hoka.logger.class=

# I/O buffer size used to read/write data from/to request/response
#hoka.buffer.size=4096
```

This section presents networking libraries.

The following schema shows the overall architecture and modules:

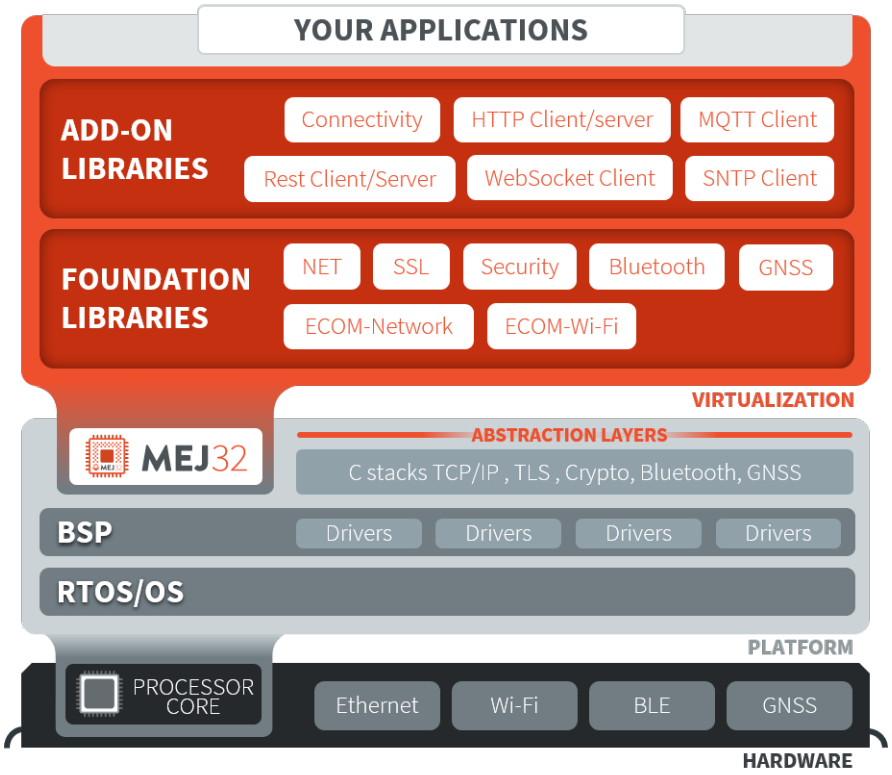


Fig. 162: Network Libraries Overview

- *Foundation Libraries*
- *Add-On Libraries*

Bluetooth

This section presents Bluetooth libraries.

Bluetooth API Library

Introduction

The Bluetooth API Library provides APIs to use BLE (Bluetooth Low Energy) in an Application.

Usage

The Bluetooth API Library is provided as a Foundation Library.

To use the Bluetooth API Library, add the following to the project build file:

Gradle (build.gradle.kts)

MMM (module.ivy)

```
implementation("ej.api:bluetooth:2.2.1")
```

```
<dependency org="ej.api" name="bluetooth" rev="2.2.1"/>
```

Building or running an Application which uses the Bluetooth API Library requires the VEE Port to provide the *Bluetooth Pack*.

Basic Knowledge and APIs

BLE is very different from TCP/IP networking. Like Wi-Fi, Bluetooth uses UHF radio waves to communicate over a short range, but it introduces an entirely unique protocol stack. It is important to understand how BLE works to develop an efficient and reliable Bluetooth application. This section explains the basics of BLE and how they are reified in the Bluetooth API.

Connection APIs

BLE introduces two roles of devices: the Central and Peripheral roles. A Central device scans nearby Peripheral devices and initiates the connection, while a Peripheral device advertises (broadcasts) and listens for connection requests. In this regard, a Central device can be thought of as a Wi-Fi Station while a Peripheral device can be thought of as a Wi-Fi Access Point.

The following sequence explains the typical connection flow between two devices:

- The Peripheral device starts advertising
- The Central device starts scanning
- The Central device initiates a connection with the Peripheral device

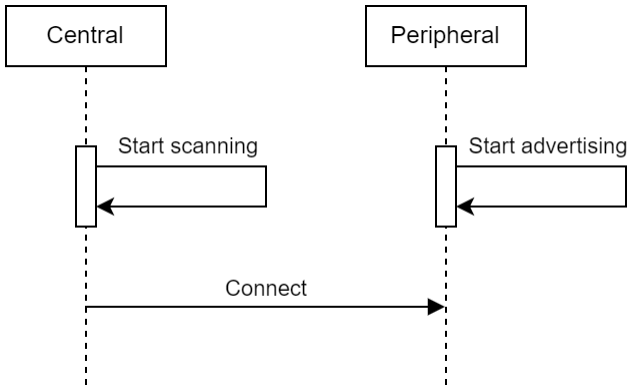


Fig. 163: Connection Procedure

A device must always enable its Bluetooth adapter using the BluetoothAdapter.enable() API before calling any other Bluetooth API. A Peripheral device can call the BluetoothAdapter.startAdvertising() API to start advertising. A Central device can call the BluetoothAdapter.startScanning() API to start scanning and the BluetoothAdapter.connect() API to initiate a connection. The BluetoothAdapter.stopAdvertising() and BluetoothAdapter.stopScanning() APIs can be called to stop advertising or scanning, however note that these operations are stopped automatically when a connection is established. A device must set the connection listener of the

adapter in order to be notified of asynchronous connection events, by calling the BluetoothAdapter.setConnectionListener() API. The ConnectionListener.onScanResult() hook is called on every scan result and the ConnectionListener.onConnected() hook is called when a connection is established.

Pairing APIs

Pairing is an optional procedure which allows to authenticate the connection by requesting a proof of possession (via a PIN code for example). The pairing procedure can be started at any time during a connection. It is often performed upon connecting or when first accessing a secure GATT service.

Here are the steps of the pairing procedure:

- Either device sends a pairing request or security request to the other device
- Both devices share their I/O capabilities
- If the I/O capabilities of the devices allow to create a connection with protection against MITM attacks, the Passkey Entry method is used:
  - The device with display capability displays a generated passkey on its user interface
  - The device with input capability reads the passkey from the user input and sends it to the device with display capability
  - The device with display capability checks that the passkey match
- Otherwise, the "Just Works" method is used and the pairing is complete. This method does not prevent from MITM attacks.

A device can call the BluetoothConnection.sendPairRequest() API to initiate pairing. The ConnectionListener.onPairRequest() hook is called when the device receives a pairing request. It can call the BluetoothConnection.sendPairResponse() API to accept or deny the pairing. The ConnectionListener.onPasskeyGenerated() hook is called when the device with display capability has generated a passkey to display. The ConnectionListener.onPasskeyRequest() hook is called when the device with input capability should provide the passkey. It can call the BluetoothConnection.sendPasskeyResponse() API to provide it.

GATT Services APIs

With BLE, devices exchange data through GATT services. BLE specifies standard services (such as the Current Time Service or the Battery Service) which allow devices to be interoperable, but BLE also allows to define custom services. Either device (Central or Peripheral or both) can provide services to the other. A device must discover the services provided by the other device before it can use them.

A device can define and provide services using the BluetoothServiceDefinition builder class and the BluetoothAdapter.addService() API. Once a connection is established, either device can discover the services of the other device by calling the BluetoothConnection.discoverServices() API. The ConnectionListener.onDiscoveryResult() hook is called for each service provided by the other device.

A service provides characteristics, which can be thought of as data channels. A characteristic has property flags, which indicate to the other devices how the characteristic can be used (whether it can be written, whether it provides notifications, etc.). A characteristic may have descriptors, which allow to describe or configure the characteristic in a specific way. Every attribute (characteristic or descriptor) has permission flags, which control its access (read-only,

read/write, requires authentication, etc.). Services and attributes are all identified by a 16-bit UUID. If a service or attribute is standard, the relevant specification indicates its UUID.

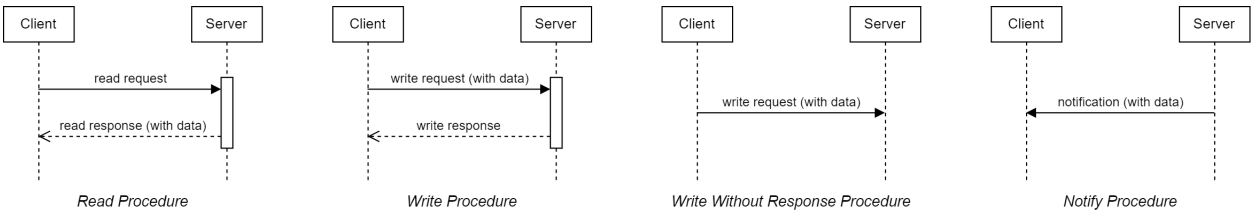
<table>
<tr><td colspan="2">Service</td></tr>
<tr><td rowspan="2">Characteristic 1</td><td>Descriptor 1</td></tr>
<tr><td>Descriptor 2</td></tr>
<tr><td rowspan="2">Characteristic 2</td><td>Descriptor 1</td></tr>
<tr><td>Descriptor 2</td></tr>
</table>

Fig. 164: Service Structure

A device can call the getter APIs of BluetoothService, BluetoothCharacteristic, BluetoothDescriptor and BluetoothAttribute to browse the content of a service.

BLE devices use characteristics to transfer data. There are 4 main procedures:

- The **Read** procedure allows the device which **discovered** the service to **request data**. The device sends a read request and the device which provides the service sends back a read response with the data.
- The **Write** procedure allows the device which **discovered** the service to **send data** and to require an acknowledgment. The device sends a write request with the data and the device which provides the service sends back a write response.
- The **Write Without Response** procedure allows the device which **discovered** the service to **send data** without expecting an acknowledgment. The device just sends a write request with the data.
- The **Notify** procedure allows the device which **provides** the service to **send data**. The device sends a notification with the data, and if it requires an acknowledgment, the device which discovered the service sends back an acknowledgment. It is a common practice to send notifications only to devices which have subscribed to the characteristic (a device can subscribe to a characteristic by sending a write request on its CCC descriptor).



For the Read procedure, a device can call the BluetoothConnection.sendReadRequest() API to send a read request. The LocalServiceListener.onReadRequest() hook is called when a device receives a read request. It can call the BluetoothConnection.sendReadResponse() API to send a read response with the data. The RemoteServiceListener.onReadCompleted() hook is called with the data when a device receives a read response.

For the Write Without Response and the Write procedures, a device can call the send BluetoothConnection.sendWriteRequest() API to send a write request with the data. The LocalServiceListener.onWriteRequest() hook is called with the data when a device receives a write request. It

can call the BluetoothConnection.sendWriteResponse() API to send a write response (in case of the write procedure). The RemoteServiceListener.onWriteCompleted() hook is called when a write request is sent (or when it receives a write response, in case of the write procedure).

For the Notify procedure, a device can call the send BluetoothConnection.sendNotification() API to send a notification with the data. The RemoteServiceListener.onNotificationReceived() hook is called with the data when a device receives a notification. The LocalServiceListener.onNotificationSent() hook is called when a notification is sent (or when it receives the acknowledgment, if one is required).

Classes Summary

Main classes:

- BluetoothAdapter (singleton): Performs operations not related to a specific device connection (scan, advertise, connect, provide GATT service)
- BluetoothConnection: Performs operations related to a specific device connection (disconnect, pair, discover GATT services, send GATT requests)
- BluetoothService: Represents a GATT service
- ConnectionListener and DefaultConnectionListener: Callbacks for all events not related to a specific GATT service
- LocalServiceListener and DefaultLocalServiceListener: Callbacks for events related to a specific provided GATT service
- RemoteServiceListener and DefaultRemoteServiceListener: Callbacks for events related to a specific discovered GATT service

Stateless and immutable classes:

- BluetoothAddress: Address (BD_ADDR) of a device
- BluetoothScanFilter: Scan result filter used when starting a scan
- BluetoothDataTypes: Data types enumeration used in advertisement payloads
- BluetoothCharacteristic: Represents a GATT characteristic
- BluetoothDescriptor: Represents a GATT descriptor
- BluetoothAttribute: Abstract base class of BluetoothCharacteristic and BluetoothDescriptor
- BluetoothUuid: UUID of a GATT service or GATT attribute
- BluetoothProperties: Properties enumeration used in GATT characteristics
- BluetoothPermissions: Permissions enumeration used when defining a GATT attribute
- BluetoothServiceDefinition: Builder class used when adding a GATT service
- BluetoothStatus: Status code enumeration used when reading/writing a GATT attribute

Use-Cases

Achieving Maximum Throughput

In some use-cases, such as when sending a large file to another device, the throughput must be as high as possible to decrease the transfer time.

Here are some guidelines to achieve the maximum throughput:

- Change the MTU to the maximum value (512 bytes) instead of the default value (23 bytes)
  - Once devices are connected, either device should send a MTU request with the maximum value
  - When the other device receives the MTU request, it should send a MTU response with the maximum value
  - Since there is no API for MTU exchange in the Bluetooth API Library, this step has to be performed in the native code
- Use a data transfer procedure which does not require an acknowledgment
  - If the service is provided by the device sending the data: use the Notify procedure without requesting an acknowledgment
  - If the service is discovered by the device sending the data: use the Write Without Response procedure
- Send the data chunks as fast as possible
  - Do not wait for the previous chunk to be delivered before sending the next chunk
  - If a chunk can not be delivered because the connection is congested, wait a bit and retry sending the chunk

Examples

MicroEJ provides two examples which show how to use the Bluetooth API. There is one example of Central device and one example of Peripheral device.

These examples can be found on GitHub. Please refer to their own README for more information on these examples.

Bluetooth Utility Library

Introduction

The Bluetooth Utility Library provides utility methods which can be useful when developing a Bluetooth Application. It depends on the *Bluetooth API Library*.

Usage

The Bluetooth Utility Library is provided as an Add-On Library.

To use the Bluetooth Utility Library, add the following to the project build file:

Gradle (build.gradle.kts)

MMM (module.ivy)

```
implementation("ej.library.iot:bluetooth-util:1.1.0")
```

```
<dependency org="ej.library.iot" name="bluetooth-util" rev="1.1.0"/>
```

Since this library is built on top of the *Bluetooth API Library*, it inherits its *requirements*.

Classes Summary

Main classes:

- AdvertisementData: Parses or builds an advertisement payload
- DescriptorHelper: Constants and utility methods related to GATT descriptors
- ServiceHelper: Utility methods related to GATT services

Stateless and immutable classes:

- AdvertisementFlags: Flags enumeration used in advertisement payloads
- AttributeNotFoundException: Exception thrown by ServiceHelper when a GATT attribute is not found

Audio

Introduction

The Audio API Library provides APIs to record and play audio in an Application.

This API can be used in various use-cases, such as:

- playing a sound
- playing a music streamed over IP or Bluetooth
- making a call
- synthesizing speech (text-to-speech)
- recognizing speech (speech-to-text)
- using a voice assistant such as Alexa or ChatGPT

Usage

The Audio API Library is provided as a Foundation Library.

To use the Audio API Library, add the following line to the project build file:

Gradle (build.gradle.kts)

MMM (module.ivy)

```
implementation("ej.api:audio:1.0.0")
```

```
<dependency org="ej.api" name="audio" rev="1.0.0"/>
```

Building or running an Application which uses the Audio API Library requires the VEE Port to provide the *Audio Pack*.

APIs

This section explains how audio concepts are reified in the Audio API.

Audio Format

When opening an audio stream, it is necessary to provide the format of the audio data. In the Audio API, the format is specified by the AudioFormat class.

The main property of an audio format is the encoding that is used to represent the data. Depending on the encoding used, the format may be described by an indeterminate number of parameters, such as the sample rate or the number of channels.

Although many audio encoding standards exist, the Audio API only provides a single AudioFormat implementation, PcmAudioFormat, which uses the PCM encoding. The Application can define additional audio formats using other encodings, but these encodings must be supported by the VEE Port to be used by the Application.

For example, the following snippet defines the "PCM 16kHz mono 16-bit little-endian signed" audio format:

```
AudioFormat FORMAT = new PcmAudioFormat(16_000, 1, 16, false, true);
```

Audio Recording

An audio recording stream can be opened by creating an AudioRecord instance. When creating an audio record, the format and the size of the native audio buffer must be provided. Since creating an AudioRecord instance allocates native resources, it should be closed with the AudioRecord.close() method in order to free these resources.

While the audio record is open, the native implementation records audio data continuously from the input device and writes it in the buffer. The AudioRecord.readBuffer() method can be used to retrieve and remove a chunk of audio data from the buffer. This method blocks until the requested size has been read or until the audio record is closed. If the audio data is not read fast enough by the application, the native implementation will discard the oldest audio data from the buffer.

For example, the following snippet records audio with an audio record and writes the audio data to a file:

```
try (OutputStream outputStream = new FileOutputStream("record.raw")) {
    try (AudioRecord audioRecord = new AudioRecord(FORMAT, 1600)) {
        byte[] buffer = new byte[480];
        while (true) {
            int bytesRead = audioRecord.readBuffer(buffer, 0, buffer.length); // read from audio record
            outputStream.write(buffer, 0, bytesRead); // write to file
        }
    }
}
```

Note: In order to avoid discontinuities in the recorded data, it is recommended to have a dedicated thread reading the buffer of the audio record. This should not be done in the UI thread as reading is a blocking operation that would prevent the UI thread from performing other tasks.

Audio Playback

An audio playback stream can be opened by creating an AudioTrack instance. When creating an audio track, the format and the size of the native audio buffer must be provided. Since creating an AudioTrack instance allocates native resources, it should be closed with the AudioTrack.close() method in order to free these resources.

While the audio track is open, the native implementation reads audio data continuously from the buffer and plays it on the output device. The AudioTrack.writeBuffer() method can be used to write a chunk of audio data in the buffer. This method blocks until the requested size has been written or until this audio track is closed. If audio data is not written fast enough by the application, the output device may play undesired silences. The AudioTrack.waitForBufferFlushed() method can be used to wait until all the audio data written in the buffer has been played back. The volume of the playback can be configured by calling AudioTrack.setVolume().

For example, the following snippet reads audio data from a resource and plays the audio with an audio track:

```
try (InputStream inputStream = MyClass.class.getResourceAsStream("/track.raw")) {
    try (AudioTrack audioTrack = new AudioTrack(FORMAT, 1600)) {
        byte[] buffer = new byte[480];
        while (true) {
            int bytesRead = inputStream.read(buffer, 0, buffer.length); // read from resource
            if (bytesRead == -1) { // EOF
                break;
            }
            audioTrack.writeBuffer(buffer, 0, bytesRead); // write to audio track
        }
        audioTrack.waitForBufferFlushed(); // play remaining audio data before closing
    }
}
```

Note: In order to avoid discontinuities in the audio playback, it is recommended to have a dedicated thread writing the buffer of the audio track. This should not be done in the UI thread as writing is a blocking operation that would

prevent the UI thread from performing other tasks.

Classes Summary

Main classes:

- AudioRecord: Represents an audio recording stream.
- AudioTrack: Represents an audio playback stream.

Stateless and immutable classes:

- AudioException: RuntimeException thrown when an error occurred during an audio operation.
- AudioFormat: Specifies the data format of an audio stream.
- PcmAudioFormat: AudioFormat which represents data with the PCM encoding.

Configuration

The *Audio Pack* can be configured by defining the following *Application Options*:

- audio.heap.size : defines the size of the Audio heap, in which the native buffers of the audio streams are allocated.
- s3.audio.input.device : defines the name of the Audio input device to use when running the Application on Simulator.
- s3.audio.output.device : defines the name of the Audio output device to use when running the Application on Simulator.

Examples

MicroEJ provides two examples which show how to use the Audio API: one example for audio recording and one for audio playback.

These examples can be found on GitHub. Please refer to their own README for more information on these examples.

MicroAI

Introduction

The MicroAI Library provides APIs to interact with trained Machine Learning models, especially to run inferences.

Usage

The MicroAI Library is provided as a Foundation Library.

To use the MicroAI Library, add the following line to the project build file:

```
implementation("ej.api:microai:2.1.0")
```

Building or running an Application which uses the MicroAI Library requires a SDK6 VEE Port that provides the *MicroAI Pack*.

Machine Learning Model Format

MicroAI API is designed to be **framework-agnostic**, meaning it does not rely on a specific Machine Learning framework like TensorFlow or ONNX.

The Application is responsible of loading the model file. Therefore, before developing an Application with MicroAI, check which model file formats are supported by your target VEE Port.

MicroEJ Simulator

If you need to use the MicroEJ Simulator, you must use a model in **TensorFlow Lite for Microcontrollers (TFLM)** format. Other model formats will not be compatible with the MicroEJ Simulator and cannot be executed within it.

Tensorflow Lite for Microcontrollers supports a limited subset of TensorFlow operations, which impacts the model architectures that it is possible to run. The supported operators list corresponds to the list in the all_ops_resolver.cc file.

APIs

MLInferenceEngine

The first action when working with MicroAI is to load the trained Machine Learning model using MLInferenceEngine class.

There are 2 ways to load a model:

- From an application resource with MLInferenceEngine(String modelPath, int inferenceMemoryPoolSize) constructor.
- From an InputStream using MLInferenceEngine(InputStream is, int inferenceMemoryPoolSize) constructor.

The MLInferenceEngine constructor will:

1. Map the model into a native data structure.
2. Build an interpreter to run the model with.
3. Allocate memory for the model's tensors.

Using an Input Stream Model

When using MLInferenceEngine(InputStream is, int inferenceMemoryPoolSize), the model is loaded inside the MicroAI heap. The size of MicroAI heap is defined from the *MicroAI Configurations*.

Note that the call to MLInferenceEngine(InputStream is, int inferenceMemoryPoolSize) will block until the model is completely retrieved/loaded.

Using an Inference Memory Pool

When using TensorFlow Lite, the tensors are allocated dynamically into the system heap.

However when using Tensorflow Lite Micro, we must configure a *inferenceMemoryPoolSize*, which is called Arena Size, where all the input, output and intermediate tensors will be allocated. This helps achieve deterministic memory usage.

To figure out which minimal value can be set, try large enough values using the Simulator, until the MLInferenceEngine succeeds. At which point you should see such log, which will help to fine tune the *inferenceMemoryPoolSize* value:

```
[microai mock] MicroInterpreter uses 1112 bytes, use this value to optimize the Arena Size
```

Note: This example is very specific to the backend used.

Code Example

Once initialized, MLInferenceEngine allows to get input/output model tensors and to run inferences on the trained model.

For example, the following snippet loads a trained model from the application resources and runs an inference on it:

```
try(MLInferenceEngine mlInferenceEngine = new MLInferenceEngine(
↪"/model.tflite", MEMORY_POOL_SIZE)) { // Initialize the inference engine.
   InputTensor inputTensor␣
↪= mlInferenceEngine.getInputTensor(0); // Get input tensor of the trained model.
   /*
    * Fill the input tensor
    */
   mlInferenceEngine.run(); // Run inference on the trained model.
   OutputTensor outputTensor␣
↪= mlInferenceEngine.getOutputTensor(0); // Get output tensor of the trained model.
   /*
    * Process output data
    */
}
```

Tensor

Tensor parameters can be retrieved from the Tensor class.

It allows to get some useful information such as the data type, the number of dimensions, the number of elements, the size in bytes or the quantization parameters.

There are 2 kinds of tensors:

- InputTensor: Offers services to load input data inside MicroAI input tensors before running an inference. Tensor input data must be one of the types supported by MicroAI (see Tensor.DataType).
- OutputTensor: Offers services to retrieve output data from MicroAI output tensors after running an inference. Tensor output data must be one of the types supported by MicroAI (see Tensor.DataType).

Classes Summary

Main classes:

- MLInferenceEngine: Loads a model, get its tensors and runs inferences on it.
- Tensor: Retrieves a tensor information.
- InputTensor: Loads input data before running an inference.
- OutputTensor: Retrieves output data after running an inference.

Stateless and immutable classes:

- Tensor.DataType: Enumerates MicroAI data types.
- Tensor.QuantizationParameters: Represents quantized parameters of a tensor.

Configuration

The *MicroAI Pack* can be configured by defining the following *Application Options*:

- microai.heap.size : defines the size of the MicroAI heap, in which the InputStream models are allocated.

Example

For example, the following snippet runs inference on model that takes 1 quantized element as input and outputs 1 float value:

```
try(MLInferenceEngine mlInferenceEngine = new MLInferenceEngine(
↪"/model.tflite", MEMORY_POOL_SIZE)) { // Initialize the inference engine.
   InputTensor inputTensor␣
↪= mlInferenceEngine.getInputTensor(0); // Get input tensor of the trained model.
   byte[] inputData = new␣
↪byte[inputTensor.getNumberElements()]; // Create an array that fits size of input tensor.

   // Fill inputData with quantized value.
   float realValue = 10f;
   Tensor.QuantizationParameters quantizationParameters␣
↪= inputTensor.getQuantizationParams(); // Get quantization parameters.
   inputData[0] = (byte) (realValue / quantizationParameters.
↪getScale() + quantizationParameters.getZeroPoint()); // Quantize the input value.
```

(continues on next page)

(continued from previous page)

```
    inputTensor.setInputData(inputData); // Load input data inside MicroAI input tensor.

    mlInferenceEngine.run(); // Run inference on the trained model.

    OutputTensor outputTensor␣
↪= mlInferenceEngine.getOutputTensor(0); // Get output tensor of the trained model.
    float[] outputData = new␣
↪float[outputTensor.getNumberElements()]; // Create an array that fits size of output tensor.

    // Retrieve and print inference result.
    outputTensor.
↪getOutputData(outputData); // Retrieve output data from MicroAI output tensor.
    System.out.println("Inference result with " + realValue + " input is " + outputData[0]);
}
```

Date and Time

Introduction

Java developers have long used the Date, Calendar and TimeZone classes for handling date and time. Java SE 8 introduced a more advanced and comprehensive Date and Time API that goes beyond simply replacing Date or Calendar. It provides a complete time model for applications.

There are many benefits of using the latest:

- Immutability: types are immutable, making thread-safe code easier to write and less prone to bugs (due to no mutable state).
- Improved API design: it offers an intuitive and developer-friendly design that better addresses the challenges of date and time manipulation. Application code is also easier to read and understand.
- Simplified date and time arithmetic: the API introduces methods for common date and time operations, simplifying tasks like adding or subtracting days, months, or years.
- Precision: it provides more precise representations for date and time values, including support for nanoseconds, which is important for applications requiring high precision.
- Comprehensive time model: it introduces new classes that deal with different concepts of time such as date without a time or time without a date, durations or periods.

  In general, it's a good practice to use the Java Time API when dealing with date, time, and time zone-related operations because of its convenient features and extensive capabilities. Yet, for straightforward timestamp handling or lightweight applications, System.currentTimeMillis() can be adequate. One aspect to keep in mind is that the Time API offers better readability and advanced operations, which might be missing when using timestamp manipulation or older APIs.

Overview

The library introduces different classes for date, time, date-time, and variations for offset and time zone. While this may seem like a lot of classes, most applications can start with only these types:

- Instant: an instantaneous point on the timeline. It can be used to store timestamps of application events.
- LocalDate: stores a date without a specific time or time zone, like 2023-09-26 .
- LocalTime: stores a time without a specific date or time zone, like 15:30 .
- LocalDateTime: stores both a date and time without a specific time zone, like 2023-09-26T15:30 . It combines LocalDate and LocalTime.
- ZonedDateTime: stores both a date and time, including a time zone. This is handy for performing precise date and time calculations while considering the time zone.
- Duration: a duration of time, measured in hours, minutes, seconds, and nanoseconds.
- Period: a duration of time in terms of years, months, and days.

Note: Working with a time zone can make calculations more complex. In many cases, the application can only work with LocalDate, LocalTime, and Instant, and then add the time zone at the user interface (UI) level.

The API has many methods, but it remains easy to handle because it sticks to consistent method prefixes:

- of : static factory method.
- get : gets a value.
- is : checks if some condition is true.
- with : equivalent to a setter for immutable objects, returns a copy with the specified argument set.
- plus : adds an amount to an object.
- minus : subtracts an amount from an object.
- to : converts this object to another type.
- at : combines this object with another.

Usage

The Date and Time API is provided as an Add-on Library.

To use the time library, add the following to the project build file:

Gradle (build.gradle.kts)

MMM (module.ivy)

```
implementation("ej.library.eclasspath:time:1.1.0")
```

```
<dependency org="ej.library.eclasspath" name="time" rev="1.1.0"/>
```

Examples

This section presents a series of small, focused examples that demonstrate various aspects of the Java Date and Time API.

Instant

The Instant class is the closest equivalent of Date. It represents a specific instant in time.

```
// Creating instants
Instant now = Instant.now(); // now
Instant␣
↪someInstant = Instant.ofEpochSecond(1695732445L); // September 26, 2023 12:47:25 PM

// Displaying
System.out.println("Seconds elapsed since epoch " + now.getEpochSecond());

// Chaining operations on instants
long secondsUntil = someInstant.plusSeconds(10).until(now, ChronoUnit.SECONDS);
System.out.println("Amount of time until another instant in seconds: " + secondsUntil);
```

LocalDate

LocalDate stores a date without a time. It is called "local" because it isn't associated with any specific time zone, similar to a wall clock. It simplifies date operations by dealing only with dates, making it suitable for scenarios not requiring time zone concerns (e.g., booking systems, calendars, date validation, etc.).

```
// Creating LocalDate instances
LocalDate today = LocalDate.now(); // Current date
LocalDate specificDate = LocalDate.of(2023, Month.JULY, 15); // July 15, 2023

// Displaying LocalDate instances
System.out.println("Today's Date: " + today);
System.out.println("Specific Date: " + specificDate);

// Performing operations
LocalDate futureDate = today.plusDays(30); // Adding 30 days to today
LocalDate pastDate = today.minusMonths(2); // Subtracting 2 months from today

// Displaying the results of operations
System.out.println("Date 30 days from today: " + futureDate);
System.out.println("Date 2 months ago from today: " + pastDate);

// Comparing LocalDate instances
boolean isAfter = specificDate.isAfter(today); // Check if specificDate is after today

// Displaying comparison results
System.out.println("Is specificDate after today? " + isAfter);
```

LocalTime

LocalTime stores a particular time of day, focusing only on the time (hour, minute, second, nanosecond), and doesn't include date or time zone details. Useful when you only need to handle time values without dates or time zones (e.g., scheduling events like alarms, stopwatch and timers, event timing, etc.).

```
// Creating LocalTime instances
LocalTime now = LocalTime.now(); // Current time
LocalTime specificTime = LocalTime.of(14, 30); // 2:30 PM

// Displaying LocalTime instances
System.out.println("Current Time: " + now);
System.out.println("Specific Time: " + specificTime);

// Performing operations
LocalTime futureTime = now.plusHours(3); // Adding 3 hours to the current time
LocalTime pastTime = now.minusMinutes(15).
↪minusSeconds(29); // Subtracting 15 minutes and 29 seconds from the current time

// Displaying the results of operations
System.out.println("Time 3 hours from now: " + futureTime);
System.out.println("Time 15 minutes ago: " + pastTime);

// Displaying time fields
System.out.println("Hour: " + now.getHour());
System.out.println("Minute: " + now.getMinute());
System.out.println("Second: " + now.getSecond());
```

LocalDateTime

LocalDateTime combines both date and time components and provides a precise timestamp. This makes it suitable for scenarios where you need to work with both date and time information, but without considering time zone conversions (e.g., timestamping, user interfaces, etc.).

```
// Creating LocalDateTime instances
LocalDateTime now = LocalDateTime.now(); // Current date and time
LocalDateTime specificDateTime␣
↪= LocalDateTime.of(2023, Month.JULY, 15, 14, 30); // July 15, 2023, 2:30 PM

// Displaying LocalDateTime instances
System.out.println("Current Date and Time: " + now);
System.out.println("Specific Date and Time: " + specificDateTime);

// Performing operations
LocalDateTime␣
↪futureDateTime = now.plusDays(30).plusHours(3); // Adding 30 days and 3 hours to now
LocalDateTime pastDateTime␣
↪= now.minusMonths(2).minusMinutes(15); // Subtracting 2 months and 15 minutes from
                                                    // now
// Displaying the results of operations
System.out.println("Date and Time 30 days and 3 hours from now: " + futureDateTime);
System.
↪out.println("Date and Time 2 months and 15 minutes ago from now: " + pastDateTime);

// Displaying date and time fields
```

(continues on next page)

(continued from previous page)

```
System.out.println("Year: " + now.getYear());
System.out.println("Month: " + now.getMonth());
System.out.println("Day of Month: " + now.getDayOfMonth());
System.out.println("Hour: " + now.getHour());
System.out.println("Minute: " + now.getMinute());
System.out.println("Second: " + now.getSecond());
System.out.println("Day of Year: " + now.get(ChronoField.DAY_OF_YEAR));
System.out.println("Day of Week: " + now.get(ChronoField.DAY_OF_WEEK));

// Displaying comparison results
System.out.println(
↪"Is specificDateTime after current date and time? " + specificDateTime.isAfter(now));
```

Duration

Duration represents a duration of time, typically measured in hours, minutes, seconds, and nanoseconds. It is used to calculate and work with time intervals, such as the amount of time between two points in time or the duration of an event. It is suitable for tasks involving precise timing, such as measuring time elapsed or setting timeouts.

```
// Creating Duration instances
Duration fiveHours = Duration.ofHours(5); // Duration of 5 hours
Duration thirtyMinutes = Duration.ofMinutes(30); // Duration of 30 minutes
Duration twoSeconds = Duration.ofSeconds(2); // Duration of 2 seconds

// Displaying Duration instances
System.out.println("5 Hours: " + fiveHours);
System.out.println("30 Minutes: " + thirtyMinutes);
System.out.println("2 Seconds: " + twoSeconds);

// Performing operations
Duration␣
↪combinedDuration = fiveHours.plus(thirtyMinutes).plusSeconds(10); // Adding durations
Duration subtractedDuration = fiveHours.minus(twoSeconds); // Subtracting durations

// Displaying the results of operations
System.out.println("Combined Duration: " + combinedDuration);
System.out.println("Subtracted Duration: " + subtractedDuration);

// Displaying duration fields
System.out.println("Hours: " + combinedDuration.toHours());
System.out.println("Minutes: " + combinedDuration.toMinutes());
System.out.println("Seconds: " + combinedDuration.getSeconds());

// Comparing Duration instances
boolean isLonger =␣
↪fiveHours.compareTo(thirtyMinutes) > 0; // Check if fiveHours is longer than thirtyMinutes
boolean␣
↪isEqual = fiveHours.equals(Duration.ofHours(5)); // Check if fiveHours is equal to 5 hours

// Displaying comparison results
System.out.println("Is fiveHours longer than thirtyMinutes? " + isLonger);
System.out.println("Is fiveHours equal to 5 hours? " + isEqual);
```

Period

Period represents a duration of time in terms of years, months, and days. It is primarily concerned with human-centric time measurements, like the length of a month or a year. It is well-suited for measuring time intervals within a calendar context. For example, it can represent periods such as 2 years, 3 months, and 5 days.

```
// Creating LocalDate instances
LocalDate date1 = LocalDate.of(2021, 6, 15); // June 15, 2021
LocalDate date2 = LocalDate.of(2023, 9, 25); // September 25, 2023

// Calculating the period between two dates
Period period = Period.between(date1, date2);

// Displaying the period
System.out.println("Period between " + date1 + " and " + date2 + ": " + period);

// Displaying period fields
System.out.println("Years: " + period.getYears());
System.out.println("Months: " + period.getMonths());
System.out.println("Days: " + period.getDays());

// Creating Period instances using factory methods
Period customPeriod = Period.of(2, 3, 5); // 2 years, 3 months, and 5 days

// Displaying the custom period
System.out.println("Custom Period: " + customPeriod);

// Performing operations on periods
Period addedPeriod = period.plus(customPeriod); // Adding periods
Period subtractedPeriod = period.minus(customPeriod); // Subtracting periods

// Displaying the results of operations
System.out.println("Added Period: " + addedPeriod);
System.out.println("Subtracted Period: " + subtractedPeriod);

// Comparing Period instances
boolean isEqual = customPeriod.
↪equals(Period.of(2, 3, 5)); // Check if customPeriod is equal to 2 years, 3 months, and 5 days

// Displaying comparison results
System.out.println("Is customPeriod equal to 2 years, 3 months, and 5 days? " + isEqual);
```

Time Zone Support

The Time API introduces multiple types for time zone management:

- ZoneId : represents a time zone identifier: a fixed offset (e.g., +0200) or a geographical region (e.g., Africa/Johannesburg).
- ZoneOffset : represents a fixed time zone offset from UTC, usually a fixed number of hours and minutes.
- ZonedDateTime : a date time with a time zone: the combination of a LocalDateTime and a ZoneId.
- ZoneRules : defines the offsets from UTC, the daylight saving time rules, and how they change over time, for a specific time zone.
- ZoneRulesProvider : provides the time zone rules to all the zone-aware classes of the library. Meant to be implemented by custom time zone rule providers.

Default Zone Rules Provider

By default, the library uses a lightweight provider designed to handle only the time zone rules for GMT (Greenwich Mean Time). This is suitable for operations on dates and times that do not depend on time zone considerations. This provider only supports the GMT zone ID. Any attempt to get a different zone ID will throw a ZoneRulesException. For example,

```
ZoneId.of("GMT"); // ok
ZoneId.of("+0200"); // ok
ZoneId.of("GMT+0530"); // ok
ZoneId.of("PST"); // throws ZoneRulesException
ZoneId.of("CST-0115"); // throws ZoneRulesException
ZoneId.of("Asia/Tokyo"); // throws ZoneRulesException

Set<String> zoneIds = ZoneId.getAvailableZoneIds(); // returned set contains only "GMT"
```

TZDB Zone Rules Provider

The library also defines a provider of zone rules for the time zones defined in the IANA Time Zone Database (TZDB).

The TZDB provider reads the zones and rules from a raw resource at runtime. Compared to the TzdbZoneRulesProvider of Java SE distributions, this implementation uses less Managed Heap at runtime, making it more suitable for embedded devices.

> **Warning:** The TZDB provider requires a target VEE Port that uses an architecture version 8.1.1 minimum (for 8.x), or 7.20.5 minimum (for 7.x).

Using the TZDB Provider

To use this provider, set the constant java.time.zone.DefaultZoneRulesProvider to java.time.zone.TzdbZoneRulesProvider in a *.constants.list file, like below:

```
java.time.zone.DefaultZoneRulesProvider=java.time.zone.TzdbZoneRulesProvider
```

It is also required to add the class name java.time.zone.TzdbZoneRulesProvider to a *.types.list file: the class name is required to instantiate the provider and can not be known at compile-time.

The raw resource from which the provider reads the zone rules is generated from the timezone database file included in the JDK/JRE installation (tzdb.dat). To generate the resource and use it in an application, do the following:

1. Locate the tzdb.dat file in a local JDK/JRE installation (path/to/JRE/lib/tzdb.dat),
2. Add the tzdb.dat file to the application resources (e.g., src/main/resources/com/mycompany/tzdb.dat),
3. Create a *.tzdb.list file in the application resources (e.g., src/main/resources/com/mycompany/myapp.tzdb.list),
4. Open the *.tzdb.list file and add the path to the tzdb.dat file (e.g., /com/mycompany/tzdb.dat).

The resource will be automatically generated when building the application or running it in the Simulator. By default, it will be embedded in the application binary (as an *internal resource*). For reference, the size of the resource is 100 KB for version 2024a .

Note: To get a tzdb.dat with the most current timezone data available, use the TZUpdater tool and run the following command:

```
java -jar tzupdater.jar -l
```

The TZUpdater tool updates the JDK/JRE instance that is used to execute the tool: copy path/to/JRE/lib/tzdb.dat into the application resources, as described above.

In addition, you can check that the version of the timezone data is correct in the logs of the Add-on Processor that generates the raw resource.

SDK 6

SDK 5

When running on Simulator or building an executable in verbose mode (with the --info Gradle option), look in the console for an output similar to:

```
[myapp:tzdb] Successfully␣
↪deserialized TZDB data: version = 2024a, zones count = 603, resource buffer size = 102532
```

where 2024a is the version of the timezone data in this example.

The Add-on Processor is executed when changes occur in the resources files. Open the Add-on Processor console and set the log level to debug . After copying a tzdb.dat file in the resources files, look in the console for an output similar to:

```
[myapp:tzdb] Successfully␣
↪deserialized TZDB data: version = 2024a, zones count = 603, resource buffer size = 102532
```

where 2024a is the version of the timezone data in this example.

If the TZDB provider can't find the resource, it will throw an exception at runtime:

```
Exception in thread "main" java.lang.ExceptionInInitializerError:␣
↪java.lang.IllegalStateException: Cannot open the tzdb binary resource
```

In this case, follow the steps described above to generate the resource, and make sure that the resource is available at runtime: when the resource is internal, the tzdb resource should be listed in the Application Resources group of the *SOAR.map* file.

Loading the TZDB Data as an External Resource

The resource can be declared as an external resource to be loaded from another location (e.g. from a FileSystem). It has to be referenced in a *.externresources.list file, in which case the SOAR will output the resource in the *External Resources Folder*. See *Application Resources* for more information about external resources.

Warning: Loading external resources requires a target VEE Port that uses the *External Resources Loader*.

Follow the steps below to declare the `tzdb` resource as an external resource:

- Create a `*.externresources.list` file in the `src/main/resources/` folder,
- Add the following path to the file:

```
/java/time/zone/tzdb
```

- Build the application executable,
- The raw resource is now available in the *External Resources Folder*. This resource must be transferred to the target device's memory and loaded from the path `/java/time/zone/tzdb`, using the *External Resources Loader*.

Migration Guide

If you're using the legacy date and time classes (Date, Calendar), it's a great time to consider migrating to the new API. This small migration guide will help you transition from the old time API to the Java Date and Time API (`java.time`). It covers some common date and time operations and demonstrates how to perform them using both approaches.

Displaying the Current Date

Legacy Time API

New Time API

```
// Create a Calendar instance representing the current date and time
Calendar calendar = Calendar.getInstance();

// Get date components from the Calendar
int year = calendar.get(Calendar.YEAR);
int month = calendar.get(Calendar.MONTH) + 1; // Months are 0-based
int day = calendar.get(Calendar.DAY_OF_MONTH);

// Display the date
System.out.println("Current Date: " + year + "-" + month + "-" + day);
```

```
// Get the current date using LocalDate
LocalDate currentDate = LocalDate.now();

// Display the date
System.out.println("Current Date: " + currentDate);
```

Calculating a Timestamp from a Date

Legacy Time API

New Time API

```
// Create a Calendar instance
Calendar calendar = Calendar.getInstance();
calendar.set(2023, 10, 06, 15, 27, 30);      // November 06, 2023 3:27:30 PM
long timeInMillis = calendar.getTimeInMillis();
```

```
// Create a LocalDateTime instance with the desired date and time
LocalDateTime localDateTime = LocalDateTime.of(2023, 10, 06, 15, 27, 30);

// Convert LocalDateTime to a timestamp from Epoch
long timeInMillis = localDateTime.toInstant(ZoneOffset.UTC).toEpochMilli();
```

Calculating Date and Time Differences

Legacy Time API

New Time API

```
public long computeDifference(Date date1, Date date2){
    return date1.getTime() - date2.getTime();
}
```

```
public long computeDifference(LocalDateTime date1, LocalDateTime date2){
    return Duration.between(date1, date2).toMillis();
}
```

Calculating the Day of the Week

Legacy Time API

New Time API

```
// Create a Calendar instance
Calendar calendar = Calendar.getInstance();

// Set a date (e.g., October 15, 2023)
calendar.set(2023, Calendar.OCTOBER, 15);

// Get the day of the week as an integer (1 = Sunday, 2 = Monday, ..., 7 = Saturday)
int dayOfWeek = calendar.get(Calendar.DAY_OF_WEEK);
```

```
// Create a LocalDate instance for a specific date (October 15, 2023)
LocalDate date = LocalDate.of(2023, 10, 15);

// Get the day of the week as an enum value (DayOfWeek)
DayOfWeek dayOfWeek = date.getDayOfWeek();
```

Handling Time Zones

Legacy Time API

New Time API

```
TimeZone timeZone = TimeZone.getTimeZone("America/New_York");
Calendar calendar = Calendar.getInstance(timeZone);
Date dateInNewYork = calendar.getTime();
```

```
ZoneId zoneId = ZoneId.of("America/New_York");
ZonedDateTime zonedDateTime = ZonedDateTime.now(zoneId);
```

Restrictions

The library's goal is to offer Application developers an API that closely mirrors the one found in Java SE 8. However, we had to make the library compatible with both pre-Java 8 features and the constraints found in embedded devices, such as limited memory size. Here are the items where the backport differs from its Java 8 counterpart:

- Non-ISO chronologies are not present (*Hijrah*, *Japanese*, *Minguo*, *ThaiBuddhist*). The overwhelming majority of applications use the ISO calendar system. Applications still have the option to introduce their own chronologies.
- No formatting or parsing methods (methods `parse`, `format`, `getDisplayName`, `ofLocale`).
- Removed the method `ZoneRulesProvider.registerProvider(ZoneRulesProvider provider)`. The unique provider is defined with the constant `java.time.zone.DefaultZoneRulesProvider`.
- Static methods in interfaces are not supported and were removed or moved (see below).
- Default methods in interfaces are not supported and were removed (pulled down in concrete types).
- Removed static methods `TemporalAdjusters.ofDateAdjuster(UnaryOperator<LocalDate> dateBasedAdjuster)` and `WeekFields.of(Locale locale)`.
- No overflow checks on calculations (removed `throws ArithmeticException` when relevant). Excessively checking for overflow in all calculations can impact performance negatively.
- No null checks on method arguments. Developers are encouraged to use the *Null Analysis* tool to detect null access and adhere to the API Javadoc specifications.
- The classes from the `java.time.zone` package do not provide a human readable implementation of `toString()`.

Note: For a comprehensive list of restrictions, refer to the `README` of the module. If some of the restrictions listed above are highly limiting and necessary for your application, please contact your MicroEJ sales representative or *our support team*.

Static Interface Methods

- `ChronoLocalDate.from(TemporalAccessor)` : *removed*
- `ChronoLocalDate.timeLineOrder()` : use `LocalDate.timeLineOrder()` instead
- `ChronoLocalDateTime.from(TemporalAccessor)` : *removed*
- `ChronoLocalDateTime.timeLineOrder()` : use `LocalDateTime.timeLineOrder()` instead
- `ChronoZonedDateTime.from(TemporalAccessor)` : *removed*
- `ChronoZonedDateTime.timeLineOrder()` : use `ZonedDateTime.timeLineOrder()` instead
- `ChronoPeriod.between(ChronoLocalDate, ChronoLocalDate)` : *removed*
- `Chronology.from(TemporalAccessor)` : use `AbstractChronology.from(TemporalAccessor)` instead

- Chronology.getAvailableChronologies() : use AbstractChronology.getAvailableChronologies() instead
- Chronology.of(String) : use AbstractChronology.of(String) instead
- Chronology.ofLocale(Locale) : *removed*

Event Queue

Principle

The Event Queue Foundation Library provides an asynchronous communication interface between the native world and the Managed world based on events.

Functional Description

Overview

The Event Queue Foundation Library allows users to send events from the native world to the Managed world. It is composed of a Java API that provides mechanisms to register specific event notifications and a C API that allows someone to send events in the queue.

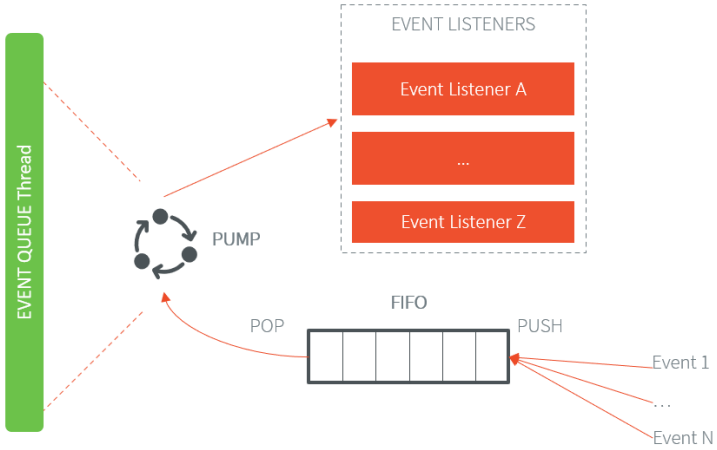


Fig. 165: Event Queue Overview

A FIFO mechanism is implemented on the native side and is system specific. The user can offer events to this FIFO by using the C or the Java API.

Event notifications are handled using event listeners (Observer design pattern). The application code has to register event listeners to be notified when new events are coming.

Then the Event Pump automatically retrieves new events pushed in the FIFO and notifies the event listeners.

Architecture

The Event Queue Foundation Library uses a dedicated thread to forward and process events. Application event listener's calls are done in the context of the Event Queue thread.

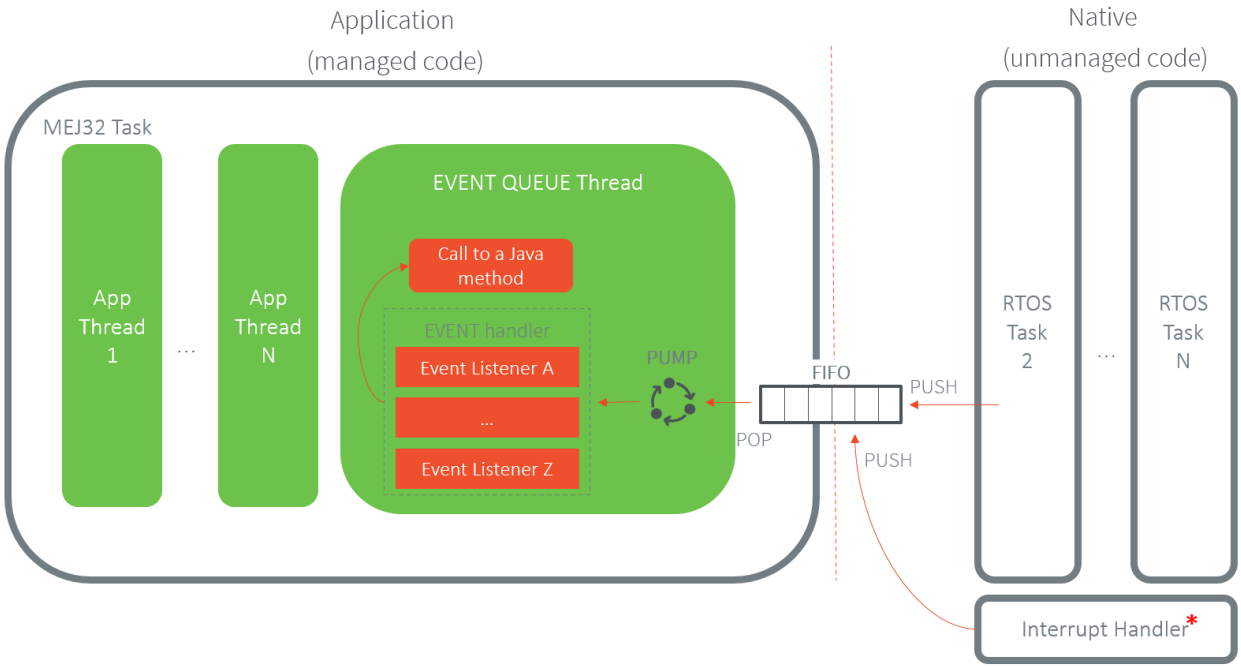


Fig. 166: Event Queue Architecture

Events reading operations are done using the SNI mechanism. Event Queue thread is suspended when the events FIFO is empty and resumed when a new event is sent.

Note: To support sending events from the Interrupt Handler, the VEE Port must provide a compatible implementation.

Event format

An event is composed of a type and, optionally, data. The type identifies the listener that will handle the event. The data is application specific and passed to the listener as a raw byte array.

The items stored in the FIFO buffer are integers (4 bytes). There are two kinds of events that can be sent over the Event Queue:

- Standard event: an event with data that fits on 24 bits. The event is stored in the FIFO as a single 4 bytes item.
- Extended event: an event with data that does not fit on 24 bits. The event is stored in the FIFO as multiple 4 bytes items.

```
+-------------+---------+-----------------------------------------------+
| Extended (1) | Type (7) | Data (if Extended==0), Length (if Extended==1) (24) |
+-------------+---------+-----------------------------------------------+
```

(continues on next page)

(continued from previous page)

```
...
+-------------------------------------------------------------------------+
|                Extended Data for extended events (32)                   | (Length bytes)
+-------------------------------------------------------------------------+
```

Format explanation:

- *Extended* (1 bit): event kind flag (0 for standard event, 1 for extended event).
- *Type* (7 bits): event type, which allows to find the corresponding event queue listener.
- *Length* (24 bits): length of the data in bytes (for extended events only).
- *Data* (24 bits): standard event data (for standard events only).
- *Extended data* (Length bytes): extended event data (for extended events only).

Event Queue listener

An application can register listeners to the Event Queue. Each listener is registered for a specific event type. The same listener can be registered several times for different event types, but each event type can only have one listener.

When the queue receives an event from the FIFO, it will get the event type and check if it is an extended event. Then it will check if a listener is registered for this event type. If so, it will call its handle method depending on the extended event flag. It will call the default listener if no listener corresponds to the event type.

You can create your Event Queue listener by implementing the EventQueueListener interface. It contains two methods that are used to handle standard and extended events.

You can register your listener using the registerListener(EventQueueListener listener) method from the EventQueue class. Internally, the Event Queue maintains a FIFO-based counter that sequentially allocates type IDs to registered listeners.

The unique type your listener uses could be stored on the Managed world and passed/stored to the C world. One way to do it is to create a native method that sends the event type to the C world during the initialization phase.

To set the default listener, you must use EventQueue.setDefaultListener(EventQueueListener listener) .

For example:

```
public static int eventType;

public static void main(String[] args) throws InterruptedException {
   EventQueue eventQueue = EventQueue.getInstance();

   // Create and register a listener.
   // eventType must be stored if you want to offer an event from the Java API.
   eventType = eventQueue.registerListener(new ExampleListener());

   // Send eventType to the C world.
   initialize(eventType);
}

/**
```

(continues on next page)

(continued from previous page)

```
* This native method will take the event type as an entry and store it in the C world.
*/
public static native void setupNativeEventType(int type);
```

Standard event

Standard events are events with data that can be stored on 24 bits.

```
+-------+---------+----------+
| 0 (1) | Type (7) | Data (24) |
+-------+---------+----------+
```

The first bit equals 0, indicating that this is a standard event.

Then there is the event type stored on 7 bits.

To finish, there is the data that you want to send to the application event listener. It is stored on 24 bits.

Offer the event

There are two ways to send a standard event through the Event Queue: from the C API or the Java API.

From C API

To send a standard event through the Event Queue using the C API, you must use the `LLEVENT_offerEvent(int32_t type, int32_t data)` method from `LLEVENT.h`.

For example:

```
// Assuming that event_type has␣
↪been passed from the Managed world through a native method after registering your listener.
int type = event_type;
int data = 12;

LLEVENT_offerEvent(type, data);
```

From Java API

To send a standard event through the Event Queue using the Java API, you must use the `offerEvent(int type, int data)` method from the `EventQueue` class.

For example:

```
EventQueue eventQueue = EventQueue.getInstance();

// Assuming␣
↪that eventType has been stored in the Managed world when you registered the listener.
int type = eventType;
int data = 12;
```

(continues on next page)

(continued from previous page)

```
eventQueue.offerEvent(type, data);
```

Handle the event

To handle a standard event, you must implement your listener handleEvent(int type, int data) method. You can process the data received by the Event Queue in this method.

First, you have to register your listener as explained *Event Queue listener* in section.

For example:

```
EventQueue queue = EventQueue.getInstance();
int type = queue.registerListener(new EventQueueListener() {

   @Override
   public void handleEvent(int type, int data) {
      System.out.println("My data is equal to: " + data);
   }
   @Override
   public void handleExtendedEvent(int type, EventDataReader eventDataReader) {
      throw new RuntimeException();
   }
});
setupNativeEventType(type);
```

Extended event

Extended events are events with data that can not be stored on 24 bits.

```
+-------+----------+------------+
| 1 (1) | Type (7) | Length (24) |
+-------+----------+------------+
...
+------------------------------+
|       Extended Data  (32)    | (Length bytes)
+------------------------------+
```

On the first 32 bits of the events, you will have:

- First bit is equal to 1, saying that this is an extended event,
- The event type stored on 7 bits,
- The length of the data following the header in bytes stored on 24 bits.

Then you will have the data. The number of bytes of the data depends on the length.

Data Alignment

To process the data from an extended event, you will use an `EventDataReader` object. You will see it more in detail in the *Handle the event* section.

With `EventDataReader` API, there are two ways to read an event:

- Read the data with `read(byte[] b, int off, int len)` or `readFully(byte[] b)` methods.
  - You will get the data in a byte array and can process it on your own in your `handleExtendedEvent(int type, EventDataReader eventDataReader)` method.
- Read the data with the methods related to the primitive types such as `readBoolean()` or `readByte()`.
  - The reader is designed to parse C-struct data.
  - To use the methods, **your fields must follow this alignment:**
    - A **boolean** (1 byte) will be 1-byte aligned.
    - A **byte** (1 byte) will be 1-byte aligned.
    - A **char** (2 bytes) will be 2-byte aligned.
    - A **double** (8 bytes) will be 8-byte aligned.
    - A **float** (4 bytes) will be 4-byte aligned.
    - An **int** (4 bytes) will be 4-byte aligned.
    - A **long** (8 bytes) will be 8-byte aligned.
    - A **short** (2 bytes) will be 2-byte aligned.
    - An **unsigned byte** (1 byte) will be 1-byte aligned.
    - A **unsigned short** (2 bytes) will be 2-byte aligned.

Offer the event

There are two ways to send an extended event through the Event Queue: from the C API or the Java API.

From C API

To send an extended event through the Event Queue using the C API, you have to use the `LLEVENT_offerExtendedEvent(int32_t type, void* data, int32_t data_length)` method from `LLEVENT.h`.

For example:

```
struct accelerometer_data {
   int x;
   int y;
   int z;
}

// Assuming that event_type has␣
↪been passed from the Managed world through a native method after registering your listener.
```

(continues on next page)

(continued from previous page)

```
int type = event_type;

struct accelerometer_data data;
data.x = 42;
data.y = 72;
data.z = 21;

LLEVENT_offerExtendedEvent(type, (void*)&data, sizeof(data));
```

From Java API

To send an extended event through the Event Queue using the Java API, you must use the `offerExtendedEvent(int type, byte[] data)` method from the EventQueue API.

For example:

```
EventQueue eventQueue = EventQueue.getInstance();

// Assuming␣
↪that eventType has been stored in the Managed world when you registered the listener.
int type = eventType;

// Array of 3 integers. Each integer is stored in 4 bytes.
byte[] accelerometerData = new byte[3*4];

// Write integers into the byte array using ByteArray API.
ByteArray.writeInt(accelerometerData, 0, 42);
ByteArray.writeInt(accelerometerData, 4, 72);
ByteArray.writeInt(accelerometerData, 8, 21);

eventQueue.offerExtendedEvent(type, accelerometerData);
```

Handle the event

To handle an extended event, you must implement your listener's `handleExtendedEvent(int type, EventDataReader eventDataReader)` method. You can process the data received by the Event Queue on this method.

It provides an EventDataReader that contains the methods needed to read the data of an extended event.

First, you have to register your listener as explained *Event Queue listener* in section.

For example:

```
EventQueue queue = EventQueue.getInstance();
int type = queue.registerListener(type, new EventQueueListener() {
   @Override
   public void handleEvent(int type, int data) {
      throw new RuntimeException();
   }
   @Override
   public void handleExtendedEvent(int type, EventDataReader eventDataReader) {
      int x = 0;
```

(continues on next page)

(continued from previous page)

```
    int y = 0;
    int z = 0;
    try {
      x = eventDataReader.readInt();
      y = eventDataReader.readInt();
      z = eventDataReader.readInt();
    } catch (IOException e) {
      System.
↪out.println("IOException while reading accelerometer values from the EventDataReader.");
    }
  ␣
↪System.out.println("Accelerometer values: X = " + x + ", Y = " + y + ", Z = " + z + ".");
  }
});
setupNativeEventType(type);
```

Mock the Event Queue

To simulate event that are normally sent through the C API, use the Event Queue Mock API from your mock.

The Event Queue Mock API dependency must be added to the project build file of your MicroEJ Mock project.

SDK 6 (build.gradle.kts)

SDK 5 (module.ivy)

For Event Queue 3.0.1 and higher:

```
implementation("com.microej.pack.event:event-mock-api:3.0.1")
```

For Event Queue 2.2.0 and lower:

```
implementation(group=
↪"com.microej.pack.event", name="event-pack", version="2.2.0", configuration="mockAPI")
```

For Event Queue 2.3.0 and higher:

```
Not Supported
```

For Event Queue 2.2.0 and lower:

```
<dependency org=
↪"com.microej.pack.event" name="event-pack" rev="2.2.0" conf="provided->mockAPI"/>
```

It provides two methods:

- EventQueueMock.offerEvent(int type, int data) is the equivalent of LLEVENT_offerEvent(int32_t type, int32_t data) method from LLEVENT.h .
- EventQueueMock.offerExtendedEvent(int type, byte[] data, int dataLength) is the equivalent of LLEVENT_offerExtendedEvent(int32_t type, void* data, int32_t data_length) method from LLEVENT.h .

Example of use:

```
// Assuming that event_type has
↪been passed from your Application through a native method after registering your listener.
int type = event_type;
int data = 12;

EventQueueMock.offerEvent(type, data);
```

Use

The Event Queue API Module must be added to the project build file of the MicroEJ Application project to use the Event Queue Foundation Library.

SDK 6 (build.gradle.kts)

SDK 5 (module.ivy)

```
implementation("ej.api:event:3.0.1")
```

For Event Queue 2.3.0 and higher:

```
Not Supported
```

For Event Queue 2.1.0 and lower:

```
<dependency org="ej.api" name="event" rev="2.1.0"/>
```

To use this API, your VEE Port must implement a compatible version. Please refer to the *VEE Porting Guide* to port the Event Queue for your project.

Warning: The Event Queue library is not designed to be *Multi-Sandbox enabled*. Therefore, a Kernel must not expose its APIs to Features.

Configuration

The *Event Queue Pack* can be configured by defining the following properties as *Constants* in SDK 6 and as *Application Options* in SDK 5:

- event.thread.name : defines the name of the Event Queue thread. Its default value is EventQueue .
- event.thread.priority : defines the priority of the Event Queue thread. Its value must be between Thread.MIN_PRIORITY and Thread.MAX_PRIORITY, and its default value is Thread.NORM_PRIORITY.
- event.thread.daemon : defines whether the Event Queue thread is a daemon thread. Its default value is false .

GNSS

Principle

The GNSS Foundation Library.

Functional Description

Interactions with the GNSS API goes through a GNSS manager. This manager works around a state machine. To perform read operations to retrieve GNSS data, the machine must be in a STARTED state. Transitions between the states are controlled by the user. *The following graph*: details the different states, their function and transitions.

Fig. 167: GNSS State Machine.

After each operations, the method *getState()* can be called on the GNSS manager instance to check weither or not the state has been successfully changed.

Turning the GNSS Engine ON

```
// Retrieve the GNSS instance
GnssManager gnssManager = GnssManager.getInstance();

// Initialize the manager and native components
gnssManager.initializeManager();

// Switch the GNSS Engine ON
gnssManager.switchOn();
```

Retrieving GNSS data

```
// Read data
gnssManager.readPosition();

// Check timestamp of latest data read to confirm a fix was found.
long timestamp = gnssManager.getTime();

// Retrieve alitude data from engine
gnssManager.getAltitude()
```

Turning the GNSS Engine OFF

```
// Retrieve the GNSS instance
GnssManager gnssManager = GnssManager.getInstance();

// Switch the GNSS Engine OFF
gnssManager.switchOff();

// Close every native components and the manager.
gnssManager.closeManager();
```

Note: *readPosition()* call will throw an IOException if an error occured during data reading, however, timestamp will still be updated.

Use

The GNSS Library is provided as a Foundation Library.

To use the GNSS API Module, add the following line to the project build file:

```
implementation("ej.api:gnss:2.0.0")
```

Building or running an Application which uses the GNSS Library requires a SDK6 VEE Port that provides the *GNSS Pack*.

JavaScript

MicroEJ allows to develop parts of an application in JavaScript. Basically, a MicroEJ Application boots in Java code, then it initializes the JavaScript runtime to run a mix of Java code and JavaScript code.

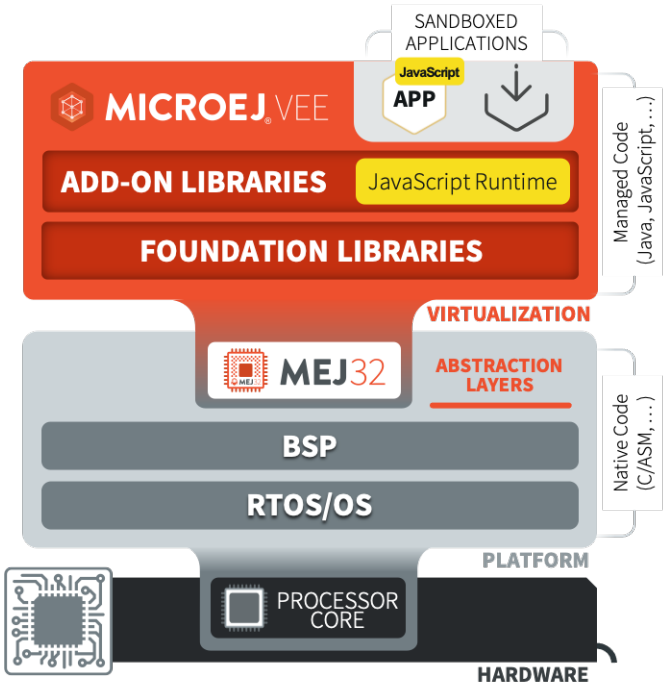


Fig. 168: MicroEJ JavaScript Overview

It supports the ECMAScript 5.1 specification, with *some limitations*. You can start playing with it by following the *Getting Started* page.

Getting Started

Let's walk through the steps required to use Javascript in your MicroEJ application:

- install the *MMM CLI (Command Line Interface)*
- create your Standalone Application project with the `init` command:

```
mmm init -Dskeleton.org=com.is2t.easyant.skeletons -Dskeleton.
↪module=firmware-singleapp -Dskeleton.rev=1.1.12 -Dproject.org=com.mycompany␣
↪-Dproject.module=myproject -Dproject.rev=1.0.0 -Dskeleton.target.dir=myproject
```

Adapt the properties values to your need. See the *MMM CLI init command documentation* for more details.

Javascript is supported in the following *Module Natures page*: - *Add-On Library*, - *Standalone Application*, - *Sandboxed Application*.

- add the js dependency in the build file:

  Gradle (build.gradle.kts)

  MMM (module.ivy)

```
implementation("com.microej.library.runtime:js:0.13.0")
```

```
<dependency org="com.microej.library.runtime" name="js" rev="0.13.0"/>
```

- add the following lines in your application main class:

```
import com.microej.js.JsErrorWrapper;
import com.microej.js.JsCode;
import com.microej.js.JsRuntime;

...

JsCode.initJs();
JsRuntime.ENGINE.runOneJob();
JsRuntime.stop();
```

- create a file named `hello.js` in the folder `src/main/js` with the following content:

```
function hello() {
    var message = "MicroEJ Javascript application!";
    print("My first", message);
}

hello()
```

- follow the steps described in the *run command documentation*
- in a terminal, go to the folder containing the `module.ivy` file and build the project with the command:

```
mmm build
```

You should see the following message at the end of the build:

```
BUILD SUCCESSFUL

Total time: 20 seconds
```

- now that your application is built, you can run it in the simulator with the command:

```
mmm run
```

You should see the following output:

```
My first  MicroEJ Javascript application!
```

You can now go further by exploring the *capabilities of the MicroEJ Javascript engine* and discovering the *commands available in the CLI*.

Sources Management

JavaScript Sources Location

The JavaScript sources of an application must be located in the project folder `src/main/js`. All JavaScript files (`*.js`) found in this folder, at any level, are processed.

JavaScript Sources Load Order

When several JavaScript files are found in the sources folder, they are loaded in alphabetical order of their relative path. For example, the following source files:

```
src
   main
      js
         components
            component1.js
            component2.js
         ui
            widgets.js
         app.js
         feature1.js
         feature2.js
```

are loaded in this order:

1. app.js
2. components/component1.js
3. components/component2.js
4. feature1.js
5. feature2.js
6. ui/widgets.js

JavaScript Sources Load Scope

All the code of the JavaScript source files are loaded in the same scope. It means a variable or function defined in a source file can be used in another one if it has been loaded first. In this example:

Listing 24: src/main/js/lib.js

```
function sum(a, b) {
   return a + b;
}
```

Listing 25: src/main/js/main.js

```
print("5 + 3 = " + sum(5, 3));
```

the file `src/main/js/lib.js` is loaded before `src/main/js/main.js` so the function `sum` can be used in `src/main/js/main.js`.

JavaScript Sources Processing

JavaScript sources need to be processed before being executed. This processing is done in the following cases:

- when building the project with *MMM*.
- when developing the project in MicroEJ SDK. The MicroEJ SDK detects any change in JavaScript sources folder (addition/update/deletion) to trigger the processing.

Examples

This section is intended to provide a set of examples to cover most of the use cases when developing JavaScript applications with MicroEJ:

Simple Application

Note: Before trying this example, make sure you have the *MMM CLI (Command Line Interface)* installed.

This example shows the minimal code for a MicroEJ JavaScript application:

- create an *Add-On Library* project or a *Sandboxed Application* project
- add the MicroEJ JavaScript dependency in the build file of your project:

  Gradle (build.gradle.kts)

  MMM (module.ivy)

```
implementation("com.microej.library.runtime:js:0.13.0")
```

```
<dependency org="com.microej.library.runtime" name="js" rev="0.13.0"/>
```

- init the JavaScript code in your Java application with:

```
import com.microej.js.JsCode;

...

JsCode.init();
```

The class `com.microej.js.JsCode` is the Java class generated from the JavaScript sources.

- ask the MicroEJ JavaScript engine to start processing the job queue with:

```
import com.microej.js.JsRuntime;

...

JsRuntime.ENGINE.run();
```

This makes the JavaScript engine process the job queue forever until the program is stopped.

- create a file with the `js` extension in the `src/main/js` folder (for example `app.js`) with the following content:

```
print("My Simple Application");
```

- build and execute the application with the *MMM CLI*:

```
$ mmm build
$ mmm run
```

The message `My Simple Application` should be displayed.

Use a Java API in JavaScript

Note: Before trying this example, make sure you have the *MMM CLI (Command Line Interface)* installed.

It is also recommended to follow the *Getting Started* page and/or the *Simple Application* example before.

In this example the JavaScript code calls a Java API. The Java API can come from the application or from any library used by the application. Let's create it in the project for this example, in a class `Calculator` (`src/main/java/com/mycompany/Calculator.java`):

```
public class Calculator {
    public int sum(int x, int y) {
        return x + y;
    }

    public int mul(int x, int y) {
        return x * y;
    }
}
```

Then in the Java Main class of the application, add the glue to expose the *Calculator* Java API to the JavaScript code and init the JavaScript engine:

```
public static void main(String[] args) throws Exception {
    // Add the "getCalculator" function in the JavaScript global object
    JsRuntime.
↪JS_GLOBAL_OBJECT.put("getCalculator", JsRuntime.createFunction(new JsClosure() {
        @Override
        @Nullable
        public Object invoke(Object thisBinding, int argsLength, Object... arguments) {
            return new Calculator();
        }
    }), false);

    // Init the JavaScript code
    JsCode.initJs();
    // Start the JavaScript engine
    JsRuntime.ENGINE.run();
}
```

You can now call the API from the JavaScript code:

```
var calc = getCalculator();
print(calc.sum(1, 2));
print(calc.mul(5, 3));
```

As you can see, the methods of the Java API *Calculator* can be used directly from the JavaScript code.

Finally, build and execute the application with the *MMM CLI*:

```
$ mmm build
$ mmm run
```

The sum and multiply results should be displayed.

For more information about communication between Java code and JavaScript code please refer to the *Communication Between Java code and JavaScript Code* page.

Create a JavaScript API from Java

Note: Before trying this example, make sure you have the *MMM CLI (Command Line Interface)* installed.

It is also recommended to follow the *Getting Started* page and/or the *Simple Application* example before.

In this example a JavaScript API is exposed from Java code. This can be useful when a specific API must be defined in JavaScript or when adapting an existing Java API to a JavaScript API.

Create a class `MyApiHostObject` (`src/main/java/com/mycompany/MyApiHostObject.java`):

```
public class MyApiHostObject extends JsObject {

    public MyApiHostObject(Object thisBinding) {

        this.put("count", new DataPropertyDescriptor(JsRuntime.createFunction(new JsClosure() {
                @Override
                @Nullable
                public Object invoke(@Nullable Object thisBinding, int argsLength, Object... arguments) {
                    String data = (String) arguments[0];
                    return Integer.valueOf(data.length());
                }
            })));

    }
}
```

This class defines a JavaScript object using the MicroEJ JavaScript API by extending the class `JsObject` . It also defines a `count` method which accepts a String parameter and returns its length.

Then in the Java Main class of the application, add the glue to expose the *MyApi* object to the JavaScript code and init the JavaScript engine:

```
public static void main(String[] args) throws Exception {
        // Add the "MyApi" function in the JavaScript global object
        JsRuntime.
↪JS_GLOBAL_OBJECT.put("MyApi", JsRuntime.createFunction(new JsClosure() {
                @Override
                @Nullable
                public Object invoke(Object thisBinding, int argsLength, Object... arguments) {
                        return new MyApiHostObject(thisBinding);
                }
        }), false);

        // Init the JavaScript code
        JsCode.initJs();
        // Start the JavaScript engine
        JsRuntime.ENGINE.run();
}
```

You can now call the new API from the JavaScript code:

```
var myApi = new MyApi();
print(myApi.count("Hello World!"));
```

Finally, build and execute the application with the *MMM CLI*:

```
$ mmm build
$ mmm run
```

The length of the string `Hello World!` (12) should be displayed.

For more information about communication between Java code and JavaScript code please refer to the *Communication Between Java code and JavaScript Code* page.

API

This page lists the API provided by the MicroEJ JavaScript engine.

Built-in Objects

The built-in objects are the API objects defined by the ECMAScript specification. This section lists all the JavaScript built-in objects and their support status in the MicroEJ JavaScript engine. For the complete reference about these built-in objects, consult the ECMAScript 5.1 specification.

For a description and usage examples of each method or property, consult a JavaScript documentation such as Mozilla Developer Reference.

Array

- Array (len)
- isArray (arg)
- toString ( )
- **[excluded]** toLocaleString ( )
- concat ([item1 [, item2 [, …]]])
- join (separator)
- pop ( )
- push ([item1 [, item2 [, …]]])
- reverse ( )
- shift ( )
- slice (start, end)
- sort (comparefn)
- **[excluded]** splice (start, deleteCount [, item1 [, item2 [, …]]])
- unshift ([item1 [, item2 [, …]]])
- indexOf (searchElement [, fromIndex])
- lastIndexOf (searchElement [, fromIndex])
- every (callbackfn [, thisArg])
- some (callbackfn [, thisArg])
- forEach (callbackfn [, thisArg])
- map (callbackfn [, thisArg])
- filter (callbackfn [, thisArg])
- **[excluded]** reduce (callbackfn [, initialValue])
- **[excluded]** reduceRight (callbackfn [, initialValue])
- length

Boolean

- Boolean (value)
- Boolean.prototype.toString ( )
- Boolean.prototype.valueOf ( )

Date

- **[excluded]**

Error

- **[excluded]**

Function

- **[excluded]** Function (p1, p2, … , pn, body)
- length
- **[excluded]** toString ( )
- apply (thisArg, argArray)
- call (thisArg [, arg1 [, arg2, …]])
- **[excluded]** bind (thisArg [, arg1 [, arg2, …]])
- [[Call]]
- [[Construct]]

Global

- NaN
- Infinity
- undefined
- **[excluded]** eval (x)
- parseInt (string , radix)
- parseFloat (string)
- isNaN (number)
- isFinite (number)
- **[excluded]** escape (string)
- **[excluded]** unescape (string)
- **[excluded]** decodeURI (encodedURI)
- **[excluded]** decodeURIComponent (encodedURIComponent)
- **[excluded]** encodeURI (uri)
- **[excluded]** encodeURIComponent (uriComponent)

JSON

- parse (text [, reviver])
- stringify (value , [replacer [, space]])

Math

- E
- LN10
- LN2
- LOG2E
- LOG10E
- PI
- SQRT1_2
- SQRT2
- abs (x)
- acos (x)
- asin (x)
- atan (x)
- atan2 (y, x)
- ceil (x)
- cos (x)
- exp (x)
- floor (x)
- log (x)
- max ([value1 [, value2 [, …]]])
- min ([value1 [, value2 [, …]]])
- pow (x, y)
- random ( )
- round (x)
- sin (x)
- sqrt (x)
- tan (x)

Number

- Number (value)
- MAX_VALUE
- MIN_VALUE
- NaN
- NEGATIVE_INFINITY
- POSITIVE_INFINITY
- **[excluded]** toString ([radix])
- **[excluded]** toLocaleString()
- valueOf ( )
- **[excluded]** toFixed (fractionDigits)
- **[excluded]** toExponential (fractionDigits)
- **[excluded]** toPrecision (precision)

Object

- Object ([value])
- Object.getPrototypeOf (O)
- Object.getOwnPropertyDescriptor (O, P)
- Object.getOwnPropertyNames (O)
- Object.create (O [, Properties])
- Object.defineProperty (O, P, Attributes)
- Object.defineProperties (O, Properties)
- **[excluded]** Object.seal (O)
- **[excluded]** Object.freeze (O)
- **[excluded]** Object.preventExtensions (O)
- Object.isSealed (O)
- Object.isFrozen (O)
- Object.isExtensible (O)
- Object.keys (O)
- toString ( )
- **[excluded]** toLocaleString ( )
- valueOf ( )
- hasOwnProperty (V)
- isPrototypeOf (V)
- propertyIsEnumerable (V)

Regex

- RegExp (pattern, flags)
- exec (string)
- test (string)
- toString ()

String

- String (value)
- fromCharCode ([char0 [, char1 [, …]]])
- toString ( )
- valueOf ( )
- charAt (pos)
- charCodeAt (pos)
- concat ([string1 [, string2 [, …]]])
- indexOf (searchString, position)
- lastIndexOf (searchString, position)
- **[excluded]** localeCompare (that)
- match (regexp)
- replace (searchValue, replaceValue)
- **[excluded]** search (regexp)
- slice (start, end)
- split (separator, limit)
- **[excluded]** substr (start [, length])
- substring (start, end)
- toLowerCase ( )
- **[excluded]** toLocaleLowerCase ( )
- toUpperCase ( )
- **[excluded]** toLocaleUpperCase ( )
- trim ( )
- length
- [[GetOwnProperty]] (P)

Host Objects

Host objects are not part of the ECMAScript specification, they are additional API provided by the MicroEJ JavaScript engine.

Global

setTimeout(function[, delay, arg1, arg2, ...])

- **description:** sets a timer which executes a function once the timer expires.
- **arguments:**

– function : the function to execute when the delay expires.

– delay (optional): the time in milliseconds that the timer must wait before executing the given function.

– arg1, arg2, ... (optional): additional arguments passed to the given function.

- **returns:** the timer object. This object can be passed to the function clearTimeout to cancel the timer.

setInterval(function[, delay, arg1, arg2, ...])

- **description:** repeatedly calls a function, with a fixed time delay between each call.
- **arguments:**

– function : the function to execute when the delay expires.

– delay (optional): the time in milliseconds that the timer must wait between each execution of the given function.

– arg1, arg2, ... (optional): additional arguments passed to the given function.

- **returns:** the timer object. This object can be passed to the function clearInterval to cancel the timer.

clearTimeout(timer)

- **description:** cancels the given timer created by a call to setTimeout .
- **arguments:**

– timer : the timer to cancel.

clearInterval(timer)

- **description:** cancels the given timer created by a call to setInterval .
- **arguments:**

– timer : the timer to cancel.

print([arg1, arg2, ...])

- **description:** prints the given arguments in the standard output. The arguments are concatenated and separated by a space. A new line is added at the end.
- **arguments:**

– arg1, arg2, ... : the list of elements to print.

Communication Between Java code and JavaScript Code

The MicroEJ engine allows to communicate between Java code and JavaScript code: Java API can be used from JavaScript code and vice-versa.

JavaScript Engine

The JavaScript code is executed in a single-threaded engine, which means only one JavaScript statement is executed at a given time. Each piece of JavaScript code that must be executed is pushed in a job queue. It is up to the engine to manage the job queue and execute the jobs.

One consequence of this design is that Java code called from a JavaScript code must not be blocker. When calling a Java API from a Javascript code, in order to avoid blocking the JavaScript engine, the Java code must return as quick as possible. Otherwise the JavaScript engine is stuck and cannot execute other JavaScript jobs. It is especially harmfull when the Java operation takes time, for example for network or IO operations. In such a case, it is therefore recommended to execute it in a new thread and return immediately.

Another consequence of the JavaScript engine design is that JavaScript code must always be executed by the engine, by the single thread. Therefore, any call to a JavaScript code from a Java code must create a job and add it to the job queue.

Calling Java from JavaScript

The MicroEJ engine allows to expose Java objects or methods to the JavaScript code by using the engine API and creating the adequate JavaScript object.

Import Java Types from JavaScript

Java objects can be exposed to JavaScript using the JavaImport mechanism. It takes a Java fully qualified name as argument and returns an object that gives access to the constructors, static methods and static fields. All the classes from the project's classpath can be imported (project's own classes and its dependencies).

For instance, the following code imports java.lang.System and prints a string calling System.out.println():

```
var System = JavaImport("java.lang.System")
System.out.println("foo");
```

Here we instantiate a Java File object and check that it exists:

```
var File = JavaImport("java.io.File")
var myFile = new File("myFile.txt")

if (myFile.exists()) {
	print("myFile.txt exists")
} else {
	print("myFile.txt does not exist")
}
```

> **Warning:** You cannot instantiate an anonymous class from an interface or an abstract class with the new keyword and JavaImport . Nevertheless, you can still access to static fields and methods.

Implement JavaScript Functions in Java Code

We can also implement JavaScript functions in Java code by adding their implementation to the global object from Java code. For example, here is the code to create a JavaScript function named javaPrint in the global scope:

```
JsRuntime.
↪JS_GLOBAL_OBJECT.put("javaPrint", JsRuntime.createFunction(new JsClosure() {
		@Override
		public Object invoke(Object thisBinding, Object... arguments) {
			System.out.println("Print from Java code: " + arguments[0]);
			return null;
		}
	}), false);
```

The function is created with a com.microej.js.objects.JsObjectFunction object created with the API JsRuntime.createFunction(JsClosure jsClosure) , and injected in the object JsRuntime.JS_GLOBAL_OBJECT which maps to the JavaScript global scope.

The function javaPrint can then be used in JS:

```
javaPrint("foo")
```

This technique can also be used to share any Java object to JavaScript. It is achieved by returning the Java object in the invoke method of the JsClosure object. For example, a Java Date object can be exposed as follows:

```
JsRuntime.JS_
↪GLOBAL_OBJECT.put("getCurrentDate", JsRuntime.createFunction(new JsClosure() {
        @Override
        public Object invoke(Object thisBinding, Object... arguments) {
                return Calendar.getInstance().getTime();
        }
}), false);
```

When a Java object is exposed in JavaScript, all its public methods can be called, therefore the JavaScript code can then use this Date object and get the time:

```
var date = getCurrentDate()
var time = date.getTime()
print("Current time: ", time)
```

for more information on how these called are managed by the MicroEJ JavaScript engine, please go to the *Foreign Function Interface* section.

Java objects can also be shared using one of the other Java JS adapter objects. With this solution, the code of the Java object is executed at engine initialisation, contrary to the previous solution where it is executed only when the JavaScript code is called. For example, here is the code to expose a Java string named `javaString` in the JavaScript global scope:

```
JsRuntime.JS_GLOBAL_OBJECT.put("javaString", "Hello World!", false);
```

The string `javaString` can then be used in JS:

```
var myString = javaString;
```

The available Java JS adapter objects are:

- `com.microej.js.objects.JsObject` : exposes a Java object as a JavaScript object
- `com.microej.js.objects.JsObjectFunction` : exposes a Java "process" as a JavaScript function (using a JsClosure object)
- `com.microej.js.objects.JsObjectString` : exposes a Java String as a JavaScript String
- `com.microej.js.objects.JsObjectArray` : exposes a Java items collection as a JavaScript Array
- `com.microej.js.objects.JsObjectBoolean` : exposes a Java Boolean as a JavaScript Boolean
- `com.microej.js.objects.JsObjectNumber` : exposes a Java Number as a JavaScript Number

Calling JavaScript from Java

The MicroEJ JavaScript engine API allows to call JavaScript code from Java code. For example, given the following JavaScript function in a file in `src/main/js` :

```
function sum(a, b) {
    print(a + " + " + b + " = " + (a+b));
}
```

it can be called from a Java piece of code with:

```
JsObjectFunction␣
↪functionObject = (JsObjectFunction) JsRuntime.JS_GLOBAL_OBJECT.get("sum");
```

(continues on next page)

(continued from previous page)

```
JsRuntime.ENGINE.addJob(functionObject,
↪ JsRuntime.JS_GLOBAL_OBJECT, new Integer(5), new Integer(3));
```

The first line gets the JavaScript function from the global scope. The second line adds a job in the JavaScript engine queue to execute the function, in the global scope, with the arguments `5` and `3`.

Passing Values Between JavaScript and Java

JavaScript base types are represented by Java objects and not Java base types. The following table shows the mapping between types in both languages:

JavaScript	Java
Number	java.lang.Integer or java.lang.Double
Boolean	java.lang.Boolean
String	java.lang.String
Null	`null` value
Undefined	`JsRuntime.JS_UNDEFINED_OBJECT` singleton

In JavaScript, a `Number` type is a 64-bits floating-point value. Nevertheless, Kifaru may use integer values (Integer Java type) when possible for performance reasons. Otherwise, Double type will be used.

Note: Prefer passing Integer values as argument to a job added to the JavaScript execution queue, or return `Integer` values when implementing a `JsClosure` instead of Double when possible.

It is not possible to retrieve the returned value of a JavaScript function from Java code. For instance, consider the following JavaScript function:

```
function sum(a, b) {
    return a + b;
}
```

When calling this function from Java code, we have no way to get the result back:

```
JsObjectFunction␣
↪functionObject = (JsObjectFunction) JsRuntime.JS_GLOBAL_OBJECT.get("sum");
JsRuntime.ENGINE.addJob(functionObject,
↪ JsRuntime.JS_GLOBAL_OBJECT, new Integer(5), new Integer(3));
```

A workaround is to modify the JavaScript function so it takes a callback object as argument:

```
function sum(a, b, callback) {
    callback.returnValue(a + b);
}
```

Here is a possible implementation of the callback object:

```
public class Callback<T> {

    @Nullable
    private T value;

    private boolean returned;

    /**
     * Gets the value returned by this callback function when ready.
     * <p>
     * A call to this method waits for the value to be ready.
     *
     * @return the value return by the callback
     */
        @Nullable
    public T getValue() {
        synchronized (this) {
            while (!this.returned) {
                try {
                    wait();
                } catch (InterruptedException e) {
                    throw new JsErrorWrapper(""); //$NON-NLS-1$
                }
            }
        }

         return this.value;
    }

    /**
     * Sets the value to return by this callback function.
     *
     * @param value
     *            the value to return
     */
    public synchronized void returnValue(@Nullable T value) {
        this.value = value;
        this.returned = true;
        notify();
    }
}
```

We can now pass the callback to the job. The Java code will wait on the `callback.getValue()` until the result is ready.

```
JsObjectFunction␣
↪functionObject = (JsObjectFunction) JsRuntime.JS_GLOBAL_OBJECT.get("sum");
Callback<Integer> callback = new Callback<>();
JsRuntime.ENGINE.addJob(functionObject,
↪ JsRuntime.JS_GLOBAL_OBJECT, new Integer(5), new Integer(3), callback);
Integer returnedValue = callback.getValue();
System.out.println("Result is " + returnedValue);
```

Tests

JavaScript applications can be tested with tests written in JavaScript. The JavaScript test files must be located in the project folder `src/test/js`. All JavaScript files (`*.js`) found in this folder, at any level, are considered as test files.

In order to setup JavaScript tests for your application, follow these steps:

- create an *Add-On Library* project or a *Standalone Application* project
- define the following configuration in the build file of the project:

  Gradle (build.gradle.kts)

  MMM (module.ivy)

```
tasks.test {
   filter {
      includeTestsMatching("*._JsTest_*Code")
   }
}
```

Add these properties nside the `ea:build` tag (if the properties already exist, replace them):

```
<ea:property name="test.run.includes.pattern" value="**/_JsTest_*Code.class"/>
<ea:property name="target.main.classes" value="${basedir}/target~/test/classes"/>
```

- add the MicroEJ JavaScript dependency in the build file of the project:

  Gradle (build.gradle.kts)

  MMM (module.ivy)

```
implementation("com.microej.library.runtime:js:0.13.0")
```

```
<dependency org="com.microej.library.runtime" name="js" rev="0.13.0"/>
```

- define the platform to use to run the tests with one of the options described in *Select a VEE Port* section
- create a file `assert.js` in the folder `src/test/resources` with the following content:

```
var assertionCount = 0;

function assert(value) {
        assertionCount++;
        if (value == 0) {
                print("assert " + assertionCount + " - FAILED");
        } else {
                print("assert " + assertionCount + " - PASSED");
        }
}
```

This method `assert` will be available in all tests to do assertions.

- create a file `test.js` in the folder `src/test/js` and write your first test:

```
var a = 5;
var b = 3;
```

(continues on next page)

(continued from previous page)

```
var sum = a + b;
assert(sum === 8);
```

- build the application in the SDK or in command line with the *MMM CLI*

  The execution of the tests produces a report available in the folder `target~/test/html` for the project.

Limitations

The MicroEJ engine supports the version 5.1 of the ECMAScript specification, with the limitations described in this page.

Unsupported Directives

Directives, such as `'use strict'`, are not supported and are considered as literal statements. Literal statements are just ignored.

Unsupported Statements

The following syntaxes are not supported by the MicroEJ JavaScript engine:

- `with (x) { }` : the `with` statement is not supported in MicroEJ since its usage is not recommended. See the reference documentation for more information.

Unsupported Built-in Objects

The unsupported built-in objects are listed in the *API section*.

Troubleshooting

Compilation error `cannot be resolved to a type` in FFI class

A compilation error can be raised when the classpath contains unexpected classes, for example:

```
Exception in thread "main" java.lang.Error: Unresolved compilation problems:
        ArrayComparisonFailure cannot be resolved to a type
        ArrayComparisonFailure cannot be resolved to a type

        at java.lang.Throwable.fillInStackTrace(Throwable.java:82)
        at java.lang.Throwable.<init>(Throwable.java:37)
        at java.lang.Error.<init>(Error.java:18)
        at com.microej.js.JsFfi.ffi_toString_0(JsFfi.java:54)
        at com.microej.js.JsCode$1$1.invoke(JsCode.java:50)
```

As described in *the FFI section*, in order to call Java methods from JavaScript code, all the methods with the given names are searched in the classpath. Since the classpath can contain test dependencies which are not available at compile time, the generated FFI can contain classes

from these dependencies and therefore fail to compile. The following classes are excluded by default:

- ej.junit.*
- org.junit.*
- junit.*
- org.hamcrest.*
- java.lang.String
- java.lang.Number

This list can be changed by setting the system property js.ffi.excludes.classes to a comma-separated list of FQN patterns. For example:

```
js.ffi.excludes.classes=ej.junit.
↪*,org.junit.*,junit.*,org.hamcrest.*,java.lang.String,java.lang.Number,com.mycompany.test.*
```

> **Warning:** Defining this property overwrites the default value, so do not forget to keep the default excluded classes (unless you know what you are doing).

Internals

JavaScript Sources Processing

The JavaScript code is not executed directly, it is first translated in Java code and compiled with the Java application code. This transpilation is done by the JavaScript Add-On Processor. This processor uses the OpenJDK Nashorn library (extracted from *jre1.8.0_92*) to parse the Javascript files.

The operations performed by this processor are summarized in this diagram:

- **Parsing**: all JavaScript source files located in the folder src/main/js and src/test/js are parsed by the Nashorn library to provide a JavaScript AST.
- **JS Validation**: validation on the JavaScript AST to detect unsupported language features (for example eval).
- **Conversion preparation**: before actually converting the JavaScript AST to a Java AST, a preparation operation is done to initialize all the lexical environments (done by JsIrVisitor).
- **Conversion**: conversion of the JavaScript AST to a Java AST.
- **Java AST cleanup/optim**: post-conversion step to cleanup and optimize the Java AST. The following operations are done: - fix imports - remove dead code - remove literal statements
- **Java sources generation**: generation of the Java sources from the Java AST.

Foreign Function Interface

As said in the section *Calling Java from JavaScript*, a JavaScript code can manipulate Java objects and call methods on Java objects. This chapter describes how does the call to methods on Java objets work.

Let `getValue()` a Java method called from JavaScript on a Java object. As long as the type of the object is not known at compile-time in the JavaScript code, all the types containing a method with the same signature are searched in the classpath. Then the JavaScript pre-processor generates a `JsFfi` class and a method that dynamically tries to find the type of the receiver object. So, when the `getValue()` method is called from JavaScript, this generated method is called.

Warning: Calling a method whose name is very common could result in a delay while calling it, and some useless methods embedded.

This example shares a Java Date of the current time:

```
JsRuntime.JS_
↪GLOBAL_OBJECT.put("getCurrentDate", JsRuntime.createFunction(new JsClosure() {
	@Override
	public Object invoke(Object thisBinding, Object... arguments) {
		return Calendar.getInstance().getTime();
	}
}), false);
```

The JavaScript can then use this Date to print the current time:

```
var date = getCurrentDate()
var time = date.getTime()
print("Current time: ", time)
```

In this case, the generated method in *JsFfi* looks like:

```
public static Object ffi_getTime_0(Object function, @ej.annotation.Nullable Object this_) {
	try {
		if (this_ instanceof JsObject || this_ instanceof String)
			return JsRuntime.functionCall(((Reference) function).getValue(), this_);
		if (this_ instanceof Calendar) {
			return ((Calendar) this_).getTime();
		}
		if (this_ instanceof Date) {
			return new Double(((Date) this_).getTime());
		}
	} catch (JsErrorWrapper e) {
		throw e;
	} catch (Throwable t) {
		throw new JsErrorWrapper(new JsObjectError.
↪TypeError("A Java exception has been thrown in generated FFI code of getTime"), t);
	}
	throw new JsErrorWrapper(new JsObjectError.TypeError("getTime"));
}
```

Storage

Introduction

The Storage interface provides a simple API to store, retrieve, and manage data entries using identifiers. The data entries can be seen as key/value pairs, with String keys and arbitrary serialized data as value (including binary).

This abstraction is useful to adapt to the target environment and to the different storage requirements. The ej.library.runtime#storage library comes with a default implementation to store the data in RAM (Application Managed Heap) which is not persistent.

We also provide the ej.library.runtime#storage-fs library which stores the data entries as files in a directory on the filesystem provided by *File System*.

It is especially designed to work in a Multi-Sandbox context where each Application is given access to a sandboxed storage, or can access a storage shared across the different Applications. Each storage being backed by a folder on a filesystem which could be created/removed on app installed/uninstalled for example. See more about this in *Multi-Sandbox Specificities*.

Usage

The Storage library offers two implementations:

- Storage in RAM:
  - Data is stored in RAM, in the *Application Managed Heap* (no file system).
  - Add the following line to the project build file:

```
implementation("ej.library.runtime:storage:1.2.0")
```

- Storage in a memory with a file system:
  - Data is stored in a file system.
  - Requires the *File System* Pack to be available in the VEE Port.
  - Add the following line to the project build file:

```
implementation("ej.library.runtime:storage-fs:1.2.0")
```

Initialization

To create a Storage instance:

- **RAM (Managed Heap) Implementation:**

```
Storage myStorage = new StorageHeap();
```

- **FileSystem Implementation:**

```
Storage myStorage = new StorageFs("my_root_folder");
```

Multi-Sandbox Specificities

Typically, when working in a multi-sandbox environment, the Applications are not given direct access to the File System API. Instead, one or multiple StorageFS instances are created by the Kernel and shared with the Applications (storage sandboxing).

Such a Storage instance can be shared using *Shared Services*, to make it available to Sandboxed Applications and other Kernel components:

- In the Kernel, register the Storage instance to be shared among all the Applications:

```
final ServiceRegistryKF␣
↪serviceRegistry = (ServiceRegistryKF) ServiceFactory.getServiceRegistry();
serviceRegistry.register(Storage.class, new StorageFs("/shared/"), true);
```

- In an Application, retrieve this instance using:

```
Storage storage = ServiceFactory.getService(Storage.class);
// use␣
↪this instance to access the Storage entries in the /shared/ folder on the device file system.
```

The Kernel-GREEN demonstrates how to implement storage sandboxing on a file system. In this architecture, the Kernel and Sandboxed Applications store their data in separate directories, as illustrated below:

```
.
  apps
    myApp1/
      ...
    myApp2/
      ...
  kernel
    ...
```

To achieve storage sandboxing:

- Define a custom Storage implementation that ensures sandboxing: SandboxedStorage.java,
- Bind the ej.storage.Storage type to the actual service implementation type that will be used for instantiation: com.microej.kernel.green.storage.SandboxedStorage . This binding is done using *System Properties* in kernel.properties.list.

Note: A SandboxedStorage instance will be created each time the service is retrieved in a new context (i.e., from different applications, the kernel). Refer to *Get a Service* for more information.

- Define SandboxedStorage as a required type in kernel.types.list.

APIs

The Storage interface defines operations for managing entries with unique identifiers.

An identifier must be a valid string composed of letters, digits, underscores, hyphens, or dots, with the first character being a letter.

Warning: The Storage in RAM (Managed Heap) implementation is not thread-safe.

Key Operations

- store(id) : Stores data for a given identifier.
- append(id) : Appends data to an existing entry.
- exists(id) : Checks if an entry exists.
- getSize(id) : Retrieves the size of an entry in bytes.
- load(id) : Loads data from a given identifier.
- modify(id, offset) : Modifies data at a specific offset within an entry.
- move(id1, id2) : Moves an entry from one identifier to another.
- getIds() : Retrieves all stored identifiers.
- remove(id) : Removes an entry with the specified identifier.

Code Example

```
Storage storage = ServiceFactory.getService(Storage.class);
String id1 = "example.data";
String id2 = "example2.data";
String id3 = "example3.data";

// 1. Store data
try (OutputStream os = storage.store(id1)) {
    os.write("Hello, MicroEJ Storage!".getBytes());
}

// 2. Append data
try (OutputStream os = storage.append(id1)) {
    os.write(" This is appended data.".getBytes());
}

// 3. Check if an entry exists
boolean exists = storage.exists(id1);

// 4. Get the size of an entry
long size = storage.getSize(id1);

// 5. Load data
try (InputStream is = storage.load(id1)) {
    byte[] buffer = new byte[1024];
```

(continues on next page)

(continued from previous page)

```
    int bytesRead;
    while ((bytesRead = is.read(buffer)) != -1) {
        System.out.print(new String(buffer, 0, bytesRead));
    }
}

// 6. Modify data at a specific offset
try (OutputStream os = storage.modify(id1, 7)) {
    os.write("Modified".getBytes());
}

// 7. Move an entry to a new ID
storage.move(id1, id2);
boolean exists1 = storage.exists(id1);
boolean exists2 = storage.exists(id2);

// 8. Get all stored IDs
String[] ids = storage.getIds();
for (String id : ids) {
    System.out.println(id);
}

// 9. Remove an entry
storage.remove(id2);
boolean existsRemoved = storage.exists(id2);

// 10. Store and remove another entry
try (OutputStream os = storage.store(id3)) {
    os.write("Temporary data".getBytes());
}
storage.remove(id3);
```

3.5.14 Development Tools

Code Instrumentation for Logging

This document explains how to add logging and tracing to MicroEJ applications and libraries with three different solutions. The aim is to help developers to report precise execution context for further debugging and monitoring.

Introduction

One straightforward way to add logs in Java code is to use the Java basic print methods: System.out.println(…).

However, this is not desirable when writing production-grade code, where it should be possible to adjust the log level:

- without having to change the original source code,
- at build-time or at runtime, as application logging will affect memory footprint and performances

Overview

This documentation describes 3 ways for logging data:

- Using Trace library: a real-time event recording library designed for performance and interaction analysis.
- Using Message library: a lightweight and simple logging library.
- Using Logging library: a complete and highly configurable standard logging library.

The use of each library will be illustrated by instrumenting the following code snippet:

```
public class Main {

   enum ApplicationState {
      INSTALLED, STARTED, STOPPED, UNINSTALLED
   }

   private static ApplicationState currentState;
   private static ApplicationState previousState;

   public static void main(String[] args) {
      currentState = ApplicationState.UNINSTALLED;
      switchState(ApplicationState.INSTALLED);
   }

   public static void switchState(ApplicationState newState) {
      previousState = currentState;
      currentState = newState;
   }
}
```

Finally, the last section describes some techniques to remove logging related code in order to reduce the memory footprint.

Log with the Trace Library

The ej.api.trace Trace library provides a way of tracing integer events. Its features and principles are described in the *Event Tracing* section.

Here is a short example of how to use this library to log the entry/exit of the switchState() method:

1. To use this library, add the following line to the project build file:

   Gradle (build.gradle.kts)

   MMM (module.ivy)

```
implementation("ej.api:trace:1.1.1")
```

```
<dependency org="ej.api" name="trace" rev="1.1.1"/>
```

2. Start by initializing a Tracer object:

```
private static final Tracer tracer = new Tracer("Application", 100);
```

In this case, Application identifies a category of events that defines a maximum of 100 different event types.

3. Next, start trace recording:

```
public static void main(String[] args) {
   Tracer.startTrace();

   currentState = ApplicationState.UNINSTALLED;
   switchState(ApplicationState.INSTALLED);
}
```

4. Use the methods Tracer.recordEvent(...) and Tracer.recordEventEnd(...) to record the entry/exit events in the method:

```
private static final int EVENT_ID = 0;

public static void switchState(ApplicationState newState) {
   tracer.recordEvent(EVENT_ID);

   previousState = currentState;
   currentState = newState;

   tracer.recordEventEnd(EVENT_ID);
}
```

The Tracer object records the entry/exit of method switchState with event ID 0.

5. Finally, to enable the Core Engine trace system, set the core.trace.enabled *option* to true.

This produces the following output:

```
[TRACE: Application] Event 0x0()
[TRACE: Application] Event End 0x0()
```

Note: The default VEE Port implementation of the Trace library prints the events to the console. See *VEE Port Implementation* for other available implementations such as *SystemView* tool.

Log with the Message Library

The ej.library.runtime.message Message library was designed to enable logging while minimizing RAM/ROM footprint and CPU usage. For that, logs are based on message identifiers, which are stored on integers instead of using of constant Strings. In addition to a message identifier, the category of the message allows the user to find the corresponding error/warning/info description. An external documentation must be maintained to describe all message identifiers and their expected arguments for each category.

Principles:

- The MessageLogger type allows for logging messages solely based on integers that identify the message content.
- Log a message by using methods MessageLogger.log(...) methods, by specifying the log level, the message category, and the message identifier. Use optional arguments to add any useful information to the log, such as a Throwable or contextual data.

- Log levels are very similar to those of the Logging library. The class ej.util.message.Level lists the available levels.
- Loggers rely on the MessageBuilder type for message creation. The messages built by the BasicMessageBuilder follow this pattern: [category]:[LEVEL]=[id] . The builder appends the specified Object arguments (if any) separated by spaces, then the full stack trace of the Throwable argument (if any).
- The FilterMessageLogger allows to filter messages actually logged based on a threshold level (defaults to INFO). The threshold level can be modified dynamically with FilterMessageLogger.setLevel(). Use the system FilterMessageLogger.INSTANCE or create a new logger to configure the level of logged messages per instance.

  Here is a short example of how to use this library to log the entry/exit of the switchState() method:

1. To use this library, add the following line to the project build file:

   Gradle (build.gradle.kts)

   MMM (module.ivy)

```
implementation("ej.library.runtime:message:2.2.1")
```

```
<dependency org="ej.library.runtime" name="message" rev="2.2.1"/>
```

2. Call the message API to log some info:

```
private static final String LOG_CATEGORY = "Application";

private static final int LOG_ID = 2;

public static void switchState(ApplicationState newState) {
   previousState = currentState;
   currentState = newState;

   FilterMessageLogger.
↪INSTANCE.log(Level.INFO, LOG_CATEGORY, LOG_ID, previousState, currentState);
}
```

   This produces the following output:

```
Application:I=2 UNINSTALLED INSTALLED
```

Log with the Logging Library

The ej.library.eclasspath.logging Logging library implements a subset of the standard Java java.util.logging package and follows the same principles:

- There is one instance of LogManager by application that manages the hierarchy of loggers.
- Find or create Logger objects using the method Logger.getLogger(String). If a logger has already been created with the same name, this logger is returned, otherwise a new logger is created.
- Each Logger created with this method is registered in the LogManager and can be retrieved using its String ID .
- A minimum level can be set to a Logger so that only messages that have at least this level are logged. The class java.util.logging.Level lists the available standard levels.

- The Logger API provides multiple methods for logging:
  - log(...) methods that send a LogRecord to the registered Handler instances. The LogRecord object wraps the String message and the log level.
  - Log level-specific methods, like severe(String msg), that call the aforementioned log(...) method with the correct level.
- The library defines a default Handler implementation, called DefaultHandler, that prints the message of the LogRecord on the standard error output stream. It also prints the stack trace of the Throwable associated with the LogRecord if there is one.

  Here is a short example of how to use this library to log the entry/exit of the switchState() method:

1. To use this library, add the following line to the project build file:

   Gradle (build.gradle.kts)

   MMM (module.ivy)

```
implementation("ej.library.eclasspath:logging:1.2.1")
```

```
<dependency org="ej.library.eclasspath" name="logging" rev="1.2.1"/>
```

2. Call the logging API to log some info text:

```
public static void switchState(ApplicationState newState) {
   previousState = currentState;
   currentState = newState;

   Logger logger = Logger.getLogger(Main.class.getName());
   logger.log(Level.
↪INFO, "The application state has changed from " + previousState.toString() + " to "
        + currentState.toString() + ".");
}
```

   This produces the following output:

```
main INFO: The application state has changed from UNINSTALLED to INSTALLED.
```

Note: Unlike the two other libraries discussed here, the Logging library is entirely based on Strings (log IDs and messages). String operations can lead to performance issues and String objects use significant ROM space. When possible, prefer using a logging solution that uses primitive types over Strings.

Remove Logging Related Code

This section describes some techniques to remove logging related code, which saves memory footprint when logging is disabled at runtime. This is typically useful when building two Firmware flavors: one for production and one for debug.

Wrap with a Constant If Statement

A boolean *constant* declared in an `if` statement can be used to fully remove portions of code. When this boolean constant is detected to be `false`, the wrapped code becomes unreachable and is not embedded.

Note: More information about the usage of constants and `if` code removal can be found in the *Classpath* section.

1. Let's consider a constant `com.mycompany.logging` declared as `false` in a resource file named `example.constants.list`.
2. Wrap the log code by an `if` statement, as follows:

```
private static final String LOG_PROPERTY = "com.mycompany.logging";

public static void switchState(ApplicationState newState) {
   previousState = currentState;
   currentState = newState;

   if (Constants.getBoolean(LOG_PROPERTY)) {
      Logger logger = Logger.getLogger(Main.class.getName());
      logger.log(Level.
↪INFO, "The application state has changed from " + previousState.toString() + " to "
         + currentState.toString() + ".");
   }
}
```

When using the Trace API (`Trace`), you can use the Tracer.TRACE_ENABLED_CONSTANT_PROPERTY constant that represents the value of the `core.trace.enabled` *option*.

Follow the same principle as before:

```
private static final int EVENT_ID = 0;

public static void switchState(ApplicationState newState) {
   if (Constants.getBoolean(Tracer.TRACE_ENABLED_CONSTANT_PROPERTY)) {
      tracer.recordEvent(EVENT_ID);
   }

   previousState = currentState;
   currentState = newState;

   if (Constants.getBoolean(Tracer.TRACE_ENABLED_CONSTANT_PROPERTY)) {
      tracer.recordEventEnd(EVENT_ID);
   }
}
```

Shrink Code Using ProGuard

ProGuard is a tool that shrinks, optimizes, and obfuscates Java code.

It optimizes bytecode as well as it detects and removes unused instructions. Therefore it can be used to remove log messages in a production binary.

A dedicated How-To is available at https://github.com/MicroEJ/How-To/tree/master/Proguard-Get-Started. It describes how to configure ProGuard to remove elements of code from the Logging library.

Event Tracing

Description

Event Tracing allows to record integer based events for debugging and monitoring purposes without affecting execution performance too heavily. Basically, it gives access to Tracer objects that are named and can produce a limited number of different event types.

A record is an event type identified by an eventID and can have a list of values. It can be a single event or a period of time with a start and an end.

Event Tracing can be accessed from two APIs:

- A Java API, provided by the Trace API module. The following dependency must be added to the build file of the MicroEJ Application project:

  Gradle (build.gradle.kts)

  MMM (module.ivy)

```
implementation("ej.api:trace:1.1.0")
```

```
<dependency org="ej.api" name="trace" rev="1.1.0"/>
```

- A C API, provided by the Foundation Library header file named LLTRACE.h .

Event Recording

Events are recorded if and only if:

- the Core Engine trace system is enabled,
- and trace recording is started.

To enable the Core Engine trace system, set the *Application Option* named core.trace.enabled to true (see also *launch configuration*). The Trace system requires at last 16 bytes of Immortal heap (see *Immortals Heap size*).

Then, multiple ways are available to start and stop the trace recording:

- by setting the *Application Option* named core.trace.autostart to true to automatically start at startup (see also *launch configuration*),
- using the Java API methods ej.trace.Tracer.startTrace() and ej.trace.Tracer.stopTrace(),
- using the C API functions LLTRACE_start(void) and LLTRACE_stop(void) .

Java API Usage

The detailed Trace API documentation is available here.

First, you need to instantiate a `Tracer` object by calling its constructor with two parameters. The first parameter, `name`, is a String that will represent the `Tracer` object group's name. The second parameter, `nbEventTypes`, is an integer representing the maximum number of event types available for the group.

```
Tracer tracer = new Tracer("MyGroup", 10);
```

Then, you can record an event by calling the recordEvent(int eventId) method. The event ID needs to be in the range `0` to `nbEventTypes-1` with `nbEventTypes` the maximum number of event types set when initializing the `Tracer` object. Methods named `recordEvent(...)` always needs the event ID as the first parameter and can have up to ten integer parameters as custom values for the event.

To record the end of an event, call the method recordEventEnd(int eventID). It will trace the duration of an event previously recorded with one of the recordEvent(int eventID) methods. The `recordEventEnd(...)` method can also have another integer parameter for a custom value for the event end. One can use it to trace the returned value of a method.

The Trace API also provides a String constant Tracer.TRACE_ENABLED_CONSTANT_PROPERTY representing the *Constant* value of `core.trace.enabled` option. This constant can be used to *remove at build time* portions of code when the trace system is disabled. To do that, just surround tracer record calls with a if statement that checks the constant's state. When the constant is set to `false`, the code inside the if statement will not be embedded with the application and thus will not impact the performances.

```
if(Constants.getBoolean(Tracer.TRACE_ENABLED_CONSTANT_PROPERTY)) {
  // This␣
↪code is not embedded if TRACE_ENABLED_CONSTANT_PROPERTY is set to false.
  tracer.recordEventEnd(0);
}
```

Examples:

- Trace a single event:

```
private static final Tracer tracer = new Tracer("Application", 100);

public static void main(String[] args) {
  Tracer.startTrace();
  tracer.recordEvent(0);
}
```

Standard Output:

```
MicroEJ START
[TRACE] [1] Declare group "Application"
[TRACE] [1] Event 0x0
```

- Trace a method with a start event showing the parameters of the method and an end event showing the result:

```
private static final Tracer tracer = new Tracer("Application", 100);
```

(continues on next page)

(continued from previous page)

```
public static void main(String[] args) {
  Tracer.startTrace();
  int a = 14;
  int b = 54;
  add(a, b);
}

public static int add(int a, int b) {
  tracer.recordEvent(1, a, b);
  int result = a + b;
  tracer.recordEventEnd(1, result);
  return result;
}
```

Standard Output:

```
MicroEJ START
[TRACE] [1] Declare group "Application"
[TRACE] [1] Event 0x1 (14 [0xE],54 [0x36])
[TRACE] [1] Event End 0x1 (68 [0x44])
```

VEE Port Implementation

More information about the existing implementations and how to implement the required natives are available in the section *Event Tracing* of the VEEPorting Guide chapter.

Advanced Event Tracing

Method invocation can be *profiled*.

Note: This feature requires Architecture version 7.17.0 or higher and is only available on the Core Engine, not on Simulator.

MicroEJ Corp. provides an implementation on Linux targets to profile an Application and generate a flamegraph for the Trace Compass tool.

Please contact *our support team* for more information about how to generate flamegraph.

VEE Debugger Proxy

Principle

The VEE debugger proxy is an implementation of the Java Debug Wire protocol (JDWP) for debugging Applications executed by MICROEJ VEE. It consists of a TCP server implementing the JDWP protocol and acting as a proxy between the IDE (debugger) and the Executable (debuggee) running on the device.

The debugger proxy allows a postmortem debug from a snapshot of the memory (core dump file for Linux/QNX targets and Intel Hex file for MCU targets) of a running Executable binary.

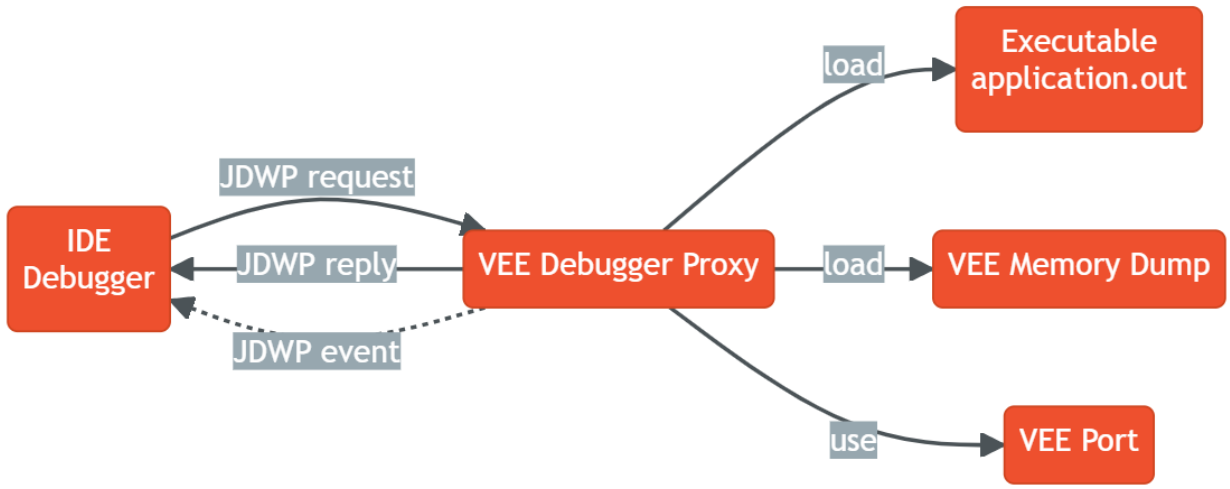


Fig. 169: Debugger Proxy Principle

Warning: The snapshot of the memory (core dump or Intel Hex files) should only be generated when the Core Engine task is stopped on one of the Core Engine hooks (LLMJVM_on_OutOfMemoryError_thrown , LLMJVM_on_Runtime_gc_done etc.) or in a native function. Otherwise, the Core Engine memory dump is not guarranted to be consistent, which may cause the VEE Debugger to crash abruptly.

Note: This feature requires Architecture version 8.1.0 or higher and works for both Mono-Sandbox and Multi-Sandbox Executables.

Installation

Download the VEE Debugger Proxy tool jdwp-server-[version].jar at https://forge.microej.com/artifactory/microej-sdk6-repository-release/com/microej/tool/jdwp-server/1.0.5/jdwp-server-1.0.5.jar .

Debugging Executable for Linux or QNX target

In order to debug an Executable for Linux or QNX target, you need to dump the memory of the running Executable and then run the VEE Debugger Proxy.

For Linux and QNX target, the memory dump must be a core dump file.

Generate a Core Dump File using GDB

Open a shell terminal on the device and enter the following commands:

```
# Instruct the Linux kernel to Dump file-backed private mappings.
echo 0x37 > /proc/self/coredump_filter
# Start GDB
gdb ./application.out
```

The following GDB script can be used to generate a core dump file:

- when the signal SIGUSR1 is received
- or when an out of memory error occurs
- or when an explicit garbage collection (GC) is done.

You can run the script directly in the GDB console.

```
## From GBD documentation:
## generate-core-file [file]
##   Produce a core dump of the inferior process.
##   The optional argument file specifies the file name where to put the core dump.
##   If not specified, the file name defaults to 'core.pid', where pid is the inferior process ID.

## Generate a core dump when the signal SIGUSR1 is received
catch signal SIGUSR1
commands
silent
generate-core-file
cont
end

## Generate a core dump when an out of memory error occurs
break LLMJVM_on_OutOfMemoryError_thrown
commands
silent
generate-core-file
cont
end

## Generate a core dump when an explicit garbage collection (GC) is done
break LLMJVM_on_Runtime_gc_done
commands
silent
generate-core-file
cont
end
```

Starts executing the Mono-Sandbox Executable under GDB:

```
# In the GDB console:
run
```

A core dump file will be generated once the Executable reach one of the breaking conditions described previously.

Run the VEE Debugger Proxy

Open a shell terminal on your workstation and run the following command:

```
java -DveePortDir=<path to VEE Port directory> \
   -Ddebugger.port=<8000> \
   -Ddebugger.out.path=<path to the Executable file (application.out)> \
   -Ddebugger.
↪features.out.path=<comma-separated list of the Feature files with debug information␣
↪(*.fodbg files). To be used if you want to debug an installed Sandboxed Application> \
   -Ddebugger.out.coredump.path=<path to the core dump file> \
   -jar jdwp-server-[version].jar
```

Open the SDK and run a *Remote Java Application Launch* to debug your code.

Debugging Executable for MCU target

The VEE Debugger Proxy for MCU target requires a memory dump of the running Executable in Intel Hex format. It provides a tool to generate a script for IAR (IAR8 or IAR9) or GDB debugger, that contains the needed commands to dump the required memory regions in Intel Hex format.

Generate VEE memory dump script

Open a shell terminal on your workstation and run the following command:

```
java -DveePortDir=<path to VEE Port directory> \
   -Ddebugger.out.path=<path to the Executable file (application.out)> \
   -cp jdwp-server-[version].jar com.microej.jdwp.VeeDebuggerCli \
   --debugger=IAR8|IAR9|GDB \
   --output=<Output directory where the script file will be generated>
```

A script file named vee-memory-dump.mac (for IAR) or vee-memory-dump.gdb (for GDB) is generated into the specified output directory.

You can now use this script to dump the memory of the running Executable.

Dump the memory of the running Executable

IAR Debugger

GNU Debugger (GDB)

Note: You must use a version of IAR Workbench for which the vee-memory-dump.mac script file is generated.

A script file generated for IAR 8.x will not work on IAR Workbench 9.x and vice versa.

In IAR Embedded Workbench:

- Register the generated vee-memory-dump.mac script file in the debugger project option:

1. Open the Debugger Project option window by clicking on Project > Options... > Debugger > Setup
2. Check the option Use macro file(s) and browse to the generated vee-memory-dump.mac file.

3. Click on OK to confirm.

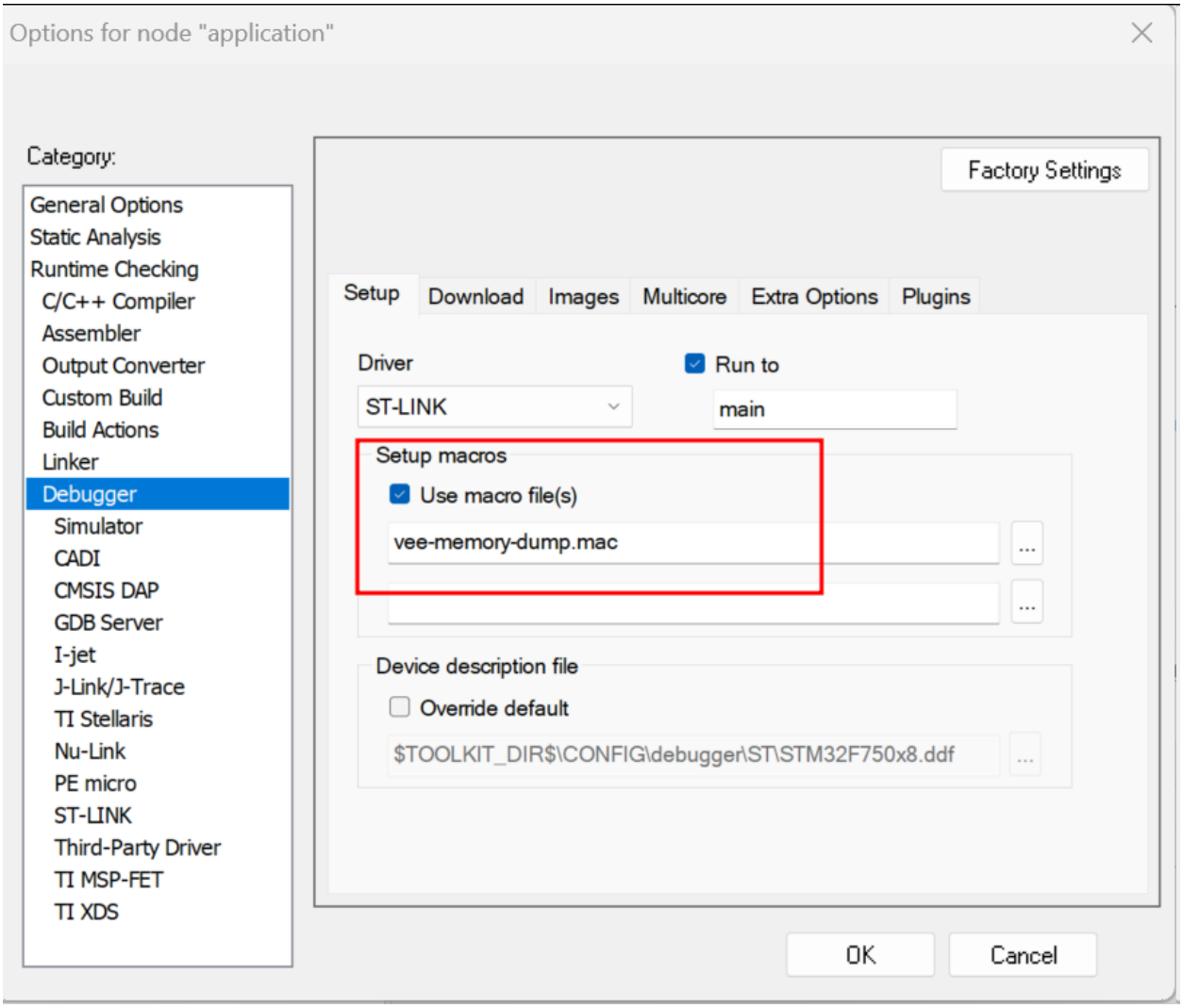


Fig. 170: IAR Debugger Project Option

- Add the macro dumpMemories() as an action expression to a code breakpoint:

1. Open IAR Breakpoints window by clicking on View > Breakpoints
2. Right click on IAR Breakpoints window and select New Breakpoint > Code
3. In the Expression text field, enter dumpMemories() and click on OK

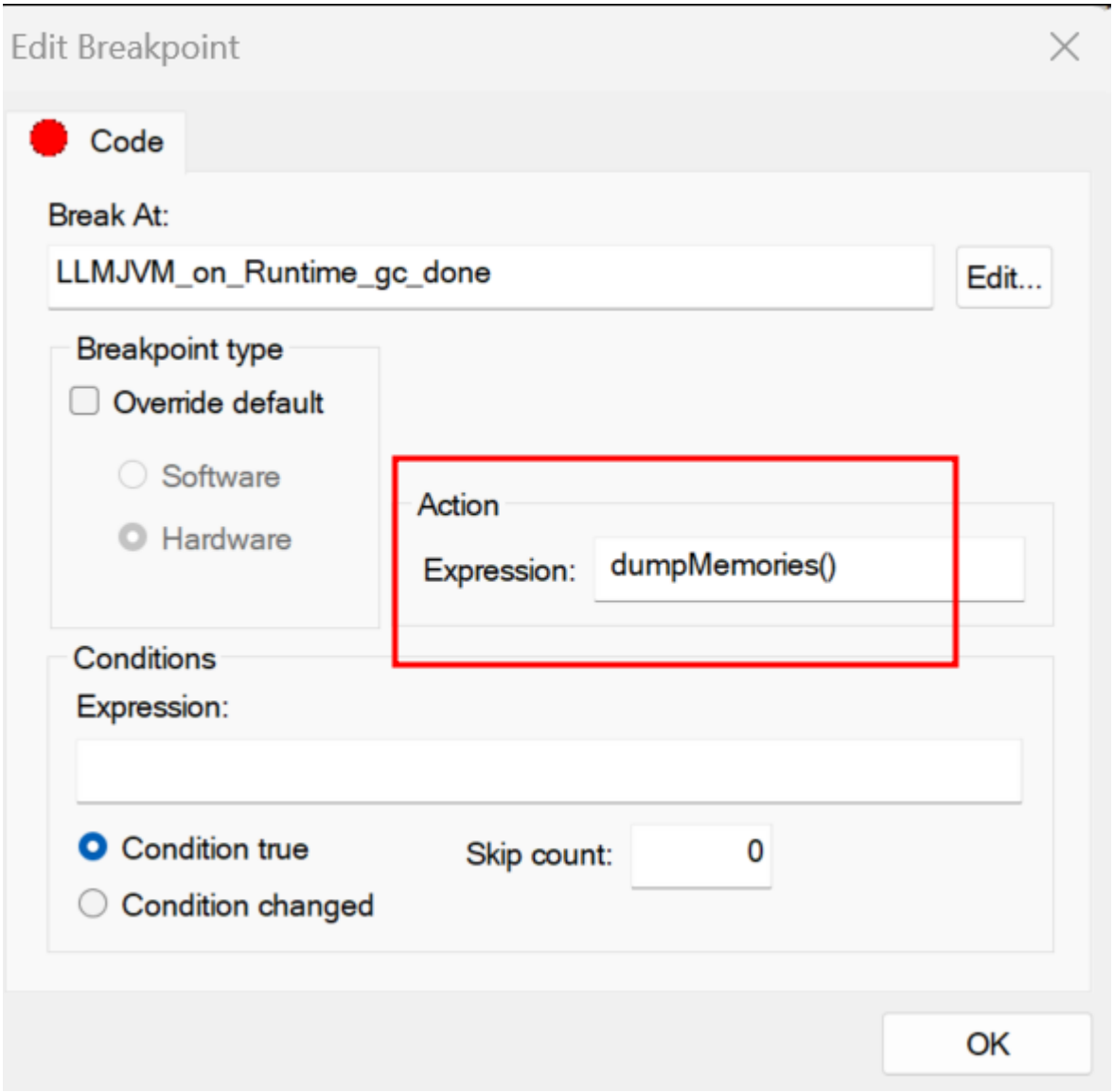


Fig. 171: IAR Breakpoint editor

When the IAR Debugger hits the specified breakpoint, the dumpMemories() macro function is executed and the memory is dumped into *.hex files.

The *.hex files are generated in the same directory as the vee-memory-dump.mac file.

In your GDB console:

- Create a breakpoint at a specific safe point (Core Engine hooks or native function)

```
# E.g. Add breakpoint at LLMJVM_on_Runtime_gc_done hook
break LLMJVM_on_Runtime_gc_done
run
```

- When the running Executable stops at the Breakpoint, run the vee-memory-dump.gdb script file to dump the memory.

```
# E.g. Run the GDB memory dump script
source [/path/to]/vee-memory-dump.gdb
```

The memory is dumped into *.hex files in the same directory as the vee-memory-dump.gdb file.

Start the VEE Debugger Proxy

Open a shell terminal on your workstation and run the following command:

```
java -DveePortDir=<path to VEE Port directory> \
   -Ddebugger.port=<8000> \
   -Ddebugger.out.path=<path to the Executable file (application.out)> \
   -Ddebugger.
↪features.out.path=<comma-separated list of the Feature files with debug information␣
↪(*.fodbg files). To be used if you want to debug an installed Sandboxed Application> \
   -Ddebugger.out.hex.path=<comma-separated list of the␣
↪memory dump files in Intel Hex format or a single file containg all the dumped memory> \
   -jar jdwp-server-[version].jar
```

Open the SDK and run a *Remote Java Application Launch* to debug your code.

Note: If you have multiple *.hex files generated in the previous step, you can if you want merge them into a single *.hex file.

It will be easier to use a single *.hex file than multiple files in the Debugger Proxy command line.

You can run the following shell script to merge all the *.hex files into a single file called all.hex for example.

Make sure to move to the directory where *.hex files are generated before running the script.

- On Windows workstation

```
set ALL_HEX="all.hex"
rem delete all.hex file if it exists
if exist "%ALL_HEX%" (del /f %ALL_HEX%)
rem merge all the *.hex files
copy /b *.hex %ALL_HEX%
```

- On Linux workstation

```
#!/usr/bin/bash
ALL_HEX="all.hex"
#delete all.hex file if it exists
test -f $ALL_HEX && rm $ALL_HEX
#merge all the *.hex files
cat *.hex > $ALL_HEX
```

Now, use this single all.hex file as value to the Debugger Proxy option -Ddebugger.out.hex.path

---

Update the State of the Debugged Application

While the VEE Debugger Proxy is active, you can debug a new application state without restarting it. Simply replace the old memory dump files with the new ones, and the VEE Debugger Proxy will automatically reload the state. After that, relaunch the *Remote Java Application Launch*.

VEE Debugger Proxy Options Summary

- **veePortDir**: The path to the VEE Port directory (must point to the *source* folder of the VEE Port.).
- **debugger.port**: The TCP server port, defaults to `8000`.
- **debugger.out.path**: The Path to the Executable file to debug (`application.out`).
- **debugger.features.out.path**: comma-separated list of the Feature files with debug information (`*.fodbg files`). This option must be used if you want to debug an installed Sandboxed Application. In this case, note that the specified Executable in `debugger.out.path` option must be the Multi-Sandbox Executable.
- **debugger.out.coredump.path**: The Path to the core dump file (conflict with **debugger.out.hex.path** option).
- **debugger.out.hex.path**: The Path to the memory dump files in Intel Hex format (conflict with **debugger.out.coredump.path** option). If you have multiple Intel Hex files, you can either merge them into a single file or list them with a comma separator, such as `[/path/to]/java_heap.hex,[/path/to]/java_stacks.hex,[/path/to]/vm_instance.hex`.

Troubleshooting

You may encounter some command line issues if you try to run the proxy on Windows PowerShell.

On Windows workstation, we recommend using `CMD` Command Prompt instead.

Dependency Discoverer

Introduction

Dependency Discoverer is a tool that lists unresolved dependencies (types, methods and fields) of a set of Java ARchive (JAR) files and .class files. It is a versatile tool and can be used in other contexts, for instance, to list every dependency of a JAR file.

It can be used through a command-line interface, with the possibility to output the result in JSON or XML format, allowing an easy scripting process.

Installation

This tool is available at https://github.com/MicroEJ/Tool-ApiDependencyDiscoverer. A JAR and Windows executable version can be downloaded from the release page. It is also possible to clone and import the project in the SDK and use it from sources.

Use

For usage information, see https://github.com/MicroEJ/Tool-ApiDependencyDiscoverer/blob/master/README.md.

MicroEJ Linker

Overview

MicroEJ Linker is a standard linker that is compliant with the Executable and Linkable File format (ELF).

MicroEJ Linker takes one or several relocatable binary files and generates an image representation using a description file. The process of extracting binary code, positioning blocks and resolving symbols is called linking.

Relocatable object files are generated by SOAR and third-party compilers. An archive file is a container of Relocatable object files.

The description file is called a Linker Specific Configuration file (lsc). It describes what shall be embedded, and how those things shall be organized in the program image. The linker outputs :

- An ELF executable file that contains the image and potential debug sections. This file can be directly used by debuggers or programming tools. It may also be converted into a another format (Intel* hex, Motorola* s19, rawBinary, etc.) using external tools, such as standard GNU binutils toolchain (objcopy, objdump, etc.).
- A map file, in XML format, which can be viewed as a database of what has been embedded and resolved by the linker. It can be easily processed to get a sort of all sizes, call graphs, statistics, etc.
- The linker is composed with one or more library loaders, according to the platform's configuration.

ELF Overview

An ELF relocatable file is split into several sections:

- allocation sections representing a part of the program
- control sections describing the binary sections (relocation sections, symbol tables, debug sections, etc.)

An allocation section can hold some image binary bytes (assembler instructions and raw data) or can refer to an interval of memory which makes sense only at runtime (statics, main stack, heap, etc.). An allocation section is an atomic block and cannot be split. A section has a name that by convention, represents the kind of data it holds. For example, `.text` sections hold binary

instructions, .bss sections hold read-write static data, .rodata hold read-only data, and .data holds read-write data (initialized static data). The name is used in the .lsc file to organize sections.

A symbol is an entity made of a name and a value. A symbol may be absolute (link-time constant) or relative to a section: Its value is unknown until MicroEJ Linker has assigned a definitive position to the target section. A symbol can be local to the relocatable file or global to the system. All global symbol names should be unique in the system (the name is the key that connects an unresolved symbol reference to a symbol definition). A section may need the value of symbols to be fully resolved: the address of a function called, address of a static variable, etc.

Linking Process

The linking process can be divided into three main steps:

1. Symbols and sections resolution. Starting from root symbols and root sections, the linker embeds all sections targeted by symbols and all symbols referred by sections. This process is transitive while new symbols and/or sections are found. At the end of this step, the linker may stop and output errors (unresolved symbols, duplicate symbols, unknown or bad input libraries, etc.)
2. Memory positioning. Sections are laid out in memory ranges according to memory layout constraints described by the lsc file. Relocations are performed (in other words, symbol values are resolved and section contents are modified). At the end of this step, the linker may stop and output errors (it could not resolve constraints, such as not enough memory, etc.)
3. An output ELF executable file and map file are generated.

   A partial map file may be generated at the end of step 2. It provides useful information to understand why the link phase failed. Symbol resolution is the process of connecting a global symbol name to its definition, found in one of the linker input units. The order the units are passed to the linker may have an impact on symbol resolution. The rules are :

- Relocatable object files are loaded without order. Two global symbols defined with the same name result in an unrecoverable linker error.
- Archive files are loaded on demand. When a global symbol must be resolved, the linker inspects each archive unit in the order it was passed to the linker. When an archive contains a relocatable object file that declares the symbol, the object file is extracted and loaded. Then the first rule is applied. It is recommended that you group object files in archives as much as possible, in order to improve load performances. Moreover, archive files are the only way to tie with relocatable object files that share the same symbols definitions.
- A symbol name is resolved to a weak symbol if - and only if - no global symbol is found with the same name.

Linker Specific Configuration File Specification

Description

A Linker Specific Configuration (Lsc) file contains directives to link input library units. An lsc file is written in an XML dialect, and its contents can be divided into two principal categories:

- Symbols and sections definitions.
- Memory layout definitions.

Listing 26: Example of Relocation of Runtime Data from FLASH to RAM

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
  An example of linker specific configuration file
-->
<lsc name="MyAppInFlash">
  <include name="subfile.lscf"/>
  <!--
    Define symbols with arithmetical and logical expressions
  -->
  <defSymbol name="FlashStart" value="0"/>
  <defSymbol name="FlashSize"  value="0x10000"/>
  <defSymbol name="FlashEnd"   value="FlashStart+FlashSize-1"/>
  <!--
    Define FLASH memory interval
  -->
  <defSection name="FLASH" start="FlashStart" size="FlashSize"/>

  <!--
    Some memory layout directives
  -->
  <memoryLayout ranges ="FLASH">
    <sectionRef name ="*.text"/>
    <sectionRef name ="*.data"/>
  </memoryLayout>
</lsc>
```

File Fragments

An lsc file can be physically divided into multiple lsc files, which are called lsc fragments. Lsc fragments may be loaded directly from the linker path option, or indirectly using the include tag in an lsc file.

Lsc fragments start with the root tag lscFragment. By convention the lsc fragments file extension is .lscf. From here to the end of the document, the expression "the lsc file" denotes the result of the union of all loaded (directly and indirectly loaded) lsc fragments files.

Symbols and Sections

A new symbol is defined using defSymbol tag. A symbol has a name and an expression value. All symbols defined in the lsc file are global symbols.

A new section is defined using the defSection tag. A section may be used to define a memory interval, or define a chunk of the final image with the description of the contents of the section.

Memory Layout

A memory layout contains an ordered set of statements describing what shall be embedded. Memory positioning can be viewed as moving a cursor into intervals, appending referenced sections in the order they appear. A symbol can be defined as a "floating" item: Its value is the value of the cursor when the symbol definition is encountered. In *the example below*, the memory layout sets the FLASH section. First, all sections named .text are embedded. The matching sections are appended in a undefined order. To reference a specific section, the section shall have a unique name (for example a reset vector is commonly called .reset or .vector, etc.). Then, the floating symbol dataStart is set to the absolute address of the virtual cursor right after embedded .text sections. Finally all sections named .data are embedded.

A memory layout can be relocated to a memory interval. The positioning works in parallel with the layout ranges, as if there were two cursors. The address of the section (used to resolve symbols) is the address in the relocated interval. Floating symbols can refer either to the layout cursor (by default), or to the relocated cursor, using the relocation attribute. A relocation layout is typically used to embed data in a program image that will be used at runtime in a read-write memory. Assuming the program image is programmed in a read only memory, one of the first jobs at runtime, before starting the main program, is to copy the data from read-only memory to RAM, because the symbols targeting the data have been resolved with the address of the sections in the relocated space. To perform the copy, the program needs both the start address in FLASH where the data has been put, and the start address in RAM where the data shall be copied.

Listing 27: Example of Relocation of Runtime Data from *FLASH* to *RAM*

```
<memoryLayout ranges="FLASH" relocation="RAM" image="true">
  <defSymbol name="DataFlashStart" value="."/>
  <defSymbol name="DataRamStart" value=" ." relocation="true"/>
  <sectionRef name=".data"/>
  <defSymbol name="DataFlashLimit" value="."/>
</memoryLayout>
```

Note: the symbol DataRamStart is defined to the start address where .data sections will be inserted in RAM memory.

Tags Specification

Here is the complete syntactical and semantical description of all available tags of the .lsc file.

Table 37: Linker Specific Configuration Tags

Tags	Attributes	Description
defSection		Defines a new section. A floating section only holds a declared size attribute. A fixed section declares at least one of the start / end attributes. When this tag is empty, the section is a runtime section, and must define at least one of the start , end or size attributes. When this tag is not empty (when it holds a binary description), the section is an image section.
	name	Name of the section. The section name may not be unique. However, it is recommended that you define a unique name if the section must be referred separately for memory positioning.
	start	Optional. Expression defining the absolute start address of the section. Must be resolved to a constant after the full load of the lsc file.
	end	Optional. Expression defining the absolute end address of the section. Must be resolved to a constant after the full load of the lsc file.
	size	Optional. Expression defining the size in bytes of the section. Invariant: (end-start)+1=size . Must be resolved to a constant after the full load of the lsc file.
	align	Optional. Expression defining the alignment in bytes of the section.
	rootSection	Optional. Boolean value. Sets this section as a root section to be embedded even if it is not targeted by any embedded symbol. See also rootSection tag.
	symbolPrefix	Optional. Used in collaboration with symbolTags . Prefix of symbols embedded in the auto-generated section. See *Auto-generated Sections*.
	symbolTags	Optional. Used in collaboration with symbolPrefix . Comma separated list of tags of symbols embedded in the auto-generated section. See *Auto-generated Sections*.
defSymbol		Defines a new global symbol. Symbol name must be unique in the linker context
	name	Name of the symbol.

continues on next page

Table 37 – continued from previous page

Tags	Attributes	Description
	type	Optional. Type of symbol usage. This may be necessary to set the type of a symbol when using third party ELF tools. There are three types: - none : default. No special type of use. - function : symbol describes a function. - data : symbol describes some data.
	value	The value "." defines a floating symbol that holds the current cursor position in a memory layout. (This is the only form of this tag that can be used as a memoryLayout directive) Otherwise value is an expression. A symbol expression must be resolved to a constant after memory positioning.
	relocation	Optional. The only allowed value is true . Indicates that the value of the symbol takes the address of the current cursor in the memory layout relocation space. Only allowed on floating symbols.
	rootSymbol	Optional. Boolean value. Sets this symbol as a root symbol that must be resolved. See also rootSymbol tag.
	weak	Optional. Boolean value. Sets this symbol as a weak symbol.
group		memoryLayout directive. Defines a named group of sections. Group name may be used in expression macros START , END , SIZE . All memoryLayout directives are allowed within this tag (recursively).
	name	The name of the group.
include		Includes an lsc fragment file, semantically the same as if the fragment contents were defined in place of the include tag.
	name	Name of the file to include. When the name is relative, the file separator is / , and the file is relative to the directory where the current lsc file or fragment is loaded. When absolute, the name describes a platform-dependent filename.
lsc		Root tag for an .lsc file.
	name	Name of the lsc file. The ELF executable output will be {name}.out , and the map file will be {name}.map
lscFragment		Root tag for an lsc file fragment. Lsc fragments are loaded from the linker path option, or included from a master file using the include tag.
memoryLayout		Describes the organization of a set of memory intervals. The memory layouts are processed in the order in which they are declared in the file. The same interval may be organized in several layouts. Each layout starts at the value of the cursor the previous layout ended. The following tags are allowed within a memoryLayout directive: defSymbol (under certain conditions), group , memoryLayoutRef , padding , and sectionRef .
	ranges	Exclusive with default. Comma-separated ordered list of fixed sections to which the layout is applied. Sections represent memory segments.
	image	Optional. Boolean value. false if not set. If true , the layout describes a part of the binary image: Only image sections can be embedded. If false , only runtime sections can be embedded.
	relocation	Optional. Name of the section to which this layout is relocated.
	name	Exclusive with ranges. Defines a named memoryLayout directive instead of specifying a concrete memory location. May be included in a parent memoryLayout using memoryLayoutRef.
memoryLayoutRef		memoryLayout directive. Provides an extension-point mechanism to include memoryLayout directives defined outside the current one.
	name	All directives of memoryLayout defined with the same name are included in an undefined order.

continues on next page

Table 37 – continued from previous page

Tags	Attributes	Description
padding		memoryLayout directive. Append padding bytes to the current cursor. Either size or align attributes should be provided.
	size	Optional. Expression must be resolved to a constant after the full load of the lsc file. Increment the cursor position with the given size.
	align	Optional. Expression must be resolved to a constant after the full load of the lsc file. Move the current cursor position to the next address that matches the given alignment. Warning: when used with relocation, the relocation cursor is also aligned. Keep in mind this may increase the cursor position with a different amount of bytes.
	address	Optional. Expression must be resolved to a constant after the full load of the lsc file. Move the current cursor position to the given absolute address.
	fill	Optional. Expression must be resolved to a constant after the full load of the lsc file. Fill padding with the given value (32 bits).
rootSection		References a section name that must be embedded. This tag is not a definition. It forces the linker to embed all loaded sections matching the given name.
	name	Name of the section to be embedded.
rootSymbol		References a symbol that must be resolved. This tag is not a definition. It forces the linker to resolve the value of the symbol.
	name	Name of the symbol to be resolved.
sectionRef		Memory layout statement. Embeds all sections matching the given name starting at the current cursor address.
	file	Select only sections defined in a linker unit matching the given file name. The file name is the simple name without any file separator, e.g. bsp.o or mylink.lsc. Link units may be object files within archive units.
	name	Name of the sections to embed. When the name ends with *, all sections starting with the given name are embedded (name completion), except sections that are embedded in another sectionRef using the exact name (without completion).
	symbol	Optional. Only embeds the section targeted by the given symbol. This is the only way at link level to embed a specific section whose name is not unique.
	force	Optional. Deprecated. Replaced by the rootSection tag. The only allowed value is true. By default, for compaction, the linker embeds only what is needed. Setting this attribute will force the linker to embed all sections that appear in all loaded relocatable files, even sections that are not targeted by a symbol.
	sort	Optional. Specifies that the sections must be sorted in memory. The value can be: - order : the sections will be in the same order as the input files - name : the sections are sorted by their file names - unit : the sections declared in an object file are grouped and sorted in the order they are declared in the object file
u4		Binary section statement. Describes the four next raw bytes of the section. Bytes are organized in the endianness of the target ELF executable.
	value	Expression must be resolved to a constant after the full load of the lsc file (32 bits value).
fill		Binary section statement. Fills the section with the given expression. Bytes are organized in the endianness of the target ELF executable.

continues on next page

Table 37 – continued from previous page

Tags	Attributes	Description
	size	Expression defining the number of bytes to be filled.
	value	Expression must be resolved to a constant after the full load of the lsc file (32 bits value).

Expressions

An attribute expression is a value resulting from the computation of an arithmetical and logical expression. Supported operators are the same operators supported in the Java language, and follow Java semantics:

- Unary operators: + , - , ~ , !
- Binary operators: + , - , * , / , % , << , >>> , >> , < , > , <= , >= , == , != , &, | , ^ , && , ||
- Ternary operator: cond ? ifTrue : ifFalse
- Built-in macros:

– START(name) : Get the start address of a section or a group of sections

– END(name) : Get the end address of a section or a group of sections

– SIZE(name) : Get the size of a section or a group of sections. Equivalent to END(name)-START(name)

– TSTAMPH() , TSTAMPL() : Get 32 bits linker time stamp (high/low part of system time in milliseconds)

– SUM(name,tag) : Get the sum of an auto-generated section (*Auto-generated Sections*) column. The column is specified by its tag name.

An operand is either a sub expression, a constant, or a symbol name. Constants may be written in decimal (127) or hexadecimal form (0x7F). There are no boolean constants. Constant value 0 means false , and other constants' values mean true . Examples of use:

```
value="symbol+3"
value="((symbol1*4)-(symbol2*3)"
```

Note: Ternary expressions can be used to define selective linking because they are the only expressions that may remain partially unresolved without generating an error. Example:

```
<defSymbol name="myFunction" value="condition ? symb1 : symb2"/>
```

No error will be thrown if the condition is true and symb1 is defined, or the condition is false and symb2 is defined, even if the other symbol is undefined.

Auto-generated Sections

The MicroEJ Linker allows you to define sections that are automatically generated with symbol values. This is commonly used to generate tables whose contents depends on the linked symbols. Symbols eligible to be embedded in an auto-generated section are of the form: prefix_tag_suffix. An auto-generated section is viewed as a table composed of lines and columns that organize symbols sharing the same prefix. On the same column appear symbols that share the same tag. On the same line appear symbols that share the same suffix. Lines are sorted in the lexical order of the symbol name. The next line defines a section which will embed symbols starting with zeroinit. The first column refers to symbols starting with zeroinit_start_; the second column refers to symbols starting with zeroinit_end_.

```
<defSection
      name=".zeroinit"
      symbolPrefix="zeroInit"
      symbolTags="start,end"
   />
```

Consider there are four defined symbols named zeroinit_start_xxx, zeroinit_end_xxx, zeroinit_start_yyy and zeroinit_end_yyy. The generated section is of the form:

```
0x00: zeroinit_start_xxx
0x04: zeroinit_end_xxx
0x08: zeroinit_start_yyy
0x0C: zeroinit_end_yyy
```

If there are missing symbols to fill a line of an auto-generated section, an error is thrown.

Execution

MicroEJ Linker can be invoked through an ANT task. The task is installed by inserting the following code in an ANT script

```
<taskdef
   name="linker"
   classname="com.is2t.linker.GenericLinkerTask"
   classpath="[LINKER_CLASSPATH]"
/>
```

[LINKER_CLASSPATH] is a list of path-separated jar files, including the linker and all architecture-specific library loaders.

The following code shows a linker ANT task invocation and available options.

```
<linker
   doNotLoadAlreadyDefinedSymbol="[true|false]"
   endianness="[little|big|none]"
   generateMapFile="[true|false]"
   ignoreWrongPositioningForEmptySection="[true|false]"
   lsc="[filename]"
   linkPath="[path1:...pathN]"
   mergeSegmentSections="[true|false]"
   noWarning="[true|false]"
   outputArchitecture="[tag]"
   outputName="[name]"
```

(continues on next page)

(continued from previous page)

```
   stripDebug="[true|false]"
   toDir="[outputDir]"
   verboseLevel="[0...9]"
>
      <!-- ELF object & archives files using ANT paths / filesets -->
      <fileset dir="xxx" includes="*.o">
      <fileset file="xxx.a">
      <fileset file="xxx.a">

      <!-- Properties that will be reported into .map file -->
      <property name="myProp" value="myValue"/>
</linker>
```

Table 38: Linker Options Details

Option	Description
doNotLoadAlreadyDefinedSymbol	Silently skip the load of a global symbol if it has already been loaded before. (false by default. Only the first loaded symbol is taken into account (in the order input files are declared). This option only affects the load semantic for global symbols, and does not modify the semantic for loading weak symbols and local symbols.
endianness	Explicitly declare linker endianness [little, big] or [none] for auto-detection. All input files must declare the same endianness or an error is thrown.
generateMapFile	Generate the .map file (true by default).
ignoreWrongPositioningForEmptySection	Silently ignore wrong section positioning for zero size sections. (false by default).
lsc	Provide a master lsc file. This option is mandatory unless the linkPath option is set.
linkPath	Provide a set of directories into which to load link file fragments. Directories are separated with a platform-path separator. This option is mandatory unless the lsc option is set.
noWarning	Silently skip the output of warning messages.
mergeSegmentSections	(*experimental*). Generate a single section per segment. This may speed up the load of the output executable file into debuggers or flasher tools. (false by default).
outputArchitecture	Set the architecture tag for the output ELF file (ELF machine id).
outputName	Specify the output name of the generated files. By default, take the name provided in the lsc tag. The output ELF executable filename will be name.out. The map filename will be name.map.
stripDebug	Remove all debug information from the output ELF file. A stripped output ELF executable holds only the binary image (no remaining symbols, debug sections, etc.).
toDir	Specify the output directory in which to store generated files. Output filenames are in the form: od + separator + value of the lsc name attribute + suffix. By default, without this option, files are generated in the directory from which the linker was launched.
verboseLevel	Print additional messages on the standard output about linking process.

Error Messages

This section lists MicroEJ Linker error messages.

Table 39: Linker-Specific Configuration Tags

Message ID	Description
0	The linker has encountered an unexpected internal error. Please contact the support hotline.
1	A library cannot be loaded with this linker. Try verbose to check installed loaders.
2	No lsc file provided to the linker.
3	A file could not be loaded. Check the existence of the file and file access rights.
4	Conflicting input libraries. A global symbol definition with the same name has already been loaded from a previous object file.
5	Completion (*) could not be used in association with the force attribute. Must be an exact name.
6	A required section refers to an unknown global symbol. Maybe input libraries are missing.
7	A library loader has encountered an unexpected internal error. Check input library file integrity.
8	Floating symbols can only be declared inside memoryLayout tags.
9	Invalid value format. For example, the attribute relocation in defSymbol must be a boolean value.
10	Missing one of the following attributes: address, size, align.
11	Too many attributes that cannot be used in association.
13	Negative padding. Memory layout cursor cannot decrease.
15	Not enough space in the memory layout intervals to append all sections that need to be embedded. Check the output map file to get more information about what is required as memory space.
16	A block is referenced but has already been embedded. Most likely a block has been especially embedded using the force attribute and the symbol attribute.
17	A block that must be embedded has no matching sectionRef statement.
19	An IO error occurred when trying to dump one of the output files. Check the output directory option and file access rights.
20	size attribute expected.
21	The computed size does not match the declared size.
22	Sections defined in the lsc file must be unique.
23	One of the memory layout intervals refers to an unknown lsc section.
24	Relocation must be done in one and only one contiguous interval.
25	force and symbol attributes are not allowed together.
26	XML char data not allowed at this position in the lsc file.
27	A section which is a part of the program image must be embedded in an image memory layout.
28	A section which is not a part of the program image must be embedded in a non-image memory layout.
29	Expression could not be resolved to a link-time constant. Some symbols are unresolved.
30	Sections used in memory layout ranges must be sections defined in the lsc file.
31	Invalid character encountered when scanning the lsc expression.
32	A recursive include cycle was detected.
33	An alignment inconsistency was detected in a relocation memory layout. Most likely one of the start addresses of the memory layout is not aligned on the current alignment.
34	An error occurs in a relocation resolution. In general, the relocation has a value that is out of range.

continues on next page

Table 39 – continued from previous page

35	symbol and sort attributes are not allowed together.
36	Invalid sort attribute value is not one of order , name , or no .
37	Attribute start or end in defSection tag is not allowed when defining a floating section.
38	Autogenerated section can build tables according to symbol names (see *Auto-generated Sections*). A symbol is needed to build this section but has not been loaded.
39	Deprecated feature warning. Remains for backward compatibility. It is recommended that you use the new indicated feature, because this feature may be removed in future linker releases.
40	Unknown output architecture. Either the architecture ID is invalid, or the library loader has not been loaded by the linker. Check loaded library loaders using verbose option.
41…43	Reserved.
44	Duplicate group definition. A group name is unique and cannot be defined twice.
45	Invalid endianness. The endianness mnemonic is not one of the expected mnemonics (little,big,none).
46	Multiple endiannesses detected within loaded input libraries.
47	Reserved.
48	Invalid type mnemonic passed to a defSymbol tag. Must be one of none , function , or data .
49	Warning. A directory of link path is invalid (skipped).
50	No linker-specific description file could be loaded from the link path. Check that the link path directories are valid, and that they contain .lsc or .lscf files.
51	Exclusive options (these options cannot be used simultaneously). For example, -linkFilename and -linkPath are exclusive; either select a master lsc file or a path from which to load .lscf files.
52	Name given to a memoryLayoutRef or a memoryLayout is invalid. It must not be empty.
53	A memoryLayoutRef with the same name has already been processed.
54	A memoryLayout must define ranges or the name attribute.
55	No memory layout found matching the name of the current memoryLayoutRef .
56	A named memoryLayout is declared with a relocation directive, but the relocation interval is incompatible with the relocation interval of the memoryLayout that referenced it.
57	A named memoryLayout has not been referenced. Every declared memoryLayout must be processed. A named memoryLayout must be referenced by a memoryLayoutRef statement.
58	SUM operator expects an auto-generated section.
59	SUM operator tag is unknown for the targetted auto-generated section.
60	SUM operator auto-generated section name is unknown.
61	An option is set for an unknown extension. Most likely the extension has not been set to the linker classpath.
62	Reserved.
63	ELF unit flags are inconsistent with flags set using the -forceFlags option.
64	Reserved.
65	Reserved.
66	Found an executable object file as input (expected a relocatable object file).
67	Reserved.
68	Reserved.
69	Reserved.
70	Not enough memory to achieve the linking process. Try to increase JVM heap that is running the linker (e.g. by adding option -Xmx1024M to the JRE command line).

Map File Interpretor

The map file interpretor is a tool that allows you to read, classify and display memory information dumped by the linker map file. The map file interpretor is a graph-oriented tool. It supports graphs of symbols and allows standard operations on them (union, intersection, subtract, etc.). It can also dump graphs, compute graph total sizes, list graph paths, etc.

The map file interpretor uses the standard Java regular expression syntax.

It is used internally by the graphical *Memory Map Analyzer* tool.

Commands:

- `createGraph graphName symbolRegExp ... section=regexp`

  `createGraph all section=.*`

  Recursively create a graph of symbols from root symbols and sections described as regular expressions. For example, to extract the complete graph of the application:

- `createGraphNoRec symbolRegExp ... section=regexp`

  The above line is similar to the previous statement, but embeds only declared symbols and sections (without recursive connections).

- `removeGraph graphName`

  Removes the graph for memory.

- `listGraphs`

  Lists all the created graphs in memory.

- `listSymbols graphName`

  Lists all graph symbols.

- `listPadding`

  Lists the padding of the application.

- `listSections graphName`

  Lists all sections targeted by all symbols of the graph.

- `inter graphResult g1 ... gn`

  Creates a graph which is the intersection of $g1/\backslash \ldots /\backslash gn$.

- `union graphResult g1 ... gn`

  Creates a graph which is the union of $g1\backslash/ \ldots \backslash/ gn$.

- `substract graphResult g1 ... gn`

  Creates a graph which is the substract of $g1\backslash \ldots \backslash gn$.

- `reportConnections graphName`

  Prints the graph connections.

- `totalImageSize graphName`

  Prints the image size of the graph.

- `totalDynamicSize graphName`

  Prints the dynamic size of the graph.

- `accessPath symbolName`

  The above line prints one of the paths from a root symbol to this symbol. This is very useful in helping you understand why a symbol is embedded.

- `echo arguments`

  Prints raw text.

- `exec commandFile`

  Execute the given commandFile. The path may be absolute or relative from the current command file.

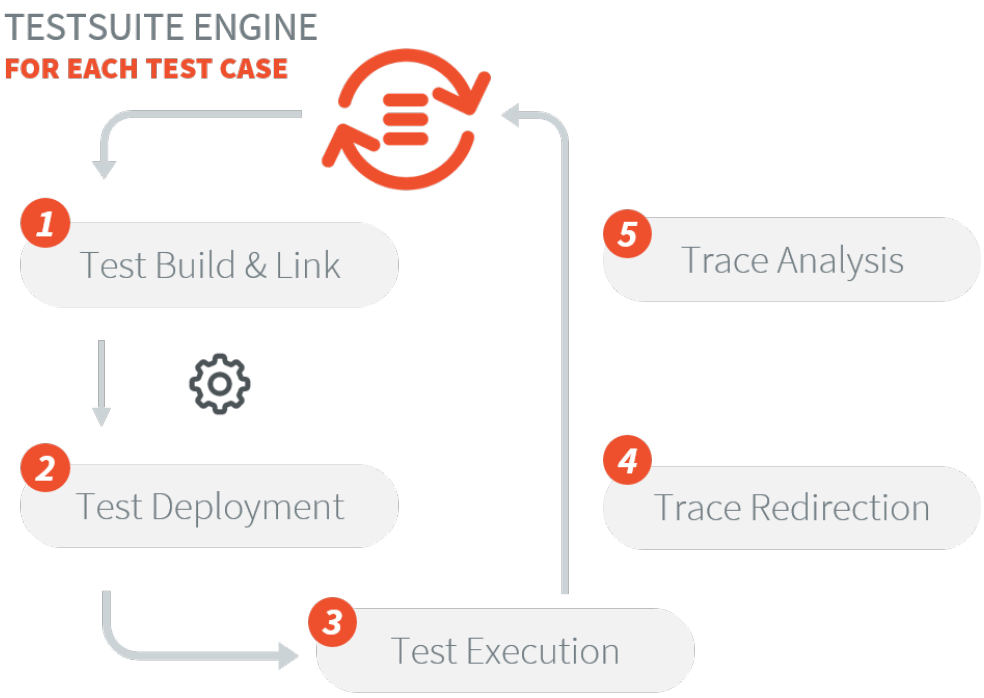
MicroEJ Testsuite Engine

Introduction

The MicroEJ Testsuite Engine is a generic tool made for validating any development project using automatic testing.

This section details advanced configuration for users who wish to integrate custom testsuites in their build flow.

The MicroEJ Testsuite Engine allows the user to test any kind of projects within the configuration of a generic Ant file.



The MicroEJ Testsuite Engine is already pre-configured for running testsuites on a VEE Port (either on Simulator or on Device).

- For Application and Libraries, refer to *Testsuite with JUnit* section.
- For Foundation Libraries Testsuites, refer to *VEE Port Testsuite* section.

Using the MicroEJ Testsuite Ant Tasks

Multiple Ant tasks are available in the `testsuite-engine.jar` provided in the *Build Kit*:

- `testsuite` allows the user to run a given testsuite and to retrieve an XML report document in a JUnit format.
- `javaTestsuite` is a subtask of the `testsuite` task, used to run a specialized testsuite for Java (will only run Java classes).
- `htmlReport` is a task which will generate an HTML report from a list of JUnit report files.

The `testsuite` Task

The following attributes are mandatory:

Table 40: testsuite task mandatory attributes

Attribute Name	Description
outputDir	The output folder of the testsuite. The final report will be generated at [outputDir]/[label]/[reportName].xml, see the testsuiteReportFileProperty and testsuiteReportDirProperty attributes.
harnessScript	The harness script must be an Ant script and it is the script which will be called for each test by the testsuite engine. It is called with a basedir located at output location of the current test.

The testsuite engine provides the following properties to the harness script giving all the informations to start the test:

Table 41: harnessScript properties

Attribute Name	Description
testsuite.test.name	The output name of the current test in the report. Default value is the relative path of the test. It can be manually set by the user. More details on the output name are available in the section *Specific Custom Properties*.
testsuite.test.path	The current test absolute path in the filesystem.
testsuite.test.properties	The absolute path to the custom properties of the current test (see the property customPropertiesExtension)
testsuite.common.properties	The absolute path to the common properties of all the tests (see the property commonProperties)
testsuite.report.dir	The absolute path to the directory of the final report.

The following attributes are optional:

Table 42: testsuite task optional attributes

Attribute Name	Description	Default value
timeOut	The time in seconds before any test is considerated as unknown. Set it to 0 to disable the time-out.	60
verboseLevel	The required level to output messages from the testsuite. Can be one of those values: error, warning, info, verbose, debug.	info
reportName	The final report name (without extension).	testsuite-report
customPropertiesExtension	The extension of the custom properties for each test. For instance, if it is set to .options, a test named xxx/Test1.class will be associated with xxx/Test1.options. If a file exists for a test, the property testsuite.test.properties is set with its absolute path and given to the harnessScript. If the test path references a directory, then the custom properties path is the concatenation of the test path and the customPropertiesExtension value.	.properties
commonProperties	The properties to apply to every test of the testsuite. Those options might be overridden by the custom properties of each test. If this option is set and the file exists, the property testsuite.common.properties is set to the absolute path of the harnessScript file.	no common properties
label	The build label.	timestamp of when the testsuite was invoked.
productName	The name of the current tested product.	TestSuite
jvm	The location of your Java SE VM to start the testsuite (the harnessScript is called as is: [jvm] [...] -buildfile [harnessScript]).	java.home location if the property is set, java otherwise.
jvmargs	The arguments to pass to the Java SE VM started for each test.	None.
testsuiteReportFileProperty	The name of the Ant property in which the path of the final report is stored. Path is [outputDir]/[label]/[reportName].xml	testsuite.report.file
testsuiteReportDirProperty	The name of the Ant property in which is store the path of the directory of the final report. Path is [outputDir]/[label].	testsuite.report.dir
testsuiteResultProperty	The name of the Ant property in which you want to have the result of the testsuite (true or false), depending if every tests successfully passed the testsuite or not. Ignored tests do not affect this result.	None

Finally, you have to give as nested element the path containing the tests.

Table 43: testsuite task nested elements

Element Name	Description
testPath	Containing all the file of the tests which will be launched by the testsuite.
testIgnored-Path (optional)	Any test in the intersection between testIgnoredPath and testPath will be executed by the testsuite, but will not appear in the JUnit final report. It will still generate a JUnit report for each test, which will allow the HTML report to let them appears as "ignored" if it is generated. Mostly used for known bugs which are not considered as failure but still relevant enough to appears on the HTML report.

Listing 28: Example of testsuite task invocation

```
<!-- Launch the testusite engine -->
<testsuite:testsuite
   timeOut="${microej.kf.testsuite.timeout}"
   outputDir="${target.test.xml}/testkf"
   harnessScript="$
↪{com.is2t.easyant.plugins#microej-kf-testsuite.microej-kf-testsuite-harness-jpf-emb.xml.file}"
   commonProperties="${microej.kf.launch.propertyfile}"
   testsuiteResultProperty="testkf.result"
   testsuiteReportDirProperty="testkf.testsuite.report.dir"
   productName="${module.name} testkf"
   jvmArgs="${microej.kf.testsuite.jvmArgs}"
   lockPort="${microej.kf.testsuite.lockPort}"
   verboseLevel="${testkf.verbose.level}"
>
   <testPath refid="target.testkf.path"/>
</testsuite:testsuite>
```

The javaTestsuite Task

This task extends the testsuite task, specializing the testsuite to only start real Java class. This task retrieves the classname of the tests from the classfile and provides new properties to the harness script:

Table 44: javaTestsuite task properties

Property Name	Description
testsuite.test.class	The classname of the current test. The value of the property testsuite.test.name is also set to the classname of the current test.
testsuite.test.classpath	The classpath of the current test.

Listing 29: Example of javaTestsuite task invocation

```
<!-- Launch testsuite -->
<testsuite:javaTestsuite
   verboseLevel="${microej.testsuite.verboseLevel}"
   timeOut="${microej.testsuite.timeout}"
   outputDir="${target.test.xml}/@{prefix}"
   harnessScript="${harness.file}"
```

(continues on next page)

(continued from previous page)

```
   commonProperties="${microej.launch.propertyfile}"
   testsuiteResultProperty="@{prefix}.result"
   testsuiteReportDirProperty="@{prefix}.testsuite.report.dir"
   productName="${module.name} @{prefix}"
   jvmArgs="${microej.testsuite.jvmArgs}"
   lockPort="${microej.testsuite.lockPort}"
   retryCount="${microej.testsuite.retry.count}"
   retryIf="${microej.testsuite.retry.if}"
   retryUnless="${microej.testsuite.retry.unless}"
>
   <testPath refid="target.@{prefix}.path"/>
   <testIgnoredPath refid="tests.@{prefix}.ignored.path" />
</testsuite:javaTestsuite>
```

The htmlReport Task

This task allow the user to transform a given path containing a sample of JUnit reports to an HTML detailed report. Here is the attributes to fill:

- A nested fileset element containing all the JUnit reports of each test. Take care to exclude the final JUnit report generated by the testsuite.
- A nested element report :

– format : The format of the generated HTML report. Must be noframes or frames . When noframes format is choosen, a standalone HTML file is generated.

– todir : The output folder of your HTML report.

– The report tag accepts the nested tag param with name and expression attributes. These tags can pass XSL parameters to the stylesheet. The built-in stylesheets support the following parameters:

* PRODUCT : the product name that is displayed in the title of the HTML report.
* TITLE : the comment that is displayed in the title of the HTML report.

Note: It is advised to set the format to noframes if your testsuite is not a Java testsuite. If the format is set to frames , with a non-Java MicroEJ Testsuite, the name of the links will not be relevant because of the non-existency of packages.

Listing 30: Example of htmlReport task invocation

```
<!-- Generate HTML report -->
<testsuite:htmlReport>
   <fileset dir="${@{prefix}.testsuite.report.dir}">
      <include name="**/*.xml"/> <!-- include unary reports -->
      <exclude name="**/bin/**/*.xml"/> <!-- exclude test bin files -->
      <exclude name="*.xml"/> <!-- exclude global report -->
   </fileset>
   <report format="noframes" todir="${target.test.html}/@{prefix}"/>
</testsuite:htmlReport>
```

Using the Trace Analyzer

This section will shortly explains how to use the Trace Analyzer. The MicroEJ Testsuite comes with an archive containing the Trace Analyzer which can be used to analyze the output trace of an application. It can be used from different forms;

- The FileTraceAnalyzer will analyze a file and research for the given tags, failing if the success tag is not found.
- The SerialTraceAnalyzer will analyze the data from a serial connection.

The TraceAnalyzer Tasks Options

Here is the common options to all TraceAnalyzer tasks:

- successTag : the regular expression which is synonym of success when found (by default .*PASSED.*).
- failureTag : the regular expression which is synonym of failure when found (by default .*FAILED.*).
- verboseLevel : int value between 0 and 9 to define the verbose level.
- waitingTimeAfterSuccess : waiting time (in s) after success before closing the stream (by default 5).
- noActivityTimeout : timeout (in s) with no activity on the stream before closing the stream. Set it to 0 to disable timeout (default value is 0).
- stopEOFReached : boolean value. Set to true to stop analyzing when input stream EOF is reached. If false , continue until timeout is reached (by default false).
- onlyPrintableCharacters : boolean value. Set to true to only dump ASCII printable characters (by default false).

The FileTraceAnalyzer Task Options

Here is the specific options of the FileTraceAnalyzer task:

- traceFile : path to the file to analyze.

The SerialTraceAnalyzer Task Options

Here is the specific options of the SerialTraceAnalyzer task:

- port : the comm port to open.
- baudrate : serial baudrate (by default 9600).
- databits : databits (5|6|7|8) (by default 8).
- stopBits : stopbits (0|1|3 for (1_5)) (by default 1).
- parity : none | odd | event (by default none).

Appendix

The goal of this section is to explain some tips and tricks that might be useful in your usage of the testsuite engine.

Specific Custom Properties

Some custom properties are specifics and retrieved from the testsuite engine in the custom properties file of a test.

- The `testsuite.test.name` property is the output name of the current test. Here are the steps to compute the output name of a test:
  - If the custom properties are enabled and a property named `testsuite.test.name` is find on the corresponding file, then the output name of the current test will be set to it.
  - Otherwise, if the running MicroEJ Testsuite is a Java testsuite, the output name is set to the class name of the test.
  - Otherwise, from the path containing all the tests, a common prefix will be retrieved. The output name will be set to the relative path of the current test from this common prefix. If the common prefix equals the name of the test, then the output name will be set to the name of the test.
  - Finally, if multiples tests have the same output name, then the current name will be followed by `__XXX`, an underscore and an integer.
- The `testsuite.test.timeout` property allow the user to redefine the time out for each test. If it is negative or not an integer, then global timeout defined for the MicroEJ Testsuite is used.

Heap Usage Monitoring

Introduction

When building a *Standalone Application*, the Managed Heap size must be specified as an *Application Option* (see *Option(text): Managed Heap size (in bytes)*). The value to set in this option depends on the maximum heap usage, and the developer can estimate it by running the application.

The Core Engine provides a Java API to introspect the heap usage at runtime. Additionally, heap usage monitoring can be enabled to compute the maximum heap usage automatically.

Here are the descriptions of the different notions related to heap usage:

- **Heap:** memory area used to store the objects allocated by the application.
- **Heap Size:** current size of the heap.
- **Maximum Heap Size:** maximum size of the heap. The heap size cannot exceed this value. See *Option(text): Managed Heap size (in bytes)*.
- **Heap Usage:** the amount of the heap currently being used to store alive objects.
- **Garbage Collector (GC):** a memory manager in charge of recycling unused objects to increase free memory.

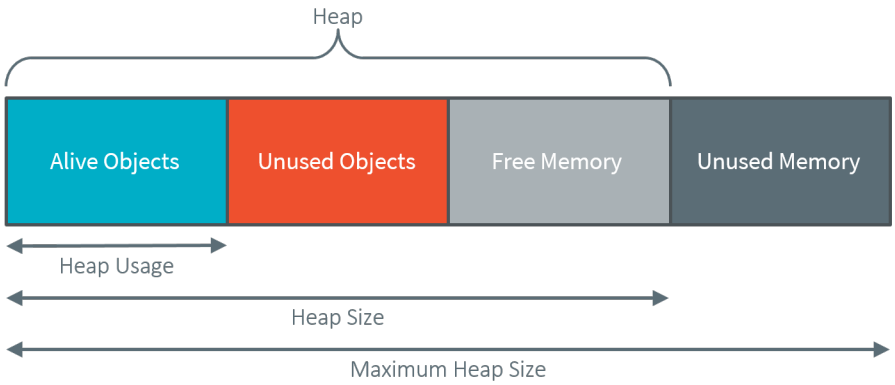


Fig. 172: Heap Structure Summary

The Java class java.lang.Runtime defines the following methods:

- gc(): Runs the garbage collector. System.gc() is an alternative means of invoking this method.
- freeMemory(): Returns the amount of free memory in the heap. This value does not include unused objects eligible for garbage collection. Calling the gc() method may result in increasing the value returned by this method.
- totalMemory(): Returns the current size of the heap. The value returned by this method may vary over time.
- maxMemory(): Returns the maximum size of the heap.

Heap Usage Introspection

The methods provided by the Runtime class allow introspecting the heap usage by comparing the heap size and the free memory size. A garbage collection must be executed before computing the heap usage to recycle all the unused objects and count only alive objects.

The application can compute the current heap usage by executing the following code:

```
Runtime runtime = Runtime.getRuntime(); // get Runtime instance
runtime.gc(); // Ensure unused objects are recycled
long heapUsage = runtime.totalMemory() - runtime.freeMemory();
```

This example gives the heap usage at a given point but not the maximum heap usage of the application.

Note: When heap usage monitoring is disabled, the heap size is fixed, and so totalMemory() and maxMemory() return the same value.

Automatic Heap Usage Monitoring

The maximum heap usage of an application's execution can be computed automatically by enabling heap usage monitoring.

Note: This feature is available in the Architecture versions 7.16.0 or higher for the Applications deployed on hardware devices (not on Simulator).

When this option is activated, an initial size for the heap must be specified, and the Core Engine increases the heap size dynamically. The value returned by totalMemory() is the current heap size. maxMemory() returns the maximum size of the heap. A call to gc() decreases the heap size to the higher value of either the heap usage or the initial heap size.

At any moment, totalMemory() returns the maximum heap usage of the current execution (assuming the maximum heap usage is higher than the initial heap size, and gc() has not been called).

See the section *Option(checkbox): Enable Managed Heap usage monitoring* to enable this option and configure the initial heap size.

Even if the heap size can vary during time, a memory section of maxMemory() bytes is allocated at link time or during the Core Engine startup. No dynamic allocation is performed when increasing the heap size.

Warning: A small initial heap size will impact the performances as the GC will be executed every time the heap size needs to be increased.

Furthermore, the smaller the heap size is, the more frequent the GC will occur. This feature should be used only for heap usage benchmarking.

Heap Usage Analysis

To analyze heap usage and see what objects are alive in the application, use the Heap Dumper & Heap Analyzer tools (*on SDK 6*, *on SDK 5*).

GUI Software Robot

This document presents how to test a GUI application with a software robot for robotic process automation (RPA).

Robot tests and traditional unit tests are different but both are useful. Traditional unit tests validate the systems through calls to the API (internal or external). Robot tests validate the systems by mimicking the human user behavior directly in the GUI. The robot implementation proposed here targets the following errors detection:

- OutOfMemory
- StackOverflow
- MEJ32 and platform libraries error
- Widget sequence validation

The following document covers:

- Recording human touch events on the simulator or on the device
- Running recorded events on the simulator or on the device

  The following document does not cover:

- The display rendering validation (this can be done using the Test Automation Tool)
- Integration of the robot into an automatic JUnit testsuite

  We will now present the basic architecture and code required to create and to run a robot within a MicroEJ application on the simulator and on the device.

  In the following sections, we assume the MicroEJ VEE Port has a display interface and a touch controller.

Overview

The robot creation process is twofold. First, we have to record and store the human user events. Second, we have to play them back with the robot.

Record the Scenario

The first step is to record the human user events.

Here is the code of the EventRecorder class that should be added to our application's project:

```
import ej.annotation.Nullable;
import ej.microui.event.Event;
import ej.microui.event.generator.Buttons;
import ej.microui.event.generator.Pointer;

/**
* Records events.
*/
public class EventRecorder {

  private long lastEventTime;

  /**
  * Creates an event recorder.
  */
  public EventRecorder() {
    this.lastEventTime = -1;
  }

  /**
  * Records an event.
  *
  * @param event
  *            the event to record.
  */
  public void recordEvent(int event) {
    String command = getEventCommand(event);
    if (command != null) {
      long currentTime = System.currentTimeMillis();
      if (this.lastEventTime == -1) {
```

(continues on next page)

(continued from previous page)

```
      this.lastEventTime = currentTime;
    }

    long delta = currentTime - this.lastEventTime;
    if (delta > 0) {
      System.out.println(getPauseCommand(delta));
    }

    System.out.println(command);

    this.lastEventTime = currentTime;
  }
}

@SuppressWarnings("nls")
private static @Nullable String getEventCommand(int event) {
  if (Event.getType(event) == Pointer.EVENT_TYPE) {
    Pointer pointer = (Pointer) Event.getGenerator(event);
    switch (Pointer.getAction(event)) {
    case Pointer.PRESSED:
      return "press(" + pointer.getX() + ", " + pointer.getY() + ");";
    case Pointer.MOVED:
    case Pointer.DRAGGED:
      return "move(" + pointer.getX() + ", " + pointer.getY() + ");";
    case Buttons.RELEASED:
      return "release(" + pointer.getX() + ", " + pointer.getY() + ");";
    default:
      return null;
    }
  } else if (Event.getType(event) == Buttons.EVENT_TYPE) {
    if (Buttons.getAction(event) == Buttons.RELEASED) {
      return "button();";
    } else {
      return null;
    }
  } else {
    return null;
  }
}

@SuppressWarnings("nls")
private static @Nullable String getPauseCommand(long delay) {
  return "pause(" + delay + ");";
}
}
```

This code records all pressed, moved, dragged and released events as well as the time between each event (we want to play our robot at the same speed as the human). EventRecorder outputs the commands on the standard output. More on this a bit later.

Set Up the Event Recorder

The events have to be recorded from the application's desktop's EventDispatcher. Here is how to override it:

```
final EventRecorder eventRecorder = new EventRecorder();

Desktop desktop = new Desktop() {

  @Override
  protected EventDispatcher createEventDispatcher() {
    return new PointerEventDispatcher(this) {

      @Override
      public boolean dispatchEvent(int event) {
        eventRecorder.recordEvent(event);

        return super.dispatchEvent(event);
      }
    };
  }
};
```

When runnning the application, the EventDispatcher will now record the events and then redirect them to its parent dispatchEvent so they can be managed normally by the application.

Set Up the Scenario Player

As we now have recorded our scenario we have to play it. For that we have to add the EventPlayer to our project:

```
/**
 * Plays events.
 */
public class EventPlayer {

  @Nullable
  private final Pointer pointer;
  @Nullable
  private final Buttons buttons;

  /**
  * Creates a robot.
  */
  public EventPlayer() {
    this.pointer = EventGenerator.get(Pointer.class, 0);
    this.buttons = EventGenerator.get(Buttons.class, 1);
  }

  /**
  * Pauses before the next action.
  *
  * @param delay
  *            the delay to pause.
  */
  public void pause(long delay) {
```

(continues on next page)

(continued from previous page)

```
  ThreadUtils.sleep(delay);
}

/**
* Generates a press event.
*
* @param x
*           the x coordinate of the pointer.
* @param y
*           the y coordinate of the pointer.
*/
public void press(int x, int y) {
  if (null != this.pointer) {
    this.pointer.reset(x, y);
  }
  if (null != this.pointer) {
    this.pointer.send(Pointer.PRESSED, 0);
  }
}

/**
* Generates a move event.
*
* @param x
*           the x coordinate of the pointer.
* @param y
*           the y coordinate of the pointer.
*/
public void move(int x, int y) {
  if (null != this.pointer) {
    this.pointer.move(x, y);
  }
}

/**
* Generates a release event.
*
* @param x
*           the x coordinate of the pointer.
* @param y
*           the y coordinate of the pointer.
*/
public void release(int x, int y) {
  if (null != this.pointer) {
    this.pointer.reset(x, y);
  }
  if (null != this.pointer) {
    this.pointer.send(Pointer.RELEASED, 0);
  }
}

/**
* Generates a button event.
*/
public void button() {
  if (null != this.buttons) {
```

(continues on next page)

(continued from previous page)

```
      this.buttons.send(Buttons.RELEASED, 0);
    }
  }
}
```

EventPlayer will play events using the EventGenerator .

We will now extend EventPlayer in order to play a specific scenario:

```
/**
* Robot scenario which reproduces the recorded human user events .
*/
public class NavigationScenario extends EventPlayer implements Runnable {

  @Override
  public void run() {
    press(344, 177);
    pause(885);
    release(344, 177);
    pause(359);
    press(184, 192);
    pause(34);
    move(185, 192);
    pause(24);
    move(188, 192);
    pause(23);
    move(191, 192);
    pause(24);
    move(196, 192);
    pause(21);
    move(206, 191);

  }
}
```

The run method from the code above already contains recorded events, you will have to replace it by the EventRecorder output you get when recording the events.

Run the Scenario

We will now create a task that will run the scenario:

```
  /**
* A robot task is able to run a given scenario.
*/
public class RobotTask {

  private boolean running;

  /**
  * Creates a demo robot.
  */
  public RobotTask() {
    this.running = false;
  }
```

(continues on next page)

(continued from previous page)

```
  /**
  * Starts the given scenario.
  *
  * @param scenario
  *            the scenario to run.
  */
  public void startScenario(final Runnable scenario) {
    if (!this.running) {
      this.running = true;

      new Thread() {
        @Override
        public void run() {
          scenario.run();
          RobotTask.this.running = false;
        }
      }.start();
    }
  }

  /**
  * Returns whether the robot is currently running.
  *
  * @return <code>true</code> if the robot is running, false otherwise</code>.
  */
  public boolean isRunning() {
    return this.running;
  }
}
```

You can now start the `RobotTask` in your application:

```
RobotTask robot = new RobotTask();
                robot.startScenario(new NavigationScenario());
```

Then, launch your application: the recorded scenario is now re-played, well done!

3.6 VEE Porting Guide

3.6.1 Introduction

A *MicroEJ Architecture* is a software package that includes the MicroEJ Runtime port to a specific target Instruction Set Architecture (ISA) and C compiler. It contains a set of libraries, tools and C header files.

A VEE Port is a MicroEJ Architecture port for a custom device.

It relies on C drivers (a.k.a. low level LL drivers) for each VEE Port feature. These drivers are implemented in the VEE Port BSP project. This project is edited in the C compiler IDE/dev environment (e.g. KEIL, GCC, IAR). For example, the MicroUI library LED feature will require a `LLUI_LED.c` that implements the native on/off IO drive.

Each VEE Port is specific to:

- a MicroEJ Architecture (MCU ISA and C compiler)
- an optional RTOS (e.g. FreeRTOS - note: the MicroEJ Runtime can run bare metal)
- a device: the OS bring up code that is device-specific (e.g. the MCU specific code/IO/RAM/Clock/Middleware… configurations)

MicroEJ Corp. provides MicroEJ Evaluation Architectures at https://repository.microej.com/modules/com/microej/architecture, and VEE Port Examples for various evaluation boards.

The VEE Porting Guide explains how the core features are accessed, configured and used to create a port of MICROEJ VEE (VEE Port) on your dedicated device. It also explains how an Application interacts with native code, and the details of the Architecture modules, including their APIs, error codes and options.

Semantics of implemented Foundation Libraries are described in their respective chapters as well as the required Abstraction Layers APIs for porting them to different targets.

VEE Port Build Process

The following figure shows the overall process to create an Executable file to deploy on a device. The first three steps are performed within MICROEJ SDK. The remaining steps are performed within the C IDE.

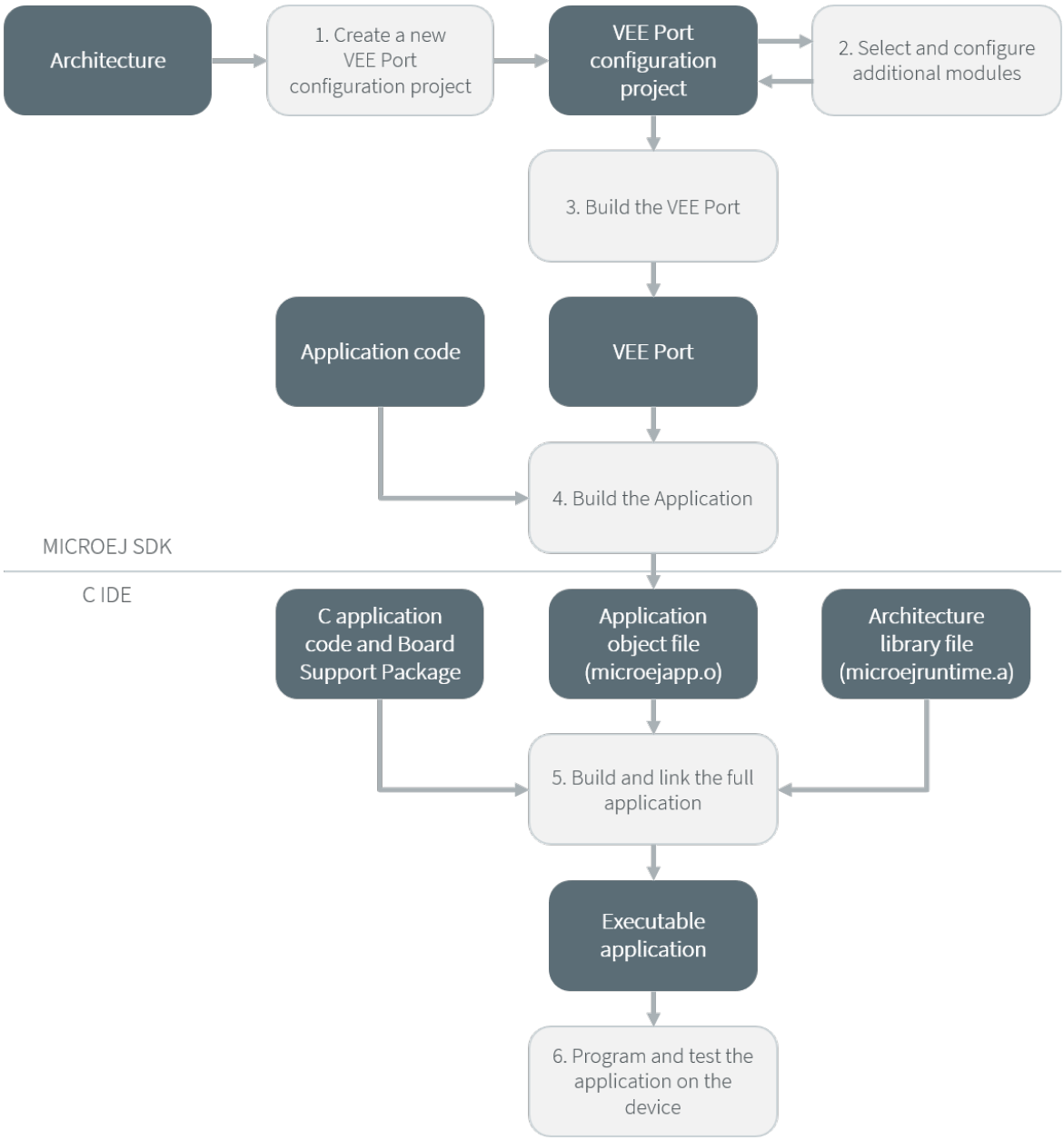


Fig. 173: Overall Build Process

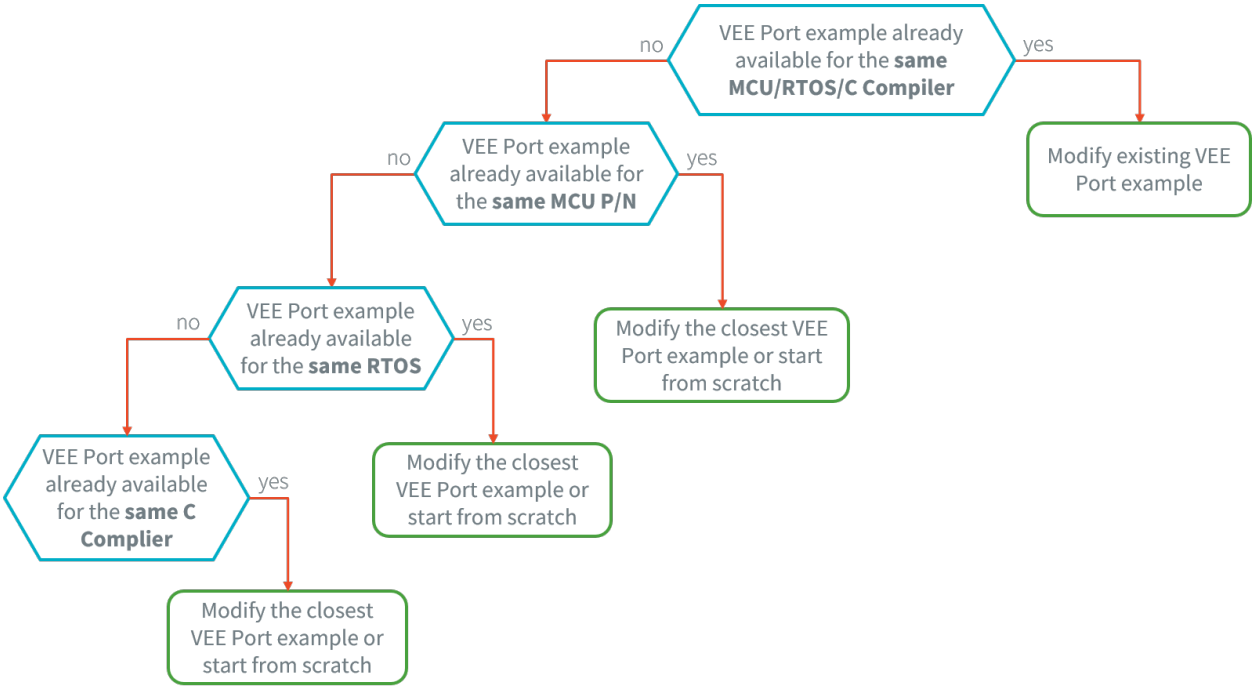
The steps are as follow:

1. Create a new VEE Port configuration project. This project describes the VEE Port to build (Architecture selection).
2. Select which modules provided by the Architecture and Packs will be installed in the VEE Port.
3. Build the VEE Port according to the choices made in steps 1 and 2.
4. Build an Application against the VEE Port in order to obtain an object file to link in the BSP.
5. Compile the BSP and link it with the Application object file that was built previously in step 4 to produce an Executable.

6. Final step: Deploy the Executable (i.e. the binary application) onto a device.

Create a VEE Port for a Custom Device

VEE Port creation can either be done from scratch or modifying an existing VEE Port example project. The following chart provides a handy guide to ease decision:



Before going on, make sure that the new device hardware is validated and at least a trace output is available. It is also a good idea to run basic hardware tests like:

- Internal and external flash programming and verification
- RAM 8/16/32 -bit read/write operations (internal and external if any)
- EEMBC Coremark benchmark to verify the CPU/buses/memory/compiler configuration

A VEE Port Example is already available for the same MCU/RTOS/C Compiler

This is the fastest way: VEE Port Examples are usually provided for a silicon vendor evaluation board. Use this VEE Port project and adapt it to your needs.

As the MCU, RTOS and compiler are the same, only the device-specific code needs to be changed (external RAM, external oscillator, communication interfaces).

VEE Port Configuration

- Modify the VEE Port project to match the device features and its associated configuration (e.g. GUI, Networking, …).

  Refer to VEE Port configuration documentation for *SDK 6* or *SDK 5* to learn more about it.

  More details on available modules can be found in the *VEE Porting Guide*.

BSP

Required actions:

- Modify the BSP C project to match the device specification:
  - Edit the scatter file/link options.
  - Edit the compilation options.
- Create/review/change the platform Low Level C drivers. They must match the device components and the MCU IO pin assignment:

Note: A number of `LL*.h` files are referenced from the project. Implement the function prototypes declared there so that the Core Engine can delegate the relevant operations to the provided BSP C functions.

Simulator

- Modify the existing Simulator Front Panel project.

A VEE Port Example is not available for the same MCU/RTOS/C Compiler

Look for an available VEE Port Example that will match in order of priority:

- same MCU part number
- same RTOS
- same C compiler

At this point, consider either to modify the closest VEE Port:

- Modify the VEE Port configuration.
- in the C IDE, start from an empty project that matches with the MCU.

Or to start from scratch a new VEE Port:

- Create the VEE Port project and refer to the selected VEE Port as a model for implementation (refer to the VEE Port Creation documentation for *SDK 6* or *SDK 5*).
- In the C IDE, start from an empty project and implement the drivers of each of the LL drivers API.

Make sure to link with:

- The `microejruntime.a` that runs the Core Engine for a specific MCU Architecture.

– The `microejapp.o` that contains the compiled Java application.

MCU

The MCU specific code can be found:

- in the C project IDE properties
- in the linker file
- the IO configuration
- in the low level driver (these drivers are usually provided by the silicon vendor)

RTOS

The LL driver is named `LLMJVM_[RTOS].c/.h` . Modify this file to match the selected RTOS.

C Compiler

The BSP project is provided for a specific compiler (that matches the selected platform architecture). Start a new project with the compiler IDE that includes the LL drivers and start the VEE Port in the `main()` function.

Platform Validation

Refer to *VEE Port Qualification* to qualify the VEE Port.

Further Assistance Needed

Please note that porting MicroEJ to a new device is also something that is part of our engineering services. Consider contacting our sales team to request a quote.

3.6.2 MicroEJ Architecture

MicroEJ Architecture features the Core Engine built for a specific instructions set (ISA) and compiler.

The Core Engine is a tiny and fast runtime associated with a Scheduler and a Garbage Collector.

MicroEJ Architecture provides implementations of the following Foundation Libraries :

- Embedded Device Configuration (see *[EDC]*).
- Beyond Profile (see *[BON]*).
- Simple Native Interface (see *[SNI]*).
- Kernel & Features (see *[KF]*).
- Shielded Plug (see *[SP]*).

The following figure shows the components involved.

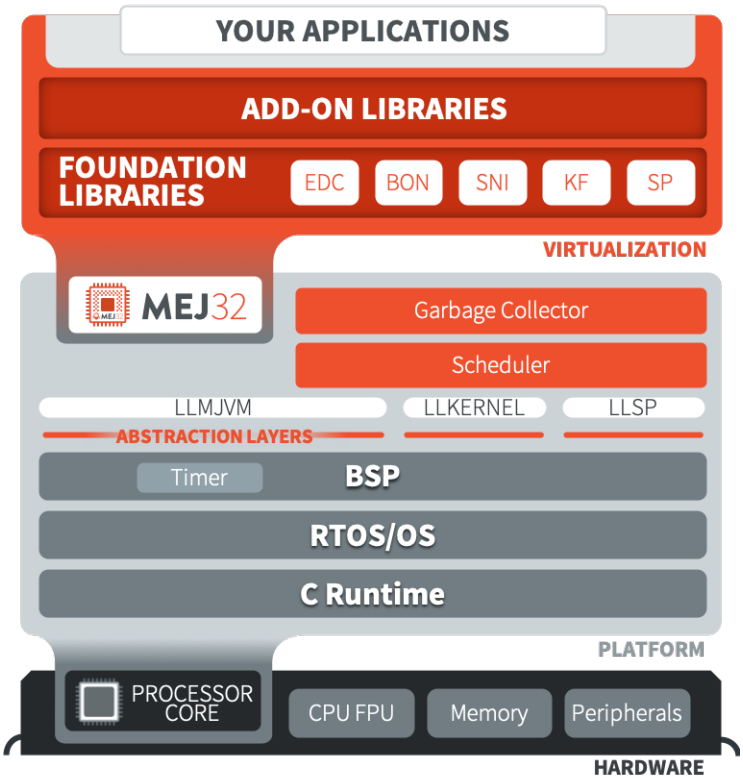


Fig. 174: MicroEJ Architecture Modules

Three Low Level APIs allow the MicroEJ Architecture to link with (and port to) external code, such as any kind of RTOS or legacy C libraries:

- Simple Native Interface (see *[SNI]*)
- Low Level Core Engine (see *LLMJVM*)
- Low Level Shielded Plug (see *LLSP*)

For further information on Architecture installation and releases, you can check these chapters:

Naming Convention

MicroEJ Architecture files ends with the .xpf extension, and are classified using the following naming convention:

```
com/microej/
↪architecture/[ISA]/[TOOLCHAIN]/[UID]/[VERSION]/[UID]-[VERSION]-[USAGE].xpf
```

- ISA : instruction set architecture (e.g. CM4 for Arm® Cortex®-M4, ESP32 for Espressif ESP32, …).
- TOOLCHAIN : C compilation toolchain (e.g. CM4hardfp_GCC48).
- UID : Architecture unique ID (e.g. flopi4G25).
- VERSION : module version (e.g. 7.12.0).

- USAGE = eval for evaluation Architectures, prod for production Architectures.

  For example, MicroEJ Architecture versions for Arm® Cortex®-M4 microcontrollers compiled with GNU CC toolchain are available at https://repository.microej.com/modules/com/microej/architecture/CM4/CM4hardfp_GCC48/flopi4G25/.

  See the VEE Port Creation documentation for *SDK 6* or *SDK 5* for usage.

Architectures Changelog

Notation

A line prefixed by [] describes a change that only applies on a specific configuration: [Core Engine Capability/Instruction Set/C Compiler] :

- Core Engine Capability
  - Mono : Mono-Sandbox (default)
  - Tiny : Tiny-Sandbox
  - Multi : Multi-Sandbox
    - Instruction Set
  - ARM9 : ARM ARM9
  - Cortex-A : ARM Cortex-A
  - Cortex-M : ARM Cortex-M
  - ESP32 : Espressif ESP32
  - RX : Renesas RX
  - x86 : Intel x86
    - C Compilation Toolchain
  - ARMCC5 : Keil ARMCC uVision v5. See also *ARM Linker Specific Options*.
  - Clang : Clang
  - GCC : GNU GCC Compiler. See also *GNU LD Specific Options*.
  - IAR : IAR Embedded Workbench for ARM. See also *IAR Linker Specific Options*.
  - QNX65 : BlackBerry QNX 6.5
  - QNX70 : BlackBerry QNX 7.0

[8.5.0] - 2025-10-09

This Architecture version requires JDK11. See *Check your JDK version* for more details.

Core Engine

- Added the ability to *dynamically configure the Managed Heap and Immortals Heap memory* at Core Engine startup.
- Optimized object allocation performance in cases of highly fragmented heaps.
- Fixed Math.nextAfter(float start, double direction), which returned `start` when `direction` is very close to `start`.
- Removed direct access to `errno` in `microejruntime.a` to avoid toolchain or libc compatibility issues.
- [Multi] - Fixed Feature.stop(), which could infinitely block the Core Engine in some cases.

Foundation Libraries

- Fixed Math.ulp() returning `NaN` instead of positive infinity when called with positive infinity.
- Fixed, in `BON`, `Timer` scheduling of tasks with a time parameter provided as a Date. The date was previously interpreted as Platform time instead of Application time.
- [Multi] - Fixed Kernel.install(java.io.InputStream) to directly throw OutOfMemoryError and StackOverflowError exceptions instead of wrapping them in ej.kf.InvalidFormatException.
- [Cortex-M] - Fixed incorrect handling of `NaN` and `Infinity` inputs in Math.cos(), Math.sin(), Math.tan(), Math.acos(), and Math.asin(), when the underlying C Math library does not process these values properly (the issue was introduced in architecture version *8.4.0*).
- [Cortex-M] - Fixed ej.bon.ByteArray.readXXX() potentialy returning an incorrect value (the issue was introduced in architecture version *8.3.0*).

Integration

- Fixed Memory Map Scripts, which counted the application code in the Core Engine category (the issue was introduced in architecture version *8.4.0*).

SOAR

- Fixed a potential internal error occurring when building an Executable with SDK6 (present in versions up to *SDK6 1.3.1*).
- Fixed SOAR reporting an 'unknown field' error when a library uses a `static` field declared in another library as a `static final` field (previously interpreted as a constant).

Tools

- Fixed, in ELF Utils, `UpdateSection` task used by KF Testsuite to support Portable Independent Executable when ASLR is enabled.

[8.4.0] - 2025-05-28

Core Engine

- Updated *External Resources Loader* implementation to use SNI 1.4 which removes allocations to the Immortals Heap.
- Fixed an issue where `LLMJVM_MONITOR_IMPL_on_thread_state_changed()` was not called when a thread was preempted by another thread due to higher priority or round-robin scheduling.
- [Multi] - Increased the *limitation* on the maximum number of threads from 63 to 127.

Integration

- Added Memory Map Scripts for new Foundation Libraries: `Audio`, `EventQueue`, `Metrology`, `MicroAI` and new Add-On Libraries: `AppConnect`, `ConnectivityManager`, `Eclasspath Time`, `Facer`, `Hoka`, `KFUtil`, `Layout`, `Message`, `NETUtil`, `Property`, `Protobuf`, `Script`, `Storage`.
- Updated Memory Map Scripts for the latest versions of Foundation Libraries: `FS`, `Security`, and Add-On Libraries: `Eclasspath Executor`, `Eclasspath IO`.
- Fixed incorrect assignment of some `.bss`, `.text`, and `.rodata` sections in Memory Map Scripts; these were previously placed in the default `BSP` category.

Simulator

- Added *Mock event tracing*.
- Added, in Front Panel, the ability to resize the window, an options toolbar, and a status bar (see *Front Panel Overview*).
- Fixed, in Front Panel, synchronization on the widget display accesses and rendering of the widgets other than display.
- Fixed initialization of an empty Immortals Heap when *Option(text): Immortals Heap size (in bytes)* is set to 0.
- Fixed the implementation of Tracer.isTraceStarted() that could return `true` when trace recording is not yet enabled in some cases.
- Fixed InputStream.reset() method on a *Resource* that could throw an unexpected IOException after the end of stream is reached.
- Fixed Front Panel not starting at boot. It was previously only displayed after the MicroUI.start() call.

SOAR

- Added an appropriate error message when a resources list file contains an invalid resource declaration.
- Increased the maximum number of blocks allowed in a method to prevent the [M200] - Maximum number of blocks reached error.

[8.3.0] - 2024-12-24

Foundation Libraries

- Fixed, in EDC, implementation of java.util.WeakHashMap.put() which could lead to a memory leak when new elements are added but never accessed.
- Fixed, in EDC, *Enable SecurityManager checks option* was not disabled by default.

Integration

- Added the declaration of *Constants* as *Application Options*.
- Updated Architecture End User License Agreement to version SDK 3.1-c.
- Fixed *Front Panel File option* option was not taken into account on VEE Ports that do not depend on UI Pack.
- Fixed an issue where Sentinel licenses were not displayed in the License Manager in some cases.
- Removed warning messages related to missing *GC mark stack size option* when building on Device.

Simulator

- Added the capability to define *Mock Options*.
- Added a check to verify compatibility with the expected MicroEJ classfile version (1.7).
- Fixed invalid mentions of SNI.closeOnGC() instead of NativeResource.closeOnGC() in the HILEngine Javadoc.
- Fixed potential crash when calling Kernel.clone() in a project that does not define a kernel.kf file.
- Fixed potential crash when booting a GUI Application on a Multi-Sandbox VEE Port.

SOAR

- Added a check to verify compatibility with the expected MicroEJ classfile version (1.7).
- Fixed precedence of a *System Property* declared as an *Application Option* to take priority over one defined in the classpath.

Tools

- Updated License Manager (Evaluation) to debug installed license from command line (see *Check Activation*).

[8.2.0] - 2024-09-19

Core Engine

- [Multi] - Increased execution quota precision
- [Multi] - Added function LLKERNEL_quantaConsumed(int32_t quanta) in LLKERNEL.h , allowing a native function to increment the quantum counter of the current execution context.
- [Multi] - Fixed watchdog which prevents the Core Engine to stop because of a pending thread.
- [Multi] - Added new functions to Low Level API LLMJVM_MONITOR_impl.h for CPU Control monitoring
  - void LLMJVM_MONITOR_IMPL_on_quota_reset(int32_t context_id, int32_t quota) : called by the Core Engine when the quota for an execution context is updated.
  - void LLMJVM_MONITOR_IMPL_on_quota_reached(int32_t context_id) : called by the Core Engine when the quota for an execution context has been reached.
  - void LLMJVM_MONITOR_IMPL_on_quantum_counters_reset() : called by the Core Engine when all the quantum counters are reset to zero.
  - void LLMJVM_MONITOR_IMPL_on_thread_added_to_context(int32_t thread_id, int32_t context_id) : called by the Core Engine when a thread is added to an execution context.

Integration

- Added Architecture tools compatibility with SDKs running on JDK 17 and JDK 21.
- Fixed message to correctly display the BSP location, ensuring compatibility with both SDK 5 and SDK 6.

Simulator

- [Multi] - Fixed, in KF , wrong assertion thrown when starting a Kernel on the Simulator with a pre-installed Application, occurring only when *assertions were enabled on Simulator*.

[8.1.1] - 2024-06-17

Core Engine

- [Multi] - Fixed call to `LLKERNEL_IMPL_freeFeature(int32_t handle)` with handle `0` when an FO file is corrupted or not compatible with the Core Engine.

Foundation Libraries

- Fixed unexpected java.lang.NullPointerException when a runtime exception or a native exception occurs in a try-with-resources block, and the method AutoCloseable.close() throws an exception.
- Fixed Throwable.getSuppressed() which exposes a private mutable member.
- Fixed Throwable.printStackTrace() which does not print suppressed exceptions.
- Fixed Throwable.printStackTrace() which erroneously printed the thread name.

Integration

- [ESP32] Fixed copy of `microejapp.s` into the C project.
- Fixed properties defined in VEE Port `release.properties` file not passed to the SOAR.
- Fixed Virtual Device that could not override HIL options for debugging the Mock.
- Fixed build and run scripts end-of-line (EOL) characters if a Linux VEE port is built on Windows.
- Removed warning messages related to missing HIL debug options when running the Simulator.

Simulator

- Fixed breakpoint not taken into account by IntelliJ Idea's debugger when a class is not loaded during the startup.
- Fixed boostrap thread which could be visible in the thread list when debugging.
- Fixed debugger error when clinit code takes time to execute.
- Fixed debugger stuck when stepping over another breakpoint in Eclipse.
- Fixed missing traces when debug logs are activated.

SOAR

- [Multi] - Fixed the 64 KB size limitation for Java Strings section that caused an `AssertionError` in the SOAR when building a Sandboxed Application.

Tools

- Fixed Resource Buffer Generator that keeps a lock on input files and prevents them from being deleted.

[8.1.0] - 2023-12-22

This Architecture version update introduces the following main features:

- Updated *Feature installation* flow to support Code chunks. A Feature can now be installed to ROM without the need of the Code size in RAM.
- Support for debugging ASLR Executables
- Support for debugging MCU targets
- Support for debugging Multi-Sandbox Executables
- Updated the options to select the *Core Engine capability*. See *Migrate Core Engine Capability Configuration*.
  - Added the VEE Port option `com.microej.runtime.capability`
  - Removed the `Multi Applications` module from the platform configuration file
  - Value of the `BON` constant `com.microej.architecture.capability` is now `mono` instead of `single` when the Core Engine capability is Mono-Sandbox.
- Support of THALES Sentinel License Manager
- Added a *default application* for early-stage VEE Port integration without the need of a SDK license.

  If you plan to migrate a VEE Port from Architecture `8.0.0` to Architecture `8.1.0`, consider the *Architecture 8.0.0 Migration* chapter.

Core Engine

- Added option *com.microej.runtime.core.gc.markstack.levels.max* to configure the maximum number of elements of the Garbage Collector's mark stack.
- In `sni.h`, clarified the behavior of `SNI_createVM()`, `SNI_startVM()`, and `SNI_destroyVM()` when restarting the Core Engine. See also the *Core Engine implementation* section.
- Fixed missing default initialization of the options *core.memory.javaheap.size* and *core.memory.immortal.size*.
- [Multi] - Added a check when `LLKERNEL_IMPL_getFeatureHandle()` returns `0`. Corresponding error code is `LLKERNEL_FEATURE_INIT_ERROR_NULL_HANDLE`.
- [Multi] - Removed Feature installation in RAM (legacy *In-Place Installation mode*). See *Migrate Your LLKERNEL Implementation*.
- [Multi] - Updated *Feature installation boot sequence*: all Feature handles are now retrieved prior to initializing them.
- [Multi] - Updated check of *Kernel UID* at the beginning of Kernel.install(java.io.InputStream), before allocating Feature sections.
- [Multi] - Updated the specification for `LLKERNEL_IMPL_allocateFeature()` function to return the handle `0` if the Feature could not be allocated.

- [Multi] - Updated the specification for `LLKERNEL_IMPL_getAllocatedFeaturesCount()` function to ensure that it returns a valid result at any time, even if it is only called by the Core Engine during startup.
- [Multi] - Updated the specifications for `LLKERNEL_IMPL_getFeatureAddressRAM()` and `LLKERNEL_IMPL_getFeatureAddressROM()` functions to return `NULL` when an incorrect index is provided. This change is only for `LLKERNEL` TCK purposes, as the Core Engine only invokes these methods with valid indices.
- [Multi] - Added an option to enable *RAM Control* at VEE Port build (disabled by default).

Foundation Libraries

- Fixed, in `BON`, ResourceBuffer.readString() which does not increment correctly the position in the buffer.
- Fixed, in `BON`, `-1` returned by ResourceBuffer.available() instead of `0` when the end of the buffer is reached.
- Fixed, in `BON`, invalid value returned by ResourceBuffer.available() on the Simulator.
- Fixed, in `BON`, potential crash when calling ResourceBuffer.close() several times on a `ResourceBuffer` loaded with the *External Resources Loader*.

Integration

- Updated Architecture End User License Agreement to version `SDK 3.1-B`.
- Removed warning messages related to missing KF options when running the SOAR or the Simulator in Mono-Sandbox.

Simulator

- Added compatibility with IntelliJ IDEA IDE to debug applications.
- Added message when waiting for a connection in debug mode.
- Fixed debugger verbose mode.
- Removed bootstrap thread from the debugger vision.
- Fixed debugger suspend count on threads handling.
- Fixed stop issue on static method entry breakpoint.

SOAR

- Fixed trimming of leading or trailing spaces in immutable strings
- [Multi] - Fixed integration of the *bytecode verifier* in Feature mode.
- [Multi] - Improved the error message thrown when no Feature definition file is found and displayed the classpath to better guide developers in identifying potential causes.

Tools

- Updated SOAR and VM Model Readers
  - Added support to retrieve the Core Engine memory regions (used by the VEE Debugger Proxy to generate a memory dump script (see *Generate VEE memory dump script*))
  - Added an API to relink the SOAR Model objects, i.e. change their associated addresses (used by the VEE Debugger Proxy to support ASLR Executables debug)
  - Added new APIs to load Kernel and Features SOAR Model objects (used by the VEE Debugger Proxy to support Multi-Sandbox Executable debug)
- [ARMCC5] - Fixed *SOAR Debug Infos Post Linker* tool to throw a dedicated error when the SOAR object file does not contain the debug section.
- [Multi] - Fixed missing first null entry in the symbol table generated by the Firmware Stripper.

[8.0.0] - 2023-06-27

Note: This Architecture requires SDK version 5.7.0 or higher (see *SDK Version*).

This major Architecture version update introduces the following main features:

- Added compatibility with dynamic linkers enabling Address Space Layout Randomization (ASLR).
- Added *Feature build on device*. For that, the SOAR has been deeply redesigned and split into multiple phases. The most noticeable change is about the *SOAR Information File* that is now composed of 3 files.
- Added Feature portability. The same .fo file can now be installed:
  - On any Executable built from the same Kernel Application (microejapp.o). The VEE Port C code can be modified and relinked without requiring to rebuild the .fo file anymore.
  - On different Kernel Applications provided some conditions are met. Basically, a .fo built on Kernel 1 can be installed on Kernel 2 if the exposed Kernel APIs are left unchanged. See *Feature Portability Control* for more details.
- Redesigned Feature installation flow. A Feature can now be installed in any byte-addressable memory mapped to the CPU's address space, including ROM. For that, LLKERNEL Low Level APIs have been fully rewritten. See *Feature installation* for more details. Former Feature installation in RAM is preserved and is now called *In-Place Installation*. Former static Feature installed by the SDK (using the Firmware Linker tool) is removed in favor of *Feature persistency* at boot.

  If you plan to migrate a VEE Port from Architecture 7.x to Architecture 8.x , consider the *Architecture 7.x Migration* chapter.

Core Engine

- Renamed *Core Engine sections* to fully respect the ELF standard naming convention.
- Removed check when passing a non-immortal array in SNI if VEE Port option `core.sni.nonimmortal.access` was set to `false`.
- Removed `LLBSP_isInReadOnlyMemory` in Core Engine Abstraction Layer (`LLBSP.h` file).
- Clarified `LLMJVM_IMPL_getCurrentTime` API contract in Core Engine Abstraction Layer (`LLMJVM_impl.h` file).
- Updated `Trace` C library from version `1.0.0` to `2.0.0`. See *Migrate Trace C Library Usage*.
  - Renamed header file `trace.h` into `LLTRACE.h` to avoid filename conflicts.
  - Renamed C functions `TRACE_xxx` into `LLTRACE_xxx`.
- Fixed potential crash when Core Engine is restarted after a call to System.exit(int).
- [Multi] - Added option *com.microej.runtime.kernel.dynamicfeatures.max* to configure the maximum number of Features that can be dynamically installed.
- [Multi] - Added option *com.microej.runtime.kf.waitstop.delay* to configure the maximum time allowed for a Feature to stop.
- [Multi] - Fixed missing release of allocated Feature buffers after Core Engine exits (*In-Place Installation* mode).

Foundation Libraries

- Updated `KF` to version `1.7`:
  - Added heap memory control: Module.getAllocatedMemory(), Kernel.setReservedMemory() and Feature.setMemoryLimit() methods.
  - Added load of a Feature resource (Feature.getResourceAsStream() method).
- Updated `KF` dynamic loader to support the new *Feature installation* flow.
- Removed Foundation Libraries API Jars and Javadoc.
- Removed *Unknown product - Unknown version* comment in auto-generated Low Level API header files.
- Removed the `Serial Communication` modules group, including the Foundation Libraries `ECOM` and `ECOM-COMM`. See *Migrate ECOM-COMM Module*.
- Removed the deprecated `Device Information` module group, including the Foundation Library `Device`. See *Migrate Device Module*.
- Fixed *Option(checkbox): Embed UTF-8 encoding* defaults to `true` when building a Standalone Application using MMM.
- Fixed `KF` to call the registered Thread.UncaughtExceptionHandler when an exception is thrown in FeatureEntryPoint.stop().
- Fixed unexpected java.lang.NullPointerException thrown by the `skip` method of an InputStream returned by Class.getResourceAsStream(). This error only occurs with a resource loaded by the External Resource Loader.
- Fixed the behavior of `available`, `read`, `skip`, `mark`, `reset` and `close` methods of an InputStream returned by Class.getResourceAsStream() and previously closed.

- Fixed the LLEXT_RES_read() Low Level API specification (the buffer passed cannot be null).
- [Mono] Fixed an unexpected FeatureFinalizer exception or infinite loop when a Standalone Application touches a KF API in some cases.
- [Tiny] Fixed an unexpected SOAR error when a Standalone Application touches a KF API.
- [Multi] Fixed exception thrown when calling Kernel.removeConverter().
- [Multi] Fixed an unexpected NullPointerException thrown by ej.kf.Kernel.<clinit> method in some cases.
- [Multi] Fixed KF watchdogs not triggered correctly when several expire at the same time.

Integration

- Added support for resolving *Front Panel in Workspace* before the included Front Panel.
- Added Memory Map Scripts for Eclasspath Math , Formatter and DateFormat .
- Updated default value of VEE Port configuration option vendorURL .
- Updated Memory Map Scripts for MicroVG library.
- Updated Memory Map Scripts for Eclasspath Executor library.
- Updated output Map file location to soar/[application_main_class].map (formerly named SOAR.map).
- Removed unused SOAR.o file. It is available at bsp/microejapp.o .
- Renamed MicroEJ launch Build dynamic Feature to Build Feature .
- [Multi] Fixed the SOAR output files from being deleted when the Clean intermediate files option is enabled.

Simulator

- Added *Mock debug* mode.
- Added missing default values for the properties s3.slow , console.logs.period , and s3.hil.timeout when launching the Simulator from the command line.
- Added a check for unsupported access to the Class instance of a primitive type (e.g. byte.class).
- Added HIL Engine debug logs when verbose option is enabled.
- Added log of the Mock classpath when verbose option is enabled.
- Added log of Mock resolution errors (class or method not found).
- Added support for mark/reset on an InputStream returned by Class.getResourceAsStream().
- Fixed “Internal limits” error in HIL engine when too many array arguments are used at the same time by one or several native methods.
- Fixed slow reading with an array of bytes of the input stream returned by Class.getResourceAsStream(String).

- Fixed configuration of the Managed Heap size using *Option(text): Managed Heap size (in bytes)*. The legacy `core.memory.javaheapsum.size` option is not more supported.
- Fixed *Option(text): Immortals Heap size (in bytes)* default value when running a Standalone Application using MMM.
- Fixed stop of the HIL Engine if Simulator was terminated before the connection is established.
- Fixed load of the Mock classes in the classpath order (left-to-right).
- Fixed the missing error check when loading an immutable file referencing an external object id (the `importObject` directive is required).
- Fixed initialization of transparent images in the Front Panel when the initial color is not fully opaque. (introduced in version *7.11.0*)
- [Multi] Fixed the computation of object sizes. The 4-byte KF header was missing.

SOAR

- Added support for *Resource* alignment constraint.
- Added a check for legacy `.system.properties` files in the *Application Classpath*. The build process is stopped and an error is reported. See *Migrate Legacy System Properties Files*.
- Added a check for unsupported access to the Class instance of a primitive type (e.g. `byte.class`).

Tools

- Updated the serial PC connector to JSSC `2.9.4`, including support for macOS aarch64 (M1 chip).
- Removed *Testsuite Engine*. If needed, the Testsuite Engine is available in the *Build Kit*.
- Removed Immutables NLS library. Use *Binary NLS* add-on library instead.
- Fixed an incorrect generation of a debug file beside the memory file when launching the Heap Dumper.
- [Multi] Added Heap Dumper support for dynamically installed Features.

[maintenance/7.20.5] - 2024-05-24

Foundation Libraries

- Fixed, in `BON`, ResourceBuffer.readString() which does not increment correctly the position in the buffer.
- Fixed, in `BON`, `-1` returned by ResourceBuffer.available() instead of `0` when the end of the buffer is reached.
- Fixed, in `BON`, invalid value returned by ResourceBuffer.available() on the Simulator.
- Fixed, in `BON`, potential crash when calling ResourceBuffer.close() several times on a `ResourceBuffer` loaded with the *External Resources Loader*.

[7.20.1] - 2023-04-10

Foundation Libraries

- Fixed Float.parseFloat(…) and Double.parseDouble(…) that don't throw a NumberFormatException when the given string is empty.
- Fixed float and double to string conversions that contain an unecessary + sign in the exponent.

[7.20.0] - 2023-04-04

Known Issues

- Float.parseFloat(…) and Double.parseDouble(…) don't throw a NumberFormatException when the given string is empty.
- Float and double to string conversions contain an unecessary + sign in the exponent.

Core Engine

- Added the capability to customize implementation of the function that performs an atomic exchange operation.
- [ESP32] - Remove default implementation of the function that performs an atomic exchange operation. The Core Engine abstraction layer implementation has to implement the C function `int32_t LLBSP_IMPL_atomic_exchange(int32_t* ptr, int32_t value)`.

Foundation Libraries

- Fixed uninitialized pointer access in the *External Resources Loader*, which can cause a system crash when reading data from a resource.

[7.19.0] - 2023-02-16

Known Issues

- Float.parseFloat(…) and Double.parseDouble(…) don't throw a NumberFormatException when the given string is empty.
- Float and double to string conversions contain an unecessary + sign in the exponent.

Core Engine

- Added the capability to customize implementation of the functions that convert strings to float/double values and vice-versa.
- [Cortex-A/Clang] - Fixed wrong float/double arguments passed to the SNI natives.

Tools

- Removed dependency on GNU `ar` program to create `microejruntime.a` archive file.

[7.18.1] - 2022-10-26

Integration

- Fixed License Manager issue with JDK 8u351 or higher ([M65] - License check failed [tampered (3)].).

[7.18.0] - 2022-09-14

Integration

- Added support for Windows 11.
- Added License Manager support for macOS aarch64 (M1 chip).
- Removed warning when launching Applications or Tools with JDK 11 (*Warning: Nashorn engine is planned to be removed from a future JDK release*).

SOAR

- Added grouping of all immutables objects in a single ELF section.

[7.17.0] - 2022-06-13

Core Engine

- Fixed potential premature evaluation timeout when Core Engine is not started at the same time as the device.
- Fixed potential crash during the call of `LLMJVM_dump` when printing information about the Garbage Collector.
- Added new functions to Low Level API `LLMJVM_MONITOR_impl.h` (see *Advanced Event Tracing*):
  - `void LLMJVM_MONITOR_IMPL_on_invoke_method(void* method)` : called by the Core Engine when an method is invoked.
  - `void LLMJVM_MONITOR_IMPL_on_return_method(void* method)` : called by the Core Engine when a method returns.

- [Cortex-M] - Added support for MCU configuration with unaligned access traps enabled (UNALIGN_TRP bit set in CCR register).

Foundation Libraries

- Updated KF to version 1.6 :
  - Added Kernel.canUninstall() method.

Integration

- Fixed some Architecture tools compatibility issues with SDKs running on JDK 11.
- Fixed missing default value for ShieldedPlug server port when running it with MMM (10082).
- Updated Memory Map Scripts for ej.microvg library.
- Updated Architecture End User License Agreement to version SDK 3.1-A .

Simulator

- Added class file major version check (<=51). Classes must be compiled for Java 7 or lower.
- Added native method signature in the stack trace of the UnsatisfiedLinkError thrown when a native method is missing.
- Fixed HIL engine method NativeInterface.getResourceContent() that generates a runtime error in the Simulator.
- Fixed error “Internal limits reached … S3 internal heap is full” when repeatedly loading a resource that is available in the classpath but not referenced in a .resources.list file.
- Fixed OutOfMemoryError when loading a large resource with Class.getResourceAsStream().
- Fixed A[].class.isAssignableFrom(B[].class) returning false instead of true when B is a subclass of A .
- Fixed potential “Internal limits reached” error when an OutOfMemoryError is thrown.
- Fixed error “Cannot pin objects anymore” when passing repeatedly immutable objects to a native method.
- Fixed properties not passed correctly to the mocks when the Virtual Device is executed from a path that contains spaces.
- [Multi] - Fixed an unexpected error when kernel.kf file is missing and KF library is used: “Please specify a ‘kernel.kf’ file to enable Kernel & Features semantics.”
- [Multi] - Fixed type double[] not recognized in kernel.api file.

SOAR

- Fixed internal error when using a BON constant in an if statement at the end of a `try` block.
- Fixed internal error when a `try` block ends with an `assert` expression while assertions are disabled.
- [Multi] - Raise a warning instead of an error when duplicated `.kf` files are detected in the Kernel classpath. Usual classpath resolution order is used to load the file (see *MicroEJ Classpath*).
- [Multi] - Fixed SOAR error when building a Feature that uses an array of basetypes that is not explicitly declared in the `kernel.api` file of the Kernel.
- [Multi] - Optimized "Build Dynamic Feature" scripts speed by removing unnecessary steps.

[7.16.3] - 2022-04-06

Core Engine

- [Cortex-M/IAR] Fix unaligned stack pointer when calling SNI native functions in ARM IAR architectures.

[7.16.2] - 2021-11-10

Core Engine

- [Cortex-M/GCC/ARMCC5] Fix unaligned stack pointer when calling SNI native functions in ARM GCC and ARMCC architectures with non-ASM Core Engines.

[7.16.1] - 2021-07-16

Core Engine

- [GCC] Fixed wrong inlined extern symbol access (affects only some GCC architectures until version `6.x`). This produces an unexpected `java.lang.OutOfMemoryError: Stacks space` exception at boot time.

[7.16.0] - 2021-06-24

Known Issues

- [Multi] - SOAR may fail to build a Feature with the following message:

```
1 : KERNEL/FEATURE ERROR
    [M25] - Type double[] is expected to be owned by the Kernel but is not embedded.
```

Workaround is to explicitly declare each array of basetypes in your `kernel.api` file:

```
<type name="int[]"/>
<type name="long[]"/>
<type name="short[]"/>
<type name="double[]"/>
```

(continues on next page)

(continued from previous page)

```
<type name="float[]"/>
<type name="byte[]"/>
<type name="char[]"/>
<type name="boolean[]"/>
```

Notes

The Device module provided by the Architecture is deprecated and will be removed in a future version. It has been moved to the Device Pack. Please update your VEE Ports.

Core Engine

- Added a dedicated error code LLMJVM_E_INITIALIZE_ERROR (-23) when LLMJVM_IMPL_initialize() , LLMJVM_IMPL_vmTaskStarted() , or LLMJVM_IMPL_shutdown() fails. Previously the generic error code LLMJVM_E_MAIN_THREAD_ALLOC (-5) was returned.
- Added automatic heap consumption fing when option com.microej.runtime.debug.heap.monitoring.enabled is set to true
- Fixed some parts of LLMJVM_checkIntegrity() code were embedded even if not called
- [Multi] - Fixed potential crash during the call of LLMJVM_checkIntegrity() when analyzing a corrupted stack (make this function robust to object references with an invalid memory address)

Foundation Libraries

- Added source code for KF , SCHEDCONTROL , SNI , SP implementations
- Updated KF API with annotations for Null analysis
- Updated SNI API with annotations for Null analysis
- Updated SP API with annotations for Null analysis
- Updated ResourceManager implementation with annotations for Null analysis
- Updated KF implementation:
  - Added missing Kernel.getAllFeatureStateListeners() method
  - Updated code for correct Null analysis detection
  - Fixed Feature.getCriticality() to throw IllegalStateException if it is in state UNINSTALLED (instead of returning NORM_CRITICALITY)
  - Fixed potential race condition between Kernel.addResourceControlListener() and Kernel.removeResourceControlListener(). Adding a new listener may not register it if another one is removed at the same time.

Integration

- Added a new task in ELF Utils library allowing to update the content of an ELF section:
  - Declaration:

```
<taskdef classpath="${platform.dir}
↪/tools/elfutils.jar" classname="com.is2t.elf.utils.AddSectionTask" name="addSection" />
```

  - Usage:

```
<addSection file="${executable.
↪file}" sectionFile="${section.file}" sectionName="${section.name}" sectionAlignment=
↪"${section.alignment}" outputDir="${output.dir}" outputName="${output.name}" />
```

- Updated Architecture End User License Agreement to version SDK 3.0-C
- Updated copyright notice of Low Level APIs header files to latest SDK default license
- Updated Architecture module with required files and configurations for correct publication in a module repository (README.md , LICENSE.txt , and CHANGELOG.md)

Simulator

- Added an option (com.microej.simulator.hil.frame.size) to configure the HIL engine max frame size
- Fixed load of an immutable byte field (sign extension)
- Fixed java.lang.String constructors String(byte[] bytes, ...) when passing characters in the range [0x80,0xFF] using default ISO-8859-1 encoding
- Fixed potential crash in debug mode when a breakpoint is set on a field access (introduced in version 7.13.0)
- Fixed wrong garbage collection of an object only referenced by an immortal object

SOAR

- Fixed the following compilation issues in if statement with BON constant:
  - too many code may be removed when the block contains a while loop
  - potential Stacks merging coherence error may be thrown when the block contains a nested try-catch statement
  - potential Stacks merging coherence error when declaring a ternary expression with Constants.getBoolean() in condition expression
- Fixed assert statement removal when it is located at the end of a then block: the else block may be executed instead of jumping over
- Removed names of arrays of basetype unless soar.generate.classnames option is set to true
- [Multi] - Fixed potential link exception when a Feature use one of the ej.bon.ByteArray methods (e.g. ej.kf.InvalidFormatException: code=51:ON_ej_bon_ByteArray_method_readUnsignedByte_AB_I_I)

- [Multi] - Fixed SOAR error (Invalid SNI method) when one of the ej.bon.Constants.getXXX() methods is declared in a kernel.api file. This issue was preventing from using BON Constants in Feature code.

Tools

- Updated Code Coverage Analyzer report generation:
  - Automatically configure src/main/java source directory beside a /bin directory if available
  - Added an option (cc.src.folders) to specify the source directory (require SDK 5.4.1 or higher)
  - Removed the analysis of generated code for synchronized statements
  - Fixed crash when loading source code with annotations
- Fixed Memory Map scripts: ClassNames group may contain duplicate sections with Types group
- Fixed load of an ELF executable when a section overlaps a segment (updated ELF Utils, Kernel Packager and Firmware Linker)
- Fixed Firmware Linker to generate output executable file at the same location than the input executable file

[7.15.1] - 2021-02-19

SOAR

- [Multi] - Fixed potential Core Engine crash when declaring a Proxy class which is abstract .

[7.15.0] - 2020-12-17

Core Engine

- Added support for applying Feature relocations

Foundation Libraries

- Updated KF implementation to apply Feature relocations using the Core Engine. The former Java implementation is deprecated but can still be enabled using the option com.microej.runtime.kf.link.relocations.java.enabled .

Integration

- Updated the Architecture naming convention: the usage level is `prod` instead of `dev` .
- Fixed generation of temporary properties file with a `.properties.list` extension instead of deprecated `.system.properties` extension.

SOAR

- Fixed crash when declaring a clinit dependency rule on a class that is loaded but not embedded.

Tools

- Fixed Memory Map Script `All` graph creation to prevent slow opening of large `.map` file in Memory Map Analyzer.

[7.14.1] - 2020-11-30

Core Engine

- [Multi/x86/QNX7] - Fixed missing multi-sandbox version

Tools

- Fixed categories for class names and SNI library in Memory Map Scripts

[7.14.0] - 2020-09-25

Notes

The following set of Architecture properties are automatically provided as `BON` constants:

- `com.microej.architecture.capability=[tiny|single|multi]`
- `com.microej.architecture.name=[architecture_uid]`
- `com.microej.architecture.level=[eval|prod]`
- `com.microej.architecture.toolchain=[toolchain_uid]`
- `com.microej.architecture.version=7.14.0`

Note: Starting from *Architecture 8.1.0*, `com.microej.architecture.capability` constant is set to `mono` instead of `single` when the Core Engine capability is Mono-Sandbox.

The following set of VEE Port properties (customer defined) are automatically provided as `BON` constants:

- `com.microej.platform.hardwarePartNumber`
- `com.microej.platform.name`

- com.microej.platform.provider
- com.microej.platform.version
- com.microej.platform.buildLabel

Foundation Libraries

- Updated EDC UTF-8 encoder to support Unicode code points as supplementary characters
- Fixed java.lang.NullPointerException thrown when java.util.WeakHashMap.put() method is called with a null key (introduced in version *7.11.0*)

Integration

- Added all options starting with com.microej. prefix as BON constants
- Added all properties defined in architecture.properties as options prefixed by com.microej.architecture.
- Added all properties defined in release.properties as options prefixed by com.microej.platform.
- Added all properties defined in script/mjvm.properties as options prefixed by com.microej.architecture.
- Added an option (com.microej.library.edc.supplementarycharacter.enabled) to enable support for supplementary characters (enabled by default)
- Updated Memory Map Scripts to extract Java static fields in a dedicated group named Statics
- Updated Memory Map Scripts to extract Java types in a dedicated group named Types
- Fixed generated Feature filename (unexpanded ${feature.output.basename} variable, introduced in version *7.13.0*)
- Fixed definition of missing default values for memory options (same values than launcher default ones)
- [Tiny,Multi] - Added display of the Core Engine capability when launching SOAR

SOAR

- [Multi] - Added a new attribute named api in Kernel soar.xml file indicating which types, methods and static fields are exposed as Kernel APIs
- [Multi] - Fixed potential link error when calling Object.clone() method on an array in Feature mode

Tools

- Updated the serial PC connector to JSSC 2.9.2 (COM port could not be open on Windows 10 using a JRE 8u261 or higher)

[7.13.3] - 2020-09-18

Core Engine

- [QNX70] - Embed method names and line numbers information in the application
- [Cortex-A/QNX70] - Fixed wrong float/double arguments passed to the SNI natives (introduced in version *7.12.0*)

Simulator

- Fixed unnecessary stacktrace dump on Long.parseLong(...) error
- Fixed UTF-8 encoded Strings not correctly printed

Tools

- Updated Memory Map Scripts for ej.library.runtime.basictool library

[7.13.2] - 2020-08-14

Core Engine

- [ARM9/QNX65] - Fixed custom convention call
- [x86/QNX70] - Fixed SIGFPE raised when overflow occurs on division
- [x86/QNX70] - Fixed issue with NaN conversion to int or long

Tools

- Fixed Feature build script for SDK 5.x (introduced in version *7.13.0*)
- Updated Memory Map Scripts for MicroUI 3 and Service libraries

[7.13.1] - 2020-07-20

Core Engine

- [ESP32] - Fixed potential PSRAM access faults by rebuilding using esp-idf v3.3.0 toolchain (simikou2)

[7.13.0] - 2020-07-03

Core Engine

- Added SNI-1.4 support, with the following new LLSNI.h Low Level APIs:
  - Added function SNI_registerResource()
  - Added function SNI_unregisterResource()
  - Added function SNI_registerScopedResource()
  - Added function SNI_unregisterScopedResource()
  - Added function SNI_getScopedResource()
  - Added function SNI_retrieveArrayElements()
  - Added function SNI_flushArrayElements()
  - Added function SNI_isResumePending()
  - Added function SNI_clearCurrentJavaThreadPendingResumeFlag()
  - Added define SNI_VERSION
  - Added define SNI_IGNORED_RETURNED_VALUE
  - Added define SNI_ILLEGAL_ARGUMENT
  - Updated the documentation of some functions to clarify the behavior
- Added a message to IllegalArgumentException thrown in an SNI call when passing a non-immortal array in SNI (only in case the VEE Port is configured to disallow the use of non-immortal arrays in SNI native calls)
- Added function LLMJVM_CheckIntegrity() to LLMJVM.h Low Level API to perform heap and internal structures integrity check
- Updated KF implementation to use SNI-1.4 to close native resources when the Feature is stopped (ej.lang.ResourceManager is now deprecated)
- Updated LLMJVM_dump() output with the following new information related to SNI-1.4 native resource management:
  - Last native method called (per thread)
  - Current native method being invoked (per thread)
  - Last native resource close hook called (per thread)
  - Current native resource close hook being invoked (per thread)
  - Pending Native Exception (per thread)
  - Pending SNI Scoped Resource to close (per thread)

  - Current Garbage Collector state: (running or not, last scanned object address, last scanned object class)
  - LLMJVM schedule request (global and per thread)
- Updated non-immortal array access from SNI default behavior (now allowed by default)
- Fixed thread state displayed by LLMJVM_dump for threads in SLEEP state
- Fixed sni.h header file function prototypes using the SNI_callback typedef
- Fixed crash when an OutOfMemoryError is thrown while creating a native exception in SNI
- [Multi] - Fixed runtime exceptions that can be implicitly thrown (such as NullPointerException) which were not automatically exposed by the Kernel
- [Multi] - Fixed passing Kernel array parameters through a shared interface method call. These parameters were passed by copy instead of by reference as specified by KF specification
- [Multi] - Fixed execution context when jumping in a catch block of a ej.kf.Proxy method (the catch block was executed in the Kernel context instead of the Feature context)
- [ARMCC5] - Fixed link error Undefined symbol __java_Ljava_lang_OutOfMemoryError_field_OOMEMethodAddr_I with ARM Compiler 5 linker (introduced in version *7.12.0*)

Foundation Libraries

- Updated SNI to version 1.4
- Updated internal library Resource-Manager-1.0 as deprecated. Use SNI-1.4 native resources instead
- Updated Thread.getId() method implementation to return the same value than SNI_getCurrentJavaThreadID() function
- Optimized SNI.toCString() method by removing a useless temporary buffer copy
- Fixed EDC implementation of String(byte[],int,int) constructor which could allocate a too large temporary buffer
- Fixed EDC implementation of Thread.interrupt() method to throw a java.lang.SecurityException when the interrupted thread cannot be modified by the the current thread
- Fixed EDC implementation to remove remaining references to java.util.SecurityManager class when it is disabled
- Fixed EDC implementation of Thread.interrupt() method that was declared final
- Fixed EDC API of Thread.interrupt() to clarify the behavior of the method
- Fixed EDC API of java.util.Calendar method to specify that non-lenient mode is not supported
- Fixed EDC API of java.io.FilterInputStream.in field to be marked @Nullable

Integration

- Updated Architecture End User License Agreement to version SDK 3.0-B

Simulator

- Added SNI-1.4 support, with the following new HIL engine APIs:
  - Added methods NativeInterface.suspendStart() and NativeInterface.suspendStop() to notify the simulator that a native is suspended so that it can schedule a thread with a lower priority
- Added KF support to dynamically install Features (.fs3 files)
- Added the capability to specify the Kernel UID from an option (see options in Simulator > Kernel > Kernel UID)
- Added object size in generated .heap dump files
- Optimized file accesses from the Application
- Fixed crash in debug mode when paused on a breakpoint in SDK and hovering a Java variable with the mouse
- Fixed potential crash in debug mode when putting a breakpoint in the SDK on a line of code declared in an inner class
- Fixed potential crash in debug mode (java.lang.NullPointerException) when a breakpoint set on a field access is hit
- Fixed potential crash in debug mode (ArrayIndexOutOfBoundsException)
- Added support for JDWP commands DisableCollection / EnableCollection in the debugger
- Fixed invalid heap dump generation in debug mode.
- Fixed crash when a Mockup implements com.is2t.hil.StartListener and this implementation throws an uncaught exception in the clinit
- Fixed verbose of missing resource only when a resource is available in the classpath but not declared in a .resources.list file
- Fixed heap consumption simulation for objects instances of classes declaring fields of type float or double
- Fixed Device UID not displayed in the Front Panel window title (introduced in version *7.11.0*)
- Fixed loading of a resource from a JAR when the path starts with /
- Fixed potential deadlock on Front Panel startup in some cases
- Fixed Thread.getState() returning TERMINATED whereas the thread is running
- Fixed Simulator which may not stop properly when closing the Front Panel window
- Fixed Front Panel which stops sending widget events when dragging out of a widget
- [Multi] - Fixed monitor that may not be released when an exception occurs in a synchronized block (introduced in version 7.10.0)
- [Multi] - Fixed invalid heap dump generation that causes heap analyzer crash
- [Multi] - Fixed potential crash (java.lang.NullPointerException) in debug mode when debugging an Application (introduced in version *7.10.0*)

- [Multi] - Fixed error when using KF library without defining a kernel.kf file in the Kernel (introduced in version *7.10.0*)

SOAR

- Added an option (soar.bytecode.verifier) to enable or disable the bytecode verifier (disabled by default)
- Removed size related limits in Architecture Evaluation version

Tools

- Added SNI-1.4 support to HIL engine
- Updated Heap Dumper to verbose information about the memory section when an overlap is detected in the HEX file
- Updated Memory Map Scripts (Security, DTLS, Device)
- Fixed License Manager (Evaluation) random crash on Windows 10 when a VEE Port is built using Build Module button
- Fixed License Manager (Evaluation) wrong UID computation after reboot when Windows 10 Hyper-V feature is enabled
- Fixed HIL engine to exit as soon as the Simulator is disconnected (avoid remaining detached processes)
- Fixed ELF to Map generating symbol addresses different from the ELF symbol addresses (introduced in version *7.11.0*)
- Fixed Heap Dumper crash when a wrong object header is encountered
- Fixed Heap Dumper failure when a memory dump is larger than the heap section
- Fixed Heap Dumper crash when loading an Intel HEX file that contains lines of type 02

[7.12.0] - 2019-10-16

Core Engine

- Updated implementation of internal OutOfMemoryError thrown with the maximum number of frames that can be dumped
- Updated LLMJVM_dump() output with the following new information:
  - Maximum number of alive threads
  - Total number of created threads
  - Maximum number of stack blocks used
  - Current number of stack blocks used
  - Objects referenced by each call frame: address, type, length (in case of arrays), string content (in case of String objects)
  - [Multi] - Kernel stale references with the name of the Feature stopped

Foundation Libraries

- Fixed EDC implementation of Throwable.getStackTrace() when called on a OutOfMemoryError thrown by Core Engine or Simulator (either the returned stack trace array was empty or a java.lang.NullPointerException was thrown)
- [Tiny] - Fixed EDC implementation of StackTraceElement.toString() (removed the character . before the type)
- [Multi] - Fixed KF implementation of Feature.start() to throw an ExceptionInInitializerError when an exception is thrown in a Feature clinit method

Simulator

- Updated implementation of internal OutOfMemoryError thrown with more than one frames dumped per thread
  - By default the 20 top frames per thread are dumped. This can be modified using S3.OutOfMemoryErrorNbFrames system property
- Fixed wrong parsing of an array of long when an element is declared with only 2 digits (e.g. 25 was parsed as 2)
- Fixed error parsing of an array of byte when an element is declared with the unsigned hexadecimal notation (e.g. 0xFF) (introduced in version *7.10.0*)
- Fixed crash when ResourceBuffer.readString() is called on a String greater than 63 characters (introduced in version *7.10.0*)
- Fixed code coverage .cc generation of classpath directories
- Fixed crash during a GC when computing the references map of a complex method (an error message is dumped with the involved method name and suggest to increase the internal stack using S3.JavaMemory.ThreadStackSize system property)
- [Multi] - Added validity check of Shared Interface declaration files (.si) according to KF specification
- [Multi] - Fixed processing of Resource Buffers declared in Feature classpath

SOAR

- Added a new option core.memory.oome.nb.frames to configure the maximum number of call frames that can be dumped when an internal OutOfMemoryError is thrown by Core Engine

Tools

- Updated Heap Dumper to verbose detected object references that are outside the heap
- Updated Heap Dumper to throw a dedicated error when an object reference does not target the beginning of an object (most likely a corrupted heap)
- Updated Heap Dumper to dump .heap.error partial file when a crash occurred during heap processing
- Fixed Heap Dumper crash when processing an object owned by a Feature which type is also owned by the Feature (was working before only when the type is owned by the Kernel)

- Fixed Firmware Linker potential negative offset generation when some sections do not appear in the same order in the ELF file than in their associated LOAD segment
- Fixed Code Coverage Analyzer potential generated empty report (wrong load of classfiles from JAR files)

[7.11.0] - 2019-06-24

Important Notes

- Java assertions execution is now disabled by default. If you experience any runtime trouble when migrating from a previous Architecture, please enable Java assertions execution both on Simulator and on Device (maybe the application code requires Java assertions to be executed).
- Calls to Security Manager are now disabled by default. If you are using the Security Manager, it must be explicitly enabled using the option described below (likely the case when building a Multi-Sandbox Firmware and its associated Virtual Device).
- Front Panel framework is now provided by the Architecture instead of the UI Pack. This allow to build a VEE Port with a Front Panel (splash screen, basic I/O, …), even if it does not provide a MicroUI port. Moreover, the Front Panel framework API has been redesigned and is now distributed using the `ej.tool.frontpanel.framework` module instead of the legacy Eclipse classpath variable.

Known Issues

- SOAR `Internal SOAR error` or `Stacks merging coherence error` thrown when an `if` statement (being removed) is declared at the end of a `try` block:

```
try {
   ...
   if (Constants.getBoolean(XXX)) { // constant resolved to false
      ... // code being removed
   }
} catch (Exception e) {
      ...
}
```

Core Engine

- Added `EDC-1.3` support for daemon threads
- Added `BON` support for ej.bon.Util.newArray(T[],int)
- [Multi/ARMCC5] - Fixed unused undefined symbol that prevent Keil MDK-ARM to link properly

Foundation Libraries

- Updated EDC to version 1.3 (see EDC-1.3 API Changelog)
  - Updated the implementation code for correct Null analysis detection (added assertions, extracted multiple field accesses into a local)
  - Fixed PrintStream.PrintStream(OutputStream, boolean) writer initialization
  - Removed useless String literals in java.lang.Throwable
- Updated UTF-8 decoder to support Unicode code points
- Updated BON to version 1.4 (see BON-1.4 API Changelog)
- Updated TRACE to version 1.1
  - Added ej.trace.Tracer.getGroupID()
  - Added a BON Constant (core.trace.enabled) to remove trace related code when tracing is disabled
- Fixed KF to call the registered Thread.UncaughtExceptionHandler when an exception is thrown by the first Feature thread

Integration

- Added new options for Java assertions execution in category Runtime (core.assertions.sim.enabled and core.assertions.emb.enabled). By default, Java assertions execution is disabled both on Simulator and on Device.
- Updated options categories (options property names left unchanged)
  - Added a new category named Runtime
  - Renamed Target to Device
  - Moved Embed All type names option from Core Engine to Runtime
  - Moved Core Engine under Device
  - Removed category Target > Debug and moved Trace options to Runtime
  - Removed category Debug and moved all sub categories under Simulator
  - Renamed category JDWP to Debug
- Added an option (com.microej.library.edc.securitymanager.enabled) to enable Security Manager runtime checks (disabled by default)

Simulator

- Added a cache to speed-up classfile loading in JARs
- Added EDC-1.3 support for daemon threads
- Added BON-1.4 support for compile-time constants (load of .constants.list resources)
- Added BON-1.4 support for ej.bon.Util.newArray()
- Added Front Panel framework
- Updated error message when reaching Simulator limits

- Removed the Bootstrapping a Smart Software Simulator message when verbose mode in enabled
- Fixed Object.clone() on an immutable object to return a new (mutable) object instead of an immutable one
- Fixed Object.clone() crash when an OutOfMemory occurs
- Fixed potential crash when calling an abstract method (some interfaces of the hierarchy were not taken into account - introduced in version *7.10.0*)
- Fixed OutOfMemory errors even if the heap is not full (resources loaded from Class.getResourceAsStream() and ResourceBuffer creation were taken into account in simulated heap memory - introduced in version *7.10.0*)
- Fixed potential crash when a GC occurs while a ResourceBuffer is opened (introduced in version *7.10.0*)
- Fixed potential debugger hangs when an exception was thrown but not caught in the same method
- [Multi] - Fixed wrong class loading in some cases
- [Multi] - Fixed wrong immutable loading in some cases

SOAR

- Added BON-1.4 support for compile-time constants (load of .constants.list resources)
- Added bytecode removal for Java assertions (when option is disabled)
- Added bytecode removal for if(ej.bon.Constants.getBoolean()) pattern
  - then or else block is removed depending on the boolean condition
  - *WARNING: Current limitation: the "if" statement cannot wrap or be nested in a "try-catch-finally" statement*
- Added *Option(checkbox): Group Methods* for grouping all the methods by type in a single ELF section
- Added an error message when microejapp.o cannot be generated because the maximum number of ELF sections (65536) is reached

Tools

- Updated License Manager (Production) to debug dongle recognition issues from command line (see *Check Activation*).
- Updated License Manager (Production) to support dongle recognition on macOS 10.14 (Mojave)
- Fixed ELF To Map to produce correct sizes from an executable generated by IAR Embedded Workbench for ARM
- Fixed Firmware Linker .ARM.exidx section generation (missing section link content)
- Updated deployment files policy for VEE Ports in Workspace, in order to be more flexible depending on the C project layout. This also allows to deploy to the same C project different Applications built with different VEE Ports

  - VEE Port configuration: in `bsp/bsp.properties`, a new option `output.dir` indicates where the files are deployed by default
    * Application (`microejapp.o`) and Runtime library (`microejruntime.a`) are deployed to `${output.dir}/lib`. Architecture header files (`*.h`) are deployed to `${output.dir}/inc/`
    * When this option is not set, the legacy behavior is left unchanged (`project.file` option in collaboration with `augmentCProject` scripts)
  - Launch configuration: `Device > Deploy` options allow to override the default VEE Port configuration in order to deploy each file into a separate folder.
- Fixed wrong ELF file generation when a section included in a LOAD segment was generated before one of the sections included in a LOAD segment declared before the first one (integrated in ELF Utils and Firmware Linker)
- Fixed wrong ELF file generation when a section included in a LOAD segment had an address which was outside its LOAD segment virtual address space (integrated in ELF Utils and Firmware Linker)

[7.10.1] - 2019-04-03

Simulator

- Fixed Object.getClass() may return a Class instance owned by a Feature for type owned by the Kernel

[7.10.0] - 2019-03-29

Core Engine

- Added internal memories checks at startup: heaps and statics memories are not allowed to overlap with `LLBSP_IMPL_isInReadOnlyMemory()`
- [Multi] - Updated Feature Kill implementation to prepare future RAM Control (fully managed by Core Engine)
- [Multi] - Updated implementation of ej.kf.Kernel: all APIs taking a Feature argument now will throw a java.lang.IllegalStateException when the Feature is not started

Foundation Libraries

- Updated `KF` library in sync with Core Engine Kill related fixes and Simulator with Kernel & Features semantic
- Updated `BON` library on Simulator (now uses the same implementation than the one used by the Core Engine)

Integration

- Added generation of architecture.properties file when building a VEE Port. (Used by SDK 5.x when manipulating VEE Ports & Virtual Devices)

Simulator

- Added Embed all types names option for Simulation
- Added memory size simulation for Managed Heap and Immortals Heap (Enabling Use target characteristics option is no more required)
- Added Kernel & Features semantic, as defined in the KF-1.4 specification
  - Fully implemented:
    - Ownership for types, object and thread execution context
    - Kernel mode
    - Context Local Static Field References
  - Partially implemented:
    - Kernel API (Type grained only)
    - Shared Interfaces are binded using direct reference links (no Proxy execution)
    - Feature.stop() does not perform the safe kill. The application cannot be stopped unless it has correctly removed all its shared references.
  - Not implemented:
    - Dynamic Feature installation from Kernel.install(java.io.InputStream)
    - Execution Rules Runtime checks

Tools

- Updated Memory Map Scripts (Bluetooth, MWT, NLS, Rcommand and AllJoyn libraries)
- Fixed Kernel Packager internal limits error when the ELF executable does not contains a .debug.soar section
- Fixed wrong ELF file generation when segment file size is different than the mem size (integrated in ELF Utils and Firmware Linker)
- Fixed Simulator COM port mapping default value (set to disabled instead of UART<->UART in order to avoid an error when launch configuration is just created)
- Fix ELF To Map: the total sections size were not equal to the segments size

[7.9.1] - 2019-01-08

Tools

- Fixed ELF objcopy generation when ELF executable file contains `0` size segments
- Fixed `Stack Trace Reader` error when ELF executable file contains relocation sections

[7.9.0] - 2018-09-20

Core Engine

- Fixed OutOfMemoryError thrown when allocating an object of the size of free memory in Immortals Heap

SOAR

- Optimized SOAR processing (up to 50% faster on applications with tens of classpath entries)

[7.8.0] - 2018-08-01

Tools

- [ARMCC5] - Updated *SOAR Debug Infos Post Linker* tool to generate the full ELF executable file

[7.7.0] - 2018-07-19

Core Engine

- Added a permanent hook `LLMJVM_on_Runtime_gc_done` called after an explicit java.lang.Runtime.gc()
- Updated internal heap header for memory dump

SOAR

- Added check for the maximum number of allowed concrete types (avoids a Core Engine link error)

Tools

- Added Heap Dumper tool

[7.6.0] - 2018-06-29

Foundation Libraries

- [Multi] - Updated BON library: a Timer owned by the Kernel can execute a TimerTask owned by a Feature

[7.5.0] - 2018-06-15

Internal Release - COTS Architecture left unchanged.

[7.4.0] - 2018-06-13

Core Engine

- Removed partial support of ej.bon.Util.throwExceptionInThread() (deprecated)
- [Multi/Linux] - Updated default configuration to always embed method names
- [Multi/Cortex-M] - Optimized KF checks execution for array & field accesses

Foundation Libraries

- Updated ej.bon.Timer to schedule ej.bon.TimerTask owned by multiple Features

Simulator

- Fixed implementation of Class.getResourceAsStream() to throw an IOException when the stream is closed

SOAR

- [GCC] - Fixed microejapp.o link with GCC 6.3

Tools

- Added a retry mechanism in the Testsuite Engine
- Added a message to suggest increasing the JVM heap when an OutOfMemoryError occurs in the Firmware Linker tool
- Fixed generation of LL header files for all cross compilation toolchains (file separator for included paths is /)
- [Cortex-A/ARMCC5] - Fixed SNI convention call issue

- [ESP32,RX] - Fixed Firmware Linker tool internal limit

[7.3.0] - 2018-03-07

Simulator

- Added an option for the IDE to customize the mockups classpath
- Fixed Deadlock in Shielded Plug remote client when interrupting a thread that waits for block modification

[7.2.0] - 2018-03-02

Core Engine

- [Multi] - Enabled quantum counter computation only when Feature quota is set
- [Cortex-M/IAR] - Updated compilation flags to -Oh

Simulator

- Added a hook in the mockup that is automatically called during the HIL engine startup
- Added dump of loaded classes when verbose option is enabled
- Fixed Runtime.freeMemory() call freeze when Emb Characteristics option is enabled
- Fixed ShieldedPlug server error after interrupting a thread that is waiting for a database block
- Fixed crash Access to a wrong reference in some cases
- Fixed java.lang.NullPointerException when interrupting a thread that has not been started
- Fixed crash when closing an HIL engine connection in some cases
- [Multi] - Fixed KF & Watchdog library link when Emb Characteristics option is enabled
- [Multi] - Fixed XML Parsing error when Emb Characteristics option is enabled

[7.1.2] - 2018-02-02

SOAR

- Fixed SNI library was added in the classpath in some cases

[maintenance/6.18.0] - 2017-12-15

Core Engine

- [Multi] - Enabled quantum counter computation only when Feature quota is set
- [Cortex-M/IAR] - Updated compilation flags to -Oh

Simulator

- Fixed Runtime.freeMemory() call freeze when Emb Characteristics option is enabled
- [Multi] - Fixed KF & Watchdog library link when Emb Characteristics option is enabled
- [Multi] - Fixed XML Parsing error when Emb Characteristics option is enabled

Tools

- Updated Kernel API Generator tool with classes filtering

[7.1.1] - 2017-12-08

Tools

- [Multi/RX] - Fixed Firmware Linker tool

[7.1.0] - 2017-12-08

Core Engine

- [Multi/RX] - Added KF support

Integration

- Fixed SNI-1.3 library name

SOAR

- [RX] - Added support for ELF symbol prefix __

Tools

- Updated `Kernel API generator` tool with classes filtering

[7.0.0] - 2017-11-07

Core Engine

- Added SNI-1.3 support
- `SNI_suspendCurrentJavaThread()` is not interruptible via Thread.interrupt() anymore

Foundation Libraries

- Updated to `SNI-1.3`

[6.17.2] - 2017-10-26

Simulator

- Fixed deadlock during bootstrap in some cases

[6.17.1] - 2017-10-25

Core Engine

- Fixed conversion of `-0.0` into a positive value

[6.17.0] - 2017-10-10

Tools

- Updated Memory Map Scripts for TRACE library

[6.16.0] - 2017-09-27

Core Engine

- Fixed External Resource Loader link error (introduced in version *6.13.0*)

[6.15.0] - 2017-09-12

Core Engine

- Added a new option to configure the maximum number of monitors that can be owned per thread (8 per thread by default, as it was fixed before)

Foundation Libraries

- Fixed ECOM-COMM internal heap calibration

SOAR

- Added log of the class loading cause

[6.14.2] - 2017-08-24

Tools

- Fixed `Firmware Linker` tool script (load `activity.xml` from the wrong folder)
- Fixed load of symbol `__java_Ljava_io_EOFException` that can be required by some linkers even if this symbol is not touched

[6.14.1] - 2017-08-02

Simulator

- Fixed Device Mockup too long initialization that may block the Front Panel Mockup

Foundation Libraries

- Fixed BON `.types.list` potential conflicts with KF

Tools

- Modified `Firmware Linker` internal scripts structure for new Virtual Devices tools

[6.13.0] - 2017-07-21

Core Engine

- Added support for ej.bon.ResourceBuffer

Foundation Libraries

- Updated to BON-1.3

SOAR

- Added support for *.resourcesext.list (resources excluded from the firmware)

Tools

- Added BON Resource Buffer generator

[6.12.0] - 2017-07-07

Core Engine

- Added a trace when IllegalMonitorStateException is thrown on a monitorexit

Tools

- Added property skip.mergeLibraries for Platform Builder.
- Updated the serial PC connector to JSSC 2.8.0 .

Simulator

- Fixed unexpexted java.lang.NullPointerException in some cases

[6.11.0] - 2017-06-13

Integration

- Fixed useless watchdog library copied in root folder

[6.11.0-beta1] - 2017-06-02

Core Engine

- Added an option to enable execution traces
- Added Low Level API LLMJVM_MONITOR_impl.h
- Added Low Level API LLTRACE_impl.h

Foundation Libraries

- Added TRACE-1.0

[6.10.0] - 2017-06-02

Core Engine

- Optimized java.lang.Runtime.gc() (removed useless heap compaction in some cases)

[6.9.2] - 2017-06-02

Integration

- Fixed missing properties in release.properties (introduced in version *v6.9.1*)
- Fixed artifacts build dependencies to private dependencies

[6.9.1] - 2017-05-29

SOAR

- [Multi] - Fixed selected methods list in report generation (removed Kernel related method)

[6.9.0] - 2017-03-15

Base version, included into SDK 4.1.

Release Notes

Foundation Libraries

The following table describes Foundation Libraries API versions implemented in MicroEJ Architectures.

Table 45: Architecture API Implementation

Architecture Range	EDC	BON	KF	SNI	SP	Trace	Device	ECOM-COMM
[8.0.0-8.3.0]	1.3	1.4	1.7	1.4	2.0	1.1	N/A[1]	N/A[2]
[7.17.0-7.20.1]	1.3	1.4	1.6	1.4	2.0	1.1	1.0	1.1
[7.13.0-7.16.0]	1.3	1.4	1.5	1.4	2.0	1.1	1.0	1.1
[7.11.0-7.12.0]	1.3	1.4	1.5	1.3	2.0	1.1	1.0	1.1
[7.10.0-7.10.1]	1.2	1.3	1.5	1.3	2.0	1.0	1.0	1.1
[7.0.0-7.9.1]	1.2	1.3	1.4	1.3	2.0	1.0	1.0	1.1
[6.13.0-6.18.0]	1.2	1.3	1.4	1.2	2.0	1.0	1.0	1.1
[6.11.0-6.12.0]	1.2	1.2	1.4	1.2	2.0	1.0	1.0	1.1
[6.9.0-6.10.0]	1.2	1.2	1.4	1.2	2.0	N/A	1.0	1.1

Default Application

As of *Architecture 8.1.0*, a default pre-built application (microejapp.o) is provided with the VEE Port. It simplifies the early-stage integration into a BSP project by eliminating the need to create and build an Application project within the SDK, thereby removing the requirement to obtain a valid SDK license.

The default pre-built application is available at [VEE_PORT_DIR]/defaultApp/microejapp.o . It prints a splash with *Architecture Characteristics* on the standard output:

```
MicroEJ START

,--.   ,--.,--.                  ,----.     ,--. ,--.   ,--.,----.,----.
|   `.'   |`--' ,--.,--.--. ,--. |  .--'     |  | \  `.'  / |  .--'|  .--'
|  |'.'|  |,--.| .--'|  .--'| .-. ||  `--, ,--. |  |   \    /  |  `--, |  `--,
|  |   |  ||  |\ `--.|  |   ' '-' '|  `--.|  '-'  /    \  /   |  `--.|  `--.
`--'   `--'`--' `--'`--'`--'    `--' `----' `----'      `--'    `----'`----'


                          **********
                        **********
                      **********
                    **********
                    **********+=--::::--=++**********
                  *******++:...          ..:=+*******
                ******+:..                 ...+******
                *****+:.                      .:+*****
                ****+....--.               .:-:  .=****
                ***+....:****-.            .:****=...=****
              ****-...=****+.              -****+....:****
              ****:...-****=.             :****+...:+***
              ****:....-++-               :++=....:+***
                ***+........              .......=****
              ******=.........          .........-******
              *******+.......++-:.....:-=+:.....=*******
              *********=.......:=++++++=:.......-*********
            ************+-...................:+************
          ******   ***********+=-:.....:-=++**********   ******
          *****    #####################################    *****
            ***    #################################      ***
```

(continues on next page)

[1] See Migrate Device Module.
[2] See Migrate ECOM-COMM Module.

(continued from previous page)

```
          *****  ******************************  *****
        ********  ##*************************** ********
        ****  ***  ###*********************##  ***  ****
        ***   *    ###******************###    *   ***
         **         #####***********#####         **
                    ##################
                     ################
                       ##############
                          ######
                    ::::::::::::::::::::::::::::
                     ::::::::::::::::::::::::
```

You successfully linked your first Application with the following Architecture characteristics:
- Name: atsauce6
- Version: 8.1.0 (20231220-1011)
- Usage: Production
- Core Engine Capability: Multi-Sandbox
- Instruction Set Architecture: x86
- Compilation Toolchain: GNU GCC (GNUvX_X86Linux)

MicroEJ END (exit code = 0)

3.6.3 MicroEJ Packs

Overview

On top of a MicroEJ Architecture can be imported MicroEJ Packs which provide additional features such as:

- *Serial Communications*,
- *Graphical User Interface*,
- *Networking*,
- *Bluetooth*,
- *Audio*,
- *MicroAI*,
- *File System*,
- etc.

Each MicroEJ Pack is optional and can be selected on demand in the VEE Port configuration. Refer to *SDK 6* or *SDK 5* documentation to learn how to do it.

Naming Convention

MicroEJ Packs are distributed in two packages:

- MicroEJ Architecture Specific Pack under the com/microej/architecture/* organization.
- MicroEJ Generic Pack under the com/microej/pack/* organization.

See the Pack Import documentation for *SDK 6* or *SDK 5* for usage.

Architecture Specific Pack

MicroEJ Architecture Specific Packs contain compiled libraries archives and are thus dependent on the MicroEJ Architecture and toolchain used in the MicroEJ Platform.

MicroEJ Architecture Specific Packs files ends with the .xpfp extension and are classified using the following naming convention:

```
com/microej/architecture/[ISA]/
↪[TOOLCHAIN]/[UID]-[NAME]-pack/[VERSION]/[UID]-[NAME]-pack-[VERSION].xpfp
```

- ISA : instruction set architecture (e.g. CM4 for Arm® Cortex®-M4, ESP32 for Espressif ESP32, ...).
- TOOLCHAIN : C compilation toolchain (e.g. CM4hardfp_GCC48).
- UID : Architecture unique ID (e.g. flopi4G25).
- NAME : pack name (e.g. ui).
- VERSION : pack version (e.g. 13.0.4).

For example, MicroEJ Architecture Specific Pack UI versions for Arm® Cortex®-M4 microcontrollers compiled with GNU CC toolchain are available at https://repository.microej.com/modules/com/microej/architecture/CM4/CM4hardfp_GCC48/flopi4G25-ui-pack/.

Generic Pack

MicroEJ Generic Packs can be imported on top of any MicroEJ Architecture.

They are classified using the following naming convention:

```
com/microej/pack/[NAME]/[NAME]-pack/[VERSION]/
```

- NAME : pack name (e.g. bluetooth).
- VERSION : pack version (e.g. 2.1.0).

For example, MicroEJ Generic Pack Bluetooth versions are available at https://repository.microej.com/modules/com/microej/pack/bluetooth/bluetooth-pack/.

Legacy Generic Pack

Legacy MicroEJ Generic Packs files end with the .xpfp extension. These Packs contain one or more VEE Port modules. They are classified using the following naming convention:

```
com/microej/pack/[NAME]/[VERSION]/[NAME]-[VERSION].xpfp
```

- NAME : pack name (e.g. net).
- VERSION : pack version (e.g. 9.2.3).

For example, the Legacy MicroEJ Generic Pack NET version 9.2.3 is available at https://repository.microej.com/modules/com/microej/pack/net/9.2.3/net-9.2.3.xpfp.

3.6.4 BSP Connection

Principle

Using a VEE Port, the user can compile an Application on that VEE Port. The result of this compilation is a microejapp.o file.

This file has to be linked with the VEE Port runtime file (microejruntime.a) and a third-party C project, called the Board Support Package (BSP), to obtain the final binary file (the Executable). For more information, please consult the *MicroEJ build process overview*.

The BSP connection can be configured by defining 4 folders where the following files are located:

- Application file (microejapp.o).
- VEE Port runtime file (microejruntime.a , also available in the VEE Port lib folder).
- VEE Port header files (*.h , also available in the VEE Port include folder).
- BSP project *build script* file (build.bat or build.sh).

Once the Application file (microejapp.o) is built, the files are then copied to these locations and the build.bat or build.sh file is executed to produce the Executable (application.out).

Note: The final build stage to produce the Executable can be done outside of the SDK, and thus the BSP connection configuration is optional.

BSP connection configuration is only required in the following cases:

- Use the SDK to produce the Executable of a Mono-Sandbox Application (recommended).
- Use the SDK to run a *VEE Port Testsuite* on device.
- Build a the Executable of a Multi-Sandbox Application.

MicroEJ provides a flexible way to configure the BSP connection to target any kind of projects, teams organizations and company build flows. To achieve this, the BSP connection can be configured either at VEE Port level or at Application level (or a mix of both).

The 3 most common integration cases are:

- Case 1: No BSP connection

The VEE Port does not know the BSP at all.

BSP connection can be configured when building the Application (absolute locations).

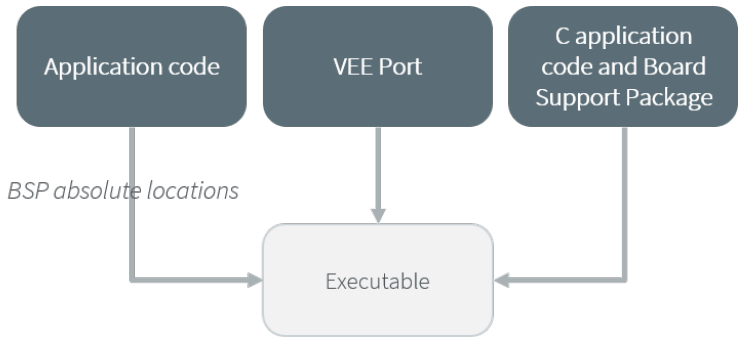


Fig. 175: VEE Port with no BSP connection

This case is recommended when:

- the Executable is built outside the SDK.
- the same VEE Port is intended to be reused on multiple BSP projects which do not share the same structure.

- Case 2: Partial BSP connection

  The VEE Port knows how the BSP is structured.

  BSP connection is configured when building the VEE Port (relative locations within the BSP), and the BSP root location is configured when building the Application (absolute directory).

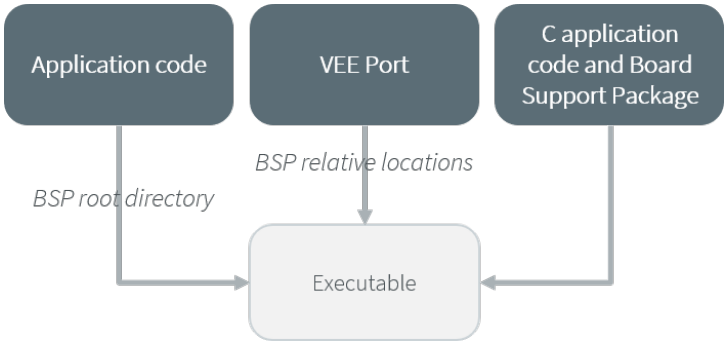


Fig. 176: VEE Port with partial BSP connection

This case is recommended when:

- the VEE Port is used to build one Application on top of one BSP.
- the Application and BSP are slightly coupled, thus making a change in the BSP just requires to build the Executable again.

- Case 3: Full BSP connection

  The VEE Port includes the BSP.

  BSP connection is configured when building the VEE Port (relative locations within the BSP), as well as the BSP root location (absolute directory). No BSP connection configuration is required when building the Application.

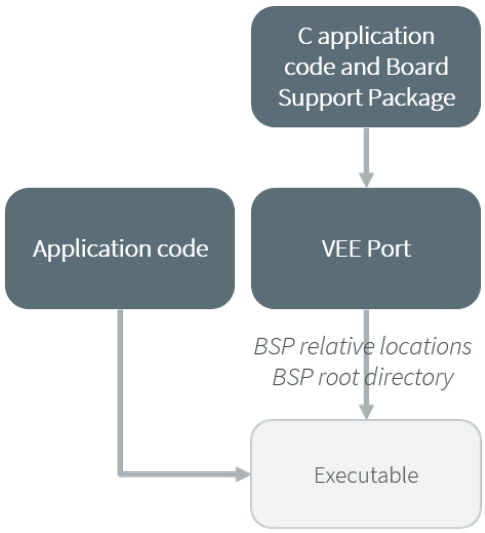


Fig. 177: VEE Port with full BSP connection

This case is recommended when:

- the VEE Port is used to build various Applications.
- the VEE Port is validated using Testsuites.
- the VEE Port and BSP are delivered as a single standalone module (same versioning), perhaps subcontracted to a team or a company outside the Application project(s).

Options

BSP connection options can be specified as VEE Port options or as Application options or a mix of both.

The following table describes the VEE Port options.

SDK 6

SDK 5

In SDK 6, the options must be set in the `configuration.properties` file of the VEE Port project , prefixed by `bsp.` .

In SDK 5, the options must be set in the `bsp/bsp.properties` file of the VEE Port configuration project.

Table 46: VEE Port Options for BSP Connection

Option Name	Description	Example
microejapp.relative.dir	The path relative to BSP root.dir where to deploy the Application file (microejapp.o).	MicroEJ/lib
microejlib.relative.dir	The path relative to BSP root.dir where to deploy the VEE Port runtime file (microejruntime.a).	MicroEJ/lib
microejinc.relative.dir	The path relative to BSP root.dir where to deploy the VEE Port header files (*.h).	MicroEJ/inc
microejscript.relative.dir	The path relative to BSP root.dir where to execute the BSP build script file (build.bat or build.sh).	Project/MicroEJ
root.dir	The 3rd-party BSP project absolute directory, to be included to the VEE Port.	c:\\Users\\user\\mybsp on Windows systems or /home/user/bsp on Unix systems.

The following table describes the Application options, which can be set as regular *Application Options*.

Table 47: Application Options for BSP Connection

Option Name	Description
deploy.bsp.microejapp	Deploy the Application file (microejapp.o) to the location defined by the VEE Port (defaults to true when VEE Port option microejapp.relative.dir is set).
deploy.bsp.microejlib	Deploy the VEE Port runtime file (microejruntime.a) to the location defined by the VEE Port (defaults to true when VEE Port option microejlib.relative.dir is set).
deploy.bsp.microejinc	Deploy the VEE Port header files (*.h) to the location defined by the VEE Port (defaults to true when VEE Port option microejinc.relative.dir is set).
deploy.bsp.microejscript	Execute the BSP build script file (build.bat or build.sh) at the location specified by the VEE Port. (defaults to false and requires microejscript.relative.dir VEE Port option to be set).
deploy.bsp.root.dir	The 3rd-party BSP project absolute directory. This option is required if at least one the 4 options described above is set to true and the VEE Port does not include the BSP.
deploy.dir.microejapp	Absolute path to the directory where to deploy the Application file (microejapp.o). An empty value means no deployment.
deploy.dir.microejlib	Absolute path to the directory where to deploy the VEE Port runtime file (microejruntime.a) to this absolute directory. An empty value means no deployment.
deploy.dir.microejinc	Absolute path to the directory where to deploy the VEE Port header files (*.h) to this absolute directory. An empty value means no deployment.
deploy.dir.microejscript	Absolute path to the directory that contains the BSP build script file (build.bat or build.sh). An empty value means no build script execution.

Note: It is also possible to configure the BSP root directory by setting the *build option* `toolchain.dir`, instead of the application option `deploy.bsp.root.dir`. This allows to build the Executable by specifying both the VEE Port (using the `target.platform.dir` option) and the BSP at build level, without having to modify the application options files.

For each *VEE Port BSP connection case*, here is a summary of the options to set:

- No BSP connection, Executable built outside the SDK

```
VEE Port Options:
  [NONE]

Application Options:
  [NONE]
```

- No BSP connection, Executable built using the SDK

```
VEE Port Options:
  [NONE]

Application Options:
  deploy.dir.microejapp=[absolute_path]
  deploy.dir.microejlib=[absolute_path]
  deploy.dir.microejinc=[absolute_path]
  deploy.dir.microejscript=[absolute_path]
```

- Partial BSP connection, Executable built outside the SDK

  SDK 6

  SDK 5

```
VEE Port Options:
  bsp.microejapp.relative.dir=[relative_path]
  bsp.microejlib.relative.dir=[relative_path]
  bsp.microejinc.relative.dir=[relative_path]

Application Options:
  deploy.bsp.root.dir=[absolute_path]
```

```
VEE Port Options:
  microejapp.relative.dir=[relative_path]
  microejlib.relative.dir=[relative_path]
  microejinc.relative.dir=[relative_path]

Application Options:
  deploy.bsp.root.dir=[absolute_path]
```

- Partial BSP connection, Executable built using the SDK

  SDK 6

  SDK 5

```
VEE Port Options:
  bsp.microejapp.relative.dir=[relative_path]
  bsp.microejlib.relative.dir=[relative_path]
  bsp.microejinc.relative.dir=[relative_path]
```

(continues on next page)

(continued from previous page)

```
  bsp.microejscript.relative.dir=[relative_path]

Application Options:
  deploy.bsp.root.dir=[absolute_path]
  deploy.bsp.microejscript=true
```

```
VEE Port Options:
  microejapp.relative.dir=[relative_path]
  microejlib.relative.dir=[relative_path]
  microejinc.relative.dir=[relative_path]
  microejscript.relative.dir=[relative_path]

Application Options:
  deploy.bsp.root.dir=[absolute_path]
  deploy.bsp.microejscript=true
```

- Full BSP connection, Executable built using the SDK

  SDK 6

  SDK 5

```
VEE Port Options:
  bsp.microejapp.relative.dir=[relative_path]
  bsp.microejlib.relative.dir=[relative_path]
  bsp.microejinc.relative.dir=[relative_path]
  bsp.microejscript.relative.dir=[relative_path]
  bsp.root.dir=[absolute_path]

Application Options:
  deploy.bsp.microejscript=true
```

```
VEE Port Options:
  microejapp.relative.dir=[relative_path]
  microejlib.relative.dir=[relative_path]
  microejinc.relative.dir=[relative_path]
  microejscript.relative.dir=[relative_path]
  root.dir=[absolute_path]

Application Options:
  deploy.bsp.microejscript=true
```

Build Script File

The BSP build script file is used to invoke the third-party C toolchain (compiler and linker) to produce the Executable (application.out).

The build script must comply with the following specification:

- On Windows operating system, it is a Windows batch file named build.bat .
- On macOS or Linux operating systems, it is a shell script named build.sh , with execution permission enabled.
- On error, the script must end with a non zero exit code.
- On success

- The Executable must be copied to a file named `application.out` in the directory from where the script has been executed.
- The script must end with zero exit code.

  Many build script templates are available for most commonly used C toolchains in the VEE Port Qualification Tools repository.

  The build script can also be launched before the VEE Port publication, see the VEE Port Publication documentation for *SDK 6* or *SDK 5* for more details.

Note: The Executable must be an ELF executable file. On Unix, the command `file(1)` can be use to check the format of a file. For example:

```
~$ file application.out
ELF 32-bit LSB executable
```

Run Script File

This script is required only for VEE Ports intended to run a *VEE Port Testsuite* on device.

The BSP run script is used to invoke a third-party tool to upload and start the Executable on device.

The run script must comply with the following specification:

- On Windows operating system, it is a Windows batch file named `run.bat` .
- On macOS or Linux operating systems, it is a shell script named `run.sh` , with execution permission enabled.
- The Executable filename is passed as first script parameter if there is one, otherwise it is the `application.out` file located in the directory from where the script has been executed.
- On error, the script must end with a non zero exit code.
- On success:
  - The Executable (`application.out`) has been uploaded and started on the device
  - The script must end with zero exit code.

  The run script can optionally redirect execution traces. If it does not implement execution traces redirection, the testsuite must be configured with the following *Standalone Application Options* in order to take its input from a TCP/IP socket server, such as Serial to Socket Transmitter (see documentation for *SDK 6* or *SDK 5*).

```
testsuite.trace.ip=localhost
testsuite.trace.port=5555
```

3.6.5 VEE Port Qualification

Introduction

A VEE Port integrates one or more Foundation Libraries with their respective Abstraction Layers.

VEE Port Qualification is the process of validating the conformance of the Abstraction Layer that implements the *Low Level APIs* of a Foundation Library.

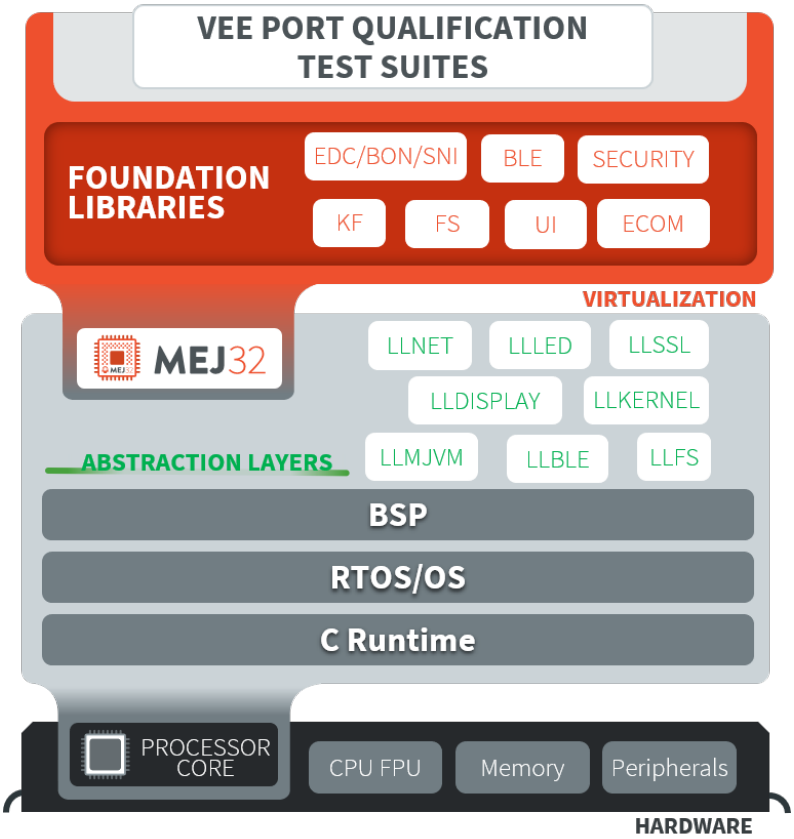


Fig. 178: VEE Port Qualification Overwiew

For each Low Level API, an Abstraction Layer implementation is required. The validation of the Abstraction Layer implementation is performed by running tests at two-levels:

- In C, by calling Low Level APIs (usually manually).
- In Java code, by calling Foundation Library APIs (usually automatically using *VEE Port Testsuite*).

The following figure depicts an example for the FS Pack:

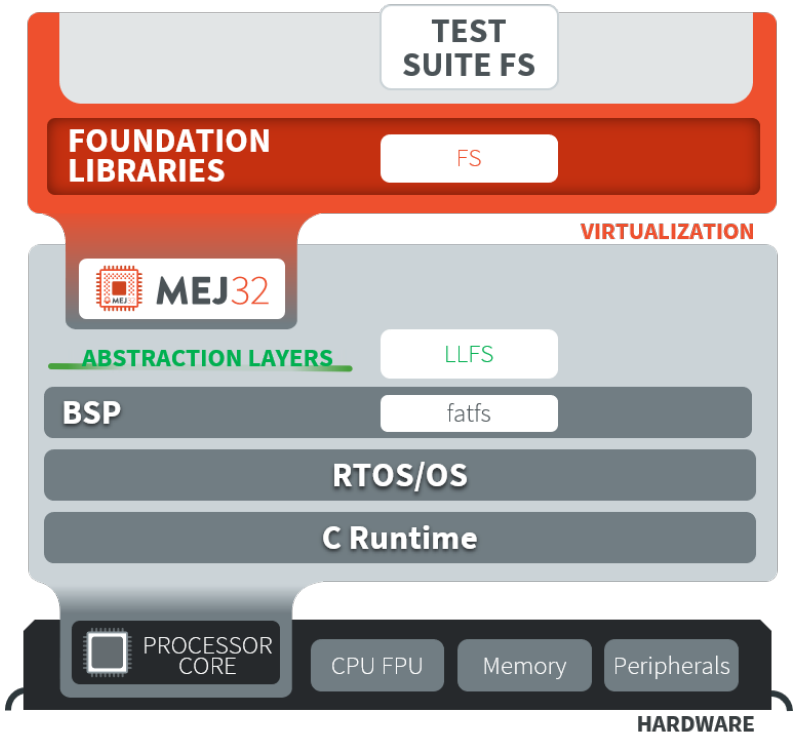


Fig. 179: VEE Port Qualification Example for FS Pack

MicroEJ provides a set of tools and pre-defined projects aimed at simplifying the steps for validating VEE Ports in the form of the VEE Port Qualification Tools (PQT):

SDK 6

SDK 5

In SDK 6, it is located at:

- the `validation` folder of the VEE Port Project Template for the Java Testsuites.
- the AbstractionLayer Tests repository for the C tests.

In SDK 5, it is located at the VEE Port Qualification Tools (PQT) repository.

VEE Port Qualification Tools Overview

The VEE Port Qualification Tools provide the following components:

- Build and Run Scripts examples:
  - Used to generate and deploy an Executable on a device by invoking a third-party toolchain for the BSP.
  - Added when integrating the BSP to the VEE Port (see *Build Script File* and *Run Script File* or check the training *Create MicroEJ Platform Build and Run Scripts*).
- C and Java Testsuites:
  - Used to validate the Low Level APIs implementations.

- Validated during the BSP development and whenever an Abstraction Layer implementation is added or changed (see *VEE Port Testsuite* or check the section *Configure and Run the Testsuite*).

- **(Only for SDK 5)** Platform Configuration Additions (PCA):
  - Used to:
    - Manage Architecture, Packs dependencies and the VEE Port build with the MicroEJ Module Manager.
    - Configure the BSP connection to call the build and run scripts.
  - Added when creating a VEE Port (see *Create a VEE Port* or check the training *Create a MicroEJ Firmware From Scratch*).

    Please refer to the VEE Port Qualification Tools README in SDK 5 or in SDK 6 for more details and the location of the components.

VEE Port Testsuite

The purpose of a VEE Port Testsuite is to validate the Abstraction Layer that implements the *Low Level APIs* of a Foundation Libraries by automatically running Java tests on the device.

The *MicroEJ Testsuite Engine* is used for building, running a Testsuite, and providing a report.

A Testsuite contains one or more tests. For each test, the Testsuite Engine will:

1. Build an Executable for the test.
2. Run the Executable onto the device.
3. Retrieve the execution traces.
4. Analyze the traces to determine whether the test has `PASSED` or `FAILED`.
5. Append the result to the Test Report.
6. Repeat until all tests of the Testsuite have been executed.

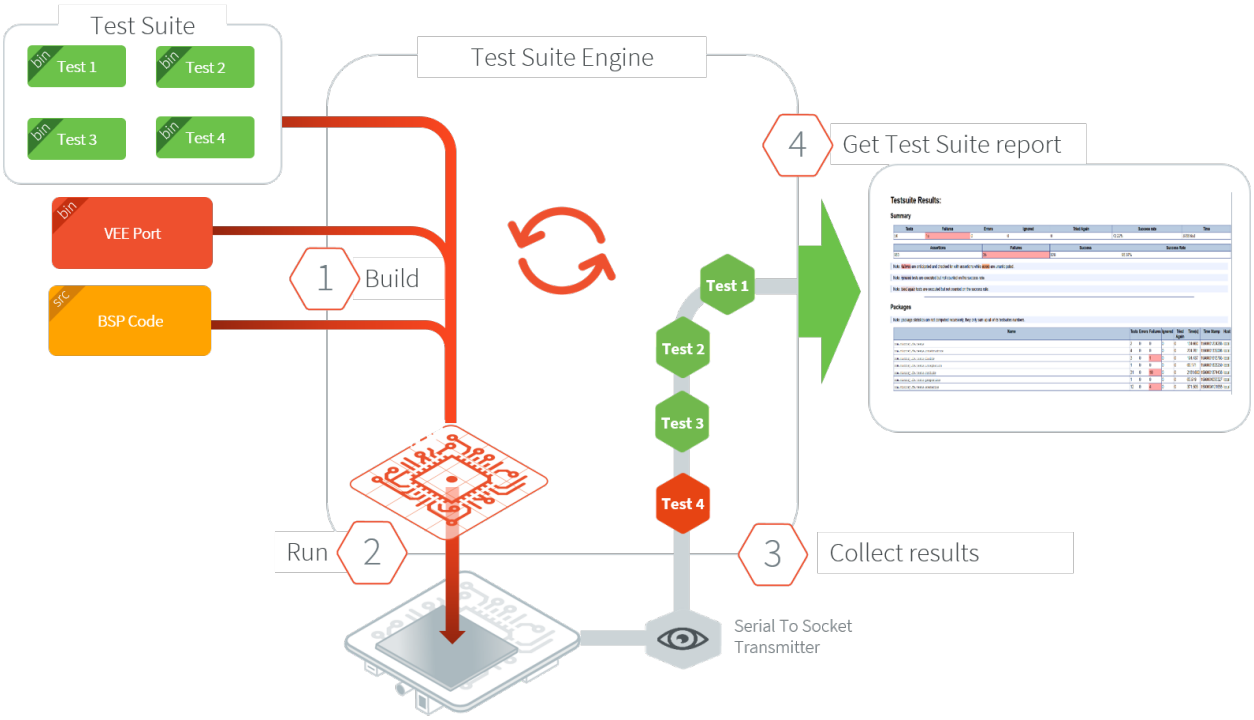


Fig. 180: VEE Port Testsuite on Device Overview

Testsuite Versioning

Foundation Libraries are integrated in a VEE Port using *Packs*. Use the Testsuite version compliant with the API version provided by the Foundation Library to validate the Abstraction Layer implementation. For example, the Testsuite FS module 3.0.3 should be used to validate the Abstraction Layer implementation of the *Low Level API FS* provided by the FS Pack 5.1.2.

Note: A Pack can provide several Foundation Libraries.

Core Engine

Table 48: Core Engine Validation

Architecture	Testsuite
7.0.0 or higher	Core Engine Testsuite

UI Pack

Table 49: MicroUI API Implementation and Validation

UI Pack	MicroUI API	Drawing API	Java Testsuite
14.5.2	3.6.1	1.0.6	14.5.2
14.5.1	3.6.1	1.0.6	14.5.1
14.4.2	3.6.0	1.0.5	14.4.2
14.4.1	3.6.0	1.0.5	14.4.1
14.3.3	3.6.0	1.0.5	14.3.3
14.3.2	3.6.0	1.0.5	14.3.2
14.3.0	3.6.0	1.0.5	14.3.0
14.2.0	3.6.0	1.0.5	14.2.0
14.1.1	3.6.0	1.0.5	14.1.1
[14.0.1-14.0.3[	3.5.0	1.0.5	1.8.1
14.0.0	3.5.0	1.0.5	1.7.0
13.7.2	3.5.0	1.0.5	1.6.0
13.7.0	3.4.0	1.0.4	1.4.0
13.6.2	3.3.0	1.0.4	1.3.1
[13.6.0-13.6.1[	3.3.0	1.0.4	1.3.0
[13.4.1-13.5.1[	[3.1.1-3.2.0]	1.0.4	1.2.0
[13.2.0-13.3.1[	3.1.1	1.0.4	1.0.0

VG Pack

Table 50: MicroVG API Implementation and Validation

VG Pack	MicroVG API	Java Testsuite
1.8.1	1.5.2	4.3.2
1.7.2	1.5.2	4.2.0
1.7.1	1.5.1	4.1.0
1.7.0	1.5.0	4.0.0
1.6.0	1.4.0	3.0.0
1.5.1	1.4.0	2.7.0
1.5.0	1.4.0	2.6.0
1.4.2	1.4.0	2.5.0
1.4.1	1.4.0	2.4.0
1.4.0	1.4.0	2.3.0
1.3.0	1.3.0	2.2.0

FS Pack

Table 51: FS API Implementation and Validation

FS Pack	FS API	Java Testsuite
[6.0.0-6.1.0[	2.1.1	3.0.8
[5.1.2-5.2.0[	2.0.6	3.0.3
[4.0.0-4.1.0[	2.0.6	On demand[1]

[1] Testsuite available on demand, please contact MicroEJ Support.

GNSS Pack

Table 52: GNSS API Implementation and Validation

GNSS Pack	GNSS API	Java Testsuite
2.0.0	2.0.0	2.0.0

Bluetooth Pack

Table 53: Bluetooth API Implementation and Validation

Bluetooth Pack	Bluetooth API	Java Testsuite
[2.3.0-2.5.0[	2.2.2	2.1.0
[2.1.0-2.3.0[	2.1.1	2.0.1
[2.0.0-2.1.0[	2.0.0	2.0.1

NET Pack

Table 54: NET, SSL and SECURITY APIs Implementations and Validations

NET Pack	NET API	SSL API	SECURITY API	NET Java Testsuite	SSL Java Testsuite	SECURITY Java Testsuite
[8.1.2-8.2.0]	1.1.0	2.1.0	N/A	3.4.0 (On demand[Page 849, 1])	3.0.1 (On demand[Page 849, 1])	N/A
9.0.0	1.1.0	2.2.0	1.3.1	3.4.0 (On demand[Page 849, 1])	3.1.4 (On demand[Page 849, 1])	1.1.0 (On demand[Page 849, 1])
[9.0.1-9.4.1]	1.1.1	2.2.0	1.3.1	3.5.2 (On demand[Page 849, 1])	3.1.4 (On demand[Page 849, 1])	1.1.0 (On demand[Page 849, 1])
[10.0.0-10.5.0]	1.1.4	2.2.3	1.4.2	4.1.2	4.0.1	1.3.1
[11.0.1-11.0.2]	1.1.4	2.2.3	1.4.2	4.1.2	4.0.2	1.3.1
11.1.0	1.1.4	2.2.3	1.7.0	4.1.2	4.0.2	1.7.0
11.2.0	1.1.4	2.2.3	1.7.0	4.1.2	4.0.2	1.8.0

Audio Pack

Table 55: Audio API Implementation and Validation

Audio Pack	Audio API	Java Testsuite
[1.0.0-1.1.0[	1.0.0	1.0.0

MicroAI Pack

Table 56: MicroAI Library and Validation

MicroAI Library	Java Testsuite
2.1.0	2.1.0 (On demand[Page 849, 1])

EVENT QUEUE Pack

Table 57: EVENT QUEUE API Implementation and Validation

EVENT QUEUE Pack	EVENT QUEUE API	Java Testsuite
2.0.1	2.0.0	2.0.0

3.6.6 Core Engine

The Core Engine is the core component of the Architecture. It executes at runtime the Application code.

Note: In the following explanations, the term *task* refers to native tasks scheduled by the underlying OS or RTOS, while *thread* refers to MicroEJ threads scheduled by the Core Engine.

Block Diagram

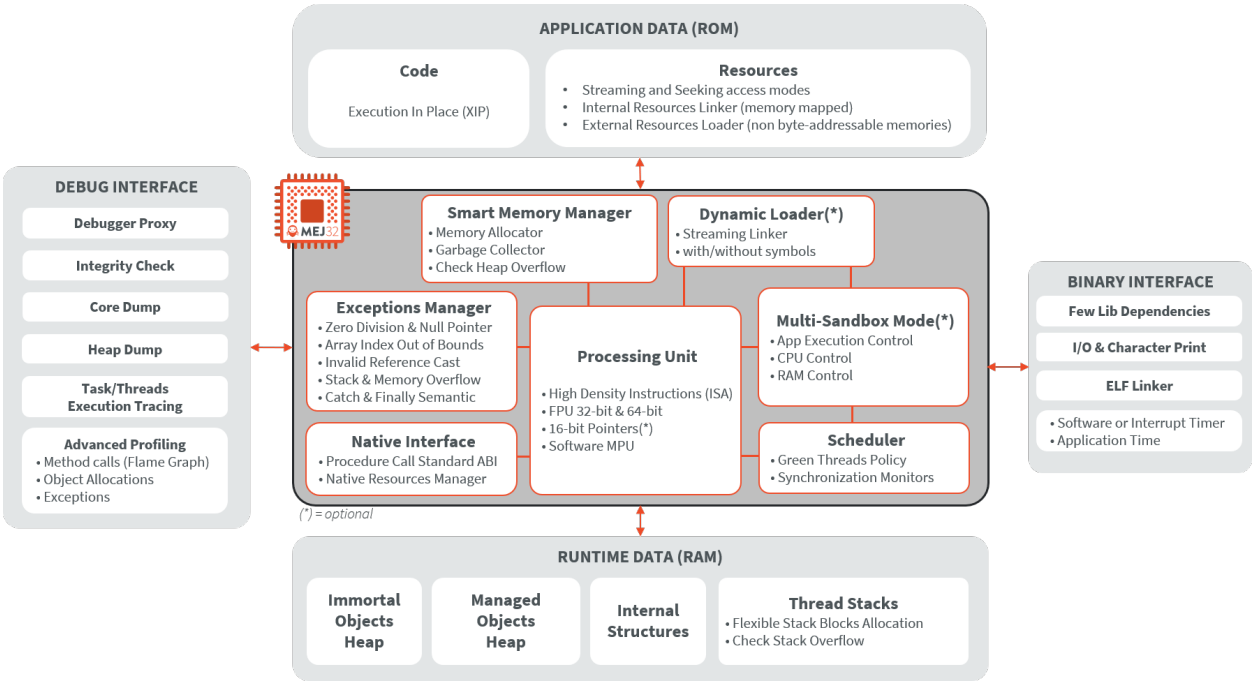


Fig. 181: Core Engine Block Diagram

Link Flow

The following diagram shows the overall build flow. Application development is performed within MICROEJ SDK. The remaining steps are performed within the C third-party IDE.

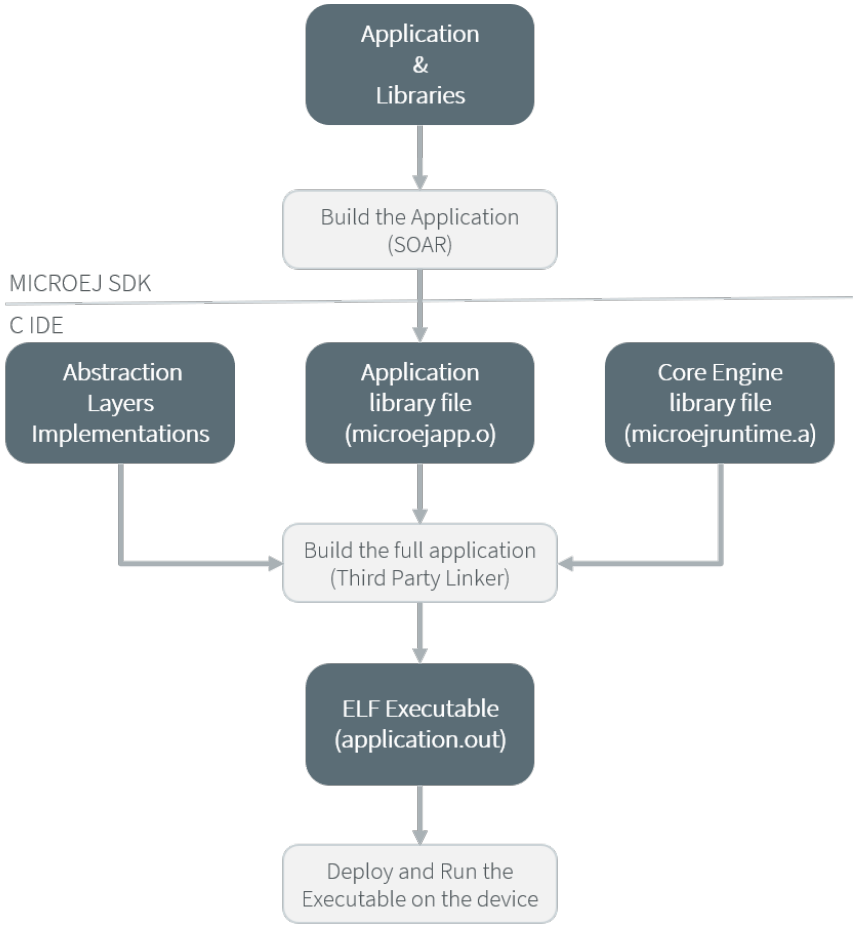


Fig. 182: Core Engine Flow

1. Step 1 consists in writing an Application against a set of Foundation Libraries available in the VEE Port.
2. Step 2 consists in compiling the Application code and the required libraries in an ELF library, using the SOAR.
3. Step 3 consists in linking the previous ELF file with the Core Engine library and a third-party BSP (OS, drivers, etc.). This step requires a third-party linker provided by a C toolchain.

Architecture

The Core Engine and its components have been compiled for one specific CPU architecture and for use with a specific C compiler.

Refer to the chapter *Architectures MCU / Compiler* for list of supported Architectures and the details of ABI and compiler options.

Threading Integration

The Core Engine implements a *green thread model*. It runs in a single task.

Green threads are threads that are internally managed by the Core Engine instead of being natively managed by the underlying OS/RTOS scheduler. The Core Engine defines a multi-threaded environment without relying on any native OS capabilities.

Therefore, the whole Managed world runs in one single task, within which the Core Engine re-creates a layer of (green) threads. One immediate advantage is that the Java-world CPU consumption is fully controlled by the task it is running in, allowing embedded engineers to easily arbitrate between the different parts of their application. In particular in an open-to-third-parties framework, the maximum CPU time given to the Managed world is fully under control at no risk, whatever the number and/or the activities of the threads.

The next illustration shows 4 tasks, with the last one running the Core Engine with 2 threads. When the last task is scheduled by the underlying OS, the Core Engine executes and schedules the threads.

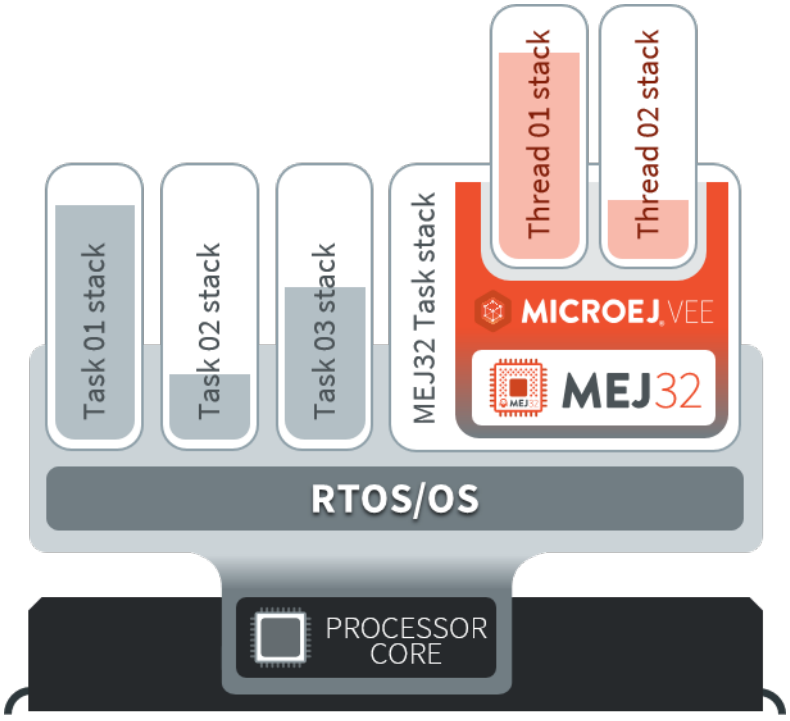


Fig. 183: A Green Threads Architecture Example

The activity of the Core Engine is defined by the Application. When the Application is blocked (i.e., when all the MicroEJ threads sleep), the task running the Core Engine sleeps.

Capabilities

The Core Engine defines 3 exclusive capabilities:

- Mono-Sandbox: capability to produce a monolithic Executable (default one).
- Multi-Sandbox: capability to produce a extensible Executable on which new applications can be dynamically installed. See section *Multi-Sandbox*.
- Tiny-Sandbox: capability to produce a compacted Executable (optimized for size). See section *Tiny-Sandbox*.

All the Core Engine capabilities may not be available on all architectures. Refer to section *Supported Core Engine Capabilities by Architecture Matrix* for more details.

To select the Core Engine capability, define the property `com.microej.runtime.capability` in the `configuration.properties` file (SDK 6) or in the `mjvm/mjvm.properties` file (SDK 5) of the VEE Port project, with one of the following values:

- `mono` for Mono-Sandbox (default value)
- `multi` for Multi-Sandbox
- `tiny` for Tiny-Sandbox

If the property `com.microej.runtime.capability` is not defined, the Mono-Sandbox Core Engine capability is used.

Implementation

The Core Engine implements the *[SNI] specification*. It is created and initialized with the C function `SNI_createVM`. Then it is started and executed in the current task by calling `SNI_startVM`. The function `SNI_startVM` returns when the Application exits or if an error occurs (see section *Error Codes*). The function `SNI_destroyVM` handles the Core Engine termination and must be called after the return of the function `SNI_startVM`.

Only one instance of the Core Engine can be created in the system, and both `SNI_createVM` and `SNI_destroyVM` should only be called once. When restarting the Core Engine, don't call `SNI_createVM` or `SNI_destroyVM` before calling `SNI_startVM` again. For more information, refer to the *Restart the Core Engine* section.

The file `LLMJVM_impl.h` that comes with the Architecture defines the API to be implemented. See section *LLMJVM: Core Engine*.

Initialization

The Low Level Core Engine API deals with two objects: the structure that represents the Core Engine, and the task that runs the Core Engine. Two callbacks allow engineers to interact with the initialization of both objects:

- `LLMJVM_IMPL_initialize`: Called once the structure representing the Core Engine is initialized.
- `LLMJVM_IMPL_vmTaskStarted`: Called when the Core Engine starts its execution. This function is called within the task of the Core Engine.

Scheduling

To support the green thread round-robin policy, the Core Engine assumes there is an RTOS timer or some other mechanism that counts (down) and fires a call-back when it reaches a specified value. The Core Engine initializes the timer using the LLMJVM_IMPL_scheduleRequest function with one argument: the absolute time at which the timer should fire. When the timer fires, it must call the LLMJVM_schedule function, which tells the Core Engine to execute a green thread context switch (which gives another MicroEJ thread a chance to run).

When several MicroEJ threads with the same priority are eligible for execution, the round-robin algorithm will automatically switch between these threads after a certain amount of time, called the *time slice*. The time slice is expressed in milliseconds, and its default value is 20 ms. It can be configured at link time with the symbol __java_round_robin_period , defined in the *linker configuration file* linkVMConfiguration.lscf located in the VEE Port folder /MICROJVM/link/ . To override the content of this file, create, in the VEE Port configuration project, a folder named /dropins/MICROJVM/link/ , and copy into this folder the file linkVMConfiguration.lscf retrieved from an existing VEE Port. Since a symbol cannot be null, the actual time slice value in milliseconds is __java_round_robin_period - 1 . Set the symbol to 1 (i.e., time slice to 0) to disable the round-robin scheduling.

> **Warning:** Modifying the time slice value is an advanced configuration that can impact the performances.
>
> Decreasing the time slice will increase the number of context switches. Therefore scheduler will use more CPU time.
>
> Increasing the time slice can create a latency with intensive threads monopolizing the CPU.

Idle Mode

When the Core Engine has no activity to execute, it calls the LLMJVM_IMPL_idleVM function, which is assumed to put the Core Engine task into a sleep state. LLMJVM_IMPL_wakeupVM is called to wake up the Core Engine task. When the Core Engine task really starts to execute again, it calls the LLMJVM_IMPL_ackWakeup function to acknowledge the restart of its activity.

Time

The Core Engine defines two different times:

- the application time: the difference, measured in milliseconds, between the current time and midnight, January 1, 1970, UTC.
- the monotonic time: this time always moves forward and is not impacted by application time modifications (NTP or Daylight Savings Time updates). It can be implemented by returning the running time since the start of the device.

The Core Engine relies on the following C functions to provide those times to the Application:

- LLMJVM_IMPL_getCurrentTime : must return the monotonic time in milliseconds if the given parameter is 1 , otherwise must return the application time in milliseconds. This function is called by the method java.lang.System.currentTimeMillis() It is also used by the Core Engine scheduler, and should be implemented efficiently.

- LLMJVM_IMPL_getTimeNanos : must return a monotonic time in nanoseconds.
- LLMJVM_IMPL_setApplicationTime : must set the difference between the current time and midnight, January 1, 1970, UTC. Implementations may apply this time to the whole underlying system or only to the Core Engine (i.e., the value returned by LLMJVM_IMPL_getCurrentTime(0)).

Configuring Memory Dynamically

To configure the heaps sizes at the Core Engine startup time rather than at link time (see *Managed Heap size configuration* and *Immortals Heap size configuration* Application options), implement the Core Engine configuration Low Level API (see *LLMJVM_CONF_impl.h*):

- LLMJVM_CONF_impl_get_managed_heap_memory() : allocates the Managed Heap.
- LLMJVM_CONF_impl_get_immortals_heap_memory() : allocates the Immortals Heap.

The Core Engine provides a default implementation for those functions which returns memory sections statically allocated using the heaps sizes Application options and third-party linker script for position. If the default implementation is overridden, the heap configuration properties are ignored unless the custom implementation uses them. These functions are called by the Core Engine in SNI_createVM() .

Error Codes

The C function SNI_createVM returns a negative value if an error occurred during the Core Engine initialization or execution. The file LLMJVM.h defines the Core Engine error code constants. The following table describes these error codes.

Table 58: Core Engine Error Codes

Error Code	Meaning
0	The Application ended normally (i.e., all the non-daemon threads are terminated or System.exit(exitCode) has been called). See section *Exit Codes*.
-1	The microejapp.o produced by SOAR is not compatible with the Core Engine (microejruntime.a). The object file has been built from another Architecture.
-2	Internal error. Invalid link configuration in the Architecture or the VEE Port.
-3	Evaluation version limitations reached: termination of the application. See section *Limitations*.
-5	Not enough resources to start the very first MicroEJ thread that executes main method. See section *Option(text): Managed Heap size (in bytes)*.
-12	Number of threads limitation reached. See sections *Limitations* and *Option(text): Number of threads*.
-13	Fail to start the Application because the specified Managed Heap is too large or too small. See section *Option(text): Managed Heap size (in bytes)* & *Limitations*
-14	Invalid Application stack configuration. The stack start or end is not eight-byte aligned, or stack block size is too small. See section *Option(text): Number of blocks in pool*.
-16	The Core Engine cannot be restarted.
-17	The Core Engine is not in a valid state because of one of the following situations: • SNI_startVM called before SNI_createVM . • SNI_startVM called while the Appplication is running. • SNI_createVM called several times.
-18	The memory used for the Managed Heap or Immortals Heap does not work properly. Read/Write memory checks failed. This may be caused by an invalid external RAM configuration. Verify _java_heap and _java_immortals sections locations.
-19	The memory used for the Application static fields does not work properly. Read/Write memory checks failed. This may be caused by an invalid external RAM configuration. Verify .bss.soar section location.
-20	KF configuration internal error. Invalid link configuration in the Architecture or the VEE Port.
-21	Number of monitors per thread limitation reached. See sections *Limitations* and *Options* .
-22	Internal error. Invalid FPU configuration in the Architecture.
-23	The function LLMJVM_IMPL_initialize defined in the Abstraction Layer implementation returns an error.
-24	The function LLMJVM_IMPL_vmTaskStarted defined in the Abstraction Layer implementation returns an error.
-25	The function LLMJVM_IMPL_shutdown defined in the Abstraction Layer implementation returns an error.
-26	The GC mark stack size

Example

The following example shows how to create and launch the Core Engine from the C world. This function (microej_main) should be called from a dedicated task.

```
#include <stdio.h>
#include "microej_main.h"
#include "LLMJVM.h"
#include "sni.h"

#ifdef __cplusplus
  extern "C" {
#endif

/**
 * @brief Creates␣
↪and starts a MicroEJ instance. This function returns when the MicroEJ execution ends.
 * @param argc arguments count
 * @param argv arguments vector
 * @param app_exit_code_ptr pointer where this function␣
↪stores the application exit code or 0 in case of error in the Core Engine. May be null.
 * @return the Core Engine error code in case of error, or 0 if the execution ends without error.
 */
int microej_main(int argc, char **argv, int* app_exit_code_ptr) {
	void* vm;
	int core_engine_error_code = -1;
	int32_t app_exit_code = 0;
	// create Core Engine
	vm = SNI_createVM();

	if (vm == NULL) {
		printf("MicroEJ initialization error.\n");
	} else {
		printf("MicroEJ START\n");

		// Error codes documentation is available in LLMJVM.h
		core_engine_error_code = (int)SNI_startVM(vm, argc, argv);

		if (core_engine_error_code < 0) {
			// Error occurred
			if (core_engine_error_code == LLMJVM_E_EVAL_LIMIT) {
				printf("Evaluation limits reached.\n");
			} else {
	␣
↪			printf("MicroEJ execution error (err = %d).\n", (int) core_engine_error_code);
			}
		} else {
			// Core Engine execution ends normally
			app_exit_code = SNI_getExitCode(vm);
			printf("MicroEJ END (exit code = %d)\n", (int) app_exit_code);
		}

		// delete Core Engine
		SNI_destroyVM(vm);
	}

	if(app_exit_code_ptr != NULL){
```

(continues on next page)

(continued from previous page)

```
            *app_exit_code_ptr = (int)app_exit_code;
        }

        return core_engine_error_code;
}

#ifdef __cplusplus
  }
#endif
```

Restart the Core Engine

The Core Engine supports the restart of the Application after the end of its execution. The application stops when all non-daemon threads are terminated or when `System.exit(exitCode)` is called. When the application ends, the C function `SNI_startVM` returns.

To restart the application, call again the `SNI_startVM` function (see the following pattern).

```
// create Core Engine (called only once)
vm = SNI_createVM();
...
// start a new execution of the Application at each iteration of the loop
while(...){
        ...
        core_engine_error_code = SNI_startVM(vm, argc, argv);
        ...
        // Get exit status passed to System.exit()
        app_exit_code = SNI_getExitCode(vm);
        ...
}
...
// delete Core Engine (called before stopping the whole system)
SNI_destroyVM(vm);
```

Note: Please note that while the Core Engine supports restart, *MicroUI* does not. Attempting to restart the Application on a VEE Port with UI support may result in undefined behavior.

Note: Please note that `SNI_createVM` and `SNI_destroyVM` should only be called once. When restarting the Core Engine, don't call `SNI_createVM` or `SNI_destroyVM` before calling `SNI_startVM` again.

Dump the State of the Core Engine

The internal Core Engine function called LLMJVM_dump allows you to dump the state of all MicroEJ threads: name, priority, stack trace, etc. This function must only be called from the Core Engine thread context and only from a native function or callback. Calling this function from another context may lead to undefined behavior and should be done only for debug purpose.

This is an example of a dump:

```
==================================
↪VM Dump ===================================
Java threads count: 3
Peak java threads count: 3
Total created java threads: 3
Last executed native function: 0x90035E3D
Last executed external hook function: 0x00000000
State: running
---------------------------------------------------
Java Thread[1026]
name="main" prio=5 state=RUNNING max_java_stack=456 current_java_stack=184

java.lang.MainThread@0xC0083C7C:
    at (native) [0x90003F65]
    at com.microej.demo.widget.main.MainPage.getContentWidget(MainPage.java:95)
        Object References:
            - com.microej.demo.widget.main.MainPage@0xC00834E0
            - com.microej.demo.widget.main.MainPage$1@0xC0082184
            - java.lang.Thread@0xC0082194
            - java.lang.Thread@0xC0082194
    at com.microej.demo.widget.common.Navigation.createRootWidget(Navigation.java:104)
        Object References:
            - com.microej.demo.widget.main.MainPage@0xC00834E0
    at com.microej.demo.widget.common.Navigation.createDesktop(Navigation.java:88)
        Object References:
            - com.microej.demo.widget.main.MainPage@0xC00834E0
            - ej.mwt.stylesheet.CachedStylesheet@0xC00821DC
    at com.microej.demo.widget.common.Navigation.main(Navigation.java:40)
        Object References:
            - com.microej.demo.widget.main.MainPage@0xC00834E0
    at java.lang.MainThread.run(Thread.java:855)
        Object References:
            - java.lang.MainThread@0xC0083C7C
    at java.lang.Thread.runWrapper(Thread.java:464)
        Object References:
            - java.lang.MainThread@0xC0083C7C
    at java.lang.Thread.callWrapper(Thread.java:449)
---------------------------------------------------
Java Thread[1281]
name="UIPump"
↪prio=5 state=WAITING timeout(ms)=INF max_java_stack=120 current_java_stack=117
external event: status=waiting

java.lang.Thread@0xC0083628:
    at ej.microui.MicroUIPump.read(Unknown Source)
        Object References:
            - ej.microui.display.DisplayPump@0xC0083640
```

(continues on next page)

(continued from previous page)

```
    at ej.microui.MicroUIPump.run(MicroUIPump.java:176)
        Object References:
            - ej.microui.display.DisplayPump@0xC0083640
    at java.lang.Thread.run(Thread.java:311)
        Object References:
            - java.lang.Thread@0xC0083628
    at java.lang.Thread.runWrapper(Thread.java:464)
        Object References:
            - java.lang.Thread@0xC0083628
    at java.lang.Thread.callWrapper(Thread.java:449)
-------------------------------------------------------
Java Thread[1536]
name="Thread1" prio=5 state=READY max_java_stack=60 current_java_stack=57

java.lang.Thread@0xC0082194:
    at java.lang.Thread.runWrapper(Unknown Source)
        Object References:
            - java.lang.Thread@0xC0082194
    at java.lang.Thread.callWrapper(Thread.java:449)
=======================================================================================

================================␣
↪Garbage Collector ================================
State: Stopped
Last analyzed object: null
Total memory: 15500
Current allocated memory: 7068
Current free memory: 8432
Allocated memory after last GC: 0
Free memory after last GC: 15500
=======================================================================================

================================␣
↪Native Resources =================================
Id          CloseFunc  Owner            Description
-------------------------------------------------------
=======================================================================================
```

See Stack Trace Reader documentation for *SDK 6* or *SDK 5* for additional info related to working with Core Engine dumps.

Dump The State Of All MicroEJ Threads From A Fault Handler

It is recommended to call the `LLMJVM_dump` API as a last resort in a fault handler. Calling `LLMJVM_dump` is undefined if the Core Engine is not paused. The call to `LLMJVM_dump` MUST be done last in the fault handler.

Trigger Core Engine Dump From Debugger

To trigger a Core Engine dump manually from the debugger, you need to set the PC (Program Counter) register to the physical memory address of the `LLMJVM_dump` function.

Follow these steps:

1. **Ensure LLMJVM_dump is not optimized out**

   Explicitly reference the `LLMJVM_dump` function in your BSP code. It is declared in the `LLMJVM.h` header file.

   Note: If the function is not used anywhere, linker optimization may remove it, making it unavailable in the final binary.

2. **Locate the symbol in the map file**

   Search for the symbol `___icetea___virtual___com_is2t_microjvm_IGreenThreadMicroJvm____dump` in your C toolchain's `.map` file. This will give you the runtime memory address of the function.

3. **Trigger the dump via the debugger**

   In your debugger, set the **PC register** to the retrieved address. Then, resume execution to trigger the dump.

Note: `LLMJVM_dump` is an alias defined in `intern/LLMJVM.h` header file. If you cannot find the symbol listed above, check the macro definition in that header file to determine the actual function name being referenced and exported.

Requirements:

- Embedded debugger is attached and the processor is halted in an exception handler.
- A way to read stdout (usually UART).

Check Internal Structure Integrity

The internal Core Engine function called `LLMJVM_checkIntegrity` checks the internal structure integrity of the Core Engine and returns its checksum.

- If an integrity error is detected, the `LLMJVM_on_CheckIntegrity_error` hook is called and this method returns `0`.
- If no integrity error is detected, a non-zero checksum is returned.

  This function must only be called from the Core Engine thread context and only from a native function or callback. Calling this function multiple times in a native function should always produce the same checksum. If the returned checksums are different, a corruption must have occurred.

  Please note that returning a non-zero checksum does not mean the Core Engine data has not been corrupted, as it is not possible for the Core Engine to detect the complete memory integrity.

The internal structures of the Core Engine that can be altered legitimately by a native function do not impact the checksum calculation. The following internal structures may be modified without affecting the checksum:

- basetype fields in Java objects or content of Java arrays of base type,
- internal structures modified by a LLMJVM function call (e.g., set a pending Java exception, suspend or resume the thread, register a resource, ...).

This function affects the performances and should only be used for debug purpose. A typical use of this API is to verify that a native implementation does not corrupt the internal structures:

```
#include <stdio.h>
#include "LLMJVM.h"

void Java_com_mycompany_MyClass_myNativeFunction(void) {
    int32_t crcBefore = LLMJVM_checkIntegrity();
    myNativeFunctionDo();
    int32_t crcAfter = LLMJVM_checkIntegrity();
    if(crcBefore != crcAfter){
        // Corrupted Core Engine internal structures
        while(1);
    }
}

// Hook called by the Core Engine when an integrity error is detected
void LLMJVM_on_CheckIntegrity_error(uint32_t errorCode, void* errorAddress) {
    ␣
↪printf("Integrity error detected at address %p (error code: %d)\n", errorAddress, errorCode);
}
```

Generic Output

The System.err stream is connected to the System.out print stream. See below for how to configure the destination of these streams.

Link

Several sections are defined by the Core Engine. Each section must be linked by the third-party linker. Read-Only (RO) sections can be placed in writable memories. In such cases, it is the responsibility of the BSP to prevent these sections from being written.

Starting from *Architecture 8.0.0*, sections have been renamed to follow the standard ELF naming convention.

Linker Sections (Architecture 8.x)

Linker Sections (Architecture 7.x)

Section name	Aim	Location	Alignment (in bytes)
.bss.microej.heap	Application heap	RW	4
.bss.microej.immortals	Application Immortals Heap	RW	4
.bss.microej.stacks	Application threads stack blocks	RW[1]	8
.bss.microej.statics	Application static fields	RW	8
.rodata.microej.resource.*	Application resources (one section per resource)	RO	16
.rodata.microej.soar	Application and library code	RO	16
.bss.microej.runtime	Core Engine internal structures	RW	8
.text.___icetea___*	Core Engine generated code	RX	ISA Specific
.bss.microej.kernel	Core Engine Multi-Sandbox section (Feature code chunk)	RW	4

Page 864

Note: During its startup, the Core Engine automatically zero-initializes the sections .bss.microej.runtime , .bss.microej.heap , and .bss.microej.immortals .

Section name	Aim	Location	Alignment (in bytes)
__java_heap	Application heap	RW	4
__java_immortals	Application Immortals Heap	RW	4
.bss.vm.stacks.java	Application threads stack blocks	RW[1]	8
.bss.soar	Application static fields	RW	8
.rodata.resources	Application resources	RO	16
.text.soar	Application and library code	RO	16
ICETEA_HEAP	Core Engine internal structures	RW[1]	8
.text.___icetea___*	Core Engine generated code	RX	ISA Specific

Note: During its startup, the Core Engine automatically zero-initializes the sections ICETEA_HEAP , __java_heap , and __java_immortals .

[1] Among all RW sections, those should be always placed into internal RAM for performance purpose.

Dependencies

The Core Engine requires an implementation of its low level APIs in order to run. Refer to the chapter *Implementation* for more information.

Installation

The Core Engine and its components are mandatory. By default, it is configured with Mono-Sandbox capability. See the *Capabilities* section to update the Core Engine with Multi-Sandbox or Tiny-Sandbox capability.

Abstraction Layer

Core Engine Abstraction Layer implementations can be found on MicroEJ Github for several RTOS.

Memory Considerations

The memory consumption of main Core Engine runtime elements are described in *the table below*.

Table 59: Memory Considerations

Runtime element	Memory	Size in bytes (Mono-sandbox)	Size in bytes (Multi-Sandbox)	Size in bytes (Tiny-Sandbox)
Object Header	RW	4	8 (+4)	4
Thread	RW	168	192 (+24)	168
Call Frame Header	RW	12	20 (+8)	12
Class Type	RO	32	36 (+4)	32
Interface Type	RO	16	24 (+8)	16

Note: To get the full size of an Object, search for the type in the *SOAR Information File* and read the attribute `instancesize` (this includes the Object header).

Note: To get the full size of a call frame, search for the method in the *SOAR Information File* and read the attribute `stacksize` (this includes the call frame header).

Use

Refer to the *MicroEJ Runtime* documentation.

3.6.7 Event Tracing

Principle

Event Tracing allows to record integer based events for debugging and monitoring purposes without affecting execution performance too heavily.

A record is an event type identified by an `eventID` and can have a list of values. It can be a single event or a period of time with a start and an end.

Events recorded are then redirected to the trace system implemented in the VEE Port.

The Trace feature relies on the implementation of natives defined in the file `LLTRACE_impl.h`. In order to enable the monitoring of events generated by the Core Engine, the implementation of functions defined in `LLMJVM_MONITOR_impl.h` is also necessary.

Dependencies

- `LLTRACE_impl.h` implementations (see *LLTRACE: Trace*).
- `LLMJVM_MONITOR_impl.h` implementations (see *LLMJVM_MONITOR: Core Engine Monitoring*).

Existing implementations

By default, when enabled, the Trace API displays a message in the standard output for every `recordEvent(...)` and `recordEventEnd(...)` method calls.

It does not print a timestamp when displaying the trace message because it can drastically affect execution performances. It only prints the ID of the recorded event followed by the values given in parameters.

A VEE Port can connect its own implementation by overriding the functions defined in the files `LLTRACE_impl.h` and `LLMJVM_MONITOR_impl.h`.

MicroEJ Corp. also provides an implementation that redirects the events to *SystemView* tool, the real-time recording and visualization tool from Segger. It is perfect for a finer understanding of the runtime behavior by showing events sequence and duration.

An implementation example for the NXP i.MX RT1170 development board VEE Port is available on Github.

Installation

To activate Event Tracing in your VEE Port, you need to follow these steps:

- Enable and start the trace *see here*

Note: This feature is provided by the Core Engine, and does not require to import a Foundation Library Pack.

Use

See *Java API Usage* section in the Application Developer Guide to use the feature in Java at the Application level.

Events can also be generated in C using the API described in the file LLTRACE.h . The functions documentation is available *here*.

Please contact *our support team* for more information about how to integrate this module.

3.6.8 Advanced Event Tracing

Principle

Core Engine allows method execution to be profiled. The following two new hooks functions are used for that:

- LLMJVM_MONITOR_IMPL_on_invoke_method called at the start of the method invocation.
- LLMJVM_MONITOR_IMPL_on_return_method called when returning from the invoked method.

Calling these functions each time a method is invoked will slow down the application execution, so these functions are not called by default when event tracing is enabled and started.

Note: This feature requires Architecture version 7.17.0 or higher and is only available on the Core Engine, not on the Simulator.

To activate them, you need to follow these steps:

- Enable and start the trace *see here*
- Tell the third-party linker program to redirect all calls to LLMJVM_invoke_method and LLMJVM_return_method symbols to respectively LLMJVM_invoke_method_with_trace and LLMJVM_return_method_with_trace symbols.

Platforms using GNU LD linker

Add the following options to the LD linker command line:

```
--require-defined=LLMJVM_invoke_method_with_trace
--defsym=LLMJVM_invoke_method=LLMJVM_invoke_method_with_trace
--require-defined=LLMJVM_return_method_with_trace
--defsym=LLMJVM_return_method=LLMJVM_return_method_with_trace
```

Platforms using IAR ILINK linker

Pass the following options to the ILINK linker program

```
--redirect LLMJVM_invoke_method=LLMJVM_invoke_method_with_trace
--redirect LLMJVM_return_method=LLMJVM_return_method_with_trace
```

3.6.9 Multi-Sandbox

Principle

The Multi-Sandbox capability of the Core Engine allows a main application (called Standalone Application) to install and execute at runtime additional applications (called Sandboxed Applications).

The Core Engine implements the *Kernel & Features Specification (KF)*. A Kernel is a Standalone Application generated on a Multi-Sandbox-enabled VEE Port. A Feature is a Sandboxed Application generated against a specific Kernel.

Functional Description

The Multi-Sandbox process extends the overall process described in *the overview of the platform process*.

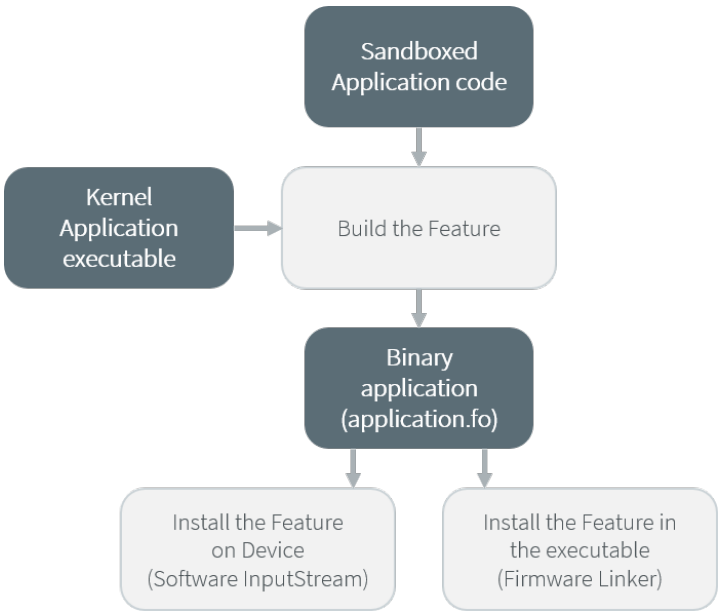


Fig. 184: Multi-Sandbox Process

Once a Kernel has been generated, additional Sandboxed Application code (Feature) can be built against the Kernel. The binary file produced (the `.fo` file) can be installed on the Kernel on which it was generated.

For more details on the build flow, please refer to *Multi-Sandbox Kernel link* and *Sandboxed Application link* sections.

Memory Considerations

Multi-Sandbox memory overhead of Core Engine runtime elements are described in *Memory Considerations table*.

Dependencies

- `LLKERNEL_impl.h` implementation (see *Feature Installation* section).

Installation

Multi-Sandbox is an option disabled by default. To enable the Multi-Sandbox capability of the Core Engine, set the property `com.microej.runtime.capability` to `multi` in the VEE Port project:

SDK 6

SDK 5

In the `configuration.properties` file:

```
com.microej.runtime.capability=multi
```

In the `mjvm/mjvm.properties` file of the VEE Port Configuration module:

```
com.microej.runtime.capability=multi
```

Note: In SDK 5, before *Architecture 8.1.0*, to enable the Multi-Sandbox capability of the Core Engine, select the Multi Applications module in the platform configuration file.

Use

The KF API Module must be added to the build file of the Application project to use *Kernel & Features Specification (KF)* library.

SDK 6

SDK 5

In the `build.gradle.kts` file:

```
implementation("ej.api:kf:1.7.0")
```

In the `module.ivy` file:

```
<dependency org="ej.api" name="kf" rev="1.7.0" />
```

This library provides a set of options. Refer to the chapter *Standalone Application Options* which lists all available options.

Feature Installation

Introduction

Feature installation is triggered by a call to the Kernel.install(InputStream) method. It consists of the following steps:

- loading Feature's content from `.fo` file,
- linking Feature's code with the Kernel,
- storing Feature's content into the target memory.

A Feature `.fo` file is composed of the following elements:

- Code: Application code (methods, types, …) as well as built-in objects (strings and immutables),
- RO Data: *Application Resources* that do not require content modification,
- RW Data: Reserved memory for Feature execution (Application static fields and Feature internal structures),
- Metadata: Temporary information required during the installation phase, such as code relocations.

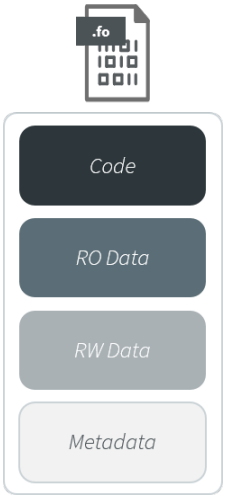


Fig. 185: Feature .fo File Content

Feature installation flow allows to install Features in any byte-addressable memory mapped to the CPU's address space. The Feature content is read chunk-by-chunk from the InputStream and progressively transferred to the target memory. Only a small amount of RAM is required. The LLKERNEL_impl.h Abstraction Layer interface provides Low Level APIs for allocating and transferring Feature content in different memory areas, including ROM.

Installation Flow

The RO Data (Application Resources) is directly transferred to the target location. The Code is divided into chunks. Each chunk is temporarily copied to RAM to be relocated. Then it is transferred to the target location.

A minimum amount of RAM is required:

- A temporary buffer is allocated in the Managed Heap for reading bytes from the InputStream,
- Metadata is allocated in the Managed Heap,
- Code chunk is temporarily copied in a memory area to be relocated (see more details below).

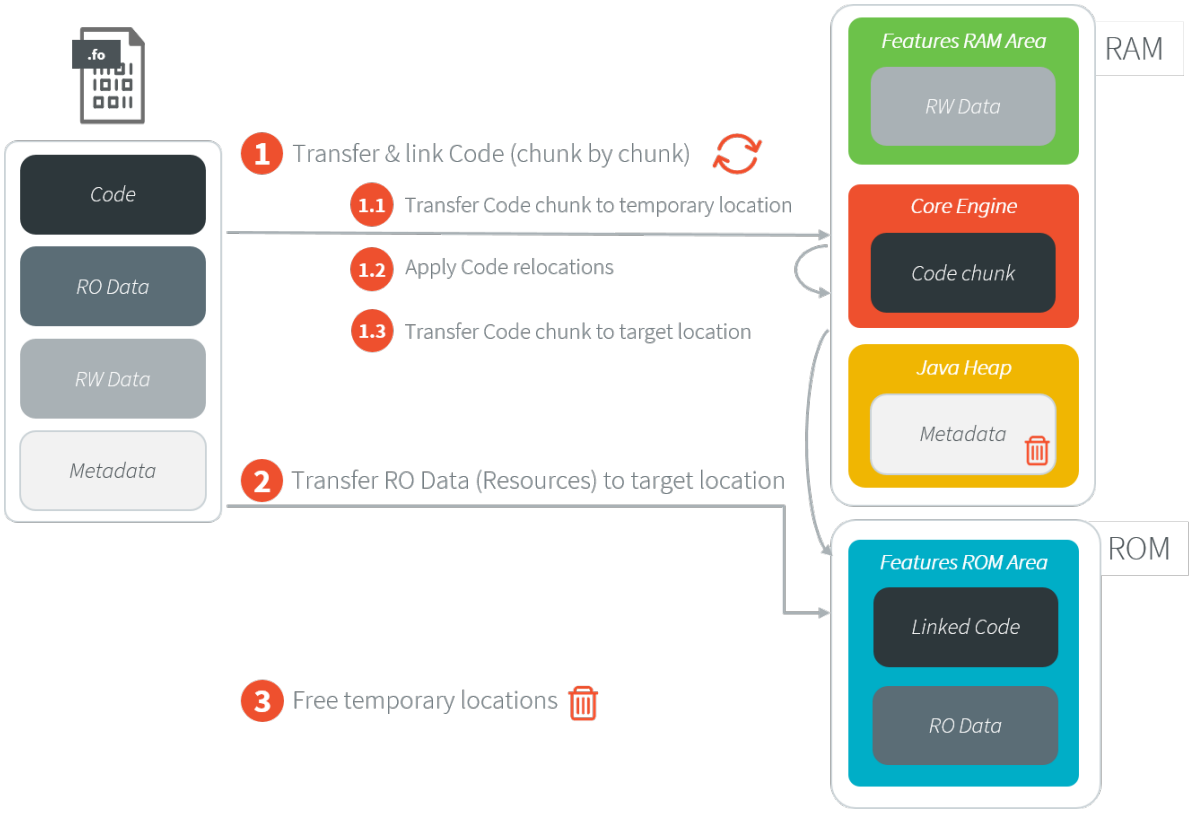


Fig. 186: Feature Installation Steps

The Abstraction Layer implementation is responsible for providing the following elements:

- the location where the Feature will be installed,
- the implementation to copy a chunk of bytes to the target location.

The detailed installation flow is described in the following sequence diagram:

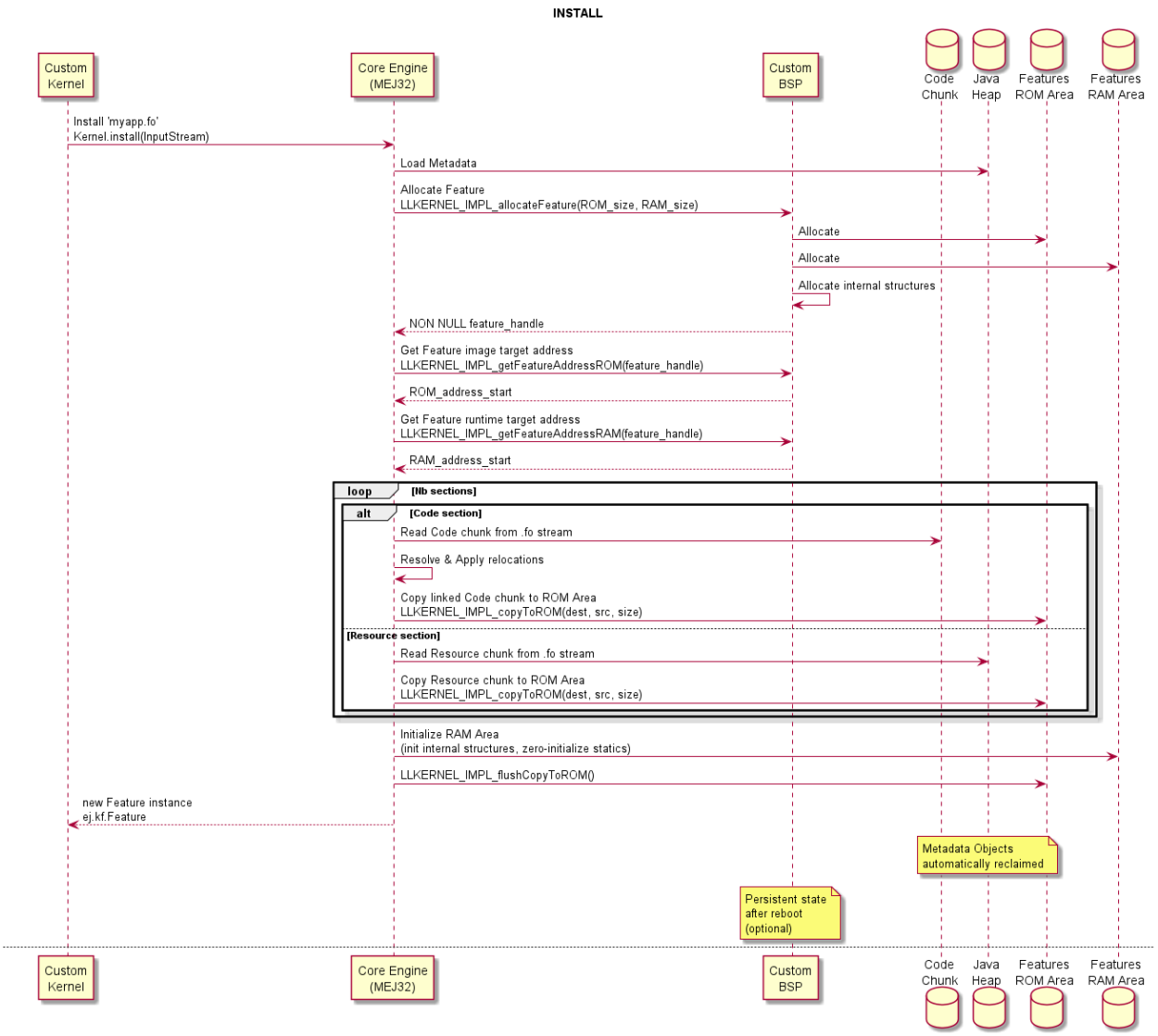


Fig. 187: Feature Installation Flow

The detailed uninstallation flow is described in the following sequence diagram:

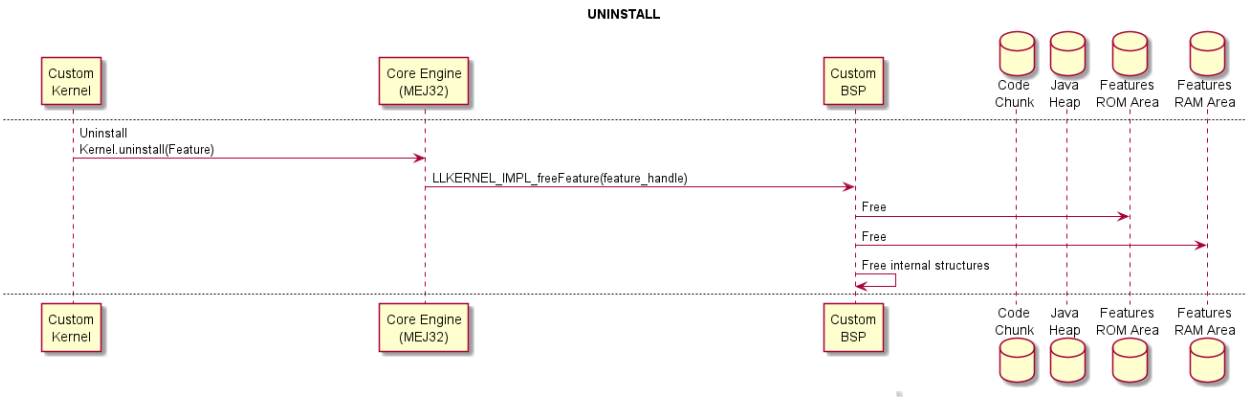


Fig. 188: Feature Uninstallation Flow

Feature Persistency

Feature Persistency is the ability of the Core Engine to gather installed Features from prior executions of the Kernel upon start up. This means that the Kernel will boot with a set of available Features that were already installed. To ensure that the Features remain available even after the device restarts, you will have to implement an Abstraction Layer that stores the Features into a Read-Only memory.

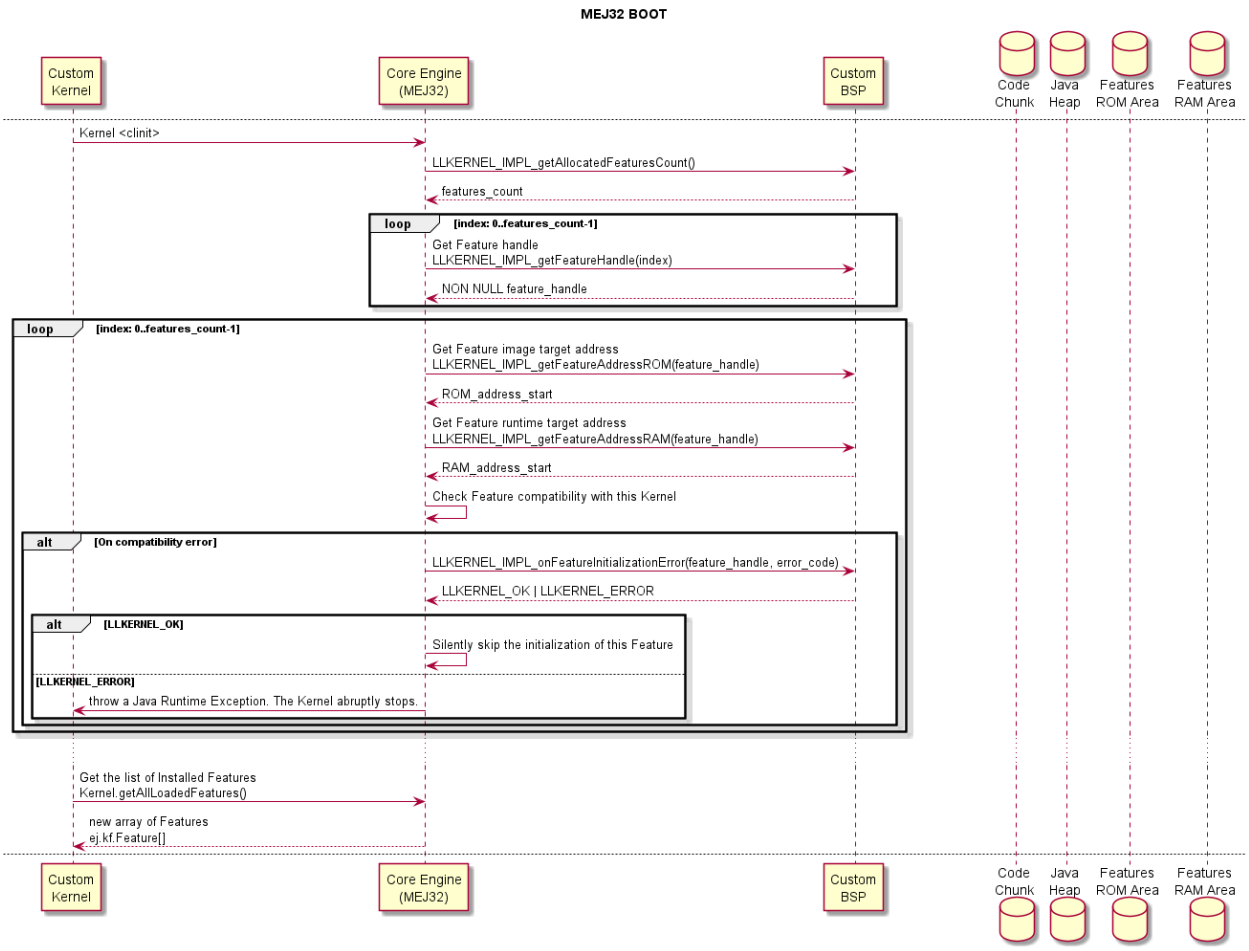


Fig. 189: Feature Installation Boot Flow

Note: Features are available in the *INSTALLED* state. It is the responsibility of the Kernel to manually start the desired Features.

Advanced Options

Code Chunk Size

Feature .fo Code section is divided into chunks that are temporary copied to RAM to be relocated. The Code chunk size can be configured with the following option:

Option Name: com.microej.soar.kernel.featurecodechunk.size

Default Value: 65536 (bytes)

A small number will reduce the RAM consumption but will increase the .fo size and will affect the installation time.

InputStream Transfer Buffer Size

When calling the Kernel.install(InputStream) method, the Feature .fo bytes are read from the InputStream using a temporary byte array allocated in the Managed Heap. The size of this array can be configured with the following option:

Option Name: com.microej.runtime.kf.link.transferbuffer.size

Default Value: 512 (bytes)

Relocation Process Yield

When a Feature file has a large amount of code, it may appear that the Core Engine blocks while applying relocations during the Feature installation. The number of relocations to apply in batch can be configured with the following option:

Option Name: com.microej.runtime.kf.link.chunk.relocations.count

Default Value: 128

Once the Core Engine has processed the given number of relocations, the thread that called the Kernel.install(InputStream) method yields the execution to other threads. A small number will give more smooth execution for threads but a slowest installation execution. A large number will make the Core Engine block for applying relocations but a faster installation execution.

Determining the Amount of Required Memory

The amount of memory required for installing a .fo file is determined by analyzing the sizes of the ELF sections.

Sections can be dumped using the standard binutils readelf tool:

```
readelf -WS application.fo
There are 8 section headers, starting at offset 0x34:

Section Headers:
[Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
[ 0]                   NULL            00000000 000000 000000 00      0   0  0
[ 1] .soar.rel         LOPROC+0        00000000 000174 000bcc 00      6   0  4
[ 2] .strtab           STRTAB          00000000 000d40 000063 00      0   0  1
[ 3] .symtab           SYMTAB          00000000 000da4 000050 10      2   1  4
[ 4] .bss.soar.feature NOBITS          00000000 000df4 000050 00   A  0   0  4
[ 5] .rodata.microej.resources PROGBITS        00000000 000e00 079080 00   A  0   0 64
[ 6] .rodata           PROGBITS        00000000 079e80 001974 00   A  0   0 16
[ 7] .shstrtab         STRTAB          00000000 07b7f4 000059 00      0   0  1
```

The following table summarizes the sections and their content:

Section	Description	Temporary Memory Location	Target Memory Location
.soar.rel	Metadata	Managed Heap	None
.strtab	Metadata	Managed Heap	None
.symbtab	Metadata	Managed Heap	None
.bss.soar.feature	RW Data	None	Features RAM area
.rodata.microej.resources	RO Data	None	Features ROM area
.rodata	Code chunk	RAM	Features ROM area
.shstrtab	Metadata	Managed Heap	None

In-Place Installation

Note: This section describes the legacy Feature installation flow, based on a malloc/free implementation in RAM. It is deprecated and available up to *Architecture 8.0.0*.

See *Migrate Your LLKERNEL Implementation* for migrating to the latest installation flow.

Feature content is installed in RAM. The required memory is allocated in the Kernel Working Buffer. This includes code, resources, static fields, and internal structures. When the Feature is uninstalled, allocated memory is reclaimed. When the Core Engine or the device restarts, the Kernel Working Buffer is reset; thus there is no persistent Feature.

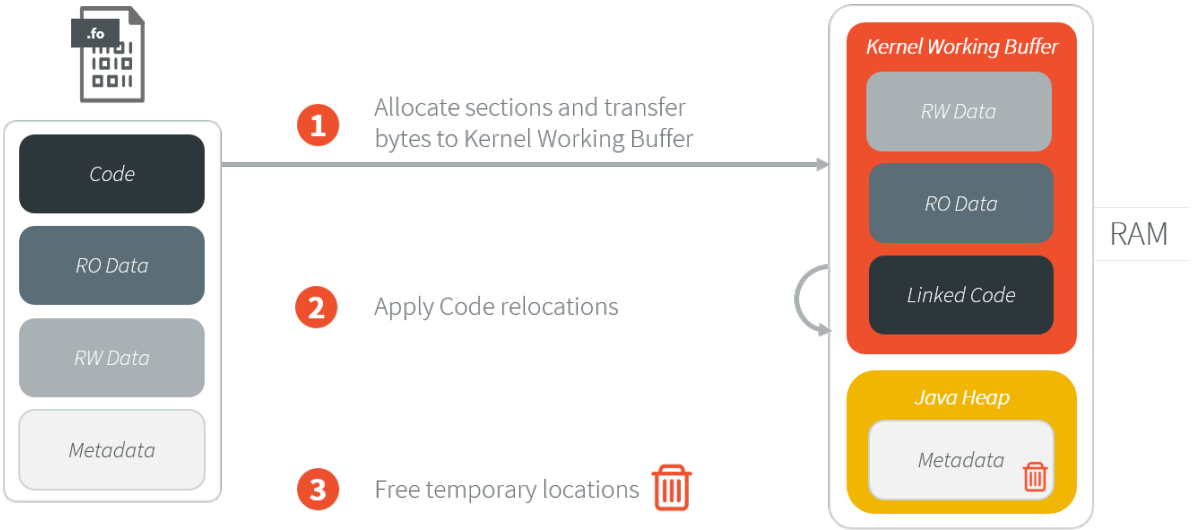


Fig. 190: In-Place Feature Installation Overview

The In-Place installation flow is described in the following sequence diagram:

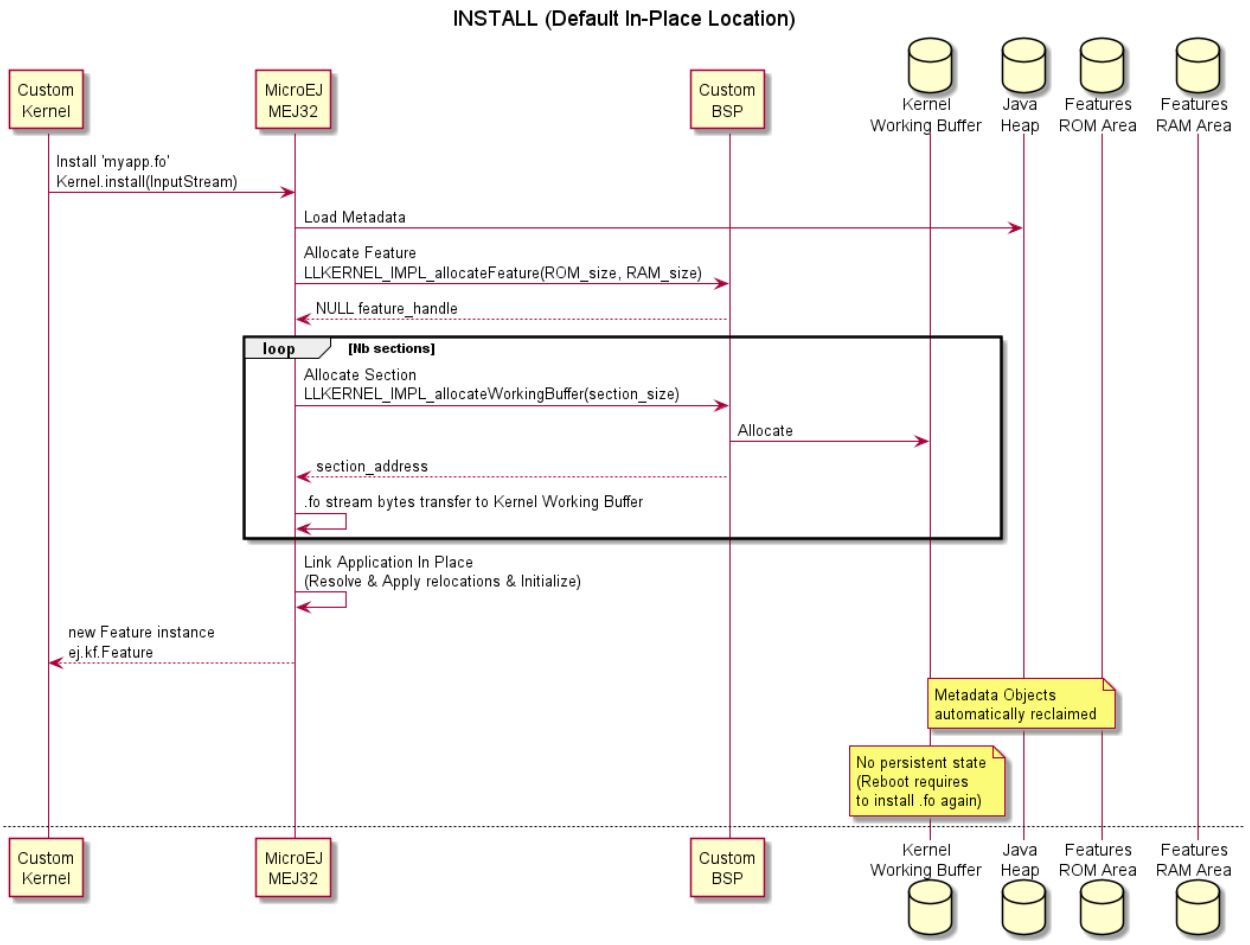


Fig. 191: In-Place Feature Installation Flow

The In-Place uninstallation flow is described in the following sequence diagram:

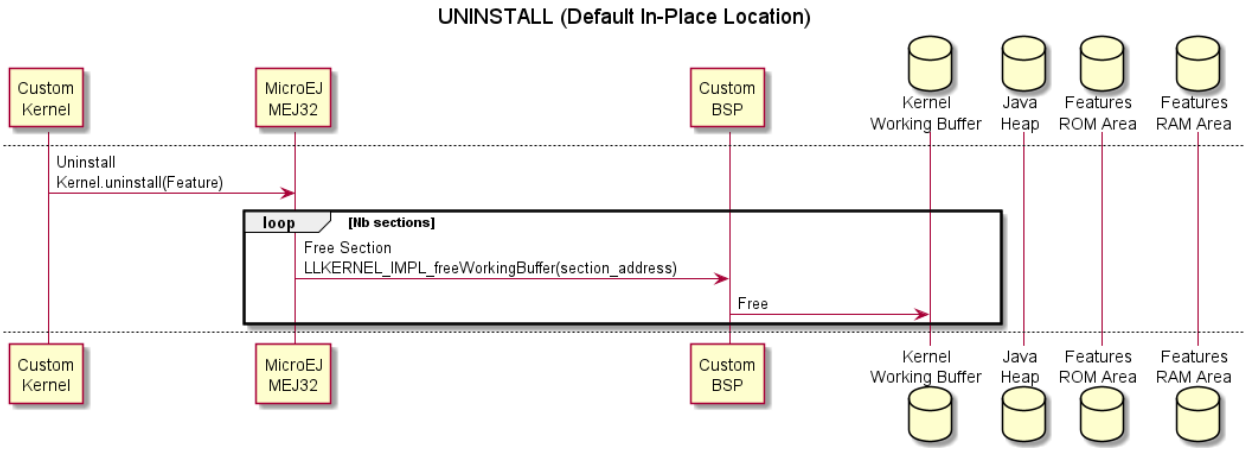


Fig. 192: In-Place Feature Uninstallation Flow

RAM Control

Note: This feature requires Architecture *8.1.0* or higher.

In a Multi-Sandbox environment, RAM Control automatically stops less critical Features when a more critical Feature cannot allocate new objects. See the *RAM Control: Feature Criticality* section of the *Kernel & Features Specification (KF)* for more details.

By default, RAM Control is disabled in the Core Engine. To enable it, set the property `com.microej.runtime.kf.ramcontrol.enabled` to `true` in the VEE Port configuration:

SDK 6

SDK 5

In the `configuration.properties` file:

```
com.microej.runtime.mjvm.com.microej.runtime.kf.ramcontrol.enabled=true
```

In the `mjvm/mjvm.properties` file of the VEE Port Configuration module:

```
com.microej.runtime.kf.ramcontrol.enabled=true
```

When RAM Control is enabled, all Foundation Libraries must declare their native resources using SNI (see `sni.h` header file). This is necessary for the automatic release of native resources when the Core Engine abruptly stops a Feature to recover heap memory. Foundation Libraries can no longer register native resources using the deprecated class `ej.lang.ResourceManager`. Attempting to do so will result in an exception being thrown.

3.6.10 Tiny-Sandbox

Principle

The Tiny-Sandbox capability of the Core Engine allows to build a Standalone Application optimized for size. This capability is suitable for environments requiring a small memory footprint.

Installation

Tiny-Sandbox is an option disabled by default. To enable the Tiny-Sandbox capability of the Core Engine, set the property `com.microej.runtime.capability` to `tiny` in the `configuration.properties` file (SDK 6) or in the `mjvm/mjvm.properties` file (SDK 5) of the VEE Port project. See the example below:

```
com.microej.runtime.capability=tiny
```

Note: In SDK 5, before *Architecture 8.1.0*, enabling the Tiny-Sandbox capability was done by setting the property `mjvm.standalone.configuration` in the `configuration.xml` file as follows:

```
<property name="mjvm.standalone.configuration" value="tiny"/>
```

See section *VEE Port Customization* for more info on the `configuration.xml` file.

Limitations

In addition to general *Limitations*:

- The maximum application code size (classes and methods) cannot exceed 256KB . This does not include application resources, immutable objects and internal strings which are not limited.
- The option SOAR > Debug > Embed all type names has no effect. Only the fully qualified names of types marked as required types are embedded.
- Incompatible with dynamic linkers enabling Address Space Layout Randomization (ASLR).

3.6.11 Foundation Libraries

Foundation Libraries are MicroEJ Core libraries that provide core runtime APIs or hardware-dependent functionalities. They allow Managed code to call native code.

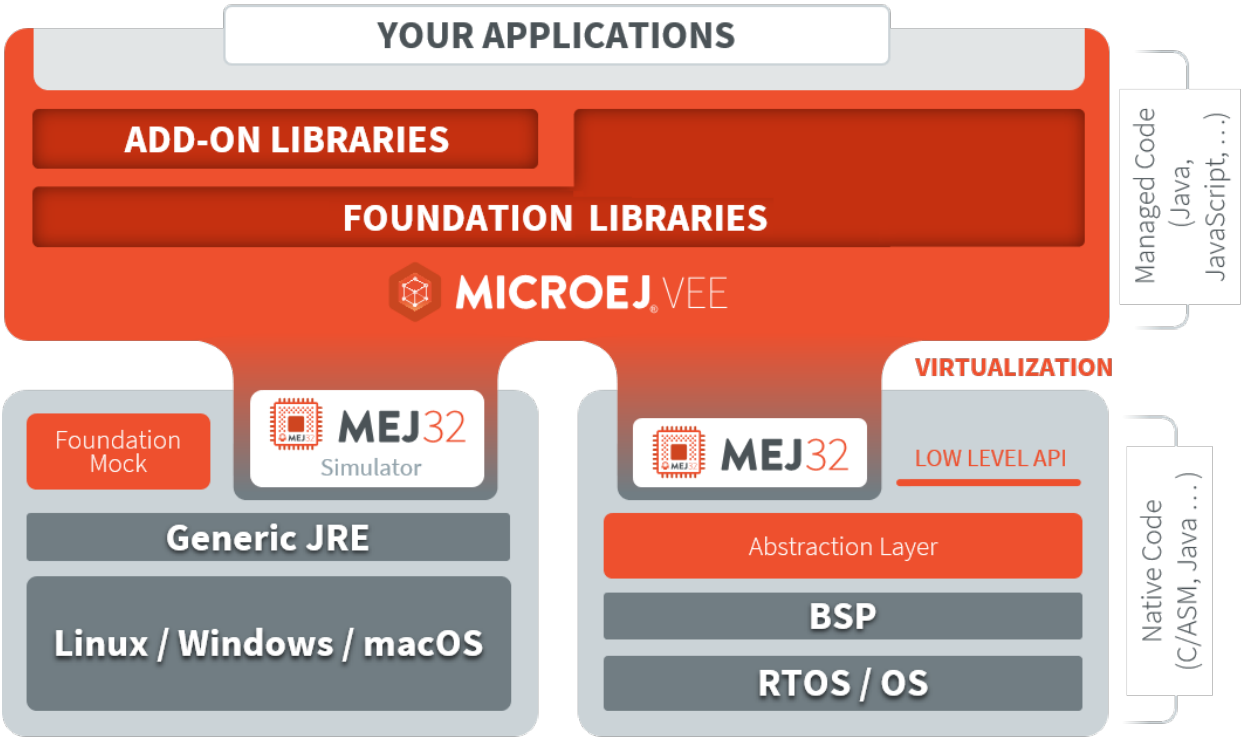


Fig. 193: MicroEJ Foundation Libraries and Add-On Libraries

Note: In SDK 5, A Foundation library is divided into 2 JARs: an API and an implementation. A Foundation library API is composed of a name and a 2 digits version (e.g. EDC-1.3) and follows the semantic versioning (http://semver.org) specification. A Foundation Library API only contains prototypes without code. Foundation Library implementations are provided by VEE Ports. From a MicroEJ Classpath, Foundation Library APIs dependencies are automatically mapped to the associated implementations provided by the VEE Port or the Virtual Device on which the application is being executed.

In SDK 6, a Foundation Library is a single JAR, containing the implementation. The API is composed of all the public classes and methods of the Library.

Foundation Libraries can be consumed by Applications and other Libraries. They are *automatically mapped to the capabilities* provided by the VEE Port or the Virtual Device used to run the Application.

MicroEJ provides a large set of *Foundation Libraries*, such as UI, Network, Audio, ... You can also create your own Foundation Library by going through the following steps described in the next sections:

1. Create the Library project
2. Create a Java class
3. Define C Header Files
4. Implement native code
5. Create the mock
6. Install the Library in the VEE Port

Create the Library Project

To create a Foundation Library, follow the instructions of the *Create a Project* page. Choose the `Library` type in the Project Wizards or the `Library Project Template` with the Command Line.

Create a Java Class

Once the Library project is created, you can now create a Java class which calls native code. Create the Java class `Sensor` in the `src/main/java/com/mycompany` folder of the Library project, with the following content:

```
package com.mycompany;

import java.io.IOException;

/**
 * Class providing a native method to access sensor value.
 * This method will be executed out of the Core Engine.
 */
public class Sensor {

   public static final int ERROR = -1;

   public static int getValue() throws IOException {
      int value = getSensorValue();
      if (value == ERROR) {
         throw new IOException("Unsupported sensor");
      }
      return value;
   }

   public static native int getSensorValue();
}
```

This class contains a native method `getSensorValue` which will be implemented by native code.

Define C Header Files

The header files define the prototype of the C functions. They must be added in the `src/main/include` folder of the Library project.

The C function name must respect the following pattern to be mapped to the Java method: `Java_<FQN>_<methodName>` , where `<FQN>` is the fully qualified name of the Java class with all `.` are replaced by `_` , and `methodName` is name of the Java method. Therefore, for the `getSensorValue` Java method of the `com.mycompany.Sensor` class, the C function must be called `Java_com_mycompany_Sensor_getSensorValue` .

Here is the content of the header file `LLMYLIB_impl.h` to create in the `src/main/include` folder of the Library project:

```
#ifndef LLMYLIB_IMPL
#define LLMYLIB_IMPL

/**
* @file
* @brief MicroEJ Sensor Low Level API
* @author My Company
* @version 1.0.0
*/

#include <stdint.h>

#ifdef __cplusplus
extern "C" {
#endif

#define LLMYLIB_IMPL_getSensorValue Java_com_mycompany_Sensor_getSensorValue

/*
* Returns the Sensor value
*/
uint32_t LLMYLIB_IMPL_getSensorValue();

#ifdef __cplusplus
}
#endif
#endif
```

The `#define` statement allows to separate the Java part and the C part. This is called the Low Level API of the Foundation Library. If the fully qualified name of the Java native method is updated, the C implementation code do not need to be updated. Refer to the *Low Level API Pattern* section for more information on Low Level APIs.

At this stage, the Library project structure should look like:

```
|- src/
|    |- main/
|    |    |- java/
|    |    |    |- com/
|    |    |    |    |- mycompany/
|    |    |    |    |    |- Sensor.java
|    |    |- include/
|    |    |    |- LLMYLIB_impl.h
|- build.gradle.kts
```

Implement Native Code

- In the C project, create a new File called `LLMYLIB_impl.c`.
- Add the C file to the compilation objects by adding it to the C Project configuration.
- Copy and paste the following code to the file:

```
#include "LLMYLIB_impl.h"
#include "sni.h"

/**
* @file
* @brief MicroEJ Sensor low level API
* @author My Company
* @version 1.0.0
*/
uint32_t LLMYLIB_IMPL_getSensorValue()
{
    return 42;
}
```

This file defines a basic C implementation of the `getSensorValue` function.

Create the Mock

If you need to use the Library in the Simulator, a Mock must be implemented. A Mock is a component that implements the native method in Java to use the Library in the Simulator. Refer to the *Mock* page to create the Mock for `Sensor` Library.

Once the Mock is created, it must be declared as a dependency of the Library, in the `build.gradle.kts` file:

```
dependencies {
    microejMock("com.mycompany:my-mock:1.0.0")
}
```

Install the Library in the VEE Port

Once the Library project is done, it must be installed in the VEE Port by adding it as a dependency in the `build.gradle.kts` file of the VEE Port project:

```
dependencies {
    api("com.mycompany:my-library:1.0.0")
}
```

When declaring the dependency with the `api` configuration, if *the VEE Port transitive resolution* is enabled, it is exposed transitively to the project depending on the VEE Port. If you do not want to expose the Library transitively, use the `implementaion` configuration instead:

```
dependencies {
    implementation("com.mycompany:my-library:1.0.0")
}
```

Test in an Application

In order to test the Library, it is recommended to create an Application that uses the VEE Port and calls the Sensor Library. The Application project must declare the VEE Port and the Library as dependencies in its `build.gradle.kts` file:

```
dependencies {
    implementation("com.mycompany:my-library:1.0.0")
    microejVee("com.mycompany:my-vee-port:1.0.0")
}
```

Then the main method of the Application must call the `Sensor` API:

```
package com.mycompany;

public class Main {

    public static void main(String[] args) {
        System.out.println("Sensor value: " + Sensor.getValue());
    }

}
```

You can now run the Application with the Simulator thanks to the `ruOnSimulator` Gradle task:

```
./gradlew runOnSimulator
```

The following trace shall appear on the standard output:

```
================ [ Initialization Stage ] ================
================ [ Launching on Simulator ] ================
Sensor value: 42
================ [ Completed Successfully ] ================
```

You can also build and deploy the Application on your device. The following trace shall appear on the standard output:

```
VM START
Sensor value: 42
VM END (exit code = 0)
```

Implement a Validation Testsuite

To validate the Foundation Library against the VEE Ports they will be used with, it is recommended to provide a *validation Testsuite*. It contains a bunch of Java tests that validate the Abstraction Layer implementing the Low Level APIs of the Foundation Library.

You can find examples of validation Testsuites in the VEE Port project template.

Extend the Link Process

Libraries can extend the Linking process using LSC files. Refer to the *MicroEJ Linker* page for more information.

These files must be added in the `src/main/lscf` folder of the Library project:

```
|- src/
|    |- main/
|    |    |- java/
|    |    |    |- com/
|    |    |    |    |- mycompany/
|    |    |    |    |    |- Sensor.java
|    |    |- include/
|    |    |    |- LLMYLIB_impl.h
|    |    |- lscf/
|    |    |    |- mylib.lscf
|- build.gradle.kts
```

Define Configurable Properties in LSC Files

The LSC files can use properties to define configurable values. For example, you may want to define a configurable heap size for your library:

```
<lscFragment>
   <defSymbol␣
↪name="__mylibrary__heap__size" value="${mylibrary.heap.size}" rootSymbol="true"/>
   <defSection name=".bss.mylibrary.heap" size="__mylibrary__heap__size" align="4"/>
   <defSymbol name=
↪"__mylibrary__heap__start" value="START(.bss.mylibrary.heap)" rootSymbol="true"/>
   <defSymbol␣
↪name="__mylibrary__heap__end" value="END(.bss.mylibrary.heap)" rootSymbol="true"/>
</lscFragment>
```

The property can be set by the Application which uses the Library, by defining the *Application Option* `mylibrary.heap.size`.

Set Default Values for the Configurable Properties

The LSC format allows to define a default value for the properties, by suffixing the property name with `:`, followed by the default value. For example, the following line defines the property `mylibrary.heap.size` with the default value `1024`:

```
<defSymbol␣
↪name="__mylibrary__heap__size" value="${mylibrary.heap.size:1024}" rootSymbol="true"/>
```

3.6.12 Native Interface Mechanisms

The Core Engine provides two ways to link MicroEJ Application code with native C code. The two ways are fully complementary, and can be used at the same time.

Shielded Plug (SP)

Principle

The Shielded Plug (SP) library provides data segregation with a clear publish-subscribe API. The data-sharing between modules uses the concept of shared memory blocks, with introspection. The database is made of blocks: chunks of RAM.

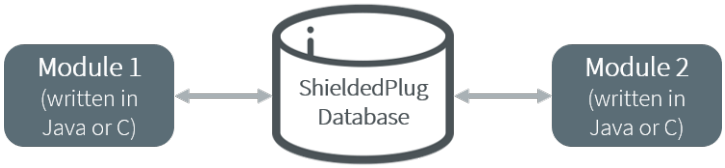


Fig. 194: A Shielded Plug Between Two Application (Java/C) Modules.

Documentation	Link
Java APIs	https://repository.microej.com/javadoc/microej_5.x/apis/ej/sp/package-summary.html
Specification	https://repository.microej.com/packages/ESR/ESR-SPE-0014-SP-2.0-B.pdf
Module	https://repository.microej.com/modules/ej/api/sp/

Functional Description

The usage of the Shielded Plug (SP) starts with the definition of a database. The implementation uses an XML file description to describe the database; the syntax follows the one proposed by the [SP] specification.

Once this database is defined, it can be accessed within the MicroEJ Application or the C application. The SP Foundation Library is accessible from the *[SP] API Module*. This library contains the classes and methods to read and write data in the database. The C header file sp.h available in the MicroEJ Platform source/include folder contains the C functions for accessing the database.

To embed the database in your binary file, the XML file description must be processed by the *SP compiler*. This compiler generates a binary file (.o) that will be linked to the overall application by the linker. It also generates two descriptions of the block ID constants, one in Java code and one in C code. These constants can be used by either the Java code or the C application modules.

Shielded Plug Compiler

A *MicroEJ tool* is available to launch the compiler. The tool name is `Shielded Plug Compiler`. It outputs:

- A description of the requested resources of the database as a binary file (`.o`) that will be linked to the overall application by the linker. It is an ELF format description that reserves both the necessary RAM and the necessary Flash memory for the Shielded Plug database.
- Two descriptions, one in Java code and one in C code, of the block ID constants to be used by either Java or C application modules.

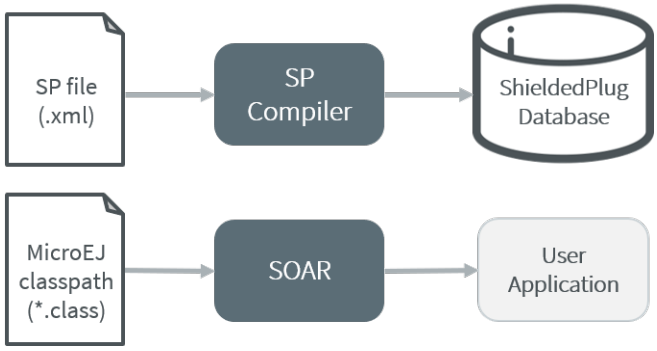


Fig. 195: Shielded Plug Compiler Process Overview

Example

Below is an example of using a database. The code that publishes the data is written in C, and the code that receives the data is written in Java code. The data is transferred using two memory blocks. `TEMP` is a scalar value, `THERMOSTAT` is a boolean.

Database Description

The database is described as follows:

```
<shieldedPlug>
   <database name="Forecast" id="0" immutable="true" version="1.0.0">
      <block id="1" name="TEMP" length="4" maxTasks="1"/>
      <block id="2" name="THERMOSTAT" length="4" maxTasks="1"/>
   </database>
</shieldedPlug>
```

Java Code

From the database description we can create an interface.

```
public interface Forecast {
   public static final int ID = 0;
   public static final int TEMP = 1;
   public static final int THERMOSTAT = 2;
}
```

Below is the task that reads the published temperature and controls the thermostat.

```
public void run(){
   ShieldedPlug database = ShieldedPlug.getDatabase(Forecast.ID);
   while (isRunning) {
      //reading the temperature every 30 seconds
      //and update thermostat status
      try {
         int temp = database.readInt(Forecast.TEMP);
         print(temp);
         //update the thermostat status
         database.writeInt(Forecast.THERMOSTAT,temp>tempLimit ? 0 : 1);
      }
      catch(EmptyBlockException e){
         print("Temperature not available");
      }
      sleep(30000);
   }
}
```

C Code

Here is a C header that declares the constants defined in the XML description of the database.

```
#define Forecast_ID 0
#define Forecast_TEMP 1
#define Forecast_THERMOSTAT 2
```

Below, the code shows the publication of the temperature and thermostat controller task.

```
void temperaturePublication() {
   ShieldedPlug database = SP_getDatabase(Forecast_ID);
   int32_t temp = temperature();
   SP_write(database, Forecast_TEMP, &temp);
}

void thermostatTask(){
   int32_t thermostatOrder;
   ShieldedPlug database = SP_getDatabase(Forecast_ID);
   while(1){
      SP_waitFor(database, Forecast_THERMOSTAT);
      SP_read(database, Forecast_THERMOSTAT, &thermostatOrder);
      if(thermostatOrder == 0) {
         thermostatOFF();
      }
```

(continues on next page)

(continued from previous page)

```
    else {
      thermostatON();
    }
  }
}
```

Dependencies

- LLSP_impl.h implementation (see *LLSP: Shielded Plug*).

Installation

The [SP] library and its relative tools are an optional feature of the VEE Port. The installation process is different in SDK 5 and SDK 6:

SDK 6

SDK 5

In the VEE Port configuration file, add the following property:

```
com.microej.runtime.shieldedplug.enabled=true
```

In the VEE Port configuration file, check Java to C Interface > Shielded Plug to install the library and its relative tools.

Use

The Shielded Plug API Module must be added to the project build file:

SDK 6 (build.gradle.kts)

SDK 5 (module.ivy)

```
implementation("ej.api:sp:2.0.4")
```

```
<dependency org="ej.api" name="sp" rev="2.0.4"/>
```

This library provides a set of options. Refer to the chapter *Standalone Application Options* which lists all available options.

MicroEJ Java H

Principle

This MicroEJ tool is useful for creating the skeleton of a C file, to which some Java native implementation functions will later be written. This tool helps prevent misses of some #include files, and helps ensure that function signatures are correct.

Functional Description

MicroEJ Java H tool takes as input one or several Java class files (*.class) from directories and / or JAR files. It looks for Java native methods declared in these class files, and generates a skeleton(s) of the C file(s).

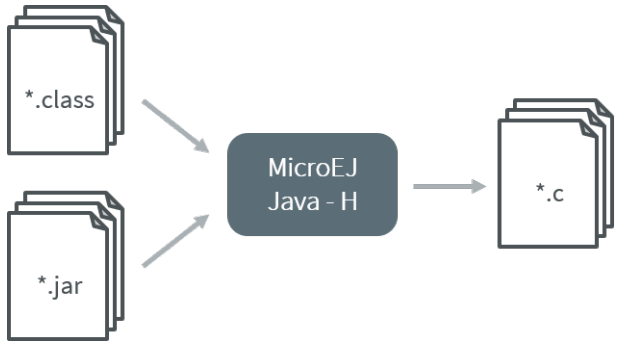


Fig. 196: MicroEJ Java H Process

Dependencies

No dependency.

Installation

SDK 6

SDK 5

In SDK 6, the MicroEJ Java H tool is enabled by default. It can be disabled in the VEE Port configuration file, by adding the following property:

```
com.microej.runtime.extensible.MicroEJavah.enabled=false
```

This is an additional tool. In the VEE Port configuration file, check Java to C Interface > MicroEJ Java H to install the tool.

Use

This chapter explains the MicroEJ tool options.

MicroEJ Java H Tool (SDK 6)

In SDK 6, the **MicroEJ Java H** tool is executed using the Gradle `execTool` task.

Run the following command from a project containing `microejVee()` dependency.

```
./gradlew execTool \
--name=ejavah \
--toolProperty=ejavah.
↪natives.path="<absolute-path-to-classes>\classes;<absolute-path-to-jar>\mylib.jar"
```

- `--name=ejavah` : Specifies the MicroEJ Java H tool.
- `ejavah.natives.path` : Defines the Java classpath to scan for native methods. It can contain directories with `.class` files, or paths to `.jar` files. Entries must be separated using the system-dependent path-separator character as defined in `File.pathSeparator` . Usualy `;` on Windows and `:` on Linux.

Generated `.h` header files are located in: `build\output\include`

When the execution is successful, the following type of log is displayed:

```
Info: com.microej.Main will not be generated as C SNI because it does not contain any native
Info: com.
↪microej.package-info will not be generated as C SNI because it does not contain any native
Info: com_microej_NativeExample.h successfully generated in build\output
```

3.6.13 External Resources Loader

Functional Description

The External Resources Loader is an optional module. When not installed, only *internal resources* are available for the MicroEJ Application. When the External Resources Loader is installed, the Core Engine tries first to retrieve the expected resource from its available list of internal resources, before asking the BSP to load it (using `LLEXT_RES_impl.h` functions).

See *Application Resources* for more information on how to declare external resources depending on its kind (raw resources, images, fonts, NLS).

Implementations

External Resources Loader module provides some Low Level API (LLEXT_RES) to let the BSP manage the external resources.

Open a Resource

The LLAPI to implement in the BSP are listed in the header file `LLEXT_RES_impl.h` . First, the framework tries to open an external resource using the `open` function. This function receives the resources path as a parameter. This path is the absolute path of the resource from the MicroEJ Application classpath (the MicroEJ Application source base directory). For example, when the resource is located here: `com.mycompany.myapplication.resource.MyResource.txt` , the given path is: `com/mycompany/myapplication/resource/MyResource.txt` .

The external resources loader implementation should, when possible, lock the resource when it is opened. Any modification of an opened resource may not be properly handled by the application.

Resource Identifier

This `open` function has to return a unique ID (positive value) for the external resource, or returns an error code (negative value). This ID will be used by the framework to manipulate the resource (read, seek, close, etc.).

Several resources can be opened at the same time. The BSP does not have to return the same identifier for two resources living at the same time. However, it can return this ID for a new resource as soon as the old resource is closed.

Resource Offset

The BSP must hold an offset for each opened resource. This offset must be updated after each call to `read` and `seek`.

Resource Inside the CPU Address Space Range

An external resource can be programmed inside the CPU address space range. This memory (or a part of memory) is not managed by the SOAR and so the resources inside are considered as external.

Most of the time the content of an external resource must be copied in a memory inside the CPU address space range in order to be accessible by the MicroEJ algorithms (draw an image etc.). However, when the resource is already inside the CPU address space range, this copy is useless. The function `LLEXT_RES_getBaseAddress` must return a valid CPU memory address in order to avoid this copy. The MicroEJ algorithms are able to target the external resource bytes without using the other `LLEXT_RES` APIs such as `read`, `mark` etc.

External Resources Folder

When working with *Application Resources* declared as external resources, the Application build process will output those external resources in a dedicated output folder named `externalResources/`.

This folder gathers all the *Application Resources* that should be deployed on the device. This folder not only contains the pre-processed resources but also all the other external resources from the project that are not pre-processed:

- Pre-processed resources: resources formatted by MICROEJ SDK to optimize footprint and/or parsing/processing. That is for example the case of *MicroUI Fonts*, *NLS Resources* and *MicroUI Images embedded in RAW format*.
- Unprocessed resources: resources embedded as-is, like *Raw Resources* or *Images* with no output format defined (e.g. image embedded as a `.png`).

The location of the `externalResources/` folder is different between SDK 5 and SDK 6:

SDK 6

SDK 5

The externalResources/ folder is located in the build/application/object folder of the application project.

It is generated when building the application for the device (*buildApplicationObjectFile*).

The externalResources/ folder is located in the output folder of the application project. This folder is defined in the Execution tab of the *MicroEJ Launch* configuration (e.g. Example-ExternalResourceLoader/com.microej.externalresourceloader.ExternalImages/externalResources).

Dependencies

- LLEXT_RES_impl.h implementation (see *LLEXT_RES: External Resources Loader*).

Installation

The External Resources Loader is an additional module. The installation process is different in SDK 5 and SDK 6:

SDK 6

SDK 5

In the VEE Port configuration file, add the following property:

```
com.microej.runtime.externalresourceloader.enabled=true
```

In the VEE Port configuration file, check External Resources Loader to install this module.

Use

The External Resources Loader is automatically used when the MicroEJ Application tries to open an external resource.

A simple implementation of the External Resources Loader is available on GitHub: Example-ExternalResourceLoader.

On Simulator

The *Application Resources* provided by the application project, pre-processed and/or unprocessed, are automatically made available to the application during the simulation.

On Device

The external resources must be deployed on the device before they can be consumed by the application.

After that, the external resources can be updated (without re-building the application) with the following development flow:

1. Update the external resources in the application project.
2. Test the application with the updated resources on the simulator.

3. When good, re-build the application, and collect the external resources in the output folder (the build is required to trigger the processing of the resources).
4. Deploy the external resources on the device (e.g. copy them to the file system of the device: removable SD card, USB mass storage, …).

3.6.14 Serial Communications

MicroEJ provides some Foundation Libraries to instantiate some communications with external devices. Each communication method has its own library. A global library called ECOM provides support for abstract communication streams (communication framework only), and a generic devices manager.

ECOM

Warning: This chapter describes the Foundation Library ECOM-1.1 .

ECOM-1.1 is discontinued since *Architecture 8.0.0*.

Principle

The Embedded COMmunication Foundation Library (ECOM) is a generic communication library with abstract communication stream support (a communication framework only). It allows you to open and use streams on communication devices such as a COMM port.

This library also provides a device manager, including a generic device registry and a notification mechanism, which allows plug&play-based applications.

This library does not provide APIs to manipulate some specific options for each communication method, but it does provide some generic APIs which abstract the communication method. After the opening step, the MicroEJ Application can use every communications method (COMM, USB etc.) as generic communication in order to easily change the communication method if needed.

Functional Description

The diagram below shows the overall process to open a connection on a hardware device.

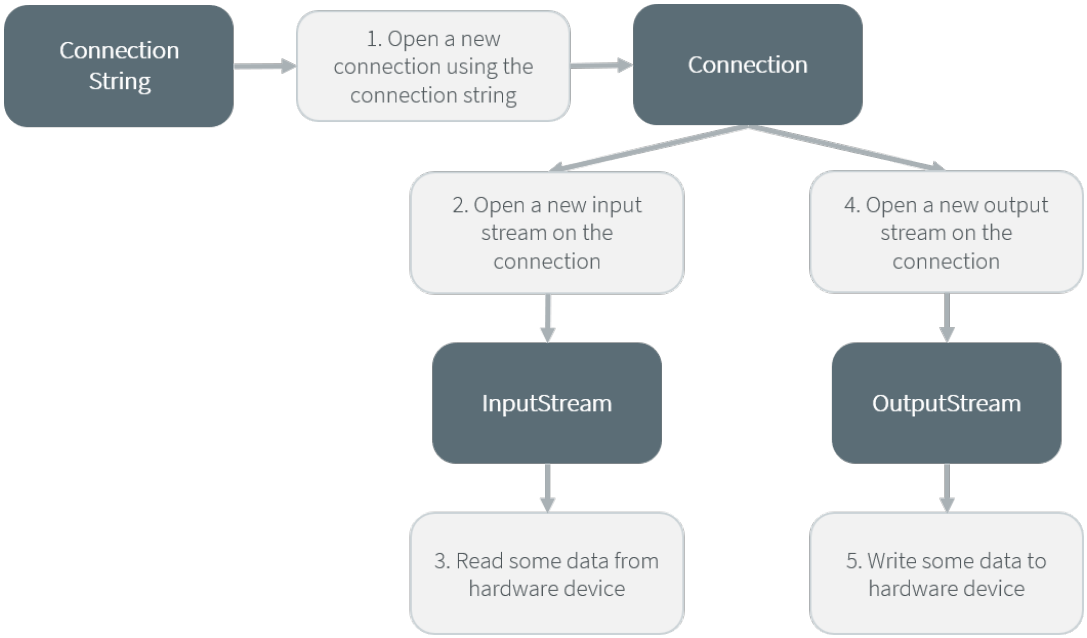


Fig. 197: ECOM Flow

1. Step 1 consists of opening a connection on a hardware device. The connection kind and its configuration are fixed by the String parameter of the method Connector.open(String).
2. Step 2 consists of opening an InputStream on the connection. This stream allows the MicroEJ Application to access the "RX" feature of the hardware device.
3. Step 3 consists of using the InputStream APIs to receive in the MicroEJ Application all hardware device data.
4. Step 4 consists of opening an OutputStream on the connection. This stream allows the MicroEJ Application to access the "TX" feature of the hardware device.
5. Step 5 consists of using the OutputStream APIs to transmit some data from the MicroEJ Application to the hardware device.

   Note that steps 2 and 4 may be performed in parallel, and do not depend on each other.

Device Management API

A device is defined by implementing ej.ecom.Device. It is identified by a name and a descriptor (ej.ecom.HardwareDescriptor), which is composed of a set of MicroEJ properties. A device can be registered/unregistered in the ej.ecom.DeviceManager.

A device registration listener is defined by implementing ej.ecom.RegistrationListener. When a device is registered to or unregistered from the device manager, listeners registered for the device type are notified. The notification mechanism is done in a dedicated thread. The mechanism can be enabled or disabled (see *Standalone Application Options*).

Dependencies

No dependency.

Installation

ECOM Foundation Library is an additional library.

SDK 6

SDK 5

In the VEE Port build file, add a dependency to the ?? pack:

```
microejPack("com.microej.pack.??:??:??")
```

In the VEE Port configuration file, check Serial Communication > ECOM to install the library and its relative tools.

Use

The ECOM API Module must be added to the Application project build file to use the ECOM library:

SDK 6 (build.gradle.kts)

SDK 5 (module.ivy)

```
implementation("ej.api:ecom:1.1.4")
```

```
<dependency org="ej.api" name="ecom" rev="1.1.4"/>
```

This Foundation Library is always required when developing a MicroEJ Application which communicates with some external devices. It is automatically embedded as soon as a sub communication library is added in the classpath.

ECOM Comm

Warning: This chapter describes the Foundation Library ECOM-COMM-1.1 .

ECOM-COMM-1.1 is deprecated in favor of ECOM-COMM-2.0 and has been removed from *Architecture 8.0.0*. See *Migrate ECOM-COMM Module* for more details.

Principle

The ECOM Comm Java library provides support for serial communication. ECOM Comm extends ECOM to allow stream communication via serial communication ports (typically UARTs). In the MicroEJ Application, the connection is established using the Connector.open() method. The returned connection is a ej.ecom.io.CommConnection, and the input and output streams can be used for full duplex communication.

The use of ECOM Comm in a custom platform requires the implementation of an UART driver. There are two different modes of communication:

- In Buffered mode, ECOM Comm manages software FIFO buffers for transmission and reception of data. The driver copies data between the buffers and the UART device.
- In Custom mode, the buffering of characters is not managed by ECOM Comm. The driver has to manage its own buffers to make sure no data is lost in serial communications because of buffer overruns.

  This ECOM Comm implementation also allows dynamic add or remove of a connection to the pool of available connections (typically hot-plug of a USB Comm port).

Functional Description

The ECOM Comm process respects the ECOM process. Please refer to the illustration *"ECOM flow"*.

Component Architecture

The ECOM Comm C module relies on a native driver to perform actual communication on the serial ports. Each port can be bound to a different driver implementation, but most of the time, it is possible to use the same implementation (i.e. same code) for multiple ports. Exceptions are the use of different hardware UART types, or the need for different behaviors.

Five C header files are provided:

- LLCOMM_impl.h

  Defines the set of functions that the driver must implement for the global ECOM comm stack, such as synchronization of accesses to the connections pool.

- LLCOMM_BUFFERED_CONNECTION_impl.h

  Defines the set of functions that the driver must implement to provide a Buffered connection

- LLCOMM_BUFFERED_CONNECTION.h

  Defines the set of functions provided by ECOM Comm that can be called by the driver (or other C code) when using a Buffered connection

- LLCOMM_CUSTOM_CONNECTION_impl.h

  Defines the set of functions that the driver must implement to provide a Custom connection

- LLCOMM_CUSTOM_CONNECTION.h

  Defines the set of functions provided by ECOM Comm that can be called by the driver (or other C code) when using a Custom connection

  The ECOM Comm drivers are implemented using standard LLAPI features. The diagram below shows an example of the objects (both Java and C code) that exist to support a Buffered connection.

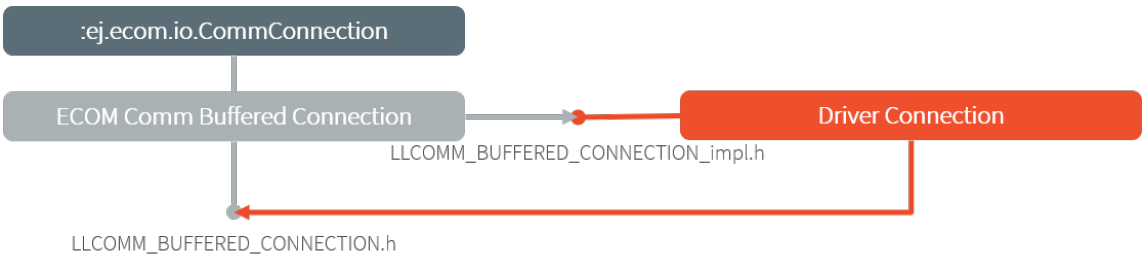


Fig. 198: ECOM Comm components

The connection is implemented with three objects[1] :

- The Java object used by the application; an instance of ej.ecom.io.CommConnection
- The connection object within the ECOM Comm C module
- The connection object within the driver

  Each driver implementation provides one or more connections. Each connection typically corresponds to a physical UART.

Comm Port Identifier

Each serial port available for use in ECOM Comm can be identified in three ways:

- An application port number. This identifier is specific to the application, and should be used to identify the data stream that the port will carry (for example, “debug traces” or “GPS data”).
- A platform port number. This is specific to the platform, and may directly identify a hardware device[2].
- A platform port name. This is mostly used for dynamic connections or on platforms having a file-system based device mapping.

  When the Comm Port is identified by a number, its string identifier is the concatenation of “com” and the number (e.g. com11).

Application Port Mapping

The mapping from application port numbers to platform ports is done in the application launch configuration. This way, the application can refer only to the application port number, and the data stream can be directed to the matching I/O port on different versions of the hardware.

Ultimately, the application port number is only visible to the application. The platform identifier will be sent to the driver.

[1] This is a conceptual description to aid understanding - the reality is somewhat different, although that is largely invisible to the implementor of the driver.

[2] Some drivers may reuse the same UART device for different ECOM ports with a hardware multiplexer. Drivers can even treat the platform port number as a logical id and map the ids to various I/O channels.

Opening Sequence

The following flow chart explains Comm Port opening sequence according to the given Comm Port identifier.

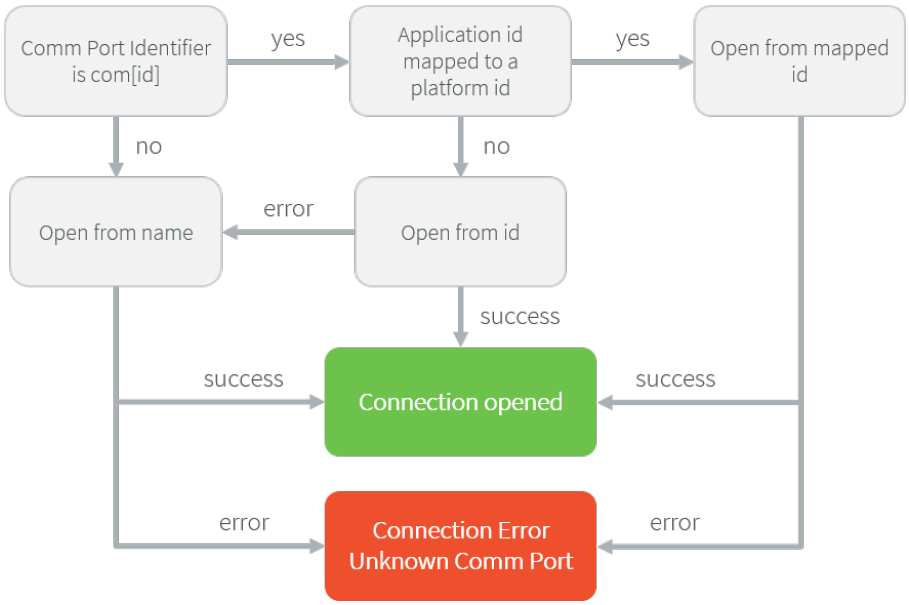


Fig. 199: Comm Port Open Sequence

Dynamic Connections

The ECOM Comm stack allows to dynamically add and remove connections from the *Driver API*. When a connection is added, it can be immediately open by the application. When a connection is removed, the connection cannot be open anymore and java.io.IOException is thrown in threads that are using it.

In addition, a dynamic connection can be registered and unregistered in ECOM device manager (see *Device Management API*). The registration mechanism is done in dedicated thread. It can be enabled or disabled, see *Standalone Application Options*.

A removed connection is alive until it is closed by the application and, if enabled, unregistered from ECOM device manager. A connection is effectively uninstalled (and thus eligible to be reused) only when it is released by the stack.

The following sequence diagram shows the lifecycle of a dynamic connection with ECOM registration mechanism enabled.

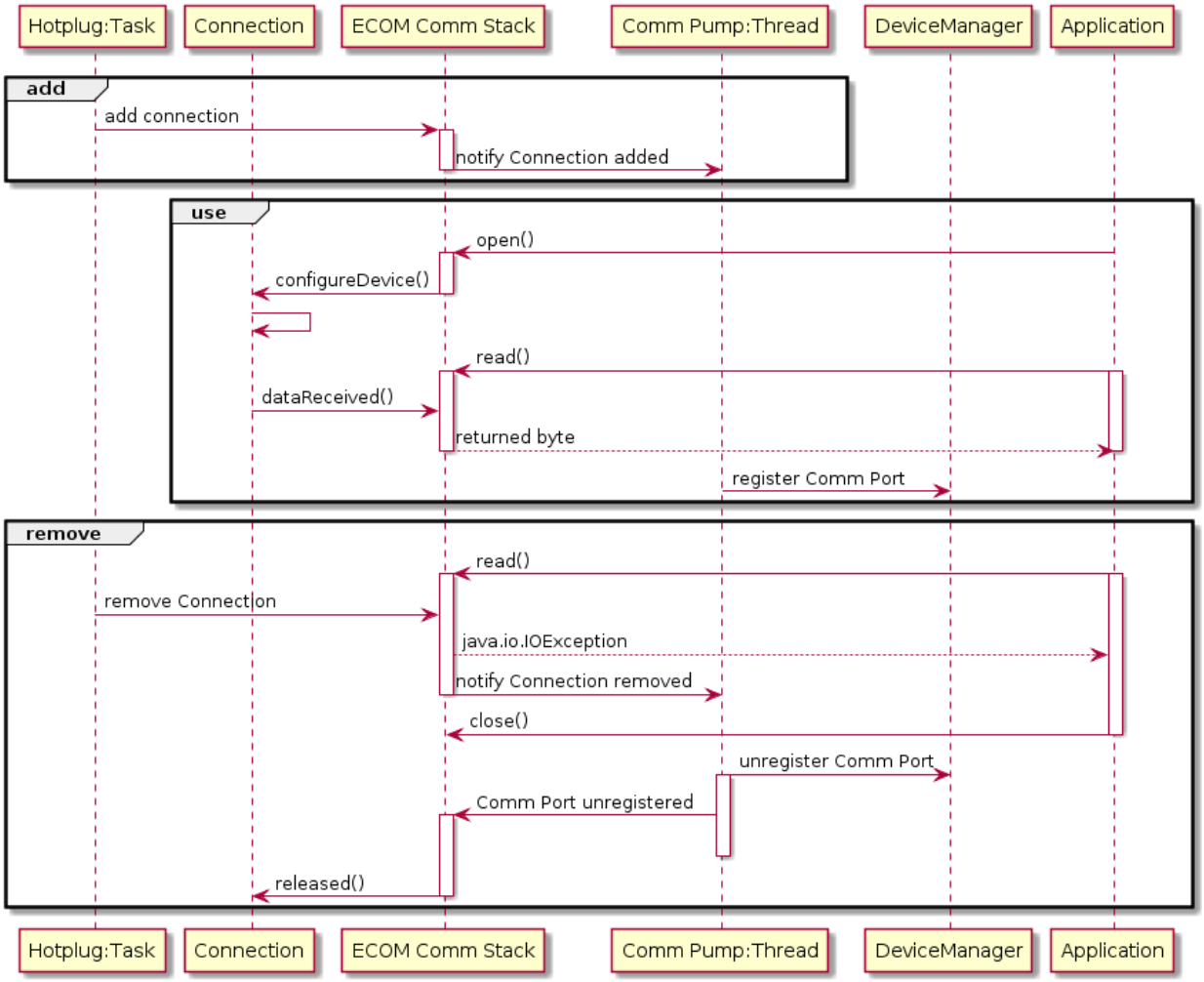


Fig. 200: Dynamic Connection Lifecycle

Java API

Opening a connection is done using ej.ecom.io.Connector.open(String url). The connection string (the `url` parameter) must start with "comm:", followed by the Comm port identifier, and a semicolon-separated list of options. Options are the baudrate, the parity, the number of bits per character, and the number of stop bits:

- baudrate=n (9600 by default)
- bitsperchar=n where n is in the range 5 to 9 (8 by default)
- stopbits=n where n is 1, 2, or 1.5 (1 by default)
- parity=x where x is odd, even or none (none by default)

All of these are optional. Illegal or unrecognized parameters cause an IllegalArgumentException.

Driver API

The ECOM Comm Low Level API is designed to allow multiple implementations (e.g. drivers that support different UART hardware) and connection instances (see Low Level API Pattern chapter). Each ECOM Comm driver defines a data structure that holds information about a connection, and functions take an instance of this data structure as the first parameter.

The name of the implementation must be set at the top of the driver C file, for example[3]:

```
#define LLCOMM_BUFFERED_CONNECTION MY_LLCOMM
```

This defines the name of this implementation of the LLCOMM_BUFFERED_CONNECTION interface to be MY_LLCOMM.

The data structure managed by the implementation must look like this:

```
typedef struct MY_LLCOMM{
   struct LLCOMM_BUFFERED_CONNECTION header;
   // extra data goes here
} MY_LLCOMM;

void MY_LLCOMM_new(MY_LLCOMM* env);
```

In this example the structure contains only the default data, in the header field. Note that the header must be the first field in the structure. The name of this structure must be the same as the implementation name (MY_LLCOMM in this example).

The driver must also declare the "new" function used to initialize connection instances. The name of this function must be the implementation name with _new appended, and it takes as its sole argument a pointer to an instance of the connection data structure, as shown above.

The driver needs to implement the functions specified in the LLCOMM_impl.h file and for each kind of connection, the LLCOMM_BUFFERED_CONNECTION_impl.h (or LLCOMM_CUSTOM_CONNECTION_impl.h) file.

The driver defines the connections it provides by adding connection objects using LLCOMM_addConnection . Connections can be added to the stack as soon as the LLCOMM_initialize function is called. Connections added during the call of the LLCOMM_impl_initialize function are static connections. A static connection is registered to the ECOM registry and cannot be removed. When a connection is dynamically added outside the Core Engine task context, a suitable reentrant synchronization mechanism must be implemented (see LLCOMM_IMPL_syncConnectionsEnter and LLCOMM_IMPL_syncConnectionsExit).

When opening a port from the MicroEJ Application, each connection declared in the connections pool will be asked about its platform port number (using the getPlatformId method) or its name (using the getName method) depending on the requested port identifier. The first matching connection is used.

The life of a connection starts with the call to getPlatformId() or getName() method. If the the connection matches the port identifier, the connection will be initialized, configured and enabled. Notifications and interrupts are then used to keep the stream of data going. When the connection is closed by the application, interrupts are disabled and the driver will not receive any more notifications. It is important to remember that the transmit and receive sides of the connection are separate Java stream objects, thus, they may have a different life cycle and one side may be closed long before the other.

[3] The following examples use Buffered connections, but Custom connections follow the same pattern.

The Buffered Comm Stream

In Buffered mode, two buffers are allocated by the driver for sending and receiving data. The ECOM Comm C module will fill the transmit buffer, and get bytes from the receive buffer. There is no flow control.

When the transmit buffer is full, an attempt to write more bytes from the MicroEJ Application will block the thread trying to write, until some characters are sent on the serial line and space in the buffer is available again.

When the receive buffer is full, characters coming from the serial line will be discarded. The driver must allocate a buffer big enough to avoid this, according to the UART baudrate, the expected amount of data to receive, and the speed at which the application can handle it.

The Buffered C module manages the characters sent by the application and stores them in the transmit buffer. On notification of available space in the hardware transmit buffer, it handles removing characters from this buffer and putting them in the hardware buffer. On the other side, the driver notifies the C module of data availability, and the C module will get the incoming character. This character is added to the receive buffer and stays there until the application reads it.

The driver should take care of the following:

- Setting up interrupt handlers on reception of a character, and availability of space in the transmit buffer. The C module may mask these interrupts when it needs exclusive access to the buffers. If no interrupt is available from the hardware or underlying software layers, it may be faked using a polling thread that will notify the C module.
- Initialization of the I/O pins, clocks, and other things needed to get the UART working.
- Configuration of the UART baudrate, character size, flow control and stop bits according to the settings given by the C module.
- Allocation of memory for the transmit and receive buffers.
- Getting the state of the hardware: is it running, is there space left in the TX and RX hardware buffers, is it busy sending or receiving bytes?

The driver is notified on the following events:

- Opening and closing a connection: the driver must activate the UART and enable interrupts for it.
- A new byte is waiting in the transmit buffer and should be copied immediately to the hardware transmit unit. The C module makes sure the transmit unit is not busy before sending the notification, so it is not needed to check for that again.

The driver must notify the C module on the following events:

- Data has arrived that should be added to the receive buffer (using the LLCOMM_BUFFERED_CONNECTION_dataReceived function)
- Space available in the transmit buffer (using the LLCOMM_BUFFERED_CONNECTION_transmitBufferReady function)

The Custom Comm Stream

In custom mode, the ECOM Comm C module will not do any buffering. Read and write requests from the application are immediately forwarded to the driver.

Since there is no buffer on the C module side when using this mode, the driver has to define a strategy to store received bytes that were not handed to the C module yet. This could be a fixed or variable side FIFO, the older received but unread bytes may be dropped, or a more complex priority arbitration could be set up. On the transmit side, if the driver does not do any buffering, the thread waiting to send something will be blocked and wait for the UART to send all the data.

In Custom mode flow control (eg. RTS/CTS or XON/XOFF) can be used to notify the device connected to the serial line and so avoid losing characters.

BSP File

The ECOM Comm C module needs to know, when the MicroEJ Application is built, the name of the implementation. This mapping is defined in a BSP definition file. The name of this file must be `bsp.xml` and must be written in the ECOM comm module configuration folder (near the `ecom-comm.xml` file). In previous example the `bsp.xml` file would contain:

Listing 31: ECOM Comm Driver Declaration (bsp.xml)

```
<bsp>
   <nativeImplementation
      name="MY_LLCOMM"
      nativeName="LLCOMM_BUFFERED_CONNECTION"
   />
</bsp>
```

where `nativeName` is the name of the interface, and `name` is the name of the implementation.

XML File

The Java platform has to know the maximum number of Comm ports that can be managed by the ECOM Comm stack. It also has to know each Comm port that can be mapped from an application port number. Such Comm port is identified by its platform port number and by an optional nickname (The port and its nickname will be visible in the MicroEJ launcher options, see *Standalone Application Options*).

A XML file is so required to configure the Java platform. The name of this file must be `ecom-comm.xml` . It has to be stored in the module configuration folder (see *Installation*).

This file must start with the node `<ecom>` and the sub node `<comms>` . It can contain several time this kind of line: `<comm platformId="A_COMM_PORT_NUMBER" nickname="A_NICKNAME"/>` where:

- `A_COMM_PORT_NUMBER` refers the Comm port the Java platform user will be able to use (see *Application Port Mapping*).
- `A_NICKNAME` is optional. It allows to fix a printable name of the Comm port.

The `maxConnections` attribute indicates the maximum number of connections allowed, including static and dynamic connections. This attribute is optional. By default, it is the number of declared Comm Ports.

Example:

Listing 32: ECOM Comm Module Configuration (ecom-comm.xml)

```
<ecom>
   <comms maxConnections="20">
      <comm platformId="2"/>
      <comm platformId="3" nickname="DB9"/>
      <comm platformId="5"/>
   </comms>
</ecom>
```

First Comm port holds the port 2, second "3" and last "5". Only the second Comm port holds a nickname "DB9".

ECOM Comm Mock

In the simulation environment, no driver is required. The ECOM Comm mock handles communication for all the serial ports and can redirect each port to one of the following:

- An actual serial port on the host computer: any serial port identified by your operating system can be used. The baudrate and flow control settings are forwarded to the actual port.
- A TCP socket. You can connect to a socket on the local machine and use netcat or telnet to see the output, or you can forward the data to a remote device.
- Files. You can redirect the input and output each to a different file. This is useful for sending precomputed data and looking at the output later on for offline analysis.

  When using the socket and file modes, there is no simulation of an UART baudrate or flow control. On a file, data will always be available for reading and will be written without any delay. On a socket, you can reach the maximal speed allowed by the network interface.

Dependencies

- ECOM (see *Serial Communications*).
- `LLCOMM_impl.h` and `LLCOMM_xxx_CONNECTION_impl.h` implmentations (see *LLCOMM: Serial Communications*).

Installation

ECOM-Comm Java library is an additional library. In the platform configuration file, check Serial Communication > ECOM-COMM to install it. When checked, the xml file `ecom-comm/ecom-comm.xml` is required during platform creation to configure the module (see *XML File*).

Use

The ECOM Comm API Module must be added to the *module.ivy* of the MicroEJ Application project to use the ECOM Comm library.

```
<dependency org="ej.api" name="ecom-comm" rev="1.1.4"/>
```

This Foundation Library is always required when developing a MicroEJ Application which communicates with some external devices using the serial communication mode.

This library provides a set of options. Refer to the chapter *Standalone Application Options* which lists all available options.

3.6.15 Graphical User Interface

Note: This chapter describes the current Graphical User Interface version 3 , provided by UI Pack version 14.0.0 or higher. The UI Pack *Changelog* and a *Migration Guide* are provided at the end of this chapter.

- If you are using the former Graphical User Interface version 3 provided by MicroEJ UI Pack version 13.x , please refer to this MicroEJ Documentation Archive.
- If you are using the former Graphical User Interface version 2 provided by MicroEJ UI Pack version up to 12.1.x , please refer to this MicroEJ Documentation Archive.

Principle

The User Interface Extension features one of the fastest graphics engines, associated with a unique int-based event management system.

This chapter describes the *UI3* notions, available since MicroEJ Architecture UI pack 13.0.0 and higher: MicroUI 3.0, Front Panel v6, Abstraction Layer APIs LLUI_xxx , etc.

The diagram below shows a simplified view of the components involved in the provisioning of User Interface Extension.

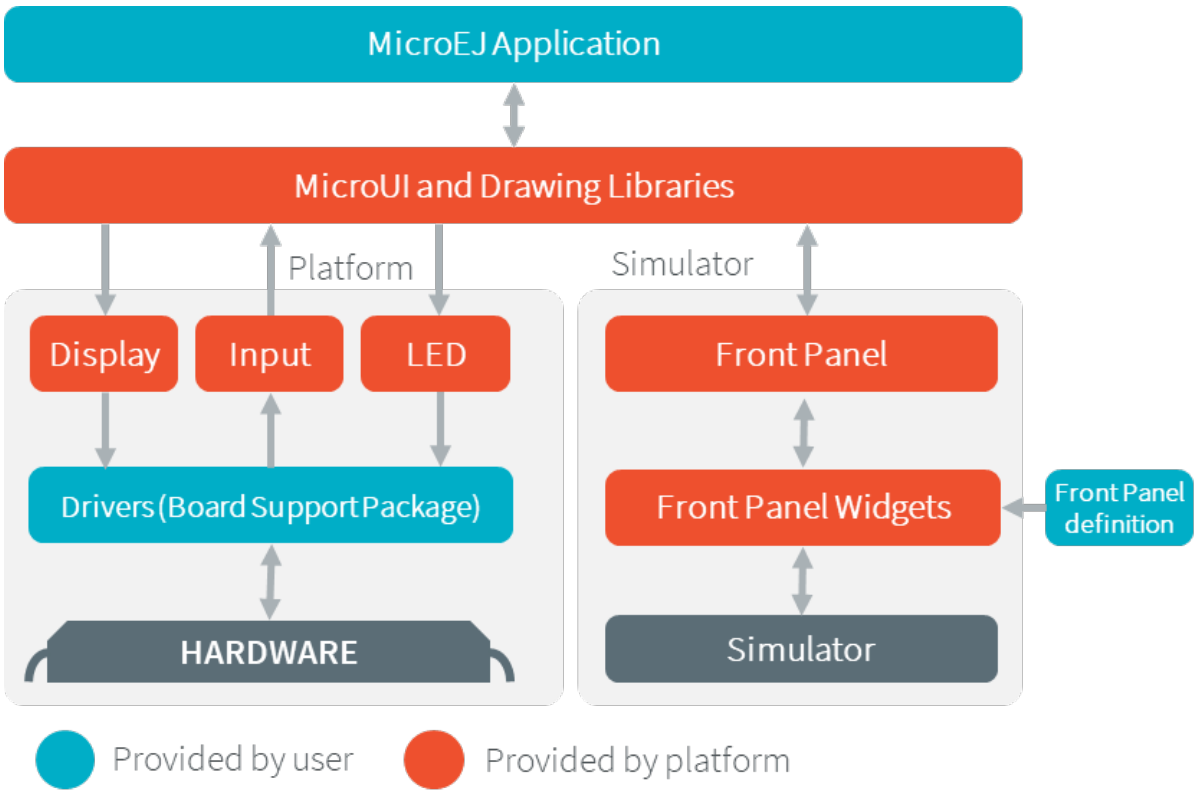


Fig. 201: Overview

The modules responsible to manage the Display, the Input and the LED are respectively called *Display module*, *Input module* and *LED module*. These three low-level parts connect MicroUI library to the user-supplied drivers code (coded in C). The drivers can use hardware accelerators like DMA and GPU to perform specific actions (buffers copy, drawings, etc.).

The MicroEJ Simulator provides all features of MicroUI library. The three modules are grouped together in a module called *Front Panel*. The Front Panel is supplied with a set of software widgets that generically support a range of input devices such as buttons, joysticks and touchscreens, and output devices such as displays and LEDs. With the help of the Front Panel Designer tool that forms part of the MicroEJ Workbench the user must define a Front Panel mock-up using these widgets.

The Display module also manages fonts and images. The fonts and images are pre-processed before compiling the application. The following diagram depicts the components involved in its design, along with the provided tools:

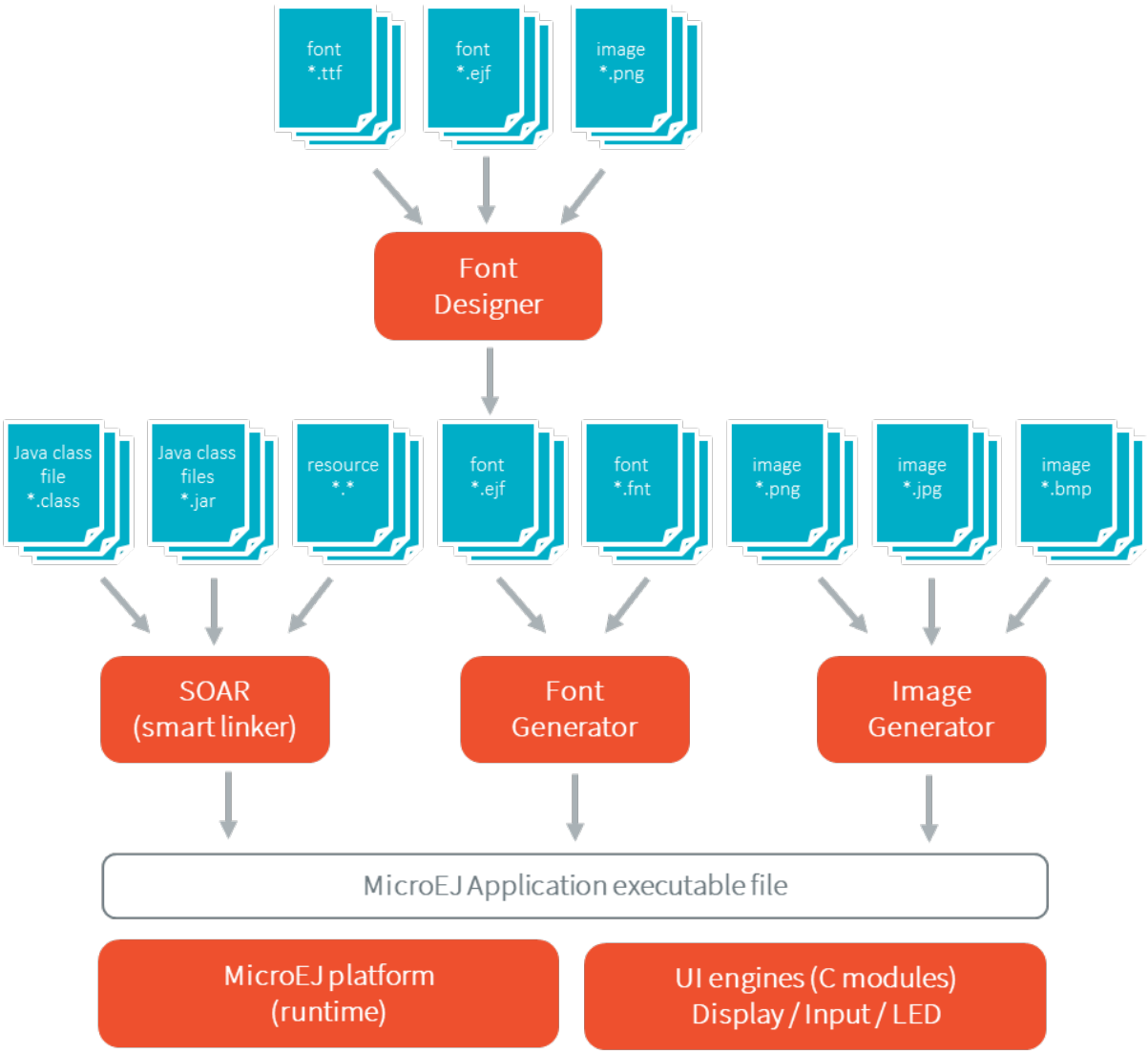


Fig. 202: The User Interface Extension Components along with a VEE Port

UI Port

This chapter summarizes all the steps to port the UI Pack: from the VEE Port Configuration project to more advanced features like using a GPU. This chapter only introduces the concepts and references the following chapters. The concepts are overviewed and incomplete (only the typical case is described).

It is recommended to follow the steps in this order:

1. Edit the VEE Port Configuration project to add the UI Pack dependency and configuration,
2. Create the Simulator extension project,
3. Port the minimal implementation of the BSP,
4. Extend the implementation by connecting a GPU.

UI Port Configuration

Principle

The first step is to update the VEE Port project: this project holds the build descriptor file (build.gradle.kts in SDK 6, module.ivy in SDK 5). This update is done in several steps, described in the sections below. Some steps are optional, depending on the capabilities of the hardware.

Warning: This chapter assumes that a valid VEE Port has been created, as described in the chapter

- *Create a VEE Port* in SDK 6
- *Create a VEE Port* in SDK 5

UI Pack Selection

The UI Pack bundles several modules, including the Graphics Engine. The Graphics Engine is a library already compiled for an MCU and a C compiler. The *MicroEJ Central Repository* provides UI Packs for a set of MCU/Compiler pairs (like for MicroEJ Architectures).

Refer to the chapter *Pack Import* to add the required UI Pack. As an example, the module dependency to add for a Cortex-M4 and GCC toolchain would be:

SDK 6 (build.gradle.kts)

SDK 5 (module.ivy)

```
dependencies {
  microejPack(
↪"com.microej.architecture.CM4.CM4hardfp_GCC48:flopi4G25-ui-pack:[UI Pack version]")
}
```

```
<dependencies>
  <!-- MicroEJ Architecture Specific Pack -->
  <dependency org="com.microej.architecture.
↪CM4.CM4hardfp_GCC48" name="flopi4G25-ui-pack" rev="[UI Pack version]"/>
</dependencies>
```

Note: The latest version of the UI Pack is 14.5.2.

UI Pack Modules

The following sections describe each module that comes with the UI Pack (purpose and configuration).

The modules provided by the UI Pack are **not installed** by default. When a module is required, it has to be enabled and configured using the VEE Port Editor.

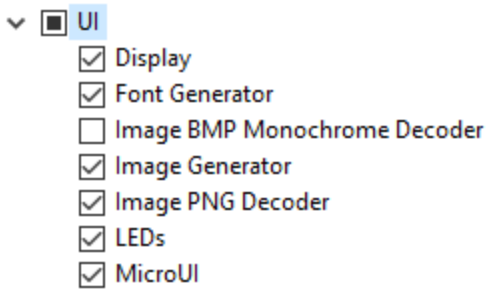


Fig. 203: UI Pack Modules

Refer to the chapter *Platform Module Configuration* to add the UI Pack modules.

Module MicroUI

MicroUI is a Foundation Library that defines a Low Level UI framework (refer to the chapter *MicroUI* for more information). The **mandatory** module MicroUI (it must be checked in the VEE Port configuration file) provides the MicroUI implementation library. It requires a static initialization step to specify what MicroUI features are available for the application layer:

1. Create the file

- [VEE Port project]/extensions/microui/microui.xml in SDK 6
- [VEE Port Configuration project]/microui/microui.xml in SDK 5

2. Edit the file as described here: *Static Initialization*.

```
<microui>

   <display name="DISPLAY"/>

   <eventgenerators>
      <command name="COMMANDS"/>
      <buttons name="BUTTONS" extended="3"/>
      <buttons name="JOYSTICK" extended="5"/>
      <touch name="TOUCH" display="DISPLAY"/>
   </eventgenerators>

   <fonts>
      <font file="resources\fonts\myfont.ejf"/>
   </fonts>

</microui>
```

Module LEDs

MicroUI provides some API to manipulate the LEDs. This module allows the UI Port to drive the LEDs. Refer to the chapter *LED* to have more information.

This module is optional: when not selected, a stub implementation is used, and the UI Port does not need to provide one.

Modules Image Decoders

Note: This chapter only applies when the device has a display.

This module adds an internal image decoder: it allows the application to embed an encoded image (e.g., PNG or BMP Monochrom) and let the Graphics Engine decode it at runtime. Both decoders (PNG and BMP Monochrom) are optional and can be selected (or not) independently. Refer to the chapter *Encoded Image* to have more information.

This module is optional: when no image decoder is embedded, the Graphics Engine relies on the UI Port (thanks to Abstraction Layer API) to decode the images.

Module Image Generator

Note: This chapter only applies when the device has a display.

This module allows decoding the application's images at compile-time. The application's images are decoded and stored in a binary format compatible with the Graphics Engine. The memory footprint of the application is higher, but the image loading time at runtime is very low. Refer to the chapter *Image Generator* to have more information.

This module is optional: when not selected, the application cannot embed generated images compatible with the Graphics Engine.

Module Font Generator

Note: This chapter only applies when the device has a display.

This module allows for embedding the MicroEJ bitmap fonts of the application. The application's fonts are decoded and stored in a binary format compatible with the Graphics Engine. Refer to the chapter *Font Generator* to have more information.

This module is optional: when not selected, the application cannot embed fonts compatible with the Graphics Engine.

Module Display

Note: This chapter only applies when the device has a display.

This chapter takes the concepts described in chapter *Display*. The first step is determining the kind of display: size, pixel format, and constraints. This information will be used later by the UI Port configuration project, the Simulator extension project, and the BSP.

Size

The size is expressed in *pixels*, often 320x240 or 480x272. This size defines the area the application can target; it can retrieve this size by calling `Display.getWidth()` and `Display.getHeight()`. It is always a rectangular area, even for the rounded displays (a square area frames a rounded display).

The display size is fixed for a display: retrieve this size in the board's datasheet.

Pixel Format

The display pixel format (or pixel structure) gives two notions: the number of bits-per-pixel and the organization of color components in these bits.

The number of bits-per-pixel (bpp) is an integer value among this list: 1, 2, 4, 8, 16, 24, or 32.

The color components organization defines how the color components (Red, Green, and Blue) are distributed in the pixel. The greater the display pixel format (in bits), the better is the definition. This format also indicates the number of bits-per-pixel. For instance, the format RGB565 is a 16-BPP format, indicating that the five MSB bits are for the Red color component, the six next bits are for the Green component, and the five LSB bits are for the Blue component. This pixel format can be symbolized by `RRRRRGGGGGGBBBBB` or `RRRR RGGG GGGB BBBB`.

The display pixel format is often fixed by the display itself (its capabilities) and by the memory bus between the MCU and the LCD module. However, the display pixel format is often configurable by the LCD controller. Note that the number of bits-per-pixel and the display size fix the required memory to allocate: `memory_size = width x height x bpp / 8`. Consequently, the pixel format may be less precise than the display capabilities depending on the memory available on the device. For instance, the RGB565 format may be used whereas the display is a 24-bit display (RGB888).

Constraints

The hardware constraints (display, bus, memory, etc.) may drive the configuration:

- The pixel format: Some hardware cannot use another pixel format other than the one of the display. This format may be standard or custom. See *Pixel Structure*.
- The size of the buffers: The available memory may be limited. This limitation can drive the chosen pixel format.
- Memory alignment: Some LCD controllers require a memory alignment on the display front buffer (alignment on 64 bits, for instance).

- Buffer width alignment: Some LCD controllers also require an alignment for each line. The line size (in pixels) in memory may be larger than the display line size (width): this is the stride. The alignment constraint may be expressed in pixels or bytes. The required memory to allocate becomes: `memory_size = stride (in pixels) x height x bpp / 8`.

Configuration

SDK 6

SDK 5

In the `configuration.properties` file of the VEE Port project, fill the file as described here: *Installation*, according to the pixel format and the display constraints.

```
com.microej.pack.display.bpp=rgb565
com.microej.pack.display.imageBuffer.memoryAlignment=32
com.microej.pack.display.memoryLayout=line
com.microej.pack.display.byteLayout=line
```

In the VEE Port Configuration project:

1. Create the file `[VEE Port Configuration project]/display/display.properties`
2. Fill the file as described here: *Installation*, according to the pixel format and the display constraints.

```
bpp=rgb565
imageBuffer.memoryAlignment=32
memoryLayout=line
byteLayout=line
```

VEE Port Build

Once modules are selected and configured, the VEE Port can be built again; see *VEE Port Build*.

Simulation

Principle

The simulation part of the UI port requires the creation (or extension) of a Front Panel project which is compatible with the UI Pack.

First, if no Front Panel project exists, follow the steps described here: *Front Panel Mock*. Then, follow the next chapters to extend the Front Panel project with UI Pack notions.

Project Extension

The Front Panel project must depend on the UI Pack. Add the following dependency to the project build file:

SDK 6 (build.gradle.kts)

SDK 5 (module.ivy)

```
implementation("ej.tool.frontpanel:framework:[Front Panel Framework version]")
implementation("com.microej.pack.ui:ui-pack:[UI Pack version]") {
  artifact {
      name = "frontpanel"
      extension = "jar"
  }
}
```

```
<dependency org="com.microej.pack.ui" name="ui-pack" rev="[UI Pack version]">
  <artifact name="frontpanel" type="jar"/>
</dependency>
```

Note: The latest version of the UI Pack is 14.5.2 and the latest version of the Front Panel Framework is 1.3.0.

See *Simulation* for more information about the Front Panel project dependencies.

LEDs

When the VEE Port Configuration project LEDs module is checked, the Front Panel project should add a widget `LED` for each led.

1. With an image editor, create an image for the LED off and an image for the LED on. Both images must have the same size.
2. Create a couple of images for each LED.
3. In the Front Panel description file, add this line for each LED:

```
<ej.fp.widget.LED label=
↪"0" x="170" y="753" ledOff="Led-0.png" ledOn="Led-GREEN.png" overlay="false"/>
```

The label must have an integer value from 0 to `NUMBER_OF_LEDS - 1`. The `ej.microui.led.Leds` class uses this value as the LED identifier in `setLedOff(int ledId)`, `setLedOn(int ledId)`, and other methods of the class.

Buttons

The widget `Button` can simulate any hardware button.

1. With an image editor, create an image for the button released and an image for the button pressed. Both images must have the same size.
2. Create a couple of images for each button.
3. In the Front Panel description file, add this line for each button:

```
<ej.fp.
↪widget.Button label="0" x="316" y="769" skin="W-U-0.png" pushedSkin="W-U-1.png"/>
```

The label must have an integer value from 0 to NUMBER_OF_BUTTONS - 1 . The label is used by the application to listen to the button.

By default, the widget sends a MicroUI Button event to the Buttons Event Generator whose name is BUTTONS and whose identifier is the button's label. To target another Buttons Event Generator, refer to the chapter *Inputs Extensions*.

Widget Button Code

```
public static class ButtonListenerToButtonEvents implements ButtonListener {

@Override
public void press(Button widget) {
  EventButton.sendPressedEvent(getMicroUIGeneratorTag(), widget.getID());
}

@Override
public void release(Button widget) {
  EventButton.sendReleasedEvent(getMicroUIGeneratorTag(), widget.getID());
}

/**
 * Gets the MicroUI␣
↪Buttons events generator tag. This generator has to match the generator set during the
 * VEE Port build in <code>microui/microui.xml</code>
 *
 * @return a MicroUI Buttons events generator tag
 */
protected String getMicroUIGeneratorTag() {
  return EventButton.COMMON_MICROUI_GENERATOR_TAG;
}
```

Application Code

To listen to the button, two ways are possible:

- By default, the current Displayable receives all events. The subclass has to implement the method boolean handleEvent(int event); .
- A class must extend the interface EventHandler , and this class must be set as the handler of the event generators Buttons:

```
Buttons[] buttonsHandlers = (Buttons[]) EventGenerator.get(Buttons.class);
for (EventGenerator buttonsHandler : generators) {
  buttonsHandler.setEventHandler(this);
}
```

Here is an example of a handler:

```
@Override
public boolean handleEvent(int event) {
```

(continues on next page)

(continued from previous page)

```
  // get the event's data
  int data = Event.getData(event);

  String state = null;

  // print its state(s)
  if (Buttons.isPressed(data)) {
    state = "pressed ";
  }
  if (Buttons.isReleased(data)) {
    state = "released ";
  }
  if (Buttons.isRepeated(data)) {
    state = "repeated ";
  }
  if (Buttons.isLong(data)) {
    state = "long ";
  }
  if (Buttons.isClicked(data)) {
    state = "clicked ";
  }
  if (Buttons.isDoubleClicked(data)) {
    state = "double-clicked ";
  }

  if (state != null) {
    System.out.print("button\t\t");

    // get the button's id
    int id = Buttons.getButtonId(data);
    System.out.print(id+" ");
    System.out.println(state);
  }

  return true;
}
```

Button to Command Event

A recommended approach is to favor Command events over Buttons events. MicroUI Command events are more generic because they are not tied to a hardware component like a physical button. Command events make the application code more flexible to hardware changes. For instance, instead of reacting to Button event 0, the application will respond to Command event `Enter` or `Up`. The application does not care about the source of the Command event: it may be the button 0, 1, 10, or any other input device.

To map a MicroUI Command on the widget Button:

1. Update the widget description by adding a `listenerClass`.

```
<ej.fp.widget.Button label="0" x="316" y="769" skin="W-U-
↪0.png" pushedSkin="W-U-1.png" listenerClass="com.is2t.microej.fp.Button2Command"/>
```

2. In the Front Panel project, create the class `com.is2t.microej.fp.Button2Command`, for instance:

```
public class Button2Command implements ej.fp.widget.Button.ButtonListener {

   public int getCommand(int buttonId) {
      // same command as EmbJPF (see buttons_listener.c)
      switch (buttonId) {
      default:
      case 0:
         return EventCommand.ESC;
      case 1:
         return EventCommand.MENU;
      }
   }

   @Override
   public void press(Button widget) {
      EventCommand.sendEvent(getCommand(widget.getID()));
   }

   @Override
   public void release(Button widget) {
      // nothing to send
   }
}
```

The application code becomes:

```
// [...]

   Command[] commandHandlers = (Command[]) EventGenerator.get(Command.class);
   for (EventGenerator commandHandler : generators) {
      commandHandler.setEventHandler(this);
   }

// [...]

@Override
public boolean handleEvent(int event) {

   // get the event's data
   int data = Event.getData(event);

   switch (data) {
   case Command.ESC:
      System.out.println("ESC");
      break;
   case Command.BACK:
      System.out.println("BACK");
      break;

   // [...]
}
```

Touch Panel

Contrary to the other input devices, no image is required because a touch panel covers the display area.

1. Retrieve the display size in pixels.
2. In the Front Panel description file, add this line:

```
<ej.fp.widget.Pointer x="185" y="395" width="480" height="272" touch="true"/>
```

By default, the widget sends a MicroUI Pointer event to the Pointer Event Generator, whose name is `TOUCH` (a touch panel is considered a Pointer with only dragged events). To target another Pointer Event Generator, refer to the chapter *Inputs Extensions*.

A snippet of application code that handles Pointer events:

```
// [...]

  Pointer[] pointerHandlers = (Pointer[]) EventGenerator.get(Pointer.class);
  for (EventGenerator pointerHandler : generators) {
    pointerHandler.setEventHandler(this);
  }

// [...]

@Override
public boolean handleEvent(int event) {
  Pointer pointer = (Pointer) Event.getGenerator(event);
  int x = pointer.getX();
  int y = pointer.getY();
  System.out.println("(" + x + "," + y + ")");
}
```

Display

The widget Display features a lot of options to simulate the hardware specificities.

1. Retrieve the display size in pixels.
2. In the Front Panel description file, add this line:

```
<ej.fp.widget.Display x="185" y="395" width="480" height="272"/>
```

For more information, refer to the java-doc of the widget Display and the chapter *Display Widget*.

Build

Once the Front Panel project is created or modified, the VEE Port must be built again (the front panel is built simultaneously with the VEE Port; see *VEE Port Build*).

BSP Port

Principle

The BSP Port (or Embedded Port) involves implementing some Abstraction Layer APIs (low-level APIs: LLAPI). There are several kinds of LLAPI:

- The mandatory LLAPI to manipulate the LEDs,
- The mandatory LLAPI to send the input events,
- The mandatory LLAPI to initialize, use and flush the drawings to the display,
- The optional LLAPI to customize the Graphics Engine to be compatible with the display constraints,
- The optional LLAPI to manipulate the optional display features (backlight, contrast, etc.),
- The optional LLAPI to add some features as new image decoders,
- The optional LLAPI to use a GPU.

The following chapters describe each group of Abstraction Layer APIs, except the GPU acceleration (see the dedicated section *GPU Port*).

MicroUI C Module

The UI Pack **requires** the *MicroUI C module*. This C module

- implements some MicroUI native functions,
- manages the drawings synchronization with the Graphics Engine,
- features an image heap allocator,
- features an input events decoder.

Before all, install the MicroUI C Module:

1. Find the correct version of the C module according to the UI Pack version; see *C Modules*.
2. Unzip it in the BSP project.
3. Add the mandatory files to the list of the BSP project's compiled files: ui_image_drawing.c , ui_drawing.c , ui_rect_util.c , LLUI_PAINTER_impl.c , ui_display_brs_single.c , LLDW_PAINTER_impl.c , ui_display_brs.c , ui_drawing_stub.c , ui_display_brs_legacy.c and ui_display_brs_predraw.c .
4. Add the optional files in the BSP project (if their associated feature is used/needed):

- LLUI_DISPLAY_HEAP_impl.c : to use another image heap allocator,
- LLUI_INPUT_LOG_impl.c and microui_event_decoder.c : to decode the MicroUI event (input events and MicroUI internal events).

5. Add the C Module's include folder to the BSP project's include directories list.

LEDs

As soon as the VEE Port Configuration project LEDs module is checked, the VEE Port features the header file LLAPI LLUI_LED_impl.h . This header must be implemented. The mandatory functions to implement are:

- LLUI_LED_IMPL_initialize : initialize the LED driver (if required) and return the available number of LEDs.
- LLUI_LED_IMPL_getIntensity : return, if possible, the LED intensity.
- LLUI_LED_IMPL_setIntensity : set the LED intensity.

Refer to *Abstraction Layer API* to have more information. Refer too to the C-doc in the header file itself.

Inputs

The VEE Port always features the header file LLAPI LLUI_INPUT_impl.h . This header must be implemented even if there is no input device: the critical section management is required by the MicroUI library itself. The mandatory functions to implement are:

- LLUI_INPUT_IMPL_initialize : can be empty if nothing is to initialize.
- LLUI_INPUT_IMPL_getInitialStateValue : empty if there is no State Event Generator.
- LLUI_INPUT_IMPL_enterCriticalSection : disable all input events (disable input devices interrupts and/or disable the OS scheduling).
- LLUI_INPUT_IMPL_leaveCriticalSection : re-enable all inputs events.

Refer to *Abstraction Layer API* to have more information. Refer too to the C-doc in the header file itself.

Display

As soon as the VEE Port Configuration project Display module is checked, the VEE Port features the header file LLAPI LLUI_DISPLAY_impl.h . This header must be implemented. The mandatory functions to implement are:

- LLUI_DISPLAY_IMPL_initialize : fill the given structure LLUI_DISPLAY_SInitData (display size, buffer address, etc.).
- LLUI_DISPLAY_IMPL_binarySemaphoreTake : *takes* the given semaphore.
- LLUI_DISPLAY_IMPL_binarySemaphoreGive : *gives* the given semaphore.
- LLUI_DISPLAY_IMPL_flush : copy/transmit the buffer content to the LCD.

Refer to *Abstraction Layer API* to have more information. Refer to the C-doc in the header file itself too.

Display: LCD Constraints

According to the LCD constraints (see *UI Port Configuration*), some additional LLAPI must be implemented:

- LLUI_DISPLAY_IMPL_convertARGBColorToDisplayColor and LLUI_DISPLAY_IMPL_convertDisplayColorToARGBColor : required when the pixel format is custom (not standard, see *Dependencies*).
- LLUI_DISPLAY_IMPL_prepareBlendingOfIndexedColors : required when the display back buffer is a LUT buffer, not a pixel buffer.
- LLUI_DISPLAY_IMPL_isDoubleBuffered : the default implementation returns always true ; only useful as information for the application.
- LLUI_DISPLAY_IMPL_isColor : the default implementation always returns true when the BPP is higher than 8; only useful as information for the application.
- LLUI_DISPLAY_IMPL_getNumberOfColors : the default implementation returns always 1 << BPP ; only useful as information for the application.

Display: Buffer Configuration

This configuration consists in declaring the available number of buffers where MicroUI can draw (back buffer) and the strategy to apply to update these buffers after a flush. Read and update the configuration file ui_configuration.h ; more details in the chapter *Display*.

Display: Optional Features

Several kinds of features can be implemented.

Hardware features:

- LLUI_DISPLAY_IMPL_setContrast and LLUI_DISPLAY_IMPL_getContrast : to configure the display contrast.
- LLUI_DISPLAY_IMPL_hasBacklight , LLUI_DISPLAY_IMPL_setBacklight and LLUI_DISPLAY_IMPL_getBacklight : to turn on or off the display backlight.

Runtime Image Decoders

The BSP can add some runtime image decoders with the runtime decoders selected in the VEE Port configuration project (modules PNG and BMP Monochrom decoders).

- LLUI_DISPLAY_IMPL_decodeImage : called by MicroUI to decode an image whose format is unknown by the internal runtime image decoders.

Image Heap Management

By default, a best-fit allocator manages the image heap. To add another allocator, implement these functions:

- LLUI_DISPLAY_IMPL_imageHeapInitialize : initialize the allocator.
- LLUI_DISPLAY_IMPL_imageHeapAllocate : allocates the expected buffer.
- LLUI_DISPLAY_IMPL_imageHeapFree : frees the given buffer.

MicroUI Image Management

These three functions are only helpful for compatibility with a GPU; see *GPU Port*.

- LLUI_DISPLAY_IMPL_getNewImageStrideInBytes
- LLUI_DISPLAY_IMPL_adjustNewImageCharacteristics
- LLUI_DISPLAY_IMPL_initializeNewImage

Testsuite

The Port Qualification Toolkit (PQT) provides a UI testsuite to validate the UI Port (see *VEE Port Testsuite* to have more information). This testsuite **must** be executed to validate the UI Port and after each modification on this UI Port (for instance, after changes to improve performances).

The UI Port testsuite is available here:

- SDK 5: https://github.com/MicroEJ/VEEPortQualificationTools/tree/master/tests/ui/ui3
- SDK 6: https://github.com/MicroEJ/Tool-Project-Template-VEEPort/tree/1.5.0/vee-port/validation/ui

The testsuite is constituted of two blocks:

- The minimal *Display* testsuite: a simple application test (with source code) to validate the mandatory functions to implement to target a Display.
- An extended *Display* testsuite: a library that tests several MicroUI drawings. This testsuite only applies when the BSP uses a GPU to perform the drawings. See *GPU Port*.

The testsuite does not check all UI Port features. However, some example projects are available in MicroEJ GitHub:

- LED: refer to the application https://github.com/MicroEJ/Example-Standalone-Foundation-Libraries/tree/master/microui.led.
- Input: refer to the application https://github.com/MicroEJ/Example-Standalone-Foundation-Libraries/tree/master/microui.input

Some other example projects are also available in MicroEJ GitHub and can be used to check if the UI Port is valid:

- Hello World: https://github.com/MicroEJ/Example-Standalone-Foundation-Libraries/tree/master/microui.helloworld
- Use of images: https://github.com/MicroEJ/Example-Standalone-Foundation-Libraries/tree/master/microui.image

GPU Port

Principle

MicroUI and *MicroUI C module* are designed to be extended using a GPU hardware drawing acceleration. This acceleration is optional and should be performed after the mandatory operations (see *BSP Port*).

A GPU can be used to draw shapes and/or images. Most of the time, the minimal implementation consists in filling the rectangles and drawing the images. The MicroUI C module is designed to let the BSP implement only the GPU features. When a drawing is not implemented over a GPU, the software implementation is automatically used instead. No extra code should be added to the BSP to use the software algorithms.

The main advantages of using a GPU are:

- the drawing is rendered faster than using the software algorithms,
- the drawing is performed asynchronously, allowing the MCU to perform other actions (no need to wait until the end of the drawing).

Existing C Modules

Some *C Modules* are available on the MicroEJ Repository. These C modules already implement compatible features with a GPU. Add the mandatory files to the list of the BSP project's compiled files to use the associated GPU (and add the C Module's include folder to the BSP project's include directories list). Refer to *C Modules* to have more information.

Port a GPU

Drawing Function

As described in *Painter Abstraction Layer API*, the idea of the implementation of `LLUI_PAINTER_impl.h` (and `LLDW_PAINTER_impl.h`) is first to manage the synchronization with the Graphics Engine and then, to dispatch the drawing itself to a third party implementation through the header file `ui_drawing.h` . The files `LLUI_PAINTER_impl.c` and `LLDW_PAINTER_impl.c` available in the MicroUI C module already perform this operation for all MicroUI drawings. Consequently, only the drawing itself should be implemented in the BSP.

For example:

```
DRAWING_Status UI_DRAWING_
↪fillRectangle(MICROUI_GraphicsContext* gc, jint x1, jint y1, jint x2, jint y2) {
  // TODO
}
```

The drawing function has to take into account these properties:

- the color: the structure `MICROUI_GraphicsContext` gives the shape color (always fully opaque),
- the clip: the `LLUI_DISPLAY.h` file provides some functions to retrieve the current `MICROUI_GraphicsContext` 's clip,
- the buffer destination address by calling the `LLUI_DISPLAY_getBufferAddress` function,
- the shape bounds: the drawing function parameters.

The drawing function must return the drawing status. This status indicates to the Graphics Engine the kind of drawing:

- synchronous drawing: the drawing is performed by the GPU and entirely performed before returning. In that case, the drawing function has to return `DRAWING_DONE` .
- asynchronous drawing: the drawing is started, maybe processed by the GPU before returning. In that case, the drawing function has to return `DRAWING_RUNNING` .

As explained above, the second case should be the rule. That means that the Graphics Engine cannot ask for another drawing (accelerated or not) before the end of the drawing currently performed by the GPU. To unlock the Graphics Engine, the GPU interrupt routine must call the Graphics Engine function `LLUI_DISPLAY_notifyAsynchronousDrawingEnd` to notify the

end of the drawing. The Graphics Engine manages the synchronization alone; no extra support in the BSP is mandatory.

Note: The end of the asynchronous drawing may occur before the end of the drawing function execution (before returning). The Graphics Engine also manages this use case, and the BSP implementation does not need to check this use case.

Fallback

A GPU may not be able to manage all the drawing functions. For instance, it cannot manage all image formats, or it cannot manage all rotation angles, etc. In that case, the drawing function can call the software drawing function instead.

```
DRAWING_Status UI_DRAWING_
↪fillRectangle(MICROUI_GraphicsContext* gc, jint x1, jint y1, jint x2, jint y2) {
  DRAWING_Status ret;
  if (!compatible_drawing(gc, x1, y1, x2, y2)) {
    UI_DRAWING_SOFT_fillRectangle(gc, x1, y1, x2, y2);
    ret = DRAWING_DONE;
  }
  else {
    gpu_fill_rect(LLUI_DISPLAY_getBufferAddress(&gc->image), x1, y1, x2, y2);
    ret = DRAWING_RUNNING;
  }
  return ret;
}
```

Image Constraints

The GPU may have strong requirements on the images:

- the pixels buffer start address alignment,
- an image stride different than the image width.

These constraints affect the compile-time images (Image Generator) and the runtime images (decoded images and MicroUI BufferedImage).

Address Alignment

In the VEE Port Configuration project, specify the property `imageBuffer.memoryAlignment`:

SDK 6

SDK 5

In SDK 6, the configuration is done in the properties file `configuration.properties` of the VEE Port project:

```
com.microej.pack.imageBuffer.memoryAlignment=16
```

In SDK 5, the configuration is done in the properties file `display/display.properties`:

```
imageBuffer.memoryAlignment=16
```

The value is the alignment in bits. This value will be taken into account by the compile-time images (Image Generator) and the runtime images.

Note: For the runtime images, this alignment value may be customized thanks to the function LLUI_DISPLAY_IMPL_adjustNewImageCharacteristics .

Stride (Compile-time Images)

The stride is dynamic, often depending on the image format and width.

For the compile-time images (Image Generator), a specific extension of the ImageGenerator is required.

1. See *Extended Mode* to create the ImageGenerator extension project.
2. Create a class that implements BufferedImageLoader . The value to be returned is expressed in pixels.

```
public class MicroUIGeneratorExtension extends BufferedImageLoader{

   private static final int ALIGNMENT_PIXELS = 16;

   @Override
   public int getStride(int defaultStride) {
      return (getWidth() + ALIGNMENT_PIXELS - 1) & ~(ALIGNMENT_PIXELS - 1);
   }
}
```

3. Create the file /META-INF/services/com.microej.tool.ui.generator.MicroUIRawImageGeneratorExtension
4. Fill it with the class name:

```
my.package.MicroUIGeneratorExtension
```

5. Build the project and copy the result in the VEE Port Configuration project, subfolder dropins/tools .
6. Rebuild the VEE Port.

Stride (Runtime Images)

For the compile-time images, the BSP has to implement the LLAPI LLUI_DISPLAY_IMPL_getNewImageStrideInBytes (the value to be returned is expressed in bytes):

```
uint32_t UI_DRAWING_getNewImageStrideInBytes(jbyte␣
↪image_format, uint32_t image_width, uint32_t image_height, uint32_t default_stride) {
    uint32_t bpp = DISPLAY_UTILS_get_bpp((MICROUI_ImageFormat)image_format);
    return (bpp >= (uint32_t)8) ? ALIGN(image_
↪width, (uint32_t)16) * (bpp / (uint32_t)8) : ALIGN(image_width, (uint32_t)8);
}
```

Testsuite

As described *here*, the Port Qualification Toolkit (PQT) provides a UI testsuite to validate the UI Port. The second block of the UI testsuite (extended *Display* testsuite) uses a library that tests several MicroUI drawings. This testsuite **must** be executed to validate the UI Port over a GPU and after each modification on this UI Port (for instance, after changes to improve performances).

MicroUI

Principle

MicroUI library defines a Low Level UI framework for embedded devices. This module allows the creation of basic Human-Machine-Interfaces (HMI), with output on a pixel-based screen.

Architecture

MicroUI library is the entry point to perform some drawings on a display and to interact with user input events. This library contains only a minimal set of basic APIs. High-level libraries can be used to have more expressive power, such as *MWT (Micro Widget Toolkit)*. In addition to this restricted set of APIs, the MicroUI implementation has been designed so that the EDC and BON footprint is minimal.

At application startup all MicroUI objects relative to the I/O devices are created and accessible. The following MicroUI methods allow you to access these objects:

- Display.getDisplay() : returns the instance of the display which drives the main display screen.
- Leds.getNumberOfLeds(): returns the numbers of available LEDs.

MicroUI is not a standalone library. It requires a configuration step and several extensions to drive I/O devices (display, inputs, LEDs).

First, MicroUI requires a configuration step in order to create these internal objects before the call to the $\mathrm{main}()$ method. The chapter *Static Initialization* explains how to perform the configuration step.

Note: This configuration step is the same for both embedded and simulated VEE Ports.

The embedded VEE Port requires some additional C libraries to drive the I/O devices. Each C library is dedicated to a specific kind of I/O device. A specific chapter is available to explain each kind of I/O device.

Table 60: MicroUI C libraries

I/O devices	Extension Name	Chapter
Graphical / pixel-based display	Display	*Display*
Inputs (buttons, joystick, touch, pointers, etc.)	Input	*Input*
LEDs	LED	*LED*

The simulation VEE Port uses a mock which simulates all I/O devices. Refer to the chapter *Simulation*.

Library ej.api.Drawing

This Foundation Library provides additional drawing APIs. This library is fully integrated in *Display* module.

Thread

Principle

The MicroUI implementation for MicroEJ uses one internal thread. This thread is created during the MicroUI initialization step, and is started by a call to MicroUI.start().

Role

This thread has several roles:

- It manages all display events (requestRender(), requestShow(), etc.).
- It reads the I/O devices inputs and dispatches them into the event generators' listeners. See input section: *Input*.
- It allows to run some piece of code using the callSerially() method.

Memory

The thread is always running. The user has to count it to determine the number of concurrent threads the Core Engine can run (see *Memory* options in *Standalone Application Options*).

Exceptions

The thread cannot be stopped with a Java exception: the exceptions are always checked by the framework.

When an exception occurs in a user method called by the internal thread (for instance render()), the current UncaughtExceptionHandler receives the exception. When no exception handler is set, a default handler prints the stack trace.

Native Calls

The MicroUI implementation for MicroEJ uses native methods to perform some actions (read input devices events, perform drawings, turn on LEDs, etc.). The library implementation has been designed to not use blocking native methods (wait input devices, wait end of drawing, etc.) which can lock the full Core Engine execution.

The specification of the native methods is to perform the action as fast as possible. The action execution may be sequential or parallel because an action is able to use a third-party device (software or hardware). In this case, some callbacks are available to notify the end of this kind of parallel actions.

However some actions have to wait the end of a previous parallel action. By consequence the caller thread is blocked until the previous action is done; in other words, until the previous

parallel action has called its callback. In this case, only the current thread is locked (because it cannot continue its execution until both actions are performed). All other threads can run, even a thread with a lower priority than current thread. If no thread has to be run, the Core Engine goes in sleep mode until the native callback is called.

Antialiasing

MicroUI provides several policies to use the antialiasing. These policies depend on several factors, including the kind of drawing and the display pixel rendering format. The main concept is that MicroUI does not allow you to draw something with a transparency level different from 255 (fully opaque). There are two exceptions: the images and the fonts.

For each pixel to draw, the antialiasing process blends the foreground color with a background color. This background color can be specified or not by the application:

- *specified*: The background color is fixed by the application (GraphicsContext.setBackgroundColor()).
- *not specified*: The background color is the original color of the destination pixel (a "read pixel" operation is performed for each pixel).

Images

Drawing an image (a pre-generated image or an image decoded at runtime) which contains some transparency levels does not depend on the display pixel rendering format. During the image drawing, each pixel is converted into 32 bits by pixel format.

This pixel format contains 8 bits to store the transparency level (alpha). This byte is used to merge the foreground pixel (image transparent pixel) with the background pixel (buffer opaque pixel). The formula to obtain the pixel is:

$$\alpha Mult = (\alpha FG * \alpha BG)/255$$

$$\alpha Out = \alpha FG + \alpha BG - \alpha Mult$$

$$COut = (CFG * \alpha FG + CBG * \alpha BG - CBG * \alpha Mult)/\alpha Out$$

The destination buffer is always opaque, so:

$$COut = (CFG * \alpha FG + CBG * (255 - \alpha Mult))/255$$

where:

- αFG is the alpha level of the foreground pixel (layer pixel),
- αBG is the alpha level of the background pixel (working buffer pixel),
- *Cxx* is a color component of a pixel (Red, Green or Blue),
- αOut is the alpha level of the final pixel.

Fonts

A font holds only a transparency level (alpha). This fixed alpha level is defined during the pre-generation of a font (see *Fonts*).

- 1 means 2 levels are managed: fully opaque and fully transparent.
- 2 means 4 levels are managed: fully opaque, fully transparent and 2 intermediate levels.
- 4 means 16 levels are managed: fully opaque, fully transparent and 14 intermediate levels.
- 8 means 256 levels are managed: fully opaque, fully transparent and 254 intermediate levels.

Note: The antialiasing mode for the fonts concerns only the fonts with more than 1 bit per pixel (see *Font Generator*).

Installation

The MicroUI library is an additional module. To enable it, refer to the *UI Port Configuration*.

Use

See *MicroUI* chapter in Application Developer Guide.

Static Initialization

Principle

The MicroUI implementation for MicroEJ requires a configuration step (also called extension step) to customize itself before application startup (see *Architecture*). This configuration step uses an XML file. In order to save both runtime execution time and flash memory, the file is processed by the Static MicroUI Initializer tool, avoiding the need to process the XML configuration file at runtime. The tool generates appropriate initialized objects directly within the MicroUI library, as well as Java and C constants files for sharing MicroUI event generator IDs.

This XML file (also called the initialization file) defines:

- The MicroUI event generators that will exist in the application in relation to low-level drivers that provide data to these event generators (see *Input*).
- Whether the application has a display; and if so, it provides its logical name.
- Which fonts will be provided to the application.

The next chapters describe succinctly the XML file. For more information about grammar, please consult appendix *MicroUI Static Initializer*.

Functional Description

The Static MicroUI Initializer tool takes as entry point the initialization file which describes the MicroUI library extension. This tool is automatically launched during the VEE Port build (see *Installation*).

The Static MicroUI Initializer tool is able to generate two files:

- A Java library which extends MicroUI library. This library is automatically added to the *MicroEJ Application classpath* when MicroUI API library is fetched. This library is used at MicroUI startup to create all instances of I/O devices (Display, EventGenerator, etc.) and contains the fonts described into the configuration file (these fonts are also called “system fonts”).

Warning: This MicroUI extension library is always generated and MicroUI library cannot run without this extension.

- A C header file (*.h). This header file contains some IDs which are used to make a link between an input device (buttons, touch) and its MicroUI event generator (see *Input*).

Note: The Front Panel project does not need a configuration file (like C header file for embedded VEE Port).

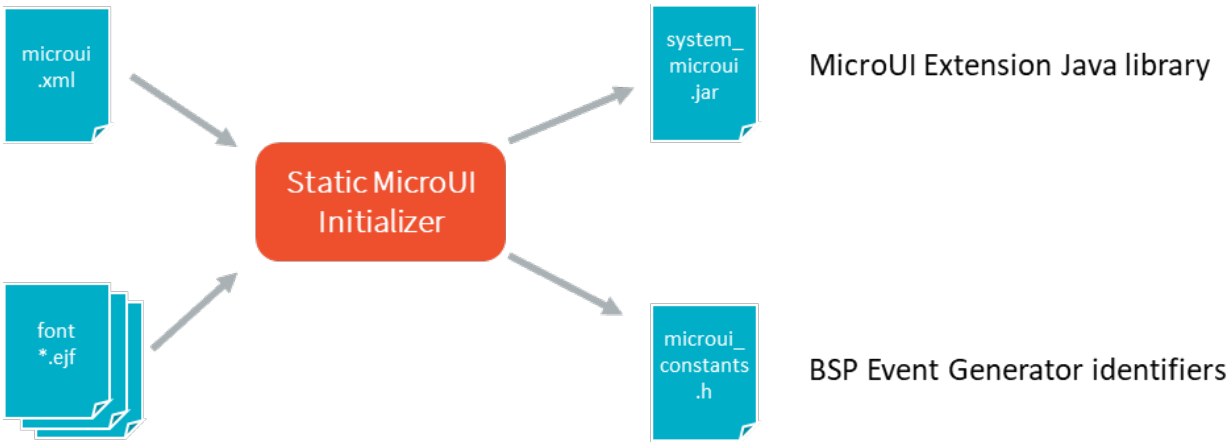


Fig. 204: Static MicroUI Initializer Process

XML File

The XML file must be created in VEE Port configuration project, in folder microui and called microui.xml .

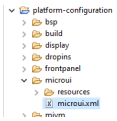


Fig. 205: Static MicroUI Initializer XML File

The XML file grammar is detailed *here*. The following list gives a short description of each element:

- Root element: The initialization file root element is <microui> and contains component-specific elements.

```
<microui>
   [ component specific elements ]
</microui>
```

- Display element: The display element augments the initialization file with the configuration of the display. The following snippet is an example of display element:

```
<display name="DISPLAY"/>
```

- Fonts element: The fonts element augments the initialization file with the fonts that are implicitly embedded within the application (also called system fonts). Applications can also embed their own fonts.

  Note: The system fonts are optional, in which case application has to provide some fonts to be able to draw characters.

  The following snippet is an example of fonts element:

```
<fonts>
   <font file="resources\fonts\myfont.ejf">
      <range name="LATIN" sections="0-2"/>
      <customrange start="0x21" end="0x3f"/>
   </font>
   <font file="C:\data\myfont.ejf"/>
</fonts>
```

- Event generators element: The eventgenerators element augments the initialization file with:
  - the configuration of the predefined MicroUI EventGenerator: Command, Buttons, States, Pointer and Touch.
  - the configuration of the generic MicroUI EventGenerator.

  The following snippet is an example of eventgenerators element:

```
<eventgenerators>
   <!-- Generic Event Generators -->
   <eventgenerator name="GENERIC" class="foo.bar.Zork">
      <property name="PROP1" value="3"/>
      <property name="PROP2" value="aaa"/>
   </eventgenerator>

   <!-- Predefined Event Generators -->
   <command name="COMMANDS"/>
   <buttons name="BUTTONS" extended="3"/>
   <buttons name="JOYSTICK" extended="5"/>
   <pointer name="POINTER" width="1200" height="1200"/>
   <touch name="TOUCH" display="DISPLAY"/>
   <states name="STATES" numbers="NUMBERS" values="VALUES"/>

</eventgenerators>

<array name="NUMBERS">
   <elem value="3"/>
```

(continues on next page)

(continued from previous page)

```
    <elem value="2"/>
    <elem value="5"/>
</array>

<array name="VALUES">
    <elem value="2"/>
    <elem value="0"/>
    <elem value="1"/>
</array>
```

XML File Example

This common MicroUI initialization file initializes MicroUI with:

- a Display,
- a Command event generator,
- a Buttons event generator which targets *n* buttons (3 first buttons having extended features),
- a Buttons event generator which targets the buttons of a joystick,
- a Pointer event generator which targets a touch panel,
- a Font whose path is relative to this file.

```
<microui>

    <display name="DISPLAY"/>

    <eventgenerators>
        <command name="COMMANDS"/>
        <buttons name="BUTTONS" extended="3"/>
        <buttons name="JOYSTICK" extended="5"/>
        <touch name="TOUCH" display="DISPLAY"/>
    </eventgenerators>

    <fonts>
        <font file="resources\fonts\myfont.ejf"/>
    </fonts>

</microui>
```

Dependencies

No dependency.

Installation

The Static Initialization tool is part of the MicroUI module (see *MicroUI*). Install the MicroUI module to install the Static Initialization tool and fill all properties in MicroUI module configuration file (which must specify the name of the initialization file).

Use

The Static MicroUI Initializer tool is automatically launched during the VEE Port build.

Abstraction Layer API

Principle

The MicroUI implementation for MicroEJ requires an Abstraction Layer implementation. This Abstraction Layer implementation finalizes the MicroUI implementation started with the static initialization step (see *Static Initialization*) for a given VEE Port.

The Abstraction Layer implementation consists in a set of headers files to implement in C to target the hardware drivers. Some functions are mandatory, others are not. Some other headers files are also available to call UI engines internal functions.

For the simulator, some Front Panel interfaces and classes allow to specify the simulated VEE Port characteristics.

Embedded VEE Port

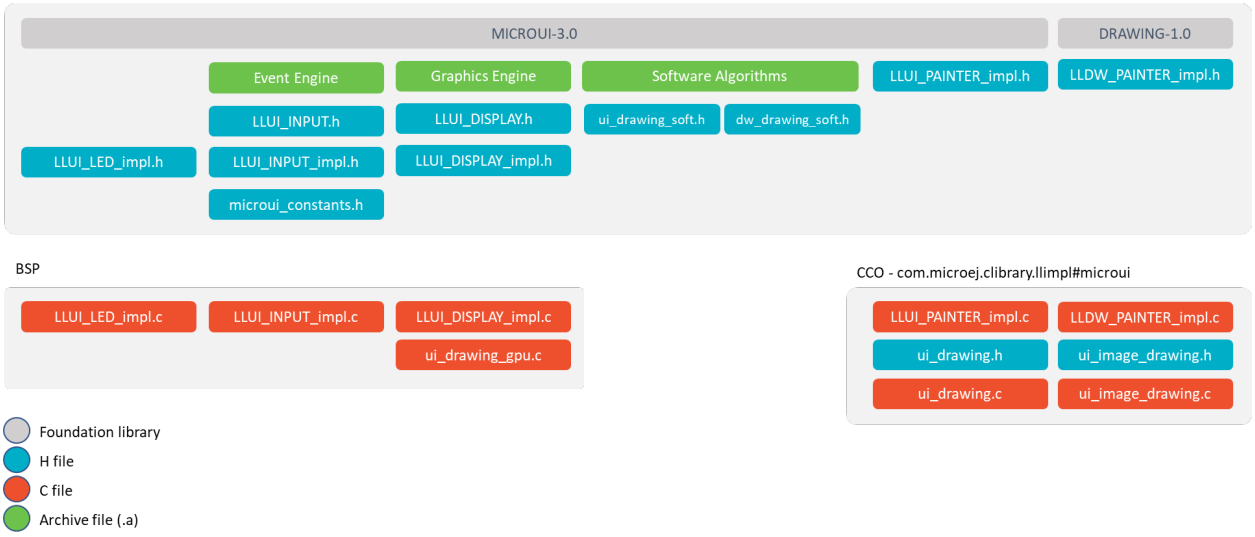


Fig. 206: MicroUI Embedded Abstraction Layer API

The specification of header files names is:

- Name starts with `LLUI_`.

- Second part name refers the UI engine: DISPLAY , INPUT , LED .
- Files whose name ends with _impl list functions to implement over hardware.
- Files whose name has no suffix list internal UI engines functions.

There are some exceptions :

- LLUI_PAINTER_impl.h and LLDW_PAINTER_impl.h list a subpart of UI Graphics Engine functions to implement (all MicroUI native drawing methods).
- ui_drawing_soft.h and dw_drawing_soft.h list all drawing methods implemented by the Graphics Engine.
- microui_constants.h is the file generated by the MicroUI Static Initializer (see *Static Initialization*).

The *MicroUI C module* provides a default implementation of the UI Pack Abstraction Layer API:

- LLUI_PAINTER_impl.c and LLDW_PAINTER_impl.c manage the synchronization with the Graphics Engine and redirect all drawings to ui_drawing.h and ui_image_drawing.h .
- ui_drawing.h and ui_image_drawing.h list all drawing methods the VEE Port can implement.
- ui_drawing.c and ui_image_drawing.c are the default implementation of ui_drawing.h and ui_image_drawing.h that redirects all drawings to ui_drawing_soft.h and dw_drawing_soft.h .

The BSP has to implement LLUI_xxx header files and optionally ui_drawing.h and ui_image_drawing.h (to draw using a GPU and/or to draw in a *custom BufferedImage*).

All header files and their aims are described in next UI engines chapters: *LED*, *Input* and *Display*.

Simulator

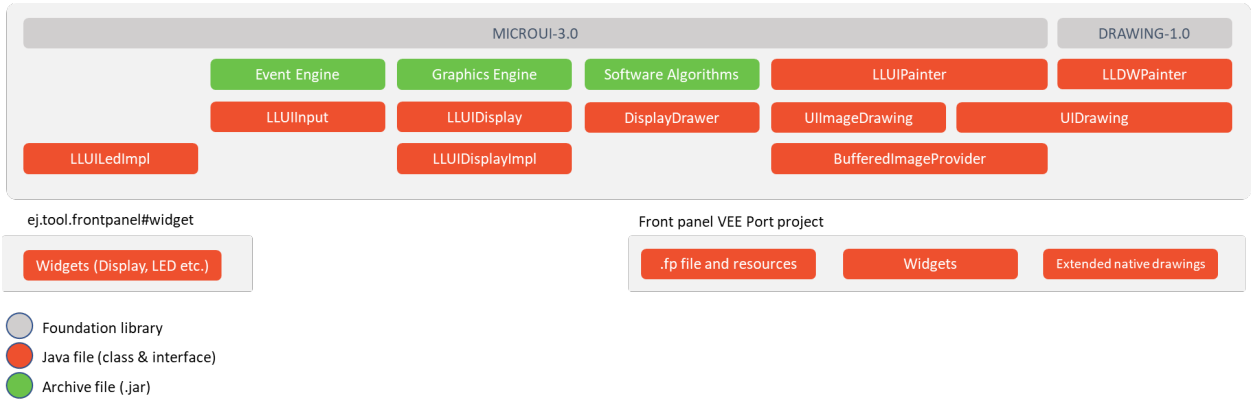


Fig. 207: MicroUI Simulator Abstraction Layer API

In the simulator the three UI engines are grouped in a mock called Front Panel. The Front Panel comes with a set of classes and interfaces which are the equivalent of headers file (*.h) of Embedded VEE Port.

The specification of class names is:

- Package are the same than the MicroUI library (ej.microui.display, ej.microui.event, ej.microui.led).
- Name start with LLUI .

- The second part of the name refers the UI engine: Display , Input , Led .
- Files whose name ends with Impl list methods to implement like in the embedded VEE Port.
- Files whose name has no suffix list internal UI engine functions.

There are some exceptions :

- LLUIPainter.java and LLDWPainter.java list a subpart of UI Graphics Engine functions (all MicroUI native drawing methods).
- UIDrawing.java and DWDrawing.java list all drawing methods the VEE Port can implement (and already implemented by the Graphics Engine).
- EventXXX list methods to create input events compatible with MicroUI implementation.

All files and their aims are described in *Simulation*.

LED

Principle

The LED module contains the C part of the MicroUI implementation which manages LED devices. This module is composed of only one element: an implementation of the Abstraction Layer APIs for the LEDs which must be provided by the BSP (see *LLUI_LED: LEDs*).

Functional Description

The LED module implements the MicroUI Leds framework. LLUI_LED specifies the Abstraction Layer APIs that receive orders from the Managed world.

The Abstraction Layer APIs are the same for the LED which is connected to a GPIO (0 or 1), to a PWM , to a bus (I2C , SPI), etc. The BSP has the responsibility of interpreting the application parameter intensity .

Typically, when the LED is connected to a GPIO , the intensity "0" means "OFF", and all other values "ON". When the LED is connected via a PWM , the intensity "0" means "OFF", and all other values must configure the PWM duty cycle signal.

The BSP should be able to return the state of an LED. If it is not able to do so (for example GPIO is not accessible in read mode), the BSP has to save the LED state in a global variable. If not, the returned value may be wrong and the application may not be able to know the LEDs states.

Abstraction Layer API

The LED module provides Abstraction Layer APIs that allow the BSP to manage the LEDs. The BSP has to implement these Abstraction Layer APIs, making the link between the MicroUI library and the BSP LEDs drivers.

The Abstraction Layer APIs to implement are listed in the header file LLUI_LEDS_impl.h . First, in the initialization function, the BSP must return the available number of LEDs the board provides. The other functions are used to turn the LEDs on and off.

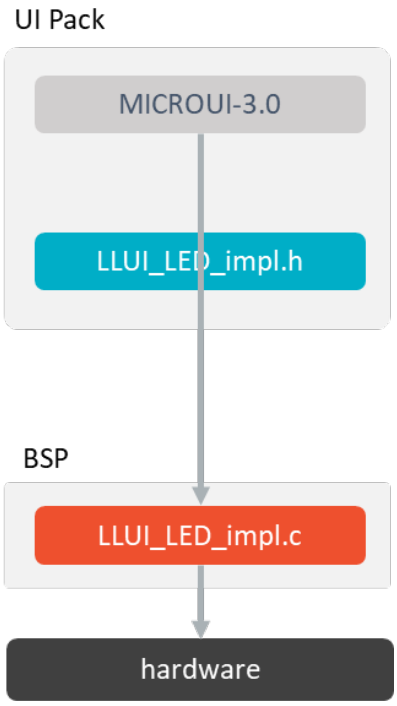


Fig. 208: Led Abstraction Layer API

When there is no LED on the board, a *stub* implementation of C library is available. This C library must be linked by the third-party C IDE when the MicroUI module is installed in the VEE Port. This stub library does not provide any Abstraction Layer API files.

Typical Implementation

This chapter helps to write a basic `LLUI_LEDS_impl.h` implementation. This implementation manages some two-state LEDs: on or off.

The pseudo-code calls external functions such as `LEDS_DRIVER_xxx` to symbolize the use of external drivers.

```
static void get_led_port_and_pin(int32_t ledID, int32_t* port, int32_t* pin)
{
  switch(ledID)
  {
    /* TODO */
    *port = ...;
    *pin = ...;
  }
}

jint LLUI_LED_IMPL_getIntensity(jint ledID)
{
  int32_t port;
  int32_t pin;
  get_led_port_and_pin(ledID, &port, &pin);
```

(continues on next page)

(continued from previous page)

```
  return GPIO_ReadPin(port, pin) ==␣
↪GPIO_PIN_RESET ? LLUI_LED_MAX_INTENSITY : LLUI_LED_MIN_INTENSITY;
}

jint LLUI_LED_IMPL_initialize(void)
{
  return DRIVER_LEDS_Init();  // return the available number of leds
}

void LLUI_LED_IMPL_setIntensity(jint ledID, jint intensity)
{
  int32_t port;
  int32_t pin;
  get_led_port_and_pin(ledID, &port, &pin);

  GPIO_WritePin(port, pin, 0 == intensity ? GPIO_PIN_RESET : GPIO_PIN_SET);
}
```

Dependencies

- MicroUI module (see *MicroUI*).
- LLUI_LED_impl.h implementation if standard implementation is chosen (see *Functional Description* and *LLUI_LED: LEDs*).

Installation

LEDs is a sub-part of MicroUI library. When the MicroUI module is installed, the LED module must be installed in order to be able to connect physical LEDs with VEE Port. If not installed, the *stub* module will be used.

In the VEE Port configuration file, check UI > LEDs to install LEDs.

Use

The MicroUI LEDs APIs are available in the class ej.microui.led. Leds.

Input

Principle

The Input module contains the C part of the MicroUI implementation which manages input devices. This module is composed of two elements:

- the C part of MicroUI input API (a built-in C archive) called Input Engine,
- an implementation of Abstraction Layer APIs for the input devices that must be provided by the BSP (see *LLUI_INPUT: Input*).

Functional Description

The Input module implements the MicroUI int -based event generators' framework. LLUI_INPUT specifies the Abstraction Layer APIs that send events to the Managed world.

Drivers for input devices must generate events that are sent, via a MicroUI Event Generator, to the application. An event generator accepts notifications from devices, and generates an event in a standard format that can be handled by the application. Depending on the MicroUI configuration, there can be several different types of event generator in the system, and one or more instances of each type.

Each MicroUI Event Generator represents one side of a pair of collaborative components that communicate using a shared buffer:

- The producer: the C driver connected to the hardware. As a producer, it sends its data into the communication buffer.
- The consumer: the MicroUI Event Generator. As a consumer, it reads (and removes) the data from the communication buffer.

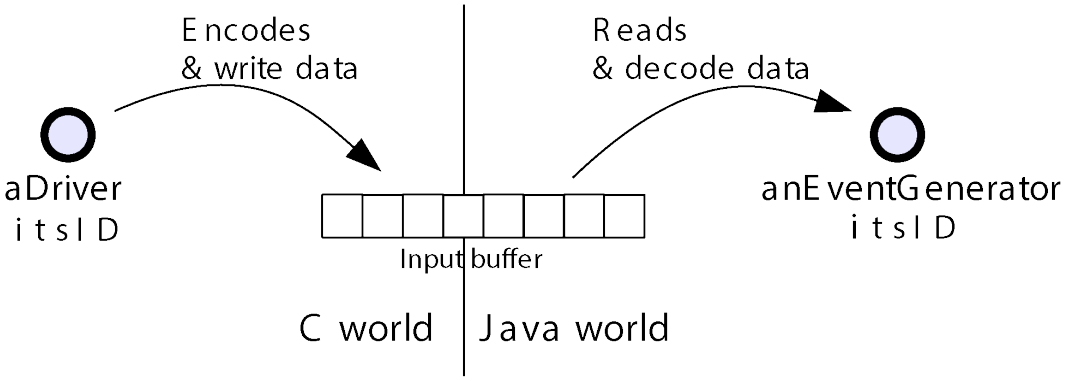


Fig. 209: Drivers and MicroUI Event Generators Communication

The LLUI_INPUT API allows multiple pairs of <driver - event generator> to use the same buffer, and associates drivers and event generators using an int ID. The ID used is the event generator ID held within the MicroUI global registry. Apart from sharing the ID used to "connect" one driver's data to its respective event generator, both entities are completely decoupled.

The MicroUI thread waits for data to be published by drivers into the "input buffer", and dispatches to the correct (according to the ID) event generator to read the received data. This "driver-specific-data" is then transformed into MicroUI events by event generators and sent to objects that listen for input activity.

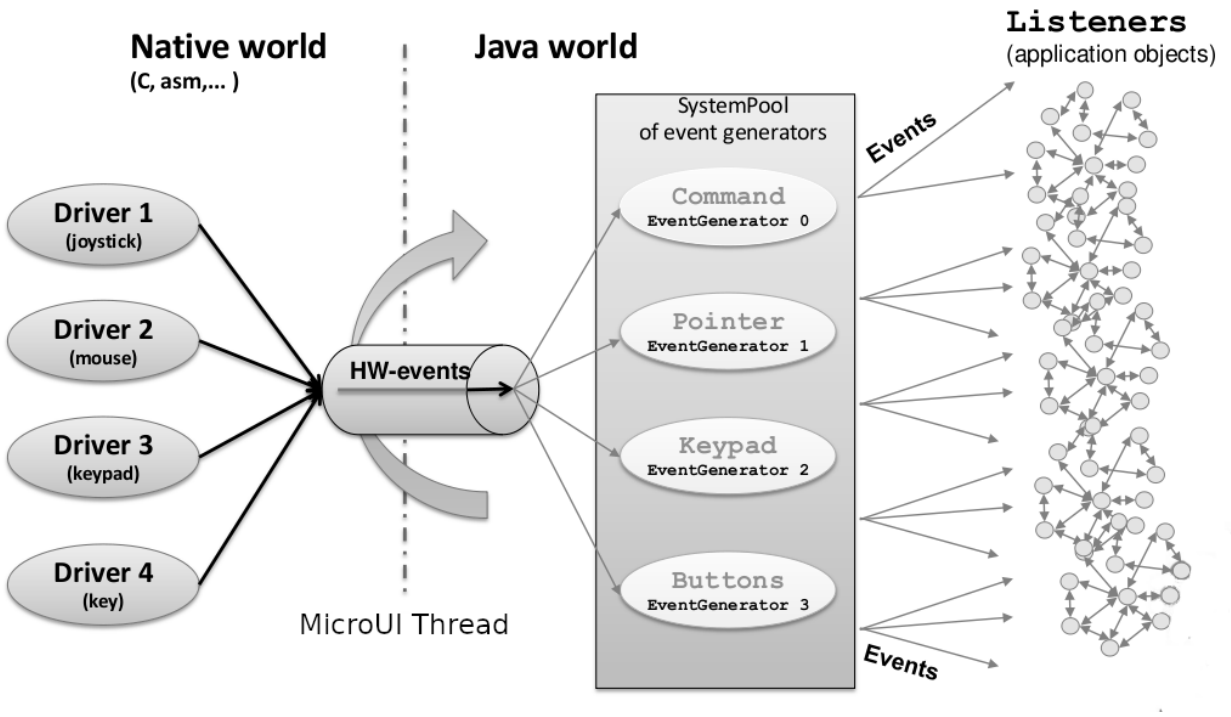


Fig. 210: MicroUI Events Framework

Driver Listener

Drivers may either interface directly with event generators, or they can send their notifications to a *Listener*, also written in C, and the listener passes the notifications to the event generator. This decoupling has two major benefits:

- The drivers are isolated from the MicroEJ libraries – they can even be existing code.
- The listener can translate the notification; so, for example, a joystick could generate pointer events.

Static Initialization

The event generators available on MicroUI startup (after the call to MicroUI.start()) are the event generators listed in the MicroUI description file (XML file). This file is a part of the MicroUI Static Initialization step (*Static Initialization*).

The order of event generators defines the unique identifier for each event generator. These identifiers are generated in a header file called microui_constants.h . The input driver (or its listener) has to use these identifiers to target a specific event generator.

If an unknown identifier is used or if two identifiers are swapped, the associated event may be never received by the application or may be misinterpreted.

Standard Event Generators

MicroUI provides a set of standard event generators: Command, Buttons, Pointer and States. For each standard generator, the Input Engine proposes a set of functions to create and send an event to this generator.

Static Initialization proposes an additional event generator: Touch . A touch event generator is a Pointer event generator whose area size is the display size where the touch panel is placed. Furthermore, contrary to a pointer, a *press* action is required to be able to have a *move* action (and so a *drag* action). The Input Engine proposes a set of functions to target a touch event generator (equal to a pointer event generator but with some constraints). The touch event generator is identified as a standard Pointer event generator, by consequence the Java application has to use the Pointer API to deal with a touch event generator.

According to the event generator, one or several parameters are required. The parameter format is event generator dependant. For instance a Pointer X-coordinate is encoded on 16 bits (0-65535 pixels).

Note: Pointer and Touch origin (point 0,0) is the top-left point.

Generic Event Generators

MicroUI provides an abstract class GenericEventGenerator (package ej.microui.event). The aim of a generic event generator is to be able to send custom events from native world to the application. These events may be constituted by only one 32-bit word or by several 32-bit words (maximum 255).

On the application side, a subclass must be implemented by clients who want to define their own event generators. Two abstract methods must be implemented by subclasses:

- eventReceived : The event generator received an event from a C driver through the Abstraction Layer API sendEvent function.
- eventsReceived : The event generator received an event made of several int s.

The event generator is responsible for converting incoming data into a MicroUI event and sending the event to its listener. It should be defined during MicroUI Static Initialization step (in the XML file, see *Static Initialization*). This allows the MicroUI implementation to instantiate the event generator on startup.

If the event generator is not available in the application classpath, a warning is thrown (with a stack trace) and the application continues. In this case, all events sent by BSP to this event generator are ignored because no event generator is able to decode them.

Abstraction Layer API

The implementation of the MicroUI Event Generator APIs provides some Abstraction Layer APIs. The BSP has to implement these Abstraction Layer APIs, making the link between the MicroUI C library inputs and the BSP input devices drivers.

The Abstraction Layer APIs to implement are listed in the header file LLUI_INPUT_impl.h . It allows events to be sent to the MicroUI implementation. The input drivers are allowed to add events directly using the event generator's unique ID (see *Static Initialization*). The drivers are fully dependent on the MicroEJ framework (a driver or a driver listener cannot be developed without MicroEJ because it uses the header file generated during the MicroUI initialization step).

To send an event to the application, the driver (or its listener) has to call one of the event engine function, listed in LLUI_INPUT.h . These functions take as parameter the MicroUI Event-Generator to target and the data. The event generator is represented by a unique ID. The data depends on the type of the event. To run correctly, the event engine requires an implementation of functions listed in LLUI_INPUT_impl.h . When an event is added, the event engine notifies MicroUI library.

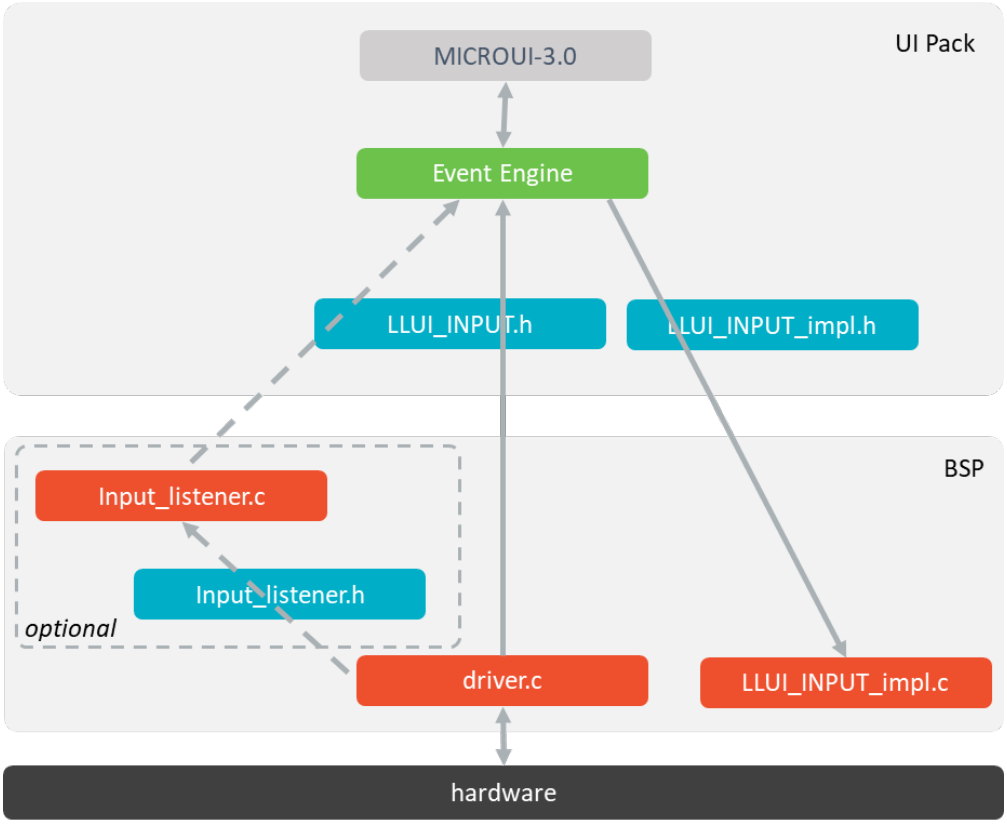


Fig. 211: Input Abstraction Layer API

When there is no input device on the board, a *stub* implementation of C library is available. This C library must be linked by the third-party C IDE when the MicroUI module is installed in the VEE Port. This stub library does not provide any Abstraction Layer API files.

Typical Implementation

This chapter helps to write a basic LLUI_INPUT_impl.h implementation. This implementation should be divided into several files:

- LLUI_INPUT_impl.c : implements LLUI_INPUT_imp.h and *receives* the input devices interrupts / callbacks (button press, touch move, etc.).
- xxx_helper.c : one helper per kind of input device (group of buttons, touch, etc.).
    - It links the input device hardware status and the software status (MicroUI event status).
    - event_generator.c : converts the input device hardware events in MicroUI events.

    The pseudo-code calls external functions such as BUTTONS_DRIVER_xxx or TOUCH_DRIVER_xxx to symbolize the use of external drivers.

LLUI_INPUT_impl.c

Its main aim is to synchronize the Input Engine with the input devices. The Input Engine holds a circular FIFO to store the input devices' events. The use of this FIFO must be performed under the critical section. The concurrent actions "an input device adds a new event in the Input Engine" and "the Input Engine reads an event from the FIFO" must not be performed simultaneously. The implementation does not need to manage the concurrency: the Input Engine automatically calls the functions LLUI_INPUT_IMPL_enterCriticalSection and LLUI_INPUT_IMPL_leaveCriticalSection when an event is added or read.

- If the input devices add events under interrupt, the critical section must disable and re-enable the input devices' interrupts.
- If the input devices add events from an OS task, the critical section must use a semaphore to prevent scheduling.
- If both modes are used (typical use case), the critical section must be designed in consequence.

  The following pseudo-code shows a typical implementation with:

- buttons under interrupt.
- touch panel from an OS task.

```
static xSemaphoreHandle _sem_input;

void LLUI_INPUT_IMPL_initialize(void)
{
  _sem_input = xSemaphoreCreateBinary();
  xSemaphoreGive(g_sem_input); // first take must pass

  BUTTONS_DRIVER_initialize();
  TOUCH_DRIVER_initialize();
}

jint LLUI_INPUT_IMPL_getInitialStateValue(jint stateMachinesID, jint stateID)
{
  // no state on this BSP
  return 0;
}

void LLUI_INPUT_IMPL_enterCriticalSection()
```

(continues on next page)

(continued from previous page)

```
{
  if (MICROEJ_FALSE == interrupt_is_in())
  {
    xSemaphoreTake(_sem_input, portMAX_DELAY);
    BUTTONS_DRIVER_disable_interrupts();
  }
  // else: already in secure state (under interrupt)
}

void LLUI_INPUT_IMPL_leaveCriticalSection()
{
  if (MICROEJ_FALSE == interrupt_is_in())
  {
    BUTTONS_DRIVER_enable_interrupts();
    xSemaphoreGive(_sem_input);
  }
  // else: already in secure state (under interrupt)
}
```

The other aim of this implementation is to *receive* the input devices' hardware events and to redirect these events to the dedicated helper.

```
// called by the touch panel dedicated task
void TOUCH_DRIVER_callback(uint8_t pressed, int32_t x, int32_t y)
{
  if (pressed)
  {
    // here, pen is down for sure
    TOUCH_HELPER_pressed(x, y);
  }
  else
  {
    // here, pen is up for sure
    TOUCH_HELPER_released();
  }
}

void GPIO_IRQHandler(int32_t button, uint32_t port, uint32_t pin)
{
  if (GPIO_PIN_SET == GPIO_ReadPin(port, pin))
  {
    // GPIO == 1 means "pressed"
    BUTTONS_HELPER_pressed(button);
  }
  else
  {
    // GPIO == 0 means "released"
    BUTTONS_HELPER_released(button);
  }
}
```

buttons_helper.c

The Input Engine's FIFO might be full. In such a case, a new input device event cannot be added. Consequently, a button *release* event should not be added to the FIFO if the previous button *press* event had not been added. This helper keeps the *software* state: the input device's state seen by the application.

Note: This helper does not convert the hardware event into a MicroUI event. It lets event_generator.c performs this job.

```
static uint8_t buttons_pressed[NUMBER_OF_BUTTONS];

void BUTTONS_HELPER_initialize(void)
{
  for(uint32_t i = 0; i < NUMBER_OF_BUTTONS; i++)
  {
    buttons_pressed[i] = MICROEJ_FALSE;
  }
}

void BUTTONS_HELPER_pressed(int32_t buttonId)
{
  // button is pressed

  if (MICROEJ_TRUE == buttons_pressed[buttonId])
  {
    // button was pressed => repeat event  (don't care if event is lost)
    EVENT_GENERATOR_button_repeated(buttonId);
  }
  else
  {
    // button was released => press event
    if (LLUI_INPUT_OK == EVENT_GENERATOR_button_pressed(buttonId) )
    {
      // the event has been managed: we can store the new button state
      // button is pressed now
      buttons_pressed[buttonId] = MICROEJ_TRUE;
    }
    // else: event has been lost: stay in "release" state
  }
}

void BUTTONS_HELPER_repeated(int32_t buttonId)
{
  // manage this repeat event like a press event to check "software" button state
  BUTTONS_HELPER_pressed(buttonId);
}

void BUTTONS_HELPER_released(int32_t buttonId)
{
  // button is now released

  if (MICROEJ_TRUE == buttons_pressed[buttonId])
  {
    // button was pressed => release event
```

(continues on next page)

(continued from previous page)

```
    if (LLUI_INPUT_OK == EVENT_GENERATOR_button_released(buttonId) )
    {
      // the event has been managed: we can store the new button state
      // button is released now
      buttons_pressed[buttonId] = MICROEJ_FALSE;
    }
    // else: event has been lost: stay in "press" state
  }
  // else: already released
}
```

touch_helper.c

The Input Engine's FIFO might be full. In such a case, a new input device event cannot be added. Consequently, a touch *move* / *drag* event should not be added to the FIFO if the previous touch *press* event had not been added. This helper keeps the *software* state: the input device's state seen by the application.

This helper also filters the touch panel events. It uses two defines FIRST_MOVE_PIXEL_LIMIT and MOVE_PIXEL_LIMIT to reduce the number of events sent to the application (values are expressed in pixels).

Note: This helper does not convert the hardware event in the MicroUI event. It lets event_generator.c performs this job.

```
// Number of pixels to generate a move after a press
#ifndef FIRST_MOVE_PIXEL_LIMIT
#error "Please set the define FIRST_MOVE_PIXEL_LIMIT (in pixels)"
#endif

// Number of pixels to generate a move after a move
#ifndef MOVE_PIXEL_LIMIT
#error "Please set the define MOVE_PIXEL_LIMIT (in pixels)"
#endif

#define DIFF(a,b)             ((a) < (b) ? (b-a) : (a-b))
#define KEEP_
↪COORD(p,n,limit)    (DIFF(p,n) <= limit ? MICROEJ_FALSE : MICROEJ_TRUE)
#define␣
↪KEEP_PIXEL(px,x,py,y,limit) (KEEP_COORD(px,x,limit) || KEEP_COORD(py,y,limit))
#define KEEP_
↪FIRST_MOVE(px,x,py,y)  (KEEP_PIXEL(px,x,py,y, FIRST_MOVE_PIXEL_LIMIT))
#define KEEP_MOVE(px,x,py,y)     (KEEP_PIXEL(px,x,py,y, MOVE_PIXEL_LIMIT))

static uint8_t touch_pressed = MICROEJ_FALSE;
static uint8_t touch_moved = MICROEJ_FALSE;
static uint16_t previous_touch_x, previous_touch_y;

void TOUCH_HELPER_pressed(int32_t x, int32_t y)
{
  // here, the pen is down for sure

  if (MICROEJ_TRUE == touch_pressed)
```

(continues on next page)

(continued from previous page)

```
  {
    // pen was down => move event

    // keep pixel according first "move" event or not
    int keep_pixel;
    if(MICROEJ_TRUE == touch_moved)
    {
      keep_pixel = KEEP_MOVE(previous_touch_x, x, previous_touch_y, y);
    }
    else
    {
      keep_pixel = KEEP_FIRST_MOVE(previous_touch_x, x, previous_touch_y, y);
    }

    if (MICROEJ_TRUE == keep_pixel)
    {
      // store the new pixel
      previous_touch_x = x;
      previous_touch_y = y;
      touch_moved = MICROEJ_TRUE;

      // send a MicroUI touch event (don't care if event is lost)
      EVENT_GENERATOR_touch_moved(x, y);
    }
    // else: same position; no need to send an event
  }
  else
  {
    // pen was up => press event
    if (LLUI_INPUT_OK == EVENT_GENERATOR_touch_pressed(x, y))
    {
      // the event has been managed: we can store the new touch state
      // touch is pressed now
      previous_touch_x = x;
      previous_touch_y = y;
      touch_pressed = MICROEJ_TRUE;
      touch_moved = MICROEJ_FALSE;
    }
    // else: event has been lost: stay in "release" state
  }
}

void TOUCH_HELPER_moved(int32_t x, int32_t y)
{
  // manage this move like a press event to check "software" touch state
  TOUCH_HELPER_pressed(x, y);
}

void TOUCH_HELPER_released(void)
{
  // here, the pen is up for sure

  if (MICROEJ_TRUE == touch_pressed)
  {
    // pen was down => release event
    if (LLUI_INPUT_OK == EVENT_GENERATOR_touch_released())
```

(continues on next page)

(continued from previous page)

```
    {
      // the event has been managed: we can store the new touch state
      // touch is released now
      touch_pressed = MICROEJ_FALSE;
    }
    // else: event has been lost: stay in "press | move" state
  }
  // else: the pen was already up
}
```

event_generator.c

This file aims to convert the events (received by LLUI_INPUT_impl.c and then filtered by xxx_helper.c) to the application through the Input Engine.

This C file should be the only C file to include the header file microui_constants.h. This header file has been generated during the VEE Port build (see *Static Initialization*). It holds some defines that describe the available list of MicroUI Event Generators. Each MicroUI Event Generator has its identifier: 0 to *n-1*.

A button event is often converted in the MicroUI Command event. That allows the application to be button-independent: the application is not waiting for button 0 or button 1 events but MicroUI Command ESC or LEFT for instance. The following pseudo-code converts the buttons events in MicroUI Command events.

Note: Each hardware event can be converted into another kind of MicroUI event. For instance, a joystick can simulate a MicroUI Pointer; a touch panel can be reduced to a set of MicroUI Commands (left, right, top, left), etc.

```
#include "microui_constants.h"

static uint32_t _get_button_command(int32_t button_id)
{
  switch (button_id)
  {
  default:
  case BUTTON_WAKEUP_ID:
    return LLUI_INPUT_COMMAND_ESC;
  case BUTTON_TAMPER_ID:
    return LLUI_INPUT_COMMAND_MENU;
  }
}

int32_t EVENT_GENERATOR_button_pressed(int32_t buttonId)
{
  return LLUI_INPUT_sendCommandEvent(MICROUI_
↪EVENTGEN_COMMANDS, _get_button_command(buttonId));
}

int32_t EVENT_GENERATOR_button_repeated(int32_t buttonId)
{
  return LLUI_INPUT_sendCommandEvent(MICROUI_
↪EVENTGEN_COMMANDS, _get_button_command(buttonId));
}
```

(continues on next page)

(continued from previous page)

```
int32_t EVENT_GENERATOR_button_released(int32_t buttonId)
{
  // do not send a Command event on the release event
  return LLUI_INPUT_OK; // the event has been managed
}

int32_t EVENT_GENERATOR_touch_pressed(int32_t x, int32_t y)
{
  return LLUI_INPUT_sendTouchPressedEvent(MICROUI_EVENTGEN_TOUCH, x, y);
}

int32_t EVENT_GENERATOR_touch_moved(int32_t x, int32_t y)
{
  return LLUI_INPUT_sendTouchMovedEvent(MICROUI_EVENTGEN_TOUCH, x, y);
}

int32_t EVENT_GENERATOR_touch_released(void)
{
  return LLUI_INPUT_sendTouchReleasedEvent(MICROUI_EVENTGEN_TOUCH);
}
```

Event Buffer

MicroUI is using a circular buffer to manage the input events. As soon as an event is added, removed, or replaced in the queue, the event engine calls the associated Abstraction Layer API (LLAPI) `LLUI_INPUT_IMPL_log_queue_xxx()`. This LLAPI allows the BSP to log this event to dump it later thanks to a call to `LLUI_INPUT_dump()`.

Note: When the functions `LLUI_INPUT_IMPL_log_queue_xxx()` are not implemented, a call to `LLUI_INPUT_dump()` has no effect (there is no default logger).

The following steps describe how the logger is called:

1. On startup, MicroUI calls `LLUI_INPUT_IMPL_log_queue_init()`: it gives the event buffer. The implementation should prepare its logger.
2. The BSP adds or replaces an event in the queue, the event engine calls `LLUI_INPUT_IMPL_log_queue_add()` or `LLUI_INPUT_IMPL_log_queue_replace()`. The implementation should store the event metadata: buffer index, event size, etc.
3. If the event cannot be added because the queue is full, the event engine calls `LLUI_INPUT_IMPL_log_queue_full()`. The implementation can print a warning, throw an error, etc.
4. MicroUI reads an event, the event engine calls `LLUI_INPUT_IMPL_log_queue_read()`. The implementation has to update its metadata (if required).

The following steps describe how the dump is performed:

1. The BSP calls `LLUI_INPUT_dump()`: the event engine starts a dump of the event buffer.
2. First, the event engine dumps the older events. It calls `LLUI_INPUT_IMPL_log_dump()` for each old event. The log type value is `0`; it means that all logs are the events or events' data already consumed (*past* events), and the first log is the latest event or data stored in the queue.

3. Then, the event engine dumps the *future* events (events not consumed yet by MicroUI). It calls `LLUI_INPUT_IMPL_log_dump()` for each new event. The log type value is `1` ; it means that all logs are the events or data not consumed yet (*future* events).
4. The *future* events can target a MicroUI object (a *Displayable* for a *requestRender* event, a *Runnable* for a *callSerially* event, etc.). The event engine notifies the logger to print the MicroUI objects by calling `LLUI_INPUT_IMPL_log_dump()` with `2` as log type value.
5. Finally, the event engine notifies the logger about the end of the dump by calling `LLUI_INPUT_IMPL_log_dump()` with `3` as log type value.

> **Warning:** The dump of MicroUI objects linked to the *future* events is only available with the MicroEJ Architectures 7.16 and higher. With older MicroEJ Architectures, nothing is dumped.

An implementation is available on the *MicroUI C module*. This logger is constituted with two files:

- `LLUI_INPUT_LOG_impl.c` : this file holds some metadata for each event. When the event engine calls `LLUI_INPUT_IMPL_log_dump()` , the logger retrieves the event metadata and calls `microui_event_decoder.c` functions. To enable this logger, set the define `UI_FEATURE_EVENT_DECODER` in `ui_configuration.h` .
- `microui_event_decoder.c` : this file describes the MicroUI events. It has to be customized with the MicroUI event generators identifiers. See `ui_configuration.h` .

Example of a dump:

```
================================
↪MicroUI FIFO Dump ================================
-------------------------------- Old Events --------------------------------
[27: 0x00000000] garbage
[28: 0x00000000] garbage
[...]
[99: 0x00000000] garbage
[00: 0x08000000] Display SHOW Displayable (Displayable index = 0)
[01: 0x00000008] Command HELP (event generator 0)
[02: 0x0d000000] Display REPAINT Displayable (Displayable index = 0)
[03: 0x07030000] Input event: Pointer pressed (event generator 3)
[04: 0x009f0063]    at 159,99 (absolute)
[05: 0x07030600] Input event: Pointer moved (event generator 3)
[06: 0x00aa0064]    at 170,100 (absolute)
[07: 0x02030700] Pointer dragged (event generator 3)
[08: 0x0d000000] Display REPAINT Displayable (Displayable index = 0)
[09: 0x07030600] Input event: Pointer moved (event generator 3)
[10: 0x00b30066]    at 179,102 (absolute)
[11: 0x02030700] Pointer dragged (event generator 3)
[12: 0x0d000000] Display REPAINT Displayable (Displayable index = 0)
[13: 0x07030600] Input event: Pointer moved (event generator 3)
[14: 0x00c50067]    at 197,103 (absolute)
[15: 0x02030700] Pointer dragged (event generator 3)
[16: 0x0d000000] Display REPAINT Displayable (Displayable index = 0)
[17: 0x07030600] Input event: Pointer moved (event generator 3)
[18: 0x00d00066]    at 208,102 (absolute)
[19: 0x02030700] Pointer dragged (event generator 3)
[20: 0x0d000000] Display REPAINT Displayable (Displayable index = 0)
[21: 0x07030100] Input event: Pointer released (event generator 3)
[22: 0x00000000]    at 0,0 (absolute)
```

(continues on next page)

(continued from previous page)

```
[23: 0x00000008] Command HELP (event generator 0)
-------------------------------- New Events --------------------------------
[24: 0x0d000000] Display REPAINT Displayable (Displayable index = 0)
[25: 0x07030000] Input event: Pointer pressed (event generator 3)
[26: 0x002a0029]    at 42,41 (absolute)
------------------------- New Events' Java objects ------------------
[java/lang/Object[2]@0xC000FD1C
    [0] com/microej/examples/microui/mvc/MVCDisplayable@0xC000BAC0
    [1] null
================================================================================
```

Notes:

- The event 24 holds an object in the events objects array (a Displayable); its object index is 0.
- An object is null when the memory slot has been used during the application execution but freed at the dump time.
- The object array' size is the maximum of non-null objects reached during application execution.
- The indices of old events are out-of-date: the memory slot is now null or reused by a newer event.
- The event 25 targets the event generator 3; the identifier is available in microui_constants.h (created during the VEE Port build, see *Static Initialization*).
- The events 27 to 99 cannot be identified (no metadata or partial event content due to circular queue management).
- Refers to the implementation on the *MicroUI C module* to have more information about the format of the event; this implementation is always up-to-date with the MicroUI implementation.

Dependencies

- MicroUI module (see *MicroUI*)
- Static MicroUI initialization step (see *Static Initialization*). This step generates a header file which contains some unique event generator IDs. These IDs must be used in the BSP to make the link between the input devices drivers and the MicroUI Event Generator s.
- LLUI_INPUT_impl.h implementation (see *LLUI_INPUT: Input*).
- The *MicroUI C module* to optionally use the *default input logger*.

Installation

Input module is a sub-part of the MicroUI library. The Input module is installed at same time than MicroUI module.

Use

The MicroUI Input APIs are available in the classes of packages ej.microui.event and ej.microui.event.generator .

Display

Principle

The Display module contains the C part of the MicroUI implementation, which manages graphical displays. This module is composed of three elements:

- the C part of MicroUI Display API (a built-in C archive) called Graphics Engine,
- an implementation of Abstraction Layer APIs for the displays (LLUI_DISPLAY) that the BSP must provide (see *LLUI_DISPLAY: Display*),
- an implementation of Abstraction Layer APIs for MicroUI drawings.

The Display module implements the MicroUI graphics framework. This framework is constituted of several notions: the display characteristics (size, format, backlight, contrast, etc.), the drawing state machine (render, flush, wait flush completed), the image life cycle, and the fonts and drawings. The main part of the Display module is provided by a built-in C archive called Graphics Engine. This library manages the drawing state machine mechanism, the images, and the fonts. The LLUI_DISPLAY implementation manages the display characteristics and the drawings.

The Graphics Engine is designed to let the BSP use an optional graphics processor unit (GPU) or an optional third-party drawing library. Each drawing can be implemented independently. If no extra framework is available, the Graphics Engine performs all drawings in software. In this case, the BSP has to perform a straightforward implementation (four functions) of the Graphics Engine Abstraction Layer.

MicroUI library also gives the possibility to perform some additional drawings that are not available as API in the MicroUI library. The Graphics Engine provides a set of functions to synchronize the drawings between them, to get the destination (and sometimes source) characteristics, to call internal software drawings, etc.

Front Panel (simulator Graphics Engine part) gives the same possibilities. The same constraints can be applied, the same drawings can be overridden or added, and the same software drawing rendering is performed (down to the pixel).

Chapters Organization

For more convenience, this chapter only describes how a display device works and how to connect it to the MicroUI Graphics Engine. Dedicated chapters deal with related concepts:

- *Buffer Refresh Strategy*: how the front buffer is refreshed.
- *Drawings*: how the drawings are performed, the use of a GPU, etc.
- *Images*: how the images are generated and drawn.
- *Fonts*: how the fonts are generated and drawn.
- *C Modules*: how the BSP extends the Graphics Engine.
- *Simulation*: how the Graphics Engine is simulated.

Display Configuration

The Graphics Engine provides a number of different configurations. The appropriate configuration should be selected depending on the capabilities of the screen and other related hardware, such as display controllers.

The policies can vary in four ways:

- the display device connection to the Graphics Engine,
- the number of buffers,
- pixel format or depth,
- the memory layout of the pixels.

Display Connection

A display is always associated with a memory buffer whose size depends on the display panel size (width and height) and the number of bits per pixel. This memory buffer holds all the pixels the display panel has to show. The display panel continuously refreshes its content by reading the data from a memory buffer. This refreshing cannot be stopped; otherwise, the image fades away. Most of the time, a new frame often appears every 16.6ms (60Hz).

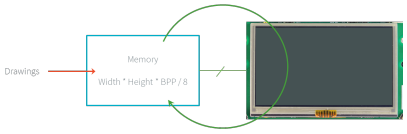


Fig. 212: Display Continuous Refresh

There are two types of connection with the MCU: Serial and Parallel.

Serial

The MCU transmits the data to show (the pixels) to the display module through a serial bus (SPI, DSI). The display module holds its memory and fills it with the received data. It continuously refreshes its content by reading the data from this memory. This memory is usually not accessible to the MCU: the MCU can only write into it with the right macro (SPI or DSI). This is the notion of **unmapped memory**.

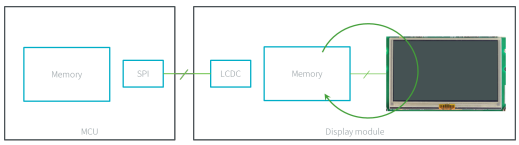


Fig. 213: Display Serial Connection

Parallel

The MCU features an LCD controller that transmits the content of an MCU's buffer to the display module. The display module doesn't hold its memory. The LCD controller continuously updates the display panel's content by reading the MCU memory data. By definition, this memory is addressed by the MCU: the MCU can write (and read) into it (the memory is in the MCU addresses range). This is the notion of **mapped memory**.

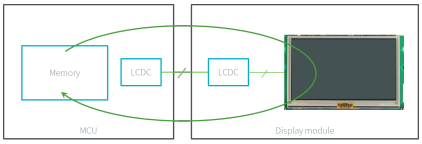


Fig. 214: Display Parallel Connection

Buffer Policy

Overview

The notion of buffer policy depends on the available number of buffers allocated in the MCU memory and on the display connection. The Graphics Engine does not depend on the type of buffer policy, and it manipulates these buffers in two steps:

1. It renders the application drawings into an MCU buffer; this buffer is called **back buffer**.
2. It *flushes* the buffer's content to the display panel; this buffer is called **front buffer**.

   The implementation of Display.flush() calls the Abstraction Layer API LLUI_DISPLAY_IMPL_flush to let the BSP update the display data.

Decision Tree

The following flow charts provide handy guides to pick the buffer policy suited to the hardware configuration.

Serial Connection

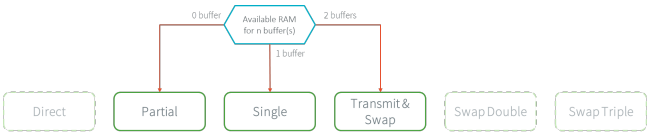


Fig. 215: Buffer Policies for Serial Connection

Parallel Connection

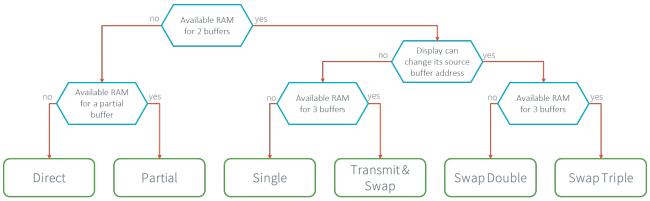


Fig. 216: Buffer Policies for Parallel Connection

Chapter Sum-up

The following table redirects to the right chapter according to the display buffer policy:

Table 61: Display Connections

Connection	Nb MCU Buffers	Chapters
Serial	partial	*Partial*
Serial	1	*Single*
Serial	2	*Transmit and Swap*
Parallel	1	*Direct*
Parallel	1 + partial	*Partial*
Parallel	2	*Swap Double* or *Single*
Parallel	3	*Swap Triple* or *Transmit and Swap*

Direct Buffer (parallel)

There is only one buffer, and the display panel continuously refreshes its content on this MCU buffer. This MCU buffer is, at the same time, the back and front buffer. Consequently, the display panel can show incomplete frames and partial drawings since the drawings can be done during the refresh cycle of the display panel. This is the notion of **direct buffer**. This buffer policy is recommended for static display-based applications and/or to save memory.

In this policy, the *flush* step has no meaning (there is only one buffer).

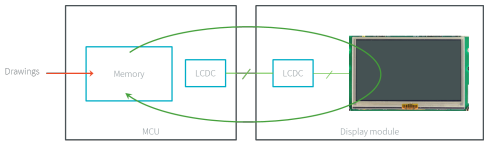


Fig. 217: Direct Buffer

Swap Double Buffer (parallel)

To prevent *flickering in the display panel*, the BSP should provide another MCU buffer (the same size as the first buffer) where the drawings are performed. The first buffer, for its part, is dedicated to the refreshing of the display panel. Double buffering avoids flickering and inconsistent rendering: it is well suited to high-quality animations. This is the notion of **double buffer**. This new buffer is usually called **back buffer**, and the first buffer is usually called **front buffer**. The two buffers in MCU memory alternately play the role of the back buffer and the front buffer. The front buffer address is alternatively changed from one buffer to the other.

The *flush* step consists in switching (or swapping) the two buffers: the front buffer becomes the back buffer, and the back buffer becomes the front buffer.

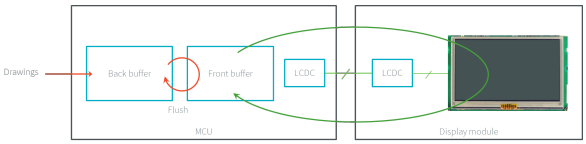


Fig. 218: Swap Double Buffer

This swap may not be atomic and may be done asynchronously: the display panel often fully refreshes an entire frame before changing its buffer address. During this time, the front buffer is used (the display panel refreshes itself on it), and the back buffer is locked (reserved for the next frame to show). Consequently, the application cannot draw again: the swapping must be performed before. As soon as the swap is done, both buffers are switched. Now, the application can draw in the new back buffer (previously the front buffer).

Swap Triple Buffer (parallel)

When the display is large, it is possible to introduce a third mapped buffer. This third buffer saves from *waiting the end of the swapping* before starting a new drawing. The buffers are usually called **back buffer 1**, **back buffer 2**, and **back buffer 3**.

The *flush* step consists in swapping two buffers and letting the application draw in the third buffer:

- The back buffer 1 is the front buffer: it is currently used by the LCD controller to refresh the display panel.
- The back buffer 2 is the next front buffer: the drawings have been done, and a *flush* is requested.
- The back buffer 3 is not used: the application can immediately draw into it without waiting for the swapping between the back buffers 1 & 2.
- When the drawings are done in the back buffer 3, this buffer becomes the next front buffer, the back buffer 2 is the front buffer, and the back buffer 1 is free.

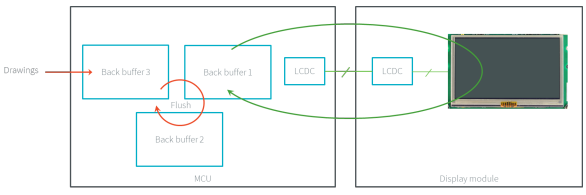


Fig. 219: Swap Triple Buffer

Single Buffer

Serial Connection

For the display connection *serial*, there are two distinct buffers: the buffer where the drawings are rendered is usually called **back buffer**, and the display module buffer **front buffer**. As long as only the back buffer is stored in the MCU-mapped memory (the front buffer is stored in the display module unmapped memory), there is only one buffer to allocate. This is the notion of **single buffer**.

The *flush* step consists in transmitting the data through the right bus (SPI, DSI).

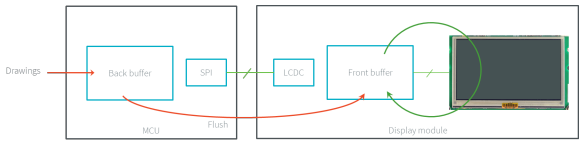


Fig. 220: Single Buffer (serial)

The display panel only shows complete frames; it cannot show partial drawings because the *flush* step is performed after all the drawings. The application cannot draw in the back buffer while the data is transmitted to the front buffer. As soon as the data is fully transmitted, the application can draw again in the back buffer.

The time to transmit the data from the back buffer to the front buffer may be long. During this time, no drawing can be anticipated, and the global framerate is reduced.

Parallel Connection

When the *swap policy* is not possible (the front buffer is mapped on a fixed MCU memory address), the policy **single buffer** can be used. Like the swap policy, this double buffering avoids flickering and inconsistent rendering: it is well suited to high-quality animations.

The *flush* step consists in copying the back buffer content to the front buffer (often by using a DMA).

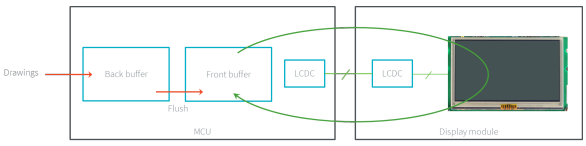


Fig. 221: Single Buffer (parallel)

When the *swap policy* can be used, the *single buffer* policy can also be used. However, there are some differences:

- In the *Swap Double* policy, the new front buffer data is available instantly. As soon as the LCD controller has updated its front buffer address, the data is ready to be read by the LCD controller. In the *Single* policy, the process of copying the data to the front buffer occurs while the LCD controller is reading it. Therefore, the buffer copy has to be faster than the LCD controller reading. If this requirement is not met, the LCD controller will read a mix of new and old data (because the buffer copy is not entirely finished).

- In the *Swap Double* policy, the synchronization with the LCD controller is more effortless. An interrupt is thrown as soon as the LCD controller has updated its front buffer address. In the *Single* policy, the copy buffer process should be synchronized with the LCD tearing signal.
- In the *Single* policy, during the copy, the destination buffer (the front buffer) is used by the copy buffer process (DMA, memcopy, etc.) and by the LCD controller. Both masters are using the same RAM section. This same RAM section switches in *Write* mode (copy buffer process) and *Read* mode (LCD controller).

Transmit and Swap Buffer

Serial Connection

When the time to transmit the data from the back buffer to the front buffer is *too long*, a second buffer can be allocated in the MCU memory. The application can use this buffer while the first buffer is transmitted. This allows to anticipate the drawings even if the first drawings are not fully transmitted. This is the notion of **transmit and swap buffer**. The buffers are usually called **back buffer 1** and **back buffer 2** (the display module's buffer is the **front buffer**).

The *flush* step consists in transmitting the back buffer data to the display module memory **and** swapping both back buffers:

- The back buffer 1 is used as *transmission* buffer.
- The back buffer 2 is not used: the application can immediately draw into it without waiting for the back buffer 1 to be transmitted.
- At the end of the drawings in the back buffer 2, the back buffer 2 takes the role of the *transmission* buffer, and the back buffer 1 is free.

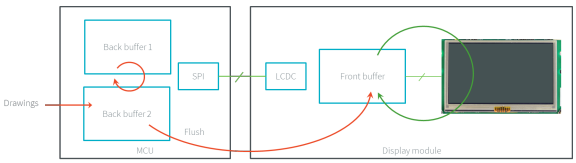


Fig. 222: Transmit and Swap (serial)

Parallel Connection

When the time to copy the data from the back buffer to the front buffer is *too long*, a third buffer can be allocated in the MCU memory. This buffer can be used by the application during the copy of the first buffer. This allows to anticipate the drawings even if the first drawings still need to be entirely copied. This is the notion of **transmit and swap buffer**. The buffers are usually called **back buffer 1** and **back buffer 2** (the third buffer is the **front buffer**). The *flush* step consists in copying the back buffer data to the front buffer **and** swapping both back buffers.

- The back buffer 1 is used as *copying* buffer.
- The back buffer 2 is not used: the application can immediately draw into it without waiting for the back buffer 1 to be copied.
- At the end of the drawings in the back buffer 2, the back buffer 2 takes the role of the *copying* buffer, and the back buffer 1 is free.

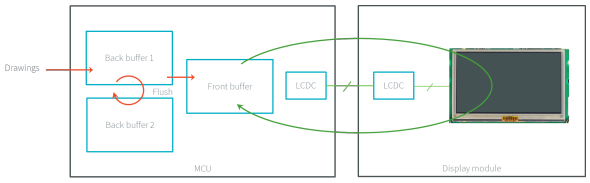


Fig. 223: Transmit and Swap (parallel)

Partial Buffer

When RAM usage is not a constraint, the back buffer is sized to store all the pixel data of the screen. But when the RAM available on the device is very limited, a partial buffer can be used instead. In that case, the buffer is smaller and can only store a part of the screen (one-third, for example).

When this technique is used, the application draws in the partial buffer. To flush the drawings, the content of the partial buffer is copied to the display (to its *internal memory* or to a *complete buffer* from which the display reads).

If the display does not have its own internal memory and if the device does not have enough RAM to allocate a complete buffer, then it is not possible to use a partial buffer. In that case, only the *direct* buffer policy can be used.

Workflow

A partial buffer of the desired size has to be allocated in RAM. If the display does not have its own internal memory, a complete buffer also has to be allocated in RAM, and the display has to be configured to read from the whole buffer.

The implementation should follow these steps:

1. First, the application draws in the partial buffer.
2. Then, to flush the drawings on the screen, the data of the partial buffer is flushed to the display (either copied to its internal memory or the complete buffer in RAM).
3. Finally, synchronization is required before starting the next drawing operation.

Dual Partial Buffer

A second partial buffer can be used to avoid the synchronization delay between two drawing cycles. While one of the two partial buffers is being copied to the display, the application can start drawing in the second partial buffer.

This technique is interesting when the copy time is long. The downside is that it either requires more RAM or requires reducing the size of the partial buffers.

Using a dual partial buffer has no impact on the application code.

Application Limitations

Using a partial buffer rather than a complete buffer may require adapting the code of the application since rendering a graphical element may require multiple passes. If the application uses MWT, a *custom render policy* has to be used.

Besides, the GraphicsContext.readPixel() and the GraphicsContext.readPixels() APIs can not be used on the graphics context of the display in partial buffer policy. Indeed, we cannot rely on the current content of the back buffer as it doesn't contain what is seen on the screen.

Likewise, the Painter.drawDisplayRegion() API can not be used in partial buffer policy. Indeed, this API reads the content of the back buffer in order to draw a region of the display. Instead of relying on the drawings that were performed previously, this API should be avoided, and the drawings should be performed again.

Using a partial buffer can have a significant impact on animation performance. Refer to *Animations* for more information on the development of animations in an application.

Implementation Example

The partial buffer demo provides an example of partial buffer implementation. This example explains how to implement partial buffer support in the BSP and how to use it in an application.

Pixel Structure

Principle

The Display module provides pre-built display configurations with a standard pixel memory layout. The layout of the bits within the pixel may be *standard* or *driver-specific*. When installing the Display module, a property `bpp` is required to specify the kind of pixel representation (see *Installation*).

Standard

When the value is one among this list: `ARGB8888` | `RGB888` | `RGB565` | `ARGB1555` | `ARGB4444` | `C4` | `C2` | `C1` , the Display module considers the pixels representation as **standard**. All standard representations are internally managed by the Display module, by the *Front Panel* and by the *Image Generator*. No specific support is required as long as a VEE Port is using a standard representation. It can:

- generate at compile-time RAW images in the same format as display pixel format,
- convert at runtime MicroUI 32-bit colors in display pixel format,
- simulate the display pixel format at runtime.

Note: The custom implementations of the image generator, some Abstraction Layer APIs, and Front Panel APIs are ignored by the Display module when a standard pixel representation is selected.

According to the chosen format, some color data can be lost or cropped.

- ARGB8888: the pixel uses 32 bits-per-pixel (alpha[8], red[8], green[8] and blue[8]).

```
u32 convertARGB8888toLCDPixel(u32 c) {
   return c;
}

u32 convertLCDPixeltoARGB8888(u32 c) {
   return c;
}
```

- RGB888: the pixel uses 24 bits-per-pixel (alpha[0], red[8], green[8] and blue[8]).

```
u32 convertARGB8888toLCDPixel(u32 c) {
   return c & 0xffffff;
}

u32 convertLCDPixeltoARGB8888(u32 c) {
   return 0
         | 0xff000000
         | c
         ;
}
```

- RGB565: the pixel uses 16 bits-per-pixel (alpha[0], red[5], green[6] and blue[5]).

```
u32 convertARGB8888toLCDPixel(u32 c) {
   return 0
         | ((c & 0xf80000) >> 8)
         | ((c & 0x00fc00) >> 5)
         | ((c & 0x0000f8) >> 3)
         ;
}

u32 convertLCDPixeltoARGB8888(u32 c) {
   return 0
         | 0xff000000
         | ((c & 0xf800) << 8)
         | ((c & 0x07e0) << 5)
         | ((c & 0x001f) << 3)
         ;
}
```

- ARGB1555: the pixel uses 16 bits-per-pixel (alpha[1], red[5], green[5] and blue[5]).

```
u32 convertARGB8888toLCDPixel(u32 c) {
   return 0
         | (((c & 0xff000000) == 0xff000000) ? 0x8000 : 0)
         | ((c & 0xf80000) >> 9)
         | ((c & 0x00f800) >> 6)
         | ((c & 0x0000f8) >> 3)
         ;
}

u32 convertLCDPixeltoARGB8888(u32 c) {
   return 0
         | ((c & 0x8000) == 0x8000 ? 0xff000000 : 0x00000000)
         | ((c & 0x7c00) << 9)
         | ((c & 0x03e0) << 6)
         | ((c & 0x001f) << 3)
```

(continues on next page)

(continued from previous page)

```
        ;
}
```

- ARGB4444: the pixel uses 16 bits-per-pixel (alpha[4], red[4], green[4] and blue[4]).

```
u32 convertARGB8888toLCDPixel(u32 c) {
   return 0
        | ((c & 0xf0000000) >> 16)
        | ((c & 0x00f00000) >> 12)
        | ((c & 0x0000f000) >> 8)
        | ((c & 0x000000f0) >> 4)
        ;
}

u32 convertLCDPixeltoARGB8888(u32 c) {
   return 0
        | ((c & 0xf000) << 16)
        | ((c & 0xf000) << 12)
        | ((c & 0x0f00) << 12)
        | ((c & 0x0f00) << 8)
        | ((c & 0x00f0) << 8)
        | ((c & 0x00f0) << 4)
        | ((c & 0x000f) << 4)
        | ((c & 0x000f) << 0)
        ;
}
```

- C4: the pixel uses 4 bits-per-pixel (grayscale[4]).

```
u32 convertARGB8888toLCDPixel(u32 c) {
   return (toGrayscale(c) & 0xff) / 0x11;
}

u32 convertLCDPixeltoARGB8888(u32 c) {
   return 0xff000000 | (c * 0x111111);
}
```

- C2: the pixel uses 2 bits-per-pixel (grayscale[2]).

```
u32 convertARGB8888toLCDPixel(u32 c) {
   return (toGrayscale(c) & 0xff) / 0x55;
}

u32 convertLCDPixeltoARGB8888(u32 c) {
   return 0xff000000 | (c * 0x555555);
}
```

- C1: the pixel uses 1 bit-per-pixel (grayscale[1]).

```
u32 convertARGB8888toLCDPixel(u32 c) {
   return (toGrayscale(c) & 0xff) / 0xff;
}

u32 convertLCDPixeltoARGB8888(u32 c) {
   return 0xff000000 | (c * 0xffffff);
}
```

Driver-Specific

The Display module considers the pixel representation as **driver-specific** when the value is one among this list: `1 | 2 | 4 | 8 | 16 | 24 | 32`. This mode is often used when the pixel representation is not `ARGB` or `RGB` but `BGRA` or `BGR` instead. This mode can also be used when the number of bits for a color component (alpha, red, green, or blue) is not standard or when the value does not represent a color but an index in a *CLUT*. This mode requires some specific support in the VEE Port:

- An extension of the image generator is mandatory: see *Extended Mode* to convert MicroUI's standard 32-bit ARGB colors to display pixel format.
- The Front Panel widget `Display` requires an extension to convert the MicroUI 32-bit colors in display pixel format and vice-versa, see *Display Widget*.
- The driver must implement functions that convert MicroUI's standard 32-bit ARGB colors to display pixel format and vice-versa: see *Color Conversions*.

The following example illustrates the use of specific format BGR565 (the pixel uses 16 bits-per-pixel (alpha[0], red[5], green[6] and blue[5]):

1. Configure the VEE Port:

   SDK 6

   SDK 5

   In SDK 6, the configuration is done in the properties file `configuration.properties` of the VEE Port project:

```
com.microej.pack.display.bpp=16
```

In SDK 5, the configuration is done in the properties file `display/display.properties`:

```
bpp=16
```

2. Image Generator:

- Create a project as described *here*.
- Create the class `com.microej.graphicalengine.generator.MicroUIGeneratorExtension` that extends the class `com.microej.tool.ui.generator.BufferedImageLoader`.
- Fill the method `convertARGBColorToDisplayColor()`:

```
public class MicroUIGeneratorExtension extends BufferedImageLoader {
  @Override
  public int convertARGBColorToDisplayColor(int color) {
    ␣
↪   return ((color & 0xf80000) >> 19) | ((color & 0x00fc00) >> 5) | ((color & 0x0000f8) << 8);
  }
}
```

- Configure the Image Generator' service loader: add the file `/META-INF/services/com.microej.tool.ui.generator.MicroUIRawImageGeneratorExtension`:

```
com.microej.graphicalengine.generator.MicroUIGeneratorExtension
```

- Build the module (click on the blue button).

- Copy the generated jar file (imageGeneratorMyPlatform.jar) in the VEE Port configuration project: /dropins/tools/ .

2. Simulator (Front Panel):

- Create the class com.microej.fp.MyDisplayExtension that implements the interface ej.fp.widget.Display.DisplayExtension :

```
public class MyDisplayExtension implements DisplayExtension {

  @Override
  public int convertARGBColorToDisplayColor(Display display, int color) {
  ␣
↪ return ((color & 0xf80000) >> 19) | ((color & 0x00fc00) >> 5) | ((color & 0x0000f8) << 8);
  }

  @Override
  public int convertDisplayColorToARGBColor(Display display, int color) {
    return␣
↪((color & 0x001f) << 19) | ((color & 0x7e00) << 5) | ((color & 0xf800) >> 8) | 0xff000000;
  }

  @Override
  public boolean isColor(Display display) {
    return true;
  }

  @Override
  public int getNumberOfColors(Display display) {
    return 1 << 16;
  }
}
```

- Configure the widget Display in the .fp file by referencing the display extension:

```
<ej.fp.widget.Display x="41"␣
↪y="33" width="320" height="240" extensionClass="com.microej.fp.MyDisplayExtension"/>
```

3. Build the VEE Port as usual

4. Update the LLUI_DISPLAY implementation by adding the following functions:

```
uint32_t LLUI_DISPLAY_IMPL_convertARGBColorToDisplayColor(uint32_t color) {
  return ((color & 0xf80000) >> 19) | ((color & 0x00fc00) >> 5) | ((color & 0x0000f8) << 8);
}

uint32_t LLUI_DISPLAY_IMPL_convertDisplayColorToARGBColor(uint32_t color) {
 return␣
↪((color & 0x001f) << 19) | ((color & 0x7e00) << 5) | ((color & 0xf800) >> 8) | 0xff000000;
}
```

CLUT

The Display module allows the targeting of a display that uses a pixel indirection table (CLUT). This kind of display is considered as generic but not standard (see *Pixel Structure*). It consists in storing color indexes in the image memory buffer instead of colors themselves.

Color Conversion

The driver must implement functions that convert MicroUI's standard 32-bit ARGB colors (see *LLUI_DISPLAY: Display*) to display color representation. For each application ARGB8888 color, the display driver has to find the corresponding color in the table. The Graphics Engine will store the index of the color in the table instead of using the color itself.

When an application color is not available in the display driver table (CLUT), the display driver can try to find the closest color or return a default color. The first solution is often quite tricky to write and can cost a lot of time at runtime. That's why the second solution is preferred. However, a consequence is that the application only uses a range of colors provided by the display driver.

Alpha Blending

MicroUI and the Graphics Engine use blending when drawing some texts or anti-aliased shapes. For each pixel to draw, the display stack blends the current application foreground color with the targeted pixel's current color or with the current application background color (when enabled). This blending *creates* some intermediate colors which the display driver manages.

Most of the time, the intermediate colors do not match with the palette. The default color is so returned, and the rendering becomes wrong. To prevent this use case, the Graphics Engine offers a specific Abstraction Layer API `LLUI_DISPLAY_IMPL_prepareBlendingOfIndexedColors(void *foreground, void *background)`.

This API is only used when a blending is required and when the background color is enabled. The Graphics Engine calls the API just before the blending and gives as a parameter the pointers on both ARGB colors.The display driver should replace the ARGB colors with the CLUT indexes. Then, the Graphics Engine will only use between both indexes.

For instance, when the returned indexes are `20` and `27`, the display stack will use the indexes `20` to `27`, where all indexes between `20` and `27` target some intermediate colors between both the original ARGB colors.

This solution requires several conditions:

- Background color is enabled, and it is an available color in the CLUT.
- The application can only use foreground colors provided by the CLUT. The VEE Port designer should give to the application developer the available list of colors the CLUT manages.
- The CLUT must provide a set of blending ranges the application can use. Each range can have its own size (different number of colors between two colors). Each range is independent. For instance, if the foreground color `RED` (`0xFFFF0000`) can be blended with two background colors `WHITE` (`0xFFFFFFFF`) and `BLACK` (`0xFF000000`), two ranges must be provided. Both the ranges have to contain the same index for the color `RED`.
- Application can only use blending ranges provided by the CLUT. Otherwise, the display driver is not able to find the range, and the default color will be used to perform the blending.

- Rendering of dynamic images (images decoded at runtime) may be wrong because the ARGB colors may be out of the CLUT range.

Memory Layout

For the display with a number of bits-per-pixel (BPP) higher or equal to 8, the Graphics Engine supports the line-by-line memory organization: pixels are laid out from left to right within a line, starting with the top line. For a display with 16 bits-per-pixel, the pixel at (0,0) is stored at memory address 0, the pixel at (1,0) is stored at address 2, the pixel at (2,0) is stored at address 4, and so on.

Table 62: Memory Layout for BPP >= 8

BPP	@ + 0	@ + 1	@ + 2	@ + 3	@ + 4
32	pixel 0 [7:0]	pixel 0 [15:8]	pixel 0 [23:16]	pixel 0 [31:24]	pixel 1 [7:0]
24	pixel 0 [7:0]	pixel 0 [15:8]	pixel 0 [23:16]	pixel 1 [7:0]	pixel 1 [15:8]
16	pixel 0 [7:0]	pixel 0 [15:8]	pixel 1 [7:0]	pixel 1 [15:8]	pixel 2 [7:0]
8	pixel 0 [7:0]	pixel 1 [7:0]	pixel 2 [7:0]	pixel 3 [7:0]	pixel 4 [7:0]

For the display with a number of bits-per-pixel (BPP) lower than 8, the Graphics Engine supports both memory organizations: line by line (pixels are laid out from left to right within a line, starting with the top line) and column by column (pixels are laid out from top to bottom within a line, starting with the left line). These byte organizations concern until 8 consecutive pixels (see *Byte Layout*). When installing the Display module, a property `memoryLayout` is required to specify the kind of pixel representation (see *Installation*).

Table 63: Memory Layout 'line' for BPP < 8 and byte layout 'line'

BPP	@ + 0	@ + 1	@ + 2	@ + 3	@ + 4
4	(0,0) to (1,0)	(2,0) to (3,0)	(4,0) to (5,0)	(6,0) to (7,0)	(8,0) to (9,0)
2	(0,0) to (3,0)	(4,0) to (7,0)	(8,0) to (11,0)	(12,0) to (15,0)	(16,0) to (19,0)
1	(0,0) to (7,0)	(8,0) to (15,0)	(16,0) to (23,0)	(24,0) to (31,0)	(32,0) to (39,0)

Table 64: Memory Layout 'line' for BPP < 8 and byte layout 'column'

BPP	@ + 0	@ + 1	@ + 2	@ + 3	@ + 4
4	(0,0) to (0,1)	(1,0) to (1,1)	(2,0) to (2,1)	(3,0) to (3,1)	(4,0) to (4,1)
2	(0,0) to (0,3)	(1,0) to (1,3)	(2,0) to (2,3)	(3,0) to (3,3)	(4,0) to (4,3)
1	(0,0) to (0,7)	(1,0) to (1,7)	(2,0) to (2,7)	(3,0) to (3,7)	(4,0) to (4,7)

Table 65: Memory Layout 'column' for BPP < 8 and byte layout 'line'

BPP	@ + 0	@ + 1	@ + 2	@ + 3	@ + 4
4	(0,0) to (1,0)	(0,1) to (1,1)	(0,2) to (1,2)	(0,3) to (1,3)	(0,4) to (1,4)
2	(0,0) to (3,0)	(0,1) to (3,1)	(0,2) to (3,2)	(0,3) to (3,3)	(0,4) to (3,4)
1	(0,0) to (7,0)	(0,1) to (7,1)	(0,2) to (7,2)	(0,3) to (7,3)	(0,4) to (7,4)

Table 66: Memory Layout 'column' for BPP < 8 and byte layout 'column'

BPP	@ + 0	@ + 1	@ + 2	@ + 3	@ + 4
4	(0,0) to (0,1)	(0,2) to (0,3)	(0,4) to (0,5)	(0,6) to (0,7)	(0,8) to (0,9)
2	(0,0) to (0,3)	(0,4) to (0,7)	(0,8) to (0,11)	(0,12) to (0,15)	(0,16) to (0,19)
1	(0,0) to (0,7)	(0,8) to (0,15)	(0,16) to (0,23)	(0,24) to (0,31)	(0,32) to (0,39)

Byte Layout

This chapter concerns only displays with a number of bits-per-pixel (BPP) smaller than 8. For this kind of display, a byte contains several pixels, and the Graphics Engine allows to customize how to organize the pixels in a byte.

Two layouts are available:

- line: The byte contains several consecutive pixels on the same line. When the end of the line is reached, padding is added in order to start a new line with a new byte.
- column: The byte contains several consecutive pixels on the same column. When the end of the column is reached, padding is added in order to start a new column with a new byte.

When installing the Display module, a property `byteLayout` is required to specify the kind of pixel representation (see *Installation*).

Table 67: Byte Layout: line

BPP	MSB							LSB
4	pixel 1				pixel 0			
2	pixel 3		pixel 2		pixel 1		pixel 0	
1	pixel 7	pixel 6	pixel 5	pixel 4	pixel 3	pixel 2	pixel 1	pixel 0

Table 68: Byte Layout: column

BPP	4	2	1
MSB	pixel 1	pixel 3	pixel 7
			pixel 6
		pixel 2	pixel 5
			pixel 4
	pixel 0	pixel 1	pixel 3
			pixel 2
		pixel 0	pixel 1
LSB			pixel 0

Display Synchronization

Overview

The Graphics Engine is designed to be synchronized with the display refresh rate by defining some points in the rendering timeline. It is optional; however, it is mainly recommended. This chapter explains why to use display tearing signal and its consequences. Some chronograms describe several use cases: with and without display tearing signal, long drawings, long flush time, etc. Times are in milliseconds. To simplify chronograms views, the display refresh rate is every 16ms (62.5Hz).

Captions definition:

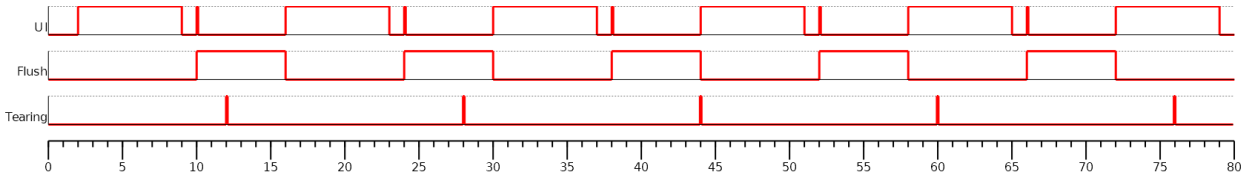
- UI: It is the UI task that performs the drawings in the back buffer. At the end of the drawings, the examples consider that the UI thread calls Display.flush() 1 millisecond after the end of the drawings. At this moment, a flush can start (the call to Display.flush() is symbolized by a simple *peak* in chronograms).
- Flush: In *single buffer* policy, it is the time to flush the content of the back buffer to the front buffer. In *double* or *triple* policy, it is the time to swap back and front buffers (the instruction

is often instantaneous but the action is usually performed at the beginning of the next display refresh rate). During this time, the back buffer is *in use*, and the UI task has to wait until the end of the swap before starting a new drawing.

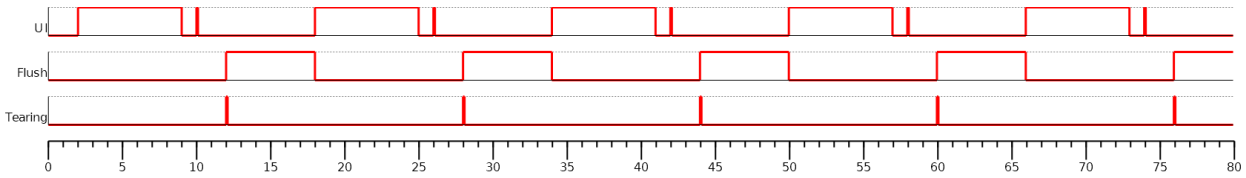
- Tearing: The peaks show the tearing signals.
- Rendering frequency: the frequency between the start of a drawing and the end of the flush.

Tearing Signal

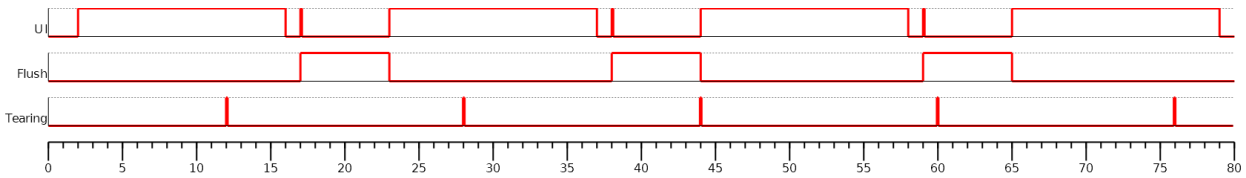
In this example, the drawing time is 7ms, the time between the end of the drawing and the call to Display.flush() is 1ms, and the flush time is 6ms. So the expected rendering frequency is 7 + 1 + 6 = 14ms (71.4Hz). Flush starts just after the call to Display.flush(), and the next drawing starts just after the end of flush. Tearing signal is not taken into consideration. As a consequence, the display content is refreshed during the display refresh time. The content can be corrupted: flickering, glitches, etc. The rendering frequency is faster than the display refresh rate.



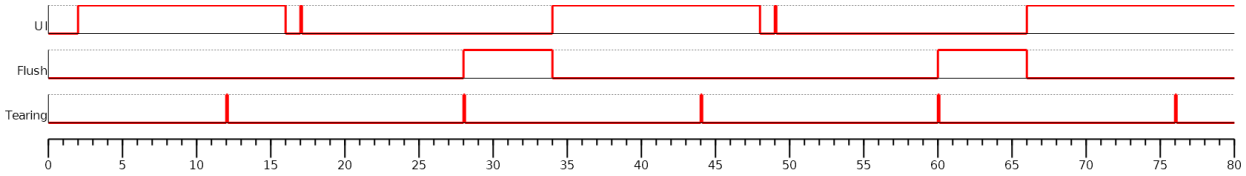
In this example, the times are identical to the previous example. The tearing signal is used to start the flush to respect the display refreshing time. The rendering frequency becomes smaller: it is cadenced on the tearing signal every 16ms (62.5Hz). During 2ms, the CPU can schedule other tasks or go into idle mode. The rendering frequency is equal to the display refresh rate.



In this example, the drawing time is 14ms, the time between the end of the drawing and the call to Display.flush() is 1ms, and the flush time is 6ms. So the expected rendering frequency is 14 + 1 + 6 = 21ms (47.6Hz). Flush starts just after the call to Display.flush(), and the next drawing starts just after the end of flush. Tearing signal is not taken into consideration.



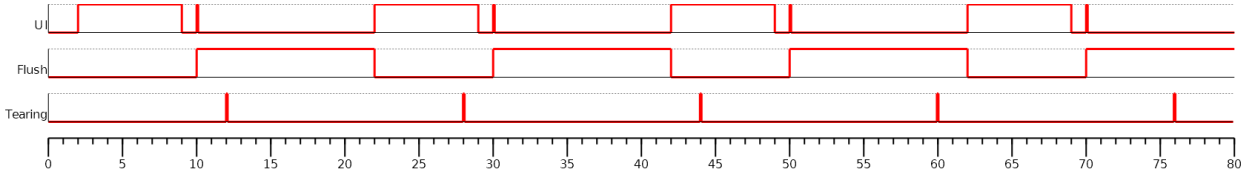
In this example, the times are identical to the previous example. The tearing signal is used to start the flush to respect the display refreshing time. The drawing time + flush time is higher than the display tearing signal period. So, the flush cannot start at every tearing peak: it is cadenced on two tearing signals every 32ms (31.2Hz). During 11ms, the CPU can schedule other tasks or go into idle mode. The rendering frequency is equal to the display refresh rate divided by two.



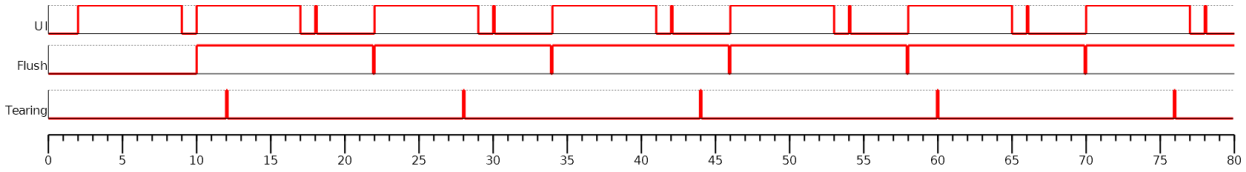
Additional Buffer

Some devices take a lot of time to flush the back buffer content to the front buffer. The following examples demonstrate the consequence of rendering frequency. The use of an additional buffer optimizes this frequency; however, it uses a lot of RAM.

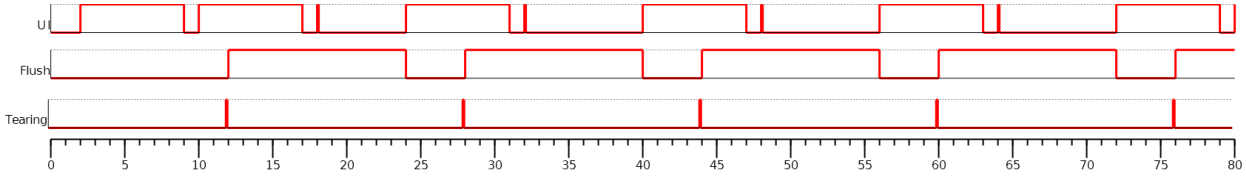
In this example, the drawing time is 7ms, the time between the end of the drawing and the call to Display.flush() is 1ms, and the flush time is 12ms. So the expected rendering frequency is 7 + 1 + 12 = 20ms (50Hz). Flush starts just after the call to Display.flush(), and the next drawing starts just after the end of flush. Tearing signal is not taken into consideration. The rendering frequency is cadenced on drawing time + flush time.



As mentioned above, the idea is to use *two back buffers*. First, the UI task is drawing in the back buffer A. Just after the call to Display.flush(), the flush can start. During the flush time (copy of the back buffer A to the front buffer), the back buffer B can be used by the UI task to continue the drawings. When the drawings in the back buffer B are done (and after the call to Display.flush()), the application cannot start a third frame by drawing into the back buffer A because the flush is using it. As soon as the flush is done, a new flush (of the back buffer B) can start. The rendering frequency is cadenced on flush time, i.e., 12ms (83.3Hz).



The previous example doesn't take into consideration the display tearing signal. With a tearing signal and only one back buffer, the frequency is cadenced on two tearing signals (see above). With two back buffers, the frequency is now cadenced on only one tearing signal despite the long flush time.



Time Sum-up

The following table resumes the previous examples times:

- It considers the display frequency is 62.5Hz (16ms).
- *Drawing time* is the time left for the application to perform its drawings and call Display.flush(). In our examples, the time between the last drawing and the call to Display.flush() is 1 ms.
- *FPS* and *CPU load* are calculated from examples times.
- *Max drawing time* is the maximum time left for the application to perform its drawings without overlapping the next display tearing signal (when tearing is enabled).

Tear-ing	Nb buffers	Drawing time (ms)	Flush time (ms)	FPS (Hz)	CPU load (%)	Max drawing time (ms)
no	1	7+1	6	71.4	57.1	
yes	1	7+1	6	62.5	50	10
no	1	14+1	6	47.6	71.4	
yes	1	14+1	6	31.2	46.9	20
no	1	7+1	12	50	40	
yes	1	7+1	12	31.2	25	8
no	2	7+1	12	83.3	66.7	
yes	2	7+1	12	62.5	50	16

Abstraction Layer API

Overview

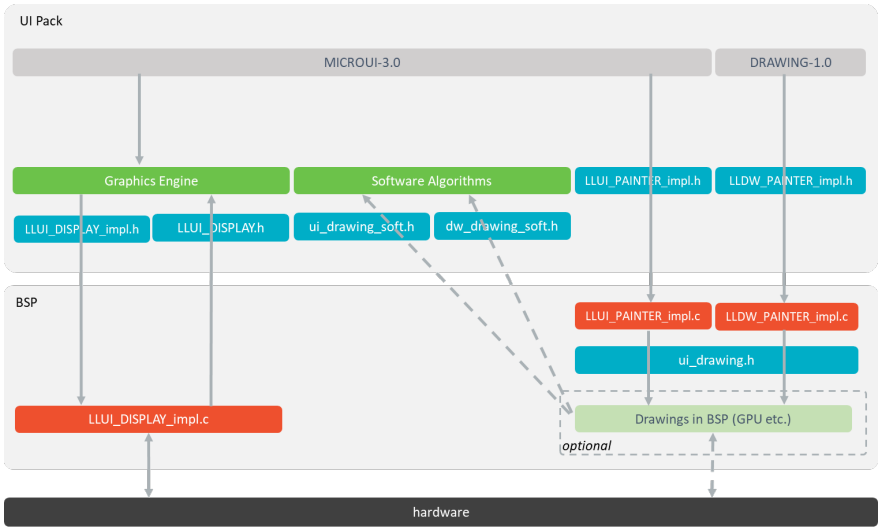


Fig. 224: Display Abstraction Layer API

- MicroUI library calls the BSP functions through the Graphics Engine and header file LLUI_DISPLAY_impl.h .

- Implementation of LLUI_DISPLAY_impl.h can call Graphics Engine functions through LLUI_DISPLAY.h .
- To perform some drawings, MicroUI uses LLUI_PAINTER_impl.h functions.
- The *MicroUI C module* provides a default implementation of the drawing native functions of LLUI_PAINTER_impl.h and LLDW_PAINTER_impl.h :

– It implements the synchronization layer, then redirects drawings implementations to ui_drawing.h .

– ui_drawing.h is already implemented by built-in software algorithms (library provided by the UI Pack).

– It is possible to implement some of the ui_drawing.h functions in the BSP to provide a custom implementation (for instance, a GPU).

– Custom implementation is still allowed to call software algorithms declared in ui_drawing_soft.h and dw_drawing_soft.h .

Display Size

The Abstraction Layer distinguishes between the display *virtual* size and the display *physical* size (in pixels).

- The display *virtual* size is the size of the area where the drawings are visible. Virtual memory size is: lcd_width * lcd_height * bpp / 8 .
- The display *physical* size is the required memory size where the *virtual* area is located. On some devices, the memory width (in pixels) is higher than the virtual width. In this way, the graphics buffer memory size is: memory_width * memory_height * bpp / 8 .

Note: The *physical* size may not be configured; in that case, the Graphics Engine considers the *virtual* size os *physical* size.

Semaphores

The Graphics Engine requires two binary semaphores to synchronize its internal states. These semaphores are reserved for the Graphics Engine. The LLUI_DISPLAY_impl.h implementation is not allowed to use these semaphores to synchronize the function LLUI_DISPLAY_IMPL_flush() with the display driver (or for any other synchronization actions). The implementation must create its semaphores in addition to these dedicated Graphics Engine's semaphores.

The binary semaphores must be configured in a state such that the semaphore must first be *given* before it can be *taken* (this initialization must be performed in the LLUI_DISPLAY_IMPL_initialize function).

Required Abstraction Layer API

Four Abstraction Layer APIs are required to connect the Graphics Engine to the display driver. The functions are listed in LLUI_DISPLAY_impl.h .

- LLUI_DISPLAY_IMPL_initialize : The initialization function is called when the application is calling MicroUI.start(). Before this call, the display is useless and doesn't need to be initialized. This function consists in initializing the LCD driver and filling the given structure LLUI_DISPLAY_SInitData . This structure has to contain pointers on the two binary semaphores, the back buffer address (see *Display Configuration*), the display *virtual* size in pixels (lcd_width and lcd_height), and optionally the display *physical* size in pixels (memory_width and memory_height).
- LLUI_DISPLAY_IMPL_binarySemaphoreTake and LLUI_DISPLAY_IMPL_binarySemaphoreGive : Two distinct functions have to be implemented to *take* and *give* a binary semaphore.
- LLUI_DISPLAY_IMPL_flush : According to the display buffer policy (see *Display Configuration*), the flush function has to be implemented. This function must not block and not perform the flush directly. Another OS task or dedicated hardware must be configured to perform the flush.

Optional Abstraction Layer API

Several optional Abstraction Layer APIs are available in LLUI_DISPLAY_impl.h . They are already implemented as *weak* functions in the Graphics Engine and return no error. These optional features concern the display backlight and contrast, display characteristics (is colored display, double buffer), color conversions (see *Pixel Structure* and *CLUT*), etc. Refer to each function comment to have more information about the default behavior.

Painter Abstraction Layer API

All MicroUI drawings (available in the Painter class) call a native function. The MicroUI native drawing functions are listed in LLUI_PAINTER_impl.h . The principle of implementing a MicroUI drawing function is described in the chapter *Drawings*.

Graphics Engine API

The Graphics Engine provides a set of functions to interact with the C archive. The functions allow the retrieval of some drawing characteristics, the synchronization of drawings between them, the notification of the end of flush and drawings, etc.

The functions are available in LLUI_DISPLAY.h .

Most APIs are thread-safe, meaning that any C task can call the Graphics Engine APIs at any time. However, some functions need to be synchronized with MicroUI (and the Core Engine task), including allocateImageBuffer() , freeImageBuffer() , and decodeImage() .

Note: This list is not exhaustive; please refer to the C documentation for LLUI_DISPLAY_IMPL_lockJob() for the complete list.

To facilitate this synchronization, the Graphics Engine provides optional supplementary Abstract Layer API: LLUI_DISPLAY_IMPL_lockJob() and LLUI_DISPLAY_IMPL_unlockJob() . The implementation of these APIs simply involves using a binary semaphore.

Important: This binary semaphore must not be shared with the *binary semaphores* given to the Graphics Engine during its initialization.

Typical Implementations

This chapter helps to write some basic LLUI_DISPLAY_impl.h implementations according to the display buffer policy (see *Display Configuration*). The pseudo-code calls external functions such as LCD_DRIVER_xxx or DMA_DRIVER_xxx to symbolize the use of external drivers.

Note: The pseudo code does not use the const ui_rect_t areas[] bounds to simplify the reading.

Common Functions

The three functions LLUI_DISPLAY_IMPL_initialize , LLUI_DISPLAY_IMPL_binarySemaphoreTake and LLUI_DISPLAY_IMPL_binarySemaphoreGive are often the same. The following example shows an implementation with FreeRTOS.

```
void LLUI_DISPLAY_IMPL_initialize(LLUI_DISPLAY_SInitData *init_data) {
  // fill the LLUI_DISPLAY_SInitData structure
  init_data->binary_semaphore_0 = (void*)xSemaphoreCreateBinary();
  init_data->binary_semaphore_1 = (void*)xSemaphoreCreateBinary();
  init_data->lcd_width = LCD_DRIVER_get_width();
  init_data->lcd_height = LCD_DRIVER_get_height();
  /* init_data->back_buffer_address = [...]; see next chapters */
}

void LLUI_DISPLAY_IMPL_binarySemaphoreTake(void *sem) {
  xSemaphoreTake((xSemaphoreHandle)sem, portMAX_DELAY);
}

void LLUI_DISPLAY_IMPL_binarySemaphoreGive(void *sem, bool under_isr) {
  if (under_isr) {
    portBASE_TYPE xHigherPriorityTaskWoken = pdFALSE;
    xSemaphoreGiveFromISR((xSemaphoreHandle)sem, &xHigherPriorityTaskWoken);
    if (xHigherPriorityTaskWoken != pdFALSE) {
      // Force a context switch here.
      portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
    }
  }
  else {
    xSemaphoreGive((xSemaphoreHandle)sem);
  }
}
```

Direct Buffer (parallel)

This policy considers the application and the LCD driver share the same buffer. In other words, all drawings made by the application are immediately shown on the display. This particular case is the easiest to write because the flush() stays empty:

```
void LLUI_DISPLAY_IMPL_initialize(LLUI_DISPLAY_SInitData *init_data) {
  // [...]

  // use same buffer between the LCD driver and the Graphics Engine
  LCD_DRIVER_initialize(lcd_buffer);
  init_data->back_buffer_address = lcd_buffer;
}

void LLUI_DISPLAY_IMPL_flush(MICROUI_
↪GraphicsContext *gc, uint8_t flush_identifier, const ui_rect_t areas[], size_t length) {
  // nothing to␣
↪flush to the LCD, just have to unlock the Graphics Engine by giving the same buffer address
  LLUI_DISPLAY_
↪setBackBuffer(flush_identifier, LLUI_DISPLAY_getBufferAddress(&gc->image), false);
}
```

Swap Double Buffer (parallel)

This buffer policy requires two buffers in RAM. The first buffer is used by the application (buffer A), and the LCD controller uses the second buffer to update the display panel (buffer B). The LCD controller is reconfigured to use the buffer A when the Graphics Engine calls the flush() function.

> Before executing the next application drawing after a flush, the Graphics Engine automatically waits for the end of the flush buffer processing: buffer B (which currently used by the LDC controller) is updated at the end of the swap. The LCD driver is responsible for unlocking the Graphics Engine by calling the function LLUI_DISPLAY_setBackBuffer() at the end of the swap.

```
static uint8_t *buffer_A;
static uint8_t *buffer_B;
static uint8_t _flush_identifier;

void LLUI_DISPLAY_IMPL_initialize(LLUI_DISPLAY_SInitData *init_data) {
  // [...]

  // use two distinct buffers between the LCD driver and the Graphics Engine
  LCD_DRIVER_initialize(buffer_B);
  init_data->back_buffer_address = buffer_A;
}

void LLUI_DISPLAY_IMPL_flush(MICROUI_
↪GraphicsContext *gc, uint8_t flush_identifier, const ui_rect_t areas[], size_t length) {
  // store the identifier of the flush used to unlock the Graphics Engine later
  _flush_identifier = flush_identifier;

  // change the LCDC address (executed at the next LCD refresh loop)
  LCDC_set_address(LLUI_DISPLAY_getBufferAddress(&gc->image));
}

// only called when reloading a new LCDC address
```

(continues on next page)

(continued from previous page)

```
void LCDC_RELOAD_IRQHandler(void) {
  LCDC_DRIVER_clear_interrupt();

  // end of the swap, unlock the Graphics Engine, update the back buffer address
  uint8_t *new_back_buffer = (LCDC_get_address() == buffer_A) ? buffer_B : buffer_A;
  LLUI_DISPLAY_
↪setBackBuffer(_flush_identifier, new_back_buffer, true); // true: called under interrupt
}
```

Swap Triple Buffer (parallel)

The behavior of this policy is very similar to that of the double buffer policy (see above): it consists in alternating between three buffers instead of two.

Single Buffer (serial)

A display connected to the CPU through a serial bus (DSI, SPI, etc.) requires the *single buffer* policy: the application uses a buffer to perform its drawings, and the buffer's content has to be transmitted to the display when the Graphics Engine is calling the `flush()` function.

The specification of the `flush()` function is to be **not** blocker (atomic). Its aim is to prepare / configure the serial bus and data to transmit and then to start the asynchronous transmission. The `flush()` function has to return as soon as possible.

Before executing the next application drawing after a flush, the Graphics Engine automatically waits for the end of the serial data transmission: the back buffer (currently used by the serial device) is updated at the end of data transmission. The serial device driver is responsible for unlocking the Graphics Engine by calling the function `LLUI_DISPLAY_setBackBuffer()` at the end of the transmission.

There are two use cases:

Hardware

The serial data transmission is performed in hardware. In that case, the serial driver must configure an interrupt to be notified about the end of the transmission.

```
static uint8_t _flush_identifier;

void LLUI_DISPLAY_IMPL_initialize(LLUI_DISPLAY_SInitData *init_data) {
  // [...]

  LCD_DRIVER_initialize();
  init_data->back_buffer_address = back_buffer;

  // initialize the serial driver & device: GPIO, etc.
  SERIAL_DRIVER_initialize();
}

void LLUI_DISPLAY_IMPL_flush(MICROUI_
↪GraphicsContext *gc, uint8_t flush_identifier, const ui_rect_t areas[], size_t length) {
  // store the identifier of the flush used to unlock the Graphics Engine later
  _flush_identifier = flush_identifier;
```

(continues on next page)

(continued from previous page)

```
  // configure the serial device to transmit n bytes
  // srcAddr == back_buffer
  SERIAL_
↪DRIVER_prepare_transmit(srcAddr, LCD_WIDTH * LCD_HEIGHT * LCD_BPP / 8);

  // configure the "end of transmission" interrupt
  SERIAL_DRIVER_enable_interrupt(END_OF_TRANSMIT);

  // start the transmission
  SERIAL_DRIVER_start();
}

void SERIAL_DEVICE_IRQHandler(void) {
  SERIAL_DRIVER_clear_interrupt();
  SERIAL_DRIVER_disable_interrupt(END_OF_TRANSMIT);

  // end of transmission, unlock the Graphics Engine without changing the back buffer address
  LLUI_DISPLAY_
↪setBackBuffer(_flush_identifier, back_buffer, true); // true: called under interrupt
}
```

Software

The serial data transmission cannot be performed in hardware or requires a software loop to transmit all data. This transmission must not be performed in the flush() function (see above). A dedicated OS task is required to perform this transmission.

```
static void *_transmit_task_semaphore;
static uint8_t _flush_identifier;

static void _task_flush(void *p_arg) {
  while(1) {
    // wait until the Graphics Engine gives the order to flush
    LLUI_DISPLAY_IMPL_binarySemaphoreTake(_transmit_task_semaphore);

    // transmit data
    SERIAL_
↪DRIVER_transmit_data(back_buffer, LCD_WIDTH * LCD_HEIGHT * LCD_BPP / 8);

    // end of flush, unlock the Graphics Engine without changing the back buffer address
    LLUI_DISPLAY_
↪setBackBuffer(_flush_identifier, back_buffer, false); // false: called outside interrupt
  }
}

void LLUI_DISPLAY_IMPL_initialize(LLUI_DISPLAY_SInitData *init_data) {
  // [...]

  LCD_DRIVER_initialize();
  init_data->back_buffer_address = back_buffer;

  // create a "flush" task and a dedicated semaphore
  _transmit_task_semaphore = (void*)xSemaphoreCreateBinary();
  xTaskCreate(_task_flush, "FlushTask", 1024, NULL, 12, NULL);
}

void LLUI_DISPLAY_IMPL_flush(MICROUI_
```

(continues on next page)

(continued from previous page)

```
↪GraphicsContext *gc, uint8_t flush_identifier, const ui_rect_t areas[], size_t length) {
  // store the identifier of the flush used to unlock the Graphics Engine later
  _flush_identifier = flush_identifier;

  // unlock the flush task
  LLUI_DISPLAY_IMPL_binarySemaphoreGive(_transmit_task_semaphore, false);
}
```

Single Buffer (parallel) and Tearing Disabled

Note: This policy should synchronize the copy buffer process with the LCD tearing signal. However, this notion is sometimes not available. This chapter describes the copy buffer process without using the tearing signal (see *next chapter*).

This buffer policy requires two buffers in RAM. The first buffer is used by the application (back buffer), and the LCD controller uses the second buffer to update the display panel (front buffer). The content of the front buffer must be updated with the content of the back buffer when the Graphics Engine is calling the `flush()` function.

The specification of the `flush()` function is to be **not** blocker (atomic, see above). Its aim is to prepare / configure the copy buffer process and then start the asynchronous copy. The `flush()` function has to return as soon as possible.

Before executing the next application drawing after a flush, the Graphics Engine automatically waits for the end of the copy buffer process: the back buffer (currently used by the copy buffer process) is updated at the end of the copy. The copy driver is responsible for unlocking the Graphics Engine by calling the function `LLUI_DISPLAY_setBackBuffer()` at the end of the copy.

There are two use cases:

Hardware

The copy buffer process is performed in hardware (DMA). In that case, the DMA driver must configure an interrupt to be notified about the end of the copy.

```
static uint8_t _flush_identifier;

void LLUI_DISPLAY_IMPL_initialize(LLUI_DISPLAY_SInitData *init_data) {
  // [...]

  // use two distinct buffers between the LCD driver and the Graphics Engine
  LCD_DRIVER_initialize(frame_buffer);
  init_data->back_buffer_address = back_buffer;

  // initialize the DMA driver: GPIO, etc.
  DMA_DRIVER_initialize();
}

void LLUI_DISPLAY_IMPL_flush(MICROUI_
↪GraphicsContext *gc, uint8_t flush_identifier, const ui_rect_t areas[], size_t length) {
  // store the identifier of the flush used to unlock the Graphics Engine later
  _flush_identifier = flush_identifier;

  // configure the DMA to copy n bytes
```

(continues on next page)

(continued from previous page)

```
  // back_buffer == LLUI_DISPLAY_getBufferAddress(&gc->image)
  DMA_DRIVER_prepare_sent(frame_
↪buffer, back_buffer, LCD_WIDTH * LCD_HEIGHT * LCD_BPP / 8); // dest / src / size

  // configure the "end of copy" interrupt
  DMA_DRIVER_enable_interrupt(END_OF_COPY);

  // start the copy
  DMA_DRIVER_start();
}

void DMA_IRQHandler(void) {
  DMA_DRIVER_clear_interrupt();
  DMA_DRIVER_disable_interrupt(END_OF_COPY);

  // end of copy, unlock the Graphics Engine without changing the back buffer address
  LLUI_DISPLAY_
↪setBackBuffer(_flush_identifier, back_buffer, true); // true: called under interrupt
}
```

Software

The copy buffer process cannot be performed in hardware or requires a software loop to copy all data (DMA linked list). This copy buffer process must not be performed in the `flush()` function. A dedicated OS task is required to perform this copy.

```
static void *_copy_task_semaphore;
static uint8_t _flush_identifier;

static void _task_flush(void *p_arg) {
  while(1) {
    int32_t size = LCD_WIDTH * LCD_HEIGHT * LCD_BPP / 8;
    uint8_t *dest = frame_buffer;
    uint8_t *src = back_buffer;

    // wait until the Graphics Engine gives the order to copy
    LLUI_DISPLAY_IMPL_binarySemaphoreTake(_copy_task_semaphore);

    // copy data
    while(size) {
      int32_t s = min(DMA_MAX_SIZE, size);
      DMA_DRIVER_copy_data(dest, src, s); // dest / src / size
      dest += s;
      src += s;
      size -= s;
    }

    // end of copy, unlock the Graphics Engine without changing the back buffer address
    LLUI_DISPLAY_
↪setBackBuffer(_flush_identifier, back_buffer, false); // false: called outside interrupt
  }
}

void LLUI_DISPLAY_IMPL_initialize(LLUI_DISPLAY_SInitData *init_data) {
  // [...]

  // use two distinct buffers between the LCD driver and the Graphics Engine
```

(continues on next page)

(continued from previous page)

```
  LCD_DRIVER_initialize(frame_buffer);
  init_data->back_buffer_address = back_buffer;

  // create a "flush" task and a dedicated semaphore
  _copy_task_semaphore = (void*)xSemaphoreCreateBinary();
  xTaskCreate(_task_flush, "FlushTask", 1024, NULL, 12, NULL);
}

void LLUI_DISPLAY_IMPL_flush(MICROUI_
↪GraphicsContext *gc, uint8_t flush_identifier, const ui_rect_t areas[], size_t length) {
  // store the identifier of the flush used to unlock the Graphics Engine later
  _flush_identifier = flush_identifier;

  // unlock the copy task
  LLUI_DISPLAY_IMPL_binarySemaphoreGive(_copy_task_semaphore, false);
}
```

Single Buffer (parallel) and Tearing Enabled

This buffer policy is the same as the previous chapter, but it uses the LCD tearing signal to synchronize the LCD refresh rate with the copy buffer process. The copy buffer process should not start during the call of `flush()` but should wait for the next tearing signal to start the copy.

There are two use cases:

Hardware

```
static uint8_t _start_DMA;
static uint8_t _flush_identifier;

void LLUI_DISPLAY_IMPL_initialize(LLUI_DISPLAY_SInitData *init_data) {
  // [...]

  // use two distinct buffers between the LCD driver and the Graphics Engine
  LCD_DRIVER_initialize(frame_buffer);
  init_data->back_buffer_address = back_buffer;

  // enable the tearing interrupt
  _start_DMA = 0;
  TE_enable_interrupt();

  // initialize the DMA driver: GPIO, etc.
  DMA_DRIVER_initialize();
}

void LLUI_DISPLAY_IMPL_flush(MICROUI_
↪GraphicsContext *gc, uint8_t flush_identifier, const ui_rect_t areas[], size_t length) {
  // store the identifier of the flush used to unlock the Graphics Engine later
  _flush_identifier = flush_identifier;

  // configure the DMA to copy n bytes
  // back_buffer == LLUI_DISPLAY_getBufferAddress(&gc->image)
  DMA_DRIVER_prepare_sent(frame_
↪buffer, back_buffer, LCD_WIDTH * LCD_HEIGHT * LCD_BPP / 8); // dest / src / size
```

(continues on next page)

(continued from previous page)

```
  // configure the "end of copy" interrupt
  DMA_DRIVER_enable_interrupt(END_OF_COPY);

  // unlock the job of the tearing interrupt
  _start_DMA = 1;
}

void TE_IRQHandler(void) {
  TE_clear_interrupt();

  if (_start_DMA) {
    _start_DMA = 0;

    // start the copy
    DMA_DRIVER_start();
  }
}

void DMA_IRQHandler(void) {
  DMA_DRIVER_clear_interrupt();
  DMA_DRIVER_disable_interrupt(END_OF_COPY);

  // end of copy, unlock the Graphics Engine without changing the back buffer address
  LLUI_DISPLAY_
↪setBackBuffer(_flush_identifier, back_buffer, true); // true: called under interrupt
}
```

Software

```
static void *_copy_task_semaphore;
static uint8_t _start_copy;
static uint8_t _flush_identifier;

static void _task_flush(void *p_arg) {
  while(1) {
    // wait until the Graphics Engine gives the order to copy
    LLUI_DISPLAY_IMPL_binarySemaphoreTake(_copy_task_semaphore);

    int32_t size = LCD_WIDTH * LCD_HEIGHT * LCD_BPP / 8;
    uint8_t *dest = frame_buffer;
    uint8_t *src = back_buffer;

    // copy data
    while(size) {
      int32_t s = min(DMA_MAX_SIZE, size);
      DMA_DRIVER_copy_data(dest, src, s); // dest / src / size
      dest += s;
      src += s;
      size -= s;
    }

    // end of copy, unlock the Graphics Engine without changing the back buffer address
    LLUI_DISPLAY_
↪setBackBuffer(_flush_identifier, back_buffer, false); // false: called outside interrupt
  }
}
```

(continues on next page)

(continued from previous page)

```
void LLUI_DISPLAY_IMPL_initialize(LLUI_DISPLAY_SInitData *init_data) {
  // [...]

  // use two distinct buffers between the LCD driver and the Graphics Engine
  LCD_DRIVER_initialize(frame_buffer);
  init_data->back_buffer_address = back_buffer;

  // create a "flush" task and a dedicated semaphore
  _copy_task_semaphore = (void*)xSemaphoreCreateBinary();
  xTaskCreate(_task_flush, "FlushTask", 1024, NULL, 12, NULL);

  // enable the tearing interrupt
  _start_copy = 0;
  TE_enable_interrupt();
}

void LLUI_DISPLAY_IMPL_flush(MICROUI_
↪GraphicsContext *gc, uint8_t flush_identifier, const ui_rect_t areas[], size_t length) {
  // store the identifier of the flush used to unlock the Graphics Engine later
  _flush_identifier = flush_identifier;

  // unlock the job of the tearing interrupt
  _start_copy = 1;
}

void TE_IRQHandler(void) {
  TE_clear_interrupt();

  if (_start_copy) {
    _start_copy = 0;

    // unlock the copy task
    LLUI_DISPLAY_IMPL_binarySemaphoreGive(_copy_task_semaphore, true);
  }
}
```

Transmit and Swap Buffer

This buffer policy is a mix between the buffer policies *Single* and *Swap Double*. It requires two back buffers: the application uses a buffer to perform its drawings and the second buffer is used to transmit the data to the display frame buffer when the Graphics Engine is calling the `flush()` function. At the end of the transmission, the application buffer becomes the transmission buffer and vice-versa.

> The subtlety consists in reusing the transmission buffer as the application buffer at the end of the transmission if, and only if, the application has not drawn anything yet in the current application buffer. This avoids handling the restoration of the past: the application reuses the same buffer as before the last flush.
>
> This policy requires a dedicated OS task that will manage the transmission and the unlocking of the Graphics Engine by calling the function `LLUI_DISPLAY_setBackBuffer()`. The specification of the `flush()` function is to be **non-blocking**. Its aim is to unlock the *flush* task. The `flush()` function has to return as soon as possible.
>
> As soon as a transmission is started, the second buffer is freed. The Graphics Engine does not need to wait for the end of the serial data transmission: the application can draw immediately

in the new back buffer. Note that the second flush has to wait the end of the first flush (the end of the transmission) before configuring and launching a new transmission.

The serial data transmission is performed in hardware or in software. In hardware, the serial driver must configure an interrupt to be notified about the end of the transmission. In software, the *transmission* step is synchronous and blocking.

Note: This pseudo implementation considers a display with a *serial* connection but the reasoning is similar with a *parallel* connection.

```
static uint8_t _flush_identifier;
static uint8_t _buffer_index;
static void *_transmit_task_semaphore;
static uint8_t _flush_identifier;

static void _task_flush(void *p_arg) {
  while(1) {
    // wait until the Graphics Engine gives the order to flush
    LLUI_DISPLAY_IMPL_binarySemaphoreTake(_transmit_task_semaphore);

    // save the flush configuration: can be modified by the next␣
↪call to flush() as soon as LLUI_DISPLAY_setBackBuffer() wakes up the Graphics Engine
    uint8_t flush_identifier = _flush_identifier;

    // retrieve the transmit buffer: the current back buffer
    uint8_t *transmit_buffer = back_buffers[_buffer_index];

    // swap both buffers (_buffer_index now points to the new back buffer)
    _buffer_index = (_buffer_index + 1) & ~1;

    // the new back buffer can be used for the next drawings
    if (LLUI_DISPLAY_setBackBuffer(flush_identifier, back_buffers[_buffer_index], false)) {

  ␣
↪   // here: the Graphics Engine is unlocked, the application can draw in the new back buffer
      // **and** can call flush() again


  ␣
↪   // configure and start the serial device to transmit n bytes (synchronous or asynchronous
      // transmission)
      SERIAL_DRIVER_
↪transmit_data(transmit_buffer, LCD_WIDTH * LCD_HEIGHT * LCD_BPP / 8);

      // wait for the end of the transmission (blocking call or use an interrupt)
      SERIAL_DRIVER_transmit_wait();

      // here: the back buffer has been sent to the LCD, the buffer can be used again for the
      // next drawings only if no new drawing has been already performed in the current back
      // buffer

      // reuse the old back buffer if no drawing has been already performed
      if (LLUI_DISPLAY_setBackBuffer(flush_identifier, transmit_buffer, false)) {
        // the new back buffer is set: cancel the previous swap to synchronize the driver with
        // the Graphics Engine
        _buffer_index = (_buffer_index + 1) & ~1;
      }
```

(continues on next page)

(continued from previous page)

```
      // else: too late to reuse this old transmission back buffer; nothing to do
    }
    else {
      // unexpected end of flush; the Graphics Engine keeps using the previous back buffer;
      // we have to cancel the buffer swap
      _buffer_index = (_buffer_index + 1) & ~1;
    }
  }
}


void LLUI_DISPLAY_IMPL_initialize(LLUI_DISPLAY_SInitData *init_data) {
  // [...]

  LCD_DRIVER_initialize();

  // buffer 0 is the first back buffer; buffer 1 is not used
  _buffer_index = 0;
  init_data->back_buffer_address = back_buffers[_buffer_index];

  // initialize the serial driver & device: GPIO, etc.
  SERIAL_DRIVER_initialize();

  // create a "flush" task and a dedicated semaphore
  _transmit_task_semaphore = (void*)xSemaphoreCreateBinary();
  xTaskCreate(_task_flush, "FlushTask", 1024, NULL, 12, NULL);
}

void LLUI_DISPLAY_IMPL_flush(MICROUI_
↪GraphicsContext *gc, uint8_t flush_identifier, const ui_rect_t areas[], size_t length) {
  // store the flush identifier to unlock the Graphics Engine later
  _flush_identifier = flush_identifier;

  // unlock the flush task
  LLUI_DISPLAY_IMPL_binarySemaphoreGive(_transmit_task_semaphore, false);
}
```

Dependencies

- MicroUI module (see *MicroUI*)
- LLUI_DISPLAY_impl.h implementation if standard or custom implementation is chosen (see *Dependencies* and *LLUI_DISPLAY: Display*).
- The *MicroUI C module*.

Installation

The Display module is a sub-part of the MicroUI library.

SDK 6

SDK 5

When the MicroUI Pack module is installed, the Display module is automatically enabled.

When the MicroUI module is installed, the Display module must be installed in order to connect the physical display with the VEE Port. If not installed, the *stub* module will be used.

In the VEE Port configuration file, check UI > Display to install the Display module.

When enabled, the Display module must be configured. This configuration step is used to choose the kind of implementation (see *Dependencies*).

SDK 6

SDK 5

In SDK 6, the configuration is done in the properties file configuration.properties of the VEE Port project. All the properties names listed below must be prefixed by com.microej.pack.display. . For example the bpp properties is defined by the com.microej.pack.display.bpp property.

In SDK 5, the configuration is done in the properties file display/display.properties .

The properties file must / can contain the following properties:

- bpp [mandatory]: Defines the number of bits per pixel the display device is using to render a pixel. The expected value is one among these lists:

  Standard formats:

  - ARGB8888 : Alpha 8 bits; Red 8 bits; Green 8 bits; Blue 8 bits,
  - RGB888 : Alpha 0 bit; Red 8 bits; Green 8 bits; Blue 8 bits (fully opaque),
  - RGB565 : Alpha 0 bit; Red 5 bits; Green 6 bits; Blue 5 bits (fully opaque),
  - ARGB1555 : Alpha 1 bit; Red 5 bits; Green 5 bits; Blue 5 bits (fully opaque or fully transparent),
  - ARGB4444 : Alpha 4 bits; Red 4 bits; Green 4 bits; Blue 4 bits,
  - C4 : 4 bits to encode linear grayscale colors between 0xff000000 and 0xffffffff (fully opaque),
  - C2 : 2 bits to encode linear grayscale colors between 0xff000000 and 0xffffffff (fully opaque),
  - C1 : 1 bit to encode grayscale colors 0xff000000 and 0xffffffff (fully opaque).

  Custom formats:

  - 32 : up to 32 bits to encode Alpha, Red, Green, and Blue (in any custom arrangement),
  - 24 : up to 24 bits to encode Alpha, Red, Green, and Blue (in any custom arrangement),
  - 16 : up to 16 bits to encode Alpha, Red, Green, and Blue (in any custom arrangement),
  - 8 : up to 8 bits to encode Alpha, Red, Green, and Blue (in any custom arrangement),
  - 4 : up to 4 bits to encode Alpha, Red, Green, and Blue (in any custom arrangement),
  - 2 : up to 2 bits to encode Alpha, Red, Green, and Blue (in any custom arrangement),
  - 1 : 1 bit to encode Alpha, Red, Green, or Blue.

  All other values are forbidden (throw a generation error).

- byteLayout [optional, the default value is "line"]: Defines the pixels data order in a byte the display device is using. A byte can contain several pixels when the number of bits per pixel (see 'bpp' property) is lower than 8. Otherwise, this property is useless. Two modes are available: the next bit(s) on the same byte can target the next pixel on the same line or the same column. In the first case, when the end of the line is reached, the next byte contains the first pixels of the next line. In the second case, when the end of the column is reached, the next byte contains the first pixels of the next column. In both cases, a new line or a new column restarts with a new byte, even if some free bits remain in the previous byte.
  - line : the next bit(s) on current byte contains the next pixel on same line (x increment),
  - column : the next bit(s) on current byte contains the next pixel on the same column (y increment).

  Note:

  - Default value is 'line'.
  - All other modes are forbidden (throw a generation error).
  - When the number of bits-per-pixels (see 'bpp' property) is higher or equal to 8, this property is useless and ignored.

- memoryLayout [optional, the default value is "line"]: Defines the pixels data order in memory the display device is using. This option concerns only the display with a bpp lower than 8 (see 'bpp' property). Two modes are available: when the byte memory address is incremented, the next targeted group of pixels is the next group on the same line or the next group on the same column. In the first case, when the end of the line is reached, the next group of pixels is the first group of the next line. In the second case, when the end of the column is reached, the next group of pixels is the first group of the next column.
  - line : the next memory address targets the next group of pixels on same line (x increment),
  - column : the next memory address targets the next group of pixels on the same column (y increment).

  Note:

  - Default value is 'line'.
  - All other modes are forbidden (throw a generation error).
  - When the number of bits-per-pixels (see 'bpp' property) is higher or equal to 8, this property is useless and ignored.

- imageBuffer.memoryAlignment [optional, default value is "4"]: Defines the image memory alignment to respect when creating an image. This notion is useful when image drawings are performed by a third-party hardware accelerator (GPU): it can require some constraints on the image to draw. This value is used by the Graphics Engine when creating a dynamic image and by the image generator to encode a RAW image. See *GPU Format Support* and *Customize MicroEJ Standard Format*. Allowed values are 1, 2, 4, 8, 16, 32, 64, 128 and 256.
- imageHeap.size [optional, the default value is "not set"]: Defines the image heap size. It is useful to fix a VEE Port heap size when building firmware in the command line. When using a MicroEJ launcher, the size set in this launcher has priority over the VEE Port value.

Use

The MicroUI Display APIs are available in the class ej.microui.display.Display.

Buffer Refresh Strategy

Overview

The Buffer Refresh Strategy (BRS) ensures that the front buffer contains all the drawings before letting the display driver flush this buffer into the display panel. The drawings are the drawings made since the last *flush* **and** the *past*. The *past* symbolizes the drawings made before the last *flush* and that has not been altered by the new drawings.

Table 69: Automatic Refresh

Drawing Steps	Back Buffer	Front Buffer
Startup		
Draw “background”		
Draw “A”		
Flush (swap)		
Draw “B”		
Refresh the past		
Flush (swap)		

This refreshing avoids running again all drawing algorithms (and layout) to fill the back buffer (here: the entire background, the “A” green background, and the “A”). Without this refreshing, the display will show the incomplete frame *Draw “B”*:

Table 70: Missing Refresh

Drawing Steps	Back Buffer	Front Buffer
Startup		
Draw “background”		
Draw “A”		
Flush (swap)		
Draw “B”		
Flush (swap)		

When the new drawings overlap the *past*, it is useless to refresh the past:

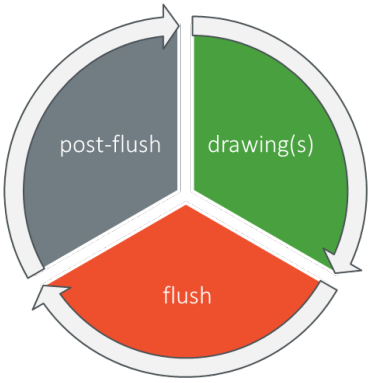
Table 71: Useless Refresh

Drawing Steps	Back Buffer	Front Buffer
Draw “C”		
Flush (swap)		
Draw “D”		
Flush (swap)		

Timeline

Basic Principle

This illustration symbolizes the basic principle of the Graphics Engine’s timeline:

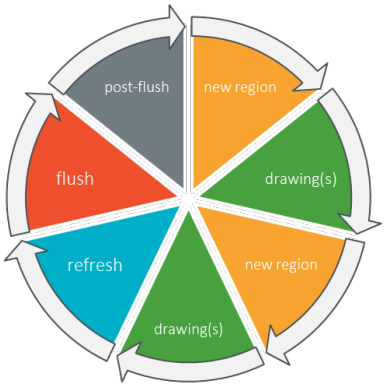


- *drawing(s)* symbolizes one or several drawings in the back buffer.
- *flush* symbolizes the call to the LLAPI `LLUI_DISPLAY_IMPL_flush()` that allows the display driver to update the display panel content according to the *display connection* (serial or parallel).
- *post-flush* symbolizes the moment between the end of flush (end of swap, end of transmission, or end of copy) and the unlocking of the Graphics Engine (the call to `LLUI_DISPLAY_setBackBuffer()`). Before this call, the Graphics Engine is not allowed to draw in the buffer.

Note: The time between the *post-flush* and *drawing(s)* depends on the application: the first drawing after a *flush* can occur immediately after the *post-flush* or later.

Additional Hooks

The Graphics Engine provides some hooks (through dedicated LLAPI) to be notified of further details:



- *new region* symbolizes that the following drawing(s) will be drawn in a region other than the previous drawings.
- *refresh* symbolizes that the last drawing has been done, and a call to LLUI_DISPLAY_IMPL_flush() will be performed just after.

  During these two new steps, the implementation can render into the back buffer (to restore the past), prepare the next flush (store the regions to flush), etc.

Implicit Region

A *region* is considered a *new implicit region* as soon as the MicroUI clip is updated **and** a drawing is performed. When a clip is considered an *implicit* region, a call to the LLAPI LLUI_DISPLAY_IMPL_newDrawingRegion(...) is performed. The following sequence illustrates when the LLAPI is called:

	Application Calls	LLAPI
1	gc.setClip(...)	
2[1]	Painter.drawXX(...)	LLUI_DISPLAY_IMPL_newDrawingRegion(..., true) LLUI_PAINTER_IMPL_drawXX(...)
3[2]	Painter.drawYY(...)	LLUI_PAINTER_IMPL_drawYY(...)
4[3]	gc.setClip(...)	
5	gc.setClip(...)	
6	Painter.drawZZ(...)	LLUI_DISPLAY_IMPL_newDrawingRegion(..., true) LLUI_PAINTER_IMPL_drawZZ(...)

Note: The very first drawing's region after a *flush* is systematically considered as *implicit*.

Explicit Region

The application can *explicitly* call the LLAPI LLUI_DISPLAY_IMPL_newDrawingRegion(...) by calling the API GraphicsContext.notifyDrawingRegion() . The LLAPI parameters are:

- the region is the current MicroUI clip,
- the argument drawing_now is valued to false : this means no drawing will follow this call (*explicit* region).

Declaring explicit regions is mainly useful when it is performed before the very first drawing. It indicates to the BRS that several regions will be altered before the next flush. These regions don't need to be restored with the past (their content will change).

[1] The LLAPI argument drawing_now is valued to true : this means a call to a drawing action will be called just after (implicit region).
[2] The second drawing uses the same region as the first one: the region is not notified again.
[3] The clip is not recognized as an implicit region because no drawing is performed just after.

	Application Calls	LLAPI
1	gc.setClip(...)	
2[4]	gc.notifyDrawingRegion(...)	LLUI_DISPLAY_IMPL_newDrawingRegion(.., false)
3[5]	Painter.drawXX(...)	LLUI_DISPLAY_IMPL_newDrawingRegion(.., true) LLUI_PAINTER_IMPL_drawXX(...)
4	Painter.drawYY(...)	LLUI_PAINTER_IMPL_drawYY(...)
5[6]	gc.notifyDrawingRegion(...)	LLUI_DISPLAY_IMPL_newDrawingRegion(.., false)
6[7]	Painter.drawZZ(...)	LLUI_PAINTER_IMPL_drawZZ(...)

Flush vs Refresh

The Graphics Engine does not store the regions (implicit or explicit). The BRS is responsible for implementing the LLAPI (the hooks, see above) and managing these regions.

When the application calls Display.flush() , the Graphics Engine immediately calls the LLAPI LLUI_DISPLAY_IMPL_refresh() . This call allows the BRS:

- to finalize (if required) the back buffer (no drawing will be performed into the buffer until the next call to LLUI_DISPLAY_setBackBuffer()),
- **and** to call the LCD driver flush function LLUI_DISPLAY_IMPL_flush() by giving the region(s) to update on the display panel.

Strategies

Several strategies are available according to different considerations:

- the *display connection* (serial or parallel),
- the *buffer policy* (direct, single, swap),
- if the *past* has to be restored,
- if the *past* is systematically restored,
- when the *past* is restored,
- etc.

The following chapters describe the strategies:

[4] The LLAPI is immediately called.
[5] The step 2 doesnt change the flow of the implicit region: a call to LLUI_DISPLAY_IMPL_newDrawingRegion(..., true) is always performed even if a call to LLUI_DISPLAY_IMPL_newDrawingRegion(..., false) is performed just before.
[6] The clip has not changed, but the LLAPI is explicitly called again.
[7] The clip has not changed, so the implicit region is not notified.

- For the single buffer policy, the restoration is useless; the recommended strategy is *Strategy: Single*.
- For the multiple buffers policy, the recommended strategy is *Strategy: Predraw*.
- The strategies *Strategy: Default*, *Strategy: Custom* and *Strategy: Legacy* can be used for other use-cases.

Strategy: Single

Principle

This strategy considers that the drawings are always performed in the same back buffer (*single* buffer policy). In this case, the restoration is useless because the back buffer always contains the past.

Serial Connection

Parallel Connection

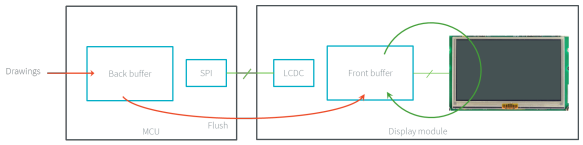


Fig. 225: Single Buffer (serial)

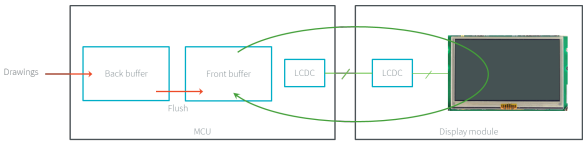


Fig. 226: Single Buffer (parallel)

The principle of this strategy is to cumulate the drawing regions. The refresh consists in transmitting these regions (a list of rectangles) that have been modified since the last flush (or a unique rectangle that encapsulates all the regions) to the LCD driver through the LLAPI LLUI_DISPLAY_IMPL_flush() .

The implicit and explicit regions have the same meaning: a dirty region to flush to the front buffer.

Behavior

The following table illustrates how the strategy works:

Table 72: Strategy “Single”

Drawing Steps	Strategy Work	Back Buffer	Front Buffer
Startup			
Implicit region *background*	Store the region *full-screen*		
Draw “background”			
Implicit region *A*	The region *A* is included in the region *full-screen*: nothing to do		
Draw “A”			
Refresh	Call LLUI_DISPLAY_IMPL_flush() (flush the region *full-screen*) Clear the list of regions		
Implicit region *B*	Store the region *B*		
Draw “B”			
Implicit region *C*	Store the region *C*		
Draw “C”			
Refresh	Call LLUI_DISPLAY_IMPL_flush() (flush the regions *B* and *C*) Clear the list of regions		

Note: This illustration considers that the clip changes before each drawing and fits the drawing’s bounds

Use

Here are the steps around the strategy describing how to use it:

1. Some drawings are performed in the back buffer.
2. A Display.flush() is asked, the Graphics Engine calls LLUI_DISPLAY_IMPL_refresh() .
3. The strategy calls LLUI_DISPLAY_IMPL_flush() .
4. The display driver has to implement LLUI_DISPLAY_IMPL_flush() , which consists in transmitting the back buffer data to the front buffer.
5. As soon as the transmission is performed, the BSP has to notify the Graphics Engine by calling LLUI_DISPLAY_setBackBuffer() , giving the same back buffer address (there is only one buffer).
6. The Graphics Engine is now unlocked, and a new drawing can start in the back buffer.

Strategy: Predraw

Principle

This strategy considers that the drawings are always performed in a buffer, and a swap with another buffer is made by the implementation of LLUI_DISPLAY_IMPL_flush() . In this case, the restoration is mandatory because the new back buffer must contain the past before the buffer swapping.

The principle of this strategy is to cumulate the drawing regions and restore them just before the very first drawing after a flush. The refresh consists in calling the LLAPI LLUI_DISPLAY_IMPL_flush() that will swap the buffers.

Some regions to restore are updated or removed according to the implicit and explicit regions *given* **before** the very first drawing after a flush. These regions are the regions that the application will alter, so it is useless to restore them. For instance, if the very first drawing after a flush fully fills the buffer (erase the buffer), the past is not restored.

The implicit and explicit regions **after** the very first drawing have the same signification: a dirty region to restore before the very first drawing after the next flush.

Behavior

The following table illustrates how the strategy works:

Table 73: Strategy “Predraw”

Drawing Steps	Strategy Work	Back Buffer	Front Buffer
Startup			
Implicit region *background*	Store the region *full-screen*		
Draw “background”			
Implicit region *A*	The region *A* is included in the region *full-screen*: nothing to do		
Draw “A”			
Refresh	Call LLUI_DISPLAY_IMPL_flush() (swap the buffers)		
Implicit region *B*	Restore the region *full-screen* expect the region *B* Clear the list of regions Store the region *B*		
Draw “B”			
Refresh	Call LLUI_DISPLAY_IMPL_flush() (swap the buffers)		
Implicit region *C*	Nothing to restore because the region *B* equals the region *C*		
Draw “C”			
Refresh	Call LLUI_DISPLAY_IMPL_flush() (swap the buffers)		

Note: This illustration considers that the clip changes before each drawing and fits the drawing’s bounds

Read the Display

Before the very first drawing after a flush, the content of the back buffer does not contain the past (the restoration has not been performed). As a consequence, the first read actions (GraphicsContext.readPixel() , Painter.drawDisplayRegion() , etc.) cannot use the back buffer as the source buffer. The algorithm has to call LLUI_DISPLAY_getSourceImage() to retrieve a pointer to the front buffer address.

Use (Swap Double Buffer)

Here are the steps around the strategy describing how to use it in *double* buffer policy.

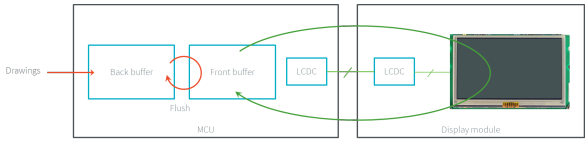


Fig. 227: Swap Double Buffer

The two buffers have the same role alternatively, back buffer and front buffer:

1. Some drawings are performed in the back buffer.
2. A `Display.flush()` is asked, the Graphics Engine calls `LLUI_DISPLAY_IMPL_refresh()`.
3. The strategy calls `LLUI_DISPLAY_IMPL_flush()`.
4. The display driver has to implement `LLUI_DISPLAY_IMPL_flush()` that consists in swapping the back and front buffers.
5. As soon as the display *uses* the new front buffer (the new back buffer is now freed), the BSP has to notify the Graphics Engine by calling `LLUI_DISPLAY_setBackBuffer()`, giving the new back buffer address (== previous front buffer).
6. The Graphics Engine is now unlocked.
7. Before the very first drawing, this strategy copies the regions to restore from the previous back buffer to the new back buffer.
8. A new drawing can start in the new back buffer.

Use (Swap Triple Buffer)

Here are the steps around the strategy describing how to use it in *triple* buffer policy.

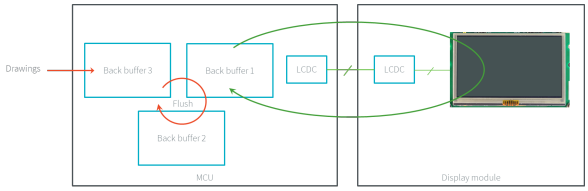


Fig. 228: Swap Triple Buffer

The three buffers have the same role alternatively: back buffers (A and B) and front buffer (C). On startup, the front buffer is mapped on buffer (C), buffer (A) is the back buffer, and the buffer (B) is not used yet:

- buffer (A): the application's back buffer
- buffer (B): free
- buffer (C): LCD driver's buffer

1. Some drawings are performed in the back buffer (A).

2. A Display.flush() is asked, the Graphics Engine calls LLUI_DISPLAY_IMPL_refresh() .
3. The strategy calls LLUI_DISPLAY_IMPL_flush() .
4. The display driver has to implement LLUI_DISPLAY_IMPL_flush() that consists in swapping the buffers: the new LCD refresh task will read the data from buffer (A), and the next drawings will be done in buffer (B), but the buffer (C) is still in use (the LCD driver keeps using this buffer to refresh the LCD).

- buffer (A): next LCD driver's buffer
- buffer (B): new the application's back buffer
- buffer (C): current LCD driver's buffer

5. The buffer (B) is immediately available (free): the BSP has to notify the Graphics Engine by calling LLUI_DISPLAY_setBackBuffer() , giving the buffer (B)'s address.
6. The Graphics Engine is now unlocked.
7. Before the very first drawing, this strategy copies the regions to restore from the previous back buffer (A) to the new back buffer (B).
8. Some drawings are performed in the back buffer (B).
9. A second Display.flush() is asked, the Graphics Engine calls LLUI_DISPLAY_IMPL_refresh() .
10. The strategy calls LLUI_DISPLAY_IMPL_flush() .
11. The system is locked: the LCD driver does not use the buffer (A) as the source buffer yet.
12. As soon as the LCD driver *uses* the buffer (A) (the LCD driver keeps using this buffer to refresh the LCD), the buffer (C) becomes available (free).

- buffer (A): current LCD driver's buffer
- buffer (B): application's back buffer
- buffer (C): free

13. The buffer (C) will now be used for the next drawings. Go to step 5.

Use (Transmit and Swap Buffer)

Here are the steps around the strategy describing how to use it in *transmit and swap* buffer policy.

Serial Connection

Parallel Connection

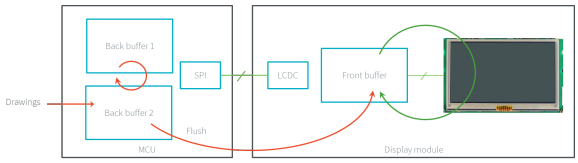


Fig. 229: Transmit and Swap (serial)

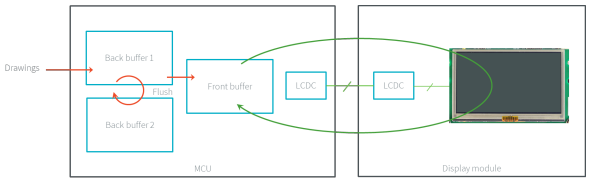


Fig. 230: Transmit and Swap (parallel)

The two buffers have the same role alternatively: back buffer and transmission buffer. On startup, the transmission buffer has yet to be used.

In this policy, the implementation of `LLUI_DISPLAY_IMPL_flush()` consists in swapping the back buffers and transmitting the content of the back buffer to the front buffer (SPI, DSI, etc.). This subtlety allows the reuse of the same back buffer after the end of the transmission: this prevents the restoration of the past.

1. Some drawings are performed in the back buffer.
2. A `Display.flush()` is asked, the Graphics Engine calls `LLUI_DISPLAY_IMPL_refresh()`.
3. The strategy calls `LLUI_DISPLAY_IMPL_flush()`.
4. The display driver has to implement `LLUI_DISPLAY_IMPL_flush()` which consists in starting the transmission of the back buffer content to the LCD device's buffer and swapping both buffers (back and transmission buffers).
5. The new back buffer is immediately available (free); the BSP has to notify the Graphics Engine by calling `LLUI_DISPLAY_setBackBuffer()`, giving the new back buffer address (== previous transmission buffer).
6. The Graphics Engine is now unlocked.
7. Before the very first drawing, this strategy copies the regions to restore from the previous back buffer to the new back buffer.
8. Some drawings are performed in the back buffer.
9. A second `Display.flush()` is asked, the Graphics Engine calls `LLUI_DISPLAY_IMPL_refresh()`.
10. The strategy calls `LLUI_DISPLAY_IMPL_flush()`.
11. The system is locked: the LCD driver still needs to finish transmitting the transmission buffer data to the LCD device's buffer.
12. As soon as the transmission is done, the BSP has to notify the Graphics Engine by calling `LLUI_DISPLAY_setBackBuffer()`, giving the new back buffer address (== previous transmission buffer).
13. The application is sleeping (doesn't want to draw in the back buffer)

Hint: Optimization: As soon as the transmission to the LCD device's buffer is done, the BSP should call again `LLUI_DISPLAY_setBackBuffer()` by giving the transmission buffer (which is now free). If the drawing has yet to start in the back buffer, the Graphics Engine will reuse this transmission buffer as a new back buffer instead of using the other one; the restoration becomes useless.

---

14. The BSP should notify the Graphics Engine again by calling LLUI_DISPLAY_setBackBuffer() , giving the transmission buffer address: the Graphics Engine will reuse this buffer for future drawings, and the strategy will not need to restore anything.

Strategy: Default

Principle

This strategy is the default strategy used when no explicit strategy is selected. This strategy is implemented in the Graphics Engine, and its behavior is minimalist. However, this strategy can be used for the *direct* buffer policy.

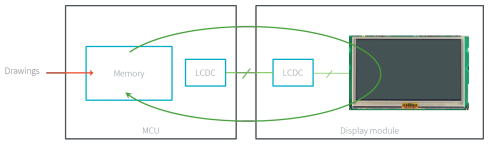


Fig. 231: Direct Buffer

This strategy considers that the drawings are always performed in the same back buffer. In this case, the restoration is useless because the buffer always contains the past. Furthermore, as the LCD driver uses the same buffer to refresh the display panel, this strategy has nothing to do.

Behavior

The following table illustrates how the strategy works:

Table 74: Strategy “Direct”

Drawing Steps	Strategy Work	Front Buffer
Startup		
Implicit region *background*		
Draw “background”		
Implicit region *A*		
Draw “A”		
Refresh	Call LLUI_DISPLAY_IMPL_flush() (nothing to do)	

Note: This illustration considers that the clip changes before each drawing and fits the drawing’s bounds

Use

Here are the steps around the strategy describing how to use it:

1. Some drawings are performed in the buffer.
2. A Display.flush() is asked, the Graphics Engine calls LLUI_DISPLAY_IMPL_refresh() .
3. The strategy calls LLUI_DISPLAY_IMPL_flush() .
4. The display driver has to implement LLUI_DISPLAY_IMPL_flush() : at least enable the LCD refresh interrupt to wait until the end of the refresh (or use a software task).
5. In the LCD refresh interrupt (here, the display panel shows the latest frame for sure), the BSP has to notify the Graphics Engine by calling LLUI_DISPLAY_setBackBuffer() , giving the same buffer address.
6. The Graphics Engine is now unlocked.
7. Some drawings are performed in the back buffer.

Strategy: Custom

Principle

This strategy symbolizes the strategy implemented by the BSP (the other strategies are implemented in the *MicroUI C Module* or in the Graphics Engine). This strategy is useful to map a specific behavior according to a specific application, the number of buffers, how the display panel is mapped, etc.

The BSP has the responsibility to implement the following functions (in addition to LLUI_DISPLAY_IMPL_flush()):

- LLUI_DISPLAY_IMPL_newDrawingRegion()
- LLUI_DISPLAY_IMPL_refresh()

> **Warning:** Both functions are already implemented as weak functions in the Graphics Engine (see *Strategy: Default*)

Behavior

The following table illustrates how the strategy works:

Table 75: Strategy “Custom”

Drawing Steps	Strategy Work	Back Buffer
Startup		
Implicit region *background*	Implement LLUI_DISPLAY_IMPL_newDrawingRegion()	
Draw “background”		
Implicit region *A*	Implement LLUI_DISPLAY_IMPL_newDrawingRegion()	
Draw “A”		
Refresh	Implement LLUI_DISPLAY_IMPL_refresh()	

Note: This illustration considers that the clip changes before each drawing and fits the drawing’s bounds

Use

Here are the steps around the strategy describing how to use it:

1. Some drawings are performed in the buffer.
2. A Display.flush() is asked, the Graphics Engine calls LLUI_DISPLAY_IMPL_refresh() .
3. The strategy has to implement LLUI_DISPLAY_IMPL_refresh() and call LLUI_DISPLAY_IMPL_flush() .
4. The display driver has to implement LLUI_DISPLAY_IMPL_flush() .
5. When the display panel shows the latest frame, the BSP has to notify the Graphics Engine by calling LLUI_DISPLAY_setBackBuffer() , giving the buffer address.
6. The Graphics Engine is now unlocked.
7. Some drawings are performed in the buffer.

Strategy: Legacy

Principle

This strategy mimics the behavior of the specification of the UI Pack 13.x, dedicated to the *multi-buffers* policies.

The specification consisted in:

1. swapping the back buffer and the front buffer at flush time,
2. letting the BSP restore itself to the back buffer with the content of the previous drawings (the past) before unlocking the Graphics Engine after a flush.

   As a consequence, the past was always available before making the very first drawing after a flush.

The strategy *Legacy* is useful to keep the behavior of the VEE Ports made for UI Pack 13.x without updating them (except the signature of the LLAPI `LLUI_DISPLAY_IMPL_flush()`). This strategy merges all drawing regions into only one rectangle (that includes all drawing regions). This single rectangle is given to the function `LLUI_DISPLAY_IMPL_flush()`.

Note: For the *single buffer policy*, it is recommended to migrate to the *strategy single*.

Behavior

The following table illustrates how the strategy works:

Table 76: Strategy "Legacy"

Drawing Steps	Strategy Work	Back Buffer	Front Buffer
Startup			
Implicit region *background*	Store the region *full-screen*		
Draw "background"			
Implicit region *A*	Calculate the bounding box of the regions *full-screen* and *A*		
Draw "A"			
Refresh	Call `LLUI_DISPLAY_IMPL_flush()` : swap the buffers **and** restore the past		
Implicit region *B*	Store the region *B*		
Draw "B"			
Refresh	Call `LLUI_DISPLAY_IMPL_flush()` : swap the buffers **and** restore the past		

Note: This illustration considers that the clip changes before each drawing and fits the drawing's bounds

Use

Here are the steps around the strategy describing how to use it:

1. Some drawings are performed in the buffer.
2. A `Display.flush()` is asked, the Graphics Engine calls `LLUI_DISPLAY_IMPL_refresh()`.
3. The strategy calls `LLUI_DISPLAY_IMPL_flush()`.
4. The display driver has to implement `LLUI_DISPLAY_IMPL_flush()` : swap the back buffer and the front buffer.
5. As soon as the display *uses* the new front buffer (the new back buffer is now freed), the BSP has to launch a copy of the new front buffer to the new back buffer (use the bounding box).

6. As soon as the copy is done (the copy may be asynchronous), the BSP has to notify the Graphics Engine by calling `LLUI_DISPLAY_setBackBuffer()`, giving the new back buffer address.
7. The Graphics Engine is now unlocked.
8. Some drawings are performed in the back buffer.

MicroUI C Module

Principle

The *MicroUI C module* features some Buffer Refresh Strategies. To select a strategy, configure the define `UI_FEATURE_BRS` in the configuration file `ui_configuration.h`:

- Set `UI_FEATURE_BRS_SINGLE` to select the strategy *Single*.
- Set `UI_FEATURE_BRS_PREDRAW` to select the strategy *Predraw*.
- Set `UI_FEATURE_BRS_LEGACY` to select the strategy *Legacy*.
- Unset the define `UI_FEATURE_BRS` to select the strategy *Default* or to implement a *Custom* strategy.

Options

Some strategies require some options to configure them. The options (some *defines*) are shared between the strategies:

- `UI_FEATURE_BRS_DRAWING_BUFFER_COUNT` (`ui_configuration.h`): configures the available number of back buffers. Used by:
  - Predraw: allowed values are `1`, `2`, or `3` (`1` is valid, but this strategy is not optimized for this use case). See the comment of the define `UI_DISPLAY_BRS_PREDRAW` to increase this value.
  - Single: allowed value is `1` (sanity check).
- `UI_FEATURE_BRS_FLUSH_SINGLE_RECTANGLE` (`ui_configuration.h`): configures the number of rectangles that the strategy gives to the implementation of `LLUI_DISPLAY_IMPL_flush()`. If not set, the number of regions depends on the strategy. If set, only one region is given: the bounding box of all drawing regions. Used by:
  - Predraw: The list of regions is often useless (the LCD driver just has to swap the back and front buffers); however, this list can be used for the buffer policy *Transmit and Swap Buffer*. Calculating the bounding box uses takes a bit of memory and time; if the bounding box is not used, it is recommended to refrain from enabling this option.
  - Single: The list of regions can be useful to refresh small parts of the front buffer.
  - Legacy: This option is never used, and the bounding box of all drawing regions is given to the implementation of `LLUI_DISPLAY_IMPL_flush()`.
- `UI_RECT_COLLECTION_MAX_LENGTH` (`ui_rect_collection.h`): configures the size of the arrays that hold a list of regions (`ui_rect_collection_t`). The default value is `8`; when the collection is full, the strategy replaces all the regions with the bounding box of all regions. Used by:
  - Predraw: number of regions to restore per back buffer.
  - Single: number of regions that the LCD driver has to flush to the front buffer.

Weak Functions

Some strategies use the function `UI_DISPLAY_BRS_restore()` to copy a region from a buffer to another buffer. A default implementation of this function is available in the C file `ui_display_brs.c`. This implementation uses the standard `memcpy`. Override this function to use a GPU for instance.

Debug Traces

The strategies log some events; see *Debug Traces* (see *"[BRS]"* comments).

Simulation

Principle

The `Display` widget in the Front Panel is able to simulate the buffer refresh strategy. It also simulates the *Buffer Policy*.

The default values are:

- Swap Double Buffer for the buffer policy.
- Predraw for the buffer refresh strategy.

Usage

The buffer policy and the refresh strategy can be configured by adding an attribute to the `Display` widget in the `.fp` file. The value of these attributes is the fully qualified name of the class implementing the buffer policy or the refresh strategy. The attributes are:

- `bufferPolicyClass` to set the buffer policy.
- `refreshStrategyClass` to set the refresh strategy.

Example:

```
<ej.fp.widget.Display
  x="0" y="0" width="480" height="272"
  bufferPolicyClass="ej.fp.widget.display.buffer.SwapTripleBufferPolicy"
  refreshStrategyClass="ej.fp.widget.display.brs.PredrawRefreshStrategy"
/>
```

Available Implementations

The available buffer policies are:

- *Swap Double Buffer*: `ej.fp.widget.display.buffer.SwapDoubleBufferPolicy`.
- *Swap Triple Buffer*: `ej.fp.widget.display.buffer.SwapTripleBufferPolicy`.
- *Direct Buffer*: `ej.fp.widget.display.buffer.DirectBufferPolicy`.
- *Single Buffer*: `ej.fp.widget.display.buffer.SingleBufferPolicy`.
- *Transmit and Swap Buffer*: `ej.fp.widget.display.buffer.TransmitSwapBufferPolicy`.

The available refresh strategies are:

- *Single*: ej.fp.widget.display.brs.SingleRefreshStrategy .
- *Predraw*: ej.fp.widget.display.brs.PredrawRefreshStrategy .
- *Legacy*: ej.fp.widget.display.brs.LegacyRefreshStrategy .

Custom Implementation

It is possible to create a new buffer policy by implementing ej.fp.widget.display.buffer.DisplayBufferPolicy .

The buffer policy is responsible for:

- Allocating the necessary buffers, usually in setDisplayProperties(Widget, int, int, int) :

```
FrontPanel.getFrontPanel().newImage(width, height, initialColor, false);
```

- Giving access to the back buffer (the buffer used to draw) in getBackBuffer() .
- Giving access to the front buffer (the buffer displayed in the Display widget) in getFrontBuffer() .
- Flushing the set of modified rectangles from the back buffer to the front buffer in flush(DisplayBufferManager, Rectangle[]) and requesting the display widget to be refreshed.

```
this.displayWidget.repaint();
```

It is possible to create a new refresh strategy by implementing ej.fp.widget.display.brs.BufferRefreshStrategy .

The refresh strategy is responsible for:

- Restoring the past to ensure that the content of the display is correct by calling DisplayBufferManager.restore(Rectangle) .
- Refreshing the display with what has been modified by calling DisplayBufferManager.flush(Rectangle[]) in refresh(DisplayBufferManager) .

  It is notified of the modified regions in newDrawingRegion(DisplayBufferManager, Rectangle, boolean) .

Drawings

Abstraction Layer

All MicroUI drawings (available in the Painter class) call a native function. These native functions are already implemented (in the *MicroUI C Module* for the Embedded VEE Port and in the *Front Panel* for the Simulator). These implementations use the Graphics Engine's software algorithms to perform the drawings.

Each drawing can be overwritten independently in the VEE Port:

- to use another software algorithm (custom algorithm, no third-party library, etc.),
- to use a GPU to perform the operation,
- to target a destination whose format is different from the display back buffer format,

- etc.

  The MicroUI native drawing functions are listed in LLUI_PAINTER_impl.h and LLDW_PAINTER_impl.h (for the Drawing library) for the Embedded VEE Port and, respectively, LUIPainter.java and LLDWPainter.java for the Simulation VEE Port.

  The implementation must handle many constraints: synchronization between drawings, Graphics Engine notification, MicroUI GraphicsContext clip and colors, dirty flush area, etc. The principle of implementing a MicroUI drawing function is described in the chapter *Custom Drawing*.

Destination Format

Since MicroUI 3.2, the destination buffer of the drawings can be different than the display back buffer format (see *MicroEJ Format: Display*). This destination buffer format can be a *standard format* (ARGB8888, A8, etc.) or a *custom format*.

See *Buffered Image* for more information about how to create buffered images with another format than the display format and how to draw in them.

Graphics Engine Software Algorithms

The Graphics Engine features a software implementation for each MicroUI and Drawing libraries drawing. These software algorithms respect the MicroUI GraphicsContext clip and use the current MicroUI GraphicsContext foreground color and optional background color.

The Graphics Engine provides a header file ui_drawing_soft.h (emb), and an implementation instance of UIDrawing that can be retrieved with ej.microui.display.LLUIDisplay.getUIDrawerSoftware() (sim) to let the VEE Port use these algorithms. For instance, a GPU may be able to draw an image whose format is RGB565 but not ARGB1555. For this image format, BSP implementation can call the UI_DRAWING_SOFT_drawImage function.

> **Warning:** These software algorithms only target buffers whose format is the display back buffer format.

MicroUI C Module

Principle

An implementation of LLUI_PAINTER_impl.h is already available on the *MicroUI C module*. This implementation respects the synchronization between drawings and the Graphics Engine notification and reduces (when possible) the MicroUI GraphicsContext clip constraints.

This implementation does not perform the drawings; it only calls the equivalent of drawing available in ui_drawing.h. This allows simplifying how to use a GPU (or a third-party library) to perform a drawing: the ui_drawing.h implementation just has to take into consideration the MicroUI GraphicsContext clip and colors. Synchronization with the Graphics Engine is already performed.

In addition to the implementation of LLUI_PAINTER_impl.h, an implementation of ui_drawing.h is already available in *MicroUI C module* (in *weak* mode). This allows to implement only the functions the GPU can perform. For a given drawing, the weak function imple-

mentation is calling the equivalent of the drawing available in ui_drawing_soft.h (this file lists all drawing functions implemented by the Graphics Engine in software).

Note: More details are available in LLUI_PAINTER_impl.h , ui_drawing.h , LLUI_Display.h , and LLUI_Display_impl.h files.

Default Implementation

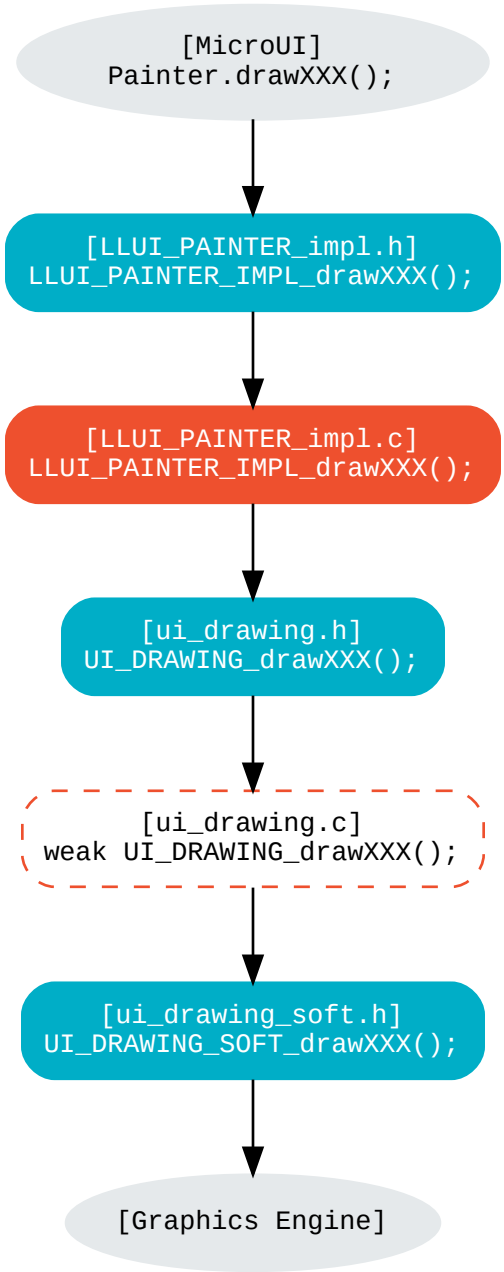
The default implementation is the most used. It takes into account:

- there is only one destination format (the display back buffer format),
- no drawing is overwritten in the BSP (no GPU, third-party library, etc.),
- *non-standard images* cannot be used as a source.

The *MicroUI C module* is designed to simplify the UI VEE Port:

- just need to add the C module in the BSP (no extra code is needed),
- flash footprint is reduced (no extra table to manage several destination formats and several sources),
- functions indirections are limited (the software drawing algorithm is called as faster as possible).

The following diagram illustrates the steps to perform a shape drawing (not an image):



LLUI_PAINTER_IMPL_drawLine (available in MicroUI C Module)

```
void LLUI_PAINTER_IMPL_
↪drawLine(MICROUI_GraphicsContext* gc, jint startX, jint startY, jint endX, jint endY) {
  // Synchronize the native function of MicroUI Painter.drawLine() with the Graphics Engine
  if (LLUI_
↪DISPLAY_requestDrawing(gc, (SNI_callback)&LLUI_PAINTER_IMPL_drawLine)) {
    // Call ui_drawing.h function
    DRAWING_Status status = UI_DRAWING_drawLine(gc, startX, startY, endX, endY);
    // Update the status of the Graphics Engine
    LLUI_DISPLAY_setDrawingStatus(status);
  }
}
```

The Graphics Engine requires synchronization between the drawings. Doing that requires a call to `LLUI_DISPLAY_requestDrawing` at the beginning of native function implementation. This function takes as a parameter the MicroUI GraphicsContext and the pointer on the native function itself. This pointer must be cast in a `SNI_callback`.

UI_DRAWING_drawLine (available in MicroUI C Module)

```
#define UI_DRAWING_DEFAULT_drawLine UI_DRAWING_drawLine
```

The function name is set with preprocessor macros. This name redirection is useful when the VEE Port features multiple destination formats (not the use-case here).

UI_DRAWING_DEFAULT_drawLine (available in MicroUI C Module)

```
// Use the compiler's 'weak' attribute
__weak DRAWING_Status UI_DRAWING_DEFAULT_
↪drawLine(MICROUI_GraphicsContext* gc, jint startX, jint startY, jint endX, jint endY) {
  // Default behavior: call the Graphics Engine's software algorithm
  return UI_DRAWING_SOFT_drawLine(gc, startX, startY, endX, endY);
}
```

Implementing the weak function only consists in calling the Graphics Engine's software algorithm. This software algorithm will respect the GraphicsContext color and clip.

Custom Implementation

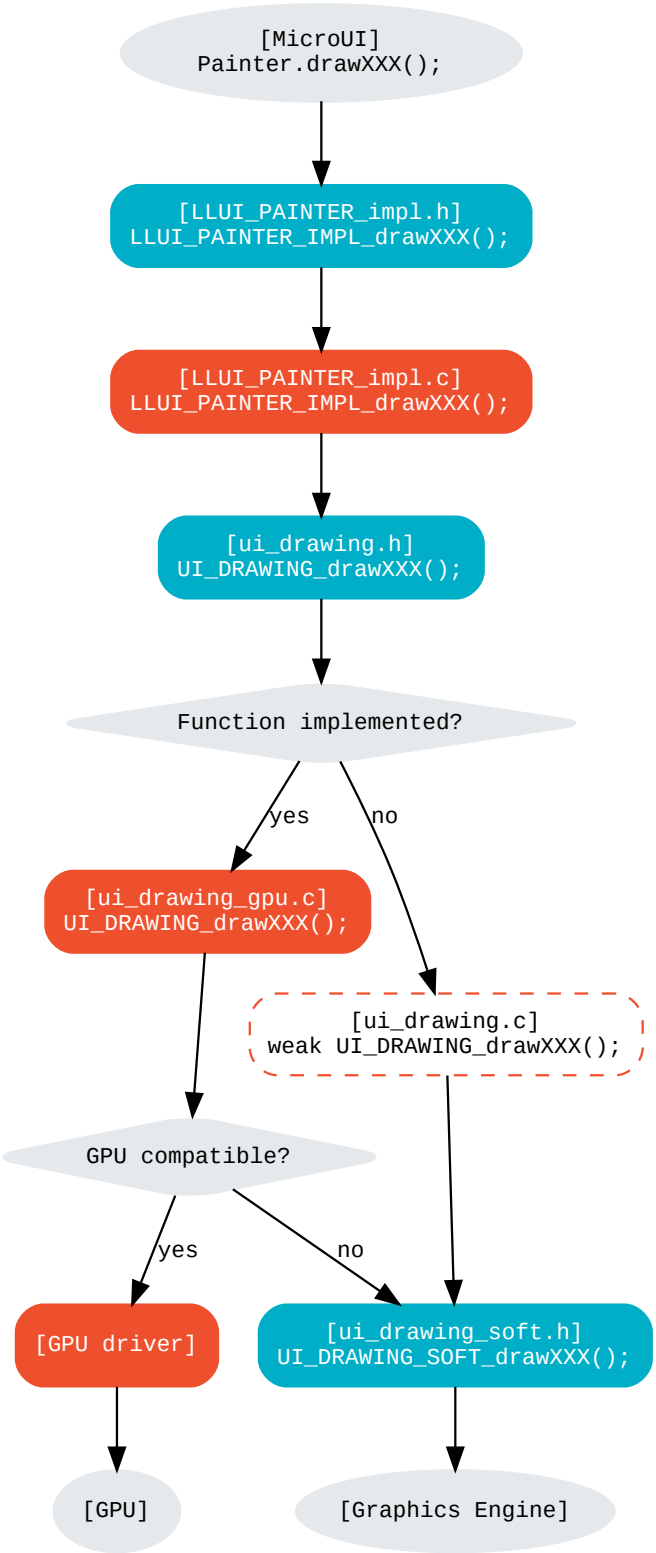
The custom implementation helps connect a GPU or a third-party library. It takes into account:

- there is only one destination format (the display back buffer format),
- *non-standard images* cannot be used as a source.

The *MicroUI C module* is designed to simplify the adding of third-party drawers:

- just need to add the C module in the BSP,
- overwrite only the expected drawing(s),
- a drawing implementation has just to respect the clip and color (synchronization with the Graphics Engine already done),
- flash footprint is reduced (no extra table to manage several destination formats and several sources),
- functions indirections are limited (the drawing algorithm is called as faster as possible).

The following diagram illustrates the steps to perform a shape drawing (not an image):


[MicroUI]
Painter.drawXXX();
[LLUI_PAINTER_impl.h]
LLUI_PAINTER_IMPL_drawXXX();
[LLUI_PAINTER_impl.c]
LLUI_PAINTER_IMPL_drawXXX();
[ui_drawing.h]
UI_DRAWING_drawXXX();
Function implemented?
yes
no
[ui_drawing_gpu.c]
UI_DRAWING_drawXXX();
[ui_drawing.c]
weak UI_DRAWING_drawXXX();
GPU compatible?
yes
no
[GPU driver]
[ui_drawing_soft.h]
UI_DRAWING_SOFT_drawXXX();
[GPU]
[Graphics Engine]

Take the same example as the default implementation (draw a line): the BSP just has to overwrite the weak function UI_DRAWING_drawLine :

UI_DRAWING_drawLine (to write in the BSP)

```
#define UI_DRAWING_GPU_drawLine UI_DRAWING_drawLine
```

The function name should be set with preprocessor macros. This name redirection is useful when the VEE Port features multiple destination formats (not the use-case here).

UI_DRAWING_GPU_drawLine (to write in the BSP)

```
// Contrary to the MicroUI C Module, this function is not "weak"
DRAWING_Status UI_DRAWING_GPU_
↪drawLine(MICROUI_GraphicsContext* gc, jint startX, jint startY, jint endX, jint endY) {

  DRAWING_Status status;

  if (is_gpu_compatible(xxx)) {
    // Can use the GPU to draw the line

    // Retrieve the destination buffer address
    uint8_t* destination_address = LLUI_DISPLAY_getBufferAddress(&gc->image);

    // Configure the GPU clip
    gpu_set_clip(startX, startY, endX, endY);

    // Draw the line
    gpu_draw_line(destination_address, startX, startY, endX, endY, gc->foreground_color);

    // GPU is running: set the right status for the Graphics Engine
    status = DRAWING_RUNNING;
  }
  else {
    // Default behavior: call the Graphics Engine's software algorithm (like "weak" function)
    status = UI_DRAWING_SOFT_drawLine(gc, startX, startY, endX, endY);
  }
  return status;
}
```

First, the drawing function must ensure the GPU can render the expected drawing. If not, the drawing function must perform the same thing as the default weak function: calls the Graphics Engine software algorithm.

The GPU drawing function usually requires the destination buffer address: the drawing function calls LLUI_DISPLAY_getBufferAddress(&gc->image); .

The drawing function has to respect the GraphicsContext clip The MICROUI_GraphicsContext structure holds the clip, and the drawer cannot perform a drawing outside this clip (otherwise, the behavior is unknown). Note the bottom-right coordinates might be smaller than the top-left (in x and/or y) when the clip width and/or height is null. The clip may be disabled (when the current drawing fits the clip); this allows to reduce runtime. See LLUI_DISPLAY_isClipEnabled() .

Hint: Several clip functions are available in LLUI_DISPLAY.h to check if a drawing fits the clip.

Finally, after the drawing, the drawing function has to return the drawing status. Most of the time, the GPU performs *asynchronous* drawings: the drawing is started but not completed. To notify the Graphics Engine, the status to return is DRAWING_RUNNING . In case of the drawing is done after the call to gpu_draw_line() , the status to return is DRAWING_DONE .

Warning: If the drawing status is not set to the Graphics Engine, the global VEE execution is locked: the Graphics Engine waits indefinitely for the status and cannot perform the next drawing.

GPU Synchronization

When a *GPU is used to perform a drawing*, the caller (MicroUI painter native method) returns immediately. This allows the application to perform other operations during the GPU rendering. However, as soon as the application is trying to perform another drawing, the previous drawing made by the GPU must be done. The Graphics Engine is designed to be synchronized with the GPU asynchronous drawings by defining some points in the rendering timeline. It is not optional: MicroUI assumes that a drawing is fully done when it starts a new one. The end of a GPU drawing must notify the Graphics Engine calling LLUI_DISPLAY_notifyAsynchronousDrawingEnd() .

Extended C Modules

Several *C Modules* are available on the MicroEJ Repositories. These modules are compatible with the MicroUI C module (they follow the rules described above) and use one GPU (a C Module per GPU). These C Modules should be fetched in the VEE Port in addition to the MicroUI C Module; it avoids re-writing the GPU port.

Simulation

Principle

This is the same principle as *MicroUI C Module* for the Embedded side:

- The drawing primitive natives called the matching method in LLUIPainter .
- The LLUIPainter synchronizes the drawings with the Graphics Engine and dispatches the drawing itself to an implementation of the interface UIDrawing .
- The Front Panel provides a software implementation of UIDrawing available by calling ej.microui.display.LLUIDisplay.getUIDrawerSoftware() .
- The DisplayDrawer implements UIDrawing and is used to draw in the display back buffer and the images with the same format.

These classes are available in the *UI Pack extension* of the Front Panel Mock.

Note: More details are available in LLUIPainter, UIDrawing, LLUIDisplay, and LLUIDisplayImpl files.

Default Implementation

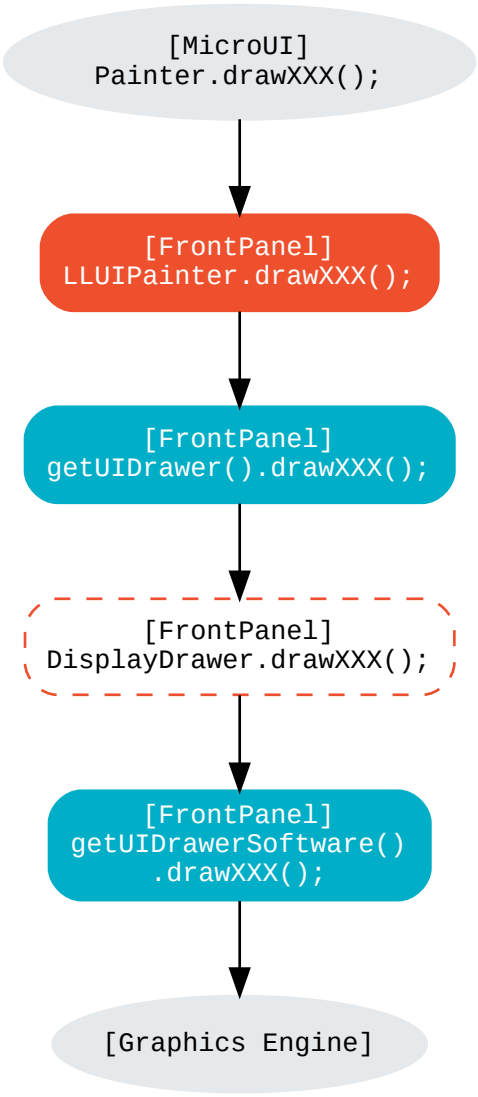
The default implementation is the most used. It considers that:

- there is only one destination format (the display back buffer format),
- no drawing is overwritten in the BSP (no third-party library),
- *non-standard images* cannot be used as a source.

The *UI Pack extension* is designed to simplify the UI VEE Port:

- Simply add the dependency to the UI Pack extension in the VEE Port Front Panel project.
- Function indirections are limited (the software drawing algorithm is called as fast as possible).

The following diagram illustrates the steps to perform a shape drawing (not an image):



LLUIPainter.drawLine (available in UI Pack extension)

```
public static void drawLine(byte[] target, int x1, int y1, int x2, int y2) {

   // Retrieve the Graphics Engine instance
   LLUIDisplay graphicalEngine = LLUIDisplay.Instance;

   // Synchronize the native function of MicroUI Painter.drawLine() with the Graphics Engine
```

(continues on next page)

(continued from previous page)

```
  synchronized (graphicalEngine) {

    // Retrieve the Front Panel instance of the MicroUI GraphicsContext (the destination)
    MicroUIGraphicsContext gc = graphicalEngine.mapMicroUIGraphicsContext(target);

    // Ask to the Graphics Engine if a drawing can be performed on the target
    if (gc.requestDrawing()) {

      // Retrieve the drawer for the GraphicsContext (by default: DisplayDrawer)
      UIDrawing drawer = getUIDrawer(gc);

      // Call UIDrawing function
      drawer.drawLine(gc, x1, y1, x2, y2);
    }
  }
}
```

The Graphics Engine requires synchronization between the drawings. To do that, the drawing is synchronized on the instance of the Graphics Engine itself.

The target (the Front Panel object that maps the MicroUI GraphicsContext) is retrieved in the native drawing method by asking the Graphics Engine to map the byte array (returned by GraphicsContext.getSNIContext()). Like the embedded side, this object holds a clip, and the drawer cannot perform a drawing outside of this clip (otherwise, the behavior is unknown).

DisplayDrawer.drawLine (available in UI Pack extension)

```
@Override
public void drawLine(MicroUIGraphicsContext gc, int x1, int y1, int x2, int y2) {
      LLUIDisplay.Instance.getUIDrawerSoftware().drawLine(gc, x1, y1, x2, y2);
}
```

The implementation of DisplayDrawer simply calls the Graphics Engine's software algorithm. This software algorithm will use the GraphicsContext color and clip.

Custom Implementation

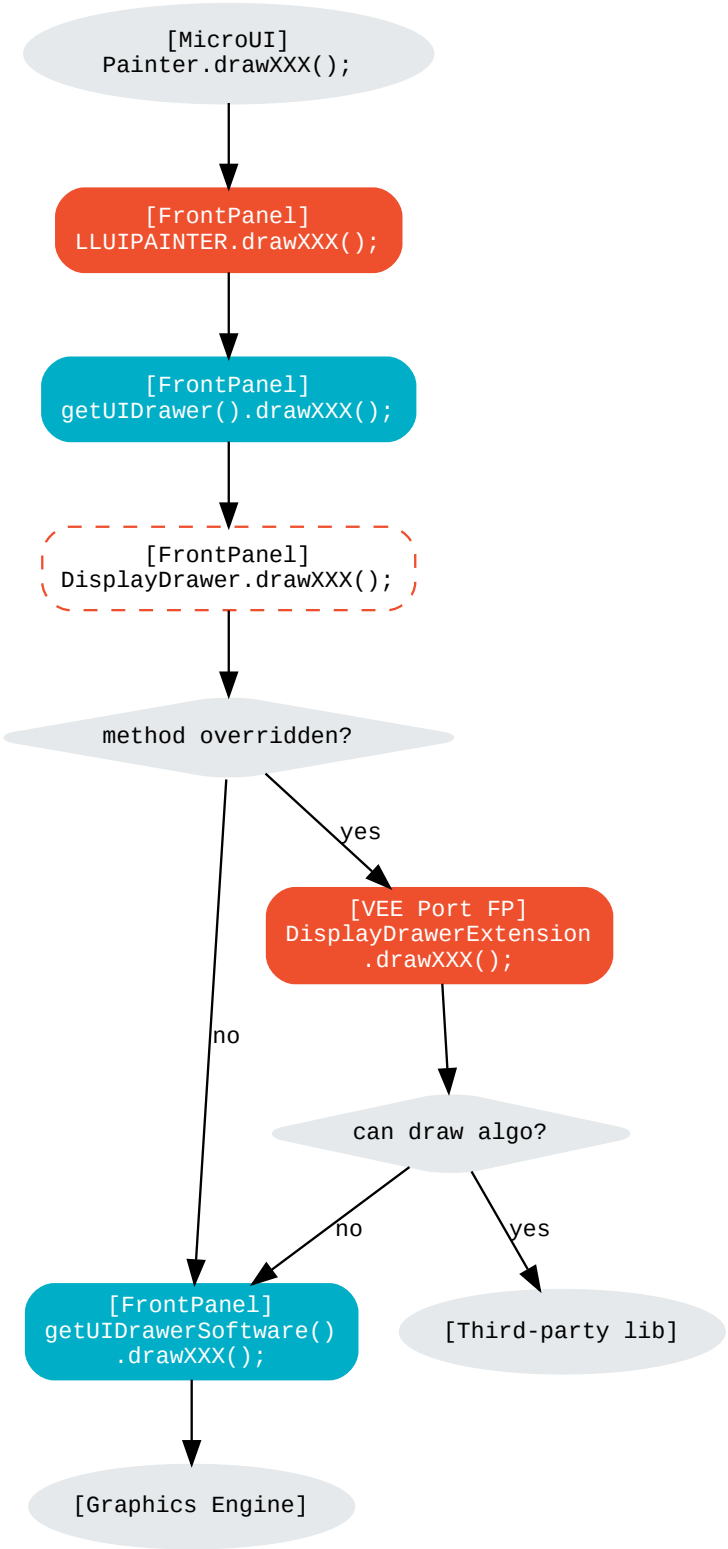
The custom implementation helps connect a third-party library or to simulate the same constraints as the embedded side (the same GPU constraints). It considers that:

- there is only one destination format (the display back buffer format),
- *non-standard images* cannot be used as a source.

The *UI Pack extension* is designed to simplify the adding of third-party drawers:

- Add the dependency to the UI Pack extension in the VEE Port Front Panel project.
- Create a subclass of DisplayDrawer (implementation of the interface UIDrawing).
- Overwrite only the desired drawing(s).
  - Each drawing implementation must comply with the clip and color (synchronization with the Graphics Engine already done).
  - Function indirections are limited (the drawing algorithm is called as fast as possible).
- Register this drawer in place of the default display drawer.

The following diagram illustrates the steps to perform a shape drawing (not an image):


[MicroUI]
Painter.drawXXX();
[FrontPanel]
LLUIPAINTER.drawXXX();
[FrontPanel]
getUIDrawer().drawXXX();
[FrontPanel]
DisplayDrawer.drawXXX();
method overridden?
yes
no
[VEE Port FP]
DisplayDrawerExtension
.drawXXX();
can draw algo?
no
yes
[FrontPanel]
getUIDrawerSoftware()
.drawXXX();
[Third-party lib]
[Graphics Engine]

Let's use the same example as the previous section (draw line function): the Front Panel project has to create its drawer based on the default drawer:

MyDrawer (to write in the Front Panel project)

```
public class MyDrawer extends DisplayDrawer {

  @Override
  public void drawLine(MicroUIGraphicsContext gc, int x1, int y1, int x2, int y2) {

    if (isCompatible(xxx)) {
      // Can use the GPU to␣
↪draw the line on the embedded side: can use another algorithm than the software algorithm

      // Retrieve the AWT Graphics2D
      Graphics2D␣
↪src = (Graphics2D)((BufferedImage)gc.getImage().getRAWImage()).getGraphics();

      // Draw the line using AWT (have to respect clip & color)
      src.setColor(new Color(gc.getRenderingColor()));
      src.drawLine(x1, y1, x2, x2);
    }
    else {
      // Default behavior: call the Graphics Engine's software algorithm
      super.drawLine(gc, x1, y1, x2, y2);
    }
  }
}
```

The Front Panel framework is running over AWT. The method `gc.getImage()` returns a `ej.fp.Image`. It is the representation of a MicroUI Image in the Front Panel framework. The method `gc.getImage().getRAWImage()` returns the implementation of the Front Panel image on the Java SE framework: an AWT BufferedImage. The AWT graphics 2D can be retrieved from this buffered image.

The MicroUI color (`gc.getRenderingColor()`) is converted to an AWT color.

The method behavior is exactly the same as the embedded side; see *Custom Implementation*.

This newly created drawer must now replace the default display drawer. There are two possible ways to register it:

- Declare it as a UIDrawing service.
- Declare it programmatically.

UIDrawing Service

- Create a new file in the resources of the Front Panel project named `META-INF/services/ej.microui.display.UIDrawing` and write the fully qualified name of the previously created drawer:

```
com.mycompany.MyDrawer
```

Programmatically

- Create an empty widget to invoke the new implementation:

```
@WidgetDescription(attributes = { })
public class Init extends Widget{
   @Override
   public void start() {
      super.start();
      LLUIDisplay.Instance.registerUIDrawer(new MyDrawer());
   }
}
```

- Invoke this widget in the .fp file:

```
<frontpanel xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="https:/
↪/developer.microej.com" xsi:schemaLocation="https://developer.microej.com .widget.xsd">
   <device name="STM32429IEVAL" skin="Board-480-272.png">
      <com.is2t.microej.fp.Init/>
      [...]
   </device>
</frontpanel>
```

Custom Drawing

Principle

MicroUI allows adding some custom drawings (drawings not listed in the MicroUI Painter classes). A custom drawing has to respect the same rules as the MicroUI drawings to avoid corrupting the MicroUI execution (flickering, memory corruption, unknown behavior, etc.).

As explained above, MicroUI implementation provides an Abstraction Layer that lists all MicroUI Painter drawing native functions and their implementations (*MicroUI C Module* and *Simulation*). The implementation of MicroUI Painter drawings should be used as a model to implement the custom drawings.

Application Method

```
// Application drawing method
protected void render(GraphicsContext gc) {

   // [...]

   // Set the GraphicsContext color
   gc.setColor(Colors.RED);
   // Draw a red line
   Painter.drawLine(gc, 0, 0, 10, 10);
   // Draw a red custom drawing
   drawCustom(gc.getSNIContext(), 5, 5);

   // [...]
}

// Custom drawing native method
private static native void drawCustom(byte[] graphicsContext, int x, int y);
```

All native functions must have a MicroUI GraphicsContext as a parameter (often the first parameter) that identifies the destination target. The application retrieves this target by calling the

method GraphicsContext.getSNIContext() . This method returns a byte array to give as-is to the drawing native method.

BSP Implementation

The native drawing function implementation pattern is:

```
void Java_com_mycompany_
↪MyPainterClass_drawCustom(MICROUI_GraphicsContext* gc, jint x, jint y) {

  // Tell the Graphics Engine if the drawing can be performed
  if (LLUI_DISPLAY_requestDrawing(gc,
↪ (SNI_callback)&Java_com_mycompany_MyPainterClass_drawCustom)) {
    DRAWING_Status status;

    // Perform the drawing (respecting clip if not disabled)
    status = custom_drawing(LLUI_DISPLAY_getBufferAddress(&gc->image), x, y);

    // Set drawing status
    LLUI_DISPLAY_setDrawingStatus(status);
  }
  // Else: refused drawing
}
```

The target (the MicroUI GraphicsContext) is retrieved in the native drawing function by mapping the MICROUI_GraphicsContext structure in MicroUI native drawing function declaration.

This implementation has to follow the same rules as the custom MicroUI drawings implementation: see *Custom Implementation*.

Simulation

Note: This chapter considers the VEE Port Front Panel project already features a custom drawer that replaces the default drawer DisplayDrawer . See *Custom Implementation*.

The native drawing function implementation pattern is as follows (see below for the explanations):

```
public static void drawCustom(byte[] target, int x, int y) {

  // Retrieve the Graphics Engine instance
  LLUIDisplay graphicalEngine = LLUIDisplay.Instance;

  // Synchronize the native function with the Graphics Engine
  synchronized (graphicalEngine) {

    // Retrieve the Front Panel instance of the MicroUI GraphicsContext (the destination)
    MicroUIGraphicsContext gc = graphicalEngine.mapMicroUIGraphicsContext(target);

    // Ask to the Graphics Engine if a drawing can be performed on the target
    if (gc.requestDrawing()) {

      // Retrieve the drawer for the GraphicsContext (by default: DisplayDrawer)
```

(continues on next page)

(continued from previous page)

```
        UIDrawing drawer = getUIDrawer(gc);

        // Call UIDrawing function
        MyDrawer.Instance.drawSomething(gc, x, y);
    }
  }
}
```

This implementation has to follow the same rules as the custom MicroUI drawings implementation: see *Custom Implementation*.

Drawing Logs

When performing drawing operations, the program may fail or encounter an incident of some kind. MicroUI offers a mechanism allowing the VEE Port to report such incidents to the application through the use of flags.

Usage Overview

When an incident occurs, the VEE Port can report it to the application by setting the *drawing log flags* stored in the graphics context. The flags will then be made available to the application. See *Drawing Logs* for more information on reading the flags in the application.

Without an intervention from the application, the drawing log flags retain their values through every call to drawing functions and are cleared when a flush is performed.

Note: The clearing of drawing log flags can be disabled at build time by the application developer.

Incidents are split into two categories:

- *Non-critical* incidents, or *warnings*, are incidents that the application developer may ignore. The flags are made available for the application to check, but without an explicit statement in the application, these incidents will be ignored silently.
- *Critical* incidents, or *errors*, are failures significant enough that the application developer should not ignore them. As for warnings, the application may check the drawing log flags explicitly. However, when flushing the display, the application checks the flags and throws an exception if an error has been reported.

Warning: As this behavior can be disabled at build time, the drawing log flags are meant to be used as a **debugging hint** when the application does not display what the developer expects. The VEE Port must **not** rely on applications throwing an exception if an error was reported or on the drawing log flags being reset after the display is flushed.

Note: Any incident may be either a *warning* or an *error*. They are differentiated with the special flag DRAWING_LOG_ERROR.

Available Constants

MicroUI offers a set of flag constants to report incidents to the application. They are defined and documented in LLUI_PAINTER_impl.h (for embedded targets) and LLUIPainter (for front panels).

Refer to the *application documentation* for the exhaustive list of drawing logs.

Hint: Sometimes, incidents may match more than one flag constant. In such cases, the VEE Port may report the incident with multiple flags by combining them with the bitwise OR operator (|), just like any other flags. For example, an out-of-memory incident occurring in an underlying drawing library may be reported with the value DRAWING_LIBRARY_INCIDENT | DRAWING_OUT_OF_MEMORY .

Embedded Targets

MicroUI exposes two functions to be used in the VEE Port. Both functions are declared in LLUI_DISPLAY.h , and their documentation is available in that file.

- LLUI_DISPLAY_reportWarning reports a warning to the application. It will set the flags passed as an argument in the graphics context. It will *not* reset the previous flag values, thus retaining all reported incidents until the application clears the flags.
- LLUI_DISPLAY_reportError reports an error to the application. It behaves similarly to LLUI_DISPLAY_reportWarning , except it will additionally set the flag DRAWING_LOG_ERROR . This special flag will cause an exception to be thrown in the application the next time the application checks the flags.

For example, if the VEE Port contains a custom implementation to draw a line that may cause an out-of-memory error, it could report this error this way:

```
void LLUI_PAINTER_IMPL_
↪drawLine(MICROUI_GraphicsContext* gc, jint startX, jint startY, jint endX, jint endY) {
  // This could cause an out-of-memory error.
  unsigned int result = custom_line_drawing();

  // Check if an error occurred.
  if (result == OUT_OF_MEMORY) {
    // If an error occurred, set the corresponding flag.
    LLUI_DISPLAY_reportError(gc, DRAWING_LOG_OUT_OF_MEMORY);
  }
}
```

Simulator

Similarly, MicroUI exposes two functions to set drawing log flags in the front panel implementation. Both functions are declared as methods of the interface MicroUIGraphicsContext and are documented there. The Graphics Engine provides an implementation for these methods.

- MicroUIGraphicsContext.reportWarning behaves like LLUI_DISPLAY_reportWarning and reports a warning to the application.
- MicroUIGraphicsContext.reportError behaves like LLUI_DISPLAY_reportError and reports an error to the application.

The front panel version of the previous example that reported an out-of-memory error would look like this:

```
public static void drawLine(byte[] target, int startX, int startY, int endX, int endY) {
  LLUIDisplay engine = LLUIDisplay.Instance;

  synchronized (engine) {
    MicroUIGraphicsContext gc = engine.mapMicroUIGraphicsContext(target);

    // This could cause an out-of-memory error.
    int result = CustomDrawings.drawLine();

    // Check if an error occurred.
    if (result == Constants.OUT_OF_MEMORY) {
      // If an error occurred, set the corresponding flag.
      gc.reportError(gc, DRAWING_LOG_OUT_OF_MEMORY);
    }
  }
}
```

Images

Overview

Principle

The Image Engine is designed to make the distinction between three kinds of MicroUI images:

- the images which can be used by the application without a loading step: class Image,
- the images which requires a loading step before being usable by the application: class ResourceImage,
- the buffered images where the application can draw into: class BufferedImage.

The first kind of image requires the Image Engine to be able to use (get, read and draw) an image referenced by its path without any loading step. The *open* step should be very fast: just have to find the image in the application resources list and create an Image object which targets the resource. No RAM memory to store the image pixels is required: the Image Engine directly uses the resource address (often in FLASH memory). And finally, *closing* step is useless because there is nothing to free (except Image object itself, via the garbage collector).

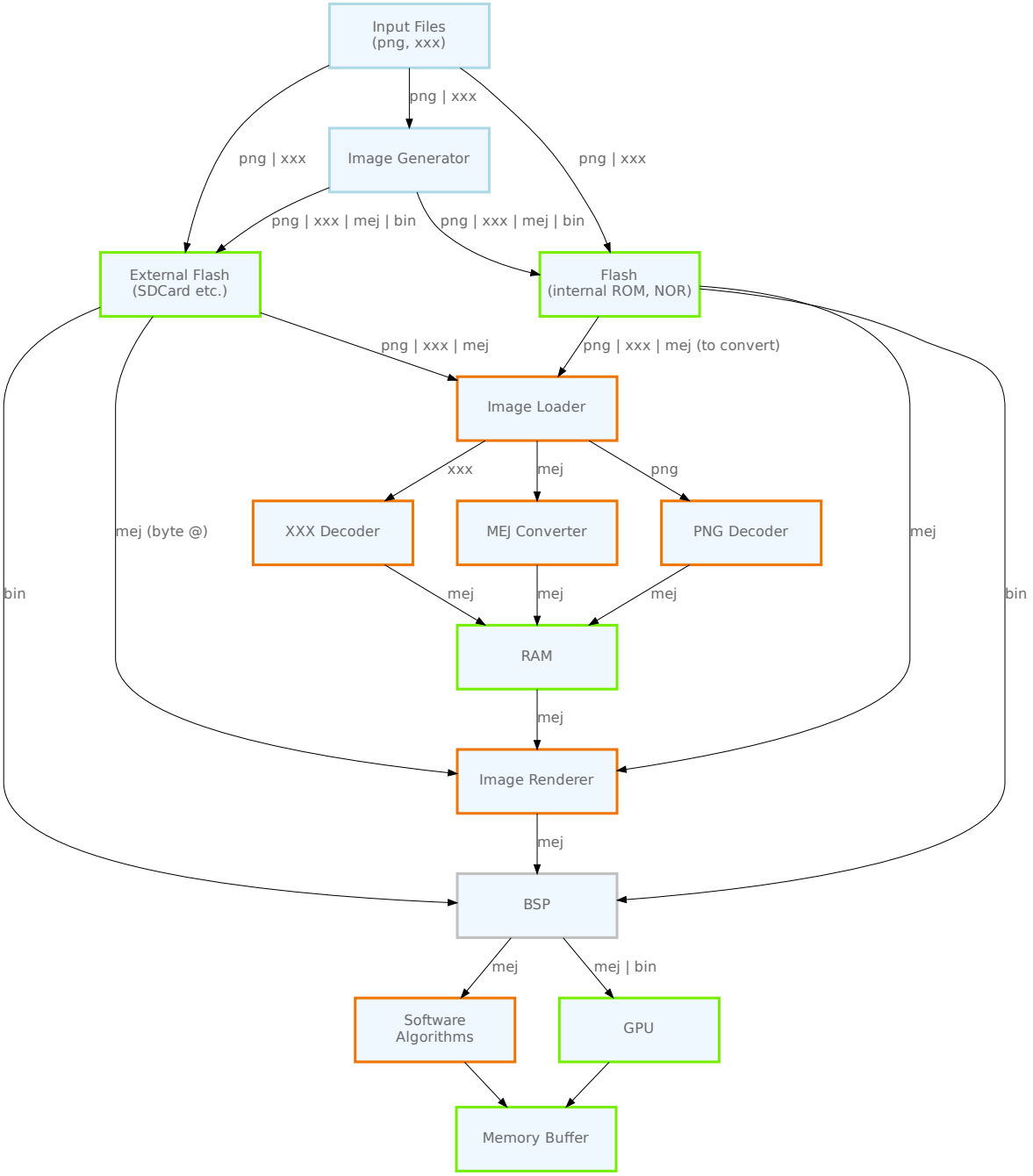
The second kind of image requires the Image Engine to be able to use (load, read and draw) an image referenced by its path with or without any loading step. When the image is understandable by the Image Engine without any loading step, the image is considered like the first kind of image (fast *open* step, no RAM memory, useless *closing* step). When a loading step is required (dynamic decoding, external resource loading, image format conversion), the *open* state becomes longer and a buffer in RAM is required to store the image pixels. By consequence a *closing* step is required to free the buffer when image becomes useless.

The third kind of image requires, by definition, a buffer to store the image data. Image Engine must be able to use (create, read and draw) this kind of image. The *open* state consists in creating a buffer. By consequence a *closing* step is required to free the buffer when the image becomes useless. Contrary to the other kinds of images, the application will be able to draw into this image.

Functional Description

The Image Engine is composed of:

- An “Image Generator” module, for converting images into a MicroEJ format (known by the Image Engine Renderer) or into a VEE Port binary format (cannot be used by the Image Engine Renderer), before runtime (pre-generated images).
- The “Image Loader” module, for loading, converting and closing the images.
- A set of “Image Decoder” modules, for converting standard image formats into a MicroEJ format (known by the Image Renderer) at runtime. Each Image Decoder is an additional module of the main module “Image Loader”.
- The “Image Renderer” module, for reading and drawing the images in MicroEJ format.



- Colors:
  - blue: off-board elements (tools, files).
  - green: hardware elements (memory, processor).

  - orange: on-board Graphics Engine elements.
  - gray: BSP.

- Line labels:

  - png : symbolizes all standard image input formats (PNG, JPG, etc.).
  - xxx : symbolizes a non-standard input format.
  - mej : symbolizes the MicroEJ output format (*MicroEJ Format: Standard*).
  - bin : symbolizes a VEE Port binary format (*Binary Format*).

    Process overview:

    1. The user specifies the pre-generated images to embed (see *Image Generator*) and / or the images to embed as regular resources (see *Encoded Image*).
    2. The files are embedded as resources with the application. The files' data are linked into the FLASH memory.
    3. When the application creates a MicroUI Image object, the Image Loader loads the image, calling the right sub Image Engine module (see *Image Generator* and *Encoded Image*) to decode the specified image.
    4. When the application draws this MicroUI Image on the display (or on buffered image), the decoded image data is used, and no more decoding is required, so the decoding is done only once.
    5. When the MicroUI Image is no longer needed, it must be closed explicitly by the application. The Image Engine Core asks the right sub Image Engine module (see *Image Generator* and *Encoded Image*) to free the image working area.

Dependencies

- MicroUI module (see *MicroUI*),
- Display module (see *Display*): the characteristics of the target display are used to configure the *Image Generator*.

Image Format

The Image Engine makes the distinction between:

- The *input format*: the format of the original image.
- The *output format*: the image format used by the Image Renderer.

Several formats are managed in input: PNG, JPEG, BMP, etc. A specific VEE Port can support additional input formats.

Several formats are managed in output: the MicroEJ formats and the binary format. The output format can be:

- Generated from the input format using the off-board tool *Image Generator* at application compile-time.
- Generated from the input format by using a *runtime decoder* of the *Image Loader* at application run-time.
- Dynamically created when using a BufferedImage.

The Image Renderer manages only the MicroEJ formats (*MicroEJ Format: Standard*, *MicroEJ Format: Display*, and *MicroEJ Format: Custom*).

The following table lists all the formats and their usage.

Table 77: Image Formats

Format	Input	Output	BufferedImage
MicroEJ Format: Display	no	yes	yes
MicroEJ Format: Standard	no	yes	yes[1]
MicroEJ Format: Grayscale	no	yes	yes[Page 1024, 1]
MicroEJ Format: RLE Compressed	no	yes	no
MicroEJ Format: Custom	no	not yet	yes[1]
Binary Format	no	yes	no
Original Input Format	yes	no	no

The following sections list all the formats and their usage.

MicroEJ Format: Display

The display back buffer holds a pixel encoding which is:

- standard: see *Standard Output Formats*,
- grayscale: see *Grayscale Output Formats*,
- non-standard: see *Display Output Format* and *Pixel Structure*.

  The non-standard display format can be customized to encode the pixel in the same encoding as the display. The number of bits per pixel and the pixel bit organization is asked during the MicroEJ format generation and when the `drawImage` algorithms are running. If the image to encode contains some transparent pixels, the output file will embed the transparency according to the display's implementation capacity. When all pixels are fully opaque, no extra information will be stored in the output file to free up some memory space.

  Notes:

- From the Image Engine point of view, the non-standard display format stays a MicroEJ format, readable by the Image Renderer.
- The required memory to encode an image with a non-standard display format is similar to *MicroEJ Format: Standard*.
- When the display format is standard or grayscale, the encoded image format is replaced by the related standard format.
- The *Graphics Engine's drawing software algorithms* only target (are only compatible with) the buffered images whose format is the same as the display format (standard or non-standard).

[1] Need some support in the VEE Port to support formats different than the display one (see Buffered Image).

MicroEJ Format: Standard

See *Standard Output Formats*.

This format requires a small header (around 20 bytes) to store the image size (width, height), format, flags (is_transparent, etc.), row stride, etc. The required memory also depends on the number of bits per pixel of the MicroEJ format:

```
required_memory = header + (image_width * image_height) * bpp / 8;
```

The pixel array is stored after the MicroEJ image file header. A padding between the header and the pixel array is added to force to start the pixel array at a memory address aligned on the number of bits-per-pixels.

The pixel order follows this rule:

```
pixel_offset = (pixel_Y * image_width + pixel_X) * bpp / 8;
```

Here are the conversions of 32-bit to each format:

- ARGB8888: 32-bit format, 8 bits for transparency, 8 per color.

```
u32 convertARGB8888toRAWFormat(u32 c){
   return c;
}
```

- ARGB4444: 16-bit format, 4 bits for transparency, 4 per color.

```
u32 convertARGB8888toRAWFormat(u32 c){
   return 0
          | ((c & 0xf0000000) >> 16)
          | ((c & 0x00f00000) >> 12)
          | ((c & 0x0000f000) >> 8)
          | ((c & 0x000000f0) >> 4)
          ;
}
```

- ARGB1555: 16-bit format, 1 bit for transparency, 5 per color.

```
u32 convertARGB8888toRAWFormat(u32 c){
   return 0
          | (((c & 0xff000000) == 0xff000000) ? 0x8000 : 0)
          | ((c & 0xf80000) >> 9)
          | ((c & 0x00f800) >> 6)
          | ((c & 0x0000f8) >> 3)
          ;
}
```

- RGB888: 24-bit format, 8 per color.

```
u32 convertARGB8888toRAWFormat(u32 c){
   return c & 0xffffff;
}
```

- RGB565: 16-bit format, 5 for red, 6 for green, 5 for blue.

```
u32 convertARGB8888toRAWFormat(u32 c){
    return 0
            | ((c & 0xf80000) >> 8)
            | ((c & 0x00fc00) >> 5)
            | ((c & 0x0000f8) >> 3)
            ;
}
```

- A8: 8-bit format, only the source image's transparency is encoded (option alpha), or the source image is pre-processed (option grayscale); see *Alpha Format*.

  Option alpha

  Option grayscale

```
u32 convertARGB8888toRAWFormat(u32 c){
  return (c >> 24) & 0xff;
}
```

```
u32 convertARGB8888toRAWFormat(u32 c){
  return 0xff - (toGrayscale(c) & 0xff);
}
```

- A4: 4-bit format, only the source image's transparency is encoded (option alpha), or the source image is pre-processed (option grayscale); see *Alpha Format*.

  Option alpha

  Option grayscale

```
u32 convertARGB8888toRAWFormat(u32 c){
  return (c >> 28) & 0xf;
}
```

```
u32 convertARGB8888toRAWFormat(u32 c){
  return (0xff - (toGrayscale(c) & 0xff)) / 0x11;
}
```

- A2: 2-bit format, only the source image's transparency is encoded (option alpha), or the source image is pre-processed (option grayscale); see *Alpha Format*.

  Option alpha

  Option grayscale

```
u32 convertARGB8888toRAWFormat(u32 c){
  return (c >> 30) & 0x3;
}
```

```
u32 convertARGB8888toRAWFormat(u32 c){
  return (0xff - (toGrayscale(c) & 0xff)) / 0x55;
}
```

- A1: 1-bit format, only the source image's transparency is encoded (option alpha), or the source image is pre-processed (option grayscale); see *Alpha Format*.

  Option alpha

  Option grayscale

```
u32 convertARGB8888toRAWFormat(u32 c){
   return (c >> 31) & 0x1;
}
```

```
u32 convertARGB8888toRAWFormat(u32 c){
   return (0xff - (toGrayscale(c) & 0xff)) / 0xff;
}
```

MicroEJ Format: Grayscale

See *Grayscale Output Formats*.

This format requires a small header (around 20 bytes) to store the image size (width, height), format, flags (is_transparent, etc.), row stride, etc. The required memory also depends on the number of bits per pixel of the MicroEJ format:

```
required_memory = header + (image_width * image_height) * bpp / 8;
```

- AC44: 4 bits for transparency, 4 bits with grayscale conversion.

```
u32 convertARGB8888toRAWFormat(u32 c){
   return 0
      | ((color >> 24) & 0xf0)
      | ((toGrayscale(color) & 0xff) / 0x11)
      ;
}
```

- AC22: 2 bits for transparency, 2 bits with grayscale conversion.

```
u32 convertARGB8888toRAWFormat(u32 c){
   return 0
      | ((color >> 28) & 0xc0)
      | ((toGrayscale(color) & 0xff) / 0x55)
      ;
}
```

- AC11: 1 bit for transparency, 1 bit with grayscale conversion.

```
u32 convertARGB8888toRAWFormat(u32 c){
   return 0
      | ((c & 0xff000000) == 0xff000000 ? 0x2 : 0x0)
      | ((toGrayscale(color) & 0xff) / 0xff)
      ;
}
```

- C4: 4 bits with grayscale conversion.

```
u32 convertARGB8888toRAWFormat(u32 c){
   return (toGrayscale(c) & 0xff) / 0x11;
}
```

- C2: 2 bits with grayscale conversion.

```
u32 convertARGB8888toRAWFormat(u32 c){
   return (toGrayscale(c) & 0xff) / 0x55;
}
```

- C1: 1 bit with grayscale conversion.

```
u32 convertARGB8888toRAWFormat(u32 c){
    return (toGrayscale(c) & 0xff) / 0xff;
}
```

The pixel order follows this rule:

```
pixel_offset = (pixel_Y * image_width + pixel_X) * bpp / 8;
```

MicroEJ Format: RLE Compressed

See *Compressed Output Formats*.

MicroEJ Format: Custom

A custom format embeds a buffer whose data are VEE Port specific. This data may be:

- a pixel buffer whose encoding is different than the formats proposed before,
- a buffer that is not a pixel buffer.

This format is identified by a specific format value between 0 and 7: see custom formats.

Images with a custom format can be used as any other image. For that, it requires some support at different levels depending on their usage:

- To convert an image to this format at compile-time and embed it, an extension of the image generator is necessary; see *VEE Port MicroEJ Custom Format*.
- To create a new one at runtime, some native extension is necessary; see *Buffered Image*.
- To use it as a source (to draw the image in another buffer), some native extension is necessary; see *Custom Format Support*.
- To use it as a destination (to draw into the image), some native extension is necessary; see *Buffered Image*.

Binary Format

This format is not compatible with the Image Renderer and MicroUI. It can be used by MicroUI addon libraries which provide their image management procedures.

Advantages:

- Encoding is known by VEE Port.
- Compression is inherent to the format itself.

Disadvantages:

- This format cannot target a MicroUI Image (unsupported format).

Original Input Format

See *Unspecified Output Format*.

An image can be embedded without any conversion/compression. This allows embedding the resource as it is to keep the source image characteristics (compression, bpp, etc.). This option produces the same result as specifying an image as a resource in the MicroEJ launcher.

The following table lists the original formats that can be decoded at run-time and/or compile-time:

- Image Generator: the off-board tool that converts an image into an output format. All AWT ImageIO default formats are supported and always enabled.
- Front Panel: the decoders embedded by the simulator part. All AWT ImageIO default formats are supported but disabled by default.
- Runtime Decoders: the decoders embedded by the embedded part.

Table 78: Original Image Formats

Type	Image Generator	Front Panel	Runtime Decoders
Graphics Interchange Format (GIF)	yes	yes[2]	no[7]
Joint Photographic Experts Group (JPEG)	yes	yes[2]	no[7]
Portable Network Graphics (PNG)	yes	yes[3]	yes[3]
Windows bitmap (BMP)	yes	yes[4]	yes/no[4]
Web Picture (WebP)	yes[5]	yes[5]	yes[6]

GPU Format Support

The MicroEJ formats *display*, *standard* and *grayscale* may be customized to be compatible with the hardware (usually GPU). It can be extended by one or several restrictions on the pixels array:

- Its start address has to be aligned on a higher value than the number of bits-per-pixels.
- A padding has to be added after each line (row stride).
- The MicroEJ format can hold a VEE Port-dependent header between the MicroEJ format header (start of file) and the pixel array. The MicroEJ format is designed to let the VEE Port encode and decode this additional header. This header is unnecessary and never used for Image Engine software algorithms.

Note: From the Image Engine point of view, the format stays a MicroEJ format, readable by the Image Engine Renderer.

Advantages:

[2] The formats are disabled by default; see Image Decoders.
[7] The UI-pack does not provide some runtime decoders for these formats, but a BSP can add its decoders (see Encoded Image).
[3] The PNG format is supported when the module PNG is selected in the VEE Port configuration file (see Encoded Image).
[4] The Monochrome BMP is supported when the module BMPM is selected in the VEE Port configuration file (see Encoded Image); the colored BMP format is only supported by the Front Panel (disabled by default, see Image Decoders).
[5] Install the tool com.microej.tool#imageio-webp-1.0.1 from the Developer Repository in the VEE Port to support the WEBP format (see Service Image Loader and Image Decoders).
[6] Install the C component com.microej.clibrary.thirdparty#libwebp-1.0.1 in the BSP to support the WEBP format at runtime.

- The GPU recognizes encoding.
- Drawing an image is often very fast.
- Supports opacity encoding.

Disadvantages:

- No compression: the image size in bytes is proportional to the number of pixels. The required memory is similar to *MicroEJ Format: Standard* when no custom header exists.

When the MicroEJ format holds another header (called `custom_header`), the required memory is:

```
required_memory = header + custom_header + (image_width * image_height) * bpp / 8;
```

The row stride allows adding some padding at the end of each line to start the next line at an address with a specific memory alignment; it is often required by hardware accelerators (GPU). The row stride is, by default, a value in relation to the image width: `row_stride_in_bytes = image_width * bpp / 8`. Thanks to the Abstraction Layer API `LLUI_DISPLAY_IMPL_getNewImageStrideInBytes`, it can be customized at image buffer creation. The required memory becomes:

```
required_memory = header + custom_header + row_stride * image_height;
```

Image Generator

Principle

The Image Generator module is an off-board tool that generates image data that is ready to be displayed without needing additional runtime memory. The two main advantages of this module are:

- A pre-generated image is already encoded in the format known by the Image Renderer (MicroEJ format) or by the VEE Port (custom binary format). The time to create an image is very fast and does not require any RAM (Image Loader is not used).
- No extra support is needed (no runtime Image Decoder).

Functional Description

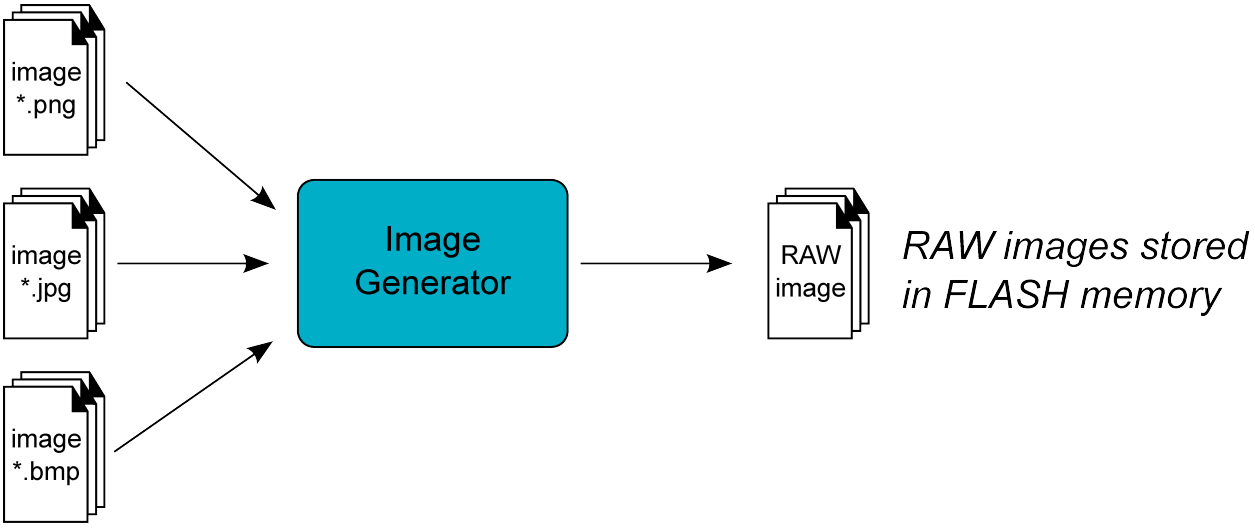


Fig. 232: Image Generator Principle

Process overview (see too *Functional Description*)

1. The user defines, in a text file, the images to load.
2. The Image Generator outputs a binary file for each image to convert.
3. The raw files are embedded as (hidden) resources within the MicroEJ Application. The binary files' data are linked into the FLASH memory.
4. When the application creates a MicroUI Image object which targets a pre-generated image, the Image Engine has only to create a link from the MicroUI image object to the data in the FLASH memory. Therefore, the loading is very fast; only the image data from the FLASH memory is used: no copy of the image data is sent to the RAM first.
5. When the MicroUI Image is no longer needed, it is garbage-collected by the VEE Port, which just deletes the useless link to the FLASH memory.

The image generator can run in two modes:

- Standalone mode: the image to convert (input files) are standard (PNG, JPEG, etc.), the generated binary files are in MicroEJ format and do not depend on VEE Port characteristics or restrictions (see *MicroEJ Format: Standard*).
- Extended mode: the image to convert (input files) may be custom, the generated binary files can be encoded in customized MicroEJ format (can depend on several VEE Port characteristics and restrictions, see *MicroEJ Format: Display*, *GPU Format Support* and *MicroEJ Format: Custom*) or the generated files are encoded in another format than MicroEJ format (binary format, see *Binary Format*).

Structure

The Image Generator module is constituted from several parts, the core part and services parts:

- “Core” part: it takes an images list file as entry point and generates a binary file (no specific format) for each file. To read a file, it redirects the reading to the available service loaders. To generate a binary file, it redirects the encoding to the available service encoders.
- “Service API” part: it provides some APIs used by the core part to load input files and to encode binary files. It also provides some APIs to customize the MicroEJ format.
- “Standard input format loader” part: this service loads standard image files (PNG, JPEG, etc.).
- “MicroEJ format generator” part: this service encodes an image in MicroEJ format.

Standalone Mode

The standalone Image Generator embeds all parts described above. By consequence, once installed in a VEE Port, the standalone image generator does not need any extended module to generate MicroEJ files from standard images files.

Extended Mode

To increase the capabilities of Image Generator, the extension must be built and added in the VEE Port. As described above this extension will be able to:

- read more input image file formats,
- extend the MicroEJ format with VEE Port characteristics,
- encode images in a third-party binary format.

To do that the Image Generator provides some services to implement. This chapter explain how to create and include this extension in the VEE Port. Next chapters explain the aim of each service.

1. Create a `std-javalib` project. The module name must start with the prefix `imageGenerator` (for instance `imageGeneratorMyVEEPort`).
2. Add the dependency:

```
<dependency org="com.microej.pack.ui" name="ui-pack" rev="x.y.z">
  <artifact name="imageGenerator" type="jar"/>
</dependency>
```

Where `x.y.z` is the UI Pack version used to build the VEE Port (minimum `13.0.0`). The `module.ivy` should look like:

```
<ivy-module version="2.0" xmlns:ea="http://www.easyant.org" xmlns:m="http://
↪www.easyant.org/ivy/maven" xmlns:ej="https://developer.microej.com" ej:version="2.0.0">

  <info organisation="com.microej.
↪microui" module="imageGeneratorMyVEEPort" status="integration" revision="1.0.0">
    <ea:build␣
↪organisation="com.is2t.easyant.buildtypes" module="build-std-javalib" revision="2.+"/>
  </info>
```

(continues on next page)

(continued from previous page)

```
  <configurations defaultconfmapping="default->default;provided->provided">
    <conf name=
↪"default" visibility="public" description="Runtime dependencies to other artifacts"/>
    <conf name="provided" visibility=
↪"public" description="Compile-time dependencies to APIs provided by the VEE Port"/>
    <conf name="documentation
↪" visibility="public" description="Documentation related to the artifact (javadoc, PDF)"/>
    <conf name="source" visibility="public" description="Source code"/>
    <conf name=
↪"dist" visibility="public" description="Contains extra files like README.md, licenses"/>
    <conf name="test" visibility=
↪"private" description="Dependencies for test execution. It is not required for normal use␣
↪of the application, and is only available for the test compilation and execution phases."/>
  </configurations>

  <publications/>

  <dependencies>
    <dependency org="com.microej.pack.ui" name="ui-pack" rev="[UI Pack version]">
      <artifact name="imageGenerator" type="jar"/>
    </dependency>
  </dependencies>
</ivy-module>
```

3. Create the folder `META-INF/services` in source folder `src/main/resources` (this folder will be filled in later).
4. When a service is added (see next chapters), build the project.
5. Copy the generated jar: `target~/artifacts/imageGeneratorMyVEEPort.jar` in the VEE Port configuration project folder: `MyVEEPort-configuration/dropins/tools/`
6. Rebuild the VEE Port.

Advanced: Test the Extension Project

To quickly test an extension project without rebuilding the VEE Port or manually exporting the project, add the *Application Option* `ej.imagegenerator.extension.project` to the absolute path of an Image Generator Extension project (e.g. `c:\mycompany\myimagegeneratorextension`). The Image Generator will use the specified Image Generator Extension project instead of the one included in the VEE Port. This feature is useful for locally testing certain changes in the Image Generator Extension project.

```
-Dej.imagegenerator.extension.project=${project_loc:myimagegeneratorextension}
```

Warning: This feature only works if the VEE Port has been built with the Image Generator module enabled.

The VEE Port will not actually contain the changes until a new VEE Port is built: the VEE Port dropins folder must be updated after any changes to the Image Generator Extension project.

Warning: Using this feature automatically disables the *image cache*.

Service Image Loader

The standalone Image Generator is not able to load all images formats, for instance SVG format. The service loader can be used to add this feature in order to generate an image file in MicroEJ format. There are two ways to populate the service loader: create a custom implementation of com.microej.tool.ui.generator.MicroUIRawImageGeneratorExtension or javax.imageio.spi.ImageReaderSpi .

MicroUIRawImageGeneratorExtension

This service allows to add a custom image reader.

1. Open image generator extension project.
2. Create an implementation of interface com.microej.tool.ui.generator.MicroUIRawImageGeneratorExtension .
3. Create the file META-INF/services/com.microej.tool.ui.generator.MicroUIRawImageGeneratorExtension and open it.
4. Note down the name of created class, with its package and classname.
5. Rebuild the image generator extension, copy it in VEE Port configuration project (dropins/tools/) and rebuild the VEE Port (see above).

Note: The class com.microej.tool.ui.generator.BufferedImageLoader already implements the interface. This implementation is used to load standard images. It can be sub-classed to add some behavior.

ImageReaderSpi

This extension is part of AWT ImageIO. By default, the ImageIO class only manages the standard image formats JPG, PNG, BMP and GIF. It allows to add some image readers by adding some implementations of the service *javax.imageio.spi.ImageReaderSpi*.

Since UI Pack 13.2.0, the Image Generator automatically includes new image decoders (new ImageIO services, see the class com.microej.tool.ui.generator.BufferedImageLoader), compiled in JAR files that follow this convention:

1. The JAR contains the service declaration /META-INF/services/javax.imageio.spi.ImageReaderSpi ,
2. The JAR filename's prefix is *imageio-*,
3. The JAR location is the VEE Port configuration project's dropins/tools/ directory.

Note: The same JAR is used by the Image Generator and by the *Front Panel*.

Customize MicroEJ Standard Format

As mentioned above (*MicroEJ Format: Display* and *GPU Format Support*), the MicroEJ format can be extended by notions specific to the VEE Port (and often to the GPU the VEE Port is using). The generated file stays a MicroEJ file format, usable by the Image Renderer. Additionally, the file becomes compatible with the VEE Port constraints.

1. Open image generator extension project.
2. Create a subclass of `com.microej.tool.ui.generator.BufferedImageLoader` (to be able to load standard images) or create an implementation of interface `com.microej.tool.ui.generator.MicroUIRawImageGeneratorExtension` (to load custom images).
3. Override method `convertARGBColorToDisplayColor(int)` if the VEE Port's display pixel encoding is not standard (see *Pixel Structure*).
4. Override method `getStride(int)` if a padding must be added after each line.
5. Override method `getOptionalHeader()` if an additional header must be added between the MicroEJ file header and pixels array. The header size is also used to align image memory address (custom header is aligned on its size).
6. Create the file `META-INF/services/com.microej.tool.ui.generator.MicroUIRawImageGeneratorExtension` and open it.
7. Note down the name of created class, with its package and classname.
8. Rebuild the image generator extension, copy it in VEE Port configuration project and rebuild the VEE Port (see above).

   If the only constraint is the pixels array alignment, the Image Generator extension is not useful:

   SDK 6

   SDK 5

   Add the property `com.microej.pack.imageBuffer.memoryAlignment` in the properties file `configuration.properties` of the VEE Port project.

   Add the property `imageBuffer.memoryAlignment` in the properties file `display/display.properties` of the VEE Port project.

   This alignment will be used by the Image Generator and also by the Image Loader.

VEE Port MicroEJ Custom Format

The Image Generator does not yet provide a service for generating the *MicroEJ Format: Custom*. A custom image can only be created at runtime, see *Buffered Image*.

VEE Port Binary Format

The Image Generator can generate a *binary file* compatible with the VEE Port (and not with the Image Renderer). This is very useful when a VEE Port features a foundation library that can use other kinds of images than the MicroUI library. The binary file can be encoded according to the user's options in the images list file.

1. Open image generator extension project.
2. Create an implementation of the interface `com.microej.tool.ui.generator.ImageConverter`.
3. Create the file `META-INF/services/com.microej.tool.ui.generator.ImageConverter` and open it.
4. Note the name of the created class, with its package and class name.
5. Rebuild the image generator extension, copy it into the VEE Port configuration project, and rebuild the VEE Port (see above).

The binary file can have two kinds of formats (see the API `OutputFileType getType()`):

- A simple resource: the binary output file is embedded as a resource; the application (or the library) can retrieve the file by using an API like `getResourceAsStream()`.
- An immutable file: the output file contains one or several *immutable objects*; the application (or the library) can retrieve the objects by using the *Beyond Profile (BON)* library.

Configuration File

The Image Generator uses a configuration file (also called the "list file") for describing images that need to be processed. The list file is a text file in which each line describes an image to convert. The image is described as a resource path, and should be available from the application classpath.

Note: The list file must be specified in the application launcher (see *Standalone Application Options*). However, all the files in the application classpath with suffix `.images.list` are automatically parsed by the Image Generator tool.

Each line can add optional parameters (separated by a ':') which define and/or describe the output file format (raw format). When no option is specified, the image is not converted and embedded as well.

Note: See *Configuration File* to understand the list file grammar.

- MicroEJ standard output format: to encode the image in a standard MicroEJ format, specify the MicroEJ format:

Listing 33: Standard Output Format Examples

```
image1:ARGB8888
image2:RGB565
image3:A4
```

- MicroEJ "Display" output format: to encode the image in the same format as the display (generic display or custom display, see *Pixel Structure*), specify `display` as output format:

Listing 34: Display Output Format Example

```
image1:display
```

- MicroEJ “GPU” output format: this format declaration is identical to standard format. It is a format that is also supported by the GPU.

Listing 35: GPU Output Format Examples

```
image1:ARGB8888
image2:RGB565
image3:A4
```

- MicroEJ ARGB1565_RLE output format (formerly RLE1): to encode the image in ARGB1565_RLE format, specify `ARGB1565_RLE` as output format:

Listing 36: ARGB1565_RLE Output Format Example

```
image1:ARGB1565_RLE
image1:RLE1 # Deprecated
```

- Without Compression: to keep original file, do not specify any format:

Listing 37: Unchanged Image Example

```
image1
```

- Binary format: to encode the image in a format only known by the VEE Port, refer to the VEE Port documentation to know which format are available.

Listing 38: Binary Output Format Example

```
image1:XXX
```

Link

Architecture 8

Architecture 7

Each image listed in an `.images.list` file (generated or not) is embedded by the application like a *resource*. An image is aligned in memory on the value specified by the Display module property `imageBuffer.memoryAlignment` (32 bits by default).

In addition to images binary files, the Image Generator module generates a linker file (`*.lscf`). This linker file declares an image section called `.rodata.images`. This section follows the next rules:

- The files are always listed in same order between two application builds.
- The section is aligned on the value specified by the Display module property `imageBuffer.memoryAlignment` (32 bits by default).
- Each file is aligned on section alignment value.

External Resources

The Image Generator manages two configuration files when the External Resources Loader is enabled. The first configuration file lists the images which will be stored as internal resources within the MicroEJ Application. The second file lists the images the Image Generator must convert and store in the External Resource Loader output directory. It is the BSP's responsibility to load the converted images into an external memory.

- Refer to the chapter *Images* to have more details how to use this kind of resources.
- Refer to the chapter *External Resource* to have more details how the Image Engine manages this kind of resources.

Installation

The Image Generator is an additional module for the MicroUI library. When the MicroUI module is installed, also install this module in order to be able to target pre-generated images.

In the VEE Port configuration file, check UI > Image Generator to install the Image Generator module. When checked, the properties file imageGenerator/imageGenerator.properties is required to specify the Image Generator extension project. When no extension is required (standalone mode only), this property is useless.

Use

The MicroUI Image APIs are available in the class ej.microui.display.Image ant its subclasses. There are no specific APIs that use a pre-generated image. When an image has been pre-processed, the MicroUI Image APIs getImage and loadImage will get/load the images.

Refer to the chapter *Standalone Application Options* (Libraries > MicroUI > Image) for more information about specifying the image configuration file.

Image Loader

Principle

The Image Loader is a module of the MicroUI runtime that:

- retrieves image data that is ready to be displayed without needing additional runtime memory,
- retrieves image data that is required to be converted into the format known by the Image Renderer (MicroEJ format),
- retrieves image in external memories (*External Resource* loader),
- converts images in MicroEJ format,
- creates a runtime buffer to manage MicroUI *Buffered Image*,
- manages dynamic images life cycle.

Note: The Image Loader is managing images to be compatible with Image Renderer. It does manage image in custom format (see *Binary Format*)

Functional Description

1. The application is using one of three ways to create a MicroUI Image object.
2. The Image Loader creates the image according to the MicroUI API, image location, image input format and image output format to be compatible with Image Renderer.
3. When the application closes the image, the Image Loader frees the RAM memory.

Images Heap

There are several ways to create a MicroUI Image. Except few specific cases, the Image Loader requires some RAM memory to store the image content in MicroEJ format. This format requires a small header as explained here: *MicroEJ Format: Standard*. It can be GPU compatible as explained here: *GPU Format Support*.

The heap size is application dependant. In the application launcher, set its size in `Libraries` > `MicroUI` > `Images heap size (in bytes)` . It will declare a section whose name is `.bss.microui.display.imagesHeap` .

By default, the Image Loader uses an internal best fit allocator to allocate the image buffers (internal Graphics Engine's allocator). Some specific *Abstraction Layer API* (LLAPI) are available to override this default implementation. These LLAPIs may be helpful to control the buffers allocation, retrieve the remaining space, etc. When not implemented by the BSP, the default internal Graphics Engine's allocator is used.

External Resource

Principle

An image is retrieved by its path (except for BufferedImage). The path describes a location in the application classpath. The resource may be generated at the same time as the application (internal resource) or be external (external resource). The Image Loader can load some images located outside the CPU addresses' space range. It uses the External Resource Loader.

When an image is located in such memory, the Image Loader copies it into RAM (into the CPU addresses' space range). Then it considers the image as an internal resource: it can continue to load the image (see following chapters). The RAM section used to load the external image is automatically freed when the Image Loader does not need it again.

The image may be located in external memory but be available in CPU addresses' space ranges (byte-addressable). In this case, the Image Loader considers the image as *internal* and does not need to copy its content in RAM.

Configuration File

Like internal resources, the Image Generator uses a *configuration file* (also called the "list file") for describing images that need to be processed. The list file must be specified in the application launcher (see *Standalone Application Options*). However, all the files in the application classpath with the suffix `.externimages.list` (or `.imagesext.list`) are automatically parsed by the Image Generator tool.

Process

This chapter describes the steps to setup the loading of an external resource from the application:

1. Add the image to the application project resources (typically in the source folder `src/main/resources` and in the package `images`).
2. Create / open the configuration file (e.g. `application.externimages.list`).
3. Add the relative path of the image and its output format (e.g. `/images/myImage.png:RGB565` see *Images*).
4. Build the application: the Image Generator converts the image in RAW format in the external resources folder (`[application_output_folder]/externalResources`).
5. Deploy the external resources to the external memory (SDCard, flash, etc.) of the device.
6. (optional) Configure the *External Resources Loader* to load from this source.
7. Build the application and run it on the device.
8. The application loads the external resource using ResourceImage.loadImage(String).
9. The image loader looks for the image and copies it in the *images heap* (no copy if the external memory is byte-addressable).
10. (optional) The image may be decoded (for instance: PNG), and the source image is removed from the images heap.
11. The external resource is immediately closed: the image's bytes have been copied in the images heap, or the image's bytes are always available (byte-addressable memory).
12. The application can use the image.
13. The application closes the image: the image is removed from the image heap.

Simulation

The Simulator automatically manages the external resources like internal resources. All images listed in `*.externimages.list` files are copied in the external resources folder, and this folder is added to the Simulator's classpath.

Image in MicroEJ Format

An image may be pre-processed (*Image Generator*) and so already in the format compatible with Image Renderer: MicroEJ format.

- When application is loading an image which is in such format and without specifying another output format, the Image Loader has just to make a link between the MicroUI Image object and the resource location. No more runtime decoder or converter is required, and so no more RAM memory.
- When application specifies another output format than MicroEJ format encoded in the image, Image Loader has to allocate a buffer in RAM. It will convert the image in the expected MicroEJ format.
- When application is loading an image in MicroEJ format stored as *External Resource*, the Image Loader has to copy the image into RAM memory to be usable by Image Renderer.

Encoded Image

Internal Decoders

An image can be encoded (PNG, JPEG, etc.). In this case Image Loader asks to its Image Decoders module if a decoder is able to decode the image. The source image is not copied in RAM (expect for images stored as *External Resource*). Image Decoder allocates the decoded image buffer in RAM first and then inflates the image. The image is encoded in MicroEJ format specified by the application, when specified. When not specified, the image in encoded in the default MicroEJ format specified by the Image Decoder itself.

The UI extension provides two internal Image Decoders modules:

- PNG Decoder: a full PNG decoder that implements the PNG format (https://www.w3.org/Graphics/PNG). Regular, interlaced, indexed (palette) compressions are handled.
- BMP Monochrome Decoder: .bmp format files that embed only 1 bit per pixel can be decoded by this decoder.

Additional Decoders

Some additional decoders can be added. Implement the function LLUI_DISPLAY_IMPL_decodeImage to add a new decoder. The implementation must respect the following rules:

- Fills the MICROUI_Image structure with the image characteristics: width, height and format.

Note: The output image format might be different than the expected format (given as argument). In this way, the Display module will perform a conversion after the decoding step. During this conversion, an out of memory error can occur because the final RAW image cannot be allocated.

- Allocates the RAW image data calling the function LLUI_DISPLAY_allocateImageBuffer . This function will allocates the RAW image data space in the display working buffer according to the RAW image format and size.
- Decodes the image in the allocated buffer.

- Waiting the end of decoding step before returning.

Abstraction Layer API

The internal and external decoders are called by the Graphics Engine when the application invokes the MicroUI API ej.microui.display.ResourceImage.loadImage(). They can also be explicitly called by any C task via the supplementary API `LLUI_DISPLAY_decodeImage()`.

Important: If the Abstraction Layer API `LLUI_DISPLAY_decodeImage()` is called outside of the Core Engine task (i.e., outside of a native Java context), synchronization functions with the Graphics Engine must be implemented (see *Graphics Engine API*).

Like MicroUI, this API provides an instruction on the output format (see *Image Format*). This instruction may not be followed by the decoder, depending on the encoding capabilities of the decoder and the Graphics Engine. The default format of the decoder is then selected instead (depends on the decoder itself).

Note: The caller can verify the RAW encoding by checking the `format` field of the `MICROUI_Image`.

Unlike the MicroUI API, which is limited to six output formats, the abstraction layer API allows for additional or compatible formats. Some formats may not be available at runtime (it is recommended to use the *Image Generator* instead). The table below indicates whether the format instruction is adhered to, replaced by a compatible format, or replaced by the default format of the decoder.

Instruction	Output Format
MICROUI_IMAGE_FORMAT_DISPLAY	*default format*
MICROUI_IMAGE_FORMAT_ARGB8888	MICROUI_IMAGE_FORMAT_ARGB8888
MICROUI_IMAGE_FORMAT_ARGB1555	MICROUI_IMAGE_FORMAT_ARGB1555
MICROUI_IMAGE_FORMAT_ARGB4444	MICROUI_IMAGE_FORMAT_ARGB4444
MICROUI_IMAGE_FORMAT_ARGB8888_PRE	MICROUI_IMAGE_FORMAT_ARGB8888_PRE
MICROUI_IMAGE_FORMAT_ARGB1555_PRE	MICROUI_IMAGE_FORMAT_ARGB1555_PRE
MICROUI_IMAGE_FORMAT_ARGB4444_PRE	MICROUI_IMAGE_FORMAT_ARGB4444_PRE
MICROUI_IMAGE_FORMAT_RGB888	MICROUI_IMAGE_FORMAT_RGB888
MICROUI_IMAGE_FORMAT_RGB565	MICROUI_IMAGE_FORMAT_RGB565
MICROUI_IMAGE_FORMAT_A1	MICROUI_IMAGE_FORMAT_A8
MICROUI_IMAGE_FORMAT_A2	MICROUI_IMAGE_FORMAT_A8
MICROUI_IMAGE_FORMAT_A4	MICROUI_IMAGE_FORMAT_A8
MICROUI_IMAGE_FORMAT_A8	MICROUI_IMAGE_FORMAT_A8
MICROUI_IMAGE_FORMAT_C1	*default format*
MICROUI_IMAGE_FORMAT_C2	*default format*
MICROUI_IMAGE_FORMAT_C4	*default format*
MICROUI_IMAGE_FORMAT_AC11	*default format*
MICROUI_IMAGE_FORMAT_AC22	*default format*
MICROUI_IMAGE_FORMAT_AC44	*default format*
MICROUI_IMAGE_FORMAT_LARGB8888	*default format*
MICROUI_IMAGE_FORMAT_UNDEFINED	*default format*
MICROUI_IMAGE_FORMAT_CUSTOM_7	*default format*
MICROUI_IMAGE_FORMAT_CUSTOM_6	*default format*
MICROUI_IMAGE_FORMAT_CUSTOM_5	*default format*
MICROUI_IMAGE_FORMAT_CUSTOM_4	*default format*
MICROUI_IMAGE_FORMAT_CUSTOM_3	*default format*
MICROUI_IMAGE_FORMAT_CUSTOM_2	*default format*
MICROUI_IMAGE_FORMAT_CUSTOM_1	*default format*
MICROUI_IMAGE_FORMAT_CUSTOM_0	*default format*

Installation

The Image Decoders modules are some additional modules to the Display module. The decoders belong to distinct modules, and either or several may be installed.

In the VEE Port configuration file, check UI > Image PNG Decoder to install the runtime PNG decoder. Check UI > Image BMP Monochrome Decoder to install the runtime BMP monochrom decoder.

Use

The MicroUI Image APIs are available in the class ej.microui.display.Image. There is no specific API that uses a runtime image. When an image has not been pre-processed (see *Image Generator*), the MicroUI Image APIs createImage* will load this image.

Image Renderer

Principle

The Image Renderer is a module of the MicroUI runtime that reads and draws the images (see *Image Format*). It calls Abstraction Layer APIs to draw and transform the images (rotation, scaling, deformation, etc.). It also includes software algorithms to perform the rendering.

Functional Description

All MicroUI image drawings are redirected to a set of Abstraction Layer APIs. All Abstraction Layer APIs are implemented by weak functions, which call software algorithms. The BSP can override this default behavior for each Abstraction Layer API independently. Furthermore, the BSP can override an Abstraction Layer API for a specific MicroEJ format (for instance ARGB8888) and call the software algorithms for all other formats.

Destination Format

Since MicroUI 3.2, the destination buffer of the drawings can be different from the display back buffer format (see *MicroEJ Format: Display*). This destination buffer format can be a *standard format* (ARGB8888, A8, etc.) or a *custom format*.

See *Buffered Image* for more information about how to create buffered images with another format than the display format and how to draw in them.

Input Formats

Standard

The Image Renderer is by default able to draw all *standard formats*. No extra support in the VEE Port is required to draw this kind of image.

The image drawing resembles a *shape drawing*. The drawing is performed by default by the *Graphics Engine Software Algorithms* and can be overridden to use a third-party library or a GPU.

Custom

A *MicroEJ Format: Custom* image can be:

- an image with a pixel buffer but whose pixel organization is not standard,
- an image with a data buffer: an image encoded with a third-party encoder (proprietary format or not),
- an image with a “command” buffer: instead of performing the drawings on pixels, the image *stores* the drawing actions to replay them later,
- etc.

The VEE Port must extend the Image Renderer to support the drawing of these images. This extension can consist in:

- decoding the image at runtime to draw it,

- using a compatible GPU to draw it,
- using a command interpreter to perform some *shape drawings*,
- etc.

To draw the custom images, the Image Renderer introduces the notion of *custom image drawer*. This drawer is an engine that has the responsibility to draw the image. Each custom image format (0 to 7) has its own image drawer.

Each drawing of a custom image is redirected to the associated image drawer.

Hint: A custom image drawer can call the UI Shapes Drawing API to draw its elements in the destination.

The implementation is not the same between the Embedded side and the Simulation. However, the concepts are the same and are described in dedicated chapters.

MicroUI C Module

Principle

As described above, an *image drawer* allows drawing the images whose format is *custom*. The *MicroUI C module* is designed to manage the notion of drawers: it does not *support* the custom formats but allows adding some additional drawers.

This support uses several weak functions and tables to redirect the image drawings. When custom drawers are not used (when the VEE Port does not need to support *custom* images), this support can be removed to reduce the memory footprint (by removing the indirection tables) and improve the performances (by reducing the number of runtime function calls).

Standard Formats Only (Default)

The default implementation can only draw images with a *standard format*. In other words, the application cannot draw a custom image. This is the most frequent use case, the only one available with MicroUI before version 3.2.

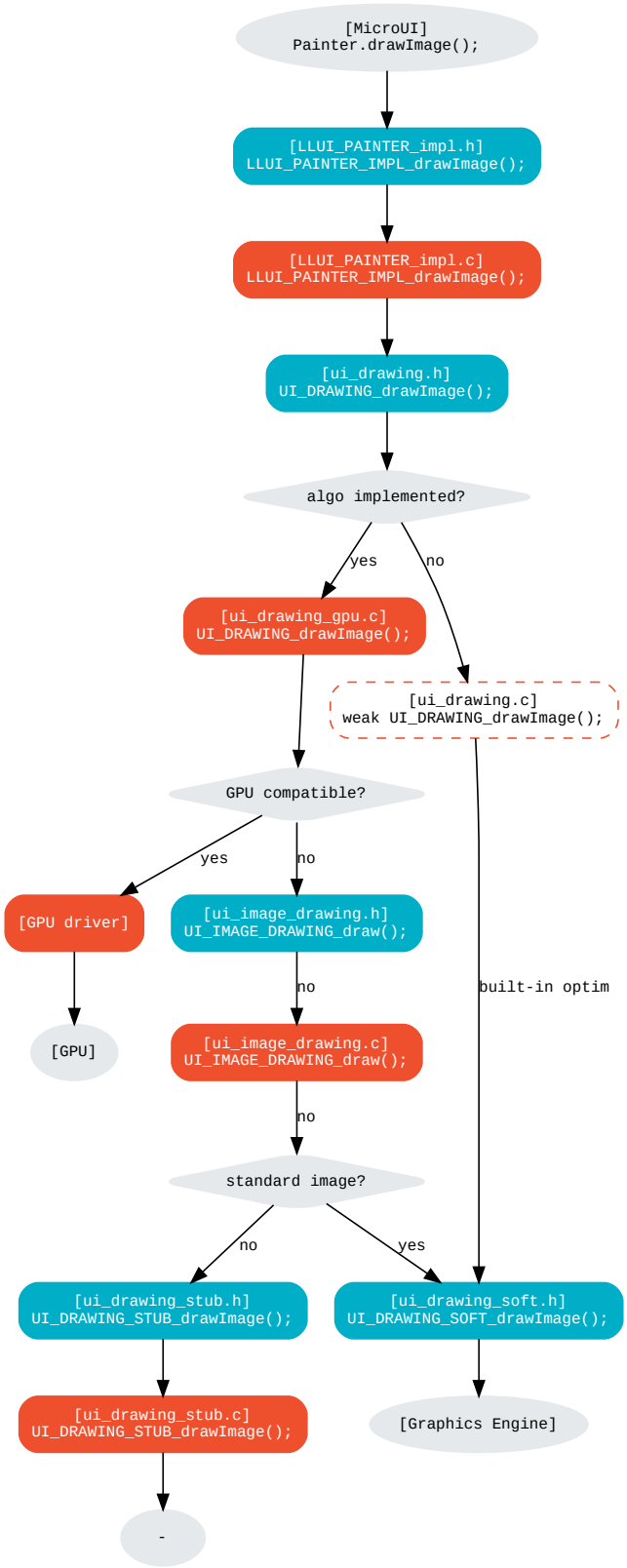
Attention: To select this implementation (to disable the custom format support), the define UI_FEATURE_IMAGE_CUSTOM_FORMATS must be unset.

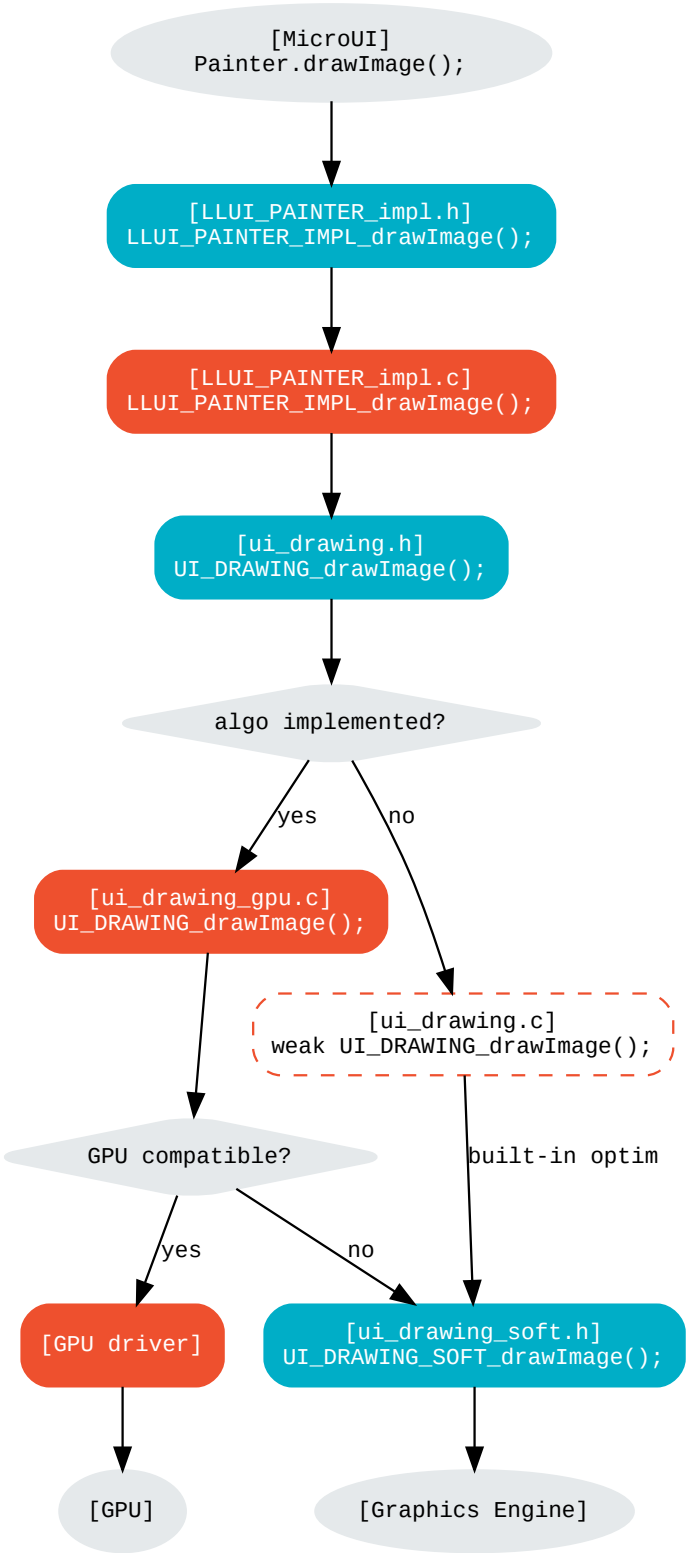
The image drawing is similar to UI_DRAWING_GPU_drawLine (see *MicroUI C Module*), but, theoretically, it should let the image drawer handle the image instead of calling the software drawer directly. However the MicroUI C Module (and the extended MicroUI modules that handle a GPU) takes advantage of the define UI_FEATURE_IMAGE_CUSTOM_FORMATS : as it is not set, the C Modules bypass the indirection to the image drawer, and as a consequence, the implementation of the weak function only consists in calling the Graphics Engine's software algorithm. This tip reduces the footprint and the CPU usage.

An implementation of a third-party GPU can optionally take advantage of the define UI_FEATURE_IMAGE_CUSTOM_FORMATS . The following diagrams illustrate the drawing of an image with or without taking advantage of the define UI_FEATURE_IMAGE_CUSTOM_FORMATS (respectively *default* and *optimized* implementation).

Default Implementation

Optimized Implementation


[MicroUI]
Painter.drawImage();
[LLUI_PAINTER_impl.h]
LLUI_PAINTER_IMPL_drawImage();
[LLUI_PAINTER_impl.c]
LLUI_PAINTER_IMPL_drawImage();
[ui_drawing.h]
UI_DRAWING_drawImage();
algo implemented?
yes
no
[ui_drawing_gpu.c]
UI_DRAWING_drawImage();
[ui_drawing.c]
weak UI_DRAWING_drawImage();
GPU compatible?
yes
no
[GPU driver]
[ui_image_drawing.h]
UI_IMAGE_DRAWING_draw();
built-in optim
[GPU]
no
[ui_image_drawing.c]
UI_IMAGE_DRAWING_draw();
no
standard image?
no
yes
[ui_drawing_stub.h]
UI_DRAWING_STUB_drawImage();
[ui_drawing_soft.h]
UI_DRAWING_SOFT_drawImage();
[ui_drawing_stub.c]
UI_DRAWING_STUB_drawImage();
[Graphics Engine]
-


[MicroUI]
Painter.drawImage();
[LLUI_PAINTER_impl.h]
LLUI_PAINTER_IMPL_drawImage();
[LLUI_PAINTER_impl.c]
LLUI_PAINTER_IMPL_drawImage();
[ui_drawing.h]
UI_DRAWING_drawImage();
algo implemented?
yes
no
[ui_drawing_gpu.c]
UI_DRAWING_drawImage();
[ui_drawing.c]
weak UI_DRAWING_drawImage();
GPU compatible?
built-in optim
yes
no
[GPU driver]
[ui_drawing_soft.h]
UI_DRAWING_SOFT_drawImage();
[GPU]
[Graphics Engine]

LLUI_PAINTER_IMPL_drawImage (available in MicroUI C Module)

Similar to LLUI_PAINTER_IMPL_drawLine , see *MicroUI C Module*.

UI_DRAWING_drawImage

```
// Available in MicroUI C Module
#define UI_DRAWING_DEFAULT_drawImage UI_DRAWING_drawImage

// To write in the BSP (optional)
#define UI_DRAWING_GPU_drawImage UI_DRAWING_drawImage
```

The function names are set with preprocessor macros. These name redirections are helpful when the VEE Port features more than one destination format (which is not the case here).

UI_DRAWING_GPU_drawImage (to write in the BSP)

Similar to UI_DRAWING_GPU_drawLine (see *MicroUI C Module*), but lets the image drawer manage the image instead of calling the software drawer directly (*Default Implementation*) or takes advantage of the define UI_FEATURE_IMAGE_CUSTOM_FORMATS (*Optimized Implementation*):

Default Implementation

Optimized Implementation

```
// Unlike the MicroUI C Module, this function is not "weak".
DRAWING_
↪Status UI_DRAWING_GPU_drawImage(MICROUI_GraphicsContext* gc, MICROUI_
↪Image* img, jint regionX, jint regionY, jint width, jint height, jint x, jint y, jint alpha) {

  DRAWING_Status status;

  if (is_gpu_compatible(xxx)) {

    // See chapter "Drawings"
    // [...]
  }
  else {
    // Let the image drawer manages the image (available in the C module)
    status␣
↪= UI_IMAGE_DRAWING_draw(gc, img, regionX, regionY, width, height, x, y, alpha);
  }
  return status;
}
```

```
// Unlike the MicroUI C Module, this function is not "weak".
DRAWING_
↪Status UI_DRAWING_GPU_drawImage(MICROUI_GraphicsContext* gc, MICROUI_
↪Image* img, jint regionX, jint regionY, jint width, jint height, jint x, jint y, jint alpha) {

  DRAWING_Status status;

  if (is_gpu_compatible(xxx)) {

    // See chapter "Drawings"
    // [...]
  }
  else {
```

(continues on next page)

(continued from previous page)

```
#if !defined(UI_FEATURE_IMAGE_CUSTOM_FORMATS)
    status␣
↪= UI_DRAWING_SOFT_drawImage(gc, img, regionX, regionY, width, height, x, y, alpha);
#else
    // Let the image drawer manages the image (available in the C module)
    status␣
↪= UI_IMAGE_DRAWING_draw(gc, img, regionX, regionY, width, height, x, y, alpha);
#endif
  }
  return status;
}
```

UI_DRAWING_DEFAULT_drawImage (available in MicroUI C Module)

```
// Use the compiler's 'weak' attribute
__weak DRAWING_Status␣
↪UI_DRAWING_DEFAULT_drawImage(MICROUI_GraphicsContext* gc, MICROUI_
↪Image* img, jint regionX, jint regionY, jint width, jint height, jint x, jint y, jint alpha) {
#if !defined(UI_FEATURE_IMAGE_CUSTOM_FORMATS)
  return␣
↪UI_DRAWING_SOFT_drawImage(gc, img, regionX, regionY, width, height, x, y, alpha);
#else
  return UI_IMAGE_DRAWING_draw(gc, img, regionX, regionY, width, height, x, y, alpha);
#endif
}
```

The define UI_FEATURE_IMAGE_CUSTOM_FORMATS is not set, so the implementation of the weak function only consists in calling the Graphics Engine's software algorithm.

Custom Format Support

In addition to the *standard formats*, this implementation allows drawing images with a *custom format*. This advanced use case is available only with MicroUI 3.2 or higher.

> **Attention:** To select this implementation, the define UI_FEATURE_IMAGE_CUSTOM_FORMATS must be set (no specific value).

The MicroUI C module uses some tables to redirect the image management to the expected extension. There is one table per Image Abstraction Layer API (draw, copy, region, rotate, scale, flip) to embed only the necessary algorithms (a table and its functions are only embedded in the final binary file if and only if the MicroUI drawing method is called).

Each table contains ten elements:

```
static const UI_IMAGE_DRAWING_draw_t UI_IMAGE_DRAWING_draw_custom[] = {
    &UI_DRAWING_STUB_drawImage,
    &UI_DRAWING_SOFT_drawImage,
    &UI_IMAGE_DRAWING_draw_custom0,
    &UI_IMAGE_DRAWING_draw_custom1,
    &UI_IMAGE_DRAWING_draw_custom2,
    &UI_IMAGE_DRAWING_draw_custom3,
    &UI_IMAGE_DRAWING_draw_custom4,
    &UI_IMAGE_DRAWING_draw_custom5,
```

(continues on next page)

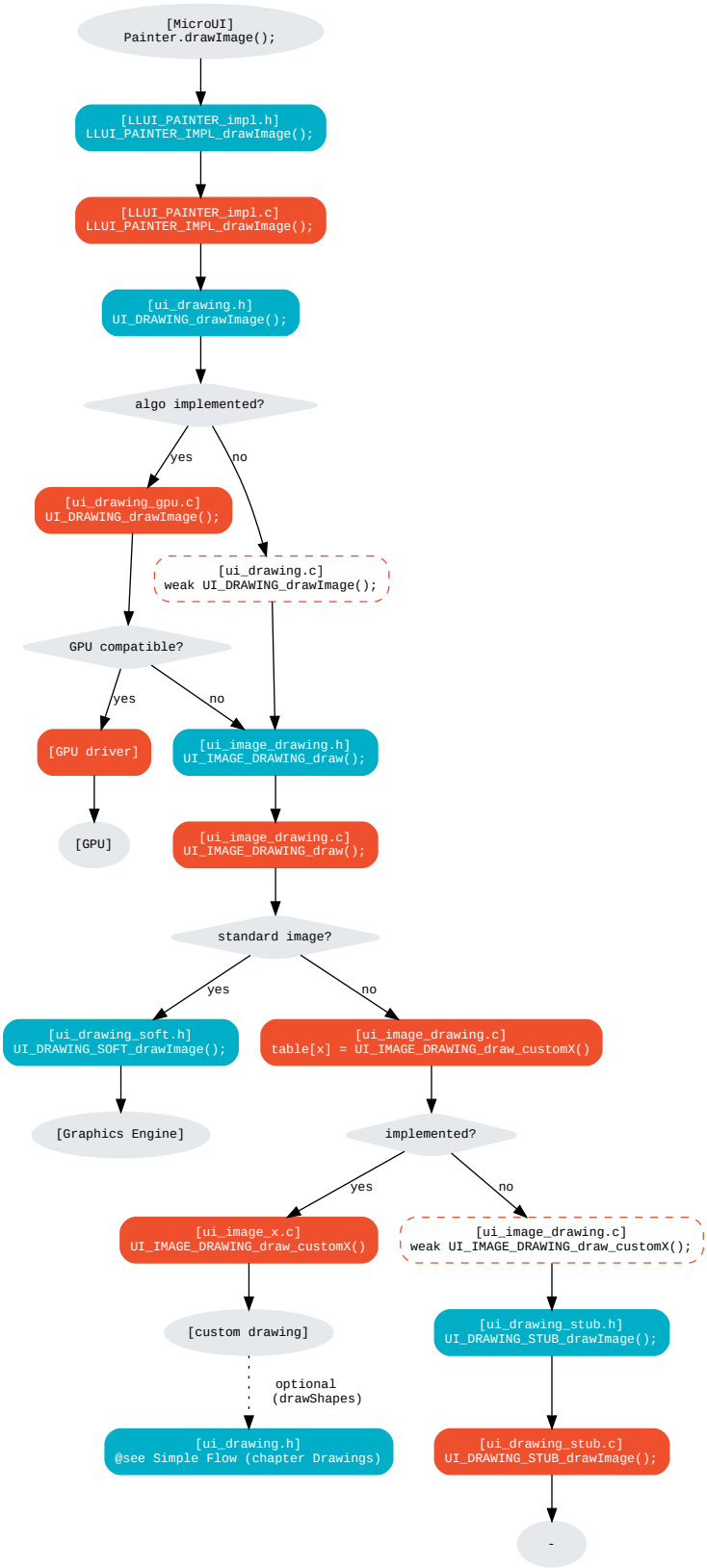
(continued from previous page)

```
    &UI_IMAGE_DRAWING_draw_custom6,
    &UI_IMAGE_DRAWING_draw_custom7,
};
```

- UI_DRAWING_STUB_drawImage is the drawing function called when the drawing function is not implemented,
- UI_DRAWING_SOFT_drawImage is the drawing function that redirects the drawing to the *Graphics Engine Software Algorithms*,
- UI_IMAGE_DRAWING_draw_customX (0 to 7) are the drawing functions for each custom format.

The MicroUI C Module retrieves the table index according to the image format.

The following diagram illustrates the drawing of an image:


[MicroUI]
Painter.drawImage();
[LLUI_PAINTER_impl.h]
LLUI_PAINTER_IMPL_drawImage();
[LLUI_PAINTER_impl.c]
LLUI_PAINTER_IMPL_drawImage();
[ui_drawing.h]
UI_DRAWING_drawImage();
algo implemented?
yes
no
[ui_drawing_gpu.c]
UI_DRAWING_drawImage();
[ui_drawing.c]
weak UI_DRAWING_drawImage();
GPU compatible?
yes
no
[GPU driver]
[ui_image_drawing.h]
UI_IMAGE_DRAWING_draw();
[GPU]
[ui_image_drawing.c]
UI_IMAGE_DRAWING_draw();
standard image?
yes
no
[ui_drawing_soft.h]
UI_DRAWING_SOFT_drawImage();
[ui_image_drawing.c]
table[x] = UI_IMAGE_DRAWING_draw_customX()
[Graphics Engine]
implemented?
yes
no
[ui_image_x.c]
UI_IMAGE_DRAWING_draw_customX()
[ui_image_drawing.c]
weak UI_IMAGE_DRAWING_draw_customX();
[custom drawing]
[ui_drawing_stub.h]
UI_DRAWING_STUB_drawImage();
optional
(drawShapes)
[ui_drawing.h]
@see Simple Flow (chapter Drawings)
[ui_drawing_stub.c]
UI_DRAWING_STUB_drawImage();
-

Take the same example as the *Standard Formats Only* implementation (draw an image):

UI_DRAWING_DEFAULT_drawImage (available in MicroUI C Module)

```
// use the compiler's 'weak' attribute
__weak DRAWING_Status␣
↪UI_DRAWING_DEFAULT_drawImage(MICROUI_GraphicsContext* gc, MICROUI_
↪Image* img, jint regionX, jint regionY, jint width, jint height, jint x, jint y, jint alpha) {
#if !defined(UI_FEATURE_IMAGE_CUSTOM_FORMATS)
  return␣
↪UI_DRAWING_SOFT_drawImage(gc, img, regionX, regionY, width, height, x, y, alpha);
#else
  return UI_IMAGE_DRAWING_draw(gc, img, regionX, regionY, width, height, x, y, alpha);
#endif
}
```

The define `UI_FEATURE_IMAGE_CUSTOM_FORMATS` is set so the implementation of the weak function redirects the image drawing to the image drawer manager (`ui_image_drawing.h`).

UI_IMAGE_DRAWING_draw (available in MicroUI C Module)

```
static const UI_IMAGE_DRAWING_draw_t UI_IMAGE_DRAWING_draw_custom[] = {
  &UI_DRAWING_STUB_drawImage,
  &UI_DRAWING_SOFT_drawImage,
  &UI_IMAGE_DRAWING_draw_custom0,
  &UI_IMAGE_DRAWING_draw_custom1,
  &UI_IMAGE_DRAWING_draw_custom2,
  &UI_IMAGE_DRAWING_draw_custom3,
  &UI_IMAGE_DRAWING_draw_custom4,
  &UI_IMAGE_DRAWING_draw_custom5,
  &UI_IMAGE_DRAWING_draw_custom6,
  &UI_IMAGE_DRAWING_draw_custom7,
};

DRAWING_
↪Status UI_IMAGE_DRAWING_draw(MICROUI_GraphicsContext* gc, MICROUI_
↪Image* img, jint regionX, jint regionY, jint width, jint height, jint x, jint y, jint alpha){
  return (*UI_IMAGE_DRAWING_draw_
↪custom[__get_table_index(gc, img)])(gc, img, regionX, regionY, width, height, x, y, alpha);
}
```

The implementation in the MicroUI C module redirects the drawing to the expected drawer. The drawer is retrieved using its format (function `__get_table_index()`):

- the format is standard but the destination is not the *display* format: index `0` is returned,
- the format is standard and the destination is the *display* format: index `1` is returned,
- the format is custom: an index from `2` to `9` is returned.

UI_IMAGE_DRAWING_draw_custom0 (available in MicroUI C Module)

```
// Use the compiler's 'weak' attribute
__weak DRAWING_Status␣
↪UI_IMAGE_DRAWING_draw_custom0(MICROUI_GraphicsContext* gc, MICROUI_
```

(continues on next page)

(continued from previous page)

```
↪Image* img, jint regionX, jint regionY, jint width, jint height, jint x, jint y, jint alpha){
  return␣
↪UI_DRAWING_STUB_drawImage(gc, img, regionX, regionY, width, height, x, y, alpha);
}
```

The default implementation of UI_IMAGE_DRAWING_draw_custom0 (same behavior for 0 to 7) consists in calling the stub implementation.

UI_DRAWING_STUB_drawImage (available in MicroUI C Module)

```
DRAWING_
↪Status UI_DRAWING_STUB_drawImage(MICROUI_GraphicsContext* gc, MICROUI_
↪Image* img, jint regionX, jint regionY, jint width, jint height, jint x, jint y, jint alpha){
  // Set the drawing log flag "not implemented"
  LLUI_DISPLAY_reportError(gc, DRAWING_LOG_NOT_IMPLEMENTED);
  return DRAWING_DONE;
}
```

The implementation only consists in setting the *Drawing log flag* DRAWING_LOG_NOT_IMPLEMENTED to notify the application that the drawing has not been performed.

Simulation

Principle

The simulation behavior is similar to the *MicroUI C Module* for the Embedded side.

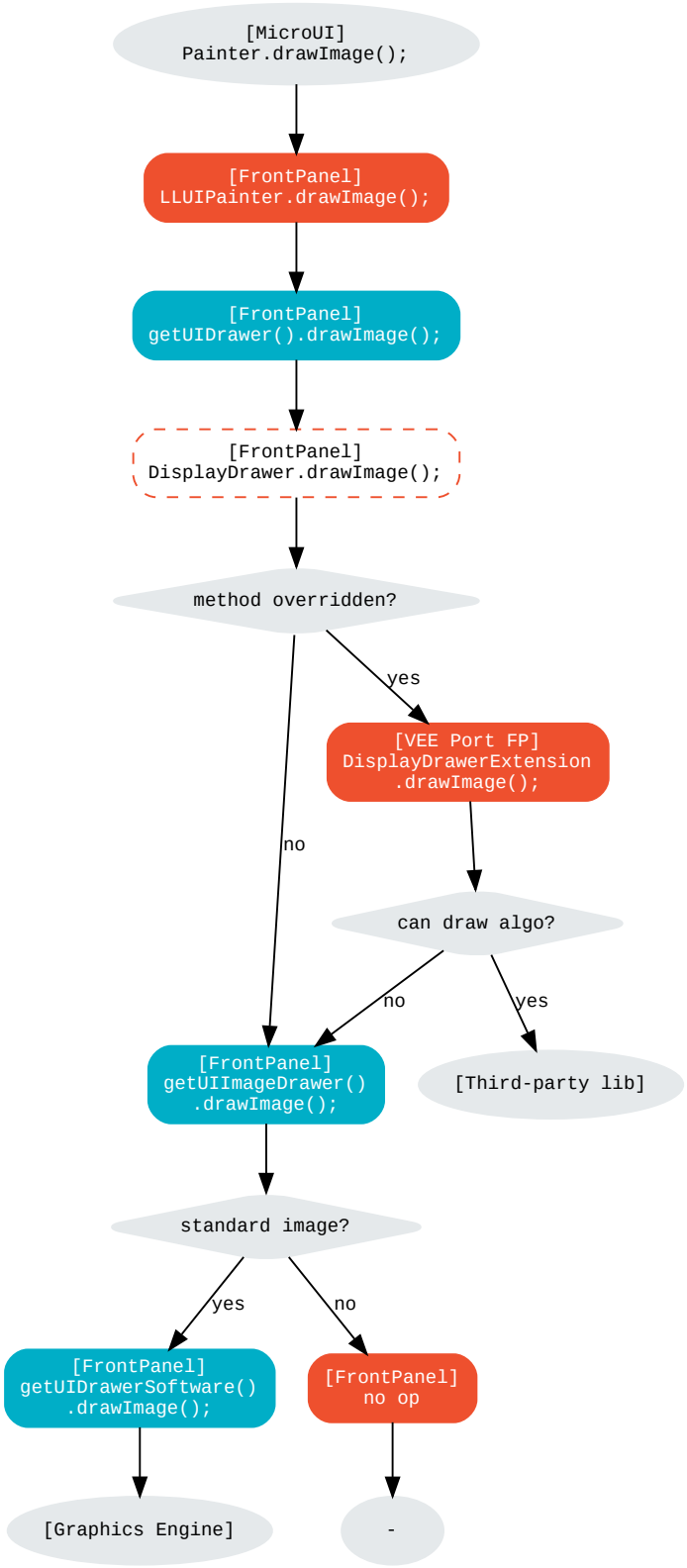
The *Front Panel* defines support for the drawers based on the Java service loader.

Standard Formats Only (Default)

The default implementation can draw images with a standard format.

Note: Contrary to the *MicroUI C Module*, the simulation does not (and doesn't need to) provide an option to disable the use of custom image.

The following diagram illustrates the drawing of an image:


[MicroUI]
Painter.drawImage();
[FrontPanel]
LLUIPainter.drawImage();
[FrontPanel]
getUIDrawer().drawImage();
[FrontPanel]
DisplayDrawer.drawImage();
method overridden?
yes
no
[VEE Port FP]
DisplayDrawerExtension
.drawImage();
can draw algo?
no
yes
[FrontPanel]
getUIImageDrawer()
.drawImage();
[Third-party lib]
standard image?
yes
no
[FrontPanel]
getUIDrawerSoftware()
.drawImage();
[FrontPanel]
no op
[Graphics Engine]
-

It is possible to override the image drawers for the standard format in the same way as the custom formats.

Custom Format Support

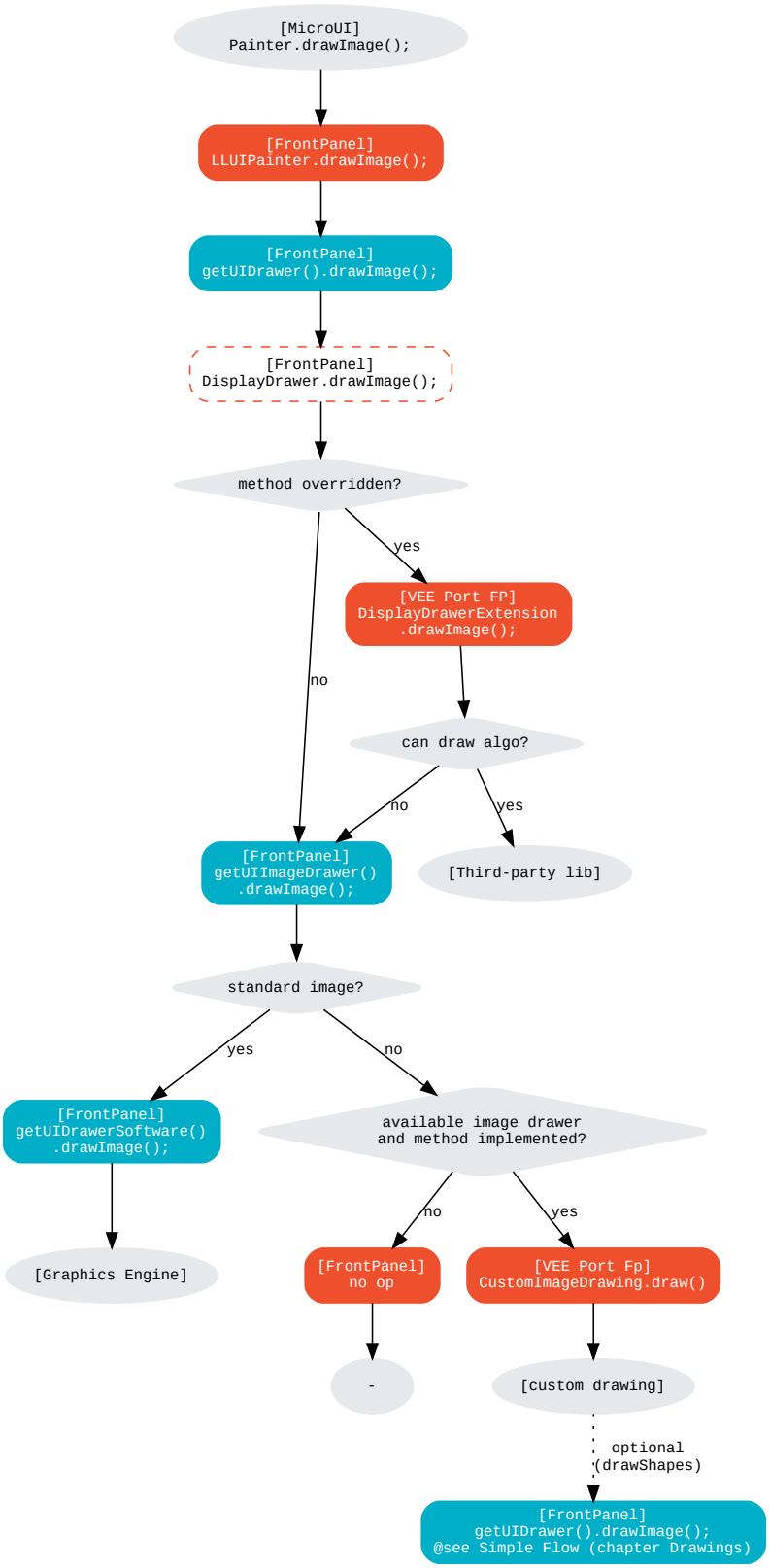
It is possible to draw images with a custom format by implementing the UIImageDrawing interface. This advanced use case is available only with MicroUI 3.2 or higher.

The UIImageDrawing interface contains one method for each image drawing primitive (draw, copy, region, rotate, scale, flip). Only the necessary methods have to be implemented. Each non-implemented method will result in calling the stub implementation.

The method handledFormat() must be implemented and returns the image format handled by the drawer.

Once created, the UIImageDrawing implementation must be registered as a service.

The following diagram illustrates the drawing of an image:


[MicroUI]
Painter.drawImage();
[FrontPanel]
LLUIPainter.drawImage();
[FrontPanel]
getUIDrawer().drawImage();
[FrontPanel]
DisplayDrawer.drawImage();
method overridden?
yes
no
[VEE Port FP]
DisplayDrawerExtension
.drawImage();
can draw algo?
no
yes
[FrontPanel]
getUIImageDrawer()
.drawImage();
[Third-party lib]
standard image?
yes
no
[FrontPanel]
getUIDrawerSoftware()
.drawImage();
available image drawer
and method implemented?
[Graphics Engine]
no
yes
[FrontPanel]
no op
[VEE Port Fp]
CustomImageDrawing.draw()
-
[custom drawing]
optional
(drawShapes)
[FrontPanel]
getUIDrawer().drawImage();
@see Simple Flow (chapter Drawings)

Let's implement the image drawer for the *CUSTOM_0* format.

```
public class MyCustomImageDrawer implements UIImageDrawing {

   @Override
   public MicroUIImageFormat handledFormat() {
      return MicroUIImageFormat.MICROUI_IMAGE_FORMAT_CUSTOM_0;
   }

   @Override
   public void draw(MicroUIGraphicsContext␣
↪gc, MicroUIImage img, int regionX, int regionY, int width, int height,
         int x, int y, int alpha) {
      MyCustomImage customImage = (MyCustomImage) img.getImage().getRAWImage();
      customImage.drawOn(gc, regionX, regionY, width, height, x, y, alpha);
   }

}
```

Now, this drawer needs to be registered as a service. This can be achieved by creating a file in the resources of the Front Panel project named `META-INF/services/ej.microui.display.UIImageDrawing` and containing the fully qualified name of the previously created image drawer.

```
com.mycompany.MyCustomImageDrawer
```

It is also possible to declare it programmatically (see where a drawer is registered in the *drawing custom* section):

```
LLUIDisplay.Instance.registerUIImageDrawer(new MyCustomImageDrawer());
```

Image Pixel Conversion

Overview

The Graphics Engine is built for a dedicated display pixel format (see *Pixel Structure*). For this pixel format, the Graphics Engine must be able to draw images with or without alpha blending and with or without transformation. In addition, it must be able to read all image formats.

The application may not use all MicroUI image drawing options and may not use all images formats. It is not possible to detect what the application needs, so no optimization can be performed at application compiletime. However, for a given application, the VEE Port can be built with a reduced set of pixel support.

All pixel format manipulations (read, write, copy) are using dedicated functions. It is possible to remove some functions or to use generic functions. The advantage is to reduce the memory footprint. The inconvenient is that some features are removed (the application should not use them) or some features are slower (generic functions are slower than the dedicated functions).

Functions

There are five pixel *conversion* modes:

- Draw an image without transformation and without global alpha blending: copy a pixel from a format to the destination format (display format).
- Draw an image without transformation and with global alpha blending: copy a pixel with alpha blending from a format to the destination format (display format).
- Draw an image with transformation and with or without alpha blending: draw an ARGB8888 pixel in destination format (display format).
- Load a ResourceImage with an output format: convert an ARGB8888 pixel to the output format.
- Read a pixel from an image (Image.readPixel() or to draw an image with transformation or to convert an image): read any pixel format and convert it to ARGB8888.

Table 79: Pixel Conversion

	Nb input formats	Nb output formats	Number of combinations
Draw image without global alpha	22	1	22
Draw image with global alpha	22	1	22
Draw image with transformation	2	1	2
Load a ResourceImage	1	6	6
Read an image	22	1	22

There are $22x1 + 22x1 + 2x1 + 1x6 + 22x1 = 74$ functions. Each function takes between 50 and 200 bytes depending on its complexity and the C compiler.

Linker File

All pixel functions are listed in a VEE Port linker file. It is possible to edit this file to remove some features or to share some functions (using generic function).

How to get the file:

1. Build VEE Port as usual.
2. Copy VEE Port file [platform]/source/link/display_image_x.lscf in the VEE Port configuration project: [VEE Port configuration project]/dropins/link/. Where x is a number that characterizes the display pixel format (see *Pixel Structure*). See next warning.
3. Perform some changes into the copied file (see after).
4. Rebuild the VEE Port: the file in the *dropins* folder is copied in the VEE Port instead of the original one.

Warning: When the display format changes (see *Installation*), the linker file suffix changes too. Perform again all the operations in the new file with the new suffix.

The linker file holds five tables, one for each use case, respectively IMAGE_UTILS_TABLE_COPY, IMAGE_UTILS_TABLE_COPY_WITH_ALPHA, IMAGE_UTILS_TABLE_DRAW, IMAGE_UTILS_TABLE_SET and IMAGE_UTILS_TABLE_READ. For each table, a comment describes how to remove an option (when possible) or how to replace an option by a generic function (if available).

Installation

The Image Renderer module is part of the MicroUI module and Display module. Install them to be able to use some images.

Use

The MicroUI image APIs are available in the class ej.microui.display.Image.

Buffered Image

Overview

MicroUI application can create an image it can draw into: the MicroUI ej.microui.display.BufferedImage. The format of this kind of image is Display (default), Standard, or Custom (see following chapters).

> **Warning:** The output format Standard and Custom depends on the VEE Port capabilities.

To create this kind of image, the Image Loader has to create a buffer in the *images heap* whose size depends on the image data size (see *Image Creation*).

Drawer

A buffered image requires a *drawer*. A drawer is an engine that has the responsibility to:

- allow the application to create Standard and Custom buffered images,
- draw into these images.

The implementation is not the same between the Embedded side and the Simulation. However, the concepts are the same and are described in dedicated chapters.

Formats

Display

This is the format used by default when no format is specified when creating a MicroUI BufferedImage .

The image format is the same as the front buffer format; in other words, its number of bits-per-pixel and its pixel bits organization are the same (see chapter *MicroEJ Format: Display*).

- Image creation: the Graphics Engine provides the capacity to create this kind of image; no specific support is required in the VEE Port.
- Draw into the image: the rules to draw into this kind of buffered image are the same as in the display back buffer; see:ref:*section_drawings*.
- Draw the image: the rules to draw this kind of buffered image are described in the chapter *image renderer standard*.

Standard

A MicroUI BufferedImage can be created specifying a *MicroEJ Format: Standard* or *MicroEJ Format: Grayscale* format.

Note: When the display format is the same as the standard format used to create the buffered image, the rules to create the image, to draw into it and to draw it are the same as the Display format. This chapter describes the use case when the format differs from the *display* format.

Unlike the display format, the VEE Port must feature a *drawer* for each standard format.

- Image creation: the drawer allows the creation of this kind of buffered image; if the VEE Port does not feature a drawer for a specific format, the MicroUI BufferedImage cannot be created, and an exception is thrown at runtime.
- Draw into the image: the drawer can implement all MicroUI drawings or just a reduced set; when a drawing is not implemented, a stub implementation (that does nothing) is used.
- Draw the image: the image is *standard*, so its rendering is *standard* also; the rules to draw this kind of buffered image are described in the chapter *image renderer standard* (no extra support needed in the VEE Port).

Custom

A MicroUI BufferedImage can be created specifying a *MicroEJ Format: Custom* formats.

Like standard formats, the VEE Port must feature a *drawer* for each custom format. It must also feature an image allocator.

- Image creation: the allocator and drawer allow to create of this kind of buffered image; if the VEE Port does not feature an allocator and a drawer for a specific format, the MicroUI BufferedImage cannot be created, and an exception is thrown at runtime.
- Draw into the image: the drawer can implement all MicroUI drawings or just a reduced set; when a drawing is not implemented, a stub implementation (that does nothing) is used.
- Draw the image: the image is *custom*, so its rendering is *custom* also; the rules to draw this kind of buffered image are described in the chapter *image renderer custom*.

MicroUI C Module

Drawer

As described above, a *drawer* allows to create and draw into buffered images whose format differs from the display format. The *MicroUI C module* is designed to manage the notion of drawers: it does not *support* the other formats than display format, but it allows to add some additional drawers.

This support uses several weak functions and tables to redirect the image creation and drawings. When this support is not used (when the VEE Port does not need to support *extra* images), this support can be removed to reduce the footprint (by removing the indirection tables) and increase the performances (by reducing the number of runtime function calls).

In addition to the Display, Standard, and Custom formats, the MicroUI C module implementation introduces the notion of *Single* and *Multiple* formats, more specifically *Single Format Implementation* and *Multiple Formats Implementation*.

Single Format Implementation (Default Implementation)

This MicroUI BufferedImage implementation can only target images with the display format. In other words, the application cannot create a MicroUI BufferedImage with a format different than the display format. This is the most frequent use case, the only one available with MicroUI before version 3.2.

Hint: To select this implementation (to disable the multi formats support), the define `UI_GC_SUPPORTED_FORMATS` must be unset or lower than `2`.

This is the default implementation.

Multiple Formats Implementation

This MicroUI BufferedImage implementation allows the creation of a MicroUI BufferedImage whose format differs from the display format. This advanced use case is available only with MicroUI 3.2 or higher.

Hint: To select this implementation, the define `UI_GC_SUPPORTED_FORMATS` must be set to `2` or more. Its value defines the available number of *extra* formats the VEE Port features.

The MicroUI C module uses some tables to redirect the image management to the expected *drawer*. There is one table per Abstraction Layer API not to embed all algorithms (a table and its functions are only embedded in the final binary file if and only if the MicroUI drawing method is called). The tables size is dimensioned according to the define value.

To manipulate the tables, the C module uses 0-based index whose value is different from the image format value. For instance, according to the VEE Port capabilities, the support image format ARGB8888 can have the index `1` for a given VEE Port and `2` for another one. This differentiation reduces the size of the tables: when the VEE Port does not support a format, no extra size in the tables is used (no empty cell).

Note: The index `0` is reserved for the *display* format.

A table holds a list of functions for a given algorithm. For instance, the following table allows redirecting the drawing `writePixel` to the drawers `0` to `2`:

```
static const UI_DRAWING_writePixel_t UI_DRAWER_writePixel[] = {
  &UI_DRAWING_writePixel_0,
  &UI_DRAWING_writePixel_1,
#if (UI_GC_SUPPORTED_FORMATS > 2)
  &UI_DRAWING_writePixel_2,
#endif
};
```

- `UI_DRAWING_writePixel_0` is the drawing function called when the image format is the display format,

- UI_DRAWING_writePixel_1 and UI_DRAWING_writePixel_2 are the drawing functions called for the images whose format are respectively identified by the index 1 and 2 (see *Image Creation* below).

  By default, the C module only manages up to 3 formats: the *display* format (index 0) and two other formats. To add another format, the C module must be customized: look for everywhere the define UI_GC_SUPPORTED_FORMATS is used and add a new cell in the tables.

Custom Format

A MicroUI BufferedImage can have a *custom* format once the Multiple Formats Implementation is selected. However, third-party support is required to render this kind of image.

Hint: In addition to the #define UI_GC_SUPPORTED_FORMATS, the #define UI_FEATURE_IMAGE_CUSTOM_FORMATS must be set. This is the same preprocessor macro used to render custom RAW images: see *Custom Format Support*.

Image Creation

Overview

Creating an image consists of several steps. The Graphics Engine manages these steps, which calls four Abstraction Layer APIs. The MicroUI C Module already implements these four LLAPI.

According to the support of multiple drawers, the C module redirects or not these LLAPI to some ui_drawing.h functions. The image creation steps are briefly described below; refer to the following chapters for more details.

1. The application asks for the creation of a buffered image.
2. The Graphics Engine calls the LLAPI LLUI_DISPLAY_IMPL_getDrawerIdentifier() : this function allows to get a drawer index related to the image format. The index 0 indicates to use the default drawer: the *display* drawer. A positive value indicates a drawer index for all other formats than the display format. A negative index indicates that the VEE Port does not support the image format (in that case, the image creation is refused, and an exception is thrown in the application).
3. Depending on the image format, the Graphics Engine calculates the minimal stride of the image. This stride can be customized to fit the GPU constraint (see *GPU Format Support*) by implementing the LLAPI LLUI_DISPLAY_IMPL_getNewImageStrideInBytes() .
4. The Graphics Engine determines the image buffer size according to the image format, size (width and height), and stride (see previous step). This size and the buffer alignment can be adjusted thanks to the LLAPI LLUI_DISPLAY_IMPL_adjustNewImageCharacteristics() . The buffer size should be larger or equal to that calculated by the Graphics Engine. If smaller, the Graphics Engine will use the initial value. For a *custom* image, the initial value is 0: the VEE Port must set a positive value; otherwise, the image creation is refused, and an exception is thrown in the application.
5. The Graphics Engine allocates the image buffer according to the values adjusted before (size and alignment).
6. Finally, the Graphics Engine calls the LLAPI LLUI_DISPLAY_IMPL_initializeNewImage() that allows the VEE Port to initialize the image buffer (often only useful for custom images).

Single Format Implementation

The MicroUI C module implements the four LLAPI to create only MicroUI BufferedImages with the *display* format.

- LLUI_DISPLAY_IMPL_getDrawerIdentifier() : the C module checks if the image format is the *display* format. If yes, it returns the index 0 indicating the Graphics Engine to use the default drawer. If not, it returns a negative index: the image creation is refused.
    - It redirects the three last LLAPI to some ui_drawing.h functions. These ui_drawing.h functions are already implemented as weak functions, which allows the VEE Port to implement only the required functions:
    - Implementation of LLUI_DISPLAY_IMPL_getNewImageStrideInBytes() calls UI_DRAWING_getNewImageStrideInBytes() , the weak function returns the stride given as parameter.
    - Implementation of LLUI_DISPLAY_IMPL_adjustNewImageCharacteristics() calls UI_DRAWING_adjustNewImageCharacteristics() , the weak function does nothing.
    - Implementation of LLUI_DISPLAY_IMPL_initializeNewImage() calls UI_DRAWING_initializeNewImage() , the weak function does nothing.

Multiple Formats Implementation

The MicroUI C module implements the four LLAPI to create a MicroUI BufferedImage with any format.

- LLUI_DISPLAY_IMPL_getDrawerIdentifier() : the C module checks if the image format is the *display* format. If yes, it returns the index 0``indicating the Graphics Engine to use the default drawer. If not, it calls the function ``UI_DRAWING_is_drawer_1() and then UI_DRAWING_is_drawer_2() . The VEE Port has the responsibility to implement at least one function. The index 1 or 2 will be assigned to the image format according to the VEE Port capabilities. The image creation is refused if no drawer is found for the given format.
    - It redirects the three last LLAPI to the associated tables:
    - Implementation of LLUI_DISPLAY_IMPL_getNewImageStrideInBytes() calls the functions of the table UI_DRAWER_getNewImageStrideInBytes[] , the weak functions return the stride given as parameter.
    - Implementation of LLUI_DISPLAY_IMPL_adjustNewImageCharacteristics() calls the functions of the table UI_DRAWER_adjustNewImageCharacteristics[] , the weak functions do nothing.
    - Implementation of LLUI_DISPLAY_IMPL_initializeNewImage() calls the functions of the table UI_DRAWER_initializeNewImage[] , the weak functions do nothing.

Display and Standard Image

For this kind of image, the implementation of the functions getNewImageStrideInBytes , adjustNewImageCharacteristics and initializeNewImage is optional: it mainly depend on the *GPU support*.

Custom Image

For the *custom* images, the implementation of the function getNewImageStrideInBytes is optional but the implementation of the functions adjustNewImageCharacteristics and initializeNewImage is mandatory:

- adjustNewImageCharacteristics has to set the image buffer size (the default value is 0 , which is an invalid size); the Graphics Engine will use this value to allocate the image buffer.
- initializeNewImage must initialize the custom image buffer.

Image Closing

The BSP has the responsibility to free the third-party resources associated with an image. Most of the time, the resources are allocated and initialized in the implementation of LLUI_DISPLAY_IMPL_initializeNewImage() (see above). When the Graphics Engine closes an image, it calls the function LLUI_DISPLAY_IMPL_freeImageResources() . Depending on whether multiple drawers are supported, the C module may redirect this LLAPI to some ui_drawing.h functions.

Single Format Implementation

The MicroUI C module provides an implementation of the LLAPI. By default, no third-party resources are associated with buffered images. Therefore, LLUI_DISPLAY_IMPL_freeImageResources() calls the weak function UI_DRAWING_freeImageResources() that does nothing.

If the function UI_DRAWING_initializeNewImage() has been implemented in the BSP, the function UI_DRAWING_freeImageResources() should be implemented too.

Multiple Formats Implementation

The MicroUI C module implements the LLAPI to let each image manager close the image resources. The implementation of LLUI_DISPLAY_IMPL_freeImageResources() calls the functions of the table UI_DRAWER_freeImageResources[] , which have default weak implementations that do nothing.

Display and Standard Image

For this kind of image, implementing the function `freeImageResources` is optional: it mainly depends on the *GPU support*.

Custom Image

For the *custom* images, the implementation of the function `freeImageResources` is optional, but often required to free the third-party resources.

Draw into the Image: Display Format

Overview

To draw into a buffered image with the display format, the same concepts to draw in the display back buffer are used: the MicroUI Abstraction Layer drawings are redirected to the `ui_drawing.h` functions (see *Drawings* for more details).

The MicroUI C module already implements all `ui_drawing.h` functions, and the drawings are redirected to the *Graphics Engine Software Algorithms*. However the function names are `UI_DRAWING_DEFAULT_drawX()` and not `UI_DRAWING_drawX()`. Thanks to the define `UI_GC_SUPPORTED_FORMATS`, the function names are redefined with C macros. This compile-time redirection allows using the same implementation (`UI_DRAWING_DEFAULT_drawX()`) when the multiple formats support is disabled or enabled (when the target is an image with the same format as the display).

The weak implementation of the function `UI_DRAWING_DEFAULT_drawX()` calls *Graphics Engine Software Algorithms*. This implementation allows a GPU or a third-party drawer to perform the rendering (see *Drawings* for more details).

Single Format Implementation

The define `UI_GC_SUPPORTED_FORMATS` is unset or lower than `2`; the compile-time redirection is:

```
#define UI_DRAWING_DEFAULT_writePixel UI_DRAWING_writePixel
```

Multiple Formats Implementation

For the images whose format is the display format (index `0`, see *Multiple Formats Implementation*), the compile-time redirection is:

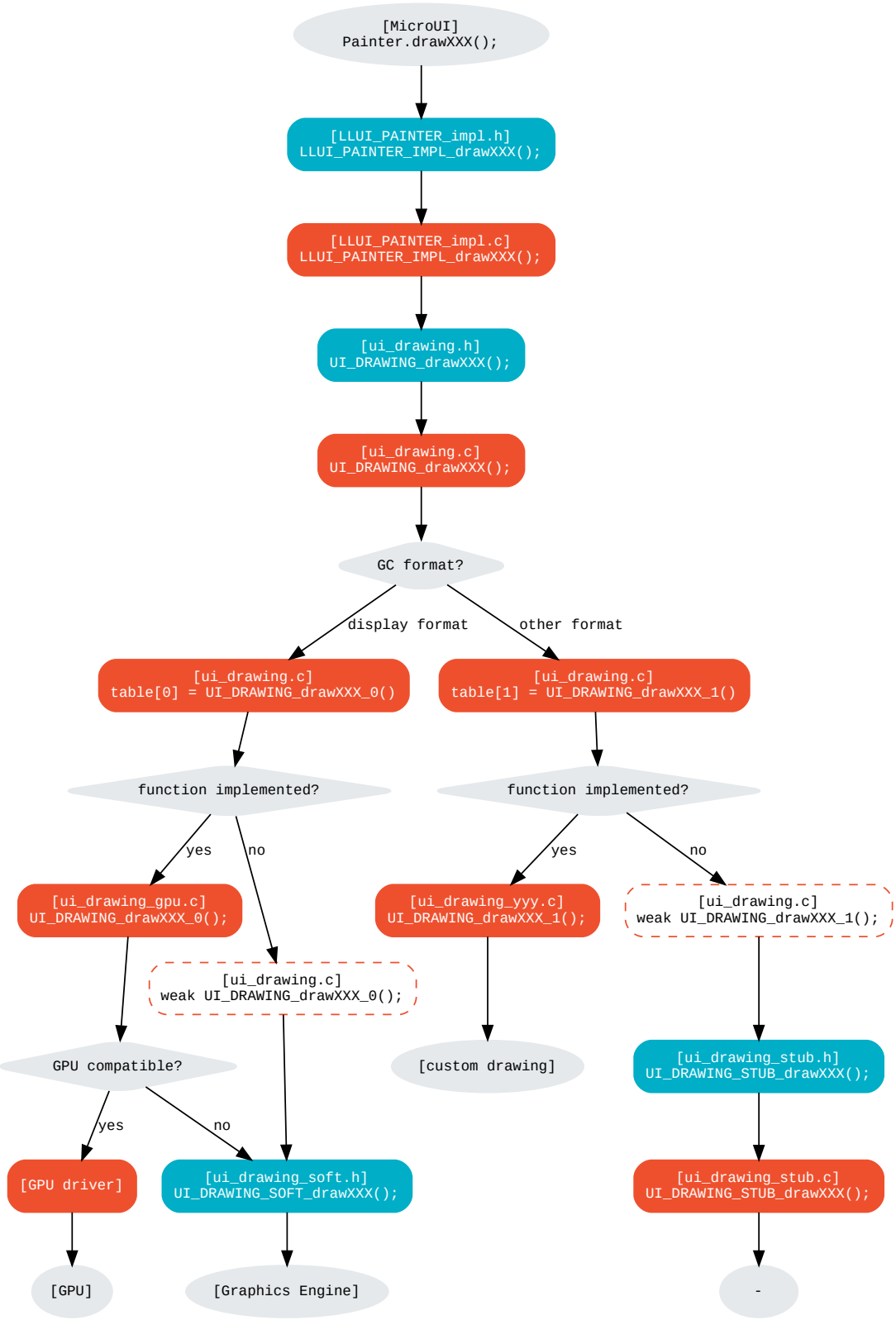
```
#define UI_DRAWING_DEFAULT_writePixel UI_DRAWING_writePixel_0
```

Draw into the Image: Non-Display Format

To draw into a buffered image with a format different than the display format, the *Multiple Formats Implementation* must be selected.

For the images whose format is not the display format (index 1 and 2), the C module provides weak implementations that do nothing.

The following diagram illustrates the drawing of a shape (not an image, see *Draw the Image: Multiple Formats Implementation*):


[MicroUI]
Painter.drawXXX();
[LLUI_PAINTER_impl.h]
LLUI_PAINTER_IMPL_drawXXX();
[LLUI_PAINTER_impl.c]
LLUI_PAINTER_IMPL_drawXXX();
[ui_drawing.h]
UI_DRAWING_drawXXX();
[ui_drawing.c]
UI_DRAWING_drawXXX();
GC format?
display format
other format
[ui_drawing.c]
table[0] = UI_DRAWING_drawXXX_0()
[ui_drawing.c]
table[1] = UI_DRAWING_drawXXX_1()
function implemented?
function implemented?
yes
no
yes
no
[ui_drawing_gpu.c]
UI_DRAWING_drawXXX_0();
[ui_drawing_yyy.c]
UI_DRAWING_drawXXX_1();
[ui_drawing.c]
weak UI_DRAWING_drawXXX_1();
[ui_drawing.c]
weak UI_DRAWING_drawXXX_0();
GPU compatible?
[custom drawing]
[ui_drawing_stub.h]
UI_DRAWING_STUB_drawXXX();
yes
no
[GPU driver]
[ui_drawing_soft.h]
UI_DRAWING_SOFT_drawXXX();
[ui_drawing_stub.c]
UI_DRAWING_STUB_drawXXX();
[GPU]
[Graphics Engine]
-

LLUI_PAINTER_IMPL_drawLine (available in MicroUI C Module)

See *MicroUI C Module*.

UI_DRAWING_drawLine (available in MicroUI C Module)

```
static const UI_DRAWING_drawLine_t UI_DRAWER_drawLine[] = {
  &UI_DRAWING_drawLine_0,
  &UI_DRAWING_drawLine_1,
#if (UI_GC_SUPPORTED_FORMATS > 2)
  &UI_DRAWING_drawLine_2,
#endif
};

DRAWING_Status UI_DRAWING_
↪drawLine(MICROUI_GraphicsContext* gc, jint startX, jint startY, jint endX, jint endY){
  // Table redirection according to the drawer index
  return (*UI_DRAWER_drawLine[gc->drawer])(gc, startX, startY, endX, endY);
}
```

The implementation in the MicroUI C module redirects the drawing to the expected drawer. The drawer is identified by the index stored in the `MICROUI_GraphicsContext` (index fixed during the image creation).

UI_DRAWING_drawLine_0 (available in MicroUI C Module)

```
#define UI_DRAWING_DEFAULT_drawLine UI_DRAWING_drawLine_0
```

The index `0` is reserved for drawing into the image whose format is the display format (see above). The function name is set thanks to a preprocessor macro to reuse the same code between Single and Multiple Formats Implementations.

The behavior after this function is similar to *Custom Implementation*.

UI_DRAWING_drawLine_1 (available in MicroUI C Module)

```
 // use the compiler's 'weak' attribute
___weak DRAWING_Status UI_DRAWING_drawLine_
↪1(MICROUI_GraphicsContext* gc, jint startX, jint startY, jint endX, jint endY){
    // Default behavior: call the stub implementation
  return UI_DRAWING_STUB_drawLine(gc, startX, startY, endX, endY);
}
```

The implementation of the weak function only consists in calling the stub implementation.

UI_DRAWING_STUB_drawLine (available in MicroUI C Module)

```
DRAWING_Status UI_DRAWING_STUB_
↪drawLine(MICROUI_GraphicsContext* gc, jint startX, jint startY, jint endX, jint endY){
  // Set the drawing log flag "not implemented"
  LLUI_DISPLAY_reportError(gc, DRAWING_LOG_NOT_IMPLEMENTED);
  return DRAWING_DONE;
}
```

The implementation only consists in setting the *Drawing log* `DRAWING_LOG_NOT_IMPLEMENTED` to notify the application that the drawing has not been performed.

UI_DRAWING_drawLine_1 (to write in the BSP)

```
// this drawer has the index 1
#define UI_DRAWING_IDENTIFIER_A8_FORMAT 1
#define UI_DRAWING_A8_is_drawer␣
↪CONCAT(UI_DRAWING_is_drawer_, UI_DRAWING_IDENTIFIER_A8_FORMAT)
#define UI_DRAWING_A8_drawLine␣
↪CONCAT(UI_DRAWING_drawLine_, UI_DRAWING_IDENTIFIER_A8_FORMAT)
```

This example illustrates how to implement the drawLine function for an image with the format A8 . The drawer should be written in its file. However, the MicroUI C module advises not to use directly the name UI_DRAWING_drawLine_1 but to use this mechanism to redirect at compile-time the call to UI_DRAWING_A8_drawLine .

- The define UI_DRAWING_IDENTIFIER_A8_FORMAT assignes the index to the A8 drawer, here 1 .
- The define UI_DRAWING_A8_is_drawer sets at compile-time the name of the is_drawer function, here: UI_DRAWING_is_drawer_1 .
- The define UI_DRAWING_A8_drawLine sets at compile-time the name of the drawLine function, here: UI_DRAWING_drawLine_1 .

UI_DRAWING_A8_is_drawer (to write in the BSP)

```
bool UI_DRAWING_A8_is_drawer(jbyte image_format) {
  return MICROUI_IMAGE_FORMAT_A8 == (MICROUI_ImageFormat)image_format;
}
```

This function (actually UI_DRAWING_is_drawer_1 thanks to the define, see above) answers true when the application tries to open a MicroUI BufferedImage with the format A8 .

UI_DRAWING_A8_drawLine (to write in the BSP)

```
DRAWING_Status UI_DRAWING_A8_
↪drawLine(MICROUI_GraphicsContext* gc, jint startX, jint startY, jint endX, jint endY){

  // Retrieve the destination buffer address
  uint8_t* destination_address = LLUI_DISPLAY_getBufferAddress(&gc->image);

  // Configure the GPU clip
  THIRD_PARTY_DRAWER_set_clip(startX, startY, endX, endY);

  // Draw the line
  THIRD_PARTY_DRAWER_draw_line(destination_address, startX, startY,
↪ endX, endY, (gc->foreground_color & 0xff) /* Use the blue component as opacity level */),

  // Here, consider the drawing as done (not an asynchronous drawing).
  return DRAWING_DONE;
}
```

This function (actually UI_DRAWING_drawLine_1 thanks to the define, see above) performs the drawing. It is very similar to *Custom Implementation*.

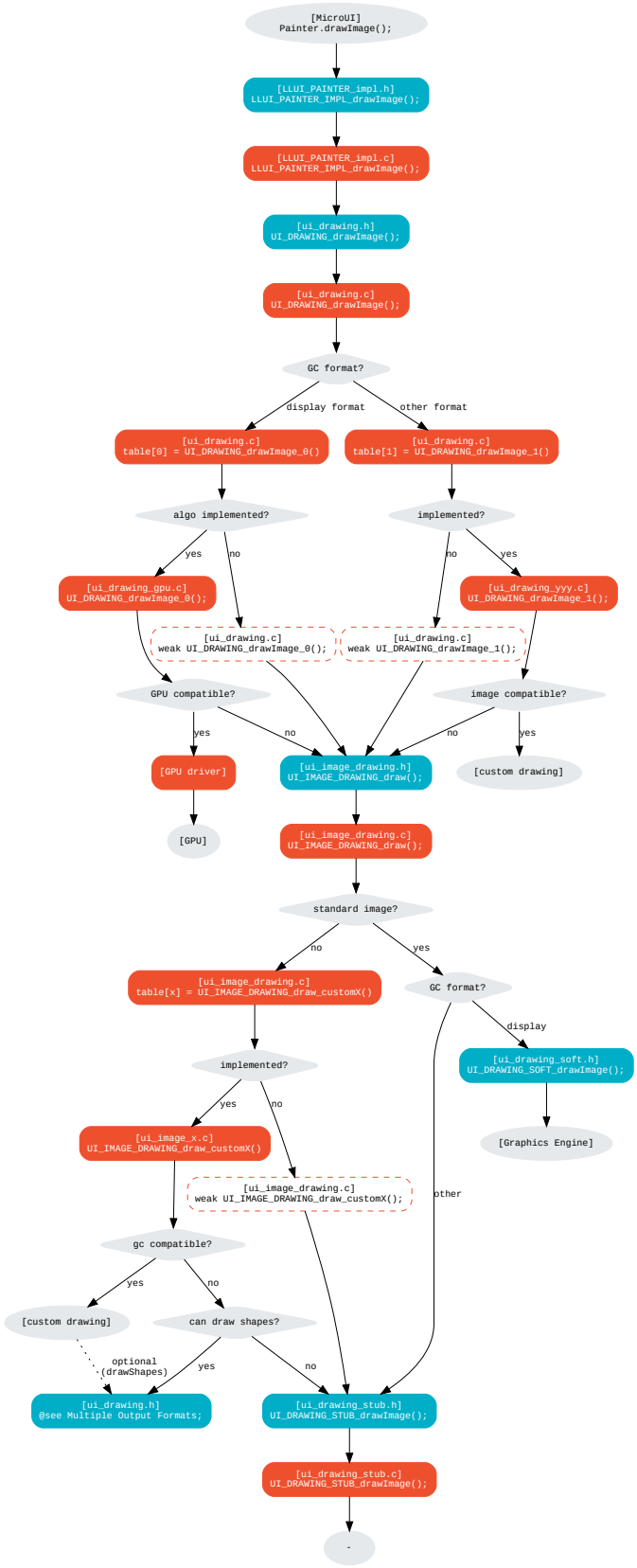
Draw the Image: Single Format Implementation

By definition, the image is a standard image (only display format is allowed), so its drawing is redirected to ui_image_drawing.h , see *Standard Formats Only (Default)*.

Draw the Image: Multiple Formats Implementation

Unlike the Single Format Implementation, the destination may be another format than the display format. Consequently, the drawer must check the image format **and** the destination format.

The following diagram illustrates the drawing of an image (draw, rotate, or scale) in another image or display back buffer (to draw a shape, see *Draw into the Image: Non-Display Format*). This diagram gathers both *draw in a custom image* and *render a custom image*.


[MicroUI]
Painter.drawImage();
[LLUI_PAINTER_impl.h]
LLUI_PAINTER_IMPL_drawImage();
[LLUI_PAINTER_impl.c]
LLUI_PAINTER_IMPL_drawImage();
[ui_drawing.h]
UI_DRAWING_drawImage();
[ui_drawing.c]
UI_DRAWING_drawImage();
GC format?
display format
other format
[ui_drawing.c]
table[0] = UI_DRAWING_drawImage_0()
[ui_drawing.c]
table[1] = UI_DRAWING_drawImage_1()
algo implemented?
implemented?
yes
no
no
yes
[ui_drawing_gpu.c]
UI_DRAWING_drawImage_0();
[ui_drawing_yyy.c]
UI_DRAWING_drawImage_1();
[ui_drawing.c]
weak UI_DRAWING_drawImage_0();
[ui_drawing.c]
weak UI_DRAWING_drawImage_1();
GPU compatible?
image compatible?
yes
no
no
yes
[GPU driver]
[ui_image_drawing.h]
UI_IMAGE_DRAWING_draw();
[custom drawing]
[GPU]
[ui_image_drawing.c]
UI_IMAGE_DRAWING_draw();
standard image?
no
yes
[ui_image_drawing.c]
table[x] = UI_IMAGE_DRAWING_draw_customX()
GC format?
display
[ui_drawing_soft.h]
UI_DRAWING_SOFT_drawImage();
implemented?
yes
no
[ui_image_x.c]
UI_IMAGE_DRAWING_draw_customX()
[Graphics Engine]
[ui_image_drawing.c]
weak UI_IMAGE_DRAWING_draw_customX();
other
gc compatible?
yes
no
[custom drawing]
can draw shapes?
optional
(drawShapes)
yes
no
[ui_drawing.h]
@see Multiple Output Formats;
[ui_drawing_stub.h]
UI_DRAWING_STUB_drawImage();
[ui_drawing_stub.c]
UI_DRAWING_STUB_drawImage();
-

The following description considers that both previous diagrams (*draw in a custom image* and *render a custom image*) have been read and understood. It only describes the *final* use-case: draw a custom image in an unknown destination (unknown destination format):

UI_IMAGE_DRAWING_draw_custom4 (to write in the BSP)

```
// This image drawer manages the custom format 4
#define UI_IMAGE_IDENTIFIER_CMD_FORMAT 4
#define UI_IMAGE_DRAWING_CMD_draw CONCAT(UI_
↪IMAGE_DRAWING_draw_custom_, UI_IMAGE_IDENTIFIER_CMD_FORMAT)

// Macro to map a custom struct "cmd_image_t*" on the MicroUI Image buffer
#define MAP_
↪CMD_ON_IMAGE(image) ((cmd_image_t*) LLUI_DISPLAY_getBufferAddress(image))

DRAWING_Status␣
↪UI_IMAGE_DRAWING_CMD_draw(MICROUI_GraphicsContext* gc, MICROUI_
↪Image* img, jint regionX, jint regionY, jint width, jint height, jint x, jint y, jint alpha){

  // Retrieve the commands list
  cmd_image_t* cmd = MAP_CMD_ON_IMAGE(img);

  for(int i = 0; i < cmd->size; i++) {
    switch (cmd->list[i].kind) {

      case COMMAND_LINE: {

        // Change the graphics context color
        gc->foreground_color = cmd->list[i].color;

        // Draw a line as usual
        UI_DRAWING_drawLine(gc, x + cmd-
↪>list[i].args[0], y + cmd->list[i].args[1], x + cmd->list[i].args[2], y + cmd->list[i].args[3]);

        break;
      }

      // All others commands
      // [...]
    }
  }

  // Restore the original color
  gc->foreground_color = original_color;

  return DRAWING_DONE;
}
```

This drawer manages a custom image with a commands buffer (a list of drawings). The image drawing consists in decoding the commands list and calling the standard shapes drawings. This drawer does not need to *recognize* the destination: the drawing of the shapes will do it.

Thanks to the define `UI_IMAGE_IDENTIFIER_CMD_FORMAT`, this drawer uses the custom format `4`.

UI_IMAGE_DRAWING_draw_custom6 (to write in the BSP)

```
// This image drawer manages the custom format 6
#define UI_IMAGE_IDENTIFIER_PROPRIETARY_FORMAT 6
#define UI_IMAGE_DRAWING_PROPRIETARY_draw CONCAT(UI_IMAGE_
↪DRAWING_draw_custom_, UI_IMAGE_IDENTIFIER_PROPRIETARY_FORMAT)

DRAWING_Status UI_IMAGE_
↪DRAWING_PROPRIETARY_draw(MICROUI_GraphicsContext* gc, MICROUI_
↪Image* img, jint regionX, jint regionY, jint width, jint height, jint x, jint y, jint alpha){

  DRAWING_Status ret;

  // Can only draw in an image with the same format as display
  if (LLUI_DISPLAY_isDisplayFormat(gc->image.format)) {
    // Call a third-party library
    THIRD_PARTY_LIB_draw_image([...]);
    ret = DRAWING_DONE; // or DRAWING_RUNNING
  }
  else {
    // Cannot draw the image: call stub implementation
    ret␣
↪= UI_DRAWING_STUB_drawImage(gc, img, regionX, regionY, width, height, x, y, alpha);
  }

  return ret;
}
```

This drawer manages an image whose format is *proprietary*. This example considers that the third-party library can only draw the image in a buffer with the display format. Otherwise, the drawing is canceled, and the stub implementation is used.

Thanks to the define `UI_IMAGE_IDENTIFIER_PROPRIETARY_FORMAT`, this drawer uses the custom format `6`.

Extended C Modules

MicroVG enables a custom format for the Buffered Vector Image. It uses the mechanisms described above and can be used as an example. See *C Modules*.

The drawings in the custom format *BVI* are implemented into the file `ui_drawing_bvi.c`.

Simulation

The simulation behavior is similar to the *MicroUI C Module* for the Embedded side.

Drawer

It is possible to draw in images with a format different than the display one by implementing the `UIDrawing` interface.

This interface contains one method for each drawing primitive. Only the necessary methods need be implemented. Each non-implemented method will result in calling the stub implementation.

The method `handledFormat()` must be implemented and returns the managed format.

Once created, the `UIDrawing` implementation must be registered as a service.

Creating an image with a standard format (different from the display one) is supported in the Front Panel as long as a `UIDrawing` is defined for this format.

Creating an image with a custom format also requires implementing the *image creation* in the VEE Port.

Image Creation

Creating images with a custom format is possible by implementing the `BufferedImageProvider` interface.

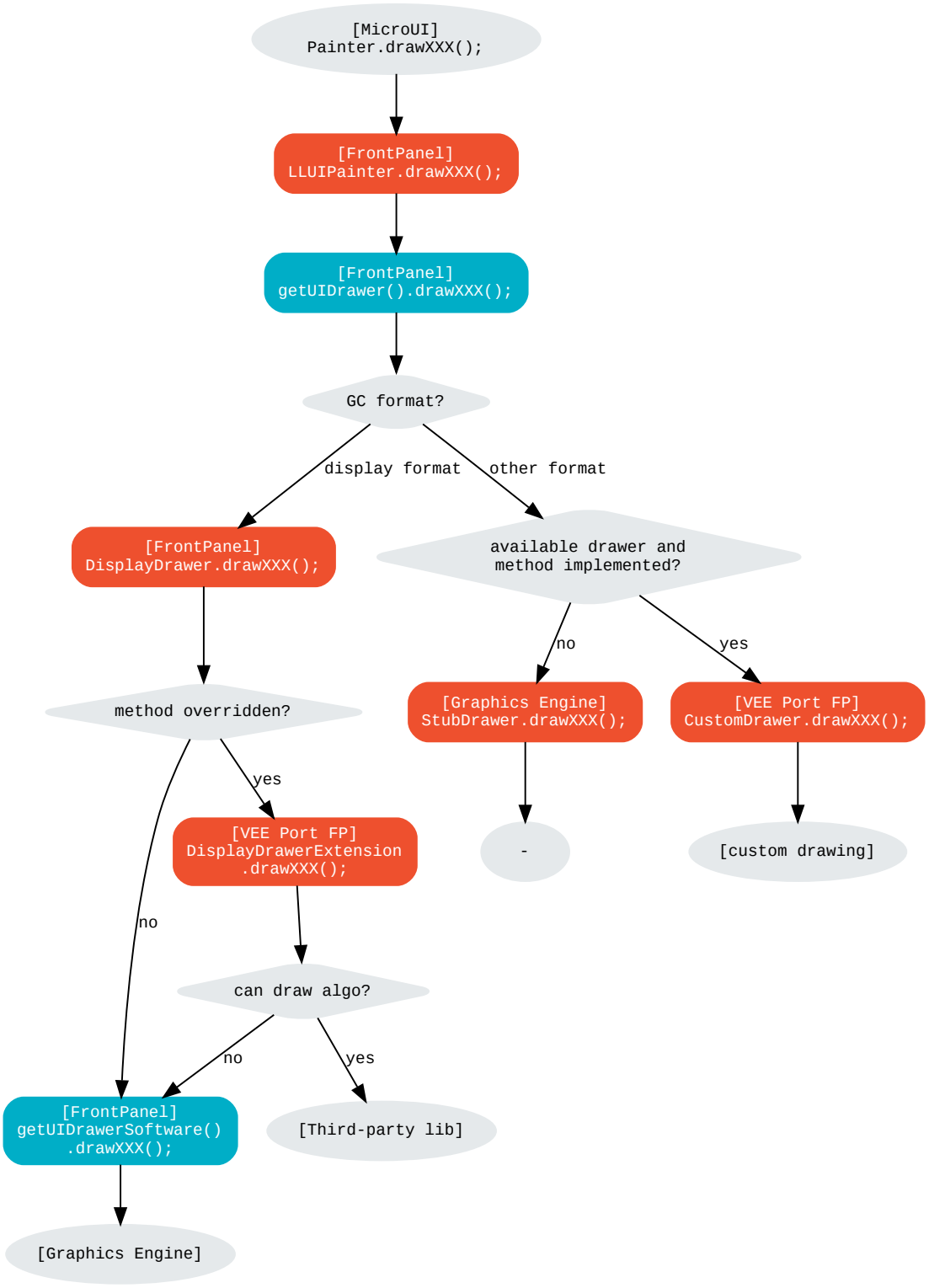
This interface extends `UIDrawing` and `UIImageDrawing` and contains a method `newBufferedImage()`. This method needs to be implemented to create the custom image. It must return an object representing the image. This object will be available in the drawing methods (*Drawer*).

The method `handledFormat()` must be implemented and returns the managed format.

Once created, the `BufferedImageProvider` implementation must be registered as a service.

Draw into the Image: Non-Display Format

The following diagram illustrates the drawing of a shape (not an image, see *Draw the Image: Multiple Formats Implementation*):


[MicroUI]
Painter.drawXXX();
[FrontPanel]
LLUIPainter.drawXXX();
[FrontPanel]
getUIDrawer().drawXXX();
GC format?
display format
other format
[FrontPanel]
DisplayDrawer.drawXXX();
available drawer and
method implemented?
no
yes
method overridden?
[Graphics Engine]
StubDrawer.drawXXX();
[VEE Port FP]
CustomDrawer.drawXXX();
yes
[VEE Port FP]
DisplayDrawerExtension
.drawXXX();
-
[custom drawing]
no
can draw algo?
no
yes
[FrontPanel]
getUIDrawerSoftware()
.drawXXX();
[Third-party lib]
[Graphics Engine]

Standard Format

Let's implement the drawer for the *ARGB8888* format (with only the draw line primitive).

```
public class MyARGB8888ImageDrawer implements UIDrawing {

  @Override
  public MicroUIImageFormat handledFormat() {
    return MicroUIImageFormat.MICROUI_IMAGE_FORMAT_ARGB8888;
  }

  @Override
  public void drawLine(MicroUIGraphicsContext gc, int x1, int y1, int x2, int y2) {
    Image image = gc.getImage();
    image.drawLine(x1, y1, x2, y2, gc.getMicroUIColor());
  }

}
```

Now, this drawer needs to be registered as a service. This can be achieved by creating a file in the resources of the Front Panel project named `META-INF/services/ej.microui.display.UIDrawing`. And its content containing the fully qualified name of the previously created image drawer.

```
com.mycompany.MyARGB8888ImageDrawer
```

It is also possible to declare it programmatically (see where a drawer is registered in the *drawing custom* section):

```
LLUIDisplay.Instance.registerUIDrawer(new MyARGB8888ImageDrawer());
```

Custom Format

Let's implement the buffered image provider for the *CUSTOM_0* format (with only the draw line primitive).

```
public class MyCustom0ImageProvider implements BufferedImageProvider {

  @Override
  public MicroUIImageFormat handledFormat() {
    return MicroUIImageFormat.MICROUI_IMAGE_FORMAT_CUSTOM_0;
  }

  @Override
  public Object newBufferedImage(int width, int height)
    // Create the image.
    return new CustomImage(width, height);
  }

  @Override
  public void drawLine(MicroUIGraphicsContext gc, int x1, int y1, int x2, int y2) {
    // Draw in the image.
    CustomImage customImage = (CustomImage) gc.getImage().getRAWImage();
    customImage.drawLine(x1, y1, x2, y2, gc.getMicroUIColor());
  }
```

(continues on next page)

(continued from previous page)

```
  @Override
  public void draw(MicroUIGraphicsContext␣
↪gc, MicroUIImage img, int regionX, int regionY, int width, int height,
      int x, int y, int alpha) {
    // Draw the image in another buffer.
    MyCustomImage customImage = (MyCustomImage) img.getImage().getRAWImage();
    customImage.drawOn(gc, regionX, regionY, width, height, x, y, alpha);
  }
}
```

Now, this buffered image provider needs to be registered as a service. This can be achieved by creating a file in the resources of the Front Panel project named `META-INF/services/ej.microui.display.BufferedImageProvider`. And its content containing the fully qualified name of the previously created buffered image provider.

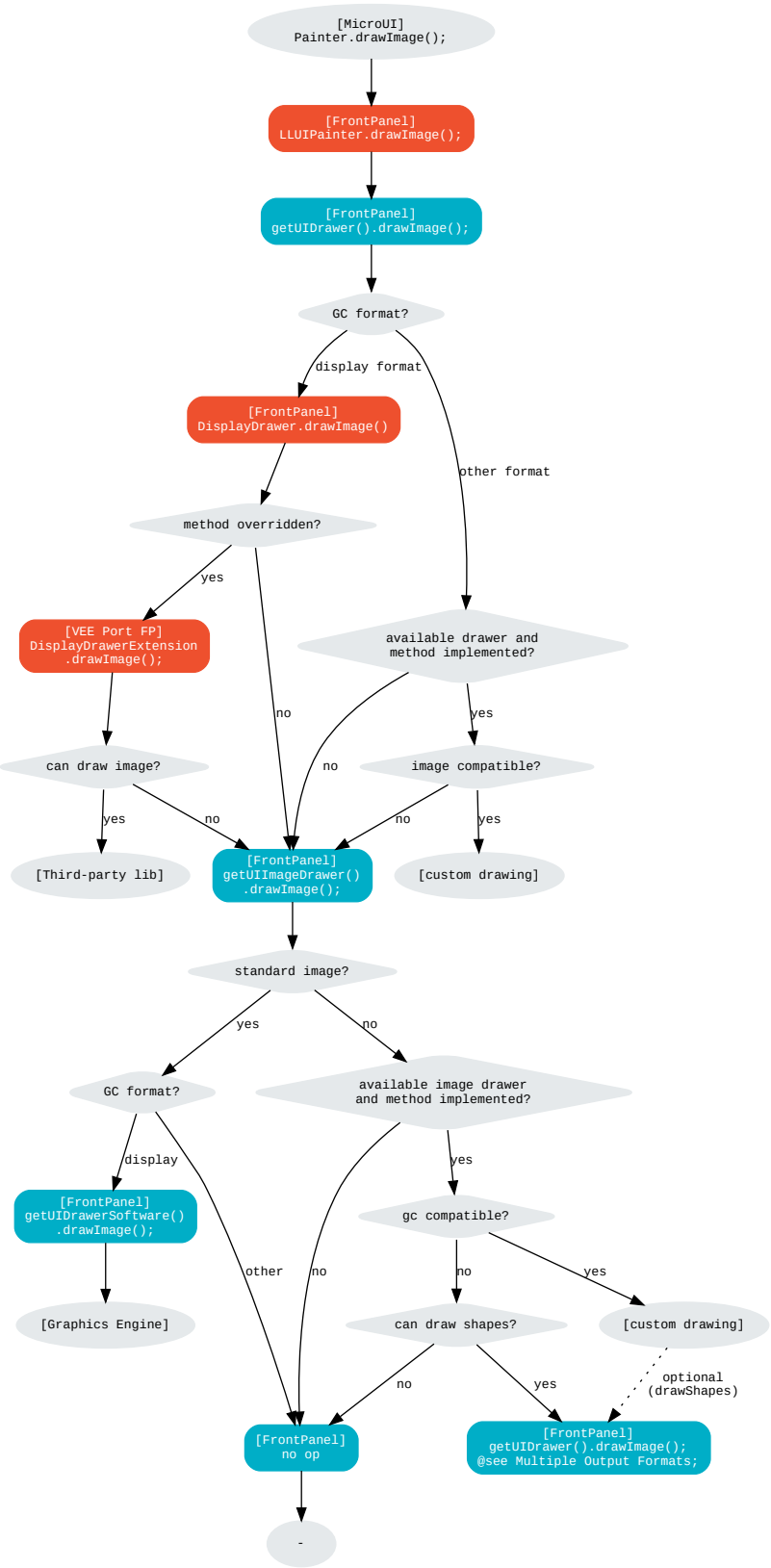
```
com.mycompany.MyCustom0ImageProvider
```

It is also possible to declare it programmatically (see where a drawer is registered in the *drawing custom* section):

```
LLUIDisplay.Instance.registerBufferedImageProvider(new MyCustom0ImageProvider());
```

Draw the Image: Multiple Formats Implementation

The following diagram illustrates the drawing of an image (draw, rotate, or scale) in another image or display back buffer (to draw a shape, see *Draw into the Image: Non-Display Format*). This diagram gathers both diagrams *draw in a custom image* and *render a custom image*.


[MicroUI]
Painter.drawImage();
[FrontPanel]
LLUIPainter.drawImage();
[FrontPanel]
getUIDrawer().drawImage();
GC format?
display format
other format
[FrontPanel]
DisplayDrawer.drawImage()
method overridden?
yes
no
[VEE Port FP]
DisplayDrawerExtension
.drawImage();
available drawer and
method implemented?
yes
no
can draw image?
image compatible?
yes
no
no
yes
[Third-party lib]
[FrontPanel]
getUIImageDrawer()
.drawImage();
[custom drawing]
standard image?
yes
no
GC format?
available image drawer
and method implemented?
display
other
no
yes
[FrontPanel]
getUIDrawerSoftware()
.drawImage();
gc compatible?
no
yes
[Graphics Engine]
can draw shapes?
[custom drawing]
no
yes
optional
(drawShapes)
[FrontPanel]
no op
[FrontPanel]
getUIDrawer().drawImage();
@see Multiple Output Formats;
-

Dependencies

- MicroUI module (see *MicroUI*),
- Display module (see *Display*).

Installation

The BufferedImage module is part of the MicroUI module and Display module. Install them to be able to use some buffered images.

Use

The MicroUI image APIs are available in the class ej.microui.display.BufferedImage.

Fonts

Overview

Principle

The notion of MicroUI Font encompasses two distinct kinds of fonts:

1. The internal font format: a simple and small bitmap format, accessible in the application by calling Font.getFont(String).
2. The *custom* (or *extended*) font format: available in MicroUI extended libraries (such as MicroVG's VectorFont), accessible in the application through subclasses of MicroUI Font.

Functional Description

Using the Font Engine consists in pregenerating the font (if required), loading the font at runtime and rendering text. The first two steps (generation and loading) are specific to the font kind (internal or custom), whereas the way to render a MicroUI Font (from the MicroUI API to the MicroUI Abstraction Layer API) is shared. This allows the application to use the same MicroUI APIs with any kind of font.

Internal Font

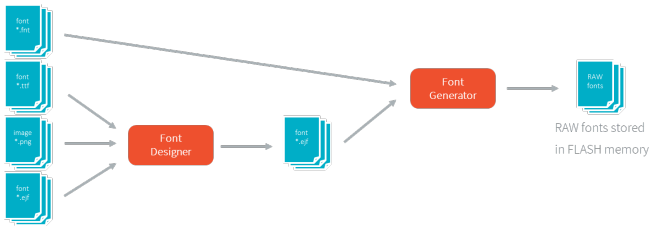


Fig. 233: Font Generation

The internal Font Engine is composed of:

- The *Font Designer* module, a graphical tool which runs within the MicroEJ IDE used to build and edit EJF fonts; it stores fonts in a VEE Port-independent format.
- The *Font Generator* module, for converting fonts from `.fnt` and VEE Port-independent format (`.ejf`) into a VEE Port-dependent format.
- The *Font Loader* module, which reads the VEE Port-dependent fonts files generated by the *Font Generator*.

The three modules are complementary:

- For standard FNT fonts, the *Font Designer* is useless and the fonts are integrated as resources by the Font Generator and loaded by the Font Loader at runtime.
- For EJF fonts, the fonts must be created and edited with the Font Designer before being integrated as resources by the Font Generator and loaded by the Font Loader at runtime.

Process overview:

1. User uses the Font Designer module to create an EJF font, and imports characters from system fonts (`*.ttf` files) and / or user images (`*.png` , `*.jpg` , `*.bmp` , etc.).
2. Font Designer module saves the font as a MicroEJ Font (`*.ejf` file).
3. The user defines, in a text file, the fonts to load (`*.fnt` and `*.ejf`).
4. The Font Generator outputs a raw file for each font to convert (the raw format is display device-dependent).
5. The raw files are embedded as (hidden) resources within the application. The raw files' data are linked into the FLASH memory.
6. When the application creates a MicroUI Font object which targets a pre-generated font, the Font Engine Core only has to link from the MicroUI Font object to the data in the FLASH memory. Therefore, the loading is very fast; only the font data from the FLASH memory is used: no copy of the font data is sent to RAM memory first.

Custom Font

The font engine for custom fonts is specific to the font itself, see *Custom Font*.

Dependencies

- MicroUI module (see *MicroUI*),
- Display module (see *Display*).

Internal Font Format

Overview

The internal font format is a binary format generated by the *Font Generator*. This is a pixelated format: each character is encoded as a grayscale bitmap image (on 1, 2, 4 or 8 bits per pixel). It does not need runtime memory allocation when loading the font or when drawing a string: the Graphics Engine's software algorithms directly read the bitmap image from the memory.

Characteristics

The binary files hold font properties:

- Font height and width, in pixels. A font has a fixed height. This height includes the white pixels at the top and bottom of each character, simulating line spacing in paragraphs. A monospace font is a font where all characters have the same width; for example, a '!' representation has the same width as a 'w'. In a proportional font, 'w' will be wider than a '!'. No width is specified for a proportional font.

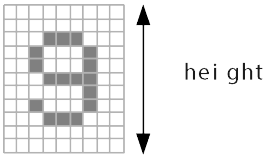


Fig. 234: Font Height

- Baseline, in pixels. All characters have the same baseline, which is an imaginary line on top of which the characters seem to stand. Characters can be partly under the line, for example 'g' or '}'. The number of pixels specified is the number of pixels above the baseline.

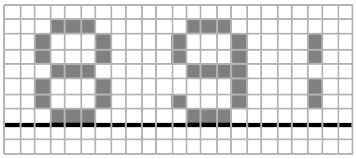


Fig. 235: Font baseline

- Space character size, in pixels. For proportional fonts, the Space character ($0x20$) is a specific character because it has no filled pixels, and so its width must be specified. For monospace, the space size is equal to the font width (and hence the same as all other characters).
- Styles: A font holds either a combination of these styles: BOLD, ITALIC, or is said to be PLAIN.
- When the selected font does not have a graphical representation of the required character, the first character in font is drawn instead.

Pixel Transparency

The Graphics Engine's software algorithms render the font according to the value stored for each pixel. If the value is 0, the pixel is not rendered. If the value is the maximum value (for example the value 3 for 2 bits-per-pixel), the pixel is rendered using the current foreground color, completely overwriting the current value of the destination pixel. For other values, the pixel is rendered by blending the selected foreground color with the current color of the destination.

If n is the number of bits-per-pixel, then the maximum value of a pixel (pmax) is $2\hat{}n - 1$. The value of each color component of the final pixel is equal to:

$$foreground * pixelValue/pmax + background * (pmax - pixelValue)/pmax$$

Languages

Supported Languages

The Graphics Engine's software algorithms manage the Unicode basic multilingual alphabet, whose characters are encoded on 16 bits, i.e. Unicode characters ranging from 0x0000 to 0xFFFF (Surrogates characters are allowed). It allows to render left-to-right or right-to-left writing systems: Latin (English, etc.), Arabic, Chinese, etc. are some supported languages. However, the rendering is always performed left-to-right, even if the string is written right-to-left. There is no support for top-to-bottom writing systems.

Complex Layout

Some languages require diacritics, contextual letters, specific character ordering, etc. and the Graphics Engine's software algorithms do not handle these specificities at runtime. To manage the complex layouts, use either of these solutions:

- Use the offline tool *Native Language Support* to automatically convert the translation messages in a characters array compatible with the Graphics Engine's software algorithms.
- Implement a complex layouter in the Abstract Layer API (see *Font Renderer*) to convert the application's strings in a character array which format is compatible with the Graphics Engine's software algorithms and call the Graphics Engine's software algorithms to draw them.

Font Generator

Principle

The Font Generator module is an off-board tool dedicated to the *internal font format*. It generates binary files ready to be displayed without the need for additional runtime memory.

Functional Description

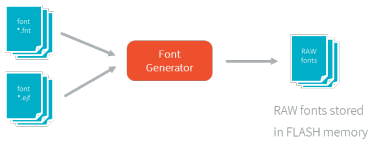


Fig. 236: Font Generator Principle

Process overview:

1. The user defines, in a text file, the fonts to load.
2. The Font Generator outputs a raw file for each font to convert.
3. The raw files are embedded as (hidden) resources within the application. The raw file's data is linked into the FLASH memory.
4. When the application draws text on the display, the font data comes directly from the FLASH memory (the font data is not copied to the RAM memory first).

Pixel Transparency

As mentioned above, each pixel of each character in an `.fnt` or `.ejf` file has one of 256 different gray-scale values. However RAW files can have 1, 2, 4 or 8 bits-per-pixel (respectively 2, 4, 16 or 256 gray-scale values). The required pixel depth is defined in the configuration file (see next chapter). The Font Generator compresses the input pixels to the required depth.

The following tables illustrate the conversion "grayscale to transparency level" (the blacker the pixel, the more opaque the encoded opacity):

- The grayscale value `0x00` is black whereas the value `0xff` is white.
- The transparency level `0x0` is fully transparent whereas the levels `0x1` (bpp == 1), `0x3` (bpp == 2) and `0xf` (bpp == 4) are fully opaque.

Table 80: Font 1-BPP RAW Conversion

Grayscale Ranges	Transparency Levels
0x00 to 0x7f	0x1
0x80 to 0xff	0x0

Table 81: Font 2-BPP RAW Conversion

Grayscale Ranges	Transparency Levels
0x00 to 0x1f	0x3
0x20 to 0x7f	0x2
0x80 to 0xdf	0x1
0xe0 to 0xff	0x0

Table 82: Font 4-BPP RAW Conversion

Grayscale Ranges	Transparency Levels
0x00 to 0x07	0xf
0x08 to 0x18	0xe
0x19 to 0x29	0xd
0x2a to 0x3a	0xc
0x3b to 0x4b	0xb
0x4c to 0x5c	0xa
0x5d to 0x6d	0x9
0x6e to 0x7e	0x8
0x7f to 0x8f	0x7
0x90 to 0xa0	0x6
0xa1 to 0xb1	0x5
0xb2 to 0xc2	0x4
0xc3 to 0xd3	0x3
0xd4 to 0xe4	0x2
0xe5 to 0xf5	0x1
0xf6 to 0xff	0x0

For 8-BPP RAW font, a transparency level is equal to $255 - \text{grayscale value}$.

Configuration File

The Font Generator uses a configuration file (called the "list file") for describing fonts that must be processed. The list file is a basic text file where each line describes a font to convert. The font file is described as a resource path, and should be available from the application classpath.

Note: The list file must be specified in the application launcher (see *Standalone Application Options*). However, all files in application classpath with suffix `.fonts.list` are automatically parsed by the Font Generator tool.

Each line can have optional parameters (separated by a ':') which define some ranges of characters to embed in the final raw file, and the required pixel depth. By default, all characters available in the input font file are embedded, and the pixel depth is 1 (i.e 1 bit-per-pixel).

Note: See *Configuration File* to understand the list file grammar.

Selecting only a specific set of characters to embed reduces the memory footprint. There are two ways to specify a character range: the custom range and the known range. Several ranges can be specified, separated by ";".

Below is an example of a list file for the Font Generator:

Listing 39: Fonts Configuration File Example

```
myfont
myfont1:latin
myfont2:latin:8
myfont3::4
```

External Resources

The Font Generator manages two configuration files when the External Resources Loader is enabled. The first configuration file lists the fonts which will be stored as internal resources within the application. The second file lists the fonts the Font Generator must convert and store in the External Resource Loader output directory. It is the BSP's responsibility to load the converted fonts into an external memory.

- Refer to the chapter *Fonts* to have more details how to use this kind of resources.
- Refer to the chapter *External Resources* to have more details how the Font Engine manages this kind of resources.

Installation

The Font Generator module is an additional tool for MicroUI library. When the MicroUI module is installed, install this module in order to be able to embed some additional fonts with the application.

If the module is not installed, the application will not be able to embed a new font. The application will be only able to use the system fonts specified during the MicroUI initialization step (see *Static Initialization*).

In the VEE Port configuration file, check `UI` > `Font Generator` to install the Font Generator module.

Use

In order to be able to embed ready-to-be-displayed fonts, you must activate the fonts conversion feature and specify the fonts configuration file.

Refer to the chapter *Standalone Application Options* (`Libraries` > `MicroUI` > `Font`) for more information about specifying the fonts configuration file.

Font Loader

Principle

The Font Loader is a module of the MicroUI runtime, dedicated to the *internal font format*, that loads font data (precomputed bitmaps of glyphs) ready to be displayed. The font data must be stored as a resource (in RAW format). Typically, these resources are generated by the *Font Generator* and embedded as internal resources or loaded from external memories (*External Resources* loader).

The loader is automatically used for all fonts retrieved through the MicroUI APIs Font.getAllFonts(), Font.getDefaultFont() and Font.getFont(String).

External Resources

Memory Management

The Font Loader is able to load some fonts located outside the CPU addresses' space range. It uses the External Resource Loader.

When a font is located in such memory, the font characters are loaded one by one from the External Memory.

The Font Loader uses a RAM buffer (External Font Heap) containing only the font character currently being drawn by the application. It is unloaded from the RAM when the Graphics Engine's software algorithms no longer need it.

The External Font Heap is stored into a RAM section called .bss.microui.display.externalFontsHeap . Its size is automatically calculated according to the external fonts used by the firmware. This size can be checked when enabling the verbose mode when building the application executable:

```
[FontGenerator] INFO: Try to encode /fonts/SourceSansPro_22px-400.ejf with bpp = 4 (selected characters range: [0x21 to 0xfc])
[FontGenerator] INFO: External fonts heap required size = 205 bytes (useless when font is an internal font).
[FontGenerator] INFO: Generation of C:\workspaces\com.microej.demo.widget\com.microej.demo.widget.common.Navigation\fonts\bin\fonts\SourceSansPro_22px-400.ejf_raw done.
[FontGenerator] INFO: Try to encode /fonts/SourceSansPro_19px-300.ejf with bpp = 4 (all available characters)
[FontGenerator] INFO: External fonts heap required size = 141 bytes (useless when font is an internal font).
[FontGenerator] INFO: Generation of C:\workspaces\com.microej.demo.widget\com.microej.demo.widget.common.Navigation\fonts\bin\fonts\SourceSansPro_19px-300.ejf_raw done.
[FontGenerator] INFO: Try to encode /fonts/SourceSansPro_16px-700.ejf with bpp = 4 (all available characters)
[FontGenerator] INFO: External fonts heap required size = 125 bytes (useless when font is an internal font).
[FontGenerator] INFO: Generation of C:\workspaces\com.microej.demo.widget\com.microej.demo.widget.common.Navigation\fonts\bin\fonts\SourceSansPro_16px-700.ejf_raw done.
[FontGenerator] INFO: Try to encode /fonts/SourceSansPro_15px-600.ejf with bpp = 4 (selected characters ranges: [0x8], [0xf], [0x21 to 0xfc], [0x20ac], [0x25cf], [0xe04b]
[FontGenerator] INFO: External fonts heap required size = 170 bytes (useless when font is an internal font).
[FontGenerator] INFO: Generation of C:\workspaces\com.microej.demo.widget\com.microej.demo.widget.common.Navigation\fonts\bin\fonts\SourceSansPro_15px-600.ejf_raw done.
[FontGenerator] INFO: Try to encode /fonts/SourceSansPro_12px-400.ejf with bpp = 4 (selected characters range: [0x21 to 0xff])
```

Fig. 237: External Font Heap size in verbose mode

However, it is possible to change this value by setting the application property ej.microui.memory.externalfontsheap.size . This option is very useful when building a kernel: the kernel may anticipate the section size required by the features.

Warning: When this size is smaller than the size required by an external font, some characters may be not drawn.

Also, the Font Loader copies a very short part of the resource (the font file) in RAM (into CPU address space range): the font header. This header remains located in RAM as long as the application is using the font. As soon as the application uses another external font, the new font replaces the old one.

Configuration File

Like internal resources, the Font Generator uses a *configuration file* (also called the "list file") for describing fonts that need to be processed. The list file must be specified in the application launcher (see *Standalone Application Options*). However, all the files in the application classpath with the suffix `.fontsext.list` are automatically parsed by the Font Generator tool.

Process

This chapter describes the steps to setup the loading of an external resource from the application:

1. Add the font to the application project resources (typically in the source folder `src/main/resources` and in the package `fonts`).
2. Create / open the configuration file (e.g. `application.fontsext.list`).
3. Add the relative path of the font and, at least, its output format (e.g. `/fonts/myFont.fnt::4`, see *Fonts*).
4. Build the application: the Font Generator converts the font in RAW format in the external resources folder (`[application_output_folder]/externalResources`).
5. Deploy the external resources to the external memory (SDCard, flash, etc.) of the device.
6. (optional) Configure the *External Resources Loader* to load from this source.
7. Build the application and run it on the device.
8. The application loads the external resource using Font.getFont(String).
9. The font loader looks for the font and only reads the font header.
10. (optional) The external resource is closed if the external resource is inside the CPU addresses' space range.
11. The application can use the font.
12. The external resource is never closed: the font characters are retrieved one by one from External Memory on demand (drawString, etc.).

Simulation

The Simulator automatically manages the external resources like internal resources. All fonts listed in `*.fontsext.list` files are copied in the external resources folder, and this folder is added to the Simulator's classpath.

Backward Compatibility

As explained *here*, the notion of Dynamic styles and the style UNDERLINED are not supported anymore by MicroUI 3. However, an external font may have been generated with an older version of the Font Generator; consequently, the generated file can hold the Dynamic style. The Font Loader can load these old versions of fonts. However, there are some runtime limitations:

- The Dynamic styles are ignored.
- The font is drawn without any dynamic algorithm.
- The font style (the style returned by Font.isBold() and Font.isItalic()) is the Dynamic style.
- For instance, when a font holds the style *bold* as dynamic style and the style *italic* as built-in style, the font is considered as *bold* + *italic*; even if the style *bold* is not rendered.

Installation

The Font Loader is part of the MicroUI module and Display module. You must install them in order to be able to use some fonts.

Use

The MicroUI font APIs are available in the class ej.microui.display.Font.

Custom Font

Principle

A custom font is a class that extends the class MicroUI Font. It is associated to a native implementation that handles, at least, the drawing of strings specified by Painter.drawString().

> **Warning:** Calling the Graphics Engine's software algorithms to draw a string with a custom font corrupts the MicroUI execution (flickering, memory corruption, unknown behavior, etc.).

Note that a MicroUI Font has a fixed height and a fixed baseline (in pixels). Yet, it can be based on a font that manage several heights. In this case, the MicroUI Font is configured with a specific height.

Requirements

The custom font class and its native implementation are responsible for:

- Providing one or several entry points to load a font (typically MyFont.getFont(*path_to_font*)).
- Defining the format of the input data (a file, a stream, a byte array, etc.).
- Loading (if required) the font on the native side.
- Managing memory (the custom font may require to allocate some memory on the native side).

- Closing the font (mainly if some native data is associated to this font).

Note: In case the input data is a file, the file can be listed in the *resources.list* file.

MicroUI Drawings

MicroUI provides several APIs to draw a string. The native implementation of the custom font should implement the associated *Abstract Layer API*. This is not mandatory but strongly recommended: the main goal is to abstract away the use of custom fonts for the application. The application should be able to manipulate any font (internal fonts or custom fonts) without using a different API for each font type.

Renderable String

The custom font class should manage the MicroUI RenderableString. A renderable string is an immutable string associated with a font. The objective of a renderable string is to be drawn faster than a standard string. Because the font data for the renderable string never changes, some information can be cached, avoiding retrieving it at drawing time.

As a consequence, the custom font should keep some data that allows one to quickly retrieve the data to draw. The data format is custom-font-dependent. In the MicroUI library, the renderable string data is represented by a byte array (see RenderableString.getSNIContext()). On the native side, this data should be cast to a structure readable by the font drawer.

Note: If there is no support for the renderable strings in the custom font, the drawings are automatically redirected to the simple drawString algorithms. Consequently, there is no advantage for the application to use the RenderableString API.

Additional APIs

The class can provide some additional methods:

- To get information on the font: ascent, descent, name, etc.
- To draw a string with additional parameters: opacity, matrix deformation, gradients, etc.

Rendering

The native implementation of the renderings (the implementation of the MicroUI Abstract Layer API to draw a string with the font) is described in the chapter *Font Renderer*.

Font Renderer

Principle

The Font Renderer is a module of the MicroUI runtime that reads and draws the strings with a font. It calls Abstraction Layer APIs to draw and transform the strings, the MicroUI RenderableString and single characters (with an optional transformation like a rotation or a scaling). In addition, the Graphics Engine provides some software algorithms to perform the rendering of the strings with the *internal font format*.

Functional Description

All MicroUI string drawings are redirected to a set of Abstraction Layer APIs. All Abstraction Layer APIs are implemented by weak functions, which call software algorithms. The BSP can override this default behavior:

- To use an advanced complex layout manager.
- To use a custom drawer.
- To handle *Custom Fonts*.

Font Formats

Internal Font Format

The Graphics Engine's software algorithms are by default able to draw an *internal font*. No extra support in the VEE Port is required to draw strings with this kind of font.

The string drawing resembles a *shape drawing*. The drawing is performed by default by the *Graphics Engine Software Algorithms* and can be overridden. This allows to enrich the *language support* by using a third-party library that provides an advanced complex layout manager.

Custom Font Format

The VEE Port must extend the Font Renderer to support the drawing of strings with a *Custom Font*. This extension can consist in:

- Decoding the font at runtime to draw it.
- Using an advanced complex layout manager.
- Using a command interpreter to perform some *shape drawings*.
- etc.

To draw strings with custom fonts, the Font Renderer introduces the notion of *custom font drawer*. This drawer is an engine that has the responsibility to draw the string with the font. Each custom font format (0 to 7) has its own font drawer.

Each drawing of a string with a custom font is redirected to the associated font drawer.

Hint: A custom font drawer can call the UI Shapes Drawing API to draw its elements in the destination.

The implementation is not the same between the Embedded side and the Simulation. However, the concepts are the same and are described in dedicated chapters.

MicroUI C Module

Principle

As described above, a *font drawer* allows drawing strings with a font which format is *custom*. The *MicroUI C module* is designed to manage the notion of drawers: it does not *support* the custom formats but allows adding some additional drawers.

This support uses several weak functions and tables to redirect the string drawings. When custom drawers are not used (when the VEE Port does not need to support *custom* fonts), this redirection can be removed to reduce the memory footprint (by removing the indirection tables) and improve the performances (by reducing the number of runtime function calls).

Internal Font Format Only (Default)

The default implementation can only draw strings with *internal fonts*. In other words, the application cannot draw with a custom font. This is the most frequent use case, the only one available with MicroUI before version 3.6.

Attention: To select this implementation (to disable the custom font support), the define UI_FEATURE_FONT_CUSTOM_FORMATS must be unset.

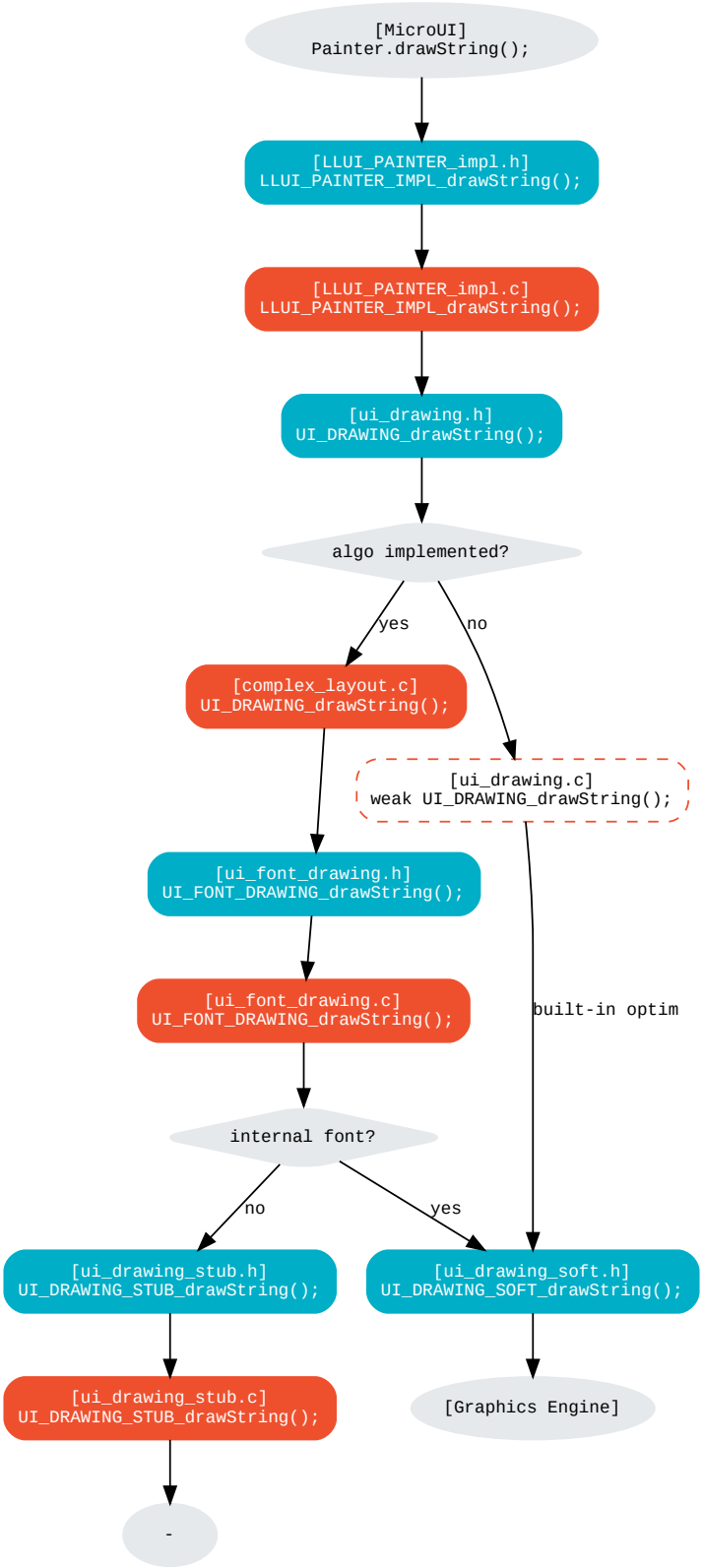
The font drawing is similar to UI_DRAWING_GPU_drawLine (see *MicroUI C Module*), except that the drawing consists in decoding the string first (to optionally apply a complex layout manager), and then calling the Graphics Engine's software algorithms to draw the string.

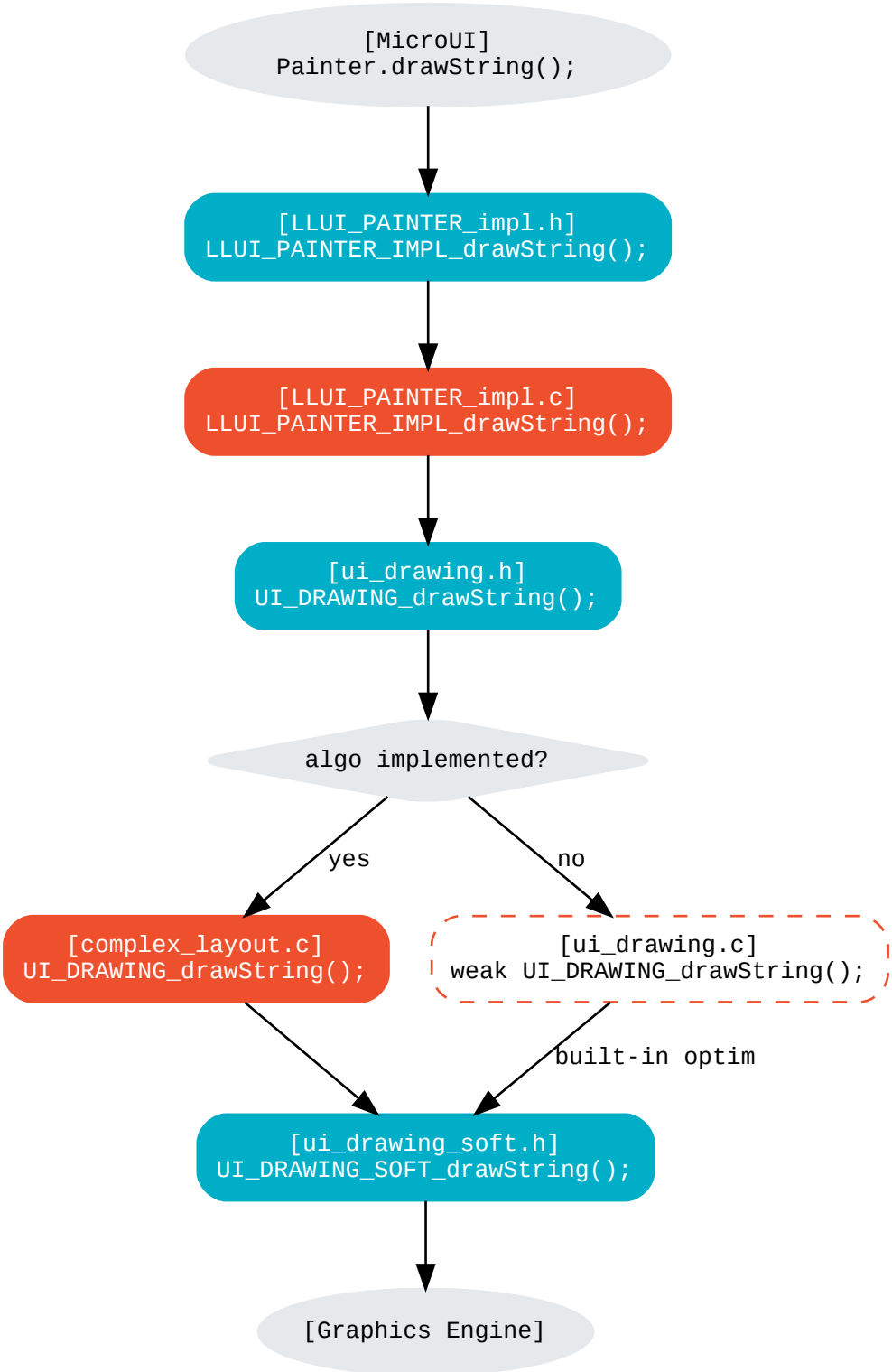
Theoretically, the weak drawer should let the font drawer handle the font instead of calling the software drawer directly. However the MicroUI C Module takes advantage of the define UI_FEATURE_FONT_CUSTOM_FORMATS : as it is not set, the C Module bypasses the indirection to the font drawer, and as a consequence the implementation of the weak function only consists in calling the Graphics Engine's software algorithm (basic string layouter, see *Languages* and software drawings). This tip reduces the footprint and the CPU usage.

An implementation of a third-party complex layouter can optionally take advantage of the define UI_FEATURE_FONT_CUSTOM_FORMATS . The following diagrams illustrate the drawing of a string with or without taking advantage of the define UI_FEATURE_FONT_CUSTOM_FORMATS (respectively *default* and *optimized* implementation).

Default Implementation

Optimized Implementation


[MicroUI]
Painter.drawString();
[LLUI_PAINTER_impl.h]
LLUI_PAINTER_IMPL_drawString();
[LLUI_PAINTER_impl.c]
LLUI_PAINTER_IMPL_drawString();
[ui_drawing.h]
UI_DRAWING_drawString();
algo implemented?
yes
no
[complex_layout.c]
UI_DRAWING_drawString();
[ui_drawing.c]
weak UI_DRAWING_drawString();
[ui_font_drawing.h]
UI_FONT_DRAWING_drawString();
[ui_font_drawing.c]
UI_FONT_DRAWING_drawString();
built-in optim
internal font?
no
yes
[ui_drawing_stub.h]
UI_DRAWING_STUB_drawString();
[ui_drawing_soft.h]
UI_DRAWING_SOFT_drawString();
[ui_drawing_stub.c]
UI_DRAWING_STUB_drawString();
[Graphics Engine]
-


[MicroUI]
Painter.drawString();
[LLUI_PAINTER_impl.h]
LLUI_PAINTER_IMPL_drawString();
[LLUI_PAINTER_impl.c]
LLUI_PAINTER_IMPL_drawString();
[ui_drawing.h]
UI_DRAWING_drawString();
algo implemented?
yes
no
[complex_layout.c]
UI_DRAWING_drawString();
[ui_drawing.c]
weak UI_DRAWING_drawString();
built-in optim
[ui_drawing_soft.h]
UI_DRAWING_SOFT_drawString();
[Graphics Engine]

LLUI_PAINTER_IMPL_drawString (available in MicroUI C Module)

Similar to LLUI_PAINTER_IMPL_drawLine , see *MicroUI C Module*.

UI_DRAWING_drawString

```
// Available in MicroUI C Module
#define UI_DRAWING_DEFAULT_drawString UI_DRAWING_drawString

// To write in the BSP (optional)
#define UI_DRAWING_LAYOUT_drawString UI_DRAWING_drawString
```

The function names are set with preprocessor macros. These name redirections are helpful when the VEE Port features more than one destination format (which is not the case here).

UI_DRAWING_LAYOUT_drawString (to write in the BSP)

Similar to UI_DRAWING_GPU_drawLine (see *MicroUI C Module*), but lets the font drawer manage the font instead of calling the software drawer directly (*Default Implementation*) or takes advantage of the define UI_FEATURE_FONT_CUSTOM_FORMATS (*Optimized Implementation*):

Default Implementation

Optimized Implementation

```
// Unlike the MicroUI C Module, this function is not "weak".
DRAWING_Status UI_DRAWING_LAYOUT_drawString(MICROUI_
↪GraphicsContext *gc, jchar *chars, jint length, MICROUI_Font *font, jint x, jint y) {

  DRAWING_Status status;
  jchar *transformed_chars = [...] ;

  // Let the font drawer manages the string (available in the C module)
  status = UI_FONT_DRAWING_drawString(gc, transformed_chars, length, font, x, y);

  return status;
}
```

```
// Unlike the MicroUI C Module, this function is not "weak".
DRAWING_Status UI_DRAWING_LAYOUT_drawString(MICROUI_
↪GraphicsContext *gc, jchar *chars, jint length, MICROUI_Font *font, jint x, jint y) {

  DRAWING_Status status;
  jchar *transformed_chars = [...] ;

#if !defined(UI_FEATURE_FONT_CUSTOM_FORMATS)
  status = UI_DRAWING_SOFT_drawString(gc, transformed_chars, length, font, x, y);
#else
  // Let the font drawer manages the string (available in the C module)
  status = UI_FONT_DRAWING_drawString(gc, transformed_chars, length, font, x, y);
#endif
  return status;
}
```

UI_DRAWING_DEFAULT_drawString (available in MicroUI C Module)

```
// Use the compiler's 'weak' attribute
__weak DRAWING_Status UI_DRAWING_DEFAULT_drawString(MICROUI_
```

(continues on next page)

(continued from previous page)

```
↪GraphicsContext *gc, jchar *chars, jint length, MICROUI_Font *font, jint x, jint y) {
#if !defined(UI_FEATURE_FONT_CUSTOM_FORMATS)
  return UI_DRAWING_SOFT_drawString(gc, chars, length, font, x, y);
#else
  return UI_FONT_DRAWING_drawString(gc, chars, length, font, x, y);
#endif
}
```

The define UI_FEATURE_FONT_CUSTOM_FORMATS is not set, so the implementation of the weak function only consists in calling the Graphics Engine's software algorithm.

Custom Font Format

In addition to the *internal font format*, this implementation allows drawing strings with a *custom font format*. This advanced use case is available only with MicroUI 3.6 or higher.

> **Attention:** To select this implementation, the define UI_FEATURE_FONT_CUSTOM_FORMATS must be set (no specific value).

The MicroUI C module uses some tables to redirect the font management to the expected extension. There is one table per Font Abstraction Layer API (draw, rotate, scale) to embed only the necessary algorithms (a table and its functions are only embedded in the final binary file if and only if the MicroUI drawing method is called).

Each table contains ten elements:

```
static const␣
↪UI_FONT_DRAWING_drawString_t UI_FONT_DRAWING_drawString_custom[] = {
  &UI_DRAWING_STUB_drawString,
  &UI_DRAWING_SOFT_drawString,
  &UI_FONT_DRAWING_drawString_custom0,
  &UI_FONT_DRAWING_drawString_custom1,
  &UI_FONT_DRAWING_drawString_custom2,
  &UI_FONT_DRAWING_drawString_custom3,
  &UI_FONT_DRAWING_drawString_custom4,
  &UI_FONT_DRAWING_drawString_custom5,
  &UI_FONT_DRAWING_drawString_custom6,
  &UI_FONT_DRAWING_drawString_custom7,
};
```

- UI_DRAWING_STUB_drawString is the drawing function called when the drawing function is not implemented,
- UI_DRAWING_SOFT_drawString is the drawing function that redirects the drawing to the *Graphics Engine Software Algorithms*,
- UI_FONT_DRAWING_drawString_customX (0 to 7) are the drawing functions for each custom format.

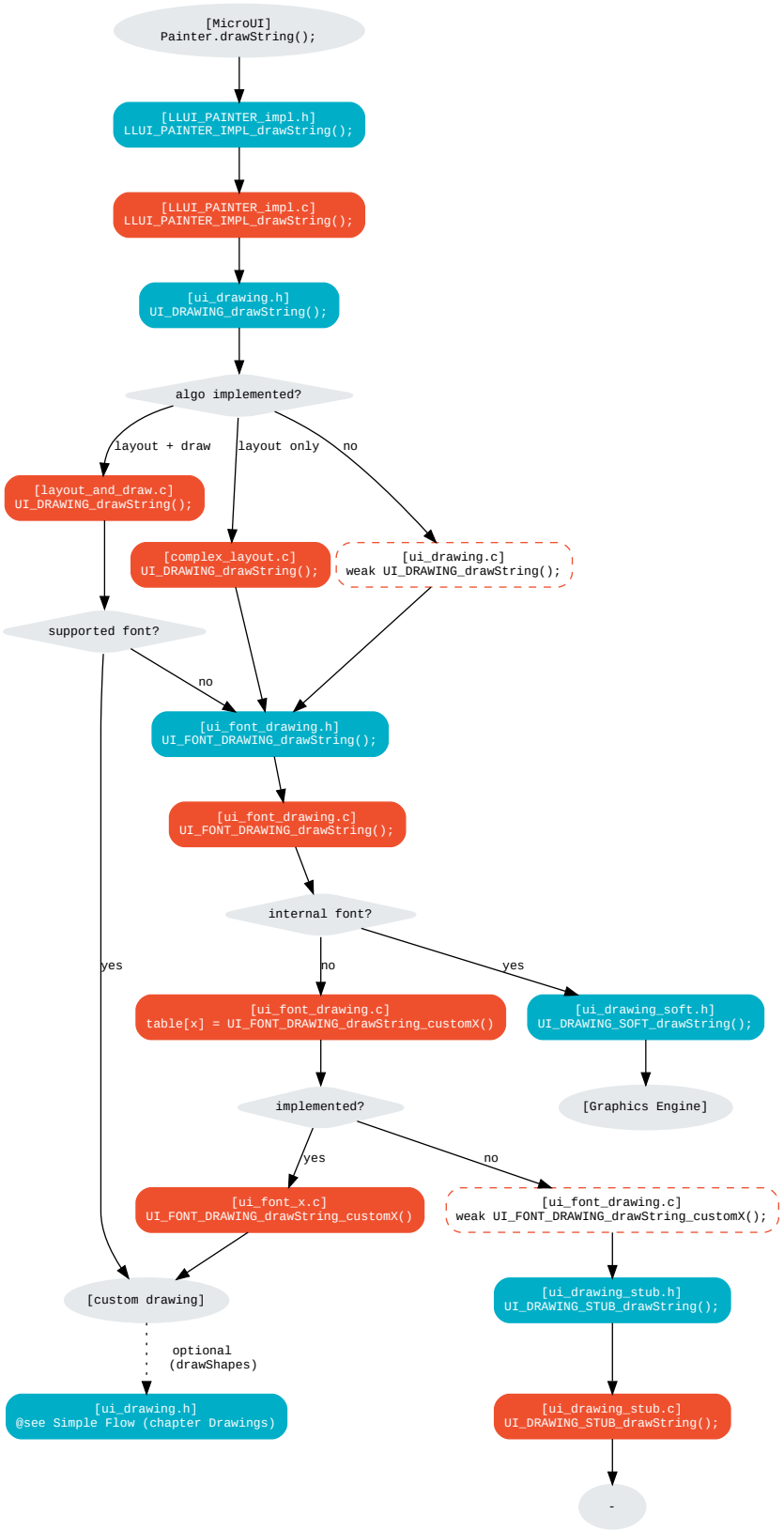
The MicroUI C Module retrieves the table index according to the font format.

The implementation of UI_DRAWING_drawString can have two behaviors:

1. It only manages the characters layouting; the drawing is performed by another C file.

2. It manages the layouting **and** the drawing; in that case, the implementation has to check if it supports the font.

   The following diagram illustrates the drawing of a string:


[MicroUI]
Painter.drawString();
[LLUI_PAINTER_impl.h]
LLUI_PAINTER_IMPL_drawString();
[LLUI_PAINTER_impl.c]
LLUI_PAINTER_IMPL_drawString();
[ui_drawing.h]
UI_DRAWING_drawString();
algo implemented?
layout + draw
layout only
no
[layout_and_draw.c]
UI_DRAWING_drawString();
[complex_layout.c]
UI_DRAWING_drawString();
[ui_drawing.c]
weak UI_DRAWING_drawString();
supported font?
no
[ui_font_drawing.h]
UI_FONT_DRAWING_drawString();
[ui_font_drawing.c]
UI_FONT_DRAWING_drawString();
internal font?
yes
no
yes
[ui_font_drawing.c]
table[x] = UI_FONT_DRAWING_drawString_customX()
[ui_drawing_soft.h]
UI_DRAWING_SOFT_drawString();
implemented?
[Graphics Engine]
yes
no
[ui_font_x.c]
UI_FONT_DRAWING_drawString_customX()
[ui_font_drawing.c]
weak UI_FONT_DRAWING_drawString_customX();
[custom drawing]
[ui_drawing_stub.h]
UI_DRAWING_STUB_drawString();
optional
(drawShapes)
[ui_drawing.h]
@see Simple Flow (chapter Drawings)
[ui_drawing_stub.c]
UI_DRAWING_STUB_drawString();
-

Take the same example as the *Internal Font Formats Only* implementation (draw a string):

UI_DRAWING_DEFAULT_drawString (available in MicroUI C Module)

```
// Use the compiler's 'weak' attribute
__weak DRAWING_Status UI_DRAWING_DEFAULT_drawString(MICROUI_
↪GraphicsContext *gc, jchar *chars, jint length, MICROUI_Font *font, jint x, jint y) {
#if !defined(UI_FEATURE_FONT_CUSTOM_FORMATS)
  return UI_DRAWING_SOFT_drawString(gc, chars, length, font, x, y);
#else
  return UI_FONT_DRAWING_drawString(gc, chars, length, font, x, y);
#endif
}
```

The define UI_FEATURE_FONT_CUSTOM_FORMATS is set so the implementation of the weak function redirects the string drawing to the font drawer manager (ui_font_drawing.h).

UI_FONT_DRAWING_draw (available in MicroUI C Module)

```
static const␣
↪UI_FONT_DRAWING_drawString_t UI_FONT_DRAWING_drawString_custom[] = {
  &UI_DRAWING_STUB_drawString,
  &UI_DRAWING_SOFT_drawString,
  &UI_FONT_DRAWING_drawString_custom0,
  &UI_FONT_DRAWING_drawString_custom1,
  &UI_FONT_DRAWING_drawString_custom2,
  &UI_FONT_DRAWING_drawString_custom3,
  &UI_FONT_DRAWING_drawString_custom4,
  &UI_FONT_DRAWING_drawString_custom5,
  &UI_FONT_DRAWING_drawString_custom6,
  &UI_FONT_DRAWING_drawString_custom7,
};

DRAWING_Status UI_FONT_DRAWING_drawString(MICROUI_
↪GraphicsContext *gc, jchar *chars, jint length, MICROUI_Font *font, jint x, jint y){
  return (*UI_FONT_
↪DRAWING_drawString_custom[_get_table_index(gc, font)])(gc, chars, length, font, x, y);
}
```

The implementation in the MicroUI C module redirects the drawing to the expected drawer. The drawer is retrieved using the font format (function _get_table_index()):

- The format is internal but the destination is not the *display* format: index 0 is returned.
- The format is internal and the destination is the *display* format: index 1 is returned.
- The format is custom: an index from 2 to 9 is returned.

UI_FONT_DRAWING_drawString_custom0 (available in MicroUI C Module)

```
// Use the compiler's 'weak' attribute
__weak DRAWING_Status UI_FONT_DRAWING_drawString_custom0(MICROUI_
↪GraphicsContext *gc, jchar *chars, jint length, MICROUI_Font *font, jint x, jint y){
  return UI_DRAWING_STUB_drawString(gc, chars, length, font, x, y);
}
```

The default implementation of UI_FONT_DRAWING_drawString_custom0 (same behavior for 0 to 7) consists in calling the stub implementation.

UI_DRAWING_STUB_drawString (available in MicroUI C Module)

```
DRAWING_Status UI_DRAWING_STUB_drawString(MICROUI_
↪GraphicsContext *gc, jchar *chars, jint length, MICROUI_Font *font, jint x, jint y){
  // Set the drawing log flag "not implemented"
  LLUI_DISPLAY_reportError(gc, DRAWING_LOG_NOT_IMPLEMENTED);
  return DRAWING_DONE;
}
```

The implementation only consists in setting the *Drawing log flag* DRAWING_LOG_NOT_IMPLEMENTED to notify the application that the drawing has not been performed.

Simulation

Principle

The simulation behavior is similar to the *MicroUI C Module* for the Embedded side.

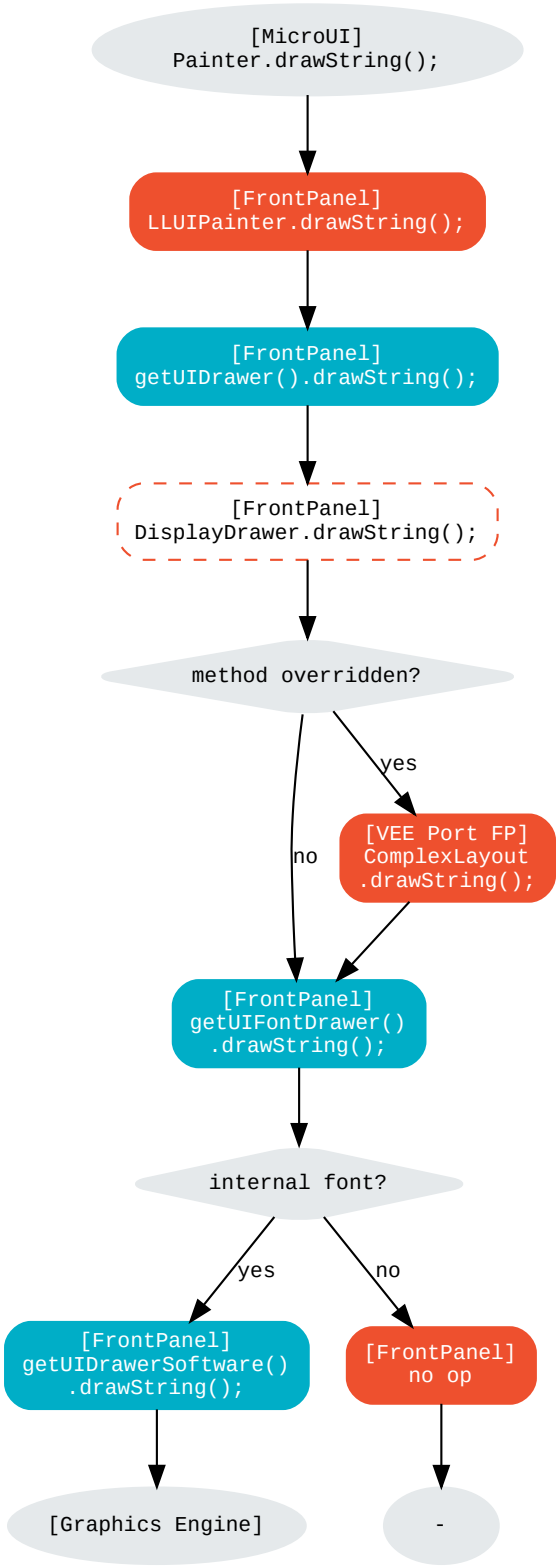
The *Front Panel* defines support for the drawers based on the Java service loader.

Internal Font Format Only (Default)

The default implementation can only draw strings with internal fonts.

Note: Contrary to the *MicroUI C Module*, the simulation does not (and doesn't need to) provide an option to disable the use of custom font.

The following diagram illustrates the drawing of a string:


[MicroUI]
Painter.drawString();
[FrontPanel]
LLUIPainter.drawString();
[FrontPanel]
getUIDrawer().drawString();
[FrontPanel]
DisplayDrawer.drawString();
method overridden?
yes
no
[VEE Port FP]
ComplexLayout
.drawString();
[FrontPanel]
getUIFontDrawer()
.drawString();
internal font?
yes
no
[FrontPanel]
getUIDrawerSoftware()
.drawString();
[FrontPanel]
no op
[Graphics Engine]
-

It is possible to override the font drawers for the internal format in the same way as the custom formats.

Custom Font Format

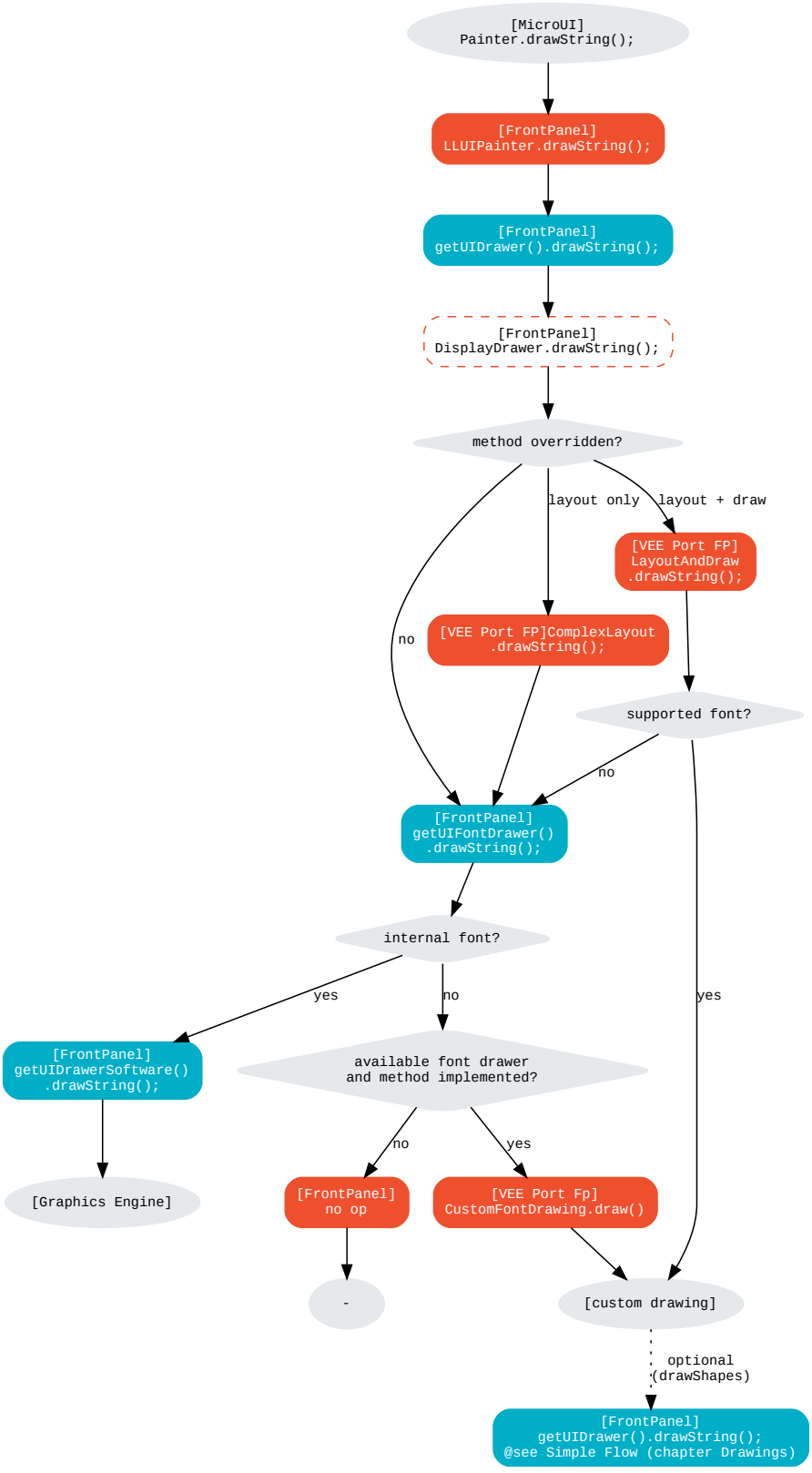
It is possible to draw fonts with a custom format by implementing the `UIFontDrawing` interface. This advanced use case is available only with MicroUI 3.6 or higher.

The `UIFontDrawing` interface contains one method for each font drawing primitive (draw, getWidth, RenderableString, rotate, scale). Only the necessary methods have to be implemented. Each non-implemented method will result in calling the stub implementation.

The method `handledFormat()` must be implemented and returns the font format handled by the drawer.

Once created, the `UIFontDrawing` implementation must be registered as a service.

The following diagram illustrates the drawing of a string:


[MicroUI]
Painter.drawString();
[FrontPanel]
LLUIPainter.drawString();
[FrontPanel]
getUIDrawer().drawString();
[FrontPanel]
DisplayDrawer.drawString();
method overridden?
layout only
layout + draw
no
[VEE Port FP]
LayoutAndDraw
.drawString();
[VEE Port FP]ComplexLayout
.drawString();
supported font?
no
[FrontPanel]
getUIFontDrawer()
.drawString();
internal font?
yes
no
yes
[FrontPanel]
getUIDrawerSoftware()
.drawString();
available font drawer
and method implemented?
[Graphics Engine]
no
yes
[FrontPanel]
no op
[VEE Port Fp]
CustomFontDrawing.draw()
-
[custom drawing]
optional
(drawShapes)
[FrontPanel]
getUIDrawer().drawString();
@see Simple Flow (chapter Drawings)

Let's implement the font drawer for the *CUSTOM_0* format.

```
public class MyCustomFontDrawer implements UIFontDrawing {

  @Override
  public MicroUIFontFormat handledFormat() {
    return MicroUIFontFormat.MICROUI_FONT_FORMAT_CUSTOM_0;
  }

  @Override
  public void draw(MicroUIGraphicsContext␣
↪gc, char[] chars, int offset, int length, MicroUIFont font, int x, int y) {
    byte[] fontData = font.getFontData();
    MyCustomFont customFont = MyCustomFont.get(fontData);
    customFont.drawOn(gc, chars, offset, length, customFont, x, y);
  }

}
```

Now, this drawer needs to be registered as a service. This can be achieved by creating a file in the resources of the Front Panel project named `META-INF/services/ej.microui.display.UIFontDrawing` and containing the fully qualified name of the previously created font drawer.

```
com.mycompany.MyCustomFontDrawer
```

It is also possible to declare it programmatically (see where a drawer is registered in the *drawing custom* section):

```
LLUIDisplay.Instance.registerUIFontDrawer(new MyCustomFontDrawer());
```

Installation

The Font Renderer is part of the MicroUI module and Display module. You must install them in order to be able to use some fonts.

Use

The MicroUI font APIs are available in the class ej.microui.display.Font.

C Modules

Principle

Several C modules implement the UI Pack's Abstraction Layer APIs. Some are generic, and some are VEE Port dependent (more precisely: GPU-dependent). The generic modules provide header files to be implemented by the specific modules. The generic C modules are available on the *Central Repository* and the specific C modules on the *Developer Repository*.

The picture below illustrates the available C modules, and the following chapters explain the aim and relations of each C module.

Note: It is a simplified view: all sources and header files of each C module are not visible.

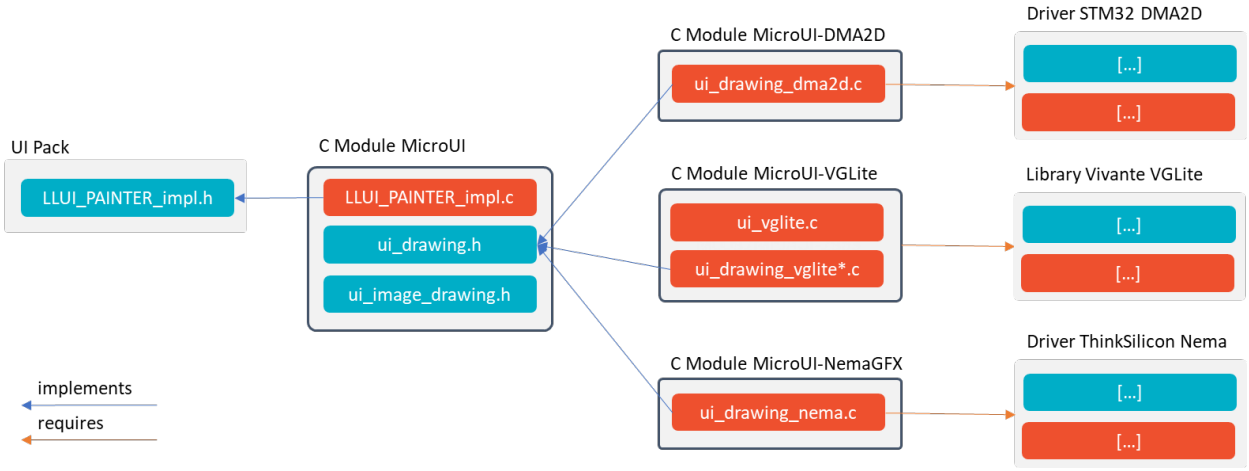


Fig. 238: MicroUI C Modules

UI Pack

The UI Pack provides a header file to implement the MicroUI drawings: LLUI_PAINTER_impl.h . See the *UI Pack* chapter to have more information.

The UI Pack and its header files are available on the Central Repository: https://repository.microej.com/modules/com/microej/pack/ui/ui-pack/.

C Module: MicroUI

This C module is divided into several parts, and each part provides an implementation of some *MicroUI Abstraction Layer APIs*. This C module is **mandatory** to use the UI Pack (the C files must be compiled in the BSP), but some C files are optional.

This C module is available on the *Central Repository*: com.microej.clibrary.llimpl#microui.

Drawings

Overview

This part aims to facilitate the MicroUI Painter classes implementation:

1. It manages the synchronization with the Graphics Engine (see LLUI_DISPLAY_requestDrawing()).
2. It checks the drawing parameters: clip, opacity, thickness, fade, image status, etc.
3. It logs the drawings (see *Debug Traces*).
4. It deports the rendering to ui_drawing.h .

   The implementation of ui_drawing.h depends on several options:

- Whether the BSP provides a *renderer* (software and / or hardware as a GPU),
- Whether the BSP is configured to handle *several destination formats*,
- Whether the BSP is configured to handle *custom image formats*.

Files

- Implements: LLUI_PAINTER_impl.h and LLDW_PAINTER_impl.h .
- C files: LLUI_PAINTER_impl.c , LLDW_PAINTER_impl.c , ui_drawing_stub.c , ui_drawing.c and ui_image_drawing.c .
- Status: mandatory.

Usage

1. Add all C files in the BSP project.
2. Check the port by running the ui validation as described in the VEE Port project template

Images Heap

Overview

This part is optional since the MicroUI Graphics Engine already includes an *Images Heap* allocator. Like MicroUI Graphics Engine's images heap allocator, the C module's images allocator is a best-fit allocator. This kind of allocator has the following constraints:

- It requires a header at the beginning of the heap section.
- It adds a header and a footer for each allocated block.
- It produces memory fragmentation: it may not allow the allocation of a block with a size equal to the free memory size.

Unlike the Graphics Engine's allocator, the C module's allocator adds some utility functions to get information about the heap:

- total size,
- free size,
- number of allocated blocks.

A third-party allocator can replace this allocator and the one in the Graphics Engine.

Files

- Implements the functions of LLUI_DISPLAY_impl.h with LLUI_DISPLAY_IMPL_imageHeap prefix.
- C file: LLUI_DISPLAY_HEAP_impl.c .
- Status: optional.

Usage

1. To use the Graphics Engine's allocator, do not add the file LLUI_DISPLAY_HEAP_impl.c in the BSP project.
2. To use the C module's allocator, add the file LLUI_DISPLAY_HEAP_impl.c in the BSP project.
3. To use a third-party allocator, do not add the file LLUI_DISPLAY_HEAP_impl.c in the BSP project and implement the LLUI_DISPLAY_IMPL_imageHeapXXX functions.

Events Logger

Overview

This part is only mandatory when the BSP calls LLUI_INPUT_dump() (see *Event Buffer*). If not included, the call to LLUI_INPUT_dump() performs nothing. It aims to log the MicroUI events and provide an events dumper.

The logger adds some metadata to each MicroUI event in a dedicated array. When the BSP is calling LLUI_INPUT_dump() , the logger is using its data to decode the events. Then, it uses an implementation of microui_event_decoder.h to describe the events.

Files

- Implements the functions of LLUI_INPUT_impl.h with LLUI_INPUT_IMPL_log_ prefix.
- C files: LLUI_INPUT_LOG_impl.c and microui_event_decoder.c .
- Status: optional.

Usage (to enable the events logger)

1. Add all C files in the BSP project.
2. Configure the options in ui_configuration.h (by default, the logger is disabled).

Buffer Refresh Strategy

Overview

This part provides three Buffer Refresh Strategies (BRS): predraw , single and legacy . Refer to the chapter *Buffer Refresh Strategy* for more information about these strategies. These strategies are optional. When no strategy is selected, the BSP should provide its own strategy. If no strategy is specified or provided, a default strategy will be used; this is a minimal, naive strategy, which should only be used when using the *Direct Buffer mode*.

Some strategies require an implementation of UI_DISPLAY_BRS_restore() (see ui_display_brs.h). A weak implementation is available; this implementation uses the function memcpy() .

Files

- Implements the functions of LLUI_DISPLAY_impl.h related to the *Buffer Refresh Strategy*: LLUI_DISPLAY_IMPL_refresh() and LLUI_DISPLAY_IMPL_newDrawingRegion() .
- C files: ui_display_brs_legacy.c , ui_display_brs_predraw.c , ui_display_brs_single.c , ui_display_brs.c and ui_rect_util.c .
- Status: optional.

Usage

1. Add all C files in the BSP project (whatever the strategy).
2. Configure the options in ui_configuration.h .
3. Comment the line #error [...]" .
4. (optional) Implement UI_DISPLAY_BRS_restore() (using a GPU, for instance).

C Module: MicroUI Over DMA2D

Overview

This C module is a specific implementation of the C module MicroUI over STM32 DMA2D (Chrom-ART Graphics Accelerator):

- It implements a set of drawings using the official Chrom-ART Graphics Accelerator API.
- It is compatible with several STM32 MCU: STM32F4XX , STM32F7XX and STM32H7XX .
- It manages several configurations of memory cache.
- It is compatible with the *multiple destination formats* module (but can only handle one destination format).
- It is compatible with the *Buffer Refresh Strategies (BRS)* predraw , single and legacy (switch).

This C module is available on the *Developer Repository*: com.microej.clibrary.llimpl#microui-dma2d.

Files

- Implements some functions of ui_drawing.h (see above).
- C file: ui_drawing_dma2d.c .
- Status: optional.

Usage

1. Install the *C Module for MicroUI* and follow its implementation rules.
2. Add the C file to the BSP project.
3. Add the BSP global define DRAWING_DMA2D_BPP to specify the destination format: 16, 24, or 32 respectively DMA2D_RGB565 , DMA2D_RGB888 and DMA2D_ARGB8888 .
4. Call UI_DRAWING_DMA2D_initialize() from LLUI_DISPLAY_IMPL_initialize() .
5. Check the port by running the ui validation as described in the VEE Port project template

Drawings

The following table describes the accelerated drawings:

Feature	Comment
Fill rectangle	
Draw image	ARGB8888, RGB888, RGB565, ARGB1555, ARGB4444, A8, A4[1]

Cache

Some STM32 MCUs use a memory cache.

This cache must be cleared before using the DMA2D:

- Before the call to HAL_DMA2D_Start_IT() .
- Before the call to HAL_DMA2D_BlendingStart_IT() .

Usage

1. Check the configuration of the define DRAWING_DMA2D_CACHE_MANAGEMENT in ui_dma2d_configuration.h .

[1] The first and last odd columns are drawn in software due to GPU memory alignment constraints.

Buffer Refresh Strategy “Predraw”

This strategy requires the copying of some regions from the front buffer to the back buffer on demand (function `UI_DISPLAY_BRS_restore()`, see above). To perform these copies, this CCO uses the `UI_DRAWING_DMA2D_xxx_memcpy()` functions.

Usage

1. The function `UI_DRAWING_DMA2D_IRQHandler()` must be called from the DMA2D IRQ routine.
2. The function `UI_DRAWING_DMA2D_memcpy_callback()` should not be implemented (useless).

Example of Implementation

```
void LLUI_DISPLAY_IMPL_flush(MICROUI_
↪GraphicsContext* gc, uint8_t flush_identifier, const ui_rect_t regions[], size_t length) {

  // store the flush identifier
  g_current_flush_identifier = flush_identifier;

  // change the front buffer address
  HAL_LTDC_SetAddress(&hLtdcHandler,
↪ (uint32_t)LLUI_DISPLAY_getBufferAddress(&gc->image), LTDC_ACTIVE_LAYER);

  // ask an interrupt for the next LCD tick
  lcd_enable_interrupt();
}

void LTDC_IRQHandler(LTDC_HandleTypeDef *hltdc) {
  // LTDC register reload
  __HAL_LTDC_ENABLE_IT(hltdc, LTDC_IT_RR);

  // notify the MicroUI Graphics Engine
  uint8_t* buffer = (uint8_t*)(BACK_
↪BUFFER == LTDC_Layer->CFBAR ? FRAME_BUFFER : BACK_BUFFER);
  LLUI_DISPLAY_setBackBuffer(g_current_flush_identifier, buffer, from_isr);
}

void DMA2D_IRQHandler(void) {
  // call CCO DMA2D function
  UI_DRAWING_DMA2D_IRQHandler();
}
```

Buffer Refresh Strategy "Single"

Usually, this strategy is used when the front buffer cannot be mapped dynamically: the same buffer is always used as the back buffer. However, the front buffer can be mapped on a memory buffer that is in the CPU address range. In that case, the UI_DRAWING_DMA2D_xxx_memcpy() functions can be used to copy the content of the back buffer to the front buffer.

Usage

1. The function UI_DRAWING_DMA2D_configure_memcpy() must be called from the implementation of LLUI_DISPLAY_IMPL_flush() .
2. The function UI_DRAWING_DMA2D_start_memcpy() must be called from the LCD controller IRQ routine.
3. The function UI_DRAWING_DMA2D_IRQHandler() must be called from the DMA2D IRQ routine.
4. The function UI_DRAWING_DMA2D_memcpy_callback() must be implemented to unlock the MicroUI Graphics Engine.

Example of Implementation

```
void LLUI_DISPLAY_IMPL_flush(MICROUI_
↪GraphicsContext* gc, uint8_t flush_identifier, const ui_rect_t regions[], size_t length) {

  // store the flush identifier
  g_current_flush_identifier = flush_identifier;

  // configure the copy to launch at the next LCD tick
  UI_DRAWING_DMA2D_configure_memcpy(LLUI_
↪DISPLAY_getBufferAddress(&gc->image), (uint8_t*)LTDC_Layer->CFBAR, regions[0].
↪x1, regions[0].y1, regions[0].x2, regions[0].y2, RK043FN48H_WIDTH, &dma2d_memcpy);

  // ask an interrupt for the next LCD tick
  lcd_enable_interrupt();
}

void LTDC_IRQHandler(LTDC_HandleTypeDef *hltdc) {
  // clear interrupt flag
  LTDC->ICR = LTDC_IER_FLAG;

  // launch the copy from the back buffer to the front buffer
  UI_DRAWING_DMA2D_start_memcpy(&dma2d_memcpy);
}

void DMA2D_IRQHandler(void) {
  // call CCO DMA2D function
  UI_DRAWING_DMA2D_IRQHandler();
}

void UI_DRAWING_DMA2D_memcpy_callback(bool from_isr) {
  // notify the MicroUI Graphics Engine
```

(continues on next page)

(continued from previous page)

```
  LLUI_DISPLAY_
↪setBackBuffer(g_current_flush_identifier, (uint8_t*)BACK_BUFFER, from_isr);
}
```

Buffer Refresh Strategy "Legacy"

This strategy requires copying the previous drawings from the front buffer to the back buffer before unlocking the MicroUI Graphics Engine. To perform this copy, this CCO uses the UI_DRAWING_DMA2D_xxx_memcpy() functions. At the end of the copy, the MicroUI Graphics Engine is unlocked: a new drawing can be performed in the new back buffer.

Usage

1. The function UI_DRAWING_DMA2D_configure_memcpy() must be called from the implementation of LLUI_DISPLAY_IMPL_flush() .
2. The function UI_DRAWING_DMA2D_start_memcpy() must be called from the LCD controller IRQ routine.
3. The function UI_DRAWING_DMA2D_IRQHandler() must be called from the DMA2D IRQ routine.
4. The function UI_DRAWING_DMA2D_memcpy_callback() must be implemented to unlock the MicroUI Graphics Engine.

Example of Implementation

```
void LLUI_DISPLAY_IMPL_flush(MICROUI_
↪GraphicsContext* gc, uint8_t flush_identifier, const ui_rect_t regions[], size_t length) {

  // store the flush identifier
  g_current_flush_identifier = flush_identifier;

  // configure the copy to launch at the next LCD tick
  UI_DRAWING_DMA2D_configure_memcpy(LLUI_
↪DISPLAY_getBufferAddress(&gc->image), (uint8_t*)LTDC_Layer->CFBAR, regions[0].
↪x1, regions[0].y1, regions[0].x2, regions[0].y2, RK043FN48H_WIDTH, &dma2d_memcpy);

  // change the front buffer address
  HAL_LTDC_SetAddress(&hLtdcHandler,
↪ (uint32_t)LLUI_DISPLAY_getBufferAddress(&gc->image), LTDC_ACTIVE_LAYER);

  // ask an interrupt for the next LCD tick
  lcd_enable_interrupt();
}

void HAL_LTDC_ReloadEventCallback(LTDC_HandleTypeDef *hltdc) {
  // LTDC register reload
  __HAL_LTDC_ENABLE_IT(hltdc, LTDC_IT_RR);

  // launch the copy from the new front buffer to the new back buffer
  UI_DRAWING_DMA2D_start_memcpy(&dma2d_memcpy);
```

(continues on next page)

(continued from previous page)

```
}

void DMA2D_IRQHandler(void) {
  // call CCO DMA2D function
  UI_DRAWING_DMA2D_IRQHandler();
}

void UI_DRAWING_DMA2D_memcpy_callback(bool from_isr) {
  // notify the MicroUI Graphics Engine
  uint8_t* buffer = (uint8_t*)(BACK_
↪BUFFER == LTDC_Layer->CFBAR ? FRAME_BUFFER : BACK_BUFFER);
  LLUI_DISPLAY_setBackBuffer(g_current_flush_identifier, buffer, from_isr);
}
```

C Module: MicroUI Over VGLite

Overview

This C module is a specific implementation of the C module MicroUI over the VGLite library 3.0.15_rev7:

- It implements a set of drawings over the official VGLite library 3.0.15_rev7.
- It is compatible with the *multiple destination formats* module.

This C module also provides a set of header files (and their implementations) to manipulate some MicroUI concepts over the VGLite library: image management, path format, etc.: ui_vglite.h and ui_drawing_vglite_path.h .

This C module is available on the *Developer Repository*: com.microej.clibrary.llimpl#microui-vglite.

Files

- Implements some functions of ui_drawing.h (see above).
- C files: mej_math.c , ui_drawing_vglite_path.c , ui_drawing_vglite_process.c , ui_drawing_vglite.c and ui_vglite.c .
- Status: optional.

Usage

1. Install the *C Module for MicroUI* and follow its implementation rules.
2. Add the C files to the BSP project.
3. Call UI_VGLITE_initialize from LLUI_DISPLAY_IMPL_initialize before calling any VGLite-related function.
4. Call UI_VGLITE_start from LLUI_DISPLAY_IMPL_initialize after configuring the VGLite library.
5. Configure the options in ui_vglite_configuration.h .

6. Comment the line `#error [...]"`.
7. Call `UI_VGLITE_IRQHandler` during the GPU interrupt routine.
8. Set the VGLite library's preprocessor define `VG_DRIVER_SINGLE_THREAD`.
9. The VGLite library must be patched to be compatible with this C module:

```
cd [...]/sdk/middleware/vglite
patch -p1 < [...]/3.0.15_rev7.patch
```

10. In the file `vglite_window.c`, add the function `VGLITE_CancelSwapBuffers()` and its prototype in `vglite_window.h`:

```
void VGLITE_CancelSwapBuffers(void) {
   fb_idx = fb_idx == 0 ? (APP_BUFFER_COUNT - 1) : (fb_idx ) - 1;
}
```

11. Check the port by running the `ui` validation as described in the VEE Port project template

Options

This C module provides some drawing algorithms that are disabled by default.

- The rendering time of a simple shape with the GPU (time in the VGLite library + GPU setup time + rendering time) is longer than with software rendering. To enable the hardware rendering for simple shapes, uncomment the definition of `VGLITE_USE_GPU_FOR_SIMPLE_DRAWINGS` in `ui_vglite_configuration.h`.
- The rendering time of an RGB565 image into an RGB565 buffer without applying an opacity (alpha == 0xff) is longer than with software rendering (as this kind of drawing consists in performing a memory copy). To enable the hardware rendering for RGB565 images, uncomment the definition of `VGLITE_USE_GPU_FOR_RGB565_IMAGES` in `ui_vglite_configuration.h`.
- ARGB8888, ARGB1555, and ARGB4444 transparent images may not be compatible with some revisions of the VGLite GPU. Older GPU revisions do not render transparent images correctly because the pre-multiplication of the pixel opacity is not propagated to the pixel color components. To force the hardware rendering for non-premultiplied transparent images when the VGLite GPU is not compatible, uncomment the definition of `VGLITE_USE_GPU_FOR_TRANSPARENT_IMAGES` in `ui_vglite_configuration.h`. Note that this limitation does not concern the VGLite GPU, which is compatible with non-premultiplied transparent images and the A8/A4 formats.

Drawings

The following table describes the accelerated drawings:

Feature	Comment
Draw line	Disabled by default (see above)
Fill rectangle	Disabled by default (see above)
Draw rounded rectangle	Disabled by default (see above)
Fill rounded rectangle	
Draw circle arc	Disabled by default (see above)
Fill circle arc	
Draw ellipse arc	Disabled by default (see above)
Fill ellipse arc	
Draw ellipse arc	Disabled by default (see above)
Fill ellipse arc	
Draw circle	Disabled by default (see above)
Fill circle	
Draw image	ARGB8888_PRE, ARGB1555_PRE, ARGB4444_PRE, RGB565, A8, A4 ARGB8888, ARGB1555, ARGB4444 (see above)
Draw thick faded point	Only with fade <= 1
Draw thick faded line	Only with fade <= 1
Draw thick faded circle	Only with fade <= 1
Draw thick faded circle arc	Only with fade <= 1
Draw thick faded ellipse	Only with fade <= 1
Draw thick line	
Draw thick circle	
Draw thick circle arc	
Draw thick ellipse	
Draw flipped image	See draw image
Draw rotated image	See draw image
Draw scaled image	See draw image

Compatibility With MCU i.MX RT595

UI Pack 13

The versions of the C Module Over VGLite (before 7.0.0) included an implementation of the Low-Level API LLUI_DISPLAY_impl.h . This support has been extracted into a dedicated C Module since the version 7.0.0 . The dedicated C Module is available on the *Developer Repository*: com.microej.clibrary.llimpl#microui-mimxrt595-evk.

Only the C Module com.microej.clibrary.llimpl#microui-vglite is useful to target the Vivante VGLite GPU to perform the MicroUI and MicroVG drawings. The C Module com.microej.clibrary.llimpl#microui-mimxrt595-evk only gives an example of an implementation compatible with the MCU i.MX RT595 MCU.

Note: For more information, see the *migration notes*.

UI Pack 14

Since UI Pack 14, this C module is not compatible anymore and is not maintained.

C Module: MicroUI Over NemaGFX

Overview

This C module is a specific implementation of the C module MicroUI over the Think Silicon NemaGFX:

- It implements a set of drawings over the official Think Silicon NemaGFX.
- It is compatible with the *multiple destination formats* module (but it can only handle one destination format).

This C module is available on the *Developer Repository*: com.microej.clibrary.llimpl#microui-nemagfx.

Files

- Implements some functions of `ui_drawing.h` (see above).
- C files: `ui_nema.c` and `ui_drawing_nema.c`.
- Status: optional.

Usage

1. Install the *C Module for MicroUI* and follow its implementation rules.
2. Add the C files to the BSP project.
3. Add `ui_nemagfx/inc` to the include path.
4. Call `UI_NEMA_initialize()` from `LLUI_DISPLAY_IMPL_initialize()`.
5. Configure the options in `ui_nema_configuration.h`.
6. Comment the line `#error [...]"`.
7. Check the port by running the `ui` validation as described in the VEE Port project template

Implementation

The MicroUI Graphics Engine waits for the end of the asynchronous drawings (performed by the GPU). The VEE Port must stop this wait with a call to the function `UI_NEMA_post_operation()` in the GPU interrupt routine.

Tip: The GPU interrupt routine is often written in the same file as the implementation of `nema_sys_init()`.

Options

This C module provides some drawing algorithms that are disabled by default.

- The rendering time of a simple shape with the GPU (time in the NemaGFX library + GPU setup time + rendering time) is longer than with software rendering. To enable the hardware rendering for simple shapes, uncomment the definition of ENABLE_SIMPLE_LINES in ui_nema_configuration.h .
- The rendering of thick faded lines with the GPU is disabled by default: the quality of the rendering is too random. To enable it, uncomment the definition of ENABLE_FADED_LINES in ui_nema_configuration.h .
- Some GPUs might not be able to render the images in specific memories. Comment the define ENABLE_IMAGE_ROTATION in ui_nema_configuration.h to not use the GPU to render the rotated images.

Drawings

The following table describes the accelerated drawings:

Feature	Comment
Draw line	
Draw horizontal line	Disabled by default (see above: ENABLE_SIMPLE_LINES)
Draw vertical line	Disabled by default (see above: ENABLE_SIMPLE_LINES)
Draw rectangle	Disabled by default (see above: ENABLE_SIMPLE_LINES)
Fill rectangle	
Draw rounded rectangle	
Fill rounded rectangle	
Draw circle	
Fill circle	
Draw image	ARGB8888, RGB565, A8
Draw thick faded line	Only with fade <= 1
Draw thick faded circle	Only with fade <= 1
Draw thick line	Disabled by default (see above: ENABLE_FADED_LINES)
Draw thick circle	
Draw rotated image	See draw image
Draw scaled image	See draw image

Compatibility

The compatibility between the components (Packs, C modules, and Libraries) is described in the *C Modules*.

Simulation

Principle

The graphical user interface uses the Front Panel mock (see *Front Panel Mock*) and some extensions (widgets) to simulate the user interactions. It is the equivalent of the three embedded modules (Display, Input and LED) of the VEE Port (see *MicroUI*).

The Front Panel enhances the development environment by allowing User Interface applications to be designed and tested on the computer rather than on the target device (which may not yet be built). The mock interacts with the user's computer in two ways:

- output: LEDs, graphical displays
- input: buttons, joystick, touch, haptic sensors

Note: This chapter completes the notions described in *Front Panel Mock* chapter.

Module Dependencies

The Front Panel project is a regular MicroEJ Module project. By default, the project depends on the Front Panel Framework which only contains the Front Panel core classes and which does not provide any Front Panel Widgets (see *Module Dependencies*). To add interactive Front Panel Widgets (typically a simulated display and input devices), add the library that provides compatible Front Panel Widgets with the Graphics Engine:

SDK 6

SDK 5

The Front Panel project is a regular Gradle project. Its dependencies in `build.gradle.kts` file should look like this example:

```
dependencies {
  implementation("ej.tool.frontpanel:widget:[Front Panel Widgets version]")
}
```

The Front Panel project is a regular MicroEJ Module project. Its dependencies in `module.ivy` file should look like this example:

```
<dependencies>
  <dependency␣
↪org="ej.tool.frontpanel" name="widget" rev="[Front Panel Widgets version]"/>
</dependencies>
```

Note: The life cycle of this library is different than the UI pack's one, see *Front Panel API*. The latest version of the Front Panel Widgets is 5.3.2.

To implement UI Pack extensions for the simulator (custom widgets compatible with the Graphics Engine, custom drawings, etc.), add the Front Panel extension API from the UI Pack (set the version used by the VEE Port):

SDK 6

SDK 5

The Front Panel project is a regular Gradle project. Its dependencies in build.gradle.kts file should look like this example:

```
dependencies {
  implementation("com.microej.pack.ui:ui-pack:[UI Pack version]") {
    artifact {
        name = "frontpanel"
        extension = "jar"
    }
  }
}
```

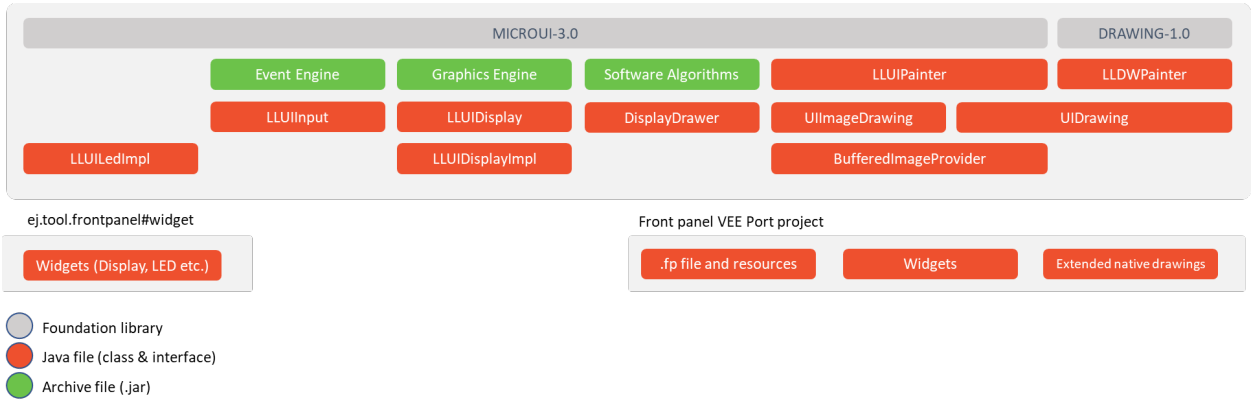
The Front Panel project is a regular MicroEJ Module project. Its dependencies in module.ivy file should look like this example:

```
<dependency org="com.microej.pack.ui" name="ui-pack" rev="[UI Pack version]">
  <artifact name="frontpanel" type="jar"/>
</dependency>
```

Warning: This extension is built for each UI Pack version. By consequence, a Front Panel project is done for a VEE Port built with the same UI Pack. When the UI Pack mismatch, some errors may occur during the Front Panel project export step, during the VEE Port build, and/or during the application runtime. The latest version of the UI Pack is 14.5.2.

MicroUI Implementation

As described *here*, the Front Panel uses an equivalent of embedded side's header files that implement MicroUI native methods.



This set of classes and interfaces is available in the module com.microej.pack.ui#ui-pack. It offers the same capacity to override some built-in drawing algorithms (internal Graphics Engine drawing algorithms), to add some custom drawing algorithms, to manipulate the MicroUI concepts (GraphicsContext, Image, etc.) in the Front Panel project, etc.

- The interface ej.microui.display.LLUIDisplay represents the MicroUI Graphics Engine (MicroUI framework). It provides methods to map MicroUI byte arrays in MicroUI Graphics Context

objects, manipulate MicroUI colors, clip, etc. An instance of this framework is available via the field Instance .

- The interface ej.microui.display.LLUIDisplayImpl all methods required by MicroUI implementation to be compatible with the MicroUI Display class implementation. See *Display Widget*.
- The class ej.microui.display.LLUIPainter implements all MicroUI drawing natives. It defines some interfaces and classes to manipulate the MicroUI concepts (GraphicsContext, Image, etc.) in the Front Panel project. Like the embedded side, this class manages the synchronization with the Graphics Engine and delegates the drawing to the interface ej.microui.display.UIDrawing .
- The interface ej.microui.display.UIDrawing defines all the drawing methods available in MicroUI. The default implementation of the methods involving images calls the matching method in ej.microui.display.UIImageDrawing . The default implementation of the other methods reports the error that the drawing is not done.
- The interface ej.microui.display.UIImageDrawing defines all the methods that draw an image. The default implementation of the methods reports the error that the drawing is not done.
- The class ej.microui.display.DisplayDrawer implements ej.microui.display.UIDrawing that draws using the Graphics Engine software algorithms.
- The classes in the package ej.drawing implement the native of the MicroUI extended library: Drawing
- The classes in the package ej.microui.event manage the input events, see *Inputs Extensions*.
- The classes in the package ej.microui.led manage the LEDs.

Display Widget

The Display widget implements the interface ej.microui.display.LLUIDisplayImpl to be compatible with the implementation of the MicroUI class Display.

Features

- *Display buffer policy and buffer refresh strategy*: simulates the display buffer policy and the buffer refresh strategy.
- *LCD refresh rate*: simulates the time between two visible frames on the hardware device.
- *LCD flush time*: simulates the time to flush the frame content to the hardware device.
    - Backlight (enabled by default): backlightFeature=true|false .
    - *Non-rectangular displays*: filter="xxx.png" . Some displays can have another appearance (for instance: circular).
    - *Standard* pixel formats.
    - *Driver-specific* pixel formats: extensionClass="xxx" . This class must be added in the Front Panel project and implement the interface ej.fp.widget.Display.DisplayExtension .

Refresh Rate

Usually a LCD is cadenced around 50-60Hz. That means the LCD can display a new frame every 16-20ms. By default this widget displays a new frame as soon as possible. It can be configured to reduce this time to simulate the hardware device.

In the widget declaration, set the attribute `refreshRate="xxx"` with a value in Hertz. A zero or negative value disables the feature.

The application can substitute the VEE Port's value by setting the property `-Dej.fp.widget.display.refreshRate=xxx` in the application launcher.

Flush Time

On a hardware device, the time to flush the frame data from the back buffer memory to the LCD is not null. According to the hardware device technology, this time varies between 3-4 ms to 10-15ms. In SPI mode, this time may be higher, around 50ms, even more. By default this widget copies the content of back buffer as faster as possible. It can be configured to reduce this time to simulate the hardware device.

In the widget declaration, set the attribute `flushTime="xxx"` with a value in milliseconds. A zero or negative value disables the feature.

The application can substitute the VEE Port's value by setting the property `-Dej.fp.widget.display.flushTime=xxx` in the application launcher.

Non-rectangular Display

The Front Panel can simulate using a filter (see *Widget*). This filter defines the pixels inside and outside the whole display area. The filter image must have the same size as the rectangular display area. A display pixel at a given position will not be rendered if the pixel at the same position in the mask is fully transparent.

Note: Usually the touch panel over the display uses the same filter to reduce the touch panel area.

Example of non-rectangular display and touch:

```
<ej.fp.widget.Display x="41" y="33" width="392" height="392" filter="mask_392.png" />
<ej.fp.widget.
↪Pointer x="41" y="33" width="392" height="392" filter="mask_392.png" touch="true"/>
```

Inputs Extensions

The input device widgets (button, joystick, touch, etc.) require a listener to know how to react on input events (press, release, move, etc.).The aim of this listener is to generate an event compatible with MicroUI Event Generator. Thereby, a button press action can become a MicroUI Buttons press event or a Command event or anything else.

A MicroUI Event Generator is known by its name. This name is fixed during the MicroUI static initialization (see *Static Initialization*). To generate an event to a specific event generator, the widget has to use the event generator name as identifier.

A Front Panel widget can:

- Force the behavior of an input action: the associated MicroUI Event Generator type is hardcoded (Buttons, Pointer, etc.), the event is hardcoded (for instance: widget button press action may be hardcoded on event generator Buttons and on the event *pressed*). Only the event generator name (identifier) should be editable by the Front Panel extension project.
- Propose a default behavior of an input action: contrary to first point, the Front Panel extension project is able to change the default behavior. For instance a joystick can simulate a MicroUI Pointer.
- Do nothing: the widget requires the Front Panel extension project to give a listener. This listener will receive all widgets action (press, release, etc.) and will have to react on it. The action should be converted on a MicroUI Event Generator event or might be dropped.

  This choice of behavior is widget dependant. Please refer to the widget documentation to have more information about the chosen behavior.

Heap Simulation

Graphics Engine is using two dedicated heaps: for the images (see *Images Heap*) and the external fonts (see *External Resources*). Front Panel partly simulates the heaps usage.

- Images heap: Front Panel simulates the heap usage when the application is creating a BufferedImage, when it loads and decodes an image (PNG, BMP, etc.) which is not a raw resource and when it converts an image in MicroEJ format in another MicroEJ format. However it does not simulate the external image copy in heap (see *External Resource*).
- External fonts heap: Front Panel does not simulate this heap (see *External Resources*). There is no rendering limitation when application is using a font which is located outside CPU addresses ranges.

Image Decoders

Front Panel uses its own internal image decoders when the associated modules have been selected (see *internal image decoders*). Some additional decoders can be added like the C-side for the embedded VEE Port (see *external image decoders*).Front Panel uses the Java AWT ImageIO API to load the encoded images.

Generic Image Decoders

The Java AWT ImageIO class holds a limited list of additional decoders. To be compliant with the embedded side, these decoders are disabled by default. To add an additional decoder, specify the property `hardwareImageDecoders.list` in Front Panel configuration properties file (see *Installation*) with one or several property values:

Table 83: Front Panel Additional Image Decoders

Type	Property value
Graphics Interchange Format (GIF)	gif
Joint Photographic Experts Group (JPEG)	jpeg or jpg
Portable Network Graphics (PNG)	png
Windows bitmap (BMP)	bmp

The decoders list is comma (,) separated. Example:

```
hardwareImageDecoders.list=jpg,bmp
```

Custom Image Decoders

Additionally, the Java AWT ImageIO class offers the possibility to add some custom image decoders by using the service `javax.imageio.spi.ImageReaderSpi`.

Since UI Pack 13.2.0, Front Panel automatically includes new image decoders (new ImageIO services, see the method `LLUIDisplayImpl.decode()`), compiled in JAR files that follow this convention:

1. The JAR contains the service declaration `/META-INF/services/javax.imageio.spi.ImageReaderSpi`,
2. The JAR filename's prefix is *imageio-*,
3. The JAR location is the VEE Port configuration project's `dropins/tools/` directory.

Note: The same JAR is used by the Front Panel and by the *Image Generator*.

Drawings

Front Panel is designed to modify the default behavior for performing *drawings*.

Image Rendering

Front Panel is designed to add the support of *custom images*.

Buffered Image

Front Panel is designed to add the support of *MicroUI BufferedImage* with a format different from the display format.

Classpath

A standard mock is running on the same JVM than the HIL Engine (see *Mock* chapter). It shares the same classpath. When the application is not using the MicroUI library (i.e., it is not an UI application, whether the VEE Port holds the MicroEJ Graphics Engine or not), the Front Panel mock runs a standard mock. When the application is using the MicroUI library, the Front Panel UI mock runs on the same JVM than the MicroEJ Simulator. In this case, the other mocks don't share the same classpath than the Front Panel mock. As a consequence, an other mock than the Front Panel mock can not send input events to MicroUI, the object created in the standard mocks's class loader are not available in the Front Panel UI's class loader (and vice versa), etc.

Since the UI Pack 13.2.0, it is possible to force to run the Front Panel UI mock in the same classpath than the HIL Engine by adding the property `-Dej.fp.hil=true` in the application JRE tab. Note that this option only works when the version of the MicroEJ Architecture used to build the VEE Port is 7.17.0 or higher.

Dependencies

- MicroUI module (see *MicroUI*),
- Display module (see *Display*): This module gives the characteristics of the graphical display that are useful for configuring the Front Panel.

Installation

Front Panel is an additional module for MicroUI library. When the MicroUI module is installed, install this module in order to be able to simulate UI drawings on the Simulator. See *Installation* to install the module.

The properties file can additional properties:

- hardwareImageDecoders.list [optional, default value is "" (*empty*)]: Defines the available list of additional image decoders provided by the hardware (see *Image Decoders*). Use comma (',') to specify several decoders among this list: bmp, jpg, jpeg, gif, png. If empty or unspecified, no image decoder is added.

  SDK 6

  SDK 5

  In SDK 6, the configuration is done in the properties file vee-port/configuration.properties of the VEE Port project.

  Define the property using the com.microej.pack.frontpanel prefix: com.microej.pack.frontpanel.hardwareImageDecoders.list=jpg,bmp,png

  In SDK 5, set the hardwareImageDecoders.list property in the properties file frontpanel/frontpanel.properties .

Use

Launch a MicroUI application on the Simulator to run the Front Panel.

Release Notes

MicroEJ Architecture Compatibility Version

The current UI Pack version is 14.5.2. The following tables describe the compatibility ranges between MicroEJ UI Packs and MicroEJ Architectures.

Standard Versions

UI Pack Range	Architecture Range	Comment
14.5.2	[7.16.0-9.0.0[	Compatibility with Architecture 8.6.0
[13.5.0-14.5.1]	[7.16.0-8.6.0[	Compatibility with Architecture 8
[13.0.0-13.4.1]	[7.16.0-8.0.0[	SNI 1.3
[12.0.0-12.1.5]	[7.11.0-8.0.0[	Move Front Panel in MicroEJ Architecture
[11.0.0-11.2.0]	[7.0.0-8.0.0[	SNI Callback feature
[9.3.1-10.0.2]	[6.13.0-7.0.0[	LLEXT link error with Architecture 6.13+ and UI 9+
[9.2.0-9.3.0]	[6.12.0-6.13.0[	SOAR can exclude some resources
[9.1.0-9.1.2]	[6.8.0-6.12.0[	Internal scripts
[8.0.0-9.0.2]	[6.4.0-6.12.0[	Manage external memories like byte addressable memories
[6.0.0-7.4.7]	[6.1.0-6.12.0[	

Maintenance Versions

UI Pack Version	UI Pack Base Version	Architecture Range	Comment
(maint) 8.0.0	7.4.7	[7.0.0-8.0.0[	SNI Callback feature

Foundation Libraries

The following table describes Foundation Libraries API versions implemented in MicroEJ UI Packs.

Table 84: MicroUI API Implementation

UI Pack Range	MicroUI	Drawing
[14.1.1-14.5.2]	3.6.0	1.0.4
[14.0.0-14.0.3]	3.5.0	1.0.4
[13.7.0-13.7.2]	3.4.0	1.0.4
[13.6.0-13.6.2]	3.3.0	1.0.4
[13.5.0-13.5.1]	3.2.0	1.0.4
[13.2.0-13.4.1]	3.1.1	1.0.4
13.1.0	3.1.0	1.0.3
[13.0.4-13.0.7]	3.0.3	1.0.2
13.0.3	3.0.2	1.0.1
[13.0.1-13.0.2]	3.0.1	1.0.0
13.0.0	3.0.0	1.0.0
[12.1.0-12.1.5]	2.4.0	
[11.1.0-11.2.0]	2.3.0	
[9.2.0-11.0.1]	2.2.0	
[9.1.1-9.1.2]	2.1.3	
9.1.0	2.1.2	
[9.0.0-9.0.2]	2.0.6	
[6.0.0-8.1.0]	2.0.0	

Abstraction Layer Interface

The following sections briefly describe Abstraction Layer interface changes (the functions the BSP has to implement). For more details, refer to the *Migration Guide*.

Note: In addition of these functions, some C modules must be added to the BSP, see *C Modules*.

Display

UI Pack Range	Changes
[14.0.0-14.5.2]	Signature of LLUI_DISPLAY_IMPL_flush() changed.
[13.0.0-13.7.2]	*UI3* format: implement LLUI_DISPLAY_impl.h : • void LLUI_DISPLAY_IMPL_initialize([...]); • void LLUI_DISPLAY_IMPL_binarySemaphoreTake([...]); • void LLUI_DISPLAY_IMPL_binarySemaphoreGive([...]); • uint8_t* LLUI_DISPLAY_IMPL_flush([...]);
[10.0.0-12.1.5]	Remove: • int32_t LLDISPLAY_IMPL_getWorkingBufferStartAddress([...]); • int32_t LLDISPLAY_IMPL_getWorkingBufferEndAddress([...]);
[8.0.0-9.4.1]	Merge in LLDISPLAY_impl.h : • LLDISPLAY_SWITCH_impl.h • LLDISPLAY_COPY_impl.h • LLDISPLAY_DIRECT_impl.h
[6.0.0-7.4.7]	*UI2* format: implement one of header file: • LLDISPLAY_SWITCH_impl.h • LLDISPLAY_COPY_impl.h • LLDISPLAY_DIRECT_impl.h

Input

UI Pack Range	Changes
[13.0.0-14.5.2]	*UI3* format: implement LLUI_INPUT_impl.h : • void LLUI_INPUT_IMPL_initialize([...]); • jint LLUI_INPUT_IMPL_getInitialStateValue([...]); • void LLUI_INPUT_IMPL_enterCriticalSection([...]); • void LLUI_INPUT_IMPL_leaveCriticalSection([...]);
[6.0.0-12.1.5]	*UI2* format: implement LLINPUT_impl.h • void LLINPUT_IMPL_initialize([...]); • int32_t LLINPUT_IMPL_getInitialStateValue([...]); • void LLINPUT_IMPL_enterCriticalSection([...]); • void LLINPUT_IMPL_leaveCriticalSection([...]);

LED

UI Pack Range	Changes
[13.0.0-14.5.2]	*UI3* format: implement LLUI_LED_impl.h : • jint LLUI_LED_IMPL_initialize([...]); • jint LLUI_LED_IMPL_getIntensity([...]); • void LLUI_LED_IMPL_setIntensity([...]);
[6.0.0-12.1.5]	*UI2* format: implement LLLEDS_impl.h • int32_t LLLEDS_IMPL_initialize([...]); • int32_t LLLEDS_IMPL_getIntensity([...]); • void LLLEDS_IMPL_setIntensity([...]);

Front Panel API

The Front Panel project must fetch the widgets compatible with the MicroEJ UI Pack fetched in the VEE Port configuration project:

- Before MicroEJ UI Pack 12.0.0 , the Front Panel project must depend on the classpath variable FRONTPANEL_WIDGETS_HOME .
- For the UI Packs 12.x.x , the Front Panel project must fetch the module ej.tool.front-panel.widget-microui.
- Since MicroEJ UI Pack 13.0.0 , the Front Panel project must depend on the module com.microej.pack.ui.ui-pack(frontpanel) (the module version is the MicroEJ Generic UI Pack version, that is always aligned with the MicroEJ UI Packs specific for MCUs).

UI Pack Range	Module	Version
13.0.0 and above	com.microej.pack.ui.ui-pack(frontpanel)	Identical to the UI Pack
[12.0.0-12.1.5]	ej.tool.frontpanel.widget-microui	1.0.0
[6.0.0-11.2.0]	n/a	n/a

The widget module ej.tool.frontpanel.widget provides some widgets compatible with the Graphics Engine. This module fetches by transitivity the module com.microej.pack.ui.ui-pack(frontpanel). When the Front Panel project does not require/use the latest Front Panel UI API, it can only fetch the widget module.

Note: This module has been moved from the MicroEJ Central Repository to the MicroEJ Developer Repository.

Widget Module Range	UI Pack Compatibility Range	Repository
[5.3.1-5.3.2]	[14.4.1-14.5.2]	Developer
[5.1.0-5.2.0]	[14.3.1-14.3.3]	Developer
5.0.0	14.3.0	Developer
[4.0.0-4.0.2]	[14.0.0-14.2.0]	Developer
3.0.0	[13.5.1-10-13.7.2]	Developer
2.2.0	[13.1.0-13.7.2]	Developer
[2.1.0-2.1.1]	[13.1.0-13.7.2]	Central
2.0.0	[13.0.0-13.7.2]	Central
1.0.1	[12.0.0-12.1.5]	Developer

To use the latest functionalities provided by the UI Pack 13.0.0 and higher, the Front Panel project must depend on the same version of the UI Pack as the VEE Port configuration project. However, if the Front Panel project does not require/use the latest Front Panel UI API, it can fetch a version of the UI Pack older than the version fetched in the VEE Port configuration project.

Image Generator API

Since MicroEJ UI Pack 13.0.0 , the Image Generator extension project must depend on module com.microej.pack.ui.ui-pack(imagegenerator). The module version is the MicroEJ Generic UI Pack version, that is always aligned with the MicroEJ UI Packs specific for MCUs.

UI Pack Range	Module	Version
13.0.0 and above	com.microej.pack.ui.ui-pack(imagegenerator)	Identical to the UI Pack

Note: Before MicroEJ UI Pack 13.0.0 , the Image Generator extension project must depend on classpath variable IMAGE-GENERATOR-x.x .

C Modules

MicroUI C Module

The MicroUI C module com.microej.clibrary.llimpl(microui) is available on MicroEJ Central Repository, see *C Modules*.

Note: Since version 14.1.1 of UI Pack, UI Pack and MicroUI C Module have the same version.

The following table describes the compatibility versions between the MicroEJ UI Packs and the C modules:

UI Pack	MicroUI C Module	Comment
14.1.1 and above	Identical to the UI Pack	
[14.0.0-14.0.3]	[4.0.0-4.1.0]	Buffer refresh strategies
[13.7.0-13.7.2]	[3.1.0-3.1.1]	Free image resources
[13.5.0-13.6.2]	3.0.0	Multiple Graphics Context output formats
[13.3.0-13.4.1]	[2.0.0-2.0.1]	Copy and draw image
[13.1.0-13.2.0]	[1.1.0-1.1.1]	Image heap, events queue, drawing limits
[13.0.0-13.0.7]	1.0.3	

Extended C Modules

Some C modules extend the main MicroUI C module. They override the default implementation to use a GPU to perform some drawings. Contrary to the main MicroUI C module, they are optional: when they are not available, the default implementation of drawings is used. The default implementation uses the Graphics Engine software algorithms.

STM32 Chrom-ART

The *DMA2D C module* targets the STM32 CPU that provides the Chrom-ART accelerator.

Note: Since version 6.0.0, this module has been moved from the MicroEJ DMA2D_Central Repository to the MicroEJ DMA2D_Developer Repository.

The following table describes the version compatibility between the MicroEJ UI Packs and the C modules:

UI Pack	Chrom-ART	MicroUI C Module	Comment
[14.5.1-14.5.2]	8.0.0	Identical to the UI Pack	New options convention
[14.1.1-14.4.2]	[7.0.0-7.0.1]	Identical to the UI Pack	Font extensibility
[14.0.1-14.0.3]	[5.0.1-6.0.0]	[4.0.1-4.1.0]	C modules harmonization
14.0.0	5.0.0	4.0.0	Buffer refresh strategies
[13.7.0-13.7.2]	4.1.0	[3.1.0-3.1.1]	Free image resources
[13.5.1-13.6.2]	4.0.0	3.0.0	Multiple Graphics Context output formats
[13.3.0-13.4.1]	[3.0.0-3.0.2]	[2.0.0-2.0.1]	Copy and draw image
[13.1.0-13.2.0]	[2.0.0-2.1.0]	[1.1.0-1.1.1]	Drawing limits
[13.0.0-13.0.7]	[1.0.6-1.0.8]	1.0.3	

Vivante VGLite

The *VGLite C module* targets the NXP CPU that provides the Vivante VGLite accelerator.

The following table describes the version compatibility between the MicroEJ UI Packs and the C modules:

UI Pack	VGLite	MicroUI C Module	Comment
[14.5.1-14.5.2]	11.0.0	Identical to the UI Pack	New options convention
[14.1.1-14.4.2]	[10.0.0-10.0.1]	Identical to the UI Pack	Font extensibility
[14.0.1-14.0.3]	9.0.0	4.1.0	VG Pack extensibility
[14.0.1-14.0.3]	8.0.1	4.0.1	C modules harmonization
14.0.0	8.0.0	4.0.0	Buffer refresh strategies
[13.7.0-13.7.2]	7.2.0	[3.1.0-3.1.1]	Free image resources
[13.5.1-13.6.2]	[6.0.0-7.1.0]	3.0.0	Multiple Graphics Context output formats
[13.3.0-13.4.1]	[3.0.0-5.0.1]	[2.0.0-2.0.1]	

The following table describes the version compatibility between the C module and the VGLite libraries (officially supported):

VGLite C Module	VGLite Libraries Range
[8.0.0-11.0.0]	3.0.15_rev7
[7.1.0-7.2.0]	3.0.15_rev4 and 3.0.15_rev7
[4.0.0-7.0.0]	3.0.15_rev4
[2.0.0-3.0.0]	3.0.11_rev3
1.0.0	3.0.4_rev2 and 3.0.4_rev4

Think Silicon NemaGFX

The *NemaGFX C module* targets the CPU that provides the NemaGFX accelerator.

The following table describes the version compatibility between the MicroEJ UI Packs and the C modules:

UI Pack	NemaGFX	MicroUI C Module	Comment
[14.5.1-14.5.2]	5.0.0	Identical to the UI Pack	New options convention
[14.1.1-14.4.2]	[4.0.0-4.0.1]	Identical to the UI Pack	Font extensibility
[14.0.1-14.0.3]	3.0.0	4.1.0	VG Pack extensibility
[14.0.1-14.0.3]	2.0.1	4.0.1	C modules harmonization
14.0.0	2.0.0	4.0.0	Buffer refresh strategies
13.7.2	1.2.0	[3.1.0-3.1.1]	Update of configuration file
13.7.0	1.1.0	[3.1.0-3.1.1]	Free image resources
[13.5.1-13.6.2]	1.0.0		

The following table describes the version compatibility between the C module and the NemaGFX libraries (officially supported):

NemaGFX C Module	NemaGFX Libraries Range
[4.0.0-5.0.0]	[1.4.8-1.4.11]
[1.0.0-3.0.0]	1.4.8

Changelog

[14.5.2] - 2026-02-13

Architecture

Fixed

- Make the UI Pack compatible with the MicroEJ Architecture 8.6.0 or higher.

MicroUI

Changed

- The BSP has the responsibility to register the images created thanks the LLAPI LLUI_DISPLAY_allocateImageBuffer() and LLUI_DISPLAY_decodeImage() .

Fixed

- Fix the restoration of the kernel event handler after stopping a feature.
- Fix external image management: a "load" in the image heap was systematically performed, whereas a "get" may be sufficient (considering memory byte addressability and memory alignment).
- Fix the SNI registering of *converted* images (only used when killing a feature):
  - a RAW image opened with another format than the RAW image format
  - a decoded image with an output format not managed by the decoder

Removed

- Remove the systematic SNI registering of BSP images (created thanks to the LLAPI LLUI_DISPLAY_allocateImageBuffer() and LLUI_DISPLAY_decodeImage()).

LLAPIs

Added

- Add the API LLUI_DISPLAY_waitAsynchronousDrawingEnd() to wait the end of an asynchronous drawing before killing a feature.
- Add the API LLUI_DISPLAY_IMPL_notifyDrawingSoft() to be notified about a complete software drawing.
- Add the API LLUI_DISPLAY_testAndSetUserFlag() to test and set some user flags into a Graphics Context.

C Module MicroUI

Added

- Add ui_trace.h that replaces ui_log.h (ui_log.h is kept for backward compatibility).

Changed

- Harmonize the traces: trace the image's base address instead of the image's buffer address.
- Trace the image RAW format.

[14.5.1] - 2025-10-13

MicroUI

Fixed

- Fix Image.get() and Image.canGet() of external RAW images: the images must respect the memory constraints (byte addressable, alignment).
- Fix the loading of external RAW images that does not respect the GPU memory constraints.
- Fix the reading in the back buffer: it could read in the wrong buffer after a fast GPU draw after a flush.

LLAPIs

Added

- Add the API LLUI_DISPLAY_IMPL_[un]lockJob() to enable the use of re-entrant functions in a task other than the MicroJvm virtual machine task.
- Add the API LLUI_DISPLAY_decodeImage() to decode an image from the native world.
- Add the API LLUI_DISPLAY_isMutable() that tells if an image is mutable or not.
- Add the API LLUI_DISPLAY_getAddress() that gives the absolute address of an image.
- Add the API LLUI_DISPLAY_getSize() that gives the image size in bytes.

C Module MicroUI

Changed

- Use the include directive #include "veeport_configuration.h" to configure the UI configuration (ui_configuration.h).

Fixed

- Fix the log identifiers of the Drawing library.

C Module DMA2D

Changed

- Use the include directive `#include "veeport_configuration.h"` to configure the DMA2D configuration (`ui_dma2d_configuration.h`).

C Module VGLite

Changed

- Use the include directive `#include "veeport_configuration.h"` to configure the VGLite configuration (`ui_vglite_configuration.h`).

Fixed

- Fix the use of the VGLite option `VG_BLIT_WORKAROUND` : it must be `0` or undefined to avoid UI testsuite issues.

C Module NemaGFX

Changed

- Use the include directive `#include "veeport_configuration.h"` to configure the NemaGFX configuration (`ui_nema_configuration.h`)..

Fixed

- Fix the useless include directive.

[14.4.2] - 2025-05-20

Known Issues

- Some SystemView logs are not identified, example: `Function #586 Returns 0x0 after 169.844 us` .
  - Reason: The new C header file `ui_log.h` (in the C Module MicroUI) defines some invalid log identifiers:

```
#define UI_LOG_drawThickFadedPoint 70
[...]
#define UI_LOG_drawRotatedCharacter 95
```

  - Workaround: Change manually the log identifiers:

```
#define UI_LOG_drawThickFadedPoint 110
#define UI_LOG_drawThickFadedLine 111
#define UI_LOG_drawThickFadedCircle 112
#define UI_LOG_drawThickFadedCircleArc 113
#define UI_LOG_drawThickFadedEllipse 114
#define UI_LOG_drawThickLine 115
#define UI_LOG_drawThickCircle 116
#define UI_LOG_drawThickEllipse 117
#define UI_LOG_drawThickCircleArc 118
```

(continues on next page)

(continued from previous page)

```
#define UI_LOG_drawFlippedImage 130
#define UI_LOG_drawRotatedImage 131
#define UI_LOG_drawScaledImage 132
#define UI_LOG_drawScaledString 133
#define UI_LOG_drawScaledRenderableString 134
#define UI_LOG_drawRotatedCharacter 135
```

VG Pack

Fixed

- Fix the VEE Port build with the UI Pack 14.4 and the VG Pack 1.7 (since UI Pack 14.4.1).

FrontPanel

Fixed

- Fix the use of the SDK5 option “Resolve in workspace” (since UI Pack 14.4.1).

C Module MicroUI

Known Issues

- see above.

C Module DMA2D

Fixed

- Fix the implementation of `UI_DRAWING_DMA2D_copyImage()` to use the right software algorithm.

C Module VGLite

Fixed

- Fix the implementation of `UI_DRAWING_VGLITE_copyImage()` to use the right software algorithm.
- Fix the premultiplication of the format ARGB1555: it is useless.

C Module NemaGFX

Fixed

- Fix the rendering of a RGB565 image with a transparent opacity.
- Fix the implementation of `UI_DRAWING_NEMA_copyImage()` to use the right software algorithm.

[14.4.1] - 2025-03-13

Known Issues

- This version is not compatible with the VG Pack (any versions):
  - Reason: A VEE Port with this UI Pack and a VG Pack cannot be built:

```
[java] microvg/ui-pack:
[java] C:\[...]-configuration\
↪target~\scripts\rip\build\autoConfiguration-microvg.xml:43: VG Pack requires␣
↪the UI Pack 14.1.0 or higher and the platform fetches the UI Pack ${ui.packVersion}.
```

  - Workaround: None, use the UI Pack 14.3.3 instead.
- Some SystemView logs are not identified, example: `Function #586 Returns 0x0 after 169.844 us` .
  - Reason: The new C header file `ui_log.h` (in the C Module MicroUI) defines some invalid log identifiers:

```
#define UI_LOG_drawThickFadedPoint 70
[...]
#define UI_LOG_drawRotatedCharacter 95
```

  - Workaround: Change manually the log identifiers:

```
#define UI_LOG_drawThickFadedPoint 110
#define UI_LOG_drawThickFadedLine 111
#define UI_LOG_drawThickFadedCircle 112
#define UI_LOG_drawThickFadedCircleArc 113
#define UI_LOG_drawThickFadedEllipse 114
#define UI_LOG_drawThickLine 115
#define UI_LOG_drawThickCircle 116
#define UI_LOG_drawThickEllipse 117
#define UI_LOG_drawThickCircleArc 118
#define UI_LOG_drawFlippedImage 130
#define UI_LOG_drawRotatedImage 131
#define UI_LOG_drawScaledImage 132
#define UI_LOG_drawScaledString 133
#define UI_LOG_drawScaledRenderableString 134
#define UI_LOG_drawRotatedCharacter 135
```

MicroUI

Changed

- Improve the trace events (SystemView description file *SYSVIEW_MicroUI.txt* is updated).
- Change some log identifiers to be backward compatible with the UI Pack since version 12.1.5.

FrontPanel

Added

- Add a button to start/stop the recording of the drawings in the Flush Visualizer.
- Add the class `Log` that provides utility methods to log the drawings.
- Add `MicroUIImage.getIdentifier()` that identifies an image (useful for the logs).
- Add the same logs as embedded side (all drawings, out of clip, drawn regions, etc.).

Changed

- Improve the content of the tree of the drawing operations in the Flush Visualizer.
- Show the time between two frames in the Flush Visualizer.
- Use the core property `core.trace.enabled` to enable the Flush Visualizer.
- Do not draw if the clip is empty (align behavior between the simulator and the embedded side).

Fixed

- Fix the execution of the Front Panel in the HIL classpath (VEE Port with UI and application without UI; since UI Pack 14.3.0).
- Fix the use of the logs in the mode `s3.board.compliant` (Use target characteristics).

Removed

- Remove the use of the property `ej.fp.display.flushVisualizer` (replaced by `core.trace.enabled`).

C Module MicroUI

Added

- Add some logs: out of clip, drawn region, etc.

Changed

- Gather all logs identifiers and functions in `ui_log.h`.
- Log all drawing parameters.

Removed

- Remove some logs of the BRS management.

Known Issues

- see above.

[14.3.3] - 2025-02-18

FrontPanel

Fixed

- Fix the five issues of the UI Pack 14.3.2 (see below).

[14.3.2] - 2025-02-10

FrontPanel

Fixed

- Fix the performance issue of LLUIDisplay.convertRegion() when using the Front Panel Framework 1.3.0.

Known Issues

> **Warning:** The following issues concern the UI Packs **14.3.0** and **14.3.2**.

Front Panel

- On non-rectangular displays (see *Display Widget*), the Front Panel widget display's filter sometimes disappears partially, causing some visual glitches around the display.
- The rendering of Front Panel widgets other than the display is truncated: the right and bottom lines (1 pixel thick) are missing. This is caused by the widget rendering clip being too small. As a workaround, you can increase the size of each widget image with empty lines on the right and bottom sides.
- With a zoom other than 1:1, some Front Panel widgets other than the display may not be rendered, or may be truncated.

Flush Visualizer

- A native Concurrent Exception may be thrown during application animations. This is due to the drawing storage not being synchronized with the Flush Visualizer text file dump.
- On non-rectangular displays (see *Display Widget*), the Front Panel widget display's filter sometimes disappears partially, causing some visual glitches around the display in the screenshot.

[14.3.0] - 2025-01-24

FrontPanel

Added

- Add the class ej.fp.widget.display.Rectangle (extracted from ej.tool.frontpanel#widget#4.0.2).
- Add all MicroUI and Drawing algorithms in the Flush Visualizer traces.
- Add a toolbar with various options (zoom, etc.).
- Add a statusbar with various information (current zoom, coordinates, etc.).

- Add the ability to resize and/or scroll the Front Panel.

Changed

- Speed up the drawing of images whose format is identical to the display, without transparency or transformation.
- Display a tree of regions and drawings in the Flush Visualizer traces.
- Use the Front Panel Framework 1.3.0 to implement `LLUIDisplay.convertRegion()`.

Fixed

- Fix the drawing order of the Front Panel widgets: use the order set in the `.fp` file and not systematically draw the display widget over the other widgets.

ImageGenerator

Fixed

- Fix the usage of an incompatible or corrupted cache (drop it).

FontGenerator

Fixed

- Fix the usage of an incompatible or corrupted cache (drop it).

[14.2.0] - 2024-11-18

ImageGenerator

Added

- Add the options `grayscale` and `alpha` for the output formats `A8`, `A4`, `A2`, `A1` and `A8_RLE`.

  - `grayscale` means black pixels are encoded as fully opaque pixels and white pixels as fully transparent pixels.
  - `alpha` only encodes the opacity element like the MicroUI `OutputFormat#A8`.
    - No option defaults to `grayscale` for backward compatibility.
    - An invalid option prevents the encoding of the image.

Changed

- Make the cache faster.

FontGenerator

Added

- Add the support of .fnt files.

Changed

- Make the cache faster.

Fixed

- Fix the computing of fonts heap usage when using cache.

[13.7.3] - 2024-10-18

FontGenerator

Fixed

- Fix the management of the kernel's fonts on the simulator (mode Multi-Sandbox).

[14.1.1] - 2024-10-17

MicroUI

- Implement MicroUI API 3.6.0 (new Font management).

Added

- Add some SystemView logs to trace the new MicroUI 3.6 APIs.

Drawing

Fixed

- Fix the invalid StringIndexOutOfBoundsException when drawing the last character of a string using TransformPainter.drawScaledSubstringBilinear().

FrontPanel

Added

- Add the classes, interfaces, enumeration, registration and utility methods to implement the drawing of strings and the handling of custom fonts.
- Add LLUIDisplay.getByteOrder() to retrieve the embedded VEE Port's memory endianness.
- Add the API LLUIDisplay.isImageClosed() to replace the deprecated API LLUIDisplay.isClosed().

Changed

- Deprecate the API LLUIDisplay.isClosed() : use LLUIDisplay.isImageClosed() instead.

ImageGenerator

Changed

- Speed-up the generation when the cache is used.

FontGenerator

Changed

- Speed-up the generation when the cache is used.

Fixed

- Fix the management of the kernel's fonts on the simulator (mode Multi-Sandbox).

LLAPIs

Added

- Add the LLAPI (implemented by the MicroUI C Module):

– LLUI_PAINTER_IMPL_drawString()
– LLUI_PAINTER_IMPL_stringWidth()
– LLUI_PAINTER_IMPL_initializeRenderableStringSNIContext()
– LLUI_PAINTER_IMPL_drawRenderableString()
– LLDW_PAINTER_IMPL_drawScaledStringBilinear()
– LLDW_PAINTER_IMPL_drawScaledRenderableStringBilinear()
– LLDW_PAINTER_IMPL_drawCharWithRotationBilinear()
– LLDW_PAINTER_IMPL_drawCharWithRotationNearestNeighbor()

- Add the software drawing APIs (implemented by the Graphics Engine):

– UI_DRAWING_SOFT_drawString()
– UI_DRAWING_SOFT_drawRenderableString()
– UI_DRAWING_SOFT_stringWidth()
– UI_DRAWING_SOFT_initializeRenderableStringSNIContext()
– UI_DRAWING_SOFT_drawChar()
– DW_DRAWING_SOFT_drawScaledStringBilinear()
– DW_DRAWING_SOFT_drawScaledRenderableStringBilinear()
– DW_DRAWING_SOFT_drawCharWithRotationBilinear()
– DW_DRAWING_SOFT_drawCharWithRotationNearestNeighbor()

- Add the enumeration MICROUI_FontFormat that lists the MicroUI custom font formats.
- Add the structure MICROUI_Font that represents a MicroUI Font (Font.getSNIContext()).
- Add the structure MICROUI_RenderableString that represents a MicroUI RenderableString (RenderableString.getSNIContext()).

- Add the API LLUI_DISPLAY_isImageClosed() to replace the deprecated API LLUI_DISPLAY_isClosed() .

Changed

- Deprecate the API LLUI_DISPLAY_isClosed() : use LLUI_DISPLAY_isImageClosed() instead.

C Module MicroUI

Added

- Add ui_configuration.h that provides macros to configure some low level APIs.
- Add ui_font_drawing.h that provides functions to handle custom fonts.
- Add the option UI_FEATURE_FONT_CUSTOM_FORMATS to enable the management of custom fonts.
- Add the option UI_FEATURE_ALLOCATOR to replace the Graphics Engine's image heap allocator by a bestfit allocator.
- Add the implementation of the following LLAPI (plus their redirections, default implementations and logs):

– LLUI_PAINTER_IMPL_drawString()
– LLUI_PAINTER_IMPL_stringWidth()
– LLUI_PAINTER_IMPL_initializeRenderableStringSNIContext()
– LLUI_PAINTER_IMPL_drawRenderableString()
– LLDW_PAINTER_IMPL_drawScaledStringBilinear()
– LLDW_PAINTER_IMPL_drawScaledRenderableStringBilinear()
– LLDW_PAINTER_IMPL_drawCharWithRotationBilinear()
– LLDW_PAINTER_IMPL_drawCharWithRotationNearestNeighbor()

- Add the handling of custom fonts (ui_font_drawing.c).

Changed

- Gather all C Module configurations in one file: ui_configuration.h .
- Harmonize the naming convention of all existing options: prefix UI_ .
- Replace the UI event decoder options:

– LLUI_DEBUG_TRACE by UI_DEBUG_PRINT
– MICROUIEVENTDECODER_ENABLED by UI_FEATURE_EVENT_DECODER
– MICROUIEVENTDECODER_EVENTGEN_COMMAND by UI_EVENTDECODER_EVENTGEN_COMMAND
– MICROUIEVENTDECODER_EVENTGEN_BUTTONS by UI_EVENTDECODER_EVENTGEN_BUTTONS
– MICROUIEVENTDECODER_EVENTGEN_TOUCH by UI_EVENTDECODER_EVENTGEN_TOUCH

- Replace the UI display BRS options:

- UI_DISPLAY_BRS by UI_FEATURE_BRS
- UI_DISPLAY_BRS_LEGACY by UI_FEATURE_BRS_LEGACY
- UI_DISPLAY_BRS_SINGLE by UI_FEATURE_BRS_SINGLE
- UI_DISPLAY_BRS_PREDRAW by UI_FEATURE_BRS_PREDRAW
- UI_DISPLAY_BRS_DRAWING_BUFFER_COUNT by UI_FEATURE_BRS_DRAWING_BUFFER_COUNT
- UI_DISPLAY_BRS_FLUSH_SINGLE_RECTANGLE by UI_FEATURE_BRS_FLUSH_SINGLE_RECTANGLE
  - Move the option UI_RECT_COLLECTION_MAX_LENGTH to ui_configuration.h .
  - Replace the option LLUI_GC_SUPPORTED_FORMATS by UI_GC_SUPPORTED_FORMATS .
  - Replace the option LLUI_IMAGE_CUSTOM_FORMATS by UI_FEATURE_IMAGE_CUSTOM_FORMATS .

    Removed

  - Remove microui_event_decoder_conf.h : import ui_configuration.h instead.
  - Remove ui_display_brs_configuration.h : import ui_configuration.h instead.

C Module DMA2D

Changed

- Make this C Module compatible with the MicroUI C Module 14.1.1.

C Module VGLite

Changed

- Make this C Module compatible with the MicroUI C Module 14.1.1.

C Module NemaGFX

Changed

- Make this C Module compatible with the MicroUI C Module 14.1.1.

[14.0.3] - 2024-10-01

MicroUI

Fixed

- Fix the IllegalArgumentException being caught and not logged in Displayable.handleEvent() .
- Fix unexpected exception when killing a feature while MicroUI is not yet started.
- Fix the permissions: getName() and getActions() returned null .

- Fix the log when calling `Image.getImage()`.

FrontPanel

Fixed

- A VEE port's buffered image provider can be ignored (which prevents opening buffered images in a format other than the display format).

LLAPIs

Changed

- Check the `LLUI_DISPLAY_SInitData` values set by the BSP (`LLUI_DISPLAY_IMPL_initialize()`) before initializing the Graphics Engine.

[14.0.2] - 2024-07-26

MicroUI

Fixed

- Fix the rendering of anti-aliased arcs for some compilers (rounding issue).
- Fix a synchronization issue that may cause invalid parameters to be sent in the flush request.

Front Panel

Fixed

- Fix the infinite waiting on a display event when killing a feature.

[14.0.1] - 2024-04-09

MicroUI

- Implement MicroUI API 3.5.0.

Fixed

- Fix the infinite waiting on a display event when killing a feature.
- Fix the dump of the events queue when it is full.

Front Panel

Added

- Add LLUIDisplayImpl.getCurrentBackBuffer() (replace LLUIDisplayImpl.getCurrentDrawingBuffer()).

Changed

- Homogenize the notions of back and front buffers.
- Set LLUIDisplayImpl.getCurrentDrawingBuffer() as deprecated: implement LLUIDisplayImpl.getCurrentBackBuffer() instead.

LLAPIs

Changed

- Homogenize the notions of back and front buffers: LLUI_DISPLAY_setBackBuffer() replaces LLUI_DISPLAY_setDrawingBuffer() .

C Module MicroUI

- New version: C Module MicroUI 4.0.1.

Changed

- Homogenize the notions of back and front buffers.

Fixed

- Disable the clip before calling UI_DISPLAY_BRS_restore() .

C Module DMA2D

- New version: C Module DMA2D 5.0.1.

Changed

- Homogenize the notions of back and front buffers.

C Module VGLite

- New version: C Module VGLite 8.0.1.

Changed

- Homogenize the notions of back and front buffers.

C Module NemaGFX

- New version: C Module NemaGFX 2.0.1.

  Changed

- Homogenize the notions of back and front buffers.

[14.0.0] - 2024-02-14

MicroUI

- Implement MicroUI API 3.5.0.

  Added

- Add `GraphicsContext.notifyDrawingRegion()` that allows the notification of a future altered region.
- Add `Format.getSNIContext()` and `OutputFormat.getSNIContext()` to identify the format in the native world.

  Changed

- Change the semantic of the content of the back buffer after a flush: the *past* is not systematically restored.
- Clarify the message when a generic event generator specified in the VEE Port is not available in the application classpath.

  Fixed

- Fix the drawing of thick faded circle arcs.
- Fix some linker issues on some Architectures:
  - Fix invalid linker issues (when MicroUI is not used or if another allocator is used).
  - Fix custom LCD format on VEE Port with ASLR mode (example: X86 with -pie option).
  - Remove some absolute symbols.
  - Replace sections `.text` by `.rodata`.

Front Panel

  Added

- Add new APIs to manage several display buffer policies and refresh strategies (BRS):
  - Add `LLUIDisplay.getSource()`.
  - Add `LLUIDisplayImpl.newDrawingRegion()`.
  - Add `LLUIDisplayImpl.getCurrentDrawingBuffer()`.
  - Add `MicroUIImage.requestReading()`

  Changed

- Remove `force` parameter in `LLUIDisplay.requestFlush()`
- Remove all parameters in `LLUIDisplayImpl.flush()` and `LLUIDisplayImpl.waitFlush()`

- Extract MicroUIImageFormat and MicroUIImage and MicroUIGraphicsContext from LLUIPainter .

Fixed

- Fix clip and drawn area computing in flush visualizer.

Removed

- Remove MicroUIGraphicsContext.setDrawingLimits() .

LLAPIs

Added

- Add the possibility to log external events in the MicroUI event group.
- Add some functions in LLUI_DISPLAY.h and LLUI_DISPLAY_impl.h to manage the display buffer refresh strategy (BRS):

– LLUI_DISPLAY_getSourceImage() .

– LLUI_DISPLAY_getImageBPP() and LLUI_DISPLAY_getFormatBPP() .

– LLUI_DISPLAY_IMPL_refresh() .

– LLUI_DISPLAY_IMPL_newDrawingRegion() .

– LLUI_DISPLAY_setDrawingBuffer() : it replaces LLUI_DISPLAY_flushDone() .

Changed

- Change the signature of the function LLUI_DISPLAY_requestFlush() : remove the boolean force (not backward compatible).
- Change the signature of the function LLUI_DISPLAY_IMPL_flush() : give a list of rectangles and a flush identifier.

Removed

- Remove the function LLUI_DISPLAY_flushDone() : replaced by LLUI_DISPLAY_setDrawingBuffer() .
- Remove the function LLUI_DISPLAY_setDrawingLimits() .
- Remove the functions LLUI_DISPLAY_logDrawingStart() and LLUI_DISPLAY_logDrawingEnd() : use standard logger instead.

C Module MicroUI

- New version: C Module MicroUI 4.0.0.

Added

- Add the possibility to log external events in the MicroUI event group.
- Add the buffer refresh strategies (BRS) Legacy, Single and Predraw.
- Add some utility functions to manipulate rectangles and collections of rectangles.

C Module DMA2D

- New version: C Module DMA2D 5.0.0.

  Added

- Add the compatibility with UI Pack 14.0.
- Add the function `UI_DRAWING_DMA2D_memcpy_callback()` to be notified about the end of the memory copy.
- Add the support of the display Buffer Refresh Strategies (BRS) `PREDRAW` and `SINGLE`.
- Add a configuration version in `ui_drawing_dma2d_configuration` (`1`).

C Module VGLite

- New version: C Module VGLite 8.0.1.
- Compatible with VGLite library `3.0.15_rev7`.

  Added

- Add the compatibility with UI Pack 14.0.

  Removed

- Remove the compatibility with the VGLite library `3.0.15_rev4`.

C Module NemaGFX

- New version: C Module NemaGFX 2.0.0.

  Added

- Add the compatibility with UI Pack 14.0.

  Fixed

- Fix `nema_draw_line()` `y1` argument.

[13.7.2] - 2023-12-21

MicroUI

Fixed

- Fix the drawing of thick faded circle arcs.

FontGenerator

Fixed

- Fix the management of the feature's fonts on the simulator (mode Multi-Sandbox).

C Module NemaGFX

- New version: C Module NemaGFX 1.2.0.

  Changed

- Disable the rendering of thick faded line with the GPU by default (see option ENABLE_FADED_LINES).
- Increase the version of the configuration file (2).

  Fixed

- Fix the drawing status when a thick line is out-of-clip (results in an infinite loop).

[13.7.0] - 2023-10-23

MicroUI

- Implement MicroUI API 3.4.0.

  Added

- Add the pre-multiplied image formats ARGB8888_PRE , ARGB1555_PRE and ARGB4444_PRE .
- Add the possibility to free third-party resources associated with images.
- Add some traces when debugging the SNI resources.

Front Panel

Added

- Add the pre-multiplied image formats ARGB8888_PRE , ARGB1555_PRE and ARGB4444_PRE .

Image Generator

Changed

- Do not enable the cache when generating external resources.

  Fixed

- Do not use cached images when there is no .images.list file.
- Do not use cached images when a VEE Port property has changed.
- Fix the handling of backslashes in list files.
- Remove debug log in script.

Font Generator

Changed

- Do not enable the cache when generating external resources.

Fixed

- Do not use cached fonts when a VEE Port property has changed.
- Fix the handling of backslashes in list files.

C Module MicroUI

- New version: C Module MicroUI 3.1.1.

Added

- Add the compatibility with UI Pack 13.7.

C Module DMA2D

- New version: C Module DMA2D 4.1.0.

Added

- Add the compatibility with UI Pack 13.7.

C Module VGLite

- New version: C Module VGLite 7.2.0.
- Compatible with VGLite libraries `3.0.15_rev4` and `3.0.15_rev7`.

Added

- Add the pre-mulitplied image formats: `ARGB8888_PRE`, `ARGB4444_PRE` and `ARGB1555_PRE`.
- Add `UI_VGLITE_need_to_premultiply()` to find out whether a color must be pre-multiplied according to the GPU's capabilities.

Fixed

- Fix the use of power quad when not available.

C Module NemaGFX

- New version: C Module NemaGFX 1.1.0.

Added

- Add the compatibility with UI Pack 13.7.

[13.6.2] - 2023-09-20

Image Generator

Fixed

- Fix handling zip/jar file entries in the cache.

Font Generator

Fixed

- Fix handling zip/jar file entries in the cache.

C Module VGLite

- New version: C Module VGLite 7.1.0.
- Compatible with VGLite libraries `3.0.15_rev4` and `3.0.15_rev7`.

  Added

- Add the compatibility with VGLite `3.0.15_rev7` (add a .patch file).

  Fixed

- Fix the use of the define `VG_BLIT_WORKAROUND` (useless).
- Fix the GPU deactivation when a drawing is not performed for any reason.
- VGLite `3.0.15_rev4`: Fix the bounding box of the `vg_lite_blit()` given to the MicroEJ Graphics Engine when the define `VG_BLIT_WORKAROUND` is set (the function `vg_lite_blit()` is not used by default).

[13.6.1] - 2023-07-26

MicroUI

Fixed

- Fix creating a BufferedImage when traces are enabled.

[13.6.0] - 2023-07-17

MicroUI

- Implement MicroUI API 3.3.0.

  Added

- Add a flag stating that an undefined character was drawn.

  Fixed

- Fix the Java compiler version used to build the MicroUI extension class to be compatible with the JDK 11.
- Fix the drawing of faded arcs and ellipses.

Front Panel

Added

- Add the drawing log flag DRAWING_LOG_MISSING_CHARACTER , stating that an undefined character was drawn.

Image Generator

Changed

- Use a cache to avoid generating images for each launch.

Font Generator

Changed

- Use a cache to avoid generating fonts for each launch.

C Module VGLite

- New version: C Module VGLite 7.0.0.
- Compatible with VGLite library 3.0.15_rev4 .
- Several additions, changes and fixes are available. Refer to the C Module VGLite 7.0.0 changelog for more information.
- The C Module has been divided in two parts to extract the NXP i.MX RT500 specific support from the generic C Module for VGLite:
  - NXP i.MX RT500 Display management: C Module RT500 7.0.0
    - Drawing over VGLite: C Module VGLite 7.0.0

C Module NemaGFX

- New C Module: C Module NemaGFX 1.0.0.
- Compatible with UI Pack 13.5.x and 13.6.0.

[13.5.1] - 2023-06-08

MicroUI

Fixed

- Fix the compatibility with MicroEJ Architecture 8 (SOAR error with internal MicroUI system properties file).

Front Panel

Fixed

- Fix consecutive calls to `LLUIDisplay.newMicroUIImage()` throwing an exception.
- Allow overriding the display drawer with a service or in a Front Panel widget.

C Module VGLite

- New version: C Module VGLite 6.0.1.
- Compatible with VGLite library `3.0.15_rev4`.

Fixed

- Fix performing drawings when the clip is disabled.

[13.5.0] - 2023-05-03

MicroUI

- Implement MicroUI API 3.2.0.

Added

- Add multi BufferedImage image formats management.
- Add custom RAM Image image formats management.
- Add drawing logs flags management.

Fixed

- Fix ellipse fading.

Drawing

Fixed

- Fix the position of arc caps.

Front Panel

Added

- Add a service to decode immutable images with a custom format.
- Add a service to create mutable images with a custom format.
- Add a service to draw into mutable images with a format different than the display format.
- Add some methods to manage the MicroUI Drawing Log flags.
- Add some methods to change the MicroUI clip and colors.

Changed

- Merge DWDrawing in UIDrawing.
- Turn UIDrawing as a service to handle drawings for a specific format.
- Change the mechanism to get the software drawer.
- Change the MicroUI image format MICROUI_IMAGE_FORMAT_LCD by MICROUI_IMAGE_FORMAT_DISPLAY.

Removed

- Remove the interfaces UIDrawingDefault and DWDrawingDefault (implement the interface UIDrawing instead).

Image Generator

Added

- Add compatibility with Architecture 8.

LLAPIs

Added

- Add some functions in LLUI_DISPLAY.h to manage the MicroUI Drawing Log flags.
- Add some functions in LLUI_DISPLAY.h to change the MicroUI clip and colors.
- Add the notion of "drawer" to identify the available drawer for a given MicroUI Image format.

Changed

- Change the MicroUI image format MICROUI_IMAGE_FORMAT_LCD by MICROUI_IMAGE_FORMAT_DISPLAY.
- Change the signature of xx_drawing_soft.h : all functions return a drawing status.

Removed

- Remove ui_drawing.h and dw_drawing.h (move them in MicroUI C Module).

C Module MicroUI

- New version: C Module MicroUI 3.0.0.

  Added

- Add support for multiple Graphics Context output formats.
- Add support for multiple Image input formats.
- Add stub implementations for all MicroUI and Drawing libraries algorithms.

C Module DMA2D

- New version: C Module DMA2D 4.0.0.

  Added

- Add the configuration file drawing_dma2d_configuration.h to enable or not the cache management (cache invalidate and clean).
- Add the compatibility with multiple Graphics Context output formats.

  Fixed

- Fix the problems with reading memory back after a DMA2D transfer on cache-enabled CPUs.
- Fix an include directive for case-sensitive filesystems.

C Module VGLite

- New version: C Module VGLite 6.0.0.
- Compatible with VGLite library 3.0.15_rev4 .

  Added

- Add the compatibility with multiple Graphics Context output formats.
- Add (or move) some utility functions in display_vglite .
- Add incident reporting with drawing log flags.

  Fixed

- Set the appropriate format for the destination buffer.
- Fix the drawing of horizontal lines.

  Removed

- Remove the notion of vg_drawer and the define VGLITE_USE_MULTIPLE_DRAWERS (replaced by multiple Graphics Context output formats).

[13.4.1] - 2023-02-06

Drawing

Fixed

- Fix thick lines drawing (when thickness is larger than length).
- Fix circle and ellipse drawing (when the diameter/axis has an even length).

Front Panel

Changed

- Increase the speed of RAW image decoding step.

Image Generator

Fixed

- Fix the VEE Port's memory alignment constraint.

C Module VGLite

- New version: C Module VGLite 5.0.1.
- Compatible with VGLite library $3.0.15_rev4$.
- Several additions, changes and fixes are available. Refer to the C Module VGLite 5.0.1 changelog for more information.

[13.4.0] - 2022-12-13

MicroUI

Fixed

- Fix the unexpected resuming of the pump thread when a new event is added to the queue if it is an other component than the MicroUI queue that has suspended the pump thread.
- Fix the flush bounds of drawCircleArc and drawEllipseArc.

Front Panel

Added

- Add some checks to not perform a drawing when it is unnecessary.

Fixed

- Fix the Front Panel representation of a BufferedImage: it is always opaque.

Image Generator

Added

- Add the image format A8_RLE.

Changed

- Rename RLE1 format in ARGB1565_RLE (keep RLE1 for backward compatibility).

Fixed

- Fix the non-generation of external images for the features.

Font Generator

Fixed

- Fix the external fonts output folder for the features.

C Module MicroUI

- New version: C Module MicroUI 2.0.1.

Changed

- Do not draw thick shapes when thickness and fade are equal to zero.

C Module DMA2D

- New version: C Module DMA2D 3.0.2.

Fixed

- Fix the flush bounds when drawing an image (must be set before calling LLUI_DISPLAY_notifyAsynchronousDrawingEnd()).

C Module VGLite

- New version: C Module VGLite 4.0.0.
- Compatible with VGLite library 3.0.15_rev4 .
- Several additions, changes and fixes are available. Refer to the C Module VGLite 4.0.0 changelog for more information.

[13.3.1] - 2022-09-09

Image Generator

Added

- Add an *Application Option* to quickly test an Image Generator Extension project.

Changed

- Increase logs when application verbosity is enabled.
- Check the stride defined by the Image Generator Extension project (throw an error if the value is incompatible with the memory alignment).

Fixed

- Fix the external resource generation: they were no longer generated (UI pack 13.3.0 regression).
- Fix the duplicate generation (as internal and external resources) of the custom `.list` file images (consider only custom `.list` file images as external resources when the prefix of the list file extension starts with `extern`).
- Fix the internal limit error when converting images with BPP lower than 8 bits (for platforms that define a rule for the image stride through an Image Generator Extension project).

[13.3.0] - 2022-09-02

MicroUI

Fixed

- Fix the Cx (x == 1 | 2 | 4) Graphics Engine's when memory layout is "column".
- Fix the consistency between Image.getImage() and Font.getFont() about starting MicroUI.

Front Panel

Added

- Add custom image formats and a service to prepare for future MicroUI functionality.

Image Generator

Fixed

- Fix the stride stored in the image when the Graphics Engine's memory layout is "column".

LLAPIs

Added

- Add custom image formats to prepare for future MicroUI functionality.
- Add LLAPI to adjust new image characteristics (size and alignment).
- Add API: `UI_DRAWING_copyImage` and `UI_DRAWING_drawRegion`.
- Add the LLUI version (== UI Pack version) in header files.

Changed

- Use type `jbyte` to identify an image format instead of `MICROUI_ImageFormat` (prevent C compiler optimization).

Removed

- Remove the MicroUI's native functions declaration with macros *(not backward compatible)*.

C Module MicroUI

- New version: C Module MicroUI 2.0.0.

Changed

- Improve `drawImage`: identify faster use cases (copy an image and draw a region with overlap).
- Use new UI Pack LLAPI: `UI_DRAWING_copyImage` and `UI_DRAWING_drawRegion`.
- Use new MicroUI's native functions declaration (not backward compatible).

C Module DMA2D for UI Pack 13.2.0 (maintenance)

- New version: C Module DMA2D 2.1.0.

Added

- Add the compatibility with the STM32H7 series.

Changed

- Manage the overlapping (draw an image on the same image).

Fixed

- Fix the limitation of UI Pack 13.x in checking the MicroUI GraphicsContext clip before filling a rectangle.

C Module DMA2D for UI Pack 13.3.0

- New version: C Module DMA2D 3.0.0.

  Added

- Add the implementation of `UI_DRAWING_drawRegion`.

  Removed

- Remove the software implementation of "image overlap" (already available in UI Pack 13.3.0).

C Module VGLite

- New version: C Module VGLite 3.0.0.
- Compatible with VGLite library `3.0.11_rev3`.
- Several additions, changes and fixes are available. Refer to the C Module VGLite 3.0.0 changelog for more information.

[13.2.0] - 2022-05-05

Integration

Changed

- Update to the latest SDK license notice.

MicroUI

- Implement MicroUI API 3.1.1.

  Changed

- Use `.rodata` sections instead of `.text` sections.

  Fixed

- Clean KF stale references when killing a feature without display context switch.
- Make sure to wait the end of an asynchronous drawing before killing a KF feature.
- Redirect the events sent to the pump to the pump's handler instead of to the event generator's handler.
- Fix the drawing of antialiased arc: caps are drawn over the arc itself (rendering issue when the GraphicsContext's background color is set).
- Fix the drawing of antialiased arc: arc is not fully drawn when (int)startAngle == (int)((startAngle + arcAngle) % 360)).
- Fix the input queue size when not already set by the application launcher.
- Fix the use of a negative `scanLength` in GraphicsContext.readPixels() and Image.readPixels().

Drawing

- Compatible with Drawing API 1.0.4.

Front Panel

Added

- Add the property `-Dej.fp.hil=true` in the application launcher to force to run the Front Panel with the Graphics Engine as a standard HIL mock (requires MicroEJ Architecture 7.17.0 or higher).
- Add `LLUIDisplayImpl.decode()` : the Front Panel project is able to read encoded image like the embedded side.
- Include automatically the AWT ImageIO services.
- Add `MicroUIImage.readPixel()` to read an image's pixel color.

Fixed

- Fix the "display context switch" and the loading of feature's font.
- Fix OOM (Managed Heap space) when opening/closing several hundreds of big RAW Images.
- Fix the synchronization with the Graphics Engine when calling GraphicsContext.setColor() or GraphicsContext.enableEllipsis().

Image Generator

Added

- Include automatically the AWT ImageIO services.
- Allow to a custom image converter to generate a file other than a binary resource.
- Allow to a custom image converter to specify the supported `.list` files.

LLAPIs

Added

- Add `LLUI_DISPLAY_readPixel` to read an image's pixel color.

C Module DMA2D

- New version: C Module DMA2D 1.0.8 for UI Pack 13.0.x (maintenance).
- New version: C Module DMA2D 2.0.0 for UI Pack 13.1.0 and UI Pack 13.2.0.

Fixed

- Fix the use of returned code when drawing images with the DMA2D.
- Clean cache before each DMA2D transfer (no-op on STM32 CPU without cache).

C Module VGLite

- New C Module: C Module VGLite 2.0.0.
- Compatible with VGLite library 3.0.11_rev3 .

Added

- Provides the *VGLite C module* 2.0.0 to target the NXP CPU that provides the Vivante VGLite accelerator.

BSP

Fixed

- Fix the IAR Embedded Workbench warnings during debug session.

[13.1.0] - 2021-08-03

MicroUI API

Removed

- Remove MicroUI and Drawing API from UI pack.

MicroUI Implementation

- Implement MicroUI API 3.1.0.

Changed

- Check Immortals Heap minimal size required by MicroUI implementation.
- Change the EventGenerator Pointer event format.
- Do no systematically use the GPU to draw intermediate steps of a shape.

Fixed

- EventGenerator's event has not to be sent to the Display's handler when EventGenerator's handler is null.
- Fill rounded rectangle: fix rendering when corner radius is higher than rectangle height.
- An external image is closed twice when the application only checks if the image is available.
- RLE1 image rendering when platform requires image pixels address alignment.
- Manage the system fonts when the Font Generator is not embedded in the platform.
- Have to wait the end of current drawing before closing an image.

Drawing Implementation

- Compatible with Drawing API 1.0.3.

Front Panel

Added

- Add `MicroUIImage.getImage(int)` : apply a rendering color on Ax images.
- Add `LLUIDisplay.convertRegion()` : convert a region according image format restrictions.
- Add `LLUIDisplayImpl.waitFlush()` : can manage an asynchronous flush.

Changed

- Compatible with new EventGenerator Pointer event format.

Fixed

- Fix OutputFormat A8 when loading an image (path or stream) or converting a RAW image.
- Fix OOM (Managed Heap space) when opening/closing several hundreds of MicroUI Images.
- Simulates the image data alignment.

LLAPIs

Added

- Add `LLUI_DISPLAY_convertDisplayColorToARGBColor()` .
- Add LLAPI to manage the *MicroUI Image heap*.
- Add LLAPI to dump the *MicroUI Events queue*.

Changed

- Change signature of `LLUI_DISPLAY_setDrawingLimits()` : remove `MICROUI_GraphicsContext*` to be able to call this function from GPU callback method.

C Module MicroUI

- New version: C Module MicroUI 1.1.0.

Added

- Add a MicroUI events logger (optional).
- Add a MicroUI images heap allocator (optional).

Fixed

- Fix comments in `LLUI_PAINTER_impl.c` and `LLDW_PAINTER_impl.c` .
- Ignore a drawing when at least one scaling factor is equal to zero.

[13.0.7] - 2021-07-30

MicroUI Implementation

Fixed

- Allow to open a font in format made with UI Pack 12.x (but cannot manage `Dynamic` styles).
- Display.flush() method is called once when MicroUI pump thread has a higher priority than the caller of Display.requestFlush().
- Display.requestFlush() is only executed once from a feature (UI deadlock).

Misc

Fixed

- Fix MMM dependencies: do not fetch the MicroEJ Architecture.

[13.0.6] - 2021-03-29

LLAPIs

Fixed

- Size of the typedef `MICROUI_Image` : do not depend on the size of the enumeration `MICROUI_ImageFormat` (`LLUI_PAINTER_impl.h`).

[13.0.5] - 2021-03-08

MicroUI Implementation

Removed

- Remove ResourceManager dependency.

Fixed

- A feature was not able to call Display.callOnFlushCompleted().
- Stop feature: prevent NullPointerException when a kernel's EventGenerator is removed from event generators pool.
- Filter DeadFeatureException in MicroUI pump.
- Drawing of thick arcs which represent an almost full circle.
- Drawing of thick faded arcs which pass by 0° angle.

Front Panel

Fixed

- Front Panel memory management: reduce simulation time.

[13.0.4] - 2021-01-15

MicroUI API

Changed

- [Changed] Include MicroUI API 3.0.3.
- [Changed] Include MicroUI Drawing API 1.0.2.

MicroUI Implementation

Fixed

- Fix each circle arc cap being drawn on both sides of an angle.
- Fix drawing of rounded caps of circle arcs when fade is 0.
- Cap thickness and fade in thick drawing algorithms.
- Clip is not checked when filling arcs, circles and ellipsis.
- Image path when loading an external image (LLEXT).
- InternalLimitsError when calling MicroUI.callSerially() from a feature.

Drawing Implementation

Fixed

- Draw deformed image is not rendered.

Image Generator

Changed

- Compatible with com.microej.pack.ui#ui-pack(imageGenerator)#13.0.4.

Fixed

- NullPointerException when trying to convert an unknown image.
- Restore external resources option in MicroEJ launcher.

[13.0.3] - 2020-12-03

MicroUI API

Changed

- [Changed] Include MicroUI API 3.0.2.
- [Changed] Include MicroUI Drawing API 1.0.1.

MicroUI Implementation

Fixed

- Reduce Managed Heap usage.
- Fix empty images heap.
- Draw image algorithm does not respect image stride in certain circumstances.
- Fix flush limits of drawThickFadedLine, drawThickEllipse and drawThickFadedEllipse.

C Module MicroUI

- New version: C Module MicroUI 1.0.3.

C Module DMA2D

- New version: C Module DMA2D 1.0.6.

[13.0.2] - 2020-10-02

- Use new naming convention: com.microej.architecture.[toolchain].[architecture]-ui-pack .

Fixed

- [ESP32] - Potential PSRAM access faults by rebuilding using esp-idf v3.3.0 toolchain - simikou2 .

C Module DMA2D

- New version: C Module DMA2D 1.0.5.

Changed

- De-init the DMA2D before re-initializing it, to reset the context at HAL level.
- Manipulate the drawing limits after being sure the DMA2D job is finished.

[13.0.1] - 2020-09-22

MicroUI API

Changed

- Include MicroUI API 3.0.1.

MicroUI Implementation

Fixed

- Throw an exception when there is no display.
- Antialiased circle may be cropped.
- FillRoundedRectangle can give invalid arguments to FillRectangle.
- Flush bounds may be invalid.
- Reduce memory footprint (Managed Heap and Immortals Heap).
- No font is loaded when an external font is not available.
- A8 color is cropped to display limitation too earlier on simulator.

Front Panel

Fixed

- Cannot use an external image decoder on Front Panel.
- Missing an API to check the overlapping between source and destination areas.

Image Generator

Fixed

- Cannot build a platform with Image Generator and without Front Panel.

LLAPIs

Fixed

- Missing a LLAPI to check the overlapping between source and destination areas.

C Module MicroUI

- New version: C Module MicroUI 1.0.2.

  Changed

- Change module organization.

C Module DMA2D

- New version: C Module DMA2D 1.0.3.

  Changed

- Remove/replace notion of `LLDISPLAY`.
- Change module organization.

  Fixed

- Fix file names.

[13.0.0] - 2020-07-30

- Integrate SDK 3.0-B license.

Architecture

Changed

- Compatible with Architecture 7.16.0 or higher (SNI 1.3).

MicroUI API

Changed

- [Changed] Include MicroUI API 3.0.0.
- [Changed] Include MicroUI Drawing API 1.0.0.

MicroUI Implementation

Added

- Manage image data (pixels) address alignment (not more fixed to 32-bits word alignment).

  Changed

- Reduce EDC dependency.
- Merge `DisplayPump` and `InputPump`: only one thread is required by MicroUI.
- Use a `bss` section to load characters from an external font instead of using Managed Heap.

  Removed

- Dynamic fonts (dynamic bold, italic, underline and ratios).

Fixed

- Lock only current thread when waiting end of flush or end of drawing (and not all threads).
- Draw anti-aliased ellipse issue (vertical line is sometimes drawn).
- Screenshot on platform whose *physical* size is higher than *virtual* size.

Known issue

- Render of draw/fill arc/circle/ellipse with an even diameter/edge is one pixel too high (center is 1/2 pixel too high).

Front Panel

Added

- Able to override MicroUI drawings algorithms like embedded platform.

Changed

- Compatible with com.microej.pack.ui#ui-pack(frontpanel)#13.0.0.
- See *Migration notes* that describe the available changes in Front Panel API.

Removed

- ej.tool.frontpanel#widget-microui has been replaced by com.microej.pack.ui#ui-pack(frontpanel) .

Image Generator

Added

- Redirects source image reading to the Image Generator extension project in order to increase the number of supported image formats in input.
- Redirects destination image generation to the Image Generator extension project in order to be able to encode an image in a custom RAW format.
- Generates a linker file in order to always link the resources in same order between two launches.

Changed

- Compatible with com.microej.pack.ui#ui-pack(imageGenerator)#13.0.0.
- See *Migration notes* that describe the available changes in Image Generator API.
- Uses a service loader to loads the Image Generator extension classes.
- Manages image data (pixels) address alignment.

Removed

- Classpath variable IMAGE-GENERATOR-x.x : Image generator extension project has to use ivy dependency com.microej.pack.ui#ui-pack(imageGenerator) instead.

Font Generator

Changed

- Used a dedicated bss section to load characters from an external font instead of using the Managed Heap.

LLAPIs

Added

- Some new functions are mandatory: see header files list, tag *mandatory*.
- Some new functions are optional: see header files list, tag *optional*.
- Some header files list the libraries ej.api.microui and ej.api.drawing natives. Provided by Abstraction Layer implementation module com.microej.clibrary.llimpl#microui.
- Some header files list the drawing algorithms the platform can implement; all algorithms are optional.
- Some header files list the internal Graphics Engine software algorithms the platform can call.

Changed

- All old header files and functions have been renamed or shared.
- See *Migration notes* that describe the available changes in LLAPI.

C Modules

Added

- Provides the C Module MicroUI 1.0.1 that extends the UI Pack 13.0.0.
- Provides the C Module DMA2D 1.0.2 that targets the STM32 CPU that provides the Chrom-ART accelerator.
- See *MicroUI C module*.

[12.1.5] - 2020-10-02

- Use new naming convention: com.microej.architecture.[toolchain].[architecture]-ui-pack .

Fixed

- [ESP32] - Potential PSRAM access faults by rebuilding using esp-idf v3.3.0 toolchain - simikou2 .

[12.1.4] - 2020-03-10

MicroUI Implementation

Fixed

- Obsolete references on Managed Heap are used (since MicroEJ UI Pack 12.0.0).

[12.1.3] - 2020-02-24

MicroUI Implementation

Fixed

- Caps are not used when drawing an anti-aliased line.

[12.1.2] - 2019-12-09

MicroUI Implementation

Fixed

- Fix Graphics Engine empty clip (empty clip had got a size of 1 pixel).
- Clip not respected when clip is set “just after or before” graphics context drawable area: first (or last) line (or column) of graphics context was rendered.

[12.1.1] - 2019-10-29

MicroUI Implementation

Fixed

- Fix Graphics Engine clip (cannot be outside graphics context).

[(maint) 8.0.0] - 2019-10-18

- Based on UI Pack 7.4.7.

Architecture

Changed

- Compatible with Architecture 7.0.0 or higher (Use SNI callback feature).

MicroUI Implementation

Fixed

- Pending flush cannot be added after an `OutOfEventException`.

[12.1.0] - 2019-10-16

MicroUI API

Changed

- Include MicroUI API 2.4.0.

MicroUI Implementation

Changed

- Prepare inlining of get X/Y/W/H methods.
- Reduce number of strings embedded by MicroUI library.

Fixed

- Pending flush cannot be added after an `OutOfEventException`.
- Display.isColor() returns an invalid value.
- Draw/fill circle/ellipse arc is not drawn when angle is negative.

[12.0.2] - 2019-09-23

MicroUI Implementation

Changed

- Change `CM4hardfp_IAR83` compiler flags.
- Remove RAW images from cache as soon as possible to reduce Managed Heap usage.
- Do not cache RAW images with their paths to reduce Managed Heap usage.

Fixed

- Remove useless exception in SystemInputPump.

[12.0.1] - 2019-07-25

MicroUI Implementation

Fixed

- Physical size is not taken in consideration.

Front Panel

Fixed

- Increase native implementation execution time.

[12.0.0] - 2019-06-24

Architecture

Changed

- Compatible with Architecture 7.11.0 or higher (Move Front Panel in Architecture).

MicroUI Implementation

Added

- Trace MicroUI events and log them on SystemView.

Changed

- Manage the Graphics Context clip on native side.
- Use Managed Heap to store images metadata instead of using icetea heap (remove option “max offscreen”).
- Optimize retrieval of all fonts.
- Ensure user buffer size is larger than LCD size.
- Use Managed Heap to store flying images metadata instead of using icetea heap (remove option “max flying images”).
- Use Managed Heap to store fill polygon algorithm’s objects instead of using icetea heap (remove option “max edges”).
- `SecurityManager` enabled as a boolean constant option (footprint removal by default).
- Remove `FlyingImage` feature using BON constants (option to enable it).

Fixed

- Wrong rendering of a fill polygon on emb.
- Wrong rendering of image overlaping on C1/2/4 platforms.
- Wrong rendering of a LUT image with more than 127 colors on emb.
- Wrong rendering of an antialiased arc with 360 angle.

- Debug option com.is2t.microui.log=true fails when there is a flying image.
- Gray scale between gray and white makes magenta.
- Minimal size of some buffers set by user is never checked.
- The format of a RAW image using “display” format is wrong.
- Dynamic image width for platform C1/2/4 may be wrong.
- Wrong pixel address when reading from a C2/4 display.
- getDisplayColor() can return a color with transparency (spec is `0x00RRGGBB`).
- A fully opaque image is tagged as transparent (ARGB8888 platform).

Front Panel

Added

- Simulate flush time (add JRE property `-Dfrontpanel.flush.time=8`).

Fixed

- A pixel read on an image is always truncated.

Front Panel Plugin

Removed

- Front Panel version 5: Move Front Panel from MicroEJ UI Pack to Architecture *(not backward compatible)*; Architecture contains now Front Panel version 6.

[11.2.0] - 2019-02-01

MicroUI Implementation

Added

- Manage extended UTF16 characters (> 0xffff).

Fixed

- IOException thrown instead of an OutOfMemory when using external resource loader.

Tools

Removed

- Remove Font Designer from pack (useless).

[11.1.2] - 2018-08-10

MicroUI Implementation

Fixed

- Fix drawing bug in thick circle arcs.

[11.1.1] - 2018-08-02

- Internal release.

[11.1.0] - 2018-07-27

- Merge 10.0.2 and 11.0.1.

MicroUI API

Changed

- Include MicroUI API 2.3.0.

MicroUI Implementation

Added

- LLDisplay : prepare round LCD.

Fixed

- Fillrect throws a hardfault on 8bpp platform.
- Rendering of a LUT image is wrong when using software algorithm.

[11.0.1] - 2018-06-05

- Based on UI Pack 11.0.0.

MicroUI Implementation

Fixed

- Image rendering may be invalid on custom display.
- Render a dynamic image on custom display is too slow.
- LRGB888 image format is always fully opaque.
- Number of colors returned when it is a custom display may be wrong.

[10.0.2] - 2018-02-15

- Based on UI Pack 10.0.1.

MicroUI Implementation

Fixed

- Number of colors returned when it is a custom display may be wrong.
- LRGB888 image format is always fully opaque.
- Render a dynamic image on custom display is too slow.
- Image rendering may be invalid on custom display.

[11.0.0] - 2018-02-02

- Based on UI Pack 10.0.1.

Architecture

Changed

- Compatible with Architecture 7.0.0 or higher (Use SNI callback feature).

MicroUI Implementation

Changed

- SNI Callback feature in the Core Engine to remove the SNI retry pattern *(not backward compatible)*.

[10.0.1] - 2018-01-03

MicroUI Implementation

Fixed

- Hard fault when using custom display stack.

[10.0.0] - 2017-12-22

Architecture

Changed

- Compatible with Architecture 6.13.0 or higher (LLEXT link error with Architecture 6.13+ and UI Pack 9+).

MicroUI Implementation

Changed

- Improve `TOP-LEFT` anchor checks.

Fixed

- Subsequent renderings may not be correctly flushed.
- Rendering of display on display was not optimized.

Front Panel

Changed

- Check the allocated memory when creating a dynamic image *(not backward compatible)*.

Misc

Added

- Option in platform builder to images heap size.

[9.4.1] - 2017-11-24

Image Generator

Fixed

- Missing some files in Image Generator module.

[9.4.0] - 2017-11-23

- Deprecated: use UI Pack 9.4.1 instead.

MicroUI Implementation

Added

- LUT image management.

Changed

- Optimize character encoding removing first vertical line when possible.

Fixed

- Memory leak when an `OutOfEventException` is thrown.
- A null Java object is not checked when using a font.

[9.3.1] - 2017-09-28

MicroUI Implementation

Fixed

- Returned X coordinates when drawing a string was considered as an error code.
- Exception when loading a font from an application.
- LLEXT link error with Architecture 6.13+ and UI 9+.

[9.3.0] - 2017-08-24

MicroUI Implementation

Fixed

- Ellipsis must not drawn when text anchor is a “manual” TOP-RIGHT .

Front Panel

Fixed

- Do not create an AWT window for each image.
- Error when trying to play with an unknown led.

[9.2.1] - 2017-08-14

Front Panel

Added

- Provide function to send a Long Button event.
- “flush” debug option.

Fixed

- Mock startup is too long.

[9.2.0] - 2017-07-21

- Merge UI Packs 9.1.2 and 9.0.2.

Architecture

Changed

- Compatible with Architecture 6.12.0 or higher (SOAR can exclude some resources).

MicroUI API

Changed

- Include MicroUI API 2.2.0.

MicroUI Implementation

Added

- Provide function to send a Long Button event (emb only).

Changed

- Use font format v5.
- A signature on RAW files.
- Allow to open a raw image with `Image.createImage(stream)`.
- Improve `Image.createImage(stream)` when stream is a memory input stream.

Fixed

- Draw region of the display on the display does not support overlap.
- Unspecified exception while loading an image with an empty name.
- Display.flush(): ymax can be higher than display.height.

Image Generator

Fixed

- Generic displays must be able to generate standard images.

Misc

Changed

- SOAR can exclude some resources (update llext output folder).

Fixed

- RI build: reduce Front Panel dependency.

[9.0.2] - 2017-04-21

- Based on UI Pack 9.0.1.

MicroUI Implementation

Fixed

- Rendering of a RAW image on grayscale display is wrong.

Image Generator

Fixed

- An Ax image may be fully opaque.

[9.1.2] - 2017-03-16

- Based on UI Pack 9.1.1.

MicroUI API

Changed

- Include MicroUI API 2.1.3.

MicroUI Implementation

Added

- Renderable strings.

Changed

- Draw string: improve time to perform it.
- Optimize antialiased circle arc drawing when fade=0.

Fixed

- ImageScale bugs.
- Draw string: some errors are not thrown.
- Font.getWidth() and getHeight() don't use ratio factor.
- Draw antialiased circle arc render issue.
- Draw antialiased circle arc render bug with 45° angles.
- MicroUI lib expects the dynamic image decoder default format.
- Wrong error code is returned when converting an image.

Image Generator

Fixed

- Use the application classpath.
- An Ax image may be fully opaque.

[9.0.1] - 2017-03-13

- Based on UI Pack 9.0.0.

MicroUI Implementation

Fixed

- Hardfault when filling a rectangle on an odd image.
- Pixel rendering on non-standard LCD is wrong.
- RZ hardware accelerator: RAW images have to respect an aligned size.
- Use the classpath when invoking the fonts and images generators.

Front Panel

Fixed

- Wrong rendering of A8 images.

Front Panel Plugin

Fixed

- Manage display mask on preview.
- Respect initial background color set by user on preview.
- Preview does not respect the real size of display.

[9.1.1] - 2017-02-14

- Based on UI Pack 9.1.0.

Misc

Fixed

- RI build: Several custom event generators in same `microui.xml` file are not embedded.

[9.1.0] - 2017-02-13

- Based on UI Pack 9.0.0.

Architecture

Changed

- Compatible with Architecture 6.8.0 or higher (Internal scripts).

MicroUI API

Changed

- Include MicroUI API 2.1.2.

MicroUI Implementation

Added

- G2D hardware accelerator.
- Hardware accelerator: add flip feature.

Fixed

- Hardfault when filling a rectangle on an odd image.
- Pixel rendering on non-standard LCD is wrong.
- RZ hardware accelerator: RAW images have to respect an aligned size.
- Use the classpath when invoking the fonts and images generators.
- Exception when flipping an image out of display bounds.
- Flipped image is translated when clip is modified.

Front Panel

Fixed

- Wrong rendering of A8 images.

Front Panel Plugin

Fixed

- Manage display mask on preview.
- Respect initial background color set by user on preview.
- Preview does not respect the real size of display.

[9.0.0] - 2017-02-02

MicroUI API

Changed

- Include MicroUI API 2.0.6.

MicroUI Implementation

Changed

- Update MicroUI to use watchdogs in KF implementation.

Fixed

- Display linker file is required even if there is no display on platform.
- MicroUI on KF: NPE when changing app quickly (in several threads).
- MicroUI on KF: NPE when stopping a Feature and there's no eventHandler in a generator.
- MicroUI on KF: Remaining K->F link when there is no default event handler registered by the Kernel.

MWT

Removed

- Remove MWT from MicroEJ UI Pack *(not backward compatible)*.

Front Panel

Added

- Optional mask on display.

Changed

- Display Device UID if available in the window title.

Tools

Changed

- Front Panel plugin: Update icons.
- Font Designer plugin: Update icons.
- Font Designer and Generator: use Unicode 9.0.0 specification.

Misc

Fixed

- Remove obsolete documentations from Front Panel And Font Designer plugins.

[8.1.0] - 2016-12-24

MicroUI Implementation

Changed

- Improve image drawing timings.
- Runtime decoders can force the output RAW image's fully opacity.

MWT

Fixed

- With two panels, the paint is done but the screen is not refreshed.
- Widget show notify method is called before the panel is set.
- Widget still linked to panel when `lostFocus()` is called.

Front Panel

Added

- Can add an additional screen on simulator.

[8.0.0] - 2016-11-17

Architecture

Changed

- Compatible with Architecture 6.4.0 or higher (Manage external memories like byte addressable memories).

MicroUI Implementation

Added

- RZ UI acceleration.
- External image decoders.
- Manage external memories like internal memories.
- Custom display stacks (hardware acceleration).

Changed

- Merge stacks `DIRECT/COPY/SWITCH` *(not backward compatible)*.

Fixed

- add KF rule: a thread cannot enter in a feature code while it owns a kernel monitor.
- automatic flush is not waiting the end of previous flush.
- Invalid image rotation rendering.
- Do not embed Images & Fonts.list of kernel API classpath in app mode.
- Invalid icetea heap allocation.
- microui image: invalid “defaultformat” and “format” fields values.

MWT

Fixed

- possible to create an inconsistent hierarchy.

Front Panel

Added

- Can decode additional image formats.

Fixed

- Cannot set initial value of StateEventGenerator.

[7.4.7] - 2016-06-14

MicroUI Implementation

Fixed

- Do not create all fonts derivations of built-in styles.
- A bold font is not flagged as bold font.
- Wrong A4 image rendering.

Front Panel

Fixed

- Cannot convert an image.

[7.4.2] - 2016-05-25

MicroUI Implementation

Fixed

- invalid image drawing for *column* display.

[7.4.1] - 2016-05-10

MicroUI Implementation

Fixed

- Restore stack 1, 2 and 4 BPP.

[7.4.0] - 2016-04-29

MicroUI Implementation

Fixed

- image A1's width is sometimes invalid.

Front Panel

Added

- Restore stack 1, 2 and 4 BPP.

[7.3.0] - 2016-04-25

MicroUI Implementation

Added

- Stack 8BPP with LUT support.

[7.2.1] - 2016-04-18

Misc

Fixed

- Remove `java` keyword in workbench extension.

[7.2.0] - 2016-04-05

Tools

Added

- Preprocess `*.xxx.list` files.

[7.1.0] - 2016-03-02

MicroUI Implementation

Added

- Manage several images RAW formats.

[7.0.0] - 2016-01-20

Misc

Changed

- Remove `@jpf.property.header@` prefix to Application options *(not backward compatible)*.

[6.0.1] - 2015-12-17

MicroUI Implementation

Fixed

- A negative clip throws an exception on simulator.

[6.0.0] - 2015-11-12

- Compatible with Architecture 6.1.0 or higher.

MicroUI Implementation

Changed

- LLDisplay for UIv2 *(not backward compatible)*.

Migration Guide

From 14.4.2 to 14.5.2

BSP

The C Module's options configuration file `ui_configuration.h` must not be modified anymore. Before installing the new C Module:

- Create (if it doesn't exist already) the global VEE Port configuration file.
- Move your own configuration from `ui_configuration.h` to `veeport_configuration.h` .
- Install the C Module MicroUI 14.5.2.
- Call the SNI API `SNI_registerResource()` for each image created thanks to the low-level API:
  - `LLUI_DISPLAY_allocateImageBuffer()`
  - `LLUI_DISPLAY_decodeImage()`

The following options (see `ui_configuration.h`) have now a value (not just commented/uncommented); check (and adjust) whether they are used outside the UI C Modules:

- `UI_FEATURE_ALLOCATOR`
- `UI_FEATURE_EVENT_DECODER`
- `UI_FEATURE_IMAGE_CUSTOM_FORMATS`
- `UI_FEATURE_FONT_CUSTOM_FORMATS`
- `UI_FEATURE_BRS_FLUSH_SINGLE_RECTANGLE`

BSP with DMA2D

- **Prerequisite:** follow the migration steps of *BSP*.

The C Module's options configuration file `ui_dma2d_configuration.h` must not be modified anymore. Before installing the new C Module:

- Move your own configuration from `ui_dma2d_configuration.h` to `veeport_configuration.h` .
- Install the C Module MicroUI over DMA2D to version 8.0.0.

BSP with VGLite

- **Prerequisite:** follow the migration steps of *BSP*.

  The C Module's options configuration file `ui_vglite_configuration.h` must not be modified anymore. Before installing the new C Module:

- Move your own configuration from `ui_vglite_configuration.h` to `veeport_configuration.h`.
- Install the C Module MicroUI over VGLite to version 11.0.0.

  The following options (see `ui_vglite_configuration.h`) have now a value (not just commented/uncommented); check (and adjust) whether they are used outside the UI C Modules:

- `VGLITE_USE_GPU_FOR_SIMPLE_DRAWINGS`
- `VGLITE_USE_GPU_FOR_RGB565_IMAGES`
- `VGLITE_USE_GPU_FOR_TRANSPARENT_IMAGES`
- `VGLITE_OPTION_TOGGLE_GPU`

BSP with NemaGFX

- **Prerequisite:** follow the migration steps of *BSP*.

  The C Module's options configuration file `ui_nema_configuration.h` must not be modified anymore. Before installing the new C Module:

- Move your own configuration from `ui_nema_configuration.h` to `veeport_configuration.h`.
- Install the C Module MicroUI over NemaGFX to version 5.0.0.

  The following options (see `ui_nema_configuration.h`) have now a value (not just commented/uncommented); check (and adjust) whether they are used outside the UI C Modules:

- `ENABLE_SIMPLE_LINES`
- `ENABLE_FADED_LINES`
- `ENABLE_IMAGE_ROTATION`

Application

- Update the SystemView description file for MicroUI traces as described *here*.

From 14.3.3 to 14.4.2

Front Panel

- Fetch Front Panel Widgets 5.3.1 (see *Module Dependencies*).
- To enable all Flush Visualizer functionalities, set the properties `core.trace.enabled` and `core.trace.autostart` to `true` (see *Installation*). The property `ej.fp.display.flushVisualizer` is not used anymore.

BSP

- Install the C Module MicroUI 14.4.2.

BSP with DMA2D

- **Prerequisite:** follow the migration steps of *BSP*.
- Install the C Module MicroUI over DMA2D to version 7.0.1.

BSP with VGLite

- **Prerequisite:** follow the migration steps of *BSP*.
- Install the C Module MicroUI over VGLite to version 10.0.1.

BSP with NemaGFX

- **Prerequisite:** follow the migration steps of *BSP*.
- Install the C Module MicroUI over NemaGFX to version 4.0.1.

Application

- Update the SystemView description file for MicroUI traces as described *here*.

From 14.3.3 to 14.4.1

UI Pack 14.4.1 is deprecated, use the UI Pack 14.4.2 instead.

From 14.2.0 to 14.3.3

Front Panel

- Fetch Front Panel Widgets 5.2.0 (see *Module Dependencies*).
- A widget that overlaps the display widget must be transparent or defined **before** the display widget in the `.fp` file.

From 14.1.1 to 14.2.0

BSP

- Install the C Module MicroUI 14.2.0.

From 14.0.3 to 14.1.1

BSP without GPU

- Install the C Module MicroUI 14.1.1.
- Configure ui/inc/ui_configuration.h , based on your previous settings in ui/inc/microui_event_decoder_conf.h and ui/inc/ui_display_brs_configuration.h .
  - Notice that the name (prefix) of the options changed, see *Changelog*.
  - If the BSP uses the MicroUI C Module's image heap allocator instead of using the Graphics Engine's image heap allocator, set the preprocessor value UI_FEATURE_ALLOCATOR=UI_FEATURE_ALLOCATOR_BESTFIT .
  - Comment the line that starts with #error (...) .
- Delete configuration files ui/inc/microui_event_decoder_conf.h and ui/inc/ui_display_brs_configuration.h .
- Add the source file in ui/src/ui_font_drawing.c to the project.
- Update c and h files and BSP configuration (if any) to use the new preprocessor values:
  - UI_DEBUG_PRINT replaces LLUI_DEBUG_TRACE
  - UI_FEATURE_EVENT_DECODER replaces MICROUIEVENTDECODER_ENABLED
  - UI_EVENTDECODER_EVENTGEN_COMMAND replaces MICROUIEVENTDECODER_EVENTGEN_COMMAND
  - UI_EVENTDECODER_EVENTGEN_BUTTONS replaces MICROUIEVENTDECODER_EVENTGEN_BUTTONS
  - UI_EVENTDECODER_EVENTGEN_TOUCH replaces MICROUIEVENTDECODER_EVENTGEN_TOUCH
  - UI_FEATURE_BRS replaces UI_DISPLAY_BRS
  - UI_FEATURE_BRS_LEGACY replaces UI_DISPLAY_BRS_LEGACY
  - UI_FEATURE_BRS_SINGLE replaces UI_DISPLAY_BRS_SINGLE
  - UI_FEATURE_BRS_PREDRAW replaces UI_DISPLAY_BRS_PREDRAW
  - UI_FEATURE_BRS_DRAWING_BUFFER_COUNT replaces UI_DISPLAY_BRS_DRAWING_BUFFER_COUNT
  - UI_FEATURE_BRS_FLUSH_SINGLE_RECTANGLE replaces UI_DISPLAY_BRS_FLUSH_SINGLE_RECTANGLE
  - UI_GC_SUPPORTED_FORMATS replaces LLUI_GC_SUPPORTED_FORMATS
  - UI_FEATURE_IMAGE_CUSTOM_FORMATS replaces LLUI_IMAGE_CUSTOM_FORMATS

BSP with DMA2D

- **Prerequisite:** follow the migration steps of *BSP without GPU*.
- Install the C Module MicroUI over DMA2D to version 7.0.0.

BSP with VGLite

- **Prerequisite:** follow the migration steps of *BSP without GPU*.
- Install the C Module MicroUI over VGLite to version 10.0.0.

BSP with NemaGFX

- **Prerequisite:** follow the migration steps of *BSP without GPU*.
- Install the C Module MicroUI over NemaGFX to version 4.0.0.

From 14.0.1 to 14.0.3

BSP without GPU

- Install the C Module MicroUI 4.1.0.

BSP with DMA2D

- Install the C Module MicroUI over DMA2D to version 6.0.0.
  - Notice that the name of the module changed from `display-dma2d` to `microui-dma2d`, and that the module was moved to the *Developer Repository*.
- In the directory `ui`, delete the following files:
  - `inc/ui_drawing_dma2d.h`,
  - `src/ui_drawing_dma2d.c`.
- Configure `ui_dma2d/inc/ui_dma2d_configuration.h`, based on your previous settings in `ui/inc/ui_drawing_dma2d_configuration.h`.
- Delete `ui/inc/ui_dma2d_configuration.h`.
- Add the source files in `ui_dma2d/src` to the project.
- Add the path `ui_dma2d/inc` to the include path.

BSP with VGLite

- Install the C Module MicroUI over VGLite to version 9.0.0.
- In the directory ui , delete the following files:

– inc/color.h ,
– inc/ui_drawing_vglite.h ,
– inc/ui_drawing_vglite_path.h ,
– inc/ui_drawing_vglite_process.h ,
– inc/ui_vglite.h ,
– src/ui_drawing_vglite.c ,
– src/ui_drawing_vglite_path.c ,
– src/ui_drawing_vglite_process.c .

- In the directory util , delete the following files:

– inc/mej_math.h ,
– src/mej_math.c ,

- In LLUI_DISPLAY_IMPL_initialize :
  – Call UI_VGLITE_initialize to initialize the VGLite controller and the VGLite library.
  – Replace the call to UI_VGLITE_init with UI_VGLITE_start .
- Configure ui_vglite/inc/ui_vglite_configuration.h , based on your previous settings in ui/inc/ui_vglite_configuration.h .
- Delete ui/inc/ui_vglite_configuration.h .
- Add the source files in ui_vglite/src to the project.
- Add the path ui_vglite/inc to the include path.

BSP with NemaGFX

- Install the C Module MicroUI over NemaGFX to version 3.0.0.
- Delete ui/src/ui_drawing_nema.c .
- Replace the following calls with the new function names:

– UI_DRAWING_NEMA_initialize becomes UI_NEMA_initialize ,
– UI_DRAWING_NEMA_post_operation becomes UI_NEMA_post_operation ,
– UI_DRAWING_NEMA_configure_memcpy becomes UI_NEMA_configure_memcpy ,
– UI_DRAWING_NEMA_start_memcpy becomes UI_NEMA_start_memcpy .

- Configure ui_nemagfx/inc/ui_nema_configuration.h , based on your previous settings in ui/inc/ui_drawing_nema_configuration.h .
- Add the source files in ui_nemagfx/src to the project.
- Add the path ui_nemagfx/inc to the include path.
- If you were using the task mode, switch to interrupt mode.

- Remove your implementation of UI_DRAWING_NEMA_IMPL_operation_submitted .
- Remove the task that called UI_DRAWING_NEMA_post_operation .
- Look for any use of the macro NEMA_INTERRUPT_MODE , which denotes a differentiation between the task mode and the interrupt mode. This macro is no longer used. Adjust your code so that only the code targeting the interrupt mode remains.
- Follow the steps detailed in the section *Implementation*.

From 13.7.x to 14.0.1

Front Panel

- Fetch Front Panel Widgets 4.0.1 (it fetches by transitivity the UI Pack 14.0.1):

```
<dependency org="ej.tool.frontpanel" name="widget" rev="4.0.1"/>
```

- Re-organize imports of all Java classes (classes MicroUIImageFormat , MicroUIImage and MicroUIGraphicsContext have been extracted from LLUIPainter).
- Remove all calls to MicroUIGraphicsContext.setDrawingLimits . This method has been removed as it is no longer needed.
- The doubleBufferFeature attribute has been removed from the Display widget. The bufferPolicyClass replaces it (see *Buffer Refresh Strategy on the Simulator*).

```
<ej.fp.widget.Display x="0" y="0" width="480
↪" height="272" bufferPolicyClass="ej.fp.widget.display.buffer.SwapDoubleBufferPolicy"/>
```

- The FlushVisualizerDisplay widget has been merged with the Display widget. To use this functionality, use the Display widget instead of the FlushVisualizerDisplay widget in the Front Panel .fp file and set the option ej.fp.display.flushVisualizer=true in the options of the application launcher.

BSP Without GPU

- Install the C Module MicroUI 4.0.1.
- Delete the VEE Port include folder (often /platform/inc).
- In the C project configuration, include the new C files ui_display_brs.c , ui_display_brs_legacy.c , ui_display_brs_predraw.c , ui_display_brs_single.c and ui_rect_util.c .
- Read the documentation about the display *Buffer Refresh Strategy*; then configure the C module by setting the right configuration in ui_display_brs_configuration.h .
- Comment the line #error "This header must [...]" .
- The next actions depend on the available numbers of buffers allocated in the MCU memories and if the front buffer is mapped on an MCU's buffer (if not, that means the LCD device owns a buffer). The following table redirects the next steps according to the display connection with the MCU:

Table 85: Transmit and/or Swap actions

Buffers	Mapped	Next Actions
2 (1+1)	no	*[Display "Transmit"]*
2	yes	*[Display "Swap double buffer"]*
3	yes	*[Display "Swap triple buffer"]*
3 (2+1)	no	*[Display "Transmit and Swap"]*

- *[Display "Transmit"]*
  - Set the value of the define `UI_DISPLAY_BRS` : `UI_DISPLAY_BRS_SINGLE` .
  - Set the value of the define `UI_DISPLAY_BRS_DRAWING_BUFFER_COUNT` : `1` .
  - Uncomment the define `UI_DISPLAY_BRS_FLUSH_SINGLE_RECTANGLE` .
  - Change the signature and the implementation of the function flush: `void LLUI_DISPLAY_IMPL_flush(MICROUI_GraphicsContext* gc, uint8_t flush_identifier, const ui_rect_t regions[], size_t length)`
    - Store (in a static field) the rectangle to flush (the array contains only one rectangle).
    - Store (in a static field) the flush identifier.
    - Unlock (immediately or wait for the LCD tearing signal interrupt) the *flush task* (hardware or software) that will transmit the back buffer data to the front buffer.
    - Remove the returned value (the back buffer address).
  - At the end of the flush (in an interrupt or at the end of the software *flush task*), replace the call to `LLUI_DISPLAY_flushDone()` with `LLUI_DISPLAY_setBackBuffer()` : it will unlock the Graphics Engine. Give the back buffer address (same address as at start-up) and the flush identifier.
- *[Display "Swap double buffer"]*
  - Set the value of the define `UI_DISPLAY_BRS` : `UI_DISPLAY_BRS_PREDRAW` .
  - Set the value of the define `UI_DISPLAY_BRS_DRAWING_BUFFER_COUNT` : `2` .
  - Change the signature and the implementation of the function flush: `void LLUI_DISPLAY_IMPL_flush(MICROUI_GraphicsContext* gc, uint8_t flush_identifier, const ui_rect_t regions[], size_t length)`
    - Store (in a static field) the back buffer address (*LLUI_DISPLAY_getBufferAddress(&gc->image)*).
    - Store (in a static field) the flush identifier.
    - Unlock (immediately or wait for the LCD tearing signal interrupt) the *swap task* (hardware or software) that will swap the back buffer and the front buffer.
    - Remove the static fields `ymin` and `ymax` (now useless).
    - Remove the returned value (the back buffer address).
  - Case of *hardware swap* (LCD *swap* interrupt): change the implementation of the LCD *swap* interrupt:
    - Remove all the code concerning the post-flush restoration (remove the *flush task* or the use of a DMA). In both cases, the call to `LLUI_DISPLAY_flushDone()` is removed.
    - Unlock the Graphics Engine by calling `LLUI_DISPLAY_setBackBuffer()` , giving the new back buffer address and the flush identifier.

- Case of *software swap* (dedicated *swap task*): change the task actions:
  - Swap back and front buffers.
  - Wait for the end of the buffers swap: ensure the LCD driver does not use the old front buffer anymore.
  - Remove all the code concerning the post-flush restoration (the call to `memcpy` or the use of a DMA). In both cases, the call to `LLUI_DISPLAY_flushDone()` is removed.
  - Unlock the Graphics Engine by calling `LLUI_DISPLAY_setBackBuffer()`, giving the new back buffer address and the flush identifier.
- *[Display "Swap triple buffer"]*
  - Set the value of the define `UI_DISPLAY_BRS`: `UI_DISPLAY_BRS_PREDRAW`.
  - Set the value of the define `UI_DISPLAY_BRS_DRAWING_BUFFER_COUNT`: `3`.
  - Change the signature and the implementation of the function flush: `void LLUI_DISPLAY_IMPL_flush(MICROUI_GraphicsContext* gc, uint8_t flush_identifier, const ui_rect_t regions[], size_t length)`
  - Store (in a static field) the back buffer address (*LLUI_DISPLAY_getBufferAddress(&gc->image)*).
  - Store (in a static field) the flush identifier.
  - Unlock (immediately or wait for the LCD tearing signal interrupt) the *swap task* that will swap the buffers.
  - Remove the static fields `ymin` and `ymax` (now useless).
  - Remove the returned value (the back buffer address).
  - In the *swap task*: change the task actions:
  - Swap buffers.
  - Remove all the code concerning the post-flush restoration (the call to `memcpy` or the use of a DMA). In both cases, the call to `LLUI_DISPLAY_flushDone()` is removed.
  - Unlock the Graphics Engine by calling `LLUI_DISPLAY_setBackBuffer()`, giving the new back buffer address and the flush identifier (the Graphics Engine can be unlocked immediately because a buffer is freed for sure).
  - Wait for the end of the buffers swap: ensure the LCD driver does not use the old front buffer anymore.
- *[Display "Transmit and Swap"]*
  - Set the value of the define `UI_DISPLAY_BRS`: `UI_DISPLAY_BRS_PREDRAW`.
  - Set the value of the define `UI_DISPLAY_BRS_DRAWING_BUFFER_COUNT`: `2`.
  - Uncomment the define `UI_DISPLAY_BRS_FLUSH_SINGLE_RECTANGLE`.
  - Change the signature and the implementation of the function flush: `void LLUI_DISPLAY_IMPL_flush(MICROUI_GraphicsContext* gc, uint8_t flush_identifier, const ui_rect_t regions[], size_t length)`
  - Store (in a static field) the rectangle to flush (the array contains only one rectangle).
  - Store (in a static field) the back buffer address (*LLUI_DISPLAY_getBufferAddress(&gc->image)*).
  - Store (in a static field) the flush identifier.

* Unlock (immediately or wait for the LCD tearing signal interrupt) the *transmit & swap task* that will transmit the current back buffer data to the front buffer, and that will swap the back buffers.
* Remove the returned value (the back buffer address).

– In the *transmit & swap task*: change the "transmit & swap" actions:

* Start the transmission of the current back buffer (called *buffer A*) data to the front buffer.
* Swap back *buffer A* and back *buffer B*.
* Wait for the end of the back buffers swap: ensure the LCD driver is now using the *buffer A* as the *transmission* buffer.
* Remove all the code concerning to the post-flush restoration (the call to `memcpy` or the use of a DMA). In both cases, the call to `LLUI_DISPLAY_flushDone()` is removed.
* Unlock the Graphics Engine by calling `LLUI_DISPLAY_setBackBuffer()`, giving the back *buffer B* address and the flush identifier.
* Wait for the end of the *transmission*: ensure the LCD driver has finished to flush the data.
* (optional) Unlock again the Graphics Engine by calling `LLUI_DISPLAY_setBackBuffer()`, giving the *buffer A* address and the flush identifier:

· The call to `LLUI_DISPLAY_setBackBuffer()` returns `false`: that means at least one drawing has been performed in the *buffer B*; there is nothing else to do.

· The call to `LLUI_DISPLAY_setBackBuffer()` returns `true`: that means no drawing has started yet in the *buffer B*. In that case, the Graphics Engine will reuse the *buffer A* as a back buffer, and the *restoration of the past* becomes useless. The back buffers swap is so canceled; update the LCD driver status in consequence.

BSP with DMA2D

- **Prerequisite:** follow the migration steps of *BSP Without GPU*.
- Install the C Module DMA2D 5.0.1.
- Check the content of the configuration file `ui_drawing_dma2d_configuration.h` (a versioning has been added).
- Comment the line `#error [...]"`.
- According to the display *Buffer Refresh Strategy*, unlock the MicroUI Graphics Engine in the LCD interrupt or the DMA2D memcpy callback (see *C Module: MicroUI Over DMA2D*).

BSP with VGLite

- **Prerequisite:** follow the migration steps of *BSP Without GPU*.
- Install the C Module VGLite 8.0.1.
- Migrate VGLite library to the version **3.0.15_rev7**.
- Modify the VGLite library **3.0.15_rev7** by applying the patch `3.0.15_rev7.patch` (see README.md near the patch file for more information).
- In the file `vglite_window.c`, add the function `VGLITE_CancelSwapBuffers()` and its prototype in `vglite_window.h`:

```
void VGLITE_CancelSwapBuffers(void) {
    fb_idx = fb_idx == 0 ? (APP_BUFFER_COUNT - 1) : (fb_idx ) - 1;
}
```

BSP with NemaGFX

- **Prerequisite:** follow the migration steps of *BSP Without GPU*.
- Install the C Module NemaGFX 2.0.1.
- Check the content of the configuration file `ui_drawing_nema_configuration.h` (new version `2`).

From 13.6.x to 13.7.2

Front Panel

- (optional) Fetch explicitly the UI Pack 13.7.2 to use the new API of the UI Pack:

```
<dependency org="com.microej.pack.ui" name="ui-pack" rev="13.7.2">
    <artifact name="frontpanel" type="jar"/>
</dependency>
```

BSP without GPU

- Install the C Module MicroUI 3.1.1.
- Optionally, implement `UI_DRAWING_freeImageResources(MICROUI_Image* image)` (single-output buffered image format) or `UI_DRAWING_freeImageResources_X(MICROUI_Image* image)` (mulitple-output buffered image formats, where `X` is the image format identifier) to free the resources associated with a buffered image when it is closed.

BSP with DMA2D

- **Prerequisite:** follow the migration steps of *BSP without GPU*.
- Install the C Module DMA2D 4.1.0.

BSP with VGLite

- **Prerequisite:** follow the migration steps of *BSP without GPU*.
- Install the C Module VGLite 7.2.0.

BSP with NemaGFX

- **Prerequisite:** follow the migration steps of *BSP without GPU*.
- Install the C Module NemaGFX 1.2.0.
- Review all options of `ui_drawing_nema_configuration.h` (version `2`).

From 13.5.x to 13.6.2

Front Panel

- (optional) Fetch Front Panel Widgets 3.0.0 to use the new features of the Front Panel Widget library:

```
<dependency org="ej.tool.frontpanel" name="widget" rev="3.0.0"/>
```

- (optional) Fetch explicitly the UI Pack 13.6.2 to use the new API of the UI Pack:

```
<dependency org="com.microej.pack.ui" name="ui-pack" rev="13.6.2">
    <artifact name="frontpanel" type="jar"/>
</dependency>
```

BSP with VGLite

These steps are for a VEE Port that manages its own implementation of `LLUI_DISPLAY_impl.h` (that did not use the old implementation which was available in this C Module):

- Install the C Module VGLite 7.1.0.
- (optional) Fetch C Module RT500 7.0.0
- Delete the following files from the file-system and from the C project configuration:

  - `inc/display_utils.h`
  - `inc/display_vglite.h`
  - `inc/drawing_vglite.h`
  - `inc/vglite_path.h`
  - `src/display_stub.c`
  - `src/display_utils.c`
  - `src/display_vglite.c`
  - `src/drawing_vglite.c`
  - `src/vglite_path.c`

- Add the new files to the C project configuration:

  - `src/ui_drawing_vglite_path.c`
  - `src/ui_drawing_vglite_process.c`
  - `src/ui_vglite.c`

- Review all imports of the removed header files.
- In the implementation of LLUI_DISPLAY_impl.h , call UI_VGLITE_init() during the initialization step.
- In the GPU interrupt rountine, call UI_VGLITE_IRQHandler() .
- Review all options of ui_vglite_configuration.h .
- Implement UI_VGLITE_IMPL_notify_gpu_xxx() instead of DISPLAY_IMPL_notify_gpu_xxx() .
- Implement UI_VGLITE_IMPL_error() instead of DISPLAY_IMPL_error() .
- Change all calls to DISPLAY_VGLITE_xxx() functions to UI_VGLITE_xxx() functions.
- Change all calls to DRAWING_VGLITE_xxx() functions to UI_DRAWING_VGLITE_PROCESS_xxx() functions.
- Change all calls to VGLITE_PATH_xxx() functions to UI_DRAWING_VGLITE_PATH_xxx() functions.
- Change all calls to DISPLAY_UTILS_xxx() functions to UI_VGLITE_xxx() functions.

BSP With MCU i.MX RT595

These steps are for a VEE Port that uses the implementation of LLUI_DISPLAY_impl.h which was available in the C Module *VGLite*:

- Install the C Module VGLite 7.1.0.
- Install the C Module RT500 7.0.0
- Follow the steps of *BSP with VGLite* (described above) except the calls to UI_VGLITE_init() and UI_VGLITE_IRQHandler() .
- Implement DISPLAY_DMA_IMPL_notify_dma_xxx() instead of DISPLAY_IMPL_notify_dma_xxx() .

BSP with NemaGFX

- Install the C Module NemaGFX 1.0.0.
- Add all the C files available in src folder.
- Configure the C project to include the inc folder.
- Read the comments in ui_drawing_nema_configuration.h and configure the C module.

From 13.4.x to 13.5.1

Front Panel

- (optional) Fetch explicitly the UI Pack 13.5.1 to use the new API of the UI Pack:

```
<dependency org="com.microej.pack.ui" name="ui-pack" rev="13.5.1">
    <artifact name="frontpanel" type="jar"/>
</dependency>
```

- Replace any calls to LLUIPainter.setDrawer() and LLDWPainter.setDrawer() to LLUIDisplay.Instance.registerUIDrawer() .
- Replace any calls to LLUIPainter.getDrawer() and LLDWPainter.getDrawer() to LLUIDisplay.Instance.getUIDrawer() .
- Replace any calls to LLUIDisplay.getDWDrawerSoftware() to LLUIDisplay.Instance.getUIDrawerSoftware() .
- Implementation of the interface UIDrawingDefault : implement the interface UIDrawing instead.
- Implementation of the interfaces DWDrawing and DWDrawingDefault : implement the interface UIDrawing instead.
- Implementation of the service BufferedImageProvider : implement handledFormat() and remove the parameter format from newBufferedImage() .
- Replace any occurrences of MICROUI_IMAGE_FORMAT_LCD by MICROUI_IMAGE_FORMAT_DISPLAY .

BSP without GPU

- Install the C Module MicroUI 3.0.0.
- Delete the VEE Port include folder (often /platform/inc).
- In the C project configuration, include the new C files ui_drawing.c , ui_image_drawing.c and ui_drawing_stub.c .

BSP with DMA2D

- **Prerequisite:** follow the migration steps of *BSP without GPU*.
- Install the C Module DMA2D 4.0.0.
- Read the comments about the cache in drawing_dma2d_configuration.h .
- Uncomment the expected define DRAWING_DMA2D_CACHE_MANAGEMENT (enable or disable the cache management).
- Delete the C files drawing_dma2d.h and drawing_dma2d.c and remove them from the C project configuration.
- In the C project configuration, include the new C file ui_drawing_dma2d.c .
- Replace the import drawing_dma2d.h by ui_drawing_dma2d.h .
- Replace the calls to functions DRAWING_DMA2D_xxx() by UI_DRAWING_DMA2D_xxx() .

BSP with VGLite

Note: The C Module is designed to target the NXP i.MX RT500; however it can be locally customized for other boards (see *[Custom project]*)

- **Prerequisite:** follow the migration steps of *BSP without GPU*.
- Install the C Module VGLite 6.0.1.
- Delete the C files `vg_drawer.h` and `vg_drawer.c` and remove them from the C project configuration.
- Verify the options in `display_configuration.h` .
- In the C project configuration, include the new C file `ui_drawing_vglite.c` .

From 13.3.x to 13.4.1

BSP without GPU

- Install the C Module MicroUI 2.0.1.

BSP with DMA2D

- **Prerequisite:** follow the migration steps of *BSP without GPU*.
- Install the C Module DMA2D 3.0.2.

BSP with VGLite

Note: The C Module is designed to target the NXP i.MX RT500; however it can be locally customized for other boards (see *[Custom project]*)

- **Prerequisite:** follow the migration steps of *BSP without GPU*.
- Install the C Module VGLite 5.0.1.
- Migrate VGLite library to the version **3.0.15_rev4**.
- Modify the VGLite library **3.0.15_rev4** by applying the patch `3.0.15_rev4.patch` (see README.md near patch file for more information).

From 13.2.x to 13.3.1

Front Panel

- (optional) Fetch explicitly the UI Pack 13.3.1 to use the new API of the UI Pack:

```
<dependency org="com.microej.pack.ui" name="ui-pack" rev="13.3.1">
    <artifact name="frontpanel" type="jar"/>
</dependency>
```

BSP without GPU

- Install the C Module MicroUI 2.0.0.

BSP with DMA2D

- **Prerequisite:** follow the migration steps of *BSP without GPU*.
- Install the C Module DMA2D 3.0.0.

BSP with VGLite

Note: The C Module is designed to target the NXP i.MX RT500; however it can be locally customized for other boards (see *[Custom project]*).

- **Prerequisite:** follow the migration steps of *BSP without GPU*.
- Install the C Module VGLite 3.0.0.
- Read the comments in `display_configuration.h` and configures the C module.
- Add all C files available in `src` folder.
- Configure the C project to include the `inc` folder.
- Modify the VGLite library **3.0.11_rev3** by applying the patch `3.0.11_rev3.patch` (see README.md near patch file for more information).
- *[Custom project]*
  - Modify or remove the C files `display_dma.c`, `display_frambuffer.c`, `LLUI_DISPLAY_impl.c`, `display_dma.c`, `vglite_support.c` and `vglite_window.c`.

From 13.1.x to 13.2.0

Front Panel

- (optional) Fetch explicitly the UI Pack 13.2.0 to use the new API of the UI Pack:

```
<dependency org="com.microej.pack.ui" name="ui-pack" rev="13.2.0">
    <artifact name="frontpanel" type="jar"/>
</dependency>
```

From 13.0.x to 13.1.0

Front Panel

- (optional) Fetch Front Panel Widgets 2.1.0 to use the new features of the Front Panel Widget library (it fetches by transitivity the UI Pack 13.1.0):

```
<dependency org="ej.tool.frontpanel" name="widget" rev="2.1.0"/>
```

- (optional) Or fetch explicitly the UI Pack 13.1.0 to use the new API of the UI Pack:

```
<dependency org="com.microej.pack.ui" name="ui-pack" rev="13.1.0">
    <artifact name="frontpanel" type="jar"/>
</dependency>
```

BSP without GPU

- Install the C Module MicroUI 1.1.1.
- Add a cast when using `MICROUI_Image*` object: `(MICROUI_ImageFormat)image->format` .
- Remove parameter `MICROUI_GraphicsContext*` when calling `LLUI_DISPLAY_setDrawingLimits()` .
- Ensure to call `LLUI_DISPLAY_setDrawingLimits()` before calling `LLUI_DISPLAY_setDrawingStatus()` or `LLUI_DISPLAY_notifyAsynchronousDrawingEnd()` .
- (optional) Add an implementation of `LLUI_DISPLAY_IMPL_image_heap_xxx` to control the *images heap allocation*; by default the internal Graphics Engine's allocator is used. Another implementation is also available in the *MicroUI C module*.
- (optional) Add the UI event logger available in the *MicroUI C module*.

BSP with DMA2D

- **Prerequisite:** follow the migration steps of *BSP without GPU*.
- Install the C Module DMA2D 2.1.0.

From 12.x to 13.0.7

VEE Port Configuration Project

- Update Architecture version: 7.16.0 or higher.
- Add the following module in the *module description file*:

```
<dependency org="com.microej.clibrary.llimpl" name="microui" rev="1.0.3"/>
```

- If not already set, set the `ea:property bsp.project.microej.dir` in the module ivy file to configure the BSP output folder where is extracted the module.

Hint: Update the BSP with the new MicroUI C module files as described *here*.

Hardware Accelerator

- Open `-configuration` project > `display` > `display.properties`
- Remove optional property `hardwareAccelerator`. If old value was `dma2d`, add the following module in the *module description file*:

```
<dependency org="com.microej.clibrary.llimpl" name="display-dma2d" rev="1.0.8"/>
```

- For the hardware accelerator DMA2D, please consult STM32F7Discovery board updates. Add the file `lldisplay_dma2d.c`, the global defines `DRAWING_DMA2D_BPP=16` (or another value) and `STM32F4XX` or `STM32F7XX`
- For the others hardware accelerators, please contact MicroEJ support.

Hint: Update the BSP with the new C modules files as described *here*.

Front Panel

This chapter resumes the changes to perform. The available changes in Front Panel API are described in *next chapter*.

- If not already done, follow the Front Panel version 6 migration procedure detailed in chapter *From 11.x to 12.1.5*.
- Fetch the new Front Panel Widget library:

```
<dependency org="ej.tool.frontpanel" name="widget" rev="2.0.0"/>
```

- ej.fp.event.MicroUIButtons has been renamed in ej.microui.event.EventButton , and all others ej.fp.event.MicroUIxxx in ej.microui.event.Eventxxx
- Display abstract class AbstractDisplayExtension (class to extend widget Display when targeting a custom display) has been converted on the interface DisplayExtension . Some methods names have changed and now take in parameter the display widget.

Front Panel API

- ej.drawing.DWDrawing
  - [Added] Equivalent of dw__drawing.h and dw__drawing__soft.h** : allows to implement some drawing algorithms and/or to use the ones provided by the Graphics Engine. The drawing methods are related to the library ej.api.drawing .
  - [Added] Interface DWDrawingDefault : default implementation of DWDrawing which calls the Graphics Engine algorithms.
- ej.drawing.LLDWPainter
  - [Added] Equivalent of module com.microej.clibrary.llimpl#microui (LLDW__PAINTER__impl.c): implements all ej.api.drawing natives and redirect them to the interface DWDrawing .
  - [Added] setDrawer(DWDrawing) : allows to configure the implementation of DWDrawing the LLDWPainter has to use. When no drawer is configured, LLDWPainter redirects all drawings to the internal Graphics Engine software algorithms.
- ej.fp.event.MicroUIButtons
  - [Removed] Replaced by EventButton .
- ej.fp.event.MicroUICommand
  - [Removed] Replaced by EventCommand .
- ej.fp.event.MicroUIEventGenerator
  - [Removed] Replaced by LLUIInput .
- ej.fp.event.MicroUIGeneric
  - [Removed] Replaced by EventGeneric .
- ej.fp.event.MicroUIPointer
  - [Removed] Replaced by EventPointer .
- ej.fp.event.MicroUIStates
  - [Removed] Replaced by EventState .
- ej.fp.event.MicroUITouch
  - [Removed] Replaced by EventTouch .
- ej.fp.widget.MicroUIDisplay
  - [Removed] Replaced by LLUIDisplayImpl . Abstract widget display class has been replaced by an interface that a widget (which should simulate a display) has to implement to be compatible with the Graphics Engine.
  - [Removed] AbstractDisplayExtension , all available implementations and setExtensionClass(String) : the standard display formats (RGB565, etc.) are internally managed by the Graphics Engine. For generic formats, some APIs are available in LLUIDisplayImpl .

  - [Removed] finalizeConfiguration() , getDisplayHeight() , getDisplayWidth() , getDrawingBuffer() , setDisplayWidth(int) , setDisplayHeight(int) , start() : LLUIDisplayImpl is not an abstract widget anymore, these notions are widget dependent.
  - [Removed] flush() .
  - [Removed] getNbBitsPerPixel() .
  - [Removed] switchBacklight(boolean) .
- ej.fp.widget.MicroUILED
  - [Removed] Replaced by LLUILedImpl . Abstract widget LED class has been replaced by an interface that a widget (which should simulate a LED) has to implement to be compatible with the Graphics Engine.
  - [Removed] finalizeConfiguration() : LLUILedImpl is not an abstract widget anymore, this notion is widget dependent.
  - [Removed] getID() : MicroUI uses the widget (which implements the interface LLUILedImpl)'s label to retrieve the LED. The LED labels must be integers from 0 to n-1 .
- ej.microui.display.LLUIDisplay
  - [Added] Equivalent of LLUI_DISPLAY.h : several functions to interact with the Graphics Engine.
  - [Added] blend(int,int,int) : blends two ARGB colors and opacity level.
  - [Added] convertARGBColorToColorToDraw(int) : crops given color to display capacities.
  - [Added] getDisplayPixelDepth() : replaces MicroUIDisplay.getNbBitsPerPixel() .
  - [Added] getDWDrawerSoftware() : gives the unique instance of Graphics Engine's internal software drawer (instance of DWDrawing).
  - [Added] getUIDrawerSoftware() : gives the unique instance of Graphics Engine's internal software drawer (instance of UIDrawing).
  - [Added] mapMicroUIGraphicsContext(byte[]) and newMicroUIGraphicsContext(byte[]) : maps the graphics context byte array (GraphicsContext.getSNIContext()) on an object which represents the graphics context in front panel.
  - [Added] mapMicroUIImage(byte[]) and newMicroUIImage(byte[]) : maps the image byte array (Image.getSNIContext()) on an object which represents the image in front panel.
  - [Added] requestFlush(boolean) : requests a call to LLUIDisplayImpl.flush() .
  - [Added] requestRender(void) : requests a call to Displayable.render() .
- ej.microui.display.LLUIDisplayImpl
  - [Added] Replaces MicroUIDisplay , equivalent of LLUI_DISPLAY_impl.h .
  - [Added] initialize() : asks to initialize the widget and to return a front panel image where the Graphics Engine will perform the MicroUI drawings.
  - [Changed] flush(MicroUIGraphicsContext, Image, int, int, int, int) : asks to flush the graphics context drawn by MicroUI in image returned by initialize() .
- ej.microui.display.LLUIPainter
  - [Added] Equivalent of module com.microej.clibrary.llimpl#microui (LLUI_PAINTER_impl.c): implements all ej.api.microui natives and redirect them to the interface UIDrawing .

  - [Added] MicroUIGraphicsContext : representation of a MicroUI GraphicsContext in front panel. This interface (implemented by the Graphics Engine) provides several function to get information on graphics context, clip, etc.
  - [Added] MicroUIGraphicsContext#requestDrawing() : allows to take the hand on the back buffer.
  - [Added] MicroUIImage : representation of a MicroUI Image in front panel. This interface (implemented by the Graphics Engine) provides several function to get information on image.
  - [Added] setDrawer(UIDrawing) : allows to configure the implementation of UIDrawing the LLUIPainter has to use. When no drawer is configured, LLUIPainter redirects all drawings to the internal Graphics Engine software algorithms.
- ej.microui.display.UIDrawing
  - [Added] Equivalent of ui_drawing.h and ui_drawing_soft.h** : allows to implement some drawing algorithms and/or to use the ones provided by the Graphics Engine. The drawing methods are related to the library ej.api.microui .
  - [Added] Interface UIDrawingDefault : default implementation of UIDrawing which calls the Graphics Engine algorithms.
- ej.microui.event.EventButton
  - [Added] Replaces MicroUIButton .
- ej.microui.event.EventCommand
  - [Added] Replaces MicroUICommand .
- ej.microui.event.EventGeneric
  - [Added] Replaces MicroUIGeneric .
- ej.microui.event.EventPointer
  - [Added] Replaces MicroUIPointer .
- ej.microui.event.EventQueue
  - [Added] Dedicated events queue used by MicroUI.
- ej.microui.event.EventState
  - [Added] Replaces MicroUIState .
- ej.microui.event.EventTouch
  - [Added] Replaces MicroUITouch .
- ej.microui.event.LLUIInput
  - [Added] Replaces MicroUIEventGenerator .
- ej.microui.led.LLUILedImpl
  - [Added] Replaces MicroUILED .

Image Generator

This chapter resumes the changes to perform. The available changes in Image Generator API are described in *next chapter*.

This chapter only concerns VEE Port with a custom display. In this case a dedicated image generator extension project is available. This project must be updated.

- Reorganize project to use source folders `src/main/java` and `src/main/resources`
- Add new `module.ivy` file:

```
<ivy-module version="2.0" xmlns:ea="http://www.easyant.org" xmlns:m="http://
↪www.easyant.org/ivy/maven" xmlns:ej="https://developer.microej.com" ej:version="2.0.0">

  <info organisation=
↪"com.is2t.microui" module="imageGenerator-xxx" status="integration" revision="1.0.0">
    <ea:build␣
↪organisation="com.is2t.easyant.buildtypes" module="build-std-javalib" revision="2.+"/>
  </info>

  <configurations defaultconfmapping="default->default;provided->provided">
    <conf name=
↪"default" visibility="public" description="Runtime dependencies to other artifacts"/>
    <conf name="provided" visibility=
↪"public" description="Compile-time dependencies to APIs provided by the VEE Port"/>
    <conf name="documentation
↪" visibility="public" description="Documentation related to the artifact (javadoc, PDF)"/>
    <conf name="source" visibility="public" description="Source code"/>
    <conf name=
↪"dist" visibility="public" description="Contains extra files like README.md, licenses"/>
    <conf name="test" visibility=
↪"private" description="Dependencies for test execution. It is not required for normal use␣
↪of the application, and is only available for the test compilation and execution phases."/>
  </configurations>

  <publications/>

  <dependencies>
    <dependency org="com.microej.pack.ui" name="ui-pack" rev="[UI Pack version]">
      <artifact name="imageGenerator" type="jar"/>
    </dependency>
  </dependencies>
</ivy-module>
```

The artifact name prefix must be `imageGenerator-` .

- Update project classpath: remove classpath variable `IMAGE-GENERATOR-x.x` and add ivy file dependency
- Instead of implementing `GenericDisplayExtension` , the extension class must extend `BufferedImageLoader` class; check class methods to override.
- Add the file `src/main/resources/META-INF/services/com.microej.tool.ui.generator.MicroUIRawImageGeneratorExtension` ; this file has to specify the class which extends the `BufferedImageLoader` class, for instance:

```
com.microej.generator.MyImageGeneratoExtension
```

- Build the easyant project

- Copy the jar in the VEE Port `-configuration` project > `dropins` folder
- Rebuild the VEE Port after any changes

Image Generator API

- com.is2t.microej.microui.image.CustomDisplayExtension
  - [Removed] Replaced by ImageConverter and MicroUIRawImageGeneratorExtension .
- com.is2t.microej.microui.image.DisplayExtension
  - [Removed]
- com.is2t.microej.microui.image.GenericDisplayExtension
  - [Removed] Replaced by ImageConverter and MicroUIRawImageGeneratorExtension .
- com.microej.tool.ui.generator.BufferedImageLoader
  - [Added] Pixelated image loader (PNG, JPEG etc.).
- com.microej.tool.ui.generator.Image
  - [Added] Representation of an image listed in a images.list file.
- com.microej.tool.ui.generator.ImageConverter
  - [Added] Generic converter to convert an image in an output stream.
- com.microej.tool.ui.generator.MicroUIRawImageGeneratorExtension
  - [Added] Graphics Engine RAW image converter: used when the image (listed in images.list) targets a RAW format known by the Graphics Engine.

Font

- Open optional font(s) in `-configuration` project > `microui/**/*.ejf`
- Remove all Dynamic styles (select None or Built-in for bold, italic and underline); the number of generated fonts must be 1 (the feature to render Dynamic styles at runtime have been removed)
- Save the file(s)

BSP

This chapter resumes the changes to perform. The available changes in LLAPI are described in *next chapter*.

- Delete all VEE Port header files (folder should be set in `-configuration` project > `bsp` > `bsp.properties` > property output.dir)
- If not possible to delete this folder, delete all UI headers files:

  - intern/LLDISPLAY*
  - intern/LLINPUT*
  - intern/LLLEDS*

- LLDISPLAY*
- LLINPUT*
- LLLEDS*
    - Replace all #include "LLDISPLAY.h" , #include "LLDISPLAY_EXTRA.h" and #include "LLDISPLAY_UTILS.h" by #include "LLUI_DISPLAY.h"
    - Replace all #include "LLDISPLAY_impl.h", #include "LLDISPLAY_EXTRA_drawing.h" and #include "LLDISPLAY_EXTRA_impl.h" by #include "LLUI_DISPLAY_impl.h"
    - Replace all LLDISPLAY_EXTRA_IMAGE_xxx by MICROUI_IMAGE_FORMAT_xxx
    - All LLDISPLAY_IMPL_xxx functions have been renamed in LLUI_DISPLAY_IMPL_xxx
    - LLUI_DISPLAY_IMPL_initialize has now the paremeter LLUI_DISPLAY_SInitData* init_data ; fill it as explained in C doc.
    - Implement new functions void LLUI_DISPLAY_IMPL_binarySemaphoreTake(void* sem) and void LLUI_DISPLAY_IMPL_binarySemaphoreGive(void* sem, bool under_isr)
    - Signature of LLUI_DISPLAY_IMPL_flush has changed
    - All LLDISPLAY_EXTRA_IMPL_xxx functions have been renamed in LLUI_DISPLAY_IMPL_xxx
    - Fix some functions signatures (LLUI_DISPLAY_IMPL_hasBacklight() , etc)
    - Remove the functions LLDISPLAY_IMPL_getGraphicsBufferAddress , LLDISPLAY_IMPL_getHeight , LLDISPLAY_IMPL_getWidth , LLDISPLAY_IMPL_synchronize , LLDISPLAY_EXTRA_IMPL_waitPreviousDrawing , LLDISPLAY_EXTRA_IMPL_error
    - Add the end of asynchronous flush copy, call LLUI_DISPLAY_flushDone
    - Add the files LLUI_PAINTER_impl.c and LLDW_PAINTER_impl.c in your C configuration project
    - Replace the prefix LLINPUT in all header files, functions and defines by the new prefix LLUI_INPUT
    - Replace the prefix LLLEDS in all header files, functions and defines by the new prefix LLUI_LED
    - Replace the prefix LLDISPLAY in all header files, functions and defines by the new prefix LLUI_DISPLAY

LLAPI

- dw_drawing_soft.h
  - [Added] List of internal Graphics Engine software algorithms to perform some drawings (related to library ej.api.drawing).
- dw_drawing.h
  - [Added] List of ej.api.drawing library's drawing functions to optionally implement in VEE Port.
- LLDISPLAY.h and intern/LLDISPLAY.h
  - [Removed]
- LLDISPLAY_DECODER.h and intern/LLDISPLAY_DECODER.h
  - [Removed]
- LLDISPLAY_EXTRA.h and intern/LLDISPLAY_EXTRA.h merged in LLUI_PAINTER_impl.h and LLDW_PAINTER_impl.h
  - [Changed] LLDISPLAY_SImage : replaced by MICROUI_Image .
  - [Removed] LLDISPLAY_SRectangle , LLDISPLAY_SDecoderImageData , LLDISPLAY_SDrawImage , LLDISPLAY_SFlipImage , LLDISPLAY_SScaleImage and LLDISPLAY_SRotateImage
- LLDISPLAY_EXTRA_drawing.h
  - [Removed]
- LLDISPLAY_EXTRA_impl.h and intern/LLDISPLAY_EXTRA_impl.h merged in LLUI_DISPLAY_impl.h , ui_drawing.h and dw_drawing.h
  - [Changed] LLDISPLAY_EXTRA_IMPL_setContrast(int32_t) : replaced by LLUI_DISPLAY_IMPL_setContrast(uint32_t) (_optional_).
  - [Changed] LLDISPLAY_EXTRA_IMPL_getContrast(void) : replaced by LLUI_DISPLAY_IMPL_getContrast(void) (_optional_).
  - [Changed] LLDISPLAY_EXTRA_IMPL_hasBackLight(void) : replaced by LLUI_DISPLAY_IMPL_hasBacklight(void) (_optional_).
  - [Changed] LLDISPLAY_EXTRA_IMPL_setBacklight(int32_t) : replaced by LLUI_DISPLAY_IMPL_setBacklight(uint32_t) (_optional_).
  - [Changed] LLDISPLAY_EXTRA_IMPL_getBacklight(void) : replaced by LLUI_DISPLAY_IMPL_getBacklight(void) (_optional_).
  - [Changed] LLDISPLAY_EXTRA_IMPL_isColor(void) : replaced by LLUI_DISPLAY_IMPL_isColor(void) (_optional_).
  - [Changed] LLDISPLAY_EXTRA_IMPL_getNumberOfColors(void) : replaced by LLUI_DISPLAY_IMPL_getNumberOfColors(void) (_optional_).
  - [Changed] LLDISPLAY_EXTRA_IMPL_isDoubleBuffered(void) : replaced by LLUI_DISPLAY_IMPL_isDoubleBuffered(void) (_optional_).
  - [Changed] LLDISPLAY_EXTRA_IMPL_getBacklight(void) : replaced by LLUI_DISPLAY_IMPL_getBacklight(void) (_optional_).
  - [Changed] LLDISPLAY_EXTRA_IMPL_fillRect(void*,int32_t,void*,int32_t) : replaced by UI_DRAWING_fillRectangle(MICROUI_GraphicsContext*,jint,jint,jint,jint) (_optional_).

  - [Changed] LLDISPLAY_EXTRA_IMPL_drawImage(void*,int32_t,void*,int32_t,void*) : replaced by UI_DRAWING_drawImage(MICROUI_GraphicsContext*,MICROUI_Image*,jint,jint,jint,jint,jint,jint,jint) (_optional_).
  - [Changed] LLDISPLAY_EXTRA_IMPL_flipImage(void*,int32_t,void*,int32_t,void*) : replaced by DW_DRAWING_drawFlippedImage(MICROUI_GraphicsContext*,MICROUI_Image*,jint,jint,jint,jint,jint,jint,DRAWING_Flip,jint) (_optional_).
  - [Changed] LLDISPLAY_EXTRA_IMPL_scaleImage(void*,int32_t,void*,int32_t,void*) : replaced by DW_DRAWING_drawScaledImageNearestNeighbor(MICROUI_GraphicsContext*,MICROUI_Image*,jint,jint,jfloat,jfloat,jint) and DW_DRAWING_drawScaledImageBilinear(MICROUI_Graphi MICROUI_Image*,jint,jint,jfloat,jfloat,jint) (_optional_).
  - [Changed] LLDISPLAY_EXTRA_IMPL_rotateImage(void*,int32_t,void*,int32_t,void*) : replaced by DW_DRAWING_drawRotatedImageNearestNeighbor(MICROUI_GraphicsContext*,MICROUI_Image*,jint,jint,jint,jint,jfloat,jint) and DW_DRAWING_drawRotatedImageBilinear(MICROUI_Gr MICROUI_Image*,jint,jint,jint,jint,jfloat,jint) (_optional_).
  - [Changed] LLDISPLAY_EXTRA_IMPL_convertARGBColorToDisplayColor(int32_t) and LLDISPLAY_EXTRA_IMPL_convertDisplayColorToARGBColor(int32_t) : replaced respectively by LLUI_DISPLAY_IMPL_convertARGBColorToDisplayColor(uint32_t) and LLUI_DISPLAY_IMPL_convertDisplayColorToARGBColor(uint32_t) (_optional_).
  - [Changed] LLDISPLAY_EXTRA_IMPL_prepareBlendingOfIndexedColors(void*,void*) : replaced by LLUI_DISPLAY_IMPL_prepareBlendingOfIndexedColors(uint32_t*,uint32_t*) (_optional_).
  - [Changed] LLDISPLAY_EXTRA_IMPL_decodeImage(int32_t,int32_t,int32_t,void*) : replaced by LLUI_DISPLAY_IMPL_decodeImage(uint8_t*,uint32_t,MICROUI_ImageFormat,MICROUI_Image*,bool*) (_optional_).
  - [Removed] LLDISPLAY_EXTRA_IMPL_getGraphicsBufferMemoryWidth(void) and LLDISPLAY_EXTRA_IMPL_getGraphicsBufferMemoryHeight(void) : replaced by elements in structure LLUI_DISPLAY_SInitData (_optional_).
  - [Removed] LLDISPLAY_EXTRA_IMPL_backlightOn(void) and LLDISPLAY_EXTRA_IMPL_backlightOff(void) .
  - [Removed] LLDISPLAY_EXTRA_IMPL_enterDrawingMode(void) and LLDISPLAY_EXTRA_IMPL_exitDrawingMode(void) .
  - [Removed] LLDISPLAY_EXTRA_IMPL_error(int32_t) .
  - [Removed] LLDISPLAY_EXTRA_IMPL_waitPreviousDrawing(void) : implementation has to call LLUI_DISPLAY_notifyAsynchronousDrawingEnd(bool) instead.

- LLDISPLAY_impl.h and intern/LLDISPLAY_impl.h merged in LLUI_DISPLAY_impl.h
  - [Changed] LLDISPLAY_IMPL_initialize(void) : replaced by LLUI_DISPLAY_IMPL_initialize(LLUI_DISPLAY_SInitData*) (_mandatory_).
  - [Changed] LLDISPLAY_IMPL_flush(int32_t,int32_t,int32_t,int32_t,int32_t) : replaced by LLUI_DISPLAY_IMPL_flush(MICROUI_GraphicsContext*,uint8_t*, uint32_t,uint32_t, uint32_t,uint32_t) (_mandatory_).
  - [Removed] LLDISPLAY_IMPL_getWidth(void) , LLDISPLAY_IMPL_getHeight(void) and LLDISPLAY_IMPL_getGraphicsBufferAddress(void) : replaced by elements in structure LLUI_DISPLAY_SInitData .
  - [Removed] LLDISPLAY_IMPL_synchronize(void) : implementation has to call LLUI_DISPLAY_flushDone(bool) instead.

- LLDISPLAY_UTILS.h and intern/LLDISPLAY_UTILS.h merged in LLUI_DISPLAY.h
  - [Changed] LLDISPLAY_UTILS_getBufferAddress(int32_t) : replaced by LLUI_DISPLAY_getBufferAddress(MICROUI_Image*) .
  - [Changed] LLDISPLAY_UTILS_setDrawingLimits(int32_t,int32_t,int32_t,int32_t) : replaced by LLUI_DISPLAY_setDrawingLimits(MICROUI_GraphicsContext*,jint,jint,jint,jint) .
  - [Changed] LLDISPLAY_UTILS_blend(int32_t,int32_t,int32_t) : replaced by LLUI_DISPLAY_blend(uint32_t,uint32_t,uint32_t) .
  - [Changed] LLDISPLAY_UTILS_allocateDecoderImage(void*) : replaced by LLUI_DISPLAY_allocateImageBuffer(MICROUI_Image*,uint8_t) .
  - [Changed] LLDISPLAY_UTILS_flushDone(void) : replaced by LLUI_DISPLAY_flushDone(bool) .
  - [Changed] LLDISPLAY_UTILS_drawingDone(void) : replaced by LLUI_DISPLAY_notifyAsynchronousDrawingEnd(bool) .
  - [Removed] LLDISPLAY_UTILS_getWidth(int32_t) , LLDISPLAY_UTILS_getHeight(int32_t) and LLDISPLAY_UTILS_getFormat(int32_t) : use MICROUI_Image elements instead.
  - [Removed] LLDISPLAY_UTILS_enterDrawingMode(void) and LLDISPLAY_UTILS_exitDrawingMode(void) .
  - [Removed] LLDISPLAY_UTILS_setClip(int32_t,int32_t,int32_t,int32_t,int32_t) .
  - [Removed] LLDISPLAY_UTILS_getClipX1/X2/Y1/Y2(int32_t) : use MICROUI_GraphicsContext elements instead.
  - [Removed] LLDISPLAY_UTILS_drawPixel(int32_t,int32_t,int32_t) and LLDISPLAY_UTILS_readPixel(int32_t,int32_t,int32_t) .
- LLDW_PAINTER_impl.h
  - [Added] List of ej.api.drawing library's native functions implemented in module com.microej.clibrary.llimpl#microui.
- LLLEDS_impl.h and intern/LLLEDS_impl.h merged in LLUI_LED_impl.h
  - [Changed] LLLEDS_MIN_INTENSITY and LLLEDS_MAX_INTENSITY : replaced respectively by LLUI_LED_MIN_INTENSITY and LLUI_LED_MAX_INTENSITY .
  - [Changed] LLLEDS_IMPL_initialize(void) : replaced by LLUI_LED_IMPL_initialize(void) .
  - [Changed] LLLEDS_IMPL_getIntensity(int32_t) : replaced by LLUI_LED_IMPL_getIntensity(jint) .
  - [Changed] LLLEDS_IMPL_setIntensity(int32_t,int32_t) : replaced by LLUI_LED_IMPL_setIntensity(jint,jint) .
- LLINPUT.h and intern/LLINPUT.h merged in LLUI_INPUT.h
  - [Changed] LLINPUT_sendEvent(int32_t,int32_t) : replaced by LLUI_INPUT_sendEvent(jint,jint) .
  - [Changed] LLINPUT_sendEvents(int32_t,int32_t*,int32_t) : replaced by LLUI_INPUT_sendEvents(jint,jint*,jint) .
  - [Changed] LLINPUT_sendCommandEvent(int32_t,int32_t) : replaced by LLUI_INPUT_sendCommandEvent(jint,jint) .

  - [Changed] LLINPUT_sendButtonPressedEvent(int32_t,int32_t) : replaced by LLUI_INPUT_sendButtonPressedEvent(jint,jint) .
  - [Changed] LLINPUT_sendButtonReleasedEvent(int32_t,int32_t) : replaced by LLUI_INPUT_sendButtonReleasedEvent()jint,jint .
  - [Changed] LLINPUT_sendButtonRepeatedEvent(int32_t,int32_t) : replaced by LLUI_INPUT_sendButtonRepeatedEvent(jint,jint) .
  - [Changed] LLINPUT_sendButtonLongEvent(int32_t,int32_t) : replaced by LLUI_INPUT_sendButtonLongEvent(jint,jint) .
  - [Changed] LLINPUT_sendPointerPressedEvent(int32_t,int32_t,int32_t,int32_t,int32_t) : replaced by LLUI_INPUT_sendPointerPressedEvent(jint,jint,jint,jint,LLUI_INPUT_Pointer) .
  - [Changed] LLINPUT_sendPointerReleasedEvent(int32_t,int32_t) : replaced by LLUI_INPUT_sendPointerReleasedEvent(jint,jint) .
  - [Changed] LLINPUT_sendPointerMovedEvent(int32_t,int32_t,int32_t,int32_t) : replaced by LLUI_INPUT_sendPointerMovedEvent(jint,jint,jint,LLUI_INPUT_Pointer) .
  - [Changed] LLINPUT_sendTouchPressedEvent(int32_t,int32_t,int32_t) : replaced by LLUI_INPUT_sendTouchPressedEvent(jint,jint,jint) .
  - [Changed] LLINPUT_sendTouchReleasedEvent(int32_t) : replaced by LLUI_INPUT_sendTouchReleasedEvent(jint) .
  - [Changed] LLINPUT_sendTouchMovedEvent(int32_t,int32_t,int32_t) : replaced by LLUI_INPUT_sendTouchMovedEvent(jint,jint,jint) .
  - [Changed] LLINPUT_sendStateEvent(int32_t,int32_t,int32_t) : replaced by LLUI_INPUT_sendStateEvent(jint,jint,jint) .
  - [Changed] LLINPUT_getMaxEventsBufferUsage(void) : replaced by LLUI_INPUT_getMaxEventsBufferUsage(void) .

- LLINPUT_impl.h and intern/LLINPUT_impl.h merged in LLUI_INPUT_impl.h
  - [Changed] LLINPUT_IMPL_initialize(void) : replaced by LLUI_INPUT_IMPL_initialize(void) (_mandatory_).
  - [Changed] LLINPUT_IMPL_getInitialStateValue(int32_t,int32_t) : replaced by LLUI_INPUT_IMPL_getInitialStateValue(jint,jint) (_mandatory_).
  - [Changed] LLINPUT_IMPL_enterCriticalSection(void) : replaced by LLUI_INPUT_IMPL_enterCriticalSection(void) (_mandatory_).
  - [Changed] LLINPUT_IMPL_leaveCriticalSection(void) : replaced by LLUI_INPUT_IMPL_leaveCriticalSection(void) (_mandatory_).

- LLUI_DISPLAY.h
  - [Added] Renaming of LLDISPLAY_UTILS.h .
  - [Added] Several functions to interact with the Graphics Engine and to get information on images, graphics context, clip, etc.
  - [Added] LLUI_DISPLAY_requestFlush(bool) : requests a call to LLUI_DISPLAY_IMPL_flush() .
  - [Added] LLUI_DISPLAY_requestRender(void) : requests a call to Displayable.render() .
  - [Added] LLUI_DISPLAY_freeImageBuffer(MICROUI_Image*) : frees an image previously allocated by LLUI_DISPLAY_allocateImageBuffer(MICROUI_Image*,uint8_t) .

  - [Added] LLUI_DISPLAY_requestDrawing(MICROUI_GraphicsContext*,SNI_callback) : allows to take the hand on the shared back buffer.
  - [Added] LLUI_DISPLAY_setDrawingStatus(DRAWING_Status) : specifies the drawing status to the Graphics Engine.
- LLUI_DISPLAY_impl.h
  - [Added] Merge of LLDISPLAY_EXTRA_impl.h and LLDISPLAY_impl.h .
  - [Added] Structure LLUI_DISPLAY_SInitData : implementation has to fill it in LLUI_DISPLAY_IMPL_initialize(LLUI_DISPLAY_SInitData*) .
  - [Added] LLUI_DISPLAY_IMPL_binarySemaphoreTake(void*) and LLUI_DISPLAY_IMPL_binarySemaphoreGive(void*,bool) : implementation has to manage a binary semaphore (_mandatory_).
  - [Added] LLUI_DISPLAY_IMPL_getNewImageStrideInBytes(MICROUI_ImageFormat, uint32_t,uint32_t,uint32_t) : allows to set an image stride different than image side (_optional_).
- LLUI_PAINTER_impl.h
  - [Added] List of ej.api.microui library's native functions implemented in module com.microej.clibrary.llimpl#microui.
  - [Added] MICROUI_ImageFormat : MicroUI Image pixel format.
  - [Added] MICROUI_Image : MicroUI Image representation.
  - [Added] MICROUI_GraphicsContext : MicroUI GraphicsContext representation.
- ui_drawing_soft.h
  - [Added] List of internal Graphics Engine software algorithms to perform some drawings (related to library ej.api.microui).
- ui_drawing.h
  - [Added] List of ej.api.microui library's drawing functions to optionally implement in VEE Port.

Custom Native Drawing Functions

- In custom UI native methods, replace LLDISPLAY_UTILS_getBufferAddress(xxx); by (uint32_t)LLUI_DISPLAY_getBufferAddress(xxx) (new function returns uint8_t*), where uint32_t xxx is replaced by MICROUI_Image* xxx or by MICROUI_GraphicsContext* xxx .
- Replace LLDISPLAY_UTILS_getFormat(xxx) by xxx->format , where uint32_t xxx is replaced by MICROUI_Image* xxx or by MICROUI_GraphicsContext* xxx .
- Replace call to LLDISPLAY_allocateDecoderImage by a call to LLUI_DISPLAY_allocateImageBuffer
- Optional: implement drawing functions listed in ui_drawing.h following the available examples in LLUI_PAINTER_impl.c and LLDW_PAINTER_impl.c files comments.

Application

- See application *Migration Guide*.

From 11.x to 12.1.5

VEE Port Configuration Project

- Update Architecture version: 7.11.0 or higher.

Front Panel

- Create a new Front Panel Project (next sections explain how to update each widget):

1. Verify that FrontPanelDesigner is at least version 6: Help > About > Installations Details > Plug-ins .
2. Create a new front panel project: File > New > Project… > MicroEJ > MicroEJ Front Panel Project , choose a name and press Finish .
3. Move files from [old project]/src to [new project]/src/main/java .
4. Move files from [old project]/resources to [new project]/src/main/resources .
5. Move files from [old project]/definitions to [new project]/src/main/resources , **except** your xxx.fp file.
6. If existing delete file [new project]/src/main/java/microui.properties .
7. Delete file [new project]/src/main/resources/.fp.xsd .
8. Delete file [new project]/src/main/resources/.fp1.0.xsd .
9. Delete file [new project]/src/main/resources/widgets.desc .
10. Open [old project]/definitions/xxx.fp .
11. Copy device attributes (name and skin) from [old project]/definitions/xxx.fp to [new project]/src/main/resources/xxx.fp .
12. Copy content of body (not body tag itself) from [old project]/definitions/xxx.fp under device group of [new project]/src/main/resources/xxx.fp .

- Widget “led2states”:

1. Rename led2states by ej.fp.widget.LED .
2. Rename the attribute id by label .

- Widget “pixelatedDisplay”:

1. Rename pixelatedDisplay by ej.fp.widget.Display .
2. Remove the attribute id .
3. (*if set*) Remove the attribute initialColor if its value is 0
4. (*if set*) Rename the attribute mask by filter ; this image must have the same size in pixels than display itself (width * height).

5. (*if set*) Rename the attribute realWidth by displayWidth .
6. (*if set*) Rename the attribute realHeight by displayHeight .
7. (*if set*) Rename the attribute transparencyLevel by alpha ; change the value: newValue = 255 - oldValue .
8. (*if set*) Remove the attribute residualFactor (not supported).
9. (*if set*) If extensionClass is specified: follow next notes.

- Widget "pixelatedDisplay": ej.fp.widget.Display Extension Class:

1. Open the class
2. Extends ej.fp.widget.MicroUIDisplay.AbstractDisplayExtension instead of com.is2t.microej.frontpanel.display.DisplayExtension .
3. Rename method convertDisplayColorToRGBColor to convertDisplayColorToARGBColor .
4. Rename method convertRGBColorToDisplayColor to convertARGBColorToDisplayColor .

- Widget "pointer":

1. Rename pointer by ej.fp.widget.Pointer .
2. Remove the attribute id .
3. (*if set*) Rename the attribute realWidth by areaWidth .
4. (*if set*) Rename the attribute realHeight by areaHeight .
5. Keep or remove the attribute listenerClass according next notes.

- Widget "pointer": ej.fp.widget.Pointer Listener Class:

  This extension class is useless if the implementation respects these rules:

  – *(a)* press method is sending a press MicroUI Pointer event.
  – *(b)* release method is sending a release MicroUI Pointer event.
  – *(c)* move method is sending a move MicroUI Pointer event.
  – *(d)* The MicroUI Pointer event generator name is POINTER when ej.fp.widget.Pointer 's touch attribute is false (or not set).
  – *(e)* The MicroUI Pointer event generator name is TOUCH when ej.fp.widget.Pointer 's touch attribute is true .

  If only *(d)* or *(e)* is different:

1. Open the listener class.
2. Extends the class ej.fp.widget.Pointer.PointerListenerToPointerEvents instead of implementing the interface com.is2t.microej.frontpanel.input.listener.PointerListener .
3. Implements the method getMicroUIGeneratorTag() .

   In all other cases:

1. Open the listener class.
2. Implements the interface ej.fp.widget.Pointer.PointerListener instead of com.is2t.microej.frontpanel.input.listener.PointerListener .

- Widget "push":

1. Rename push by ej.fp.widget.Button .
2. Rename the attribute id by label .
3. (*if set*) Review filter image: this image must have the same size in pixels than the button skin .
4. (*if set*) Remove the attribute hotkey (not supported).
5. Keep or remove the attribute listenerClass according next notes.

- Widget "push": ej.fp.widget.Button Listener Class:

  This extension class is useless if the implementation respects these rules:

  – *(a)* press method is sending a press MicroUI Buttons event with button label (equals to old button id) as button index.

  – *(b)* release method is sending a release MicroUI Buttons event with button label (equals to old button id) as button index.

  – *(c)* The MicroUI Buttons event generator name is BUTTONS .

  If only *(c)* is different:

1. Open the listener class.
2. Extends the class ej.fp.widget.Button.ButtonListenerToButtonEvents instead of implementing the interface com.is2t.microej.frontpanel.input.listener.ButtonListener .
3. Overrides the method getMicroUIGeneratorTag() .

   In all other cases:

1. Open the listener class.
2. Implements the interface ej.fp.widget.Button.ButtonListener instead of com.is2t.microej.frontpanel.input.listener.ButtonListener .

- Widget "repeatPush":

1. Rename repeatPush by ej.fp.widget.RepeatButton .
2. (*if set*) Remove the attribute sendPressRelease (not supported).
3. Same rules than widget *push*.

- Widget "longPush":

1. Rename longPush by ej.fp.widget.LongButton .
2. Same rules than widget *push*.

- Widget "joystick":

1. Rename joystick by ej.fp.widget.Joystick .
2. Remove the attribute id .
3. (*if set*) Rename the attribute mask by filter ; this image must have the same size in pixels than joystick skin .
4. (*if set*) Remove the attribute hotkeys (not supported).
5. Keep or remove the attribute listenerClass according next notes.

- Widget "joystick": ej.fp.widget.Joystick Listener Class:

  This extension class is useless if the implementation respects these rules:

- *(a)* press methods are sending some MicroUI Command events UP, DOWN, LEFT, RIGHT and SELECT.
- *(b)* repeat methods are sending same MicroUI Command events UP, DOWN, LEFT, RIGHT and SELECT.
- *(c)* release methods are sending nothing.
- *(d)* The MicroUI Command event generator name is JOYSTICK.

If only *(d)* is different:

1. Open the listener class
2. Extends the class ej.fp.widget.Joystick.JoystickListenerToCommandEvents instead of implementing the interface com.is2t.microej.frontpanel.input.listener.JoystickListener.
3. Overrides the method getMicroUIGeneratorTag().

In all other cases:

1. Open the listener class.
2. Implements the interface ej.fp.widget.Joystick.JoystickListener instead of com.is2t.microej.frontpanel.input.listener.JoystickListener.

- Others Widgets:

  These widgets may have not been migrated. Check in ej.tool.frontpanel.widget library if some widgets are compatible or write your own widgets.

Application

- See application *Migration Guide*.

From 10.x to 11.2.0

VEE Port Configuration Project

- Update Architecture version: 7.0.0 or higher.

From 9.x to 10.0.2

VEE Port Configuration Project

- Update Architecture version: 6.13.0 or higher.
- Edit display/display.properties
- Add property imagesHeap.size=xxx; this value fixes the images heap size when using the VEE Port in command line (to build a firmware)
- In VEE Port linker file (standalone mode with MicroEJ linker): remove the image heap reserved section and put the section .bss.microui.display.imagesHeap instead.

BSP

- In BSP linker file: remove the image heap reserved section and put the section `.bss.microui.display.imagesHeap` instead
- Edit `LLDISPLAY*.c` : remove the functions `LLDISPLAY_IMPL_getWorkingBufferStartAddress` and `LLDISPLAY_IMPL_getWorkingBufferEndAddress`

Application

- See application *Migration Guide*.

From 8.x to 9.4.1

VEE Port Configuration Project

- Update Architecture version: 6.13.0 or higher.

Application

- See application *Migration Guide*.

From 7.x to 8.1.0

VEE Port Configuration Project

- Update Architecture version: 6.4.0 or higher.
- Edit `display/display.properties` : remove property `mode=xxx`

BSP

- Edit `LLDISPLAY*.c`
- For LLDISPLAY `SWITCH`
  - Remove the function `LLDISPLAY_SWITCH_IMPL_getDisplayBufferAddress()`
  - Replace the function `void LLDISPLAY_SWITCH_IMPL_getDisplayBufferAddress()` by `int32_t LLDISPLAY_IMPL_flush()`
  - In this function, return the old front buffer address
  - Replace the function `LLDISPLAY_COPY_IMPL_getBackBufferAddress()` by `LLDISPLAY_IMPL_getGraphicsBufferAddress()`
- For LLDISPLAY `COPY`
  - Replace the function `void LLDISPLAY_COPY_IMPL_copyBuffer()` by `int32_t LLDISPLAY_IMPL_flush()`
  - In this function, return the back buffer address (given in argument)

  - Replace the function LLDISPLAY_COPY_IMPL_getBackBufferAddress() by LLDISPLAY_IMPL_getGraphicsBufferAddress()
- For LLDISPLAY DIRECT
  - Add the function void LLDISPLAY_IMPL_synchorize(void) (do nothing)
  - Add the function int32_t LLDISPLAY_IMPL_flush()
  - In this function, just return the back buffer address (given in argument)
- Replace h file LLDISPLAY_SWITCH_IMPL.h , LLDISPLAY_COPY_IMPL.h or LLDISPLAY_DIRECT_IMPL.h by LLDISPLAY_IMPL.h
- Replace all functions LLDISPLAY_SWITCH_IMPL_xxx , LLDISPLAY_COPY_IMPL_xxx and LLDISPLAY_DIRECT_IMPL_xxx by LLDISPLAY_IMPL_xxx
- Remove the argument int32_t type from getWidth and getHeight

STM32 VEE Ports with DMA2D only

- In VEE Port configuration project, edit display/display.properties
- Add property hardwareAccelerator=dma2d
- In BSP project, edit LLDISPLAY*.c
- simplify following functions (see STM32F7Discovery board implementation)

```
LLDISPLAY_EXTRA_IMPL_fillRect
LLDISPLAY_EXTRA_IMPL_drawImage
LLDISPLAY_EXTRA_IMPL_waitPreviousDrawing
```

- Add the following function

```
void LLDISPLAY_EXTRA_IMPL_error(int32_t errorCode)
{
  printf("lldisplay error: %d\n", errorCode);
  while(1);
}
```

- Launch an application with images and fillrect
- Compile, link and debug the BSP
- Set some breakpoints on three functions
- Ensure the functions are called

3.6.16 Vector Graphics

Principle

The Vector Graphics Pack features an extension of the *User Interface Pack* that implements the *MicroVG API*.

The diagram below shows a simplified view of the components involved in the provisioning of Vector Graphics Extension.

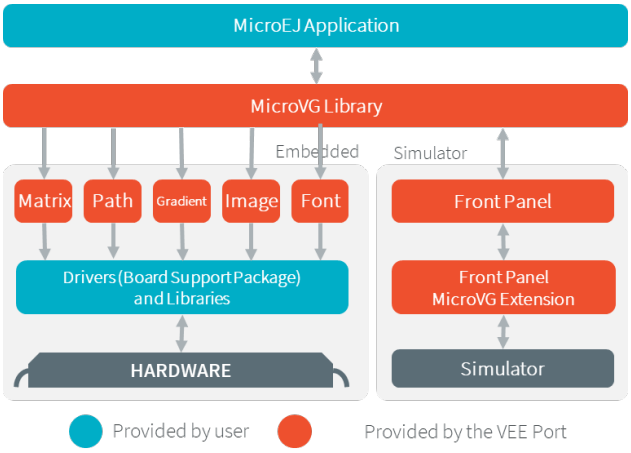


Fig. 239: Overview

The modules responsible to manage the Matrix, the Path, the Gradient, the Image and the Font are respectively called *Matrix module*, *Path module*, *Gradient module*, *Image module* and *Font module*. These five low-level parts connect the MicroVG library to the user-supplied drivers code (coded in C). The drivers can use hardware accelerators like GPU to perform specific actions (matrix computations, path rendering, font decoding, etc.).

The MicroEJ Simulator provides all features of the MicroVG library. The five modules are grouped in a module called *Front Panel*. The Front Panel is an extension of the *UI Pack's Front Panel mock*.

MicroVG

Principle

MicroVG library is an extension of the MicroUI library and provides vector drawing capabilities.

Architecture

MicroVG library is the entry point to perform some vectorial drawings on display. This library contains only a minimal set of basic APIs. As a result, high-level libraries can be used to have more expressive power. In addition to this restricted set of APIs, the MicroVG implementation has been designed to minimize the EDC, BON, and MicroUI footprint.

Native Calls

Like MicroUI, the MicroVG implementation for MicroEJ uses native methods to perform some actions (manipulate matrices, perform drawings, decode and render fonts, etc.). The library implementation has been designed not to block native methods (wait until the end of the drawing, etc.), which can lock the complete Core Engine execution.

Refer to the *MicroUI implementation* to have more details about the native calls.

Installation

The MicroVG *Pack* module must be installed in your VEE Port:

SDK 6 (build.gradle.kts)

SDK 5 (module.ivy)

```
microejPack("com.microej.pack.vg:vg-pack:[VG Pack version]")
```

```
<dependency org="com.
↪microej.pack.vg" name="vg-pack" rev="[VG Pack version]" conf="default->default"/>
```

Note: The latest version of the VG Pack is 1.8.1.

The VG Pack will be automatically available after a VEE Port rebuild.

When installed, the MicroVG Pack module must be configured.

SDK 6

SDK 5

In SDK 6, the configuration is done in the properties file `configuration.properties` of the VEE Port project. All the properties names listed below must be prefixed by `com.microej.pack.microvg.` . For example the `implementation` properties is defined by the `com.microej.pack.microvg.implementation` property.

In SDK 5, the configuration is done in the properties file `microvg/microvg.properties` .

This configuration allows to configure the *Image Generator* and the front panel to fit a specific GPU. This properties file must contain a property named `implementation` . Two values are currently available:

- `nema` : to be compatible with the *Think Silicon Nema VG* GPU.
- `vglite` : to be compatible with the *Vivante VGLite* GPU.

Example:

SDK 6

SDK 5

```
com.microej.pack.microvg.implementation=nema
```

```
implementation=nema
```

A custom extension can be used to target another GPU. The name of the property `implementation` is used to identify the *Custom Extension* and the *Front Panel Extension*.

Use

See *MicroVG* chapter in Application Developer Guide.

Abstraction Layer API

Principle

The MicroVG implementation for MicroEJ requires an Abstraction Layer implementation. The Abstraction Layer implementation consists of a set of header files to implement in C to target the hardware drivers.

The VG Pack's embedded Front Panel extension implements all MicroVG features for the simulator.

Embedded VEE Port

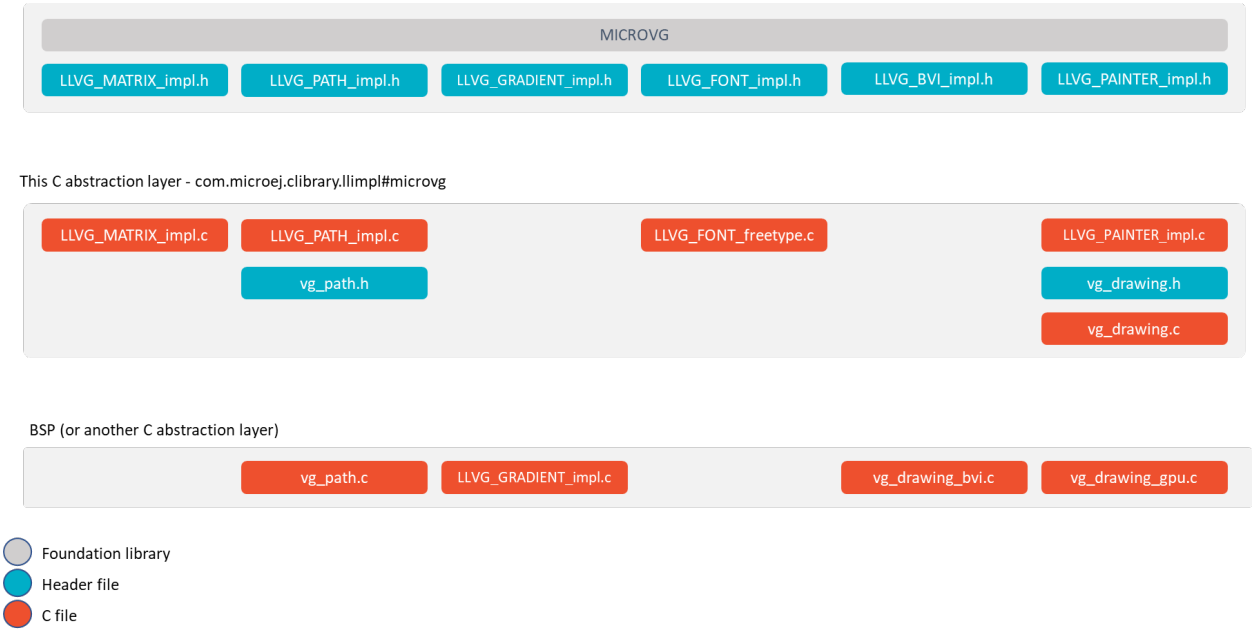


Fig. 240: MicroVG Embedded Abstraction Layer API

The specification of header files names is:

- Name starts with `LLVG_`.
- Second part's name refers to the VG engine: `MATRIX`, `PATH`, `GRADIENT`, `BVI` (image), `FONT`.
- All file's name ends with `_impl`: all functions must be implemented over hardware or in software.

A *master* header file initializes the native Vector Graphics engine: see *LLVG: VectorGraphics*. All other header files and their aims are described in next VG engines chapters: *Matrix*, *Path*, *Gradient*, *Image* and *Font*.

Simulator

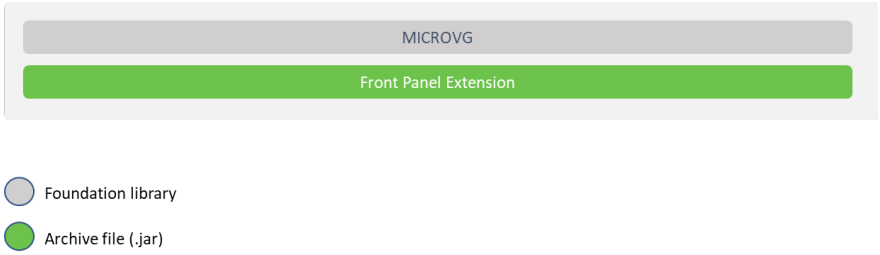


Fig. 241: MicroVG Simulator Abstraction Layer API

The Simulator's five VG engines are grouped in a *Front Panel extension*.

This engine provides the compatibility with Vivante VGLite and Think Silicon Nema VG GPUs and it can be extended to fit another GPU.

Matrix

Principle

The Matrix module contains the C part of the MicroVG implementation, which manages arithmetics matrices. This module is composed of only one element: an implementation of Abstraction Layer APIs to create and manipulate the matrices.

Functional Description

The Matrix module implements the framework of the MicroVG Matrix. It provides Abstraction Layer APIs that manipulate the matrices: fill an identity matrix, do a translation, a rotation, or a scaling and concatenate two matrices.

A matrix is a 3x3 matrix, and its elements are encoded in `float` (32-bit values):

- matrix_memory[0] = matrix[0][0];
- matrix_memory[1] = matrix[0][1];
- matrix_memory[2] = matrix[0][2];
- matrix_memory[3] = matrix[1][0];
- matrix_memory[4] = matrix[1][1];
- matrix_memory[5] = matrix[1][2];
- matrix_memory[6] = matrix[2][0];
- matrix_memory[7] = matrix[2][1];
- matrix_memory[8] = matrix[2][2];

The buffer where the matrix is encoded is stored in the Managed Heap.

Abstraction Layer API

The Abstraction Layer APIs that have to be implemented are listed in the header file LLVG_MATRIX_impl.h (see *LLVG_MATRIX: Matrix*):

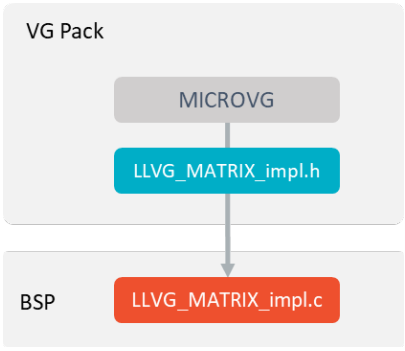


Fig. 242: Matrix Abstraction Layer API

- MicroVG library calls the BSP functions through the header file LLVG_MATRIX_impl.h .
- The *C module MicroVG* provides a default implementation of this header file: LLVG_MATRIX_impl.c .
- This file is automatically copied in the BSP project when fetching the C module during the VEE Port build.

Use

The MicroVG Matrix APIs are available in the class ej.microvg. Matrix.

Path

Principle

The Path module contains the C part of the MicroVG implementation, which manages vector paths. This module is composed of two elements:

- an implementation of Abstraction Layer APIs to create path elements compatible with the hardware,
- an implementation of Abstraction Layer APIs for MicroVG drawings.

Functional Description

The Path module implements the framework of the MicroVG Path. It provides Abstraction Layer APIs that create and merge some paths in a VEE Port-specific format. After the path creation and encoding, the path data should not change when the application draws it: the encoded format should be used by the VEE Port-specific implementation (generally GPU).

A path is a succession of commands. The command encoding is implementation specific; however, the float format is recommended.

List of commands:

- LLVG_PATH_CMD_CLOSE : MicroVG “CLOSE” command.
- LLVG_PATH_CMD_MOVE : MicroVG “MOVE ABS” command.
- LLVG_PATH_CMD_MOVE_REL : MicroVG “MOVE REL” command.
- LLVG_PATH_CMD_LINE : MicroVG “LINE ABS” command.
- LLVG_PATH_CMD_LINE_REL : MicroVG “LINE REL” command.
- LLVG_PATH_CMD_QUAD : MicroVG “QUAD ABS” command.
- LLVG_PATH_CMD_QUAD_REL : MicroVG “QUAD REL” command.
- LLVG_PATH_CMD_CUBIC : MicroVG “CUBIC ABS” command.
- LLVG_PATH_CMD_CUBIC_REL : MicroVG “CUBIC REL” command.

The buffer where the commands are encoded is stored in the Managed Heap. The buffer size is automatically increased by the MicroVG implementation when no more commands can be added.

A path is drawn with a color or with a *linear gradient*.

Abstraction Layer API

There are two separate Abstraction Layer API header files (see *LLVG_PATH: Vector Path*):

- LLVG_PATH_impl.h specifies the Abstraction Layer APIs used to create and encode the path.
- LLVG_PAINTER_impl.h lists the Abstraction Layer APIs called by VectorGraphicsPainter to draw the path.

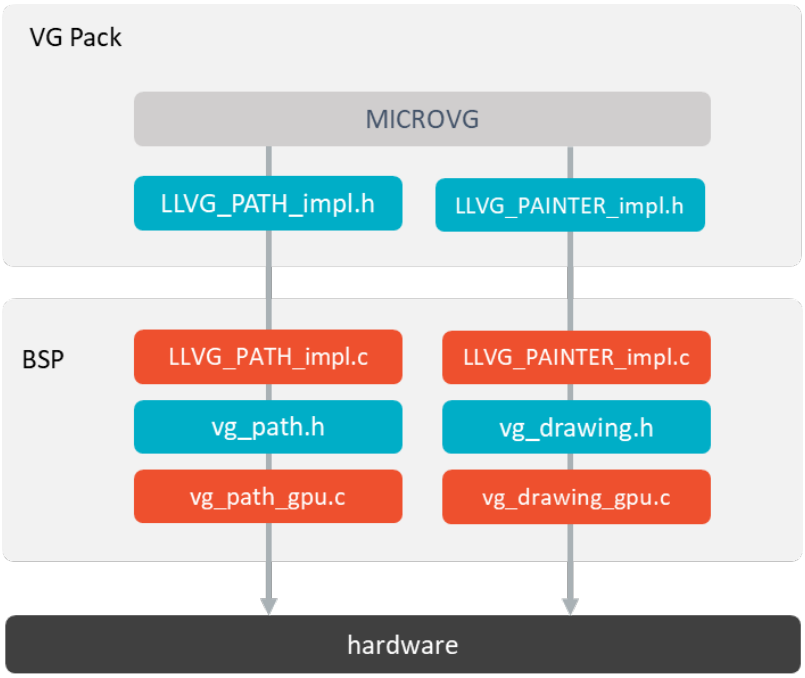


Fig. 243: Path Abstraction Layer API

- MicroVG library calls the BSP functions through the header files LLVG_PATH_impl.h and LLVG_PAINTER_impl.h .

- The *C module MicroVG* provides a default implementation of LLVG_PATH_impl.h : it manages the path buffer creation and filling, then redirect the command encoding to vg_path.h .
- This C module also provides an implementation of LLVG_PAINTER_impl.c that synchronizes the drawing with the MicroUI Graphics Engine and redirects the drawing itself to a third-party drawer through vg_drawing.h .
- A C module dedicated to a GPU provides an implementation of this drawer (vg_drawing_gpu.c) that implements the drawings over the GPU library (it also manages the *Gradient*).
- This dedicated GPU C module provides an implementation of vg_path.h (vg_path_gpu.c) that encodes the path commands.
- These files are automatically copied in the BSP project when fetching the C modules during the VEE Port build.

Use

The MicroVG Path APIs are available in the class ej.microvg. Path.

Gradient

Principle

The Gradient module contains the C part of the MicroVG implementation, which manages linear gradients. This module is composed of only one element: an implementation of the Abstraction Layer APIs to create gradient elements compatible with the hardware.

Functional Description

The Gradient module implements the framework of the MicroVG LinearGradient. It provides Abstraction Layer APIs that consist in creating a linear gradient in a VEE Port-specific format. After the gradient creation and encoding, the gradient data should not change when the application draws it: the encoded format should be used by the VEE Port-specific implementation (generally GPU).

A linear gradient is a succession of colors at different positions. The colors from the MicroVG library implementation are encoded in the 32-bit format: ARGB8888. The color encoding in the gradient is a VEE Port-specific implementation.

The buffer where the gradient is encoded is stored in the Managed Heap. The MicroVG implementation on demand automatically increases the buffer size.

Abstraction Layer API

There are two separate Abstraction Layer API header files (see *LLVG_GRADIENT: Vector Linear Gradient*):

- LLVG_GRADIENT_impl.h specifies the Abstraction Layer APIs used to create and encode the gradient.
- LLVG_PAINTER_impl.h lists the Abstraction Layer APIs called by VectorGraphicsPainter to draw the path.

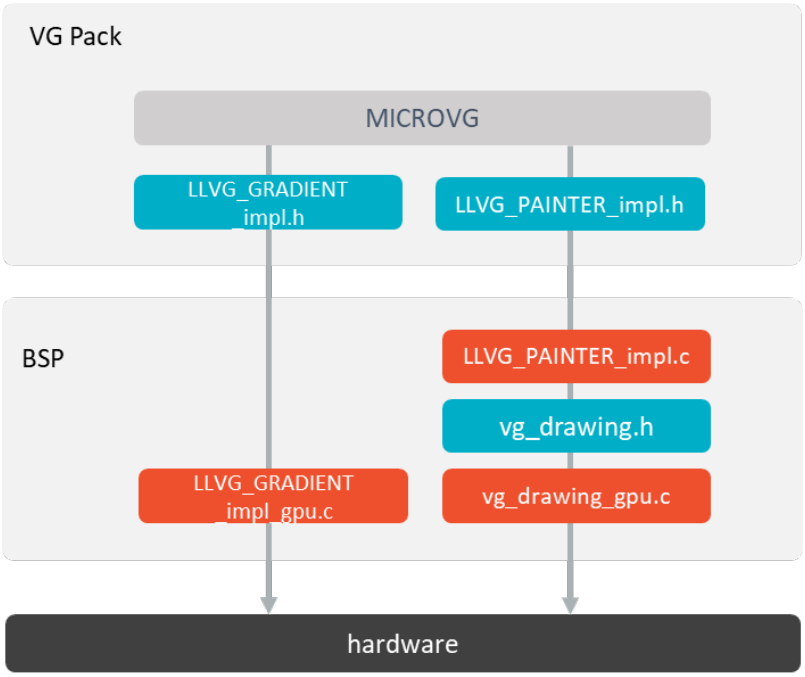


Fig. 244: Gradient Abstraction Layer API

- MicroVG library calls the BSP functions through the header files LLVG_GRADIENT_impl.h and LLVG_PAINTER_impl.h .
- The *C module MicroVG* provides an implementation of LLVG_PAINTER_impl.c that synchronizes the drawing with the MicroUI Graphics Engine and redirects the drawing itself to a third-party drawer through vg_drawing.h ..
- A C module dedicated to a GPU provides an implementation of this drawer (vg_drawing_gpu.c) that implements the drawings over the GPU library.
- This dedicated GPU C module provides an implementation of LLVG_GRADIENT_impl.h (LLVG_GRADIENT_impl_gpu.c) that encodes the gradient.
- These files are automatically copied in the BSP project when fetching the C modules during the VEE Port build.

Use

The MicroVG Gradient APIs are available in the class ej.microvg. LinearGradient.

Image

Principle

The Image module contains the part of the MicroVG implementation which manages vectorial images. This module is composed of several elements:

- an *offline tool* that converts standard vector images in a binary format compatible with the *Rendering Engine*,
- an implementation of Abstraction Layer APIs to manipulate image files,
- an implementation of Abstraction Layer APIs for MicroVG drawings.

Compile-time Image

Overview

The Image module implements the MicroVG VectorImage framework. It provides an offline tool that consists in opening and decoding an image file and some Abstraction Layer APIs that manipulate the image at runtime.

A compile-time image file:

- is either an AVD (Android Vector Drawable) or a Scalable Vector Graphics (SVG),
- is identified by the resource name,
- is encoded in a binary format compatible with the *image renderer*,
- can be stored as an internal resource or an external one (see *External Memory*),
- is an immutable image: the application cannot draw into it.

Image Generator

The offline tool is an extension of the MicroUI *Image Generator*. This tool is automatically installed during the VEE Port build.

The tool converts:

- The Android Vector Drawable (AVD): this kind of image can hold linear gradients, animations on colors, opacity, path transformations, etc.
- The Scalable Vector Graphics (SVG): this kind of image is partially supported: linear gradients but no animations. It is advised to convert the SVG files into AVD files before using the Image Converter tool.

The tool generates a binary (RAW) file compatible with the *Rendering Engine*. The RAW file consists in a series of vector paths and animations.

To list the images to convert, the tool uses the application list files whose extension is .vectorimage.list . The generator provides an option to encode the path data (the path's points):

it can be stored on signed 8, 16, 32-bit words or in `float` format. Respectively, the options are `VG8`, `VG16`, `VG32` and `VGF`.

This is an example of a `vectorimage.list` file:

```
# Convert an AVD in float format
/avd_image_1.xml:VGF
# Convert an AVD in signed 16-bit format
/path/to/avd_image_2.xml:VG16
# Convert an SVG in signed 8-bit format
/svg_image.svg:VG8
```

The image generator must be extended to encode the binary data in a format compatible with the GPU.

Default Extensions

The image generator provides some implementations that targets different GPUs:

- `NemaImageGenerator` : generates binary files compatible with the *C Module: MicroVG Over NemaVG*.
- `VgliteImageGenerator` : generates binary files compatible with the *C Module: MicroVG Over VGLite*.

Refer to the chapter *Installation* for more information about the image generator configuration or to the next chapter to target another GPU.

Custom Extension

The Image Generator can be extended to target a new GPU. This extension follows the same rules than the *MicroUI Image Generator extension*.

1. Create a `std-javalib` project. The module name must start with the prefix `imageGenerator` (for instance `imageGeneratorMyGPU`).
2. Add the dependency:

```
<dependency org="com.microej.pack.vg" name="vg-pack" rev="x.y.z">
  <artifact name="vg-imageGenerator" type="jar"/>
</dependency>
```

Where `x.y.z` is the VG Pack version used to build the VEE Port (minimum `1.6.0`). The `module.ivy` should look like:

```
<ivy-module version="2.0" xmlns:ea="http://www.easyant.org" xmlns:m="http://
↪www.easyant.org/ivy/maven" xmlns:ej="https://developer.microej.com" ej:version="2.0.0">

  <info organisation="com.
↪microej.microui" module="imageGeneratorMyGPU" status="integration" revision="1.0.0">
    <ea:build␣
↪organisation="com.is2t.easyant.buildtypes" module="build-std-javalib" revision="2.+"/>
  </info>

  <configurations defaultconfmapping="default->default;provided->provided">
    <conf name=
↪"default" visibility="public" description="Runtime dependencies to other artifacts"/>
    <conf name="provided" visibility=
```

(continues on next page)

(continued from previous page)

```
↪"public" description="Compile-time dependencies to APIs provided by the VEE Port"/>
    <conf name="documentation
↪" visibility="public" description="Documentation related to the artifact (javadoc, PDF)"/>
    <conf name="source" visibility="public" description="Source code"/>
    <conf name=
↪"dist" visibility="public" description="Contains extra files like README.md, licenses"/>
    <conf name="test" visibility=
↪"private" description="Dependencies for test execution. It is not required for normal use␣
↪of the application, and is only available for the test compilation and execution phases."/>
  </configurations>

  <publications/>

  <dependencies>
    <dependency org="com.microej.pack.vg" name="vg-pack" rev="[VG Pack version]">
      <artifact name="vg-imagegenerator" type="jar"/>
    </dependency>
  </dependencies>
</ivy-module>
```

3. Implements the interface `ej.microvg.image.ImageGenerator` ; the name of the class must be `ej.microvg.image.[Prefix]ImageGenerator` where `[Prefix]` is the name that will be set in the VEE Port configuration file (see *Installation*).
4. Build the project.
5. Copy the generated jar: `target~/artifacts/imageGeneratorMyGPU.jar` in the VEE Port configuration project folder: `MyVEEPort-configuration/dropins/tools/`
6. Rebuild the VEE Port.

MicroVG Library

To load this kind of image, the application has to call VectorImage.getImage(). This API takes the image relative path: `/avd_image_1.xml` or `/path/to/avd_image_2.xml` or `/svg_image.svg` .

The implementation uses the Abstraction Layer API to retrieve the image. No data is stored in the Managed Heap (except the VectorImage object's instance).

Resource Vector Image

The Image module implements the MicroVG ResourceVectorImage framework.

Filtered Image

MicroVG VectorImage.filterImage() API allows to transform an image using a 4x5 color matrix. The result of the image transformation is stored in the MicroUI *Images Heap*. MicroVG ports for dedicated GPU (Low Level implementation) are responsible of the deallocation of this generated image. Implementations are available for *MicroVG Over VGLite* and *MicroVG Over NemaVG*.

External Memory

Principle

MicroVG provides the API ResourceVectorImage.loadImage(). This is an extension of the compile-time images (the concepts are exactly the same), but it allows a load of a RAW image stored in an external memory that is not byte-addressable.

An external image loaded from byte-addressable memory is processed the same way than any compile-time image. For an image loaded from an external memory which is not byte-addressable, its data must be copied into byte-addressable memory before the image can be used for drawings. By default (see *C Modules*), the image data is copied into MicroUI *Images Heap*. The implementation is responsible for the image's lifecycle: allocation and release (already implemented in the *C Modules*).

Configuration File

Like compile-time images, the *Image Generator* uses a list file whose extension is `.externvectorimages.list`. The rules are exactly the sames than the compile-time images.

Process

The process to open a Vector Image from an external memory is exactly the same than the loading of *an external MicroUI Image*.

The following steps describe how to setup the loading of an external resource from the application:

1. Add the image to the application project resources (typically in the source folder `src/main/resources` and in the package `images`).
2. Create / open the configuration file (e.g. `application.externvectorimages.list`).
3. Add the relative path of the image and its output format (e.g. `/images/myImage.avd:VGF` see *Image Generator*).
4. Build the application: the Image Generator converts the image in RAW format in the external resources folder (`[application_output_folder]/externalResources`).
5. Deploy the external resources to the external memory (SDCard, flash, etc.) of the device.
6. (optional) Configure the *External Resources Loader* to load from this source.

7. Build the application and run it on the device.
8. The application loads the external resource using ResourceVectorImage.loadImage().
9. The image loader looks for the image and copies it in the *images heap* (no copy if the external memory is byte-addressable).
10. The external resource is immediately closed: the image's bytes have been copied in the images heap, or the image's bytes are always available (byte-addressable memory).
11. The application can use the image.
12. The application closes the image: the image is removed from the image heap.

Simulation

The Simulator automatically manages the external resources like internal resources. All images listed in `*.externvectorimages.list` files are copied in the external resources folder, and this folder is added to the Simulator's classpath.

Buffered Vector Image

This image is a ResourceVectorImage that the application can draw into. More specifically, the drawings are not *performed* but *stored*.

The concept consists in storing the compatible MicroUI drawings[1] and all MicroVG drawings into a command list. The application can then play this list of commands applying (or not) a global transformation.

Note: The implementation uses the concept of MicroUI *custom* format (the custom `Format.CUSTOM_7`).

The way to register the drawing commands is strongly linked to the targeted GPU:

- The paths and gradients are stored to be used directly by the GPU to render the image (prevent runtime modifications before the image rendering).
- Depending on the GPU capabilities (a GPU may be able to draw a MicroUI anti-aliased line but not an aliased line), some MicroUI drawing API may be implemented (see *Buffered Image*).

  As a consequence, the implementation is dedicated to the GPU. The *C Modules* provide some implementations, and the Front Panel (for the Simulation) features the same limitations as the embedded side (it is not possible to store a MicroUI drawing in the simulator if the embedded side is not able to perform it).

[1] The compatible MicroUI drawings depend on the GPU Port; see C Modules.

Runtime Image

The third-party library *VectorImageLoader* features an API to load an Android Vector Drawable (AVD) at runtime. This API creates a ResourceVectorImage

This library uses a simple XML parser (for performance and footprint convenience) that limits compatibility with the AVD specification. For instance, this loader does not manage the animations.

The *Vector Image Generator* can generate a compatible AVD file in the `.vectorimage.list`, using `AVD` as output format.

```
# Convert an AVD into a compatible AVD format
/avd_image.xml:AVD
# Convert an SVG into a compatible AVD format
/svg_image.svg:AVD
```

Rendering Engine

The Vector Image Rendering Engine has the responsibility of drawing the vector images. The destination is the display back buffer, a MicroUI *BufferedImage* or a MicroVG BufferedVectorImage.

Three transformations can be applied when drawing a vector image:

- a global path transformation (3x3 matrix)
- a color transformation (4x5 color matrix)
- an opacity (value between 0 and 255)

The *C Modules* and the Front Panel already implement this engine.

Abstraction Layer API

There are two separate Abstraction Layer API header files:

- `LLVG_BVI_impl.h` specifies the Abstraction Layer APIs used to open and manage the BufferedVectorImage cycle-life.
- `LLVG_PAINTER_impl.h` lists the Abstraction Layer APIs called by VectorGraphicsPainter to draw an image (compile-time, runtime, or buffered vector image).

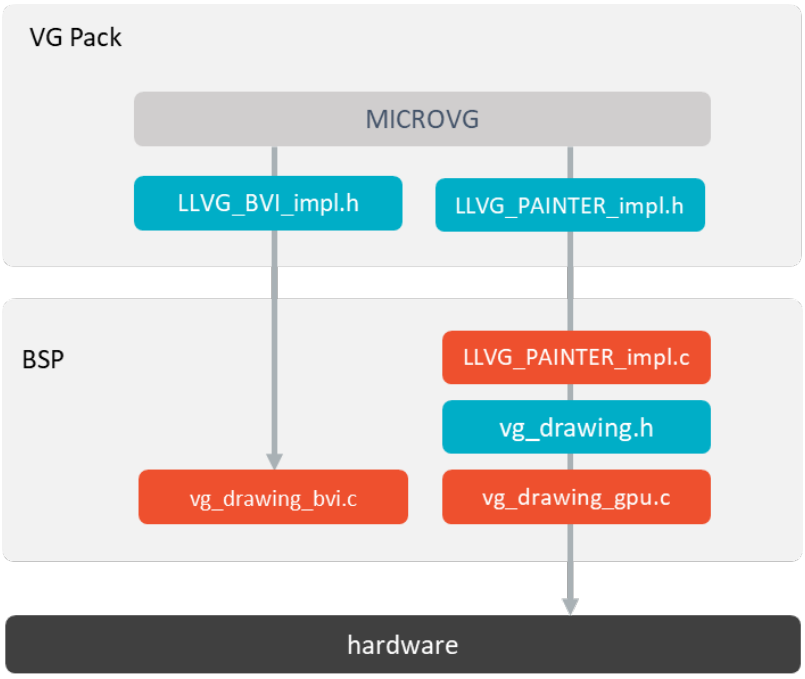


Fig. 245: Image Abstraction Layer API

- MicroVG library calls the BSP functions through the header files LLVG_BVI_impl.h and LLVG_PAINTER_impl.h .
- The *C module MicroVG* an implementation of LLVG_PAINTER_impl.c that synchronizes the drawing with the MicroUI Graphics Engine and redirects the drawing itself to a third-party drawer through vg_drawing.h .
- A C module dedicated to a GPU provides an implementation of this drawer (vg_drawing_gpu.c) that implements the drawings over the GPU library.
- This dedicated GPU C module provides an implementation of LLVG_BVI_impl.h (LLVG_BVI_impl_gpu.c): the implementation is specific to the target (the GPU): the format of the RAW paths, gradients, and animations are GPU compliant.
- These files are automatically copied in the BSP project when fetching the C modules during the VEE Port build.

Simulation

The implementation of the MicroVG library is included in the VG Pack. No specific support is required to retrieve and use the images.

Use

The MicroVG Font APIs are available in the class ej.microvg. VectorImage.

Font

Principle

The Font module contains the C part of the MicroVG implementation, which manages vectorial fonts. This module is composed of two elements:

- an implementation of Abstraction Layer APIs to manipulate font files,
- an implementation of Abstraction Layer APIs for MicroVG drawings.

Functional Description

The Font module implements the MicroVG VectorFont framework. It provides Abstraction Layer APIs that consist in opening and decoding a font file and getting the font's characteristics.

A font file:

- is either a TTF or an OTF,
- is identified by the resource name,
- can be stored as internal resource or external (see *External Fonts*).

No data is stored in the Managed Heap. The implementation is responsible for the font's cycle life: allocation and release.

A font is used to draw a string with a color or with a *linear gradient*.

Abstraction Layer API

There are two separate Abstraction Layer API header files (see *LLVG_FONT: Vector Font*):

- LLVG_FONT_impl.h specifies the Abstraction Layer APIs used to open and retrieve the font's characteristics.
- LLVG_PAINTER_impl.h lists the Abstraction Layer APIs called by VectorGraphicsPainter to draw a string with the font.

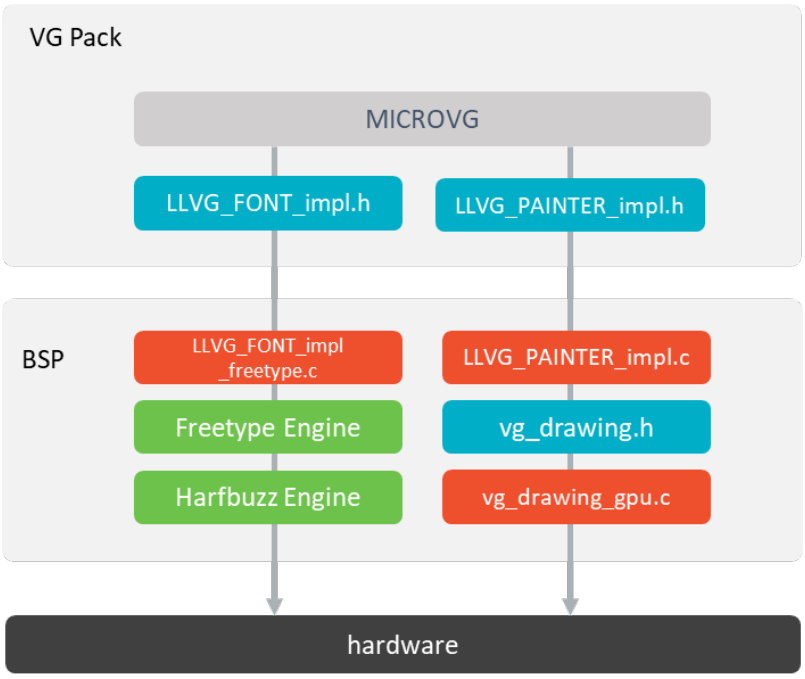


Fig. 246: Font Abstraction Layer API

- MicroVG library calls the BSP functions through the header files `LLVG_FONT_impl.h` and `LLVG_PAINTER_impl.h`.
- The *C module MicroVG* provides a default implementation of `LLVG_FONT_impl.h` over FreeType. It also redirects the *complex layout* to a third party C module.
- This C module also provides an implementation of `LLVG_PAINTER_impl.c` that synchronizes the drawing with the MicroUI Graphics Engine and redirects the drawing itself to a third-party drawer through `vg_drawing.h`.
- A C module dedicated to a GPU provides an implementation of this drawer (`vg_drawing_gpu.c`) that implements the drawings over the GPU library.
- The *C module Harfbuzz* provides an implementation of *complex layout*.
- These files are automatically copied in the BSP project when fetching the C modules during the VEE Port build.

External Memory

Principle

MicroVG does not provide some Low Level API to make the distinction between a font loaded from different kind of memories (internal or external, byte-addressable or not). The Low Level implementation (*C Modules MicroVG and FreeType*) features the font management from an external memory which is not byte-addressable when the VEE Port provides an implementation of the *External Resources Loader*.

Configuration File

A Vector Font file is a *simple* resource. To specify this resource as an external resource, the font file path must be listed in a `.externresources.list` file in addition with the `.resources.list` file (see *Application Resources*).

Process

The following steps describe how to setup the loading of an external resource from the application:

1. Add the font to the application project resources (typically in the source folder `src/main/resources` and in the package `fonts`).
2. Create / open the configuration files (e.g. `application.resources.list` and `application.externresources.list`).
3. In both files, add the relative path of the font (e.g. `/fonts/myFont.ttf`).
4. Build the application: the processed external resources are copied into the external resources folder (`[application_output_folder]/externalResources`).
5. Deploy the external resources to the external memory (SDCard, flash, etc.) of the device.
6. (optional) Configure the *External Resources Loader* to load from this source.
7. Build the application and run it on the device.
8. The application loads the external resource using ej.microvg.VectorFont.loadFont().
9. FreeType (*C Modules*) recognizes this resource as external resource; it configures itself to manage this resource differently than an internal resource (see *Library: FreeType* to have more details).
10. The application can use the font.

Simulation

The Simulator automatically manages the external resources like internal resources. All images listed in `*.externresources.list` files are copied in the external resources folder, and this folder is added to the Simulator's classpath.

Use

The MicroVG Font APIs are available in the class `ej.microvg.VectorFont`.

C Modules

Principle

Several C modules implement the VG Pack's Abstraction Layer APIs. Some are generic, and some are VEE Port dependent (more precisely: GPU dependent). The generic modules provide header files to be extended by the specific modules. The generic C modules are available on the *Central Repository* and the specific C modules on the *Developer Repository*.

The following picture illustrates the available C modules and their relations for an implementation that uses:

- FreeType library for the font renderer and the font layouter in simple layout mode.
- HarfBuzz library for the font layouter in complex layout mode.
- *GPU* library symbolizes the library for the drawing of vector paths over a GPU.

The following chapters explain the aim and relations of each C module.

Note: It is a simplified view: all sources and headers files of each C module are not visible.

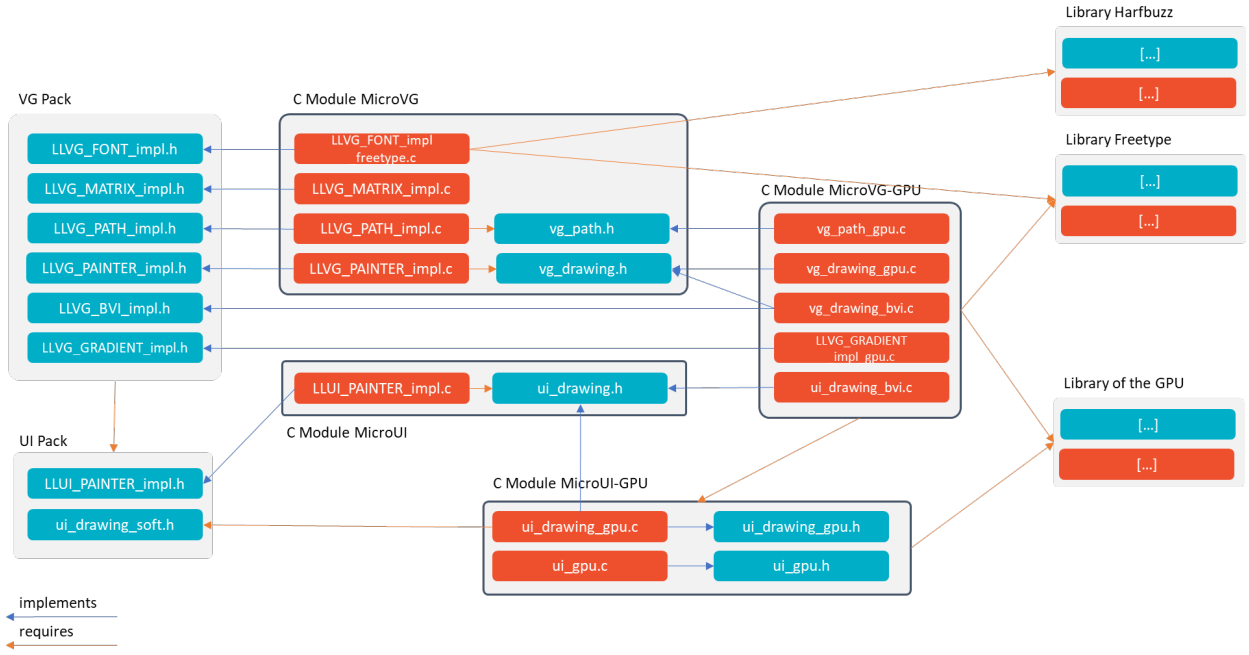


Fig. 247: MicroVG C Modules

UI Pack & MicroUI C Modules

The UI Pack provides a header file to implement the MicroUI drawings: LLUI_PAINTER_impl.h . See *C Modules* chapter to have more information.

VG Pack

The VG Pack provides a set of header files to implement the MicroVG concepts. The header files are described in the dedicated chapters: *Matrix module*, *Path module*, *Gradient module*, *Image module* and *Font module*.

The VG Pack is an extension of the UI Pack. The VG Pack's header files require the UI Pack's header files to manipulate the MicroUI concepts. Consequently, the VG Pack must be installed on a VEE Port that fetches a UI Pack.

The VG Pack and its header files are available on the *Central Repository*: com.microej.pack.vg#vg-pack.

C Module: MicroVG

Description

This generic C module provides an implementation of MicroVG concepts: matrix, path, gradient and font; respectively LLVG_MATRIX_impl.c , LLVG_PATH_impl_single.c / LLVG_PATH_impl_dual.c , LLVG_GRADIENT_impl_stub.c and LLVG_FONT_impl_freetype.c .

- Matrix (see Matrix module's *Abstraction Layer API*): a basic software implementation.
- Path (see Path module's *Abstraction Layer API*): a generic implementation that manages the command buffer's life cycle and dispatches the command encoding to a 3rd-party header file vg_path.h .
- Gradient (see Gradient module's *Abstraction Layer API*): a generic minimal implementation that only handles a single color (resulting in filling paths with a solid color). To fully handle linear gradients, the API from LLVG_GRADIENT_impl.h must be implemented.
- Font (see Font module's *Abstraction Layer API*): an implementation of vector font over FreeType: open font file and retrieve font's characteristics.
- The MicroVG painter native functions are implemented in LLVG_PAINTER_impl.c and the drawings are redirected to vg_drawing.h .
- Image management is too specific to the GPU and is not implemented in this C module.

  This C module is available on the *Central Repository*: com.microej.clibrary.llimpl#microvg.

Dependencies

This generic C module requires some specific modules:

- Path and Gradient require a C module specific to a VEE Port (to a GPU format).
- Font requires the FreeType library and optionally the HarfBuzz library to manage the *complex layout*.

Usage

1. Install the *C Module for MicroUI* and follow its implementation rules.
2. Add all C files in the BSP project.
3. Configure the options in the header file vg_configuration.h .
4. To optionally use the ej.microvg.BufferedVectorImage, the support of *custom image format* is required. As described *here*, set the define UI_GC_SUPPORTED_FORMATS to 2 or more in ui_configuration.h .
5. To optionally use a MicroVG vector font ej.microvg.VectorFont in the MicroUI drawings (for instance ej.microui.display.Painter.drawString()), the support of *custom font format* is required. As described *here*, set the define UI_FEATURE_FONT_CUSTOM_FORMATS in ui_configuration.h .
6. Select one of the GPU implementations (see next chapters).

Library: FreeType

Description

The FreeType library compatible with MicroEJ is packaged in a C module on the *Developer Repository*: com.microej.clibrary.thirdparty#freetype.

This C module provides a fork of FreeType 2.14.1.

Memory Heap Configuration

The FreeType library requires a memory Heap for FreeType internal objects allocated when a font file is loaded (see https://freetype.org/freetype2/docs/design/design-4.html). The size of this heap depends on the number of fonts loaded in parallel and on the fonts themselves. This size is defined by VG_FEATURE_FREETYPE_HEAP_SIZE in vg_configuration.h .

All fonts do not require the same heap size. FreeType heap usage can be monitored using the following configurations:

- MICROVG_MONITOR_HEAP defined in vg_helper.h
- MEJ_LOG_MICROVG and MEJ_LOG_INFO_LEVEL defined in mej_log.h

Principle

1. The Application loads a font with ej.microvg.VectorFont.loadFont().
- If the resource is internal or external from byte-addressable memory, the FreeType library is configured to read directly from that resource memory section.
- Else, if the resource is external from non-byte-addressable memory, the FreeType library is configured to use the *external loader* to read from that memory.
- At this point, the font resources are allocated and the font generic data (including baseline & height metrics) is loaded on the FreeType dedicated heap.
2. The Application requests metrics.
- For generic metrics, already loaded data is directly used (and scaled to the font size used).
- For text-dependent metrics: computed by loading metrics of every glyph required by the input string (the glyphs bitmaps are not actually rendered here).
3. The Application requests drawings.
- For every character to draw:
  - the associated glyph is loaded,
  - the bitmap is rendered for the given font size and
  - the character is drawn in the given graphic context.
4. The Application unloads the font with ej.microvg.VectorFont.close().
- Any resource associated with the font is released.
- At this point, any attempt to use the font will result in an exception.

Library: HarfBuzz

The library HarfBuzz compatible with MicroEJ is packaged in a C module on the *Developer Repository*: com.microej.clibrary.thirdparty#harfbuzz.

This C module provides HarfBuzz 12.3.0 with build scripts patched and additional source files.

The HarfBuzz library requires a memory Heap for HarfBuzz internal objects allocated when a font file is loaded. The size of this heap depends on the number of fonts loaded in parallel and on the fonts themselves. This size is defined by VG_FEATURE_HARFBUZZ_HEAP_SIZE_HEAP in vg_configuration.h .

All fonts do not require the same heap size. The MICROVG_MONITOR_HEAP define in vg_helper.h and MEJ_LOG_MICROVG and MEJ_LOG_INFO_LEVEL defines in mej_log.h can be used to monitor the HarfBuzz heap evolution.

FreeType and HarfBuzz libraries are not sharing the same heap, but this could easilly be done by updating ft_system.c and hb-alloc.c files.

C Module: MicroVG Over VGLite

Overview

This C module is a specific implementation of the VG Pack drawings over the official Vivante VGLite library (that targets some GPU with vector graphics acceleration):

- It implements the MicroVG API LLVG_impl.h in LLVG_impl_vglite.c .
- It implements the MicroVG API LLVG_GRADIENT_impl.h in LLVG_GRADIENT_impl_vglite.c .
- It implements the MicroVG API vg_drawing.h in vg_drawing_vglite.c .
- It implements the MicroVG API vg_path.h in vg_path_vglite.c .
- It implements the MicroVG Image management (draw a compile-time image, create a BufferedVectorImage, etc.) in vg_drawing_vglite_image.c .
- It provides an implementation of MicroVG drawings to the MicroVG BufferedVectorImage: vg_drawing_bvi.c .
- It also implements MicroUI drawings to the MicroVG BufferedVectorImage: ui_drawing_bvi.c .

The implementation requires:

- the concepts of the C module MicroVG,
- the concepts of the C module MicroUI over VGLite,
- the FreeType library,
- the Vivante VGLite library.

This C module is available on the *Developer Repository*: com.microej.clibrary.llimpl#microvg-vglite.

> **Warning:** The library must be patched to be compatible with the C module "MicroUI over VGLite". Consult the C module's ReadMe file for more information.

Usage

1. Install the *C Module for MicroUI for VGLite* and follow its implementation rules.
2. Install the *C Module for MicroVG* and follow its implementation rules.
3. Add all C files in the BSP project.
4. Check the port by running the vg validation as described in the VEE Port project template

C Module: MicroVG Over NemaVG

Overview

This C module is a specific implementation of the VG Pack drawings over the official Think Silicon Nema VG library (that targets some GPU with vector graphics acceleration):

- It implements the MicroVG API `vg_drawing.h` in `vg_drawing_nema.c`.
- It implements the MicroVG Image management (draw a compile-time image, create a BufferedVectorImage, etc.): `vg_drawing_nema_image.c`.
- It provides an implementation of MicroVG drawings to the MicroVG BufferedVectorImage: `vg_drawing_bvi.c`.
- It also implements MicroUI drawings to the MicroVG BufferedVectorImage: `ui_drawing_bvi.c`.

The implementation requires:

- the concepts of the C module MicroVG,
- the concepts of the C module MicroUI over NemaVG,
- the FreeType library,
- the Think Silicon NemaVG library.

This C module is available on the *Developer Repository*: com.microej.clibrary.llimpl#microvg-nema.

Usage

1. Install the *C Module for MicroUI for NemaGFX* and follow its implementation rules.
2. Install the *C Module for MicroVG* and follow its implementation rules.
3. Add all C files in the BSP project.
4. Check the port by running the `vg` validation as described in the VEE Port project template
5. Add the defines `RESX` (== LCD width) and `RESY` (== LCD height) in `vg/inc/vg_configuration.h` or in the C project global options (used to initialize the Nema VG library, see `nema_vg_init()`).

Compatibility

The compatibility between the components (Packs, C modules, and Libraries) is described in the *Release Notes*.

Simulation

Principle

The VG Pack embeds an implementation of the *UI Pack's Front Panel mock* that implements the equivalent of the five embedded modules (Matrix, Path, Gradient, Image and Font). This implementation has to be extended to simulate the same characteristics and limitations as the embedded modules. The aim of this extension is to:

- decode the images encoded for a specific GPU (see *Image Generator*)
- simulate the limitations of the MicroVG implementation over a GPU (unsupported blend modes, MicroUI drawings in a `BufferedVectorImage`, etc.).

Default Extensions

Two extensions are available that simulate the Vivante VGLite (compatible with the *C Module: MicroVG Over VGLite*) and Think Silicon NemaVG (compatible with the *C Module: MicroVG Over NemaVG*) GPUs. To select one of them, set the right name (respectively `vglite` and `nema`) in the VEE Port configuration file (see *Installation*).

Custom Extension

Overview

A custom extension can be selected to simulate another GPU. This extension consists of the implementation of several interfaces and classes. In the VEE Port's Front Panel project:

1. Add the dependency:

```
<dependency org="com.microej.pack.vg" name="vg-pack" rev="x.y.z">
  <artifact name="vg-frontpanel" type="jar"/>
</dependency>
```

   Where `x.y.z` is the VG Pack version used to build the VEE Port (minimum `1.6.0`).

2. Implement one or several interfaces / classes to simulate the embedded GPU (see next chapters).
3. Rebuild the VEE Port.

Image Decoder

To decode the images generarated by the *image generator*, implement the interface `ej.microvg.image.ImageDecoder` in the VEE Port's Front Panel project. The name of the class must be `ej.microvg.image.[Prefix]ImageDecoder` where `[Prefix]` is the name that will be set in the VEE Port configuration file (see *Installation*).

Note: This implementation is mandatory to allow the use of encoded images.

Display Drawer

This feature allows to simulate the same limitations of the GPU to draw the MicroVG drawings (path, gradient, blend modes, etc.) in the display buffer (and in the BufferedImage with same format than the display buffer).

Note: This implementation is optional; by default, all the MicroVG drawings are implemented.

1. Extend the class ej.microvg.display.DisplayDrawer ; the name of the class must be ej.microvg.display.[Prefix]DisplayDrawer where [Prefix] is the name that will be set in the VEE Port configuration file (see *Installation*).
2. Override the method drawPath() to simulate some limitations.

Buffered Vector Image

On the embedded side, the MicroUI drawings (line, rectangle, etc.) must be explicitly implemented to be stored in a BufferedVectorImage (a drawing should not be stored if the GPU is not able to render it). The Front Panel extension allows to simulate the same limitations.

Note: This implementation is optional; by default, all the MicroUI drawings in a BufferedVectorImage are disabled (whereas all the MicroVG drawings are enabled).

The Front Panel extension already provides an engine that implements all MicroUI drawings in a BufferedVectorImage . Each drawing of this extension can be used independently to simulate the behavior of the embedded side.

1. Implement the interface ej.microvg.bvi.BufferedVectorImageDrawer ; the name of the class must be ej.microvg.bvi.[Prefix]BufferedVectorImageDrawer where [Prefix] is the name that will be set in the VEE Port configuration file (see *Installation*).
2. Implement one or several MicroUI drawing methods.
3. Call the corresponded engine's drawing method, for instance: LLVGEngine.Instance.getBufferedVectorImageDrawer().fillRectangle(xxx) .

Installation

Refer to the chapter *Installation* for more information about the front panel configuration.

Use

Launch a MicroVG application on the Simulator to run the Front Panel extension.

Release Notes

UI Pack Compatibility Version

The current VG Pack version is 1.8.1. The following table describes the compatibility ranges between VG and UI Packs.

VG Pack Range	UI Pack Range	Comment
1.8.1	[14.4.1-14.5.2]	Trace management
[1.7.0-1.7.2]	[14.1.1-14.5.2]	MicroUI Font extensibility
[1.5.0-1.6.0]	[14.0.0-15.0.0[	UI Pack major version
[1.3.0-1.4.2]	[13.5.0-14.0.0[	BufferedImage with custom format
[1.1.0-1.2.1]	[13.3.0-14.0.0[	Internal feature
[1.0.0-1.0.1]	[13.2.0-14.0.0[	

Foundation Libraries

The following table describes Foundation Libraries API versions implemented in MicroEJ VG Packs.

Table 86: MicroVG API Implementation

VG Pack Range	MicroVG
[1.7.0-1.8.1]	1.5.0
[1.4.0-1.6.0]	1.4.0
1.3.0	1.3.0
1.2.1	1.2.0
1.1.0	1.1.0
[1.0.0-1.0.1]	1.0.0

C Modules Compatibility Version

The C modules are described *here*.

Several generic C modules are available for a given version of the VG Pack. In addition to generic C modules, the specific implementations of the VG Pack over Vivante VGLite and Think Silicon NemaVG extend the main MicroVG C module. They override the default implementation to use a GPU to perform the drawings. Contrary to the main MicroVG C module, they are optional: when they are not available, the default implementation of drawings is used. The default implementation does nothing (no drawing) and throws the *drawing log* `DRAWING_LOG_NOT_IMPLEMENTED` .

These C Modules fetches automatically by transitivity the generic MicroUI and MicroVG C modules and the *Extended C Module* for the selected GPU.

The next tables summarizes the VG Packs with:

- the UI Pack (see upper),
- the specific C module that implements MicroVG over the GPU (called `VG-xxx` in next tables),
- the specific C module that implements MicroUI over the GPU (called `UI-xxx` in next tables), see *Extended C Modules*,

- the VG Pack C module,
- the UI Pack C module: see *UI Pack*,
- Freetype and HarfBuzz

> **Warning:** Compatible versions are more restrictive than for use of the UI pack (and its C modules) alone.

Vivante VGLite

The *VGLite C module* targets the NXP CPU that provides the Vivante VGLite accelerator.

The following table describes the version compatibility between the MicroEJ VG Packs, the UI Packs, the generic C modules and the VGLite C modules:

VG Pack	UI Pack	VG-VGLite	UI-VGLite	MicroVG C Module	MicroUI C Module	FreeType	Harf-Buzz	Comment
1.8.1	[14.5.1-14.5.2]	[0.0.1	11.0.0	8.0.1	[14.5.1-14.5.2]	5.0.1	4.0.0	New options convention
1.8.1	[14.4.1-14.4.2]	[0.0.1	10.0.0	8.0.1	[14.4.1-14.4.2]	5.0.1	4.0.0	Trace management
1.7.2	[14.5.1-14.5.2]	[1.0.2	11.0.0	7.0.2	[14.5.1-14.5.2]	[3.0.0-4.0.0[	[2.0.0-3.0.1[	New options convention
1.7.2	[14.1.1-14.4.2]	[9.0.2	10.0.0	7.0.2	[14.1.1-14.4.2]	[3.0.0-4.0.0[	[2.0.0-3.0.1[	Misc issues
1.7.1	[14.5.1-14.5.2]	[1.0.1	11.0.0	7.0.1	[14.5.1-14.5.2]	[3.0.0-4.0.0[	[2.0.0-3.0.1[	New options convention
1.7.1	[14.1.1-14.4.2]	[9.0.1	10.0.0	7.0.1	[14.1.1-14.4.2]	[3.0.0-4.0.0[	[2.0.0-3.0.1[	BVI issues
[1.5.0-1.7.0[	[14.5.1-14.5.2]	[1.0.0	11.0.0	7.0.0	[14.5.1-14.5.2]	[3.0.0-4.0.0[	[2.0.0-3.0.1[	New options convention
[1.5.0-1.7.0[	[14.1.1-14.4.2]	[9.0.0	10.0.0	7.0.0	[14.1.1-14.4.2]	[3.0.0-4.0.0[	[2.0.0-3.0.1[	MicroUI Font extensibility
1.6.0	[14.0.1-14.0.3]	[8.0.1	9.0.0	6.0.1	4.1.0	[3.0.0-4.0.0[	[2.0.0-3.0.1[	VG Pack extensibility
1.5.1	[14.0.0-14.0.3]	[7.0.1	8.0.1	5.0.0	4.0.1	2.0.2	1.0.2	Scissor issue
1.5.0	[14.0.1-14.0.3]	[7.0.1	8.0.1	5.0.0	4.0.1	2.0.2	1.0.2	C modules harmonization
1.5.0	14.0.0	7.0.0	8.0.0	5.0.0	4.0.0	2.0.2	1.0.2	UI Pack major version
1.4.2	[13.7.0-13.7.2]	[6.1.1	7.2.0	4.0.0	3.1.1	2.0.2	1.0.2	Very long paths
[1.4.0-1.4.1]	[13.7.0-13.7.2]	[6.0.0-6.1.0[	7.2.0	3.0.1	3.1.0	2.0.2	1.0.2	Free image resources
[1.4.0-1.4.1]	[13.5.1-13.6.2]	[6.1.0	7.1.0	3.0.1	3.0.0	2.0.2	1.0.2	VGLite *3.0.15_rev7*
1.4.0	[13.5.1-13.6.2]	[6.0.0	7.0.0	3.0.1	3.0.0	2.0.2	1.0.2	Blend modes
1.3.0	[13.5.1-13.6.2]	[5.0.1	6.0.1	3.0.0	3.0.0	2.0.2	1.0.2	Buffered vector image
1.2.1	[13.3.0-13.4.1]	[4.0.3	5.0.1	2.1.0	2.0.1	2.0.2	1.0.2	Image raw format
1.1.1	[13.3.0-13.4.1]	[3.0.2	3.0.0	2.0.0	2.0.0	2.0.2	1.0.2	

Think Silicon NemaVG

The *NemaVG C module* targets the CPU that provides the NemaVG accelerator.

The following table describes the version compatibility between the MicroEJ VG Packs and the C modules:

VG Pack	UI Pack	VG-NemaVG	UI-NemaGFX	MicroVG C Module	MicroUI C Module	FreeType	Harf-Buzz	Comment
1.8.1	[14.5.1-14.5.2]	[1.0.1	5.0.0	8.0.1	[14.5.1-14.5.2]	5.0.1	4.0.0	New options convention
1.8.1	[14.4.1-14.4.2]	[1.0.1	4.0.0	8.0.1	[14.4.1-14.4.2]	5.0.1	4.0.0	Trace management
1.7.2	[14.5.1-14.5.2]	[1.0.2	5.0.0	7.0.2	[14.5.1-14.5.2]	[3.0.0-4.0.0]	[2.0.0-3.0.1]	New options convention
1.7.2	[14.1.1-14.4.2]	[1.0.2	4.0.0	7.0.2	[14.1.1-14.4.2]	[3.0.0-4.0.0]	[2.0.0-3.0.1]	Misc issues
1.7.1	[14.5.1-14.5.2]	[1.0.1	5.0.0	7.0.1	[14.5.1-14.5.2]	[3.0.0-4.0.0]	[2.0.0-3.0.1]	New options convention
1.7.1	[14.1.1-14.4.2]	[1.0.1	4.0.0	7.0.1	[14.1.1-14.4.2]	[3.0.0-4.0.0]	[2.0.0-3.0.1]	BVI issues
[1.6.0-1.7.0]	[14.5.1-14.5.2]	[1.0.0	5.0.0	7.0.0	[14.5.1-14.5.2]	[3.0.0-4.0.0]	[2.0.0-3.0.1]	New options convention
[1.6.0-1.7.0]	[14.1.1-14.4.2]	[1.0.0	4.0.0	7.0.0	[14.1.1-14.4.2]	[3.0.0-4.0.0]	[2.0.0-3.0.1]	MicroUI Font extensibility
1.6.0	[14.0.1-14.0.3]	[1.0.1	3.0.0	6.0.1	4.1.0	[3.0.0-4.0.0]	[2.0.0-3.0.1]	

The following table describes the version compatibility between the C module and the NemaVG libraries (officially supported):

NemaVG C Module	NemaVG Libraries Range
[4.0.0-5.0.0]	[1.1.6-1.1.7]
[1.0.1-3.0.1]	1.1.6

FreeType

The *FreeType C module* provides a fork of FreeType.

The following table describes the version of FreeType included in the C module:

C Module Range	FreeType
5.0.1	2.14.1
4.0.0	2.13.3
3.0.0	2.11.0

HarfBuzz

The *HarfBuzz C module* provides HarfBuzz with additional source files.

The following table describes the version of HarfBuzz included in the C module:

C Module Range	HarfBuzz
4.0.0	12.3.0
[3.0.0-3.0.1]	10.0.1
2.0.0	4.2.1

Changelog

[1.8.1] - 2026-01-27

UI Pack

Changed

- Compatible with UI Pack 14.4.1 (Trace management).

Note: This pack fixes the memory leak that occurs when a feature is killed while a BufferedVectorImage, an external RAW image, or a filtered image is open. To avoid potential side effects from the release of objects (images, fonts) retained by a feature during the killing of that feature, ensure that no third-party components (e.g., GPU) are using these objects at the time of the kill. UI packs with versions higher than 14.5.1 provide the blocking API `LLUI_DISPLAY_waitAsynchronousDrawingEnd()`. This API is stubbed on the VG Pack for backward compatibility issues between VG Pack 1.8.1 and UI Packs [14.4.0, 14.5.1]. However, it is highly recommended to use a UI Pack version greater than 14.5.1.

MicroVG

Changed

- Harmonize the trace events (SystemView description file `SYSVIEW_MicroVG.txt` is updated).

Fixed

- Fix the memory leak when a feature is killed whereas a BufferedVectorImage, an external RAW image or a filtered image is opened.
- Fix the closing of a font that was opened twice, which caused both fonts (same object) to close simultaneously.
- Fix the clearing of a BufferedVectorImage when its clip is null: the content was not deleted at all.

Removed

- Remove the cache of the vector fonts: each call to `VectorFont.loadFont()` creates a new instance.

Front Panel

Changed

- Trace the drawing functions like the embedded side.

Fixed

- Fix the `nema` image decoder: fill rule was not correctly decoded.

C Module MicroVG

Added

- Add the implementation of the MicroVG native function LLVG_IMPL_initialize() ; the C Module *GPU* cannot implement this function anymore.

Changed

- Change the internal format (SNI context) of the Vector images (native methods are not backward compatible).
- Move the image trace events from native side to library side.
- Improve traces of drawing functions.
- Change the type of the parameter glyph_idx in the function VG_HELPER_layout_load_glyph() .
- Change the options configuration to follow the general rule of vg_conguration.h : default values are 0 (means "disabled"):
  - VG_FEATURE_FREETYPE_TTF
  - VG_FEATURE_FREETYPE_OTF
  - VG_FEATURE_FREETYPE_COLORED_EMOJI
  - VG_FEATURE_FONT_COMPLEX_LAYOUT
  - VG_FEATURE_FONT_EXTERNAL
  - VG_FEATURE_BUFFERED_VECTOR_IMAGE

Removed

- Remove the function VG_HELPER_initialize() (replaced by implementation of the function LLVG_IMPL_initialize()).
- Remove the weak implementation of the function VG_DRAWING_initialize() : the C Module *GPU* must initialize its components.
- Remove the traces "image" (moved in MicroVG library).

C Module VGLite

Changed

- Move the C Module initialization in the function VG_DRAWING_initialize() (specification of C Module MicroVG 8.0.1).

Fixed

- Fix the memory leak when a feature is killed whereas a BufferedVectorImage, an external RAW image or a filtered image is opened.

Removed

- Remove the file LLVG_impl_vglite.c .
- Remove the traces "image" (moved in MicroVG library).

C Module NemaVG

Changed

- Move the C Module initialization in the function `VG_DRAWING_initialize()` (specification of C Module MicroVG 8.0.1).

Fixed

- Fix the memory leak when a feature is killed whereas a BufferedVectorImage, an external RAW image or a filtered image is opened.

Removed

- Remove the file `LLVG_impl_vglite.c`.
- Remove the traces “image” (moved in MicroVG library).

C Module FreeType

Changed

- Upgrade FreeType from the version 2.13.3 (2024-08-11) to the version 2.14.1 (2025-09-11).
- Make the C Module compatible with the new option configuration of C Module MicroVG 8.0.1.

C Module HarfBuzz

Changed

- Upgrade HarfBuzz from the version 10.0.1 (2024-09-24) to the version 12.3.0 (2025-12-27).
- Make the C Module compatible with the new option configuration of C Module MicroVG 8.0.1.

[1.7.2] - 2025-09-15

MicroVG

Fixed

- Include `SYSVIEW_MicroVG.txt` in the library jar file (similarly to MicroUI library).

Front Panel

Fixed

- Fix the drawing of gradient in a BufferedVectorImage.

Vector Image Converter

Fixed

- Fix the generation of RAW vector images when building a feature.

C Module MicroVG

Fixed

- Fix the usage of LLTRACE_impl.h with the Architecture 8.x.
- Fix the default options in vg_conguration.h : some options are disabled by default.

C Module VGLite

Fixed

- Fix the initialization of the BufferedVectorImage support.
- Fix the drawing of a filtered image with opacity.

C Module NemaVG

Fixed

- Fix the initialization of the BufferedVectorImage support.
- Fix the invalid include directive to display.h (now requires the defines RESX and RESY).
- Fix the drawing of a filtered image with opacity.
- Fix the rendering of some texts on a BufferedVectorImage (invalid bounding box).

[1.7.1] - 2025-04-09

Front Panel

Fixed

- Fix the drawing bounds of the BufferedVectorImage's elements.
- Fix the drawing of a BufferedVectorImage outside the destination's clip.
- Fix the drawing of the BufferedVectorImage's elements which are outside the BufferedVectorImage's clip.
- Fix the drawing of gradient paths with zero or one color.
- Fix the management of transparent gradients for the NemaVG GPU.

C Module MicroVG

Fixed

- Fix the dependency to Freetype (MicroUI Font drawings with vector fonts).

C Module VGLite

Fixed

- Fix the drawing of a BufferedVectorImage when applying a rotation.
- Fix the drawing bounds of the BufferedVectorImage's elements.
- Fix the use of clip when drawing a BufferedVectorImage that contains vector images.
- Fix the clip when drawing a vector image in a BufferedVectorImage.
- Fix the drawing of gradient paths with zero or one color.

Removed

- Remove useless and invalid external field declaration.
- Remove dependency to a specific version of Freetype: the C Module Freetype must be fetched independently.

C Module NemaVG

Fixed

- Fix the drawing of a BufferedVectorImage when applying a rotation.
- Fix the drawing bounds of the BufferedVectorImage's elements.
- Fix the use of clip when drawing a BufferedVectorImagee that contains vector images.
- Fix the clip when drawing a vector image in a BufferedVectorImage.
- Fix the drawing of gradient paths with zero or one color.

[1.7.0] - 2024-10-09

UI Pack

Changed

- Compatible with UI Pack 14.1.1 (Font extensibility).

MicroVG

- Implement MicroVG API 1.5.

C Module FreeType

Changed

- Upgrade FreeType from the version 2.11.0 (2021-07-18) to the version 2.13.3 (2024-08-11).
- Separate the original FreeType files from MicroEJ's files.
- Use the original FreeType directory layout.
- Add a patch file to override FreeType's memory management without modifying the original file.

C Module HarfBuzz

Changed

- Upgrade HarfBuzz from the version 4.2.1 (2022-04-24) to the version 10.0.1 (2024-09-24).
- Separate the original HarfBuzz files from MicroEJ's files.

[1.6.0] - 2024-07-19

MicroVG

Added

- Allow the `nema` value for the MicroVG implementation (see *Installation*).

Changed

- Compatible with UI Pack 14.0.1.

Front Panel

Added

- Support `nema` image format and limitations (blend modes `SCREEN` and `MULTIPLY` are not supported).

LLAPIs

Added

- Add the LLAPI LLVG_MATRIX_IMPL_transformPoint .
- Add the new type MICROVG_Image .

Changed

- Change the semantic of the LLAPI LLVG_GRADIENT_IMPL_initializeGradient : the positions array cannot be NULL .
- Add const modifiers for parameters of the following functions:

– LLVG_GRADIENT_IMPL_initializeGradient ,

– LLVG_MATRIX_IMPL_copy ,

– LLVG_MATRIX_IMPL_multiply ,

– LLVG_MATRIX_IMPL_setConcat ,

– LLVG_MATRIX_IMPL_concatenate ,

– LLVG_MATRIX_IMPL_postConcat .

- Use the new type MICROVG_Image in the signature of the following functions:

– LLVG_BVI_IMPL_map_context ,

– LLVG_PAINTER_IMPL_drawImage .

[1.5.1] - 2024-04-11

Front Panel

Fixed

- Fix the drawing of a BufferedVectorImage in a BufferedVectorImage with a clip.
- Fix the drawing of a glyph with path overlap.

C Module VGLite

Fixed

- Fix the drawing of a VG RAW image in a BufferedVectorImage with a clip.

[1.5.0] - 2024-02-15

UI Pack

Changed

- Compatible with UI Pack 14.0.0 (Major version).

MicroVG

Fixed

- Fix the exception when loading a font or an image with an empty path.
- Fix the release of the `BufferedVectorImage` resources.

Front Panel

Fixed

- Fix the memory leak on images (ResourceVectorImage and BufferedVectorImage).

LLAPIs

Fixed

- Fix comment in header file `LLVG_BVI_impl.h`.

C Module MicroVG

Added

- Add the API `freeImageResources` that allows to fix the release of the `BufferedVectorImage` resources.

Fixed

- Fix traces when debugging the SNI resources with external resource support.
- Remove an unused include.
- Do not define Freetype variables if `VG_FEATURE_FONT` is not defined.
- Do not call `MICROVG_PATH_initialize()` if `VG_FEATURE_PATH` is not defined.

C Module VGLite

Fixed

- Fix the storing of color matrices in the `BufferedVectorImage`.

[1.4.2] - 2023-11-13

MicroVG

Added

- Add some traces when debugging the SNI resources.

Fixed

- Fix dynamic paths larger than 64 KB.

Front Panel

Fixed

- Fix dynamic paths larger than 64 KB.

C Module MicroVG

Added

- Add some traces when debugging the SNI resources (external VectorFont).

Fixed

- Fix dynamic paths larger than 64 KB.
- Fix some comments.

C Module VGLite

Fixed

- Fix some comments.
- Fix the dynamic path drawing on i.MX RT1170 Evaluation Kit (use the same quality of paths as vector images).
- Fix the path drawing on i.MX RT1170 Evaluation Kit (disable the color pre-multiplication).
- Fix the rendering of some blending modes on i.MX RT1170 Evaluation Kit by disabling the GPU pre-multiplication when required.

[1.4.1] - 2023-09-21

MicroVG

Fixed

- Fix the path command “move relative”.

C Module VGLite

Added

- Add the compatibility with VGLite $3.0.15_rev7$.

Fixed

- Fix the use of the define $VG_BLIT_WORKAROUND$ (useless).
- Fix the GPU deactivation when a drawing is not performed for any reason.

[1.4.0] - 2023-07-21

UI Pack

Fixed

- Fix the UI Pack minimal compatible version (13.5.0).

MicroVG

Added

- Add SystemView event traces (feature available with C Module MicroVG 3.0.1).

Changed

- Compatible with MicroVG API 1.4.

Fixed

- Fix path bounds computation.

C Module MicroVG

Fixed

- Fix the SystemView trace identifiers.
- Fix the documentation of `MICROVG_HELPER_get_utf()`.
- Fix FreeType fonts closing twice.

C Module VGLite

Added

- Add support for DST_OUT and PLUS blend modes (VG Pack 1.4.0).

Fixed

- Fix performing drawings when the clip is disabled.
- Fix the SystemView trace identifiers.
- Remove the include of the unknown header file `trace_vglite.h` (require a re-build of FreeType library).

[1.3.0] - 2023-05-10

UI Pack

Changed

- Compatible with UI Pack 13.5.0 (BufferedImage with custom format).

MicroVG

Changed

- Compatible with MicroVG API 1.3.

Front Panel

Fixed

- Simplify pixel data conversion after drawing.

C Module MicroVG

Added

- Add the compatibility with multiple Graphics Context output formats (UI Pack 13.5.0).
- Add stub implementations for all MicroVG library algorithms.
- Add LLVG_PAINTER_impl.c to implement all MicroVG drawings and dispatch them to vg_drawing.h (like MicroUI and LLUI_PAINTER_impl.c / ui_drawing.h).
- Add the MicroVG BufferedVectorImage definition (the functions to implement to draw into it).

Changed

- C Module MicroVG now depends on C Module MicroUI (to manage the support of multiple Graphics Context output formats).

Fixed

- Remove an extraneous file.
- Fix issue when measuring string width in complex layout mode.

Removed

- Remove the useless implementation of *LLVG_PATH_IMPL_mergePaths* (useless since VG Pack 1.2).
- Remove partial Freetype implementation that manipulates the font's glyphs as bitmaps (not compatible anymore with VG pack 1.3.0).

C Module VGLite

Added

- Add the implementation of all MicroUI, Drawing and MicroVG drawings in MicroVG BufferedVectorImage.
- Add incident reporting with drawing log flags (UI Pack 13.5.0).

Changed

- Merge BufferedVectorImage and RAW formats.
- Simplify the gradient modification according to the caller translation.

Fixed

- Fix the path to render during a *path data* animation.

Removed

- Remove LLVG_BVI_impl.c: code is merged in LLVG_RAW_impl.c.
- Remove (move) some utility functions to C Module MicroUI-VGLite.
- Remove *draw String* native functions implementation (implemented in C Module MicroVG).

[1.2.1] - 2023-02-06

Front Panel

Fixed

- Fix the cropped images when using GraphicsContext clip and translation.

C Module VGLite

Fixed

- Fix the drawing of RAW images with multiple gradients in BufferedVectorImage.
- Fix a deadlock when drawing an empty BufferedVectorImage.
- Fix the interface between FreeType and MicroVG (remove useless parameter).
- Fix the synchronization with the Graphics Engine when a VG drawing is not performed (draw path, draw gradient, draw string).

[1.2.0] - 2022-12-30

MicroVG

Changed

- Compatible with MicroVG API 1.2.
- Change the VectorImage internal format: *raw* format instead of *immutables* format.

Front Panel

Fixed

- Fix the redirection of fillEllipseArc to the right software algorithm.

Vector Image Converter

Added

- Add “fill alpha” animations to gradient elements.

C Module MicroVG

Added

- Add LLVG_MATRIX_IMPL_multiply(c,a,b) (C = AxB): faster than setConcat when destination and source target the same matrix.
- Add an entry point to initialize the path engine on startup.

Changed

- Prevent a copy in a temp matrix when calling postXXX functions.

Fixed

- Fix A.setConcat(B,A) .

C Module VGLite

Added

- Add the compatibility with VGLite 3.0.15_rev4 (not backward compatible).
- Add the VectorImage in binary format management (RAW format).
- Add loading of VectorImage from external resource system.

Changed

- Reduce the gradient footprint in BufferedVectorImage .
- Harmonize the use of vg_drawer.h functions (instead of VG_DRAWER_drawer_t functions) in BufferedVectorImage .
- Use the global fields *VGLite paths* instead of functions fields (prevent dynamic allocation on task stack).

Fixed

- Fix the drawing of a text in a BufferedVectorImage : do not wake-up the GPU.
- Fix the constants used in get_command_parameter_number() function (no side-effect).

[1.1.1] - 2022-09-05

UI Pack

Changed

- Compatible with UI Pack 13.3.0 (Internal feature).

MicroVG

Changed

- Compatible with MicroVG API 1.1.
- Change color animation interpolation (match Android formula).

Fixed

- Fix NullPointerException while sorting TranslateXY VectorDrawableObjectAnimator in vectorimage-converter.

LLAPIs

Added

- Add LLAPI to close a font: `LLVG_FONT_IMPL_dispose()`.

Changed

- Manage the font *complex layout*.
- Returns an error code when drawing something.

C Module MicroVG

Added

- Add `microvg_configuration.h` versioning.
- Add an option to load a VectorFont from the external resources.
- Add an option to select the text layouter between FreeType and Harfbuzz.
- Add a function to apply an opacity on a color.
- Add the text layout.

Changed

- Configure FreeType from `microvg_configuration.h` header file.

C Module VGLite

Added

- Add the BufferedVectorImage feature (BVI).

Changed

- Manage the closed fonts.
- Move ftvglite.c and ftvglite.h to C Module FreeType.
- Extract text layout to C Module MicroVG.
- Get fill rule configuration from each glyph FT_Outline->flags instead of defaulting it to VG_LITE_FILL_EVEN_ODD.
- Use the MicroUI over VGLite's Vectorial Drawer mechanism.
- Join character bboxes at baseline for drawStringOnCircle.

[1.0.1] - 2022-05-16

MicroVG

Fixed

- Fix incorrect transformation of animated paths while creating a filtered image.

[1.0.0] - 2022-05-13

- Initial release.

UI Pack

- Compatible with UI Pack 13.2.0 or higher.

MicroVG

- Compatible with MicroVG API 1.0.0.

Migration Guide

From 1.7.2 to 1.8.1

VEE Port Configuration project

- Update the UI Pack to version 14.4.1 or higher.

Application

The font cache has been removed. Each call to `VectorFont.loadFont()` creates a new instance of the font. Consequently, a font should not be loaded during rendering but rather in the initialization phase (for example, in the *attach* method of a widget). It is also essential to close the font when it is no longer in use (for example, in the *detach* method of a widget).

BSP

The C Module's options configuration file `vg_configuration.h` must not be modified anymore. Before installing the new C Module:

- Create (if it doesn't exist already) the global VEE Port configuration file.
- Move your own configuration from `vg_configuration.h` to `veeport_configuration.h`.
- Install the C Module MicroVG 8.0.1.

The following options (see `vg_configuration.h`) have now a value (not just commented/uncommented); check (and adjust) whether they are used outside the VG C Modules:

- `VG_FEATURE_FREETYPE_TTF`
- `VG_FEATURE_FREETYPE_OTF`
- `VG_FEATURE_FREETYPE_COLORED_EMOJI`
- `VG_FEATURE_FONT_COMPLEX_LAYOUT`
- `VG_FEATURE_FONT_EXTERNAL`
- `VG_FEATURE_BUFFERED_VECTOR_IMAGE`

BSP with VGLite

- **Prerequisite:** follow the migration steps of *BSP*.
- Delete the file `LLVG_impl_vglite.c`
- Install the C Module MicroVG over VGLite 10.0.1.

BSP with NemaVG

- **Prerequisite:** follow the migration steps of *BSP*.
- Delete the file `LLVG_impl_nema.c`
- Install the C Module MicroVG over NemaVG 3.0.1.

FreeType

- **Prerequisite:** follow the migration steps of *BSP*.
- Install the C Module Freetype 5.0.1.
- In case of standalone library: include the folder of the configuration file `vg_configuration.h` in the include directives and rebuild the library.

Note: The update of FreeType (from the version 2.13.3 to the version 2.14.1) is optional. In that case, update the files `ft_ttf_wrapper.c`, `ft_otf_wrapper.c` and `ftmodule.h`:

- Replace `#ifdef VG_FEATURE_FREETYPE_TTF` by `#if defined VG_FEATURE_FREETYPE_TTF && (VG_FEATURE_FREETYPE_TTF == 1)`
- Replace `#ifdef VG_FEATURE_FREETYPE_OTF` by `#if defined VG_FEATURE_FREETYPE_OTF && (VG_FEATURE_FREETYPE_OTF == 1)`

HarfBuzz

- **Prerequisite:** follow the migration steps of *BSP*.
- Install the C Module Harfbuzz 4.0.0.
- Rebuild the library.

Note: The update of HarfBuzz (from the version 10.0.1 to the version 12.3.0) is optional. In that case, update the file `hb-alloc.c`:

- Replace `#if defined(VG_FEATURE_FONT_COMPLEX_LAYOUT)` by `#if defined VG_FEATURE_FONT_COMPLEX_LAYOUT && (VG_FEATURE_FONT_COMPLEX_LAYOUT == 1)`

From 1.7.1 to 1.7.2

BSP with VGLite

- Install the C Module MicroVG 7.0.2.
- Install the C Module MicroVG over VGLite 9.0.2.

BSP with NemaVG

- Install the C Module MicroVG 7.0.2.
- Install the C Module MicroVG over NemaVG 2.0.2.
- Add the defines `RESX` and `RESY` in `vg/inc/vg_configuration.h` or in the C project global options.

From 1.7.0 to 1.7.1

BSP with VGLite

- Install the C Module MicroVG over VGLite 9.0.1.

BSP with NemaVG

- Install the C Module MicroVG over NemaVG 2.0.1.

From 1.6.0 to 1.7.0

VEE Port Configuration project

- Update the UI Pack to version 14.1.1 or higher.

BSP with VGLite

- Follow the migration steps of *BSP with VGLite* for the new UI Pack.
- Install the C Module MicroVG over VGLite 9.0.0.
- Set the define `UI_GC_SUPPORTED_FORMATS` to `2` or more in `ui_configuration.h`.
- Set the define `UI_FEATURE_FONT_CUSTOM_FORMATS` in `ui_configuration.h`.

BSP with NemaVG

- Follow the migration steps of *BSP with NemaGFX* for the new UI Pack.
- Install the C Module MicroVG over NemaVG 2.0.0.
- Set the define `UI_GC_SUPPORTED_FORMATS` to `2` or more in `ui_configuration.h`.
- Set the define `UI_FEATURE_FONT_CUSTOM_FORMATS` in `ui_configuration.h`.

FreeType

Optionally, update the *C Module FreeType* to use FreeType version 2.13.3:

- Delete the directory `thirdparty/freetype`.
- Remove `thirdparty/freetype/inc` and `thirdparty/freetype/inc/ftvector` from your include path.
- Add `thirdparty/freetype/include`, `thirdparty/freetype/src` and `freetype_support/src` to your include path.
- Remove `thirdparty/freetype/lib/freetype.a` from your build path.
- Build FreeType and add the library and support files following the C Module FreeType's instructions.

HarfBuzz

Optionally, update the *C Module HarfBuzz* to use HarfBuzz version 10.0.1:

- Delete the directory `thirdparty/harfbuzz`.
- Remove `thirdparty/harfbuzz/inc` from your include path.
- Add `thirdparty/harfbuzz/src` to your include path.
- Remove `thirdparty/harfbuzz/lib/harfbuzz.a` and `thirdparty/harfbuzz/src/hb-alloc.c` from your build path.
- Build HarfBuzz and add the library and support files following C Module HarfBuzz's instructions.

From 1.5.x to 1.6.0

VEE Port Configuration project

- Update the UI Pack to version 14.0.1 or higher.
- Specify the `implementation` property, as written in the section *Installation*.

BSP

- Update your implementation of the following functions to match the signature changes.

– `LLVG_GRADIENT_IMPL_initializeGradient`,
– `LLVG_MATRIX_IMPL_copy`,
– `LLVG_MATRIX_IMPL_multiply`,
– `LLVG_MATRIX_IMPL_setConcat`,
– `LLVG_MATRIX_IMPL_concatenate`,
– `LLVG_MATRIX_IMPL_postConcat`,
– `LLVG_BVI_IMPL_map_context`,
– `LLVG_PAINTER_IMPL_drawImage`.

BSP with VGLite

- Follow the migration steps of *BSP with VGLite* for the new UI Pack.
- Install the C Module MicroVG over VGLite 8.0.1.
- Delete the following file in the `ui` directory:

– `src/ui_drawing_bvi.c`.

- Delete the following files in the `vg` directory:

– `inc/microvg_font_freetype.h`,
– `inc/microvg_gradient.h`,
– `inc/microvg_helper.h`,

- inc/microvg_path.h ,
- inc/microvg_trace.h ,
- src/LLVG_BVI_stub.c ,
- src/LLVG_FONT_freetype.c ,
- src/LLVG_FONT_stub.c ,
- src/LLVG_GRADIENT_impl.c ,
- src/LLVG_impl.c ,
- src/LLVG_PATH_impl.c ,
- src/LLVG_PATH_stub.c ,
- src/microvg_helper.c ,
- inc/vg_bvi_vglite.h ,
- inc/vg_drawing_vglite.h ,
- inc/vg_vglite_helper.h ,
- src/LLVG_GRADIENT_impl_vglite.c ,
- src/LLVG_impl_vglite.c ,
- src/vg_drawing_bvi.c ,
- src/vg_drawing_vglite_image.c ,
- src/vg_drawing_vglite.c ,
- src/vg_path_vglite.c ,
- src/vg_vglite_helper.c .
    - Delete the properties files cco_microvg.properties and cco_microvg-vglite.properties .
    - Build the VEE Port.
    - Configure the C Module MicroVG in vg/inc/vg_configuration.h , based on your previous settings in vg/inc/microvg_configuration.h .
    - Delete vg/inc/microvg_configuration.h .
    - Add the source files in vg/src and vg_vglite/src to the project.
    - Add the path vg_vglite/inc to the include path.

From 1.4.x to 1.5.1

VEE Port Configuration Project

- Update UI Pack version: 14.0.0 or higher.

BSP with VGLite

- Follow the migration steps of *BSP with VGLite* for the new UI Pack.
- Install the C Modules MicroVG 5.0.0 and MicroVG-VGLite 7.0.1.

From 1.3.x to 1.4.2

BSP with VGLite

- Follow the migration steps of *BSP with VGLite* for the new UI Pack.
- Install the C Modules MicroVG 4.0.0 and MicroVG-VGLite 6.1.1.
- Delete the content of `dropins/include` folder.
- Build the FreeType library (in case of a dedicated project), and then the BSP.

From 1.2.x to 1.3.0

VEE Port Configuration Project

- Update UI Pack version: 13.5.0 or higher.

BSP with VGLite

- Follow the migration steps *BSP with VGLite* for the new UI Pack.
- Install the C Modules MicroVG 3.0.0 and MicroVG-VGLite 5.0.1.
- Delete the content of `dropins/include` folder.
- Delete the C files `freetype_bitmap_helper.h`, `freetype_bitmap_helper.c`, `LLVG_BVI_impl.c`, `LLVG_FONT_PAINTER_freetype_bitmap.c` and `LLVG_PATH_PAINTER_vglite.c` and remove them from the C project configuration.
- In the C project configuration, include the new C files `ui_drawing_bvi.c`, `LLVG_BVI_stub.c`, `LLVG_PAINTER_impl.c`, `vg_drawing_bvi.c`, `vg_drawing_stub.c`, `vg_drawing_vglite.c` and `vg_drawing.c`.
- In the C project configuration, set the define `LLUI_GC_SUPPORTED_FORMATS=2` to enable the BufferedVectorImage support.
- Verify the options in `microvg_configuration.h`.
- Build the FreeType library (in case of a dedicated project), and then the BSP.

3.6.17 Networking

Principle

MicroEJ provides some Foundation Libraries to initiate raw TCP/IP protocol-oriented communications and secure this communication by using Secure Socket Layer (SSL) or Transport Layer Security (TLS) cryptographic protocols.

The diagram below shows a simplified view of the components involved in the provisioning of a Java network interface.

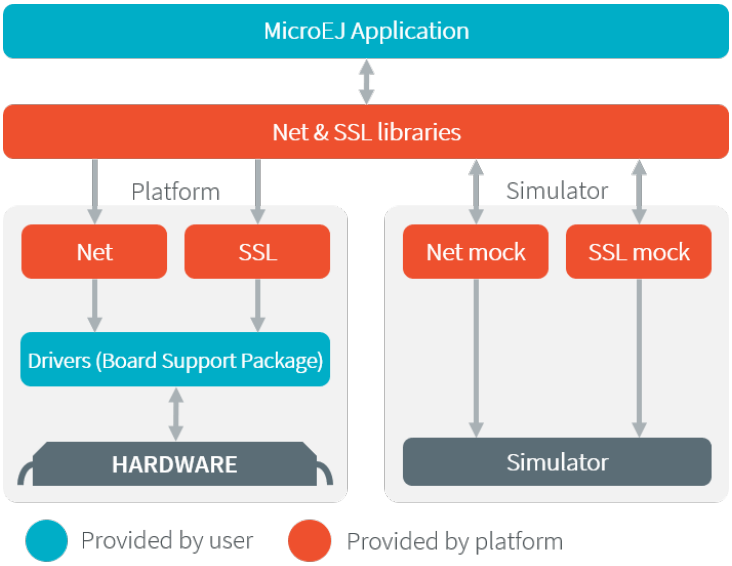


Fig. 248: Overview

Net and SSL low level parts connects the Net and SSL libraries to the user-supplied drivers code (coded in C).

The MicroEJ Simulator provides all features of Net and SSL libraries. This one takes part of the network settings stored in the operating system on which the Simulator will be launched.

Network Core Engine

Principle

The Net module defines a low-level network framework for embedded devices. This module allows you to manage connection (TCP)- or connectionless (UDP)-oriented protocols for client/server networking applications.

Functional Description

The Net library includes two sub-protocols:

- UDP: a connectionless-oriented protocol that allows communication with the server or client side in a non-reliable way. No handshake mechanisms, no guarantee on delivery, and no order in packet sending.
- TCP: a connection-oriented protocol that allows communication with the server or client side in a reliable way. Handshakes mechanism used, bytes ordered, and error checking performed upon delivery.

Dependencies

- LLNET_CHANNEL_impl.h , LLNET_SOCKETCHANNEL_impl.h , LLNET_STREAMSOCKETCHANNEL_impl.h , LLNET_DATAGRAMSOCKETCHANNEL_impl.h , LLNET_DNS_impl.h , LLNET_NETWORKADDRESS_impl.h , LLNET_NETWORKINTERFACE_impl.h (see *LLNET: Network*).

Installation

NET is an additional module. To enable it, the Net *Pack* (which bundles several libraries: Net, SSL & Security) must be installed in your VEE Port:

SDK 6 (build.gradle.kts)

SDK 5 (module.ivy)

```
microejPack("com.microej.pack.net:net-pack:11.0.2")
```

```
<dependency org="com.microej.pack.net" name="net-pack" rev="11.0.2"/>
```

Then, using the VEE Port Editor (see *Platform Module Configuration*), enable the Net library (API, Impl & Mock):

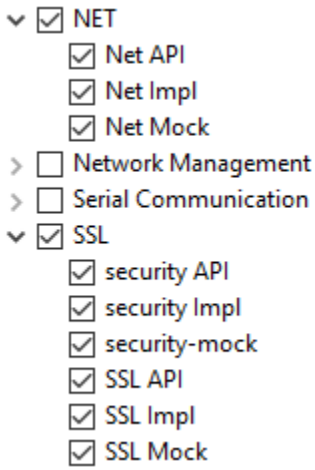


Fig. 249: Net Pack Modules

Initialization

When porting the Net library the initialize function shall make sure the underlying network stack is initialized. The entry point for this initialization is the following native function: LLNET_CHANNEL_impl_initialize:

```
/**
 * @brief Initializes the TCP/IP stack components.
 *
 * @note Throws NativeIOException on error.
 *
 * @see LLNET_ERRORS.h header file for error codes.
 */
void LLNET_CHANNEL_IMPL_initialize(void);
```

It is called during the initialization phase of the Net library so it will run before the application starts.

In most of the VEE Ports a macro llnet_init() is provided to call any specific code required before using the network:

```
// ...

#include "LLNET_configuration.h" // Macro is defined in this header.
#include "LLNET_CHANNEL_impl.h"

void LLNET_CHANNEL_IMPL_initialize(void) {
  llnet_init(); // Custom initialization here.
  // ...
}

// ...
```

An example of how the network is initialized with the same requirements can be found here in the NXP VEE Port for i.MX RT1170 EVK.

Use

The Net API Module must be added to the Application project build file to use the NET library:

SDK 6 (build.gradle.kts)

SDK 5 (module.ivy)

```
implementation("ej.api:net:1.1.4")
```

```
<dependency org="ej.api" name="net" rev="1.1.4"/>
```

This library provides a set of options. Refer to the chapter *Standalone Application Options* which lists all available options.

SSL

Principle

SSL (Secure Sockets Layer) library provides APIs to create and establish an encrypted connection between a server and a client. It implements the standard SSL/TLS (Transport Layer Security) protocol that manages client or server authentication and encrypted communication. Mutual authentication is supported since SSL API 2.1.0 .

Functional Description

The SSL/TLS process includes two sub-protocols :

- Handshake protocol: consists that a server presents its digital certificate to the client to authenticate the server's identity. The authentication process uses public-key encryption to validate the digital certificate and confirm that a server is in fact the server it claims to be.
- Record protocol: after the server authentication, the client and the server establish cipher settings to encrypt the information they exchange. This provides data confidentiality and integrity.

Dependencies

- Network core module (see *Network Core Engine*).
- LLNET_SSL_CONTEXT_impl.h and LLNET_SSL_SOCKET_impl.h implementations (see *LLNET_SSL: SSL*).

Installation

SSL is an additional module. To enable it, the Net *Pack* (which bundles several libraries: Net, SSL & Security) must be installed in your VEE Port:

SDK 6 (build.gradle.kts)

SDK 5 (module.ivy)

```
microejPack("com.microej.pack.net:net-pack:11.0.2")
```

```
<dependency org="com.microej.pack.net" name="net-pack" rev="11.0.2"/>
```

Then, using the VEE Port Editor (see *Platform Module Configuration*), enable the SSL library (API, Impl & Mock):

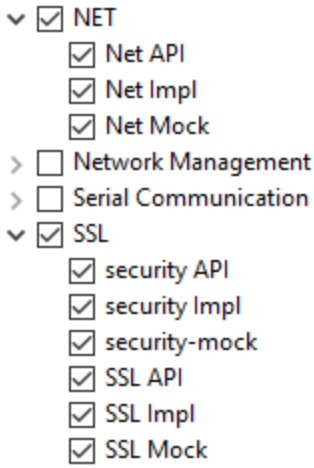


Fig. 250: Net Pack Modules

Use

The SSL API module must be added to the Application project build file to use the SSL library:

SDK 6 (build.gradle.kts)

SDK 5 (module.ivy)

```
implementation("ej.api:ssl:2.2.3")
```

```
<dependency org="ej.api" name="ssl" rev="2.2.3"/>
```

Network Interfaces Management

Overview

The Network Foundation Library provides a way to manage and configure TCP/IP network interfaces.

Dependencies

- Network core module (see *Network Core Engine*).
- LLECOM_NETWORK_impl.h implementation (see *LLECOM_NETWORK: Network Interfaces*).

Installation

The Network *Pack* module must be installed in your VEE Port.

SDK 6 (build.gradle.kts)

SDK 5 (module.ivy)

```
microejPack("com.microej.pack.ecom-network:ecom-network-pack:1.0.0")
```

```
<dependency␣
↪org="com.microej.pack.ecom-network" name="ecom-network-pack" rev="1.0.0" />
```

Then the VEE Port project must be rebuilt (*VEE Port Build*).

Use

The Network API Module module must be added to the Application project build file to use the Network library:

SDK 6 (build.gradle.kts)

SDK 5 (module.ivy)

```
implementation("ej.api:ecom-network:2.1.1")
```

```
<dependency org="ej.api" name="ecom-network" rev="2.1.1"/>
```

Wi-Fi

Overview

The Wi-Fi Foundation Library provides a way to manage and configure Wi-Fi access points.

Dependencies

- Network core module (see *Network Core Engine*).
- LLECOM_WIFI_impl.h implementation (see *LLECOM_WIFI: Wi-Fi Management*).

Installation

Wi-Fi is an additional module. To enable it, the Wi-Fi *Pack* module must be installed in your VEE Port:

SDK 6 (build.gradle.kts)

SDK 5 (module.ivy)

```
microejPack("com.microej.pack.ecom-wifi:ecom-wifi-pack:1.0.0")
```

```
<dependency org="com.microej.pack.ecom-wifi" name="ecom-wifi-pack" rev="1.0.0" />
```

Then the VEE Port project must be rebuilt (*VEE Port Build*).

Use

The Wi-Fi API Module must be added to the Application project build file to use the NET library:

SDK 6 (build.gradle.kts)

SDK 5 (module.ivy)

```
implementation("ej.api:ecom-wifi:2.2.2")
```

```
<dependency org="ej.api" name="ecom-wifi" rev="2.2.2"/>
```

3.6.18 Bluetooth

Principle

The Bluetooth Foundation Library defines a low-level Bluetooth framework for embedded devices. It allows you to manage abstract Bluetooth connections without worrying about the native underlying Bluetooth kind.

Functional Description

The MicroEJ Application manages Bluetooth elements using Adapter/Connection/Service/Characteristic/Descriptor/etc abstraction. The Bluetooth implementation made for each MicroEJ Platform is responsible for surfacing the native Bluetooth specific behavior.

Dependencies

- LLBLUETOOTH_impl.h implementation (see *LLBLUETOOTH: Bluetooth*).
  - A sample implementation based on the Bluedroid stack can be found in the Espressif ESP32-S3 VEE Port.

Installation

Bluetooth is an additional module. To enable it, the Bluetooth *Pack* module must be installed in your VEE Port:

SDK 6 (build.gradle.kts)

SDK 5 (module.ivy)

```
microejPack("com.microej.pack.bluetooth:bluetooth-pack:2.4.1")
```

```
<dependency org="com.microej.pack.bluetooth" name="bluetooth-pack" rev="2.4.1" />
```

Use

See *Bluetooth API* chapter in Application Developer Guide.

3.6.19 Audio

Principle

The Audio Foundation Library defines a low-level Audio framework for embedded devices. It allows to manage abstract audio streams without worrying about the native underlying Audio device kind.

Functional Description

The MicroEJ Application manages Audio devices using audio streams abstraction. The Audio implementation made for each VEE Port is responsible for surfacing the native Audio device specific behavior.

Dependencies

- LLAUDIO_RECORD_impl.h and LLAUDIO_TRACK_impl.h implementations (see *LLAUDIO: Audio*).

Installation

Audio is an additional module. To enable it, the Audio *Pack* module must be installed in your VEE Port:

SDK 6 (build.gradle.kts)

SDK 5 (module.ivy)

```
microejPack("com.microej.pack.audio:audio-pack:1.0.0")
```

```
<dependency org="com.microej.pack.audio" name="audio-pack" rev="1.0.0"/>
```

Use

See *Audio API* chapter in Application Developer Guide.

3.6.20 MicroAI

Principle

The MicroAI Foundation Library defines a low-level Machine Learning framework for embedded devices. It allows to run inferences on trained Machine Learning models.

Functional Description

A typical Application using MicroAI will load a model binary file, read its input/output characteristics, and finally perform an inference.

The MicroAI integration in a VEE port relies on a native AI framework (Tensorflow-lite, ONNX Runtime, etc…) to implement all of the above functionalities.

Dependencies

- LLML_impl.h implementation (see *LLML: MicroAI*).
    - A port of MicroAI for TensorFlow Lite can be found in MicroAI Abstraction Layer for TensorFlow Lite.

Installation

MicroAI is an additional module. To enable it, the MicroAI *Pack* must be installed in your VEE Port:

```
microejPack("ej.api:microai:2.1.0")
```

Use

See *MicroAI API* chapter in Application Developer Guide.

3.6.21 Event Queue

Principle

The Event Queue Foundation Library provides an asynchronous communication interface between the native world and the Managed world based on events. Its functional architecture and usage are documented in the *Application Developer Guide*.

Dependencies

- LLEVENT_impl.h and LLEVENT.h implementations (see *LLEVENT: Event Queue*).

Installation

The Event Queue *Pack* module must be installed in your VEE Port:

SDK 6 (build.gradle.kts)

SDK 5 (module.ivy)

For Event Queue 2.3.0 and higher:

```
microejPack("ej.api:event:2.3.0")
```

For Event Queue 2.2.0 and lower:

```
microejPack(group=
↪"com.microej.pack.event", name="event-pack", version="2.2.0", configuration="mockAPI")
```

For Event Queue 2.3.0 and higher:

```
Not Supported
```

For Event Queue 2.2.0 and lower:

```
<dependency␣
↪org="com.microej.pack.event" name="event-pack" rev="2.2.0" transitive="false"/>
```

3.6.22 File System

Principle

The FS Foundation Library defines a low-level File System framework for embedded devices. It allows you to manage abstract files and directories without worrying about the native underlying File System kind.

Functional Description

The MicroEJ Application manages File System elements using File/Directory abstraction. The FS implementation made for each MicroEJ Platform is responsible for surfacing the native File System specific behavior.

Dependencies

- LLFS_impl.h and LLFS_File_impl.h implementations (see *LLFS: File System*).

Installation

FS is an additional module. To enable it, the FS *Pack* module must be installed in your VEE Port:

SDK 6 (build.gradle.kts)

SDK 5 (module.ivy)

```
microejPack("com.microej.pack:fs:6.0.4")
```

```
<dependency org="com.microej.pack" name="fs" rev="6.0.4"/>

Then, using␣
↪the VEE Port Editor (see :ref:`platform__module__configuration`), enable the `FS` library.
```

When installed, the FS Pack module must be configured.

The FS module defines two pre-configured File System types: Unix and FatFS. Some characteristics don't need to be specified for these File System types, but they can be overridden if needed. For example, specifying a Unix File System type will automatically set the file separator to /.

If none of the pre-configured File System types correspond to the File System used in the C project, the Custom type can be used. When this type is selected, all the File System characteristics must be specified in the properties file.

SDK 6

SDK 5

In SDK 6, the configuration is done in the properties file configuration.properties of the VEE Port project. All the properties names listed below must be prefixed by com.microej.pack.fs.. For example the fs properties is defined by the com.microej.pack.fs.fs property.

In SDK 5, the configuration is done in the properties file fs/fs.properties. This properties file specifies the characteristics of the File System used in the C project (case sensitivity, root directory, file separator, etc.).

The list below describes the properties that can be defined in the configuration file:

- fs: Defines the type of File System used in the C project (optional, the default value is Unix). This property can have one of the following values:
  - Unix: select this configuration when using a Unix-like File System (case-sensitive, file separator is /).
  - FatFS: select this configuration when using FatFS File System (case-insensitive, file separator is /).
  - Custom: select this configuration when using another type of File System.
- root.dir: Defines the File System root volume. This property is optional for Unix and FatFS (/ by default for both).
- user.dir: Defines the File System user directory. This property is optional for FatFS (/usr/ by default).
- java.io.tmpdir: Defines the File System temporary directory. This property is optional for Unix and FatFS (/tmp/ by default for both).
- file.separator: Defines the File System file separator. This property is optional for Unix and FatFS (/ by default for both).
- path.separator: Defines the File System path separator. This property is optional for Unix and FatFS (: by default for both).
- case.sensitivity: Defines the case sensitivity of the File System. This property is optional for Unix (caseSensitive by default) and FatFS (caseInsensitive by default). This property can have one of the following values:
  - caseSensitive: the File System is case-sensitive.
  - caseInsensitive: the File System is case-insensitive.

Properties File Template

SDK 6

SDK 5

The following snippet can be used as a template for `configuration.properties` file for the File System configuration properties:

```
# Defines the type of File System used in the C project.
# Possible values are:
#   - FatFs
#   - Unix
#   - Custom
# @optional, default value is "Unix"
#com.microej.pack.fs.fs=

# Defines the File System root volume.
# @optional for the following File System types:
#   - FatFs (default value is "/")
#   - Unix (default value is "/")
# @mandatory for the following File System type:
#   - Custom
#com.microej.pack.fs.root.dir=

# Defines the File System user directory.
# @optional for the following File System type:
#   - FatFs (default value is "/usr")
# @mandatory for the following File System types:
#   - Unix
#   - Custom
#com.microej.pack.fs.user.dir=

# Defines the File System temporary directory.
# @optional for the following File System types:
#   - FatFs (default value is "/tmp")
#   - Unix (default value is "/tmp")
# @mandatory for the following File System type:
#   - Custom
#com.microej.pack.fs.java.io.tmpdir=

# Defines the File System file separator.
# @optional for the following File System types:
#   - FatFs (default value is "/")
#   - Unix (default value is "/")
# @mandatory for the following File System type:
#   - Custom
#com.microej.pack.fs.file.separator=

# Defines the File System path separator.
# @optional for the following File System types:
#   - FatFs (default value is ":")
#   - Unix (default value is ":")
# @mandatory for the following File System type:
#   - Custom
#com.microej.pack.fs.path.separator=

# Defines the case sensitivity of the File System.
```

(continues on next page)

(continued from previous page)

```
# Valid values are ”caseInsensitive” and ”caseSensitive”.
# @optional for the following File System types:
#   - FatFs (default value is ”caseInsensitive”)
#   - Unix (default value is ”caseSensitive”)
# @mandatory for the following File System type:
#   - Custom
#com.microej.pack.fs.case.sensitivity=
```

The following snippet can be used as a template for fs.properties file:

```
# Defines the type of File System used in the C project.
# Possible values are:
#   - FatFs
#   - Unix
#   - Custom
# @optional, default value is ”Unix”
#fs=

# Defines the File System root volume.
# @optional for the following File System types:
#   - FatFs (default value is ”/”)
#   - Unix (default value is ”/”)
# @mandatory for the following File System type:
#   - Custom
#root.dir=

# Defines the File System user directory.
# @optional for the following File System type:
#   - FatFs (default value is ”/usr”)
# @mandatory for the following File System types:
#   - Unix
#   - Custom
#user.dir=

# Defines the File System temporary directory.
# @optional for the following File System types:
#   - FatFs (default value is ”/tmp”)
#   - Unix (default value is ”/tmp”)
# @mandatory for the following File System type:
#   - Custom
#java.io.tmpdir=

# Defines the File System file separator.
# @optional for the following File System types:
#   - FatFs (default value is ”/”)
#   - Unix (default value is ”/”)
# @mandatory for the following File System type:
#   - Custom
#file.separator=

# Defines the File System path separator.
# @optional for the following File System types:
#   - FatFs (default value is ”:”)
#   - Unix (default value is ”:”)
# @mandatory for the following File System type:
#   - Custom
```

(continues on next page)

(continued from previous page)

```
#path.separator=

# Defines the case sensitivity of the File System.
# Valid values are "caseInsensitive" and "caseSensitive".
# @optional for the following File System types:
#   - FatFs (default value is "caseInsensitive")
#   - Unix (default value is "caseSensitive")
# @mandatory for the following File System type:
#   - Custom
#case.sensitivity=
```

Use

The FS API Module must be added to the project build file to use the FS library:

SDK 6 (build.gradle.kts)

```
implementation("ej.api:fs:2.1.1")
```

SDK 5 (module.ivy)

```
<dependency org="ej.api" name="fs" rev="2.1.1"/>
```

3.6.23 GNSS

Principle

The GNSS module defines a low-level GNSS framework for embedded devices. It allows you to manage the underlying GNSS engine and retrieve data such as the longitude, latitude, velocity and time.

Functional Description

The MicroEJ Application manages the GNSS engine using GnssManager abstraction. The GNSS implementation made for each MicroEJ VEE Port is responsible for surfacing the native GNSS specific behavior.

GNSS Mock

GNSS values on simulator can be controlled through an UI that can be activated and deactivated with the property : com.is2t.gnss.embedded.mock.ui .

Dependencies

- LLGNSS_impl.h implementations (see *LLGNSS: GNSS (Global Navigation Satellite System)*).

Installation

GNSS is an additional module. The GNSS *Pack* module must be installed in your VEE Port:

```
microejPack("com.microej.pack.gnss:gnss-pack:2.0.0")
```

Use

See *GNSS API* chapter in Application Developer Guide.

3.6.24 Hardware Abstraction Layer

Principle

The Hardware Abstraction Layer (HAL) Foundation Library features API that target IO devices, such as GPIOs, analog to/from digital converters (ADC / DAC), etc. The API are very basic in order to be as similar as possible to the BSP drivers.

Functional Description

The MicroEJ Application configures and uses some physical GPIOs, using one unique identifier per GPIO. The HAL implementation made for each MicroEJ Platform has the responsibility of verifying the veracity of the GPIO identifier and the valid GPIO configuration.

Theoretically, a GPIO can be reconfigured at any time. For example a GPIO is configured in OUTPUT first, and later in ADC entry. However the HAL implementation can forbid the MicroEJ Application from performing this kind of operation.

Identifier

Basic Rule

MicroEJ Application manipulates anonymous identifiers used to identify a specific GPIO (port and pin). The identifiers are fixed by the HAL implementation made for each MicroEJ Platform, and so this implementation is able to make the link between the MicroEJ Application identifiers and the physical GPIOs.

- A port is a value between 0 and n - 1, where n is the available number of ports.
- A pin is a value between 0 and m - 1, where m is the maximum number of pins per port.

Generic Rules

Most of time the basic implementation makes the link between the port / pin and the physical GPIO following these rules:

- The port 0 targets all MCU pins. The first pin of the first MCU port has the ID 0, the second pin has 1; the first pin of the next MCU port has the ID m (where m is the maximum number of pins per port), etc. Examples:

```
/* m = 16 (16 pins max per MCU port) */
mcu_pin = application_pin & 0xf;
mcu_port = (application_pin >> 4) + 1;
```

```
/* m = 32 (32 pins max per MCU port) */
mcu_pin = application_pin & 0x1f;
mcu_port = (application_pin >> 5) + 1;
```

- The port from 1 to n (where n is the available number of MCU ports) targets the MCU ports. The first MCU port has the ID 1, the second has the ID 2, and the last port has the ID n.
- The pin from 0 to m - 1 (where m is the maximum number of pins per port) targets the port pins. The first port pin has the ID 0, the second has the ID 1, and the last pin has the ID m - 1.

  The implementation can also normalize virtual and physical board connectors. A physical connector is a connector available on the board, and which groups several GPIOs. The physical connector is usually called JPn or CNn, where n is the connector ID. A virtual connector represents one or several physical connectors, and has a *name*; for example ARDUINO_DIGITAL.

  Using a unique ID to target a virtual connector allows you to make an abstraction between the MicroEJ Application and the HAL implementation. For exmaple, on a board A, the pin D5 of ARDUINO_DIGITAL port will be connected to the MCU portA, pin12 (GPIO ID = 1, 12). And on board B, it will be connected to the MCU port5, pin0 (GPIO ID = 5, 0). From the MicroEJ Application point of view, this GPIO has the ID 30, 5.

  Standard virtual connector IDs are:

```
ARDUINO_DIGITAL = 30;
ARDUINO_ANALOG = 31;
```

  Finally, the available physical connectors can have a number from 64 to 64 + i - 1, where i is the available number of connectors on the board. This allows the application to easily target a GPIO that is available on a physical connector, without knowing the corresponding MCU port and pin.

```
JP3 = 64;
JP6 = 65;
JP11 = 66;
```

Configuration

A GPIO can be configured in any of five modes:

- Digital input: The MicroEJ Application can read the GPIO state (for example a button state).
- Digital input pull-up: The MicroEJ Application can read the GPIO state (for example a button state); the default GPIO state is driven by a pull-up resistor.
- Digital output: The MicroEJ Application can set the GPIO state (for example to drive an LED).
- Analog input: The MicroEJ Application can convert some incoming analog data into digital data (ADC). The returned values are values between `0` and `n - 1`, where `n` is the ADC precision.
- Analog output: The MicroEJ Application can convert some outgoing digital data into analog data (DAC). The digital value is a percentage (0 to 100%) of the duty cycle generated on selected GPIO.

Dependencies

- `LLHAL_impl.h` implementation (see *LLHAL: Hardware Abstraction Layer*).

Installation

HAL is an additional module. To enable it, the Audio *Pack* module must be installed in your VEE Port:

SDK 6 (build.gradle.kts)

SDK 5 (module.ivy)

```
microejPack("com.microej.pack:hal:2.0.2")
```

```
<dependency org="com.microej.pack" name="hal" rev="2.0.2"/>
```

Then, using the VEE Port Editor (see *Platform Module Configuration*), enable the *HAL* library.

Use

The HAL API Module must be added to the project build file to use the HAL library:

SDK 6 (build.gradle.kts)

SDK 5 (module.ivy)

```
implementation("ej.api:hal:1.0.4")
```

```
<dependency org="ej.api" name="hal" rev="1.0.4"/>
```

3.6.25 Device Information

Principle

The Device Foundation Library provides access to the device information. This includes the architecture name and a unique identifier of the device for this architecture.

Dependencies

- LLDEVICE_impl.h implementation (see *LLDEVICE: Device Information*).

Installation

Device is an additional module. To enable it, the Device Information *Pack* module must be installed in your VEE Port:

SDK 6 (build.gradle.kts)

SDK 5 (module.ivy)

```
microejPack("com.microej.pack.device:device-pack:1.2.0")
```

```
<dependency org="com.microej.pack.device" name="device-pack" rev="1.2.0"/>
```

Then, using the VEE Port Editor (see *Platform Module Configuration*), enable the *Device Information* library.

When installed, the Device Information Pack module can be configured.

SDK 6

SDK 5

In SDK 6, the configuration is done in the properties file configuration.properties of the VEE Port project. All the properties names listed below must be prefixed by com.microej.pack.device. . For example the architecture properties is defined by the com.microej.pack.device.architecture property.

In SDK 5, the configuration is done in the properties file device/device.properties .

The list below describes the properties that can be defined in the configuration file:

- architecture [optional, default value is "Virtual Device"]: Defines the value returned by the ej.util.Device.getArchitecture() method on the Simulator.
- id.length [optional]: Defines the size of the ID returned by the ej.util.Device.getId() method on the Simulator.

Use

The Device API Module must be added to the project build file to use the Device Information library:

SDK 6 (build.gradle.kts)

SDK 5 (module.ivy)

```
implementation("ej.api:device:1.0.2")
```

```
<dependency org="ej.api" name="device" rev="1.0.2"/>
```

3.6.26 Security

Principle

The Security Foundation Library provides standard Java API (part of the Java Cryptography Architecture) for cryptographic operations: cipher, digest, MAC, signature, secure random & key/certificate management. It relies on a native crypto engine (such as Mbed TLS, OpenSSL or wolfSSL).

Dependencies

- The `LLSEC_*.h` implementations (see *LLSEC: Security*).

Installation

Security is an additional module. To enable it, the Net *Pack* (which bundles several libraries: Net, SSL & Security) must be installed in your VEE Port:

SDK 6 (build.gradle.kts)

SDK 5 (module.ivy)

```
microejPack("com.microej.pack.net:net-pack:11.0.2")
```

```
<dependency org="com.microej.pack.net" name="net-pack" rev="11.0.2"/>
```

Then, using the VEE Port Editor (see *Platform Module Configuration*), enable the Security library (API, Impl & Mock):

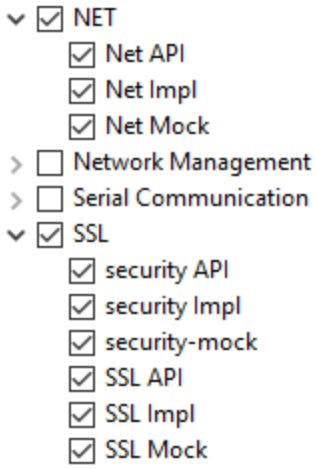


Fig. 251: Net Pack Modules

Use

The Security API module must be added to the Application project build file to use the Security library:

SDK 6 (build.gradle.kts)

SDK 5 (module.ivy)

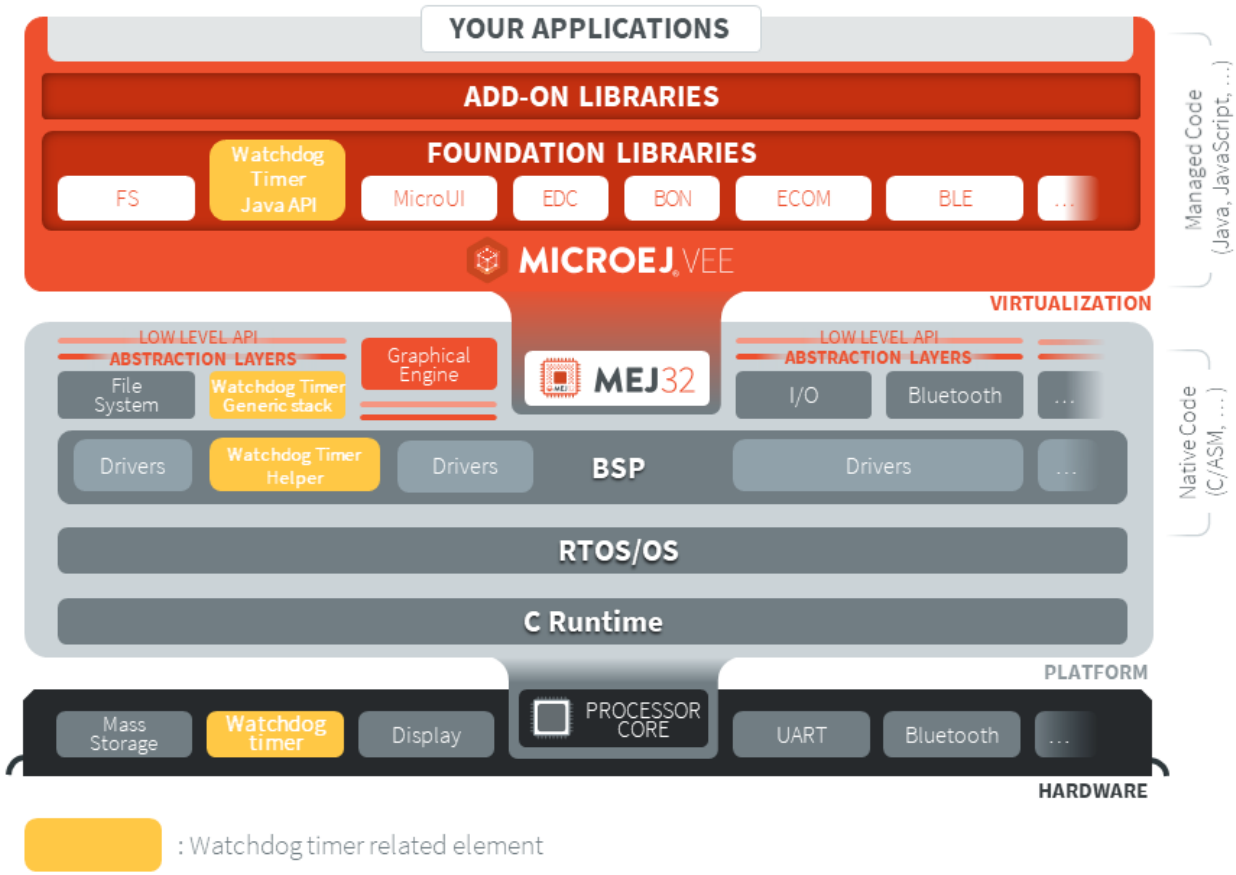
```
implementation("ej.api:security:1.6.0")
```

```
<dependency org="ej.api" name="security" rev="1.6.0"/>
```

3.6.27 Watchdog Timer

Overview

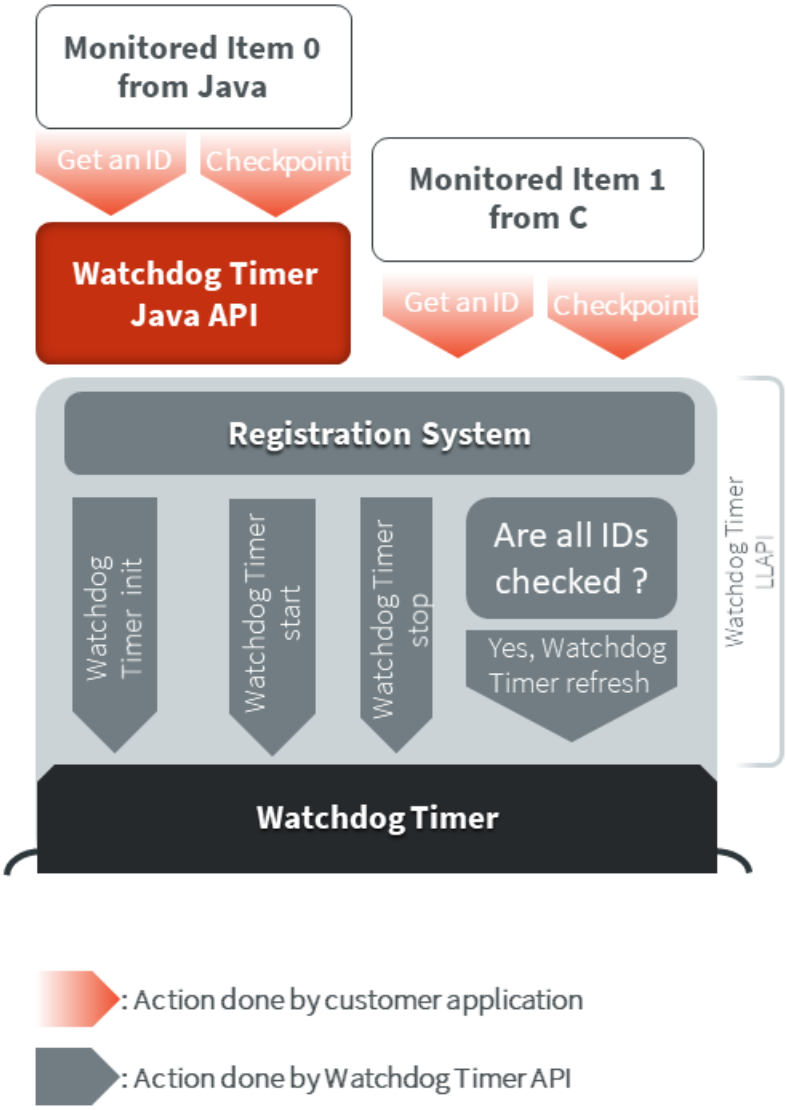
The WatchdogTimer Foundation Library provides a way to handle hardware watchdog timer. A watchdog is particularly useful if you want to monitor different items of your software system during the runtime. The figure below shows watchdog elements at each level of a MicroEJ project:



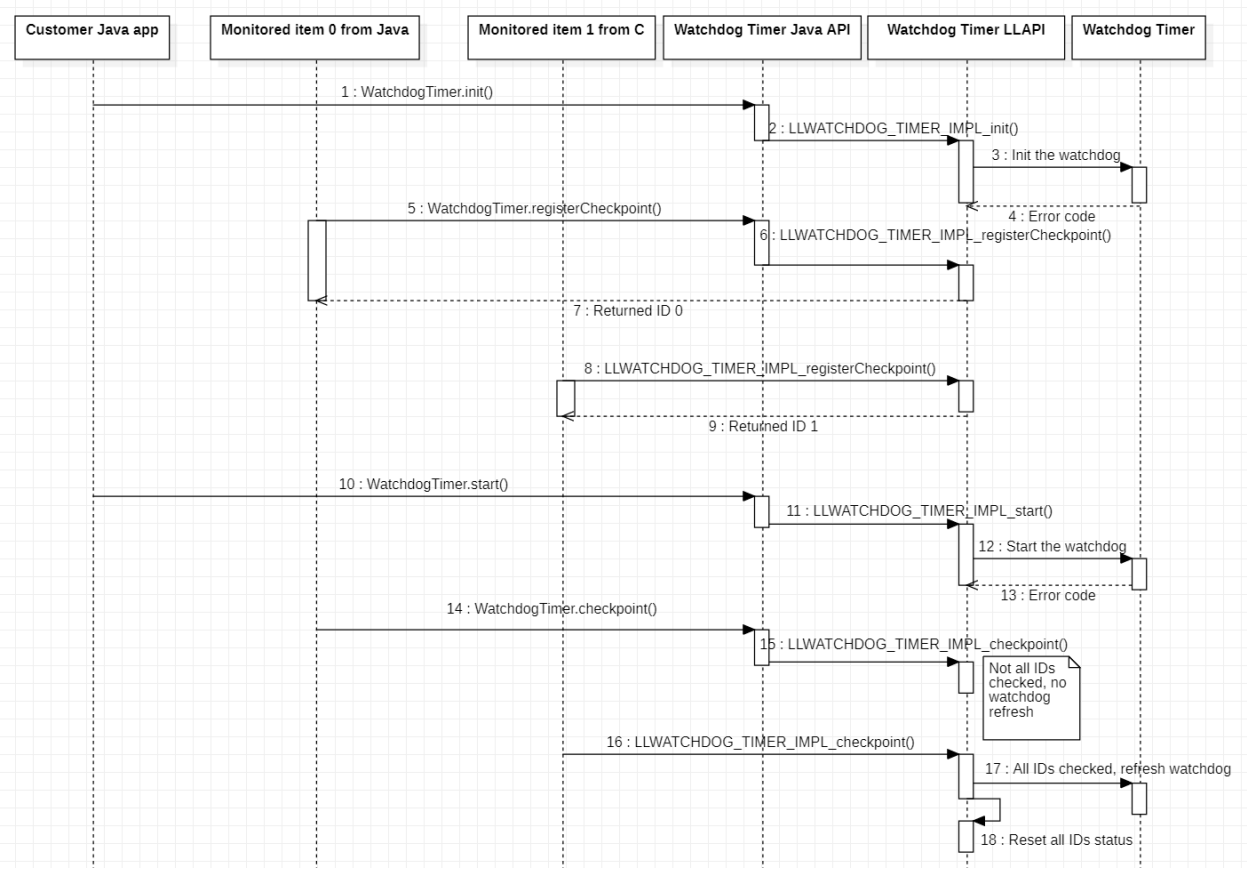
Principle

The Watchdog Timer API is in **two parts**, the first part drives the **watchdog timer** itself. The second part of the API implements a **checkpoint registration system** linked to the watchdog timer.

The checkpoint registration system allows the user to add checkpoints monitored by the hardware watchdog timer. Each checkpoint registered by the Watchdog Timer API must attest their activity before the watchdog timeout, otherwise a **hardware reset** is performed. The high level diagram below summarizes interactions between the user application, the Watchdog Timer API and the Watchdog timer.



The particularity of this library is that it can be either used in Managed code or in native code inside the BSP or even both of them. The use of this library in the BSP in C is relevant when the user needs to monitor an item of the software system which is not executed by the Core Engine. The sequence diagram below shows a standard use of the Watchdog API in Java code and in C code.



Mock Implementation

When you run your Application on the Simulator, the watchdog timer must be emulated. To do so, a Java Timer Task is used which emulates the watchdog timer.

The Mock implementation does not perform a hardware reset when the false watchdog timer triggers.

Dependencies

- This library needs to be used with the Watchdog Timer Generic C implementation developed for this purpose, its module name is watchdog-timer-generic .
- LLWATCHDOG_TIMER_impl.h implemented by the Watchdog Timer C implementation (see *LLWATCHDOG_TIMER: Watchdog Timer*).
- watchdog_timer_helper.h implementation needed by the Watchdog Timer C implementation (see *LLWATCHDOG_TIMER: Watchdog Timer*).

Installation

Watchdog Timer is an API composed of a *Pack* module and a C component module. You need both of them in your VEE Port to install the API.

In the Platform configuration project, (`-configuration` suffix), add the following dependencies to *module.ivy* file:

```
<dependency␣
↪org="com.microej.pack.watchdog-timer" name="watchdog-timer-pack" rev="2.0.1" />
<dependency␣
↪org="com.microej.clibrary.llimpl" name="watchdog-timer-generic" rev="3.0.1"/>
```

The Platform project must be rebuilt (*VEE Port Build*).

The Watchdog Timer *Pack* module must be installed in your VEE Port:

SDK 6 (build.gradle.kts)

SDK 5 (module.ivy)

```
microejPack("com.microej.pack.watchdog-timer:watchdog-timer-pack:2.0.1")
```

```
<dependency␣
↪org="com.microej.pack.watchdog-timer" name="watchdog-timer-pack" rev="2.0.1" />
```

As well as the C Component module:

SDK 6 (build.gradle.kts)

SDK 5 (module.ivy)

Install the Watchdog Timer C Component module in your VEE Port.

```
<dependency␣
↪org="com.microej.clibrary.llimpl" name="watchdog-timer-generic" rev="3.0.1"/>
```

Then the VEE Port project must be rebuilt (*VEE Port Build*).

Then, you have to implement functions that match the `LLWATCHDOG_TIMER_IMPL_*_action` pattern which is required by the Watchdog C implementation.

Use in an Application

The WatchdogTimer API Module must be added to the project build file to use the Watchdog library:

SDK 6 (build.gradle.kts)

SDK 5 (module.ivy)

```
implementation("ej.api:watchdog-timer:2.0.0")
```

```
<dependency org="ej.api" name="watchdog-timer" rev="2.0.0"/>
```

Java Code Eexample

Here is an example that summarizes all features in a simple use case. The checkpoint is performed in a TimerTask scheduled to run every 5 seconds. To use TimerTask in your Java application, add the following BON API dependency:

SDK 6 (build.gradle.kts)

SDK 5 (module.ivy)

```
implementation("ej.api:bon:1.4.4")
```

```
<dependency org="ej.api" name="bon" rev="1.4.4" />
```

Then, you can use this example code:

```
// Test a simple watchdog timer use case
public static void main(String[] args) {

	if (WatchdogTimer.isResetCause()) {
		System.out.println("Watchdog timer triggered the last board reset!"); //$NON-NLS-1$
	} else {
		System.out.println("Watchdog timer DID NOT triggered the last board reset!"); //$NON-NLS-1$
	}

	WatchdogTimer.init();
	System.out.println("Watchdog timer initialized to trigger after " + WatchdogTimer.getWatchdogTimeoutMs() + " ms."); //$NON-NLS-1$

	TimerTask checkpointTask = new TimerTask() {

		private final int checkpointId = WatchdogTimer.registerCheckpoint();

		@Override
		public void run() {
			// We attest our task activity using the checkpoint method.

			// Since this is our only checkpoint registered, the watchdog timer is refreshed.
			WatchdogTimer.checkpoint(this.checkpointId);
			System.out.println("Task performed watchdog checkpoint with the ID " + this.checkpointId); //$NON-NLS-1$
		}
	};

	// We schedule our task to be executed every 5 seconds.
	Timer timer = new Timer();
	final int DELAY = 0;
	final int PERIOD = 5000; // We assume that the watchdog timeout period is higher than 5000 milliseconds.
	timer.schedule(checkpointTask, DELAY, PERIOD);

	// Everything is ready, we launch the watchdog
	WatchdogTimer.start();
	System.out.println("Watchdog started!");

	// Let the checkpointTask runs for a minute.
```

(continues on next page)

(continued from previous page)

```
        final int WAIT_A_MINUTE = 60000; // 60 000 milliseconds to wait a minute
        try {
                Thread.sleep(WAIT_A_MINUTE);
        } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
        }

        // Our program is finished. Now we stop the checkpointTask and the watchdog.
        timer.cancel();
        WatchdogTimer.stop(); // This method also unregisters all checkpoints.
        System.out.println("Monitored task stopped and Watchdog timer stopped.");
}
```

Use in C inside the BSP

Once the Platform is configured to use the Watchdog Timer API as explained in Installation section, you can use functions defined in LLWATCHDOG_TIMER_impl.h .

Note that compared to the Java API, you have to get error codes returned by functions to check if the function is executed correctly since you have no access to exceptions generated for the Java code.

The Watchdog Timer Low Level API provides a set of functions with the same usage as in Java code. Here is the list of the watchdog Low Level API functions:

```
LLWATCHDOG_
↪TIMER_IMPL_init()                    // refer to ej.hal.WatchdogTimer.init()
LLWATCHDOG_
↪TIMER_IMPL_start()                   // refer to ej.hal.WatchdogTimer.start()
LLWATCHDOG_
↪TIMER_IMPL_stop()                    // refer to ej.hal.WatchdogTimer.stop()
LLWATCHDOG_TIMER_
↪IMPL_registerCheckpoint()        // refer to ej.hal.WatchdogTimer.registerCheckpoint()
LLWATCHDOG_TIMER_
↪IMPL_unregisterCheckpoint()      // refer to ej.hal.WatchdogTimer.unregisterCheckpoint()
LLWATCHDOG_
↪TIMER_IMPL_checkpoint()              // refer to ej.hal.WatchdogTimer.checkpoint()
LLWATCHDOG_
↪TIMER_IMPL_isResetCause()            // refer to ej.hal.WatchdogTimer.isResetCause()
LLWATCHDOG_TIMER_IMPL_
↪getWatchdogTimeoutMs()      // refer to ej.hal.WatchdogTimer.getWatchdogTimeoutMs()
```

There is an additional function in LLWATCHDOG_TIMER_impl.h compared to the Java API. This is LLWATCHDOG_TIMER_IMPL_refresh , because a low level implementation of this function is required for the library. However, the user does not need and should not use this function on his own.

Code example in C

Here is an example that summarizes main features in a simple use case. The checkpoint is performed in a FreeRTOS task scheduled to attest its activity to the watchdog every 5 seconds.

```
#include <stdio.h>
#include <stdint.h>

#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "semphr.h"

#include "LLWATCHDOG_TIMER_impl.h"

#define MONITORED_TASK_STACK_SIZE 1024
#define TASK_SLEEP_
↪TIME_MS 5000 // We sleep for 5 seconds, assuming that the watchdog timeout is higher.

/*-----------------------------------------*/

static void my_monitored_task( void *pvParameters ){
	// We get an ID from watchdog registration system for this new checkpoint
	int32_t checkpoint_id = LLWATCHDOG_TIMER_IMPL_registerCheckpoint();

	for(;;){
		vTaskDelay( TASK_SLEEP_TIME_MS / portTICK_PERIOD_MS);
		// Since this is our only checkpoint registered, the watchdog timer is refreshed.
		LLWATCHDOG_TIMER_IMPL_checkpoint(checkpoint_id);
  ␣
↪		printf("MonitoredTask with ID = %d did watchdog checkpoint!\n", checkpoint_id);
	}
}

/*-----------------------------------------*/

int main( void ){
	xTaskHandle handle_monitored_task;

	/* Check if last reset was done by the Watchdog timer. */
	if(LLWATCHDOG_TIMER_IMPL_isResetCause()){
		printf("Watchdog timer triggered the last reset, we stop the program now! \n");
		return -1;
	}

	/* Setup the Watchdog Timer*/
	if(WATCHDOG_TIMER_ERROR == LLWATCHDOG_TIMER_IMPL_init()){
		printf("Failed to init watchdog timer in main. \n");
	} else{
		printf("Watchdog timer initialized␣
↪to trigger after %d ms \n", LLWATCHDOG_TIMER_IMPL_getWatchdogTimeoutMs());
	}

	/* Start the Watchdog Timer*/
	if(WATCHDOG_TIMER_ERROR == LLWATCHDOG_TIMER_IMPL_start()){
		printf("Failed to start watchdog timer in main. \n");
	} else{
		printf("Watchdog started!\n");
```

(continues on next page)

(continued from previous page)

```
    }

    /* Create the monitored task. */
    xTaskCreate( my_monitored_task, "MonitoredTask", MONITORED_
↪TASK_STACK_SIZE, NULL, tskIDLE_PRIORITY, &handle_monitored_task);

    /* Start the scheduler. */
    printf("Starting scheduler...\n");
    vTaskStartScheduler();

    return 0;
}
```

3.6.28 SystemView

Note: In the following explanations, the term *task* refers to native tasks scheduled by the underlying OS or RTOS, while *thread* refers to MicroEJ threads scheduled by the Core Engine.

Principle

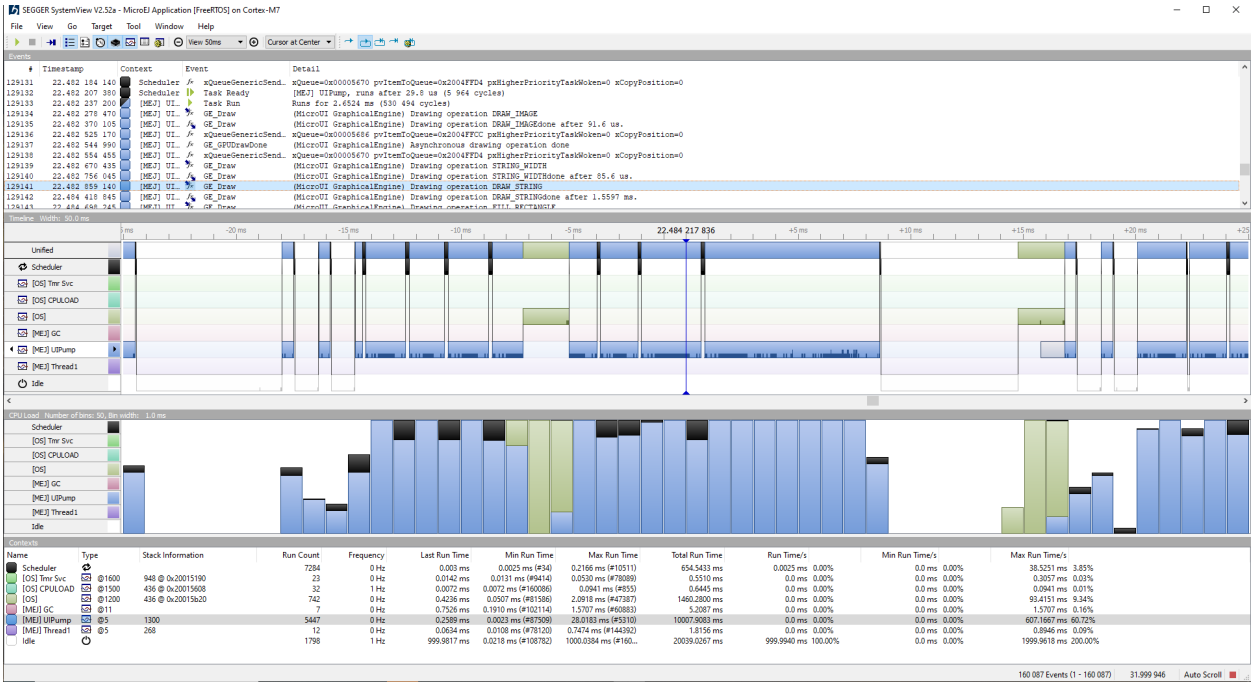
SystemView is a real-time recording and visualization tool for embedded systems that reveals the actual runtime behavior of an application, going far deeper than the system insights provided by debuggers. This is particularly effective when developing and working with complex embedded systems comprising multiple threads and interrupts.

A specific SystemView extension made by MicroEJ allows to trace the OS tasks and the MicroEJ threads at the same time. This chapter explains how to add the SystemView feature to a VEE Port and set it up.

Note: MicroEJ supports integration with SystemView on FreeRTOS 9 and FreeRTOS 10.

Note: SystemView provides implementations for several combinations of toolchain and CPU architectures. If your target does not match a default configuration, please read the section *Non default CPU recommendations* in order to know how to implement the required functions.

Here is an example when analyzing the Demo Widget running on the STM32F7508-DK VEE Port.



References

- https://www.segger.com/products/development-tools/systemview/
- https://www.segger.com/downloads/jlink/UM08027

Pre-requisites

SEGGER SystemView: http://segger.com/downloads/systemview/.

Note: This SystemView section has been written for SystemView version V2.52a. Later versions may need modification to the following steps.

Note: This download page also contains links to download the “Target Sources” package. The following sections of this documentation will go through the installation of those. If your BSP project already integrates a specific version, make sure to download & install the matching version of the SystemView program.

Integrate SystemView in your VEE Port

For detailed instructions on the integration of SystemView Target Sources in your BSP, please refer to the “Getting started with SystemView on the target” section of the SystemView User Manual. This documentation focuses on the integration with FreeRTOS. If you are integrating with a different RTOS, make sure to check the samples in the Target Sources package.

Apply FreeRTOS patch

In FreeRTOS v9.x & v10.x, SystemView requires a patch to FreeRTOS source files in order to inject OS event hooks when SystemView profiling is enabled.

Starting from FreeRTOS v11: SystemView integrates with FreeRTOS new built-in mechanism for profiling OS events (Trace Hook Macros). You can skip this step if you use FreeRTOS v11.

Apply SystemView for FreeRTOS patch as described in the documentation (see section 4.7.5). The patch is available in the Systemview installation folder: <installation_dir>/Src/Sample/FreeRTOSVxx .

Note: For FreeRTOS v10.2.0, the official patch may not work. Here is an unofficial patch

Add source files to your BSP for SystemView with MicroEJ/FreeRTOS integration

1. Add <installation_dir>/Src/Config/SEGGER_SYSVIEW_Config_FreeRTOS.c in your BSP.

   This file can be modified to fit your system configuration:

- Update SYSVIEW_APP_NAME , SYSVIEW_DEVICE_NAME , and SYSVIEW_RAM_BASE defines to fit your system information.
- To add MicroEJ threads management in SystemView tasks initialization:

– Add these includes #include "task.h" , #include "LLMJVM_MONITOR_SYSVIEW.h" , #include "LLTRACE_SYSVIEW_configuration.h" , #include "SEGGER_SYSVIEW_configuration.h" and the include that declares the external variable pvMEJCoreEngineTask . pvMEJCoreEngineTask must be the FreeRTOS task handle used to create the Core Engine task. Initializes this variable at NULL before the call of the FreeRTOS scheduler.

– In function _cbSendSystemDesc(void) , add this instruction: SEGGER_SYSVIEW_SendSysDesc("N="SYSVIEW_APP_NAME", D="SYSVIEW_DEVICE_NAME",O=FreeRTOS"); before SEGGER_SYSVIEW_SendSysDesc("I#15=SysTick"); .

– Replace the Global function section with this code:

```
/*********************************************************************
*
*       Global functions
*
**********************************************************************
*/

SEGGER_SYSVIEW_OS_API SYSVIEW_MICROEJ_X_OS_TraceAPI;

static void SYSVIEW_MICROEJ_X_OS_SendTaskList(void){
  SYSVIEW_X_OS_TraceAPI.pfSendTaskList();

// The strategy to send tasks info is different in post mortem and live analysis.
#if (1 == SEGGER_SYSVIEW_POST_MORTEM_MODE)
  /**
  * POST MORTEM analysis
```

(continues on next page)

(continued from previous page)

```
  *
  * Using the post mortem analysis, FreeRTOS␣
↪tasks regularly call the SYSVIEW_MICROEJ_X_OS_SendTaskList() function when
  * a packet (systemview event)␣
↪is sent to the SEGGER circular buffer. It is necessary because the information of tasks
  * must be␣
↪regularly uploaded in the circular buffer in order to provide a valid analysis at any moment.
  * Consequently, we only allow to call LLMJVM_
↪MONITOR_SYSTEMVIEW_send_task_list() when the current task is the Core Engine.
  */

  /* Obtain the handle of the current task. */
  TaskHandle_t xHandle = xTaskGetCurrentTaskHandle();
  configASSERT( xHandle ); // Check the handle is not NULL.

  // Check if the current␣
↪task handle is the Core Engine task handle. pvMEJCoreEngineTask is an external variable.
  if( xHandle == pvMEJCoreEngineTask){
    // Launched␣
↪by the Core Engine, we execute LLMJVM_MONITOR_SYSTEMVIEW_send_task_list()
    LLMJVM_MONITOR_SYSTEMVIEW_send_task_list();
  }
#else
  /**
  * LIVE analysis
  *
  * Using␣
↪the live analysis, the call of SYSVIEW_MICROEJ_X_OS_SendTaskList() is triggered by
  * the SystemView␣
↪Software through the J-Link probe. Consequently, the Core Engine task will never call
  * the function LLMJVM_
↪MONITOR_SYSTEMVIEW_send_task_list(). However, if the Core Engine task is created,
  * the function must be called LLMJVM_MONITOR_SYSTEMVIEW_send_task_list().
  */
  // Check␣
↪if the Core Engine task handle is not NULL. pvMEJCoreEngineTask is an external variable.
  if( NULL != pvMEJCoreEngineTask){
    // The Core Engine␣
↪task is running, we execute LLMJVM_MONITOR_SYSTEMVIEW_send_task_list()
    LLMJVM_MONITOR_SYSTEMVIEW_send_task_list();
  }
#endif
}

void SEGGER_SYSVIEW_Conf(void) {
  SYSVIEW_
↪MICROEJ_X_OS_TraceAPI.pfGetTime = SYSVIEW_X_OS_TraceAPI.pfGetTime;
  SYSVIEW_MICROEJ_
↪X_OS_TraceAPI.pfSendTaskList = SYSVIEW_MICROEJ_X_OS_SendTaskList;

  SEGGER_SYSVIEW_Init(SYSVIEW_TIMESTAMP_FREQ, SYSVIEW_
↪CPU_FREQ, &SYSVIEW_MICROEJ_X_OS_TraceAPI, _cbSendSystemDesc);
  SEGGER_SYSVIEW_SetRAMBase(SYSVIEW_RAM_BASE);
}
```

2. Add in your BSP the MicroEJ C module files for SystemView: com.microej.clibrary.third-party#systemview. These files correspond to the sources provided in the installation folder of

Systemview (<installation_dir>/Src/SEGGER) with a patch for MicroEJ (including recommended configuration and prefixes for task names).

3. Add in your BSP the MicroEJ C module files for SystemView FreeRTOS support. These files correspond to the sources provided in the installation folder of Systemview (<installation_dir>/Src/Sample/FreeRTOSVXX and <installation_dir>/Src/Sample/FreeRTOSVXX/Config) with a patch for MicroEJ.

- FreeRTOS 10: com.microej.clibrary.thirdparty#systemview-freertos10
- For other versions of FreeRTOS, please contact *our support team*.

Not all OS events are useful to the profiling analysis. It can be useful to filter out events from systick/queue/... in order to reduce the load and improve the stability of the analysis. To disable these traces on FreeRTOS, modify SEGGER_SYSVIEW_FreeRTOS.h :

- Comment the defines beginning by traceISR .
- Comment the defines beginning by traceQUEUE .

4. Add in your BSP the Abstraction Layer implementation of the *Java Trace API* for SystemView by adding C module files in your BSP: com.microej.clibrary.llimpl#trace-systemview

Configure FreeRTOS for SystemView

Open FreeRTOSConfig.h and make these changes:

1. add #define INCLUDE_xTaskGetIdleTaskHandle 1
2. add #define INCLUDE_pxTaskGetStackStart 1
3. add #define INCLUDE_uxTaskPriorityGet 1
4. comment the line #define traceTASK_SWITCHED_OUT() if defined
5. comment the line #define traceTASK_SWITCHED_IN() if defined
6. add #include "SEGGER_SYSVIEW_FreeRTOS.h" at the end of the file

Modify startup code of your BSP

1. Enable SystemView on startup (before creating the first OS task): call SEGGER_SYSVIEW_Conf(); . The following include directive is required: #include "SEGGER_SYSVIEW.h" .
2. Print the RTT block address to the serial port on startup: printf("SEGGER_RTT block address: %p\n", &(_SEGGER_RTT)); . The following include directive is required: #include "SEGGER_RTT.h" .

Note: This is useful if SystemView does not automatically find the RTT block address. See section *RTT Control Block Not Found* for more details. You may also find the RTT block address in RAM by searching _SEGGER_RTT in the .map file generated with the firmware binary.

---

3. Add a call to SYSVIEW_setMicroJVMTask(pvCreatedTask); just after creating the OS task. The handler to give is the parameter of type TaskHandle_t passed to the xTaskCreate function. This will register the Core Engine OS task.

Add description files to Systemview installation folder

Copy the file SYSVIEW_MicroEJ.txt of the C module com.microej.clibrary.llimpl#trace-systemview to the SystemView installation path, such as SEGGER/SystemView_V252a/Description/ . If your VEE Port integrates MicroUI, also add the trace descriptions files from *Debug Traces*.

Non default CPU recommendations

As mentioned in the SEGGER documentation, SystemView can be used on any CPU. In the case of a CPU not supported by default, the following macros that redirect to 4 functions must be implemented:

- SEGGER_SYSVIEW_GET_TIMESTAMP()
- SEGGER_SYSVIEW_GET_INTERRUPT_ID()
- SEGGER_SYSVIEW_LOCK()
- SEGGER_SYSVIEW_UNLOCK()

These 4 macros are defined by default in the file SEGGER_SYSVIEW_ConfDefaults.h and should be re-defined in the file SEGGER_SYSVIEW_configuration.h when it is necessary.

The function macro SEGGER_SYSVIEW_GET_TIMESTAMP() retrieves the system timestamp for SystemView events. On most devices the system timestamp must be generated by a timer. With the default configuration, the system timestamp is retrieved from the user-provided function SEGGER_SYSVIEW_X_GetTimestamp() . The recommended minimum accuracy for this is on the order of magnitude of microseconds.

The function macro SEGGER_SYSVIEW_GET_INTERRUPT_ID() returns the currently active interrupt. On Cortex-M devices the active vector can be read from the ICSR register. On other devices, the active interrupt can either be retrieved from the interrupt controller directly, can be saved in a variable in the generic interrupt handler, or has to be assigned manually in each interrupt routine.

The function macro SEGGER_SYSVIEW_LOCK() recursively locks SystemView transfers from being interrupted, by disabling the interrupts. Recording a SystemView event must not be interrupted by recording another event. By default, this function is implemented with the function macro SEGGER_RTT_LOCK() . However, this definition may be empty for your system. In this case, implement the function to disable interrupt and context switching.

The function macro SEGGER_SYSVIEW_UNLOCK() recursively unlocks SystemView transfers from being interrupted, by retoring the previous interrupt state. Follow the same recommendations as for the function macro SEGGER_SYSVIEW_LOCK() .

Warning: Empty implementations of SEGGER_RTT_LOCK() and SEGGER_RTT_LOCK() will not cause an error at link time, so check the implementation of these two function macros carefully.

Note: If the target is not connected to a J-Link probe, post mortem analysis is still possible.

Post Mortem analysis data extraction

First of all, in the file SEGGER_SYSVIEW_configuration.h , be sure that the macro SEGGER_SYSVIEW_POST_MORTEM_MODE is set to 1 and increase the size of the SEGGER_SYSVIEW_RTT_BUFFER_SIZE . Then, start manually the SystemView recording by calling SEGGER_SYSVIEW_Start() at runtime.

1. When the system crashed or all tests are done, attach with a debugger to the system and halt it.
2. Get the SystemView RTT buffer (Usually _SEGGER_RTT.aUp[1].pBuffer).

   When using a debugger, the SEGGER_RTT buffer can be located using the Expressions tab by adding the tracking of the expression _SEGGER_RTT symbol.
3. Save the data from pBuffer + WrOff until the end of the buffer to a file.
4. Append the data from pBuffer until pBuffer + RdOff - 1 to the file.
5. Save the file as *.SVdat or *.bin.

   The append step and the step 5 can be done in one cat instruction as shown below:

```
cat sysview_dump_1_wroff.bin sysview_dump_1_rdoff.bin > concat_dump_1.bin
```

The file generated can now be read by the SystemView software.

Note: Instructions on how to retrieve the right SEGGER RTT buffer are also available on the SEGGER website.

Usage

Trace application events

To enable events recording, refer to the *Event Recording* section to configure the required *Application Options*.

Add custom events to the SystemView analysis

MicroEJ Architecture can generate specific events that allow monitoring of current thread, Java exceptions, Java allocations, … as well as custom application events. Please refer to the *Event Tracing* section.

For custom application events, the first step is to add logs to the Java application using a dedicated Tracer . Please read the documentation page *Code Instrumentation for Logging*. Below is an example of Tracer usage:

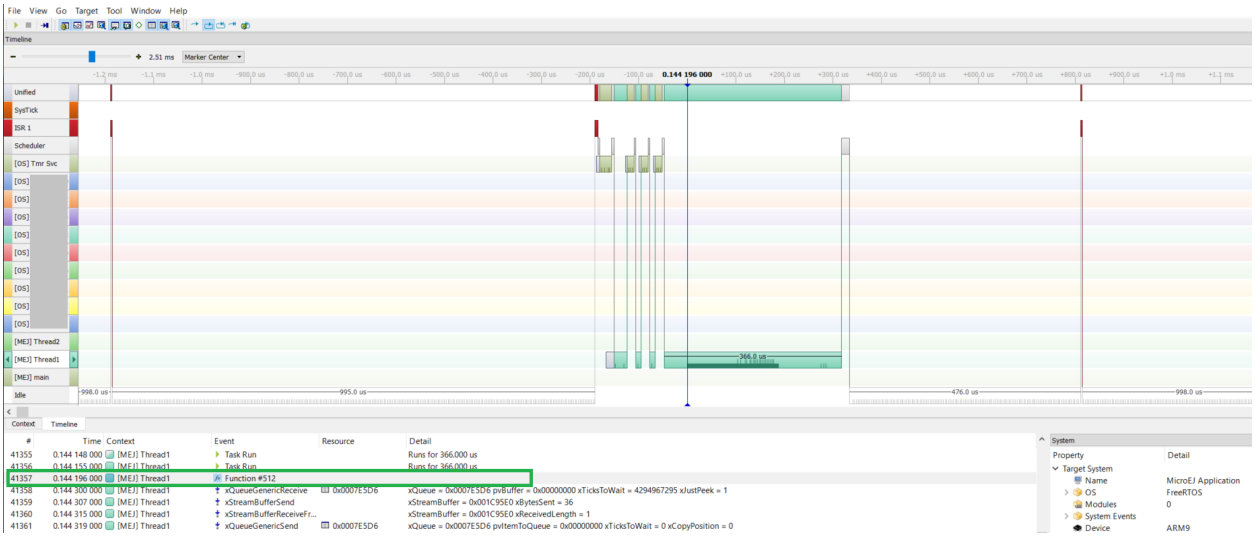
```
Tracer tracer = new Tracer("MyGroup", 10);

if (Constants.getBoolean(Tracer.TRACE_ENABLED_CONSTANT_PROPERTY)) {
   // This␣
↪code is not embedded if TRACE_ENABLED_CONSTANT_PROPERTY is set to false.
   tracer.recordEvent(0);
}

// Do some actions HERE...
```

(continues on next page)

(continued from previous page)

```
if (Constants.getBoolean(Tracer.TRACE_ENABLED_CONSTANT_PROPERTY)) {
   // This␣
↪code is not embedded if TRACE_ENABLED_CONSTANT_PROPERTY is set to false.
   tracer.recordEventEnd(0);
}
```

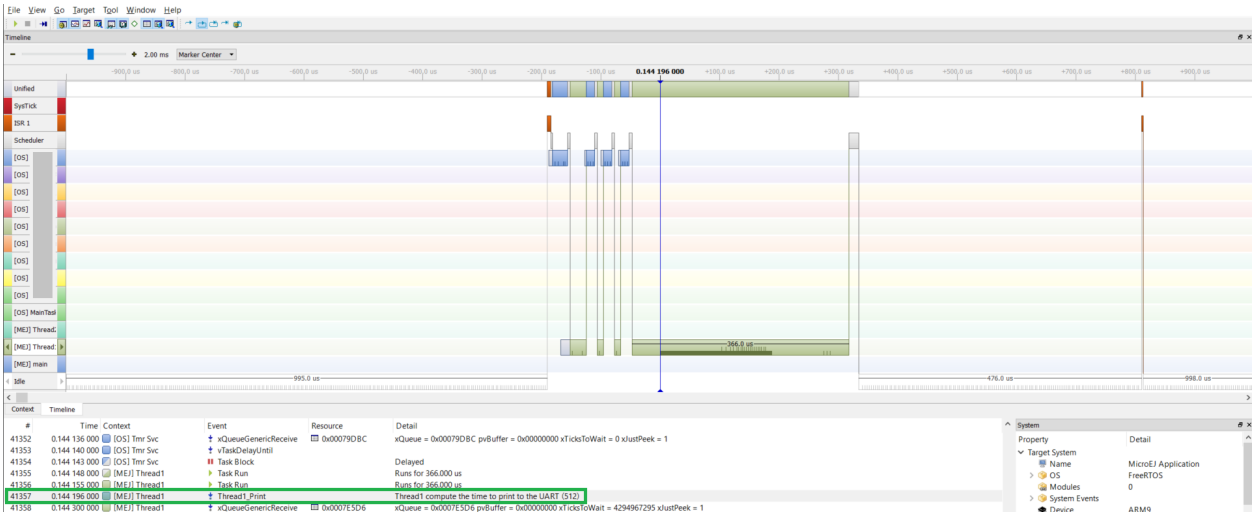
Run this application on the target with traces enabled and record the SystemView analysis. Then, search for the event in the SystemView logs with the timeline. In this example, the new event type is function #512 , see the screenshot below:



Now, it is necessary to indicate to SystemView how to decode this new event. To do that, you can either edit the file SYSVIEW_MicroEJ.txt or add a new text file in the SystemView install folder SEGGER/SystemView_VXXX/Description . In the chosen file, add the line as shown below:

```
512      Thread1_Print   Thread1 compute the time to print to the UART (512)
```

Finally, reload your analysis with SystemView and this time the new event should be decoded.



You can add more information if you use the versions of the Tracer API with more parameters. Then, if you want these parameters to appear in the SystemView timeline view, use the modifiers below:

- %b - Display parameter as binary.
- %B - Display parameter as hexadecimal string (e.g. 00 AA FF ...).
- %d - Display parameter as signed decimal integer.
- %D - Display parameter as time value.
- %I - Display parameter as a resource name if the resource id is known to SystemView.
- %p - Display parameter as 4 byte hexadecimal integer (e.g. 0xAABBCCDD).
- %s - Display parameter as string.
- %t - Display parameter as a task name if the task id is known to SystemView.
- %u - Display parameter as unsigned decimal integer.
- %x - Display parameter as hexadecimal integer.

Check the other text files provided by SEGGER for more examples in the install directory SEGGER/SystemView_VXXX/Description .

Core Engine OS Task

The *Core Engine* task is the native OS task that executes the Core Engine internals & the Application threads. The provided SystemView configuration replaces (splits) the execution segments of this OS task with (into) the different components that are actually executed. This simplifies profiling analysis by exposing the execution segments of the Scheduler, Garbage Collector & the different Application threads (with their names, see the section below) directly into SystemView's timeline, along with the other native OS tasks.

OS Tasks and Threads Names

To make a distinction between the OS tasks and the MicroEJ threads, a prefix is added to the OS tasks names ([OS]) and the threads names ([MEJ]).

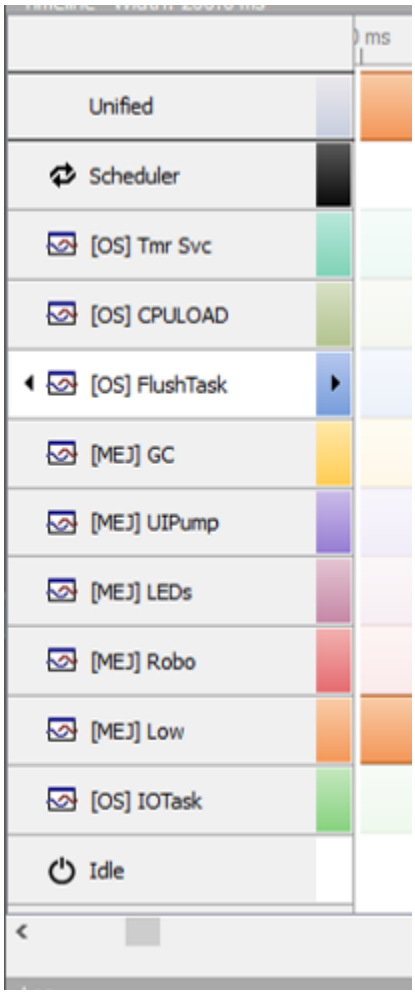


Fig. 252: OS Tasks and Threads Names

Note: SystemView limits the number of characters to 32. The prefix length is included in these 32 characters; consequently, the end of the original OS task or thread name can be cropped.

OS Tasks and Threads Priorities

SystemView lists the OS tasks and threads according to their priorities. However, the priority notion does not have the same signification when talking about OS tasks or threads: a thread priority depends on the Core Engine OS task priority.

As a consequence, a thread with the priority 5 may not appear between an OS task with the priority 4 and another OS task with priority 6:

- if the Core Engine OS task priority is 3, the thread must appear below an OS task with priority 4.
- if the Core Engine OS task priority is 7, the thread must appear above an OS task with priority 6.

To keep a consistent line ordering in SystemView, the priorities sent to the SystemView client respect the following rules:

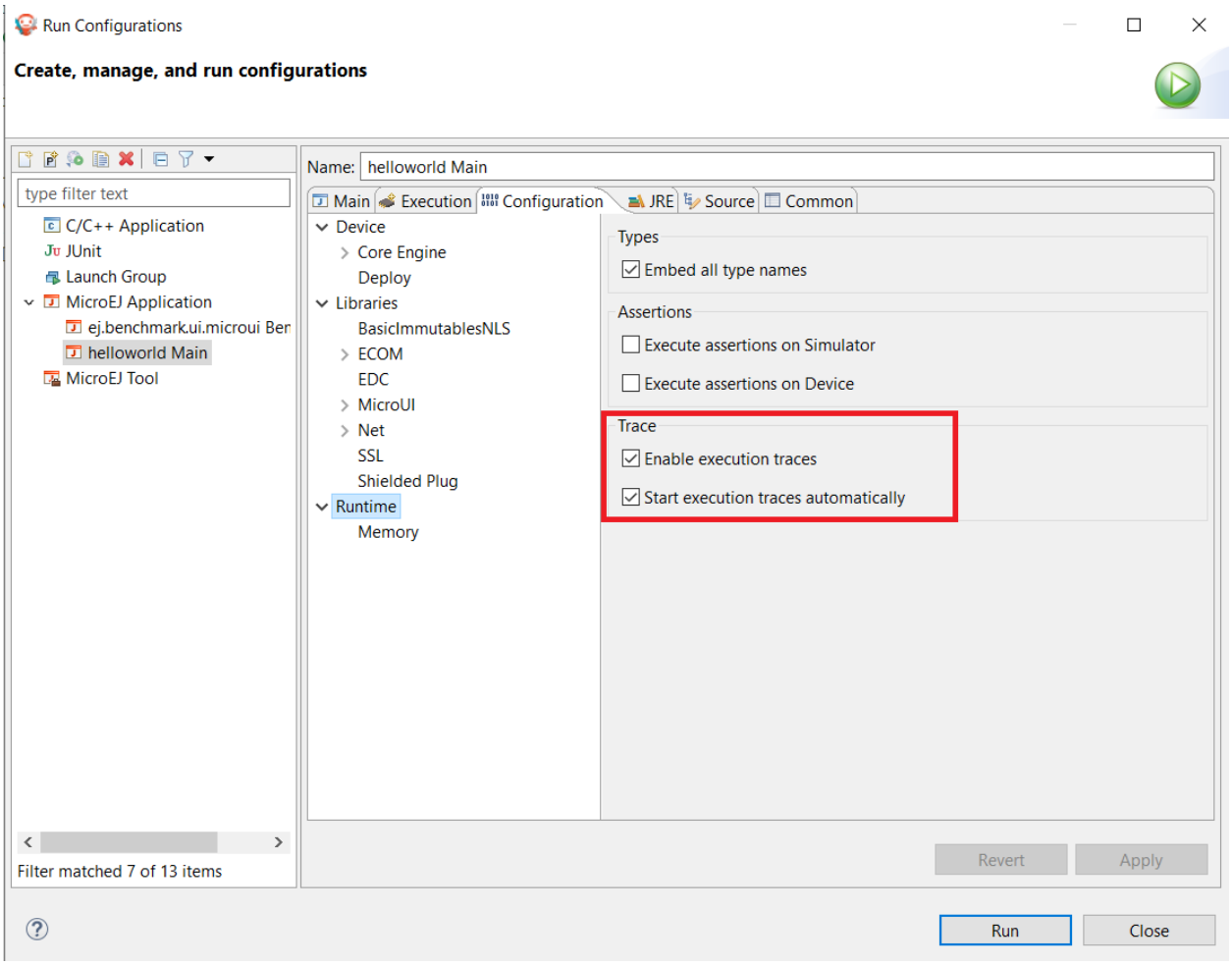
- OS task: $priority_sent = task_priority * 100$.
- MicroEJ thread: $priority_sent = MicroJvm_task_priority * 100 + thread_priority$.

Troubleshooting

SystemView doesn't see any activity in MicroEJ Tasks

You have to enable runtime traces of your Java application.

- In Run > Run configuration, select your Java application launcher.
- Then, go to Configuration tab > Runtime > Trace.
- Finally, check checkboxes Enable execution traces and Start execution traces automatically as shown in the picture below.
- Rebuild your firmware with the new Java application version, which should fix the issue.

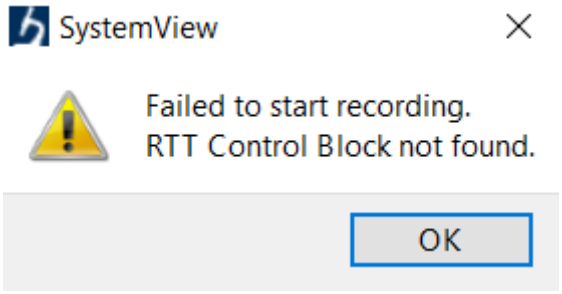


You may only check the first checkbox when you know when you want to start the trace recording. For more information, please refer to the *Event Recording* section to configure the required *Application Options*.

OVERFLOW Events in SystemView

Depending on the application, OVERFLOW events can be seen in System View. To mitigate this problem, the default *SEGGER_SYSVIEW_RTT_BUFFER_SIZE* can be increased from the default 1kB to a more appropriate size of 4kB. Still, if OVERFLOW events are still visible, the user can further increase this configuration found in */YourPlatformProject-bsp/projects/microej/thirdparty/systemview/inc/SEGGER_SYSVIEW_configuration.h* .

RTT Control Block Not Found



- Get the RTT block address from the standard output by resetting the board (it is printed at the beginning of the firmware program),
- In SystemView, select `Target` > `Start recording` ,
- In `RTT Control Block Detection` , select `Address` and put the address retrieved. You can also try with `Search Range` option.

RTT block found by SystemView but no traces displayed

- Be sure that your MCU is running. The BSP may use semi-hosting traces that block the MCU execution if the application is running out of a Debug session.
- You can check the state of the MCU using J-Link tools such as J-Link Commander and Ozone to start a Debug session.

Bus hardfault when running SystemView without Core Engine

The function LLMJVM_MONITOR_SYSTEMVIEW_send_task_list(); triggers a Bus Hardfault when no Core Engine is launched. To solve this issue, comment this function call out in SEGGER_SYSVIEW_Config_FreeRTOS.c when you run SystemView without launching the Core Engine.

Partial or wrong analysis with warning messages in the logs

It may happen that logs are not recorded well, with the following messages displayed in the log window:

```
Warning: Decoding 32-bit value failed. Bit 5 has continuation set
Warning: Error during file analysis.
```

The cause of this issue can be a wrong implementation of the locking function macros SEGGER_SYSVIEW_LOCK() and SEGGER_SYSVIEW_UNLOCK() . This problem may occur on a system target that is not supported by default in SystemView sources.

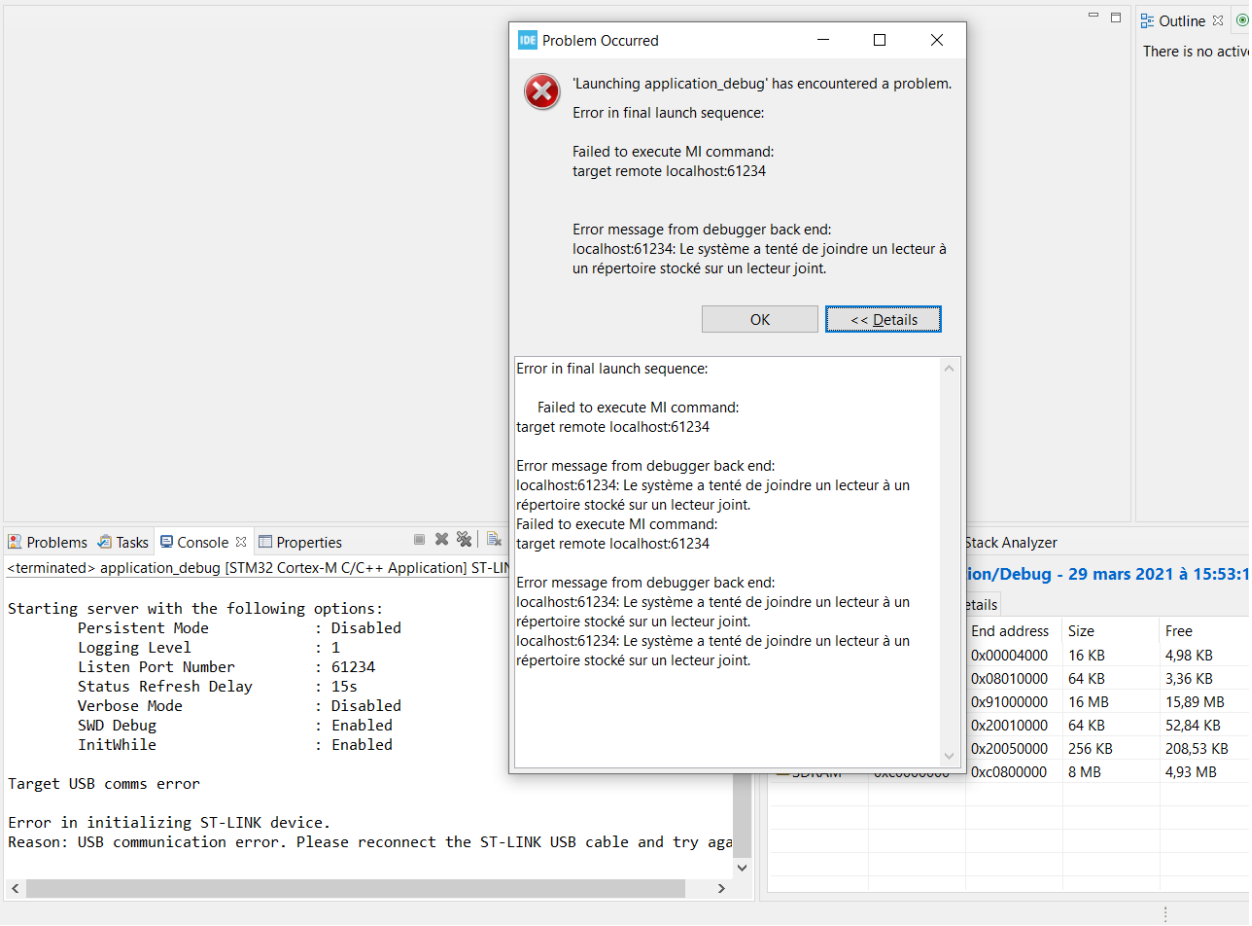
This issue has been discussed in the SEGGER forum topic here: https://forum.segger.com/index.php/Thread/8336-SOLVED-SystemView-stops-working-with-Error-140-and-141/

SystemView for STM32 ST-Link Probe

SystemView software requires a J-Link probe. If your target board uses an ST-Link probe, it is possible to re-flash the ST-LINK on board with a J-Link firmware. See instructions provided by SEGGER Microcontroller https://www.segger.com/products/debug-probes/j-link/models/other-j-links/st-link-on-board/ for more details.

If you cannot flash a firmware for an STM32 device after replacing the J-Link firmware with the ST-Link original one:

- Use ST_Link utility program to update the ST_Link firmware, go to `ST-LINK` > `Firmware update` .
- Then, try to flash again.



3.6.29 Simulation

Principle

The MicroEJ Platform provides an accurate MicroEJ Simulator that runs on workstations. Applications execute in an almost identical manner on both the workstation and on target devices. The MicroEJ Simulator features IO simulation, JDWP debug coupled with Eclipse, accurate Managed Heap dump, and an accurate Java scheduling policy (the same as the embedded one).[1]

Functional Description

In order to simulate external stimuli that come from the native world (that is, "the C world"), the MicroEJ Simulator has a Hardware In the Loop interface, HIL, which performs the simulation of Java-to-C calls. All Java-to-C calls are rerouted to an HIL engine. Indeed HIL is a replacement for the *[SNI]* interface.

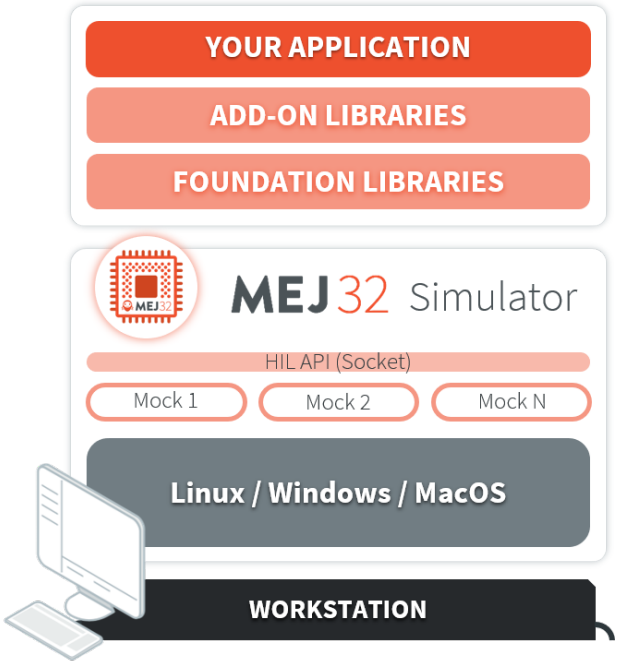


Fig. 253: The HIL Connects the MicroEJ Simulator to the Workstation.

The "simulated C world" is made of Mocks that simulate native code (such as drivers and any other kind of C libraries), so that the MicroEJ Application can behave the same as the device using the MicroEJ Platform.

The MicroEJ Simulator and the HIL are two processes that run in parallel: the communication between them is through a socket connection. Mocks run inside the process that runs the HIL engine.

[1] Only the execution speed is not accurate. The Simulator speed can be set to match the average MicroEJ Platform speed in order to adapt the Simulator speed to the desktop speed.

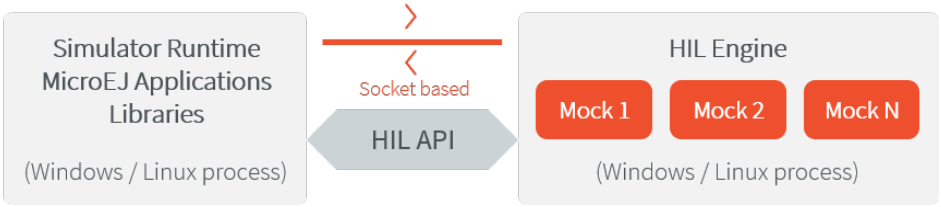


Fig. 254: A MicroEJ Simulator connected to its HIL Engine via a socket.

Dependencies

No dependency.

Installation

The Simulator is a built-in feature of MicroEJ Platform architecture.

Use

To run an application in the Simulator, create a MicroEJ launch configuration by right-clicking on the main class of the application, and selecting Run As > MicroEJ Application .

This will create a launch configuration configured for the Simulator, and will run it.

Mock

Principle

The HIL engine is a Java standard-based engine that runs Mocks. A Mock is a jar file containing some Java classes that simulate natives for the Simulator. Mocks allow applications to be run unchanged in the Virtual Device while still appearing to interact with native code.

Functional Description

As with *SNI*, HIL is responsible for finding the method to execute as a replacement for the native Java method that the MicroEJ Simulator tries to run. Following the SNI specification philosophy, the matching algorithm uses a naming convention. When a native method is called in the MicroEJ Simulator, it requests that the HIL engine execute it. The corresponding Mock executes the method and provides the result back to the MicroEJ Simulator.

Fig. 255: The MicroEJ Simulator Executes a Native Java Method foo() .

Example

```
package com.mycompany;

import java.io.IOException;

/**
 * Class providing a native method to access sensor value.
 * This method will be executed out of the Core Engine.
 */
public abstract class Sensor {

    public static final int ERROR = -1;

    public int getValue() throws IOException {
        int value = getSensorValue();
        if (value == ERROR) {
            throw new IOException("Unsupported sensor");
        }
        return value;
    }

    public static native int getSensorValue();
}
```

To implement the native method `getSensorValue()`, you need to create a MicroEJ mock project containing the same `Sensor` class on the same `com.mycompany` package.

To create a new MicroEJ mock project:

SDK 6

SDK 5

Follow the steps described in *SDK 6 User Guide - Create a Project* depending on your IDE.

- Select `File` > `New` > `Module Project`,
- Fill the module information (project name, module organization, name and revision),
- Select the `microej-mock` skeleton,
- Click on `Finish`.

The following code is the required `Sensor` class of the created Mock project:

```
package com.mycompany;

import java.util.Random;

/**
 * Java standard class included in a Mock jar file.
 * It implements the native method using a Java method.
 */
public class Sensor {

    /**
     * Constants
     */
```

(continues on next page)

(continued from previous page)

```
    private static final Random RANDOM = new Random();

    /**
     * Implementation of native method "getSensorValue()"
     *
     * @return Simulated sensor value
     */
    public static int getSensorValue() {
        // For the simulation, Mock returns a random value
        return RANDOM.nextInt();
    }

}
```

Note: The visibility of the native method implemented in the mock must be public regardless of the visibility of the native method in the application. Otherwise the following exception is raised: java.lang.UnsatisfiedLinkError: No such method in remote class .

Mocks Design Support

Interface

The MicroEJ Simulator interface is defined by static methods on the Java class com.is2t.hil.NativeInterface .

Array Type Arguments

Both *[SNI]* and HIL allow arguments that are arrays of base types. By default the contents of an array are NOT sent over to the Mock. An "empty copy" is sent by the HIL engine, and the contents of the array must be explicitly fetched by the Mock. The array within the Mock can be modified using a regular assignment. Then to apply these changes in the MicroEJ Simulator, the modifications must be flushed back. There are two methods provided to support fetch and flush between the MicroEJ Simulator and the HIL:

- refreshContent : initializes the array argument from the contents of its MicroEJ Simulator counterpart.
- flushContent : propagates (to the MicroEJ Simulator) the contents of the array that is used within the HIL engine.

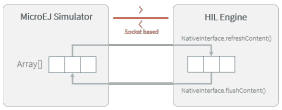


Fig. 256: An Array and Its Counterpart in the HIL Engine.

Below is a typical usage.

```
public static void foo(char[] chars, int offset, int length){
        NativeInterface ni = HIL.getInstance();
        //inside the Mock
        ni.refreshContent(chars, offset, length);
        chars[offset] = 'A';
        ni.flushContent(chars, offset, 1);
}
```

Blocking Native Methods

Some native methods block until an event has arrived *[SNI]*. Such behavior is implemented in native using the following three functions:

- int32_t SNI_suspendCurrentJavaThread(int64_t timeout)
- int32_t SNI_getCurrentJavaThreadID(void)
- int32_t SNI_resumeJavaThread(int32_t id)

This behavior is implemented in a Mock using the following methods on a lock object:

- Object.wait(long timeout): Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.
- Object.notifyAll(): Wakes up all the threads that are waiting on this object's monitor.
- NativeInterface.notifySuspendStart() : Notifies the Simulator that the current native is suspended so it can schedule a thread with a lower priority.
- NativeInterface.notifySuspendEnd() : Notifies the Simulator that the current native is no more suspended. Lower priority threads in the Simulator will not be scheduled anymore.

```
public static byte[] data = new byte[BUFFER_SIZE];
public static int dataLength = 0;
private static Object lock = new Object();

    // Mock native method
    public static void waitForData() {
            NativeInterface ni = HIL.getInstance();
            // inside the Mock
            // wait until the data is received
            synchronized (lock) {
                    while (dataLength == 0) {
                            try {
                                    ni.notifySuspendStart();
                                    lock.wait(); // equivalent to lock.wait(0)
                            } catch (InterruptedException e) {
                                    // Use the error code specific to your library
                                    throw new NativeException(-1, "InterruptedException", e);
                            } finally {
                                    ni.notifySuspendEnd();
                            }
                    }
            }
    }

// Mock data reader thread
public static void notifyDataReception() {
```

(continues on next page)

(continued from previous page)

```
    synchronized (lock) {
        dataLength = readFromInputStream(data);
        lock.notifyAll();
    }
}
```

Resource Management

Every Java class can play the role of a small read-only file system root: The stored files are called "Java resources" and are accessible using a path as a String.

The MicroEJ Simulator interface allows the retrieval of any resource from the original Managed world, using the `getResourceContent` method.

```
public static void bar(byte[] path, int offset, int length) {
  NativeInterface ni = HIL.getInstance();
  ni.refreshContent(path, offset, length);
  String pathStr = new String(path, offset, length);
  byte[] data = ni.getResourceContent(pathStr);
  ...
}
```

Note: By default the maximum HIL frame size in bytes is `262144`. If needed, the size can be increased by setting *com.microej.simulator.hil.frame.size* application option.

`com.is2t.hil.BrokenConnection` will be thrown by the HIL Engine if a frame is larger than the maximum HIL frame size (e.g. dealing with large Java resources).

Synchronous Terminations

To terminate the whole simulation (MicroEJ Simulator and HIL), use the `stop()` method.

```
public static void windowClosed() {
    HIL.getInstance().stop();
}
```

Define a Mock Option

Starting from *Architecture 8.3.0*, it is possible to define a mock option using the following *Application Option*:

```
microej.mock.property.mymockoption=mymockvalue
```

Then the option can be retrieved as a System Property in the mock:

```
public static void myNativeImplementation() {
    String myOption = System.getProperty("mymockoption"); // returns "mymockvalue"
}
```

Dependencies

A Mock project must include the Mock API module dependency for bi-directional communication with the Simulator via the HIL Engine.

SDK 6

SDK 5

Add a dependency to the Mock API in the `build.gradle.kts` file:

```
compileOnly("com.microej.tool.runtime:mock-api:2.5.0")
```

The Mock API is automatically provided by the `microej-mock` project skeleton.

Other dependencies to standard Java modules can be added, such as the *JavaFX Mock*.

Installation

In a VEE Port

SDK 6

SDK 5

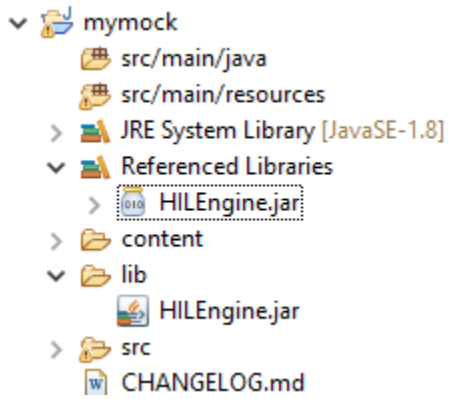
- *Create a Mock project*.
  - Add the Mock as a dependency of your VEE Port project:
  - either as a project dependency if both projects are in the same multi-project:

```
microejMock(project(":myMockProject"))
```

  - or as a module dependency:

```
microejMock("com.mycompany:my-mock:1.0.0")
```

First create a new *module project* using the `microej-mock` skeleton.



Once implemented, right-click on the repository project and select `Build Module`.

Once the module is built, the mock can be installed in a VEE Port in one of the two ways:

- by adding the mock module as a regular VEE Port *module dependency* (if your VEE Port configuration project contains a `module.ivy` file),
- or by manually copying the JAR file `[mock_project]/target~/rip/mocks/[mock_name].jar` to the *VEE Port configuration* mock dropins folder `dropins/mocks/dropins/` .

  Make sure the option *Resolve Foundation Libraries in Workspace* is enabled to use the mock without having to install it after each modification during development.

In an Application

You can also install a Mock from an Application project, for example when a native function is added directly in the Application project. When a Mock is declared in an Application, it is automatically added in the VEE Port used to build or run the Application.

SDK 6

SDK 5

> **Warning:** A Mock can be installed from an SDK 6 Application project only if the VEE Port used in the project is an SDK 6 VEE Port.

- add the Mock as a dependency of your Application project:
  - either as a project dependency if both projects are in the same multi-project:

```
microejMock(project(":myMockProject"))
```

  - or as a module dependency:

```
microejMock("com.mycompany:my-mock:1.0.0")
```

Installing a Mock from an Application project is not supported in SDK 5.

Use

Once installed, a Mock is used automatically by the Simulator when the MicroEJ Application calls a native method which is implemented into the Mock.

JavaFX

JavaFX is an open-source library for creating modern Java user interfaces that is highly portable. It can be used to quickly create graphical Mocks for your VEE Port.

The installation instructions depend on the SDK version:

SDK 6

SDK 5

- Add JavaFX as a compile-time dependency in your Mock project:

```
compileOnly(group=
↪"com.microej.tool", name="javafx", version="1.2.0", configuration="provided")
```

- If your VEE Port contains at least one Mock, add JavaFX as a Mock dependency in your VEE Port project:

```
microejMock("com.microej.tool:javafx:1.2.0")
```

- If your SDK is running on JDK 8, the Oracle JDK contains JavaFX, so this version allows you to use it right now in your project.
- If your SDK is running on JDK 11, JavaFX must be added as an additional dependency to your Mock and VEE Port project. For that, MicroEJ Corp. provides a ready-to-use packaged module for all supported OS versions.

```
<dependency org="com.microej.tool" name="javafx" rev="1.2.0" />
```

The Module serves two purposes, depending on whether it is added to a Mock or a VEE Port project:

- In a Mock project, JavaFX is added as a compile-time dependency, its content is not included in the Mock.
- If your VEE Port contains at least one Mock, JavaFX must be added to the VEE Port project in order to embed its content in the VEE Port.

Warning: There is a known issue with JavaFX and Apple Silicon computers. The task runOnSimulator fails with an error

```
libc++abi: terminating due to uncaught exception of type NSException
Exiting /Users/microej/
↪Git/J0059_Example-Mock-Framework/custom-widgets-app/build/vee/scripts/hil.xml.
Exception in thread "thread2" java.lang.UnsatisfiedLinkError: Broken connection with client
        at java.lang.Throwable.fillInStackTrace(Throwable.java:82)
        at java.lang.Throwable.<init>(Throwable.java:37)
        at java.lang.Error.<init>(Error.java:18)
        at java.lang.LinkageError.<init>(LinkageError.java:18)
        at java.lang.UnsatisfiedLinkError.<init>(UnsatisfiedLinkError.java:10)
        at com.microej.example.mockframework.Main$1.run(Main.java:45)
        at com.is2t.bon.timer.TimerTaskList.runLaunchedTasks(TimerTaskList.java:237)
        at ej.bon.Timer.run(Timer.java:431)
        at java.lang.Thread.run(Thread.java:325)
        at java.lang.Thread.runWrapper(Thread.java:387)
```

This issue affects most JDK distributions. As a workaround, we recommend to use one of the following Eclipse Temurin distributions: 17.0.9 or 21.0.1 to avoid this issue.

You need to make your MICROEJ SDK installation point to one of the decompressed JDK archives given above.

Note that newer versions of these JDKs (17.0.10 or higher, and 21.0.2 or higher) will have this issue. Installation from the .pkg distribution might be overwritten by a newer version.

Mock Framework

The Mock Framework is a library based on JavaFX, it aims to ease the development of mock UIs.

The Mock Framework provides a set of widgets. It allows to automatically generate the native method implementation of an application and link it with the widgets of the mock UI.

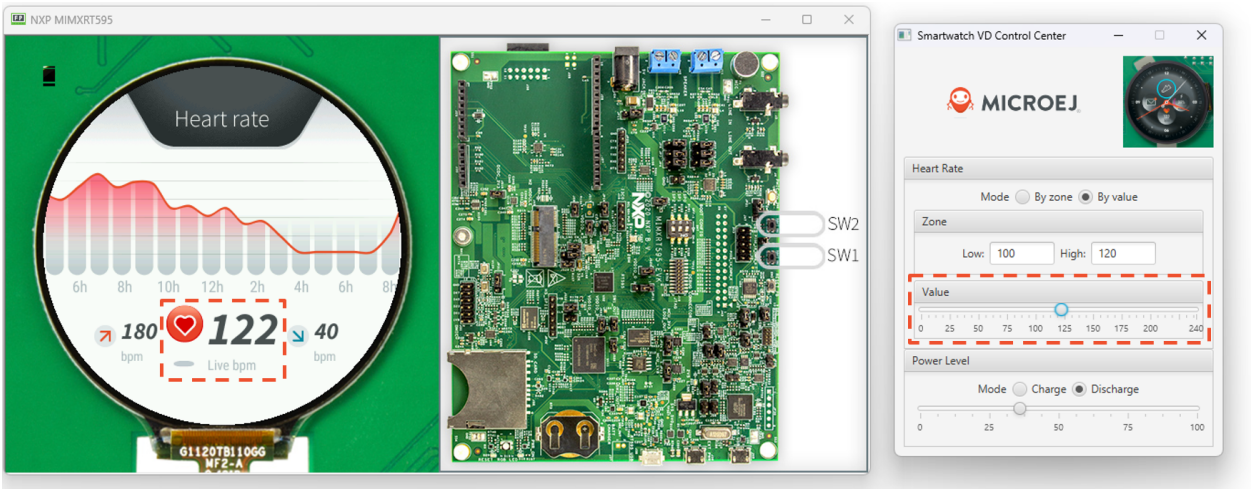


Fig. 257: Mock Framework used to mock Heart Rate sensor on Wearable Demo.

Usage

The following steps should be followed to create a mock using the Mock Framework:

- Create Mock Framework Properties to bind the native methods to the mock UI,
- Create Widgets to manipulate the values of the above Properties,
- Create a Dashboard to hold the Widgets.

Mock Framework Property

The Mock Framework uses a property system to bind widgets to the native methods. A property holds a value and can trigger listeners when updated.

A property must extend `MockProperty` and be annotated with `@Property` :

```
@Property
public class MyProperty extends MockProperty {
    ...
}
```

The annotation is used by the framework to find any property declared in the mock project. The property can then be retrieved from its class:

```
MyProperty property = MockupApplication.getProperty(MyProperty.class);
```

There are ready to use implementations of `MockProperty` :

- `BooleanProperty`

- IntegerProperty
- LongProperty
- FloatProperty
- DoubleProperty
- NumberProperty
- StringProperty
- FileProperty

Getter and Setter Attributes

Let's consider the following application code that defines getter and setter native methods:

```
package com.microej.example;

public class RandomService {

   private RandomService() {
   }

   /**
   * Gets the service state.
   *
   * @return true if the service is enabled, false otherwise.
   */
   public static native boolean getEnable();

   /**
   * Sets the service state.
   *
   * @param enable
   *           enables or disables the service.
   */
   public static native void setEnable(boolean enable);

   ...
```

The native method implementation code can be generated using the following attributes in the Property annotation:

- getter="<method name>" for the native method that retrieves a value from the mock.
- setter="<method name>" for the native method that sets a value in the mock.

```
@Property(getter = "com.microej.example.RandomService.getEnable")
public class MyProperty extends BooleanProperty {
   ...
}
```

or

```
@Property(getter = "com.microej.example.
↪RandomService.getEnable", setter = "com.microej.example.RandomService.setEnable")
public class MyProperty extends BooleanProperty {
```

(continues on next page)

(continued from previous page)

```
    ...
}
```

These attributes are optional. When no attribute is specified, the corresponding code will not be generated.

The <method name> is the fully qualified name of the method, it must contain the package, the name of the class in which the native is implemented, and the native method name. It must not contain parenthesis and arguments.

Note that the class containing the getter and the setter can be different.

Property values can be changed from the mock code with getValue() and setValue() methods:

```
/* Get MyProperty value */
boolean state = MockupApplication.getProperty(MyProperty.class).getValue();

/* Set MyProperty value */
MockupApplication.getProperty(MyProperty.class).setValue(!state);
```

Mock Framework Widgets

The Mock Framework provides some widgets to manipulate properties.

Interacting with the widget modifies the underlying property, and similarly, updating the property value modifies the state of the widget.

- CheckBox : sets the value of a BooleanProperty . The property is set to true when the box is checked, and false otherwise.
- NumberSlider : sets a value of a NumberProperty between the bounds defined by the property. The bounds and the position of the slider are automatically updated with the property.
- BoundsSetter : displays the upper and lower bounds defined in the NumberProperty . This class is abstract and needs to be used as a specialized subclass, such as IntegerBoundsSetter for integer values.
- Container : widget that contains other Mock Framework Widgets. By default, it displays contained widgets vertically.
- Choice : widget container providing a radio button list for each contained widget. Only the selected widget will be enabled and the other disabled (not clickable).
- FileChooser : displays a button that opens a standard platform file dialog for selecting a file. Once the file is selected, the property is updated with the corresponding file object.
- TitledWidget : decorator to add a title to a widget.
- ImageSwapWidget : stores two images and shows one of them based on a boolean property.
- JavaFxWidget : abstract class that can be extended to create custom widgets using JavaFX components.

Mock Framework Dashboard

The Mock Framework Dashboard represents the window that is opened at Application startup on Simulator. It holds the Mock widgets.

Mock widgets can be bound to Mock properties by passing the property class as an argument of the Mock widget.

A Dashboard must extend `AbstractDashboard` and be annotated with `@DashBoard` annotation.

```
@DashBoard(title = "My Mock DashBoard")
public class MockDashBoard extends AbstractDashboard {

  public MockDashBoard() {
    addWidget(new CheckBox("Enable RandomService", MyProperty.class));
  }
}
```

`@DashBoard` attributes are optional, find below the list of available ones:

- `title` : sets the title of the mock window, default title is *VD Control Center*.
- `icon` : sets the icon of the mock window, the path is relative to the `src/main/resources` folder of the mock project (e.g. `icon="images/myIcon.png"`).
- `width` : sets the width of the mock window. The default value is negative. When not specified, the system sets the mock window size automatically.
- `height` : sets the height of the mock window. The default value is negative. When not specified, the system sets the mock window size automatically.

Examples

- Mock Framework Examples demonstrate the use of the Mock Framework.

Installation

SDK 6

SDK 5

- Add the JavaFX dependency to your VEE Port project (see *JavaFX* for more details):

```
microejMock("com.microej.tool:javafx:1.2.0")
```

- Add the Mock Framework and JavaFX libraries to your Mock project dependencies:

```
implementation("com.microej.library.mock:mock-framework:1.0.1")
compileOnly(group=
↪"com.microej.tool", name="javafx", version="1.2.0", configuration="provided")
```

- Add the Mock Framework and JavaFX annotation processors libraries to your Mock project dependencies:

```
annotationProcessor("com.microej.library.mock:mock-framework:1.0.1")
annotationProcessor(group=
↪"com.microej.tool", name="javafx", version="1.2.0", configuration="provided")
```

- Add the JavaFX dependency to your VEE Port project if required (see *JavaFX* for more details):

```
<dependency org="com.microej.tool" name="javafx" rev="1.2.0" />
```

- Add the Mock Framework and JavaFX libraries to your Mock project dependencies:

```
<dependency org="com.microej.library.mock" name="mock-framework" rev="1.0.1" />
<dependency org="com.microej.tool" name="javafx" rev="1.2.0" />
```

Event Tracing

Starting from *Architecture 8.4.0*, it is possible to record events from the Mock. The *Mock API* provides the same API (ej.trace.Tracer) as the one used for *Application Event Tracing*:

```
Tracer tracer = new Tracer("MyMockGroup", 10);
tracer.recordEvent(1);
tracer.recordEvent(2);
tracer.recordEvent(3);
```

Events are recorded only if *Event Recording is enabled*.

By default, the Simulator traces are printed on the standard console.

```
[TRACE: MyMockGroup] Event 0x1()
[TRACE: MyMockGroup] Event 0x2()
[TRACE: MyMockGroup] Event 0x3()
```

However, it is also possible to connect other trace recorders as needed. In particular, you can export traces in the *.SVdat format for analysis with the *Segger SystemView* tool. For more information, please contact our support team.

Shielded Plug Mock

General Architecture

The Shielded Plug Mock simulates a Shielded Plug *[SP]* on desktop computer. This mock can be accessed from the MicroEJ Simulator, the hardware platform or a Java SE application.

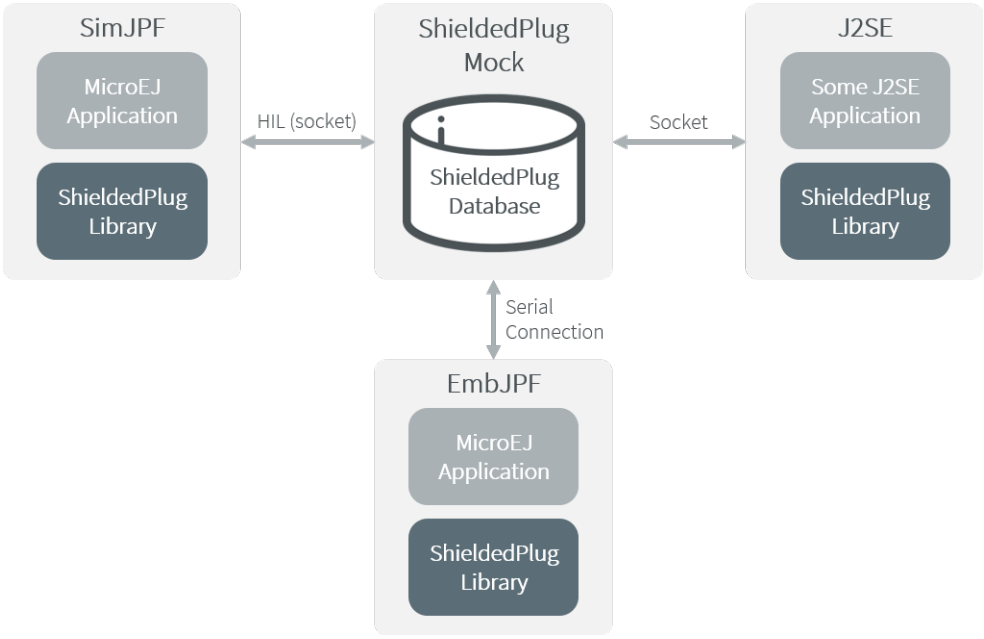


Fig. 258: Shielded Plug Mock General Architecture

Configuration

The mock socket port can be customized for Java SE clients, even though several Shielded Plug mocks with the same socket port cannot run at the same time. The default socket port is 10082.

The Shielded Plug mock is a standard MicroEJ Application. It can be configured using Java properties:

- sp.connection.address
- sp.connection.port

Front Panel Mock

Principle

A major strength of the MicroEJ environment is that it allows applications to be developed and tested in a Simulator rather than on the target device, which might not yet be built. To make this possible for devices that controls operated by the user, the Simulator must connect to a "mock" of the control panel (the "Front Panel") of the device. The Front Panel generates a graphical representation of the device, and is displayed in a window on the user's development machine when the application is executed in the Simulator.

The Front Panel has been designed to be an implementation of MicroUI library (see *Simulation*). However it can be use to show a hardware device, blink a LED, interact with user without using MicroUI library.

Functional Description

1. Creates a new Front Panel project.
2. Creates an image of the required Front Panel. This could be a photograph or a drawing.
3. Defines the contents and layout of the Front Panel by editing an XML file (called an fp file). Full details about the structure and contents of fp files can be found in chapter *Front Panel*.
4. Creates images to animate the operation of the controls (for example button down image).
5. Creates Front Panel *Widgets* that make the link between the application and the user interactions.
6. Previews the Front Panel to check the layout of controls and the events they create, etc.
7. Exports the Front Panel project into a MicroEJ VEE Port project.

The Front Panel Project

Creating a Front Panel Project

SDK 6

SDK 5

A Front Panel project template is available as part of the VEE Port project template. Clone this template to create a Front Panel project.

A Front Panel project is created using the New Front Panel Project wizard. Select:

New > Project… > MicroEJ > Front Panel Project

The wizard will appear:

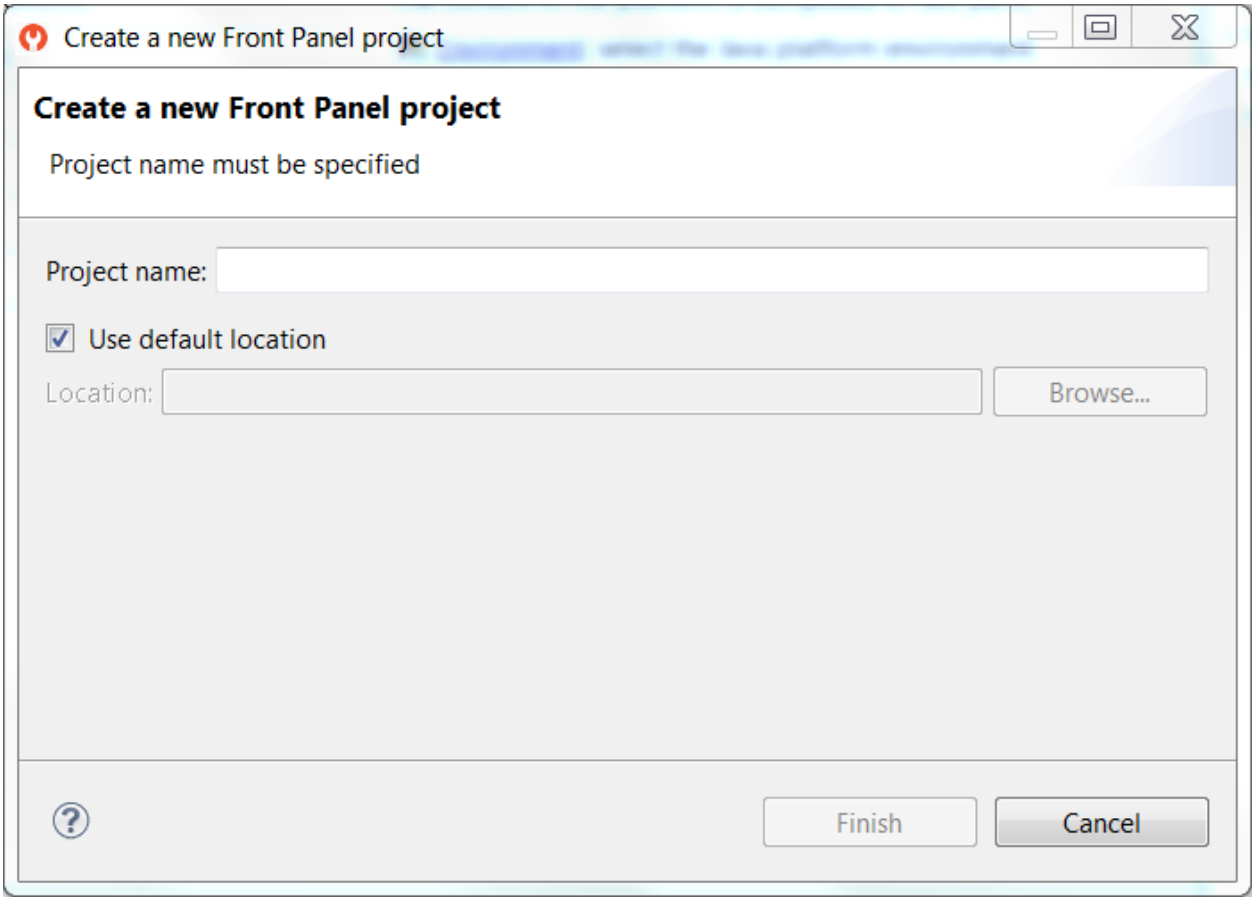


Fig. 259: New Front Panel Project Wizard

Enter the name for the new project.

Project Contents

SDK 6

SDK 5

A Front Panel project has the following structure and contents:

- The src/main/java folder is provided for the definition of Front Panel Widgets. It is initially empty. The creation of these classes will be explained later.
- The src/main/resources folder holds the file or files that define the contents and layout of the Front Panel, with a .fp extension (the fp file or files), plus images used to create the Front Panel. A newly created project will have a single fp file with the same name as the project, as shown above. The contents of fp files are detailed later in this document.
- The build.gradle.kts file is the project build descriptor.

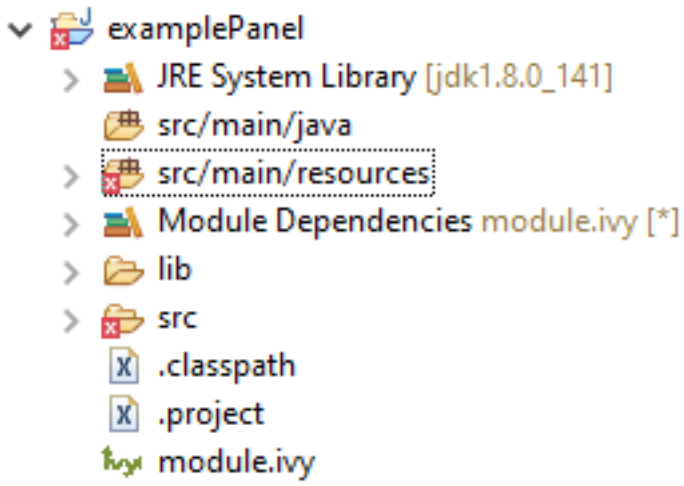


Fig. 260: Project Contents

A Front Panel project has the following structure and contents:

- The src/main/java folder is provided for the definition of Front Panel Widgets. It is initially empty. The creation of these classes will be explained later.
- The src/main/resources folder holds the file or files that define the contents and layout of the Front Panel, with a .fp extension (the fp file or files), plus images used to create the Front Panel. A newly created project will have a single fp file with the same name as the project, as shown above. The contents of fp files are detailed later in this document.
- The JRE System Library is referenced, because a Front Panel project needs to support the writing of Java for the Listeners (and DisplayExtensions).
- The Modules Dependencies contains the libraries for the Front Panel simulation, the widgets it supports and the types needed to implement Listeners (and DisplayExtensions).
- The lib contains a local copy of Modules Dependencies.

Module Dependencies

SDK 6

SDK 5

The Front Panel project is a regular Gradle project. Its build.gradle.kts file should look like this example:

```
plugins {
  id("com.microej.gradle.mock-frontpanel")
}

group = "com.mycompany"
version = "0.1.0-RC"

dependencies {
  implementation("ej.tool.frontpanel:framework:1.1.1")
}
```

The Front Panel project is a regular MicroEJ Module project. Its `module.ivy` file should look like this example:

```
<ivy-module version="2.0" xmlns:ea=
↪"http://www.easyant.org" xmlns:ej="https://developer.microej.com" ej:version="2.0.0">
  <info organisation=
↪"com.mycompany" module="examplePanel" status="integration" revision="1.0.0"/>

  <configurations defaultconfmapping="default->default;provided->provided">
    <conf name=
↪"default" visibility="public" description="Runtime dependencies to other artifacts"/>
    <conf name="provided" visibility=
↪"public" description="Compile-time dependencies to APIs provided by the platform"/>
  </configurations>

  <dependencies>
    <dependency org="ej.tool.frontpanel" name="framework" rev="1.1.1"/>
  </dependencies>
</ivy-module>
```

The Front Panel Framework contains the Front Panel core classes, mainly the ability to create your own Front Panel *Widget* to simulate user interactions.

Note: Some Front Panel Widgets are available to interact with the MicroUI devices (display, input devices, etc.), see *Simulation*.

Front Panel File

File Content

The Front Panel engine takes an XML file (the `.fp` file) as input. It describes the panel using widgets: they simulate the drivers, sensors and actuators of the real device. The Front Panel engine generates the graphical representation of the real device, and is displayed in a window on the user's development machine when the application is executed in the Simulator.

The following example file describes a simple board with one LED:

```
<?xml version="1.0"?>
<frontpanel
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="https://developer.microej.com"
  xsi:schemaLocation="https://developer.microej.com .widget.xsd">

  <device name="MyBoard" skin="myboard.png">
    <ej.fp.widget.LED x="131" y="127" skin="box_led.png"/>
  </device>
</frontpanel>
```

The `device` `skin` must refer to a `png` file in the `src/main/resources` folder. This image is used to render the background of the Front Panel. The widgets are drawn on top of this background.

The `device` contains the elements that define the widgets that make up the Front Panel. The name of the widget element defines the type of widget. The set of valid types is determined by the Front Panel Designer. Every widget element defines a `label`, which must be unique for widgets of this type (optional or not), and the `x` and `y` coordinates of the position of the widget

within the Front Panel (0,0 is top left). There may be other attributes depending on the type of the widget.

The file and tags specifications are available in chapter *Front Panel*.

Note: The .fp file grammar has changed since the UI Pack version 12.0.0 (Front Panel core has been moved to MicroEJ Architecture starting from version 7.11.0). A quick migration guide is available: open VEE Port configuration file .Platform, go to Content tab, click on module Front Panel. The migration guide is available in Details box.

Editing Front Panel Files

The .fp file is a standard XML file, so it can be edited with any XML Editor. For example in Eclipse, you can open it using the Eclipse XML editor (right-click on the .fp file, select Open With > XML Editor). This editor features syntax highlighting and checking, and content-assist based on the schema (XSD file) referenced in the fp file. This schema is a hidden file within the project's definitions folder. An incremental builder checks the contents of the fp file each time it is saved and highlights problems in the Eclipse Problems view, and with markers on the fp file itself.

A preview of the Front Panel can be obtained by opening the Front Panel Preview. This tool is available in Eclipse only:

- in SDK 5, it is installed by default.
- in SDK 6, it must be installed by following the instructions on the *Development Tools* page.

  Once installed, open it in Window > Show View > Other... > MicroEJ > Front Panel Preview.

  The preview is updated each time the .fp file is saved.

  A typical working layout is shown below.

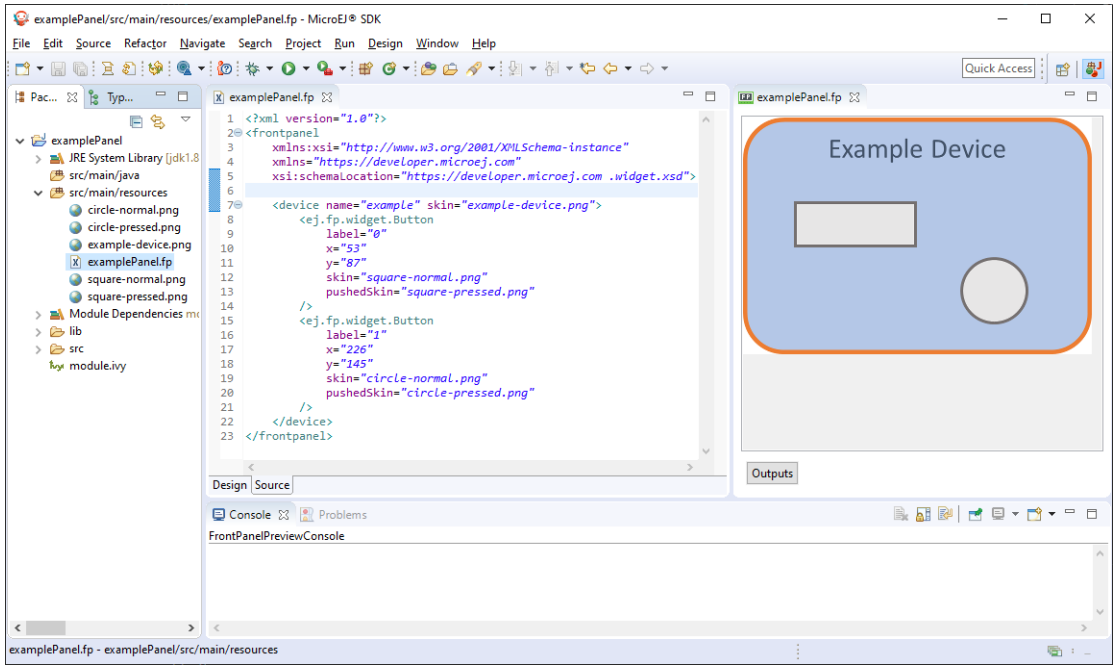


Fig. 261: Working Layout Example

Within the XML editor, content-assist is obtained by pressing CTRL + SPACE keys. The editor will list all the elements valid at the cursor position, and insert a template for the selected element.

Multiple Front Panel Files

A Front Panel project can contain multiple .fp files. All those files are compiled when exporting the Front Panel project to a VEE Port (or during VEE Port build). It may be useful to have two or more representations of a board (skin, device layout, display size, etc…). When running the simulator, by default, the .fp file declared by the *VEE Port configuration*, is used (or a random one if no default is configured). To pick a specific one, set the *Application Option* frontpanel.file to a Front Panel simple file name included in the VEE Port (e.g. myproduct.fp).

Widget

Description

A widget is a subclass of Front Panel Framework class ej.fp.Widget . The library ej.tool.frontpanel#widget provides a set of widgets which are Graphics Engine compatible (see *Simulation*). To create a new widget (or a subclass of an existing widget), have a look on available widgets in this library.

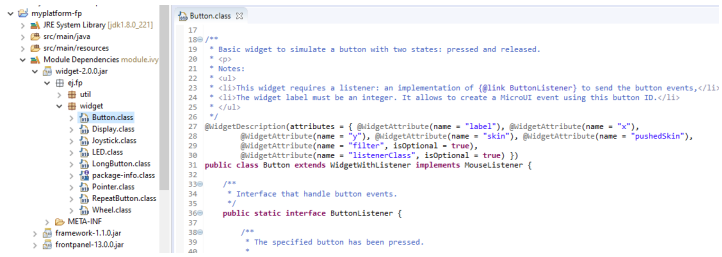


Fig. 262: Front Panel Widgets

A widget is recognized by the fp file as soon as its class contains a @WidgetDescription annotation. The annotation contains several @WidgetAttribute . An attribute has got a name and tells if it is an optional attribute of widget (by default an attribute is mandatory).

This is the description of the widget LED :

```
@WidgetDescription(attributes = { @WidgetAttribute(name = "x"),
    @WidgetAttribute(name = "y"), @WidgetAttribute(name = "skin")})
```

As soon as a widget is created (with its description) in Front Panel project, the fp file can use it. Close and reopen fp file after creating a new widget. In device group, press CTRL + SPACE keys to visualize the available widgets: the new widget can be added.

```
<ej.fp.widget.LED x="170" y="753" skin="box_led.png" />
```

Each attribute requires the set methods in the widget source code. For instance, the widget LED (or its hierarchy) contains the following methods for sure:

- setX(int) ,
- setY(int) ,
- setskin(Image) .

The set method parameter's type fixes the expected value in fp file. If the attribute cannot match the expected type, an error is throw when editing fp file. Widget master class already provides a set of standard attributes:

- setFilter(Image) : apply a filtering image which allows to crop input area (*Input Device Filters*).
- setWidth(int) and setHeight(int) : limits the widget size.
- setLabel(String) : specifies an identifier to the widget.
- setOverlay(boolean) : draws widget skin with transparency or not.
- setSkin(Image) : specifies the widget skin.
- setX(int) and setY(int) : specifies widget position.

Notes:

- Widget class does not specify if an attribute is optional or not. It it the responsability to the subclass.
- The label is often used as identifier. It also allows to retrieve a widget calling Device.getDevice().getWidget(Class<T>, String) . Some widgets are using this identifier as an integer label. It is the responsability to the widget to fix the signification of the label.

- The widget size is often fixed by the its skin (which is an image). See Widget.finalizeConfiguration() : it sets the widget size according the skin if the skin has been set; even if methods setWidth() and setHeight() have been called before.

Runtime

The Front Panel engine parsing the fp file at application runtime. The widget methods are called in two times. First, engine creates widget by widget:

1. widget's constructor: Widget should initialize its own fields which not depend on widget attributes (not valorized yet).
2. setXXX() : Widget should check if given attribute value matches the expected behavior (the type has been already checked by caller). For instance if a width is not negative. On error, implementation can throw an IllegalArgumentException . These checks must not depend on other attributes because they may have not already valorized.
3. finalizeConfiguration() : Widget should check the coherence between all attributes: they are now valorized.

   During these three calls, all widgets are not created yet. And so, by definition, the main device (which is a widget) not more. By consequence, the implementation must not try to get the instance of device by calling Device.getDevice() . Furthermore, a widget cannot try to get another widget by calling Device.getDevice().getWidget(s) . If a widget depend on another widget for any reason, the last checks can be performed in start() method. This method is called when all widgets and main device are created. Call to Device.getDevice() is allowed.

   The method showYourself() is only useful when visualizing the fp file during its editing (use Eclipse view Front Panel Preview). This method is called when clicking on button Outputs .

Example

The following code is a simple widget LED. MicroEJ Application can interact with it using native methods on() and off() of class ej.fp.widget.LED :

```
package ej.fp.widget;

import ej.fp.Device;
import ej.fp.Image;
import ej.fp.Widget;
import ej.fp.Widget.WidgetAttribute;
import ej.fp.Widget.WidgetDescription;

/**
* Widget LED declaration. This class must have the same package than
* <code>LED</code> in MicroEJ application. This is required by the simulator to
* retrieve the implementation of native methods.
*/
@WidgetDescription(attributes␣
↪= { @WidgetAttribute(name = "x"), @WidgetAttribute(name = "y"),
     @WidgetAttribute(name = "skin") })
public class LED extends Widget {

   boolean on; // false init
```

(continues on next page)

(continued from previous page)

```
  /**
  * Called by the plugin when clicking on <code>Outputs</code> button from Front
  * Panel Preview.
  */
  @Override
  public void showYourself(boolean appearSwitchedOn) {
    update(appearSwitchedOn);
  }

  /**
  * Called by framework to render the LED.
  */
  @Override
  public Image getCurrentSkin() {
    // when LED is off, hide its skin returning null
    return on ? getSkin() : null;
  }

  /**
  * MicroEJ application native
  */
  public static void on() {
    update(true);
  }

  /**
  * MicroEJ application native
  */
  public static void off() {
    update(false);
  }

  private static void update(boolean on) {

    // retrieve the LED (there is only one LED on device)
    LED led = Device.getDevice().getWidget(LED.class);

    // update its state
    led.on = on;

    // ask to repaint it
    led.repaint();
  }
}
```

Empty Widget

By definition a widget may not contain an attribute. This kind of widget is useful to perform something at Front Panel startup, for instance to start a thread to pick up data somewhere.

The widget description is `@WidgetDescription(attributes = { })`. In `start()` method, a custom behavior can be performed. In `fp` file, the widget declaration is `<com.mycompany.Init/>` (where `Init` is an example of widget name).

Input Device Filters

The widgets which simulate the input devices use images (or “skins”) to show their current states (pressed and released). The user can change the state of the widget by clicking anywhere on the skin: it is the active area. This active area is, by default, rectangular.

These skins can be associated with an additional image called a filter. This image defines the widget’s active area. It is useful when the widget is not rectangular.

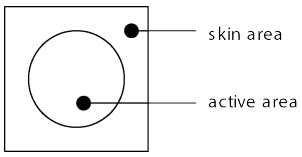


Fig. 263: Active Area

The filter image must have the same size as the skin image. The active area is delimited by the fully opaque pixels. Every pixel in the filter image which is not fully opaque is considered not part of the active area.

Extension / Customization

Since UI Pack 14.3.0 it is also possible to extend the Front Panel window.

There are several zones available for extension: the tool bar, the status bar and the sides of the device. It may be convenient for example to concentrate all the mocks into one window instead of several ones.

The device widget can be retrieved using FrontPanel.getFrontPanel().getDeviceWidget(). From this widget, it is possible to retrieve the extensible zones.

Tool Bar

The tool bar can be retrieved with (javax.swing.JToolBar) ((javax.swing.JComponent) FrontPanel.getFrontPanel().getDeviceWidget()).getParent().getParent().getComponent(1). Then it is possible to add one or several actions using JToolBar.add(). A good practice is to add a separator before adding the actions using JToolBar.addSeparator().

For example:

```
JToolBar toolBar = (JToolBar) ((JComponent)␣
↪FrontPanel.getFrontPanel().getDeviceWidget()).getParent().getParent().getComponent(1);
ImageIcon␣
↪myActionIcon = new ImageIcon(getClass().getClassLoader().getResource("myIcon.png"));
AbstractAction myAction = new AbstractAction(null, myActionIcon) {
  private static final long serialVersionUID = 1L;

  @Override
  public void actionPerformed(ActionEvent e) {
    // Execute my action.
  }
};
```

(continues on next page)

(continued from previous page)

```
toolBar.addSeparator();
toolBar.add(myAction);
```

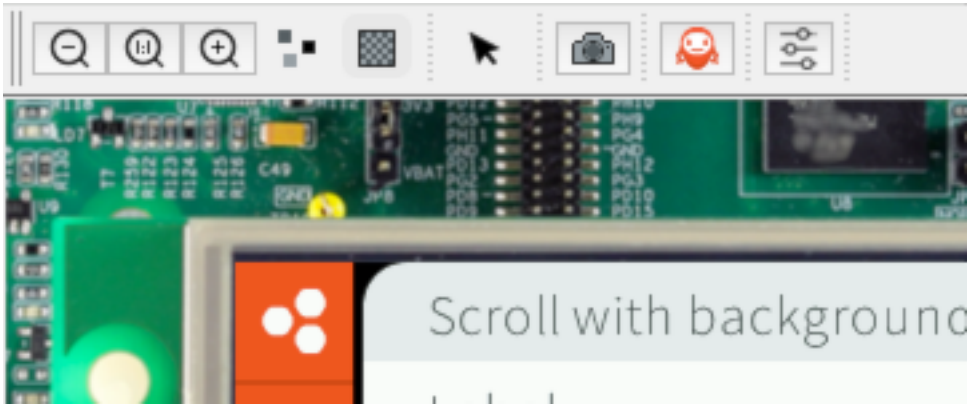


Fig. 264: An example of action in the toolbar with the MicroEJ mascot as icon.

Status Bar

The status bar can be retrieved with (javax.swing.JPanel) ((javax.swing.JComponent) FrontPanel.getFrontPanel().getDeviceWidget()).getParent().getParent().getComponent(2) .

The status bar contains 3 zones: one on the left, one on the center, one on the right. Each zone is a component included in the status bar and can be retrieved with its position, respectively 0 , 1 and 2 . In each zone, it is possible to add one or several components. A good practice is to add a separator before adding the components.

For example:

```
JPanel statusBar = (JPanel) ((JComponent)␣
↪FrontPanel.getFrontPanel().getDeviceWidget()).getParent().getParent().getComponent(2);
JPanel statusBarLeft = (JPanel) statusBar.getComponent(0);
JLabel myLabel = new JLabel("My information");
JSeparator mySeparator = new JSeparator(SwingConstants.VERTICAL);
Dimension myLabelPreferredSize = mySeparator.getPreferredSize();
myLabelPreferredSize.height = myLabel.getPreferredSize().height;
mySeparator.setPreferredSize(myLabelPreferredSize);
statusBarLeft.add(mySeparator);
statusBarLeft.add(myLabel);
```

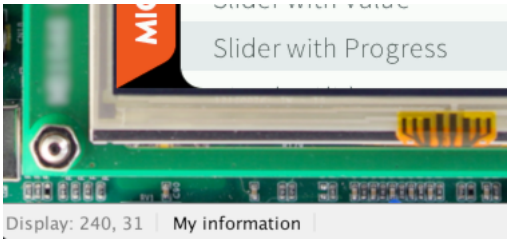


Fig. 265: An example of simple information in the status bar.

Around the Device

The panel containing the device can be retrieved with (java.swing.JPanel) ((javax.swing.JComponent) FrontPanel.getFrontPanel().getDeviceWidget()).getParent()

The panel uses a border layout, the device being in the center. It is possible to add widgets around the device, on north, west, east or south.

For example:

```
JPanel panel
↪= (JPanel) ((JComponent) FrontPanel.getFrontPanel().getDeviceWidget()).getParent();
panel.add(new JLabel("My left label"), BorderLayout.WEST);
panel.add(new JLabel("My top label"), BorderLayout.NORTH);
panel.add(new JButton("My bottom button"), BorderLayout.SOUTH);
JPanel myPanel = new JPanel();
myPanel.setLayout(new BoxLayout(myPanel, BoxLayout.Y_AXIS));
panel.add(myPanel, BorderLayout.EAST);
myPanel.add(new JLabel("My controls"));
myPanel.add(new JButton("My first control"));

// Update the window to be resized to fit the new widgets.
JFrame frame = (JFrame) panel.getParent().getParent().getParent().getParent();
frame.pack();
```

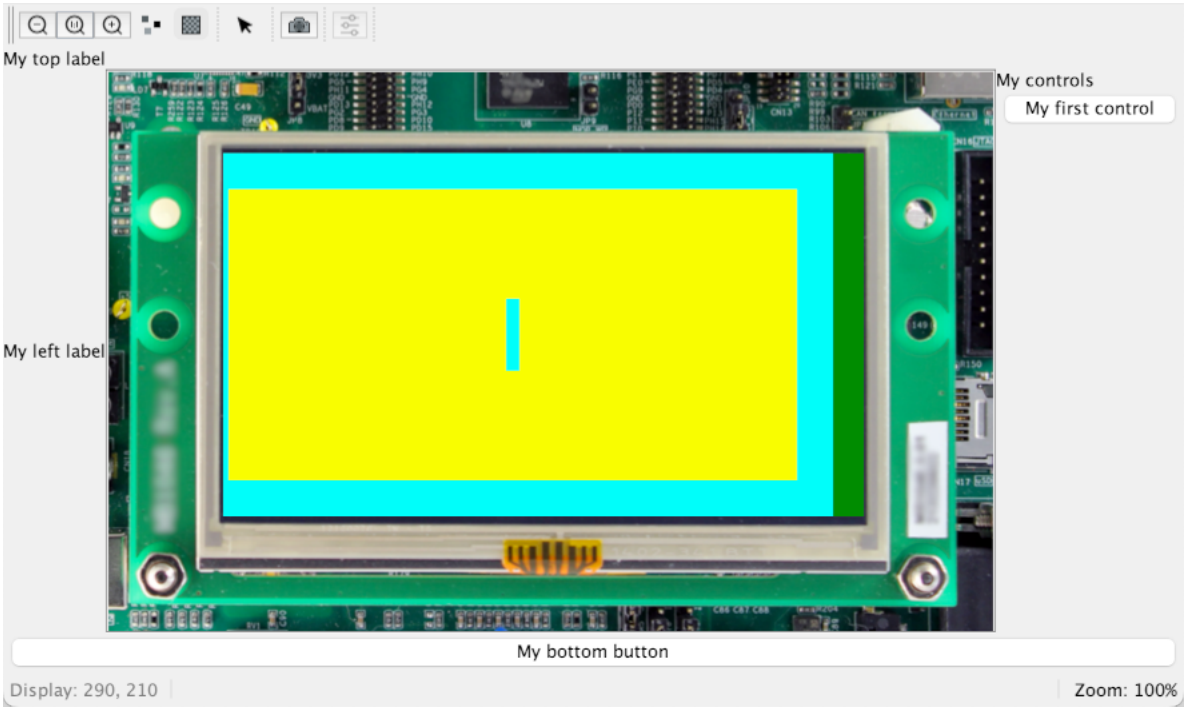


Fig. 266: An example of widgets around the device.

Installation

In the *VEE Port configuration* file, check Front Panel to install the Front Panel module. When checked, the properties file frontpanel/frontpanel.properties is required during VEE Port creation to configure the module. This configuration step is used to identify and configure the Front Panel.

The properties file must / can contain the following properties:

- project.name [mandatory]: Defines the name of the Front Panel project (same workspace as the VEE Port configuration project). If the project name does not exist, a new project will be created.
- fpFile.name [optional, default value is "" (*empty*)]: Defines the Front Panel file (*.fp) the application has to use by default when several fp files are available in project.

Advanced: Test the Front Panel Project

Note: Starting from SDK 5.7.0 and *Architecture 8.0.0*, the Front Panel projects are automatically resolved in the workspace, so this section and the property ej.fp.project are obsolete since. See *Resolve Foundation Libraries in Workspace* for more details.

If the Front Panel project has been created with a SDK version lower than 5.7.0, a project option must be updated:

- right-click on the Module Dependencies entry.
- click on Properties .
- go to the Classpath tab.
- check the Resolve dependencies in workspace option.

To quickly test a Front Panel project without rebuilding the VEE Port or manually exporting the project, add the *Application Option* ej.fp.project to the absolute path of a Front Panel project (e.g. c:\mycompany\myfrontpanel-fp). The Simulator will use the Front Panel project specified instead of the one included in the VEE Port. This feature is useful for locally testing some changes in the Front Panel project.

```
-Dej.fp.project=${project_loc:myfrontpanel-fp}
```

Warning: This feature only works if the VEE Port has been built with the Front Panel module enabled and the VEE Port does not contain the changes until a new VEE Port is built.

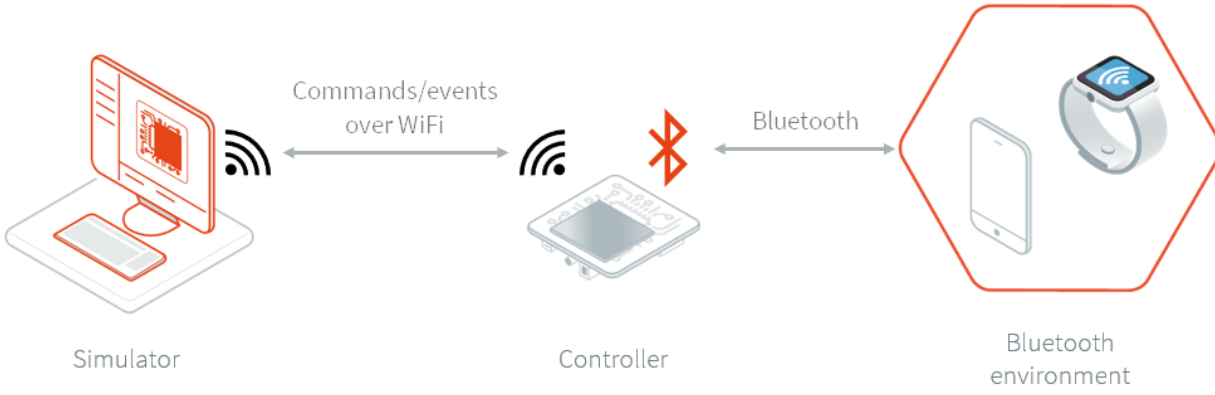
Use

Launch an application on the Simulator to run the Front Panel.

Bluetooth Mock

Overview

To run a MicroEJ Application that uses the Bluetooth API Library on MicroEJ Simulator, a Bluetooth Controller is required:



The Bluetooth Controller is a dedicated device that communicates with the Simulator to perform the Bluetooth operations of the Application. This means that the Simulator delegates the Bluetooth operations (scanning, advertising, discovering services, connecting, etc.) to a real Bluetooth device. This allows testing a Bluetooth application on any development environment.

The Bluetooth Controller firmware is provided for the inexpensive ESP32-S3-DevKitC-1-N8R8 board.

Requirements

- An ESP32-S3-DevKitC-1-N8R8 board.
- A Bluetooth Controller firmware (see section below).
- The Espressif Flash Download Tools.

Controller Firmwares

There are multiple versions of the Controller firmware, each compatible with a specific range of Bluetooth Pack versions.

The following table indicates the firmware to use depending on the Bluetooth Pack version integrated in the VEE Port:

Bluetooth Pack Versions	Controller Firmware
[2.0.0-2.3.0[	1.0.0
[2.3.0-2.6.0[	1.1.1
[2.6.0-3.0.0[	2.0.0

Usage

To simulate a Bluetooth application, follow these three steps:

- Install the controller firmware
- Configure the Wi-Fi network
- Run the application on the Simulator

If you are facing any issues, check the *Troubleshooting* section.

Controller Firmware Installation

To install the controller firmware on the ESP32-S3 board, follow these steps:

- Plug in the ESP32-S3 board to your computer,
- Find the associated COM port,
- In the flash download tool:
  - select the `ESP32-S3` chip type
  - browse the firmware file (`Executable-Bluetooth-Mock-Controller-ESP32-S3-x.y.z.bin`)
  - set the target address to `0`
  - set the SPI speed to `80MHz` and the SPI mode to `DIO`
  - select the appopriate COM port and set the baudrate to `460800`
  - start the flash download

With the flash download tool from Espressif, you should end with something similar to this :

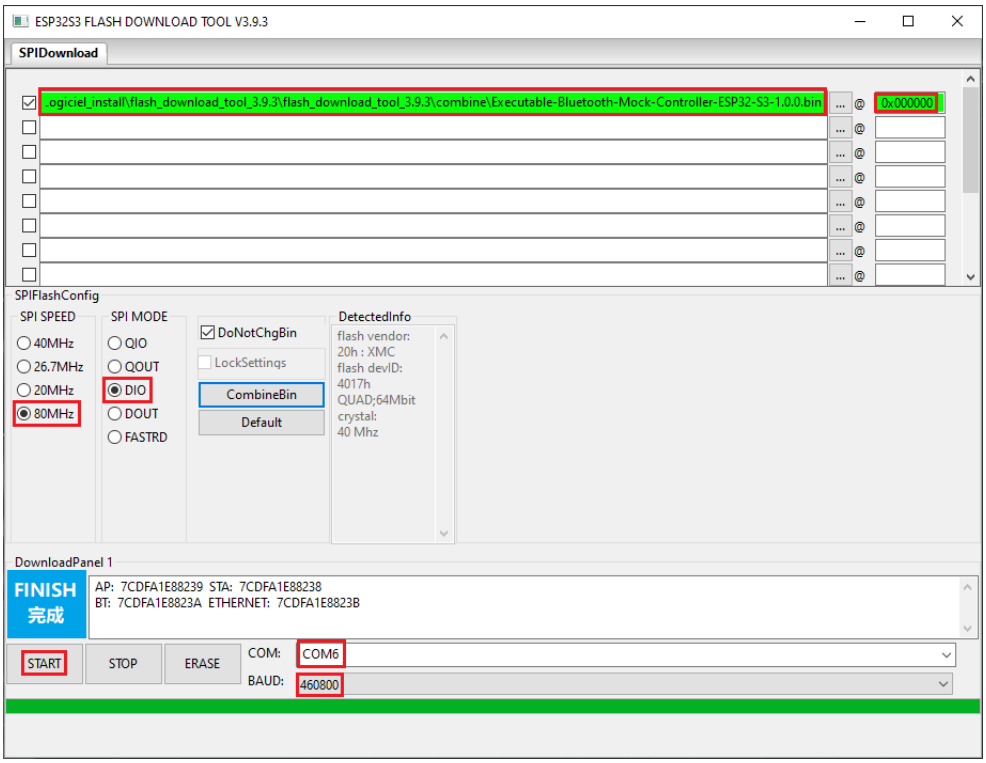
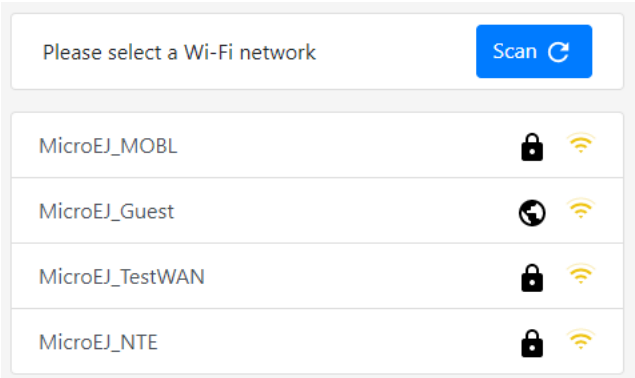


Fig. 267: Bluetooth Controller Flash Download Tool Configuration

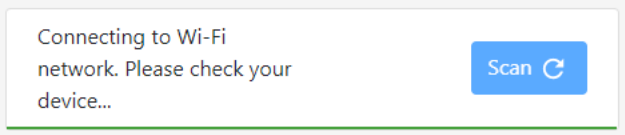
Wi-Fi Setup

To configure the Wi-Fi network used by the controller:

1. Connect your computer to the Wi-Fi network “BLE-Mock-Controller-[hexa device id]” mounted by the controller.
2. Open a browser and connect to http://192.168.4.1/ to access the Wi-Fi setup interface :



3. Select the desired network and provide the required information if asked. If an error occurs during the connection, retry this step.
4. In case the ESP32-S3 board is successfully connected to the desired network, the web page should looks like this:



Additionally, the serial output of the ESP32-S3 board shows connection status.

5. Connect your computer back to this network: your computer and the controller must be in the same network.
6. Reboot the ESP32-S3 board. At this point, the configuration is saved and the board will reconnect automatically every time it is powered up. If the board fails to connect, the configuration will be available again for re-configuration.

Simulation

To configure the Bluetooth Controller for the simulation:

1. Get the IP address of the controller from its logs:

2. Configure the application options:

   SDK 6

   SDK 5

   Set the following *application options*:

```
################################
↪##############################################################
# Bluetooth Options
################################
↪##############################################################

# Set the Bluetooth simulation mode.
# Supported values: 'stub' (default), 'net' (Bluetooth Controller)
s3.bluetooth.mode=net

# When using `net` mode, set the IP address of the Bluetooth Controller.
s3.bluetooth.net.host=192.168.x.y
```

   In the *Run configuration* panel, set the simulation mode to “Controller (over net)” and configure the connection options.

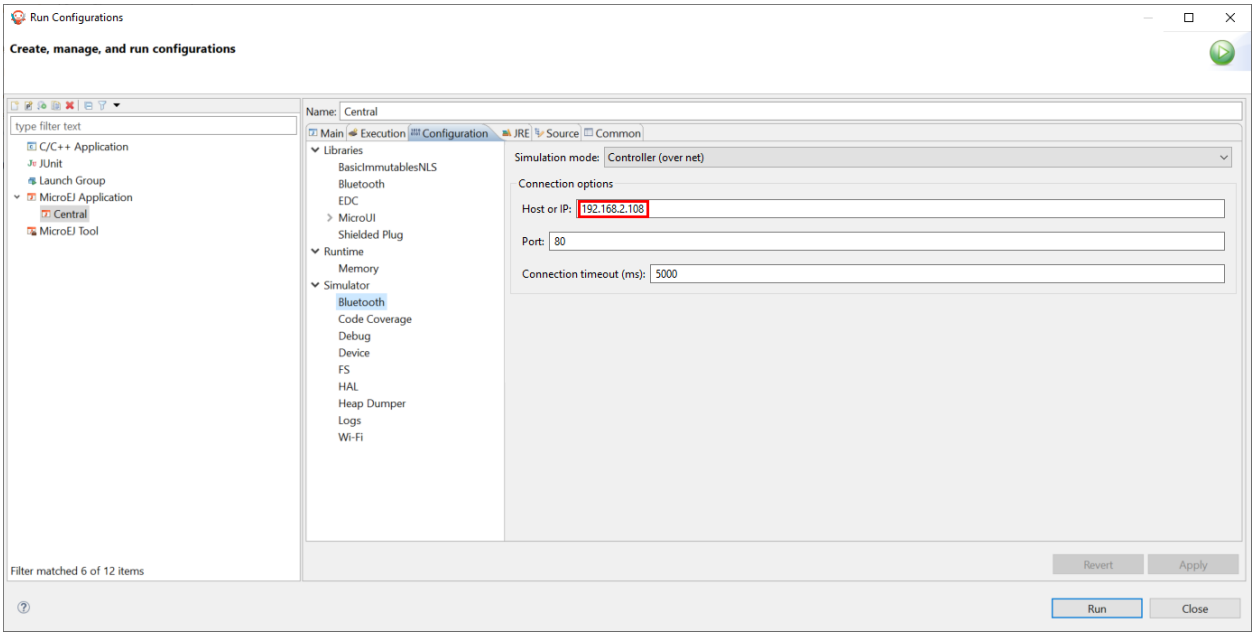


Fig. 268: Bluetooth Mock Configuration

3. Run the application on the simulator. Note that when starting a new simulation, the Bluetooth Controller is reset to its initial state.

Troubleshooting

Network Setup Errors

I can't find the "BLE-Mock-Controller-[hexa device id]" access point

The signal of this Wi-Fi access point may be weaker than the surrounding access points. Try to reduce the distance between the controller and your computer; and rescan. If it's not possible, try using a smartphone instead (only a browser will be required to set up the network configuration).

I want to override the network configuration

If the Wi-Fi credentials are not valid anymore, the controller restarts the network setup phase. Yet, in case the credentials are valid but you want to change them, erase the flash and reflash the firmware.

"Invalid parameter type: 0x47 expected 0x53" error

Reboot the ESP32-S3 board. The controller restarts and connects to the Wi-Fi.

Simulation Errors

Error during the simulation: mock could not connect to controller

This error means the mock process (Simulator) could not initialize the connection with the controller. Please check that the ESP32-S3 board is connected to the network (see logs in the serial port output) and that your computer is in the same network.

3.6.30 Appendices

SNI Native API

This chapter describes the SNI (Simple Native Interface) API. For more information on SNI specification, see *Native Interface Mechanisms* documentation.

The following API corresponds to Architecture version 8.5.0.

sni.h

Defines

SNI_VERSION (0x010400)

JTRUE (1)

JFALSE (0)

JNULL (0)

SNI_IGNORED_RETURNED_VALUE (0)

SNI_OK (0)

SNI_ERROR (-1)

SNI_ILLEGAL_ARGUMENT (-2)

SNI_getArrayLength (array) (*((*jint**)(((uint8_t*)(array))-sizeof(*jint*))))

Typedefs

typedef int8_t jbyte

typedef uint8_t jboolean

typedef uint16_t jchar

typedef int16_t jshort

typedef int32_t jint

typedef float jfloat

typedef double jdouble

typedef int64_t jlong

typedef void (* SNI_closeFunction)(void *resource)

Pointer to a function that closes the given native resource.

Param resource
the native resource to close.

typedef void (* SNI_getDescriptionFunction)(void *resource, char *buffer, uint32_t bufferLength)

Pointer to a function that gets the description of the given native resource.

The function stores the description string in the user-supplied buffer buffer of length bufferLength . At most bufferLength bytes are stored (the string may be truncated if bufferLength is too small). The string always includes a terminating null byte ('\0').

Param resource
the native resource to describe.

Param buffer
the buffer where the description is stored.

Param bufferLength
the length of the given buffer.

typedef void (* SNI_callback)(void)

Functions

bool SNI_isImmortalArray (void *javaArray)

Checks if the given Java array is immortal.

An immortal Java array remains at a fixed memory location and will not be moved or changed by Java's garbage collection routine.

An immortal Java array can be used safely from another native task, an interrupt handler or can be used as a DMA buffer.

A non-immortal Java array can be used only during the execution of an SNI native or an SNI callback.

A reference to a non-immortal Java array must not be kept in C between two SNI native executions.

Parameters

- javaArray – is a Java array retrieved from an SNI native arguments.

Returns
true if the given Java array is immortal or null, otherwise returns false .

int32_t SNI_throwNativeException (int32_t errorCode, const char *message)

Throws a new ej.sni.NativeException after the end of the current native method or SNI callback. NativeException class is a subclass of java.lang.RuntimeException . NativeException instances are unchecked exceptions, they do not need to be declared in the native method throws clause.

The virtual machine keeps a reference to the message until the end of the current native and does a copy of the message only when the current native returns. The message pointer must not reference a memory allocated locally in the stack.

If the current Java thread is suspended this function returns SNI_ERROR and no exception is thrown.

Calling this function while an exception is already pending will replace the previous exception.

If there is not enough space in the Java heap to allocate the exception, then an OutOfMemoryError will be thrown in Java.

Parameters

- errorCode – value that can be retrieved in Java using the NativeException.getErrorCode() method.
- message – is a null-terminated string that will be used to generate the message returned by NativeException.getMessage() . May be null.

Returns

SNI_OK on success. Returns SNI_ERROR if this function is not called within the virtual machine task or if the current thread is suspended.

int32_t SNI_throwNativeIOException (int32_t errorCode, const char *message)

Throws a new ej.sni.NativeIOException after the end of the current native method or SNI callback. NativeIOException class is a subclass of java.io.IOException . NativeIOException instances are checked exceptions, they need to be declared in the native method throws clause (e.g., throws IOException).

If the native method declaration is not compatible with NativeIOException (i.e., the throws clause does not specify NativeIOException or one of its superclasses) then a code>NativeException is thrown instead.

The virtual machine keeps a reference to the message until the end of the current native and does a copy of the message only when the current native returns. The message pointer must not reference a memory allocated locally in the stack.

If the current Java thread is suspended this function returns SNI_ERROR and no exception is thrown.

Calling this function while an exception is already pending will replace the previous exception.

If there is not enough space in the Java heap to allocate the exception, then an OutOfMemoryError will be thrown in Java.

Parameters

- errorCode – is a value that can be retrieved in Java using the NativeIOException.getErrorCode() method.
- message – a null-terminated string that will be used to generate the message returned by NativeIOException.getMessage() method. May be null.

Returns

SNI_OK on success. Returns SNI_ERROR if this function is not called within the virtual machine task or if the current thread is suspended.

bool SNI_isExceptionPending (void)

Checks if an exception is pending in the current Java thread (i.e., a successful call to SNI_throwNativeException or SNI_throwNativeIOException has been made prior to this call).

Returns

true if an exception is pending in the current Java thread, returns false otherwise.

int32_t SNI_clearPendingException (void)

Clears any pending exception for the current Java thread.

Returns

SNI_OK on success. Returns SNI_ERROR if the function is not called within the virtual machine task.

int32_t SNI_registerResource (void *resource, *SNI_closeFunction* close, *SNI_getDescriptionFunction* getDescription)

This function must be called when a native resource (file, socket, ...) has been created to register it in the virtual machine.

The virtual machine keeps track of the native resources to prevent any memory leak when the virtual machine or an application is stopped.

In a Multi-Sandbox environment with KF profile enabled, the given close function is called when the application that has created the resource is stopped. If the resource has been created by the Kernel or if KF profile is disabled, the close function is called when the virtual machine stops.

To unregister the given native resource, call SNI_unregisterResource() .

A native resource is uniquely identified by the pair resource , close . Several native resources with the same resource value can be registered as long as close is different.

If a native resource is registered several times with the same resource and close values, this function returns SNI_ILLEGAL_ARGUMENT (unless it has been unregistered before a second registration).

This function can be called once per native call. Calling this function several times within the same native will return SNI_ERROR .

If an error occurs while registering the resource at the end of the native, then the close function is called and an exception is thrown when returning in Java.

Parameters

- resource – the created native resource to register.
- close – the function called if the virtual machine wants to automatically close the resource.
- getDescription – the function called if the virtual machine wants to retrieve a description of the native resource. This pointer function may be null.

Returns

SNI_OK on success. Returns SNI_ILLEGAL_ARGUMENT if close is null or if the given native resource has already been registered. Returns SNI_ERROR if the function is not called within the virtual machine task or if this function has been called several times in the same native.

int32_t SNI_unregisterResource (void *resource, *SNI_closeFunction* close)

This function must be called when a native resource previously registered with SNI_registerResource() has been reclaimed. It will unregister the given native resource so the virtual machine won't call later the close function.

This method does not call the close function. It is up to the caller to call the appropriate function to close the resource.

The close parameter is used to uniquely identify the registered resource. If the given native resource is not found, this function returns SNI_ILLEGAL_ARGUMENT.

Parameters

- resource – a resource previously registered with SNI_registerResource().
- close – the close argument given to SNI_registerResource() when the native resource has been registered.

Returns

SNI_OK on success. Returns SNI_ILLEGAL_ARGUMENT if the given native resource is not registered. Returns SNI_ERROR if the function is not called within the virtual machine task.

int32_t SNI_registerScopedResource (void *resource, *SNI_closeFunction* close, *SNI_getDescriptionFunction* getDescription)

Registers a scoped native resource.

These native resources are similar to standard native resources except they consume less memory, they are faster and they are closed automatically when the native returns back to Java. This function should be used when a resource is allocated in a native and released in a callback of the same native. This will prevent memory leaks that can occur when a thread is suspended using SNI_suspendCurrentJavaThreadWithCallback() and the virtual machine or the current application is stopped.

In a Multi-Sandbox environment with KF profile enabled, the given close function is also called when the application that has created the scoped native resource is stopped. If the scoped native resource has been created by the Kernel or if KF profile is disabled, the close function is called when the virtual machine stops.

To unregister the given scoped native resource, call SNI_unregisterScopedResource().

If several scoped native resources are registered within the same native context (or subsequent callback), then this function returns SNI_ERROR (unless the previously registered scoped native resource has been unregistered).

Parameters

- resource – the created scoped native resource to register.
- close – the function called when the native returns back to Java.
- getDescription – the function called if the virtual machine wants to retrieve a description of the scoped native resource. This pointer function may be null.

Returns

SNI_OK on success. Returns SNI_ILLEGAL_ARGUMENT if close is null. Returns SNI_ERROR if the function is not called within the virtual machine task or if this function has been called several time in the same native context.

int32_t SNI_unregisterScopedResource (void)

Unregisters the scoped native resource previously registered with SNI_registerScopedResource().

After a call to this function, a new scoped native resource can be registered.

Usually this function does not need to be called because the registered scoped native resource is automatically closed and unregistered when the native returns back to Java.

This method does not call the close function given previously to SNI_registerScopedResource() . It is up to the caller to call the appropriate function to close the resource.

Returns

SNI_OK on success. Returns SNI_ERROR if the function is not called within the virtual machine task or if no scoped native resource has been previously registered.

int32_t SNI_getScopedResource (void **resourcePtr, *SNI_closeFunction* *closePtr, *SNI_getDescriptionFunction* *getDescriptionPtr)

Gets the information of the scoped native resource previously registered with SNI_registerScopedResource() .

Parameters

- resourcePtr – output parameter. This function will store in this pointer the resource value previously given to SNI_registerScopedResource() . May be null.
- closePtr – output parameter. This function will store in this pointer the close value previously given to SNI_registerScopedResource() . May be null.
- getDescriptionPtr – output parameter. This function will store in this pointer the getDescription value previously given to SNI_registerScopedResource() . May be null.

Returns

SNI_OK on success. Returns SNI_ERROR if the function is not called within the virtual machine task or if no scoped native resource has been previously registered.

int32_t SNI_retrieveArrayElements (*jbyte* *java_array, *jint* java_start, *jint* java_length, int8_t *buffer, uint32_t buffer_length, int8_t **out_buffer, uint32_t *out_length, bool refresh_content)

Gets the body of a Java byte array so it can be used outside of the scope of the current native function.

If the given Java array is an immortal array, then out_buffer is set to reference directly the element at the offset java_start in the given Java array and out_length is set to java_length . If the given Java array is not an immortal array, then out_buffer is set to reference buffer , *out_length is set to the smaller value of java_length and buffer_length , and then, if refresh_content is true, *out_length bytes are copied from the content of the Java array into buffer .

Since the returned array may be a copy of the Java array, changes made to the returned array will not necessarily be reflected in the original Java array until SNI_flushArrayElements() is called.

The function SNI_flushArrayElements() doesn't have to be called if changes made to the returned array don't need to be reflected in the original Java array.

Parameters

- java_array – the Java array.
- java_start – the start offset in array java_array of the first element to get.
- java_length – the maximum number of bytes to get.
- buffer – the buffer to use if the given Java array is not immortal.
- buffer_length – length in bytes of buffer .
- out_buffer – output pointer to the retrieved elements.
- out_length – the number of elements retrieved.

- refresh_content – specifies whether or not the retrieved array will be initialized with the elements of the given Java array.

Returns

SNI_OK on success. Returns SNI_ILLEGAL_ARGUMENT if java_start is negative, java_length is negative, java_length is greater than SNI_getArrayLength(java_array) - java_start , or one of the given pointers is null.

int32_t SNI_flushArrayElements (*jbyte* *java_array, *jint* java_start, *jint* java_length, int8_t *buffer, uint32_t buffer_length)

If necessary, this function copies back all the content of buffer to the Java array.

If the given Java array is an immortal array, this function does nothing. If the given Java array is not an immortal array, then this function copies back all the content of buffer to the Java array at the offset java_start .

Parameters

- java_array – the Java array.
- java_start – the start offset in array java_array of the first element to release.
- java_length – the maximum number of bytes to release.
- buffer – the buffer to use if the given Java array is not immortal.
- buffer_length – length in bytes of buffer .

Returns

SNI_OK on success. Returns SNI_ILLEGAL_ARGUMENT if java_start is negative, java_length is negative, java_length is greater than SNI_getArrayLength(java_array) - java_start , buffer_length is greater than java_length , or one of the given pointers is null.

void * SNI_createVM (void)

Creates and initializes a virtual machine. This function MUST be called once before a call to SNI_startVM() .

Only one instance of the virtual machine can be created in the system, and both SNI_createVM() and SNI_destroyVM() should only be called once.

Returns

null if an error occurred, otherwise returns a virtual machine instance.

int32_t SNI_startVM (void *vm, int32_t argc, char **argv)

Starts the specified virtual machine and calls the main() method of the Java application with the given String arguments. This function returns when the Java application ends.

The Java application ends when all the non-daemon threads are terminated or when the Java method System.exit(int) is called.

To restart the application, call again the SNI_startVM() function without calling SNI_createVM() or SNI_destroyVM() before.

Parameters

- vm – a pointer returned by the SNI_createVM() function.
- argc – number of string arguments in argv .
- argv – array of string arguments given to the Java main() method. May be null.

Returns
0 when the virtual machine ends normally or a negative value when an error occurred (see constant values LLMJVM_E_* defined in LLMJVM.h).

int32_t SNI_getExitCode (void *vm)

Call this function after virtual machine execution to get the Java application exit code.

Parameters

- vm – a pointer returned by the SNI_createVM() function.

Returns
the value given to the System.exit(exitCode) or 0 if the Java application ended without calling System.exit(exitCode).

void SNI_destroyVM (void *vm)

Releases all the virtual machine resources. This function must be called after the return of the function SNI_startVM(). The vm pointer is no longer valid after this call.

Only one instance of the virtual machine can be created in the system, and both SNI_createVM() and SNI_destroyVM() should only be called once.

Parameters

- vm – a pointer returned by the SNI_createVM() function.

int32_t SNI_getCurrentJavaThreadID (void)

Gets the ID of the current Java thread.

The returned ID is the same than the one returned by the Java method Thread.getId().

Returns
the ID of the current Java thread or SNI_ERROR if this function is not called within the virtual machine task.

int32_t SNI_suspendCurrentJavaThread (int64_t timeout)

Causes the current Java thread to pause its Java execution after the end of the current native method. This function is not blocking. The current Java thread will resume its execution after the reception of an external event or after timeout milliseconds.

If the pending resume flag of the current Java thread is set, the Java thread is not suspended and its pending resume flag is cleared (see SNI_isResumePending()).

If an exception is pending, the thread is not suspended and this function returns SNI_ERROR.

The result of calling this function several times during the same native execution is unpredictable.

Parameters

- timeout – duration in milliseconds of the pause. If timeout is zero, the thread waits forever until resumed.

Returns
SNI_OK if the request is accepted. Returns SNI_ERROR if this function is not called within the virtual machine task or if an exception is pending.

int32_t SNI_suspendCurrentJavaThreadWithCallback (int64_t timeout, *SNI_callback* sniCallback, void *callbackSuspendArg)

Causes the current Java thread to pause its Java execution after the end of the current native method. This function is not blocking. The current Java thread will resume its execution after the reception of an external event or after timeout milliseconds.

If a non-null SNI callback is specified, then this callback is called when the Java thread is resumed. The callback must have the same signature as the current executed native (same parameters and same return type). The callback will be executed in the virtual machine task.

The value given to callbackSuspendArg can be retrieved in the SNI callback using SNI_getCallbackArgs() .

If the pending resume flag of the current Java thread is set, the Java thread is not suspended and its pending resume flag is cleared (see SNI_isResumePending()).

If an exception is pending, the thread is not suspended, this function returns SNI_ERROR , and the callback is not called.

The result of calling this function several times during the same native execution is unpredictable.

Parameters

- timeout – duration in milliseconds of the pause. If timeout is zero, the thread waits forever until resumed.
- sniCallback – function called when the Java thread is resumed. May be null.
- callbackSuspendArg – argument that can be retrieved in the callback using SNI_getCallbackArgs() . May be null.

Returns

SNI_OK if the request is accepted. Returns SNI_ERROR if this function is not called within the virtual machine task or if an exception is pending.

int32_t SNI_javaThreadYield (*SNI_callback* sniCallback, void *callbackArg)

Causes the current Java thread to yield its execution to another Java thread after the end of the current native method and then to call the given callback when it will be scheduled again.

The callback must have the same signature as the current executed native (same parameters and same return type). The callback will be executed in the virtual machine task.

This function can be used to split the execution of a long native method into several increments leaving some execution time to other Java threads.

If an exception is pending, this function returns SNI_ERROR , and the callback is not called.

The result of calling this function several times during the same native execution is unpredictable.

Parameters

- sniCallback – function called when the Java thread is scheduled again. May be null.
- callbackArg – argument that can be retrieved in the callback using SNI_getCallbackArgs() . May be null.

Returns

SNI_OK on success. Returns SNI_ERROR if this function is not called within the virtual machine task or if an exception is pending.

int32_t SNI_getCallbackArgs (void **callbackSuspendArgPtr, void **callbackResumeArgPtr)

Retrieves the arguments given to an SNI callback with the function SNI_javaThreadYield() , SNI_suspendCurrentJavaThreadWithCallback() or SNI_resumeJavaThreadWithArg() .

This function should be called in an SNI callback. If the functions SNI_suspendCurrentJavaThreadWithCallback() or SNI_javaThreadYield() have not been called previously for the current native or one of its callback, the value returned in callbackSuspendArgPtr is undefined. If the function SNI_resumeJavaThreadWithArg() has not been called previously for the current native or one of its callback, the value returned in callbackResumeArgPtr is undefined.

Parameters

- callbackSuspendArgPtr – pointer to an argument. This function will store in this pointer the callbackSuspendArg value previously given to SNI_suspendCurrentJavaThreadWithCallback() or callbackArg value previously given to SNI_javaThreadYield() . May be null.
- callbackResumeArgPtr – pointer to an argument. This function will store in this pointer the callbackResumeArg value previously given to SNI_resumeJavaThreadWithArg() . May be null.

Returns

SNI_OK on success. Returns SNI_ERROR if this function is not called within the virtual machine task.

int32_t SNI_resumeJavaThread (int32_t javaThreadID)

Resumes the given Java thread if it is suspended. If the Java thread is not suspended, its pending resume flag is set and the next call of SNI_suspendCurrentJavaThread() will not suspend the thread.

Parameters

- javaThreadID – ID of the Java thread to resume. The Java thread ID can be retrieved using SNI_getCurrentJavaThreadID() function.

Returns

SNI_ERROR if the given Java thread ID is invalid, otherwise returns SNI_OK .

int32_t SNI_resumeJavaThreadWithArg (int32_t javaThreadID, void *callbackResumeArg)

Resumes the given Java thread if it is suspended. If the Java thread is not suspended, its pending resume flag is set and the next call of SNI_suspendCurrentJavaThread() will not suspend the thread.

Parameters

- javaThreadID – ID of the Java thread to resume. The Java thread ID can be retrieved using SNI_getCurrentJavaThreadID() function.
- callbackResumeArg – argument that can be retrieved in the callback given to SNI_suspendCurrentJavaThreadWithCallback() using SNI_getCallbackArgs() . May be null.

Returns

SNI_ERROR if the given Java thread ID is invalid, otherwise returns SNI_OK .

bool SNI_isResumePending (int32_t javaThreadID)

Tests whether the given Java thread has been resumed while it was not suspended.

The pending resume flag is set when calling SNI_resumeJavaThread() or SNI_resumeJavaThreadWithArg(). It is cleared when calling SNI_suspendCurrentJavaThread() or SNI_suspendCurrentJavaThreadWithCallback(), or when the Java thread actually resumes its execution. The pending resume flag is not modified by a timeout that occurs during a pause.

The pending resume flag of the given Java thread is unaffected by this method.

If the given Java thread ID is invalid, this method does nothing and returns false.

Parameters

- javaThreadID – ID of the Java thread to test. The Java thread ID can be retrieved using SNI_getCurrentJavaThreadID() function.

Returns

true if the pending resume flag of the given Java thread is set, false otherwise.

bool SNI_clearCurrentJavaThreadPendingResumeFlag (void)

Clears the pending resume flag of the current Java thread (see SNI_isResumePending()).

If this method is not called within the virtual machine task, it does nothing and returns false.

Returns

true if the pending resume flag of the current Java thread was set before clearing it, false otherwise.

Low Level API

This chapter describes succinctly the available Low Level API, module by module. The exhaustive documentation of each LLAPI function is available in the LLAPI header files themselves. The required header files to implement are automatically copied in the folder include of the VEE Port at build time.

Low Level API Pattern

Principle

Each time the user has to supply the C code that links a VEE Port component to the target hardware, a *Low Level API* is defined. There is a standard pattern for the definition and implementation of these APIs. Each interface has a name and is specified by two header files:

- [INTERFACE_NAME].h specifies the functions that make up the public API of the implementation. In some cases the user code will never act as a client of the API, and so will never use this file.
- [INTERFACE_NAME]_impl.h specifies the functions that must be coded by the user in the implementation.

The user creates *implementations* of the interfaces, each captured in a separate C source file. In the simplest form of this pattern, only one implementation is permitted, as shown in the illustration below.

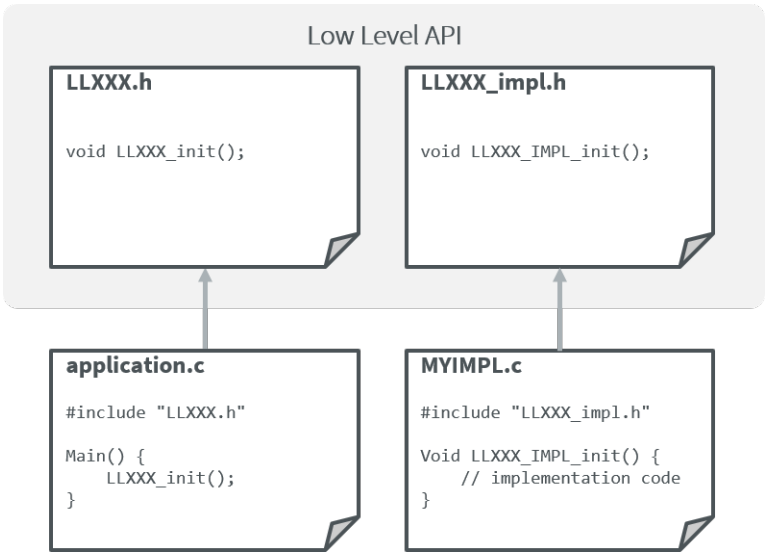


Fig. 269: Low Level API Pattern (single implementation)

The following figure shows a concrete example of an LLAPI. The C world (the board support package) has to implement a send function and must notify the library using a receive function.

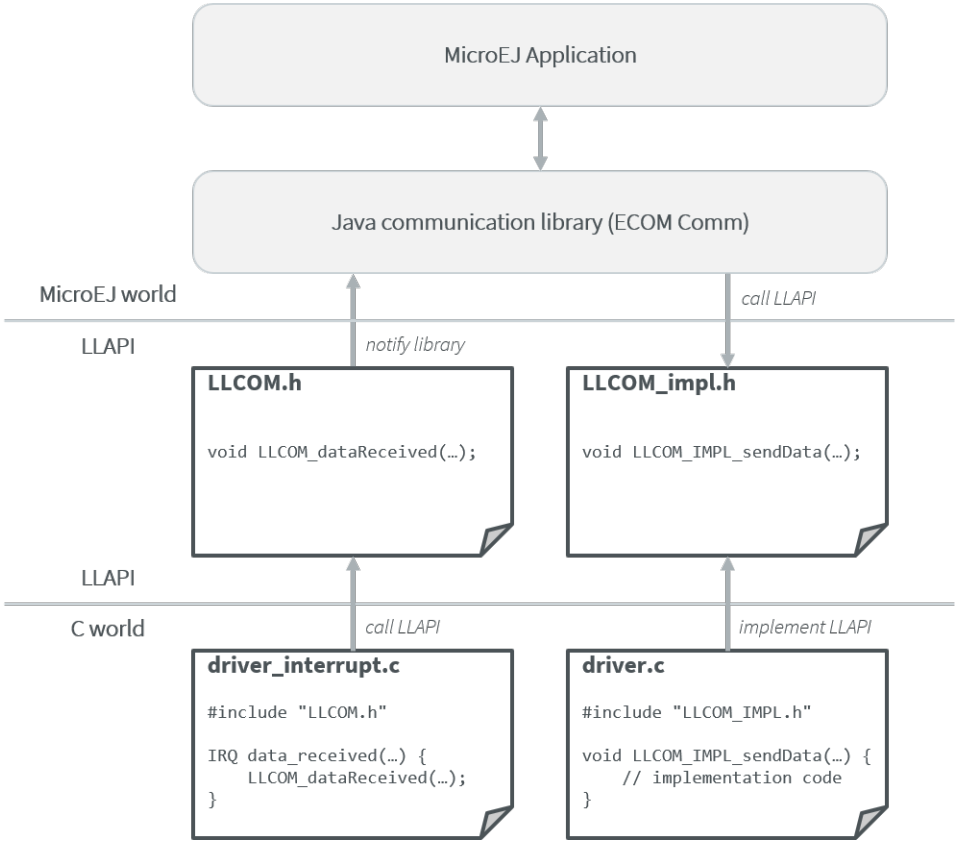


Fig. 270: Low Level API Example

Multiple Implementations and Instances

When a Low Level API allows multiple implementations, each implementation must have a unique name. At run-time there may be one or more instances of each implementation, and each instance is represented by a data structure that holds information about the instance. The address of this structure is the handle to the instance, and that address is passed as the first parameter of every call to the implementation.

The illustration below shows this form of the pattern, but with only a single instance of a single implementation.

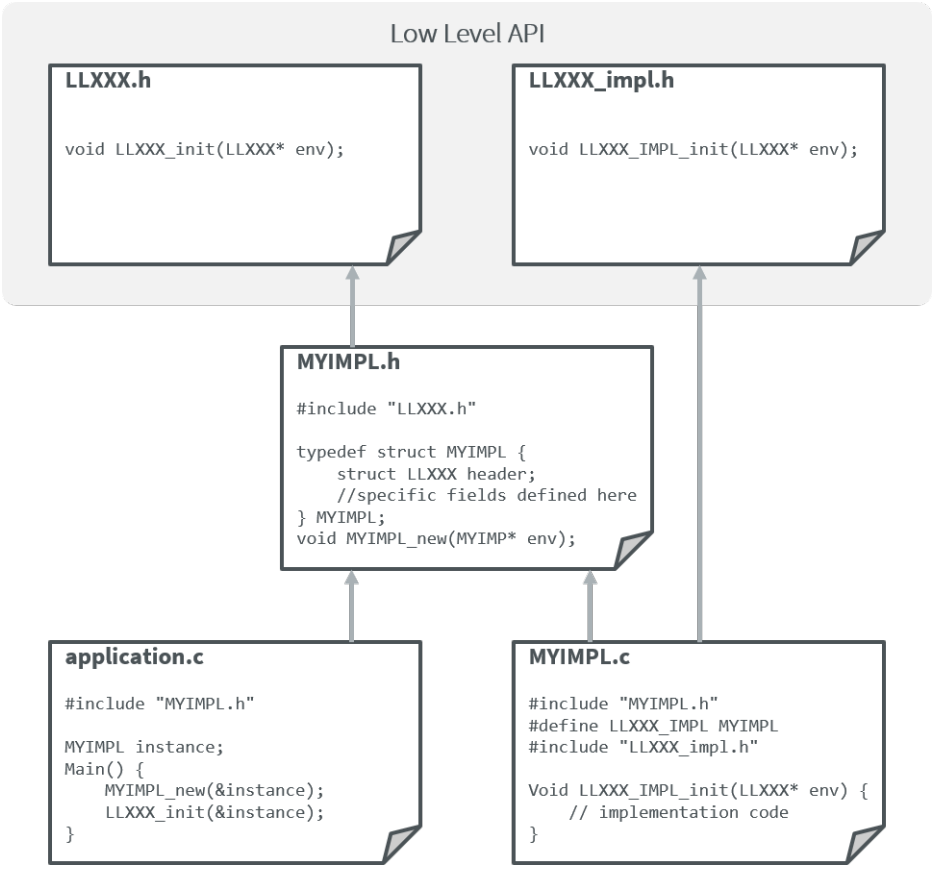


Fig. 271: Low Level API Pattern (multiple implementations/instances)

The #define statement in MYIMPL.c specifies the name given to this implementation.

LLMJVM: Core Engine

The following API corresponds to architecture version 8.5.0.

LLMJVM_impl.h

Set of functions that the BSP must implement to launch and schedule the Core Engine

Defines

LLMJVM_OK (0)

Returned value when function call has succeeded.

LLMJVM_INTERRUPTED (1)

Returned value when current task has been interrupted during function call.

LLMJVM_ERROR (-1)

Returned value when function call has failed.

Functions

int32_t LLMJVM_IMPL_initialize (void)

This function is called once during Core Engine creation (i.e., when the SNI_createVM() function is called). It may be used to initialize specific data.

Returns
LLMJVM_OK on success, *LLMJVM_ERROR* on error.

int32_t LLMJVM_IMPL_vmTaskStarted (void)

This function is called once during the Core Engine startup by the Core Engine task (i.e., when the SNI_startVM() function is called). It can be useful if the Core Engine support needs to know the Core Engine task.

Returns
LLMJVM_OK on success, *LLMJVM_ERROR* on error.

int32_t LLMJVM_IMPL_scheduleRequest (int64_t absoluteTime)

Schedule an alarm (or timer) that will be triggered at the given absolute (system) time. If an alarm is already scheduled for an earlier time this function must do nothing, otherwise it must configure the alarm. If the given absolute time has already been reached, this function must call *LLMJVM_schedule*. Previously scheduled alarm must be canceled, only one alarm is scheduled at the same time. The scheduled alarm must call the function *LLMJVM_schedule* when it is triggered. The specified time is in milliseconds.

Returns
LLMJVM_OK on success, *LLMJVM_ERROR* on error.

int32_t LLMJVM_IMPL_idleVM (void)

Causes the Core Engine RTOS task to sleep until it is woken up by the LLMJVM_wakeupVM function. This function is called by the Core Engine task.

Returns

LLMJVM_OK if wakeup occurred, *LLMJVM_INTERRUPTED* if the Core Engine task has been interrupted, or *LLMJVM_ERROR* on error.

int32_t LLMJVM_IMPL_wakeupVM (void)

Wake up the Core Engine RTOS task. If the Core Engine task is not sleeping, the wakeup stays pending and the Core Engine will not sleep on the next LLMJVM_idleVM call unless there is a call to LLMJVM_ackWakeup between this call and the next LLMJVM_idleVM call. This function must be called only by the Core Engine code. If a task wants to wake up the Core Engine, it must use the *LLMJVM_schedule* function (which may in turn call this function).

Returns

LLMJVM_OK on success, *LLMJVM_ERROR* on error.

int32_t LLMJVM_IMPL_ackWakeup (void)

Clears any outstanding LLMJVM_wakeupVM request. After calling this function, a call to LLMJVM_idleVM will result in a wait even if LLMJVM_wakeupVM has been called previously (provided no other LLMJVM_wakeupVM call has occurred since the call to this function. This function must cancel the alarm previously scheduled with LLMJVM_scheduleRequest. This function is called by the Core Engine task.

Returns

LLMJVM_OK on success, *LLMJVM_ERROR* on error.

int32_t LLMJVM_IMPL_getCurrentTaskID (void)

Returns the ID of the current OS task.

This function may be called within the Core Engine task or another OS task.

int32_t LLMJVM_IMPL_shutdown (void)

This function is called during Core Engine end (i.e., at the end of the SNI_startVM() function). It may be used to freed specific data.

Returns

LLMJVM_OK on success, *LLMJVM_ERROR* on error.

void LLMJVM_IMPL_setApplicationTime (int64_t t)

Sets the application time. The application time is the difference, measured in milliseconds, between the current time and midnight, January 1, 1970 UTC. This time does not change the monotonic time.

Parameters

- t – the application time to set in milliseconds.

int64_t LLMJVM_IMPL_getCurrentTime (uint8_t monotonic)

Gets the monotonic or the application time in milliseconds.

The monotonic time always moves forward and is not impacted by application time modifications (NTP or Daylight Savings Time updates). It can be implemented by returning the running time since the start of the device. This time is the value returned by the Java method Util.platformTimeMillis() .

The application time is the difference, measured in milliseconds, between the current time and midnight, January 1, 1970 UTC. This time is the value returned by the Java methods System.currentTimeMillis() and Util.currentTimeMillis() .

Parameters

- monotonic – if 1, get the monotonic time, otherwise get the application time.

int64_t LLMJVM_IMPL_getTimeNanos (void)

Gets the current timestamp in nanoseconds. Only elapsed time between two calls is meaningful.

Returns
the current timestamp in nanoseconds

int32_t LLMJVM_schedule (void)

Notify the Core Engine that it must wake up or schedule a new thread.

Returns
LLMJVM_OK on success, *LLMJVM_ERROR* on error.

LLMJVM.h

Set of functions provided by Core Engine that can be called by the BSP

Defines

LLMJVM_E_OK (0)

GreenThread Core Engine API. Exit code returned when the application ends normally (i.e., all the non-daemon threads are terminated or System.exit(exitCode) has been called)

LLMJVM_E_SOAR_FILE_INCOMPATIBLE (-1)

Exit code returned when the SOAR used for the application is not compatible with the Core Engine.

LLMJVM_E_BAD_LINK (-2)

Exit code returned when the link specific configuration breaks Core Engine requirements.

LLMJVM_E_EVAL_LIMIT (-3)

Exit code returned when the application ends because of the evaluation version limitations.

LLMJVM_E_MAIN_THREAD_ALLOC (-5)

Exit code returned when the Main thread can not be launched (for instance if thread stack can not be allocated).

LLMJVM_E_TOO_MANY_THREADS (-12)

Exit code returned when the specified maximum number of threads exceeds Core Engine limits.

LLMJVM_E_INV_HEAP_SIZE (-13)

Exit code returned when the Java heap size is too small or too large.

LLMJVM_E_INV_JAVA_STACK_MEM (-14)

Exit code returned when the Java stacks memory is invalid. Check the stacks memory alignment or size.

LLMJVM_E_CANNOT_RESTART (-16)

Exit code returned when the Core Engine cannot be restarted.

LLMJVM_E_INV_STATE (-17)

Exit code returned when the Core Engine is not in a valid state.

LLMJVM_E_INV_HEAP_MEMORY (-18)

Exit code returned when the memory used for the Java heap or the immortal heap does not work properly. This may be caused by an invalid external RAM configuration.

LLMJVM_E_INV_STATIC_MEMORY (-19)

Exit code returned when the memory used for the Java static fields does not work properly. This may be caused by an invalid external RAM configuration.

LLMJVM_E_INV_KF_RULES_CONFIGURATION (-20)

Exit code returned when KF rules configuration section is not correct. Check linker script files.

LLMJVM_E_INV_TOO_MANY_MONITORS_PER_THREAD (-21)

Exit code returned when the maximum number of monitors per threads exceeds Core Engine limits.

LLMJVM_E_INV_FPU_CONVENTION_CALL (-22)

Exit code returned when FPU convention call requirements are not satisfied.

LLMJVM_E_INITIALIZE_ERROR (-23)

Exit code returned when LLMJVM_IMPL_initialize() function defined in the abstraction layer returns an error.

LLMJVM_E_VM_TASK_STARTED_ERROR (-24)

Exit code returned when LLMJVM_IMPL_vmTaskStarted() function defined in the abstraction layer returns an error.

LLMJVM_E_SHUTDOWN_ERROR (-25)

Exit code returned when LLMJVM_IMPL_shutdown() function defined in the abstraction layer returns an error.

LLMJVM_E_INV_GC_MARK_STACK_SIZE (-26)

Exit code returned when the GC mark stack size is too small.

LLMJVM_E_INV_LINK (-27)

Exit code returned when the application object file has not been correctly linked by the third-party linker.

LLMJVM_E_INV_MANAGED_HEAP_CONFIG (-28)

Exit code returned when LLMJVM_CONF_impl_get_managed_heap_memory function returns an error when trying to configure a managed heap.

LLMJVM_E_INV_IMMORTALS_HEAP_CONFIG (-29)

Exit code returned when LLMJVM_CONF_impl_get_immortals_heap_memory function returns an error when trying to configure immortals heap.

LLMJVM_E_ERROR (-1)

Error code returned when an error occurred.

LLMJVM_E_ILLEGAL_ARGUMENT (-2)

Error code returned when an illegal argument has been given.

LLMJVM_E_INTERRUPTED (1)

Error code returned by *LLMJVM_suspendCurrentJavaThread* when suspend is not done.

Functions

void LLMJVM_initialize (void)

Initializes the Core Engine. This function MUST be called once before a call to *LLMJVM_start*.

int32_t LLMJVM_start (int32_t argc, void *argv)

Starts the Core Engine and call the main method of the Java application with the given String arguments. This function returns when the Java application ends.

The Java application ends when there is no more Java thread to run or when the Java method System.exit(int) is called.

Parameters

- argc – java main argument count.

- argv – java main argument vector. An array of String (char**).

Returns
LLMJVM_E_OK when the Core Engine ends normally or a negative value when an error occurred during startup.

int32_t LLMJVM_getExitCode (void)

Call this method after Core Engine execution to get the Java application exit code.

Returns
the value given to the System.exit(exitCode) or 0 if the Java application ends without calling System.exit(exitCode) .

void LLMJVM_dump (void)

Prints the state of the Core Engine to the standard output stream. For each Java thread, the Java stack trace, the name, the state and the priority are printed.

This function must only be called from the Core Engine thread context and only from a native function or callback. Calling this function from another context may lead to undefined behavior and should be done only for debug purpose.

int32_t LLMJVM_checkIntegrity (void)

Checks internal structure integrity of Core Engine and returns its checksum.

If an integrity error is detected, the void LLMJVM_on_CheckIntegrity_error(uint32_t errorCode, void* errorAddress) hook is called and this method returns 0 .

If no integrity error is detected, a non-zero checksum is returned.

This function must only be called from the Core Engine thread context and only from a native function or callback. Calling this function multiple times in a native function must always produce the same checksum. If the returned checksums are different, a corruption must have occurred.

Please note that returning a non-zero checksum does not mean the Core Engine data has not been corrupted, as it is not possible for the Core Engine to detect the complete memory integrity.

The internal structures of the Core Engine that can be altered legitimately by a native function do not impact the checksum calculation. The following internal structures may be modified without affecting the checksum:

- basetype fields in Java objects or content of Java arrays of base type,
- internal structures modified by a LLMJVM function call (e.g., set a pending Java exception, suspend or resume the Java thread, register a resource, …).

This function affects performance and should only be used for debug purpose

A typical use of this API is to verify that a native implementation does not corrupt the internal structures:

Returns
a 32 bits integrity checksum if the integrity check has run without errors, or 0 if an error has been detected.

uint8_t LLMJVM_isImmortalArray (void *sniJavaArray)

Checks if the given Java array is immortal or not.

An immortal Java array can be used safely from another native task, an interrupt handler or a DMA.

A non-immortal Java array can be used only during the execution of an SNI native or an SNI callback. A reference to a non-immortal Java array must not be kept in C between to SNI native executions.

Parameters

- sniJavaArray – is a Java array retrieved from an SNI native arguments. This pointer reference the first element of an array and not the header of the object.

Returns 1
if the given Java array is immortal or null, otherwise returns 0.

int32_t LLMJVM_throwNativeException (int32_t errorCode, void *message, uint8_t ioException)

If ioException is false:

Throws a new ej.sni.NativeException after the end of the current native method or SNI callback. NativeException class is a subclass of java.lang.RuntimeException . NativeException instances are unchecked exceptions, they do not need to be declared in the native method throws clause.

If ioException is true:

Throws a new ej.sni.NativeIOException after the end of the current native method or SNI callback. NativeIOException class is a subclass of java.io.IOException . NativeIOException instances are checked exceptions, they need to be declared in the native method throws clause (e.g., throws IOException). If the native method declaration is not compatible with NativeIOException (i.e., the throws clause does not specify NativeIOException or one of its superclasses) then a code>NativeException is thrown instead.

If an SNI callback must be executed after the end of the current native method then it will be executed first and then the exception will be thrown.

Calling this function while an exception is already pending will replace the previous exception.

Parameters

- errorCode – is a value that can be retrieved in Java using the Native(IO)Exception.getErrorCode() method.
- message – is a null-terminated string that will be used to generate the message returned by Native(IO)Exception.getMessage() . It may be null.
- ioException – if true throws a NativeIOException, if false throws a NativeException.

Returns
LLMJVM_E_OK on success. Returns *LLMJVM_E_ERROR* if the function is called outside of the VM Task.

uint8_t LLMJVM_isExceptionPending (void)

Returns 1 if an exception is pending in the current Java thread (i.e., a successful call to *LLMJVM_throwNativeException* or throwNativeIOException(int, Ram) has been made prior to this call), returns 0 otherwise.

int32_t LLMJVM_clearPendingException (void)

Clears any pending exception for the current Java thread.

Returns

LLMJVM_E_OK on success. Returns *LLMJVM_E_ERROR* if the function is called outside of the VM Task.

int32_t LLMJVM_registerResource (void *resource, void *close, void *getDescription)

This function must be called when a native resource (file, socket, ...) has been created to register it in the Core Engine.

The Core Engine keeps track of the native resources to prevent any memory leak when the Core Engine or an application is stopped.

In a Multi-Sandbox environment with KF profile enabled, the given close function is called when the application that has created the resource is stopped. If the resource has been created by the Kernel or if KF profile is disabled, the close function is called when the Core Engine stops.

To unregister the given native resource, call *LLMJVM_unregisterResource*.

A native resource is uniquely identified by the pair resource , close . Several native resources with the same resource value can be registered as long as close is different.

If a native resource is registered several times with the same resource and close values, this method returns *LLMJVM_E_ILLEGAL_ARGUMENT* (unless it has been unregistered before a second registration).

This function can be called once per native call. Calling this function several time within the same native will return *LLMJVM_E_ERROR*.

If an error occurs while registering the resource, then the close function is called and an exception is thrown when returning in Java.

Parameters

- resource – the created native resource to register.
- close – the function called if the Core Engine wants to automatically close the resource.
- getDescription – the function called if the Core Engine wants to retrieve a description of the native resource. This pointer function may be null.

Returns

LLMJVM_E_OK on success. Returns *LLMJVM_E_ILLEGAL_ARGUMENT* if close is null or if the given native resource has already been registered. Returns *LLMJVM_E_ERROR* if the function is not called within the Core Engine task or if this function has been called several time in the same native.

int32_t LLMJVM_unregisterResource (void *resource, void *close)

This function must be called when a native resource previously registered with *LLMJVM_registerResource* has been reclaimed. It will unregister the given native resource so the Core Engine won't call later the close function.

This method does not call the close function. It is up to the caller to call the appropriate function to close the resource.

The close parameter is used to uniquely identify the registered resource. If the given native resource is not found, this method returns *LLMJVM_E_ILLEGAL_ARGUMENT*.

Parameters

- resource – a resource previously registered with *LLMJVM_registerResource*.
- close – the close argument given to *LLMJVM_registerResource* when the native resource has been registered.

Returns

LLMJVM_E_OK on success. Returns *LLMJVM_E_ILLEGAL_ARGUMENT* if the given native resource is not registered. Returns *LLMJVM_E_ERROR* if the function is not called within the Core Engine task.

int32_t LLMJVM_registerScopedResource (void *resource, void *close, void *getDescription)

Registers a scoped native resource.

These native resources are similar to standard native resources but they consume less memory, they are faster and they are closed automatically when the native returns back to Java. This function should be used when a resource is allocated in a native and released in a callback of the same native. This will prevent memory leaks that can occur when a thread is suspended using SNI_suspendCurrentJavaThreadWithCallback() and the Core Engine or the current application is stopped.

In a Multi-Sandbox environment with KF profile enabled, the given close function is called when the application that has created the scoped native resource is stopped. If the scoped native resource has been created by the Kernel or if KF profile is disabled, the close function is called when the Core Engine stops.

To unregister the given scoped native resource, call *LLMJVM_unregisterScopedResource*.

If several scoped native resources are registered within the same native context (or subsequent callback), then this function returns *LLMJVM_E_ERROR* (unless the previously registered scoped native resource has been unregistered).

Parameters

- resource – the created scoped native resource to register.
- close – the function called if the Core Engine wants to automatically close the resource.
- getDescription – the function called if the Core Engine wants to retrieve a description of the scoped native resource. This pointer function may be null.

Returns

LLMJVM_E_OK on success. Returns *LLMJVM_E_ILLEGAL_ARGUMENT* if close is null. Returns *LLMJVM_E_ERROR* if the function is not called within the Core Engine task or if this function has been called several time in the same native context.

int32_t LLMJVM_unregisterScopedResource (void)

Unregisters the scoped native resource previously registered with *LLMJVM_registerScopedResource*.

After a call to this function, a new scoped native resource can be registered.

Usually this function does not need to be called because the registered scoped native resource is automatically closed and unregistered when the native returns back to Java.

Returns

LLMJVM_E_OK on success. Returns *LLMJVM_E_ERROR* if the function is not called within the Core Engine task or if no scoped native resource has been previously registered.

int32_t LLMJVM_getScopedResource (void *resourcePtr, void *closePtr, void *getDescriptionPtr)

Gets the information of the scoped native resource previously registered with *LLMJVM_registerScopedResource*.

Parameters

- resourcePtr – output parameter. This function will store in this pointer the resource value previously given to *LLMJVM_registerScopedResource*. May be null.
- closePtr – output parameter. This function will store in this pointer the close value previously given to *LLMJVM_registerScopedResource*. May be null.
- getDescriptionPtr – output parameter. This function will store in this pointer the getDescription value previously given to *LLMJVM_registerScopedResource*. May be null.

Returns

LLMJVM_E_OK on success. Returns *LLMJVM_E_ERROR* if the function is not called within the Core Engine task or if no scoped native resource has been previously registered.

int32_t LLMJVM_getJavaResource (void *path, int32_t pathLength, void *resource)

Gets the Java resource data identified by the given path.

Parameters

- path – string that contains the path. It must start with '/'.
- pathLength – length of the path string.
- resource – structure where the information of the resource are stored.

Returns

LLMJVM_E_OK on success. Returns *LLMJVM_E_ERROR* if the resource is not found. Returns *LLMJVM_E_ILLEGAL_ARGUMENT* if pathLength is lower than 1 or resource is null.

int32_t LLMJVM_getCurrentJavaThreadID (void)

Returns the ID of the current Java thread. This function should be called within the Core Engine task. Returns *LLMJVM_E_ERROR* if this function is not called within the Core Engine task.

int32_t LLMJVM_suspendCurrentJavaThread (int64_t timeout, void *callback, void *callbackSuspendArg, uint8_t interruptible)

Causes the current Java thread to pause its Java execution after the end of the current native method. This function is not blocking. The current Java thread will resume its execution after the reception of an external event or after timeout milliseconds.

If a resume has been done on this thread before calling this function, the thread is not paused.

The result of calling this method several times during the same native execution is unpredictable.

If a callback is specified, then this callback is called before resuming the Java execution.

Parameters

- timeout – duration in milliseconds of the pause. If timeout is zero, then time is not taken into consideration and the thread simply waits until resumed.
- callback – method to call when the timeout is reached or the sendExternalEvent(VMEngine) method is called on the current Java thread. May be null.

- callbackSuspendArg – argument that can be retrieved in the callback using *LLMJVM_getCallbackArgs*. May be null.
- interruptible – if true the suspend can be interrupted with a call to Thread.interrupt(), otherwise the suspend cannot be interrupted.

Returns

LLMJVM_E_ERROR if the method has been called outside of the VM Task. Returns *LLMJVM_E_OK* if the pause is effective (i.e., the thread will suspend its execution at the end of the current native). Returns *LLMJVM_E_INTERRUPTED* if interruptible is true and an interrupt is pending (see Thread.interrupt()).

int32_t LLMJVM_javaThreadYield (void *callback, void *callbackArg)

Causes the current Java thread to call Thread.yield() after the end of the current native method and then to call the given callback when it will be scheduled again.

The result of calling this method several times during the same native execution is unpredictable.

Parameters

- callback – method to call when the thread will be scheduled. May be null.
- callbackArg – argument that can be retrieved from the callback using *LLMJVM_getCallbackArgs*. May be null.

Returns

LLMJVM_E_ERROR if the method has been called outside of the VM Task. Returns *LLMJVM_E_OK* on success.

int32_t LLMJVM_getCallbackArgs (void *suspendArgPtr, void *resumeArgPtr)

This method can be called from inside a SNI callback to retrieve arguments given to the callback with *LLMJVM_javaThreadYield*, *LLMJVM_suspendCurrentJavaThread* or *LLMJVM_resumeJavaThread*.

Parameters

- suspendArgPtr – Out parameter. Pointer to the argument (void**) that will be filled-in by this method with the callbackSuspendArg value given to *LLMJVM_suspendCurrentJavaThread* or callbackArg value given to *LLMJVM_javaThreadYield*. May be null.
- resumeArgPtr – Out parameter. Pointer to the argument (void**) that will be filled-in by this method with the callbackResumeArg value given to *LLMJVM_resumeJavaThread*. May be null.

Returns

LLMJVM_E_ERROR if the method has been called outside of the VM Task, otherwise returns *LLMJVM_E_OK*.

int32_t LLMJVM_resumeJavaThread (int32_t javaThreadID, void *callbackResumeArg)

Resume the given Java thread if it is suspended. If the Java thread is not paused, this resume stays pending. Next call of *SNI_suspendCurrentJavaThread()* will return immediately.

Parameters

- javaThreadID – is the ID of the Java thread to resume.
- callbackResumeArg – argument that can be retrieved from the callback using *LLMJVM_getCallbackArgs*. May be null.

Returns
LLMJVM_E_ERROR if the given Java thread ID is invalid, otherwise returns *LLMJVM_E_OK*.

uint8_t LLMJVM_isResumePending (int32_t javaThreadID)

Tests whether the given Java thread has been resumed while it was not suspended.

The pending resume flag is set when calling *LLMJVM_resumeJavaThread*. It is cleared when calling *LLMJVM_suspendCurrentJavaThread*, or when the Java thread actually resumes its execution. The pending resume flag is not modified by a timeout that occurs during a pause.

The pending resume flag of the given Java thread is unaffected by this method.

If the given Java thread ID is invalid, this method does nothing and returns false.

Parameters

- javaThreadID – ID of the Java thread to test. The Java thread ID can be retrieved using *LLMJVM_getCurrentJavaThreadID* function.

Returns 1
if the pending resume flag of the given Java thread is set, 0 otherwise.

uint8_t LLMJVM_clearCurrentJavaThreadPendingResumeFlag (void)

Clears the pending resume flag of the current Java thread (see *LLMJVM_isResumePending*).

If this method is not called within the Core Engine task, it does nothing and returns false.

Returns 1
if the pending resume flag of the current Java thread was set before clearing it, 0 otherwise.

LLBSP_impl.h

Set of extra functions that the BSP must implement.

Functions

void LLBSP_IMPL_putchar (int32_t c)

Writes the character c , cast to an unsigned char, to stdout stream.

This function is used by the default implementation of the Java System.out .

LLMJVM_CONF_impl.h

Set of functions that can be implemented to configure the Core Engine at startup.

Functions

int LLMJVM_CONF_impl_get_managed_heap_memory (void **out_ptr, size_t *out_total_size, size_t *out_initial_size)

Optional user-implemented callback to provide a memory region for the Managed Heap.

The Core Engine calls this function to obtain:

- a pointer to a memory block that will be used as the Managed Heap,
- its total size in bytes,
- its initial usable size in bytes (the size the Core Engine will start using, and may extend up to the total size as needed).

The memory returned should preferably be aligned to 4 bytes, but if it is not, the framework will handle alignment itself by adjusting the start address and may reduce the usable size accordingly.

The initial size and the total size must be set to the same value, unless you want to activate Automatic Heap Usage Monitoring. To enable Automatic Heap Usage Monitoring, set the initial size to a value strictly less than the total size: the framework will then start by using the initial size, and may extend its usage up to the total size as needed.

This function is called once during Core Engine creation (i.e., when the SNI_createVM() function is called).

This function is optional. The Core Engine provides a default weak implementation, which returns the address, the total size, and the initial size as defined by the application, typically using the .bss.microej.heap linker section and associated symbols. Users can override this function to provide a custom Managed Heap configuration.

Parameters

- out_ptr – Pointer to a void* variable to receive the address of the memory block to be used as the Managed Heap.
- out_total_size – Pointer to a size_t variable to receive the total size of the memory block in bytes referenced by out_ptr .
- out_initial_size – Pointer to a size_t variable that receives the initial usable size in bytes of the memory block. By default (when Automatic Heap Usage Monitoring is not desired), this value must be equal to the total size. To enable Automatic Heap Usage Monitoring, set this value lower than the total size.

Returns 0
on success, -1 on error.

int LLMJVM_CONF_impl_get_immortals_heap_memory (void **out_ptr, size_t *out_size)

Optional user-implemented callback to provide a memory region for the Immortals Heap.

The Core Engine calls this function to obtain:

- a pointer to a memory block that will be used as the Immortals Heap,
- its size in bytes.

The memory returned should preferably be aligned to 4 bytes, but if it is not, the framework will handle alignment itself by adjusting the start address and may reduce the usable size accordingly.

This function is called once during Core Engine creation (i.e., when the SNI_createVM() function is called).

This function is optional. The Core Engine provides a default weak implementation, which returns the address and the size as defined by the application, typically using the .bss.microej.immortals linker section and associated symbols. Users can override this function to provide a custom Immortals Heap configuration.

Parameters

- out_ptr – Pointer to a void* variable to receive the address of the memory block to be used as the Immortals Heap.
- out_size – Pointer to a size_t variable to receive the size of the memory block in bytes.

Returns 0
on success, -1 on error.

LLKERNEL: Multi-Sandbox

The following API corresponds to architecture version 8.5.0.

LLKERNEL_impl.h

Set of functions that the BSP must implement to manage memory allocation of dynamically installed Applications.

Defines

LLKERNEL_OK (0)

Returned value when function call has succeeded.

LLKERNEL_ERROR (-1)

Returned value when function call has failed.

LLKERNEL_RAM_AREA_ALIGNMENT (8)

The RAM area alignment constraint.

LLKERNEL_ROM_AREA_ALIGNMENT (16)

The ROM area alignment constraint.

LLKERNEL_FEATURE_INIT_ERROR_CORRUPTED_CONTENT (1)

Error code thrown by LLKERNEL_onFeatureInitializationError function when the Installed Feature content is corrupted. The checksum computed during the installation does not match the actual computed checksum.

LLKERNEL_FEATURE_INIT_ERROR_INCOMPATIBLE_KERNEL_WRONG_UID (2)

Error code thrown by LLKERNEL_onFeatureInitializationError function when the Installed Feature has not been built for this Kernel UID or a compatible Kernel UID.

LLKERNEL_FEATURE_INIT_ERROR_TOO_MANY_INSTALLED (3)

Error code thrown by LLKERNEL_onFeatureInitializationError function when there is no internal free slot to connect the Installed Feature.

LLKERNEL_FEATURE_INIT_ERROR_ALREADY_INSTALLED (4)

Error code thrown by LLKERNEL_onFeatureInitializationError function when the Installed Feature matches an already Installed Feature.

LLKERNEL_FEATURE_INIT_ERROR_INCOMPATIBLE_KERNEL_WRONG_ADDRESSES (5)

Error code thrown by LLKERNEL_onFeatureInitializationError function when the Installed Feature has been linked on a Kernel Address Table with different content.

LLKERNEL_FEATURE_INIT_ERROR_ROM_OVERLAP (6)

Error code thrown by LLKERNEL_onFeatureInitializationError function when the Installed Feature ROM area overlaps the ROM area of an already Installed Feature.

LLKERNEL_FEATURE_INIT_ERROR_RAM_OVERLAP (7)

Error code thrown by LLKERNEL_onFeatureInitializationError function when the Installed Feature RAM area overlaps the RAM area of an already Installed Feature.

LLKERNEL_FEATURE_INIT_ERROR_RAM_ADDRESS_CHANGED (8)

Error code thrown by LLKERNEL_onFeatureInitializationError function when the RAM address returned by LLKERNEL_getFeatureAddressRAM is not equals to the RAM address returned when the Feature has been linked on this Kernel.

LLKERNEL_FEATURE_INIT_ERROR_NULL_HANDLE (9)

Error code thrown by LLKERNEL_onFeatureInitializationError function when a previous call to LLKERNEL_getFeatureHandle returned 0 .

Functions

int32_t LLKERNEL_IMPL_allocateFeature (int32_t size_ROM, int32_t size_RAM)

Allocates a new Feature and reserves its ROM and RAM areas. If the Feature cannot be allocated, it must return 0 .

Warning: ROM and RAM areas reservation must take care of address alignment constraints required by LLKERNEL_getFeatureAddressROM and LLKERNEL_getFeatureAddressRAM.

Parameters

- size_ROM – the size in bytes to allocate in ROM
- size_RAM – the size in bytes to allocate in RAM

Returns a handle that uniquely identifies the allocated Feature, or 0 if the Feature could not be allocated.

void LLKERNEL_IMPL_freeFeature (int32_t handle)

Releases a Feature previously allocated using LLKERNEL_allocateFeature and frees its ROM and RAM areas.

Parameters

- handle – the Feature handle

int32_t LLKERNEL_IMPL_getAllocatedFeaturesCount (void)

Retrieves the count of currently installed Features in memory.

A Feature is considered installed in memory if it has been successfully allocated using the LLKERNEL_allocateFeature method and has not been freed with the LLKERNEL_freeFeature method. This count may include Features that were installed in a previous execution of this Kernel.

Upon calling this method, the Core Engine will retrieve Feature handles by invoking LLKERNEL_getFeatureHandle for each index within the range [0.. LLKERNEL_getAllocatedFeaturesCount-1)] .

See also:

LLKERNEL_getFeatureHandle

Returns

the current count of Features installed in memory, 0 if there are none.

int32_t LLKERNEL_IMPL_getFeatureHandle (int32_t index)

Retrieves the handle of a Feature that has been installed in memory.

This method is invoked by the Core Engine immediately following a call to LLKERNEL_getAllocatedFeaturesCount for each index within the range [0.. LLKERNEL_getAllocatedFeaturesCount-1] .

See also:

LLKERNEL_getAllocatedFeaturesCount

Parameters

- index – an index within the range [0..LLKERNEL_getAllocatedFeaturesCount-1] .

Returns

a unique handle that identifies the Feature at the specified index, or 0 if no valid handle is found.

void * LLKERNEL_IMPL_getFeatureAddressRAM (int32_t handle)

Gets the address of the allocated RAM area for the given Feature. The returned address must be *LLKERNEL_RAM_AREA_ALIGNMENT* bytes aligned.

Parameters

- handle – the Feature handle

Returns

the start address in bytes of the allocated RAM area, or NULL if the Feature handle is not valid.

void * LLKERNEL_IMPL_getFeatureAddressROM (int32_t handle)

Gets the address of the allocated ROM area for the given Feature. The returned address must be *LLKERNEL_ROM_AREA_ALIGNMENT* bytes aligned.

Parameters

- handle – the Feature handle

Returns

the start address in bytes of the allocated ROM area, or NULL if the Feature handle is not valid.

int32_t LLKERNEL_IMPL_copyToROM (void *dest_address_ROM, void *src_address, int32_t size)

Copies a sequence of bytes to ROM area. This function does not require that all bytes are really copied to the destination before it returns.

Parameters

- dest_address_ROM – the start address in ROM (destination)
- src_address – the start address of bytes to copy (source)
- size – the number of bytes to copy

Returns

LLKERNEL_OK on success, *LLKERNEL_ERROR* on error.

int32_t LLKERNEL_IMPL_flushCopyToROM (void)

This function is called once all calls to LLKERNEL_copyToROM are done for the currently allocated Feature. This function must block until all bytes are really copied to ROM area.

Returns

LLKERNEL_OK on success, *LLKERNEL_ERROR* on error.

int32_t LLKERNEL_IMPL_onFeatureInitializationError (int32_t handle, int32_t error_code)

This function is called during Kernel boot when the initialization of an allocated Feature failed.

Parameters

- handle – a Feature handle returned by LLKERNEL_getFeatureHandle
- error_code – the error code indicating the initialization error cause (see FEATURE_INIT_ERROR_* constants)

Returns

LLKERNEL_OK to skip the initialization of this Feature and continue with the next allocated Feature, or *LLKERNEL_ERROR* if the Kernel must abruptly stop.

LLKERNEL.h

Set of functions provided by the Core Engine that can be called by the BSP.

Functions

int32_t LLKERNEL_quantaConsumed (int32_t quanta)

Notifies the scheduler that the current thread has consumed the given quanta.

The scheduler will increment the execution counter for the context of the current thread's owner. This function allows explicitly specifying a quanta that reflects the computational complexity of a native function execution.

Parameters

- quanta – the quanta consumed

Returns

LLKERNEL_ERROR if this function is not called within the Core Engine task or if quanta is negative. Otherwise returns E_OK.

LLSP: Shielded Plug

Header Files

The implementation of the Shielded Plug assumes some support from the underlying RTOS. It is mainly related to provide some synchronization when reading / writing into Shielded Plug blocks.

- LLSP_IMPL_syncWriteBlockEnter and LLSP_IMPL_syncWriteBlockExit are used as a semaphore by RTOS tasks. When a task wants to write to a block, it "locks" this block until it has finished to write in it.
- LLSP_IMPL_syncReadBlockEnter and LLSP_IMPL_syncReadBlockExit are used as a semaphore by RTOS tasks. When a task wants to read a block, it "locks" this block until it is ready to release it.

  The *[SP] specification* provides a mechanism to force a task to wait until new data has been provided to a block. The implementation relies on functions LLSP_IMPL_wait and LLSP_IMPL_wakeup to block the current task and to reschedule it.

LLEXT_RES: External Resources Loader

Principle

This LLAPI allows to use the External Resource Loader. When installed, the External Resource Loader is notified when the Core Engine is not able to find a resource (an image, a file etc.) in the resources area linked with the Core Engine.

When a resource is not available, the Core Engine invokes the External Resource Loader in order to load an unknown resource. The External Resource Loader uses the LLAPI EXT_RES to let the BSP loads or not the expected resource. The implementation has to be able to load several files in parallel.

Header Files

One header file is provided:

- LLEXT_RES_impl.h

  Defines the set of functions that the BSP must implement to load some external resources.

LLCOMM: Serial Communications

Header Files

Four C header files are provided:

- LLCOMM_BUFFERED_CONNECTION_impl.h

  Defines the set of functions that the driver must implement to provide a Buffered connection

- LLCOMM_BUFFERED_CONNECTION.h

  Defines the set of functions provided by ECOM Comm that can be called by the driver (or other C code) when using a Buffered connection

- LLCOMM_CUSTOM_CONNECTION_impl.h

  Defines the set of functions that the driver must implement to provide a Custom connection

- LLCOMM_CUSTOM_CONNECTION.h

  Defines the set of functions provided by ECOM Comm that can be called by the driver (or other C code) when using a Custom connection

LLUI_INPUT: Input

LLUI_INPUT API is composed of the following files:

- the file LLUI_INPUT_impl.h that defines the functions to be implemented
- the file LLUI_INPUT.h that provides the functions for sending events

Implementation

LLUI_INPUT_IMPL_initialize is the first function called by the input engine, and it may be used to initialize the underlying devices and bind them to event generator IDs.

LLUI_INPUT_IMPL_enterCriticalSection and LLUI_INPUT_IMPL_exitCriticalSection need to provide the Input Engine with a critical section mechanism for synchronizing devices when sending events to the internal event queue. The mechanism used to implement the synchronization will depend on the VEE Port configuration (with or without RTOS), and whether or not events are sent from an interrupt context.

LLUI_INPUT_IMPL_getInitialStateValue allows the input stack to get the current state for devices connected to the MicroUI States event generator, such as switch selector, coding wheels, etc.

Sending Events

The LLUI_INPUT API provides two generic functions for a C driver to send data to its associated event generator:

- LLUI_INPUT_sendEvent : Sends a 32-bit event to a specific event generator, specified by its ID. If the input buffer is full, the event is not added, and the function returns LLUI_INPUT_NOK ; otherwise it returns LLUI_INPUT_OK .
- LLUI_INPUT_sendEvents : Sends a frame constituted by several 32-bit events to a specific event generator, specified by its ID. If the input buffer cannot receive the whole data, the frame is not added, and the function returns LLUI_INPUT_NOK ; otherwise it returns LLUI_INPUT_OK .

Events will be dispatched to the associated event generator that will be responsible for decoding them (see *Generic Event Generators*).

The UI extension provides an implementation for each of MicroUI's built-in event generators. Each one has dedicated functions that allows a driver to send them structured data without needing to understand the underlying protocol to encode/decode the data. *The following table* shows the functions provided to send structured events to the predefined event generators:

Table 87: LLUI_INPUT API for predefined event generators

Function name	Default event generator kind[Page 1380, 1]	Comments
LLUI_INPUT_sendCommandEvent	Command	Constants are provided that define all standard MicroUI commands [MUI].
LLUI_INPUT_sendButtonPressedEvent LLUI_INPUT_sendButtonReleasedEvent LLUI_INPUT_sendButtonRepeatedEvent LLUI_INPUT_sendButtonLongEvent	Buttons	In the case of chronological sequences (for example, a RELEASE that may occur only after a PRESSED), it is the responsibility of the driver to ensure the integrity of such sequences.
LLUI_INPUT_sendPointerPressedEvent LLUI_INPUT_sendPointerReleasedEvent LLUI_INPUT_sendPointerMovedEvent	Pointer	In the case of chronological sequences (for example, a RELEASE that may occur only after a PRESSED), it is the responsibility of the driver to ensure the integrity of such sequences. Depending on whether a button of the pointer is pressed while moving, a DRAG and/or a MOVE MicroUI event is generated.
LLUI_INPUT_sendStateEvent	States	The initial value of each state machine (of a States) is retrieved by a call to LLUI_INPUT_IMPL_getInitialStateValue that must be implemented by the device. Alternatively, the initial value can be specified in the XML static configuration.
LLUI_INPUT_sendTouchPressedEvent LLUI_INPUT_sendTouchReleasedEvent LLUI_INPUT_sendTouchMovedEvent	Pointer	In the case of chronological sequences (for example, a RELEASE that may only occur after a PRESSED), it is the responsibility of the driver to ensure the integrity of such sequences. These APIs will generate a DRAG MicroUI event instead of a MOVE while they represent a touch pad over a display.

[1] The implementation class is a subclass of the MicroUI class of the column.

Event Buffer

Functions LLUI_INPUT_IMPL_log_xxx allow logging the use of event buffer. Implementation of these LLAPIs is already available on the Central Repository (LLUI_INPUT_LOG_impl.c). This implementation is using an array to add some metadata to each event. This metadata is used when the BSP is calling LLUI_INPUT_dump() . When no implementation is included in the BSP, the call to LLUI_INPUT_dump() has no effect (no available logger).

LLUI_DISPLAY: Display

Initialization

Each Graphics Engine gets initialized by calling the function LLUI_DISPLAY_IMPL_initialize : It asks its display driver to initialize itself. The implementation function has to fill the given structure LLUI_DISPLAY_SInitData . This structure allows to retrieve the size of the virtual and physical screen, the back buffer address (where MicroUI is drawing). The implementation has to *give* two binary semaphores.

Image Heap

The display driver must reserve a runtime memory buffer for creating dynamic images when using MicroUI ResourceImage and BufferedImage classes methods. The display driver may choose to reserve an empty buffer. Thus, calling MicroUI methods will result in a MicroUIException exception.

The section name is .bss.microui.display.imagesHeap .

Functions LLUI_DISPLAY_IMPL_imageHeapXXX allow to control the image buffers allocation in the image heap. Implementation of these LLAPIs is already available on the Central Repository (LLUI_DISPLAY_HEAP_impl.c). This implementation is using a best fit allocator. It can be updated to log the allocations, the remaining space, etc. When no implementation is included in the BSP, the default Graphics Engine's allocator (a best fit allocator) is used.

External Font Heap

The display driver must reserve a runtime memory buffer for loading external fonts (fonts located outside CPU addresses ranges). The display driver may choose to reserve an empty buffer. Thus, calling MicroUI Font methods will result in empty drawings of some characters.

The section name is .bss.microui.display.externalFontsHeap .

Flush and Synchronization

The back buffer (graphics buffer) address defined in the Initialization function is the address for the very first drawing. The content of this buffer is flushed to the external display memory by the function LLUI_DISPLAY_flush. The parameters define one or several rectangular regions of the content that have changed during the last drawing action and that must be flushed to the front buffer (dirty area). This function should be atomic: the implementation has to start another task or a hardware device (often a DMA) to perform the flush.

As soon as the Application performs a new drawing, the Graphics Engine locks the thread. It will automatically be unlocked when the BSP calls LLUI_DISPLAY_setBackBuffer at the end of the flush.

Display Characteristics

Function LLUI_DISPLAY_IMPL_isColor directly implements the method from the MicroUI Display class of the same name. The default implementation always returns true when the number of bits per pixel is higher than 4.

Function LLUI_DISPLAY_IMPL_getNumberOfColors directly implements the method from the MicroUI Display class of the same name. The default implementation returns a value according to the number of bits by pixel, without taking into consideration the alpha bit(s).

Function LLUI_DISPLAY_IMPL_isDoubleBuffered directly implements the method from the MicroUI Display class of the same name. The default implementation returns true. When LLAPI implementation targets a display in direct mode, this function must be implemented and return false.

Contrast

LLUI_DISPLAY_IMPL_setContrast and LLUI_DISPLAY_IMPL_getContrast are called to set/get the current display contrast intensity. The default implementations don't manage the contrast.

BackLight

LLUI_DISPLAY_IMPL_hasBacklight indicates whether the display has backlight capabilities.

LLUI_DISPLAY_IMPL_setBacklight and LLUI_DISPLAY_IMPL_getBacklight are called to set/get the current display backlight intensity.

Color Conversions

The following functions are only useful (and called) when the display is not a standard display, see *Pixel Structure*.

LLUI_DISPLAY_IMPL_convertARGBColorToDisplayColor is called to convert a 32-bit ARGB MicroUI color in 0xAARRGGBB format into the "driver" display color.

LLUI_DISPLAY_IMPL_convertDisplayColorToARGBColor is called to convert a display color to a 32-bit ARGB MicroUI color.

CLUT

The function LLUI_DISPLAY_IMPL_prepareBlendingOfIndexedColors is called when drawing an image with indexed color. See *CLUT* to have more information about indexed images.

Image Decoders

The API LLUI_DISPLAY_IMPL_decodeImage allows to add some additional *image decoders*.

LLUI_LED: LEDs

Principle

The LEDs engine provides Low Level APIs for connecting LED drivers. The file LLUI_LED_impl.h , which comes with the LEDs engine, defines the API headers to be implemented.

Initialization

The first function called is LLUI_LED_IMPL_initialize , which allows the driver to initialize all LED devices. This method must return the available number of LEDs. Each LED has a unique identifier. The first LED has the ID 0, and the last has the ID NbLEDs – 1.

This UI extension provides support to efficiently implement the set of methods that interact with the LEDs provided by a device. Below are the relevant C functions:

- LLUI_LED_IMPL_getIntensity : Get the intensity of a specific LED using its ID.
- LLUI_LED_IMPL_setIntensity : Set the intensity of an LED using its ID.

LLVG: VectorGraphics

Principle

The *VG Pack* provides a Low Level API for initializing the Vector Graphics engine. The file LLVG_impl.h , which comes with the VG Pack, defines the API headers to be implemented.

Initialization

The function LLVG_IMPL_initialize is the first native function called by the MicroVG implementation. It allows to initialize all C components: GPU initialization, Font engine, heap management, etc.

LLVG_MATRIX: Matrix

Principle

The *Matrix module* provides Low Level APIs for manipulating matrices. The file LLVG_MATRIX_impl.h , which comes with the Matrix module, defines the API headers to be implemented.

Implementation

The matrix functions are divided in four groups:

1. identity and copy: fill an identity matrix or copy a matrix to another one.
2. setXXX: erase the content of the matrix by an operation (translate, rotation, scaling, concatenate).
3. xxx (no prefix): perform an operation with the matrix as first argument: M' = M * xxx(x, y) where xxx is the operation (translate, rotation, scaling, concatenate).
4. postXXX: perform an operation with the matrix as second argument: M' = xxx(x, y) * M where xxx is the operation (translate, rotation, scaling, concatenate).

LLVG_PATH: Vector Path

Principle

The *Path module* provides Low Level APIs for creating paths in target specific format. The file LLVG_PATH_impl.h , which comes with the Path module, defines the API headers to be implemented. The file LLVG_PAINTER_impl.h defines the API headers to be implemented to draw the paths (with a color or a gradient).

Creation

The header file LLVG_PATH_impl.h allows to convert a MicroVG library format path in a buffer that represents the same vectorial path in the target specific format (generally GPU format).

The first function called is LLVG_PATH_IMPL_initializePath , which allows the implementation to initialize the path buffer. The buffer is allocated in the Managed Heap and its size is fixed by the MicroVG implementation. When the buffer is too small for the target specific format, the implementation has to return the expected buffer size instead of the keyword LLVG_SUCCESS .

The next steps consist in appending some commands in the path buffer. The command encoding depends on the target specific format. When the buffer is too small to add the new command, the implementation has to return a value that indicates the number of bytes the array must be enlarged with.

List of commands:

- LLVG_PATH_CMD_CLOSE : MicroVG “CLOSE” command.
- LLVG_PATH_CMD_MOVE : MicroVG “MOVE ABS” command.

- LLVG_PATH_CMD_MOVE_REL : MicroVG “MOVE REL” command.
- LLVG_PATH_CMD_LINE : MicroVG “LINE ABS” command.
- LLVG_PATH_CMD_LINE_REL : MicroVG “LINE REL” command.
- LLVG_PATH_CMD_QUAD : MicroVG “QUAD ABS” command.
- LLVG_PATH_CMD_QUAD_REL : MicroVG “QUAD REL” command.
- LLVG_PATH_CMD_CUBIC : MicroVG “CUBIC ABS” command.
- LLVG_PATH_CMD_CUBIC_REL : MicroVG “CUBIC REL” command.

List of operations:

- LLVG_PATH_IMPL_appendPathCommand1 : Adds a command with 1 point parameter in the array.
- LLVG_PATH_IMPL_appendPathCommand2 : Adds a command with 2 points parameter in the array.
- LLVG_PATH_IMPL_appendPathCommand3 : Adds a command with 3 points parameter in the array.

A path is automatically closed by the MicroVG implementation (by adding the command LLVG_PATH_CMD_CLOSE). A path can be reopened (function LLVG_PATH_IMPL_reopenPath), that consists in removing the last added command (LLVG_PATH_CMD_CLOSE command) from the buffer.

Drawing

The header file LLVG_PAINTER_impl.h provides the functions called by the Application via VectorGraphicsPainter to draw a path.

- A path can be drawn with a 32-bit color (ARGB8888): LLVG_PAINTER_IMPL_drawPath .
- A path can be drawn with a *linear gradient*: LLVG_PAINTER_IMPL_drawGradient .

The drawing destination is symbolized by a MicroUI GraphicsContext: a pointer to a MICROUI_GraphicsContext instance. Like MicroUI Painter natives, the implementation has to *synchronize the drawings* with the MicroUI Graphics Engine.

LLVG_GRADIENT: Vector Linear Gradient

Principle

The *Gradient module* provides Low Level APIs for creating linear gradients in target specific format. The file LLVG_GRADIENT_impl.h , which comes with the Gradient module, defines the API headers to be implemented.

Implementation

Only one function has to be implemented: LLVG_GRADIENT_IMPL_initializeGradient. It consists in encoding the MicroVG LinearGradient in a buffer that represents the linear gradient in target specific format (generally GPU format).

This function allows the implementation to initialize the gradient buffer. The buffer is allocated in the Managed Heap and its size is fixed by the MicroVG implementation. When the buffer is too small for the target specific format, the implementation has to return the expected buffer size instead of the keyword LLVG_SUCCESS.

LLVG_FONT: Vector Font

Principle

The *Font module* provides Low Level APIs for decoding fonts (LLVG_FONT_impl.h) and rendering texts (LLVG_PAINTER_impl.h). Both header files, which come with the Font module, define the API headers to be implemented.

Initialization

The first function called is LLVG_FONT_IMPL_load_font, which allows the driver to open a font file from its name. This function takes a parameter to configure the text rendering engine:

- Simple layout: uses the glyph advance metrics and the font kerning table.
- Complex layout: uses the font GPOS and GSUB tables.

See VectorFont for more information.

The implementation must manage its own heap to keep the font opened. The font's data are disposed by a call to LLVG_FONT_IMPL_dispose.

Font Characteristics

The other functions in LLVG_FONT_impl.h consist in retrieving some font characteristics according a text and a font size: string width, string height, baseline, etc.

See VectorFont for more information.

Drawing

The header file LLVG_PAINTER_impl.h provides the functions called by the Application via VectorGraphicsPainter to draw a path.

- A string can be drawn with a 32-bit color (ARGB8888): LLVG_PAINTER_IMPL_drawString.
- A string can be drawn with a *linear gradient*: LLVG_PAINTER_IMPL_drawStringGradient.
- A string can be draw on a circle: LLVG_PAINTER_IMPL_drawStringOnCircle and LLVG_FONT_PAINTER_IMPL_drawStringOnCircleGradient.

The drawing destination is symbolized by a MicroUI GraphicsContext: a pointer to a MICROUI_GraphicsContext instance. Like MicroUI Painter natives, the implementation has to *synchronize the drawings* with the MicroUI Graphics Engine.

LLNET: Network

Header Files

Several header files are provided:

- LLNET_CHANNEL_impl.h

  Defines a set of functions that the BSP must implement to initialize the Net native component. It also defines some configuration operations to setup a network connection.

- LLNET_SOCKETCHANNEL_impl.h

  Defines a set of functions that the BSP must implement to create, connect and retrieve information on a network connection.

- LLNET_STREAMSOCKETCHANNEL_impl.h

  Defines a set of functions that the BSP must implement to do some I/O operations on connection oriented socket (TCP). It also defines function to put a server connection in accepting mode (waiting for a new client connection).

- LLNET_DATAGRAMSOCKETCHANNEL_impl.h

  Defines a set of functions that the BSP must implement to do some I/O operations on connectionless oriented socket (UDP).

- LLNET_DNS_impl.h

  Defines a set of functions that the BSP must implement to request host IP address associated to a host name or to request Domain Name Service (DNS) host IP addresses setup in the underlying system.

- LLNET_NETWORKADDRESS_impl.h

  Defines a set of functions that the BSP must implement to convert string IP address or retrieve specific IP addresses (lookup, localhost or loopback IP address).

- LLNET_NETWORKINTERFACE_impl.h

  Defines a set of functions that the BSP must implement to retrieve information on a network interface (MAC address, interface link status, etc.).

LLNET_SSL: SSL

Header Files

Three header files are provided:

- LLNET_SSL_CONTEXT_impl.h

  Defines a set of functions that the BSP must implement to create a SSL Context and to load CA (Certificate Authority) certificates as trusted certificates.

- LLNET_SSL_SOCKET_impl.h

  Defines a set of functions that the BSP must implement to initialize the SSL native components, to create an underlying SSL Socket and to initiate a SSL session handshake. It also defines some I/O operations such as LLNET_SSL_SOCKET_IMPL_write or LLNET_SSL_SOCKET_IMPL_read used for encrypted data exchange between the client and the server.

- LLNET_SSL_X509_CERT_impl.h

  Defines a function named LLNET_SSL_X509_CERT_IMPL_parse for certificate parsing. This function checks if a given certificate is an X.509 digital certificate and returns its encoded format type : Distinguished Encoding Rules (DER) or Privacy-Enchanced Mail (PEM).

LLECOM_NETWORK: Network Interfaces

Header Files

One header file is provided:

- LLECOM_NETWORK_impl.h

  Defines the set of functions that the BSP must implement to manage and configure and TCP/IP network interfaces.

LLECOM_WIFI: Wi-Fi Management

Header Files

One header file is provided:

- LLECOM_WIFI_impl.h

  Defines the set of functions that the BSP must implement to manage and configure Wi-FI access points.

LLBLUETOOTH: Bluetooth

Header Files

Two header files are provided:

- LLBLUETOOTH_defines.h

  Defines constants and types which are used by the functions to implement.

- LLBLUETOOTH_impl.h

  Defines the set of functions that the BSP must implement to manage and configure and Bluetooth module.

LLAUDIO: Audio

Header Files

Three header files is provided:

- LLAUDIO_defines.h

  Defines constants and types which are used by the functions to implement.

- LLAUDIO_RECORD_impl.h

  Defines the set of functions that the BSP must implement for audio recording.

- LLAUDIO_TRACK_impl.h

  Defines the set of functions that the BSP must implement for audio playback.

LLML: MicroAI

Header Files

One header file is provided:

- LLML_impl.h

  Defines the set of functions that the BSP must implement to load a model file, set the input data and read output data. This API also implements a number of helper functions to get the model characteristics such as tensor sizes, quantization parameters, etc…

LLEVENT: Event Queue

Header Files

Two header files are provided:

- LLEVENT_impl.h

  Defines the set of functions that the BSP must implement to manage, offer/handle events from the Event Queue.

- LLEVENT.h

  Defines the set of functions that the BSP must implement to use the Event Queue from the native side.

LLFS: File System

Header Files

Two C header files are provided:

- LLFS_impl.h

Defines a set of functions that the BSP must implement to initialize the FS native component. It also defines some functions to manage files, directories and retrieve information about the underlying File System (free space, total space, etc.).

- LLFS_File_impl.h

  Defines a set of functions that the BSP must implement to do some I/O operations on files (open, read, write, close, etc.).

LLGNSS: GNSS (Global Navigation Satellite System)

Header Files

One header file is provided:

- LLGNSS_impl.h

  Defines the set of functions that the BSP must implement to manage and configure GNSS engine.

LLHAL: Hardware Abstraction Layer

Header Files

One header file is provided:

- LLHAL_impl.h

  Defines the set of functions that the BSP must implement to configure and drive some MCU GPIO.

LLDEVICE: Device Information

Header Files

One C header file is provided:

- LLDEVICE_impl.h

  Defines a set of functions that the BSP must implement to get the Architecture name and unique device identifier.

LLWATCHDOG_TIMER: Watchdog Timer

Header Files

One C header file is provided:

- LLWATCHDOG_TIMER_impl.h

  Defines a set of functions that can be used at BSP level if required.

  This C header file contains functions to implement:

- watchdog_timer_helper.h

  Defines a set of functions that the BSP must implement to link the VEE Port watchdog timer to the Watchdog Timer library.

LLSEC: Security

Header Files

Several C header files are provided:

- LLSEC_CIPHER_impl.h

  Defines a set of functions that must be implemented by the BSP in order to decrypt and encrypt data using cryptographic ciphers.

- LLSEC_CONSTANTS.h

  Defines constants for certificates encoding formats.

- LLSEC_DIGEST_impl.h

  Defines a set of functions that must be implemented by the BSP in order to support message digest algorithms such as SHA-1 or SHA-256.

- LLSEC_ERRORS.h

  Defines the Security API error return codes.

- LLSEC_KEY_FACTORY_impl.h

  Defines a set of functions that must be implemented by the BSP in order to get keys informations such as algorithm or encoded form.

- LLSEC_KEY_PAIR_GENERATOR_impl.h

  Defines a set of functions that must be implemented by the BSP in order to generate private/public key pairs.

- LLSEC_MAC_impl.h

  Defines a set of functions that must be implemented by the BSP in order to support MAC algorithms.

- LLSEC_PRIVATE_KEY_impl.h

  Defines a set of functions that must be implemented by the BSP in order to encode private keys in DER format.

- LLSEC_PUBLIC_KEY_impl.h

  Defines a set of functions that must be implemented by the BSP in order to encode public keys.

- LLSEC_RANDOM_impl.h

  Defines a set of functions that must be implemented by the BSP in order to generate random data.

- LLSEC_SIG_impl.h

  Defines a set of functions that must be implemented by the BSP in order to support signatures functionalities.

- LLSEC_X509_CERT_impl.h

  Defines a set of functions that must be implemented by the BSP in order to manage X509 certificates operations like getting the public key, extracting the issuer, etc.

LLTRACE: Trace

The following API corresponds to architecture version 8.5.0.

Principle

The *Trace module* provides Low Level APIs to produce lightweight events for debugging and monitoring purposes. The file LLTRACE_impl.h , which comes with the Core Engine, defines the API headers to be implemented.

LLTRACE_impl.h

Set of functions that the BSP must implement to launch and schedule the Core Engine.

Functions

void LLTRACE_IMPL_start (void)

Starts to record the events.

By default, the trace system is stopped.

This function must not be called by the application. Use the corresponding function TRACE_*() .

void LLTRACE_IMPL_stop (void)

Stops to record the events. When the record system is stopped, any call to the LLTRACE_IMPL_record_event_* functions must not record any events.

By default, the trace system is stopped.

This function must not be called by the application. Use the corresponding function TRACE_*() .

bool LLTRACE_IMPL_is_started (void)

Tests whether the trace system is started or not.

This function must not be called by the application. Use the corresponding function TRACE_*() .

Returns
true if the trace system is started, false otherwise.

int32_t LLTRACE_IMPL_declare_event_group (const char *group_name, int32_t nb_event_types)

Declare a new group of events with the given name. The created group will define nb_event_types types of events. The event IDs available for this group are from 0 to nb_event_types -1.

This function must not be called by the application. Use the corresponding function TRACE_*() .

Parameters

- group_name – name that identifies the group. Must not be null.
- nb_event_types – maximum number of event types available for the group.

Returns

the ID of the created group or -1 on error.

void LLTRACE_IMPL_record_event_void (int32_t group_id, int32_t event_id)

Record an event for the given group. The given event ID must be valid for the specified group (i.e., between 0 and nb_event_types -1 included where nb_event_types is the value given to LLTRACE_IMPL_declare_event_group()).

The record is done only if the trace system is started (see LLTRACE_IMPL_start()).

This function must not be called by the application. Use the corresponding function TRACE_*() .

Parameters

- group_id – ID of the group returned by LLTRACE_IMPL_declare_event_group() .
- event_id – ID of the event.

void LLTRACE_IMPL_record_event_u32 (int32_t group_id, int32_t event_id, uint32_t value1)

Record an event for the given group. The given event ID must be valid for the specified group (i.e., between 0 and nb_event_types -1 included where nb_event_types is the value given to LLTRACE_IMPL_declare_event_group()).

The record is done only if the trace system is started (see LLTRACE_IMPL_start()).

This function must not be called by the application. Use the corresponding function TRACE_*() .

Parameters

- group_id – ID of the group returned by LLTRACE_IMPL_declare_event_group() .
- event_id – ID of the event.
- value1 – custom value for the event.

void LLTRACE_IMPL_record_event_u32x2 (int32_t group_id, int32_t event_id, uint32_t value1, uint32_t value2)

Record an event for the given group. The given event ID must be valid for the specified group (i.e., between 0 and nb_event_types -1 included where nb_event_types is the value given to LLTRACE_IMPL_declare_event_group()).

The record is done only if the trace system is started (see LLTRACE_IMPL_start()).

This function must not be called by the application. Use the corresponding function TRACE_*() .

Parameters

- group_id – ID of the group returned by LLTRACE_IMPL_declare_event_group() .
- event_id – ID of the event.
- value1 – custom value for the event.
- value2 – custom value for the event.

void LLTRACE_IMPL_record_event_u32x3 (int32_t group_id, int32_t event_id, uint32_t value1, uint32_t value2, uint32_t value3)

Record an event for the given group. The given event ID must be valid for the specified group (i.e., between 0 and nb_event_types -1 included where nb_event_types is the value given to LLTRACE_IMPL_declare_event_group()).

The record is done only if the trace system is started (see LLTRACE_IMPL_start()).

This function must not be called by the application. Use the corresponding function TRACE_*() .

Parameters

- group_id – ID of the group returned by LLTRACE_IMPL_declare_event_group() .
- event_id – ID of the event.
- value1 – custom value for the event.
- value2 – custom value for the event.
- value3 – custom value for the event.

void LLTRACE_IMPL_record_event_u32x4 (int32_t group_id, int32_t event_id, uint32_t value1, uint32_t value2, uint32_t value3, uint32_t value4)

Record an event for the given group. The given event ID must be valid for the specified group (i.e., between 0 and nb_event_types -1 included where nb_event_types is the value given to LLTRACE_IMPL_declare_event_group()).

The record is done only if the trace system is started (see LLTRACE_IMPL_start()).

This function must not be called by the application. Use the corresponding function TRACE_*() .

Parameters

- group_id – ID of the group returned by LLTRACE_IMPL_declare_event_group() .
- event_id – ID of the event.
- value1 – custom value for the event.
- value2 – custom value for the event.
- value3 – custom value for the event.
- value4 – custom value for the event.

void LLTRACE_IMPL_record_event_u32x5 (int32_t group_id, int32_t event_id, uint32_t value1, uint32_t value2, uint32_t value3, uint32_t value4, uint32_t value5)

Record an event for the given group. The given event ID must be valid for the specified group (i.e., between 0 and nb_event_types -1 included where nb_event_types is the value given to LLTRACE_IMPL_declare_event_group()).

The record is done only if the trace system is started (see LLTRACE_IMPL_start()).

This function must not be called by the application. Use the corresponding function TRACE_*() .

Parameters

- group_id – ID of the group returned by LLTRACE_IMPL_declare_event_group() .
- event_id – ID of the event.
- value1 – custom value for the event.
- value2 – custom value for the event.
- value3 – custom value for the event.
- value4 – custom value for the event.
- value5 – custom value for the event.

void LLTRACE_IMPL_record_event_u32x6 (int32_t group_id, int32_t event_id, uint32_t value1, uint32_t value2, uint32_t value3, uint32_t value4, uint32_t value5, uint32_t value6)

Record an event for the given group. The given event ID must be valid for the specified group (i.e., between 0 and nb_event_types -1 included where nb_event_types is the value given to LLTRACE_IMPL_declare_event_group()).

The record is done only if the trace system is started (see LLTRACE_IMPL_start()).

This function must not be called by the application. Use the corresponding function TRACE_*() .

Parameters

- group_id – ID of the group returned by LLTRACE_IMPL_declare_event_group() .
- event_id – ID of the event.
- value1 – custom value for the event.
- value2 – custom value for the event.
- value3 – custom value for the event.
- value4 – custom value for the event.
- value5 – custom value for the event.
- value6 – custom value for the event.

void LLTRACE_IMPL_record_event_u32x7 (int32_t group_id, int32_t event_id, uint32_t value1, uint32_t value2, uint32_t value3, uint32_t value4, uint32_t value5, uint32_t value6, uint32_t value7)

Record an event for the given group. The given event ID must be valid for the specified group (i.e., between 0 and nb_event_types -1 included where nb_event_types is the value given to LLTRACE_IMPL_declare_event_group()).

The record is done only if the trace system is started (see LLTRACE_IMPL_start()).

This function must not be called by the application. Use the corresponding function TRACE_*() .

Parameters

- group_id – ID of the group returned by LLTRACE_IMPL_declare_event_group() .
- event_id – ID of the event.
- value1 – custom value for the event.
- value2 – custom value for the event.
- value3 – custom value for the event.
- value4 – custom value for the event.
- value5 – custom value for the event.
- value6 – custom value for the event.
- value7 – custom value for the event.

void LLTRACE_IMPL_record_event_u32x8 (int32_t group_id, int32_t event_id, uint32_t value1, uint32_t value2, uint32_t value3, uint32_t value4, uint32_t value5, uint32_t value6, uint32_t value7, uint32_t value8)

Record an event for the given group. The given event ID must be valid for the specified group (i.e., between 0 and nb_event_types -1 included where nb_event_types is the value given to LLTRACE_IMPL_declare_event_group()).

The record is done only if the trace system is started (see LLTRACE_IMPL_start()).

This function must not be called by the application. Use the corresponding function TRACE_*() .

Parameters

- group_id – ID of the group returned by LLTRACE_IMPL_declare_event_group() .
- event_id – ID of the event.
- value1 – custom value for the event.
- value2 – custom value for the event.
- value3 – custom value for the event.
- value4 – custom value for the event.
- value5 – custom value for the event.
- value6 – custom value for the event.
- value7 – custom value for the event.
- value8 – custom value for the event.

void LLTRACE_IMPL_record_event_u32x9 (int32_t group_id, int32_t event_id, uint32_t value1, uint32_t value2, uint32_t value3, uint32_t value4, uint32_t value5, uint32_t value6, uint32_t value7, uint32_t value8, uint32_t value9)

Record an event for the given group. The given event ID must be valid for the specified group (i.e., between 0 and nb_event_types -1 included where nb_event_types is the value given to LLTRACE_IMPL_declare_event_group()).

The record is done only if the trace system is started (see LLTRACE_IMPL_start()).

This function must not be called by the application. Use the corresponding function TRACE_*() .

Parameters

- group_id – ID of the group returned by LLTRACE_IMPL_declare_event_group() .
- event_id – ID of the event.
- value1 – custom value for the event.
- value2 – custom value for the event.
- value3 – custom value for the event.
- value4 – custom value for the event.
- value5 – custom value for the event.
- value6 – custom value for the event.
- value7 – custom value for the event.
- value8 – custom value for the event.
- value9 – custom value for the event.

void LLTRACE_IMPL_record_event_u32x10 (int32_t group_id, int32_t event_id, uint32_t value1, uint32_t value2, uint32_t value3, uint32_t value4, uint32_t value5, uint32_t value6, uint32_t value7, uint32_t value8, uint32_t value9, uint32_t value10)

Record an event for the given group. The given event ID must be valid for the specified group (i.e., between 0 and nb_event_types -1 included where nb_event_types is the value given to LLTRACE_IMPL_declare_event_group()).

The record is done only if the trace system is started (see LLTRACE_IMPL_start()).

This function must not be called by the application. Use the corresponding function TRACE_*() .

Parameters

- group_id – ID of the group returned by LLTRACE_IMPL_declare_event_group() .
- event_id – ID of the event.
- value1 – custom value for the event.
- value2 – custom value for the event.
- value3 – custom value for the event.
- value4 – custom value for the event.
- value5 – custom value for the event.
- value6 – custom value for the event.
- value7 – custom value for the event.
- value8 – custom value for the event.
- value9 – custom value for the event.
- value10 – custom value for the event.

void LLTRACE_IMPL_record_event_end (int32_t group_id, int32_t event_id)

Record the end of the execution of an event for the given group. Call this function to trace the duration of an event previously record with one of the LLTRACE_IMPL_record_event_*()

void LLTRACE_IMPL_record_event_end_u32 (int32_t group_id, int32_t event_id, uint32_t value1)

Record the end of the execution of an event for the given group. Call this function to trace the duration of an event previously record with one of the LLTRACE_IMPL_record_event_()

LLTRACE.h

Set of functions provided by Core Engine that can be called by the BSP to generate event traces.

Functions

void LLTRACE_start (void)

Starts to record the events.

By default, the trace system is stopped.

void LLTRACE_stop (void)

Stops to record the events. When the record system is stopped, any call to the LLTRACE_record_event_* functions will not record any events.

By default, the trace system is stopped.

bool LLTRACE_is_started (void)

Tests whether the trace system is started or not.

Returns
true if the trace system is started, false otherwise.

int32_t LLTRACE_declare_event_group (const char *group_name, int32_t nb_event_types)

Declare a new group of events with the given name. The created group defines nb_event_types types of events. The event IDs available for this group are from 0 to nb_event_types -1.

Parameters

- group_name – name that identifies the group. Must not be null.
- nb_event_types – maximum number of event types available for the group.

Returns
the ID of the created group or -1 on error.

void LLTRACE_record_event_void (int32_t group_id, int32_t event_id)

Record an event for the given group. The given event ID must be valid for the specified group (i.e., between 0 and nb_event_types -1 included where nb_event_types is the value given to LLTRACE_declare_event_group()).

The record is done only if the trace system is started (see LLTRACE_start()).

Parameters

- group_id – ID of the group returned by LLTRACE_declare_event_group() .
- event_id – ID of the event.

void LLTRACE_record_event_u32 (int32_t group_id, int32_t event_id, uint32_t value0)

Record an event for the given group. The given event ID must be valid for the specified group (i.e., between 0 and nb_event_types -1 included where nb_event_types is the value given to LLTRACE_declare_event_group()).

The record is done only if the trace system is started (see LLTRACE_start()).

Parameters

- group_id – ID of the group returned by LLTRACE_declare_event_group() .
- event_id – ID of the event.
- value1 – custom value for the event.

void LLTRACE_record_event_u32x2 (int32_t group_id, int32_t event_id, uint32_t value1, uint32_t value2)

Record an event for the given group. The given event ID must be valid for the specified group (i.e., between 0 and nb_event_types -1 included where nb_event_types is the value given to LLTRACE_declare_event_group()).

The record is done only if the trace system is started (see LLTRACE_start()).

Parameters

- group_id – ID of the group returned by LLTRACE_declare_event_group() .
- event_id – ID of the event.
- value1 – custom value for the event.
- value2 – custom value for the event.

void LLTRACE_record_event_u32x3 (int32_t group_id, int32_t event_id, uint32_t value1, uint32_t value2, uint32_t value3)

Record an event for the given group. The given event ID must be valid for the specified group (i.e., between 0 and nb_event_types -1 included where nb_event_types is the value given to LLTRACE_declare_event_group()).

The record is done only if the trace system is started (see LLTRACE_start()).

Parameters

- group_id – ID of the group returned by LLTRACE_declare_event_group() .
- event_id – ID of the event.
- value1 – custom value for the event.
- value2 – custom value for the event.
- value3 – custom value for the event.

void LLTRACE_record_event_u32x4 (int32_t group_id, int32_t event_id, uint32_t value1, uint32_t value2, uint32_t value3, uint32_t value4)

Record an event for the given group. The given event ID must be valid for the specified group (i.e., between 0 and nb_event_types -1 included where nb_event_types is the value given to LLTRACE_declare_event_group()).

The record is done only if the trace system is started (see LLTRACE_start()).

Parameters

- group_id – ID of the group returned by LLTRACE_declare_event_group() .
- event_id – ID of the event.
- value1 – custom value for the event.
- value2 – custom value for the event.
- value3 – custom value for the event.
- value4 – custom value for the event.

void LLTRACE_record_event_u32x5 (int32_t group_id, int32_t event_id, uint32_t value1, uint32_t value2, uint32_t value3, uint32_t value4, uint32_t value5)

Record an event for the given group. The given event ID must be valid for the specified group (i.e., between 0 and nb_event_types -1 included where nb_event_types is the value given to LLTRACE_declare_event_group()).

The record is done only if the trace system is started (see LLTRACE_start()).

Parameters

- group_id – ID of the group returned by LLTRACE_declare_event_group() .
- event_id – ID of the event.
- value1 – custom value for the event.
- value2 – custom value for the event.
- value3 – custom value for the event.
- value4 – custom value for the event.
- value5 – custom value for the event.

void LLTRACE_record_event_u32x6 (int32_t group_id, int32_t event_id, uint32_t value1, uint32_t value2, uint32_t value3, uint32_t value4, uint32_t value5, uint32_t value6)

Record an event for the given group. The given event ID must be valid for the specified group (i.e., between 0 and nb_event_types -1 included where nb_event_types is the value given to LLTRACE_declare_event_group()).

The record is done only if the trace system is started (see LLTRACE_start()).

Parameters

- group_id – ID of the group returned by LLTRACE_declare_event_group() .
- event_id – ID of the event.
- value1 – custom value for the event.
- value2 – custom value for the event.
- value3 – custom value for the event.
- value4 – custom value for the event.
- value5 – custom value for the event.
- value6 – custom value for the event.

void LLTRACE_record_event_u32x7 (int32_t group_id, int32_t event_id, uint32_t value1, uint32_t value2, uint32_t value3, uint32_t value4, uint32_t value5, uint32_t value6, uint32_t value7)

Record an event for the given group. The given event ID must be valid for the specified group (i.e., between 0 and nb_event_types -1 included where nb_event_types is the value given to LLTRACE_declare_event_group()).

The record is done only if the trace system is started (see LLTRACE_start()).

Parameters

- group_id – ID of the group returned by LLTRACE_declare_event_group() .
- event_id – ID of the event.
- value1 – custom value for the event.
- value2 – custom value for the event.
- value3 – custom value for the event.
- value4 – custom value for the event.
- value5 – custom value for the event.
- value6 – custom value for the event.
- value7 – custom value for the event.

void LLTRACE_record_event_u32x8 (int32_t group_id, int32_t event_id, uint32_t value1, uint32_t value2, uint32_t value3, uint32_t value4, uint32_t value5, uint32_t value6, uint32_t value7, uint32_t value8)

Record an event for the given group. The given event ID must be valid for the specified group (i.e., between 0 and nb_event_types -1 included where nb_event_types is the value given to LLTRACE_declare_event_group()).

The record is done only if the trace system is started (see LLTRACE_start()).

Parameters

- group_id – ID of the group returned by LLTRACE_declare_event_group().
- event_id – ID of the event.
- value1 – custom value for the event.
- value2 – custom value for the event.
- value3 – custom value for the event.
- value4 – custom value for the event.
- value5 – custom value for the event.
- value6 – custom value for the event.
- value7 – custom value for the event.
- value8 – custom value for the event.

void LLTRACE_record_event_u32x9 (int32_t group_id, int32_t event_id, uint32_t value1, uint32_t value2, uint32_t value3, uint32_t value4, uint32_t value5, uint32_t value6, uint32_t value7, uint32_t value8, uint32_t value9)

Record an event for the given group. The given event ID must be valid for the specified group (i.e., between 0 and nb_event_types -1 included where nb_event_types is the value given to LLTRACE_declare_event_group()).

The record is done only if the trace system is started (see LLTRACE_start()).

Parameters

- group_id – ID of the group returned by LLTRACE_declare_event_group().
- event_id – ID of the event.
- value1 – custom value for the event.
- value2 – custom value for the event.
- value3 – custom value for the event.
- value4 – custom value for the event.
- value5 – custom value for the event.
- value6 – custom value for the event.
- value7 – custom value for the event.
- value8 – custom value for the event.
- value9 – custom value for the event.

void LLTRACE_record_event_u32x10 (int32_t group_id, int32_t event_id, uint32_t value1, uint32_t value2, uint32_t value3, uint32_t value4, uint32_t value5, uint32_t value6, uint32_t value7, uint32_t value8, uint32_t value9, uint32_t value10)

Record an event for the given group. The given event ID must be valid for the specified group (i.e., between 0 and nb_event_types -1 included where nb_event_types is the value given to LLTRACE_declare_event_group()).

The record is done only if the trace system is started (see LLTRACE_start()).

Parameters

- group_id – ID of the group returned by LLTRACE_declare_event_group().
- event_id – ID of the event.

- value1 – custom value for the event.
- value2 – custom value for the event.
- value3 – custom value for the event.
- value4 – custom value for the event.
- value5 – custom value for the event.
- value6 – custom value for the event.
- value7 – custom value for the event.
- value8 – custom value for the event.
- value9 – custom value for the event.
- value10 – custom value for the event.

void LLTRACE_record_event_end (int32_t group_id, int32_t event_id)

Record the end of the execution of an event for the given group. Call this function to trace the duration of an event previously record with one of the LLTRACE_record_event_*()

void LLTRACE_record_event_end_u32 (int32_t group_id, int32_t event_id, uint32_t value1)

Record the end of the execution of an event for the given group. Call this function to trace the duration of an event previously record with one of the LLTRACE_record_event_()

LLMJVM_MONITOR: Core Engine Monitoring

Principle

The Core Engine Monitoring provides Low Level APIs to track the Core Engine behavior at run-time through events and sending task information. The file LLMJVM_MONITOR_impl.h , which comes with the Core Engine, defines the API headers to be implemented.

LLMJVM_MONITOR_impl.h

Set of functions that the BSP must implement to enable trace recording from Core Engine events.

Functions

void LLMJVM_MONITOR_IMPL_initialize (bool auto_start)

This function is called once during Core Engine startup. It may be used to initialize specific data.

When auto_start is true, the monitoring traces must be generated now (i.e., call the LLTRACE_start() function). If auto_start is false, the monitoring traces must not be generated until the function LLTRACE_start() is called. Even if auto_start is false, the Core Engine will call the LLMJVM_MONITOR_IMPL_* functions.

Parameters

- auto_start – true to start monitoring now, false otherwise.

void LLMJVM_MONITOR_IMPL_on_shutdown (void)

This function is called during Core Engine end. It may be used to freed specific data.

void LLMJVM_MONITOR_IMPL_on_thread_create (int32_t thread_id)

This function is called by the Core Engine when a new Java thread is started.

Implementation can use the function MJVM_MONITOR_get_thread_info() to retrieve more information about the created thread.

Parameters

- thread_id – ID of the created thread.

void LLMJVM_MONITOR_IMPL_on_thread_modified (int32_t thread_id)

This function is called by the Core Engine when the priority or the name of a Java thread is modified.

Implementation can use the function MJVM_MONITOR_get_thread_info() to retrieve more information about the modified thread.

Parameters

- thread_id – ID of the modified thread.

void LLMJVM_MONITOR_IMPL_on_thread_state_changed (int32_t thread_id, *MJVM_MONITOR_state_t* new_state)

This function is called by the Core Engine when the state of a Java thread is modified (e.g., when a thread is waiting, running, etc.).

Parameters

- thread_id – ID of the modified thread.
- new_state – New state of the thread.

void LLMJVM_MONITOR_IMPL_on_idle (void)

This function is called by the Core Engine when it enter in idle mode (i.e., all the Java threads are sleeping).

void LLMJVM_MONITOR_IMPL_on_gc_start (int32_t current_thread_id)

This function is called by the Core Engine when a garbage collector operation is started.

Parameters

- current_thread_id – ID of the thread that raises the GC.

void LLMJVM_MONITOR_IMPL_on_gc_stop (int32_t current_thread_id)

This function is called by the Core Engine when a garbage collector operation is finished.

Parameters

- current_thread_id – ID of the thread that raises the GC.

void LLMJVM_MONITOR_IMPL_on_invoke_method (void *method)

This function is called by the Core Engine when an method is invoked.

Parameters

- method – the invoked method.

void LLMJVM_MONITOR_IMPL_on_return_method (void *method)

This function is called by the Core Engine when a method returns.

Parameters

- method – the invoked method.

void LLMJVM_MONITOR_IMPL_on_allocate (void *type, int32_t size, void *method, void *instruction_address, int32_t total_memory, int32_t free_memory, bool immortal)

This function is called by the Core Engine when a new Java object is allocated.

Parameters

- type – Class of the Java object.
- size – Size in bytes of the Java object.
- method – The Java method that allocates the object.
- instruction_address – Address of the Java instruction that allocates the object.
- total_memory – Total size of the Java heap in bytes.
- free_memory – Amount of the free memory in the Java heap in bytes.
- immortal – true is the allocated object is immortal, false otherwise.

void LLMJVM_MONITOR_IMPL_on_exception (void *exception_type, void *throw_method, void *throw_instruction_address, void *catch_method, void *catch_instruction_address, const char *message)

This function is called by the Core Engine when a Java exception is thrown.

Parameters

- exception_type – Class of the Java exception.
- throw_method – The Java method that throws the exception.
- throw_instruction_address – Address of the Java instruction that throws the exception.
- catch_method – The Java method that catches the exception.
- catch_instruction_address – Address of the Java instruction that catches the exception.
- message – Message of the exception.

void LLMJVM_MONITOR_IMPL_on_quota_reset (int32_t context_id, int32_t quota)

This function is called by the Core Engine when the quota for an execution context is updated.

Parameters

- context_id – ID of the execution context being updated.
- quota – The new quota assigned to the execution context.

void LLMJVM_MONITOR_IMPL_on_quota_reached (int32_t context_id)

This function is called by the Core Engine wwhen the quota for an execution context has been reached.

Parameters

- context_id – ID of the execution context which quota is reached.

void LLMJVM_MONITOR_IMPL_on_quantum_counters_reset ( )

This function is called by the Core Engine when all the quantum counters are reset to zero.

void LLMJVM_MONITOR_IMPL_on_thread_added_to_context (int32_t thread_id, int32_t context_id)

This function is called by the Core Engine when a thread is added to an execution context.

Parameters

- thread_id – ID of the added thread.
- context_id – ID of the execution context.

MJVM_MONITOR.h

Set of functions provided by Core Engine Monitoring that can be called by the BSP.

Functions

int32_t MJVM_MONITOR_get_threads_ids (int32_t *threads_ids, int32_t threads_ids_length)

Fills in the int32_t array threads_ids with the ID of the alive Java threads.

If the given array is not large enough to contains all the Java threads IDs, the threads_ids_length first Java threads IDs are stored.

This function must be called within the Core Engine task.

Parameters

- threads_ids – pointer to an int32_t array of threads_ids_length elements.
- threads_ids_length – length of threads_ids array.

Returns

the number of alive Java threads or MJVM_MONITOR_ERROR if this function is not called within the Core Engine task.

int32_t MJVM_MONITOR_get_thread_info (int32_t thread_id, *MJVM_MONITOR_thread_info_t* *thread_info, char *name, int32_t name_length)

Fills in the given thread_info structure and name array with the info of the thread and the name of the thread.

Given name array may null. In this case, the name of the thread is not filled in. If given name array is not large enough to copy the name of the thread, then the name is truncated to name_length -1 characters (last character is '\0').

This function must be called within the Core Engine task.

Parameters

- thread_id – ID of the thread.
- thread_info – pointer to the structure to fill in.
- name – pointer to a char array to fill in with the name of the thread. May be null.
- name_length – length in bytes of the given name array.

Returns

MJVM_MONITOR_OK on success, MJVM_MONITOR_INVALID_THREAD_ID if the given thread ID is invalid, MJVM_MONITOR_INVALID_THREAD_INFO if thread_info is null

void MJVM_MONITOR_dump_vm (void)

Prints the state of the Core Engine to the standard output stream. For each Java thread, the Java stack trace, the name, the state and the priority are printed.

MJVM_MONITOR_types.h

Types and macros used in the LLMJVM_MONITOR MJVM_MONITOR APIs.

Defines

MJVM_MONITOR_OK (0)

Returned value when function call has succeeded.

MJVM_MONITOR_ERROR (-1)

Returned value when function call has failed.

MJVM_MONITOR_INVALID_THREAD_INFO (-2)

Returned value when an invalid MJVM_MONITOR_ThreadInfo struct is given to a function.

MJVM_MONITOR_INVALID_THREAD_ID (-3)

Returned value when an invalid Java thread ID is given to a function.

Enums

enum MJVM_MONITOR_state_t

Specifies the states of the Java threads.

Values:

enumerator MJVM_MONITOR_STATE_READY = 1

In the ready state, all functional prerequisites for a transition into the running state exist, and the thread only waits for allocation of the processor. The scheduler decides which ready thread is executed next.

enumerator MJVM_MONITOR_STATE_RUNNING = 2

In the running state, the CPU is assigned to the thread, so that its instructions can be executed. Only one thread can be in this state at any point in time, while all the other states can be adopted simultaneously by several threads.

enumerator MJVM_MONITOR_STATE_WAITING = 3

Thread state for a waiting thread. A thread is in the waiting state due to calling one of the following methods:

- Thread.sleep
- Object.wait
- Thread.join
- SNI_suspendCurrentJavaThread

enumerator MJVM_MONITOR_STATE_MONITOR_QUEUED = 4

In the monitor queued state, a thread cannot continue execution because a monitor it tries to acquire is already owned by another thread. When this monitor is released, it will try to gain the ownership of the monitor. If it succeeds, it will be transferred into the ready state, otherwise it will remain in the monitor queued state.

enumerator MJVM_MONITOR_STATE_TERMINATED = 6

In the terminated state the thread is passive and has completed its execution.

struct MJVM_MONITOR_thread_info_t

#include <MJVM_MONITOR_types.h>

Public Members

int32_t id

ID of the thread.

int32_t priority

Java priority of the thread.

MJVM_MONITOR_state_t state

Current state of the thread.

int32_t max_stack_usage

Maximum Java stack usage in bytes. -1 if not available.

MicroEJ Foundation Libraries

EDC

Error Messages

When an exception is thrown by the runtime, the error message

Generic:E=<messageId>

is issued, where <messageId> meaning is defined in the next table:

Table 88: Generic Error Messages

Message ID	Description
1	Negative offset.
2	Negative length.
3	Offset + length > object length.

When an exception is thrown by the implementation of the EDC API, the error message

EDC-1.2:E=<messageId>

is issued, where <messageId> meaning is defined in the following table:

Table 89: EDC Error Messages

Message ID	Description
-4	No native stack found to execute the Java native method.
-3	Maximum stack size for a thread has been reached. See *Threads and Stacks Allocation*.
-2	No stack block could be allocated with the given size. See *Threads and Stacks Allocation*.
-1	The stack space is full. See *Threads and Stacks Allocation*.
1	A closed stream is being written/read.
2	The operation Reader.mark() is not supported.
3	lock is null in Reader(Object lock).
4	String index is out of range.
5	Argument must be a positive number.
6	Invalid radix used. Must be from Character.MIN_RADIX to Character.MAX_RADIX.
7	Operation Reader.reset() is not supported.
8	String is empty.
9	Start index is out of range.
10	End index is out of range.
11	A throwable cannot suppress itself in Throwable.addSuppressed(Throwable exception).
12	Given exception is null in Throwable.addSuppressed(Throwable exception).

Exit Codes

The MicroEJ Application can stop its execution by calling the method System.exit(). To retrieve the appplication exit code (or exit status), use the C function SNI_getExitCode() after the end of SNI_startVM() (see sni.h header file). If the MicroEJ Application ended without calling System.exit() then SNI_getExitCode() returns 0.

The error codes returned by SNI_startVM() are defined in the section *Error Codes*.

SNI

Error Messages

The following error messages are issued at runtime.

Table 90: [SNI] Run Time Error Messages.

Message ID	Description
-1	Not enough blocks.
-2	Reserved.
-3	Max stack blocks per thread reached.

KF

Definitions

Feature Definition Files

A Feature is a group of types, resources and *[BON]* immutables objects defined using two files that shall be in Application classpath:

- [featureName].kf , a Java properties file. Keys are described in *the "Feature definition file properties" table below*.
- [featureName].cert , an X509 certificate file that uniquely identifies the Feature

Table 91: Feature definition file properties

Key	Usage	Description
entryPoint	Mandatory	The fully qualified name of the class that implements ej.kf.FeatureEntryPoint
immutables	Optional	Semicolon separated list of paths to *[BON]* immutable files owned by the Feature. [BON] immutable file is defined by a / separated path relative to application classpath
resources	Optional	Semicolon separated list of resource names owned by the Feature. Resource name is defined by Class.getResourceAsStream(String)
requiredTypes	Optional	Comma separated list of fully qualified names of required types. (Types that may be dynamically loaded using Class.forName()).
types	Optional	Comma separated list of fully qualified names of types owned by the Feature. A wildcard is allowed as terminal character to embed all types starting with the given qualified name (a.b.C,x.y.*)
version	Mandatory	String version, that can retrieved using ej.kf.Module.getVersion()

Kernel Definition Files

Kernel definition files are mandatory if one or more Feature definition file is loaded and are named kernel.kf and kernel.cert . kernel.kf must only define the version key. All types, resources and immutables are automatically owned by the Kernel if not explicitly set to be owned by a Feature.

Kernel API Files

Kernel API file definition is explained here: *Kernel API*.

Access Error Codes

When an instruction is executed that will break a *Kernel & Features Specification (KF)* insulation semantic rule, a java.lang.IllegalAccessError is thrown, with an error code composed of two parts: [source][errorKind] .

- source : a single character indicating the kind of Java element on which the access error occurred (*Table "Error codes: source"*)
- errorKind : an error number indicating the action on which the access error occurred (*Table "Error codes: kind"*)

Table 92: Error codes: source

Character	Description
A	Error thrown when accessing an array
I	Error thrown when calling a method
F	Error thrown when accessing an instance field
M	Error thrown when entering a synchronized block or method
P	Error thrown when passing a parameter to a method call
R	Error thrown when returning from a method call
S	Error thrown when accessing a static field

Table 93: Error codes: kind

Id	Description
1	An object owned by a Feature is being assigned to an object owned by the Kernel, but the current context is not owned by the Kernel
2	An object owned by a Feature is being assigned to an object owned by another Feature
3	An object owned by a Feature is being accessed from a context owned by another Feature
4	A synchronize on an object owned by the Kernel is executed in a method owned by a Feature
5	A call to a feature code occurs while owning a Kernel monitor

Feature Installation and Loading Error Codes

When an error occurs during the installation of a Feature via the Kernel.install(InputStream) method, or during the loading of installed Features at bootstrap, the system may throw an ej.kf.InvalidFormatException.

This exception carries a specific error code, which can be retrieved by calling the getMessage() method on the InvalidFormatException instance. The returned message will follow the format: code=<id> , where id corresponds the specific error encountered.

- Error code 3 indicates that an unexpected exception occurs during Feature installation (e.g. code=3:<exception_type>).

  When the exception type is java.lang.OutOfMemoryError , it may indicate there is not enough space in the Managed Heap to load the *Feature's metadata section*. If sufficient memory is available or if a different exception type is reported (e.g. code=3:java.lang.ArrayIndexOutOfBoundsException), this likely points to a *corrupted Feature file*.

- Error codes ranging from 50 to 100 , thrown during Feature installation, indicate an error returned by the LLKERNEL Abstraction Layer.

Table 94: Feature Installation Error Codes Related to LLKERNEL Implementation

Id	Description
51	The LLKERNEL_impl_copyToROM() function returned an error.
52	The *maximum number of installed Features* is reached.
54	The address returned by the last call to LLKERNEL_impl_getFeatureAddressRAM() or LLKERNEL_impl_getFeatureAddressROM() does not match the alignment constraint.
55	The address returned by the last call to LLKERNEL_impl_getFeatureAddressROM() overlaps with a previously returned address.
56	The address returned by the last call to LLKERNEL_impl_getFeatureAddressRAM() overlaps with a previously returned address.
58	The .fo file cannot be installed on this Kernel because it contains a code chunk section with a size greater than this Kernel code chunk size.
59	The LLKERNEL_IMPL_allocateFeature() function returned 0 . The Feature cannot be allocated in memory.

Those thrown at Kernel boot indicate an issue with an already installed Feature, such as memory corruption or Feature link incompatibility after Kernel update.

Table 95: Installed Feature Loading Error Codes

Id	Description
53	The installed Feature content has been corrupted (CRC check error).
57	The address returned by the last call to LLKERNEL_impl_getFeatureAddressRAM() is not the same than the one returned when the Feature has been installed.
60	The LLKERNEL_IMPL_getFeatureHandle() function returned 0 . The handle for an installed Feature could not be retrieved as expected.

- Error codes ranging from 150 to 180 , thrown during Feature installation, typically indicate an invalid or a *corrupted Feature file*.

Corrupted Feature File

A corrupted Feature is most likely the result of an issue during the streaming transfer of the .fo file. This is typically caused by an incorrect implementation of the java.io.InputStream instance passed to the Kernel.install(InputStream) method.

If your InputStream is provided by a Foundation Library, ensure that your VEE Port has been successfully *qualified*. For example, when using a java.io.FileInputStream, verify that the *File System* port has been fully validated. The Feature loader relies on InputStream.skip(long), which is not commonly used. Be sure this method is correctly implemented, particularly with regard to handling the 64-bit long argument.

ECOM

Warning: This chapter describes the Foundation Library ECOM-1.1 .

ECOM-1.1 is discontinued since *Architecture 8.0.0*.

Error Messages

When an exception is thrown by the implementation of the ECOM API, the error message

ECOM-1.1:E=<messageId>

is issued, where <messageId> meaning is defined in the next table:

Table 96: ECOM Error Messages

Message ID	Description
1	The connection has been closed. No more action can be done on this connection.
2	The connection has already been closed.
3	The connection description is invalid. The connection cannot be opened.
4	The connection stream has already been opened. Only one stream per kind of stream (input or output stream) can be opened at the same time.
5	Too many connections have been opened at the same time. The platform is not able to open a new one. Try to close useless connections before trying to open the new connection.

ECOM Comm

Error Messages

When an exception is thrown by the implementation of the ECOM-COMM API, the error message

ECOM-COMM:E=<messageId>

is issued, where <messageId> meaning is defined in the next table:

Table 97: ECOM-COMM error messages

Message ID	Description
1	The connection descriptor must start with "comm:"
2	Reserved.
3	The Comm port is unknown.
4	The connection descriptor is invalid.
5	The Comm port is already open.
6	The baudrate is unsupported.
7	The number of bits per character is unsupported.
8	The number of stop bits is unsupported.
9	The parity is unsupported.
10	The input stream cannot be opened because native driver is not able to create a RX buffer to store the incoming data.
11	The output stream cannot be opened because native driver is not able to create a TX buffer to store the outgoing data.
12	The given connection descriptor option cannot be parsed.

Event Queue

Error Messages

When an exception is thrown by the implementation of the Event Queue API, the error message

EVENT-<APIversion>:E=<messageId>

is issued, where <messageId> meaning is defined in the next table:

Table 98: Event Queue Error Messages

Message ID	Description
-1	Invalid event type.
-2	FIFO buffer is full.
-3	Data or data buffer value exceeds 24 bits.
-4	Maximum number of types has already been registered.
-5	No listener is registered for the provided type.
-6	Default listener cannot be null.
-7	Not enough bytes remaining in the extended event.
-8	Less than 2 bytes remaining in the extended data.
-9	Less than 4 bytes remaining in the extended data.
-10	Less than 8 bytes remaining in the extended data.
-11	The buffer is too small to store the extended event data.
-12	No byte remaining in the data of the extended event.

MicroUI

Error Messages

See *Error Messages*.

FS

Error Messages

When an exception is thrown by the implementation of the FS API, the error message

FS:E=<messageId>

is issued, where <messageId> meaning is defined in the next table:

Table 99: File System Error Messages

Message ID	Description
-1	End of File (EOF).
-2	An error occurred during a File System operation.
-3	File System not initialized.

GNSS

Error Messages

When an exception is thrown by the implementation of the GNSS API, the error message

GNSS-<APIversion>:E=<messageId>

is issued, where <messageId> meaning is defined in the next table:

Table 100: GNSS Error Messages

Message ID	Description
-1	Unknown GNSS native error.
-2	An error occurred during data read.
-3	Read data is invalid.
-4	An operation needing the GNSS engine to be started was attemtped with the engine off.

Net

Error Messages

When an exception is thrown by the implementation of the Net API, the error message

NET-1.1:E=<messageId>

is issued, where <messageId> meaning is defined in the next table:

Table 101: Net Error Messages

Message ID	Description
-2	Permission denied.
-3	Bad socket file descriptor.
-4	Host is down.
-5	Network is down.
-6	Network is unreachable.
-7	Address already in use.
-8	Connection abort.
-9	Invalid argument.
-10	Socket option not available.
-11	Socket not connected.
-12	Unsupported network address family.
-13	Connection refused.
-14	Socket already connected.
-15	Connection reset by peer.
-16	Message size to be sent is too long.
-17	Broken pipe.
-18	Connection timed out.
-19	Not enough free memory.
-20	No route to host.
-21	Unknown host.
-23	Native method not implemented.
-24	The blocking request queue is full, and a new request cannot be added now.

continues on next page

Table 101 – continued from previous page

Message ID	Description
-25	Network not initialized.
-26	Socket not bound.
-27	Native file descriptor already initialized.
-28	Socket closed.
-29	Wrong native MAC address size returned.
-30	Invalid port.
-31	Not a datagram socket.
-32	Socket already created.
-255	Unknown error.

SSL

Error Messages

When an exception is thrown by the implementation of the SSL API, the error message

SSL-2.0:E=<messageId>

is issued, where <messageId> meaning is defined in the next table:

Table 102: SSL Error Messages

Message ID	Description
-2	Connection reset by the peer.
-3	Connection timed out.
-5	Dispatch blocking request queue is full, and a new request cannot be added now.
-6	Certificate parsing error.
-7	The certificate data size bigger than the immortal buffer used to process certificate.
-8	No trusted certificate found.
-9	Basic constraints check failed: Intermediate certificate is not a CA certificate.
-10	Subject/issuer name chaining error.
-21	Wrong block type for RSA function.
-22	RSA buffer error: Output is too small, or input is too large.
-23	Output buffer is too small, or input is too large.
-24	Certificate AlogID setting error.
-25	Certificate public-key setting error.
-26	Certificate date validity setting error.
-27	Certificate subject name setting error.
-28	Certificate issuer name setting error.
-29	CA basic constraint setting error.
-30	Extensions setting error.
-31	Invalid ASN version number.
-32	ASN get int error: invalid data.
-33	ASN key init error: invalid input.
-34	Invalid ASN object id.
-35	Not null ASN tag.
-36	ASN parsing error: zero expected.
-37	ASN bit string error: wrong id.
-38	ASN OID error: unknown sum id.
-39	ASN date error: bad size.

continues on next page

Table 102 – continued from previous page

Message ID	Description
-40	ASN date error: current date before.
-41	ASN date error: current date after.
-42	ASN signature error: mismatched OID.
-43	ASN time error: unknown time type.
-44	ASN input error: not enough data.
-45	ASN signature error: confirm failure.
-46	ASN signature error: unsupported hash type.
-47	ASN signature error: unsupported key type.
-48	ASN key init error: invalid input.
-49	ASN NTRU key decode error: invalid input.
-50	X.509 critical extension ignored.
-51	ASN no signer to confirm failure (no CA found).
-52	ASN CRL signature-confirm failure.
-53	ASN CRL: no signer to confirm failure.
-54	ASN OCSP signature-confirm failure.
-60	ECC input argument is wrong type.
-61	ECC ASN1 bad key data: invalid input.
-62	ECC curve sum OID unsupported: invalid input.
-63	Bad function argument provided.
-64	Feature not compiled in.
-65	Unicode password too big.
-66	No password provided by user.
-67	AltNames extensions too big.
-70	AES-GCM Authentication check fail.
-71	AES-CCM Authentication check fail.
-80	Cavium Init type error.
-81	Bad alignment error, no alloc help.
-82	Bad ECC encrypt state operation.
-83	Bad padding: message wrong length.
-84	Certificate request attributes setting error.
-85	PKCS#7 error: mismatched OID value.
-86	PKCS#7 error: no matching recipient found.
-87	FIPS mode not allowed error.
-88	Name constraint error.
-89	Random Number Generator failed.
-90	FIPS Mode HMAC minimum key length error.
-91	RSA Padding error.
-92	Export public ECC key in ANSI format error: Output length only set.
-93	In Core Integrity check FIPS error.
-94	AES Known Answer Test check FIPS error.
-95	DES3 Known Answer Test check FIPS error.
-96	HMAC Known Answer Test check FIPS error.
-97	RSA Known Answer Test check FIPS error.
-98	DRBG Known Answer Test check FIPS error.
-99	DRBG Continuous Test FIPS error.
-100	AESGCM Known Answer Test check FIPS error.
-101	Process input state error.
-102	Bad index to key rounds.
-103	Out of memory.

continues on next page

Table 102 – continued from previous page

Message ID	Description
-104	Verify problem found on completion.
-105	Verify mac problem.
-106	Parse error on header.
-107	Weird handshake type.
-108	Error state on socket.
-109	Expected data, not there.
-110	Not enough data to complete task.
-111	Unknown type in record header.
-112	Error during decryption.
-113	Received alert: fatal error.
-114	Error during encryption.
-116	Need peer's key.
-117	Need the private key.
-118	Error during RSA private operation.
-119	Server missing DH parameters.
-120	Build message failure.
-121	Client hello not formed correctly.
-122	The peer subject name mismatch.
-123	Non-blocking socket wants data to be read.
-124	Handshake layer not ready yet; complete first.
-125	Premaster secret version mismatch error.
-126	Record layer version error.
-127	Non-blocking socket write buffer full.
-128	Malformed buffer input error.
-129	Verify problem on certificate and check date/time on your device.
-130	Verify problem based on signature.
-131	PSK client identity error.
-132	PSK server hint error.
-133	PSK key callback error.
-134	Record layer length error.
-135	Can't decode peer key.
-136	The peer sent close notify alert.
-137	Wrong client/server type.
-138	The peer didn't send the certificate.
-140	NTRU key error.
-141	NTRU DRBG error.
-142	NTRU encrypt error.
-143	NTRU decrypt error.
-150	Bad ECC Curve Type or unsupported.
-151	Bad ECC Curve or unsupported.
-152	Bad ECC Peer Key.
-153	ECC Make Key failure.
-154	ECC Export Key failure.
-155	ECC DHE shared failure.
-157	Not a CA by basic constraint.
-159	Bad Certificate Manager error.
-160	OCSP Certificate revoked.
-161	CRL Certificate revoked.
-162	CRL missing, not loaded.

continues on next page

Table 102 – continued from previous page

Message ID	Description
-165	OCSP needs a URL for lookup.
-166	OCSP Certificate unknown.
-167	OCSP responder lookup fail.
-168	Maximum chain depth exceeded.
-171	Suites pointer error.
-172	No PEM header found.
-173	Out of order message: fatal.
-174	Bad KEY type found.
-175	Sanity check on ciphertext failed.
-176	Receive callback returned more than requested.
-178	Need peer certificate for verification.
-181	Unrecognized host name error.
-182	Unrecognized max fragment length.
-183	Key Use digitalSignature not set.
-185	Key Use keyEncipherment not set.
-186	Ext Key Use server/client authentication not set.
-187	Send callback out-of-bounds read error.
-188	Invalid renegotiation.
-189	Peer sent different certificate during SCR.
-190	Finished message received from peer before receiving the Change Cipher message.
-191	Sanity check on message order.
-192	Duplicate handshake message.
-193	Unsupported cipher suite.
-194	Can’t match cipher suite.
-195	Bad certificate type.
-196	Bad file type.
-197	Opening random device error.
-198	Reading random device error.
-199	Windows cryptographic init error.
-200	Windows cryptographic generation error.
-201	No data is waiting to be received from the random device.
-202	Unknown error.

Tools Options and Error Codes

Immutable Files Related Error Messages

The following error messages are issued at SOAR time (link phase) and not at runtime.

Table 103: Errors when parsing immutable files at link time.

Message ID	Description
0	Duplicated ID in immutable files. Each immutable object should have a unique ID in the SOAR image.
1	An immutable file refers to an unknown field of an object.
2	Tried to assign the same object field twice.
3	All immutable object fields should be defined in the immutable file description.
4	The assigned value does not match the expected Java type.
5	An immutable object refers to an unknown ID.
6	The length of the immutable object does not match the length of the assigned object.
7	The type defined in the file doesn't match the Java expected type.
8	Generic error while parsing an immutable file.
9	Cycle detected in an alias definition.
10	An immutable object is an instance of an abstract class or an interface.
11	Unknown XML attribute in an immutable file.
12	A mandatory XML attribute is missing.
13	The value is not a valid Java literal.
14	Alias already exists.

SNI

The following error messages are issued at SOAR time and not at runtime.

Table 104: [SNI] Link Time Error Messages.

Message ID	Description
363	Argument cannot be a reference.
364	Argument can only be from a base type array.
365	Return type must be a base type.
366	Method must be a static method.

SP Compiler

Options

Table 105: Shielded Plug Compiler Options.

Option name	Description
-verbose[e...e]	Extra messages are printed out to the console according to the number of 'e'.
-descriptionFile file	XML Shielded Plug description file. Multiple files allowed.
-waitingTaskLimit value	Maximum number of task/threads that can wait on a block: a number between 0 and 7. -1 is for no limit; 8 is for unspecified.
-immutable	When specified, only immutable Shielded Plugs can be compiled.
-output dir	Output directory. Default is the current directory.
-outputName name	Output name for the Shielded Plug layout description. Default is "shielded_plug".
-endianness name	Either "little" or "big". Default is "little".
-outputArchitecture value	Output ELF architecture. Only "ELF" architecture is available.
-rwBlockHeaderSize value	Read/Write header file value.
-genIdsC	When specified, generate a C header file with block ID constants.
-cOutputDir dir	Output directory of C header files. Default is the current directory.
-cConstantsPrefix prefix	C constants name prefix for block IDs.
-genIdsJava	When specified, generate Java interfaces file with block ID constants.
-jOutputDir dir	Output directory of Java interfaces files. Default is the current directory.
-jPackage name	The name of the package for Java interfaces.

Error Messages

Table 106: Shielded Plug Compiler Error Messages.

Message ID	Description
0	Internal limits reached.
1	Invalid endianness.
2	Invalid output architecture.
3	Error while reading / writing files.
4	Missing a mandatory option.

NLS Immutables Creator

Table 107: NLS Immutables Creator Errors Messages

ID	Type	Description
1	Error	Error reading the nls list file: invalid path, input/output error, etc.
2	Error	Error reading the nls list file: The file contents are invalid.
3	Error	Specified class is not an interface.
4	Error	Invalid message ID. Must be greater than or equal to 1.
5	Error	Duplicate ID. Both messages use the same message ID.
6	Error	Specified interface does not exist.
7	Error	Specified message constant is not visible (must be public).
8	Error	Specified message constant is not an integer.
9	Error	No locale file is defined for the specified header.
10	Error	IO error: Cannot create the output file.
11	Warning	Missing message value.
12	Warning	There is a gap (or gaps) in messages constants.
13	Warning	Specified property does not denote a message.
14	Warning	Invalid properties header file. File is ignored.
15	Warning	No message is defined for the specified header.
16	Warning	Invalid property.

MicroUI Static Initializer

Inputs

The XML file used as input by the MicroUI Static Initialization Tool may contain tags related to the Input component as described below.

Listing 40: Event Generators Description

```
<eventgenerators>
<!-- Generic Event Generators -->
   <eventgenerator name="GENERIC" class="foo.bar.Zork">
      <property name="PROP1" value="3"/>
      <property name="PROP2" value="aaa"/>
   </eventgenerator>

   <!-- Predefined Event Generators -->
   <command name="COMMANDS"/>
   <buttons name="BUTTONS" extended="3"/>
   <buttons name="JOYSTICK" extended="5"/>
   <pointer name="POINTER" width="1200" height="1200"/>
   <touch name="TOUCH" display="DISPLAY"/>
   <states name="STATES" numbers="NUMBERS" values="VALUES"/>

</eventgenerators>

<array name="NUMBERS">
   <elem value="3"/>
   <elem value="2"/>
   <elem value="5"/>
</array>
```

(continues on next page)

(continued from previous page)

```
<array name="VALUES">
   <elem value="2"/>
   <elem value="0"/>
   <elem value="1"/>
</array>
```

Table 108: Event Generators Static Definition

Tag	Attributes	Description
eventgenerators		The list of event generators.
	priority	*Optional.* An integer value. Defines the internal display thread priority. Default value is 5.
eventgenerator		Describes a generic event generator. See also *Generic Event Generators*.
	name	The logical name.
	class	The event generator class (must extend the ej.microui.event.generator.GenericEventGenerator class). This class must be available in the MicroEJ Application classpath.
property		A generic event generator property. The generic event generator will receive this property at startup, via the method setProperty.
	name	The property key.
	value	The property value.
command		The default event generator Command.
	name	The logical name.
buttons		The default event generator Buttons.
	name	The logical name.
	extended	*Optional.* An integer value. Defines the number of buttons which support the MicroUI extended features (elapsed time, click and double-click).
pointer		The default event generator Pointer.
	name	The logical name.
	width	An integer value. Defines the pointer area width.
	height	An integer value. Defines the pointer area heigth.
	extended	*Optional.* An integer value. Defines the number of pointer buttons (right click, left click, etc.) which support the MicroUI extended features (elapsed time, click and double-click).
touch		The default event generator Touch.
	name	The logical name.
	display	Logical name of the Display with which the touch is associated.
states		An event generator that manages a group of state machines. The state of a machine is changed by sending an event using LLUI_INPUT_sendStateEvent.
	name	The logical name.
	numbers	The logical name of the array which defines the number of state machines for this States generator, and their range of state values. The IDs of the state machines start at 0. The number of state machines managed by the States generator is equal to the size of the numbers array, and the value of each entry in the array is the number of different values supported for that state machine. State machine values for state machine i can be in the range 0 to numbers[i] -1.
	values	*Optional.* The logical name of the array which defines the initial state values of the state machines for this States generator. The values array must be the same size as the numbers array. If initial state values are specified using a values array, then the LLUI_INPUT_IMPL_getInitialStateValue function is not called; otherwise that function is used to establish the initial values[1]
array		An array of values.
	name	The logical name.
elem		A value.
		An integer value.

Display

The display component augments the static initialization file with:

- The configuration of each display.
- Fonts that are implicitly embedded within the application (also called system fonts). Applications can also embed their own fonts.

```
<display name="DISPLAY"/>

<fonts>
   <font file="resources\fonts\myfont.ejf">
      <range name="LATIN" sections="0-2"/>
      <customrange start="0x21" end="0x3f"/>
   </font>
   <font file="C:\data\myfont.ejf"/>
</fonts>
```

Table 109: Display Static Initialization XML Tags Definition

Tag	Attributes	Description
display		The display element describes one display.
	name	The logical name of the display.
	priority	*Deprecated*. This value is not taken in consideration. Use MicroEj application launcher option instead.
	default	*Deprecated*. This value is not taken in consideration.
fonts		The list of system fonts. The system fonts are available for all displays.
font		A system font.
	file	The font file path. The path may be absolute or relative to the XML file.
range		A font generic range.
	name	The generic range name (LATIN , HAN , etc.)
	sections	*Optional.* Defines one or several sub parts of the generic range. "1": add only part 1 of the range "1-5": add parts 1 to 5 "1,5": add parts 1 and 5 These combinations are allowed: "1,5,6-8" add parts 1, 5, and 6 through 8 By default, all range parts are embedded.
customrange		A font-specific range.
	start	UTF16 value of the very first character to embed.
	end	UTF16 value of the very last character to embed.

[1] Exception: When using MicroEJ Platform, where there is no equivalent to the LLUI_INPUT_IMPL_getInitialStateValue function. If no values array is provided, and the MicroEJ Platform is being used, all state machines take 0 as their initial state value.

Front Panel

FP File

XML Schema

```
<?xml version=”1.0”?>
<frontpanel
  xmlns:xsi=”http://www.w3.org/2001/XMLSchema-instance”
  xmlns=”https://developer.microej.com”
  xsi:schemaLocation=”https://developer.microej.com .widget.xsd”>

  <device name=”example” skin=”example-device.png”>
    <ej.fp.widget.[type] x=”22” y=”51” [widget-attributes]/>
    <ej.fp.widget.[type] x=”30” y=”125” [widget-attributes]/>
    <!-- ... -->
  </device>
</frontpanel>
```

File Specification

Table 110: FP File Specification

Tag	Attributes	Description
frontpanel		The root element.
	xmlns:xsi	Invariant tag[1]
	xmlns	Invariant tag[2]
	xsi:schemaLocation	Invariant tag[3]
device		The device’s root element.
	name	The device’s logical name.
	skin	Refers to a PNG file which defines the device background.
ej.fp.widget.xxx		Defines the widget to use. Refer to the widget documentation.
	label	All widget should provide this identifier. Sometimes it is used as string, sometimes as integer
	x	The widget x-coordinate.
	y	The widget y-coordinate.

[1] Must be http://www.w3.org/2001/XMLSchema-instance
[2] Must be https://developer.microej.com
[3] Must be https://developer.microej.com .widget.xsd

HIL Engine

Below are the HIL Engine options:

Table 111: HIL Engine Options

Option name	Description
-verbose[e....e]	Extra messages are printed out to the console (add extra e to get more messages).
-ip <address>	MicroEJ Simulator connection IP address (A.B.C.D). By default, set to localhost.
-port <port>	MicroEJ Simulator connection port. By default, set to 8001.
-connectTimeout <timeout>	timeout in s for MicroEJ Simulator connections. By default, set to 10 seconds.
-excludes <name[sep]name>	Types that will be excluded from the HIL Engine class resolution provided mocks. By default, no types are excluded.
-mocks <name[sep]name>	Mocks are either .jar file or .class files.

Heap Dumping

XML Schema

Below is the XML schema for heap dumps.

Table 112: XML Schema for Heap Dumps

```
<?xml version='1.0' encoding='UTF-8'?>
<!--
    Schema

    Copyright 2012 IS2T. All rights reserved.

   IS2T PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
-->

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <!-- root element: heap -->
    <xs:element name="heap">
        <xs:complexType>
            <xs:choice minOccurs="0" maxOccurs="unbounded">
                <xs:element ref="class"/>
                <xs:element ref="object"/>
                <xs:element ref="array"/>
                <xs:element ref="stringLiteral"/>
            </xs:choice>
        </xs:complexType>
    </xs:element>
```

continues on next page

Table 112 – continued from previous page

```
<!-- class element -->
<xs:element name="class">
   <xs:complexType>
      <xs:choice minOccurs="0" maxOccurs="unbounded">
         <xs:element ref="field"/>
      </xs:choice>
      <xs:attribute name="name" type="xs:string" use = "required"/>
      <xs:attribute name="id" type="xs:string" use = "required"/>
      <xs:attribute name="superclass" type="xs:string"/>
   </xs:complexType>
</xs:element>
```

```
<!-- object element-->
<xs:element name="object">
   <xs:complexType>
      <xs:choice minOccurs="0" maxOccurs="unbounded">
         <xs:element ref="field"/>
      </xs:choice>
      <xs:attribute name="id" type="xs:string" use = "required"/>
      <xs:attribute name="class" type="xs:string" use = "required"/>
      <xs:attribute name="createdAt" type="xs:string" use = "optional"/>
      <xs:attribute name="createdInThread" type="xs:string" use = "optional"/>
      <xs:attribute name="createdInMethod" type="xs:string"/>
      <xs:attribute name="tag" type="xs:string" use = "required"/>
   </xs:complexType>
</xs:element>
```

```
<!-- array element-->
<xs:element name="array" type = "arrayTypeWithAttribute"/>
<!-- stringLiteral element-->
<xs:element name="stringLiteral">
   <xs:complexType>
      <xs:sequence>
         <xs:element minOccurs ="4" maxOccurs="4" ref="field "/>
      </xs:sequence>
      <xs:attribute name="id" type="xs:string" use = "required"/>
      <xs:attribute name="class" type="xs:string" use = "required"/>
   </xs:complexType>
</xs:element>
```

continues on next page

Table 112 – continued from previous page

```
  <!-- field element:  child of class, object and stringLiteral-->
    <xs:element name="field">
      <xs:complexType>
        <xs:attribute name="name" type="xs:string" use = "required"/>
        <xs:attribute name="id" type="xs:string" use = "optional"/>
        <xs:attribute name="value" type="xs:string" use = "optional"/>
        <xs:attribute name="type" type="xs:string" use = "optional"/>
      </xs:complexType>
    </xs:element>

    <xs:simpleType name = "arrayType">
      <xs:list itemType="xs:integer"/>
    </xs:simpleType>



<!-- complex type "arrayTypeWithAttribute". type of array element-->
    <xs:complexType name = "arrayTypeWithAttribute">
      <xs:simpleContent>
        <xs:extension base="arrayType">
          <xs:attribute name="id" type="xs:string" use = "required"/>
          <xs:attribute name="class" type="xs:string" use = "required"/>
          <xs:attribute name="createdAt" type="xs:string" use = "optional"/>
          <xs:attribute name="createdInThread" type="xs:string" use = "optional"/>
          <xs:attribute name="createdInMethod" type="xs:string" use = "optional"/>
          <xs:attribute name="length" type="xs:string" use = "required"/>
          <xs:attribute name="elementsType" type="xs:string" use = "optional"/>
          <xs:attribute name="type" type="xs:string" use = "optional"/>
        </xs:extension>
      </xs:simpleContent>
    </xs:complexType>

  </xs:schema>
```

File Specification

Types referenced in heap dumps are represented in the internal classfile format (*Internal classfile Format for Types*). Fully qualified names are names separated by the / separator (For example, a/b/C).

Listing 41: Internal classfile Format for Types

```
Type = <BaseType> | <ClassType> | <ArrayType>
BaseType: B(byte), C(char), D(double), F(float), I(int), J(long), S(short), Z(boolean),
ClassType: L<ClassName>;
ArrayType: [<Type>
```

Tags used in the heap dumps are described in the table below.

Table 113: Tag Descriptions

Tags	Attributes	Description
heap		The root element.
class		Element that references a Java class.
	name	Class type (<ClassType>)
	id	Unique identifier of the class.
	superclass	Identifier of the superclass of this class.
object		Element that references a Java object.
	id	Unique identifier of this object.
	class	Fully qualified name of the class of this object.
array		Element that references a Java array.
	id	Unique identifier of this array.
	class	Fully qualified name of the class of this array.
	elementsType	Type of the elements of this array.
	length	Array length.
stringLiteral		Element that references a java.lang.String literal.
	id	Unique identifier of this object.
	class	Id of java.lang.String class.
field		Element that references the field of an object or a class.
	name	Name of this field.
	id	Object or Array identifier, if it holds a reference.
	type	Type of this field, if it holds a base type.
	value	Value of this field, if it holds a base type.

Architectures MCU / Compiler

Principle

The MicroEJ C libraries have been built for a specific processor (a specific MCU architecture) with a specific C compiler. The third-party linker must make sure to link C libraries compatible with the MicroEJ C libraries. This chapter details the compiler version, flags and options used to build MicroEJ C libraries for each processor.

Some processors include an optional floating point unit (FPU). This FPU is single precision (32

bits) and is compliant with IEEE 754 standard. It can be disabled when not in use, thus reducing power consumption. There are two steps to use the FPU in an application. The first step is to tell the compiler and the linker that the microcontroller has an FPU available so that they will produce compatible binary code. The second step is to enable the FPU during execution. This is done by writing to CPAR in the SystemInit() function. Even if there is an FPU in the processor, the linker may still need to use runtime library functions to deal with advanced operations. A program may also define calculation functions with floating numbers, either as parameters or return values. There are several Application Binary Interfaces (ABI) to handle floating point calculations. Hence, most compilers provide options to select one of these ABIs. This will affect how parameters are passed between caller functions and callee functions, and whether the FPU is used or not. There are three ABIs:

- Soft ABI without FPU hardware. Values are passed via integer registers.
- Soft ABI with FPU hardware. The FPU is accessed directly for simple operations, but when a function is called, the integer registers are used.
- Hard ABI. The FPU is accessed directly for simple operations, and FPU-specific registers are used when a function is called, for both parameters and the return value.

  It is important to note that code compiled with a particular ABI might not be compatible with code compiled with another ABI. MicroEJ modules, including the Core Engine, use the hard ABI.

Supported Core Engine Capabilities by Architecture Matrix

The following table lists the supported Core Engine capabilities by MicroEJ Architectures.

Table 114: Supported Core Engine Capabilities by MicroEJ Architecture Matrix

Core Engine Architectures		Capabilities		
MCU	Compiler	Mono- Sandbox	Tiny- Sandbox	Multi- Sandbox
ARM Cortex-M0	GCC	YES	YES	NO
ARM Cortex-M33	IAR Embedded Workbench for ARM	YES	YES	YES
ARM Cortex-M33	GCC	YES	NO	YES
ARM Cortex-M4	IAR Embedded Workbench for ARM	YES	YES	YES
ARM Cortex-M4	GCC	YES	NO	YES
ARM Cortex-M4	Keil uVision	YES	NO	YES
ARM Cortex-M7	IAR Embedded Workbench for ARM	YES	NO	YES
ARM Cortex-M7	GCC	YES	NO	YES
ARM Cortex-M7	Keil uVision	YES	NO	YES
ARMv7A	GCC	YES	YES	YES
ARMv7VE	GCC	YES	YES	YES
ESP32	ESP-IDF	YES	NO	YES

ARM Cortex-M0

Table 115: ARM Cortex-M0 Compilers

Compiler	Version	Flags and Options	Module
GCC	4.8	-mabi=aapcs -mcpu=cortex-m0 -mlittle-endian -mthumb	flopi0G22

ARM Cortex-M33

MicroEJ supports Cortex-M33 core with DSP extension using *Cortex-M4 architectures*.

ARM Cortex-M4

Table 116: ARM Cortex-M4 Compilers

Compiler	Build Version	Known Compatible Versions	Flags and Options	Module
Keil uVision	5.18.0.0	5.x	--cpu Cortex-M4.fp --apcs=/hardfp --fpmode=ieee_no_fenv	flopi4A20
GCC	4.8	4.x, 5.x, 6.x, 7.x, 8.x, 9.x	-mabi=aapcs -mcpu=cortex-m4 -mlittle-endian -mfpu=fpv4-sp-d16 -mfloat-abi=hard -mthumb	flopi4G25
IAR Embedded Workbench for ARM	8.32.1.18638	8.x, 9.x	--cpu Cortex-M4F --fpu VFPv4_sp	flopi4I35

Note:

- Cortex-M4 architectures are compiled using hardfp convention call.

ARM Cortex-M7

Table 117: ARM Cortex-M7 Compilers

Compiler	Build Version	Known Compatible Versions	Flags and Options	Module
Keil uVision	5.18.0.0	5.x	--cpu Cortex-M7.fp.sp --apcs=/hardfp --fpmode=ieee_no_fenv	flopi7A21
GCC	4.8	4.x, 5.x, 6.x, 7.x, 8.x, 9.x	-mabi=aapcs -mcpu=cortex-m7 -mlittle-endian -mfpu=fpv5-sp-d16 -mfloat-abi=hard -mthumbb	flopi7G26
IAR Embedded Workbench for ARM	8.32.1.18638	8.x, 9.x	--cpu Cortex-M7 --fpu VFPv5_sp	flopi7I36

ARMv7A (ARMv7-A without integer division extension: Cortex-A5/Cortex-A8/Cortex-A9)

Table 118: ARMv7A Compilers

Compiler	Build Version	Known Compatible Versions	Flags and Options	Module
GCC	10.3	4.x, 5.x, 6.x, 7.x, 8.x, 9.x, 10.x	-mabi=aapcs-linux -march=armv7-a -mlittle-endian -mfpu=vfp -mfloat-abi=hard -mthumb	oliveARMv7A_2

ARMv7VE (ARMv7-A with integer division extension: Cortex-A7/Cortex-A15)

Table 119: ARMv7VE Compilers

Compiler	Build Version	Known Compatible Versions	Flags and Options	Module
GCC	10.3	4.x, 5.x, 6.x, 7.x, 8.x, 9.x, 10.x	-mabi=aapcs-linux -march=armv7ve -mlittle-endian -mfpu=vfp -mfloat-abi=hard -mthumb	oliveARMv7VE_1

ESP32

Table 120: Espressif ESP32 Compilers

Compiler	Version	Flags and Options	Module Name	Module Version
GCC (ESP-IDF)	5.2.0 (crosstool-ng-1.22.0-80-g6c4433a)	-mlongcalls	simikou1	Any
GCC (ESP-IDF)	5.2.0 (crosstool-ng-1.22.0-80-g6c4433a)	-mlongcalls -mfix-esp32-psram-cache-issue	simikou2	Up to 7.13.0 (included)
GCC (ESP-IDF)	5.2.0 (crosstool-ng-1.22.0-96-g2852398)	-mlongcalls -mfix-esp32-psram-cache-issue	simikou2	7.12.2 or higher
GCC (ESP-IDF)	8.2.0 (crosstool-NG esp-2019r2)	-mlongcalls	simikou3	7.16.0 or higher
GCC (ESP-IDF)	5.2.0 (crosstool-ng-1.22.0-97-gc752ad5)	-mlongcalls -mfix-esp32-psram-cache-issue	simikou4	7.12.2 or higher
GCC (ESP-IDF)	8.4.0 (crosstool-NG esp-2021r1)	-mlongcalls	simikou5	7.16.1 or higher
GCC (ESP-IDF)	8.4.0 (crosstool-NG esp-2021r1)	-mlongcalls -mfix-esp32-psram-cache-issue -mfix-esp32-psram-cache-strategy=memw	simikou6	7.16.1 or higher
GCC (ESP-IDF)	11.2.0 (crosstool-NG esp-2022r1)	-mlongcalls	simikou7	7.20.1 or higher

IAR Linker Specific Options

This section lists options that must be passed to IAR linker for correctly linking the MicroEJ object file (microejapp.o) generated by the SOAR.

--no_range_reservations

MicroEJ SOAR generates ELF absolute symbols to define some *Link-Time Option* (0 based values). By default, IAR linker allocates a 1 byte section on the fly, which may cause silent sections placement side effects or a section overlap error when multiple symbols are generated with the same absolute value:

Error[Lp023]: absolute placement (in [0x00000000-0x000000db]) overlaps with absolute symbol [...]

The option --no_range_reservations tells IAR linker to manage an absolute symbol as described by the ELF specification.

--diag_suppress=Lp029

MicroEJ SOAR generates internal veneers that may be interpreted as illegal code by IAR linker, causing the following error:

Error[Lp029]: instruction validation failure in section "C:\xxx\microejapp.o[.text.___icetea___virtual____1xxx#1126]": nested IT blocks. Code in wrong mode?

The option --diag_suppress=Lp029 tells IAR linker to ignore instructions validation errors.

GNU LD Specific Options

--start-group --end-group

By default the GNU linker does not search unresolved symbols in previously loaded files and can cause undefined reference errors. To solve this issue, either change the load order of libraries (put microejapp.o first) or guard the libraries with the options --start-group and --end-group .

ARM Linker Specific Options

ARM linker (armlink) is the linker included in ARM Compiler and Keil MDK-ARM development tools.

Fix Unexpected Undefined Symbol

The ARM linker requires to resolve all symbols before detecting some that are not transitively required for linking the Executable. This typically happen when linking ELF object files containing dead code or debug functions that are compiled but not intended to be linked. If such functions refer to unresolved symbols, you may need to define a fake symbol to make the linker happy. You can declare it in your BSP project or directly in your VEE Port as following:

- Create a file MICROJVM/link/armlink-weak.lscf in the *dropins* directory of your VEE Port configuration project.
- Edit the file and declare as many symbols as required. See also the *MicroEJ Linker* chapter for more details on the MicroEJ linker file syntax.

```
<lscFragment>
  <defSymbol name="[symbolName]" value="0" rootSymbol="true" weak="true"/>
</lscFragment>
```

The weak symbol(s) will be directly defined in the application object file (microejapp.o).

Link the SOAR Debug Section

When building an Application, the *SOAR* generates a dedicated ELF debug section named `.debug.soar` in the application object file (`microejapp.o`). This section is used by debug tools such as the *Stack Trace Reader* or the *Heap Dumper*. It is also used by the SOAR itself for *building Features* on a Kernel.

Unfortunately, the ARM linker does not link this section in the output ELF executable, even with debug mode enabled. If you try to load the raw executable produced by the ARM linker, the tools will fail with a *no debug section* error. Here is an example with the *Stack Trace Reader*:

```
================ [ MicroEJ Core Engine Trace ] ================
[INFO] Paste the MicroEJ core engine stack trace here.
1 : PROXY ERROR
  [M8] - The file XXX is not␣
↪a valid image file or has no debug informations (can't read file: XXX (no debug section)).
```

To be able to use debug tools, the debug section must be manually linked and injected in the Executable. This is done using the *SOAR debug infos post-linker tool*.

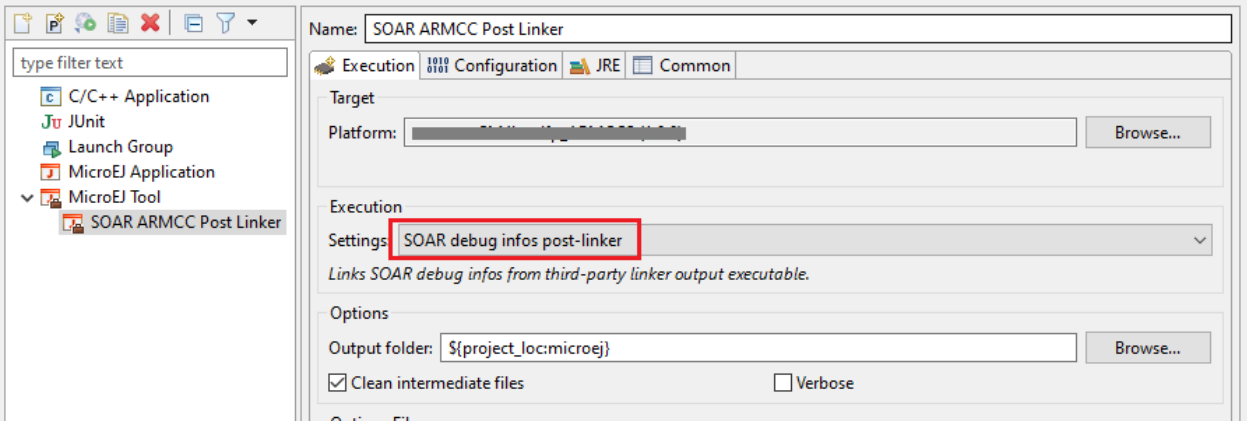


Fig. 272: SOAR debug infos post-linker tool Selection

This tool takes two file options:

- `soar.object.file`: the internal object file produced by the SOAR when building the Application. It can be found in the *Launch Output Folder* at `soar/[application_main_class].o`.
- `output.executable.file`: the Executable file produced by the ARM linker that includes the linked Application.

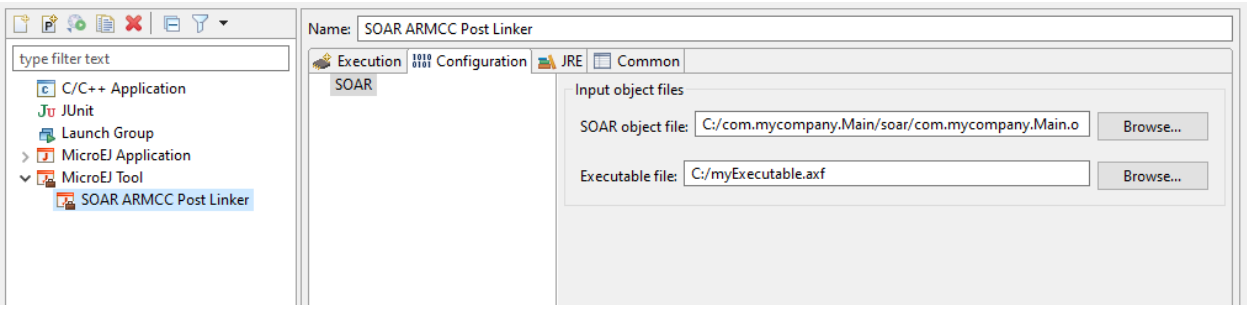


Fig. 273: SOAR debug infos post-linker tool Configuration

Once executed, it produces a new Executable file beside the original one with the .microej extension suffix

```
================ [ SOARDebugInfosPostLinker ] ================
Successfully generated c:\myExecutable.axf.microej.

SUCCESS
```

This file now contains the linked .debug.soar section so that it can be used by the debug tools.

Former Platform Migration

This chapter describes the steps to migrate a former MicroEJ Platform in its latest form described in *Create a VEE Port* chapter.

As a reminder, this new form brings two main features:

- Both Platform *build* and *dependencies declaration* are managed by *MicroEJ Module Manager*. This allows a fully automated build and continuous integration.
- The configuration of the target Board Support Package (BSP) has been revisited to support any *BSP Connection cases*.

Former MicroEJ Platforms were usually distributed by MicroEJ Corp. in an all-in-one ZIP file also called *fullPackaging*.

In this document, the MicroEJ Platform for STMicroelectronics STM32F746G-DISCO board will be used as an example.

The following figure shows the *fullPackaging* structure once extracted.

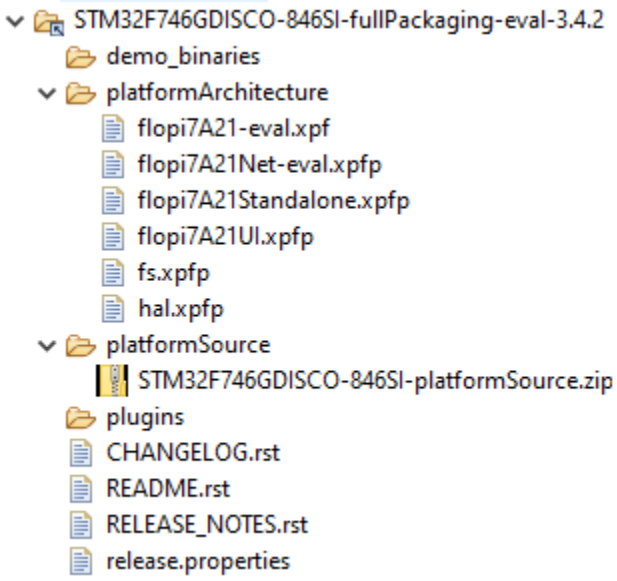


Fig. 274: STM32F746G-DISCO Platform Full Packaging Structure

The migration steps are:

1. Create a *Module Repository* to store the *MicroEJ Architecture* and *MicroEJ Packs* used by the Platform.

2. Import the Platform Configuration Additions into the Platform Configuration project.
3. Update the Front Panel project configuration.
4. Configure the *BSP Connection*.
5. Add the *Build Script* and *Run Script*.

Note: The migration of a Platform requires at least the version 5.4.0 of the SDK.

Create an Architecture Repository

The first step is to create an Architecture Repository containing the Architecture and Packs provided in the platformArchitecture directory of the *fullPackaging* package.

Note: If the Architecture and Packs used by the Platform are already stored in the module repositories provided by MicroEJ Corp (*Central Repository*, *Developer Repository*), or in your organization's repositories, then move to the next step.

By default, we provide the steps to extend the default *MicroEJ SDK settings file configuration* with local Architecture and Packs modules. The following steps can be adapted to custom *settings file*.

- Create a new empty project named architecture-repository
- Create a new file named ivysettings.xml with the following content and update the included settings file according to your MicroEJ SDK version (see *SDK Version*)

```
<?xml version="1.0" encoding="UTF-8"?>
<ivysettings>
  <property name="local.repo.url" value="${ivy.settings.dir}" override="false"/>

  <!--
     Include default settings file for MicroEJ SDK version:
     - MICROEJ SDK 5.4.0 or higher: ${user.home}/.microej/microej-ivysettings-5.4.xml
     - MICROEJ SDK 5.0.0 to 5.3.1: ${user.home}/.microej/microej-ivysettings-5.xml
     - MICROEJ SDK 4.1.x: ${user.home}/.ivy2/microej-ivysettings-4.1.xml
  -->
  <include file="${user.home}/.microej/microej-ivysettings-5.xml"/>

  <settings defaultResolver="ArchitectureResolver"/>

  <resolvers>
    <chain name="ArchitectureResolver">
      <filesystem m2compatible="true">
        <artifact pattern="${local.repo.url}/${microej.artifact.pattern}" />
        <ivy pattern="${local.repo.url}/${microej.ivy.pattern}" />
      </filesystem>
      <resolver ref="${microej.default.resolver}"/>
    </chain>
  </resolvers>
</ivysettings>
```

- Copy the Architecture file (.xpf) into the correct directory following its *naming convention*).
  - Open or extract the Architecture file (.xpf)

- Open the release.properties file to retrieve the naming convention mapping:

* architecture is the ISA (e.g. CM7)
* toolchain is the TOOLCHAIN (e.g. CM7hardfp_ARMCC5)
* name is the UID (e.g. flopi7A21)
* version is the VERSION (e.g. 7.11.0)

For example, in the STM32F746G-DISCO Platform, the Architecture file flopi7A21-eval.xpf shall be copied and renamed to architecture-repository/com/microej/architecture/CM7/CM7hardfp_ARMCC5/flopi7A21/7.11.0/flopi7A21-7.11.0-eval.xpf .

- Copy the Architecture Specific Packs files (.xpfp) into the correct directory following MicroEJ Naming Convention (see *Pack Import*) with the exception of the Standalone pack that should not be imported (e.g. named flopi7A21Standalone.xpfp).
- Open or extract the Architecture Specific Pack (.xpfp).

Note: The Architecture Specific Packs have the UID of the Architecture in their name (e.g. flopi7A21UI.xpfp) and their release_pack.properties file contains the information of the Architecture.

- Open the release_pack.properties file to retrieve the naming convention mapping:

* architecture is the ISA (e.g. CM7)
* toolchain is the TOOLCHAIN (e.g. CM7hardfp_ARMCC5)
* name is the UID (e.g. flopi7A21)
* packName is the NAME (e.g. ui)
* packVersion is the VERSION (e.g. 12.0.1)

For example, in the STM32F746G-DISCO Platform, the Architecture Specific Pack UI flopi7A21UI.xpfp shall be copied and renamed to architecture-repository/com/microej/architecture/CM7/CM7hardfp_ARMCC5/flopi7A21-ui-pack/12.0.1/flopi7A21-ui-pack-12.0.1.xpfp .

- Copy the Legacy Generic Packs (.xpfp files) into the correct directory following MicroEJ Naming Convention (see *Pack Import*).
- Open or extract the Generic Pack (.xpfp).

Note: The release_pack.properties of Legacy Generic Packs does not contain information about Architecture.

- Open the release_pack.properties file:

* packName is the NAME (e.g. fs)
* packVersion is the VERSION (e.g. 4.0.2)

For example, in the STM32F746G-DISCO Platform, the Legacy Generic Pack FS fs.xpfp shall be copied and renamed to architecture-repository/com/microej/pack/fs/4.0.2/fs-4.0.2.xpfp .

- Configure MicroEJ Module Manager to use the Architecture Repository:
- Go to Window > Preferences > MicroEJ > Module Manager

- In Module Repository set Settings File: to ${workspace_loc:architecture-repository/ivysettings.xml} .
- Apply and Close

  Here is the layout of the Architecture Repository for the STM32F746G-DISCO Platform.

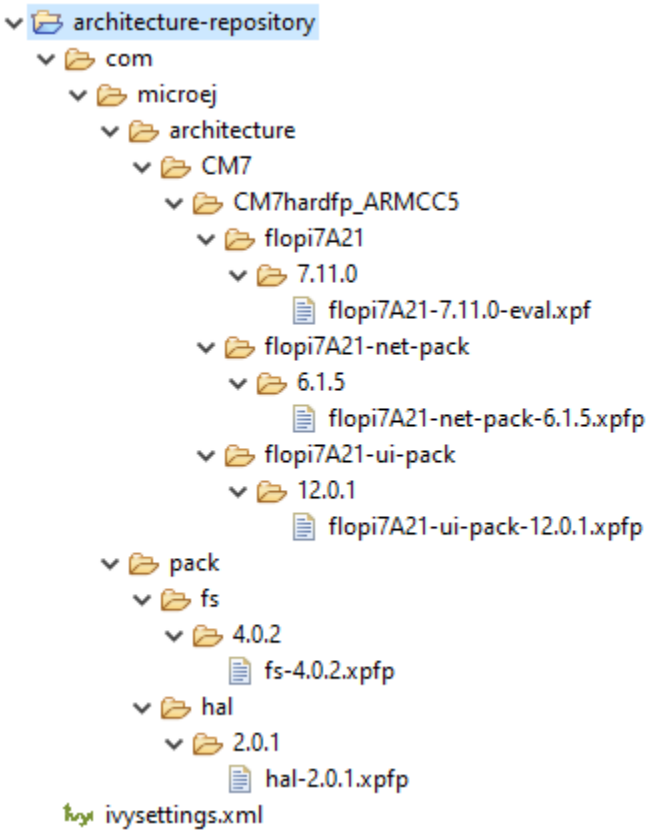
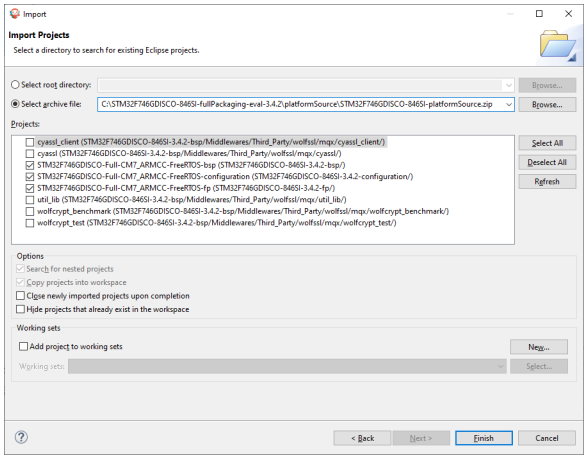


Fig. 275: Architecture Repository for STM32F746G-DISCO fullPackaging

Import the Former Platform Sources

- Go to File > Import… > General > Existing Projects into Workspace .
- Browse to the archive file that contains the platform sources, like in the example below.



- Select the -configuration , -fp and -bsp projects prefixed with the Platform name (e.g., STM32F746GDISCO-Full-CM7_ARMCC-FreeRTOS).
- Click Finish .

Install the Platform Configuration Additions

- Rename the file bsp.properties located in the Platform Configuration Project to bsp2.properties (save it for later).
- Install Platform Configuration Additions, by following instructions described at https://github.com/MicroEJ/VEEPortQualificationTools/blob/master/framework/platform/README.rst.
  Files within the content folder have to be copied to the -configuration project (e.g. STM32F746GDISCO-Full-CM7_ARMCC-FreeRTOS-configuration).
- Edit the module.properties file and set com.microej.platformbuilder.platform.filename to the name of the platform configuration file (e.g. STM32F746GDISCO.platform).
- Update the default name of the Platform module in the module.ivy . Replace with <info organisation="com.microej.platform.st.stm32f746g-disco" module="Platform" status="integration" revision="1.0.0"> .
- Update the module.ivy with the Architecture and Packs dependencies.

  Here is the module dependencies declared for the STM32F746G-DISCO Platform.

Listing 42: STM32F746GDISCO-Full-CM7_ARMCC-FreeRTOS-configuration/module.ivy

```
<dependencies>
  <!-- MicroEJ Architecture -->
  <dependency org=
↪"com.microej.architecture.CM7.CM7hardfp_ARMCC5" name="flopi7A21" rev="7.11.0">
    <artifact name=
↪"flopi7A21" m:classifier="${com.microej.platformbuilder.architecture.usage}" ext="xpf"/>
  </dependency>

  <!-- MicroEJ Architecture Specific Packs  -->
  <dependency org="com.
↪microej.architecture.CM7.CM7hardfp_ARMCC5" name="flopi7A21-ui-pack" rev="12.0.1">
    <artifact name="flopi7A21-ui-pack" ext="xpfp"/>
  </dependency>
```

(continues on next page)

(continued from previous page)

```
  <dependency org="com.
↪microej.architecture.CM7.CM7hardfp_ARMCC5" name="flopi7A21-net-pack" rev="6.1.5">
    <artifact name="flopi7A21-net-pack" ext="xpfp"/>
  </dependency>

  <!-- Legacy MicroEJ Generic Packs -->
  <dependency org="com.microej.pack" name="fs" rev="4.0.2">
    <artifact name="fs" ext="xpfp"/>
  </dependency>
  <dependency org="com.microej.pack" name="hal" rev="2.0.1">
    <artifact name="hal" ext="xpfp"/>
  </dependency>

</dependencies>
```

Update the Front Panel Configuration

- In -configuration/frontpanel/frontpanel.properties set the project.name to the folder name that contains the front-panel (e.g. project.name=STM32F746GDISCO-Full-CM7_ARMCC-FreeRTOS-fp).

  At this state, the Platform is not connected to the BSP yet, but you can check that everything is properly configured so far by building it:

- Right-click on the -configuration project and select Build Module
- Import the Platform built into the workspace by following the instructions available at the end of the build logs (see logs example below).

```
module-platform:report:
␣
 ↪␣
 ↪␣
 ↪␣
 ↪[echo]␣
 ↪␣
 ↪  ␣
 ↪==========================================================================
    [echo]      Platform has␣
 ↪been built in this directory 'C:\STM32F746GDISCO-Platform-CM7hardfp_ARMCC5-0.1.0'.
    [echo]      To import this project in your MicroEJ SDK workspace (if not already available):
    [echo]␣
 ↪      - Select 'File' > 'Import...' > 'General' > 'Existing Projects into Workspace' > 'Next'
    [echo]       - Check 'Select root directory
 ↪' and browse 'C:\STM32F746GDISCO-Platform-CM7hardfp_ARMCC5-0.1.0' > 'Finish'
␣
 ↪␣
 ↪␣
 ↪␣
 ↪[echo]␣
 ↪␣
 ↪  ␣
 ↪==========================================================================
```

At this stage the Platform is built and imported in the workspace, so you can create a Standalone Application and run it on the Simulator (see *Create a MicroEJ Standalone Application*).

Note: If the build failed, it might be because the Architecture and Packs can not be retrieved from the Architecture Repository. Ensure that the Architecture Repository is correctly configured and that it contains the required artifacts (as described in *the first step*).

Configure the BSP Connection

This section explains how to configure a full BSP Connection on the STM32F746G-DISCO Platform. See *BSP Connection* for more information.

- Open -configuration/bsp/bsp.properties .
- Comment out and set the following variables:

– root.dir

– microejapp.relative.dir

– microejlib.relative.dir

– microejinc.relative.dir

– microejscript.relative.dir

For example:

```
# Specify the MicroEJ Application file ('microejapp.o') parent directory.
# This is a '/' separated directory relative to 'bsp.root.dir'.
microejapp.relative.dir=Projects/STM32746G-Discovery/Applications/MicroEJ/platform/lib

# Specify the MicroEJ Platform runtime file ('microejruntime.a') parent directory.
# This is a '/' separated directory relative to 'bsp.root.dir'.
microejlib.relative.dir=Projects/STM32746G-Discovery/Applications/MicroEJ/platform/lib

# Specify MicroEJ Platform header files ('*.h') parent directory.
# This is a '/' separated directory relative to 'bsp.root.dir'.
microejinc.relative.dir=Projects/STM32746G-Discovery/Applications/MicroEJ/platform/inc

# Specify BSP external scripts files ('build.bat' and 'run.bat') parent directory.
# This is a '/' separated directory relative to 'bsp.root.dir'.
microejscript.relative.dir=Projects/STM32746G-Discovery/Applications/MicroEJ/scripts

# Specify the BSP root directory.
↪ Can use ${project.parent.dir} which target the parent of platform configuration project
# For example, '${project.parent.
↪dir}/PROJECT-NAME-bsp' specifies a BSP project beside the '-configuration' project
root.dir=${project.parent.dir}/STM32F746GDISCO-Full-CM7_ARMCC-FreeRTOS-bsp/
```

The paths to microejXXX.relative.dir can be inferred by looking at the output.dir value in bsp2.properties saved earlier. For example on the STM32F746G-DISCO project, its value is ${workspace}/${project.prefix}-bsp/Projects/STM32746G-Discovery/Applications/MicroEJ/platform .

- The BSP project path ${workspace}/${project.prefix}-bsp becomes ${project.parent.dir}/STM32F746GDISCO-Full-CM7_ARMCC-FreeRTOS-bsp/ .
- Projects/STM32746G-Discovery/Applications/MicroEJ/platform is the path to the Application file, Platform header and runtime files. MicroEJ convention is to put the Application file and Platform runtime files to platform/lib/ and the Platform header files to platform/inc/ .

- *Build Script File* and *Run Script File* are PCA-specific and did not exist before. By convention we put them in a scripts/ directory.

  The paths to microejXXX.relative.dir can be also be checked by looking at the C TOOLCHAIN configuration of the BSP. For example on the STM32F746G-DISCO project, the BSP configuration is located at STM32F746GDISCO-Full-CM7_ARMCC-FreeRTOS-bsp/Projects/STM32746G-Discovery/Applications/MicroEJ/MDK-ARM/Project.uvprojx .

- In Project > Options for Target 'standalone'… > C/C++ > Include Paths contains ../platform/inc . This corresponds to the microejinc.relative.dir relative the TOOLCHAIN project's file.

- In the Project pane, there is a folder MicroEJ/Libs that contains microejruntime.lib and microejapp.o .

  – Right-click on microejruntime.lib > Options for File 'XXX'… . The Path is ../platform/lib/microejruntime.lib . This corresponds to the microejlib.relative.dir .

  – Right-click on microejapp.o > Options for File 'XXX'… . The Path is ../platform/lib/microejapp.o . This corresponds to the microejapp.relative.dir .

- Rebuild the platform (Right-click on the -configuration project and select Build Module)

  At this stage the Platform is connected to the BSP so you can build and program a Firmware (see *Run on the Device*).

Add the Build and Run Scripts

The final stage consists of adding the *Build Script*, to automate the build of a Firmware, and the *Run Script*, to automate the programming of a MicroEJ Firmware onto the device.

The Platform Qualification Tools provides examples of Build Script and Run Script for various C TOOLCHAIN here. *This training* also describes the steps to create and use these scripts.

On the STM32F746G-DISCO, the C TOOLCHAIN used is Keil uVision.

- Create the directory pointed by microejscript.relative.dir (e.g. STM32F746GDISCO-Full-CM7_ARMCC-FreeRTOS-bsp\Projects\STM32746G-Discovery\Applications\MicroEJ\scripts).
- Copy the example scripts from the Platform Qualification Tools for the C TOOLCHAIN of the BSP (e.g. PlatformQualificationTools/framework/platform/scripts/KEILuV5/)
- Configure the scripts. Refer to the documentation in the scripts comments for this step.
- Enable the execution of the build script:

  – Go to Run > Run Configurations…

  – Select the launch configuration

  – Go to Configuration > Device > Deploy

  – Ensure Execute the MicroEJ build script (build.bat) at a location known by the 3rd-party BSP project. is checked.

Use the Platform in Module Projects

Module projects may require the Platform, for example to build an Application or to run a Testsuite. One way of selecting the Platform in a module project is to declare it as a module dependency (see *Select a VEE Port*).

In case a former Platform is loaded this way in your existing module projects, the dependency has to be updated. In this example, the Platform would now be selected like this:

```
<dependency org="com.microej.platform.st.stm32f746g-
↪disco" name="Platform" rev="1.0.0" conf="platform->default" transitive="false"/>
```

This also requires that your module projects use a compatible version of the associated build type (the build type relates to the *Module Natures*). As stated before, loading a Platform in its latest form requires at least the version 5.4.0 of the SDK. Therefore, make sure to use versions of the build types that come with the SDK 5.4.0 and above. Here is a brief summary of the minimum version for the most common module natures:

- *Add-On Library*: build type com.is2t.easyant.buildtypes#build-microej-javalib version 5.0.0 and above.
- *Standalone Application*: build type com.is2t.easyant.buildtypes#build-firmware-singleapp version 1.4.0 and above.
- *Sandboxed Application*: build type com.is2t.easyant.buildtypes#build-application version 8.0.0 and above.

Going further

Now that the Platform is connected to the BSP it can leverage the Java Testsuites provided by the Platform Qualification Tools. See *Configure and Run the Testsuite* documentation for a step by step explanation on how to do so.

Architecture 8.0.0 Migration

This chapter describes the steps to migrate a VEE Port from Architecture 8.0.0 to Architecture 8.1.0 .

As a reminder, refer to the *Architecture 8.1.0 Changelog* section for the complete list of changes and updates.

Migrate Core Engine Capability Configuration

The selection of the *Core Engine capability* is now done via the property com.microej.runtime.capability . Refer to one of the sections below depending on your desired capability.

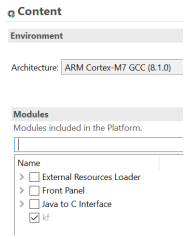
If you use the property com.microej.platformbuilder.module.multi.enabled , update your Platform Configuration Additions to the version 2.1.0 or higher. It is also recommended to delete the property com.microej.platformbuilder.module.multi.enabled and to use the property com.microej.runtime.capability instead.

Mono-Sandbox

Mono-Sandbox remains the default capability and no changes are required to your VEE Port configuration.

Multi-Sandbox

In the Platform Editor, the Multi Applications (kf) module now appears in gray:



Unselect the kf module and follow the instructions from the *Multi-Sandbox installation* section.

Tiny-Sandbox

The property mjvm.standalone.configuration used to select the Tiny-Sandbox capability is now deprecated. It is recommended to remove the definition of this property from the configuration.xml file and follow the instructions from the *Tiny-Sandbox installation* section.

Migrate Your LLKERNEL Implementation

This section only applies if your LLKERNEL was based on legacy *In-Place Installation mode*. The Kernel Working Buffer no longer exists. The functions LLKERNEL_IMPL_allocateWorkingBuffer() and LLKERNEL_IMPL_freeWorkingBuffer() are no more called and can be simply removed from your implementation.

Memory allocation for the Features will now use the function LLKERNEL_IMPL_allocateFeature() . The following code is a LLKERNEL_impl.c template for migrating your current implementation using this API. The code logic based on a malloc/free implementation does not need to be changed.

```
#include <stdlib.h>
#include <string.h>

#include "LLKERNEL_impl.h"

// Your implementation of malloc()
#define KERNEL_MALLOC(size) malloc((size_t)(size))

// Your implementation of free()
#define KERNEL_FREE(addr) free((void*)(addr))

// Your implementation of 'ASSERT(0)'
#define KERNEL_ASSERT_FAIL() while(1)
```

(continues on next page)

(continued from previous page)

```
// Utility macros for allocating RAM and ROM areas with required alignment constraints
#define KERNEL_AREA_GET_MAX_SIZE(size, alignment) ((size)+((alignment)-1))
#define KERNEL_AREA_GET_START_
↪ADDRESS(addr, alignment) ((void*)((((int32_t)(addr))+(alignment)-1)&~((alignment)-1)))

typedef struct installed_feature{
  void* ROM_area;
  void* RAM_area;
} installed_feature_t;

int32_t LLKERNEL_IMPL_allocateFeature(int32_t size_ROM, int32_t size_RAM) {
  int32_t ret = 0;
  int total_size = sizeof(installed_feature_t);
  total_size += KERNEL_
↪AREA_GET_MAX_SIZE(size_ROM, LLKERNEL_ROM_AREA_ALIGNMENT);
  total_size += KERNEL_
↪AREA_GET_MAX_SIZE(size_RAM, LLKERNEL_RAM_AREA_ALIGNMENT);

  void* total_area = KERNEL_MALLOC(total_size);
  if(NULL != total_area){
    installed_feature_t* f = (installed_feature_t*)total_area;
    f->ROM_area = KERNEL_AREA_GET_START_ADDRESS((void*)(((int32_
↪t)f)+((int32_t)sizeof(installed_feature_t))), LLKERNEL_ROM_AREA_ALIGNMENT);
    f->RAM_area = KERNEL_AREA_GET_START_ADDRESS((void*)(((int32_
↪t)f->ROM_area)+size_ROM), LLKERNEL_RAM_AREA_ALIGNMENT);
    ret = (int32_t)f;
  } // else out of memory

  return ret;
}

void LLKERNEL_IMPL_freeFeature(int32_t handle) {
  KERNEL_FREE(handle);
}

int32_t LLKERNEL_IMPL_getAllocatedFeaturesCount(void) {
  // No persistency support
  return 0;
}

int32_t LLKERNEL_IMPL_getFeatureHandle(int32_t allocation_index) {
  // No persistency support
  KERNEL_ASSERT_FAIL();
}

void* LLKERNEL_IMPL_getFeatureAddressRAM(int32_t handle) {
  return ((installed_feature_t*)handle)->RAM_area;
}

void* LLKERNEL_IMPL_getFeatureAddressROM(int32_t handle) {
  return ((installed_feature_t*)handle)->ROM_area;
}

int32_t LLKERNEL_
↪IMPL_copyToROM(void* dest_address_ROM, void* src_address, int32_t size) {
  memcpy(dest_address_ROM, src_address, size);
```

(continues on next page)

(continued from previous page)

```
  return LLKERNEL_OK;
}

int32_t LLKERNEL_IMPL_flushCopyToROM(void) {
  return LLKERNEL_OK;
}

int32_
↪t LLKERNEL_IMPL_onFeatureInitializationError(int32_t handle, int32_t error_code) {
  // No persistency support
  KERNEL_ASSERT_FAIL();
  return 0;
}
```

Architecture 7.x Migration

This chapter describes the steps to migrate a VEE Port from Architecture 7.x to Architecture 8.0.0 .

As a reminder, refer to the *Architecture 8.0.0 Changelog* section for the complete list of changes and updates.

Update Platform Configuration Additions

Architecture 8.0.0 now directly integrates the *BSP Connection* mechanism. Consequently, Platform Configuration Additions files have been separated in two directories:

- content-sdk-5 : files required for building the VEE Port using SDK 5.x (MMM)
- content-architecture-7 : files required for building the Executable using Architecture 7.x.

  See https://github.com/MicroEJ/VEEPortQualificationTools/blob/master/framework/platform/README.rst for more details.

  Your VEE Port must be updated to remove files that are now included in Architecture 8:

  - Delete [name]-configuration/build/module/module-dropins directory.
  - Delete [name]-configuration/build/module/module-dropins.ant file.
  - Delete [name]-configuration/build/platform/platform-deploy.ant file.
  - Delete [name]-configuration/build/platform/platform-kf.ant file.
  - Download the latest content-sdk-5 directory. Your local files must be overridden.
  - Edit your *module.ivy* and put back your module *name*, *version*, *organisation* and *<dependencies>* content.
  - Edit your *module.properties* and put back your options (if they have changed from default ones).
  - Delete the following files from your [name]-configuration/dropins directory:
    - scripts/init-bsp/*
    - scripts/init-license-checker/*
    - scripts/checkOS.xml
    - scripts/deployInBSP.xml

- scripts/deployInBSPCommon.xml
- scripts/deployToolBSPRun*
- scripts/fullLink*
- tools/license-checker.jar
- workbenchExtension-launchScriptFramework.jar

  - Rebuild your VEE Port.
  - Rebuild your Executable.

Update BSP with new Sections Names

The Core Engine sections have been renamed to respect the standard ELF convention. See *Core Engine Link* section for further details.

All references to section names in your BSP must be updated. This is usually only used in your linker script file, but section names are sometimes also hardcoded in the C Code. Here is an example of a GNU LD script highlighting the typical changes that must be made:

```
/* Linker Script Template for Architecture 7.x */
SECTIONS
{
  /* .text sections (code) */
  .text_ro :
  {
    . = ALIGN(4);
    *(.text)
    *(.text*)
  } >ROM

  /* .rodata sections (constants, strings, etc.) */
  .rodata_ro :
  {
    . = ALIGN(4);
    *(.rodata)
    *(.rodata*)
    *(.text.soar)
    *(.rodata.resources)
  } >ROM

  /* readwrite region initializers (mandatory) */
  .data :
  {
    . = ALIGN(4);
    *(.data)
    *(.data*)
  } >RAM AT> ROM

  /* .bss sections (statics, buffers, etc.) */
  .bss_rw :
  {
    . = ALIGN(4);
    *(.bss)
    *(.bss*)
    *(COMMON)
    *(.bss.vm.stacks.java)
    *(ICETEA_HEAP)
    *(_java_heap)
    *(_java_immortals)
  } >RAM
}
```

```
/* Linker Script Template for Architecture 8.x */
SECTIONS
{
  /* .text sections (code) */
  .text_ro :
  {
    . = ALIGN(4);
    *(.text)
    *(.text*)
  } >ROM

  /* .rodata sections (constants, strings, etc.) */
  .rodata_ro :
  {
    . = ALIGN(4);
    *(.rodata)
    *(.rodata*)
    *(.rodata.microej.soar)
    *(.rodata.microej.resource.*)
  } >ROM

  /* readwrite region initializers (mandatory) */
  .data :
  {
    . = ALIGN(4);
    *(.data)
    *(.data*)
  } >RAM AT> ROM

  /* .bss sections (statics, buffers, etc.) */
  .bss_rw :
  {
    . = ALIGN(4);
    *(.bss)
    *(.bss*)
    *(COMMON)
    *(.bss.microej.stacks)
    *(.bss.microej.runtime)
    *(.bss.microej.heap)
    *(.bss.microej.immortals)
  } >RAM
}
```

Fig. 276: Example of LD Script File Migration

Remove LLBSP_IMPL_isInReadOnlyMemory

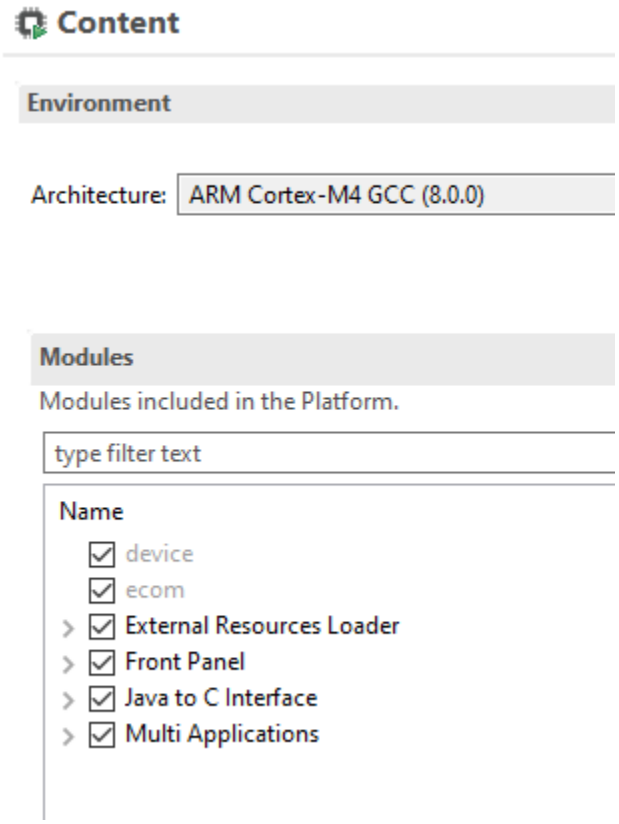
The `LLBSP_IMPL_isInReadOnlyMemory` function has been removed since it is no more called by the Core Engine. You can simply remove your implementation function.

Migrate Built-in Modules

The following built-in legacy modules have been removed from the Architecture:

- Device
- ECOM-COMM

In the Platform Editor, these modules now appear in gray with Architecture `8.x` :



To remove these modules, open the `.platform` file using a text editor and remove the following XML elements:

```
<group name="device"/>
<group name="ecom"/>
```

Migrate Device Module

The latest Device Pack available on the *Central Repository* is backward compatible with the built-in Architecture module.

The following dependency must be added to the *module.ivy* of the VEE Port configuration project:

```
<dependency org="com.microej.pack.device" name="device-pack" rev="1.1.1" />
```

Migrate ECOM-COMM Module

The Foundation Library ECOM-COMM-1.1 has been removed from *Architecture 8.0.0*. It is now replaced by ECOM-COMM-2.0 which is distributed in its own Pack.

There are two migration options:

- either migrate to the latest ECOM-COMM-2.0 Pack,
- or integrate the legacy ECOM-COMM-1.1 Pack files as-is into your VEE Port dropins directory.

Contact our support team to get the best migration strategy and detailed instructions.

Migrate Your LLKERNEL Implementation

The following code is a LLKERNEL_impl.c template for migrating your current implementation of Feature installation in RAM. This is now called *In-Place Installation*. Your code logic for managing allocated blocks does not need to be changed. As there is no installation in ROM, most of the new functions do not need to be implemented.

```
#include "LLKERNEL_impl.h"

void* LLKERNEL_IMPL_allocateWorkingBuffer(int32_t size) {
  // Paste here the code of your former 'LLKERNEL_IMPL_allocate' function
}

void LLKERNEL_IMPL_freeWorkingBuffer(void* chunk_address) {
  // Paste here the code of your former 'LLKERNEL_IMPL_free' function
}

int32_t LLKERNEL_IMPL_allocateFeature(int32_t size_ROM, int32_t size_RAM) {
  return 0;
}

int32_t LLKERNEL_IMPL_getAllocatedFeaturesCount(void) {
  return 0;
}

void LLKERNEL_IMPL_freeFeature(int32_t handle) {
  // Paste here your implementation of 'ASSERT(0)'
}

int32_t LLKERNEL_IMPL_getFeatureHandle(int32_t allocation_index) {
  // Paste here your implementation of 'ASSERT(0)'
```

(continues on next page)

(continued from previous page)

```
  return 0;
}

void* LLKERNEL_IMPL_getFeatureAddressRAM(int32_t handle) {
  // Paste here your implementation of 'ASSERT(0)'
  return 0;
}

void* LLKERNEL_IMPL_getFeatureAddressROM(int32_t handle) {
  // Paste here your implementation of 'ASSERT(0)'
  return 0;
}

int32_t LLKERNEL_
↪IMPL_copyToROM(void* dest_address_ROM, void* src_address, int32_t size) {
  // Paste here your implementation of 'ASSERT(0)'
  return 0;
}

int32_t LLKERNEL_IMPL_flushCopyToROM(void) {
  // Paste here your implementation of 'ASSERT(0)'
  return 0;
}

int32_
↪t LLKERNEL_IMPL_onFeatureInitializationError(int32_t handle, int32_t error_code) {
  // Paste here your implementation of 'ASSERT(0)'
  return 0;
}
```

Migrate Trace C Library Usage

In Architecture 8.0.0, the Trace C library's version has been updated from 1.0.0 to 2.0.0. This new version introduces the following backward incompatible changes:

- C header file trace.h has been renamed into LLTRACE.h.
- The functions declared in this header have been renamed from TRACE_xxx to LLTRACE_xxx.

If you have included trace.h in a C file, the compilation will fail with an error message similar to one of the following messages:

- fatal error: trace.h: No such file or directory
- Fatal Error[Pe1696]: cannot open source file "trace.h"

To fix this issue, you can either migrate to version 2.0.0 of the Trace library or provide a backward compatibility layer.

To migrate to version 2.0.0, you need to make the following changes:

- Replace the directives #include "trace.h" with #include "LLTRACE.h".
- Replace any references to the TRACE_xxx functions (e.g., TRACE_record_event_void) with references to the corresponding LLTRACE_xxx function (e.g., LLTRACE_record_event_void).

If you decide not to modify existing code, you can create and add to your project a `trace.h` file with the following content:

```
#ifndef TRACE_H
#define TRACE_H

/**
 * Trace library API backward compatibility layer.
 * Allows to use Trace API 1.0.0 (Architecture 7.x) in a VEE Port
 * that includes Trace API 2.0.0 (Architecture 8.x).
 */

#include "LLTRACE.h"

#ifdef __cplusplus
	extern "C" {
#endif

#define TRACE_start LLTRACE_start
#define TRACE_start LLTRACE_start
#define TRACE_stop LLTRACE_stop
#define TRACE_is_started LLTRACE_is_started
#define TRACE_declare_event_group LLTRACE_declare_event_group
#define TRACE_record_event_void LLTRACE_record_event_void
#define TRACE_record_event_u32 LLTRACE_record_event_u32
#define TRACE_record_event_u32x2 LLTRACE_record_event_u32x2
#define TRACE_record_event_u32x3 LLTRACE_record_event_u32x3
#define TRACE_record_event_u32x4 LLTRACE_record_event_u32x4
#define TRACE_record_event_u32x5 LLTRACE_record_event_u32x5
#define TRACE_record_event_u32x6 LLTRACE_record_event_u32x6
#define TRACE_record_event_u32x7 LLTRACE_record_event_u32x7
#define TRACE_record_event_u32x8 LLTRACE_record_event_u32x8
#define TRACE_record_event_u32x9 LLTRACE_record_event_u32x9
#define TRACE_record_event_u32x10 LLTRACE_record_event_u32x10
#define TRACE_record_event_end LLTRACE_record_event_end
#define TRACE_record_event_end_u32 LLTRACE_record_event_end_u32

#ifdef __cplusplus
	}
#endif

#endif //TRACE_H
```

Migrate Legacy System Properties Files

Legacy System Properties files (`*.system.properties`) are no more supported by Architecture `8.0.0` . These files must be renamed to `*.properties.list` files (see *System Properties* for more details).

To facilitate the migration, legacy System Properties files are detected by SOAR and the following error is thrown:

```
1 : SOAR-L ERROR :
[M78] - System properties file [properties/xxx.
↪system.properties] in classpath entry [...] must be renamed to [properties/xxx.properties.list].
```

The following modules declare legacy System Properties files in older versions. Make sure to

update the module to the specified version or a newer release in your projects.

- Pack NET version 9.4.2 .
- Add-On library eclasspath-logging version 1.2.1 .
- Testsuite FS version 3.0.7 .

Update Application Options

The following Application Options must be updated in the Applications configuration when upgrading to Architecture 8.0.0 :

- Replace the com.microej.soar.groupMethodsByType.enabled property by com.microej.soar.groupMethods.enabled . See *Option(checkbox): Group Methods* for more details.

C Modules Installation

This section describes how to install a C module on your VEE Port, depending on your SDK version the first steps may vary.

Fetching the module source files

SDK 5

SDK 6

Under the SDK 5, follow these steps:

- Go to the location of your C module, for example the C module for microUI 14.2.0 is located here.
- Get the dependency declaration, for the previous example this would be: <dependency org="com.microej.clibrary.llimpl" name="microui" rev="14.2.0" /> .
- Add it inside the block <dependencies> from the file module.ivy of your VEE Port *-configuration project.
- **If you update a C module:** Remove the .properties file in the folder *-bsp/projects/microej corresponding to the desired C module. For example with microui C module, its .properties file is named cco_microui.properties .
- Rebuild the VEE Port: in the SDK 5 Project Explorer, right click on the VEE Port module *-configuration > build module .

Under the SDK 6, follow these steps:

- Go to the location of your C module, for example the C module for microUI 14.2.0 is located here.
- Download the archive file with the .cco extension.
- Unzip the content of this file.
- The source files are located in the folder bsp/ .
- Copy the **content** of the bsp/ folder into your VEE Port at the path bsp/vee/port/ .

C module configuration and firmware build

- Update the toolchain build of your BSP (IAR, CMake, etc…) to include any new files if this is the case.
- Configure the C module if required (typically, the configuration is located in files suffixed with *_configuration.h).
- Now you should be able to build your BSP firmware.

3.7 Kernel Developer Guide

3.7.1 Overview

Introduction

The Kernel Developer's Guide describes how to create a Kernel Application. A Kernel Application is a *Standalone Application* that can be extended (statically or dynamically) to install, run, and control the execution of new applications called *Sandboxed Applications*.

The intended audience of this document are Java developers and system architects who plan to design and build their own Kernel.

Here is a non-exhaustive list of the activities to be done by Kernel Developers:

- Integrating the Kernel Application with a VEE Port to produce a Multi-Sandbox Executable and Virtual Device
- *Defining the set of APIs* that will be exposed to Applications, optionally by maintaining a custom *Runtime Environment*
- Managing lifecycles of applications (deciding when to install, start, stop and uninstall them)
- Defining and applying permissions on system resources (rules & policies)
- Managing connectivity
- Controlling and monitoring resources

This document takes as prerequisite that a VEE Port is available for the target device (see *VEE Porting Guide*). This document also assumes that the reader is familiar with the development and deployment of Applications (see *Application Developer Guide*) and specifics of developing Sandboxed Applications (see *Sandboxed Application*).

Terms and Definitions

A *Multi-Sandbox VEE Port* is a VEE Port with the Multi-Sandbox capability of the Core Engine enabled (see the chapter *Multi-Sandbox* of the *VEE Porting Guide*). A Multi-Sandbox Executable can only be built with a Multi-Sandbox VEE Port.

A *Virtual Device* is the Multi-Sandbox Executable counterpart for developing a Sandboxed Application. It provides the Kernel functional simulation part. Usually it also provides a mean to directly deploy a Sandboxed Application on the target device running the Multi-Sandbox Executable (this is called *Local Deployment*).

Overall Architecture

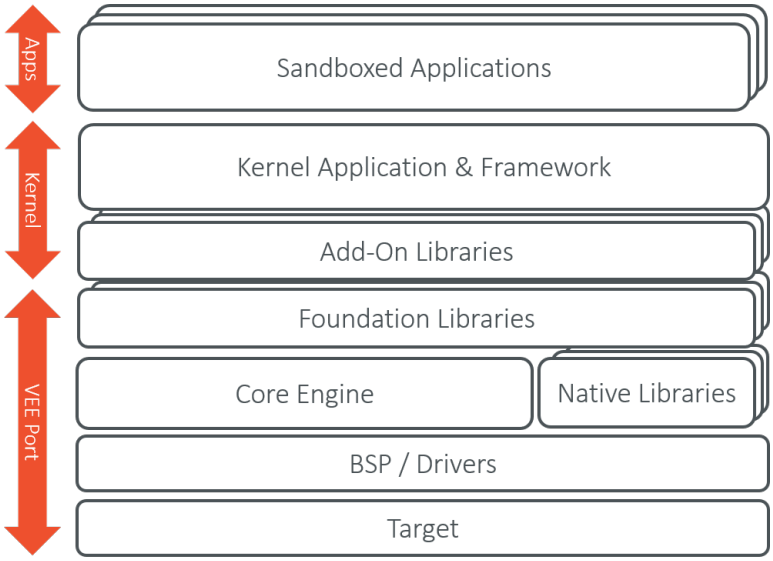


Fig. 277: Kernel Boundary Overview

Input and Output Artifacts

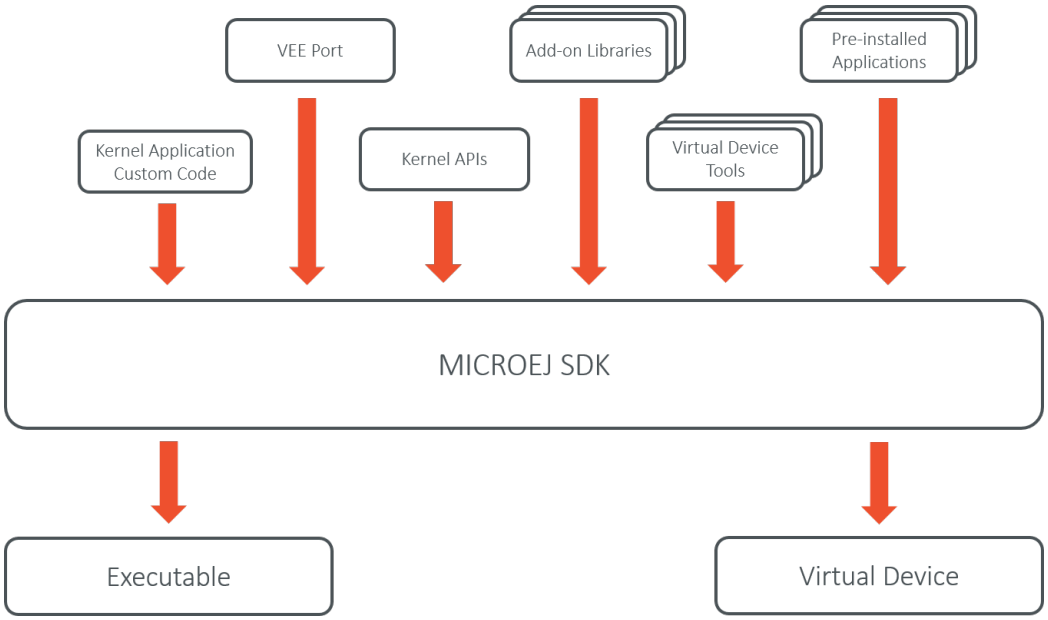


Fig. 278: Kernel Input and Output Artifacts

Kernel Build Flow

The following describes the Kernel build flow.

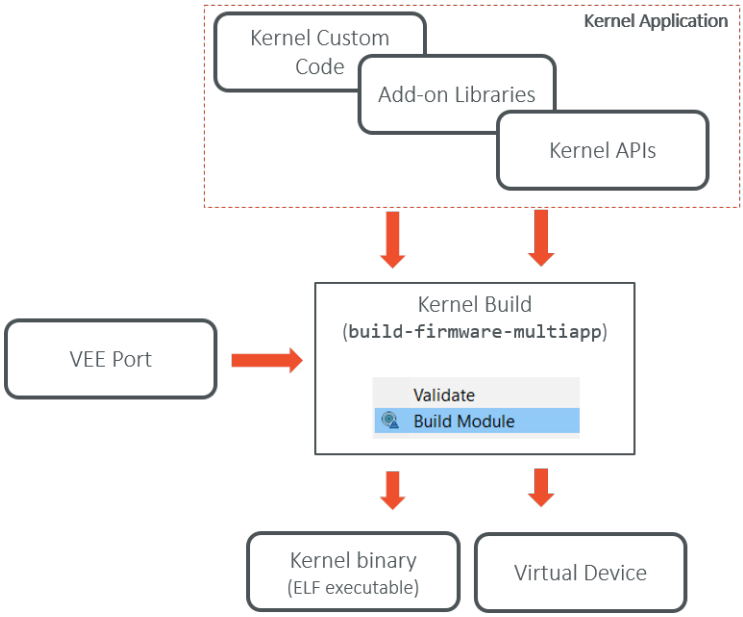


Fig. 279: Kernel Build Flow

The Virtual Device builder performs the following steps:

- Remove the embedded part of the VEE Port (including MEJ32).
- Append Add-On Libraries and *pre-installed Applications* into the runtime classpath. See *Kernel Module Configuration* section for specifying the dependencies.
- Add a custom license allowing Virtual Device redistribution.
- Generate the Runtime Environment from the Kernel APIs.

Kernel Implementation Libraries

Kernel implementations must cover the following topics:

- The kernel entry point implementation, that deals with configuring the different policies, registering kernel services and converters, and starting applications.
- The storage infrastructure implementation: mapping the Storage service on an actual data storage implementation. There are multiple implementations of the data storage, provided in different artifacts that will be detailed in dedicated sections.
- The applications management infrastructure: how application code is stored in memory and how the lifecycle of the code is implemented. Again, this has multiple alternative implementations, and the right module must be selected at build time to cover the specific Kernel needs.
- The simulation support: how the Virtual Device implementation reflects the Executable implementation, with the help of specific artifacts.
- The Kernel API definition: not all the classes and methods used to implement the Kernel Application are actually exposed to the Sandboxed Applications. There are some artifacts available that expose some of the libraries to the applications, these ones can be picked when the Kernel is assembled.

- The Kernel types conversion and other KF-related utilities: Kernel types instances owned by one application can be transferred to another application through a Shared Interface. For that to be possible, a conversion proxy must be registered for this kernel type.
- Tools libraries: tools that plug into the SDK, extending them with features that are specific to the Kernel, like deployment of an application, a management console, …

Getting started

Get familiar with Kernel development by following the training *Get Started with Kernel*.

3.7.2 Kernel Creation

Create a new Project

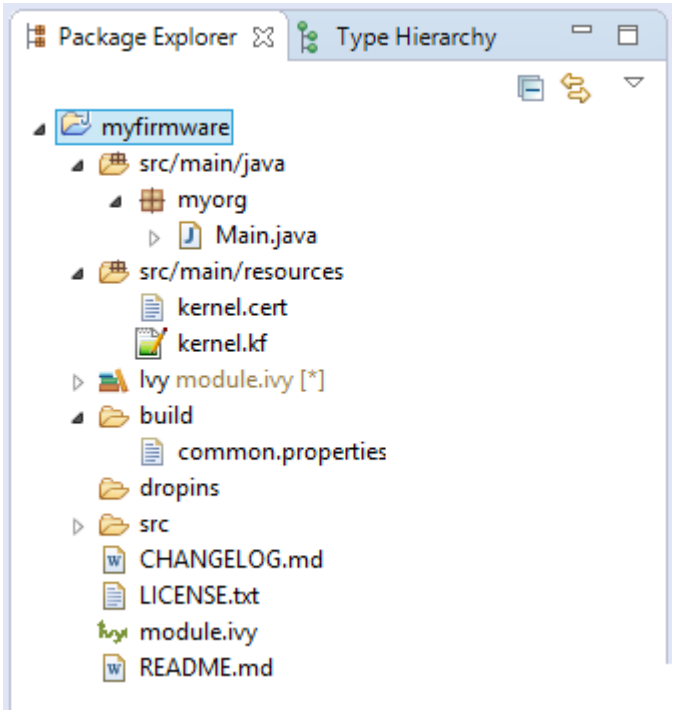
To create a new Kernel project:

SDK 6

SDK 5

Create a new *Application* following the steps described in *SDK 6 User Guide - Create a Project* depending on your IDE.

- First create a new *Kernel Application*.
- A new project is generated into the workspace:



Configure a VEE Port

Before building the Kernel, you need to have a VEE Port with Multi-Sandbox capability. To enable the Multi-Sandbox capability in your VEE Port configuration, follow the instructions from the *Multi-Sandbox* section.

Once the VEE Port is available, configure the target VEE Port in your Kernel project:

- To configure a VEE Port with SDK 6, see *Select a VEE Port*.
- To configure a VEE Port with SDK 5, see *Select a VEE Port*.

Build the Executable and Virtual Device

The Multi-Sandbox Executable and the Virtual Device of the Kernel Application must be built to allow the development and execution of Sandboxed Applications.

SDK 6

SDK 5

- The Executable can be built by executing the Gradle `buildExecutable` task:

```
./gradlew buildExecutable
```

The Executable is generated in the `build/application/executable` folder of the project. For more information about the build of the Executable depending on your IDE, refer to the *Build an Executable* page.

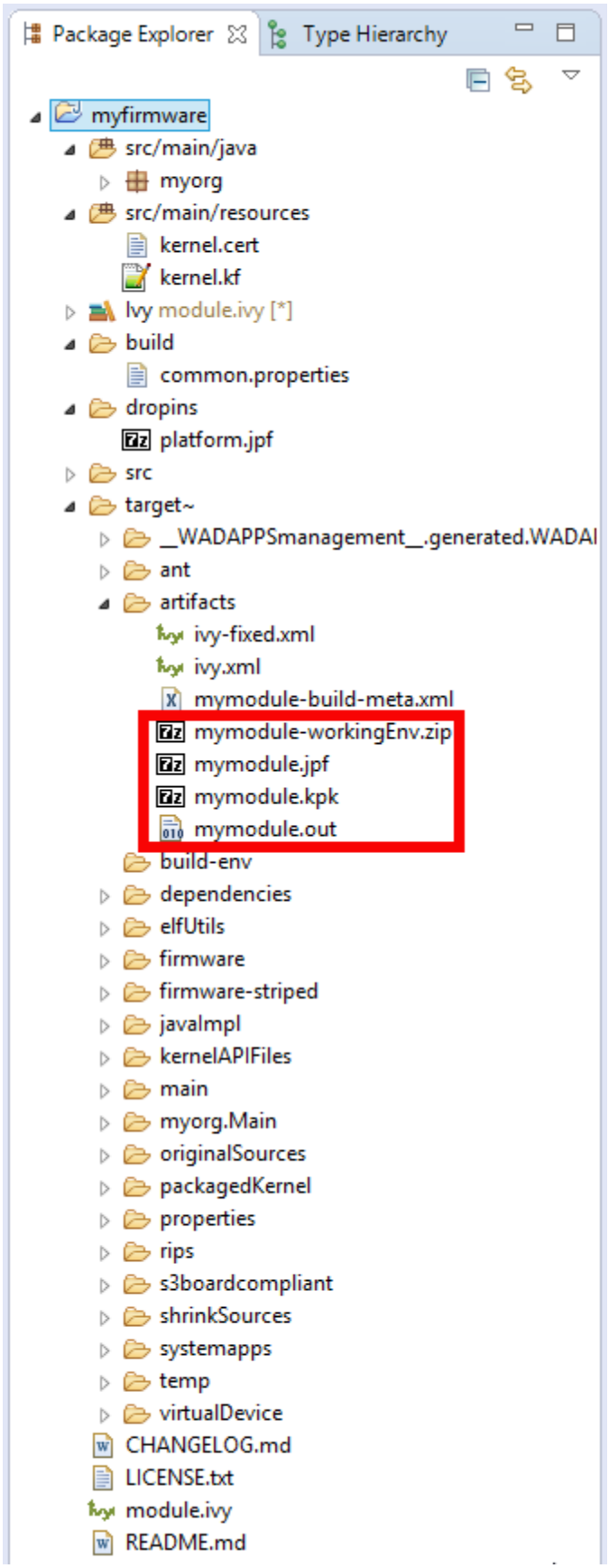
- The Virtual Device can be built by executing the Gradle `buildVirtualDevice` task:

```
./gradlew buildVirtualDevice
```

The Virtual Device is generated in the `build/virtualDevice` folder of the project. For more information about the build of the Virtual Device depending on your IDE, refer to the *Build a Virtual Device* page.

In the Package Explorer, right-click on the project and select `Build Module`. The build of the Executable and Virtual Device may take several minutes. Once the build has succeeded, the folder `myfirmware` > `target~` > `artifacts` contains the firmware output artifacts (see *Input and Output Artifacts*) :

- `mymodule.out` : The Executable to be programmed on device.
- `mymodule.kpk` : The Kernel package to be imported in a MicroEJ Forge instance.
- `mymodule.vde` : The Virtual Device to be imported in the SDK.
- `mymodule-workingEnv.zip` : This file contains all files produced by the build phase (intermediate, debug and report files).


Package Explorer
Type Hierarchy
myfirmware
src/main/java
myorg
src/main/resources
kernel.cert
kernel.kf
Ivy module.ivy [*]
build
common.properties
dropins
platform.jpf
src
target~
__WADAPPSmanagement__.generated.WADAI
ant
artifacts
ivy-fixed.xml
ivy.xml
mymodule-build-meta.xml
mymodule-workingEnv.zip
mymodule.jpf
mymodule.kpk
mymodule.out
build-env
dependencies
elfUtils
firmware
firmware-striped
javaImpl
kernelAPIFiles
main
myorg.Main
originalSources
packagedKernel
properties
rips
s3boardcompliant
shrinkSources
systemapps
temp
virtualDevice
CHANGELOG.md
LICENSE.txt
module.ivy
README.md

Build the Executable in the Workspace

Note: The build of the Executable in the Workspace is available for SDK 5 only. Refer to the *Build an Executable* page to build the Executable with SDK 6.

It is possible to build the Executable using a *MicroEJ Launch* rather than the regular module build. This speeds-up the build time thanks to MicroEJ Module Manager workspace resolution and Eclipse incremental compilation.

- Import the Kernel project and all Sandboxed Application projects in the same workspace,
- Prepare a MicroEJ Application launch for the Kernel as a regular *Standalone Application*,
- Prepare a MicroEJ Application launch for each Sandboxed Application using *Build Dynamic Feature* settings.

The following figure shows the overall build flow:

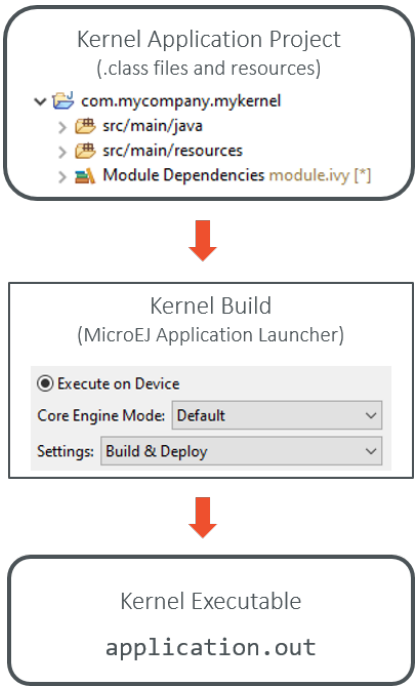


Fig. 280: Kernel Build Flow using MicroEJ Launches

Expose APIs

A Kernel must define the set of classes, methods and static fields all applications are allowed to use.

Note: According to the *Kernel & Features Specification (KF)*, no API is open by default to Sandboxed Applications.

This can be done either by declaring *Kernel APIs* or by defining a *Runtime Environment*.

The main difference is from the Application development point of view. In the first case, the Application project still declares standard module dependencies. This is the good starting point for quickly building a Kernel with Applications based on the MicroEJ modules as-is. In the second case, the Application project declares the runtime environment dependency. This is the preferred way in case you intend to build and maintain a dedicated Applications ecosystem.

A Kernel API or a Runtime Environment module must be added as dependency of the project:

SDK 6

SDK 5

- A Kernel API module is added as a dependency with the configuration `implementation`.

```
dependencies {
  implementation("com.microej.kernelapi:edc:1.2.0")
  implementation("ej.api:edc:1.3.4")
}
```

> **Warning:** Unlike SDK 5 (MMM), Kernel API dependencies are not transitively fetched with SDK 6. They must therefore be explicitly added.

- A Runtime Environment module is added as a dependency with the configuration `microejRuntimeEnvironment`.

```
dependencies {
  microejRuntimeEnvironment("com.mycompany:myruntime-environment:1.0.0")
}
```

- A Kernel API or a Runtime Environment module is added as a dependency with the configuration `kernelapi->default`.

```
<dependency␣
↪org="com.microej.kernelapi" name="edc" rev="1.2.0" conf="kernelapi->default"/>
```

Implement a Security Policy

A complete section about how to setup a security policy is available in the *Application security policy* page.

Add Pre-installed Applications

On a device, when a Kernel starts, it will load all the installed applications from persistent memory (ROM)[1].

To mimic this behavior on a Virtual Device, add an Application as dependency of the Kernel project:

SDK 6

SDK 5

with the configuration `microejApplication`

[1] Assuming your VEE Port implements the Feature Persistency functionality.

```
dependencies {
    microejApplication("com.mycompany:myapp:0.1.0")
}
```

with the configuration systemapp-vd->application

```
<dependency␣
↪org="com.mycompany" name="myapp" rev="0.1.0" conf="systemapp-vd->application"/>
```

The provided Application is installed in the Virtual Device only, refer to the *Application Linking* page to install Applications on the target device.

Kernel Application Configuration with SDK 5

Note: This section concerns SDK 5 only. For more information about the configuration of a Kernel Application with SDK 6, refer to the *Application* section.

Module Configuration

The build-firmware-multiapp build type defines additional configurations, used to specify the different kind of firmware inputs (see *Input and Output Artifacts*) as dependencies.

The following table lists the different configuration mapping usage where a dependency line is declared:

```
<dependency org="..." name="..." rev="..." conf="[Configuration Mapping]"/>
```

Table 121: Configurations Mapping for build-firmware-multiapp Build Type

Configuration Mapping	Dependency Kind	Usage
vdruntime->default	Add-On Library (JAR)	Embedded in the Virtual Device only, not in the Executable
default->default; vdruntime->default	Add-On Library (JAR)	Embedded in both the Executable and the Virtual Device
platform->default	VEE Port	VEE Port dependency used to build the Executable and the Virtual Device. There are other ways to select the VEE Port (see *Select a VEE Port*)
kernelapi->default	Runtime Environment (JAR)	See *Runtime Environment*
systemapp-vd->application	Application (WPK)	Included to the Virtual Device as pre-installed Application.

Example of minimal firmware dependencies.

The following example defines a Kernel that exposes all APIs of EDC library.

```
<dependencies>
   <dependency org="ej.api" name="edc" rev="1.2.0" conf="provided" />
   <!-- Runtime API (set of Kernel API files) -->
```

(continues on next page)

(continued from previous page)

```
    <dependency␣
↪org="com.microej.kernelapi" name="edc" rev="1.0.0" conf="kernelapi->default"/>
</dependencies>
```

Build Options

The *Kernel Application module nature* section describes all the options available for building a Kernel module.

Build only a Virtual Device with a pre-existing Kernel

Copy/Paste the `.kpk` file into the folder `dropins`

3.7.3 Kernel APIs

Kernel API files (`kernel.api`) specify among all types owned by the Kernel which ones **must** be used by Features, and for those types which members (method, and static fields) are allowed to be accessed by Features. When a type is not declared in a Kernel API, the Kernel and each Feature **can** have their own version of that type, but if a type is declared in a Kernel API file only the Kernel version will be used by the Kernel and all the Features.

For mode details refer to the *Class Spaces* chapter of the *Kernel & Features Specification (KF)*.

Kernel API Definition

A Kernel API file is an XML file named `kernel.api` declared at the root of one or more path composing the *Application classpath*.

Listing 43: Kernel API Example for exposing System.out.println API

```
<require>
   <type name="java.io.PrintStream"/>
   <type name="java.lang.String"/>
   <type name="java.lang.System"/>
   <field name="java.lang.System.out"/>
   <method name="java.io.PrintStream.println(java.lang.String)void"/>
</require>
```

The table below describes the format of the XML elements. The full XML schema is available in the *Kernel & Features Specification (KF)*.

Table 122: XML elements specification

Tag	Attributes	Description
require		The root element
field		Static field declaration. Declaring a field as a Kernel API automatically sets the declaring type as a Kernel API
	name	Fully qualified name on the form [type].[fieldName]
method		Method or constructor declaration. Declaring a method or a constructor as a Kernel API automatically sets the declaring type as a Kernel API
	name	Fully qualified name on the form [type].[methodName]([typeArg1,...,typeArgN)typeReturned . Types are fully qualified names or one of a base type as described by the Java language (boolean , byte , char , short , int , long , float , double) When declaring a constructor, methodName is the single type name. When declaring a void method or a constructor, typeReturned is void
type		Type declaration, allowed to be loaded from a Feature using Class.forName()
	name	Fully qualified name on the form [package].[package].[typeName]

Writing Kernel APIs

This section lists different ways to help to write kernel.api files.

Default Kernel APIs Derivation

MicroEJ Corp. provides predefined kernel API files for the most common libraries provided by a Kernel. These files are packaged as MicroEJ modules in the *Developer Repository* under the com/microej/kernelapi organisation.

The packaged file kernel.api can be extracted from the JAR file and edited in order to keep only desired types, methods and fields.

Kernel API Generator

MicroEJ Kernel API Generator is a tool that helps to generate a `kernel.api` file based on a Java classpath:

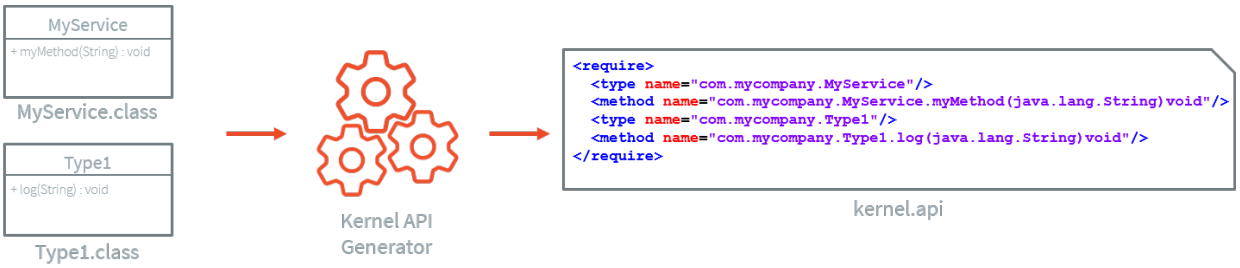


Fig. 281: Kernel API Generator Schematic

Refer to the *Kernel API Generator* documentation to run the tool.

3.7.4 Runtime Environment

Principle

A Runtime Environment is a module nature for defining the set of APIs available to an Application developer on a Kernel. It is built by aggregating a set of *Kernel APIs*.

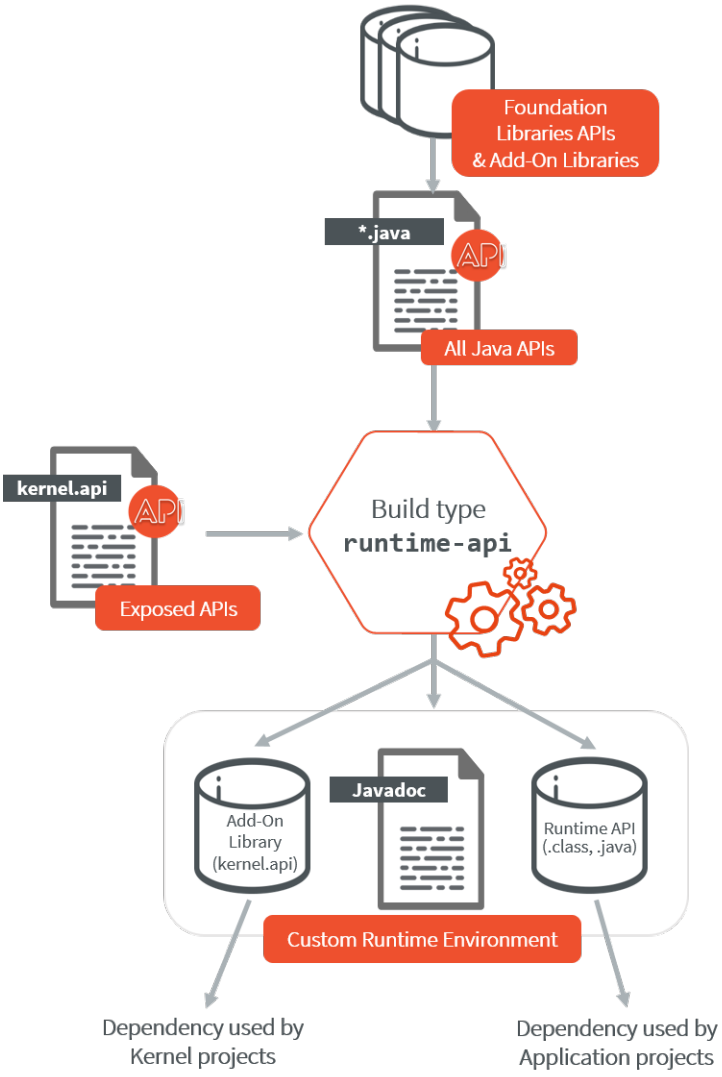
Depending on the SDK that you are using, refer to the *SDK 6 module nature* page or to the *SDK 5 module nature* page for more information about Runtime Environment module nature.

Building a Runtime Environment is one of the 2 solutions to define the APIs of a Kernel, as described in the section *Expose APIs*. Having the set of APIs named and versioned in a Runtime Environment allows to maintain, share and document it outside of a specific Kernel implementation.

Once built, a Runtime Environment module contains the following elements:

- A JAR file with the whole library of APIs (`.class` files and `.java`), used by Application projects to compile Application code;
- A JAR file with the `kernel.api` file defined in the module (if any), used by Kernel projects to fetch all the `kernel.api` files (by transitivity) to expose APIs when building the Firmware and the Virtual Device;
- A JAR file with the Javadoc of the APIs for documentation.

The following figure shows the overall build flow:



Create a new Runtime Environment Module

SDK 6

SDK 5

A Runtime Environment *Gradle project* is created with the `com.microej.gradle.runtime-environment` plugin.

```
plugins {
  id("com.microej.gradle.runtime-environment") version "1.4.0"
}
```

A Runtime Environment *module project* is created with the `runtime-api` skeleton.

```
<info organisation=
↪"com.mycompany" module="myruntimeapi" status="integration" revision="1.0.0">
  <ea:build␣
↪organisation="com.is2t.easyant.buildtypes" module="build-runtime-api" revision="4.0.+">
```

(continues on next page)

(continued from previous page)

```
  </ea:build>
</info>
```

Kernel APIs as Dependencies

The Kernel APIs can be declared as dependencies of the module. For example, the following dependencies declare a Runtime Environment that aggregates all classes, methods and fields defined by EDC, KF, BON, MicroUI and BasicTool Kernel APIs modules.

SDK 6

SDK 5

```
dependencies {
  implementation("com.microej.kernelapi:edc:1.2.0")
  api("ej.api:edc:1.3.4")
  implementation("com.microej.kernelapi:kf:2.1.0")
  api("ej.api:kf:1.5.1")
  implementation("com.microej.kernelapi:bon:1.4.0")
  api("ej.api:bon:1.4.0")
  implementation("com.microej.kernelapi:microui:3.6.0")
  api("ej.api:microui:3.6.0")
  implementation("com.microej.kernelapi:basictool:1.4.0")
  api("ej.library.runtime:basictool:1.7.0")
}
```

Warning:

- Unlike SDK 5 (MMM), Kernel API dependencies are not transitively fetched with SDK 6. Therefore, they must be explicitly added.
- The Libraries dependencies must be declared with *api* (as shown in the example above) to be consumable by the Kernel.

```
<dependencies>
  <dependency org="com.microej.kernelapi" name="edc" rev="1.0.6"/>
  <dependency org="com.microej.kernelapi" name="kf" rev="2.0.3"/>
  <dependency org="com.microej.kernelapi" name="bon" rev="1.1.1"/>
  <dependency org="com.microej.kernelapi" name="microui" rev="3.1.0"/>
  <dependency org="com.microej.kernelapi" name="basictool" rev="1.7.0"/>
</dependencies>
```

The libraries modules are fetched transitively from the Kernel APIs dependencies. For example, the dependency com.microej.kernelapi#edc;1.0.6 fetches the library ej.api#edc;1.2.3.

It is also possible to force the version of the libraries to use by declaring them as direct dependencies. This is typically used to get a latest version of the library with improvements such as Javadoc fixes or Null Analysis annotations. In this example:

```
<dependencies>
  <dependency org="com.microej.kernelapi" name="edc" rev="1.0.6"/>

  <dependency org="ej.api" name="edc" rev="1.3.4"/>
</dependencies>
```

The Runtime Environment uses the version 1.3.4 of the EDC library instead of the version 1.2.3 fetched transitively by the dependency com.microej.kernelapi#edc;1.0.6 .

Kernel APIs as Project File

The Kernel APIs can also be defined in a file in the Runtime Environment directly. The file must be named kernel.api and stored in the src/main/resources folder.

Add Add-On Processors

Note: This feature is available for SDK 5 only.

When the Runtime Environment includes an Add-On Library which uses an Add-On Processor, this Add-On Processor must be declared as a direct dependency in the Runtime Environment.

The Add-On Processor dependency line can be retrieved as follows:

- In your target *module repository*, go to the Add-On Library folder,
- Open the ivy-[version].xml file,
- Search for the dependency line with conf="addon-processor->addon-processor"

```
<ivy-module xmlns:ea="http://www.easyant.org" xmlns:ej="https://developer.microej.
↪com" xmlns:m="http://ant.apache.org/ivy/maven" version="2.0" ej:version="2.0.0">
  <info organisation="com.mycompany
↪" module="mylibrary" revision="M.m.p" status="release" publication="20220523165033">
    ...
  </info>
  <configurations>
    ...
    <conf name=
↪"addon-processor" visibility="public" description="Addon processors dependencies."/>
  </configurations>
  <publications>
    ...
  </publications>
  <dependencies>
    <dependency␣
↪org="ej.api" name="edc" rev="1.3.3" conf="default->default;provided->provided"/>
    ...
    <dependency org="com.mycompany.addon
↪" name="mylibrary-processor" rev="x.y.z" conf="addon-processor->addon-processor"/>
    ...
  </dependencies>
</ivy-module>
```

- In the Runtime Environment *module description file*, declare the addon-processor configuration in the list of configurations

```
<conf name=
↪"addon-processor" visibility="public" description="Add-On Processors dependencies."/>
```

- Paste the Add-On Processor dependency line

Warning: If the Add-On library version is changed, the Add-On Processor version must be updated.

Here is a list of known libraries using an Add-On Processor:

- NLS:

```
<dependency org="com.microej.tool.addon.runtime" name=
↪"binary-nls-processor" rev="<version>" conf="addon-processor->addon-processor"/>
```

- Wadapps:

```
<dependency org="ej.tool.addon.wadapps"␣
↪name="wadapps-processor" rev="<version>" conf="addon-processor->addon-processor"/>
```

- JavaScript:

```
<dependency org="com.microej.tool.addon.runtime
↪" name="js-processor" rev="<version>" conf="addon-processor->addon-processor"/>
```

Use a Runtime Environment in an Application

The Runtime Environment dependency must be declared in the Application project as following:

SDK 6

SDK 5

For an Application:

```
microejRuntimeEnvironment("com.mycompany:myruntime-environment:1.0.0")
```

For a Kernel Application:

```
implementation("com.mycompany:myruntime-environment:1.0.0")
```

```
<dependency org="com.
↪mycompany" name="myruntime-environment" rev="1.0.0" conf="provided->runtimeapi"/>
```

Note: If you want to add an other library dependency, make sure it is has been built on this Runtime Environment. Otherwise this could lead to inconsistent situations, for example by using an API not available at runtime. An other approach is to add it to the Runtime Environment.

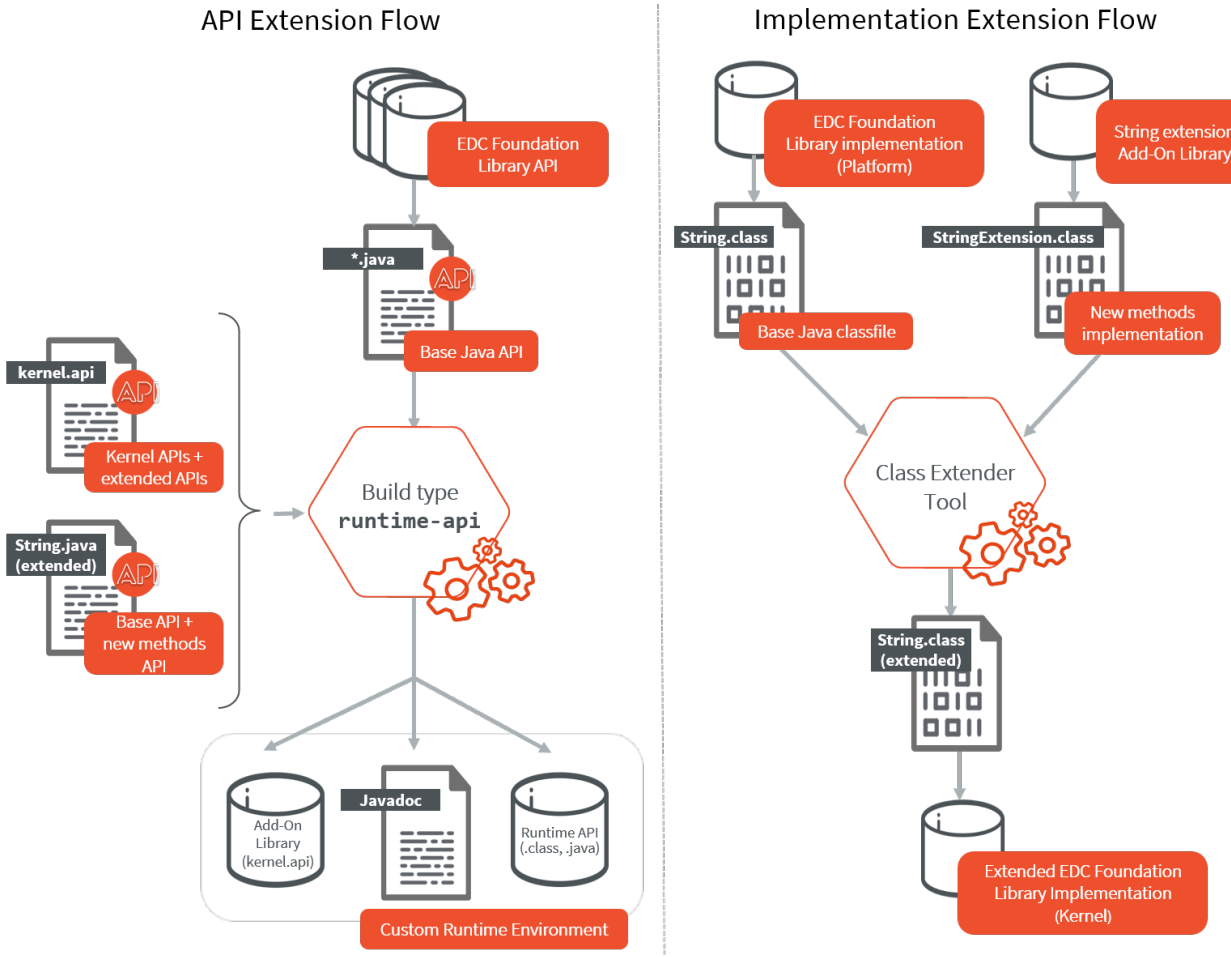
Extend a Runtime Environment

Note: This feature is available for SDK 5 only.

In a Kernel, Foundation and Add-On libraries can be extended by adding new methods to their existing classes. For example, it allows to add new methods to the class java.lang.String of the module ej.api#edc. This is done thanks to the Class Extender tool. This tool works at binary level and is able to inject methods from one class to another. Extensions can thus be independently compiled and be retrieved by the Kernel and applied during a Multi-Sandbox Executable build.

To make the extensions available to Application developers, the Runtime Environment has to be extended too.

The following diagram illustrates the process of extending the default java.lang.String class from *[EDC]* from a Kernel developer point of view:



The extension must be applied in 2 locations:

1. In the Runtime Environment. This ensures that Applications developers can see and use the new methods. The custom Runtime Environment must contain the following element:

- the API to extend, as a dependency. Here this is the EDC Foundation Library API, which contains the java.lang.String class we want to extend. We can add it transitively through its kernelapi:

```
<dependency org="com.microej.kernelapi" name="edc" rev="1.0.6"/>
```

- a *Kernel API* file definition in the `src/main/resources` folder which includes the new methods. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<require>
  <method name="java.lang.String.myNewMethod(int)java.lang.String"/>
  <method name="java.lang.String.myOtherNewMethod()void"/>
</require>
```

- the new version of the Java source of the API to extend. This class overrides the original class fetched from the dependency. Therefore it must include all the methods, the ones existing in the original class as well as the new methods, with their Javadoc specification. In our example, we must add a new `String.java` source file in the `src/main/java/java/lang` folder, and add the new methods:

```
public String myNewMethod(int number) {
  return "My number is " + number;
 }

public void myOtherNewMethod() {
  System.out.println("Hello!");
}
```

This class overrides the java.lang.String class fetched from the EDC dependency.

Once built, the custom Runtime Environment contains the new methods and can be used in the Applications projects.

2. In the Kernel. The EDC implementation is extended during the Kernel build thanks to the Class Extender tool. Refer to the Class Extender tool README and especially to the chapter `Include Class Extender During Firmware Project Build` to learn how to integrate it in a Kernel build.

   MicroEJ Corp. provides some ready-to-use extension modules:

- com.microej.library.runtime#string-regex: String methods based on Regular Expressions (e.g. `String.split()`, `String.replaceAll()`)
- com.microej.library.runtime#string-format: String formatting utility methods (e.g. `String.format()`)

3.7.5 Kernel UID

The Kernel UID is a sequence of bytes that uniquely identifies the Kernel. This UID is generated by SOAR from Java code content, Platform characteristics and a timestamp. Two Kernels built from the same Kernel Application code will not share the same UID.

The Kernel UID is used by *Core Engine* to check if an Application can be installed on a Kernel. During the *Application build*, the resulting `.fo` file embeds the Kernel UID on which it has been built.

During Kernel.install(), the UID embedded in the `.fo` is compared with the Kernel UID. By default, if both UIDs are equal the Application installation continues. Otherwise it is stopped. See also *Feature Portability Control* for `.fo` installation on different Kernels.

The Kernel UID can be retrieved at runtime using Kernel.getInstance() .getUID().

3.7.6 Sandboxed Application Lifecycle

The lifecycle of an Sandboxed Application is managed by the Kernel.

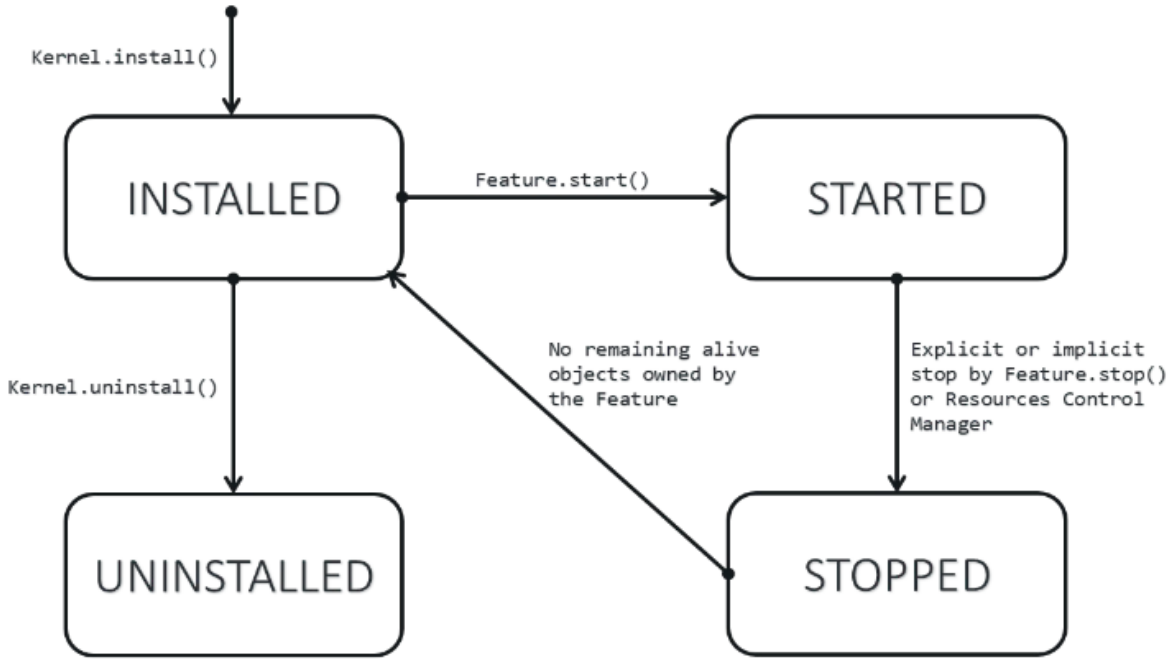


Fig. 282: Sandboxed Application Lifecycle

An Application is in one of the following states:

- **INSTALLED**: the Application has been successfully linked to the Kernel and is not running. There are no references from the Kernel to objects owned by this Application.
- **STARTED**: the Application has been started and is running.
- **STOPPED**: the Application has been stopped and all its owned threads and execution contexts are terminated. The memory and resources are not yet reclaimed.
- **UNINSTALLED**: the Application has been unlinked from the Kernel.

3.7.7 Define a Security Policy

A security policy allows the Kernel to prevent an Application from accessing resources or calling specific APIs. Whereas APIs *exposed by the Kernel* allows to control at build-time, the security policy here controls the APIs called at runtime.

Defining a security policy is done by implementing the standard SecurityManager class. Basically, all sensitive APIs exposed by the Kernel have already been protected by a *Permission Check*. Each time an Application calls such API, the Security Manager SecurityManager.checkPermission(Permission) implementation verifies that the requested permission has been granted to the Application.

Register a Security Manager

The first step for the Kernel is to register a SecurityManager implementation. Usually this is done in the Kernel boot sequence, before starting to run Applications. However, the Security Manager can be updated at any-time, and a Kernel can also register different implementations during its execution.

For the purpose of ROM footprint optimization, permission checks calls are disabled by default to avoid extra code processing if the system owner does not want to use the Security Manager. In order to activate this feature, set the *Option(checkbox): Enable SecurityManager checks* option.

Once the Security Manager checks are enabled, you can then implement your own security policy.

To apply a security policy, instantiate a SecurityManager and register it with System.setSecurityManager(SecurityManager) method.

```
// create an instance of SecurityManager
SecurityManager sm = new SecurityManager() {
  @Override
  public void checkPermission(java.security.Permission perm) {
    // implement here the Application Security Policy
  };
};
// set the Security Manager
System.setSecurityManager(sm);
```

The next section will guide you to implement the desired security policy.

Implement a Security Manager

The implementation of the SecurityManager.checkPermission(Permission) method first retrieves the owner of the requested Permission, then checks if it is a Feature (not the Kernel), and finally, checks the permission according to the given Feature.

The following code snippet shows the typical implementation:

```
public class MyKernelSecurityManager extends SecurityManager {

  @Override
  public void checkPermission(Permission perm) {
    Module caller = Kernel.getContextOwner();
    if(caller == Kernel.getInstance()) {
      // Kernel has all the rights: no checks
    }
    else{
      Feature callerApp = (Feature)caller;
      Kernel.enter();
      if(!isAllowed(callerApp, perm)) {
        throw new SecurityException();
      }
    }
  }

  private boolean isAllowed(Feature callerApp, Permission perm) {
    // implement here the Application Security Policy
  }
```

(continues on next page)

(continued from previous page)

```
}
```

You are now ready to implement your policy to decide whether the given permission is granted to the given Application.

The KF-Util module provides ready-to-use implementations, described in the next sections. Examples of integration are also available in the Kernel-GREEN project on GitHub.

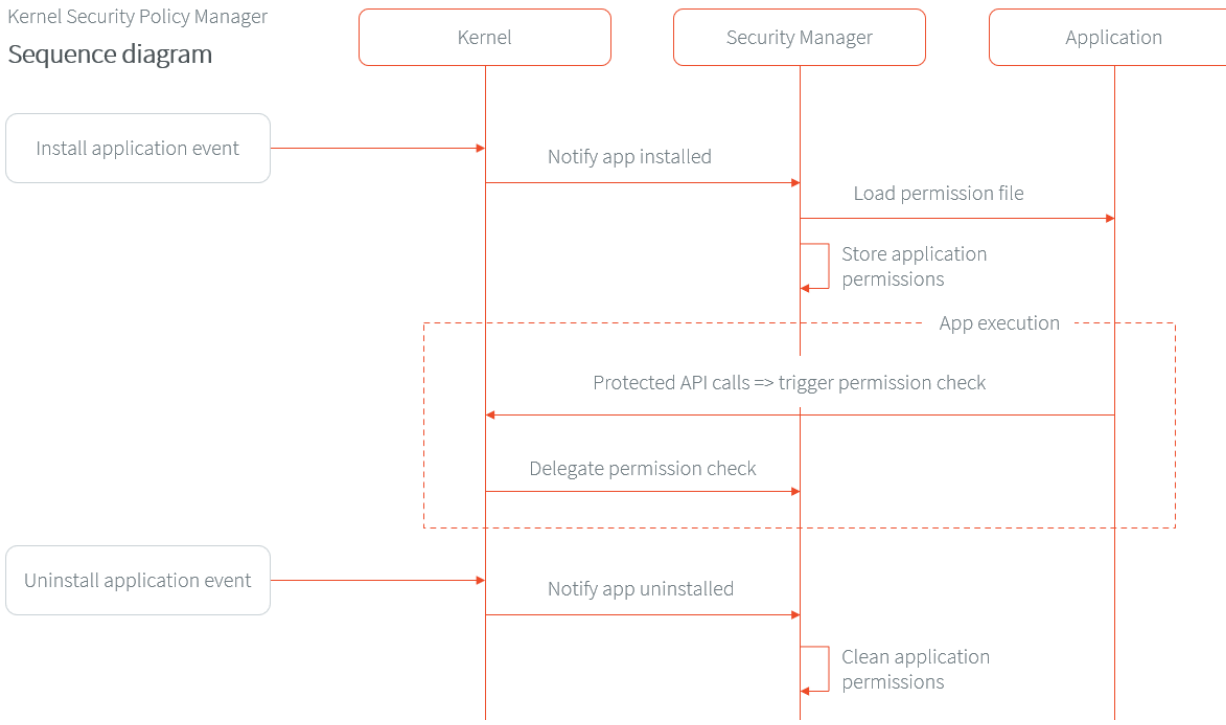
Security Manager with Application Declared Permissions

This Security Manager provides a ready-to-use implementation based on permissions declared by the Application itself. It assumes the application and its permissions file have been approved beforehand by a moderator.

Principle

Basically, Applications embed a policy resource file that describes the permissions they need at runtime. This file is then loaded when the Application is installed. Finally, this Security Manager checks if the permission has been granted to the calling Application. If a permission check is triggered but has not been declared, the Security Manager throws a SecurityException.

Here is a sequence diagram to describe the entire flow from Feature installation to uninstallation:



Policy File Format

An Application must define its Application policy file as a *resource*. By default, the resource name must be `/feature.policy.json`.

The policy file format is described in JSON, which is the default syntax supported by this Security Manager.

Before going further we strongly advise to take a look to the java.security.Permission specification and its class hierarchy to fully understand the way permissions work (name, action).

The Application policy file declares the list of required java.security.Permission classes, names and actions as following:

```
{
  "permissions": {
    "<permissionClassName1>":{
      "<permissionName1>":["<permissionAction1>","<permissionAction2>"],
      "<permissionName2>":["<permissionAction1>"]
    },
    "<permissionClassName2>":{
      "<permissionName3>":["<permissionAction3>"]
    }
  }
}
```

The permission `name` and `action` attributes are specific to the permission implementation. Therefore each permission has its own definition of what a name is.

The following keywords allow more flexibility over the content of the file:

- the `*` (wildcard) symbol means "any". It can be used for permission class name, permission name and permission actions.
- the `null` keyword represents a Java `null` value. It can be used for permission name and permission actions.

Policy File Example

Here is now an example of what a real JSON Application policy file can look like:

```
{
  "permissions": {
    "ej.microui.display.DisplayPermission":{
      "*":[]
    },
    "ej.microui.event.EventPermission":{
      "null":["null"]
    },
    "ej.microui.display.FontPermission":{},
    "ej.microui.display.ImagePermission":{
      "null":["*"]
    },"ej.microui.MicroUIPermission":{
      "*":["start"]
    },"java.net.SocketPermission":{
      "www.microej.com":["connect","resolve"]
    },"java.util.PropertyPermission":{
```

(continues on next page)

(continued from previous page)

```
      "property":["write","read"]
    },"java.lang.RuntimePermission":{
      "exit":[]
    }
  }
}
```

To simplify the file structure you can also choose to have an empty object value for permission class name or/and permission actions such as shown in the example above:

```
{
  "permissions": {
    "ej.microui.display.DisplayPermission":{
      "*":[]
    },
    "ej.microui.display.FontPermission":{},
    "java.lang.RuntimePermission":{
      "exit":[]
      }
    }
}
```

This example:

- allows the usage of any permission name and any actions for the ej.microui.display.DisplayPermission permission.
- allows the usage of any permission name and any actions for the ej.microui.display.FontPermission permission.
- allows the exit permission name and any actions for the java.lang.RuntimePermission permission.

  Using an empty value or the * wildcard is left to the developer preference and should be processed in the exact same way by the security policy resource loader.

Kernel Implementation

Here are the steps to integrate this Security Manager in your Kernel:

1. Add the dependency to the KF-Util library in the Kernel build file

   Gradle (build.gradle.kts)

   MMM (module.ivy)

```
implementation("com.microej.library.util:kf-util:2.8.0")
```

```
<dependency org="com.microej.library.util" name="kf-util" rev="2.8.0"/>
```

1. Make sure to embed java.security.Permission class names

   If the Kernel does not embed all class names (see *Stripping Class Names from an Application*), the specified Permission class names must be embedded by declaring them as *Required Types*. Any permission check done on a permission class without embedded name will result in a SecurityException.

2. Create the policy resource loader. By default, the library comes with a policy resource loader for the JSON format.

```
SecurityPolicyResourceLoader loader = new JsonSecurityPolicyLoader();
```

You can also define your own format for the policy resource file by implementing the *_SecurityPolicyResourceLoader‘* interface. Optionally, you can change the Application file policy name, by setting the *System Property* `feature.policy.name` (defaults to `/feature.policy.json`).

3. Create the KernelSecurityPolicyManager instance with the policy resource loader

```
SecurityManager sm =  new KernelSecurityPolicyManager(loader);
```

4. Register this instance as the current Security Manager

```
System.setSecurityManager(sm);
```

Note: To log every authorized access, change the logger level to `FINE` in the Kernel system properties such as `.level=FINE`.

Security Manager with Permission Dispatch

This Security Manager provides a template for dispatching the permission check per kind of java.security.Permission class. The Kernel implementation must provide instances of FeaturePermissionCheckDelegate to specify the behavior of the SecurityManager.checkPermission(Permission) for each permission class. If a permission check is done and no delegate for its permission is found, a SecurityException is thrown. The policy grants all applications the permission for a list of permission classes and logs all protected accesses by Applications.

Here are the steps to integrate this Security Manager in your Kernel:

1. Add the dependency to the KF-Util library in the Kernel build file

Gradle (build.gradle.kts)

MMM (module.ivy)

```
implementation("com.microej.library.util:kf-util:2.8.0")
```

```
<dependency org="com.microej.library.util" name="kf-util" rev="2.8.0"/>
```

1. Create the KernelSecurityManager instance

```
KernelSecurityManager sm = new KernelSecurityManager(loader);
```

2. Create a new class that implements the FeaturePermissionCheckDelegate interface like MySocketPermissionCheckDelegate below.

```
public class CustomPermissionCheckDelegate implements FeaturePermissionCheckDelegate {

    @Override
    public void checkPermission(Permission permission, Feature feature) {
        SocketPermission sPerm = (SocketPermission)permission;
```

(continues on next page)

(continued from previous page)

```
        // implement here the SocketPermission check for this Application

    }

}
```

3. Associate an instance of this FeaturePermissionCheckDelegate subclass with the java.security.Permission to be checked (like SocketPermission in the example below) by means of the Security Manager.

```
sm.setFeaturePermissionDelegate(SocketPermission.
↪class, new MySocketPermissionCheckDelegate());
```

   This code will apply the logic inside of the MySocketPermissionCheckDelegate#checkPermission(Permission permission, Feature feature) method to all mapped permissions (such as SocketPermission.class for this specific example).

4. Repeat the two previous steps for each supported java.security.Permission class.
5. Register this instance as the current Security Manager

```
System.setSecurityManager(sm);
```

Note: The Kernel-GREEN uses this Security Manager template to log all the Permission checks on the standard output.

3.7.8 Resource Manager

The Resource Manager is a software component of the KF-Util library. It allows the kernel to apply a set of *controls*, defined in a policy file, to a feature. A *control* denotes either a resource limit (CPU, RAM, flash storage, network bandwidth) or the feature's priority level.

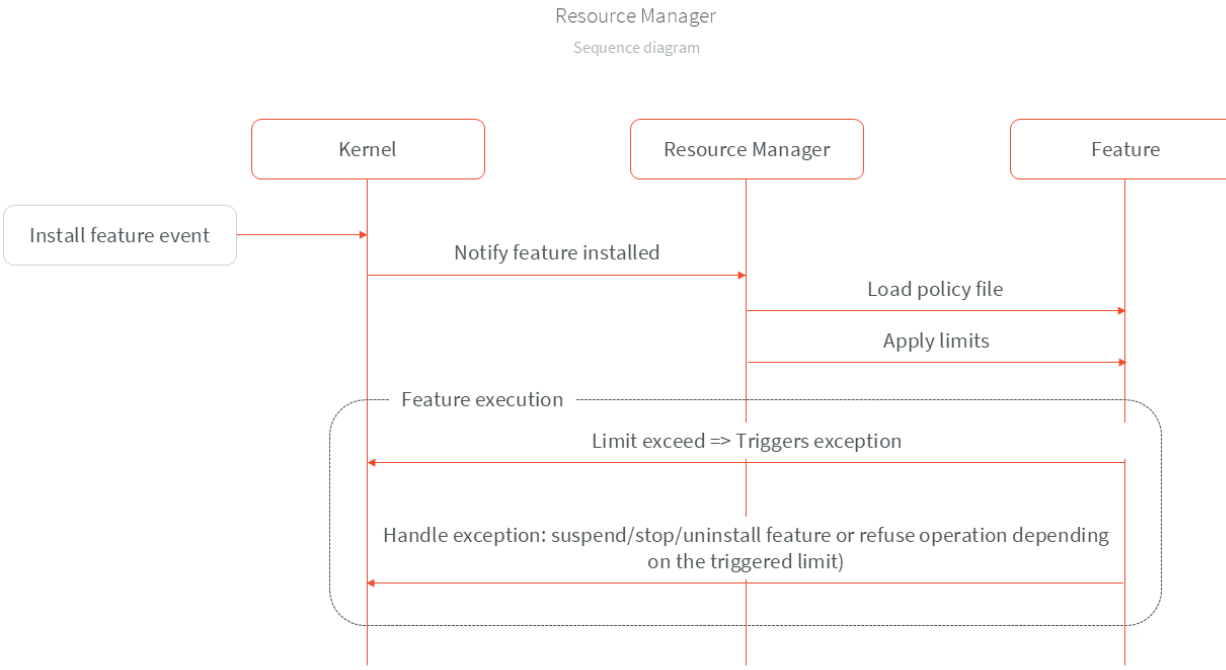
Principle

When a feature is installed, the kernel checks whether it embeds a policy file. A policy file is a description file formatted in JSON. If present, the policy file is loaded and its rules are enforced throughout the lifecycle of the feature. It is assumed that the policy file has been approved beforehand by a moderator.

The behaviour of each control is described as follows:

- **Priority**: Sets the priority of the threads owned by the feature.
- **CPU**: When a feature has reached its *CPU quota*, it is suspended until all other features consume their own quotas.
- **RAM**: If a feature exceeds its RAM limit, an OutOfMemoryError is raised and the feature is stopped and uninstalled. For more information about RAM control, please refer to this *section*.
- **Flash storage**: If the flash storage limit is exceeded, a StorageLimitException is thrown.
- **Network bandwidth**: Exceeding the daily bandwidth limit on the specified subnet triggers a BandwidthLimitException .

The following diagram illustrates a simplified flow of the Resource Manager.



Policy File Format

To define a policy, a feature must include a policy file as a *resource*. By default, the resource name is set to `/feature.policy.json`. This filename can be overridden in the kernel configuration via a *system property*, as shown below:

```
# Feature policy file
feature.policy.name=/feature.policy.json
```

The policy file is provided in JSON format. An example is shown below:

```
{
  "priority": 5,
  "resources": {
    "limit": {
      "cpu": 30,
      "ram": 6400,
      "flash": 1024
    }
  },
  "networkData": {
    "upstream": [
      {
        "dailyDataLimit": 1024,
        "cidrSubnet": "127.0.0.1/24"
      }
    ]
  }
}
```

Attributes

- **priority**: Acceptable values for the priority level range from 1 to 10.
- **cpu**: CPU usage limit expressed as a percentage (relative to a calibrated 100% baseline).
- **ram**: RAM allocation limit expressed in bytes.
- **flash**: Flash memory limit expressed in bytes. Flash storage operations are done over FS. Please refer to StorageFs for details.

- networkData:
  - **cidrSubnet**: Allowed IP range in CIDR notation.
  - **dailyDataLimit**: Daily upstream limit expressed in bytes.

Kernel Implementation

Here are the steps to integrate the Resource Manager in your kernel:

1. Add the KF-Util library dependency in the kernel build file.

   Gradle (build.gradle.kts)

   MMM (module.ivy)

```
implementation("com.microej.library.util:kf-util:3.1.0")
```

```
<dependency org="com.microej.library.util" name="kf-util" rev="3.1.0"/>
```

2. Create an instance of the Resource Manager.

```
FeaturePolicyEnforcer policyEnforcer = new FeaturePolicyEnforcer();
```

3. Apply the policy when a feature is installed in the stateChanged(Feature feature, @Nullable Feature.State previousState) method of your FeatureStateListener implementation.

```
public class CustomFeatureStateListener implements FeatureStateListener {
    @Override
    public void stateChanged(Feature feature, @Nullable State previousState) {
        switch (feature.getState()) {
        case INSTALLED:
            if (previousState == null) {
                try {
                    // The policyEnforcer instance is already created.
                    policyEnforcer.applyPolicy(feature);
                } catch (Exception exception) {
                    // Log or handle exception.
                }
            }
            break;

        // Other states.

        default:
            break;
        }
```

(continues on next page)

(continued from previous page)

```
    }
}
```

Note: Steps 2 and 3 are already implemented within the SandboxedModuleManager and ApplicationModule wrappers provided by the KF-Util library. If you're using these wrappers, no additional code is required.

3.7.9 Kernel and Features Communication

Kernel and Features can communicate with each other by sharing interface implementation instances at runtime.

In this section you will learn:

- How two (or more) Feature(s) can communicate with each other.
- How the Kernel can communicate with a Feature.

Below are defined several terms that will be used throughout this page:

- *Shared Interface* is a mechanism specific to MicroEJ Multi-Sandbox that allows exchanging object instances between Features.
  - Service represents an object instance (i.e an interface implementation)
  - Shared Services is a MicroEJ helper module that eases sharing services within a Multi-Sandbox context; it provides generic APIs that can be re-implemented as needed.
  - Registry or Service Registry represents the actual implementation of Shared Services APIs. MicroEJ provides such registries for KF but custom registries can be implemented as needed.

Shared Services

Services can be shared by means of the ej.Service library.

The Shared Services mechanism relies on a registry system that mostly consists in a Java map of class types to object instances (Map<Class<?>, Object>).

Each Feature owns a local registry in which it can register and get services within its own context; services registered in a local context cannot be retrieved by the Kernel nor any other Feature.

The Kernel also has a local registry in which it can register services that can be used within its own context but not from the context of Features.

Finally there exists a unique shared service registry contains all the registered shared services; this registry is available to all Features and to the Kernel as well.

Security policies can be implemented to restrict the usage of certain services by certain Features.

Note: The following sections relate to the existing KF implementation of the ej.Service library available in the KF-Util module ; you can however do your own custom implementation depending as needed.

Communication between Features

The KF specification does not allow Features to access object instances from other Features directly: access can only be done by means of a proxy of the target object instance.

This is made possible through the Shared Interfaces mechanism. More information about proxies can be found in the *Shared Interfaces* section.

In a nutshell Shared Interfaces and Shared Services are two complementary notions: the Shared Interfaces mechanism is responsible for setting up the capability of sharing an instance between Features whereas Shared Services offer a way to get, store and retrieve these instances once correctly set up.

Register a Service

The following line of code allows you to easily register a Service instance.

```
ServiceFactory.register(MyInterface.class,myInterface)
```

When registering a service from a Feature there are two possible options:

- The registered service is not a Shared Interface; in this case the service instance will be registered in a local service registry and only available from the Feature itself.
- The registered service is a Shared Interface; in this case the service instance will be registered in the Shared Service Registry and therefore available to any other Features that has a proxy for this instance.

For Features to use the Shared Interfaces mechanism, a Kernel must provide:

- an API for a first Feature to register its Shared Interface, and for a second Feature to get a proxy on it (by means of the ej.Service library)
- a set of registered Kernel types converters (see below)

Get a Service

The following line of code allows you to easily get a Service instance.

```
MyInterface myInterface = ServiceFactory.getService(MyInterface.class)
```

When getting a service instance from a Feature, the service instance is searched in the following order:

1. In the Local Registry, check for an instance registered by the Feature.
2. In the Shared Registry, check for an instance registered by the Feature itself.
3. In the Shared Registry, check for an instance registered (publicly) by the Kernel.
4. In the Shared Registry, check for an instance registered as a Shared Interface by an other Feature.

Communication between Kernel and Feature

The Kernel can also communicate with Features using Shared Services, by exposing object instances to Features in the shared registry.

Register a Service

From the Kernel side two distinct APIs may be used to register a Service, depending on whether the service must be registered locally or not.

You can use the generic ej.Service API that will automatically register the service instance in the **local** Kernel service registry.

```
ServiceFactory.
↪register(MyInterface.class,myInterface) //accessible within the Kernel context only
```

Or you can specify in which registry the Kernel should register the service by using the ServiceRegistryKF API from the KF-Util module as depicted below.

By doing so the service instance is exposed in the Shared Registry.

Note: To allow the usage of Kernel APIs by Features, you must make sure that the Kernel registers the necessary Kernel APIs. Learn more about Kernel API. Use of extra APIs from ServiceRegistryKF to specify the registry is reserved for the Kernel and will throw an exception if used from a Feature context.

Get a Service

The following line of code allows you to easily get a Service instance.

```
MyInterface myInterface = ServiceFactory.getService(MyInterface.class)
```

When getting a service instance from the Kernel, the service instance is searched in the following order:

1. In the Local Registry, check for an instance registered by the Kernel.
2. In the Shared Registry, check for an instance registered by the Kernel.
3. In the Shared Registry, check for an instance registered as Shared Interface by an other Feature.

   If no instance was found, an attempt is made to create a new instance of the provided type from *System Properties*.

   This property binds the service type (the property key) to the actual service implementation type (the property value) that will be used for instantiation.

   For example, in order to allow an instance of the ej.bon.Timer service to be created automatically if not present, the following property must be set:

```
ej.bon.Timer=ej.bon.Timer
```

Note: Since the service type and the implementation type are dynamically bound using class reflection, both types must be declared as *Required Types*.

Implement a Registry

In case the existing KF implementation of Shared Services does not fit your needs, you can implement your own registry system classes using the Kernel.bind() KF API.

This API allows a consumer Feature for remote use of an instance which type is owned by another Feature or the Kernel. In case the type is owned by another Feature, the returned instance is a Proxy of the shared instance. In case the type is owned by the Kernel, the returned instance is the conversion result of the shared instance to the Kernel type; for this to happen a suitable *Converter* must be registered.

As an example the steps below describe how to implement a generic Shared Interface service that relies on the Kernel.bind() API.

1. Declare the following class in your Kernel

```
package com.microej.example;

import ej.kf.Feature;
import ej.kf.Feature.State;
import ej.kf.FeatureStateListener;
import ej.kf.Kernel;
import ej.kf.Module;

/**
* Example of Kernel APIs for registering a generic Shared Interface service.
*/
public class GlobalService {

   private static Object GLOBAL_SERVICE;
   static {
      // automatically unregister the global service when the Feature is stopped.
      Kernel.addFeatureStateListener(new FeatureStateListener() {

         @Override
         public void stateChanged(Feature feature, State previousState) {
            synchronized (GlobalService.class) {
               ␣
↪              if (GLOBAL_SERVICE != null && Kernel.getOwner(GLOBAL_SERVICE) == feature
                        && previousState == State.STARTED) {
                     GLOBAL_SERVICE = null;
                  }
               }
            }
      });
   }

   /**
   * Basic API to register a Feature service. <br>
   * The service is automatically unregistered when the Feature is stopped.
   *
   * @param service
   *           the service being registered. It must implement a shared interface.
   */
   public synchronized static void registerService(Object service) {
      Kernel.enter();
      GLOBAL_SERVICE = service;
   }
```

(continues on next page)

(continued from previous page)

```
  /**
  * Basic API to retrieve a Feature service. <br>
  *
  * @param <T>
  *           the interface type
  *
  * @param serviceClass
  *           the interface of the service being retrieved. It must implement a shared interface.
  * @return the binded service or <code>null</code> if no registered service
  */
  @SuppressWarnings("unchecked")
  public synchronized static <T> T getService(Class<T> serviceClass) {
     Module contextOwner = Kernel.getContextOwner();
     Kernel.enter();
     if (GLOBAL_SERVICE == null) {
        return null;
     }
     return Kernel.bind((T) GLOBAL_SERVICE, serviceClass, (Feature) contextOwner);
  }
}
```

1. Declare the following exposed APIs in your `kernel.api` file (refer to Kernel API Definition for details)

```
<method name="com.microej.example.GlobalService.registerService(java.lang.Object)void" />
<method␣
↪name="com.microej.example.GlobalService.getService(java.lang.Class)java.lang.Object" />
```

1. Your App1 is ready to register a Shared Interface as a service

```
MySharedInterface service = new MySharedInterface();
GlobalService.registerService(service);
```

1. Your App2 is ready to retrieve a Shared Interface as a service

```
MySharedInterface service = GlobalService.getService(MySharedInterface.class))
service.use();
```

Kernel Types Converter

The Shared Interface mechanism allows to transfer an object instance of a Kernel type from one Feature to an other (see *Transferable Types* section).

To do that, the Kernel must *register a new Kernel type converter*. See the Converter class and Kernel.addConverter() method for more details.

The table below shows some converters defined in the com.microej.library.util#kf-util library.

Table 123: Example of Available Kernel Types Converters

Type	Converter Class	Conversion Rule
java.lang.Boolean	BooleanConverter	Clone by copy
java.lang.Byte	ByteConverter	Clone by copy
java.lang.Character	CharacterConverter	Clone by copy
java.lang.Short	ShortConverter	Clone by copy
java.lang.Integer	IntegerConverter	Clone by copy
java.lang.Float	FloatConverter	Clone by copy
java.lang.Long	LongConverter	Clone by copy
java.lang.Double	DoubleConverter	Clone by copy
java.lang.String	StringConverter	Clone by copy
java.io.InputStream	InputStreamConverter	Create a Proxy reference
java.util.Date	DateConverter	Clone by copy
java.util.List<T>	ListConverter	Clone by copy with recursive element conversion
java.util.Map<K,V>	MapConverter	Clone by copy with recursive keys and values conversion

Implement a Kernel Type Converter

To make a type transferable, create a converter for this type by implementing the ej.kf.Converter interface.

A converter is free to decide the kind of conversion to apply, depending on the managed type. For example, a converter for immutables instances of types such as `String` will most likely return a copy (clone), whereas a converter for instances of types such as `InputStream` will most likely return a wrapper on the original object.

The example below shows how the `String` converter is implemented:

```
import ej.kf.Converter;
import ej.kf.Feature;
import ej.kf.Kernel;

/**
* A {@link␣
↪Converter} for {@link String}. This converter creates a clone of the original {@link String}.
*/
public class StringConverter implements Converter<String> {

   @Override
   public String convert(String arg0, Feature targetOwner) throws IllegalAccessError {
      try {
            return Kernel.clone(arg0, targetOwner);
      } catch (CloneNotSupportedException e) {
            // should not occur
            throw new AssertionError();
      }
   }

   @Override
   public Class<String> getType() {
      return String.class;
   }

}
```

3.7.10 Multi-Sandbox Enabled Libraries

A Multi-Sandbox enabled library is a Foundation Library or an Add-On Library that can be embedded by a Kernel with its APIs exposed to Features.

A library requires specific code for enabling Multi-Sandbox in the following cases:

- it implements an internal global state: lazy initialization of a singleton, registry of callbacks, internal cache, …,
- it provides access to native resources that must be controlled using a Security Manager.

Otherwise, the library is called a stateless library. A stateless library is Multi-Sandbox enabled by default: it can be embedded by the Kernel, and its APIs are directly exposed to Features without code modification.

Note: This chapter generally applies to any Kernel code, not just libraries.

Manage Internal Global State

A library may define code that performs modifications of its internal state, for example:

- lazy initialization of a singleton,
- registering/un-registering a callback,
- maintaining an internal global cache, …

By default, calling one of these APIs from a Feature context will throw one of the following errors:

```
java.lang.IllegalAccessError: KF:E=S1
  at <Kernel Method>
  ...
  at <Feature Method>

java.lang.IllegalAccessError: KF:E=F1
  at <Kernel Method>
  ...
  at <Feature Method>
```

The reason is that the Core Engine rejects assigning a Feature object in a static field or an instance field owned by the Kernel. See the *KF library access error codes* for more details. This prevents unwanted object links from the Kernel to the Feature, which would lead to stale references when stopping the Feature.

The library code must be adapted to implement the desired behavior when the code is called from a Feature context. The following sections describe the most common strategies applied on a concrete example:

- declaring a static field local to the Feature,
- allowing a field assignment in Kernel mode,
- using existing Multi-Sandbox enabled data structures.

Declare a Static Field Local to the Feature

The *Kernel & Features Specification (KF)* defines Context Local Storage for static fields. This implies that the Core Engine allocates a dedicated memory slot to store the static field for each execution context (the Kernel and each Feature).

Context Local Storage for static fields is typically used when the library defines a lazy initialized singleton. A lazy initialized singleton is a singleton that is only allocated the first time it is required. This is how is implemented the well-known Math.random() method:

```
public class Math{
   private static Random RandomGenerator;

   public static double random() {
      if(RandomGenerator == null) {
         RandomGenerator = new Random();
      }
      return RandomGenerator.nextDouble();
   }
}
```

To enable this code for Multi-Sandbox, you can simply declare the static field local to the context. For that, create a `kernel.intern` file at the root of the library or Kernel classpath (e.g., in the `src/main/resources` directory) with the following content:

```
<kernel>
   <contextLocalStorage name="java.lang.Math.RandomGenerator"/>
</kernel>
```

When the method is called in a new context, the static field is read to `null`, and then a new object will be allocated and assigned to the local static field. Thus, each context will create its own instance of the `Random` singleton on demand.

Note: By default, reading a static field for the first time in a new context returns `null`. However, it is possible to write dedicated code to initialize the static field before its first read access. See section *§4.3.3 Context Local Static Field References* of the *Kernel & Features Specification (KF)* for more details.

Allow a Field Assignment in Kernel Mode

It is possible to assign a Feature object in a static field or an instance field owned by the Kernel only if the Kernel owns the current context. Such an assignment must be removed before stopping the Feature. The common way is to register a FeatureStateListener at Kernel boot. This gives a hook to remove Kernel links to Feature objects when a Feature moves to the `STOPPED` state.

```
Kernel.addFeatureStateListener(new FeatureStateListener() {

   @Override
   public synchronized void stateChanged(Feature feature, State previousState) {
      if (feature.getState() == State.STOPPED) {
         // Here, remove Kernel->Feature references
      }
   }
};
```

Without this, the Feature will remain in the STOPPED state. Therefore, it will not be possible to uninstall it or start it again until the link is removed. The remaining Feature objects referenced by the Kernel are called Kernel stale references.

Note: To help debug your Kernel, Kernel stale references are displayed by the *Core Engine dump*.

Use Existing Multi-Sandbox Enabled Data Structures

MicroEJ Corp. provides ready-to-use classes on the shelf that are Multi-Sandbox enabled. Among them, we can cite the following:

- KernelObservable : Implementation of Observable that can handle observers from any Module.
- KFList : Implementation of a Collection with multi-context support.
- SharedPropertyRegistry : Map of key/value properties.
- SharedServiceRegistry : Map of API/implementation services.

Please contact *our support team* for more details on usage.

Implement a Security Manager Permission Check

A Multi-Sandbox enabled Foundation Library should protect Feature from accessing native resources. This is done by requesting a check to the current SecurityManager defined by the Kernel.

The following code is the typical code that must be written at the beginning of API methods.

```
void myAPIThatOpensAccessToANativeResource(){

  if (Constants.getBoolean("com.microej.library.edc.securitymanager.enabled")) {
    // Here, the Security Manager support is enabled.

    SecurityManager securityManager = System.getSecurityManager();
    if (securityManager != null) {
      // Here, the Kernel has registered a Security Manager

      // Create␣
↪a Permission with relevant parameters for the Security Manager to render the permission
      MyResourcePermission p = new MyResourcePermission();

      // Request the permission check.
      // If the Kernel rejects the permission, it will throw a SecurityException
      securityManager.checkPermission(p);
    }
  }

  // Implementation code
  // ...

}
```

Note: The code is wrapped by a static check of the *Option(checkbox): Enable SecurityManager checks*. By default, this option is disabled, so the SOAR automatically removes the code. This allows you to use your library in

a Mono-Sandbox environment where ROM footprint matters. Your Kernel shall enable this option to trigger the Security Manager checks. See *Implement a Security Policy* for more details.

Known Foundation Libraries Behavior

This section details the Multi-Sandbox semantic that has been added to Foundation Libraries in order to be Multi-Sandbox enabled. Most of the Foundation Libraries provided by MicroEJ Corp. are Multi-Sandbox enabled unless the library documentation (e.g., `README.md`) mentions specific limitations.

MicroUI

Note: This chapter describes the current MicroUI version `3`, provided by UI Pack version `13.0.0` or higher. If you are using the former MicroUI version `2` (provided by MicroEJ UI Pack version up to `12.1.x`), please refer to this MicroEJ Documentation Archive.

Physical Display Ownership

The physical display is owned by only one context at a time (the Kernel or one Feature). The following cases may trigger a physical display owner switch:

- during a call to Display.requestShow(Displayable), Display.requestHide(Displayable), Display.requestRender() or Display.requestFlush(): after the successful permission check, it is assigned to the context owner.
- during a call to MicroUI.callSerially(Runnable): after the successful permission check it is assigned to owner of the `Runnable` instance.

The physical display switch performs the following actions:

- If a Displayable instance is currently shown on the `Display`, the method Displayable.onHidden() is called,
- All pending events (input events, display flushes, call serially runnable instances) are removed from the display event serializer,
- System Event Generators handlers are reset to their default EventHandler instance,
- The pending event created by calling Display.callOnFlushCompleted(Runnable) is removed and will be never added to the display event serializer.

Warning: The display switch is performed immediately when the current thread is the MicroUI thread itself (during a MicroUI event such as a MicroUI.callSerially(Runnable)). The caller looses the display and its next requests during same MicroUI event will throw a new display switch. Caller should call future display owner's code (which will ask a display switch) in a dedicated MicroUI.callSerially(Runnable) event.

The call to Display.callOnFlushCompleted(Runnable) has no effect when the display is not assigned to the context owner.

Automatically Reclaimed Resources

Instances of ResourceImage and Font are automatically reclaimed when a Feature is stopped.

BON

Kernel Timer

A Kernel Timer instance can handle TimerTask instances owned by the Kernel or any Features.

It should not be created in *clinit code*, otherwise you may have to manually declare *explicit clinit dependencies*.

Automatically Reclaimed Resources

TimerTask instances are automatically canceled when a Feature is stopped.

ECOM

The ej.ecom.DeviceManager registry allows to share devices across Features. Instances of ej.ecom.Device that are registered with a Shared Interface type are made accessible through a Proxy to all other Features that embed the same Shared Interface (or an upper one of the hierarchy).

ECOM-COMM

Instances of ej.ecom.io.CommConnection are automatically reclaimed when a Feature is stopped.

Event Queue

The *Event Queue library* is not designed to be Multi-Sandbox enabled. Therefore, a Kernel must not expose its APIs to Features.

FS

Instances of java.io.FileInputStream, java.io.FileOutputStream are automatically reclaimed when a Feature is stopped.

NET

Instances of java.net.Socket, java.net.ServerSocket, java.net.DatagramSocket are automatically reclaimed when a Feature is stopped.

SSL

Instances of javax.net.ssl.SSLSocket are automatically reclaimed when a Feature is stopped.

3.7.11 Setup a KF Testsuite

A KF testsuite can be executed when building a Foundation Library or an Add-On library, and usually extends the tests written for the *default library testsuite* to verify the behavior of this library when its APIs are exposed by a Kernel.

A KF testsuite is composed of a set of KF tests, each KF test itself is a minimal Multi-Sandbox Executable composed of a Kernel and zero or more Features.

Enable the Testsuite

In an existing library project:

SDK 6

SDK 5

- Create the `src/test/resources/projects` directory.
- Follow the instructions to *setup a testsuite on the Simulator*.
- In the build script file, replace the line:

```
microej.useMicroejTestEngine(this)
```

by:

```
microej.useMicroejTestEngine(this, TestTarget.EMB, TestMode.PROJECT)
```

- Add the `import` statements at the beginning of the file:

```
import com.microej.gradle.plugins.TestMode
import com.microej.gradle.plugins.TestTarget
```

- Add the required properties as follows:

```
val test by getting(JvmTestSuite::class) {
  microej.useMicroejTestEngine(this, TestTarget.EMB, TestMode.PROJECT)

  targets {
    all {
      testTask.configure {
        doFirst {
            systemProperties = mapOf(
              "microej.testsuite.properties.microejtool.deploy.name" to "deployToolBSPRun",
```

(continues on next page)

(continued from previous page)

```
                // Configure␣
↪the TCP/IP address and port if the VEE Port Run script does not redirect execution traces
                "microej.testsuite.properties.testsuite.trace.ip" to "localhost",
                "microej.testsuite.properties.testsuite.trace.port" to "5555",
        ␣
↪           // Tell the testsuite engine that the VEE Port Run script redirects execution traces.
        ␣
↪           // Uncomment this line and comment the 2 lines above if the VEE Port supports it.
                //"microej.testsuite.properties.launch.test.trace.file" to "true"
            )
        }
    }
  }
}
```

- Create the `src/test/projects` directory,
- Edit the `module.ivy` and insert the following line within the `<ea:build>` XML element:

```
<ea:plugin␣
↪organisation="com.is2t.easyant.plugins" module="microej-kf-testsuite" revision="+" />
```

- Configure the option `artifacts.resolver` to the name of the resolver used to import KF test dependencies. The name must be one of the resolver names defined in your *settings file*. If you are using the default settings file, set the option to `MicroEJChainResolver`. This option is usually set as a global *MMM option*.

Add a KF Test

A KF test is a structured directory placed in the `src/test/resources/projects` directory in SDK 6, or in the `src/test/projects` directory in SDK 5. The creation of a KF is done as follows:

SDK 6

SDK 5

- In the `src/test/resources/projects` directory, create a new directory `newTest` for the KF test.
- In the `src/test/java` folder, create a new Interface with the following content:

```
import org.junit.Test;

public interface MyTest {

  @Test
  void newTest();
}
```

Note: The name of a KF test is free. For each KF test, a method with the same name and annotated with `@Test` must be defined in the Java Interface.

- Within the `newTest` directory, create the sub-projects:
  - Create a new `kernel` directory and *create a Kernel*. The Kernel must depend on your Library project:

```
implementation("org.example:myLibrary")
```

Note: The Library project is used as an included build when running one of its KF tests. Therefore, it is not mandatory to specify the dependency version here.

- Create a new `app` directory and *create an Application*.
- Create a `settings.gradle.kts` file and add the following content:

```
include("kernel", "app")
```

Each KF test must contain a Kernel project named `kernel`. If the KF Test also contains one or more Feature projects, their names must be prefixed by `app`.

The KF Testsuite structure shall be similar to the following figure:

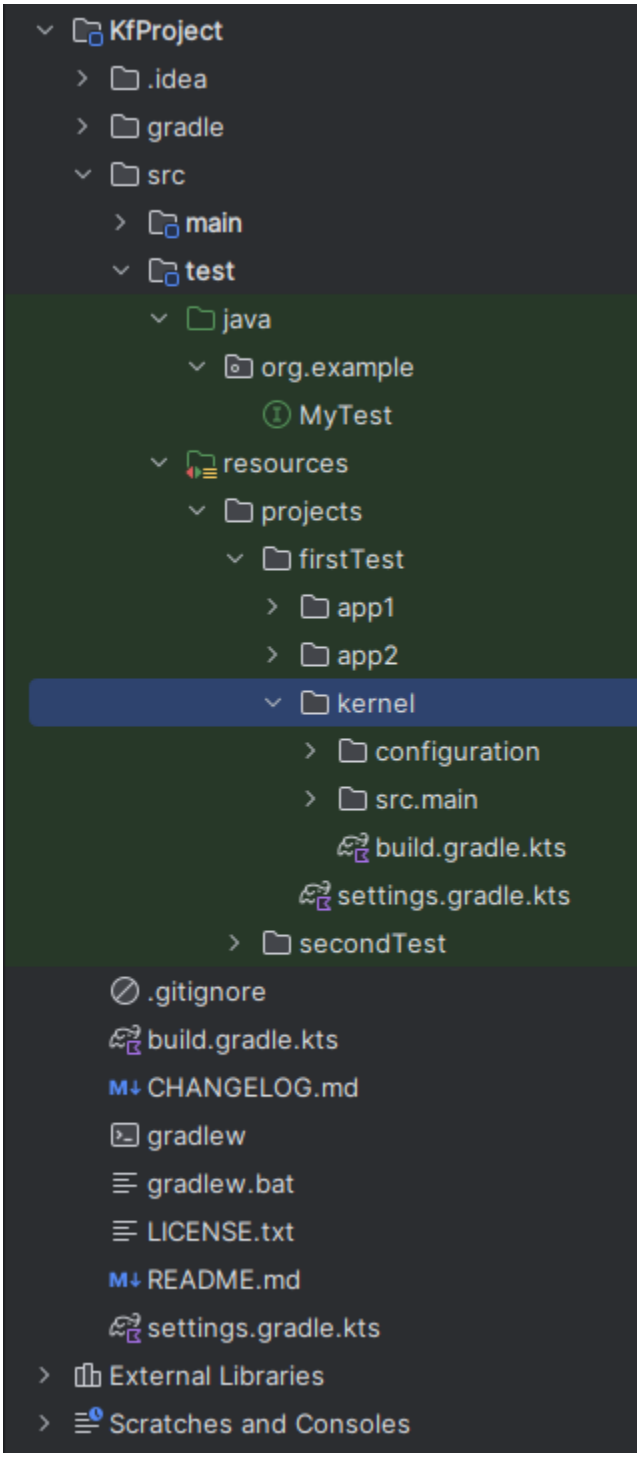


Fig. 283: KF Testsuite Overall Structure

- Create a new directory for the KF test
- Within this directory, create the sub-projects:
  - Create a new directory for the Kernel project and initialize it using the microej-javalib *skeleton*,
  - Create a new directory for the Feature project and initialize it using the application *skeleton*,

  - Create a new directory for the Firmware project and initialize it using the firmware-multiapp *skeleton*.

    The names of the project directories are free, however MicroEJ suggests the following naming convention, assuming the KF test directory is [TestName] :

- [TestName]-kernel for the Kernel project,
- [TestName]-app[1..N] for Feature projects,
- [TestName]-firmware for the Firmware project.

The KF Testsuite structure shall be similar to the following figure:

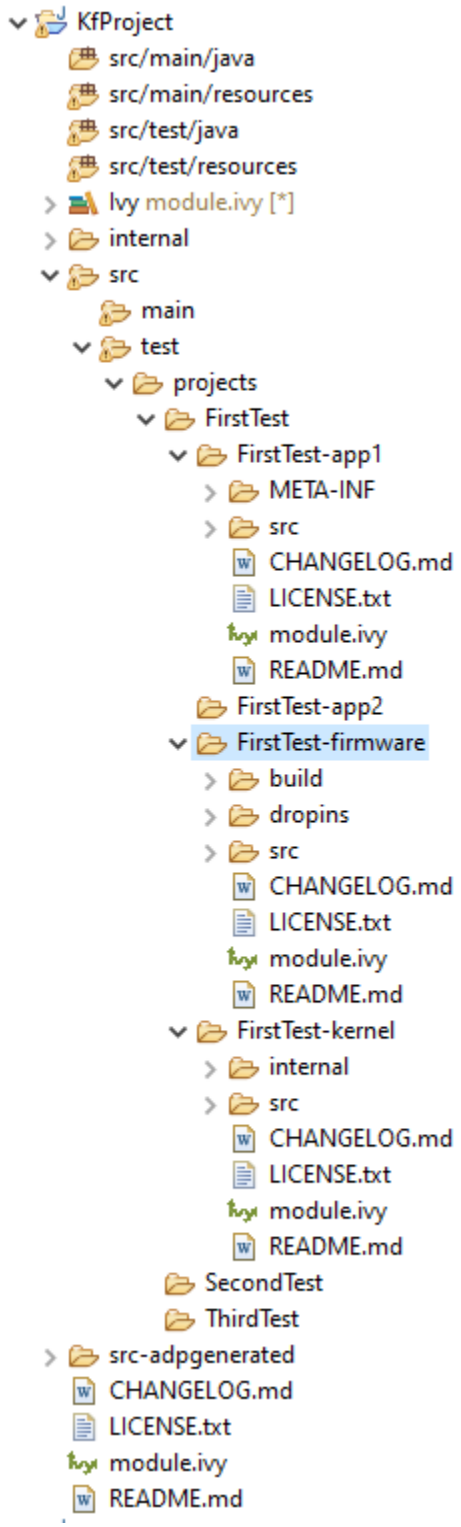


Fig. 284: KF Testsuite Overall Structure

All the projects will be built automatically in the right order based on their dependencies.

KF Testsuite Options (SDK 5 only)

It is possible to configure the same options defined by *Testsuite Options* for the KF testsuite, by using the prefix `microej.kf.testsuite.properties` instead of `microej.testsuite.properties`.

3.7.12 Kernel Linking

This chapter describes how a Kernel Application is linked.

Basically, a Kernel Application is linked as a Standalone Application. The main difference is that a Kernel Application defines *Kernel APIs*, and requires to embed additional information that will be used later to build a Sandboxed Application against this Kernel (by linking with the Kernel APIs). Such additional information is called the Kernel Metadata.

Link Flow

The following figure shows the general process of linking an Executable, applied to a Kernel Application.

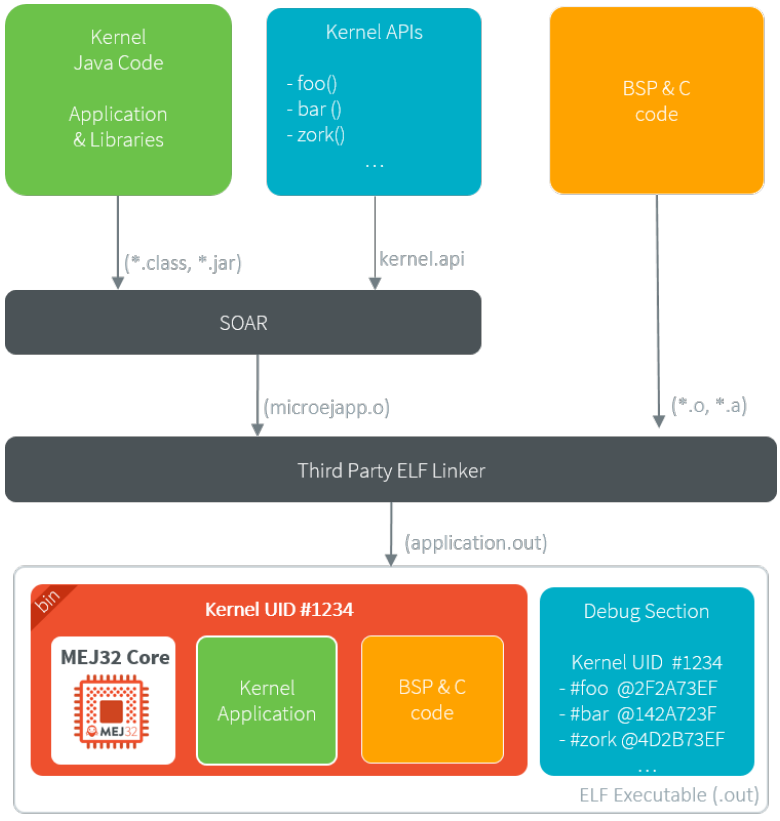


Fig. 285: Kernel Link Flow

The Platform must be configured with *Multi-Sandbox capability*.

By default, the Kernel Metadata is included in the `.debug.soar` section which also serves for debug purpose (*Stack Trace Reader*, *Heap Dumper*). Particularly, it contains resolved absolute addresses of Kernel APIs.

Kernel Metadata Generation

To *build a Sandboxed Application on Device*, the Kernel Metadata must be exported after the Firmware link from the .debug.soar section of the executable. This step is not necessary to *build a Sandboxed Application Off Board*.

The Kernel Metadata can be exported from an existing Firmware executable file by using the Kernel Metadata Generator tool. It produces a .kdat file that will be used to link the Sandboxed Applications on device.

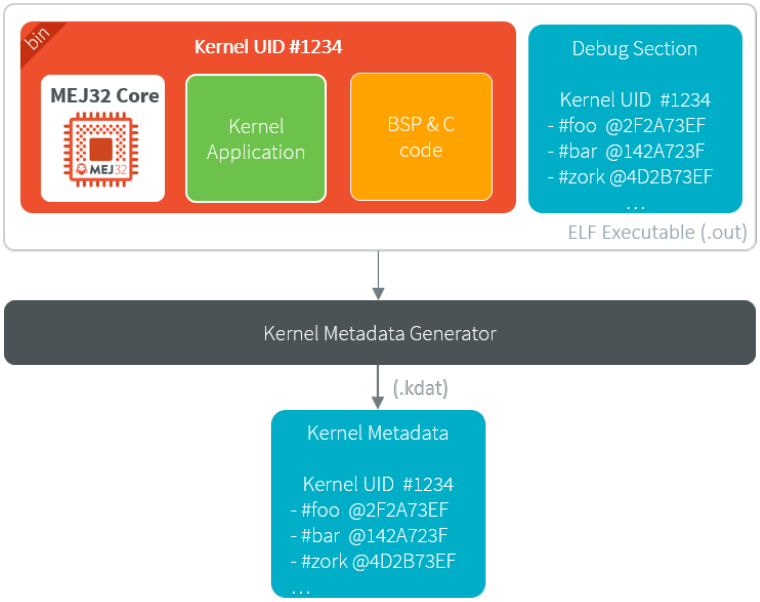


Fig. 286: Kernel Metadata Generator

The .kdat file is optimized for size. When linking a Sandboxed Application .fso file, only the required metadata will be loaded in Managed Heap. It will be loaded from a standard InputStream, so that it can be stored to a memory that is not accessible from the CPU's address space.

Note: The Kernel Metadata .kdat file can also be integrated in a Firmware executable file using post-link tools such as binutils objcopy, provided a dedicated section has been reserved by the third-party linker.

Feature Portability Control

A Kernel can install .fo files that have been built on other Kernels, provided this Kernel complies with other Kernels according to a set of rules declared hereafter. This is called *Feature Portability Control*, as the verification is performed during the new Kernel build, with no impact on the Feature dynamic installation.

Principle

During a Kernel build, SOAR can verify this Kernel preserves the portability of any `.fo` files built on a previous Kernel using *the Kernel metadata file*. If the portability is preserved, the *UID* of the previous Kernel is embedded in the new Kernel, allowing `.fo` files built on the previous Kernel to be installed as well. Otherwise, SOAR fails with an error indicating the broken rule(s).

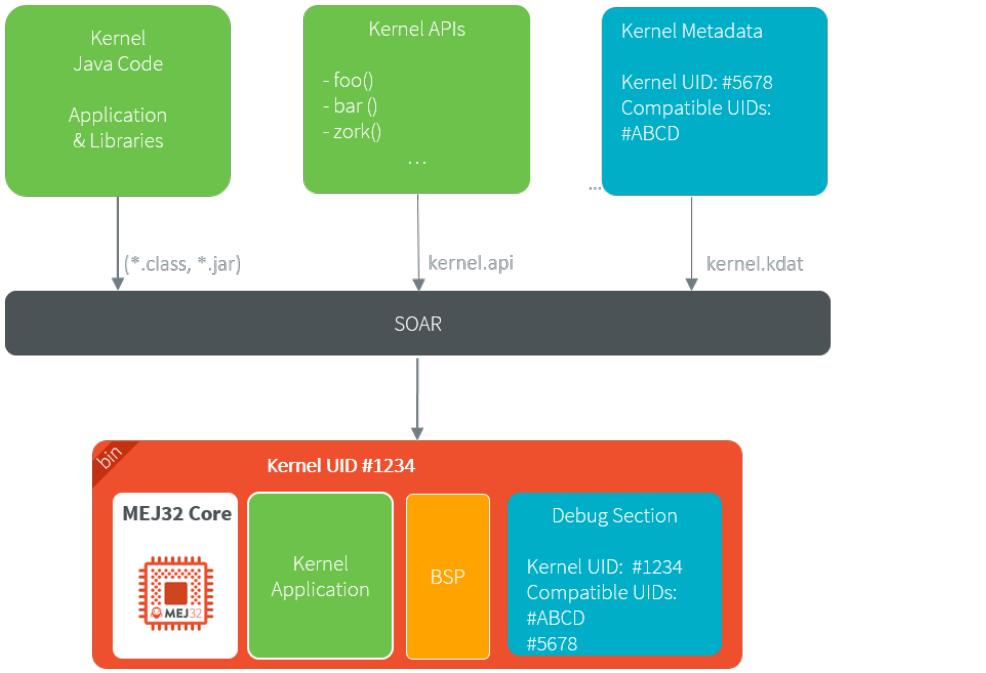


Fig. 287: Feature Portability Control Principle

Enable

Note: This is a new functionality that requires Architecture *8.0.0* or higher.

Add the following *Application Options* to your Kernel project:

- `com.microej.soar.kernel.featureportabilitycontrol.enabled` : `true` to enable Feature Portability Control. Any other value disables Feature Portability Control (the following option is ignored).
- `com.microej.soar.kernel.featureportabilitycontrol.metadata.path` : Path to the previous Kernel Metadata file (`.kdat` file) for which retrocompatibility should be maintained.

Portability Rules

A Kernel Application can install a .fo file that has been built against another Kernel Application if the Kernel Application code has not changed or if the modifications respect the portability rules. Here is the list of the modifications that can be done while preserving the portability:

- Modify method code, except if *Method Devirtualization* or *Method Inlining* has changed.
- Add a new type (including declared as Kernel API),
- Add a new static method (including declared as Kernel API),
- Add a new instance method in a type **not declared** as Kernel API,
- Add a new instance method with private visibility in a type **declared** as Kernel API,
- Add a new static field (including declared as Kernel API),
- Add a new instance field in a type **not declared** as Kernel API,
- Rename an instance field with private visibility in a type **declared** as Kernel API,
- Modify a Java type, method, or static field **not declared** as Kernel API (code, signature, hierarchy)
- Remove a Java type, method, or static field **not declared** as Kernel API

Both Kernel Applications must be built from Platforms based on the same Architecture version.

Any other modifications will break the Feature portability. For example, the following modifications will not preserve the portability:

- Remove a Java type, method or static field **declared** as Kernel API,
- Add or remove an instance method in a type **declared** as Kernel API, even if the method is **not declared** as Kernel API,
- Add or remove an instance field in a type **declared** as Kernel API,
- Modify method or field signature **declared** as Kernel API (name, declaring type, static vs instance member, …),
- Modify hierarchy of a type **declared** as Kernel API.

3.7.13 Application Linking

This chapter describes how a Sandboxed Application is built so that it can be (dynamically) installed on a Kernel. The build output file of a Sandboxed Application against a Kernel is called a Feature, hence the f letter used in the extension name of the related files (.fso and .fo files).

SOAR Build Phases

When building a Sandboxed Application to a Feature, SOAR processing is divided in two phases:

1. **SOAR Resolver**: loads the set of application .class files and resources. Among the various steps, mention may be made of:

- Computing the transitive closure from the application entry points of all required elements (types, methods, fields, strings, immutables, resources, system properties),
- Computing the *clinit order*.

The result is an object file that ends with .fso extension. The .fso file is a portable file that can be linked on any compatible Kernel (see *FSO Compatibility*).

2. **SOAR Optimizer**: links a .fso file against a specific Kernel. Among the various steps, mention may be made of:

- Linking to the expected Kernel APIs (types, methods, fields) according to the JVM specification[1],
- Generating the MEJ32 instructions,
- Building the virtualization tables.

The result is an object file that ends with the .fo extension. By default, the .fo file is specific to a Kernel: it can only be installed on the Kernel it has been linked to. Rebuilding a Kernel implies to run this phase again, unless the application has been built for the previous Kernel (see *Feature Portability*).

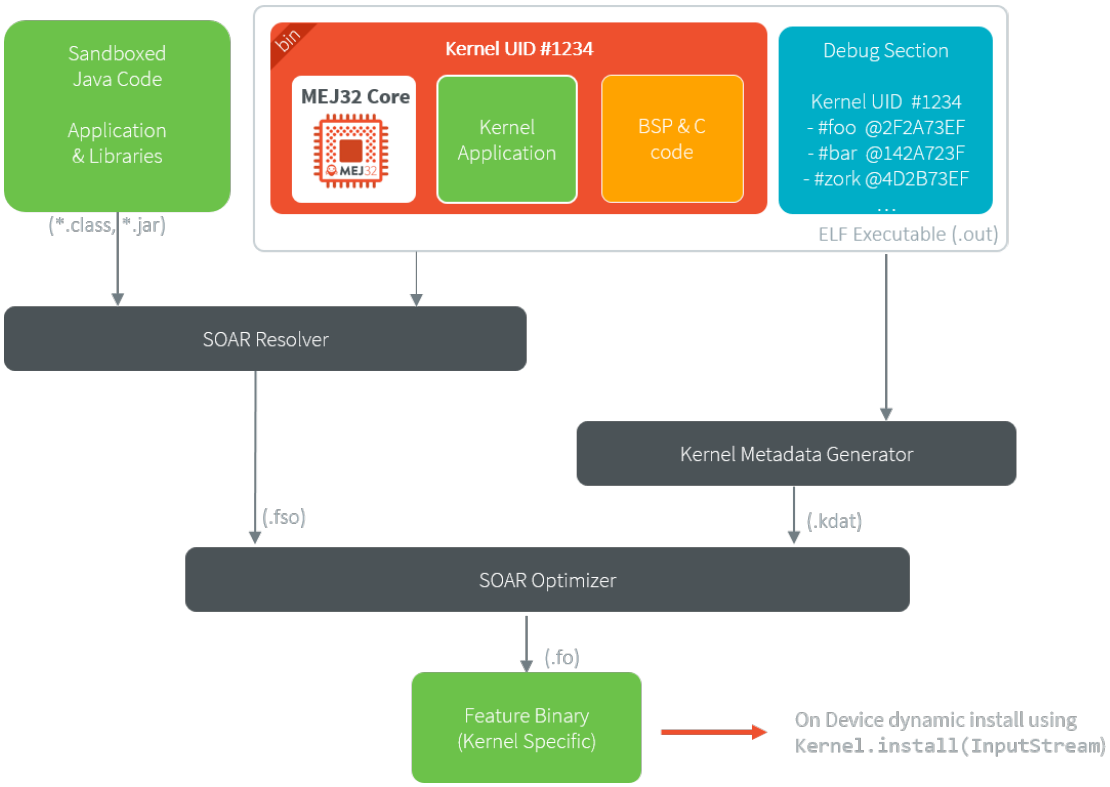


Fig. 288: Sandboxed Application Build Flow

The Feature .fo file can be deployed to the Device using Kernel.install() method.

[1] Tim Lindholm & Frank Yellin, The Java Virtual Machine Specification, Second Edition, 1999

Feature Build Off Board

A Sandboxed Application can be built to a Feature (.fo file) as follows:

SDK 6

SDK 5

- Refer to the *Build a Feature file* page to build the Feature.

Use a *MicroEJ Application Launch* configured as follows:

- Set the Settings field in the Execution tab to Build Dynamic Feature .
- Set the Kernel field in the Configuration tab to a Multi-Sandboxed Firmware (.out ELF executable file).

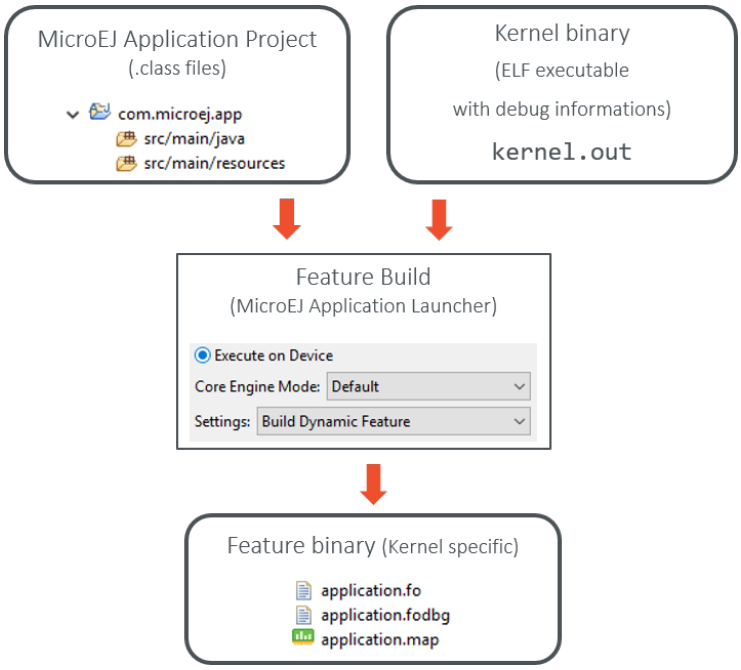


Fig. 289: Feature Build Flow using MicroEJ Launch

Feature Build On Device

The SOAR Optimizer is packaged to a Foundation Library named SOAR, thus this phase can be executed directly on Device.

General Workflow

Here are the typical steps to achieve:

- Build the Sandboxed Application on any compatible Kernel to get the .fso file,
- Transfer the .fso file on Device by any mean,
- Generate the *Kernel Metadata* for the Kernel on which the .fso file is being linked,
- Transfer the .kdat file on Device by any mean,
- Write a MicroEJ Standalone Application for building the .fso file:
  - implement a com.microej.soar.KernelMetadataProvider to provide an InputStream to load the .kdat file,
  - provide an InputStream to load the .fso file,
  - provide an OutputStream to store the .fo file,
  - call FeatureOptimizer.build() method.

  Then the .fo file can be dynamically installed using Kernel.install().

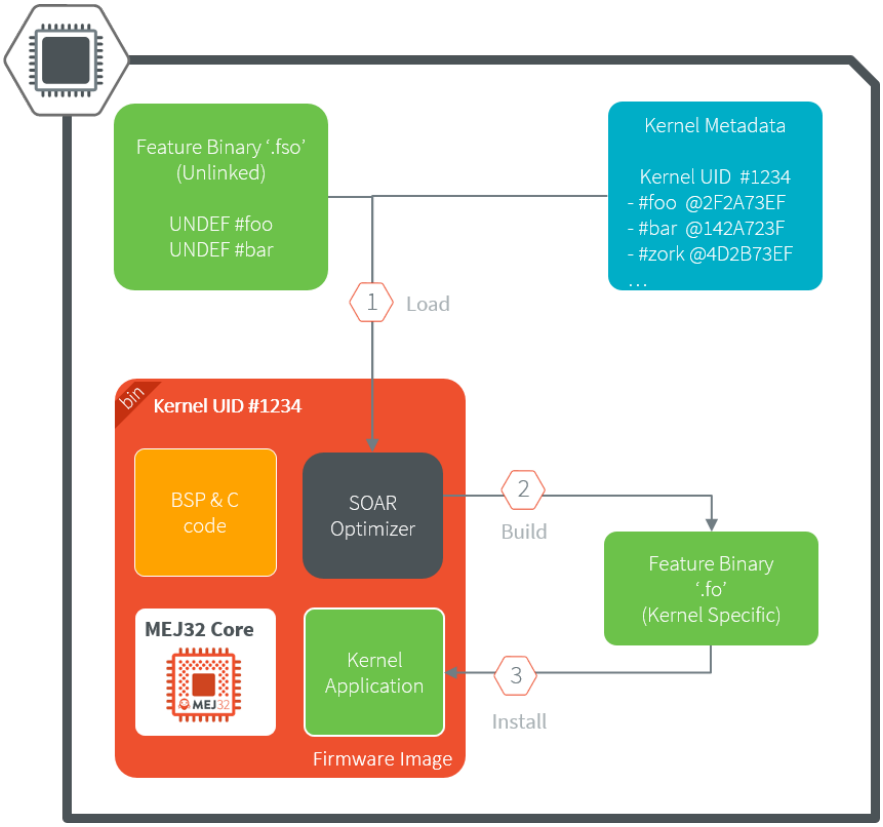


Fig. 290: Sandboxed Application Build on Device

Note: Although this is common, it is not required to run the SOAR Optimizer phase on the Kernel that will dynamically install the .fo . There is no relationship between SOAR and KF Foundation Libraries.

Implement the Kernel

SOAR Optimizer can be integrated on any Standalone Application providing the following dependencies:

SDK 6

SDK 5

```
implementation("ej.api:edc:1.3.3")
implementation("com.microej.api:soar:1.0.0")
implementation("ej.library.eclasspath:collections:1.4.0")
```

```
<dependency org="ej.api" name="edc" rev="1.3.3" />
<dependency org="com.microej.api" name="soar" rev="1.0.0" />
<dependency org="ej.library.eclasspath" name="collections" rev="1.4.0" />
```

The following code template illustrates the usage of the SOAR Foundation Library:

```
package com.microej.example;

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

import com.microej.soar.FeatureOptimizer;
import com.microej.soar.FeatureOptimizerException;
import com.microej.soar.KernelMetadataProvider;

/**
* This is a template code that shows the typical steps to follow for building a
* .fo file from a .fso file on Device.
*/
public class TemplateFSOBuild {

   /**
   * Your Platform specific {@link KernelMetadataProvider} implementation.
   */
   private static final class MyKernelMetadataProvider implements KernelMetadataProvider {

      @Override
      public InputStream openInputStream(int offset) throws IOException {
         // Return␣
↪an InputStream to the Kernel Metadata resource (.kdat file) at the given offset in bytes.
         return null; // TODO
      }

      @Override
      public String toString() {
         // Here,
↪ return a printable representation of this Kernel Metadata Provider (for debug purpose only)
```

(continues on next page)

(continued from previous page)

```
        return "Kernel Metadata loaded from ..."; // TODO
    }
  }

  /**
  * A method that builds a .fso file to a .fo file.
  */
  public static void build() {
    // Create the KernelMetadataProvider instance
    KernelMetadataProvider kernelMetadataProvider = new MyKernelMetadataProvider();

    // Load the .fso InputStream
    InputStream fsoInputStream = null; // TODO

    // Prepare the target OutputStream where to store the .fo
    OutputStream foOutputStream = null; // TODO

    // Create the FeatureOptimizer instance
    FeatureOptimizer featureOptimizer;
    try {
      featureOptimizer = new FeatureOptimizer(kernelMetadataProvider);
    } catch (FeatureOptimizerException e) {
      // Handle Kernel Metadata cannot be loaded
      e.printStackTrace(); // TODO
      return;
    }

    // Build
    try {
      featureOptimizer.build(fsoInputStream, foOutputStream);
    } catch (FeatureOptimizerException e) {
      // Handle .fso cannot be built to .fo
      e.printStackTrace(); // TODO
    }
  }
}
```

FSO Compatibility

A .fso file can be linked on any Kernel providing all the following conditions:

- its Architecture has the same endianness than the Architecture on which the .fso file has been produced,
- its Architecture version is compatible[2] with the Architecture version on which the .fso file has been produced,
- it provides the required APIs according to the JVM specificationPage 1504, 1.

A current limitation is that if the Sandboxed Application declares an immutable object, SOAR Optimizer will resolve fields within the same class rather than considering the entire class hierarchy.

[2] New version is greater than or equals to the old one within the same major version.

Feature Portability

By default, a .fo file can only be installed on the Kernel on which it has been linked.

Starting from *Architecture 8.0.0*, the same Feature file can be installed on different Kernels. This is called *Feature Portability*. Thus it is not required to rebuild the .fo file in the following cases:

- Relinking the executable (memory layout changes),
- Recompiling the C code,
- Rebuilding the Kernel Application, if *Feature Portability Control* has been enabled.

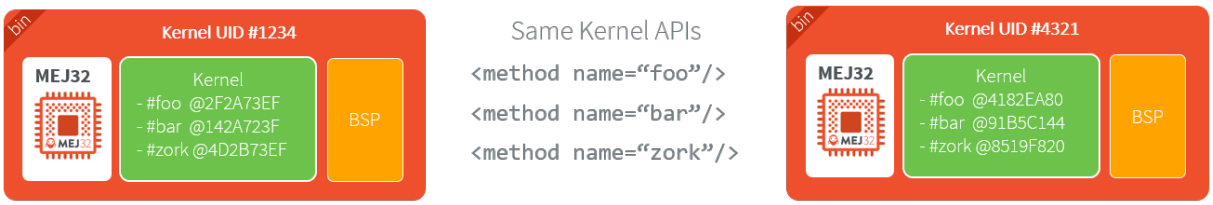


Fig. 291: Feature Portability Overview

3.8 VEE Wear User Guide

VEE Wear is a comprehensive wearable software solution. It aggregates multiple libraries, applications and tools dedicated to the development of wearable software.

VEE Wear Framework

The *VEE Wear Framework* provides a turnkey, customizable wearable firmware.

This framework contains the following components:

- a Multi-Sandboxed wearable Kernel
- common wearable apps (Settings, *Facer*, Health, Stopwatch, etc.)
- a wearable API to develop custom wearable apps

Low Power Facer Engine

VEE Wear supports the Facer Engine, expanding Facer's extensive watch face catalog of 500,000 faces across all smartwatches, including low-power RTOS watches, enriching the user experience for all users. For further information about the MicroEJ and Facer partnership, please contact your MicroEJ sales representative.

Android Compatibility Kit

The *Android Compatibility Kit* comprises two technologies:

1. Support for Android development tools, including Android Studio and Gradle, is provided by MICROEJ SDK 6.
2. Support for MicroEJ applications on the Android OS (Android Runtime).

   The support provided by Android development tools and Android runtime is especially beneficial when running the same application on both an Android processor and an MCU. This scenario occurs, for example, when implementing Android off-loading with a big-little architecture.

   Moreover, the Android runtime enables the execution of the same application on an Android smartphone. For instance, in scenarios such as building a watch face or app picker within a companion smartphone app, the MicroEJ application code can be directly utilized to display the app or watch face. This eliminates the necessity to develop a similar version specifically for Android.

iOS Compatibility Kit

The *iOS Compatibility Kit* feature provides support for MicroEJ applications on iOS (iOS Runtime). This runtime enables the execution of the same application on an iOS smartphone. For example, when creating a watch face or app picker within a companion smartphone app, the MicroEJ application code can be directly used to display the app or watch face. This eliminates the necessity to develop a separate version for iOS.

MicroEJ's offloading framework

VEE Wear incorporates an *offloading framework* designed to optimize power consumption in a big-little architecture, where an application processor runs Android, and a companion MCU operates MICROEJ VEE. By alternating the execution of applications between low-consumption MCUs and powerful MPUs, this setup guarantees maximum power efficiency, thereby conserving battery life. The offloading framework encompasses an inter-processor communication framework and efficient low-power profile management.

3.8.1 VEE Wear Framework

The VEE Wear Framework allows developers to build a VEE Wear Kernel executable and develop VEE Wear Apps.

The Framework contains the following components:

- the source code of the VEE Wear Kernel
- the source code of common VEE Wear Apps (health, settings, etc.)
- the VEE Wear Services library

Note: The source code is available on demand. You can contact *MicroEJ Support* to evaluate VEE Wear.

VEE Wear Kernel

The VEE Wear Kernel is the core Application running on the wearable device. It manages the lifecycle of VEE Wear Apps.

The Kernel is board agnostic and can be fully customized to any hardware. It must be built from its source code (*available on demand*) with the VEE Port corresponding to the target hardware. The build generates an executable file and a Virtual Device which must be provided to app developers so that they can build their Apps.

The VEE Wear Kernel Application requires the following amount of memory:

- RAM: around 400KB (mainly for Managed Heap and buffered images)
- ROM: from 250KB to 1MB for each App (mainly for its images)

The Kernel provides the following APIs to the Apps:

Library	Version
EDC	1.3
BON	1.4
Trace	1.1
FS	2.1
MicroUI	3.1
Drawing	1.0
MicroVG	1.2
Audio	1.0
Bluetooth	2.2
VEE Wear Services	1.1

VEE Wear Services Library

The VEE Wear Services library allows VEE Wear Apps to communicate with the Kernel and with other Apps.

The library provides interfaces to services that are implemented by the Kernel:

Service	Provided Features
ComponentService	register App components
DeviceService	get device information
ExternalResourceService	create and delete external resources
FontService	get fonts
HealthService	get health information
NavigationService	navigate across the UI
TimeService	get time information

The KernelServiceProvider class provides static methods to get an instance of every service. It also provides a shared instance of Timer, which can be used by any App to schedule tasks without creating an additional thread.

The library also provides interfaces to components that are implemented by the Apps:

Component	Provided Features
Activity	shows a UI allowing to interact with a specific feature of the device
Watchface	shows the current time and brief data with unique visuals
Renderable	renders the UI of an Activity or Watchface
ComplicationDataSource	provides data that can be displayed on watchface complications

Using ComponentService, Apps can register their own components into the Kernel. Once a component is registered, its lifecycle is managed by the Kernel: the Kernel will call the methods of the component when the user navigates in the relevant menus of the watch.

To use the VEE Wear Services, add the following line to the project build file:

```
implementation("com.microej.library.wear:wear-services:1.1.0")
```

VEE Wear Apps

Apps are the cornerstone of VEE Wear: they provide exciting and unique features that make a smartwatch unforgettable. With VEE Wear, Apps can be deployed over USB or Bluetooth and installed dynamically on the device. Partial software update allows manufacturers to provide new features and bug fixes without having to validate and deploy an entire firmware.

Creating an App

To create an App project, follow these steps:

- *Create an SDK 6 Application project*.
  - Open the `build.gradle.kts` file.
  - Add a dependency to the VEE Wear Services library: add `implementation("com.microej.library.wear:wear-services:1.1.0")` in the `dependencies` block.

Implementing the Entry Point

The entry point of an App contains the code that will be called by the Kernel when the App is installed and uninstalled. The name of the entry point class must be *declared* in the `build.gradle.kts` file. The class should implement the FeatureEntryPoint interface.

The entry point can perform any operation, such as registering components, adding Bluetooth services or running timer tasks. However, the entry point is not the place to show a UI: this should be done by the Activities or Watchfaces registered by the App.

Note: Apps can use any library, as long as it doesn't require a Foundation Library that is not provided by the Kernel.

Implementing an Activity

Activities show a UI that allows the user to interact with a specific feature of the device. They are listed in the Activity Launcher:

Activity Launcher

Activity

To implement an Activity, create a class that implements the Activity interface and its methods:

- getName() should return the name of the Activity. This is the name that is visible in the Activity Launcher.
- renderIcon() should render the icon of the Activity in the given region. This is the icon that is visible in the Activity Launcher.
- createRenderable() should create the Renderable that renders the Activity (see *Implementing a Renderable*).

Once the Activity is implemented, make sure it is registered in the Kernel by calling ComponentService.registerActivity(). This method is usually called in the entry point of the App.

Implementing a Watchface

Watchfaces show the current time and brief data with unique visuals. They are listed in the Watchface Picker.

Watchface Picker

Watchface

To implement a Watchface, create a class that implements the Watchface interface and its methods:

- renderPreview() should render a preview of the UI in the given region. This is the preview that is visible in the Watchface Picker.
- createRenderable() should create the Renderable that renders the Watchface (see *Implementing a Renderable*). The renderable can use the ComponentService.getComplicationDataSources() API to render complications that show data from third-party Apps.

Once the Wathface is implemented, make sure it is registered in the Kernel by calling ComponentService.registerWatchface(). This method is usually called in the entry point of the App.

Implementing a Renderable

A Renderable is a user interface which can be rendered on a graphics context during a transition and which can be shown in fullscreen on the display.

To implement a Renderable, create a class that implements the Renderable interface and its methods:

- onAttached() should load the resources necessary for rendering the UI.
- onDetached() should dispose the resources loaded in onAttached().
- render() should render the UI on the given graphics context.

- showOnDisplay() should show on the display a Displayable that will render the UI in fullscreen and handle events.

The wear-util library provides 2 generic implementations of Renderable:

- RenderableDesktop : Renderable based on a Desktop. Most often used for Activities that display rich UIs with widgets and scrolls.
- RenderableDisplayable : Renderable based on a Displayable. Can be used in most cases, for simple UIs or for performance/memory efficiency.

Implementing a Complication Data Source

A Complication Data Source provides data which can be displayed on a watchface complication. Complication Data Sources can provide a text, an icon and a progress value. Watchfaces can render complications the way they want using the data provided by the source.

Watchface with complications

To implement a Complication Data Source, create a class that implements the ComplicationDataSource interface and its methods:

- hasText(), hasIcon() and hasProgress() should return whether the source provides the associated information.
- getText() and getProgress() should return the associated information.
- renderIcon() should render the icon in the given region.

Once the Complication Data Source is implemented, make sure it is registered in the Kernel by calling ComponentService.registerComplicationDataSource(). This method is usually called in the entry point of the App.

Building an App

Selecting the Kernel

To be able to build your App, you must add a dependency to the *VEE Wear Kernel* by following these steps:

- Open the `build.gradle.kts` file.
- Define the path to the VEE Wear Framework on your machine: add `val veeWearFramework = "/path/to/VEE-Wear-Framework"`
- Add a dependency to the VEE Wear Kernel: add `microejVee(files("$veeWearFramework/Virtual-Device", "$veeWearFramework/kernel.out"))` in the `dependencies` block.

Building the App

Once the project is configured, the App can be built like any MicroEJ Application:

- To test an App on simulator, launch the `runOnSimulator` Gradle task.
- To build an App, launch the `buildFeature` Gradle task. You can then install the App by deploying the `build/feature/application/application.fo` Feature file to the watch over USB or Bluetooth.

3.8.2 Android Compatibility Kit

MicroEJ provides a set of tools and libraries to run applications powered by MicroEJ on Android and Wear OS. This allows for the same application to be developed, simulated, tested, and executed on MicroEJ VEE and Android alike.

Having the same code ready for both Android and MicroEJ VEE opens up a wide range of use cases, including but not limited to:

- Develop derivative products based on small MCUs or low-cost MPUs with limited resources where Android cannot be used as it is inherently resource-intensive.
- Reduce energy consumption by enabling two processors to coexist to distribute tasks between a very powerful processor powered by Android and a low-power processor powered by MicroEJ.

Below are some examples from the wearable segment that illustrate these use cases:

- Watch faces can be developed once and deployed on both a smartwatch (MicroEJ VEE) and its companion smartphone app (Android), enabling consistent functionality and appearance across both devices. This provides a good user experience for the user while minimizing code duplication and maintenance for the developer.
- Power efficiency is a significant concern with wearable as sophisticated features often come at a high cost in terms of power consumption. An *offloading* framework can reduce power usage by executing the same application alternately on a low-consumption MCU and a powerful MPU. Operating in standby mode as much as possible on the MCU is one of the strategies to achieve energy efficiency.

Overview

The Android Compatibility Kit is composed of two main components:

- A runtime: applications developed on MicroEJ can run on the Android platform thanks to the Android-based implementation of the MicroEJ Foundation libraries and dedicated support libraries.
- A developer kit: the MICROEJ SDK 6 and a Gradle plugin provide the necessary support for developing applications in Android Studio using Gradle.

Workflow

Below is a general overview of the workflow when developing a product that targets both MicroEJ and Android-powered devices.

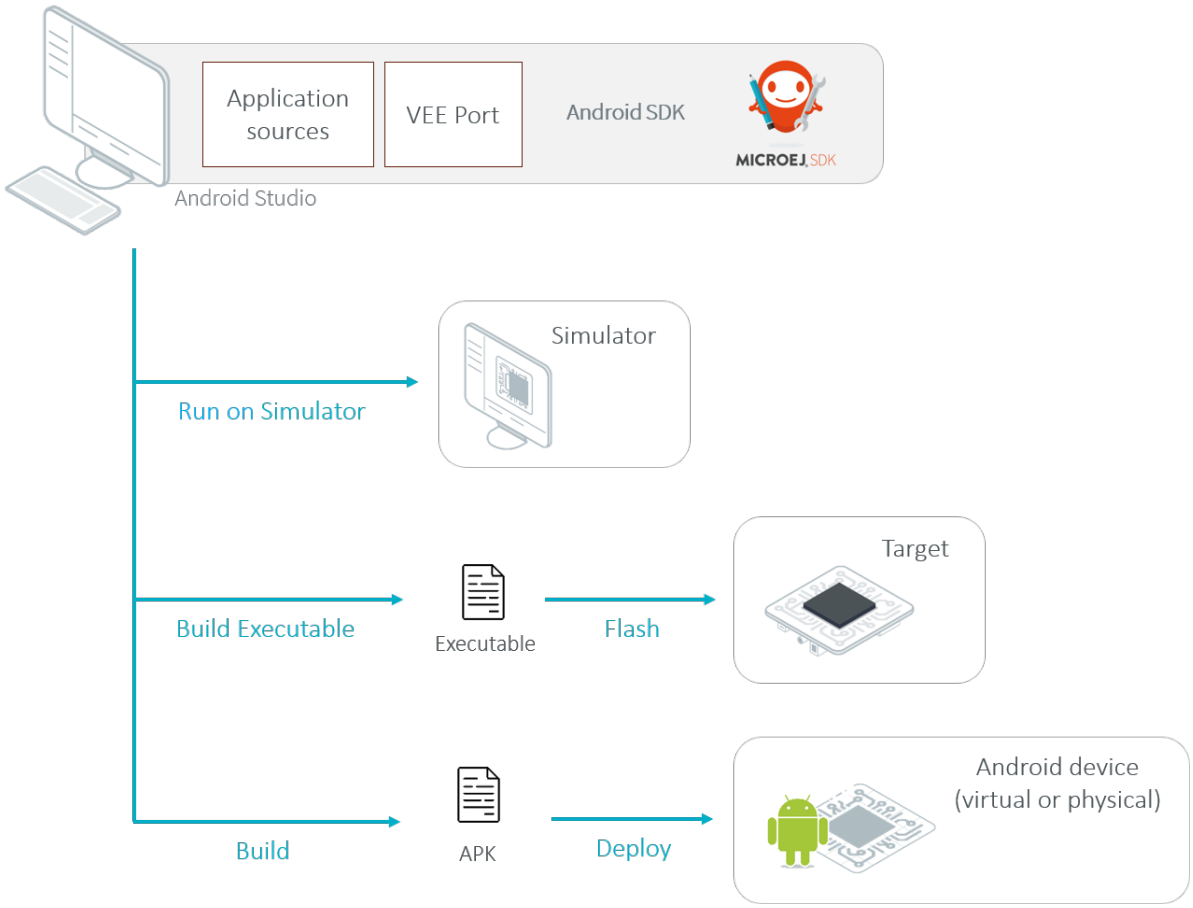


Fig. 292: Workflow Diagram

Software Architecture

Applications designed to run in MicroEJ VEE can also run on Android, thanks to a specific implementation of MicroEJ Foundation libraries based on Android libraries.

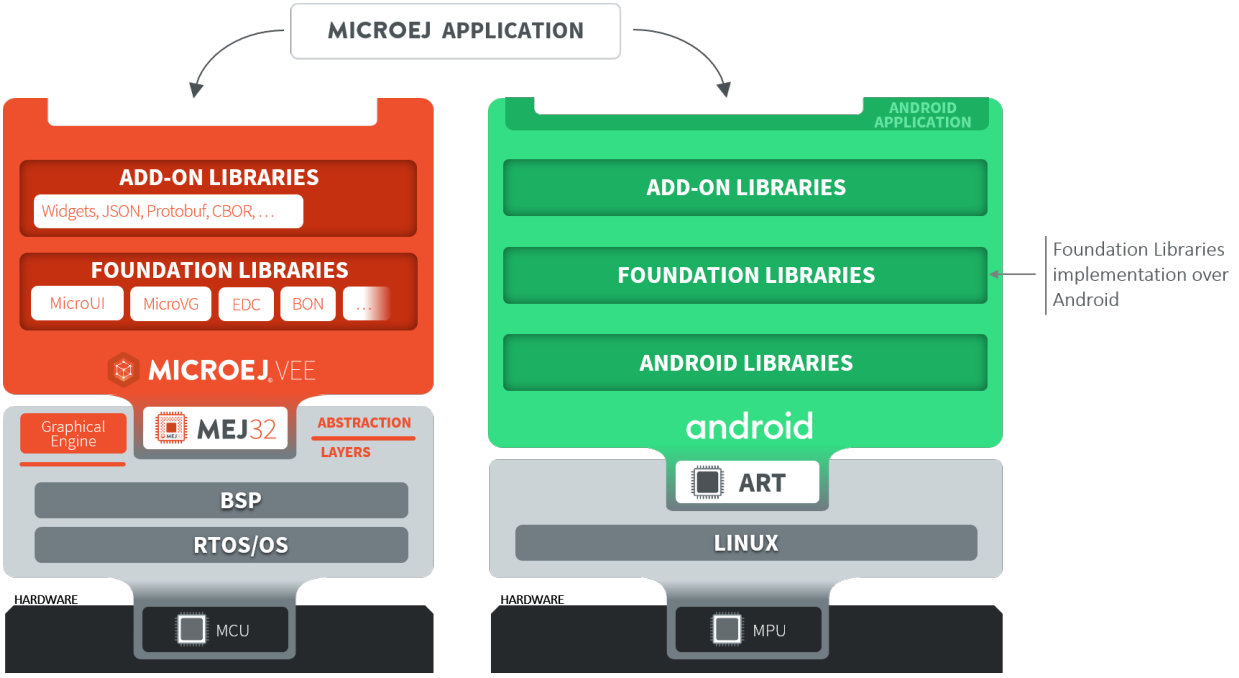


Fig. 293: Software Architecture

Available APIs and Features

- Foundation Libraries
  - EDC
  - BON
  - SNI
  - MICROUI
  - MICROVG
  - TRACE
- All compatible Add-on Libraries
- Supported Resources
  - Images
  - Fonts (FNT, EJF, TTF, OTF)
  - Android Vector Drawables (AVD)
  - SVG
  - NLS (including Android XML and PO formats)

  - Constants
  - Properties
- Custom native APIs can be implemented over Android libraries to make them executable on Android devices.

Installation

To develop applications compatible with both Android and MicroEJ VEE, it is required to use *MICROEJ SDK 6*. The MICROEJ SDK provides Gradle plugins that allow for seamless integration in Android Studio.

For comprehensive installation instructions, read the *SDK 6 Installation Guide*. Follow the steps related to Android Studio when relevant.

In addition to installing MICROEJ SDK 6, make sure to comply with the requirements listed below.

JDK Version

The Android Compatibility Kit, like MICROEJ SDK 6, is compatible with a JDK 11 or higher LTS version. The JDK version to use will depend on the Android Gradle Plugin (AGP) being used by your Android project. Starting from version 8, AGP requires a minimum JDK 17. If you intend to use JDK 11, you will need to specify a compatible version of AGP (e.g., 7.4.2). Refer to the Android Gradle plugin release notes for more information.

Configure Repositories

The *SDK 6 repositories configuration* references the MicroEJ module repositories which are required for resolving the SDK Gradle plugins and modules. Working with Android plugins and modules involves extending this configuration to include additional repositories that are essential for Android development.

- download and copy this file in <USER_HOME>/.gradle/init.d/ .

Project Setup

This chapter will guide you through the process of creating a project for having an application compatible with both Android and MicroEJ VEE.

The recommended project structure to get started is to have a basic multi-project build that contains a root project and two subprojects: one subproject for the Android/Wear OS application and one subproject for the MicroEJ Application. The MicroEJ Application defines code that will run on both MicroEJ VEE and Android, while the Android application includes wrapper code and logic specific to Android.

What follows is the directory and file structure of a typical project:

```
android-app/
   src
   microej.properties      # MicroEJ Application Options for Android/Wear OS
   build.gradle.kts
microej-app/
```

(continues on next page)

(continued from previous page)

```
    src
    configuration/
|       common.properties   # MicroEJ Application Options for MicroEJ VEE
    build.gradle.kts
  build.gradle.kts
  settings.gradle.kts
```

Create or Import an Android project

The Android documentation covers the process of creating apps for diverse form factors, including smartphones and wearable devices. Read Create a Project and follow the guidelines before proceeding. If you are creating a project from scratch, we recommend using the Empty Activity template.

Note: The project template in Android Studio defines a default repositories configuration in the settings.gradle.kts file of the project like below:

```
pluginManagement {
  repositories {
    google()
    mavenCentral()
    gradlePluginPortal()
  }
}
dependencyResolutionManagement {
  repositoriesMode.set(RepositoriesMode.FAIL_ON_PROJECT_REPOS)
  repositories {
    google()
    mavenCentral()
  }
}
```

Please note that this will override the repositories configuration defined during the *installation process*, based on Gradle initialization scripts. Set the configuration according to your preference, but we suggest removing these lines at first from the settings file to get started.

Assuming that a Gradle project with an Android application is now opened in Android Studio, do the following:

- Open the build.gradle.kts file at the root of the project.
- Add these lines to the plugins block:

```
id("com.microej.gradle.application") version "0.17.0" apply false
id("com.microej.android.gradle.plugins.android") version "0.3.6" apply false
```

Create or Import a MicroEJ Application

The next step is adding the module that contains the MicroEJ Application to the Gradle project.

Create a MicroEJ Application

Import an existing MicroEJ Application

- Click on File > New > New Module… .
- Select Java or Kotlin Library .
- Set the name of the module in the Library Name field.
- Set the package name of the module in the Package name field.
- Enter a name for the main Java class of the application in the Class name field.
- Select the language Java in the Language field.
- Select Kotlin DSL in the Build configuration language field.
- Click on Finish .

The module created by Android Studio is a standard Java module (Gradle java-library plugin). The build.gradle.kts file has to be updated to make it a MicroEJ Application module:

- Open the build.gradle.kts file.
- Erase its whole content.
- Add the com.microej.gradle.application plugin in the build.gradle.kts file:

```
plugins {
  id("com.microej.gradle.application")
}
```

- Add the following microej block in the build.gradle.kts file:

```
microej {
  applicationEntryPoint = "com.mycompany.Main"
}
```

where the property applicationEntryPoint is set to the Full Qualified Name of the main class of the application. This class must define a main() method and is the entry point of the application.

- Declare the dependencies required by your application in the dependencies block of the build.gradle.kts file. The EDC library is always required in the build path of an Application project, as it defines the minimal runtime environment for embedded devices:

```
dependencies {
   implementation("ej.api:edc:1.3.5")
}
```

If you have already developed a MicroEJ Application, you can import it in the project.

Note: If the MicroEJ Application has been created with the *SDK 5* or lower, it is required to first migrate it to *SDK 6*. Read the comprehensive *Migration Guide* before proceeding.

- Click on File > New > Import Module… .
- Browse to the source directory of the Gradle project.
- Set the module name.
- Click on Finish .

Note: Android Studio may use the Groovy DSL to include the imported module. The result is the creation of a setting.gradle file that shadows the configuration in the settings.gradle.kts file. If that occurs, merge the relevant content of the setting.gradle file into the existing settings.gradle.kts and remove the setting.gradle .

- Set the microejConflictResolutionRulesEnabled property to false in the build.gradle.kts file:

```
microej {
  microejConflictResolutionRulesEnabled = false
  ...
}
```

Note: The MicroEJ Gradle plugin comes with additional conflict resolution rules compared to Gradle's default behavior. This can make the build fail when working with Android dependencies, so it is recommended to use Gradle's default conflict management in this case. These extra rules can be disabled by setting the microejConflictResolutionRulesEnabled property to false in the microej configuration block. Read *Manage Resolution Conflicts* for more details.

- Ensure that the Gradle settings file includes the Android and MicroEJ modules, like in this example:

```
include(":android-app")
include(":microej-app")
```

- To synchronize your project files, select Sync Now from the notification bar that appears after making changes.

  When the Gradle project has been reloaded, it should compile successfully, without any error.

Configure the Android Application

The next steps show how to configure the Android or Wear OS application to declare the MicroEJ Application.

- Open the build.gradle.kts file of the Android application.
- Add the com.microej.android.gradle.plugins.android plugin:

```
plugins {
   id("com.android.application")
   id("com.microej.android.gradle.plugins.android")
   ...
}
```

- Add a dependency to the MicroEJ support library depending on the target (Android or Wear OS).

  Android

  Wear OS

  ```
  dependencies {
     implementation("com.microej.android.support:microej-application:2.0.1")
     ...
  }
  ```

  The support library `microej-application` allows running a MicroEJ Application in an Android Activity using the MicroEJ support engine.

  ```
  dependencies {
     implementation("com.microej.android.support:microej-wearos:2.0.1")
     implementation("androidx.wear.watchface:watchface:1.1.1")
     implementation("androidx.wear.watchface:watchface-guava:1.1.1")
     ...
  }
  ```

  The support library `microej-wearos` allows running a MicroEJ Application in a Wear OS WatchFaceService using the MicroEJ support engine.

- Add a dependency to the MicroEJ Application using the `microejApp` configuration, for example:

```
dependencies {
   microejApp(project(":microej-app"))
   ...
}
```

  where `microej-app` is the name of the subproject that contains the MicroEJ Application.

- Add a dependency to a VEE Port, for example:

```
dependencies {
   microejVee("com.mycompany:veeport:1.0.0")
   ...
}
```

  There are multiple options for providing a VEE Port in your project. Read *Select a VEE Port* to explore the available options.

  Note: It is required to select a VEE Port that's configured to build MicroEJ Applications for Android. Read the *VEE Port section* to learn how to configure a VEE Port for this purpose.

- Add a file named `microej.properties` at the root of the Android application. This file sets the MicroEJ Application Options when running on Android. It is similar in principle to *defining Application Options* for the embedded device. Depending on the target device (Android or embedded device), the content may differ.

- Select Sync Now from the notification bar to synchronize your project files.

Start the MicroEJ Application

The final step involves calling the entry point of the MicroEJ Application from within the Android or Wear OS application.

Android

Wear OS

Assuming that the Android application declares an activity in the AndroidManifest.xml :

- Open the corresponding activity Java/Kotlin file.
- Make MicroEJActivity the superclass of this class.
- Override the method getApplicationMainClass() and make it return the Full Qualified Name of the main class of the MicroEJ Application.

  This is an example of a simple activity:

  Kotlin

  Java

```
class MainActivity : MicroEJActivity() {
  override fun getApplicationMainClass(): String {
    return "com.mycompany.Main";
  }
}
```

```
public class MainActivity extends MicroEJActivity {
  @Override
  protected String getApplicationMainClass() {
    return "com.mycompany.Main";
  }
}
```

When the activity is created, it instantiates the main class of the MicroEJ Application and invokes its main() method.

Assuming that the Wear OS application declares a watch face service in the AndroidManifest.xml :

- Open the corresponding watch face service Java/Kotlin file.
- Make MicroEJWatchFaceService the superclass of this class.
- Override the method getApplicationMainClass() and make it return the Full Qualified Name of the main class of the MicroEJ Application.

  This is an example of a simple activity:

  Kotlin

  Java

```
class MyWatchFaceService : MicroEJWatchFaceService() {
  override fun getApplicationMainClass(): String {
    return "com.mycompany.Main";
```

(continues on next page)

(continued from previous page)

```
  }
}
```

```
public class MyWatchFaceService extends MicroEJWatchFaceService {
  @Override
  protected String getApplicationMainClass() {
    return "com.mycompany.Main";
  }
}
```

When the watch face service is created, it instantiates the main class of the MicroEJ Application and invokes its `main()` method.

Select Sync Now from the notification bar to synchronize your project files.

Run on MicroEJ VEE and Android

The application can now be deployed to both MicroEJ VEE and Android environments.

The deployment of an application designed to use the Android Compatibility Kit has nothing specific compared to other MicroEJ or Android applications. This means that you can refer to the dedicated documentation for this matter:

- for MicroEJ VEE: refer to sections *Run On Simulator*, *Build Executable* and *Run On Device*.
- for Android: refer to the official Android documentation.
- for Wear OS: refer to the official Wear OS documentation.

VEE Port

This section explains how to configure a VEE Port so that it provides the capability to build a MicroEJ Application for Android.

Once it is configured, the VEE Port can thus be used to build a MicroEJ Application for Android, in addition to standard features such as building a MicroEJ Application for an Embedded Device and running it on the Simulator.

VEE Port Configuration

The configuration steps ensure that the VEE Port provides build scripts and implementations of Foundation Libraries which are specific to Android.

These files are gathered in Android Packs. Each Android Pack provides support for one or multiple Foundation Libraries. The Core Android Pack is absolutely necessary to be able to build any MicroEJ Application for Android. Additional Android Packs should be included depending on the Foundation Libraries provided by the VEE Port.

To declare an Android Pack dependency, edit the `module.ivy` file of the VEE Port and add the following line within the `<dependencies>` element:

```
<dependency␣
↪org="com.microej.android.pack" name="[NAME]-android-pack" rev="[VERSION]"/>
```

MicroEJ Android Packs

MicroEJ provides Android Packs for some Foundation Libraries:

Name	Module	Implemented Libraries
Core Android Pack	com.microej.android.pack#core-android-pack	EDC, BON, SNI, Trace
UI Android Pack	com.microej.android.pack#ui-android-pack	MicroUI, Drawing
VG Android Pack	com.microej.android.pack#vg-android-pack	MicroVG

Note: Some Foundation Libraries such as FS and NET do not require an Android Pack as their APIs are already implemented by the Android SDK.

For more information on the usage and limitations of each Android Pack, refer to its README.

Custom Android Packs

A MicroEJ Application may call native methods, which require a different implementation on each execution target (embedded device, Simulator or Android). Therefore if an Application is executed on Android, the VEE Port should provide an implementation of these native methods for Android. This dedicated implementation is called an Android mock, and it is usually packaged in an Android Pack. This section explains how to develop a custom Android Pack including an Android mock.

Note: Currently, VEE Ports and their components can not be developed in *SDK 6*. This means that Android Packs must be developed with *SDK 5* and MMM, and cannot be developed with Android Studio and Gradle.

Setting Android SDK Environment Variable

Since the Android mock will be compiled using Android SDK, you should have Android Studio and Android SDK installed. If it is not set already on your system, you should set the ANDROID_HOME environment variable.

You can follow these steps to find the Android SDK location on your system:

- In Android Studio, select File > Settings… .
- In the settings dialog, find Android SDK and copy the path set as Android SDK Location .

  On Windows, this path is typically C:\Users\[USER]\AppData\Local\Android\Sdk .

  Make sure to restart MicroEJ SDK after setting the environment variable.

Creating the Android Pack Module

The first step is to create the custom-android-pack project:

- In MicroEJ SDK, select File > New > Project… .
- In the wizard dialog, select MicroEJ > Module Project and click on Next > .
- In the new module dialog, type custom-android-pack as Project Name and as Module , choose the Organization and Revision of your choice, select product-java as Skeleton and click on Finish .

  By default, the library built by the module is not packaged as an Android mock. To make sure that the library is added to the list of Android mocks, edit the module.ivy file of the project and add the following lines within the <ea:build> element:

```
<ea:property name="target.main.artifact.rip.relativedir" value="android/mocks/dropins"/>
```

Compiling against Android SDK

By default, the library is compiled against the JRE. Both the Eclipse project and the MMM build must be configured to compile against Android SDK rather than the JRE.

First, the JRE must be replaced by Android SDK in the build path of the Eclipse project:

- Right-click on the project, select Build Path > Configure Build Path… .
- In the properties dialog, open the Libraries tab, select JRE System Library , click on Remove and click on Add Variable… .
- In the classpath entry dialog, click on Configure Variables… .
- In the variables dialog, click on New… .
- In the new variable dialog, type ANDROID_HOME as Name , type the Android SDK location as Path and click on OK .
- Back to the variables dialog, click on Apply and Close .
- Back to the classpath entry dialog, select the ANDROID_HOME variable and click on Extend… .
- In the variable extension dialog, browse the platforms/android-[VERSION]/android.jar file and click on OK .
- Back to the properties dialog, click on Apply and Close .

  Finally, the JRE must be replaced by Android SDK in the build path of the MMM module:
- Edit the module.ivy file and add the following lines within the <ea:build> element:

```
<ea:property name="include.java.runtime" value="false"/>
<ea:property name="no.obfuscation" value="true"/>
```

- Create a file named module.ant at the root of the project with the following content:

```
<project name="custom-android-pack" xmlns:ea="antlib:org.apache.easyant">
    <target name=
↪"-custom-android-pack:augment-classpath" extensionOf="abstract-compile:compile-ready">
        <property environment="env"/>
        <ea:path pathid="compile.main.classpath" overwrite="prepend">
          ␣
↪   <fileset file="${env.ANDROID_HOME}/platforms/android-[VERSION]/android.jar"/>
        </ea:path>
    </target>
</project>
```

- In this module.ant , replace [VERSION] in the <fileset> element by the minimum Android SDK version required by your Android mock.

Implementing the Android mock

You can add the Java source code of your Android mock into the src/main/java folder of the project. At runtime, the Android mock will be added to the classpath before the code of the Application and before its dependencies. This allows you to replace the implementation of any Java class in an Android mock. The recommended practice is to replace only the classes which include native methods.

Using the Android Pack in the VEE Port

To build the Android Pack, right-click on the project and select Build Module .

The Android Pack can be included in a VEE Port by declaring a dependency in the module.ivy of the VEE Port as explained in the first subsection:

```
<dependency org="[ORGANIZATION]" name="custom-android-pack" rev="[VERSION]"/>
```

3.8.3 iOS Compatibility Kit

MicroEJ provides a set of tools and libraries to run applications powered by MicroEJ on iOS. This allows for the same application to be developed, simulated, tested, and executed on MicroEJ VEE and iOS alike.

Thanks to the iOS Compatibility Kit, watch faces can be developed once and deployed on both a smartwatch (MicroEJ VEE) and its companion smartphone app (iOS), enabling consistent functionality and appearance across both devices. This provides a good user experience for the user while minimizing code duplication and maintenance for the developer.

Software Architecture

The iOS Compatibility Kit provides a JDK which can be used to compile and run Java code on iOS. The code of the MicroEJ Application and of the libraries it depends on is executed on a Java SE VM started by the iOS app.

The JDK runtime includes JavaFX to be able to display Applications which use *MicroUI* or *MicroVG*.

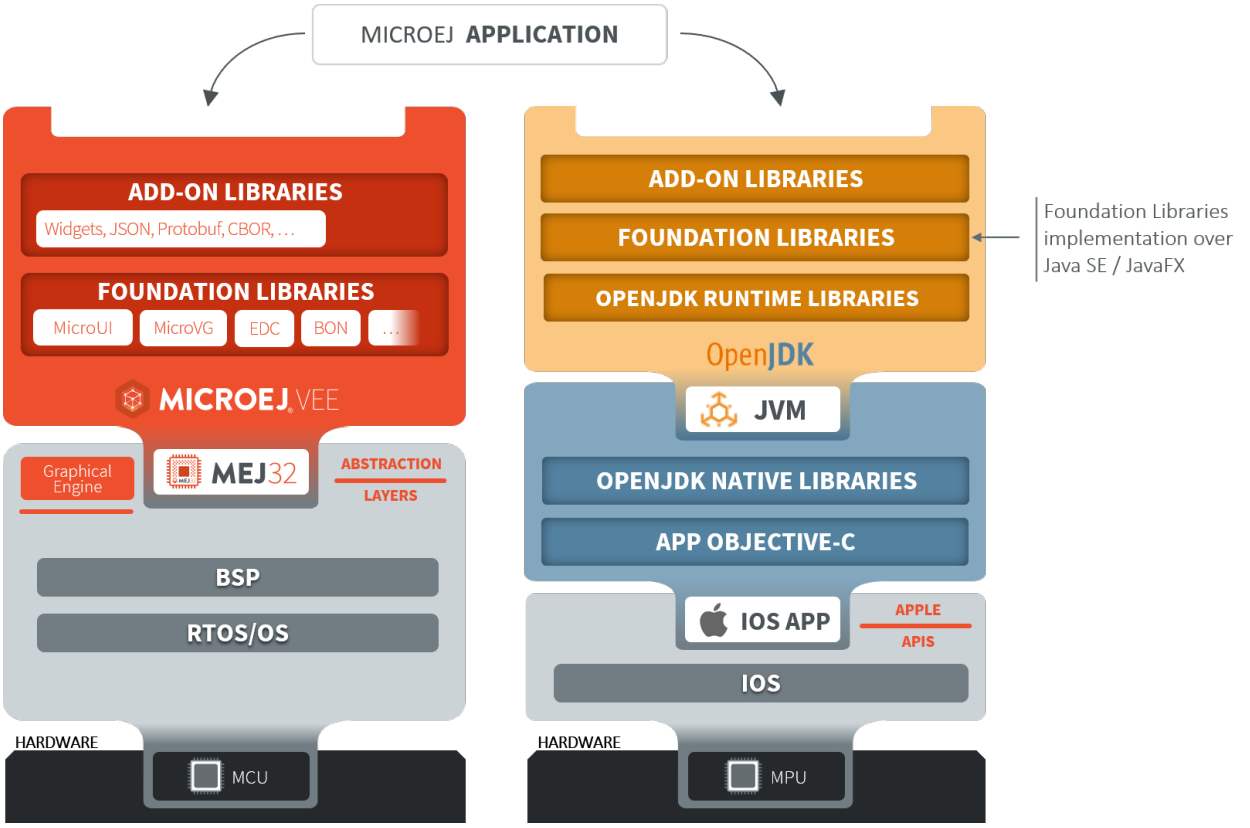


Fig. 294: Software Architecture

Workflow

The iOS app can be developed and built on Xcode as a regular iOS application. To run a MicroEJ Application, the Xcode project is configured to embed:

- the runtime libraries of the JDK (JARs and native libraries),
- the implementation of the MicroEJ Foundation Libraries compiled against Java SE and JavaFX APIs,
- the code and resources of the MicroEJ Application and of the Add-On Libraries that it depends on.

Evaluation

The iOS Compatibility Kit is available on demand. You can contact *MicroEJ Support* to evaluate this solution.

3.8.4 Offloading

Many high-end smartwatches rely on a microprocessor running Android. The power consumption of these devices is fairly high and show an average battery life of one or two days. Integrating an extra low-power microcontroller into the watch's hardware enables the delegation of specific tasks from the main microprocessor, resulting in an increased battery life. Keeping a powerful microprocessor on the hardware ensures the ability to display high-performance animations and access the Android ecosystem.

Solution

VEE Wear offers a comprehensive solution for software development on this dual architecture:

- MicroEJ VEE enables the execution of applications written in high-level code through virtualization on the microcontroller.
- the *Android Compatibility Kit* allows the execution of the same application on the microprocessor without the need for re-implementation.
- the Offloading Framework provides the ability to switch the application context between the two processors depending on their capabilities and on the application flow.

One Code, Two Targets

MicroEJ Application development shares the same programming language as Android Application development. Thanks to this similarity, any MicroEJ Application code is compatible with the Android runtime environment. The *Android Compatibility Kit* provides the tools and libraries to execute a MicroEJ Application on Android.

Using this solution, the application code can be programmed once and executed both on the low-power microcontroller and the high-power microprocessor.

Offloading Framework

The offloading strategy is a set of rules defined by the design of the watch.

Here is a non-exhaustive list of common offloading rules:

- switching to the low-power microcontroller after a few seconds of user inactivity (often called ambient mode)
- switching to the high-power microprocessor when starting a high-performance animation
- switching to the high-power microprocessor when navigating to a menu which is only available on Android

  Switching from a processor to the other may require to synchronize the state of the software and to provide the necessary data for the other processor to take over. The time to wake-up the processor and to synchronize the data has to be taken into account when designing the software architecture.

Once these rules have been decided, the Offloading Framework APIs can be used to wake up the other processor, to synchronize data, to be notified when the other processor is ready, to hand over the control of the display, to put the processor to sleep, etc.

Evaluation

The Android Compatibility Kit can be evaluated by following *its documentation*.

An demonstration with a sample offloading framework is available on demand. You can contact *MicroEJ Support* to test this demonstration.

3.8.5 VEE Script

Introduction

VEE Script is a scripting language supported on MICROEJ VEE.

Its syntax is a subset of JavaScript. This subset includes most expressions but excludes statements and blocks.

The strengths of VEE Script are its low footprint, its optimized performance, its customizable semantics and its ability to evaluate expressions depending on the runtime context.

To have the best runtime performance, expressions are preprocessed offboard. During this phase, expressions are parsed, converted into ASTs and serialized into a binary file. At runtime, the ASTs are deserialized from the binary file and the expressions can be evaluated efficiently.

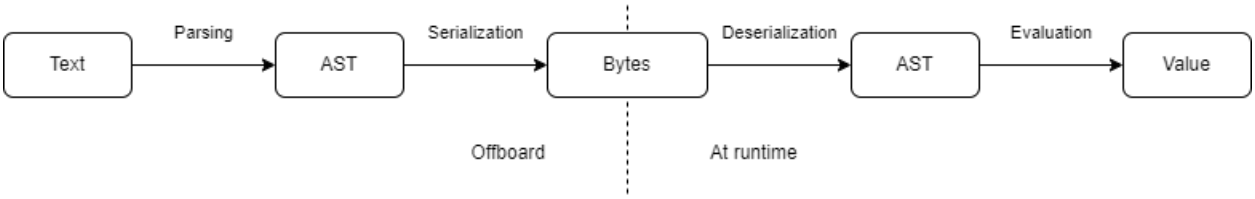


Fig. 295: Preprocessing and Interpretation Flow

VEE Script can be used for various use cases, including the design and rendering of dynamic GUIs and real-time information displays. For instance, VEE Script is currently used in the VEE Wear Kernel to design and render watchfaces.

Tags and Functions

VEE Script introduces tags and functions, which can be evaluated to different values depending on the runtime context. For example, a "*current time*" tag or a "*generate random value*" function would return a different value every time they are interpreted.

Moreover, tags and functions semantics are defined by the product. This allows every product to define its own DSL, specific to the type of device and to its unique features. For example, a smartwatch could introduce a tag for battery level while a washing machine could introduce a tag for remaining cycle time.

Examples

For example, the following script could be used to display the heart rate:

```
product.evaluateTag('heart_rate') + ' BPM'
```

And the following script could be used to animate the position of a widget:

```
100 * product.applyFunction('sin', product.evaluateTag('current_time') / 1000 * 3.14)
```

Expression Types

VEE Script supports the following expression types:

- literal
- unary operation (+, -)
- binary operation (*, /, %, +, -, <, >, <=, >=, ==, !=, &&, ||)
- conditional (?:)
- tag evaluation
- function application

Value Types

Once an expression is evaluated, its value has one of the following types:

- boolean
- number
- string

Language Syntax

The scripting language is a subset of JavaScript. Its syntax and semantics are specified by ECMAScript 5.1.

The following table summarizes the grammar rules of the language:

Grammar Rule	Example	Expression Type
<boolean>	*true*	Boolean literal
<number>	*3.5*	Number literal
'<string>'	*'hello world'*	String literal
<operator> <expression>	*-10*	Unary operation
<expression> <operator> <expression>	*1 + 2*	Binary operation
<expression> ? <expression> : <expression>	*2 > 1 ? 5 : 10*	Conditional
product.evaluateTag('<tag>')	*product.evaluateTag('battery')*	Tag evaluation
product.applyFunction('<function>', [expressions])	*product.applyFunction('sqrt', 9)*	Function application
product.constants.<constant>	*product.constants.pi*	Literal

Language Configuration

Since the semantics of the language depend on the product, the expression parser and interpreter must be configured.

The following information must be supplied to the parser:

- the name and value of every constant
- the name of every tag
- the name of every function

The following information must be supplied to the interpreter:

- the semantics of every tag
- the semantics of every function

Evaluation

The VEE Script parser and interpreter are available on demand. You can contact *MicroEJ Support* to evaluate this solution.

3.9 VEE Energy User Guide

VEE Energy is a purpose-built application framework designed for the next generation of energy devices, including smart meters, gateways, and connected grid infrastructure. Optimized for low-power MCUs and MPUs, VEE Energy delivers high performance while maintaining a minimal memory footprint, making it ideal for cost-effective energy solutions. VEE Energy includes all critical features from the MicroEJ runtime, ensuring:

- Comprehensive Framework integrating security, file system, networking and IoT
- Safe and secure Apps deployment
- Extensive data handling capabilities with Edge AI and signal processing
- Support for industry-standard protocols like DLMS/COSEM

MicroEJ Multi-Sandboxing

MicroEJ Multi-Sandboxing enables secure execution of multiple isolated applications on a single device, allowing dynamic app deployment and updates without compromising system stability. By leveraging lightweight virtualization, it optimizes resource usage on constrained hardware while ensuring robust security and scalability.

- Resource Security Manager: Enhances system reliability by ensuring that a faulty app does not compromise the entire system's performance.
- App Management and Analytics: Allows for local management and analysis of applications, providing valuable insights and control.

Energy Framework

VEE Energy Framework supports essential metrology functions, data storage, and power switching for comprehensive meter management. Energy analytics VEE Energy facilitates local analytics with Digital Signal Processing (DSP) and Edge Artificial Intelligence (AI), enabling advanced data analysis and decision-making on the device.

IoT and Energy protocols

VEE Energy integrates support for extensive connectivity and communication protocols, facilitating the design of AMI2.0 energy meters:

- Standard IoT and connectivity stacks part of the *VEE networking add-on libraries*.
- Energy specific stacks like DLMS, DNP3, OCCP, Modbus, IEC61850 with partners like Kalkitech.
- Support for Smart Home protocols like Matter to bridge energy with smart home and HVAC.

For more information about VEE Energy, please contact your MicroEJ sales representative.

3.10 Training Courses

Welcome to the MicroEJ training courses!

This documentation offers online courses to help you learn the essentials of Firmware development using MICROEJ SDK.

Whether you're a beginner or more advanced, you will gain a solid understanding of Application development using MICROEJ SDK and MICROEJ VEE integration on custom devices. Additionally, you will discover the feature-rich ecosystem provided to developers to ease developments on a daily basis.

Each training will specify its prerequisites. For courses involving hands-on activities, you may need to use one of the following development kits:

- NXP i.MX RT1170 Evaluation Kit.
- STM32F7508-DK Evaluation Kit.

Alternatively, the hands-on activities can be fully completed using MICROEJ SDK Simulator.

3.10.1 For Beginners

MICROEJ SDK Basics

Description

This training session introduces you to the MicroEJ ecosystem, guiding you through the process of creating your first application project. You will run your project on both a simulator and an actual device using MICROEJ SDK.

What you will learn:

- Understand the VEE Port concepts
- Create and run your first application project with MICROEJ SDK
- Learn how to configure your Application Project
- Use a Front Panel project
- Get an overview of MICROEJ SDK Development Tools
- Call a C function from Managed Code
- Blink an LED from Managed Code

Intended Audience

This training is designed for developers looking to gain a first understanding of the MicroEJ development flow.

Prerequisites

To get the most out of this training, participants should have:

- A good understanding of the *Overview* section.
- A basic knowledge of the *MicroEJ Glossary*.
- Access to the NXP i.MX RT1170 Evaluation Kit.

Training Resources

Unit 1: MICROEJ SDK Basics

This training is provided as a PDF, along with a downloadable `.zip` package of resources:

MICROEJ SDK 6

MICROEJ SDK 5

-
- Resources
-
- Resources

Unit 2: Getting Familiar with Gradle

MICROEJ SDK6 relies on the Gradle build system.

If you are not familiar with Gradle yet, we encourage you to follow the Introduction to Gradle for Developers training to get familiar with it.

Mastering MICROEJ SDK Development Tools

Description

This training describes the available tools provided by MICROEJ SDK to accelerate your product development.

Intended Audience

This training is designed for developers who want to learn about the tools available for developing and debugging embedded applications.

Prerequisites

To get the most out of this training, participants should have:

- A good understanding of the *Overview* section.
- A basic knowledge of the *MicroEJ Glossary*.
- Access to the NXP i.MX RT1170 Evaluation Kit.

Training Resources

Unit 1: Master logging within your Application

- Read the following documentation: *Code Instrumentation for Logging*.

Unit 2: MicroEJ Development Tools Overview

This training is provided as a PDF, along with a downloadable .zip package of resources:

- (for MICROEJ SDK 6)
- Resources

C / Managed Code Communication

Description

In this training, you will get an in depth view of the different ways to perform C / Managed Code communication using Native Interface Mechanisms.

What you will learn:

- SNI Overview
- Exposing a C API in Java code
- Immortal Arrays
- Error Management
- Blocking Functions and Asynchronous Code
- Referencing C resources from Managed Code
- Overview of C / Managed Code Communication libraries: Event Queue, UI Pump, Shielded Plug

Intended Audience

This training is designed for developers who want to learn about communication mechanisms between C and Managed Code.

Prerequisites

To get the most out of this training, participants should have:

- A good understanding of the *Overview* section.
- A basic knowledge of the *MicroEJ Glossary*.
- A basic knowledge of C programming language.
- A basic knowledge of Java programming language.

Training Resources

Unit 1: C / Managed Code Communication with MICROEJ VEE

This training provides a comprehensive overview of the various methods for C / Managed Code communication.

This training is provided as a PDF, along with a downloadable `.zip` package of resources:

-
- Resources

Note: The SNI and Blocking Functions section requires to use the NXP i.MX RT1170 Evaluation Kit to run the hand's on.

The rest of the training is board agnostic.

Unit 2: Implementing a Blocking Java Native Method with SNI

This training describes a use case showing how to properly implement a blocking native method:

- *Implement a Blocking Java Native Method with SNI*

Implement a Blocking Java Native Method with SNI

This training describes the good practices to follow when implementing a blocking native method in C. A native method is a method declared in Java code with the `native` keyword and implemented in C using the *Simple Native Interface (SNI)*.

Intended Audience

The audience for this document is Platform developers who want to implement Abstraction Layer interfaces.

Prerequisites

The following document assumes the reader already has a setup ready to run a MicroEJ Standalone Application on a target device.

The following document also assumes the reader is familiar with the *Simple Native Interface (SNI)* mechanism. If not, the CallingCFromJava GitHub example shows the minimum steps required to create a Java program that makes a call to a C function via SNI.

Overview

The Core Engine implements a green thread architecture with all the threads executed within one single RTOS/OS task. Thus, it embeds its own scheduler that controls the execution of the threads. With such an architecture, the Core Engine cannot preempt a thread that executes a native method. Therefore a blocking native method will prevent the execution of other threads. To mitigate the contention, a native method must explicitly yield its current use of the processor.

This training will explain how to use SNI to implement a blocking Java native method without blocking the execution of other threads.

Requirements

A MicroEJ Platform with (at least):

- EDC-1.3.
- SNI-1.4.

Example Code

Let's start with a MicroEJ Standalone Application that calls a blocking Java method implemented in C.

The following example waits for a button event and prints the index of the pressed button.

The MicroEJ Application code:

```
package example;

public class NativeCCallExample {

   public static void main(String[] args) {

      while (true) {
         System.out.println("Waiting for a button event...");
         int buttonIndex = waitButton();
         System.out.println("Button pressed: " + buttonIndex);
```

(continues on next page)

(continued from previous page)

```
        }
    }

    public static native int waitButton();
}
```

The C implementation of the `waitButton()` native has been written in pseudo-code. It should be adapted according to the BSP of the target board.

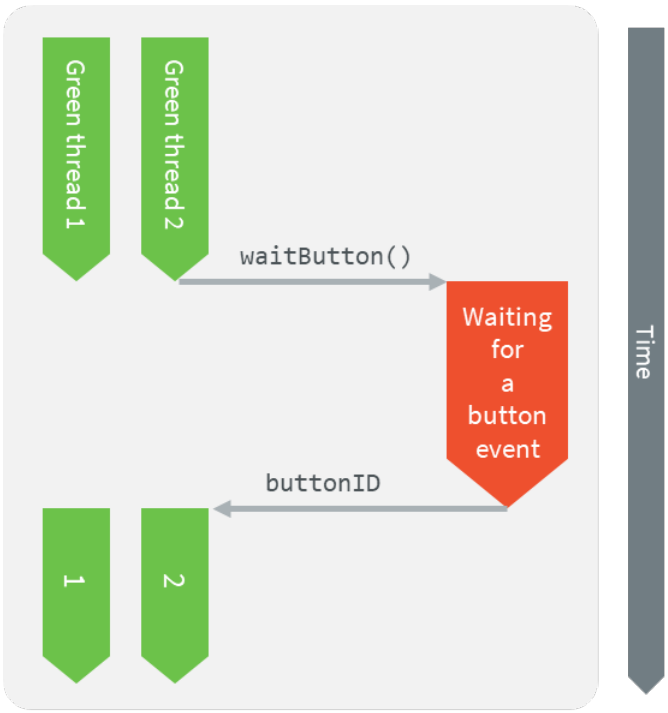
```
#include "semaphore.h"
#include "sni.h"

static int pressed_button_index;
static Semaphore button_semaphore;

void button_init(){
    button_semaphore = SemaphoreCreateBinary();
}

jint Java_example_NativeCCallExample_waitButton(){
    // Wait for a button event
    semaphoreTake(button_semaphore);
    return pressed_button_index;
}

/** Interrupt request handler called when a button is pressed. */
int buttonIRQ(int button_index){
    pressed_button_index = button_index;
    semaphoreGiveFromISR(button_semaphore);
}
```

Application Behavior

In this example, the execution of the `waitButton()` native method will block until a button is pressed. In other words, while `Java_example_NativeCCallExample_waitButton()` has not returned, no other thread can be scheduled.

This is because the native function is called in the same RTOS/OS task as the Java application.

This schematic explains what is going on:



The MicroEJ VEE RTOS task

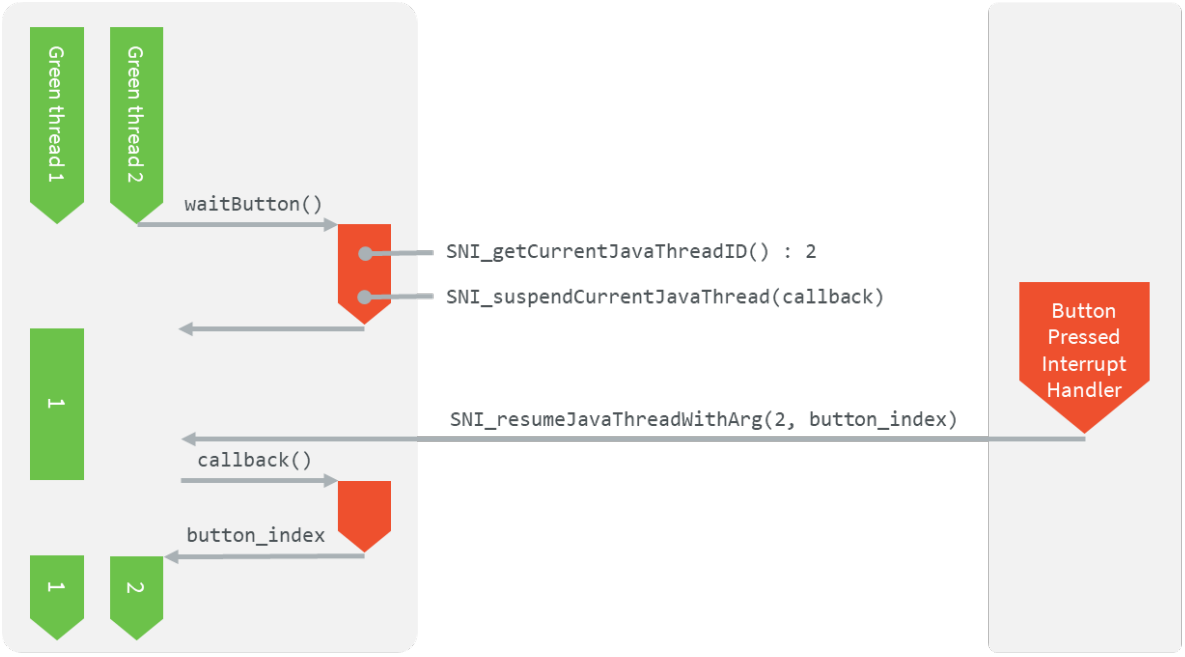
Implement a Non-Blocking Method

This section will explain how to update the example code to make a non-blocking method.

Here is a summary of what will be done in C:

- Signal the Core Engine to suspend the current thread when the native function returns.
- Remove the blocking operations from the native function so that it returns immediately.
- Implement a callback function that returns the index of the pressed button.
- Register this callback function in the Core Engine to call it when the thread is resumed.
- Resume the thread when a button is pressed.

This schematic summarizes the steps described above:



Update the C Native Function Implementation

Step 1: Update the C Native Function

The Java_example_NativeCCallExample_waitButton() function will now suspend the current thread. It will also store the information required to resume it and return the index of the pressed button.

The SNI functions used in this example are defined in sni.h . See this header file for a more detailed description of the API.

- Store the ID of the thread that called the function. This ID should be stored in a global variable. It is used to resume the thread when a button is pressed.

```
java_thread_id = SNI_getCurrentJavaThreadID();
```

- Signal the Core Engine to suspend the current thread and specify the callback function to be called when the thread is resumed. Let's call the callback function waitButton_callback() .

```
SNI_suspendCurrentJavaThreadWithCallback(0, (SNI_callback)waitButton_callback, NULL);
```

The function SNI_suspendCurrentJavaThreadWithCallback() returns immediately. The current thread is actually suspended when the native function returns.

The value returned by the Java_example_NativeCCallExample_waitButton() doesn't matter anymore. The callback function will be in charge of returning the value.

The updated Java_example_NativeCCallExample_waitButton() function should look like this:

```
static int32_t java_thread_id;

jint Java_example_NativeCCallExample_waitButton(){

    java_thread_id = SNI_getCurrentJavaThreadID();

    SNI_
↪suspendCurrentJavaThreadWithCallback(0, (SNI_callback)waitButton_callback, NULL);

    return SNI_IGNORED_RETURNED_VALUE; // Returned value not used
}
```

Step 2: Update the Button Interrupt Function

The role of the button interrupt is now to resume the thread when a button event occurs. Update it this way:

```
int buttonIRQ(int button_index){
    SNI_resumeJavaThreadWithArg(java_thread_id, (void*)button_index);
}
```

The button's index is passed to the function `SNI_resumeJavaThreadWithArg()` so that the callback retrieves it when the thread is resumed.

Step 3: Implement the Callback Function

The callback function must have the same signature as the SNI native (same parameters and return type): `jint waitButton_callback()`.

The callback function is automatically called by the thread when it is resumed. Use the `SNI_getCallbackArgs()` function to retrieve the arguments that was previously given to the `SNI_suspendCurrentJavaThreadWithCallback()` or `SNI_resumeJavaThreadWithArg()` functions.

```
jint waitButton_callback()
{
    int button_index;
    SNI_getCallbackArgs(NULL, (void*)&button_index);
    return (jint)button_index; // Actual value returned to Java
}
```

Application Behavior

In this configuration, calling the native method `waitButton()` will still return only when a button is pressed, but it will not prevent other threads from being scheduled.

Simulate Hardware Behavior using Mocks

Description

In this training, you will learn how to use mocks in order to enhance application development on Simulator.

Intended Audience

This training is designed for developers who want to gain a first understanding of extending the Simulator with mocks.

Prerequisites

To get the most out of this training, participants should have:

- A good understanding of the *Overview* section.
- A basic knowledge of the *MicroEJ Glossary*.

Training Resources

Unit 1: Read the Mock Developer Guide

Read the *Mock* Developer Guide to get familiar with Mock development.

Unit 2: Run Mock Framework Examples

Those examples are showing how to implement mocks using the Mock Framework.

Run the samples on Simulator to get familiar with the Mock Framework:

- Mock Framework Examples (for MICROEJ SDK 6)

3.10.2 For Application Developers

Get Started with Sandboxed Applications

Description

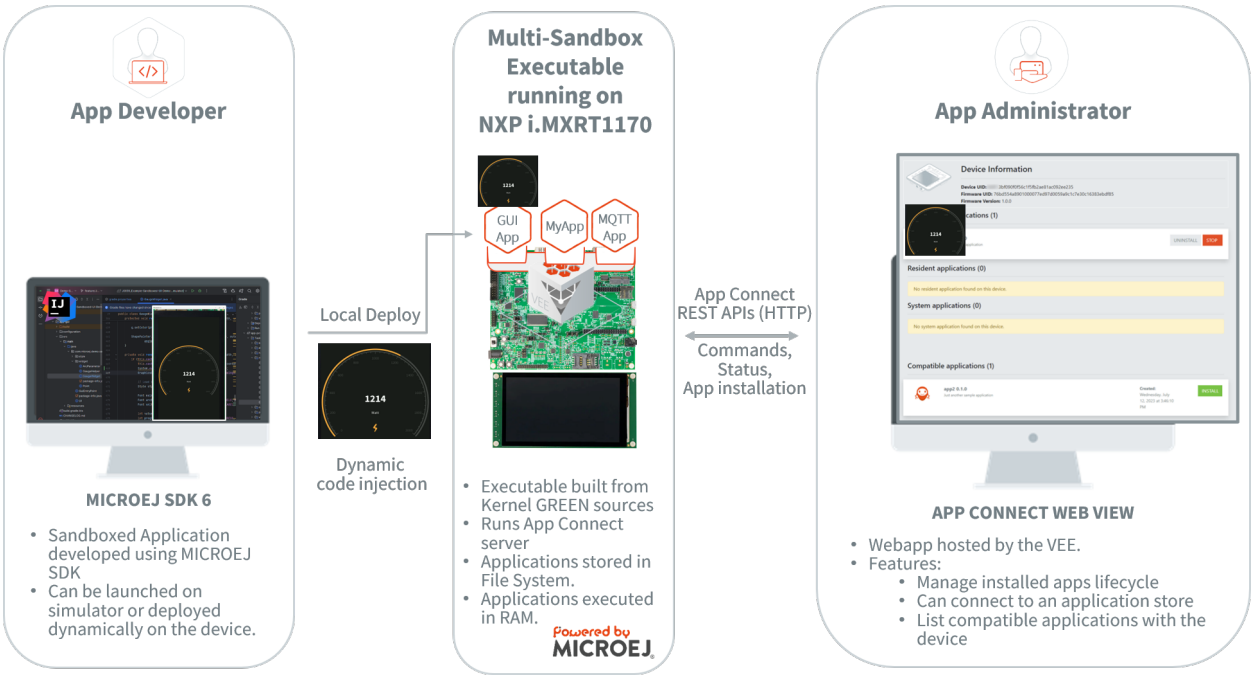
This training will show you how to run Sandboxed Applications on top of a pre-built Multi-Sandbox Executable for the NXP i.MXRT1170 Evaluation Kit.

During this training, you will learn how to run Sandboxed Applications in Simulation using the Virtual Device and on NXP i.MXRT1170 Evaluation Kit:

1. *Run the Demo-Sandboxed-Applications*.
2. *Create Sandboxed Application and run it*.

In case you are not familiar with MicroEJ, please visit Discover MicroEJ to understand the principles of our technology.

The below schematic summarizes the Multi-Sandbox features that will be demonstrated in this training:



Intended Audience

This training is designed for application developers looking to gain a first understanding of the Multi-Sandbox development flow.

Prerequisites

Note: This Getting Started has been tested in the following conditions:

- Windows 10/11.
- Multi-Sandbox Executable: NXP-MIMXRT1170-3.0.0_GCC_GREEN-2.1.1.elf.
- Demo-Sandboxed-Applications 2.0.0 .

To get the most out of this training, participants should have:

- A good understanding of the *Overview* section.
- A basic knowledge of the *MicroEJ Glossary*.

Hardware setup

To follow this training, you need:

- A NXP i.MX RT1170 Evaluation Kit EVKB.
- A RK055HDMIPI4MA0 display panel.
- A FAT32-formatted microSD card.
- An Ethernet cable

Environment Setup

To follow this training, you need:

- An RS232 Terminal (e.g. Termite).

Also, follow the steps described in the sections below to complete your environment setup.

Install MICROEJ SDK 6

Follow *MICROEJ SDK 6 installation Guide*. IntelliJ IDEA is used in this training, but feel free to use your favorite IDE.

Accept the MICROEJ SDK EULA

You may have to accept the SDK EULA if you haven't already done it, please have a look at *SDK EULA Acceptation*.

Hardware Setup for NXP i.MXRT1170 EVKB

Refer to the NXP i.MXRT1170 EVKB *Hardware Setup* guide.

Additionally, make sure to:

- Insert a micro-SD card (FAT32-formatted) in the board connector.
- Connect the 1GB Ethernet connector to the internet.

Flash the Multi-Sandbox Executable on your NXP i.MXRT1170 Evaluation Kit

Download the Multi-Sandbox Executable: NXP-MIMXRT1170-3.0.0_GCC_GREEN-2.1.1.elf.

Note: The Multi-Sandbox Executable used in this documentation is built from the Kernel GREEN sources. Refer to the *Get Started with Kernel* training to build your own Multi-Sandbox Executable.

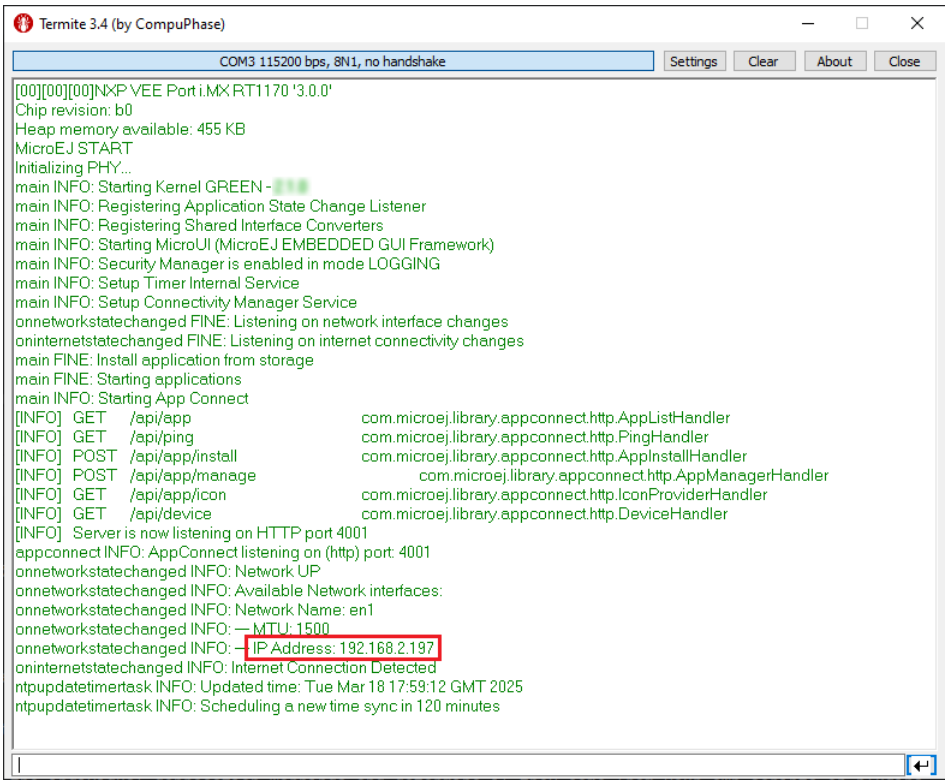
If you want more information about this Multi-Sandbox Executable, the Javadoc and the Release notes are available in this directory.

Flash the Multi-Sandbox Executable using LinkServer :

- Download and install LinkServer for Microcontroller (minimum version 1.6.133).

- Once installed, the LinkServer installation folder must be set on your Path. To do so:
  - Open the Edit the system environment variables application on Windows.
  - Click on the Environment Variables... button.
  - Select Path variable under the User variables section and edit it.
  - Click on New and point to the LinkServer installation folder located where you installed LinkServer (e.g. nxp/LinkServer_1.6.133/).
- Run the command LinkServer flash MIMXRT1176xxxxx:MIMXRT1170-EVKB load NXP-MIMXRT1170-3_0_0_GCC_GREEN-2.1.1.elf .

  Set up the logs output:
- Get the COM port where your board is connected (if you are using Windows, you can open your Device Manager from the Windows menu).
- Set up a serial terminal (e.g. Termite) to see output logs from the board. Refer to the NXP i.MXRT1170 EVKB *Hardware Setup* to setup the serial terminal.
- Press the board's reset button.
- Get your board's IP address. You will find it in the logs output:



Congratulations! The Multi-Sandbox Executable is running on the NXP i.MXRT1170 Evaluation Kit and is ready to be used.

Training Course

Run the Demo-Sandboxed-Applications

The Demo-Sandboxed-Applications showcases the communication between Sandboxed Applications using the Shared Interfaces.

The following projects are provided:

- `app-power-provider` : application responsible for providing random power values in the system.
- `app-gui` : application responsible for visualizing the power values provided by `app-power-provider` .
- `app-mqtt-publisher` : application responsible for publishing the power values provided by `app-power-provider` to an MQTT topic.
- `sharedinterface` : shared library between apps that defines the shared interface for inter-app communication.

> **Warning:** If you are using your own Multi-Sandbox Executable to run this demo, make sure to configure the *Images Heap* size of your Kernel project to the size of your screen. The formula to compute the required size is $SCREEN_WIDTH \times SCREEN_HEIGHT \times BPP / 2$.
>
> Example with a 1280x720 screen in RGB565 format (16BPP). Images Heap size should be:
>
> - 1280x720x16/2 = 1843200 bytes

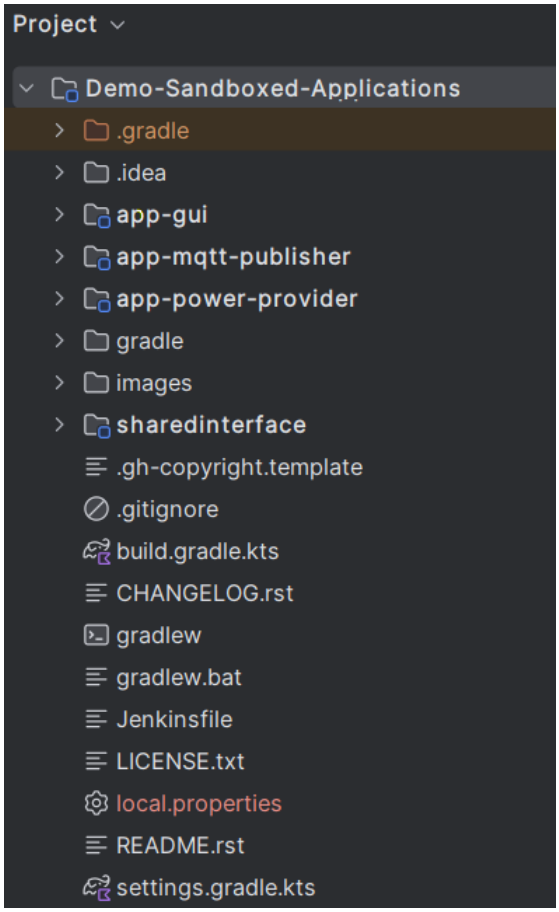
Import the Project

Import the project into your IDE:

Note: If you are using an IDE other than IntelliJ, please have a look at *Import a Project* section.

- If you are on the Welcome Screen, click on the `Open` button. Otherwise click either on `File` > `Open…` .
- Select the `Demo-Sandboxed-Applications` directory located where you downloaded it and click on the `OK` button.

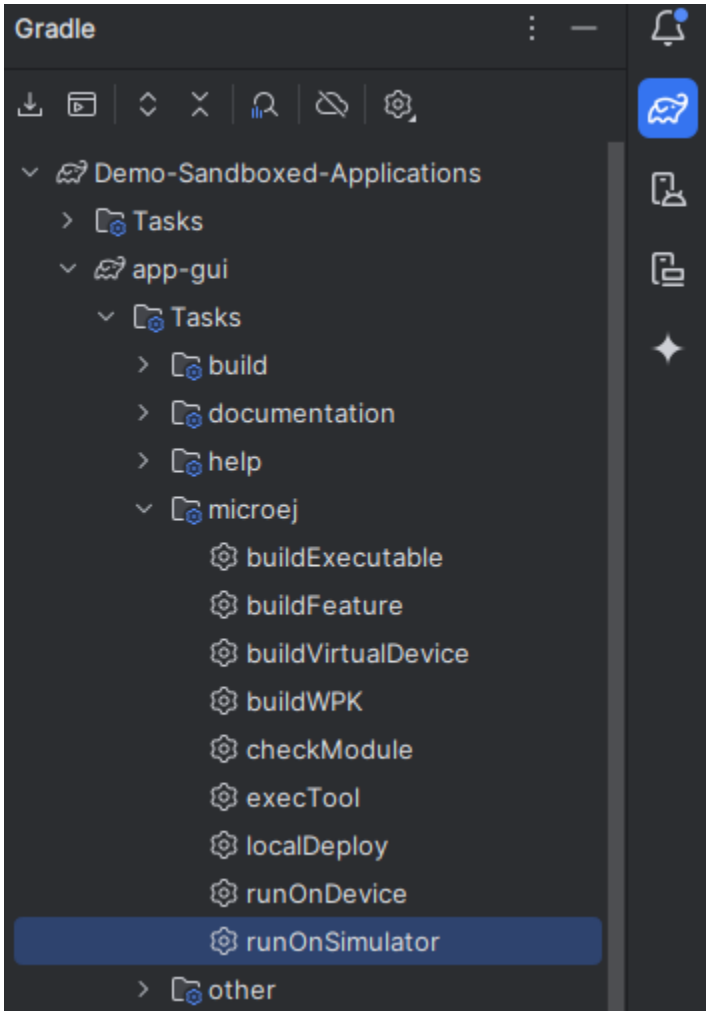
  The Gradle project should now be imported in IntelliJ, your workspace contains the following project in the `Projects` view:



Run the Demo-Sandboxed-Applications on the Virtual Device

In order to execute the applications on the Virtual Device, use the Gradle `runOnSimulator` task:

- Double-click on the `runOnSimulator` task of one the applications (`app-gui` , `app-power-provider` , `app-mqtt-publisher`):



Note: Each application is bound to each other. Running one of the applications on the Simulator will run all the applications.

The Virtual Device starts and executes the Demo-Sandboxed-Applications:

Run the Demo on the NXP i.MXRT1170 Evaluation Kit

The `Demo-Sandboxed-Applications` applications can be deployed on the NXP i.MXRT1170 Evaluation Kit using the `localDeploy` tool. Each application needs to be deployed individually.

Configure the `localDeploy` tool in the project:

- Open the `gradle.properties` file located at the root of the `Demo-Sandboxed-Applications` project.
- Get your board's IP address and add it to the `ipAddress` field.

Open the Gradle tasks view to deploy the applications on your NXP i.MXRT1170 Evaluation Kit. Deploy them in the following order:

1. Deploy `app-power-provider` : double-click on the `app-power-provider > Tasks > microej > localDeploy` task.
2. Deploy `app-gui` : double-click on the `app-gui > Tasks > microej > localDeploy` task.
3. Deploy `app-mqtt-publisher` : double-click on the `app-mqtt-publisher > Tasks > microej > localDeploy` task.

The applications are running on the NXP i.MXRT1170 Evaluation Kit:

Subscribe to the MQTT Topic to Get the Power Values

By default, the `app-mqtt-publisher` publishes the power values to a public MQTT Broker: test.mosquitto.org .

The MQTT topic is randomly generated at each application startup. It can be seen in the console logs:

Use an MQTT client to subscribe to the topic, for example, using Docker:

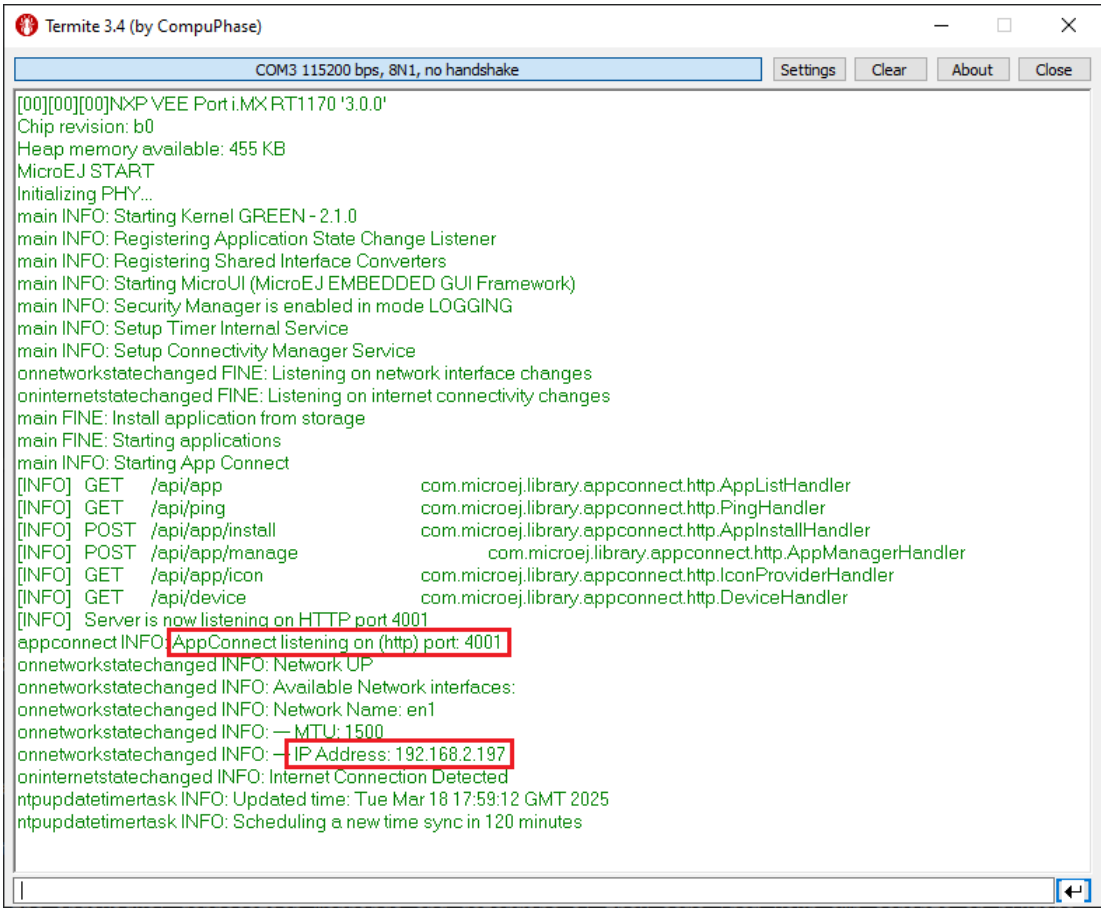
```
docker run -it --rm eclipse-mosquitto mosquitto_sub␣
↪-h test.mosquitto.org -p 1883 -q 1 -t microej/demo/sandbox/power_[YOUR_TOPIC_ID]
```

Every time a power value is published, it can be seen in the MQTT client console:

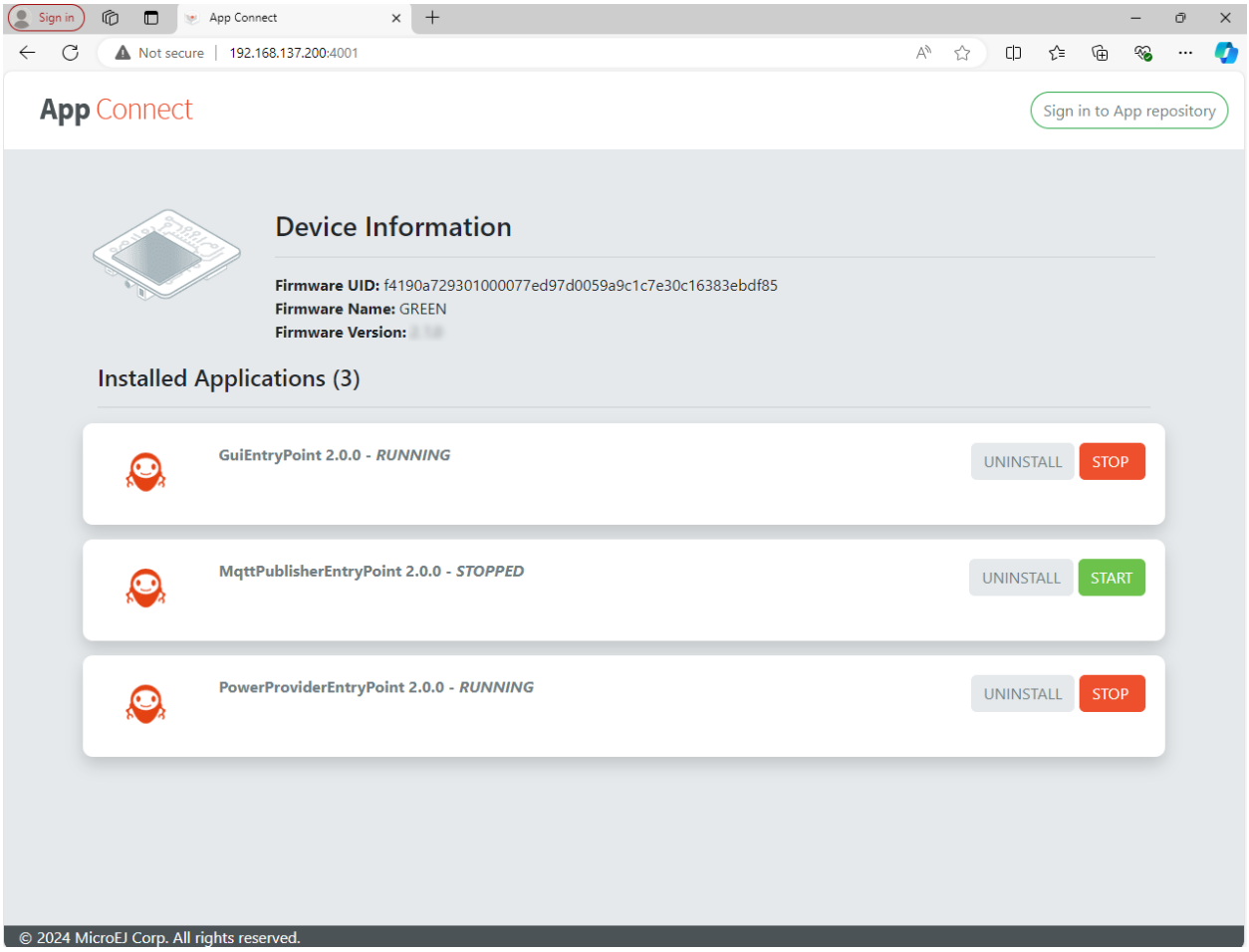
Managing Applications Lifecycle

The Multi-Sandbox Executable provides a web server to interact with the applications. Either through a Web UI or an HTTP API. This server listens on port 4001 by default.

The Server URL can be seen in the console logs:



Open the URL in a web browser, the installed applications can be seen. They can be Started / Stopped / Uninstalled :


Sign in
App Connect
Not secure
192.168.137.200:4001
App Connect
Sign in to App repository
Device Information
Firmware UID: f4190a729301000077ed97d0059a9c1c7e30c16383ebdf85
Firmware Name: GREEN
Firmware Version:
Installed Applications (3)
GuiEntryPoint 2.0.0 - RUNNING
UNINSTALL
STOP
MqttPublisherEntryPoint 2.0.0 - STOPPED
UNINSTALL
START
PowerProviderEntryPoint 2.0.0 - RUNNING
UNINSTALL
STOP
© 2024 MicroEJ Corp. All rights reserved.

Well Done!

Now you know how to run Sandboxed Applications on a Multi-Sandbox Executable!

The next step is about creating a Sandboxed Application project from scratch and running it on the Virtual Device and on the NXP i.MXRT1170 board.

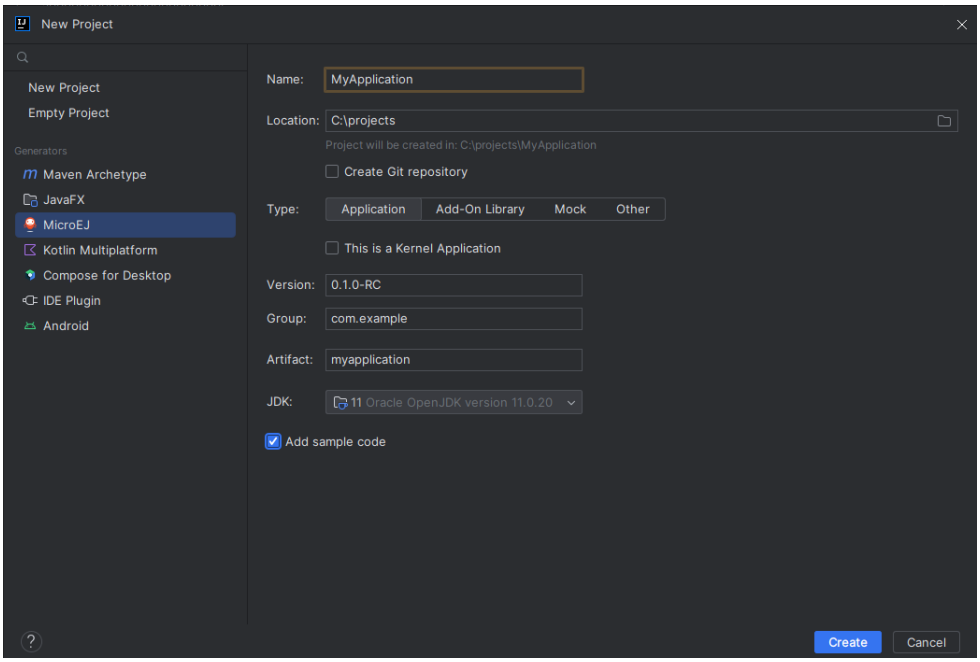
Create and Run a Sandboxed Application

Create the Sandboxed Application Project

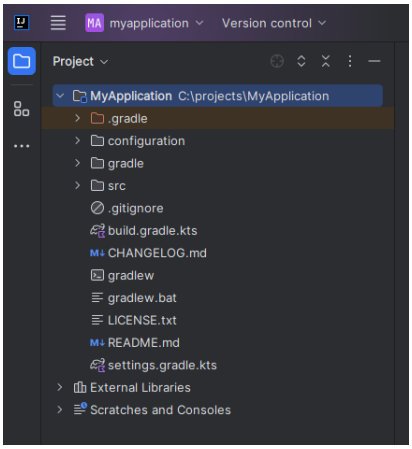
Note: If you are using an IDE other than IntelliJ IDEA, please have a look at *Create a Project* section.

Create a new Sandboxed Application project as follows in IntelliJ IDEA:

- Click on `File` > `New` > `Project…` .
- Select `MicroEJ` .
- Fill in the name of the project in the `Name` field.
- Select the location of the project in the `Location` field.
- Select the `Application` project type.
- Fill the version of the artifact to publish in the `Version` field.
- Fill the group of the artifact to publish in the `Group` field.
- Fill the name of the artifact to publish in the `Artifact` field.
- Keep the default JDK in the `JDK` field.



- Click on the `Create` button. The new project is created and opened:



Run the Sandboxed Application on the Virtual Device

MicroEJ provides ready to use kernels on the Developer Repository.

The `MyApplication` project needs to be configured to use a kernel:

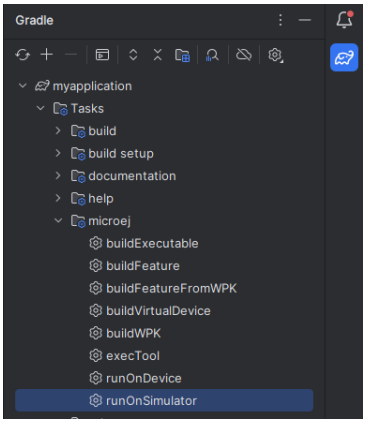
- Open the build.gradle.kts file of the `MyApplication` project.
- Add the dependency to the NXP i.MXRT1170 kernel: microejVee("com.microej.kernel:NXP-MIMXRT1170-3_0_0_GCC_GREEN:2.1.1")

Note: To use your own Kernel, refer to the procedure described in *Run a Sandboxed Application on your Multi-Sandbox Executable and Virtual Device*.

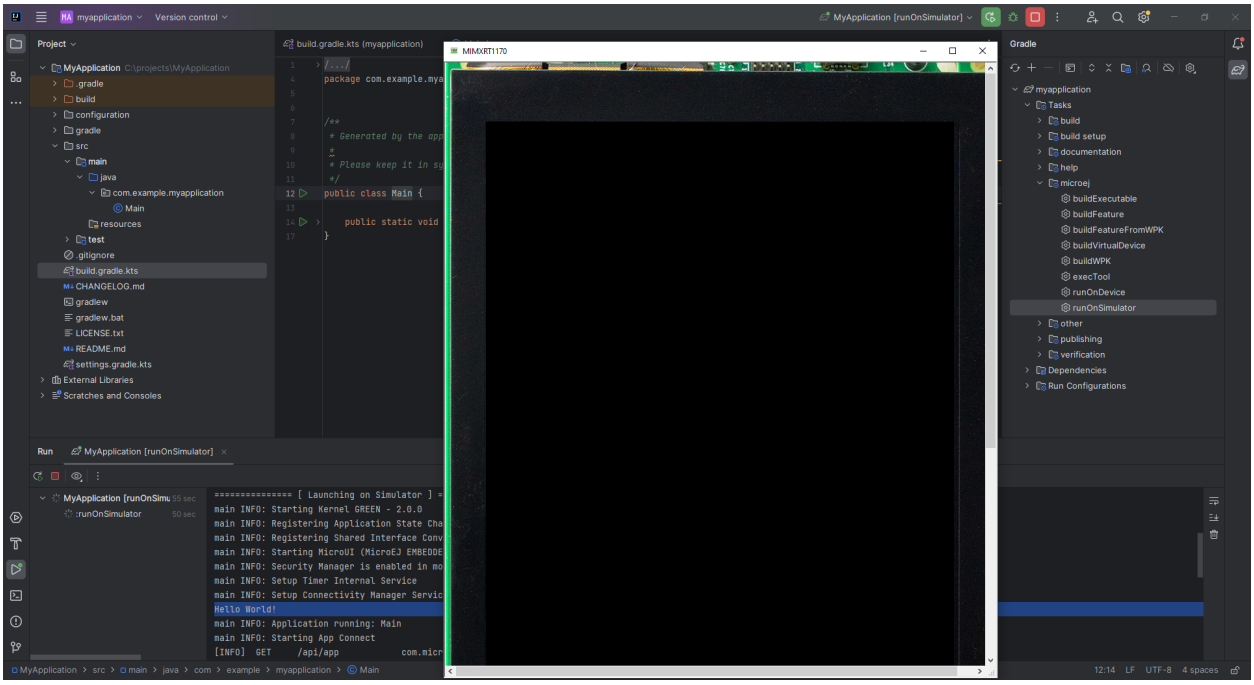
In order to execute the `MyApplication` project on the Virtual Device, the SDK provides the Gradle `runOnSimulator` task.

Note: If you are using an IDE other than IntelliJ IDEA, please have a look at *Run on Simulator* section.

- Double-click on the `runOnSimulator` task in the Gradle tasks view. It may take a few seconds to start:



The Virtual Device starts and executes the `MyApplication` project. The Hello World! message can be seen in the console:



Run the Sandboxed Application on the NXP i.MXRT1170 Evaluation Kit

The Multi-Sandbox Executable embeds a server that listens for Sandboxed Applications deployment commands.

The `MyApplication` project can be deployed on the NXP i.MXRT1170 Evaluation Kit using the `localDeploy` tool. This tool will deploy the application on the NXP i.MXRT1170 Evaluation Kit through your local network.

Configure the `localDeploy` tool in `MyApplication` project:

- Open the `build.gradle.kts` file of the `MyApplication` project.
- Paste the following code at the beginning of the file:

```
import com.microej.gradle.tasks.BuildFeatureTask
import okhttp3.MediaType.Companion.toMediaType
import okhttp3.MultipartBody
import okhttp3.OkHttpClient
import okhttp3.Request
import okhttp3.RequestBody.Companion.asRequestBody
import java.util.*
```

- Paste the following code at the end of the file:

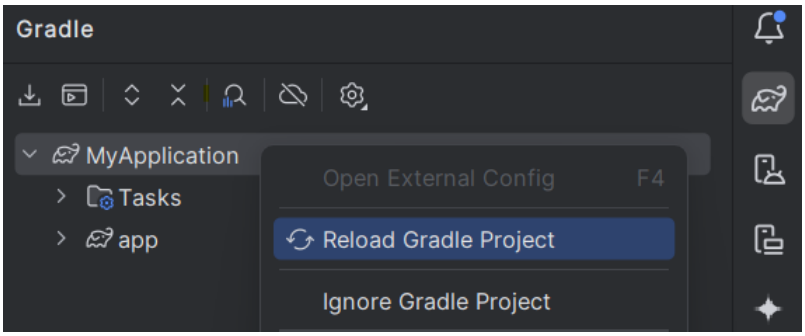
```
val buildFeatureTask = tasks.withType(BuildFeatureTask::class).named("buildFeature")
tasks.register("localDeploy") {
dependsOn("buildFeature")
group = "microej"

// Adjust the following variables to your needs
val boardIP = "<Board IP Address>" // board ip address
val boardPort = 4001 // AppConnect port
val force = true // overwrote existing app with same name
val start = true // start app after install
// Note:␣
↪if your metadata (feature.kf) is part of '/src/main/resources', modify this path accordingly
val featureKFFilePath = "generated/microej-app-wrapper/feature-resources/feature.kf"

doLast {
  val applicationFOFile = buildFeatureTask.get().featureFile.get().asFile
  val properties = Properties()
  project.
↪layout.buildDirectory.file(featureKFFilePath).get().asFile.inputStream().use(properties::load)
  val appName␣
↪= properties.getProperty("name") ?: error("App name not found in $featureKFFilePath")
  val appVersion =␣
↪properties.getProperty("version") ?: error("App version not found in $featureKFFilePath")

  println("Deploying app $appName $appVersion to board at $boardIP:$boardPort")
  val url = "http:/
↪/$boardIP:$boardPort/api/app/install?force=$force&start=$start&name=$appName"
  val client = OkHttpClient()
  val multipartBody = MultipartBody.Builder().setType(MultipartBody.FORM) //
    .addFormDataPart(
    "binary",
    applicationFOFile.name,
    applicationFOFile.asRequestBody("application/octet-stream".toMediaType())
```

(continues on next page)

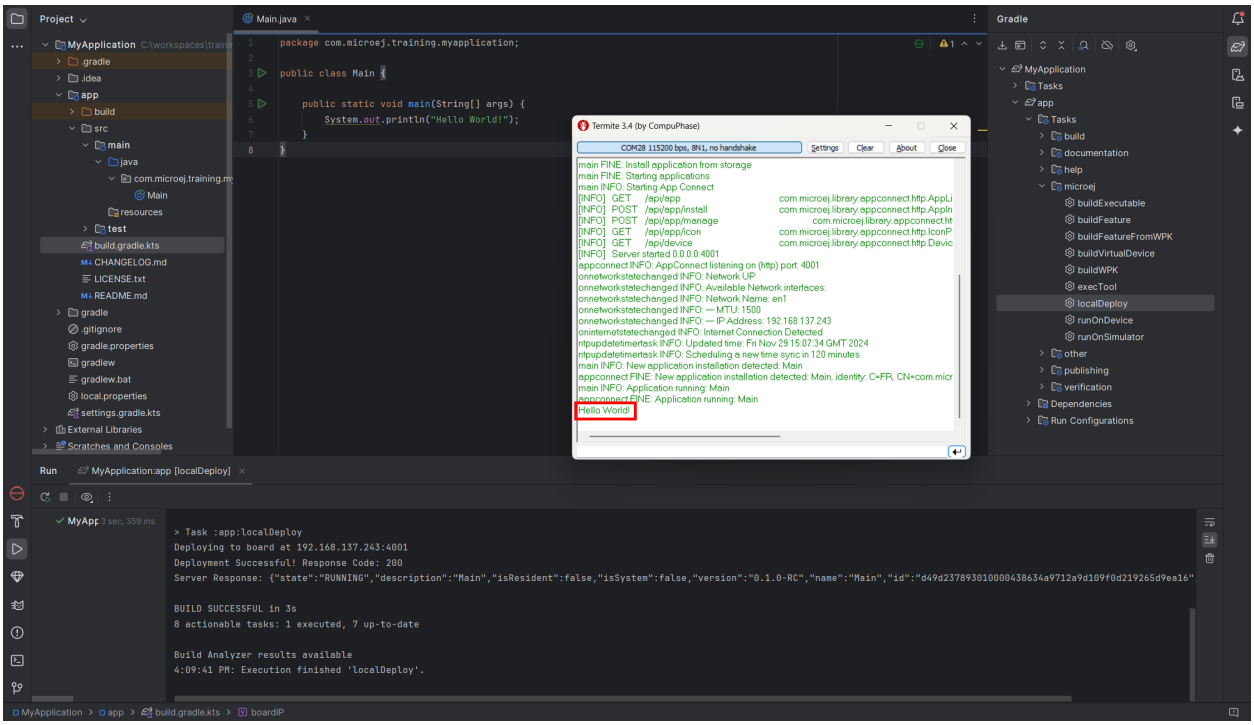
(continued from previous page)

```
        )//
        .build()
    val request = Request.Builder().url(url).post(multipartBody).build()
    client.newCall(request).execute().use { response ->
        if (response.isSuccessful) {
        println("Deployment Successful! Response Code: ${response.code}")
        println("App info: ${response.body?.string()}")
        } else {
        System.err.println("Deployment Failed. Response Code: ${response.code}")
        System.err.println("Cause: ${response.body?.string()}")
        }
    }
}
}

buildscript {
repositories {
    maven {
        name = "mavenCentral"
        url = uri("https://repo.maven.apache.org/maven2/")
    }
}
dependencies {
    classpath("com.squareup.okhttp3:okhttp:4.12.0")
}
}
```

- Update the `boardIP` variable with your board IP address.
- Reload the Gradle project:



- A `localDeploy` task is now visible in the `microej` tasks list.
- Double-click on the `localDeploy` task to deploy `MyApplication` on the board.
- `MyApplication` is successfully deployed and the `Hello World!` is displayed in the serial terminal:



Note: If you update your application, just run the `localDeploy` task again to test the updated application on your board!

Well Done!

Now you know how to create a Sandboxed Application project from scratch and run it on your device!

Going Further

You have now successfully executed Sandboxed Applications on an embedded device, so what's next?

If you are a Kernel Developer, you can follow the *For Kernel Developers* courses to get familiar with Kernel development.

If you are an application developer, you can continue to explore MicroEJ's APIs and functionalities by running and studying our samples at GitHub:

Foundation Libraries	Eclasspath	IoT
This project gathers all the basic examples of the foundation libraries.	This project gathers all the examples of eclasspath.	This project gathers simple applications using net libraries.
https://github.com/MicroEJ/Example-Foundation-Libraries	https://github.com/MicroEJ/Example-Eclasspath	https://github.com/MicroEJ/Example-IOT

You can also learn how to build bigger and better applications by reading our *Application Developer Guide*.

If you are an embedded engineer, you could look at our VEE port examples at GitHub. And to learn how create custom VEE ports, you can read our *VEE Porting Guide*.

You can also follow the *Kernel Developer Guide* for more information on our multi-application framework or read about our powerful wearable solution called *VEE Wear*.

Last but not least, you can choose to learn about specific topics by following one of our many *Training Courses* ranging from how to easily debug applications to setting up a Continuous Integration process and a lot of things in between.

MicroEJ Java Programming Practices

Description

This training describes some rules and tools aimed at improving the quality of a Java code to simplify its maintenance. It makes up a minimum consistent set of rules which can be applied in any situation, especially on embedded systems where performance and low memory footprint matter. Also be aware of MicroEJ runtime specificities by reading the *Language* page.

Intended Audience

This training is designed for Application developers who want to enhance the quality of their Java code.

Prerequisites

To get the most out of this training, participants should have a good knowledge of Java programming language.

Readable Code

This section describes rules to get a readable code, in order to facilitate:

- the maintenance of an existing code with multiple developers contributions (e.g., merge conflicts, reviews).
- the landing to a new code base when the same rules are applied across different modules and components.

Naming Convention

Naming of Java elements (package, class, method, field and local) follows the Camel Case convention.

- Package names are written fully in lower case (no underscore).
- Package names are singular (e.g. `ej.animal` instead of `ej.animals`).
- Class names are written in upper camel case.
- Interfaces are named in the same way as classes (see below).
- Method and instance field names are written in lower camel case.
- Static field names are written in lower camel case.
- Constant names are written in fully upper case with underscore as word separator.
- Enum constant names are written in fully upper case with underscores as word separators.
- Local (and parameter) names are written in lower camel case.
- When a name contains an acronym, capitalize only the first letter of the acronym (e.g. for a local with the `HTTP` acronym, use `myHttpContext` instead of `myHTTPContext`).

  It is also recommended to use full words instead of abbreviations (e.g. `MyProxyReference` instead of `MyProxyRef`).

Interfaces and Subclasses Naming Convention

An Interface is named after the feature it exposes. It does not have a `I` prefix because it hurts readability and may cause naming issues when potentially converted to/from an abstract class.

The classes implementing an interface are named after the interface and how they implement it. Using `Impl` suffix is not recommended because it does not express the implementation specificity. If there is no specificity, maybe there is no need to have an interface.

Example: an interface `Storage` (that allows to load/store data) may have several implementations, such as `StorageFs` (on a file system), `StorageDb` (on a database), `StorageRam` (volatile storage in RAM).

Visibility

Here is a list of the usage of each Java element visibility:

- `public` : API.
- `protected` : API for subclasses.
- `package` : component intern API (collaboration inside a package).
- `private` : internal structure, cache, lazy, etc.

By default, all instance fields must be private.

Package visibility can be used by writing the comment `/* package */` in place of the modifier.

Javadoc

Javadoc comments convention is based on the official documentation.

Note: Javadoc is written in HTML format and doesn't accept XHTML format: tags must not be closed. For example, use only a `<p>` between two paragraphs.

Here is a list of the rules to follow when writing Javadoc:

- All APIs (see *Visibility*) must have a full Javadoc (classes, methods, and fields).
- Add a dot at the end of all phrases.
- Add `@since` tag when introducing a new API.
- Do not hesitate to use links to help the user to navigate in the API (`@see` , `@link`).
- Use the `@code` tag in the following cases:
  - For keywords (e.g. `{@code null}` or `{@code true}`).
  - For names and types (e.g. `{@code x}` or `{@code Integer}`).
  - For code example (e.g. `{@code new Integer(Integer.parseInt(s))}`).

Here is a list of additional rules for methods:

- The first sentence starts with the third person (as if there is *This method* before).
- All parameters and returned values must be described.

- Put as much as possible information in the description, keep `@param` and `@return` minimal.
- Start `@param` with a lower case and usually with *the* or *a*.
- Start `@return` with a lower case as if the sentence starts with *Returns*.
- Avoid naming parameters anywhere other than in `@param`. If the parameter is renamed afterward, the comment is not changed automatically. Prefer using *the given xxx*.

Code Convention

Official documentation: https://www.oracle.com/java/technologies/javase/codeconventions-introduction.html

Class Declaration

The parts of a class or interface declaration must appear in the order suggested by the Code Convention for the Java Programming Language:

- Constants.
- Class (static) fields.
- Instance fields.
- Constructors
- Methods

Fields Order

For a better readability, the fields (class and instance) must be ordered by scope:

1. `public`
2. `protected`
3. `package`
4. `private`

Methods Order

It is recommended to group related methods together. It helps for maintenance:

- when searching for a bug on a specific feature,
- when refactoring a class into several ones.

Modifiers Order

Class and member modifiers, when present, appear in the order recommended by the Java Language Specification:

public protected private abstract default static final transient volatile synchronized native strictfp

Code Style and Formatting

MicroEJ defines a formatting profile for .java files, which is automatically set up when creating a new *Module Project Skeleton*.

Note: MicroEJ SDK automatically applies formatting when a .java file is saved. It is also possible to manually apply formatting on specific files:

- In Package Explorer, select the desired files, folders or projects,
- then go to Source > Format. The processed files must not have any warning or error.

Here is the list of formatting rules included in this profile:

- Indentation is done with 1 tab.
- Braces are mandatory with if, else, for, do, and while statements, even when the body is empty or contains only a single statement.
- Braces follow the Kernighan and Ritchie style (Egyptian brackets) described below:
  - No line break before the opening brace.
  - Line break after the opening brace.
  - Line break before the closing brace.
  - Line break after the closing brace, only if that brace terminates a statement or terminates the body of a method, constructor, or named class. For example, there is no line break after the brace if it is followed by else or a comma.
- One statement per line.
- Let the formatter automatically wraps your code when a statement needs to be wrapped.

Here is a list of additional formatting rules that are not automatically applied:

- Avoid committing commented code (other than to explain an optimization).
- All methods of an interface are public. There is no need to specify the visibility (easier to read).

Note: Most of these rules are checked by *Code Analysis with SonarQube™*.

Best Practices

This section describes rules made of best practices, well-known restrictions of the Java Programming Language, and more generally Object Oriented paradigm. Due to the resource constraints related to CPU, RAM, or FLASH usage, some Java best practices can be counterproductive when used in an embedded software development context. This section also exists to discuss such limitations. Be also aware that there is no absolute truth when talking about these limitations, you should keep in mind that depending on your hardware some may apply some may not.

Common Pitfalls

- Object.equals(Object) and Object.hashCode() methods must be overridden in pairs. See *Equals and Hashcode*.
    - Do not assign fields in field declaration but in the constructor.
    - Do not use non-final method inside the constructor.
    - Do not overburden the constructor with logic.
    - Do not directly store an array given by parameter.
    - Save object reference from a field to a local before using it (see *Local Extraction*).

Simplify Maintenance

- Extract constants instead of using magic numbers.
- Use parenthesis for complex operation series; it simplifies the understanding of operator priorities.
- Write short lines. This can be achieved by extracting locals (see *Local Extraction*).
- Use a limited number of parameters in methods (or perhaps a new type is needed).
- Create small methods with little complexity. When a method gets too complex, it should be split.
- Use $+$ operator only for single-line string concatenation. Use an explicit StringBuilder otherwise.
- Use component-oriented architecture to separate concerns. If a class is intended to be instantiated using Class.newInstance(), add a default constructor (without parameters).

Basic Optimizations

- Avoid explicitly initializing fields to 0 or null, because they are zero-initialized by the runtime. A $//\mathrm{VM_DONE}$ comment can be written to understand the optimization.
- Avoid using built-in thread safe types (Vector, Hashtable, StringBuffer, etc.). Usually synchronization has to be done at a higher level.
- Avoid serializing/deserializing data from byte arrays using manual bitwise operations, use ByteArray utility methods instead.

Local Extraction

Local extraction consists of storing the result of an expression before using it, for example:

```
Object myLocale = this.myField;
if (myLocale != null) {
  myLocale.myMethod();
}
```

It improves the Java code in many ways:

- self documentation: gives a name to a computed result.
- performance and memory footprint: avoids repeated access to same elements and extract loop invariants.
- thread safety: helps to avoid synchronization issues or falling into unwanted race conditions.
- code pattern detection: helps automated tools such as Null Analysis.

Equals and Hashcode

The purpose of these methods is to uniquely and consistently identify objects. The most common use of these methods is to compare instances in collections (list or set elements, map keys, etc.).

The Object.equals(Object) method implements an equivalence relation (defined in the Javadoc) with the following properties:

- It is reflexive: for any reference value x, `x.equals(x)` must return `true`.
- It is symmetric: for any reference values x and y, `x.equals(y)` must return `true` if and only if `y.equals(x)` returns `true`.
- It is transitive: for any reference values x, y, and z, if `x.equals(y)` returns `true` and `y.equals(z)` returns `true`, then `x.equals(z)` must return `true`.
- It is consistent: for any reference values x and y, multiple invocations of `x.equals(y)` consistently return `true` or consistently return `false`, provided no information used in equals comparisons on the object is modified.
- For any non-null reference value x, `x.equals(null)` must return `false`.

  Avoid overriding the `equals(Object)` method in a subclass of a class that already overrides it; it could break the contract above. See *Effective Java* book by *Joshua Bloch* for more information.

  If the `equals(Object)` method is implemented, the `hashCode()` method must also be implemented. The `hashCode()` method follows these rules (defined in the Javadoc):

- It must consistently return the same integer when invoked several times.
- If two objects are equal according to the `equals(Object)` method, then calling the `hashCode()` method on each of the two objects must produce the same integer result.
- In the same way, it should return distinct integers for distinct objects.

  The `equals(Object)` method is written that way:

- Compare the argument with `this` using the `==` operator. If both are equals, return `true`. This test is for performance purposes, so it is optional and may be removed if the object has few fields.

- Use an `instanceof` to check if the argument has the correct type. If not, return `false`. This check also validates that the argument is not null.
- Cast the argument to the correct type.
- For each field, check if that field is equal to the same field in the casted argument. Return `true` if all fields are equal, `false` otherwise.

```
@Override
public boolean equals(Object o) {
  if (o == this) {
    return true;
  }
  if (!(o instanceof MyClass)) {
    return false;
  }
  MyClass other = (MyClass)o;
  return field1 == other.field1 &&
    (field2 == null ? other.field2 == null : field2.equals(other.field2));
}
```

The goal of the Object.hashCode() is to produce different values for unequal objects. A good hashcode is uniformly distributed among hash buckets (for instance in HashMap, HashSet, etc.)

The `hashCode()` method is written that way:

- Choose any prime number such as `31` (that is large enough so that the number of buckets is unlikely to be divisible by it) or a bigger one.
- Create a result local intialized with the hashcode of the most significant field.
- For each remaining field, multiply the previous result with the prime plus the hash code of the field and store it as the result.
- Return the result.
- Only the fields used in `equals()` must be used.
- Derivative fields, that are computed from fields already included in computing of `hashCode()` can be ignored.
- Precomputing the hashcode may be convenient for performance purpose (especially when fields are final).
- The hashcode can also be lazy initialized the first time it is requested.

Depending on its type, the hash code of a field is:

- Boolean: `(f ? 1231 : 1237)`.
- Byte, char, short, int: `(int) f`.
- Long: `(int)(f ^ (f >>> 32))`.
- Float: `Float.floatToIntBits(f)`.
- Double: `Double.doubleToLongBits(f)` and the same as for a long.
- Object: `(f == null ? 0 : f.hashCode())`.
- Array: add the hash codes of all its elements (depending on their type).
- The hashcode of a null field is 0.

```
private static final int PRIME = 31;

@Override
public int hashCode() {
  int result = field0;
  result = PRIME * result + (field1 ? -1 : 1);
  result = PRIME * result + (field2 == null ? 0 : field2.hashCode());
  return result;
}
```

- Prefer using “foo”.equals(string) to avoid potential null accesses.

```
String s = null;
// Null safe
"foo".equals(s);
// NullPointerException
s.equals("foo");
```

Autoboxing and Numbers

- Avoid using boxed primitives (such as Integer, Byte, Float classes) if not needed. Most of the time using boxed primitives leads to autoboxing (the process of converting primitives to boxed primitives and the other way around), which can be CPU intensive due to casting.

```
// Boxed primitive type example
Integer boxedInteger = Integer.valueOf(5);
// Primitive basetype
int unboxedInteger = boxedInteger.intValue();
// Autoboxing example
List<Integer> integerList = new ArrayList<>();
// Here you "autobox" the basetype into its corresponding primitive type
list.add(5);
```

- Prefer 32-bit floats for embedded performance. Double operations are more CPU intensive.

Generic Types

- Do not use parameterized types as raw types such as using the Collection without specifying the type parameter, prefer using a parameterized type as designed (it ensures type safety, avoid explicit type casting, and improve code readability). Generic and parameterized types are a compile time feature, it won’t impact runtime performances and memory footprint.

```
// Prefer
ArrayList<Foo> paramList = new ArrayList<>();
paramList.add(new Foo("I'm foo!"));

// Compiler will trigger an error if you try to add a wrong type here
paramList.add(new Bar("I'm bar!"));

// Over
ArrayList list = new ArrayList();
list.add(new Foo("I'm another foo!"));
```

Memory Use of Objects

- The Java bytecode specification defines a 32-bit operand stack model. Declaring local variables with types that require fewer bits (byte, short, char) results in additional conversion and casting instructions during execution. However this is not applicable when declaring Java instance fields, which are optimized for size in the internal Java object structure. The organization of fields in memory is left to the runtime implementation.
- Operations on local variables in Java are happening using the thread's own stack (by loading and storing values onto the stack). Local variables are tied to their scope/context usually their associated method. Objects are stored in Managed Heap.
- *Memory Considerations* and *Limitations* are also documentation pages that describe the memory use of Objects and the limitations of the MicroEJ runtime.
- You rarely need to trigger a Garbage Collection (GC) manually through System.gc(). A use case example that would require a manual GC trigger is when you need an accurate memory usage of the Managed Heap (before a call to Runtime.getRuntime().freeMemory()).
- Prefer using an array for fixed memory usage against dynamic data structure. If you do not need the convenience of dynamically allocated types, it is most of the time more efficient (CPU wise) to use arrays. Dynamical allocated types such as collection types tend to check for size and have mechanisms to enlarge on-the-fly the data structure. Using an array prevent that but obviously you keep the runtime checks.

```
// Prefer
int[] array = new int[size];

// Over (when applicable)
ArrayList<Integer> arrayList = new ArrayList<>();
```

- Try calibrating data structure by giving it a size at initialization (avoid automatically enlarging them when needed).

```
// Try initializing the ArrayList with a known size
Collection<String> colors = new ArrayList<String>(500);
```

- To use the cloning mechanism provided by Java, here are the rules to respect:
  - Always implement `Cloneable`.
  - `bar.clone() != bar` is `True`.
  - `bar.clone().getClass() == bar.getClass()` is `True`.
  - `bar.clone().equals(bar)` is `True`.
  - Use deep copies for your implementation of `.clone()` over shallow copies. Shallow copies mean clones are tied to their original instance.

```
// Prefer
@Override
protected Object clone() throws CloneNotSupportedException {
    Bar newClone = (Bar) super.clone();
    newClone.setField(newClone.getField().clone());
    return newClone;
}

// Over
```

(continues on next page)

(continued from previous page)

```
@Override
protected Object clone() throws CloneNotSupportedException {
    return super.clone();
}
```

- For specific memory size optimizations, see *Optimize the Memory Footprint of an Application*.

Reflection

- MicroEJ does not embed the fully qualified name of all classes in the final binary. As such you need to explicitly specify which type names to embed using `*.types.list` files (see *Types*).
- Java reflection forces to embed the fully qualified name of Java elements. As such it can be costly in persistent memory. MicroEJ has made the choice to only allow Class.forName(), Class.getName(), Class.getSimpleName(), and Class.newInstance() methods from the reflection framework.

BON Constants

- Consider using BON constants, they allow for sections of code to not be embedded in the final binary depending on the constant value. Constants are resolved at binary level without having to recompile the sources. More information can be found at this *Constants* section of the documentation.
- BON constants are preferred to System properties for the following reasons:
  - BON constants are compile time checked whereas System properties are often used with runtime checks.
  - System properties allow only String values (meaning String comparison most of the time).
  - System properties checks do not allow to completely remove the code from the binary (so they are more costly in code memory space).

Enums

- Avoid Enum types in your code, use int constants when possible. Enum types are costly at runtime.

Concurrency

- Do not implement applications that expect a behavior of the underlying task scheduler. Make your synchronization between threads explicit.
- Best pratices for synchronization:
  - Small exclusion zones, large exclusion zones usually means thread wait longer.
  - Use Executors.
  - For the use of explicit synchronization and use of monitors, you can consult this article.

- There is no explicit way to kill a Java thread. A well designed thread that is long running checks for interrupts at regular intervals and acts on interrupt signals. More information can be found here.

Serialization

- The "native" serialization of the standard JVM is not implemented by MicroEJ. This mechanism has historically introduced numerous compatibility issues and has since been officially deprecated. Synchronization and serialization should be handled at the application level. See data formats commonly used with MICROEJ VEE: *Data Serialization*.

Annotations

- MicroEJ supports only compile-time annotations. The usual annotations we encourage to use are `@Override`, the Null Analysis annotations, and `@Deprecated`.
- Another typical use case of annotations is for declaring JUnit tests. See *Test a Project* for more information.
- You can also define your custom annotations in conjonction with add-on processors.

Polymorphism, Inheritance, and Interfaces

- Prefer interfaces to abstract classes for the following reasons:
  - it easily integrates with existing classes, add the `implements` to existing classes, it is harder to do with abstract classes,
  - interfaces allow the easy notion of mixin,
  - interfaces allow for the creation nonhierarchical types.
- The SOAR tries to make method calls direct as much as it can, see *Method Devirtualization* for more information.

Exceptions

Here are in no particular order best pratices around managing exceptions in Java:

- Use existing exceptions for your API, e.g., there is no need to create a `MyModelOptionException` when IllegalArgumentException exists.
- Use checked exceptions for recoverable errors, use unchecked exceptions for programming errors or code violations.
  - Checked exceptions allows to complete your API with its exceptional conditions.
  - Unchecked exceptions are throwables such as errors and runtime exceptions, they usually indicate a violation of some fundamental rules of Java (such as ArrayIndexOutOfBoundsException).
  - It is a good pratice to have your custom unchecked exceptions to extend RuntimeException.
  - Do not use unchecked exceptions to not be bothered using `throws` in your methods.
- If you want an "undying" thread, you should catch all Throwable.
- Avoid exception masking (e.g., doing nothing in a catch clause).

```
// Do not do this
try{
  // Some code causing an Exception
} catch (Exception e){
  //  You should do something here
}

//Prefer
try{
  // Some code causing an Exception
} catch (Exception e){
  // You could do log it
  logger.log(Level.SEVERE, "Severe error message");
  // or you could rethrow it, by tweaking the exceptional type
  throw new MyException(e);
}
```

- It is a good practice to set your custom Thread.UncaughtExceptionHandler to improve the robustness of your application. It could set per thread or at application level.

```
public class MyHandler implements Thread.UncaughtExceptionHandler {

  public void uncaughtException(Thread thread, Throwable e) {
    // Process what to do
    logger.log(Level.SEVERE, "Uncaught exception: " + e.getMessage());
    e.printStackTrace();
  }

}
```

- Automatically close resources using try-with-resources.
- For more information on Exception as well as a hierarchy of common exceptions please read this article.
- For information on how errors are handled in the native functions, please refer to *Error Handling in Native Methods* chapter.

Data Encapsulation and Fields

- Keep your fields private by default.
- Provide field getters and setters when needed. - Do not directly return an internal array or an internal non-immutable Object. Once returned the caller could modify "your" instance without warning or synchronization.
- Use `final` for public basetype fields because:
  - By default it forces fields to be read-only.
  - It ensures thread safety.
  - It forces you to consider if the field should have right access and communicate intent to other developers.

Native Interfaces

- Several mechanisms exist to communicate between managed and native worlds. Find more information at *Native Interface Mechanisms*.

Usage of Inner Classes

- Prefer static inner classes when needed because there is a performance impact on accessing the outer class instance.
- Non static inner classes keep a reference to an instance of the outer class.

```
public class OuterClass {

   // Avoid non-static inner class (an instance of this class is stored in the outer class)
   private class InnerClass1 {
      public void message() {
         System.out.println("This is a non-static inner class.");
      }
   }

   // Prefer static inner class (the instances are shared among all instances of OuterClass)
   public static class InnerClass2 {
      public void message() {
         System.out.println("This is a static inner class.");
      }
   }
}
```

- Prefer short inner classes for readability (if your inner class gets too complex it surely deserves its own file).

Usage of Clinits

- The clinit order is done statically by the SOAR before the execution, as such clinits shall be limited to class internal constant initialization, with as less as possible dependency. *Class Initialization Code* describes how MicroEJ deals with class initialization.

About Limitations

- For a deeper look at what is allowed in terms of number of fields or methods in a class, maximum number of threads and more: please consult *Limitations*.

Inlining

- For better CPU performance at runtime, the SOAR implements some inlining techniques more information at *Method Inlining*.

Binary creation from classpath

- Not all files found in the classpath are embedded in a MicroEJ Application, to manage embedded resources consult *Application Resources*.
- In the same philosophy the SOAR does not embed every unused types from the classpath in the final binary. More information at *MicroEJ Classpath*.
- The SOAR also strips the unused methods from the code.

Immutables and Immortals

- MICROEJ VEE defines two additional categories of objects: Immutables (objects that cannot change) and Immortals (objects that cannot be garbage-collected). More information below.

  – *Persistent Immutable Objects*

  – *Immortal Objects*

Loop Invariants

- Avoid unnecessary operations in loop (e.g., accessing a Collection size if not changing, accessing fields, etc.), consider using primitive types for loop variables, and minimize object creation.

```
// Prefer␣
↪"caching" class fields in a local variable when it does not depend on the loop operations
int localNumberToUse = this.numberToUse;
for (int i = 0; i < 10000; i++) {
   result = thingsToMultiply * localNumberToUse
}

// Prefer accessing Collection size outside of a loop
Collection<String> colors = new ArrayList<>();
colors.add("Red");
colors.add("Green");
colors.add("Blue");

// Retrieve the size only once
int size = colors.size();
for (String color : colors) {
   // Cheaper access for each loop
   System.out.println(color + " is a color in an ArrayList of " + size + " colors.");
}
```

- A *foreach* loop is a shorter way to write a loop over collections or arrays. It eliminates the need for explicit indexing and provides better readability.

```
int[] scores = {90, 85, 95, 88};

for (int score : scores) {
  System.out.println("Score: " + score);
}
```

Use of I/O Classes

- Be mindful of the use of IO classes and their buffered version. While buffered types such as BufferedInputStream are classes that improve the performance of input/output operations by reducing the number of I/O calls, these types do it by consuming more memory.

Logging

- Use BON constants to enable and disable logging traces in your code to conserve ROM space, see *Constants*.
- Use Logger over System.out.println.

Array Copy

- When doing memory transfers on arrays use System.arraycopy() when possible as it is optimized to run nearly as fast as a native $\mathrm{memmove}$.

Switch Statements

- Try to optimize your switch statement with contiguous case values resulting in a faster implementation.
- The switch/case statements are generated by the Java compiler in two ways depending on the cases density. Prefer declaring consecutive cases (*table_switch*) for performance ($\mathrm{O}(1)$) and slightly smaller code memory footprint instead of *lookup_switch* ($\mathrm{O}(\log\ \mathrm{N})$).

Related Tools

This section points to tools aimed at helping to improve code quality.

Unit Testing

Here is a list of rules when writing tests (see *Application Testing*):

- Prefer black-box tests (with a maximum coverage).
- Here is the test packages naming convention:
  - Suffix package with .test for black-box tests.
  - Use the same package for white-box tests (allow to use classes with package visibility).

Code Analysis with SonarQube™

SonarQube is an open source platform for continuous inspection of code quality. SonarQube offers reports on duplicated code, coding standards, unit tests, code coverage, code complexity, potential bugs, comments, and architecture.

To set it up on your MicroEJ application project, please refer to this documentation. It describes the following steps:

- How to run a SonarQube server locally.
- How to run an analysis using a dedicated script.
- How to run an analysis during a module build.

Code Instrumentation

We have tools allow the profiling and logging of Java code behavior, please refer to *Code Instrumentation for Logging*.

Sandboxed Application Development

Description

In this training, you will get an introduction to Application development in a Multi-Sandbox context.

What you will learn:

- Sandboxed Applications Concepts: from creation to deployment.
- Inter Application Communication: focus on Shared interface.

Intended Audience

This training is designed for developers who want to gain a first understanding of Application development in a Multi-Sandbox context.

Prerequisites

To get the most out of this training, participants should have access to:

- The STM32F7508-DK Evaluation Kit.
- A mini USB cable.
- A micro SD card.
- An ethernet cable.

Training Resources

Unit 1: Sandboxed Application Training

This training is provided as a PDF:

MICROEJ SDK 5

MICROEJ SDK 6

-

Coming soon!

Unit 2: Run Demo Sandboxed Application

Run the Demo Sandboxed Application on an Evaluation Kit. This demo showcases the communication between Sandboxed Applications using the Shared Interfaces API:

MICROEJ SDK 6

MICROEJ SDK 5

- Demo Sandboxed Application
- Demo Sandboxed Application

Optimize the Memory Footprint of an Application

Description

This training explains how to analyze the memory footprint of an application and provides a set of common rules aimed at optimizing both ROM and RAM footprint.

Intended Audience

This training is designed for Java engineers and Firmware integrators who want to execute a MicroEJ Application on a memory-constrained device.

Prerequisites

To get the most out of this training, participants should have a good knowledge of Java programming language.

Introduction

Usually, the application development is already started when the developer starts thinking about its memory footprint. Before jumping into code optimizations, it is recommended to list every area of improvement and estimate for each area how much memory can be saved and how much effort it requires.

Without performing the memory analysis first, the developer might start working on a minor optimization that takes a lot of effort for little improvements. In contrast, he could work on a major optimization, allowing faster and bigger improvements. Moreover, each optimization described hereafter may allow significant memory savings for an application while it may not be relevant for another application.

How to Analyze the Footprint of an Application

This section explains the process of analyzing the footprint of a MicroEJ Application and the tools used during the analysis.

Suggested footprint analysis process:

1. Build the MicroEJ Application
2. Analyze $\mathrm{SOAR.map}$ with the *Memory Map Analyzer*
3. Analyze $\mathrm{soar/*.xml}$ with an XML editor
4. Link the MicroEJ Application with the BSP
5. Analyze the map file generated by the third-party linker with a text editor

Footprint analysis tools:

- The *Memory Map Analyzer* allows to analyze the memory consumption of different features in the RAM and ROM.
- The *Heap Dumper & Heap Analyzer* allow to understand the contents of the Managed Heap and find problems such as memory leaks.
- The API Dependency Discoverer allows to analyze a piece of code to detect all its dependencies.

How to Analyze the Files Generated by the MicroEJ Linker

The MicroEJ Application linker generates files useful for footprint analysis, such as the SOAR map file and the SOAR information file. To understand how to read these files, please refer to the *SOAR Output Files* documentation.

How to Analyze a Map File Generated by a Third-Party Linker

A $<\mathrm{firmware}>\mathrm{.map}$ file is generated by the C toolchain after linking the MicroEJ Application with the BSP. This section explains how a map file generated by GCC is structured and how to browse it. The structure is not the same on every compiler, but it is often similar.

File Structure

This file is composed of 5 parts:

- Archive member included to satisfy reference by file . Each entry contains two lines. The first line contains the referenced archive file location and the compilation unit. The second line contains the compilation unit referencing the archive and the symbol called.
- Allocating common symbols . Each entry contains the name of a global variable, its size, and the compilation unit where it is defined.
- Discarded input sections . Each entry contains the name and the size of a section that has not been embedded in the firmware.
- Memory Configuration . Each entry contains the name of a memory, its address, its size, and its attributes.
- Linker script and memory map . Each entry contains a line with the name and compilation unit of a section and one line per symbol defined in this section. Each of these lines contains the name, the address, and the size of the symbol.

Finding the Size of a Section or Symbol

For example, to know the thread stacks' size, search for the .bss.vm.stacks.java section in the Linker script and memory map part. The size associated with the compilation unit is the size used by the thread stacks.

The following snippet shows that the .bss.vm.stacks.java section takes 0x800 bytes.

```
.bss.vm.stacks.java
                0x20014070␣
↪       0x800 ..\..\..\..\..\..\..\.microej\CM4hardfp_GCC48\application\microejapp.o
                0x20014070                ___icetea_____6bss_6vm_6stacks_6java$$Base
                0x20014870                ___icetea_____6bss_6vm_6stacks_6java$$Limit
```

See *Core Engine Link* documentation for more information on Core Engine sections.

How to Reduce the Image Size of an Application

Generic coding rules can be found in the following training: *MicroEJ Java Programming Practices*.

This section provides additional coding rules and good practices to reduce the image size (ROM) of an application.

Application Resources

Resources such as images and fonts take a lot of memory. For every .list file, make sure that it does not embed any unused resource.

Only resources declared in a .list file will be embedded. Other resources available in the *application classpath* will not be embedded and will not have an impact on the application footprint.

Fonts

Default Font

By default, in a *MicroEJ Platform configuration* project, a so-called system font is declared in the `microui.xml` file.

When generating the MicroEJ Platform, this file is copied from the configuration project to the actual MicroEJ Platform project. It will later be converted to binary format and linked with your MicroEJ Application, even if you use fonts different from the system font.

Therefore, you can comment the system font from the `microui.xml` file to reduce the ROM footprint of your MicroEJ Application if this one does not rely on the system font. Note that you will need to rebuild the MicroEJ Platform and then the application to benefit from the footprint reduction.

See the *Display Element* section of the *Static Initialization* documentation for more information on system fonts.

Character Ranges

When creating a font, you can reduce the list of characters embedded in the font at several development stages:

- On font creation: see the *Removing Unused Characters* section of *Font Designer* documentation.
- On application build: see the *Fonts* section of *MicroEJ Classpath* documentation.

Pixel Transparency

You can also make sure that the BPP encoding used to achieve transparency for your fonts do not exceed the following values:

- The pixel depth of your display device.
- The required alpha level for a good rendering of your font in the application.

  See the *Fonts* section of *MicroEJ Classpath* documentation for more information on how to achieve that.

External Storage

To save storage on internal flash, you can access fonts from an external storage device.

See the *External Resources* section of the *Font Generator* documentation for more information on how to achieve that.

Internationalization Data

Implementation

MicroEJ provides the *Native Language Support (NLS)* library to handle internationalization.

See https://github.com/MicroEJ/Example-NLS for an example of the use of the NLS library.

External Storage

The default NLS implementation fetches text resources from internal flash, but you can replace it with your own implementation to fetch them from another location.

See *External Resources Loader* documentation for additional information on external resources management.

Images

Encoding

If you are tight on ROM but have enough RAM and CPU power to decode PNG images on the fly, consider storing your images as PNG resources. If you are in the opposite configuration (lots of ROM, but little RAM and CPU power), consider storing your images in raw format.

See *Image Generator* documentation for more information on how to achieve that.

Color Depth (BPP)

Make sure to use images with a color depth not exceeding the one of your display to avoid the following issues:

- Waste of memory.
- Differences between the rendering on the target device and the original image resource.

External Storage

To save storage on internal flash, the application can access the images from an external storage device.

See *External Resources Loader* documentation for more information on how to achieve that.

Application Code

The following application code guidelines are recommended in order to minimize the size of the application:

- Check libraries versions and changelogs regularly. Latest versions may be more optimized.
- Avoid manipulating String objects:
  - For example, prefer using integers to represent IDs.
  - Avoid overriding Object.toString() for debugging purposes. This method will always be embedded even if it is not called explicitly.
  - Avoid using the *logging library* or `System.out.println()`, use the *trace library* or the *message library* instead. The logging library uses strings, while the trace and message libraries use integer-based error codes.
  - Avoid using the string concatenation operator (`+`), use an explicit StringBuilder instead. The code generated by the `+` operator is not optimal and is bigger than when using manual `StringBuilder` operations.
- Avoid manipulating wrappers such as Integer and Long objects, use primitive types instead. Such objects have to be allocated in Managed Heap memory and require additional code for boxing and unboxing.
- Avoid declaring Java Enumerations (`enum`), declare compile-time constants of primitives types instead (e.g. `static final int I = 0;`). The Java compiler creates an Enum object in the Managed Heap for each enumeration item, as well as complex class initialization code.
- Avoid using the service library, use singletons or Constants.getClass() instead. The service library requires embedding class reflection methods and the type names of both interfaces and implementations.
- Avoid using the Java Collections Framework. This OpenJDK standard library has not been designed for memory constrained devices.
  - Use raw arrays instead of List objects. The ArrayTools class provides utility methods for common array operations.
  - Use PackedMap objects instead of Map objects. It provides similar APIs and features with lower Managed Heap usage.
- Use ej.bon.Timer instead of deprecated `java.util.Timer`. When both class are used, almost all the code is embedded twice.
- Use *BON constants* in the following cases if possible:
  - when writing debug code or optional code, use the `if (Constants.getBoolean()) { ... }` pattern. That way, the optional code will not be embedded in the production firmware if the constant is set to `false`.
  - replace the use of *System Properties* by BON constants when both keys and values are known at compile-time. System Properties should be reserved for runtime lookup. Each property requires embedding its key and its value as intern strings.
- Check for useless or duplicate synchronization operations in call stacks, in order reduce the usage of `synchronized` statements. Each statement generates additional code to acquire and release the monitor.
- Avoid declaring exit statements (`break` , `continue` , `throw` or `return`) that jump out of a `synchronized` block. At each exit point, additional code is generated to release the monitor properly.

- Avoid declaring exit statements (break , continue , throw or return) that jump out of a try/finally block. At each exit point, the code of the finally block is generated (duplicated). This also applies on every try-with-resources block since a finally block is generated to close the resource properly.
- Avoid overriding Object.equals(Object) and Object.hashCode(), use == operator instead if it is sufficient. The *correct implementation of these methods* requires significant code.
- Avoid calling equals() and hashCode() methods directly on Object references. Otherwise, the method of every embedded class which overrides the method will be embedded.
- Avoid creating inlined anonymous objects (such as new Runnable() { ... } objects), implement the interface in a existing class instead. Indeed, a new class is created for each inlined object. Moreover, each enclosed final variable is added as a field of this anonymous class.
- Avoid accessing a private field of a nested class. The Java compiler will generate a dedicated method instead of a direct field access. This method is called *synthetic*, and is identified by its name prefix: access$.
- Replace constant arrays and objects initialization in static final fields by *immutables objects*. Indeed, initializing objects dynamically generates code which takes significant ROM and requires execution time.
- Check if some features available in software libraries are not already provided by the device hardware. For example, avoid using java.util.Calendar (full Gregorian calendar implementation) if the application only requires basic date manipulation provided by the internal real-time clock (RTC).

MicroEJ Platform Configuration

The following configuration guidelines are recommended in order to minimize the size of the application:

- Check MicroEJ Architecture and Packs versions and changelogs regularly. Latest versions may be more optimized.
- Configure the Platform to use the *tiny* capability of the Core Engine. It reduces application code size by ~20%, provided that the application code size is lower than 256KB (resources excluded).
- Disable unnecessary modules in the .platform file. For example, disable the Image PNG Decoder module if the application does not load PNG images at runtime.
- Don't embed unnecessary *pixel conversion* algorithms. This can save up to ~8KB of code size but it requires knowing the format of the resources used in the application.
- Select your embedded C compilation toolchain with care, prefer one which will allow low ROM footprint with optimal performance. Check the compiler options:
  - Check documentation for available optimization options (-Os on GCC). These options can also be overridden per source file.
  - Separate each function and data resource in a dedicated section (-ffunction-sections -fdata-sections on GCC).
- Check the linker optimization options. The linker command line can be found in the project settings, and it may be printed during link.
  - Only embed necessary sections (--gc-sections option on GCC/LD).

  - Some functions, such as the `printf` function, can be configured to only implement a subset of the public API (for example, remove `-u __printf_float` option on GCC/LD to disable printing floating point values).

- In the map file generated by the third-party linker, check that every embedded function is necessary. For example, hardware timers or HAL components may be initialized in the BSP but not used in the application. Also, debug functions such as *SystemView* may be disconnected when building the production firmware.

Application Configuration

The following application configuration guidelines are recommended in order to minimize the size of the application:

- Disable class names generation by setting the `soar.generate.classnames` option to `false` (default). Class names are only required when using Java reflection. In such case, the name of a specific class will be embedded only if is explicitly required. See *Stripping Class Names from an Application* section for more information.
- Remove UTF-8 encoding support by setting the `cldc.encoding.utf8.included` option to `false`. The default encoding (`ISO-8859-1`) is enough for most applications.
- Remove `SecurityManager` checks by setting the `com.microej.library.edc.securitymanager.enabled` option to `false`. This feature is only useful for Multi-Sandbox firmwares.

  For more information on how to set an option, please refer to the *Defining an Option with SDK 5 or lower* section.

Stripping Class Names from an Application

Embedding the name of every class is convenient when starting a new MicroEJ Application, but it is rarely necessary and takes a lot of ROM. This section explains how to embed only the required class names of an application.

Removing All Class Names

First, make sure to set the *Application option* `soar.generate.classnames` to `false` (default value).

For more information on how to set an option, please refer to the *Defining an Option with SDK 5 or lower* section.

Listing Required Class Names

Some class names may be required by an application to work properly. These class names must be explicitly specified in a `*.types.list` file.

The code of the application must be checked for all uses of the Class.forName(), Class.getName() and Class.getSimpleName() methods. For each of these method calls, if the class name if absolutely required and can not be known at compile-time, add it to a `*.types.list` file. Otherwise, remove the use of the class name.

The following sections illustrates this on concrete use cases.

Case of Service Library

The ej.service.ServiceLoader class of the service library is a dependency injection facility. It can be used to dynamically retrieve the implementation of a service.

The assignment between a service API and its implementation is done in *.properties.list files. Both the service class name and the implementation class name must be embedded (i.e., added in a *.types.list file).

For example:

```
# example.properties.list
com.example.MyService=com.example.MyServiceImpl
```

```
# example.types.list
com.example.MyService
com.example.MyServiceImpl
```

Case of Properties Loading

Some properties may be loaded by using the name of a class to determine the full name of the property. For example:

```
Integer.getInteger(MyClass.class.getName() + ".myproperty");
```

In this case, it can be replaced with the actual string. For example:

```
Integer.getInteger("com.example.MyClass.myproperty");
```

Case of Logger and Other Debugging Facilities

Logging mechanisms usually display the name of the classes in traces. It is not necessary to embed these class names. The *Stack Trace Reader* can decipher the output.

How to Reduce the Runtime Size of an Application

You can find generic coding rules in the following training: *MicroEJ Java Programming Practices*.

This section provides additional coding rules and good practices in order to reduce the runtime size (RAM) of an application.

Application Code

The following application code guidelines are recommended in order to minimize the size of the application:

- Avoid using the default constructor of collection objects, use constructors that allow to set the initial capacity. For example, use the ArrayList(int initialCapacity) constructor instead of the default one which will allocate space for ten elements.

- Adjust the type of `int` fields (32 bits) according to the expected range of values being stored (`byte` for 8 bits signed integers, `short` for 16 bits signed integers, `char` for 16 bits unsigned integers).
- When designing a generic and reusable component, allow the user to configure the size of any buffer allocated internally (either at runtime using a constructor parameter, or globally using a BON constant). That way, the user can select the optimal buffer size depending on his use-case and avoid wasting memory.
- Avoid allocating immortal arrays to call native methods, use regular arrays instead. Immortal arrays are never reclaimed and they are not necessary anymore when calling a native method.
- Reduce the maximum number of parallel threads. Each thread requires a dedicated internal structure and one or more stack blocks.
  - Avoid creating threads on the fly for asynchronous execution, use shared thread instances instead (ej.bon.Timer, Executor, MicroUI.callSerially(Runnable), ...).
- When designing Graphical User Interface:
  - Avoid creating mutable images (BufferedImage instances) to draw in them and render them later, render graphics directly on the display instead. Mutable images require allocating a lot of memory from the images heap.
  - Make sure that your Widget hierarchy is as flat as possible (avoid any unnecessary Container). Deep widget hierarchies take more memory and can reduce performance.

MicroEJ Platform Configuration

The following configuration guidelines are recommended in order to minimize the runtime size of the application:

- Check the size of the stack of each RTOS task. For example, 1.0KB may be enough for the Core Engine task but it can be increased to allow deep native calls. See *Debugging Stack Overflows* section for more information.
- Check the size of the heap allocated by the RTOS (for example, `configTOTAL_HEAP_SIZE` for FreeRTOS).
- Check that the size of the back buffer matches the size of the display. Use a *partial buffer* if the back buffer does not fit in the RAM.

Debugging Stack Overflows

If the size you allocate for a given RTOS task is too small, a stack overflow will occur. To be aware of stack overflows, proceed with the following steps when using FreeRTOS:

1. Enable the stack overflow check in `FreeRTOS.h` :

```
#define configCHECK_FOR_STACK_OVERFLOW 1
```

2. Define the hook function in any file of your project (`main.c` for example):

```
void vApplicationStackOverflowHook(TaskHandle_t xTask, signed char *pcTaskName) { }
```

3. Add a new breakpoint inside this function
4. When a stack overflow occurs, the execution will stop at this breakpoint

For further information, please refer to the FreeRTOS documentation.

Application Configuration

The following application configuration guidelines are recommended in order to minimize the size of the application.

For more information on how to set an option, please refer to the *Defining an Option with SDK 5 or lower* documentation.

Managed Heap and Immortals Heap

- Configure the *Immortals Heap* option to be as small as possible. You can get the minimum value by calling Immortals.freeMemory() after the creation of all the immortal objects.
- Configure the *Managed Heap* option to fit the needs of the application. You can get it by using the *Heap Usage Monitoring Tool*.

Thread Stacks

- Configure the *maximum number of threads* option. This number can be known accurately by counting in the code how many `Thread` and `Timer` objects may run concurrently. You can call Thread.getAllStackTraces() or Thread.activeCount() to know what threads are running at a given moment.
- Configure the *number of allocated thread stack blocks* option. This can be done empirically by starting with a low number of blocks and increasing this number as long as the application throws a `StackOverflowError` .
- Configure the *maximum number of blocks per thread* option. The best choice is to set it to the number of blocks required by the most greedy thread. Another acceptable option is to set it to the same value as the total number of allocated blocks.
- Configure the *maximum number of monitors per thread* option. This number can be known accurately by counting the number of concurrent `synchronized` blocks. This can also be done empirically by starting with a low number of monitors and increasing this number as long as no exception occurs. Either way, it is recommended to set a slightly higher value than calculated.

Core Engine Dump

The `LLMJVM_dump()` function declared in `LLMJVM.h` may be called to print information on alive threads such as their current and maximum stack block usage. This function may be called from the application by exposing it in a *native function*. See *Dump the State of the Core Engine* section for usage.

More specifically, the `Peak java threads count` value printed in the dump can be used to configure the maximum number of threads. The `max_java_stack` and `current_java_stack` values printed for each thread can be used to configure the number of stack blocks.

MicroUI Images Heap

- Configure the *images heap* to be as small as possible. You can compute the optimal size empirically. It can also be calculated accurately by adding the size of every image that may be stored in the images heap at a given moment. One way of doing this is to inspect every occurrence of BufferedImage() allocations and ResourceImage usage of loadImage() methods.

Application Testing

Description

In this training, you will learn the basics to develop and run unit tests on an application using the JUnit framework.

What you will learn:

- Basics of JUnit.
- How to create and run JUnit tests for an Add-on Library.

Intended Audience

This training is designed for Application developers who want to automate their tests.

Prerequisites

To get the most out of this training, participants should have:

- A basic knowledge of Java programming language.
- Access to the NXP i.MX RT1170 Evaluation Kit.

Training Resources

This training is provided as a PDF:

- (for MICROEJ SDK 6)

3.10.3 For VEE Port Developers

VEE Port Creation for a Custom Device

Description

In this training, you will follow the typical steps followed by a Firmware developer integrating MICROEJ VEE on a target device.

What you will do:

- Get a running “HelloWorld” in C.
- Install a MicroEJ Architecture that matches the target device/RTOS/toolchain.

- Create a VEE Port project.
- Integrate MICROEJ VEE in the BSP / RTOS.
- Create and Run a HelloWorld Application.
- Create Build and Run Scripts.

  No Evaluation kit is required to follow this training. Everything is emulated on the developer's PC.

Intended Audience

This training is designed for developers who want to gain a first understanding of MICROEJ VEE Port flow.

Prerequisites

To get the most out of this training, participants should have:

- A basic knowledge of C programming language.
- An experience in embedded software development.

Training Resources

Create a MicroEJ Firmware From Scratch

This training explains how to create a MicroEJ Firmware from scratch. It goes trough the typical steps followed by a Firmware developer integrating MicroEJ with a C Board Support Package (BSP) for a target device.

In this training, the target device is a a Luminary Micro Stellaris. Though this device is no longer available on the market, it has two advantages:

- The QEMU PC System emulator can emulate the device.
- FreeRTOS provides an official Demo BSP.

  Consequently, no board is required to follow this training. Everything is emulated on the developer's PC.

  The training should take 1 hour to complete (excluding the installation time of MicroEJ SDK and Windows Subsystem Linux (WSL)).

Intended Audience

The audience for this document is Firmware engineers who want to understand how MicroEJ is integrated to a C Board Support Package.

In addition, this training should be of interest to all developers wishing to familiarize themselves with the low level components of a MicroEJ Firmware such as: *MicroEJ Architecture*, *MicroEJ Platform*, *Low Level API* and *BSP connection*.

Introduction

The following steps are usually followed when starting a new project:

1. Pick a target device (that meets the requirements of the project).
2. Setup a RTOS and a toolchain that support the target device.
3. Adapt the RTOS port if needed.
4. Install a *MicroEJ Architecture* that matches the target device/RTOS/toolchain.
5. Setup a new *MicroEJ Platform* connected to the Board Support Package (BSP).
6. Implement *Low Level API*.
7. Validate the resulting MicroEJ Platform with the Platform Qualification Tools (PQT).
8. Develop the *MicroEJ Application*.

This training describes step by step how to go from the FreeRTOS BSP to a MicroEJ Application that runs on the MicroEJ Platform and prints the classic "Hello, World!".

In this training:

- The target device is a Luminary Micro Stellaris which is emulated by QEMU (QEMU Stellaris boards).
- The RTOS is FreeRTOS and the toolchain is GNU CC fo ARM.

All modifications to FreeRTOS BSP made for this training are available at https://github.com/MicroEJ/FreeRTOS/tree/tuto-microej-firmware-from-scratch.

Note: The implementation of the Low Level API and their validation with the Platform Qualification Tools (PQT) will be the topic of another training.

Prerequisites

- MicroEJ SDK 5 from version 5.3.0 to latest (distribution 20.10). Can be downloaded from https://repository.microej.com/packages/SDK (tested on MicroEJ SDK distribution 20.10)
- Windows 10 or higher with Windows Subsystem for Linux (WSL). See the installation guide.
- A Linux distribution installed on WSL (Tested on Ubuntu 19.10 eoan and Ubuntu 20.04 focal).

Note: In WSL, use the command `lsb_release -a` to print the current Ubuntu version.

A code editor such as Visual Studio Code is also recommended to edit BSP files.

Overview

The next sections describe step by step how to build a MicroEJ Firmware that runs a HelloWorld MicroEJ Application on the emulated device.

The steps to follow are:

1. Setup the development environment (assuming the prerequisites are satisfied).
2. Get a running BSP
3. Build the MicroEJ Platform
4. Create the HelloWorld MicroEJ Application
5. Implement the minimum Low Level API to run the application

This training goes through trials and errors every Firmware developers may encounter. It provides a solution after each error rather than providing the full solution in one go.

Setup the Development Environment

This section assumes the prerequisites have been properly installed.

In WSL:

1. Update apt's cache: `sudo apt-get update`
2. Install qemu-system-arm and GNU CC toolchain for ARM: `sudo apt-get install -y qemu-system-arm gcc-arm-none-eabi build-essential subversion`
3. The rest of this training will use the folder `src/tuto-from-scratch/` in the Windows home folder.
4. Create the folder: `mkdir -p /mnt/c/Users/${USER}/src/tuto-from-scratch` (the `-p` option ensures all the directories are created).
5. Go into the folder: `cd /mnt/c/Users/${USER}/src/tuto-from-scratch/`
6. Clone FreeRTOS and its submodules: `git clone -b V10.3.1 --recursive https://github.com/FreeRTOS/FreeRTOS.git` (this may takes some time)

Note: Use the right-click to paste from the Windows clipboard into WSL console. The right-click is also used to copy from the WSL console into the Windows clipboard.

Get Running BSP

This section presents how to get running BSP based on FreeRTOS that boots on the target device.

1. Go into the target device sub-project: `cd FreeRTOS/FreeRTOS/Demo/CORTEX_LM3S811_GCC`
2. Build the project: `make`

Ignoring the warnings, the following error appears during the link:

```
CC    hw_include/osram96x16.c
LD    gcc/RTOSDemo.axf
arm-none-eabi-ld: section .text.startup LMA [0000000000002b24,
↪0000000000002c8f] overlaps section .data LMA [0000000000002b24,0000000000002b27]
make: *** [makedefs:191: gcc/RTOSDemo.axf] Error 1
```

Insert the following fixes in the linker script file named standalone.ld (thanks to http://roboticravings.blogspot.com/2018/07/freertos-on-cortex-m3-with-qemu.html).

Note: WSL can start the editor Visual Studio Code. type code . in WSL. . represents the current directory in Unix.

Listing 44: https://github.com/MicroEJ/FreeRTOS/commit/48248eae13baebf7df9638cd8da6fbfe1a735a9c

```
diff --git a/FreeRTOS/Demo/CORTEX_LM3S811_
↪GCC/standalone.ld b/FreeRTOS/Demo/CORTEX_LM3S811_GCC/standalone.ld
-- a/FreeRTOS/Demo/CORTEX_LM3S811_GCC/standalone.ld
+++ b/FreeRTOS/Demo/CORTEX_LM3S811_GCC/standalone.ld
@@ -42,7 +42,15 @@ SECTIONS
        _etext = .;
    } > FLASH

-   .data : AT (ADDR(.text) + SIZEOF(.text))
+   .ARM.exidx :
+   {
+       *(.ARM.exidx*)
+       *(.gnu.linkonce.armexidx.*)
+   } > FLASH
+
+   _begin_data = .;
+
+   .data : AT ( _begin_data )
    {
        _data = .;
        *(vtable)
```

This is the output of the git diff command. Lines starting with a - should be removed. Lines starting with a + should be added.

Note: The patch(1) can be used to apply the patch. Assuming WSL shell is in FreeRTOS/Demo/CORTEX_LM3S811_GCC directory:

1. Install dos2unix utility: sudo apt install dos2unix
2. Convert all files to unix line-ending: find -type f -exec dos2unix {} \;
3. Copy the content of the code block in a file named linker.patch (every lines of the code block must be copied in the file).
4. Apply the patch: patch -l -p4 < linker.patch .

It is also possible to paste the diff directly into the console:

1. In WSL, invoke patch -l -p4 . The command starts, waiting for input on stdin (the standard input).

2. Copy the diff and paste it in WSL
3. Press enter
4. Press Ctrl-d Ctrl-d (press the Control key + the letter d twice).

3. Run the build again: make
4. Run the emulator with the generated kernel: qemu-system-arm -M lm3s811evb -nographic -kernel gcc/RTOSDemo.bin

   The following error appears and then nothing:

```
ssd0303: error: Unknown command: 0x80
ssd0303: error: Unexpected byte 0xe3
ssd0303: error: Unknown command: 0x80
ssd0303: error: Unexpected byte 0xe3
ssd0303: error: Unknown command: 0x80
ssd0303: error: Unexpected byte 0xe3
ssd0303: error: Unknown command: 0x80
ssd0303: error: Unexpected byte 0xe3
ssd0303: error: Unknown command: 0x80
ssd0303: error: Unexpected byte 0xe3
ssd0303: error: Unknown command: 0x80
ssd0303: error: Unexpected byte 0xe3
ssd0303: error: Unknown command: 0x80
ssd0303: error: Unexpected byte 0xe3
ssd0303: error: Unknown command: 0x80
ssd0303: error: Unexpected byte 0xe3
ssd0303: error: Unknown command: 0x80
ssd0303: error: Unexpected byte 0xe3
```

5. Press Ctrl-a x (press Control + the letter a , release, press x) to the end the QEMU session. The session ends with QEMU: Terminated .

Note: The errors can be safely ignored. They occur because the OLED controller emulated receive incorrect commands.

At this point, the target device is successfully booted with the FreeRTOS kernel.

FreeRTOS Hello World

This section describes how to configure the BSP to print text on the QEMU console.

The datasheet of the target device (LM3S811 datasheet) describes how to use the UART device and an example implementation for QEMU is available here).

Here is the patch that implements putchar(3) and puts(3) and prints Hello World .

Listing 45: https://github.com/MicroEJ/FreeRTOS/commit/d09ec0f5cbdf69ca97a5ac15f8b905538aa4c61e

```
diff --git a/FreeRTOS/Demo/
↪CORTEX_LM3S811_GCC/main.c b/FreeRTOS/Demo/CORTEX_LM3S811_GCC/main.c
-- a/FreeRTOS/Demo/CORTEX_LM3S811_GCC/main.c
+++ b/FreeRTOS/Demo/CORTEX_LM3S811_GCC/main.c
```

(continues on next page)

(continued from previous page)

```
@@ -134,9 +134,25 @@ SemaphoreHandle_t xButtonSemaphore;
 QueueHandle_t xPrintQueue;

 /*-----------------------------------------*/
+#define UART0BASE ((volatile int*) 0x4000C000)
+
+int putchar (int c){
+    (*UART0BASE) = c;
+    return c;
+}
+
+int puts(const char *s) {
+    while (*s) {
+        putchar(*s);
+        s++;
+    }
+    return putchar('\n');
+}

 int main( void )
 {
+    puts("Hello, World! puts function is working.");
+
    /* Configure the clocks, UART and GPIO. */
    prvSetupHardware();
```

Rebuild and run the newly generated kernel: `make && qemu-system-arm -M lm3s811evb -nographic -kernel gcc/RTOSDemo.bin` (press `Ctrl-a x` to interrupt the emulator).

```
make: Nothing to be done for 'all'.
Hello, World! puts function is working.
ssd0303: error: Unknown command: 0x80
ssd0303: error: Unexpected byte 0xe3
ssd0303: error: Unknown command: 0x80
ssd0303: error: Unexpected byte 0xe3
ssd0303: error: Unknown command: 0x80
ssd0303: error: Unexpected byte 0xe3
ssd0303: error: Unknown command: 0x80
ssd0303: error: Unexpected byte 0xe3
ssd0303: error: Unknown command: 0x80
ssd0303: error: Unexpected byte 0xe3
ssd0303: error: Unknown command: 0x80
ssd0303: error: Unexpected byte 0xe3
ssd0303: error: Unknown command: 0x80
ssd0303: error: Unexpected byte 0xe3
ssd0303: error: Unknown command: 0x80
ssd0303: error: Unexpected byte 0xe3
ssd0303: error: Unknown command: 0x80
ssd0303: error: Unexpected byte 0xe3
QEMU: Terminated
```

With this two functions implemented, `printf(3)` is also available.

Listing 46: https://github.com/MicroEJ/FreeRTOS/commit/1f7e7ee014754a4dcb4f6c5a470205e02f6ac3c8

```
diff --git a/FreeRTOS/Demo/
↪CORTEX_LM3S811_GCC/main.c b/FreeRTOS/Demo/CORTEX_LM3S811_GCC/main.c
-- a/FreeRTOS/Demo/CORTEX_LM3S811_GCC/main.c
+++ b/FreeRTOS/Demo/CORTEX_LM3S811_GCC/main.c
@@ -149,9 +149,11 @@ int puts(const char *s) {
        return putchar('\n');
 }

+#include <stdio.h>
+
 int main( void )
 {
-       puts("Hello, World! puts function is working.");
+       printf("Hello, World! printf function is working.\n");

        /* Configure the clocks, UART and GPIO. */
        prvSetupHardware();
```

At this point, the character output on the UART is implemented in the FreeRTOS BSP. The next step is to create the MicroEJ Platform and MicroEJ Application.

Create a MicroEJ Platform

This section describes how to create and configure a MicroEJ Platform compatible with the FreeRTOS BSP and GCC toolchain.

- A MicroEJ Architecture is a software package that includes the *MicroEJ Runtime* port to a specific target Instruction Set Architecture (ISA) and C compiler. It contains a set of libraries, tools and C header files. The MicroEJ Architectures are provided by MicroEJ SDK.
- A MicroEJ Platform is a port of a MicroEJ Architecture for a custom device. It contains the MicroEJ configuration and the BSP (C source files).

  When selecting a MicroEJ Architecture, special care must be taken to ensure the compatibility between the toolchain used in the BSP and the toolchain used to build the Core Engine included in the MicroEJ Architecture.

  The list of MicroEJ Architectures supported is listed *here*. MicroEJ Evaluation Architectures provided by MicroEJ Corp. can be downloaded from MicroEJ Architectures Repository.

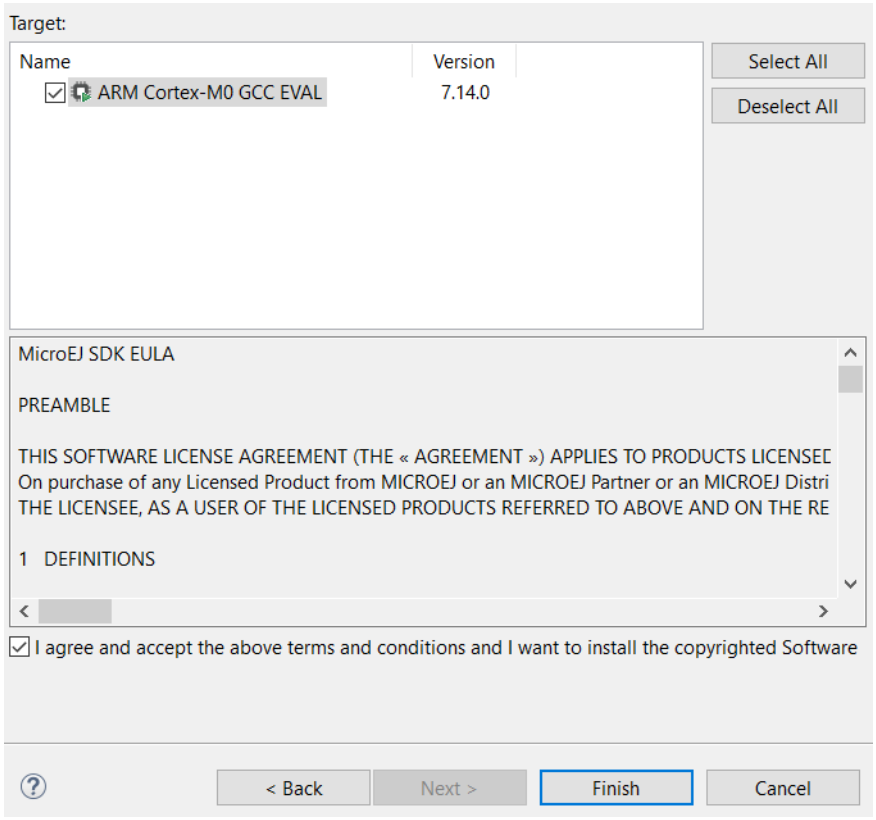
  There is no CM3 in MicroEJ Architectures Repository and the Arm® Cortex®-M3 MCU is not mentioned in the *capabilities matrix*. This means that the MicroEJ Architectures for Arm® Cortex®-M3 MCUs are no longer distributed for evaluation. Download the latest MicroEJ Architecture for Arm® Cortex®-M0 instead (the Arm® architectures are binary upward compatible from Arm®v6-M (Cortex®-M0) to Arm®v7-M (Cortex®-M3)).

Import the MicroEJ Architecture

This step describes how to import a *MicroEJ Architecture*.

1. Start MicroEJ SDK on an empty workspace. For example, create an empty folder `workspace` next to the `FreeRTOS` git folder and select it.
2. Keep the default MicroEJ Repository
3. Download the latest MicroEJ Architecture for Arm® Cortex®-M0 instead: https://repository.microej.com/modules/com/microej/architecture/CM0/CM0_GCC48/flopi0G22/maintenance/7.20.3/flopi0G22-7.20.3-eval.xpf
4. Import the MicroEJ Architecture in MicroEJ SDK

1. File > Import > MicroEJ > Architectures
2. select the MicroEJ Architecture file downloaded
3. Accept the license and click on Finish



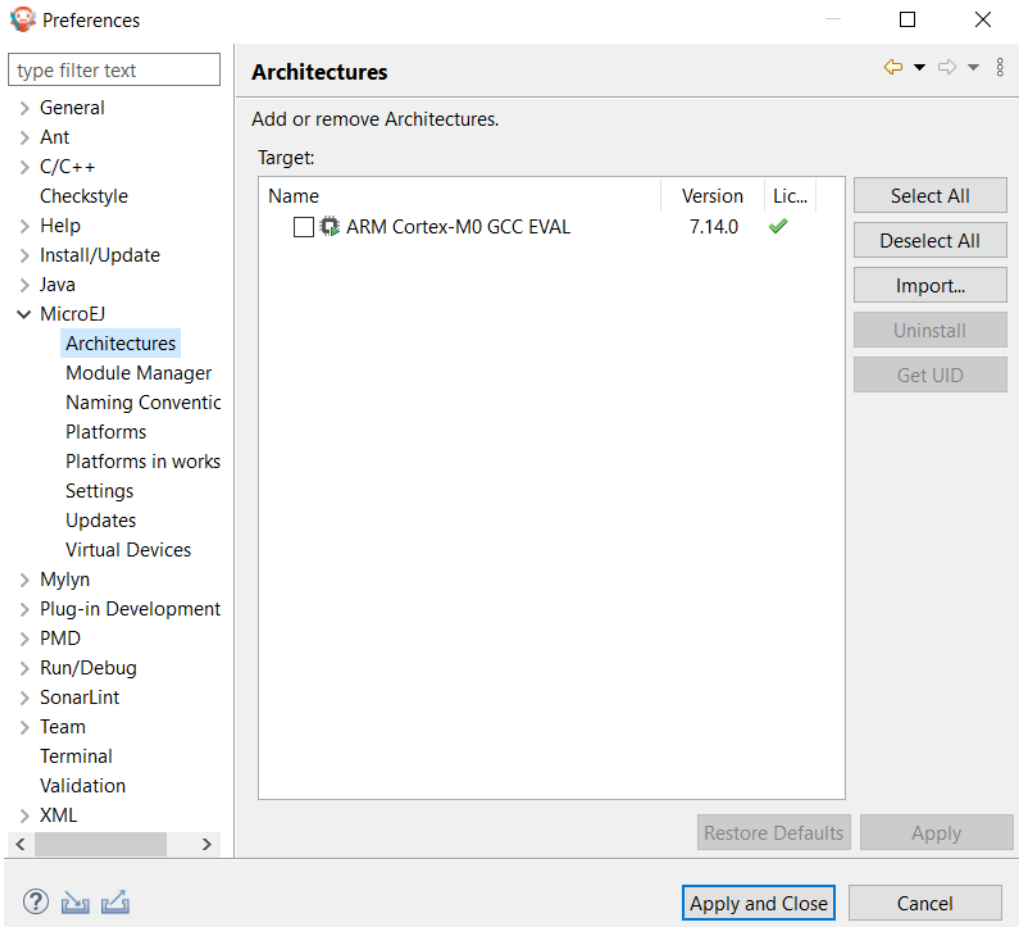
Install an Evaluation License

This step describes how to create and activate an *Evaluation License* for the MicroEJ Architecture previously imported.

1. Select the Window > Preferences > MicroEJ > Architectures menu .
2. Click on the architectures and press Get UID .
3. Copy the UID. It will be needed when requesting a license.
4. Go to https://license.microej.com.
5. Click on Create a new account link.
6. Create an account with a valid email address. A confirmation email will be sent a few minutes after. Click on the confirmation link in the email and login with the account.
7. Click on Activate a License .
8. Set Product P/N: to 9PEVNLDBU6IJ .
9. Set UID: to the UID generated before.
10. Click on Activate .

- The license is being activated. An activation mail should be received in less than 5 minutes. If not, please contact contact *our support team*.
- Once received by email, save the attached zip file that contains the activation key.

11. Go back to Microej SDK.
12. Select the Window > Preferences > MicroEJ menu.
13. Press Add… .
14. Browse the previously downloaded activation key archive file.
15. Press OK . A new license is successfully installed.
16. Go to Architectures sub-menu and check that all architectures are now activated (green check).
17. Microej SDK is successfully activated.

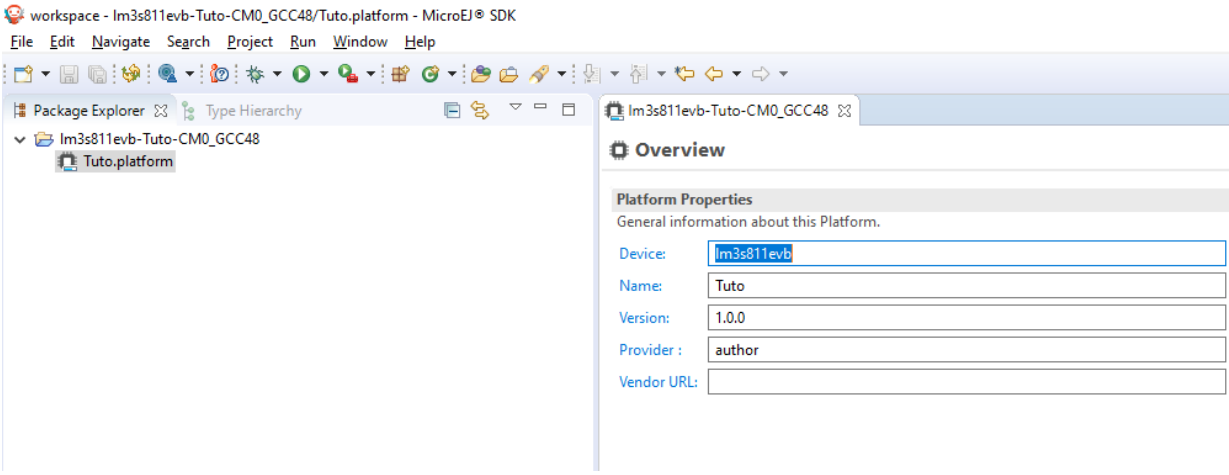


Create the MicroEJ Platform

This step describes how to create a new *MicroEJ Platform* using the MicroEJ Architecture previously imported.

1. Select File > New > Platform Project .
2. Ensure the Architecture selected is the MicroEJ Architecture previously imported.
3. Ensure the Create from a platform reference implementation box is unchecked.
4. Click on Next button.
5. Fill the fields:

- Set Device: to lm3s811evb
- Set Name: to Tuto



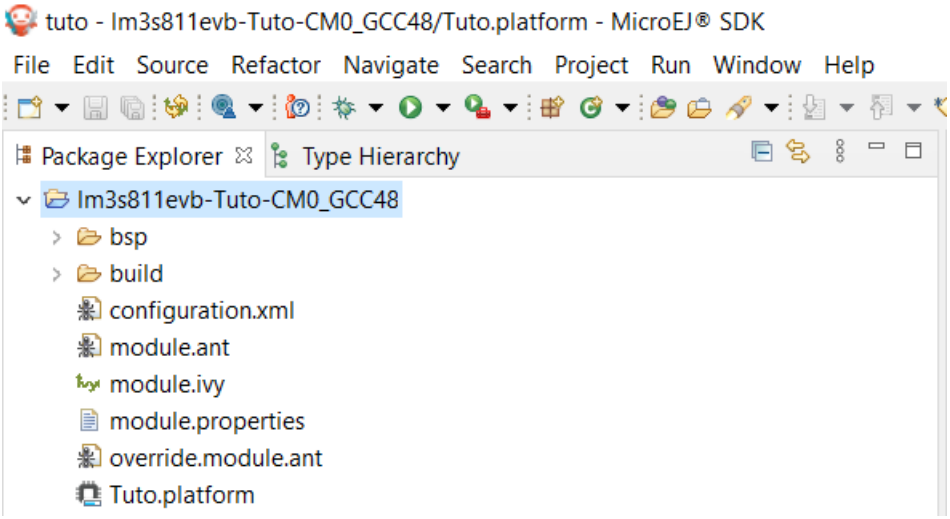
Setup the MicroEJ Platform

This step describes how to configure the MicroEJ Platform previously created. For more information on this topic, please refer to *VEE Port Project Creation*.

The Platform Configuration Additions provide a flexible way to configure the *BSP connection* between the MicroEJ Platform and MicroEJ Application to the BSP. In this training, the Partial BSP connection is used. That is, the MicroEJ SDK will output all MicroEJ files (C headers, MicroEJ Application `microejapp.o`, MicroEJ Runtime `microejruntime.a`, ...) in a location known by the BSP. The BSP is configured to compile and link with those files.

For this training, that means that the final binary is produced by invoking `make` in the FreeRTOS BSP.

1. Install the Platform Configuration Additions by copying all the files within the `content` folder in the MicroEJ Platform folder.



Note: The content directory contains files that must be installed in a MicroEJ Platform configuration directory (the directory that contains the .platform file). It can be automatically downloaded using the following command line:

```
svn export --force https://github.com/MicroEJ/VEEPortQualificationTools/
↪tags/2.6.0/framework/platform/content [path_to_platform_configuration_directory]
```

2. Edit the file bsp/bsp.properties as follow:

```
# Specify the MicroEJ Application file ('microejapp.o') parent directory.
# This is a '/' separated directory relative to 'bsp.root.dir'.
microejapp.relative.dir=microej/lib

# Specify the MicroEJ Platform runtime file ('microejruntime.a') parent directory.
# This is a '/' separated directory relative to 'bsp.root.dir'.
microejlib.relative.dir=microej/lib

# Specify MicroEJ Platform header files ('*.h') parent directory.
# This is a '/' separated directory relative to 'bsp.root.dir'.
microejinc.relative.dir=microej/inc
```

3. Edit the file module.ivy and add the MicroEJ Architecture as a dependency:

```
<dependencies>
      <dependency org="com.microej.
↪architecture.CM0.CM0_GCC48" branch="maintenance" name="flopi0G22" rev="7.20.3">
            <artifact name=
↪"flopi0G22" m:classifier="${com.microej.platformbuilder.architecture.usage}" ext="xpf"/>
      </dependency>
</dependencies>
```

4. Edit the file module.properties and set the MicroEJ platform filename:

```
# Platform configuration file (relative to this project).
com.microej.platformbuilder.platform.filename=Tuto.platform
```

5. Right-click on the platform project and click on Build Module .
6. The following message appears in the console:

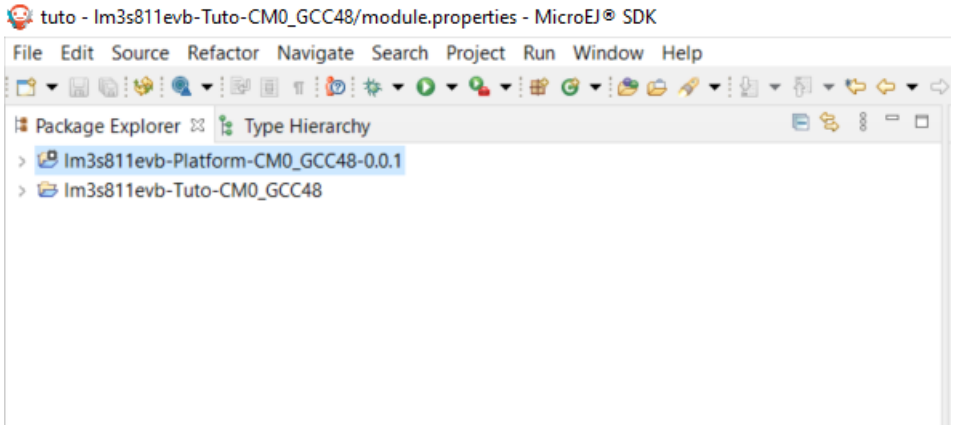
```
module-platform:report:
␣
 ↪␣
 ↪␣
 ↪␣
 ↪␣
 ↪␣
 ↪  [echo]     ␣
 ↪==========================================================================
       [echo]      Platform has been built in this directory␣
↪'C:\Users\user\src\tuto-from-scratch\workspace/lm3s811evb-Platform-CM0_GCC48-0.0.1'.
     ␣
 ↪  [echo]      To import this project in your MicroEJ SDK workspace (if not already available):
       [echo]␣
 ↪      - Select 'File' > 'Import...' > 'General' > 'Existing Projects into Workspace' > 'Next'
       [echo]       - Check 'Select root directory' and browse 'C:\Users\
```

(continues on next page)

(continued from previous page)

```
↪user\src\tuto-from-scratch\workspace/lm3s811evb-Platform-CM0_GCC48-0.0.1' > 'Finish'
␣
↪␣
↪␣
↪␣
↪␣
↪␣
↪ [echo]
↪=============================================================================

BUILD SUCCESSFUL
```

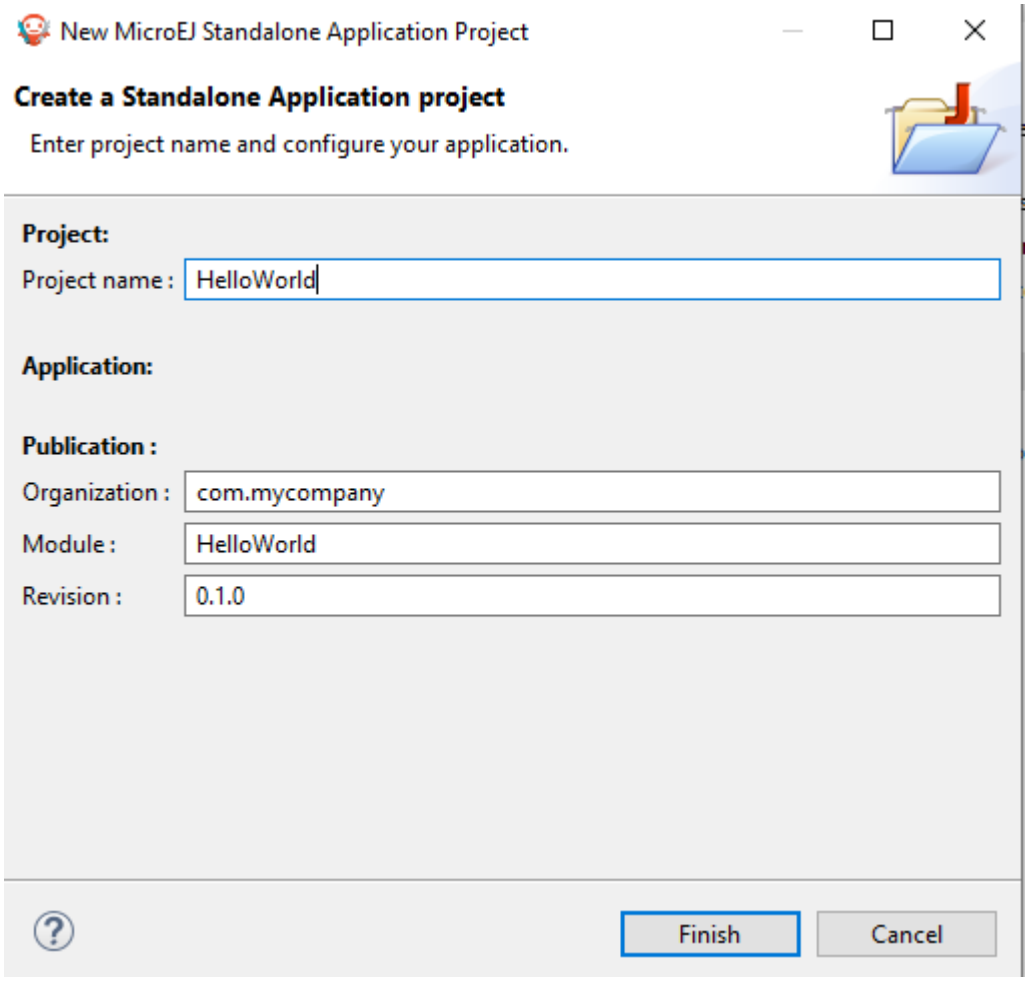
7. Follow the instructions to import the generated platform in the workspace:



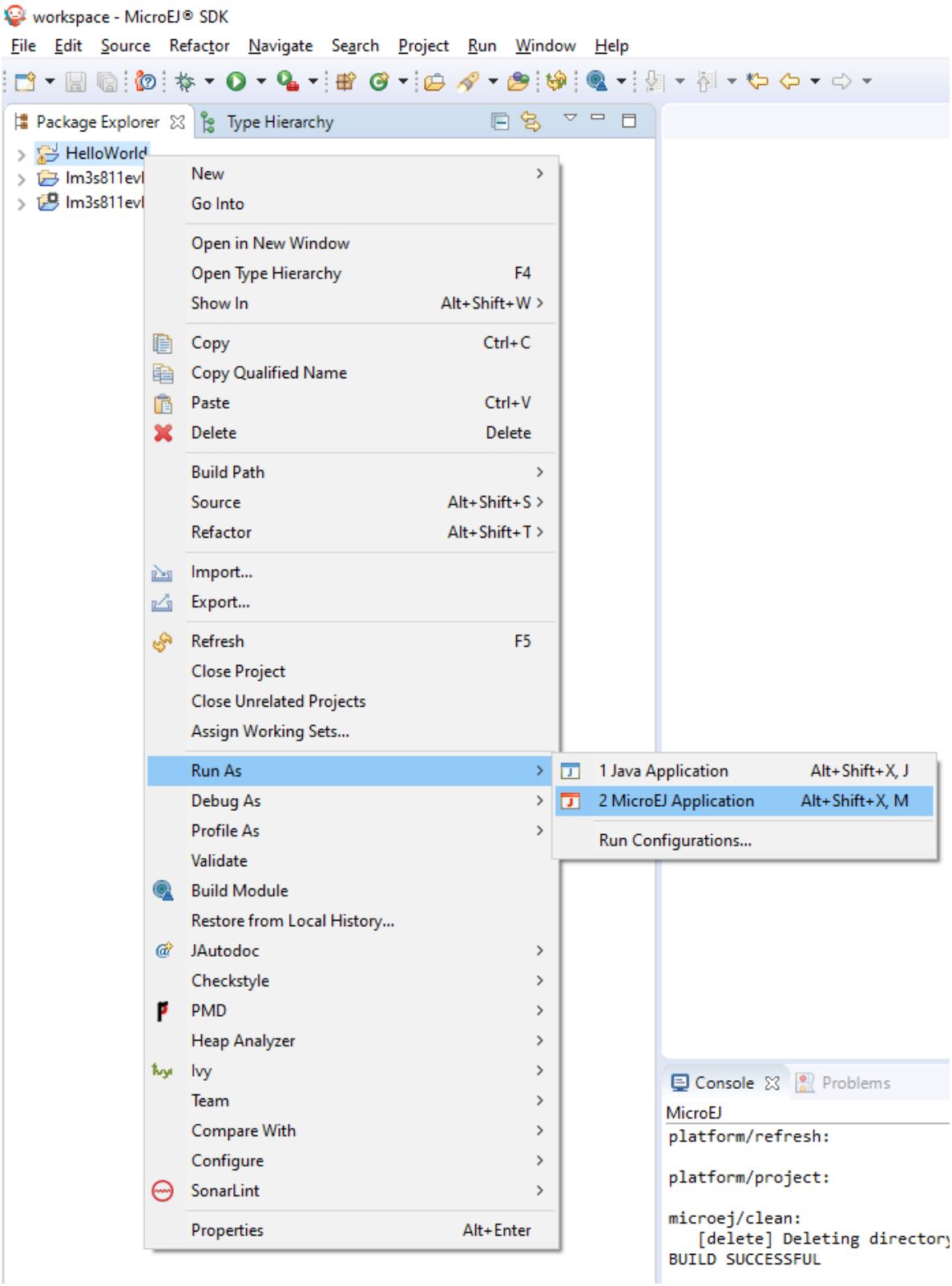
At this point, the MicroEJ Platform is ready to be used to build MicroEJ Applications.

Create MicroEJ Application HelloWorld

1. Select File > New > Standalone Application Project .
2. Set the name to HelloWorld and click on Finish



3. Run the application in Simulator to ensure it is working properly. Right-click on HelloWorld project > Run As > MicroEJ Application



The following message appears in the console:

```
================ [ Initialization Stage ] ================
================ [ Launching on Simulator ] ================
Hello World!
================ [ Completed Successfully ] ================

SUCCESS
```

Configure BSP Connection in MicroEJ Application

This step describes how to configure the *BSP connection* for the HelloWorld MicroEJ Application and how to build the MicroEJ Application that will run on the target device.

For a MicroEJ Application, the BSP connection is configured in the `PROJECT-NAME/build/emb.properties` file.

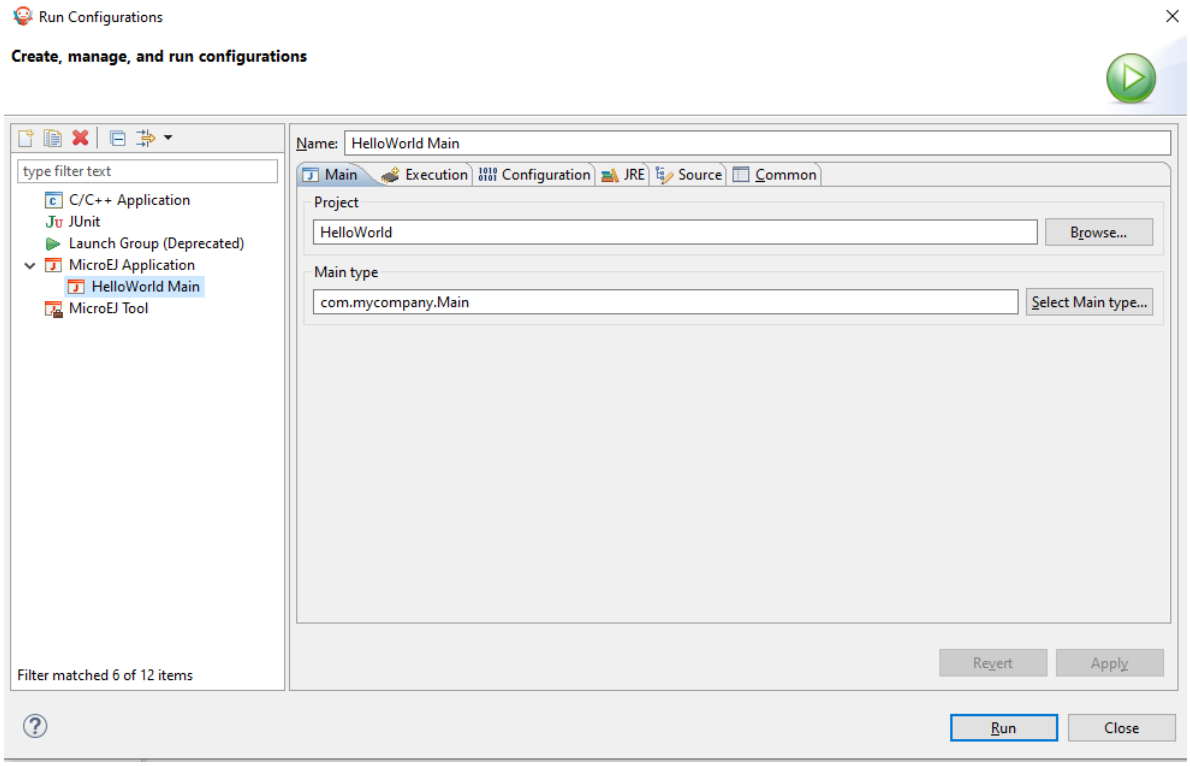
1. Create a file `HelloWorld/build/emb.properties` with the following content:

```
core.memory.immortal.size=0
core.memory.javaheap.size=1024
core.memory.threads.pool.size=4
core.memory.threads.size=1
core.memory.thread.max.size=4
deploy.bsp.microejapp=true
deploy.bsp.microejlib=true
deploy.bsp.microejinc=true
deploy.bsp.
↪root.dir=[absolute_path] to FreeRTOS\\FreeRTOS\\Demo\\CORTEX_LM3S811_GCC
```

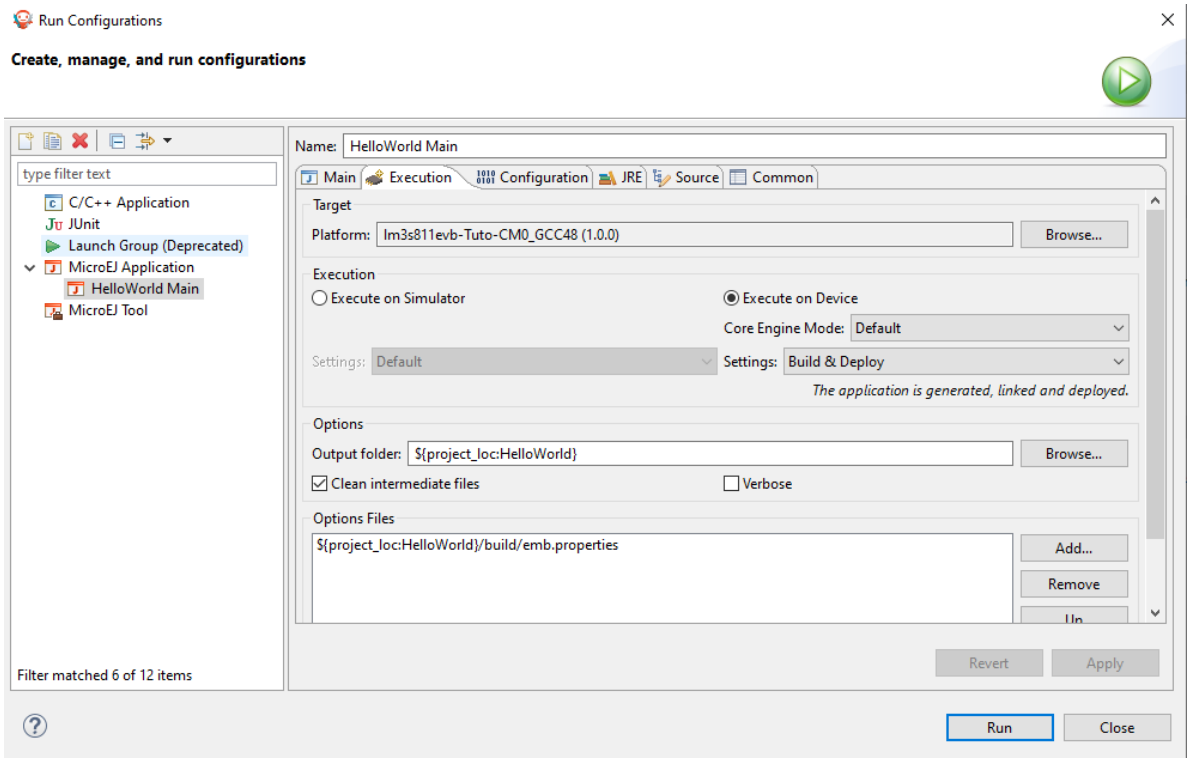
Note: Assuming the WSL current directory is `FreeRTOS/FreeRTOS/Demo/CORTEX_LM3S811_GCC`, use the following command to find the `deploy.bsp.root.dir` path with proper escaping:

```
pwd | sed -e 's|/mnt/c/|C:\\\\|' -e 's|/|\\\\|g'
```

2. Open Run > Run configurations…
3. Select the HelloWorld launcher configuration



4. Select `Execution` tab.
5. Change the execution mode from `Execute on Simulator` to `Execute on Device` .
6. Add the file `build/emb.properties` to the options files



7. Click on Run

```
================ [ Initialization Stage ] ================
Platform connected to BSP location 'C:\Users\user\src\tuto-from-scratch\FreeRTOS\
↪FreeRTOS\Demo\CORTEX_LM3S811_GCC' using application option 'deploy.bsp.root.dir'.
================ [ Launching SOAR ] ================
================ [ Launching Link ] ================
================ [ Deployment ] ================
MicroEJ files for the 3rd-party BSP project are generated to 'C:\
↪Users\user\src\tuto-from-scratch\workspace\HelloWorld\com.mycompany.Main\platform'.
The MicroEJ application (microejapp.o) has been deployed to: 'C:\Users\user\src\
↪tuto-from-scratch\FreeRTOS\FreeRTOS\Demo\CORTEX_LM3S811_GCC\microej\lib'.
The MicroEJ platform library (microejruntime.a) has been deployed to: 'C:\Users\user\src\
↪tuto-from-scratch\FreeRTOS\FreeRTOS\Demo\CORTEX_LM3S811_GCC\microej\lib'.
The MicroEJ platform header files (*.h) have been deployed to: 'C:\Users\user\src\
↪tuto-from-scratch\FreeRTOS\FreeRTOS\Demo\CORTEX_LM3S811_GCC\microej\inc'.
================ [ Completed Successfully ] ================

SUCCESS
```

At this point, the HelloWorld MicroEJ Application is built and deployed in the FreeRTOS BSP.

MicroEJ and FreeRTOS Integration

This section describes how to finalize the integration between MicroEJ and FreeRTOS to get a working firmware that runs the HelloWorld MicroEJ Application built previously.

In the previous section, when the MicroEJ Application was built, several files were added to a new folder named `microej/` .

```
$ pwd
/mnt/c/
↪Users/user/src/tuto-from-scratch/FreeRTOS/FreeRTOS/Demo/CORTEX_LM3S811_GCC
$ tree microej/
microej/
   inc
      BESTFIT_ALLOCATOR.h
      BESTFIT_ALLOCATOR_impl.h
      LLBSP_impl.h
      LLMJVM.h
      LLMJVM_MONITOR_impl.h
      LLMJVM_impl.h
      LLTRACE_impl.h
      MJVM_MONITOR.h
      MJVM_MONITOR_types.h
      intern
         BESTFIT_ALLOCATOR.h
         BESTFIT_ALLOCATOR_impl.h
         LLBSP_impl.h
         LLMJVM.h
         LLMJVM_impl.h
         trace_intern.h
      sni.h
      trace.h
   lib
      microejapp.o
      microejruntime.a
```

(continues on next page)

(continued from previous page)

```
3 directories, 19 files
```

- The microej/lib folder contains the HelloWorld MicroEJ Application object file (microejapp.o) and the MicroEJ Runtime. The final binary must be linked with these two files.
- The microej/inc folder contains several C header files used to expose MicroEJ Low Level APIs. The functions defined in files ending with the _impl.h suffix should be implemented by the BSP.

  To summarize, the following steps remain to complete the integration between MicroEJ and the FreeRTOS BSP:

- Implement minimal Low Level APIs
- Invoke the Core Engine
- Build and link the firmware with the MicroEJ Runtime and MicroEJ Application

Minimal Low Level APIs

The purpose of this training is to demonstrate how to develop a minimal MicroEJ Architecture, it is not to develop a complete MicroEJ Architecture. Therefore this training implements only the required functions and provides stub implementation for unused features. For example, the following implementation does not support scheduling.

The two headers that must be implemented are LLBSP_impl.h and LLMJVM_impl.h .

1. In the BSP, create a folder named microej/src (next to the microej/lib and microej/inc folders).
2. Implement LLBSP_impl.h in LLBSP.c :

Listing 47: microej/src/LLBSP.c

```
#include "LLBSP_impl.h"

extern void _etext(void);
uint8_t LLBSP_IMPL_isInReadOnlyMemory(void* ptr)
{
  return ptr < &_etext;
}

/**
 * Writes the character <code>c</code>, cast to an unsigned char, to stdout stream.
 *␣
↪This function is used by the default implementation of the Java <code>System.out</code>.
 */
void LLBSP_IMPL_putchar(int32_t c)
{
  putchar(c);
}
```

- The implementation of LLBSP_IMPL_putchar reuses the putchar implemented previously.
- The rodata section is defined in the linker script standalone.ld . The flash memory starts at 0 and the end of the section is stored in the _etex symbol.

3. Implement LLMJVM_impl.h in LLMJVM_stub.c (all functions are stubbed with a dummy implementation):

Listing 48: microej/src/LLMJVM_stub.c

```
#include "LLMJVM_impl.h"


int32_t LLMJVM_IMPL_initialize()
{
	return LLMJVM_OK;
}

int32_t LLMJVM_IMPL_vmTaskStarted()
{
	return LLMJVM_OK;
}

int32_t LLMJVM_IMPL_scheduleRequest(int64_t absoluteTime)
{
	return LLMJVM_OK;
}

int32_t LLMJVM_IMPL_idleVM()
{
	return LLMJVM_OK;
}

int32_t LLMJVM_IMPL_wakeupVM()
{
	return  LLMJVM_OK;
}

int32_t LLMJVM_IMPL_ackWakeup()
{
	return LLMJVM_OK;
}

int32_t LLMJVM_IMPL_getCurrentTaskID()
{
	return (int32_t) 123456;
}

void LLMJVM_IMPL_setApplicationTime(int64_t t)
{

}

int64_t LLMJVM_IMPL_getCurrentTime(uint8_t system)
{
  return 0;
}

int64_t LLMJVM_IMPL_getTimeNanos()
{
	return 0;
}
```

(continues on next page)

(continued from previous page)

```
int32_t LLMJVM_IMPL_shutdown(void)
{
        return LLMJVM_OK;
}
```

The microej folder in the BSP has the following structure:

```
$ pwd
/mnt/c/
↪Users/user/src/tuto-from-scratch/FreeRTOS/FreeRTOS/Demo/CORTEX_LM3S811_GCC
$ tree microej/
microej/
   inc
      BESTFIT_ALLOCATOR.h
      BESTFIT_ALLOCATOR_impl.h
      LLBSP_impl.h
      LLMJVM.h
      LLMJVM_MONITOR_impl.h
      LLMJVM_impl.h
      LLTRACE_impl.h
      MJVM_MONITOR.h
      MJVM_MONITOR_types.h
      intern
         BESTFIT_ALLOCATOR.h
         BESTFIT_ALLOCATOR_impl.h
         LLBSP_impl.h
         LLMJVM.h
         LLMJVM_impl.h
         trace_intern.h
      sni.h
      trace.h
   lib
      microejapp.o
      microejruntime.a
   src
      LLBSP.c
      LLMJVM_stub.c

4 directories, 21 files
```

Invoke the Core Engine

The Core Engine is created and initialized with the C function SNI_createVM. Then it is started and executed in the current RTOS task by calling SNI_startVM. The function SNI_startVM returns when the MicroEJ Application exits. Both functions are declared in the C header sni.h.

Listing 49: https://github.com/MicroEJ/FreeRTOS/commit/7ae8e79f9c811621569ccb90c46b1dcda91da35d

```
diff --git a/FreeRTOS/Demo/
↪CORTEX_LM3S811_GCC/main.c b/FreeRTOS/Demo/CORTEX_LM3S811_GCC/main.c
-- a/FreeRTOS/Demo/CORTEX_LM3S811_GCC/main.c
+++ b/FreeRTOS/Demo/CORTEX_LM3S811_GCC/main.c
@@ -150,11 +150,14 @@ int puts(const char *s) {
```

(continues on next page)

(continued from previous page)

```
 }

 #include <stdio.h>
+#include "sni.h"

 int main( void )
 {
       printf("Hello, World! printf function is working.\n");

+       SNI_startVM(SNI_createVM(), 0, NULL);
+
    /* Configure the clocks, UART and GPIO. */
    prvSetupHardware();
```

Build and Link the Firmware with the MicroEJ Runtime and MicroEJ Application

To build and link the firmware with the MicroEJ Runtime and MicroEJ Application, the BSP port must be modified to:

1. Use the MicroEJ header files in folder microej/inc
2. Use the source files folder microej/src that contains the Low Level API implementation LLBSP.c and LLMJVM_stub.c
3. Compile and link LLBSP.o and LLMJVM_stub.o
4. Link with MicroEJ Application (microej/lib/microejapp.o) and MicroEJ Runtime (microej/lib/microejruntime.a)

The following patch updates the BSP port Makefile to do it:

Listing 50: https://github.com/MicroEJ/FreeRTOS/commit/257d9e1d123be0342029e2930c0073dd5a4a2b2d

```
-- a/FreeRTOS/Demo/CORTEX_LM3S811_GCC/Makefile
+++ b/FreeRTOS/Demo/CORTEX_LM3S811_GCC/Makefile
@@ -29,8 +29,10 @@ RTOS_SOURCE_DIR=../../Source
 DEMO_SOURCE_DIR=../Common/Minimal

 CFLAGS+=-I hw_
↪include -I . -I ${RTOS_SOURCE_DIR}/include -I ${RTOS_SOURCE_DIR}/portable/
↪GCC/ARM_CM3 -I ../Common/include -D GCC_ARMCM3_LM3S102 -D inline=
+CFLAGS+= -I microej/inc

 VPATH=
↪${RTOS_SOURCE_DIR}:${RTOS_SOURCE_DIR}/portable/MemMang:${RTOS_
↪SOURCE_DIR}/portable/GCC/ARM_CM3:${DEMO_SOURCE_DIR}:init:hw_include
+VPATH+= microej/src

 OBJS=${COMPILER}/main.o        \
         ${COMPILER}/list.o     \
@@ -44,9 +46,12 @@ OBJS=${COMPILER}/main.o       \
         ${COMPILER}/semtest.o \
         ${COMPILER}/osram96x16.o

+OBJS+= ${COMPILER}/LLBSP.o ${COMPILER}/LLMJVM_stub.o
```

(continues on next page)

(continued from previous page)

```
+
 INIT_OBJS= ${COMPILER}/startup.o

 LIBS= hw_include/libdriver.a
+LIBS+= microej/lib/microejruntime.a microej/lib/microejapp.o
```

Then build the firmware with `make`. The following error occurs at link time.

```
  CC    microej/src/LLMJVM_stub.c
  LD    gcc/RTOSDemo.axf
↪
↪                                    arm-none-eabi-ld: error: microej/lib/microejruntime.a(sni_
↪vm_startup_greenthread.o) uses VFP register arguments, gcc/RTOSDemo.axf does not
arm-none-eabi-ld: failed to merge
↪target specific data of file microej/lib/microejruntime.a(sni_vm_startup_greenthread.o)
arm-none-eabi-ld: gcc/RTOSDemo.axf section `ICETEA_HEAP' will not fit in region `SRAM'
arm-none-eabi-ld: region `SRAM' overflowed by 4016 bytes
microej/lib/microejapp.o: In function `__java_internStrings_end':
```

The RAM requirements of the BSP (with printf), FreeRTOS, the MicroEJ Application and MicroEJ Runtime do not fit in the 8k of SRAM. It is possible to link within 8k of RAM by customizing a *MicroEJ Tiny-Sandbox* on a baremetal device (without a RTOS) but this is not the purpose of this training.

Instead, this training will switch to another device, the Luminary Micro Stellaris LM3S6965EVB. This device is almost identical as the LM3S811EVB but it has 256k of flash memory and 64k of SRAM. Updating the values in the linker script `standalone.ld` is sufficient to create a valid BSP port for this device.

Instead of continuing to work with the LM3S811 port, create a copy, named CORTEX_LM3S6965_GCC:

```
$ cd ..
$ pwd
/mnt/c/Users/user/src/tuto-from-scratch/FreeRTOS/FreeRTOS/Demo
$ cp -r CORTEX_LM3S811_GCC/ CORTEX_LM3S6965_GCC
$ cd CORTEX_LM3S6965_GCC
```

The BSP path defined by the property `deploy.bsp.root.dir` in the MicroEJ Application must be updated as well.

The rest of the training assumes that everything is done in the `CORTEX_LM3S6965_GCC` folder.

Then update the linker script `standlone.ld`:

Listing 51: https://github.com/MicroEJ/FreeRTOS/commit/0e2e31d8a510d37178c340051bab636902471eea

```
diff --git a/FreeRTOS/Demo/CORTEX_LM3S6965_
↪GCC/standalone.ld b/FreeRTOS/Demo/CORTEX_LM3S6965_GCC/standalone.ld
-- a/FreeRTOS/Demo/CORTEX_LM3S6965_GCC/standalone.ld
+++ b/FreeRTOS/Demo/CORTEX_LM3S6965_GCC/standalone.ld
@@ -28,8 +28,8 @@

 MEMORY
 {
```

(continues on next page)

(continued from previous page)

```
-   FLASH (rx) : ORIGIN = 0x00000000, LENGTH = 64K
-   SRAM (rwx) : ORIGIN = 0x20000000, LENGTH = 8K
+    FLASH (rx) : ORIGIN = 0x00000000, LENGTH = 256K
+    SRAM (rwx) : ORIGIN = 0x20000000, LENGTH = 64K
 }

 SECTIONS
```

The new command to run the firmware with QEMU is: `qemu-system-arm -M lm3s6965evb -nographic -kernel gcc/RTOSDemo.bin`.

Rebuild the firmware with `make`. The following error occurs:

```
  CC    microej/src/LLMJVM_stub.c
  LD    gcc/RTOSDemo.axf
↪
↪                                microej/lib/microejapp.o: In function `__java__internStrings__end':
C:\Users\user\src\tuto-from-scratch\workspace\HelloWorld\com.
↪mycompany.Main\SOAR.o:(.text.soar+0x1b3e): undefined reference to `ist__mowana__vm__
↪GenericNativesPool____com__1is2t__1vm__1support__1lang__1SupportNumber__1parseLong'
C:\Users\user\src\tuto-from-scratch\workspace\HelloWorld\
↪com.mycompany.Main\SOAR.o:(.text.soar+0x1cea): undefined reference to `ist__mowana__
↪vm__GenericNativesPool____com__1is2t__1vm__1support__1lang__1SupportNumber__
↪1toStringLongNative'          C:\Users\user\src\tuto-from-scratch\workspace\HelloWorld\
↪com.mycompany.Main\SOAR.o:(.text.soar+0x1e3e): undefined reference to `ist__mowana__
↪vm__GenericNativesPool____com__1is2t__1vm__1support__1lang__1Systools__1appendInteger'
C:\Users\user\src\tuto-from-scratch\workspace\
↪HelloWorld\com.mycompany.Main\SOAR.o:(.text.soar+0x1f2a): undefined reference
↪to `ist__mowana__vm__GenericNativesPool____java__1lang__1System__1getMethodClass'
C:\Users\user\src\tuto-from-scratch\workspace\
↪HelloWorld\com.mycompany.Main\SOAR.o:(.text.soar+0x1e3e): undefined reference to `ist__
↪mowana__vm__GenericNativesPool____com__1is2t__1vm__1support__1lang__1Systools__1appen
... skip ...
C:\Users\user\src\tuto-from-scratch\workspace\
↪HelloWorld\com.mycompany.Main\SOAR.o:(.text.soar+0x31d6): undefined reference
↪to `ist__mowana__vm__GenericNativesPool____java__1lang__1System__1initializeProperties'
C:\Users\user\src\tuto-from-scratch\workspace\
↪HelloWorld\com.mycompany.Main\SOAR.o:(.text.soar+0x37b6): undefined reference
↪to `ist__mowana__vm__GenericNativesPool____java__1lang__1Thread__1storeException'
C:\Users\user\src\tuto-
↪from-scratch\workspace\HelloWorld\com.mycompany.Main\SOAR.o:(.text.soar+0x37c8):
↪undefined reference to `ist__microjvm__NativesPool____java__1lang__1Thread__1execClinit'
microej/lib/microejapp.
↪o: In function `___icetea___getSingleton___com__is2t__microjvm__mowana__VMTask':
C:\Users\user\src\tuto-from-scratch\workspace\HelloWorld\com.mycompany.Main\
↪SOAR.o:(.text.___icetea___getSingleton___com__is2t__microjvm__mowana__VMTask+0xc):
↪undefined reference to `com__is2t__microjvm__mowana__VMTask____getSingleton'
microej/lib/microejapp.
↪o: In function `___icetea___getSingleton___com__is2t__microjvm__IGreenThreadMicroJvm':
... skip ...
microej/lib/microejapp.o: In function `TRACE__record__event__u32x3__ptr':
C:\Users\
↪user\src\tuto-from-scratch\workspace\HelloWorld\com.mycompany.Main\SOAR.o:(.rodata.
↪TRACE__record__event__u32x3__ptr+0x0): undefined reference to `TRACE__default__stub'
microej/lib/microejapp.o: In function `TRACE__record__event__u32x4__ptr':
C:\Users\
↪user\src\tuto-from-scratch\workspace\HelloWorld\com.mycompany.Main\SOAR.o:(.rodata.
```

(continues on next page)

(continued from previous page)

```
↪TRACE__record__event__u32x4__ptr+0x0): undefined reference to `TRACE__default__stub'
microej/lib/microejapp.o:C:\Users\user\src\tuto-from-
↪scratch\workspace\HelloWorld\com.mycompany.Main\SOAR.o:(.rodata.TRACE__record__
↪event__u32x5__ptr+0x0): more undefined references to `TRACE__default__stub' follow
make: *** [makedefs:196: gcc/RTOSDemo.axf] Error 1
```

This error occurs because microejruntime.a refers to symbols in microejapp.o but is declared after in the linker command line. By default, the GNU LD linker does not search unresolved symbols into archive files loaded previously (see man ld for a description of the start-group option). To solve this issue, either invert the declaration of LIBS (put microejapp.o first) or guard the libraries declaration with --start-group and --end-group in makedefs. This training uses the later.

Listing 52: https://github.com/MicroEJ/FreeRTOS/commit/4b23ea2e77112f053368718d299ff8db826ddde1

```
diff --git a/FreeRTOS/Demo/CORTEX_
↪LM3S6965_GCC/makedefs b/FreeRTOS/Demo/CORTEX_LM3S6965_GCC/makedefs
-- a/FreeRTOS/Demo/CORTEX_LM3S6965_GCC/makedefs
+++ b/FreeRTOS/Demo/CORTEX_LM3S6965_GCC/makedefs
@@ -196,13 +196,13 @@ ifeq (${COMPILER}, gcc)
          echo ${LD} -T ${SCATTER_${notdir ${@:.axf=}}}    \
                  --entry ${ENTRY_${notdir ${@:.axf=}}} \
                  ${LDFLAGSgcc_${notdir ${@:.axf=}}}    \
-                 ${LDFLAGS} -o ${@} ${^}              \
-                 '${LIBC}' '${LIBGCC}';               \
+                  ${LDFLAGS} -o ${@} --start-group ${^} \
+                  '${LIBC}' '${LIBGCC}' --end-group;    \
       fi
       @${LD} -T ${SCATTER_${notdir ${@:.axf=}}}    \
             --entry ${ENTRY_${notdir ${@:.axf=}}} \
             ${LDFLAGSgcc_${notdir ${@:.axf=}}}    \
-            ${LDFLAGS} -o ${@} ${^}              \
-            '${LIBC}' '${LIBGCC}'
+             ${LDFLAGS} -o ${@} --start-group ${^} \
+             '${LIBC}' '${LIBGCC}' --end-group
       @${OBJCOPY} -O binary ${@} ${@:.axf=.bin}
 endif
```

Rebuild with make. The following error occurs:

```
 LD     gcc/RTOSDemo.axf
microej/lib/microejruntime.
↪a(VMCOREMicroJvm___131.o): In function `VMCOREMicroJvm___1131_____1__11046':
_131.
↪c:(.text.VMCOREMicroJvm___1131_____1__11046+0x20): undefined reference to `fmodf'
microej/lib/microejruntime.
↪a(VMCOREMicroJvm___131.o): In function `VMCOREMicroJvm___1131_____1__11045':
_131.c:(.text.VMCOREMicroJvm___1131_____1__11045+0x2c): undefined reference to `fmod'
microej/lib/microejruntime.a(iceTea__lang__Math.o): In function `iceTea__lang__Math____cos':
Math.c:(.text.iceTea__lang__Math____cos+0x2a): undefined reference to `cos'
microej/lib/microejruntime.a(iceTea__lang__Math.o): In function `iceTea__lang__Math____sin':
Math.c:(.text.iceTea__lang__Math____sin+0x2a): undefined reference to `sin'
microej/lib/microejruntime.a(iceTea__lang__Math.o): In function `iceTea__lang__Math____tan':
Math.c:(.text.iceTea__lang__Math____tan+0x2a): undefined reference to `tan'
microej/
```

(continues on next page)

(continued from previous page)

```
↪lib/microejruntime.a(iceTea_lang_Math.o): In function `iceTea_lang_Math____acos___D':
Math.c:(.text.iceTea_lang_Math____acos___D+0x18): undefined reference to `acos'
microej/
↪lib/microejruntime.a(iceTea_lang_Math.o): In function `iceTea_lang_Math____acos(void)':
Math.c:(.text.iceTea_lang_Math____acos___F+0x12): undefined reference to `acosf'
microej/lib/microejruntime.a(iceTea_lang_Math.o): In function `iceTea_lang_Math____asin':
Math.c:(.text.iceTea_lang_Math____asin+0x18): undefined reference to `asin'
microej/
↪lib/microejruntime.a(iceTea_lang_Math.o): In function `iceTea_lang_Math____atan':
Math.c:(.text.iceTea_lang_Math____atan+0x2): undefined reference to `atan'
microej/
↪lib/microejruntime.a(iceTea_lang_Math.o): In function `iceTea_lang_Math____atan2':
Math.c:(.text.iceTea_lang_Math____atan2+0x2): undefined reference to `atan2'
microej/lib/microejruntime.a(iceTea_lang_Math.o): In function `iceTea_lang_Math____log':
Math.c:(.text.iceTea_lang_Math____log+0x2): undefined reference to `log'
microej/lib/microejruntime.
↪a(iceTea_lang_Math.o): In function `iceTea_lang_Math_(...)(long long,  *)':
Math.c:(.text.iceTea_lang_Math____exp+0x2): undefined reference to `exp'
microej/lib/microejruntime.
↪a(iceTea_lang_Math.o): In function `iceTea_lang_Math_(char,...)(int, long)':
Math.c:(.text.iceTea_lang_Math____ceil+0x2): undefined reference to `ceil'
microej/
↪lib/microejruntime.a(iceTea_lang_Math.o): In function `iceTea_lang_Math____floor':
... skip ...
```

This error occurs because the Math library is missing. The rule for linking the firmware is defined in the file makedefs . Replicating how the libc is managed, the following patch finds the libm.a library and add it at link time:

Listing 53: https://github.com/MicroEJ/FreeRTOS/commit/a202f43948c41b848ebfbc8c53610028c454b66f

```
diff --git a/FreeRTOS/Demo/CORTEX_
↪LM3S6965_GCC/makedefs b/FreeRTOS/Demo/CORTEX_LM3S6965_GCC/makedefs
-- a/FreeRTOS/Demo/CORTEX_LM3S6965_GCC/makedefs
+++ b/FreeRTOS/Demo/CORTEX_LM3S6965_GCC/makedefs
@@ -102,
↪6 +102,11 @@ LIBGCC=${shell ${CC} -mthumb -march=armv6t2 -print-libgcc-file-name}
 #
 LIBC=${shell ${CC} -mthumb -march=armv6t2 -print-file-name=libc.a}

+#
+# Get the location of libm.a from the GCC front-end.
+#
+LIBM=${shell ${CC} -mthumb -march=armv6t2 -print-file-name=libm.a}
+
 #
 # The command for extracting images from the linked executables.
 #
@@ -197,12 +202,12 @@ ifeq (${COMPILER}, gcc)
                 --entry ${ENTRY_${notdir ${@:.axf=}}} \
                 ${LDFLAGSgcc_${notdir ${@:.axf=}}}    \
                 ${LDFLAGS} -o ${@} --start-group ${^} \
-                '${LIBC}' '${LIBGCC}' --end-group;    \
+                 '${LIBM}' '${LIBC}' '${LIBGCC}' --end-group; \
         fi
```

(continues on next page)

(continued from previous page)

```
	@${LD} -T ${SCATTER_${notdir ${@:.axf=}}}    \
	      --entry ${ENTRY_${notdir ${@:.axf=}}} \
	      ${LDFLAGSgcc_${notdir ${@:.axf=}}}    \
	      ${LDFLAGS} -o ${@} --start-group ${^} \
-	      '${LIBC}' '${LIBGCC}' --end-group
+	       '${LIBM}' '${LIBC}' '${LIBGCC}' --end-group;
	@${OBJCOPY} -O binary ${@} ${@:.axf=.bin}
 endif
```

Rebuild with `make`. The following error occurs:

```
  CC    microej/src/LLMJVM_stub.c
  LD    gcc/RTOSDemo.axf
/usr/lib/gcc/arm-none-
↪eabi/6.3.1/../../../arm-none-eabi/lib/thumb/libc.a(lib_a-sbrkr.o): In function `_sbrk_r':
/build/newlib-jo3xW1/newlib-2.4.0.20160527/build/arm-none-eabi/thumb/
↪newlib/libc/reent/../../../../../../../newlib/libc/reent/sbrkr.c:58: undefined reference to `_sbrk'
make: *** [makedefs:196: gcc/RTOSDemo.axf] Error 1
```

Instead of implementing a stub `_sbrk` function, this training uses the `libnosys.a` which provides stub implementation for various functions.

Listing 54: https://github.com/MicroEJ/FreeRTOS/commit/eb208d846f52c0695c06456b540e412ba96e640a

```
diff --git a/FreeRTOS/Demo/CORTEX_
↪LM3S6965_GCC/makedefs b/FreeRTOS/Demo/CORTEX_LM3S6965_GCC/makedefs
-- a/FreeRTOS/Demo/CORTEX_LM3S6965_GCC/makedefs
+++ b/FreeRTOS/Demo/CORTEX_LM3S6965_GCC/makedefs
@@␣
↪-107,6 +107,11 @@ LIBC=${shell ${CC} -mthumb -march=armv6t2 -print-file-name=libc.a}
 #
 LIBM=${shell ${CC} -mthumb -march=armv6t2 -print-file-name=libm.a}

+#
+# Get the location of libnosys.a from the GCC front-end.
+#
+LIBNOSYS=${shell ${CC} -mthumb -march=armv6t2 -print-file-name=libnosys.a}
+
 #
 # The command for extracting images from the linked executables.
 #
@@ -202,12 +207,12 @@ ifeq (${COMPILER}, gcc)
                       --entry ${ENTRY_${notdir ${@:.axf=}}} \
                       ${LDFLAGSgcc_${notdir ${@:.axf=}}}    \
                       ${LDFLAGS} -o ${@} --start-group ${^} \
-                      '${LIBM}' '${LIBC}' '${LIBGCC}' --end-group; \
+                       '${LIBNOSYS}' '${LIBM}' '${LIBC}' '${LIBGCC}' --end-group; \
         fi
 	@${LD} -T ${SCATTER_${notdir ${@:.axf=}}}    \
 	      --entry ${ENTRY_${notdir ${@:.axf=}}} \
 	      ${LDFLAGSgcc_${notdir ${@:.axf=}}}    \
 	      ${LDFLAGS} -o ${@} --start-group ${^} \
-	      '${LIBM}' '${LIBC}' '${LIBGCC}' --end-group;
+	       '${LIBNOSYS}' '${LIBM}' '${LIBC}' '${LIBGCC}' --end-group;
 	@${OBJCOPY} -O binary ${@} ${@:.axf=.bin}
 endif
```

Rebuild with `make`. The following error occurs:

```
  CC    microej/src/LLMJVM_stub.c
  LD    gcc/RTOSDemo.axf
/usr/lib/gcc/arm-
↪none-eabi/6.3.1/../../../arm-none-eabi/lib/thumb/libnosys.a(sbrk.o): In function `_sbrk':
/build/newlib-jo3xW1/newlib-2.4.0.20160527/build/arm-none-eabi/
↪thumb/libgloss/libnosys/../../../../../libgloss/libnosys/sbrk.c:21: undefined reference to `end'
make: *** [makedefs:201: gcc/RTOSDemo.axf] Error 1
```

The `_sbrk` implementation needs the `end` symbol to be defined. Looking at the implementation, the `end` symbol corresponds to the beginning of the C heap. This training uses the end of the `.bss` segment as the beginning of the C heap.

Listing 55: https://github.com/MicroEJ/FreeRTOS/commit/898f2e6cd492616b4ccaabc136cafa76ef038690

```
diff --git a/FreeRTOS/Demo/CORTEX_LM3S6965_
↪GCC/standalone.ld b/FreeRTOS/Demo/CORTEX_LM3S6965_GCC/standalone.ld
-- a/FreeRTOS/Demo/CORTEX_LM3S6965_GCC/standalone.ld
+++ b/FreeRTOS/Demo/CORTEX_LM3S6965_GCC/standalone.ld
@@ -64,5 +64,6 @@ SECTIONS
         *(.bss)
         *(COMMON)
         _ebss = .;
+        end = .;
     } > SRAM
 }
```

Then rebuild with `make`. There should be no error. Finally, run the firmware in QEMU with the following command:

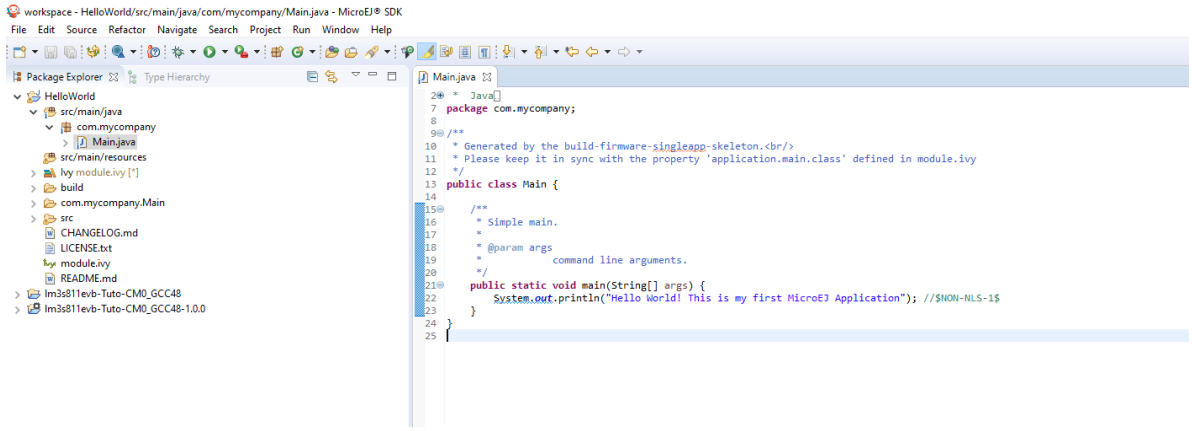
```
qemu-system-arm -M lm3s6965evb -nographic -kernel gcc/RTOSDemo.bin
```

```
Hello, World! printf function is working.
Hello World!
QEMU: Terminated // press Ctrl-a x to end the QEMU session
```

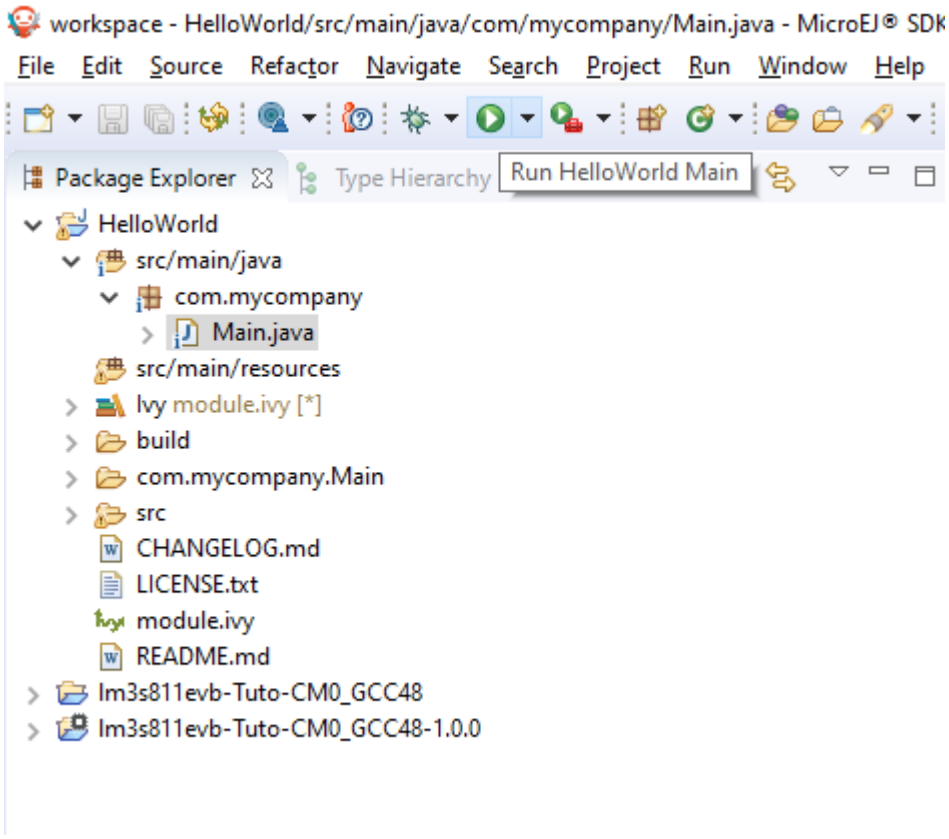
The first `Hello, World!` is from the `main.c` and the second one from the MicroEJ Application.

To make this more obvious:

1. Update the MicroEJ Application to print `Hello World! This is my first MicroEJ Application`



2. Rebuild the MicroEJ Application



On success, the following message appears in the console:

```
================ [ Initialization Stage ] ================
Platform␣
↪connected to BSP location 'C:\Users\user\src\tuto-from-scratch\FreeRTOS\FreeRTOS\
↪Demo\CORTEX_LM3S6965_GCC' using application option 'deploy.bsp.root.dir'.
================ [ Launching SOAR ] ================
================ [ Launching Link ] ================
================ [ Deployment ] ================
MicroEJ files for the 3rd-party BSP project are generated to 'C:\
↪Users\user\src\tuto-from-scratch\workspace\HelloWorld\com.mycompany.Main\platform'.
The MicroEJ application (microejapp.o) has been deployed to: 'C:\Users\user\src\
↪tuto-from-scratch\FreeRTOS\FreeRTOS\Demo\CORTEX_LM3S6965_GCC\microej\lib'.
The MicroEJ platform library (microejruntime.a) has been deployed to: 'C:\Users\user\src\
↪tuto-from-scratch\FreeRTOS\FreeRTOS\Demo\CORTEX_LM3S6965_GCC\microej\lib'.
The MicroEJ platform header files (*.h) have been deployed to: 'C:\Users\user\src\
↪tuto-from-scratch\FreeRTOS\FreeRTOS\Demo\CORTEX_LM3S6965_GCC\microej\inc'.
================ [ Completed Successfully ] ================

SUCCESS
```

3. Then rebuild and run the firmware:

```
$ make && qemu-system-arm -M lm3s6965evb -nographic -kernel gcc/RTOSDemo.bin

  LD    gcc/RTOSDemo.axf
Hello, World! printf function is working.
```

(continues on next page)

(continued from previous page)

```
Hello World! This is my first MicroEJ Application
QEMU: Terminated
```

Congratulations!

At this point of the training:

- The MicroEJ Platform is connected to the BSP (BSP partial connection).
- The MicroEJ Application is deployed within a known location of the BSP (in `microej/` folder).
- The FreeRTOS LM3S6965 port:
  - provides the minimal Low Level API to run the MicroEJ Application
  - compiles and links FreeRTOS with the MicroEJ Application and MicroEJ Runtime
  - runs on QEMU

The next steps recommended are:

- Complete the implementation of the Low Level APIs (implement all functions in `LLMJVM_impl.h`).
- Validate the implementation with the PQT Core.
- Follow the *Create MicroEJ Platform Build and Run Scripts* training to get this MicroEJ Platform fully automated for build and execution.

Create MicroEJ Platform Build and Run Scripts

This training describes all the steps to create *MicroEJ Platform* build and run scripts and shows how to use them.

Intended Audience

The audience for this document is Platform engineers who want to

- validate their MicroEJ Platform using automated *MicroEJ testsuites*.
- prepare their MicroEJ Platform for automated builds and continuous integration using *MicroEJ Module Manager*.
- ease *MicroEJ Standalone Application* development by simplifying the Firmware build for Java developers.
- configure their MicroEJ Platform with full *BSP connection*.

Prerequisites

This training is a direct continuation of *Create a MicroEJ Firmware From Scratch* training. It should have been completed before starting this one.

Introduction

Build and Run scripts are normalized entry points to

- build a MicroEJ Firmware linked to the Board Support Package,
- deploy and run the Firmware on a device.

External tools only need to run these scripts without additional knowledge about the toolchain or deployment tools.

See *Build Script File* and *Run Script File* sections for more information about these scripts. Script examples are provided in Platform Qualification Tools repository.

Overview

In the previous *Create a MicroEJ Firmware From Scratch* training, the final binary is produced by invoking `make` in the FreeRTOS BSP. The command to type is dependant of the toolchain used. The Firmware is then executed in QEMU but could have been instead flashed to a device with another specific command. This training explain how to write *build* and *run* scripts for these two tasks.

The next sections will

- describe step-by-step how to create the build and run scripts both for unix-like systems (Bash scripts) and Windows systems (batch files). These scripts automate Firmware build and execution in QEMU as presented in *Create a MicroEJ Firmware From Scratch* training.
- show a practical usage of these scripts in a MicroEJ development flow. This will allow to configure a MicroEJ Standalone Application to build the Firmware in MicroEJ SDK.

Finally, this training describes how to convert the MicroEJ Platform from partial BSP connection to full BSP connection.

Create Build and Run Scripts

This section describes how to write build and run scripts.

There are two scripts:

1. `build.[sh|bat]` which calls the C toolchain to build and link the Firmware file. It also ensures that the output file is called `application.out` and is located in the directory from where the script was called.
2. `run.[sh|bat]` which deploys and runs `application.out` on the device. In this training, it will only run the Firmware with QEMU instead of flashing it on real hardware.

Each of these scripts come in two flavors: `.sh` for unix-like systems, and `.bat` for Windows systems.

First, create a `microej/scripts` directory in BSP project:

```
$ pwd
/mnt/c/
↪Users/user/src/tuto-from-scratch/FreeRTOS/FreeRTOS/Demo/CORTEX_LM3S6965_GCC
$ mkdir microej/scripts
```

Note: The scripts created in the next sections will be put in this directory.

Create *build.sh* and *run.sh* Scripts

Warning: Make sure the build and run scripts have the execution permission.

1. Create a file called `build.sh` in the `microej/scripts` directory with the following content:

```
#!/bin/bash

# Save application current directory and jump one level above scripts
CURRENT_DIRECTORY=$(pwd)

# Move to the directory where the Makefile is
cd $(dirname "$0")/../..

# Build the firmware
make

# Copy output the the current directory while renaming it
cp gcc/RTOSDemo.bin  $CURRENT_DIRECTORY/application.out

# Restore application directory
cd $CURRENT_DIRECTORY/
```

2. Verify that the script successfully built your Firmware and put it in the current directory with the name `application.out` .

```
$ pwd
/mnt/c/
↪Users/user/src/tuto-from-scratch/FreeRTOS/FreeRTOS/Demo/CORTEX_LM3S6965_GCC
$ make clean
$ microej/scripts/build.sh
  CC    init/startup.c
  CC    main.c
  CC    ../../Source/list.c
  CC    ../../Source/queue.c
  CC    ../../Source/tasks.c
[..]
  130 |  __attribute__
↪__( ( always_inline ) ) static inline uint8_t ucPortCountLeadingZeros( uint32_t ulBitmap )
      |                                                ^~~~~~~~~~~~~~~~~~~~~~~~
  LD    gcc/RTOSDemo.axf
$ ls *.out
application.out
```

3. Check that `application.out` successfully runs with QEMU:

```
$ qemu-system-arm -M lm3s6965evb -nographic -kernel application.out
Hello, World! printf function is working.
Hello World!
QEMU: Terminated // press Ctrl-a x to end the QEMU session
```

4. Create a file called run.sh in the microej/scripts directory with the following content:

```
#!/bin/bash

# Add some text to the console before launch
echo -e "\033[0;32m## Start application in QEMU."
echo -e "## Use 'Ctrl-a x' to quit.\e[0m"

# Launch application with QEMU
qemu-system-arm -M lm3s6965evb -nographic -kernel application.out
```

5. We can now run the Firmware we just built with the run.sh script:

```
$ pwd
/mnt/c/
↪Users/user/src/tuto-from-scratch/FreeRTOS/FreeRTOS/Demo/CORTEX_LM3S6965_GCC
$ microej/scripts/run.sh
## Start application in QEMU.
## Use 'Ctrl-a x' to quit.
Hello, World! printf function is working.
Hello World!
```

Note: This script is very simple because our Firmware is just run with QEMU instead of real hardware. To deploy the Firmware on a device, the script would have to setup and call a flash tool. See for instance the build and run scripts for Espressif-ESP-WROVER-KIT-V4.1.

Create *build.bat* and *run.bat* Scripts

As our toolchain has only be configured for Linux in WSL, we create wrappers that call shell scripts through WSL. We could also decide to directly invoke QEMU for Windows instead. This is just a implementation choice for this Platform.

1. Create a file called build.bat in the microej/scripts directory with the following content:

```
@echo off
SETLOCAL ENABLEEXTENSIONS

REM Reset ERRORLEVEL between multiple run in the same shell
SET ERRORLEVEL=0

REM Save application current directory and jump to scripts directory
SET CURRENT_DIRECTORY=%CD%
CD "%~dp0"

REM Get the script directory in a Unix path format
FOR /F %%i in ('WSL pwd') DO SET SCRIPT_DIRECTORY=%%i

REM Restore application directory
```

(continues on next page)

(continued from previous page)

```
CD %CURRENT_DIRECTORY%

REM Run the bash build script with WSL
WSL %SCRIPT_DIRECTORY%/build.sh

IF %ERRORLEVEL% NEQ 0 (
    EXIT /B %ERRORLEVEL%
)
```

2. Calling this script in PowerShell should produce the following result:

```
PS C:\Users\user\src\tuto-from-scratch\
↪FreeRTOS\FreeRTOS\Demo\CORTEX_LM3S6965_GCC> microej\scripts\build.bat
  CC    init/startup.c
  CC    main.c
  CC    ../../Source/list.c
  CC    ../../Source/queue.c
  CC    ../../Source/tasks.c
 [...]
  CC    microej/src/LLMJVM_stub.c
  LD    gcc/RTOSDemo.axf
Current DIR /mnt/c/Users/user/src/
↪tuto-from-scratch/FreeRTOS/FreeRTOS/Demo/CORTEX_LM3S6965_GCC/microej/scripts
        1 file(s) moved.
```

Note: This prints the full build output if it is the first build (or after a `make clean`) otherwise it prints `make: Nothing to be done for 'all'`.

3. Create a file called `run.bat` in the `microej/scripts` directory with the following content:

```
@echo off
SETLOCAL ENABLEEXTENSIONS

REM Reset ERRORLEVEL between multiple run in the same shell
SET ERRORLEVEL=0

REM Save application current directory and jump to scripts directory
SET CURRENT_DIRECTORY=%CD%
CD "%~dp0"

REM Get the script directory in a Unix path format
FOR /F %%i in ('WSL pwd') DO SET SCRIPT_DIRECTORY=%%i

REM Restore application directory
CD %CURRENT_DIRECTORY%

REM Run the bash run script with WSL
WSL %SCRIPT_DIRECTORY%/run.sh

IF %ERRORLEVEL% NEQ 0 (
   EXIT /B %ERRORLEVEL%
)
```

4. Calling this script in PowerShell should produce the following result:

```
C:\Users\user\src\tuto-
↪from-scratch\FreeRTOS\FreeRTOS\Demo\CORTEX_LM3S6965_GCC\application.out
1 file(s) copied.
## Start application in QEMU.
## Use 'Ctrl-a x' to quit.
Hello, World! printf function is working.
Hello World!
```

Use Build Script in MicroEJ SDK

In this section, we illustrate how build script is used in practice to ease the Firmware build for Java developers in MicroEJ SDK.

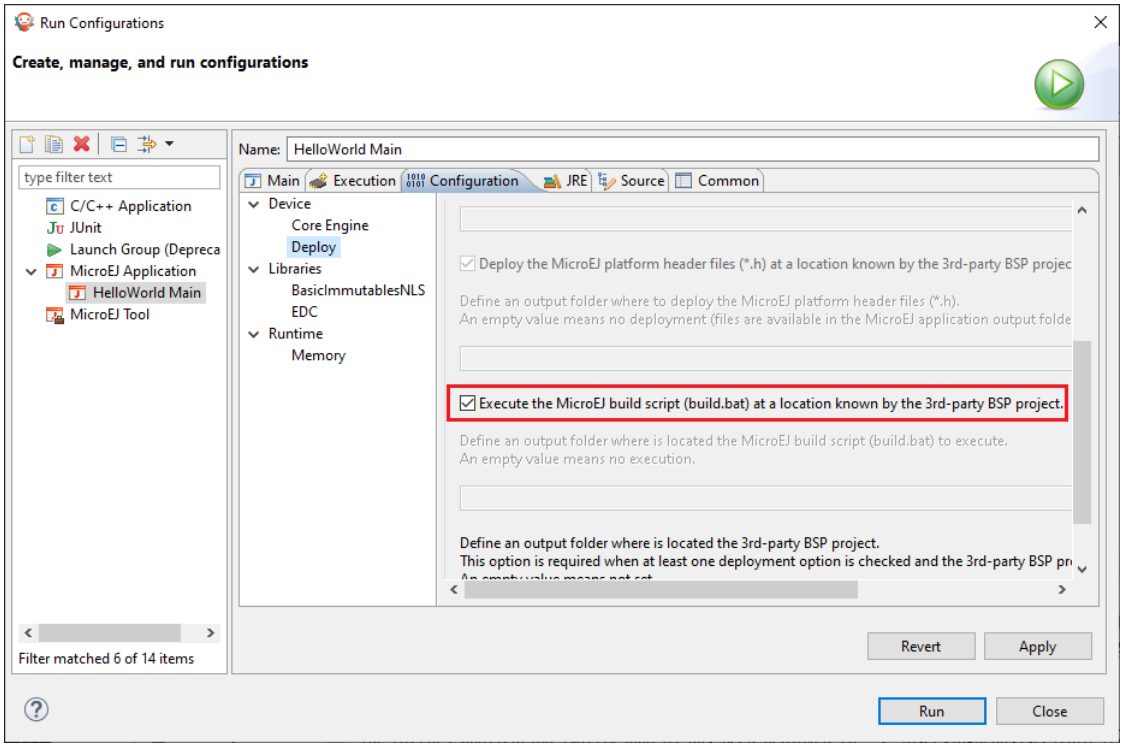
We will configure a MicroEJ Standalone Application to enable full Firmware build (application + BSP + link) when building the *HelloWorld* application.

We will then configure a full BSP connection. This will remove the need to configure the path of the BSP root directory as a MicroEJ Standalone Application option. Please refer to *BSP connection cases* section and *BSP connection options* for more details.

Note: Build and run scripts do not require to configure a full BSP connection. This last part has only be added to allow a MicroEJ Standalone Application project to be built independently from the BSP.

Build Firmware from MicroEJ SDK

1. Right click on the HelloWorld application project
2. In the menu, select Run As > Run Configurations…
3. Select the Configuration tab
4. Select Device > Deploy entry in the configurations menu
5. Check Execute the MicroEJ script (build.bat) at the location known by the 3rd-party BSP project checkbox



6. Click on the `Run` button. It should print the following:

```
================ [ Initialization Stage ] ================
Platform
↪connected to BSP location 'C:\Users\user\src\tuto-from-scratch\FreeRTOS\FreeRTOS\
↪Demo\CORTEX_LM3S6965_GCC' using application option 'deploy.bsp.root.dir'.
[INFO ] Launching in Evaluation mode. Your UID is 0120202834374C4A.
================ [ Launching SOAR ] ================
================ [ Launching Link ] ================
================ [ Deployment ] ================
MicroEJ files for the 3rd-party BSP project are generated to 'C:\
↪Users\user\src\tuto-from-scratch\workspace\HelloWorld\com.mycompany.Main\platform'.

FAIL
The following error occurred while executing this line:
C:\Users\user\src\tuto-from-scratch\workspace\lm3s811evb-Platform-CM0_GCC48-
↪0.0.1\source\scripts\deploy.xml:30: The following error occurred while executing this line:
C:\Users\user\src\tuto-from-scratch\workspace\lm3s811evb-Platform-CM0_GCC48-0.0.
↪1\source\scripts\deployInBSP.xml:97: The following error occurred while executing this line:
C:\Users\user\src\tuto-from-scratch\workspace\lm3s811evb-Platform-CM0_GCC48-0.0.
↪1\source\scripts\deployInBSP.xml:260: Option 'deploy.bsp.microejscript' is enabled but this
↪Platform does no define a well-known location. Either update the Platform configuration
↪(option 'deploy.bsp.microejscript.relative.dir' in 'bsp/bsp.properties') or disable this option.
```

7. Edit the file `bsp/bsp.properties` as follow:

```
# Specify BSP external scripts files ('build.bat' and 'run.bat') parent directory.
# This is a '/' separated directory relative to 'bsp.root.dir'.
microejscript.relative.dir=microej/scripts
```

8. Rebuild your Platform (right-click on the platform configuration project and click on `Build Module`)

9. Run the *HelloWorld* launcher once again. This should print the following result:

```
================ [ Initialization Stage ] ================
Platform connected to BSP location 'C:\Users\user\src\tuto-from-scratch\FreeRTOS\
↪FreeRTOS\Demo\CORTEX_LM3S6965_GCC' using platform option 'deploy.bsp.root.dir'.
[INFO ] Launching in Evaluation mode. Your UID is 0120202834374C4A.
↪================ [ Launching SOAR ] ================
================ [ Launching Link ] ================
================ [ Deployment ] ================
MicroEJ files for the 3rd-party BSP project are generated␣
↪to 'C:\Users\user\Workspaces\_test_fw_tuto\HelloWorld\com.mycompany.Main\platform'.
The MicroEJ application (microejapp.o) has been deployed to: 'C:\Users\user\src\
↪tuto-from-scratch\FreeRTOS\FreeRTOS\Demo\CORTEX_LM3S6965_GCC\microej\lib'.
The MicroEJ platform library (microejruntime.a) has been deployed to: 'C:\Users\user\src\
↪tuto-from-scratch\FreeRTOS\FreeRTOS\Demo\CORTEX_LM3S6965_GCC\microej\lib'.
The MicroEJ platform header files (*.h) have been deployed to: 'C:\Users\user\src\
↪tuto-from-scratch\FreeRTOS\FreeRTOS\Demo\CORTEX_LM3S6965_GCC\microej\inc'.
Execution of script 'C:\Users\user\src\tuto-from-scratch\FreeRTOS\
↪FreeRTOS\Demo\CORTEX_LM3S6965_GCC\microej\scripts\build.bat' started...
LD    gcc/RTOSDemo.axf
Current␣
↪DIR /mnt/c/Users/user/Workspaces/_test_fw_tuto/HelloWorld/com.mycompany.Main
Execution of script 'C:\Users\user\src\tuto-from-scratch\
↪FreeRTOS\FreeRTOS\Demo\CORTEX_LM3S6965_GCC\microej\scripts\build.bat' done.
================ [ Completed Successfully ] ================

SUCCESS
```

Reading the traces, we see that the *HelloWorld* application (*microejapp.o*) and the MicroEJ Platform library (*microejruntime.a*) have been deployed to the suitable BSP location. Then the `build.bat` script has been executed to rebuild the BSP and link the Firmware. The output is the `application.out` binary that can be flashed on the device (or run on QEMU).

Convert from partial BSP connection to full BSP connection (optional)

In this section, we configure the BSP root directory in the Platform. Such configuration is called *full BSP connection*: the MicroEJ Platform includes the BSP, and any MicroEJ Standalone Application can be built against this MicroEJ Platform without extra configuration.

When launching the `HelloWorld` application from MicroEJ SDK, the launcher knows how to find the BSP because we have configured its path in `HelloWorld/build/emb.properties` file which is imported in the launcher (this file has been configured in *Create a MicroEJ Firmware From Scratch* training).

1. Cut `deploy.bsp.root.dir` property value from `HelloWorld/build/emb.properties` file
2. Paste the value in `bsp/bsp.properties` as follow:

```
# Specify the BSP root directory.
↪ Can use ${project.parent.dir} which target the parent of platform configuration project
# For example, '$
↪{workspace}/${project.prefix}-bsp' specifies a BSP project beside the '-configuration' project
root.dir=[absolute_path] to FreeRTOS\\FreeRTOS\\Demo\\CORTEX_LM3S811_GCC
```

3. Rebuild your MicroEJ Platform (right-click on the platform configuration project and click on `Build Module`)

   The MicroEJ Platform is now fully connected to the BSP.

4. Launch HelloWorld project from Eclipse launcher, it should print the following result:

```
================ [ Initialization Stage ] ================
Platform␣
↪connected to BSP location 'C:\Users\user\src\tuto-from-scratch\FreeRTOS\FreeRTOS\
↪Demo\CORTEX_LM3S6965_GCC' using platform option 'root.dir' in 'bsp/bsp.properties'.
[INFO ] Launching in Evaluation mode. Your UID is 0120202834374C4A.
================ [ Launching SOAR ] ================
================ [ Launching Link ] ================
================ [ Deployment ] ================
MicroEJ files for the 3rd-party BSP project are generated to 'C:\
↪Users\user\src\tuto-from-scratch\workspace\HelloWorld\com.mycompany.Main\platform'.
The MicroEJ application (microejapp.o) has been deployed to: 'C:\Users\user\src\
↪tuto-from-scratch\FreeRTOS\FreeRTOS\Demo\CORTEX_LM3S6965_GCC\microej\lib'.
The MicroEJ platform library (microejruntime.a) has been deployed to: 'C:\Users\user\src\
↪tuto-from-scratch\FreeRTOS\FreeRTOS\Demo\CORTEX_LM3S6965_GCC\microej\lib'.
The MicroEJ platform header files (*.h) have been deployed to: 'C:\Users\user\src\
↪tuto-from-scratch\FreeRTOS\FreeRTOS\Demo\CORTEX_LM3S6965_GCC\microej\inc'.
Execution of script 'C:\Users\user\src\tuto-from-scratch\FreeRTOS\
↪FreeRTOS\Demo\CORTEX_LM3S6965_GCC\microej\scripts\build.bat' started...
  LD     gcc/RTOSDemo.axf
Current DIR /mnt/c/Users/user/src/
↪tuto-from-scratch/FreeRTOS/FreeRTOS/Demo/CORTEX_LM3S6965_GCC/microej/scripts
Execution of script 'C:\Users\user\src\tuto-from-scratch\
↪FreeRTOS\FreeRTOS\Demo\CORTEX_LM3S6965_GC  C\microej\scripts\build.bat' done.
================ [ Completed Successfully ] ================

SUCCESS
```

Note: You can notice the difference in the second line of the trace that now says that the connection is using platform option root.dir' in 'bsp/bsp.properties' instead of using platform option 'deploy.bsp.root.dir' in the previous launch.

The application launcher does not know anymore where the BSP is located. Nevertheless it still builds a Firmware ready to be flashed. The link to the BSP is now configured in the MicroEJ Platform. Any MicroEJ Standalone Application can be built against this MicroEJ Platform with no BSP specific option.

Going Further

- More about build and run scripts in *Build Script File* and *Run Script File* sections
- Some build scripts examples from Platform Qualification Tools
- Follow the *Configure and Run the Testsuite* documentation to learn how to run an automated testsuite
- Perform the *Setup an Automated Build using Jenkins and Artifactory* training to learn how to automate the build of a MicroEJ Platform module

Unit 1: Create a MicroEJ Firmware From Scratch

Step by step hands-on that explains how to create a VEE Port for a custom device:

MICROEJ SDK 5

MICROEJ SDK 6

- *Create a MicroEJ Firmware From Scratch*
  - Coming soon!

Unit 2: Create VEE Port Build and Run Scripts

Continuation of Unit 1, create VEE Port Build and Run scripts to ease Application development and VEE Port validation:

MICROEJ SDK 5

MICROEJ SDK 6

- *Create MicroEJ Platform Build and Run Scripts*
  - Coming soon!

Qualify & Test your VEE Port

Description

This training describes how to validate the integration of MICROEJ VEE on a device using PQT (Port Qualification Tools).

VEE Port Qualification is the process of validating the conformance of the Abstraction Layer that implements the *Low Level APIs* of a Foundation Library.

What you will learn:

- Reminder of Foundation Libraries architecture.
- Testsuite role in Abstraction Layer validation.
- Running a Testsuite on a Device

Intended Audience

This training is designed for VEE Port developers who want to validate an Abstraction Layer implementation or understand how to automatically run a Testsuite on their device.

Prerequisites

To get the most out of this training, participants should have:

- A good understanding of the *Overview* section.
- A basic knowledge of the *MicroEJ Glossary*.
- Access to the ESP-WROVER-KIT V4.1 Evaluation Kit.

Training Resources

Unit 1: Introduction to Port Qualification Tool

Read the Introduction section of the *VEE Port Qualification* documentation.

Unit 2: Run a Testsuite on a Device

Step-by-step hands-on to run a VEE Port Testsuite on a device using the Port Qualification Tools:

MICROEJ SDK 5

MICROEJ SDK 6

- *Run the FS Testsuite on ESP32-WROVER VEE Port*
  - Coming soon!

VEE Port Programming Practices

Description

This page gives a collection of recommended practices that are essential when porting VEE to new hardware and existing embedded software stacks.

Intended Audience

This training is designed for embedded developers who want to enhance the quality and security of their C code.

Prerequisites

To get the most out of this training, participants should have a good knowledge of the C programming language and Board Support Package (BSP) development.

A Word about Security

This paragraph serves as a quick overview, outlining the processes, design philosophies, and concrete mechanisms that should consider to ensure system integrity and security. Key enforcements should include:

- code reviews (code should be created by several entities in collaboration to ensure quality, maintainability and security),
- static analysis (use tools such as cppcheck or SonarQube),
- threat modeling (what are you trying to protect, from whom, it could also be beneficial to do some penetration testing),
- memory and thread separation (isolate your treatments, segregate your information based on applicative needs),
- and cryptographic support (remove out-of-date standards, ensure proper storage of keys and certificates, ensure sensitive data is not transiting via easily accessible unsecured memories, etc.).

You should also consider covering:

- system-level concerns (boot, OTA updates, root of trust),
- vulnerability reporting and mitigation strategies (CVEs management and vulnerability recovery, use SBOM tools for automatic detection),
- and certification strategies (software-only, hardware-anchored, or full product certification).

You should aim to guide developers and architects to integrating security practices throughout your codebase and ecosystem on a day-to-day basis.

Recommended Practices

Embedded software presents several challenges, like traditional software but with the added pressure due to resource constraints. However, many common issues can be mitigated by following security rules and design principles. MicroEJ already follows certain design principles, for your own VEE Port development, it is recommended to read the following links from recognized sources:

- https://cyber.gouv.fr/sites/default/files/2022/04/anssi-guide-rules_for_secure_c_language_software_development-v1.4.pdf
- https://docenti.ing.unipi.it/~a009435/issw/extra/c-coding-standard.pdf
- https://www.linuxfoundation.org/research/cra-compliance-best-practices?hsLang=en
- https://training.linuxfoundation.org/training/cybersecurity-essentials-lfc108/
- https://partner-security.withgoogle.com/docs/iot_requirements

Abstraction Layers Development

Error Handling in Native Methods

This section describes the best practices for handling errors in native methods.

Here are the common ways of reporting errors to the Managed Code in a native method:

- Throw an exception with SNI_throwNativeException or SNI_throwNativeIOException.
- Return an error code.

  For implementation errors or invalid requests (for example, a negative size array or a NULL pointer, more generally anything that is outside the validity domain of the method): call SNI_throwNativeException. In this case, the error is not caught by the Managed Code (uncaught exception in Java). Here are the good practices when using this function:

- The error string must be wrapped in a macro that can be disabled to save footprint.
- The error code can be anything that helps the developer to fix the issue (not necessarily errors documented in the Abstraction Layer API; it can also be a backend error code).
- A log message can be added to describe the error; the log string must also be wrapped in a macro that can be disabled to save footprint.
- Check that the return statement of SNI_throwNativeException is SNI_OK; you can do it with an assert. Return SNI_IGNORED_RETURNED_VALUE.

  For errors that must be handled by the Java implementation, use one of the two following methods. In each case, the method used to handle errors and the possible error codes must be specified in the Abstraction Layer API:

- Call SNI_throwNativeIOException. This error is primarily used for Input/Output errors, but it can be used in other cases when an exception needs to be thrown. In this case, the error is caught by the Managed Code (caught exception in Java).
  - Use an error code that is documented in the Abstraction Layer API. Keep a reduced number of codes; each error code implies a different processing on the Managed Code side. Use -1 for the default error.
  - The error string must be wrapped in a macro that can be disabled to save footprint. It can contain a backend-specific message (for example, a strerror() string).
  - A log message can be added to specify the error and add a backend error code; again, the log string must be wrapped in a macro that can be disabled to save footprint.
  - Check that the return statement of SNI_throwNativeIOException is SNI_OK; you can do it with an assert. Return SNI_IGNORED_RETURNED_VALUE.
- Use the return code of the native function.
  - Use only error codes documented in the Abstraction Layer API.
  - As an error implies a negative return value, this cannot be used for functions that normally return negative values (for example, temperature sensors).

  For more information on how to handle exceptions in the Managed Code, please refer to *Exceptions* chapter.

Debug a HardFault

Description

In this training, you will get insights on how to debug a hardfault.

Intended Audience

This training is designed for VEE Port developers who want to debug a hardfault.

Prerequisites

To get the most out of this training, participants should have:

- A good knowledge of embedded software development on Microcontrollers.
- A good understanding of *Simple Native Interface Specification (SNI)* concepts.

Introduction

When the application crashes, it can result from a HardFault triggered by the MCU.

The following sections explain:

1. What are exceptions, HardFaults, and the exception handler.
2. What to do in case of Memory Corruptions.
3. What to do when a HardFault occurs.

Useful Resources

- IAR System: Debugging a HardFault on Cortex-M https://mypages.iar.com/s/article/Debugging-a-HardFault-on-Cortex-M
- ESP-IDF Programming Guide: Fatal Errors https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/fatal-errors.html
- Using Cortex-M3/M4/M7 Fault Exceptions MDK Tutorial http://www.keil.com/appnotes/files/apnt209.pdf

Exceptions, HardFaults And Exception Handler

From ARM Architecture Reference Manual

An exception causes the processor to suspend program execution to handle an event, such as an externally generated interrupt or an attempt to execute an undefined instruction. Exceptions can be generated by internal and external sources. Normally, when an exception is taken, the processor state is preserved immediately, before handling the exception. This means that, when the event has been handled, the original state can be restored and program execution resumed from the point where the exception was taken.

For example, an *IRQ request* is an exception that can be recovered by handling the hardware request properly. On the other hand, an *Undefined Instruction* exception suggests a more severe system failure that might not be recoverable.

The exceptions that cannot be recovered are named **HardFaults**.

From ARM Architecture Reference Manual

When an exception is taken, processor execution is forced to an address that corresponds to the type of exception. This address is called the **exception vector** for that exception.

The code pointed by the exception vector is named **exception handler**. Therefore, a dedicated exception handler can be configured for all exceptions, including HardFaults.

Possible exceptions can be:

- Data Abort exception (access to unknown address)
- Undefined Instruction exception (execute code that is not valid)
- …

Check the hardware documentation for the complete list of exceptions.

What To Do In Exception Handlers?

For all HardFault handlers, the following data are available and must be printed:

- Name and value of all registers available
- Name of the handler
- Address of the failing instruction

Optionally:

- Content of the stack
- Call function `LLMJVM_dump` (from `LLMJVM.h`) to display the Core Engine state (see *Dump the State of the Core Engine*)

Refer to the architecture documentation for how to configure the exception interrupt vector.

Memory Protection Unit (MPU)

A Memory Protection Unit (MPU) is a hardware unit that provides memory protection. An MPU allows privileged software to define memory regions and their policy. The policy describes who can access the memory.

For example, configure the heap and stack of a task to be accessible from the task itself only. The MPU generates an exception if another task or a device driver attempts to access the memory region.

If applicable, configure the MPU should to protect the application.

- Check the RTOS documentation if it supports MPU.

  For example, FreeRTOS includes FreeRTOS-MPU https://www.freertos.org/Security/04-FreeRTOS-MPU-memory-protection-unit.

- Configure the MPU to configure the access to the JVM heap and stack to prevent any other native threads from altering this area. Refer to *this section* for the list of section names defined by the Core Engine.

Memory Corruption

Memory corruption can result in the following symptoms:

- The address of the failing instruction is in a data section.
- The trace is incomplete or incorrect.
- The address of the failing instruction is located in the Garbage Collector (GC).

The cause(s) of a memory corruption can be:

- A native (C) function has a bug and writes to an incorrect memory location
- A native stack overflow
- A native heap overflow
- A device incorrectly initialized or misconfigured.
- …

When the HardFault occurs in the Core Engine task, one of its internal structures may be corrupted (such as the Managed Heap or Threads stacks). Add `LLMJVM_checkIntegrity` call in checkpoints of the BSP code to identify the timeslot of the memory corruption. Typically, you can check a native with:

```
void Java_com_mycompany_MyClass_myNativeFunction(void) {
    int32_t crcBefore = LLMJVM_checkIntegrity();
    myNativeFunctionDo();
    int32_t crcAfter = LLMJVM_checkIntegrity();
    if(crcBefore != crcAfter){
        // Corrupted memory in Core Engine internal structures
        while(1);
    }
}
```

Investigation

Determine which memory regions are affected and determine which components are responsible for the corruption.

- List all the memories available and their specifics:
  - Access mode (addressable, DMA, …)
  - Cache mechanism? L1, L2
- Is low-power enabled for CPU and peripherals? Is the memory disabled/changed to save power?
- Get the memory layout of the project:
  - What are the code sections for the BSP and the Application?
  - Where are the BSP stack and heap? What about the Application stack and heap?

  - Where is the Java Immortals Heap?
  - Where are the Java strings?
  - Where is the MicroEJ UI buffer?
  - Besides the Java immortals, what are the other intersection point between the Java application and the BSP? (e.g., a temporary RAM buffer for JPEG decoder).
  - Please refer to the *Core Engine Link section* to locate the Application sections, and to the *Standalone Application Options* for their sizes.
- Implement a CRC of the *hot sections* when entering/leaving all natives. *Hot Sections* are memory sections used by both Java code and native code (e.g., C or ASM).
- Move the C stack at the beginning of the memory to trigger a crash when it overflows (instead of corrupting the memory).

When a HardFault Occurs

Extract Information and Coredump

Attach an embedded debugger and get the following information:

- stack traces and registers information for each stack frame
- memory information
  - the whole memory, if possible
  - otherwise, get the *hot sections*
    * BSP and Managed Heap and stack
    * UI buffer
    * Immortals Heap
    * sections where the Java application and BSP are working together
- *Trigger Core Engine Dump From Debugger*
- Check which function is located at the address inside the PC register.
  - It can be done either in Debug mode or by searching inside the generated .map file.

Memory Dump Analysis

- Run the Heap Dumper to check the application heap has not been corrupted.
- Make sure the native stack is not full (usually, there shall have the remaining initialization patterns in memory on top of the stack, such as `0xDEADBEEF`)

Trigger a Core Engine Dump

LLMJVM_dump function is provided by LLMJVM.h . This function prints a snapshot of the Core Engine's current state. The output includes the list of threads and their stack traces.

See *this section* to learn more about LLMJVM_dump .

Delegate Blocking Operations using Async Worker

Description

In this training, you will get an introduction to the Async Worker C Component that helps to delegate blocking operations from VEE RTOS context to another RTOS task context.

Intended Audience

This training is designed for VEE Port developers who want to explore blocking native method implementation in depth.

Prerequisites

To get the most out of this training, participants should have:

- A good knowledge of C programming language.
- A good understanding of *Unit 1: C / Managed Code Communication with MICROEJ VEE* training (especially the Blocking Functions and Asynchronous Code section).

Training Resources

Unit 1: Async Worker Overview

-

Unit 2: Implementation Guide

The Async Worker C Component is implemented in several VEE Port Examples.

Refer to the following guide to implement asynchronously SNI functions using Async Worker:

- Async Worker Implementation Guide

3.10.4 For Kernel Developers

Get Started with Kernel

This Getting Started is a step-by-step guide explaining how to build a Multi-Sandbox Executable for the NXP i.MXRT1170 Evaluation Kit.

This Executable will be built from the Kernel GREEN sources.

The Multi-Sandbox Executable built in this Getting Started can then be used to complete the *Get Started with Sandboxed Applications* Getting Started.

During this Getting Started, you will learn how to:

- Get the Kernel GREEN,
- Configure it,
- Build the project and get the modules: Virtual Device, Executable, Javadoc, …
- Run a Sandboxed Application project on this Multi-Sandbox Executable.

In case you are not familiar with MicroEJ, please visit Discover MicroEJ to understand the principles of our technology.

Prerequisites

Note: This Getting Started has been tested in the following conditions:

- Windows 10/11.
- Kernel GREEN $2.1.1$.
- NXP i.MXRT1170 VEE Port $3.0.0$.

To get the most out of this training, participants should have:

- A good understanding of the *Overview* section.
- A basic knowledge of the *MicroEJ Glossary*.

Hardware setup

To follow this training, you need:

- A NXP i.MX RT1170 Evaluation Kit EVKB.
- A RK055HDMIPI4MA0 display panel.
- A FAT32-formatted microSD card.
- An Ethernet cable

Environment Setup

To follow this Getting Started, you need to:

Install MICROEJ SDK 6

Follow *MICROEJ SDK 6 installation Guide*, IntelliJ is used on this Getting Started but feel free to use your favorite IDE,

Accept the MICROEJ SDK EULA

You may have to accept the SDK EULA if you haven't already done it, please have a look at *SDK EULA Acceptation*.

Setup the NXP i.MXRT1170 EVKB

Follow this *Environment Setup* guide to get a development environment up and running for NXP i.MXRT1170 EVKB.

Additionally, make sure to:

- Insert a micro-SD card (FAT32-formatted) in the board connector,
- Connect the 1GB Ethernet connector to the internet.

Congratulations! You have finished the setup of your environment. You are now ready to discover how to build and flash the `Kernel-GREEN` .

Set up the Kernel GREEN on your IDE

Import the Project

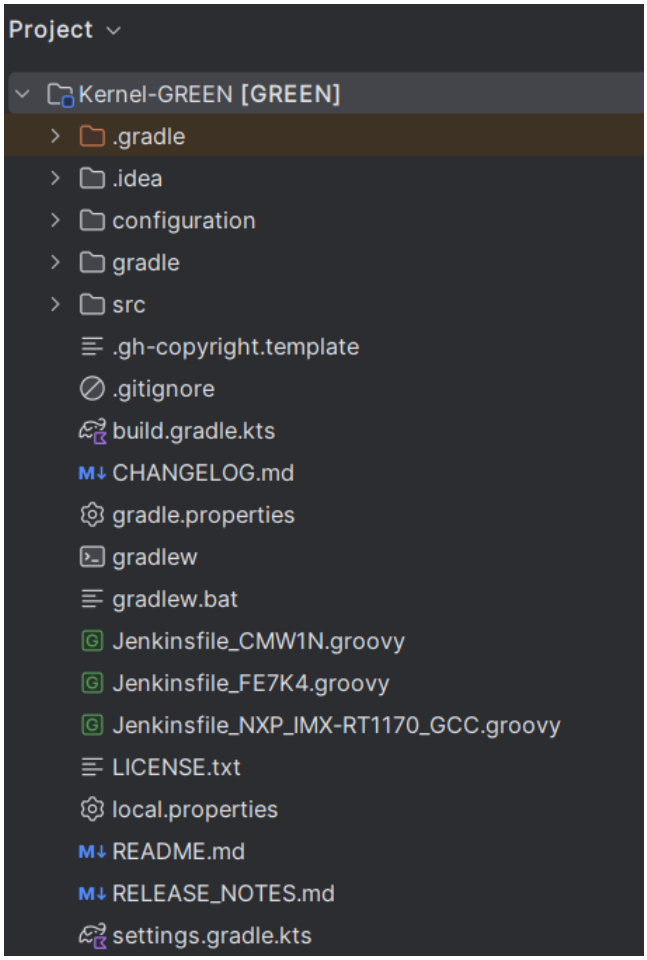
Clone or download the Kernel GREEN sources,

The first step is to import the `Kernel-GREEN` into your IDE:

Note: If you are using an IDE other than IntelliJ, please have a look at *Import a Project* section.

- If you are in the Welcome Screen, click on the `Open` button. Otherwise click either on `File` > `Open…` ,
- Select the `Kernel-GREEN` directory located where you downloaded it and click on the `OK` button.

The Gradle project should now be imported in IntelliJ, your workspace contains the following project in the `Projects` view:



Configure the Project

Select a VEE Port

As Applications need to depend on a VEE Port in order to be built, so does a Kernel project. Refer to the Prerequisites section of the Kernel GREEN README to learn more about the VEE Port requirements.

By default, the `Kernel-GREEN` project already uses the NXP i.MXRT1170 VEE Port 3.0.0 .

Kernel Configuration

The configuration/common.properties file provides a default configuration for the Kernel (runtime heap size, maximum number of threads, images heap size, etc.).

This configuration will also impact Sandboxed Applications running on top of the Kernel (e.g. max number of threads per application).

Check the Kernel GREEN README for more information about the Kernel configuration.

Run the Kernel GREEN on NXP i.MXRT1170 Evaluation Kit

Make sure to have completed all the *Environment Setup* steps before going further.

Build the Executable for the NXP i.MXRT1170 Evaluation Kit

In order to build the Executable of the `Kernel-GREEN`, the SDK provides the `buildExecutable` Gradle task.

Note: If you are using an IDE other than IntelliJ, please have a look at *Build an Executable* section. Come back on this page if you need to activate an Evaluation License.

- Double-click on the `buildExecutable` task in the Gradle tasks view.
- The build stops with a failure.
- Go to the top project in the console view and scroll up to get the following error message:

```
=============== [ Initialization Stage ] ===============
Platform connected to BSP location [redacted] using platform
 included 'bsp' folder.
[INFO ] Launching in Evaluation mode. Your UID is [redacted]
[ERROR] Invalid license check (No license found).
> Task :watch-vg:buildExecutable FAILED
23 actionable tasks: 7 executed, 16 up-to-date

FAILURE: Build failed with an exception.
```

- Copy the UID. It will be required to activate your Evaluation license.

  Request your *Evaluation License*:

- Request your Evaluation license by following the *Request your Activation Key* instructions. You will be asked to fill the machine UID field with the UID you copied before.
- When you have received your activation key by email, drop it in the license directory by following the *Install the License Key* instructions (drop the license key zip file to the ~/.microej/licenses/ directory).

  Now your Evaluation license is installed, you can relaunch your Kernel build by double-clicking on the `buildExecutable` task in the Gradle tasks view. It may take some time.

  The gradle task deploys the Kernel in the BSP and then builds the BSP using Make.

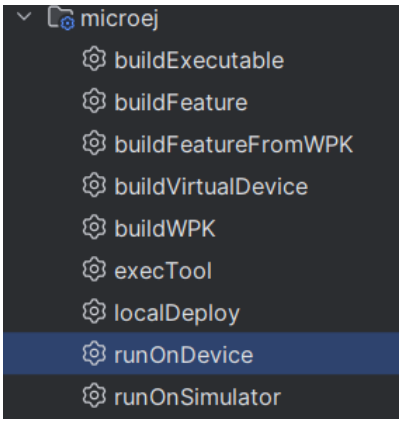
The `Kernel-GREEN` is built and ready to be flashed on a NXP i.MXRT1170 Evaluation Kit once the hardware setup is completed.

Flash the Kernel GREEN on the NXP i.MXRT1170 Evaluation Kit

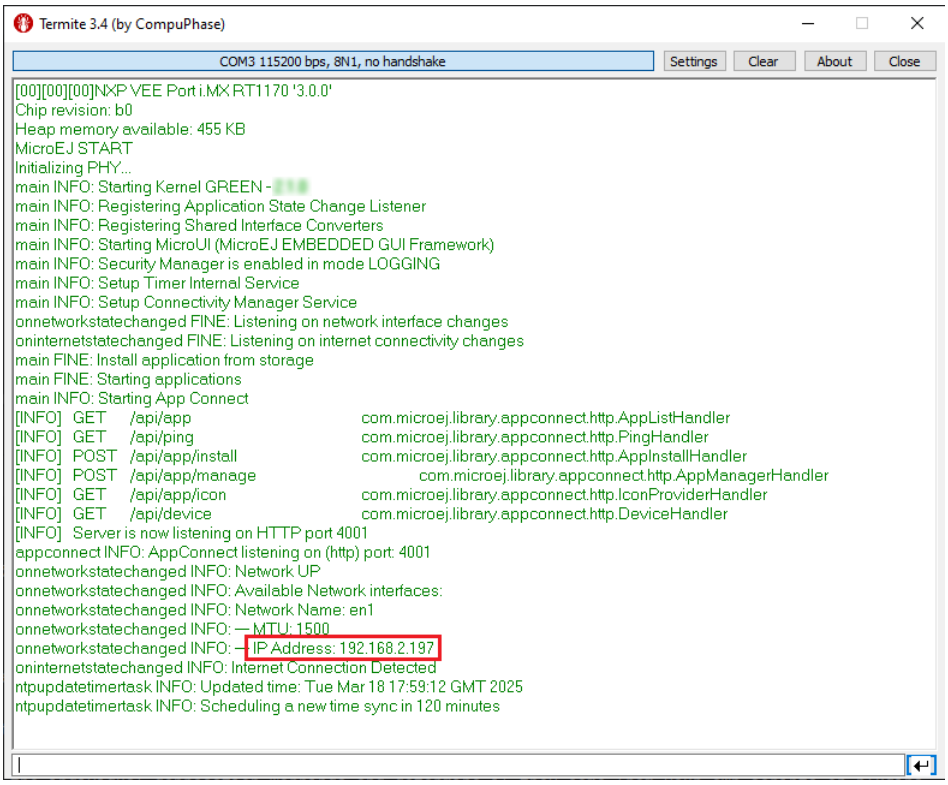
In order to flash the `Kernel-GREEN` on the NXP i.MXRT1170 Evaluation Kit, the application provides the Gradle `runOnDevice` task.

Note: If you are using an IDE other than IntelliJ, please have a look at *Run on Device* section.

- Double-click on the `runOnDevice` task in the Gradle tasks view. It may take some time:



Once the Executable is flashed, you should see the `Kernel-GREEN` traces in your console:



Well Done!

Now you know how to build and run a Multi-Sandbox Executable!

If you want to learn how to run a Sandboxed Application on your Multi-Sandbox Executable, you can continue this Getting Started.

Run a Sandboxed Application on your Multi-Sandbox Executable and Virtual Device

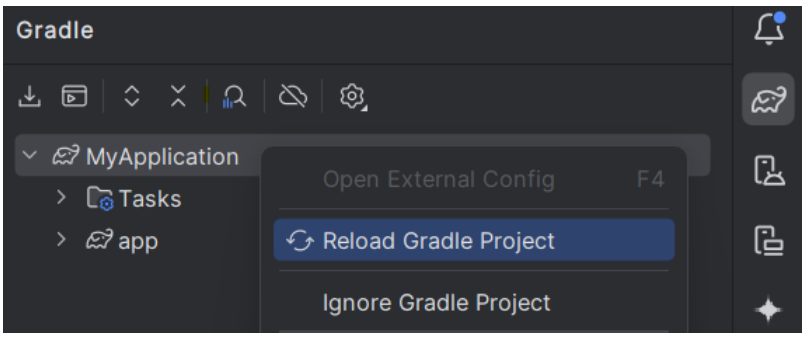
Follow the *Create and Run a Sandboxed Application* Getting Started to create and run a Sandboxed Application project on your Multi-Sandbox Executable and Virtual Device.

Note: When it comes to selecting the Multi-Sandbox Executable and Virtual Device (section *Run the Sandboxed Application on the Virtual Device*), follow this procedure:

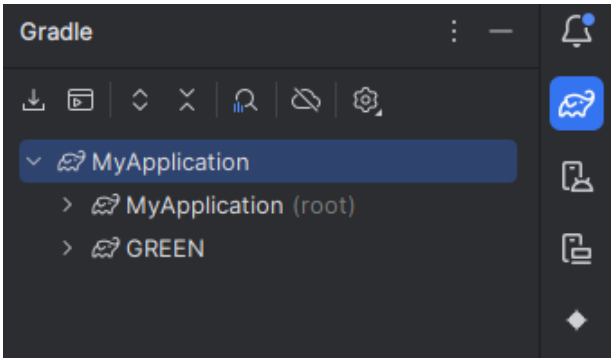
- Open the settings.gradle.kts files located at the root of your MyApplication project,
- Include the Kernel-GREEN project as a dependency of the MyApplication project,

```
rootProject.name = "MyApplication"
includeBuild("C:\\YOUR_PATH\\Kernel-GREEN")
```

- Reload the Gradle project:



- The Kernel-GREEN project should now appear in the Gradle tasks view:



- Open the build.gradle.kts file of the MyApplication project,
- Add the dependency to the Kernel-GREEN project: microejVee("com.microej.kernel:GREEN:2.1.1")

Note: Make sure that the version corresponds to the version defined in your Kernel GREEN `build.gradle.kts` file.

- Come back to the *Run the Sandboxed Application on the Virtual Device* Getting Started.

Going Further

You have now successfully executed a Sandboxed Application on an embedded device so what's next?

If you are an application developer you can continue to explore MicroEJ's APIs and functionalities by running and studying our samples at GitHub:

Foundation Libraries	Eclasspath	IoT
This project gathers all the basic examples of the foundation libraries.	This project gather all the examples of eclasspath.	This project gathers simple applications using net libraries.
https://github.com/MicroEJ/Example-Foundation-Libraries	https://github.com/MicroEJ/Example-Eclasspath	https://github.com/MicroEJ/Example-IOT

You can also learn how to build bigger and better applications by reading our *Application Developer Guide*.

If you are an embedded engineer you could look at our VEE port examples at GitHub. And to learn how create custom VEE ports you can read our *VEE Porting Guide*.

You can also follow the *Kernel Developer Guide* for more information on our multi-application framework or read about our powerful wearable solution called *VEE Wear*.

Last but not least you can choose to learn about specific topics by following one of our many *Training Courses* ranging from how to easily debug application to setting up a Continuous Integration process and a lot of things in between.

Kernel Development

Description

In this training, you will get an introduction to Kernel development in a Multi-Sandbox context.

What you will learn:

- Introduction to Kernel & Features concepts
- Running a Sandboxed Application on a Kernel
- Sandboxing notions
- Kernel services
- Building Kernel modules
- Introduction to Kernel-GREEN

Intended Audience

This training is designed for developers who want to gain a first understanding of Kernel development.

Prerequisites

To get the most out of this training, participants should have access to:

- Access to the NXP i.MX RT1170 Evaluation Kit.
- A mini USB cable.
- A micro SD card.

Training Resources

Unit 1: Kernel Development with MICROEJ SDK

Training for MICROEJ SDK 6 is coming soon!

Unit 2: Kernel-GREEN

Kernel "GREEN", a MicroEJ Kernel Application project is a Kernel code example that offers basic services for developing a Sandboxed Application and deploying it easily on your Device. Additionally, this project serves as a starting point for Kernel developers to learn and demonstrate most of the Multi-Sandboxing capabilities of the MicroEJ technology.

To complete this unit, follow the steps of the Kernel GREEN README on GitHub.

3.10.5 Graphical User Interface

Develop GUI with MicroEJ

Description

In this training, you will learn how to develop GUIs using the MicroEJ Graphical Framework. This training is raster-oriented.

What you will learn:

- Composing a GUI using widgets
- Tuning the look of widgets using style, images and fonts
- Changing the rendering of widgets
- Interacting with the GUI (e.g., touch or button)
- Animating a GUI
- Qualifying a GUI:
  - Benchmarking

- Debugging
- Testing

  Using an evaluation kit is optional to complete this training, as it can be fully done on the MICROEJ SDK Simulator.

Intended Audience

This training is designed for developers who want to develop GUI Applications with MicroEJ.

Prerequisites

To get the most out of this training, participants should have:

- A good knowledge of the Java programming language.
- A basic understanding of common GUI software design patterns.

Training Resources

How to Validate GUIs

Description

This document explains how to validate Graphical User Interfaces. It describes common pitfalls that can affect GUI performances, provides tools that allow to detect performance issues and how to solve them, and finally offers ways to test GUIs automatically.

Intended Audience

This training is for developers wishing to develop GUI Applications in MICROEJ VEE.

Prerequisites

To get the most out of this training, participants should have a good knowledge of Java programming language.

Implementing GUIs Efficiently

Before using more advanced UI debugging techniques, the global application code quality should be reviewed. An overall good code quality will help to get good UI performances. It will help to get more efficient code and allow easier debugging and maintenance.

Documents and Tools to Improve Application Code Quality

Here is a list of documents or tools that help to improve the quality of application code:

- The *Improve the Quality of Java Code* training explains how to improve the Quality of Java Code.
- The *Tutorial-UI* training provides guidelines to start developing an efficient GUI.
- The SonarQube™ source code quality analyzer allows to analyze an Application or Library code quality.

Using Recent Versions of UI Libraries

Using the latest UI libraries (MicroUI, MWT, Widget, etc.) available may solve some performance issues. The most recent UI libraries fix some bugs that may affect performance and they provide tools and libraries that allow to implement more performant UIs.

Memory Management

The Java management of memory may affect UI performances:

- Too much memory allocation/deallocation for UI resources (Images, Fonts).
- Too much object instantiation will lead to a big Managed Heap size. In some use cases, the garbage collection may lead to the UI slowing down.

To avoid those pitfalls:

- Calibrate the memories (Managed Heap, Images heap, etc.).
- Uses memory debugging tools:

– *Optimize Memory Footprint* training.

– *Memory inspection tools*.

– *Heap Analyzer*.

Format of UI Resources

One crucial aspect of optimizing an application is choosing the right image formats. Images can have a significant impact on an app's performance and memory usage. Therefore, selecting the best image format is essential. It helps reduce memory usage, speed up the app, and improve its overall performance.

MicroUI manages two kind of images, mutable and immutable images.

Mutable images are graphical resources that can be created and modified at runtime. The application can draw into such images. More information about mutable images can be found *here*.

As their name suggests, immutable images can not be modified. They are the most commonly used kind of images, this section will focus on them.

Decoding Immutable Images

Immutable images can be converted for display either during the build-time process, using the Image Generator, or at run-time, utilizing the appropriate decoder library.

The decision between these two approaches depends on the project's specific requirements. **Decoding at run-time** is a good choice when storage space is limited and offers greater flexibility. However, it may require more processing power and result in slower performance. Conversely, **decoding at build time** reduces the computational workload during run-time and is well-suited for devices with stringent performance demands, though it usually require more storage and it may sacrifice some flexibility in the process.

Format of Immutable Images

There are multiple output formats that can be used to convert the images, you can find them here: *Output Formats*.

Choosing the right output format is important to get the best performance:

- For opaque images, choose a format that has no transparency, RGB565 is usually sufficient.
- For a pictogram to colorize A4 is usually sufficient. The image can be colorized at runtime.
- The image format can be compressed, see *Compressed Output Formats*

The expected result of each format can be seen here: *Formats expected result*

Images Heap

Mutable images and immutable images decoded at runtime require some memory to be used. Please go to the *Images Heap* section for more information.

Benchmarking GUIs

The process of rendering a frame of the UI consists of several parts:

- Drawing of the UI:
  - MWT processing (layout of widgets and widget rendering process).
  - Drawing of the UI (MicroUI drawing method execution).
- Display flush, see *Buffer Policy*.
- Buffer Refresh Strategy (BRS), see *Buffer Refresh Strategy*.

Some tools are available to identify which part of this process affect the most the GUI performance.

SystemView

The SystemView tool can be used to trace the UI actions (drawings, flush, etc.) and detect which ones are the most time-consuming. The documentation of SystemView is available *here*. The MicroUI traces should be configured in SystemView in order to see the UI actions performed, it can be done by following *this documentation*. Custom traces can be added and logged from the Java application to record specific actions.

MicroUI Flush Visualizer

A perfect application has 100% of its display area drawn. This is the total area covered by the sum of the area drawn by the drawing operations. A value of 200% indicates the area drawn is equivalent to twice the surface of the entire display. A total area drawn between 100% to 200% is the norm in practice because widgets often overlap. However, if the total area drawn is bigger than 200%, that means that the total surface of the display was drawn more than twice, meaning that a lot of time could be spent drawing things that are never shown.

The MicroUI Flush Visualizer tool can be used to investigate potential performance bottlenecks in UI applications running on the Simulator by showing the pixel surface drawn between two MicroUI flushes.

The documentation of MicroUI Flush Visualizer is available *here*.

Debugging GUIs

High-level Debugging and Optimizations

This section provides insights into common issues affecting performances on the high-level side. The following advices will help reduce the MWT processing and drawing time.

Widget Hierarchy and Layout

Keeping the widget hierarchy as simple as possible will help to reduce the “MWT processing” part time. Improving the widget hierarchy design may help reduce the number of widgets or the number of them that are rendered when a certain part of the UI is updated.

Here are tools that allow to detect issues with the widget hierarchy:

- *Widget debug utilities* provides tools to visualize the widget tree, count the numbers of widgets or see their bounds.
- *MWT bounds highlighting* allows to visualize the bounds of the widgets, it is useful to detect overlapping widgets.

Bad Use of requestRender and requestLayout

The requestRender method requests a render of the widget on the display.

The requestLayout method requests a layout of all the widgets in the sub-hierarchy of this widget. It will compute the size and position of the widgets as setting their styles. requestLayout will trigger a render request after the layout.

A common mistake is to call requestRender just after a requestLayout . This will trigger two renders and thus affect the UI performances.

Another common issue is to request a layout where a render request would have been enough. If the size, position or style of the widgets didn't change requestRender is enough, requestLayout would have a longer processing time. This is especially true for animation where we want each frame to be processed as fast as possible.

Documentation about rendering and layout is available *here*.

Animations Implementation

There are a few implementations possible for animations with MicroEJ. The way widgets are animated should be chosen according to the use case and the limitation of the hardware.

Animator

The MWT's Animator allows to execute animations as fast as possible, it waits for the low-level screen flush to be done and directly triggers a new render. Thus the Animator will give the best framerate possible but will also consume a lot of CPU processing time.

TimerTask

A TimerTask can be used to execute an animation at a fixed framerate. This technique is very useful to set a fixed period for the animation but will cause issues if the time to render a frame is longer than that period, this will lead to missed frames. Some frames can take longer to render if their content is more complex or if the CPU is already used by another non-UI thread.

The framerate set when using a TimerTask for animation should be defined wisely, the time to render frames and the CPU utilization should be taken into consideration.

Animator and TimerTask mix

A mix of the Animator and TimeTask approaches could be implemented in order to set a fixed framerate but also to rely on the screen flush.

Hardware and Low-level Debugging and Optimizations

This section provides insights into the main spots to check regarding the low-level and the hardware.

Please see the VEE Porting Guide *Graphical User Interface* documentation for more information about the UI port.

At Project Level

Compiling Optimization Options

The project should be configured to bring the best performances with compiling optimization options correctly set up.

RTOS Tasks Environment

The priority of the UI task should be set high enough to avoid too many preemptions that may induce bad UI performances.

Another point that should be taken into consideration is the amount of other tasks that are running at the same time as the UI task. The total workload may be too high for the CPU, therefore, the UI task cannot get access to the required amount of computing power.

At Hardware Level

Hardware Capabilities

MCUs and SoCs may have access to various hardware IPs to speed up the UI. The UI port should exploit all of them to get the best performance. First of all, the GPU should be used if it is available on the system. Then, driving a display implies intensive memory usage, a DMA should be used whenever it's possible.

For more information about the flush policy, please read our documentation about *Display*.

Hardware Configuration

Each of the hardware components such as SPI, DMA or LCD controller must be configured to bring the best performances achievable. This implies to read carefully the datasheet of the MCU and the display and determine for example the best frequency and communication mode possible.

Another example of configuration with DMAs, a DMA has often a burst mode to transfer data, the UI port should use this mode to maximize performance.

Buffers Location in Memory

An important step during the development of the UI integration is the memory location of the buffers that will use the GUI to draw to the display. In an MCU, there may be different types of RAM available that have different properties in terms of quantity and speed. The fastest RAM should be chosen for the buffers if its size allows it.

Flush Policy

As described in the *Display* page, there are several flush policies that can be implemented. The best flush policy should be selected according to the hardware capabilities. Generally, the best flush policy is the switch mode.

Testing GUIs

Before applying UI debugging or optimization techniques, the application behaviour should be tested. There are different ways of doing this.

Test a GUI Application with a Software Robot

It is possible to test the GUI of an application via robotic process automation (RPA). Robot tests mimic the human user behavior in the GUI and can help detect various errors by automating behavior which otherwise would cost too much effort and/or time to execute manually.

Here are the steps required to use a robot in the MicroEJ environment:

- Record the robot input events
  - For this, you need a simple EventHandler which intercepts incoming events, for example from a Pointer, then passes them on to the real event handler.
- Start the usage of the new 'Watcher' logic after the UI has started
  - With this, the watching of the Pointer envents is initiated for the whole application.
- Create a Robot
  - The robot is a simple class which uses its own Pointer to move and press at the coordinates it has been instructed.
  - The robot should have a method which starts a series of instructions to move the Pointer.
- Execute the Robot method containing the instructions
  - The intercepting Event Handling will record and for example log the input.

  This simple way of automating GUI actions can be used to carry out real use cases and evaluate the results.

  The *How to test a GUI application with a (software) robot* training provides detailed insight into this topic.

Test a GUI Application with the Test Automation Tool

To execute regression tests automatically and monitor minor changes in a GUI, you can use the Test Automation Tool. The Test Automation Tool allows to automatically test UIs.

The tool comparison functionality can be integrated with JUnit tests.

For detailed information about the tool usage, please check the README in the repository.

Unit 1: Coding Challenges

This project contains coding challenges to learn the basics of UI development with MicroEJ. The training is divided into multiple steps, each helping the user create the views of a smartwatch.

- Slides: Tutorial-UI Training Manual (for MICROEJ SDK 6)
- Resources

Unit 2: GUI Validation

This guide provides an overview of the technics to properly implement, debug and automatically test a GUI application.

- *How to Validate GUIs*

Mastering MICROEJ SDK GUI Development Tools

Description

This training describes the available GUI tools provided by MICROEJ SDK to accelerate and hone your UI and UX development.

Intended Audience

This training is designed for developers who want to learn about the tools available for developing and debugging embedded applications most notably rendering and performance issues.

Prerequisites

To get the most out of this training, participants should have:

- A good understanding of the *Overview* section.
- A basic knowledge of the *MicroEJ Glossary*.
- A look at the introductory and setup sections of *Mastering MICROEJ SDK Development Tools*.
- Access to the NXP i.MX RT1170 Evaluation Kit.

Training Resources

Unit 1: MicroEJ Development GUI Tools Overview

This training is provided as a PDF, along with a downloadable .zip package of resources:

- (for MICROEJ SDK 6)
- Resources

Grayscale Display Considerations

Description

In this training, you will get an overview of the considerations to take into account when developing a GUI application using a grayscale display.

Intended Audience

This training is designed for embedded system developers, hardware engineers, and product designers who are responsible for choosing a display format for their product.

Prerequisites

To get the most out of this training, participants should have:

- A basic understanding of display technologies, color representation and depth,
- A basic understanding of MicroEJ Graphics Engine.

Training

First and foremost, grayscale displays can represent a range of shades between black and white.

The Graphics Engine supports the following grayscale formats by default:

- C4 : the pixel uses 4 bits-per-pixel (4BPP),
- C2 : the pixel uses 2 bits-per-pixel (2BPP),
- C1 : the pixel uses 1 bit-per-pixel (1BPP, a.k.a monochrome).

See *Pixel Structure* documentation for more information.

The below sections present the key considerations when choosing a grayscale display, using the widely adopted RGB565 colored format as a reference point.

Application Considerations

Testing a “Colored” Application on a Grayscale Display

It is possible to test a “colored” application on a grayscale display.

The straightforward way is to *update the display format of the VEE Port* to simulate the application using a grayscale format.

It is recommended that the C4 format be tested first to obtain a grayscale rendering. Then, the application resources should be adapted to render properly in C1 or C2 .

Example when switching from RGB565 to C4 format:

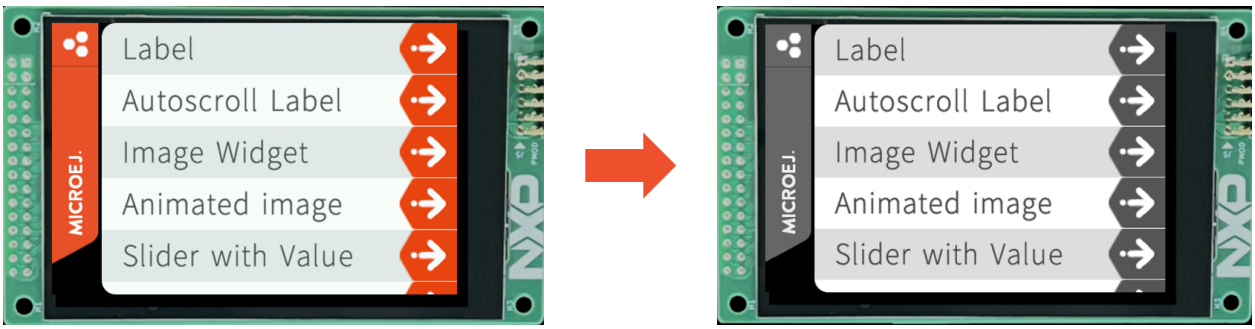


Fig. 296: RGB565 vs C4 Display Format

Note: Conversion functions are used at runtime to convert the application colors to the proper display format (see *Pixel Structure*)

For example: if a red rectangle is drawn, the color will be converted using the color conversion algorithm corresponding to the display format.

Anti-Aliasing

The anti-aliasing is managed like on colored displays.

It is recommended that the fonts and images be converted to the display format. It allows you to anticipate a wrong blending if your resource has too many bit-per-pixels compared to the display. (e.g. if the display format is C2 , embed your Fonts in 1BPP or 2BPP format instead of 4BPP).

Detect Display Format at Runtime

If needed, the APIs of the Display class can be used to detect the display format on the application side (isColor , getPixelDepth).

Raster Fonts Considerations

Anti-aliased raster fonts (*.fnt , *.ejf) may have a bad rendering when it comes to displaying them in C1 format (1BPP):

- 4BPP font format:

Animated image

- 1BPP font format:

It is recommended that the font files be re-generated to have a good rendering in C1 format.

Refer to *Fonts* documentation to learn more about Fonts configuration.

Footprint Considerations

The document below compares the footprint of RGB565 (16BPP) and C1 format (1BPP , monochrome).

> **Warning:** For the C1 format, memory alignment constraints can potentially increase the results presented below.

ROM Footprint

RAM Footprint

Application Resources

The application resources are the main item to consider in terms of footprint reduction.

For images, make sure to *set the image format* according to your display format.

Example with 3 images:

- Image A with size 32x32 , respecting the display format (fully opaque, e.g. RGB585 format)
- Image B with size 32x32 fully transparent (e.g. A8 format)
- Image C with size 32x32 with a mix of colors and transparency (e.g. ARGB4444 format)

Images footprint on a RGB565 display format (16BPP):

- Image A footprint embedded in RGB565 format (32x32x(**16**/8)) + header ~ 2kB
- Image B footprint embedded in A8 format (32x32x(**8**/8)) + header ~ 1 kB
- Image C footprint embedded in ARGB4444 format (32x32x(**8**/8)) + header ~ 2kB

Images footprint on a C1 display format (1BPP):

- Image A footprint embedded in C1 format (32x32x(**1**/8)) + header ~ 0.12 kB
- Image B footprint embedded in A1 format (32x32x(**1**/8)) + header ~ 0.12 kB
- Image C footprint embedded in AC11 format (32x32x(**2**/8)) + header ~ 0.25kB

For Fonts, make sure to *set the font format* according to your display format.

Example with the SourceSansPro_15px-600.ejf font:

- Font footprint on a RGB565 display format (16BPP):
  - 4 bit-per-pixel format ~ 9.4kB (obtained by building an application and checking the *SOAR Map File*)
- Font footprint on a C1 display format (1BPP):

- 1 bit-per-pixel format ~ 3.5kB (obtained by building an application and checking the *SOAR Map File*)

UI Libraries

There is no ROM footprint difference depending on the display format, the libraries are already optimized.

Only the algorithms used in the application are embedded in the final executable.

Graphics Engine

Some Graphics Engine algorithms (e.g., color conversion algorithms) can be removed if the application does not use them. Check the *Image Pixel Conversion* linker file configuration for more information.

Example with C1 display format:

- If the application does not contain images embedded in ARGB8888 format, the color conversion algorithm from ARGB8888 to C1 can be removed.

Note: Note that the Graphics Engine is already footprint-optimized. Removing those algorithms will not significantly reduce its ROM footprint.

Display Buffer Size

The *Display Buffer Size* can be significantly reduced when changing the display format.

Example:

- Let's assume a 320 x 240 SPI display is used (only 1 display buffer in RAM),
- RGB565 display format (16BPP):
- Display buffer footprint: 320x240x(**16**/8) = 150kB
- C1 display format (1BPP):
- Display buffer footprint: 320x240x(**1**/8) ~ 9.3kB

Images Heap

If the application uses the *Images Heap*, its size can be reduced as it now stores images in a lighter format.

Example of Images Heap sized to store a 100x100 image:

- RGB565 display format: 100x100x(**16**/8) ~ 19.5Kb
- C1 display format: 100x100x(**1**/8) ~ 1.2Kb

Debug a GUI Application Freeze

Description

In this training, you will get insights on how to instrument a frozen GUI application.

Intended Audience

This training is designed for Application developers who want to debug a GUI freeze in their application.

Prerequisites

To get the most out of this training, participants should have a basic knowledge of Java programming language and GUI development within MicroEJ environment.

Introduction

When an application User Interface freezes and becomes unresponsive, in most cases, one of the following conditions applies:

- An unrecoverable system failure occurred, like a HardFault, and the RTOS tasks are not scheduled anymore.
- The RTOS task that runs the Core Engine is never given CPU time (suspended or blocked).
- The RTOS task that runs the Core Engine is executing never-ending native code (infinite loop in native implementation for example).
- A Java method executes a long-running operation in the MicroUI thread (also called Display Pump thread).
- The application code is unable to receive or process user input events.

The following sections explain how to instrument the code to locate the issue when the UI freeze occurs. The steps followed are:

1. Check if the RTOS properly schedules the Core Engine task.
2. Check if the Core Engine properly schedules all threads.
3. Check if the Core Engine properly schedules the MicroUI thread.
4. Check if Input Events are properly processed.

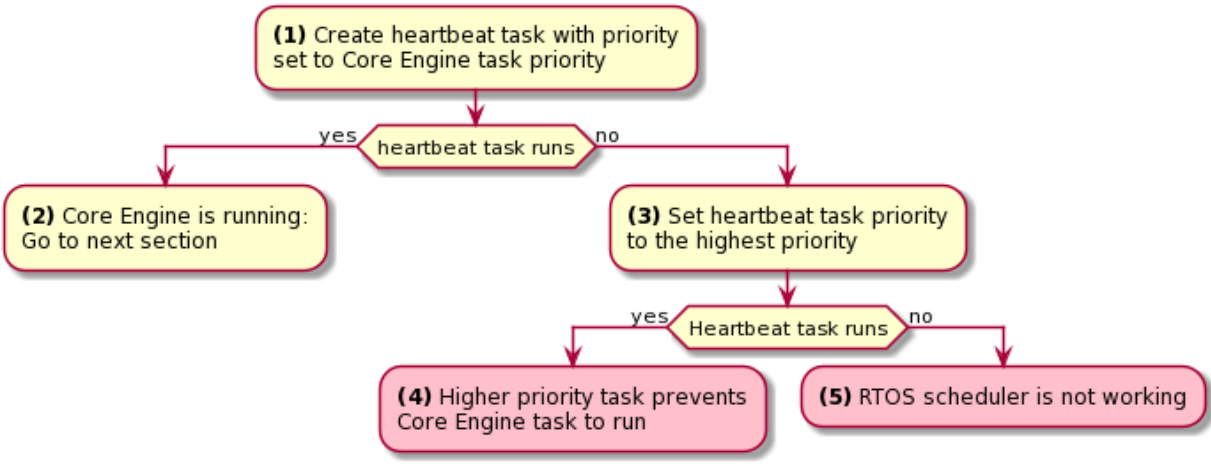
Note:

- The checks of the schedulers are possible with *SystemView* and *MicroUI Debug Traces*.
- The Input Events check is possible with the *LLUI_INPUT_dump API*.

Check RTOS Tasks Scheduling

Let's start at low level by figuring out if the RTOS is scheduling tasks correctly. If possible, use a debugger or *SystemView*; if not, use the heartbeat task described below.

The following flow chart summarizes the investigation steps with a heartbeat task:



(1) Make one of the RTOS tasks acts like a heartbeat: create a dedicated task and make it report in some way at a regular pace (print a message on standard output, blink an LED, use SystemView, etc.). Set the heartbeat task priority to the same priority as the Core Engine task.

(2) In this configuration, if the heartbeat is still running when the UI freeze occurs, we can go a step further and check whether the Core Engine is still scheduling threads or not. See next section *Check Threads Scheduling*.

(3) If the heartbeat doesn't run when the UI freeze occurs, set the heartbeat task priority to the maximum priority.

> **Warning:** Some RTOS use a task to schedule the RTOS timers. The heartbeat task priority must be lower than the RTOS timers priority.

(4) In this configuration, if the heartbeat is still running when the UI freeze occurs, then an RTOS task with a priority higher than the Core Engine task keeps using the CPU. Use the RTOS specific tools to identify what is the faulty task.

(5) If the heartbeat doesn't run when the UI freeze occurs, then the RTOS scheduler is not scheduling anything. This can be caused by an RTOS timer task or an interrupt handler that never returns, or a crash of the RTOS scheduler.

Check Threads Scheduling

As a reminder, the threading model implemented by Core Engine is called green thread: it defines a multi-threaded environment without relying on any native RTOS capabilities. Therefore, all threads run in a single RTOS task. For more details, please refer to the *Core Engine* section. A quick way to check if the threads are scheduled correctly is, here again, to make one of the threads print a heartbeat message. Copy/paste the following snippet in the `main()` method of the application:

```
TimerTask task = new TimerTask() {

    @Override
    public void run() {
        System.out.println("Alive");
    }
```

(continues on next page)

(continued from previous page)

```
};
Timer timer = new Timer();
timer.schedule(task, 10_000, 10_000);
```

This code creates a new thread that will print the message `Alive` on the standard output every 10 seconds.

Assuming no one canceled the `Timer`, if the `Alive` printouts stop when the UI freeze occurs, then it can mean that:

- The Core Engine stopped scheduling the threads.
- Or that one or more threads with a higher priority prevent the threads with a lower priority from running.

Here are a few suggestions:

- Ensure no threads with a high priority prevent the scheduling of the other threads. For example, convert the above example with a dedicated thread with the highest priority:

```
Thread thread = new Thread(new Runnable() {

   @Override
   public void run() {
         while (true) {
               try {
                     Thread.sleep(10_000);
                     System.out.println("Alive");
               } catch (InterruptedException e) {
                     e.printStackTrace();
               }
         }
   }
});
thread.setPriority(Thread.MAX_PRIORITY);
thread.start();
```

- The RTOS task that runs the Core Engine might be suspended or blocked. Check if some API call is suspending the task or if a shared resource could be blocking it.
- When a Java native method is called, it calls its C counterpart function in the RTOS task that runs the Core Engine. While the C function is running, no other Java methods can run because the Core Engine waits for the C function to finish. Consequently, no thread can ever run again if the C function never returns. Therefore, spot any suspect native functions and trace every entry/exit to detect faulty code.

Please refer to *Implementation Details* if you encounter issues when implementing the heart-beat.

Check UI Thread Liveness

Now, what if the `Alive` heartbeat runs while the UI is frozen? Threads are getting scheduled, but the UI thread (also called Display Pump thread) does not process display events.

Let's make the heartbeat snippet above execute in the UI thread. Simply wraps the `System.out.println("Alive")` with a callSerially():

```
TimerTask task = new TimerTask() {

    @Override
    public void run() {
        System.out.println("TimerTask Alive");
        MicroUI.callSerially(new Runnable() {

            @Override
            public void run() {
                System.out.println("UI Alive");
            }
        });
    }

    @Override
    public void uncaughtException(Timer timer, Throwable e) {
        // Default implementation of this method would cancel the task.
        // Let's just ignore uncaught exceptions for debug purposes.
        e.printStackTrace();
    }
};
Timer timer = new Timer();
timer.schedule(task, 10_000, 10_000);
```

In case this snippet prints `TimerTask Alive` but not `UI alive` when the freeze occurs, then there are a few options:

- The application might be processing a long operation in the UI thread, for example:
  - infinite/indeterminate loops
  - network/database access
  - heavy computations
  - Thread.sleep()/Object.wait()
  - `SNI_suspendCurrentJavaThread()` in native call

  When doing so, any other UI-related operation will not be processed until completion, leading the display to be unresponsive. Any code that runs in the UI thread might be responsible. Look for code executed as a result of calls to:
  - `repaint()` : code in renderContent()
  - `revalidate()` / `revalidateSubTree()` : code in `validateContent()` and `setBoundsContent()`
  - handleEvent()
  - callSerially(): code wrapped in such calls will be executed in the UI thread
- The UI thread has terminated.

  As a general rule, avoid running extended operations in the UI thread, follow the general pattern and use a dedicated thread/executor instead:

```
ExecutorService executorService = ServiceLoaderFactory.
↪getServiceLoader().getService(ExecutorService.class, SingleThreadExecutor.class);
executorService.execute(new Runnable() {

    @Override
    public void run() {

        // (... long non-UI operation ...)

        // optional: update the UI upon completion
        Display.getDefaultDisplay().callSerially(new Runnable() {

        @Override
        public void run() {
                    // update display code (will be executed in UI thread)
        }
    });
    }
});
```

Check Input Events Processing

Another case worth looking at is whether the application is processing user input events as it should. The UI may look “frozen” only because it doesn’t react to input events. Replace the desktop instance with the one below to log all user inputs.

```
Desktop desktop = new Desktop() {

    @Override
    public EventHandler getController() {
        EventHandler controller = super.getController();
        return new EventHandler() {
        @Override
            public boolean handleEvent(int event) {
            System.
↪out.println("Desktop.handleEvent() received event of type " + Event.getType(event));
            return controller.handleEvent(event);
            }
        };
    }
};
```

It is also possible to display the content of MicroUI Event Buffer with the LLUI_INPUT_IMPL_log_XXX API. Please refer to *the Event Buffer documentation* for more information.

Implementation Details

Threads Creation

The number of threads in the MicroEJ Application must be sufficient to support the creation of additional threads when using Timer and Thread. The number of available threads can be updated in the launch configuration of the application (see *Option(text): Number of threads*).

If it is not possible to increase the number of available threads (for example, because the memory is full), try to reuse another thread but not the UI thread.

UART Not Available

If the UART output is not available, use another method to signal that the heartbeat task is running (e.g., blink an LED, use SystemView).

3.10.6 Connectivity

Networking Basics

Description

This training aggregates samples that showcase network connectivity with MicroEJ.

Evaluation kit is optional to complete this training, it can be fully done on MICROEJ SDK Simulator.

Intended Audience

This training is designed for developers who want to create Applications with network connectivity.

Prerequisites

To get the most out of this training, participants should have:

- A basic knowledge of Java programming language.
- A basic knowledge of networking concepts.

Training Resources

Run the samples provided in the *Networking* documentation to get familiar with networking features.

3.10.7 For DevOps

Setup an Automated Build using Jenkins and Artifactory

Description

Such environment setup facilitates continuous integration (CI) and continuous delivery (CD), which improves productivity across your development ecosystem, by automatically:

- building modules when source code changes
- saving build results
- reproducing builds
- archiving binary modules

This guide should take 1 hour to complete.

Intended Audience

The audience for this document is engineers who are in charge of integrating MICROEJ SDK 6 to their continuous integration environment.

In addition, this document should be of interest to all developers wishing to understand how MicroEJ works with headless module builds.

Prerequisites

- An internet connection.
- Docker and Docker Compose V2 on Linux, Windows or Mac
- Git 2.x installed, with Git executable in path. We recommend installing Git Bash if your operating system is Windows (https://gitforwindows.org/).

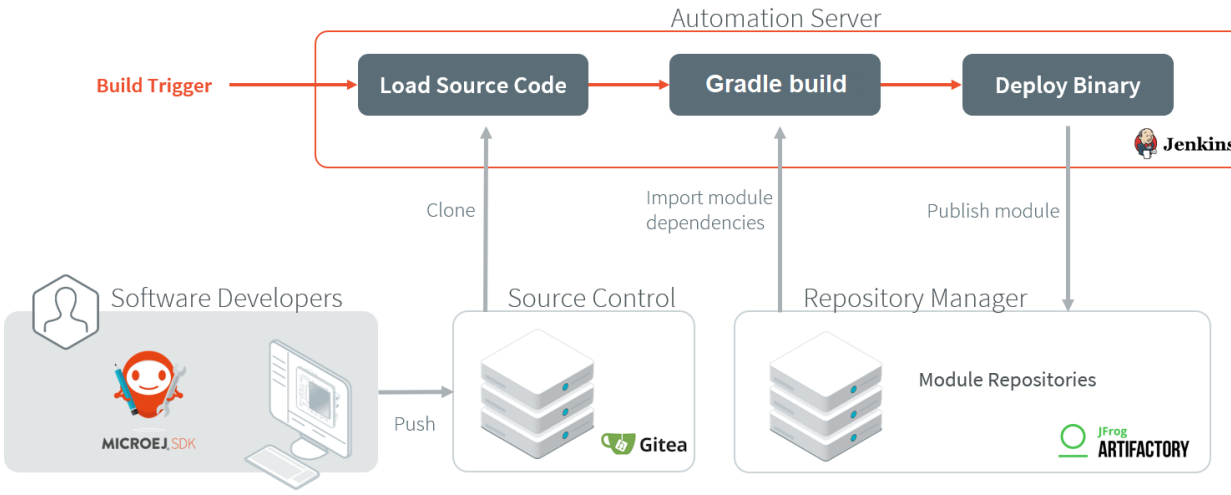
  This training was tested with Jenkins 2.462.3 , Artifactory 7.71.5 and Gitea 1.23.4 .

Note: For SDK 5, please refer to this MicroEJ Documentation Archive, section 9.6 Setup an Automated Build using Jenkins and Artifactory .

Introduction

The overall build and deployment flow of a module can be summarized as follows:

1. Some event triggers the build process (i.e module source changed, user action, scheduled routine, etc.)
2. The module source code is retrieved from the Source Control System
3. The module dependencies are imported from the Repository Manager
4. The Automation Server then proceeds to building the module
5. If the build is successful, the module binary is deployed to the Repository Manager



Overview

The next sections describe step by step how to setup the build environment and build your first MicroEJ module.

The steps to follow are:

1. Run and setup Jenkins, Artifactory and Gitea
2. Create a simple MicroEJ module (Hello World)
3. Create a new Jenkins job for the Hello World module
4. Build the module

In order to simplify the steps, this training will be performed locally on a single machine.

Artifactory will host MicroEJ modules in 3 repositories:

- microej-module-repository : repository initialized with pre-built MicroEJ modules, a mirror of the *Central Repository*
- custom-modules-snapshot : repository where custom snapshot modules will be published
- custom-modules-release : repository where custom release modules will be published

Prepare your Docker environment

This section assumes the prerequisites have been properly installed.

1. Create a new directory, inside create a file named docker-compose.yaml and copy this content:

```
version: '3'
services:
  artifactory:
    image: releases-docker.jfrog.io/jfrog/artifactory-oss:7.71.5
    container_name: artifactory
    environment:
      - JF_ROUTER_ENTRYPOINTS_EXTERNALPORT=8082
    ports:
```

(continues on next page)

(continued from previous page)

```
    - 8082:8082 # for router communication
    - 8081:8081 # for artifactory communication
    - 8085:8085 # for artifactory federation communication
  volumes:
    - artifactory:/var/opt/jfrog/artifactory
    - /etc/localtime:/etc/localtime:ro
  restart: always
  logging:
    driver: json-file
    options:
      max-size: "50m"
      max-file: "10"
  ulimits:
    nproc: 65535
    nofile:
      soft: 32000
      hard: 40000

gitea:
  image: gitea/gitea:1.23.4
  container_name: gitea
  environment:
    - USER_UID=1000
    - USER_GID=1000
  restart: always
  volumes:
    - gitea:/data
    - /etc/timezone:/etc/timezone:ro
    - /etc/localtime:/etc/localtime:ro
  ports:
    - "3000:3000"
    - "222:22"

jenkins:
  image: jenkins_master
  container_name: jenkins
  build:
    dockerfile: Dockerfile
  restart: always
  ports:
    - 50000:50000
    - 8080:8080
  volumes:
    - jenkins:/var/jenkins_home
    - /var/run/docker.sock:/var/run/docker.sock
  links:
    - gitea
    - artifactory

volumes:
  gitea:
  artifactory:
  jenkins:
```

2. Create another file named `Dockerfile` and copy this content:

```
FROM jenkins/jenkins:2.462.3
USER root
RUN apt-get update -qq \
    && apt-
↪get install -qqy apt-transport-https ca-certificates curl gnupg2 software-properties-common
RUN curl -fsSL https://download.docker.com/linux/debian/gpg | apt-key add -
RUN add-apt-repository \
   "deb [arch=amd64] https://download.docker.com/linux/debian \
   $(lsb_release -cs) \
   stable"
RUN apt-get update  -qq \
    && apt-get -y install docker-ce
RUN usermod -aG docker jenkins
```

3. In this directory, launch the command `docker compose up -d`. After a few moments you should have three running containers (named jenkins, gitea and artifactory).

   Using `docker compose ps` will show if containers started properly. Logs can be viewed with `docker compose logs`.

Get a Module Repository

A Module Repository is a portable ZIP file that bundles a set of modules for extending the MicroEJ development environment. Please consult the *Module Repository* section for more information.

This training uses the MicroEJ Central Repository, which is the Module Repository used by MICROEJ SDK to fetch dependencies when starting an empty workspace. It bundles Foundation Library APIs and numerous Add-On Libraries.

Next step is to download a local copy of this repository:

1. Visit the Central Repository on the MicroEJ Developer website.
2. Navigate to the `Production Setup` section.
3. Click on the `offline repository` link. This will download the Central Repository as a ZIP file.

Setup Artifactory

Configure Artifactory

For demonstration purposes we will allow anonymous users to deploy modules in the repositories:

1. Once Artifactory container is started, go to `http://localhost:8082/`.
2. Login to Artifactory for the first time using the default `admin` account (Username: `admin`, Password: `password`).
3. Skip the installation wizard if it appears.
4. Go to `Administration` > `User Management` > `Settings`.
5. In the `User Security Configuration` section, check `Allow Anonymous Access`.

6. Click on Save .
7. Go to Administration > User Management > Permissions .
8. Click on Anything entry (do not check the line), then go to Users tab
9. Click on anonymous and check Deploy/Cache permission in the Selected Users Repositories category.
10. Click on Save .

Next steps will involve uploading large files, so we have to increase the file upload maximum size accordingly:

1. Go to Administration > Artifactory > General > Settings .
2. In the General Settings section, change the value of File Upload In UI Max Size (MB) to 1024 then click on Save .

Create Repositories

We will now create and configure the repositories. Let's start with the repository for the future built snapshot modules:

1. Go to Administration > Repositories > Repositories in the left menu.
2. Click on Add Repositories > Local Repository
3. Select Maven .
4. Set Repository Key field to custom-modules-snapshot and click on Create Local Repository .

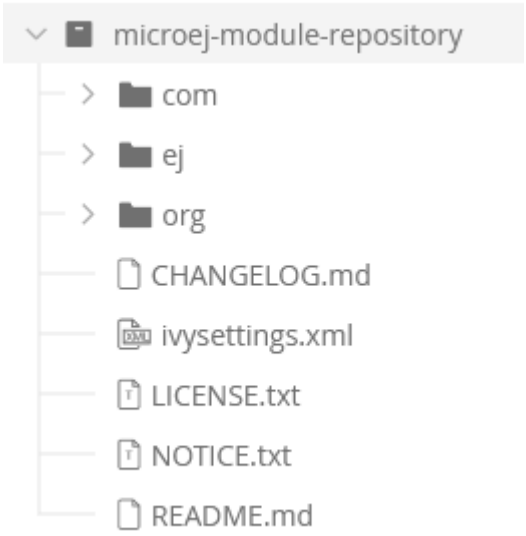
Repeat the same steps for the other repositories with the Repository Key field set to custom-modules-release and microej-module-repository .

Import MicroEJ Repositories

In this section, we will import MicroEJ repositories into Artifactory repositories to make them available to the build server.

1. Go to Administration > Artifactory > Import & Export > Repositories .
2. Scroll to the Import Repository from Zip section.
3. As Target Local Repository , select microej-module-repository in the list.
4. Click on Select file and select the MicroEJ module repository zip file (central-repository-[version].zip) that you downloaded earlier (please refer to section *Get a Module Repository*).
5. Click Upload . At the end of upload, click on Import . Upload and import may take some time.

Artifactory is now hosting all required MicroEJ modules. Go to Application > Artifactory > Artifacts and check that the repository microej-module-repository does contain modules as shown in the figure below.



Setup Gitea

Install Gitea

1. Once the Gitea container is started, go to http://localhost:3000/ .
2. Don't change anything on the Initial Configuration , click on Install Gitea
3. Click on Register account and create one. The first created user become the administrator.

Configure Gitea

1. At the top right click on the arrow then New Repository
2. As Repository Name set helloworld , leave the other options as default.
3. Click Create Repository .

Setup Jenkins

Install Jenkins

1. Once Jenkins container is started, go to http://localhost:8080/ .
2. To unlock Jenkins, copy/paste the generated password that has been written in the container log. Click on `Continue` .
3. Select option `Install suggested plugins` and wait for plugins installation.
4. Fill in the `Create First Admin User` form. Click `Save and continue` .
5. Click on `Save and finish` , then on `Start using Jenkins` .

Configure Jenkins

1. Go to `Manage Jenkins` > `Plugins` .
2. Add Docker Pipeline plugin:

1. Go to `Available plugins` section.
2. Search *Docker Pipeline*.
3. Install it and restart Jenkins

Build a new Module using Jenkins

Since your environment is now setup, it is time to build your first module from Jenkins and check it has been published to Artifactory. Let's build an "Hello World" Sandboxed Application project.

Create a new MicroEJ Module

Now we will create an Application project that we'll push to a Git repository. In this example, we will use a plublic template but you can create an application project from scratch with the *SDK 6 User Guide*.

Note: For demonstration purposes, we'll create a new project and share it on a local Git bare repository. You can adapt the following sections to use an existing MicroEJ project and your own Git repository.

1. Clone https://github.com/MicroEJ/Tool-Project-Template-Application repository.
2. Go to the repository directory and type the following commands (replace <admin_user> by Gitea user)

```
git remote rename origin old-origin
git remote add origin http://localhost:3000/<admin_user>/helloworld.git
git push --set-upstream origin --all
```

Note: We need the IP address of the Docker Bridge Network, here we consider that it's `172.17.0.1` but you can check with the command `ip addr show docker0` on the Docker host.

1. On the project directory, create a file named `Jenkinsfile` and copy this content inside:

```
pipeline {
    agent {
        docker {
            image 'eclipse-temurin:11-jdk'
            args '-e ACCEPT_MICROEJ_SDK_EULA_V3_1C=YES'
        }
    }
    stages {
        stage('Build and publish') {
            steps {
                sh './gradlew build publish -Penable.ivy.descriptor.default=false --init-script init.gradle.kts'
            }
        }
    }
}
```

2. Update `build.gradle.kts` file and set line 20 with this text. Building or running an Application with the SDK requires a VEE Port, so we use RT1170 Vee Port for this example:

```
microejVee("com.nxp.vee.mimxrt1170:evk_platform:2.2.0")
```

3. Create a file named `init.gradle.kts` and copy this content inside:

```
settingsEvaluated {

  val artifactorRepositoryUrl = "http://172.17.0.1:8082/artifactory"

  dependencyResolutionManagement {
    repositories {
      /* MicroEJ Central repository for Maven/Gradle modules */
      maven {
        name = "microEJForgeCentral"
        url = uri("${artifactorRepositoryUrl}/microej-module-repository")
        isAllowInsecureProtocol = true
      }
      /* MicroEJ Developer repository for Maven/Gradle modules */
      maven {
        name = "microEJForgeDeveloper"
        url = uri("https://forge.microej.com/artifactory/microej-developer-repository-release")
      }
      /* MicroEJ SDK 6 repository for Maven/Gradle modules */
      maven {
        name = "microEJForgeSDK6"
        url = uri("https://forge.microej.com/artifactory/microej-sdk6-repository-release/")
      }
      /* MicroEJ Central repository for Ivy modules */
      ivy {
        name = "microEJForgeCentralIvy"
        url = uri("${artifactorRepositoryUrl}/microej-module-repository")
```

(continues on next page)

(continued from previous page)

```
        patternLayout {
          artifact("[organisation]/[module]/[revision]/[artifact]-[revision](-[classifier])(.[ext])")
          ivy("[organisation]/[module]/[revision]/ivy-[revision].xml")
              setM2compatible(true)
        }
        isAllowInsecureProtocol = true
      }
      /* MicroEJ Developer repository for Ivy modules */
      ivy {
        name = "microEJForgeDeveloperIvy"
        url = uri("https://forge.microej.com/artifactory/microej-developer-repository-release")
        patternLayout {
          artifact("[organisation]/[module]/[revision]/[artifact]-[revision](-[classifier])(.[ext])")
          ivy("[organisation]/[module]/[revision]/ivy-[revision].xml")
          setM2compatible(true)
        }
      }
      /* MicroEJ SDK 6 repository for Ivy modules */
      ivy {
        name = "microEJForgeSDK6Ivy"
        url = uri("https://forge.microej.com/artifactory/microej-sdk6-repository-release/")
        patternLayout {
          artifact("[organisation]/[module]/[revision]/[artifact]-[revision](-[classifier])(.[ext])")
          ivy("[organisation]/[module]/[revision]/ivy-[revision].xml")
          setM2compatible(true)
        }
      }
    }

    /**
     * Publish repositories
     */
    pluginManager.withPlugin("maven-publish") {
      configure<PublishingExtension> {
        repositories {
          maven {
            name = "artifactoryRepository"
            url = uri("${artifactorRepositoryUrl}/custom-modules-snapshot")
            isAllowInsecureProtocol = true
          }
        }
      }
    }
  }


  /**
   * Plugins repositories
   */
  pluginManagement {
    repositories {
      /* MicroEJ Central repository for Maven/Gradle modules */
      maven {
        name = "microEJForgeCentral"
        url = uri("${artifactorRepositoryUrl}/microej-module-repository")
```

(continues on next page)

(continued from previous page)

```
        isAllowInsecureProtocol = true
    }
    /* MicroEJ Developer repository for Maven/Gradle modules */
    maven {
        name = "microEJForgeDeveloper"
        url = uri("https://forge.microej.com/artifactory/microej-developer-repository-release")
    }
    /* MicroEJ SDK 6 repository for Maven/Gradle modules */
    maven {
        name = "microEJForgeSDK6"
        url = uri("https://forge.microej.com/artifactory/microej-sdk6-repository-release/")
    }
    /* MicroEJ Central repository for Ivy modules */
    ivy {
      name = "microEJForgeCentralIvy"
      url = uri("${artifactorRepositoryUrl}/microej-module-repository")
      patternLayout {
        artifact("[organisation]/[module]/[revision]/[artifact]-[revision](-[classifier])(.[ext])")
        ivy("[organisation]/[module]/[revision]/ivy-[revision].xml")
        setM2compatible(true)
      }
      isAllowInsecureProtocol = true
    }
    /* MicroEJ Developer repository for Ivy modules */
    ivy {
      name = "microEJForgeDeveloperIvy"
      url = uri("https://forge.microej.com/artifactory/microej-developer-repository-release")
      patternLayout {
        artifact("[organisation]/[module]/[revision]/[artifact]-[revision](-[classifier])(.[ext])")
        ivy("[organisation]/[module]/[revision]/ivy-[revision].xml")
        setM2compatible(true)
        }
      }
    /* MicroEJ SDK 6 repository for Ivy modules */
    ivy {
      name = "microEJForgeSDK6Ivy"
      url = uri("https://forge.microej.com/artifactory/microej-sdk6-repository-release/")
      patternLayout {
        artifact("[organisation]/[module]/[revision]/[artifact]-[revision](-[classifier])(.[ext])")
        ivy("[organisation]/[module]/[revision]/ivy-[revision].xml")
        setM2compatible(true)
      }
    }
  }
}
```

This file configures the MicroEJ Module Manager to import and publish modules from the Artifactory repositories described in this training. Please refer to the *Settings File* section for more details.

Note: At this point, the content of the directory Tool-Project-Template-Application should look like the following:

```
build.gradle.kts
CHANGELOG.md
configuration
    common.properties
gradle
    wrapper
        gradle-wrapper.jar
        gradle-wrapper.properties
gradlew
gradlew.bat
init.gradle.kts
Jenkinsfile
LICENSE.txt
README.md
settings.gradle.kts
src
    main
        java
            com
                mycompany
                    myapplication
                        Main.java
        resources
    test
        java
        resources
```

1. Push these modifications to Gitea repository

Create a New Jenkins Job

Start by creating a new job for building our application.

1. Go to Jenkins dashboard.
2. Click on `New Item` .
3. Set item name to `Hello World` .
4. Select `Multibranch Pipeline` .
5. Validate with `Ok` button.
6. In `General` tab set `Display Name` to `Hello World`
7. In `Branch Sources` , click on `Add Source` > `Git` .
8. Add `Project Repository` http://172.17.0.1:3000/<admin_user>/helloworld.git

9. Click on Save .

Build the "Hello World" Application

Let's run the job!

In Jenkins Hello World dashboard, click on master branch, then click on Build Now .

Note: You can check the build progress by clicking on the build progress bar and showing the Console Output .

At the end of the build, the module is published to http://localhost:8082/artifactory/list/custom-modules-snapshot/com/mycompany/my-application/ .

Congratulations!

At this point of the training:

- Artifactory is hosting your module builds and MicroEJ modules.
- Jenkins automates the build process using SDK6.

The next recommended step is to adapt Gradle/Jenkins/Artifactory configuration to your ecosystem and development flow.

Appendix

This section discusses some of the customization options.

Customize Jenkins

Jenkins jobs are highly configurable, following options and values are recommended by MicroEJ, but they can be customized at your convenience.

In General tab:

1. Check Discard old builds and set Max # of builds to keep value to 15 .
2. Click on Advanced button, and check Block build when upstream project is building .

In Build triggers tab:

1. Check Poll SCM , and set a CRON-like value (for example H/30 * * * * to poll SCM for changes every 30 minutes).

   In Post-build actions tab:

1. Add post-build action Publish JUnit test result report :
2. Set Test report XMLs to **/target~/test/xml/**/test-report.xml, **/target~/test/xml/**/*Test.xml .

Note: The error message '**/target~/test/xml/**/test-report.xml' doesn't match anything: '**' exists but not '**/target~/test/xml/**/test-report.xml' will be displayed since no build has been executed yet. These folders will be generated during the build.

3. Check Retain long standard output/error .
4. Check Do not fail the build on empty test results

3.11 SDK 5 User Guide

MICROEJ SDK is an integrated environment to create software applications for MicroEJ-ready devices. The SDK provides tools to write applications and run them on a virtual (simulated) or real device. The capability to execute an application in a simulated environment allows to quickly test changes done in the application code and hence provides a short development feedback loop.

Since the purpose of the SDK is to develop for targeted MCU/MPU computers (IoT, wearable, etc.), it is a cross-development tool. But unlike standard low-level cross-development tools, the SDK offers unique services like hardware simulation and local deployment to the target hardware.

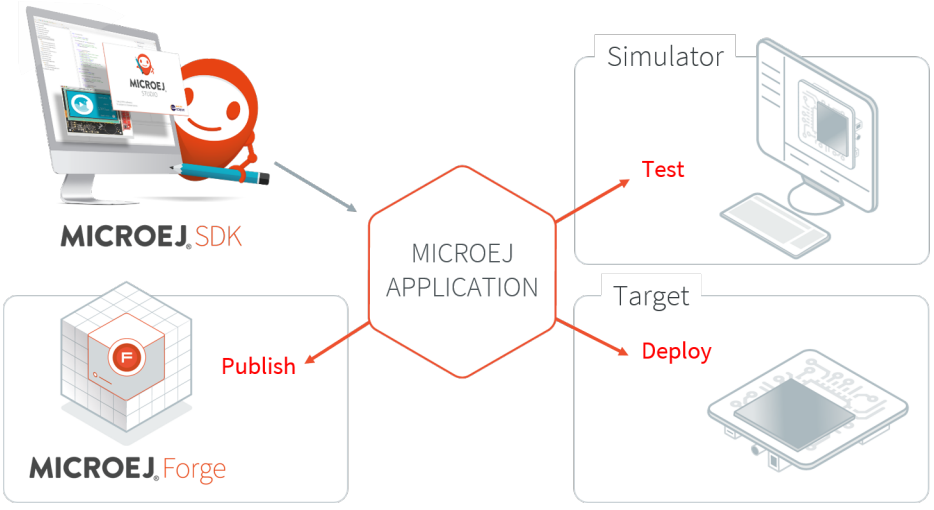


Fig. 297: MicroEJ Application Development Overview

The integrated environment is composed of the following main elements:

- **SDK Version 5.x**, an Integrated Development Environment (IDE) for writing and building Applications. It is based on Eclipse Java edition and relies on the integrated Java compiler (JDT).

  It is also packaged with Eclipse to produce a **SDK Distribution**.

- **MicroEJ Module Manager**, the module and build manager used to compile and package any kind of *modules natures*. It provides a Command Line Interface to build modules, especially used in a Continuous Integration environment. See *MicroEJ Module Manager* section for more details.

- **Architecture**, the software package that includes the MEJ32 port to a target instruction set and a C compiler, SOAR, core libraries and Simulator. See *MicroEJ Architecture* section for more details.

  The SDK allows to connect repositories hosting software modules in source and binary form. By default, it is configured with the repositories provided MicroEJ Corp.:

- **Central and Developer Repository**, the modules repositories containing all the libraries required to develop an Application. See *Module Repositories* section for more details.

- **Github Repositories**, source repositories with examples and demos. See *GitHub Repositories* section for more details.

  The SDK is licensed under the *SDK End User License Agreement (EULA)*. The following figure shows a detailed view of the elements.

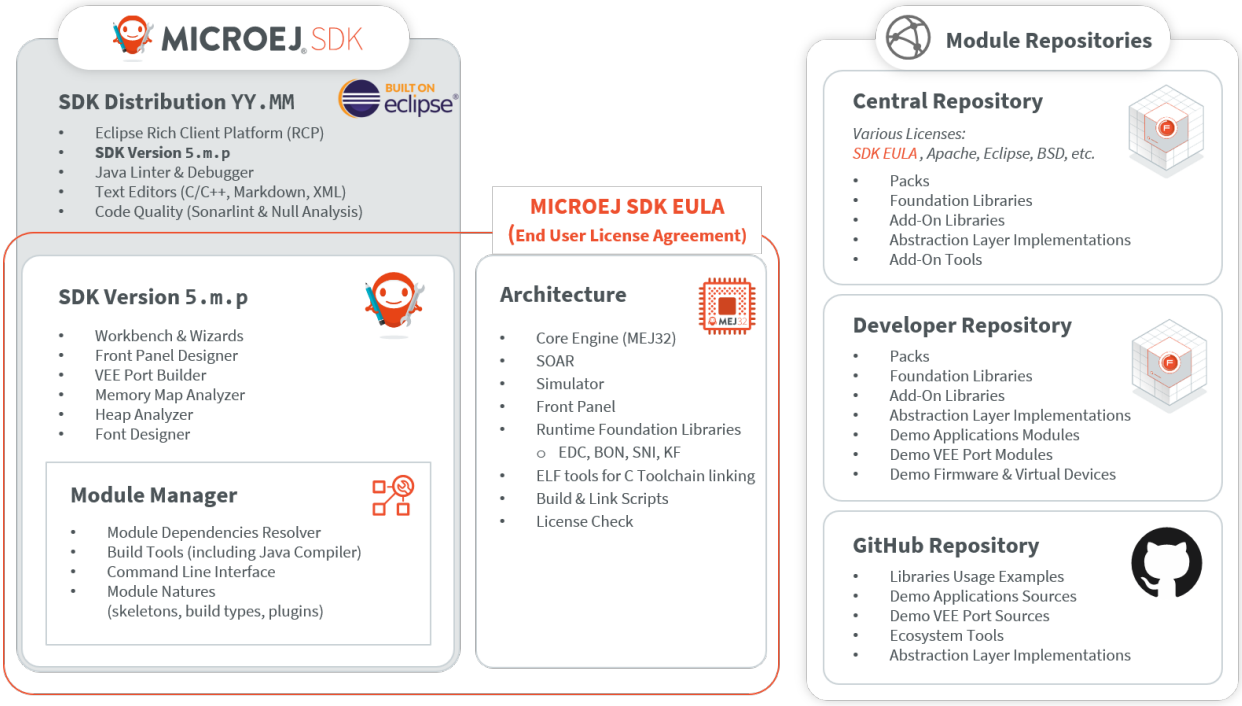


Fig. 298: SDK Detailed View

3.11.1 Installation

This chapter will guide you through the installation process of the SDK Distribution on your workstation.

If you want to evaluate MicroEJ, we recommend that you refer to the *Getting Started* chapter, which will guide you to install an SDK Distribution compatible with the Getting Started tutorials.

Otherwise, follow the instructions of the *Install Latest SDK Distribution* page to install the latest SDK Distribution compatible with your needs.

Fig. 299: SDK Splash Screen

Install Latest SDK Distribution

This section will guide you through the installation process of the latest SDK Distribution 24.01 using the step-by-step executable installer.

The SDK Distribution 24.01 requires a JDK 11 and thus can only work with an *Architecture 7.17.0* or higher. In all other cases, please jump to *Install SDK Distribution 21.11* section. See also the *System Requirements* page for more information on the list of supported environments.

Note: Launching the SDK Distribution installer requires administrator privileges and a JDK 11 installed by default on your workstation. If you don't have one of them or if you do not want to modify your default settings, please jump to *Install Portable SDK Distribution* section.

Download SDK Distribution

Download the SDK Distribution 24.01 installer for your operating system:

- Windows (.exe)
- Linux (.zip)
- macOS x86_64 - Intel chip (.zip)
- macOS aarch64 - M1 chip (.zip) (requires *Architecture 7.18.0 or higher*)

Check JDK Version

From the version 22.06 , the SDK Distribution installer requires a JDK 11 installed by default on your workstation. If you don't have any JDK installed, see the *Get JDK* section.

Check the default Java version by running the following command in a new terminal:

```
> java -version

openjdk version "11.0.15" 2022-04-19
OpenJDK Runtime Environment Temurin-11.0.15+10 (build 11.0.15+10)
OpenJDK 64-Bit Server VM Temurin-11.0.15+10 (build 11.0.15+10, mixed mode)
```

Now you can proceed with the installation steps.

Install SDK Distribution

- Launch the installer executable
  - On Windows, start MicroEJ-SDK-Installer-Win64-24.01.exe .
  - On Linux, unzip MicroEJ-SDK-Installer-Linux64-24.01.zip and start MicroEJ-SDK-Installer-Linux64-1.3.0.sh .
  - On macOS, unzip MicroEJ-SDK-Installer-MacOS-24.01.zip and start MicroEJ-SDK-Installer-MacOS-1.3.0.app .
  - Or unzip MicroEJ-SDK-Installer-MacOS-A64-24.01.zip and start MicroEJ-SDK-Installer-MacOS-A64-1.3.0.app . In case of error, check your app has not been put in quarantine (see *macOS troubleshooting* section)

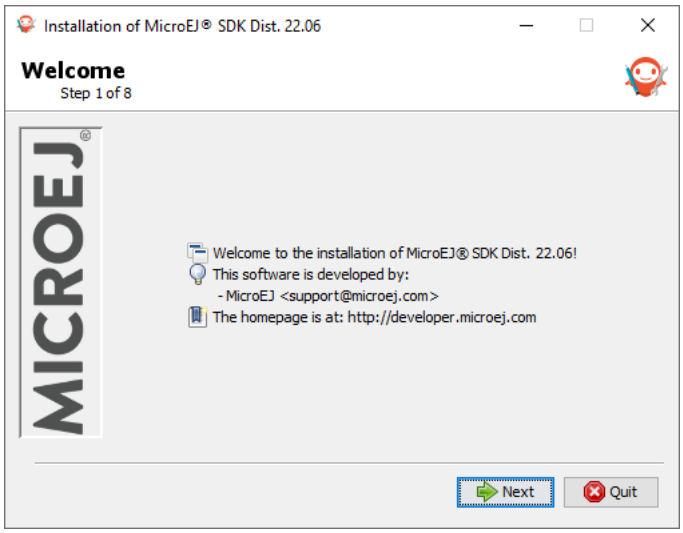


Fig. 300: Welcome to the installer

- Click on the Next button.
- Select I accept the terms of this license agreement. . Then click on the Next button.

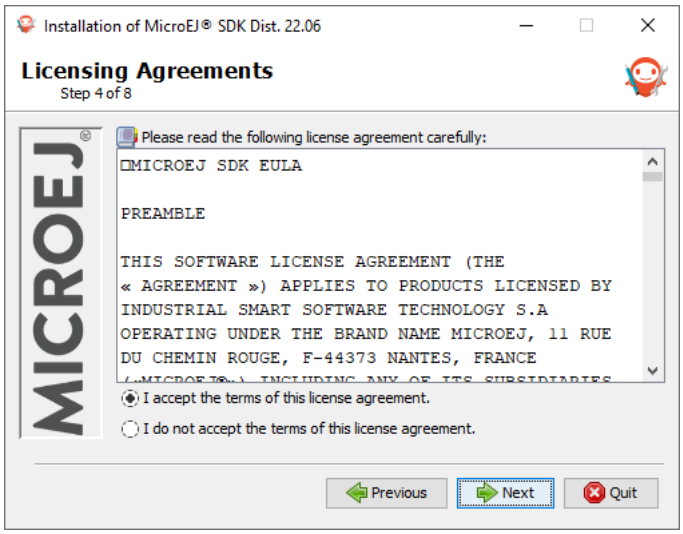


Fig. 301: Accept the terms of this license agreement

- Select the installation path of your SDK. By default it is C:\Program Files\MicroEJ\MicroEJ-SDK-24.01 for Windows. Then click on the Next button.

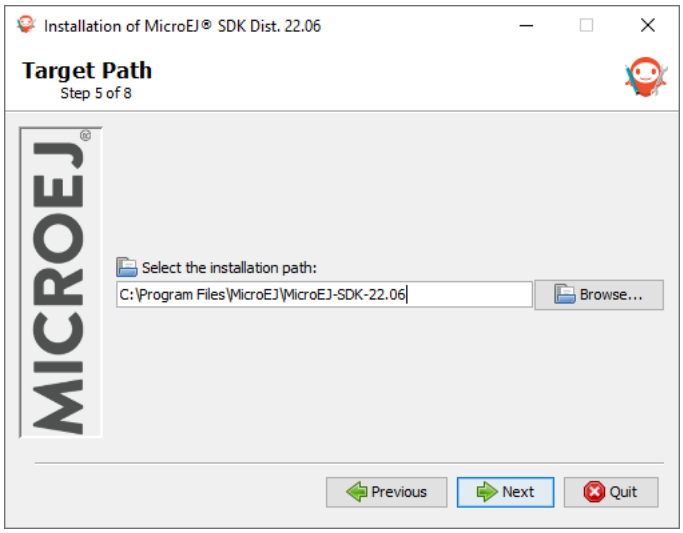


Fig. 302: Choose the installation path

- Click on the OK button to confirm the installation path.

Warning: On MacOS, the path may duplicate the last directory if you opened it in the File Selection Dialog instead of selecting it.

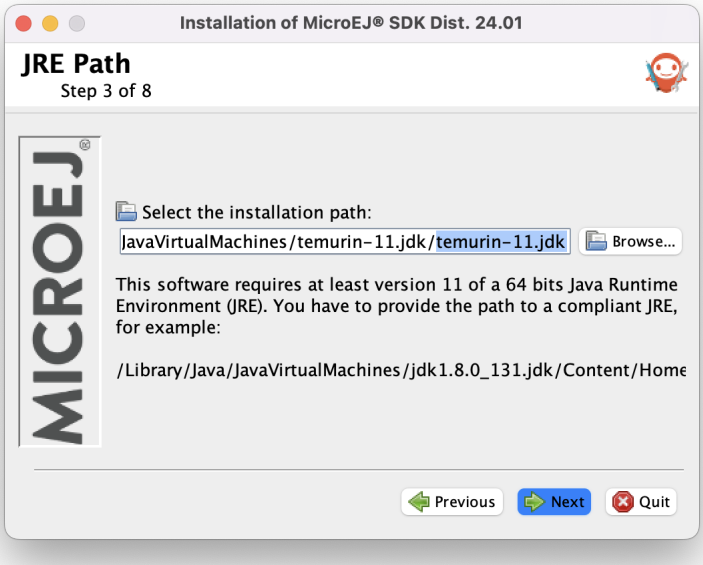


Fig. 303: Duplicated directory in path on MacOS

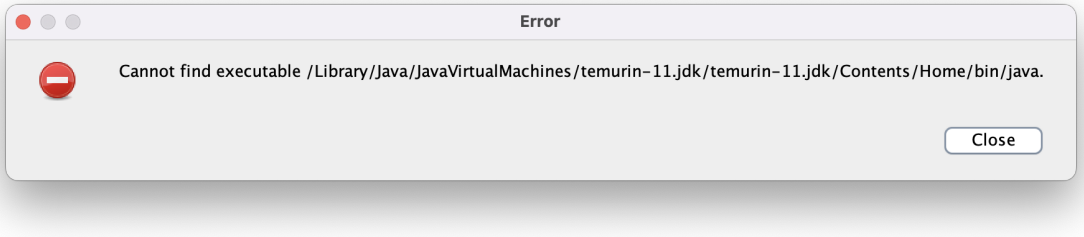


Fig. 304: Error when path is not a valid JDK path

Be sure the path points to JDK root dir, and not to Contents/Home , and click next.

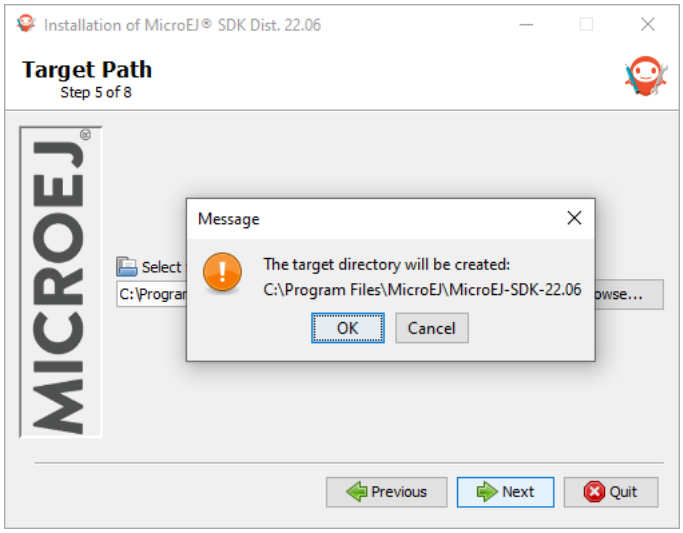


Fig. 305: Confirm your installation path

- Wait until the installation is done. Then click on the Next button.

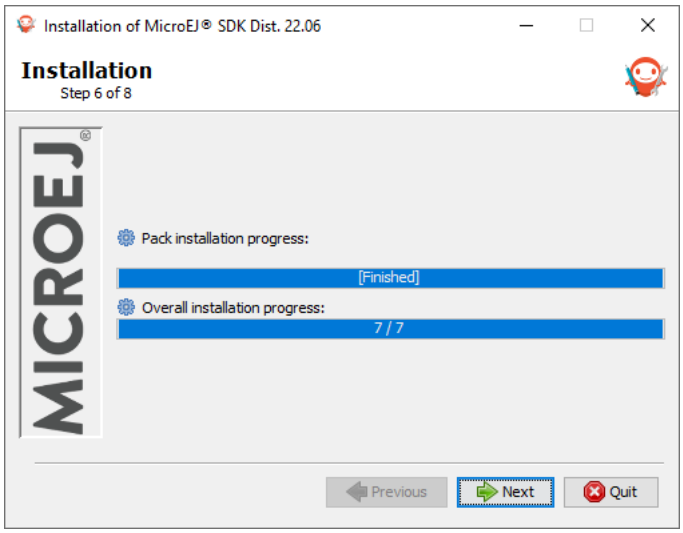


Fig. 306: Installation in progress

- Select options depending on your own preferences. Then click on the Next button.

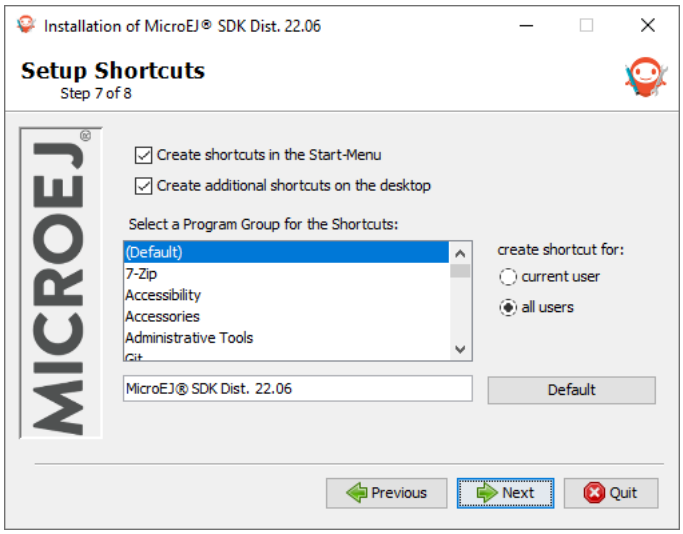


Fig. 307: Select the options

- The installation has completed successfully. Click on the Done button.

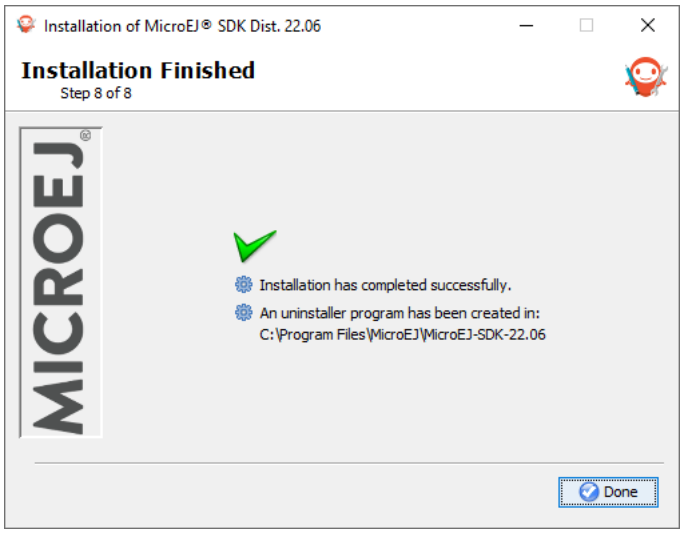


Fig. 308: Your installation has completed successfully

The SDK Distribution is now installed on your computer. You can launch it from your application launcher or by executing the `MicroEJ` executable in the installation path.

Warning: SDK 5 fails to start after installation because the JVM classpath has not been correctly set in configuration file. To fix it, you need to

- browse SDK 5 files by selecting `Show Package Contents` in the right click menu of `MicroEJ-SDK-24.01.app` .

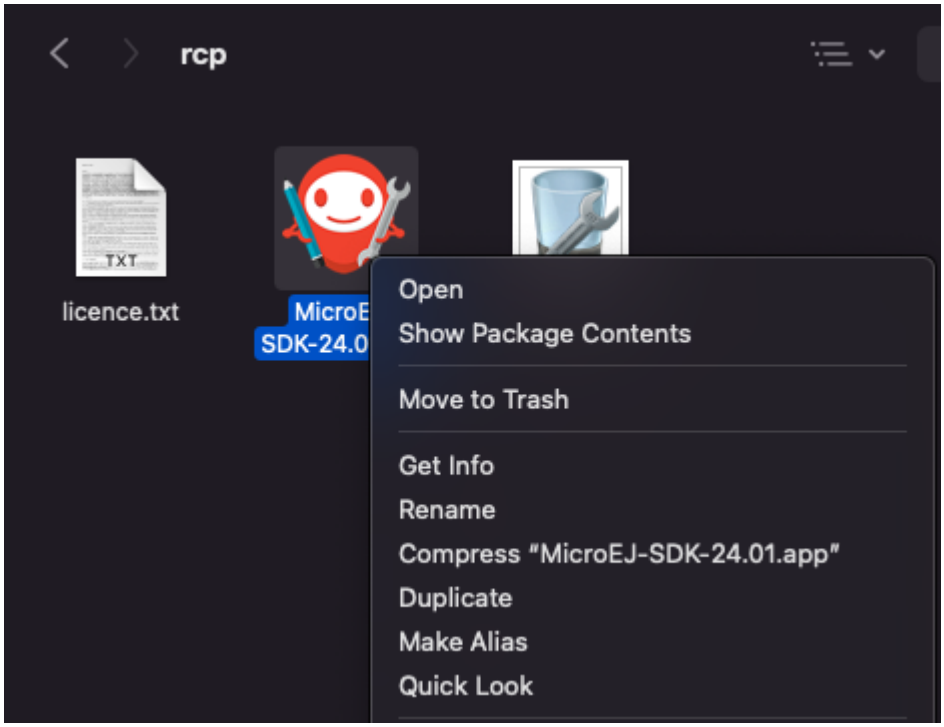


Fig. 309: Show Package Contents of MicroEJ-SDK-24.01.app

- edit Contents/Eclipse/eclipse.ini and add Contents/Home to the path defined for the -vm argument. For instance

```
-vm
/Library/Java/JavaVirtualMachines/temurin-11.jdk/Contents/Home/bin
```

Once the SDK is started, it is recommended to check if updates are available (see *Update SDK Version* section). If your are running SDK on Windows OS, it is also strongly recommended to configure *Windows defender exclusion rules*.

Update SDK Version

Once you have an SDK Distribution installed, you can update the included SDK Version to a newer version.

Note: If you want to know which SDK version is currently installed in your SDK Distribution, see the *SDK Version* chapter.

To update your SDK Distribution to a newer SDK version, follow the next steps:

- Select `Help` > `Check for updates` .

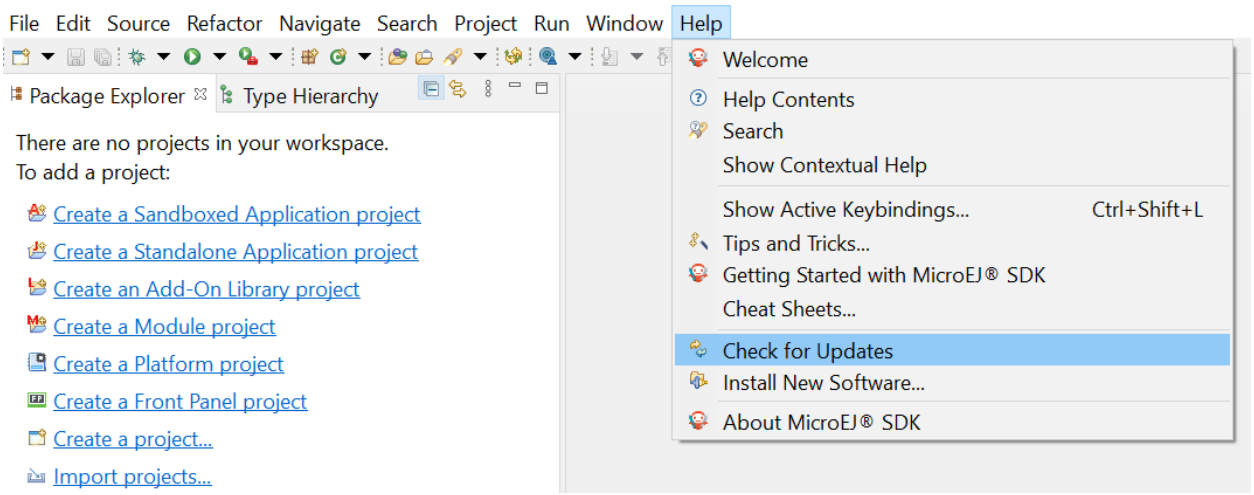


Fig. 310: Check for updates

- If your SDK is up-to-date, you will see the following screen:

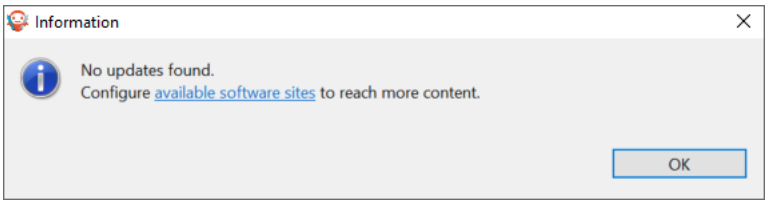


Fig. 311: No update available

- If an update is available, you will see the following screen:

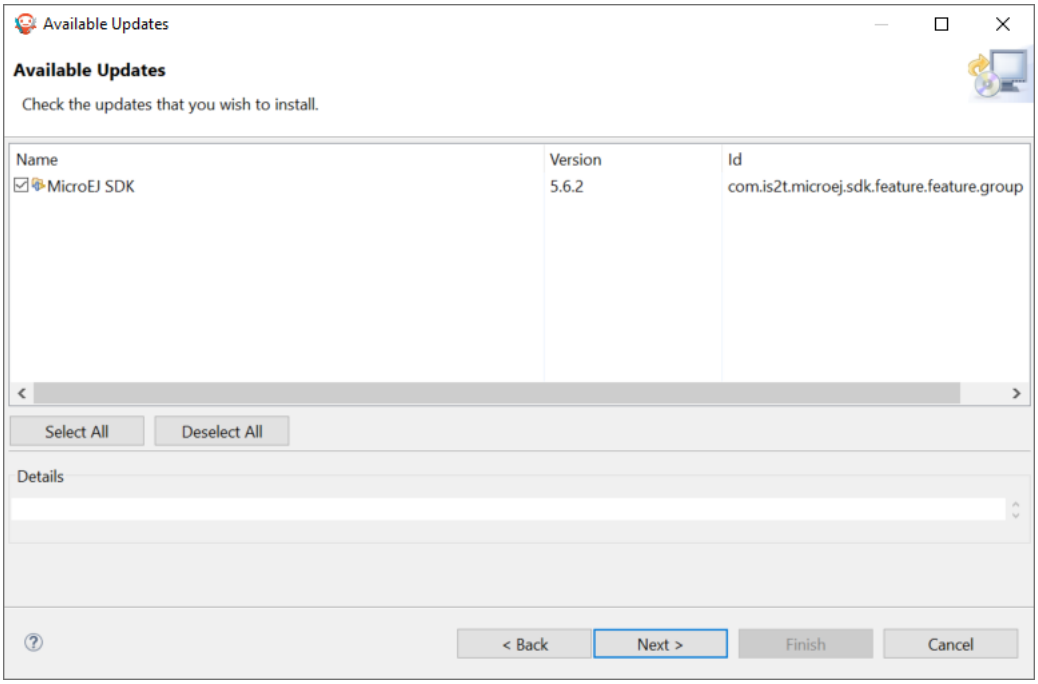


Fig. 312: Update available

- Check the version you want to install. Then click on the `Next` button.
- Review and confirm the updates. Then click on the `Next` button.

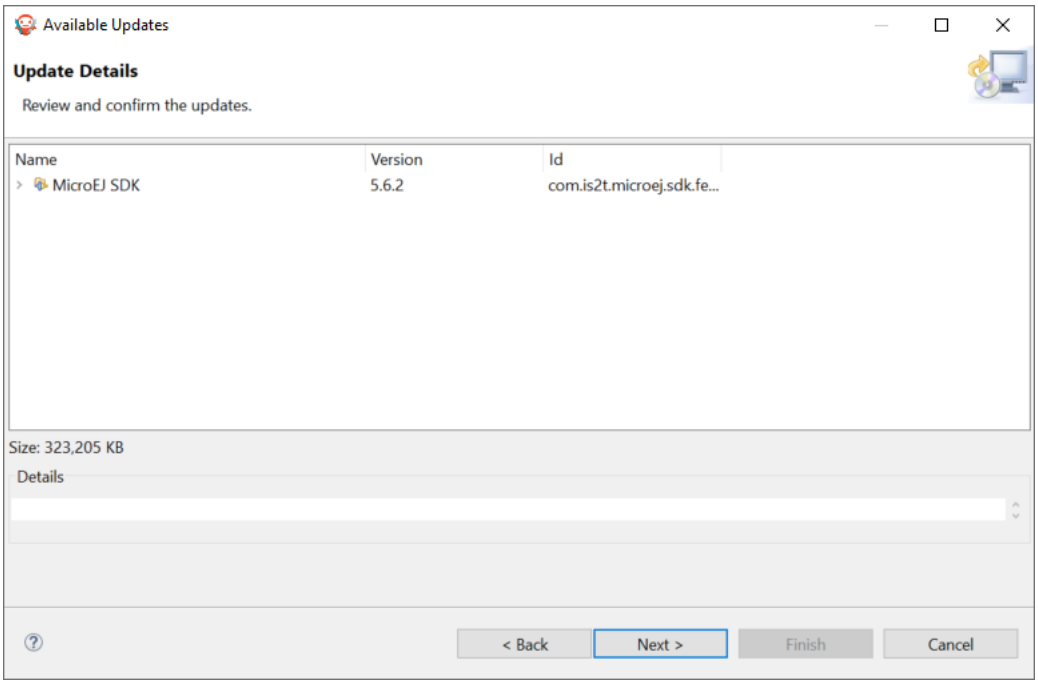


Fig. 313: Review the updates

- Select `I accept the terms of the license agreements.` . Then click on the `Finish` button.

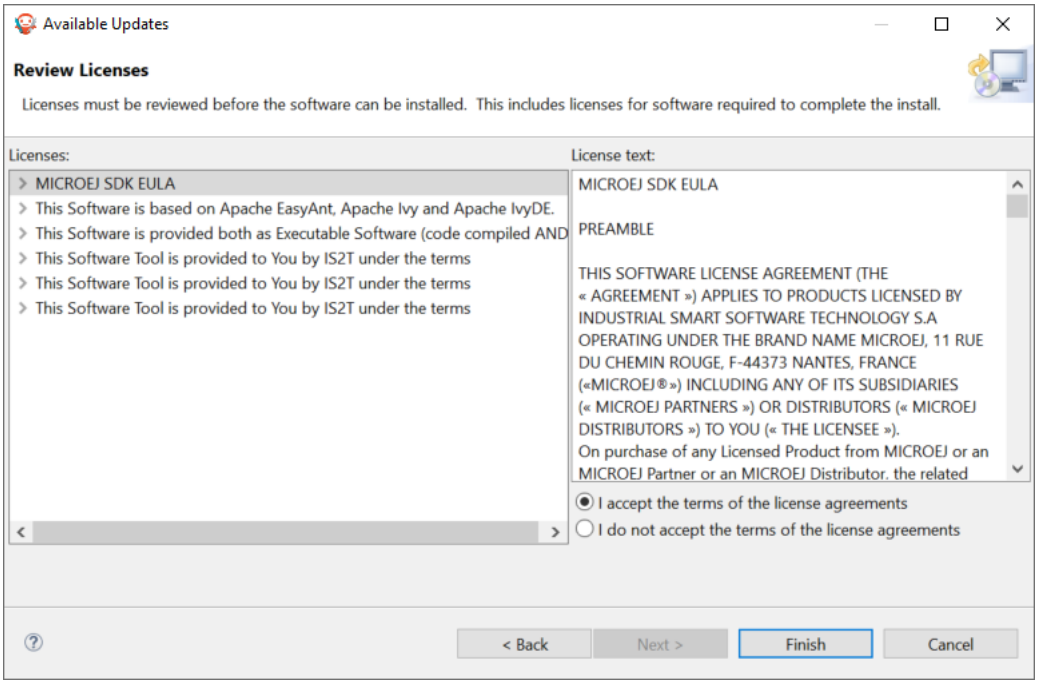


Fig. 314: Accept the terms of the license agreement

- Wait until the Software Update pop-up appears. Then click on the Restart Now button.

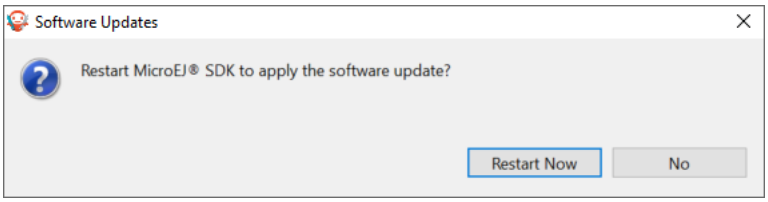


Fig. 315: Restart your SDK.

The update of your SDK is done.

Install Other SDK Distributions

Install Portable SDK Distribution

The portable package allows you to install the SDK Distribution when the use of the SDK Distribution installer is not possible or not desired, for example:

- you do not have administrator privileges on your workstation;
- you want to install SDK Distribution 23.07 but JDK 11 is not your default JDK version;
- you want to install SDK Distribution up to 21.11 but JDK 8 is not your default JDK version.

Perform the following steps:

- Download the Portable SDK Distribution for your operating system:

SDK Distribution	JDK Version	Windows	Linux	macOS x86_64 (Intel chip)	macOS aarch64 (M1 chip)[1]
24.01	11	Portable (.zip)	Portable (.zip)	Portable (.zip)	Portable (.zip)
23.07	11	Portable (.zip)	Portable (.zip)	Portable (.zip)	Portable (.zip)
21.11	8	Portable (.zip)	Portable (.zip)	Portable (.zip)	*N/A*

- Once downloaded, extract the zip file in a local directory of your choice
- Edit the MicroEJ-SDK.ini file
- Configure the path to the JDK version indicated above by adding the option -vm at the beginning of the file. If you don't have any JDK installed, see the *Get JDK* section.

```
-vm
[path_to_jdk]/bin
-startup
plugins/org.eclipse.equinox.launcher_1.6.400.v20210924-0641.jar
...
```

- Start the SDK by executing MicroEJ-SDK.exe on Windows or MicroEJ-SDK on Linux or macOS.

  Once the SDK is started, it is recommended to check if updates are available (see *Update SDK Version* section). If your are running SDK on Windows OS, it is also strongly recommended to configure *Windows defender exclusion rules*.

Install SDK Distribution 21.11

This section will guide you through the installation process of the SDK Distribution 21.11 using the step-by-step executable installer.

The SDK Distribution 21.11 requires a JRE or a JDK 8 and is not available for macOS with M1 chips. See the *System Requirements* page for more information on the list of supported environments.

Note: Launching the SDK Distribution installer requires administrator privileges and a JDK 8 installed by default on your workstation. If you don't have one of them or if you do not want to modify your default settings, please jump to *Install Portable SDK Distribution* section.

[1] SDK Distribution for macOS aarch64 (M1 chip) requires Architecture 7.18.0 or higher.

Download SDK Distribution

Download the SDK Distribution 21.11 installer for your operating system:

- Windows (.exe)
- Linux (.zip)
- macOS x86_64 - Intel chip (.zip)

Check JDK Version

The SDK Distribution 21.11 installer requires a JDK 8 installed by default on your workstation. If you don't have any JDK installed, see the *Get JDK* section.

Check the default Java version by running the following command in a new terminal:

```
> java -version

java version "1.8.0_281"
Java(TM) SE Runtime Environment (build 1.8.0_281-b09)
Java HotSpot(TM) 64-Bit Server VM (build 25.281-b09, mixed mode)
```

Now you can proceed with the installation steps.

Install SDK Distribution

- Launch the installer executable
  - On Windows, start MicroEJ-SDK-Installer-Win64-21.11.exe .
  - On Linux, unzip MicroEJ-SDK-Installer-Linux64-21.11.zip and start MicroEJ-SDK-Installer-Linux64-21.11.sh .
  - On macOS, unzip MicroEJ-SDK-Installer-Linux64-21.11.zip and start MicroEJ-SDK-Installer-MacOS-21.11.app . In case of error, check your app has not been put in quarantine (see *macOS troubleshooting* section).

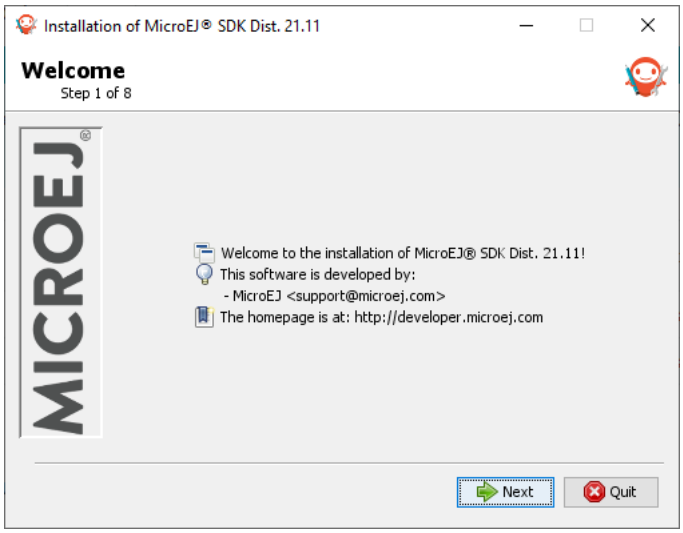


Fig. 316: Welcome to the installer

- Click on the Next button.
- Select I accept the terms of this license agreement. . Then click on the Next button.

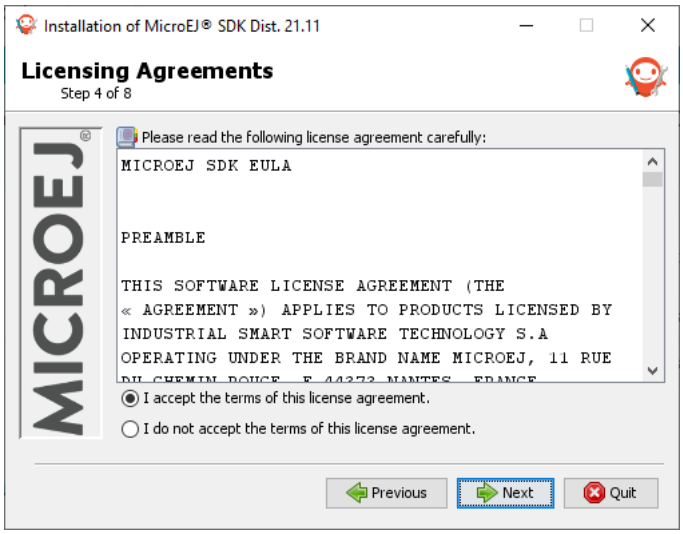


Fig. 317: Accept the terms of this license agreement

- Select the installation path of your SDK. By default it is C:\Program Files\MicroEJ\MicroEJ-SDK-21.11 for Windows. Then click on the Next button.

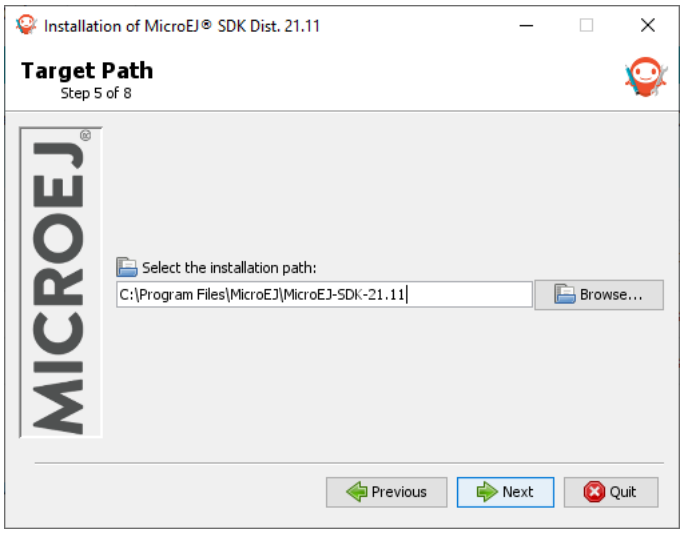


Fig. 318: Choose the installation path

- Click on the OK button to confirm the installation path.

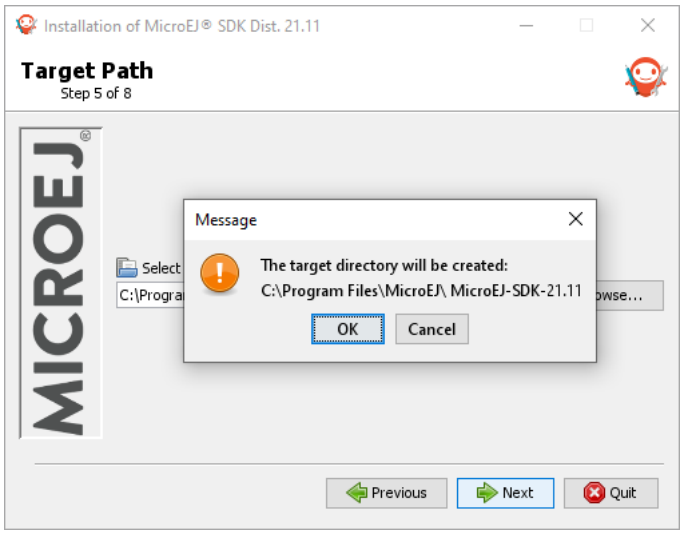


Fig. 319: Confirm your installation path

- Wait until the installation is done. Then click on the Next button.

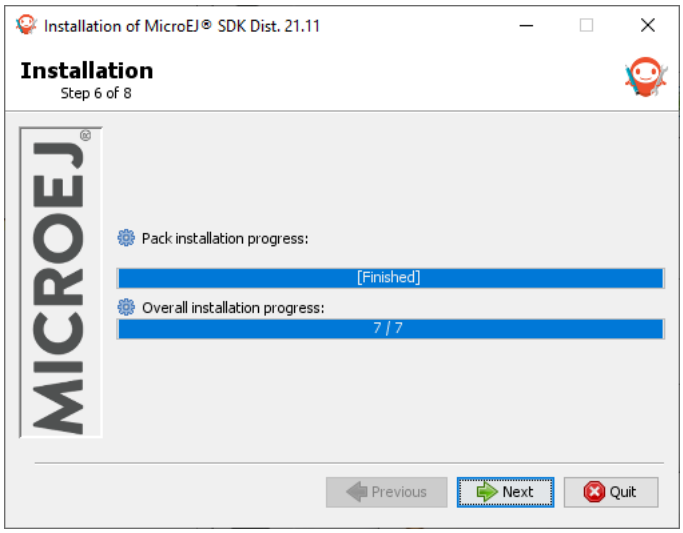


Fig. 320: Installation in progress

- Select options depending on your own preferences. Then click on the Next button.

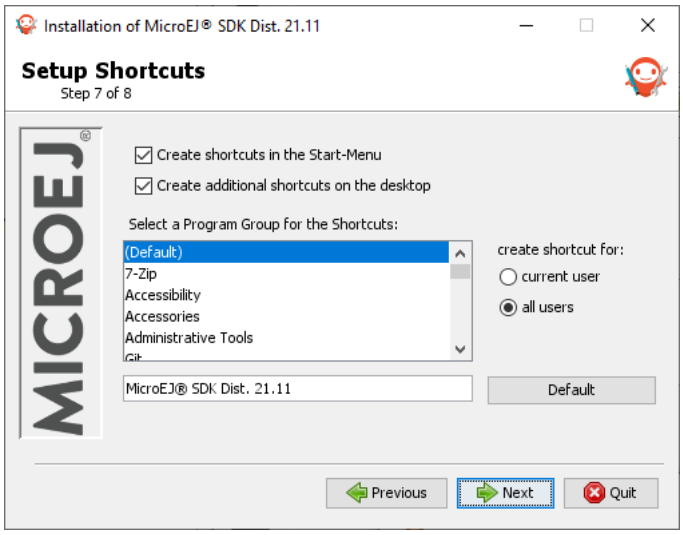


Fig. 321: Select the options

- The installation has completed successfully. Click on the Done button.

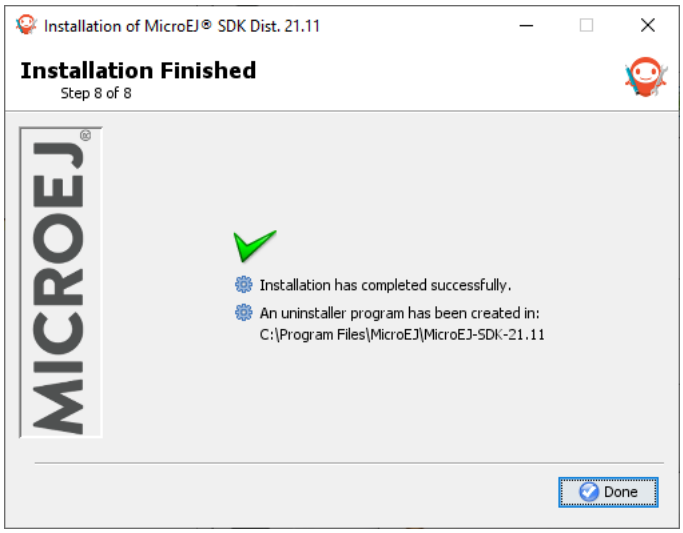


Fig. 322: Your installation has completed successfully

The SDK Distribution is now installed on your computer. You can launch it from your application launcher or by executing the `MicroEJ` executable in the installation path.

Once the SDK is started, it is recommended to check if updates are available (see *Update SDK Version* section). If your are running SDK on Windows OS, it is also strongly recommended to configure *Windows defender exclusion rules*.

This section applies when the installation of the *latest SDK Distribution* via the installer does not fit your case:

- you want to install the latest SDK Distribution compatible with JDK8, see *Install SDK Distribution 21.11* section.
- you want to install an SDK Distribution with no native installer, see *Install Portable SDK Distribution* section.
- you want to install an old SDK Distribution. The following table gives you access to all the SDK 5.x Distributions download links.

SDK Distribution	JDK Version	Windows	Linux	macOS x86_64 (Intel chip)	macOS aarch64 (M1 chip)[1]	SDK Version	Eclipse Version
24.01	11	• Installer (.exe) • Portable (.zip)	• Installer (.zip) • Portable (.zip)	• Installer (.zip) • Portable (.zip)	• Installer (.zip) • Portable (.zip)	5.8.2	2022-03
23.07	11	• Installer (.exe) • Portable (.zip)	• Installer (.zip) • Portable (.zip)	• Installer (.zip) • Portable (.zip)	• Installer (.zip) • Portable (.zip)	5.8.0	2022-03
23.02	11	• Installer (.exe) • Portable (.zip)	• Installer (.zip) • Portable (.zip)	• Installer (.zip) • Portable (.zip)	• Installer (.zip) • Portable (.zip)	5.7.0	2022-12
22.06	11	• Installer (.exe) • Portable (.zip)	• Installer (.zip) • Portable (.zip)	• Installer (.zip) • Portable (.zip)	• Installer (.zip) • Portable (.zip)	5.6.0	2022-03
21.11	8	• Installer (.exe) • Portable (.zip)	• Installer (.zip) • Portable (.zip)	• Installer (.zip) • Portable (.zip)	*N/A*	5.5.0	2020-06
21.03	8	• Installer (.exe) • Portable (.zip)	• Installer (.zip) • Portable (.zip)	• Installer (.zip) • Portable (.zip)	*N/A*	5.4.0	2020-06
20.12	8	• Installer (.exe) • Portable (.zip)	• Installer (.zip) • Portable (.zip)	• Installer (.zip) • Portable (.zip)	*N/A*	5.3.1	2020-06
20.10	8	• Installer (.exe) • Portable (.zip)	• Installer (.zip) • Portable (.zip)	• Installer (.zip) • Portable (.zip)	*N/A*	5.3.0	2020-06
20.07	8	• Installer	• Installer	• Installer	*N/A*	5.2.0	4.7.2

Finally, if you need an older SDK Distribution, browse the SDK Downloads Page.

System Requirements

- Hardware :
  - Intel x64 (Dual-core i5 minimum) or macOS AArch64 (M1) processor
  - 4GB RAM (minimum)
  - 2GB Disk (minimum)
- Operating Systems :
  - Windows 11, Windows 10, Windows 8.1 or Windows 8
  - Linux distributions (tested on Ubuntu 18.04, 20.04 and 22.04)
    - As of SDK Distribution 20.10 (based on Eclipse 2020-06), Ubuntu 16.04 is not supported.
    - For headless installations, the libwebkit2gtk-4.0-37 package is required additionally.
    - The OpenJDK distribution from Ubuntu is not supported. The Eclipse Temurin distribution is recommended.
  - macOS x86_64 with Intel chip (tested on version 10.13 High Sierra, 10.14 Mojave)
  - macOS aarch64 with M1 chip (tested on version 12.0.1 Monterey), from SDK Distribution 22.06 (requires *Architecture 7.18.0 or higher*)
  - **Java Runtime Environment :**

    The compatible JRE/JDK version depends on the Distribution, the SDK and the Architecture version. This table lists the supported combinations:

Distribution	SDK	Architecture	JRE/JDK
>= 22.06	>= 5.6.0	>= 7.17.0	JDK 11
<= 21.11	>= 5.6.0	>= 7.17.0	JRE or JDK 8 or 11
<= 21.11	< 5.6.0	*	JRE or JDK 8
<= 21.11	*	< 7.17.0	JRE or JDK 8

    The combinations not listed here are not supported. For the supported combinations, tests have been done with both the Oracle and the Eclipse Adoptium JDK builds.

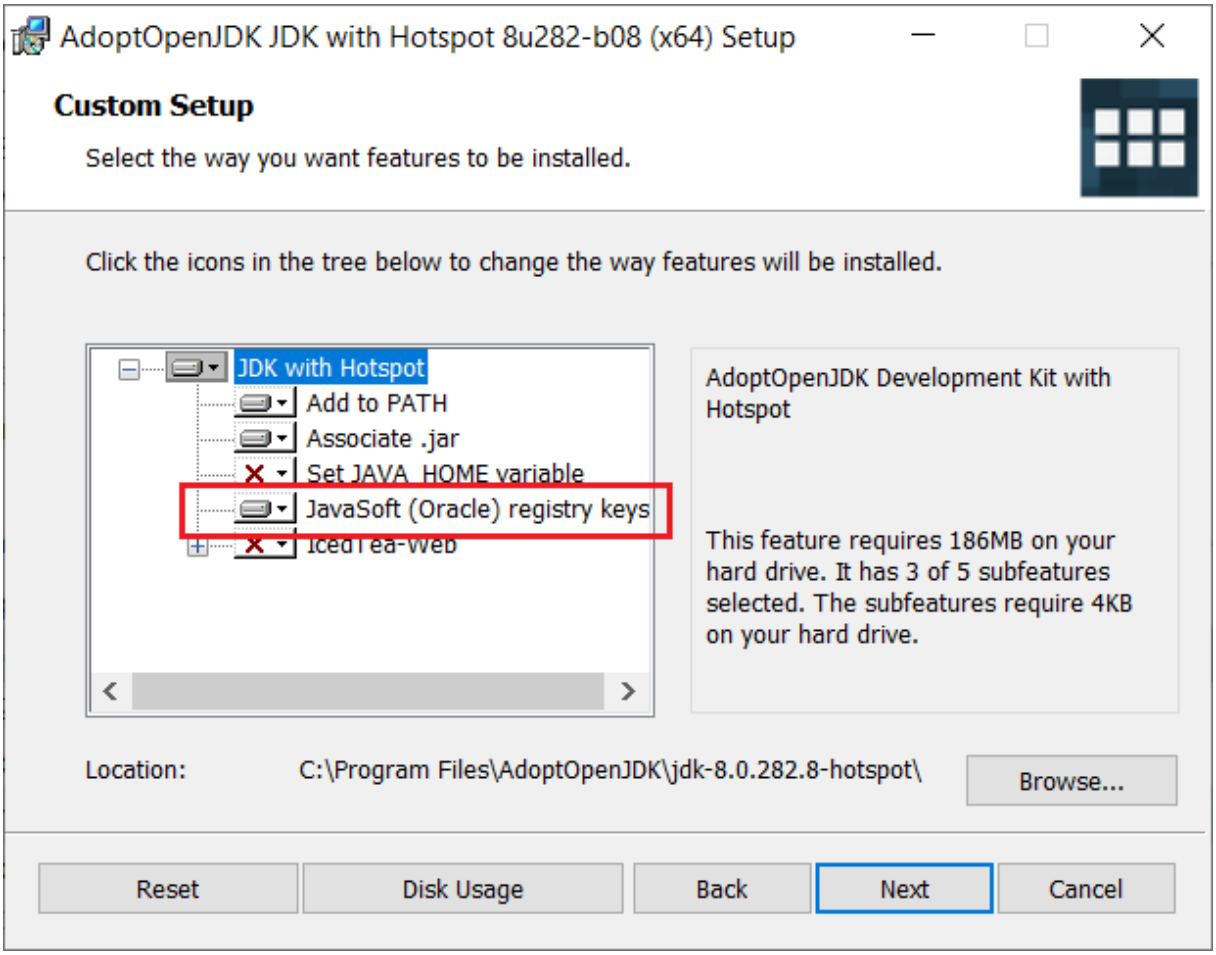
Warning: It is important to note that the SDK Distributions 22.06 and higher require a JDK 11 (not a JRE) and can be used only with an *Architecture* 7.17.0 or more.

[1] SDK Distribution for macOS aarch64 (M1 chip) requires Architecture 7.18.0 or higher.

Get JDK

You can download and install a JDK from Adoptium or Oracle.

Warning: Up to version 23.02 of the SDK Distribution, when installing the Eclipse Temurin/AdoptOpenJDK build on Windows, the option JavaSoft (Oracle) registry keys must be enabled:

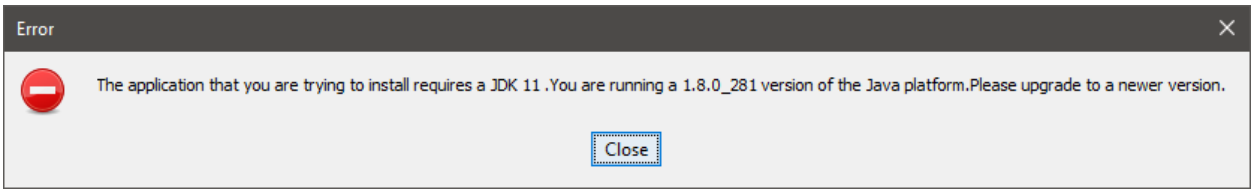


Without this option, the SDK installer cannot find the JDK and the message The application requires a Java Runtime Environment 8 is displayed.

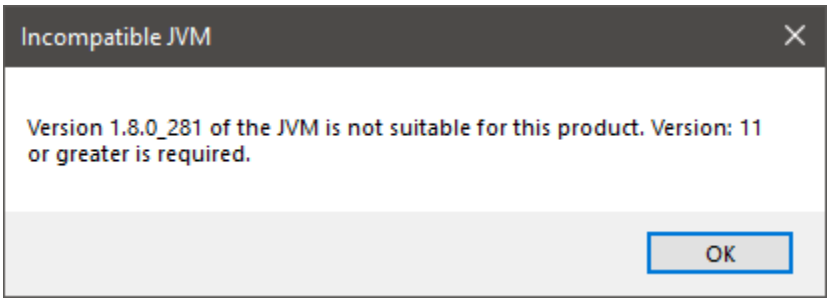
Troubleshooting

Incompatible Default Java Version

When launching the installer, you may get the following error: `The application you are trying to install requires a JDK11` .



Or when launching the SDK, you may get the following error: `Version: 11 or greater is required` .



The default Java version installed on your system is not compatible. You have two options:

- either install a JDK 11 as your default JVM. If you are on Windows OS and your SDK Distribution version is 23.02 or lower, ensure you enabled JavaSoft (Oracle) registry keys during the JDK installation (see *Get JDK*),
- or install the *portable SDK Distribution* if you don't want to modify your default JVM version.

  This latter case is recommended if you are installing SDK Dist. 22.06 or higher while you already have active projects based on SDK Dist. 21.11 .

Windows Specifics

If you are using Windows Defender as your default antivirus software, the SDK may be slowed down as it manipulates lots of JAR files (which are ZIP files) that are regularly analyzed.

To improve the SDK experience, please find below a list of folders that should be excluded from Windows Defender monitoring:

- %USERPROFILE%\.eclipse
- %USERPROFILE%\.ivy2
- %USERPROFILE%\.microej
- %USERPROFILE%\.p2
- %USERPROFILE%\AppData\Local\Temp\microej

- C:\Program Files\MicroEJ or the custom directory where the SDK has been installed
- your workspace(s) folder(s)

The exclusion page is available in the Settings application (Windows Security > Virus & threat protection > Manage settings > Exclusions > Add or remove exclusions).

Linux Specifics

Starting the SDK on a linux distribution may produce troubles such as missing content pages. This is related to incomplete Eclipse SWT configuration (see Eclipse GTK wiki page).

One solution is to configure Eclipse as follows:

- Add the next lines to MicroEJ-SDK.ini , before -vmargs argument:

```
--launcher.GTK_Version 2
```

- Ensure GTK is correctly installed (libwebkitgtk packet)
- Configure the following environment variables

```
MOZILLA_FIVE_HOME=/usr/lib/mozilla
LD_LIBRARY_PATH=${MOZILLA_FIVE_HOME}:${LD_LIBRARY_PATH}
```

- Restart the SDK
- Check there is not more SWT/MOZILLA related errors (Window > Show View > Other... > General > Error Log)

MacOS Specifics

When launching the SDK using the .app file, you may encounter the following message:

```
"MicroEJ-SDK-xx.xx" is damaged and can't be opened. You should move it to the Trash.
```

or this one:

```
"MicroEJ-SDK-xx.xx" cannot be opened because the developer cannot be verified.
```

This is due to macOS putting applications in quarantine when downloaded with a browser. Use this command to remove the SDK application from quarantine:

```
sudo xattr -rd com.apple.quarantine sdk.app
```

where sdk.app is the SDK file name.

3.11.2 Licenses

SDK EULA

MICROEJ SDK is licensed under the SDK End User License Agreement (EULA), which covers the following elements:

- SDK Tools & Plugins packaged in the SDK *5.x* Version,
- Architectures,
- Modules published to the *Central Repository* with the SDK EULA license, such as GUI or Networking Pack (see *Central Repository Licensing* for more details).

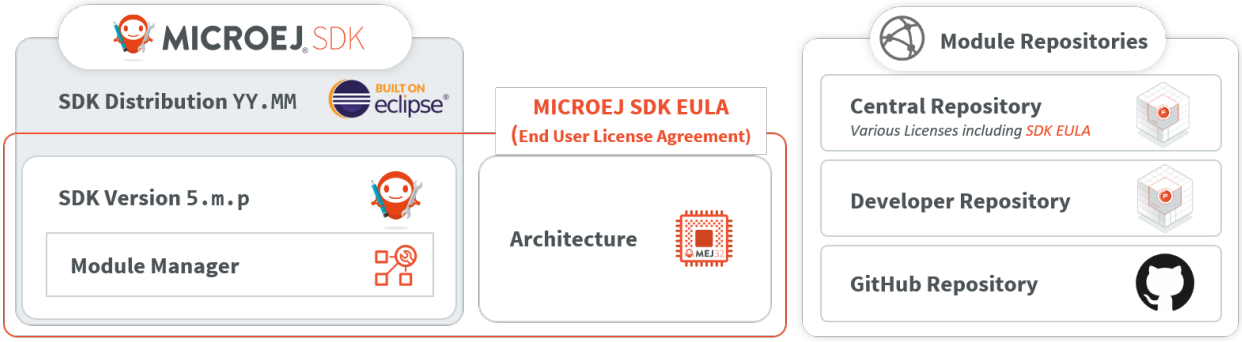


Fig. 323: SDK EULA Coverage

License Manager Overview

Architectures are distributed in two different versions:

- Evaluation Architectures, associated with a software license key. They can be downloaded at https://repository.microej.com/modules/com/microej/architecture/.
- Production Architectures, associated with a hardware license key stored on a USB dongle. They can be requested to *our support team*.

The license manager is provided with Architectures and then integrated into VEE Ports, consequently:

- Evaluation licenses will be shown only if at least one Evaluation Architecture or VEE Port built from an Evaluation Architecture has been imported in the SDK.
- Production licenses will be shown only if at least one Production Architecture or VEE Port built from a Production Architecture has been imported in the SDK.

The list of installed licenses is available in the SDK preferences dialog page in `Window` > `Preferences` > `MicroEJ` :

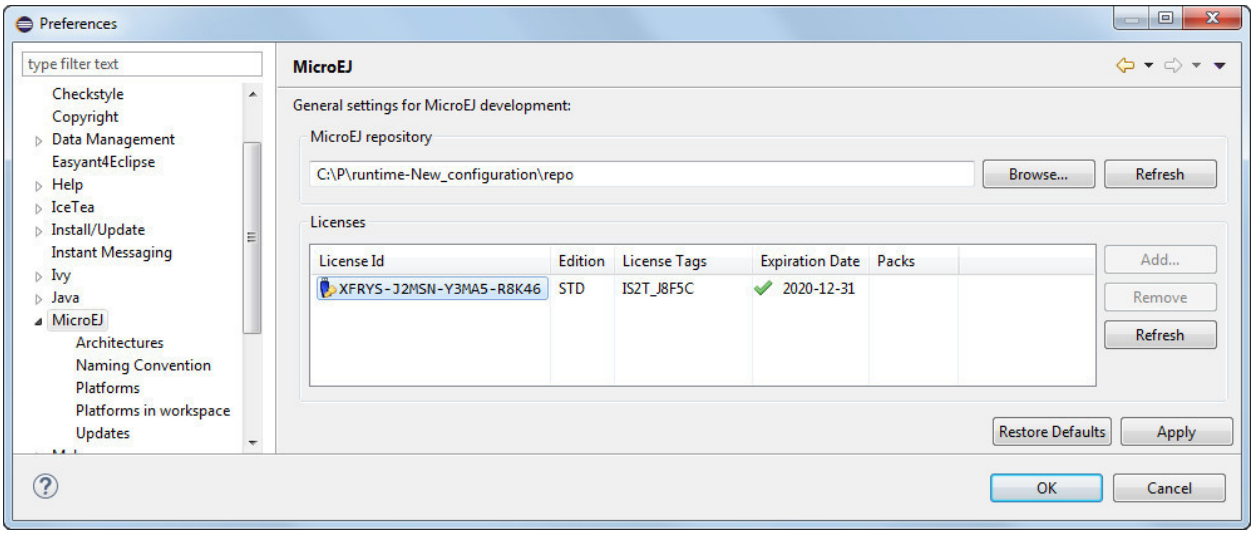


Fig. 324: License Manager View

License Check

The table below summarizes where the license is checked.

Application	Run on Simulator (Virtual Device)	Build on Device	Documentation Link
Standalone Application or Kernel Application	NO	**YES**	*Run on the Device*
Sandboxed Application	NO	NO	*Application Linking*

Evaluation Licenses

This section should be considered when using Evaluation Architectures, which use software license keys. A machine UID needs to be provided to activate an Evaluation license on the MicroEJ Licenses Server. The machine UID is a 16 hexadecimal digits number.

Get your Machine UID

Retrieving the machine UID depends on the kind of VEE Port being evaluated.

If your VEE Port is already *imported in Package Explorer* and built with *MicroEJ Module Manager*, the Architecture has been automatically imported. The machine UID will be displayed when building a *Standalone Application on device*.

```
[INFO ] Launching in Evaluation mode. Your UID is XXXXXXXXXXXXXXXX.
[ERROR] Invalid license check (No license found).
```

Otherwise, an Architecture or VEE Port should have been manually imported from the SDK preferences page. The machine UID can be retrieved as follows:

- Go to `Window` > `Preferences` > `MicroEJ` ,

- Select either Architectures , Platforms in workspace or Platforms ,
- Click on one of the available items,
- Press the Get UID button to get the machine UID.

Note: To access this Get UID option, at least one Evaluation Architecture or VEE Port must have been imported before (see *License Manager Overview*).

Copy the UID. It will be needed when requesting a license.

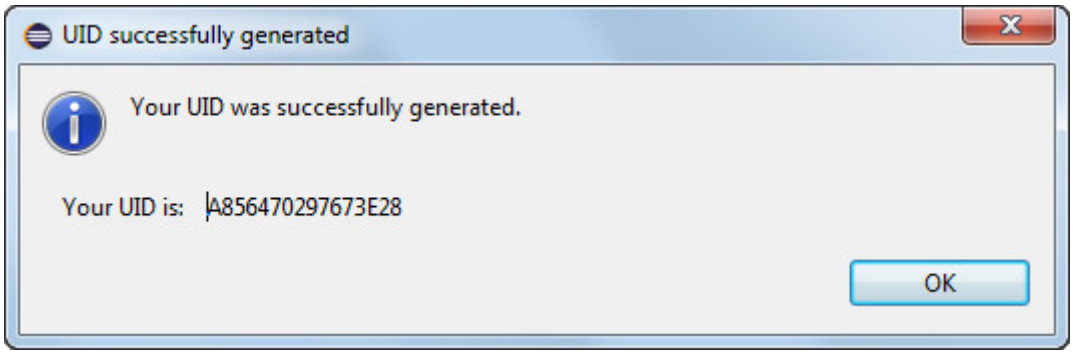


Fig. 325: Machine UID for Evaluation License

Request your Activation Key

- Go to MicroEJ Licenses Server https://license.microej.com.
- Click on Create a new account link.
- Create your account with a valid email address. You will receive a confirmation email a few minutes after. Click on the confirmation link in the email and log in with your new account.
- Click on Activate a License .
- Set Product P/N: to 9PEVNLDBU6IJ .
- Set UID: to the machine UID you copied before.
- Click on Activate .
- The license is being activated. You should receive your activation by email in less than 5 minutes. If not, please contact *our support team*.
- Once received by email, save the attached zip file that contains your activation key.

Install the License Key

If your VEE Port is already *imported in Package Explorer* and built with *MicroEJ Module Manager*, the license key zip file must be simply dropped to the `~/.microej/licenses/` directory (create it if it doesn't exist).

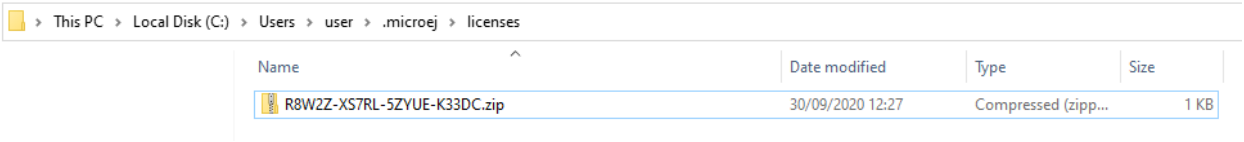


Fig. 326: MicroEJ Shared Licenses Directory

Note: The SDK Preferences page will be automatically refreshed when building a *Standalone Application on device*.

Otherwise, the license key must be installed as follows:

- Go back to the SDK.
- Select the Window > Preferences > MicroEJ menu.
- Press Add… .
- Browse the previously downloaded activation key archive file.
- Press OK. A new license is successfully installed.
- Go to Architectures sub-menu and check that all Architectures are now activated (green check).
- Your SDK is successfully activated.

  If an error message appears, the license key could not be installed. (see section *Troubleshooting*). A license key can be removed from the key-store by selecting it and by clicking on Remove button.

Troubleshooting

Unable to add an Evaluation license key in the SDK

Consider this section when an error message appears while adding the Evaluation license key. Before contacting *our support team*, please check the following conditions:

- Key is corrupted (wrong copy/paste, missing characters, or extra characters)
- Key has not been generated for the installed environment
- Key has not been generated with the machine UID
- Machine UID has changed since submitting license request and no longer matches license key
- Key has not been generated for one of the installed Architectures (no license manager able to load this license)

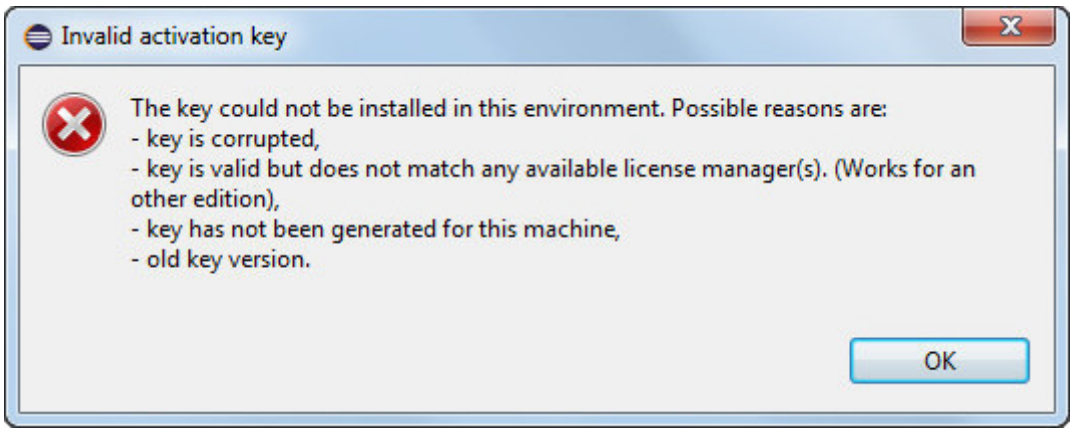


Fig. 327: Invalid License Key Error Message

Machine UID has changed

This can occur when the hardware configuration of the machine is changed (especially when the network interfaces have changed).

In this case, you can either request a new activation key for this new UID or go back to the previous hardware configuration.

Production Licenses

This section should be considered when using Production Architectures, which use hardware license keys stored on a USB dongle.

Fig. 328: MicroEJ USB Dongle

Note: If your USB dongle has been provided to you by your sales representative and you don't have received an activation certificate by email, it may be a pre-activated dongle. Then you can skip the activation steps and directly jump to the *Check Activation* section.

Request your Activation Key

- Go to license.microej.com.
- Click on `Create a new account` link.
- Create your account with a valid email address. You will receive a confirmation email a few minutes after. Click on the confirmation link in the email and login with your new account.
- Click on `Activate a License` .
- Set `Product P/N:` to **The P/N on the activation certificate**.
- Enter your UID: serial number printed on the USB dongle label (8 alphanumeric char.).
- Click on `Activate` and check the confirmation message.
- Click on `Confirm your registration` .
- Enter the **Registration Code provided on the activation certificate**.
- Click on `Submit` .
- Your Activation Key will be sent to you by email as soon as it is available (12 business hours max.).

Note: You can check the `My Products` page to verify your product registration status, the Activation Key availability, and download the Activation Key when available.

Once the Activation Key is available, download and save the Activation Key ZIP file to a local directory.

Activate your USB Dongle

This section contains instructions that will allow you to flash your USB dongle with the proper activation key.

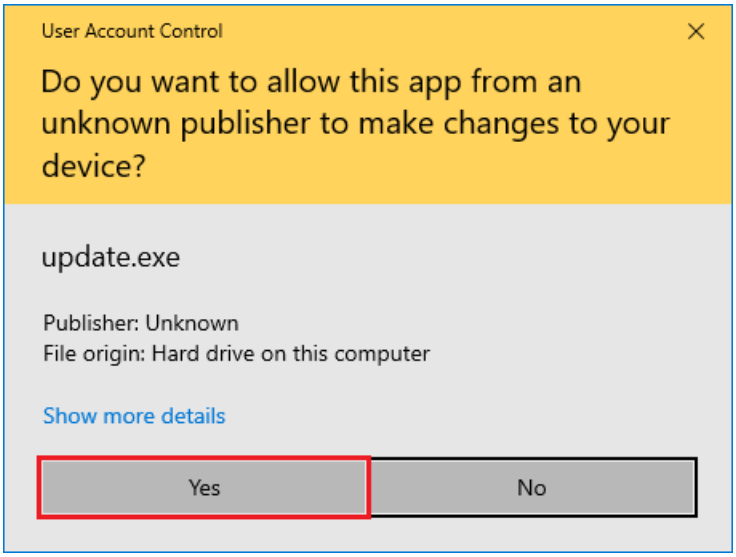
You shall ensure that the following prerequisites are met :

- Your *operating system* is Windows
- The USB dongle is plugged and recognized by your operating system (see *Troubleshooting* section)
- No more than one USB dongle is plugged into the computer while running the update tool
- The update tool is not launched from a network drive or a USB key
- The activation key you downloaded is the one for the dongle UID on the sticker attached to the dongle (each activation key is tied to the unique hardware ID of the dongle).

You can then proceed to the USB dongle update:

- Unzip the `Activation Key` file to a local directory
- Enter the directory just created by your ZIP extraction tool.
- Launch the executable program.

- Accept running the unsigned software if requested (Windows 10/11)



- Click on the `Update` button (no password needed)

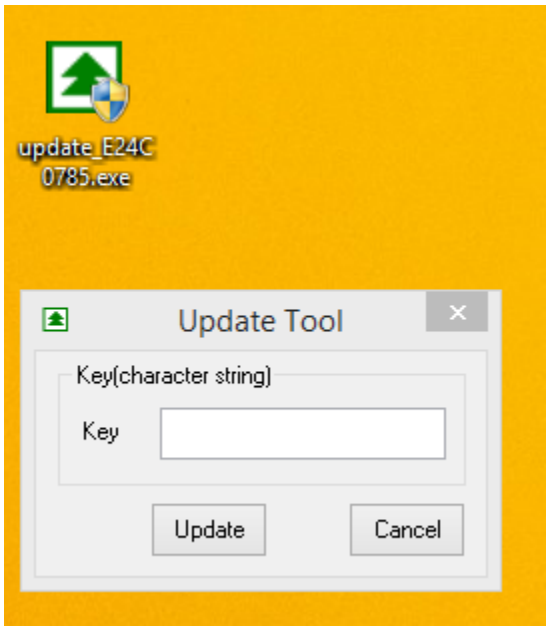


Fig. 329: Dongle Update Tool

- On success, an `Update successfully` message shall appear. On failure, an `Error key or no proper rockey` message may appear.

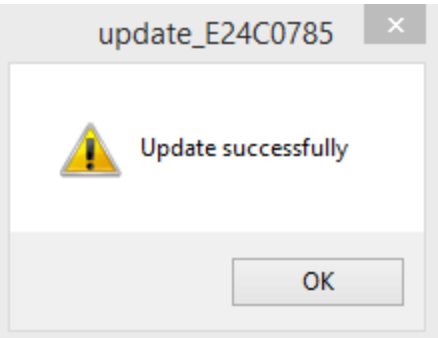


Fig. 330: Successful Dongle Update

Check Activation

This section contains instructions that will allow you to verify that your USB dongle has been properly activated.

Check Activation in the SDK

Note: Production licenses will be shown only if at least one Production Architecture or VEE Port has been imported before (see *License Manager Overview*).

In the SDK,

- Go to `Window` > `Preferences` > `MicroEJ` ,
- Go to `Architectures` , `Platforms in workspace` or `Platforms` sub-menu and check that all items are now activated (green check).

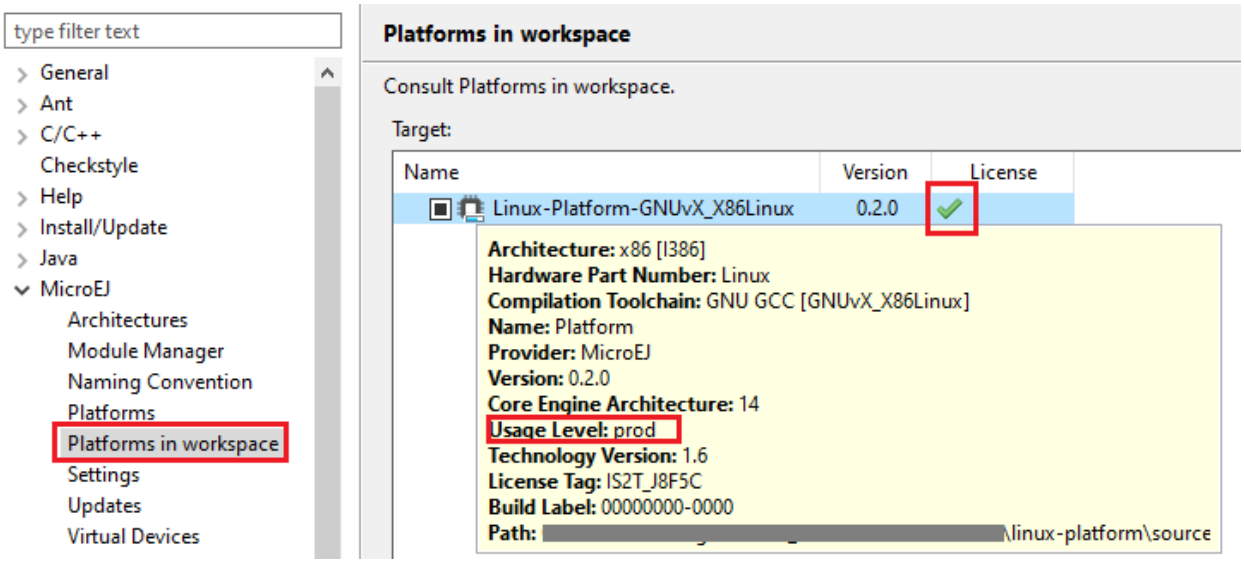


Fig. 331: License Status OK

If the license is still not recognized (red cross), check with the following command line tool to get more information.

Check Activation with the Command Line Tool

To get more details on connected USB dongle(s), run the debug tool as following:

1. Open a terminal.
2. Change directory to a Production VEE Port.
3. Execute the command:

   Architecture v8.1.0 or higher

   Architecture v8.0.0 or lower

```
java -Djava.
↪library.path=resources/os/[OS_NAME] -jar licenseManager/licenseManagerProduct.jar
```

```
java -Djava.
↪library.path=resources/os/[OS_NAME] -jar licenseManager/licenseManagerUsbDongle.jar
```

with OS_NAME set to Windows64 for Windows OS, Linux64 for Linux OS, Mac for macOS x86_64 (Intel chip) or MacA64 for macOS aarch64 (Apple Silicon chip).

If your USB dongle has been properly activated, you should get the following output:

```
[DEBUG] ===== MicroEJ Dongle Debug Tool =====
[DEBUG] => Detected dongle UID: XXXXXXXX.
[DEBUG] => Dongle UID has valid MicroEJ data: XXXXXXXX (only the first one is listed).
[DEBUG] =>␣
↪Detected MicroEJ License XXXXX-XXXXX-XXXXX-XXXXX - valid until YYYY-MM-DD.
[DEBUG] ===== SUCCESS =====
```

USB Dongle on GNU/Linux

For GNU/Linux Users (Ubuntu at least), by default, the dongle access has not been granted to the user, you have to modify udev rules. Please create a /etc/udev/rules.d/91-usbdongle.rules file with the following contents:

```
ACTION!="add", GOTO="usbdongle_end"
  SUBSYSTEM=="usb", GOTO="usbdongle_start"
  SUBSYSTEMS=="usb", GOTO="usbdongle_start"
  GOTO="usbdongle_end"

  LABEL="usbdongle_start"

  ATTRS{idVendor}=="096e" , ATTRS{idProduct}=="0006" , MODE="0666"

  LABEL="usbdongle_end"
```

Then, restart udev: sudo /etc/init.d/udev restart

You can check that the device is recognized by running the lsusb command. The output of the command should contain a line similar to the one below for each dongle: Bus 002 Device 003: ID 096e:0006 Feitian Technologies, Inc.

USB Dongle with Docker on Linux

If you use the SDK Docker image on a Linux host to build an Executable, the dongle must be mapped to the Docker container. First, it requires to add a symlink on the dongle by following the instructions of the *USB Dongle on GNU/Linux* section but with this `/etc/udev/rules.d/91-usbdongle.rules` file:

```
ACTION!="add", GOTO="usbdongle_end"
  SUBSYSTEM=="usb", GOTO="usbdongle_start"
  SUBSYSTEMS=="usb", GOTO="usbdongle_start"
  GOTO="usbdongle_end"

  LABEL="usbdongle_start"

  ATTRS{idVendor}==
↪"096e" , ATTRS{idProduct}=="0006" , MODE="0666" , SYMLINK+="microej_dongle"

  LABEL="usbdongle_end"
```

Then the symlink has to be mapped in the Docker container by adding the following option in the Docker container creation command line:

```
--device /dev/microej_dongle:/dev/bus/usb/999/microej_dongle
```

The `/dev/microej_dongle` symlink can be mapped to any device path as long as it is in `/dev/bus/usb`.

USB Dongle with WSL

Note: The following steps have been tested on WSL2 with Ubuntu 22.04.2 LTS.

To use a USB dongle with WSL, you first need to install *usbipd* following the steps described in Microsoft WSL documentation:

First, check that WSL2 is installed on your system. If not, install it or update it following Microsoft Documentation

Then, you need install usbipd-win v4.0.0 or higher on Windows from usbipd-win Github repository.

And then, install usbipd and update hardware database inside you WSL installation:

```
sudo apt install linux-tools-generic hwdata
sudo update-
↪alternatives --install /usr/local/bin/usbip usbip /usr/lib/linux-tools/*-generic/usbip 20
```

Add the udev rule described in *USB Dongle on GNU/Linux*, and restart udev:

```
sudo /etc/init.d/udev restart
```

Ensure your USB dongle is plugged, then start a PowerShell terminal in administrator mode.

List the connected devices with the following command:

```
usbipd.exe list
```

You should see your USB dongle connected with VID:PID==096e:0006 :

```
PS C:\Users\user> usbipd list
Connected:
BUSID  VID:PID    DEVICE                                                        STATE
2-
↪6     0c45:674c  Integrated Webcam, Integrated IR Webcam, USB DFU              Not shared
2-8    0a5c:5843  Dell ControlVault w/ Fingerprint Touch Sensor, Microsoft ...  Not shared
2-10   8087:0033  Intel(R) Wireless Bluetooth(R)                                Not shared
3-1    0bda:8153  Realtek USB GbE Family Controller                             Not shared
4-6    413c:c010  Dell DA310                                                    Not shared
6-4    096e:0006  USB Input Device                                              Not shared
6-6    046d:0819  USB Video Device, USB Audio Device                            Not shared
7-1    045e:0084  USB Input Device                                              Not shared
7-2    04d9:1400  USB Input Device                                              Not shared
7-3    10d5:55a2  USB Input Device                                              Not shared
```

Here the BUSID is 6-4 .

Bind and attach the dongle to WSL:

```
usbipd.exe bind --busid <BUSID>
usbipd.exe attach --wsl  --busid <BUSID>
```

Open a bash terminal in your WSL instance, and check the USB dongle is successfully mounted with the following command:

```
lsusb
```

You should see your USB dongle connected with ID 096e:0006 :

```
Bus 002 Device 001: ID 1d6b:0003 Linux Foundation 3.0 root hub
Bus 001 Device 002: ID␣
↪096e:0006 Feitian Technologies, Inc. HID Dongle (for OEMs - manufacturer string is "OEM")
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
```

Note: You'll need to follow these steps each time you system is rebooted or the dongle is plugged/unplugged.

Troubleshooting

This section contains instructions to check that your operating system correctly recognizes your USB dongle.

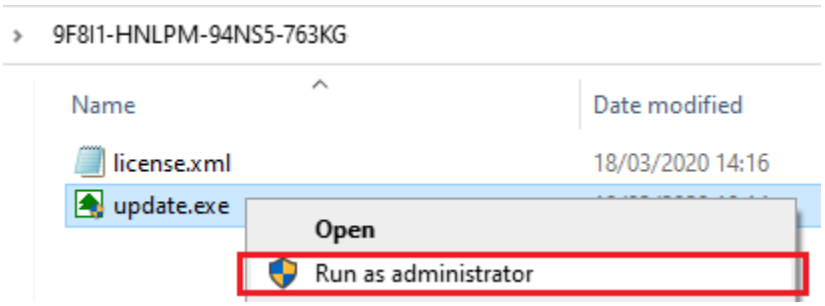
Windows Troubleshooting

- If the *dongle activation* failed with `No rockey` message, check there is one and only one dongle recognized with the following hardware ID :

```
HID\VID_096E&PID_0006&REV_0201
```

Go to the `Device Manager` > `Human Interface Devices` and check among the `USB Input Device` entries that the `Details` > `Hardware Ids` property match the ID mentioned before.

- If the *dongle activation* was successful with `Update successfully` message but the license does not appear in the SDK or is not updated, try to activate again by starting the executable with administrator privileges:



- If the following error message is thrown when building an Executable, either the dongle plugged is a verbatim dongle or it has not been successfully *activated*:

```
Invalid license check (Dongle found is not compatible).
```

VirtualBox Troubleshooting

In a VirtualBox virtual machine, USB drives must be enabled to be recognized correctly. Make sure to enable the USB dongle by clicking on it in the VirtualBox menu `Devices` > `USB` .

To make this setting persistent, go to `Devices` > `USB` > `USB Settings…` and add the USB dongle in the `USB Devices Filters` list.

WSL Troubleshooting

Check that your dongle is attached to WSL from PowerShell:

```
usbipd.exe list
```

You should have a line saying `Attached - Ubuntu` :

```
PS C:\Users\sdkuser> usbipd.exe list
BUSID  VID:PID    DEVICE                                                        STATE
2-1    096e:0006  USB Input Device                                              Attached - Ubuntu
2-6    0c45:6a10  Integrated Webcam                                             Not attached
2-10   8087:0026  Intel(R) Wireless Bluetooth(R)                                Not attached
3-1    045e:0823  USB Input Device                                              Not attached
3-4    046d:c31c  USB Input Device                                              Not attached
```

In you WSL console, the dongle must also be recognized. Ckeck by using `lsusb` :

```
skduser@host:~/workspaces/docs$ lsusb
Bus 002 Device 001: ID 1d6b:0003 Linux Foundation 3.0 root hub
Bus 001 Device 003: ID␣
↪096e:0006 Feitian Technologies, Inc. HID Dongle (for OEMs - manufacturer string is "OEM")
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
```

This might not be sufficient. If you're still facing license issues, restart udev, abd attach your dongle to WSL once again.

Note: Hibernation may have unattached your dongle. Reload udev, unplug/plug your dongle and attach it from PowerShell.

Dongle not detected in the licenses screen

If the USB dongle is plugged and activated but not visible in the menu `Window` > `Preferences` > `MicroEJ` , please check that you have an active VEE Port in `Window` > `Preferences` > `MicroEJ` > `Platforms in workspace` .

Then, ensure that the VEE Port has been built in `prod` configuration, this can be checked with the architecture dependency inside the file `module.ivy` . If no VEE Ports are visible in your current workspace, please build a VEE Port configured to the `prod` mode and this should fix the issue.

Remote USB Dongle Connection

When the dongle cannot be physically plugged to the machine running the SDK (cloud builds, virtualization, missing permissions, …), it can be configured using USB redirection over IP network.

There are many hardware and software solutions available on the market. Among others, this has been tested with https://www.net-usb.com/ and https://www.virtualhere.com/. Please contact *our support team* for more details.

3.11.3 Standalone Application

Platform Import

A Platform is required to run a Standalone Application on the Simulator or build the Firmware binary for the target device.

The *VEE Porting Guide* describes how to create a Platform from scratch for any kind of device. In addition, MicroEJ Corp. provides Platforms for various development boards (see https://repository.microej.com/index.php?resource=JPF).

Platforms are distributed in two packages:

- **Source Platform**. The source files are imported into the workspace. This is the default case.
- **Binary Platform**. A `.jpf` file is imported into the *MicroEJ repository*. As of MicroEJ SDK `5.3.0`, this package is deprecated.

Source Platform Import

Import from Folder

This section applies when the Platform files are already available on a local folder. This is likely the case when the files are checked out from a Version Control System, such as a local git repository clone.

Note: If you are going to import a Platform from MicroEJ Github, you can follow the specific *GitHub Repositories* section instead (the projects will be automatically imported).

- Select File > Import… > General > Existing Projects into Workspace > Select root directory > Browse… .
- Select the root directory. The wizard will automatically discover projects to import.
- Click on the Finish button.

Import from Zip File

This section applies when the Platform files are packaged in a `.zip` file.

- Select File > Import… > General > Existing Projects into Workspace > Select archive file > Browse… .
- Select the zip of the project (e.g., `x.zip`). The wizard will automatically discover projects to import.
- Click on the Finish button.

Platform Build

Platforms are usually shared with only the Platform configuration files. Once the projects are imported, follow the platform-specific documentation to build the Platform.

Once imported or built, a Platform project should be available as follows:

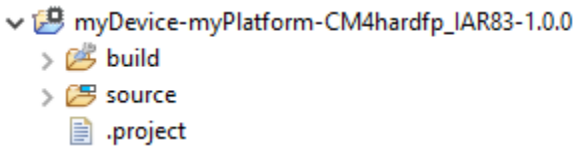


Fig. 332: Platform Project

The source folder contains the Platform content which can be set to the target.platform.dir option.

Binary Platform Import

After downloading the Platform .jpf file, launch MicroEJ SDK and follow these steps to import the Platform:

- Open the Platform view in MicroEJ SDK, select Window > Preferences > MicroEJ > Platforms . The view should be empty on a fresh install of the tool.

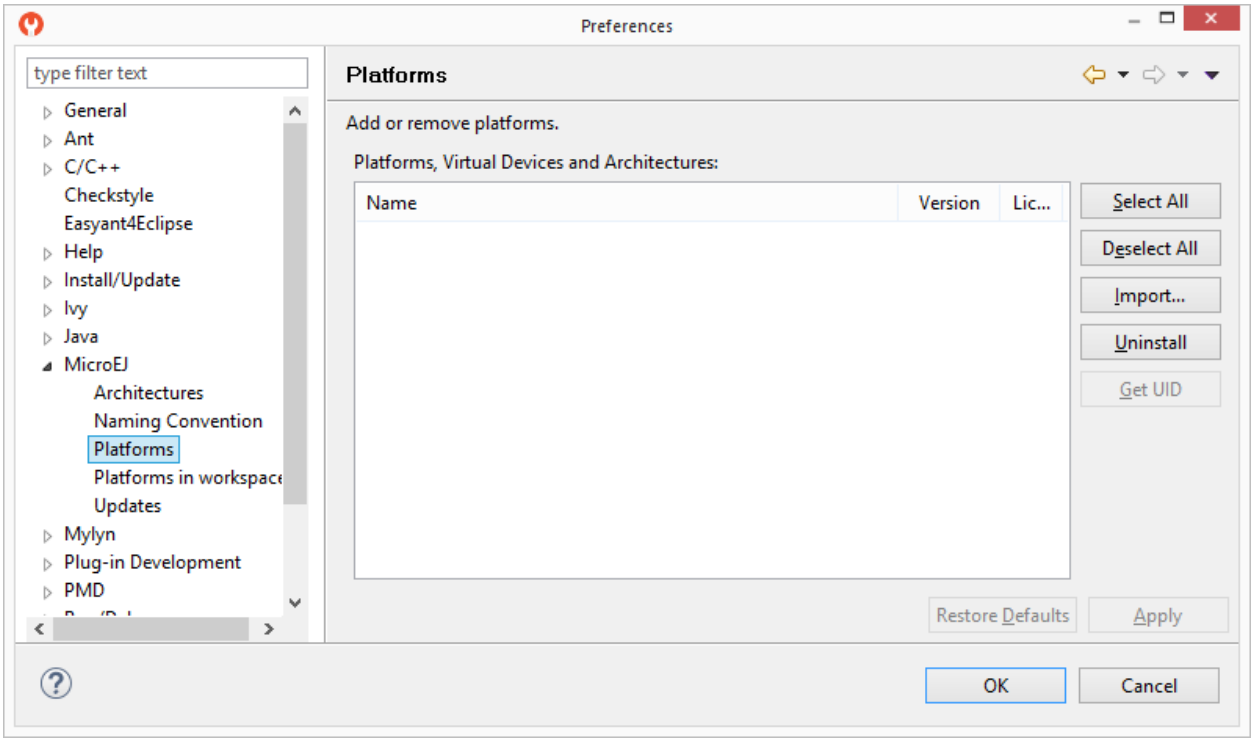


Fig. 333: Platform Import

- Press Import… button.

- Choose Select File… and use the Browse option to navigate to the .jpf file containing your Platform, then read and accept the license agreement to proceed.

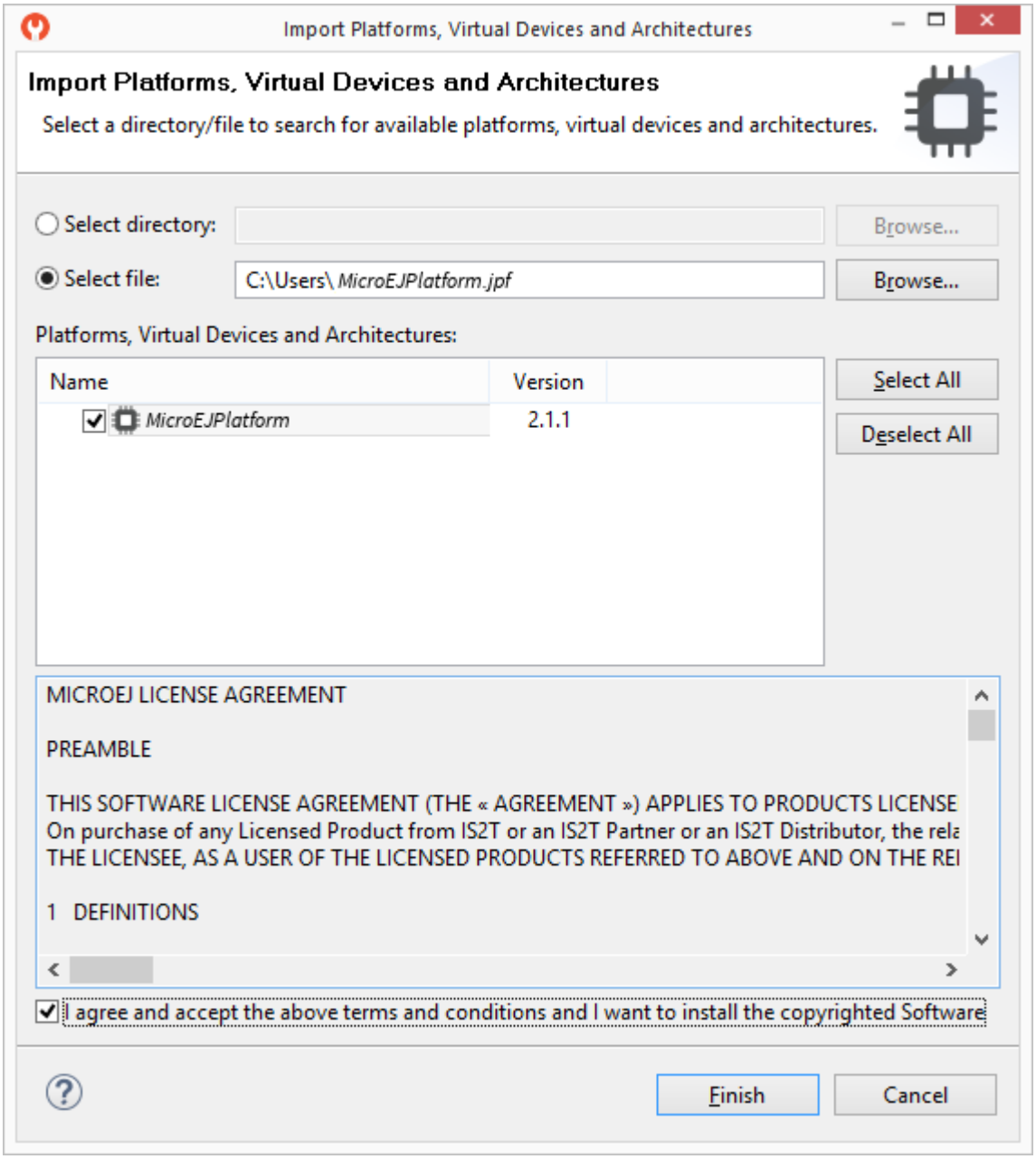


Fig. 334: Platform Selection

- The Platform should now appear in the Platforms view, with a green valid mark.

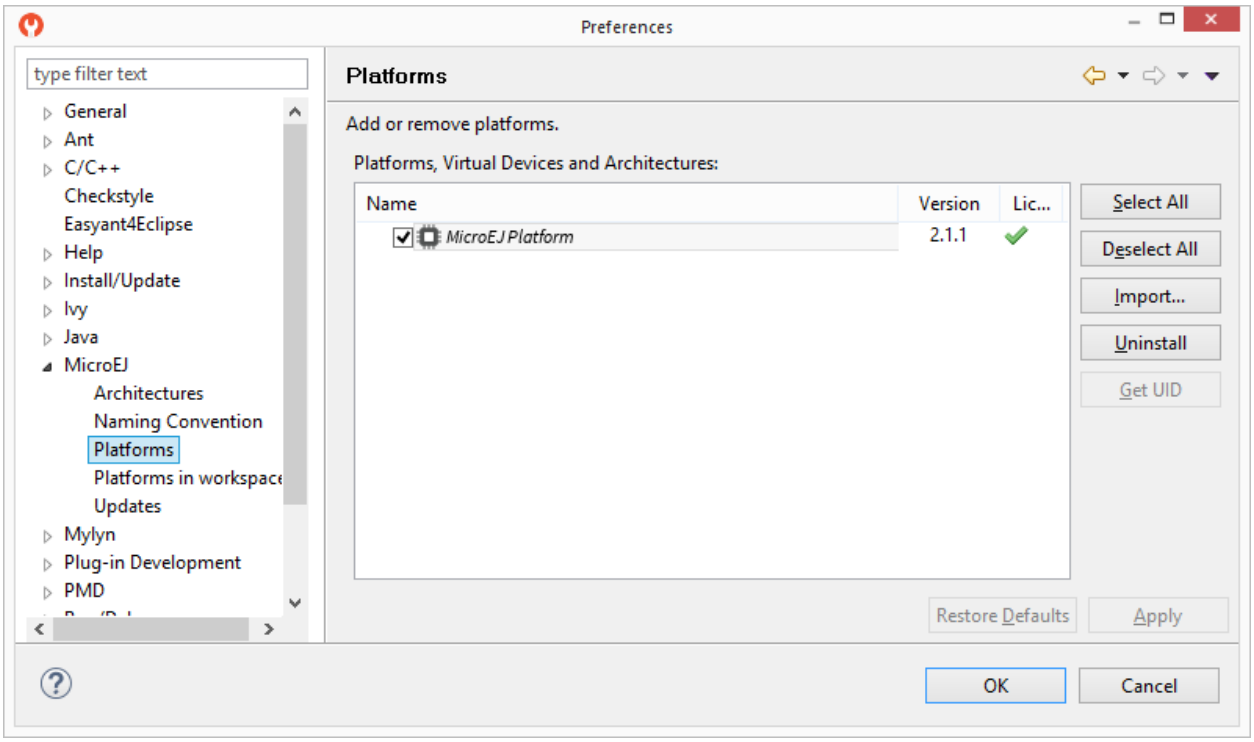


Fig. 335: Platform List

Build and Run an Application

Create a MicroEJ Standalone Application

Note: This section is releated to the version 5 and lower of the SDK. If you use the SDK 6, please refer to the page *Create a Project*.

- Create a project in your workspace. Select File > New > Standalone Application Project .

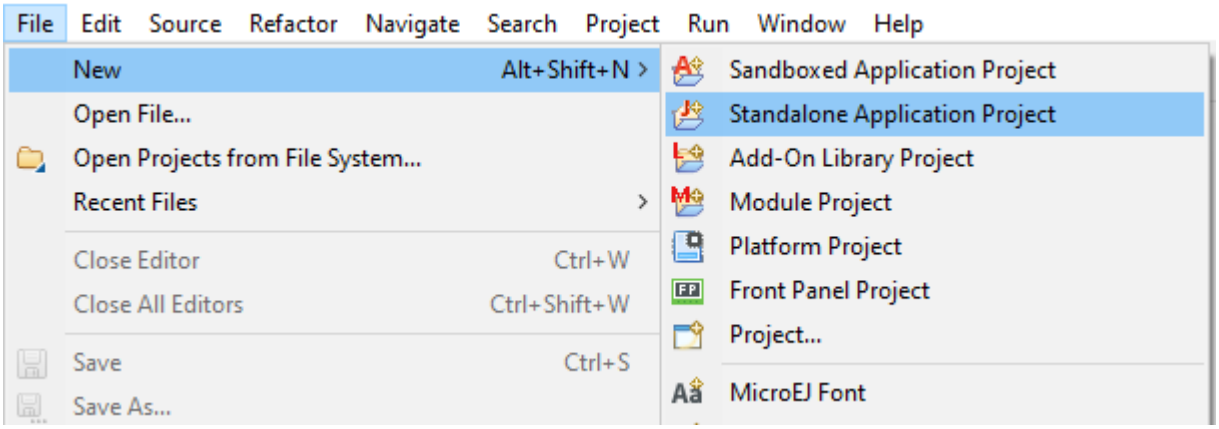


Fig. 336: New MicroEJ Standalone Application Project

- Fill in the Application template fields, the project name field will automatically duplicate in the following fields. For this tutorial, the project name is hello . Click on Finish . A template project is automatically created and ready to use, this project already contains all folders wherein developers need to put content:

– src/main/java : Folder for future sources

– src/main/resources : Folder for future resources (raw resources, images, fonts, nls)

– module.ivy : *Module description file*, dependencies description for the current project

- A Main class already exists in the package com.mycompany and prints “Hello World!” :

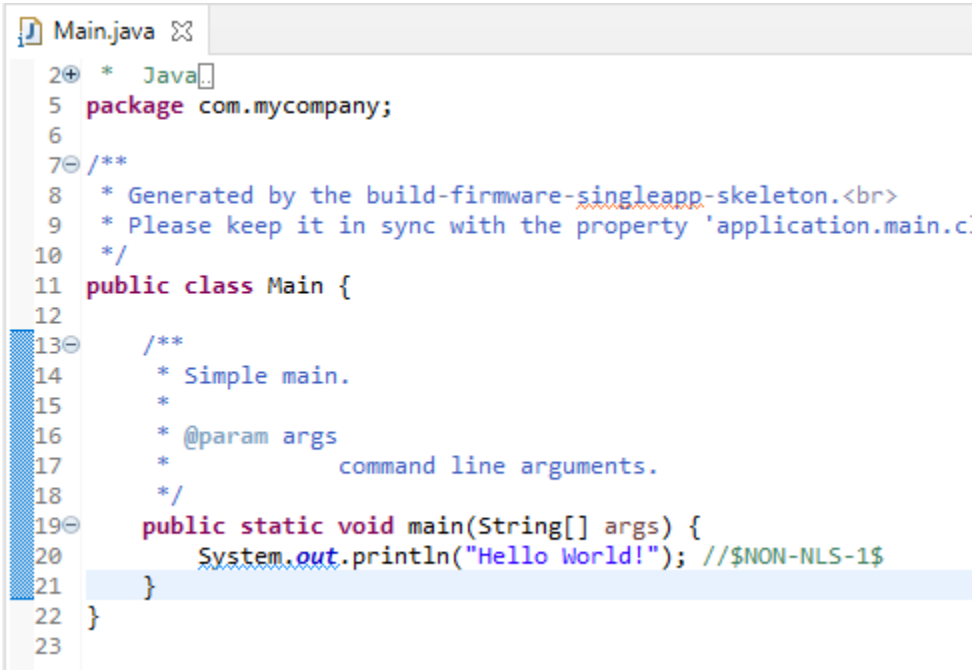


Fig. 337: MicroEJ Application Content

The main Application is now ready to be executed. See next sections.

Run on the Simulator

Note: This section is releated to the version 5 and lower of the SDK. If you use the SDK 6, please refer to the page *Run on Simulator*.

Note: *A Platform must have been imported* to run the Application. If several Platforms have been imported, the target Platform can be selected in the *Launcher's Execution tab*.

To run the sample project on Simulator, select it in the left panel then right-click and select Run > Run as > MicroEJ Application .

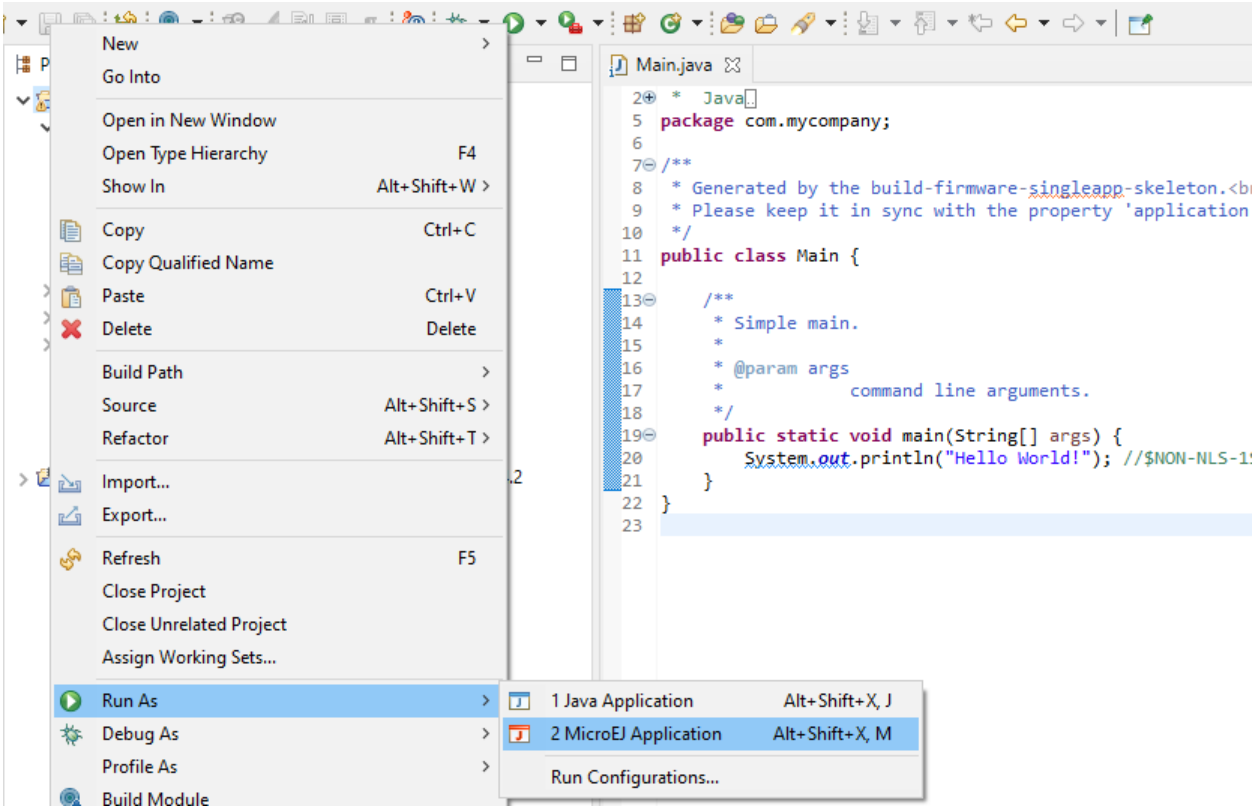


Fig. 338: MicroEJ Launcher Shortcut

MicroEJ SDK console will display Launch steps messages.

```
================ [ Initialization Stage ] ================
================ [ Launching on Simulator ] ================
Hello World!
================ [ Completed Successfully ] ================

SUCCESS
```

Run on the Device

Build the Application

- Open the run dialog (`Run` > `Run Configurations...`).
- Select the `MicroEJ Application` > `Hello Main` that is created by the previous chapter.
- Open `Execution` tab and select `Execute on Device` .
- Set `Settings` checkbox to `Build & Deploy` .

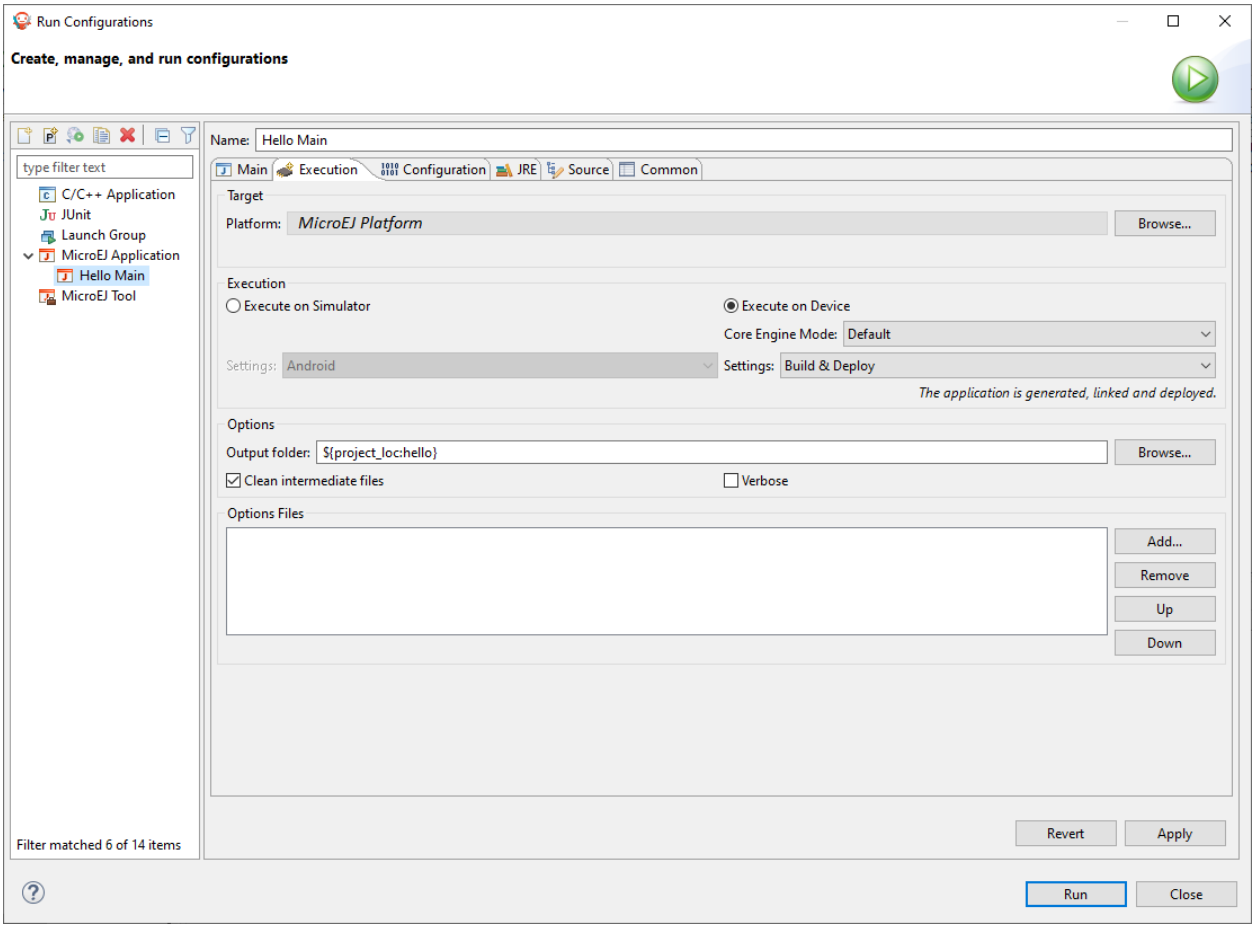


Fig. 339: Execution on Device

- Click Run : The Application is compiled and the Application, the runtime library and the header files are automatically deployed to the locations defined in your Platform *BSP connection* settings.

```
================ [ Deployment ] ================
MicroEJ files for the 3rd-party BSP project␣
↪are generated to '<application-project>/<fully-qualified-name-of-main-class>/platform'.
The␣
↪MicroEJ application (microejapp.o) has been deployed to: '<path-to-deployment-location>'.
The MicroEJ␣
↪platform library (microejruntime.a) has been deployed to: '<path-to-deployment-location>'.
The␣
↪MicroEJ platform header files (*.h) have been deployed to: '<path-to-deployment-location>'.
================ [ Completed Successfully ] ================

SUCCESS
```

Build the Executable File

If your Platform has configured a *build script* file, the final Application linking can be triggered from the launcher:

- Open `Configuration` tab and select `Device` > `Deploy` . The options to deploy the Application, runtime library and header files have already been set in the previous step.
- Check `Execute the MicroEJ build script (build.bat) at a location known by the 3rd-party BSP project` .

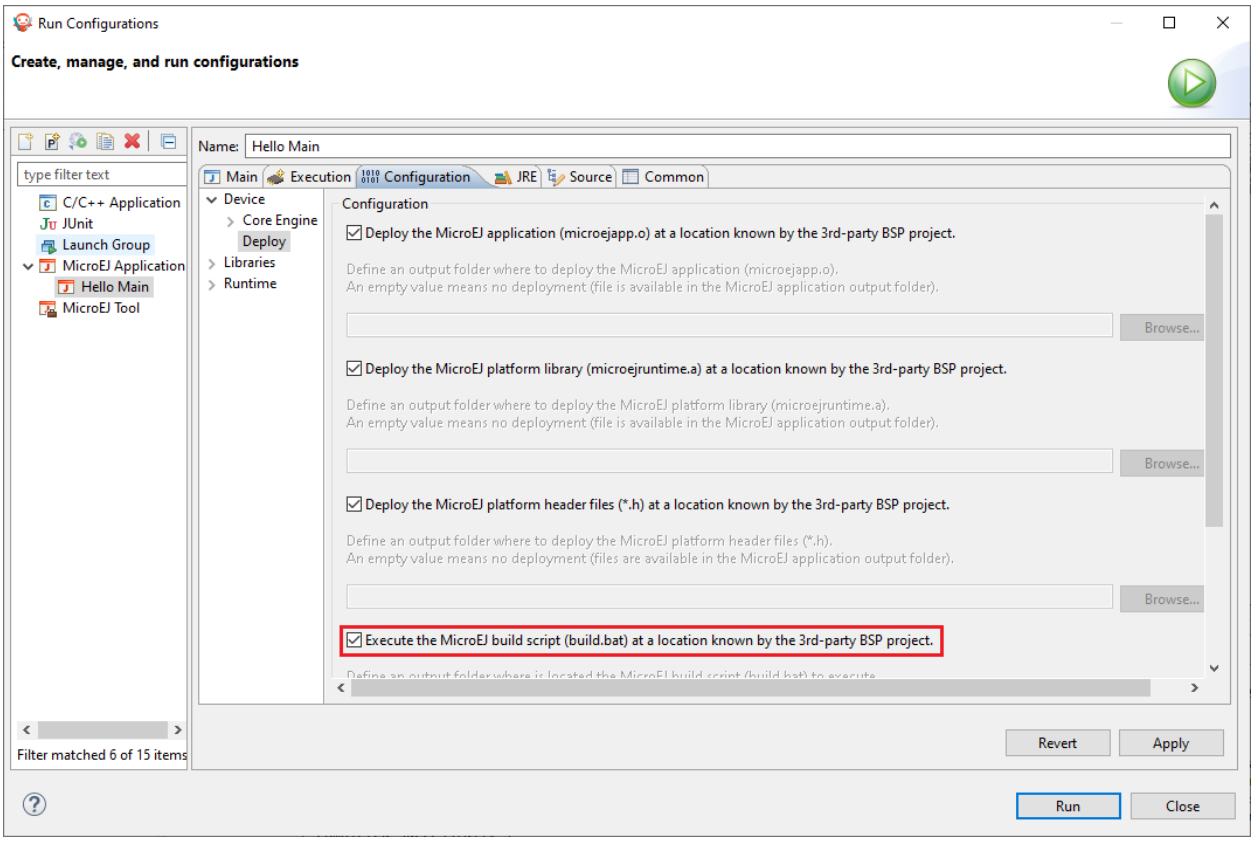


Fig. 340: BSP Connection Application Options

Note: The table *MicroEJ Application Options for BSP Connection* specifies the Application options that can be set depending on the BSP connection configured by the Platform.

- Click `Apply` and `Run` : the final executable `application.out` file is generated in the directory from where the script has been executed and can now be deployed on your Device using the appropriate flash tool.

MicroEJ Launch

The MicroEJ launch configuration sets up the Applications environment (main class, target, and Application options), and then launches a script for execution.

Execution is done either on the Simulator or on the Device. In this latter case, it may depend on external tools such as target memory programming.

Main Tab

The Main tab allows you to set in order:

1. The main project of the application.
2. The main class of the application containing the main method.
3. Types required in your application that are not statically embedded from the main class entry point. Most required types are those that may be loaded dynamically by the application, using the Class.forName() method.
4. Binary resources that need to be embedded by the application. These are usually loaded by the application using the Class.getResourceAsStream() method.
5. Immutable objects' description files. See the *[BON 1.2] ESR documentation* for use of immutable objects.

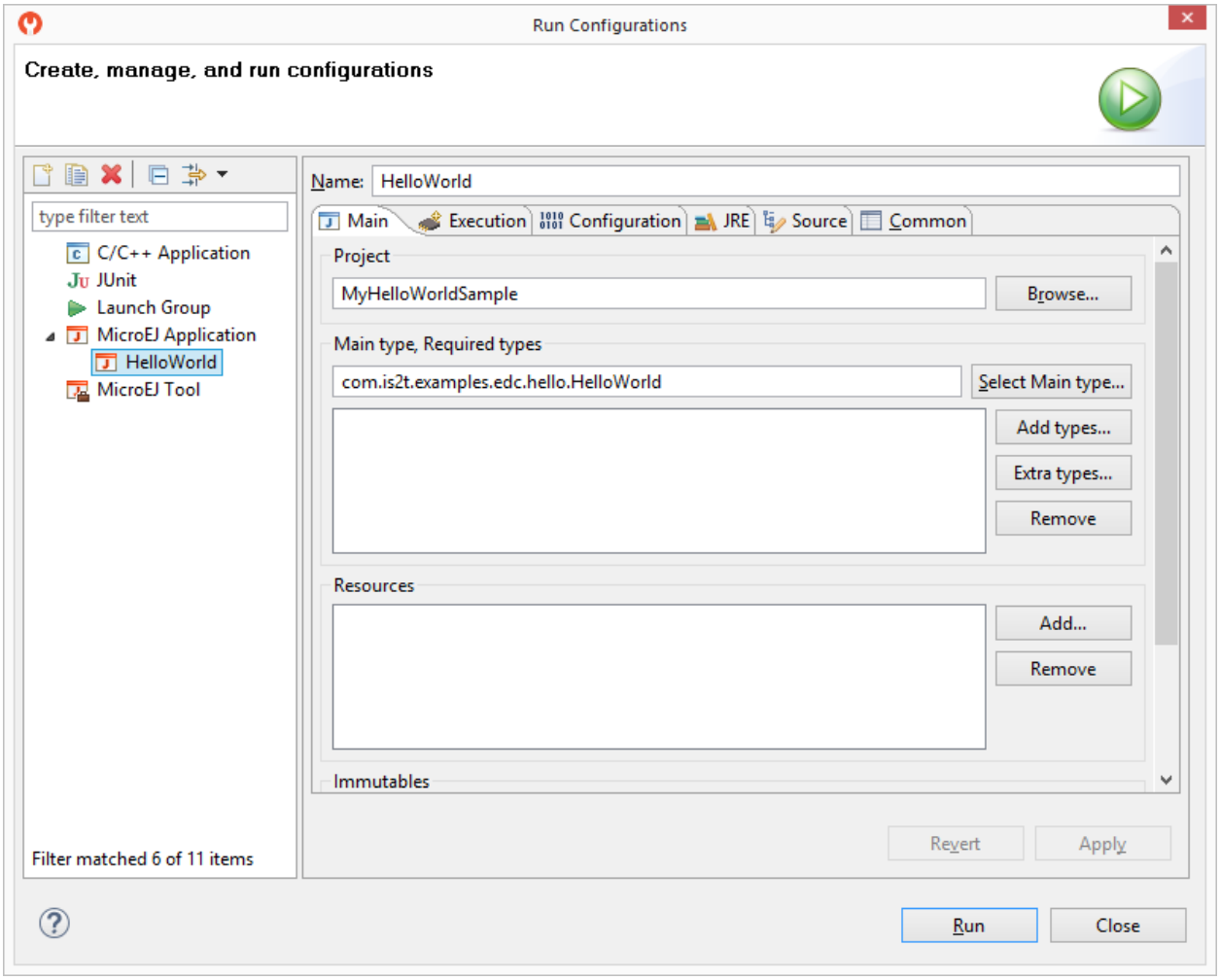


Fig. 341: MicroEJ Launch Application Main Tab

Execution Tab

The next tab is the Execution tab. Here the target needs to be selected. Choose between execution on a MicroEJ Platform or on a MicroEJ Simulator. Each of them may provide multiple launch settings. This page also allows you to keep generated, intermediate files and to print verbose options (advanced debug purpose options).

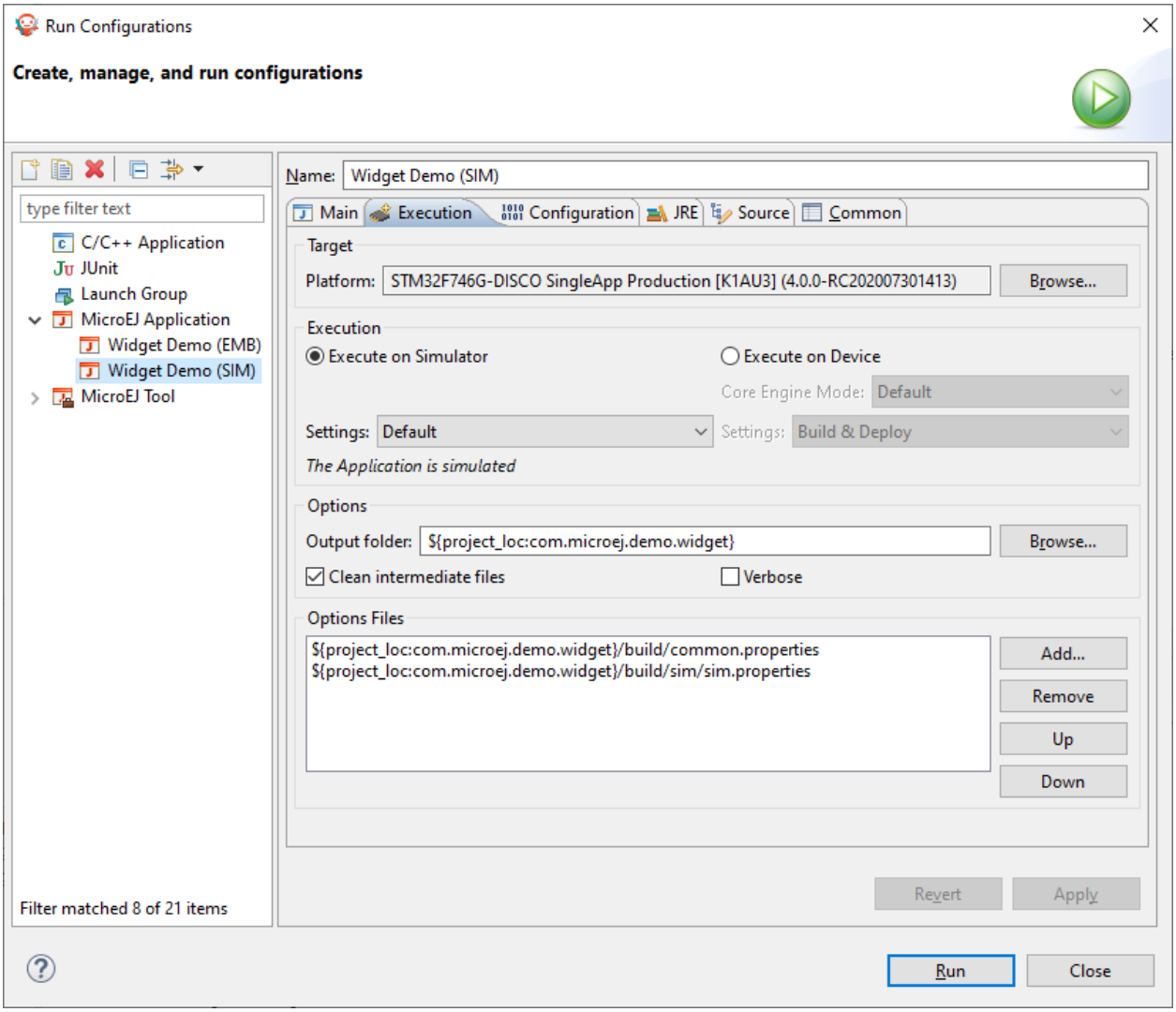


Fig. 342: MicroEJ Launch Application Execution Tab

Configuration Tab

The next tab is the Configuration tab. This tab shows the available *Application options*.

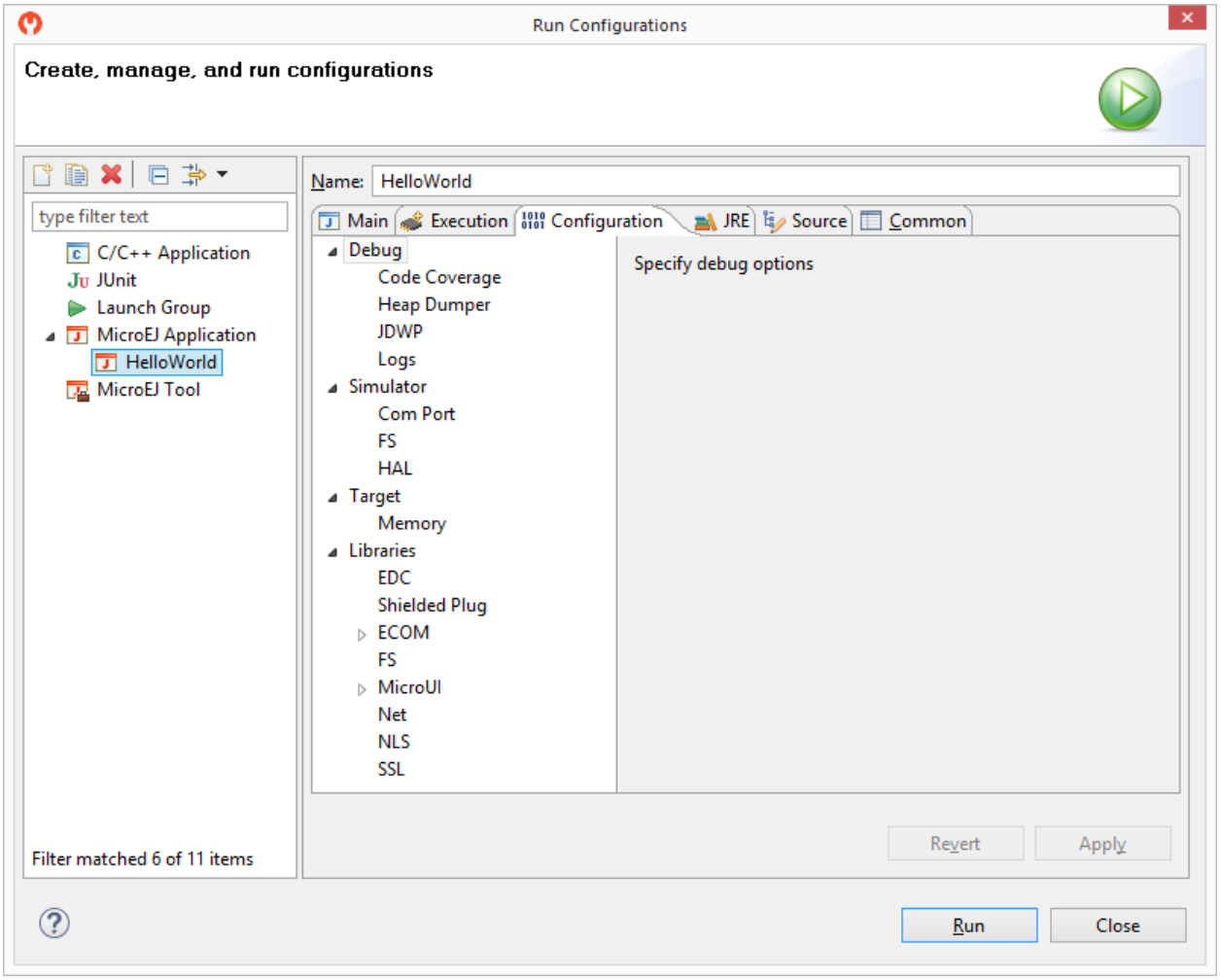
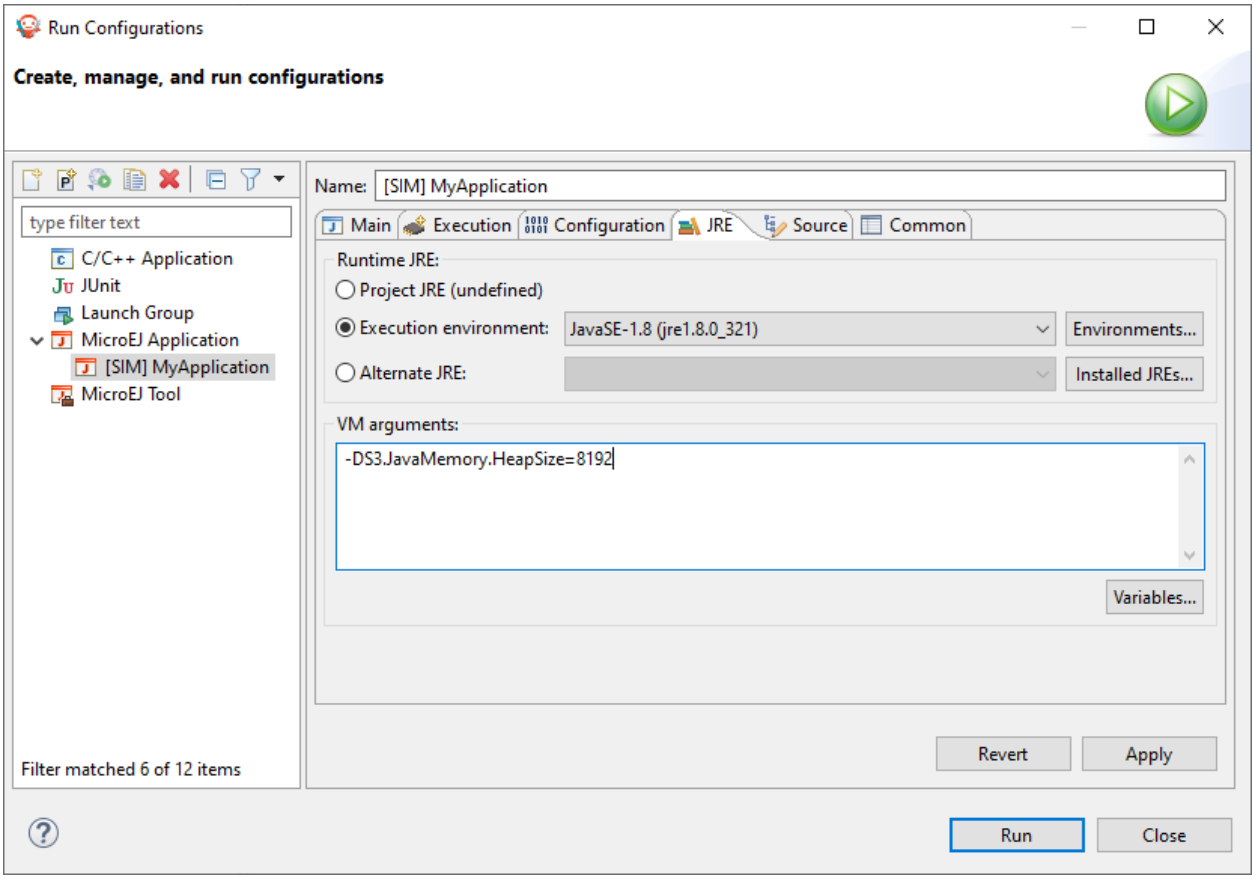


Fig. 343: Configuration Tab

JRE Tab

The next tab is the JRE tab. This tab allows you to configure the Java Runtime Environment used for running the underlying launch script. It does not configure the MicroEJ Application execution. The VM Arguments text field allows you to set vm-specific options, which are typically used to increase memory spaces:

- To modify heap space to 1024MB, set the -Xmx1024M option.
- To modify string space (also called PermGen space) to 256MB, set the -XX:PermSize=256M -XX:MaxPermSize=256M options.
- To set thread stack space to 512MB, set the -Xss512M option.
- To set an *advanced Application option*, declare a system property with the following pattern -D[OPTION_KEY]>=[OPTION_VALUE]



Source Tab

The next tab is the `Source` tab. By default, it is automatically configured to connect your Add-On Libraries sources dependencies. To connect your Platform Foundation Library sources, please refer to the section *Foundation Library Sources*.

Common Tab

The last tab is the `Common` tab. This is a default Eclipse tab that allows to configure your launch. Particularly, you can configure the *console encoding*. Refer to Eclipse help for more details on other available options.

A Standalone Application is the main Application that is directly linked to the C code to produce an Executable. Such an application must define a main entry point (i.e., a class containing a `public static void main(String[])` method).

The *next chapters* explain how to build and run a Standalone Application.

3.11.4 Sandboxed Application

Create a First Application

Now that the *purposes of the Sandboxed Applications have been explained*, let's create a first application.

A Sandboxed Application project can be created in the SDK with the menu `File` > `New` > `Sandboxed Application Project` .

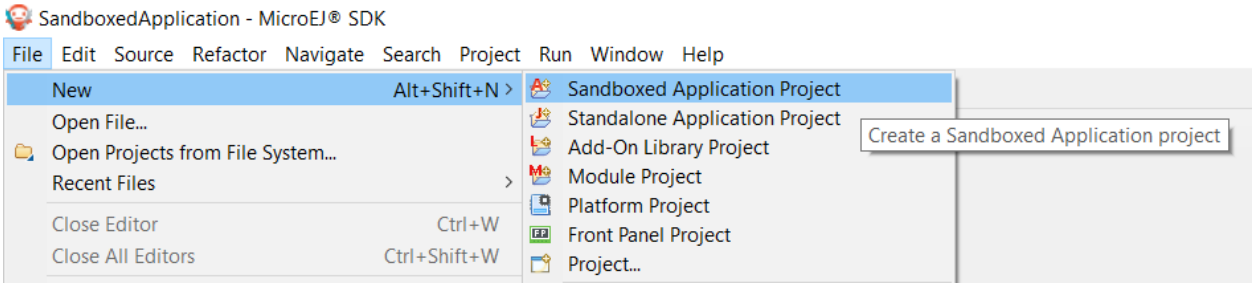


Fig. 344: Sandboxed Application Project Creation Menu

The project creation window is displayed:

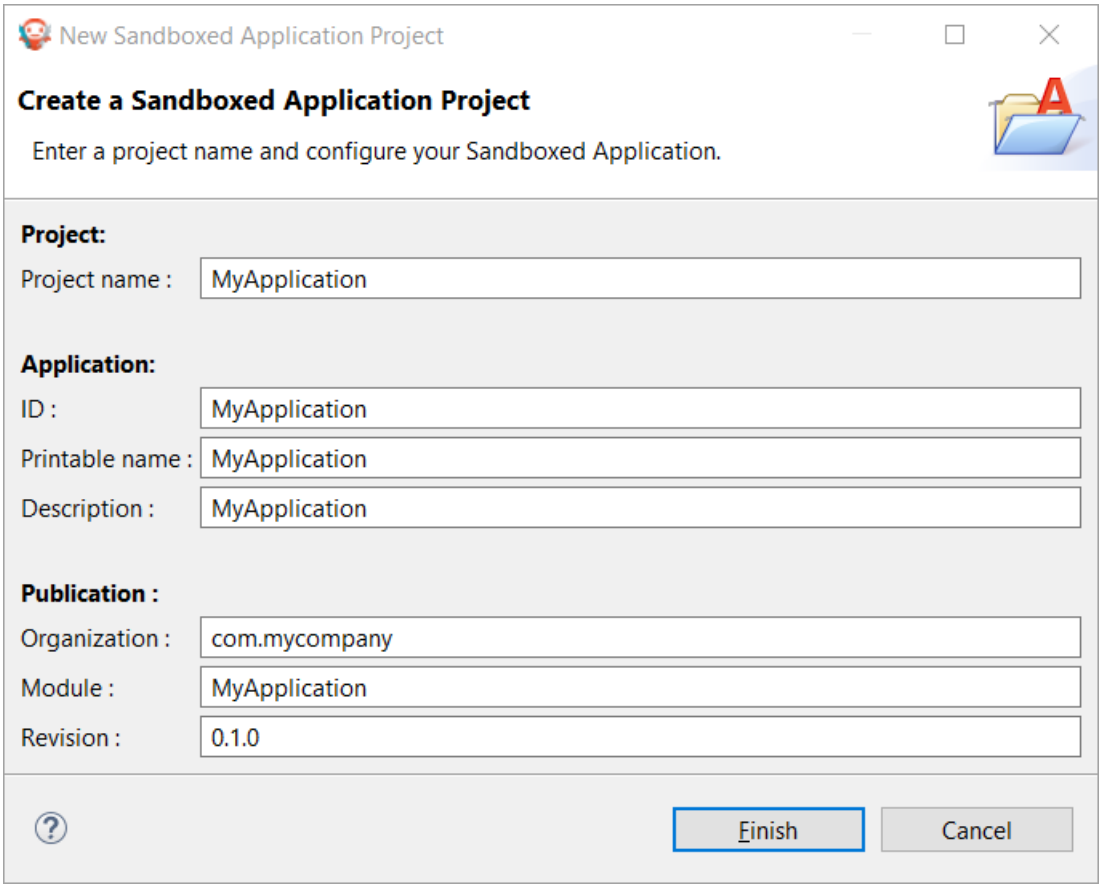


Fig. 345: Sandboxed Application Project Creation Form

Once the Application information are fulfilled and validated, the project is created with the following structure:

`src/main/java`
: Application Java sources;

`src/main/resources`
: Application resources (raw resources, images, fonts, nls);

`module.ivy`
: *Module description file*, containing build information and dependencies of the project.

The next sections describe the required files to have your first basic Application.

Entry Point

A Sandboxed Application must contain a class implementing the `ej.kf.FeatureEntryPoint` interface in the `src/main/java` folder:

```
package com.mycompany;

import ej.kf.FeatureEntryPoint;
```

(continues on next page)

(continued from previous page)

```
public class MyApplication implements FeatureEntryPoint {

    @Override
    public void start() {
        System.out.println("Feature MyApplication started!");
    }

    @Override
    public void stop() {
        System.out.println("Feature MyApplication stopped!");
    }
}
```

This class is the entry point of the Application. The method `start` is called when the Application is started. It is considered as the main method of the Sandboxed Application. The method `stop` is called when the Application is stopped. Please refer to the *Sandboxed Application Lifecycle* chapter to learn more about the Applications lifecycle.

The `src/main/java` folder is also the place to add all the other Java classes of the Application.

Configuration

A Sandboxed Application project must contain a file with the `.kf` extension in the `src/main/resources` folder. This file contains the configuration of the Application. Here is an example:

```
name=MyApplication
entryPoint=com.mycompany.MyApplication
types=*
version=0.1.0
```

It contains the following properties:

- **name**: the name of the Application
- **entryPoint**: the Full Qualified Name of the class implementing `ej.kf.FeatureEntryPoint`
- **types**: this property defined the types included in the Application and must always be `*` (do not forget the space at the end)
- **version**: the version of the Application

SSL Certificate

A Sandboxed Application requires a certificate for identification. It must be located in the `src/main/resources` folder of the project. The project created by the SDK provides a sample certificate. This certificate is sufficient for testing, but it is recommended to provide your own.

Module Descriptor

The `module.ivy` file is the *Module description file* which contains the project information and declares all the libraries required by the Application. See *MicroEJ Module Manager* for more information.

The dependencies must contain at least a module containing the `ej.kf.FeatureEntryPoint` class, for example the KF library:

SDK 6

SDK 5

```
implementation("ej.api:kf:1.7.0")
```

```
<dependency org="ej.api" name="kf" rev="1.7.0" />
```

Run on the Simulator

Note: This page is releated to the version 5 and lower of the SDK. If you use the SDK 6, please refer to the page *Run on Simulator*.

Once *a Sandboxed Application project has been created*, it can be tested on the Simulator.

The Simulator requires a Virtual Device to execute the Application. Please refer to the *Kernel Developer Guide* to learn how to get or create one.

From the SDK

In order to test a Sandboxed Application in the SDK, the first thing to do is to import the Virtual Device of the Multi-Sandbox Executable:

- go to Window > Preferences > MicroEJ > Virtual Devices
- click on Import…
- the Virtual Device can be provided as a folder or as a `.vde` file, select the adequate format and the Virtual Device resource
- check the License checkbox to accept it
- click on Finish

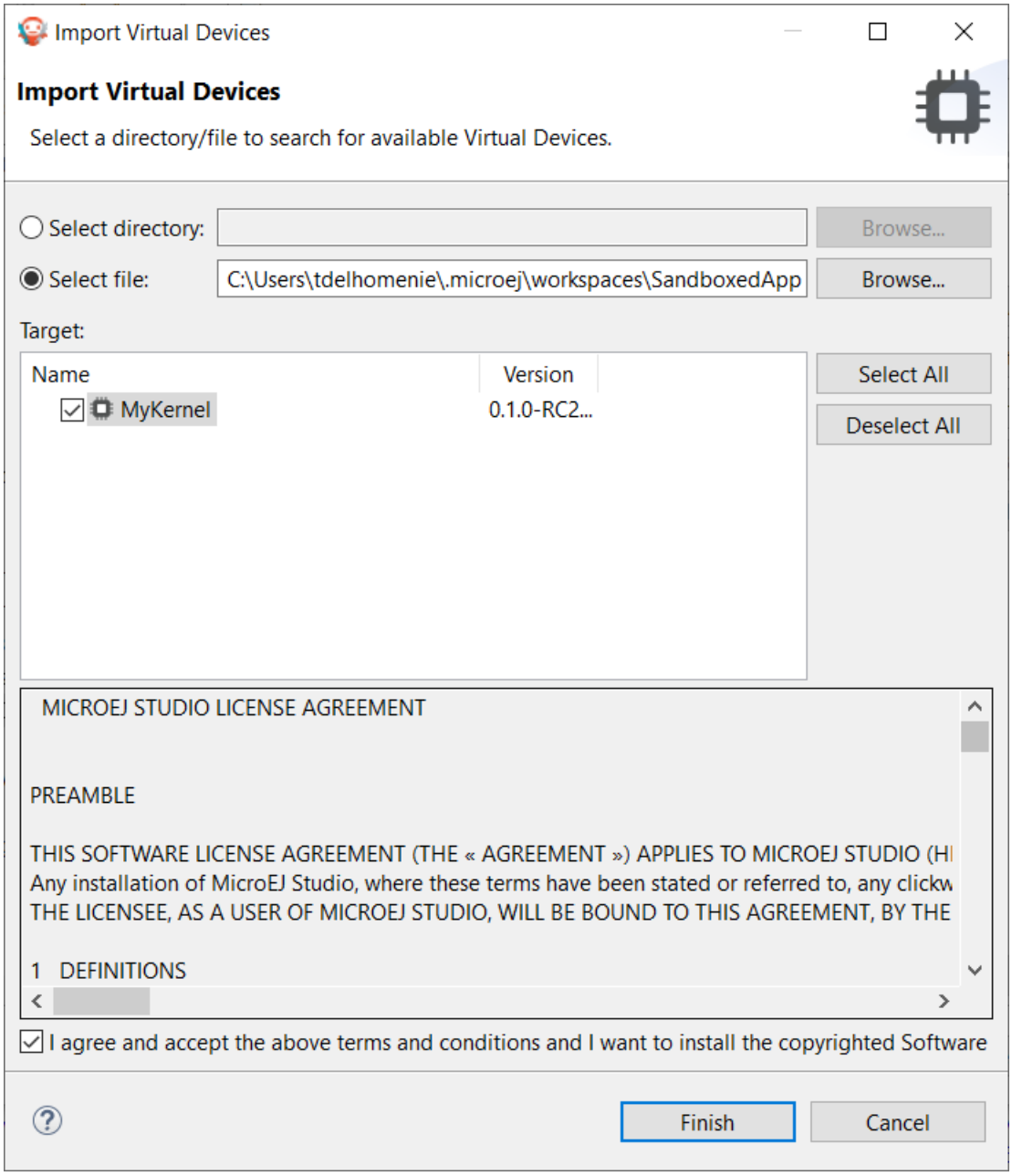


Fig. 346: Virtual Device Import

Now the Application can be executed by right-clicking on its project, then clicking on `Run As` > `MicroEJ Application` .

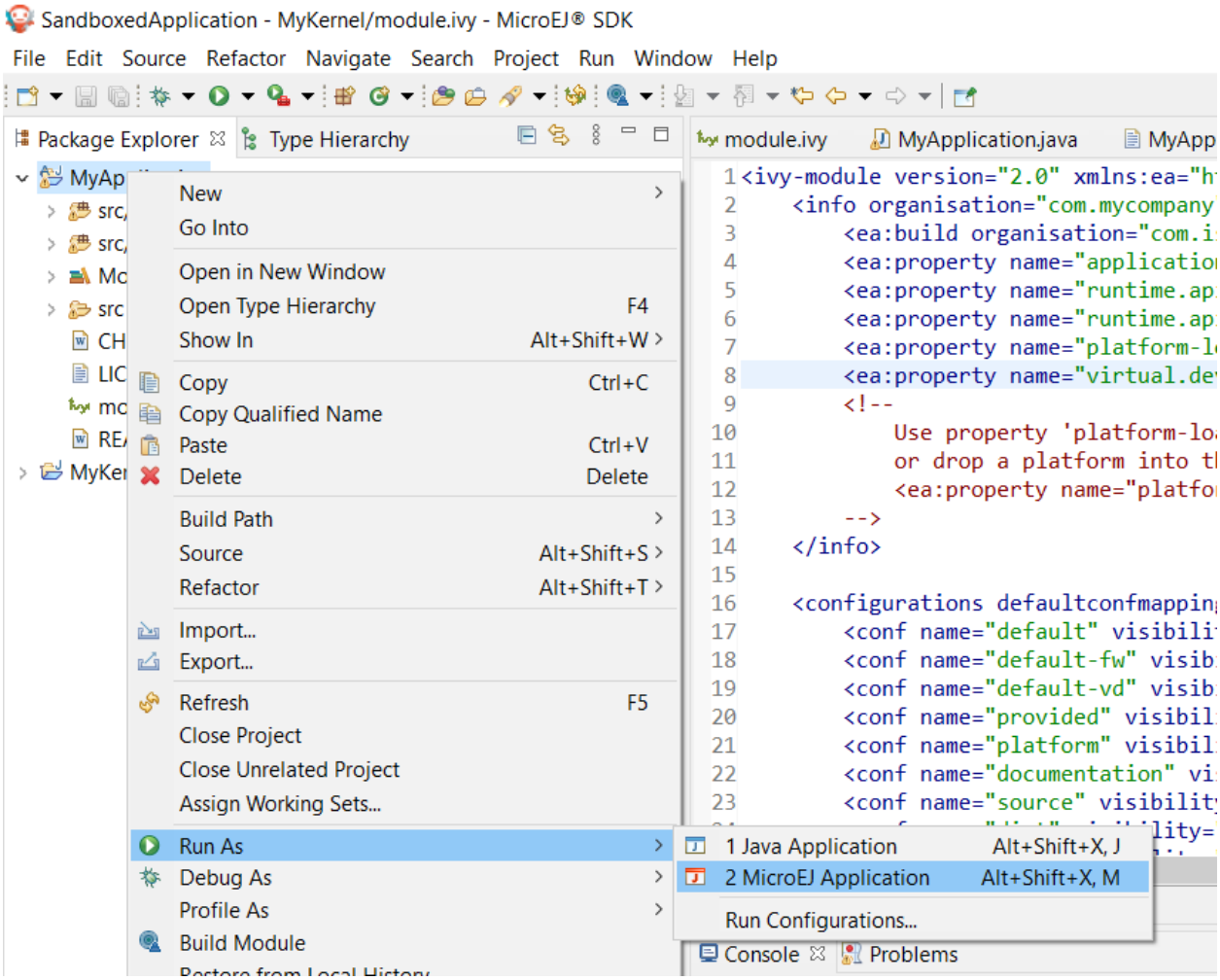


Fig. 347: Sandboxed Application Run

If there is only one Virtual Device imported in the SDK, it is automatically used to execute the Application. Otherwise, you have to select the one you want to use.

With the Application created in the section *Create a First Application*, the output should be:

```
================ [ Initialization Stage ] ================
================ [ Converting fonts ] ================
================ [ Converting images ] ================
================ [ Launching on Simulator ] ================
KERNEL Hello World!
=> Starting Feature MyApplication
Feature MyApplication started!
================ [ Completed Successfully ] ================

SUCCESS
```

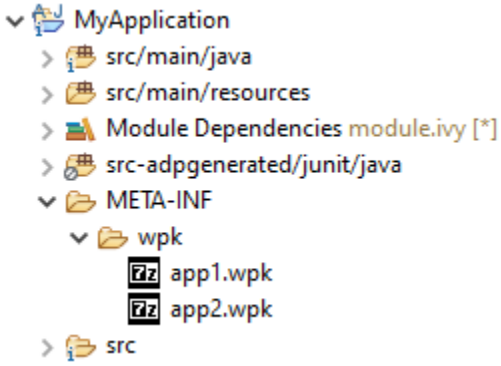
Run Multiple Sandboxed Applications

It is possible to execute additional Sandboxed Applications besides the main Sandboxed Application project. This is typically useful when you want to test the integration of a Sandboxed Application that communicates with an other one, for example through a *Shared Interface*.

The additional Sandboxed Applications must have been previously built in its binary format (WPK, see *Remote Deployment* section). Then, to include them:

- Select the Sandboxed Application project,
- Create the `META-INF/wpk` folders,
- Drop any `*.wpk` files in the `META-INF/wpk` folder.

Your Sandboxed Application project shall look like:



Now, when launching the Sandboxed Application project, these additional Sandboxed Applications will also be executed on the Virtual Device.

From the Command Line Interface

An Sandboxed Application can also be launched on the Simulator via the Command Line Interface. Before continuing, make sure *the Command Line Interface is installed and correctly configured*.

In your favorite terminal application, go to the root folder of the Application and execute the following commands:

```
mmm build
mmm run -Dplatform-loader.target.platform.file=/path/to/the/virtual-device.vde
```

With the Application created in the section *Create a First Application*, the output should be:

```
MicroEJ Simulator is being launched. Relax and enjoy...
================ [ Initializing Easyant ] ================
================ [ Resolving and retrieving dependencies ] ================
================ [ Compiling sources ] ================
================ [ Loading platform ] ================
================ [ Initialization Stage ] ================
================ [ Converting fonts ] ================
```

(continues on next page)

(continued from previous page)

```
================ [ Converting images ] ================
================ [ Launching on Simulator ] ================
KERNEL Hello World!
=> Starting Feature MyApplication
Feature MyApplication started!
================ [ Completed Successfully ] ================

SUCCESS
```

Note that the Virtual Device location can also be configured in the `module.ivy` file of the Sandboxed Application project:

```
<ea:property␣
↪name="platform-loader.target.platform.file" value="/path/to/the/virtual-device.vde"/>
```

The Virtual Device can also be provided differently, for example from a dependency in the `module.ivy` file. Refer to the *Select a VEE Port* section for the list of available capabilities.

Run on the Device

The deployment of a Sandboxed Application on a device depends on the Kernel implementation. We can group them in two categories:

- Local Deployment: the device is connected to the developer's computer, the SDK builds the `.fo` from the workspace project classes and transfers it on the device (recommended during application development).
- Remote Deployment: the Application is built, then the device connects a Repository where the Application is stored, and deploys it over the air using a device management system (production deployment).

  In both cases, deploying a Sandboxed Application requires that a Multi-Sandbox Executable is running on the device. Please refer to the *Kernel Developer Guide* to learn how to build it or browse the Resources Repository for Multi-Sandbox demo Firmware available for popular hardware evaluation kits.

Local Deployment

Deploying an Application on a device locally is the easiest way to test it since it only requires:

- the Application project sources imported in the SDK,
- the device programmed with a Multi-Sandbox Executable that provides the Local Deployment capability (you can browse the Resources Repository for available demos of such Multi-Sandbox Executable),
- the device connected to the developer's computer either on the same network (LAN) or using a serial wire, depending on the Firmware capabilities.

  If these prerequisites are fulfilled:

- duplicate the Run Configuration created in the chapter *Run on the Simulator*,

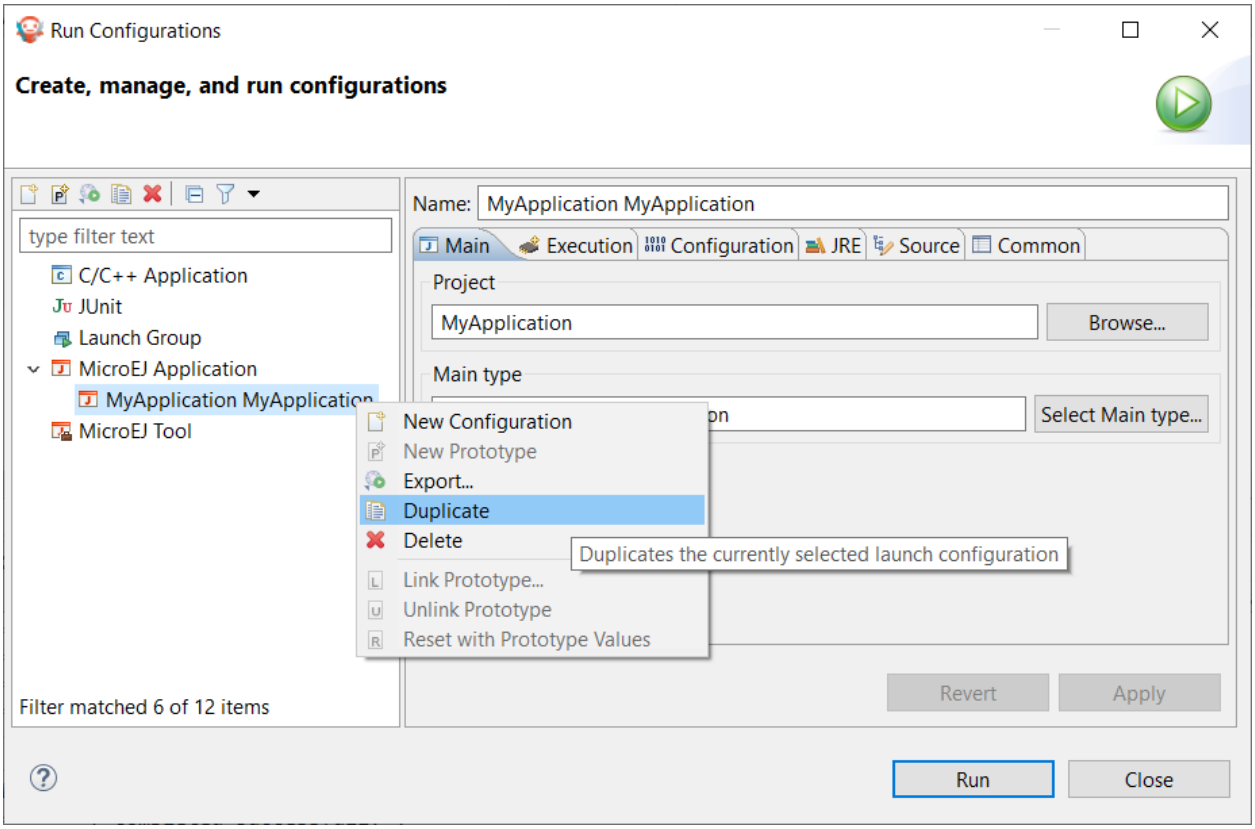


Fig. 348: Duplicate Run Configuration

- rename the duplicated Run Configuration, for example by prefixing by (Local),
- in the Execution tab, modify the Execution mode to Execute on Device,

Note: The selected Platform must be a Virtual Device (VDE) including the Local Deployment capability, not a VEE Port.

- select the option Local Deployment (Socket) in the Settings list. Note that depending on the device capability, the virtual device may implement a local deployment over a Comm Port.

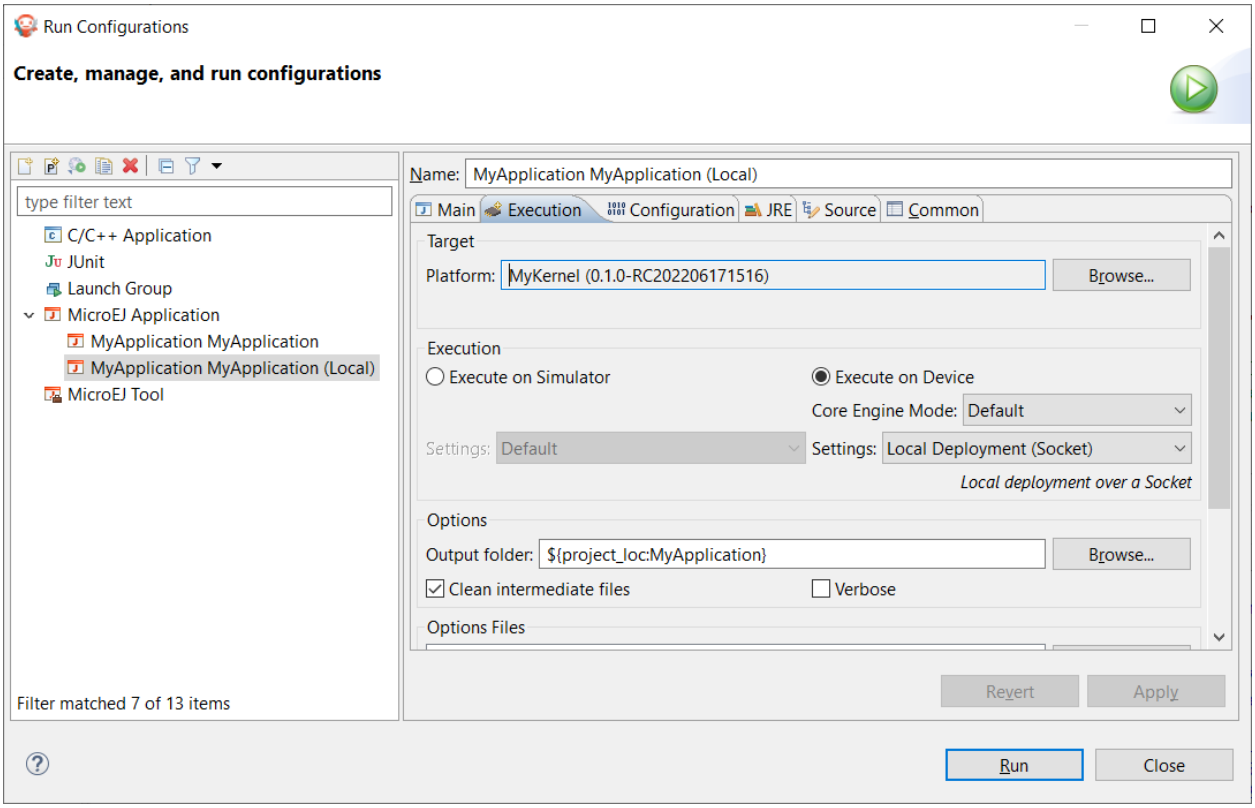


Fig. 349: Configure Run Configuration

- go to the `Configuration` tab,
- select the item `Local Deployment (Socket)`,
- set the IP address of the device in the `Host` field,
- click on the `Run` button to deploy the Application on the board.

  The Console output should be:

```
================ [ Initialization Stage ] ================
================ [ Converting fonts ] ================
================ [ Converting images ] ================
================ [ Build Application ] ================
================ [ Completed Successfully ] ================
================ [ Deploy on 192.168.0.7:4000 ] ================
================ [ Completed Successfully ] ================

SUCCESS
```

The Application is deployed on the device and automatically started. You can use a Serial terminal to get the traces of the Application:

```
KERNEL Hello World!
=> Starting Feature MyApplication
Feature MyApplication started!
```

Remote Deployment

Remote Deployment requires building and publishing the Sandboxed Application module. To do so, in the SDK, right-click on the Sandboxed Application project and click on Build Module .

The build process will display messages in the console, ending up the following message:

```
[echo] project hello published locally with version 0.1.0-RC201907091602

BUILD SUCCESSFUL

Total time: 1 minute 6 seconds
```

The files produced by the build process are located in a dedicated target~/artifacts folder in the project and is published to the target module repository declared in *MicroEJ Module Manager settings file*.

The file that ends with .wpk (the WPK file) is a portable file that contains all necessary binary data to build .fo files on any compatible Multi-Sandbox Executable. Then, the WPK file can be published to a MICROEJ FORGE instance. Please contact *our support team* if you want to get more information on MICROEJ FORGE and automated Applications deployment through a device management system.

A Sandboxed Application is an Application that can run over a Multi-Sandbox Executable.

The Application development flow requires the following elements:

- a Virtual Device, a software package including the resources and tools required for building and testing an application for a specific device. A Virtual Device will simulate all capabilities of the corresponding hardware board:
  - Computation and Memory
  - Communication channels (e.g., Network, USB …)
  - Display
  - User interaction
- an hardware device that has been previously programmed with a Multi-Sandbox Executable. Virtual Devices and Multi-Sandbox Executable share the same version (there is a 1:1 mapping).

The *next chapters* explain how to create, test and publish Sandboxed Applications.

3.11.5 Module Repository

A module repository is a module that bundles a set of modules in a portable ZIP file. It is a tree structure where modules organizations and names are mapped to folders.

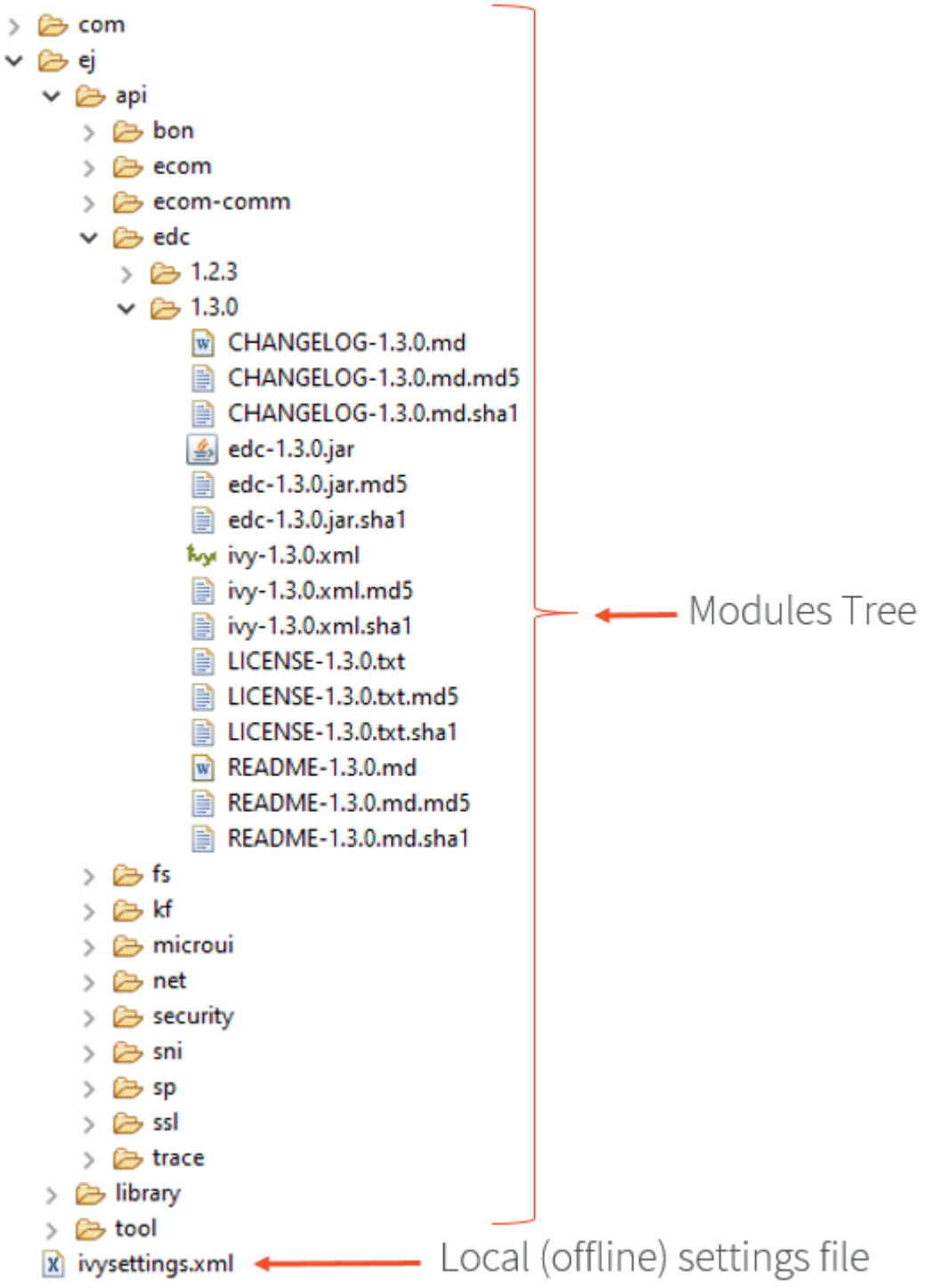


Fig. 350: Example of MicroEJ Module Repository Tree

A module repository takes its input modules from other repositories, usually the *MicroEJ Central Repository* which is itself built by MicroEJ Corp. as a module repository.

A module repository is often called an offline repository as it includes the settings file for a local configuration in MicroEJ SDK. It can also be imported in MicroEJ Forge.

Create a Repository Project

In the SDK, first create a new *module project* using the `artifact-repository` skeleton.

- The `ivysettings.xml` *settings file* describes how to import the modules of this repository when it is extracted locally on file system. This file will be packaged at the root of the zip file and does not need to be modified.
- The `module.ivy` file describes how to build repository and lists the module dependencies that will be included in this repository.

Configure Resolver for Input Modules

MicroEJ Module Manager (MMM) needs to import dependencies to build the module repository. The location fetched by MMM is defined by a resolver. The resolver is configured with the parameter `bar.populate.from.resolver`. The preset value is the resolver provided by default in MicroEJ SDK configuration, which is connected to *MicroEJ Central Repository*.

```
<ea:property name="bar.populate.from.resolver" value="MicroEJChainResolver"/>
```

The `MicroEJChainResolver` is a URL resolver defined in `<USER_HOME>\.microej\microej-ivysettings-[VERSION].xml` that points to MicroEJ Central Repository.

Configure Consistency Check

The module repository consistency check consists in verifying that each declared module can be imported using the settings file provided by the repository. Especially, it ensures that all module transitive dependencies are also available.

It is enabled by default to avoid further issues for repository users such as *Unresolved Dependency*. This is done by the following option:

```
<ea:property name="skip.retrieve.checker" value="false"/>
```

Moreover, to ensure the repository will be compliant with the *MMM specification*, add the following option:

```
<ea:property name="bar.check.as.v2.module" value="true"/>
```

Advanced Options

There are other advanced options that do not need to be modified by default. These options are described in the `module.ivy` generated by the skeleton.

See also *Module Repository* for more details.

Include Modules

Modules bundled into the module repository must be declared in the `dependencies` element of the `module.ivy` file.

Include a Single Module

To add a module, declare the *module dependency* using the `artifacts` configuration:

```
<dependencies>
  <dependency conf="artifacts->*"␣
↪transitive="false" org="[module_org]" name="[module_name]" rev="[module_version]" />

  <!-- ... other dependencies ... -->
</dependencies>
```

For example, to add the `ej.api.edc` library version `1.2.3`, write the following line:

```
<dependency conf="artifacts->*" transitive="false" org="ej.api" name="edc" rev="1.2.3" />
```

Note: We recommended to manually describe each dependency of the module repository, in order to keep full control of the included modules as well as included modules versions. Module dependencies can still be transitively included by setting the dependency attribute `transitive` to `true`. In this case, the included module versions are those that have been resolved when the module was built.

Multiple versions of the same module can be included by declaring each dependency using a different configuration. The `artifacts` configuration has to be derived with a new name as many times as there are different versions to include.

```
<configurations defaultconfmapping="default->default;provided->provided">
  <conf name="artifacts" visibility="private"/>
  <conf name="artifacts_1" visibility="private"/>
  <conf name="artifacts_2" visibility="private"/>

  <!-- ... other configurations ... -->
</configurations>

<dependencies>
  <dependency conf="artifacts->*" transitive=
↪"false" org="[module_org]" name="[module_name]" rev="[module_version_1]" />
  <dependency conf="artifacts_1->*" transitive=
↪"false" org="[module_org]" name="[module_name]" rev="[module_version_2]" />
  <dependency conf="artifacts_2->*" transitive=
↪"false" org="[module_org]" name="[module_name]" rev="[module_version_3]" />

  <!-- ... other dependencies ... -->
</dependencies>
```

Include a Module Repository

To add all the modules already included in an other module repository, add the configuration `repository` if it does not exist:

```
<configurations defaultconfmapping="default->default;provided->provided">
  <!-- ... other configurations ... -->
  <conf name="repository
↪" visibility="private" description="Repository to be embedded in the repository" />

</configurations>
```

Then declare the module repository dependency using the `repository` configuration:

```
<dependencies>
  <dependency conf="repository->*" transitive=
↪"false" org="[repository_org]" name="[repository_name]" rev="[repository_version]" />

  <!-- ... other dependencies ... -->
</dependencies>
```

Generate Javadoc

An overall Javadoc can be generated beside the included modules. It is built from of all Java elements of all libraries included in the module repository.

Javadoc generation is disabled in the `module.ivy` generated by the skeleton. To enable javadoc generation, remove `skip.javadoc` option or set it to `false`.

There are also javadoc specific options such as Java packages exclusion. Please refer to `*javadoc*` options of *Module Repository* reference documentation.

As of *SDK 5.3.0*, the *module dependency* line that defines a Java type is shown in the top menu.

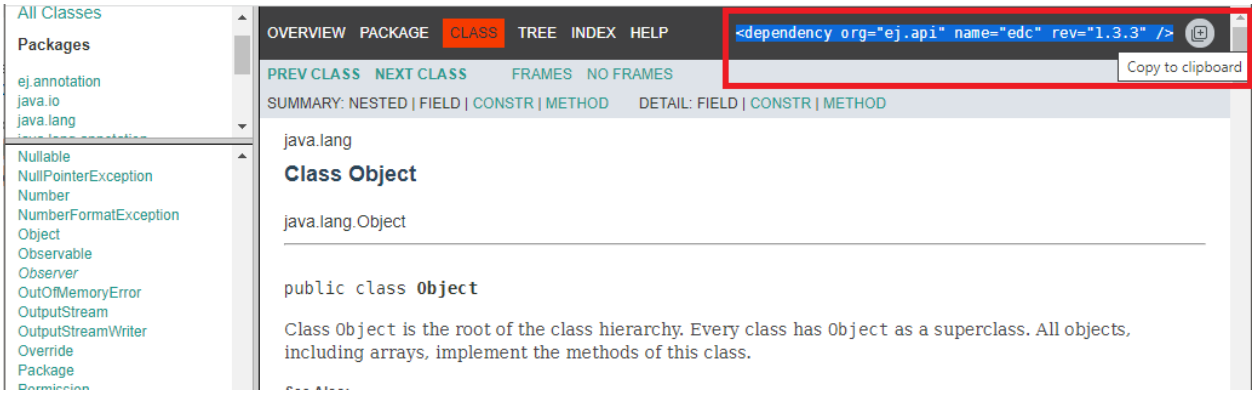


Fig. 351: Example of Javadoc Module Dependency

Build the Repository

In the Package Explorer, right-click on the repository project and select `Build Module`.

The build consists of two steps:

1. Gathers all module dependencies. The whole repository content is created under `target~/mergedArtifactsRepository` folder.
2. Checks the repository consistency. For each module, it tries to import it from this repository and fails the build if at least one of the dependencies cannot be resolved.

The module repository `.zip` file is built in the `target~/artifacts/` folder. This file may be published along with a `CHANGELOG.md`, `LICENSE.txt` and `README.md`.

Use the Offline Repository

By default, when starting an empty workspace, MicroEJ SDK is configured to import dependencies from *MicroEJ Central Repository*.

To configure MicroEJ SDK to import dependencies from a local module repository, follow these steps:

1. Open the *MMM preferences page*: `Window` > `Preferences` > `MicroEJ` > `Module Manager`.
2. In `Module Manager` group, click on `Import Repository`.
3. Select the module repository `.zip` file, and then click on `Finish`.

The import may take some time. The module repository is unzipped in the folder `${user.dir}/.microej/repositories`, and the settings are updated.

3.11.6 Select a VEE Port

Note: This page is releated to the version 5 and lower of the SDK. If you use the SDK 6, please refer to the page *Select a VEE Port*.

Building or running an Application or a *Testsuite* with MMM requires a VEE Port.

There are 4 different ways to provide a VEE Port for a module project:

- Set the *build option* `platform-loader.target.platform.file` to the path of a VEE Port file (`.zip`, `.jpf` or `.vde`).
- Set the *build option* `platform-loader.target.platform.dir` to the path of the `source` folder of an already imported *Source VEE Port*.
- Declare a *module dependency* with the conf `platform`:

```
<dependency␣
↪org="myorg" name="myname" rev="1.0.0" conf="platform->default" transitive="false"/>
```

- Copy a VEE Port file to the dropins folder. The default dropins folder location is `[module_project_dir]/dropins`. It can be changed using the *build option* `platform-loader.target.platform.dropins`.

Note: Using a VEE Port in the `.zip` format requires at least the version `5.4.0` of the SDK.

At least 1 of these 4 ways is required to build an Application with a VEE Port. If several ways are used, the following rules are applied:

- If `platform-loader.target.platform.file` or `platform-loader.target.platform.dir` is set, the other options are ignored.
- If the the module project defined several VEE Ports, the build fails. For example the following cases are not allowed:
  - Setting a VEE Port with the option `platform-loader.target.platform.file` and another one with the option `platform-loader.target.platform.dir`
  - Declaring a VEE Port as a dependency and adding a platform in the `dropins` folder
  - Declaring 2 VEE Ports as Dependencies
  - Adding 2 VEE Ports in the `dropins` folder

Refer to the *Platform Loader* section for a complete list of options.

3.11.7 Module Natures

Note: This page is releated to the version 5 and lower of the SDK. If you use the SDK 6, please refer to the page *Module Natures*.

This page describes the most common module natures as follows:

- **Skeleton Name**: the *project skeleton* name.
- **Build Type Name**: the build type name, derived from the module nature name: `com.is2t.easyant.buildtypes#build-[NATURE_NAME]`.
- **Documentation**: a link to the documentation.
- **SDK Menu**: the menu to the direct wizard in the SDK (if available). Any module nature can be created with the default wizard from `File` > `New` > `Module Project`.
- **Configuration**: properties that can be defined to configure the module. Properties are defined inside the `ea:build` tag of the *module.ivy* file, using `ea:property` tag as described in the section *Build Options*. A module nature also inherits the build options from the listed *Natures Plugins*.

Add-On Library

Skeleton Name: `microej-javalib`

Build Type Name: `com.is2t.easyant.buildtypes#build-microej-javalib`

Documentation: *Libraries*

SDK Menu: `File` > `New` > `Add-On Library Project`

Configuration:

This module nature inherits the build options of the following plugins:

- *Java Compilation*

- *Platform Loader*
- *Javadoc*
- *Testsuite*
- *Artifact Checker*

Add-On Processor

Skeleton Name: addon-processor

Build Type Name: com.is2t.easyant.buildtypes#build-addon-processor

Configuration:

This module nature inherits the build options of the following plugins:

- *Java Compilation*
- *Java SE Unit Tests*
- *Artifact Checker*

Foundation Library API

Skeleton Name: microej-javaapi

Build Type Name: com.is2t.easyant.buildtypes#build-microej-javaapi

Documentation: *Libraries*

Configuration:

This module nature inherits the build options of the following plugins:

- *Java Compilation*
- *Javadoc*
- *Artifact Checker*

This module nature defines the following dedicated build options:

Name	Description	Default
microej.lib.name	Platform library name on the form: [NAME]-[VERSION]-api . - [NAME] : name of the implemented Foundation Library API module. - [VERSION] : version of the implemented Foundation Library API module without patch (Major.minor).	Not set
rip.printableName	Printable name for the Platform Editor.	Not set

Foundation Library Implementation

Skeleton Name: microej-javaimpl

Build Type Name: com.is2t.easyant.buildtypes#build-microej-javaimpl

Documentation: *Libraries*

Configuration:

This module nature inherits the build options of the following plugins:

- *Java Compilation*
- *Testsuite*
- *Artifact Checker*[3]

This module nature defines the following dedicated build options:

Name	Description	Default
microej.lib.implfor	Execution target. Possible values are *emb* (only on Device), *sim* (only Simulator) and *common* (both).	common

Kernel Application

Skeleton Name: firmware-multiapp

Build Type Name: com.is2t.easyant.buildtypes#build-firmware-multiapp

Documentation: *Kernel Developer Guide*

Configuration:

This module nature inherits the build options of the following plugins:

- *Java Compilation*
- *Platform Loader*
- *Javadoc*
- *Artifact Checker*Page 1742, 3

This module nature defines the following dedicated build options:

Name	Description	Default
application.main.class	Full Qualified Name of the main class of the kernel. This option is required.	Not set
runtime.api.name	Name of the Runtime API of the kernel. This option is ignored when a *Runtime API* is declared in the dependencies.	RUNTIME
runtime.api.version	Version of the Runtime API of the kernel. This option is ignored when a *Runtime API* is declared in the dependencies.	1.0
skip.build.virtual.device	When this property is set (any value), the virtual device is not built.	Not set
virtual.device.sim.only	When this property is set (any value), the Executable is not built.	Not set
launch.properties.jvm	Additional options to pass to the JVM for building the Executable.	-Xmx1024M

[3] Require SDK version 5.5.0 or higher.

Meta Build

Skeleton Name: microej-meta-build

Build Type Name: com.is2t.easyant.buildtypes#microej-meta-build

Documentation: *Meta Build*

Configuration:

This module nature defines the following dedicated build options:

Name	Description	Default
metabuild.root	Path of the root folder containing the modules to build.	${basedir}/. .
private.modules.file	Name of the file listing the private modules to build.	private. modules.list
public.modules.file	Name of the file listing the public modules to build.	public. modules.list

Mock

Skeleton Name: microej-mock

Build Type Name: com.is2t.easyant.buildtypes#build-microej-mock

Documentation: *Mock*

Configuration:

This module nature inherits the build options of the following plugins:

- *Java Compilation*
- *Java SE Unit Tests*
- *Artifact Checker*[3]

Module Repository

Skeleton Name: artifact-repository

Build Type Name: com.is2t.easyant.buildtypes#build-artifact-repository

Documentation: *Module Repository*

Configuration:

This module nature inherits the build options of the following plugins:

- *Artifact Checker*

This module nature defines the following dedicated build options:

Name	Description	Default
architecture.configurations.includes[1]	Comma-separated list of configurations to include for the Architecture modules. Set dist,eval or dist,prod to include only evaluation or production Architectures or dist,eval,prod to include both.	dist,eval
bar.check.as.v2.module	When this property is set to true, the artifact checker uses the MicroEJ Module Manager semantic.	false
bar.javadoc.dir	Path of the folder containing the generated javadoc.	${target}/javadoc
bar.notification.email.from	The email address used as the from address when sending the notification emails.	Not set
bar.notification.email.host	The hostname of the mail service used to send the notification emails.	Not set
bar.notification.email.password	The password used to authenticate on the mail service.	Not set
bar.notification.email.port	The port of the mail service used to send the notification emails	Not set
bar.notification.email.ssl	When this property is set to true, SSL/TLS is used to send the notification emails.	Not set
bar.notification.email.to	The notification email address destination.	Not set
bar.notification.email.user	The username used to authenticate on the mail service.	Not set
bar.populate.from.resolver	Name of the resolver used to fetch the modules to populate the repository.	fetchRelease
bar.populate.ivy.settings.file	Path of the Ivy settings file used to fetch the modules to populate the repository.	${project.ivy.settings.file}
bar.populate.repository.conf	Ivy configuration of included repositories. The modules of the repositories declared as dependency with this configuration are included in the built repository.	repository
bar.test.haltonerror	When this property is set to true, the artifact checker stops at the first error.	false
javadoc.excludes	Comma-separated list of packages to exclude from the javadoc.	Empty string
javadoc.includes	Comma-separated list of packages to include in the javadoc.	** (all packages)
javadoc.modules.excludes[2]	Comma-separated list of modules to exclude from the javadoc.	Empty string
skip.artifact.checker	When this property is set to true, all artifact checkers are skipped.	Not set
skip.email	When this property is set (any value), the notification email is not sent. Otherwise the bar.notification.* properties are required.	Not set
skip.javadoc	Prevents the generation of the javadoc.	false
skip.javadoc.deprecated	Prevents the generation of any deprecated API at all in the javadoc.	true

[1] Require SDK version 5.4.0 or higher.
[2] Require SDK version 5.6.0 or higher.

Runtime Environment

Skeleton Name: runtime-api

Build Type Name: com.is2t.easyant.buildtypes#build-runtime-api

Documentation: *Runtime Environment*

Configuration:

This module nature inherits the configuration properties of the following plugins:

- *Artifact Checker*

Sandboxed Application

Skeleton Name: application

Build Type Name: com.is2t.easyant.buildtypes#build-application

Documentation: *Sandboxed Application*

SDK Menu: File > New > Sandboxed Application Project

Configuration:

This module nature inherits the build options of the following plugins:

- *Java Compilation*
- *Platform Loader*
- *Javadoc*
- *Testsuite*
- *Artifact Checker*

Standalone Application

Skeleton Name: firmware-singleapp

Build Type Name: com.is2t.easyant.buildtypes#build-firmware-singleapp

Documentation: *Standalone Application*

SDK Menu: File > New > Standalone Application Project

Configuration:

This module nature inherits the build options of the following plugins:

- *Java Compilation*
- *Platform Loader*
- *Javadoc*[Page 1742, 3]
- *Testsuite*[Page 1742, 3]
- *Artifact Checker*[Page 1742, 3]

This module nature defines the following dedicated build options:

Name	Description	Default
application.main.class	Full Qualified Name of the main class of the application. This option is required.	Not set
skip.build.virtual.device	When this property is set (any value), the virtual device is not built.	Not set
virtual.device.sim.only	When this property is set (any value), the Executable is not built.	Not set
launch.properties.jvm	Additional options to pass to the JVM for building the Executable.	-Xmx1024M

Studio Rebranding

Skeleton Name: microej-studio-rebrand

Build Type Name: com.is2t.easyant.buildtypes#build-izpack

Configuration:

The skeleton template contains all the necessary files for a Studio that is ready to build. The main elements are:

- HOWTO.md : This file describes the minimum configuration required to build the Studio template as it is.
- module.ivy : This file describes all available build options and dependencies.
- branding-resources : This folder contains default resources that can be replaced with your own to customize the Studio. These resources include names, images, icons, and license files.

Natures Plugins

This page describes the most common module nature plugins as follows:

- **Documentation**: link to documentation.
- **Module Natures**: list of *Module Natures* using this plugin.
- **Configuration**: properties that can be defined to configure the module. Properties are defined inside the ea:build tag of the *module.ivy* file, using ea:property tag as described in the section *Build Options*.

Java Compilation

Module Natures:

This plugin is used by the following module natures:

- *Add-On Library*
- *Foundation Library API*
- *Foundation Library Implementation*
- *Standalone Application*
- *Sandboxed Application*

Configuration:

This plugin defines the following build options:

Name	Description	Default
javac.debug.level	Comma-separated list of levels for the Java compiler debug mode.	lines,source,vars
javac.debug.mode	When this property is set to true, the Java compiler is set in debug mode.	false
src.main.java	Path of the folder containing the Java sources.	${basedir}/src/main/java

Platform Loader

Documentation: *Select a VEE Port*

Module Natures:

This plugin is used by the following module natures:

- *Add-On Library*
- *Standalone Application*
- *Sandboxed Application*

Configuration:

This plugin defines the following build options:

Name	Description	Default
platform-loader.platform.dir	Path of the folder to unzip the loaded platform to.	${target}/platform
platform-loader.skip.load.platform	When this property is set to true, the platform is not loaded. It must be already available in the directory defined by the property platform-loader.platform.dir . Use with caution: the platform content may be modified during the build (e.g. in case of Testsuite or Virtual Device build).	false
platform-loader.target.platform.conf	The ivy configuration used to retrieved the platform if fetched via dependencies.	platform
platform-loader.target.platform.dir	Path of the root folder of the platform to use in the build. See *Select a VEE Port* section for Platform Selection rules.	Not set
platform-loader.target.platform.dropins	Absolute or relative (to the project root folder) path of the folder where the platform can be found (see *Select a VEE Port*).	dropins
platform-loader.target.platform.file	Path of the platform file to use in the build. See *Select a VEE Port* section for Platform Selection rules.	Not set

Javadoc

Module Natures:

This plugin is used by the following module natures:

- *Add-On Library*
- *Foundation Library API*
- *Sandboxed Application*

Configuration:

This plugin defines the following build options:

Name	Description	Default
src.main.java	Path of the folder containing the Java sources.	${basedir}/src/main/java
javadoc.file.encoding	Encoding used for the generated Javadoc.	UTF-8
javadoc.failonerror	When this property is set to true, the build is stopped if an error is raised during the Javadoc generation.	true
javadoc.failonwarning	When this property is set to true, the build is stopped if a warning is raised during the Javadoc generation.	false
target.reports	Path of the base folder for reports.	${target}/reports
target.javadoc	Path of the base folder where the Javadoc is generated.	${target.reports}/javadoc
target.javadoc.main	Path of the folder where the Javadoc is generated.	${target.javadoc}/main
javadoc-microej.overview.html	Path of the HTML template file used for the Javadoc overview page.	${src.main.java}/overview.html if exists, otherwise a default template.
target.artifacts	Path of the packaged artifacts.	${target}/artifacts
target.artifacts.main.javadoc.jar.name	Name of the packaged JAR containing the generated Javadoc (stored in folder target.artifacts).	${module.name}-javadoc.jar
javadoc.publish.conf	Ivy configuration used to publish the Javadoc artifact.	documentation

Testsuite

Documentation: *Testsuite with JUnit*

Module Natures:

This plugin is used by the following module natures:

- *Add-On Library*
- *Foundation Library API*
- *Foundation Library Implementation*
- *Standalone Application*
- *Sandboxed Application*

Configuration:

This plugin defines the following build options:

Name	Description	Default
microej.testsuite.cc.excludes.classes	Pattern of classes excluded from the code coverage analysis.	Not set
microej.testsuite.retry.count	A test execution may not be able to produce the success trace for an external reason, for example an unreliable harness script that may lose some trace characters or crop the end of the trace. For all these unlikely reasons, it is possible to configure the number of retries before a test is considered to have failed.	0
microej.testsuite.timeout	The time in seconds before a test is considered as failed. Set it to 0 to disable the timeout.	60
microej.testsuite.properties.[name]	Inject an *Application Option* named [name] for all tests. For example, declaring the build option microej.testsuite.properties.core.memory.javaheap.size will configure the Managed Heap size of all tests.	Not applicable
microej.testsuite.properties.launch.test.trace.file	Set this property to true if your *VEE Port Run script* redirects execution traces.	Not set
microej.testsuite.properties.s3.cc.activated	When this property is set to true, the code coverage analysis is enabled.	true
microej.testsuite.properties.testsuite.trace.ip	The TCP/IP address to connect for retrieving test execution traces. This property is required if your *VEE Port Run script* does not redirect execution traces.	Not set
microej.testsuite.properties.testsuite.trace.port	The TCP/IP port to connect for retrieving test execution traces. This property is required if your *VEE Port Run script* does not redirect execution traces.	Not set
microej.testsuite.properties.testsuite.trace.timeout	The time in seconds without activity on the standard output before the trace analysis is stopped.	75
cc.src.folders	Path to the folders containing the Java sources used for code coverage analysis.	Java source folder (src/main/java) and Add-On Processor generated source folders (src-adpgenerated/*)[4]
microej.testsuite.verbose	When this property is set to true, the verbose trace level is enabled.	false
test.run.excludes.pattern	Pattern of classes excluded from the testsuite execution.	Empty string (no test)
test.run.failonerror	When this property is set to true, the build fails if an error is raised.	true
target.vm.name	The execution target (S3 to execute on Simulator, MICROJVM to execute on the Device).	S3
test.run.includes.pattern	Pattern of classes included in the testsuite execution.	**/* (all tests)
skip.test	When this property is set (any value), the tests are not executed.	Not set

[4] Option cc.src.folders is not set by default for SDK versions lower than 5.5.0 .

Java SE Unit Tests

> **Warning:** This plugin is reserved for tools written in Java Standard Edition. Tests classes must be created in the folder `src/test/java` of the project. See *Testsuite* section for MicroEJ tests.

Module Natures:

This plugin is used by the following module natures:

- *Add-On Processor*
- *Mock*

Configuration:

This plugin defines the following build options:

Name	Description	Default
test.run.excludes.pattern	Pattern of classes excluded from the testsuite execution.	Empty string (no test)
test.run.failonerror	When this property is set to true, the build fails if an error is raised.	`true`
test.run.includes.pattern	Pattern of classes included in the testsuite execution.	`**/*` (all tests)
skip.test	When this property is set (any value), the tests are not executed.	Not set

Artifact Checker

Module Natures:

This plugin is used by the following module natures:

- *Add-On Library*
- *Foundation Library API*
- *Standalone Application*
- *Sandboxed Application*
- *Module Repository*

Configuration:

This plugin defines the following build options:

Name	Description	Default
run.artifact.checker	When this property is set (any value), the artifact checker is executed.	Not set
skip.addonconf.checker	When this property is set to true, the addon configurations checker is not executed.	Not set
skip.changelog.checker	When this property is set to true, the changelog checker is not executed.	Not set
skip.foundation-conf.checker	When this property is set to true, the foundation configurations checker is not executed.	Not set
skip.license.checker	When this property is set to true, the license checker is not executed.	Not set
skip.nullanaly-sis.checker[5]	When this property is set to true, the null analysis checker is not executed.	Not set
skip.publicconf.checker	When this property is set to true, the public configurations checker is not executed.	Not set
skip.readme.checker	When this property is set to true, the readme checker is not executed.	Not set
skip.retrieve.checker	When this property is set to true, the retrieve checker is not executed.	Not set

Global Build Options

The following *Build Options* are available in any module:

Name	Description	Default
target	Path of the build directory target~ .	${basedir}/target~

3.11.8 Debug an Application

Note: This page is releated to the version 5 and lower of the SDK. If you use the SDK 6, please refer to the page *Debug on Simulator*.

Debug on Simulator

To debug an application on Simulator, select it in the left panel then right-click and select Debug As > MicroEJ Application .

[5] Require SDK version 5.5.0 or higher.

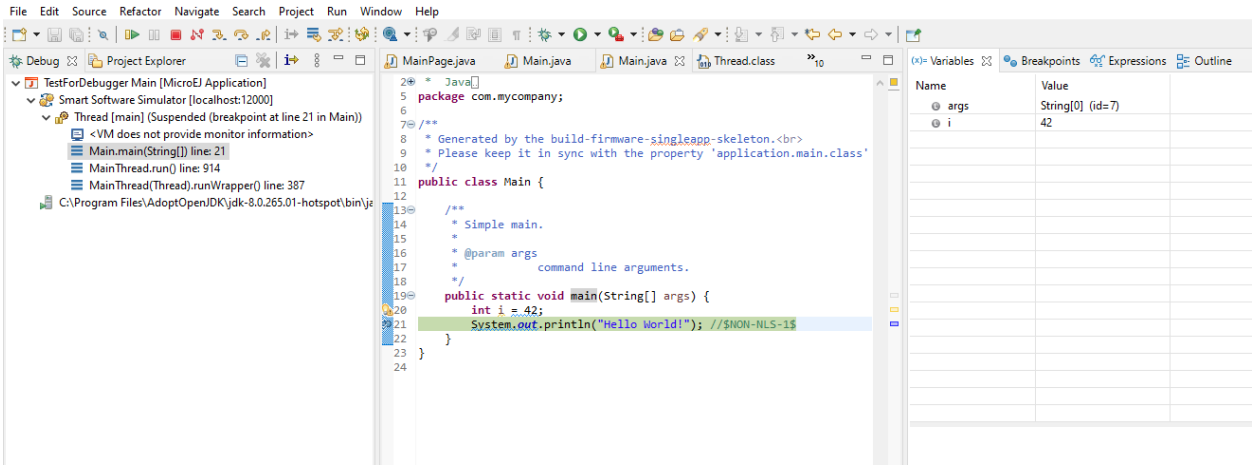


Fig. 352: MicroEJ Development Tools Overview of the Debugger on Simulator

Debug on Device

To debug an application on device, first run the *MicroEJ debugger proxy*, and run a Remote Java Application launch:

- Go to `Run` > `Debug Configurations` > `Remote Java Application`
- Set the informations about the project to debug, the proxy connections properties, etc.
- Click on `Debug`

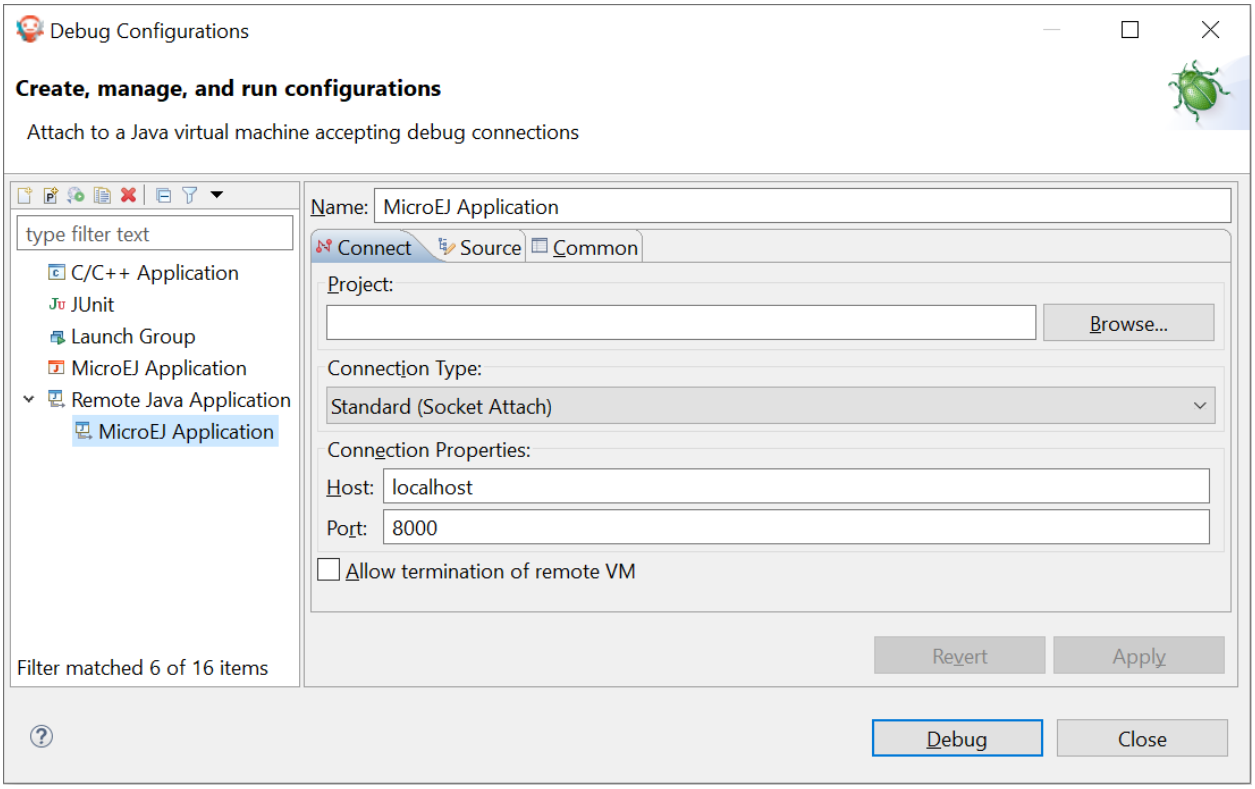


Fig. 353: MicroEJ Development Tools Overview of the Remote Java Application

- In the SDK, open the Debug perspective (Window > Perspective > Open Perspective > Other… > Debug) to show the current debugging process.
- Click on the Pause icon to access the thread list, including their stack trace

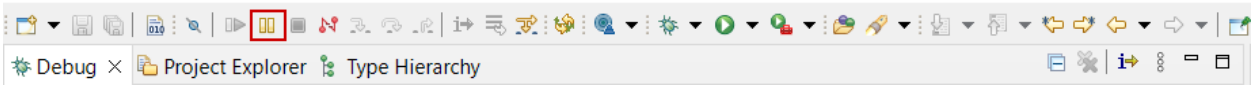


Fig. 354: MicroEJ Development Tools Overview of the Debugger on Board Toolbar

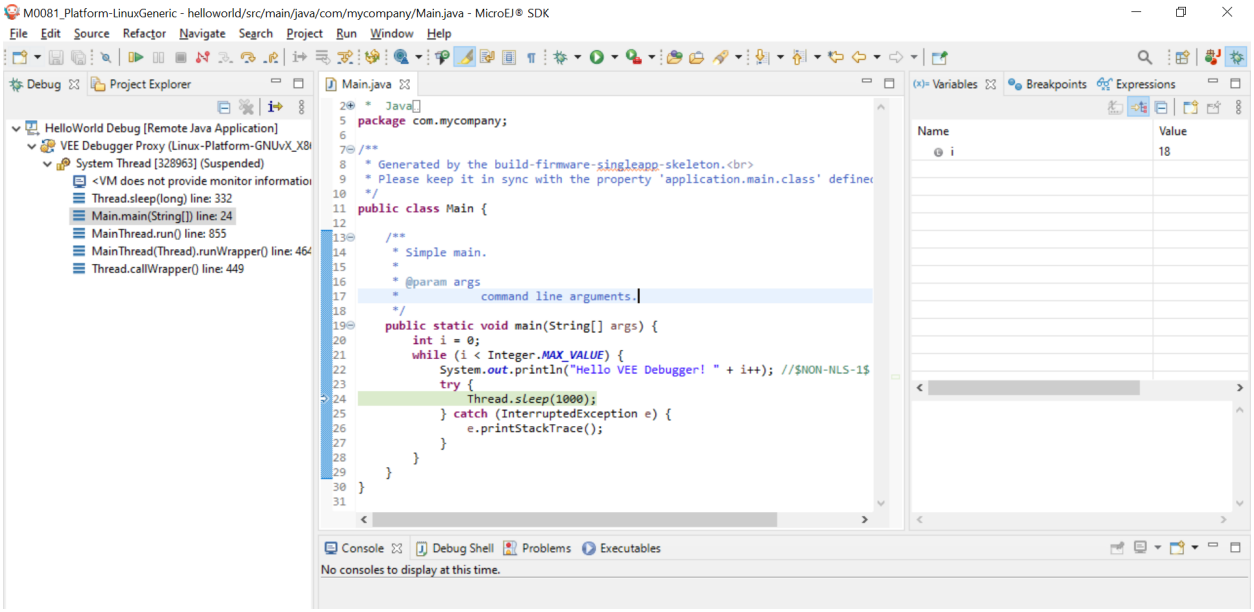


Fig. 355: MicroEJ Development Tools Overview of the Debugger on Board

It makes use of Eclipse Java debugger client. If you are unfamiliar with Java debugging or Eclipse IDE, see Debugging the Eclipse IDE for Java Developers to get started.

You can also debug with IntelliJ IDEA. For more information on IntelliJ IDEA Remote debug process, see IntelliJ IDEA Remote debug

Get Library Sources

All libraries included in MicroEJ SDK are provided with their source code and resources. The way the sources are retrieved depends on the kind of library (Add-On Library or Foundation Library).

Add-On Library Sources

Add-On Library sources are packaged in a dedicated file named [module_name]-source.jar available in the module directory:

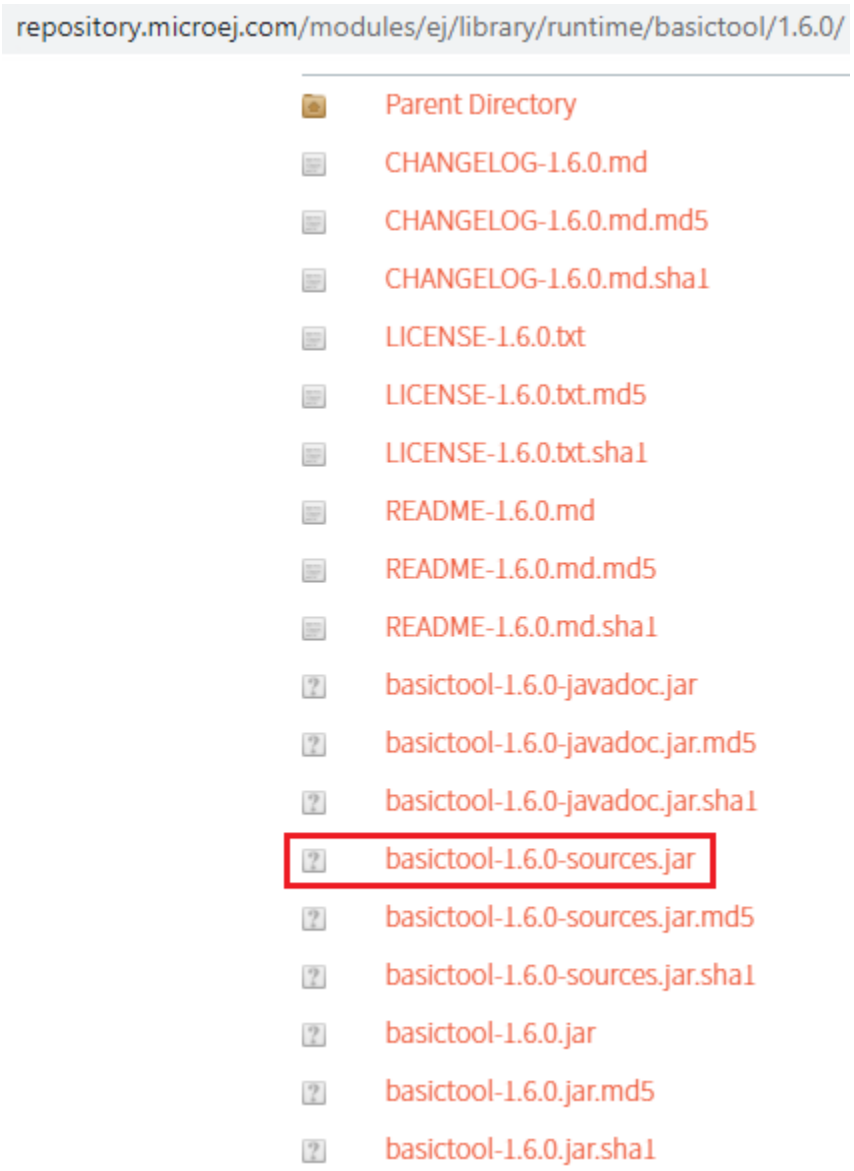


Fig. 356: Add-On Library Sources Location

In the SDK, sources are automatically connected to Eclipse JDT when the new Add-On Library is added as a *module dependency*.

On any Java element (type, method, field), press F3 or CTRL-Click to open the implementation:

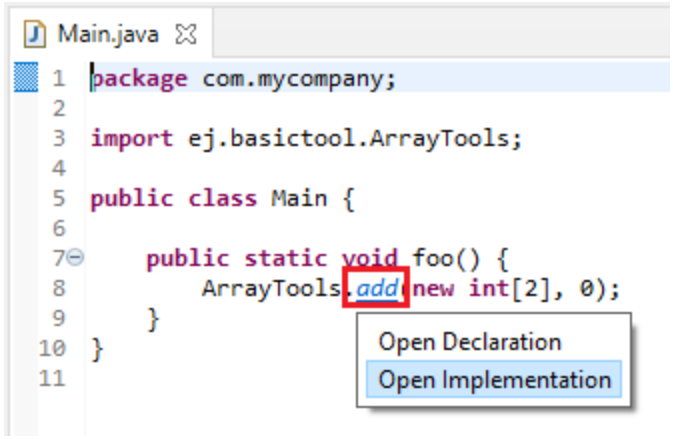


Fig. 357: Add-On Library Open Implementation

Then the implementation class is open in read-only mode.

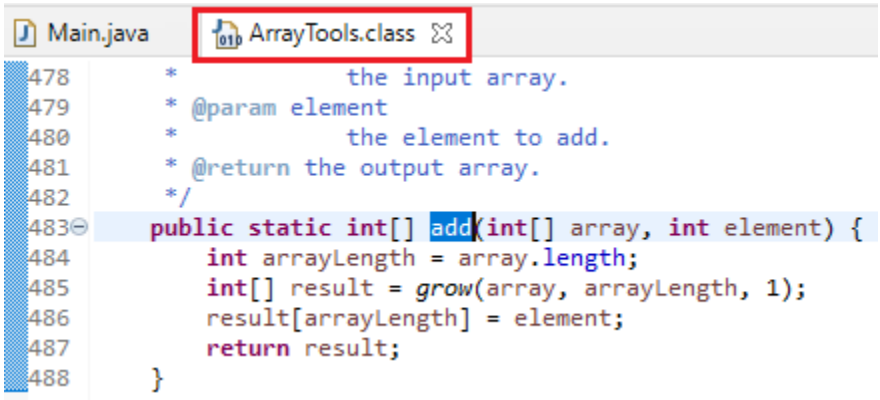


Fig. 358: Add-On Library Read-Only Source Code

Foundation Library Sources

Foundation Library sources are directly included in the implementation file (JAR file) provided by the Platform.

They are located in the following Platform folders:

- `javaLibs` for generic Foundation Libraries (defaults).
- `MICROJVM/javaLibs` for Foundation Libraries specific to the Core Engine.
- `S3/javaLibs` for Foundation Libraries specific to the Simulator.

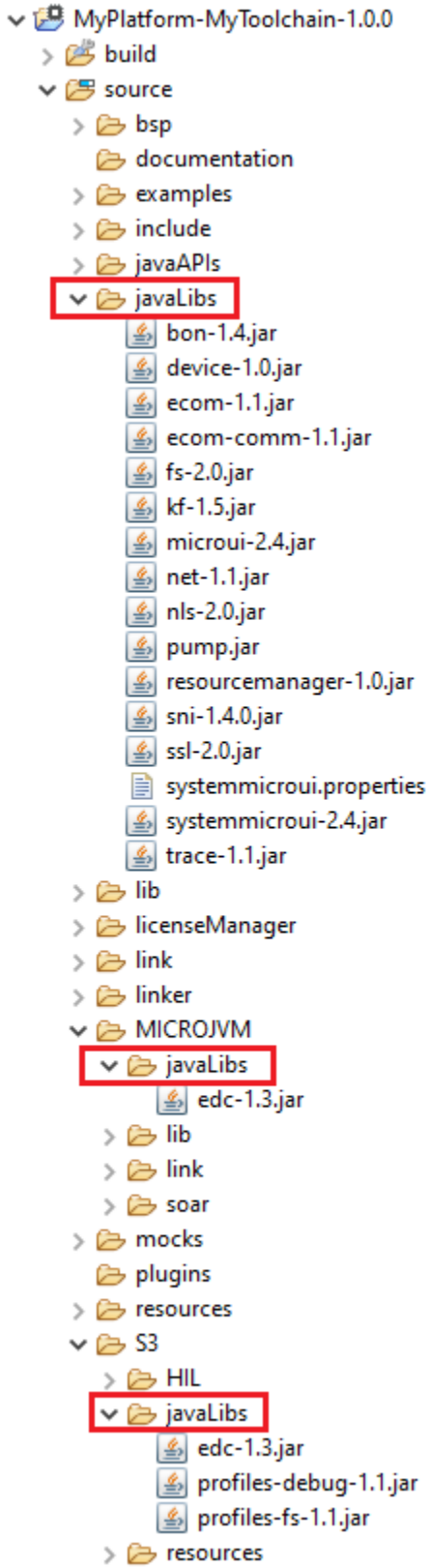


Fig. 359: Foundation Library Platform Folders

In the SDK, sources can be connected while debugging an Application on Simulator. This ensures to get the exact source code which is executed on your Platform.

Here are the steps to attach Foundation Library sources from a Platform loaded in the workspace:

- Open a *MicroEJ Application launch*,
- Select the `Source` tab (see also *Source Tab*),
- Click on `Add…` button,
- Select `Archive` item and press `OK` ,

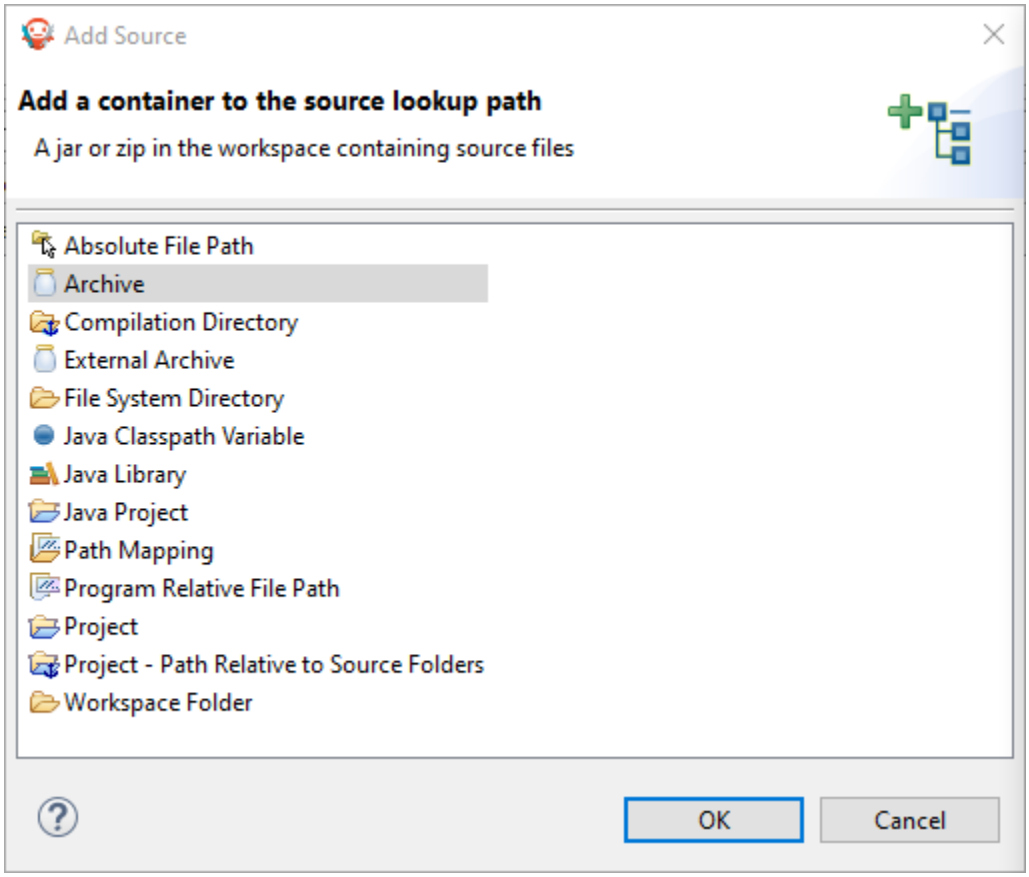


Fig. 360: Add Foundation Library Sources to MicroEJ Application Launch

- Select the Foundation Libraries from Platform folders and press `OK` ,

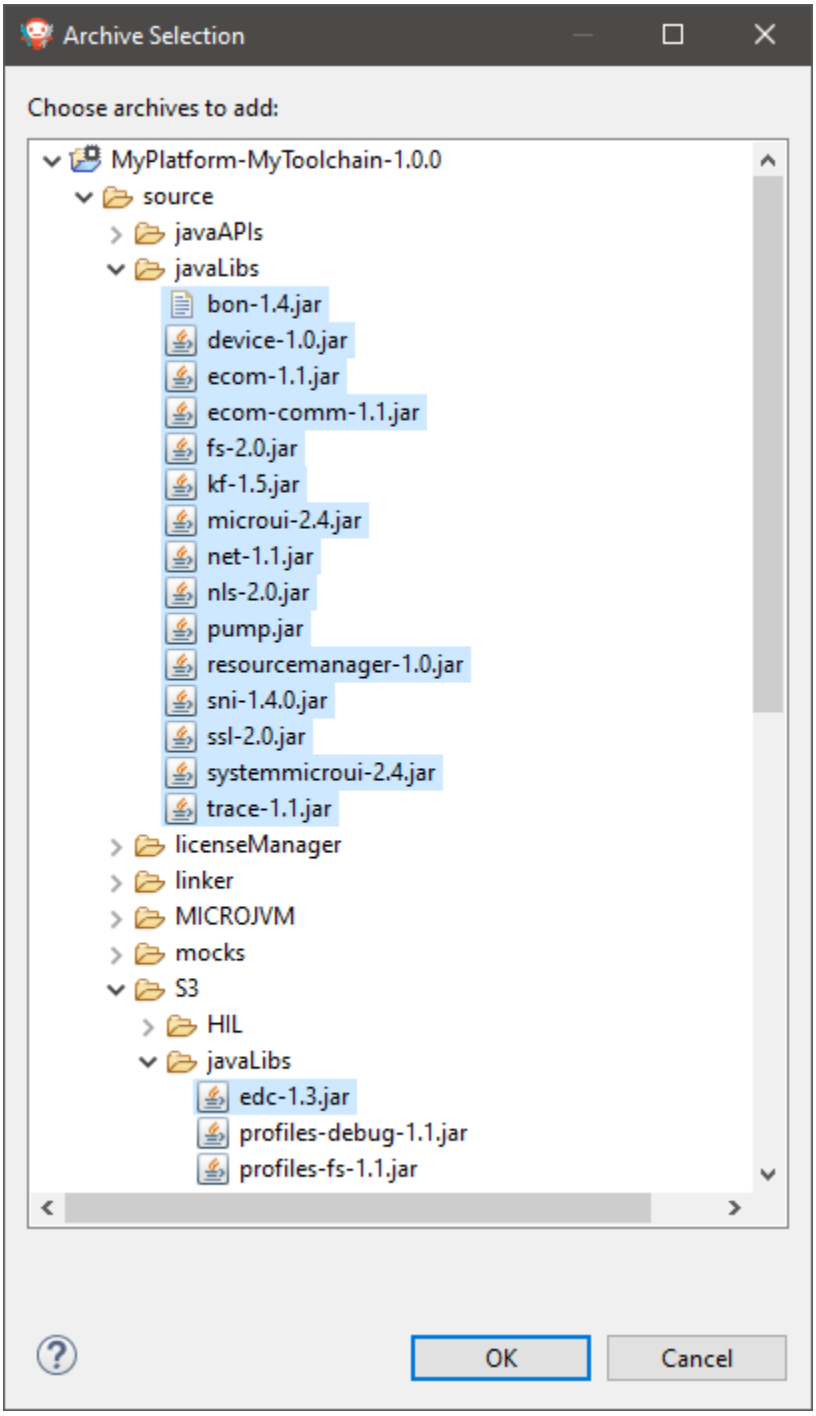


Fig. 361: Select Foundation Libraries Implementation files

> **Warning:** You must select the libraries from the Platform project corresponding to the execution Platform (see *Execution Tab*).

In the debug session the implementation sources will be now displayed.

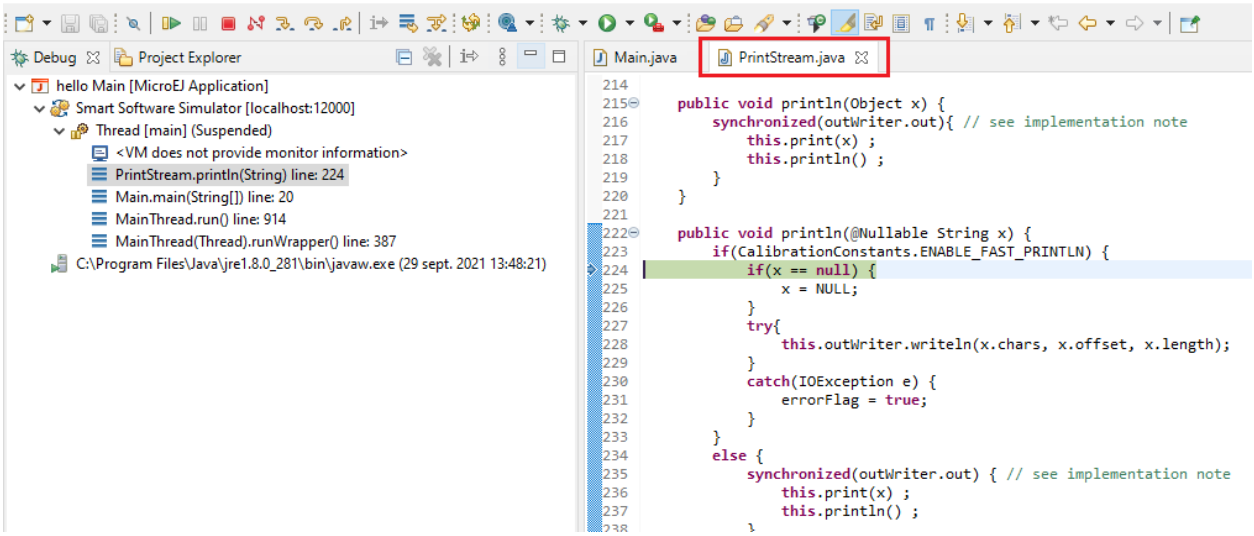


Fig. 362: Foundation Library Read-Only Source Code

3.11.9 Development Tools

MicroEJ provides a number of tools to assist with various aspects of development. Some of these tools are run using MicroEJ Tool configurations, and created using the Run Configurations dialog of the MicroEJ SDK. A configuration must be created for the tool before it can be used.

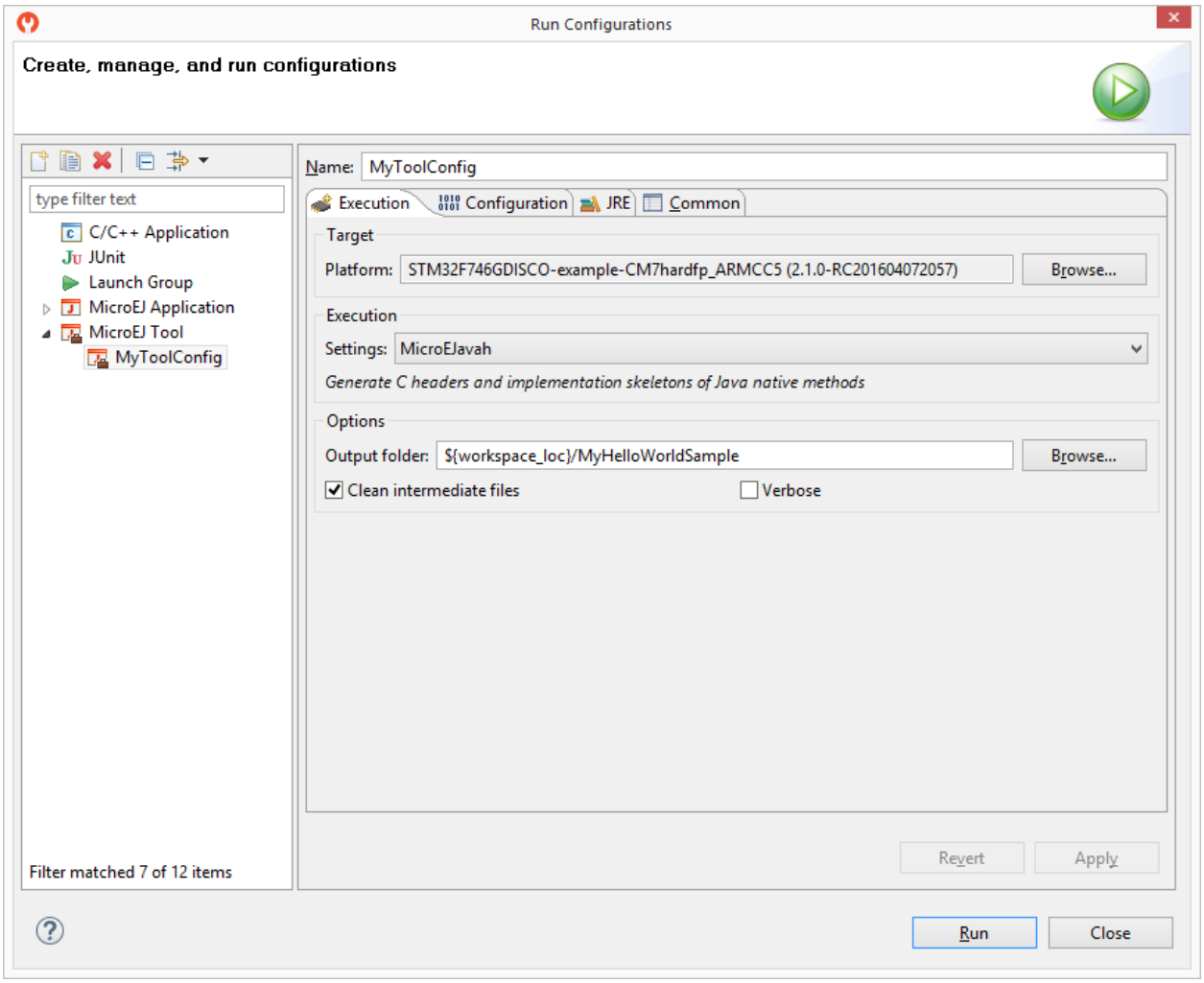


Fig. 363: MicroEJ Tool Configuration

The above figure shows a tool configuration being created. In the figure, the MicroEJ Platform has been selected, but the selection of which tool to run has not yet been made. That selection is made in the Execution Settings… box. The Configuration tab then contains the options relevant to the selected tool.

Testsuite with JUnit

The SDK allows to run unit tests using the standard JUnit API during the build process of a Library or an Application. The *MicroEJ Testsuite Engine* runs tests on a VEE Port and outputs a JUnit XML report.

Principle

JUnit testing can be enabled when using the microej-javalib (MicroEJ Add-On Library) or the microej-application (MicroEJ Applications) build type. JUnit test cases processing is automatically enabled when the following dependency is declared in the module.ivy file of the project.

```
<dependency conf="test->*" org="ej.library.test" name="junit" rev="1.6.2"/>
```

When a new JUnit test case class is created in the src/test/java folder, a JUnit processor generates MicroEJ compliant classes into a specific source folder named src-adpgenerated/junit/java. These files are automatically managed and must not be edited manually.

JUnit Compliance

MicroEJ is compliant with a subset of JUnit version 4. MicroEJ JUnit processor supports the following annotations: @After, @AfterClass, @Before, @BeforeClass, @Ignore, @Test.

Each test case entry point must be declared using the org.junit.Test annotation (@Test before a method declaration). Please refer to JUnit documentation to get details on usage of other annotations.

Setup a Platform for Tests

Before running tests, a target platform must be configured.

Execution in SDK

In order to execute the Testsuite in the SDK, a target platform must be configured in the MicroEJ workspace. The following steps assume that a platform has been previously imported into the MicroEJ Platform repository (or available in the Workspace):

- Go to Window > Preferences > MicroEJ > Platforms (or Platforms in workspace).
- Select the desired platform on which to run the tests.
- Press F2 to expand the details.
- Select the the platform path and copy it to the clipboard.
- Go to Window > Preferences > Ant > Runtime and select the Properties tab.
- Click on Add Property... button and set a new property named target.platform.dir with the platform path pasted from the clipboard.

Execution during module build

In order to execute the Testsuite during the build of the module, a target platform must be configured in the module project as described in the section *Select a VEE Port*.

Setup a Project with a JUnit Test Case

This section describes how to create a new JUnit Test Case starting from a new MicroEJ library project.

- First create a new *module project* using the microej-javalib skeleton. A new project named mylibrary is created in the workspace.
- Right-click on the src/test/java folder and select New > Other... menu item.
- Select the Java > JUnit > New JUnit Test Case wizard.
- Enter a test name and press Finish . A new JUnit test case class is created with a default failing test case.

Build and Run a JUnit Testsuite

- Right-click on the mylibrary project and select Build Module . After the library is built, the testsuite engine launches available test cases and the build process fails in the console view.
- On the mylibrary project, right-click and select Refresh . A target~ folder appears with intermediate build files. The JUnit report is available at target~\test\xml\TEST-test-report.xml .
- Double-click on the file to open the JUnit testsuite report.
- Modify the test case by replacing

```
fail("Not yet implemented");
```

with

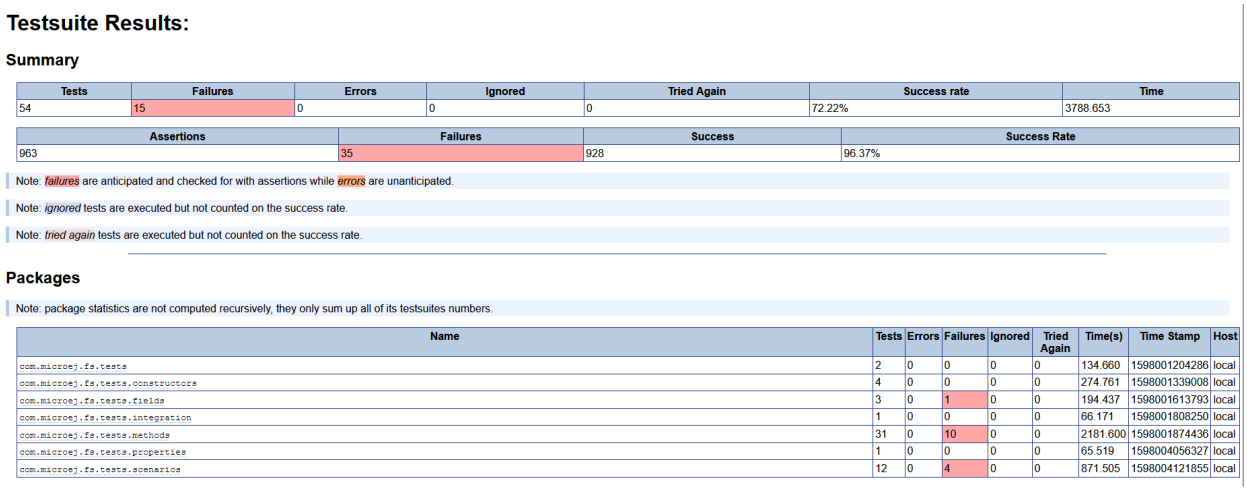
```
Assert.assertTrue(true);
```

- Right-click again on the mylibrary project and select Build Module . The test is now successfully executed on the target platform so the MicroEJ Add-On Library is fully built and published without errors.
- Double-click on the JUnit testsuite report to see the test has been successfully executed.

Testsuite Reports

Once a testsuite is completed, the following testsuite reports are generated:

- JUnit HTML report in the module project location target~/test/html/test/junit-noframes.html . This report contains a summary and the execution trace of every executed test.



Testsuite Results:

Summary

Tests	Failures	Errors	Ignored	Tried Again	Success rate	Time
54	15	0	0	0	72.22%	3788.653

Assertions	Failures	Success	Success Rate
963	35	928	96.37%

Note: *failures* are anticipated and checked for with assertions while *errors* are unanticipated.

Note: *ignored* tests are executed but not counted on the success rate.

Note: *tried again* tests are executed but not counted on the success rate.

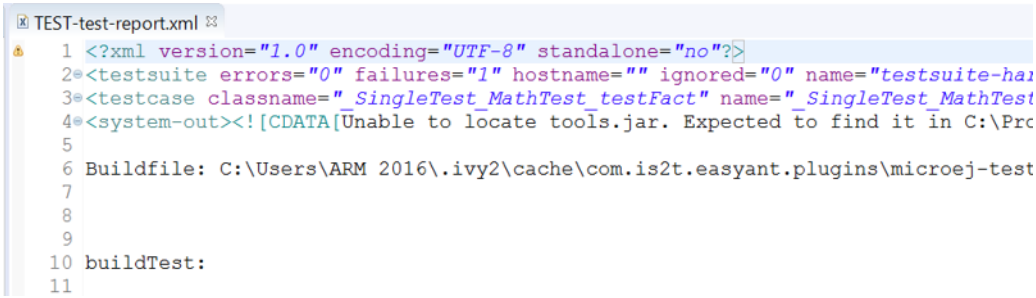
Packages

Note: package statistics are not computed recursively, they only sum up all of its testsuites numbers.

Name	Tests	Errors	Failures	Ignored	Tried Again	Time(s)	Time Stamp	Host
com.microej.fs.tests	2	0	0	0	0	134.660	1598001204286	local
com.microej.fs.tests.constructors	4	0	0	0	0	274.761	1598001339008	local
com.microej.fs.tests.fields	3	0	1	0	0	194.437	1598001613793	local
com.microej.fs.tests.integration	1	0	0	0	0	66.171	1598001808250	local
com.microej.fs.tests.methods	31	0	10	0	0	2181.600	1598001874436	local
com.microej.fs.tests.properties	1	0	0	0	0	65.519	1598004056327	local
com.microej.fs.tests.scenarios	12	0	4	0	0	871.505	1598004121855	local

Fig. 364: Example of MicroEJ Testsuite HTML Report

- JUnit XML report in the module project location target~/test/xml/TEST-test-report.xml .



TEST-test-report.xml

```
1  <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2  <testsuite errors="0" failures="1" hostname="" ignored="0" name="testsuite-ha
3  <testcase classname="_SingleTest_MathTest_testFact" name="_SingleTest_MathTest
4  <system-out><![CDATA[Unable to locate tools.jar. Expected to find it in C:\Pro
5
6  Buildfile: C:\Users\ARM 2016\.ivy2\cache\com.is2t.easyant.plugins\microej-test
7
8
9
10 buildTest:
11
```

Fig. 365: Example of MicroEJ Testsuite XML Report

XML report file can also be opened in the JUnit View. Right-click on the file > Open With > JUnit View :

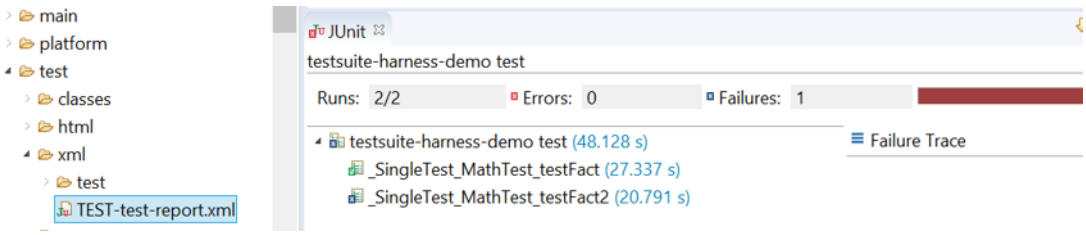


Fig. 366: Example of MicroEJ Testsuite XML Report in JUnit View

If executed on device, the Firmware binary produced for each test is available in module project location target~/test/xml/<TIMESTAMP>/bin/<FULLY-QUALIFIED-CLASSNAME>/application.out .

Configure the Execution on your Device

By default, the Testsuite is configured to execute tests on the Simulator using Mocks declared by the VEE Port. You can switch the default configuration to execute tests on your Device. For that, your VEE Port must implement the *BSP Connection*.

Also, a device must be connected to your workstation both for programming the Executable and getting output traces. Consult your VEE Port specific documentation for setup.

Here is a summary of the options to add (see *Testsuite Options* and *BSP Connection Options* for more details).

```
<!-- Execute tests on Device -->
<ea:property name="target.vm.name" value="MICROJVM"/>

<!-- Enable Executable built using the SDK -->
<ea:property name="microej.testsuite.properties.deploy.bsp.microejscript" value="true"/>
<ea:property␣
↪name="microej.testsuite.properties.microejtool.deploy.name" value="deployToolBSPRun"/>

<!-- Tell the testsuite engine that your VEE Port Run script redirects execution traces -->
<ea:property name="microej.testsuite.properties.launch.test.trace.file" value="true"/>
<!-- Configure TCP/
↪IP address and port if your VEE Port Run script does not redirect execution traces -->
<ea:property name="microej.testsuite.properties.testsuite.trace.ip" value="127.0.0.1"/>
<ea:property name="microej.testsuite.properties.testsuite.trace.port" value="5555"/>
```

Warning: If your VEE Port Run script does not redirect execution traces, the *Serial to Socket Transmitter* tool must have been started before running the Testsuite.

Advanced Configurations

Autogenerated Test Classes

The JUnit processor generates test classes into the src-adpgenerated/junit/java folder. This folder contains:

_AllTestClasses.java file
: A single class with a main entry point that sequentially calls all declared test methods of all JUnit test case classes.

AllTests[TestCase].java files
: For each JUnit test case class, a class with a main entry point that sequentially calls all declared test methods.

SingleTest[TestCase]_[TestMethod].java files
: For each test method of each JUnit test case class, a class with a main entry point that calls the test method.

JUnit Test Case to MicroEJ Test Case

The *MicroEJ Testsuite Engine* allows to select the classes that will be executed, by adding the following configuration in the project build file:

MMM (module.ivy)

Gradle (build.gradle.kts)

```
<ea:property␣
↪name="test.run.includes.pattern" value="[MicroEJ Test Case Include Pattern]"/>
```

```
tasks.test {
  filter {
    includeTestsMatching([MicroEJ Test Case Include Pattern])
  }
}
```

The following configuration considers all JUnit test methods of the same class as a single MicroEJ test case (default behavior). If at least one JUnit test method fails, the whole test case fails in the JUnit report.

MMM (module.ivy)

Gradle (build.gradle.kts)

```
<ea:property name="test.run.includes.pattern" value="**/_AllTests_*.class"/>
```

```
tasks.test {
  filter {
    includeTestsMatching("*._AllTests_*")
  }
}
```

The following configuration considers each JUnit test method as a dedicated MicroEJ test case. Each test method is viewed independently in the JUnit report, but this may slow down the test-suite execution because a new deployment is done for each test method.

MMM (module.ivy)

Gradle (build.gradle.kts)

```
<ea:property name="test.run.includes.pattern" value="**/_SingleTest_*.class"/>
```

```
tasks.test {
  filter {
    includeTestsMatching("*._SingleTest_*")
  }
}
```

Testsuite Options (SDK 5 only)

The *MicroEJ Testsuite Engine* can be configured with specific options which can be added to the `module.ivy` file of the project running the testsuite, within the `<ea:build>` XML element.

Testsuite options are described in the *Testsuite Module Nature* section.

Test Specific Options

The *MicroEJ Testsuite Engine* allows to define *Standalone Application Options* specific to each test case. This can be done by defining a file with the same name as the generated test case file with the `.properties` extension instead of the `.java` extension. The file must be put in the `src/test/resources` folder and within the same package than the test case file.

Stack Trace Reader

Principle

Stack Trace Reader is a MicroEJ tool that reads and decodes the MicroEJ stack traces. When an exception occurs, the Core Engine prints the stack trace on the standard output `System.out`. The class names, non-required types names(see *Types*), and method names obtained are encoded with a MicroEJ internal format. This internal format prevents embedding all class names and method names in the executable image to save some memory space. The Stack Trace Reader tool allows you to decode the stack traces by replacing the internal class names and method names with their real names. It also retrieves the line numbers in the MicroEJ Application.

Functional Description

The Stack Trace Reader reads the debug information from the fully linked ELF file (the ELF file that contains the Core Engine, the other libraries, the BSP, the OS, and the compiled MicroEJ Application). It prints the decoded stack trace.

When *Multi-Sandbox capability* is enabled, the Stack Trace Reader can simultaneously decode heterogeneous stack traces with lines owned by different MicroEJ Sandboxed Applications and the firmware. Lines owned by the firmware can be decoded with the firmware debug information file (optionally made available by your firmware provider).

Dependencies

No dependency.

Installation

This tool is a built-in Architecture tool.

Use (Standalone Application)

For example, write the following new line to dump the currently executed stack trace on the standard output.

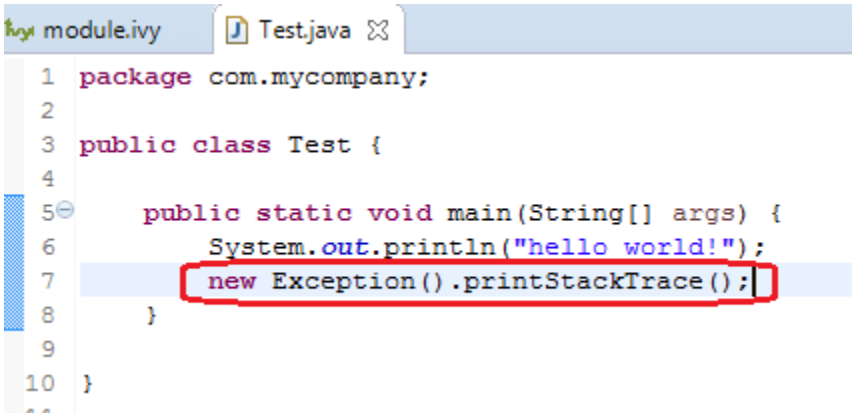


Fig. 367: Code to Dump a Stack Trace

To decode an application stack trace, the Stack Trace Reader tool requires the application executable ELF file. In the case of a platform with full BSP connection (see *BSP Connection Cases*), the file is `application.out` in the output folder. In the other cases, the ELF file is generated by the C toolchain when building the BSP project (usually a `.out` or `.axf` file).

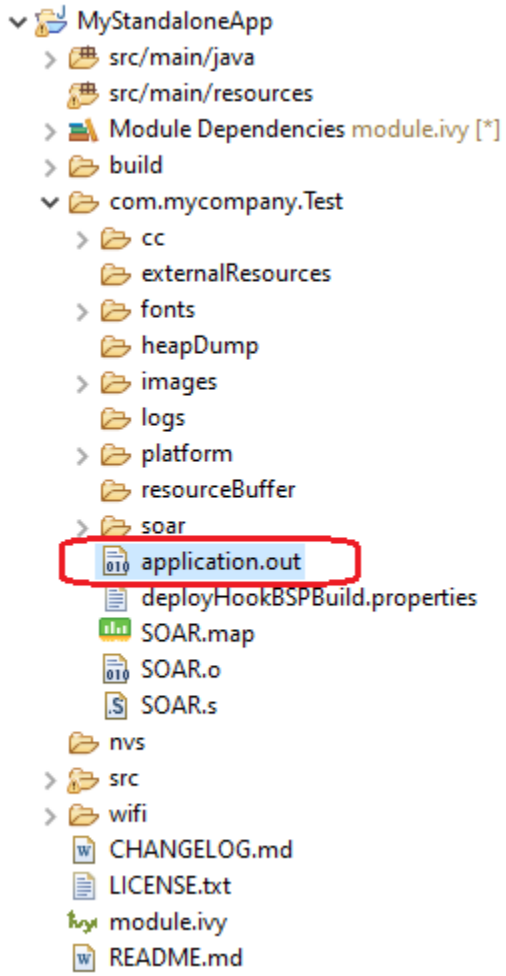


Fig. 368: Application Binary File

On successful deployment, the application is started on the device and the following trace is dumped on standard output.

```
VM START
Hello World!
Exception in thread "main" java.lang.Exception
    at java.lang.System.@M:0x3f407778:0x3f407782@
    at java.lang.Throwable.@M:0x3f408030:0x3f408046@
    at java.lang.Throwable.@M:0x3f4089cc:0x3f4089e6@
    at com.mycompany.Test.@M:0x3f40762c:0x3f407652@
    at java.lang.MainThread.@M:0x3f407a84:0x3f407a98@
    at java.lang.Thread.@M:0x3f408b88:0x3f408b94@
    at java.lang.Thread.@M:0x3f408c74:0x3f408c7f@
VM END (exit code = 0)
```

Fig. 369: Stack Trace Output

To create a new MicroEJ Tool configuration, right-click on the application project and click on

Run As… > Run Configurations… .

Create a new MicroEJ Tool configuration. In the Execution tab, select your target platform, then select the Stack Trace Reader tool. Set an output folder in the Output folder field.

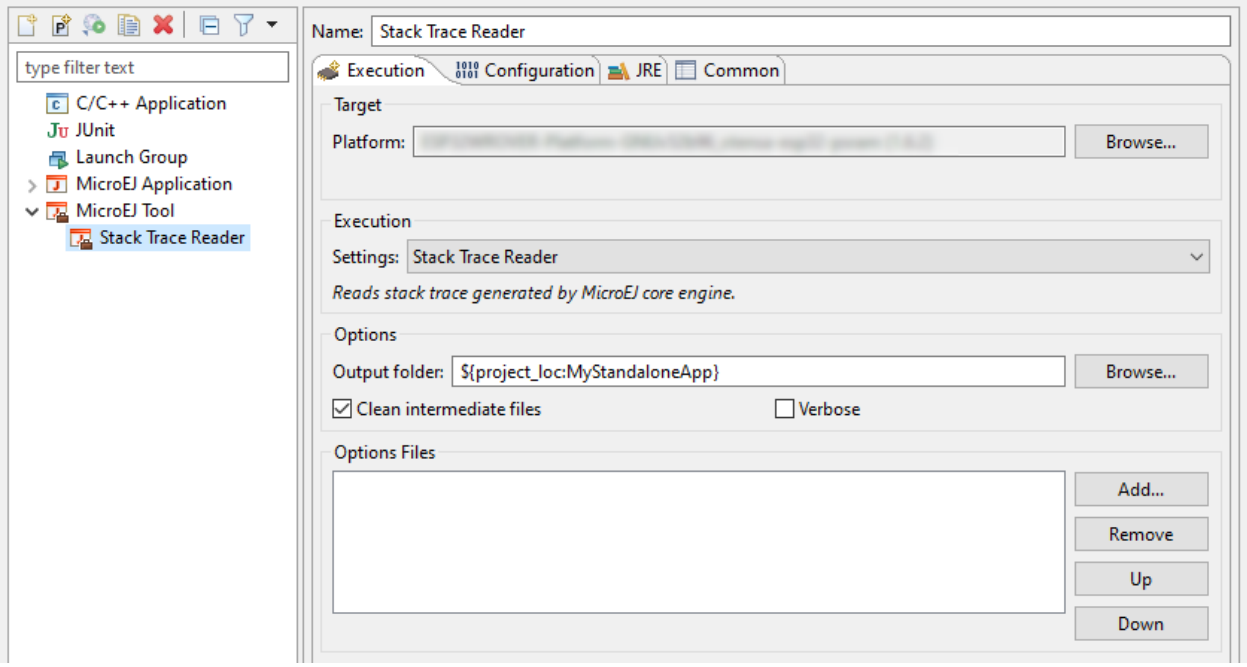


Fig. 370: Stack Trace Reader Tool Configuration (Platform Selection)

In Configuration tab, browse the previously generated application binary file with debug information (application.out in case of a Standalone Application with full BSP connection)

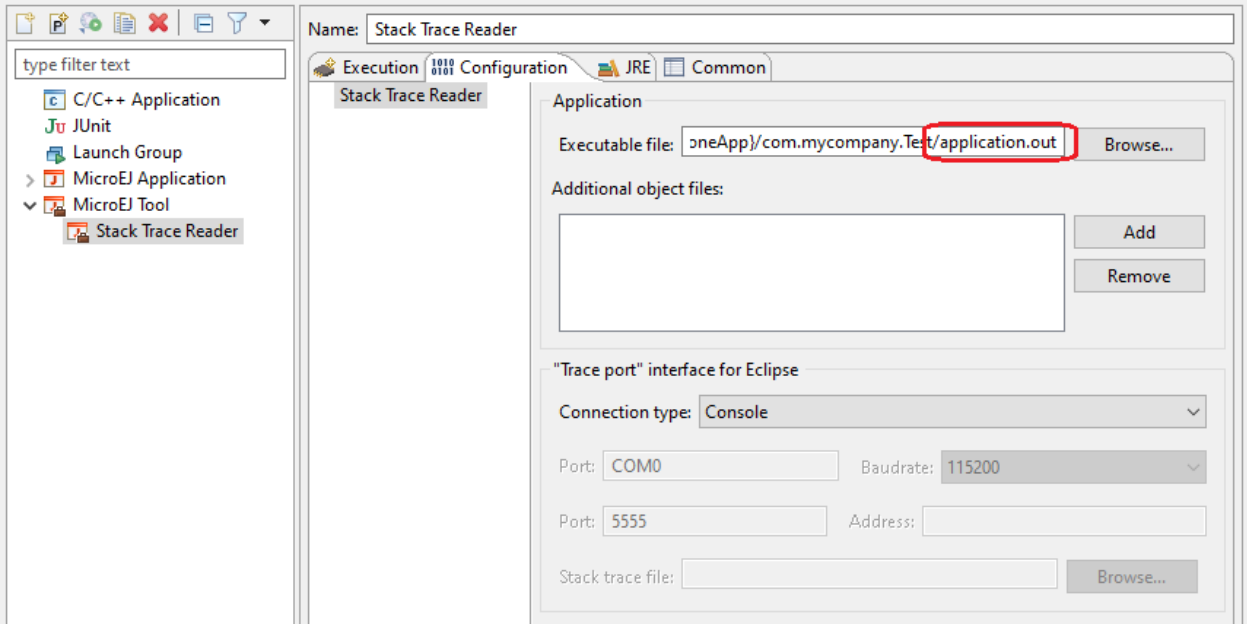


Fig. 371: Stack Trace Reader Tool Configuration (Standalone Application)

Click on Run button and copy/paste the trace into the Eclipse console. The decoded trace is dumped and the line corresponding to the application hook is now readable.

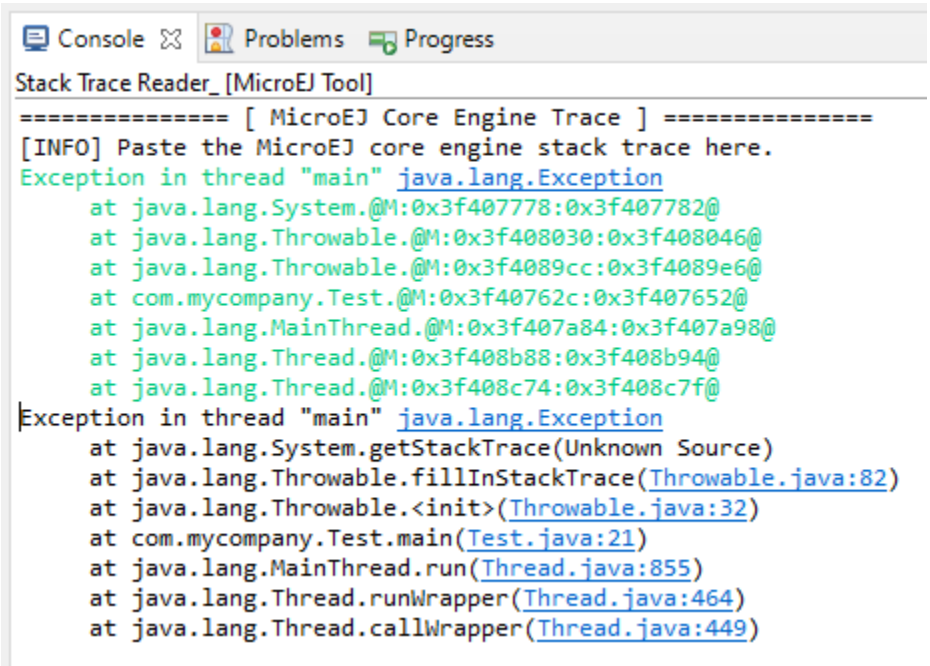


Fig. 372: Stack Trace Reader Console

Use (Sandboxed Application)

For example, write the following new line to dump the currently executed stack trace on the standard output.

```
public class MyBackgroundCode implements BackgroundService {

    @Override
    public void onStart() {
        // TODO Auto-generated method stub
        System.out.println("MyBackgroundCode: Hello World");
        new Throwable().printStackTrace();
    }
```

Fig. 373: Code to Dump a Stack Trace

To decode an application stack trace, the Stack Trace Reader tool requires the application binary file with debug information (application.fodbg in the output folder). Note that the file uploaded on the device is application.fo (stripped version without debug information).

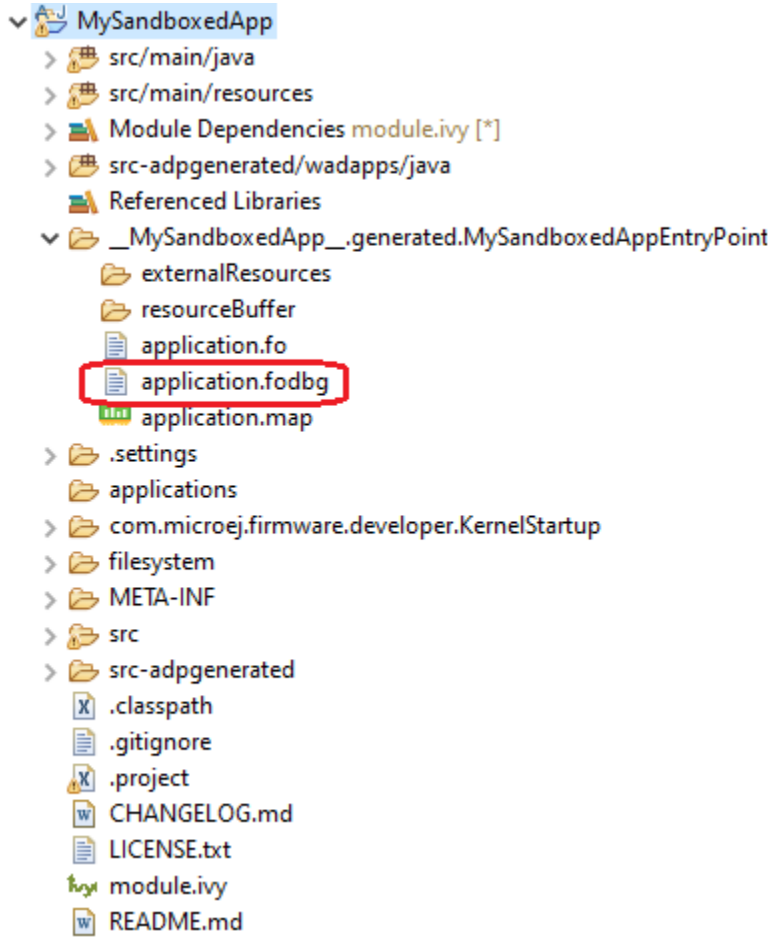


Fig. 374: Application Binary File with Debug Information

On successful deployment, the application is started on the device and the following trace is dumped on standard output.

```
com.microej.wadapps.kf.abstractfeatureapplicationstorage INFO: Start MySandboxedApp
MyBackgroundCode: Hello World
Exception in thread "ej.wadapps.app.default" java.lang.Throwable
    at java.lang.System.@M:0x805a97c:0x805a98c@
    at java.lang.Throwable.@M:0x807b8e0:0x807b8f6@
    at java.lang.Throwable.@M:0x8076f4c:0x8076f68@
    at com.microej.example.MyBackgroundCode.@F:a5db2a4477010000d37548f1e20224d0b875cb968936fb41:0xc03800f0@@M:0xc0380b7c:0xc0380ba4@
    at Exception in thread "ej.wadapps.app.default" java/lang/Throwable
   at java/lang/System.@M:0x0805A97C:0x0805A98C@
   at java/lang/Throwable.@M:0x0807B8E0:0x0807B8F6@
   at java/lang/Throwable.@M:0x08076F4C:0x08076F68@
   at com/microej/example/MyBackgroundCode.@F:a5db2a4477010000d37548f1e20224d0b875cb968936fb41:0xC03800F0@@M:0xC0380B7C:0xC0380BA4@
   at ej/wadapps/app/BackgroundServiceProxy.@F:fa7a45517201000073783c876987b55b8e3aaa8e1d407fd1:0x900A6BC0@@M:0x900AB508:0x900AB515@
   at com/microej/wadapps/management/util/BackgroundsManager.@F:fa7a45517201000073783c876987b55b8e3aaa8e1d407fd1:0x900A6BC0@@M:0x900AA780:0x900AA792@
   at com/microej/wadapps/management/util/BackgroundsManager.@F:fa7a45517201000073783c876987b55b8e3aaa8e1d407fd1:0x900A6BC0@@M:0x900ABF14:0x900ABF52@
   at ej/observable/Observable.@F:fa7a45517201000073783c876987b55b8e3aaa8e1d407fd1:0x900A6BC0@@M:0x900ABA10:0x900ABA40@
   at com/microej/wadapps/management/util/BackgroundServicesListImpl.@F:fa7a45517201000073783c876987b55b8e3aaa8e1d407fd1:0x900A6BC0@@M:0x900AD864:0x900AD894@
   at ej/wadapps/management/BackgroundServicesListProxy.@F:a5db2a4477010000d37548f1e20224d0b875cb968936fb41:0xC03800F0@@M:0xC0380A28:0xC0380A36@
   at __MySandboxedApp__/generated/MySandboxedAppActivator.@F:a5db2a4477010000d37548f1e20224d0b875cb968936fb41:0xC03800F0@@M:0xC0380C54:0xC0380C82@
   at ej/components/registry/impl/AbstractRegistry.@M:0x08078E48:0x08078E72@
   at ej/components/registry/util/BundleRegistryHelper.@M:0x0806E6E8:0x0806E702@
   at __MySandboxedApp__/generated/MySandboxedAppEntryPoint.@F:a5db2a4477010000d37548f1e20224d0b875cb968936fb41:0xC03800F0@@M:0xC0380B04:0xC0380B2E@
   at ej/kf/Kernel$2.@M:0x08055858:0x08055890@
   at java/lang/Thread.@M:0x0807C4F0:0x0807C506@
   at java/lang/Thread.@M:0x0807C398:0x0807C3A4@
   at java/lang/Thread.@M:0x0807C488:0x0807C493@
```

Fig. 375: Stack Trace Output

To create a new MicroEJ Tool configuration, right-click on the application project and click on Run As… > Run Configurations… .

Create a new MicroEJ Tool configuration. In the Execution tab, select your target platform, then select the Stack Trace Reader tool. Set an output folder in the Output folder field.

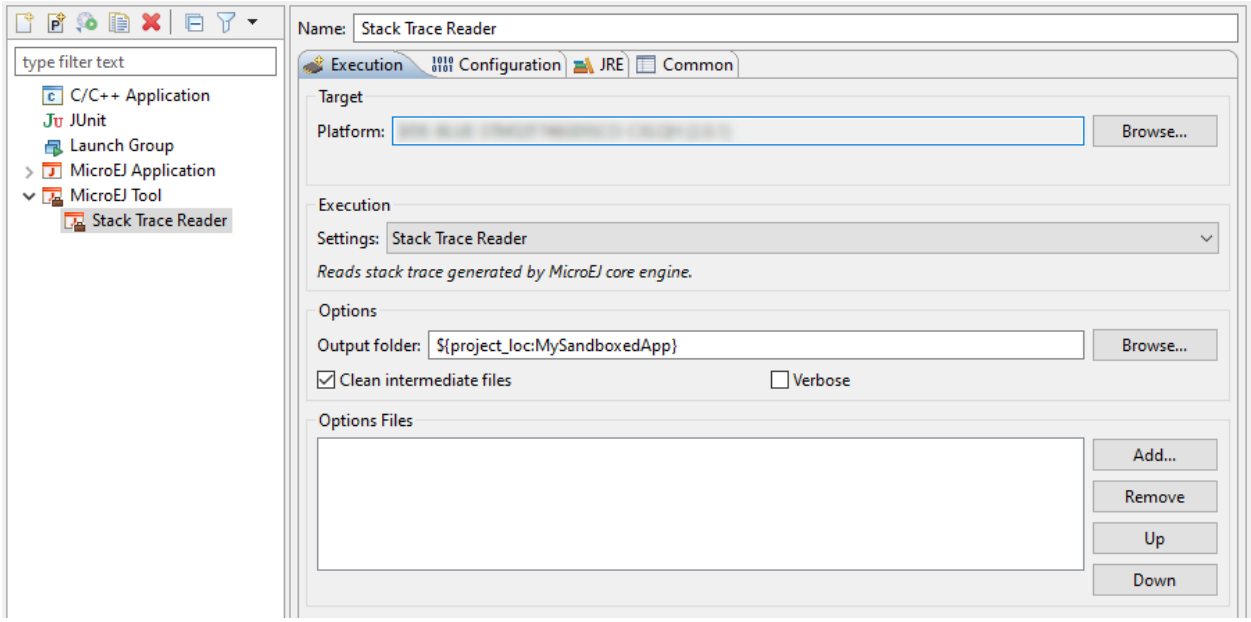


Fig. 376: Stack Trace Reader Tool Configuration (Virtual Device Selection)

In the Configuration tab, if the Kernel executable file is available to you (usually named firmware.out and located in your Virtual Device files), you can browse for it in

the Executable file field, and then add your previously generated application binary file with debug information (application.fodbg in case of a Sandboxed Application) in the Additional object files field.

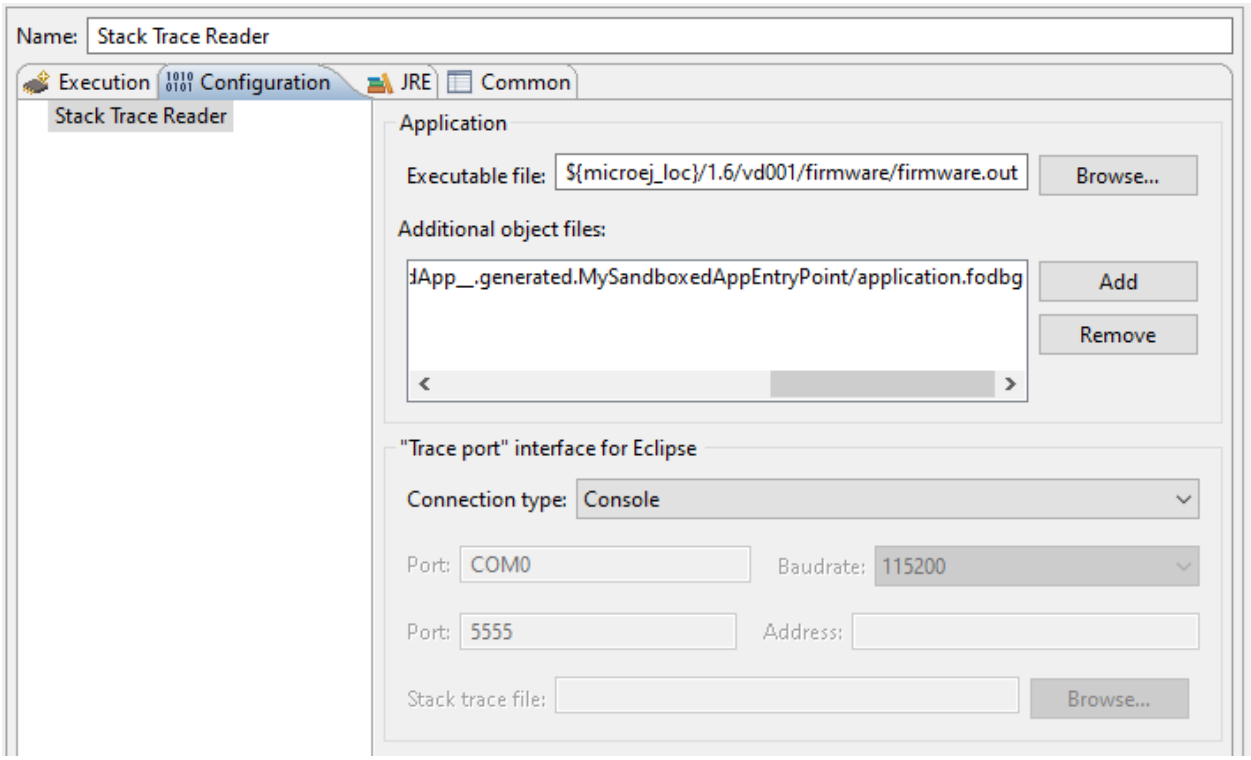


Fig. 377: Select the Kernel Executable File

To check where the Kernel executable file of your Virtual Device is located, if you have access to it, go to Window > Preferences > MicroEJ > Virtual Devices , hover over your Virtual Device in the list and wait until an information popup appears. Press F2 to get all the informations and the path to the directory of your Virtual Device should appear in the list.

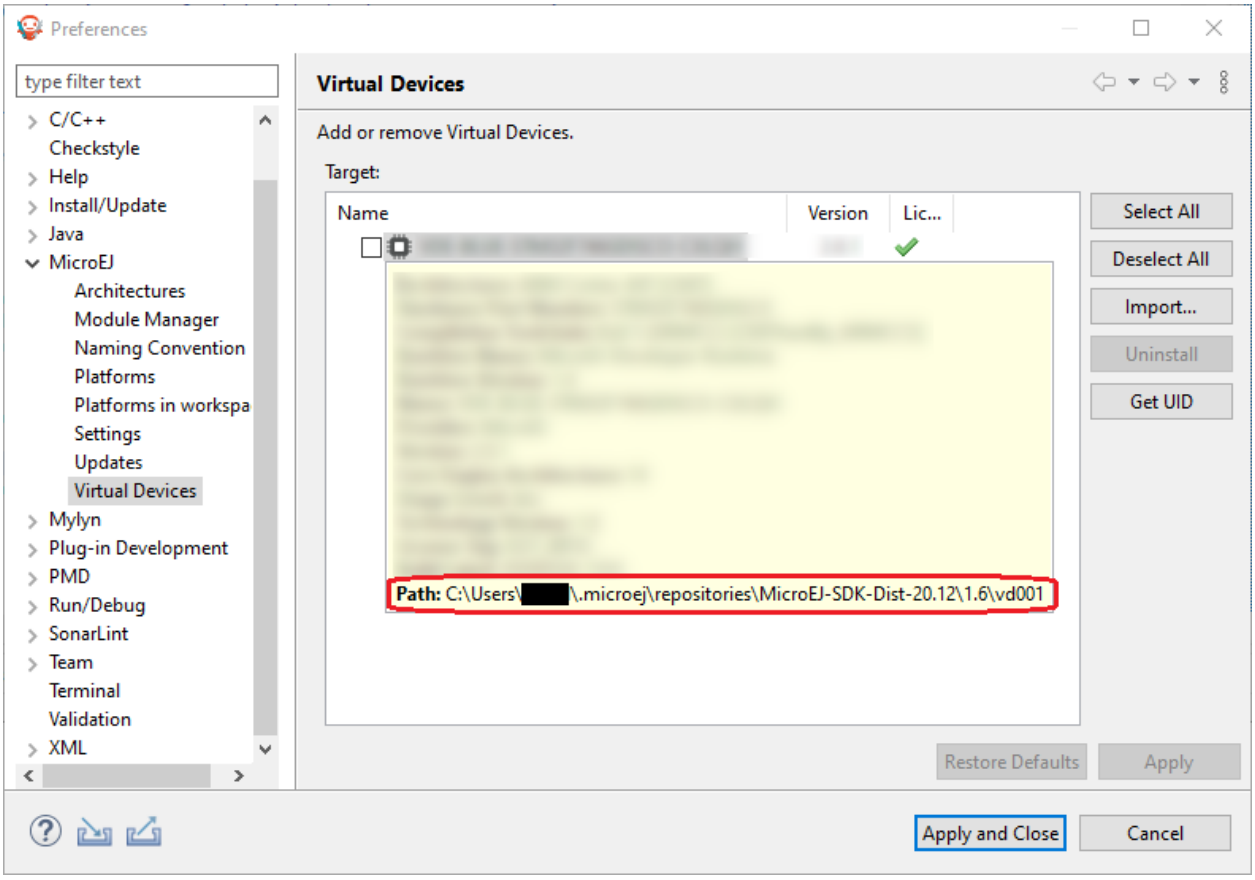


Fig. 378: Location of the Virtual Device Directory

In this directory, the Kernel executable file should be named `firmware.out` in the `/firmware` sub-directory.

If you do not have access to the Kernel executable file, you can still get some information from the Stack Trace Reader using the application binary file only. In the Configuration tab, browse the previously generated application binary file with debug information (`application.fodbg` in case of a Sandboxed Application)

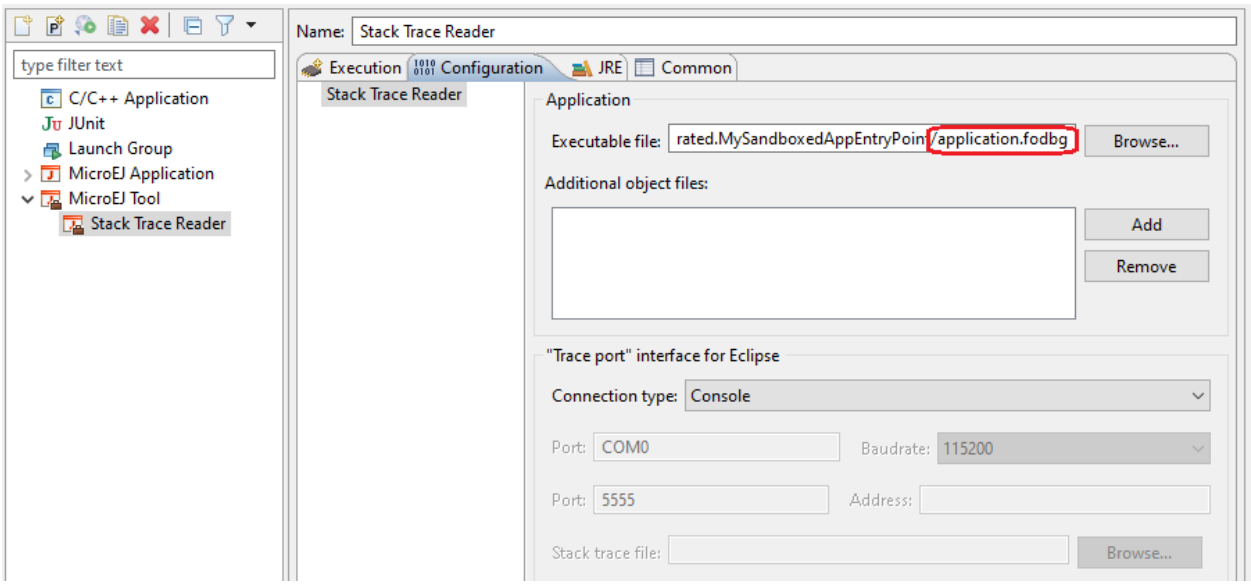


Fig. 379: Stack Trace Reader Tool Configuration (Sandboxed Application)

Click on Run button and copy/paste the trace into the Eclipse console. The decoded trace is dumped and the line corresponding to the application hook is now readable.

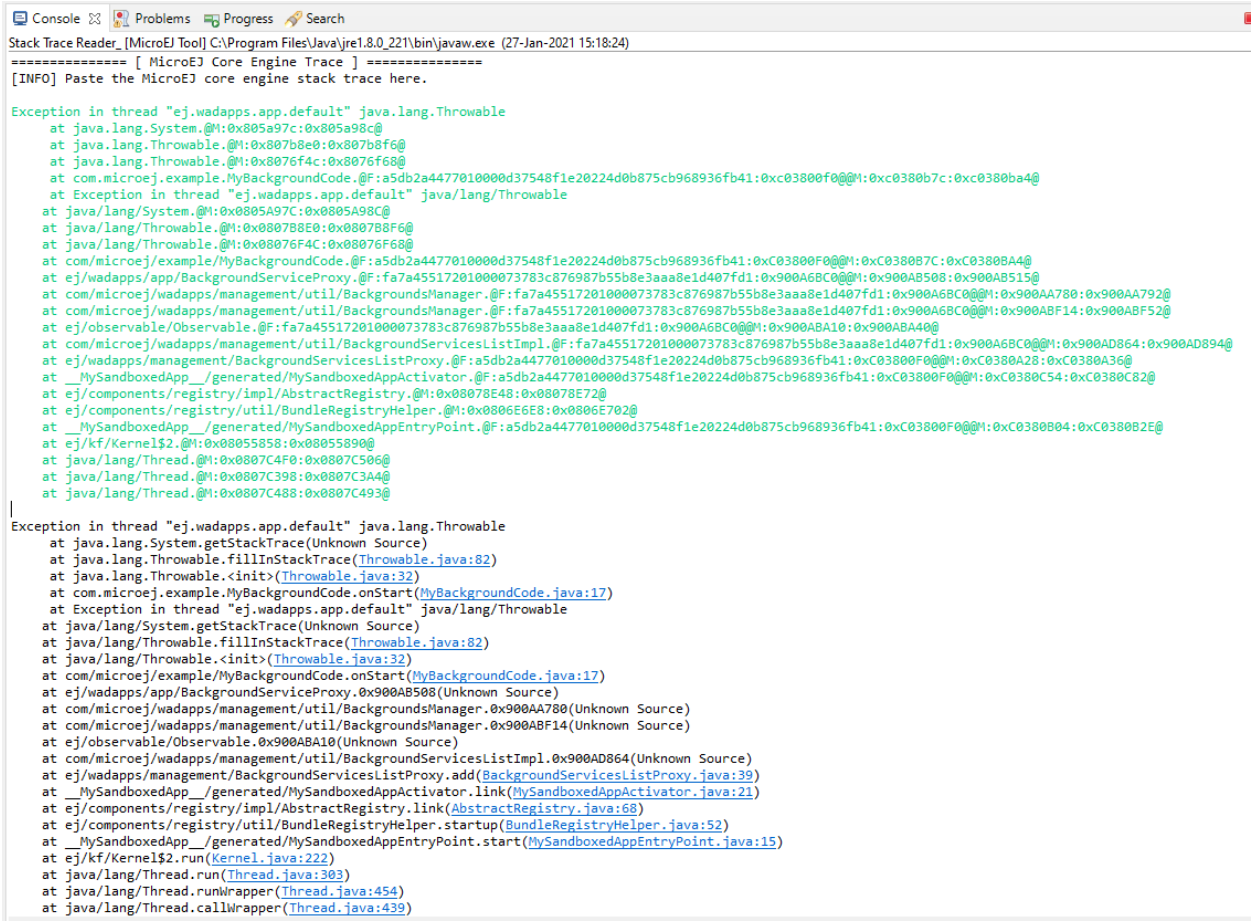


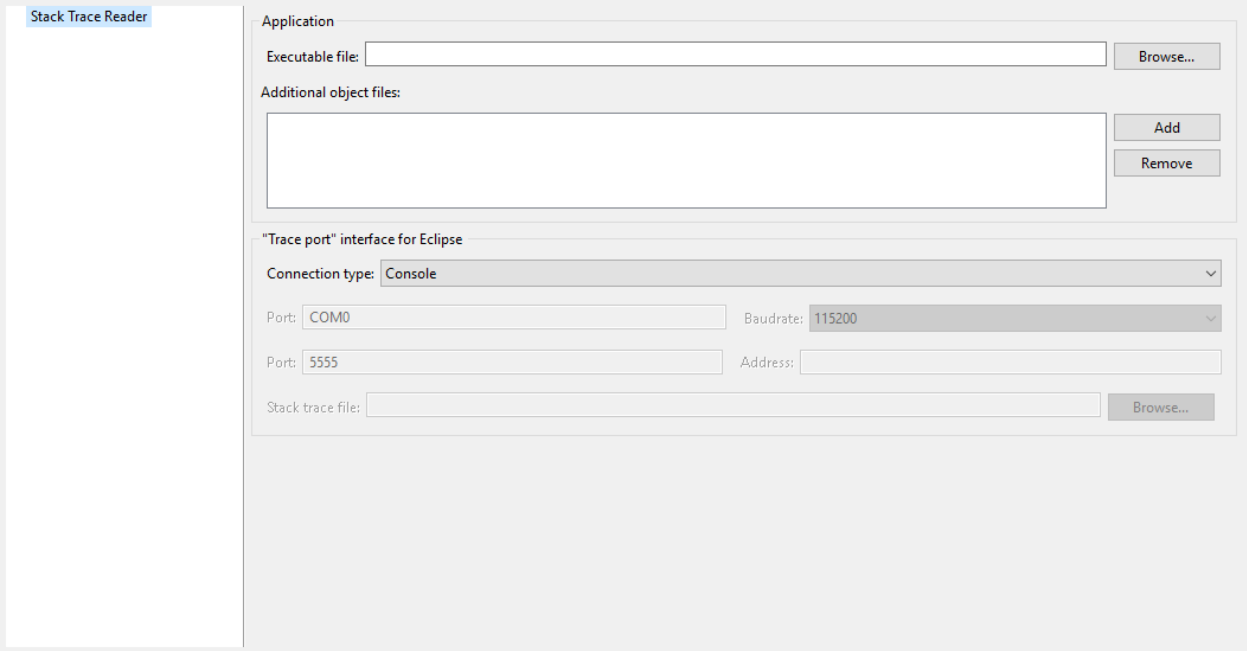
Fig. 380: Stack Trace Reader Console

Other debug information files can be appended using the Additional object files option.

Stack Trace Reader Options

The following section explains MicroEJ tool options.

Category: Stack Trace Reader



Group: Application

Option(browse): Executable file

Option Name: application.file

Default value: (empty)

Description:

Specify the full path of a full linked elf file.

Option(list): Additional object files

Option Name: additional.application.files

Default value: (empty)

Group: “Trace port” interface for Eclipse

Description:

This group describes the hardware link between the device and the PC.

Option(combo): Connection type

Option Name: proxy.connection.connection.type

Default value: Console

Available values:

Uart (COM)

Socket

File

Console

Description:

Specify the connection type between the device and PC.

Option(text): Port

Option Name: pcboardconnection.usart.pc.port

Default value: COM0

Description:

Format: port name

Specifies the PC COM port:

Windows - COM1 , COM2 , ... , COM*n*

Linux - /dev/ttyS0 , /dev/ttyS1 , ... , /dev/ttyS*n*

Option(combo): Baudrate

Option Name: pcboardconnection.usart.pc.baudrate

Default value: 115200

Available values:

9600

38400

57600

115200

Description:

Defines the COM baudrate for PC-Device communication.

Option(text): Port

Option Name: pcboardconnection.socket.port

Default value: 5555

Description:

IP port.

Option(text): Address

Option Name: pcboardconnection.socket.address

Default value: (empty)

Description:

IP address, on the form A.B.C.D.

Option(browse): Stack trace file

Option Name: pcboardconnection.file.path

Default value: (empty)

Code Coverage Analyzer

Principle

The Simulator features an option to output .cc (Code Coverage) files that represent the use rate of functions of an application. It traces how the opcodes are really executed.

Functional Description

The Code Coverage Analyzer scans the output .cc files, and outputs an HTML report to ease the analysis of methods coverage. The HTML report is available in a folder named htmlReport in the same folder as the .cc files generated by enabling the *Code Coverage option* .

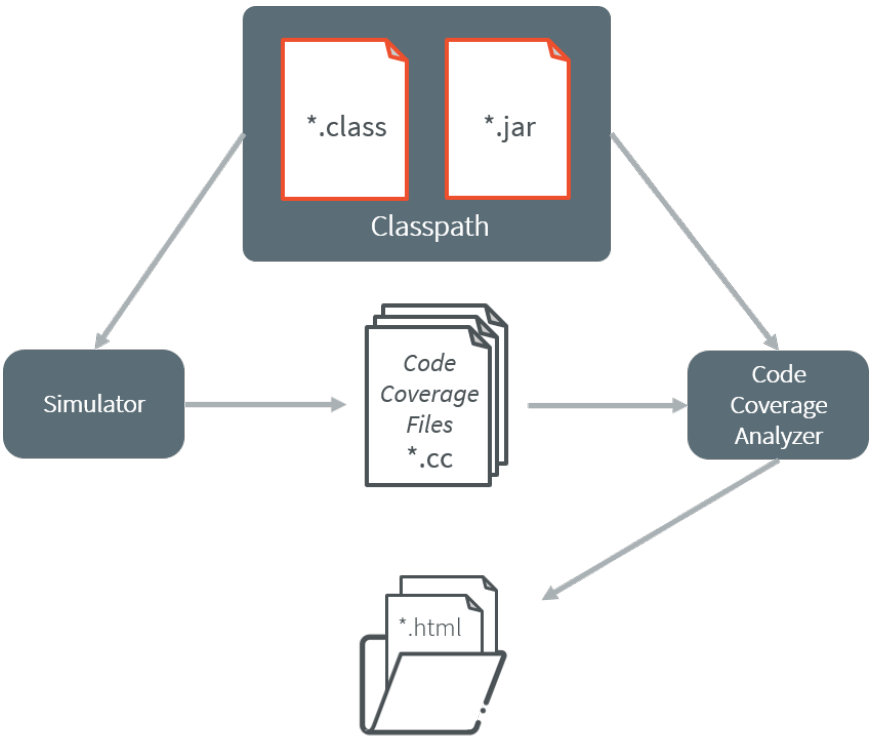


Fig. 381: Code Coverage Analyzer Process

Dependencies

In order to work properly, the Code Coverage Analyzer should input the .cc files. The .cc files relay the classpath used during the execution of the Simulator to the Code Coverage Analyzer. Therefore the classpath is considered to be a dependency of the Code Coverage Analyzer.

Installation

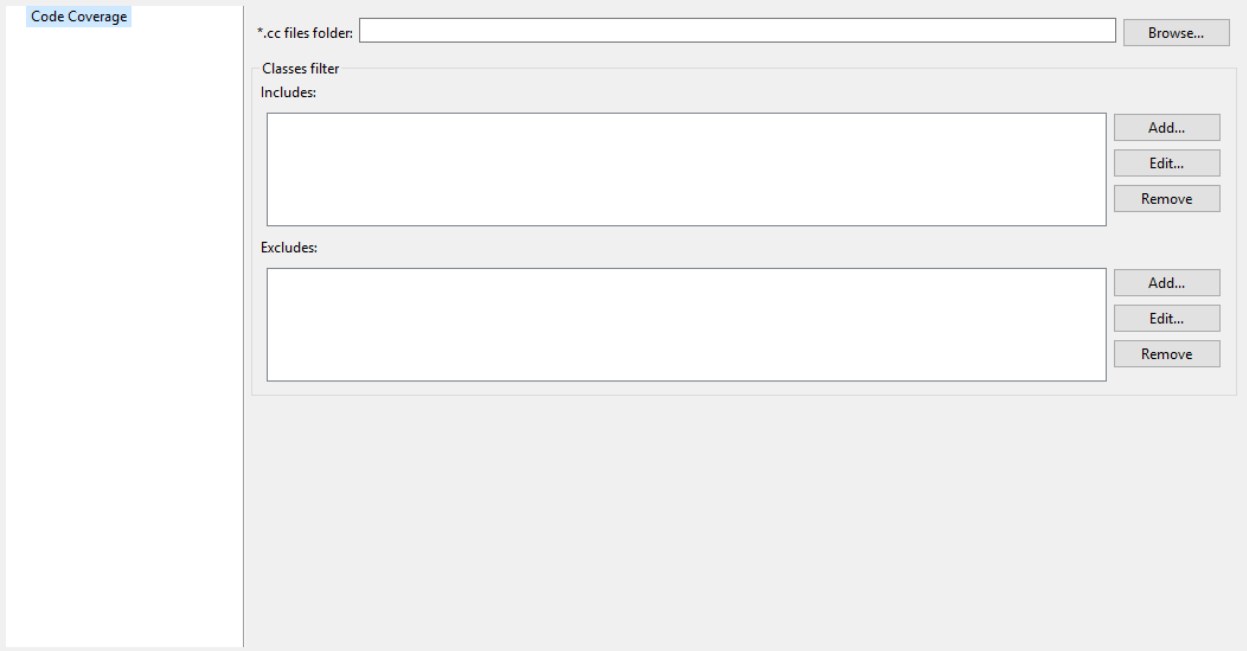
This tool is a built-in Architecture tool.

Use

A MicroEJ tool is available to launch the Code Coverage Analyzer tool. The tool name is Code Coverage Analyzer.

Two levels of code analysis are provided, the Java level and the bytecode level. Also provided is a view of the fully or partially covered classes and methods. From the HTML report index, just use hyperlinks to navigate into the report and source / bytecode level code.

Category: Code Coverage



Option(browse): *.cc files folder

Option Name: cc.dir

Default value: (empty)

Description:

Specify a folder which contains the cc files to process (*.cc).

Group: Classes filter

Option(list): Includes

Option Name: cc.includes

Default value: (empty)

Description:

List packages and classes to include to code coverage report. If no package/class is specified, all classes found in the project classpath will be analyzed.

Examples:

packageA.packageB.* : includes all classes which are in package packageA.packageB

packageA.packageB.className : includes the class packageA.packageB.className

Option(list): Excludes

Option Name: cc.excludes

Default value: (empty)

Description:

List packages and classes to exclude to code coverage report. If no package/class is specified, all classes found in the project classpath will be analyzed.

Examples:

packageA.packageB.* : excludes all classes which are in package packageA.packageB

packageA.packageB.className : excludes the class packageA.packageB.className

Heap Dumper & Heap Analyzer

Introduction

Heap Dumper is a tool that takes a snapshot of the heap. Generated files (with the .heap extension) are available in the application output folder.

The Heap Analyzer is a set of tools to help developers understand the contents of the Managed Heap and find problems such as memory leaks. For its part, the Heap Analyzer plugin is able to open dump files. It helps you analyze their contents thanks to the following features:

- memory leaks detection
- objects instances browse
- heap usage optimization (using immortal or immutable objects)

The Heap

The heap is a memory area used to hold Java objects created at runtime. Objects persist in the heap until they are garbage collected. An object becomes eligible for garbage collection when there are no longer any references to it from other objects.

Heap Dump

A heap dump is an XML file that provides a snapshot of the heap contents at the moment the file is created. It contains a list of all the instances of both class and array types that exist in the heap. For each instance, it records:

- The time at which the instance was created
- The thread that created it
- The method that created it

For instances of class types, it also records:

- The class
- The values in the instance's non-static fields

For instances of array types, it also records:

- The type of the contents of the array
- The contents of the array

  For each referenced class type, it records the values in the static fields of the class.

Heap Analyzer Tools

The Heap Analyzer is an Eclipse plugin that adds three tools to the MicroEJ environment.

Tool name	Number of input files	Purpose
Heap Viewer	1	Shows what instances are in the heap, when they were created, and attempts to identify problem areas
Progressive Heap Usage	1 or more	Shows how the number of instances in the heap has changed over time
Compare	2	Compares two heap dumps, showing which objects were created, or garbage collected, or have changed values

Heap Dumper

The Heap Dumper generates `.heap` files. There are two implementations:

1. **Integrated with the Simulator**: Dumps `.heap` files directly from the Managed Heap.
2. **Heap Dumper Tool**: Generates `.heap` files from `.hex` files, which must be retrieved from the device using tools like GDB.

   Heap dumps should be performed after a call to System.gc() to exclude discardable objects.

Simulator

To generate a Heap dump of an application running on the Simulator:

1. Set the `s3.inspect.heap` application property to `true`.
2. Update your application code to call the System.gc() method where you need a Heap dump.
3. Run the application on the Simulator.

   When the System.gc() method is called:

- If called from the application, the `.heap` file is generated in the `build/output/<fqnMainClass>/heapDump/` folder of the project, where `<fqnMainClass>` is the Fully Qualified Name of the application's main class, e.g., `com.mycompany.Main`.
- If called from a test class, the `.heap` file is generated in the `build/testsuite/output/<buildDate>/bin/<fqnMainClass>/heapDump/` folder of the project, where `<fqnMainClass>` is the Fully Qualified Name of the generated main class and `<buildDate>` is the test execution date, e.g., `build/testsuite/output/20240625-1605-24/bin/com.mycompany._AllTests_MyTest/heapDump/`.

Device

To generate a Heap dump of an application running on a device:

1. Update your application code to call the System.gc() method where you need a Heap dump.
2. Build the executable and deploy it on the device.
3. Start a debug session.
4. Add a breakpoint to the `LLMJVM_on_Runtime_gc_done` Core Engine hook. This function is called by the Core Engine when the System.gc() method is done. Alternatively, for out of memory errors, add a breakpoint to the `LLMJVM_on_OutOfMemoryError_thrown` Core Engine hook.
5. Resume execution until the breakpoint is reached. You are now ready to dump the memory.

Note: Core Engine hooks may be inlined by the third-party linker. If the symbol is not accessible to the debugger, declare them in your VEE Port:

```
void LLMJVM_on_Runtime_gc_done(){
  //No need to add code to the function
}

void LLMJVM_on_OutOfMemoryError_thrown(){
  //No need to add code to the function
}
```

Retrieve the `.hex` file from the device

If you are in a Mono-Sandbox context and do not use *dynamic Heap configuration*, you only need to dump the Core Engine heap section. Example GDB commands:

```
b LLMJVM_on_Runtime_gc_done
b LLMJVM_on_OutOfMemoryError_thrown
continue
dump ihex memory heap.hex &_java_heap_start &_java_heap_end
```

You now have the `.hex` file and need to extract the Heap dump.

- Otherwise, dump all MICROEJ VEE memories. See *VEE memory dump* for how to generate a script for your debugger that dumps all required sections.

  You then have to merge all `.hex` files into a single one for the next step.

Convert .hex dump to .heap dump

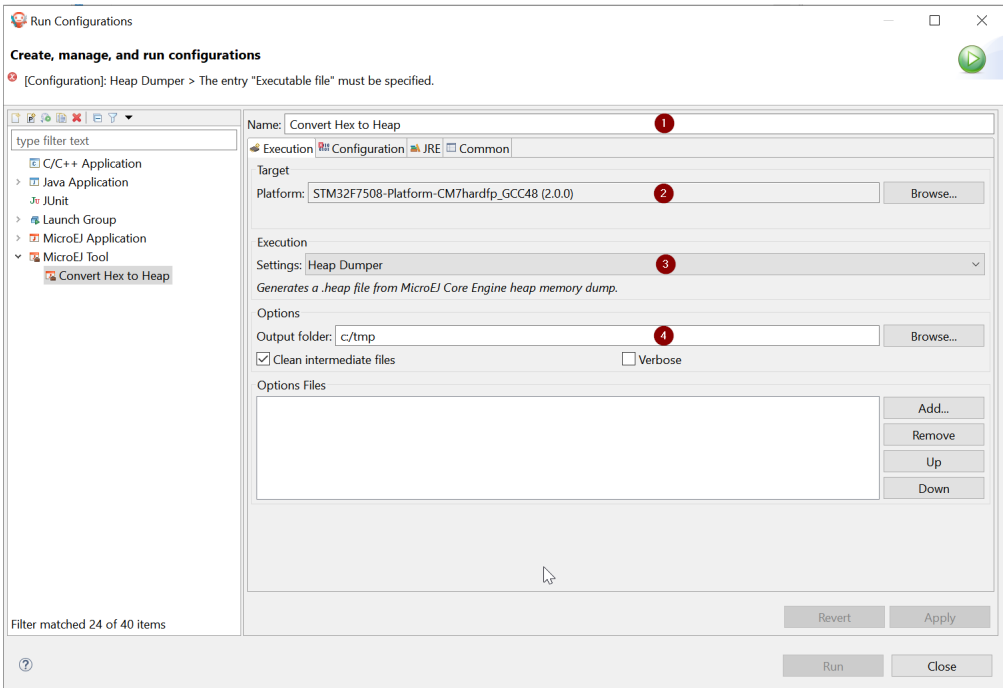
To convert the Heap dump from .hex to .heap, use the Heap Dumper tool.

The Heap Dumper should be available in your VEE Port and can be configured and executed from the SDK Run Configurations.

From the SDK top menu, go to Run > Run Configurations...

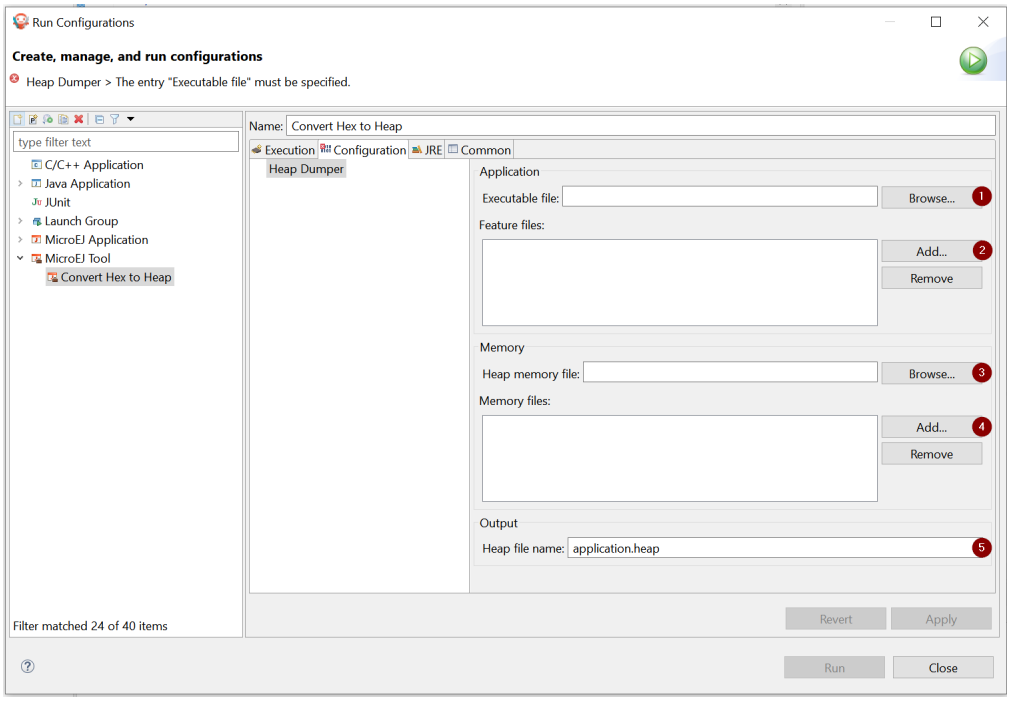
1. Right-click on MicroEJ Tool and select New Configuration.
2. Configure the Execution tab:

1. Set the tool name, e.g., Convert Hex to Heap.
2. Select the platform in the Target > Platform field.
3. Select the Heap Dumper tool from the Execution > Settings list.
4. Set the Output folder path, where the .heap file will be generated.



3. Switch to the Configuration tab and configure it:

1. Set the path to the firmware executable ELF file.
2. Add the full path of application files with debug information (.fodbg files).
3. Set the full path of the heap memory dump, in Intel Hex format.
4. Add the full path of additional memory files in Intel Hex format (installed feature areas, dynamic features table, etc.).
5. Set the heap file output name, e.g., application.heap.



4. Finally, click Apply and Run .

   Now you can open the generated .heap file in the Heap Viewer.

Heap Viewer

To open the Heap Viewer tool, select a heap dump XML file in the Package Explorer , right-click on it and select Open With > Heap Viewer

Alternatively, right-click on it and select Heap Analyzer > Open heap viewer

This will open a Heap Viewer tool window for the selected heap dump[1].

The Heap Viewer works in conjunction with two views:

1. The Outline view
2. The Instance Browser view

   These views are described below.

   The Heap Viewer tool has three tabs, each described below.

[1] Although this is an Eclipse editor, it is not possible to edit the contents of the heap dump.

Outline View

The Outline view shows a list of all the types in the heap dump, and for each type shows a list of the instances of that type. When an instance is selected it also shows a list of the instances that refer to that instance. The Outline view is opened automatically when an Heap Viewer is opened.

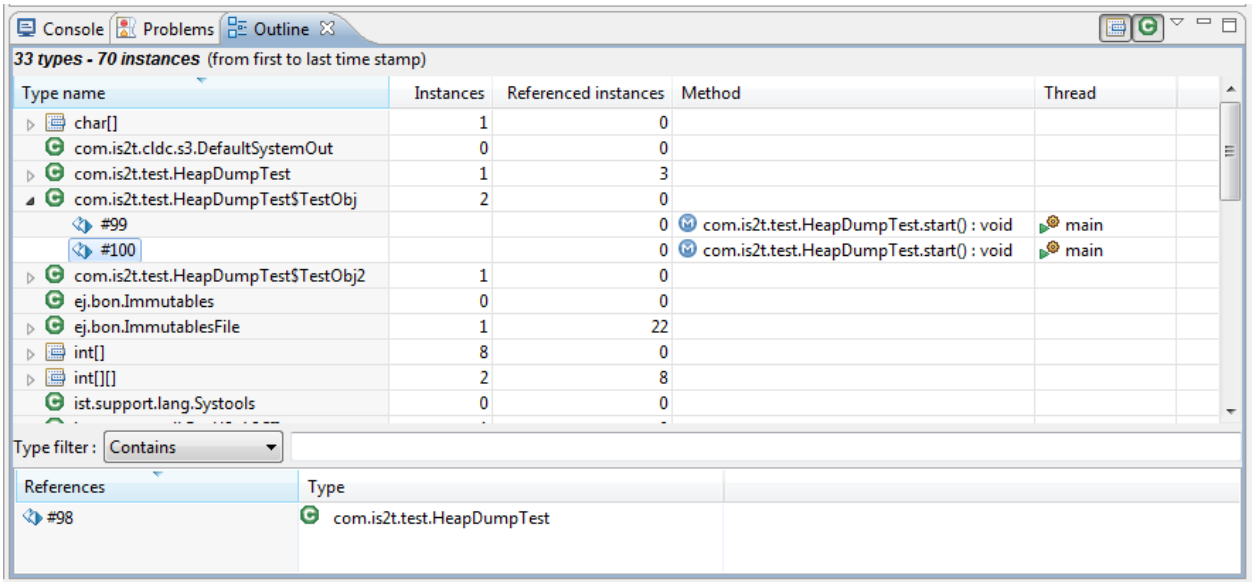


Fig. 382: Outline View

Instance Browser View

The Instance Browser view opens automatically when a type or instance is selected in the Outline view. It has two modes, selected using the buttons in the top right corner of the view. In 'Fields' mode it shows the field values for the selected type or instance, and where those fields hold references it shows the fields of the referenced instance, and so on. In 'Reference' mode it shows the instances that refer to the selected instance, and the instances that refer to them, and so on.

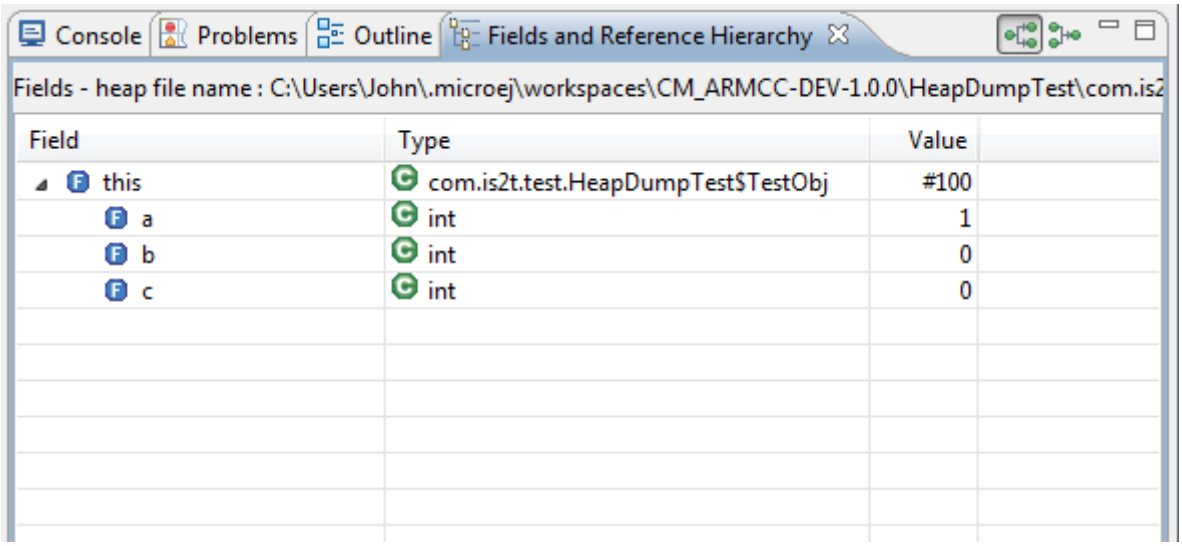


Fig. 383: Instance Browser View - Fields mode

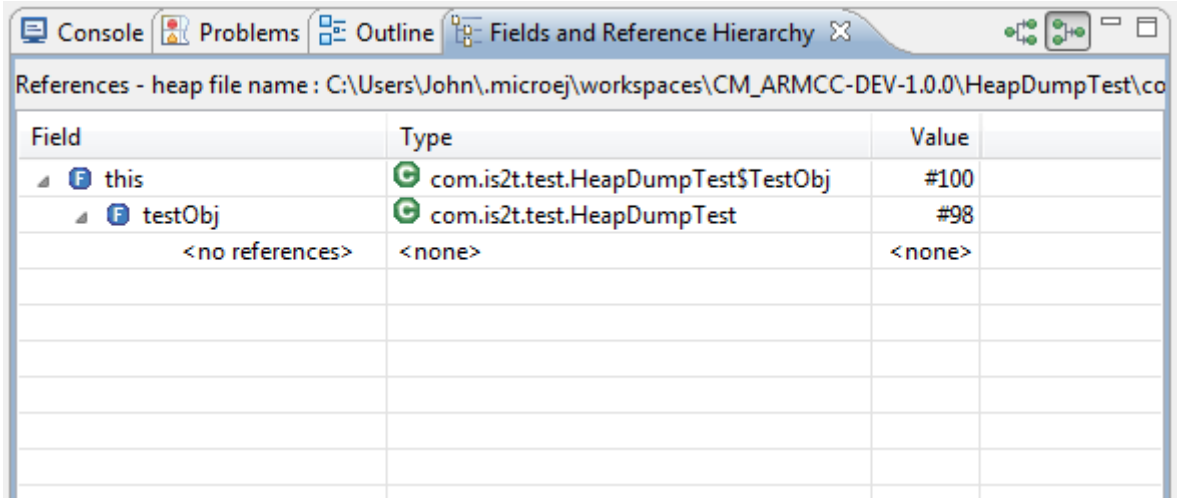


Fig. 384: Instance Browser View - References mode

Heap Usage Tab

The Heap usage page of the Heap Viewer displays four bar charts. Each chart divides the total time span of the heap dump (from the time stamp of the earliest instance creation to the time stamp of the latest instance creation) into a number of periods along the x axis, and shows, by means of a vertical bar, the number of instances created during the period.

- The top-left chart shows the total number of instances created in each period, and is the only chart displayed when the Heap Viewer is first opened.
- When a type or instance is selected in the Outline view the top-right chart is displayed. This chart shows the number of instances of the selected type created in each time period.
- When an instance is selected in the Outline view the bottom-left chart is displayed. This chart

shows the number of instances created in each time period by the thread that created the selected instance.

- When an instance is selected in the Outline view the bottom-right chart is displayed. This chart shows the number of instances created in each time period by the method that created the selected instance.

Fig. 385: Heap Viewer - Heap Usage Tab

Clicking on the graph area in a chart restricts the Outline view to just the types and instances that were created during the selected time period. Clicking on a chart but outside of the graph area restores the Outline view to showing all types and instances[2].

The button Generate graphViz file in the top-right corner of the Heap Usage page generates a file compatible with graphviz (www.graphviz.org).

The section *Heap Usage Monitoring* shows how to compute the maximum heap usage.

[2] The Outline can also be restored by selecting the All types and instances option on the drop-down menu at the top of the Outline view.

Dominator Tree Tab

The Dominator tree page of the Heap Viewer allows the user to browse the instance reference tree which contains the greatest number of instances. This can be useful when investigating a memory leak because this tree is likely to contain the instances that should have been garbage collected.

The page contains two tree viewers. The top viewer shows the instances that make up the tree, starting with the root. The left column shows the ids of the instances – initially just the root instance is shown. The Shallow instances column shows the number of instances directly referenced by the instance, and the Referenced instances column shows the total number of instances below this point in the tree (all descendants).

The bottom viewer groups the instances that make up the tree either according to their type, the thread that created them, or the method that created them.

Double-clicking an instance in either viewer opens the Instance Browser view (if not already open) and shows details of the instance in that view.

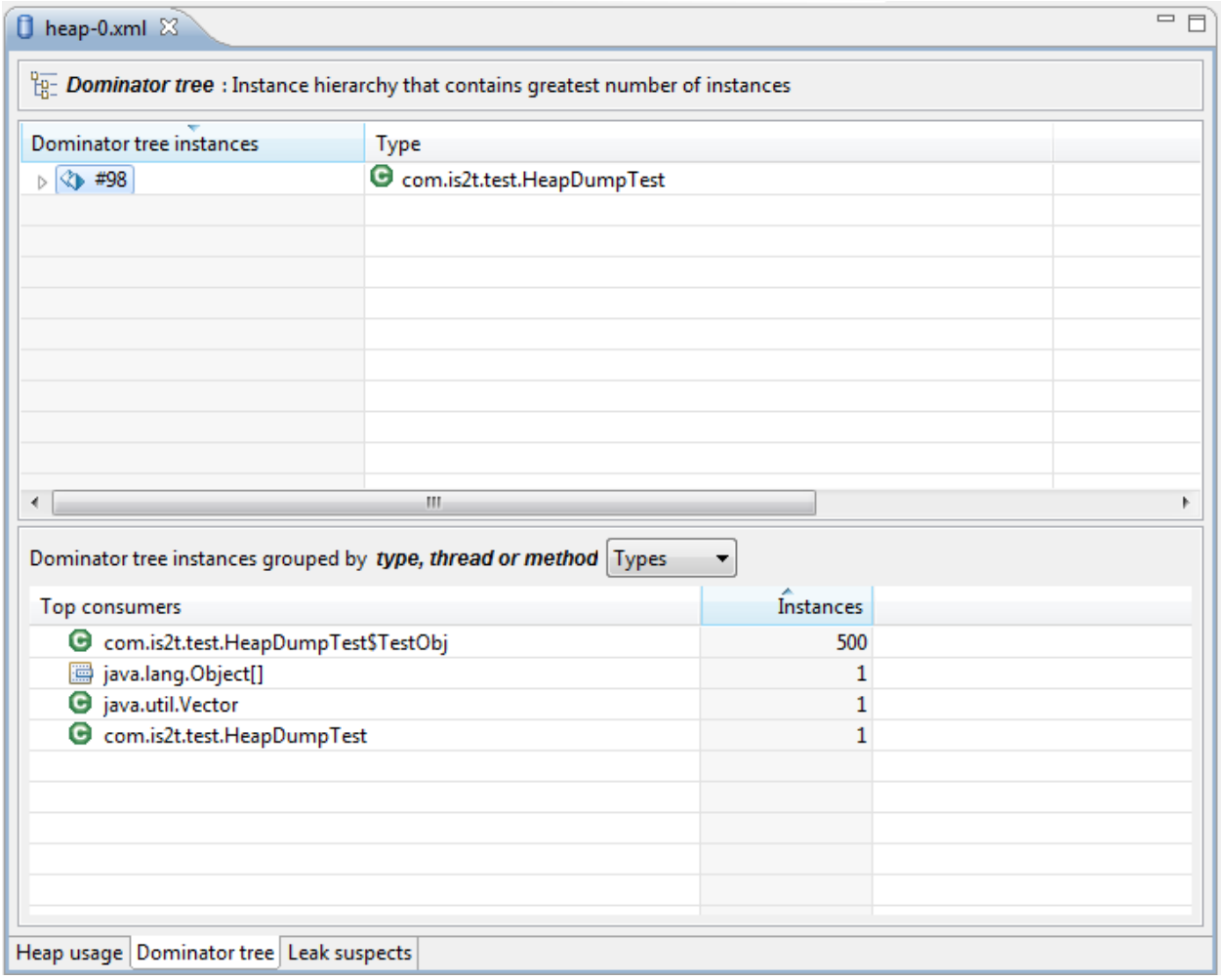


Fig. 386: Heap Viewer - Dominator Tree Tab

Leak Suspects Tab

The Leak suspects page of the Heap Viewer shows the result of applying heuristics to the relationships between instances in the heap to identify possible memory leaks.

The page is in three parts.

- The top part lists the suspected types (classes). Suspected types are classes which, based on numbers of instances and instance creation frequency, may be implicated in a memory leak.
- The middle part lists accumulation points. An accumulation point is an instance that references a high number of instances of a type that may be implicated in a memory leak.
- The bottom part lists the instances accumulated at an accumulation point.

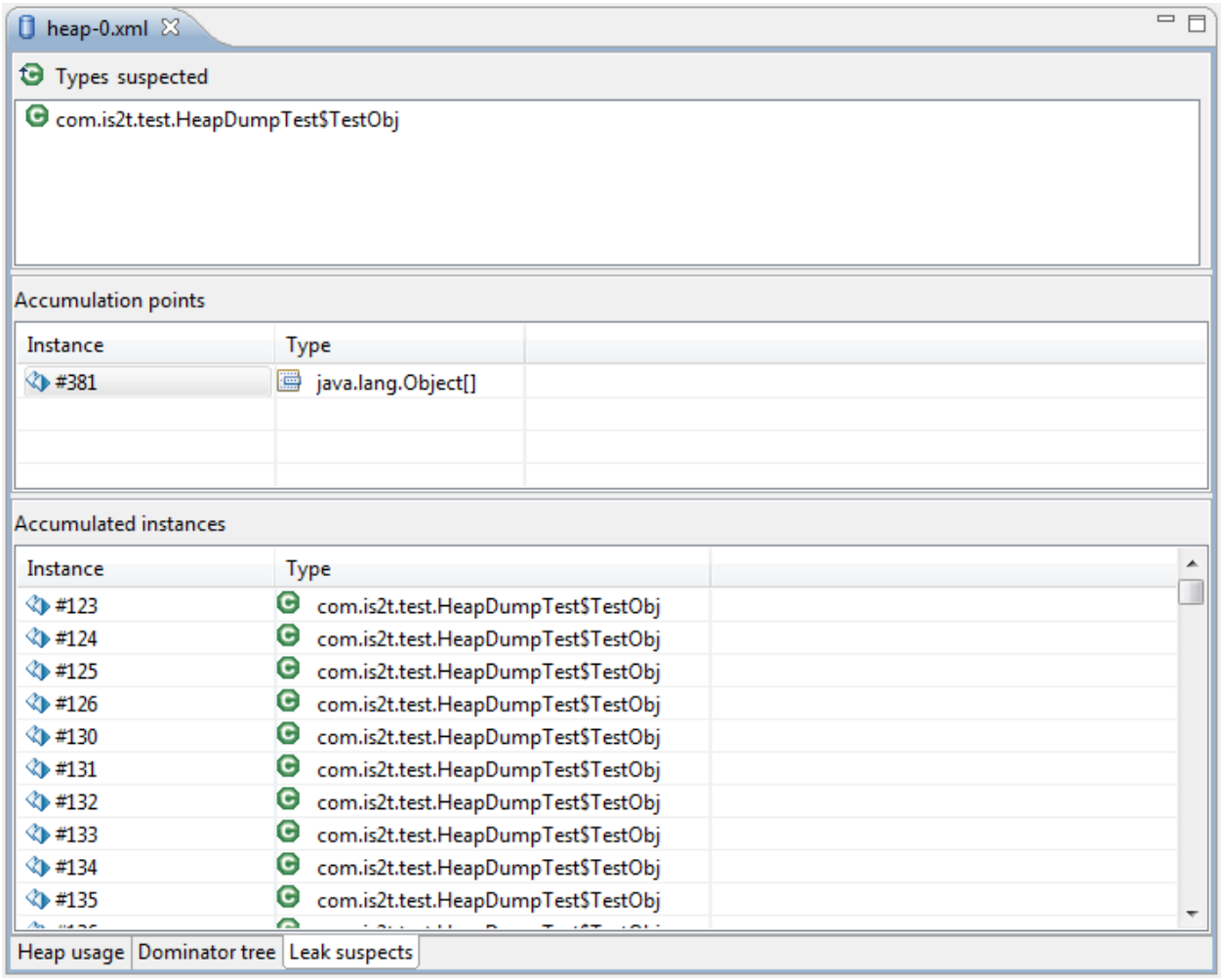


Fig. 387: Heap Viewer - Leak Suspects Tab

Progressive Heap Usage

To open the Progressive Heap Usage tool, select one or more heap dump XML files in the Package Explorer , right-click and select Heap Analyzer > Show progressive heap usage

This tool is much simpler than the Heap Viewer described above. It comprises three parts.

- The top-right part is a line graph showing the total number of instances in the heap over time, based on the creation times of the instances found in the heap dumps.
- The left part is a pane with three tabs, one showing a list of types in the heap dump, another a list of threads that created instances in the heap dump, and the third a list of methods that created instances in the heap dump.
- The bottom-left is a line graph showing the number of instances in the heap over time restricted to those instances that match with the selection in the left pane. If a type is selected, the graph shows only instances of that type; if a thread is selected the graph shows only instances created by that thread; if a method is selected the graph shows only instances created by that method.

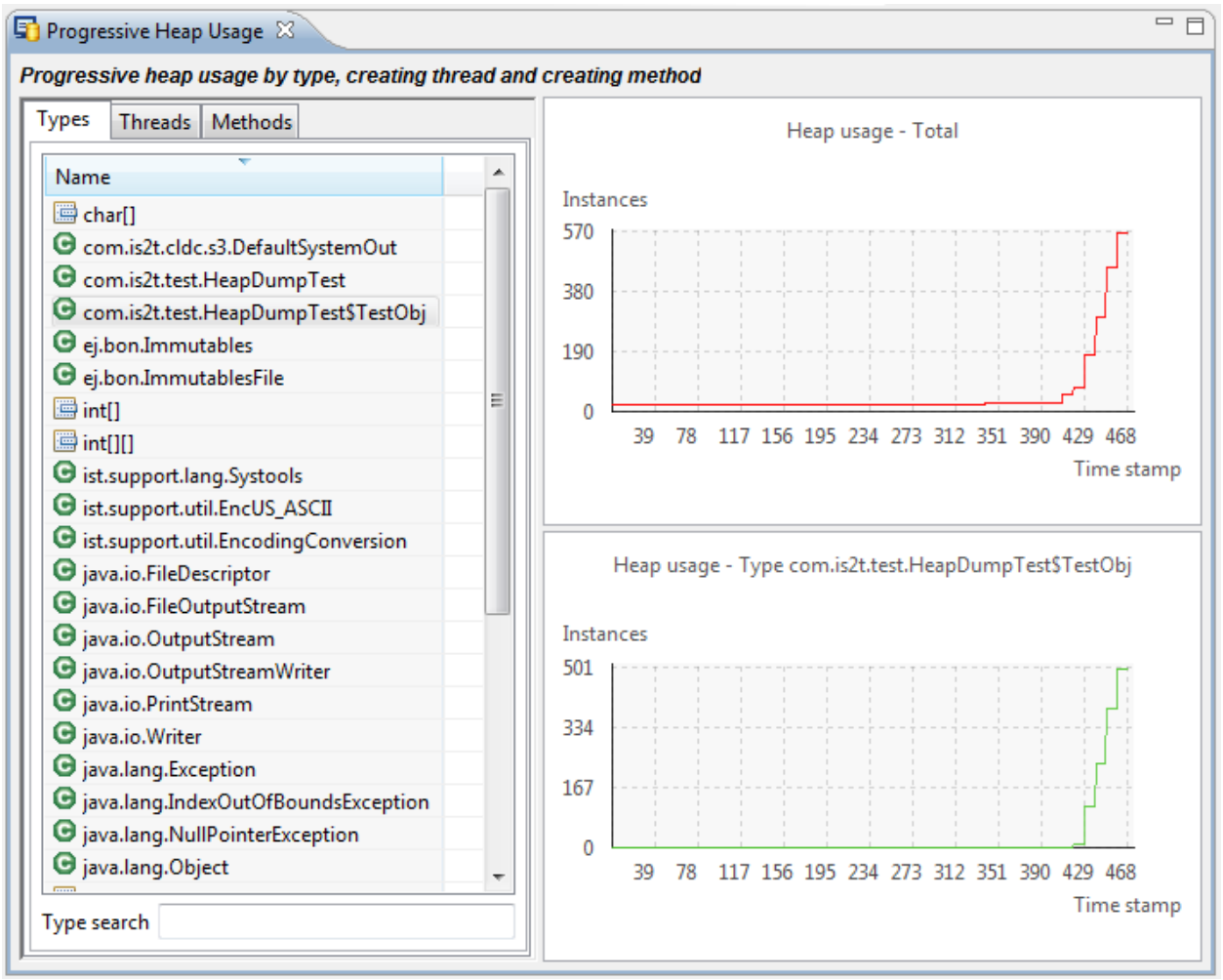


Fig. 388: Progressive Heap Usage

Compare Heap Dumps

The Compare tool compares the contents of two heap dump files. To open the tool select two heap dump XML files in the Package Explorer, right-click and select Heap Analyzer > Compare

The Compare tool shows the types in the old heap on the left-hand side, and the types in the new heap on the right-hand side, and marks the differences between them using different colors.

Types in the old heap dump are colored red if there are one or more instances of this type which are in the old dump but not in the new dump. The missing instances have been garbage collected.

Types in the new heap dump are colored green if there are one or more instances of this type which are in the new dump but not in the old dump. These instances were created after the old heap dump was written.

Clicking to the right of the type name unfolds the list to show the instances of the selected type.

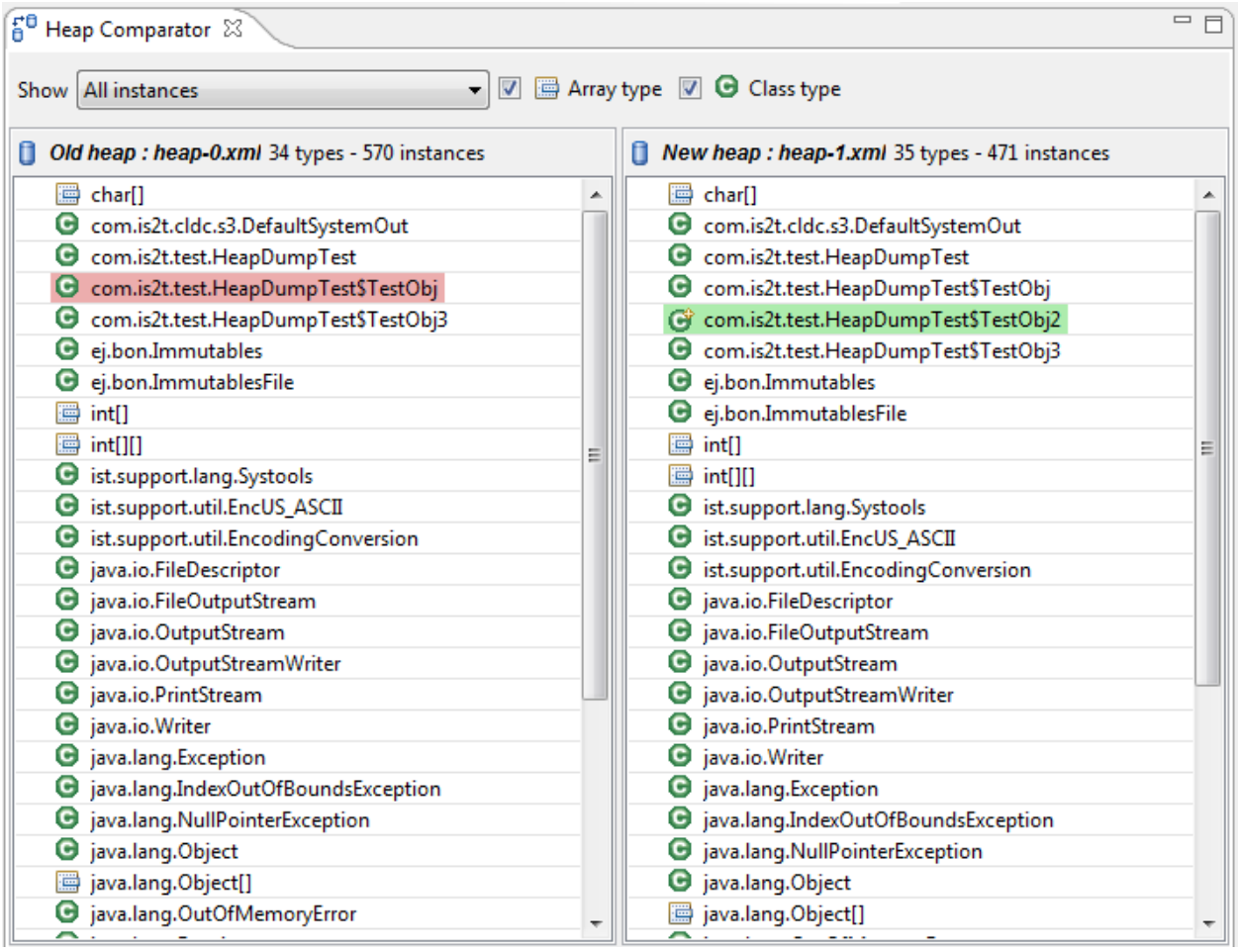


Fig. 389: Compare Heap Dumps

The combo box at the top of the tool allows the list to be restricted in various ways:

- All instances – no restriction.

- Garbage collected and new instances – show only the instances that exist in the old heap dump but not in the new dump, or which exist in the new heap dump but not in the old dump.
- Persistent instances – show only those instances that exist in both the old and new dumps.
- Persistent instances with value changed – show only those instances that exist in both the old and new dumps and have one or more differences in the values of their fields.

Instance Fields Comparison View

The Compare tool works in conjunction with the Instance Fields Comparison view, which opens automatically when an instance is selected in the tool.

The view shows the values of the fields of the instance in both the old and new heap dumps, and highlights any differences between the values.

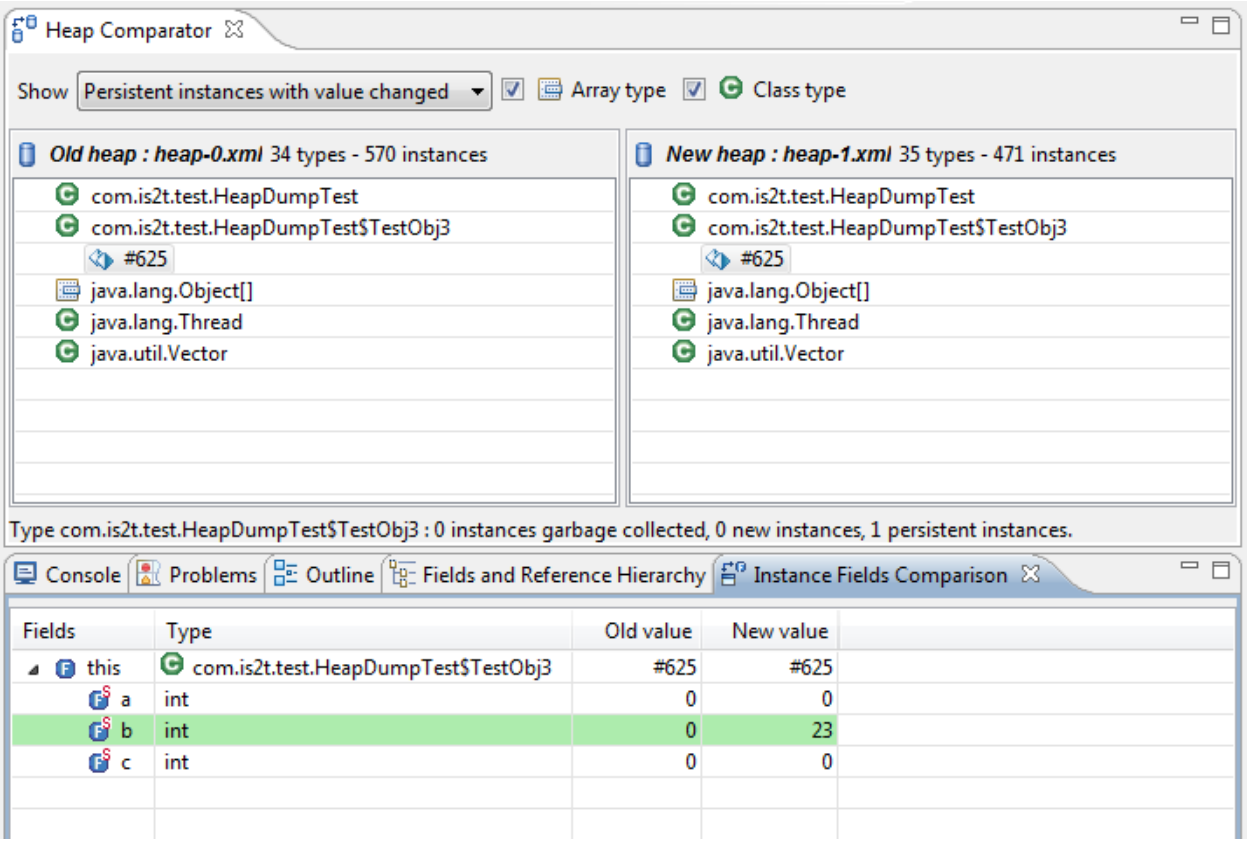


Fig. 390: Instance Fields Comparison view

Serial to Socket Transmitter

Principle

The MicroEJ serialToSocketTransmitter is a piece of software which transfers all bytes from a serial port to a tcp client or tcp server.

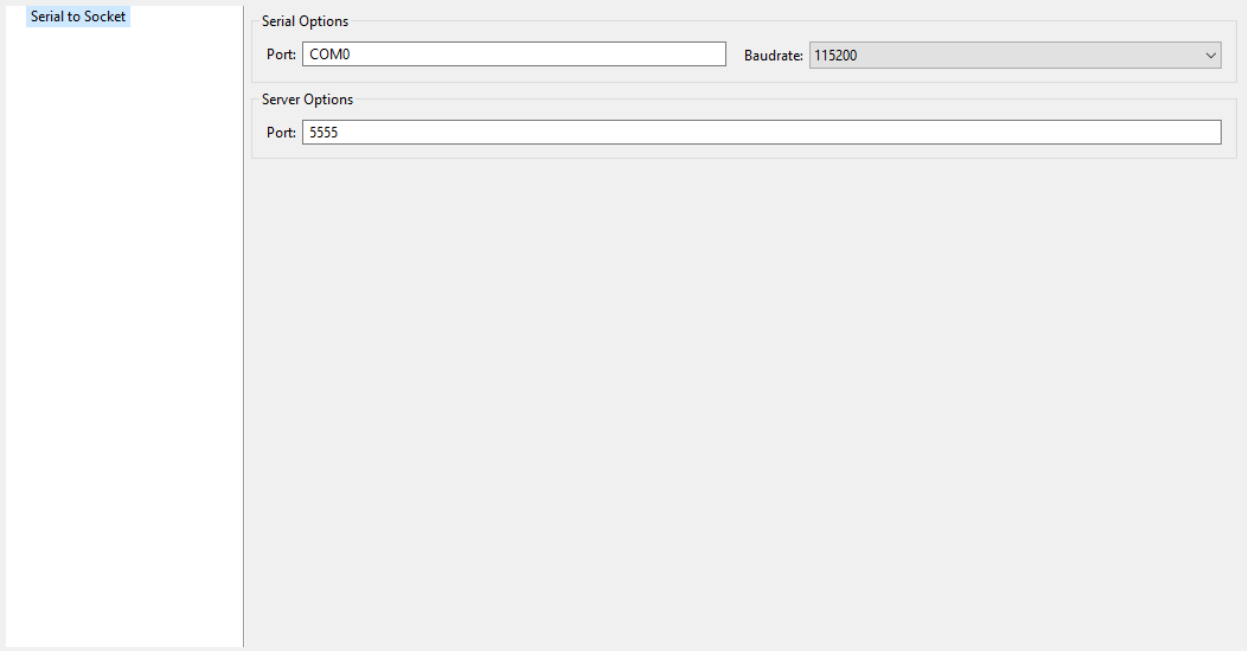
Installation

This tool is a built-in Architecture tool.

Use

This chapter explains MicroEJ tool options.

Category: Serial to Socket



Group: Serial Options

Option(text): Port

Option Name: serail.to.socket.comm.port

Default value: COM0

Description: Defines the COM port:

Windows - COM1 , COM2 , ... , COM*n*

Linux - /dev/ttyS0 , /dev/ttyUSB0 , ... , /dev/ttyS*n* , /dev/ttyUSB*n*

Option(combo): Baudrate

Option Name: serail.to.socket.comm.baudrate

Default value: 115200

Available values:

9600

38400

57600

115200

Description: Defines the COM baudrate.

Group: Server Options

Option(text): Port

Option Name: serail.to.socket.server.port

Default value: 5555

Description: Defines the server IP port.

Memory Map Analyzer

Principle

When a MicroEJ Application is linked with the MicroEJ Workbench, a Memory MAP file is generated. The Memory Map Analyzer (MMA) is an Eclipse plug-in made for exploring the map file. It displays the memory consumption of different features in the RAM and ROM.

Functional Description

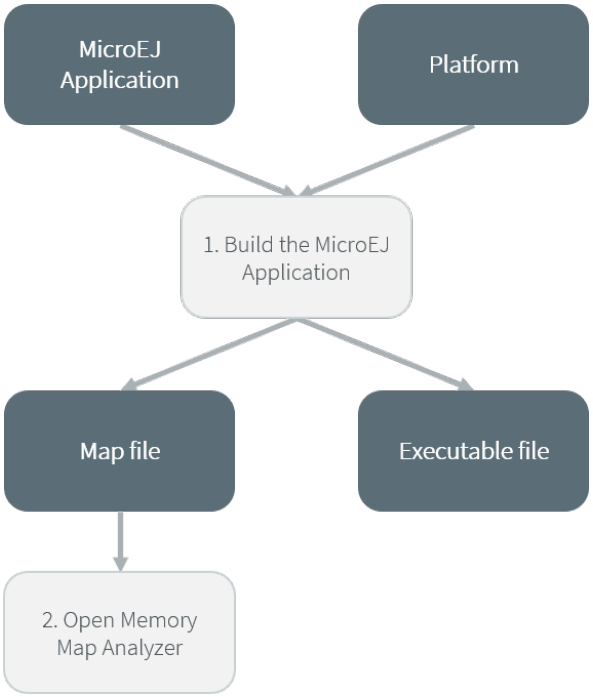


Fig. 391: Memory Map Analyzer Process

In addition to the executable file, the MicroEJ Platform generates a map file. Double click on this file to open the Memory Map Analyzer.

Dependencies

No dependency.

Installation

This tool is a built-in SDK tool.

Use

The map file is available in the MicroEJ Application project output directory.

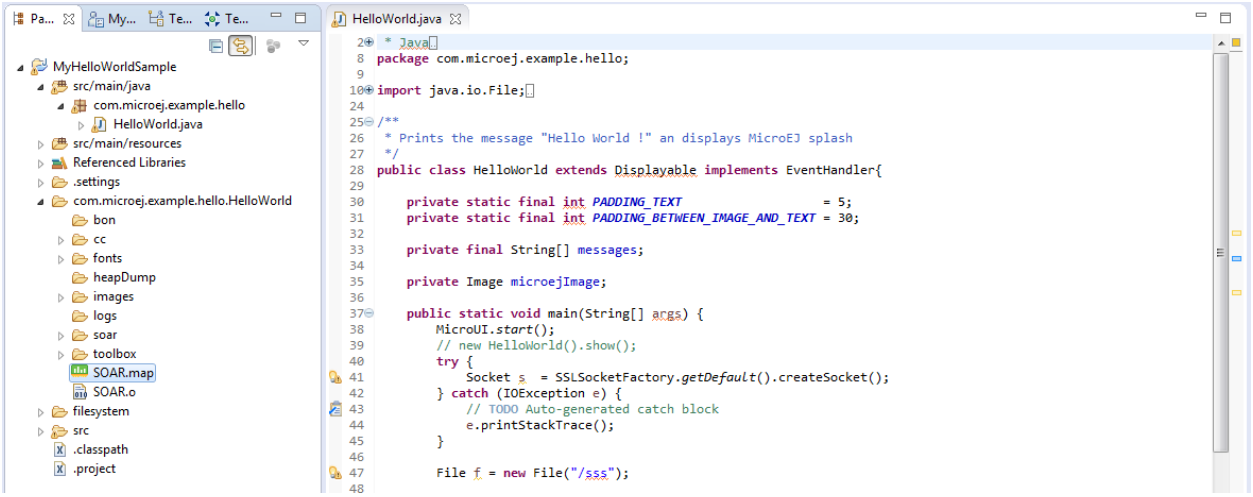


Fig. 392: Retrieve Map File

Select an item (or several) to show the memory used by this item(s) on the right. Select “All” to show the memory used by all items. This special item performs the same action as selecting all items in the list.

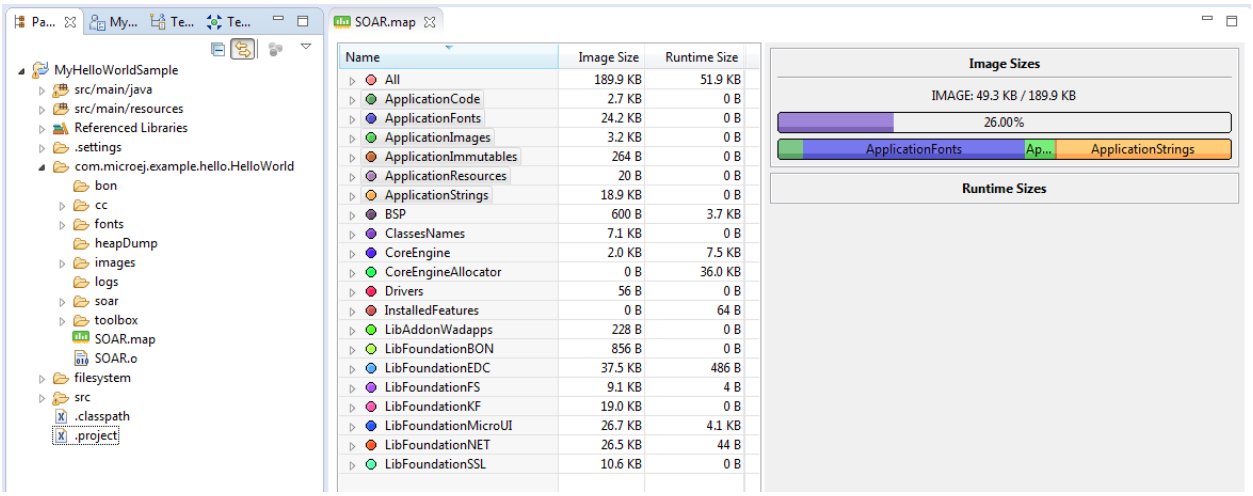


Fig. 393: Consult Full Memory

Select an item in the list, and expand it to see all symbols used by the item. This view is useful in understanding why a symbol is embedded.

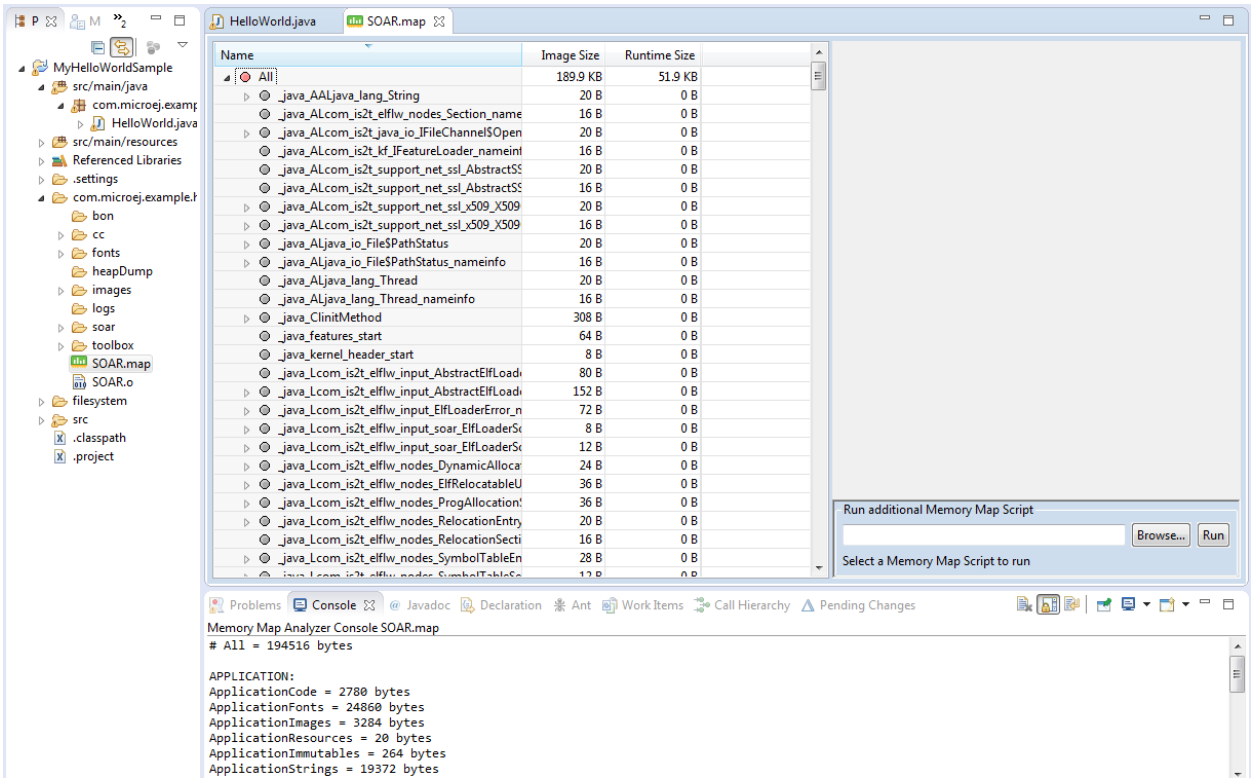


Fig. 394: Detailed View

Null Analysis

NullPointerException thrown at runtime is one of the most common causes for failure of Java programs. The Null Analysis tool can detect such programming errors (misuse of potential `null` Java values) at compile-time.

The following example of code shows a typical Null Analysis error detection in MicroEJ SDK.

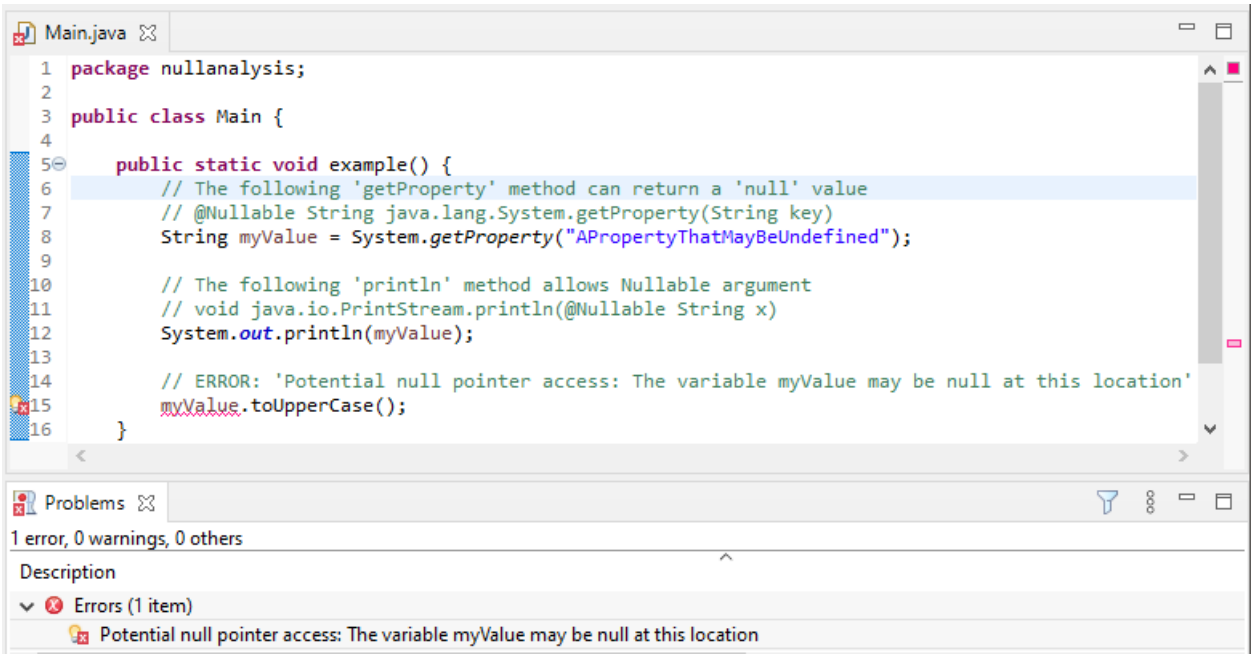


Fig. 395: Example of Null Analysis Detection

Principle

The Null Analysis tool is based on Java annotations. Each Java field, method parameter and method return value must be marked to indicate whether it can be `null` or not.

Once the Java code is annotated, *module projects* must be configured to enable Null Analysis detection in MicroEJ SDK.

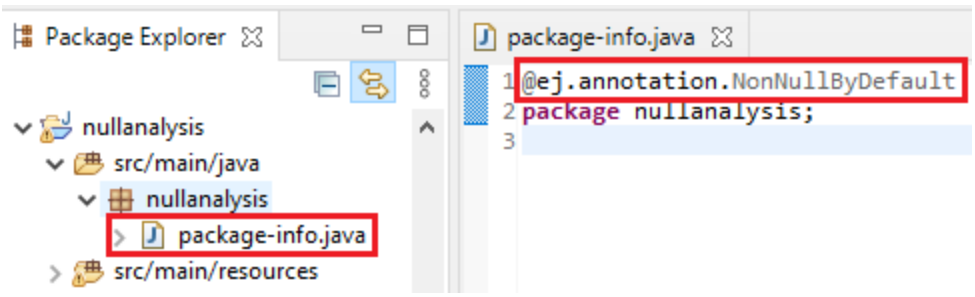
Java Code Annotation

MicroEJ defines its own annotations:

- @NonNullByDefault: Indicates that all fields, method return values or parameters can never be null in the annotated package or type. This rule can be overridden on each element by using the Nullable annotation.
- @Nullable: Indicates that a field, local variable, method return value or parameter can be null.
- @NonNull: Indicates that a field, local variable, method return value or parameter can never be null.

MicroEJ recommends to annotate the Java code as follows:

- In each Java package, create a `package-info.java` file and annotate the Java package with `@NonNullByDefault`. This is a common good practice to deal with non `null` elements by default to avoid undesired NullPointerException. It enforces the behavior which is already widely outlined in Java coding rules.



- In each Java type, annotate all fields, methods return values and parameters that can be null with `@Nullable`. Usually, this information is already available as textual information in the field or method Javadoc comment. The following example of code shows where annotations must be placed:

```
@Nullable
public Object thisFieldCanBeNull;

@Nullable
public Object thisMethodCanReturnNull() {
    return null;
}

public void thisMethodParameterCanBeNull(@Nullable Object param) {

}
```

Module Project Configuration

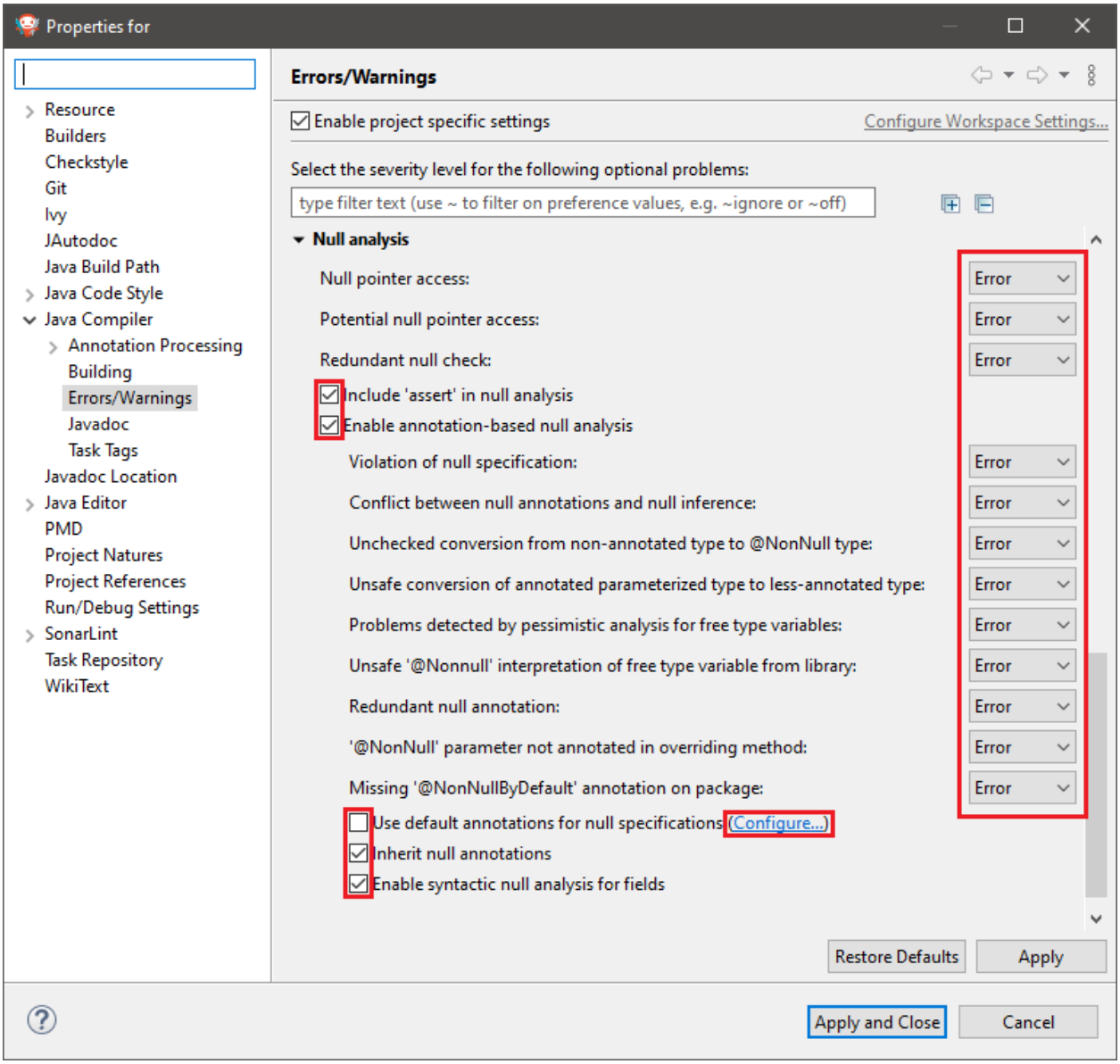
Requirements

EDC-1.3.3 or higher is required when MicroEJ SDK `5.3.0` or higher is used. See EDC 1.3.3 Changelog for more details.

Project configuration

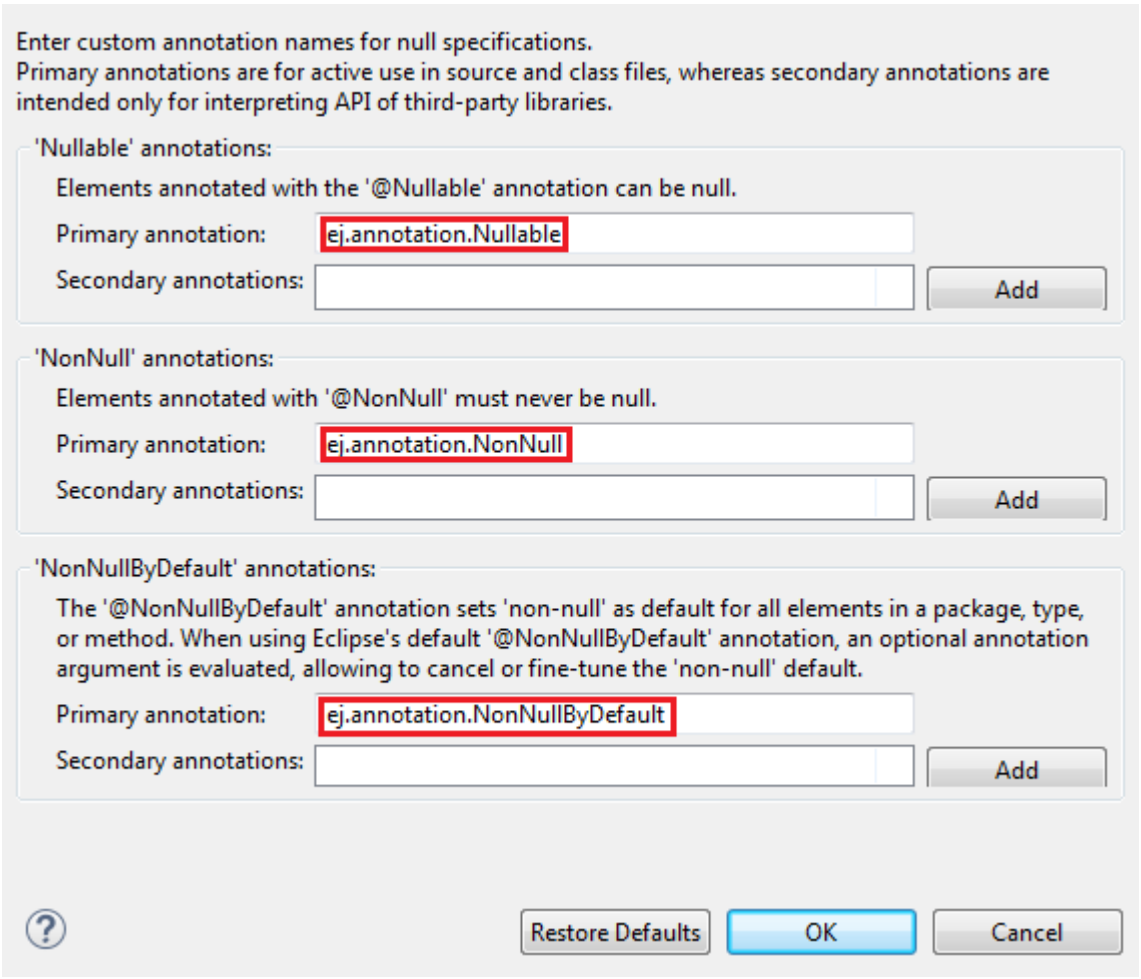
To enable the Null Analysis tool, a *module project* must be configured as follows:

- In the Package Explorer, right-click on the module project and select `Properties`,
- Navigate to `Java Compiler` > `Errors/Warnings`,
- In the `Null analysis` section, configure options as follows:

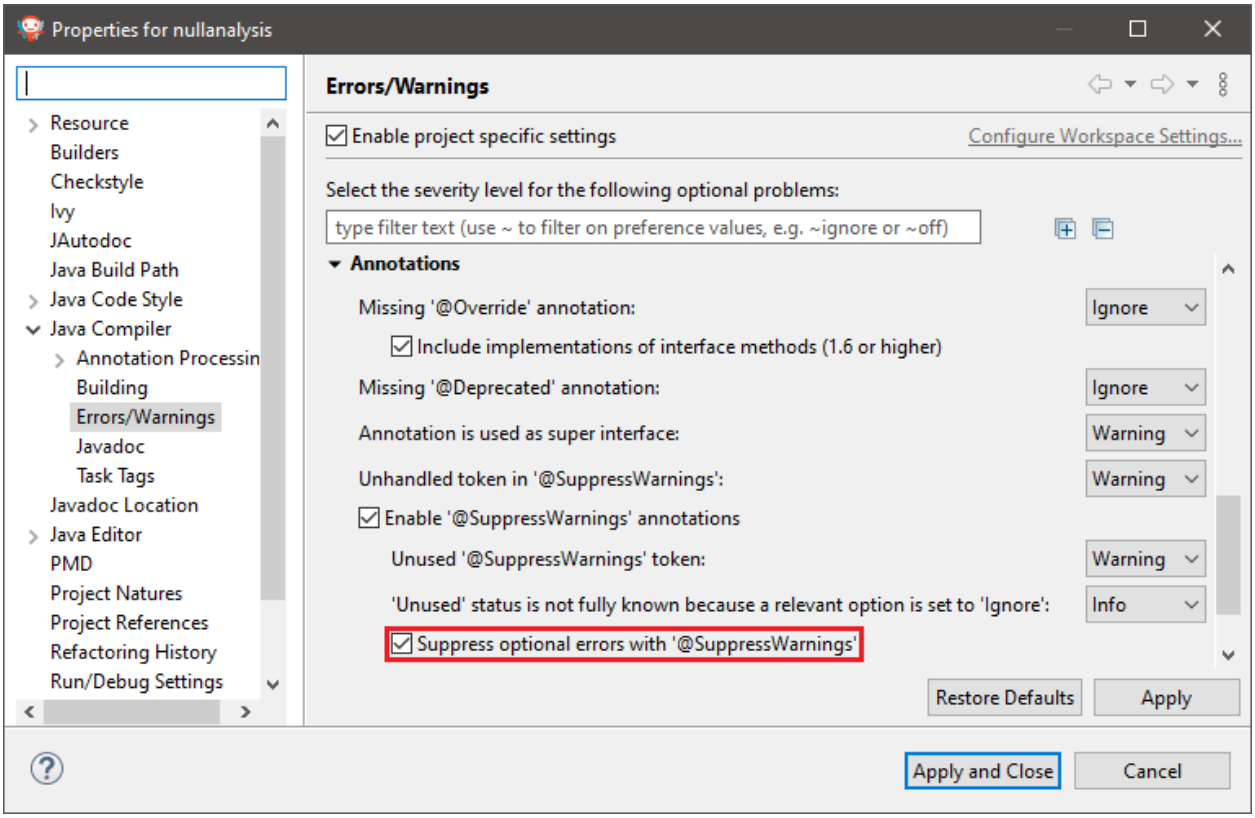


- Click on the Configure… link to configure MicroEJ annotations:

  - ej.annotation.Nullable
  - ej.annotation.NonNull
  - ej.annotation.NonNullByDefault



- In the `Annotations` section, check `Suppress optional errors with '@SuppressWarnings'` option:



This option allows to fully ignore Null Analysis errors in advanced cases using `@SuppressWarnings("null")` annotation.

If you have multiple projects to configure, you can then copy the content of the `.settings` folder to an other *module project*.

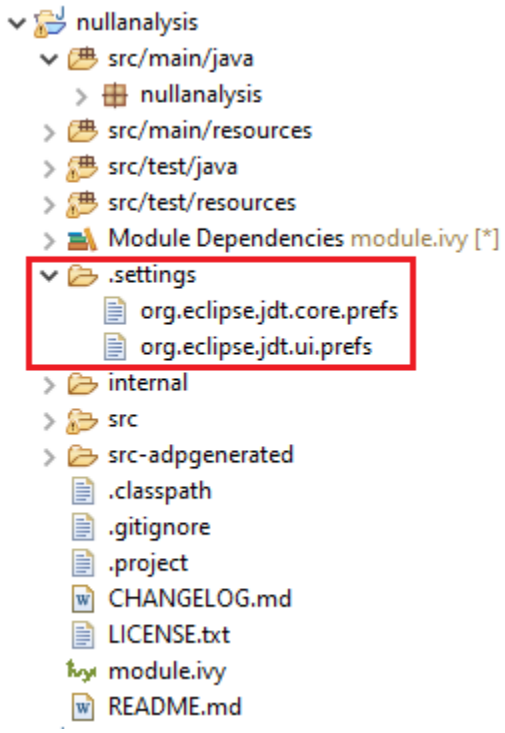


Fig. 396: Null Analysis Settings Folder

> **Warning:** You may lose information if your target module project already has custom parameterization or if it was created with another MicroEJ SDK version. In case of any doubt, please configure the options manually or merge with a text file comparator.

MicroEJ Libraries

Many libraries available on *Central Repository* are annotated with Null Analysis. If you are using a library which is not yet annotated, please contact *our support team*.

For the benefit of Null Analysis, some APIs have been slightly constrained compared to the Javadoc description. Here are some examples to illustrate the philosophy:

- System.getProperty(String key, String def) does not accept a `null` default value, which allows to ensure the returned value is always non `null`.
- Collections of the Java Collections Framework that can hold `null` elements (e.g. HashMap) do not accept `null` elements. This allows APIs to return `null` (e.g. HashMap.get(Object)) only when an element is not contained in the collection.

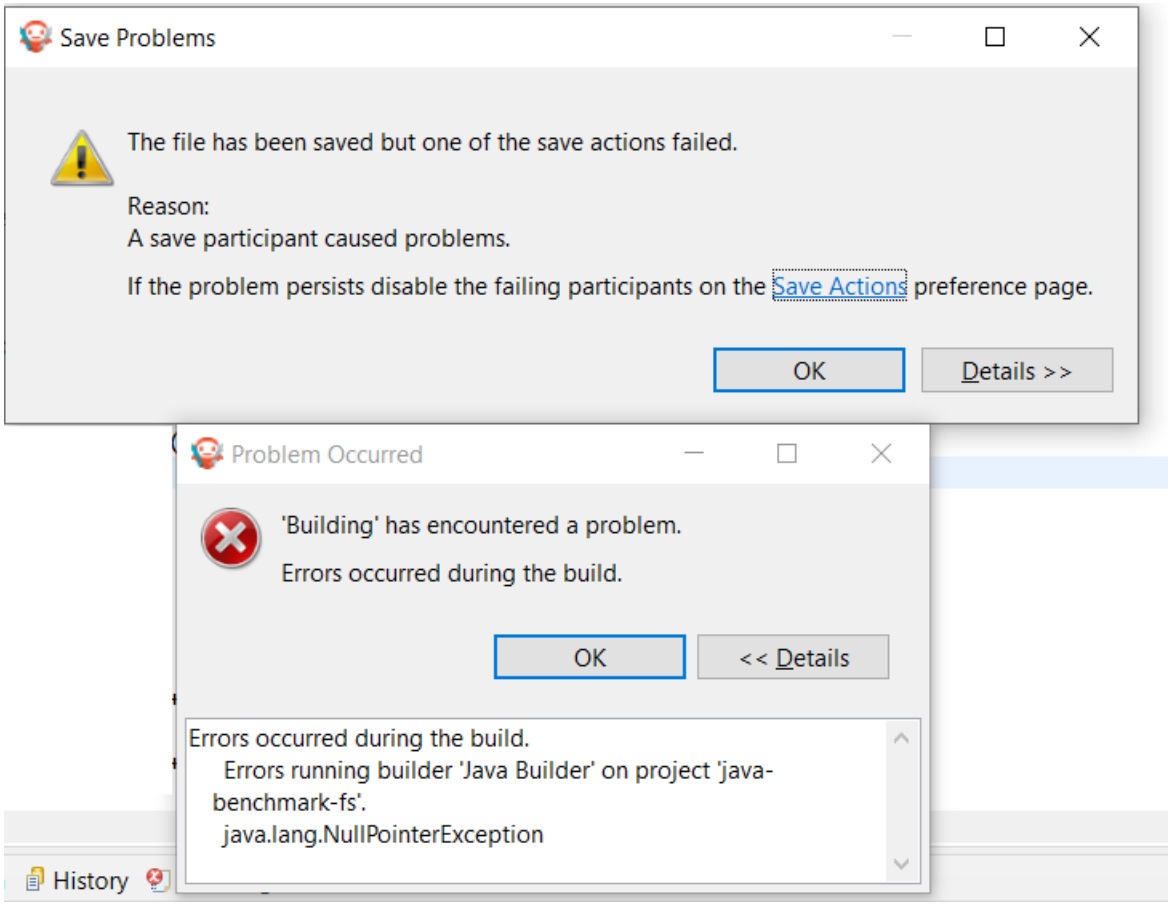
  Implementations are left unchanged and still comply with the Javadoc description whether the Null Analysis is enabled or not. So if these additional constraints are not acceptable for your project, please disable Null Analysis.

Advanced Use

For more information about Null Analysis and inter-procedural analysis, please visit Eclipse JDT Null Analysis documentation.

Troubleshooting

The project cannot build anymore after Null Analysis setup




```
java.lang.NullPointerException
  at org.eclipse.
↪jdt.internal.compiler.lookup.BinaryTypeBinding.getMethods(BinaryTypeBinding.java:1348)
  at org.eclipse.jdt.
↪internal.compiler.lookup.AnnotationBinding.setMethodBindings(AnnotationBinding.java:238)
  at org.eclipse.jdt.internal.
↪compiler.lookup.LookupEnvironment.createAnnotation(LookupEnvironment.java:995)
  at org.eclipse.jdt.internal.
↪compiler.lookup.AnnotationBinding.buildTargetAnnotation(AnnotationBinding.java:191)
  at org.eclipse.jdt.internal.
↪compiler.lookup.AnnotationBinding.addStandardAnnotations(AnnotationBinding.java:79)
  at org.eclipse.jdt.internal.
↪compiler.lookup.BinaryTypeBinding.retrieveAnnotations(BinaryTypeBinding.java:1698)
```

(continues on next page)

(continued from previous page)

```
 at org.eclipse.
↪jdt.internal.compiler.lookup.ReferenceBinding.getAnnotations(ReferenceBinding.java:1054)
 at org.eclipse.jdt.internal.compiler.
↪lookup.BinaryTypeBinding.evaluateTypeQualifierDefault(BinaryTypeBinding.java:2021)
 at org.eclipse.jdt.internal.compiler.
↪lookup.BinaryTypeBinding.getNonNullByDefaultValue(BinaryTypeBinding.java:1999)
 at org.eclipse.jdt.internal.compiler.lookup.
↪BinaryTypeBinding.scanTypeForNullDefaultAnnotation(BinaryTypeBinding.java:1943)
 at org.eclipse.jdt.
↪internal.compiler.lookup.BinaryTypeBinding.cachePartsFrom(BinaryTypeBinding.java:470)
 at org.eclipse.jdt.internal.
↪compiler.lookup.LookupEnvironment.createBinaryTypeFrom(LookupEnvironment.java:1055)
 at org.eclipse.jdt.internal.
↪compiler.lookup.LookupEnvironment.createBinaryTypeFrom(LookupEnvironment.java:1036)
 at org.eclipse.jdt.internal.compiler.Compiler.accept(Compiler.java:308)
 at org.eclipse.
↪jdt.internal.compiler.lookup.LookupEnvironment.askForType(LookupEnvironment.java:326)
 at org.eclipse.jdt.internal.compiler.lookup.PackageBinding.getType(PackageBinding.java:195)
 at org.eclipse.
↪jdt.internal.compiler.lookup.PackageBinding.initDefaultNullness(PackageBinding.java:325)
 at org.eclipse.
↪jdt.internal.compiler.lookup.PackageBinding.getDefaultNullness(PackageBinding.java:339)
 at org.eclipse.jdt.internal.compiler.lookup.
↪BinaryTypeBinding.scanTypeForNullDefaultAnnotation(BinaryTypeBinding.java:1965)
 at org.eclipse.jdt.
↪internal.compiler.lookup.BinaryTypeBinding.cachePartsFrom(BinaryTypeBinding.java:470)
 at org.eclipse.jdt.internal.
↪compiler.lookup.LookupEnvironment.createBinaryTypeFrom(LookupEnvironment.java:1055)
 at org.eclipse.jdt.internal.
↪compiler.lookup.LookupEnvironment.createBinaryTypeFrom(LookupEnvironment.java:1036)
 at org.eclipse.jdt.internal.compiler.Compiler.accept(Compiler.java:308)
 at org.eclipse.
↪jdt.internal.compiler.lookup.LookupEnvironment.askForType(LookupEnvironment.java:326)
 at org.eclipse.
↪jdt.internal.compiler.lookup.LookupEnvironment.getType(LookupEnvironment.java:1705)
 at org.eclipse.jdt.internal.
↪compiler.lookup.LookupEnvironment.getResolvedType(LookupEnvironment.java:1633)
 at org.eclipse.jdt.internal.compiler.
↪lookup.LookupEnvironment.getResolvedJavaBaseType(LookupEnvironment.java:1645)
 at org.eclipse.jdt.internal.
↪compiler.lookup.AnnotationBinding.buildTargetAnnotation(AnnotationBinding.java:134)
 at org.eclipse.jdt.internal.
↪compiler.lookup.AnnotationBinding.addStandardAnnotations(AnnotationBinding.java:79)
 at org.eclipse.jdt.internal.
↪compiler.lookup.BinaryTypeBinding.retrieveAnnotations(BinaryTypeBinding.java:1698)
 at org.eclipse.
↪jdt.internal.compiler.lookup.ReferenceBinding.getAnnotations(ReferenceBinding.java:1054)
 at org.eclipse.jdt.internal.compiler.
↪lookup.BinaryTypeBinding.evaluateTypeQualifierDefault(BinaryTypeBinding.java:2021)
 at org.eclipse.jdt.internal.compiler.
↪lookup.BinaryTypeBinding.getNonNullByDefaultValue(BinaryTypeBinding.java:1999)
 at org.eclipse.jdt.internal.compiler.lookup.
↪BinaryTypeBinding.scanTypeForNullDefaultAnnotation(BinaryTypeBinding.java:1943)
 at org.eclipse.jdt.
↪internal.compiler.lookup.BinaryTypeBinding.cachePartsFrom(BinaryTypeBinding.java:470)
 at org.eclipse.jdt.internal.
```

(continues on next page)

(continued from previous page)

```
↪compiler.lookup.LookupEnvironment.createBinaryTypeFrom(LookupEnvironment.java:1055)
 at org.eclipse.jdt.internal.
↪compiler.lookup.LookupEnvironment.createBinaryTypeFrom(LookupEnvironment.java:1036)
 at org.eclipse.jdt.internal.compiler.Compiler.accept(Compiler.java:308)
 at org.eclipse.
↪jdt.internal.compiler.lookup.LookupEnvironment.askForType(LookupEnvironment.java:326)
 at org.eclipse.jdt.internal.compiler.lookup.PackageBinding.getType(PackageBinding.java:195)
 at org.eclipse.jdt.
↪internal.compiler.lookup.PackageBinding.isViewedAsDeprecated(PackageBinding.java:314)
 at org.eclipse.jdt.internal.
↪compiler.lookup.ReferenceBinding.isViewedAsDeprecated(ReferenceBinding.java:1745)
 at org.eclipse.jdt.
↪internal.compiler.lookup.BinaryTypeBinding.cachePartsFrom(BinaryTypeBinding.java:566)
 at org.eclipse.jdt.internal.
↪compiler.lookup.LookupEnvironment.createBinaryTypeFrom(LookupEnvironment.java:1055)
 at org.eclipse.jdt.internal.
↪compiler.lookup.LookupEnvironment.createBinaryTypeFrom(LookupEnvironment.java:1036)
 at org.eclipse.jdt.internal.compiler.Compiler.accept(Compiler.java:308)
 at org.eclipse.
↪jdt.internal.compiler.lookup.LookupEnvironment.askForType(LookupEnvironment.java:257)
 at org.eclipse.
↪jdt.internal.compiler.lookup.LookupEnvironment.getType(LookupEnvironment.java:1703)
 at org.eclipse.jdt.internal.compiler.
↪lookup.BinaryTypeBinding.getNonNullByDefaultValue(BinaryTypeBinding.java:1995)
 at org.eclipse.jdt.internal.compiler.lookup.
↪BinaryTypeBinding.scanTypeForNullDefaultAnnotation(BinaryTypeBinding.java:1943)
 at org.eclipse.jdt.
↪internal.compiler.lookup.BinaryTypeBinding.cachePartsFrom(BinaryTypeBinding.java:470)
 at org.eclipse.jdt.internal.
↪compiler.lookup.LookupEnvironment.createBinaryTypeFrom(LookupEnvironment.java:1055)
 at org.eclipse.jdt.internal.
↪compiler.lookup.LookupEnvironment.createBinaryTypeFrom(LookupEnvironment.java:1036)
 at org.eclipse.jdt.internal.compiler.Compiler.accept(Compiler.java:308)
 at org.eclipse.
↪jdt.internal.compiler.lookup.LookupEnvironment.askForType(LookupEnvironment.java:326)
 at org.eclipse.jdt.internal.compiler.lookup.PackageBinding.getType(PackageBinding.java:195)
 at org.eclipse.
↪jdt.internal.compiler.lookup.PackageBinding.initDefaultNullness(PackageBinding.java:325)
 at org.eclipse.
↪jdt.internal.compiler.lookup.PackageBinding.getDefaultNullness(PackageBinding.java:339)
 at org.eclipse.jdt.internal.compiler.lookup.
↪BinaryTypeBinding.scanTypeForNullDefaultAnnotation(BinaryTypeBinding.java:1965)
 at org.eclipse.jdt.
↪internal.compiler.lookup.BinaryTypeBinding.cachePartsFrom(BinaryTypeBinding.java:470)
 at org.eclipse.jdt.internal.
↪compiler.lookup.LookupEnvironment.createBinaryTypeFrom(LookupEnvironment.java:1055)
 at org.eclipse.jdt.internal.
↪compiler.lookup.LookupEnvironment.createBinaryTypeFrom(LookupEnvironment.java:1036)
 at org.eclipse.jdt.internal.compiler.Compiler.accept(Compiler.java:308)
 at org.eclipse.
↪jdt.internal.compiler.lookup.LookupEnvironment.askForType(LookupEnvironment.java:326)
 at org.eclipse.
↪jdt.internal.compiler.lookup.LookupEnvironment.getType(LookupEnvironment.java:1705)
 at org.eclipse.jdt.internal.
↪compiler.lookup.LookupEnvironment.getResolvedType(LookupEnvironment.java:1633)
 at org.eclipse.jdt.internal.compiler.
```

(continues on next page)

(continued from previous page)

```
↪lookup.LookupEnvironment.getResolvedJavaBaseType(LookupEnvironment.java:1645)
 at org.eclipse.jdt.internal.compiler.lookup.Scope.getJavaLangObject(Scope.java:2961)
 at org.eclipse.jdt.internal.compiler.lookup.ClassScope.connectSuperclass(ClassScope.java:1065)
 at org.
↪eclipse.jdt.internal.compiler.lookup.ClassScope.connectTypeHierarchy(ClassScope.java:1246)
 at org.eclipse.jdt.internal.compiler.
↪lookup.CompilationUnitScope.connectTypeHierarchy(CompilationUnitScope.java:367)
 at org.eclipse.jdt.internal.
↪compiler.lookup.LookupEnvironment.completeTypeBindings(LookupEnvironment.java:518)
 at org.eclipse.jdt.internal.compiler.Compiler.internalBeginToCompile(Compiler.java:878)
 at org.eclipse.jdt.internal.compiler.Compiler.beginToCompile(Compiler.java:394)
 at org.eclipse.jdt.internal.compiler.Compiler.compile(Compiler.java:444)
 at org.eclipse.jdt.internal.compiler.Compiler.compile(Compiler.java:426)
 at org.eclipse.
↪jdt.internal.core.builder.AbstractImageBuilder.compile(AbstractImageBuilder.java:386)
 at␣
↪org.eclipse.jdt.internal.core.builder.BatchImageBuilder.compile(BatchImageBuilder.java:214)
 at org.eclipse.
↪jdt.internal.core.builder.AbstractImageBuilder.compile(AbstractImageBuilder.java:318)
 at org.eclipse.jdt.internal.core.builder.BatchImageBuilder.build(BatchImageBuilder.java:79)
 at org.eclipse.jdt.internal.core.builder.JavaBuilder.buildAll(JavaBuilder.java:275)
 at org.eclipse.jdt.internal.core.builder.JavaBuilder.build(JavaBuilder.java:192)
 at org.eclipse.core.internal.events.BuildManager$2.run(BuildManager.java:832)
 at org.eclipse.core.runtime.SafeRunner.run(SafeRunner.java:45)
 at org.eclipse.core.internal.events.BuildManager.basicBuild(BuildManager.java:220)
 at org.eclipse.core.internal.events.BuildManager.basicBuild(BuildManager.java:263)
 at org.eclipse.core.internal.events.BuildManager$1.run(BuildManager.java:316)
 at org.eclipse.core.runtime.SafeRunner.run(SafeRunner.java:45)
 at org.eclipse.core.internal.events.BuildManager.basicBuild(BuildManager.java:319)
 at org.eclipse.core.internal.events.BuildManager.basicBuildLoop(BuildManager.java:371)
 at org.eclipse.core.internal.events.BuildManager.build(BuildManager.java:392)
 at org.eclipse.core.internal.events.AutoBuildJob.doBuild(AutoBuildJob.java:154)
 at org.eclipse.core.internal.events.AutoBuildJob.run(AutoBuildJob.java:244)
 at org.eclipse.core.internal.jobs.Worker.run(Worker.java:63)
```

You may encounter the two popup windows and the full stack trace above when your version of EDC is too old. To fix this issue, please use EDC-1.3.3 or higher with MicroEJ SDK 5.3.0 or higher.

3.11.10 IDE

The SDK provides an Integrated Development Environment (IDE) for creating and building Applications. It is based on Eclipse Java Edition and relies on the integrated Java Compiler (JDT).

Startup

When starting the SDK, it prompts you to select the last used workspace or a default workspace on the first run.

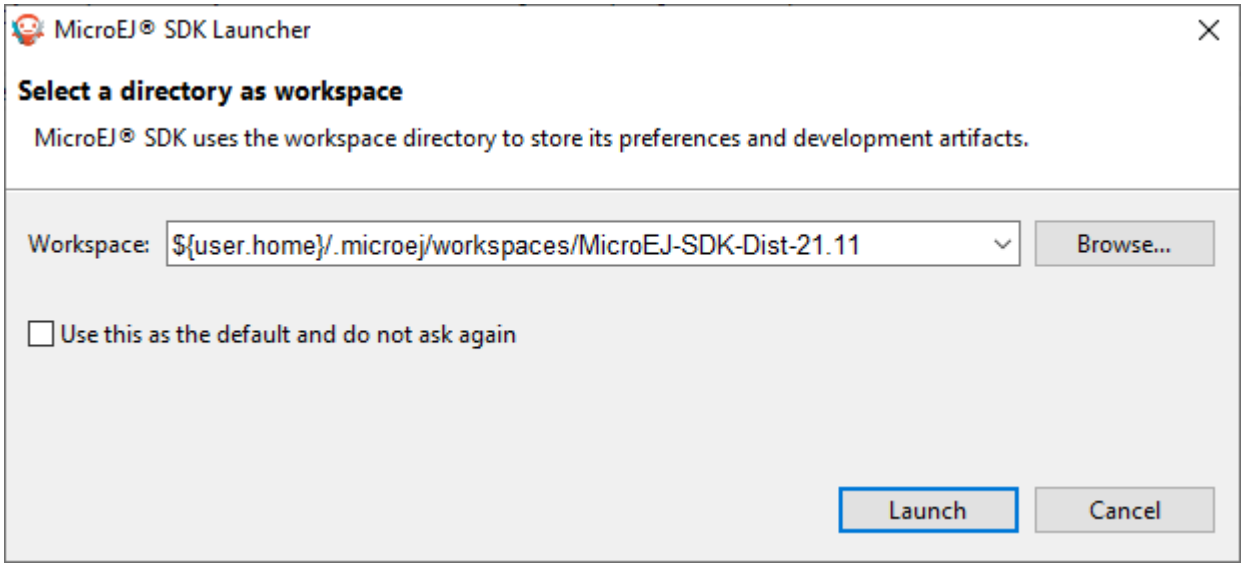


Fig. 397: Workspace selection

A workspace is the Eclipse main folder where are imported a set of projects containing the source code.

When loading a new workspace, the SDK prompts for the location of the MicroEJ repository, where Architectures, Platforms or Virtual Devices will be imported.

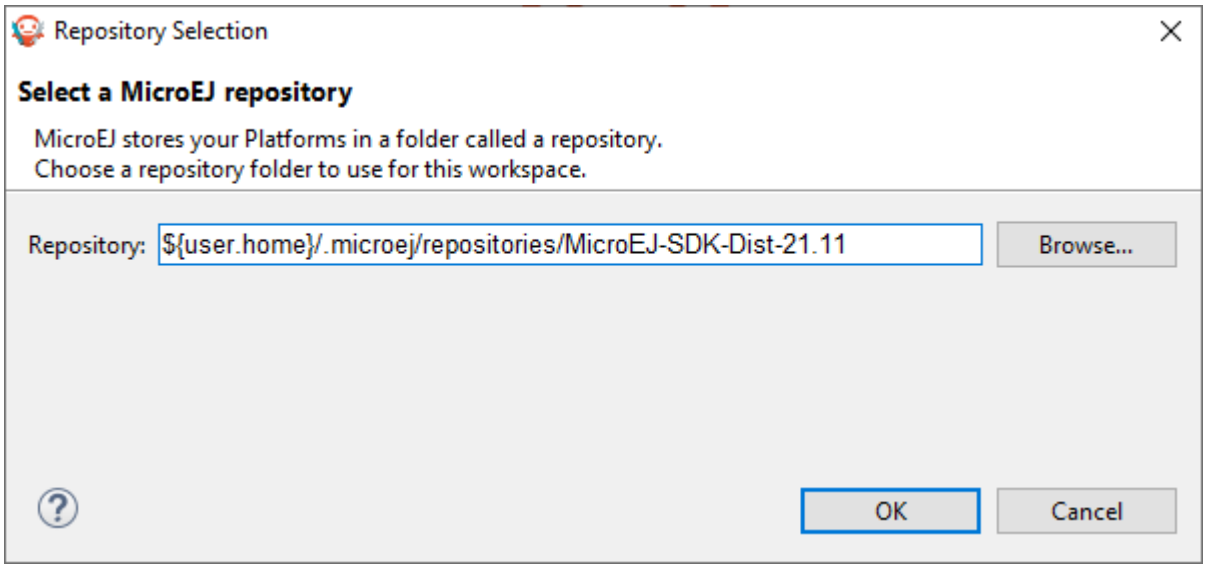


Fig. 398: Repository selection

By default, the SDK suggests to point to the default repository on your operating system, located at ${user.home}/.microej/repositories/[version] . You can select an alternative loca-

tion. Another common practice is to define a local repository relative to the workspace, so that the workspace is self-contained, without external file system links and can be shared within a zip file.

Resolve Dependencies in Workspace

When resolving the modules' dependencies, if the project of a dependency is imported and opened in the same workspace as the module, the project is directly used for compilation and execution instead of using the dependency, provided that the dependency's project has the same version as the one required by the module.

For example, suppose that the workspace contains a module myApp and its dependency mylib :

Fig. 399: A module and its dependency opened in the same workspace

If the mylib project's version is 1.0.0 , it is used for compilation and execution. Otherwise the published artifact is downloaded from the artifact repository.

To avoid a dependency to be resolved in the workspace, you can close the corresponding project or remove it from the workspace.

Warning: If you open, close, import or remove a project, you must refresh the dependency resolution of other previously imported projects by clicking on the Resolve All button :

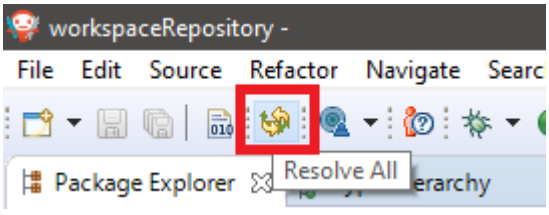


Fig. 400: Resolve all the workspace projects

Resolve Foundation Libraries in Workspace

A *Foundation Library* is composed of :

- An API project that contains Java classes, methods and fields used at compile time with their associated Javadoc,
- An Implementation project that contains the runtime code executed by the Platform and Low Level C header files

Beside Foundation Library projects, there is usually a *Mock* project that contains the implementation of native methods for simulation.

Note: To learn how to setup a Foundation Library, please consult the How-to available on https://github.com/MicroEJ/How-To/tree/master/FoundationLibrary-Get-Started.

When the API is set as a dependency, the Implementation project is automatically used at runtime if it is opened in the workspace.

If a Mock project or a Front Panel project is also opened in the workspace, it is automatically used for execution on Simulator.

Note: When opened in the workspace, Foundation Library Implementation projects, Mock projects and Front Panel projects are loaded, regardless of their version, prior to the ones provided by the Platform (if any).

To avoid the use of an Implementation project, a Mock project or a Front Panel project, uncheck the `Resolve Foundation Library in workspace` option in `Window` > `Preferences` > `MicroEJ` > `Settings` .

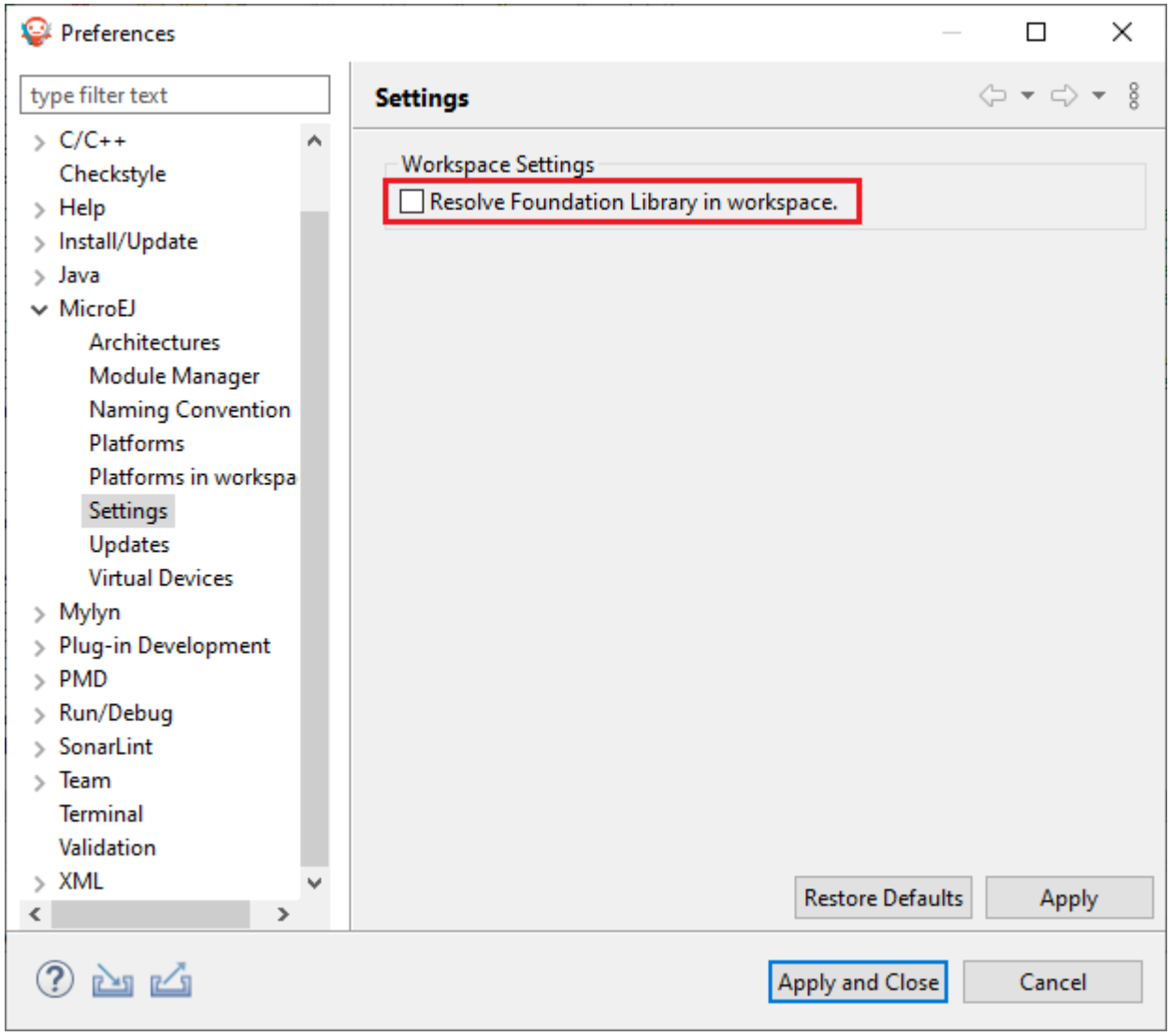


Fig. 401: Resolve Foundation Library in workspace

Resolve Front Panel in Workspace

A Front Panel is a "mock" of the control panel of the device. The Front Panel generates a graphical representation of the device, and is displayed in a window on the user's development machine when the application is executed in the Simulator.

Note: To learn more about Front Panels, consult the *Front Panel section*.

When a Front Panel project is opened in the workspace, it is automatically used at runtime when launching the Simulator.

Note: This feature requires SDK version 5.7.0 or higher and Architecture version 8.0 or higher.

If the workspace contains several Front Panel projects, they are all automatically used by the

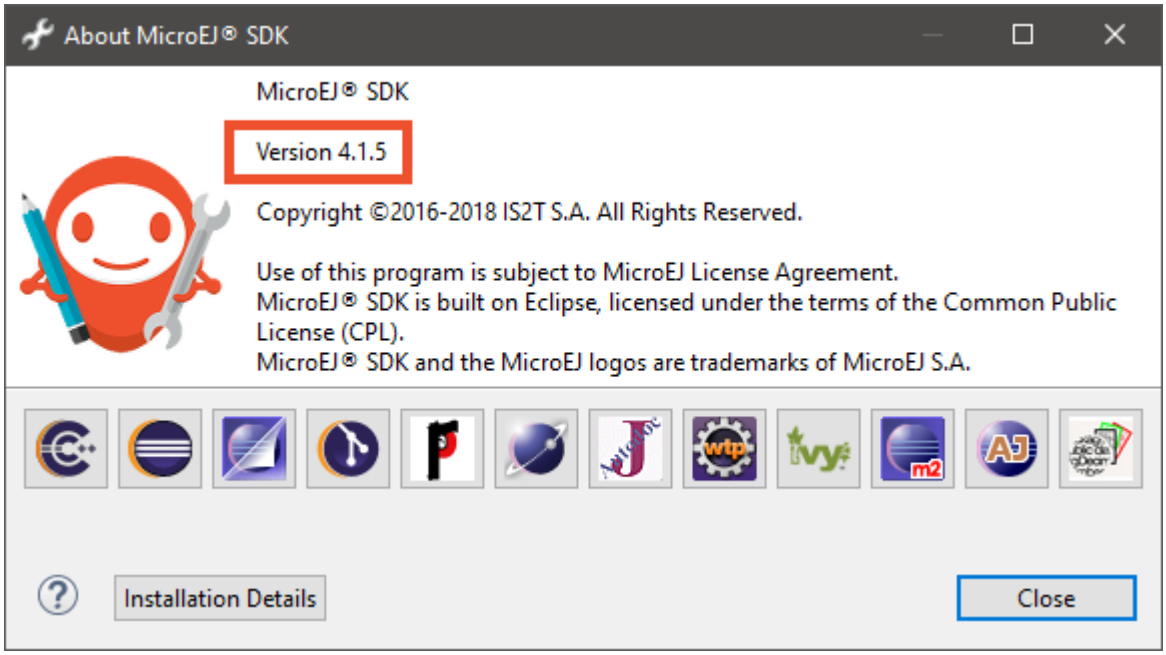
Simulator, which can very probably causes issues. You can select the Front Panel you want to use by closing all the other Front Panel projects.

Also, a Front Panel project can contain several Front Panel descriptor files. Refer to the *Multiple Front Panel Files* section to know how to select the file you want to use.

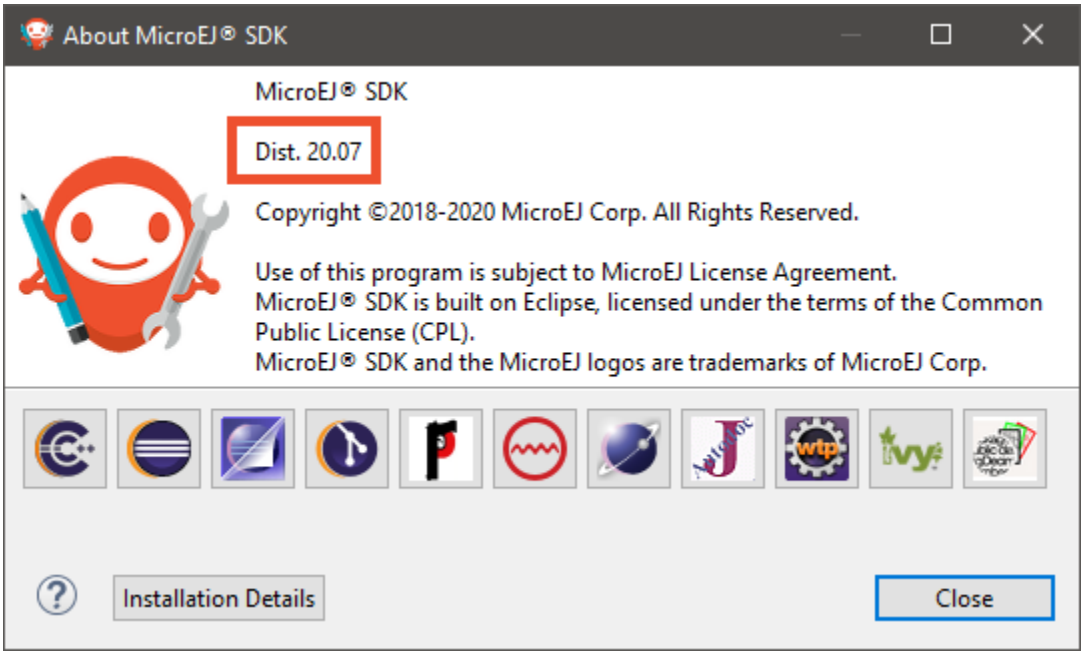
3.11.11 SDK Version

In the SDK, go to `Help` > `About MicroEJ SDK` menu.

In case of SDK `4.1.x`, the SDK version is directly displayed, such as `4.1.5`:

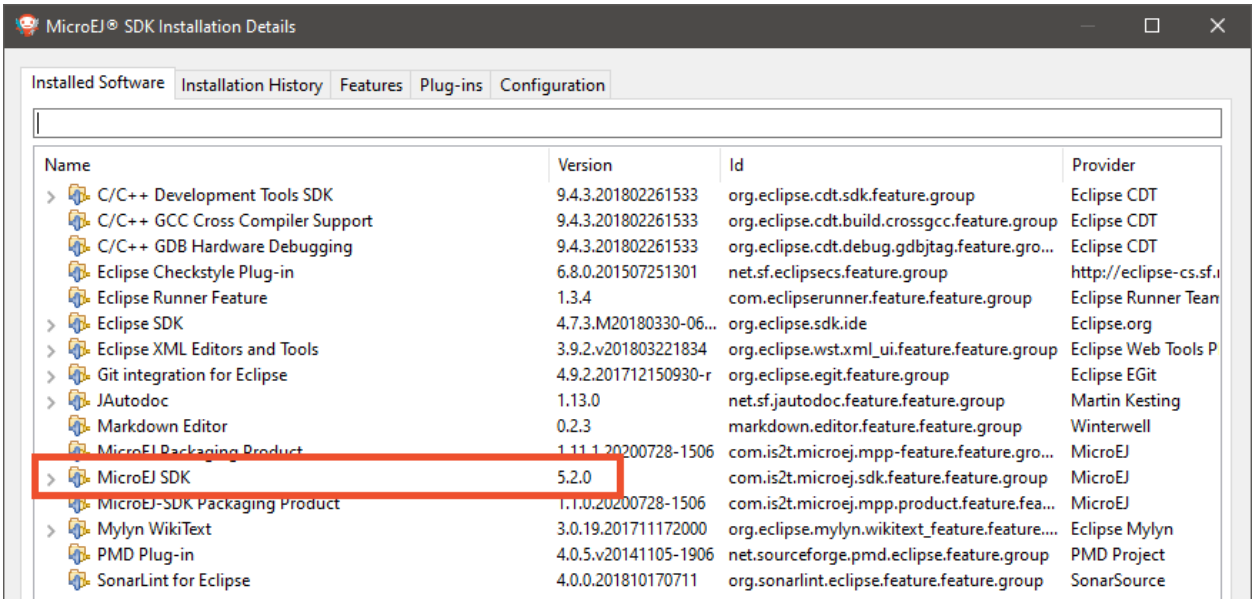


In case of SDK `5.x`, the value displayed is the SDK distribution, such as `19.05` or `20.07`:



To retrieve the SDK version that is currently installed in this distribution, proceed with the following steps:

- Click on the Installation Details button,
- Click on the Installed Software tab,
- Retrieve the version of entry named MicroEJ SDK .



3.11.12 MicroEJ Module Manager

Introduction

Modern electronic device design involves many parts and teams to collaborate to finally obtain a product to be sold on its market. MicroEJ encourages modular design which involves various stake holders: hardware engineers, UX designers, graphic designers, drivers/BSP engineers, software engineers, etc.

Modular design is a design technique that emphasizes separating the functionality of an application into independent, interchangeable modules. Each module contains everything necessary to execute only one aspect of the desired functionality. In order to have team members collaborate internally within their team and with other teams, MicroEJ provides a powerful modular design concept, with smart module dependencies, controlled by the MicroEJ Module Manager (MMM). MMM frees engineers from the difficult task of computing module dependencies. Engineers specify the bare minimum description of the module requirements.

The following schema introduces the main concepts detailed in this chapter.

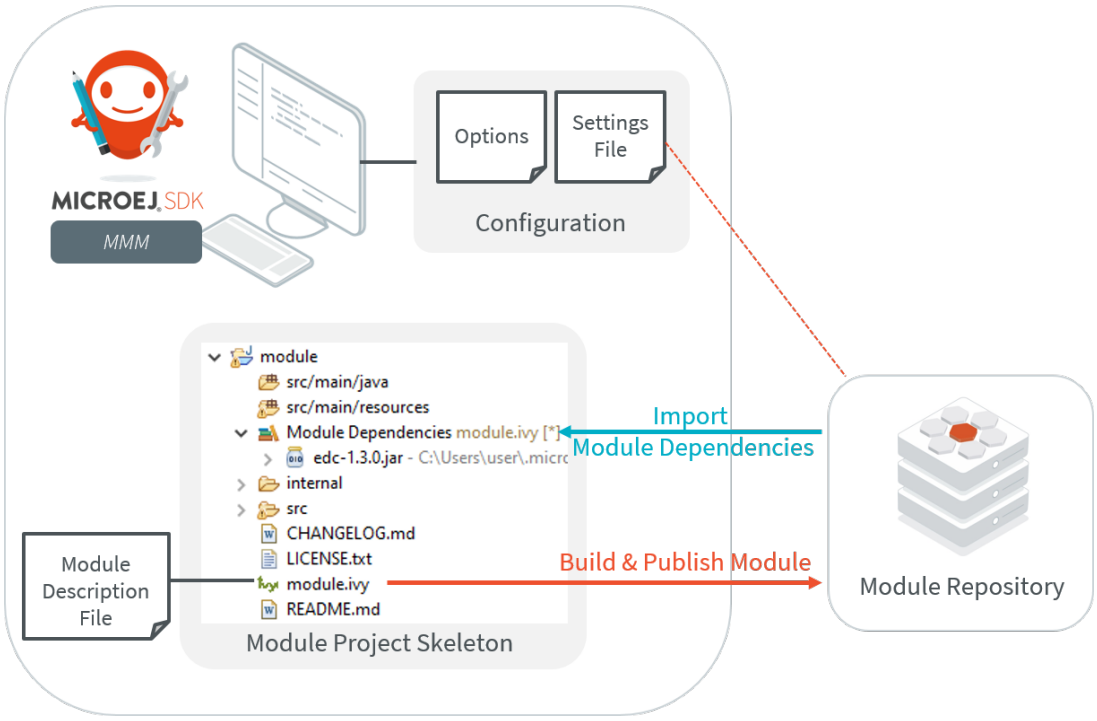


Fig. 402: MMM Overview

MMM is based on the following tools:

- Apache Ivy (http://ant.apache.org/ivy) for dependencies resolution and module publication;
- Apache EasyAnt (https://ant.apache.org/easyant/history/trunk/reference.html) for module build from source code.

Specification

MMM provides a non ambiguous semantic for dependencies resolution. Please consult the MMM specification available on https://developer.microej.com/packages/documentation/TLT-0831-SPE-MicroEJModuleManager-2.0-E.pdf.

Module Project Skeleton

In the SDK, a new MicroEJ module project is created as follows:

- Select File > New > Project… ,
- Select MicroEJ > Module Project [1],
- Fill the module information (project name, module organization, name and revision),
- Select one of the suggested skeletons depending on the desired *module nature*,
- Click on Finish .

The project is created and a set of files and directories are generated from the selected skeleton.

Note: When an empty Eclipse project already exists or when the skeleton has to be created within an existing directory, the MicroEJ module is created as follows:

- In the *Package Explorer*, click on the parent project or directory,
- Select File > New > Other… ,
- Select EasyAnt > EasyAnt Skeleton .

Module Description File

A module description file is an Ivy configuration file named module.ivy , located at the root of each MicroEJ module project. It describes the *module nature* (also called build type) and dependencies to other modules.

```
<ivy-module version=
↪"2.0" xmlns:ea="http://www.easyant.org" xmlns:m="http://ant.apache.org/ivy/extra"
                    xmlns:ej="https://developer.microej.com" ej:version="2.0.0">
   <info␣
↪organisation="[organisation]" module="[name]" status="integration" revision="[version]">
      <ea:build organisation=
↪"com.is2t.easyant.buildtypes" module="[buildtype_name]" revision="[buildtype_version]">
         <ea:property name="[buildoption_name]" value="[buildoption_value]"/>
      </ea:build>
   </info>

   <configurations defaultconfmapping="default->default;provided->provided">
      <conf name="default" visibility="public"/>
      <conf name="provided" visibility="public"/>
      <conf name="documentation" visibility="public"/>
      <conf name="source" visibility="public"/>
```

(continues on next page)

[1] If using SDK versions lower than 5.2.0 , please refer to the following section.

(continued from previous page)

```
    <conf name="dist" visibility="public"/>
    <conf name="test" visibility="private"/>
  </configurations>

  <publications>
  </publications>

  <dependencies>
    <dependency org="[dep_organisation]" name="[dep_name]" rev="[dep_version]"/>
  </dependencies>
</ivy-module>
```

Enable MMM Semantic

The MMM semantic is enabled in a module by adding the MicroEJ XML namespace and the `ej:version` attribute in the `ivy-module` node:

```
<ivy-module xmlns:ej="https://developer.microej.com" ej:version="2.0.0">
```

Note: Multiple namespaces can be declared in the `ivy-module` node.

MMM semantic is enabled in the module created with the *Module Project Skeleton*.

Module Dependencies

Module dependencies are added to the `dependencies` node as follow:

```
<dependencies>
  <dependency org="[dep_organisation]" name="[dep_name]" rev="[dep_version]"/>
</dependencies>
```

When no matching rule is specified, the default matching rule is `compatible`.

Dependency Matching Rule

The following matching rules are specified by MMM:

Name	Range Notation	Semantic
compatible	[M.m.p-RC, (M+1).0.0-RC[	Equal or up to next major version. Default if not set.
equivalent	[M.m.p-RC, M.(m+1).0-RC [	Equal or up to next minor version
greaterOrEqual	[M.m.p-RC, ∞[	Equal or greater versions
perfect	[M.m.p-RC, M.m.(p+1)-RC[	Exact match (strong dependency)

Set the matching rule of a given dependency with `ej:match="matching rule"`. For example:

```
<dependency␣
↪org="[dep_organisation]" name="[dep_name]" rev="[dep_version]" ej:match="perfect" />
```

Dependency Visibility

- A dependency declared `public` is transitively resolved by upper modules. The default when not set.
- A dependency declared `private` is only used by the module itself, typically for:
  - Bundling the content into the module
  - Testing the module

The visibility is set by the configurations declared in the `configurations` node. For example:

```
<configurations defaultconfmapping="default->default;provided->provided">
   <conf name="[conf_name]" visibility="private"/>
</configurations>
```

The configuration of a dependency is specified by setting the `conf` attribute, for example:

```
<dependency org=
↪"[dep_organisation]" name="[dep_name]" rev="[dep_version]" conf="[conf_name]->*" />
```

Build Options

MMM builds can be configured by settings options in the `module.ivy` file using the `ea:property` tag inside the `ea:build` tag:

```
<ea:build organisation="..." module="..." revision="x.y.z">
   <ea:property name="[build_option_name]" value="[build_option_value]"/>
</ea:build>
```

Refer to the documentation of *Module Natures* for the list of available build options for each Module Nature.

The options can also be defined via System Properties. If an option is defined as both System Property and `ea:property` tag, the value passed as System Property takes precedence.

Automatic Update Before Resolution

The Easyant plugin `ivy-update` can be used to automatically update the version (attribute `rev`) of every module dependencies declared.

```
<info␣
↪organisation="[organisation]" module="[name]" status="integration" revision="[version]">
   <ea:plugin org="com.is2t.easyant.plugins" name="ivy-update" revision="1.+" />
</info>
```

When the plugin is enabled, for each *module dependency*, MMM will check the version declared in the module file and update it to the highest version available which satisfies the matching rule of the dependency.

SDK Configuration

By default, when starting an empty workspace, the SDK is configured to import dependencies from *MicroEJ Central Repository* and to publish built modules to a local directory. The repository configuration is stored in a *settings file* (ivysettings.xml), and the default one is located at <USER_HOME>\.microej\microej-ivysettings-[VERSION].xml

Preferences Page

The MMM preferences page in the SDK is available at Window > Preferences > MicroEJ > Module Manager Page 1819, 1.

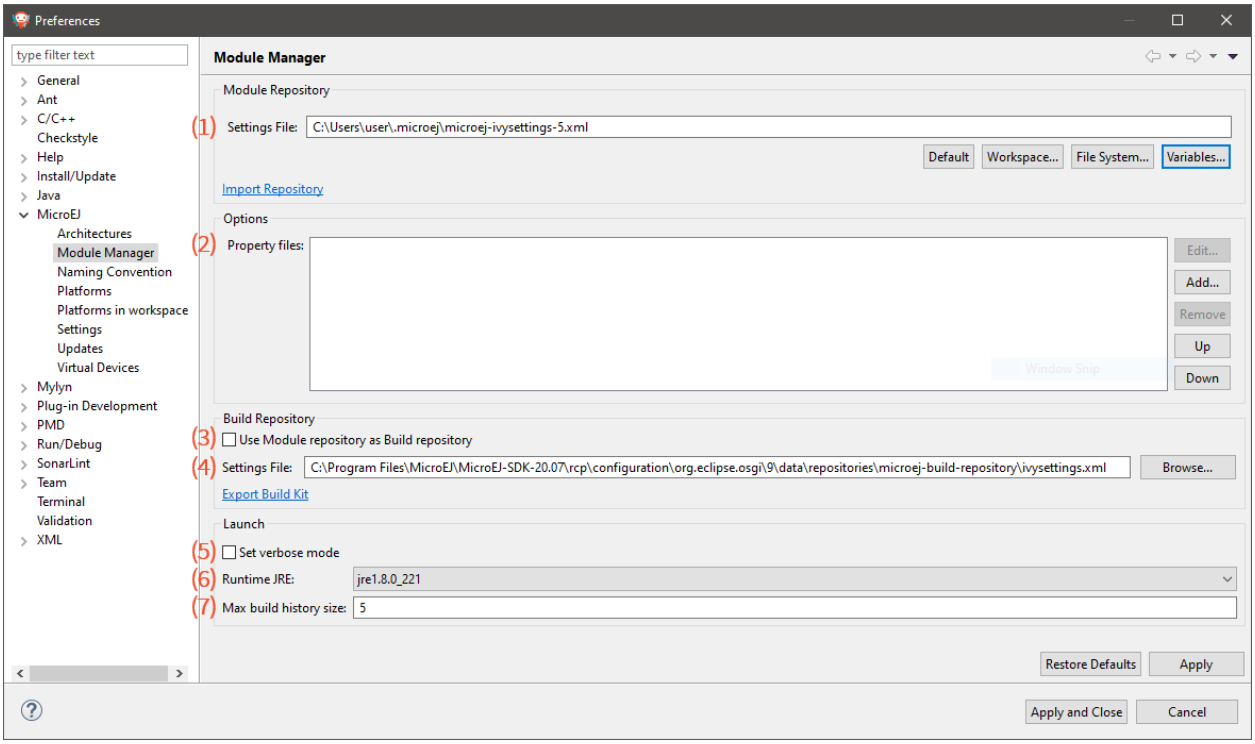
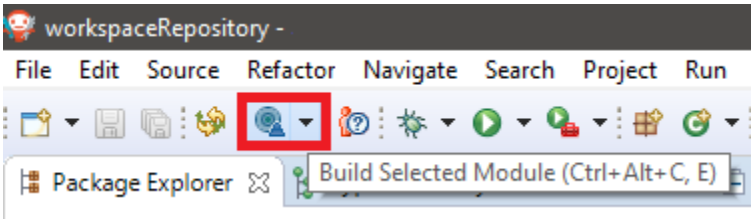


Fig. 403: MMM Preferences Page

This page allows to configure the following elements:

1. Settings File : the file describing how to connect *module repositories*. See the *settings file* section.
2. Options : files declaring MMM options. See the *Options* section.
3. Use Module repository as Build repository : the *settings file* for connecting the build repository in place of the one bundled in the SDK. This option shall not be enabled by default and is reserved for advanced configuration.
4. Build repository Settings File : the *settings file* for connecting the build repository in place of the one bundled in the SDK. This option is automatically initialized the first time the SDK is launched. It shall not be modified by default and is reserved for advanced configuration.

5. Set verbose mode : to enable advanced debug traces when building a module.
6. Runtime JRE : the Java Runtime Environment that executes the build process.
7. Max build history size : the maximum number of previous builds available in Build Module shortcut list:



Settings File

The settings file is an XML file that describes how MMM connects local or online *module repositories*. The file format is described in Apache Ivy documentation.

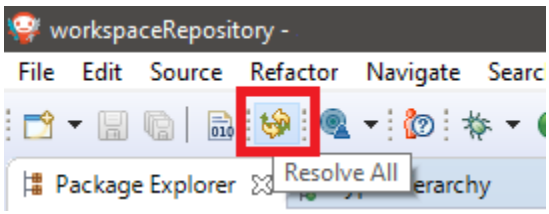
To configure MMM to a custom settings file (usually from an *offline repository*):

1. Set Settings file to a custom ivysettings.xml settings file[Page 1819, 1],
2. Click on Apply and Close button

If the workspace is not empty, it is recommended to trigger a full resolution and rebuild all the projects using this new repository configuration:

1. Clean caches
   - In the Package Explorer, right-click on a project;
   - Select Ivy > Clean all caches .
2. Resolve projects using the new repository

   To resolve all the workspace projects, click on the Resolve All button in the toolbar:



   To only resolve a subset of the workspace projects:
   - In the Package Explorer, select the desired projects,
   - Right-click on a project and select Ivy > Clean all caches .
3. Trigger Add-On Library processors for automatically generated source code
   - Select Project > Clean… ,
   - Select Clean all projects ,
   - Click on Clean button.

Options

Options can be used to parameterize a *module description file* or a *settings file*. Options are declared as key/value pairs in a standard Java properties file, and are expanded using the ${my_property} notation.

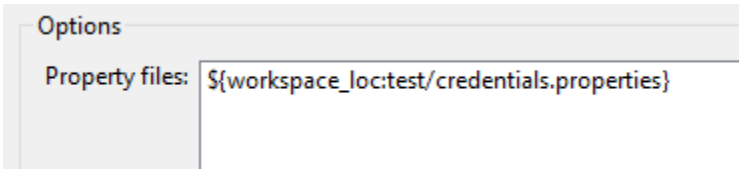
A typical usage in a *settings file* is for extracting repository server credentials, such as HTTP Basic access authentication:

1. Declare options in a properties file

credentials.properties

```
# User specific credentials
artifactory.username=myusername
artifactory.password=AKCKLzp2JHRLDyFvmTPMXocXiiU1Cna47ei9UcC9iE65UdgJrJu24ZTYieX9CwwMa3WYkjCD9

```

2. Register this property file to MMM options



3. Use this option in a *settings file*

```
<credentials host="artifactory.corp" realm="Artifactory Realm" username="${artifactory.username}" passwd="${artifactory.password}" />
```

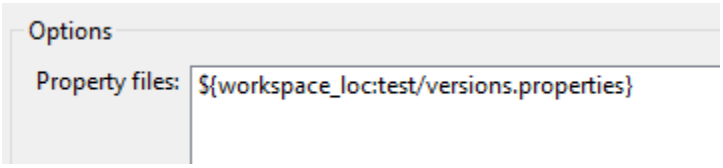
A typical usage in a *module description file* is for factorizing dependency versions across multiple modules projects:

1. Declare an option in a properties file

versions.properties

```
# Specify the EDC version used in this workspace
edc.version=1.3.0

```

2. Register this property file to MMM options



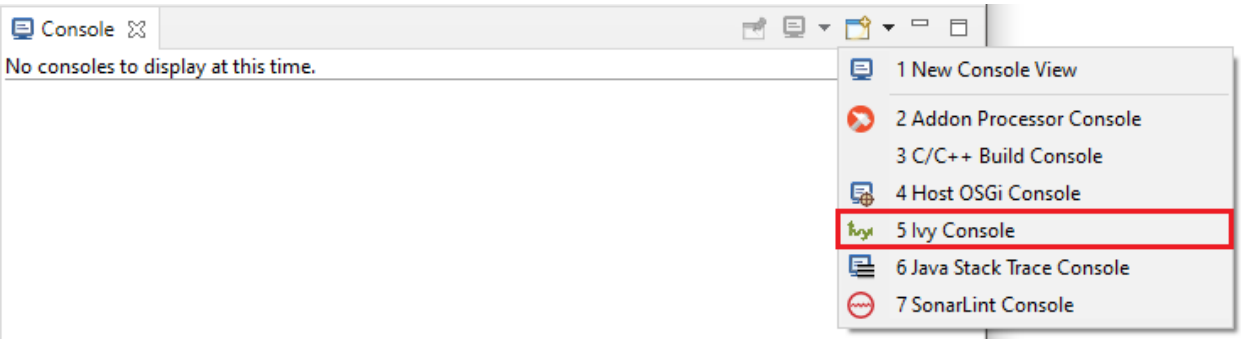
3. Use this option in a *module description file*

```
22    <dependencies>
23        <!--
24            Use the EDC version defined by MMM configuration
25        -->
26        <dependency org="ej.api" name="edc" rev="${edc.version}" />
27    </dependencies>
28 </ivy-module>
```

Resolution Logs

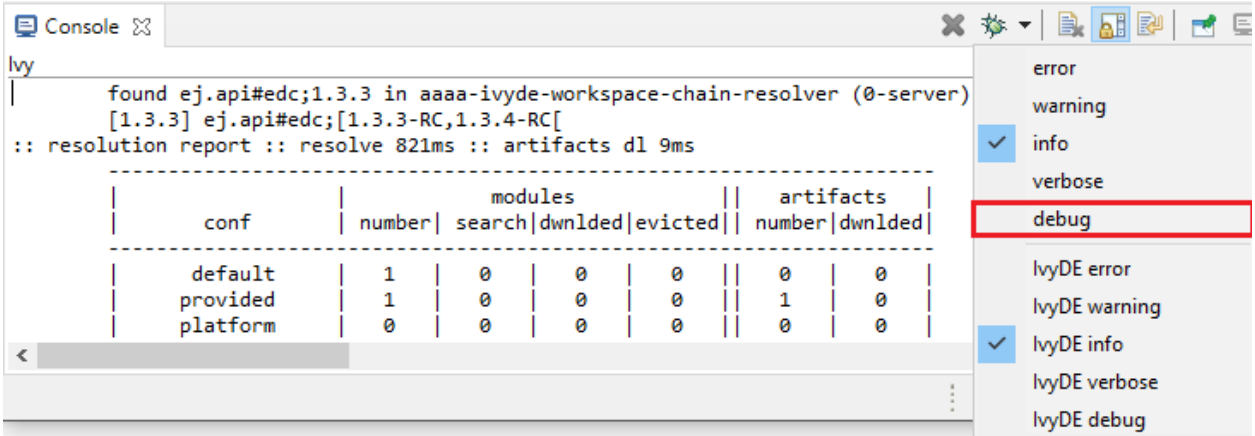
Resolution logs of module projects imported in the workspace are available from the console view:

- Select `Windows` > `Show View` > `Console` ,
- In the Console view, click on the console window icon and select `Ivy console` :



To enable the verbose mode:

- In the `Ivy console` view, click on the debug icon and select *debug* instead of *info* (defaults):



This triggers the full workspace resolution with verbose mode enabled.

Module Build

In the SDK, the build of a MicroEJ module project can be started as follows:

- In the *Package Explorer*, right-click on the project,
- Select Build Module .

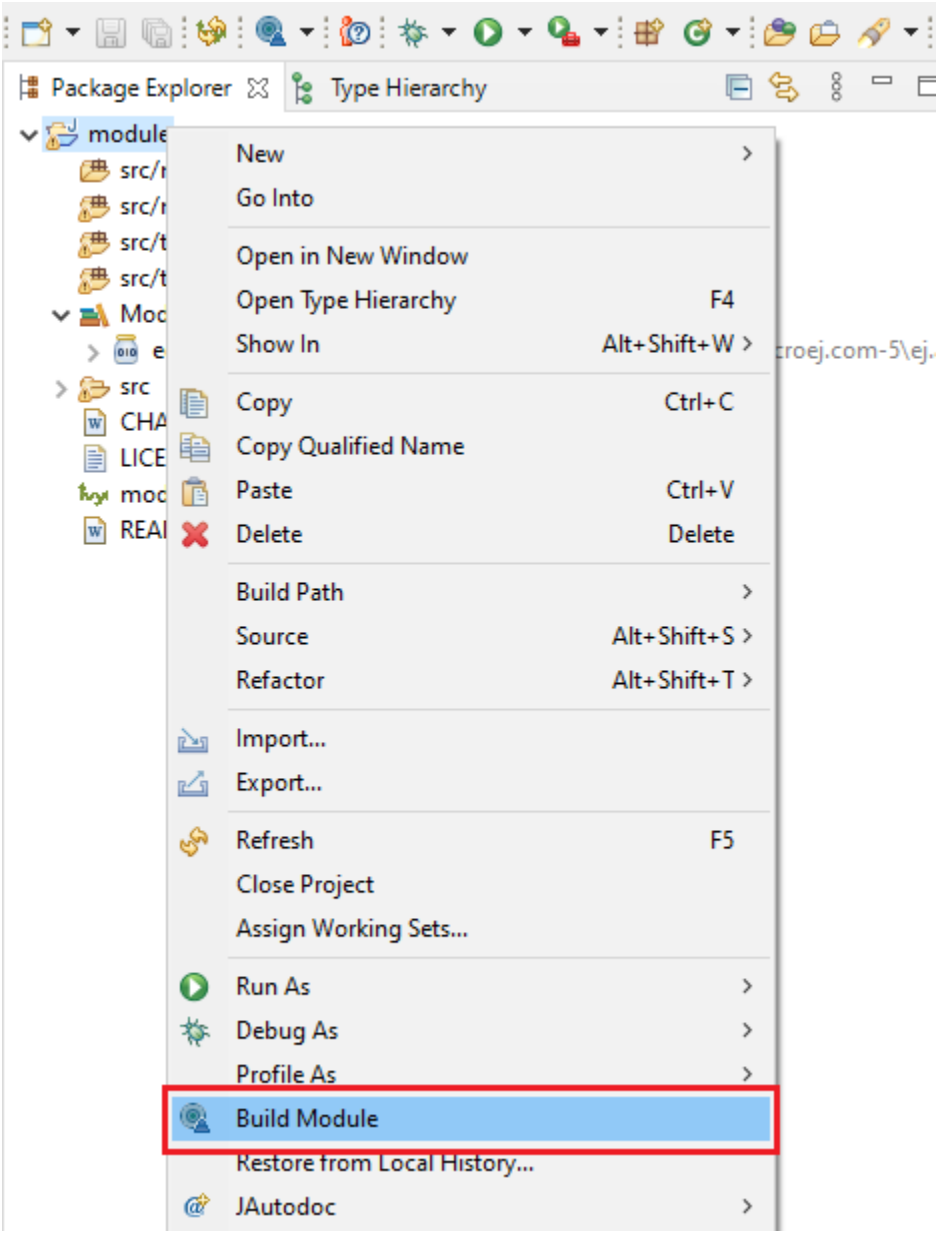


Fig. 404: Module Build

The build of a module can take time depending on

- the *module nature* to build,

- the number and the size of module dependencies to download,
- the repository connection bandwidth, …

The module build logs are redirected to the integrated console.

Alternatively, the build of a MicroEJ module project can be started from the build history:

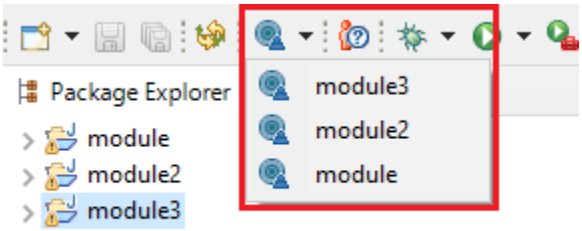


Fig. 405: Module Build History

Build Kit

The Module Manager Build Kit is a consistent set of tools, scripts, configuration and artifacts required for building modules in command-line mode. Starting from SDK 5.4.0, it also contains a *Command Line Interface* (CLI). The Build Kit allows to work in headless mode (e.g. in a terminal) and to build your modules using a Continuous Integration tool.

The Build Kit is bundled with the SDK and can be exported using the following steps:[2]

- Select File > Export > MicroEJ > Module Manager Build Kit ,
- Choose an empty Target directory ,
- Click on the Finish button.

Once the Build Kit is fully exported, the directory content shall look like:

```
/
  bin
    mmm
    mmm.bat
  conf
  lib
  microej-build-repository
    ant-contrib
    com
    ...
    ivysettings.xml
  microej-module-repository
    ivysettings.xml
  release.properties
```

- Add the bin directory of the Build Kit directory to the PATH environment variable of your machine.

[2] If using SDK versions lower than 5.4.0 , please refer to the following section.

- Make sure the JAVA_HOME environment variable is set and points to a JRE/JDK installation or that java executable is in the PATH environment variable (Java 8 is required)
- Confirm that the installation works fine by executing the command mmm --version . The result should display the MMM CLI version.

  The mmm tool can run on any supported *Operating Systems*:

- on Windows, either in the command prompt using the Windows batch script mmm.bat or in MinGW environments such as Git BASH using the bash script mmm .
- on macOS and Linux distributions using the bash script mmm .

> **Warning:** When copying a build kit from one system to another, make sure to delete the cache of the buildtype repository (/microej-build-repository/cache) which may contain absolute paths.

The build repository (microej-build-repository directory) contains scripts and tools for building modules. It is specific to a SDK version and shall not be modified by default.

The module repository (microej-module-repository directory) contains a default *Settings File* for importing modules from *Central Repository* and this local repository (modules that are locally built will be published to this directory). You can override with custom settings or by extracting an *offline repository*.

To go further with headless builds, please consult *the next chapter* for command line builds, and this *tutorial* to setup MicroEJ modules build in continuous integration environments.

Command Line Interface

Starting from version 5.4.0 , the SDK provides a Command Line Interface (CLI). Please refer to the *Build Kit* section for installation details.

The following operations are supported by the MMM CLI:

- creating a module project
- cleaning a module project
- building a module project
- running a MicroEJ Application project on the Simulator
- publishing a module in a module repository

Usage

In order to use the MMM CLI for your project:

- go to the root directory of your project
- run the following command

```
mmm [COMMAND] [OPTION]...
```

where COMMAND is the command to execute (for example mmm build). The available commands are:

- help : display help information about the specified command

- init : create a new project
- clean : clean the project
- build : build the project
- publish : build the project and publish the module
- run : run the MicroEJ Application project on the Simulator

The available options are:

- --help (-h): show the help message and exit
- --version (-V): print version information and exit
- --build-repository-settings-file (-b): path of the Ivy settings file for build scripts and tools. Defaults to ${CLI_HOME}/microej-build-repository/ivysettings.xml .
- --module-repository-settings-file (-r): path of the Ivy settings file for modules. Defaults to ${CLI_HOME}/microej-module-repository/ivysettings.xml .
- --ivy-file (-f): path of the project's Ivy file. Defaults to ./module.ivy .
- --verbose (-v): verbose mode. Disabled by default. Add this option to enable verbose mode.
- -Dxxx=yyy : any additional option passed as system properties.

When no command is specified, MMM CLI executes Easyant with custom targets using the --targets (-t) option (defaults to clean,verify).

Shared configuration

In order to share configuration across several projects, these parameters can be defined in the file ${user.home}/.microej/.mmmconfig . This file uses the TOML format. Parameters names are the same than the ones passed as system properties, except the character _ is used as a separator instead of - . The parameters defined in the [options] section are passed as system properties. Here is an example:

```
build_repository_settings_file = "/home/johndoe/ivy-configuration/ivysettings.xml"
module_repository_settings_file = "/home/johndoe/ivy-configuration/ivysettings.xml"
ivy_file = "ivy.xml"

[options]
my.first.property = "value1"
my.second.property = "value2"
```

Warning:

- TOML values must be surrounded with double quotes
- Backslash characters (\ ) must be doubled (for example a Windows path C:\\Users\\johndoe\\ivysettings.xml)

Command line options take precedence over those defined in the configuration file. So if the same option is defined in both locations, the value defined in the command line is used.

Commands

init

The command `init` creates a new project (executes Easyant with `skeleton:generate` target). The skeleton and project information must be passed with the following system properties:

- `skeleton.org` : organisation of the skeleton module. Defaults to `com.is2t.easyant.skeletons` .
- `skeleton.module` : name of the skeleton module. Mandatory, defaults to `microej-javalib` .
- `skeleton.rev` : revision of the skeleton module. Mandatory, defaults to `+` (meaning the latest released version).
- `project.org` : organisation of the project module. Mandatory, defaults to `com.mycompany` .
- `project.module` : name of the project module. Mandatory, defaults to `myproject` .
- `project.rev` : revision of the project module. Defaults to `0.1.0` .
- `skeleton.target.dir` : relative path of the project directory (created if it does not exist). Mandatory, defaults to the current directory.

For example

```
mmm init -Dskeleton.org=com.is2t.easyant.skeletons␣
↪-Dskeleton.module=microej-javalib -Dskeleton.rev=4.2.8 -Dproject.org=com.mycompany␣
↪-Dproject.module=myproject -Dproject.rev=1.0.0 -Dskeleton.target.dir=myproject
```

If one of these properties is missing, it will be asked in interactive mode:

```
$ mmm init␣
↪-Dskeleton.org=com.is2t.easyant.skeletons -Dskeleton.module=microej-javalib -Dskeleton.
↪rev=4.2.8 -Dproject.org=com.mycompany -Dproject.module=myproject -Dproject.rev=1.0.0

...

-skeleton:check-generate:
    [input] skipping input as property skeleton.org has already been set.
    [input] skipping input as property skeleton.module has already been set.
    [input] skipping input as property skeleton.rev has already been set.
    [input] The path␣
↪where the skeleton project will be unzipped [/home/tdelhomenie/microej/working/skeleton]
```

To force the non-interactive mode, the property `skeleton.interactive.mode` must be set to `false` . In non-interactive mode the default values are used for missing non-mandatory properties, and the creation fails if mandatory properties are missing.

```
$ mmm init -Dskeleton.org=com.
↪is2t.easyant.skeletons -Dskeleton.module=microej-javalib -Dskeleton.rev=4.2.8 -Dproject.
↪org=com.mycompany -Dskeleton.target.dir=myproject -Dskeleton.interactive.mode=false

...

* Problem Report:

expected property 'project.module': Module name of YOUR project
```

clean

The command `clean` cleans the project (executes Easyant with `clean` target). For example

```
mmm clean
```

cleans the project.

build

The command build builds the project (executes Easyant with clean,verify targets). For example

```
mmm build -f ivy.xml -v
```

builds the project with the Ivy file ivy.xml and in verbose mode.

publish

The command publish builds the project and publishes the module. This command accepts the publication target as a parameter, amongst these values:

- local (default value): executes the clean,publish-local Easyant target, which publishes the project with the resolver referenced by the property local.resolver in the *Settings File*.
- shared : executes the clean,publish-shared Easyant target, which publishes the project with the resolver referenced by the property shared.resolver in the *Settings File*.
- release : executes the clean,release Easyant target, which publishes the project with the resolver referenced by the property release.resolver the *Settings File*.

For example

```
mmm publish local
```

builds the project and publishes the module using the local resolver.

run

The command run runs the application on the Simulator (executes Easyant with compile, simulator:run targets). It has the following requirements:

- to run on the Simulator, the project must be configured with one of the following *Module Natures*:
  - *Sandboxed Application*
  - *Standalone Application*
  - *Add-On Library*
    - the property application.main.class must be set to the Fully Qualified Name of the application main class (for example com.mycompany.Main)
    - a MicroEJ Platform must be provided (see *Select a VEE Port* section)
    - *Standalone Application Options* must be defined using properties file under in the build directory (see *Using a Properties File* section)
    - the module must have been built once before running the Simulator. So the mmm build command must be executed before running the Simulator the first time or after a project clean (mmm clean command).

Note: The next times, it is not required to rebuild the module if source code files have been modified. The content of src/main/java and src/main/resources folders are automatically compiled by mmm run command before running the Simulator.

For example

```
mmm run -D"platform-loader.target.platform.file"="/path/to/the/platform.zip"
```

runs the application on the given platform.

The Simulator can be launched in debug mode by setting the property `execution.mode` of the application file `build/commons.properties` to `debug`:

```
execution.mode=debug
```

The debug port can be defined with the property `debug.port`. Go to *Simulator Debug options section* for more details.

help

The command `help` displays the help for a command. For example

```
mmm help run
```

displays the help of the command `run`.

Build System Options

MMM allows to modify the behavior of a build via System options. These options must be passed as system properties, using *CLI* `-D` option or via the *SDK Configuration options*. MMM provides the following options:

- `easyant.debug.port` : defines the debug port and triggers the debug mode for the build execution.

Meta Build

A Meta Build is a module allowing to build other modules. It is typically used in a project containing multiple modules. The Meta Build module serves as an entry point to build all the modules of the project.

Meta Build creation

- In the SDK, select File > New > Module Project .

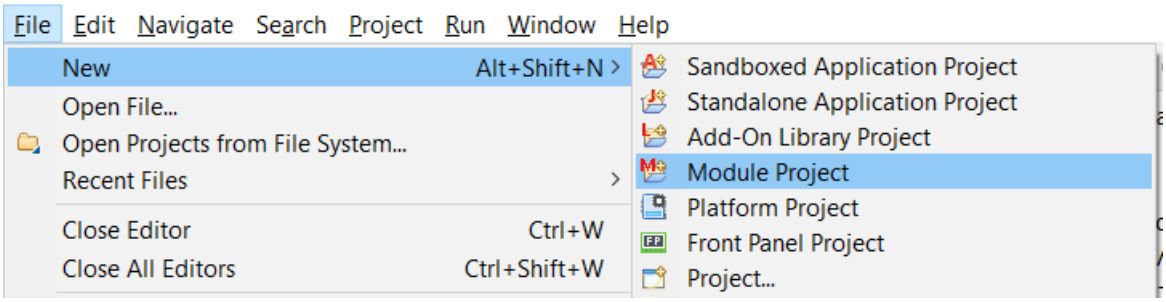


Fig. 406: New Meta Build Project

- Fill in the fields Project name , Organization , Module and Revision , then select the Skeleton named microej-meta-build
- Click on Finish . A template project is automatically created and ready to use.

Meta Build configuration

The main element to configure in a meta build is the list of modules to build. This is done in 2 files, located at the root folder:

- public.modules.list which contains the list of the modules relative paths to build and publish.
- private.modules.list which contains the list of the modules relative paths to build. These modules are not published but only stored in a private and local repository in order to be fetched by the public modules.

The format of these files is a plain text file with one module path by line, for example:

```
module1
module2
module3
```

These paths are relative to the meta build root folder, which is set by default to the parent folder of the meta build module (..). For this reason, a meta build module is generally created at the same level of the other modules to build. Here is a typical structure of a meta build:

```
/
  module1
    ...
    module.ivy
  module2
    ...
    module.ivy
  module3
    ...
    module.ivy
  metabuild
    private.modules.list
    public.modules.list
    module.ivy
```

The modules build order is calculated based on the dependency information. If a module is a dependency of another module, it is built first.

For a complete list of configuration options, please refer to *Meta Build Module Nature* section.

Troubleshooting

Unresolved Dependency

If the following message appears when resolving module dependencies:

```
:: problems summary ::
:::: WARNINGS
   module not found: com.mycompany#mymodule;[M.m.p-RC,M.m.(p+1)-RC[

            ::::::::::::::::::::::::::::::::::::::::::::::
```

(continues on next page)

(continued from previous page)

```
        ::          UNRESOLVED DEPENDENCIES         ::

        ::::::::::::::::::::::::::::::::::::::::::::::

        :: com.mycompany#mymodule;[M.m.p-RC,M.m.(p+1)-RC[: not found

        ::::::::::::::::::::::::::::::::::::::::::::::
```

First, check that either a released module com.mycompany/mymodule/M.m.p or a snapshot module com.mycompany/mymodule/M.m.p-RCYYYYMMDD-HHMM exists in your module repository.

- If the module does not exist,
  - if it is declared as a *direct dependency*, the module repository is not compatible with your source code. You can either check if an other module version is available in the repository or add the missing module to the repository.
  - otherwise, this is likely a missing transitive module dependency. The module repository is not consistent. Check the module repository settings file and that consistency check has been enabled during the module repository build (see *Configure Consistency Check*).
- If the module exists, this may be either a configuration issue or a network connection error. We have to find the cause in the resolution logs.

Note:

The activation of the verbose mode depends on how the resolution has been launched:

- if the error occurs during workspace resolution, configure the verbose mode of *resolution logs*,
- if the error occurs while building a module from workspace, check the verbose mode option in *preferences page*,
- if the error occurs while building a module from command line, set the verbose mode option in *command line options*.

For URL repositories, find:

```
trying https:/
↪/[MY_REPOSITORY_URL]/[MY_REPOSITORY_NAME]/com.mycompany/mymodule/
tried https:/
↪/[MY_REPOSITORY_URL]/[MY_REPOSITORY_NAME]/com.mycompany/mymodule/
```

For filesystem repository, find:

```
trying [MY_REPOSITORY_PATH]/com.mycompany/mymodule/
tried [MY_REPOSITORY_PATH]/com.mycompany/mymodule/
```

If your module repository URL or filesystem path does not appear, check your *settings file*. This is likely a missing resolver.

Otherwise, if your module repository is an URL, this may be a network connection error between MMM (the client) and the module repository (the server). First, check for *Invalid Certificate* issue.

Otherwise, the next step is to debug at the HTTP level:

```
HTTP response status: [RESPONSE_
↪CODE] url=https://[MY_REPOSITORY_URL]/com.mycompany/mymodule/
CLIENT␣
↪ERROR: Not Found url=https://[MY_REPOSITORY_URL]/com.mycompany/mymodule/
```

Depending on the HTTP error code:

- 401 Unauthorized : check your settings file credentials configuration.
- 404 Not Found : add the following options to log raw HTTP traffic:

```
-Dorg.apache.commons.
↪logging.Log=org.apache.commons.logging.impl.SimpleLog -Dorg.apache.commons.logging.
↪simplelog.showdatetime=true -Dorg.apache.commons.logging.simplelog.log.org.apache.
↪http=DEBUG -Dorg.apache.commons.logging.simplelog.log.org.apache.http.wire=ERROR
```

Particularly, Ivy requires the HTTP HEAD request which may be disabled by some servers.

Invalid Certificate

If the following message appears when resolving module dependencies:

```
HttpClientHandler: sun.security.validator.ValidatorException:␣
↪PKIX path building failed: sun.security.provider.certpath.SunCertPathBuilderException:␣
↪unable to find valid certification path to requested target url=[artifactory address]
```

This can be raised in several cases, such as:

- an artifact repository configured in the MicroEJ Module Manager settings using a self-signed SSL certificate or a SSL certificate not trusted by the JDK.
- the requests to an artifact repository configured in the MicroEJ Module Manager settings are redirected to a proxy server using a SSL certificate not trusted by the JDK.

  In all cases, the SSL certificate (used by the artifact repository server or the proxy) must be added to the JDK trust store that is running MicroEJ Module Manager. Ask your System Administrator, or retrieve the SSL certificate and add it to the JDK trust store:

- on Windows

1. Install Keystore Explorer.
2. Start Keystore Explorer, and open file [JRE_HOME]/lib/security/cacerts or [JDK_HOME]/jre/lib/security/cacerts with the password changeit . You may not have the right to modify this file. Edit rights if needed before opening it or open Keystore Explorer with admin rights.
3. Click on Tools , then Import Trusted Certificate .
4. Select your certificate.
5. Save the cacerts file.

- on Linux/macOS

1. Open a terminal.
2. Make sure the JDK's bin folder is in the PATH environment variable.
3. Execute the following command:

```
keytool -importcert -v -noprompt -trustcacerts -alias myAlias␣
↪-file /path/to/the/certificate.pem -keystore /path/to/the/truststore -storepass changeit
```

If the problem still occurs, set the `javax.net.debug` property to `all` to enable SSL protocol traces:

- when using the MMM CLI, add the property in the command line with: `-Djavax.net.debug=all`
- when using the `Build Module` button in the SDK, add the property in the MicroEJ Module Manager options as described in the section *Options*
- when resolving the dependencies on a project in the SDK with the button `Ivy` > `Resolve`, add the following line at the end of the file `MicroEJ-SDK.ini` located at the root of the SDK installation:

```
-Djavax.net.debug=all
```

and start the SDK from a terminal.

In all cases, such logs should appear in the terminal or in the SDK console:

```
...
javax.net.ssl|DEBUG|01|main|2022-09-09 18:22:20.
↪828 CEST|SSLContextImpl.java:428|System property jdk.tls.client.cipherSuites is set to 'null'
javax.net.ssl|DEBUG|01|main|2022-
↪09-09 18:22:20.871 CEST|SSLCipher.java:464|jdk.tls.keyLimits: entry = AES/GCM/
↪NoPadding KeyUpdate 2^37. AES/GCM/NOPADDING:KEYUPDATE = 137438953472
javax.net.ssl|DEBUG|01|main|2022-09-09 18:22:20.892 CEST|SSLContextImpl.java:402|Ignore␣
↪disabled cipher suite: TLS_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA
...
```

There should be a trace at the beginning which indicates the path of the truststore used by the JDK:

```
javax.net.ssl|FINE|01|main|2022-
↪09-05 14:34:38.631 CEST|TrustStoreManager.java:112|trustStore is: /path/to/the/truststore
```

The error very probably occurs during the handshake phase of the SSL negotiation. There should be the following trace before the error:

```
Consuming server Certificate handshake message
```

The traces below this one indicates the SSL certificate (or the SSL certificates chain) presented by the server. This certificate or one of the root or intermediate certificates must be added in the JDK truststore as explained previously.

Target “simulator:run” does not exist

If the following message appears when executing the `mmm run` command:

```
* Problem Report:

Target "simulator:run" does not exist in the project "my-app".
```

it means that the command `run` is not supported by the build type declared by your module project. Make sure it is one of the following ones:

- build-application , with version 7.1.0 or higher
- build-microej-javalib , with version 4.2.0 or higher
- build-firmware-singleapp , with version 1.3.0 or higher

Could not load SWT library

Trying
to use MMM Build Kit on headless system may lead to this error when building a VEE Port.
To fix this on Ubuntu (tested on 22.04), install the libwebkit2gtk-4.0-37 package.

Note: Also see our full Docker image configuration: https://github.com/MicroEJ/Tool-SDK-Docker/blob/67a9f4397a9c1d5608a244e2778c0cfecc5c6113/5.8.2-jdk11/Dockerfile

systemmicroui.xml:47: Terminated with errors

Trying to build
a VEE Port may lead to an "Internal limits reached" for which the log file contains this error.
This is caused
by using an incompatible JDK distribution (such as openjdk-11-jdk Ubuntu distribution).
To fix this, change your JDK distribution. See *recommended JDK distributions*.

Former SDK Versions (lower than 5.2.0)

This section describes MMM configuration elements for SDK versions lower than 5.2.0 .

New MicroEJ Module Project

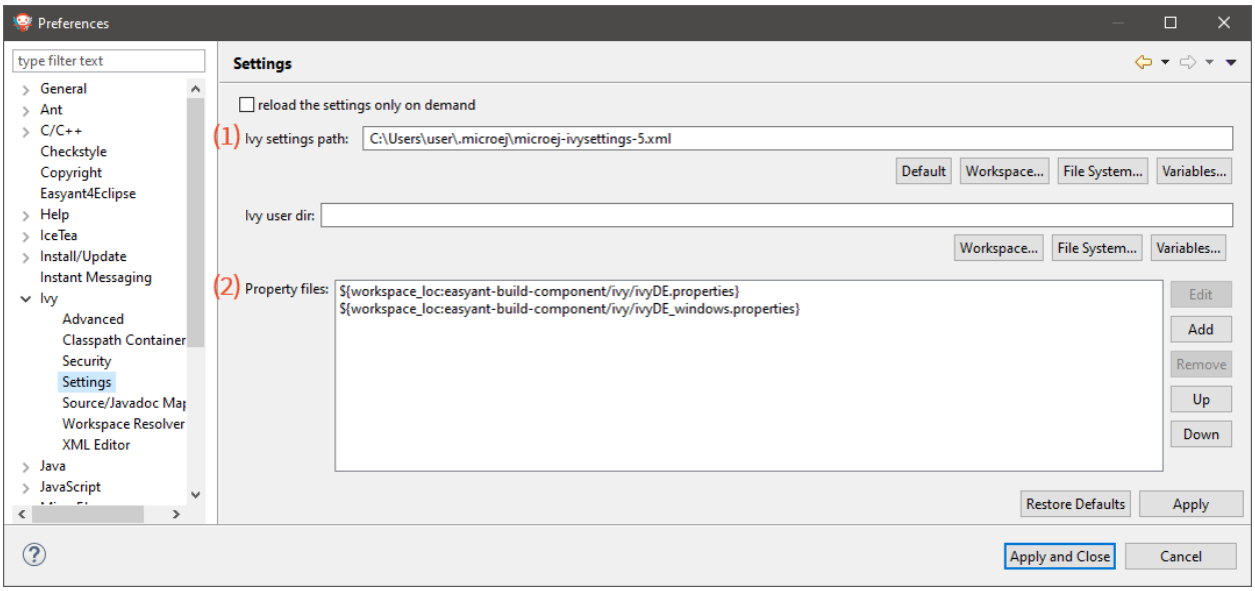
The New MicroEJ Module Project wizard is available at File > New > Project… , EasyAnt > EasyAnt Project .

Preferences Pages

MMM Preferences Pages are located in two dedicated pages. The following pictures show the options mapping using the same options numbers declared in *Preferences Page*.

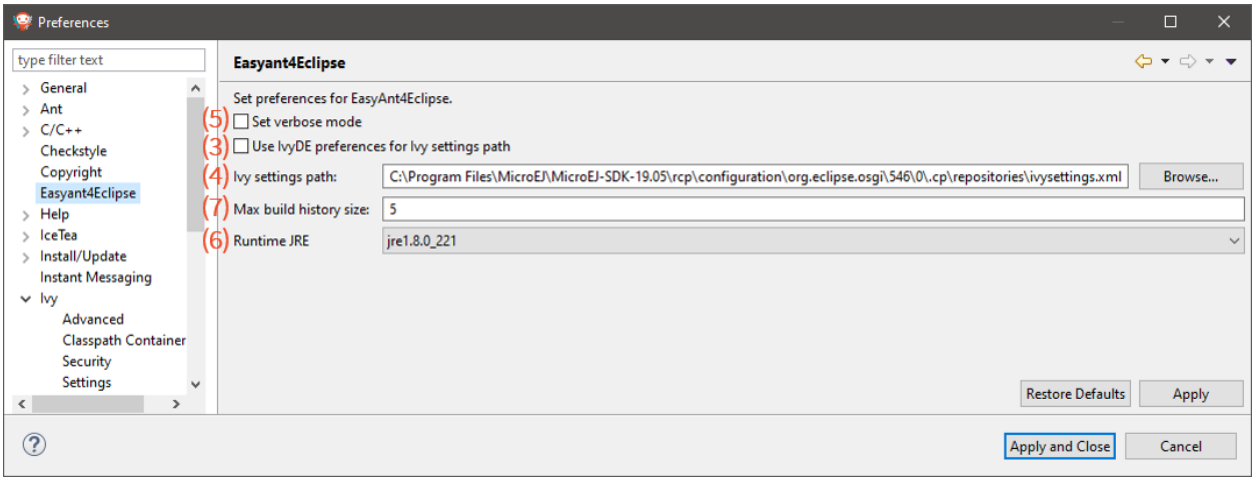
Ivy Preferences Page

The Ivy Preferences Page is available at Window > Preferences > Ivy > Settings .



Easyant Preferences Page

The Easyant Preferences Page is available at Window > Preferences > EasyAnt4Eclipse .



Build Kit

- Create an empty directory (e.g. mmm_sdk_[version]_build_kit),
- Locate your SDK installation plugins directory (by default, C:\Program Files\MicroEJ\MicroEJ SDK-[version]\rcp\plugins on Windows OS),
- Open the file com.is2t.eclipse.plugin.easyant4e_[version].jar with an archive manager,
- Extract the directory lib to the target directory,
- Open the file com.is2t.eclipse.plugin.easyant4e.offlinerepo_[version].jar with an archive manager,
- Navigate to directory repositories ,
- Extract the file named microej-build-repository.zip for SDK 5.x or is2t_repo.zip for SDK 4.1.x to the target directory.

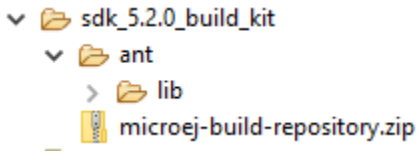
Former SDK Versions (from 5.2.0 to 5.3.x)

Build Kit

The Build Kit is bundled with the SDK and can be exported using the following steps:

- Select File > Export > MicroEJ > Module Manager Build Kit ,
- Choose an empty Target directory ,
- Click on the Finish button.

  Once the Build Kit is fully exported, the directory content shall look like:



3.11.13 VEE Port

Create a VEE Port

This section describes the steps to create a new VEE Port with the SDK, and options to connect it to an external Board Support Package (BSP) as well as a third-party C toolchain.

Note: The creation of a VEE Port with this guide requires at least the version 5.4.0 of the SDK.

Note: If you own a legacy VEE Port, you can either create your VEE Port again from scratch, or follow the *Former Platform Migration* chapter.

VEE Port Project Creation

The first step is to create a VEE Port configuration project:

- Select File > New > Project... > General > Project ,
- Enter a Project name . The name is arbitrary and can be changed later. The usual convention is [PLATFORM_NAME]-configuration ,
- Click on Finish button. A new empty project is created,
- Install the latest Platform Configuration Additions by following instructions described at https://github.com/MicroEJ/VEEPortQualificationTools/blob/master/framework/platform/README.rst.

– Files within the content-sdk-5 folder must be copied to the configuration project folder.

– Files within the content-architecture-7 must be copied to the configuration project folder only if you are using an Architecture version 7.x . If you are using an Architecture version 8.x , the files are already included and **must not** be copied.

You should get a MicroEJ Platform configuration project that looks like:

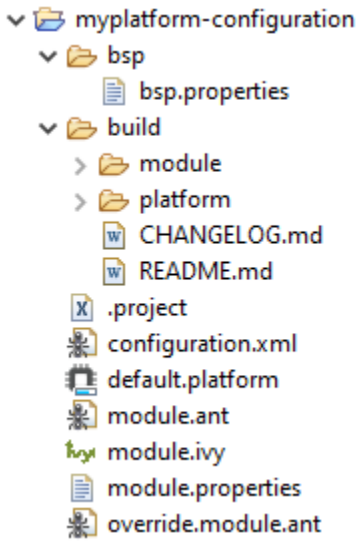


Fig. 407: MicroEJ Platform Configuration Project Skeleton

Note: The version of installed Platform Configuration Additions is indicated in the CHANGELOG file.

Architecture Selection

The next step is to select a *MicroEJ Architecture* compatible with your device instructions set and C compiler.

MicroEJ Corp. provides MicroEJ Evaluation Architectures for most common instructions sets and compilers at https://repository.microej.com/modules/com/microej/architecture.

Please refer to the chapter *Architectures MCU / Compiler* for the details of ABI and compiler options.

If the requested MicroEJ Architecture is not available for evaluation or to get a MicroEJ Production Architecture, please contact your MicroEJ sales representative or *our support team*.

Once you know which Architecture to use, add it as a dependency of the VEE Port configuration project as described below:

- Edit the *Module Description File* `module.ivy` to declare the MicroEJ Architecture dependency:

```
<dependencies>

  <dependency␣
↪org="com.microej.architecture.[ISA].[TOOLCHAIN]" name="[UID]" rev="[VERSION]">
    <artifact name="[UID]" m:classifier="[USAGE]" ext="xpf"/>
  </dependency>

</dependencies>
```

The name of the module dependency needed for your Platform can be found in the chapter *Architectures MCU / Compiler*. Check the table of your corresponding Architecture and follow the link in the `Module` column.

For example, to declare the MicroEJ Evaluation Architecture version `7.14.0` for Arm® Cortex®-M4 microcontrollers compiled with GNU CC toolchain:

```
<dependencies>

  <dependency␣
↪org="com.microej.architecture.CM4.CM4hardfp_GCC48" name="flopi4G25" rev="7.14.0">
    <artifact name="flopi4G25" m:classifier="eval" ext="xpf"/>
  </dependency>

</dependencies>
```

And the module for this Architecture is located in the *Central Repository* at https://repository.microej.com/modules/com/microej/architecture/CM4/CM4hardfp_GCC48/flopi4G25/7.14.0/.

Note: The Platform Configuration Additions allow to select the Architecture `USAGE` using the option `com.microej.platformbuilder.architecture.usage`. Edit the file `module.properties` to set the property to `prod` to use a Production Architecture and to `eval` to use an Evaluation Architecture.

Pack Import

MicroEJ Pack provides additional features on top of the MicroEJ Architecture such as Graphical User Interface or Networking.

Note: MicroEJ Packs are optional. You can skip this section if you intend to integrate MicroEJ runtime only with custom libraries.

To add a MicroEJ Pack, add it as a dependency of the VEE Port configuration project as described below:

- Edit the *Module Description File* `module.ivy` as follows:

```
<dependencies>
  <!-- MicroEJ Architecture Specific Pack -->
  <dependency org="com.microej.
↪architecture.[ISA].[TOOLCHAIN]" name="[UID]-[NAME]-pack" rev="[VERSION]"/>

  <!-- MicroEJ Generic Pack -->
  <dependency␣
↪org="com.microej.pack.[NAME]" name="[NAME]-pack" rev="[VERSION]"/>

  <!-- Legacy MicroEJ Generic Pack -->
  <dependency org="com.microej.pack" name="[NAME]" rev="[VERSION]"/>

</dependencies>
```

For example, to declare the MicroEJ Architecture Specific Pack UI version 13.0.4 for MicroEJ Architecture `flopi4G25` on Arm® Cortex®-M4 microcontrollers compiled with GNU CC toolchain:

```
<dependencies>
  <!-- MicroEJ Architecture Specific Pack -->
  <dependency org="com.
↪microej.architecture.CM4.CM4hardfp_GCC48" name="flopi4G25-ui-pack" rev="13.0.4"/>

</dependencies>
```

To declare the MicroEJ Generic Pack Bluetooth version 2.1.0:

```
<dependencies>
  <!-- MicroEJ Generic Pack  -->
  <dependency org="com.microej.pack.bluetooth" name="bluetooth-pack" rev="2.1.0"/>

</dependencies>
```

And to declare the Legacy MicroEJ Generic Pack Net version 9.2.3:

```
<dependencies>
  <!-- Legacy MicroEJ Generic Pack -->
  <dependency org="com.microej.pack" name="net" rev="9.2.3"/>

</dependencies>
```

> **Warning:** *MicroEJ Architecture Specific Packs* and *Legacy MicroEJ Generic Packs* provide Platform modules that are **not installed** by default. See *Platform Module Configuration* section for more details.

VEE Port Build

The VEE Port can be built either from the SDK or from the *MMM CLI*. To build the VEE Port from the SDK, perform a regular *Module Build*:

- Right-click on the VEE Port Configuration project,
- Select Build Module .

To build the VEE Port from the MMM CLI:

- Set the eclipse.home property to the path of your SDK, using -Declipse.home=<path> in the command line or using the *Shared configuration*.

By default, the SDK's path is one of the following directories:

- on Windows: C:\Program Files\MicroEJ\MicroEJ-SDK-<YY.MM>\rcp
- on Linux: /home/<user>/MicroEJ/MicroEJ-SDK-<YY.MM>/rcp
- on macOS: /Applications/MicroEJ/MicroEJ-SDK-<YY.MM>/rcp/MicroEJ-SDK-<YY.MM>.app/Contents/Eclipse

- From the VEE Port Configuration project, execute the command: mmm

In both cases, the build starts and the build logs are redirected to the integrated console. Once the build is terminated, you should get the following message:

```
module-platform:report:
␣
↪␣
↪[echo]␣
↪␣
↪␣
↪␣
↪␣
↪==========================================================================
  [echo]    ␣
↪ Platform has been built in this directory 'C:\tmp\mydevice-Platform-[TOOLCHAIN]-0.1.0'.
  [echo]      To import this project in your MicroEJ SDK workspace (if not already available):
  [echo]␣
↪      - Select 'File' > 'Import...' > 'General' > 'Existing Projects into Workspace' > 'Next'
  [echo]       - Check 'Select␣
↪root directory' and browse 'C:\tmp\mydevice-Platform-[TOOLCHAIN]-0.1.0' > 'Finish'
␣
↪␣
↪[echo]␣
↪␣
↪␣
↪␣
↪␣
↪==========================================================================

BUILD SUCCESSFUL
```

(continues on next page)

(continued from previous page)

```
Total time: 43 seconds
```

Then, import the VEE Port directory to your SDK workspace as mentioned in the report. You should get a ready-to-use VEE Port project in the workspace available for the MicroEJ Application project to run on. You can also check the VEE Port availability in: Window > Preferences > MicroEJ > Platforms in workspace .

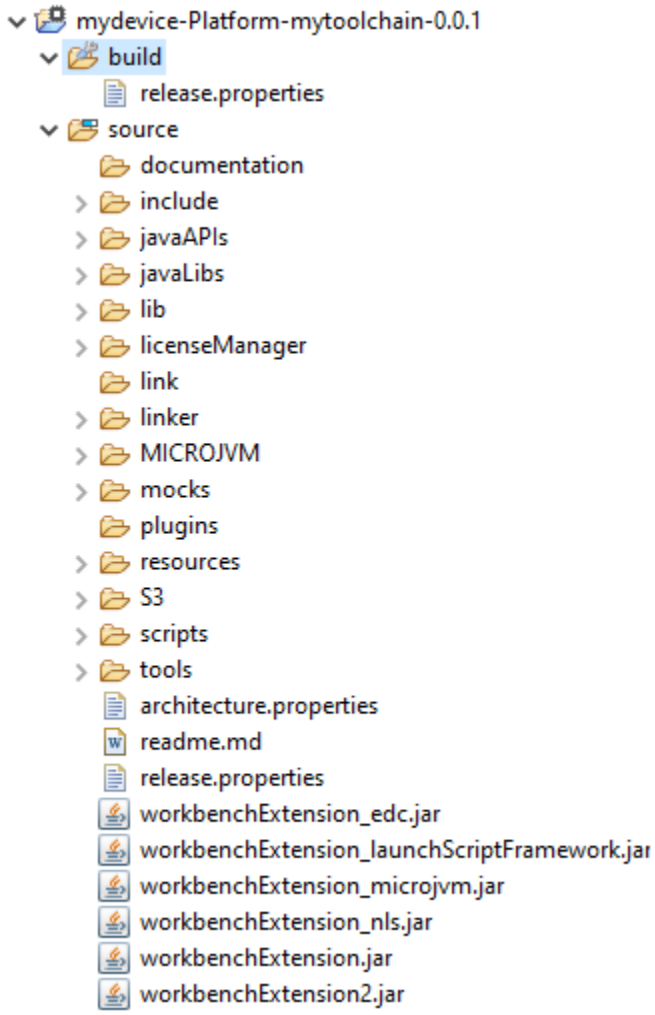


Fig. 408: VEE Port Project

This step is only required the first time the VEE Port is built, or if the VEE Port properties have changed (i.e, name, version). When the same VEE Port is built again, the Platform project should be automatically refreshed after a few seconds. In case of any doubt, right-click on the VEE Port project and select Refresh to get the new content.

Platform Module Configuration

The primary mechanism for augmenting the capabilities of a Platform is to add modules to it.

A MicroEJ module is a group of related files (Foundation Libraries, scripts, link files, C libraries, Simulator Mock, tools, etc.) that together provide all or part of a Platform capability.

A module can extend a Architecture with additional features such as:

- Runtime Capability (e.g. *Multi-Sandbox*, *External Resources Loader*) ,
- Foundation Library Implementation (e.g. *MicroUI*, *NET*),
- Simulator (e.g. *Front Panel Mock*),
- Tool (e.g. *MicroEJ Java H*).

VEE Port modules provided by *MicroEJ Generic Packs* are automatically installed during the *VEE Port build* and do not require extra configuration. They are not displayed in the VEE Port Editor.

VEE Port modules provided by *MicroEJ Architectures*, *MicroEJ Architecture Specific Packs* and *Legacy MicroEJ Generic Packs* are **not installed** by default. They must be enabled and configured using the VEE Port Editor.

Before opening the VEE Port Editor, the VEE Port must have been built once to let *MicroEJ Module Manager* resolve and download MicroEJ Architecture and Packs locally. Then import them in the SDK as follows:

- Select `File` > `Import` > `MicroEJ` > `Architectures` ,
- Browse `myplatform-configuration/target~/dependencies` folder (contains `.xpf` and `.xpfp` files once the VEE Port is built),
- Check the `I agree and accept the above terms and conditions…` box to accept the license,
- Click on `Finish` button. This may take some time.

Once imported, double-click on the `default.platform` file to open the VEE Port Editor.

From the VEE Port Editor, select the `Content` tab to access the modules selection. VEE Port modules can be selected/deselected from the `Modules` frame.

VEE Port modules are organized in groups. When a group is selected, by default all its modules are selected. To view all the modules making up a group, click on the Expand All icon on the top-right of the frame. This will let you select/deselect on a per-module basis. Note that individual module selection is not recommended and that it is only available when the module has been imported.

The description and contents of an item (group or module) are displayed next to the list when an item is selected.

All the selected VEE Port modules will be installed in the VEE Port.

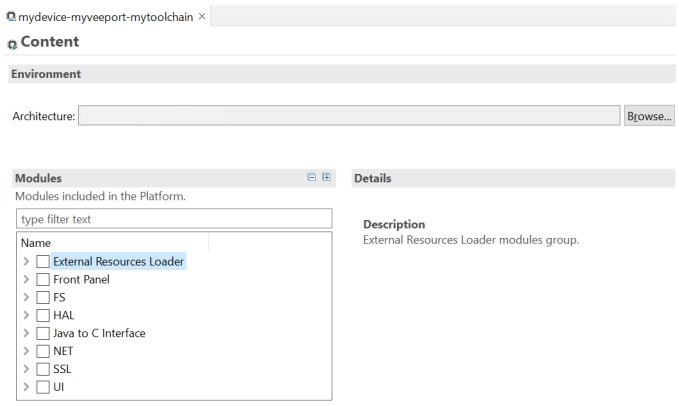


Fig. 409: VEE Port Configuration Modules Selection

Each selected VEE Port module can be customized by creating a `[module]` folder (named after the module name), next to the `.platform` file definition. It may contain:

- A `[module].properties` file named after the module name. These properties will be injected in the execution context prefixed by the module name. Some properties might be needed for the configuration of some modules. Please refer to the modules documentation for more information.
- A `bsp.xml` file which provides additional information about the BSP implementation of Low Level APIs.

  This file must start with the node <bsp> . It can contain several lines like this one: <nativeName="A_LLAPI_NAME" nativeImplementation name="AN_IMPLEMENTATION_NAME"/>

  where:

  – A_LLAPI_NAME refers to a Low Level API native name. It is specific to the MicroEJ C library which provides the Low Level API.

  – AN_IMPLEMENTATION_NAME refers to the implementation name of the Low Level API. It is specific to the BSP; and more specifically, to the C file which does the link between the MicroEJ C library and the C driver.

  These files will be converted into an internal format during the MicroEJ Platform build.

- Optional module specific files and folders

  Modifying one of these files requires to *build the Platform* again.

Note: It is possible to quickly rebuild the Platform from the Platform Editor if only the Platform module configuration has changed. Click on the `Build Platform` link on the `Overview` tab of the Platform Editor.

VEE Port Customization

The VEE Port configuration project can contain an optional `dropins` folder. The full content of this folder will be copied in the VEE Port during the build. This feature allows to add or overwrite libraries, tools, etc. into the VEE Port.

The dropins folder organization should respect the PlatVEE Portform files and folders organization. For instance, the tools are located in the sub-folder `tools` . Launch a VEE Port build without the dropins folder to see how the VEE Port files and folders are organized. Then fill the dropins folder with additional features and build again the VEE Port to get a customized VEE Port.

Files in the dropins folder have priority. If one file has the same path and name as a file already installed in the VEE Port, the file from the dropins folder will be selected first.

The VEE Port build can also be customized by updating the `configuration.xml` file next to the `.platform` file. This Ant script can extend one or several of the extension points available. By default, you should not have to change the default configuration script.

Modifying one of these files requires to *build the Platform* again.

VEE Port Publication

The publication of the built VEE Port to a *module repository* is disabled by default. It can be enabled by setting the `skip.publish` property to `false` in the `module.properties` file of the VEE Port configuration project .

The publication is kept disabled by default in the project sources because developers usually use the locally built VEE Port in the workspace. However, the publication is required in a Continuous Integration environment. This can be done by leaving the `skip.publish` property to `true` in the project sources and by overwriting it in the command launched by the Continuous Integration environment, for example:

```
mmm publish shared -Dskip.publish=false
```

If the VEE Port is configured with *Full BSP connection*, the build script can be launched to validate that the BSP successfully compiles and links before the VEE Port is published. It can be enabled by setting the `com.microej.platformbuilder.bsp.build.enabled` property to `true` in the `module.properties` file of the VEE Port configuration project (defaults to `false` if not set).

BSP Connection

In order to build the Executable of an Application, the BSP Connection must be configured. Refer to the *BSP Connection* section for more details.

Platform API Documentation

The Platform API documentation provides a comprehensive HTML Javadoc that combines all the Foundation Library APIs.

It can be built using the following steps:

- Create a new *module repository project*.
- Enable module repository javadoc generation (see *Generate Javadoc*).
- Go to your Platform build directory and browse `source/javaLibs` and `source/MICROJVM/javaLibs` directories. You will find Foundation Libraries implementations JAR files in the following pattern: `<module_name>-<major>.<minor>.jar`.

  Example: `EDC-1.3.jar`: `module_name` = `edc`, `major` = `1`, `minor` = `3`.
- For each Foundation Library your want to include,
  - Retrieve its api module in either the *Central Repository*, *Developer Repository* or your custom repository. Most of the Foundation Library APIs provided by MicroEJ are available under the `ej.api` organization.

    Example: EDC is on the Central Repository (https://repository.microej.com/modules/ej/api/edc/)
  - Get the latest available patch version corresponding to your `<major>.<minor>` version. This allows to benefit from the latest javadoc fixes and updates for the corresponding version.

    Example: `ej.api#edc#1.3.5`: `patch``=``5`
  - Declare a dependency line in the module repository.

```
<dependency conf="artifacts->*" transitive=
↪"false" org="<org>" name="<module_name>" rev="<major>.<minor>.<patch>" />
```

Example:

```
<dependency␣
↪conf="artifacts->*" transitive="false" org="ej.api" name="edc" rev="1.3.5" />
```

- Build the module repository.

  The Platform API documentation is available in `<module_repository_project>/target~/artifacts/<module_repository_name>-javadoc.zip`.

Link-Time Option

It is possible to define custom *Application options* that can be passed to the BSP through an ELF symbol defined at link-time, hence the term *link-time option*. This allows to provide configuration options to the Application developer without the need to rebuild the BSP source code.

To define a link-time option, first choose an option name with only alphanumeric characters (`[a-zA-Z][a-zA-Z0-9]*` without spaces).

Proceed with the following steps by replacing `[my_option]` with your option name everywhere:

- Create a folder inside your *VEE Port Customization* part (e.g: `[platform]-configuration/dropins/scripts/init-[my_option]`)

- Create an init script file and put it inside (e.g: `[platform]-configuration/dropins/scripts/init-[my_option]/init-[my_option].xml` file). Here is the init script file template content:

```
<project name="[my_option]-init">
  <target name="init/execution/[my_option]" extensionOf="init/execution" if="onBoard">
    <!-- Set option default value -->
    <property name="[my_option]" value="0"/>

    <!-- Create tmp dir -->
    <local name="link.files.dir"/>
    <microejtempfile deleteonexit="true" prefix="link[my_option]" property="link.files.dir"/>
    <mkdir dir="${link.files.dir}"/>
    <!-- Get tmp link file name -->
    <local name="link.[my_option]"/>
    <property name="link.[my_option]" value="${link.files.dir}/[my_option].lscf" />
    <echoxml file="${link.[my_option]}" append="false">
      <lscFragment>
        <defSymbol name="[my_option]" value="${[my_option]}" rootSymbol="true"/>
      </lscFragment>
    </echoxml>
    <!-- Add link file in linker's link files path -->
    <augment id="partialLink.lscf.path">
      <path location="${link.files.dir}"/>
      <path location="${jpf.dir}/link"/>
    </augment>
  </target>
</project>
```

- In your BSP source code, define an ELF symbol `[my_option]` can then be used inside C files in your BSP with:

```
// Declare the symbol as an extern global
extern int [my_option];

void my_func(void){
   // Get the symbol value
   int [my_option]_value = ((int)(&[my_option]));

   // Get the symbol value
   if([my_option]_value == 1){
     ...
   }
   else{
     ...
   }
}
```

> **Warning:** A Link-time option should avoid to be set to `0`. Some third-party linkers consider such symbols as undefined, even if they are declared.

Test a VEE Port

VEE Port Qualification

The SDK provides the capability to test a VEE Port. While you can create your own tests (see *Create a VEE Port Testsuite*), MicroEJ provides a set of tools and pre-defined projects aimed at simplifying the steps for validating VEE Ports in the form of the *VEE Port Qualification Tools (PQT)*.

Please refer to the VEE Port Qualification Tools README to learn how to setup and run this Qualification process.

Create a VEE Port Testsuite

A VEE Port Testsuite is composed of two projects:

- the Testsuite module: the project that contains test cases. Test cases are written in *Junit*. When this project is built, it produces a versionned library. See *Testsuite Versioning* for available Testsuite modules for the most common Packs provided by MicroEJ Corp.
- the Testsuite runner: the project that contains the configuration for its execution on a VEE Port. When this project is built, it runs the Testsuite on a Device and generates the Testsuite report.

Note: Creating a VEE Port Testsuite requires SDK `5.6.0` or higher.

Create the Testsuite Module

The Testsuite module contains the tests of the Foundation Library to be qualified.

Create the Testsuite Module Project

A new Testsuite module is created using the `microej-javaimpl` Skeleton (see *Foundation Library Implementation*).

To create the Testsuite module, click on: File > New > Project… then select MicroEJ > Module Project

Fill up the following fields of the form:

- Project name (e.g: `myFoundationLib-testsuite`).
- Organization (e.g: `com.mycompany`).
- Module (e.g: `myFoundationLib-testsuite`).
- Revision (version of your Testsuite module).
- Select the Skeleton: `microej-javaimpl` .

Then, create two test source folders:

- Right-click on your project.
- Click on: New > Source Folder .

- Fill up the Folder name field of the form with: src/test/java and for the second folder: src/test/resources .

  You should get a Foundation Library Testsuite project that looks like:

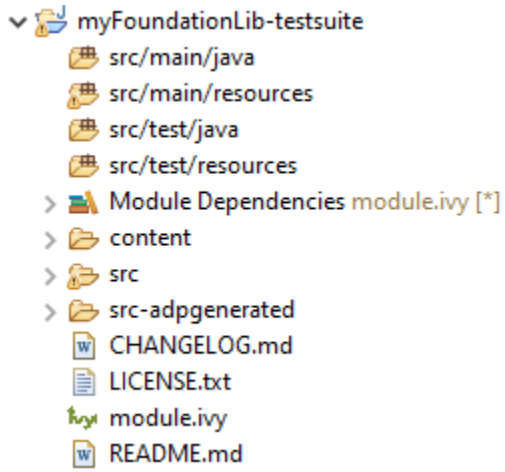


Fig. 410: Foundation Library Testsuite Project Skeleton

Your Testsuite module project is created and ready to be setup.

Configure the Testsuite Module Project

Open the module.ivy file and follow steps below:

- Edit the module ivy-module > info > ea:build node to update rip.printableName :

```
<ea:build organisation="com.is2t.
↪easyant.buildtypes" module="build-microej-javaimpl" microej.lib.name="myFoundationLib-
↪testsuite-1.0" rip.printableName="myFoundationLib Testsuite Impl" revision="5.2.+">
```

- Add the following properties in the ivy-module > info node:

```
<ea:property name="skip.test" value="set"/>
<ea:property name="target.main.classes" value="${basedir}/target~/test/classes"/>
<ea:property name="addon-processor.src.test.java.path.ref.name" value="src.java.path"/>
```

- Update the JUnit dependency to:

```
<dependency org="ej.library.test" name="junit" rev="1.7.1" conf="default;test->*"/>
```

- Add a module.ant file at the root of the Testsuite project with the following content:

```
<project>
      <target name="BuildTestTarget"␣
↪extensionOf="abstract-compile:compile-ready" depends="resources-std:copy-test-resources">
            <augment id="src.java.path">
                  <path location="${basedir}/src/test/java" />
                  <path location="${target}/adpgenerated/src-adpgenerated/junit/java"/>
```

(continues on next page)

(continued from previous page)

```
            </augment>
        </target>
</project>
```

Note: An error on module.ant file can occurred with message Target resources-std:copy-test-resources does not exist in this project . Please ignore it.

Create a New Test Case

Right click on src/test/java , then click on New > Class . Fill Name: with the MyTest and then click on Finish . Copy/paste the following example in MyTest.java file:

```
import org.junit.Assert;
import org.junit.Test;

public class MyTest {

        @Test
        public static void Test() {
                Assert.assertTrue(true);
        }
}
```

The console output on the Simulator for this test should be:

```
================ [ Initialization Stage ] ================
================ [ Launching on Simulator ] ================
OK: Test
PASSED: 1
================ [ Completed Successfully ] ================

SUCCESS
```

Build the Testsuite Module

Once the test cases are implemented, you can *build the module*. The next step is to create a Testsuite Runner. The Testsuite Runner will fetch the Testsuite Module dependency.

Create the Testsuite Runner

The Testsuite runner project contains configuration files for running a Testsuite module on a Device using a VEE Port.

Create the Testsuite Runner Project

- To create the Testsuite runner project, click on: File > New > Other... > MicroEJ > Module Project .
- Fill up the following fields of the form:
  - Project name
  - Organization
  - Module
  - Revision (version of your Testsuite module)
  - Select the Skeleton: microej-testsuite
- Inside the module.ivy file, add the dependency to the Testsuite module as following:

```
<dependency org="com.mycompany" name=
↪"myFoundationLib-testsuite" rev="0.1.0" conf="test->default;provided->provided"/>
```

- Inside the module.ant , add the following ANT target to configure trace redirection options :

```
<target name="tracefile:init" extensionOf="abstract-test:test-ready">
    <!-- Set the launch.test.trace.file when the testsuite.trace.ip properties is not set -->
    <condition property="microej.testsuite.properties.launch.test.trace.file">
        <not>
            <isset property="microej.testsuite.properties.testsuite.trace.ip" />
        </not>
    </condition>
</target>
```

- Create the file override.module.ant at the root of the project. Add the following content to configure the load of testsuite options:

```
<project name="myFoundationlib.testsuite.override" xmlns:ac="antlib:net.sf.antcontrib">
    <!-- Load options from 'local.properties' beside this file -->
    <ac:if>
        <available file="local.properties" type="file"/>
        <ac:then>
            <property file="local.properties"/>
        </ac:then>
    </ac:if>
    <!-- Load options from 'config.properties' beside this file -->
    <property file="config.properties"/>
</project>
```

- Create the following .properties files:
  - {PROJECT_LOC}/validation/microej-testsuite-common.properties : see microej-testsuite-common.properties template.
  - {PROJECT_LOC}/config.properties : see config.properties template.

Note: {PROJECT_LOC} refers here to the location of your Testsuite runner project.

Configure and Run the Testsuite

Follow the *Run the FS Testsuite on ESP32-WROVER VEE Port* tutorial to configure your VEE Port and run the Testsuite on your Device.

Run the FS Testsuite on ESP32-WROVER VEE Port

This guide describes all the steps to configure and run a *VEE Port Testsuite* on a device using the Port Qualification Tools.

The target device is the Espressif ESP32-WROVER-KIT V4.1 board and the Filesystem Testsuite for *FS* module will be used as an example.

This guide should take 1 hour to complete (excluding the VEE Port setup from the Getting Started page).

Prerequisites

This guide assumes the following:

- Good knowledge of the *MicroEJ Glossary* and *Build Workflow*.
- MICROEJ SDK distribution 20.07 or more (see *SDK Version*).
- The WROVER VEE Port has been properly setup (i.e., it can be used to generate a Mono-Sandbox Executable).

  The instructions of this page can be adapted to run the testsuite on any other VEE Port providing:

- An implementation of *LLFS: File System* version 1.0.2 in com.microej.pack#fs-4.0.3.
- A partial or full *BSP Connection*.

Note: This documentation can also be adapted to run other testsuites in addition to the Filesystem Testsuite presented here.

Introduction

This guide presents a local setup of the *VEE Port Testsuite* for the *FS* Foundation Library on a concrete device (not on Simulator).

In essence, a Foundation Library provides an API to be used by an Application or an Add-On Library.

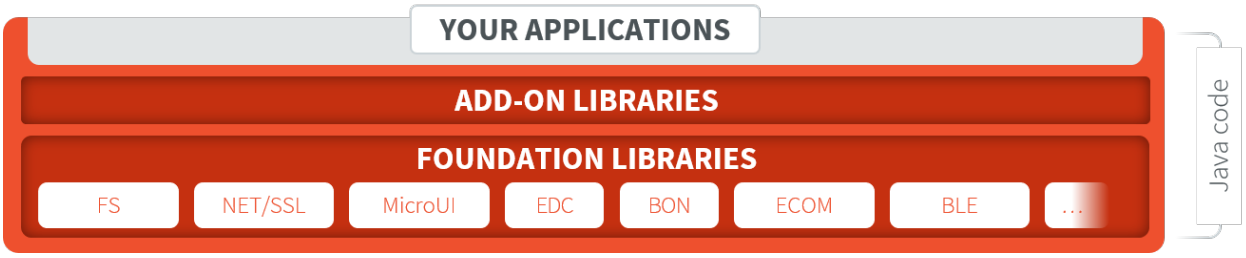


Fig. 411: MicroEJ Foundation Libraries, Add-On Libraries and Application

For example, the Java file system API java.io.File is provided by the MicroEJ Foundation Library named FS. The Abstraction Layer of each Foundation API must be implemented in C in the Board Support Package. The Testsuite is used to validate the C code implementation of the Abstraction Layer.

Import the Testsuite

Follow these steps to import the Filesystem Testsuite into the workspace from the Port Qualification Tools:

- Clone or download the Port Qualitification Tools project 2.3.0.
- Select File > Import... .
- Select Existing Projects into Workspace .
- Set Select the root directory to the directory tests/fs in the Port Qualification Tools fetched in the previous step.
- Ensure Copy projects into workspace is checked.
- Click on Finish .

  The project java-testsuite-fs should now be available in the workspace.

Configure the Testsuite

Select the Testsuite Version

For a given Foundation Library version, a specific Testsuite version should be used to validate the Abstraction Layer implementation. Please refer to *Testsuite Versioning* to determine the correct Testsuite version to use.

On the WROVER VEE Port, the FS Testsuite version to use is specified in {VEE Port}-configuration/testsuites/fs/README.md . The Test Suite version must be set in the module.ivy of the java-testsuite-fs project (e.g. java-testsuite-fs/module.ivy). For example:

```
<dependency org="com.microej.pack.fs" name="fs-testsuite" rev="3.0.3"/>
```

Configure the VEE Port BSP Connection

Several properties must be defined depending on the type of BSP Connection used by the VEE Port.

For a Application, these properties are set using the launcher of the application. For a Testsuite, the properties are defined in a file named `config.properties` in the root folder of the Testsuite. For example, see this example of config.properties file.

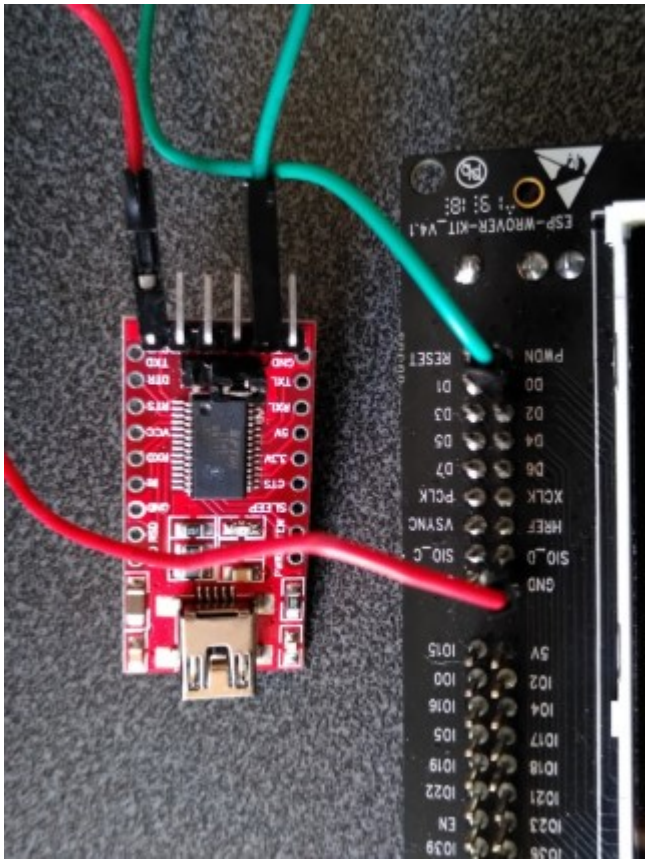
See *BSP Connection* for an explanation of the properties. See the comments in the file for a details description of each properties. The `microej.testsuite.properties.deploy.*` and `target.platform.dir` properties are required.

Configure Execution Trace Redirection

When the Testsuite is executed, the Testsuite Engine must read the trace to determine the result of the execution. To do that, we will use the *Serial to Socket Transmitter* tool to redirect the execution traces dumped to a COM port.

The WROVER VEE Port used in this documentation is particular because the UART port is already used to flash the device. Thus, a separate UART port must be used for the trace output.

This VEE Port defines the option `microej.testsuite.properties.debug.traces.uart` to redirect traces from standard input to UART.



See the Testsuite Configuration section of the WROVER VEE Port documentation for more details.

Start Serial To Socket

The *Serial to Socket Transmitter* tool can be configured to listen on a particular COM port and redirect the output on a local socket. The properties microej.testsuite.properties.testsuite.trace.ip and microej.testsuite.properties.testsuite.trace.port must be configured.

Follow these steps to create a launcher for Serial To Socket Transmitter:

- Select Run > Run Configurations… .
- Right-click on MicroEJ Tool > New .
- In the Execution tab:
  - Set Name to Serial To Socket Transmitter .
  - Select a VEE Port available in the workspace in Target > Platform .
  - Select Serial To Socket Transmitter in Execution > Settings .
  - Set the Output folder to the workspace.
- In the Configuration tab:
  - Set the correct COM port and baudrate for the device in Serial Options .
  - Set a valid port number in Server Options > Port . This port is the same as the one set in config.properties as microej.testsuite.properties.testsuite.trace.port .

Configure the Testsuite Specific Options

Depending on the Testsuite and the specificities of the device, various properties may be required and adjusted. See the file validation/microej-testsuite-common.properties (for example https://github.com/MicroEJ/VEEPortQualificationTools/blob/2.3.0/tests/fs/java/java-testsuite-fs/validation/microej-testsuite-common.properties) and the README of the Testsuite for a description of each property.

On the WROVER VEE Port, the configuration files config.properties and microej-testsuite-common.properties are provided in {VEE Port}-configuration/testsuites/fs/ .

In config.properties , the property target.platform.dir must be set to the absolute path to the VEE Port. For example C:/P0065_ESP32-WROVER-Platform/ESP32-WROVER-Xtensa-FreeRTOS-platform/source .

Run the Testsuite

To run the Testsuite, right-click on the Testsuite module and select Build Module.

Configure the Tests to Run

It is possible to exclude some tests from being executed by the Test Suite Engine.

To speed-up the execution, let's configure it to run only a small set of tests. In the following example, only the classes that match TestFilePermission are executed. This configuration goes into the file config.properties in the folder of the testsuite.

```
# Comma separated list of patterns of files that must be included
# test.run.includes.pattern=**/Test*.class
test.run.includes.pattern=**/TestFilePermission*.class
# Comma separated list of patterns of files that must be excluded (defaults to inner classes)
test.run.excludes.pattern=**/*$*.class
```

Several reasons might explain why to exclude some tests:

- **Iterative development**. Test only the Abstraction Layer that is currently being developed. The full Testsuite must still be executed to validate the complete implementation.
- **Known bugs in the Foundation Library**. The latest version of the Test Suite for a given Foundation Library might contain regression tests or tests for new features. If the VEE Port doesn't use the latest Foundation Library, then it can be necessary to exclude the new tests.
- **Known bugs in the Foundation Library implementation**. The project might have specific requirements that prevent a fully compliant implementation of the Foundation Library.

Examine the Testsuite Report

Once the Testsuite is completed, open the HTML *Testsuite Report* stored in java-testsuite-fs/target~/test/html/test/junit-noframes.html.

At the beginning of the file, a summary is displayed. Below, all execution traces for each test executed are available.

If necessary, the binaries produced and ran on the device by the Test Suite Engine are available in target~/test/xml/<TIMESTAMP>/bin/<FULLY-QUALIFIED-CLASSNAME>/application.out.

The following image shows the testsuite report fully passed:

Testsuite Results:

Summary

Tests	Failures	Errors	Ignored	Tried Again	Success rate	Time
41	0	0	0	0	100.00%	2683.271

Assertions	Failures	Success	Success Rate
3310	0	3310	100.00%

Note: *failures* are anticipated and checked for with assertions while *errors* are unanticipated.

Note: *ignored* tests are executed but not counted on the success rate.

Note: *tried again* tests are executed but not counted on the success rate.

Packages

Note: package statistics are not computed recursively, they only sum up all of its testsuites numbers.

Name	Tests	Errors	Failures	Ignored	Tried Again	Time(s)	Time Stamp	H
com.microej.fs.tests	2	0	0	0	0	132.951	1599225360034	lo
com.microej.fs.tests.constructors	4	0	0	0	0	254.649	1599225493041	lo
com.microej.fs.tests.fields	2	0	0	0	0	129.236	1599225747722	lo
com.microej.fs.tests.integration	1	0	0	0	0	62.722	1599225876968	lo
com.microej.fs.tests.methods	22	0	0	0	0	1406.581	1599225939694	lo
com.microej.fs.tests.properties	1	0	0	0	0	63.021	1599227346480	lo
com.microej.fs.tests.scenarios	9	0	0	0	0	634.111	1599227409508	lo

3.11.14 How-to Guides

How To Add IAR to MICROEJ SDK Docker Image

This document presents how to create a Dockerfile with MICROEJ SDK version 5.x and Cross-platform Build Tools for Arm to build a MicroEJ application. You can use this image in your automated CI.

Prerequisites

- A recent version of IAR BXARM and its user licence.

  This training was tested with MICROEJ SDK 5.8.1-jdk11 , IAR 9.30.1 , and Docker 24.0.6 .

Create the Dockerfile

Here is our final Dockerfile. We will explain each specific step below.

```
FROM microej/sdk:5.8.1-jdk11

USER root
SHELL ["/bin/bash", "-c"]

ARG IAR_BXARM_VERSION=9.30.1
ARG IAR_BXARM_PACKAGE="bxarm-$IAR_BXARM_VERSION.deb"

COPY ressources/$IAR_BXARM_PACKAGE /tmp/$IAR_BXARM_PACKAGE
RUN apt-get update && apt-get␣
↪install sudo libsqlite3-0 libxml2 tzdata dos2unix /tmp/$IAR_BXARM_PACKAGE -y && \
   apt-get clean autoclean autoremove && rm -rf /var/lib/apt/lists/* /tmp/*.deb

ENV␣
↪PATH="/opt/iarsystems/bxarm/arm/bin/:/opt/iarsystems/bxarm/common/bin/:$PATH"
ENV IAR_LICENSE_SERVER=$IAR_LICENSE_SERVER_IP

# Set workdir
```

(continues on next page)

(continued from previous page)

```
WORKDIR ${HOME}

ADD run.sh /run.sh
RUN chmod a+x /run.sh

# Good practice, switch back to user.
USER ${user}

ENTRYPOINT ["/run.sh"]
```

1. In a new directory create a file named `Dockerfile`.
2. We use MICROEJ SDK base image, they are available on docker hub. In your Dockerfile add this code:

```
FROM microej/sdk:5.8.1-jdk11
```

3. Add IAR BXARM deb package in a directory named `resources`.
4. Add the package info to your Dockerfile (update the version with the one you want to use):

```
ARG IAR_BXARM_VERSION=9.30.1
ARG IAR_BXARM_PACKAGE="bxarm-$IAR_BXARM_VERSION.deb"
```

5. Copy the package to a temporary directory.

```
COPY ressources/$IAR_BXARM_PACKAGE /tmp/$IAR_BXARM_PACKAGE
```

6. Install this package along with any others required packages.

```
RUN apt-get update && apt-get␣
↪install sudo libsqlite3-0 libxml2 tzdata dos2unix /tmp/$IAR_BXARM_PACKAGE -y && \
apt-get clean autoclean autoremove && rm -rf /var/lib/apt/lists/* /tmp/*.deb
```

7. Set IAR path and license server address:

```
ENV␣
↪PATH="/opt/iarsystems/bxarm/arm/bin/:/opt/iarsystems/bxarm/common/bin/:$PATH"
ENV IAR_LICENSE_SERVER=$IAR_LICENSE_SERVER_IP
```

8. Finally create a `run.sh` script with the following content:

```
lightlicensemanager setup -s $IAR_LICENSE_SERVER
exec "$@"
```

3.11.15 Release Notes

Starting from SDK version `5.0.0`, Architectures are distributed separately from the Integrated Development Environment. Evaluation Architectures can be downloaded from the Architectures Repository.

The SDK is now packaged into an Eclipse P2 repository (https://repository.microej.com/p2/sdk), allowing partial updates and installation on any compatible Eclipse version. The historical version (`5`) of MicroEJ is reused for the P2 repository delivery.

MicroEJ Corp. continues to regularly build all-in-one packages, called *Distributions*, including the SDK and dedicated OS installers. This distribution has a separate versioning, which follows modern convention: [YY].[MM] .

3.11.16 SDK Distribution Changelog

[24.01] - 2024-01-31

Note: This release requires a JDK 11 and therefore an Architecture 7.17.0 or higher. Please refer to *System Requirements* for more details.

- Included *SDK 5.8.2*.
- Enabled the "Terminate and Relaunch while launching" launcher option by default when a new Workspace is created.

[23.07] - 2023-07-03

Note: This release requires a JDK 11 and therefore an Architecture 7.17.0 or higher. Please refer to *System Requirements* for more details.

- Included *SDK 5.8.0*.
- Downgraded to Eclipse version 2022-03 to fix incompatibilities of components with JDK 11.
- Fixed no JDK found error when launching the installer in the case the JDK path has not been set in the Windows registry.

[23.02] - 2022-02-28

Note: This release requires a JDK 11 and therefore an Architecture 7.17.0 or higher. Please refer to *System Requirements* for more details.

- Included *SDK 5.7.0*.
- Updated to Eclipse version 2022-12 .

[22.06] - 2022-06-29

Note: This release requires a JDK 11 and therefore an Architecture 7.17.0 or higher. Please refer to *System Requirements* for more details.

- Included *SDK 5.6.0*.
- Added support for macOS aarch64 (M1 chip).
- Updated to Eclipse version 2022-03 .

- Changed required Java Runtime to JDK 11 (JRE and other versions are not supported anymore).

[21.11] - 2021-11-15

Note: This release prepares for a future JRE 11 support. However, the only officially supported JRE version is still JRE 8. Please refer to *System Requirements* for more details.

- Included *SDK 5.5.0*.
- Updated installer to accept both JRE 8 and JRE 11.
- Fixed error `Error while loading manipulator` when installing SDK updates on MacOS.
- Updated End User License Agreement.

[21.03] - 2021-03-25

- Included *SDK 5.4.0*.
- Updated End User License Agreement.

  KNOWN ISSUES:

- The following error occurs when installing an SDK update on MacOS:

```
!MESSAGE Error while loading manipulator.
!STACK 0
java.lang.IllegalStateException: Error while loading manipulator.
    at org.eclipse.equinox.
↪internal.p2.touchpoint.eclipse.LazyManipulator.loadDelegate(LazyManipulator.java:64)
    at org.eclipse.equinox.
↪internal.p2.touchpoint.eclipse.LazyManipulator.getConfigData(LazyManipulator.java:117)
    at org.eclipse.equinox.internal.p2.touchpoint.
↪eclipse.actions.UninstallBundleAction.uninstallBundle(UninstallBundleAction.java:57)
    at org.eclipse.equinox.internal.
↪p2.touchpoint.eclipse.actions.UninstallBundleAction.execute(UninstallBundleAction.java:33)
    at org.eclipse.equinox.internal.p2.
↪engine.ParameterizedProvisioningAction.execute(ParameterizedProvisioningAction.java:42)
    at org.eclipse.equinox.internal.p2.engine.Phase.mainPerform(Phase.java:186)
    at org.eclipse.equinox.internal.p2.engine.Phase.perform(Phase.java:99)
    at org.eclipse.equinox.internal.p2.engine.PhaseSet.perform(PhaseSet.java:50)
    at org.eclipse.equinox.internal.p2.engine.Engine.perform(Engine.java:80)
    at org.eclipse.equinox.internal.p2.engine.Engine.perform(Engine.java:48)
    at org.eclipse.equinox.
↪p2.operations.ProvisioningSession.performProvisioningPlan(ProvisioningSession.java:181)
    at org.eclipse.
↪equinox.p2.operations.ProfileModificationJob.runModal(ProfileModificationJob.java:76)
    at org.eclipse.equinox.p2.operations.ProvisioningJob.run(ProvisioningJob.java:190)
    at org.eclipse.core.internal.jobs.Worker.run(Worker.java:63)
```

  The workaround is to replace `/eclipse/plugins/` by `/Eclipse/plugins/` (capital `E`) in `MicroEJ-SDK-21.03.app\Contents\Eclipse\eclipse.ini` .

- See *SDK 5.4.0 Known Issues* section

[20.12] - 2020-12-11

- Included *SDK 5.3.1*
- Disabled Java version check when updating SDK (see *known issues of SDK Distribution 20.10*)

[20.10] - 2020-10-30

- Included *SDK 5.3.0*
- Updated to Eclipse version 2020-06
- Fixed low quality MacOS SDK icons

Note: Starting with this release, only 64bits JRE are supported because 32bits JRE support has been removed since Eclipse version 2018-12 .

KNOWN ISSUES:

- Projects configured with Null Analysis must be updated to import EDC API 1.3.3 or higher in order to avoid an Eclipse JDT builder error (see also this link for more details).
- The default settings file for connecting MicroEJ Central Repository is not automatically installed. To connect to the MicroEJ Central Repository, follow the procedure:
  - For Windows, create the folder: C:\Users\%USERNAME%\.microej .
  - For Linux, create the folder: /home/$USER/.microej .
  - For macos, create the folder: /Users/$USER/.microej .
  - Download and save this file microej-ivysettings-5.xml to the previously created .microej folder.
- By default, a check is done on the JRE version required by the plugins on install/update. Since CDT requires JRE 11, it prevents to install/update a newer SDK version. The CDT documentation explains that this can be bypassed by disabling the option Windows > Preferences > Install/Update > Verify provisioning operation is compatible with currently running JRE .

[20.07] - 2020-07-28

- Included *MicroEJ SDK 5.2.0*
- Updated the default microej repository folder name (replaced SDK version by the distribution number)
- Added Dist. prefix in installer name (e.g. MicroEJ SDK Dist. 20.07) to avoid confusion between SDK distribution vs SDK version
- Updated SDK End User License Agreement
- Disabled popup window when installing a SDK update site (allow to install unsigned content by default)

[19.05] - 2019-05-17

- Included *SDK version 5.1.0*
- Updated MicroEJ icons (16x16 and 32x32)
- Updated the publisher of Windows executables (MicroEJ instead of IS2T SA.)
- Updated the JRE link to download in case the default JRE is not compatible. (https://www.java.com is deprecated)

[19.02] - 2019-02-22

- Updated to Eclipse Oxygen version 4.7.2
- Included *SDK version 5.0.1*
- Included Sonarlint version 4.0.0

3.11.17 SDK Changelog

[5.9.0] - 2024-07-23

General

- Hide MicroEJ project wizards when SDK 5 is installed in an Eclipse distribution including SDK 6 plugins.

MicroEJ Module Manager

General

- Upgraded Testsuite Engine to version 5.8.1 to separate test cases in the testsuite reports and escape console output in HTML report to prevent code injection.

Build Types

- Set default Java test compile version to 1.8 for build-std-javalib and build-microej-mock build types.
- New build types added:
  - build-application#9.3.0
  - build-firmware-singleapp#2.4.0
  - build-microej-javaimpl#5.3.0
  - build-microej-javalib#6.3.0
  - build-microej-mock#2.2.1
  - build-microej-testsuite#4.3.0
  - build-std-javalib#3.3.1

[5.8.2] - 2024-01-31

General

- Added the `--keep-going` option for the MMM Command Line Interface to continue the build of the meta-build when a subproject fails.

MicroEJ Module Manager

General

- Upgraded Front Panel plugin to version 6.3.0 to use FP framework dependency only by default.

Build Types

- Set default Java compile version to 1.8 for `build-std-javalib` build type.
- Fixed hardcoded dependency line in generated javadoc of artifacts repositories.
- Fixed incompatibility of the Artifact Checker with modules published with the SDK 6.
- New build types added:
  - build-firmware-multiapp#8.2.0
  - build-firmware-singleapp#2.3.0
  - build-std-javalib#3.3.0

[5.8.1] - 2023-09-19

General

- Fixed unreadable tooltip because of black text on black background for the VEE Ports and Architectures views.
- Fixed wrong value for the example in the StackOverflow error message in the Memory Map Analyzer plugin.
- Fixed Configuration tab content disappearing when navigating in Run Configurations.

MicroEJ Module Manager

General

- Remove legacy configuration fields for application project wizard (Application ID, Printable Name and Description).

Build Types

- New build types added:
  - None

Skeletons

- Add section in README of the build-addon-processor skeleton to document how to override a generated source file.

[5.8.0] - 2023-07-03

General

- Added improvements in Outline view and Instance Browser view (new Owner column, new filters) of the Heap Viewer.
- Enabled on/off tags in the MicroEJ Java format profile.
- Updated Code template for Widget.handleEvent to use MWT 3 API.
- Fixed default Ivy settings file not created at startup.
- Fixed topological order in Application classpath.

MicroEJ Module Manager

General

Build Types

- Added Gradle dependency line in the generated Javadoc of an artifact repository (build-artifact-repository build type).
- New build types added:
  - build-artifact-repository#3.4.0
  - build-izpack#3.3.0

Skeletons

- Removed META-INF folder from firmware-multiapp skeleton.

[5.7.0] - 2023-02-27

General

- Added latest BSD license and SDK/BSD license and deprecate ESR.
- Added the capability to resolve a Front Panel dependency as a project in the workspace, as any other module type.
- Added the capability to resolve a Front Panel Mock dependency as a project in the workspace, as any other module type.
- Added the support to fetch Maven modules from MMM projects.
- Changed the error message displayed by the Memory Map Analyzer to show the real error message.
- Fixed build error when an ADP is opened in the workspace.
- Fixed slowness issue during Ivy resolution on Windows with JDK 11.
- Fixed syntaxic coloration lost in an opened module.ivy file after an SDK restart.
- Fixed inadequate colors in editors and console in Dark theme.
- Fixed failing Ivy resolution after an SDK restart.
- Fixed the freeze of the Heap Analyzer when opening a large heap file or clicking on a large byte array.
- Fixed error when building a VEE Port using the Build Platform button in the `.platform` file.
- Fixed "Resolve Foundation Library in workspace" option unchecked after closing and re-opening the workspace.
- Fixed Addon Processor modules not resolved when opened in the workspace.

MicroEJ Module Manager

General

- Fixed release version of a runtime API module.
- Fixed build of a module that uses the `obf-proguard` plugin with JDK 11.
- Upgraded ProGuard to version 7.2.1 to support JDK 11.
- Fixed Application external jars resolution at compile time.
- Fixed resolution in workspace error depending on a Mock's name.
- Fixed error message when an Easyant target is executed in a folder that does not contain a `module.ivy` file.

Build Types

- Fixed build-std-javalib compilation with JDK 11.
- Fixed Artifact Checker's execution on build-std-javalib .
- Fixed build-artifact-repository build type which couldn't find the previous release of the repository to merge it.
- New build types added:
  - build-addon-processor#2.2.0
  - build-application#9.2.0
  - build-artifact-repository#3.3.0
  - build-custom#2.2.0
  - build-firmware-customizer#3.2.0
  - build-firmware-multiapp#8.1.0
  - build-firmware-singleapp#2.2.0
  - build-izpack#3.2.0
  - build-microej-extension#2.2.0
  - build-microej-javaapi#5.2.0
  - build-microej-javaimpl#5.2.0
  - build-microej-javalib#6.2.0
  - build-microej-mock#2.2.0
  - build-microej-ri#3.2.0
  - build-microej-testsuite#4.2.0
  - build-product-java#2.2.0
  - build-runtime-api#4.1.0
  - build-std-javalib#3.2.0

Skeletons

- Aligned Kernel APIs dependencies between runtime-api and firmware-multiapp skeletons.
- Changed default compilation level to Java 8 for Mock projects.
- Fixed Build Executable options to make the "No BSP Connection" work.

[5.6.2] - 2022-08-31

General

- Fixed error when opening some heap dump files.
- Fixed error when saving a EJF file with the Font Designer.

MicroEJ Module Manager

General

- Fixed invalid module name when using spaces in the project name.

Skeletons

- Fixed wrong package name in the class generated when creating a firmware-multiapp project.

[5.6.1] - 2022-07-08

General

- Removed check on JRE version when opening a workspace.

[5.6.0] - 2022-06-29

General

- Added support for JDK 11.
- Changed Easyant targets executed by `mmm build` from `clean,verify` to `clean,package`.
- Upgraded Front Panel plugin to version 6.1.3 to remove warning on fp framework.
- Updated Workspace settings to ignore errors in Ant build files by default.
- Fixed error when opening a Heap Dump file not part of the workspace.
- Fixed error when opening a Map file not part of the workspace.
- Removed Resources Center view.

MicroEJ Module Manager

General

- Added the capability to override module organisation/name/revision with *Build System Options*.
- Added error message when using non-supported Eclipse Link Folders.
- Updated End User License Agreement.

- Fixed MMM failure when resolving a dependency with a version containing a number with 4 digits.
- Fixed error when building a meta-build project with public sub-modules and using target verify .

Build Types

- Added support for Kernel Runtime Environments (build-firmware-multiapp , build-runtime-api and build-application).
- Added option javadoc.modules.excludes to exclude modules from Javadoc generation when building a module repository.
- New build types added:
  - build-addon-processor#2.1.0
  - build-application#9.1.0
  - build-artifact-repository#3.2.0
  - build-custom#2.1.0
  - build-firmware-customizer#3.1.0
  - build-firmware-multiapp#8.0.0
  - build-firmware-singleapp#2.1.0
  - build-izpack#3.1.0
  - build-microej-extension#2.1.0
  - build-microej-javaapi#5.1.0
  - build-microej-javaimpl#5.1.0
  - build-microej-javalib#6.1.0
  - build-microej-mock#2.1.0
  - build-microej-ri#3.1.0
  - build-microej-testsuite#4.1.0
  - build-product-java#2.1.0
  - build-runtime-api#4.0.0
  - build-std-javalib#3.1.0

Build Plugins

- Updated elf-utils plugin to load the ELF related tools from the architecture/platform.

Skeletons

- Added JUnit dependency to all Java module skeletons (including default JUnit tests pattern).
- Updated `firmware-singleapp` and `firmware-multiapp` skeletons for building the executable by default.
- Updated Sandboxed Application skeleton (`application`) to be compatible with any Kernel (based on KF `FeatureEntryPoint`).

[5.5.3] - 2022-05-03

MicroEJ Module Manager

- Fixed error `Can't parse module descriptor` when building a Module on Windows with a JDK 8.0.331+.

[5.5.2] - 2021-12-22

General

- Fixed Addon Processors of a project in a workspace being applied to others projects.

MicroEJ Module Manager

Build Plugins

- Updated Log4j in Artifact Checker and Cobertura plugins to version 2.17.0.

[5.5.1] - 2021-12-02

General

- Fixed wrong category name in `New Project` wizard.

[5.5.0] - 2021-11-15

Note: This release prepares for a future JRE 11 support. However, the only officially supported JRE version is still JRE 8. Please refer to *System Requirements* for more details.

General

- Added Add-On Processor resolution in workspace.
- Updated tools for both JRE 8 and JRE 11 compatibility.
- Fixed corrupted font file created by the Font designer when importing large number of glyphs.
- Updated Architecture version check during Pack import (greaterOrEqual instead of compatible). This allows to import *Architecture Specific Pack* and *Legacy Generic Pack* on future Architecture versions 8.x .
- Updated End User License Agreement.

MicroEJ Module Manager

- Added bin folder to .gitignore file of module natures Java project skeleton.
- Added Null Analysis configuration to artifact-checker . When building a module repository, null analysis configuration is only checked on the highest module version included in the repository.
- Added Eclipse Public License v2.0 to the list of default licenses allowed for artifact-checker .
- Clarified input messages of mmm init command.
- Updated artifact-checker plugin binding to target verify . This allow module checks to be executed on builds triggered by a pull request (no publication).
- Fixed missing artifact-checker plugin to some module natures (custom , firmware-multiapp , firmware-singleapp , microej-javaimpl , microej-mock , microej-testsuite , product-java).
- Fixed mmm run execution on a firmware-singleapp module (do not trigger the Firmware build).
- Fixed kf-testsuite plugin test project build.
- Added support of branch analysis with Sonar.
- Added ability to package private dependencies to mock module natures (configuration embedded).
- Added testsuite and javadoc plugin to firmware-singleapp module nature.
- Added ssh deployment to microej-kf-testsuite plugin.
- Updated firmware-multiapp to remove the bsp directory in Virtual Devices.
- Updated firmware-multiapp to allow Virtual Devices for launching a specific main class other than the Kernel main class. This is useful for running JUnit tests using a Virtual Device instead of a Platform.
- Updated firmware-multiapp to allow Virtual Devices for automatically launching a Sandboxed Application project in the SDK.
- Updated firmware-multiapp to automatically configure the Virtual Device Kernel UID when a Firmware is built.
- Fixed firmware-multiapp skeleton default dependencies with only modules available in MicroEJ Central Repository.
- Fixed firmware-multiapp unexpected build error when no declared pre-installed Application.

- Fixed firmware-multiapp build which may fail an unexpected Unresolved Dependencies error the first time, for Kernel APIs module dependencies (configuration kernelapi) or Virtual Device specific modules dependencies (configuration default-vd).
- Fixed firmware-multiapp unexpected build error when no Application (.wpk file) found in the dropins folder.
- Fixed firmware-multiapp unexpected build error when no declared pre-installed Application.
- Fixed firmware-singleapp and firmware-multiapp skeletons wrong package name generation for the default Main class.
- Fixed artifact-repository changelog check for modules with a snapshot version.
- New build types added:
  - build-addon-processor#2.0.0
  - build-application#9.0.0
  - build-artifact-repository#3.0.0
  - build-custom#2.0.0
  - build-firmware-customizer#3.0.0
  - build-firmware-multiapp#7.0.0
  - build-firmware-singleapp#2.0.0
  - build-izpack#3.0.0
  - build-microej-extension#2.0.0
  - build-microej-javaapi#5.0.0
  - build-microej-javaimpl#5.0.0
  - build-microej-javalib#6.0.0
  - build-microej-mock#2.0.0
  - build-microej-ri#3.0.0
  - build-microej-testsuite#4.0.0
  - build-product-java#2.0.0
  - build-runtime-api#3.0.0
  - build-std-javalib#3.0.0
  - microej-meta-build#3.0.0

[5.4.1] - 2021-04-16

Note: This release is both compatible with Eclipse version 2020-06 and Eclipse Oxygen, so it can still be installed on a previous SDK Distribution.

MicroEJ Module Manager

- Fixed missing repository configuration in artifact-repository skeleton (this configuration is required to include modules bundled in an other module repository)
- Fixed missing some old build types versions that were removed by error. (introduced in SDK 5.4.0 , please refer to the *Known Issues* section for more details)
- Fixed wrong version of module built in a meta-build (module was published with the module version instead of the snapshot version)
- Fixed code coverage analysis on source code (besides on bytecode) thanks to the property cc.src.folders (only for architectures in version 7.16.0 and beyond)
- New build types added:
  - microej-meta-build#2.0.1

[5.4.0] - 2021-03-25

Note: This release is both compatible with Eclipse version 2020-06 and Eclipse Oxygen, so it can still be installed on a previous SDK Distribution.

Known Issues

- Some older build types versions have been removed by error. Consequently, using SDK 5.4.0 , it may be not possible to build modules that have been created with an older SDK version (For example, MicroEJ GitHub code). The list of missing build types:
  - build-application 7.0.2
  - build-microej-javalib 4.1.1
  - build-firmware-singleapp 1.2.10
  - build-microej-extension 1.3.2

General

- Added MicroEJ Module Manager Command Line Interface in Build Kit
- Added ignore optional compilation problems in Addon Processor generated source folders
- Added logs to Standalone Application build indicating the mapping of Foundation Libraries to the Platform
- Updated End User License Agreement
- Added the latest HIL Engine API to mock-up skeleton (native resources management)
- Updated the Architecture import wizard to automatically accept Pack licenses when the Architecture license is accepted

MicroEJ Module Manager

General

- Added JSCH library to execute MicroEJ testsuites on Device through ssh
- Added pre-compilation phase before executing Addon Processor to have compiled classes available
- Updated the default settings file to import modules from MicroEJ Developer repository (located at `${user.dir}\.microej\microej-ivysettings-5.4.xml`)

Build Types

- Updated all relevant build types to load the Platform using the `platform` configuration instead of the `test` configuration:
  - Sandboxed Application (`application`)
  - Foundation Library Implementation (`javaimpl`)
  - Addon Library (`javalib`)
  - MicroEJ Testsuite (`testsuite`)
- Updated Module Repository to allow to partially include an Architecture module (`eval` and/or `prod`)
- Fixed potential Addon Processor error `NoClassDefFoundError: ej/tool/addon/util/Message` depending on the resolution order
- Removed javadoc generation for `microej-extension`
- New build types added:
  - build-application#8.0.0
  - build-artifact-repository#2.3.0
  - build-firmware-singleapp#1.4.0
  - build-microej-extension#1.4.0
  - build-microej-javaimpl#4.0.0
  - build-microej-javalib#5.0.0
  - build-microej-testsuite#3.0.0

Build Plugins

- Updated Addon Processor to fail the build when an error is detected. Error messages are dumped to the build logs.
- Updated Platform Loader to handle Platform module (`.zip` file)
- Updated Platform Loader to handle Virtual Device module (`.vde` file) declared as a dependency. It worked before only by using the `dropins` folder.
- Updated Platform Loader to list the Platforms locations when too many Platform modules are detected

Skeletons

- Fixed wrong `README.md` generation for `artifact-repository` skeleton
- Removed useless files in `microej-javaapi`, `microej-javaimpl` and `microej-extension` skeletons (intern changelog and `.dbk` file)

[5.3.1] - 2020-12-11

Note: This release is both compatible with Eclipse version `2020-06` and Eclipse Oxygen, so it can still be installed on a previous SDK Distribution.

General

- Fixed missing default settings file for connecting MicroEJ Central Repository when starting a fresh install (introduced in `5.3.0`)

MicroEJ Module Manager

Build Plugins

- Fixed potential build error when computing Sonar classpath from dependencies (`ivy:cachepath` task was sometimes using a wrong cache location)

Skeletons

- Fixed skeleton dependency to EDC API 1.3.3 to avoid an Eclipse JDT builder error when Null Analysis is enabled (see *known issues of SDK Distribution 20.10*)

[5.3.0] - 2020-10-30

Note: This release is both compatible with Eclipse version `2020-06` and Eclipse Oxygen, so it can still be installed on a previous SDK Distribution.

Known Issues

- Library module build may lead to unexpected `Unresolved Dependencies` error in some cases (in `sonar:init` target / `ivy:cachepath` task). Workaround is to trigger the library build again.

General

- Fixed various plugins for Eclipse version 2020-06 compatibility (icons, project explorer menu entries)
- Fixed closed module.ivy files after an SDK restart that were opened before
- Removed license check before launching an Application on Device
- Disabled Activate on new event option of the Error Log view to prevent popup of this view when an internal error is thrown
- Removed license check before Platform build
- Updated filter of the Launch Group configuration (exclude the deprecated Eclipse CDT one)
- Fixed inclusion of mock project dependencies in launcher mock classpath
- Enhance error message in Platform editor (.platform files) when the required Architecture has not been imported (displays Architecture information)

MicroEJ Module Manager

General

- Fixed workspace default settings file when clicking on the Default button
- First wrong resolved dependency when ChainResolver returnFirst option is enabled and the module to resolve is already in the cache
- Fixed potential build module crash (Not comparable issue) when resolving module dependencies across multiple configurations

Build Types

- Exclude packs from artifact checker when building a module repository
- Merged Foundation & Add-On Libraries javadoc when building a module repository
- Added Module dependency line for each type in module repository javadoc
- Added an option to skip deprecated types, fields, methods in module repository javadoc
- Allow to include or exclude Java packages in module repository javadoc
- Added an option skip.publish to skip artifacts publication in build-custom build type
- Allow to define Application options from build option using the platform-launcher.inject. prefix
- Added generation and publication of code coverage report after a testsuite execution. The report generation is enabled under the following conditions:
  - at least one test is executed,
  - tests are executed on Simulator,
  - build option s3.cc.activated is set to true (default),
  - the Platform is based on an Architecture version 7.12.0 or higher

  - if testing a Foundation Library (using microej-testsuite), build option microej.testsuite.cc.jars.name.regex must be set to match the simple name of the library being covered (e.g. edc-*.jar or microui-*.jar)
- Fixed sonar false negative Null Analysis detection in some cases
- Added a better error message for Studio rebrand build when izpack.microej.product.location option is missing
- Deprecated build-microej-ri and disabled documentation generation (useless docbook toolchains have been removed to reduce the bundle size: -150MB)
- New build types added:
  - build-artifact-repository#2.0.1
  - build-custom#1.2.0
  - build-firmware-singleapp#1.2.10
  - build-microej-ri#2.4.0

Skeletons

- Fixed microej-mock content script initialization folder name

[5.2.0] - 2020-07-28

General

- Added Dist. prefix in default workspace and repository name to avoid confusion between SDK distribution vs SDK version
- Replaced Version by Dist. in Help > About MicroEJ® SDK menu. The SDK version is available in Installation Details view.
- Replaced IS2T S.A. and MicroEJ S.A. by MicroEJ Corp. in Help > About MicroEJ® SDK menu.
- Updated Front Panel plugin to version 6.1.1
- Removed MicroEJ Copyright in Java class template and skeletons files
- Fixed Stopping a MicroEJ launch in the progress view doesn't stop the launch

MicroEJ Module Manager

General

- Added a new configuration page (Window > Preferences > Module Manager). This page is a merge of formerly named Easyant4Eclipse preferences page and Ivy Settings relevant options for MicroEJ.
- Added Export > MicroEJ > Module Manager Build Kit wizard, to extract the files required for automating MicroEJ modules builds out of the IDE.

- Added New > MicroEJ > Module Project wizard (formerly named New Easyant Project), with module fields content assist and alphabetical sort of the skeletons list
- Added Import > MicroEJ > Module Repository wizard to automatically configure workspace with a module repository (directory or zip file)
- Added New MicroEJ Add-On Library Project wizard to simplify Add-On Library skeleton project creation
- Updated the build repository (microej-build-repository.zip) to be self contained with its owns ivysettings.xml
- Updated Virtual Device Player (firmware-singleapp) launcher-windows.bat (use launcher-windows-verbose.bat to get logs)
- Renamed the classpath container to Module Dependencies instead of Ivy
- Fixed Addon Processor src-adpgenerated folder generation when creating or importing a project with the same name than a previously deleted one
- Fixed implementation of settings ChainResolver returnFirst option
- Fixed Ivy module resolution being blocked from time to time

Build Types

- Fixed meta build to publish correct snapshot revisions for built dependencies. (Indirectly fixes ADP resolution issue when an Add-On Library and its associated Addon Processor were built together using a meta build)
- Fixed potential infinite loop when building a Modules Repository with MMM semantic enabled
- Fixed javadoc not being generated in artifactory repository build when skip.javadoc is set to false
- Added the capability to build partial modules repository, by using the user provided ivysettings.xml file to check the repository consistency
- Added the possibility to partially extend the build repository in a module repository. The build repository can be referenced by a file system resolver using the property ${microej-build-repository.repo.dir}
- Added the possibility to include a module repository into an other module repository (using new configuration repository->*)
- Added the possibility to bundle a set of Virtual Devices when building a branded Studio. They are automatically imported to the MicroEJ repository when booting on a new workspace.
- Added the possibility to bundle a Module Repository when building a branded Studio. It is automatically imported and settings file is configured when booting on a new workspace.

Build Plugins

- Added variables @MMM_MODULE_ORGANISATION@, @MMM_MODULE_NAME@ and @MMM_MODULE_VERSION@ for README.md file
- Fixed microej-kf-testsuite repository access issue (introduced in SDK 5.0.0).
- Fixed artifact-checker to accept revisions surrounded by brackets (as specified by https://keepachangelog.com/en/1.0.0/)

Skeletons

- Updated module.ivy indentation characters with tabs instead of spaces
- Updated CHANGELOG.md formatting
- Updated and standardized README.md files
- Updated dependencies in module.ivy to use the latest versions
- Added .gitignore to ignore the target~ and src-adpgenerated folder where the module is built
- Added Sandboxed Application WPK dropins folder (META-INF/wpk)
- Removed conf provided in module.ivy for foundation libraries dependencies
- Remove MicroEJ internal site reference in module.ant file
- Fixed corrupted library workbenchExtension-api.jar in microej-extension skeleton
- Fixed corrupted library HILEngine.jar in microej-mock skeleton
- Fixed javadoc content issue in Main class firmware-singleapp skeleton

Misc

- Updated End User License Agreement
- Added support for generating Application Options in reStructured Text format

[5.1.2] - 2020-03-09

MicroEJ Module Manager

- Fixed potential build error when generating fixed dependencies file (fixdeps task was sometimes using a wrong cache location)
- Fixed topogical sort of classpath dependencies when building using Build Module (same as in IvyDE classpath sorted view)
- Fixed resolution of modules with a version 0.m.p when transitively fetched (an error was thrown with the range [1.m.p-RC,1.m.(p+1)-RC[)
- Fixed missing classpath dependencies to prevent an error when building a standard JAR with JUnit tests

[5.1.1] - 2019-09-26

General

- Fixed files locked in Platform in workspace projects preventing the Platform from being deleted or rebuilt

[5.1.0] - 2019-05-17

General

- Updated MicroEJ icons (16x16 and 32x32)
- Fixed potential long-blocking operation when launching an application on a Virtual Device on Windows 10 (Windows defender performs a slow analysis on a zip file when it is open for the first time since OS startup)
- Fixed missing ADP resolution on a fresh MicroEJ installation
- Fixed ADP source folders order generation in .classpath (alphabetical sort of the ADP id)
- Fixed Run As... > MicroEJ Application automatic launcher creation: when selecting a Platform in workspace , an other platform of the repository was used instead
- Fixed Memory Map Analyzer load of mapping scripts from Virtual Devices
- Fixed MMM and ADP resolution when importing a zip project in a fresh MicroEJ install
- Fixed ADP crash when a project declares dependencies without a source folder
- Fixed inability to debug an application on a Virtual Device if option execution.mode was specified in firmware build properties (now SDK options cannot be overridden)
- Updated Front Panel plugin to comply with the new Front Panel engine
  - The Front Panel engine has been refactored and moved from UI Pack to Architecture (UI pack 12.0.0 requires Architecture version 7.11.0 or higher)
  - New Front Panel Project wizard now generates a project skeleton for this new Front Panel engine, based on MMM
  - Legacy Front Panel projects for UI Pack v11.1.0 or higher are still valid
- Updated Virtual Device builder to speed-up Virtual Device boot time (pre-installed Applications are now extracted at build time)
- Fixed inability to select a Platform in workspace in a MicroEJ Tool launch configuration
- Fixed broken title in MicroEJ export menu (Platform Export)

MicroEJ Module Manager

Build Plugins

- Added a new option application.project.dir passed to launch scripts with the workspace project directory
- Updated MMM to throw a non ambiguous error message when a module.ivy configured for MMM declares versions with legacy Ivy range notation
- Updated MicroEJ Central Repository cache directory to ${user.dir}\.microej\caches\repository.microej.com-[version] instead of ${user.dir}\.ivy2
- Updated Update Module Dependencies... to be disabled when module.ivy cannot be loaded. The menu entry is now grayed when the project does not declare an IvyDE classpath container
- Fixed wrong resolution order when a module is both resolved in the repository and the workspace (the workspace module must always take precedence to the module resolved in the repository)
- Fixed useless unknown resolver trace when cache is used by multiple Ivy settings configurations with different resolver names.
- Fixed slow Add-On Processor generation. The classpath passed to ADP modules could contain the same entry multiple times, which leads each ADP module to process the same classpath multiple times.
- Fixed misspelled recommendation message when a build failed
- Fixed Update Module Dependencies... tool: wrong ej:match="perfect" added where it was expected to be compatible
- Fixed Update Module Dependencies... tool: parse error when module.ivy contains <artifact type="rip"/> element
- Fixed resolution and publication of a module declared with an Ivy branch
- Fixed character '-' rejected in module organisation (according to MMM specification 2.0-B)
- Fixed ADP resolution error when the Add-On Processor module was only available in the cache
- Fixed potential build crash depending on the build kit classpath order (error was This module requires easyant [0.9,+])
- Fixed product-java broken skeleton

Build Types

- Updated Platform Loader error message when the property platform-loader.target.platform.dir is set to an invalid directory
- Fixed meta build property substitution in *.modules.list files
- Fixed missing publications for README.md and CHANGELOG.md files
- Update skeletons to fetch latest libraries (Wadapps Framework v1.10.0 and Junit v1.5.0)
- Updated README.md publication to generate MMM usage and the list of Foundation Libraries dependencies
- Added a new build nature for building platform options pages (microej-extension)

- Updated Virtual Device builder to speed-up Virtual Device boot time (pre-installed Applications are now extracted at build time)
- Fixed Virtual Device Player builder (dependencies were not exported into the zip file) and updated `firmware-singleapp` skeleton with missing configurations

Skeletons

- Updated `CHANGELOG.md` based on `Keep a Changelog` specification (https://keepachangelog.com/en/1.0.0/)
- Updated offline module repository skeleton to fetch in a dedicated cache directory under `${user.dir}/.microej/caches`

[5.0.1] - 2019-02-14

General

- Removed Wadapps Code generation (see migration notes below)
- Added support for MicroEJ Module Manager semantic (see migration notes below)
- Added a dedicated view for Virtual Devices in MicroEJ Preferences
- Removed Platform related views and menus in the SDK (Import/Export and Preferences)
- Added Studio rebranding capability (product name, icons, splash screen and installer for Windows)
- Added a new meta build version, with simplified syntax for multi-projects build (see migration notes below)
- Added a skeleton for building offline module repositories
- Added support for importing extended characters in Fonts Designer
- Allow to import Virtual Devices with `.vde` extension (`*.jpf` import still available for backward compatibility)
- Removed legacy selection for Types, Resources and Immutables in MicroEJ Launch Configuration (replaced by `*.list` files since MicroEJ `4.0`)
- Enabled IvyDE workspace dependencies resolution by default
- Enabled MicroEJ workspace Foundation Libraries resolution by default
- Added possibility for Architectures to check for a minimum required version of SDK (`sdk.min.version` property)
- Updated `New Standalone Application Project` wizard to generate a Mono-Sandbox Executable skeleton
- Updated Virtual Device Builder to manage Sandboxed Applications (compatible with Architectures Products `*_7.10.0` or newer)
- Updated Virtual Device Builder to include kernel options (now options are automatically filled for the application developer on Simulator)

MicroEJ Module Manager

Build Plugins

- Added IvyDE resolution from properties defined in `Windows` > `Preferences` > `Ant` > `Runtime` > `Properties`
- Fixed *Illegal character in path* error that may occur when running an Add-On Processor
- Fixed IvyDE crash when defining an Ant property file with Eclipse variables

Build Types

- Kept only latest build types versions (skeletons updated)
- Updated metabuild to execute tests by default for private module dependencies
- Removed remaining build dependencies to JDK (Java code compiler and Javadoc processors). All MicroEJ code is now compiled using the JDT compiler
- Introduced a new plugin for executing custom testsuite using MicroEJ testsuite engine
- Fixed *MalformedURLException* error in Easyant trace
- Fixed Easyant build crash when an Ivy settings file contains a cache definitions with a wildcard
- Updated Platform Builder to keep track in the Platform of the architecture on which it has been built (`architecture.properties`)
- Updated Virtual Device Builder to generate with `.vde` extension
- Updated Multi-Sandbox Executable Builder to embed (Sim/Emb) specific modules (Add-On libraries and pre-installed Applications)
- Fixed `build-microej-ri` v1.2.1 missing dependencies (embedded in SDK 4.1.5)

Skeletons

- Updated all skeletons: migrated to latest build types, added more comments, copyright cleanup and configuration for MicroEJ Module Manager semantic)
- Added the latest HIL Engine API to mock-up skeleton (Start and Stop listeners hooks)

3.11.18 Build Types per SDK

- SDK 5.9.0
  - build-addon-processor#2.2.0
  - build-application#9.3.0
  - build-artifact-repository#3.4.0
  - build-custom#2.2.0
  - build-firmware-customizer#3.2.0
  - build-firmware-multiapp#8.2.0

  - build-firmware-singleapp#2.4.0
  - build-izpack#3.3.0
  - build-microej-extension#2.2.0
  - build-microej-javaapi#5.2.0
  - build-microej-javaimpl#5.3.0
  - build-microej-javalib#6.3.0
  - build-microej-mock#2.2.1
  - build-microej-ri#3.2.0
  - build-microej-testsuite#4.3.0
  - build-product-java#2.2.0
  - build-runtime-api#4.1.0
  - build-std-javalib#3.3.1
  - microej-meta-build#3.0.0
- SDK 5.8.2
  - build-addon-processor#2.2.0
  - build-application#9.2.0
  - build-artifact-repository#3.4.0
  - build-custom#2.2.0
  - build-firmware-customizer#3.2.0
  - build-firmware-multiapp#8.2.0
  - build-firmware-singleapp#2.3.0
  - build-izpack#3.3.0
  - build-microej-extension#2.2.0
  - build-microej-javaapi#5.2.0
  - build-microej-javaimpl#5.2.0
  - build-microej-javalib#6.2.0
  - build-microej-mock#2.2.0
  - build-microej-ri#3.2.0
  - build-microej-testsuite#4.2.0
  - build-product-java#2.2.0
  - build-runtime-api#4.1.0
  - build-std-javalib#3.3.0
  - microej-meta-build#3.0.0
- SDK 5.8.0 and SDK 5.8.1
  - build-addon-processor#2.2.0
  - build-application#9.2.0

  - build-artifact-repository#3.4.0
  - build-custom#2.2.0
  - build-firmware-customizer#3.2.0
  - build-firmware-multiapp#8.1.0
  - build-firmware-singleapp#2.2.0
  - build-izpack#3.3.0
  - build-microej-extension#2.2.0
  - build-microej-javaapi#5.2.0
  - build-microej-javaimpl#5.2.0
  - build-microej-javalib#6.2.0
  - build-microej-mock#2.2.0
  - build-microej-ri#3.2.0
  - build-microej-testsuite#4.2.0
  - build-product-java#2.2.0
  - build-runtime-api#4.1.0
  - build-std-javalib#3.2.0
  - microej-meta-build#3.0.0
- SDK 5.7.0
  - build-addon-processor#2.2.0
  - build-application#9.2.0
  - build-artifact-repository#3.3.0
  - build-custom#2.2.0
  - build-firmware-customizer#3.2.0
  - build-firmware-multiapp#8.1.0
  - build-firmware-singleapp#2.2.0
  - build-izpack#3.2.0
  - build-microej-extension#2.2.0
  - build-microej-javaapi#5.2.0
  - build-microej-javaimpl#5.2.0
  - build-microej-javalib#6.2.0
  - build-microej-mock#2.2.0
  - build-microej-ri#3.2.0
  - build-microej-testsuite#4.2.0
  - build-product-java#2.2.0
  - build-runtime-api#4.1.0
  - build-std-javalib#3.2.0

  - microej-meta-build#3.0.0
- SDK 5.6.2, 5.6.1 and SDK 5.6.0
  - build-addon-processor#2.1.0
  - build-application#9.1.0
  - build-artifact-repository#3.2.0
  - build-custom#2.1.0
  - build-firmware-customizer#3.1.0
  - build-firmware-multiapp#8.0.0
  - build-firmware-singleapp#2.1.0
  - build-izpack#3.1.0
  - build-microej-extension#2.1.0
  - build-microej-javaapi#5.1.0
  - build-microej-javaimpl#5.1.0
  - build-microej-javalib#6.1.0
  - build-microej-mock#2.1.0
  - build-microej-ri#3.1.0
  - build-microej-testsuite#4.1.0
  - build-product-java#2.1.0
  - build-runtime-api#4.0.0
  - build-std-javalib#3.1.0
  - microej-meta-build#3.0.0
- SDK 5.5.3, SDK 5.5.2, SDK 5.5.1 and SDK 5.5.0
  - build-addon-processor#2.0.0
  - build-application#9.0.0
  - build-artifact-repository#3.0.0
  - build-custom#2.0.0
  - build-firmware-customizer#3.0.0
  - build-firmware-multiapp#7.0.0
  - build-firmware-singleapp#2.0.0
  - build-izpack#3.0.0
  - build-microej-extension#2.0.0
  - build-microej-javaapi#5.0.0
  - build-microej-javaimpl#5.0.0
  - build-microej-javalib#6.0.0
  - build-microej-mock#2.0.0
  - build-microej-ri#3.0.0

  - build-microej-testsuite#4.0.0
  - build-product-java#2.0.0
  - build-runtime-api#3.0.0
  - build-std-javalib#3.0.0
  - microej-meta-build#3.0.0
- SDK 5.4.1
  - build-addon-processor#1.0.3
  - build-application#8.0.0
  - build-artifact-repository#2.3.0
  - build-custom#1.2.0
  - build-firmware-customizer#2.0.1
  - build-firmware-multiapp#5.1.2
  - build-firmware-singleapp#1.4.0
  - build-izpack#2.0.1
  - build-microej-extension#1.4.0
  - build-microej-javaapi#4.0.4
  - build-microej-javaimpl#4.0.0
  - build-microej-javalib#5.0.0
  - build-microej-mock#1.0.3
  - build-microej-ri#2.4.0
  - build-microej-testsuite#3.0.0
  - build-product-java#1.2.4
  - build-runtime-api#2.0.2
  - build-std-javalib#2.0.1
  - microej-meta-build#2.0.1
- SDK 5.4.0
  - build-addon-processor#1.0.3
  - build-application#8.0.0
  - build-artifact-repository#2.3.0
  - build-custom#1.2.0
  - build-firmware-customizer#2.0.1
  - build-firmware-multiapp#5.1.2
  - build-firmware-singleapp#1.4.0
  - build-izpack#2.0.1
  - build-microej-extension#1.4.0
  - build-microej-javaapi#4.0.4

  - build-microej-javaimpl#4.0.0
  - build-microej-javalib#5.0.0
  - build-microej-mock#1.0.3
  - build-microej-ri#2.4.0
  - build-microej-testsuite#3.0.0
  - build-product-java#1.2.4
  - build-runtime-api#2.0.2
  - build-std-javalib#2.0.1
  - microej-meta-build#2.0.0
- SDK 5.3.1 and SDK 5.3.0
  - build-addon-processor#1.0.3
  - build-application#7.0.2
  - build-artifact-repository#2.0.1
  - build-custom#1.2.0
  - build-firmware-customizer#2.0.1
  - build-firmware-multiapp#5.1.2
  - build-firmware-singleapp#1.2.10
  - build-izpack#2.0.1
  - build-microej-extension#1.3.2
  - build-microej-javaapi#4.0.4
  - build-microej-javaimpl#3.2.2
  - build-microej-javalib#4.1.1
  - build-microej-mock#1.0.3
  - build-microej-ri#2.4.0
  - build-microej-testsuite#2.2.2
  - build-product-java#1.2.4
  - build-runtime-api#2.0.2
  - build-std-javalib#2.0.1
  - microej-meta-build#2.0.0
- SDK 5.2.0
  - build-addon-processor#1.0.3
  - build-application#7.0.2
  - build-artifact-repository#1.6.2
  - build-custom#1.1.3
  - build-firmware-customizer#2.0.1
  - build-firmware-multiapp#5.1.2

  - build-firmware-singleapp#1.2.9
  - build-izpack#2.0.1
  - build-microej-extension#1.3.2
  - build-microej-javaapi#4.0.4
  - build-microej-javaimpl#3.2.2
  - build-microej-javalib#4.1.1
  - build-microej-mock#1.0.3
  - build-microej-ri#2.3.1
  - build-microej-testsuite#2.2.2
  - build-product-java#1.2.4
  - build-runtime-api#2.0.2
  - build-std-javalib#2.0.1
  - microej-meta-build#2.0.0
- SDK 5.1.2, SDK 5.1.1 and SDK 5.1.0
  - build-addon-processor#1.0.3
  - build-application#7.0.2
  - build-artifact-repository#1.6.0
  - build-custom#1.1.3
  - build-firmware-customizer#2.0.1
  - build-firmware-multiapp#5.1.2
  - build-firmware-singleapp#1.2.9
  - build-izpack#2.0.1
  - build-microej-extension#1.3.2
  - build-microej-javaapi#4.0.4
  - build-microej-javaimpl#3.2.2
  - build-microej-javalib#4.1.1
  - build-microej-mock#1.0.3
  - build-microej-ri#2.3.1
  - build-microej-testsuite#2.2.2
  - build-product-java#1.2.4
  - build-runtime-api#2.0.2
  - build-std-javalib#2.0.1
  - microej-meta-build#2.0.0

3.11.19 Migration Notes

From 5.2.x to 5.3.x or more

This section applies if MicroEJ SDK 5.3.x is started on a workspace that was previously created using MicroEJ SDK 5.2.x .

Workspace migration warning

Starting with the MicroEJ SDK Distribution 20.10, when opening a workspace which has been created with an older MicroEJ Distribution, a message is displayed with the following warning:

```
The workspace was written with an older version.
↪ Continue and update workspace which may make it incompatible with older versions?
```

This is a generic warning from Eclipse which can be safely ignored as long as you don't intend to open it back with an older MicroEJ SDK Distribution then.

From 5.1.x to 5.2.x

This section applies if MicroEJ SDK 5.2.x is started on a workspace that was previously created using MicroEJ SDK 5.1.x .

Enable New Wizards Shortcuts in MicroEJ Perspective

Eclipse perspective settings are stored in the workspace metadata, so the new wizards shortcuts (Add-On Library Project and Module Project) are not visible in the File > New menu.

The MicroEJ perspective must be reset to its default settings as following:

- Click on Windows > Perspective > Open Perspective > Other… menu
- Select MicroEJ perspective
- Click on Windows > Perspective > Reset Perspective… menu
- Click on Yes button to accept to reset the MicroEJ perspective to its defaults.

The new wizards shortcuts are now visible into File > New menu.

Re-enable the Ivy Preferences Pages (Advanced Use)

The original Window > Preferences > Ivy pages can be re-enabled as following:

- Close all running instances of the SDK
- Edit MicroEJ-SDK.ini and add the property -Dorg.apache.ivy.showAdvancedPrefs=true
- Start the SDK again
- Go to Window > Preferences > Module Manager page

A new link Ivy settings should appear on the bottom of the page. It opens a popup window with the original Ivy preferences pages.

From 4.1.x to 5.x

This section applies if MicroEJ SDK 5.x is started on a workspace that was previously created using MicroEJ SDK 4.1.x .

Wadapps Application Update

The Wadapps code generator has been moved from IDE to an Addon Processor coming with ej.library.wadapps.framework module (v1.9.0 or higher is required).

A Wadapps Application Project can be updated as follows:

- Right-click on the project, then Configure > Remove Sandboxed Application Nature
- Right-click on the project, then Configure > Add Sandboxed Application Nature
- Update module.ivy dependency to fetch ej.library.wadapps.framework version 1.9.0 (or perform MicroEJ Module Manager update as defined below)
- Delete remaining folder src/.generated~ if any
- Check that project compiles and folder src-adpgenerated/wadapps is generated

MicroEJ Module Manager Update

It is highly recommended to migrate module.ivy to the MicroEJ Module Manager semantic, since the default Ivy resolution will be no more maintained in future versions.

The module.ivy can be updated as follows:

- Right-click on module.ivy , then Update Module Dependencies…

This has for effect to both migrate the module.ivy to the MicroEJ Module Manager semantic and also to update dependencies version to the latest available in the target repository.

Meta build Project Update

A project using microej-meta-build version 1.x can be updated to version 2.x as follows:

- Edit module.ivy
  - Replace the microej-meta-build version by 2.0.+
  - Update all properties declaration to append the metabuild.inject. prefix (e.g. <ea:property name="skip.test" value="true" /> must be updated to <ea:property name="metabuild.inject.skip.test" value="true" />)
  - Optionally remove or comment the root folder declaration as it is the default. (<ea:property name="metabuild.root" value=".." />)

- Delete module.properties. It only contains the property easyant.fork.build=true. This property is now automatically set by easyant-build-component since version 1.12.0. Otherwise it must be explicitly injected by the build system as an Ant property: easyant.inject.easyant.fork.build=true
- Extract from override.module.ant the projects declarations lines:
  - Extract the project declarations of local.submodule.dirs.id into a new file named private.modules.list (one project per line)
  - Extract the project declarations of submodule.dirs.id into a new file names public.modules.list (one project per line)
- Delete override.module.ant

  The new file system structure shall look like:

```
metabuild-project
    module.ivy
    private.modules.list
    public.modules.list
```

3.12 Get Support

If any questions, the best starting point is to consult the MicroEJ Forum. Feel free to create a new topic if there is no relevant content for your issue. MicroEJ Corp. engineers are listening to the forum activity, so you can expect to get a quick reply.

Otherwise, you can contact our support team.

In both cases, please provide as much information as possible on your installed environment (the table below is an example):

Delivery	**Name**
MICROEJ SDK	**SDK 6**: Gradle plugin version **1.4.0** (see *Changelog*) **SDK 5**: SDK Distribution **24.01** / SDK Version **5.8.2** (see *SDK Version*)
MicroEJ Architecture	ARM Cortex-M4 / IAR Production version **8.4.0** (see *MicroEJ Architecture*)
Module Repository	https://repository.microej.com/packages/repository/3.51.0/central-repository-3.51.0.zip (see *Central Repository*)
C compiler	IAR Embedded Workbench for Arm **v9.60.0**
Host Operating System	Windows **11** (see *System Requirements*)

3.13 About MicroEJ

MicroEJ's mission is to democratize virtualization and Object Oriented Programming (OOP) to the embedded world. These two technologies, widely used in computers and smartphones, radically simplifies how device software is built, from prototyping to hardware choice, by integrating simulation, systemic software reuse, modularity, agility, continuous integration, automated testing and software component update in the development process.

The virtualized environment provided by MICROEJ VEE on-device platform allows for software development on virtual devices, exact "virtual twins" of real electronic configurations.

Since several configurations can be tested and evaluated within days, it is therefore much easier to build several prototypes while capitalizing on the code that has already been built as “ready-to-use” binary software assets.

MicroEJ also offers an integrated development environment, called MICROEJ SDK, which provides one of the widest ranges of standard and specialized tools and libraries, making it possible to easily develop applications implementing IoT connectivity, graphical interfaces, security, and real-time processing of data (Edge Computing).

Browse this documentation to discover MicroEJ technology, learn about application and platform development, and begin your coding journey thanks to a comprehensive range of dedicated training courses.

For more information about MicroEJ, go to: https://www.microej.com/.