MicroEJ Documentation

MicroEJ Corp.

Revision dd017efa

Dec 18, 2020

Copyright 2008-2020, 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 MicroEJ Glossary **2**

2 Overview **4**
2.1 MicroEJ Editions . . . 4
2.1.1 Introduction . . . 4
2.1.2 Determine the MicroEJ Studio/SDK Version . . . 5
2.2 Licenses . . . 7
2.2.1 Overview . . . 7
2.2.2 License Manager . . . 7
2.2.3 Evaluation Licenses . . . 7
2.2.4 Production Licenses . . . 9
2.3 MicroEJ Runtime . . . 13
2.3.1 Language . . . 13
2.3.2 Scheduler . . . 13
2.3.3 Garbage Collector . . . 14
2.3.4 Foundation Libraries . . . 14
2.4 MicroEJ Libraries . . . 14
2.5 MicroEJ Central Repository . . . 15
2.6 Embedded Specification Requests . . . 15
2.7 MicroEJ Firmware . . . 15
2.7.1 Bootable Binary with Core Services . . . 15
2.7.2 Specification . . . 16
2.8 Introducing MicroEJ SDK . . . 16
2.9 Introducing MicroEJ Studio and Virtual Devices . . . 17
2.10 Perform Online Getting Started . . . 18
2.11 GitHub Repositories . . . 19
2.12 System Requirements . . . 24

3 Application Developer Guide **25**
3.1 Introduction . . . 25
3.2 Local Workspaces and Repositories . . . 25
3.3 Standalone Application . . . 26
3.3.1 Download and Install a MicroEJ Platform . . . 26
3.3.2 Build and Run an Application . . . 28
3.3.3 Build Output Files . . . 33
3.3.4 MicroEJ Launch . . . 34
3.3.5 Application Options . . . 38
3.3.6 SOAR . . . 65
3.4 Sandboxed Application . . . 66
3.4.1 Sandboxed Application Structure . . . 66

3.4.2 Application Publication . . . 67
3.4.3 Shared Interfaces . . . 67
3.5 Virtual Device . . . 71
3.5.1 Using a Virtual Device for Simulation . . . 71
3.5.2 Runtime Environment . . . 71
3.6 MicroEJ Module Manager . . . 72
3.6.1 Introduction . . . 72
3.6.2 Specification . . . 73
3.6.3 Module Project Skeleton . . . 73
3.6.4 Module Description File . . . 74
3.6.5 MicroEJ Module Manager Configuration . . . 74
3.6.6 Build Kit . . . 78
3.6.7 Former MicroEJ SDK Versions . . . 78
3.7 Module Natures . . . 80
3.7.1 Module Repository . . . 80
3.8 MicroEJ Classpath . . . 84
3.8.1 Application Classpath . . . 84
3.8.2 Classpath Load Model . . . 85
3.8.3 Classpath Elements . . . 86
3.9 Application Resources . . . 89
3.9.1 Images . . . 89
3.9.2 Fonts . . . 90
3.9.3 Native Language Support . . . 91
3.10 Development Tools . . . 92
3.10.1 Testsuite with JUnit . . . 93
3.10.2 Stack Trace Reader . . . 97
3.10.3 Code Coverage Analyzer . . . 104
3.10.4 Heap Dumper & Heap Analyzer . . . 107
3.10.5 ELF to Map File Generator . . . 118
3.10.6 Serial to Socket Transmitter . . . 120
3.10.7 Memory Map Analyzer . . . 121
3.10.8 Event Tracing . . . 124
3.11 Advanced Tools . . . 126
3.11.1 MicroEJ Linker . . . 126
3.11.2 Testsuite Engine . . . 139
3.11.3 MicroEJ Testsuite Engine . . . 139
3.12 Graphical User Interface . . . 145
3.12.1 MicroUI . . . 146
3.12.2 MWT (Micro Widget Toolkit) . . . 175
3.12.3 Widgets and Examples . . . 191

4 Platform Developer Guide **194**
4.1 Introduction . . . 194
4.1.1 Scope . . . 194
4.1.2 Intended Audience . . . 194
4.1.3 MicroEJ Architecture Modules Overview . . . 194
4.2 MicroEJ Platform . . . 196
4.2.1 Process Overview . . . 196
4.2.2 Concepts . . . 197
4.2.3 MicroEJ Platform Creation . . . 202
4.2.4 Platform Qualification . . . 212
4.3 MicroEJ Core Engine . . . 213
4.3.1 Functional Description . . . 213
4.3.2 Architecture . . . 214

4.3.3 Capabilities . . . 215
4.3.4 Implementation . . . 215
4.3.5 Generic Output . . . 218
4.3.6 Link . . . 218
4.3.7 Dependencies . . . 219
4.3.8 Installation . . . 219
4.3.9 Use . . . 219
4.4 Multi-Sandbox . . . 220
4.4.1 Principle . . . 220
4.4.2 Functional Description . . . 220
4.4.3 Firmware Linker . . . 221
4.4.4 Memory Considerations . . . 221
4.4.5 Dependencies . . . 221
4.4.6 Installation . . . 221
4.4.7 Use . . . 221
4.5 Tiny application . . . 222
4.5.1 Principle . . . 222
4.5.2 Installation . . . 222
4.5.3 Limitations . . . 222
4.6 Native Interface Mechanisms . . . 222
4.6.1 Simple Native Interface (SNI) . . . 222
4.6.2 Shielded Plug (SP) . . . 226
4.6.3 MicroEJ Java H . . . 229
4.7 External Resources Loader . . . 230
4.7.1 Principle . . . 230
4.7.2 Functional Description . . . 230
4.7.3 Implementations . . . 230
4.7.4 External Resources Folder . . . 231
4.7.5 Dependencies . . . 231
4.7.6 Installation . . . 231
4.7.7 Use . . . 231
4.8 Serial Communications . . . 231
4.8.1 ECOM . . . 232
4.8.2 ECOM Comm . . . 233
4.9 Graphical User Interface . . . 241
4.9.1 Principle . . . 241
4.9.2 MicroUI . . . 242
4.9.3 Static Initialization . . . 245
4.9.4 Low Level API . . . 249
4.9.5 LED . . . 250
4.9.6 Input . . . 252
4.9.7 Display . . . 256
4.9.8 Images . . . 278
4.9.9 Fonts . . . 296
4.9.10 Simulation . . . 303
4.10 Networking . . . 306
4.10.1 Principle . . . 306
4.10.2 Network Core Engine . . . 307
4.10.3 SSL . . . 308
4.11 File System . . . 308
4.11.1 Principle . . . 308
4.11.2 Functional Description . . . 309
4.11.3 Dependencies . . . 309
4.11.4 Installation . . . 309

4.11.5 Use . . . 309
4.12 Hardware Abstraction Layer . . . 309
4.12.1 Principle . . . 309
4.12.2 Functional Description . . . 310
4.12.3 Identifier . . . 310
4.12.4 Configuration . . . 311
4.12.5 Dependencies . . . 311
4.12.6 Installation . . . 311
4.12.7 Use . . . 311
4.13 Device Information . . . 311
4.13.1 Principle . . . 311
4.13.2 Dependencies . . . 312
4.13.3 Installation . . . 312
4.13.4 Use . . . 312
4.14 Event Tracing . . . 312
4.14.1 Introduction . . . 312
4.14.2 SystemView . . . 312
4.15 Simulation . . . 315
4.15.1 Principle . . . 315
4.15.2 Functional Description . . . 315
4.15.3 Dependencies . . . 316
4.15.4 Installation . . . 316
4.15.5 Use . . . 316
4.15.6 Mock . . . 316
4.15.7 Shielded Plug Mock . . . 321
4.15.8 Front Panel Mock . . . 322
4.15.9 Bluetooth LE Mock . . . 330
4.16 Limitations . . . 336
4.17 Appendices . . . 336
4.17.1 Appendix A: Low Level API . . . 336
4.17.2 Appendix B: MicroEJ Foundation Libraries . . . 345
4.17.3 Appendix C: Tools Options and Error Codes . . . 355
4.17.4 Appendix D: Architectures MCU / Compiler . . . 366

5 Kernel Developer Guide 370
5.1 Overview . . . 370
5.1.1 Introduction . . . 370
5.1.2 Terms and Definitions . . . 370
5.1.3 Overall Architecture . . . 371
5.1.4 Firmware Build Flow . . . 375
5.1.5 Virtual Device Build Flow . . . 376
5.2 Kernel & Features Specification . . . 376
5.3 Getting Started . . . 377
5.3.1 Online Getting Started . . . 377
5.3.2 Create an Empty Firmware from Scratch . . . 377
5.3.3 MicroEJ Demo VEE Flavors . . . 380
5.4 Build Firmware . . . 381
5.4.1 Workspace Build . . . 383
5.4.2 Headless Build . . . 385
5.4.3 Runtime Environment . . . 386
5.4.4 Resident Applications . . . 386
5.4.5 Advanced . . . 387
5.5 Writing Kernel APIs . . . 389
5.5.1 Default Kernel APIs Derivation . . . 389

5.5.2 Build a Kernel API Module . . . 390
5.5.3 Kernel API Generator . . . 390
5.6 Communication between Features . . . 392
5.6.1 Kernel Type Converters . . . 392
5.7 API Documentation . . . 392
5.8 Multi-Sandbox Enabled Libraries . . . 392
5.8.1 MicroUI . . . 392
5.8.2 ECOM . . . 393
5.8.3 ECOM-COMM . . . 393
5.8.4 FS . . . 393
5.8.5 NET . . . 393
5.8.6 SSL . . . 393
5.9 Setup a KF Testsuite . . . 393
5.9.1 Enable the Testsuite . . . 393
5.9.2 Add a KF Test . . . 394
5.9.3 KF Testsuite Options . . . 396

6 Tutorials **397**
6.1 Understand How to Build a MicroEJ Firmware and its Dependencies . . . 397
6.1.1 The Components . . . 397
6.1.2 How to Build . . . 400
6.1.3 Get Support . . . 402
6.2 Create a MicroEJ Platform for a Custom Device . . . 402
6.2.1 Introduction . . . 402
6.2.2 A MicroEJ Platform Project is already available for the same MCU/RTOS/C Compiler . . . 403
6.2.3 A MicroEJ Platform Project is not available for the same MCU/RTOS/C Compiler . . . 404
6.2.4 Platform Validation . . . 405
6.2.5 Further Assistance Needed . . . 405
6.3 Create a MicroEJ Firmware From Scratch . . . 405
6.3.1 Intended Audience . . . 405
6.3.2 Introduction . . . 405
6.3.3 Prerequisites . . . 406
6.3.4 Overview . . . 406
6.3.5 Setup the Development Environment . . . 407
6.3.6 Get Running BSP . . . 407
6.3.7 FreeRTOS Hello World . . . 409
6.3.8 Create a MicroEJ Platform . . . 411
6.3.9 Create MicroEJ Application HelloWorld . . . 415
6.3.10 Configure BSP Connection in MicroEJ Application . . . 418
6.3.11 MicroEJ and FreeRTOS Integration . . . 420
6.4 Setup an Automated Build using Jenkins and Artifactory . . . 431
6.4.1 Intended Audience . . . 431
6.4.2 Introduction . . . 431
6.4.3 Prerequisites . . . 432
6.4.4 Overview . . . 432
6.4.5 Install the Build Tools . . . 433
6.4.6 Get a Module Repository . . . 433
6.4.7 Setup Artifactory . . . 434
6.4.8 Setup Jenkins . . . 437
6.4.9 Build a new Module using Jenkins . . . 439
6.4.10 Appendix . . . 442
6.5 Improve the Quality of Java Code . . . 443
6.5.1 Intended Audience . . . 443
6.5.2 Readable Code . . . 443

6.5.3 Best Practices . . . 446
6.5.4 Related Tools . . . 449
6.6 Optimize the Memory Footprint of an Application . . . 449
6.6.1 Intended Audience . . . 449
6.6.2 Introduction . . . 450
6.6.3 How to Analyze the Footprint of an Application . . . 450
6.6.4 How to Reduce the Image Size of an Application . . . 451
6.6.5 How to Reduce the Runtime Size of an Application . . . 456
6.7 Explore Data Serialization Formats . . . 458
6.7.1 Intended Audience . . . 459
6.7.2 XML . . . 459
6.7.3 JSON . . . 460
6.8 Instrument Java Code for Logging . . . 463
6.8.1 Intended Audience . . . 463
6.8.2 Introduction . . . 463
6.8.3 Overview . . . 463
6.8.4 Log with the Trace Library . . . 464
6.8.5 Log with the Message Library . . . 465
6.8.6 Log with the Logging Library . . . 466
6.8.7 Remove Logging Related Code . . . 467

7 About MicroEJ **470**

Index **471**

Welcome to MicroEJ developer documentation. Browse the following chapters to familiarize yourself with MicroEJ Technology and understand the principles of app and platform development with MicroEJ.

- The Glossary chapter describes MicroEJ terminology.
- The Overview chapter introduces MicroEJ products and technology.
- The Application Developer Guide presents Java applications development and debugging tools.
- The Platform Developer Guide teaches you how to integrate a C Board Support 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 Tutorials chapter covers a variety of topics related to developing with the MicroEJ ecosystem.

CHAPTER

ONE

MICROEJ GLOSSARY

This glossary defines the technical terms upon which the MicroEJ Virtual Execution Environment is built.

Add-On Library A MicroEJ Add-On Library is a pure managed code (Java) library. It runs over one or more MicroEJ Foundation Libraries.

Application A MicroEJ Application is a software program that runs on a Powered by MicroEJ device.

- **Standalone Application** MicroEJ Standalone Application is a MicroEJ Application that is directly linked to the C code to produce a MicroEJ Mono-Sandbox Firmware. It is edited using MicroEJ SDK.
- **Sandboxed Application** A MicroEJ Sandboxed Application is a MicroEJ Application that can run over a MicroEJ Multi-Sandbox Firmware. It can be linked either statically or dynamically.
- **System Application** A MicroEJ System Application is a MicroEJ Sandboxed Application that is statically linked to a MicroEJ Multi-Sandbox Firmware, as it is part of the initial image and cannot be removed.
- **Kernel Application** A MicroEJ Kernel Application is a MicroEJ Standalone Application that implements the ability to be extended to produce a MicroEJ Multi-Sandbox Firmware.

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

Core Engine MicroEJ Core Engine is a scalable runtime for resource-constrained embedded devices running on 32-bit microcontrollers or microprocessors. MicroEJ Core Engine allows devices to run multiple and mixed Java and C software applications.

Firmware A MicroEJ Firmware is the result of the binary link of a MicroEJ Standalone Application with a MicroEJ Platform. The firmware is a binary program that can be programmed into the flash memory of a device.

- **Mono-Sandbox Firmware** A MicroEJ Mono-Sandbox Firmware is a MicroEJ Firmware that implements an unmodifiable set of functions. (previously MicroEJ Single-app Firmware)
- **Multi-Sandbox Firmware** A MicroEJ Multi-Sandbox Firmware is a MicroEJ Firmware that implements the ability to be extended, by exposing a set of APIs and a memory space to link MicroEJ Sandboxed Applications. (previously MicroEJ Multi-app Firmware)

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

Mock A MicroEJ Mock is a mockup of a Board Support Package capability that mimics an hardware functionality for the MicroEJ Simulator.

Module Manager MicroEJ Module Manager downloads, installs and controls the consistency of all the dependencies and versions required to build and publish a MicroEJ asset. It is based on Semantic Versioning specification.

Platform A MicroEJ Platform integrates a MicroEJ Architecture, one or more Foundation Libraries with their respective Abstraction Layers and the board support package (BSP) for the target Device. It also includes associated MicroEJ Mocks for the MicroEJ Simulator.

SDK MicroEJ SDK allows MicroEJ Firmware developers to build a MicroEJ-ready device, by integrating a MicroEJ Architecture with both Java and C software on their device.

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

Studio MicroEJ Studio allows application developers to write a MicroEJ Sandboxed Application, run it on a Virtual Device, deploy it on a MicroEJ-ready device, and publish it to a MicroEJ Forge instance.

Virtual Device A MicroEJ Virtual Device is a software package that includes the simulation part of a MicroEJ Firmware: runtime, libraries and application(s). It can be run on any PC without the need of MicroEJ Studio. In case a MicroEJ Multi-Sandbox Firmware, it is also used for testing a MicroEJ Sandboxed Application in MicroEJ Studio.

CHAPTER

TWO

OVERVIEW

2.1 MicroEJ Editions

2.1.1 Introduction

MicroEJ offers a comprehensive toolset to build the embedded software of a device. The toolset covers two levels in device software development:

- MicroEJ SDK for device firmware development
- MicroEJ Studio for application development

The firmware will generally be produced by the device OEM, it includes all device drivers and a specific set of MicroEJ functionalities useful for application developers targeting this device.

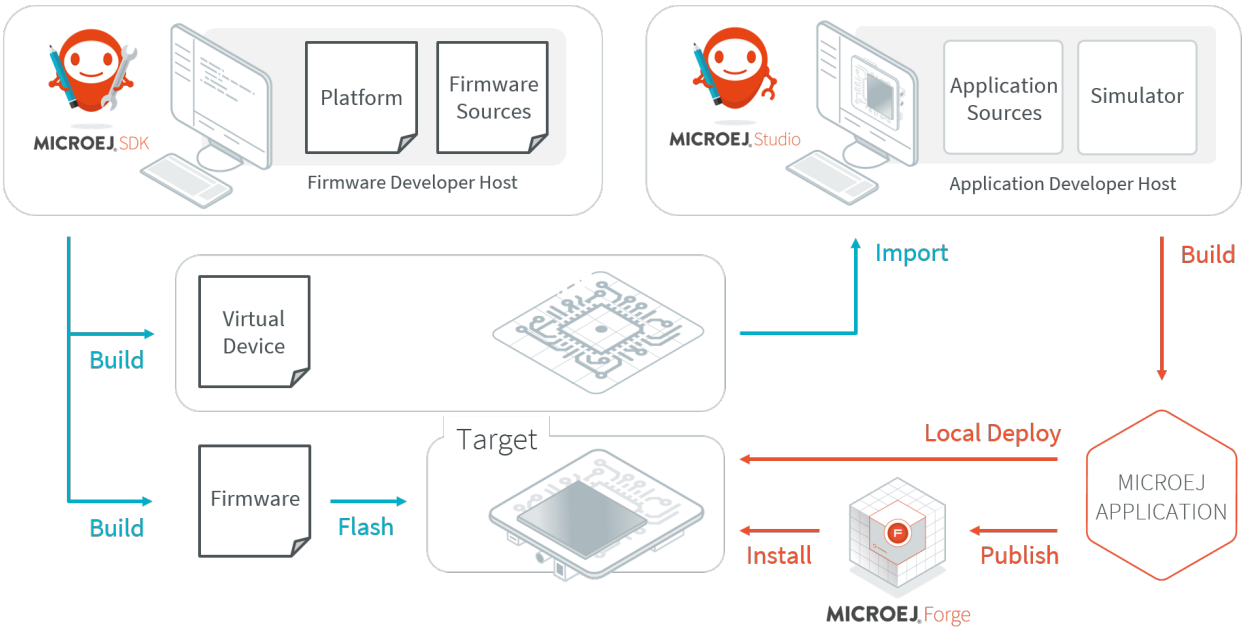


Fig. 1: MicroEJ Development Tools Overview

Using the MicroEJ SDK tool, a firmware developer will produce two versions of the MicroEJ binary, each one able to run applications created with the MicroEJ Studio tool:

- A MicroEJ Firmware binary to be flashed on OEM devices;

- A Virtual Device which will be used as a device simulator by application developers.

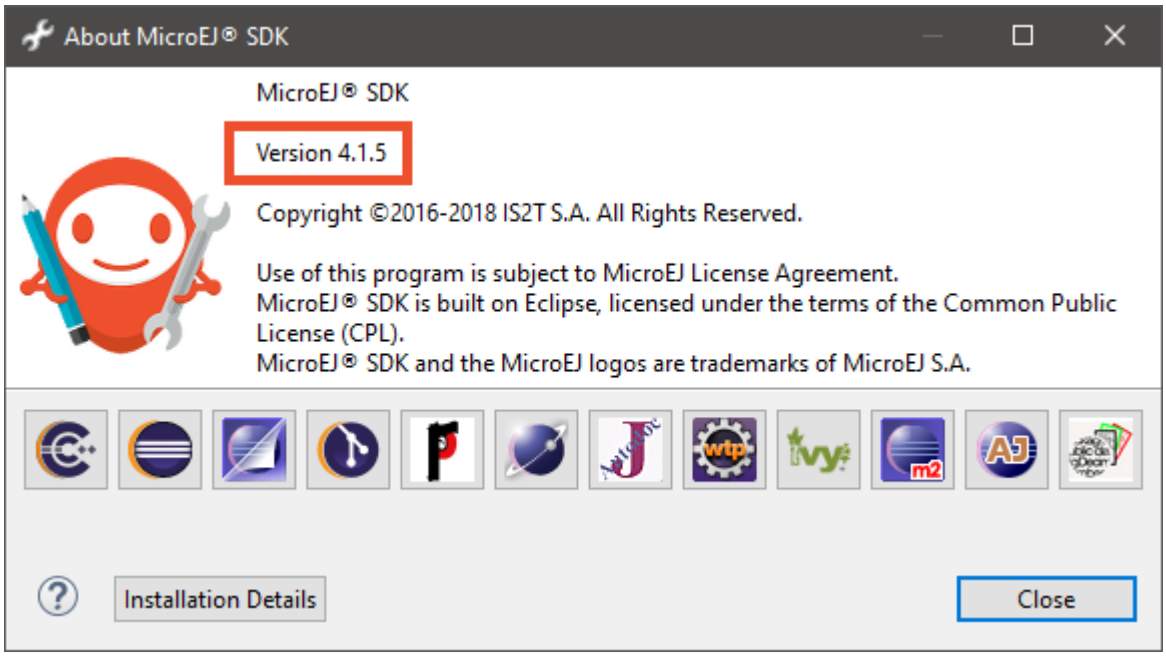
Using the MicroEJ Studio tool, an application developer will be able to:

- Import Virtual Devices matching his target hardware in order to develop and test applications on the Simulator;
- Deploy the application locally on an hardware device equipped with the MicroEJ Firmware;
- Package and publish the application on a MicroEJ Forge Instance, enabling remote end users to install it on their devices. For more information about MicroEJ Forge, please consult https://www.microej.com/product/forge.

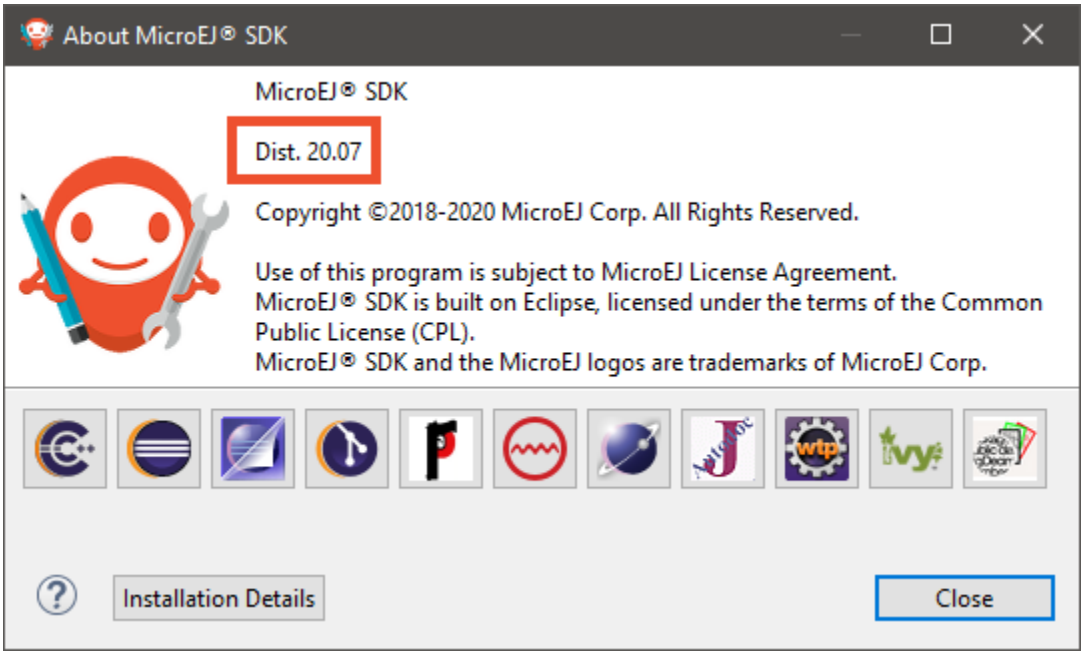
2.1.2 Determine the MicroEJ Studio/SDK Version

In MicroEJ Studio/SDK, go to `Help` > `About MicroEJ SDK` menu.

In case of MicroEJ SDK `4.1.x`, the MicroEJ SDK version is directly displayed, such as `4.1.5`:

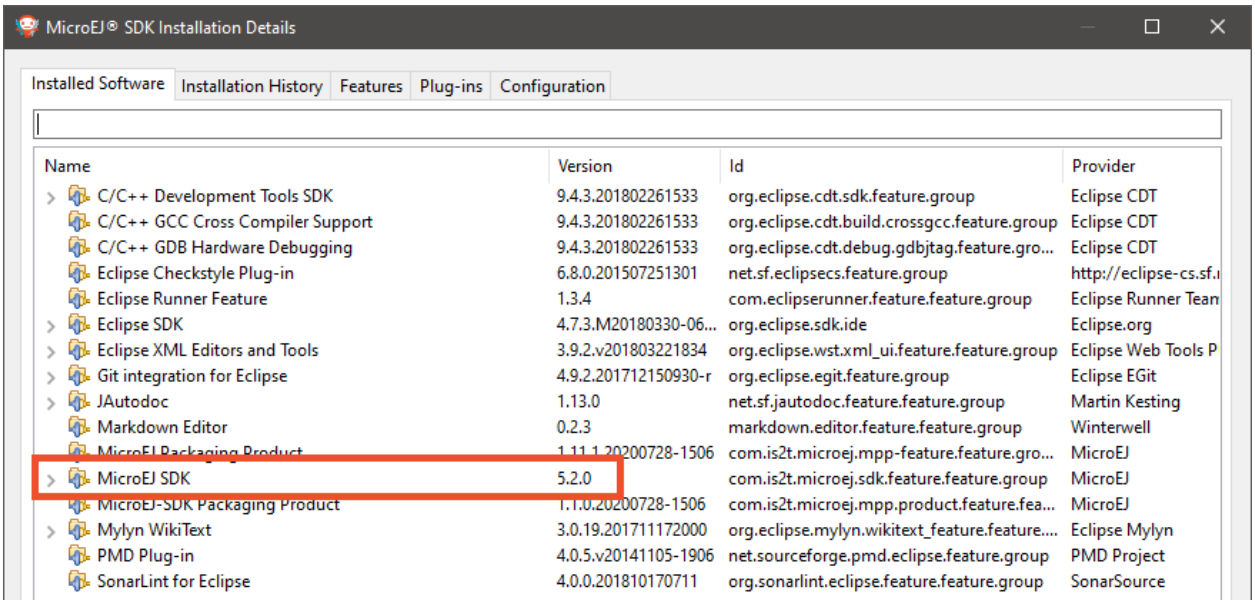


In case of MicroEJ SDK `5.x`, the value displayed is the MicroEJ SDK distribution, such as `19.05` or `20.07`:



To retrieve the MicroEJ 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` (or `MicroEJ Studio`).



2.2 Licenses

2.2.1 Overview

MicroEJ Architectures are distributed in two different versions:

- Evaluation Architectures, associated with a software license key. Can be downloaded at https://repository.microej.com/architectures/.
- Production Architectures, associated with an hardware license key stored on a USB dongle. Can be requested to MicroEJ support team support@microej.com.

Licenses list is available in MicroEJ SDK preferences dialog page in Window > Preferences > MicroEJ :

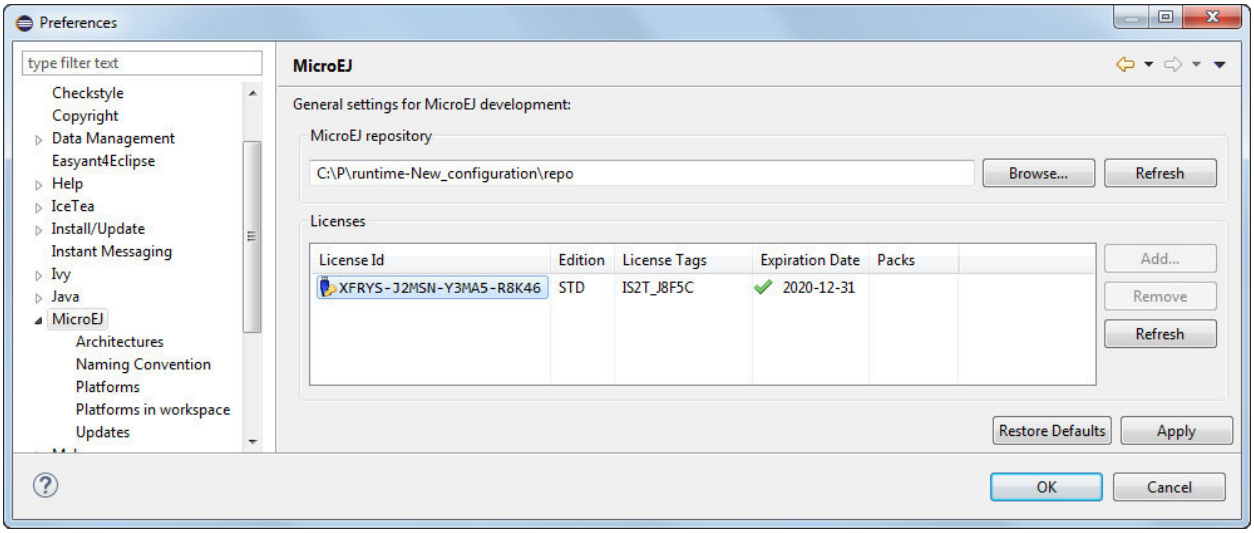


Fig. 2: MicroEJ Licenses View

2.2.2 License Manager

The license manager is provided with MicroEJ Architectures and this then integrated to Platforms, consequently:

- Evaluation licenses will be shown only if at least one Evaluation Architecture or Platform built from an Evaluation Architecture has been imported in MicroEJ SDK.
- Production licenses will be shown only if at least one Production Architecture or Platform built from a Production Architecture has been imported in MicroEJ SDK.

See sections *MicroEJ Architecture Import* and *Download and Install a MicroEJ Platform* for more information.

2.2.3 Evaluation Licenses

This section should be considered when using Evaluation Architectures, which use software license keys.

Get your Machine UID

To activate an Evaluation Architecture, a machine UID needs to be provided to the key server.

This information is available from the preferences page:

- Go to Window > Preferences > MicroEJ ,
- Select either Architectures or Platforms ,
- Click on one of the available Architectures or Platforms,
- Press the Get UID button to get the machine UID.

Note: To access this Get UID option, at least one Evaluation Architecture must have been imported before (see *License Manager*).

Copy the UID. It will be needed when requesting a license.

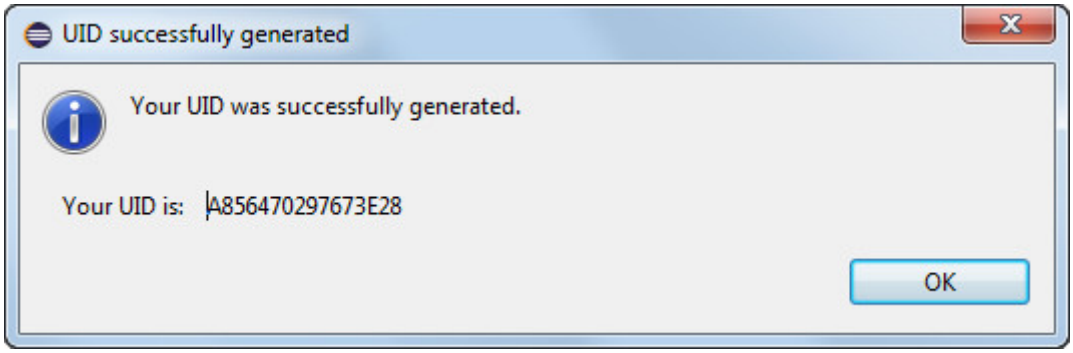


Fig. 3: Machine UID for Evaluation License

Request your Activation Key

- Go to 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 login with your new account.
- Click on Activate a License .
- Set Product P/N: to `9PEVNLDBU6IJ` .
- Set UID: to the 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 support@microej.com.
- Once received by email, save the attached zip file that contains your activation key.

Install the License Key

- Go back to MicroEJ 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 MicroEJ 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 key-store by selecting it and by clicking on Remove button.

Troubleshooting

Consider this section when an error message appears while adding the Evaluation license key. Before contacting MicroEJ support, 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)

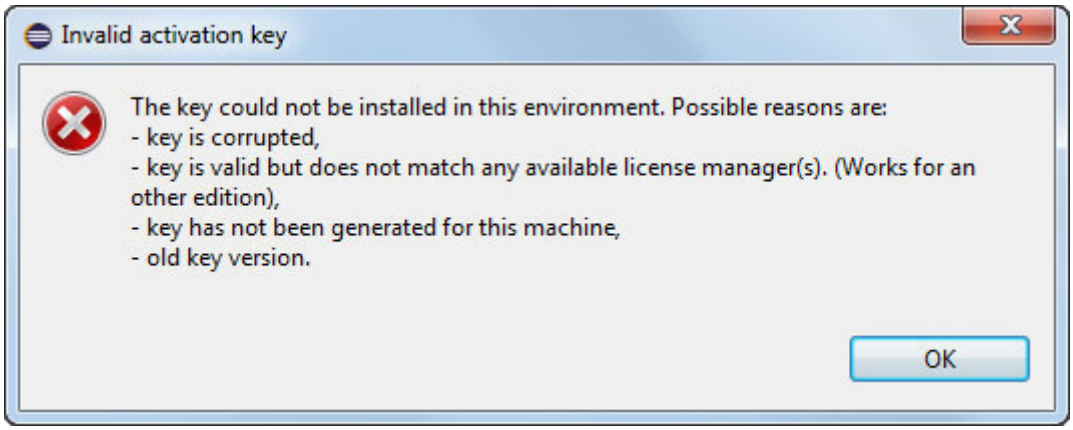


Fig. 4: Invalid License Key Error Message

2.2.4 Production Licenses

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

Fig. 5: 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 *Check Activation on MicroEJ SDK* 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 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 to 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 to flash your USB dongle with the proper activation key.

You shall ensure that the following prerequisites are met :

- The USB dongle is plugged and recognized by your operating system (see *Troubleshooting* section)
- No more than one USB dongle is plugged to the computer while running the update tool
- The update tool is not launched from a Network drive or from 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.
- Click on the Update button (no password needed)

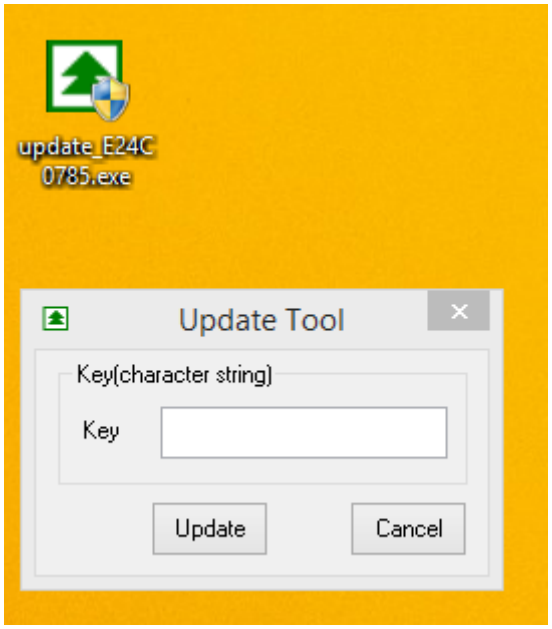


Fig. 6: Dongle Update Tool

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

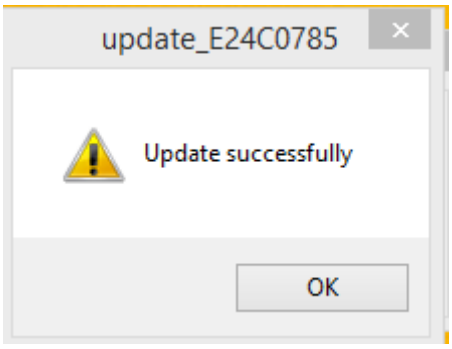


Fig. 7: Successful dongle update

Check Activation on MicroEJ SDK

Note: Production licenses will be shown only if at least one Production Architecture has been imported before (see *License Manager*).

- Go back to MicroEJ SDK,
- Go to Window > Preferences > MicroEJ ,
- Go to Architectures or Platforms sub-menu and check that all Production Architectures or Platforms are now activated (green check).

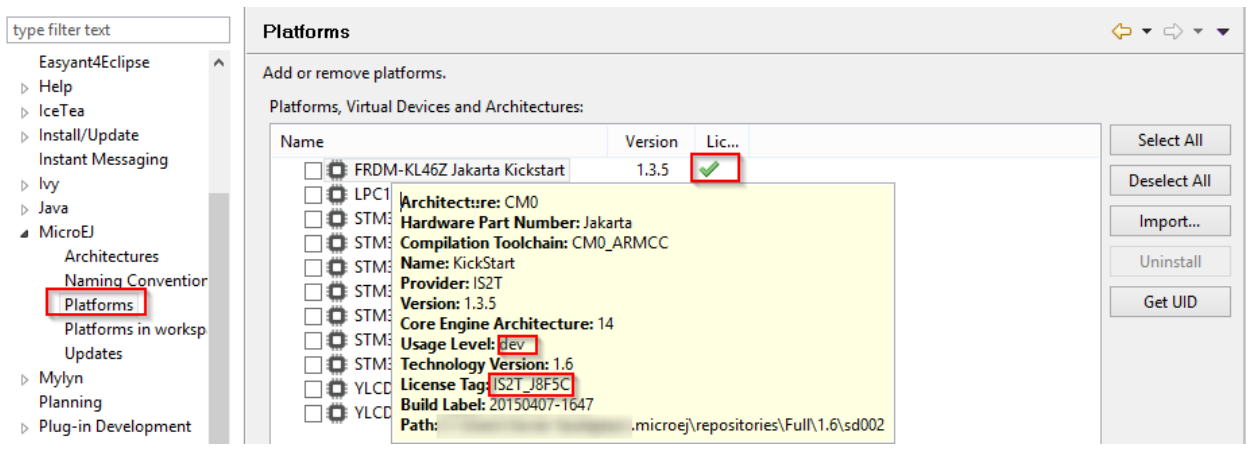


Fig. 8: Platform License Status OK

Troubleshooting

This section contains instructions to check that your USB dongle is correctly recognized by your operating system.

GNU/Linux Troubleshooting

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: `/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.`

Windows Troubleshooting

For Windows users, each dongle shall be recognized with the following hardware ID :

```
HID\VID_096E&PID_0006&REV_0109
```

On Windows 8.1, go to Device Manager > Human Interface Devices and check among the `USB Input Device` entries that the `Details` > `Hardware Ids` property match the ID mentioned before.

VirtualBox Troubleshooting

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

In order to make this setting persistent, go to `Devices` > `USB` > `USB Settings...` and add the USB dongle in the `USB Devices Filters` list.

2.3 MicroEJ Runtime

2.3.1 Language

MicroEJ is compatible with the Java language version 7.

Java source code is compiled by the Java compiler[1] into the binary format specified in the JVM specification[2]. This binary code needs to be linked before execution: .class files and some other application-related files (see *MicroEJ Classpath*) are compiled to produce the final application that the MicroEJ Runtime can execute.

MicroEJ complies with the deterministic class initialization (<clinit>) order specified in *[BON]*. 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.

2.3.2 Scheduler

The MicroEJ Architecture features a green thread platform that can interact with the C world *[SNI]*. The (green) thread policy is as follows:

- preemptive for different priorities,
- round-robin for same priorities,
- “priority inheritance protocol” when priority inversion occurs.[3]

MicroEJ stacks (associated with the threads) automatically adapt their sizes according to the thread requirements: Once the thread has finished, its associated stack is reclaimed, freeing the corresponding RAM memory.

[1] The JDT compiler from the Eclipse IDE.
[2] Tim Lindholm & Frank Yellin, The Java™ Virtual Machine Specification, Second Edition, 1999
[3] This protocol raises the priority of a thread (that is holding a resource needed by a higher priority task) to the priority of that task.

2.3.3 Garbage Collector

The MicroEJ Architecture includes a state-of-the-art memory management system, the Garbage Collector (GC). It manages a bounded piece of RAM memory, devoted to the Java world. The GC automatically frees dead Java objects, and defragments the memory in order to optimize RAM usage. This is done transparently while the MicroEJ Applications keep running.

2.3.4 Foundation Libraries

Embedded Device Configuration (EDC)

The Embedded Device Configuration specification defines the minimal standard runtime environment for embedded devices. It defines all default API packages:

- java.io
- java.lang
- java.lang.annotation
- java.lang.ref
- java.lang.reflect
- java.util

Beyond Profile (BON)

[BON] defines a suitable and flexible way to fully control both memory usage and start-up sequences on devices with limited memory resources. It does so 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.
- Defining and accessing compile-time constants.

2.4 MicroEJ Libraries

A MicroEJ Foundation Library is a MicroEJ Core library that provides core runtime APIs or hardware-dependent functionality. A Foundation library is divided into 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 MicroEJ Platforms. From a MicroEJ Classpath, Foundation Library APIs dependencies are automatically mapped to the associated implementations provided by the Platform or the Virtual Device on which the application is being executed.

A MicroEJ Add-On Library is a MicroEJ library that is implemented on top of MicroEJ Foundation Libraries (100% full Java code). A MicroEJ Add-On Library is distributed in a single JAR file, with a 3 digits version and provides its associated source code.

Foundation and Add-On Libraries are added to MicroEJ Classpath by the application developer as module dependencies (see *MicroEJ Module Manager*).

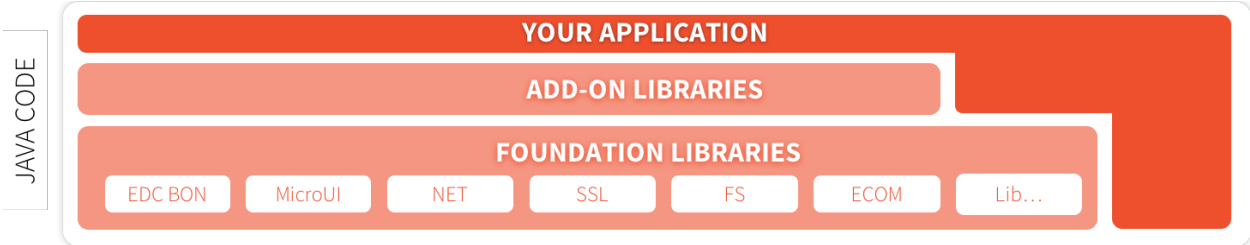


Fig. 9: MicroEJ Foundation Libraries and Add-On Libraries

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/.

2.5 MicroEJ Central Repository

The MicroEJ Central Repository is the binary repository maintained by MicroEJ Corp. It contains Foundation Library APIs and numerous Add-On Libraries. Foundation Libraries APIs are distributed under the organization `ej.api` and `com.microej.api` . All other artifacts are Add-On Libraries.

By default, MicroEJ SDK is configured to connect online MicroEJ Central Repository. The MicroEJ Central Repository can be downloaded locally for offline use. Please follow the steps described at https://developer.microej.com/central-repository/.

To consult its *libraries* APIs documentation, please visit https://developer.microej.com/microej-apis/.

2.6 Embedded Specification Requests

MicroEJ implements the following ESR Consortium specifications:

[BON]	http://e-s-r.net/download/specification/ESR-SPE-0001-BON-1.2-F.pdf
[SNI]	http://e-s-r.net/download/specification/ESR-SPE-0012-SNI_GT-1.2-H.pdf
[SP]	http://e-s-r.net/download/specification/ESR-SPE-0014-SP-2.0-A.pdf
[MUI]	http://e-s-r.net/download/specification/ESR-SPE-0002-MICROUI-2.0-B.pdf
[KF]	http://e-s-r.net/download/specification/ESR-SPE-0020-KF-1.4-F.pdf

2.7 MicroEJ Firmware

2.7.1 Bootable Binary with Core Services

A MicroEJ Firmware is a binary software program that can be programmed into the flash memory of a device. A MicroEJ Firmware includes an instance of a MicroEJ runtime linked to:

- underlying native libraries and BSP + RTOS,
- MicroEJ libraries and application code (C and Java code).

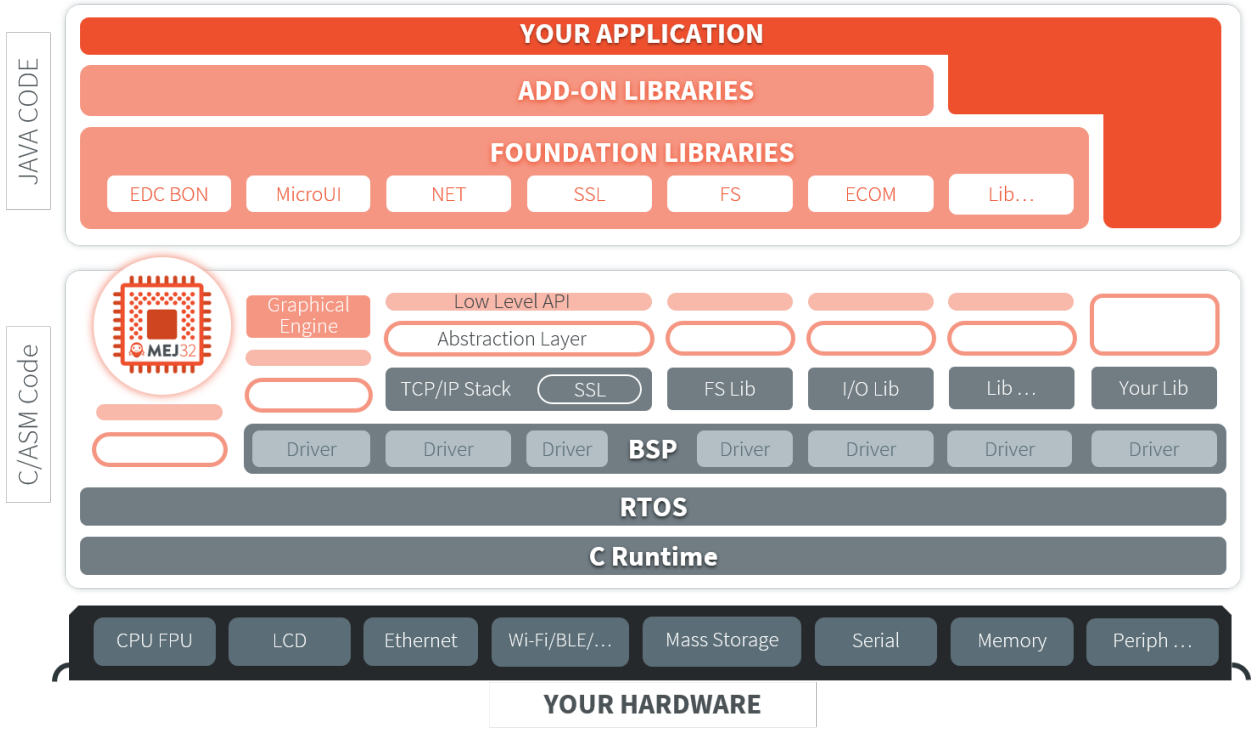


Fig. 10: MicroEJ Firmware Architecture

2.7.2 Specification

The set of libraries included in the firmware and its dimensioning limitations (maximum number of simultaneous threads, open connections, …) are firmware specific. Please refer to https://developer.microej.com/5/getting-started-studio.html for evaluation firmware release notes.

2.8 Introducing MicroEJ SDK

MicroEJ SDK provides tools based on Eclipse to develop software applications for MicroEJ-ready devices. MicroEJ SDK allows application developers to write MicroEJ Applications and run them on a virtual (simulated) or real device.

This document is a step-by-step introduction to application development with MicroEJ SDK. The purpose of MicroEJ SDK is to develop for targeted MCU/MPU computers (IoT, wearable, etc.) and it is therefore a cross-development tool.

Unlike standard low-level cross-development tools, MicroEJ SDK offers unique services like hardware simulation and local deployment to the target hardware.

Application development is based on the following elements:

- MicroEJ SDK, the integrated development environment for writing applications. It is based on Eclipse and is relies on the integrated Java compiler (JDT). It also provides a dependency manager for managing MicroEJ Libraries (see *MicroEJ Module Manager*). The current distribution of MicroEJ SDK (`20.10`) is built on top of Eclipse 2020-06.
- MicroEJ Platform, a software package including the resources and tools required for building and testing an application for a specific MicroEJ-ready device. MicroEJ Platforms are imported into MicroEJ SDK within a

local folder called MicroEJ Platforms repository. Once a MicroEJ Platform is imported, an application can be launched and tested on Simulator. It also provides a mean to locally deploy the application on a MicroEJ-ready device.

- MicroEJ-ready device, an hardware device that will be programmed with a MicroEJ Firmware. A MicroEJ Firmware is a binary instance of MicroEJ runtime for a target hardware board.

Starting from scratch, the steps to go through the whole process are detailed in the following sections of this chapter :

- *Download and install a MicroEJ Platform*
- *Build and run your first application on Simulator*
- *Build and run your first application on target hardware*

2.9 Introducing MicroEJ Studio and Virtual Devices

MicroEJ Studio provides tools based on Eclipse to develop software applications for MicroEJ-ready devices. MicroEJ Studio allows application developers to write MicroEJ Applications, run them on a virtual (simulated) or real device, and publish them to a MicroEJ Forge instance.

This document is an introduction to application development with MicroEJ Studio. The purpose of MicroEJ Studio is to develop for targeted MCU/MPU computers (IoT, wearable, etc.) and it is therefore a cross-development tool.

Unlike standard low-level cross-development tools, MicroEJ Studio offers unique services like hardware simulation, deployment to the target hardware and final publication to a MicroEJ Forge instance.

Application development is based on the following elements:

- MicroEJ Studio, the integrated development environment for writing applications. It is based on Eclipse and relies on the integrated Java compiler (JDT). It also provides a dependency manager for managing MicroEJ Libraries (see *MicroEJ Module Manager*). The current distribution of MicroEJ Studio (`19.05`) is built on top of Eclipse Oxygen (https://www.eclipse.org/oxygen/).
- MicroEJ Virtual Device, a software package including the resources and tools required for building and testing an application for a specific MicroEJ-ready 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.

  Virtual Devices are imported into MicroEJ Studio within a local folder called MicroEJ Repository. Once a Virtual Device is imported, an application can be launched and tested on Simulator. It also provides a mean to locally deploy the application on a MicroEJ-ready device.
- MicroEJ-ready device, a hardware device that has been previously programmed with a MicroEJ Firmware. A MicroEJ Firmware is a binary instance of MicroEJ runtime for a target hardware board. MicroEJ-ready devices are built using MicroEJ SDK. MicroEJ Virtual Devices and MicroEJ Firmwares share the same version (there is a 1:1 mapping).

The following figure gives an overview of MicroEJ Studio possibilities:

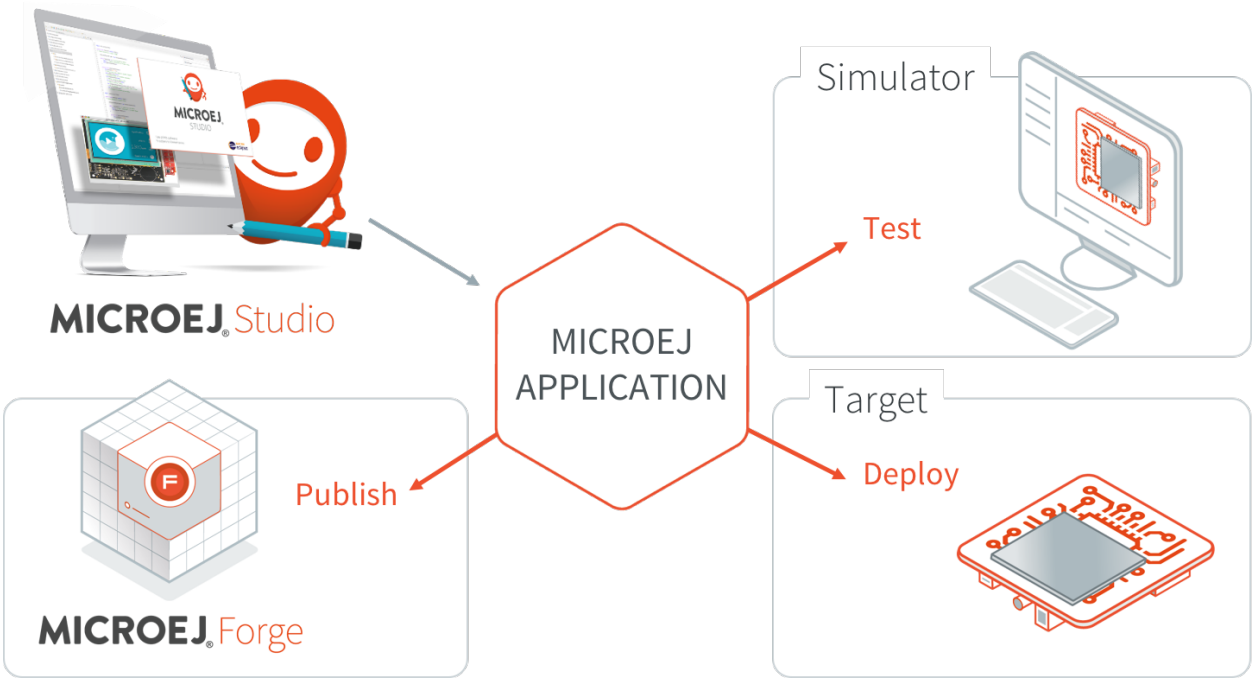


Fig. 11: MicroEJ Application Development Overview

2.10 Perform Online Getting Started

MicroEJ Studio Getting Started is available on https://developer.microej.com/5/getting-started-studio.html.

Starting from scratch, the steps to go through the whole process are:

1. Setup a board and test a MicroEJ Firmware:
    - Select between one of the available boards;
    - Download and install a MicroEJ Firmware on the target hardware;
    - Deploy and run a MicroEJ demo on board.
2. Setup and learn to use development tools:
    - Download and install MicroEJ Studio;
    - Download and install the corresponding Virtual Device for the target hardware;
    - Download, build and run your first application on Simulator;
    - Build and run your first application on target hardware.

The following figure gives an overview of the MicroEJ software components required for both host computer and target hardware:

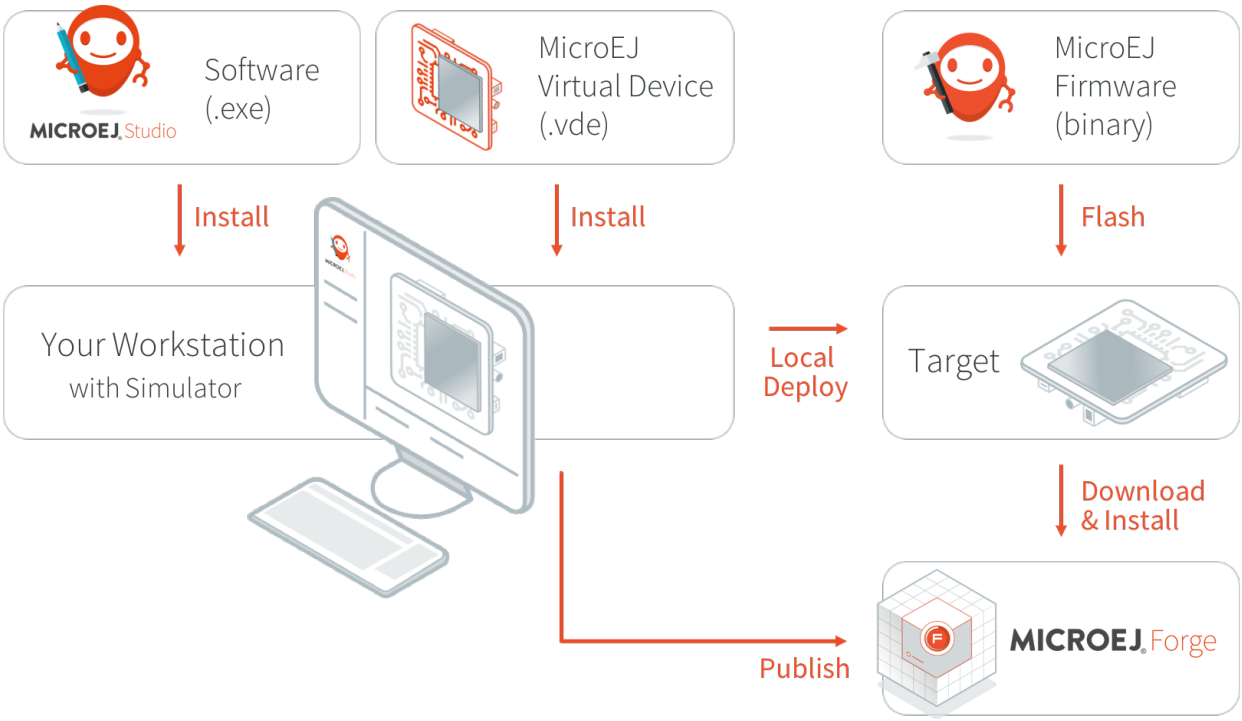


Fig. 12: MicroEJ Studio Development Imported Elements

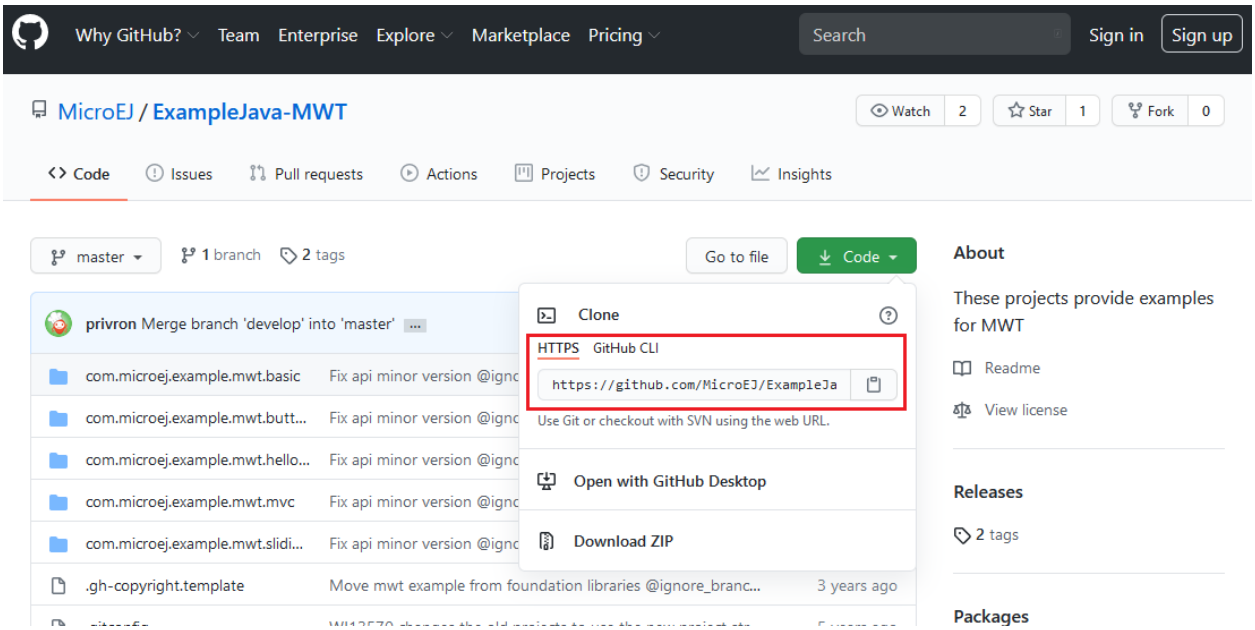
2.11 GitHub Repositories

A large number of examples, libraries, demos and tools are shared on MicroEJ GitHub account: https://github.com/MicroEJ.

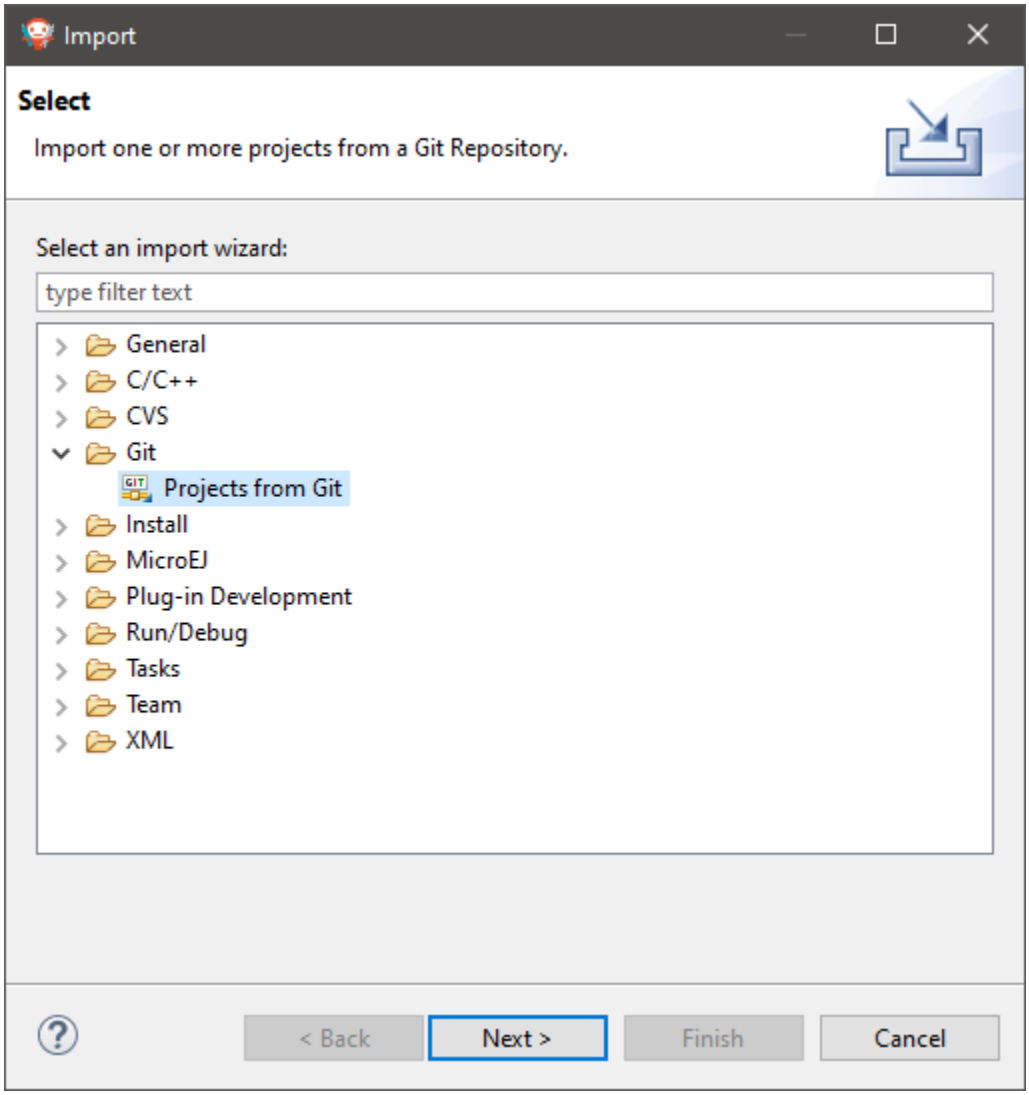
Most of these GitHub repositories contain projects ready to be imported in MicroEJ SDK. This section explains the steps to import them in MicroEJ SDK, using the MWT Examples repository.

Note: MicroEJ SDK Distribution includes the Eclipse plugin for Git.

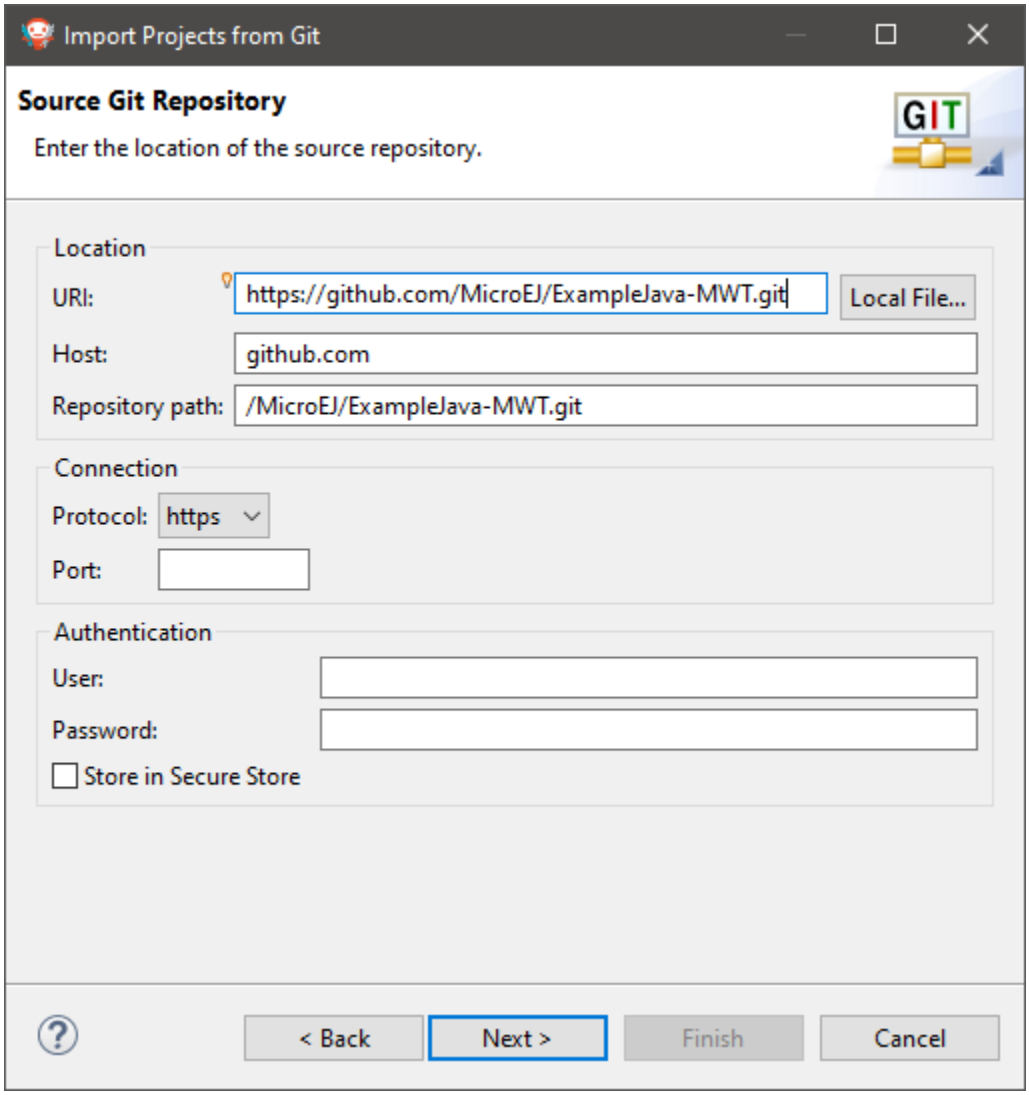
First, from the GitHub page, copy the repository URI (HTTP address) from the dedicated field in the right menu (highlighted in red):



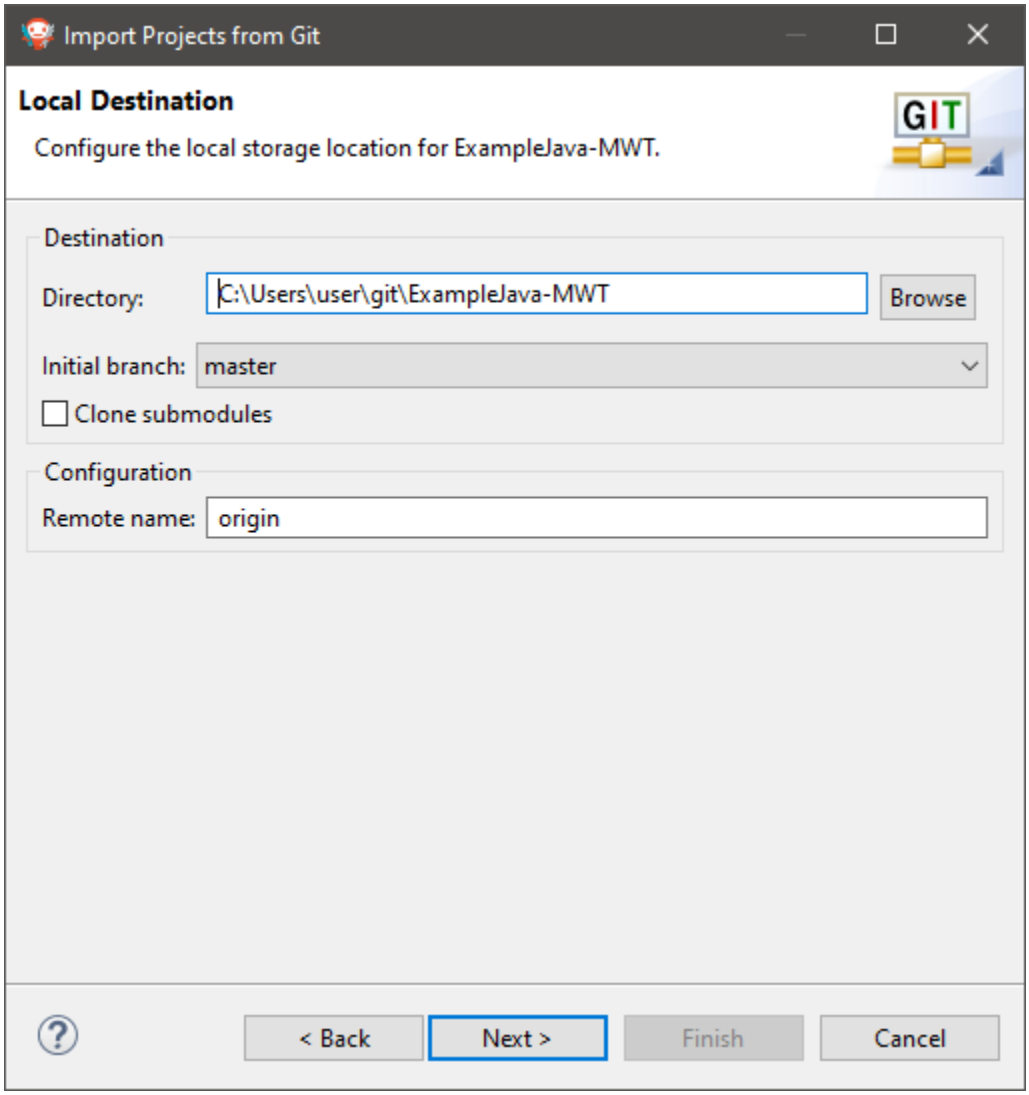
In MicroEJ SDK, to clone and import the project from the remote Git repository into the MicroEJ workspace, select File > Import > Git > Projects from Git wizard.



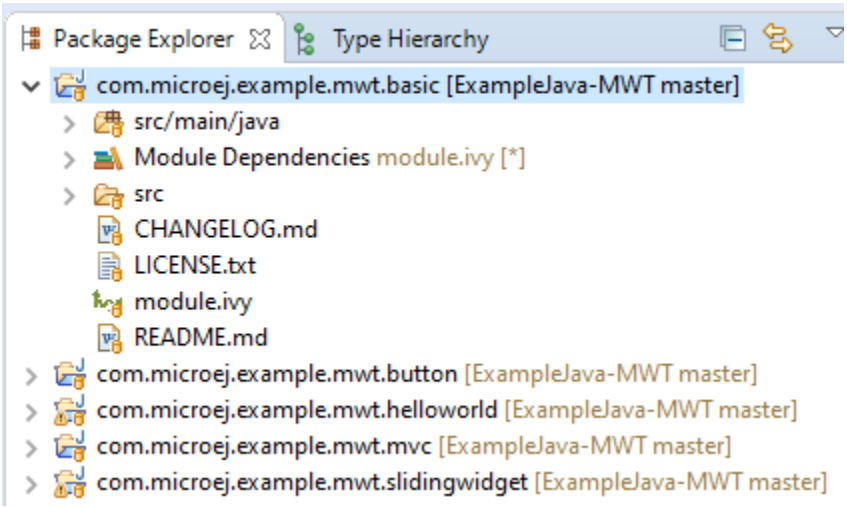
Click Next , select Clone URI , click Next and paste the remote repository address in the URI field. For this repository, the address is https://github.com/MicroEJ/ExampleJava-MWT.git. If the HTTP address is a valid repository, the other fields are filed automatically.



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.

2.12 System Requirements

MicroEJ SDK and MicroEJ Studio

- **Intel x64 PC with minimum :**
  - Dual-core Core i5 processor
  - 4GB RAM
  - 2GB Disk
- **Operating Systems :**
  - Windows 10, Windows 8.1 or Windows 8
  - Linux distributions (tested on Ubuntu 18.04 and 20.04) - As of SDK 20.10 (based on Eclipse 2020-06), Ubuntu 16.04 is not supported.
  - Mac OS X (tested on version 10.13 High Sierra, 10.14 Mojave)
- **Java :**
  - JRE or JDK 8 (OpenJDK or Oracle JDK)

CHAPTER

THREE

APPLICATION DEVELOPER GUIDE

3.1 Introduction

The following sections of this document shall prove useful as a reference when developing applications for MicroEJ. They cover concepts essential to MicroEJ Applications design.

In addition to these sections, by going to https://developer.microej.com/, you can access a number of helpful resources such as:

- Libraries from the MicroEJ Central Repository (https://developer.microej.com/central-repository/);
- Application Examples as source code from MicroEJ Github Repositories (https://github.com/MicroEJ);
- Documentation (HOWTOs, Reference Manuals, APIs javadoc…).

MicroEJ Applications are developed as standard Java applications on Eclipse JDT, using Foundation Libraries. MicroEJ SDK allows you to run / debug / deploy MicroEJ Applications on a MicroEJ Platform.

Two kinds of applications can be developed on MicroEJ: MicroEJ Standalone Applications and MicroEJ Sanboxed Applications.

A MicroEJ Standalone Application is a MicroEJ Application that is directly linked to the C code to produce a MicroEJ Firmware. Such application must define a main entry point, i.e. a class containing a `public static void main(String[])` method. MicroEJ Standalone Applications are developed using MicroEJ SDK.

A MicroEJ Sandboxed Application is a MicroEJ Application that can run over a Multi-Sandbox Firmware. It can be linked either statically or dynamically. If it is statically linked, it is then called a System Application as it is part of the initial image and cannot be removed. MicroEJ Sandboxed Applications are developed using MicroEJ Studio.

3.2 Local Workspaces and Repositories

When starting MicroEJ SDK, it prompts you to select the last used workspace or a default workspace on the first run. A workspace is a main folder where to find a set of projects containing MicroEJ source code.

When loading a new workspace, MicroEJ SDK prompts for the location of the MicroEJ repository, where the MicroEJ Architectures, Platforms or Virtual Devices will be imported. By default, MicroEJ SDK suggests to point to the default MicroEJ repository on your operating system, located at `${user.home}/.microej/repositories/[version]`. You can select an alternative location. 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.

3.3 Standalone Application

3.3.1 Download and Install a MicroEJ Platform

MicroEJ SDK being a cross development tool, it does not build software targeted to your host desktop platform. In order to run MicroEJ Applications, a target hardware is required. Several commercial targets boards from main MCU/MPU chip manufacturers can be prepared to run MicroEJ Applications, you can also run your applications without one of these boards with the help of a MicroEJ Simulator.

A MicroEJ Platform is a software package including the resources and tools required for building and testing an application for a specific MicroEJ-ready device. MicroEJ Platforms are available at https://developer.microej.com/5/getting-started-sdk.html.

After downloading the MicroEJ Platform, launch MicroEJ SDK on your desktop to start the process of Platform installation :

- Open the Platform view in MicroEJ SDK, select Window > Preferences > MicroEJ > Platforms . The view should be empty on a fresh install of the tool

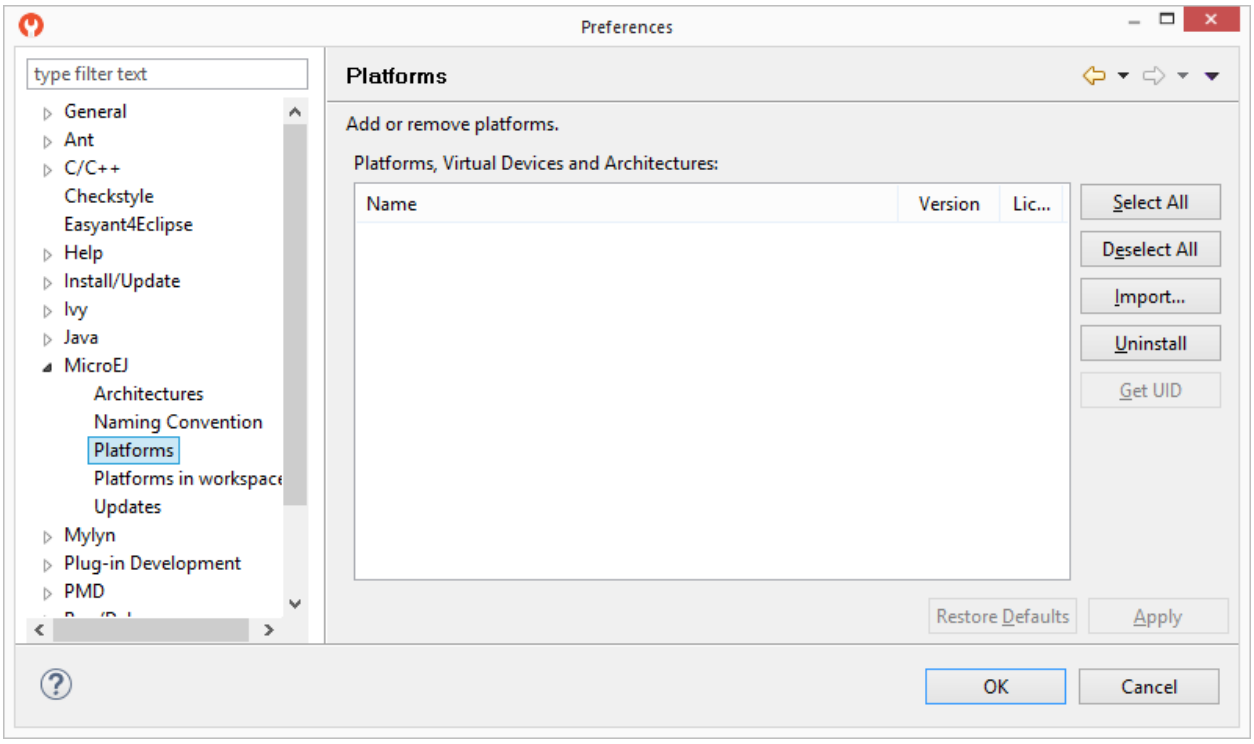


Fig. 1: MicroEJ Platform Import

- Press Import… button.
- Choose Select File… and use the Browse option to navigate to the `.jpf` file containing your MicroEJ Platform, then read and accept the license agreement to proceed.

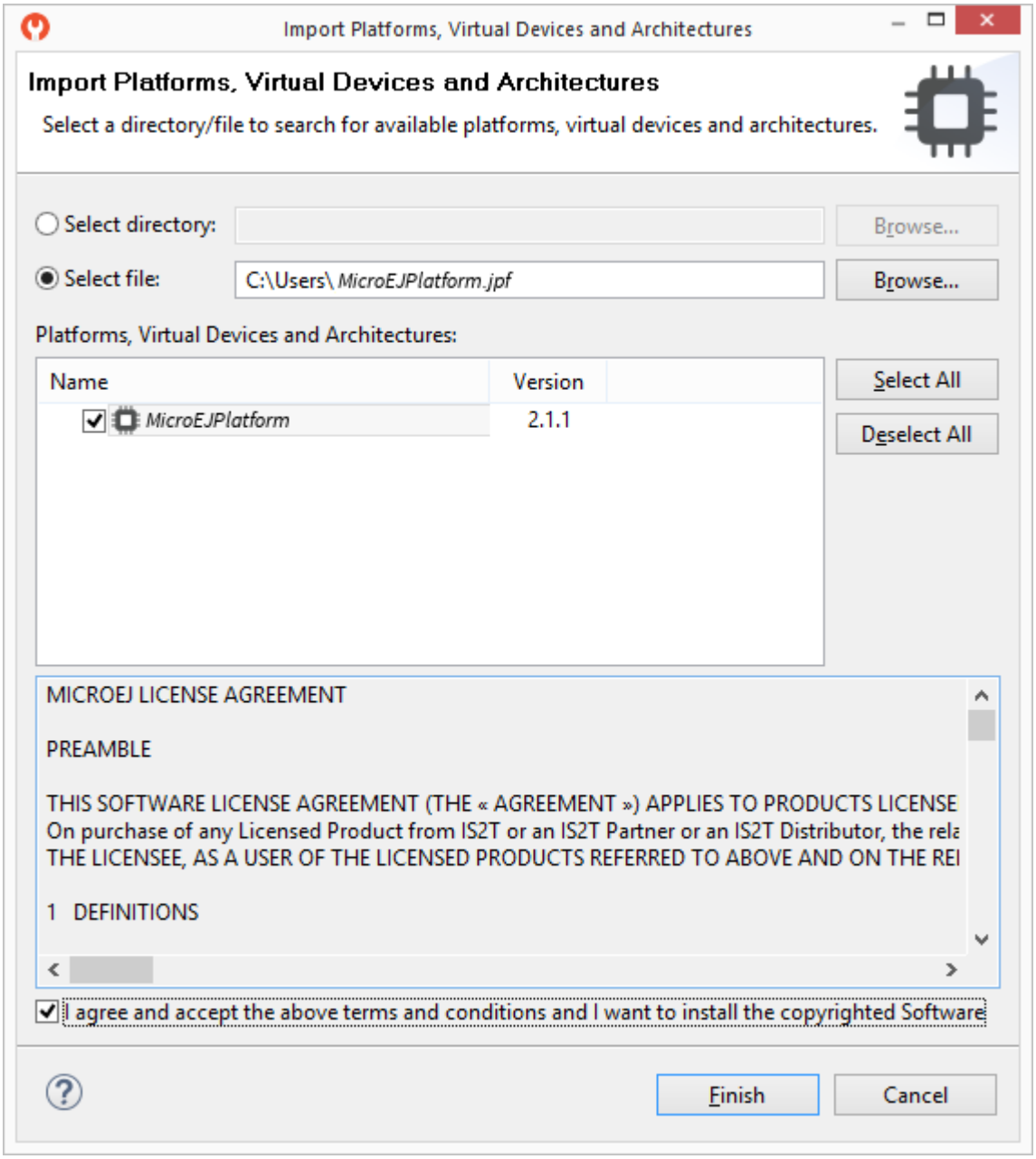


Fig. 2: MicroEJ Platform Selection

- The MicroEJ Platform should now appear in the Platforms view, with a green valid mark.

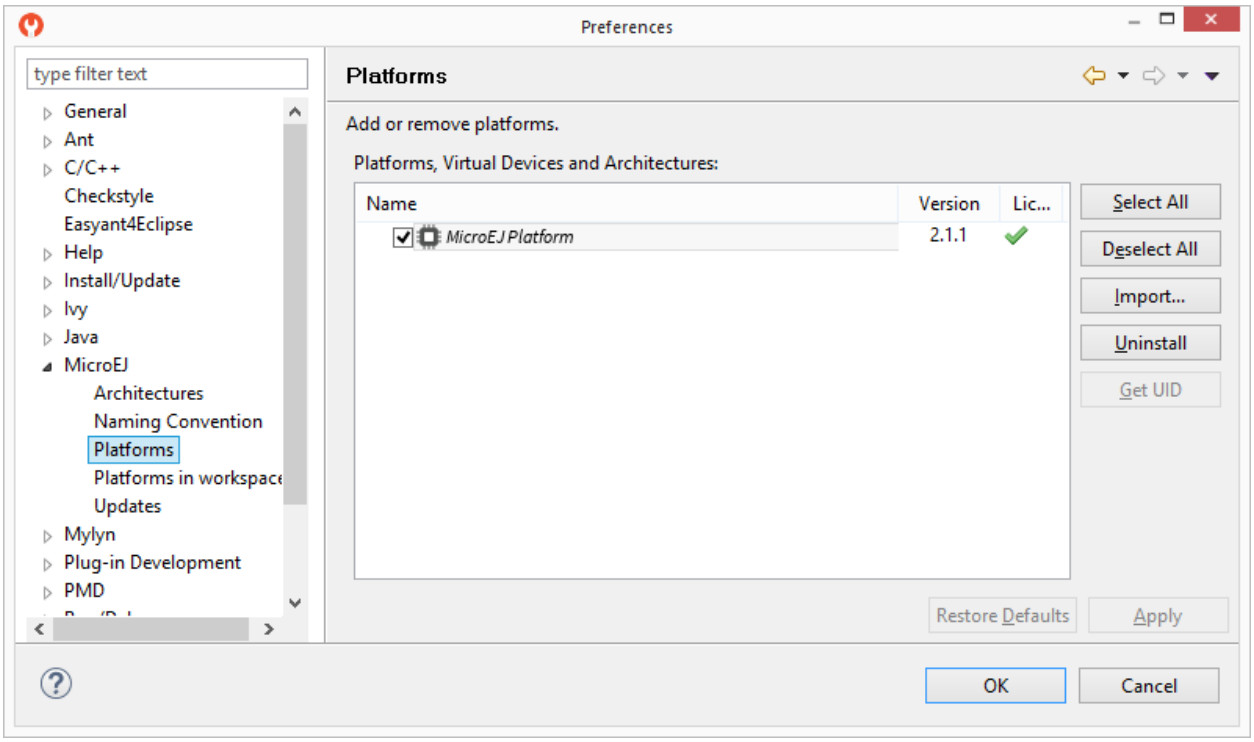


Fig. 3: MicroEJ Platform List

3.3.2 Build and Run an Application

Create a MicroEJ Standalone Application

- Create a project in your workspace. Select File > New > MicroEJ Standalone Application Project .

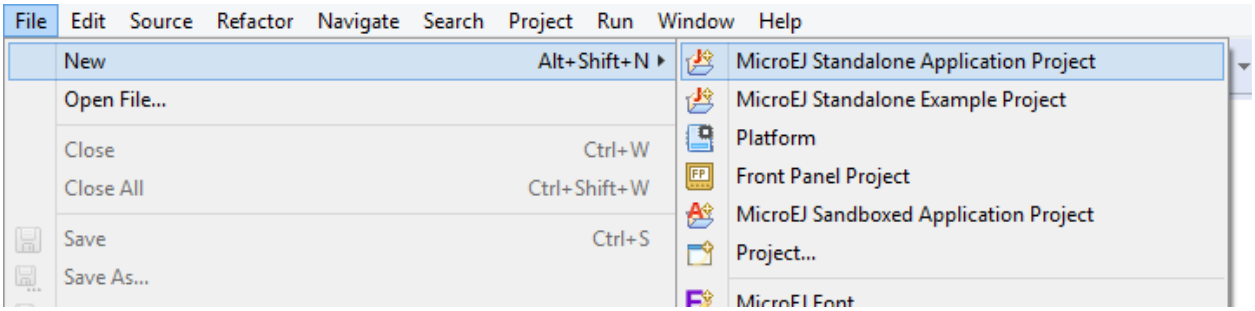


Fig. 4: New MicroEJ Standalone Application Project

- Fill in the application template fields, the Project name field will automatically duplicate in the following fields. 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 (images, fonts, etc.)

  - `META-INF` : Sandboxed Application configuration and resources
  - `module.ivy` : Ivy input file, dependencies description for the current project

- Right click on the source folder `src/main/java` and select New > Package . Give a name: `com.mycompany` . Click on Finish .

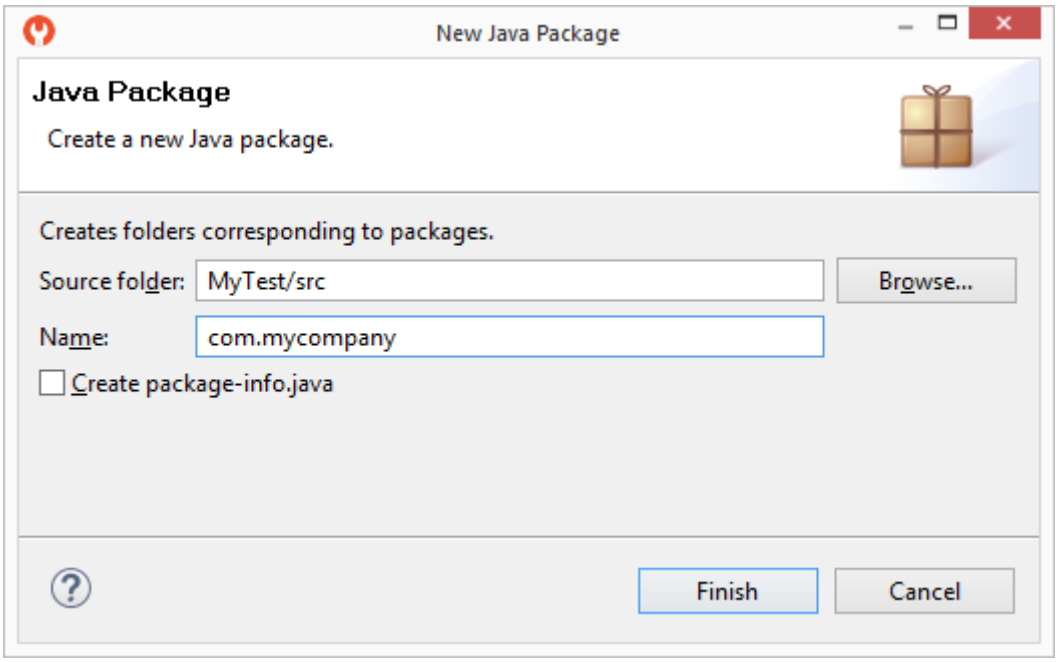


Fig. 5: New Package

- The package `com.mycompany` is available under `src/main/java` folder. Right click on this package and select New > Class . Give a name: `Test` and check the box `public static void main(String[] args)` . Click on Finish .

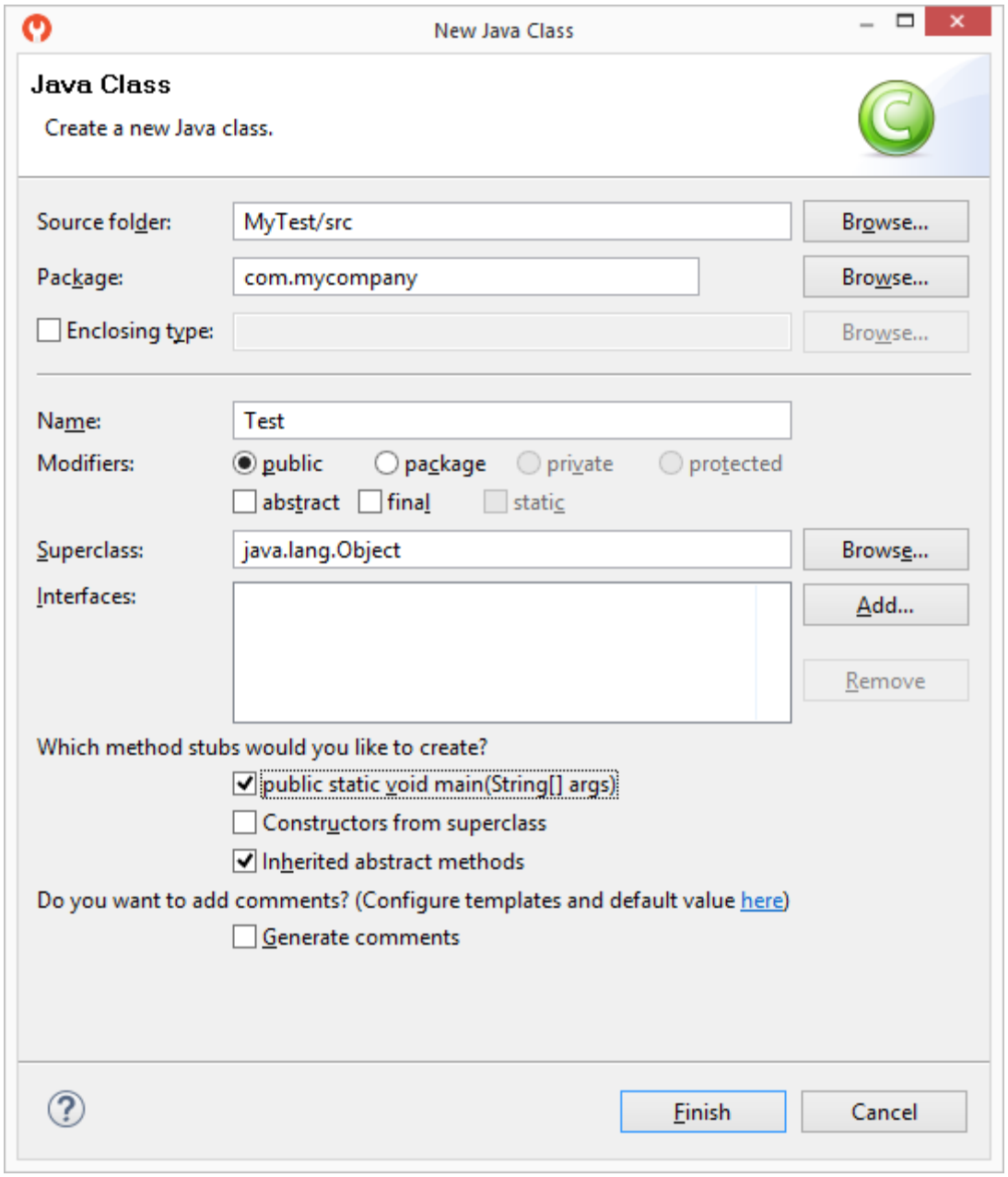


Fig. 6: New Class

- The new class has been created with an empty `main()` method. Fill the method body with the following lines:

```
System.out.println("hello world!");
```

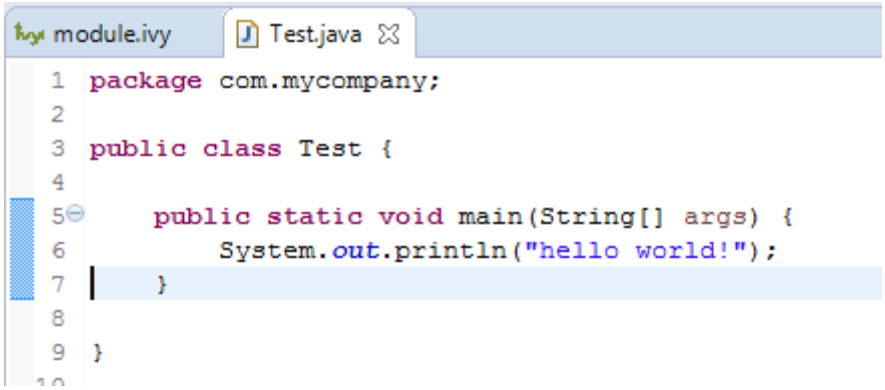


Fig. 7: MicroEJ Application Content

The test application is now ready to be executed. See next sections.

Run on the Simulator

To run the sample project on Simulator, select it in the left panel then right-click and select Run > Run as > MicroEJ Application .

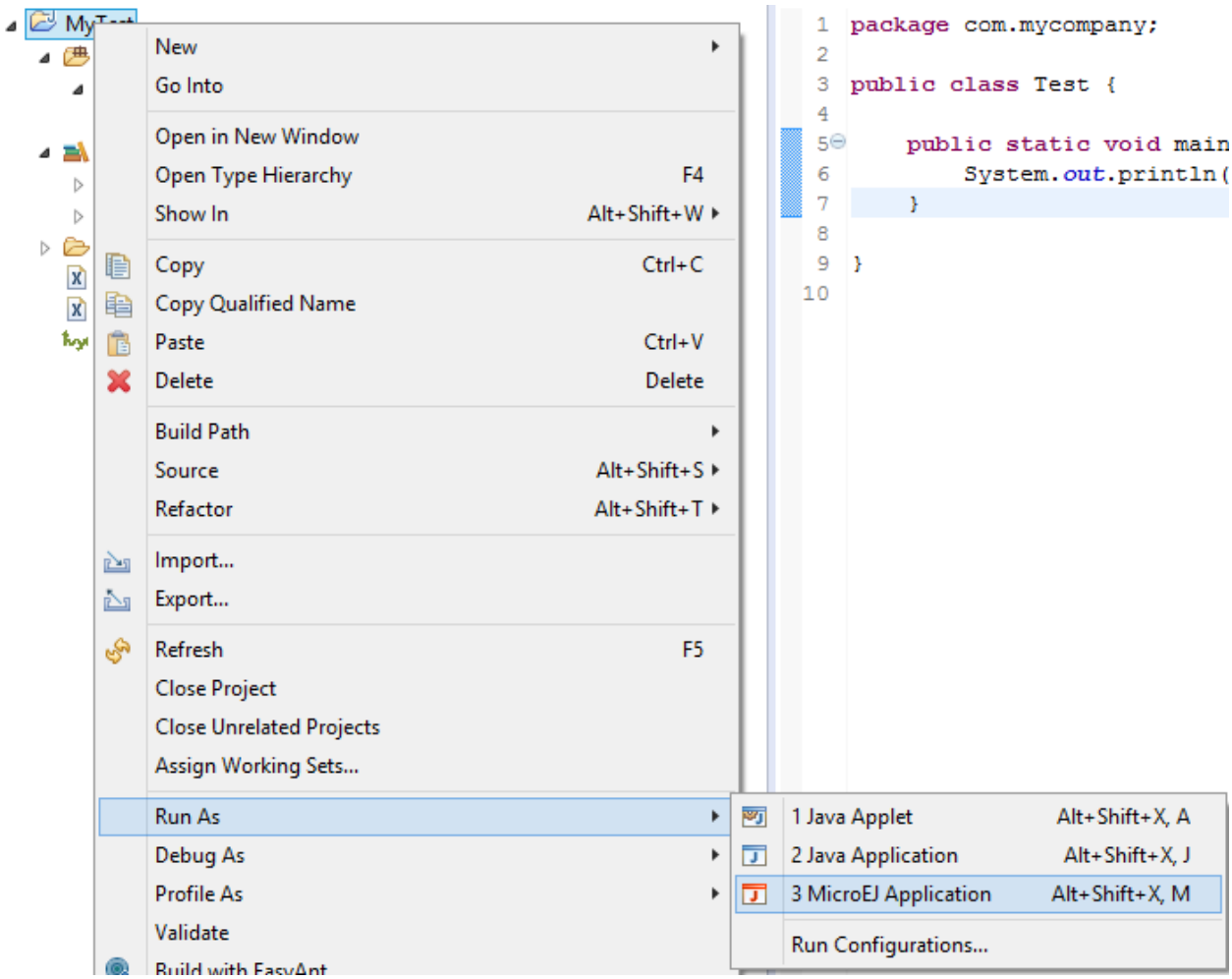


Fig. 8: MicroEJ Development Tools Overview

MicroEJ SDK console will display Launch steps messages.

```
=============== [ Initialization Stage ] ===============
=============== [ Launching on Simulator ] ===============
hello world!
=============== [ Completed Successfully ] ===============

SUCCESS
```

Run on the Hardware Device

Compile an application, connect the hardware device and deploy on it is hardware dependant. These steps are described in dedicated documentation available inside the MicroEJ Platform. This documentation is accessible from the MicroEJ Resources Center view.

Note: MicroEJ Resources Center view may have been closed. Click on Help > MicroEJ Resources Center to reopen it.

Open the menu Manual and select the documentation `[hardware device] MicroEJ Platform`, where `[hardware device]` is the name of the hardware device. This documentation features a guide to run a built-in application on MicroEJ Simulator and on hardware device.

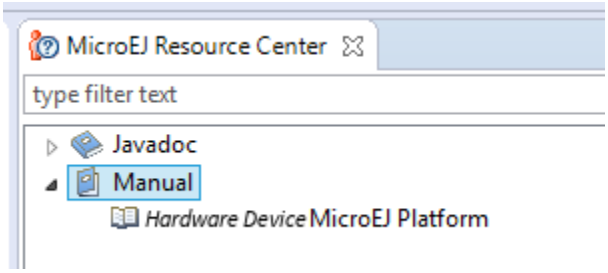


Fig. 9: MicroEJ Platform Guide

3.3.3 Build Output Files

When building a MicroEJ Application, multiple files are generated next to the ELF file. These 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.

The following image shows an example of output folder:

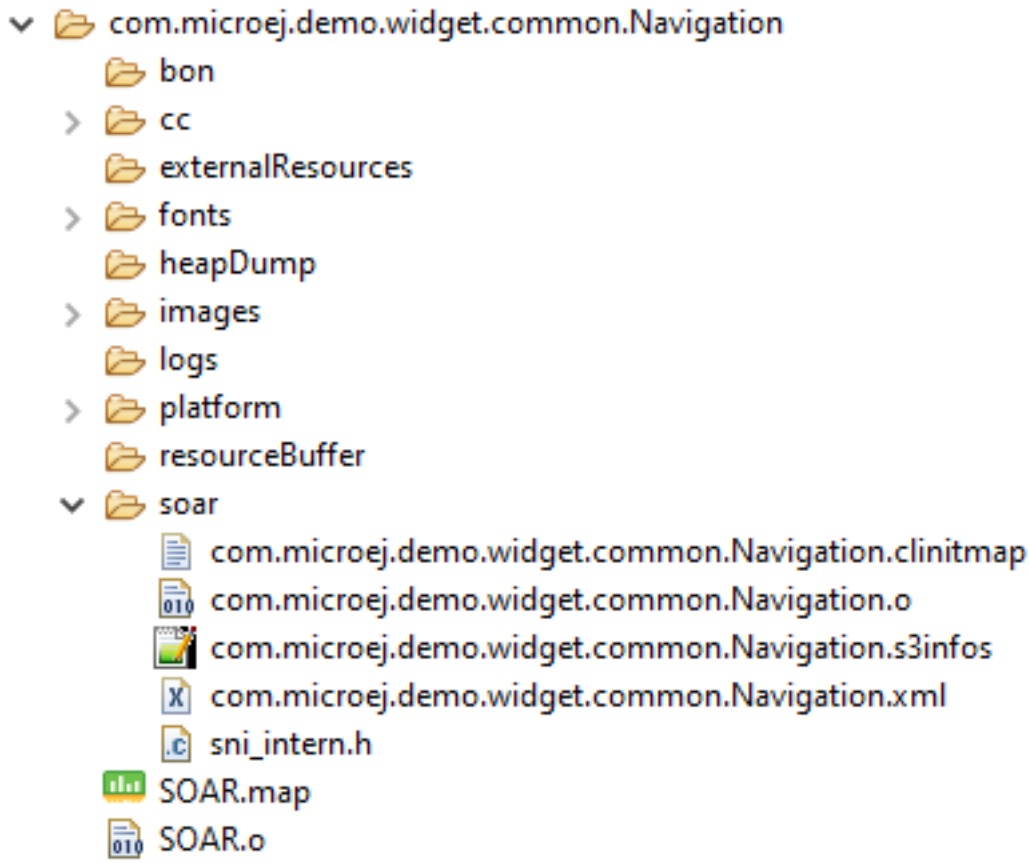


Fig. 10: Build Output Files

The SOAR Map File

The `SOAR.map` file lists every embedded symbol of the application (section, Java class or method, etc.) and its size in ROM or RAM. 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/<main class>.xml` file can be opened using any XML editor.

This file contains the list of the following embedded elements:

- method (in `selected_methods` tag)
- *resource* (in `selected_resources` tag)
- *system property* (in `java_properties` tag)
- string (in `selected_internStrings` tag)
- *type* (in `selected_types` tag)
- *immutable* (in `selected_immutables` tag)

3.3.4 MicroEJ Launch

The MicroEJ launch configuration sets up the *MicroEJ Applications* environment (main class, resources, target platform, and platform-specific options), and then launches a MicroEJ launch script for execution.

Execution is done on either the MicroEJ Platform or the MicroEJ Simulator. The launch operation is platform-specific. It may depend on external tools that the platform requires (such as target memory programming). Refer to the platform-specific documentation for more information about available launch settings.

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.

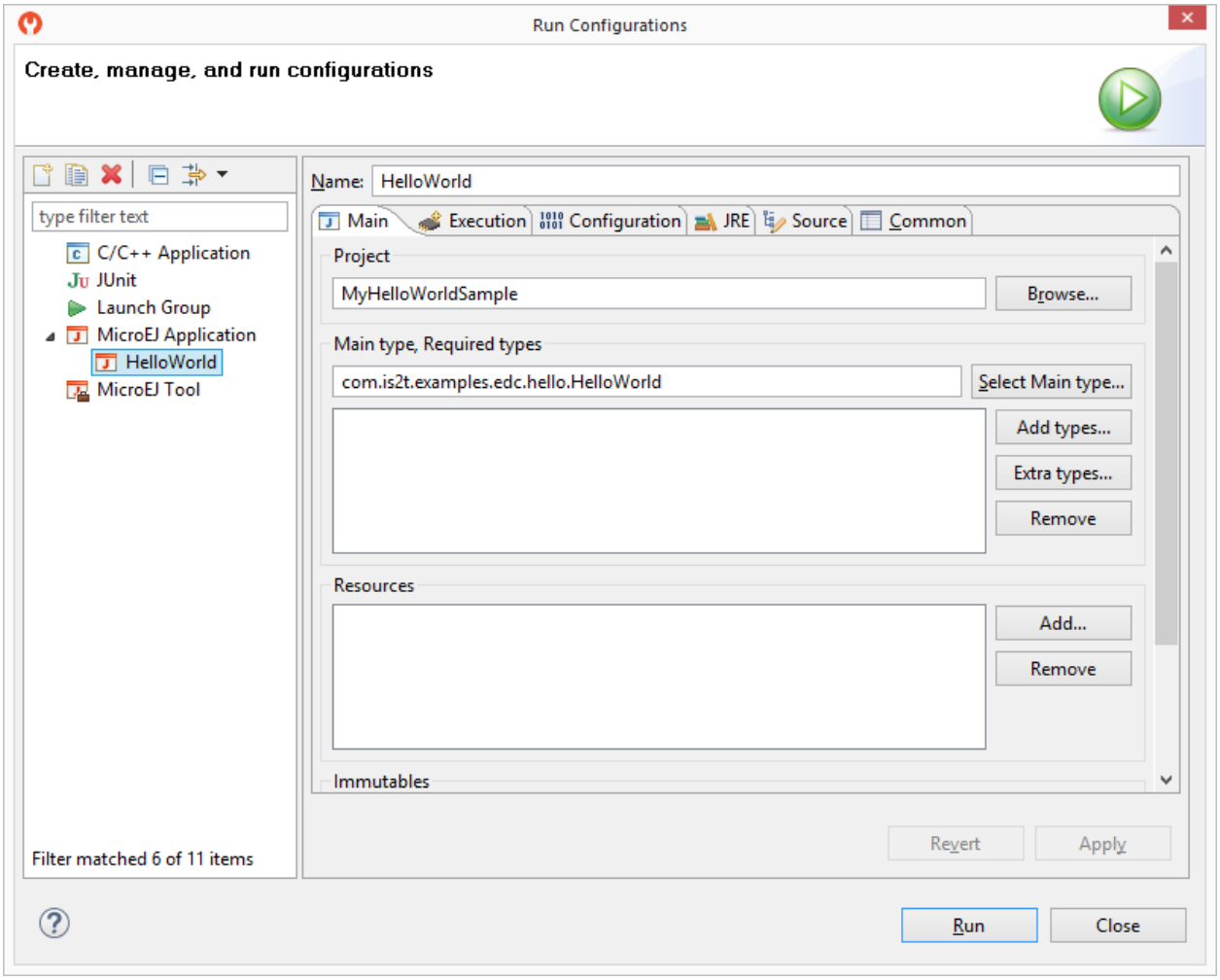


Fig. 11: MicroEJ Launch Application Main Tab

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).

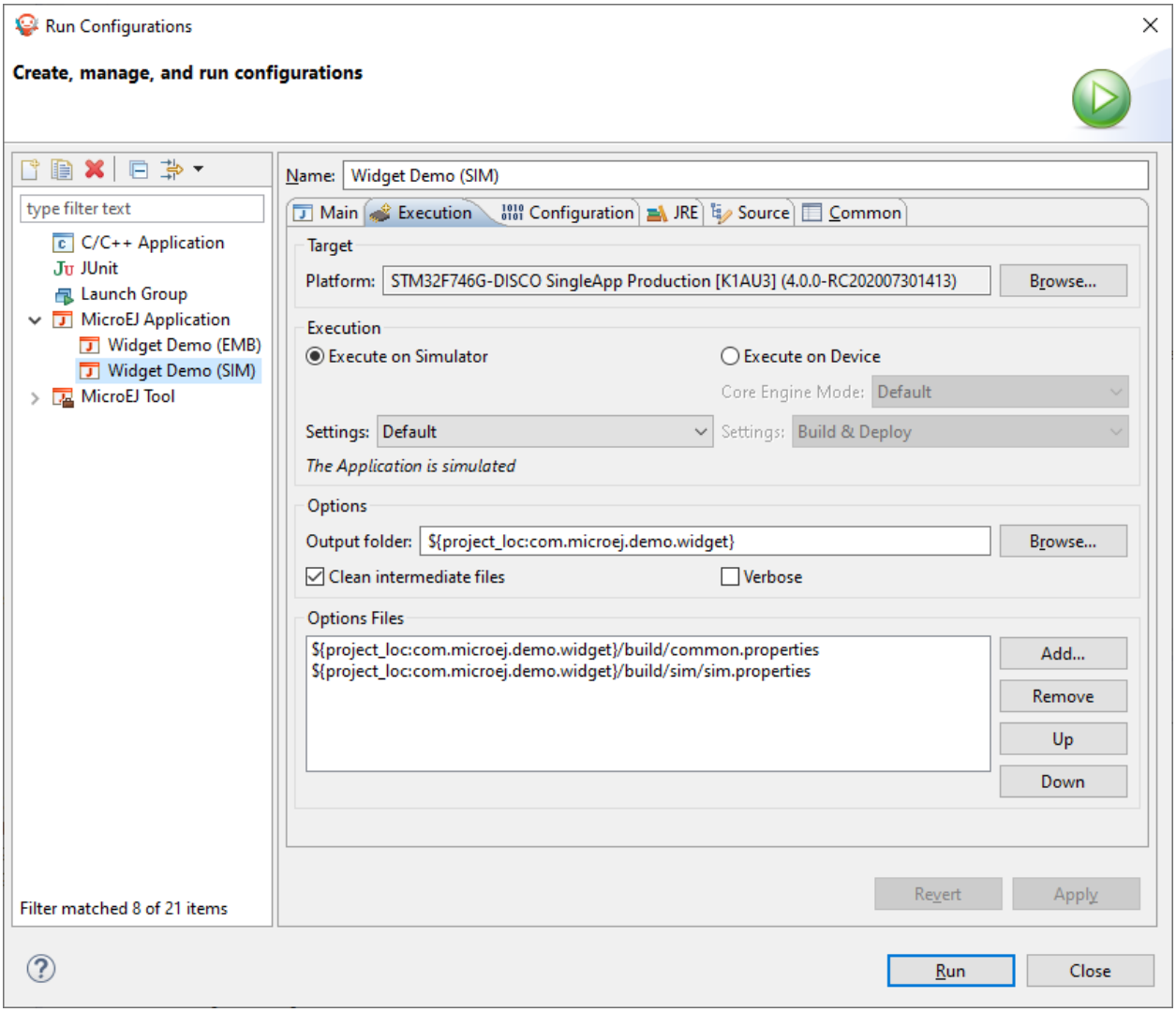


Fig. 12: MicroEJ Launch Application Execution Tab

Configuration Tab

The next tab is the Configuration tab. This tab contains all platform-specific options.

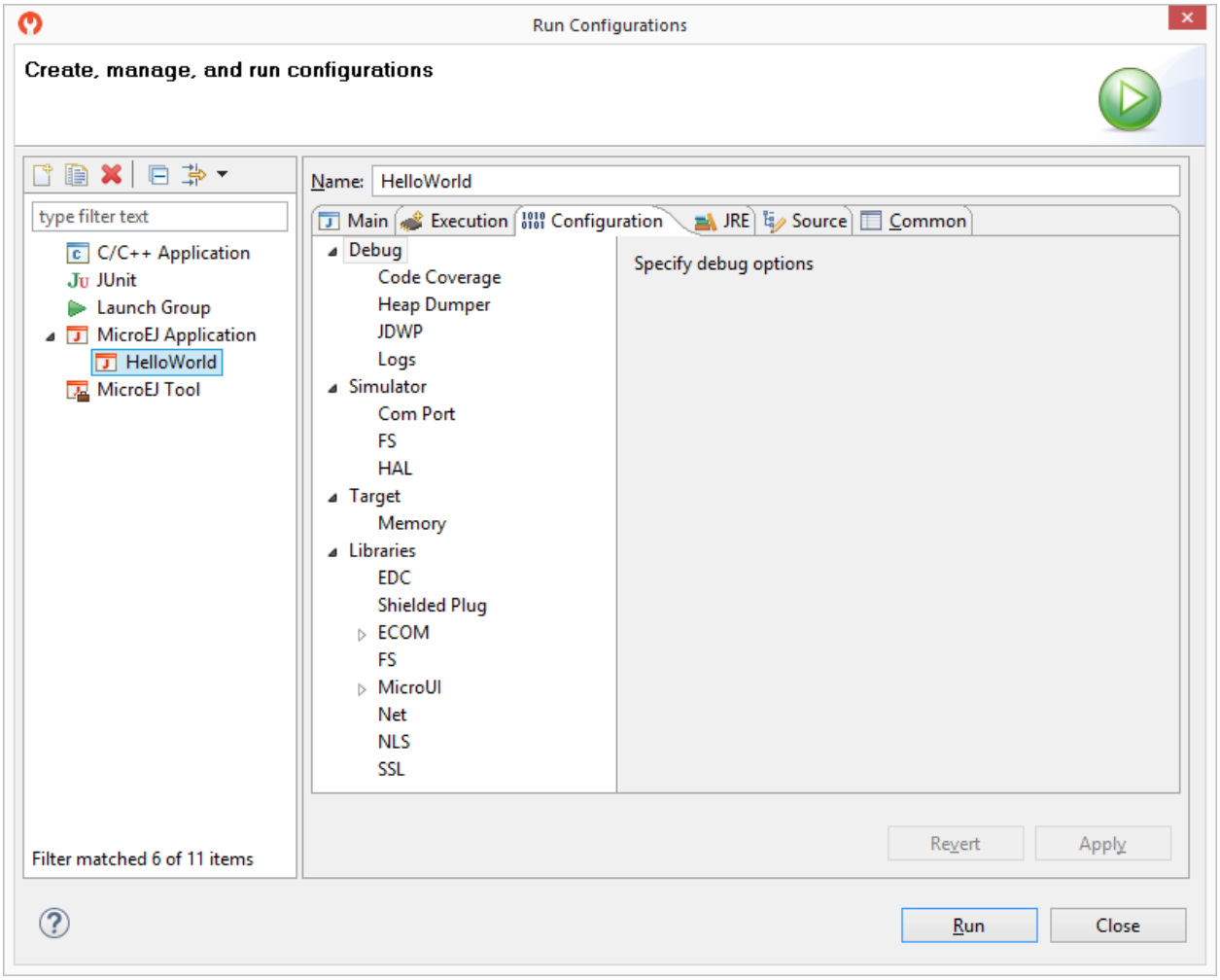


Fig. 13: Configuration Tab

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.

Other Tabs

The next tabs (Source and Common tabs) are the default Eclipse launch tabs. Refer to Eclipse help for more details on how to use these launch tabs.

3.3.5 Application Options

Introduction

To run a MicroEJ Standalone Application on a MicroEJ Platform, a set of options must be defined. Options can be of different types:

- Memory Allocation options (e.g set the Java Heap size). These options are usually called link-time options.
- Simulator & Debug options (e.g. enable periodic Java 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 MicroEJ Architecture. Please consult the appropriate MicroEJ Pack documentation for options related to other Foundation Libraries (MicroUI, NET, SSL, FS, ...) integrated to the Platform.

Notice that some options may not be available, in the following cases:

- Option is specific to the MicroEJ Core Engine capability (*tiny*/*single*/*multi*) which is integrated in the targeted Platform.
- Option is specific to the target (MicroEJ Core Engine on Device or Simulator).
- Option has been introduced in a newer version of the MicroEJ Architecture which is integrated in the targeted Platform.
- Options related to *Board Support Package (BSP) connection*.

Defining an Option

A MicroEJ 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 MicroEJ Studio/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.

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 MicroEJ 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:

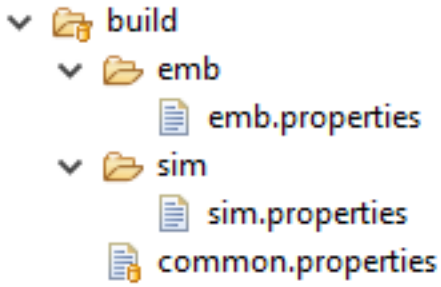


Fig. 14: Build Options Folder

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=false`.

To use the options declared in properties files in a launcher, perform the following steps:

1. In MicroEJ Studio/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`

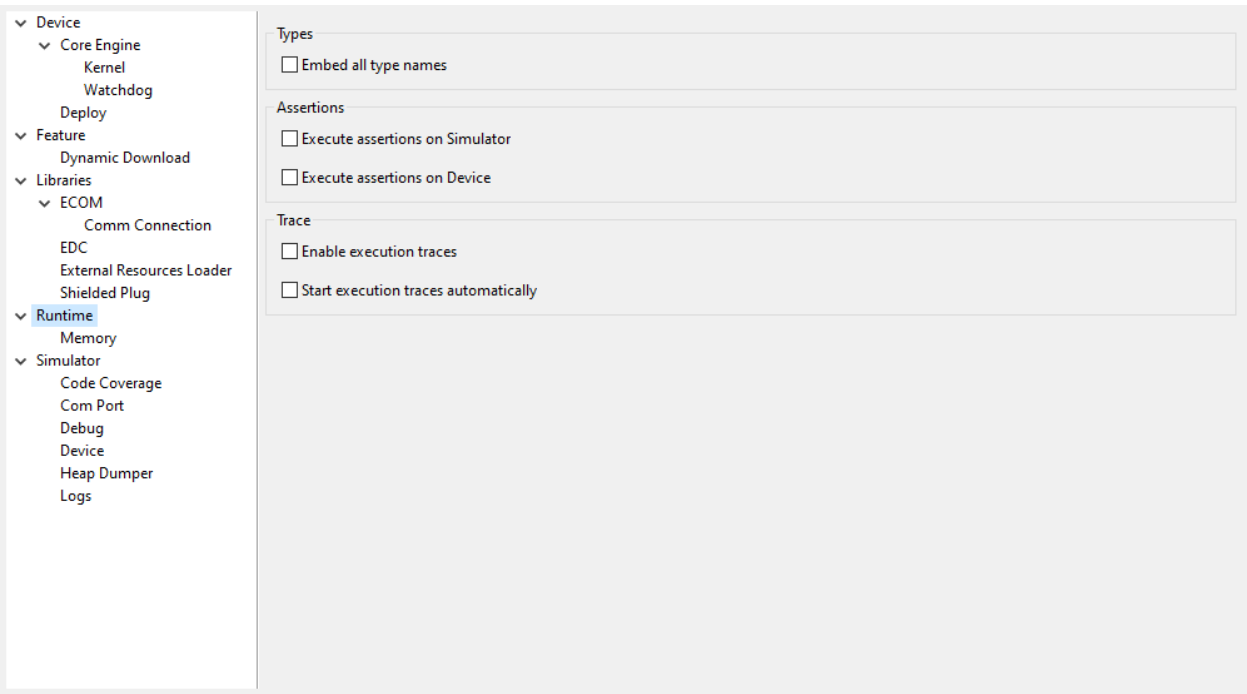
tab, hovering the pointer over an option field will show the location of the properties file that defines the option.

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.

Category: Runtime



Group: Types

Option(checkbox): Embed all type names

Option Name: `soar.generate.classnames`

Default value: `true`

Description:

Embed the name of all types. When this option is disabled, only names of declared required types are embedded.

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`

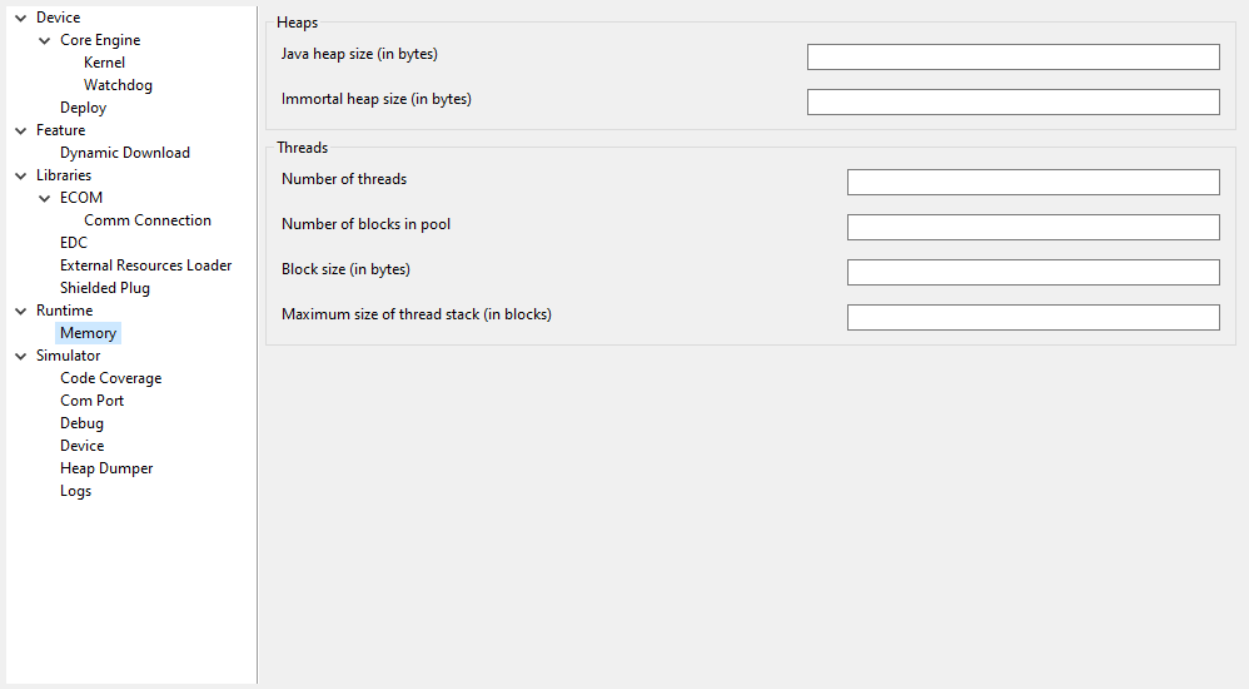
Default value: `false`

Option(checkbox): Start execution traces automatically

Option Name: `core.trace.autostart`

Default value: `false`

Category: Memory



Group: Heaps

Option(text): Java heap size (in bytes)

Option Name: `core.memory.javaheap.size`

Default value: `65536`

Description:

Specifies the Java heap size in bytes.

A Java heap contains live Java objects. An OutOfMemory error can occur if the heap is too small.

Option(text): Immortal heap size (in bytes)

Option Name: `core.memory.immortal.size`

Default value: `4096`

Description:

Specifies the Immortal heap size in bytes.

The Immortal heap contains allocated Immortal objects. An OutOfMemory error can occur if the heap is too small.

Group: Threads

Description:

This group allows the configuration of application and library thread(s). A thread needs a stack to run. This stack is allocated from a pool and this pool contains several blocks. Each block has the same size. At thread startup the thread uses only one block for its stack. When the first block is full it uses another block. The maximum number of blocks per thread must be specified. When the maximum number of blocks for a thread is reached or when there is no free block in the pool, a StackOverflow error is thrown. When a thread terminates all associated blocks are freed. These blocks can then be used by other threads.

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)

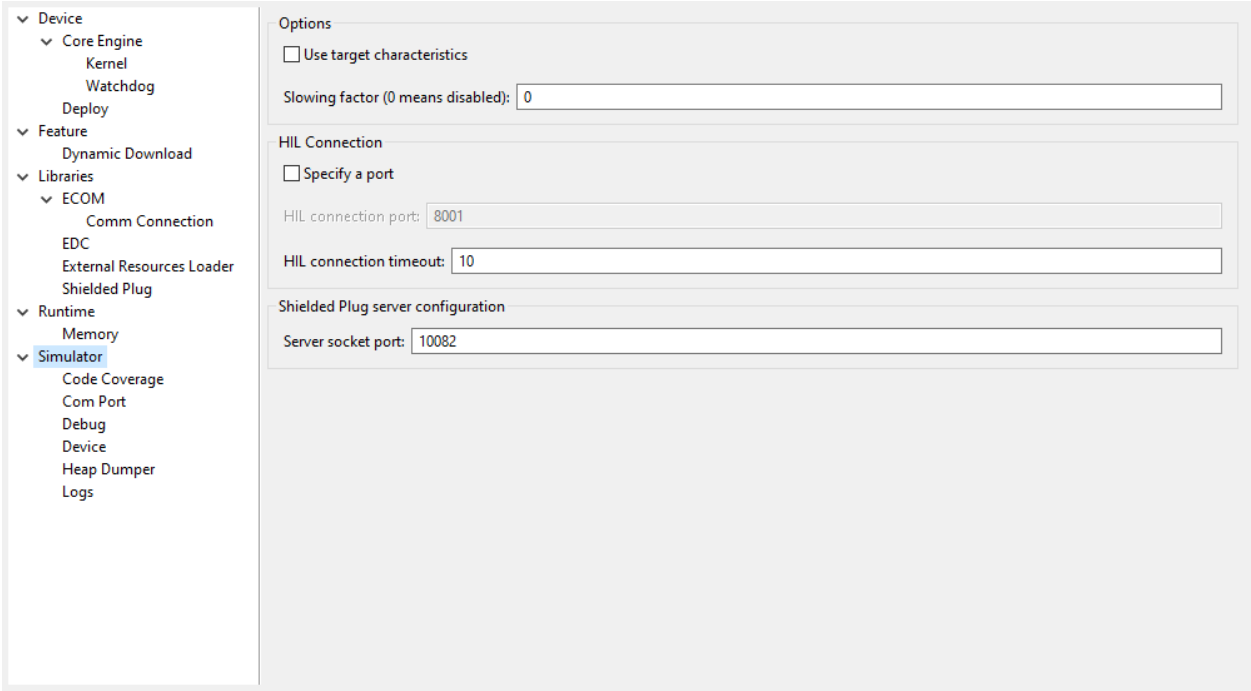
Option Name: `core.memory.thread.max.size`

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

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): HIL connection 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): HIL connection timeout

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.

Group: Shielded Plug server configuration

Description:

This group allows configuration of the Shielded Plug database.

Option(text): Server socket port

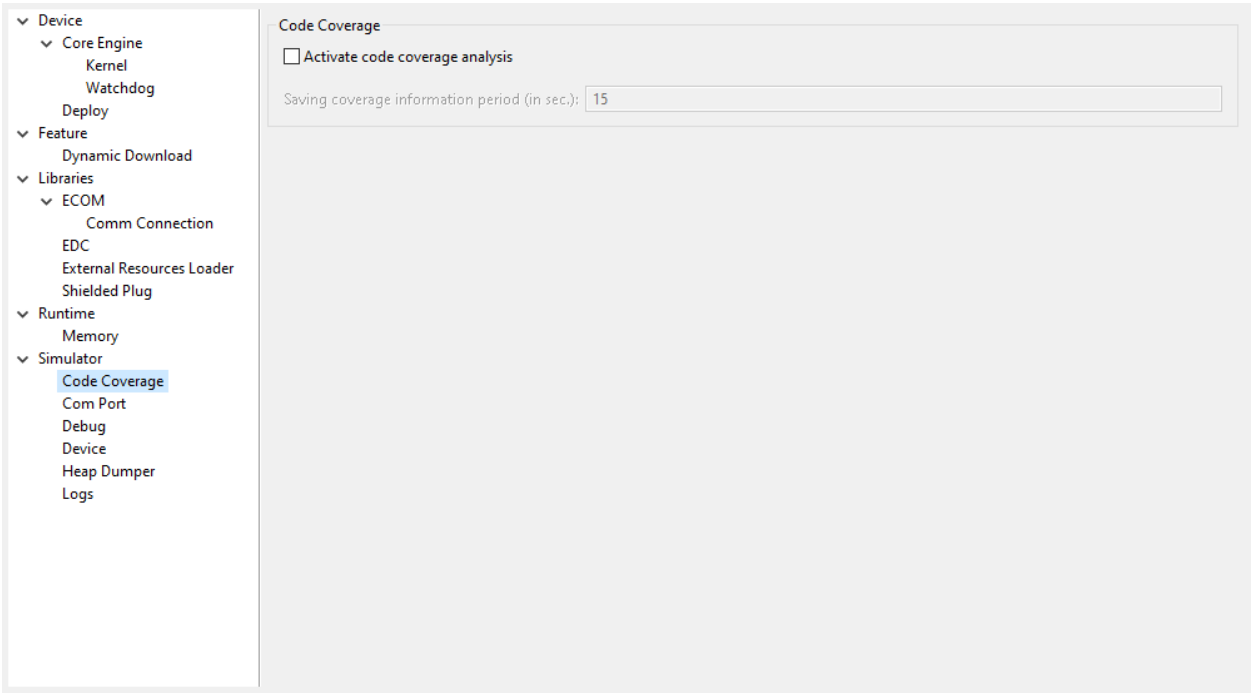
Option Name: `sp.server.port`

Default value: `10082`

Description:

Set the Shielded Plug server socket port.

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.

Option(text): Saving coverage information period (in sec.)

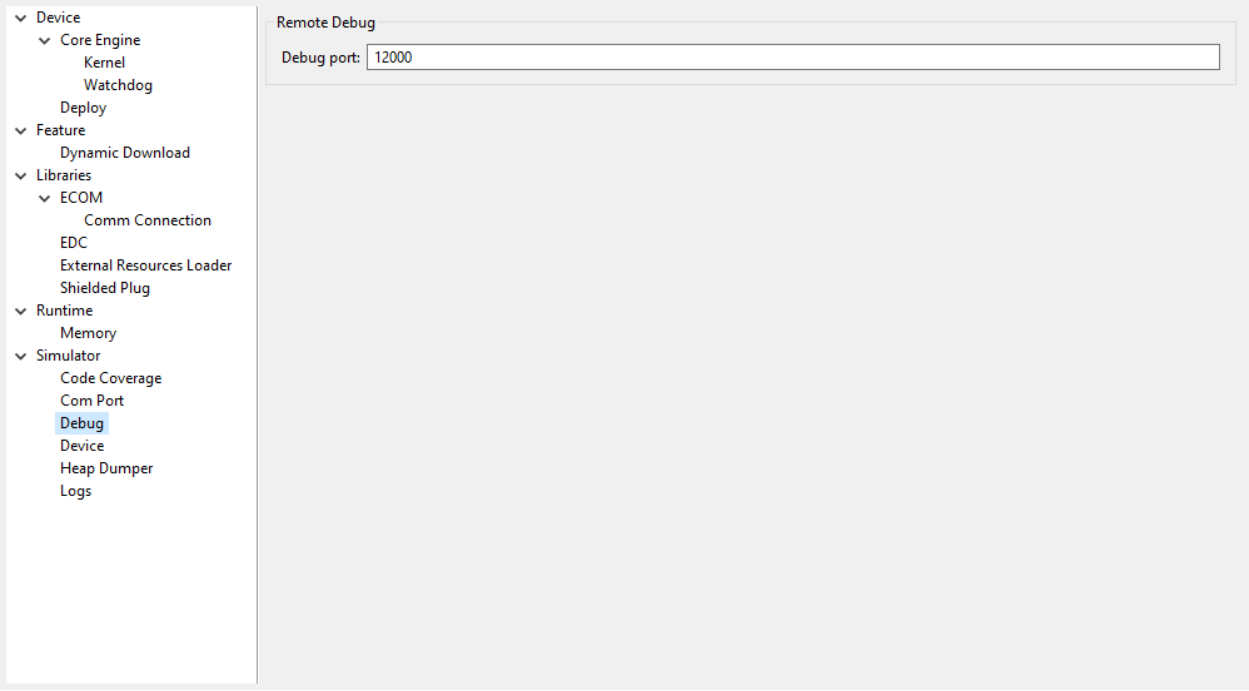
Option Name: `s3.cc.thread.period`

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`

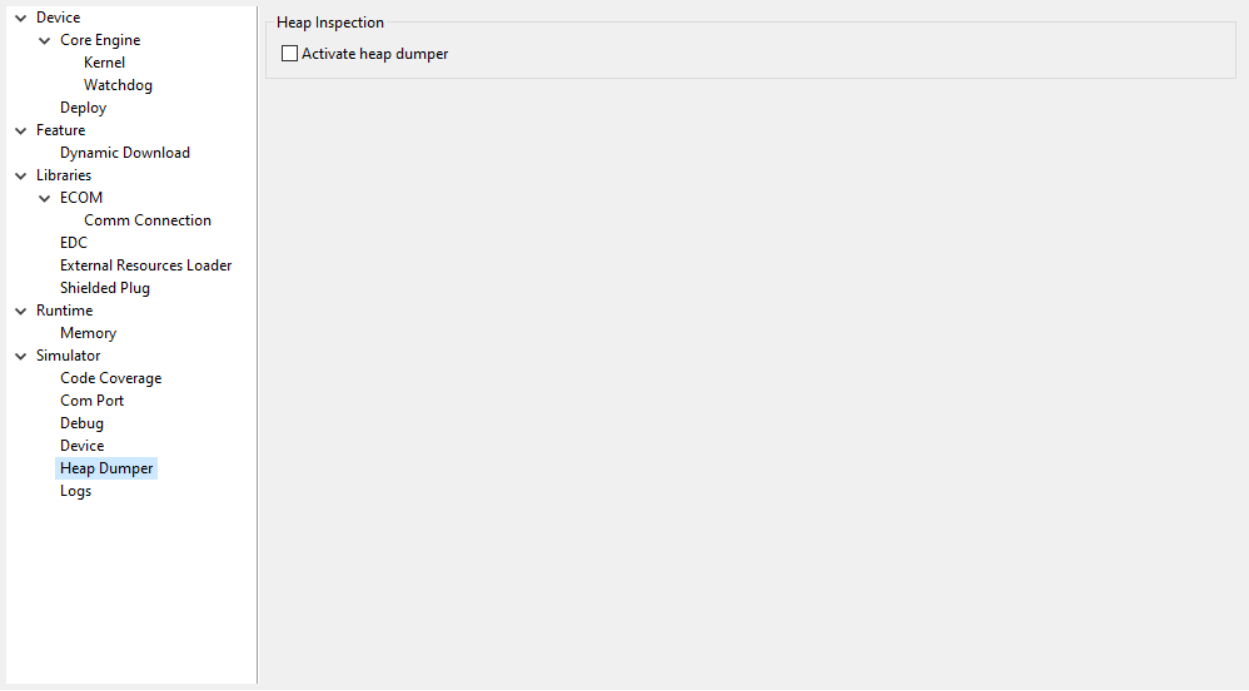
Description:

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

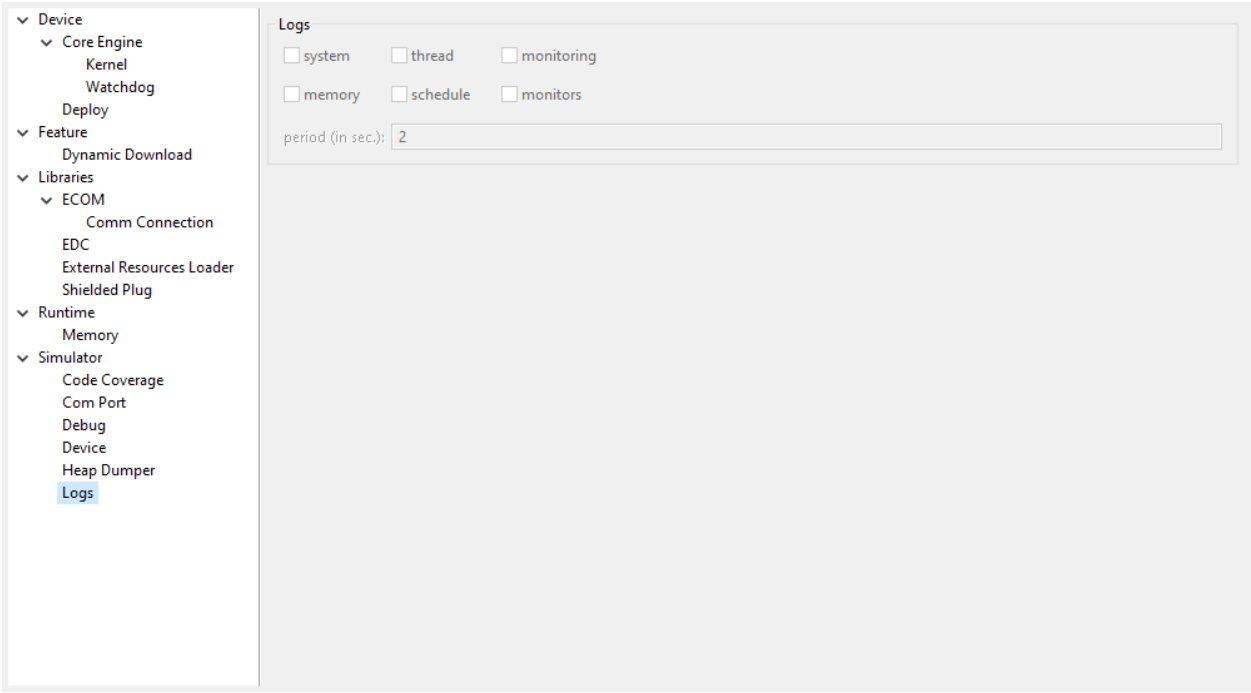
Option Name: `s3.inspect.heap`

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 platform 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 platform 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 platform 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`

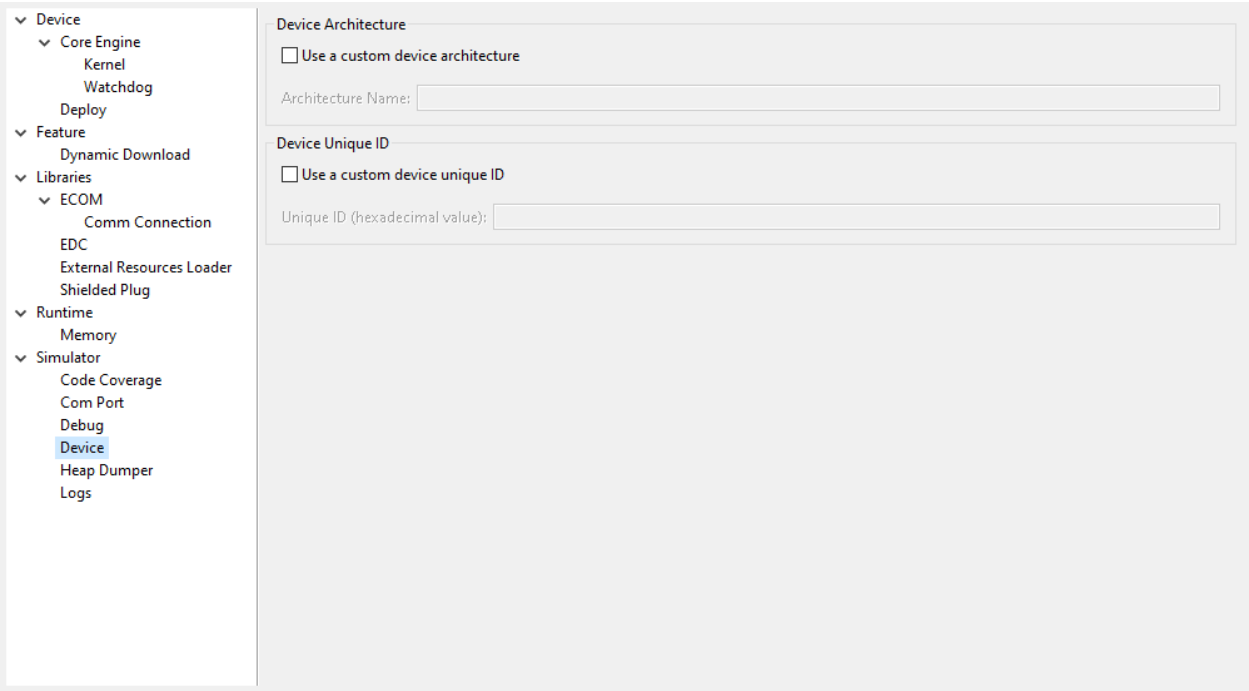
Description:

Format: Positive `integer`

Values: [0-60]

Defines the periodicity of periodical logs.

Category: Device



Group: Device Architecture

Option(checkbox): Use a custom device architecture

Option Name: `s3.mock.device.architecture.option.use`

Default value: `false`

Option(text): Architecture Name

Option Name: `s3.mock.device.architecture.option`

Default value: `(empty)`

Group: Device Unique ID

Option(checkbox): Use a custom device unique ID

Option Name: `s3.mock.device.id.option.use`

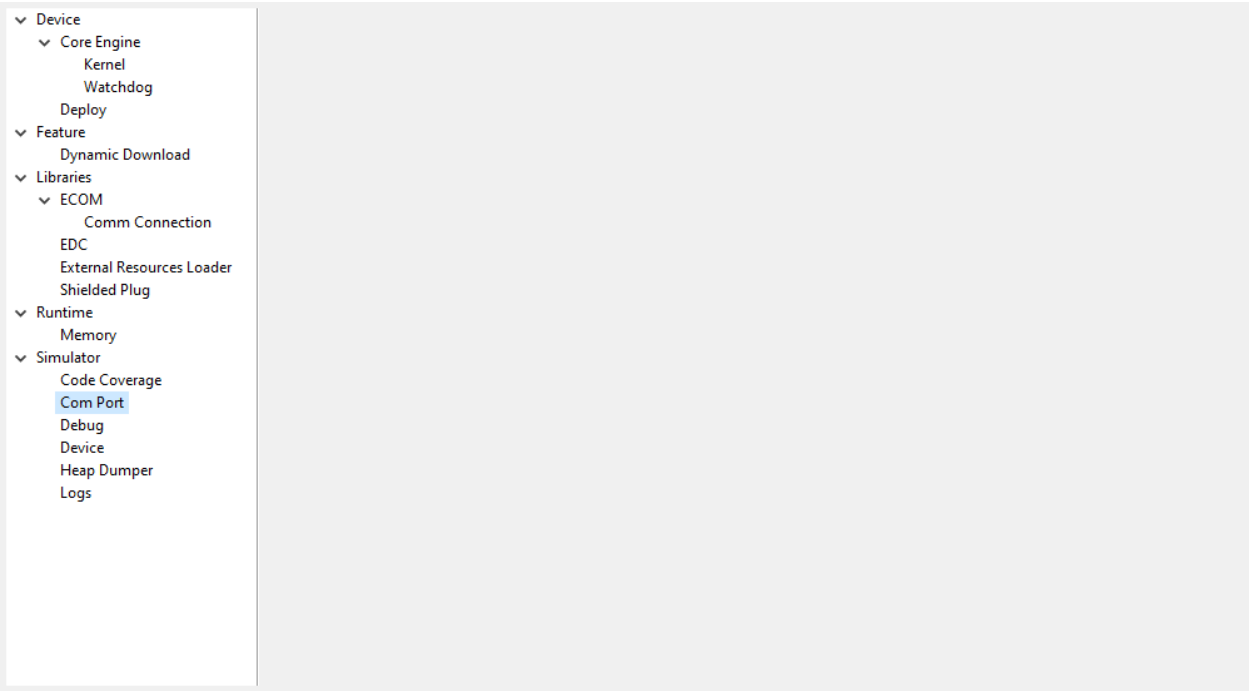
Default value: `false`

Option(text): Unique ID (hexadecimal value)

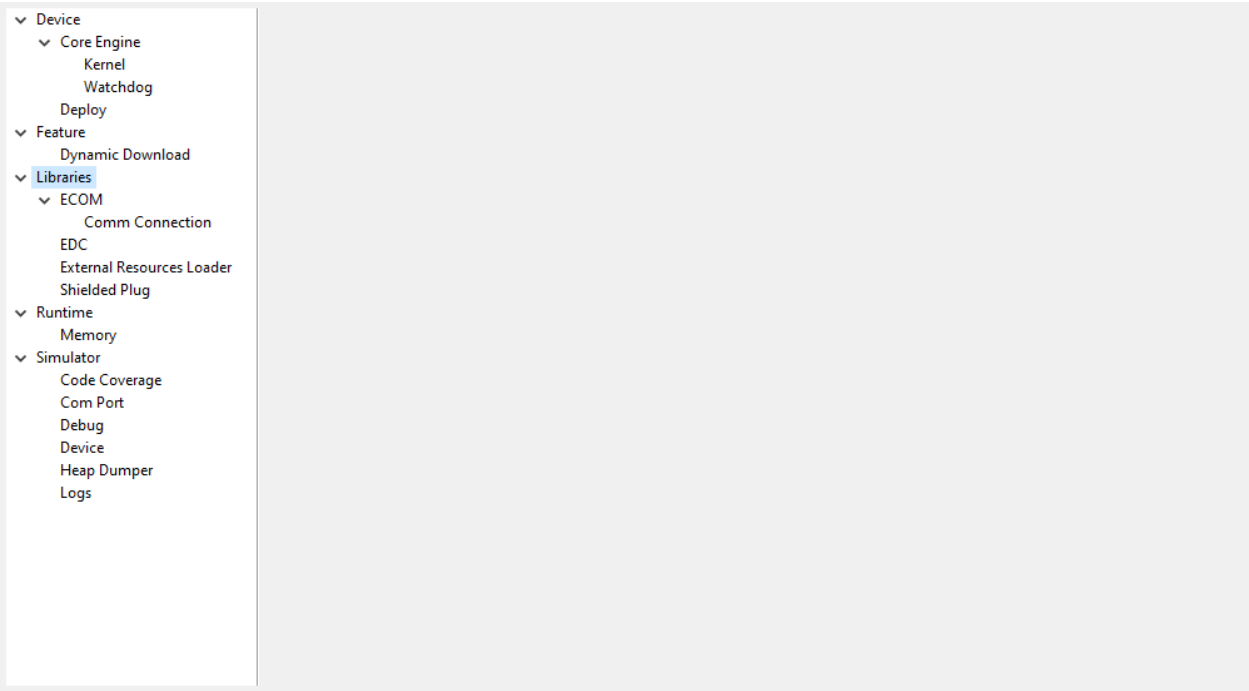
Option Name: `s3.mock.device.id.option`

Default value: `(empty)`

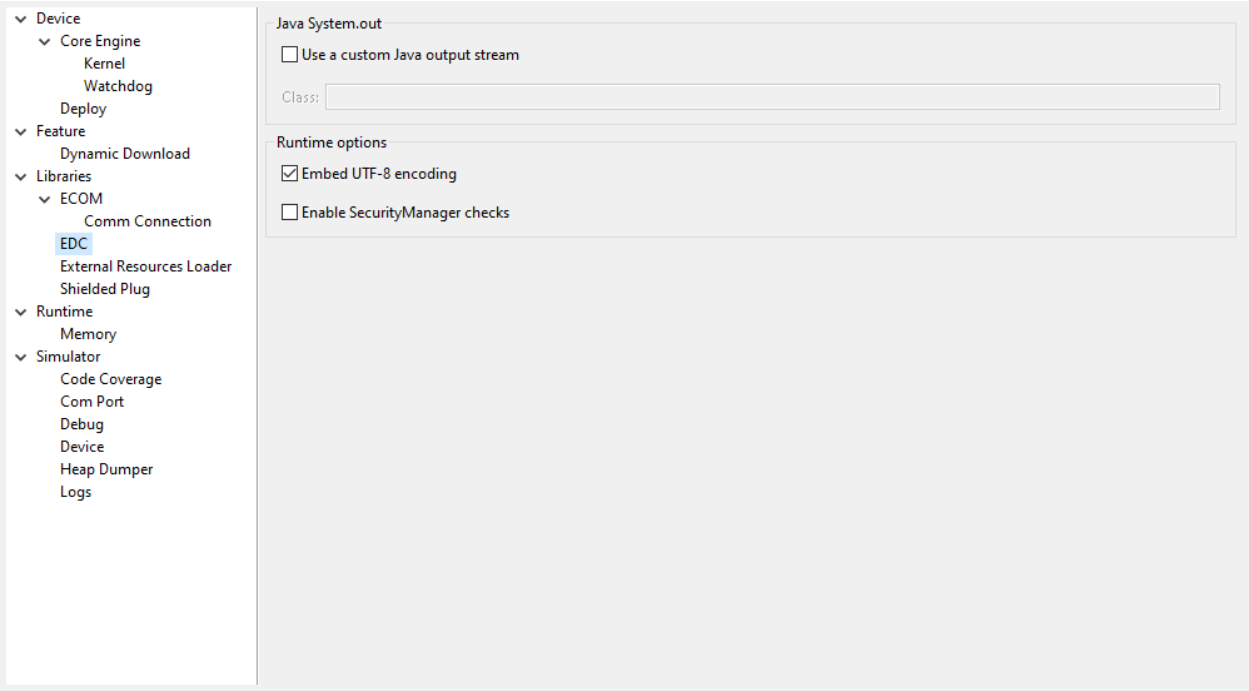
Category: Com Port



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 JPF 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 JPF will try to load this class using the `Class.forName()` method. If the given class is not available, the JPF 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

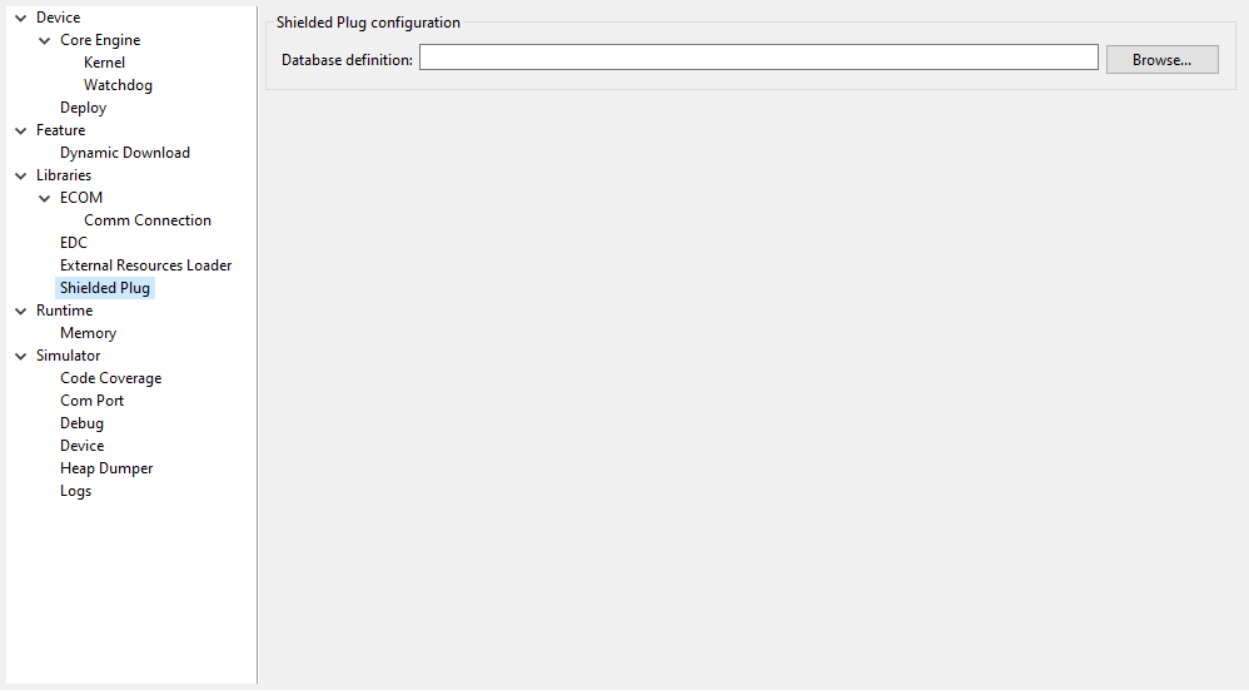
Option Name: `com.microej.library.edc.securitymanager.enabled`

Default value: `false`

Description:

Enable the security manager runtime checks.

Category: Shielded Plug



Group: Shielded Plug configuration

Description:

Choose the database XML definition.

Option(browse): Database definition

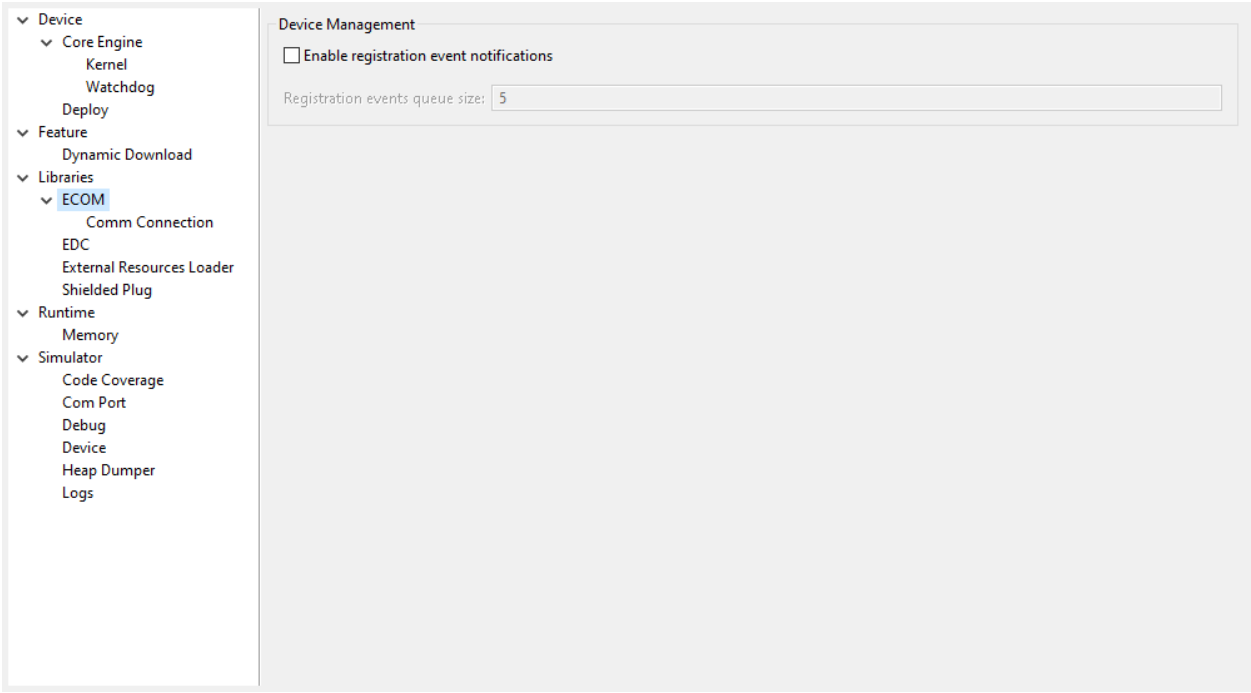
Option Name: `sp.database.definition`

Default value: `(empty)`

Description:

Choose the database XML definition.

Category: ECOM



Group: Device Management

Option(checkbox): Enable registration event notifications

Option Name: `com.is2t.ecom.eventpump.enabled`

Default value: `false`

Description:

Enables notification of listeners when devices are registered or unregistered. When a device is registered or unregistered, a new `ej.ecom.io.RegistrationEvent` is added to an event queue. Then events are processed by a dedicated thread that notifies registered listeners.

Option(text): Registration events queue size

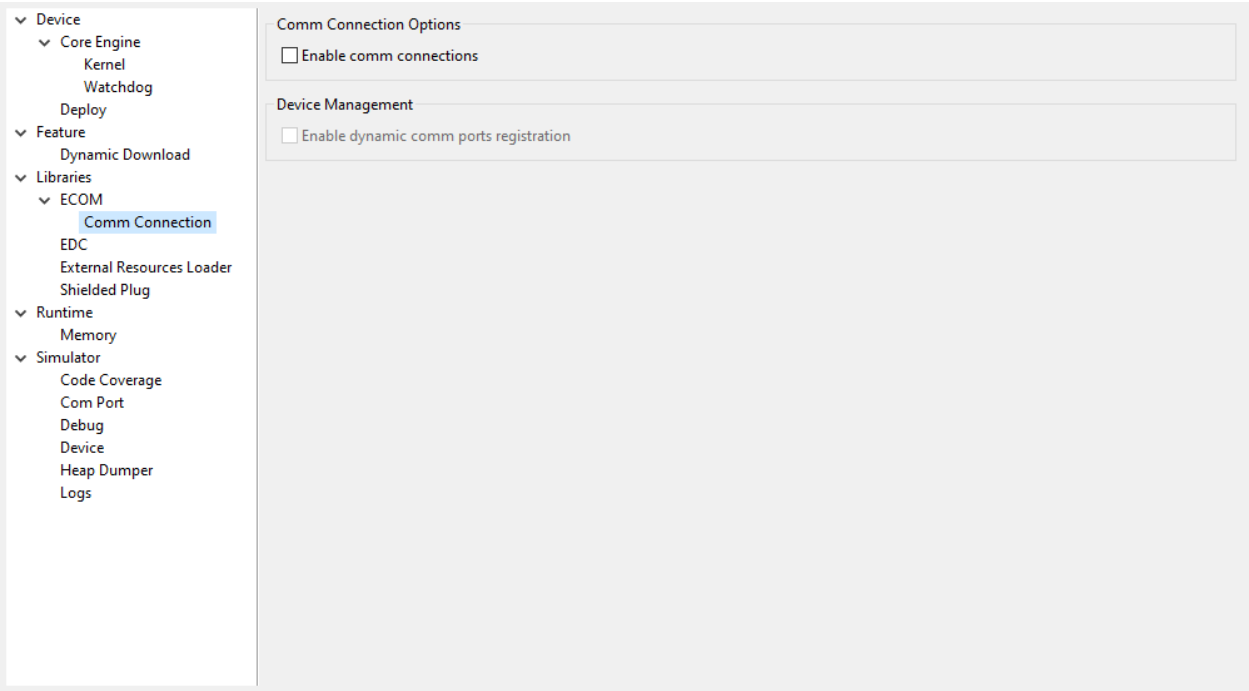
Option Name: `com.is2t.ecom.eventpump.size`

Default value: `5`

Description:

Specifies the size (in number of events) of the registration events queue.

Category: Comm Connection



Group: Comm Connection Options

Description:

This group allows comm connections to be enabled and application-platform mappings set.

Option(checkbox): Enable comm connections

Option Name: `use.comm.connection`

Default value: `false`

Description:

When checked application is able to open a `CommConnection`.

Group: Device Management

Option(checkbox): Enable dynamic comm ports registration

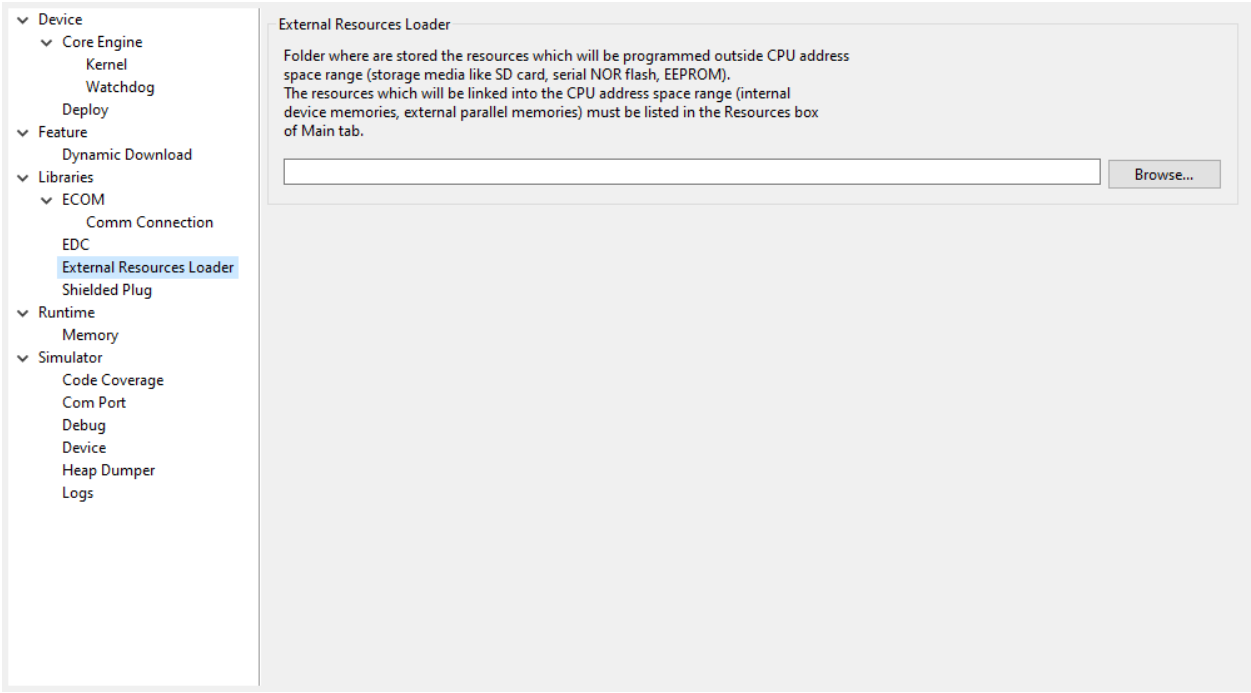
Option Name: `com.is2t.ecom.comm.registryPump.enabled`

Default value: `false`

Description:

Enables registration (or unregistration) of ports dynamically added (or removed) by the platform. A dedicated thread listens for ports dynamically added (or removed) by the platform and adds (or removes) their `CommPort` representation to the ECOM `DeviceManager`.

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):

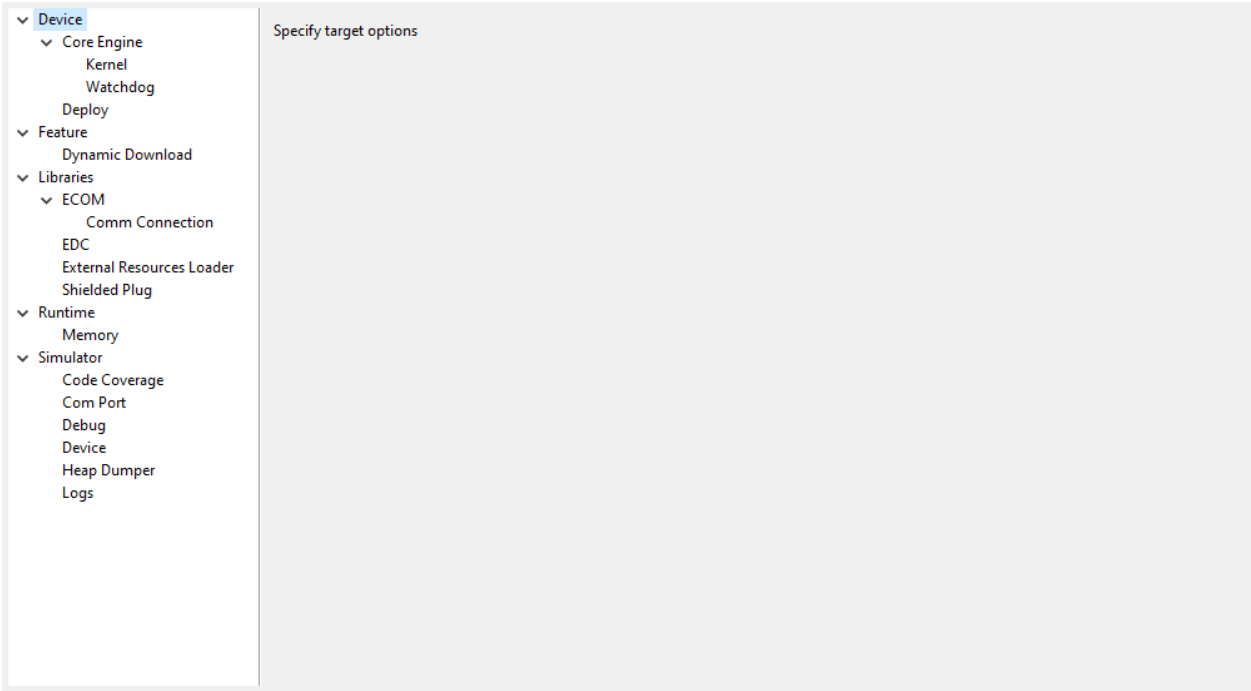
Option Name: `ej.externalResources.input.dir`

Default value: `(empty)`

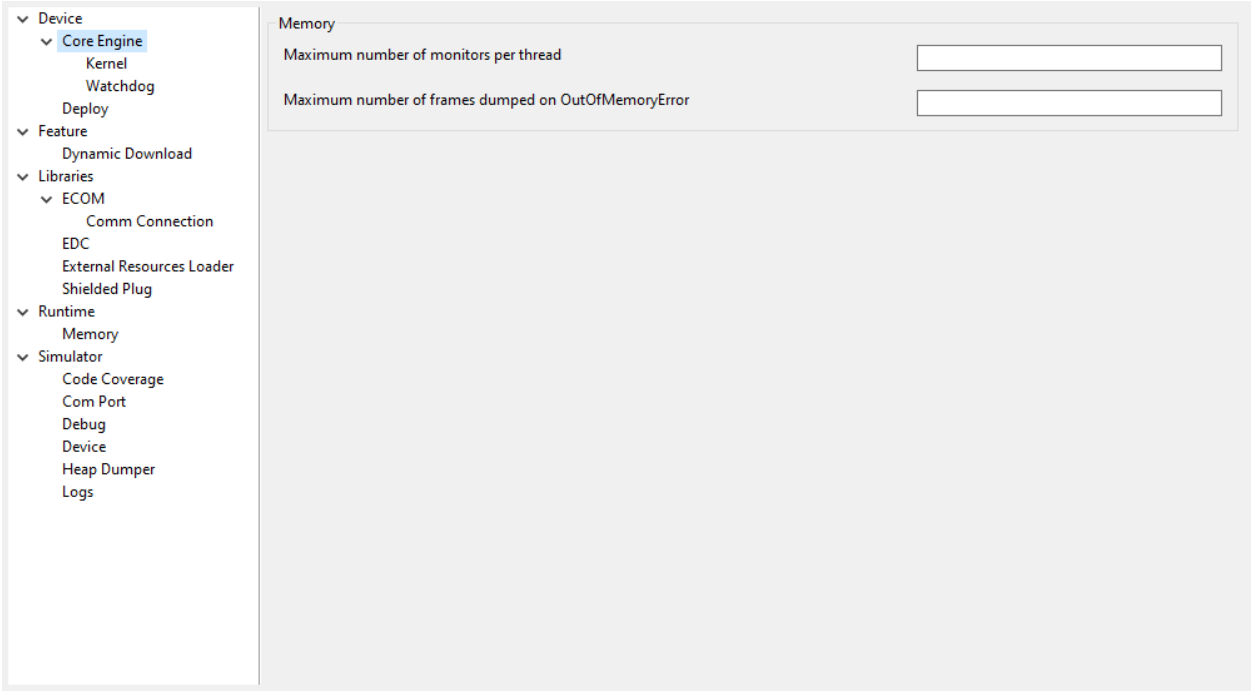
Description:

Browse to specify the external resources folder..

Category: Device



Category: Core Engine



Group: Memory

Option(text):

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):

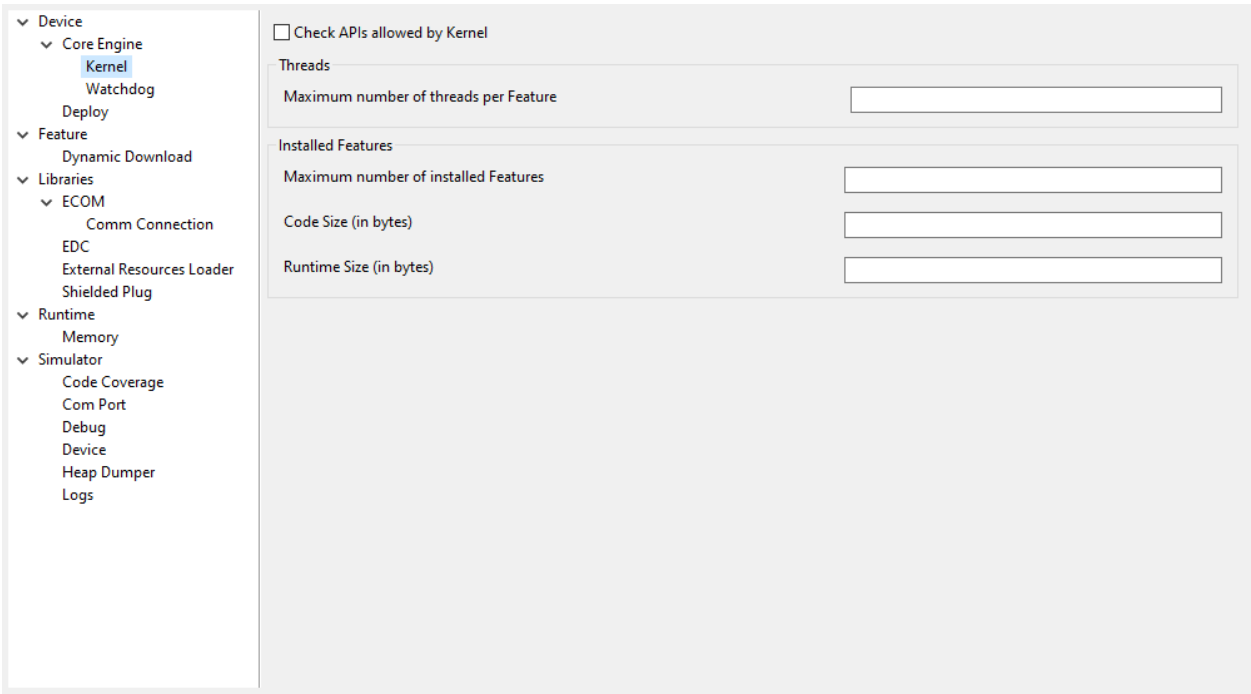
Option Name: `core.memory.oome.nb.frames`

Default value: `5`

Description:

Specifies the maximum number of stack frames that can be dumped to the standard output when Core Engine throws an OutOfMemoryError.

Category: Kernel



Option(checkbox): Check APIs allowed by Kernel

Option Name: `apis.check.enable`

Default value: `true`

Group: Threads

Option(text):

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.

Group: Installed Features

Option(text):

Option Name: `core.memory.installed.features.max`

Default value: `0`

Description:

Specifies the maximum number of installed Features that can be added to this Kernel.

Option(text):

Option Name: `core.memory.installed.features.text.size`

Default value: `0`

Description:

Specifies the size in bytes reserved for installed Features code.

Option(text):

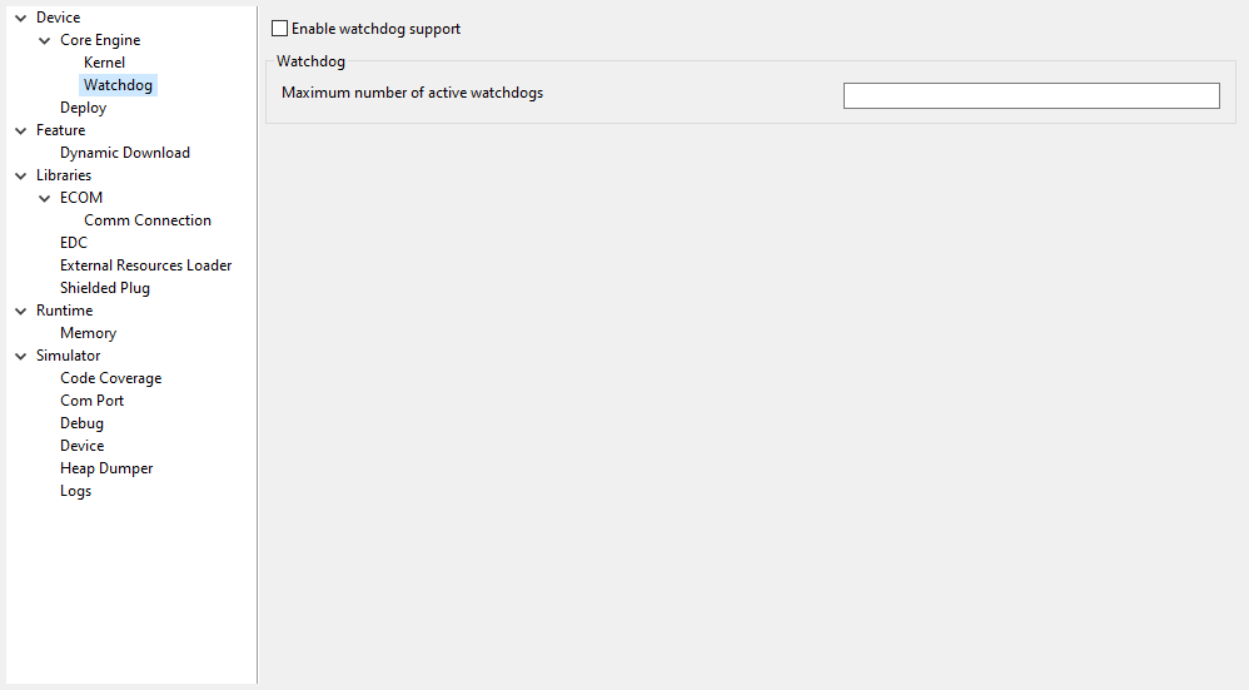
Option Name: `core.memory.installed.features.bss.size`

Default value: `0`

Description:

Specifies the size in bytes reserved for installed Features runtime memory.

Category: Watchdog



Option(checkbox): Enable watchdog support

Option Name: `enable.watchdog.support`

Default value: `true`

Group: Watchdog

Option(text):

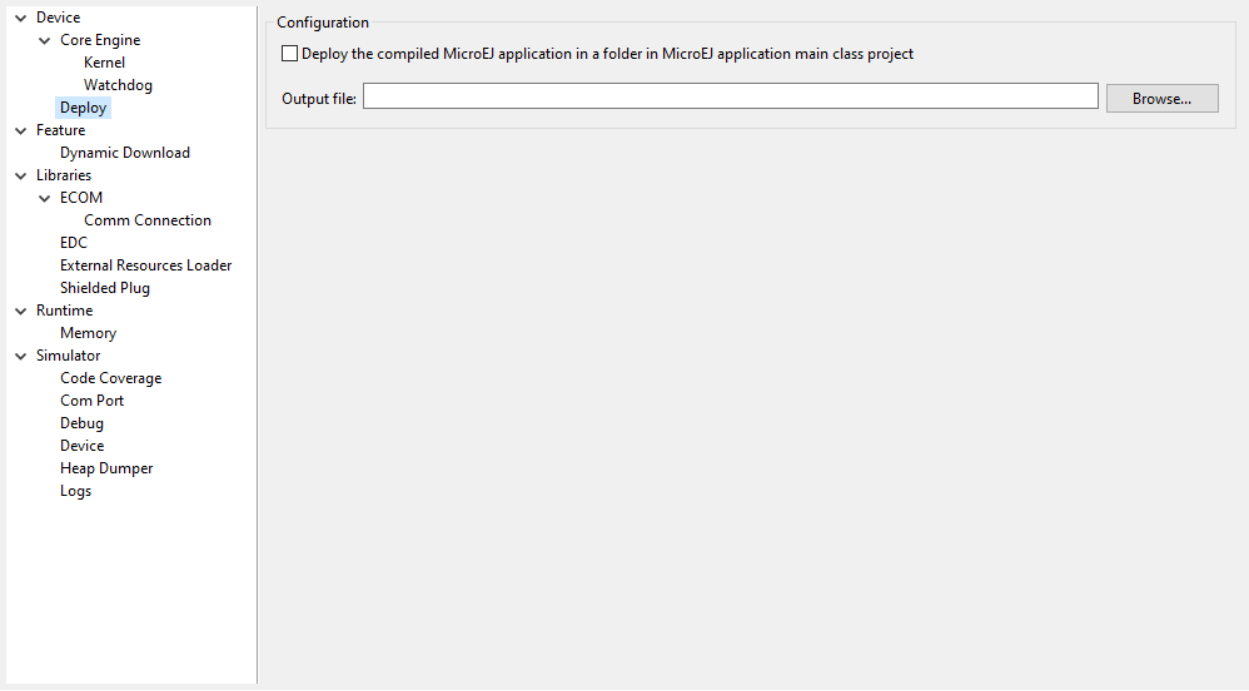
Option Name: `maximum.active.watchdogs`

Default value: `4`

Description:

Specifies the maximum number of active watchdogs at the same time.

Category: Deploy



Description:

Configures the output location where store the MicroEJ Application, the MicroEJ platform libraries and header files.

Group: Configuration

Option(checkbox): Deploy the compiled MicroEJ Application in a folder in MicroEJ Application main class project

Default value: `true`

Description:

Deploy the compiled MicroEJ Application in a folder in MicroEJ Application's main class project.

Option(browse): Output file

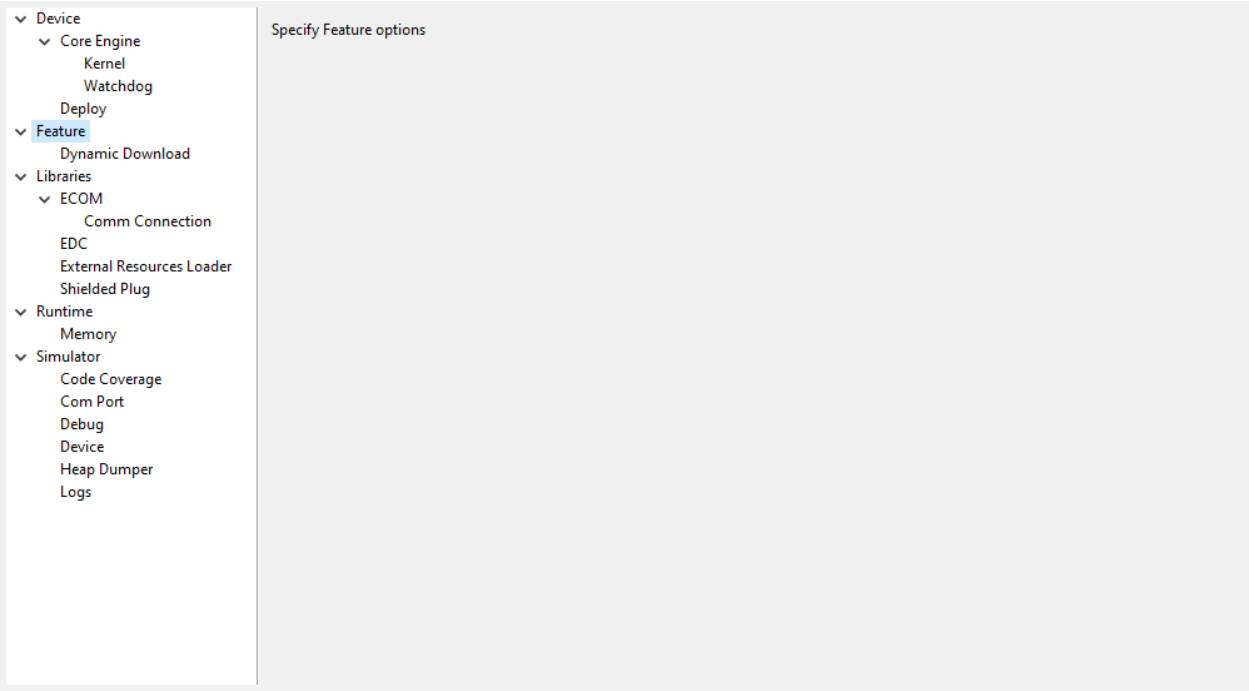
Option Name: `deploy.copy.filename`

Default value: `(empty)`

Description:

Choose an output file location where copy the compiled MicroEJ Application.

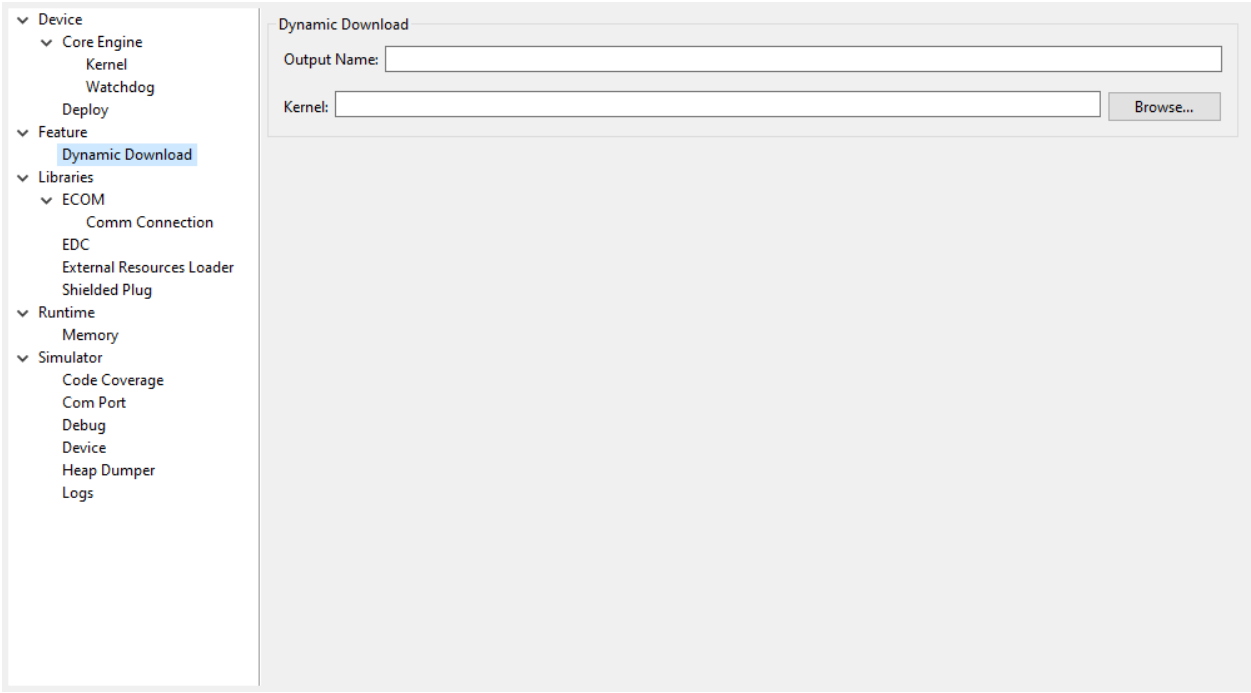
Category: Feature



Description:

Specify Feature options

Category: Dynamic Download



Group: Dynamic Download

Option(text): Output Name

Option Name: `feature.output.basename`

Default value: `application`

Option(browse): Kernel

Option Name: `kernel.filename`

Default value: `(empty)`

3.3.6 SOAR

SOAR complies with the deterministic class initialization (`<clinit>`) order specified in *[BON]*. 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.

An explicit clinit dependency can be declared by creating an XML file with the `.clinitdesc` extension in the application classpath. The file has the following format:

```
<?xml version='1.0' encoding='UTF-8'?>
<clinit>
```

(continues on next page)

(continued from previous page)

```
    <type name="T1" depends="T2"/>
</clinit>
```

where `T1` and `T2` are fully qualified names on the form `a.b.C`. This explicitly forces SOAR to create a dependency from `T1` to `T2`, and therefore cuts a potentially detected dependency from `T2` to `T1`.

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

3.4 Sandboxed Application

3.4.1 Sandboxed Application Structure

Application Skeleton Creation

The first step to explore a Sandboxed Application structure is to create a new project.

First select File > New > MicroEJ Sandboxed Application Project :

Fill in the application template fields, the Project name field will automatically duplicate in the following fields.

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 (images, fonts, etc.);

`META-INF` Sandboxed Application configuration and resources;

`module.ivy` Ivy input file, dependencies description for the current project.

Sources Folder

The project source folder (`src/main`) contains two subfolders: `java` and `resources`. `java` folder will contain all `*.java` files of the project, whereas `resources` folder will contain elements that the application needs at runtime like raw resources, images or character fonts.

META-INF Folder

The `META-INF` folder contains several folders and a manifest file. They are described hereafter.

`certificate` (folder) Contains certificate information used during the application deployment.

`libraries` (folder) Contains a list of additional libraries useful to the application and not resolved through the regular transitive dependency check.

`properties` (folder) Contains an `application.properties` file which contains application specific properties that can be accessed at runtime.

`services` **(folder)** Contains a list of files that describe local services provided by the application. Each file name represents a service class fully qualified name, and each file contains the fully qualified name of the provided service implementation.

`wpk` **(folder)** Contains a set of applications (`.wpk` files) that will be started when the application is executed on the Simulator.

`MANIFEST.MF` **(file)** Contains the information given at project creation, extra information can be added to this file to declare the entry points of the application.

`module.ivy` File

The `module.ivy` file describes all the libraries required by the application at runtime. The Ivy classpath container lists all the modules that have been automatically resolved from the content of `module.ivy` . See *MicroEJ Module Manager* for more informations about MicroEJ Module Manager.

3.4.2 Application Publication

Build the WPK

When the application is ready for deployment, the last step in MicroEJ Studio is to create the WPK (Wadapps PacKage) file that is intended to be published on a MicroEJ Forge instance for end users.

In MicroEJ Studio, right-click on the Sandboxed Application project name and select `Build Module` .

The WPK build process will display messages in MicroEJ 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
```

Publish on a MicroEJ Forge Instance

The WPK file produced by the build process is located in a dedicated `target~/artifacts` folder in the project.

The `.wpk` file is ready to be uploaded to a MicroEJ Forge instance. Please consult https://community.microej.com for more information.

3.4.3 Shared Interfaces

Principle

The Shared Interface mechanism provided by MicroEJ Core Engine is an object communication bus based on plain Java interfaces where method calls are allowed to cross MicroEJ Sandboxed Applications boundaries. The Shared Interface mechanism is the cornerstone for designing reliable Service Oriented Architectures on top of MicroEJ. 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.

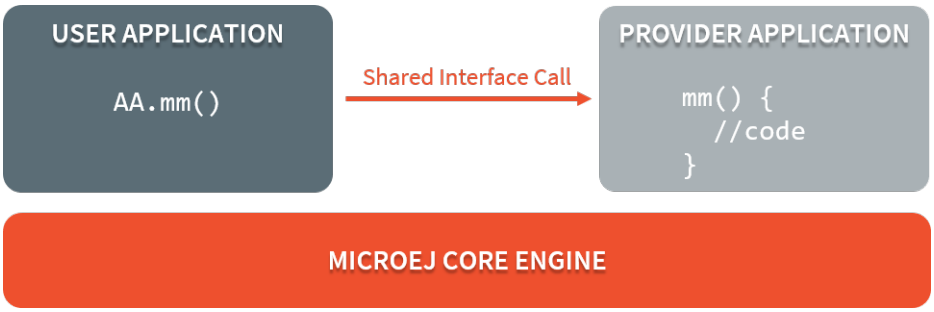


Fig. 15: Shared Interface Call Mechanism

Shared Interface Creation

Creation of a shared interface follows three steps:

- Interface definition,
- Proxy implementation,
- Interface registration.

Interface Definition

The definition of a shared interface starts by defining a standard Java interface.

```
package mypackage;
public interface MyInterface{
    void foo();
}
```

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` suffix with the following format:

```
<sharedInterfaces>
    <sharedInterface name="mypackage.MyInterface"/>
</sharedInterfaces>
```

Shared interface identification files must be placed at the root of a path of the application classpath. For a MicroEJ Sandboxed Application project, it is typically placed in `src/main/resources` folder.

Some restrictions apply to shared interface compared to standard java interfaces:

- Types for parameters and return values must be transferable types;
- Thrown exceptions must be classes owned by the MicroEJ Firmware.

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.

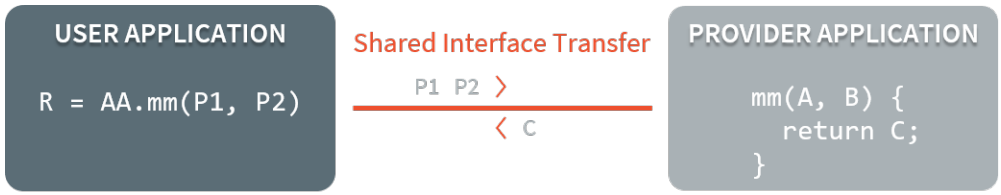


Fig. 16: Shared Interface Parameters Transfer

Shared Interface Types Transfer Rules describes the rules applied depending on the element to be transferred.

Table 1: 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 or forbidden
Array of base types	Any	Application	Clone by copy
Arrays of references	Any	Application	Clone and transfer rules applied again on each element
Shared Interface	Application	Application	Passing by indirect reference (Proxy creation)
Any Class, Array or Interface	Application	Application	Forbidden

Objects created by an application which class is owned by the Kernel can be transferred to another application if this has been authorized by the Kernel. The list of eligible types that can be transferred is Kernel specific, so you have to consult the firmware specification. *MicroEJ Evaluation Firmware Example of Transfer Types* lists Kernel types allowed to be transferred through a shared interface call. When an argument transfer is forbidden, the call is abruptly stopped and a `java.lang.IllegalAccessError` is thrown by MicroEJ Core Engine.

Table 2: MicroEJ Evaluation Firmware Example of Transfer Types

Type	Rule
`java.lang.String`	Clone by copy
`java.io.InputStream`	Proxy reference creation
`java.util.Map<String,String>`	Clone by deep copy

Proxy Class Implementation

The Shared Interface mechanism is based on automatic proxy objects created by the underlying MicroEJ Core Engine, so that each application can still be dynamically stopped and uninstalled. This offers a reliable way for users and providers to handle the relationship in case of a broken link.

Once a Java interface has been declared as Shared Interface, a dedicated implementation is required (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 excep-

tions, application killed…). The MicroEJ Core Engine will allocate instances of this class when an implementation owned by another application is being transferred to this application.

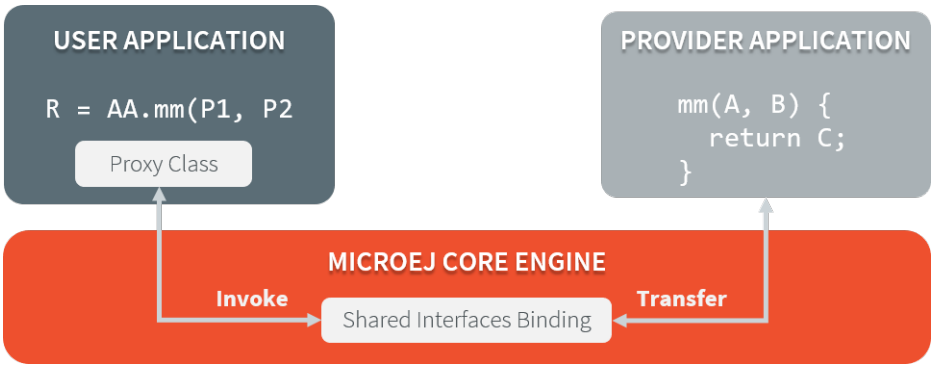


Fig. 17: Shared Interfaces Proxy Overview

A proxy class is implemented and executed on the client side, each method of the implemented interface must be defined according to the following pattern:

```
package mypackage;

public class MyInterfaceProxy extends Proxy<MyInterface> implements MyInterface {

    @Override
    public void foo(){
        try {
            invoke(); // perform remote invocation
        } catch (Throwable e) {
            e.printStackTrace();
        }
    }
}
```

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). Remote invocation methods are defined in the super class `ej.kf.Proxy` and are named `invokeXXX()` where `XXX` is the kind of return type. As this class is part of the application, the application developer has the full control on the Proxy implementation and is free to insert additional code such as logging calls and errors for example.

Table 3: 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

3.5 Virtual Device

3.5.1 Using a Virtual Device for Simulation

The Virtual Device includes the same custom MicroEJ Core, libraries and System 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, MicroEJ Studio ensures that it does not depend on any API that is unavailable on the device.

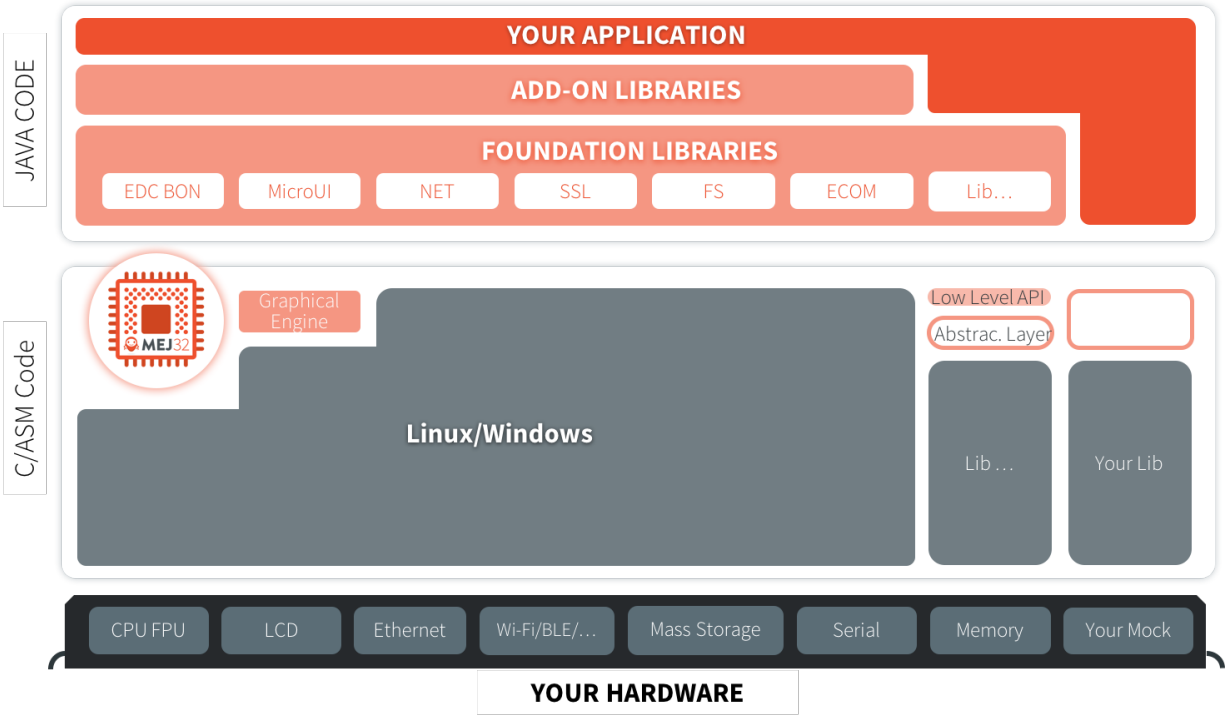


Fig. 18: MicroEJ Virtual Device Architecture

3.5.2 Runtime Environment

The set of MicroEJ APIs exposed by a Virtual Device (and therefore provided by its associated firwmare) is documented in Javadoc format in the MicroEJ Resource Center (Window > Show View > MicroEJ Resource Center).

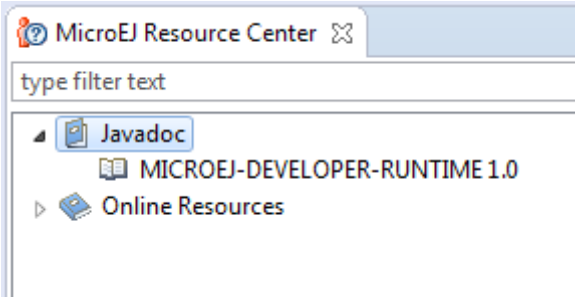


Fig. 19: MicroEJ Resource Center APIs

3.6 MicroEJ Module Manager

3.6.1 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.

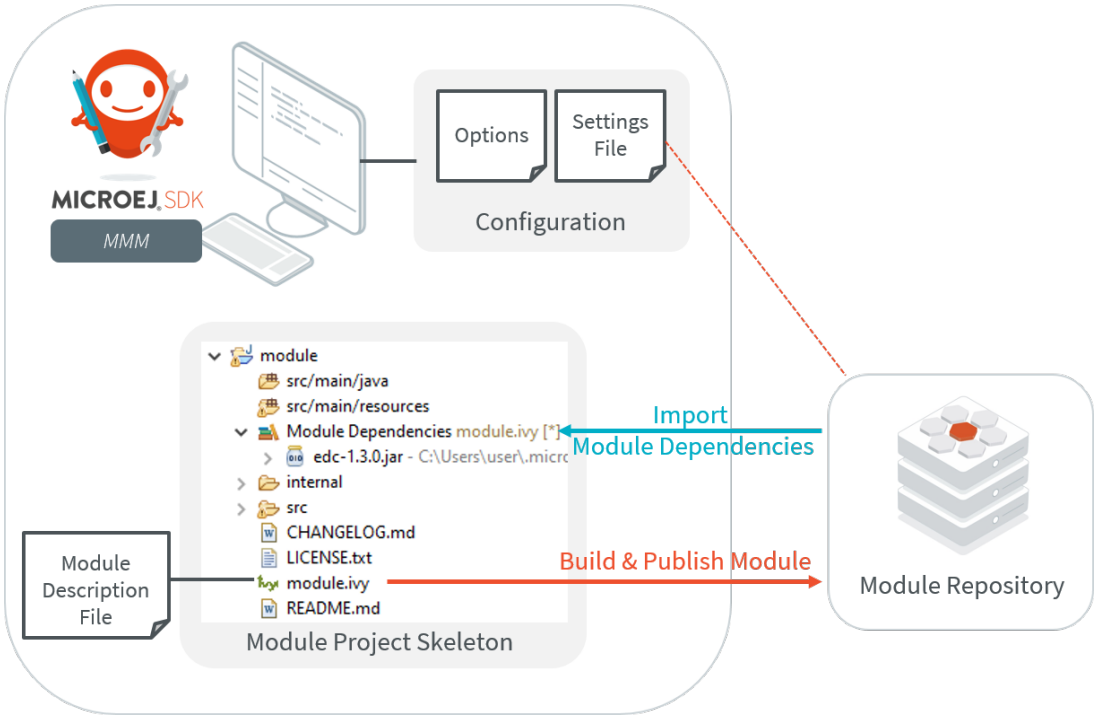


Fig. 20: MMM Overview

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.

3.6.2 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-D.pdf.

3.6.3 Module Project Skeleton

In MicroEJ SDK, a new MicroEJ module project is created as following:

- Select File > New > Project... ,
- Select MicroEJ > 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 .

[1] If using MicroEJ SDK versions lower than `5.2.0`, please refer to the *following section*.

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 following:

- In the *Package Explorer*, click on the parent project or directory,
- Select File > New > Other... ,
- Select EasyAnt > EasyAnt Skeleton .

3.6.4 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"/>
        <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>
```

3.6.5 MicroEJ Module Manager Configuration

By default, when starting an empty workspace, MicroEJ SDK is configured to import dependencies from *MicroEJ Central Repository* and to publish built modules to a local folder. 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 is available at Window > Preferences > MicroEJ > Module Manager [1].

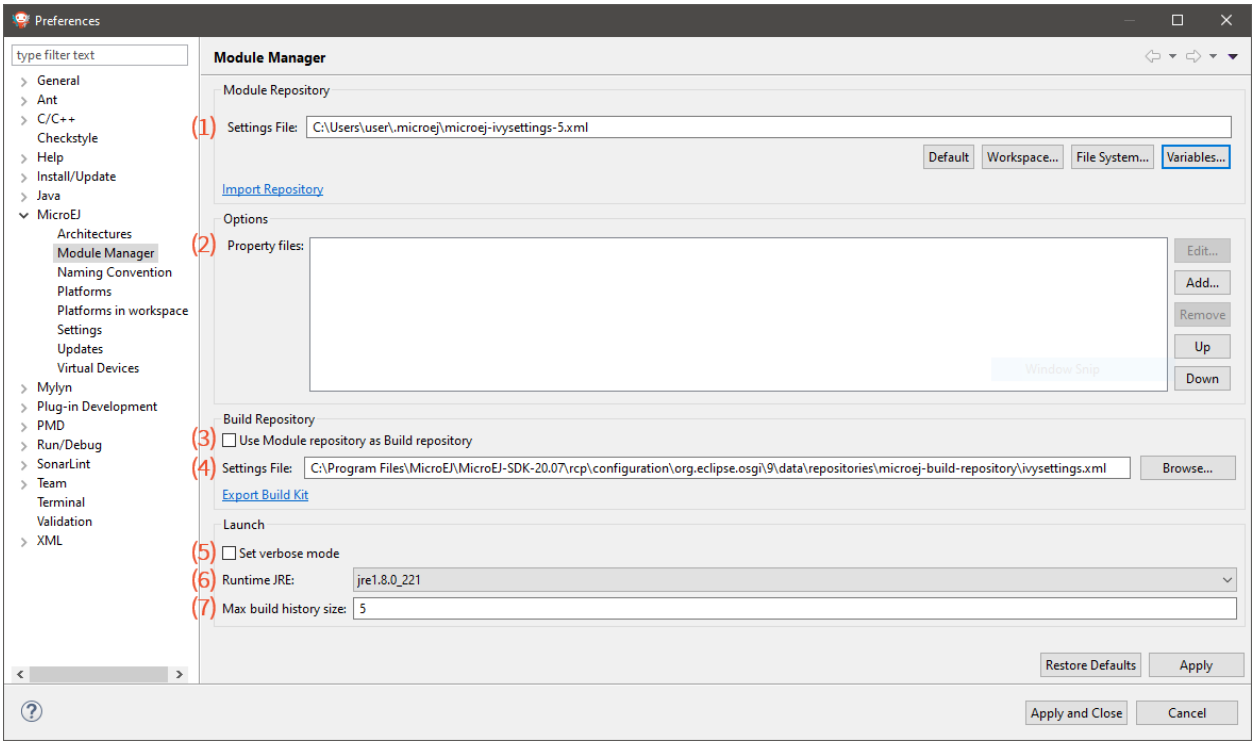
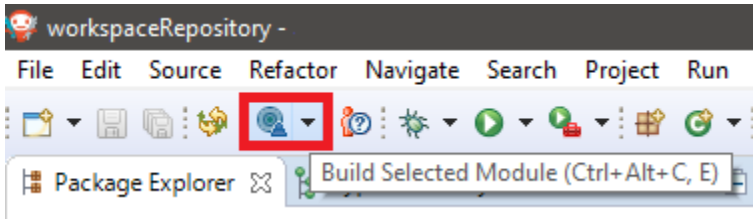


Fig. 21: MMM Preferences Page

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 MicroEJ 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 MicroEJ SDK. This option is automatically initialized the first time MicroEJ 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.

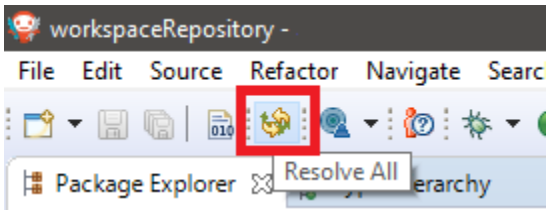
To configure MMM to a custom settings file (usually from an *offline repository*):

1. Set Settings file to a custom `ivysettings.xml` settings file[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

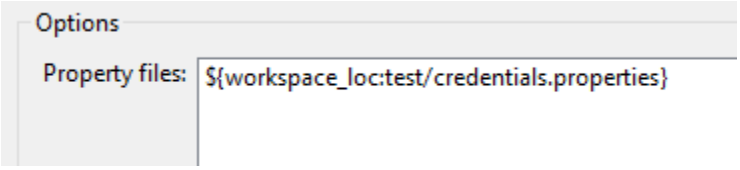
Options can be used to parameterize a *module description file* or a *settings file*. Options are declared as key/value pairs in a standard Java properties file, and are expanded using the `${my_property}` notation.

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
1 # User specific credentials
2 artifactory.username=myusername
3 artifactory.password=AKCKLzp2JHRLDyFvmTPMXocXiiU1Cna47ei9UcC9iE65UdgJrJu24ZTYieX9CwwMa3WYkjCD9
4
```

2. Register this property file to MMM options



3. Use this option in a *settings file*

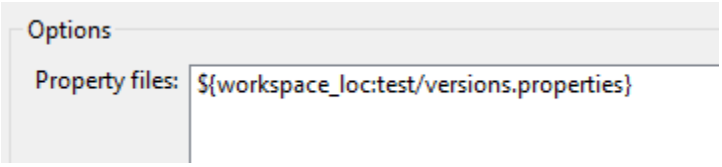
```
38
39  <credentials host="artifactory.corp" realm="Artifactory Realm" username="${artifactory.username}" passwd="${artifactory.password}" />
40
```

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
1 # Specify the EDC version used in this workspace
2 edc.version=1.3.0
3
```

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>
```

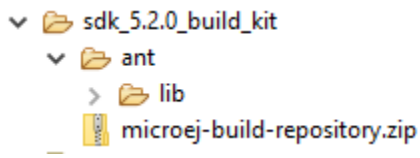
3.6.6 Build Kit

The Module Manager build kit is the consistent set of tools and scripts required for building modules.

It is bundled with MicroEJ SDK and can be exported to run in headless mode using the following steps:[1]

- 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:



To go further with headless builds, please consult Tool-CommandLineBuild for command line builds, and this *tutorial* to setup MicroEJ modules build in continuous integration environments).

3.6.7 Former MicroEJ SDK Versions

This section describes MMM configuration elements for MicroEJ SDK versions lower than `5.2.0`.

New MicroEJ Module Project

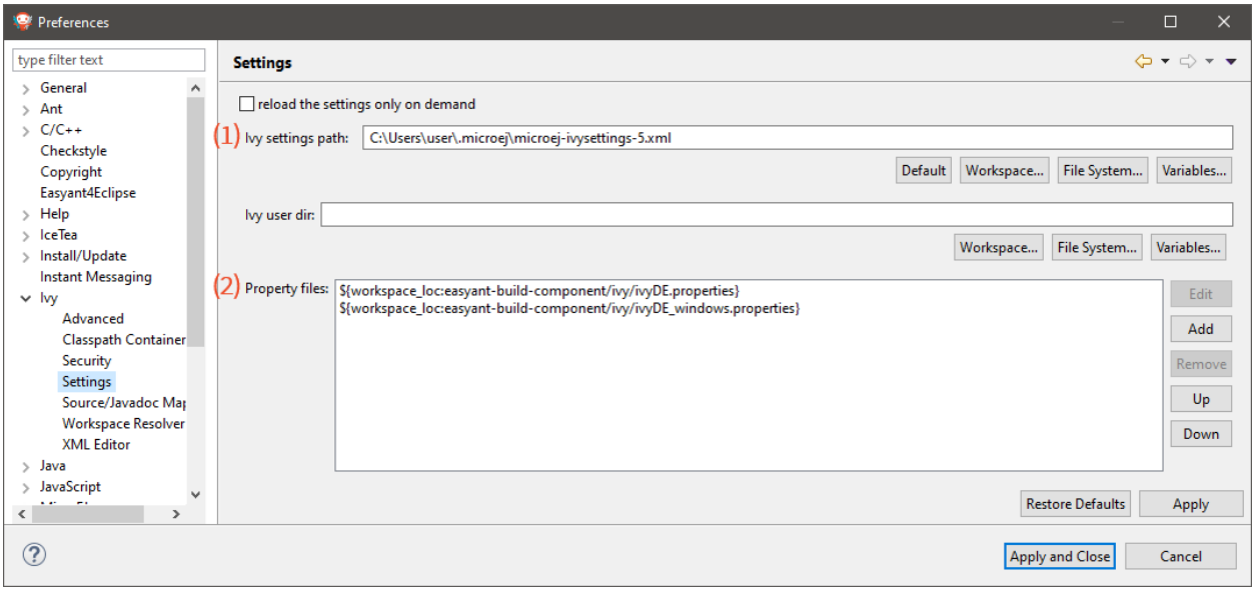
The New MicroEJ Module Project wizard is available at File > New > Project… , EasyAnt > EasyAnt Project .

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*.

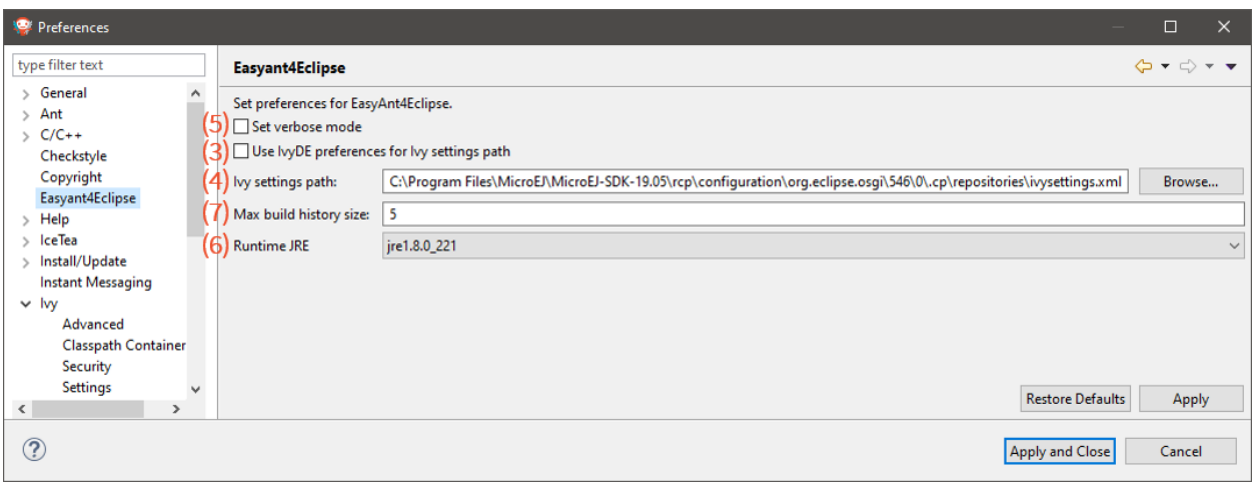
Ivy Preferences Page

The Ivy Preferences Page is available at Window > Preferences > Ivy > Settings .



Easyant Preferences Page

The Easyant Preferences Page is available at Window > Preferences > EasyAnt4Eclipse .



Export the 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 MicroEJ SDK `5.x` or `is2t_repo.zip` for MicroEJ SDK `4.1.x` to the target directory.

3.7 Module Natures

The following table describes the *project skeleton* name for most common MicroEJ Module Natures.

Table 4: MicroEJ Module Natures Summary

Module Nature	Skeleton Name	Direct Wizard
Add-On Library	*microej-javalib*	File > New > MicroEJ Add-On Library Project
Mock	*microej-mock*	
Module Repository	*artifact-repository*	
Sandboxed Application	*application*	File > New > MicroEJ Sandboxed Application Project
Standalone Application	*firmware-singleapp*	File > New > MicroEJ Standalone Application Project

3.7.1 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.

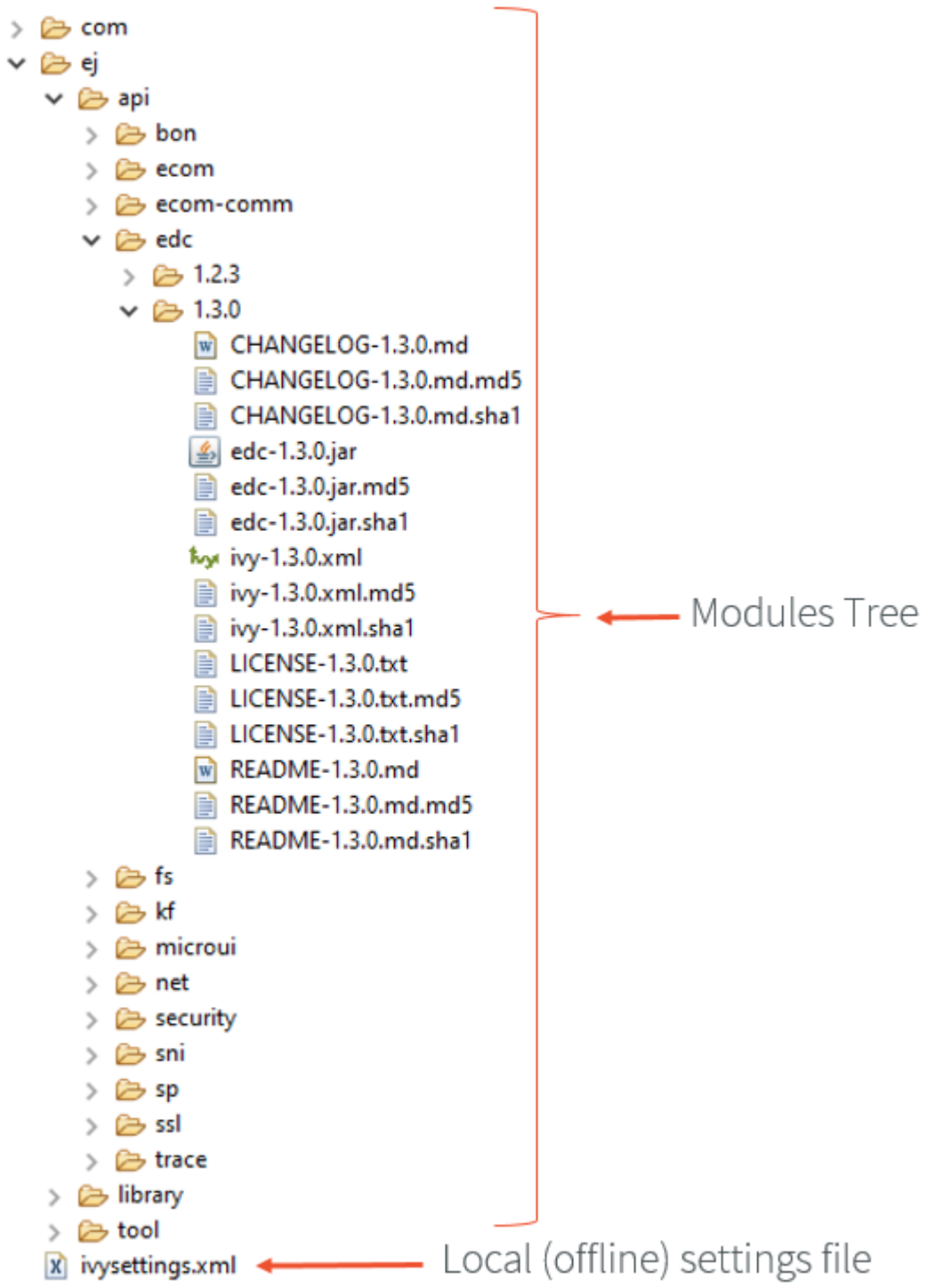


Fig. 22: Example of MicroEJ Module Repository Tree

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 MicroEJ 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.

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"/>
```

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.

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, 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>
```

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:

1. Unzip the module repository `.zip` file to the folder of your choice,
2. *Configure MMM settings file* using the `ivysettings.xml` file located at the root of the folder where the repository has been extracted.

3.8 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 MicroEJ 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.

3.8.1 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. 23: MicroEJ Application Classpath Mapping

3.8.2 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.

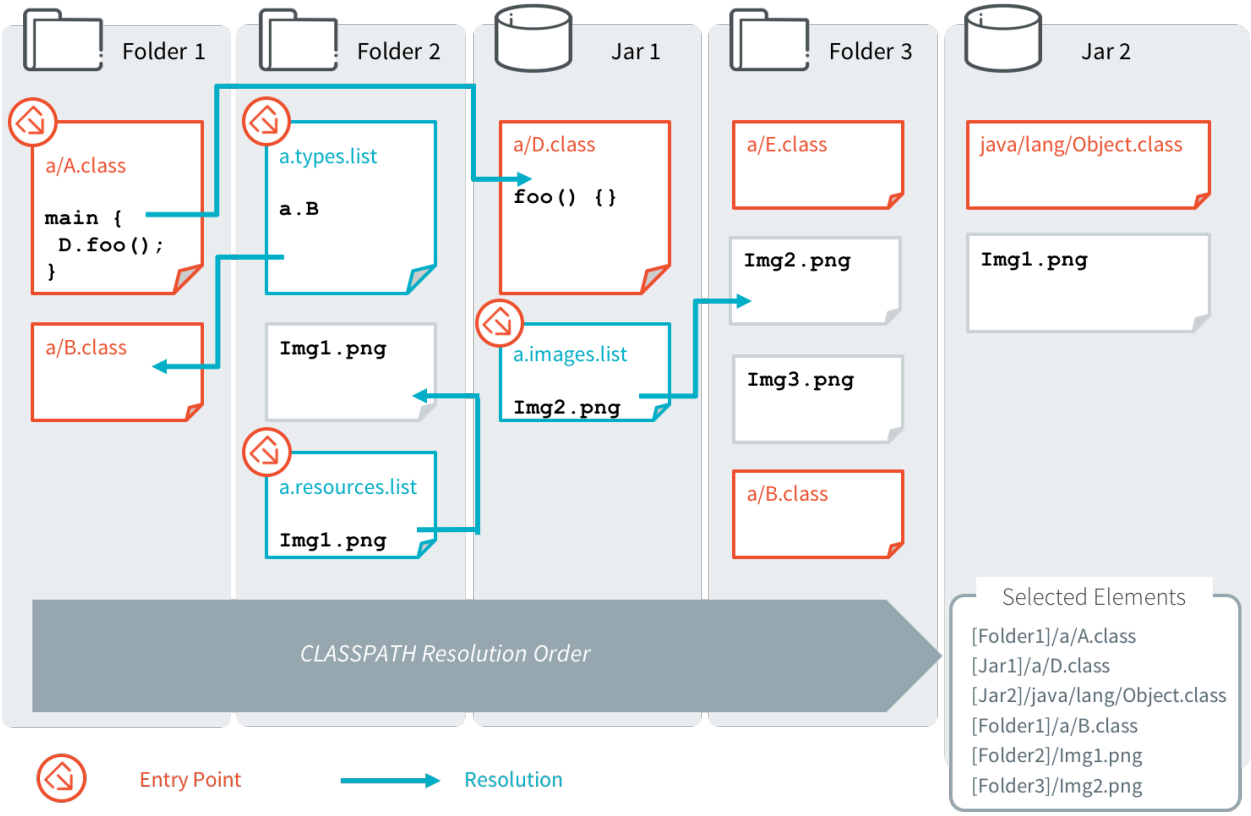


Fig. 24: Classpath Load Principle

3.8.3 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*;
- Raw resources, described in section *Raw Resources*;
- Immutables Object data files, described in Section *Immutable Objects*;
- Images, Fonts and Native Language Support (NLS) resources, 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 MicroEJ 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 MicroEJ Sandboxed Application, it is a class that implements `ej.kf.FeatureEntryPoint`, declared in the `Application-EntryPoint` entry in `META-INF/MANIFEST.MF` file.

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()`).

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
java.util.Vector
```

Raw Resources

Raw resources are binary files that need to be embedded by the application so that they may be dynamically retrieved with a call to `Class.getResourceAsStream(java.io.InputStream)`. Raw Resources are declared in MicroEJ Classpath using `*.resources.list` files. 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
```

Others resources types are supported in MicroEJ Classpath, see *Application Resources* for more details.

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 declared in MicroEJ Classpath `*.properties.list` files. The file format is a standard Java properties file. Example:

Listing 1: 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 Applications Options. This can be done by setting the option with a specific prefix in their name:

- Properties for both the MicroEJ Core Engine and the MicroEJ Simulator : name starts with `microej.java.property.*`
- Properties for the MicroEJ Simulator : name starts with `sim.java.property.*`
- Properties for the MicroEJ Core Engine : name starts with `emb.java.property.*`

For example, to define the property `myProp` with the value `theValue`, set the following option :

Listing 2: Example of MicroEJ Property Definition in Launch Configuration

```
microej.java.property.myProp=theValue
```

Option can also be set in the `VM arguments` field of the `JRE` tab of the launch using the -D option (e.g. `-Dmicroej.java.property.myProp=theValue`).

Constants

Note: This feature require *[BON]* version `1.4` which is available in MicroEJ Runtime starting from MicroEJ Architecture version `7.11.0`.

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 3: Example of Contents of a BON constants File

```
# The following property is embedded as a constant
com.mycompany.myconstantkey=com.mycompany.myconstantvalue
```

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 `"com.mycompany.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 4: Example of `if` code removal using a BON boolean constant

```
if (Constants.getBoolean("com.mycompany.myconstantkey")) {
       System.out.println("this code and the constant string will be fully removed when the 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*.

3.9 Application Resources

Application resources are the following *Classpath Elements*:

- *Images*
- *Fonts*
- *Native Language Support*

3.9.1 Images

Overview

Images are graphical resources that can be accessed with a call to `ej.microui.display.Image.createImage()`. To be displayed, these images have to be converted from their source format to the display raw format. The conversion can either be done at :

- build-time (using the image generator tool),
- run-time (using the relevant decoder library).

Images that must be processed by the image generator tool are declared in MicroEJ Classpath `*.images.list` files. 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 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 (although listing files without format specifier has no impact on the image generator processing, it is advised to specify them in the `*.images.list` files anyway, as it makes the run-time processing behavior explicit). 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 bits format without transparency (decoded at build-time)
com/mycompany/MyImage2.png:RGB565

# The following image is embedded
# as a 16 bits format with transparency (decoded at build-time)
com/mycompany/MyImage3.png:ARGB1555
```

3.9.2 Fonts

Overview

Fonts are graphical resources that can be accessed with a call to `ej.microui.display.Font.getFont()`. To be displayed, these fonts have to be converted at build-time from their source format to the display raw format by the font generator tool. Fonts that must be processed by the font generator tool are declared in MicroEJ Classpath `*.fonts.list` files. 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 MicroEJ font file (usually with a `.ejf` file 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.ejf

# The following font is embedded with only the latin
# unicode range without transparency
com/mycompany/MyFont2.ejf:latin

# The following font is embedded with all characters
# with 2 levels of transparency
com/mycompany/MyFont2.ejf::2
```

MicroEJ font files conventionally end with the `.ejf` suffix and are created using the Font Designer (see *Font Designer*).

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. 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` : Embed all characters from 0x21 to 0x49 (included);
- `myfont:0x21-0x49,0x55` : Embed all characters from 0x21 to 0x49 and character 0x55;
- `myfont:0x21-0x49;0x55` : Same as previous, but done by declaring two ranges.

Known Range

A known range is a range defined by the "Unicode Character Database" version 9.0.0 available on https://home.unicode.org/. Each range is composed of sub ranges that have a unique id.

- `myfont:basic_latin` : Embed all *Basic Latin* characters;
- `myfont:basic_latin;arabic` : Embed all *Basic Latin* characters, and all *Arabic* characters.

Transparency

The second parameter is for specifying the font transparency level (`1` , `2` , `4` or `8`).

Examples:

- `myfont:latin:4` : Embed all latin characters with 4 levels of transparency
- `myfont::2` : Embed all characters with 2 levels of transparency

3.9.3 Native Language Support

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.

Messages must be defined in PO files in the MicroEJ Classpath of the application. Here is an example:

```
msgid ""
msgstr ""
"Language: en_US\n"
"Language-Team: English\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=UTF-8\n"

msgid "Label1"
msgstr "My label 1"

msgid "Label2"
msgstr "My label 2"
```

These PO files have to be converted to be usable by the application. In order to let the build system know which PO files to process, they must be referenced in MicroEJ Classpath `*.nls.list` files. The file format of these `*.nls.list` files 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. Here is an example, let's call it *i18n.nls.list*:

```
com.mycompany.myapp.Labels
com.mycompany.myapp.Messages
```

For each line, PO files whose name starts with the interface name (`Messages` and `Labels` in the example) are retrieved from the MicroEJ Classpath and used to generate:

- a Java interface with the given FQN, containing a field for each `msgid` of the PO 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*.po` and located in the MicroEJ Classpath. PO files are generally suffixed by their locale (`Labels_en_US.po`) but it is only for convenience since the suffix is not used, the locale is extracted from the PO file's metadata.

Once the generation is done, the application can use the Java interfaces to get internationalized messages, for example:

```
import com.mycompany.myapp.Labels;

public class MyClass {

   String label = Labels.Label1;

   ...
```

The generation is triggered when building the application or after a change done in any PO or `*.nls.list` files. This allows to always have the Java interfaces up-to-date with the translations and to use them immediately.

The NLS API module must be added to the *module.ivy* of the MicroEJ Application project, in order to allow access to the NLS library.

```
<dependency org="ej.library.runtime" name="nls" rev="3.0.1"/>
```

3.10 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.

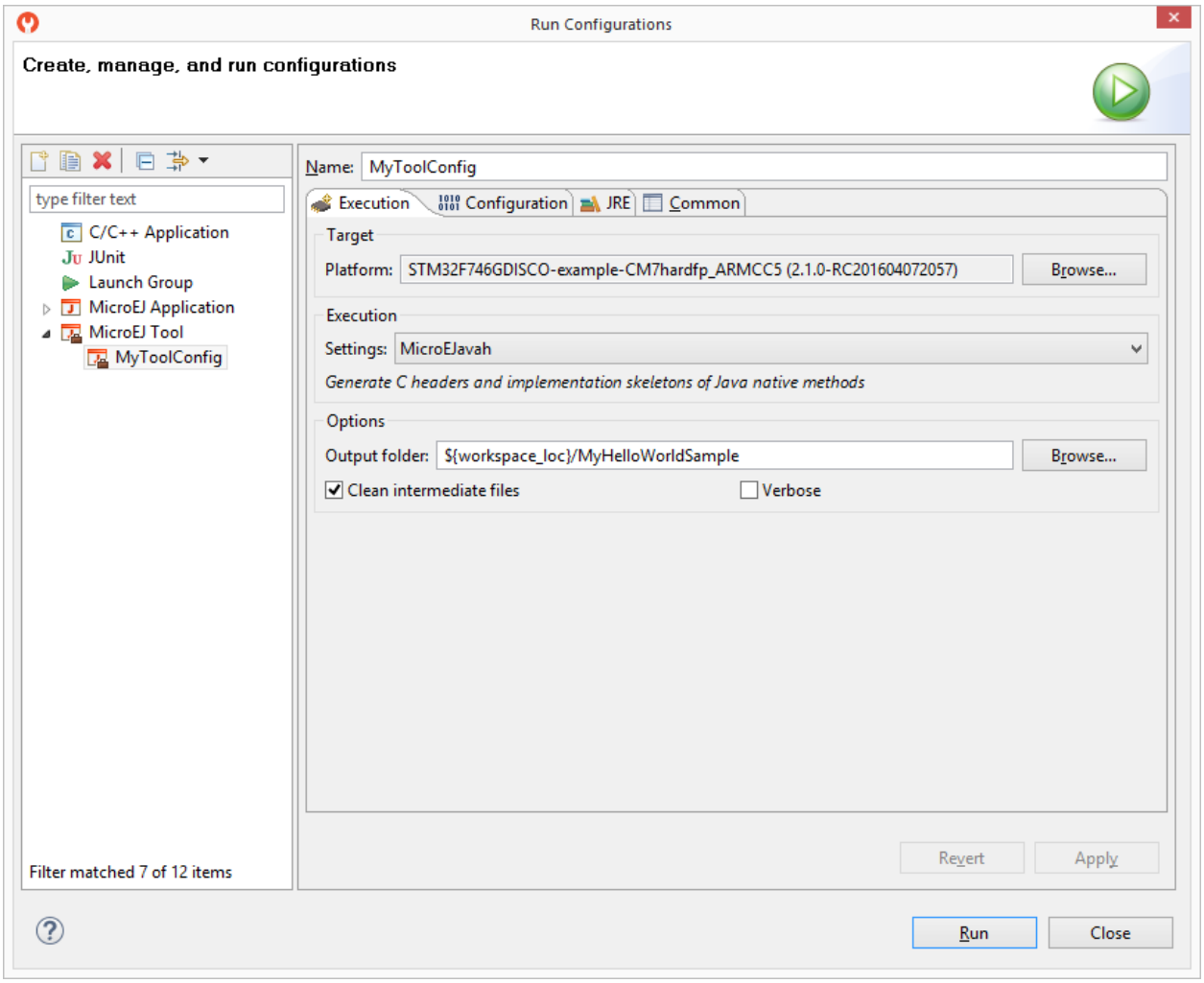


Fig. 25: MicroEJ Tool Configuration

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.

3.10.1 Testsuite with JUnit

MicroEJ allows to run unit tests using the standard JUnit API during the build process of a MicroEJ library or a MicroEJ Application. The *Testsuite Engine* runs tests on a target Platform 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="1.5.0"/>
```

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 in the MicroEJ workspace. The following steps assume that a platform has been previously imported into the MicroEJ Platform repository.

Go to Window > Preferences > MicroEJ > Platforms and 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.

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, a testsuite report is generated:

- in HTML format in module project location `target~/test/html/test/junit-noframes.html` . At the beginning of the file a summary is displayed, then all execution traces for each test executed are available.

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. 26: Example of MicroEJ Testsuite HTML Report

- in JUnit XML format, in 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. 27: Example of MicroEJ Testsuite XML Report

XML report file can also be open in the JUnit View. Right-click on the file > Open With > JUnit View :

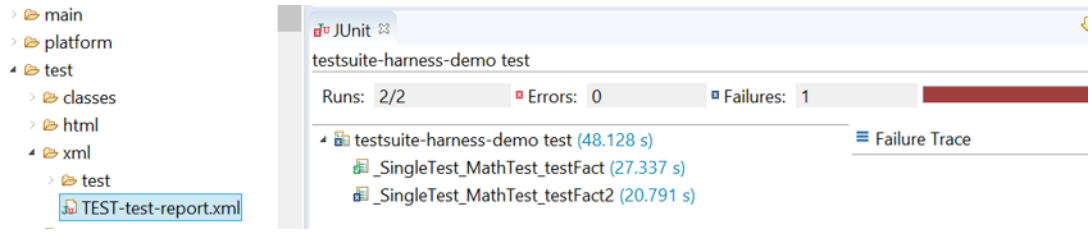


Fig. 28: Example of MicroEJ Testsuite XML Report in JUnit View

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`.

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 enty 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 *Testsuite Engine* allows to select the classes that will be executed, by setting the following property in the project `module.ivy` file.

```
<ea:property name="test.run.includes.pattern" value="[MicroEJ Test Case Include Pattern]"/>
```

The following line consider all JUnit test methods of the same class as a single MicroEJ test case (default behaviour). If at least one JUnit test method fails, the whole test case fails in the JUnit report.

```
<ea:property name="test.run.includes.pattern" value="**/_AllTests_*.class"/>
```

The following line consider 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 testsuite execution because a new deployment is done for each test method.

```
<ea:property name="test.run.includes.pattern" value="**/_SingleTest_*.class"/>
```

Run a Single Test Manually

Each test can be run independently as each class contains a main entry point.

In the `src-adpgenerated/junit/java` folder, right-click on the desired autogenerated class (`_SingleTest_[TestCase]_[TestMethod].java`) and select Run As > MicroEJ Application.

The test is executed on the selected Platform and the output result is dumped into the console.

Testsuite Options

The *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.

- Application Option Injection

  It is possible to inject an *Application Option* for all the tests, by adding to the original option the `microej.testsuite.properties.` prefix:

```
<ea:property name="microej.testsuite.properties.[application_option_name]" value="[application_
↪option_value]"/>
```

- Retry Mechanism

  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:

```
<ea:property name="microej.testsuite.retry.count" value="[nb_of_retries]"/>
```

  By default, when a test has failed, it is not executed again (option value is set to `0`).

Test Specific Options

The *Testsuite Engine* allows to define *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.

3.10.2 Stack Trace Reader

Principle

Stack Trace Reader is a MicroEJ tool which reads and decodes the MicroEJ stack traces. When an exception occurs, the MicroEJ Core Engine prints the stack trace on the standard output `System.out`. The class names, non required types (see *Types*) names and method names obtained are encoded with a MicroEJ internal format. This internal format prevents the embedding of all class names and method names in the flash, in order 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 number in the MicroEJ Application.

Functional Description

The Stack Trace Reader reads the debug info from the fully linked ELF file (the ELF file that contains the MicroEJ Core Engine, the other libraries, the BSP, the OS, and the compiled MicroEJ Application). It prints the decoded stack trace.

Dependencies

No dependency.

Installation

This tool is a built-in platform tool.

Use

Write a 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. 29: Code to Dump a Stack Trace

Write a new line to dump the currently executed stack trace on the standard output.

```
module.ivy    Test.java
1  package com.mycompany;
2
3  public class Test {
4
5      public static void main(String[] args) {
6          System.out.println("hello world!");
7          new Exception().printStackTrace();
8      }
9
10 }
```

Fig. 30: Code to Dump a Stack Trace

To be able 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 which is uploaded on the device is `application.fo` (stripped version without debug information).

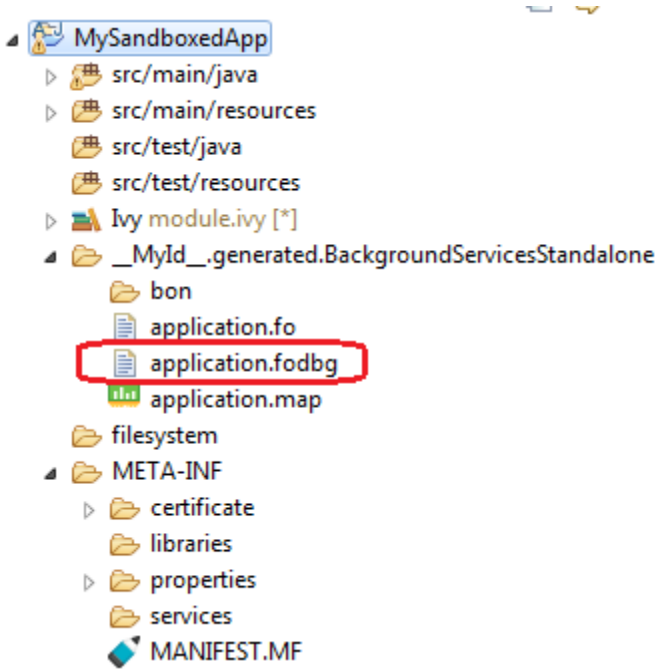


Fig. 31: Application Binary File with Debug Information

On successful deployment, the application is started on the device and the following trace is dumped on standard output.

```
MyBackgroundCode: Hello World
Exception in thread "ej.wadapps.app.default" java.lang.Throwable
     at java.lang.System.@M:0x803b334:0x803b344@
     at java.lang.Throwable.@M:0x8046aec:0x8046b02@
     at java.lang.Throwable.@M:0x805a0fc:0x805a11d@
     at appEntry.MyBackgroundCode.@F:1d48b23d5b010000d37548f1e20224d0b875cb968936fb41:0xc03800e0@@M:0xc0380bf8:0xc0380c20@
```

Fig. 32: Stack Trace Output

To create a new MicroEJ Tool configuration, right-click on the application project and click on Run As… > Run Configurations… .

In Execution tab, select the Stack Trace Reader tool.

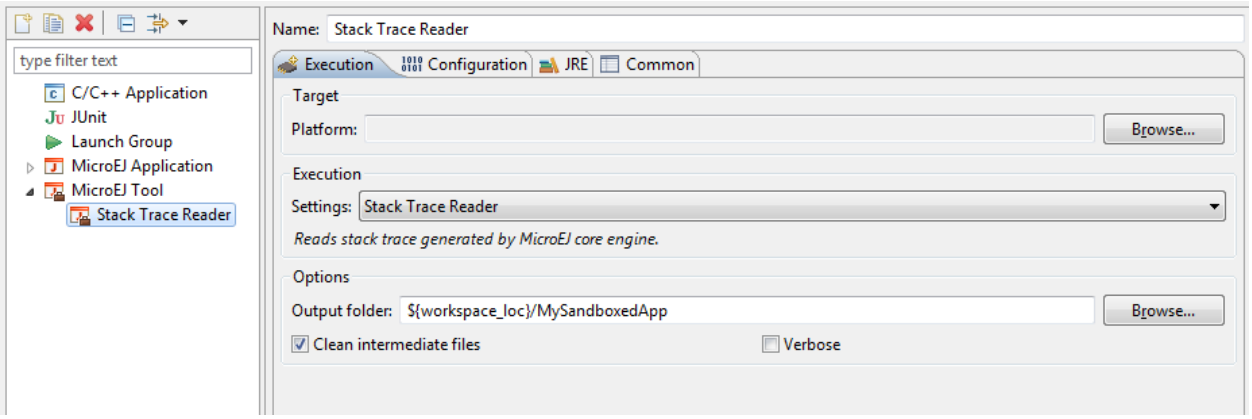


Fig. 33: Select Stack Trace Reader Tool

In Configuration tab, browse the previously generated application binary file with debug information (`application.fodbg` in case of a Sandboxed Application or `application.out` in case of a Standalone Application)

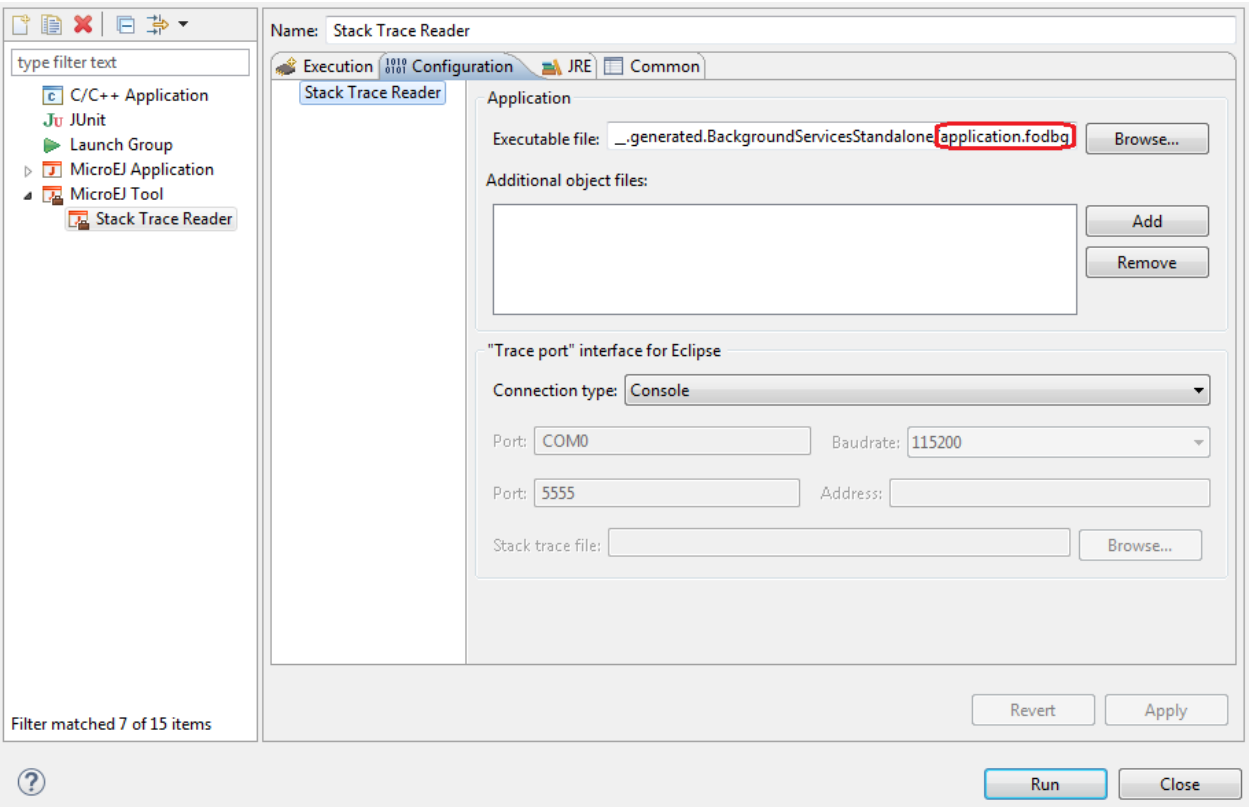


Fig. 34: Stack Trace Reader Tool Configuration (Sandboxed Application)

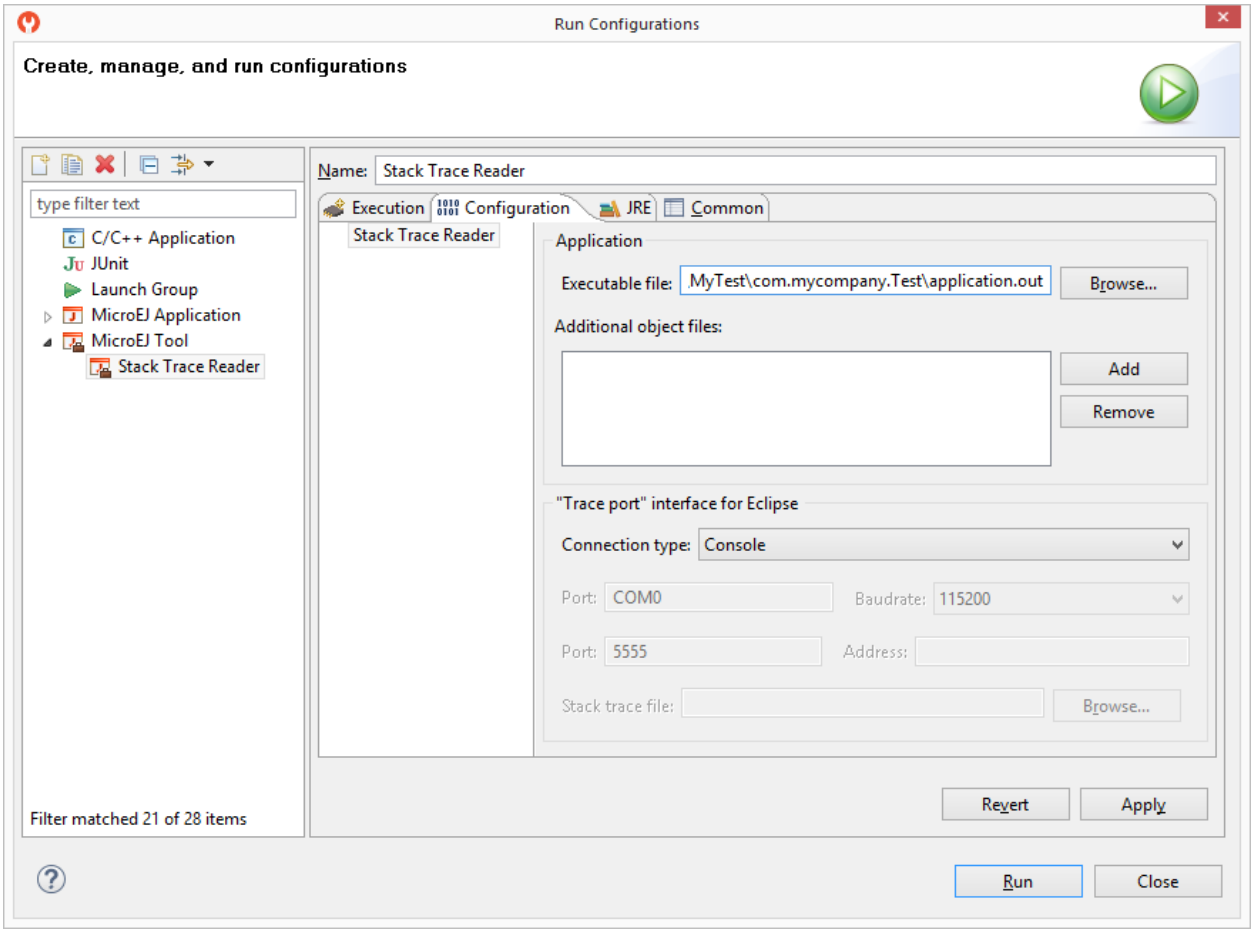


Fig. 35: Stack Trace Reader Tool Configuration (Standalone Application)

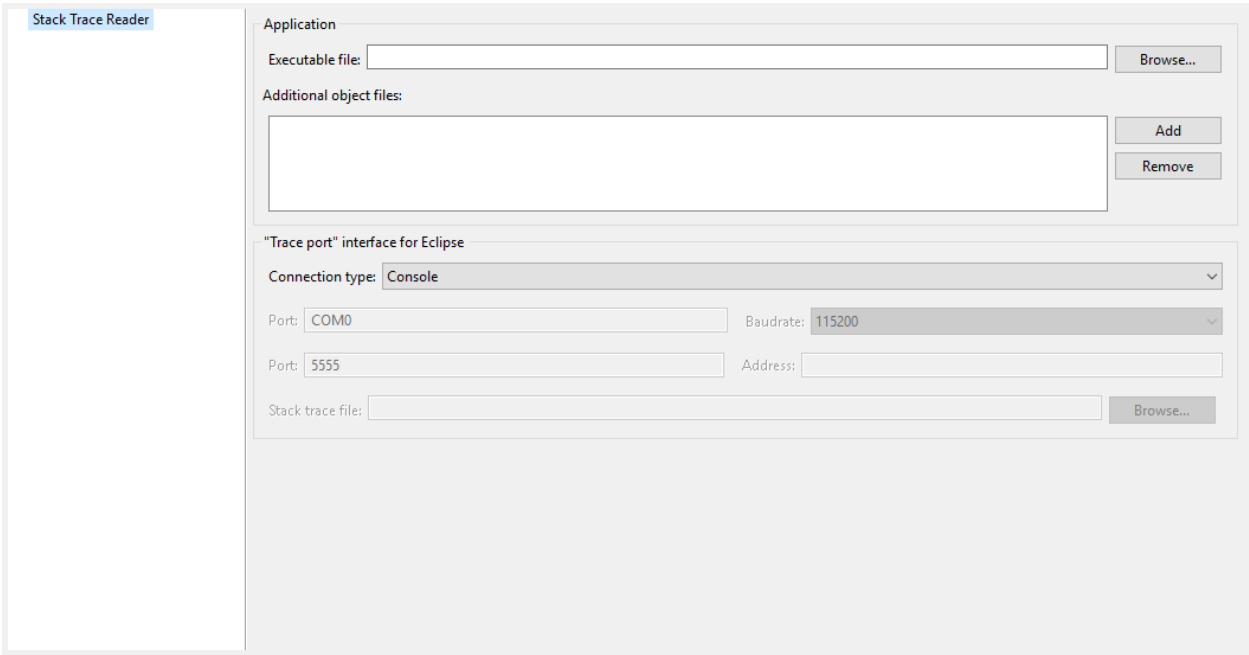
Click on Run button and copy/paste the trace into the Eclipse console. The decoded trace is dumped and the line corresponding to the application hook is now readable.

Fig. 36: Read the Stack Trace

The stack trace reader can simultaneously decode heterogeneous stack traces with lines owned by different applications and the firmware. Other debug information files can be appended using the Additional object files option. Lines owned by the firmware can be decoded with the firwmare debug information file (optionally made available by your firmware provider).

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)`

3.10.3 Code Coverage Analyzer

Principle

The MicroEJ 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.

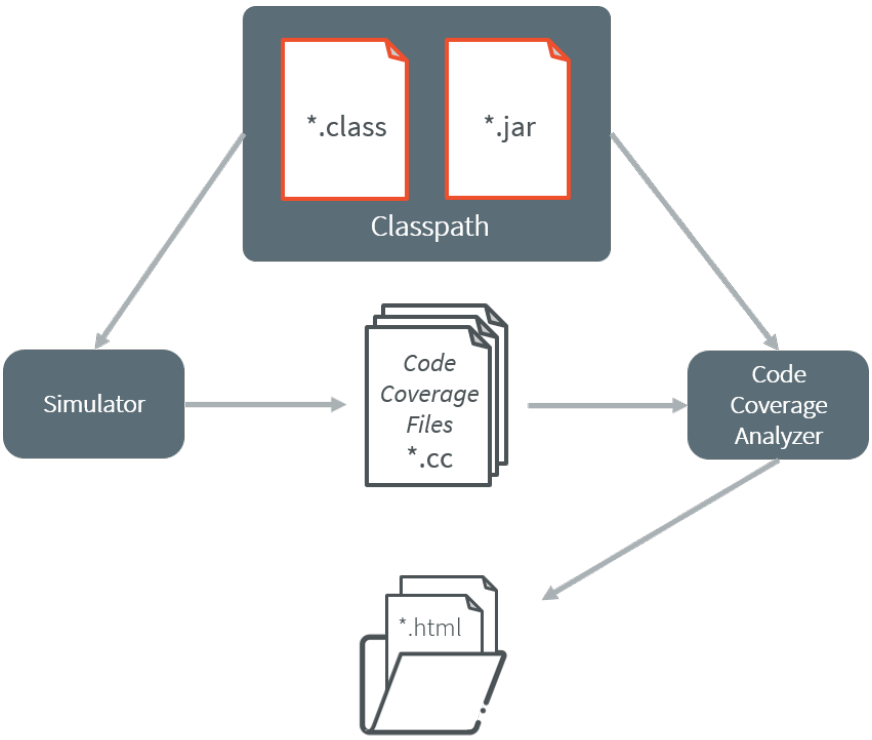


Fig. 37: Code Coverage Analyzer Process

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

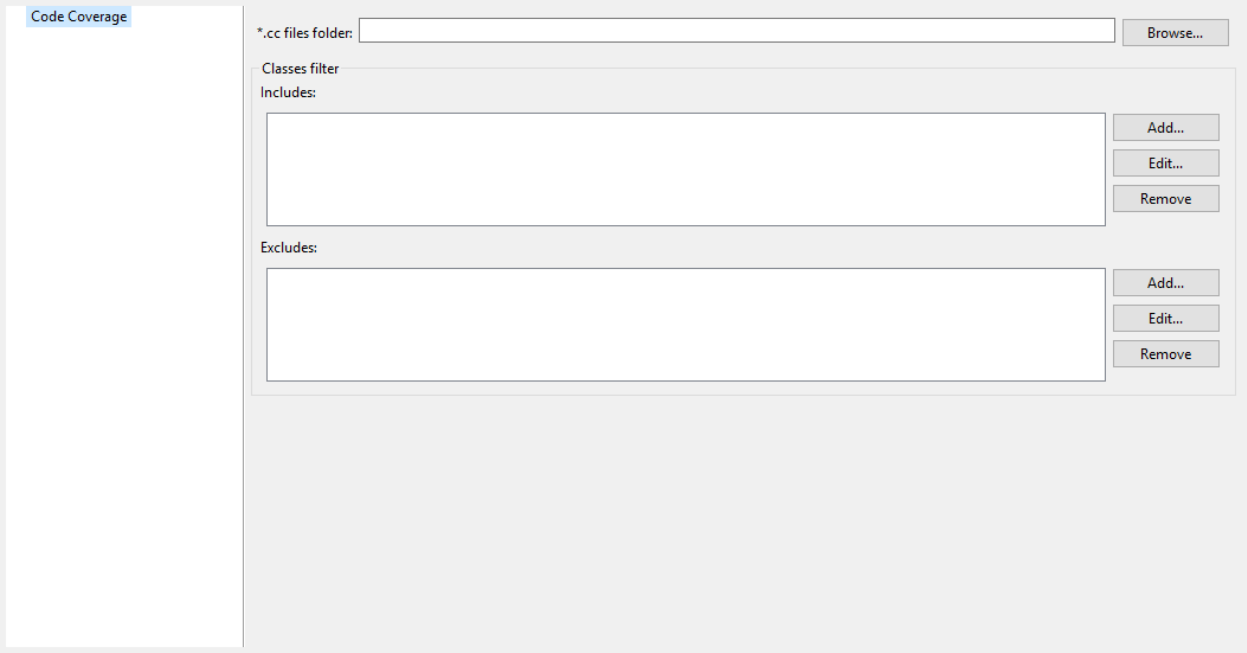
This tool is a built-in platform tool.

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`

3.10.4 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 on the application output folder. Note that it works only on simulations. It is a built-in platform tool and has no dependencies.

The Heap Analyzer is a set of tools to help developers understand the contents of the Java heap and find problems such as memory leaks. For its part, the Heap Analyzer plug-in 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

When the Heap Dumper option is activated, the garbage collector process ends by performing a dump file that represent a snapshot of the heap at this moment. Thus, to generate such dump files, you must explicitly call the System.gc() method in your code, or wait long enough for garbage collector activation.

The heap dump file contains the list of all 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:

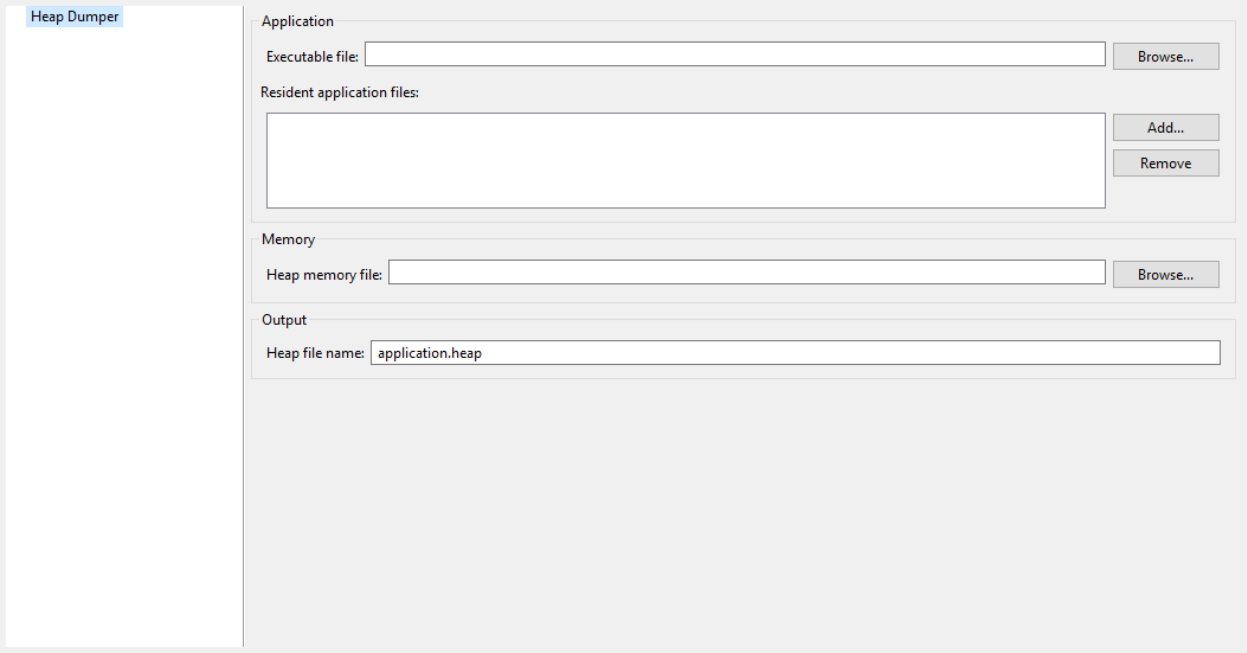
- the class
- the values in the instance's non-static fields

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.

Category: Heap Dumper



Group: Application

Option(browse): Executable file

Option Name: `application.filename`

Default value: `(empty)`

Description:

Specify the full path of a full linked ELF file.

Option(list): Resident application files

Option Name: `additional.application.filenames`

Default value: `(empty)`

Description:

Specify the full path of resident applications `.out` files linked by the Firmware Linker.

Group: Memory

Option(browse): Heap memory file

Option Name: `heap.filename`

Default value: `(empty)`

Description:

Specify the full path of heap memory dump, in Intel Hex format.

Group: Output

Option(text): Heap file name

Option Name: `output.name`

Default value: `application.heap`

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.

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.

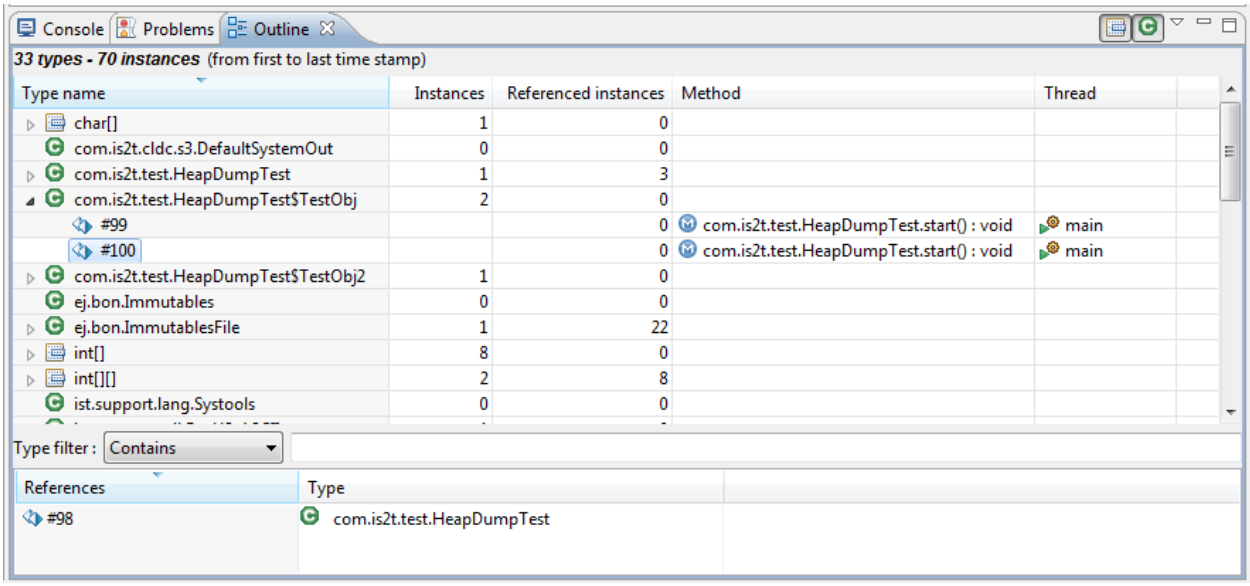


Fig. 38: Outline View

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.

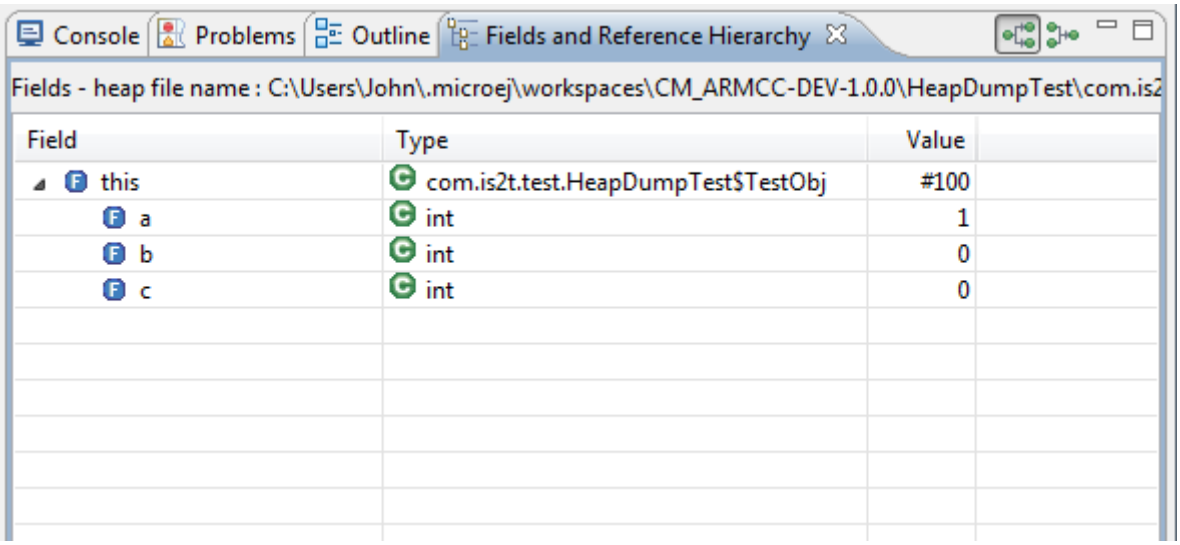


Fig. 39: Instance Browser View - Fields mode

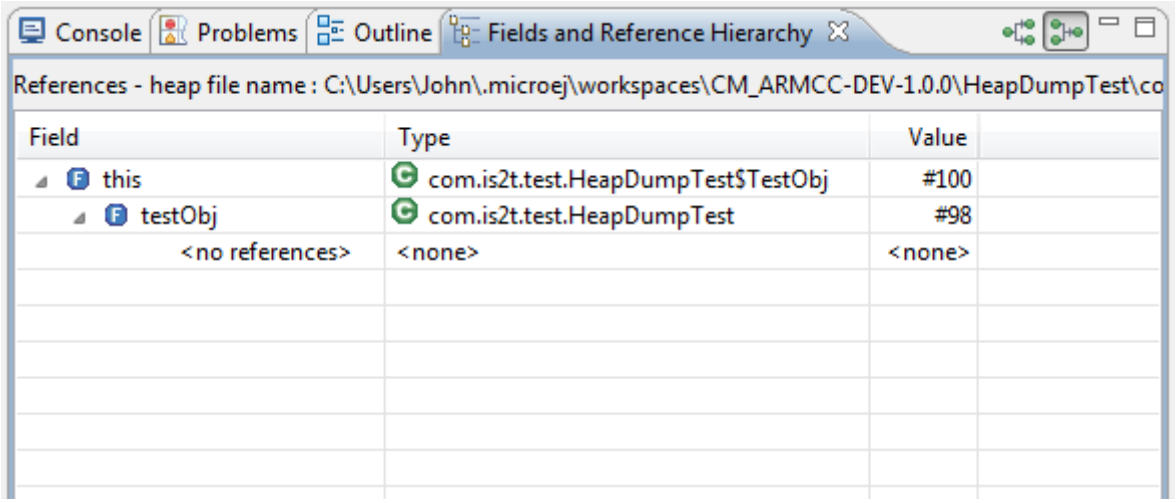


Fig. 40: Instance Browser View - References mode

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. 41: 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).

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).

[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.

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.

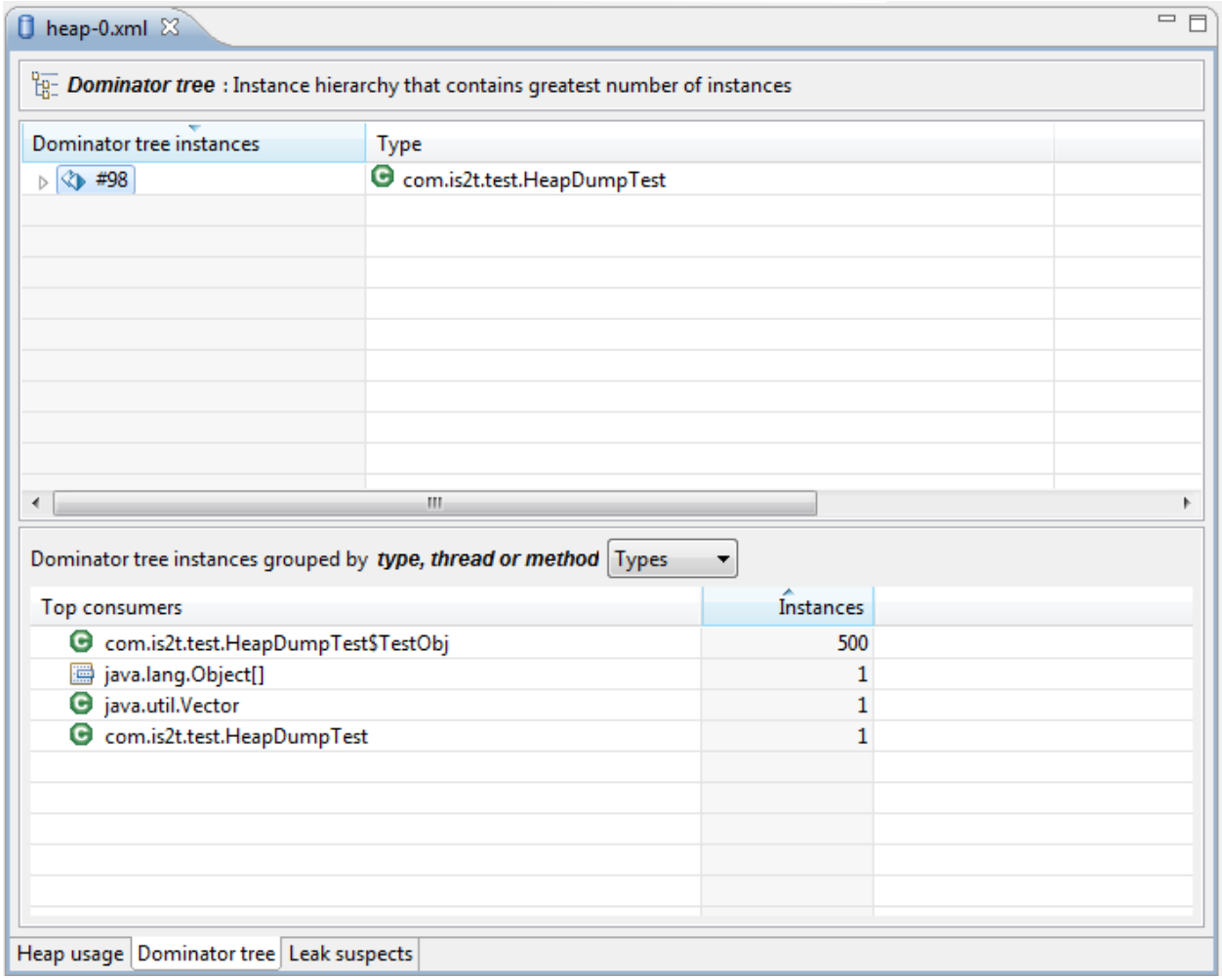


Fig. 42: Heap Viewer - Dominator Tree Tab

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.

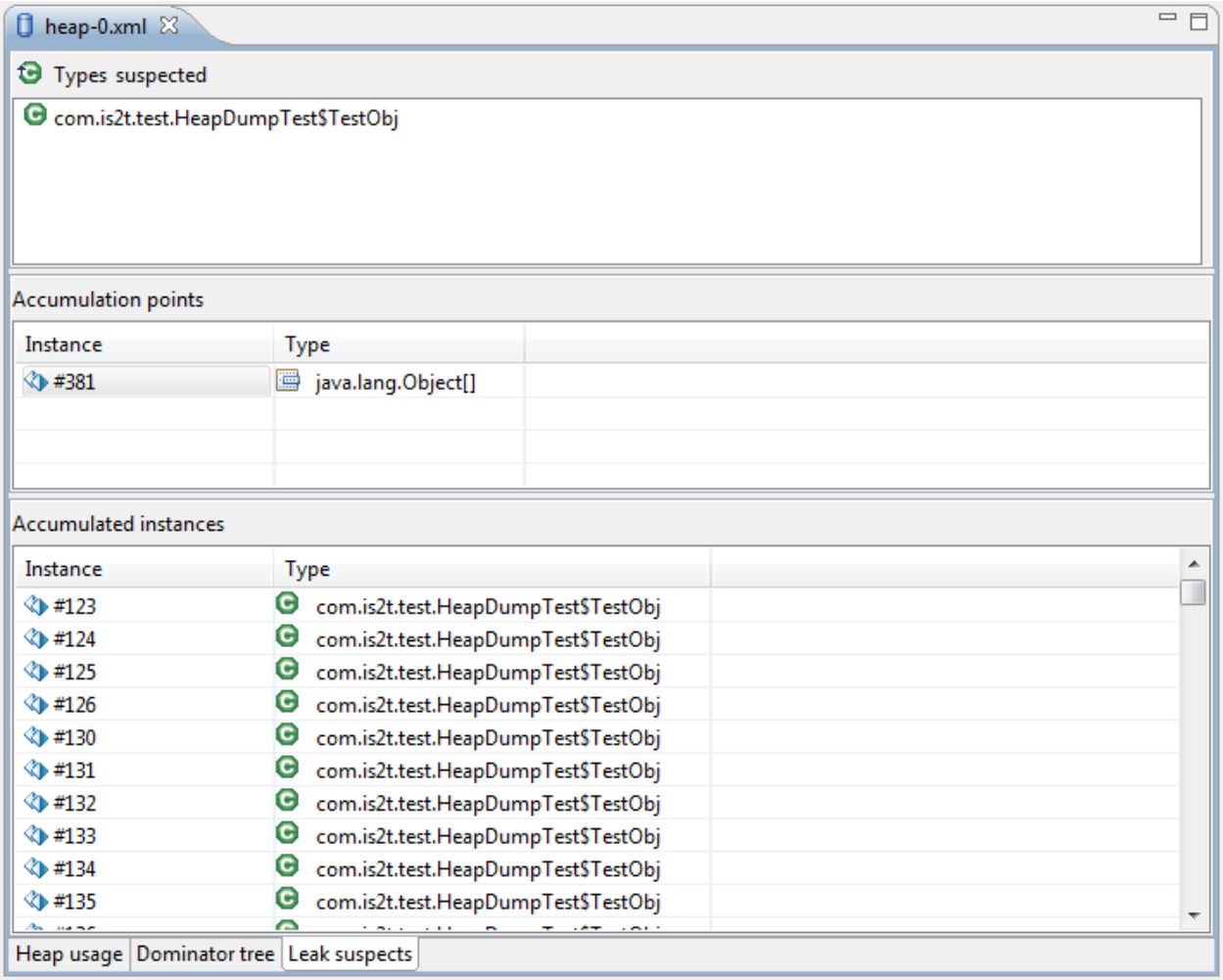


Fig. 43: Heap Viewer - Leak Suspects Tab

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.

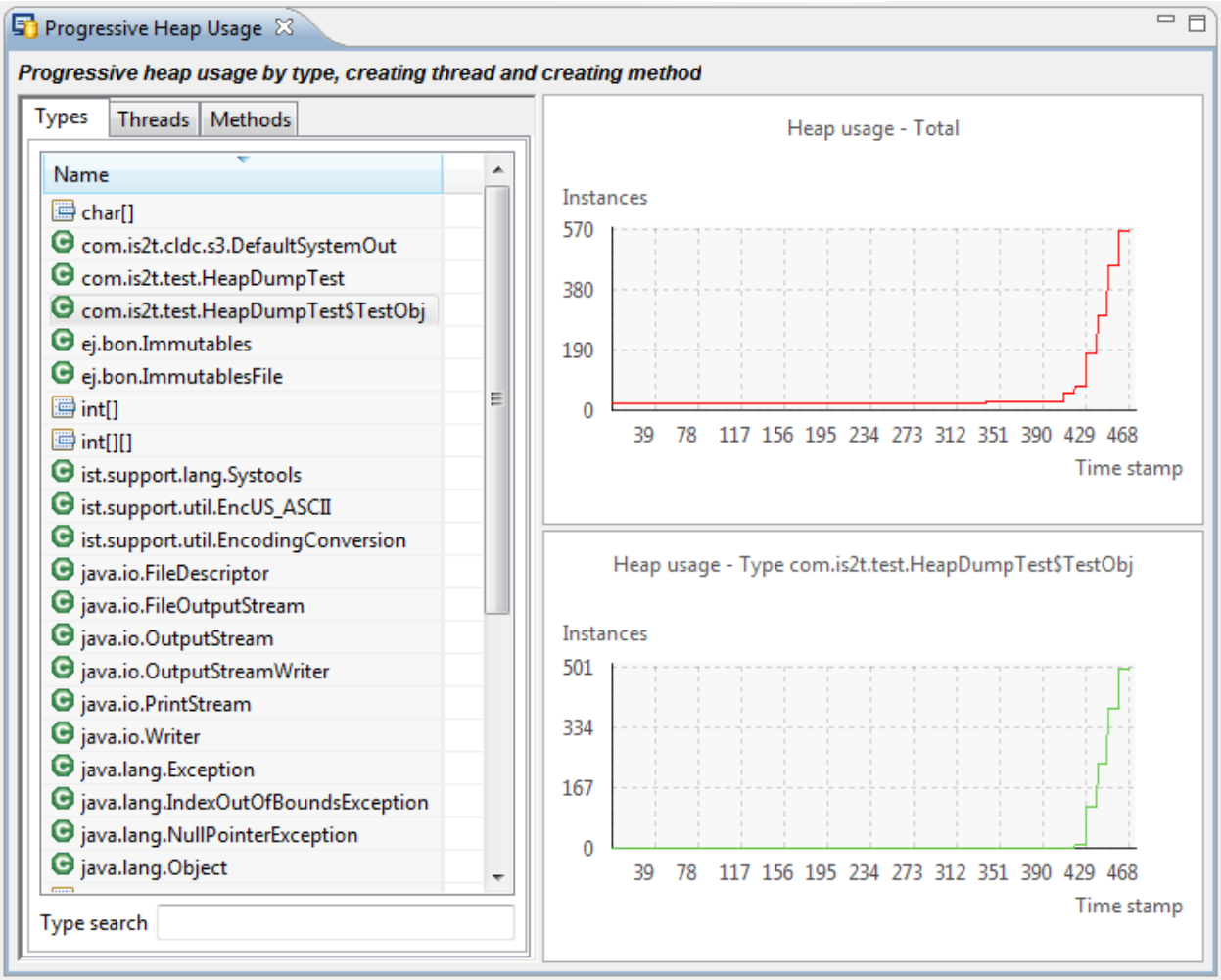


Fig. 44: Progressive Heap Usage

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.

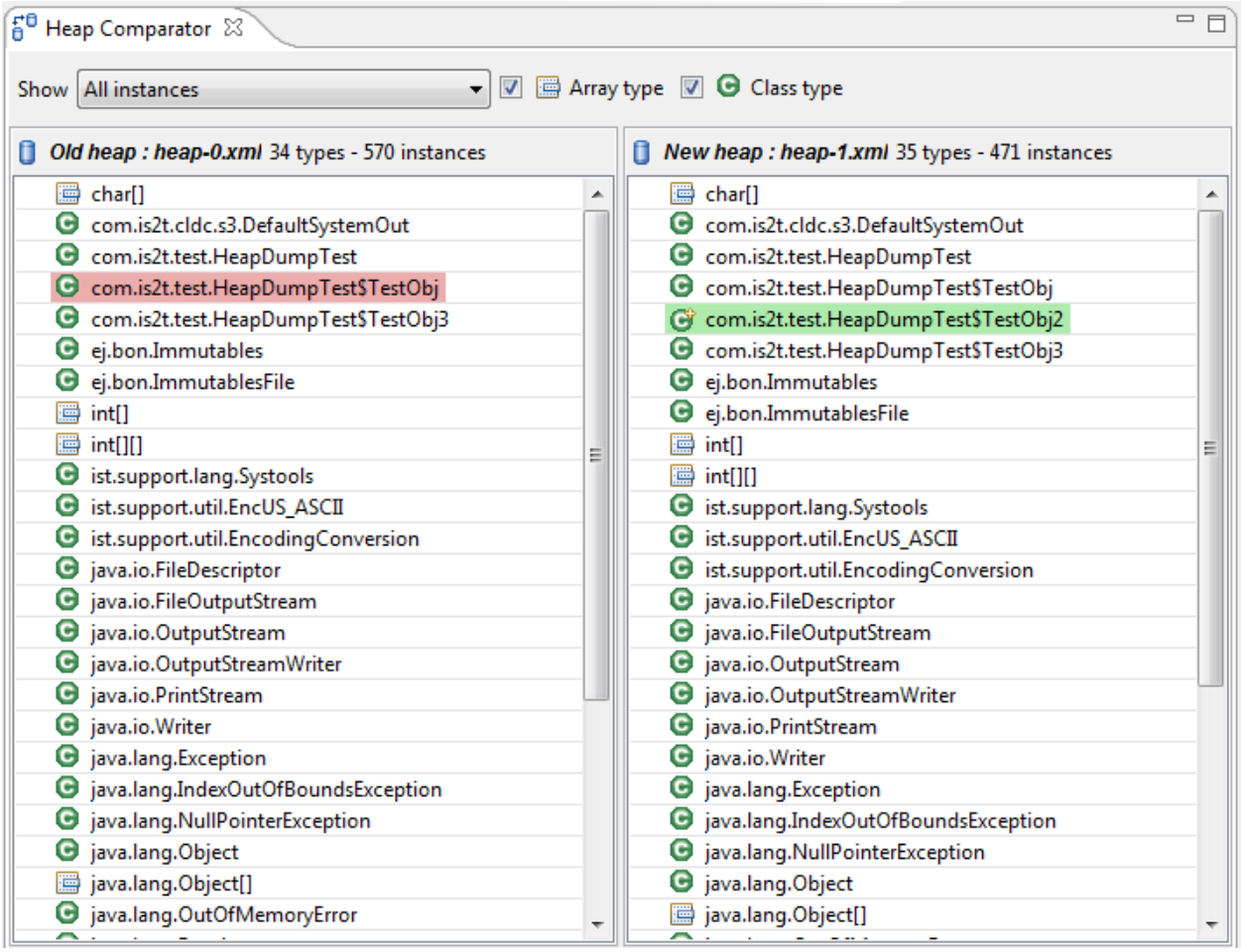


Fig. 45: Compare Heap Dumps

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.

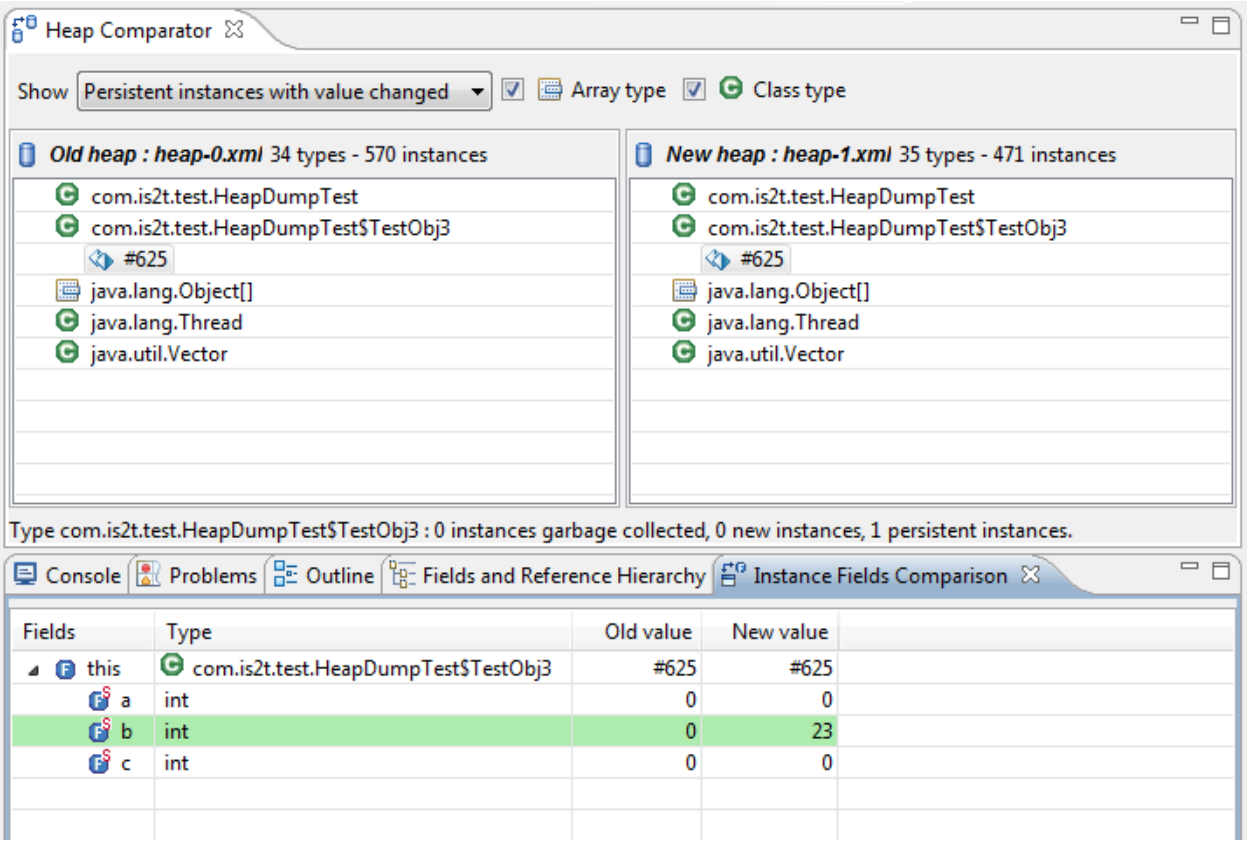


Fig. 46: Instance Fields Comparison view

3.10.5 ELF to Map File Generator

Principle

The ELF to Map generator takes an ELF executable file and generates a MicroEJ compliant `.map` file. Thus, any ELF executable file produced by third party linkers can be analyzed and interpreted using the *Memory Map Analyzer*.

Functional Description

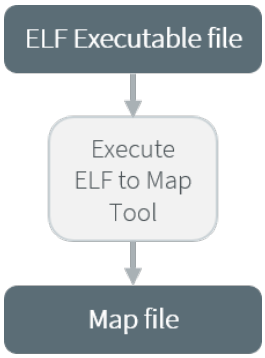


Fig. 47: ELF To Map Process

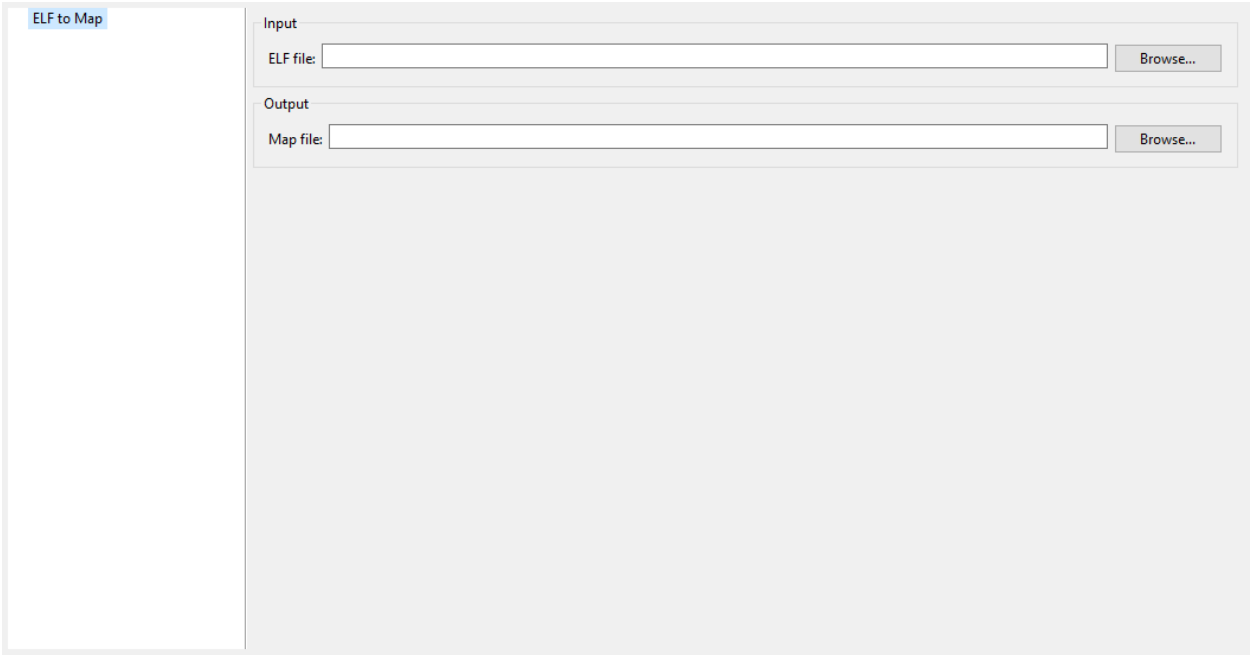
Installation

This tool is a built-in platform tool.

Use

This chapter explains MicroEJ tool options.

Category: ELF to Map



Group: Input

Option(browse): ELF file

Option Name: `input.file`

Default value: `(empty)`

Group: Output

Option(browse): Map file

Option Name: `output.file`

Default value: `(empty)`

3.10.6 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.

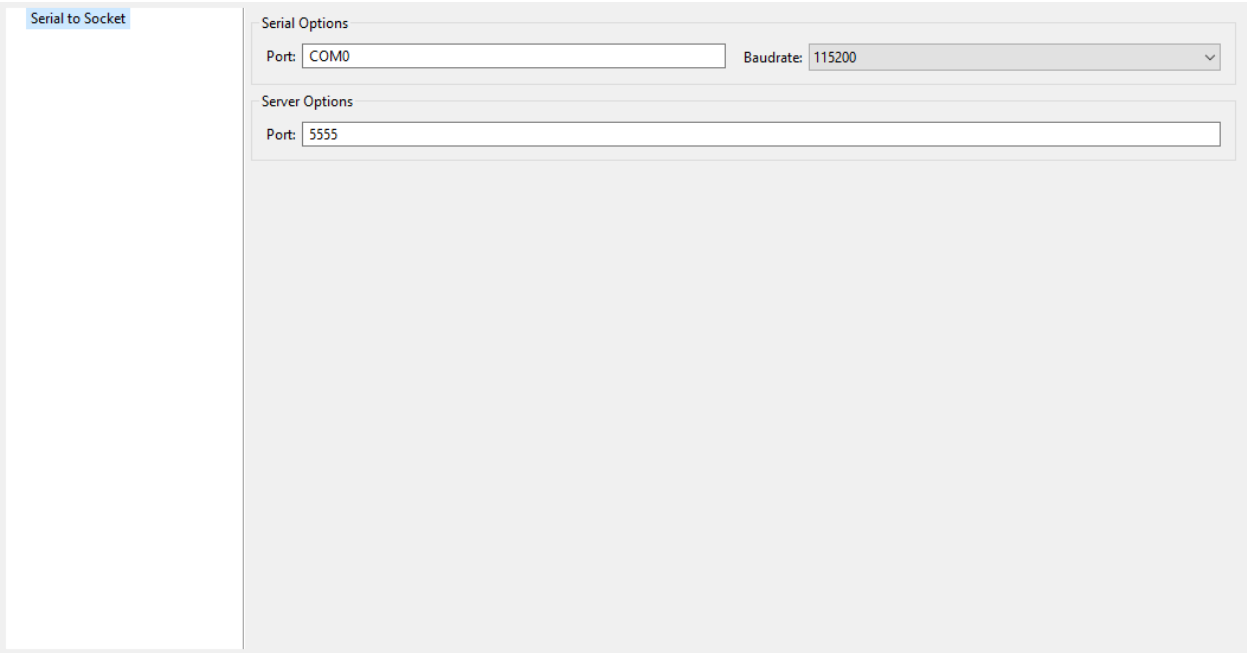
Installation

This tool is a built-in platform 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.

3.10.7 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

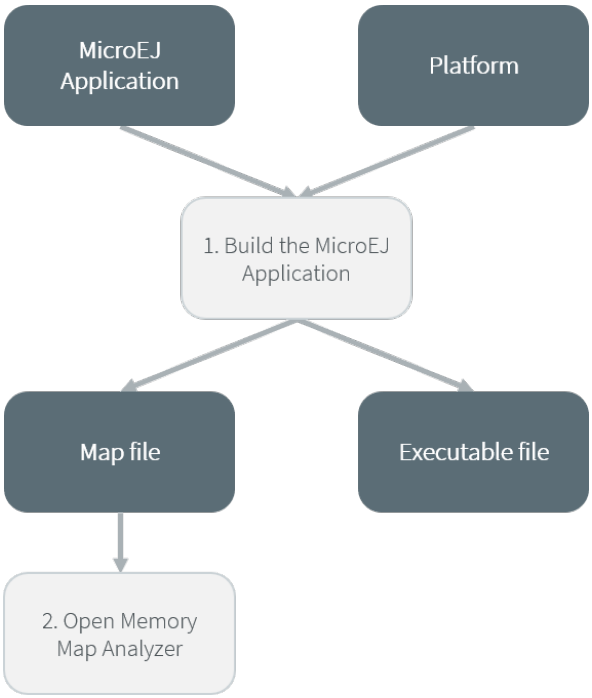


Fig. 48: Memory Map Analyzer Process

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 platform tool.

Use

The map file is available in the MicroEJ Application project output directory.

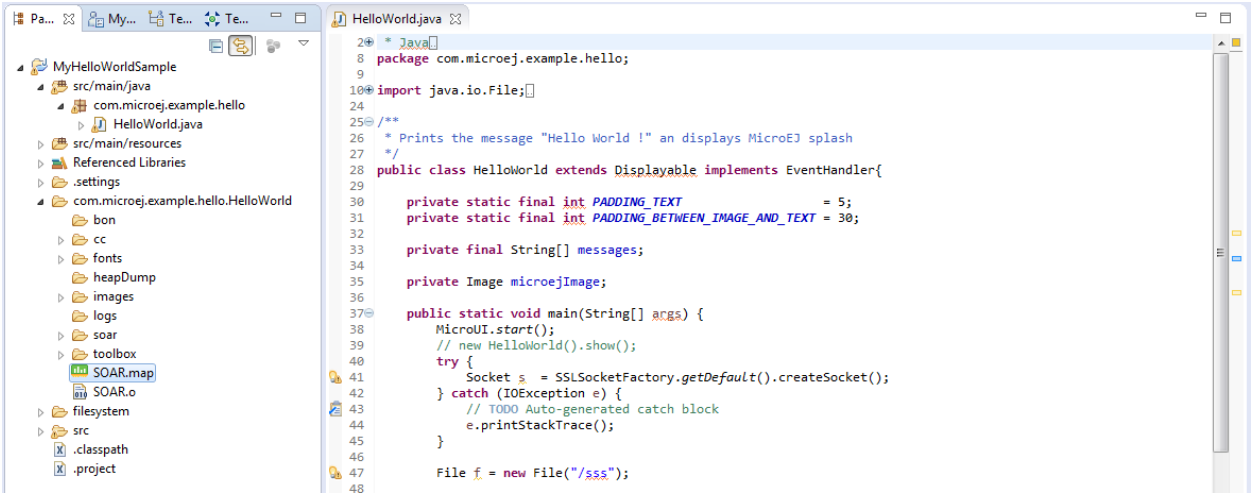


Fig. 49: Retrieve Map File

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.

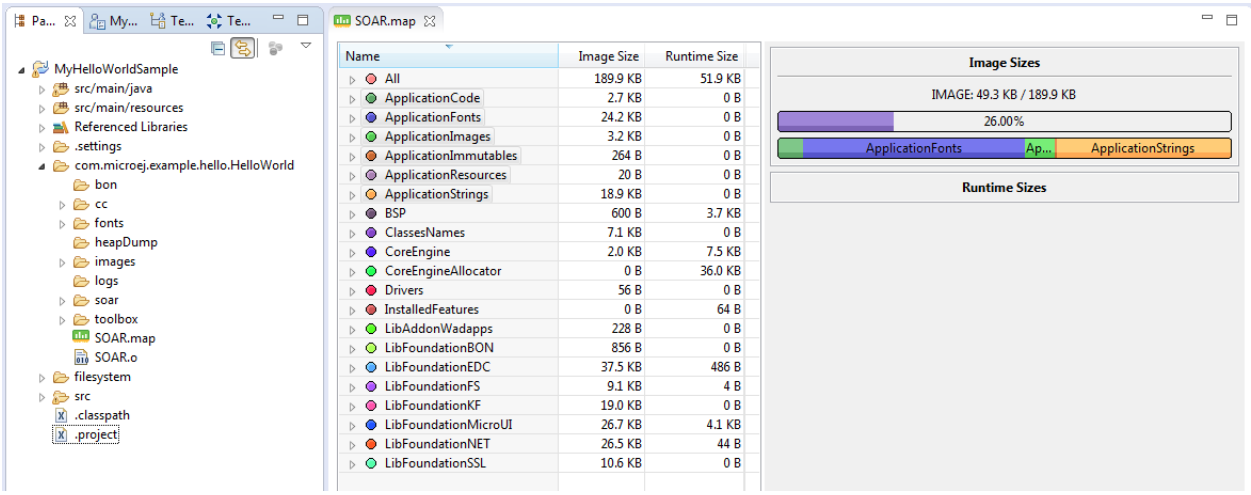


Fig. 50: Consult Full Memory

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.

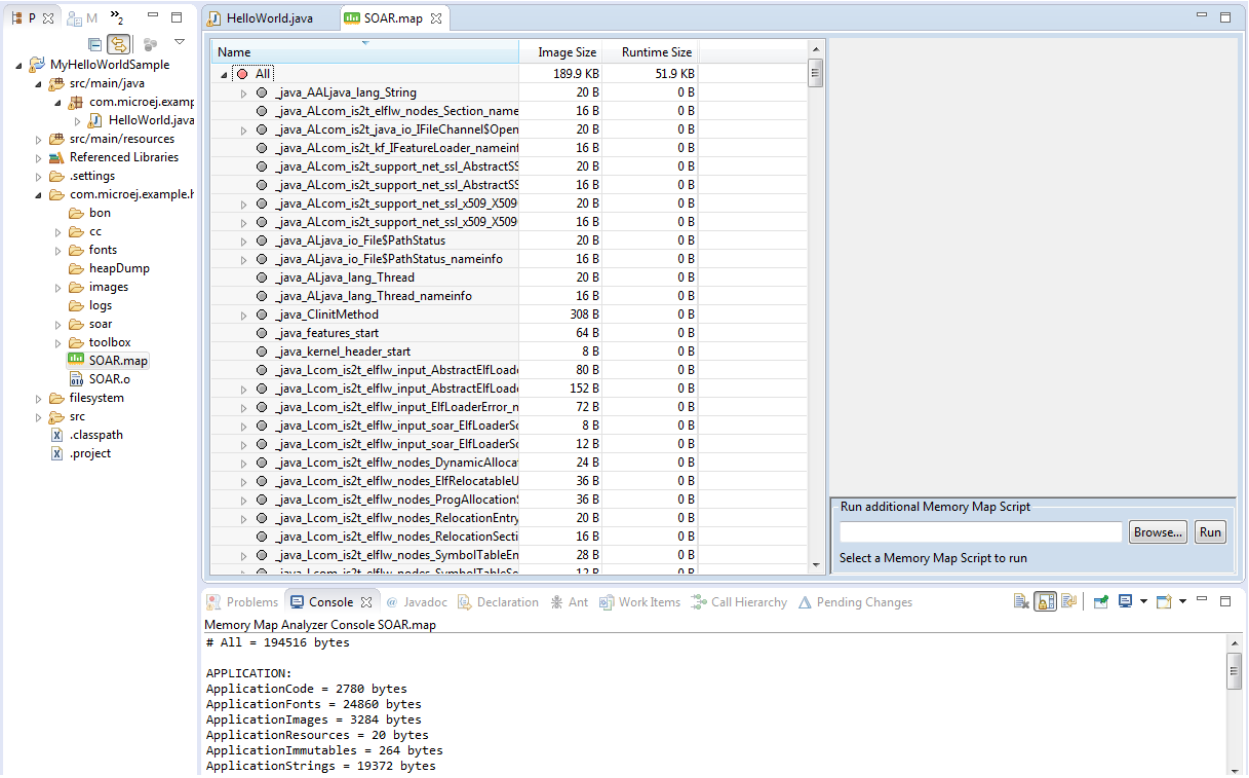


Fig. 51: Detailed View

3.10.8 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 *module.ivy* of the MicroEJ Application project:

```
<dependency org="ej.api" name="trace" rev="1.1.0"/>
```

- A C API, provided by the Platform header file named `LLTRACE_impl.h`.

Events are recorded if and only if:

- the MicroEJ Core Engine trace system is enabled,
- and trace recording is started.

To enable the MicroEJ Core Engine trace system, set the *Application Option* named `core.trace.enabled` to `true` (see also *launch configuration*).

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_IMPL_start(void)` and `LLTRACE_IMPL_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)` 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:

  ```
  VM 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);

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:

```
VM START
[TRACE] [1] Declare group "Application"
[TRACE] [1] Event 0x1 (14 [0xE],54 [0x36])
[TRACE] [1] Event End 0x1 (68 [0x44])
```

Platform Implementation

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.

See *Event Tracing* section for Platform specific implementations such as *SystemView*.

3.11 Advanced Tools

3.11.1 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 5: 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 6: 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 5: 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.

Continued on next page

Table 5 – continued from previous page

Tags	Attributes	Description
	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.
	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.

Continued on next page

Table 5 – continued from previous page

Tags	Attributes	Description
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.
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).

Continued on next page

Table 5 – continued from previous page

Tags	Attributes	Description
	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).
file		Binary section statement. Fills the section with the given expression. Bytes are organized in the endianness of the target ELF executable.
	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]"
```

(continues on next page)

(continued from previous page)

```
    ignoreWrongPositioningForEmptySection="[true|false]"
    lsc="[filename]"
    linkPath="[path1:...pathN]"
    mergeSegmentSections="[true|false]"
    noWarning="[true|false]"
    outputArchitecture="[tag]"
    outputName="[name]"
    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 6: 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 7: Linker-Specific Configuration Tags

Message ID	Description
0	The linker has encountered an unexpected internal error. Please contact the support hotline.

Continued on next page

Table 7 – continued from previous page

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.
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.

Continued on next page

Table 7 – continued from previous page

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/\ ... /\gn`.

- `union graphResult g1 ... gn`

  Creates a graph which is the union of `g1\/ ...\/ gn`.

- `substract graphResult g1 ... gn`

  Creates a graph which is the substract of `g1\ ... \ 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.

3.11.2 Testsuite Engine

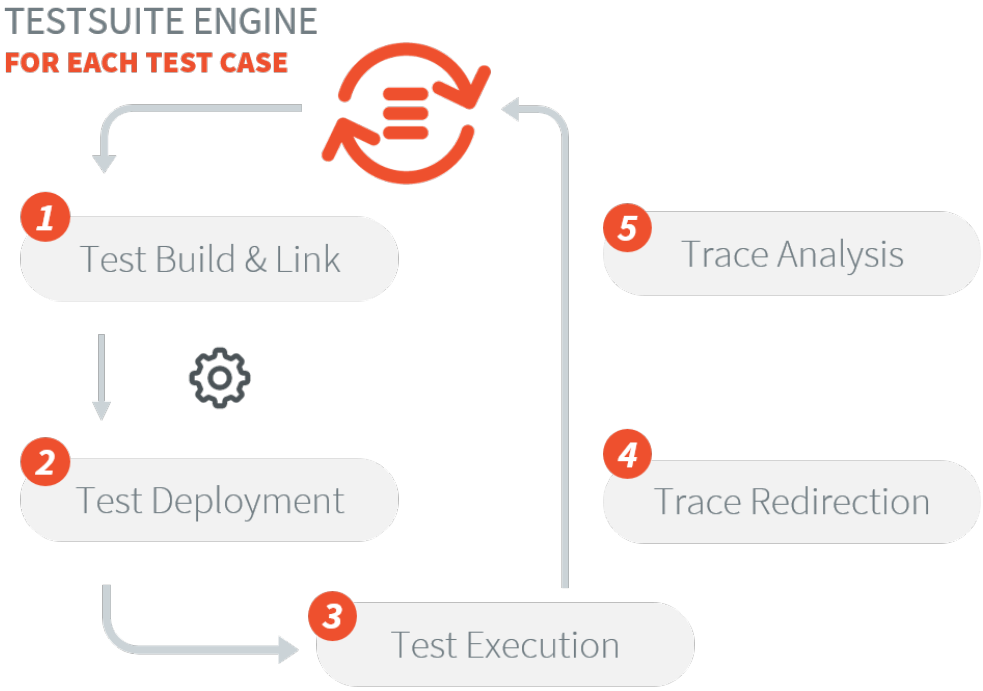
3.11.3 MicroEJ Testsuite Engine

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 MicroEJ Platform (either on Simulator or on Device).

- For Application and Libraries, refer to *Testsuite with JUnit* section.
- For Foundation Libraries Testsuites, refer to *Platform 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 8: `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 9: `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 10: `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 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 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 11: `testsuite` task nested elements

Element Name	Description
`testPath`	Containing all the file of the tests which will be launched by the testsuite.
`testIgnoredPath` (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 7: 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 12: `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 8: Example of javaTestsuite task invocation

```
<!-- Launch test suite -->
<testsuite:javaTestsuite
    verboseLevel="${microej.testsuite.verboseLevel}"
    timeOut="${microej.testsuite.timeout}"
    outputDir="${target.test.xml}/@{prefix}"
    harnessScript="${harness.file}"
    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"/>
```

(continues on next page)

(continued from previous page)

```
    <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 test suite 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 9: 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.

3.12 Graphical User Interface

This section presents libraries relative to the user interface.

The following schema shows the overall architecture and modules:

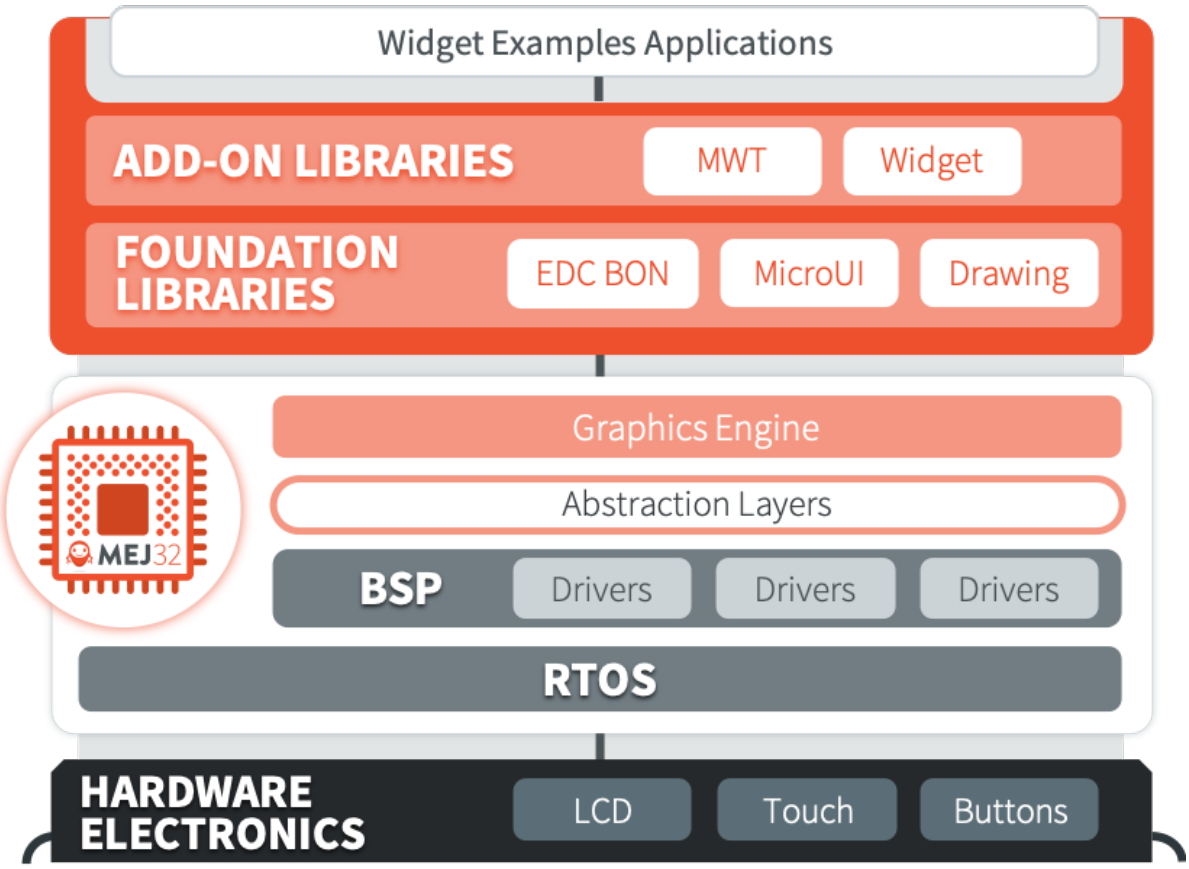


Fig. 52: Graphical User Interface Overview

Note: This chapter describes the current Graphical User Interface version `3`, provided by UI Pack version `13.0.0` or higher. If you are using the former Graphical User Interface version `2` (provided by MicroEJ UI Pack version up to `12.4.x`), please refer to this MicroEJ Documentation Archive.

3.12.1 MicroUI

Introduction

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 by reifying hardware capabilities.

To use the MicroUI Foundation Library, add MicroUI API module to a *module description file*:

```
<dependency org="ej.api" name="microui" rev="3.0.0"/>
```

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 a *module description file*:

```
<dependency org="ej.api" name="drawing" rev="1.0.0"/>
```

Images

Overview

Images are graphical resources that can be accessed with a call to ej.microui.display.Image.getImage() or ej.microui.display.ResourceImage.loadImage() . To be displayed, these images have to be converted from their source format to the display raw format. The conversion can either be done at:

- build-time (using the image generator tool),
- run-time (using the relevant decoder library).

Images that must be processed by the image generator tool are declared in *MicroEJ Classpath* `*.images.list` files. 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 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 (although listing files without format specifier has no impact on the image generator processing, it is advised to specify them in the `*.images.list` files anyway, as it makes the run-time processing behavior explicit). 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 bits format without transparency (decoded at build-time)
com/mycompany/MyImage2.png:RGB565

# The following image is embedded
# as a 16 bits format with transparency (decoded at build-time)
com/mycompany/MyImage3.png:ARGB1555
```

[1] These APIs were formerly included in MicroUI `2.x`

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'
```

Images Heap

The images heap is used to allocate the pixel data of:

- mutable images (i.e. BufferedImage instances)
- images which are not byte-addressable, such as images opened with an input stream
- images which are byte-addressable but converted to a different output format

In other words, every image which can not be retrieved using Image.getImage() is saved on the images heap.

The size of the images heap can be configured with the `ej.microui.memory.imagesheap.size` property.

Output Formats

Without Compression

When no output format is set in the images list file, the image is embedded without any conversion / compression. This allows you to embed the resource as well, in order to keep the source image characteristics (compression, bpp, etc.). This option produces the same result as specifiying an image as a resource in the MicroEJ launcher.

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
```

Standard Output Formats

Depending on the target hardware, several generic output formats are available. Some formats may be directly managed by the BSP display driver. Refer to the platform specification to retrieve the list of natively supported formats.

Advantages:

- The pixels layout and bits 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);
- Supports or not the alpha encoding: select the most suitable format for the image to encode.

Disadvantages:

- No compression: the image size in bytes is proportional to the number of pixels, the transparency, and the bits-per-pixel;
- Slower than `display` format when the display driver does not recognize the format: a pixel conversion is required at runtime.

Select one the following format to use a generic format among this list: `ARGB8888`, `RGB888`, `ARGB4444`, `ARGB1555`, `RGB565`, `A8`, `A4`, `A2`, `A1`, `C4`, `C2`, `C1`, `AC44`, `AC22` and `AC11`. The following snippets describe the color conversion for each format:

- ARGB8888: 32 bits format, 8 bits for transparency, 8 per color.

```
int convertARGB8888toRAWFormat(int c){
    return c;
}
```

- RGB888: 24 bits format, 8 per color. Image is always fully opaque.

```
int convertARGB8888toRAWFormat(int c){
    return c & 0xffffff;
}
```

- ARGB4444: 16 bits format, 4 bits for transparency, 4 per color.

```
int convertARGB8888toRAWFormat(int c){
    return 0
            | ((c & 0xf0000000) >> 16)
            | ((c & 0x00f00000) >> 12)
            | ((c & 0x0000f000) >> 8)
            | ((c & 0x000000f0) >> 4)
            ;
}
```

- ARGB1555: 16 bits format, 1 bit for transparency, 5 per color.

```
int convertARGB8888toRAWFormat(int c){
    return 0
            | (((c & 0xff000000) == 0xff000000) ? 0x8000 : 0)
            | ((c & 0xf80000) >> 9)
            | ((c & 0x00f800) >> 6)
            | ((c & 0x0000f8) >> 3)
            ;
}
```

- RGB565: 16 bits format, 5 or 6 per color. Image is always fully opaque.

```
int convertARGB8888toRAWFormat(int c){
    return 0
            | ((c & 0xf80000) >> 8)
            | ((c & 0x00fc00) >> 5)
            | ((c & 0x0000f8) >> 3)
            ;
}
```

- A8: 8 bits format, only transparency is encoded. The color to apply when drawing the image, is the current GraphicsContext color.

```
int convertARGB8888toRAWFormat(int c){
    return 0xff - (toGrayscale(c) & 0xff);
}
```

- A4: 4 bits format, only transparency is encoded. The color to apply when drawing the image, is the current GraphicsContext color.

```
int convertARGB8888toRAWFormat(int c){
    return (0xff - (toGrayscale(c) & 0xff)) / 0x11;
}
```

- A2: 2 bits format, only transparency is encoded. The color to apply when drawing the image, is the current GraphicsContext color.

```
int convertARGB8888toRAWFormat(int c){
    return (0xff - (toGrayscale(c) & 0xff)) / 0x55;
}
```

- A1: 1 bit format, only transparency is encoded. The color to apply when drawing the image, is the current GraphicsContext color.

```
int convertARGB8888toRAWFormat(int c){
    return (0xff - (toGrayscale(c) & 0xff)) / 0xff;
}
```

- C4: 4 bits format with grayscale conversion. Image is always fully opaque.

```
int convertARGB8888toRAWFormat(int c){
    return (toGrayscale(c) & 0xff) / 0x11;
}
```

- C2: 2 bits format with grayscale conversion. Image is always fully opaque.

```
int convertARGB8888toRAWFormat(int c){
    return (toGrayscale(c) & 0xff) / 0x55;
}
```

- C1: 1 bit format with grayscale conversion. Image is always fully opaque.

```
int convertARGB8888toRAWFormat(int c){
    return (toGrayscale(c) & 0xff) / 0xff;
}
```

- AC44: 4 bits for transparency, 4 bits with grayscale conversion.

```
int convertARGB8888toRAWFormat(int c){
    return 0
        | ((color >> 24) & 0xf0)
        | ((toGrayscale(color) & 0xff) / 0x11)
        ;
}
```

- AC22: 2 bits for transparency, 2 bits with grayscale conversion.

```
int convertARGB8888toRAWFormat(int c){
    return 0
        | ((color >> 28) & 0xc0)
        | ((toGrayscale(color) & 0xff) / 0x55)
        ;
}
```

- AC11: 1 bit for transparency, 1 bit with grayscale conversion.

```
int convertARGB8888toRAWFormat(int c){
    return 0
        | ((c & 0xff000000) == 0xff000000 ? 0x2 : 0x0)
        | ((toGrayscale(color) & 0xff) / 0xff)
        ;
}
```

Examples:

```
image1:ARGB8888
image2:RGB565
image3:A4
```

Display Output Format

This format 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 display pixel format allow it.

Disadvantages:

- No compression: the image size in bytes is proportional to the number of pixels.

```
image1:display
```

RLE1 Output Format

The image engine can display embedded images that are encoded into a compressed format which encodes several consecutive pixels into one or more 16-bit words. This encoding manages a maximum alpha level of 2 (alpha level is always assumed to be 2, even if the image is not transparent).

- Several consecutive pixels have the same color (2 words):
  - First 16-bit word specifies how many consecutive pixels have the same color (pixels colors converted in RGB565 format, without opacity data).

  - Second 16-bit word is the pixels' color in RGB565 format.
- Several consecutive pixels have their own color (1 + n words):
  - First 16-bit word specifies how many consecutive pixels have their own color;
  - Next 16-bit word is the next pixel color.
- Several consecutive pixels are transparent (1 word):
  - 16-bit word specifies how many consecutive pixels are transparent.

Advantages:

- Supports fully opaque and fully transparent encoding.
- Good compression when several consecutive pixels respect one of the three previous rules.

Disadvantages:

- Drawing an image is slightly slower than when using Display format.
- Not designed for images with many different pixel colors: in such case, the output file size may be larger than the original image file.

```
image1:RLE1
```

Image Generator Error Messages

These errors can occur while preprocessing images.

Table 13: 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	Warning	The specified format is not managed by the image generator: The image will be not converted.
17	Warning	The image has been already loaded with another output format. The image will be not converted.

Fonts

Overview

Fonts are graphical resources that can be accessed with a call to ej.microui.display.Font.getFont(). To be displayed, these fonts have to be converted at build-time from their source format to the display raw format by the font generator tool. Fonts that must be processed by the font generator tool are declared in MicroEJ Classpath `*.fonts.list` files. 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 MicroEJ font file (usually with a `.ejf` file 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.ejf

# The following font is embedded with only the latin
# unicode range without transparency
com/mycompany/MyFont2.ejf:latin

# The following font is embedded with all characters
# with 2 levels of transparency
com/mycompany/MyFont2.ejf::2
```

MicroEJ font files conventionally end with the `.ejf` suffix and are created using the Font Designer (see *Font Designer*).

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://www.unicode.org).

Table 14: 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

Continued on next page

Table 14 – continued from previous page

Name	Tag	Start	End
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
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

Continued on next page

Table 14 – continued from previous page

Name	Tag	Start	End
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
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

Continued on next page

Table 14 – continued from previous page

Name	Tag	Start	End
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
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

Continued on next page

Table 14 – continued from previous page

Name	Tag	Start	End
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 transparency level encoded in EJF file).

Examples:

- `myfont:latin:4` : Embed all latin characters with 16 levels of transparency
- `myfont::2` : Embed all characters with 4 levels of transparency

Font Generator Error Messages

Table 15: 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	Warning	The specified entry is invalid: The current entry is ignored.
11	Warning	The specified entry does not contain a list of characters: The current entry is ignored.
12	Warning	The specified entry does not contain a list of identifiers: The current entry is ignored.
13	Warning	The specified entry is an invalid width: The current entry is ignored.
14	Warning	The specified entry is an invalid height: the current entry is ignored.
15	Warning	The specified entry does not contain the characters' addresses: The current entry is ignored.
16	Warning	The specified entry does not contain the characters' bitmaps: The current entry is ignored.
17	Warning	The specified entry bits-per-pixel value is invalid: The current entry is ignored.
18	Warning	The specified range is invalid: The current entry is ignored.
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	Warning	Dynamic styles are not supported. Only a PLAIN font can be encoded. The current entry is ignored.
24	Warning	Underlined style is not supported. Only a BOLD and ITALIC font can be set. The current entry is ignored.

Font Designer

Principle

The Font Designer module is a graphical tool (Eclipse plugin) that runs within the MicroEJ IDE used to build and edit MicroUI fonts. It stores fonts in a platform-independent format.

Functional Description

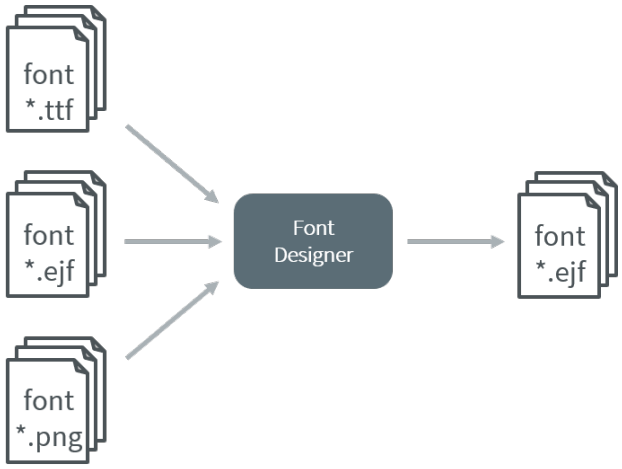


Fig. 53: Font Generation

Font Management

Create a MicroEJ Font

To create a MicroEJ 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 a MicroEJ 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.

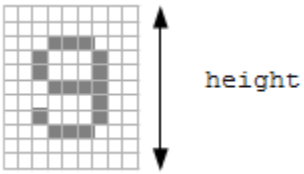


Fig. 54: Font Height

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 '}'.

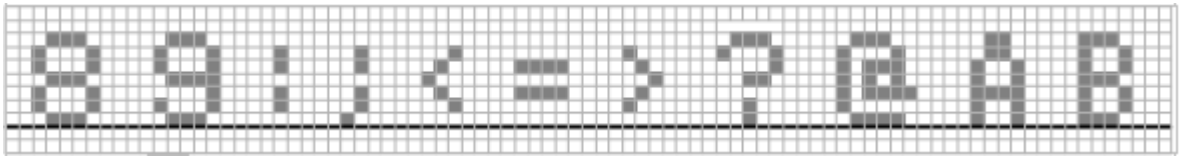


Fig. 55: The Baseline

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 to create a font file which 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`. By consequence, the styles option must be left to the default option.

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`. Use only `None` option. Otherwise an error at MicroEJ application compiletime will occur (incompatible font file).

Identifiers

A number of identifiers can be attached to a MicroUI 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 MicroEJ 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.

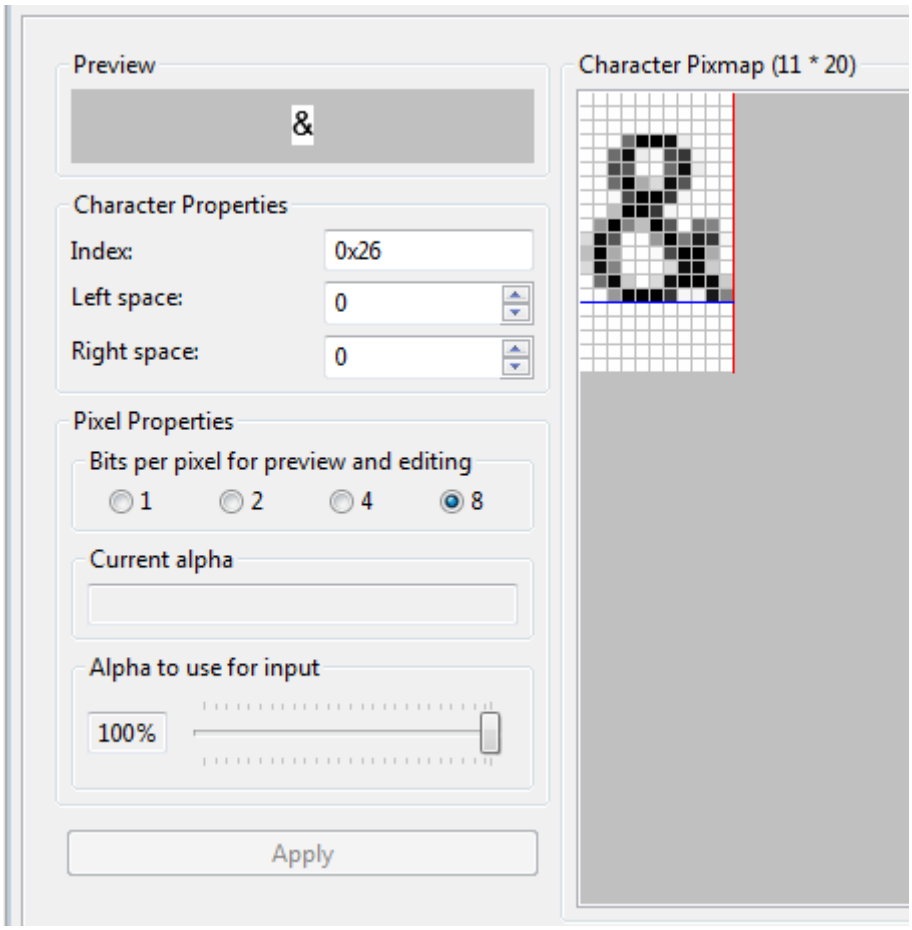


Fig. 56: Character Editor

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.

Working With 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.

Previewing a Font

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.

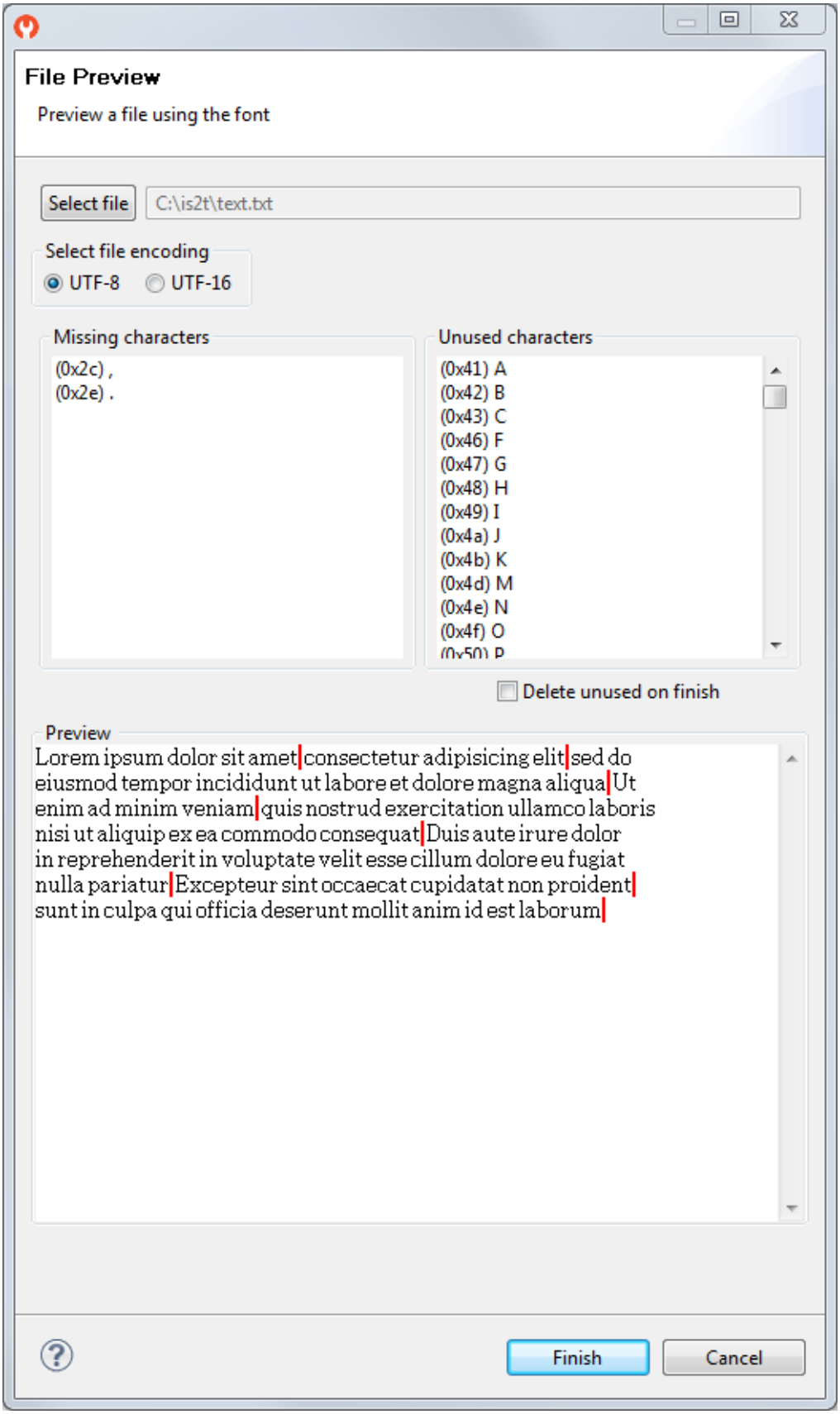


Fig. 57: Font Preview

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 a MicroEJ Font

A MicroEJ Font must be converted to a format which is specific to the targeted platform. 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 environment.

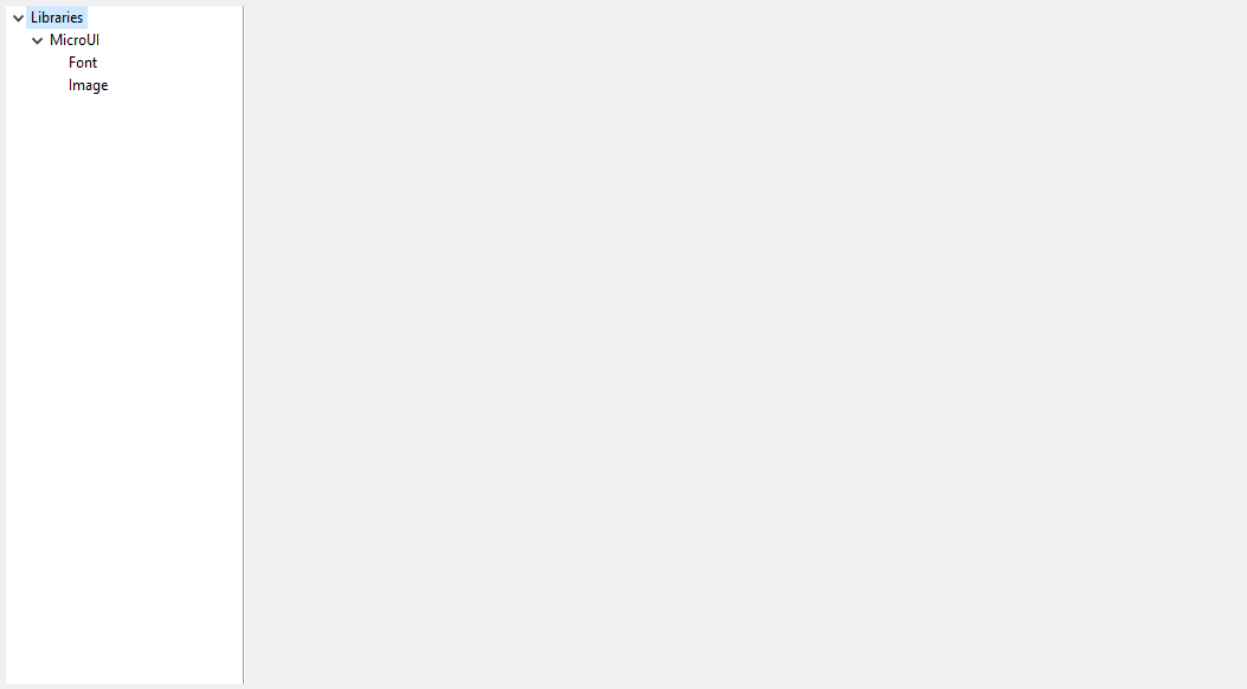
Use

Create a new `ejf` font file or open an existing one in order to open the Font Designer plugin.

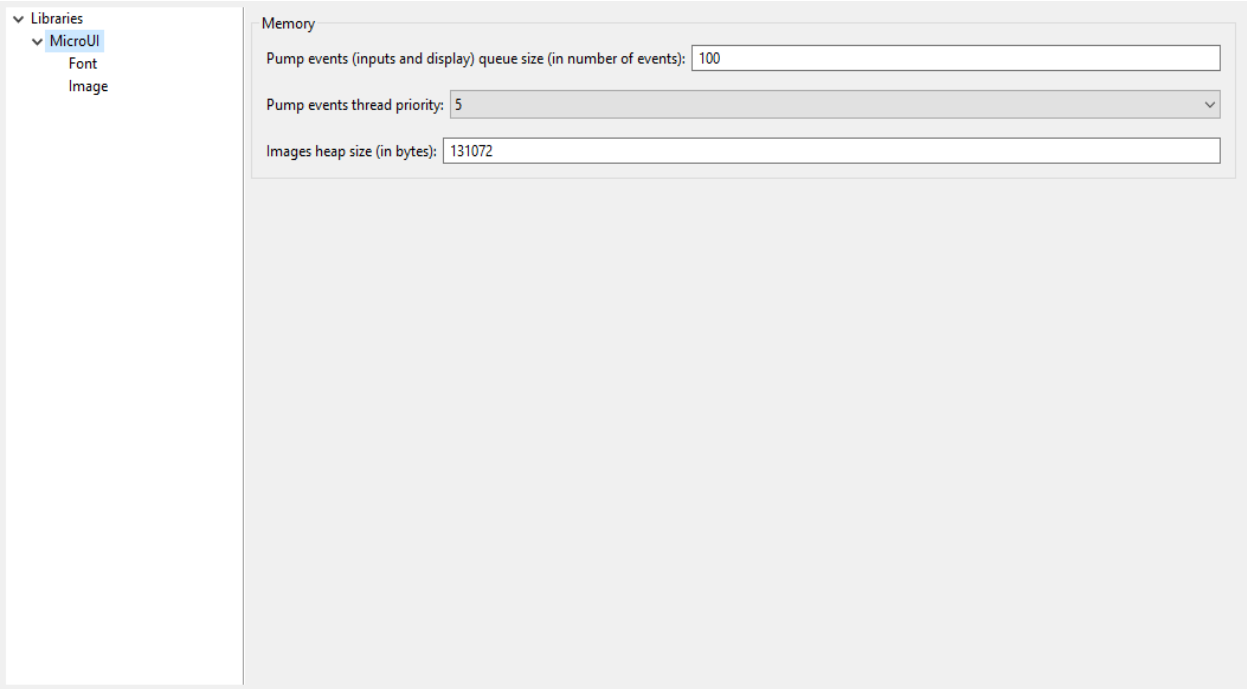
Application Options

MicroUI libraries and its tools provide a set of options. See *Application Options* to have more information about the application options.

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)

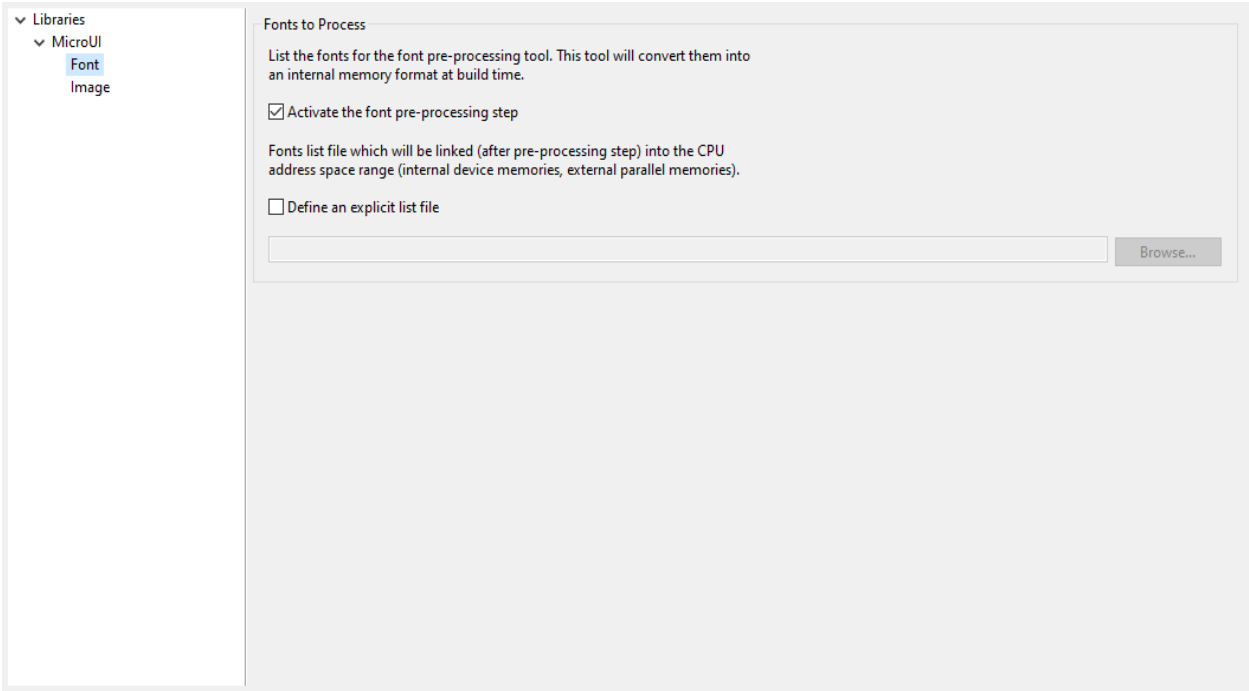
Option Name: `ej.microui.memory.imagesheap.size`

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):

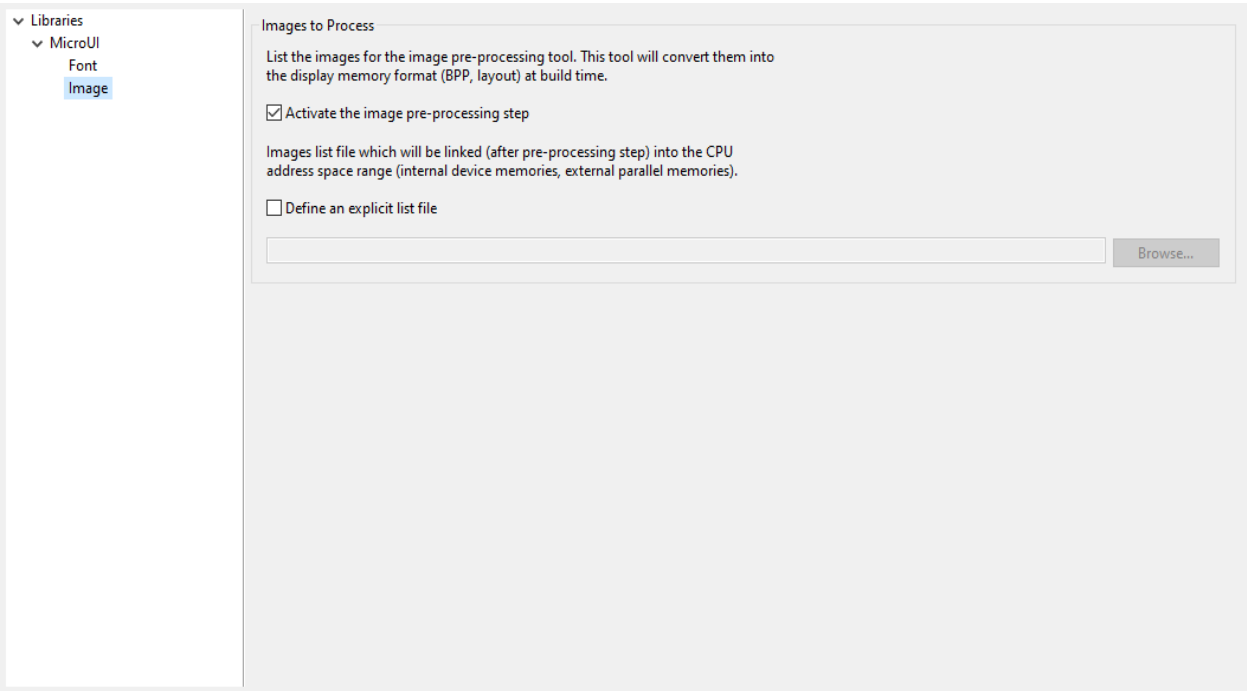
Option Name: `ej.microui.fontConverter.file`

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 logs several actions when traces are enabled. This chapter explains the traces identifiers. Some events data are described in next tables.

`[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 first event data number (0x0).
- DD is the second event data.
- EE is the second event data number (0x1).
- etc.

Table 16: 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%` (see Image Type), identifier is `%1%` and memory size is `%2%`.	
0xa (10)	Flush back buffer; position (`%0%` , `%1%`) size (`%2%` * `%3%`).	
0xb (11)	Flush done.	
0xc (12)	Start internal drawing operation `%0%` (see Drawing Type).	End of drawing `%0%` (see Drawing Type)
0xd (13)	Start drawing operation `%0%` (see Drawing Type).	End of drawing `%0%` (see Drawing Type)
0xe (14)	Unknown event.	
0xf (15)	Asynchronous drawing operation done.	
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%`.	
0x17 (23)	Replace event `%0%` by `%1%` at index `%2%`; queue length is `%3%`.	
0x18 (24)	Read event `%0%` at index `%1%`.	

Table 17: 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 18: Create Image

Event ID	Description
0x0 (0)	Create BufferedImage
0x1 (1)	Create Image from path
0x2 (2)	Create Image from InputStream

Table 19: 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 20: 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

Continued on next page

Table 20 – continued from previous page

Event ID	Description
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
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 character with rotation (bilinear)
0x39 (57)	Draw character with rotation (simple)
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 fli
0xc9 (201)	Draw image with rotation (simple)
0xca (202)	Draw image with rotation (bilinear)
0xcb (203)	Draw image with scalling (simple)
0xcc (204)	Draw image with scalling (bilinear)

The traces are *SystemView* compatible. The following text can be copied in a file called `SYSVIEW_MicroUI.txt` and copied in SystemView installation folder.

```
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
```

(continues on next page)

(continued from previous page)

```
NamedType UIEvent 14=REPAINT_CURRENT_DISPLAYABLE
NamedType UIEvent 15=KF_STOP_FEATURE

NamedType UINewImage 0=MUTABLE_IMAGE
NamedType UINewImage 1=IMAGE_FROM_PATH
NamedType UINewImage 2=IMAGE_FROM_INPUTSTREAM

NamedType UIImageData 0=NEW_IMAGE
NamedType UIImageData 1=LOAD_MICROEJ
NamedType UIImageData 2=NEW_ENCODED
NamedType UIImageData 3=NEW_MICROEJ_EXTERNAL
NamedType UIImageData 4=NEW_ENCODED_EXTERNAL
NamedType UIImageData 5=MEMORY_INPUTSTREAM
NamedType UIImageData 6=BYTEARRAY_INPUTSTREAM
NamedType UIImageData 7=GENERIC_INPUTSTREAM
NamedType UIImageData 8=LINK_IMAGE

NamedType GEDraw 1=WRITE_PIXEL
NamedType GEDraw 2=DRAW_LINE
NamedType GEDraw 3=DRAW_HORIZONTALLINE
NamedType GEDraw 4=DRAW_VERTICALLINE
NamedType GEDraw 5=DRAW_RECTANGLE
NamedType GEDraw 6=FILL_RECTANGLE
NamedType GEDraw 7=UNKNOWN
NamedType GEDraw 8=DRAW_ROUNDEDRECTANGLE
NamedType GEDraw 9=FILL_ROUNDEDRECTANGLE
NamedType GEDraw 10=DRAW_CIRCLEARC
NamedType GEDraw 11=FILL_CIRCLEARC
NamedType GEDraw 12=DRAW_ELLIPSEARC
NamedType GEDraw 13=FILL_ELLIPSEARC
NamedType GEDraw 14=DRAW_ELLIPSE
NamedType GEDraw 15=FILL_ELLIPSE
NamedType GEDraw 16=DRAW_CIRCLE
NamedType GEDraw 17=FILL_CIRCLE
NamedType GEDraw 18=DRAW_ARGB
NamedType GEDraw 19=DRAW_IMAGE

NamedType GEDraw 50=DRAW_POLYGON
NamedType GEDraw 51=FILL_POLYGON
NamedType GEDraw 52=GET_IMAGEARGB
NamedType GEDraw 53=DRAW_STRING
NamedType GEDraw 54=DRAW_DEFORMED_STRING
NamedType GEDraw 55=DRAW_IMAGE_DEFORMED
NamedType GEDraw 56=DRAW_CHAR_ROTATION_BILINEAR
NamedType GEDraw 57=DRAW_CHAR_ROTATION_SIMPLE
NamedType GEDraw 58=STRING_WIDTH
NamedType GEDraw 59=GET_PIXEL

NamedType GEDraw 100=DRAW_THICKFADEDPOINT
NamedType GEDraw 101=DRAW_THICKFADEDLINE
NamedType GEDraw 102=DRAW_THICKFADEDCIRCLE
NamedType GEDraw 103=DRAW_THICKFADEDCIRCLEARC
NamedType GEDraw 104=DRAW_THICKFADEDELLIPSE
NamedType GEDraw 105=DRAW_THICKLINE
NamedType GEDraw 106=DRAW_THICKCIRCLE
NamedType GEDraw 107=DRAW_THICKELLIPSE
```

(continues on next page)

(continued from previous page)

```
NamedType GEDraw 108=DRAW_THICKCIRCLEARC

NamedType GEDraw 200=DRAW_FLIPPEDIMAGE
NamedType GEDraw 201=DRAW_ROTATEDIMAGENEARESTNEIGHBOR
NamedType GEDraw 202=DRAW_ROTATEDIMAGEBILINEAR
NamedType GEDraw 203=DRAW_SCALEDIMAGENEARESTNEIGHBOR
NamedType GEDraw 204=DRAW_SCALEDIMAGEBILINEAR

#
# MicroUI
#
0        UI_EGEvent                  (MicroUI) Execute EventGenerator event %UIEvent (generatorID = %u,␣
↪data = %p)   | (MicroUI) EventGenerator event %UIEvent done
1        UI_DROPEvent        (MicroUI) Drop event %p
2        UI_InputEvent       (MicroUI) Execute native input event %UIEvent (generatorID = %u, event =
↪%p)    | (MicroUI) Native input event %UIEvent done
3        UI_DisplayEvent    (MicroUI) Execute display event %UIEvent (event = %p)                      ␣
↪             | (MicroUI) Display event %UIEvent done
4        UI_UserEvent        (MicroUI) Execute user event %p                                            ␣
↪                            | (MicroUI) User event %p done
5        UI_OpenImage       (MicroUI) Create %UINewImage                                                ␣
↪             | (MicroUI) Image created; id = %p
6        UI_ImageData       (MicroUI) %UINewImage ( %UIImageData ): id = %p; size = %d*%d

#
# MicroUI Graphics Engine
#
10       GE_FlushStart      (MicroUI GraphicalEngine) Flush back buffer (%u,%u) (%u*%u)
11       GE_FlushDone       (MicroUI GraphicalEngine) Flush done
12       GE_DrawInternal    (MicroUI GraphicalEngine) Drawing operation %GEDraw                   |␣
↪(MicroUI GraphicalEngine) Drawing operation %GEDraw done
13       GE_Draw            (MicroUI GraphicalEngine) Drawing operation %GEDraw                   |␣
↪(MicroUI GraphicalEngine) Drawing operation %GEDraw done
14       GE_Unknown                  (MicroUI GraphicalEngine) Unknown event
15       GE_GPUDrawDone     (MicroUI GraphicalEngine) Asynchronous drawing operation done

#
# MicroUI Input Engine
#
20       IE_InvalidEvent    (MicroUI Input Engine) Invalid event: %p
21       IE_QueueFull       (MicroUI Input Engine) Queue full, cannot add event %p
22       IE_AddEvent        (MicroUI Input Engine) Add event %p (index = %u / queue length = %u)
23       IE_ReplaceEvent    (MicroUI Input Engine) Replace event %p by %p (index = %u / queue length =
↪%u)
24       IE_ReadEvent       (MicroUI Input Engine) Read event %p (index %u)
```

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 following table:

Table 21: MicroUI Error Messages

Message ID	Description
1	Another `EventGenerator` cannot be added into the system pool (max 254).
0	[platform issue] Result of MicroUI static initialization step seems invalid. MicroUI cannot start. Please fix MicroUI static initialization step (see *Static Initialization*) and rebuild the platform.
-1	MicroUI is not started; call MicroUI.start() before using a MicroUI API.
-2	Unknown event generator class name.
-3	Deadlock. Cannot wait for an event in the same thread that runs events. `Display.waitFlushCompleted()` must not be called in the MicroUI thread (for example in `render` method).
-4	Resource's path must be relative to the classpath (start with '/') or resource is not available.
-5	The resource data cannot be read for unknown reason.
-6	The resource has been closed and cannot be used anymore.
-7	Out of memory. Not enough memory to allocate the `Image` 's buffer. Try to close some useless images and retry opening the new image, or increase the size of the MicroUI images heap.
-8	The platform cannot decode this kind of image, because the required runtime image decoder is not available in the platform.
-9	This exception is thrown when the FIFO of the internal MicroUI thread is full. In this case, no more event (such as `requestRender` , input events, etc.) can be added into it. Most of time this error occurs when: • There is a user thread which performs too many calls to the method `requestRender` without waiting for the end of the previous drawing. • Too many input events are pushed from an input driver to the MicroUI thread (for example some touch events).
-10	There is no display on the platform.
-11	There is no font (platform and application).

3.12.2 MWT (Micro Widget Toolkit)

Introduction

MWT is a toolkit that simplifies the creation and use of graphical user interface widgets on a pixel-based display.

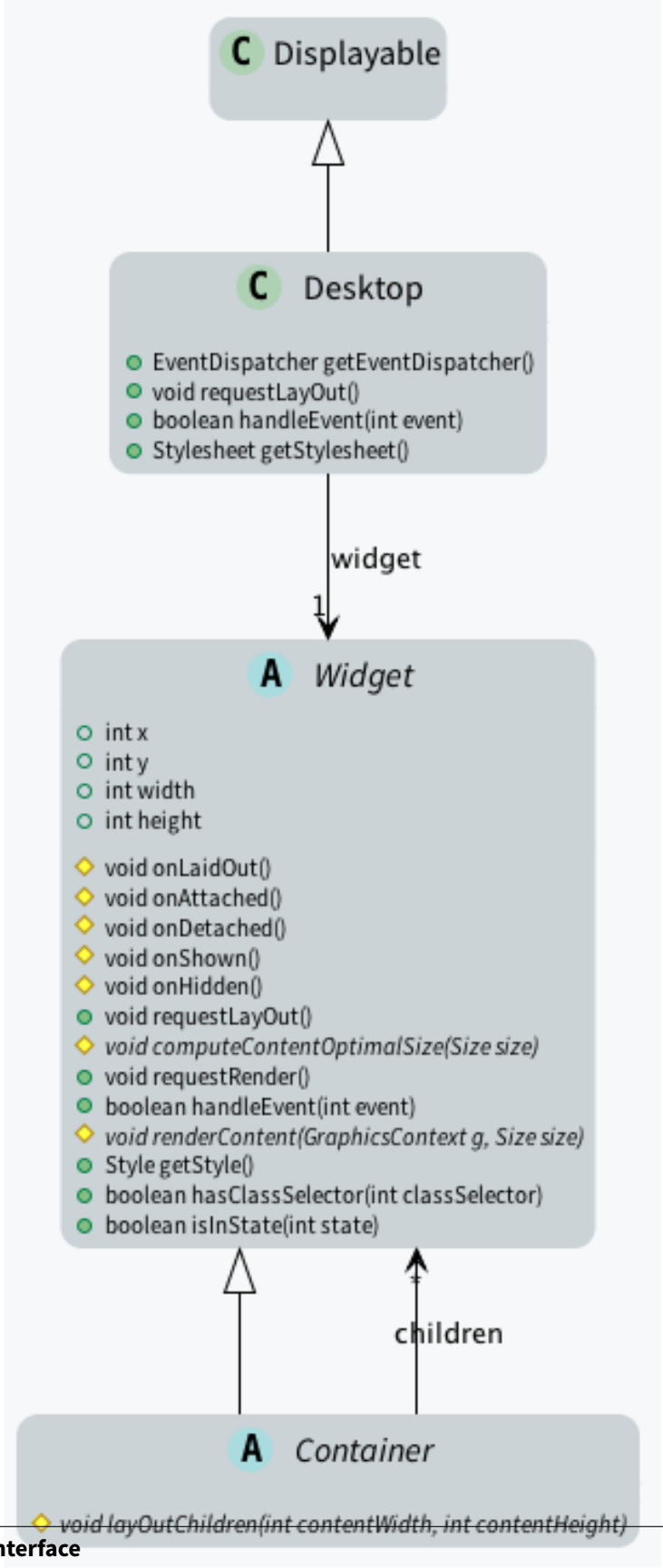
The aim of this library is to be sufficient to create complex applications with a minimal framework. It provides the main concepts without managing particular needs. Specific needs can be met by a MWT expert by creating new widgets, adding more complex concepts, etc. The flexibility of the MWT open framework allows the selection of only what is necessary for the application in order to guarantee lightweight applications and fast execution.

To use the MWT library, add the following line to a *module description file*:

```
<dependency org="ej.library.ui" name="mwt" rev="3.1.0"/>
```

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()`.

Lay Out

All widgets are laid out at once during the lay out process. This process can be started by `Desktop.requestLayOut()`, `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 position and size for each widget.

Once the position and size of a widget is set, the widget is notified by a call to `onLaidOut()`.

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.

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

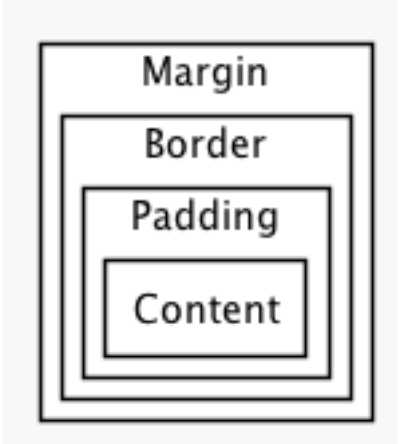
The horizontal and vertical alignments are used to position the content of the widget within its bounds.

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));
```

(continues on next page)

(continued from previous page)

```
desktop.setStylesheet(stylesheet);
```

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.

Desktop and widget states

Desktop and widgets pass through different states. Once created, they can be attached, then they can be shown.

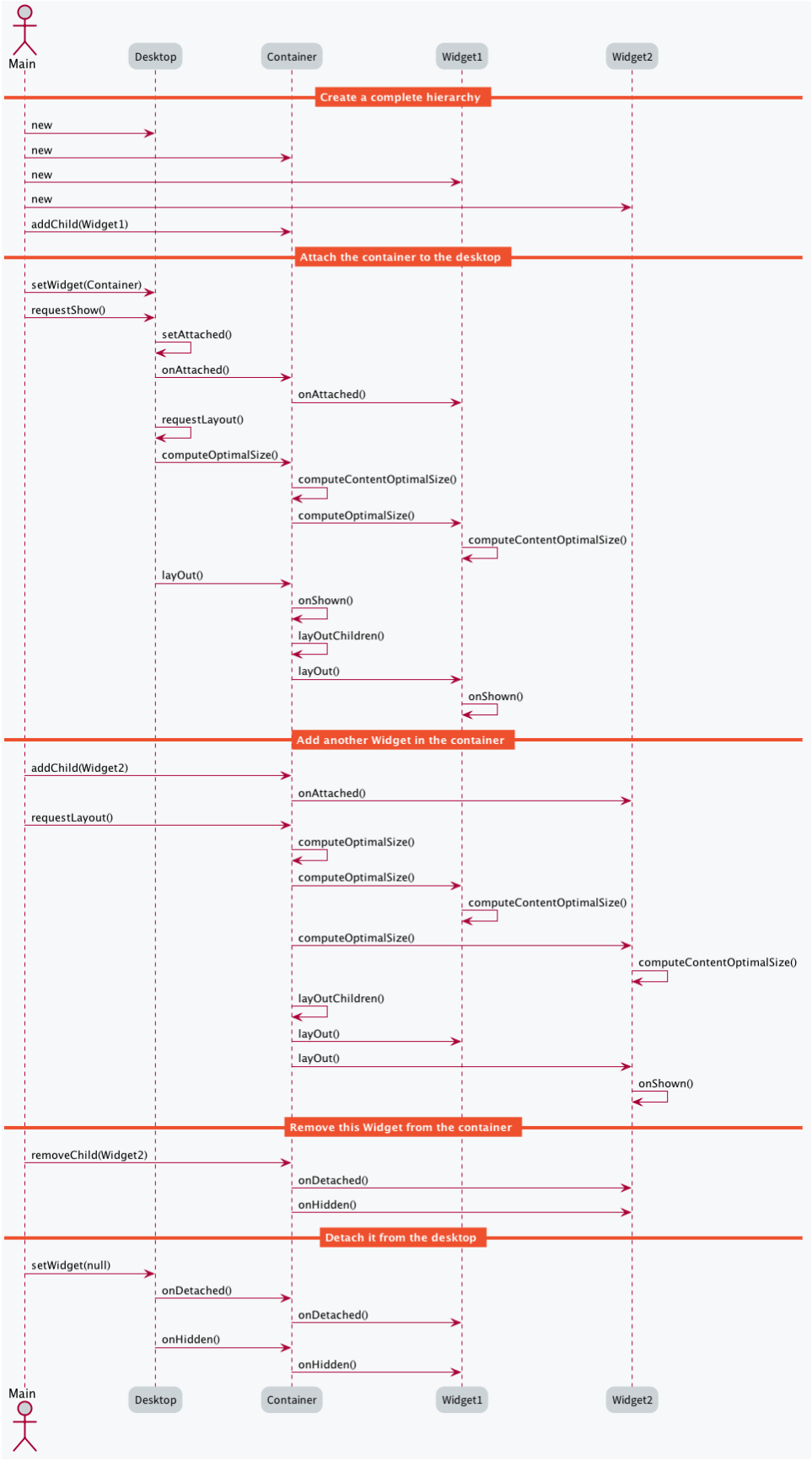
A desktop is attached automatically as soon as it is shown on the display. It can also be attached manually by calling `Desktop.setAttached()` . It could be used to render the desktop (and its widgets) on an image for example.

A widget is considered as attached when it is contained by a desktop that is attached.

In the same way, by default, a widget is shown when its desktop is shown. But for optimization purpose, a container can control when its children are shown or hidden. A typical use case is when the widgets are moved outside the display.

Once a widget is attached, it means that it is ready to be shown (for instance, the necessary resources are allocated). In other words, once attached a widget is ready to be rendered (on an image or on the display).

Once a widget is shown, it means that it is intended to be rendered on the display. While shown, it may start a periodic refresh or an animation.


Main
Desktop
Container
Widget1
Widget2
Create a complete hierarchy
new
new
new
new
addChild(Widget1)
Attach the container to the desktop
setWidget(Container)
requestShow()
setAttached()
onAttached()
onAttached()
requestLayout()
computeOptimalSize()
computeContentOptimalSize()
computeOptimalSize()
computeContentOptimalSize()
layOut()
onShown()
layOutChildren()
layOut()
onShown()
Add another Widget in the container
addChild(Widget2)
onAttached()
requestLayout()
computeOptimalSize()
computeOptimalSize()
computeContentOptimalSize()
computeOptimalSize()
computeContentOptimalSize()
layOutChildren()
layOut()
layOut()
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

The following sections will present several ways to customize and extend the framework to better fit your needs.

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 and Examples* 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 should be big enough to contain all the drawings of the widget.

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 a label:

```
@Override
protected void computeContentOptimalSize(Size size) {
        Font font = getStyle().getFont();
        int width = font.stringWidth(this.text);
        int height = font.getHeight();
        size.setSize(width, height);
}
```

Rendering the content of the widget

The `renderContent()` method is called by the MWT framework in order to render the content 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 renders the content of a label:

```
@Override
protected void renderContent(GraphicsContext g, int contentWidth, int contentHeight) {
        Style style = getStyle();
        g.setColor(style.getColor());
        Painter.drawString(g, style.getFont(), this.text, 0, 0);
}
```

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;
}
```

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()`

For example, the `onAttached()` 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();
}
```

For example, the `onShown()` may be overridden to start an animation:

```
@Override
protected void onShown() {
        Animator animator = ServiceFactory.getService(Animator.class);
        animator.startAnimation(this);
}
```

Likewise, the `onHidden()` method may be overridden to stop an animation:

```
@Override
protected void onHidden() {
        Animator animator = ServiceFactory.getService(Animator.class);
```

(continues on next page)

(continued from previous page)

```
        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 passed in the constructor of the widget or be fetched from a service provider. 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 final Animator animator;

        private long startTime;
        private long elapsedTime;

        public MyAnimatedWidget(Animator animator) {
                this.animator = animator;
        }

        @Override
        protected void onShown() {
                this.startTime = System.currentTimeMillis();
                this.animator.startAnimation(this);
        }

        @Override
        protected void onHidden() {
                this.animator.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 if necessary. 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 currentTimeMillis) {
        // update widget state
        this.elapsedTime = currentTimeMillis - 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 `System.currentTimeMillis()` 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);
}
```

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, Size size)` 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, Size size) {
        g.setColor(Colors.WHITE);
        Painter.fillRectangle(g, 0, 0, size.getWidth(), size.getHeight());
        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);
```

3.12.3 Widgets and Examples

Widget library

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.

The widget library does not provide any example. However, the widget demo provides examples for these widgets.

Source

To use the widgets provided by the widget library, add the following line to a *module description file*:

```
<dependency org="ej.library.ui" name="widget" rev="3.1.0"/>
```

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.

Widget demo

The widget demo 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 demo, clone the following GitHub repository: https://github.com/MicroEJ/Demo-Widget. You can then import the `com.microej.demo.widget` project into your workspace to see the source of the widgets and their associated examples.

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

Widgets:

- `Checkbox` : displays a text and a square which can be checked or unchecked.
- `RadioButton` : displays a text and a circle which can be checked or unchecked.
- `ProgressBar` : displays an animated bar indicating that the user should wait for an estimated amount of time.
- `IndeterminateProgressBar` : displays an animated bar indicating that the user should wait for an indeterminate amount of time.
- `Toggle` : displays a text and a switch that can be checked or unchecked.

Containers:

- `Split` : lays out two children horizontally or vertically, by giving each child a portion of the available space.
- `ScrollableList` : lays out its widgets the same way as a regular list, but provides an optimization when added to a scroll.

MWT examples

The MWT Examples repository provides various examples which extend or customize the MWT framework.

Source

To run the examples and read the source code of these examples, clone the following GitHub repository: https://github.com/MicroEJ/ExampleJava-MWT. You can then import the multiple project into your workspace to see the source of each example and to run it on Simulator or on your board.

Provided examples

- `com.microej.example.mwt.attribute` : shows how to customize the style of widgets using attributes selectors, similar to CSS.
- `com.microej.example.mwt.focus` : shows how to introduce focus management in your project.
- `com.microej.example.mwt.lazystylesheet` : shows how to use a lazy stylesheet rather than the default stylesheet implementation.
- `com.microej.example.mwt.mvc` : shows how to develop responsive widgets using a MVC design pattern and how to display a cursor image representing the pointer.

CHAPTER

FOUR

PLATFORM DEVELOPER GUIDE

4.1 Introduction

4.1.1 Scope

This document explains how the core features of MicroEJ Architecture are accessed, configured and used by the MicroEJ Platform builder. It describes the process for creating and augmenting a MicroEJ Architecture. This document is concise, but attempts to be exact and complete. Semantics of implemented Foundation Libraries are described in their respective specifications. This document includes an outline of the required low level drivers (LLAPI) for porting the MicroEJ Architectures to different real-time operating systems (RTOS).

MicroEJ Architecture is state-of-the-art, with embedded MicroEJ runtimes for MCUs. They also provide simulated runtimes that execute on workstations to allow software development on "virtual hardware."

4.1.2 Intended Audience

The audience for this document is software engineers who need to understand how to create and configure a MicroEJ Platform using the MicroEJ Platform builder. This document also explains how a MicroEJ Application can interoperate with C code on the target, and the details of the MicroEJ Architecture modules, including their APIs, error codes and options.

4.1.3 MicroEJ Architecture Modules Overview

MicroEJ Architecture features the MicroEJ Core Engine: a tiny and fast runtime associated with a Garbage Collector. It provides four built-in Foundation Libraries :

- *[BON]*
- EDC
- *[SNI]*
- *[SP]*

The following figure shows the components involved.

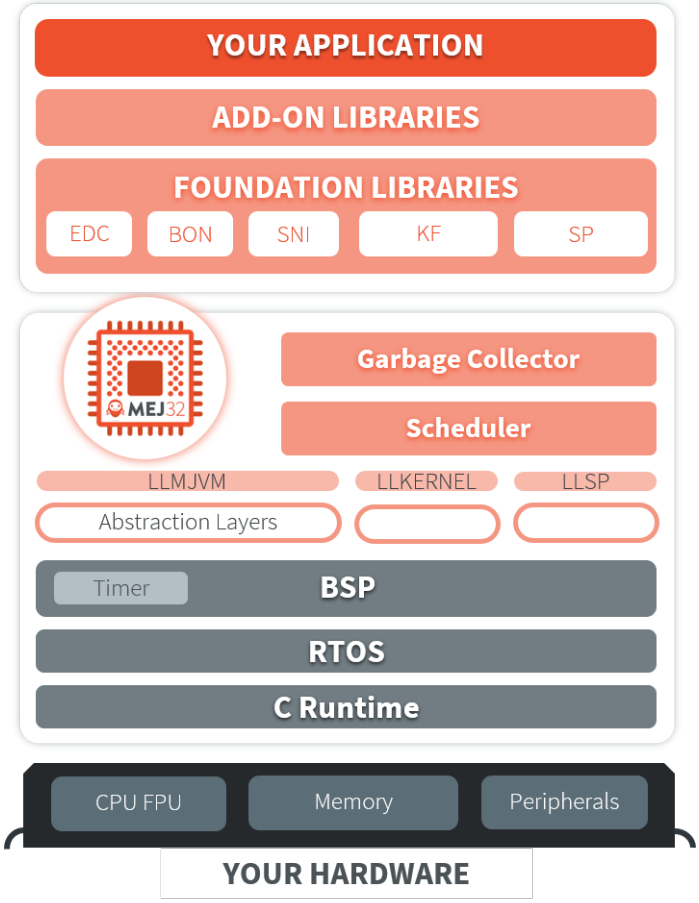


Fig. 1: MicroEJ Architecture Runtime Modules: Tools, Libraries and APIs

Three APIs allow the device architecture runtime to link with (and port to) external code, such as any kind of RTOS or legacy C libraries. These three APIs are

- Simple Native Interface (*[SNI]*)
- Low Level MicroEJ Core Engine (LLMJVM)
- Low Level Shielded Plug (LLSP)

MicroEJ Architecture features additional Foundation Libraries and modules to extend the kernel:

- serial communication,
- UI extension (User Interface)
- networking
- file system
- etc.

Each additional module is optional and selected on demand during the MicroEJ Platform configuration.

4.2 MicroEJ Platform

4.2.1 Process Overview

This section summarizes the steps required to build a MicroEJ Platform and obtain a binary file to deploy on a board.

The following figure shows the overall process. The first three steps are performed within the MicroEJ Platform builder. The remaining steps are performed within the C IDE.

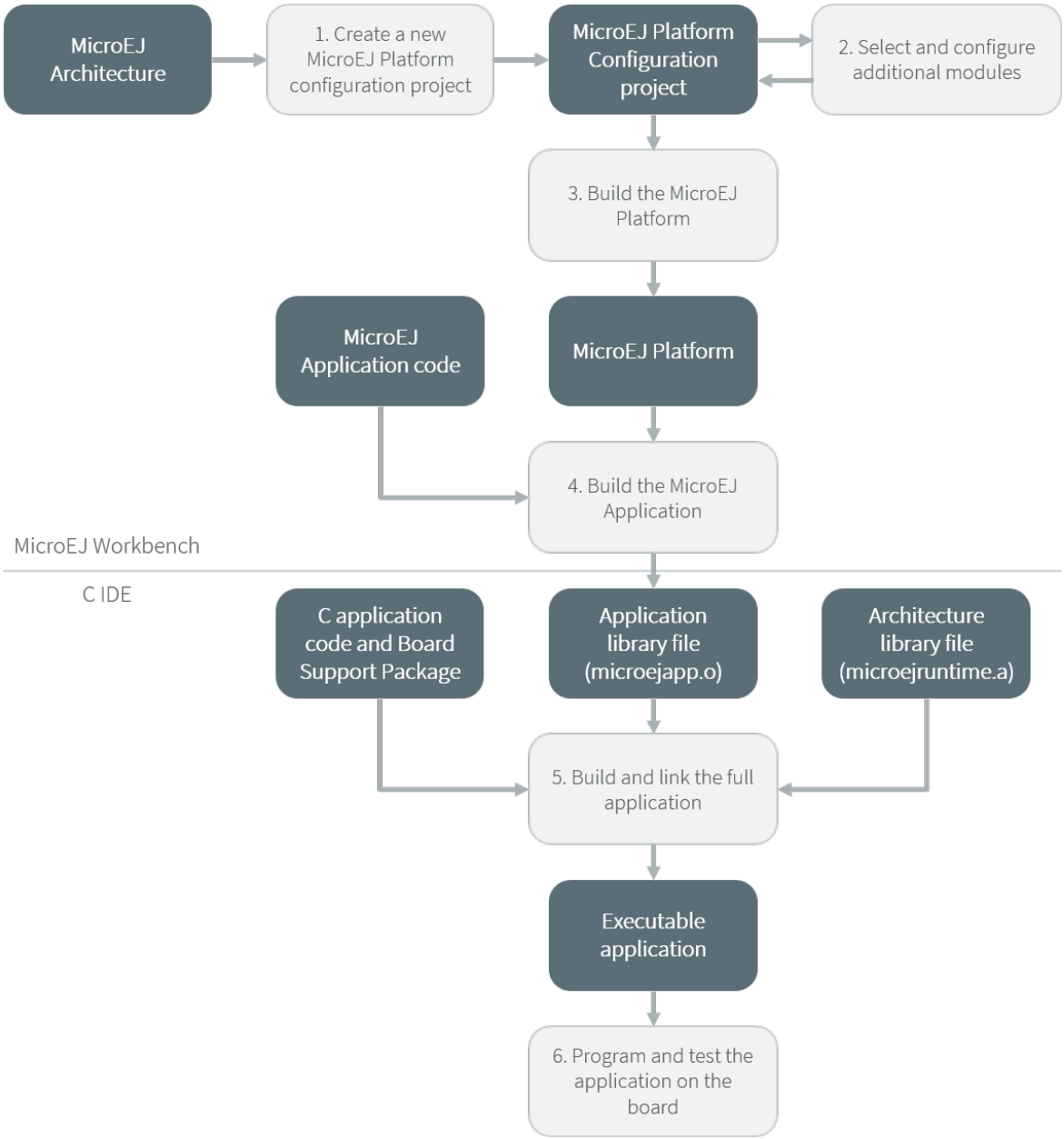


Fig. 2: Overall Process

The steps are as follow:

1. Create a new MicroEJ Platform configuration project. This project describes the MicroEJ Platform to build (MicroEJ Architecture, metadata, etc.).
2. Select which modules provided by the MicroEJ Architecture will be installed in the MicroEJ Platform.
3. Build the MicroEJ Platform according to the choices made in steps 1 and 2.
4. Compile a MicroEJ Application against the MicroEJ Platform in order to obtain an application file to link in the BSP.
5. Compile the BSP and link it with the MicroEJ Application that was built previously in step 4 to produce a MicroEJ Firmware.
6. Final step: Deploy MicroEJ Firmware (i.e. the binary application) onto a board.

4.2.2 Concepts

MicroEJ Platform

A MicroEJ Platform includes development tools and a runtime environment.

The runtime environment consists of:

- A MicroEJ Core Engine.
- Some Foundation Libraries.
- Some C libraries.

The development tools are composed of:

- Java APIs to compile MicroEJ Application code.
- Documentation: this guide, library specifications, etc.
- Tools for development and compilation.
- Launch scripts to run the simulation or build the binary file.
- Eclipse plugins.

MicroEJ Platform Configuration

A MicroEJ Platform is described by a `.platform` file. This file is usually called `[name].platform`, and is stored at the root of a MicroEJ Platform configuration project called `[name]-configuration`.

The configuration file is recognized by the MicroEJ Platform builder. The MicroEJ Platform builder offers a visualization with two tabs:

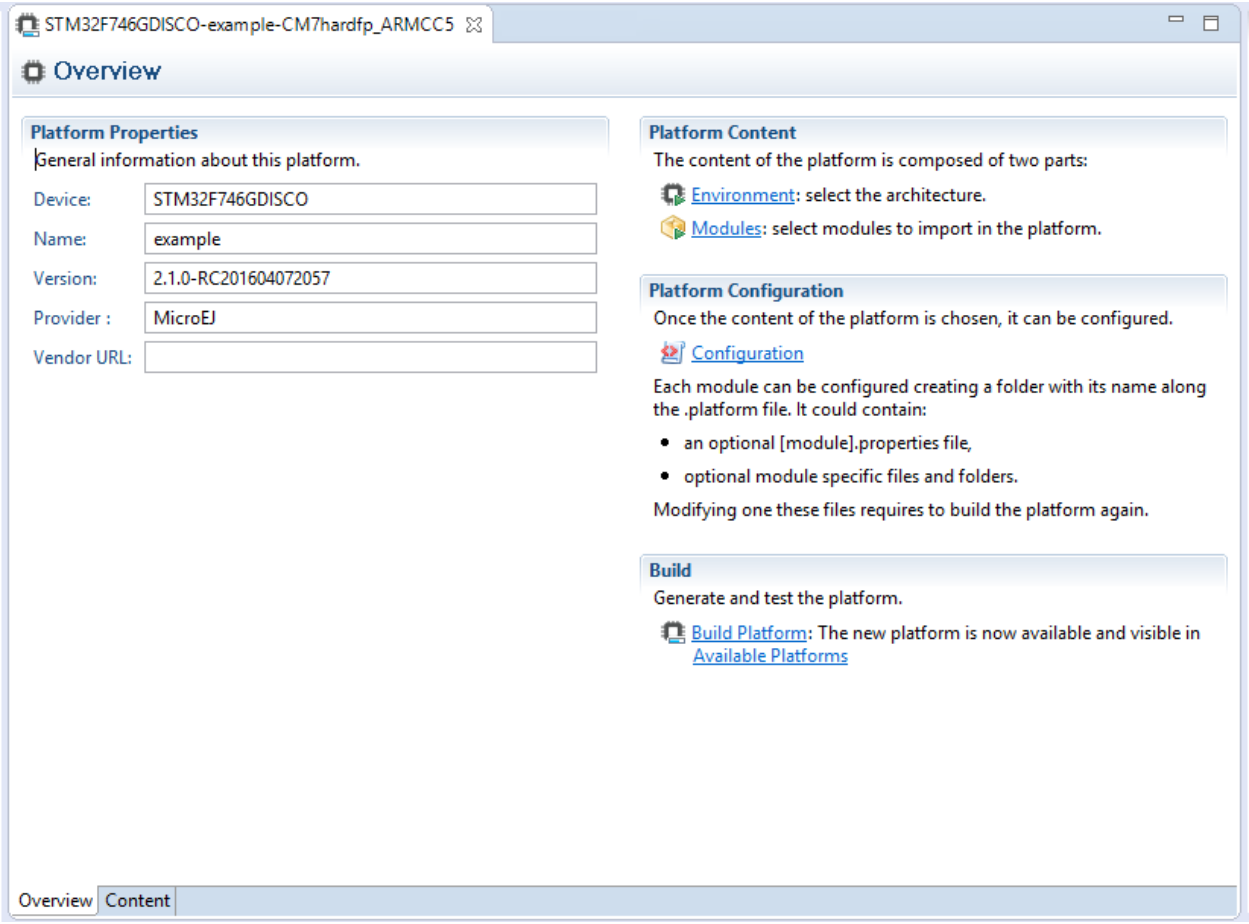


Fig. 3: MicroEJ Platform Configuration Overview Tab

This tab groups the basic platform information used to identify it: its name, its version, etc. These tags can be updated at any time.

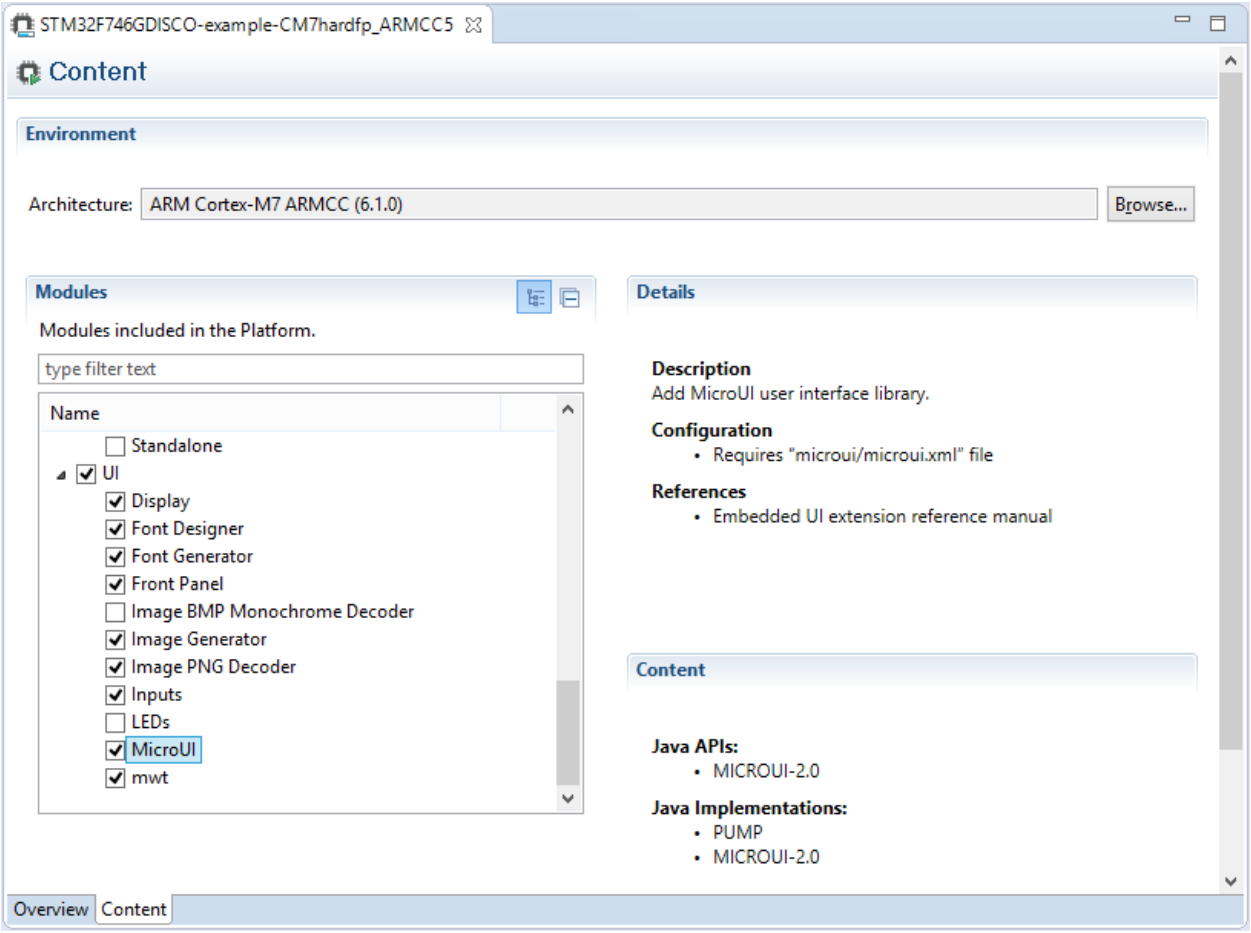


Fig. 4: MicroEJ Platform Configuration Content Tab

This tab shows all additional modules (see *Modules*) which can be installed into the platform in order to augment its features. The modules are sorted by groups and by functionality. When a module is checked, it will be installed into the platform during the platform creation.

Modules

The primary mechanism for augmenting the capabilities of a *MicroEJ Platform* is to add modules to it.

A MicroEJ module is a group of related files (Foundation Libraries, scripts, link files, C libraries, Simulator, tools, etc.) that together provide all or part of a platform capability. Generally, these files serve a common purpose. For example, providing an API, or providing a library implementation with its associated tools.

The list of modules is in the second tab of the platform configuration tab. A module may require a configuration step to be installed into the platform. The Modules Detail view indicates if a configuration file is required.

Low Level API Pattern

Principle

Each time the user must supply C code that connects a platform component to the target, a *Low Level API* is defined. There is a standard pattern for the 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.

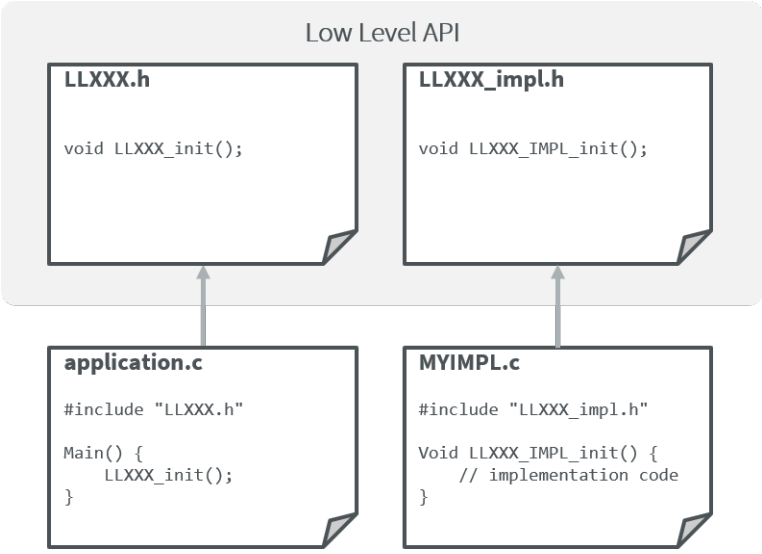


Fig. 5: Low Level API Pattern (single implementation)

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.

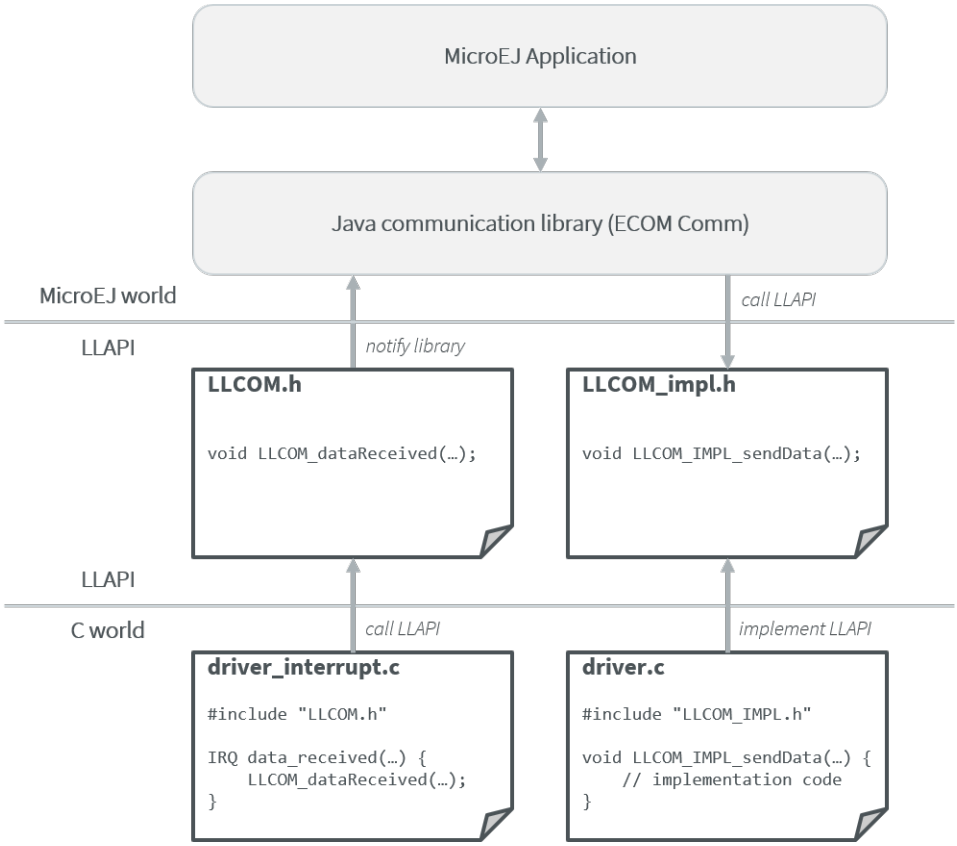


Fig. 6: Low Level API Example

Multiple Implementations and Instances

When a Low Level API allows multiple implementations, each implementation must have a unique name. At runtime 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.

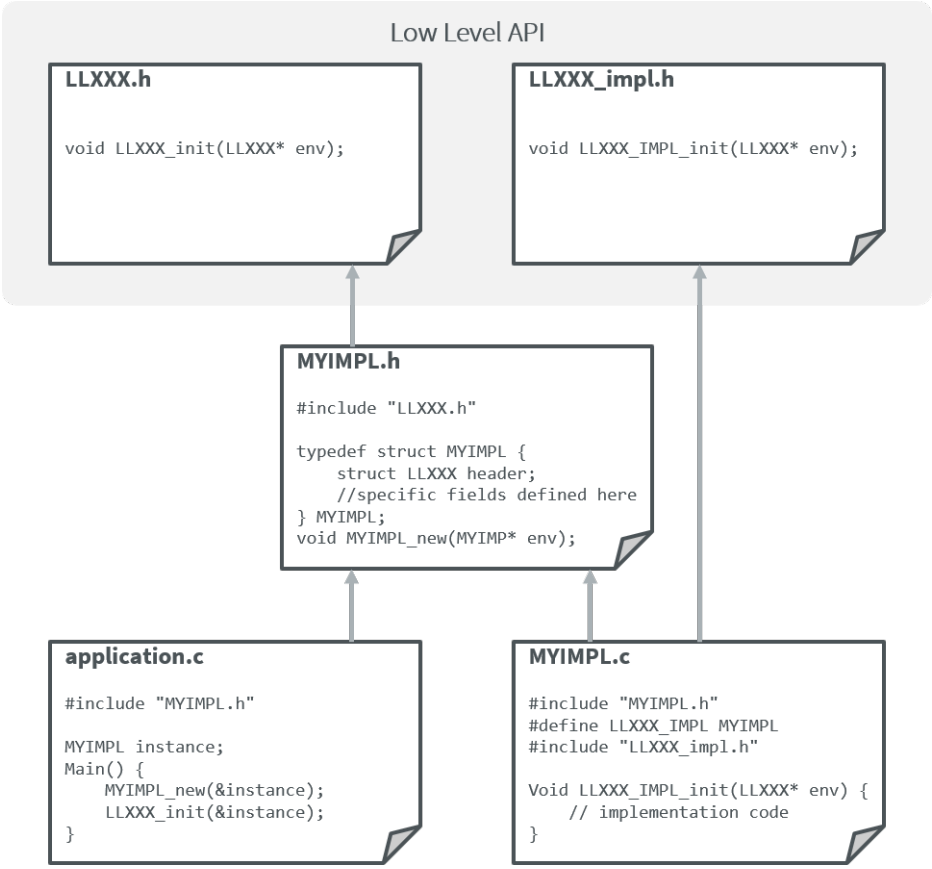


Fig. 7: Low Level API Pattern (multiple implementations/instances)

The `#define` statement in `MYIMPL.c` specifies the name given to this implementation.

4.2.3 MicroEJ Platform Creation

This section describes the steps to create a new MicroEJ Platform in MicroEJ SDK, and options to connect it to an external Board Support Package (BSP) as well as a third-party C toolchain.

MicroEJ SDK must be started on a new empty *workspace*.

MicroEJ Architecture Import

The first step is to choose and import a *MicroEJ Architecture*. MicroEJ Architectures for most common microcontroller instructions sets and compilers can be downloaded from https://repository.microej.com/architectures/[1].

MicroEJ Architecture files ends with the `.xpf` extension, and are classified using the following folder 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`).

[1] MicroEJ Architectures for production can be retrieved from the license server.

- `VERSION` : module version (e.g. `7.12.0`).
- `UID` : Architecture unique ID (e.g. `flopi4G25`).
- `USAGE` = `eval` for evaluation Architectures, `prod` for production Architectures.

For example, MicroEJ Architecture versions for Arm® Cortex®-M4 microcontrollers compiled with GNU CC toolchain is available at https://repository.microej.com/architectures/com/microej/architecture/CM4/CM4hardfp_GCC48/flopi4G25/.

Once you downloaded a MicroEJ Architecture file, proceed with the following steps to import it in MicroEJ SDK:

- Select File > Import > MicroEJ > Architectures .
- Browse an `.xpf` file or a folder that contains one or more an `.xpf` files.
- Check the I agree and accept the above terms and conditions... box to accept the license.
- Click on Finish button.

Note: A MicroEJ Architecture can be imported using MicroEJ Module Manager, by adding the following line in a *module description file*:

```
<dependency org="com.microej.architecture.[ISA].[TOOLCHAIN]" name="[UID]" rev="[VERSION]" conf="default
↪">
  <artifact name="[UID]" m:classifier="[USAGE]" ext="xpf"/>
</dependency>
```

MicroEJ Platform Configuration

The next step is to create a MicroEJ Platform configuration:

- Select File > New > MicroEJ Platform Project... .
- Click on Next button. The Configure Target Architecture page allows to select the MicroEJ Architecture that contains a minimal MicroEJ Platform and a set of compatible modules targeting a processor architecture and a compilation toolchain. This environment can be changed later.
  - Click on Browse... button to select one of the installed MicroEJ Architecture.
  - Check the Create from a platform reference implementation box to use one of the available implementation. Uncheck it if you want to provide your own implementation or if no reference implementation is available.
- Click on Next button. The `Configure platform properties` page contains the identification of the MicroEJ Platform to create. Most fields are mandatory, you should therefore set them. Note that their values can be modified later on.
- Click on Finish button. A new project `[device]-[name]-[toolchain]` is being created containing a `[name].platform` file. The Platform Editor shall then open.
- Install Platform Configuration Additions. Files within the `content` folder have to be copied to the configuration project folder, by following instructions described at https://github.com/MicroEJ/PlatformQualificationTools/blob/master/framework/platform/README.rst.

You should get a MicroEJ Platform configuration project that looks like:

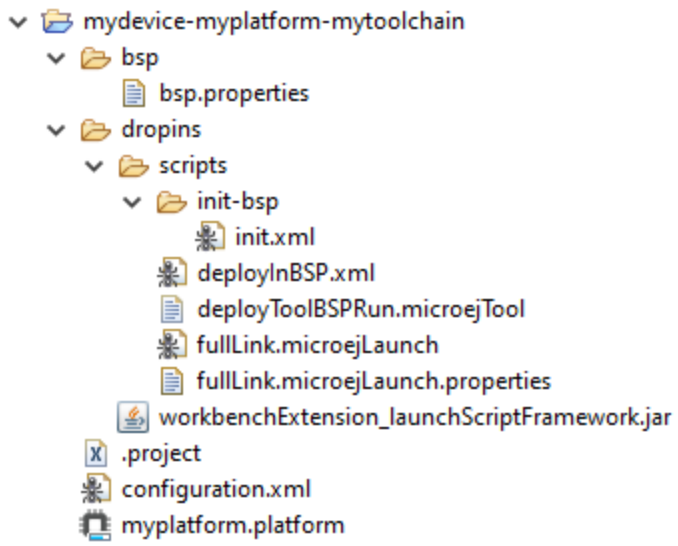


Fig. 8: MicroEJ Platform Configuration Project Skeleton

Groups / Modules Selection

From the Platform Editor, select the Content tab to access the Platform modules selection. Modules can be selected/deselected from the Modules frame.

Modules are organized into groups. When a group is selected, by default, all its modules are selected. To view the modules making up a group, click on the Show/Hide modules 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.

The description and contents of an item (group or module) are displayed beside the list on item selection.

All the checked modules will be installed in the Platform.

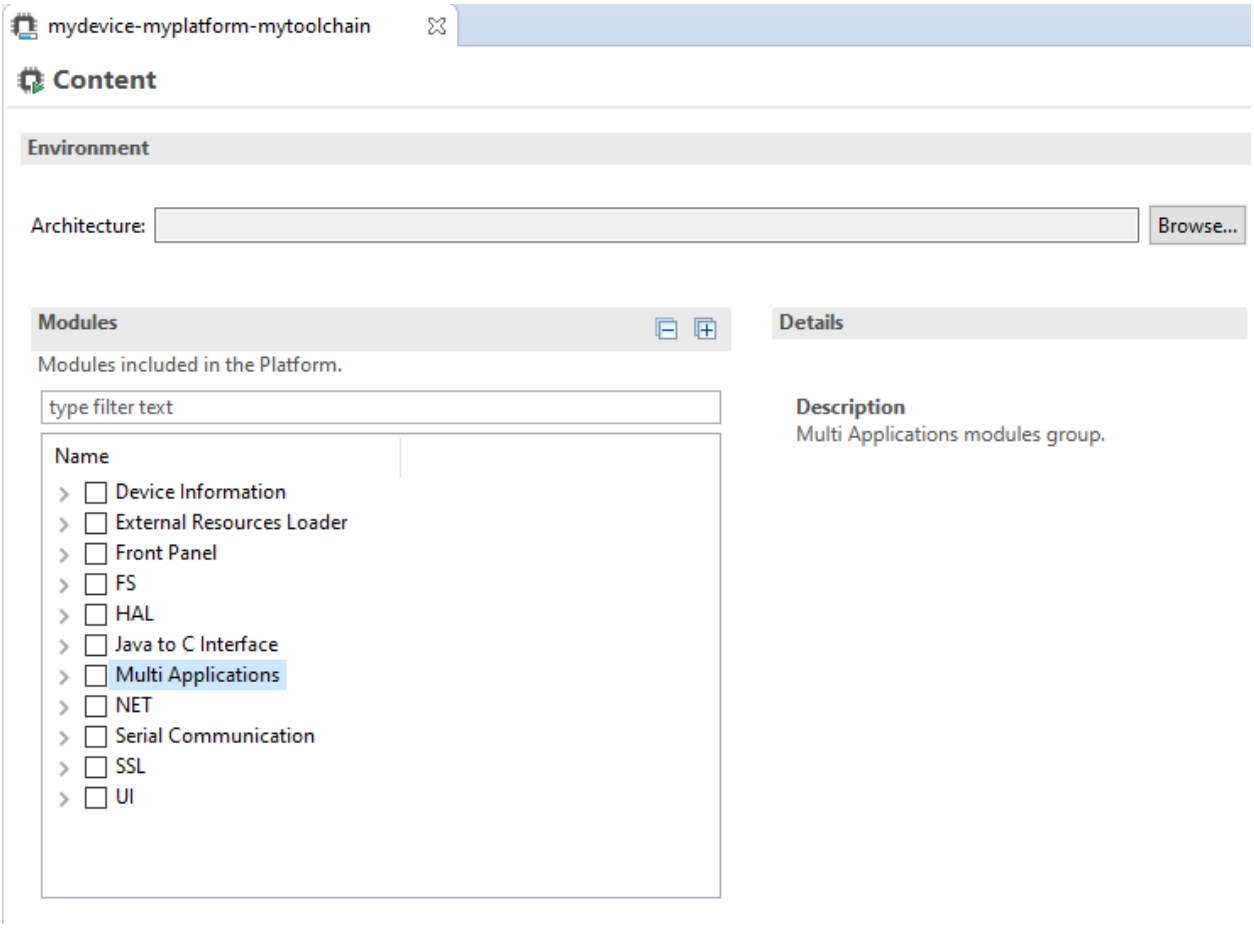


Fig. 9: MicroEJ Platform Configuration Modules Selection

Modules Customization

Each selected module can be customized by creating a [module] folder named after the module beside the `[name].platform` definition. It may contain:

- An optional [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.
- Optional module specific files and folders.

Modifying one of these files requires to build the Platform again.

Platform Customization

Platforms can be customized by creating a `configuration.xml` Ant file beside the `[name].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.

Here is a template for a `configuration.xml` Ant file:

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="configuration">

    <!--
            Define "project.dir" property that references the directory
            where this file is located.
    -->
    <dirname property="project.dir" file="${ant.file.configuration}"/>

</project>
```

Configuration project (the project which contains the `[name].platform` file) can contain an optional `dropins` folder. The contents of this folder will be copied integrally into the final Platform. This feature allows to add some additional libraries, tools etc. into the Platform.

The dropins folder organization should respect the final Platform files and folders organization. For instance, the tools are located in the sub-folder `tools`. Launch a Platform build without the dropins folder to see how the Platform files and folders organization is. Then fill the dropins folder with additional features and build again the Platform to obtain an advanced Platform.

The dropins folder files are kept in priority. If one file has the same path and name as another file already installed into the Platform, the dropins folder file will be kept.

Modifying one of these files requires to build the Platform again.

BSP Connection

Principle

Using a MicroEJ Platform, the user can compile a MicroEJ Application on that Platform. The result of this compilation is a `microejapp.o` file.

This file has to be linked with the MicroEJ Platform runtime file (`microejruntime.a`) and a third-party C project, called the Board Support Package (BSP) , to obtain the final binary file (MicroEJ Firmware). 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:

- MicroEJ Application file (`microejapp.o`).
- MicroEJ Platform runtime file (`microejruntime.a` , also available in the Platform `lib` folder).
- MicroEJ Platform header files (`*.h` , also available in the Platform `include` folder).
- BSP project *build script* file (`build.bat` or `build.sh`).

Once the MicroEJ 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 final executable file (`application.out`).

Note: The final build stage to produce the executable file can be done outside of MicroEJ SDK, and thus the BSP connection configuration is optional.

BSP connection configuration is only required in the following cases:

- Use MicroEJ SDK to produce the final executable file of a Mono-Sandbox Firmware (recommended).
- Use MicroEJ SDK to run a *MicroEJ Testsuite* on device.
- Build a Multi-Sandbox Firmware.

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 MicroEJ Platform level or at MicroEJ Application level (or a mix of both).

The 3 most common integration cases are:

- Case 1: No BSP connection

  The MicroEJ Platform does not know the BSP at all.

  BSP connection can be configured when building the MicroEJ Application (absolute locations).

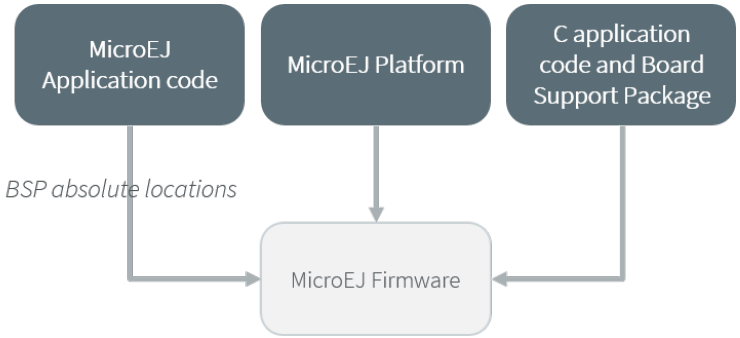


Fig. 10: MicroEJ Platform with no BSP connection

  This case is recommended when:

  - the MicroEJ Firmware is built outside MicroEJ SDK.
  - the same MicroEJ Platform is intended to be reused on multiple BSP projects which do not share the same structure.

- Case 2: Partial BSP connection

  The MicroEJ Platform knows how the BSP is structured.

  BSP connection is configured when building the MicroEJ Platform (relative locations within the BSP), and the BSP root location is configured when building the MicroEJ Application (absolute directory).

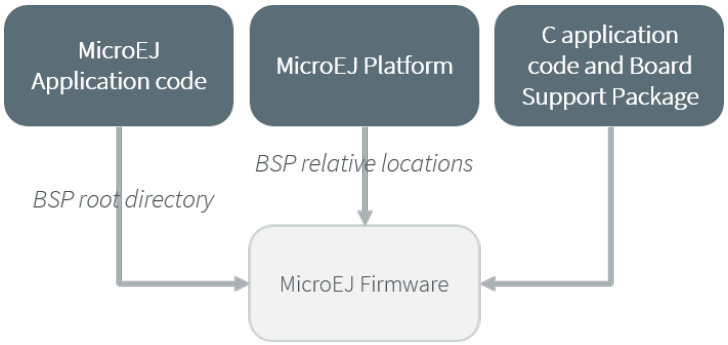


Fig. 11: MicroEJ Platform with partial BSP connection

  This case is recommended when:

  - the MicroEJ Platform is used to build one MicroEJ Application on top of one BSP.

  - the Application and BSP are slightly coupled, thus making a change in the BSP just require to build the firmware again.

- Case 3: Full BSP connection

  The MicroEJ Platform includes the BSP.

  BSP connection is configured when building MicroEJ Platform (relative locations within the BSP), as well as the BSP root location (absolute directory). No BSP connection configuration is required when building the MicroEJ Application.

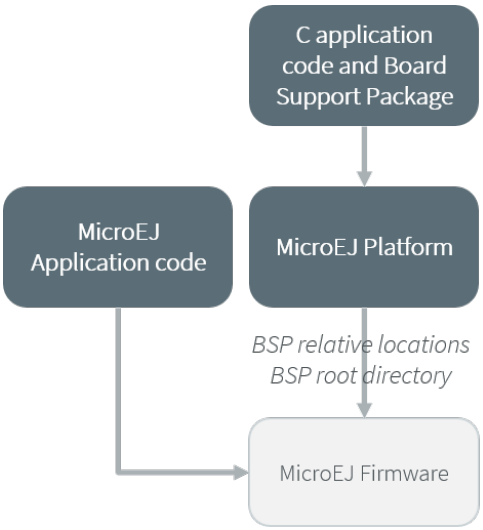


Fig. 12: MicroEJ Platform with full BSP connection

  This case is recommended when:

  - the MicroEJ Platform is used to build various MicroEJ Applications.
  - the MicroEJ Platform is validated using MicroEJ testsuites.
  - the MicroEJ Platform 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 Platform options or as Application options or a mix of both.

The following table describes Platform options, configured in `bsp` > `bsp.properties` file of the Platform configuration project.

Table 1: MicroEJ Platform Options for BSP Connection

Option Name	Description	Example
`microejapp.relative.dir`	The path relative to BSP `root.dir` where to deploy the MicroEJ Application file (`microejapp.o`).	`MicroEJ/lib`
`microejlib.relative.dir`	The path relative to BSP `root.dir` where to deploy the MicroEJ Platform runtime file (`microejruntime.a`).	`MicroEJ/lib`
`microejinc.relative.dir`	The path relative to BSP `root.dir` where to deploy the MicroEJ Platform 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 Platform.	`c:\\Users\\user\\mybsp` on Windows systems or `/home/user/bsp` on Unix systems.

The following table describes Application options, configured as regular *MicroEJ Application Options*.

Table 2: MicroEJ Application Options for BSP Connection

Option Name	Description
`deploy.bsp.microejapp`	Deploy the MicroEJ Application file (`microejapp.o`) to the location defined by the Platform (defaults to `true` when Platform option `microejapp.relative.dir` is set).
`deploy.bsp.microejlib`	Deploy the MicroEJ Platform runtime file (`microejruntime.a`) to the location defined by the Platform (defaults to `true` when Platform option `microejlib.relative.dir` is set).
`deploy.bsp.microejinc`	Deploy the MicroEJ Platform header files (`*.h`) to the location defined by the Platform (defaults to `true` when Platform option `microejinc.relative.dir` is set).
`deploy.bsp.microejscript`	Execute the BSP build script file (`build.bat` or `build.sh`) present at the location defined by the Platform. (defaults to `false` and requires `microejscript.relative.dir` Platform 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 Platform does not includes the BSP.
`deploy.dir.microejapp`	Deploy the MicroEJ Application file (`microejapp.o`) to this absolute directory. An empty value means no deployment.
`deploy.dir.microejlib`	Deploy the MicroEJ Platform runtime file (`microejruntime.a`) to this absolute directory. An empty value means no deployment.
`deploy.dir.microejinc`	Deploy the MicroEJ Platform header files (`*.h`) to this absolute directory. An empty value means no deployment.
`deploy.bsp.microejscript`	Execute the BSP build script file (`build.bat` or `build.sh`) present in this absolute directory. An empty value means no deployment.

Note: It is also possible to configure the BSP root directory using the build option named `toolchain.dir`, instead of the application option `deploy.bsp.root.dir`. This allow to configure a MicroEJ Firmware by specifying both the Platform (using the `target.platform.dir` option) and the BSP at build level, without having to modify the application options files.

For each *Platform BSP connection case*, here is a summary of the options to set:

- No BSP connection, executable file built outside MicroEJ SDK

```
Platform Options:
  [NONE]

Application Options:
  [NONE]
```

- No BSP connection, executable file built using MicroEJ SDK

```
Platform Options:
  [NONE]

Application Options:
  deploy.dir.microejapp=[absolute_path]
  deploy.dir.microejlib=[absolute_path]
  deploy.dir.microejinc=[absolute_path]
  deploy.bsp.microejscript=[absolute_path]
```

- Partial BSP connection, executable file built outside MicroEJ SDK

```
Platform 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 file built using MicroEJ SDK

```
Platform 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 file built using MicroEJ SDK

```
Platform 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]
```

(continues on next page)

(continued from previous page)

```
Application Options:
  deploy.bsp.microejscript=true
```

Build Script File

The BSP build script file is responsible to invoke the third-party C toolchain (compiler and linker) to produce the final executable file (`application.out`).

The build script must implement the following specification:

- On Windows operating system, it is a Windows batch file named `build.bat` .
- On Mac OS X or Linux operating systems, it is a shell script named `build.sh` , with execution permission enabled.
- On build error, the script must end with a non zero exit code.
- On success
  - The executable must be copied to the file `application.out` in the folder 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 Platform Qualification Tools repository.

Low Level APIs Implementation Files

Some MicroEJ Architecture modules require some additional information about the BSP implementation of Low Level APIs.

This information must be stored in each module's configuration folder, in a file named `bsp.xml` .

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.

Example:

```
<bsp>
    <nativeImplementation name="COMM_DRIVER" nativeName="LLCOMM_BUFFERED_CONNECTION"/>
</bsp>
```

These files will be converted into an internal format during the MicroEJ Platform build.

MicroEJ Platform Build

To build the MicroEJ Platform, click on the Build Platform link on the Platform configuration Overview tab.

It will create a MicroEJ Platform in the workspace available for the MicroEJ Application project to run on. The MicroEJ Platform will be available in: Window > Preferences > MicroEJ > Platforms in workspace .

4.2.4 Platform Qualification

Introduction

Platform Qualification is the process of validating implementations of *Low Level APIs* provided by a Platform connected to a custom Board Support Package.

Such validation is performed by running tests at two-levels:

- In C, by calling Low Level APIs (usually manually).
- In Java, by calling Foundation Library APIs (usually automatically using *Platform Testsuite*).

MicroEJ provides a set of tools and pre-defined projects aimed at simplifying the steps for validating Platforms. Please refer to https://github.com/MicroEJ/PlatformQualificationTools.

Platform Testsuite

The purpose of a MicroEJ Platform Testsuite is to validate an implementation of a Low Level API by automatically running Java tests on the device.

The *Testsuite Engine* is used for building, running a Testsuite, and providing a report.

A Platform Testsuite contains one or more tests. For each test, the Testsuite Engine will:

1. Build a MicroEJ Firmware for the test.
2. Program the MicroEJ Firmware 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.

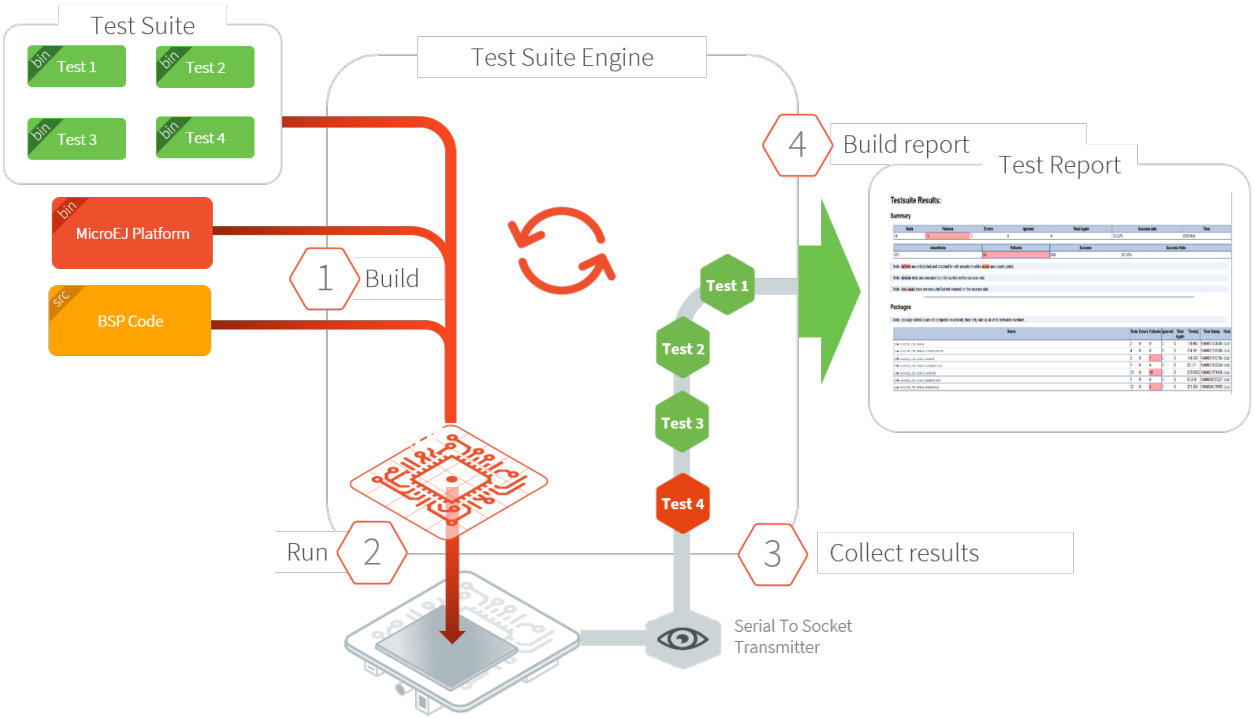


Fig. 13: Platform Testsuite on Device Overview

4.3 MicroEJ Core Engine

The MicroEJ Core Engine (also called the platform engine) and its components represent the core of the platform. It is used to compile and execute at runtime the MicroEJ Application code.

4.3.1 Functional Description

The following diagram shows the overall process. The first two steps are performed within the MicroEJ Workbench. The remaining steps are performed within the C IDE.

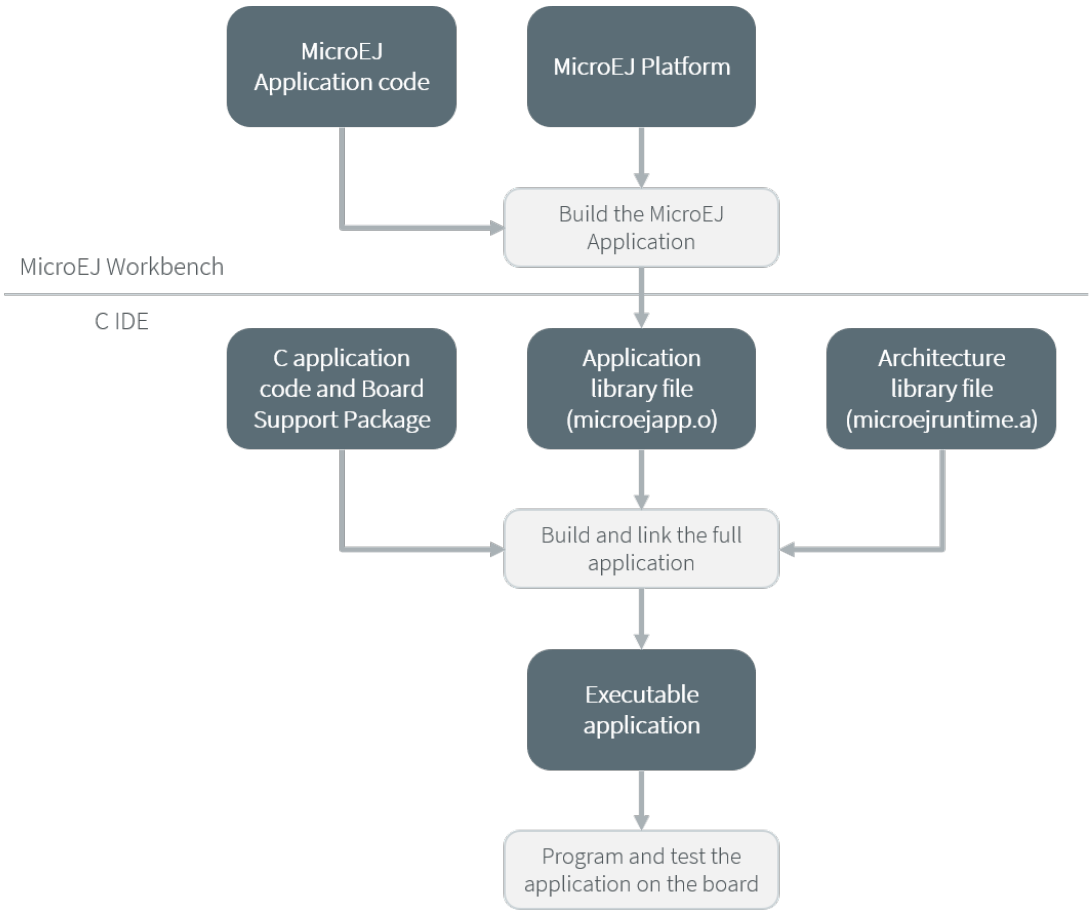


Fig. 14: MicroEJ Core Engine Flow

1. Step 1 consists in writing a MicroEJ Application against a set of Foundation Libraries available in the platform.
2. Step 2 consists in compiling the MicroEJ 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 MicroEJ Core Engine library and a third-party BSP (OS, drivers, etc.). This step may require a third-party linker provided by a C toolchain.

4.3.2 Architecture

The MicroEJ Core Engine and its components have been compiled for one specific CPU architecture and for use with a specific C compiler.

The architecture of the platform engine is called green thread architecture, it runs in a single RTOS task. Its behavior consists in scheduling MicroEJ threads. The scheduler implements a priority preemptive scheduling policy with round robin for the MicroEJ threads with the same priority. In the following explanations the term “RTOS task” refers to the tasks scheduled by the underlying OS; and the term “MicroEJ thread” refers to the Java threads scheduled by the MicroEJ Core Engine.

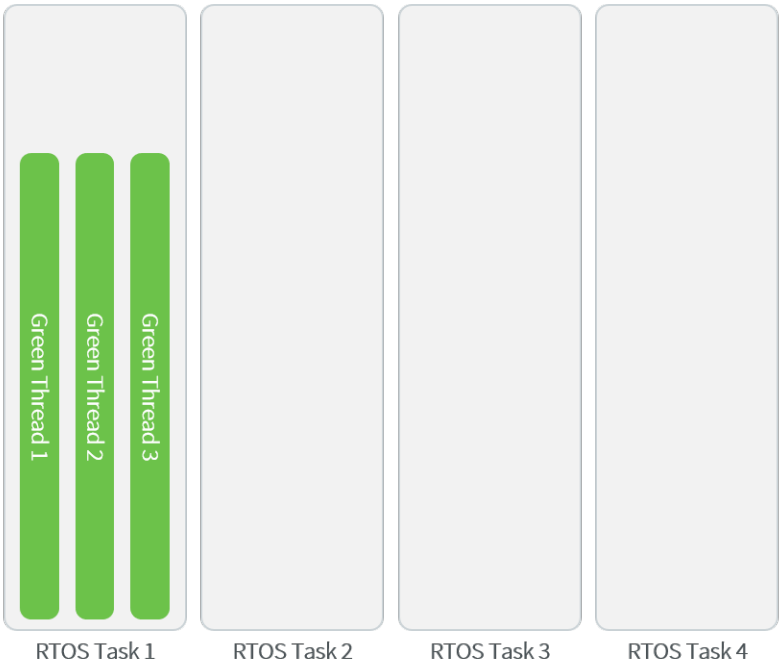


Fig. 15: A Green Threads Architecture Example

The activity of the platform is defined by the MicroEJ Application. When the MicroEJ Application is blocked (when all MicroEJ threads are sleeping), the platform sleeps entirely: The RTOS task that runs the platform sleeps.

The platform is responsible for providing the time to the MicroEJ world: the precision is 1 millisecond.

4.3.3 Capabilities

MicroEJ Core Engine defines 3 exclusive capabilities:

- Mono-sandbox : capability to produce a monolithic firmware (default one).
- Multi-Sandbox : capability to produce a extensible firmware on which new applications can be dynamically installed. See section *Multi-Sandbox*.
- Tiny application : capability to produce a compacted firmware (optimized for size). See section *Tiny application*.

All MicroEJ Core Engine capabilities may not be available on all architectures. Refer to section *Supported MicroEJ Core Engine Capabilities by Architecture Matrix* for more details.

4.3.4 Implementation

The platform implements the *[SNI] specification*. It 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. The function `SNI_destroyVM` handles the platform termination.

The file `LLMJVM_impl.h` that comes with the platform defines the API to be implemented. The file `LLMJVM.h` that comes with the platform defines platform-specific exit code constants. (See *LLMJVM: MicroEJ Core Engine*.)

Initialization

The Low Level MicroEJ Core Engine API deals with two objects: the structure that represents the platform, and the RTOS task that runs the platform. Two callbacks allow engineers to interact with the initialization of both objects:

- `LLMJVM_IMPL_initialize` : Called once the structure representing the platform is initialized.
- `LLMJVM_IMPL_vmTaskStarted` : Called when the platform starts its execution. This function is called within the RTOS task of the platform.

Scheduling

To support the green thread round-robin policy, the platform 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 platform 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 platform to execute a green thread context switch (which gives another MicroEJ thread a chance to run).

Idle Mode

When the platform has no activity to execute, it calls the `LLMJVM_IMPL_idleVM` function, which is assumed to put the RTOS task of the platform into a sleep state. `LLMJVM_IMPL_wakeupVM` is called to wake up the platform task. When the platform task really starts to execute again, it calls the `LLMJVM_IMPL_ackWakeup` function to acknowledge the restart of its activity.

Time

The platform defines two times:

- the application time: The difference, measured in milliseconds, between the current time and midnight, January 1, 1970, UTC.
- the system time: The time since the start of the device. This time is independent of any user considerations, and cannot be set.

The platform relies on the following C functions to provide those times to the MicroEJ world:

- `LLMJVM_IMPL_getCurrentTime` : Depending on the parameter (`true` / `false`) must return the application time or the system time. This function is called by the MicroEJ method `System.currentTimeMillis()` . It is also used by the platform scheduler, and should be implemented efficiently.
- `LLMJVM_IMPL_getTimeNanos` : must return the system time in nanoseconds.
- `LLMJVM_IMPL_setApplicationTime` : must set the difference between the current time and midnight, January 1, 1970, UTC.

Example

The following example shows how to create and launch the MicroEJ Core Engine from the C world. This function (`mjvm_main`) should be called from a dedicated RTOS task.

```
#include <stdio.h>
#include "mjvm_main.h"
#include "LLMJVM.h"
#include "sni.h"
```

(continues on next page)

(continued from previous page)

```
void mjvm_main(void)
{
    void* vm;
    int32_t err;
    int32_t exitcode;

    // create VM
    vm = SNI_createVM();

    if(vm == NULL)
    {
        printf("VM initialization error.\n");
    }
    else
    {
        printf("VM START\n");
        err = SNI_startVM(vm, 0, NULL);

        if(err < 0)
        {
            // Error occurred
            if(err == LLMJVM_E_EVAL_LIMIT)
            {
                printf("Evaluation limits reached.\n");
            }
            else
            {
                printf("VM execution error (err = %d).\n", err);
            }
        }
        else
        {
            // VM execution ends normally
            exitcode = SNI_getExitCode(vm);
            printf("VM END (exit code = %d)\n", exitcode);
        }

        // delete VM
        SNI_destroyVM(vm);
    }
}
```

Debugging

The internal MicroEJ Core Engine function called `LLMJVM_dump` allows you to dump the state of all MicroEJ threads: name, priority, stack trace, etc. This function can be called at any time and from an interrupt routine (for instance from a button interrupt).

This is an example of a dump:

```
============ VM Dump ============
2 java threads
--------------------------------
Java Thread[3]
```

(continues on next page)

(continued from previous page)

```
name="SYSINpmp" prio=5 state=WAITING

java/lang/Thread:
    at com/is2t/microbsp/microui/natives/NSystemInputPump.@134261800
 [0x0800AC32]
    at com/is2t/microbsp/microui/io/SystemInputPump.@134265968
 [0x0800BC80]
    at ej/microui/Pump.@134261696
 [0x0800ABCC]
    at ej/microui/Pump.@134265872
 [0x0800BC24]
    at java/lang/Thread.@134273964
 [0x0800DBC4]
    at java/lang/Thread.@134273784
 [0x0800DB04]
    at java/lang/Thread.@134273892
 [0x0800DB6F]
--------------------------------
Java Thread[2]
name="DISPLpmp" prio=5 state=WAITING

java/lang/Thread:
    at java/lang/Object.@134256392
 [0x08009719]
    at ej/microui/FIFOPump.@134259824
 [0x0800A48E]
    at ej/microui/io/DisplayPump.134263016
 [0x0800B0F8]
    at ej/microui/Pump.@134261696
 [0x0800ABCC]
    at ej/microui/Pump.@134265872
 [0x0800BC24]
    at ej/microui/io/DisplayPump.@134262868
 [0x0800B064]
    at java/lang/Thread.@134273964
 [0x0800DBC4]
    at java/lang/Thread.@134273784
 [0x0800DB04]
    at java/lang/Thread.@134273892
 [0x0800DB6F]
================================
```

See *Stack Trace Reader* for additional info related to working with VM dumps.

4.3.5 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.

4.3.6 Link

Several sections are defined by the MicroEJ Core Engine. Each section must be linked by the third-party linker.

Table 3: Linker Sections

Section name	Aim	Location	Alignment (in bytes)
`.bss.features.installed`	Resident applications statics	RW	4
`.bss.soar`	Application static	RW	8
`.bss.vm.stacks.java`	Application threads stack blocks	RW	8
`ICETEA_HEAP`	MicroEJ Core Engine internal heap	Internal RW	8
`_java_heap`	Application heap	RW	4
`_java_immortals`	Application immortal heap	RW	4
`.rodata.resources`	Application resources	RO	16
`.rodata.soar.features`	Resident applications code and resources	RO	4
`.shieldedplug`	Shielded Plug data	RO	4
`.text.soar`	Application and library code	RO	16

Note: Sections `ICETEA_HEAP`, `_java_heap` and `_java_immortals` are zero-initialized at MicroEJ Core Engine startup.

4.3.7 Dependencies

The MicroEJ Core Engine requires an implementation of its low level APIs in order to run. Refer to the chapter *Implementation* for more information.

4.3.8 Installation

The MicroEJ Core Engine and its components are mandatory. In the platform configuration file, check Multi Applications to install the MicroEJ Core Engine in "Multi-Sandbox" mode. Otherwise, the "Single application" mode is installed.

4.3.9 Use

The EDC API Module must be added to the *module.ivy* of the MicroEJ Application Project. This MicroEJ module is always required in the build path of a MicroEJ project; and all others libraries depend on it. This library provides a set of options. Refer to the chapter *Application Options* which lists all available options.

```
<dependency org="ej.api" name="edc" rev="1.3.3"/>
```

The BON API Module must also be added to the *module.ivy* of the MicroEJ Application project in order to access the *[BON] library*.

```
<dependency org="ej.api" name="bon" rev="1.4.0"/>
```

4.4 Multi-Sandbox

4.4.1 Principle

The Multi-Sandbox capability of the MicroEJ Core Engine allows a main application (called Standalone Application) to install and execute at runtime additional applications (called sandboxed applications).

The MicroEJ Core Engine implements the *[KF] specification*. A Kernel is a Standalone Application generated on a Multi-Sandbox-enabled platform. A Feature is a sandboxed application generated against a Kernel.

A sandboxed application may be dynamically downloaded at runtime or integrated at build-time within the executable application.

Note that the Multi-Sandbox is a capability of the MicroEJ Core Engine. The MicroEJ Simulator always runs an application as a Standalone Application.

4.4.2 Functional Description

The Multi-Sandbox process extends the overall process described in *the overview of the platform process*.

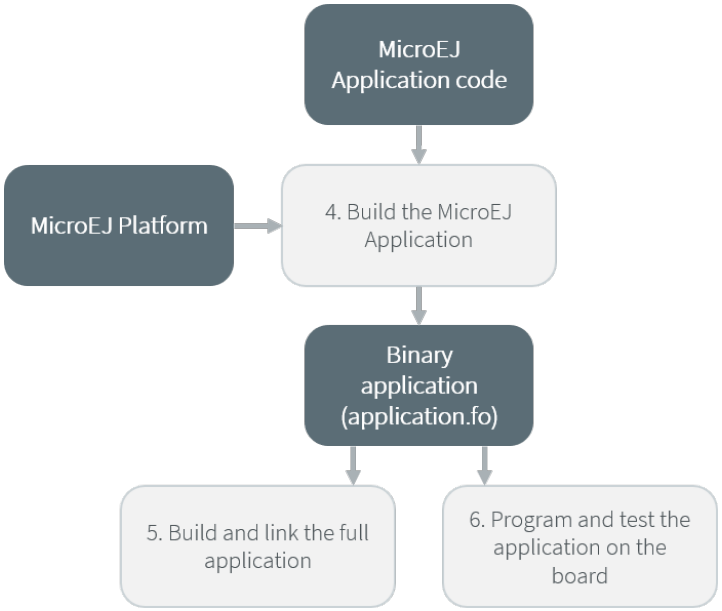


Fig. 16: Multi-Sandbox Process

Once a Kernel has been generated, additional MicroEJ Application code (Feature) can be built against the Kernel by :

- Creating one launch configuration per feature.
- Setting the Settings field in the Execution tab of each feature launch configuration to Build Dynamic Feature .
- Setting the Kernel field in the Configuration tab of each feature launch configuration to the ... .

using the MicroEJ Application launch named Build Dynamic Feature. The binary application file produced (`application.fo`) is compatible only for the Kernel on which it was generated. Generating a new Kernel requires that you generate the Features again on this Kernel.

The Features built can be deployed in the following ways:

- Downloaded and installed at runtime by software. Refer to the *[KF] specification* for `ej.kf.Kernel` install APIs.
- Linked at build-time into the executable application. Features linked this way are then called Installed Features. The Kernel should have been generated with options for dimensioning the maximum size (code, data) for such Installed Features. Features are linked within the Kernel using the Firmware linker tool.

4.4.3 Firmware Linker

A MicroEJ tool is available to link Features as Installed Features within the executable application. The tool name is Firmware Linker. It takes as input the executable application file and the Feature binary code into which to be linked. It outputs a new executable application file, including the Installed Feature. This tool can be used to append multiple Features, by setting as the input file the output file of the previous pass.

4.4.4 Memory Considerations

Multi-Sandbox memory overhead of MicroEJ Core Engine runtime elements are described in *the table below*.

Table 4: Multi-Sandbox Memory Overhead

Runtime element	Memory	Description
Object	RW	4 bytes
Thread	RW	24 bytes
Stack Frame	RW	8 bytes
Class Type	RO	4 bytes
Interface Type	RO	8 bytes

4.4.5 Dependencies

- `LLKERNEL_impl.h` implementation (see *LLKERNEL: Multi-Sandbox*).

4.4.6 Installation

Multi-Sandbox is an additional module, disabled by default.

To enable Multi-Sandbox of the MicroEJ Core Engine, in the platform configuration file, check Multi Applications .

4.4.7 Use

The KF API Module must be added to the *module.ivy* of the MicroEJ Application project in order to allow access to *[KF]* library.

```
<dependency org="ej.api" name="kf" rev="1.4.4"/>
```

This library provides a set of options. Refer to the chapter *Application Options* which lists all available options.

4.5 Tiny application

4.5.1 Principle

The Tiny application capability of the MicroEJ Core Engine allows to build a main application optimized for size. This capability is suitable for environments requiring a small memory footprint.

4.5.2 Installation

Tiny application is an option disabled by default. To enable Tiny application of the MicroEJ Core Engine, set the property `mjvm.standalone.configuration` in `configuration.xml` file as follows:

```
<property name="mjvm.standalone.configuration" value="tiny"/>
```

See section *Platform Customization* for more info on the `configuration.xml` file.

4.5.3 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.

4.6 Native Interface Mechanisms

The MicroEJ 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.

4.6.1 Simple Native Interface (SNI)

Principle

[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.
- Access an Immortal array in a C function (see the *[BON] specification* to learn about immortal objects).

[SNI] does not allow you to:

- Access or create a Java object in a C function.
- Access Java static variables in a C function.
- Call Java methods from a C function.

[SNI] provides some Java APIs to manipulate some data arrays between Java and the native (C) world.

Functional Description

[SNI] defines how to cross the barrier between the Java world and the native world:

- Call a C function from Java.
- Pass parameters to the C function.
- Return a value from the C world to the Java world.
- Manipulate (read & write) shared memory both in Java and C : the immortal space.

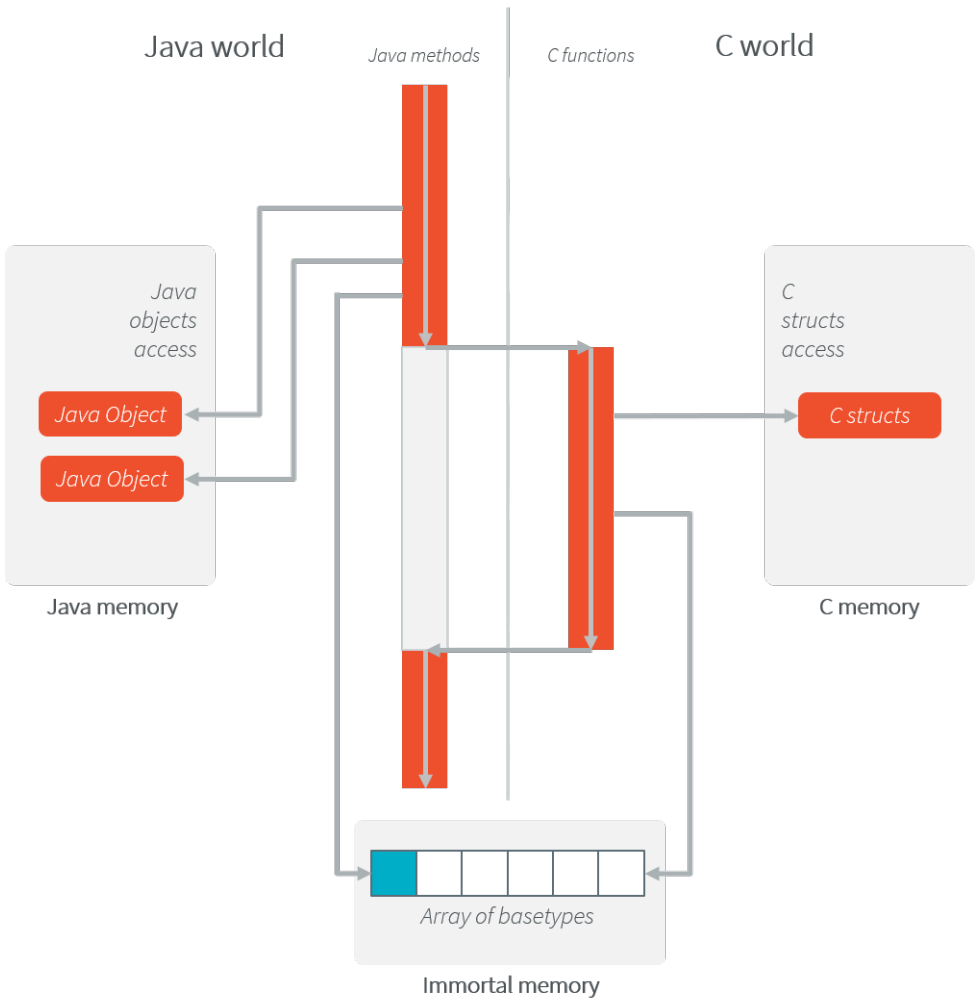


Fig. 17: *[SNI]* Processing

The above illustration shows both Java and C code accesses to shared objects in the immortal space, while also accessing their respective memory.

Example

```
package example;
```

(continues on next page)

(continued from previous page)

```
import java.io.IOException;

/**
 * Abstract class providing a native method to access sensor value.
 * This method will be executed out of virtual machine.
 */
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
    }
}
```

```
// 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;
            }
```

Synchronization

A call to a native function uses the same RTOS task as the RTOS task used to run all Java green threads. So during this call, the MicroEJ Core Engine cannot schedule other Java threads.

[SNI] defines C functions that provide controls for the green threads' activities:

- `int32_t SNI_suspendCurrentJavaThread(int64_t timeout)` : Suspends the execution of the Java 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 green thread is suspended until either an RTOS 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 Java 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 green thread.
- `int32_t SNI_resumeJavaThread(int32_t id)` : Resumes the green thread with the given ID. If the thread is not suspended, the resume stays pending.

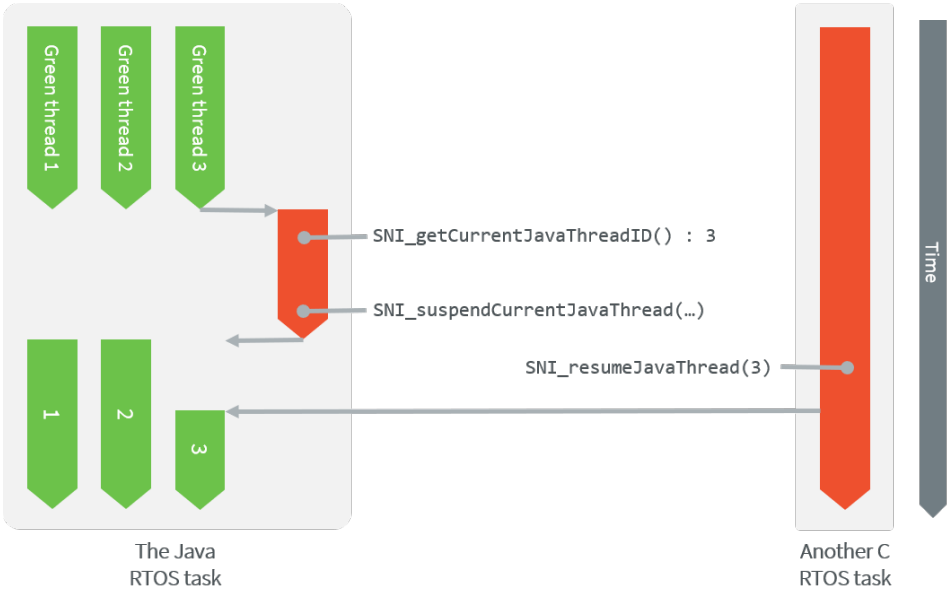


Fig. 18: Green Threads and RTOS Task Synchronization

The above illustration shows a green thread (GT3) 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 RTOS task may later resume the Java green thread.

Dependencies

No dependency.

Installation

The *[SNI]* library is a built-in feature of the platform, so there is no additional dependency to call native code from Java. In the platform configuration file, check Java to C Interface > SNI API to install the additional Java APIs in order to manipulate the data arrays.

Use

The SNI API module must be added to the *module.ivy* of the MicroEJ Application project, in order to allow access to the *[SNI]* library.

```
<dependency org="ej.api" name="sni" rev="1.3.1"/>
```

4.6.2 Shielded Plug (SP)

Principle

The Shielded Plug *[SP]* 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.

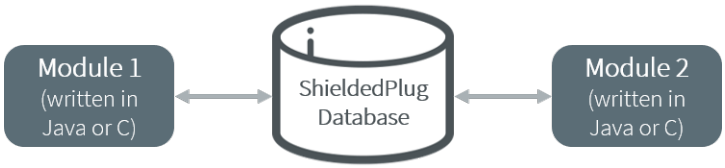


Fig. 19: A Shielded Plug Between Two Application (Java/C) Modules.

Functional Description

The usage of the Shielded Plug (SP) starts with the definition of a database. The implementation of the *[SP]* for the MicroEJ Platform 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. See also the Java documentation from the MicroEJ Workbench resources center ("Javadoc" menu). The C header file `sp.h` available in the MicroEJ Platform `source/MICROJVM/include` folder contains the C functions for accessing the database.

To embed the *[SP]* 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 and one in C. These constants can be used by either the Java or the C application modules.

Shielded Plug Compiler

A MicroEJ tool is available to launch the *[SP]* compiler tool. 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 and one in C, of the block ID constants to be used by either Java or C application modules.

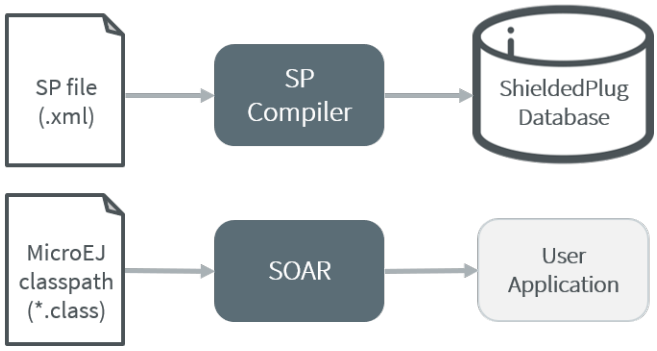


Fig. 20: Shielded Plug Compiler Process Overview

Example

Below is an example of using a database *[SP]*. The code that publishes the data is written in C, and the code that receives the data is written in Java. 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
```

(continues on next page)

(continued from previous page)

```
            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();
      }
      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 platform. In the platform 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 *module.ivy* of the MicroEJ Application project in order to allow access to the *[SP]* library.

```
<dependency org="ej.api" name="sp" rev="2.0.2"/>
```

This library provides a set of options. Refer to the chapter *Application Options* which lists all available options.

4.6.3 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).

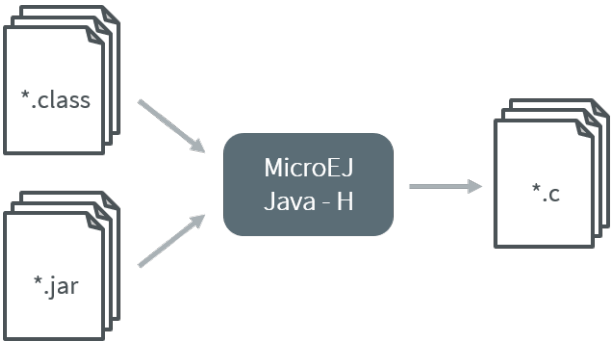


Fig. 21: MicroEJ Java H Process

Dependencies

No dependency.

Installation

This is an additional tool. In the platform configuration file, check Java to C Interface > MicroEJ Java H to install the tool.

Use

This chapter explains the MicroEJ tool options.

4.7 External Resources Loader

4.7.1 Principle

A *resource* is, for a MicroEJ Application, the contents of a file. This file is known by its path (its relative path from the MicroEJ Application classpath) and its name. The file may be stored in RAM, flash, or external flash; and it is the responsibility of the MicroEJ Core Engine and/or the BSP to retrieve and load it.

MicroEJ Platform makes the distinction between two kinds of resources:

- Internal resource: The resource is taken into consideration during the MicroEJ Application build. The SOAR step loads the resource and copies it into the same C library as the MicroEJ Application. Like the MicroEJ Application, the resource is linked into the CPU address space range (internal device memories, external parallel memories, etc.).

  The available list of internal resources to embed must be specified in the MicroEJ Application launcher (MicroEJ launch). Under the "Resources" tab, select all internal resources to embed in the final binary file.

- External resource: The resource is not taken into consideration by MicroEJ. It is the responsibility of the BSP project to manage this kind of resource. 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 some specific Low Level API (LLAPI) C functions: `LLEXT_RES_impl.h`. These functions allow the MicroEJ Application to load some external resources.

4.7.2 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 MicroEJ 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).

4.7.3 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`.

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 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.

4.7.4 External Resources Folder

The External Resource Loader module provides an option (MicroEJ launcher option) to specify a folder for the external resources. This folder has two roles:

- It is the output folder used by some extra generators during the MicroEJ Application build. All output files generated by these tools will be copied into this folder. This makes it easier to retrieve the exhaustive list of resources to program on the board.
- This folder is taken into consideration by the Simulator in order to simulate the availability of these resources. When the resources are located in another computer folder, the Simulator is not able to load them.

If not specified, this folder is created (if it does not already exist) in the MicroEJ project specified in the MicroEJ launcher. Its name is `externalResources`.

4.7.5 Dependencies

- `LLEXT_RES_impl.h` implementation (see *LLEXT_RES: External Resources Loader*).

4.7.6 Installation

The External Resources Loader is an additional module. In the platform configuration file, check External Resources Loader to install this module.

4.7.7 Use

The External Resources Loader is automatically used when the MicroEJ Application tries to open an external resource.

4.8 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.

4.8.1 ECOM

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.

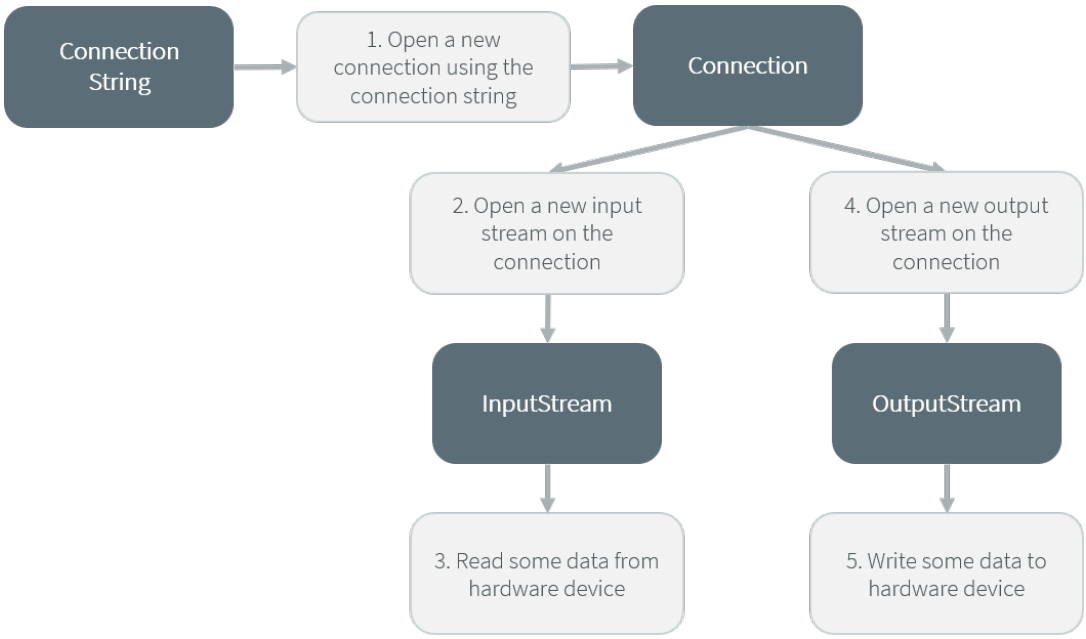


Fig. 22: ECOM Flow

1. Step 1 consists of opening a connection on a hardware device. The connection kind and its configuration are fixed by the parameter `String connectionString` of the method `Connection.open`.
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 Java thread. The mechanism can be enabled or disabled (see *Application Options*).

Dependencies

No dependency.

Installation

ECOM Foundation Library is an additional library. In the platform configuration file, check Serial Communication > ECOM to install the library.

Use

The ECOM API Module must be added to the *module.ivy* of the MicroEJ Application project in order to allow access to the ECOM library.

```
<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.

4.8.2 ECOM Comm

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) that exist to support a Buffered connection.

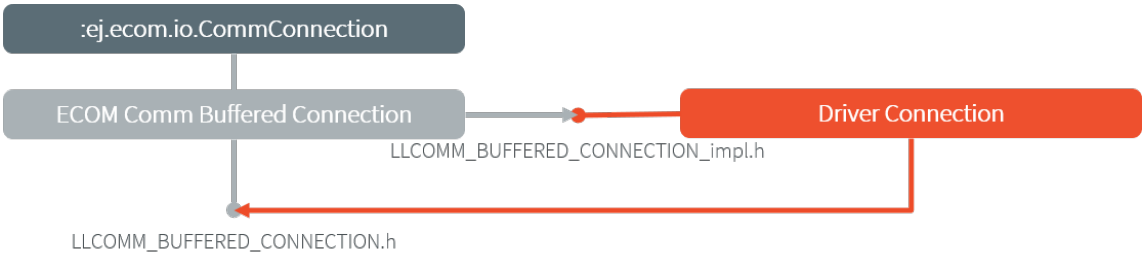


Fig. 23: ECOM Comm components

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

[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.

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 an 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.

Opening Sequence

The following flow chart explains Comm Port opening sequence according to the given Comm Port identifier.

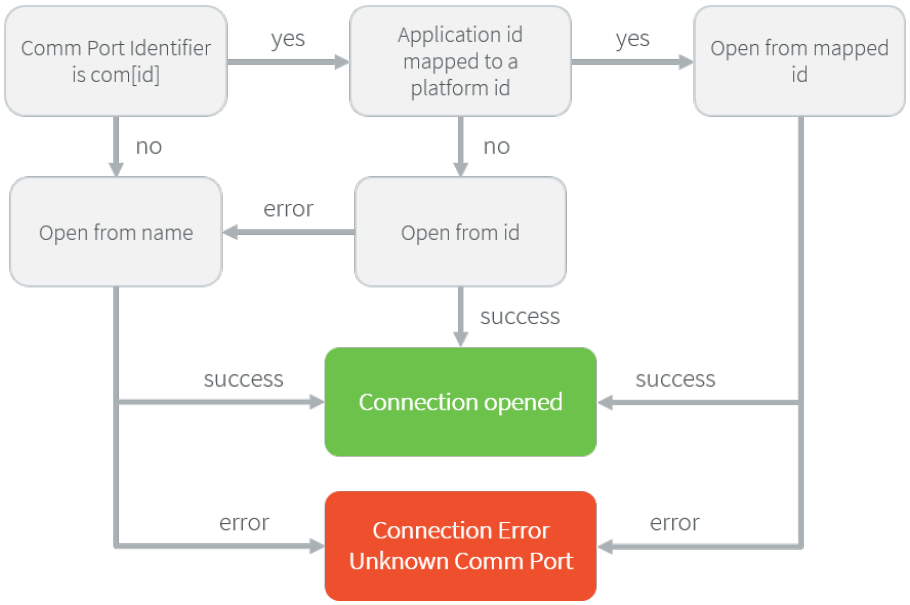


Fig. 24: Comm Port Open Sequence

[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.

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 *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.

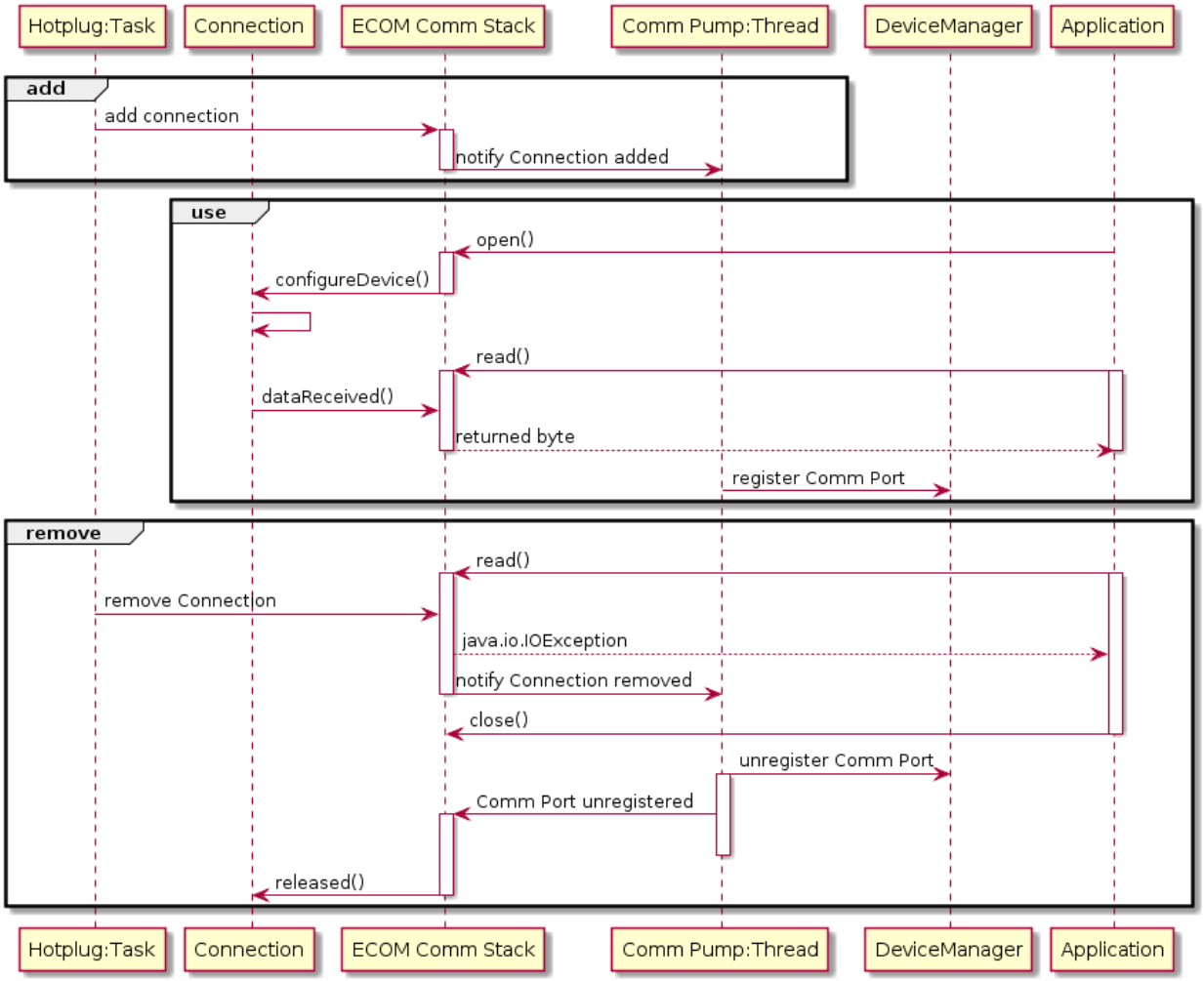


Fig. 25: Dynamic Connection Lifecycle

Java API

Opening a connection is done using `ej.ecom.io.Connector.open(String name)` . The connection string (the `name` 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 MicroJVM task context, a suitable reentrant synchronization mechanism must be implemented (see `LLCOMM_IMPL_syncConnectionsEnter` and `LLCOMM_IMPL_syncConnectionsExit`).

[3] The following examples use Buffered connections, but Custom connections follow the same pattern.

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.

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 Java 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 Java 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 1: 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 *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 2: 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 in order to allow access to 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 *Application Options* which lists all available options.

4.9 Graphical User Interface

Note: This chapter describes the current Graphical User Interface version `3`, provided by UI Pack version `13.0.0` or higher. 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.

4.9.1 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, Low Level APIs `LLUI_xxx`, etc.

The diagram below shows a simplified view of the components involved in the provisioning of User Interface Extension.

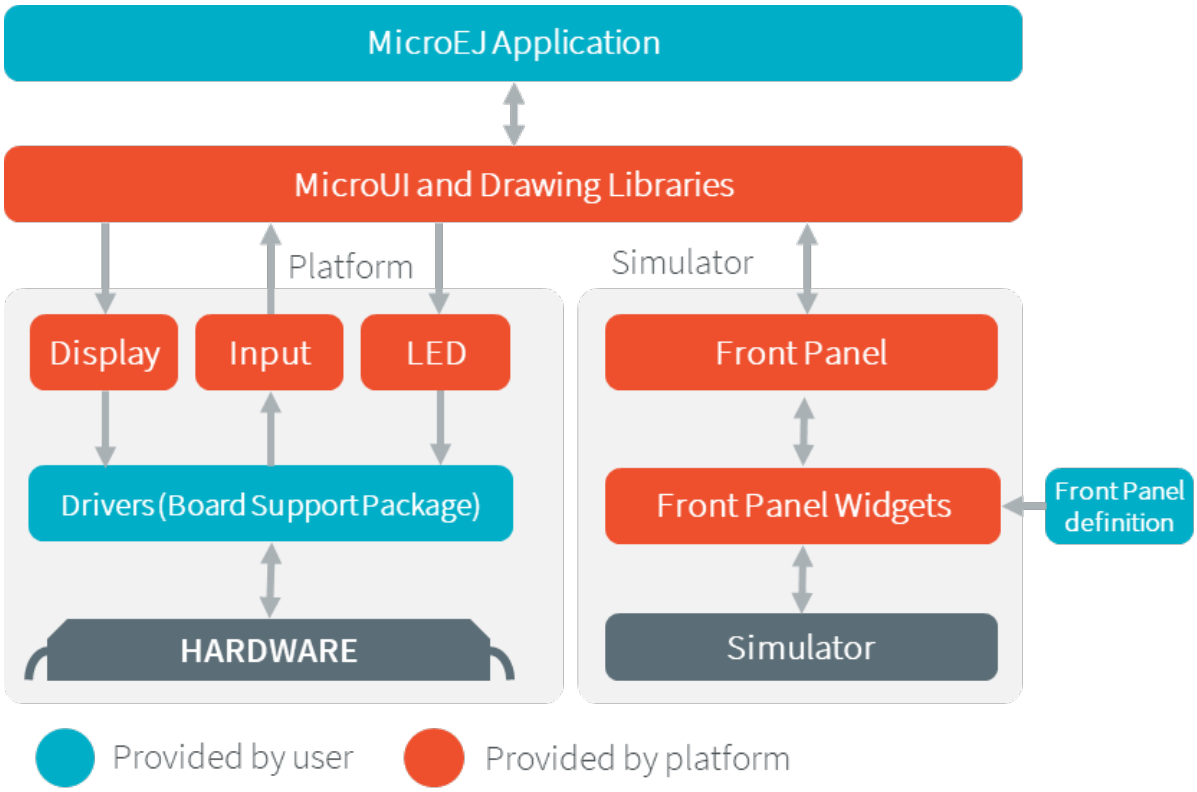


Fig. 26: Overview

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 MicroEJ application. The following diagram depicts the components involved in its design, along with the provided tools:

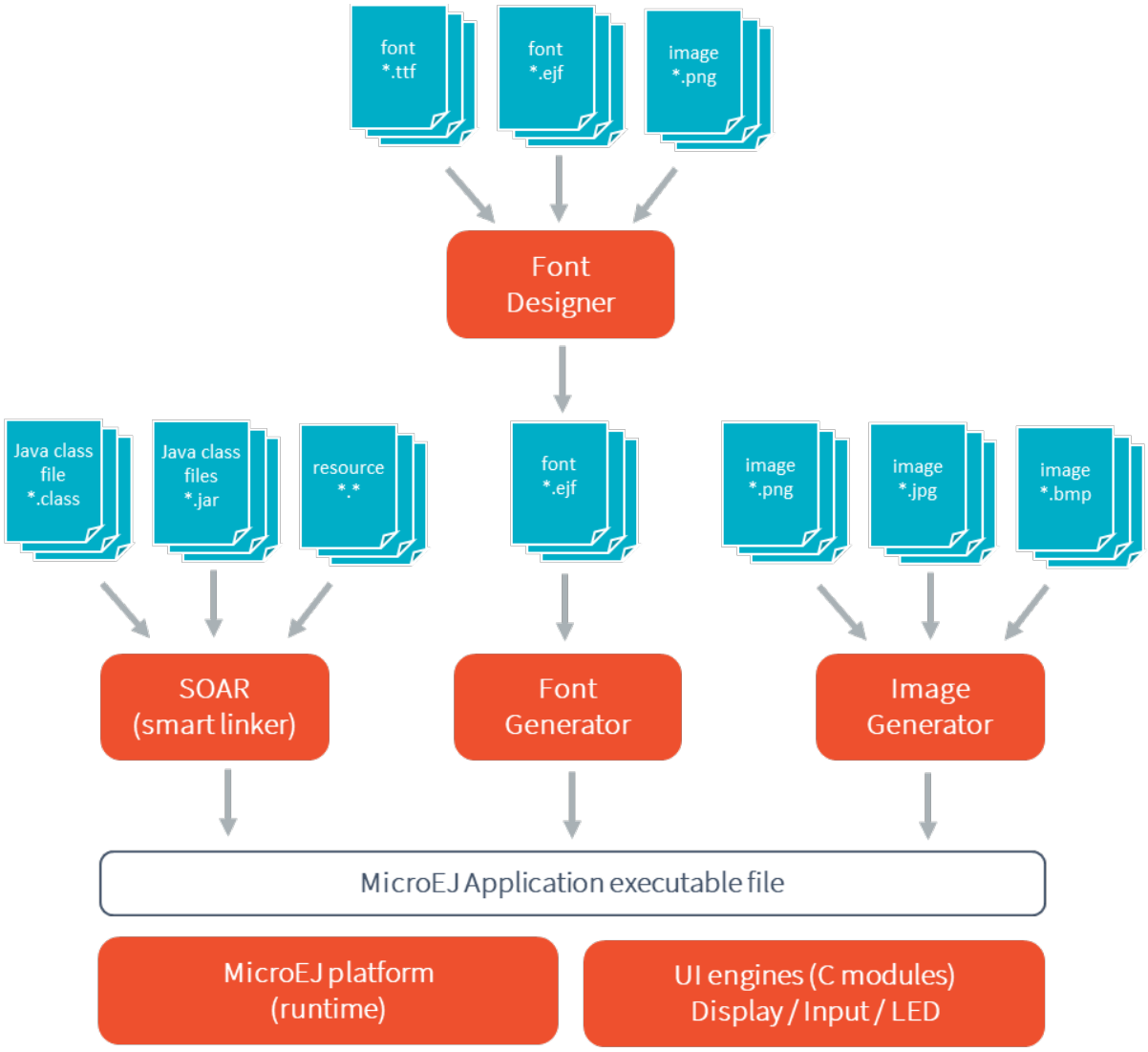


Fig. 27: The User Interface Extension Components along with a Platform

4.9.2 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 MicroEJ 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 `main()` method. The chapter *Static Initialization* explains how to perform the configuration step.

Note: This configuration step is the same for both embedded and smulated platforms.

The embedded platform 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 5: 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 platform uses a mock which simulates all I/O devices. Refer to the chapter *Simulation*.

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 MicroEJ Core Engine can run (see *Memory* options in *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 MicroEJ 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 Java thread is locked (because it cannot continue its execution until both actions are performed). All other Java threads can run, even a thread with a lower priority than current thread. If no thread has to be run, MicroEJ Core Engine goes in sleep mode until the native callback is called.

Transparency

MicroUI provides several policies to use the transparency. 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.

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.

Installation

The MicroUI library is an additional module. In the platform configuration file, check UI > MicroUI to install the library. When checked, the XML file microui > microui.xml is required during platform creation in order to configure the module. This configuration step is used to extend the MicroUI library. Refer to the chapter *Static Initialization* for more information about the MicroUI Initialization step.

Use

See *MicroUI* chapter in Application Developer Guide.

4.9.3 Static Initialization

Principle

The MicroUI implementation for MicroEJ requires a configuration step (also called extension step) to customize itself before MicroEJ 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 MicroEJ platform 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 platform).

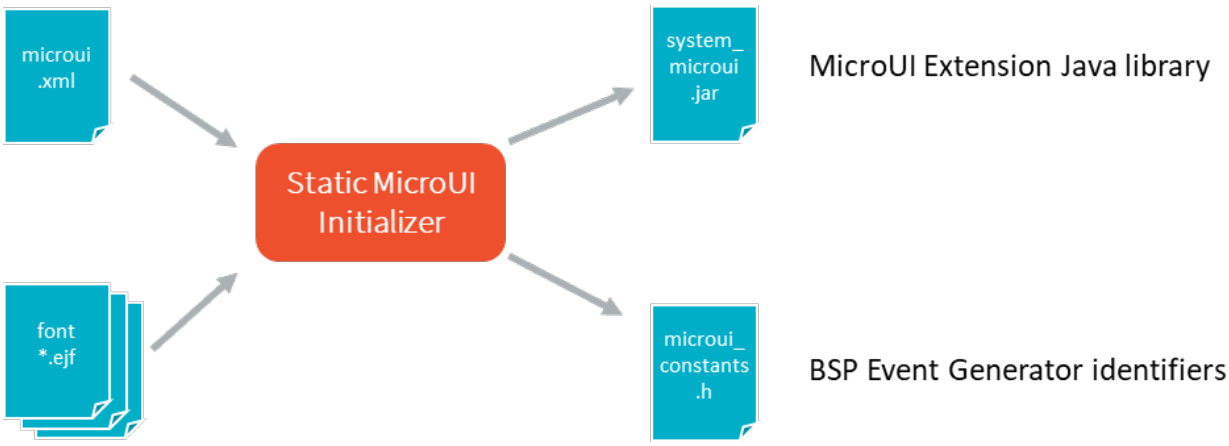


Fig. 28: Static MicroUI Initializer Process

XML File

The XML file must be created in platform configuration project, in folder `microui` and called `microui.xml` .

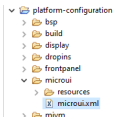


Fig. 29: Static MicroUI Initializer XML File

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 Event Generator: Command, Buttons, States, Pointer and Touch.
  - the configuration of the generic MicroUI Event Generator.

  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"/>
    <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>
```

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 MicroEJ platform build.

4.9.4 Low Level API

Principle

The MicroUI implementation for MicroEJ requires a Low Level implementation. This Low Level implementation finalizes the MicroUI implementation started with the static initialization step (see *Static Initialization*) for a given MicroEJ platform.

The Low Level implementation consists of 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 platform characteristics.

Embedded Platform

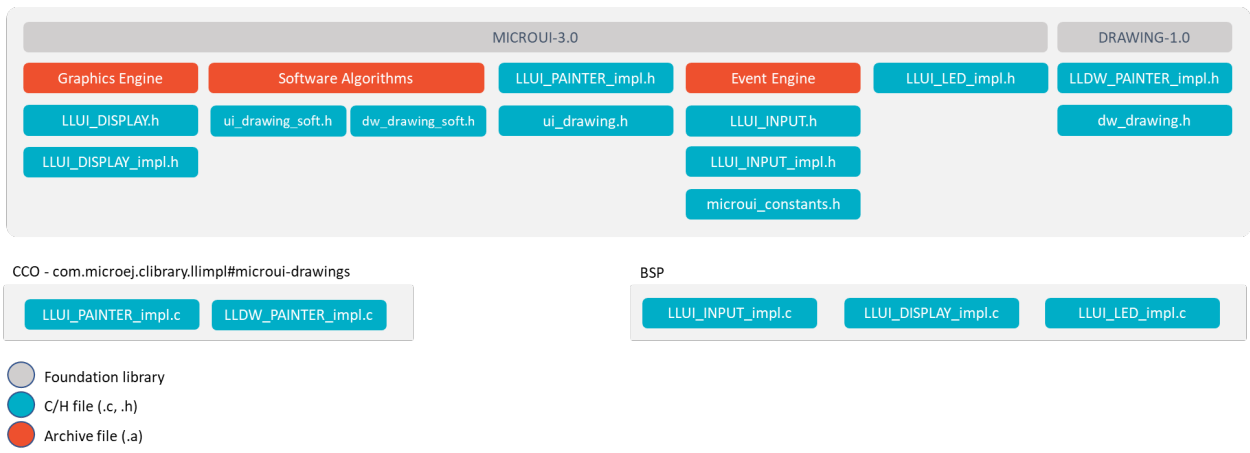


Fig. 30: MicroUI Embedded Low Level API

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.h` and `dw_drawing.h` list all drawing methods the platform can implement.
- `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*).

All header files and their aims are described in next UI engines chapters: *LED*, *Input* and *Display*.

Simulator

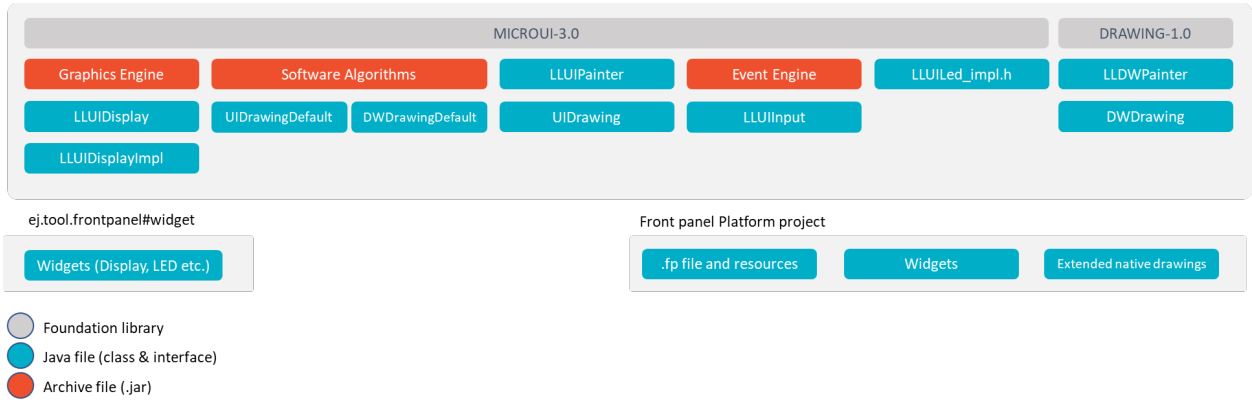


Fig. 31: MicroUI Simulator Low Level API

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 Platform .

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` .
- Second part name refers the UI engine: `Display` , `Input` , `Led`
- Files whose name ends with `Impl` list methods to implement like embedded platform.
- Files whose name has no suffix list internal UI engines 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 platform 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*.

4.9.5 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 Low Level 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 Low Level APIs that receive orders from the Java world.

The Low Level 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 MicroEJ 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 MicroEJ Application may not be able to know the LEDs states.

Low Level API

The LED module provides Low Level APIs that allow the BSP to manage the LEDs. The BSP has to implement these Low Level APIs, making the link between the MicroUI library and the BSP LEDs drivers.

The Low Level 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.

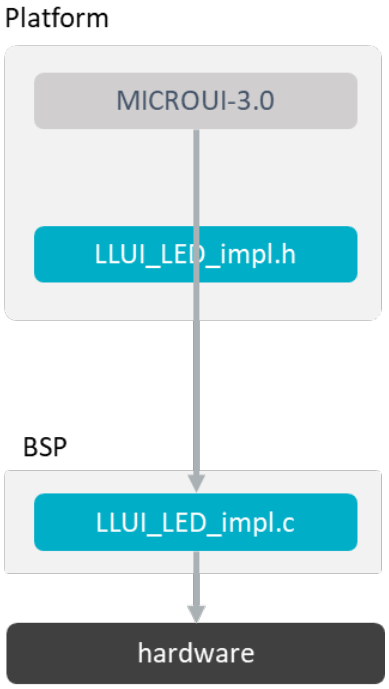


Fig. 32: Led Low Level API

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 MicroEJ Platform. This stub library does not provide any Low Level API files.

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 MicroEJ Platform. If not installed, the *stub* module will be used.

In the platform configuration file, check UI > LEDs to install LEDs.

Use

The MicroUI LEDs APIs are available in the class ej.microui.led.Leds.

4.9.6 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 a Low Level 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 Low Level APIs that send events to the Java world.

Drivers for input devices must generate events that are sent, via a MicroUI Event Generator, to the MicroEJ 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.

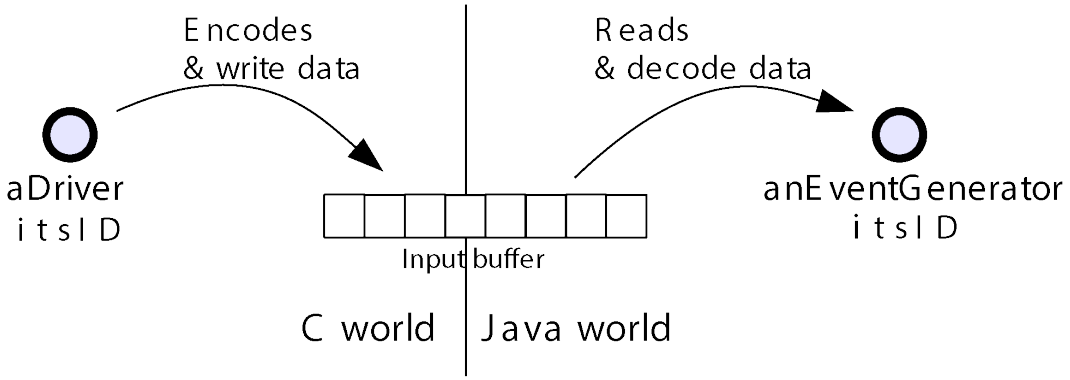


Fig. 33: Drivers and MicroUI Event Generators Communication

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.

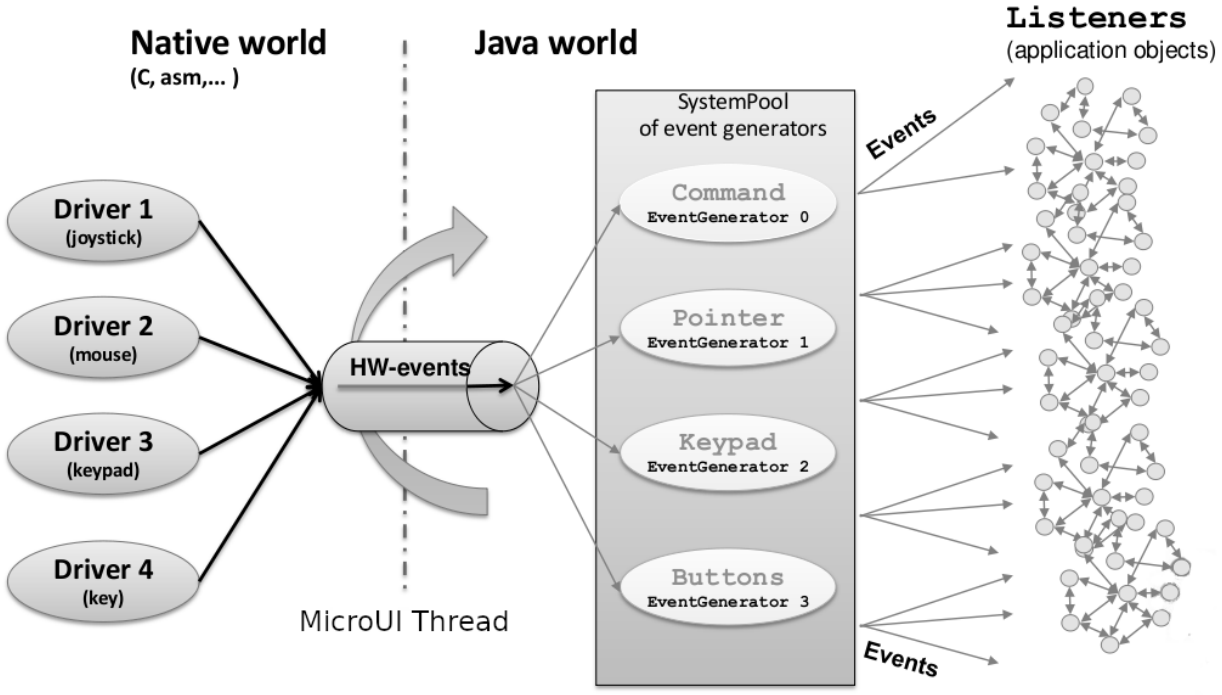


Fig. 34: MicroUI Events Framework

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 bene-

fits:

- 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 MicroEJ 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).

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 MicroEJ 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 Low Level APIs `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.

Low Level API

The implementation of the MicroUI `Event Generator` APIs provides some Low Level APIs. The BSP has to implement these Low Level APIs, making the link between the MicroUI C library `inputs` and the BSP input devices drivers.

The Low Level 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 MicroEJ 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 EventGenerator 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.

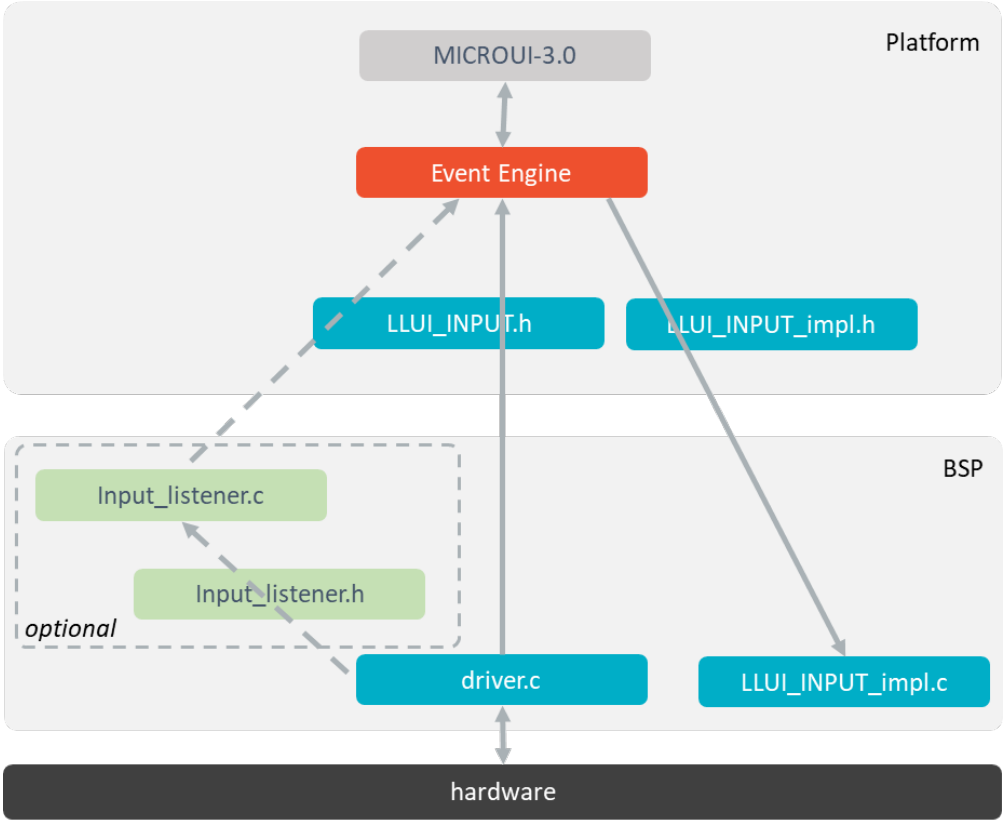


Fig. 35: Input Low Level API

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 MicroEJ Platform. This stub library does not provide any Low Level API files.

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*).

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`.

4.9.7 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 a Low Level APIs for the displays (LLUI_DISPLAY) that the BSP must provide (see *LLUI_DISPLAY: Display*),
- an implementation of a Low Level APIs for MicroUI drawings.

Functional Description

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 images life cycle, 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 fonts. The display characteristics and the drawings are managed by the `LLUI_DISPLAY` implementation.

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 very simple implementation (four functions) of the Graphics Engine low-level APIs.

MicroUI library also gives the possibility to perform some additional drawings which are not available as API in MicroUI library. The Graphics Engine gives 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. Same constraints can be applied, same drawings can be overridden or added, same software drawing rendering is performed (down to the pixel).

Display Configurations

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 modes can vary in three ways:

- the buffer mode: double-buffer, simple buffer (also known as *direct*),
- the memory layout of the pixels,
- pixel format or depth.

Buffer Modes

Overview

When using the double buffering technique, the memory into which the application draws (called graphics buffer or back buffer) is not the memory used by the screen to refresh it (called frame buffer or display buffer). When everything has been drawn consistently from the application point of view, the back buffer contents are synchronized with the display buffer. Double buffering avoids flickering and inconsistent rendering: it is well suited to high quality animations.

For more static display-based applications, and/or to save memory, an alternative configuration is to use only one buffer, shared by both the application and the screen.

Displays addressed by one of the standard configurations are called *generic displays*. For these generic displays, there are three buffer modes: switch, copy and direct. The following flow chart provides a handy guide to selecting the appropriate buffer mode according to the hardware configuration.

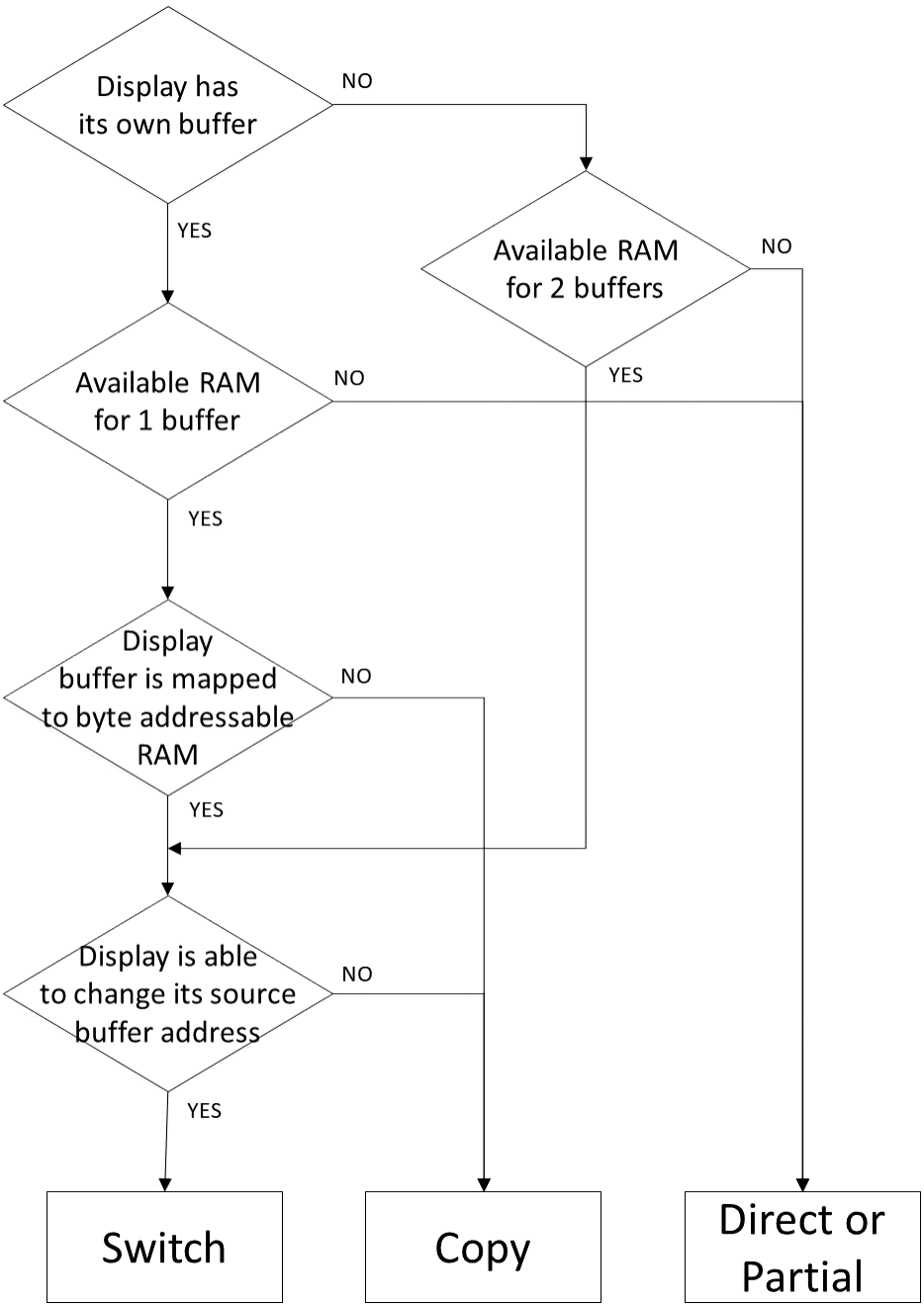


Fig. 36: Buffer Modes

Implementation

The Graphics Engine does not depend on the type of buffer mode. The implementation of Display.flush() calls the Low Level API `LLUI_DISPLAY_IMPL_flush` to let the BSP to update the display data. This function should be atomic and the implementation has to return the new graphics buffer address (back buffer address). In `direct` and `copy` modes, this address never changes and the implementation has always to return the back buffer address. In `switch` mode, the implementation has to return the old display frame buffer address.

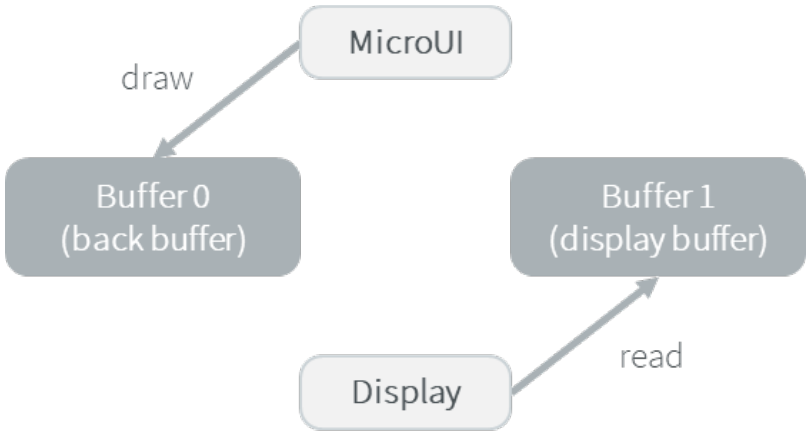
The next sections describe the work to do for each mode.

Switch

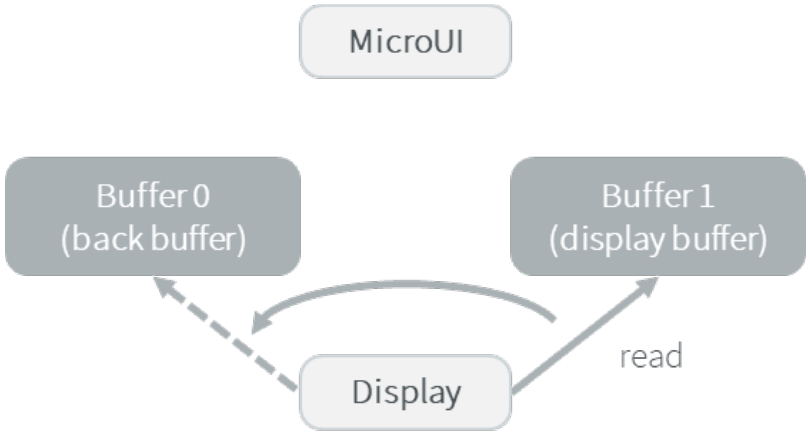
The switch mode is a double-buffered mode where two buffers in RAM alternately play the role of the back buffer and the display buffer. The display source is alternatively changed from one buffer to the other. Switching the source address may be done asynchronously. The synchronize function is called before starting the next set of draw operations, and must wait until the driver has switched to the new buffer.

Synchronization steps are described *below*.

- *Step 1:* Drawing
  MicroUI is drawing in buffer 0 (back buffer) and the display is reading its contents from buffer 1 (display buffer).



- *Step 2:* Switch
  The drawing is done. Set that the next read will be done from buffer 0.
  Note that the display "hardware component" asynchronously continues to read data from buffer 1.

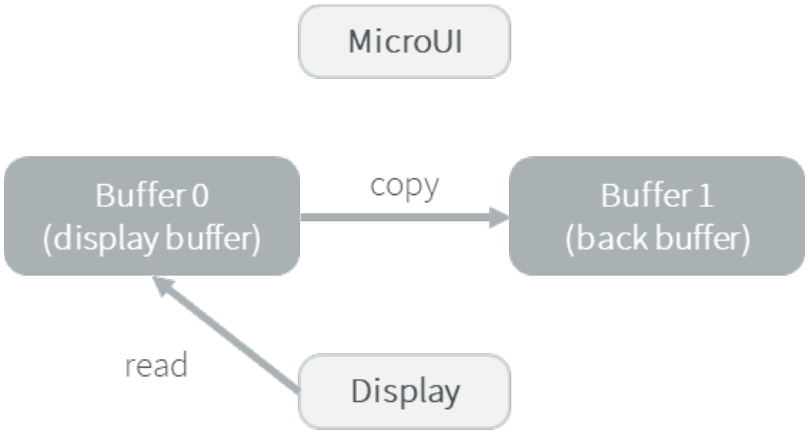


- *Step 3:* Copy
  A copy from the buffer 0 (new display buffer) to the buffer 1 (new back buffer) must be done to keep the contents of the current drawing. The copy routine must wait until the display has finished the switch, and start asynchronously by comparison with the MicroUI drawing routine (see next step).
  This copy routine can be done in a dedicated RTOS task or in an interrupt routine. The copy should start after the display "hardware component" has finished a full buffer read to avoid flickering.
  Usually a tearing signal from the display at the end of the read of the previous buffer (buffer 1) or at the

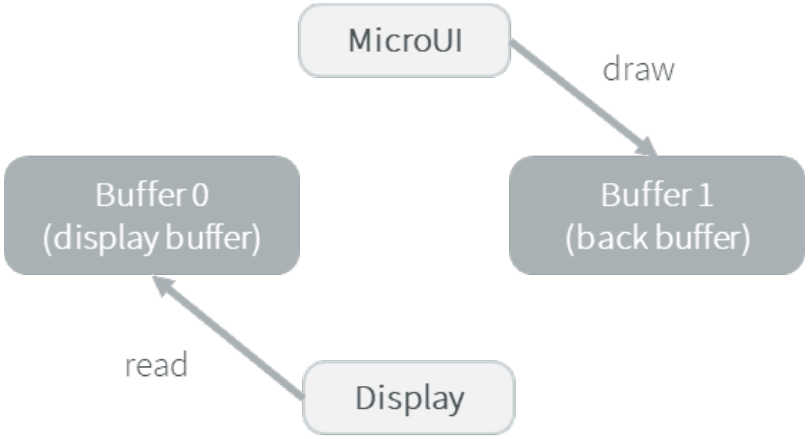
beginning of the read of the new buffer (buffer 0) throws an interrupt. The interrupt routine starts the copy using a DMA.

If it is not possible to start an asynchronous copy, the copy must be performed in the MicroUI drawing routine, at the beginning of the next step.

Note that the copy is partial: only the parts that have changed need to be copied, lowering the CPU load.



- *Step 4:* Synchronisation
  Waits until the copy routine has finished the full copy.
  If the copy has not been done asynchronously, the copy must start after the display has finished the switch. It is a blocking copy because the next drawing operation has to wait until this copy is done.
- *Step 5:* Next draw operation
  Same behavior as step 1 with buffers reversed.

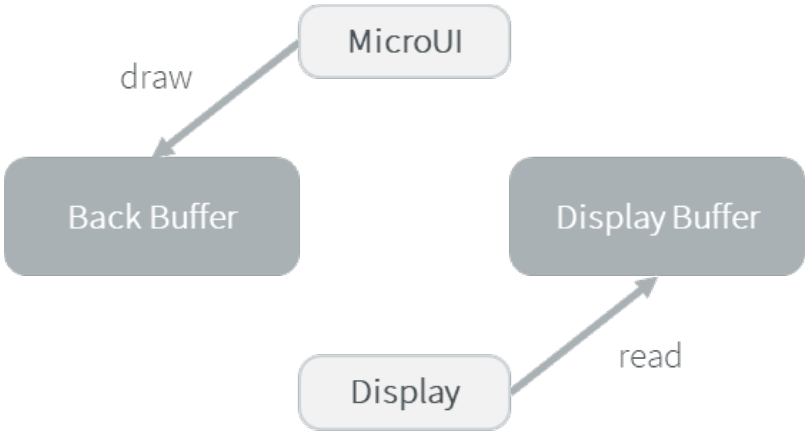


Copy

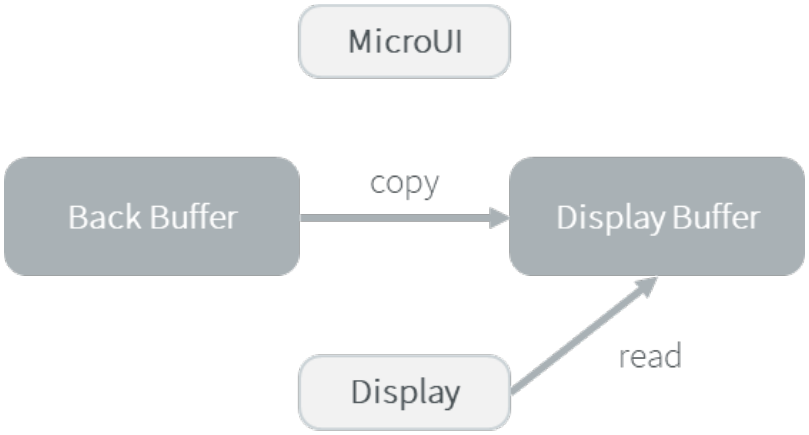
The copy mode is a double-buffered mode where the back buffer is in RAM and has a fixed address. To update the display, data is sent to the display buffer. This can be done either by a memory copy or by sending bytes using a bus, such as SPI or I2C.

Synchronization steps are described *below*.

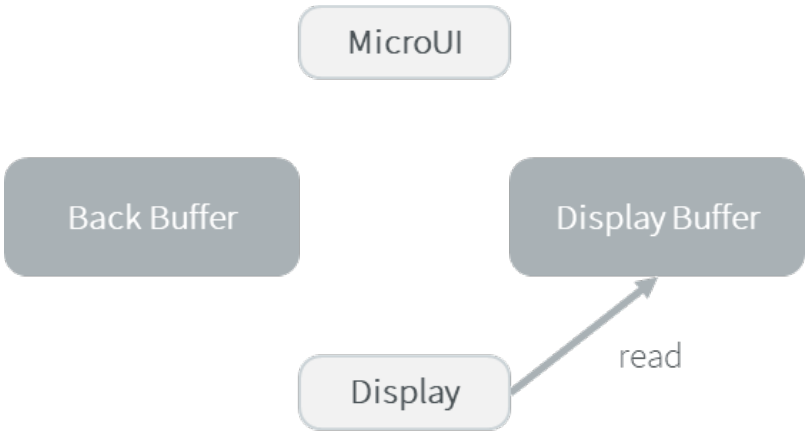
- *Step 1:* Drawing
  MicroUI is drawing in the back buffer and the display is reading its content from the display buffer.



- *Step 2:* Copy
  The drawing is done. A copy from the back buffer to the display buffer is triggered.
  Note that the implementation of the copy operation may be done asynchronously – it is recommended to wait until the display "hardware component" has finished a full buffer read to avoid flickering. At the implementation level, the copy may be done by a DMA, a dedicated RTOS task, interrupt, etc.

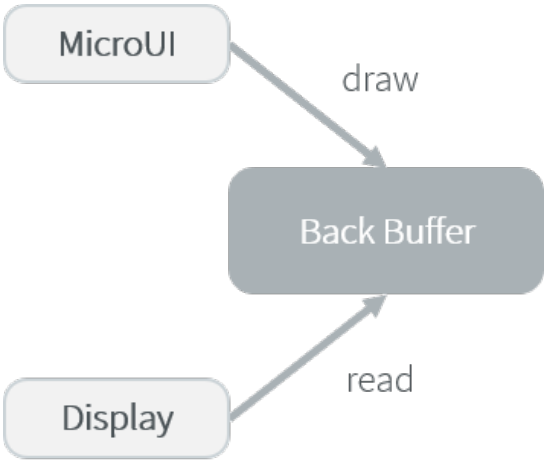


- *Step 3:* Synchronization
  The next drawing operation waits until the copy is complete.



Direct

The direct mode is a single-buffered mode where the same memory area is used for the back buffer and the display buffer (*See illustration below*). Use of the direct mode is likely to result in "noisy" rendering and flickering, but saves one buffer in runtime memory.



Partial Buffer

In the case where RAM usage is not a constraint, the graphics buffer is sized to store all the pixel data of the screen. However, 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 mode 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 complete 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 sent to the display (either copied to its internal memory or to the complete buffer in RAM).
3. Finally, a synchronization is required before starting the next drawing operation.

Dual Partial Buffer

A second partial buffer can be used to avoid the synchronization delay before 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 requires more RAM or to reduce 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 mode. 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 mode. 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 which 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.

Byte Layout

This chapter concerns only display 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 same line. When the end of line is reached, a padding is added in order to start a new line with a new byte.
- column: The byte contains several consecutive pixels on same column. When the end of column is reached, a 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 pixels representation (see *Installation*).

Table 6: Byte Layout: line

<table>
<tr><td>BPP</td><td>MSB</td><td></td><td></td><td></td><td></td><td></td><td></td><td>LSB</td></tr>
<tr><td>4</td><td colspan="4">pixel 1</td><td colspan="4">pixel 0</td></tr>
<tr><td>2</td><td colspan="2">pixel 3</td><td colspan="2">pixel 2</td><td colspan="2">pixel 1</td><td colspan="2">pixel 0</td></tr>
<tr><td>1</td><td>pixel 7</td><td>pixel 6</td><td>pixel 5</td><td>pixel 4</td><td>pixel 3</td><td>pixel 2</td><td>pixel 1</td><td>pixel 0</td></tr>
</table>

Table 7: 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

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 8: 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 the 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 pixels representation (see *Installation*).

Table 9: 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 10: 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 11: 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 12: 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)

Pixel Structure

The Display module provides pre-built display configurations with 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*).

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**. 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)
```

(continues on next page)

(continued from previous page)

```
            ;
}
```

- 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)
            ;
}
```

- 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);
}
```

When the value is one among this list: `1` | `2` | `4` | `8` | `16` | `24` | `32`, the Display module considers the pixel representation as **driver-specific**. In this case, the driver must implement functions that convert MicroUI's standard 32 bits ARGB colors to display color representation (see *LLUI_DISPLAY: Display*). 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*.

Low Level API

Overview

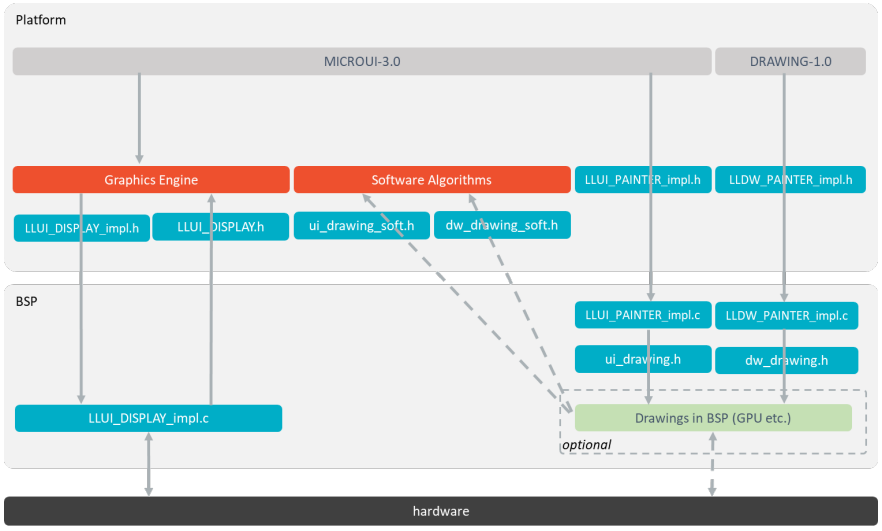


Fig. 37: Display Low Level API

- 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 drawing native functions are implemented in the CCO `com.microej.clibrary.llimpl#microui` ; this CCO must be included in BSP.
- This CCO redirects drawings the implementation of `ui_drawing.h` .
- `ui_drawing.h` is already implemented by *software algorithms* library (not represented in previous picture).
- `ui_drawing.h` can be too implemented in BSP to use a GPU for instance.
- This Implementation is allowed to call *software algorithms* through `ui_drawing_soft.h` header file.
- MicroEJ library `Drawing` performs same operations with header files `LLDW_PAINTER_impl.h` , `dw_drawing_impl.h` and `dw_drawing.h` ; and with C file `LLDW_PAINTER_impl.c` also available in CCO `com.microej.clibrary.llimpl#microui` .

Required Low Level API

Some four Low Level APIs are required to connect the Graphics Engine on the display driver. The functions are listed in `LLUI_DISPLAY_impl.h` .

- `LLUI_DISPLAY_IMPL_initialize` : The initialization function is called when MicroEJ application is calling `MicroUI.start()` . Before this call, the display is useless and don't need to be initialized. This function consists in initializing the LCD driver and in filling the given structure `LLUI_DISPLAY_SInitData` . This structure has to contain pointers on two binary semaphores (see after), the back buffer address (see *Display Configurations*), the display *virtual* size in pixels and optionally the display *physical* size in pixels. The display *virtual* size is the size of the area where the drawings are visible. The display *physical* size is the required memory size where the area is located. Virtual memory size is: `display_width * display_height * bpp / 8` . On some devices the memory width (in pixels) is higher than virtual width. In this way, the graphics buffer memory size is: `memory_width * memory_height * bpp / 8` .
- `LLUI_DISPLAY_IMPL_binarySemaphoreTake` and `LLUI_DISPLAY_IMPL_binarySemaphoreGive` : The Graphics Engine requires two binary semaphores to synchronize its internal states. 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 `LLUI_DISPLAY_IMPL_initialize` function). Two distinct functions have to be implemented to *take* and *give* a binary semaphore.
- `LLUI_DISPLAY_IMPL_flush` : According the display buffer mode (see *Display Configurations*), the `flush` function has to be implemented. This function must not be blocking and not performing the copy directly. Another OS task or a dedicated hardware must be configured to perform the buffer copy.

Optional Low Level API

Several optional Low Level API 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 constrast, display characteristics (is colored display, double buffer), colors conversions (see *Pixel Structure* and *CLUT*), etc. Refer to each function comment to have more information about the default behavior.

Painter Low Level API

All MicroUI drawings (available in `Painter` class) are calling a native function. The MicroUI native drawing functions are listed in `LLUI_PAINTER_impl.h` . The implementation must take care about a lot of constraints: synchronization between drawings, Graphics Engine notification, MicroUI `GraphicsContext` clip and colors, flush dirty area, etc. The principle of implementing a MicroUI drawing function is described in the chapter *Drawing Native*.

An implementation of `LLUI_PAINTER_impl.h` is already available on MicroEJ Central Repository. This implementation respects the synchronization between drawings, the Graphics Engine notification, reduce (when possible) the MicroUI `GraphicsContext` clip constraints and update (when possible) the flush dirty area. This implementation does not perform the drawings. It only calls the equivalent of drawing available in `ui_drawing.h`. This allows to simplify how to use a GPU (or a third-party library) to perform a drawing: the `ui_drawing.h` implementation has just to take in consideration the MicroUI `GraphicsContext` clip and colors and flush dirty area. 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 Graphics Engine (in *weak* mode). This allows to implement only the functions the GPU is able to perform. For a given drawing, the weak function implementation is calling the equivalent of drawing available in `ui_drawing_soft.h`. This file lists all drawing functions implemented by the Graphics Engine.

The Graphics Engine implementation of `ui_drawing_soft.h` is performing the drawings in software. However some drawings can call another `ui_drawing.h` function. For instance `UI_DRAWING_SOFT_drawHorizontalLine` is calling `UI_DRAWING_fillRectangle` in order to use a GPU if available. If not available, the weak implementation of `UI_DRAWING_fillRectangle` is calling `UI_DRAWING_SOFT_fillRectangle` and so on.

The BSP implementation is also allowed to call `ui_drawing_soft.h` algorithms, one or several times per function to implement. For instance, a GPU may be able to draw an image whose format is RGB565. But if the image format is ARGB1555, BSP implementation can call `UI_DRAWING_SOFT_drawImage` function.

Graphics Engine API

The Graphics Engine provides a set of functions to interact with the C archive. The functions allow to retrieve some drawing characteristics, synchronize drawings between them, notify the end of flush and drawings, etc.

The functions are available in `LLUI_DISPLAY.h`.

Drawing Native

As explained before, MicroUI implementation provides a dedicated header file which lists all MicroUI Painter drawings native function. The implementation of these functions has to respect several rules to not corrupt the MicroUI execution (flickering, memory corruption, unknown behavior, etc.). These rules are already respected in the CCO available in MicroEJ Central Repository. In addition, MicroUI allows to add some custom drawings. The implementation of MicroUI Painter native drawings should be used as model to implement the custom drawings.

All native functions must have a `MICROUI_GraphicsContext*` as parameter (often first parameter). This identifies the destination target: the MicroUI GraphicsContext. This target is retrieved in MicroEJ application calling the method `GraphicsContext.getSNIContext()`. This method returns a byte array which is directly mapped on the `MICROUI_GraphicsContext` structure in MicroUI native drawing function declaration.

A graphics context holds a clip and the drawer is not allowed to perform a drawing outside this clip (otherwise the behavior is unknown). Note the bottom-right coordinates might be smaller than 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()`.

Note: Several clip functions are available in `LLUI_DISPLAY.h` to check if a drawing fits the clip.

The Graphics Engine requires the synchronization between the drawings. To do that, it requires a call to `LLUI_DISPLAY_requestDrawing` at the beginning of native function implementation. This function takes as parameter the graphics context and the pointer on the native function itself. This pointer must be casted in a `SNI_callback`.

The drawing function must update the next Display.flush() area (dirty area). If not performed, the next call to Display.flush() will not call `LLUI_DISPLAY_IMPL_flush()` function.

The native function implementation pattern is:

```
void Java_com_mycompany_MyPainterClass_myDrawingNative(MICROUI_GraphicsContext* gc, ...)
{
   // tell to the Graphics Engine if drawing can be performed
   if (LLUI_DISPLAY_requestDrawing(gc, (SNI_callback)&Java_com_mycompany_MyPainterClass_
↪myDrawingNative))
   {
      DRAWING_Status status;

      // perform the drawings (respecting clip if not disabled)
      [...]

      // update new flush dirty area
      LLUI_DISPLAY_setDrawingLimits(gc, ...);

      // set drawing status
      LLUI_DISPLAY_setDrawingStatus(DRAWING_DONE); // or DRAWING_RUNNING;
   }
   // else: refused drawing
}
```

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 recommanded. 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).

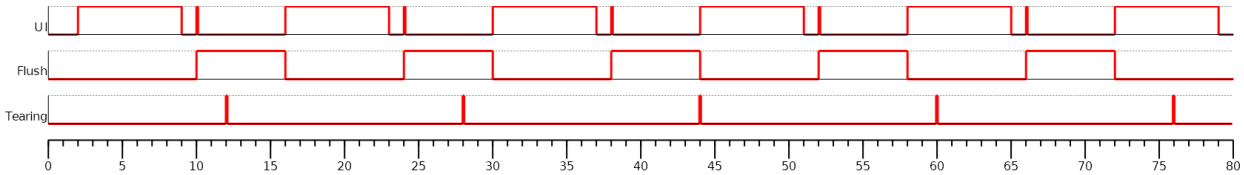
Captions definition:

- UI: It is the UI task which 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 *copy* mode, it is the time to transfer the content of back buffer to display buffer. In *switch* mode, it is the time to swap back and display buffers (often instantaneous) and the time to recopy the content of new display buffer to new back buffer. During this time, the back buffer is *in use* and UI task has to wait the end of copy before starting a new drawing.
- Tearing: The peaks show the tearing signals.
- Rendering frequency: the frequency between the start of a drawing to the end of flush.

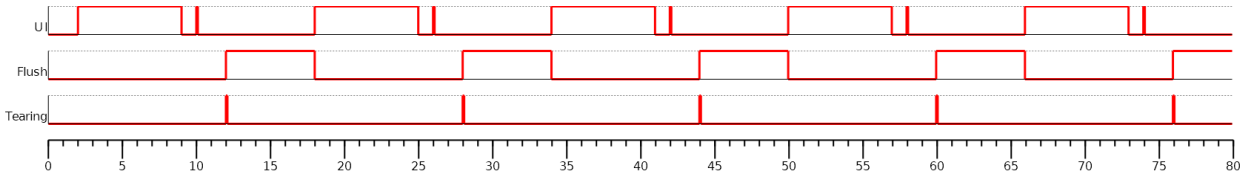
Tearing Signal

In this example, the drawing time is 7ms, the time between the end of 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 in

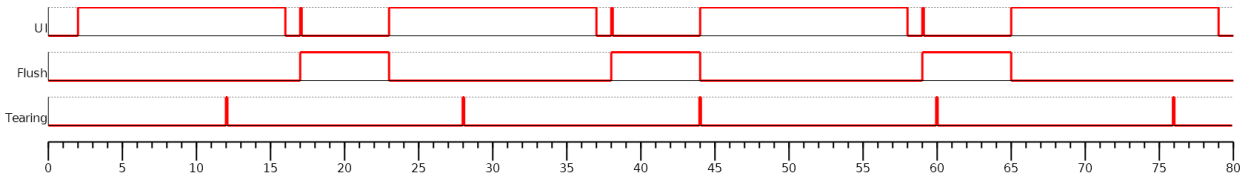
consideration. By consequence the display content is refreshed during the display refresh time. The content can be corrupted: flickering, glitches, etc. The rendering frequency is faster than display refresh rate.



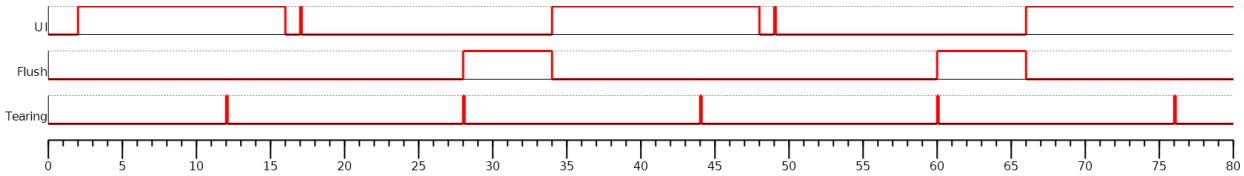
In this example, the times are identical to previous example. The tearing signal is used to start the flush in respecting the display refreshing time. The rendering frequency becomes smaller: it is cadenced on the tearing signal, every 16ms (62.5Hz). During 2ms, the CPU can schedule other tasks or goes in idle mode. The rendering frequency is equal to display refresh rate.



In this example, the drawing time is 14ms, the time between the end of 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 in consideration.



In this example, the times are identical to previous example. The tearing signal is used to start the flush in respecting the display refreshing time. The drawing time + flush time is higher than 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 goes in idle mode. The rendering frequency is equal to display refresh rate divided by two.

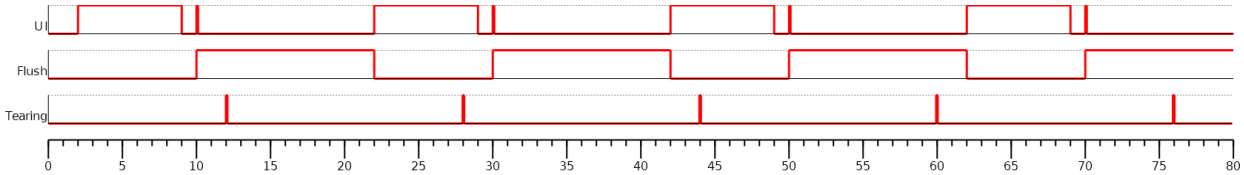


Additional Buffer

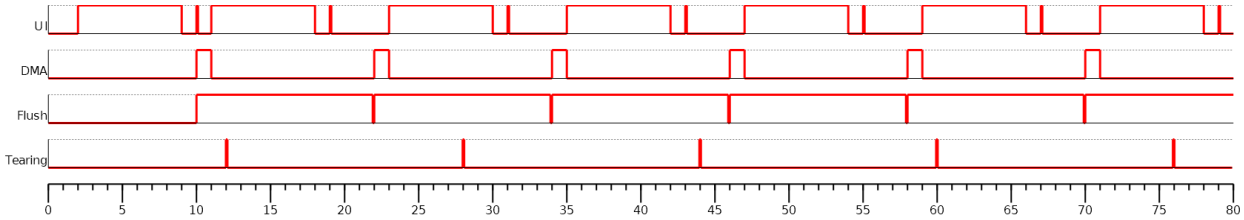
Some devices take a lot of time to send back buffer content to display buffer. The following examples demonstrate the consequence on rendering frequency. The use of an additional buffer optimizes this frequency, however it uses a lot of RAM memory.

In this example, the drawing time is 7ms, the time between the end of 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

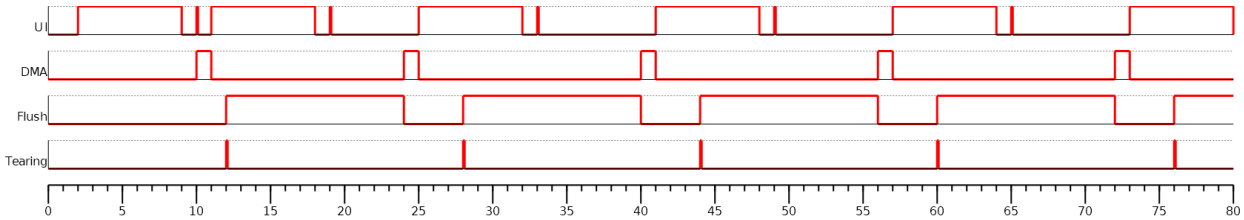
after the call to Display.flush() and the next drawing starts just after the end of flush. Tearing signal is not taken in consideration. The rendering frequency is cadenced on drawing time + flush time.



As mentionned above, the idea is to use two back buffers. First, UI task is drawing in back buffer A . Just after the call to Display.flush(), the flush can start. At same moment, the content of back buffer A is copied in back buffer B (use a DMA, copy time is 1ms). During the flush time (copy of back buffer A to display buffer), the back buffer B can be used by UI task to continue the drawings. When the drawings in back buffer B are done (and after call to Display.flush()), the DMA copy of back buffer B to back buffer A cannot start: the copy can only start when the flush is fully done because the flush is using the back buffer A . As soon as the flush is done, a new flush (and DMA copy) can start. The rendering frequency is cadenced on flush time, ie 12ms (83.3Hz).



The previous example doesn't take in consideration the display tearing signal. With tearing signal and only one back buffer, the frequency is cadenced on two tearing signals (see previous chapter). 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 consider the display frequency is 62.5Hz (16ms).
- *Drawing time* is the time let to 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 1ms.
- *FPS* and *CPU load* are calculated from examples times.
- *Max drawing time* is the maximum time let to the application to perform its drawings, without overlapping next display tearing signal (when tearing is enabled).

Tearing	Nb buffers	Drawing time (ms)	Flush time (ms)	DMA copy 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	1	83.3	66.7	
yes	2	7+1	12	1	62.5	50	11

GPU Synchronization

When a GPU is used to perform a drawing, the caller (MicroUI painter native method) returns immediately. This allows the MicroEJ application to perform other operations during the GPU rendering. However, as soon as the MicroEJ 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 considers a drawing is fully done when it starts a new one. The end of GPU drawing must notify the Graphics Engine calling `LLUI_DISPLAY_drawingDone()` .

Antialiasing

Fonts

The antialiasing mode for the fonts concerns only the fonts with more than 1 bit per pixel (see *Font Generator*).

Background Color

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 MicroEJ 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).

CLUT

The Display module allows to target display which uses a pixel indirection table (CLUT). This kind of display are considered as generic but not standard (see *Pixel Structure*). It consists to store color indices in image memory buffer instead of colors themselves.

Color Conversion

The driver must implement functions that convert MicroUI's standard 32 bits 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. First solution is often quite difficult 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 has only to use 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 current color or with the current application background color (when enabled). This blending *creates* some intermediate colors which are managed by the display driver.

Most of 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 Low Level 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 parameter the pointers on the both ARGB colors. The display driver should replace the ARGB colors by the CLUT indices. Then the Graphics Engine will only use between both indices.

For instance, when the returned indices are `20` and `27`, the display stack will use the indices `20` to `27`, where all indices 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.
- Application can only use foreground colors provided by the CLUT. The platform 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 CLUT range.

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 MicroEJ application may not use all MicroUI image drawings 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 platform 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 formats and convert it in ARGB8888.

Table 13: 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 platform 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 platform as usual.
2. Copy platform file `[platform]/source/link/display_image_x.lscf` in platform configuration project: `[platform configuration project]/dropins/link/`. `x` is a number which characterizes the display pixel format (see *Pixel Structure*). See next warning.
3. Perform some changes into the copied file (see after).
4. Rebuild the platform: the *dropins* file is copied in the platform instead of the original one.

> **Warning:** When the display format in `[platform configuration project]/display/display.properties` changes, the linker file suffix changes too. Perform again all operations in new file with 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).

Library ej.api.Drawing

This library is a Foundation Library which provides additional drawings API. This library is fully integrated in Display module. It requires an implementation of its Low Level API: LLDW_PAINTER_impl.h . These functions are implemented in the same CCO as LLUI_PAINTER_impl.h : com.microej.clibrary.llimpl#microui-drawings . Like MicroUI painter's natives, the functions are redirected to dw_drawing.h . A default implementation of these functions is available in Software Algorithms module (in weak). This allows the BSP to override one or several APIs.

Dependencies

- MicroUI module (see *MicroUI*)
- LLUI_DISPLAY_impl.h implementation if standard or custom implementation is chosen (see *Dependencies* and *LLUI_DISPLAY: Display*).

Installation

The Display module is a sub-part of the MicroUI library. When the MicroUI module is installed, the Display module must be installed in order to be able to connect the physical display with the MicroEJ Platform. If not installed, the *stub* module will be used.

In the platform configuration file, check UI > Display to install the Display module. When checked, the properties file display > display.properties is required during platform creation to configure the module. This configuration step is used to choose the kind of implementation (see *Dependencies*).

The properties file must / can contain the following properties:

- bpp [mandatory]: Defines the number of bits per pixels the display device is using to render a pixel. Expected value is one among these both list:

  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, 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-pixels (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 on the same column. In first case, when the end of line is reached, the next byte contains the first pixels of next line. In second case, when the end of column is reached, the next byte contains the first pixels of next column. In both cases, a new line or a new column restarts with a new byte, even if it remains some free bits in 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 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 than 8, this property is useless and ignored.

- `memoryLayout` [optional, 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 same column. In first case, when the end of line is reached, the next group of pixels is the first group of next line. In second case, when the end of column is reached, the next group of pixels is the first group of 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 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 than 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 images 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 *MicroEJ Format: GPU* and *Custom MicroEJ Format*. Allowed values are 1, 2, 4, 8, 16, 32, 64, 128 and 256.
- `imageHeap.size` [optional, default value is "not set"]: Defines the images heap size. Useful to fix a platform heap size when building a firmware in command line. When using a MicroEJ launcher, the size set in this launcher is priority to the platform value.

Use

The MicroUI Display APIs are available in the class ej.microui.display.Display.

4.9.8 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).

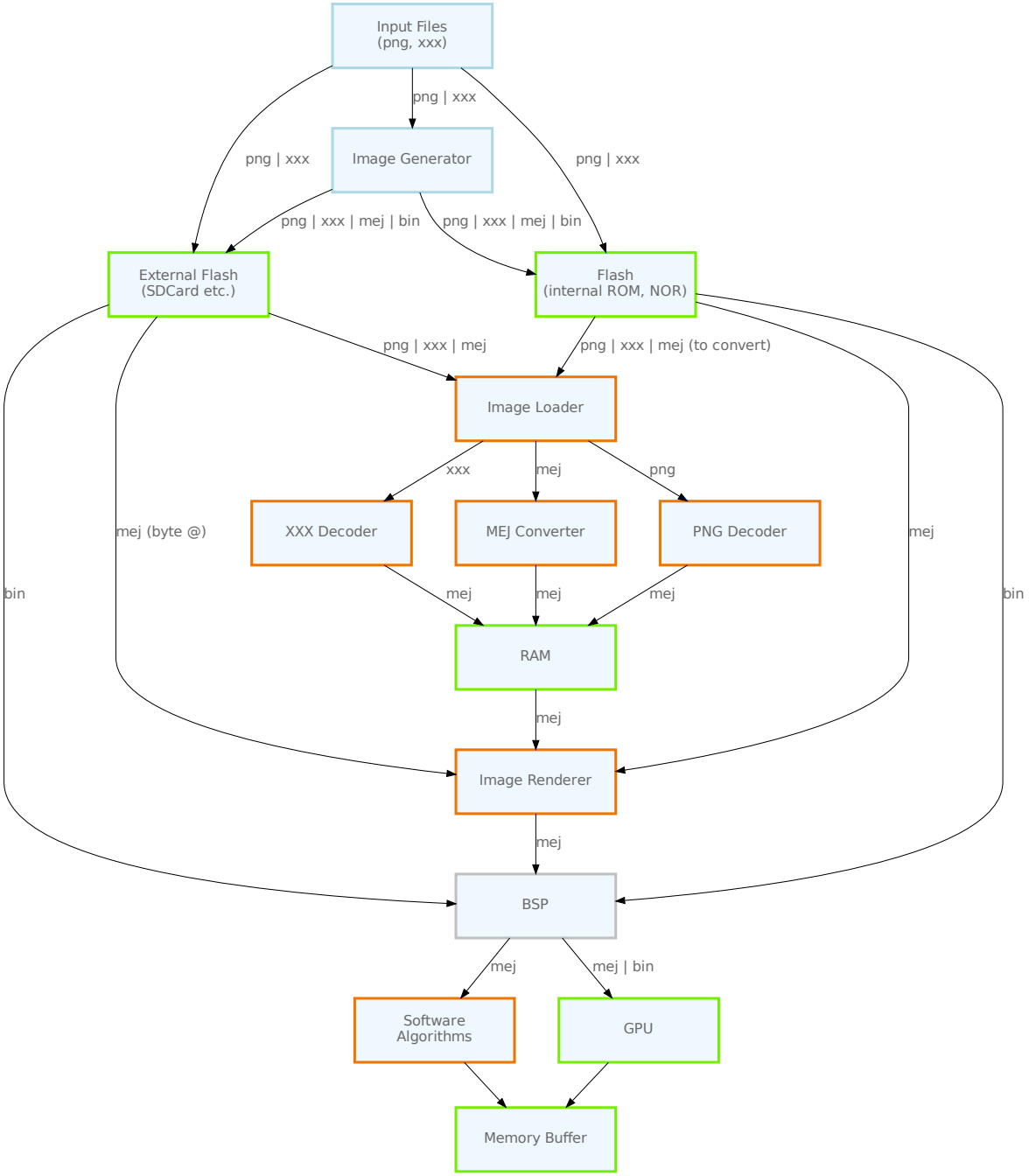
The second kind of image requires the Image Engine to be able to use (load, read and draw) an image referenced by its path with or without any loading step. When the image is understandable by the Image Engine without any loading step, the image is considered like the first kind of image (fast *open* 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 pixels. 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 platform 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 platform 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 MicroEJ Application. The files' data are linked into the FLASH memory.
3. When the MicroEJ 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 MicroEJ 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.

Image Format

The Image Engine makes the distinction between the *input formats* (how an image is encoded) and the *output formats* (how the image is used by the platform and/or the Image Renderer).The Image Engine manages several standard formats in input: PNG, JPEG, BMP, etc. In addition, an input format may be custom (platform dependant, unsupported image format by default). It manages two formats in output: the MicroEJ format (known by the Image Renderer) and the binary format.

Each Image Engine can manage one or several input formats. However the Image Renderer manages only the MicroEJ format (*MicroEJ Format: Standard*, *MicroEJ Format: Display* and *MicroEJ Format: GPU*). The binary output format (*Binary Format*)is fully platform dependant and can be used to encode some images which are not usable by MicroUI standard API.

MicroEJ Format: Standard

Several MicroEJ format encodings are available. Some encodings may be directly managed by the display driver. Refers to the platform specification to retrieve the list of better formats.

Advantages:

- The pixels layout and bits 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).
- Supports or not the alpha encoding: select the better format according to the image to encode.

Disadvantages:

- No compression: the image size in bytes is proportional to the number of pixels, the transparency, and the number of bits-per-pixel.
- Slower than `display` format when the display driver does not recognize the format: a pixel conversion is required at runtime.

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 number of bits-per-pixels of MicroEJ format:

```
required_memory = header + (image_width * image_height) * bpp / 8;
```

The pixels array is stored after the MicroEJ image file header. A padding between the header and the pixels array is added to force to start the pixels array at a memory address aligned on number of bits-per-pixels.

Select one the following format to use a generic format among this list: `ARGB8888` , `RGB888` , `ARGB4444` , `ARGB1555` , `RGB565` , `A8` , `A4` , `A2` , `A1` , `C4` , `C2` , `C1` , `AC44` , `AC22` and `AC11` . The following snippets describe the color conversion for each format:

- ARGB8888: 32 bits format, 8 bits for transparency, 8 per color.

```
u32 convertARGB8888toRAWFormat(u32 c){
    return c;
}
```

- RGB888: 24 bits format, 8 per color. Image is always fully opaque.

```
u32 convertARGB8888toRAWFormat(u32 c){
    return c & 0xffffff;
}
```

- ARGB4444: 16 bits 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 bits 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)
            ;
}
```

- RGB565: 16 bits format, 5 or 6 per color. Image is always fully opaque.

```
u32 convertARGB8888toRAWFormat(u32 c){
    return 0
            | ((c & 0xf80000) >> 8)
            | ((c & 0x00fc00) >> 5)
            | ((c & 0x0000f8) >> 3)
```

(continues on next page)

(continued from previous page)

```
            ;
}
```

- A8: 8 bits format, only transparency is encoded. The color to apply when drawing the image, is the current GraphicsContext color.

```
u32 convertARGB8888toRAWFormat(u32 c){
    return 0xff - (toGrayscale(c) & 0xff);
}
```

- A4: 4 bits format, only transparency is encoded. The color to apply when drawing the image, is the current GraphicsContext color.

```
u32 convertARGB8888toRAWFormat(u32 c){
    return (0xff - (toGrayscale(c) & 0xff)) / 0x11;
}
```

- A2: 2 bits format, only transparency is encoded. The color to apply when drawing the image, is the current GraphicsContext color.

```
u32 convertARGB8888toRAWFormat(u32 c){
    return (0xff - (toGrayscale(c) & 0xff)) / 0x55;
}
```

- A1: 1 bit format, only transparency is encoded. The color to apply when drawing the image, is the current GraphicsContext color.

```
u32 convertARGB8888toRAWFormat(u32 c){
    return (0xff - (toGrayscale(c) & 0xff)) / 0xff;
}
```

- C4: 4 bits format with grayscale conversion. Image is always fully opaque.

```
u32 convertARGB8888toRAWFormat(u32 c){
    return (toGrayscale(c) & 0xff) / 0x11;
}
```

- C2: 2 bits format with grayscale conversion. Image is always fully opaque.

```
u32 convertARGB8888toRAWFormat(u32 c){
    return (toGrayscale(c) & 0xff) / 0x55;
}
```

- C1: 1 bit format with grayscale conversion. Image is always fully opaque.

```
u32 convertARGB8888toRAWFormat(u32 c){
    return (toGrayscale(c) & 0xff) / 0xff;
}
```

- AC44: 4 bits for transparency, 4 bits with grayscale conversion.

```
u32 convertARGB8888toRAWFormat(u32 c){
    return 0
        | ((color >> 24) & 0xf0)
        | ((toGrayscale(color) & 0xff) / 0x11)
```

(continues on next page)

(continued from previous page)

```
        ;
}
```

- 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)
        ;
}
```

The pixels order in MicroEJ file follows this rule:

```
pixel_offset = (pixel_Y * image_width + pixel_X) * bpp / 8;
```

MicroEJ Format: Display

The display can hold a pixel encoding which is not standard (see *Pixel Structure*). The MicroEJ format can be customized to encode the pixel in same encoding than display. The number of bits-per-pixels and the pixel bits organisation 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 in order to free up some memory space.

Note: From Image Engine point of view, the format stays a MicroEJ format, readable by the Image Renderer.

Advantages:

- Encoding is identical to display encoding.
- Drawing an image is often very fast (simple memory copy when the display pixel encoding does not hold the opacity level).

Disadvantages:

- No compression: the image size in bytes is proportional to the number of pixels. The required memory is similar to *MicroEJ Format: Standard*.

MicroEJ Format: GPU

The MicroEJ format may be customized to be platform's GPU compatible. It can be extanded 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 platform dependant header, located between MicroEJ format header (start of file) and pixels array. The MicroEJ format is designed to let the platform encodes and decodes this additional header. For Image Engine software algorithms, this header is useless and never used.

Note: From Image Engine point of view, the format stays a MicroEJ format, readable by the Image Engine Renderer.

Advantages:

- Encoding is recognized by the GPU.
- 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 there is no custom header.

When MicroEJ format holds another header (called `custom_header`), the required memory depends is:

```
required_memory = header + custom_header + (image_width * image_height) * bpp / 8;
```

The row stride allows to add some padding at the end of each line in order to start 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 with the image width: `row_stride_in_bytes = image_width * bpp / 8`. It can be customized at image buffer creation thanks to the Low Level API `LLUI_DISPLAY_IMPL_getNewImageStrideInBytes`. The required RAM memory becomes:

```
required_memory = header + custom_header + row_stride * image_height;
```

MicroEJ Format: RLE1

The Image Engine can display embedded images that are encoded into a compressed format which encodes several consecutive pixels into one or more 16-bit words. This encoding only manages fully opaque and fully transparent pixels.

- Several consecutive pixels have the same color (2 words).
  - First 16-bit word specifies how many consecutive pixels have the same color (pixels colors converted in RGB565 format, without opacity data).
  - Second 16-bit word is the pixels' color in RGB565 format.
- Several consecutive pixels have their own color (1 + n words).
  - First 16-bit word specifies how many consecutive pixels have their own color.
  - Next 16-bit word is the next pixel color.
- Several consecutive pixels are transparent (1 word).
  - 16-bit word specifies how many consecutive pixels are transparent.

  - Not designed for images with many different pixel colors: in such case, the output file size may be larger than the original image file.

Advantages:

- Supports fully opaque and fully transparent encoding.
- Good compression when several consecutive pixels respect one of the three previous rules.

Disadvantages:

- Drawing an image is slightly slower than when using Display format.

The file format is quite similar to *MicroEJ Format: Standard*.

Binary Format

This format is not compatible with the Image Renderer and by MicroUI. It is can be used by MicroUI addon libraries which provide their own images managements.

Advantages:

- Encoding is known by platform.
- Compression is inherent to the format itself.

Disadvantages:

- This format cannot be used to target a MicroUI Image (unsupported format).

Without Compression

An image can be embedded without any conversion / compression. This allows to embed the resource as it is, in order 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.

Advantages:

- Conserves the image characteristics.

Disadvantages:

- Requires an image runtime decoder.
- Requires some RAM in which to store the decoded image in MicroEJ format.

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 platform (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

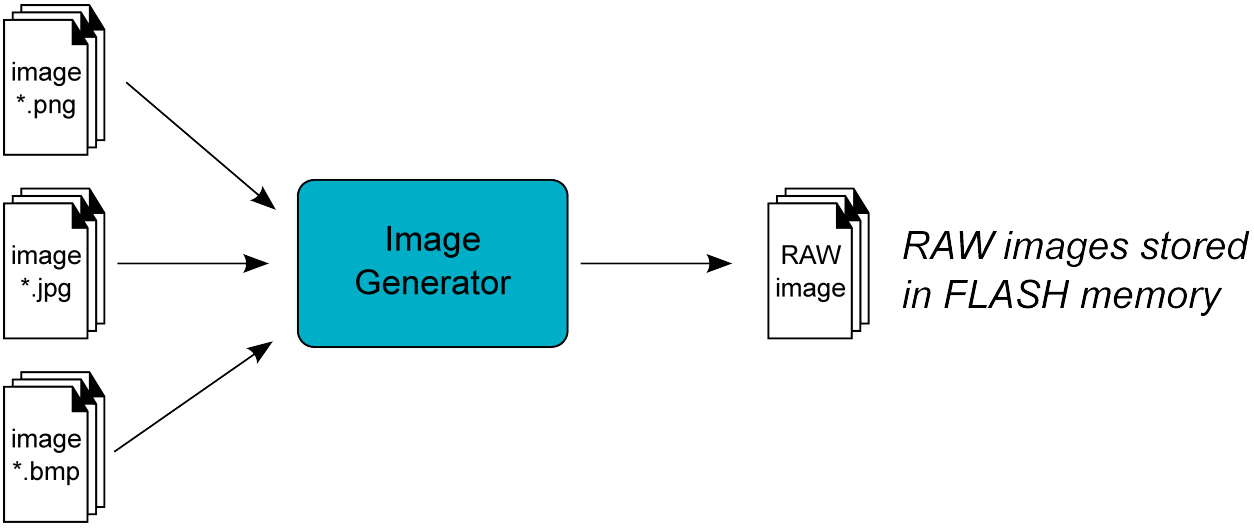


Fig. 38: Image Generator Principle

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 MicroEJ 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 platform, 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 platform 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 platform characteristics and restrictions, see *MicroEJ Format: Display* and *MicroEJ Format: GPU*) 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 platform, 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 platform. As described above this extension will be able to:

- read more input image file formats,
- extand the MicroEJ format with platform 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 platform. 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 `imageGeneratorMyPlatform`).
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 platform (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.is2t.microui" module="imageGeneratorMyPlatform" 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 platform"/>
      <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"/>
```

(continues on next page)

(continued from previous page)

```
      <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="13.0.0">
         <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 easyant project.
5. Copy the generated jar: `target~/artifacts/imageGeneratorMyPlatform.jar` in the platform configuration project folder: `MyPlatform-configuration/dropins/tools/`
6. Rebuild the platform.

Warning: The dropins folder must be updated (and platform built again) after any changes in the image generator extension project.

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.

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 platform configuration project and rebuild the platform (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.

Custom MicroEJ Format

As mentionned above (*MicroEJ Format: Display* and *MicroEJ Format: GPU*), the MicroEJ format can be extanded by notions specific to the platform (and often to the GPU the platform is using). The generated file stays a MicroEJ file format, usable by the Image Renderer. Additionally, the file becomes compatible with the platform 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 platform'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 platform configuration project and rebuild the platform (see above).

If the only constraint is the pixels array aligment, the Image Generator extension is not useful:

1. Open platform configuration file `display/display.properties`.
2. Add the property `imageBuffer.memoryAlignment`.
3. Build again the platform.

This alignment will be used by the Image Generator and also by the Image Loader.

Platform Binary Format

As mentionned above (*Binary Format*), the Image Generator is able to generate a binary file compatible with platform (and not compatible with Image Renderer). This is very useful when a platform library offers the possibility to use other kinds of images than MicroUI library. The binary file can be encoded according to the options the user gives 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 down the name of created class, with its package and classname.
5. Rebuild the image generator extension, copy it in platform configuration project and rebuild the platform (see above).

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 MicroEJ Application launcher (see *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 3: 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 4: 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 5: GPU Output Format Examples

```
image1:ARGB8888
image2:RGB565
image3:A4
```

- MicroEJ RLE1 output format: to encode the image in RLE1 format, specify `RLE1` as output format:

Listing 6: RLE1 Output Format Example

```
image1:RLE1
```

- Without Compression: to keep original file, do not specify any format:

Listing 7: Unchanged Image Example

```
image1
```

- Binary format: to encode the image in a format only known by the platform, refer to the platform documentation to know which format are available.

Listing 8: Binary Output Format Example

```
image1:XXX
```

Linker File

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 MicroEJ 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 with 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.

Dependencies

- Image Renderer module (see *Image Renderer*).
- Display module (see *Display*): This module gives the characteristics of the graphical display that are useful to configure the Image Generator.

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 platform 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 *Application Options* (`Libraries` > `MicroUI` > `Image`) for more information about specifying the image configuration file.

Image Loader

Principle

The Image Loader module is an on-board engine 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 memory loader),
- converts images in MicroEJ format,
- creates a runtime buffer to manage MicroUI BufferedImage,

- 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 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.

Memory

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: *MicroEJ Format: GPU*.

The heap size is application dependant. In MicroEJ 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`.

BufferedImage

MicroUI application is able to create an image where it is allowed to draw into: the MicroUI `BufferedImage`. The image format is the same than the display format; in other words, its number of bits-per-pixel and its pixel bits organization are the same. The display pixel format can be standard or custom (see *Pixel Structure*). To create this kind of image, the Image Loader has just to create a buffer in RAM whose size depends on the image size (see *MicroEJ Format: Display*).

External Resource

An image is retrieved by its path (except for `BufferedImage`). The path describes a location in application classpath. The resource may be generated at same time than application (internal resource) or be external (external resource). The Image Loader is able to 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 next chapters). The RAM section used to load the external image is automatically freed when the Image Loader do not need it again.

The image may be located in external memory but be available in CPU addresses' space ranges (byte-adressable). In this case the Image Loader considers the image as *internal* and does not need to copy its content in RAM memory.

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 specifiying 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 located in external memory, the Image Loader has to copy the image into RAM memory to be usable by Image Renderer.

Encoded Image

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 located in an external memory). 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.

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 the RAW image format and size.
- Decodes the image in the allocated buffer.
- Waiting the end of decoding step before returning.

Dependencies

- Image Renderer module (see *Image Renderer*)

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 platform 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.

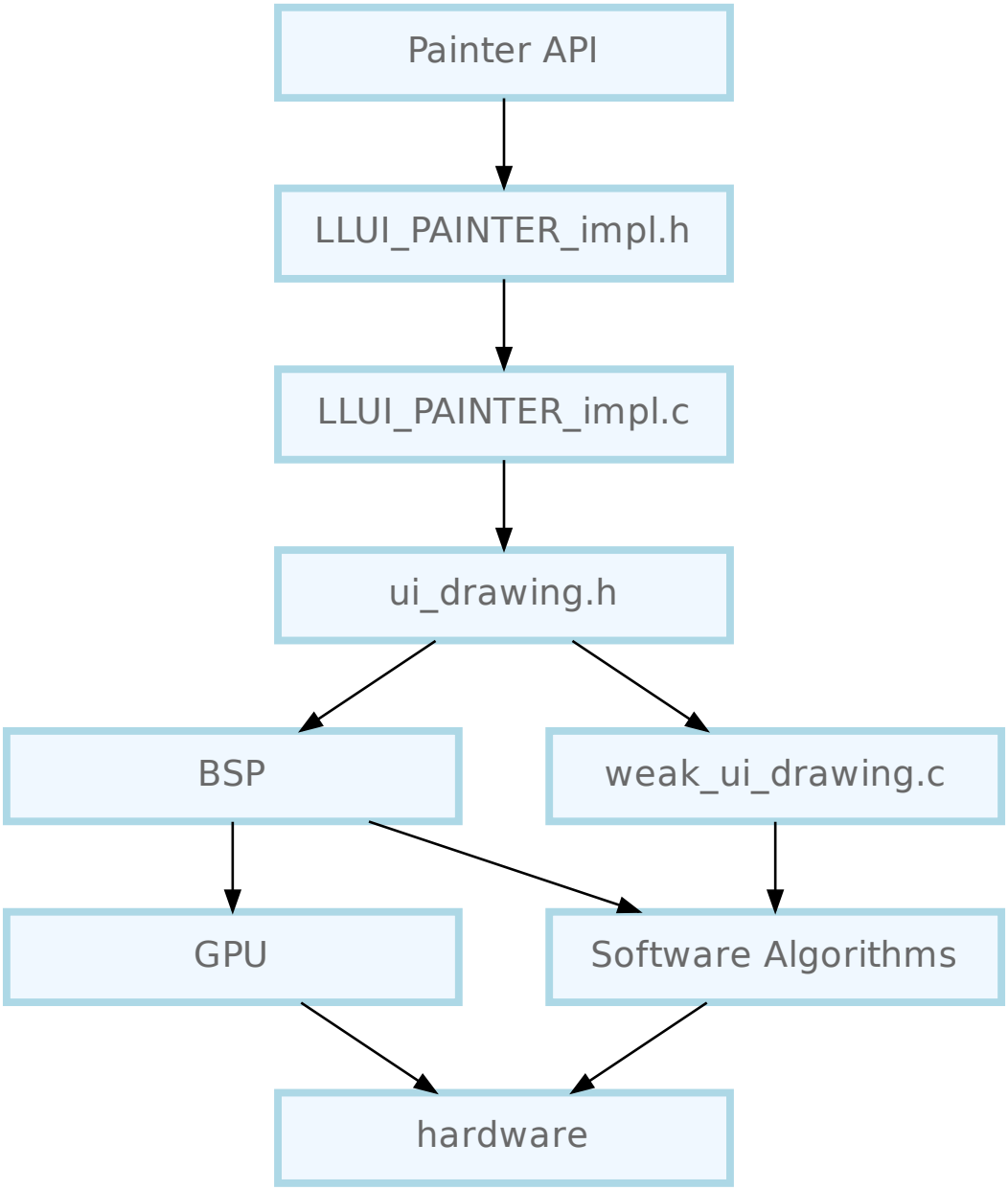
Image Renderer

Principle

The Image Renderer is an on-board engine that reads and draws the image encoded in MicroEJ format (see *Image Format*) . It calls Low LevelLow Level APIs to draw and transform the images (rotation, scaling, deformation, etc.). It also includes software algorithms to perform the rendering.

Functional Description

The engine redirects all MicroUI images drawings to a set of Low Level API. All Low Level API are implemented by weak functions which call software algorithms. The BSP has the possibility to override this default behavior for each Low Level API independently. Furthermore, the BSP can override a Low Level API for a specific MicroEJ format (for instance `ARGB8888`) and call the software algorithms for all other formats.



Dependencies

- MicroUI module (see *MicroUI*),
- Display module (see *Display*).

Installation

Image Renderer module is part of the MicroUI module and Display module. Install them in order to be able to use some images.

Use

The MicroUI image APIs are available in the class `ej.microui.display.Image`.

4.9.9 Fonts

Overview

Principle

The Font Engine is composed of:

- A "Font Designer" module: a graphical tool which runs within the MicroEJ IDE used to build and edit MicroUI fonts; it stores fonts in a platform-independent format. See *Font Designer*.
- A "Font Generator" module, for converting fonts from the platform-independent format into a platform-dependent format.
- The "Font Renderer" module which decodes and renders at application runtime the platform-dependent fonts files generated by the "Font Generator".

The three modules are complementary: a MicroUI font must be created and edited with the Font Designer before being integrated as a resource by the Font Generator. Finally the Font Renderer uses the generated fonts at runtime.

Functional Description

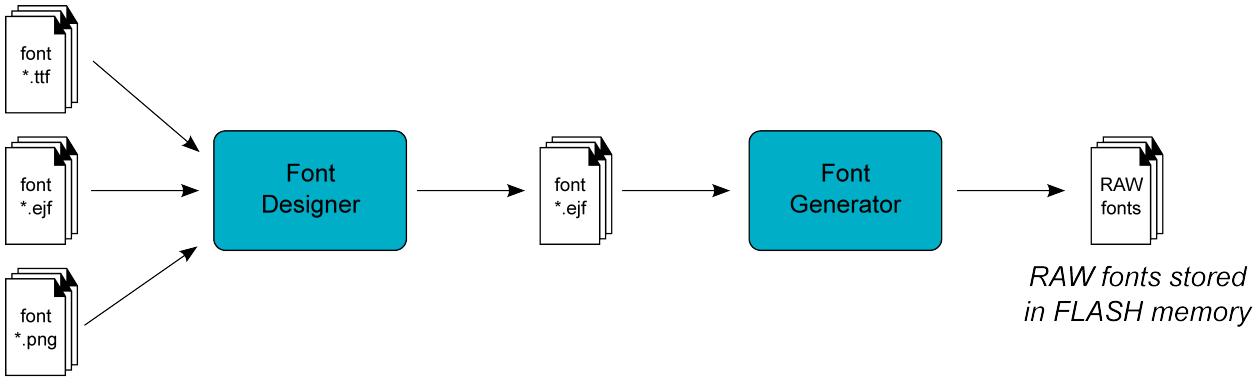


Fig. 39: Font Generation

Process overview:

1. User uses the Font Designer module to create a new 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.
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 MicroEJ Application. The raw files' data are linked into the FLASH memory.
6. When the MicroEJ Application creates a MicroUI Font object which targets a pre-generated image, 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.

Font Characteristics

Font Format

The Font Engine provides fonts that conform to the Unicode Standard. The `.ejf` files hold font properties:

- Identifiers: Fonts hold at least one identifier that can be one of the predefined Unicode scripts or a user-specified identifier. The intention is that an identifier indicates that the font contains a specific set of character codes, but this is not enforced.
- 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.

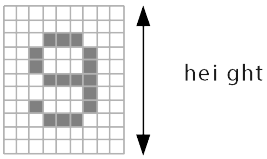


Fig. 40: Font Height

- 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.

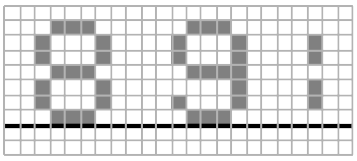


Fig. 41: Font baseline

- 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.

Multiple filters may apply at the same time, combining their transformations on the displayed characters.

Pixel Transparency

The Font Renderer renders the font according the 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^n 1`. The value of each color component of the final pixel is equal to:

$$foreground * pixelValue/pmax + background * (pmax - pixelValue)/pmax$$

Language

Supported Languages

The Font Renderer manages the Unicode basic multilingual languages, whose characters are encoded on 16-bit, i.e. Unicodes from 0x0000 to 0xFFFF. It allows to render left-to-right or right-to-left writing systems: Latin (English, etc.), Arabic, Chinese, etc. are some supported languages. Note that the rendering is always performed left-to-right, even if the string are written right-to-left. There is no support for top-to-bottom writing systems. Some languages require diacritics and contextual letters; the Font Renderer manages simple rules in order to combine several characters.

Arabic Support

The Font Renderer manages the ARABIC font specificities: the diacritics and contextual letters.

To render an Arabic text, the Font Renderer requires several points:

- To determinate if a character has to overlap the previous character, the Font Renderer uses a specific range of ARABIC characters: from `0xfe70` to `0xfefc`. All other characters (ARABIC or not) outside this range are considered *classic* and no overlap is performed. Note that several ARABIC characters are available outside this range, but the same characters (same representation) are available inside this range.
- The application strings must use the UTF-8 encoding. Furthermore, in order to force the use of characters in the range `0xfe70` to `0xfefc`, the string must be filled with the following syntax: '`\ufee2\ufedc\ufe91\u0020\ufe8e\ufe92\ufea3\ufeae\ufee3`'; where `\uxxxx` is the UTF-8 character encoding.
- The application string and its rendering are always performed from left to right. However the string contents are managed by the application itself, and so can be filled from right to left. To write the text:

  مكبا بحرم

  the string characters must be : '`\ufee2\ufedc\ufe91\u0020\ufe8e\ufe92\ufea3\ufeae\ufee3`'. The Font Renderer will first render the character '`\ufee2`', then '`\ufedc`,' and so on.
- Each character in the font (in the `ejf` file) must have a rendering compatible with the character position. The character will be rendered by the Font Renderer as-is. No support is performed by the Font Renderer to obtain a *linear* text.

Font Generator

Principle

The Font Generator module is an off-board tool that generates fonts ready to be displayed without the need for additional runtime memory. It outputs a raw file for each converted font.

Functional Description

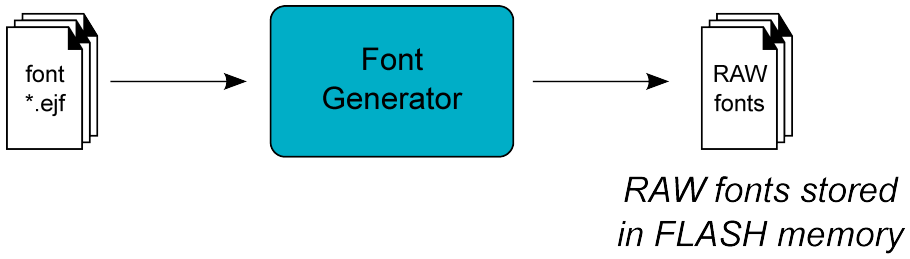


Fig. 42: Font Generator Principle

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 MicroEJ Application. The raw file's data is linked into the FLASH memory.
4. When the MicroEJ Application draws text on the display (or on an image), 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 `.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 illustrates the conversion "grayscale to transparency level". The grayscale value '0x00' is black whereas value '0xff' is white. The transparency level '0x0' is fully transparent whereas level '0x1' (bpp == 1), '0x3' (bpp == 2) or '0xf' (bpp == 4) is fully opaque.

Table 14: Font 1-BPP RAW Conversion

Grayscale Ranges	Transparency Levels
0x00 to 0x7f	0x1
0x80 to 0xff	0x0

Table 15: 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 16: 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 - 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 MicroEJ Application launcher (see *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 9: 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 with the MicroEJ 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.

Dependencies

- Font Renderer module (see *Font Renderer*)

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 MicroEJ Application.

If the module is not installed, the platform user will not be able to embed a new font with his/her MicroEJ Application. He/she will be only able to use the system fonts specified during the MicroUI initialization step (see *Static Initialization*).

In the platform 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 *Application Options* (`Libraries` > `MicroUI` > `Font`) for more information about specifying the fonts configuration file.

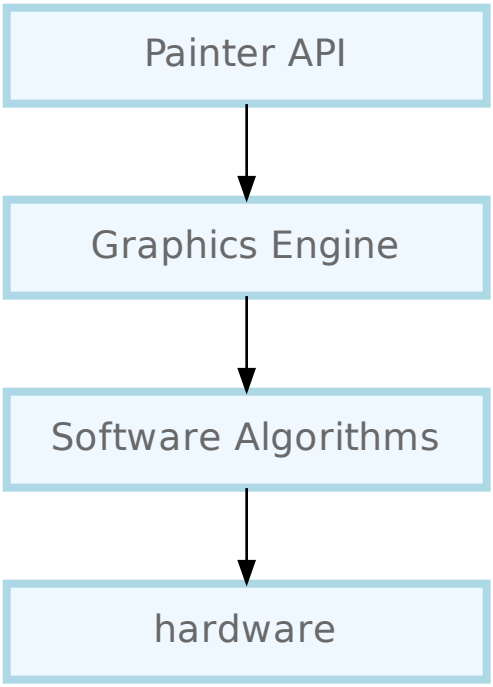
Font Renderer

Principle

The Font Renderer is included in the MicroUI module (see *MicroUI*) for the application side; and is included in the Display module (see *Display*) for the C side.

Functional Description

The Graphics Engine redirects all MicroUI font drawings to the internal software algorithms. There is no indirection to a set of Low Level API.



External Resources

The Font Renderer 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 Renderer copies a very short part of the resource (the font file) into a RAM memory (into CPU addresses space range): the font header. This header stays located in RAM until MicroEJ Application is using the font. As soon as the MicroEJ Application uses another external font, new font replaces the old one. Then, on MicroEJ Application demand, the Font Renderer loads some extra information from the font into the RAM memory (the font meta data, the font pixels, etc.). This extra information is automatically unloaded from RAM when the Font Renderer no longer needs them.

This extra information 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. However it is possible to change this value by setting the MicroEJ 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.

Dependencies

- MicroUI module (see *MicroUI*),

- Display module (see *Display*).

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.

4.9.10 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 MicroEJ Platform (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. 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="widget" rev="1.0.0"/>
   </dependencies>
</ivy-module>
```

It depends at least on the Front Panel framework. This framework contains the Front Panel core classes. The dependencies can be reduced to:

```
<dependencies>
   <dependency org="ej.tool.frontpanel" name="framework" rev="1.1.0"/>
</dependencies>
```

To be compatible with Display module's Graphics Engine, the project must depend on an extension of Front Panel framework. This extension provides some interfaces and classes the Front Panel is using to target simulated display and input devices. The extension does not provide any widgets. It is the equivalent of the embedded Low Level API. It fetches by transitivity the Front Panel framework, so the Front Panel framework dependency does not need to be specified explicitly:

```
<dependencies>
   <dependency org="com.microej.pack.ui" name="ui-pack" rev="13.0.0">
      <artifact name="frontpanel" type="jar"/>
   </dependency>
</dependencies>
```

Warning: This extension is built for each UI pack version. By consequence a Front Panel project is made for a platform built with the same UI pack. When the UI pack mismatch, some errors may occur during the Front Panel project export step, during the platform build and/or during the application runtime.

The Front Panel extension does not provide any widgets. Some compatible widgets are available in a third library. The life cycle of this library is different than the UI pack's one. New widgets can be added to simulate new kind of displays, input devices, etc. This extension fetches by transitivity the Front Panel extension, so this extension dependency does not need to be specified explicitly:

```
<dependencies>
   <dependency org="ej.tool.frontpanel" name="widget" rev="2.0.0"/>
</dependencies>
```

Warning: The minimal version `2.0.0` is required to be compatible with UI pack 13.0.0 and higher. By default, when creating a new Front Panel project, the widget dependency version is `1.0.0`.

Widget Display

By default, a display area is rectangular. Some displays can have another appearance (for instance: circular). The Front Panel is able to simulate that using a filter (see *Widget*) . This filter defines the pixels inside and outside the real display area. The filter image must have the same size than display rectangular area. A display pixel at a given position will be not rendered if the pixel at the same position in mask is fully transparent.

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 *Memory*) 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*). Front Panel can add some additional decoders like the C-side for the embedded platform (see *external image decoders*). However, the exhaustive list of additional decoders is limited (Front Panel is using the Java AWT `ImageIO` API). 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 17: 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
```

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.

Use

Launch a MicroUI application on the Simulator to run the Front Panel.

4.10 Networking

4.10.1 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.

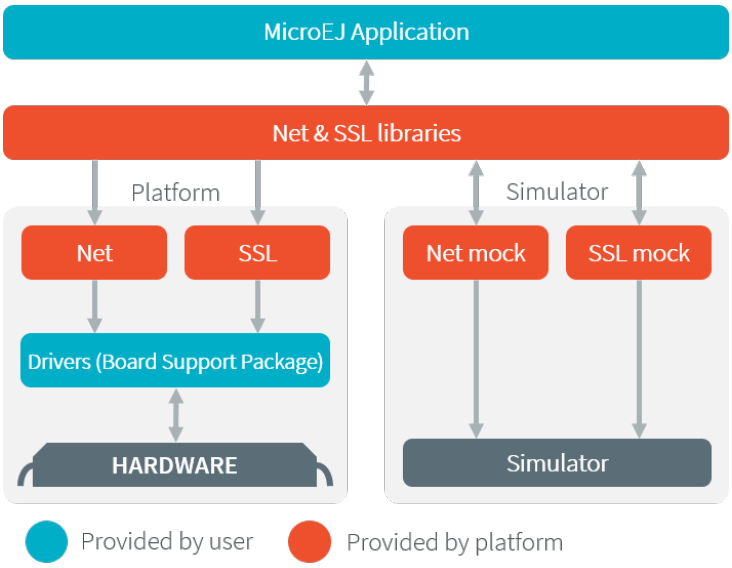


Fig. 43: Overview

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.

4.10.2 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

Network is an additional module. In the platform configuration file, check NET to install this module. When checked, the properties file `net` > `net.properties` is required during platform creation to configure the module. This configuration step is used to customize the kind of TCP/IP native stack used and the Domain Name System (DNS) implementation.

The properties file must / can contain the following properties:

- `stack` [optional, default value is "custom"]: Defines the kind of TCP/IP interface used in the C project.
  - `custom`: Select this configuration to make a "from scratch" implementation glue between the C Network Core Engine and the C project TCP/IP interface.
  - `bsd` : Select this configuration to use a BSD-like library helper to implement the glue between the C Network Core Engine and the C project TCP/IP interface. This property requires that the C project provides a TCP/IP native stack with a Berkeley Sockets API and a `select` mechanism.
- `dns` [optional, default value is "native"]: Defines the kind of Domain Name System implementation used.
  - `native` : Select this configuration to implement the glue between the C Network Core Engine DNS part and the C project TCP/IP interface.
  - `soft` : Select this configuration if you want a software implementation of the DNS part. Only the IPs list of the DNS server must be provided by the C Network Core Engine glue.

Use

The Net API Module must be added to the *module.ivy* of the MicroEJ Application project in order to allow access to the Net library.

```
<dependency org="ej.api" name="net" rev="1.1.1"/>
```

This library provides a set of options. Refer to the chapter *Application Options* which lists all available options.

4.10.3 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. In the platform configuration file, check `SSL` to install the module.

Use

The SSL API module must be added to the *module.ivy* of the MicroEJ Application project, in order to allow access to the SSL library.

```
<dependency org="ej.api" name="ssl" rev="2.2.0"/>
```

4.11 File System

4.11.1 Principle

The FS module 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.

4.11.2 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.

4.11.3 Dependencies

- `LLFS_impl.h` and `LLFS_File_impl.h` implementations (see *LLFS: File System*).

4.11.4 Installation

FS is an additional module. In the platform configuration file, check FS to install it. When checked, the properties file `fs` > `fs.properties` are required during platform creation in order to configure the module.

The properties file must / can contain the following properties:

- `fs` [optional, default value is "Custom"]: Defines the kind of File System native stack used in the C project.
  - `Custom` : select this configuration to make a specific File System portage.
  - `FatFS` : select this configuration to use FatFS native File System-compliant settings.
- `root.dir` [optional, for a FatFS File System. Mandatory, for a Custom File System.]: Defines the native File System root volume (default value is "/" for FatFS).
- `user.dir` [optional, for a FatFS File System. Mandatory, for a Custom File System.]: Defines the native File System user directory (default value is "/usr" for FatFS).
- `tmp.dir` [optional, for a FatFS File System. Mandatory, for a Custom File System.]: Defines the native File System temporary directory (default value is "/tmp" for FatFS).
- `file.separator` [optional, for a FatFS File System. Mandatory, for a Custom File System.]: Defines the native File System file separator (default value is "/" for FatFS).
- `path.separator` [optional, for a FatFS File System. Mandatory, for a Custom File System.]: Defines the native File System path separator (default value is ":" for FatFS).

4.11.5 Use

The FS API Module must be added to the *module.ivy* of the MicroEJ Application project in order to allow access to the FS library.

```
<dependency org="ej.api" name="fs" rev="2.0.6"/>
```

4.12 Hardware Abstraction Layer

4.12.1 Principle

The Hardware Abstraction Layer (HAL) 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.

4.12.2 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.

4.12.3 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;
```

4.12.4 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.

4.12.5 Dependencies

- `LLHAL_impl.h` implementation (see *LLHAL: Hardware Abstraction Layer*).

4.12.6 Installation

HAL is an additional module. In the platform configuration file, check HAL to install the module.

4.12.7 Use

The HAL API Module must be added to the *module.ivy* of the MicroEJ Application project in order to allow access to the ECOM library.

```
<dependency org="ej.api" name="hal" rev="1.0.4"/>
```

4.13 Device Information

4.13.1 Principle

The Device library provides access to the device information. This includes the architecture name and a unique identifier of the device for this architecture.

4.13.2 Dependencies

- `LLDEVICE_impl.h` implementation (see *LLDEVICE: Device Information*).

4.13.3 Installation

Device Information is an additional module. In the platform configuration file, check Device Information to install it. When checked, the property file `device > device.properties` may be defined during platform creation to customize the module.

The properties file must / can contain the following properties:

- `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.

4.13.4 Use

The Device API Module must be added to the *module.ivy* of the MicroEJ Application project in order to allow access to the Device library.

```
<dependency org="ej.api" name="device" rev="1.0.2"/>
```

4.14 Event Tracing

4.14.1 Introduction

A MicroEJ Platform can connect its own implementation of *Event Tracing* by overriding the functions defined in the `LLTRACE_impl.h` file.

MicroEJ also provides an Abstraction Layer 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.

4.14.2 SystemView

Principle

SystemView is a real-time recording and visualization tool for embedded systems that reveals the true 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: SystemView can ensure a system performs as designed, can track down inefficiencies, and show unintended interactions and resource conflicts, with a focus on the details of every single system tick.

A specific SystemView extension made by MicroEJ allows to trace both RTOS tasks and MicroEJ Java threads at the same time.

This section explains how to add SystemView in a MicroEJ Platform and how to connect it to a Board Support Package with FreeRTOS.

Note: MicroEJ Abstraction Layer implementation for SystemView is compatible with FreeRTOS `9` and FreeRTOS `10`.

References

- https://www.segger.com/products/development-tools/systemview/
- https://www.segger.com/downloads/jlink/UM08027

Installation

SystemView consists on installing several items in the BSP. The following steps describe them and must be performed in right order. In case of SystemView is already available in the BSP, apply only modifications made by MicroEJ on SystemView files and SystemView for FreeRTOS files to enable MicroEJ Java threads monitoring.

1. Download and install SystemView: http://segger.com/downloads/systemview/.
2. Apply SystemView for FreeRTOS patch as described in documentation (https://www.segger.com/downloads/jlink/UM08027); patch is available in installation folder `SEGGER\SystemView\Src\Sample\FreeRTOSVxx`.
3. Add `SEGGER\SystemView\Src\Sample\FreeRTOSVxx\Config\SEGGER_SYSVIEW_Config_FreeRTOS.c` in your BSP.
4. Add SystemView for MicroEJ CCO files in your BSP: `com.microej.clibrary.thirdparty.systemview-1.3.0` (or check the differences between pre-installed SystemView and CCO files)
5. Add SystemView for MicroEJ for FreeRTOS CCO files in your BSP: `com.microej.clibrary.thirdparty.systemview-freertosxx-1.1.0` (or check the differences between pre-installed SystemView and CCO files)
6. Install the implementation of MicroJvm monitoring over SystemView by adding CCO files in your BSP: `com.microej.clibrary.llimpl.trace-systemview-2.1.0`
7. Make FreeRTOS compatible with SystemView: open `FreeRTOSConfig.h` and:
   - add `#define INCLUDE_xTaskGetIdleTaskHandle 1`
   - add `#define INCLUDE_pxTaskGetStackStart 1`
   - add `#define INCLUDE_uxTaskPriorityGet 1`
   - add `#include "SEGGER_SYSVIEW_FreeRTOS.h"` at the end of file
8. Enable SystemView on startup (before creating first OS task): call `SEGGER_SYSVIEW_Conf();`
9. Add a call to `SYSVIEW_setMicroJVMTask(pvCreatedTask);` just after creating the OS task which launch the MicroJvm. The handler to give is the one filled by `xTaskCreate` function.

MicroJvm Task

The MicroJvm task is an OS task and has got a particular role: it manages the MicroEJ Java threads. As soon as the MicroEJ application is started (when calling `SNI_startVM`), the jobs performed during the MicroJvm task (events, semaphores etc.) are dispatched to the current MicroEJ Java thread. By consequence, this task is useless when the MicroEJ application is running.

SystemView for MicroEJ disables the visibility of the MicroJvm task when the MicroEJ application is running. It simplifies the SystemView client debugging.

Task and Thread Names

To make a distinction between the OS tasks and the MicroEJ Java thread, a prefix is added to the task/thread name: “[OS] ” or “[MEJ] “.

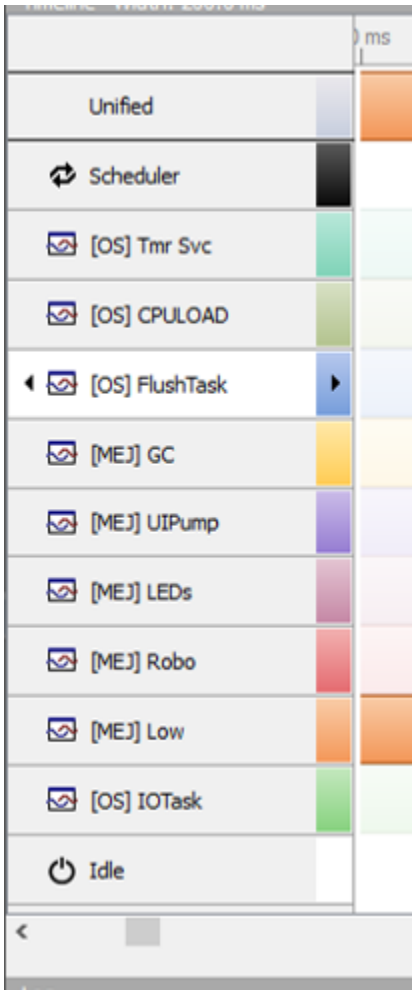


Fig. 44: OS and Thread Names

Note: SystemView limits the number of characters to 32. The prefix length is included in these 32 characters and by consequence the original task/thread name can be cropped.

Task and Thread Priorities

SystemView lists the tasks and threads according their priorities. However the priority notion has not the same signification when talking about tasks or threads: a thread priority depends on the MicroJvm task priority.

By consequence a thread with the priority 5 must not be included between a task with the priority 4 and other task with priority 6 . It depends on MicroJvm task priority:

- if this task priority is 3 , the thread has got a lower priority than task with priority 4 .

- if this task priority is `7`, the thread has got a higher priority than task with priority `6`.

To allow SystemView making this distinction, the priorities sent to SystemView client respect the following rules:

- it is an OS task: `priority_sent = task_priority * 100`.
- it is a MicroEJ Java thread: `priority_sent = MicroJvm_task_priority * 100 + thread_priority`.

4.15 Simulation

4.15.1 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 Java heap dump, and an accurate Java scheduling policy (the same as the embedded one).[1]

4.15.2 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.

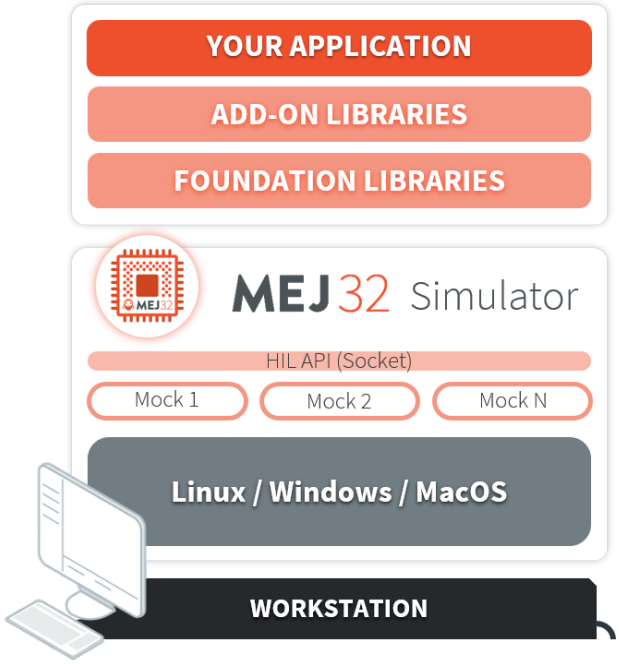


Fig. 45: The HIL Connects the MicroEJ Simulator to the Workstation.

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.

[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.

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.

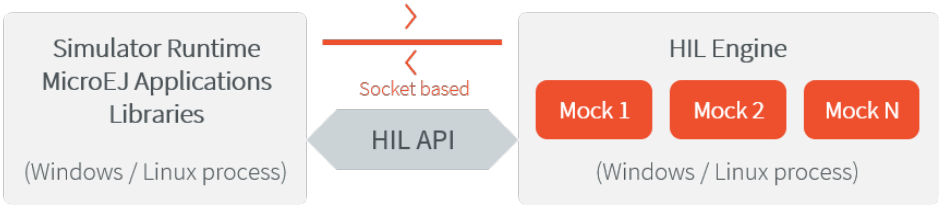


Fig. 46: A MicroEJ Simulator connected to its HIL Engine via a socket.

4.15.3 Dependencies

No dependency.

4.15.4 Installation

The Simulator is a built-in feature of MicroEJ Platform architecture.

4.15.5 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.

4.15.6 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 Simulator while still (apparently) interacting 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]* 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.

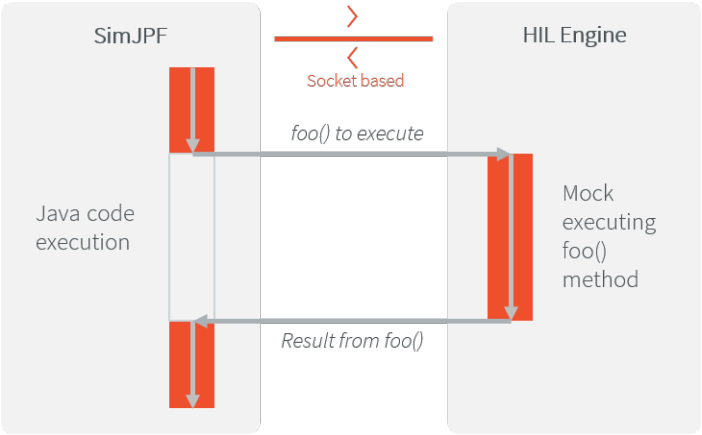


Fig. 47: The MicroEJ Simulator Executes a Native Java Method `foo()`.

Example

```
package example;

import java.io.IOException;

/**
 * Abstract class providing a native method to access sensor value.
 * This method will be executed out of virtual machine.
 */
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
    }
}
```

To implement the native method `getSensorValue(int sensorID)`, you need to create a MicroEJ standard project containing the same `Sensor` class on the same `example` package. To do so, open the Eclipse menu `File > New > Project... > Java > Java Project` in order to create a MicroEJ standard project.

The following code is the required `Sensor` class of the created Mock project:

```
package example;

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
     */
    private static final int SENSOR_ERROR = -1;
    private static final int POTENTIOMETER_ID = 3;

    private static final Random RANDOM = new Random();

    /**
     * Implementation of native method "getSensorValue()"
     *
     * @param sensorID Sensor ID
     * @return Simulated sensor value
     */
    public static int getSensorValue(int sensorID) {
        if( sensorID == POTENTIOMETER_ID ) {
            // For the simulation, Mock returns a random value
            return RANDOM.nextInt();
        }
        return SENSOR_ERROR;
    }

}
```

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. 48: 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.

```
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(){
      //inside the Mock
      //wait until the data is received
      synchronized (lock) {
            while(dataLength == 0) {
                  try {
```

(continues on next page)

(continued from previous page)

```
                        lock.wait(); // equivalent to lock.wait(0)
                    } catch (InterruptedException e) {
                        Thread.currentThread().interrupt();
                        // Use the error code specific to your library
                        throw new NativeException(-1, "InterruptedException", e);
                    }
                }
            }
}

//Mock data reader thread
public static void notifyDataReception() {
        synchronized (lock) {
                dataLength = readFromInputStream(data);
                lock.notifyAll();
        }
}
```

Resource Management

In Java, every 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 Java 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);
        ...
}
```

Synchronous Terminations

To terminate the whole simulation (MicroEJ Simulator and HIL), use the stop() method.

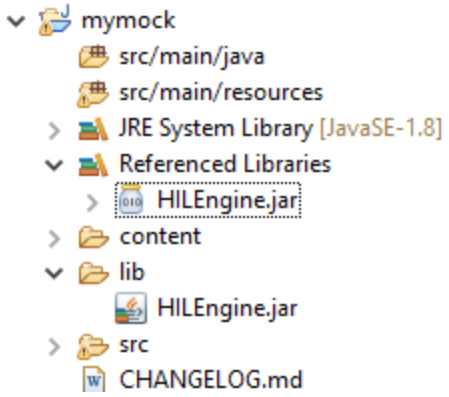
```
public static void windowClosed() {
        HIL.getInstance().stop();
}
```

Dependencies

The HIL Engine API is automatically provided by the `microej-mock` project skeleton.

Installation

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 Platform in one of the two ways:

- by adding the mock module as a regular Platform module dependency (if your Platform configuration project contains a `module.ivy` file),
- or by manually copying the JAR file `[mock_project]\target~\rip\mocks\[mock_name].jar` to the *Platform configuration* mock dropins folder `dropins/mocks/dropins/`.

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.

4.15.7 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 J2SE application.

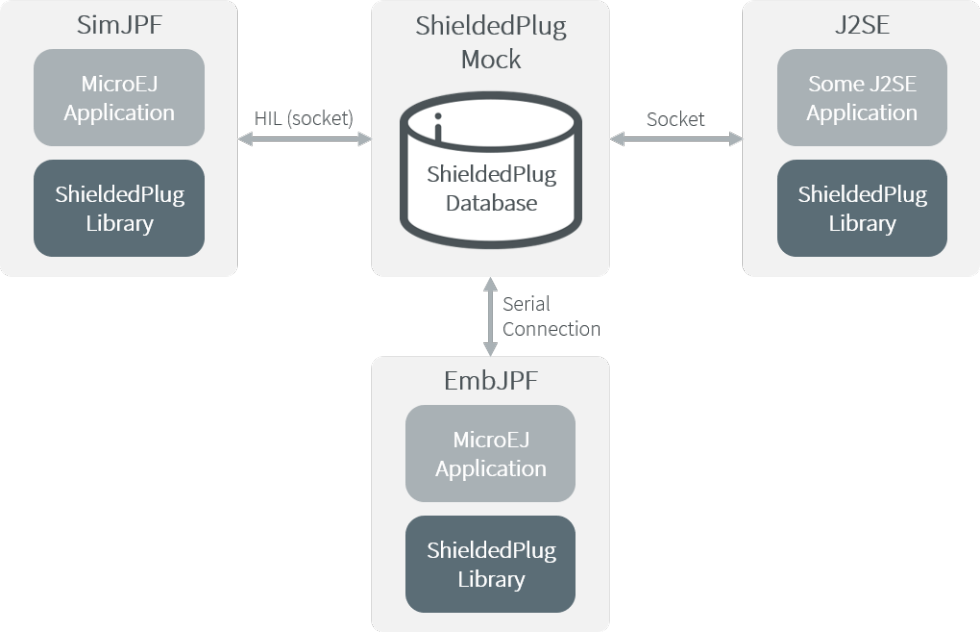


Fig. 49: Shielded Plug Mock General Architecture

Configuration

The mock socket port can be customized for J2SE 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`

4.15.8 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 *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 Platform project.

The Front Panel Project

Creating 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:

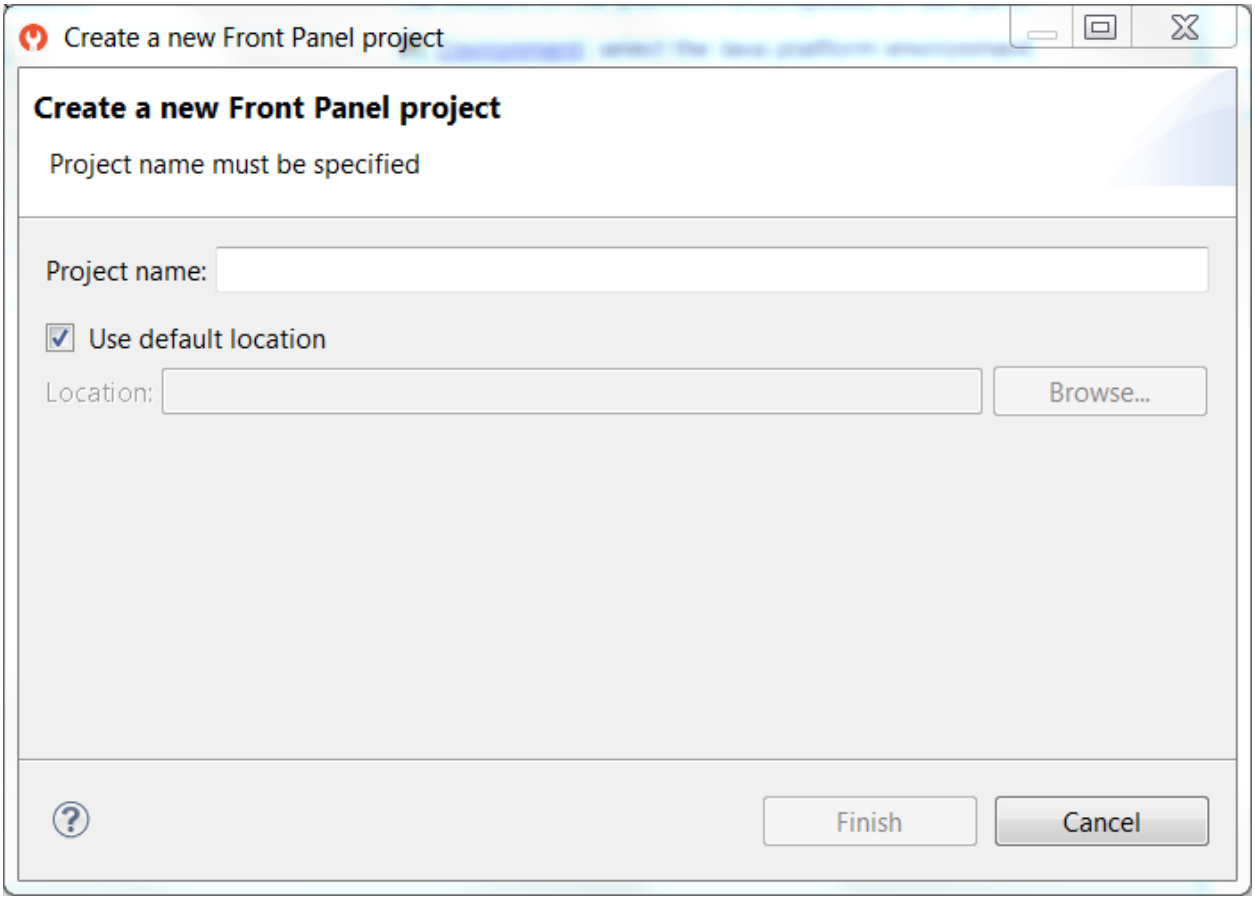


Fig. 50: New Front Panel Project Wizard

Enter the name for the new project.

Project Contents

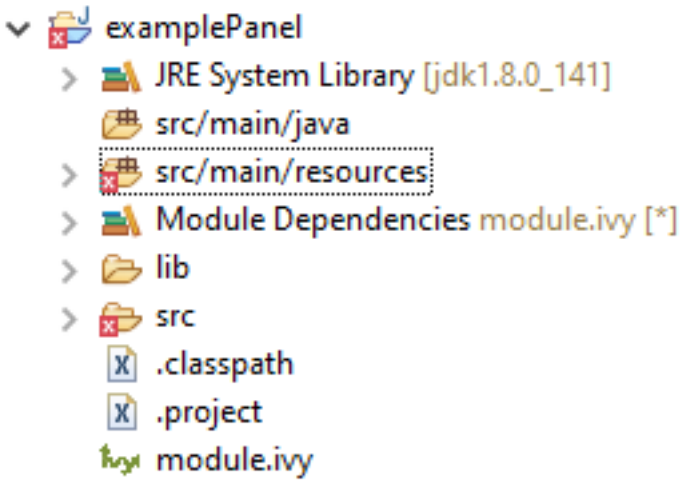


Fig. 51: Project Contents

A Front Panel project has the following structure and contents:

- The `src/main/java` folder is provided for the definition of `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

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="widget" rev="1.0.0"/>
```

(continues on next page)

(continued from previous page)

```
   </dependencies>
</ivy-module>
```

The dependency `ej.tool.frontpanel#widget` is only useful for MicroUI application (see *Simulation*). The dependencies block must be manually updated to depend only on the Front Panel framework. This framework contains the Front Panel core classes:

```
<dependencies>
   <dependency org="ej.tool.frontpanel" name="framework" rev="1.0.0"/>
</dependencies>
```

The Front Panel framework does not provide any widgets. Widgets have to be added to simulate user interactions.

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 `7.11.0`). A quick migration guide is available: open Platform configuration file `.Platform`, go to `Content` tab, click on module `Front Panel`. The migration guide is available in `Details` box.

Editing Front Panel Files

To edit a `.fp` file, 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 (`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.

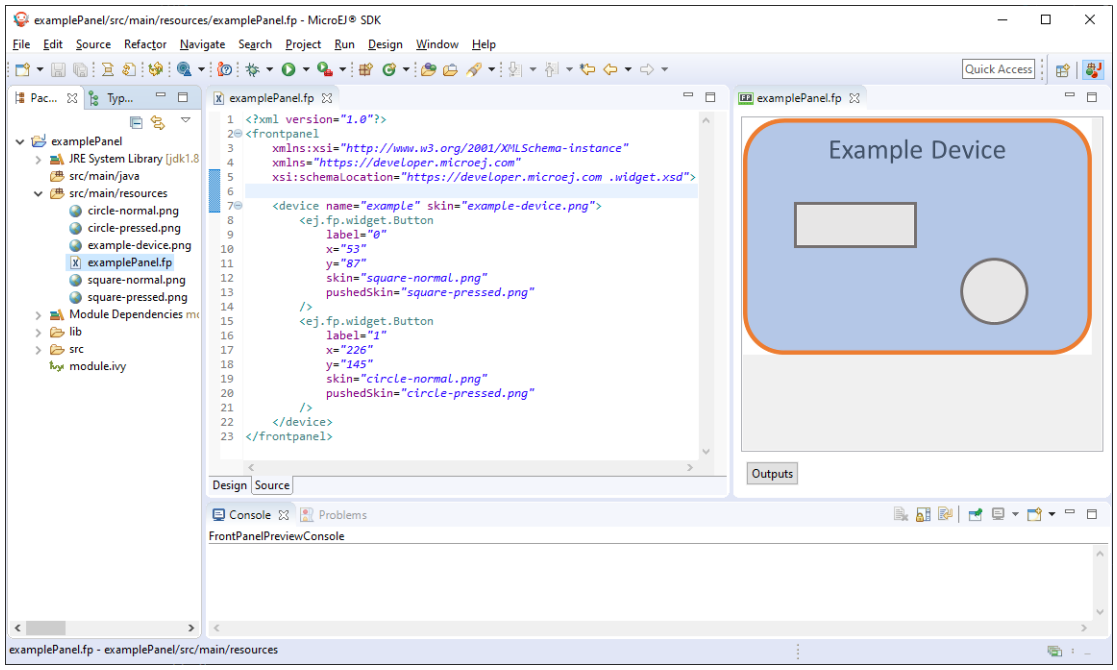


Fig. 52: Working Layout Example

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 `fp` files are compiled when exporting the Front Panel project in a Platform (or during Platform build). It is useful to have two or more representation of a board (size, devices layout, display size, etc.). By default the Simulator will chooses the default `.fp` file declared by the Platform, or will defaults to a random one. To choose a specific one, set the *Application Option* `frontpanel.file` to a Front Panel simple file name included in the Platform (e.g. `mycompany.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.

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

   /**
   * 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.
   */
```

(continues on next page)

(continued from previous page)

```
   @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.

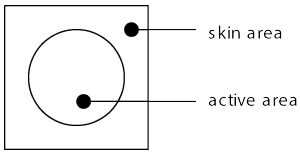


Fig. 53: Active Area

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.

Installation

In the *platform configuration* file, check Front Panel to install the Front Panel module. When checked, the properties file `frontpanel` > `frontpanel.properties` is required during platform 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 platform 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.

To test a Front Panel project without rebuilding the platform or without exporting manually the project, add the *Application Option* `ej.fp.project` to a Front Panel Project absolute path (e.g. `c:\\mycompany\\myfrontpanel.fp`). The Simulator will use the specified Front Panel project prior to the one included by the Platform.

Note: This feature works only if the Platform has been built with the Front Panel module enabled.

Warning: This feature is useful to test locally some changes in Front Panel project. The Platform does not contain the changes until a new Platform is built.

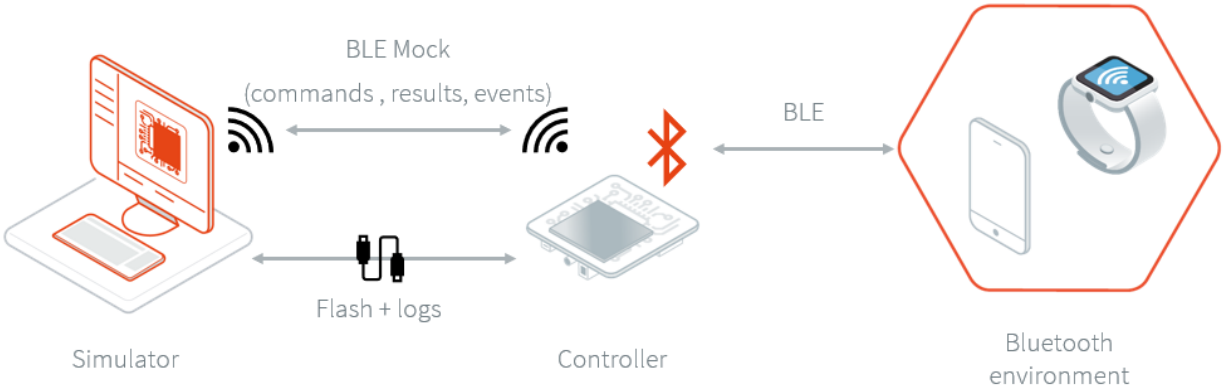
Use

Launch an application on the Simulator to run the Front Panel.

4.15.9 Bluetooth LE Mock

Overview

To run a MicroEJ Application that uses the Bluetooth LE Foundation Library (ej.api.bluetooth) on MicroEJ Simulator, a Bluetooth LE mock controller must be set up first:



The Bluetooth LE mock controller is a hardware mock of the Bluetooth LE library. It means the Simulator uses a real Bluetooth LE device to scan other devices, advertise, discover services, connect, pair, etc… This design enables testing of apps in a real-world environment.

The Bluetooth LE mock controller implementation is provided for the ESP32-DevKitC board reference. Other implementations or sources can be provided on request.

Requirements

- A ESP32-DevKitC board.
- A Bluetooth LE mock controller firmware.
- A tool to flash the firmware like https://www.espressif.com/en/tools-type/flash-download-tools.

Usage

To simulate a Bluetooth LE application, follow these three steps:

- Set up the controller
- Set up the network configuration
- Run the application on the Simulator

If your are facing any issues, check the *Troubleshooting* section.

Controller Setup

To set up the controller, follow these steps:

- Plug-in the ESP32-DevKitC board to your computer,
- Find the associated COM port,
- In the flash tool:
  - select the chip “ESP32 DownloadTool”
  - browse for the firmware file
  - set the offset to 0x000000
  - set the COM port

- set the baudrate to 921 600
- start the flash download

With the flash download tool from Espressif, you should end with something similar to this :

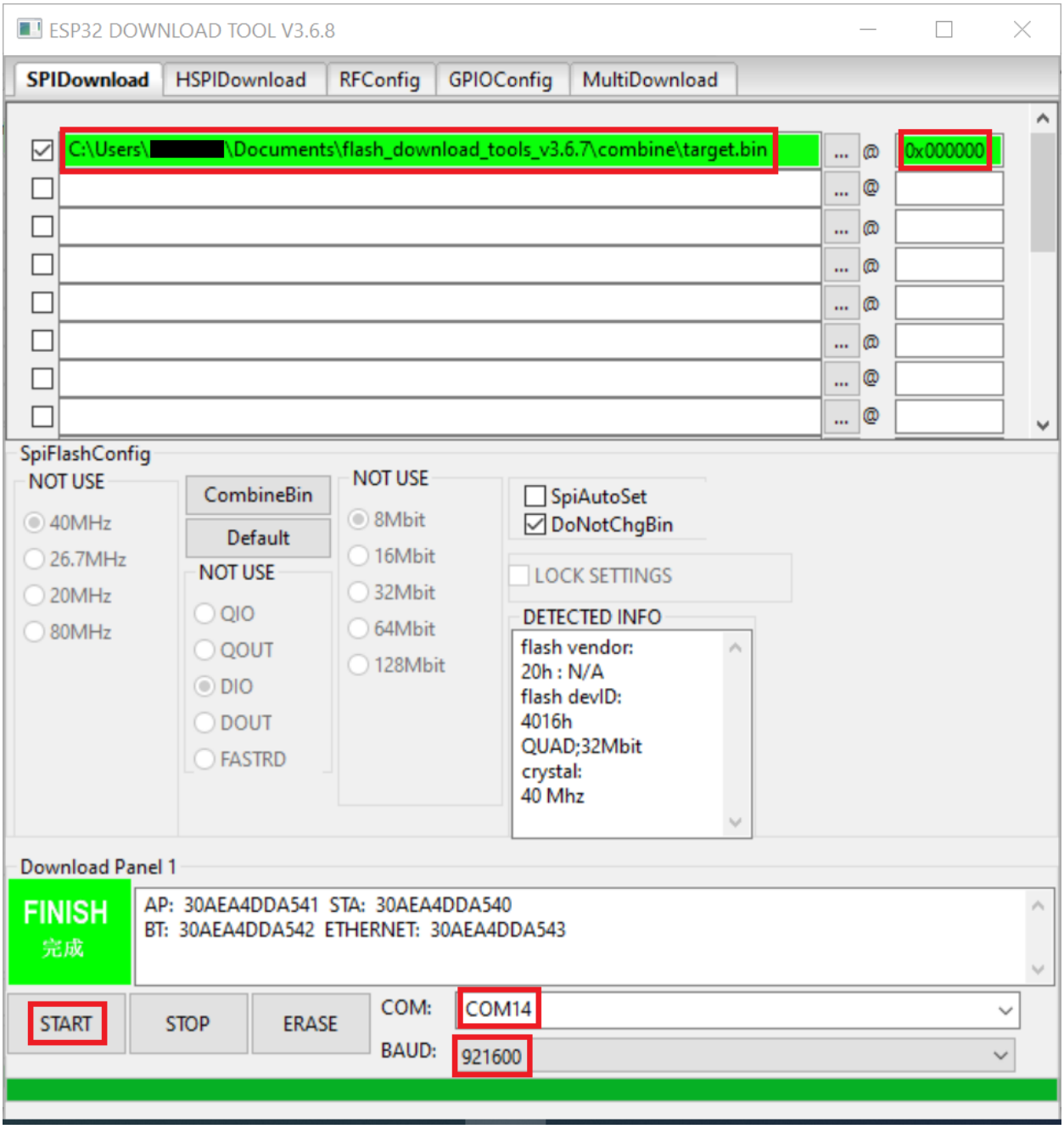
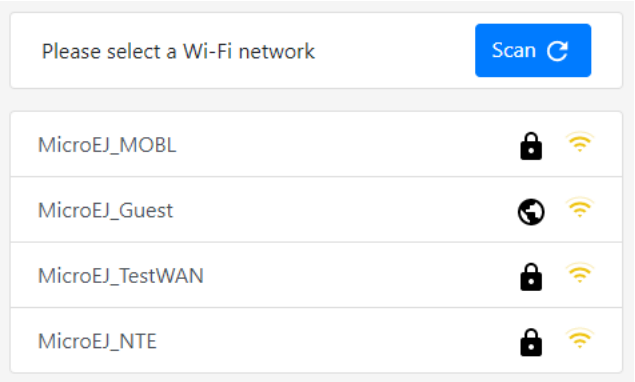


Fig. 54: Bluetooth LE Flash Download Tool Configuration

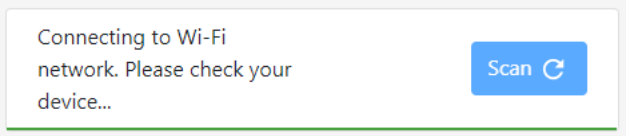
Network Setup

To configure the network:

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 device is successfully connected to the desired network, the web page should looks like this:



   Additionally, the serial output of the device shows connection status.
5. Connect your computer back to this network : your computer and the controller must be in the same network.

Simulation

It is possible to run the Simulator as many times as necessary using the same setup. Also, rebooting the controller will automatically set up the network with the saved configuration.

The IP address of the controller is available in the logs :

```
Termite 3.4 (by CompuPhase)
COM6 115200 bps, 8N1, no handshake
Hoka:F=7 5138 /192.168.4.2
Hoka:I=1
Hoka:F=4 5139 /192.168.4.2
Hoka:F=7 5139 /192.168.4.2
Hoka:I=1
Hoka:F=4 5140 /192.168.4.2
Hoka:I=5 5140 /192.168.4.2 202 Accepted /join
Hoka:F=7 5140 /192.168.4.2
Hoka:I=3
I (117372) wifi: station: 50:eb:71:25:96:71 leave, AID = 1
I (117372) wifi: n:1 0, o:1 1, ap:1 1, sta:255 255, prof:1
[ESP32 Wifi Driver][WARNING] Event 16 received, not treated
I (117402) wifi: flush txq
I (117402) wifi: stop sw txq
I (117402) wifi: lmac stop hw txq
I (117402) wifi: mode : sta (30:ae:a4:dd:91:10)
I (119822) wifi: n:11 0, o:1 0, ap:255 255, sta:11 0, prof:1
I (120372) wifi: state: init -> auth (b0)
I (120382) wifi: state: auth -> assoc (0)
I (120392) wifi: state: assoc -> run (10)
I (120412) wifi: connected with MicroEJ_TestWAN, channel 11
I (120412) wifi: pm start, type: 1

[1B][0;32mI (121772) event: sta ip: 192.168.80.33, mask: 255.255.255.0, gw: 192.168.80.1[1B][0m
ej.bluetooth.bluetoothwificontroller INFO: Succesfully joined the wifi network
ej.bluetooth.bluetoothwificontroller INFO: Saving credentials...
ej.bluetooth.bluetoothwificontroller INFO: Credentials saved
ej.bluetooth.bluetoothwificontroller INFO: Starting server at 192.168.80.33 on port 80
remotecommandserver INFO: Server listening on port 80
```

Before running your Bluetooth LE application on the Simulator, in the *Run configuration* panel, set the simulation mode to “Controller (over net)” and configure the Bluetooth LE mock settings.

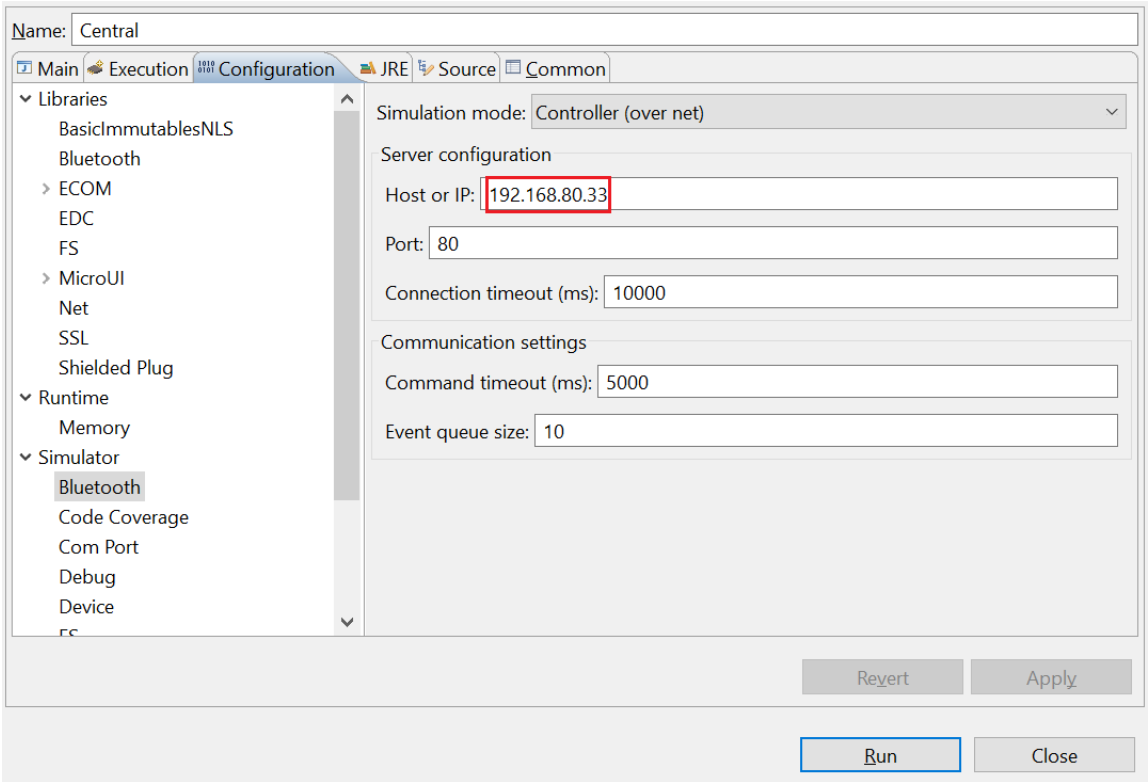


Fig. 55: Bluetooth LE Mock Configuration

Launching the application on the Simulator will restore the controller to its initial state (the BLE adapter is disabled).

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.

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 device is connected to the network (see logs in the serial port output) and that your computer is in the same network.

4.16 Limitations

Table 18: Platform Limitations

Item		EVAL	DEV
Number of Classes		4000	4000
Number of methods per class		3000	65000
Total number of methods		4000	unlimited
Class / Interface hierarchy depth		127 max	127 max
Number of monitors[1] per thread		8 max	8 max
Numbers of exception handlers per method		63 max	63 max
Number of fields	Base type	65000	65000
	References	31	31
Number of statics	boolean + byte	limited	65000
	short + char	limited	65000
	int + float	limited	65000
	long + double	limited	65000
	References	limited	65000
Method size		65000	65000
Time limit		60 minutes	unlimited
Number of threads		62	62

4.17 Appendices

4.17.1 Appendix A: 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 MicroEJ Platform at platform build time.

LLMJVM: MicroEJ Core Engine

Naming Convention

The Low Level MicroEJ Core Engine API, the `LLMJVM` API, relies on functions that need to be implemented. The naming convention for such functions is that their names match the `LLMJVM_IMPL_*` pattern.

Header Files

Three C header files are provided:

[1] No more than n different monitors can be held by one thread at any time.

- LLMJVM_impl.h

  Defines the set of functions that the BSP must implement to launch and schedule the virtual machine

- LLMJVM.h

  Defines the set of functions provided by virtual machine that can be called by the BSP when using the virtual machine

- LLBSP_impl.h

  Defines the set of extra functions that the BSP must implement.

LLKERNEL: Multi-Sandbox

Naming Convention

The Low Level Kernel API, the `LLKERNEL` API, relies on functions that need to be implemented. The naming convention for such functions is that their names match the `LLKERNEL_IMPL_*` pattern.

Header Files

One C header file is provided:

- LLKERNEL_impl.h

  Defines the set of functions that the BSP must implement to manage memory allocation of dynamically installed applications.

LLSP: Shielded Plug

Naming Convention

The Low Level Shielded Plug API, the `LLSP` API, relies on functions that need to be implemented. The naming convention for such functions is that their names match the `LLSP_IMPL_*` pattern.

Header Files

The implementation of the *[SP]* for the MicroEJ Platform 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 MicroEJ Core Engine is not able to find a resource (an image, a file etc.) in the resources area linked with the MicroEJ Core Engine.

When a resource is not available, the MicroEJ 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.

Naming Convention

The Low Level API, the `LLEXT_RES` API, relies on functions that need to be implemented. The naming convention for such functions is that their names match the `LLEXT_RES_IMPL_*` pattern.

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

Naming Convention

The Low Level Comm API (LLCOMM), relies on functions that need to be implemented by engineers in a driver. The names of these functions match the `LLCOM_BUFFERED_CONNECTION_IMPL_*` or the `LLCOM_CUSTOM_CONNECTION_IMPL_*` pattern.

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 platform 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 *Dependencies*).

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 19: LLUI_INPUT API for predefined event generators

Function name	Default event generator kind[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.

Event Buffer

The maximum usage of the internal event buffer may be retrieved at runtime using the LLUI_INPUT_getMaxEventsBufferUsage function. This is useful for tuning the size of the buffer.

LLUI_DISPLAY: Display

Principle & Naming Convention

The Graphics Engine provides some Low Level APIs to connect a display driver. The file LLUI_DISPLAY_impl.h defines the API headers to be implemented. For the APIs themselves, the naming convention is that their names match the *_IMPL_* pattern when the functions need to be implemented:

- LLUI_DISPLAY_IMPL_initialize

[1] The implementation class is a subclass of the MicroUI class of the column.

- `LLUI_DISPLAY_IMPL_binarySemaphoreTake`
- `LLUI_DISPLAY_IMPL_binarySemaphoreGive`
- `LLUI_DISPLAY_IMPL_flush`

Some additional Low Level APIs allow you to connect display extra features. These Low Level APIs are not required. When they are not implemented, a default implementation is used (weak function). It concerns backlight, contrast, etc.

This describes succinctly some `LLUI_DISPLAY_IMPL` functions. Please refer to documentation inside header files to have more information.

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 too *give* two binary semaphores.

Image Heap

The display driver must reserve a runtime memory buffer for creating dynamic images when using MicroUI `ResouceImage` 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` .

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 set in 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 the rectangular area of the content which has changed during the last drawing action, and which must be flushed to the display 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 copy.

As soon as the application performs a new drawing, the Graphics Engine locks the thread. It will automatically unlocked when the BSP will call `LLUI_DISPLAY_flushDone` at the end of the copy,

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.

Naming Convention

The Low Level APIs rely on functions that must be implemented. The naming convention for such functions is that their names match the `*_IMPL_*` pattern.

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.

LLNET: Network

Naming Convention

The Low Level API, the `LLNET` API, relies on functions that need to be implemented. The naming convention for such functions is that their names match the `LLNET_IMPL_*` pattern.

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

Naming Convention

The Low Level API, the `LLNET_SSL` API, relies on functions that need to be implemented. The naming convention for such functions is that their names match the `LLNET_SSL_*` pattern.

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).

LLFS: File System

Naming Convention

The Low Level File System API (LLFS), relies on functions that need to be implemented by engineers in a driver. The names of these functions match the `LLFS_IMPL_*` and the `LLFS_File_IMPL_*` pattern.

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.).

LLHAL: Hardware Abstraction Layer

Naming Convention

The Low Level API, the `LLHAL` API, relies on functions that need to be implemented. The naming convention for such functions is that their names match the `LLHAL_IMPL_*` pattern.

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

Naming Convention

The Low Level Device API (LLDEVICE), relies on functions that need to be implemented by engineers in a driver. The names of these functions match the `LLDEVICE_IMPL_*` pattern.

Header Files

One C header file is provided:

- LLDEVICE_impl.h

  Defines a set of functions that the BSP must implement to get the platform architecture name and unique device identifier.

4.17.2 Appendix B: 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 20: 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 next table:

Table 21: 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. Increase the maximum size of the thread stack parameter.
-2	No Java stack block could be allocated with the given size. Increase the Java stack block size.
-1	The Java stack space is full. Increase the Java stack size or the number of Java stack blocks.
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`.

Exit Codes

The RTOS task that runs the MicroEJ runtime may end, especially when the MicroEJ Application calls `System.exit` method. By convention, a negative value indicates abnormal termination.

Table 22: MicroEJ Platform exit codes

Message ID	Meaning
0	The MicroEJ Application ended normally.
-1	The SOAR and the MicroEJ Platform are not compatible.
-2	Incompatible link configuration (`lsc` file) with either the SOAR or the MicroEJ Platform.
-3	Evaluation version limitations reached: termination of the application.
-5	Not enough resources to start the very first MicroEJ thread that executes `main` method.
-12	Maximum number of threads reached.
-13	Fail to start the MicroEJ Platform because the specified MicroEJ heap is too large.
-14	Invalid stack space due to a link placement error.
-15	The application has too many static (the requested static head is too large).
-16	The MicroEJ Core Engine cannot be restarted.

SNI

Error Messages

The following error messages are issued at runtime.

Table 23: [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 24: 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 Definition

Kernel types, methods and static fields allowed to be accessed by Features must be declared in `kernel.api` file. Kernel API file is an XML file (see *example "Kernel API XML Schema"* and *table "XML elements specification"*).

Listing 10: 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>
        </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 25: 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]`

Access Error Codes

When an instruction is executed that will break a *[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 26: Error codes: source

Ch aracter	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 27: 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

Loading Features Dynamically

Features may be statically embedded with the Kernel or dynamically built against a Kernel. To build a Feature binary file, select `Build Dynamic Feature` MicroEJ Platform `Execution` tab. The generated file can be dynamically loaded by the Kernel runtime using `ej.kf.Kernel.load(InputStream)` .

ECOM

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 28: 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 29: 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.

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 30: 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.

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 31: 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.
-25	Network not initialized.
-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 32: 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.

Continued on next page

Table 32 – continued from previous page

Message ID	Description
-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.
-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.

Continued on next page

Table 32 – continued from previous page

Message ID	Description
-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.
-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.

Continued on next page

Table 32 – continued from previous page

Message ID	Description
-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.
-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.

4.17.3 Appendix C: Tools Options and Error Codes

SOAR

When a generic exception is thrown by the SOAR, the error message

`SOAR ERROR [M<messageId>] <message>`

is issued, where `<messageId>` and `<message>` meanings are defined in the next table.

Table 33: SOAR Error Messages.

Message ID	Description
0	The SOAR process has encountered some internal limits.
1	Unknown option.
2	An option has an invalid value.
3	A mandatory option is not set.
4	A filename given in options does not exist .
5	Failed to write the output file (access permissions required for `-toDir` and `-root` options).
6	The given file does not exist.
7	I/O error while reading a file.
8	An option value refers to a directory, instead of a file.
9	An option value refers to a file, instead of a directory or a jar file.
10	Invalid entry point class or no `main()` method.
11	An information file can not be generated in its entirety.
12	Limitations of the evaluation version have been reached.
13	I/O rrror while reading a jar file.
14	IO Error while writing a file.
15	I/O error while reading a jar file: unknown entry size.
16	Not enough memory to load a jar file.
17	The specified SOAR options are exclusive.
18	XML syntax error for some given files.
19	Unsupported float representation.
23	A clinit cycle has been detected. The clinit cycle can be cut either by simplifying the application clinit code or by explicitly declaring clinit dependencies. Check the generated `.clinitmap` file for more information.
50	Missing code: Java code refers to a method not found in specified classes.
51	Missing code: Java code refers to a class not found in the specified classpath.
52	Wrong class: Java code refers to a field not found in the specified class.
53	Wrong class: A Java classfile refers to a class as an interface.
54	Wrong class: An abstract method is found in a non-abstract class.
55	Wrong class: illegal access to a method, a field or a type.
56	Wrong class: hierarchy inconsistency; an interface cannot be a superclass of a class.
57	Circularity detected in initializion sequence.
58	Option refers twice to the same resource. The first reference is used.
59	Stack inconsistency detected.
60	Constant pool inconsistency detected.
61	Corrupted classfile.
62	Missing native implementation of a native method.
63	Cannot read the specified resource file.
64	The same property name cannot be defined in two different property files.
65	Bad license validity.
66	Classfiles do not contain debug line table information.

Continued on next page

Table 33 – continued from previous page

Message ID	Description
67	Same as 51.
150	SOAR limit reached: The specified method uses too many arguments.
151	SOAR limit reached: The specified method uses too many locals.
152	SOAR limit reached: The specified method code is too large.
153	SOAR limit reached: The specified method catches too many exceptions.
154	SOAR limit reached: The specified method defines a stack that is too large.
155	SOAR limit reached: The specified type defines too many methods.
156	SOAR limit reached: Your application defines too many interfaces.
157	SOAR limit reached: The specified type defines too many fields.
158	SOAR limit reached: your application defines too many types.
159	SOAR limit reached: Your application defines too many static fields.
160	SOAR limit reached: The hierarchy depth of the specified type is too high.
161	SOAR limit reached: Your application defines too many bundles.
162	SOAR limit reached: Your application defines too deep interface hierarchies.
163	SOAR limit reached: Your application defines too many cnocrete types.
164	SOAR limit reached: Your application defines too many reference fields in a class.
251	Error in converting an IEE754 float(32) or double(64) to a fixed-point arithmetic number
300	Corrupted class: invalid dup_x1 instruction usage.
301	Corrupted class: invalid dup_x2 instruction usage.
302	Corrupted class:invalid dup_x2 instruction usage.
303	Corrupted class:invalid dup2_x1 instruction usage.
304	Corrupted class:invalid dup2_x1 instruction usage.
305	Corrupted class:invalid dup2_x2 instruction usage.
306	Corrupted class: invalid dup2 instruction usage.
307	Corrupted class:invalid pop2 instruction usage.
308	Corrupted class:invalid swap instruction usage.
309	Corrupted class: Finally blocks must be inlined.
350	SNI incompatibility: Some specified type should be an array.
351	SNI incompatibility: Some type should define some specified field.
352	SNI incompatibility: The specified field is not compatible with SNI.
353	SNI incompatibility: The specified type must be a class.
354	SNI incompatibility: The specified static field must be defined in the specified type.
355	SNI file error: The data must be an integer.
356	SNI file error : unexpected tag
357	SNI file error : attributes <name>, <descriptor>, <index> and <size> are expected in the specified tag.
358	SNI file error : invalid SNI tag value.
359	Error parsing the SNI file.
360	XML Error on parsing the SNI file.
361	SNI incompatibility : illegal call to the specified data.
362	No stack found for the specified native group.
363	Invalid SNI method: The argument cannot be an object reference.
364	Invalid SNI method: The array argument must only be a base type array.
365	Invalid SNI method: The return type must be a base type.
366	Invalid SNI method: The method must be static.

Immutable Files Related Error Messages

The following error messages are issued at SOAR time (link phase) and not at runtime.

Table 34: 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 35: [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 36: 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 37: 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 38: 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 11: 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 39: 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 *Dependencies*.
	`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.
	`listener`	*Optional.* Default listener's logical name. Only a display is a valid listener. If no listener is specified the listener is the default display.
`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.
	`listener`	*Optional.* Default listener's logical name. Only a display is a valid listener. If no listener is specified, then the listener is the default display.
`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).
	`listener`	*Optional.* Default listener's logical name. Only a display is a valid listener. If no listener is specified, then the listener is the default display.
`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).
	`listener`	*Optional.* Default listener's logical name. Only a display is a valid listener. If no listener is specified, then the listener is the default display.
`touch`		The default event generator `Touch`.
	`name`	The logical name.
	`display`	Logical name of the Display with which the touch is associated.
	`listener`	*Optional.* Default listener's logical name. Only a display is a valid listener. If no listener is specified, then the listener is the default display.
`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.

Continued on next page

Table 39 – continued from previous page

Tag	Attributes	Description
	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]
	listener	*Optional.* Default listener's logical name. Only a display is a valid listener. If no listener is specified, then the listener is the default display.
array		An array of values.
	name	The logical name.
elem		A value.
	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>
```

[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.

Table 40: 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.

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 41: 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.

HIL Engine

Below are the HIL Engine options:

Table 42: 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.

[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"

Table 43: 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>
```

```
<!-- 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>
```

Continued on next page

Table 43 – continued from previous page

```
<!-- 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>
```

```
   <!-- 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 12: 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 44: 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.

4.17.4 Appendix D: 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 MicroEJ Core Engine, use the hard ABI.

Supported MicroEJ Core Engine Capabilities by Architecture Matrix

The following table lists the supported MicroEJ Core Engine capabilities by MicroEJ Architectures.

Table 45: Supported MicroEJ Core Engine Capabilities by MicroEJ Architecture Matrix

MicroEJ Core Engine Architectures		Capabilities		
MCU	Compiler	Single application	Tiny application	Multi applications
ARM Cortex-M0	GCC	YES	YES	NO
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
ESP32	ESP-IDF	YES	NO	YES

ARM Cortex-M0

Table 46: ARM Cortex-M0 Compilers

Compiler	Version	Flags and Options	Module
GCC	4.8	`-mabi=aapcs -mcpu=cortex-m0 -mlittle-endian -mthumb`	flopi0G22

ARM Cortex-M4

Table 47: ARM Cortex-M4 Compilers

Compiler	Version	Flags and Options	Module
Keil uVision	5.18.0.0	`--cpu Cortex-M4.fp --apcs=/hardfp --fpmode=ieee_no_fenv`	flopi4A20
GCC	4.8	`-mabi=aapcs -mcpu=cortex-m4 -mlittle-endian -mfpu=fpv4-sp-d16 -mfloat-abi=hard -mthumb`	flopi4G25
IAR Embedded Workbench for ARM	8.32.1.18631	`--cpu Cortex-M4F --fpu VFPv4_sp`	flopi4I35

Note: Since MicroEJ 4.0, Cortex-M4 architectures are compiled using `hardfp` convention call.

ARM Cortex-M7

Table 48: ARM Cortex-M7 Compilers

Compiler	Version	Flags and Options	Module
Keil uVision	5.18.0.0	`--cpu Cortex-M7.fp.sp --apcs=/hardfp --fpmode=ieee_no_fenv`	flopi7A21
GCC	4.8	`-mabi=aapcs -mcpu=cortex-m7 -mlittle-endian -mfpu=fpv5-sp-d16 -mfloat-abi=hard -mthumbb`	flopi7G26
IAR Embedded Workbench for ARM	8.32.1.18631	`--cpu Cortex-M7 --fpu VFPv5_sp`	flopi7I36

ESP32

Table 49: 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	`7.13.1` 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 options (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.

CHAPTER

FIVE

KERNEL DEVELOPER GUIDE

5.1 Overview

5.1.1 Introduction

The Kernel Developer's Guide describes how to create a MicroEJ Multi-Sandbox Firmware, i.e. a firmware that can be extended (statically or dynamically) to 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 firmware.

Here is a non-exhaustive list of the activities to be done by Multi-Sandbox Firmware Developers:

- Defining a list of APIs that will be exposed to applications
- Managing lifecycles of applications (deciding when to install, start, stop and uninstall them)
- Integrating applications (called resident applications)
- Defining and applying permissions on system resources (rules & policies)
- Managing connectivity
- Controlling and monitoring resources

This document takes as prerequisite that a MicroEJ Platform is available for the target device (see *Platform Developer Guide*). This document also assumes that the reader is familiar with the development and deployment of MicroEJ Applications (see *Application Developer Guide*) and specifics of developing Sandboxed Applications (see *Sandboxed Application*).

5.1.2 Terms and Definitions

A *Resident Application* is a Sandboxed Application that is linked into a Multi-Sandbox Firmware.

A *Multi-Sandbox Platform* is a Platform with the Multi Sandbox capability of the MicroEJ Core Engine enabled (see the chapter *Multi-Sandbox* of the *Platform Developer Guide*). A Multi-Sandbox Firmware can only be built with a Multi-Sandbox Platform.

A *Mono-Sandbox Firmware* is produced by building and linking a Standalone Application with a Platform.

A *Virtual Device* is the Multi-Sandbox Firmware counterpart for developing a Sandboxed Application in MicroEJ Studio. It provides the firmware functional simulation part. Usually it also provides a mean to directly deploy a Sandboxed Application on the target device running a Multi-Sandbox Firmware (this is called *Local Deployment*). In case of dynamic application deployment, the Virtual Device must be published on MicroEJ Forge instance in order to execute an internal batch applications build for this device.

5.1.3 Overall Architecture

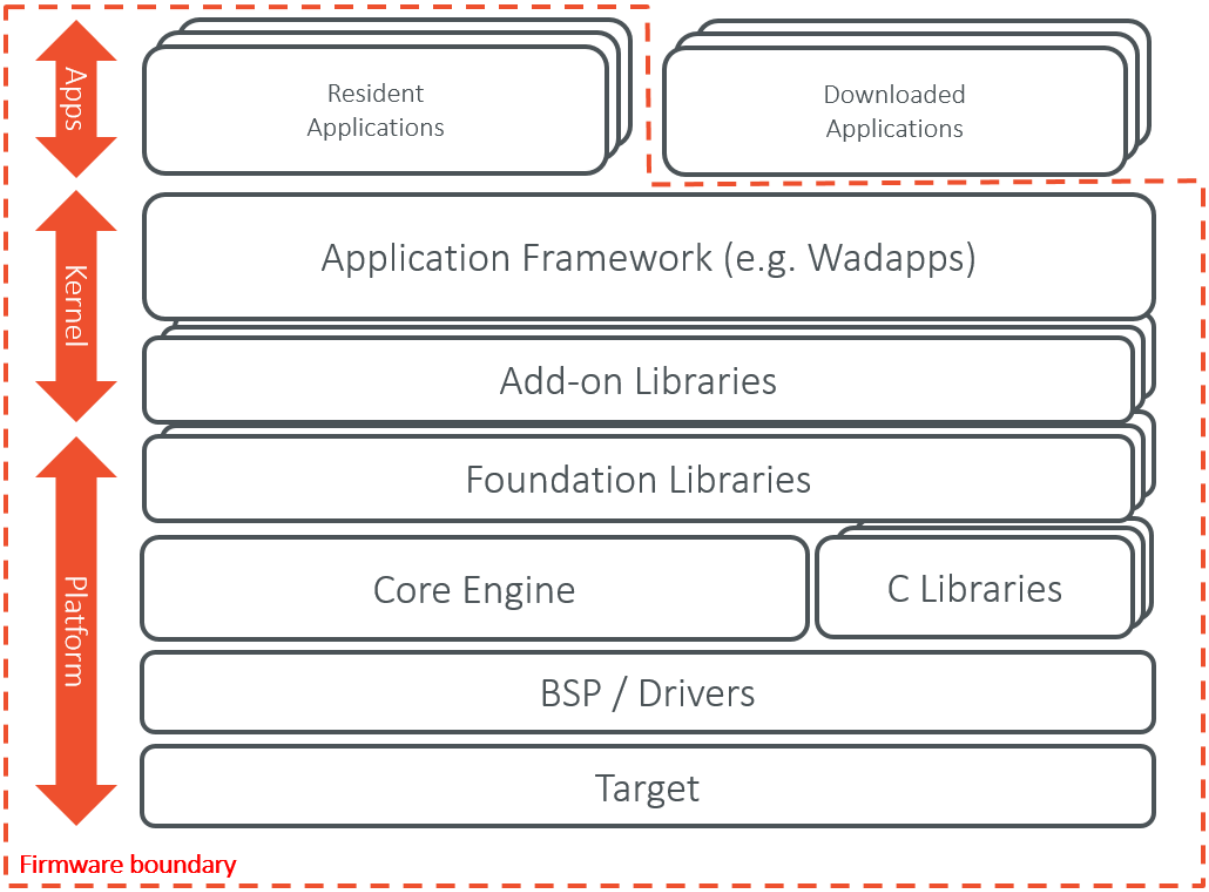


Fig. 1: Firmware Boundary Overview

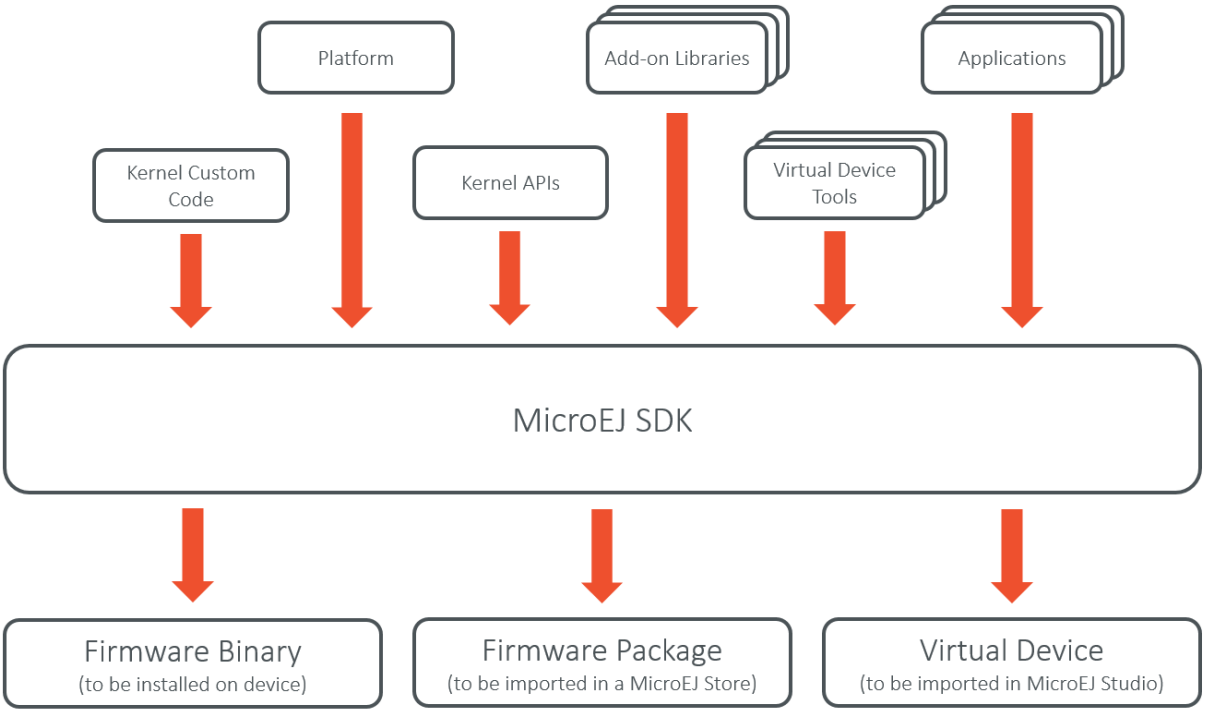


Fig. 2: Firmware Input and Output Artifacts

Firmware Implementation Libraries

Firmware implementations must cover the following topics:

- The firmware's 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 firmware needs.
- The simulation support: how the Virtual Device implementation reflects the firmware implementation, with the help of specific artifacts.
- The Kernel API definition: not all the classes and methods used to implement the firmware's kernel are actually exposed to the applications. There are some artifacts available that expose some of the libraries to the applications, these ones can be picked when the firmware 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 MicroEJ Studio or SDK, extending them with feature that are specific to the firmware, like deployment of an application, a management console, …
- System Applications: pre-built applications that can be embedded as resident apps into a firmware. Some of them are user-land counter parts of the Kernel, implementing the application lifecycle for the firmware's ap-

plication framework (e.g. the Wadapps Framework). These “Kernel System Applications” rely on a dedicated set of interfaces to interact with the Kernel, this interface being defined in a dedicated module.

5.1.4 Firmware Build Flow

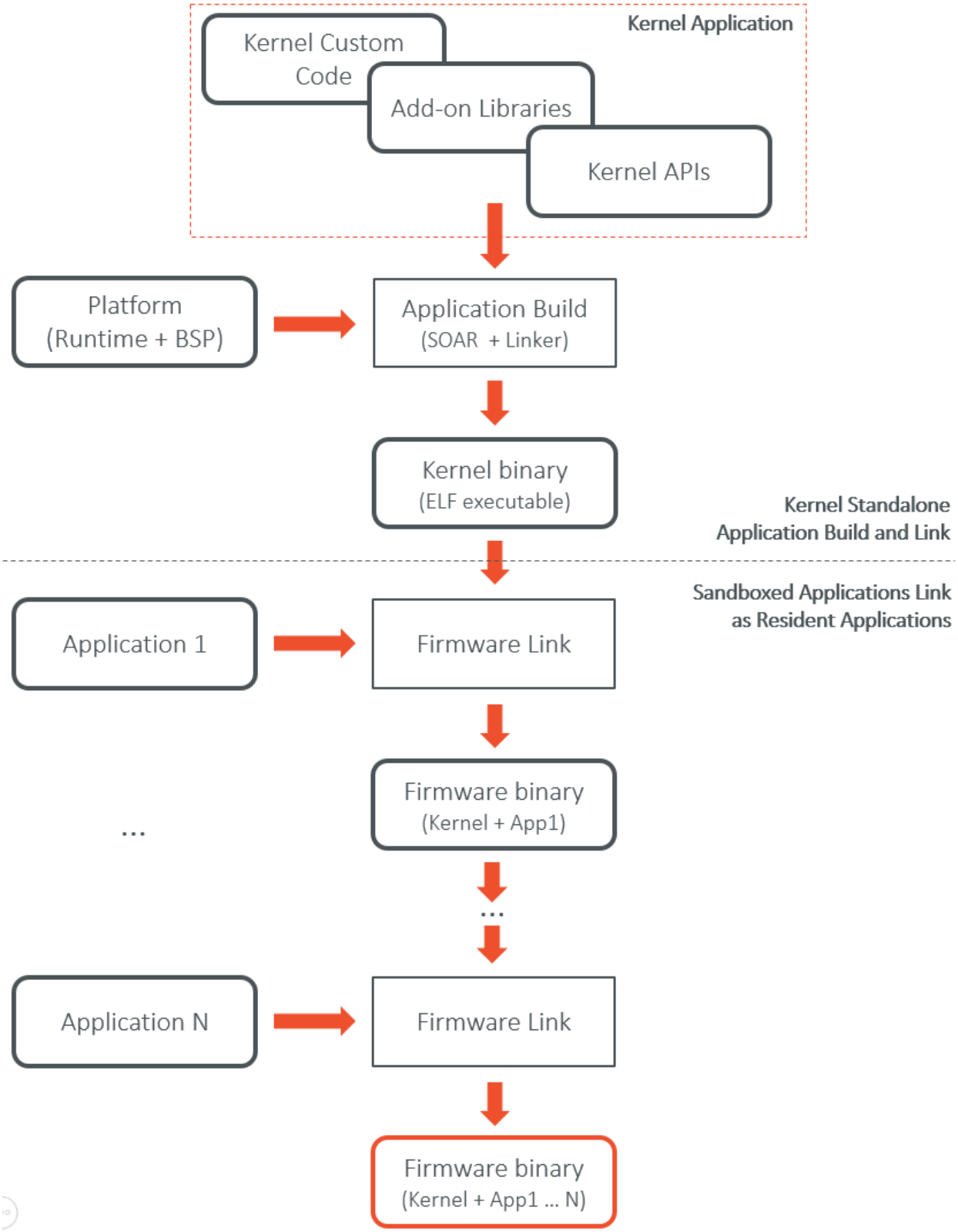


Fig. 3: Firmware Build Flow (Kernel + Resident Applications)

5.1.5 Virtual Device Build Flow

The Virtual Device is automatically built at the same time than the firmware when using the `build-firmware-multiapp` build type (see *Headless Build*). The Virtual Device builder performs the following steps:

- Remove the embedded part of the platform (compiler, linker and runtime).
- Append Add-On Libraries and Resident Applications into the runtime classpath. (See *Ivy Configurations*) for specifying the dependencies).
- Turn the Platform (MicroEJ SDK) license to Virtual Device (MicroEJ Studio) license so that it can be freely distributed.
- Generate the Runtime Environment from the Kernel APIs.

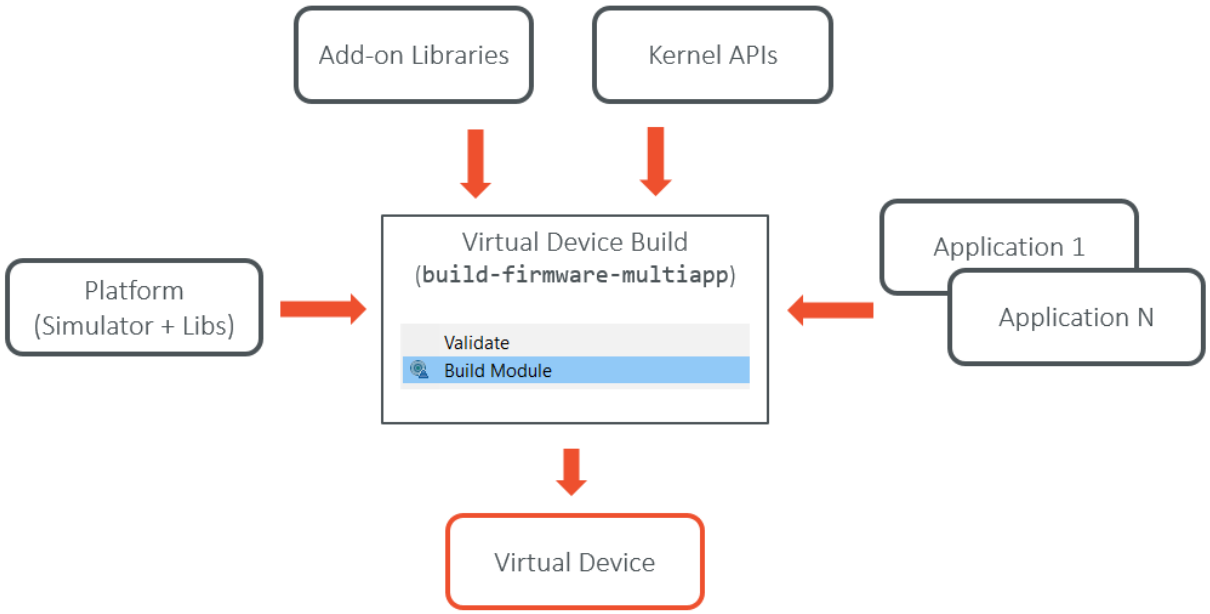


Fig. 4: Virtual Device Build Flow

5.2 Kernel & Features Specification

Kernel & Features semantic (KF) allows an application code to be split between multiples parts: the main application, called the *Kernel* and zero or more sandboxed applications called *Features*.

The Kernel part is mandatory and 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 considered as the Kernel and called a Standalone Application. Even if there are more applications in the platform, there is still only one entry point. This entry point is the Kernel. Applications (downloaded or preinstalled) are "code extensions" (called "Features"), that are called by the Kernel. These Features are fully controlled by the Kernel: they can be installed, started, stopped and uninstalled at any time independently of the system state (particularily, a Feature never depends on an other Feature to be stopped).

The complete *[KF] specification* is available at http://www.e-s-r.net/download/specification/ESR-SPE-0020-KF-1.4-F.pdf

5.3 Getting Started

5.3.1 Online Getting Started

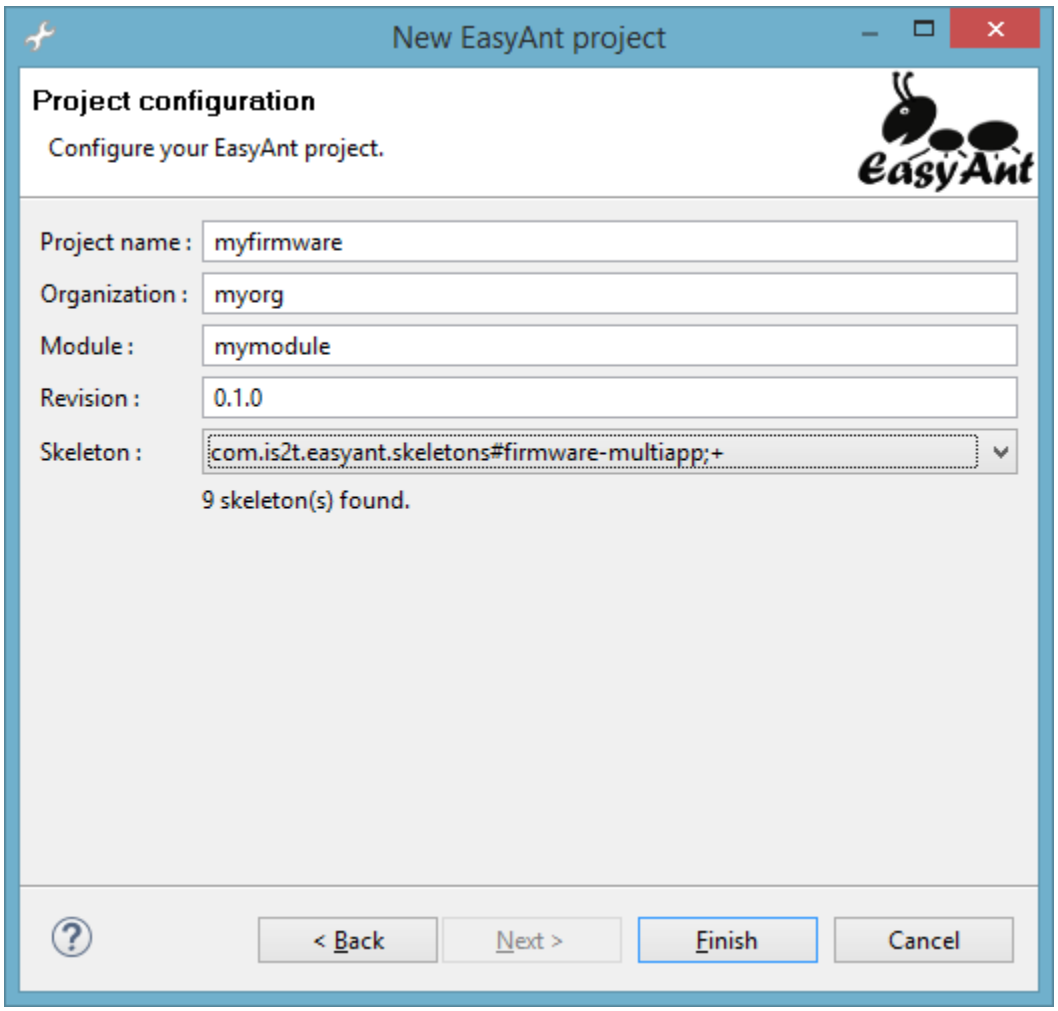
The MicroEJ Multi-Sandbox Firmware Getting Started is available on MicroEJ GitHub repository, at https://github.com/MicroEJ/Example-MinimalMultiAppFirmware.

The file `README.md` provides a step by step guide to produce a minimal firmware on an evaluation board on which new applications can be dynamically deployed through a serial or a TCP/IP connection.

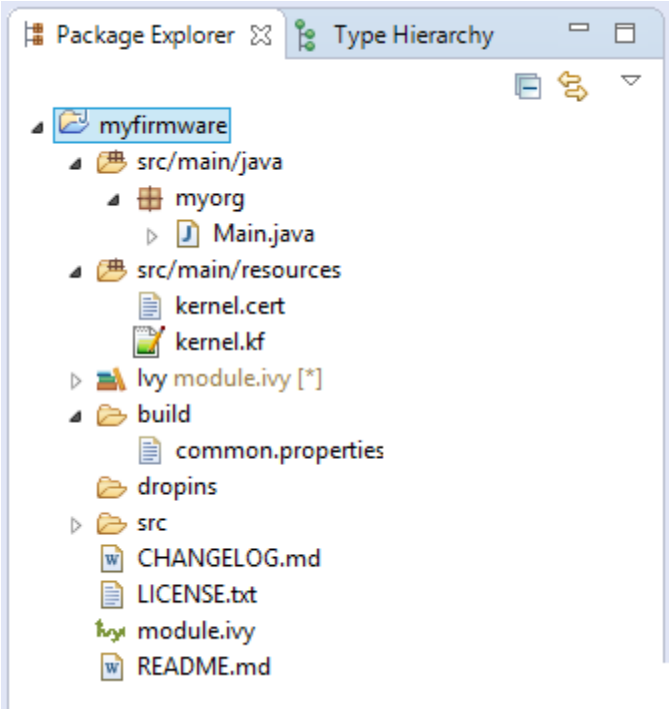
5.3.2 Create an Empty Firmware from Scratch

Create a new Firmware Project

First create a new *module project* using the `build-firmware-multiapp` skeleton.



A new project is generated into the workspace:



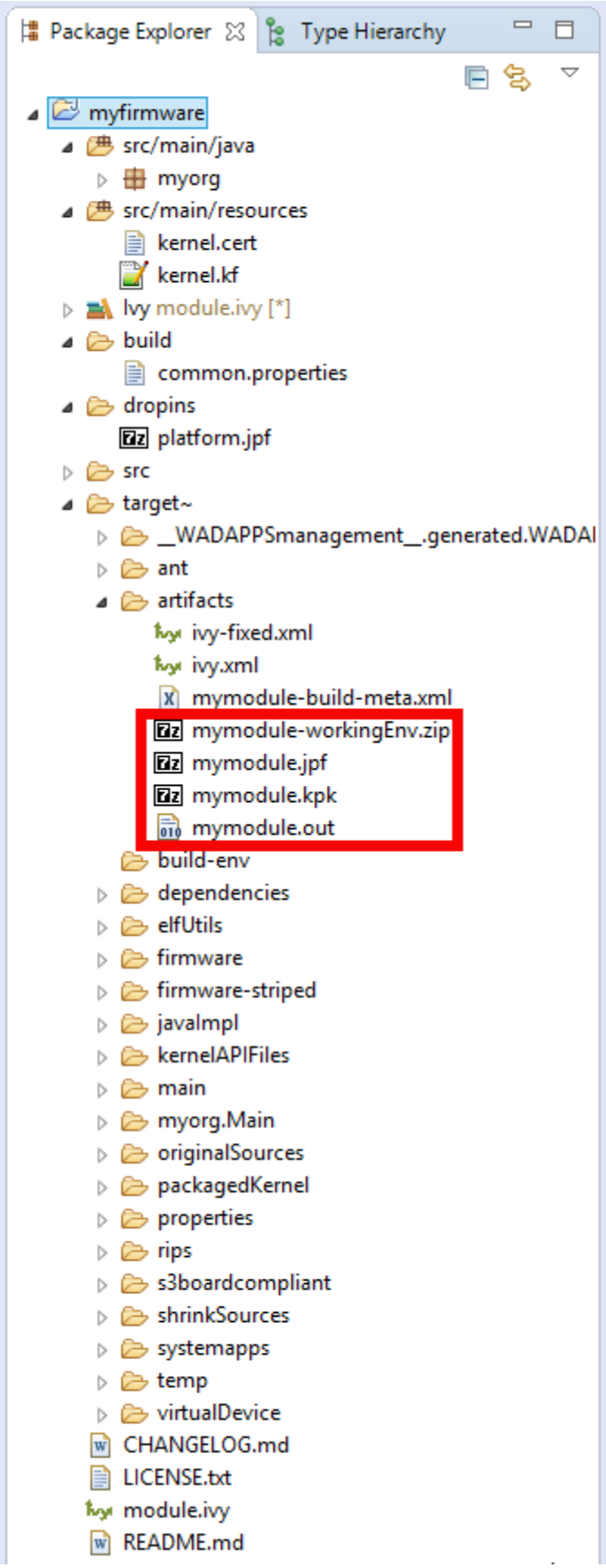
Setup a Platform

Before building the firmware, a target platform must be configured. The easiest way to do it is to copy a platform file into the `myfirmware > dropins` folder. Such file usually ends with `.jpf`. For other ways to setup the input platform to build a firmware see *Change the Platform used to Build the Firmware and the Virtual Device*.

Build the Firmware

In the Package Explorer, right-click on the firmware project and select Build Module. The build of the Firmware and Virtual Device may take several minutes. When the build is succeed, the folder `myfirmware > target~ > artifacts` contains the firmware output artifacts (see *Firmware Input and Output Artifacts*) :

- `mymodule.out` : The Firmware Binary to be programmed on device.
- `mymodule.kpk` : The Firmware Package to be imported in a MicroEJ Forge instance.
- `mymodule.vde` : The Virtual Device to be imported in MicroEJ Studio.
- `mymodule-workingEnv.zip` : This file contains all files produced by the build phasis (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

5.3.3 MicroEJ Demo VEE Flavors

This set of APIs is proposed as examples of industrial or commercial typical products APIs.

What is a MicroEJ Demo Runtime Environment?

A MicroEJ Runtime Environment defines a set of MicroEJ APIs exposed to a MicroEJ Sandboxed Application. Here are the default runtimes provided for evaluation by MicroEJ. Any runtime can be customized with MicroEJ SDK for a specific product.

MicroEJ Demo Runtime Environment	EDC/B-ON/KF	COMP/WADAPPS	MICROUI/MWT	LEDS/BUTTONS	NET/CONNECTIVITY	SSL	HAL	ECOM/COMM
MicroEJ-Developer	✓	✓	✓	✓	✓		✓	✓
MicroEJ-UI	✓	✓	✓	✓			✓	✓
MicroEJ-Headless	✓	✓		✓	✓		✓	✓
MicroEJ-BLE	✓	✓	✓	✓	✓	✓	✓	✓

You can find below what are the different APIs included in the Runtime Environment:

API	Purpose
EDC	Core APIs for the execution.
B-ON	Memory Usage control and Sequences start-up.
KF	Required by the implementation of Shared Interfaces, an inter-application communication process.
COMP or COMPONENTS	Lightweight Services Framework.
WADAPPS	Wadapps Application Framework.
MICROUI/MWT	Main UI library for MicroEJ and the Widgets framework based on MicroUI.
LEDS or MICROUI-LEDS	UI library specific to LEDs.
BUTTONS or MICROUI-BUTTONS	UI library specfic to buttons.
NET	Socket (TCP/UDP) library.
CONNECT or CONNECTIVITY	Network connectivity detection library.
SSL	Secure Socket Layer.
BLE	Bluetooth Low Energy support.
HAL	GPIO Access (digital and analog)
ECOM	Device access framework.
COMM or ECOM-COMM	Serial ports support for the ECOM.

What is a MicroEJ Demo Flavor?

A MicroEJ Demo Flavor is a composition of a set of runtime services, resident applications and a given MicroEJ Runtime Environment. Any flavor can be customized with MicroEJ SDK for a specific product.

MicroEJ Demo Flavor	MicroEJ Runtime Environment	Services Resident Apps				UI Resident Apps		
		Management	App-Metadata-Storage	CommandServer-Socket	NTP	About	AppList	Forge Connect
Green	MicroEJ-Developer	✓	✓	✓	✓			
Blue	MicroEJ-Developer	✓	✓	✓	✓	✓	✓	✓
Red	MicroEJ-UI	✓	✓			✓	✓	
Purple	MicroEJ-Headless	✓	✓	✓	✓			
Black	MicroEJ-BLE	✓	✓	✓	✓			✓

You can find below what are the different System Apps included in the Flavor:

System Apps	Purpose
Management	Contains the implementation of Wadapps framework services, required by all VEE.
App-Metadata-Storage	Stores some Applications Metadata (icons, descriptions) so that it can be locally used by a MicroEJ Companion.
CommandServer-Socket	Allows the deployment of MicroEJ Applications through a local network connection.
NTP	Synchronizes the time of the device.
About	Displays version information about the VEE.
AppList	An application browser, can be used to start, stop or uninstall the applications, or display their descriptions and version information.
Forge Connect	Displays a desktop that allows the management of MicroEJ Applications using a connection to MICROEJ FORGE.
Settings	Displays the VEE parameters and settings.

5.4 Build Firmware

Prerequisite of this chapter: minimum understanding of *MicroEJ Module Manager*.

5.4.1 Workspace Build

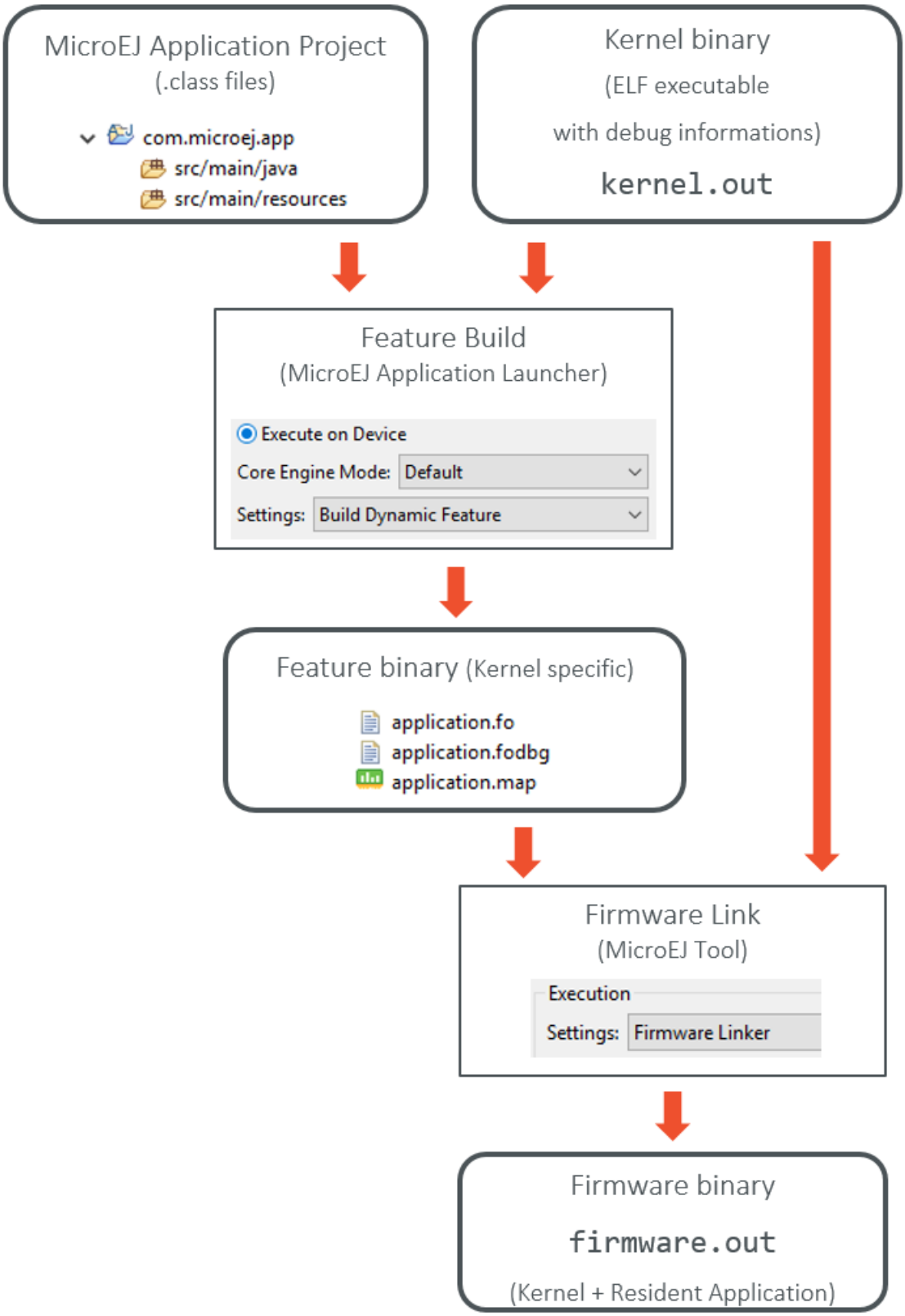
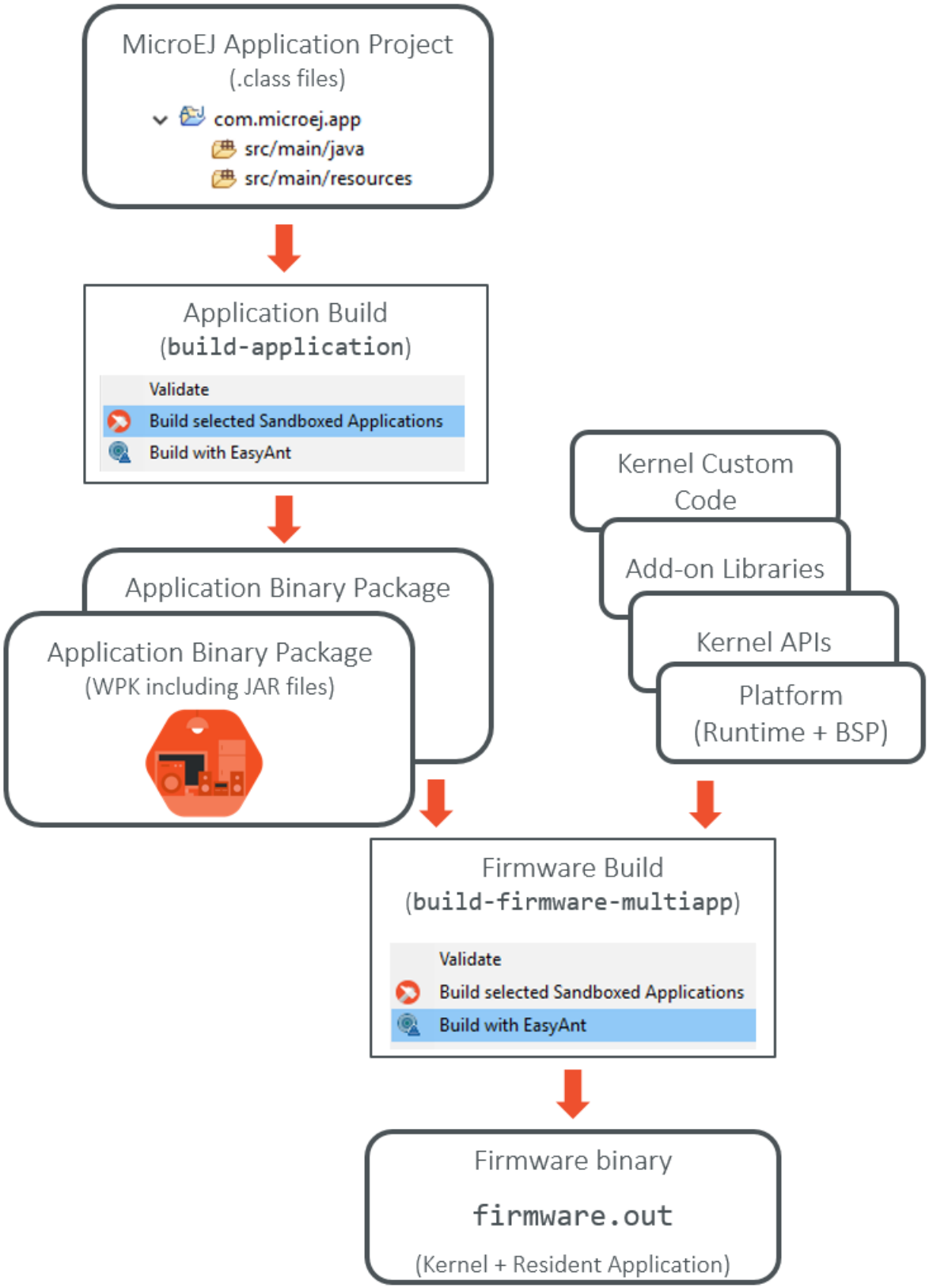


Fig. 5: Firmware Build Flow in MicroEJ SDK Workspace

5.4.2 Headless Build



5.4.3 Runtime Environment

A Firmware define a runtime environment which is the set of classes, methods and fields all applications are allowed to use. In most of the cases the runtime environment is an aggregation of several kernel APIs built with *module project* `build-runtime-api` skeleton.

```
<info organisation="myorg" module="mymodule" status="integration"
revision="1.0.0">
   <ea:build organisation="com.is2t.easyant.buildtypes" module="build-runtime-api" revision="2.+">
   <ea:plugin org="com.is2t.easyant.plugins" module="clean-artifacts" revision="2.+" />
   <ea:property name="clean.artifacts.max.keep" value="2" />
   <ea:property name="runtime.api.name" value="RUNTIME"/>
   <ea:property name="runtime.api.version" value="1.0"/>
   </ea:build>
</info>
```

The `runtime.api.name` property define the name of the runtime environment (it is required by the build type) , and the `runtime.api.version` property define it version. If the property `runtime.api.version` is not provided the build type computes it using the revision of the ivy module.

```
<dependencies>
   <dependency org="com.microej.kernelapi" name="edc" rev="[1.0.4-RC0,1.0.5-RC0[" transitive="false"/>
   <dependency org="com.microej.kernelapi" name="kf" rev="[2.0.1-RC0,2.0.2-RC0[" transitive="false"/>
   <dependency org="com.microej.kernelapi" name="bon" rev="[1.0.4-RC0,1.0.5-RC0[" transitive="false"/>
   <dependency org="com.microej.kernelapi" name="wadapps" rev="[1.2.2-RC0,1.2.3-RC0[" transitive="false
↪"/>
   <dependency org="com.microej.kernelapi" name="components" rev="[1.2.2-RC0,1.2.3-RC0[" transitive=
↪"false"/>
</dependencies>
```

This runtime environment aggregate all classes, methods and fields defined by `edc,kf,bon,wadapps,components` kernel APIs.

The documentation of a runtime environment is packaged into the Virtual Device as HTML javadoc (Help > MicroEJ Resource Center > Javadoc).

Specify the Runtime Environment of the Firmware

While building a firmware, two ways exist to specify the runtime environment:

- By using one or more ivy dependencies of `kernel API` artifacts. In this case we must set properties `runtime.api.name` and `runtime.api.version`.
- By using the ivy dependency `runtimeapi` module.

5.4.4 Resident Applications

A MicroEJ Sandboxed Application can be dynamically installed from a MicroEJ Forge instance or can be directly linked into the Firmware binary at built-time. In this case, it is called a Resident Application.

The user can specify the Resident Applications in two different ways:

- Set the property `build-systemapps.dropins.dir` to a folder with contains all the resident applications.
- Add ivy dependencyy on each resident application:

```
<dependency org="com.microej.app.wadapps" name="management"
rev="[2.2.2-RC0,3.0.0-RC0[" conf="systemapp->application"/>
```

All Resident Applications are also available for the Virtual Device, if a resident application should only be available for the Firmware, use an ivy dependency with the ivy configuration `systemapp-fw` instead of `systemapp`, like:

```
<dependency org="com.microej.app.wadapps" name="management" rev="[2.2.2-RC0,3.0.0-RC0[" conf="systemapp-
↪fw->application"/>
```

5.4.5 Advanced

MicroEJ Firmware `module.ivy`

The following section describes *module description file* (`module.ivy`) generated by the `build-firmware-multiapp` skeleton.

Ivy info

```
<info organisation="org" module="module" status="integration"
revision="1.0.0">
    <ea:build organisation="com.is2t.easyant.buildtypes" module="build-firmware-multiapp" revision="2.+
↪"/>
    <ea:property name="application.main.class" value="org.Main" />
    <ea:property name="runtime.api.name" value="RUNTIME" />
    <ea:property name="runtime.api.version" value="0.1.0" />
</info>
```

The property `application.main.class` is set to the fully qualified name of the main java class. The firmware generated from the skeleton defines its own runtime environment by using ivy dependencies on several `kernel API` instead of relying on a runtime environment module. As consequence, the `runtime.api.name` and `runtime.api.version` properties are specified in the firmware project itself.

Ivy Configurations

The `build-firmware-multiapp` build type requires the following configurations, used to specify the different kind of firmware inputs (see *Firmware Input and Output Artifacts*) as Ivy dependencies.

```
<configurations defaultconfmapping="default->default;provided->provided">
    <conf name="default" visibility="public"/>
    <conf name="provided" visibility="public"/>
    <conf name="platform" visibility="public"/>
    <conf name="vdruntime" visibility="public"/>
    <conf name="kernelapi" visibility="private"/>
    <conf name="systemapp" visibility="private"/>
    <conf name="systemapp-fw" visibility="private"/>
</configurations>
```

The following table lists the different configuration mapping usage where a dependency line is declared:

```
<dependency org="..." name="..." rev="..." conf="[Configuration Mapping]"/>
```

Table 1: Configurations Mapping for `build-firmware-multiapp` Build Type

Configuration Mapping	Dependency Kind	Usage
`provided->provided`	Foundation Library (`JAR`)	Expected to be provided by the platform. (e.g. `ej.api.*` module)
`default->default`	Add-On Library (`JAR`)	Embedded in the firmware only, not in the Virtual Device
`vdruntime->default`	Add-On Library (`JAR`)	Embedded in the Virtual Device only, not in the firmware
`default->default; vdruntime->default`	Add-On Library (`JAR`)	Embedded in both the firmware and the Virtual Device
`platform->platformDev`	Platform (`JPF`)	Platform dependency used to build the firmware and the Virtual Device. There are other ways to select the platform (see *Change the Platform used to Build the Firmware and the Virtual Device*)
`kernelapi->default`	Runtime Environment (`JAR`)	See *Runtime Environment*
`systemapp->application`	Application (`WPK`)	Linked into both the firmware and the Virtual Device as resident application. There are other ways to select resident applications (see *Resident Applications*)
`systemapp-fw->application`	Application (`WPK`)	Linked into the firmware only as resident application.

Example of minimal firmware dependencies.

The following example firmware contains one system app (`management`), and defines an API that contains all types, methods, and fields from `edc,kf,wadapps,components` .

```
<dependencies>
    <dependency org="ej.api" name="edc" rev="[1.2.0-RC0,2.0.0-RC0[" conf="provided" />
    <dependency org="ej.api" name="kf" rev="[1.4.0-RC0,2.0.0-RC0[" conf="provided" />
    <dependency org="ej.library.wadapps" name="framework" rev="[1.0.0-RC0,2.0.0-RC0[" />
    <dependency org="com.microej.library.wadapps.kernel" name="common-impl" rev="[3.0.0-RC0,4.0.0-RC0[" />
    <dependency org="com.microej.library.wadapps" name="admin-kf-default" rev="[1.2.0-RC0,2.0.0-RC0[" />
    <!-- Runtime API (set of Kernel API files) -->
    <dependency org="com.microej.kernelapi" name="edc" rev="[1.0.0-RC0,2.0.0-RC0[" conf="kernelapi->default"/>
    <dependency org="com.microej.kernelapi" name="kf" rev="[2.0.0-RC0,3.0.0-RC0[" conf="kernelapi->default"/>
    <dependency org="com.microej.kernelapi" name="wadapps" rev="[1.0.0-RC0,2.0.0-RC0[" conf="kernelapi->default"/>
    <dependency org="com.microej.kernelapi" name="components" rev="[1.0.0-RC0,2.0.0-RC0[" conf="kernelapi->default"/>
    <!-- System apps -->
    <dependency org="com.microej.app.wadapps" name="management"
    rev="[2.2.2-RC0,3.0.0-RC0[" conf="systemapp->application"/>
</dependencies>
```

Change the set of Properties used to Build a Firmware

The build use the file `build/common.properties` to configure the build process.

Change the Platform used to Build the Firmware and the Virtual Device

To build a firmware and a Virtual Device a platform must be specified. Four different ways are possible to do so:

- Use an Ivy dependency.

```
<dependency org="myorg" name="myname" rev="1.0.0" conf="platform->platformDev" transitive="false"/
↪>
```

- Copy/Paste a platform file into the folder defined by the property `platform-loader.target.platform.dropins` (by default it value is `dropins`).
- Set the property `platform-loader.target.platform.file`.

```
<ea:property name="platform-loader.target.platform.file" value="/path-to-a-platform-file/" />
```

- Set the property `platform-loader.target.platform.dir`.

```
<ea:property name="platform-loader.target.platform.dir" value="/path-to-a-platform-folder/" />
```

Build only a Firmware

Set the property `skip.build.virtual.device`

```
<ea:property name="skip.build.virtual.device" value="SET" />
```

Build only a Virtual Device

Set the property `virtual.device.sim.only`

```
<ea:property name="virtual.device.sim.only" value="SET" />
```

Build only a Virtual Device with a pre-existing Firmware

Copy/Paste the `.kpk` file into the folder `dropins`

5.5 Writing Kernel APIs

This section lists different ways to help to write `kernel.api` files.

5.5.1 Default Kernel APIs Derivation

MicroEJ provides predefined kernel API files for the most common libraries provided by a Kernel. These files are packaged as MicroEJ modules 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.

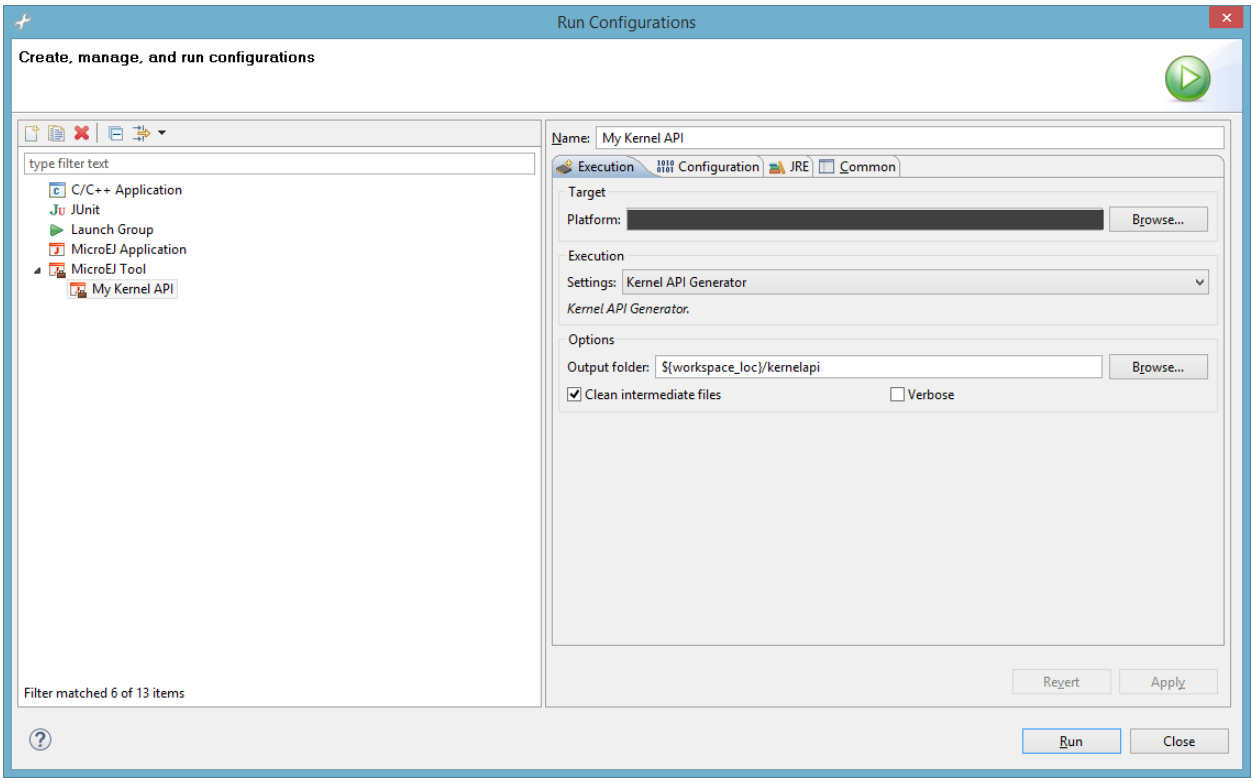
5.5.2 Build a Kernel API Module

- First create a new *module project* using the `microej-kernelapi` skeleton.
- Create the `kernel.api` file into the `src` folder.
- Right-click on the project and select Build Module .

5.5.3 Kernel API Generator

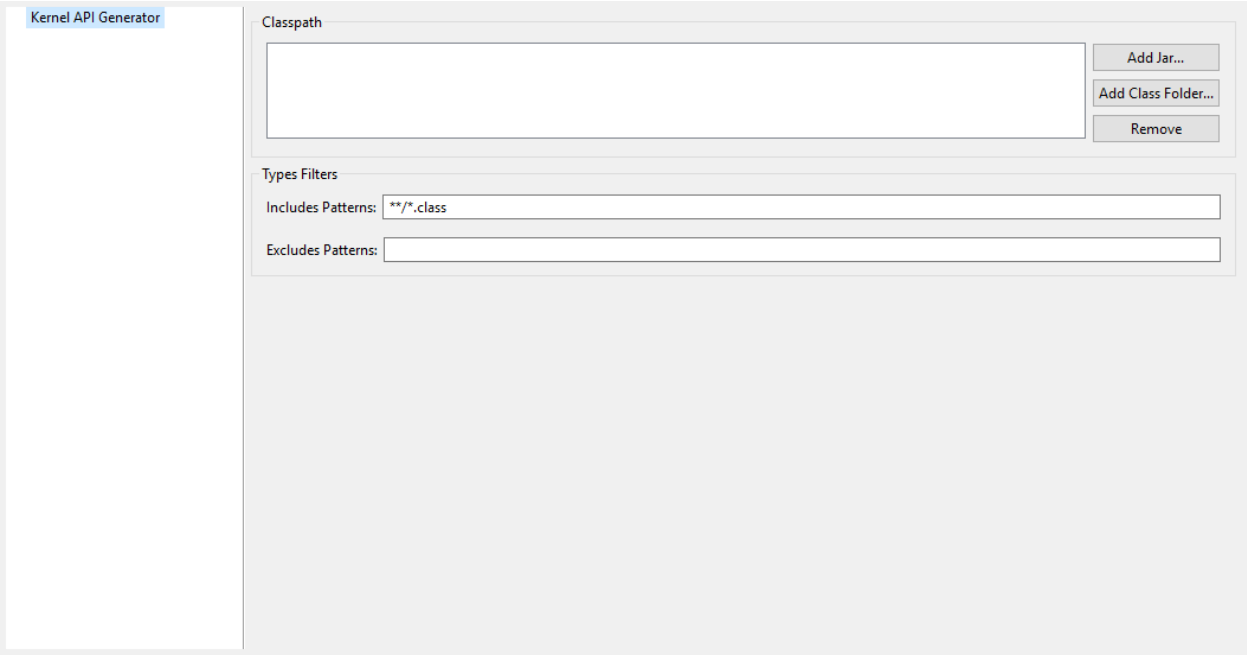
MicroEJ Kernel API Generator is a tool that help to generate a `kernel.api` file based on a Java classpath.

In MicroEJ SDK, create a new MicroEJ Tool launch, Run > Run Configurations > MicroEJ Tool , choose your Platform, select Kernel API Generator for the Settings options, and don't forget to set the output folder.



Define the classpath to use in the Configuration tab, and Press Run . A `kernel.api` file is generated in the output folder and it contains all classes, methods and fields found in the given classpath.

Category: Kernel API Generator



Group: Classpath

Option(list):

Option Name: `kernel.api.generator.classpath`

Default value: `(empty)`

Group: Types Filters

Option(text): Includes Patterns

Option Name: `kernel.api.generator.includes.patterns`

Default value: `**/*.class`

Description: Comma separated list of ANT Patterns for types to include.

Option(text): Excludes Patterns

Option Name: `kernel.api.generator.excludes.patterns`

Default value: `(empty)`

Description: Comma separated list of ANT Patterns for types to exclude.

5.6 Communication between Features

Features can communicate together through the use of shared interfaces. The mechanism is described in *Chapter Shared Interfaces* of the Application Developer's Guide.

5.6.1 Kernel Type Converters

The shared interface mechanism allows to transfer an object instance of a Kernel type from one Feature to an other. To do that, the Kernel must register a new converter (See *API Documentation* `Kernel.addConverter()` method).

5.7 API Documentation

The full API documentation of the Kernel & Features Foundation Library is available in MicroEJ SDK (Help > MicroEJ Resource Center > Javadoc > KF [version]).

5.8 Multi-Sandbox Enabled Libraries

A multi-Sandbox enabled library is a foundation or Add-On Library which can be embedded into the kernel and exposed as API. MicroEJ Foundation Libraries provided in MicroEJ SDK are already multi-Sandbox enabled. A stateless library - i.e. a library that does not contain any method modifying an internal global state - is multi-Sandbox enabled by default.

This section details the multi-Sandbox semantic that have been added to MicroEJ Foundation Libraries in order to be multi-Sandbox enabled.

5.8.1 MicroUI

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 `ej.microui.display.Display.requestShow(ej.microui.display.Displayable)` : after the successful permission check, it is assigned to the context owner.
- during a call to `ej.microui.MicroUI.callSerially(java.lang.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 `ej.microui.event.EventHandler` instance.

Automatically Reclaimed Resources

Instances of `ej.microui.display.ResourceImage`, `ej.microui.display.Font` are automatically reclaimed when a Feature is stopped.

5.8.2 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).

5.8.3 ECOM-COMM

Instances of `ej.ecom.io.CommConnection` are automatically reclaimed when a Feature is stopped.

5.8.4 FS

Instances of `java.io.FileInputStream`, `java.io.FileOutputStream` are automatically reclaimed when a Feature is stopped.

5.8.5 NET

Instances of `java.net.Socket`, `java.net.ServerSocket`, `java.net.DatagramSocket` are automatically reclaimed when a Feature is stopped.

5.8.6 SSL

Instances of `javax.net.ssl.SSLSocket` are automatically reclaimed when a Feature is stopped.

5.9 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 MicroEJ Multi-Sandbox Firmware composed of a Kernel and zero or more Features.

5.9.1 Enable the Testsuite

In an existing library project:

- 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="+" />
```

5.9.2 Add a KF Test

A KF test is a structured directory placed in the `src/test/projects` directory.

- Create a new directory for the KF test
- Within this directory, create the sub-projects:
  - Create a new *module project* for the Kernel using the `microej-javalib` skeleton,
  - Create a new *module project* for the Feature using the `application` skeleton,
  - Create a new *module project* for the Firmware 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:

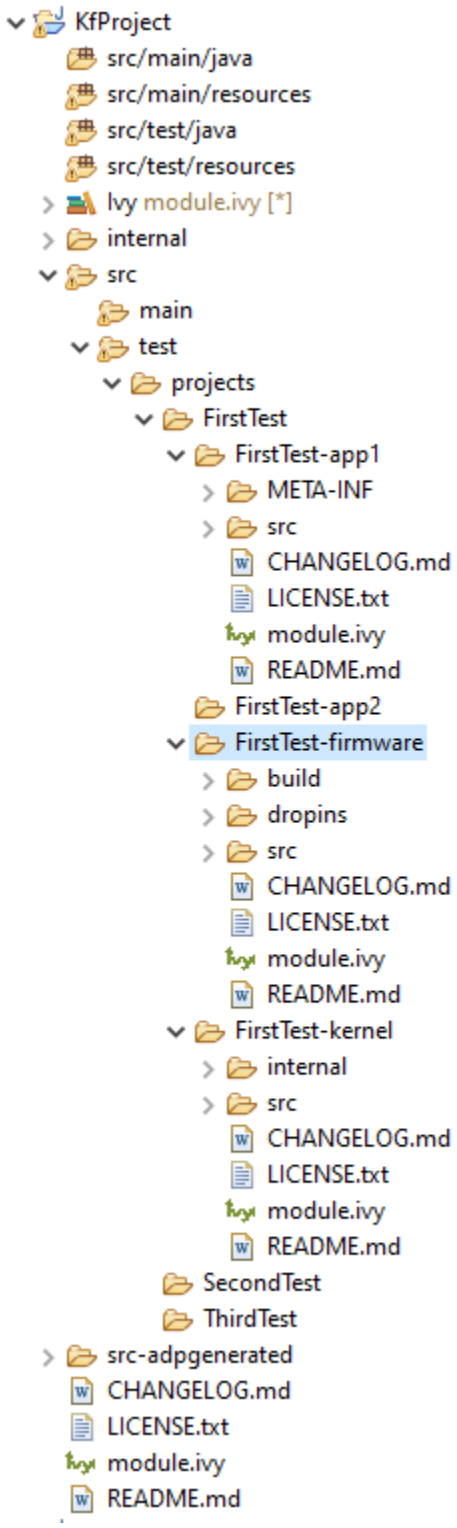


Fig. 6: KF Testsuite Overall Structure

All the projects will be built automatically in the right order based on their dependencies.

5.9.3 KF Testsuite Options

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`.

CHAPTER

SIX

TUTORIALS

6.1 Understand How to Build a MicroEJ Firmware and its Dependencies

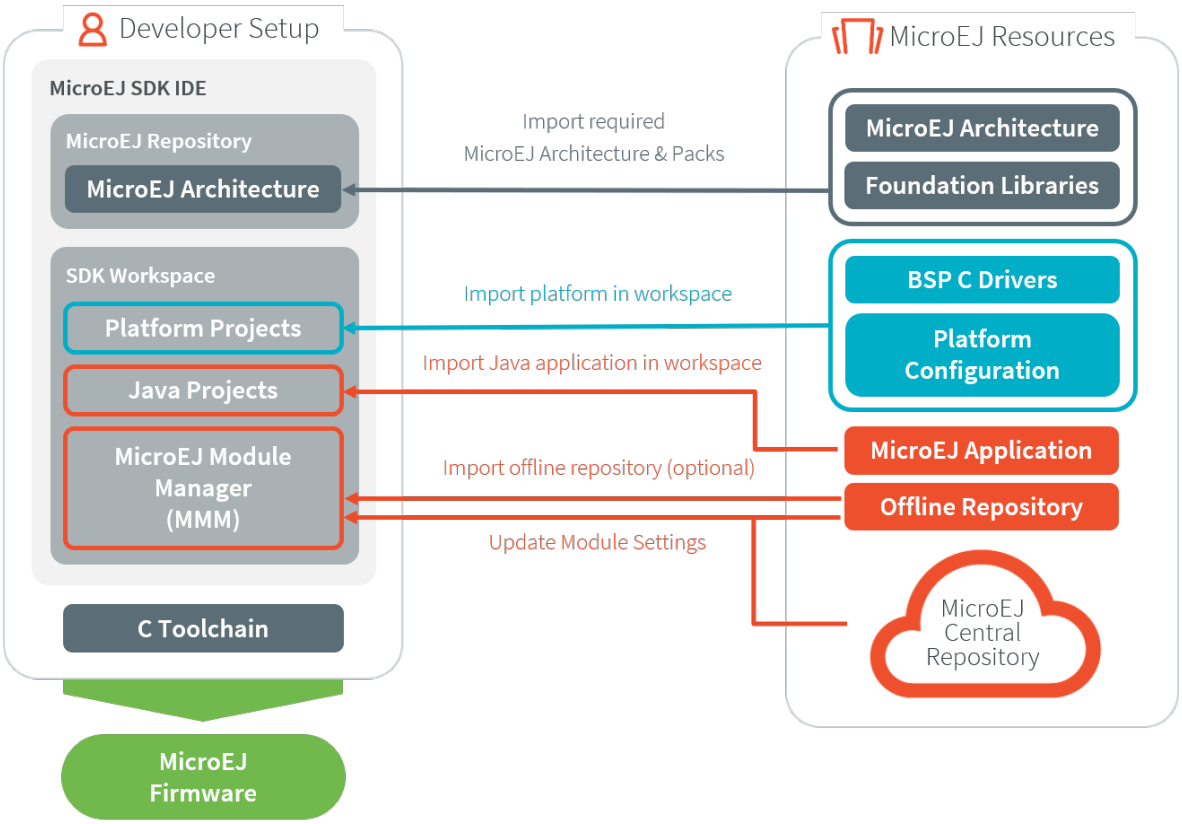
A MicroEJ Firmware is built from several input resources and tools. Each component has dependencies and requirements that must be carefully respected in order to build a firmware.

This document describes the components, their dependencies and the process involved in the build of a MicroEJ Firmware.

Good knowledge of the *MicroEJ Glossary* is required.

6.1.1 The Components

As depicted in the following image, several resources and tools are used to build a MicroEJ Firmware.



MicroEJ Architecture (.xpf, .xpfp)

A MicroEJ Architecture contains the runtime port to a target instruction set (ISA) and a C compiler (CC) and MicroEJ Foundation Libraries.

The MicroEJ Architectures are distributed into two formats:

- EVAL: evaluation license with runtime limitations (explained in SDK developer guide).
- PROD: production license (only MicroEJ sales & Customer Care team distribute this version).

The supported MicroEJ Architectures are listed here https://developer.microej.com/mej32-embedded-runtime-architectures/#arch

The MicroEJ Architecture is either provided from:

- For EVAL license only: the MicroEJ Repository at https://repository.microej.com/architectures/
- For PROD license only: SDK license site https://license.microej.com/ (MyProduct > Download additional products will list the downloads available). See *Production Licenses* for help with PROD license.
- MicroEJ sales or customer care team if the requested architecture is not listed as available.

See *MicroEJ Architecture Import* for a description on how to import a MicroEJ Architecture.

MicroEJ Platform Source (.zip)

This package includes:

- a C Board Support Package (BSP) with C drivers and an optional RTOS
- a MEJ32 Architecture
- the abstraction layers implementation of the platform architecture and foundations libraries
- the MicroEJ Simulator and its associated MicroEJ Mocks

The platform .zip files contain:

- `<platform>-configuration` : The configuration of the MicroEJ Platform
- `<platform>-bsp` : The C code for the board-specific files (drivers).
- `<platform>-fp` : Front Panel mockup for the simulator.

See *MicroEJ Platform Creation* to learn how to create a MicroEJ Platform using a MicroEJ Platform Source project.

Depending on the project's requirements, the MicroEJ Platform can be connected in various ways to the BSP; see *BSP Connection* for more information on how to do it.

MicroEJ Application

A MicroEJ Application is a Java project that can be configured (in the Run configurations ... properties):

1. to either run on:
   - a simulator (computer desktop),
   - a device (actual embedded hardware).
2. to setup:

- memory (example: Java heap, Java stack),
- foundation libraries,
- etc.

To run on a device, the application is compiled and optimized for a specific MicroEJ Platform. It generates a `microejapp.o` (native object code) linked with the `<platform>-bsp` project.

To import an existing MicroEJ Application as a zipped project in the SDK:

- Go to File > Import... > General > Existing Projects into Workspace > Select archive file > Browse... .
- Select the zip of the project (e.g. `x.zip`).
- And select Finish import.

See *Create a MicroEJ Standalone Application* for more information on how to create, configure, and develop a MicroEJ Application.

C Toolchain (GCC, KEIL, IAR, ...)

Used to compile and link the following files into the final firmware (binary, hex, elf, ... that will be programmed on the hardware):

- the `microejapp.o` (application),
- the `microejruntime.lib` or `microejruntime.a` (platform),
- the BSP C files (drivers).

Module Repository

A Module Repository provides the modules required to build MicroEJ Platforms and MicroEJ Applications.

- The MicroEJ Central Repository is an online repository of software modules (libraries, tools, etc.), see https://repository.microej.com/. This repository can also be used as an offline repository, see https://developer.microej.com/central-repository/.
- (Optional) It can be extended with an offline repository (`.zip`) that can be imported in the workspace (see *Use the Offline Repository*):

See *Module Repository* for more information.

Dependencies Between Components

- A MicroEJ Architecture targets a specific instruction set (ISA) and a specific C compiler (CC).
  - The C toolchain used for the MicroEJ Architecture must be the same as the one used to compile and link the BSP project and the MicroEJ Firmware.
- A MicroEJ Platform consists of the aggregation of both a MicroEJ Architecture and a BSP with a C toolchain.
  - Changing either the MicroEJ Architecture or the C toolchain results in a change of the MicroEJ Platform.
- A MicroEJ Application is independent of the MicroEJ Architecture.
  - It can run on any MicroEJ Platform as long the platform provides the required APIs.

- To run a MicroEJ Application on a new device, create a new MicroEJ Platform for this device with the exact same features. The MicroEJ Application will not require any change.

6.1.2 How to Build

The process to build a MicroEJ Firmware is two-fold:

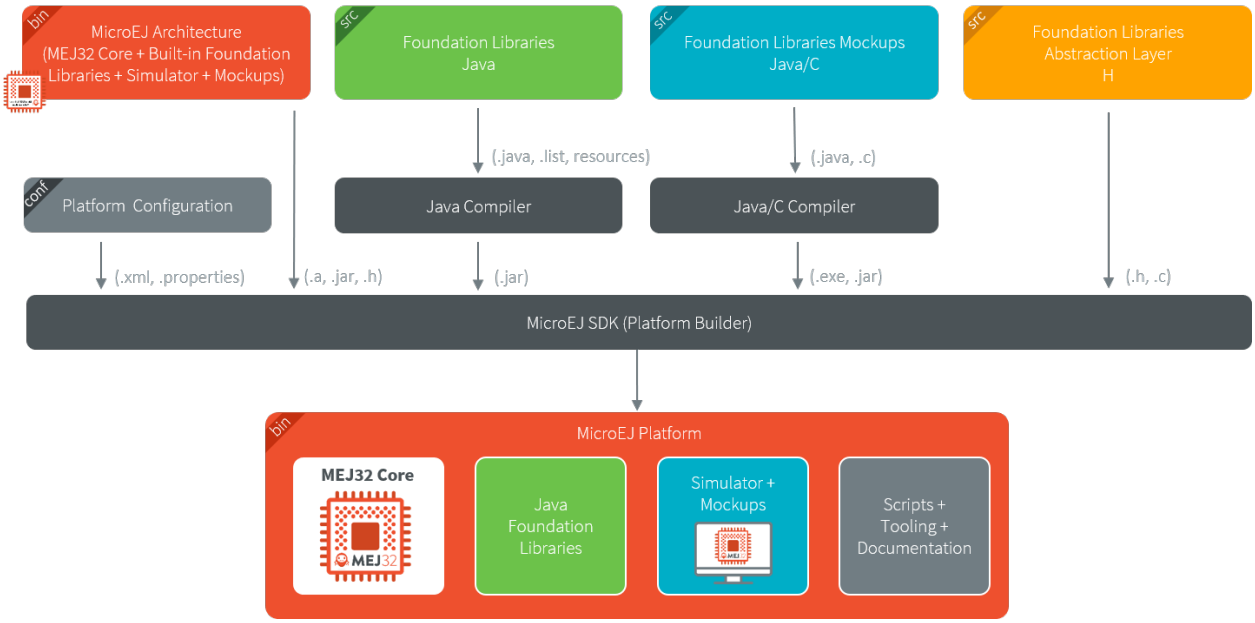
1. Build a MicroEJ Platform
2. Build a MicroEJ Application

The MicroEJ Application is compiled against the MicroEJ Platform to produce the MicroEJ Firmware deployed on the target device.

Note: The MicroEJ Application also runs onto the MicroEJ Simulator using the mocks provided by the MicroEJ Platform.

Build a MicroEJ Platform

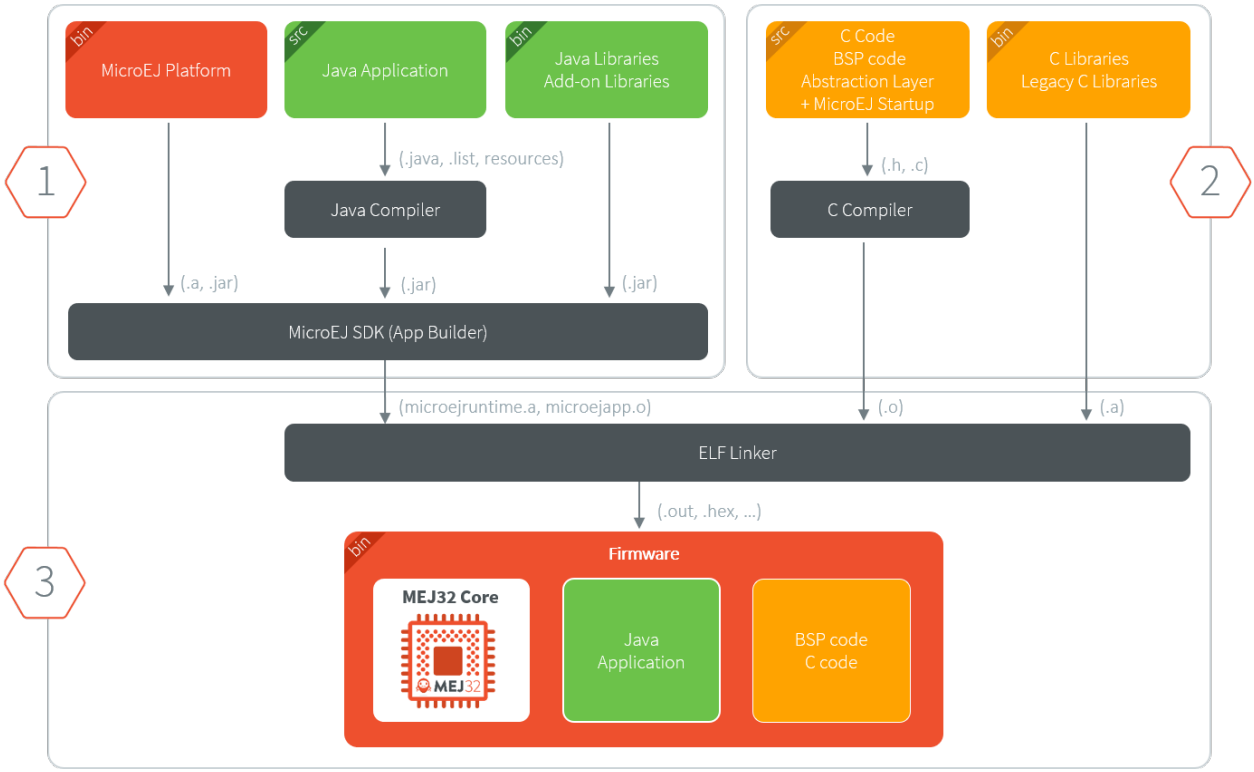
The next schema presents the components and process to build a MicroEJ Platform.



Build a MicroEJ Firmware

The next schema presents the steps to build a MicroEJ Mono-Sandbox Firmware (previously known as MicroEJ Single-app Firmware). The steps are:

1. Build the MicroEJ Application into a `microejapp.o` using MicroEJ SDK
2. Compile the BSP C sources into `.o` using the C toolchain
3. the BSP (`.o`) and the MicroEJ Application (`microejapp.o`) and the MicroEJ Platform (`microejruntime.a`) are linked by the C toolchain to produce a final ELF or binary called MicroEJ Firmware (e.g. `application.out`).



See *BSP Connection* for more information on how to connect the MicroEJ Platform to the BSP.

Dependencies Between Processes

- Rebuild the MicroEJ Platform:
  - When the MicroEJ Architecture (`.xpf`) changes.
  - When a MicroEJ Foundation Library (`.xpfp`) changes.
  - When a Foundation Library changes, either when
    * The public API (`.java` or `.h`) changes.
    * The front-panel or mock implementation (`.java`) changes.
- Rebuild of the MicroEJ Platform is not required:
  - When the implementation (`.c`) of a Foundation Library changes.
  - When the BSP (`.c`) changes.
  - When the MicroEJ Application changes.
- Rebuild MicroEJ Application:
  - When it changes.
  - When the MicroEJ Platform changes.
- Rebuild the BSP:
  - When it changes.
  - When the MicroEJ Platform changes.
- Rebuild the MicroEJ Firmware:

- When the MicroEJ Application (`microejapp.o`) changes.
- When the BSP (`*.o`) changes.
- When the MicroEJ Platform (`microejruntime.a`) changes.

6.1.3 Get Support

If any questions, feel free to contact our support team with the following information (the table below is an example):

Delivery	Name
MicroEJ SDK	Distribution 20.07 / Version 5.2.0 (see *Determine the MicroEJ Studio/SDK Version*)
MEJ32 XPF	ARM Cortex-M4 / IAR DEV
Platform	1.0.0
Application	1.2.4
Module Repository	https://repository.microej.com/packages/repository/2.5.0/microej-5_0-2.5.0.zip
C compiler	IAR 8.40.1

6.2 Create a MicroEJ Platform for a Custom Device

6.2.1 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. The MicroEJ Architectures are provided by MicroEJ SDK.

A MicroEJ Platform is a MicroEJ Architecture port for a custom device. It contains the MicroEJ configuration and the BSP (C source files).

MicroEJ Corp. provides MicroEJ Evaluation Architectures at https://repository.microej.com/architectures/, and MicroEJ Platform demo projects for various evaluation boards at https://repository.microej.com/index.php?resource=JPF.

We recommend reading the *MicroEJ Firmware* section to get an overview of MicroEJ Firmware build flow.

The following document assumes the reader is familiar with the *Platform Developer Guide*.

Each MicroEJ Platform is specific to:

- a MicroEJ Architecture (MCU ISA and C compiler)
- an optional RTOS (e.g. FreeRTOS - note: the MicroEJ OS 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)

In this document we will address the following items:

- MicroEJ Platform Configuration project (in MicroEJ SDK)
- MicroEJ Simulator (in MicroEJ SDK)
- Platform BSP (in a C IDE/Compiler like GCC/KEIL/IAR)

The MicroEJ Platform relies on C drivers (aka low level LL drivers) for each of the platform feature. These drivers are implemented in the platform BSP project. This project is edited in the C compiler IDE/dev environment (e.g. KEIL, GCC, IAR). E.g. the MicroUI library LED feature will require a `LLUI_LED.c` that implements the native on/off IO drive.

The following sections explain how to create a MicroEJ Platform for a custom device starting from an existing MicroEJ Platform project whether it is configured for the same MCU/RTOS/C Compiler or not.

In the following, we assume 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
- See the Platform Qualification Tools used to qualify MicroEJ Platforms.

6.2.2 A MicroEJ Platform Project is already available for the same MCU/RTOS/C Compiler

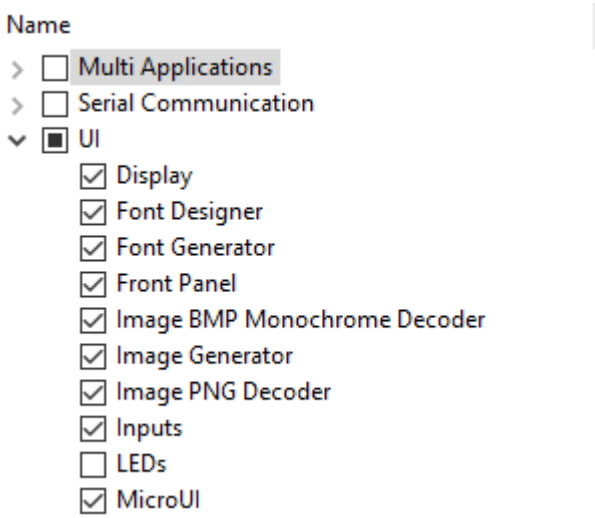
This is the fastest way: the MicroEJ Platform is usually provided for a silicon vendor evaluation board. Import this platform in MicroEJ SDK.

As the MCU, RTOS and compiler are the same, only the device specific code needs to be changed (external RAM, external oscillator, communication interfaces).

Platform

In MicroEJ SDK

- modify the `.platform` from the MicroEJ Platform (`xxx-configuration` project) to match the device features and its associated configuration (e.g. `UI->Display`).



More details on available modules can be found in the *Platform Developer 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 JVM can delegate the relevant operations to the provided BSP C functions.

Simulator

In MicroEJ SDK

- modify the existing Simulator Front Panel `xxx-fp` project

6.2.3 A MicroEJ Platform Project is not available for the same MCU/RTOS/C Compiler

Look for an available MicroEJ Platform 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 MicroEJ Platform

- In MicroEJ SDK: modify the platform configuration.
- in the C IDE: start from an empty project that match with the MCU.

Or to start from scratch a new MicroEJ Platform

- In MicroEJ SDK: create the MicroEJ Platform and refer to the selected MicroEJ Platform as a model for implementation. (refer to *MicroEJ Platform Configuration*)
- 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 JVM for the 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 MicroEJ Platform in the `main()` function.

6.2.4 Platform Validation

Use the Platform Qualification Tools to qualify the MicroEJ Platform built.

6.2.5 Further Assistance Needed

Please note that porting MicroEJ to a new device is also something that is part of our engineering services. Consider contacting sales@microej.com to request a quote.

6.3 Create a MicroEJ Firmware From Scratch

This tutorial 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 tutorial, 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 tutorial. Everything is emulated on the developer's PC.

The tutorial should take 1 hour to complete (excluding the installation time of MicroEJ SDK and Windows Subsystem Linux (WSL)).

6.3.1 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 tutorial 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*.

6.3.2 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 tutorial 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 tutorial:

- 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 tutorial 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 tutorial.

6.3.3 Prerequisites

- MicroEJ SDK version 5.1.0 or higher (distribution 19.05). Can be downloaded from https://developer.microej.com/ (direct link)
- Windows 10 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.

6.3.4 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 tutorial 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.

6.3.5 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 tutorial 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.

6.3.6 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.

```
diff --git a/FreeRTOS/Demo/CORTEX_LM3S811_GCC/standalone.ld b/FreeRTOS/Demo/CORTEX_LM3S811_GCC/
↪standalone.ld

index 8ee3fe2f8..b771ff834 100644
--- a/FreeRTOS/Demo/CORTEX_LM3S811_GCC/standalone.ld
+++ b/FreeRTOS/Demo/CORTEX_LM3S811_GCC/standalone.ld
@@ -42,7 +42,15 @@ SECTIONS
         _etext = .;
     } > FLASH
```

(continues on next page)

(continued from previous page)

```
-    .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
```

(continues on next page)

(continued from previous page)

```
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.

6.3.7 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).

The following code implements the `putchar(3)` and `puts(3)` functions:

```
#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');
}
```

And here is the patch that implements both functions and prints `Hello World`.

```
diff --git a/FreeRTOS/Demo/CORTEX_LM3S811_GCC/main.c b/FreeRTOS/Demo/CORTEX_LM3S811_GCC/main.c
index 107517c00..3ea4c23a4 100644
--- a/FreeRTOS/Demo/CORTEX_LM3S811_GCC/main.c
+++ b/FreeRTOS/Demo/CORTEX_LM3S811_GCC/main.c
@@ -134,9 +134,25 @@ SemaphoreHandle_t xButtonSemaphore;
 QueueHandle_t xPrintQueue;

 /*-----------------------------------------------------------*/
+#define UART0BASE ((volatile int*) 0x4000C000)
+
+int putchar (int c){
+    (*UART0BASE) = c;
+    return c;
+}
```

(continues on next page)

(continued from previous page)

```
+
+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
QEMU: Terminated
```

With this two functions implemented, `printf(3)` is also available.

```
diff --git a/FreeRTOS/Demo/CORTEX_LM3S811_GCC/main.c b/FreeRTOS/Demo/CORTEX_LM3S811_GCC/main.c
index 76440e60e..f24007597 100644
--- 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.");
```

(continues on next page)

(continued from previous page)

```
+       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.

6.3.8 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 MicroEJ Core Engine included in the MicroEJ Architecture.

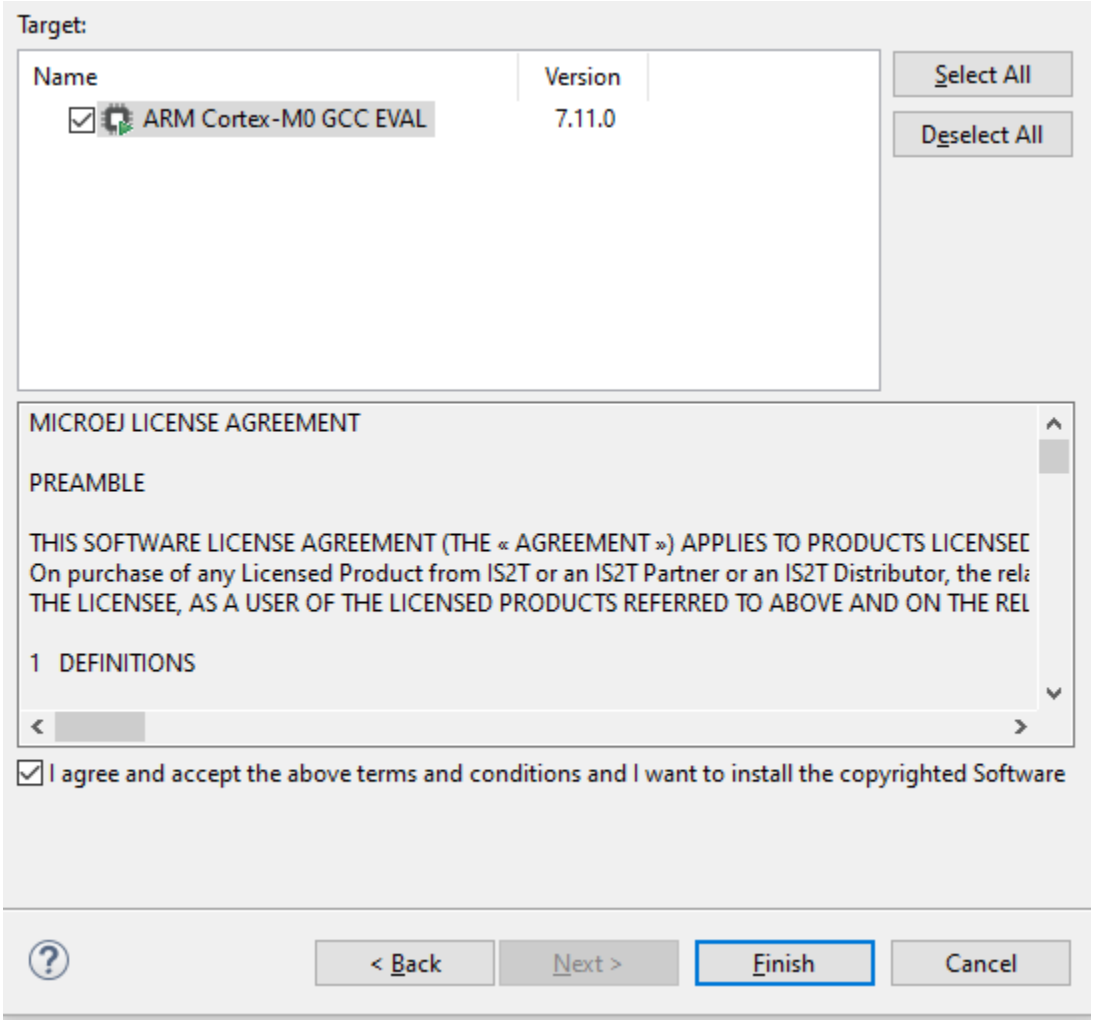
The list of MicroEJ Architectures supported is listed at https://docs.microej.com/en/latest/PlatformDeveloperGuide/appendix/toolchain.html. MicroEJ Evaluation Architectures provided by MicroEJ Corp. can be downloaded from MicroEJ Architectures Repository.

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/architectures/com/microej/architecture/CM0/CM0_GCC48/flopi0G22/7.11.0/flopi0G22-7.11.0-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

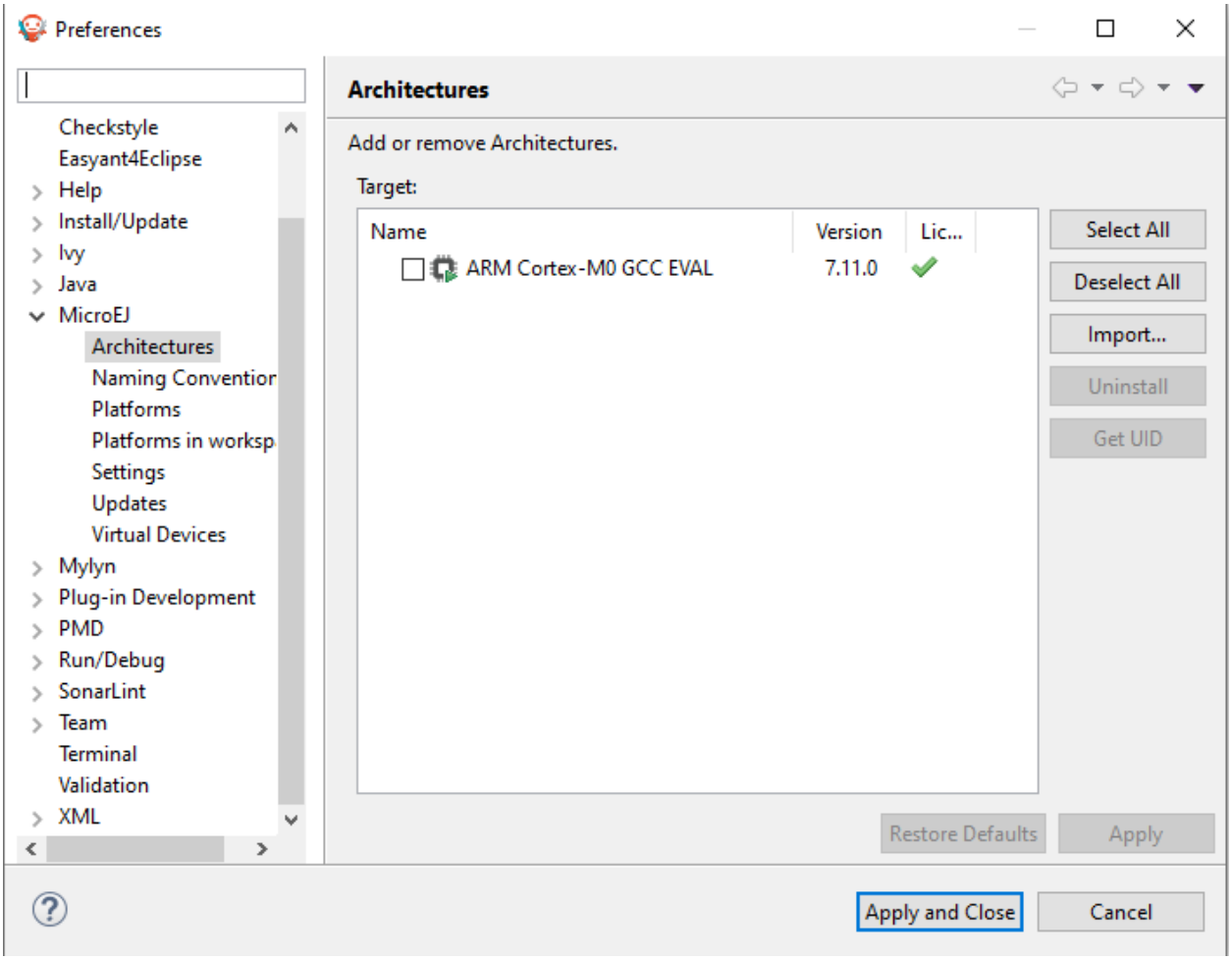


Install an Evaluation License

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 support@microej.com.
    - 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.

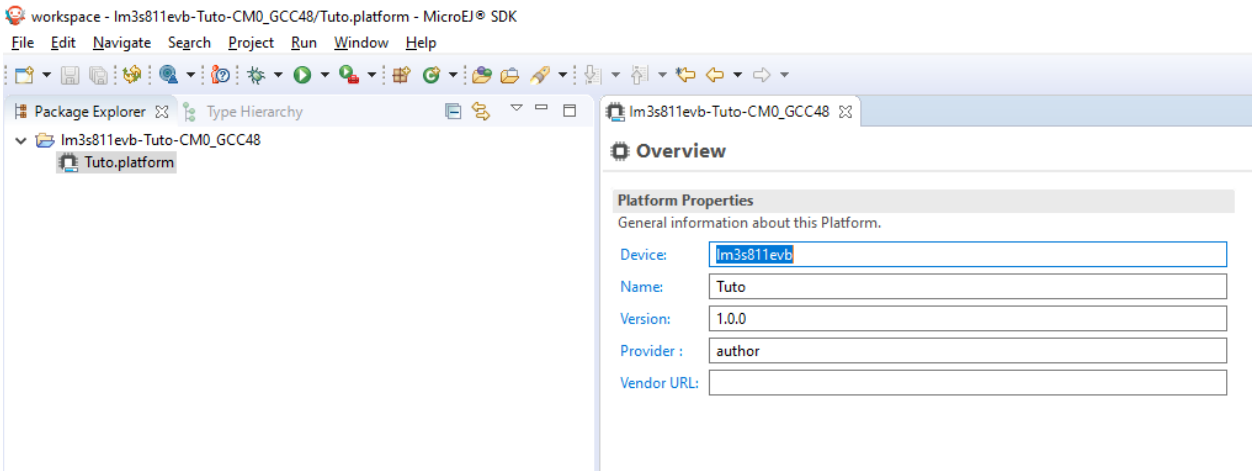


Create the MicroEJ Platform

This step describes how to create a new *MicroEJ Platform* using the MicroEJ Architecture previously imported.

1. Select File > New > MicroEJ 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`



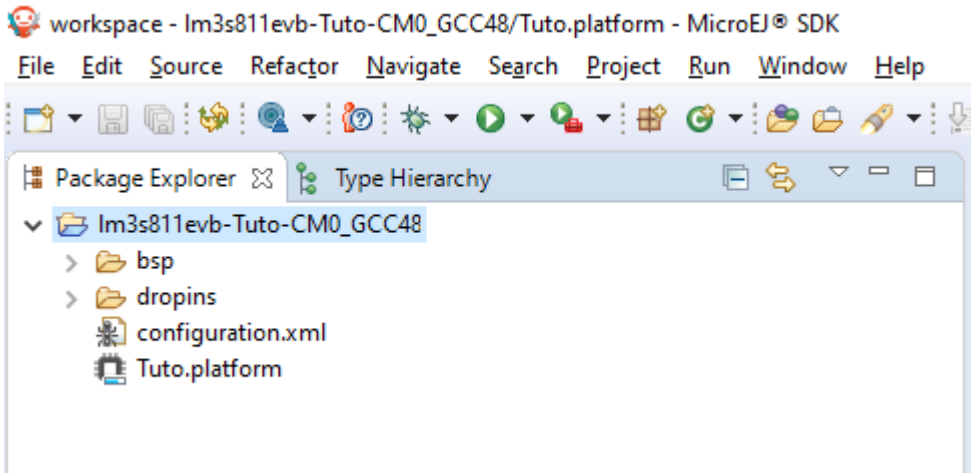
Setup the MicroEJ Platform

This step describes how to configure the MicroEJ Platform previously created.

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 tutorial, 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 tutorial, 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.



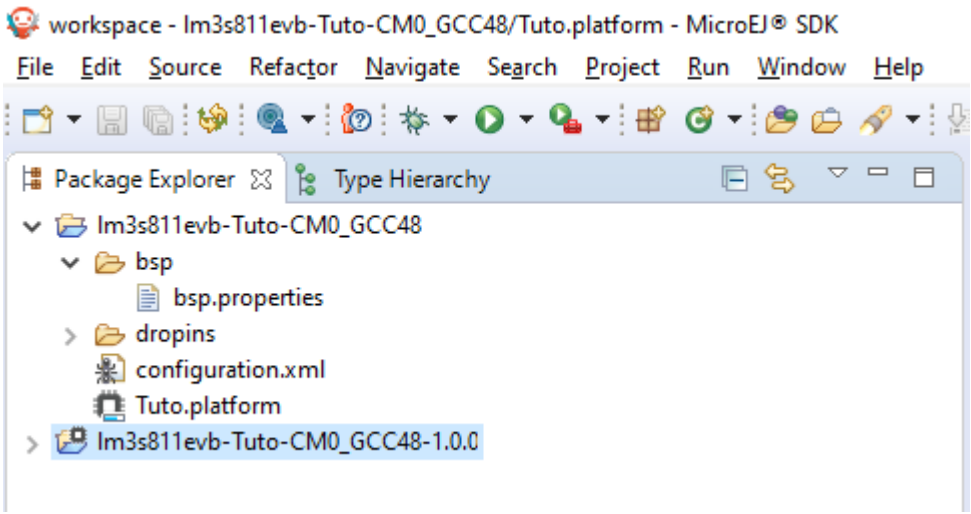
Note: The `content` directory contains files that must be installed in a MicroEJ Platform configuration directory (the directory that contains the `.platform` file). It can be automatically downloaded using the following command line:

```
svn checkout https://github.com/MicroEJ/PlatformQualificationTools/trunk/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
```

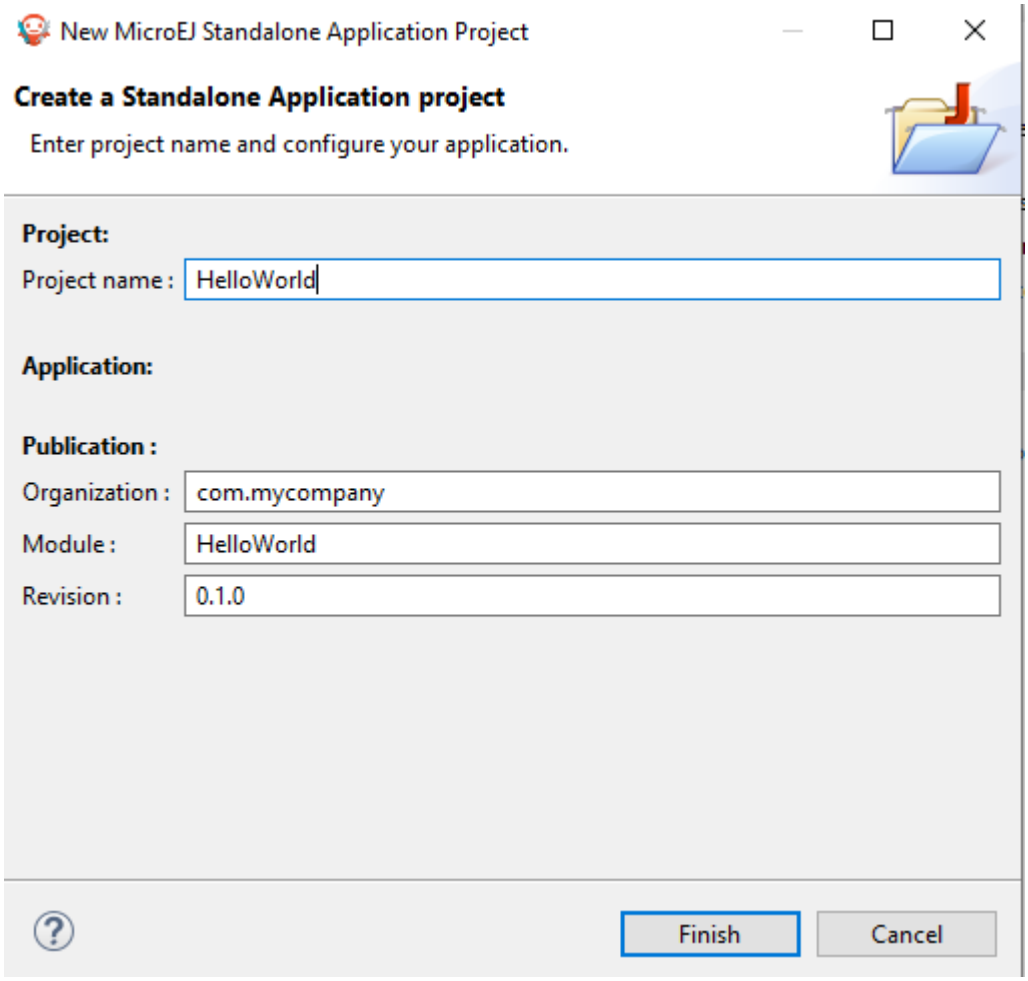
3. Open the `.platform` file and click on `Build Platform`. The MicroEJ Platform will appear in the workspace.



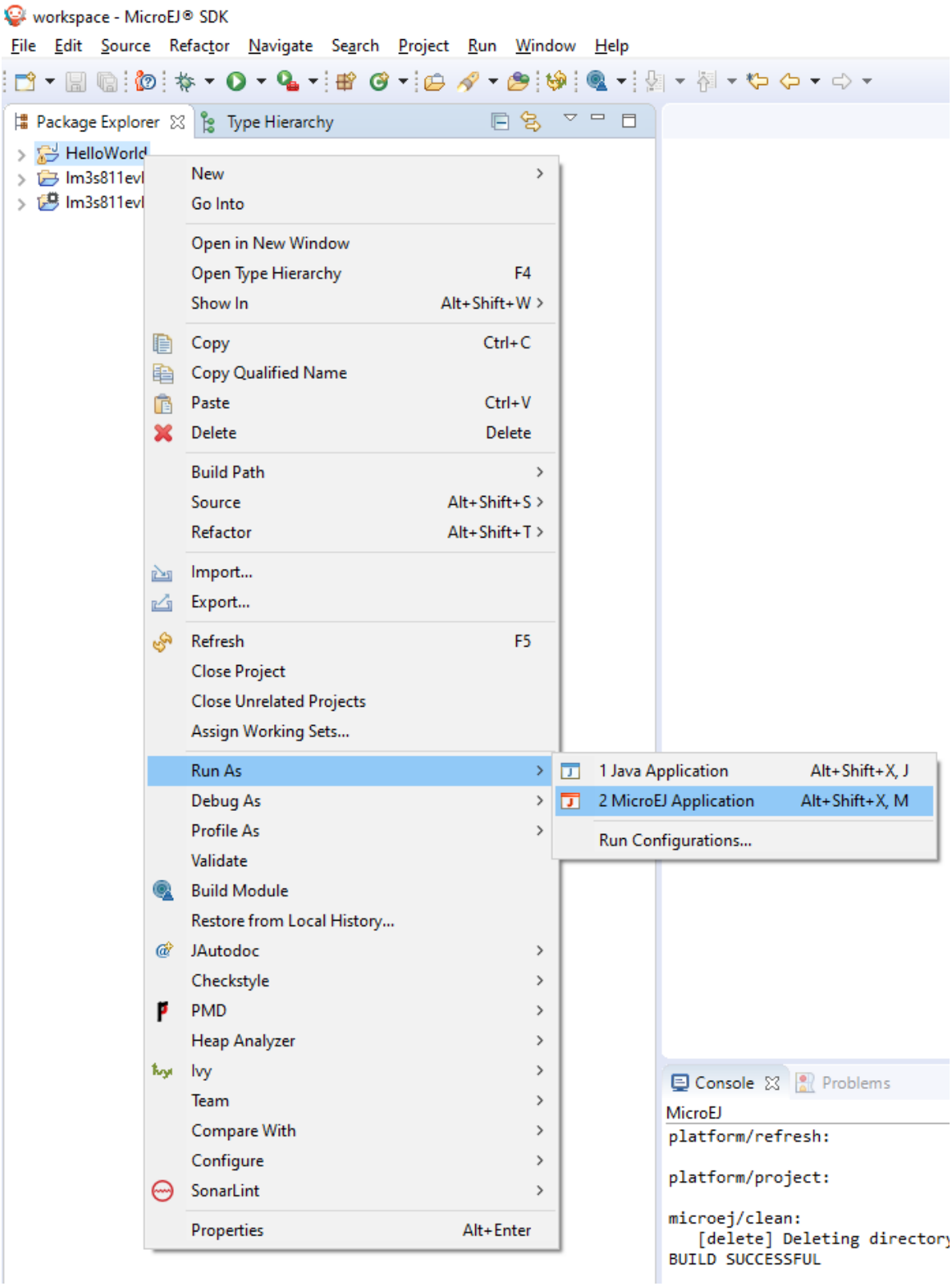
At this point, the MicroEJ Platform is ready to be used to build MicroEJ Applications.

6.3.9 Create MicroEJ Application HelloWorld

1. Select File > New > MicroEJ 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
```

6.3.10 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/common.properties` file.

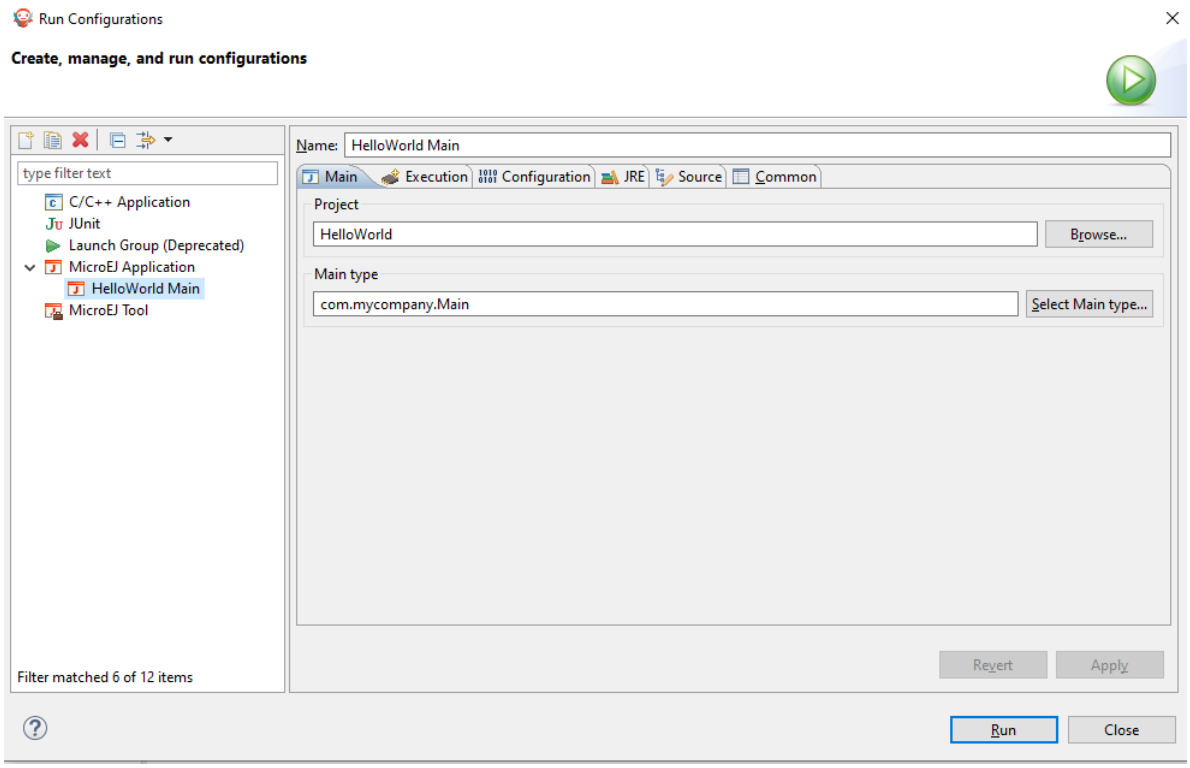
1. Create a file `HelloWorld/build/emb.properties` with the following content:

```
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
```

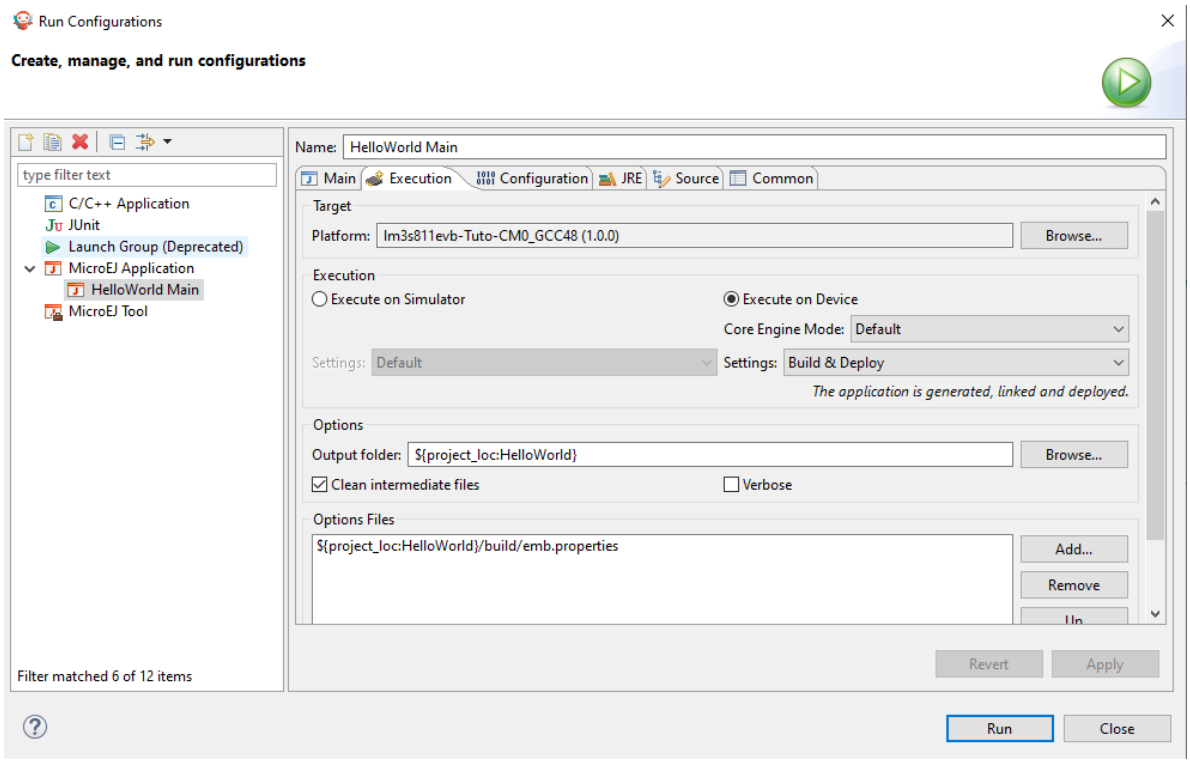
Note: Assuming the WSL current directory is `FreeRTOS/FreeRTOS/Demo/CORTEX_LM3S811_GCC`, use the following command to find the `deploy.bsp.root.dir` path with proper escaping:

```
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.

6.3.11 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 MicroEJ Core Engine
- Build and link the firmware with the MicroEJ Runtime and MicroEJ Application

Minimal Low Level APIs

The purpose of this tutorial is to demonstrate how to develop a minimal MicroEJ Architecture, it is not to develop a complete MicroEJ Architecture. Therefore this tutorial 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 1: 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 2: 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;
}

int32_t LLMJVM_IMPL_shutdown(void)
{
        return LLMJVM_OK;
```

(continues on next page)

(continued from previous page)

```
}
```

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 MicroEJ Core Engine

The MicroEJ 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`.

```
diff --git a/FreeRTOS/Demo/CORTEX_LM3S811_GCC/main.c b/FreeRTOS/Demo/CORTEX_LM3S811_GCC/main.c
index d5728f976..644710120 100644
--- 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) {
 }

 #include <stdio.h>
+#include "sni.h"

 int main( void )
 {
        printf("Hello, World! printf function is working.\n");
```

(continues on next page)

(continued from previous page)

```
+       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:

```
index 814cc6f7e..bbcad47b3 100644
--- 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
+
 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
```

(continues on next page)

(continued from previous page)

```
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 Application* on a baremetal device (without a RTOS) but this is not the purpose of this tutorial.

Instead, this tutorial 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 tutorial assumes that everything is done in the `CORTEX_LM3S6965_GCC` folder.

Then update the linker script `standlone.ld`:

```
diff --git a/FreeRTOS/Demo/CORTEX_LM3S6965_GCC/standalone.ld b/FreeRTOS/Demo/CORTEX_LM3S6965_GCC/
↪standalone.ld
index b771ff834..e3719ea30 100644
--- a/FreeRTOS/Demo/CORTEX_LM3S6965_GCC/standalone.ld
+++ b/FreeRTOS/Demo/CORTEX_LM3S6965_GCC/standalone.ld
@@ -28,8 +28,8 @@

 MEMORY
 {
-    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'
```

(continues on next page)

(continued from previous page)

```
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.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 tutorial uses the later.

```diff
diff --git a/FreeRTOS/Demo/CORTEX_LM3S6965_GCC/makedefs b/FreeRTOS/Demo/CORTEX_LM3S6965_GCC/makedefs
index 1a8f4dab5..66b482804 100644
--- 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}
```

(continues on next page)

(continued from previous page)

```
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/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:

```
diff --git a/FreeRTOS/Demo/CORTEX_LM3S6965_GCC/makedefs b/FreeRTOS/Demo/CORTEX_LM3S6965_GCC/makedefs
index 66b482804..80f812829 100644
--- 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=}}}    \
```

(continues on next page)

(continued from previous page)

```
                        ${LDFLAGS} -o ${@} --start-group ${^} \
-                       '${LIBC}' '${LIBGCC}' --end-group;    \
+                       '${LIBM}' '${LIBC}' '${LIBGCC}' --end-group; \
         fi
        @${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 tutorial uses the `libnosys.a` which provides stub implementation for various functions.

```
diff --git a/FreeRTOS/Demo/CORTEX_LM3S6965_GCC/makedefs b/FreeRTOS/Demo/CORTEX_LM3S6965_GCC/makedefs
index 80f812829..9de8150a5 100644
--- 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 tutorial uses the end of the `.bss` segment as the beginning of the C heap.

```diff
diff --git a/FreeRTOS/Demo/CORTEX_LM3S6965_GCC/standalone.ld b/FreeRTOS/Demo/CORTEX_LM3S6965_GCC/
↪standalone.ld
index e3719ea30..e86294b5f 100644
--- 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:

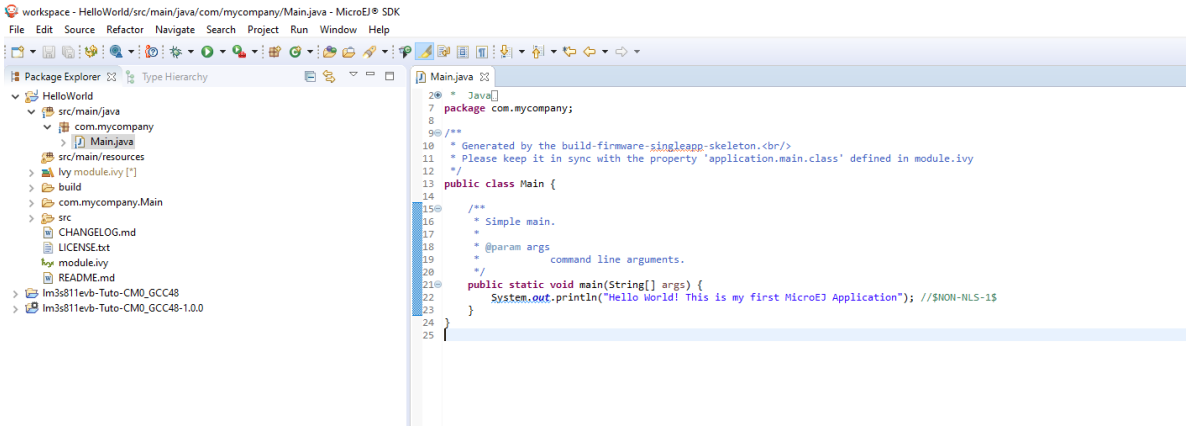
```
qemu-system-arm -M lm3s6965evb -nographic -kernel gcc/RTOSDemo.bin
```

```
Hello, World! printf function is working.
Hello World!
QEMU: Terminated // press Ctrl-a x to end the QEMU session
```

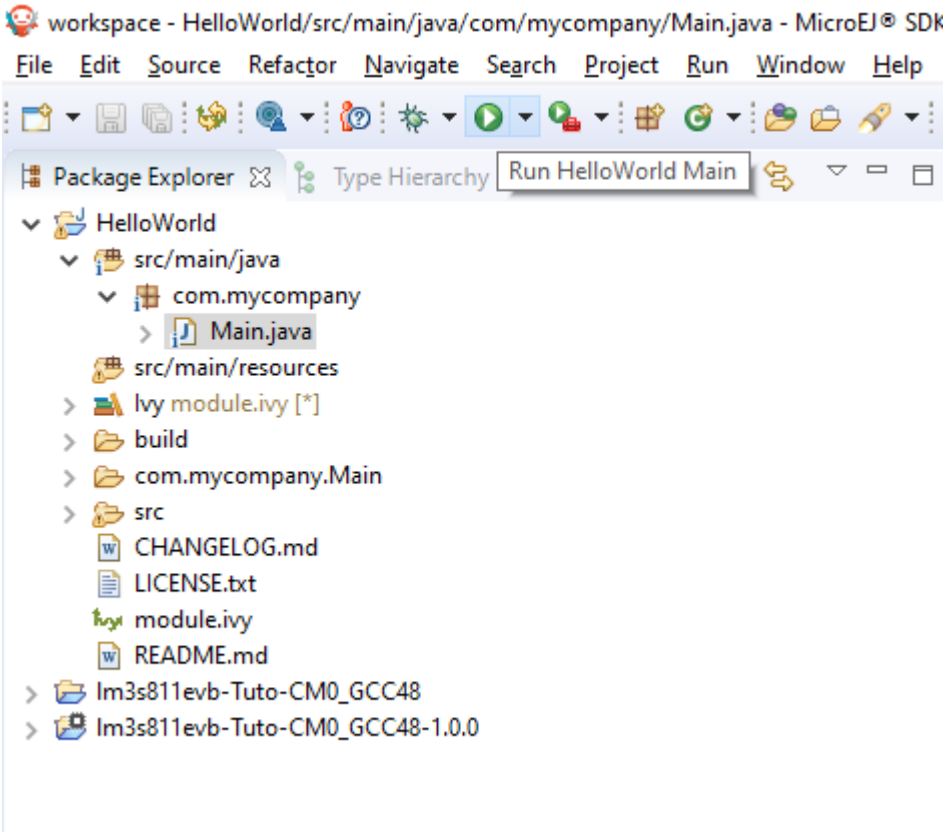
The first `Hello, World!` is from the `main.c` and the second one from the MicroEJ Application.

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.
Hello World! This is my first MicroEJ Application
QEMU: Terminated
```

Congratulations!

At this point of the tutorial:

- 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.

6.4 Setup an Automated Build using Jenkins and Artifactory

This tutorial explains how to setup an environment for automating *MicroEJ Module build* and deployment using Jenkins and JFrog Artifactory.

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

The tutorial should take 2 hours to complete.

6.4.1 Intended Audience

The audience for this document is engineers who are in charge of integrating *MicroEJ Module Manager (MMM)* to their continuous integration environment.

In addition, this tutorial should be of interest to all developers wishing to understand how MicroEJ works with headless module builds.

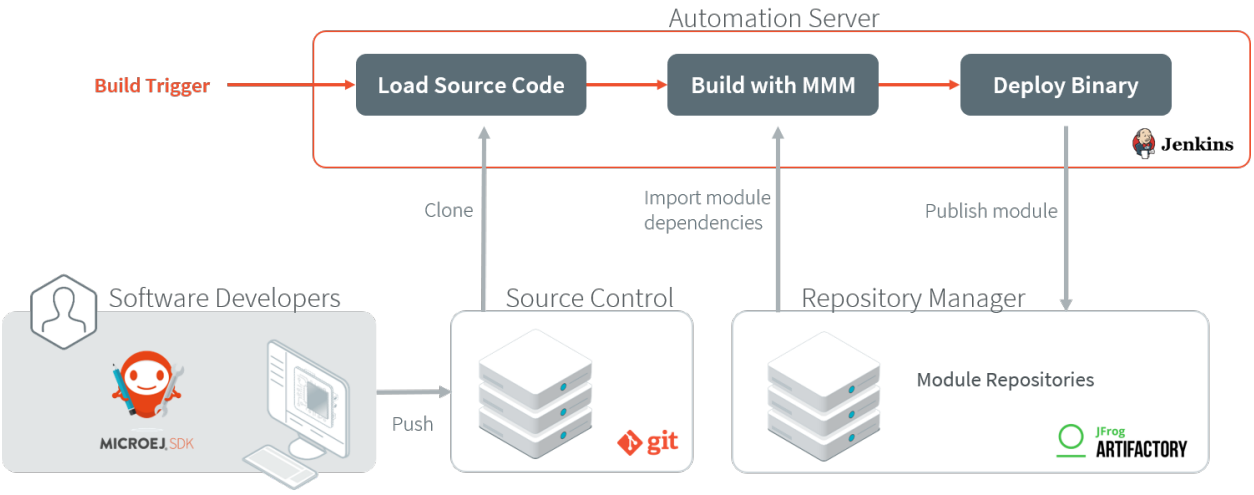
For those who are only interested by command line module build, consider using the CommandLineBuild tool.

6.4.2 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



6.4.3 Prerequisites

- MicroEJ SDK `4.1.5` or higher.
- Git `2.x` installed, with Git executable in path. We recommend installing Git Bash if your operating system is Windows (https://git-for-windows.github.io/).
- Apache Ant `1.9.x` installed (https://ant.apache.org/bindownload.cgi).
- Java Development Kit (JDK) `1.8.x`.

This tutorial was tested with Jenkins `2.235.3` and Artifactory `6.20.1`.

6.4.4 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. Install and setup MicroEJ build tools, Jenkins and Artifactory
2. Create a Jenkins job template for MMM builds
3. Create a simple MicroEJ module (Hello World)
4. Create a new Jenkins job for the Hello World module
5. Build the module

For the purposes of simplifying the steps, this tutorial will be performed locally on a single machine.

Artifactory will host MicroEJ modules, divided in 3 repositories:

- `microej-module-repository` : repository initialized with pre-built MicroEJ modules, a mirror of the *Central Repository*
- `microej-build-repository` : repository initialized with build scripts and tools exported from MicroEJ SDK
- `libs-snapshot-local` : repository where will be published custom modules

6.4.5 Install the Build Tools

This section assumes the prerequisites have been properly installed.

1. Locate your JDK installation directory (typically something like `C:\Program Files\Java\jdk1.8.0_[version]` on Windows).
2. Set the environment variable `JAVA_HOME` to point to the `bin` directory (for example `C:\Program Files\Java\jdk1.8.0_[version]\bin`).
3. Set the environment variable `JRE_HOME` to point to the `jre` directory (for example `C:\Program Files\Java\jdk1.8.0_[version]\jre`).
4. Download the pre-configured settings file by cloning the following git repository:

   ```
   git clone --recursive https://github.com/MicroEJ/Tool-CommandLineBuild.git
   ```

5. Create a directory named `buildKit` in the `Tool-CommandLineBuild` directory.
6. Export the MicroEJ build kit from your MicroEJ SDK version to the `buildKit` directory, by following the steps described *here*.
7. Set the environment variable `MICROEJ_BUILD_TOOLS_HOME` to point to the `Tool-CommandLineBuild` directory

Note: At this point, the content of the directory `Tool-CommandLineBuild` should look like the following:

```
Tool-CommandLineBuild
├── buildKit
│   ├── ant
│   │   └── lib
│   │       ├── ant.jar
│   │       ├── ant-launcher.jar
│   │       └── ...
│   └── microej-build-repository.zip (or is2t_repo.zip)
├── easyant
│   └── ...
├── ivy
│   ├── ivysettings-artifactory.xml
│   └── ...
└── ...
```

6.4.6 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.

In the following, we will use 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 Working Offline section.
3. Click on the offline repository link. This will download the Central Repository as a ZIP file.

6.4.7 Setup Artifactory

Install and Start Artifactory

1. Download Artifactory here: https://api.bintray.com/content/jfrog/artifactory/jfrog-artifactory-oss-\protect\T1\textdollarlatest.zip;bt_package=jfrog-artifactory-oss-zip.
2. Unzip downloaded archive, then navigate to `bin` directory (by default `artifactory-oss-[version]/bin`).
3. Run `artifactory.bat` or `artifactory.sh` depending on your operating system. After initialization, the terminal should print the message Artifactory successfully started . In case an error occurs, check that `JAVA_HOME` and `JRE_HOME` environment variables are correct.
4. Go to http://localhost:8081/.
5. Login to Artifactory for the first time using the default `admin` account (Username: `admin`, Password: `password`).
6. On the Welcome wizard, set the administrator password, then click Next ,
7. Configure proxy server (if any) then click Next , or click Skip .
8. On Create Repositories page, select Maven then click on Create .
9. Click on Finish .

Artifactory is up and running.

Configure Artifactory

For demonstration purposes we will allow anonymous users to deploy modules in the repositories.

1. Go to Admin > Security > Security Configuration .
2. In the General Security Settings section, check Allow Anonymous Access . Click Save .
3. Go to Admin > Security > Permissions .
4. Click on Anything entry (do not check the line), then go to Users tab, click on Anonymous and check Deploy/Cache permission. Click Save and finish .

Next steps will involve uploading large files, so we have to augment the file upload maximum size accordingly:

1. Go to Admin > General Configuration .
2. In the General Settings section, change the value of File Upload Max Size (MB) to `1024` then click on Save .

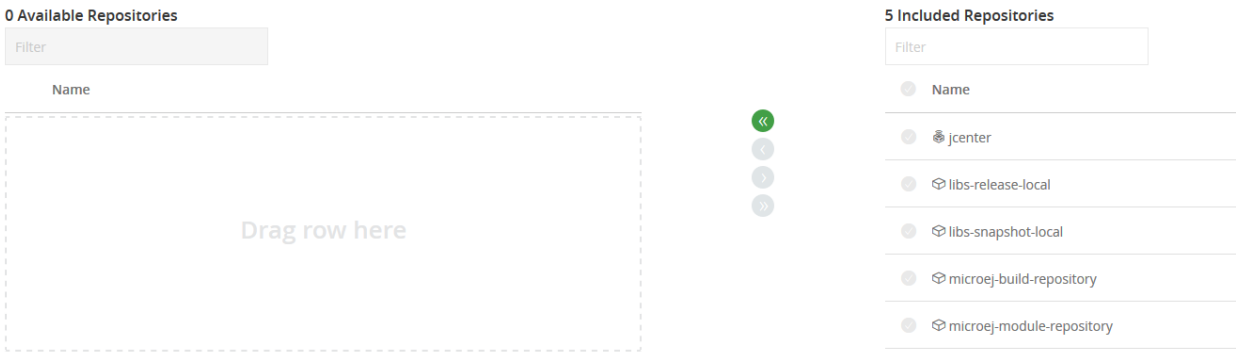
Configure Repositories

First step is to configure to pre-defined repository for the future snapshot modules built.

1. Go to Admin > Repositories > Local .
2. Click on `libs-snapshot-local` repository, then check Handle Releases and uncheck Handle Snapshots . Click Save and finish .

Next step is to create the repositories that will hold the MicroEJ modules.

1. Go to Admin > Repositories > Local .
2. Click on New , and select Maven .
3. Set Repository Key field to `microej-module-repository` , then uncheck Handle Snapshots . Click on Save and finish .
4. Click on New , and select Maven .
5. Set Repository Key field to `microej-build-repository` , then uncheck Handle Snapshots . Click on Save and finish .
6. **Make these two repositories accessible by default:**
    1. Go to Admin > Security > Permissions .
    2. Click on Anything entry (do not check the line)
    3. On the Resources tab, drag repositories `microej-module-repository` and `microej-build-repository` from the Available repositories area to the Included Repositories area.
    4. Click on Save & Finish .



Import MicroEJ Repositories

In this section, we will import MicroEJ repositories into Artifactory repositories to make them available to the build server.

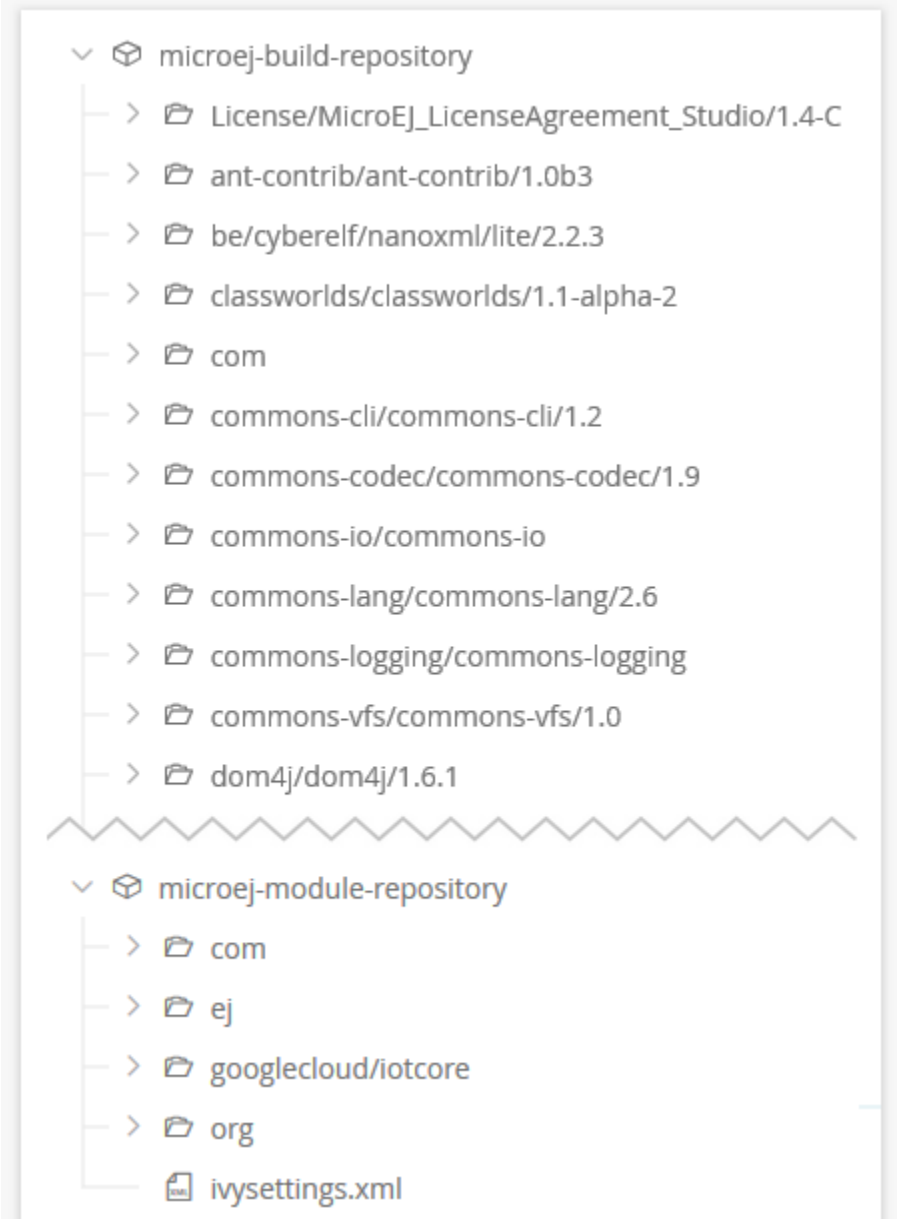
1. Go to Admin > Import & Export > Repositories .
2. Scroll to the Import Repository from Zip section.
3. **Import the MicroEJ Module Repository:**
    1. As Target Local Repository , select `microej-module-repository` in the list.
    2. As Repository Zip File , select MicroEJ module repository zip file (`microej-[MicroEJ version]-[version].zip`) that you downloaded earlier (please refer to section *Get a Module Repository*).

3. Click Upload . At the end of upload, click on Import . Upload and import may take some time.

4. **Import the MicroEJ Build Repository:**

   1. As Target Local Repository , select `microej-build-repository` in the list.
   2. As Repository Zip File , select MicroEJ Build Repository zip file (`microej-build-repository.zip` or `is2t_repo.zip`) that you exported from MicroEJ SDK earlier (please refer to section *Install the Build Tools*).
   3. 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 Artifacts and check that repositories `microej-module-repository` and `microej-build-repository` do contain modules as shown in the figure below.



6.4.8 Setup Jenkins

Install Jenkins

1. Download Jenkins WAR (Web Archive) here: http://mirrors.jenkins.io/war-stable/latest/jenkins.war
2. Open a terminal and type the following command: `java -jar [path/to/downloaded/jenkinswar]/jenkins.war`. After initialization, the terminal will print out Jenkins is fully up and running.
3. Go to http://localhost:8080/.
4. To unlock Jenkins, copy/paste the generated password that has been written in the terminal log. Click on Continue.
5. Select option Install suggested plugins and wait for plugin installation.
6. Fill in the Create First Admin User form. Click Save and continue.
7. Click on Save and finish, then on Start using Jenkins.

Configure Jenkins

First step is to configure JDK and Ant installations:

1. Go to Manage Jenkins > Global Tool Configuration.
2. **Add JDK installation:**
    1. Scroll to JDK section.
    2. Click on Add JDK.
    3. Set Name to `JDK [jdk_version]` (for example `JDK 1.8`).
    4. Uncheck Install automatically.
    5. Set JAVA_HOME to `path/to/jdk[jdk_version]` (for example `C:\Program Files\Java\jdk1.8.0_[version]` on Windows).
3. **Add Ant installation:**
    1. Scroll to Ant section.
    2. Click on Add Ant.
    3. Set Name to `Ant 1.9`.
    4. Uncheck Install automatically.
    5. Set ANT_HOME to `path/to/apache-ant-1.9.[version]`.
4. Click on Save.

Create a Job Template

1. Go to Jenkins dashboard.
2. Click on New item to create a job template.

3. Set item name to `Template - MMM from Git`.
4. Select Freestyle project .
5. Click on Ok .

In General tab:

1. Check This project is parametrized and add String parameter named `easyant.module.dir` with default value to `$WORKSPACE/TO_REPLACE`. This will later point to the module sources.

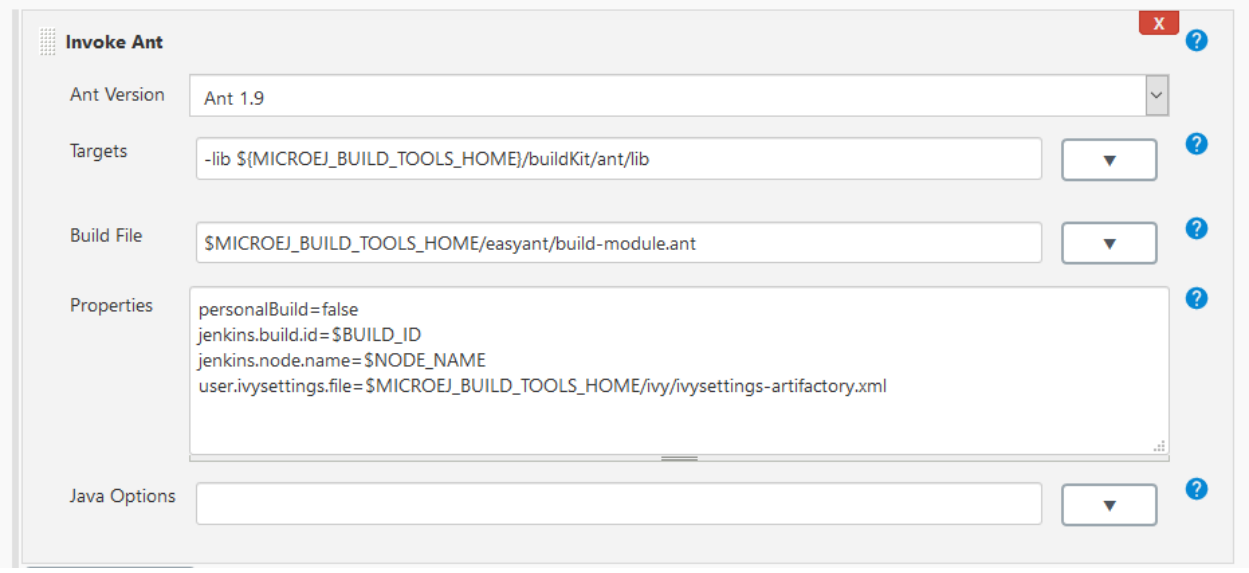
In Source Code Management tab:

1. Select Git source control:
2. Set Repository URL value to `TO_REPLACE`,
3. Set Branch Specifier value to `origin/master`,
4. In Additional Behaviours , click on Add , select Advanced sub-modules behaviors , then check Recursively update submodules .

In Build tab:

1. **Add build step Invoke Ant :**
   - As Ant version , select `Ant 1.9`.
   - Set Targets to value `-lib ${MICROEJ_BUILD_TOOLS_HOME}/buildKit/ant/lib`.
   - In Advanced , set Build file to value `$MICROEJ_BUILD_TOOLS_HOME/easyant/build-module.ant`.
   - In Advanced , expand Properties text field then add the following Ant properties:

```
personalBuild=false
jenkins.build.id=$BUILD_ID
jenkins.node.name=$NODE_NAME
user.ivysettings.file=$MICROEJ_BUILD_TOOLS_HOME/ivy/ivysettings-artifactory.xml
```



Finally, click on Save .

6.4.9 Build a new Module using Jenkins

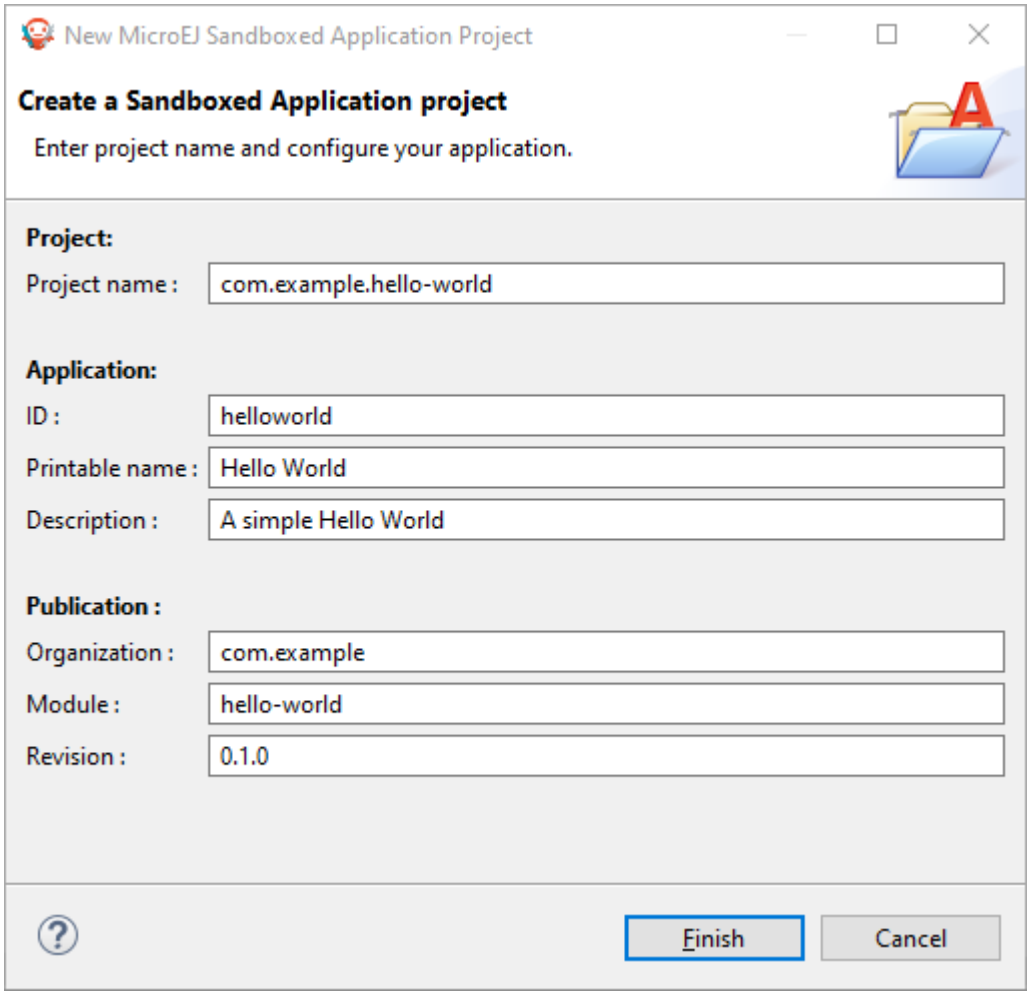
Since your environment is now setup, it is time to build your first module from Jenkins and check it has been published to Artifactory. Let's build an "Hello World" Sandboxed Application project.

Create a new MicroEJ Module

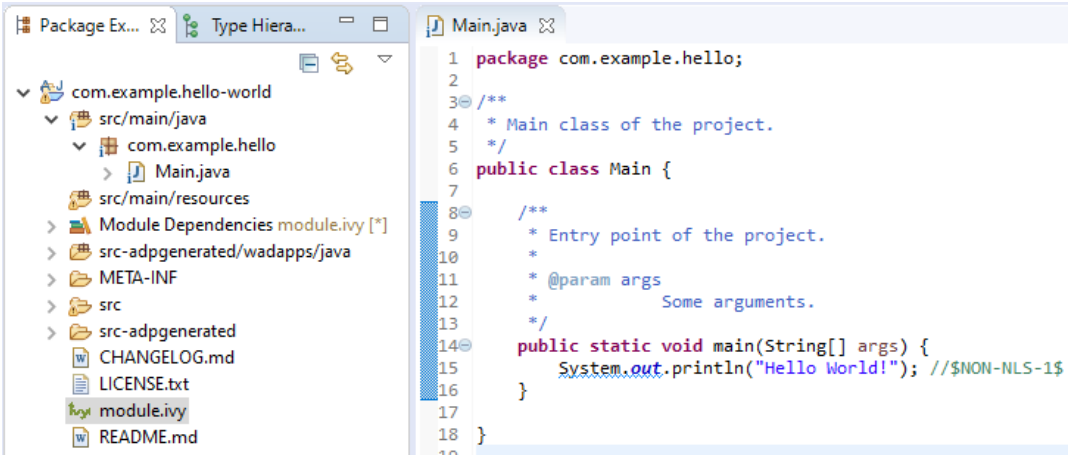
In this example, we will create a very simple module using the Sandbox Application buildtype (`build-application`) that we'll push to a Git repository.

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.

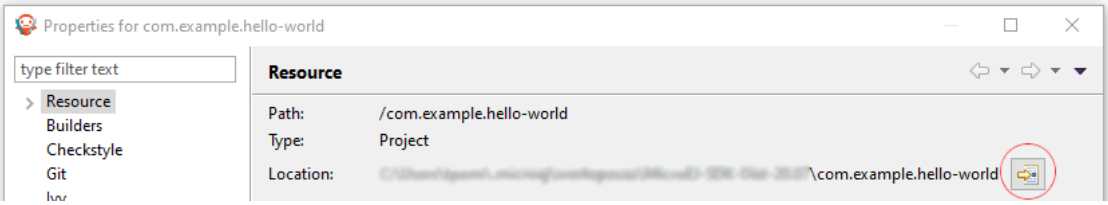
1. Start MicroEJ SDK.
2. Go to File > New > MicroEJ Sandboxed Application Project .
3. Fill in the template fields, set Project name to `com.example.hello-world` .



4. Click Finish . This will create the project files and structure.
5. Right-click on source folder `src/main/java` and select New > Package . Set a name to the package and click Finish .
6. Right-click on the new package and select New > Class . Set a name to the class and check `public static void main(String[] args)` , then click Finish .



7. **Locate the project files**
    1. In the Package Explorer view, right-click on the project then click on Properties .
    2. Select Resource menu.
    3. Click on the arrow button on line Location to show the project in the system explorer.



8. Open a terminal from this directory and type the following commands:

```
git init --bare ~/hello_world.git
git init
git remote add origin ~/hello_world.git
git add com.example.hello-world
git commit -m "Add Hello World application"
git push --set-upstream origin master
```

Note: For more details about MicroEJ Applications development, refer to the *Application Developer Guide*.

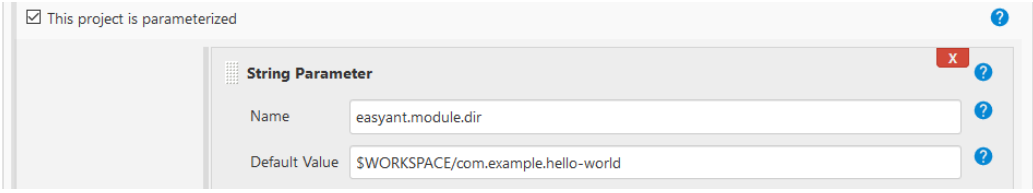
Create a New Jenkins Job

Start by creating a new job, from the job template, for building our application.

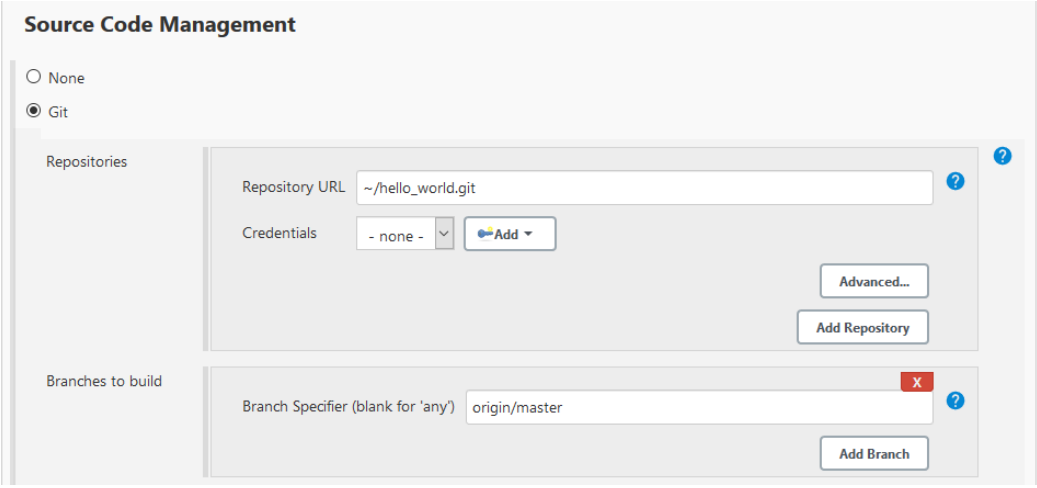
1. Go to Jenkins dashboard.
2. Click on New Item .
3. Set item name to `Hello World` .
4. In Copy from field, type `Template - MMM from Git` (autocomplete enabled).
5. Validate with Ok button.

The job configuration page opens, let's replace all the `TO_REPLACE` placeholders from the job template with correct values:

1. In General tab, set `easyant.module.dir` to value `$WORKSPACE/com.example.hello-world` .



2. In Source Code Management , edit Repository URL to `~/hello_world.git` .



3. Click on Save .

Build the "Hello World" Application

Let's run the job!

In Jenkins' `Hello World` dashboard, click on Build with Parameters , then click on Build .

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:8081/artifactory/list/libs-snapshot-local/com/example/hello-world/.

Congratulations!

At this point of the tutorial:

- Artifactory is hosting your module builds and MicroEJ modules.
- Jenkins automates the build process using *MicroEJ Module Manager*.

The next steps recommended are:

- Adapt Jenkins/Artifactory configuration to your ecosystem and development flow.

6.4.10 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` .
3. Check Retain long standard output/error .
4. Check Do not fail the build on empty test results

Add a Self-Signed Certificate

In case your Artifactory instance uses a self-signed SSL certificate, you might fall into this error when fetching 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]
```

The authority has to be added to the trust store of the JRE/JDK that is running Artifactory. Here is a way to do it:

1. Install Keystore Explorer.
2. Start Keystore Explorer, and open file `[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.
3. Click on Tools , then Import Trusted Certificate .
4. Select your certificate.
5. Save the `cacerts` file.

6.5 Improve the Quality of Java Code

This tutorial 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.

6.5.1 Intended Audience

The audience for this document is engineers who are developing any kind of Java code (application or library).

6.5.2 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.
- 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`).

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 `/*default*/` 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 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:

- Class and member modifiers, when present, must appear in the order recommended by the Java Language Specification: `public protected private abstract default static final transient volatile synchronized native strictfp`.
- 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).

- The parts of a class or interface declaration must appear in the order suggested by the Code Convention for the Java Programming Language:
  - Class (static) fields. First, the public class fields, then the protected, then package level (no access modifier), and then the private.
  - Instance fields. First, the public class fields, then the protected, then package level (no access modifier), and then the private.
  - Constructors
  - Methods

Note: Most of these rules are checked by *Code Analysis with SonarQube™*.

6.5.3 Best Practices

This section describes rules made of best practices and well-known restrictions of the Java Programming Language and more generally Object Oriented paradigm.

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.
- Do not directly return an internal array.
- 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 `//VM_DONE` comment can be written to understand the optimization.
- 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 (`O(1)`) and slightly smaller code memory footprint instead of *lookup_switch* (`O(log N)`).
- 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 a 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 Object.hashCode() method is written that way:

- Choose a prime number.
- Create a result local, whatever the value (usually the prime number).
- For each field, multiply the previous result with the prime plus the hash code of the field and store it as the result.
- Return the result.

Depending on its type, the hash code of a field is:

- Boolean: `(f ? 0 : 1)`.
- 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).

```
private static final int PRIME = 31;

@Override
public int hashCode() {
```

(continues on next page)

(continued from previous page)

```
  int result = PRIME;
  result = PRIME * result + field1;
  result = PRIME * result + (field2 == null ? 0 : field2.hashCode());
  return result;
}
```

6.5.4 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 *Testsuite with JUnit*):

- 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.

6.6 Optimize the Memory Footprint of an Application

This tutorial 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.

6.6.1 Intended Audience

The audience for this document is Java engineers and Firmware integrators who are going to execute a MicroEJ Application on a memory-constrained device.

6.6.2 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.

6.6.3 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 `SOAR.map` with the *Memory Map Analyzer*
3. Analyze `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 Java 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 *Build Output Files* documentation.

How to Analyze a Map File Generated by a Third-Party Linker

A `<firmware>.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 MicroEJ Core Engine sections.

6.6.4 How to Reduce the Image Size of an Application

Generic coding rules can be found in the following tutorial: *Improve the Quality of Java Code*.

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.

Compression

The default NLS implementation relies on text resources that are not compressed, but you can use your own encoding to load them from compressed resources.

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 Logger or `System.out.println()` , use the *trace library* instead. The logging library uses strings, while the trace library only uses 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 Java heap memory and require additional code for boxing and unboxing.
- 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 Java 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.
- 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 MicroEJ 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`. 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* section.

Stripping Class Names from an Application

By default, when a Java class is used, its name is embedded too. A class is used when one of its methods is called, for example. 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, the default behavior is inverted by defining the *Application option* `soar.generate.classnames` to `false`.

For more information on how to set an option, please refer to the *Defining an Option* 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.

6.6.5 How to Reduce the Runtime Size of an Application

You can find generic coding rules in the following tutorial: *Improve the Quality of Java Code*.

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 require a dedicated internal structure and VM stack blocks.
  - Avoid creating threads on the fly for asynchronous execution, use shared thread instances instead (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 MicroJVM 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* documentation.

Java 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 *Java heap* option to fit the needs of the application. You can get the maximum heap usage by calling Runtime.freeMemory() after System.gc() at different moments in the application's lifecycle. The profiling library can be used for this.

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.

VM 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 *Debugging* 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.

6.7 Explore Data Serialization Formats

This tutorial highlights some data serialization formats that are provided on MicroEJ Central Repository and their usage through basic code samples.

6.7.1 Intended Audience

The audience for this document is Application engineers who want to implement data serialization. In addition, this tutorial should be of interest to software architects who are looking for a suitable data format for their use case.

6.7.2 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 Parser Use In MicroEJ SDK

The XML API Module must be added to the *module.ivy* of the MicroEJ Application project in order to allow access to the KXML library.

```
<dependency org="org.kxml2" name="kxml2" rev="2.3.2"/>
```

Example Of Use

An example available at https://github.com/MicroEJ/Example-XML. It presents how to use XML data exchange for your MicroEJ 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 MicroEJ Platform (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
```

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
```

6.7.3 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 Parser Use In MicroEJ SDK

The JSON API Module must be added to the *module.ivy* of the MicroEJ Application project in order to allow access to the JSON library.

```
<dependency org="org.json.me" name="json" rev="1.3.0"/>
```

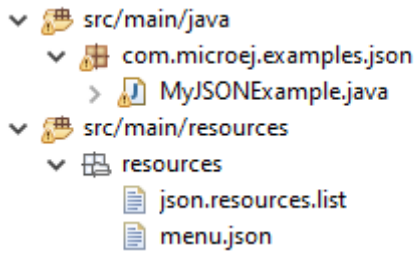
The instantiation and use of the parser is pretty straightforward. First you need to get the JSON content as a `String`, and then create a `org.json.me.JSONObject` instance with the string. If the string content is a valid JSON content, you should have an workable `JSONObject` to browse.

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 *Raw 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
```

(continues on next page)

(continued from previous page)

```
* 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);

                } 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 MicroEJ Simulator should print the following trace:

```
=============== [ Initialization Stage ] ===============
```

(continues on next page)

(continued from previous page)

```
=============== [ Launching Simulator ] ===============
The menuitem content of popup menu is:
[{"value":"New","onclick":"CreateNewDoc()"},{"value":"Open","onclick":"OpenDoc()"},{"value":"Close",
↪"onclick":"CloseDoc()"}]
=============== [ Completed Successfully ] ===============

SUCCESS
```

6.8 Instrument Java Code 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.

6.8.1 Intended Audience

The audience for this document is application developers who are looking for ways to add logging to their MicroEJ applications and libraries.

It should also be of interest to Firmware engineers how are looking for adjusting the log level while keeping low memory footprint and good performances.

6.8.2 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

6.8.3 Overview

In this tutorial, we will describe 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.

Through this tutorial, we will illustrate the usage of each library by instrumenting the following code snippet:

```
public class Main {

   enum ApplicationState {
      INSTALLED, STARTED, STOPPED, UNINSTALLED
   }

   private static ApplicationState currentState;
```

(continues on next page)

(continued from previous page)

```
   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.

6.8.4 Log with the Trace Library

The library ej.api.trace 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. Add the following dependency to the `module.ivy`:

```
<dependency org="ej.api" name="trace" rev="1.1.0"/>
```

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:

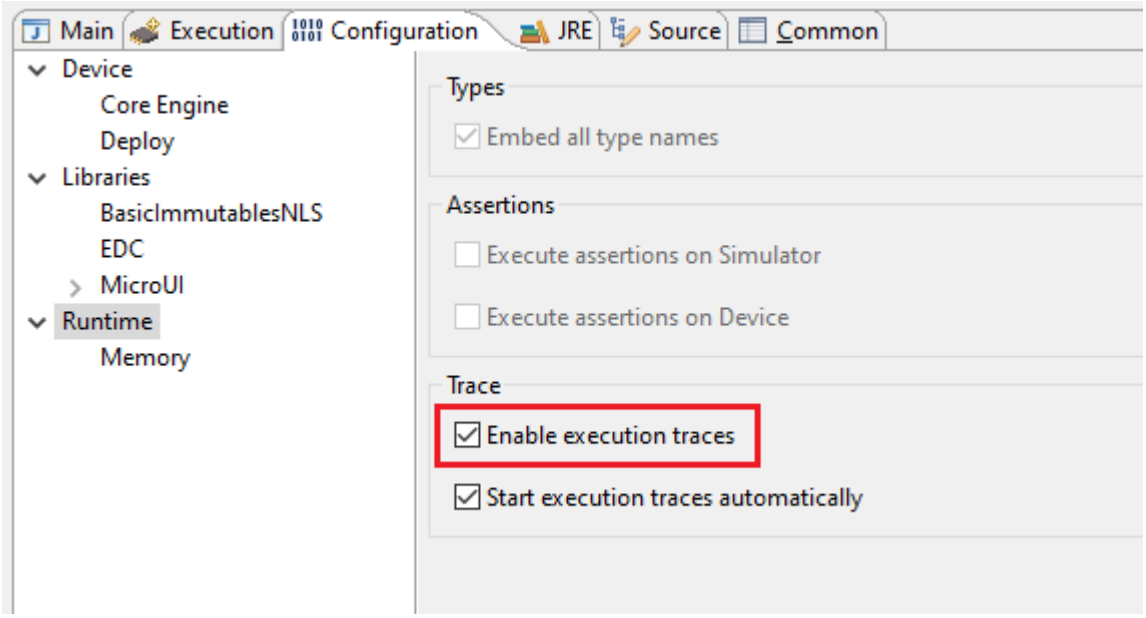
```
public static void main(String[] args) {
   Tracer.startTrace();

   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 MicroEJ Core Engine trace system, set the `core.trace.enabled` *option* to `true`. This can be done from a *launch configuration*: check Runtime > Enable execution traces option.



This produces the following output:

```
[TRACE: Application] Event 0x0()
[TRACE: Application] Event End 0x0()
```

Note: The default Platform implementation of the `Trace` library prints the events to the console. See *Platform Implementation* for other available implementations such as *SystemView* tool.

6.8.5 Log with the Message Library

The library ej.library.runtime.message 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).

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 this dependency line in the `module.ivy`:

```
<dependency org="ej.library.runtime" name="message" rev="2.1.0"/>
```

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;

   BasicMessageLogger.INSTANCE.log(Level.INFO, LOG_CATEGORY, LOG_ID, previousState, currentState);
}
```

This produces the following output:

```
Application:I=2 UNINSTALLED INSTALLED
```

6.8.6 Log with the Logging Library

The library ej.library.eclasspath.logging 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. Add the following dependency to the `module.ivy`:

```
<dependency org="ej.library.eclasspath" name="logging" rev="1.1.0"/>
```

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.

6.8.7 Remove Logging Related Code

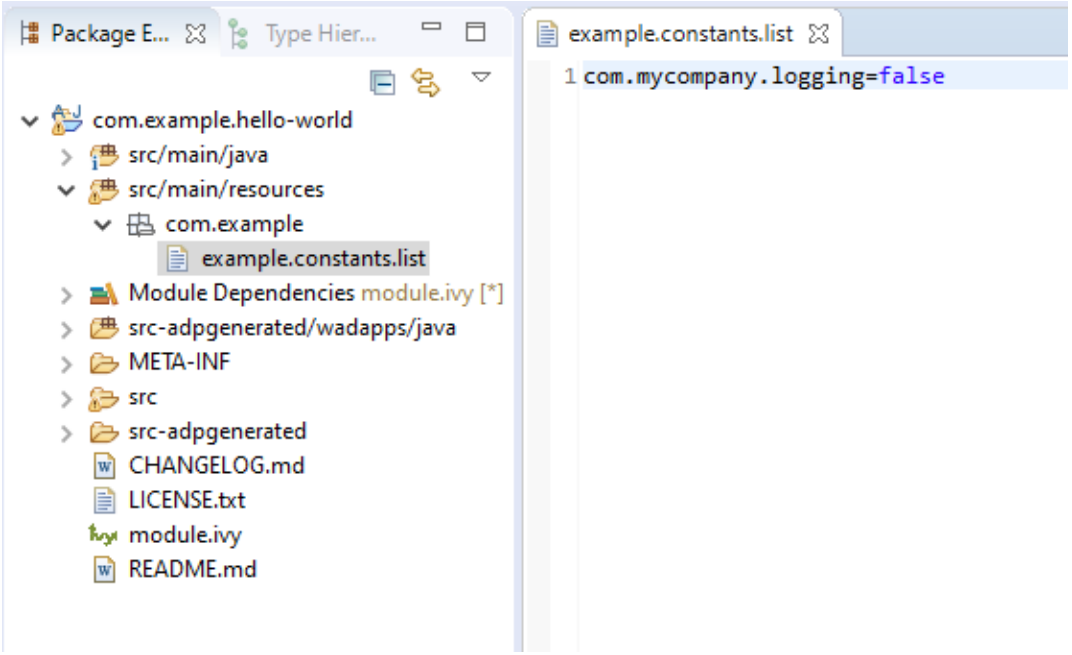
This section describes some techniques to remove logging related code, which saves memory footprint when logging is disabled at runtime. This is typically useful when building two Firmware flavors: one for production and one for debug.

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 (`ej.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.

CHAPTER

SEVEN

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 tutorials.

For more information about MicroEJ, go to : https://www.microej.com/.

INDEX

A

Add-On Library, **2**
Application, **2**
Architecture, **2**

C

Core Engine, **2**

F

Firmware, **2**
Foundation Library, **2**

M

Mock, **2**
Module Manager, **3**

P

Platform, **3**

S

SDK, **3**
Simulator, **3**
Studio, **3**

V

Virtual Device, **3**