MicroEJ Documentation



MicroEJ Corp.

Revision e90f3495

Feb 15, 2024

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

CONTENTS

1 MicroEJ Glossary **2**

2 Overview **4**
2.1 Getting Started . . . 4
2.2 MICROEJ VEE . . . 5
2.3 MICROEJ SDK . . . 6

3 SDK 5 User Guide **9**
3.1 Installation . . . 10
3.1.1 Install Latest SDK Distribution . . . 11
3.1.2 Update SDK Version . . . 16
3.1.3 Install Other SDK Distributions . . . 18
3.1.4 System Requirements . . . 26
3.1.5 Troubleshooting . . . 27
3.2 Licenses . . . 29
3.2.1 SDK EULA . . . 29
3.2.2 License Manager Overview . . . 30
3.2.3 License Check . . . 30
3.2.4 Evaluation Licenses . . . 31
3.2.5 Production Licenses . . . 33
3.3 Standalone Application . . . 42
3.3.1 Platform Import . . . 42
3.3.2 Build and Run an Application . . . 46
3.3.3 MicroEJ Launch . . . 51
3.4 Sandboxed Application . . . 56
3.4.1 Create a First Application . . . 56
3.4.2 Run on the Simulator . . . 58
3.4.3 Run on the Device . . . 62
3.5 Module Repository . . . 65
3.5.1 Create a Repository Project . . . 67
3.5.2 Configure Resolver for Input Modules . . . 67
3.5.3 Configure Consistency Check . . . 67
3.5.4 Advanced Options . . . 67
3.5.5 Include Modules . . . 68
3.5.6 Generate Javadoc . . . 69
3.5.7 Build the Repository . . . 70
3.5.8 Use the Offline Repository . . . 70
3.6 Platform Selection . . . 70
3.7 Module Natures . . . 71
3.7.1 Add-On Library . . . 71

3.7.2 Add-On Processor . . . 72
3.7.3 Foundation Library API . . . 72
3.7.4 Foundation Library Implementation . . . 73
3.7.5 Kernel Application . . . 73
3.7.6 Meta Build . . . 74
3.7.7 Mock . . . 74
3.7.8 Module Repository . . . 74
3.7.9 Runtime Environment . . . 76
3.7.10 Sandboxed Application . . . 76
3.7.11 Standalone Application . . . 76
3.7.12 Studio Rebranding . . . 77
3.7.13 Natures Plugins . . . 77
3.7.14 Global Build Options . . . 83
3.8 Debug an Application . . . 83
3.8.1 Debug on Simulator . . . 83
3.8.2 Debug on Device . . . 84
3.8.3 Get Library Sources . . . 85
3.9 Development Tools . . . 91
3.9.1 Test Suite with JUnit . . . 92
3.9.2 Stack Trace Reader . . . 98
3.9.3 Code Coverage Analyzer . . . 111
3.9.4 Heap Dumper & Heap Analyzer . . . 114
3.9.5 Serial to Socket Transmitter . . . 125
3.9.6 Memory Map Analyzer . . . 127
3.9.7 Null Analysis . . . 130
3.10 IDE . . . 141
3.10.1 Startup . . . 141
3.10.2 Resolve Dependencies in Workspace . . . 142
3.10.3 Resolve Foundation Libraries in Workspace . . . 143
3.10.4 Resolve Front Panel in Workspace . . . 145
3.11 SDK Version . . . 145
3.12 MicroEJ Module Manager . . . 147
3.12.1 Introduction . . . 147
3.12.2 Specification . . . 148
3.12.3 Module Project Skeleton . . . 148
3.12.4 Module Description File . . . 148
3.12.5 SDK Configuration . . . 151
3.12.6 Module Build . . . 155
3.12.7 Build Kit . . . 156
3.12.8 Command Line Interface . . . 157
3.12.9 Build System Options . . . 161
3.12.10 Meta Build . . . 161
3.12.11 Troubleshooting . . . 162
3.12.12 Former SDK Versions (lower than 5.2.0) . . . 166
3.12.13 Former SDK Versions (from 5.2.0 to 5.3.x) . . . 167
3.13 Release Notes . . . 168
3.14 SDK Distribution Changelog . . . 168
3.14.1 [24.01] - 2024-01-31 . . . 168
3.14.2 [23.07] - 2023-07-03 . . . 168
3.14.3 [23.02] - 2022-02-28 . . . 168
3.14.4 [22.06] - 2022-06-29 . . . 169
3.14.5 [21.11] - 2021-11-15 . . . 169
3.14.6 [21.03] - 2021-03-25 . . . 169
3.14.7 [20.12] - 2020-12-11 . . . 170

3.14.8 [20.10] - 2020-10-30 . . . 170
3.14.9 [20.07] - 2020-07-28 . . . 171
3.14.10 [19.05] - 2019-05-17 . . . 171
3.14.11 [19.02] - 2019-02-22 . . . 171
3.15 SDK Changelog . . . 171
3.15.1 [5.8.2] - 2024-01-31 . . . 171
3.15.2 [5.8.1] - 2023-09-19 . . . 172
3.15.3 [5.8.0] - 2023-07-03 . . . 173
3.15.4 [5.7.0] - 2023-02-27 . . . 173
3.15.5 [5.6.2] - 2022-08-31 . . . 175
3.15.6 [5.6.1] - 2022-07-08 . . . 175
3.15.7 [5.6.0] - 2022-06-29 . . . 175
3.15.8 [5.5.3] - 2022-05-03 . . . 177
3.15.9 [5.5.2] - 2021-12-22 . . . 177
3.15.10 [5.5.1] - 2021-12-02 . . . 177
3.15.11 [5.5.0] - 2021-11-15 . . . 177
3.15.12 [5.4.1] - 2021-04-16 . . . 179
3.15.13 [5.4.0] - 2021-03-25 . . . 180
3.15.14 [5.3.1] - 2020-12-11 . . . 182
3.15.15 [5.3.0] - 2020-10-30 . . . 182
3.15.16 [5.2.0] - 2020-07-28 . . . 184
3.15.17 [5.1.2] - 2020-03-09 . . . 186
3.15.18 [5.1.1] - 2019-09-26 . . . 186
3.15.19 [5.1.0] - 2019-05-17 . . . 187
3.15.20 [5.0.1] - 2019-02-14 . . . 189
3.16 Build Types per SDK . . . 190
3.17 Migration Notes . . . 196
3.17.1 From 5.2.x to 5.3.x or more . . . 196
3.17.2 From 5.1.x to 5.2.x . . . 196
3.17.3 From 4.1.x to 5.x . . . 197

4 SDK 6 User Guide 199
4.1 Getting Started . . . 200
4.1.1 NXP . . . 200
4.2 Installation . . . 223
4.2.1 System Requirements . . . 223
4.2.2 Check your JDK version . . . 224
4.2.3 Install Gradle . . . 224
4.2.4 Configure Repositories . . . 225
4.2.5 Install the IDE . . . 225
4.2.6 Install the IDE Plugin . . . 225
4.3 Licenses . . . 230
4.3.1 SDK EULA . . . 230
4.3.2 License Manager Overview . . . 230
4.3.3 License Check . . . 230
4.3.4 SDK EULA Acceptation . . . 231
4.3.5 Evaluation Licenses . . . 231
4.3.6 Production Licenses . . . 232
4.4 Scope and Limitations . . . 239
4.5 Create a Project . . . 240
4.5.1 Configure a Project . . . 249
4.5.2 Create a subproject in an existing project . . . 252
4.5.3 Gradle Wrapper . . . 257
4.6 Import a Project . . . 258

4.7 Select a VEE Port . . . 261
4.7.1 Using a Module Dependency . . . 261
4.7.2 Using a Local VEE Port Directory . . . 261
4.7.3 Using a Local VEE Port Archive . . . 262
4.8 Run on Simulator . . . 262
4.8.1 Verbose Mode . . . 264
4.8.2 Debug on Simulator . . . 264
4.8.3 Generate Code Coverage . . . 265
4.8.4 Generate Heap Dump . . . 265
4.9 Build an Executable . . . 265
4.10 Run on Device . . . 267
4.11 Select a Kernel . . . 268
4.11.1 Using a Module Dependency . . . 268
4.11.2 Using a Local Kernel . . . 269
4.12 Build a Feature file . . . 269
4.13 Build a Virtual Device . . . 271
4.13.1 Add a Pre-Installed Application in a Virtual Device . . . 272
4.13.2 Add a Kernel API in a Virtual Device . . . 273
4.14 Add a Dependency . . . 273
4.14.1 Configurations . . . 273
4.14.2 Version . . . 274
4.14.3 Dependencies Repositories . . . 274
4.15 Test a Project . . . 274
4.15.1 JUnit Compliance . . . 275
4.15.2 Gradle Integration . . . 275
4.15.3 Test on Simulator . . . 275
4.15.4 Test on Device . . . 279
4.15.5 Test on J2SE VM . . . 280
4.15.6 Test Suite Reports . . . 280
4.15.7 Mixing tests . . . 281
4.15.8 Configure the Testsuite Engine . . . 284
4.15.9 Inject Application Options . . . 285
4.16 Publish a Project . . . 286
4.17 Development Tools . . . 287
4.17.1 Stack Trace Reader . . . 288
4.17.2 Code Coverage Analyzer . . . 292
4.17.3 Memory Map Analyzer . . . 294
4.17.4 Heap Dumper & Heap Analyzer . . . 298
4.17.5 Font Designer . . . 309
4.17.6 Local Deployment Socket . . . 317
4.17.7 Null Analysis . . . 319
4.18 Manage Versioning . . . 326
4.19 Manage Resolution Conflicts . . . 326
4.20 Migrate an MMM Project . . . 328
4.20.1 Project structure . . . 328
4.20.2 Build Descriptor File . . . 329
4.20.3 Build Scripts . . . 333
4.21 Module Natures . . . 333
4.21.1 Add-On Library . . . 334
4.21.2 Application . . . 334
4.21.3 J2SE Library . . . 335
4.21.4 Tasks . . . 336
4.21.5 Global Properties . . . 342
4.22 Troubleshooting . . . 342

4.22.1 Java Compiler Version Issue . . . 342
4.22.2 Unresolved Dependency . . . 344
4.22.3 Invalid SSL Certificate . . . 345
4.22.4 Failing Resolution in adp Task . . . 346
4.22.5 Missing Version for Publication . . . 346
4.22.6 Fail to load a VEE Port as dependency . . . 347
4.22.7 Slow Build because of File System Watching . . . 347
4.22.8 Missing Tasks in the Gradle view of Android Studio . . . 348
4.23 Tutorials . . . 349
4.23.1 Branding an Eclipse IDE . . . 349
4.23.2 Creating and Using an Offline Repository . . . 355
4.24 How-to Guides . . . 358
4.24.1 How To Define a Specific Java Home for Gradle . . . 358
4.24.2 How To Pass a Property to Project Build Script . . . 358
4.24.3 How To Skip a Gradle Task . . . 359
4.24.4 How To Automatically reload a Gradle project . . . 360
4.24.5 How To Add a Repository . . . 362
4.24.6 How To Resolve Dependencies in the IDE . . . 363
4.24.7 How to Install MicroEJ Plugin Snapshot Version on Android Studio or IntelliJ IDEA . . . 364
4.24.8 How To Build a Project . . . 365
4.24.9 How To Build an Executable With Multiple VEE Ports . . . 366
4.24.10 How To Create a Custom Configuration in the IDE . . . 368
4.25 Appendices . . . 371
4.25.1 Virtual Device . . . 371
4.25.2 Dependencies Configurations . . . 373
4.26 Changelog . . . 374
4.26.1 [0.15.0] - 2024-01-26 . . . 374
4.26.2 [0.14.0] - 2024-01-03 . . . 375
4.26.3 [0.13.0] - 2023-11-10 . . . 376
4.26.4 [0.12.1] - 2023-10-16 . . . 376
4.26.5 [0.12.0] - 2023-10-13 . . . 376
4.26.6 [0.11.1] - 2023-09-22 . . . 377
4.26.7 [0.11.0] - 2023-09-22 . . . 377
4.26.8 [0.10.0] - 2023-09-13 . . . 377
4.26.9 [0.9.0] - 2023-09-01 . . . 378
4.26.10 [0.8.0] - 2023-07-13 . . . 378
4.26.11 [0.7.0] - 2023-06-26 . . . 378
4.26.12 [0.6.0] - 2023-05-30 . . . 379
4.26.13 [0.5.0] - 2023-03-24 . . . 379
4.26.14 [0.4.0] - 2023-01-27 . . . 380
4.26.15 [0.3.0] - 2022-12-09 . . . 380
4.26.16 [0.2.0] - 2022-05-17 . . . 381
4.26.17 [0.1.0] - 2022-05-03 . . . 381
4.27 Migration Notes . . . 382
4.27.1 From 0.14.0 to 0.15.0 . . . 382
4.27.2 From 0.11.1 to 0.12.0 . . . 383
4.27.3 From 0.10.0 to 0.11.0 . . . 383
4.27.4 From 0.8.0 to 0.9.0 . . . 384

5 Application Developer Guide **385**
5.1 Introduction . . . 385
5.2 MicroEJ Runtime . . . 385
5.2.1 Language . . . 385
5.2.2 Core Libraries . . . 386

5.2.3 Scheduler . . . 388
5.2.4 Garbage Collector . . . 388
5.2.5 Limitations . . . 388
5.3 SOAR . . . 389
5.3.1 Java Symbols Encoding . . . 389
5.3.2 Class Initialization Code . . . 389
5.3.3 Method Devirtualization . . . 390
5.3.4 Method Inlining . . . 390
5.3.5 Binary Code Verifier . . . 391
5.4 SOAR Output Files . . . 392
5.4.1 Launch Output Folder . . . 392
5.4.2 Published Module Files . . . 393
5.4.3 The SOAR Map File . . . 394
5.4.4 The SOAR Information File . . . 394
5.5 Virtual Device . . . 396
5.6 MicroEJ Classpath . . . 397
5.6.1 Application Classpath . . . 398
5.6.2 Classpath Load Model . . . 399
5.6.3 Classpath Elements . . . 399
5.7 Application Resources . . . 404
5.8 Standalone Application . . . 405
5.8.1 Introduction . . . 405
5.8.2 Standalone Application Options . . . 405
5.8.3 Defining an Option with SDK 6 . . . 406
5.8.4 Defining an Option with SDK 5 or lower . . . 406
5.8.5 Category: Runtime . . . 408
5.8.6 Category: Simulator . . . 412
5.8.7 Category: Libraries . . . 423
5.8.8 Category: Device . . . 427
5.8.9 Category: Feature . . . 435
5.9 Sandboxed Application . . . 435
5.9.1 Fundamental Concepts . . . 435
5.9.2 Shared Interfaces . . . 436
5.10 Character Encoding . . . 442
5.10.1 Default Encoding . . . 442
5.10.2 UTF-8 Encoding . . . 442
5.10.3 Custom Encoding . . . 442
5.10.4 Console Output . . . 442
5.11 Limitations . . . 443
5.12 GitHub Repositories . . . 444
5.12.1 Repository Import . . . 444
5.12.2 MicroEJ GitHub Badges . . . 449
5.13 Module Repositories . . . 449
5.13.1 Central Repository . . . 449
5.13.2 Developer Repository . . . 450
5.13.3 Content Organization . . . 451
5.14 Libraries . . . 451
5.14.1 Graphical User Interface . . . 452
5.14.2 Native Language Support . . . 584
5.14.3 Networking . . . 591
5.14.4 Date and Time . . . 608
5.14.5 Event Queue . . . 619
5.14.6 JavaScript . . . 627
5.15 Development Tools . . . 650

5.15.1 Event Tracing . . . 650
5.15.2 VEE Debugger Proxy . . . 653
5.15.3 Dependency Discoverer . . . 660
5.15.4 MicroEJ Linker . . . 661
5.15.5 MicroEJ Test Suite Engine . . . 674
5.15.6 Heap Usage Monitoring . . . 681

6 VEE Porting Guide 684
6.1 Introduction . . . 684
6.1.1 Scope . . . 684
6.1.2 Intended Audience . . . 684
6.2 MicroEJ Platform . . . 684
6.2.1 Introduction . . . 684
6.2.2 Build Process . . . 685
6.2.3 Concepts . . . 686
6.3 MicroEJ Architecture . . . 691
6.3.1 Naming Convention . . . 692
6.3.2 Architectures Changelog . . . 693
6.3.3 Release Notes . . . 725
6.4 MicroEJ Packs . . . 726
6.4.1 Overview . . . 726
6.4.2 Naming Convention . . . 726
6.5 Platform Creation . . . 727
6.5.1 Architecture Selection . . . 727
6.5.2 Platform Configuration . . . 728
6.5.3 Pack Import . . . 729
6.5.4 Platform Build . . . 730
6.5.5 Platform Module Configuration . . . 732
6.5.6 Platform Customization . . . 734
6.5.7 Platform Publication . . . 734
6.5.8 BSP Connection . . . 735
6.5.9 Platform API Documentation . . . 740
6.5.10 Link-Time Option . . . 741
6.6 VEE Port Qualification . . . 742
6.6.1 Introduction . . . 742
6.6.2 VEE Port Qualification Tools Overview . . . 744
6.6.3 VEE Port Test Suite . . . 745
6.6.4 Create a VEE Port Test Suite . . . 746
6.6.5 Test Suite Versioning . . . 750
6.7 Core Engine . . . 751
6.7.1 Block Diagram . . . 752
6.7.2 Link Flow . . . 752
6.7.3 Architecture . . . 753
6.7.4 Capabilities . . . 754
6.7.5 Implementation . . . 755
6.7.6 Generic Output . . . 763
6.7.7 Link . . . 763
6.7.8 Dependencies . . . 764
6.7.9 Installation . . . 764
6.7.10 Abstraction Layer . . . 764
6.7.11 Memory Considerations . . . 764
6.7.12 Use . . . 765
6.8 Advanced Event Tracing . . . 765
6.8.1 Principle . . . 765

6.8.2 Platforms using GNU LD linker . . . 765
6.8.3 Platforms using IAR ILINK linker . . . 766
6.9 Multi-Sandbox . . . 766
6.9.1 Principle . . . 766
6.9.2 Functional Description . . . 766
6.9.3 Memory Considerations . . . 767
6.9.4 Dependencies . . . 767
6.9.5 Installation . . . 767
6.9.6 Use . . . 767
6.9.7 Feature Installation . . . 767
6.9.8 RAM Control . . . 776
6.10 Tiny-Sandbox . . . 776
6.10.1 Principle . . . 776
6.10.2 Installation . . . 776
6.10.3 Limitations . . . 777
6.11 Native Interface Mechanisms . . . 777
6.11.1 Simple Native Interface (SNI) . . . 777
6.11.2 Shielded Plug (SP) . . . 781
6.11.3 MicroEJ Java H . . . 784
6.12 External Resources Loader . . . 785
6.12.1 Functional Description . . . 785
6.12.2 Implementations . . . 785
6.12.3 External Resources Folder . . . 786
6.12.4 Dependencies . . . 786
6.12.5 Installation . . . 787
6.12.6 Use . . . 787
6.13 Serial Communications . . . 787
6.13.1 ECOM . . . 787
6.13.2 ECOM Comm . . . 789
6.14 Graphical User Interface . . . 797
6.14.1 Principle . . . 797
6.14.2 UI Port . . . 799
6.14.3 MicroUI . . . 815
6.14.4 Static Initialization . . . 818
6.14.5 Abstraction Layer API . . . 821
6.14.6 LED . . . 823
6.14.7 Input . . . 826
6.14.8 Display . . . 839
6.14.9 Drawings . . . 875
6.14.10 Images . . . 893
6.14.11 Fonts . . . 945
6.14.12 C Modules . . . 954
6.14.13 Simulation . . . 959
6.14.14 Release Notes . . . 966
6.14.15 Changelog . . . 971
6.14.16 Migration Guide . . . 1005
6.15 Vector Graphics . . . 1028
6.15.1 Principle . . . 1028
6.15.2 MicroVG . . . 1028
6.15.3 Abstraction Layer API . . . 1029
6.15.4 Matrix . . . 1031
6.15.5 Path . . . 1032
6.15.6 Gradient . . . 1034
6.15.7 Image . . . 1036

6.15.8 Font . . . 1041
6.15.9 C Modules . . . 1044
6.15.10 Simulation . . . 1048
6.15.11 Release Notes . . . 1049
6.15.12 Changelog . . . 1050
6.15.13 Migration Guide . . . 1053
6.16 Networking . . . 1054
6.16.1 Principle . . . 1054
6.16.2 Network Core Engine . . . 1054
6.16.3 SSL . . . 1056
6.16.4 Network Interfaces Management . . . 1057
6.16.5 Wi-Fi . . . 1058
6.17 Bluetooth . . . 1058
6.17.1 Principle . . . 1058
6.17.2 Functional Description . . . 1059
6.17.3 Overview . . . 1059
6.17.4 Dependencies . . . 1059
6.17.5 Installation . . . 1059
6.17.6 Use . . . 1059
6.18 Event Queue . . . 1059
6.18.1 Principle . . . 1059
6.18.2 Dependencies . . . 1060
6.18.3 Installation . . . 1060
6.19 File System . . . 1060
6.19.1 Principle . . . 1060
6.19.2 Functional Description . . . 1060
6.19.3 Dependencies . . . 1060
6.19.4 Installation . . . 1060
6.19.5 Use . . . 1062
6.20 Hardware Abstraction Layer . . . 1062
6.20.1 Principle . . . 1062
6.20.2 Functional Description . . . 1063
6.20.3 Identifier . . . 1063
6.20.4 Configuration . . . 1064
6.20.5 Dependencies . . . 1064
6.20.6 Installation . . . 1064
6.20.7 Use . . . 1064
6.21 Device Information . . . 1065
6.21.1 Principle . . . 1065
6.21.2 Dependencies . . . 1065
6.21.3 Installation . . . 1065
6.21.4 Use . . . 1065
6.22 Security . . . 1065
6.22.1 Principle . . . 1065
6.22.2 Dependencies . . . 1066
6.22.3 Installation . . . 1066
6.22.4 Use . . . 1066
6.23 Watchdog Timer . . . 1066
6.23.1 Overview . . . 1066
6.23.2 Principle . . . 1067
6.23.3 Mock Implementation . . . 1069
6.23.4 Dependencies . . . 1069
6.23.5 Installation . . . 1070
6.23.6 Use in an Application . . . 1070

6.23.7 Code example in Java . . . 1070
6.23.8 Use in C inside the BSP . . . 1071
6.23.9 Code example in C . . . 1072
6.24 SystemView . . . 1073
6.24.1 Principle . . . 1073
6.24.2 References . . . 1074
6.24.3 Installation . . . 1074
6.24.4 MicroEJ Core Engine OS Task . . . 1078
6.24.5 OS Tasks and Java Threads Names . . . 1078
6.24.6 OS Tasks and Java Threads Priorities . . . 1079
6.24.7 Use . . . 1080
6.24.8 Troubleshooting . . . 1080
6.24.9 RTT block found by SystemView but no traces displayed . . . 1082
6.24.10 Bus hardfault when running SystemView without Java Virtual Machine (JVM) . . . 1082
6.24.11 SystemView for STM32 ST-Link Probe . . . 1082
6.25 Simulation . . . 1083
6.25.1 Principle . . . 1083
6.25.2 Functional Description . . . 1083
6.25.3 Dependencies . . . 1085
6.25.4 Installation . . . 1085
6.25.5 Use . . . 1085
6.25.6 Mock . . . 1085
6.25.7 Shielded Plug Mock . . . 1091
6.25.8 Front Panel Mock . . . 1092
6.25.9 Bluetooth LE Mock . . . 1101
6.26 Appendices . . . 1105
6.26.1 Low Level API . . . 1105
6.26.2 MicroEJ Foundation Libraries . . . 1122
6.26.3 Tools Options and Error Codes . . . 1131
6.26.4 Architectures MCU / Compiler . . . 1140
6.26.5 Former Platform Migration . . . 1147
6.26.6 Architecture 8.0.0 Migration . . . 1156
6.26.7 Architecture 7.x Migration . . . 1158

7 Kernel Developer Guide 1165
7.1 Overview . . . 1165
7.1.1 Introduction . . . 1165
7.1.2 Terms and Definitions . . . 1165
7.1.3 Overall Architecture . . . 1166
7.1.4 Input and Output Artifacts . . . 1166
7.1.5 Kernel Build Flow . . . 1166
7.1.6 Kernel Implementation Libraries . . . 1167
7.2 Kernel & Features Specification . . . 1168
7.3 Getting Started . . . 1168
7.4 Kernel Creation . . . 1168
7.4.1 Create a new Project . . . 1168
7.4.2 Configure a VEE Port . . . 1169
7.4.3 Build the Executable and Virtual Device . . . 1169
7.4.4 Expose APIs . . . 1171
7.4.5 Implement a Security Policy . . . 1171
7.4.6 Add Pre-installed Applications . . . 1172
7.4.7 Build the Executable in the Workspace . . . 1172
7.4.8 Kernel Application Configuration . . . 1173
7.5 Kernel APIs . . . 1174

7.5.1 Kernel API Definition . . . 1174
7.5.2 Writing Kernel APIs . . . 1175
7.6 Runtime Environment . . . 1177
7.6.1 Principle . . . 1177
7.6.2 Create a new Runtime Environment Module . . . 1178
7.6.3 Use a Runtime Environment in an Application . . . 1181
7.6.4 Extend a Runtime Environment . . . 1181
7.7 Kernel UID . . . 1183
7.8 Sandboxed Application Lifecycle . . . 1183
7.9 Kernel and Features Communication . . . 1184
7.9.1 Shared Services . . . 1185
7.9.2 Communication between Features . . . 1185
7.9.3 Communication between Kernel and Feature . . . 1186
7.9.4 Implement a Registry . . . 1187
7.9.5 Kernel Types Converter . . . 1189
7.10 Multi-Sandbox Enabled Libraries . . . 1189
7.10.1 Manage Internal Global State . . . 1190
7.10.2 Implement a Security Manager Permission Check . . . 1192
7.10.3 Known Foundation Libraries Behavior . . . 1192
7.11 Setup a KF Test Suite . . . 1194
7.11.1 Enable the Test Suite . . . 1195
7.11.2 Add a KF Test . . . 1195
7.11.3 KF Test Suite Options . . . 1197
7.12 Kernel Linking . . . 1197
7.12.1 Link Flow . . . 1197
7.12.2 Kernel Metadata Generation . . . 1198
7.12.3 Feature Portability Control . . . 1199
7.13 Application Linking . . . 1200
7.13.1 SOAR Build Phases . . . 1200
7.13.2 Feature Build Off Board . . . 1202
7.13.3 Feature Build On Device . . . 1202
7.13.4 FSO Compatibility . . . 1205
7.13.5 Feature Portability . . . 1205

8 VEE Wear User Guide 1207
8.1 Android Compatibility Kit . . . 1208
8.1.1 Overview . . . 1209
8.1.2 Installation . . . 1211
8.1.3 Project Setup . . . 1211
8.1.4 VEE Port . . . 1217
8.2 iOS Compatibility Kit . . . 1220
8.2.1 Software Architecture . . . 1220
8.2.2 Workflow . . . 1221
8.2.3 Evaluation . . . 1222
8.3 Offloading . . . 1222
8.3.1 Solution . . . 1222
8.3.2 Evaluation . . . 1223

9 Tutorials 1224
9.1 Understand How to Build a Firmware and its Dependencies . . . 1224
9.1.1 The Components . . . 1224
9.1.2 How to Build . . . 1227
9.2 Create a MicroEJ Platform for a Custom Device . . . 1229
9.2.1 Introduction . . . 1229

9.2.2 A MicroEJ Platform Project is already available for the same MCU/RTOS/C Compiler 1230
9.2.3 A MicroEJ Platform Project is not available for the same MCU/RTOS/C Compiler 1231
9.2.4 Platform Validation 1232
9.2.5 Further Assistance Needed 1232
9.3 Create a MicroEJ Firmware From Scratch 1232
9.3.1 Intended Audience 1232
9.3.2 Introduction 1232
9.3.3 Prerequisites 1233
9.3.4 Overview 1233
9.3.5 Setup the Development Environment 1234
9.3.6 Get Running BSP 1234
9.3.7 FreeRTOS Hello World 1236
9.3.8 Create a MicroEJ Platform 1238
9.3.9 Create MicroEJ Application HelloWorld 1243
9.3.10 Configure BSP Connection in MicroEJ Application 1246
9.3.11 MicroEJ and FreeRTOS Integration 1248
9.4 Add IAR to MicroEJ SDK Docker Image 1261
9.4.1 Prerequisites 1261
9.4.2 Create the Dockerfile 1261
9.5 Create MicroEJ Platform Build and Run Scripts 1262
9.5.1 Intended Audience 1262
9.5.2 Prerequisites 1262
9.5.3 Introduction 1263
9.5.4 Overview 1263
9.5.5 Create Build and Run Scripts 1263
9.5.6 Use Build Script in MicroEJ SDK 1267
9.5.7 Going Further 1271
9.6 Setup an Automated Build using Jenkins and Artifactory 1271
9.6.1 Intended Audience 1271
9.6.2 Introduction 1271
9.6.3 Prerequisites 1272
9.6.4 Overview 1272
9.6.5 Prepare your Docker environment 1273
9.6.6 Get a Module Repository 1274
9.6.7 Setup Artifactory 1275
9.6.8 Setup Gitea 1276
9.6.9 Configure Gitea 1277
9.6.10 Setup Jenkins 1277
9.6.11 Build a new Module using Jenkins 1277
9.6.12 Appendix 1283
9.7 Improve the Quality of Java Code 1284
9.7.1 Intended Audience 1284
9.7.2 Readable Code 1284
9.7.3 Best Practices 1288
9.7.4 Related Tools 1291
9.8 Optimize the Memory Footprint of an Application 1291
9.8.1 Intended Audience 1292
9.8.2 Introduction 1292
9.8.3 How to Analyze the Footprint of an Application 1292
9.8.4 How to Reduce the Image Size of an Application 1293
9.8.5 How to Reduce the Runtime Size of an Application 1299
9.9 Explore Data Serialization Formats 1301
9.9.1 Intended Audience 1301
9.9.2 XML 1301

9.9.3 JSON . . . 1303
9.9.4 CBOR . . . 1306
9.10 Instrument Java Code for Logging . . . 1307
9.10.1 Intended Audience . . . 1307
9.10.2 Introduction . . . 1307
9.10.3 Overview . . . 1307
9.10.4 Log with the Trace Library . . . 1308
9.10.5 Log with the Message Library . . . 1309
9.10.6 Log with the Logging Library . . . 1310
9.10.7 Remove Logging Related Code . . . 1311
9.11 Run a Test Suite on a Device . . . 1313
9.11.1 Intended Audience and Scope . . . 1313
9.11.2 Prerequisites . . . 1313
9.11.3 Introduction . . . 1314
9.11.4 Import the Test Suite . . . 1314
9.11.5 Configure the Test Suite . . . 1315
9.11.6 Run the Test Suite . . . 1317
9.11.7 Configure the Tests to Run . . . 1317
9.11.8 Examine the Test Suite Report . . . 1318
9.12 Implement a Blocking Java Native Method with SNI . . . 1318
9.12.1 Intended Audience . . . 1318
9.12.2 Prerequisites . . . 1318
9.12.3 Overview . . . 1319
9.12.4 Requirements . . . 1319
9.12.5 Example Code . . . 1319
9.12.6 Implement a Non-Blocking Method . . . 1321
9.13 Get Started With GUI . . . 1323
9.13.1 Setup your Environment . . . 1323
9.13.2 Starting MicroUI . . . 1325
9.13.3 Basic Drawing on Screen . . . 1329
9.13.4 Animation . . . 1333
9.13.5 Creating Widgets . . . 1335
9.13.6 Using Layouts . . . 1337
9.13.7 Style . . . 1340
9.13.8 Images . . . 1345
9.13.9 Advanced Styling . . . 1347
9.13.10 Event Handling . . . 1350
9.13.11 Fonts . . . 1351
9.13.12 Scroll List . . . 1358
9.13.13 Creating a Contact List using Scroll List . . . 1360
9.13.14 Internationalization . . . 1362
9.14 Discover Embedded Debugging Techniques . . . 1366
9.14.1 Intended Audience . . . 1366
9.14.2 Debugging Tools . . . 1366
9.14.3 Use Case 1: Debugging a GUI Application Freeze . . . 1375
9.14.4 Use Case 2: Debugging a HardFault . . . 1380
9.15 How to Add Emojis to a Vector Font . . . 1383
9.15.1 Intended Audience . . . 1384
9.15.2 Prerequisites . . . 1384
9.15.3 Append the Emoji Glyphs . . . 1384
9.16 How to Validate GUIs . . . 1386
9.16.1 Implementing GUIs Efficiently . . . 1386
9.16.2 Benchmarking GUIs . . . 1388
9.16.3 Debugging GUIs . . . 1389

9.16.4 Testing GUIs . . . 1391
9.17 How to Test a GUI Application with a (Software) Robot . . . 1392
9.17.1 Overview . . . 1392
9.17.2 Record the Scenario . . . 1392
9.17.3 Set Up the Scenario Player . . . 1395
9.17.4 Run the Scenario . . . 1397

10 Get Support **1399**

11 About MicroEJ **1400**

Index **1401**

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

- The Glossary chapter describes MicroEJ terminology.
- The Overview chapter introduces MicroEJ products and technology.
- The SDK User Guide chapter presents MICROEJ SDK (Software Development Kit).
- The Application Developer Guide presents how to develop a Java or JavaScript application on MICROEJ VEE (Virtual Execution Environment).
- The VEE Porting Guide teaches you how to integrate MICROEJ VEE into a C Board Support Package as well as simulation configurations.
- The Kernel Developer Guide introduces you to advanced concepts, such as partial updates and dynamic app life cycle workflows.
- The VEE Wear User Guide addresses the development of smartwatch applications using VEE Wear, specifically designed for low-power MCU and MPU.
- The Tutorials chapter covers a variety of topics related to developing with the MicroEJ ecosystem.

CHAPTER

ONE

MICROEJ GLOSSARY

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

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

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

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

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

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

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

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

Executable
An Executable is the result of the binary link of a Standalone Application with a VEE Port. It can be programmed into the flash memory of a device. (formerly called a Firmware)

Mono-Sandbox Executable
A Mono-Sandbox Executable is an Executable that implements an unmodifiable set of functions. (formerly called a Single-app Firmware)

Multi-Sandbox Executable
A Multi-Sandbox Executable is an Executable that implements the ability to be extended, by exposing a set of APIs and a memory space to link Sandboxed Applications. (formerly called a Multi-app Firmware)

Foundation Library

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

MICROEJ SDK

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

MICROEJ VEE

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

Mock

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

Module Manager

MicroEJ Module Manager (MMM) 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.

Simulator

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

VEE Port

A VEE Port is an implementation of MICROEJ VEE for a target device. It integrates an Architecture, one or more Foundation Libraries with their respective Abstraction Layers, and the board support package (BSP). It also includes associated Mocks for the Simulator. (formerly called Platform)

Virtual Device

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

CHAPTER

TWO

OVERVIEW

The MicroEJ product line offers profitable solutions to device manufacturers, application developers and service providers for:

- Device software development at lower cost and effort,
- Application development and deployment for generating extra revenue streams with services and data.

MicroEJ solutions enable delivery of user experience and business models similar to mobile Internet (smartphones and tablets) for embedded devices with strong cost constraints and strict resource limitations (processor performance, RAM and flash memory footprint, low-power). It also combines the techniques, methods and tools that drove the PC and mobile Internet software industry, with the complex technical foundations of embedded systems (fragmented processor architectures and diverse hardware-dependent software).

With MicroEJ solutions, you will use proven methods that cut software development time and cost. You will create software that delivers incredible user experience and adjusts to the needs of your business.

2.1 Getting Started

MicroEJ Getting Started is available on https://developer.microej.com/get-started/.

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

- Download and install an SDK Distribution;
- Select between one of the available boards;
- Import a demo Application;
- Download and install the corresponding VEE Port for the target hardware;
- Run the Application on Simulator with a Virtual Device;
- Build the Application for the target hardware to produce a Firmware;
- Deploy the Firmware on the board.

The following figure gives an overview of the SDK workflow:

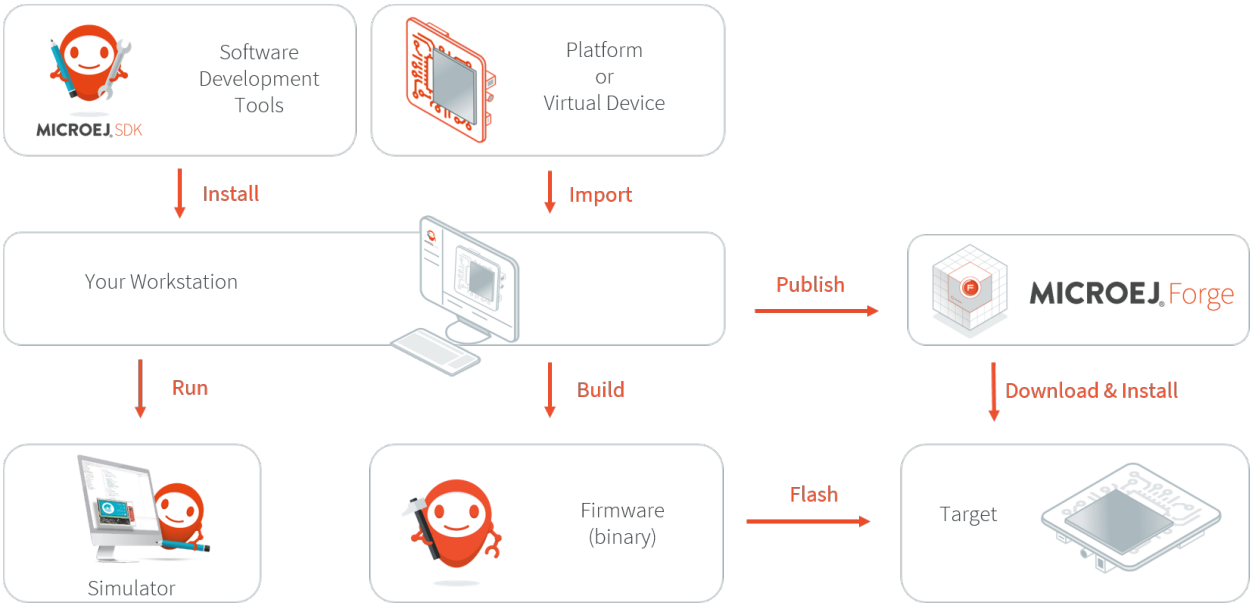


Fig. 1: SDK Workflow Overview

2.2 MICROEJ VEE

MicroEJ VEE (Virtual Execution Environment) is an applications container for resource-constrained embedded devices running on microcontrollers or microprocessors. It allows devices to run multiple and mixed managed code (Java, JavaScript) and C software applications.

MicroEJ VEE provides a fully configurable set of services that can be expanded, including:

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

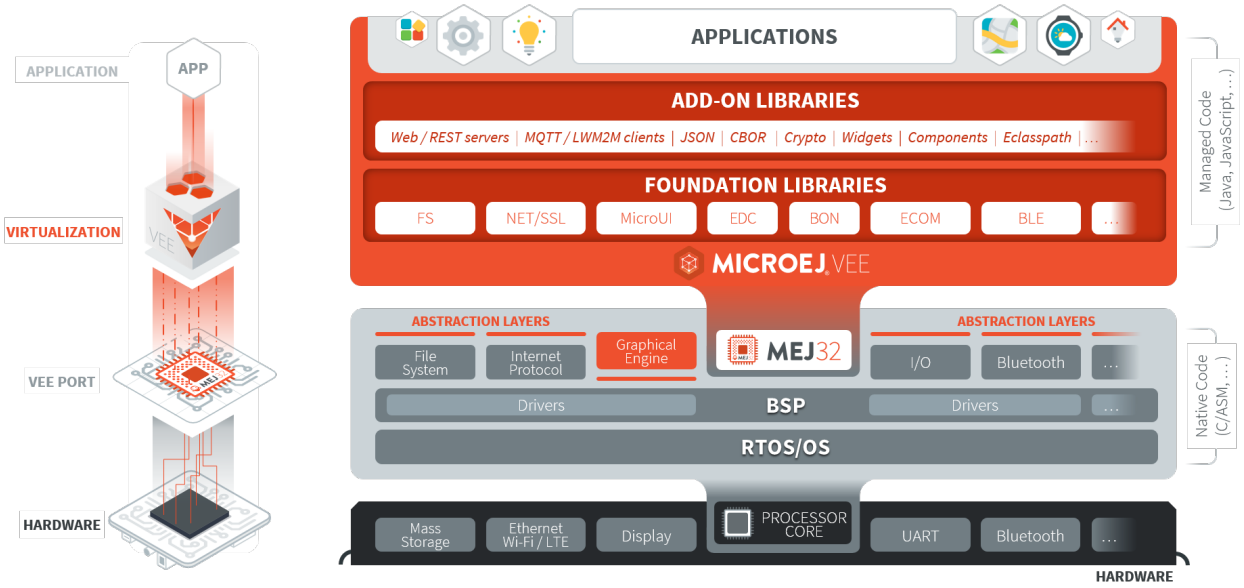


Fig. 2: MICROEJ VEE Overview

2.3 MICROEJ SDK

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

- Device Firmware development
- 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.

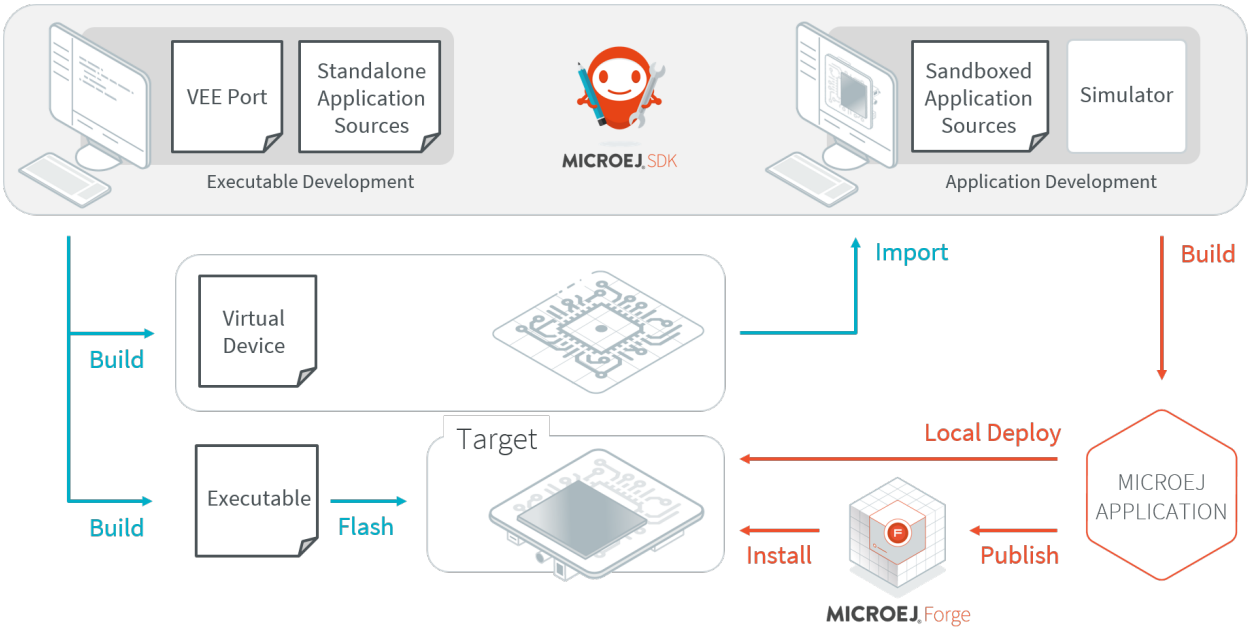


Fig. 3: SDK Workflow Overview

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

- An Executable binary to be flashed on OEM devices;
- A Virtual Device which will be used as a device simulator by application developers.

Using the SDK, 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 a hardware device equipped with the 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.

The following diagram outlines the SDK content. Please refer to the *SDK 5 User Guide* chapter for more details on the SDK and its usage.

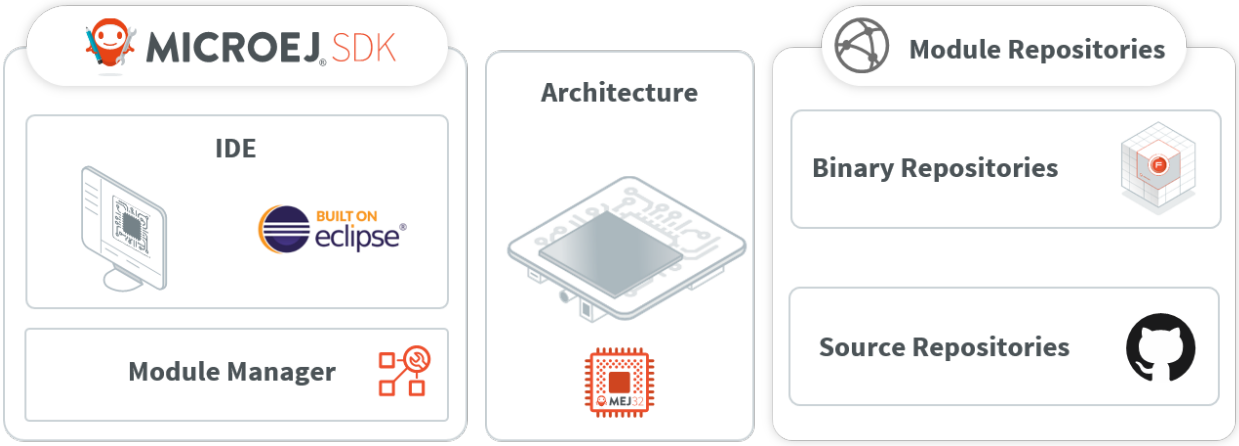


Fig. 4: SDK Components Overview

CHAPTER

THREE

SDK 5 USER GUIDE

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

Since the purpose of the SDK is to develop for targeted MCU/MPU computers (IoT, wearable, etc.), it is a cross-development tool. But unlike standard low-level cross-development tools, the SDK offers unique services like hardware simulation and local deployment to the target hardware.

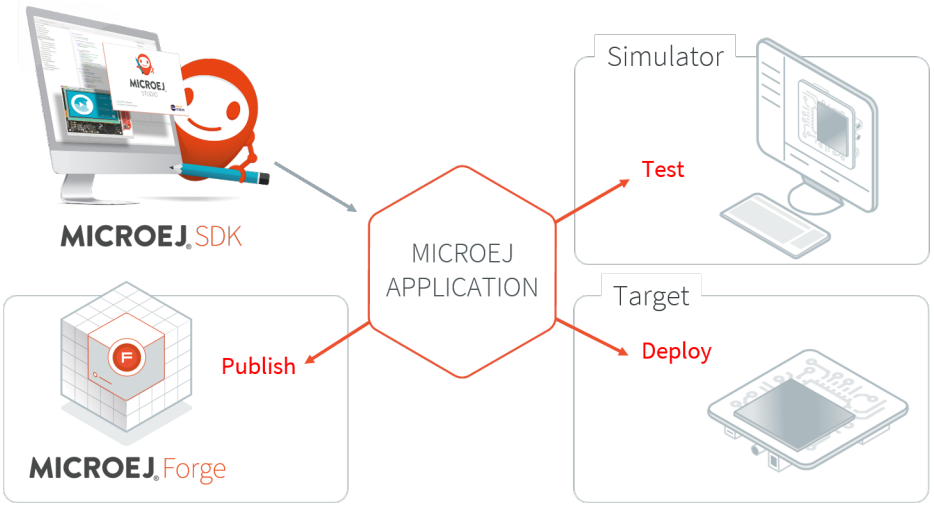


Fig. 1: MicroEJ Application Development Overview

The integrated environment is composed of the following main elements:

- **SDK Version 5.x**, an Integrated Development Environment (IDE) for writing and building Applications. It is based on Eclipse Java edition and relies on the integrated Java compiler (JDT).

  It is also packaged with Eclipse to produce a **SDK Distribution**.

- **MicroEJ Module Manager**, the module and build manager used to compile and package any kind of *modules natures*. It provides a Command Line Interface to build modules, especially used in a Continuous Integration environment. See *MicroEJ Module Manager* section for more details.

- **Architecture**, the software package that includes the MEJ32 port to a target instruction set and a C compiler, SOAR, core libraries and Simulator. See *MicroEJ Architecture* section for more details.

The SDK allows to connect repositories hosting software modules in source and binary form. By default, it is configured with the repositories provided MicroEJ Corp.:

- **Central and Developer Repository**, the modules repositories containing all the libraries required to develop an Application. See *Module Repositories* section for more details.
- **Github Repositories**, source repositories with examples and demos. See *GitHub Repositories* section for more details.

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

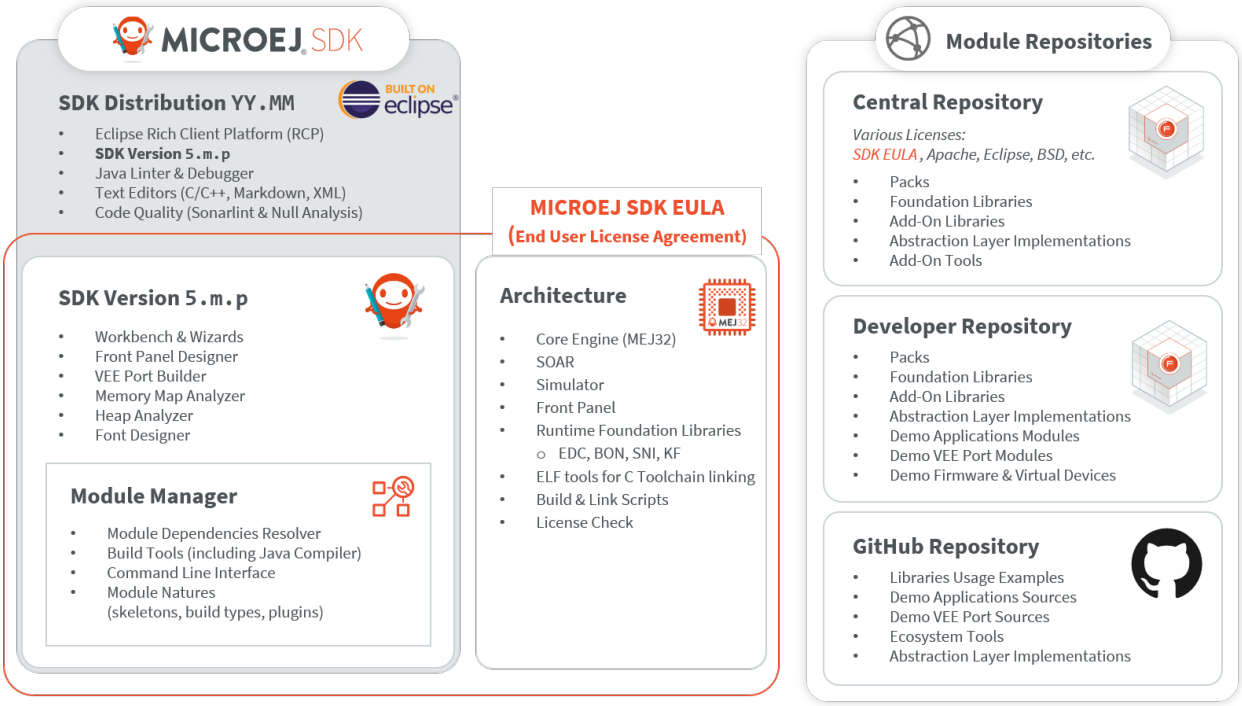


Fig. 2: SDK Detailed View

3.1 Installation

This chapter will guide you through the installation process of the SDK Distribution on your workstation.

If you want to evaluate MicroEJ, we recommend that you refer to the *Getting Started* chapter, which will guide you to install an SDK Distribution compatible with the Getting Started tutorials.

Otherwise, follow the instructions of the *Install Latest SDK Distribution* page to install the latest SDK Distribution compatible with your needs.



Fig. 3: SDK Splash Screen

3.1.1 Install Latest SDK Distribution

This section will guide you through the installation process of the latest SDK Distribution `24.01` using the step-by-step executable installer.

The SDK Distribution `24.01` requires a JDK 11 and thus can only work with an *Architecture 7.17.0* or higher. In all other cases, please jump to *Install SDK Distribution 21.11* section. See also the *System Requirements* page for more information on the list of supported environments.

Note: Launching the SDK Distribution installer requires administrator privileges and a JDK 11 installed by default on your workstation. If you don't have one of them or if you do not want to modify your default settings, please jump to *Install Portable SDK Distribution* section.

Download SDK Distribution

Download the SDK Distribution `24.01` installer for your operating system:

- Windows (.exe)
- Linux (.zip)
- macOS x86_64 - Intel chip (.zip)
- macOS aarch64 - M1 chip (.zip) (requires *Architecture 7.18.0 or higher*)

Check JDK Version

From the version `22.06`, the SDK Distribution installer requires a JDK 11 installed by default on your workstation. If you don't have any JDK installed, see the *Get JDK* section.

Check the default Java version by running the following command in a new terminal:

```
> java -version

openjdk version "11.0.15" 2022-04-19
OpenJDK Runtime Environment Temurin-11.0.15+10 (build 11.0.15+10)
OpenJDK 64-Bit Server VM Temurin-11.0.15+10 (build 11.0.15+10, mixed mode)
```

Now you can proceed with the installation steps.

Install SDK Distribution

- Launch the installer executable
  - On Windows, start `MicroEJ-SDK-Installer-Win64-24.01.exe`.
  - On Linux, unzip `MicroEJ-SDK-Installer-Linux64-24.01.zip` and start `MicroEJ-SDK-Installer-Linux64-1.3.0.sh`.
  - On macOS, unzip `MicroEJ-SDK-Installer-MacOS-24.01.zip` and start `MicroEJ-SDK-Installer-MacOS-1.3.0.app`.
  - Or unzip `MicroEJ-SDK-Installer-MacOS-A64-24.01.zip` and start `MicroEJ-SDK-Installer-MacOS-A64-1.3.0.app`. In case of error, check your app has not been put in quarantine (see *macOS troubleshooting* section)

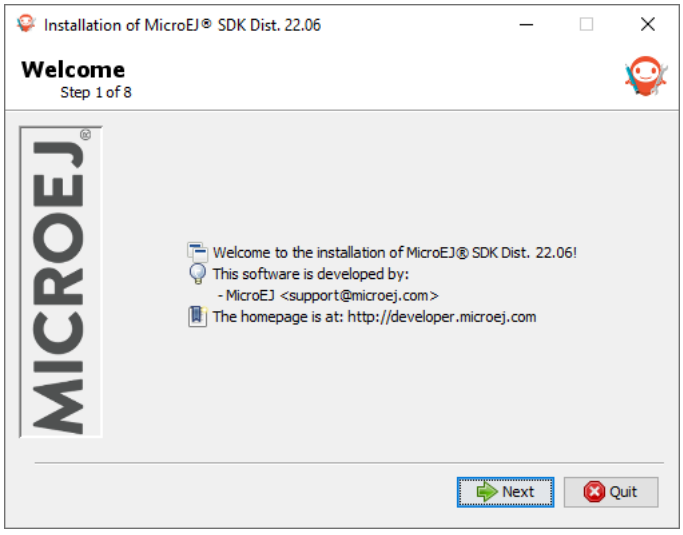


Fig. 4: Welcome to the installer

- Click on the `Next` button.
- Select `I accept the terms of this license agreement.`. Then click on the `Next` button.

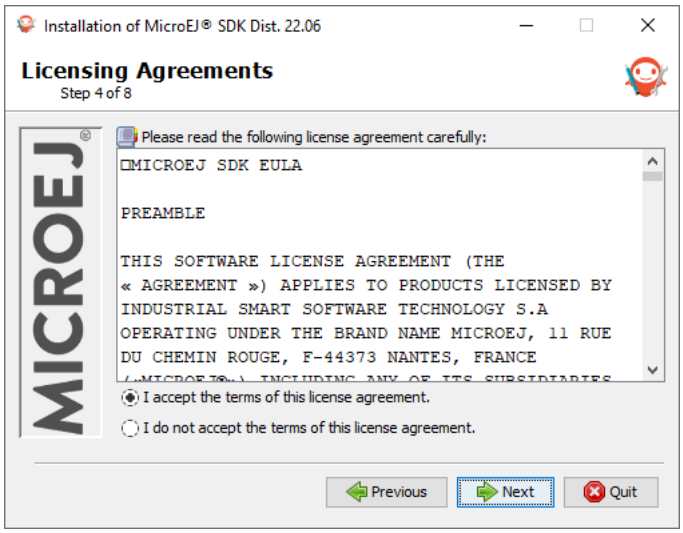


Fig. 5: Accept the terms of this license agreement

- Select the installation path of your SDK. By default it is `C:\Program Files\MicroEJ\MicroEJ-SDK-24.01` for Windows. Then click on the `Next` button.

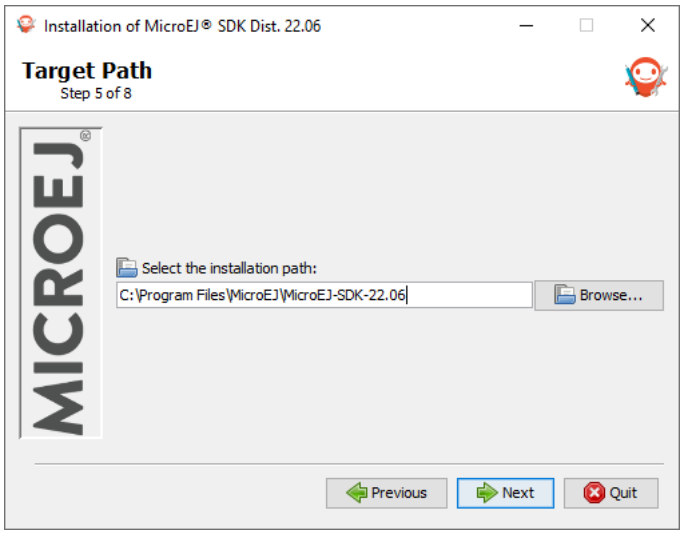


Fig. 6: Choose the installation path

- Click on the `OK` button to confirm the installation path.

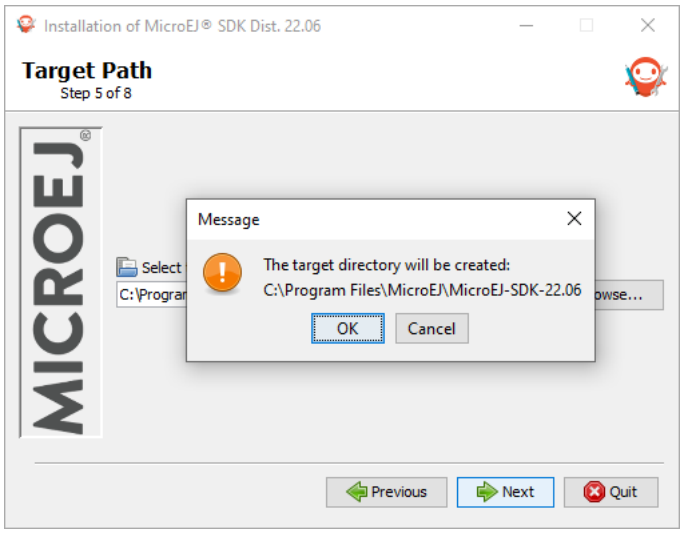


Fig. 7: Confirm your installation path

- Wait until the installation is done. Then click on the `Next` button.

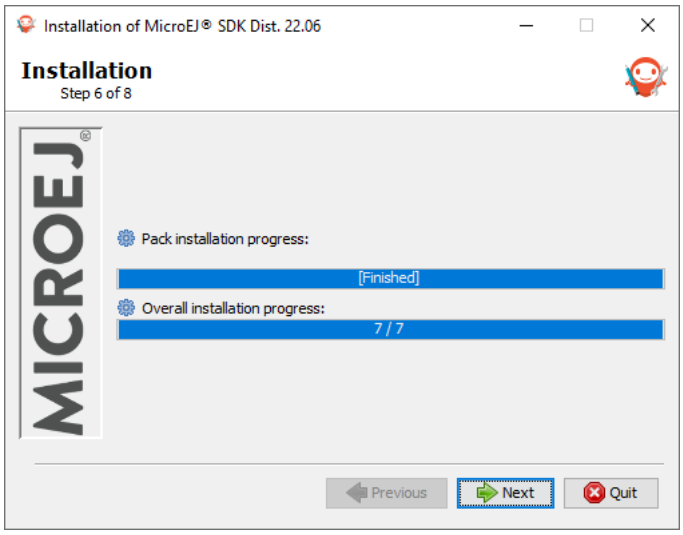


Fig. 8: Installation in progress

- Select options depending on your own preferences. Then click on the `Next` button.

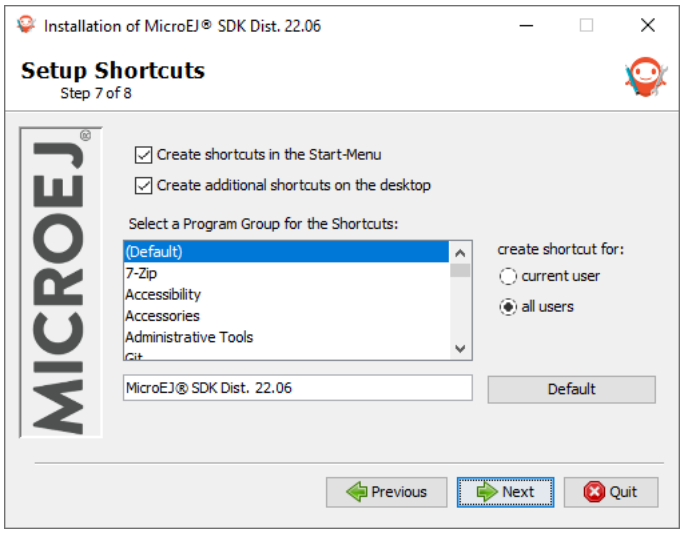


Fig. 9: Select the options

- The installation has completed successfully. Click on the `Done` button.

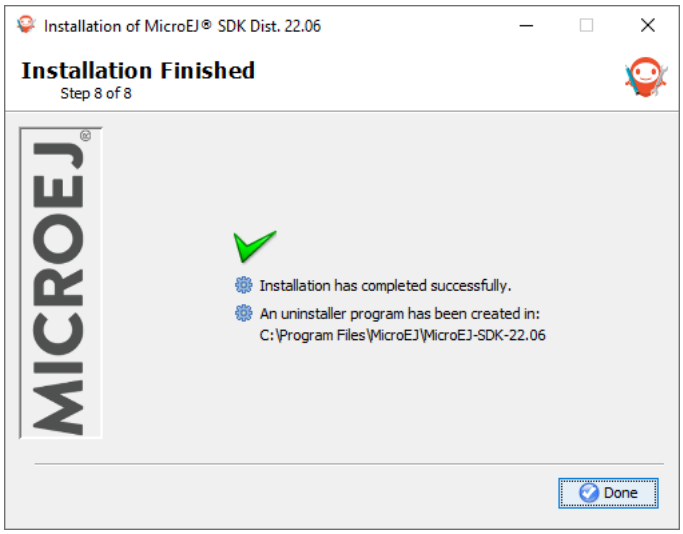


Fig. 10: Your installation has completed successfully

The SDK Distribution is now installed on your computer. You can launch it from your application launcher or by executing the `MicroEJ` executable in the installation path.

Once the SDK is started, it is recommended to check if updates are available (see *Update SDK Version* section). If your are running SDK on Windows OS, it is also strongly recommended to configure *Windows defender exclusion rules*.

3.1.2 Update SDK Version

Once you have an SDK Distribution installed, you can update the included SDK Version to a newer version.

Note: If you want to know which SDK version is currently installed in your SDK Distribution, see the *SDK Version* chapter.

To update your SDK Distribution to a newer SDK version, follow the next steps:

- Select `Help` > `Check for updates` .

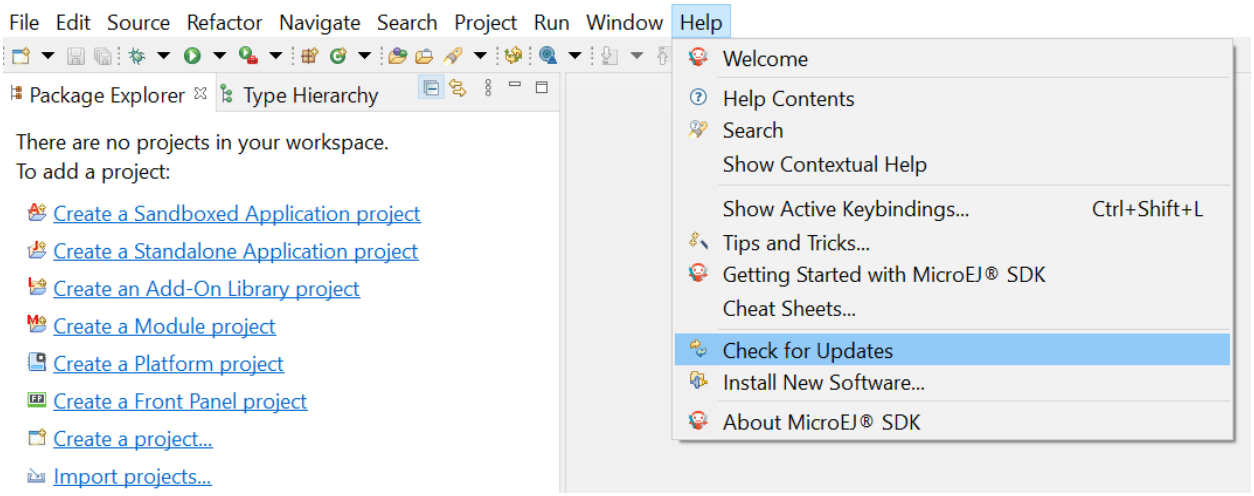


Fig. 11: Check for updates

- If your SDK is up-to-date, you will see the following screen:

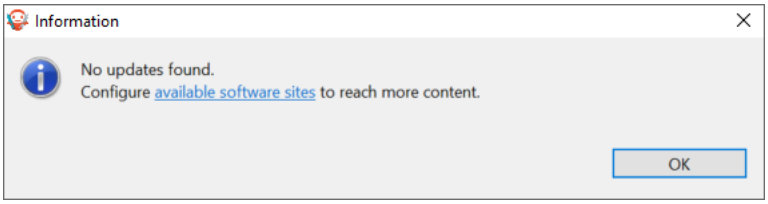


Fig. 12: No update available

- If an update is available, you will see the following screen:

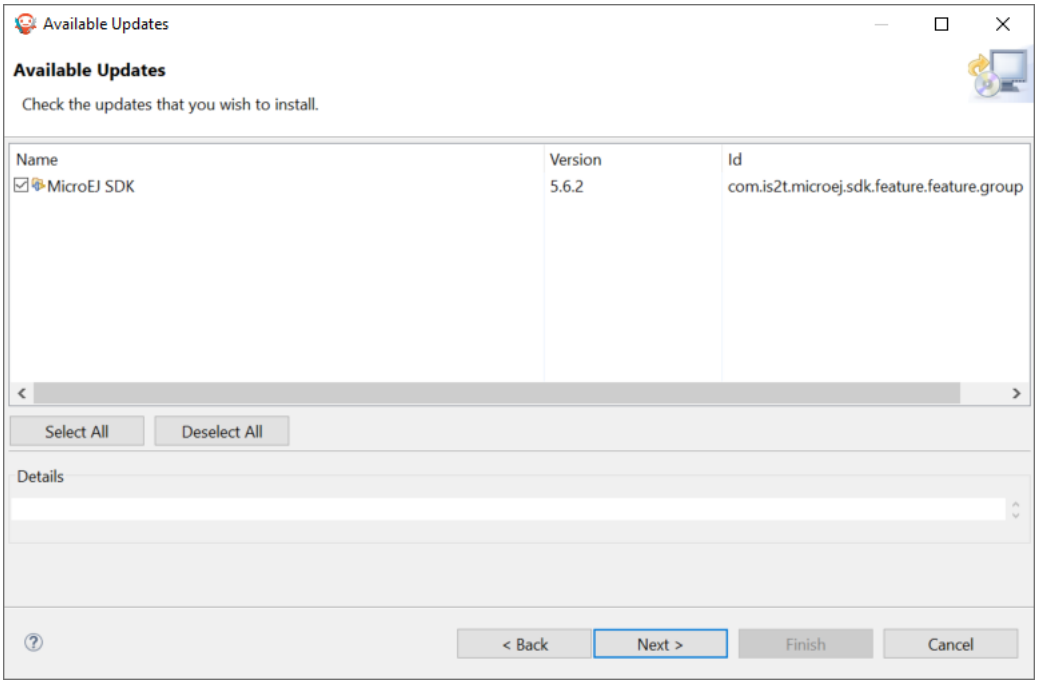


Fig. 13: Update available

- Check the version you want to install. Then click on the `Next` button.
- Review and confirm the updates. Then click on the `Next` button.

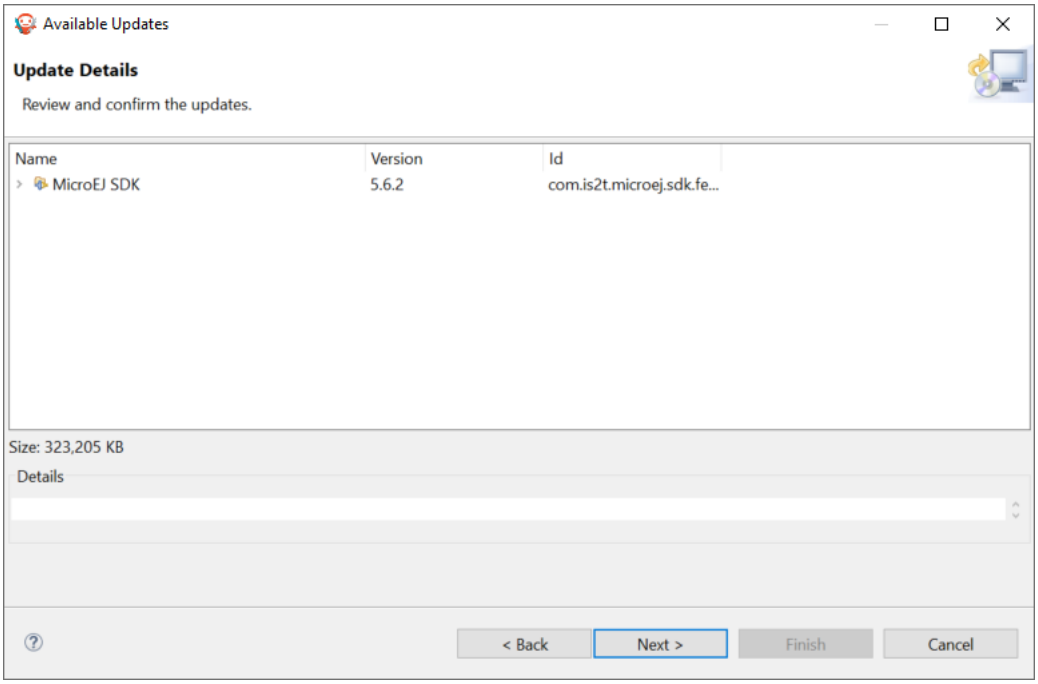


Fig. 14: Review the updates

- Select `I accept the terms of the license agreements.` . Then click on the `Finish` button.

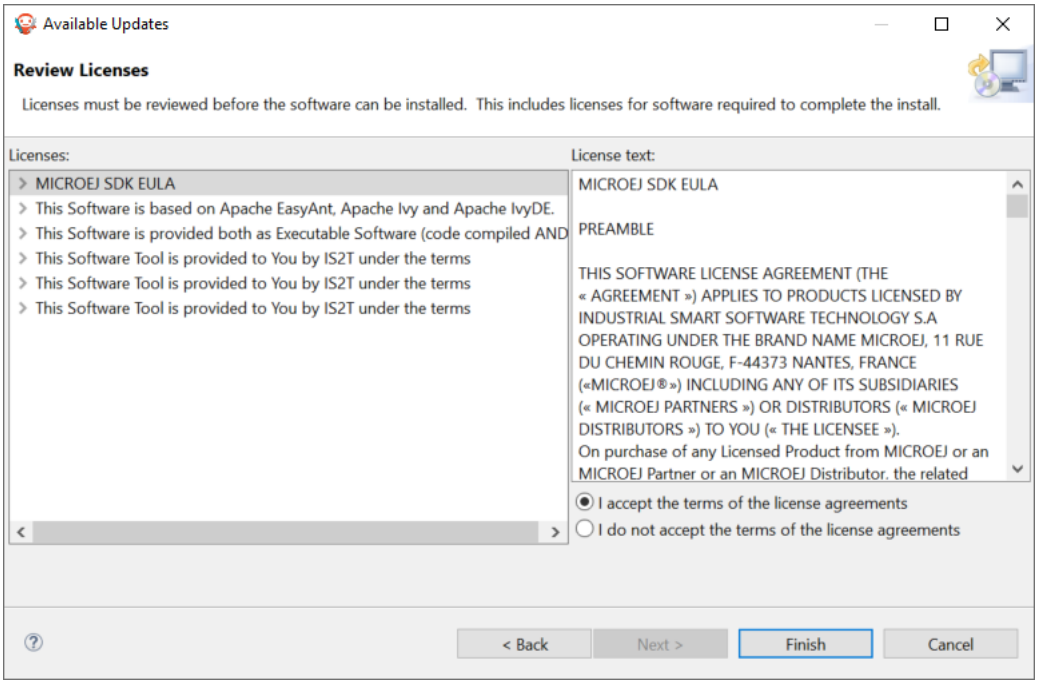


Fig. 15: Accept the terms of the license agreement

- Wait until the Software Update pop-up appears. Then click on the Restart Now button.

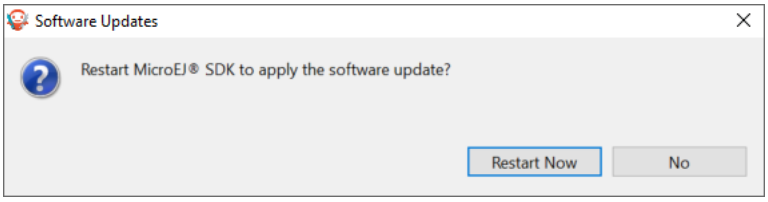


Fig. 16: Restart your SDK.

The update of your SDK is done.

3.1.3 Install Other SDK Distributions

Install Portable SDK Distribution

The portable package allows you to install the SDK Distribution when the use of the SDK Distribution installer is not possible or not desired, for example:

- you do not have administrator privileges on your workstation;
- you want to install SDK Distribution `23.07` but JDK 11 is not your default JDK version;
- you want to install SDK Distribution up to `21.11` but JDK 8 is not your default JDK version.

Perform the following steps:

- Download the Portable SDK Distribution for your operating system:

SDK Distribution	JDK Version	Windows	Linux	macOS x86_64 (Intel chip)	macOS aarch64 (M1 chip)[1]
24.01	11	Portable (.zip)	Portable (.zip)	Portable (.zip)	Portable (.zip)
23.07	11	Portable (.zip)	Portable (.zip)	Portable (.zip)	Portable (.zip)
21.11	8	Portable (.zip)	Portable (.zip)	Portable (.zip)	*N/A*

- Once downloaded, extract the zip file in a local directory of your choice
- Edit the `MicroEJ-SDK.ini` file
- Configure the path to the JDK version indicated above by adding the option `-vm` at the beginning of the file. If you don't have any JDK installed, see the *Get JDK* section.

```
-vm
[path_to_jdk]/bin
-startup
plugins/org.eclipse.equinox.launcher_1.6.400.v20210924-0641.jar
...
```

- Start the SDK by executing `MicroEJ-SDK.exe` on Windows or `MicroEJ-SDK` on Linux or macOS.

Once the SDK is started, it is recommended to check if updates are available (see *Update SDK Version* section). If your are running SDK on Windows OS, it is also strongly recommended to configure *Windows defender exclusion rules*.

Install SDK Distribution 21.11

This section will guide you through the installation process of the SDK Distribution `21.11` using the step-by-step executable installer.

The SDK Distribution `21.11` requires a JRE or a JDK 8 and is not available for macOS with M1 chips. See the *System Requirements* page for more information on the list of supported environments.

Note: Launching the SDK Distribution installer requires administrator privileges and a JDK 8 installed by default on your workstation. If you don't have one of them or if you do not want to modify your default settings, please jump to *Install Portable SDK Distribution* section.

---

[1] SDK Distribution for macOS aarch64 (M1 chip) requires *Architecture 7.18.0 or higher*.

Download SDK Distribution

Download the SDK Distribution `21.11` installer for your operating system:

- Windows (.exe)
- Linux (.zip)
- macOS x86_64 - Intel chip (.zip)

Check JDK Version

The SDK Distribution `21.11` installer requires a JDK 8 installed by default on your workstation. If you don't have any JDK installed, see the *Get JDK* section.

Check the default Java version by running the following command in a new terminal:

```
> java -version

java version "1.8.0_281"
Java(TM) SE Runtime Environment (build 1.8.0_281-b09)
Java HotSpot(TM) 64-Bit Server VM (build 25.281-b09, mixed mode)
```

Now you can proceed with the installation steps.

Install SDK Distribution

- Launch the installer executable
  - On Windows, start `MicroEJ-SDK-Installer-Win64-21.11.exe`.
  - On Linux, unzip `MicroEJ-SDK-Installer-Linux64-21.11.zip` and start `MicroEJ-SDK-Installer-Linux64-21.11.sh`.
  - On macOS, unzip `MicroEJ-SDK-Installer-Linux64-21.11.zip` and start `MicroEJ-SDK-Installer-MacOS-21.11.app`. In case of error, check your app has not been put in quarantine (see *macOS troubleshooting* section).

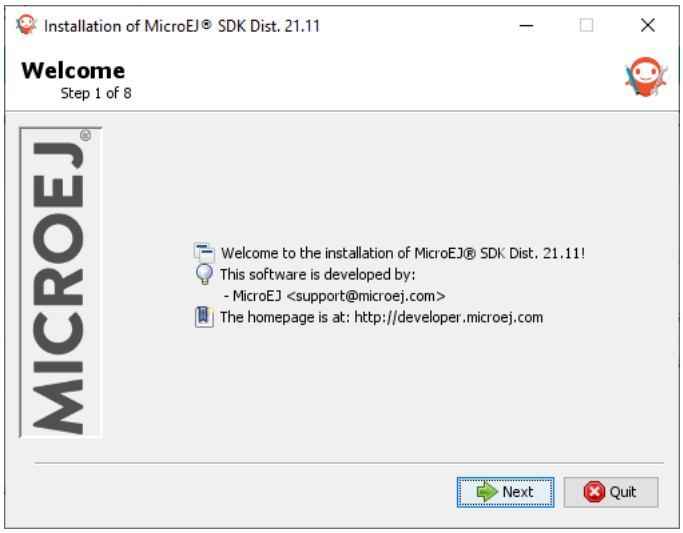


Fig. 17: Welcome to the installer

- Click on the `Next` button.
- Select `I accept the terms of this license agreement.` . Then click on the `Next` button.

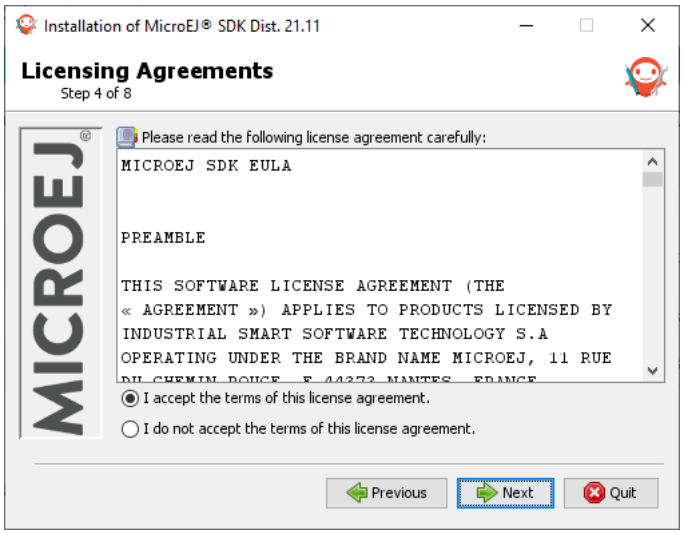


Fig. 18: Accept the terms of this license agreement

- Select the installation path of your SDK. By default it is `C:\Program Files\MicroEJ\MicroEJ-SDK-21.11` for Windows. Then click on the `Next` button.

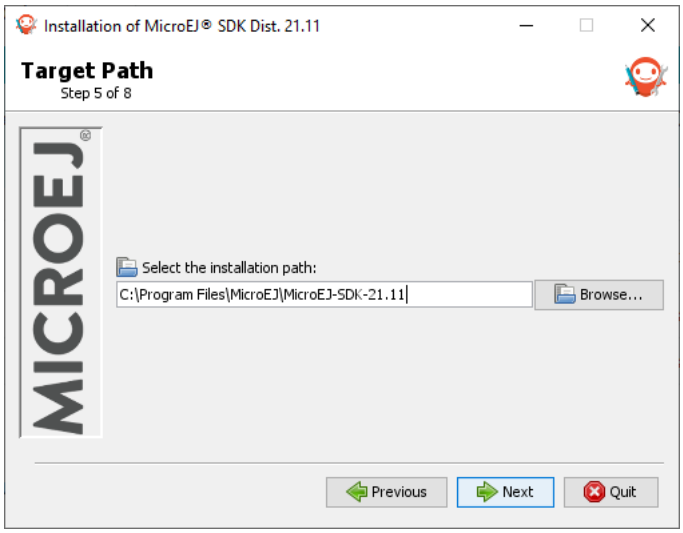


Fig. 19: Choose the installation path

- Click on the OK button to confirm the installation path.

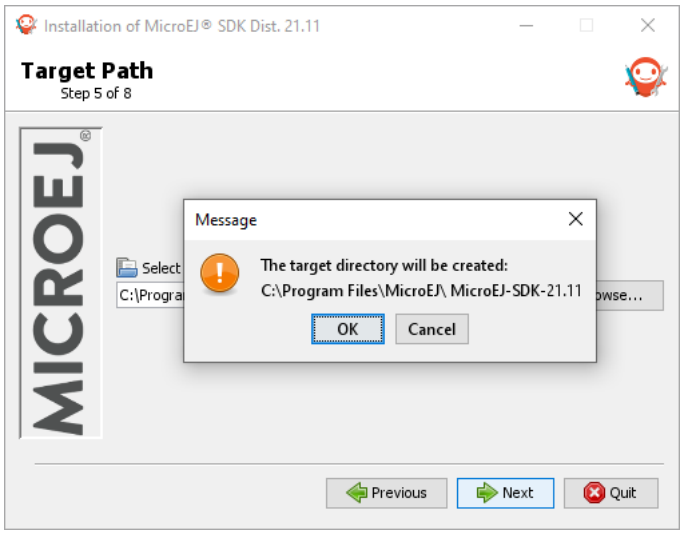


Fig. 20: Confirm your installation path

- Wait until the installation is done. Then click on the Next button.

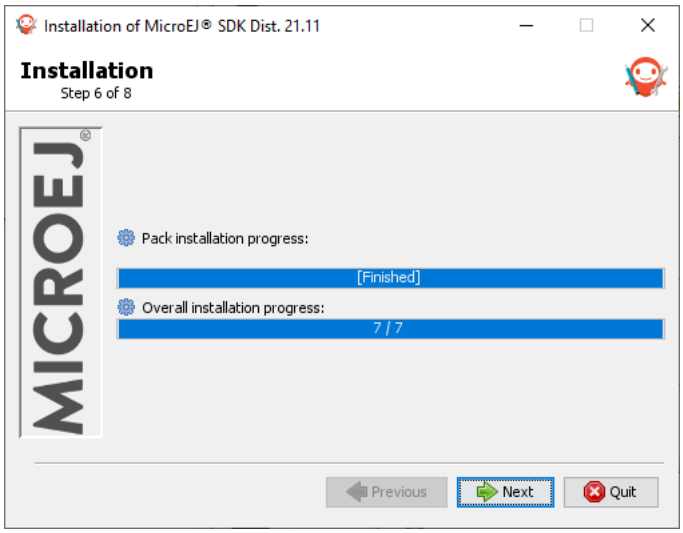


Fig. 21: Installation in progress

- Select options depending on your own preferences. Then click on the Next button.

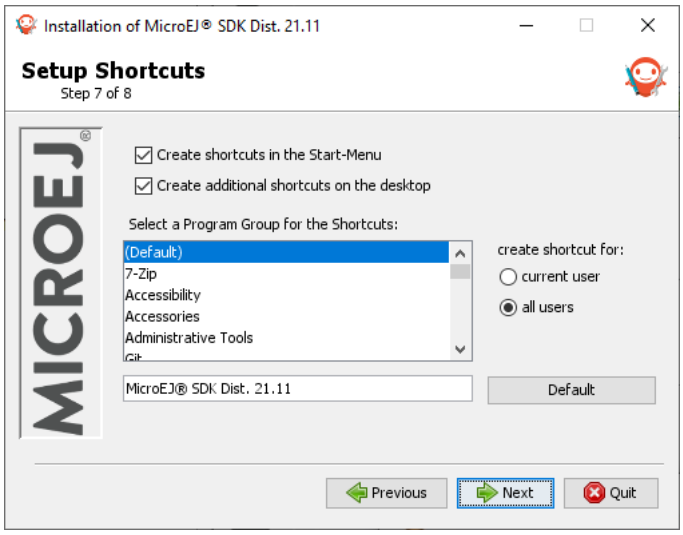


Fig. 22: Select the options

- The installation has completed successfully. Click on the Done button.

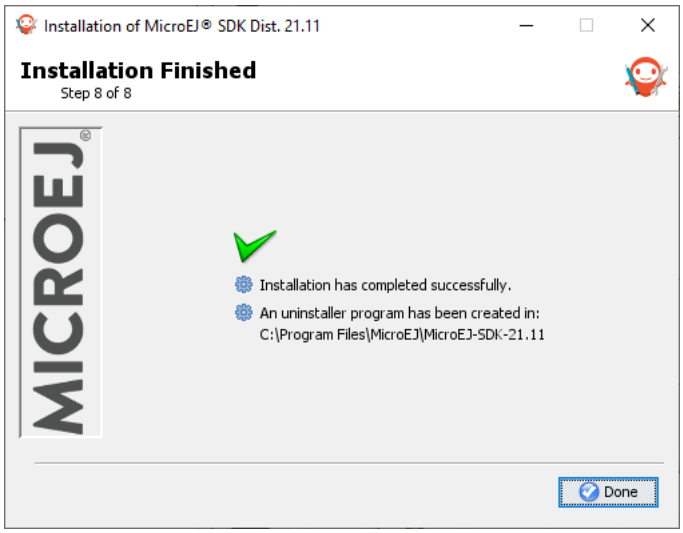


Fig. 23: Your installation has completed successfully

The SDK Distribution is now installed on your computer. You can launch it from your application launcher or by executing the `MicroEJ` executable in the installation path.

Once the SDK is started, it is recommended to check if updates are available (see *Update SDK Version* section). If your are running SDK on Windows OS, it is also strongly recommended to configure *Windows defender exclusion rules*.

This section applies when the installation of the *latest SDK Distribution* via the installer does not fit your case:

- you want to install the latest SDK Distribution compatible with JDK8, see *Install SDK Distribution 21.11* section.
- you want to install an SDK Distribution with no native installer, see *Install Portable SDK Distribution* section.
- you want to install an old SDK Distribution. The following table gives you access to all the SDK `5.x` Distributions download links.

SDK Distribution	JDK Version	Windows	Linux	macOS x86_64 (Intel chip)	macOS aarch64 (M1 chip)[1]	SDK Version	Eclipse Version
24.01	11	• Installer (.exe) • Portable (.zip)	• Installer (.zip) • Portable (.zip)	• Installer (.zip) • Portable (.zip)	• Installer (.zip) • Portable (.zip)	5.8.2	2022-03
23.07	11	• Installer (.exe) • Portable (.zip)	• Installer (.zip) • Portable (.zip)	• Installer (.zip) • Portable (.zip)	• Installer (.zip) • Portable (.zip)	5.8.0	2022-03
23.02	11	• Installer (.exe) • Portable (.zip)	• Installer (.zip) • Portable (.zip)	• Installer (.zip) • Portable (.zip)	• Installer (.zip) • Portable (.zip)	5.7.0	2022-12
22.06	11	• Installer (.exe) • Portable (.zip)	• Installer (.zip) • Portable (.zip)	• Installer (.zip) • Portable (.zip)	• Installer (.zip) • Portable (.zip)	5.6.0	2022-03
21.11	8	• Installer (.exe) • Portable (.zip)	• Installer (.zip) • Portable (.zip)	• Installer (.zip) • Portable (.zip)	*N/A*	5.5.0	2020-06
21.03	8	• Installer (.exe) • Portable (.zip)	• Installer (.zip) • Portable (.zip)	• Installer (.zip) • Portable (.zip)	*N/A*	5.4.0	2020-06
20.12	8	• Installer (.exe) • Portable (.zip)	• Installer (.zip) • Portable (.zip)	• Installer (.zip) • Portable (.zip)	*N/A*	5.3.1	2020-06
20.10	8	• Installer (.exe) • Portable (.zip)	• Installer (.zip) • Portable (.zip)	• Installer (.zip) • Portable (.zip)	*N/A*	5.3.0	2020-06
20.07	8	• Installer	• Installer	• Installer	*N/A*	5.2.0	4.7.2

Finally, if you need an older SDK Distribution, browse the SDK Downloads Page.

3.1.4 System Requirements

- **Hardware :**
  - Intel x64 (Dual-core i5 minimum) or macOS AArch64 (M1) processor
  - 4GB RAM (minimum)
  - 2GB Disk (minimum)
- **Operating Systems :**
  - Windows 11, Windows 10, Windows 8.1 or Windows 8
  - Linux distributions (tested on Ubuntu 18.04, 20.04 and 22.04) - As of SDK Distribution `20.10` (based on Eclipse 2020-06), Ubuntu 16.04 is not supported.
  - macOS x86_64 with Intel chip (tested on version 10.13 High Sierra, 10.14 Mojave)
  - macOS aarch64 with M1 chip (tested on version 12.0.1 Monterey), from SDK Distribution `22.06` (requires *Architecture 7.18.0 or higher*)
- **Java Runtime Environment :**

The compatible JRE/JDK version depends on the Distribution, the SDK and the Architecture version. This table lists the supported combinations:

Distribution	SDK	Architecture	JRE/JDK
>= 22.06	>= 5.6.0	>= 7.17.0	JDK 11
<= 21.11	>= 5.6.0	>= 7.17.0	JRE or JDK 8 or 11
<= 21.11	< 5.6.0	*	JRE or JDK 8
<= 21.11	*	< 7.17.0	JRE or JDK 8

The combinations not listed here are not supported. For the supported combinations, tests have been done with both the Oracle and the Eclipse Adoptium JDK builds.

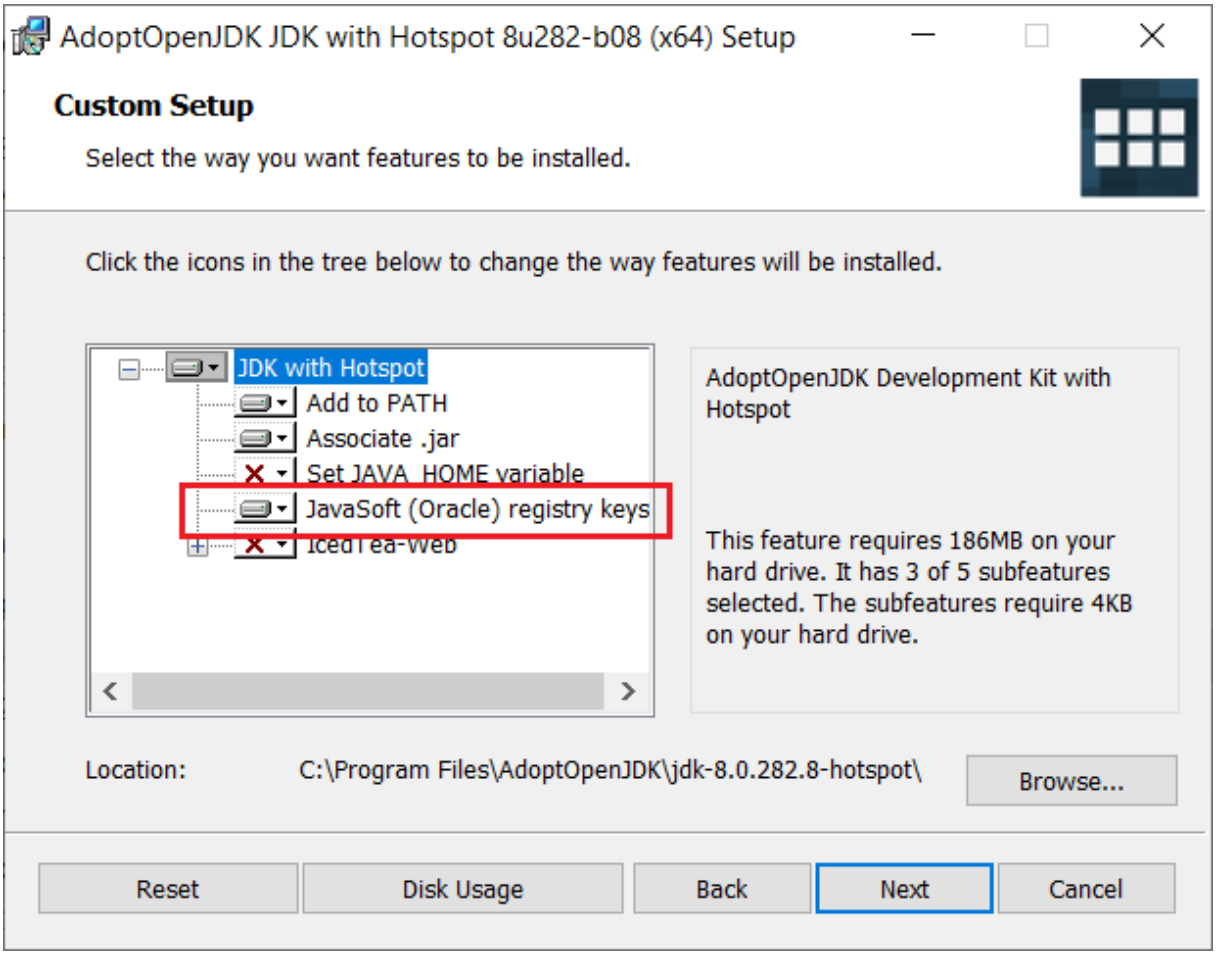
Warning: It is important to note that the SDK Distributions `22.06` and higher require a JDK `11` (not a JRE) and can be used only with an *Architecture* `7.17.0` or more.

Get JDK

You can download and install a JDK from Adoptium or Oracle.

Warning: Up to version 23.02 of the SDK Distribution, when installing the Eclipse Temurin/AdoptOpenJDK build on Windows, the option `JavaSoft (Oracle) registry keys` must be enabled:

[1] SDK Distribution for macOS aarch64 (M1 chip) requires *Architecture 7.18.0 or higher*.

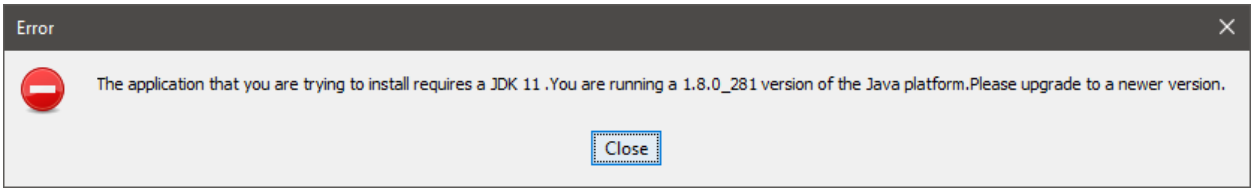


Without this option, the SDK installer cannot find the JDK and the message `The application requires a Java Runtime Environment 8` is displayed.

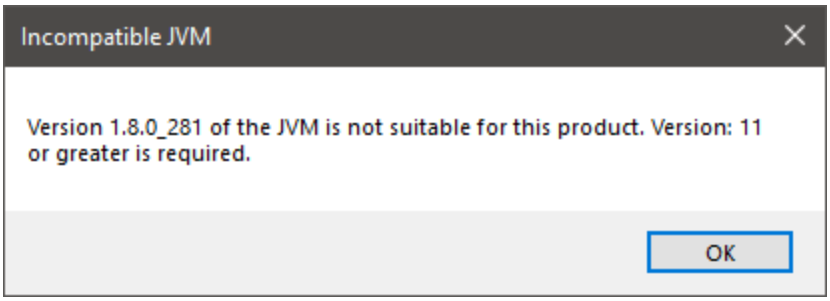
3.1.5 Troubleshooting

Incompatible Default Java Version

When launching the installer, you may get the following error: `The application you are trying to install requires a JDK11` .



Or when launching the SDK, you may get the following error: `Version: 11 or greater is required` .



The default Java version installed on your system is not compatible. You have two options:

- either install a JDK 11 as your default JVM. If you are on Windows OS and your SDK Distribution version is 23.02 or lower, ensure you enabled `JavaSoft (Oracle) registry keys` during the JDK installation (see *Get JDK*),
- or install the *portable SDK Distribution* if you don't want to modify your default JVM version.

This latter case is recommended if you are installing SDK Dist. `22.06` or higher while you already have active projects based on SDK Dist. `21.11`.

Windows Specifics

If you are using Windows Defender as your default antivirus software, the SDK may be slowed down as it manipulates lots of JAR files (which are ZIP files) that are regularly analyzed.

To improve the SDK experience, please find below a list of folders that should be excluded from Windows Defender monitoring:

- `%USERPROFILE%\.eclipse`
- `%USERPROFILE%\.ivy2`
- `%USERPROFILE%\.microej`
- `%USERPROFILE%\.p2`
- `%USERPROFILE%\AppData\Local\Temp\microej`
- `C:\Program Files\MicroEJ` or the custom directory where the SDK has been installed
- your workspace(s) folder(s)

The exclusion page is available in the `Settings` application (Windows Security > Virus & threat protection > Manage settings > Exclusions > Add or remove exclusions).

Linux Specifics

Starting the SDK on a linux distribution may produce troubles such as missing content pages. This is related to incomplete Eclipse SWT configuration (see Eclipse GTK wiki page).

One solution is to configure Eclipse as follows:

- Add the next lines to `MicroEJ-SDK.ini`, before `-vmargs` argument:

```
--launcher.GTK_Version 2
```

- Ensure GTK is correctly installed (`libwebkitgtk` packet)

- Configure the following environment variables

```
MOZILLA_FIVE_HOME=/usr/lib/mozilla
LD_LIBRARY_PATH=${MOZILLA_FIVE_HOME}:${LD_LIBRARY_PATH}
```

- Restart the SDK
- Check there is not more SWT/MOZILLA related errors (Window > Show View > Other... > General > Error Log)

MacOS Specifics

When launching the SDK using the `.app` file, you may encounter the following message:

```
"MicroEJ-SDK-xx.xx" is damaged and can't be opened. You should move it to the Trash.
```

or this one:

```
"MicroEJ-SDK-xx.xx" cannot be opened because the developer cannot be verified.
```

This is due to macOS putting applications in quarantine when downloaded with a browser. Use this command to remove the SDK application from quarantine:

```
sudo xattr -rd com.apple.quarantine sdk.app
```

where `sdk.app` is the SDK file name.

3.2 Licenses

3.2.1 SDK EULA

MICROEJ SDK is licensed under the SDK End User License Agreement (EULA), which covers the following elements:

- SDK Tools & Plugins packaged in the SDK *5.x* Version,
- Architectures,
- Modules published to the *Central Repository* with the SDK EULA license, such as GUI or Networking Pack (see *Central Repository Licensing* for more details).

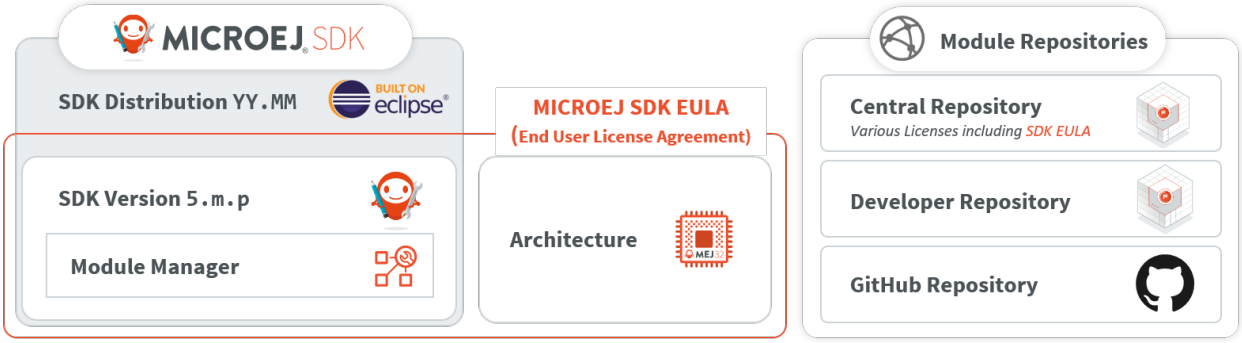


Fig. 24: SDK EULA Coverage

3.2.2 License Manager Overview

Architectures are distributed in two different versions:

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

The license manager is provided with Architectures and then integrated into VEE Ports, consequently:

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

The list of installed licenses is available in the SDK preferences dialog page in `Window` > `Preferences` > `MicroEJ` :

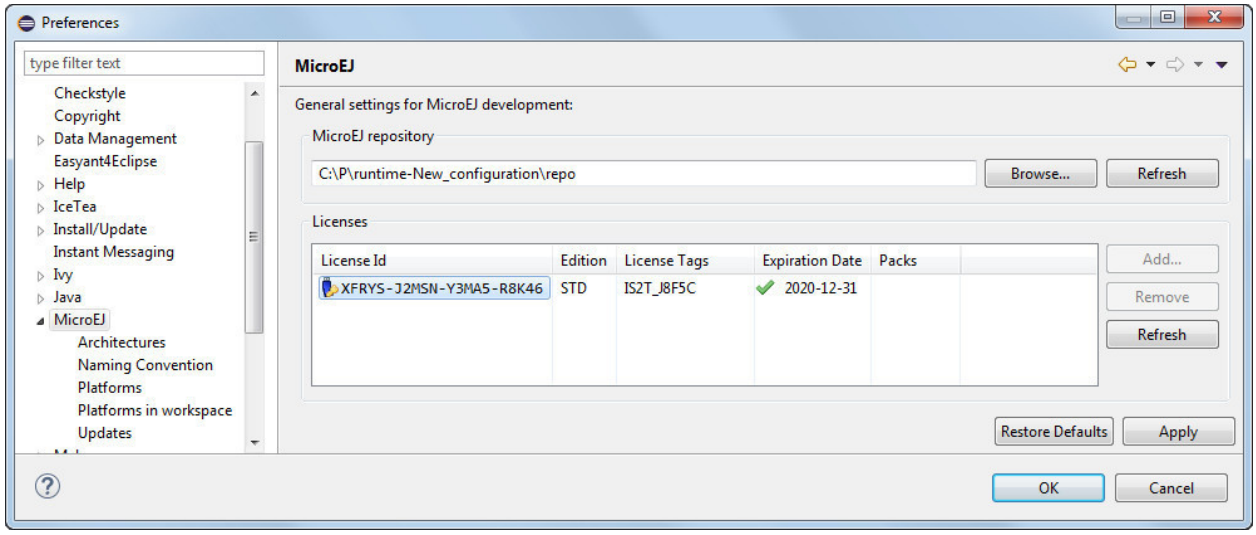


Fig. 25: License Manager View

3.2.3 License Check

The table below summarizes where the license is checked.

Application	Run on Simulator (Virtual Device)	Build on Device	Documentation Link
Standalone Application or Kernel Application	NO	**YES**	*Run on the Device*
Sandboxed Application	NO	NO	*Application Linking*

3.2.4 Evaluation Licenses

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

Get your Machine UID

Retrieving the machine UID depends on the kind of VEE Port being evaluated.

If your VEE Port is already *imported in Package Explorer* and built with *MicroEJ Module Manager*, the Architecture has been automatically imported. The machine UID will be displayed when building a *Standalone Application on device*.

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

Otherwise, an Architecture or VEE Port should have been manually imported from the SDK preferences page. The machine UID can be retrieved as follows:

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

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

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

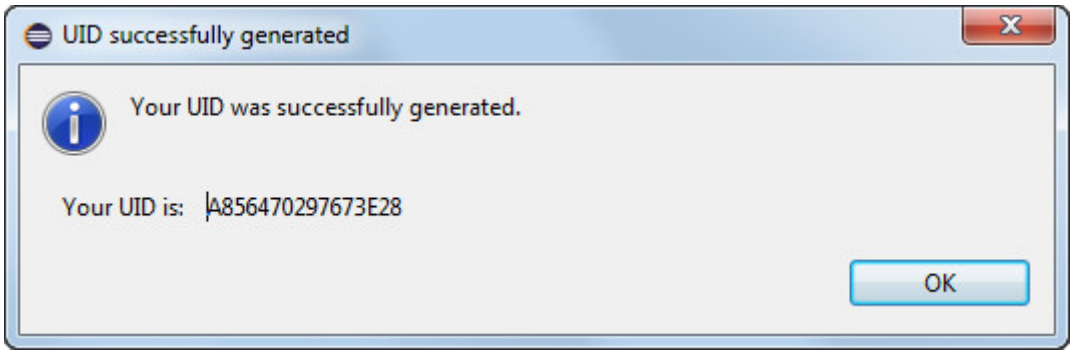


Fig. 26: Machine UID for Evaluation License

Request your Activation Key

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

Install the License Key

If your VEE Port is already *imported in Package Explorer* and built with *MicroEJ Module Manager*, the license key zip file must be simply dropped to the `~/.microej/licenses/` directory (create it if it doesn't exist).

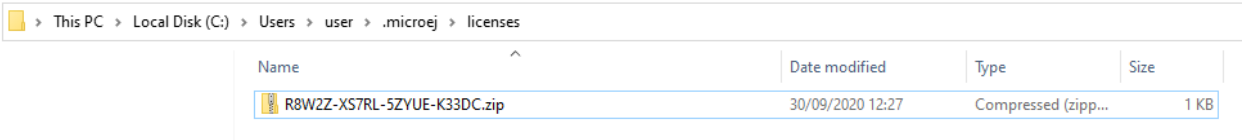


Fig. 27: MicroEJ Shared Licenses Directory

Note: The SDK Preferences page will be automatically refreshed when building a *Standalone Application on device*.

Otherwise, the license key must be installed as follows:

- Go back to the SDK.
- Select the Window > Preferences > MicroEJ menu.
- Press Add… .
- Browse the previously downloaded activation key archive file.
- Press OK. A new license is successfully installed.
- Go to Architectures sub-menu and check that all Architectures are now activated (green check).
- Your SDK is successfully activated.

If an error message appears, the license key could not be installed. (see section *Troubleshooting*). A license key can be removed from the key-store by selecting it and by clicking on Remove button.

Troubleshooting

Unable to add an Evaluation license key in the SDK

Consider this section when an error message appears while adding the Evaluation license key. Before contacting *our support team*, please check the following conditions:

- Key is corrupted (wrong copy/paste, missing characters, or extra characters)
- Key has not been generated for the installed environment
- Key has not been generated with the machine UID
- Machine UID has changed since submitting license request and no longer matches license key
- Key has not been generated for one of the installed Architectures (no license manager able to load this license)

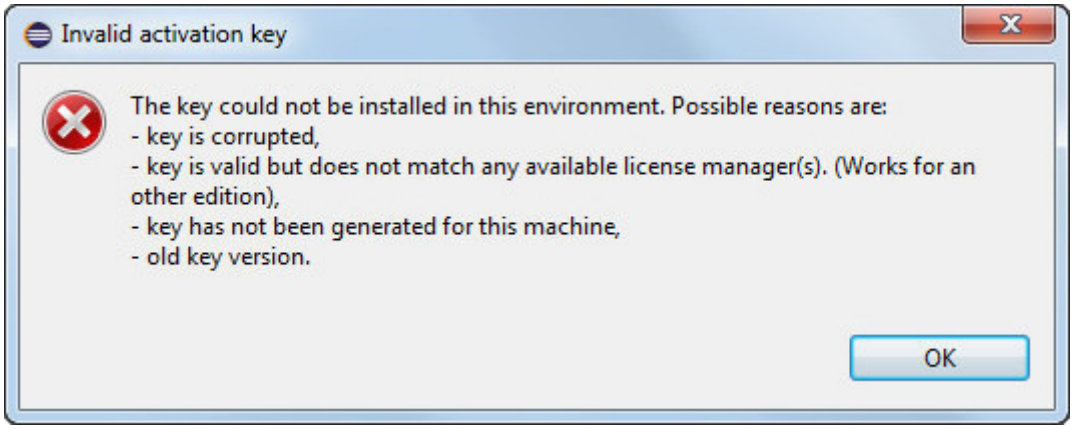


Fig. 28: Invalid License Key Error Message

Machine UID has changed

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

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

3.2.5 Production Licenses

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



Fig. 29: MicroEJ USB Dongle

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

Request your Activation Key

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

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

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

Activate your USB Dongle

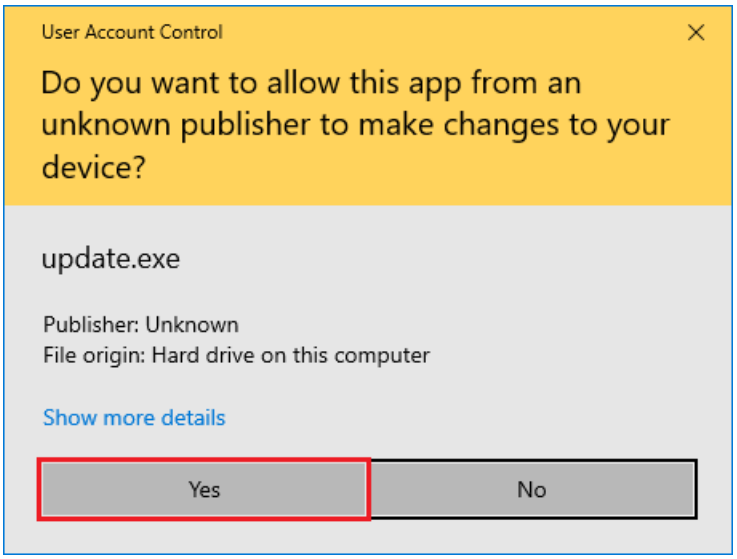
This section contains instructions that will allow you to flash your USB dongle with the proper activation key.

You shall ensure that the following prerequisites are met :

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

You can then proceed to the USB dongle update:

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



- Click on the `Update` button (no password needed)

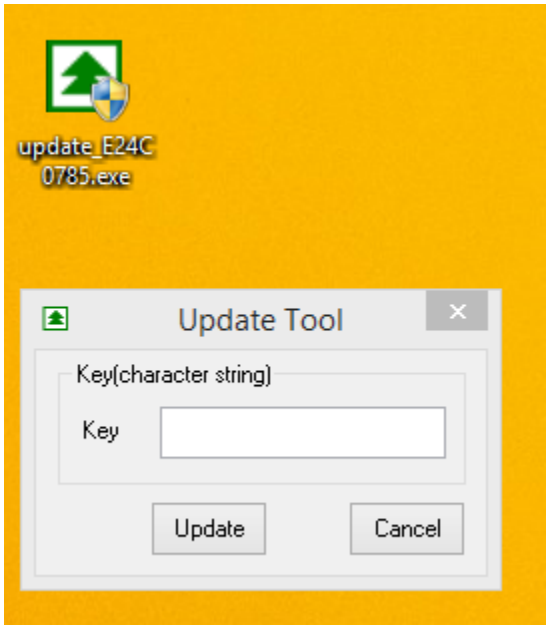


Fig. 30: Dongle Update Tool

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

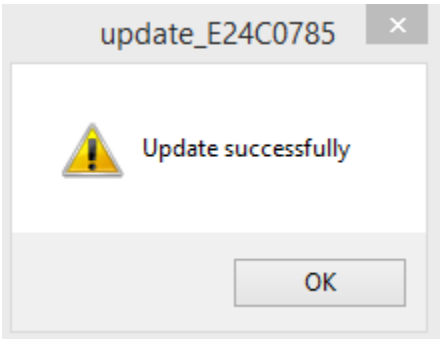


Fig. 31: Successful Dongle Update

Check Activation

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

Check Activation in the SDK

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

In the SDK,

- Go to `Window` > `Preferences` > `MicroEJ` ,
- Go to `Architectures` , `Platforms in workspace` or `Platforms` sub-menu and check that all items are now activated (green check).

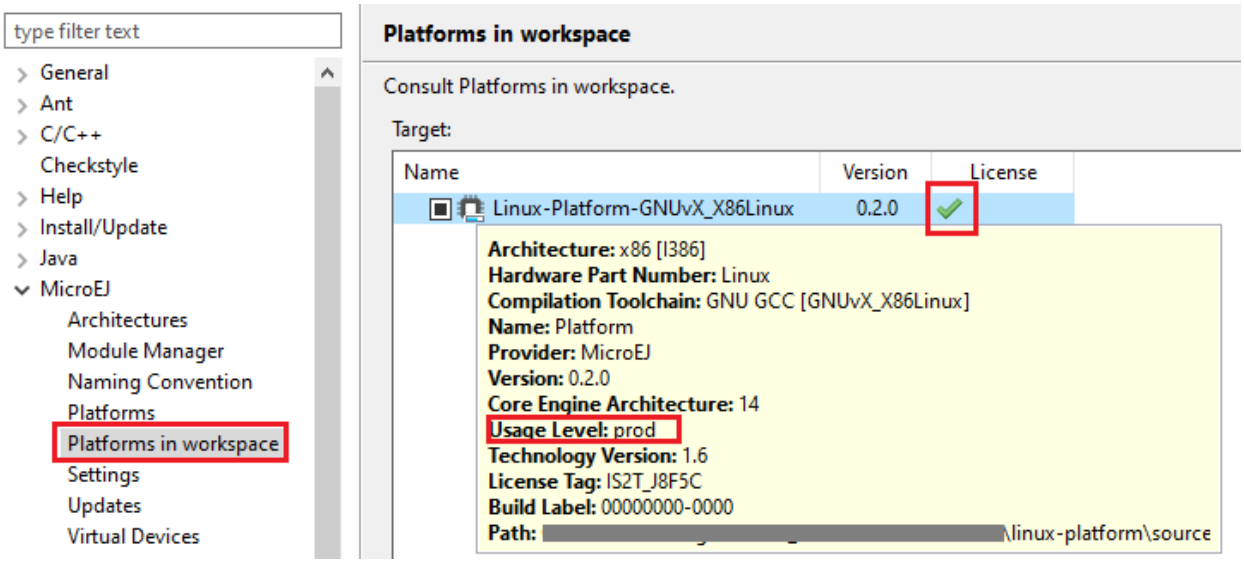


Fig. 32: License Status OK

If the license is still not recognized (red cross), check with the following command line tool to get more information.

Check Activation with the Command Line Tool

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

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

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

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

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

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

USB Dongle on GNU/Linux

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

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

    LABEL="usbdongle_start"

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

    LABEL="usbdongle_end"
```

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

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

USB Dongle with Docker on Linux

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

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

    LABEL="usbdongle_start"

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

    LABEL="usbdongle_end"
```

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

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

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

USB Dongle with WSL

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

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

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

Then, you need install usbipd-win on Windows from usbipd-win Github repository.

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

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

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

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

You then need to unplug and plug your dongle again before attaching the dongle to WSL from powershell:

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

The `<BUSID>` can be obtainted with the following powershell command:

```
usbipd wsl list
```

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

Troubleshooting

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

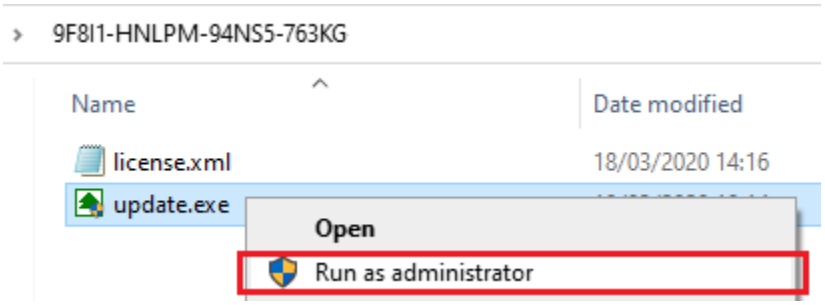
Windows Troubleshooting

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

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

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

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



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

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

VirtualBox Troubleshooting

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

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

WSL Troubleshooting

Check that your dongle is attached to WSL from Powershell:

```
usbipd wsl list
```

You should have a line saying `Attached - Ubuntu` :

```
PS C:\Users\sdkuser> usbipd.exe wsl list
BUSID  VID:PID    DEVICE                                                  ␣
↪STATE
2-1    096e:0006  USB Input Device                                        ␣
↪Attached - Ubuntu
2-6    0c45:6a10  Integrated Webcam                                       ␣
```

(continues on next page)

(continued from previous page)

```
↪Not attached
2-10   8087:0026  Intel(R) Wireless Bluetooth(R)                              ␣
↪Not attached
3-1    045e:0823  USB Input Device                                            ␣
↪Not attached
3-4    046d:c31c  USB Input Device                                            ␣
↪Not attached
```

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

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

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

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

Dongle not detected in the licenses screen

If the USB dongle is plugged and activated but not visible in the menu Window > Preferences > MicroEJ , please check that you have an active VEE Port in Window > Preferences > MicroEJ > Platforms in workspace .

Then, ensure that the VEE Port has been built in `prod` configuration, this can be checked with the architecture dependency inside the file `module.ivy` . If no VEE Ports are visible in your current workspace, please build a VEE Port configured to the `prod` mode and this should fix the issue.

Remote USB Dongle Connection

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

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

3.3 Standalone Application

3.3.1 Platform Import

A Platform is required to run a Standalone Application on the Simulator or build the Firmware binary for the target device.

The *VEE Porting Guide* describes how to create a Platform from scratch for any kind of device. In addition, MicroEJ Corp. provides Platforms for various development boards (see https://repository.microej.com/index.php?resource=JPF).

Platforms are distributed in two packages:

- **Source Platform**. The source files are imported into the workspace. This is the default case.
- **Binary Platform**. A `.jpf` file is imported into the *MicroEJ repository*. As of MicroEJ SDK `5.3.0`, this package is deprecated.

Source Platform Import

Import from Folder

This section applies when the Platform files are already available on a local folder. This is likely the case when the files are checked out from a Version Control System, such as a local git repository clone.

Note: If you are going to import a Platform from MicroEJ Github, you can follow the specific *GitHub Repositories* section instead (the projects will be automatically imported).

- Select File > Import... > General > Existing Projects into Workspace > Select root directory > Browse... .
- Select the root directory. The wizard will automatically discover projects to import.
- Click on the Finish button.

Import from Zip File

This section applies when the Platform files are packaged in a `.zip` file.

- Select File > Import... > General > Existing Projects into Workspace > Select archive file > Browse... .
- Select the zip of the project (e.g., `x.zip`). The wizard will automatically discover projects to import.
- Click on the Finish button.

Platform Build

Platforms are usually shared with only the Platform configuration files. Once the projects are imported, follow the platform-specific documentation to build the Platform.

Once imported or built, a Platform project should be available as follows:

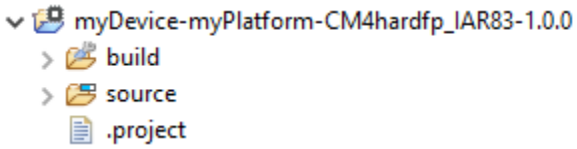


Fig. 33: Platform Project

The `source` folder contains the Platform content which can be set to the `target.platform.dir` option.

Binary Platform Import

After downloading the Platform `.jpf` file, launch MicroEJ SDK and follow these steps to import the Platform:

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

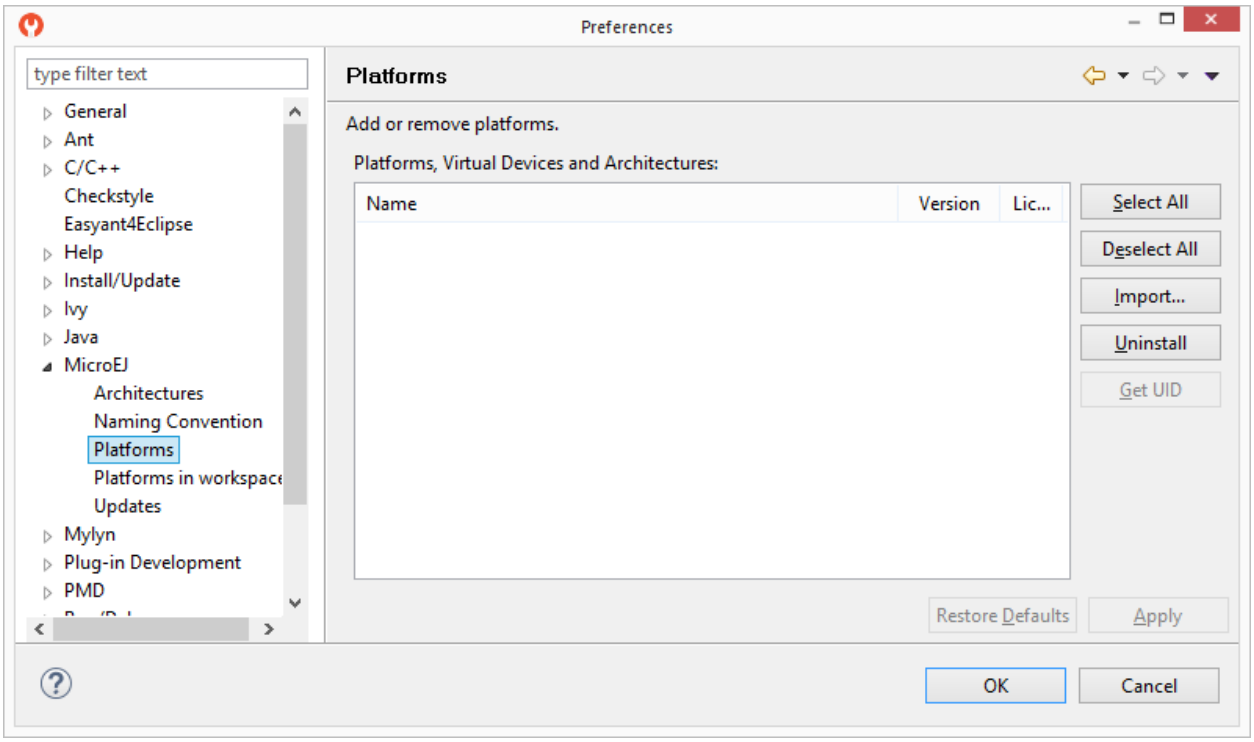


Fig. 34: Platform Import

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

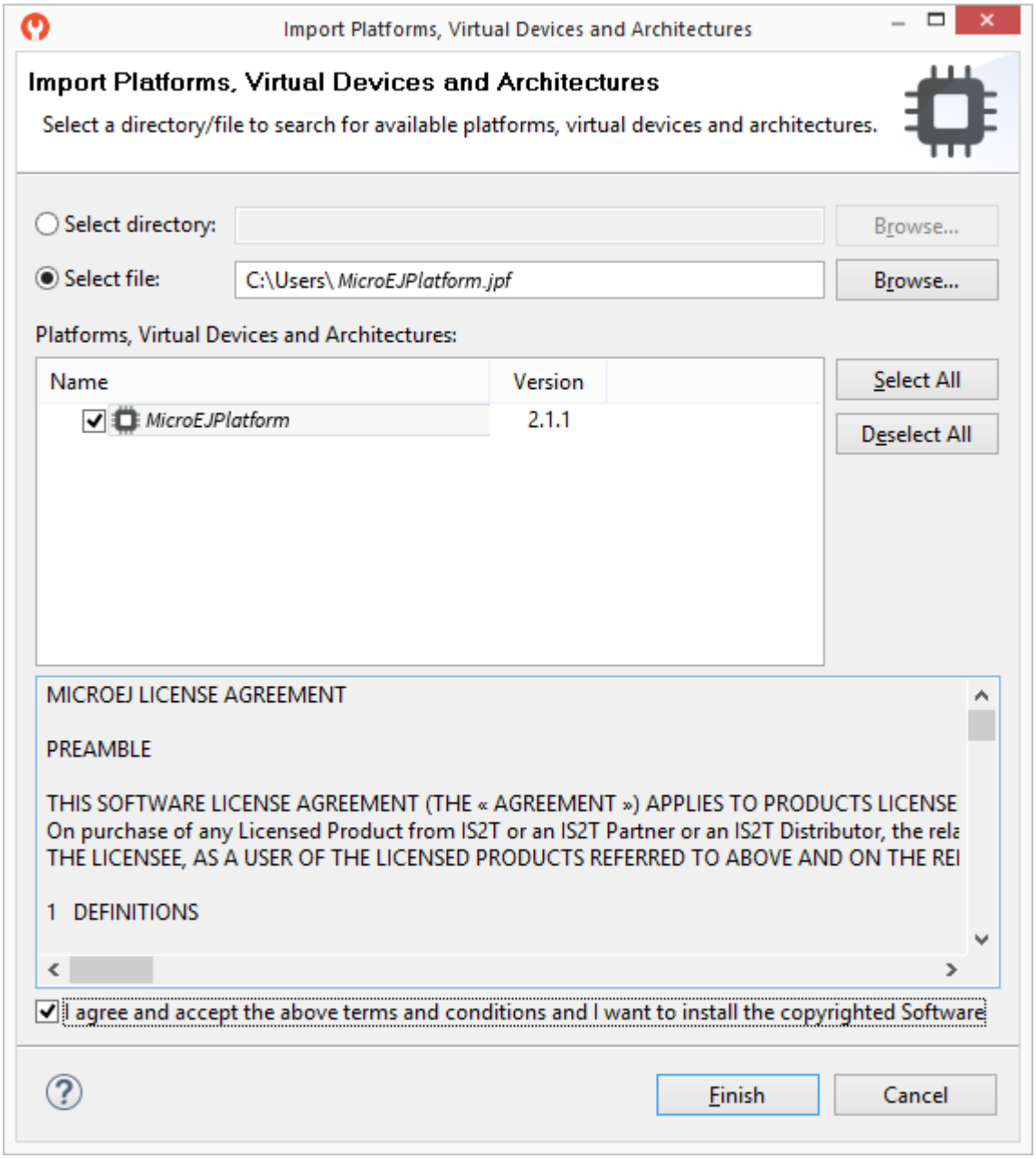


Fig. 35: Platform Selection

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

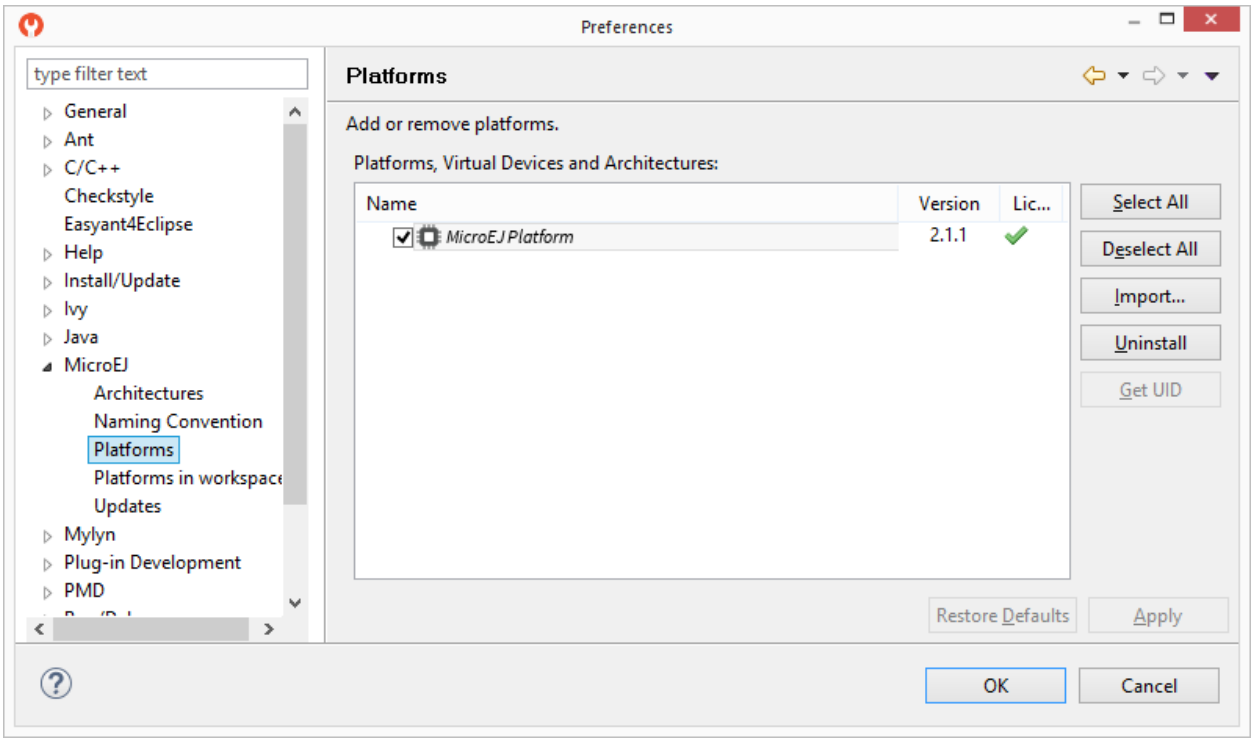


Fig. 36: Platform List

3.3.2 Build and Run an Application

Create a MicroEJ Standalone Application

Note: This section is releated to the version 5 and lower of the SDK. If you use the SDK 6, please refer to the page *Create a Project*.

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

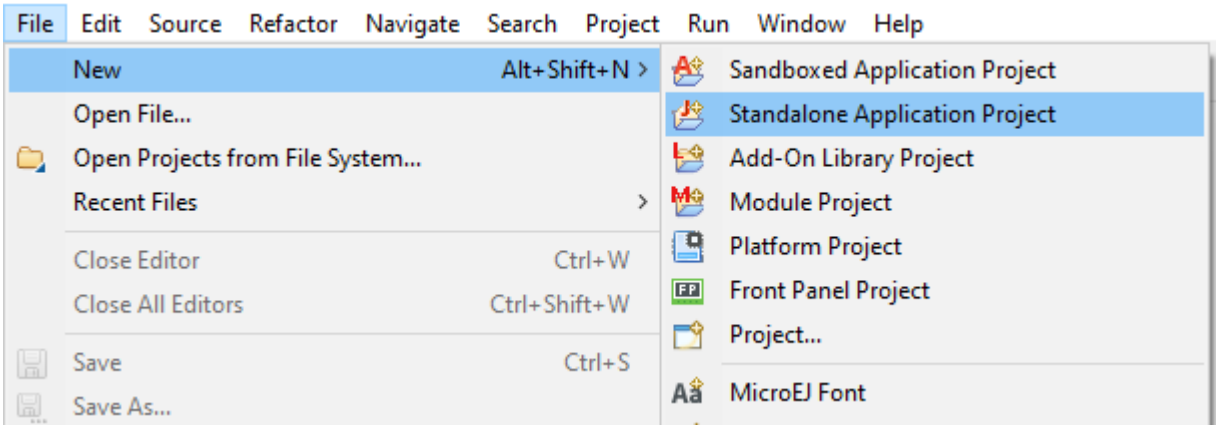


Fig. 37: New MicroEJ Standalone Application Project

- Fill in the Application template fields, the project name field will automatically duplicate in the following fields. For this tutorial, the project name is `hello`. Click on Finish. A template project is automatically created and ready to use, this project already contains all folders wherein developers need to put content:
  - `src/main/java`: Folder for future sources
  - `src/main/resources`: Folder for future resources (raw resources, images, fonts, nls)
  - `module.ivy`: *Module description file*, dependencies description for the current project
- A `Main` class already exists in the package `com.mycompany` and prints "Hello World!":

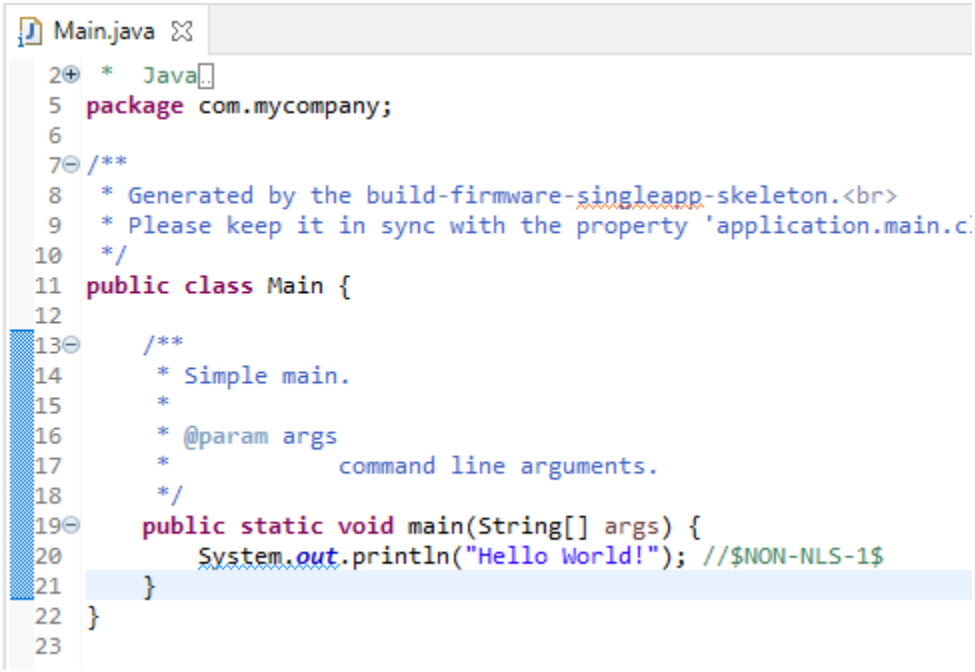


Fig. 38: MicroEJ Application Content

The main Application is now ready to be executed. See next sections.

Run on the Simulator

Note: This section is releated to the version 5 and lower of the SDK. If you use the SDK 6, please refer to the page *Run on Simulator*.

Note: *A Platform must have been imported* to run the Application. If several Platforms have been imported, the target Platform can be selected in the *Launcher's Execution tab*.

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

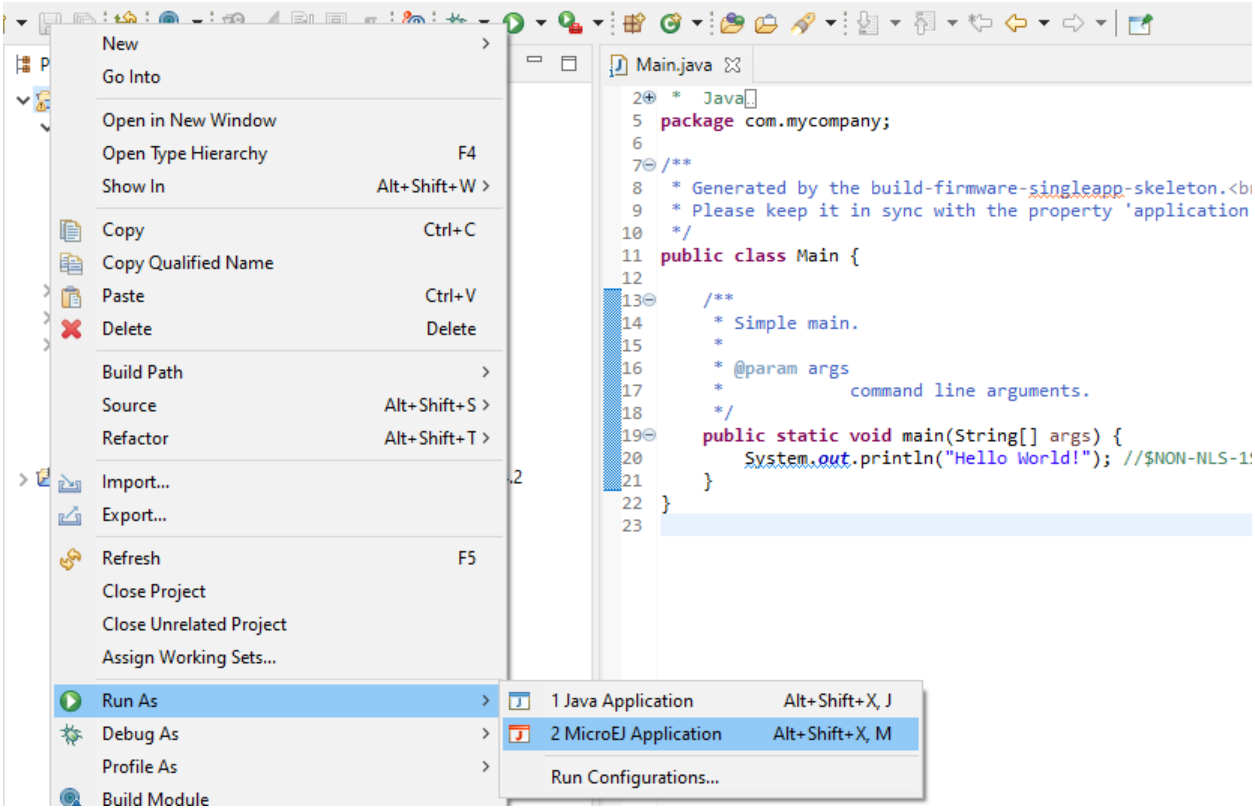


Fig. 39: MicroEJ Launcher Shortcut

MicroEJ SDK console will display Launch steps messages.

```
============== [ Initialization Stage ] ==============
============== [ Launching on Simulator ] ==============
Hello World!
============== [ Completed Successfully ] ==============

SUCCESS
```

Run on the Device

Build the Application

- Open the run dialog (`Run` > `Run Configurations...`).
- Select the `MicroEJ Application` > `Hello Main` that is created by the previous chapter.
- Open `Execution` tab and select `Execute on Device` .
- Set `Settings` checkbox to `Build & Deploy` .

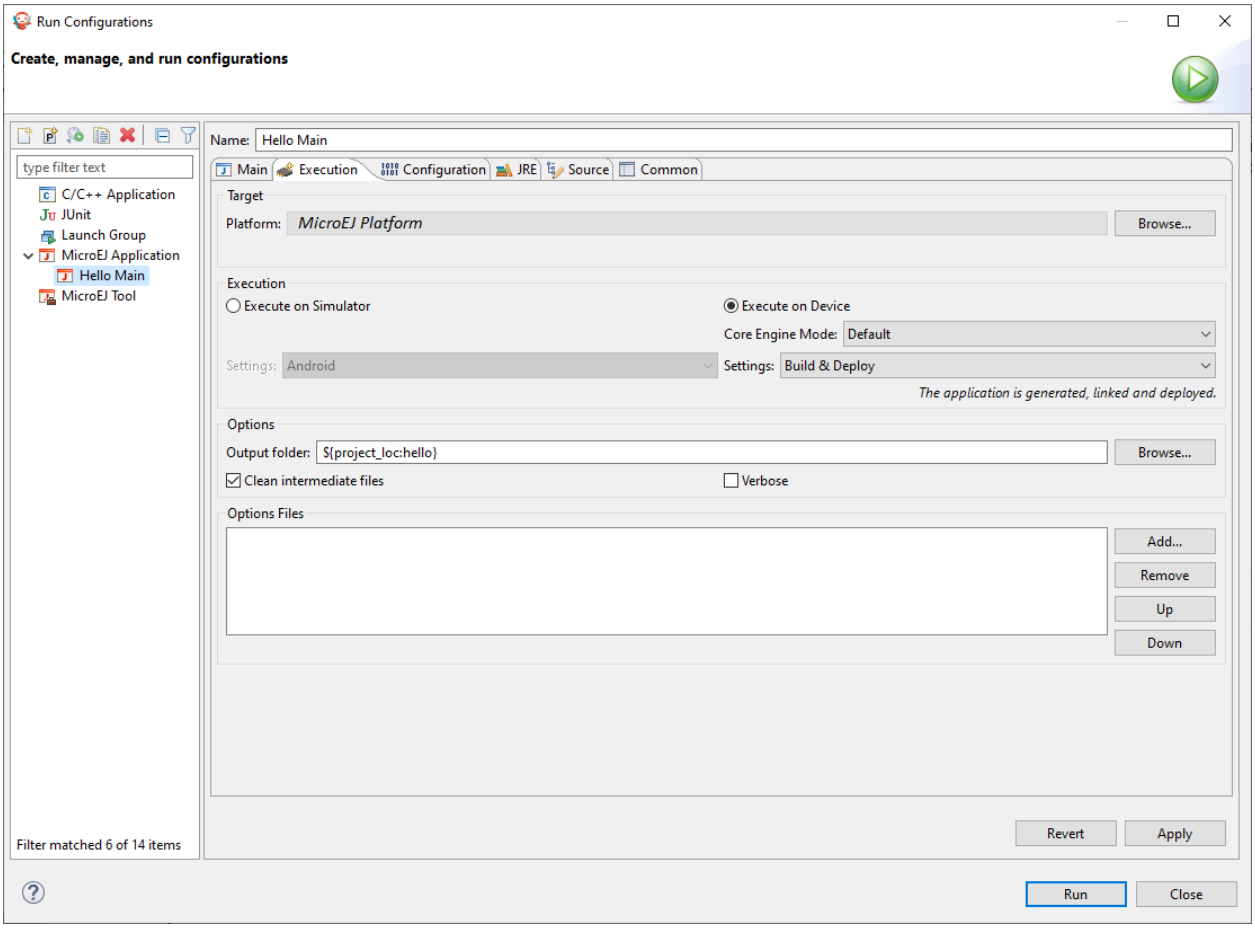


Fig. 40: Execution on Device

- Click `Run` : The Application is compiled and the Application, the runtime library and the header files are automatically deployed to the locations defined in your Platform *BSP connection* settings.

```
=============== [ Deployment ] ===============
MicroEJ files for the 3rd-party BSP project are generated to '<application-project>/<fully-
↪qualified-name-of-main-class>/platform'.
The MicroEJ application (microejapp.o) has been deployed to: '<path-to-deployment-location>'.
The MicroEJ platform library (microejruntime.a) has been deployed to: '<path-to-deployment-
↪location>'.
The MicroEJ platform header files (*.h) have been deployed to: '<path-to-deployment-location>
↪'.
=============== [ Completed Successfully ] ===============

SUCCESS
```

Build the Executable File

If your Platform has configured a *build script* file, the final Application linking can be triggered from the launcher:

- Open `Configuration` tab and select `Device` > `Deploy` . The options to deploy the Application, runtime library and header files have already been set in the previous step.
- Check `Execute the MicroEJ build script (build.bat) at a location known by the 3rd-party BSP project` .

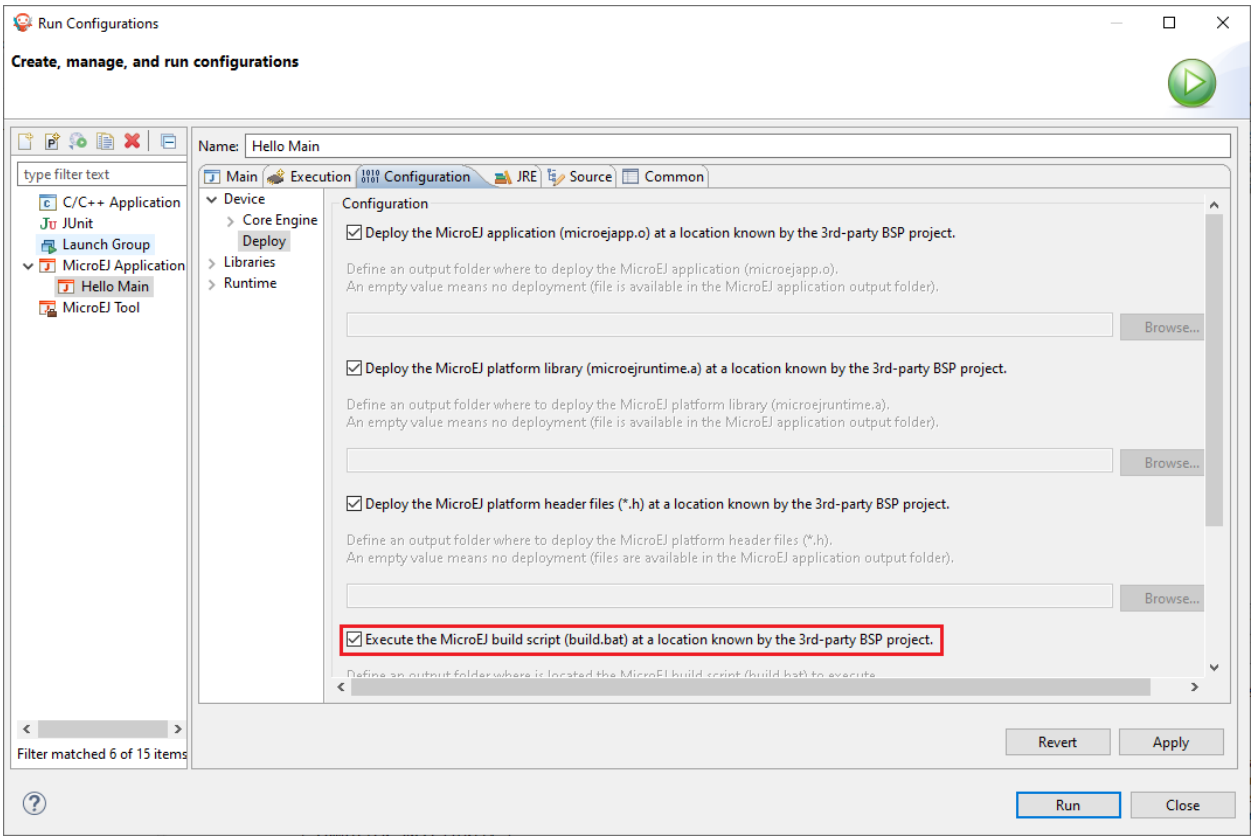


Fig. 41: BSP Connection Application Options

Note: The table *MicroEJ Application Options for BSP Connection* specifies the Application options that can be set depending on the BSP connection configured by the Platform.

- Click `Apply` and `Run` : the final executable `application.out` file is generated in the directory from where the script has been executed and can now be deployed on your Device using the appropriate flash tool.

3.3.3 MicroEJ Launch

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

Execution is done either on the Simulator or on the Device. In this latter case, it may depend on external tools such as target memory programming.

Main Tab

The Main tab allows you to set in order:

1. The main project of the application.
2. The main class of the application containing the main method.
3. Types required in your application that are not statically embedded from the main class entry point. Most required types are those that may be loaded dynamically by the application, using the Class.forName() method.
4. Binary resources that need to be embedded by the application. These are usually loaded by the application using the Class.getResourceAsStream() method.
5. Immutable objects' description files. See the *[BON 1.2] ESR documentation* for use of immutable objects.

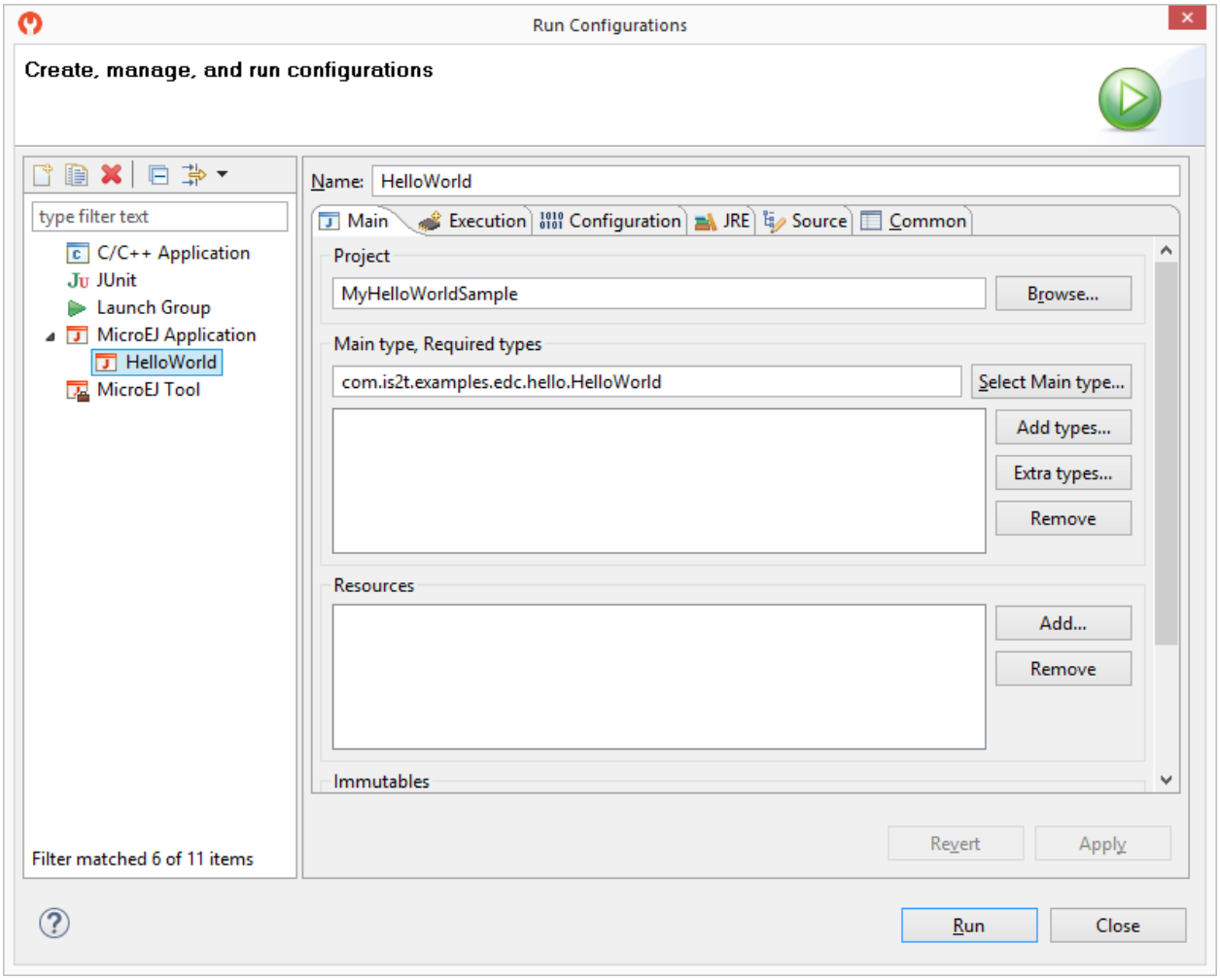


Fig. 42: MicroEJ Launch Application Main Tab

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).

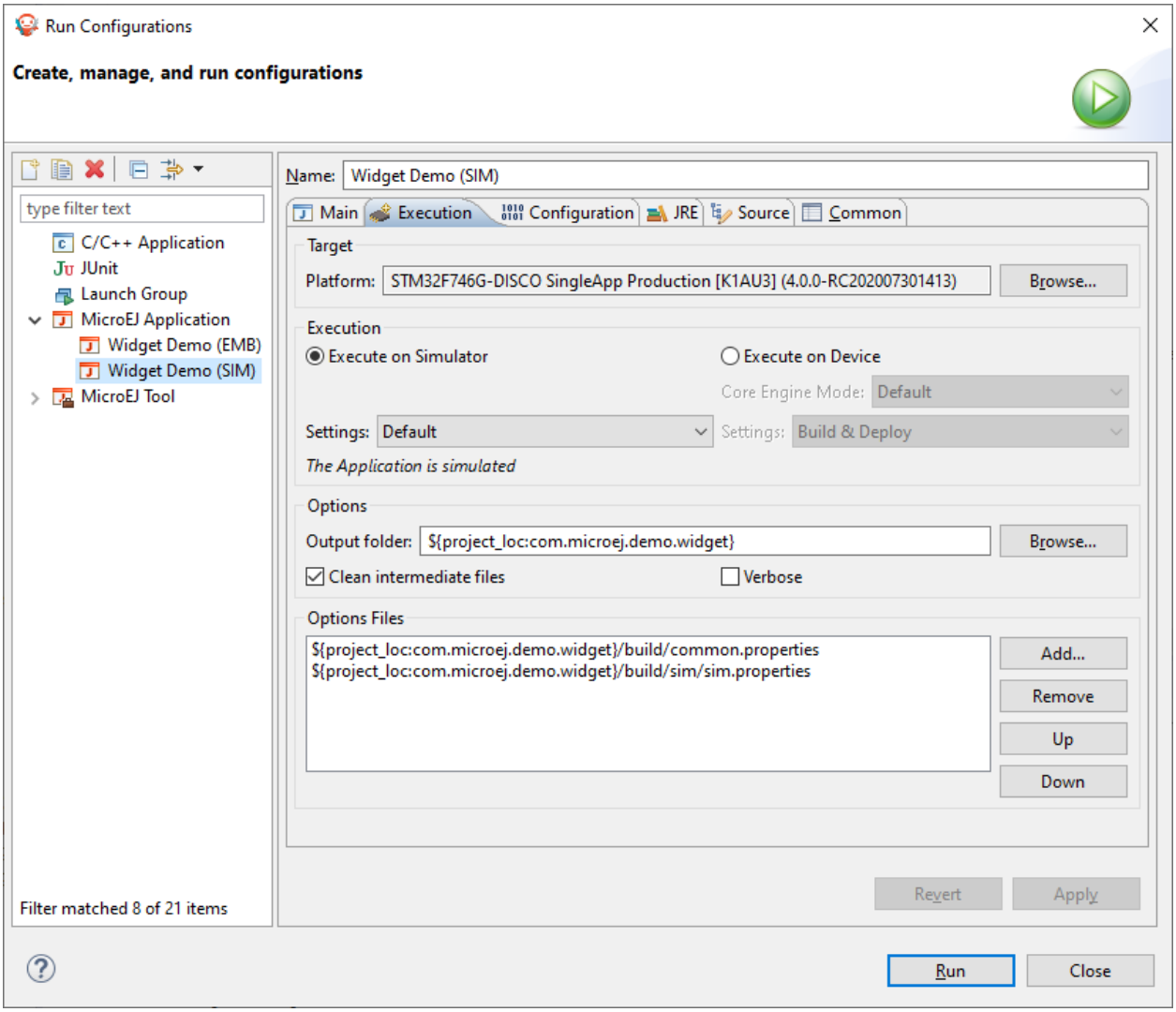


Fig. 43: MicroEJ Launch Application Execution Tab

Configuration Tab

The next tab is the Configuration tab. This tab shows the available *Application options*.

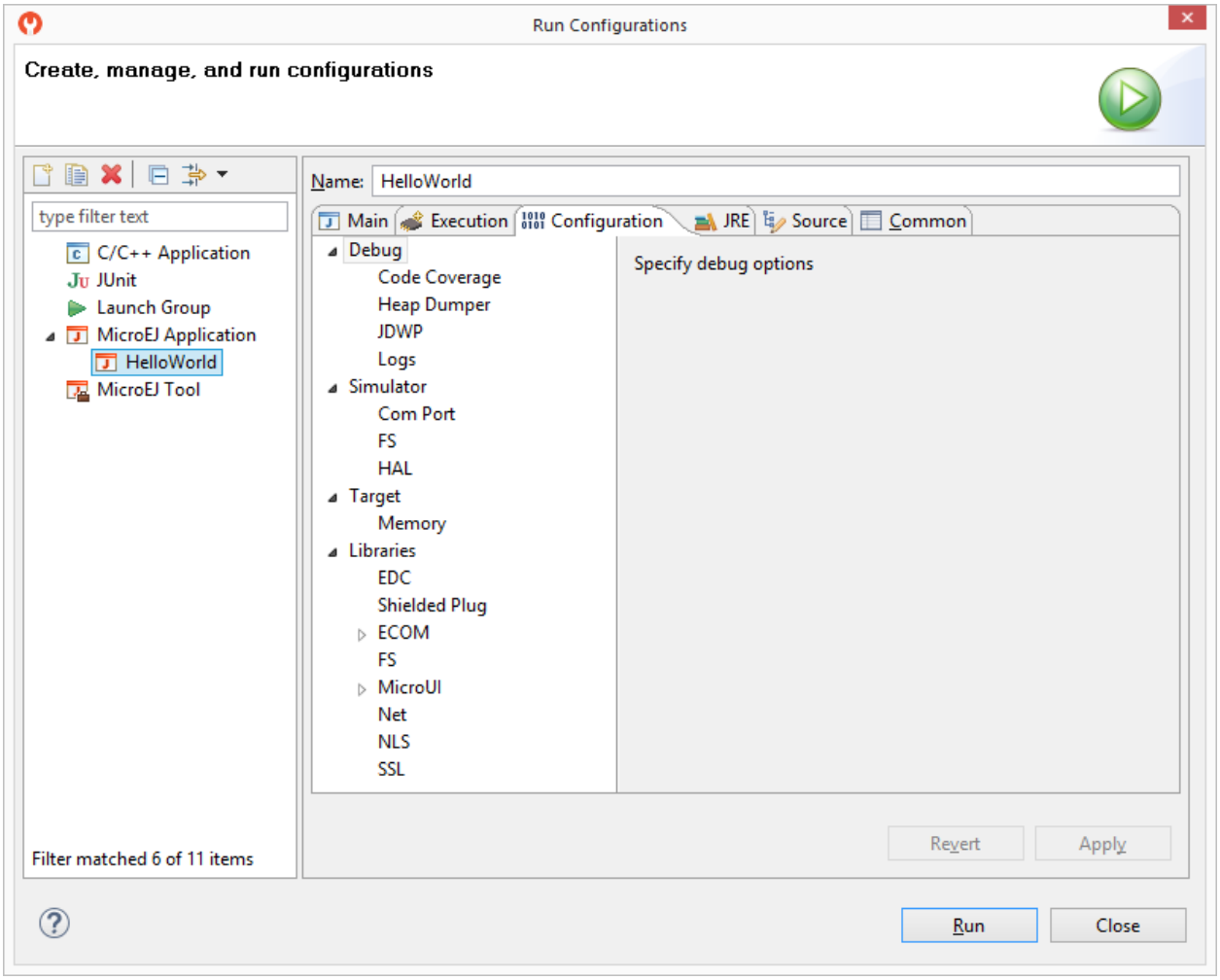
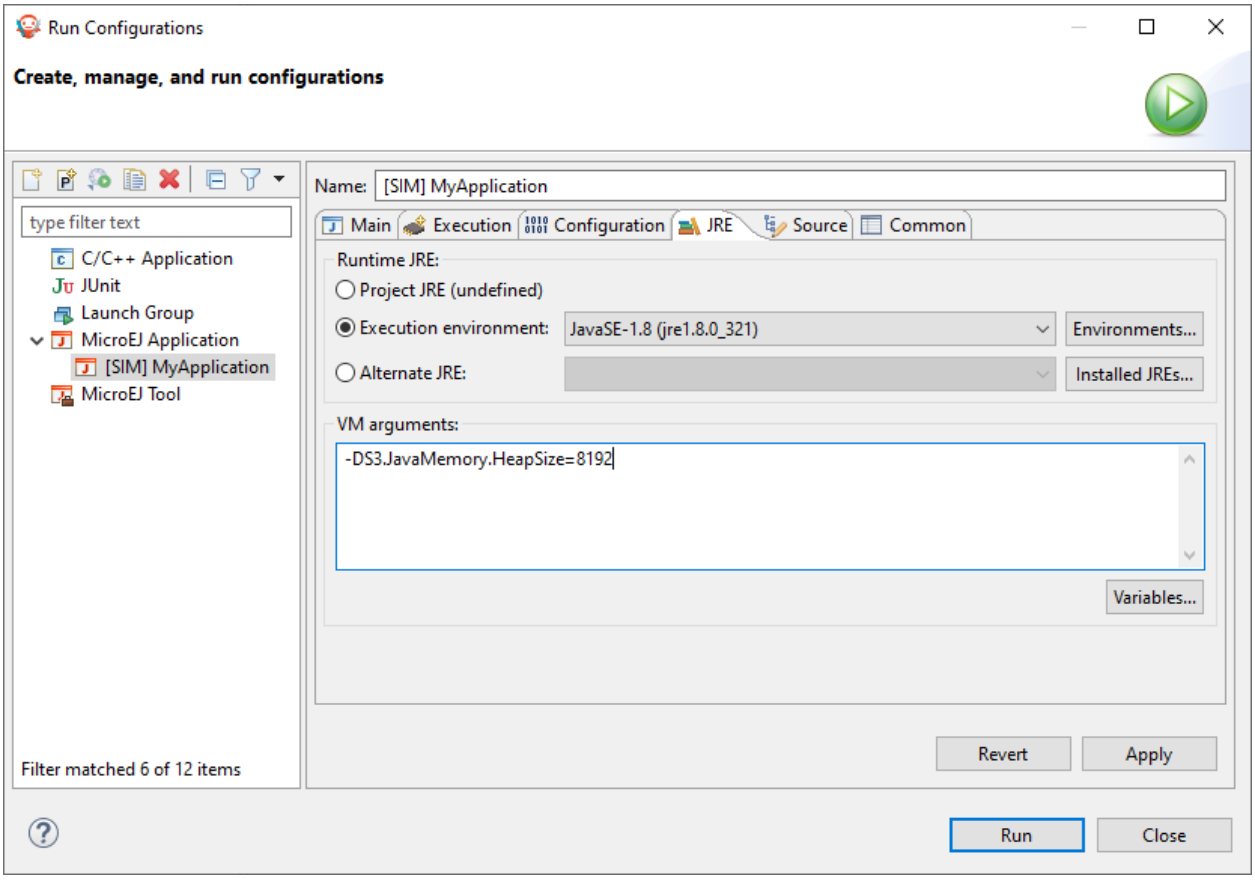


Fig. 44: Configuration Tab

JRE Tab

The next tab is the JRE tab. This tab allows you to configure the Java Runtime Environment used for running the underlying launch script. It does not configure the MicroEJ Application execution. The `VM Arguments` text field allows you to set vm-specific options, which are typically used to increase memory spaces:

- To modify heap space to 1024MB, set the `-Xmx1024M` option.
- To modify string space (also called PermGen space) to 256MB, set the `-XX:PermSize=256M -XX:MaxPermSize=256M` options.
- To set thread stack space to 512MB, set the `-Xss512M` option.
- To set an *advanced Application option*, declare a system property with the following pattern `-D[OPTION_KEY]>=[OPTION_VALUE]`



Source Tab

The next tab is the Source tab. By default, it is automatically configured to connect your Add-On Libraries sources dependencies. To connect your Platform Foundation Library sources, please refer to the section *Foundation Library Sources*.

Common Tab

The last tab is the Common tab. This is a default Eclipse tab that allows to configure your launch. Particularly, you can configure the *console encoding*. Refer to Eclipse help for more details on other available options.

A Standalone Application is a Java Application directly linked to the C code to produce an Executable. Such an application must define a main entry point (i.e., a class containing a `public static void main(String[])` method).

The *next chapters* explain how to build and run a Standalone Application.

3.4 Sandboxed Application

3.4.1 Create a First Application

Now that the *purposes of the Sandboxed Applications have been explained*, let's create a first application.

A Sandboxed Application project can be created in the SDK with the menu `File` > `New` > `Sandboxed Application Project` .

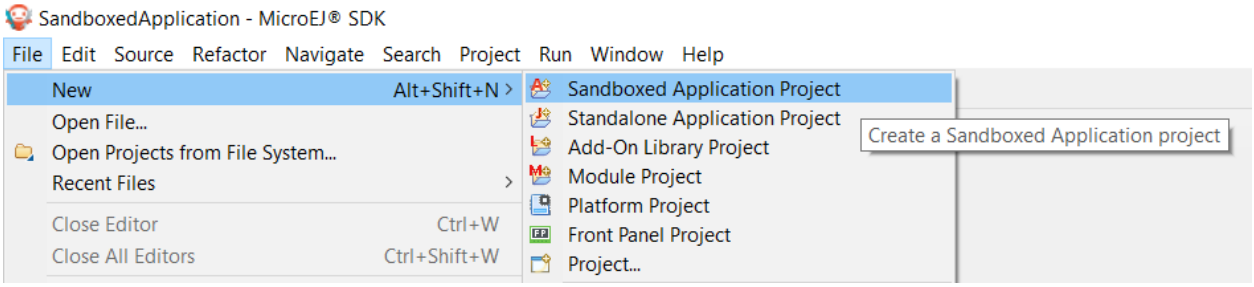


Fig. 45: Sandboxed Application Project Creation Menu

The project creation window is displayed:

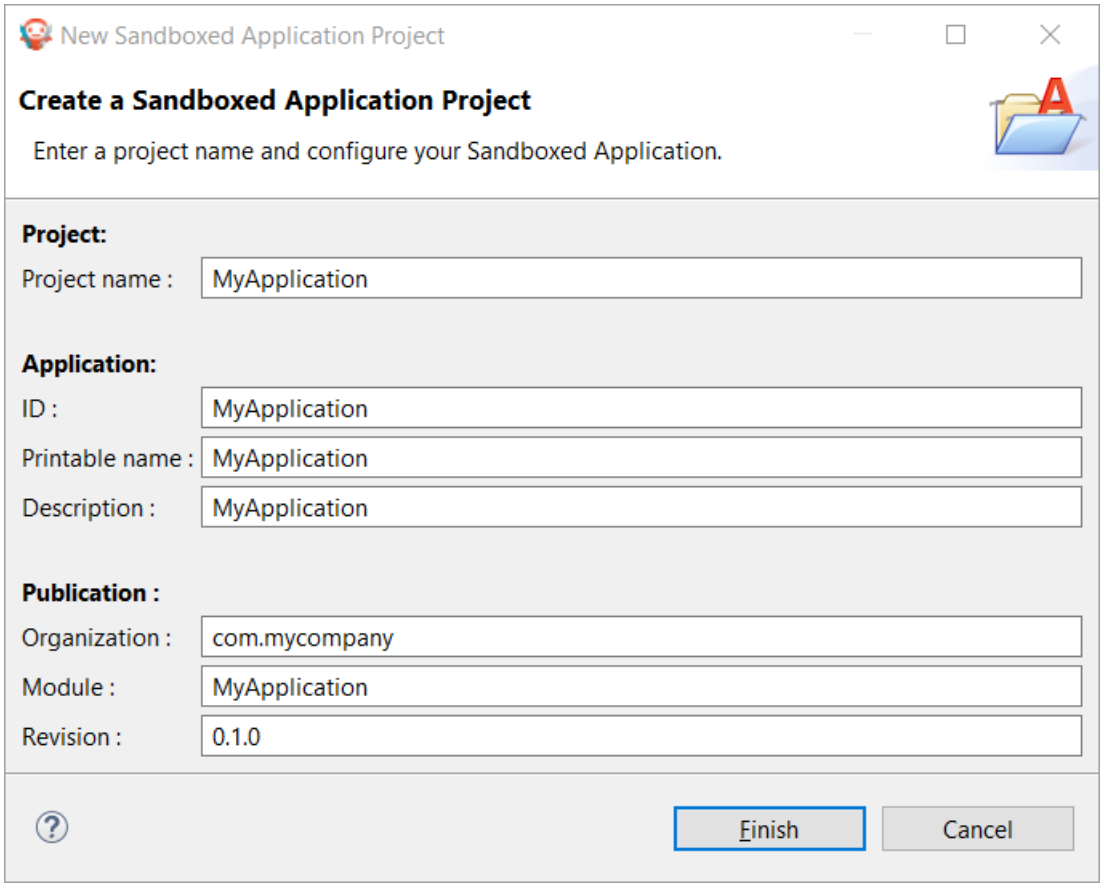


Fig. 46: Sandboxed Application Project Creation Form

Once the Application information are fulfilled and validated, the project is created with the following structure:

`src/main/java`
: Application Java sources;

`src/main/resources`
: Application resources (raw resources, images, fonts, nls);

`module.ivy`
: *Module description file*, containing build information and dependencies of the project.

The next sections describe the required files to have your first basic Application.

Entry Point

A Sandboxed Application must contain a class implementing the `ej.kf.FeatureEntryPoint` interface in the `src/main/java` folder:

```
package com.mycompany;

import ej.kf.FeatureEntryPoint;

public class MyApplication implements FeatureEntryPoint {

    @Override
    public void start() {
        System.out.println("Feature MyApplication started!");
    }

    @Override
    public void stop() {
        System.out.println("Feature MyApplication stopped!");
    }
}
```

This class is the entry point of the Application. The method `start` is called when the Application is started. It is considered as the main method of the Sandboxed Application. The method `stop` is called when the Application is stopped. Please refer to the *Sandboxed Application Lifecycle* chapter to learn more about the Applications lifecycle.

The `src/main/java` folder is also the place to add all the other Java classes of the Application.

Configuration

A Sandboxed Application project must contain a file with the `.kf` extension in the `src/main/resources` folder. This file contains the configuration of the Application. Here is an example:

```
name=MyApplication
entryPoint=com.mycompany.MyApplication
types=*
version=0.1.0
```

It contains the following properties:

- **name**: the name of the Application
- **entryPoint**: the Full Qualified Name of the class implementing `ej.kf.FeatureEntryPoint`

- **types**: this property defined the types included in the Application and must always be `*` (do not forget the space at the end)
- **version**: the version of the Application

SSL Certificate

A Sandboxed Application requires a certificate for identification. It must be located in the `src/main/resources` folder of the project. The project created by the SDK provides a sample certificate. This certificate is sufficient for testing, but it is recommended to provide your own.

Module Descriptor

The `module.ivy` file is the *Module description file* which contains the project information and declares all the libraries required by the Application. See *MicroEJ Module Manager* for more information.

The dependencies must contain at least a module containing the `ej.kf.FeatureEntryPoint` class, for example the KF library:

```
<dependency org="ej.api" name="kf" rev="1.6.1" />
```

3.4.2 Run on the Simulator

Note: This page is releated to the version 5 and lower of the SDK. If you use the SDK 6, please refer to the page *Run on Simulator*.

Once *a Sandboxed Application project has been created*, it can be tested on the Simulator.

The Simulator requires a Virtual Device to execute the Application. Please refer to the *Kernel Developer Guide* to learn how to get or create one.

From the SDK

In order to test a Sandboxed Application in the SDK, the first thing to do is to import the Virtual Device of the Multi-Sandbox Executable:

- go to Window > Preferences > MicroEJ > Virtual Devices
- click on Import...
- the Virtual Device can be provided as a folder or as a `.vde` file, select the adequate format and the Virtual Device resource
- check the License checkbox to accept it
- click on Finish

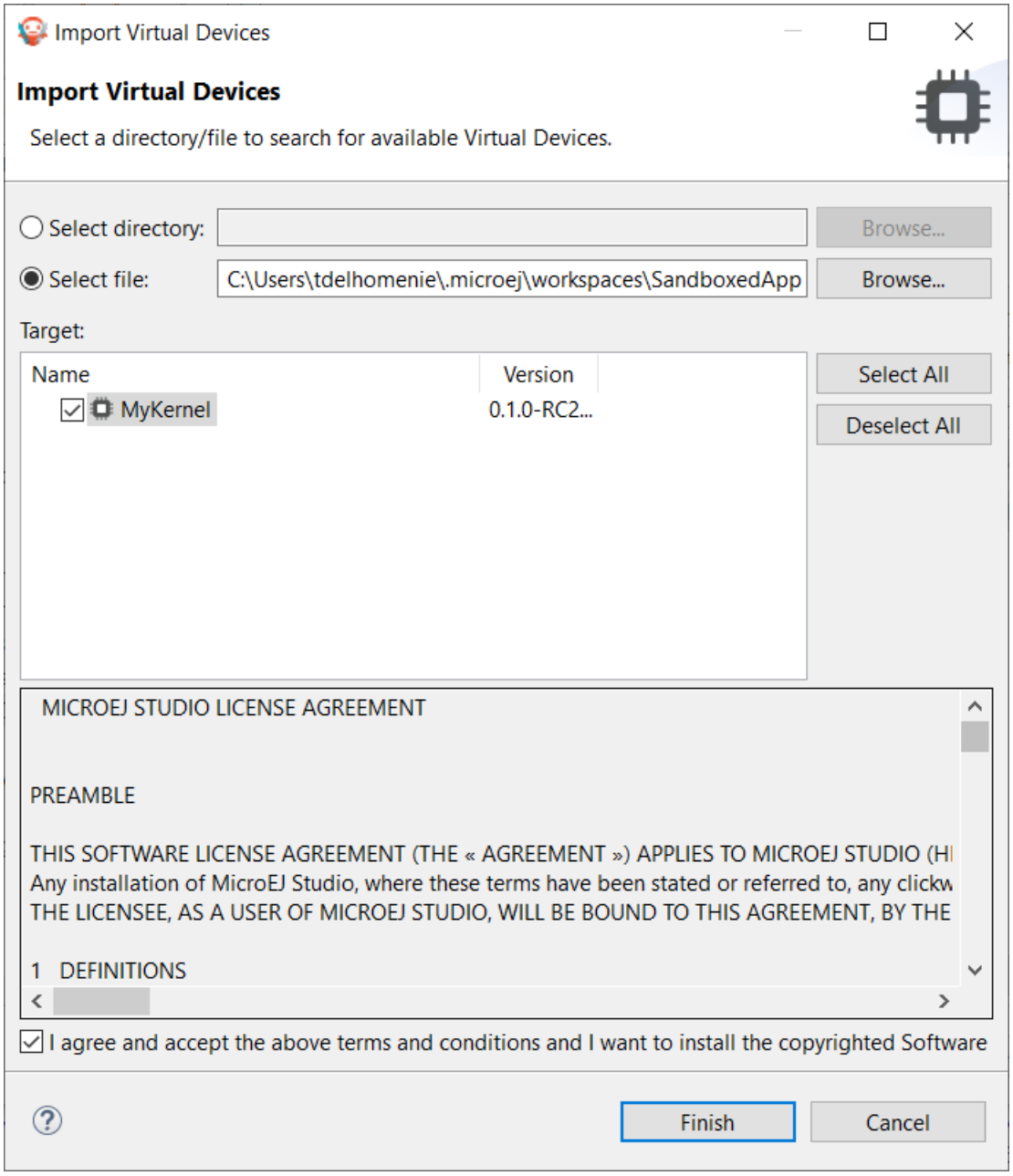


Fig. 47: Virtual Device Import

Now the Application can be executed by right-clicking on its project, then clicking on `Run As` > `MicroEJ Application` .

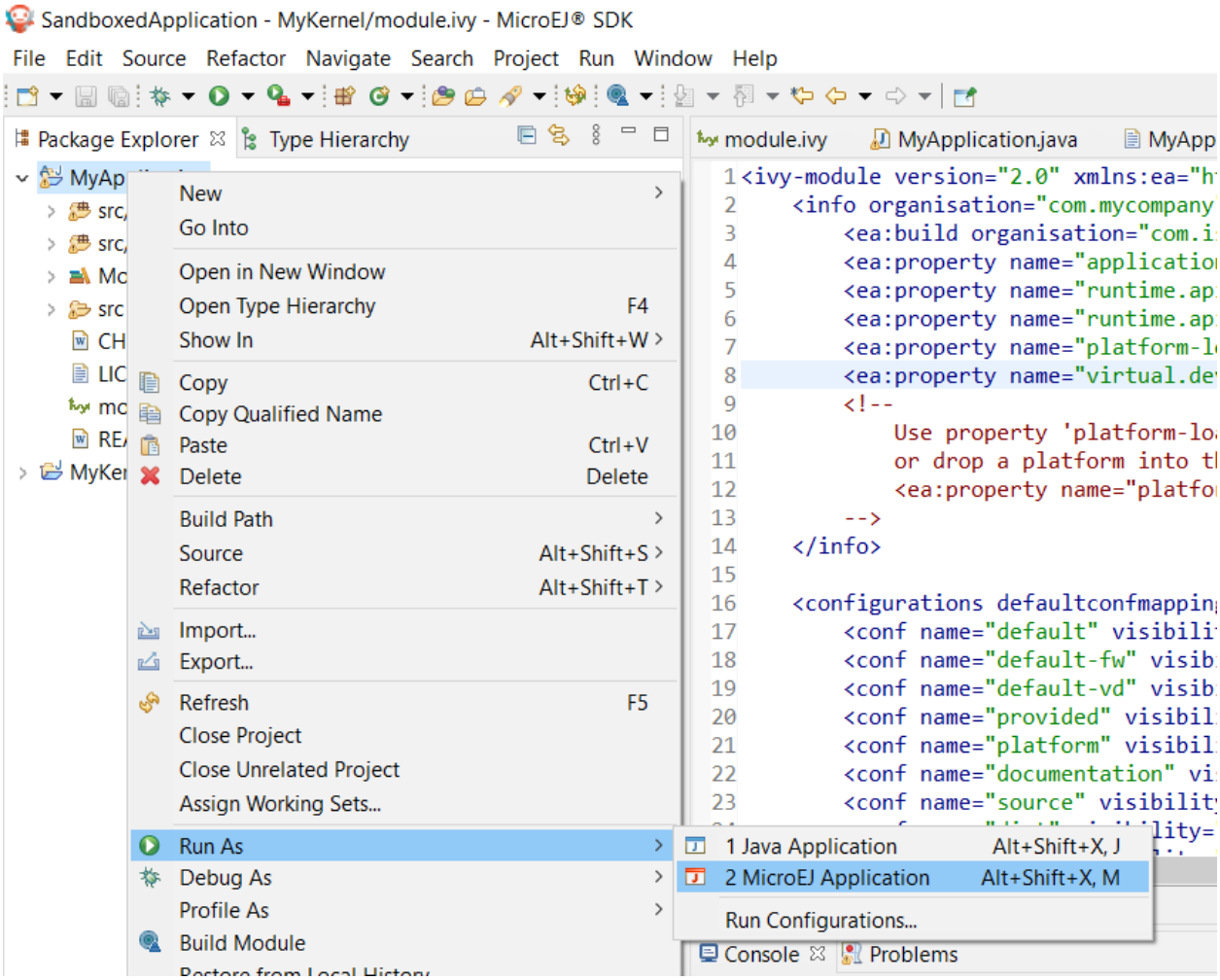


Fig. 48: Sandboxed Application Run

If there is only one Virtual Device imported in the SDK, it is automatically used to execute the Application. Otherwise, you have to select the one you want to use.

With the Application created in the section *Create a First Application*, the output should be:

```
============== [ Initialization Stage ] ==============
============== [ Converting fonts ] ==============
============== [ Converting images ] ==============
============== [ Launching on Simulator ] ==============
KERNEL Hello World!
=> Starting Feature MyApplication
Feature MyApplication started!
============== [ Completed Successfully ] ==============

SUCCESS
```

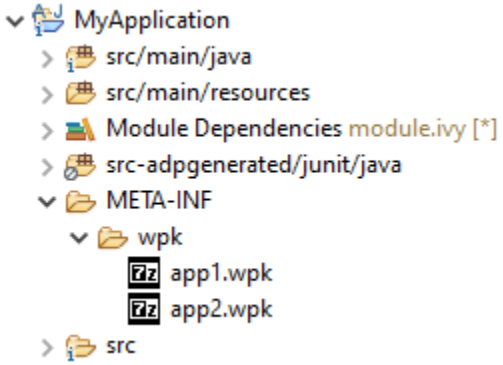
Run Multiple Sandboxed Applications

It is possible to execute additional Sandboxed Applications besides the main Sandboxed Application project. This is typically useful when you want to test the integration of a Sandboxed Application that communicates with an other one, for example through a *Shared Interface*.

The additional Sandboxed Applications must have been previously built in its binary format (WPK, see *Remote Deployment* section). Then, to include them:

- Select the Sandboxed Application project,
- Create the `META-INF/wpk` folders,
- Drop any `*.wpk` files in the `META-INF/wpk` folder.

Your Sandboxed Application project shall look like:



Now, when launching the Sandboxed Application project, these additional Sandboxed Applications will also be executed on the Virtual Device.

From the Command Line Interface

An Sandboxed Application can also be launched on the Simulator via the Command Line Interface. Before continuing, make sure *the Command Line Interface is installed and correctly configured*.

In your favorite terminal application, go to the root folder of the Application and execute the following commands:

```
mmm build
mmm run -Dplatform-loader.target.platform.file=/path/to/the/virtual-device.vde
```

With the Application created in the section *Create a First Application*, the output should be:

```
MicroEJ Simulator is being launched. Relax and enjoy...
=============== [ Initializing Easyant ] ===============
=============== [ Resolving and retrieving dependencies ] ===============
=============== [ Compiling sources ] ===============
=============== [ Loading platform ] ===============
=============== [ Initialization Stage ] ===============
=============== [ Converting fonts ] ===============
=============== [ Converting images ] ===============
```

(continues on next page)

(continued from previous page)

```
=============== [ Launching on Simulator ] ===============
KERNEL Hello World!
=> Starting Feature MyApplication
Feature MyApplication started!
=============== [ Completed Successfully ] ===============

SUCCESS
```

Note that the Virtual Device location can also be configured in the `module.ivy` file of the Sandboxed Application project:

```
<ea:property name="platform-loader.target.platform.file" value="/path/to/the/virtual-device.
↪vde"/>
```

The Virtual Device can also be provided differently, for example from a dependency in the `module.ivy` file. Refer to the *Platform Selection* section for the list of available capabilities.

3.4.3 Run on the Device

The deployment of a Sandboxed Application on a device depends on the Kernel implementation. We can group them in two categories:

- Local Deployment: the device is connected to the developer's computer, the SDK builds the `.fo` from the workspace project classes and transfers it on the device (recommended during application development).
- Remote Deployment: the Application is built, then the device connects a Repository where the Application is stored, and deploys it over the air using a device management system (production deployment).

In both cases, deploying a Sandboxed Application requires that a Multi-Sandbox Executable is running on the device. Please refer to the *Kernel Developer Guide* to learn how to build it or browse the Resources Repository for Multi-Sandbox demo Firmware available for popular hardware evaluation kits.

Local Deployment

Deploying an Application on a device locally is the easiest way to test it since it only requires:

- the Application project sources imported in the SDK,
- the device programmed with a Multi-Sandbox Executable that provides the Local Deployment capability (you can browse the Resources Repository for available demos of such Multi-Sandbox Executable),
- the device connected to the developer's computer either on the same network (LAN) or using a serial wire, depending on the Firmware capabilities.

If these prerequisites are fulfilled:

- duplicate the Run Configuration created in the chapter *Run on the Simulator*,

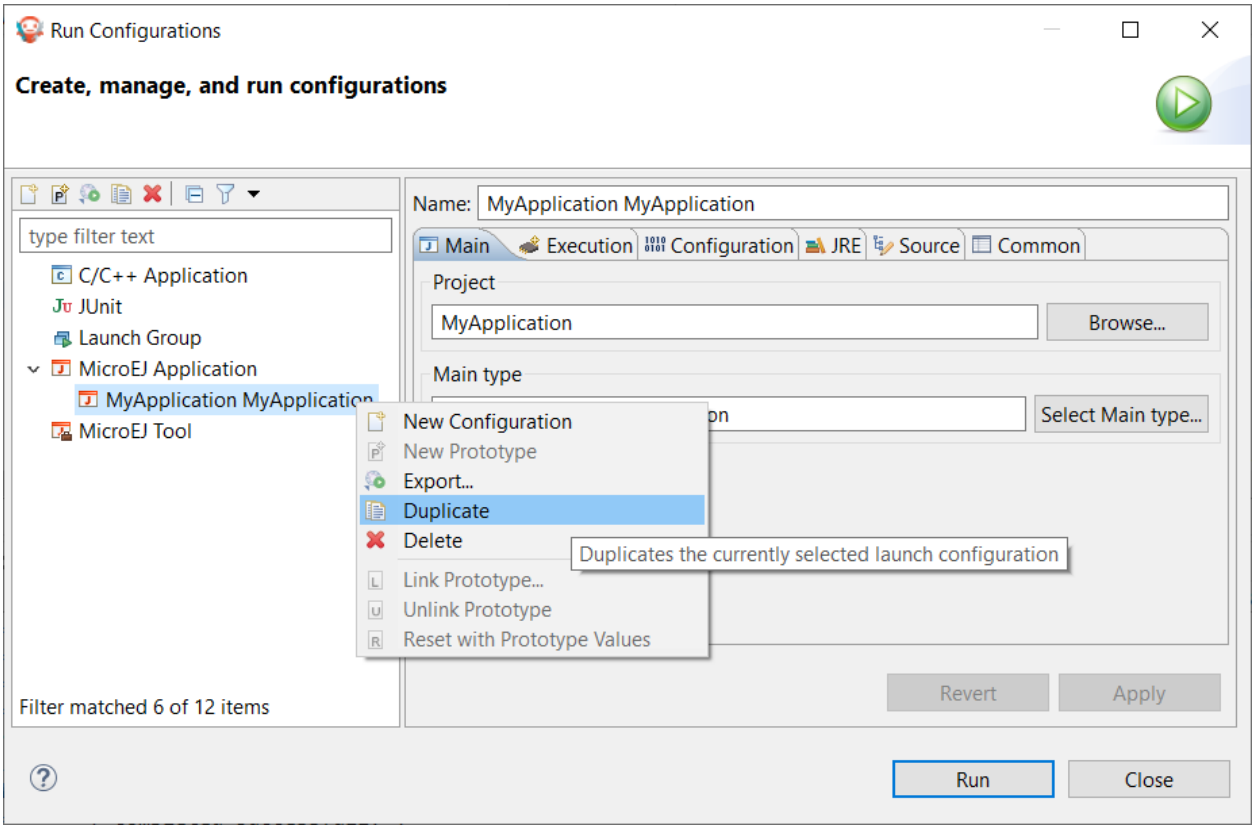


Fig. 49: Duplicate Run Configuration

- rename the duplicated Run Configuration, for example by prefixing by `(Local)`,
- in the Execution tab, modify the Execution mode to Execute on Device,

> **Note:** The selected Platform must be a Virtual Device (VDE) including the Local Deployment capability, not a VEE Port.

- select the option Local Deployment (Socket) in the Settings list. Note that depending on the device capability, the virtual device may implement a local deployment over a Comm Port.

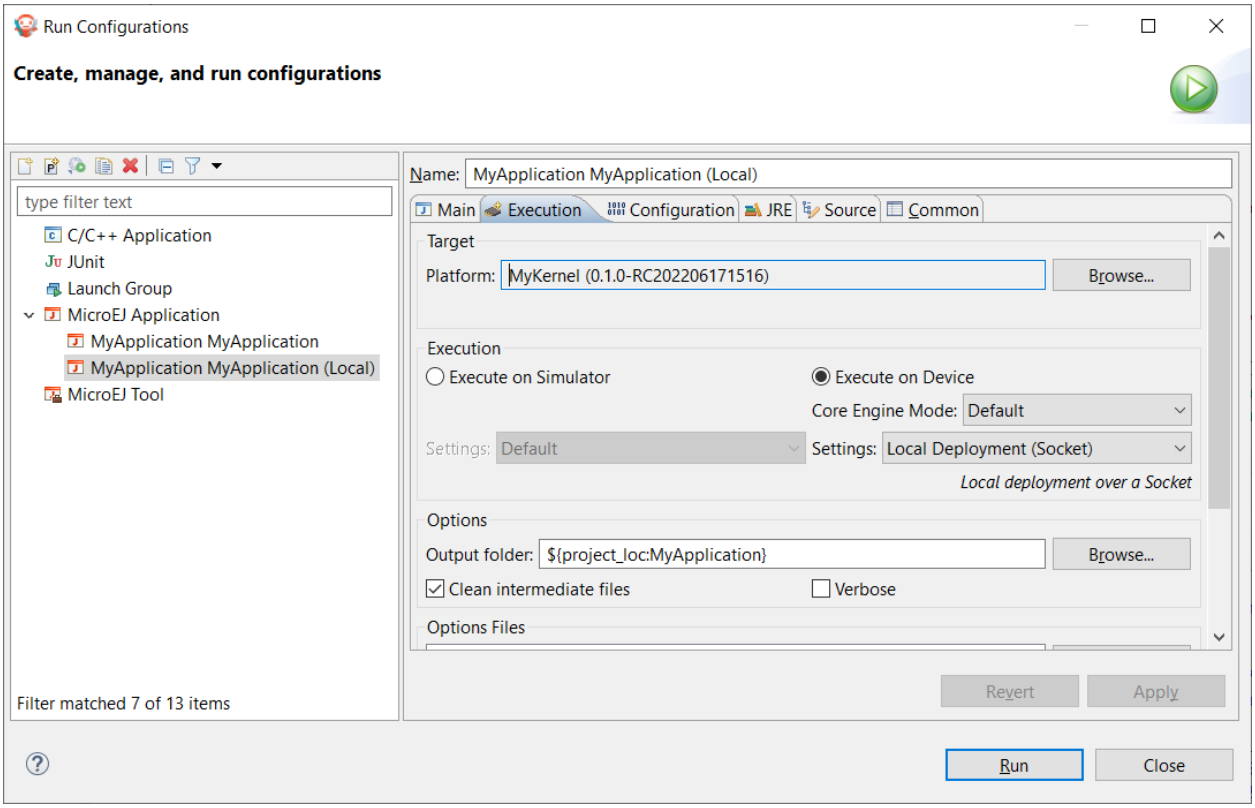


Fig. 50: Configure Run Configuration

- go to the `Configuration` tab,
- select the item `Local Deployment (Socket)` ,
- set the IP address of the device in the `Host` field,
- click on the `Run` button to deploy the Application on the board.

The Console output should be:

```
=============== [ Initialization Stage ] ===============
=============== [ Converting fonts ] ===============
=============== [ Converting images ] ===============
=============== [ Build Application ] ===============
=============== [ Completed Successfully ] ===============
=============== [ Deploy on 192.168.0.7:4000 ] ===============
=============== [ Completed Successfully ] ===============

SUCCESS
```

The Application is deployed on the device and automatically started. You can use a Serial terminal to get the traces of the Application:

```
KERNEL Hello World!
=> Starting Feature MyApplication
Feature MyApplication started!
```

Remote Deployment

Remote Deployment requires building and publishing the Sandboxed Application module. To do so, in the SDK, right-click on the Sandboxed Application project and click on Build Module .

The build process will display messages in the console, ending up the following message:

```
[echo] project hello published locally with version 0.1.0-RC201907091602

BUILD SUCCESSFUL

Total time: 1 minute 6 seconds
```

The files produced by the build process are located in a dedicated `target~/artifacts` folder in the project and is published to the target module repository declared in *MicroEJ Module Manager settings file*.

The file that ends with `.wpk` (the WPK file) is a portable file that contains all necessary binary data to build `.fo` files on any compatible Multi-Sandbox Executable. Then, the WPK file can be published to a MICROEJ FORGE instance. Please contact *our support team* if you want to get more information on MICROEJ FORGE and automated Applications deployment through a device management system.

A Sandboxed Application is an Application that can run over a Multi-Sandbox Executable.

The Application development flow requires the following elements:

- a Virtual Device, a software package including the resources and tools required for building and testing an application for a specific device. A Virtual Device will simulate all capabilities of the corresponding hardware board:
  - Computation and Memory
  - Communication channels (e.g., Network, USB . . .)
  - Display
  - User interaction
- an hardware device that has been previously programmed with a Multi-Sandbox Executable. Virtual Devices and Multi-Sandbox Executable share the same version (there is a 1:1 mapping).

The *next chapters* explain how to create, test and publish Sandboxed Applications.

3.5 Module Repository

A module repository is a module that bundles a set of modules in a portable ZIP file. It is a tree structure where modules organizations and names are mapped to folders.

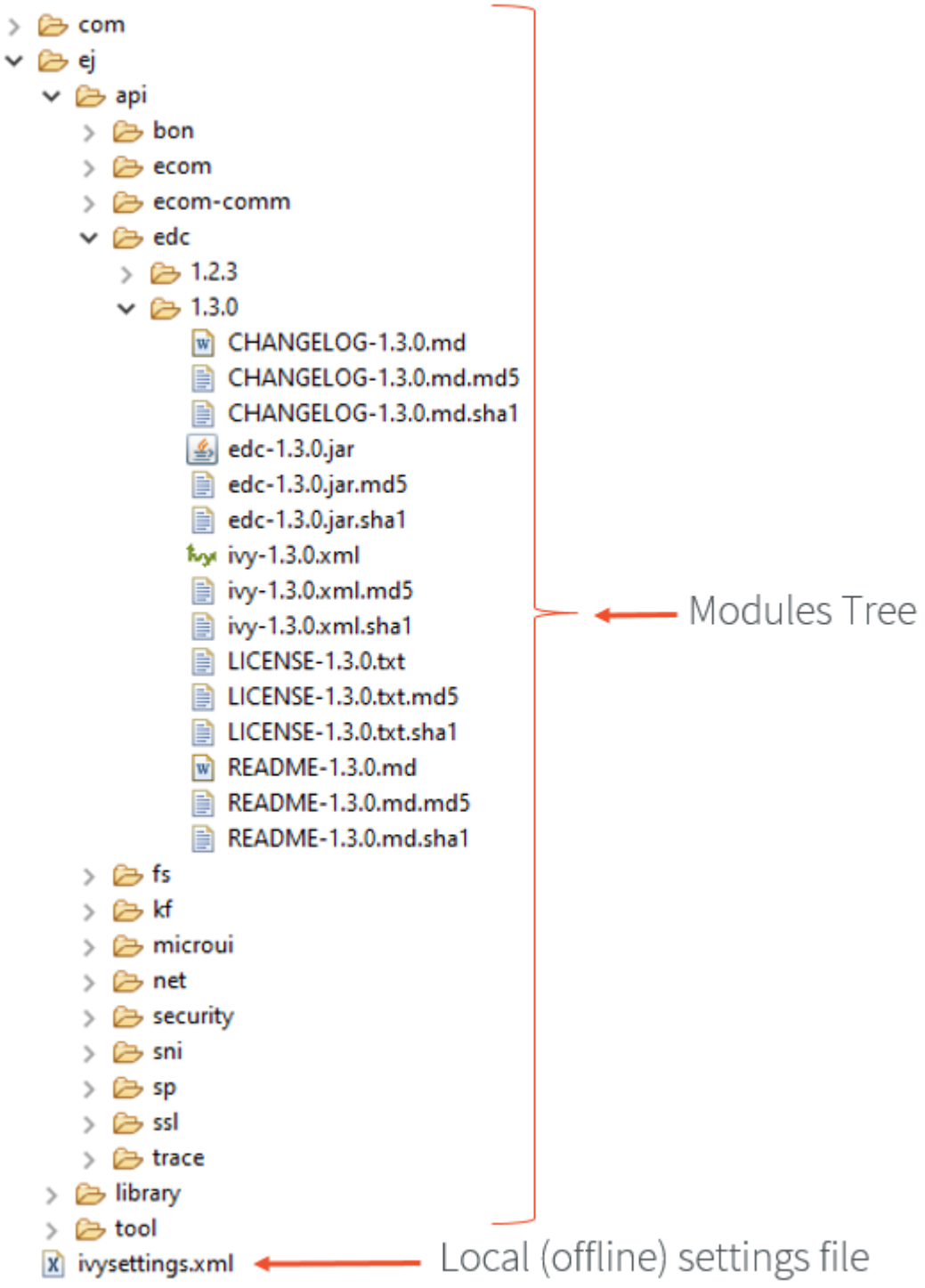


Fig. 51: Example of MicroEJ Module Repository Tree

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.

3.5.1 Create a Repository Project

In the SDK, first create a new *module project* using the `artifact-repository` skeleton.

- The `ivysettings.xml` *settings file* describes how to import the modules of this repository when it is extracted locally on file system. This file will be packaged at the root of the zip file and does not need to be modified.
- The `module.ivy` file describes how to build repository and lists the module dependencies that will be included in this repository.

3.5.2 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.

3.5.3 Configure Consistency Check

The module repository consistency check consists in verifying that each declared module can be imported using the settings file provided by the repository. Especially, it ensures that all module transitive dependencies are also available.

It is enabled by default to avoid further issues for repository users such as *Unresolved Dependency*. This is done by the following option:

```
<ea:property name="skip.retrieve.checker" value="false"/>
```

Moreover, to ensure the repository will be compliant with the *MMM specification*, add the following option:

```
<ea:property name="bar.check.as.v2.module" value="true"/>
```

3.5.4 Advanced Options

There are other advanced options that do not need to be modified by default. These options are described in the `module.ivy` generated by the skeleton.

See also *Module Repository* for more details.

3.5.5 Include Modules

Modules bundled into the module repository must be declared in the `dependencies` element of the `module.ivy` file.

Include a Single Module

To add a module, declare the *module dependency* using the `artifacts` configuration:

```
<dependencies>
   <dependency conf="artifacts->*" transitive="false" org="[module_org]" name="[module_name]
↪" rev="[module_version]" />

   <!-- ... other dependencies ... -->
</dependencies>
```

For example, to add the `ej.api.edc` library version `1.2.3`, write the following line:

```
<dependency conf="artifacts->*" transitive="false" org="ej.api" name="edc" rev="1.2.3" />
```

Note: We recommended to manually describe each dependency of the module repository, in order to keep full control of the included modules as well as included modules versions. Module dependencies can still be transitively included by setting the dependency attribute `transitive` to `true`. In this case, the included module versions are those that have been resolved when the module was built.

Multiple versions of the same module can be included by declaring each dependency using a different configuration. The `artifacts` configuration has to be derived with a new name as many times as there are different versions to include.

```
<configurations defaultconfmapping="default->default;provided->provided">
   <conf name="artifacts" visibility="private"/>
   <conf name="artifacts_1" visibility="private"/>
   <conf name="artifacts_2" visibility="private"/>

   <!-- ... other configurations ... -->
</configurations>

<dependencies>
   <dependency conf="artifacts->*" transitive="false" org="[module_org]" name="[module_name]
↪" rev="[module_version_1]" />
   <dependency conf="artifacts_1->*" transitive="false" org="[module_org]" name="[module_
↪name]" rev="[module_version_2]" />
   <dependency conf="artifacts_2->*" transitive="false" org="[module_org]" name="[module_
↪name]" rev="[module_version_3]" />

   <!-- ... other dependencies ... -->
</dependencies>
```

Include a Module Repository

To add all the modules already included in an other module repository, add the configuration `repository` if it does not exist:

```
<configurations defaultconfmapping="default->default;provided->provided">
   <!-- ... other configurations ... -->
   <conf name="repository" visibility="private" description="Repository to be embedded in␣
↪the repository" />

</configurations>
```

Then declare the module repository dependency using the `repository` configuration:

```
<dependencies>
   <dependency conf="repository->*" transitive="false" org="[repository_org]" name=
↪"[repository_name]" rev="[repository_version]" />

   <!-- ... other dependencies ... -->
</dependencies>
```

3.5.6 Generate Javadoc

An overall Javadoc can be generated beside the included modules. It is built from of all Java elements of all libraries included in the module repository.

Javadoc generation is disabled in the `module.ivy` generated by the skeleton. To enable javadoc generation, remove `skip.javadoc` option or set it to `false`.

There are also javadoc specific options such as Java packages exclusion. Please refer to `*javadoc*` options of *Module Repository* reference documentation.

As of *SDK 5.3.0*, the *module dependency* line that defines a Java type is shown in the top menu.

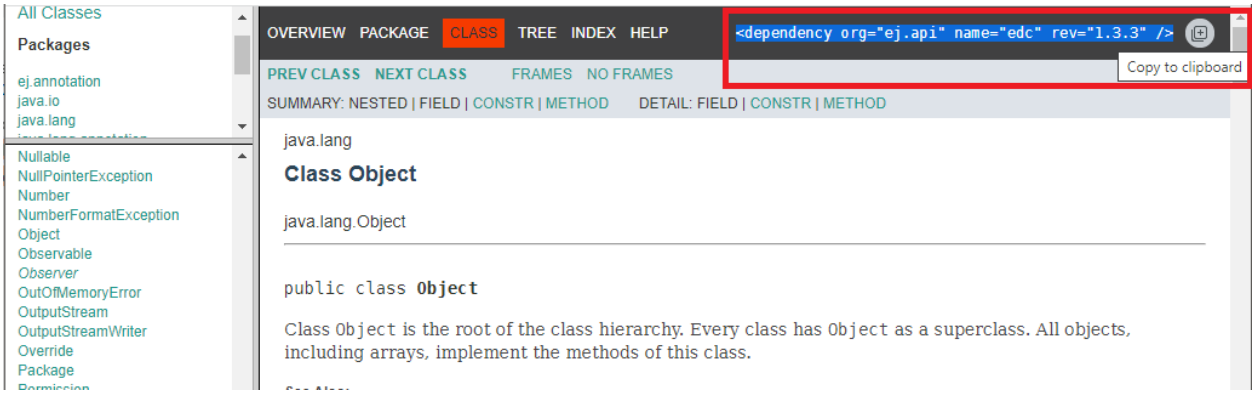


Fig. 52: Example of Javadoc Module Dependency

3.5.7 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`.

3.5.8 Use the Offline Repository

By default, when starting an empty workspace, MicroEJ SDK is configured to import dependencies from *MicroEJ Central Repository*.

To configure MicroEJ SDK to import dependencies from a local module repository, follow these steps:

1. Open the *MMM preferences page*: Window > Preferences > MicroEJ > Module Manager.
2. In Module Manager group, click on Import Repository.
3. Select the module repository `.zip` file, and then click on Finish.

The import may take some time. The module repository is unzipped in the folder `${user.dir}/.microej/repositories`, and the settings are updated.

3.6 Platform Selection

Note: This page is releated to the version 5 and lower of the SDK. If you use the SDK 6, please refer to the page *Select a VEE Port*.

Building or running an Application or a *Test Suite* with MMM requires a Platform.

There are 4 different ways to provide a Platform for a module project:

- Set the *build option* `platform-loader.target.platform.file` to the path of a Platform file (`.zip`, `.jpf` or `.vde`).
- Set the *build option* `platform-loader.target.platform.dir` to the path of the `source` folder of an already imported *Source Platform*.
- Declare a *module dependency* with the conf `platform`:

```
<dependency org="myorg" name="myname" rev="1.0.0" conf="platform->default"␣
↪transitive="false"/>
```

- Copy a Platform file to the dropins folder. The default dropins folder location is `[module_project_dir]/dropins`. It can be changed using the *build option* `platform-loader.target.platform.dropins`.

Note: Using a Platform in the `.zip` format requires at least the version `5.4.0` of the SDK.

At least 1 of these 4 ways is required to build an application with a platform. If several ways are used, the following rules are applied:

- If `platform-loader.target.platform.file` or `platform-loader.target.platform.dir` is set, the other options are ignored.
- If the the module project defined several platforms, the build fails. For example the following cases are not allowed:
  - Setting a platform with the option `platform-loader.target.platform.file` and another one with the option `platform-loader.target.platform.dir`
  - Declaring a platform as a dependency and adding a platform in the `dropins` folder
  - Declaring 2 platforms as Dependencies
  - Adding 2 platforms in the `dropins` folder

Refer to the *Platform Loader* section for a complete list of options.

3.7 Module Natures

Note: This page is releated to the version 5 and lower of the SDK. If you use the SDK 6, please refer to the page *Module Natures*.

This page describes the most common module natures as follows:

- **Skeleton Name**: the *project skeleton* name.
- **Build Type Name**: the build type name, derived from the module nature name: `com.is2t.easyant.buildtypes#build-[NATURE_NAME]`.
- **Documentation**: a link to the documentation.
- **SDK Menu**: the menu to the direct wizard in the SDK (if available). Any module nature can be created with the default wizard from File > New > Module Project.
- **Configuration**: properties that can be defined to configure the module. Properties are defined inside the `ea:build` tag of the *module.ivy* file, using `ea:property` tag as described in the section *Build Options*. A module nature also inherits the build options from the listed *Natures Plugins*.

3.7.1 Add-On Library

Skeleton Name: `microej-javalib`

Build Type Name: `com.is2t.easyant.buildtypes#build-microej-javalib`

Documentation: *Libraries*

SDK Menu: File > New > Add-On Library Project

Configuration:

This module nature inherits the build options of the following plugins:

- *Java Compilation*
- *Platform Loader*
- *Javadoc*
- *Test Suite*
- *Artifact Checker*

3.7.2 Add-On Processor

Skeleton Name: `addon-processor`

Build Type Name: `com.is2t.easyant.buildtypes#build-addon-processor`

Configuration:

This module nature inherits the build options of the following plugins:

- *Java Compilation*
- *J2SE Unit Tests*
- *Artifact Checker*

3.7.3 Foundation Library API

Skeleton Name: `microej-javaapi`

Build Type Name: `com.is2t.easyant.buildtypes#build-microej-javaapi`

Documentation: *Libraries*

Configuration:

This module nature inherits the build options of the following plugins:

- *Java Compilation*
- *Javadoc*
- *Artifact Checker*

This module nature defines the following dedicated build options:

Name	Description	Default
microej.lib.name	Platform library name on the form: `[NAME]-[VERSION]-api` . - `[NAME]` : name of the implemented Foundation Library API module. - `[VERSION]` : version of the implemented Foundation Library API module without patch (`Major.minor`).	Not set
rip.printableName	Printable name for the Platform Editor.	Not set

3.7.4 Foundation Library Implementation

Skeleton Name: `microej-javaimpl`

Build Type Name: `com.is2t.easyant.buildtypes#build-microej-javaimpl`

Documentation: *Libraries*

Configuration:

This module nature inherits the build options of the following plugins:

- *Java Compilation*
- *Test Suite*
- *Artifact Checker*[3]

This module nature defines the following dedicated build options:

Name	Description	Default
microej.lib.implfor	Execution target. Possible values are *emb* (only on Device), *sim* (only Simulator) and *common* (both).	`common`

3.7.5 Kernel Application

Skeleton Name: `firmware-multiapp`

Build Type Name: `com.is2t.easyant.buildtypes#build-firmware-multiapp`

Documentation: *Kernel Developer Guide*

Configuration:

This module nature inherits the build options of the following plugins:

- *Java Compilation*
- *Platform Loader*
- *Javadoc*
- *Artifact Checker*[3]

This module nature defines the following dedicated build options:

Name	Description	Default
application.main.class	Full Qualified Name of the main class of the kernel. This option is required.	Not set
runtime.api.name	Name of the Runtime API of the kernel. This option is ignored when a *Runtime API* is declared in the dependencies.	`RUNTIME`
runtime.api.version	Version of the Runtime API of the kernel. This option is ignored when a *Runtime API* is declared in the dependencies.	`1.0`
skip.build.virtual.device	When this property is set (any value), the virtual device is not built.	Not set
virtual.device.sim.only	When this property is set (any value), the Executable is not built.	Not set
launch.properties.jvm	Additional options to pass to the JVM for building the Executable.	-Xmx1024M

[3] Require SDK version `5.5.0` or higher.

3.7.6 Meta Build

Skeleton Name: `microej-meta-build`

Build Type Name: `com.is2t.easyant.buildtypes#microej-meta-build`

Documentation: *Meta Build*

Configuration:

This module nature defines the following dedicated build options:

Name	Description	Default
metabuild.root	Path of the root folder containing the modules to build.	`${basedir}/..`
private.modules.file	Name of the file listing the private modules to build.	`private.modules.list`
public.modules.file	Name of the file listing the public modules to build.	`public.modules.list`

3.7.7 Mock

Skeleton Name: `microej-mock`

Build Type Name: `com.is2t.easyant.buildtypes#build-microej-mock`

Documentation: *Mock*

Configuration:

This module nature inherits the build options of the following plugins:

- *Java Compilation*
- *J2SE Unit Tests*
- *Artifact Checker*[3]

3.7.8 Module Repository

Skeleton Name: `artifact-repository`

Build Type Name: `com.is2t.easyant.buildtypes#build-artifact-repository`

Documentation: *Module Repository*

Configuration:

This module nature inherits the build options of the following plugins:

- *Artifact Checker*

This module nature defines the following dedicated build options:

Name	Description	Default
architecture.configurations.includes[1]	Comma-separated list of configurations to include for the Architecture modules. Set `dist,eval` or `dist,prod` to include only evaluation or production Architectures or `dist,eval,prod` to include both.	`dist,eval`
bar.check.as.v2.module	When this property is set to true, the artifact checker uses the MicroEJ Module Manager semantic.	`false`
bar.javadoc.dir	Path of the folder containing the generated javadoc.	`${target}/javadoc`
bar.notification.email.from	The email address used as the from address when sending the notification emails.	Not set
bar.notification.email.host	The hostname of the mail service used to send the notification emails.	Not set
bar.notification.email.password	The password used to authenticate on the mail service.	Not set
bar.notification.email.port	The port of the mail service used to send the notification emails	Not set
bar.notification.email.ssl	When this property is set to true, SSL/TLS is used to send the notification emails.	Not set
bar.notification.email.to	The notification email address destination.	Not set
bar.notification.email.user	The username used to authenticate on the mail service.	Not set
bar.populate.from.resolver	Name of the resolver used to fetch the modules to populate the repository.	`fetchRelease`
bar.populate.ivy.settings.file	Path of the Ivy settings file used to fetch the modules to populate the repository.	`${project.ivy.settings.file}`
bar.populate.repository.conf	Ivy configuration of included repositories. The modules of the repositories declared as dependency with this configuration are included in the built repository.	`repository`
bar.test.haltonerror	When this property is set to true, the artifact checker stops at the first error.	`false`
javadoc.excludes	Comma-separated list of packages to exclude from the javadoc.	Empty string
javadoc.includes	Comma-separated list of packages to include in the javadoc.	`**` (all packages)
javadoc.modules.excludes[2]	Comma-separated list of modules to exclude from the javadoc.	Empty string
skip.artifact.checker	When this property is set to true, all artifact checkers are skipped.	Not set
skip.email	When this property is set (any value), the notification email is not sent. Otherwise the `bar.notification.*` properties are required.	Not set
skip.javadoc	Prevents the generation of the javadoc.	`false`
skip.javadoc.deprecated	Prevents the generation of any deprecated API at all in the javadoc.	`true`

[1] Require SDK version `5.4.0` or higher.
[2] Require SDK version `5.6.0` or higher.

3.7.9 Runtime Environment

Skeleton Name: `runtime-api`

Build Type Name: `com.is2t.easyant.buildtypes#build-runtime-api`

Documentation: *Runtime Environment*

Configuration:

This module nature inherits the configuration properties of the following plugins:

- *Artifact Checker*

3.7.10 Sandboxed Application

Skeleton Name: `application`

Build Type Name: `com.is2t.easyant.buildtypes#build-application`

Documentation: *Sandboxed Application*

SDK Menu: File > New > Sandboxed Application Project

Configuration:

This module nature inherits the build options of the following plugins:

- *Java Compilation*
- *Platform Loader*
- *Javadoc*
- *Test Suite*
- *Artifact Checker*

3.7.11 Standalone Application

Skeleton Name: `firmware-singleapp`

Build Type Name: `com.is2t.easyant.buildtypes#build-firmware-singleapp`

Documentation: *Standalone Application*

SDK Menu: File > New > Standalone Application Project

Configuration:

This module nature inherits the build options of the following plugins:

- *Java Compilation*
- *Platform Loader*
- *Javadoc*[Page 73, 3]
- *Test Suite*[Page 73, 3]
- *Artifact Checker*[Page 73, 3]

This module nature defines the following dedicated build options:

Name	Description	Default
application.main.class	Full Qualified Name of the main class of the application. This option is required.	Not set
skip.build.virtual.device	When this property is set (any value), the virtual device is not built.	Not set
virtual.device.sim.only	When this property is set (any value), the Executable is not built.	Not set
launch.properties.jvm	Additional options to pass to the JVM for building the Executable.	-Xmx1024M

3.7.12 Studio Rebranding

Skeleton Name: `microej-studio-rebrand`

Build Type Name: `com.is2t.easyant.buildtypes#build-izpack`

Configuration:

The skeleton template contains all the necessary files for a Studio that is ready to build. The main elements are:

- `HOWTO.md` : This file describes the minimum configuration required to build the Studio template as it is.
- `module.ivy` : This file describes all available build options and dependencies.
- `branding-resources` : This folder contains default resources that can be replaced with your own to customize the Studio. These resources include names, images, icons, and license files.

3.7.13 Natures Plugins

This page describes the most common module nature plugins as follows:

- **Documentation**: link to documentation.
- **Module Natures**: list of *Module Natures* using this plugin.
- **Configuration**: properties that can be defined to configure the module. Properties are defined inside the `ea:build` tag of the *module.ivy* file, using `ea:property` tag as described in the section *Build Options*.

Java Compilation

Module Natures:

This plugin is used by the following module natures:

- *Add-On Library*
- *Foundation Library API*
- *Foundation Library Implementation*
- *Standalone Application*
- *Sandboxed Application*

Configuration:

This plugin defines the following build options:

Name	Description	Default
javac.debug.level	Comma-separated list of levels for the Java compiler debug mode.	`lines, source, vars`
javac.debug.mode	When this property is set to true, the Java compiler is set in debug mode.	`false`
src.main.java	Path of the folder containing the Java sources.	`${basedir}/src/main/java`

Platform Loader

Documentation: *Platform Selection*

Module Natures:

This plugin is used by the following module natures:

- *Add-On Library*
- *Standalone Application*
- *Sandboxed Application*

Configuration:

This plugin defines the following build options:

Name	Description	Default
platform-loader.platform.dir	Path of the folder to unzip the loaded platform to.	`${target}/platform`
platform.loader.skip.load.platform	When this property is set to true, the platform is not loaded. It must be already available in the directory defined by the property `platform-loader.platform.dir` . Use with caution: the platform content may be modified during the build (e.g. in case of Testsuite or Virtual Device build).	`false`
platform-loader.target.platform.conf	The Ivy configuration used to retrieved the platform if fetched via dependencies.	`platform`
platform-loader.target.platform.dir	Path of the root folder of the platform to use in the build. See *Platform Selection* section for Platform Selection rules.	Not set
platform-loader.target.platform.dropins	Absolute or relative (to the project root folder) path of the folder where the platform can be found (see *Platform Selection*).	`dropins`
platform-loader.target.platform.file	Path of the platform file to use in the build. See *Platform Selection* section for Platform Selection rules.	Not set

Javadoc

Module Natures:

This plugin is used by the following module natures:

- *Add-On Library*
- *Foundation Library API*
- *Sandboxed Application*

Configuration:

This plugin defines the following build options:

Name	Description	Default
src.main.java	Path of the folder containing the Java sources.	`${basedir}/src/main/java`
javadoc.file.encoding	Encoding used for the generated Javadoc.	`UTF-8`
javadoc.failonerror	When this property is set to true, the build is stopped if an error is raised during the Javadoc generation.	`true`
javadoc.failonwarning	When this property is set to true, the build is stopped if a warning is raised during the Javadoc generation.	`false`
target.reports	Path of the base folder for reports.	`${target}/reports`
target.javadoc	Path of the base folder where the Javadoc is generated.	`${target.reports}/javadoc`
target.javadoc.main	Path of the folder where the Javadoc is generated.	`${target.javadoc}/main`
javadoc-microej.overview.html	Path of the HTML template file used for the Javadoc overview page.	`${src.main.java}/overview.html` if exists, otherwise a default template.
target.artifacts	Path of the packaged artifacts.	`${target}/artifacts`
target.artifacts.main.javadoc.jar.name	Name of the packaged JAR containing the generated Javadoc (stored in folder `target.artifacts`).	`${module.name}-javadoc.jar`
javadoc.publish.conf	Ivy configuration used to publish the Javadoc artifact.	`documentation`

Test Suite

Documentation: *Test Suite with JUnit*

Module Natures:

This plugin is used by the following module natures:

- *Add-On Library*
- *Foundation Library API*
- *Foundation Library Implementation*
- *Standalone Application*
- *Sandboxed Application*

Configuration:

This plugin defines the following build options:

Name	Description	Default
microej.testsuite.cc.excludes.classes	Pattern of classes excluded from the code coverage analysis	Not set
microej.testsuite.retry.count	A test execution may not be able to produce the success trace for an external reason, for example an unreliable harness script that may lose some trace characters or crop the end of the trace. For all these unlikely reasons, it is possible to configure the number of retries before a test is considered to have failed.	`0`
microej.testsuite.timeout	The time in seconds before a test is considered as failed. Set it to `0` to disable the timeout.	`60`
microej.testsuite.properties.[name]	Inject an *Application Option* named `[name]` for all tests. For example, declaring the build option `microej.testsuite.properties.core.memory.javaheap.size` will configure the Java heap size of all tests.	Not applicable
microej.testsuite.properties.launch.execution.trace	Set this property to `true` if your *VEE Port Run script* redirects execution traces.	Not set
microej.testsuite.properties.syso.is.enabled	When this property is set to true, the code coverage analysis is enabled.	`true`
microej.testsuite.properties.testsuite.trace.ip	The TCP/IP address to connect for retrieving test execution traces. This property is required if your *VEE Port Run script* does not redirect execution traces.	Not set
microej.testsuite.properties.testsuite.trace.port	The TCP/IP port to connect for retrieving test execution traces. This property is required if your *VEE Port Run script* does not redirect execution traces.	Not set
microej.testsuite.properties.testsuite.trace.timeout	The time in seconds without activity on the standard output before the test trace analysis is stopped.	`75`
cc.src.folders	Path to the folders containing the Java sources used for code coverage analysis.	Java source folder (`src/main/java`) and Add-On Processor generated source folders (`src-adpgenerated/*`)[4]
microej.testsuite.verbose	When this property is set to true, the verbose trace level is enabled.	`false`
test.run.excludes.pattern	Pattern of classes excluded from the test suite execution.	Empty string (no test)
test.run.failonerror	When this property is set to true, the build fails if an error is raised.	`true`
target.vm.name	The execution target (`S3` to execute on Simulator, `MICROJVM` to execute on the Device).	`S3`
test.run.includes.pattern	Pattern of classes included in the test suite execution.	`**/*` (all tests)
skip.test	When this property is set (any value), the tests are not executed.	Not set

[4] Option `cc.src.folders` is not set by default for SDK versions lower than `5.5.0`.

J2SE Unit Tests

Warning: This plugin is reserved for tools written in Java Standard Edition. Tests classes must be created in the folder `src/test/java` of the project. See *Test Suite* section for MicroEJ tests.

Module Natures:

This plugin is used by the following module natures:

- *Add-On Processor*
- *Mock*

Configuration:

This plugin defines the following build options:

Name	Description	Default
test.run.excludes.pattern	Pattern of classes excluded from the test suite execution.	Empty string (no test)
test.run.failonerror	When this property is set to true, the build fails if an error is raised.	`true`
test.run.includes.pattern	Pattern of classes included in the test suite execution.	`**/*` (all tests)
skip.test	When this property is set (any value), the tests are not executed.	Not set

Artifact Checker

Module Natures:

This plugin is used by the following module natures:

- *Add-On Library*
- *Foundation Library API*
- *Standalone Application*
- *Sandboxed Application*
- *Module Repository*

Configuration:

This plugin defines the following build options:

Name	Description	Default
run.artifact.checker	When this property is set (any value), the artifact checker is executed.	Not set
skip.addonconf.checker	When this property is set to true, the addon configurations checker is not executed.	Not set
skip.changelog.checker	When this property is set to true, the changelog checker is not executed.	Not set
skip.foundationconf.checker	When this property is set to true, the foundation configurations checker is not executed.	Not set
skip.license.checker	When this property is set to true, the license checker is not executed.	Not set
skip.nullanalysis.checker[5]	When this property is set to true, the null analysis checker is not executed.	Not set
skip.publicconf.checker	When this property is set to true, the public configurations checker is not executed.	Not set
skip.readme.checker	When this property is set to true, the readme checker is not executed.	Not set
skip.retrieve.checker	When this property is set to true, the retrieve checker is not executed.	Not set

3.7.14 Global Build Options

The following *Build Options* are available in any module:

Name	Description	Default
`target`	Path of the build directory `target~`.	`${basedir}/target~`

3.8 Debug an Application

Note: This page is releated to the version 5 and lower of the SDK. If you use the SDK 6, please refer to the page *Debug on Simulator*.

3.8.1 Debug on Simulator

To debug an application on Simulator, select it in the left panel then right-click and select Debug As > MicroEJ Application.

[5] Require SDK version `5.5.0` or higher.

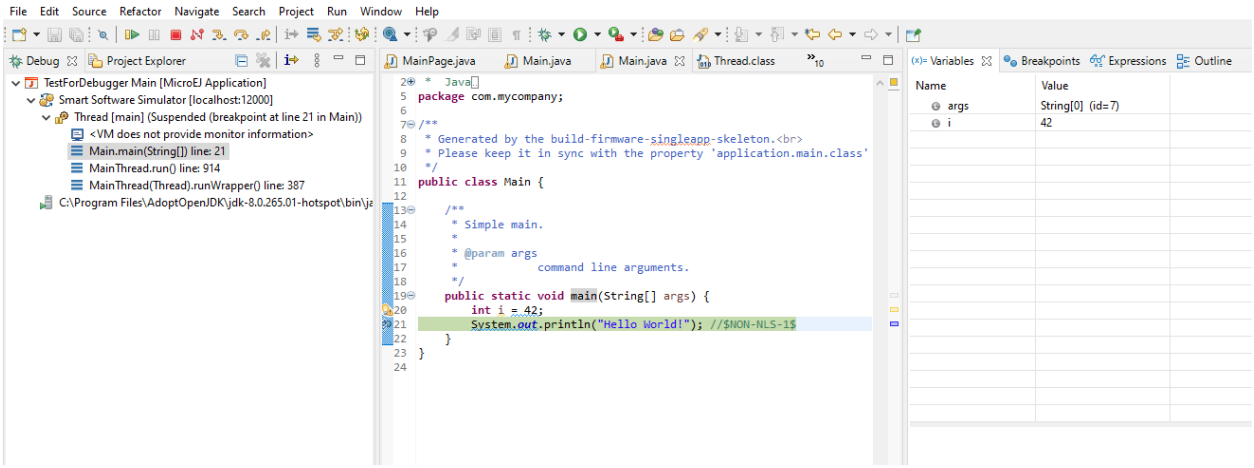


Fig. 53: MicroEJ Development Tools Overview of the Debugger on Simulator

3.8.2 Debug on Device

To debug an application on device, first run the *MicroEJ debugger proxy*, and run a Remote Java Application launch:

- Go to `Run` > `Debug Configurations` > `Remote Java Application`
- Set the informations about the project to debug, the proxy connections properties, etc.
- Click on `Debug`

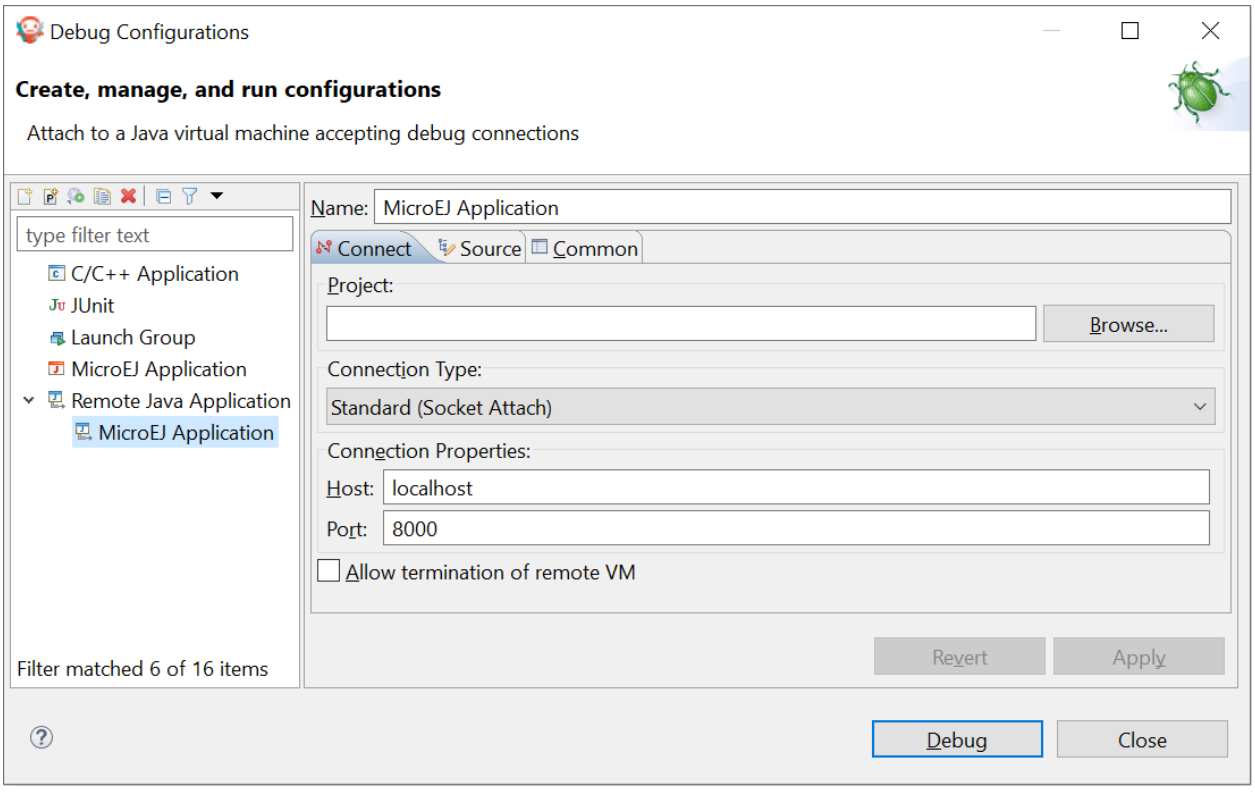


Fig. 54: MicroEJ Development Tools Overview of the Remote Java Application

In the SDK, open the Debug perspective (Window > Perspective > Open Perspective > Other... > Debug) to show the current debugging process.

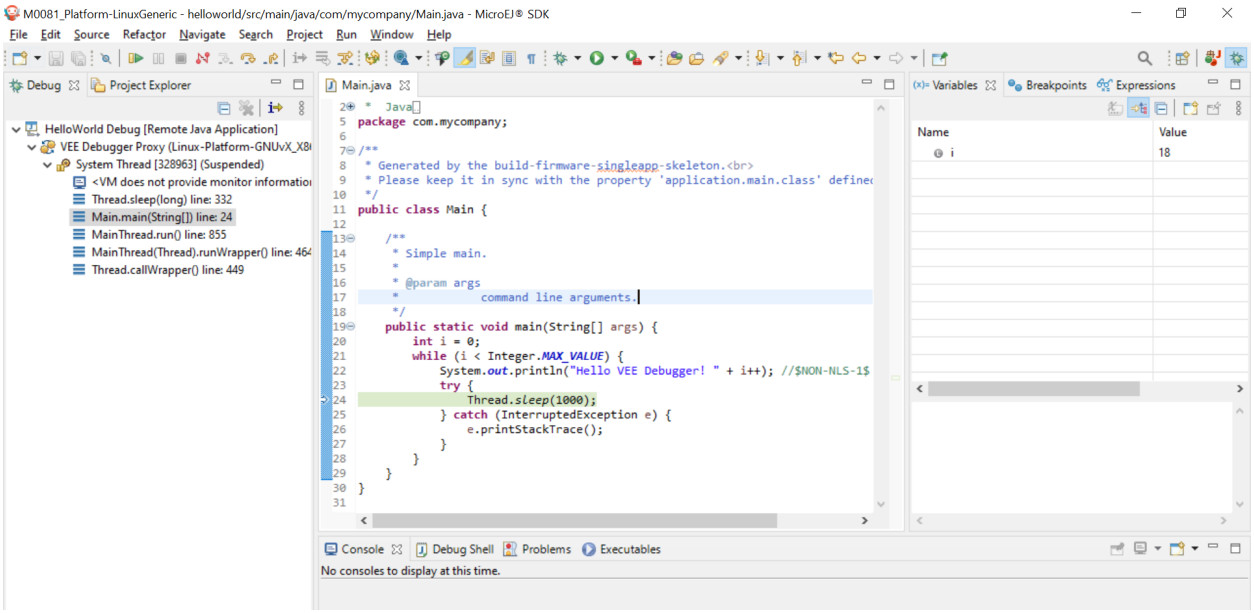


Fig. 55: MicroEJ Development Tools Overview of the Debugger on Board

It makes use of Eclipse Java debugger client. If you are unfamiliar with Java debugging or Eclipse IDE, see Debugging the Eclipse IDE for Java Developers to get started.

You can also debug with IntelliJ IDEA. For more information on IntelliJ IDEA Remote debug process, see IntelliJ IDEA Remote debug

3.8.3 Get Library Sources

All libraries included in MicroEJ SDK are provided with their source code and resources. The way the sources are retrieved depends on the kind of library (Add-On Library or Foundation Library).

Add-On Library Sources

Add-On Library sources are packaged in a dedicated file named `[module_name]-source.jar` available in the module directory:

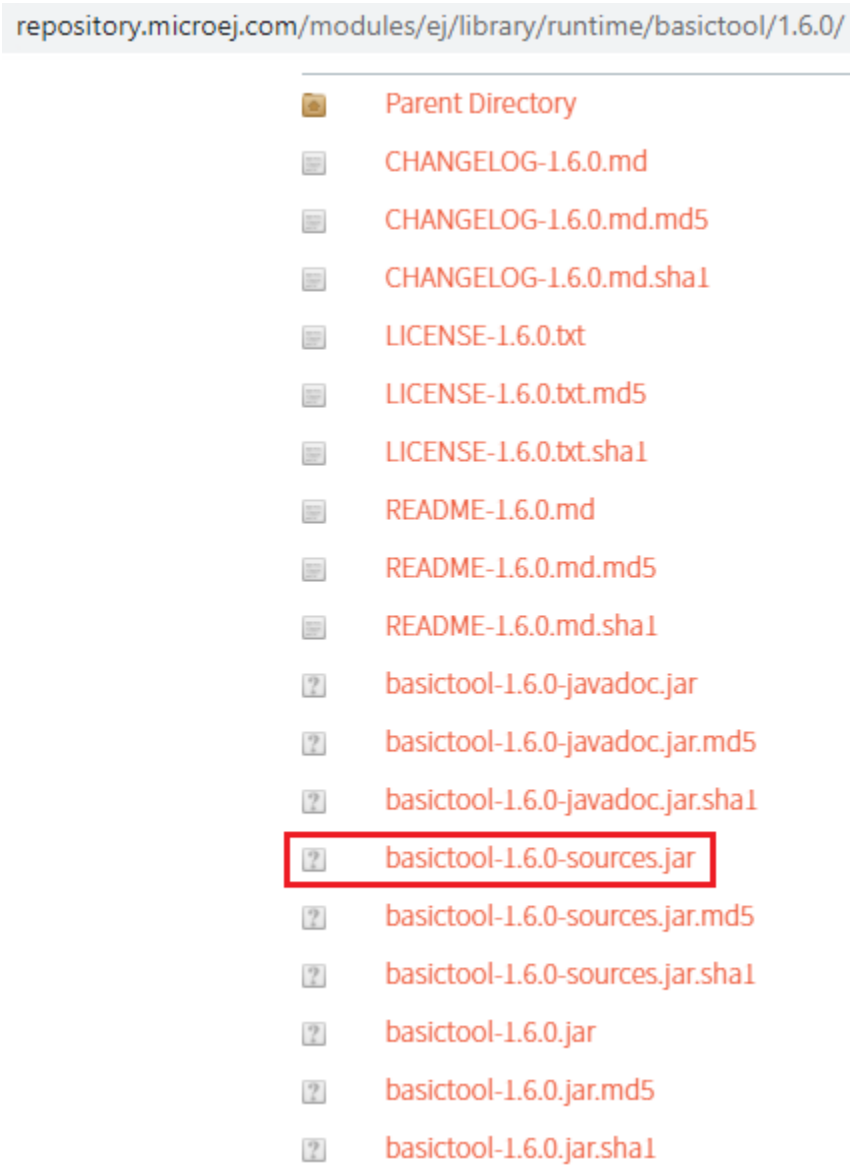


Fig. 56: Add-On Library Sources Location

In the SDK, sources are automatically connected to Eclipse JDT when the new Add-On Library is added as a *module dependency*.

On any Java element (type, method, field), press `F3` or `CTRL-Click` to open the implementation:

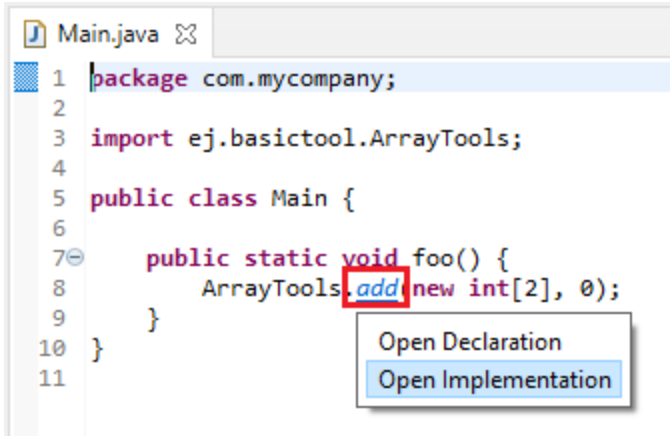


Fig. 57: Add-On Library Open Implementation

Then the implementation class is open in read-only mode.

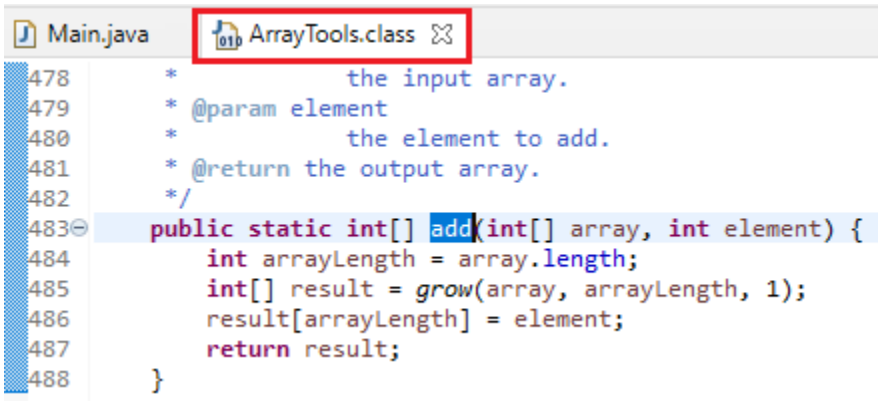


Fig. 58: Add-On Library Read-Only Source Code

Foundation Library Sources

Foundation Library sources are directly included in the implementation file (JAR file) provided by the Platform. They are located in the following Platform folders:

- `javaLibs` for generic Foundation Libraries (defaults).
- `MICROJVM/javaLibs` for Foundation Libraries specific to the MicroEJ Core Engine.
- `S3/javaLibs` for Foundation Libraries specific to the Simulator.

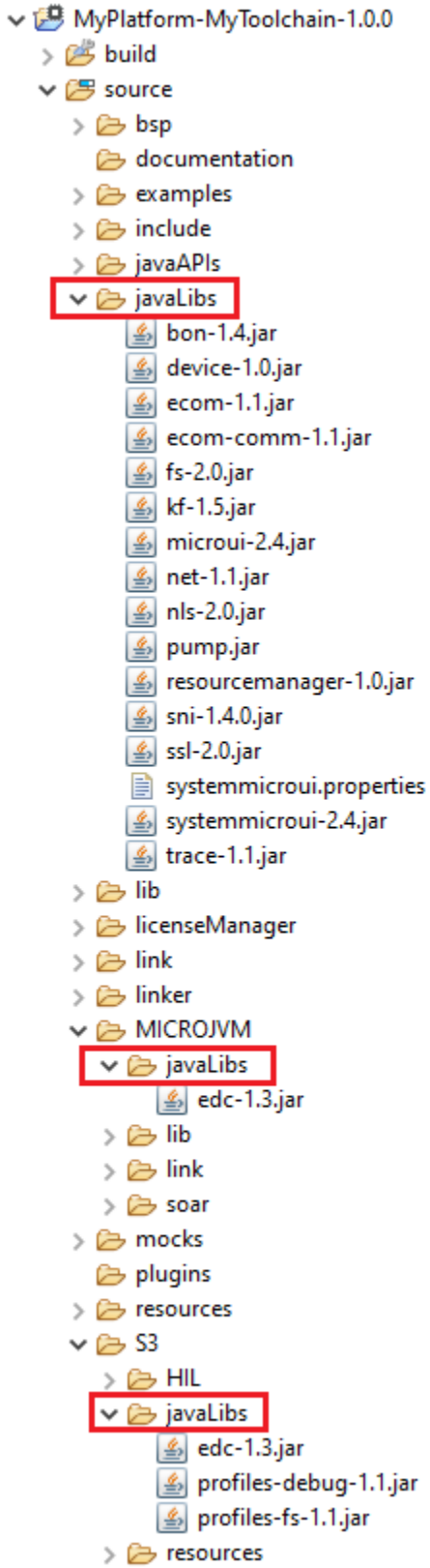


Fig. 59: Foundation Library Platform Folders

In the SDK, sources can be connected while debugging an Application on Simulator. This ensures to get the exact source code which is executed on your Platform.

Here are the steps to attach Foundation Library sources from a Platform loaded in the workspace:

- Open a *MicroEJ Application launch*,
- Select the `Source` tab (see also *Source Tab*),
- Click on `Add…` button,
- Select `Archive` item and press `OK` ,

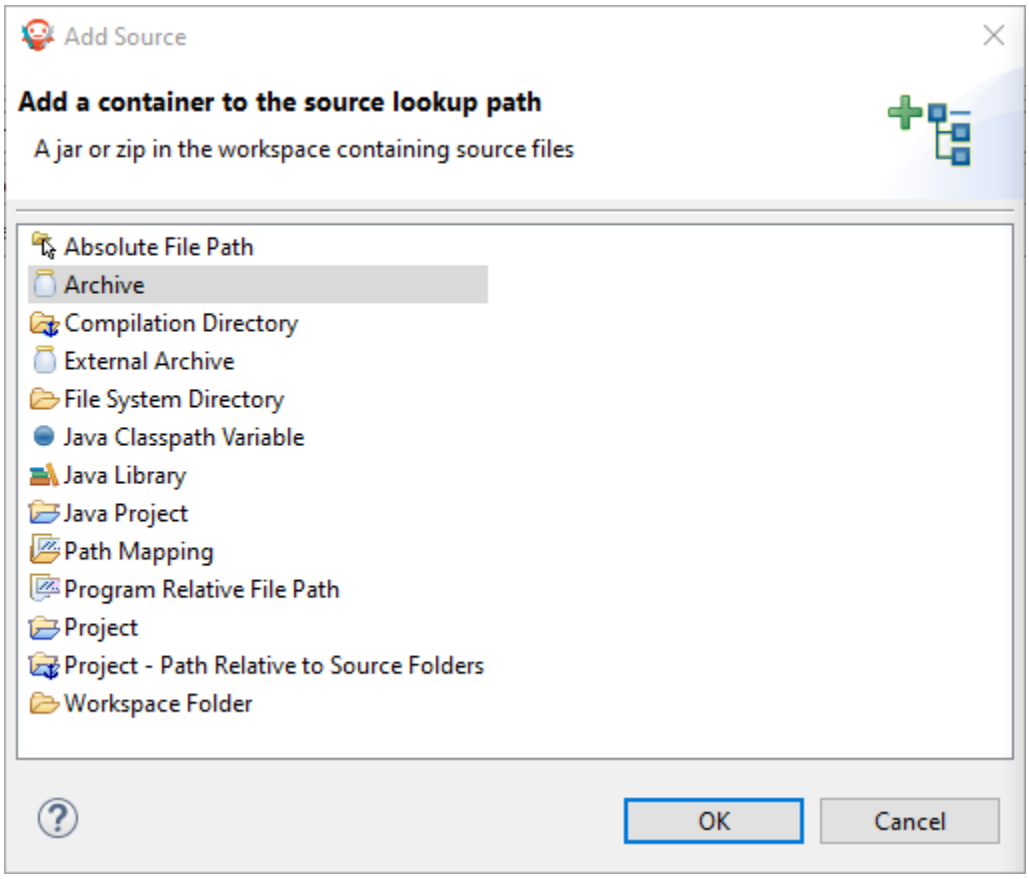


Fig. 60: Add Foundation Library Sources to MicroEJ Application Launch

- Select the Foundation Libraries from Platform folders and press `OK` ,

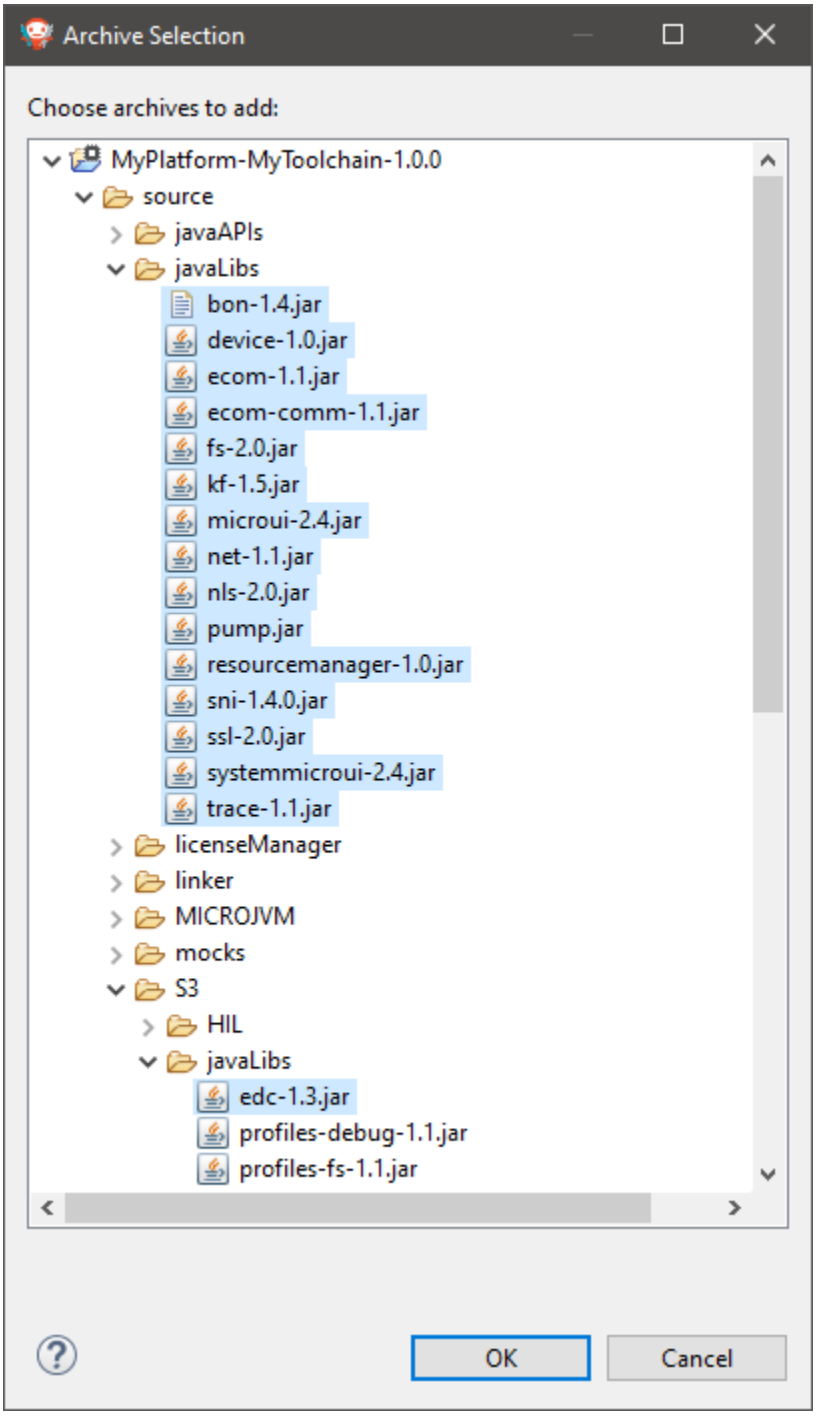


Fig. 61: Select Foundation Libraries Implementation files

Warning: You must select the libraries from the Platform project corresponding to the execution Platform (see *Execution Tab*).

In the debug session the implementation sources will be now displayed.

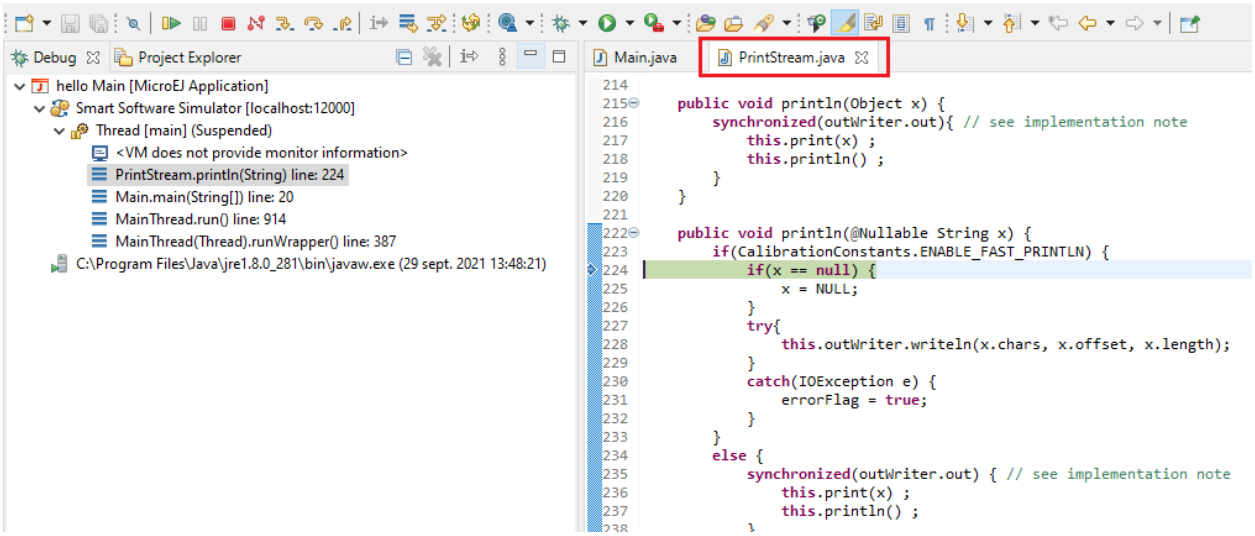


Fig. 62: Foundation Library Read-Only Source Code

3.9 Development Tools

MicroEJ provides a number of tools to assist with various aspects of development. Some of these tools are run using MicroEJ Tool configurations, and created using the Run Configurations dialog of the MicroEJ SDK. A configuration must be created for the tool before it can be used.

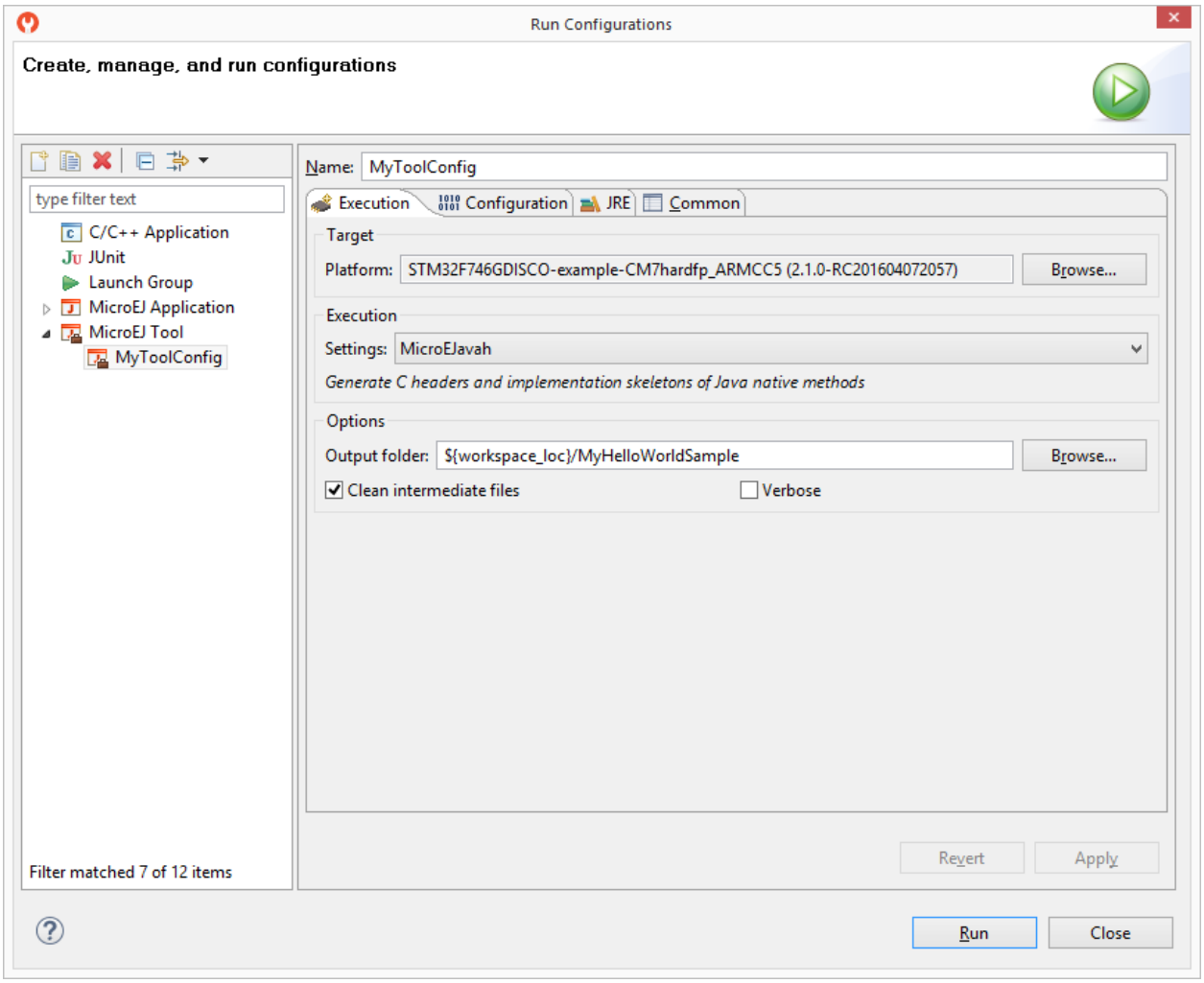


Fig. 63: MicroEJ Tool Configuration

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.

3.9.1 Test Suite with JUnit

The SDK allows to run unit tests using the standard JUnit API during the build process of a Library or an Application. The *MicroEJ Test Suite Engine* runs tests on a VEE Port and outputs a JUnit XML report.

Principle

JUnit testing can be enabled when using the `microej-javalib` (MicroEJ Add-On Library) or the `microej-application` (MicroEJ Applications) build type. JUnit test cases processing is automatically enabled when the following dependency is declared in the `module.ivy` file of the project.

```
<dependency conf="test->*" org="ej.library.test" name="junit" rev="1.6.2"/>
```

When a new JUnit test case class is created in the `src/test/java` folder, a JUnit processor generates MicroEJ compliant classes into a specific source folder named `src-adpgenerated/junit/java`. These files are automatically managed and must not be edited manually.

JUnit Compliance

MicroEJ is compliant with a subset of JUnit version 4. MicroEJ JUnit processor supports the following annotations: `@After`, `@AfterClass`, `@Before`, `@BeforeClass`, `@Ignore`, `@Test`.

Each test case entry point must be declared using the `org.junit.Test` annotation (`@Test` before a method declaration). Please refer to JUnit documentation to get details on usage of other annotations.

Setup a Platform for Tests

Before running tests, a target platform must be configured.

Execution in SDK

In order to execute the Test Suite in the SDK, a target platform must be configured in the MicroEJ workspace. The following steps assume that a platform has been previously imported into the MicroEJ Platform repository (or available in the Workspace):

- Go to Window > Preferences > MicroEJ > Platforms (or Platforms in workspace).
- Select the desired platform on which to run the tests.
- Press `F2` to expand the details.
- Select the the platform path and copy it to the clipboard.
- Go to Window > Preferences > Ant > Runtime and select the Properties tab.
- Click on Add Property... button and set a new property named `target.platform.dir` with the platform path pasted from the clipboard.

Execution during module build

In order to execute the Test Suite during the build of the module, a target platform must be configured in the module project as described in the section *Platform Selection*.

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 Test Suite

- Right-click on the `mylibrary` project and select `Build Module` . After the library is built, the test suite 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 test suite report.
- Modify the test case by replacing

```
fail("Not yet implemented");
```

with

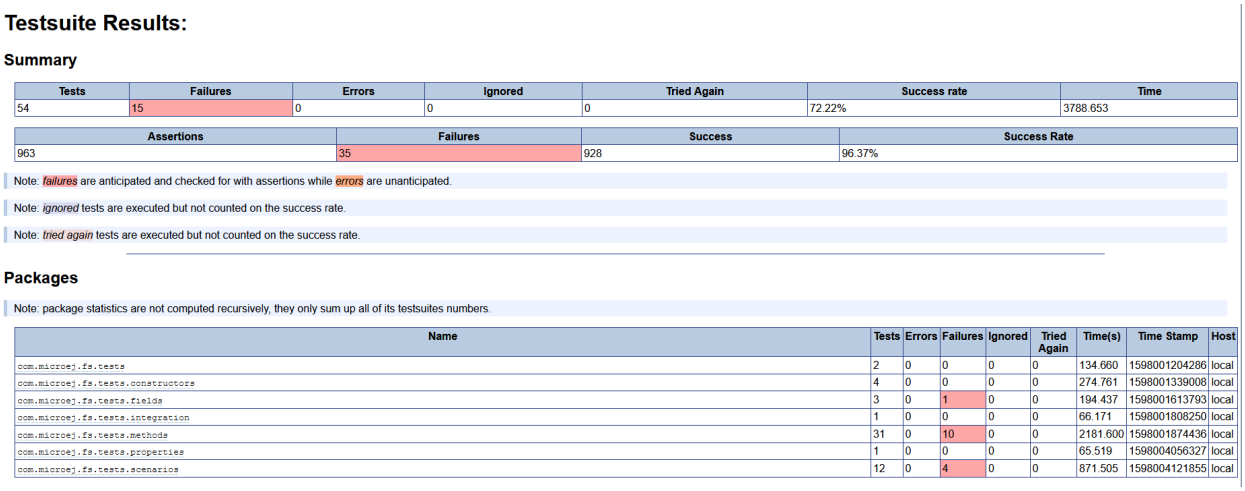
```
Assert.assertTrue(true);
```

- 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 test suite report to see the test has been successfully executed.

Test Suite Reports

Once a test suite is completed, the following test suite reports are generated:

- JUnit HTML report in the module project location `target~/test/html/test/junit-noframes.html` . This report contains a summary and the execution trace of every executed test.



Testsuite Results:

Summary

Tests	Failures	Errors	Ignored	Tried Again	Success rate	Time
54	15	0	0	0	72.22%	3788.653

Assertions	Failures	Success	Success Rate
963	35	928	96.37%

Note: *failures* are anticipated and checked for with assertions while *errors* are unanticipated.

Note: *ignored* tests are executed but not counted on the success rate.

Note: *tried again* tests are executed but not counted on the success rate.

Packages

Note: package statistics are not computed recursively, they only sum up all of its testsuites numbers.

Name	Tests	Errors	Failures	Ignored	Tried Again	Time(s)	Time Stamp	Host
com.microej.fs.tests	2	0	0	0	0	134.660	1598001204286	local
com.microej.fs.tests.constructors	4	0	0	0	0	274.761	1598001339008	local
com.microej.fs.tests.fields	3	0	1	0	0	194.437	1598001613793	local
com.microej.fs.tests.integration	1	0	0	0	0	66.171	1598001808250	local
com.microej.fs.tests.methods	31	0	10	0	0	2181.600	1598001874436	local
com.microej.fs.tests.properties	1	0	0	0	0	65.519	1598004056327	local
com.microej.fs.tests.scenarios	12	0	4	0	0	871.505	1598004121855	local

Fig. 64: Example of MicroEJ Test Suite HTML Report

- JUnit XML report in the module project location `target~/test/xml/TEST-test-report.xml` .

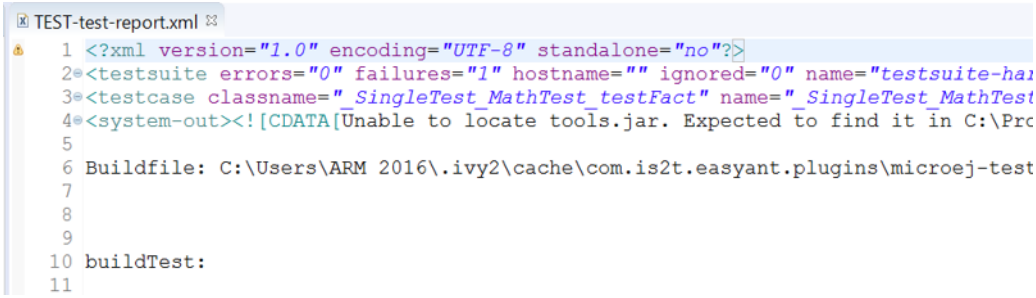


Fig. 65: Example of MicroEJ Test Suite XML Report

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

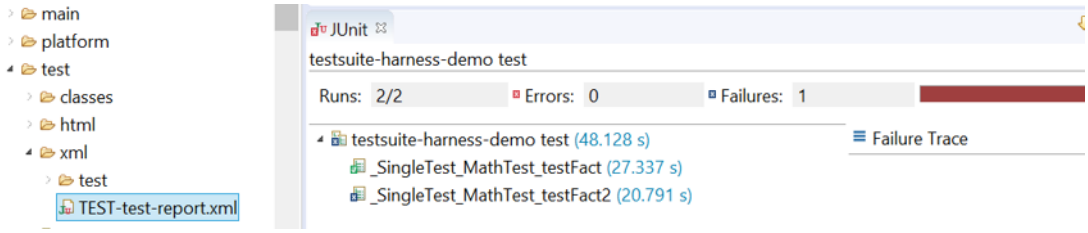


Fig. 66: Example of MicroEJ Test Suite XML Report in JUnit View

If executed on device, the Firmware binary produced for each test is available in module project location `target~/test/xml/<TIMESTAMP>/bin/<FULLY-QUALIFIED-CLASSNAME>/application.out` .

Configure the Execution on your Device

By default, the Test Suite is configured to execute tests on the Simulator using Mocks declared by the VEE Port. You can switch the default configuration to execute tests on your Device. For that, your VEE Port must implement the *BSP Connection*.

Also, a device must be connected to your workstation both for programming the Executable and getting output traces. Consult your VEE Port specific documentation for setup.

Here is a summary of the options to add (see *Testsuite Options* and *BSP Connection Options* for more details).

```
<!-- Execute tests on Device -->
<ea:property name="target.vm.name" value="MICROJVM"/>

<!-- Enable Executable built using the SDK -->
<ea:property name="microej.testsuite.properties.deploy.bsp.microejscript" value="true"/>
<ea:property name="microej.testsuite.properties.microejtool.deploy.name" value=
↪"deployToolBSPRun"/>

<!-- Tell the testsuite engine that your VEE Port Run script redirects execution traces -->
<ea:property name="microej.testsuite.properties.launch.test.trace.file" value="true"/>
<!-- Configure TCP/IP address and port if your VEE Port Run script does not redirect␣
↪execution traces -->
<ea:property name="microej.testsuite.properties.testsuite.trace.ip" value="127.0.0.1"/>
<ea:property name="microej.testsuite.properties.testsuite.trace.port" value="5555"/>
```

Warning: If your VEE Port Run script does not redirect execution traces, the *Serial to Socket Transmitter* tool must have been started before running the Test Suite.

Advanced Configurations

Autogenerated Test Classes

The JUnit processor generates test classes into the `src-adpgenerated/junit/java` folder. This folder contains:

`_AllTestClasses.java` file
: A single class with a main entry point that sequentially calls all declared test methods of all JUnit test case classes.

`_AllTests_[TestCase].java` files
: For each JUnit test case class, a class with a main entry point that sequentially calls all declared test methods.

`_SingleTest_[TestCase]_[TestMethod].java` files
: For each test method of each JUnit test case class, a class with a main entry point that calls the test method.

JUnit Test Case to MicroEJ Test Case

The *MicroEJ Test Suite Engine* allows to select the classes that will be executed, by adding the following configuration in the project build file:

MMM (module.ivy)

Gradle (build.gradle.kts)

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

```
tasks.test {
   filter {
      includeTestsMatching([MicroEJ Test Case Include Pattern])
   }
}
```

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

MMM (module.ivy)

Gradle (build.gradle.kts)

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

```
tasks.test {
   filter {
      includeTestsMatching("*._AllTests_*")
   }
}
```

The following configuration considers each JUnit test method as a dedicated MicroEJ test case. Each test method is viewed independently in the JUnit report, but this may slow down the test suite execution because a new deployment is done for each test method.

MMM (module.ivy)

Gradle (build.gradle.kts)

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

```
tasks.test {
   filter {
      includeTestsMatching("*._SingleTest_*")
   }
}
```

Test Suite Options (SDK 5 only)

The *MicroEJ Test Suite Engine* can be configured with specific options which can be added to the `module.ivy` file of the project running the test suite, within the `<ea:build>` XML element.

Test Suite options are described in the *Test Suite Module Nature* section.

Test Specific Options

The *MicroEJ Test Suite Engine* allows to define *Standalone Application Options* specific to each test case. This can be done by defining a file with the same name as the generated test case file with the `.properties` extension instead of the `.java` extension. The file must be put in the `src/test/resources` folder and within the same package than the test case file.

3.9.2 Stack Trace Reader

Principle

Stack Trace Reader is a MicroEJ tool that 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 names(see *Types*), and method names obtained are encoded with a MicroEJ internal format. This internal format prevents embedding all class names and method names in the executable image to save some memory space. The Stack Trace Reader tool allows you to decode the stack traces by replacing the internal class names and method names with their real names. It also retrieves the line numbers in the MicroEJ Application.

Functional Description

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

When *Multi-Sandbox capability* is enabled, the stack trace reader can simultaneously decode heterogeneous stack traces with lines owned by different MicroEJ Sandboxed Applications and the firmware. Lines owned by the firmware can be decoded with the firmware debug information file (optionally made available by your firmware provider).

Dependencies

No dependency.

Installation

This tool is a built-in Architecture tool.

Use (Standalone Application)

For example, write the following new line to dump the currently executed stack trace on the standard output.

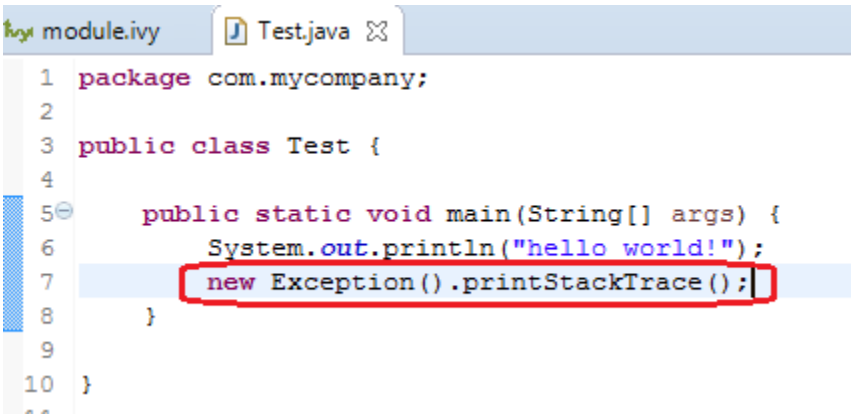


Fig. 67: Code to Dump a Stack Trace

To decode an application stack trace, the stack trace reader tool requires the application executable ELF file. In the case of a platform with full BSP connection (see *BSP Connection Cases*), the file is `application.out` in the output folder. In the other cases, the ELF file is generated by the C toolchain when building the BSP project (usually a `.out` or `.axf` file).

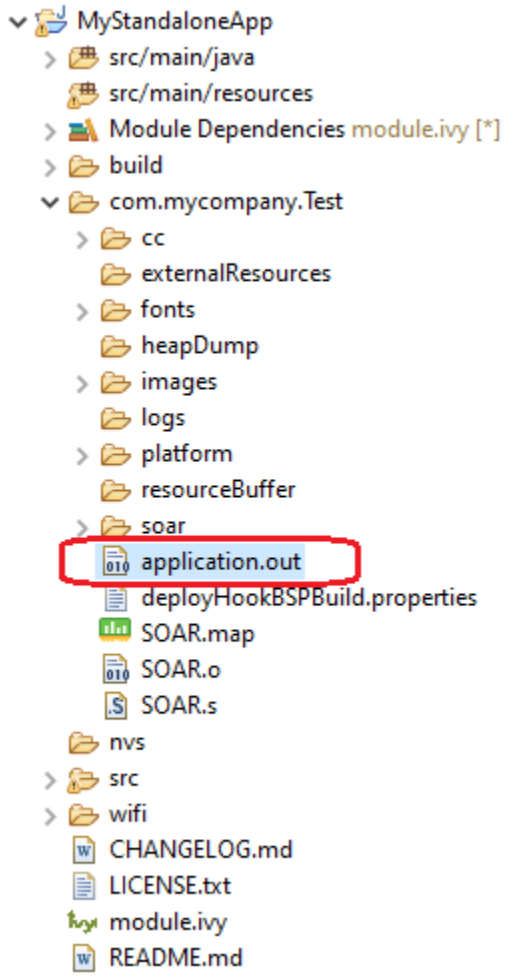


Fig. 68: Application Binary File

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

```
VM START
Hello World!
Exception in thread "main" java.lang.Exception
    at java.lang.System.@M:0x3f407778:0x3f407782@
    at java.lang.Throwable.@M:0x3f408030:0x3f408046@
    at java.lang.Throwable.@M:0x3f4089cc:0x3f4089e6@
    at com.mycompany.Test.@M:0x3f40762c:0x3f407652@
    at java.lang.MainThread.@M:0x3f407a84:0x3f407a98@
    at java.lang.Thread.@M:0x3f408b88:0x3f408b94@
    at java.lang.Thread.@M:0x3f408c74:0x3f408c7f@
VM END (exit code = 0)
```

Fig. 69: Stack Trace Output

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

Run Configurations… .

Create a new MicroEJ Tool configuration. In the Execution tab, select your target platform, then select the Stack Trace Reader tool. Set an output folder in the Output folder field.

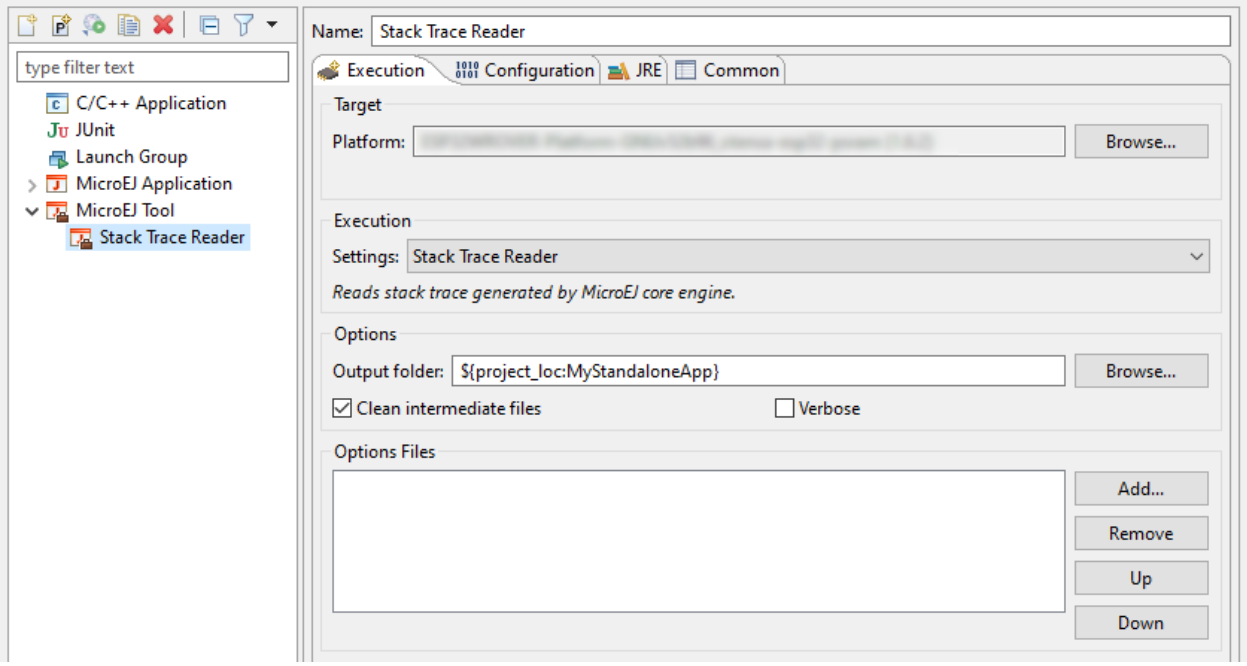


Fig. 70: Stack Trace Reader Tool Configuration (Platform Selection)

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

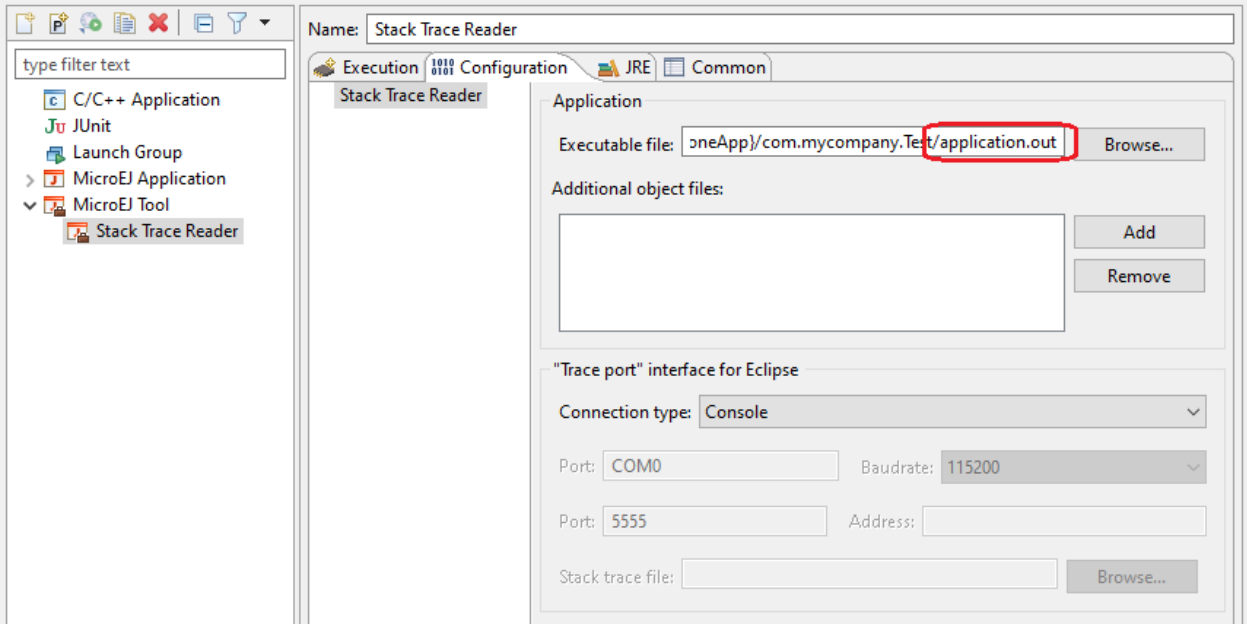


Fig. 71: Stack Trace Reader Tool Configuration (Standalone Application)

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.

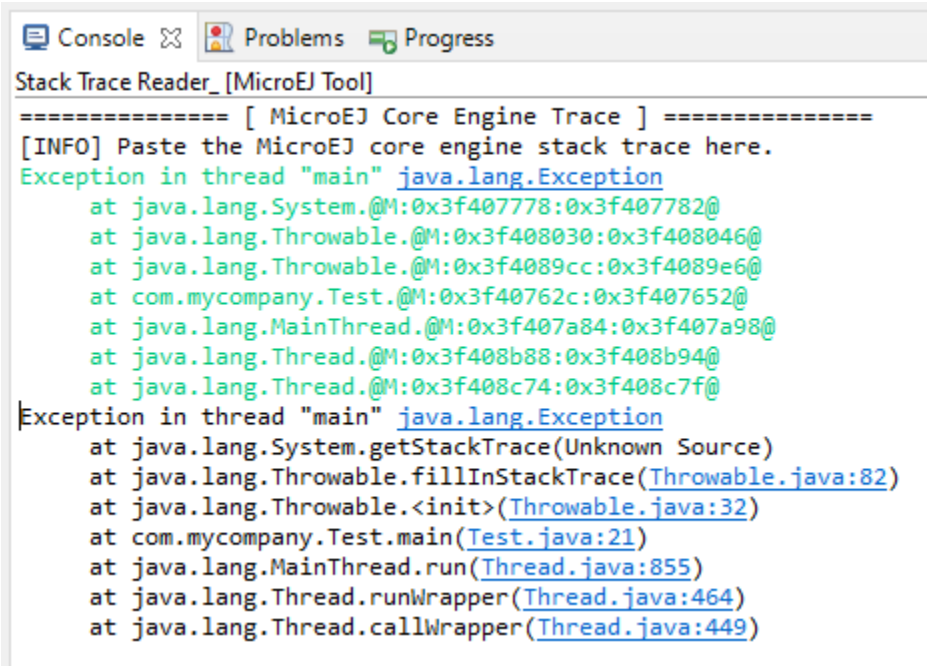


Fig. 72: Stack Trace Reader Console

Use (Sandboxed Application)

For example, write the following new line to dump the currently executed stack trace on the standard output.

```
public class MyBackgroundCode implements BackgroundService {

    @Override
    public void onStart() {
        // TODO Auto-generated method stub
        System.out.println("MyBackgroundCode: Hello World");
        new Throwable().printStackTrace();
    }
```

Fig. 73: Code to Dump a Stack Trace

To decode an application stack trace, the stack trace reader tool requires the application binary file with debug information (`application.fodbg` in the output folder). Note that the file uploaded on the device is `application.fo` (stripped version without debug information).

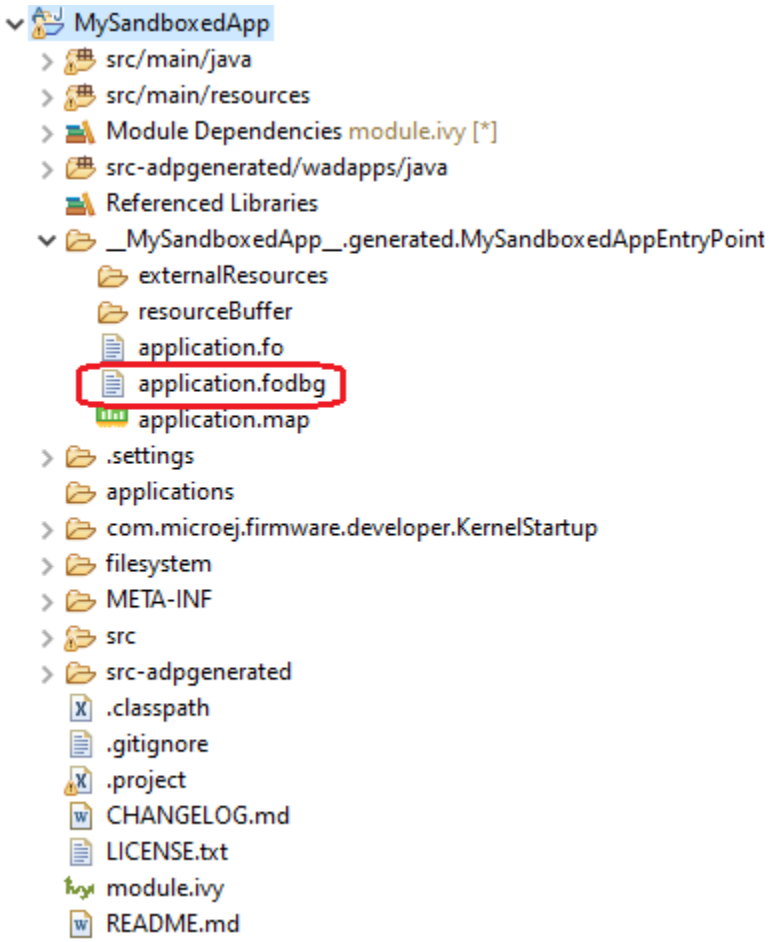


Fig. 74: Application Binary File with Debug Information

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

```
com.microej.wadapps.kf.abstractfeatureapplicationstorage INFO: Start MySandboxedApp
MyBackgroundCode: Hello World
Exception in thread "ej.wadapps.app.default" java.lang.Throwable
    at java.lang.System.@M:0x805a97c:0x805a98c@
    at java.lang.Throwable.@M:0x807b8e0:0x807b8f6@
    at java.lang.Throwable.@M:0x8076f4c:0x8076f68@
    at com.microej.example.MyBackgroundCode.@F:a5db2a4477010000d37548f1e20224d0b875cb968936fb41:0xc03800f0@@M:0xc0380b7c:0xc0380ba4@
    at Exception in thread "ej.wadapps.app.default" java/lang/Throwable
   at java/lang/System.@M:0x0805A97C:0x0805A98C@
   at java/lang/Throwable.@M:0x0807B8E0:0x0807B8F6@
   at java/lang/Throwable.@M:0x08076F4C:0x08076F68@
   at com/microej/example/MyBackgroundCode.@F:a5db2a4477010000d37548f1e20224d0b875cb968936fb41:0xC03800F0@@M:0xC0380B7C:0xC0380BA4@
   at ej/wadapps/app/BackgroundServiceProxy.@F:fa7a4551720100007378 3c876987b55b8e3aaa8e1d407fd1:0x900A6BC0@@M:0x900AB508:0x900AB515@
   at com/microej/wadapps/management/util/BackgroundsManager.@F:fa7a4551720100007378 3c876987b55b8e3aaa8e1d407fd1:0x900A6BC0@@M:0x900AA780:0x900AA792@
   at com/microej/wadapps/management/util/BackgroundsManager.@F:fa7a4551720100007378 3c876987b55b8e3aaa8e1d407fd1:0x900A6BC0@@M:0x900ABF14:0x900ABF52@
   at ej/observable/Observable.@F:fa7a4551720100007378 3c876987b55b8e3aaa8e1d407fd1:0x900A6BC0@@M:0x900ABA10:0x900ABA40@
   at com/microej/wadapps/management/util/BackgroundServicesListImpl.@F:fa7a4551720100007378 3c876987b55b8e3aaa8e1d407fd1:0x900A6BC0@@M:0x900AD864:0x900AD894@
   at ej/wadapps/management/BackgroundServicesListProxy.@F:a5db2a4477010000d37548f1e20224d0b875cb968936fb41:0xC03800F0@@M:0xC0380A28:0xC0380A36@
   at __MySandboxedApp__/generated/MySandboxedAppActivator.@F:a5db2a4477010000d37548f1e20224d0b875cb968936fb41:0xC03800F0@@M:0xC0380C54:0xC0380C82@
   at ej/components/registry/impl/AbstractRegistry.@M:0x08078E48:0x08078E72@
   at ej/components/registry/util/BundleRegistryHelper.@M:0x0806E6E8:0x0806E702@
   at __MySandboxedApp__/generated/MySandboxedAppEntryPoint.@F:a5db2a4477010000d37548f1e20224d0b875cb968936fb41:0xC03800F0@@M:0xC0380B04:0xC0380B2E@
   at ej/kf/Kernel$2.@M:0x08055858:0x08055890@
   at java/lang/Thread.@M:0x0807C4F0:0x0807C506@
   at java/lang/Thread.@M:0x0807C398:0x0807C3A4@
   at java/lang/Thread.@M:0x0807C488:0x0807C493@
```

Fig. 75: Stack Trace Output

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

Create a new MicroEJ Tool configuration. In the Execution tab, select your target platform, then select the Stack Trace Reader tool. Set an output folder in the Output folder field.

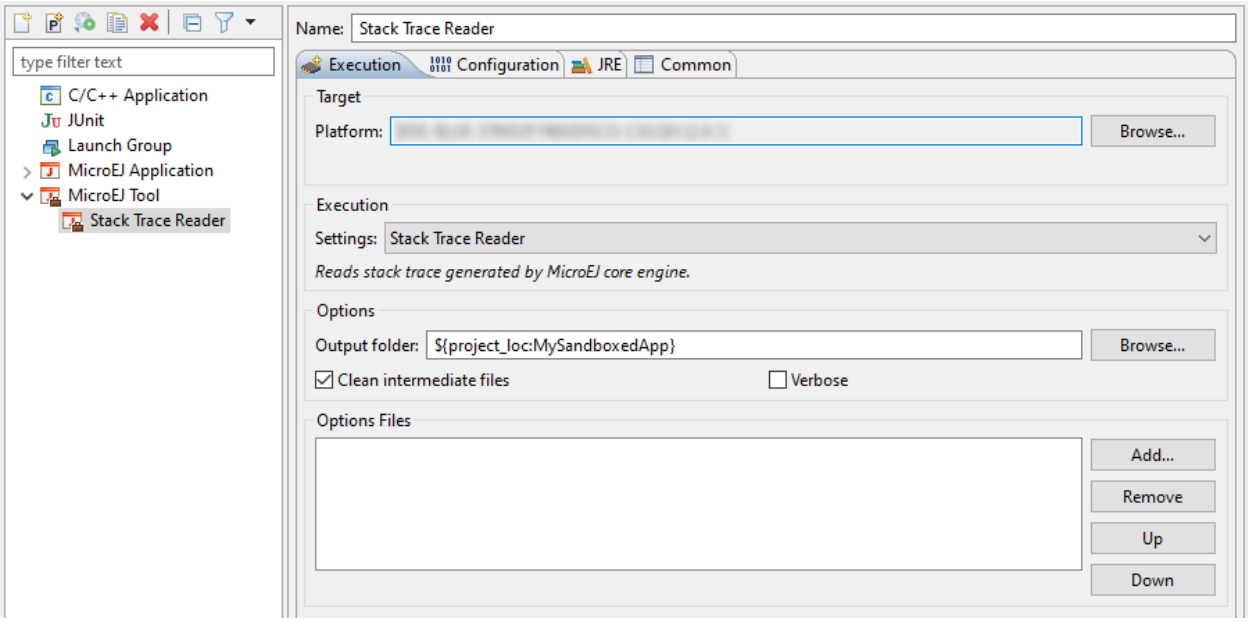


Fig. 76: Stack Trace Reader Tool Configuration (Virtual Device Selection)

In the Configuration tab, if the Kernel executable file is available to you (usually named `firmware.out` and located in your Virtual Device files), you can browse for it in the Executable file field, and then add your previously generated application binary file with debug information (`application.fodbg` in case of a Sandboxed Application) in the Additional object files field.

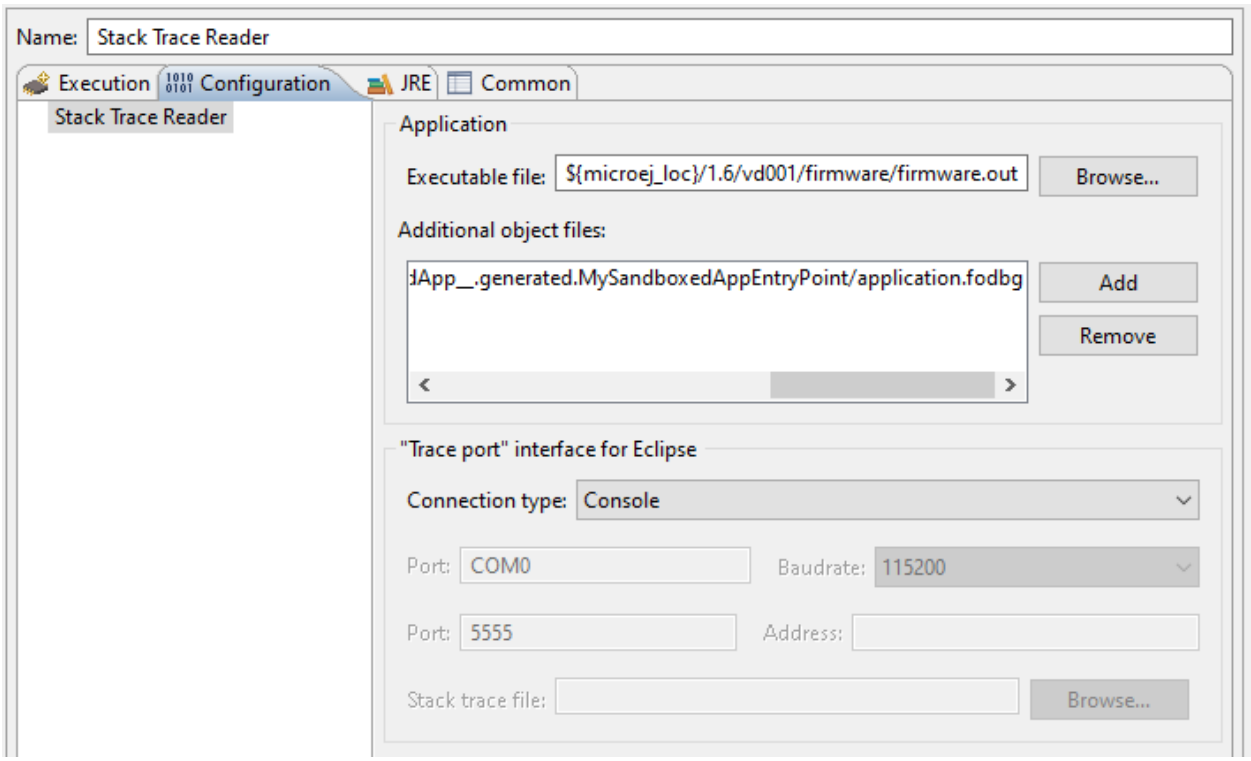


Fig. 77: Select the Kernel Executable File

To check where the Kernel executable file of your Virtual Device is located, if you have access to it, go to `Window` > `Preferences` > `MicroEJ` > `Virtual Devices`, hover over your Virtual Device in the list and wait until an information popup appears. Press `F2` to get all the informations and the path to the directory of your Virtual Device should appear in the list.

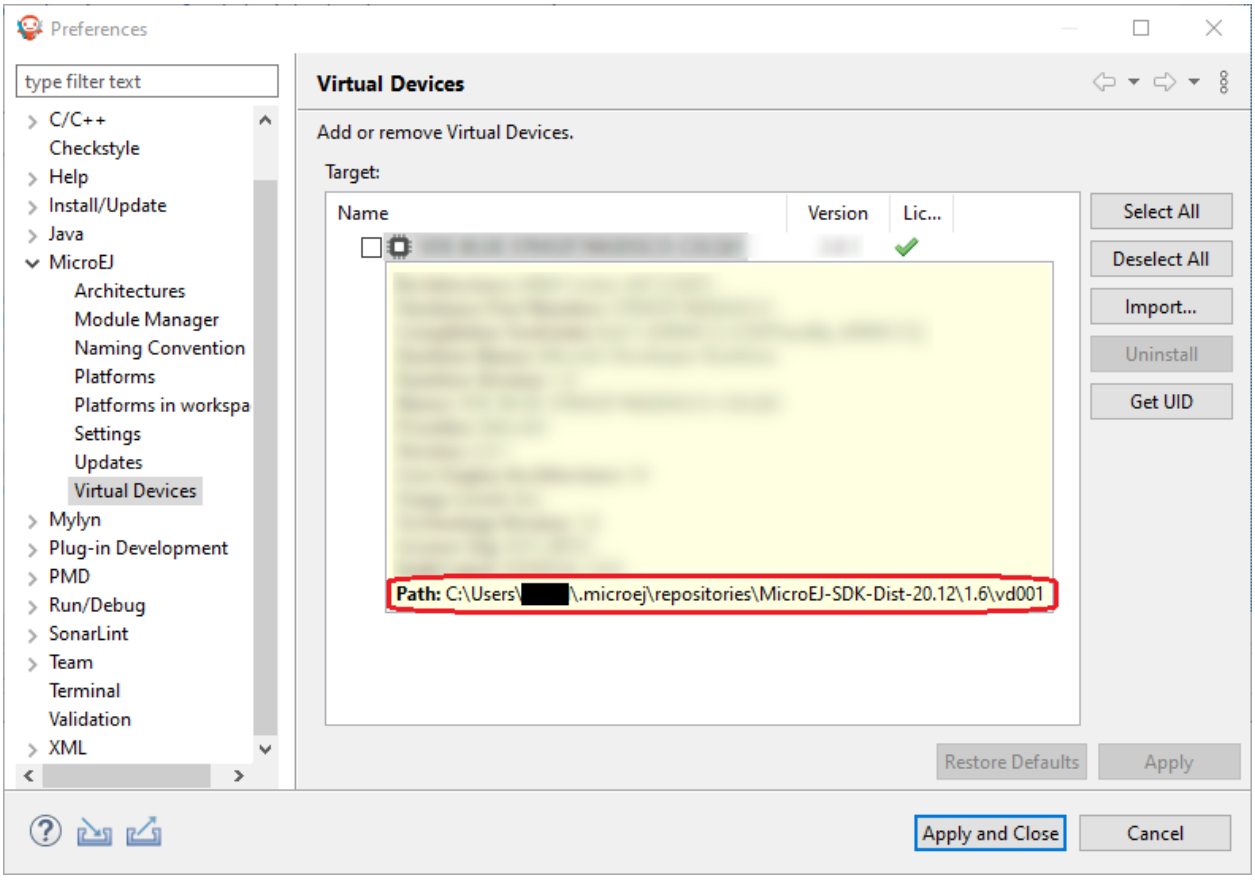


Fig. 78: Location of the Virtual Device Directory

In this directory, the Kernel executable file should be named `firmware.out` in the `/firmware` sub-directory.

If you do not have access to the Kernel executable file, you can still get some information from the Stack Trace Reader using the application binary file only. In the Configuration tab, browse the previously generated application binary file with debug information (`application.fodbg` in case of a Sandboxed Application)

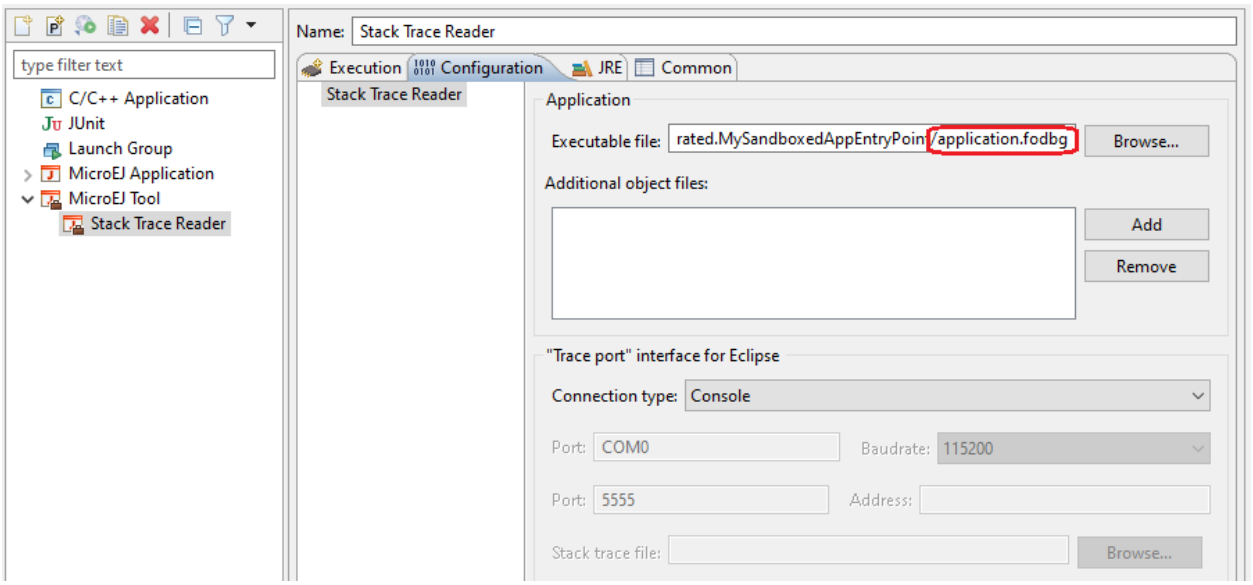


Fig. 79: Stack Trace Reader Tool Configuration (Sandboxed Application)

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

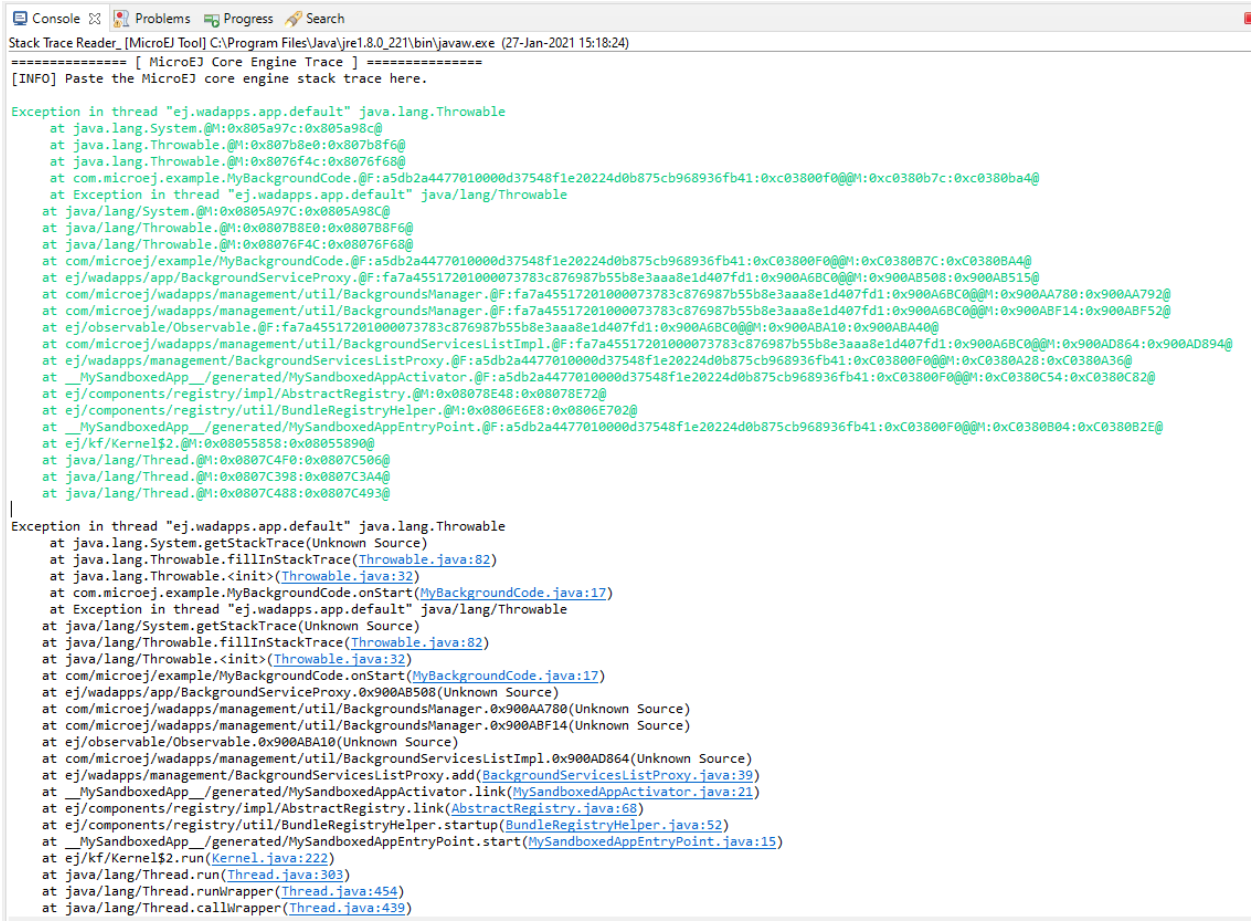


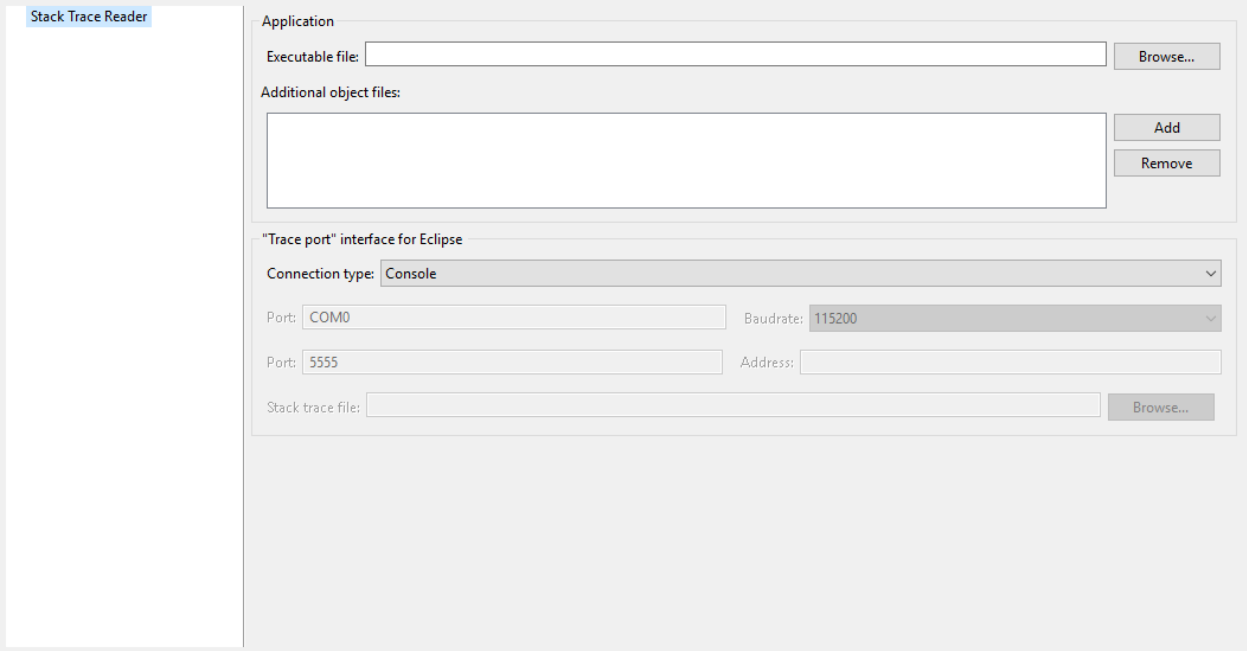
Fig. 80: Stack Trace Reader Console

Other debug information files can be appended using the `Additional object files` option.

Stack Trace Reader Options

The following section explains MicroEJ tool options.

Category: Stack Trace Reader



Group: Application

Option(browse): Executable file

Option Name: `application.file`

Default value: `(empty)`

Description:

Specify the full path of a full linked elf file.

Option(list): Additional object files

Option Name: `additional.application.files`

Default value: `(empty)`

Group: “Trace port” interface for Eclipse

Description:

This group describes the hardware link between the device and the PC.

Option(combo): Connection type

Option Name: `proxy.connection.connection.type`

Default value: `Console`

Available values:

`Uart (COM)`

`Socket`

`File`

`Console`

Description:

Specify the connection type between the device and PC.

Option(text): Port

Option Name: `pcboardconnection.usart.pc.port`

Default value: `COM0`

Description:

Format: `port name`

Specifies the PC COM port:

Windows - `COM1` , `COM2` , `...` , `COM*n*`

Linux - `/dev/ttyS0` , `/dev/ttyS1` , `...` , `/dev/ttyS*n*`

Option(combo): Baudrate

Option Name: `pcboardconnection.usart.pc.baudrate`

Default value: `115200`

Available values:

`9600`

`38400`

`57600`

`115200`

Description:

Defines the COM baudrate for PC-Device communication.

Option(text): Port

Option Name: `pcboardconnection.socket.port`

Default value: `5555`

Description:

IP port.

Option(text): Address

Option Name: `pcboardconnection.socket.address`

Default value: `(empty)`

Description:

IP address, on the form A.B.C.D.

Option(browse): Stack trace file

Option Name: `pcboardconnection.file.path`

Default value: `(empty)`

3.9.3 Code Coverage Analyzer

Principle

The Simulator features an option to output .cc (Code Coverage) files that represent the use rate of functions of an application. It traces how the opcodes are really executed.

Functional Description

The Code Coverage Analyzer scans the output .cc files, and outputs an HTML report to ease the analysis of methods coverage. The HTML report is available in a folder named htmlReport in the same folder as the .cc files generated by enabling the *Code Coverage option* .

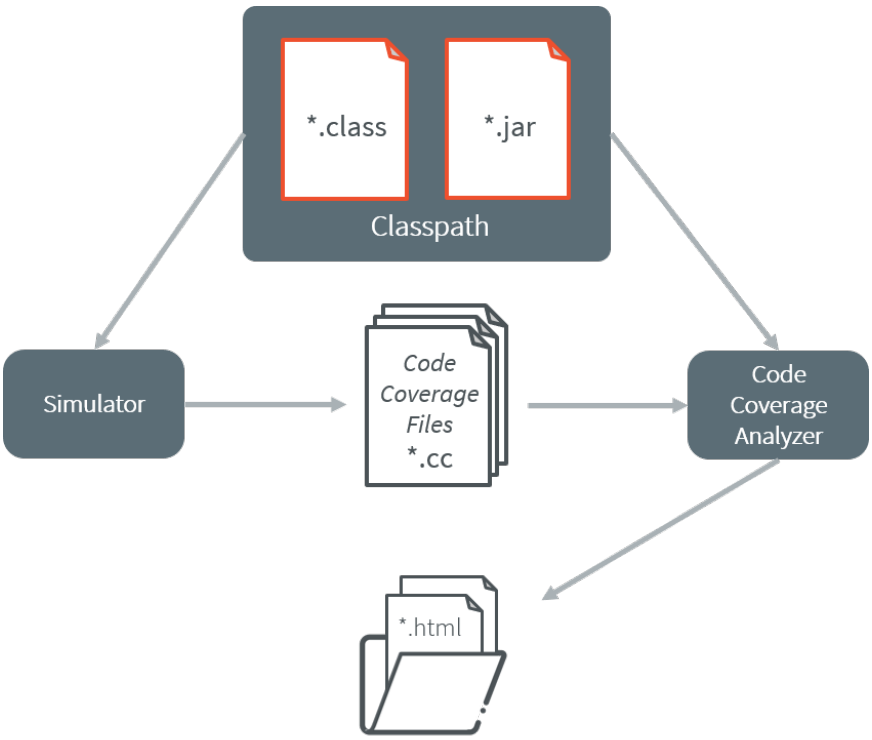


Fig. 81: Code Coverage Analyzer Process

Dependencies

In order to work properly, the Code Coverage Analyzer should input the .cc files. The .cc files relay the classpath used during the execution of the Simulator to the Code Coverage Analyzer. Therefore the classpath is considered to be a dependency of the Code Coverage Analyzer.

Installation

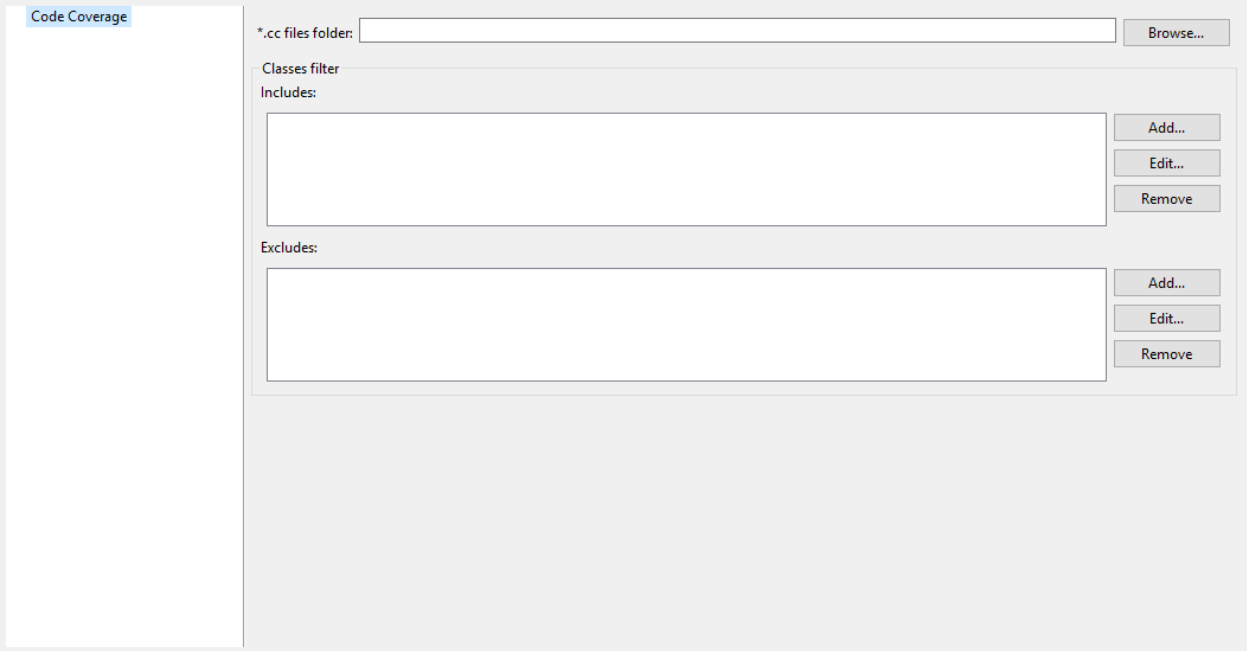
This tool is a built-in Architecture tool.

Use

A MicroEJ tool is available to launch the Code Coverage Analyzer tool. The tool name is Code Coverage Analyzer.

Two levels of code analysis are provided, the Java level and the bytecode level. Also provided is a view of the fully or partially covered classes and methods. From the HTML report index, just use hyperlinks to navigate into the report and source / bytecode level code.

Category: Code Coverage



Option(browse): *.cc files folder

Option Name: `cc.dir`

Default value: `(empty)`

Description:

Specify a folder which contains the cc files to process (*.cc).

Group: Classes filter

Option(list): Includes

Option Name: `cc.includes`

Default value: `(empty)`

Description:

List packages and classes to include to code coverage report. If no package/class is specified, all classes found in the project classpath will be analyzed.

Examples:

`packageA.packageB.*` : includes all classes which are in package `packageA.packageB`

`packageA.packageB.className` : includes the class `packageA.packageB.className`

Option(list): Excludes

Option Name: `cc.excludes`

Default value: `(empty)`

Description:

List packages and classes to exclude to code coverage report. If no package/class is specified, all classes found in the project classpath will be analyzed.

Examples:

`packageA.packageB.*` : excludes all classes which are in package `packageA.packageB`

`packageA.packageB.className` : excludes the class `packageA.packageB.className`

3.9.4 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 in the application output folder.

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 plugin is able to open dump files. It helps you analyze their contents thanks to the following features:

- memory leaks detection
- objects instances browse
- heap usage optimization (using immortal or immutable objects)

The Heap

The heap is a memory area used to hold Java objects created at runtime. Objects persist in the heap until they are garbage collected. An object becomes eligible for garbage collection when there are no longer any references to it from other objects.

Heap Dump

A heap dump is an XML file that provides a snapshot of the heap contents at the moment the file is created. It contains a list of all the instances of both class and array types that exist in the heap. For each instance, it records:

- The time at which the instance was created
- The thread that created it
- The method that created it

For instances of class types, it also records:

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

For instances of array types, it also records:

- The type of the contents of the array

- The contents of the array

For each referenced class type, it records the values in the static fields of the class.

Heap Analyzer Tools

The Heap Analyzer is an Eclipse plugin that adds three tools to the MicroEJ environment.

Tool name	Number of input files	Purpose
Heap Viewer	1	Shows what instances are in the heap, when they were created, and attempts to identify problem areas
Progressive Heap Usage	1 or more	Shows how the number of instances in the heap has changed over time
Compare	2	Compares two heap dumps, showing which objects were created, or garbage collected, or have changed values

Heap Dumper

When the Heap Dumper option is activated, the garbage collector process ends by performing a dump file that represents a snapshot of the heap at this moment. To generate such dump files, you must explicitly call the `System.gc()` method in your code.

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:

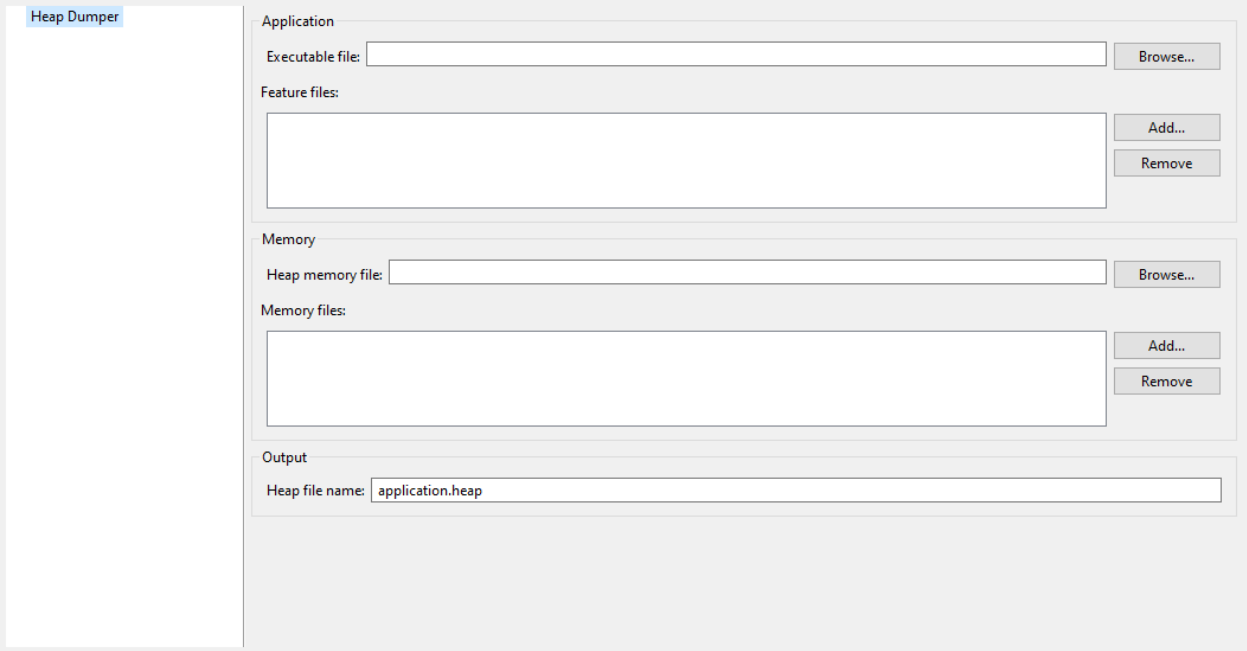
- the class
- the values in the instance's non-static fields

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): Feature files

Option Name: `additional.application.filenames`

Default value: `(empty)`

Description:

Specify the full path of Feature files with debug information (`.fodbg` files).

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.

Option(list): Memory files

Option Name: `additional.memory.filenames`

Default value: `(empty)`

Description:

Specify the full path of additional memory files in Intel Hex format (Installed Feature areas, Dynamic Features table, ...).

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.

[1] Although this is an Eclipse 'editor', it is not possible to edit the contents of the heap dump.

Outline View

The Outline view shows a list of all the types in the heap dump, and for each type shows a list of the instances of that type. When an instance is selected it also shows a list of the instances that refer to that instance. The Outline view is opened automatically when an Heap Viewer is opened.

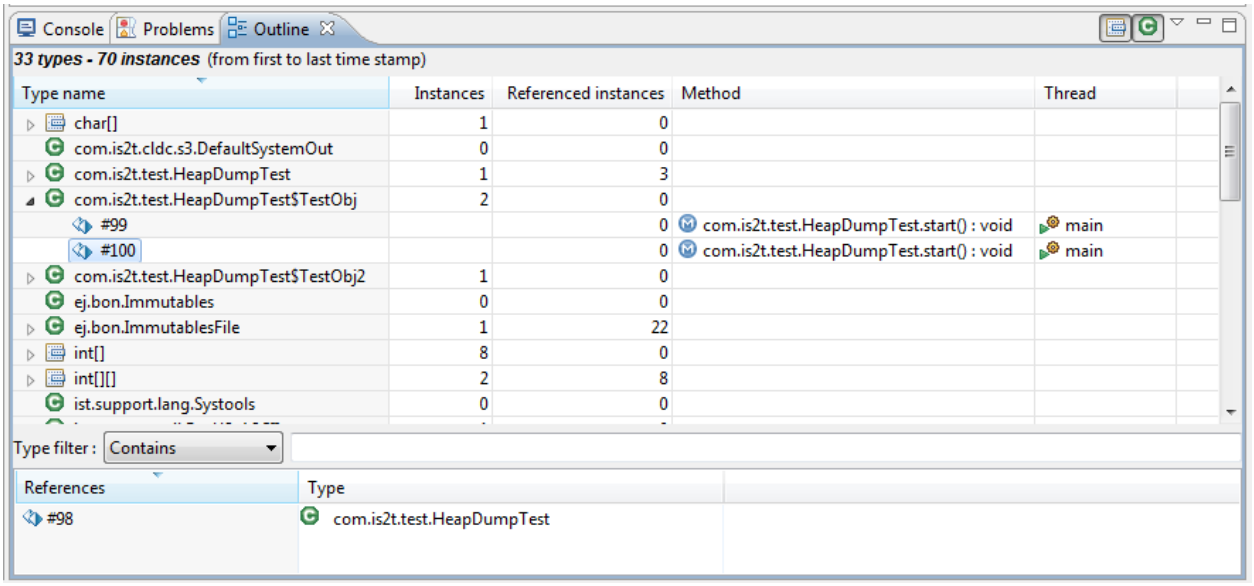


Fig. 82: Outline View

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.

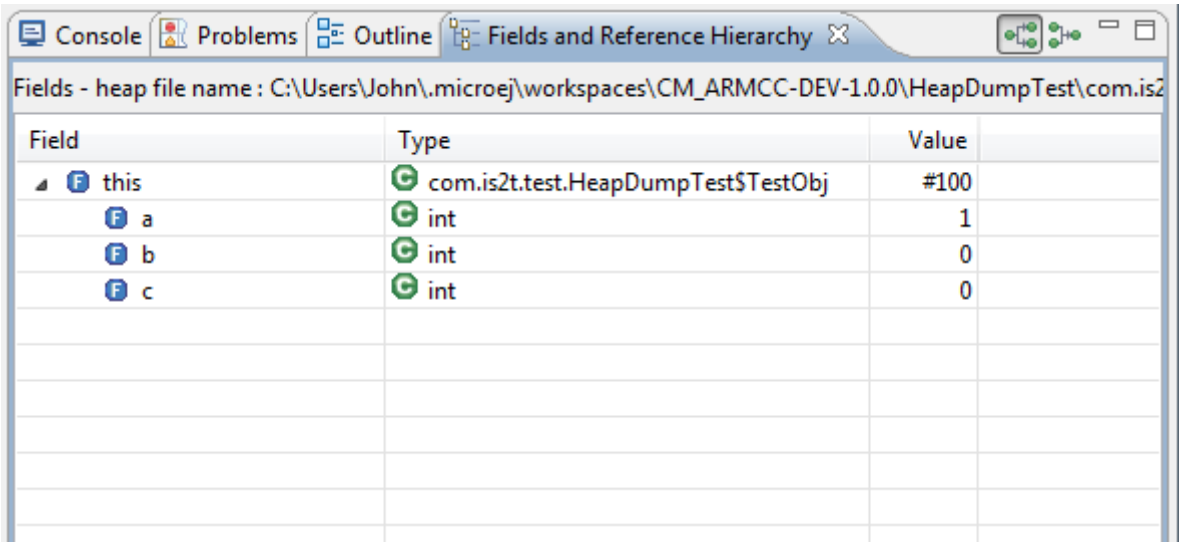


Fig. 83: Instance Browser View - Fields mode

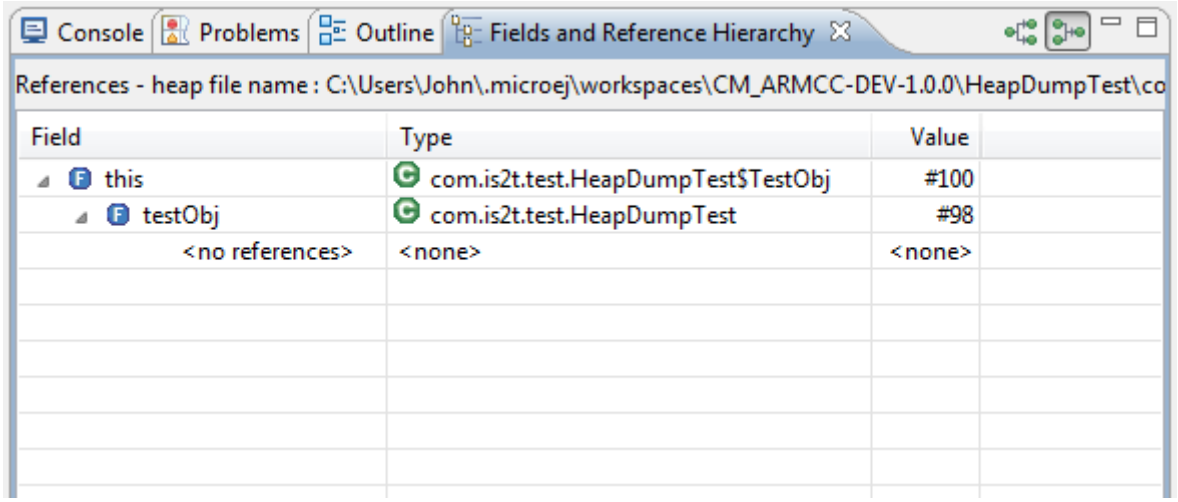


Fig. 84: Instance Browser View - References mode

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. 85: Heap Viewer - Heap Usage Tab

Clicking on the graph area in a chart restricts the Outline view to just the types and instances that were created during the selected time period. Clicking on a chart but outside of the graph area restores the Outline view to showing all types and instances[2].

The button Generate graphViz file in the top-right corner of the Heap Usage page generates a file compatible with graphviz (www.graphviz.org).

The section *Heap Usage Monitoring* shows how to compute the maximum heap usage.

[2] The Outline can also be restored by selecting the All types and instances option on the drop-down menu at the top of the Outline view.

Dominator Tree Tab

The Dominator tree page of the Heap Viewer allows the user to browse the instance reference tree which contains the greatest number of instances. This can be useful when investigating a memory leak because this tree is likely to contain the instances that should have been garbage collected.

The page contains two tree viewers. The top viewer shows the instances that make up the tree, starting with the root. The left column shows the ids of the instances – initially just the root instance is shown. The Shallow instances column shows the number of instances directly referenced by the instance, and the Referenced instances column shows the total number of instances below this point in the tree (all descendants).

The bottom viewer groups the instances that make up the tree either according to their type, the thread that created them, or the method that created them.

Double-clicking an instance in either viewer opens the Instance Browser view (if not already open) and shows details of the instance in that view.

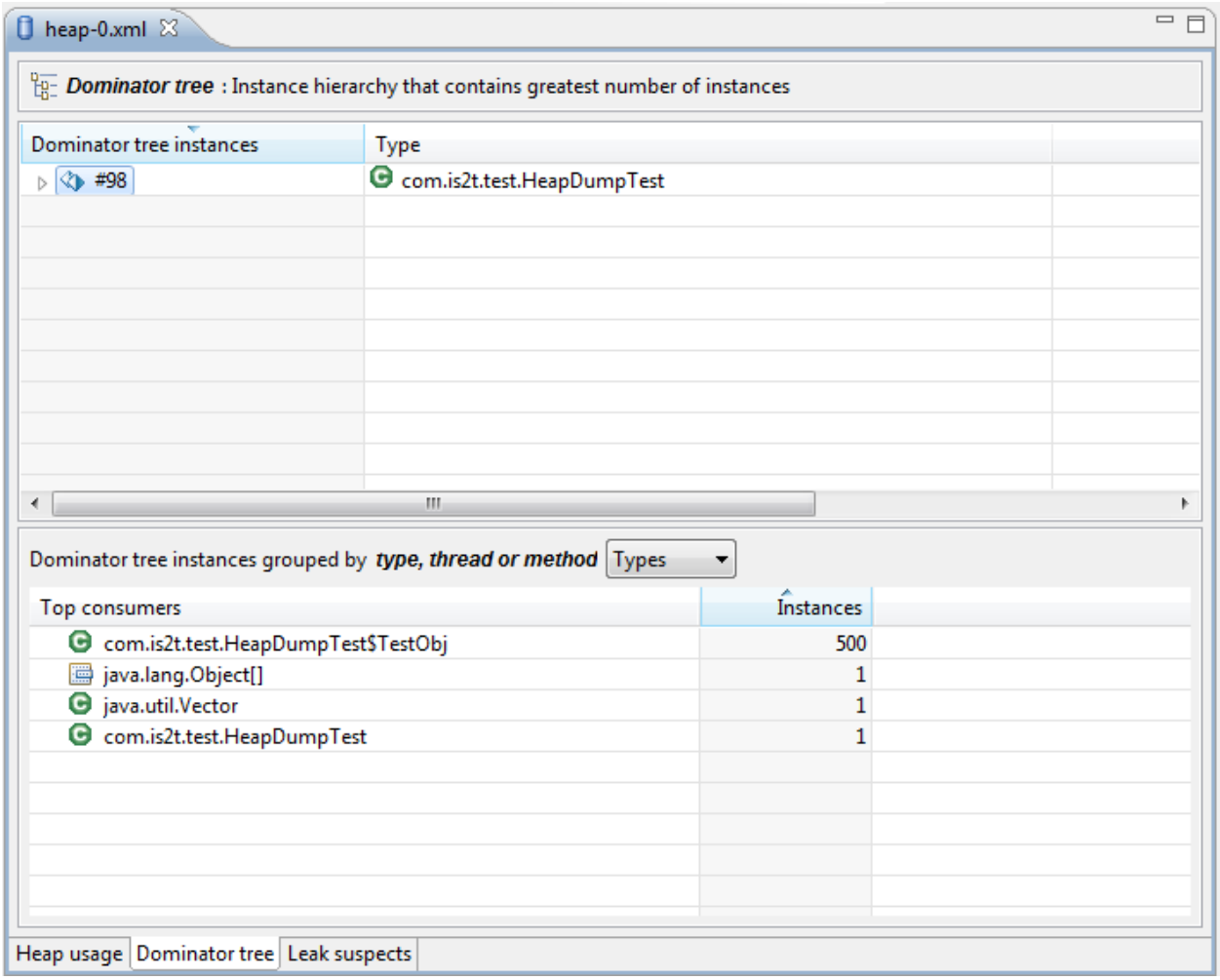


Fig. 86: Heap Viewer - Dominator Tree Tab

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.

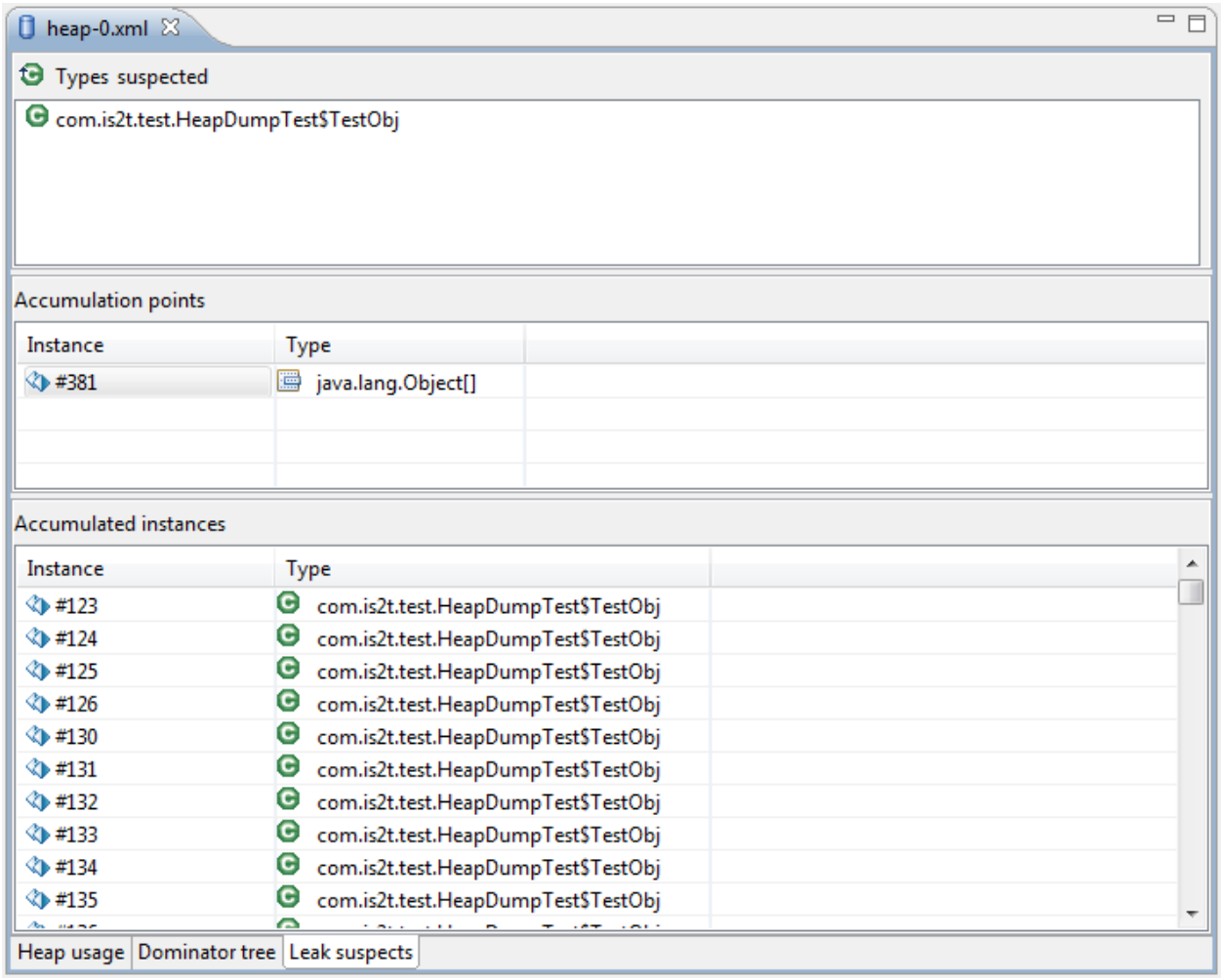


Fig. 87: Heap Viewer - Leak Suspects Tab

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.

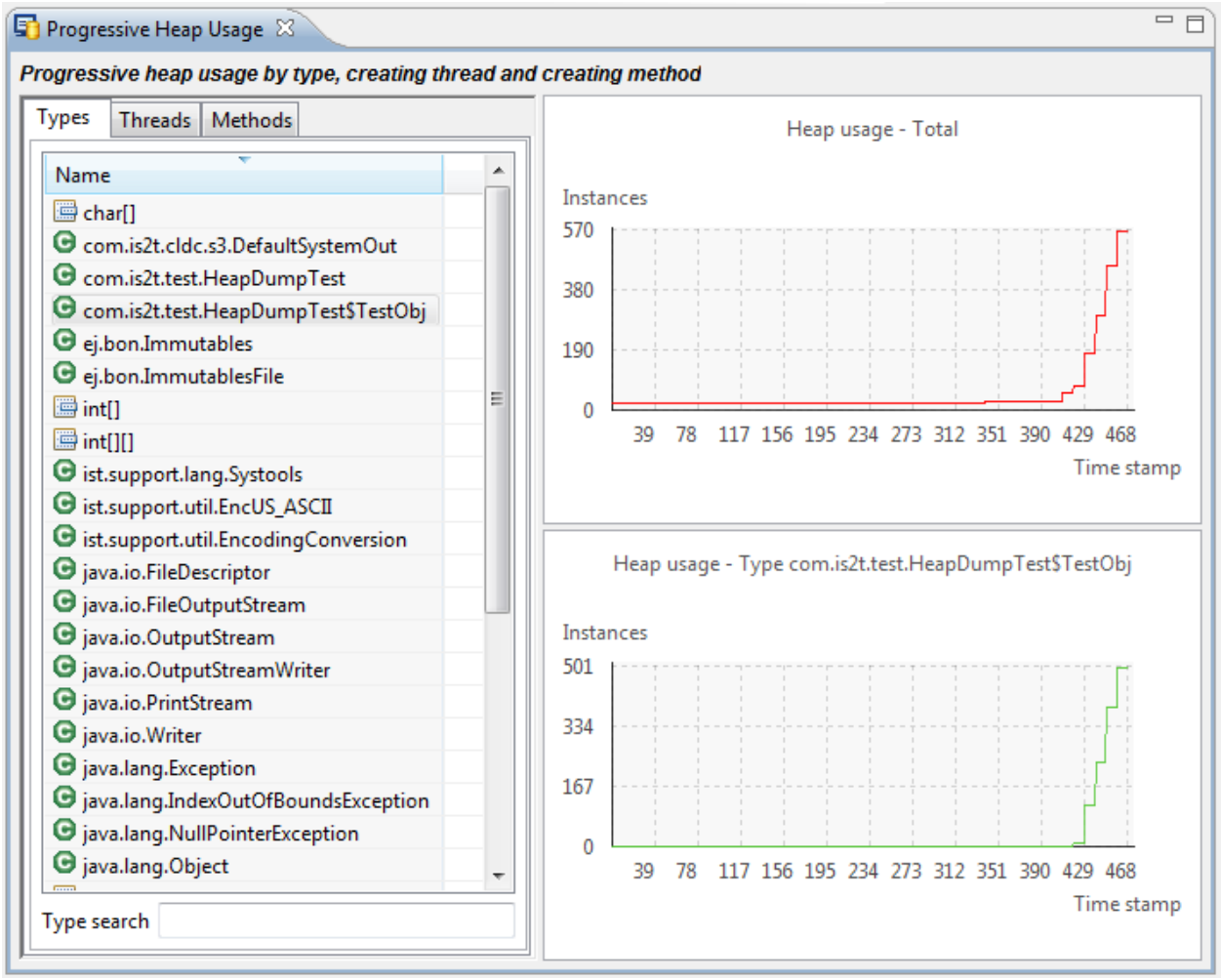


Fig. 88: Progressive Heap Usage

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.

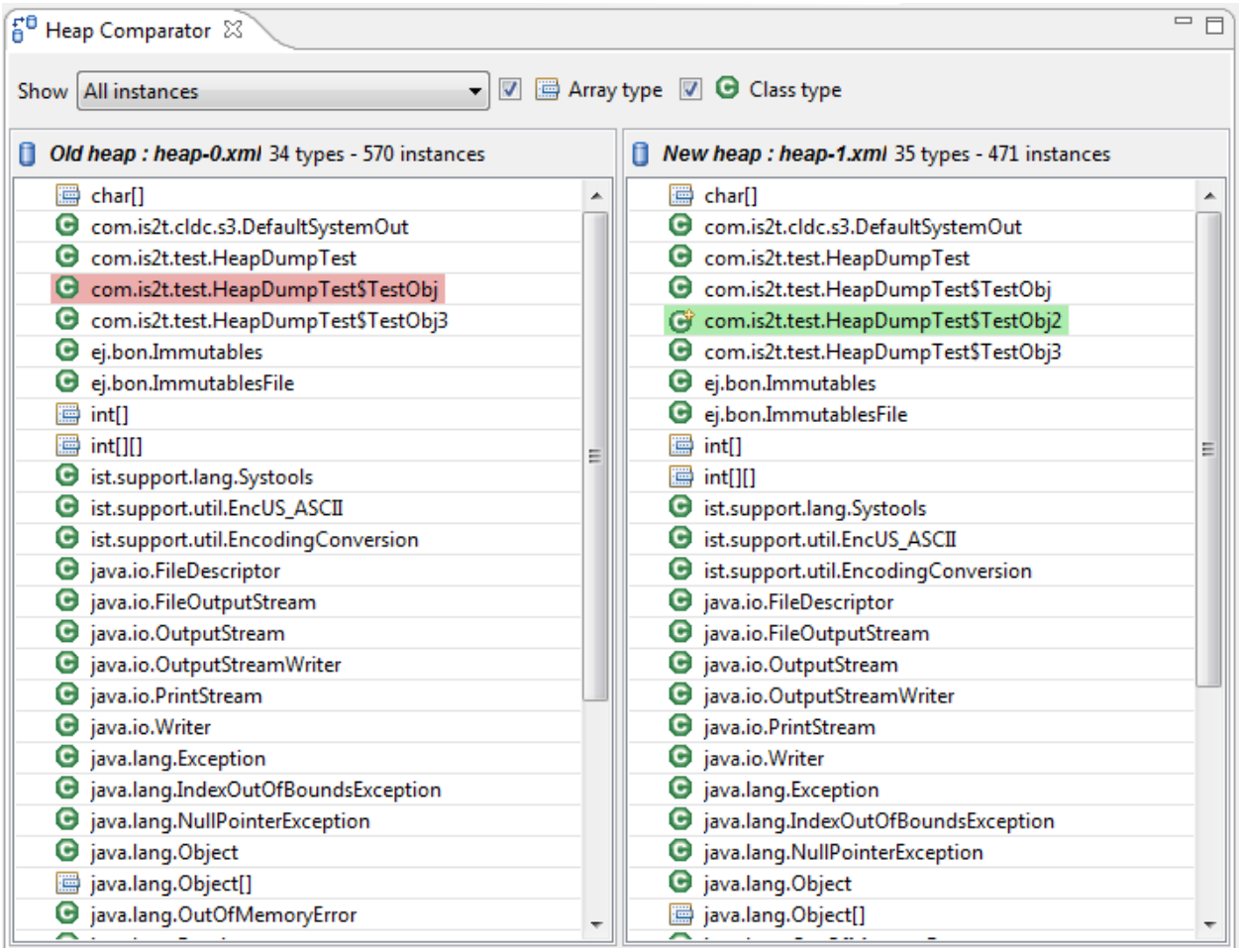


Fig. 89: Compare Heap Dumps

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.

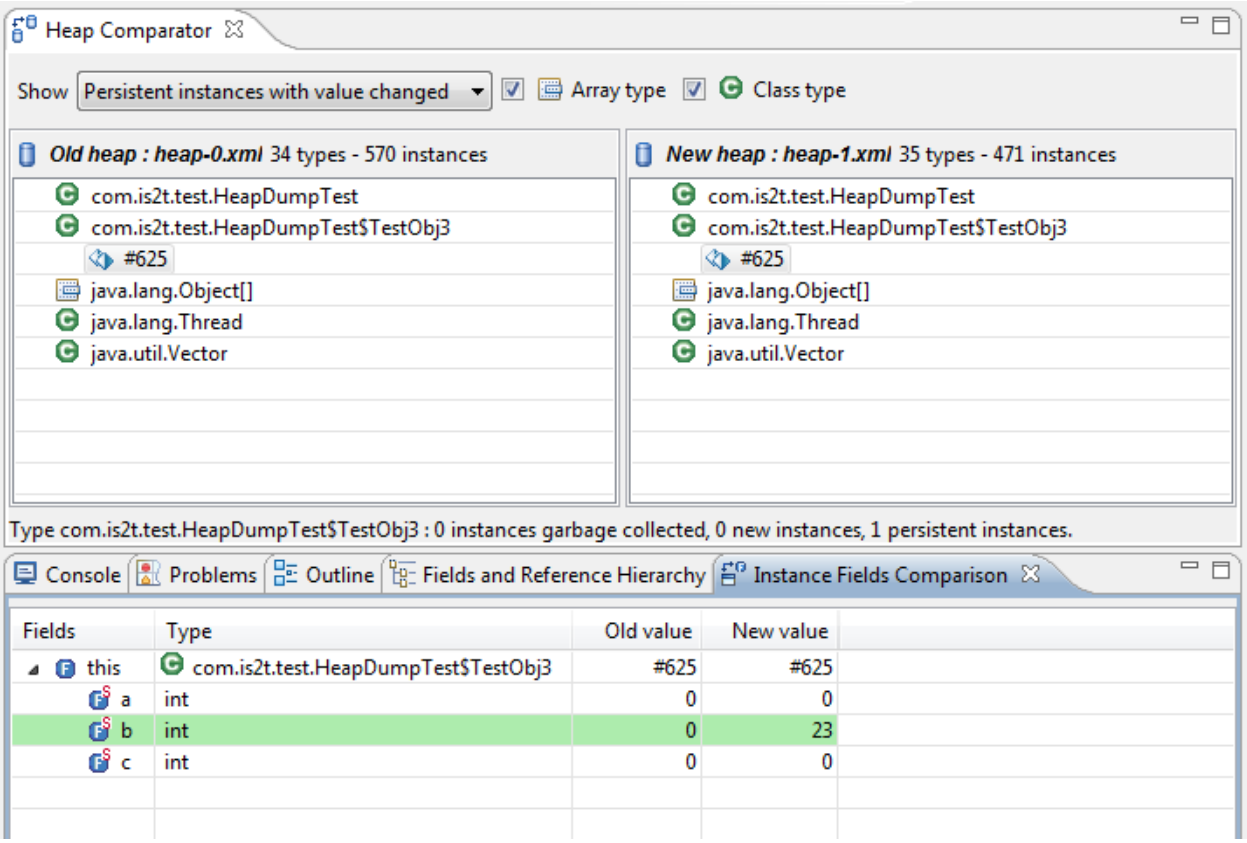


Fig. 90: Instance Fields Comparison view

3.9.5 Serial to Socket Transmitter

Principle

The MicroEJ serialToSocketTransmitter is a piece of software which transfers all bytes from a serial port to a tcp client or tcp server.

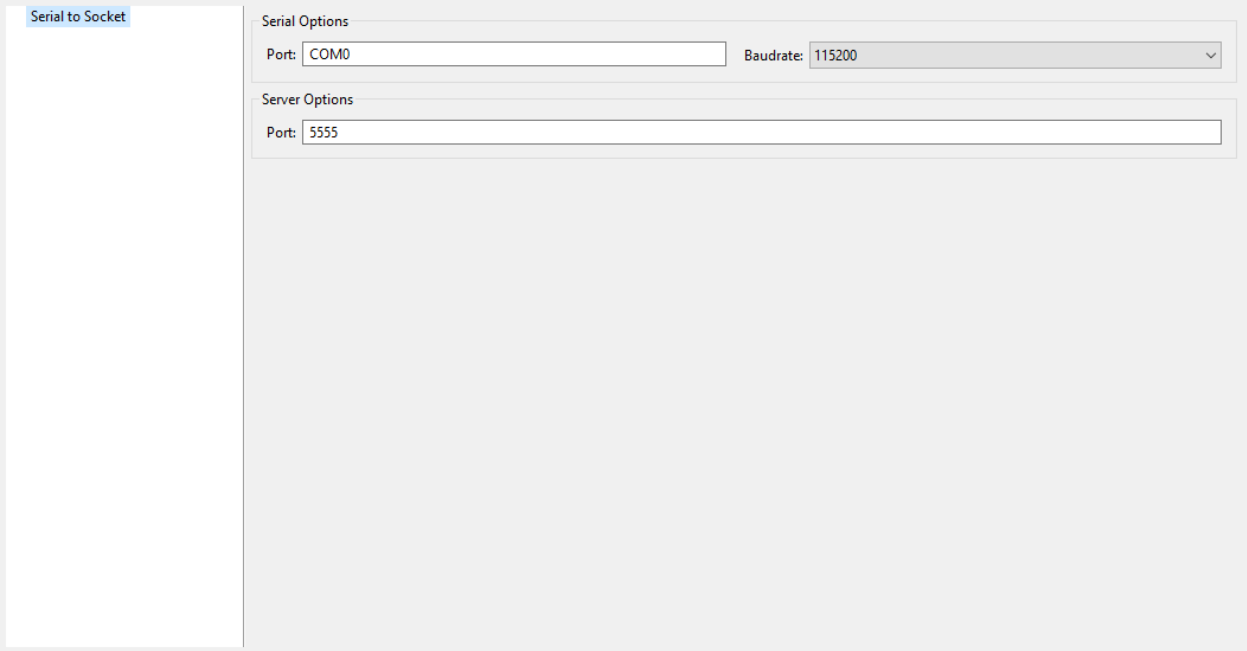
Installation

This tool is a built-in Architecture tool.

Use

This chapter explains MicroEJ tool options.

Category: Serial to Socket



Group: Serial Options

Option(text): Port

Option Name: `serail.to.socket.comm.port`

Default value: `COM0`

Description: Defines the COM port:

Windows - `COM1` , `COM2` , `...` , `COM*n*`

Linux - `/dev/ttyS0` , `/dev/ttyUSB0` , `...` , `/dev/ttyS*n*` , `/dev/ttyUSB*n*`

Option(combo): Baudrate

Option Name: `serail.to.socket.comm.baudrate`

Default value: `115200`

Available values:

`9600`

`38400`

`57600`

`115200`

Description: Defines the COM baudrate.

Group: Server Options

Option(text): Port

Option Name: `serail.to.socket.server.port`

Default value: `5555`

Description: Defines the server IP port.

3.9.6 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

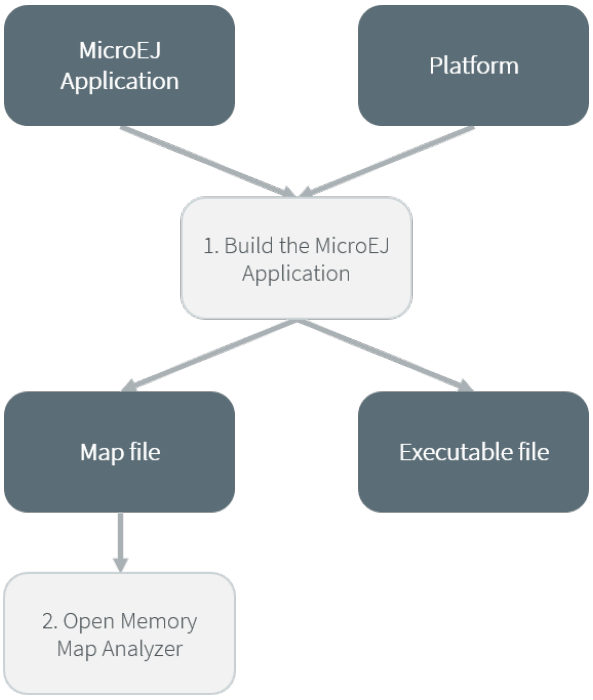


Fig. 91: Memory Map Analyzer Process

In addition to the executable file, the MicroEJ Platform generates a map file. Double click on this file to open the Memory Map Analyzer.

Dependencies

No dependency.

Installation

This tool is a built-in SDK tool.

Use

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

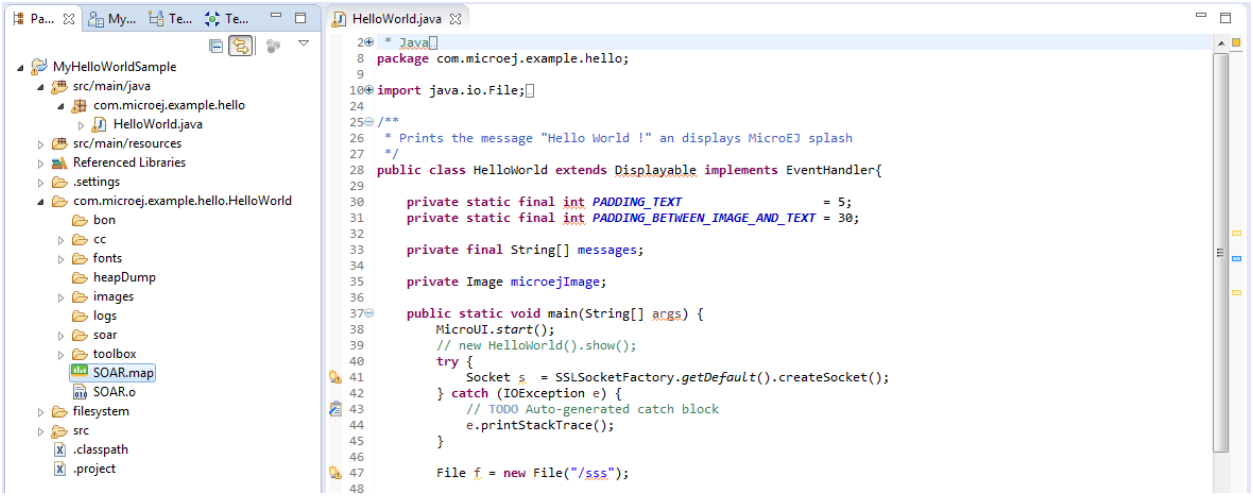


Fig. 92: Retrieve Map File

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.

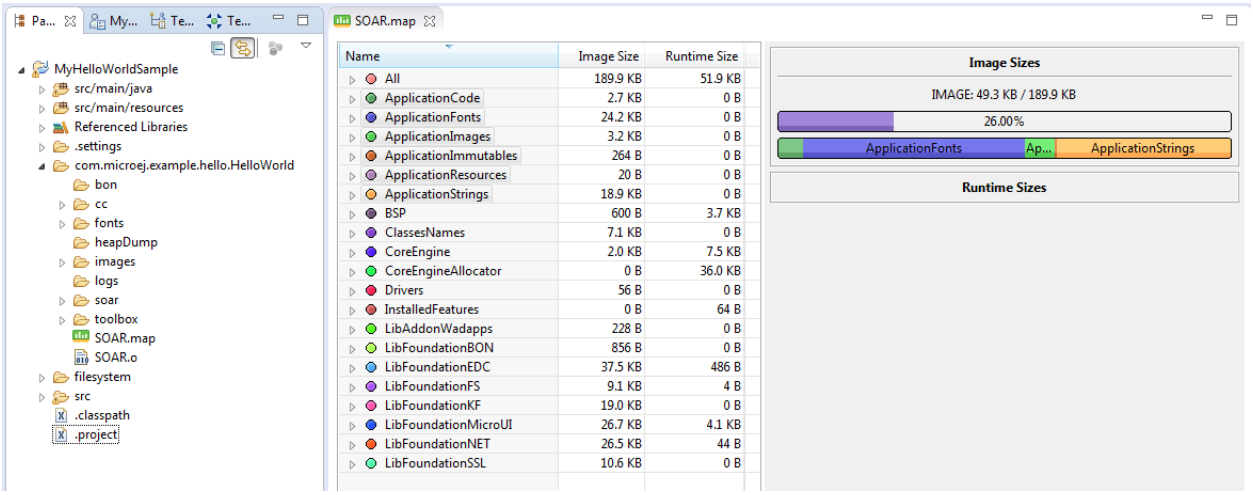


Fig. 93: Consult Full Memory

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.

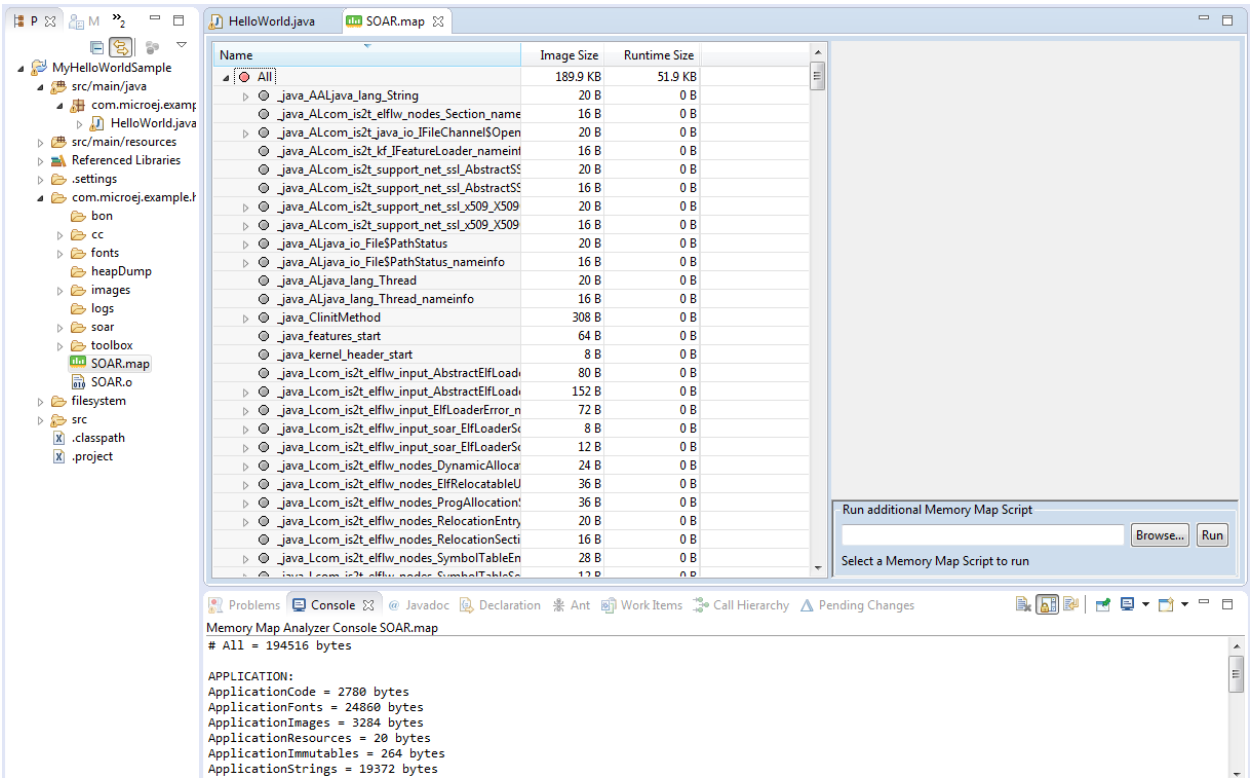


Fig. 94: Detailed View

3.9.7 Null Analysis

NullPointerException thrown at runtime is one of the most common causes for failure of Java programs. The Null Analysis tool can detect such programming errors (misuse of potential `null` Java values) at compile-time.

The following example of code shows a typical Null Analysis error detection in MicroEJ SDK.

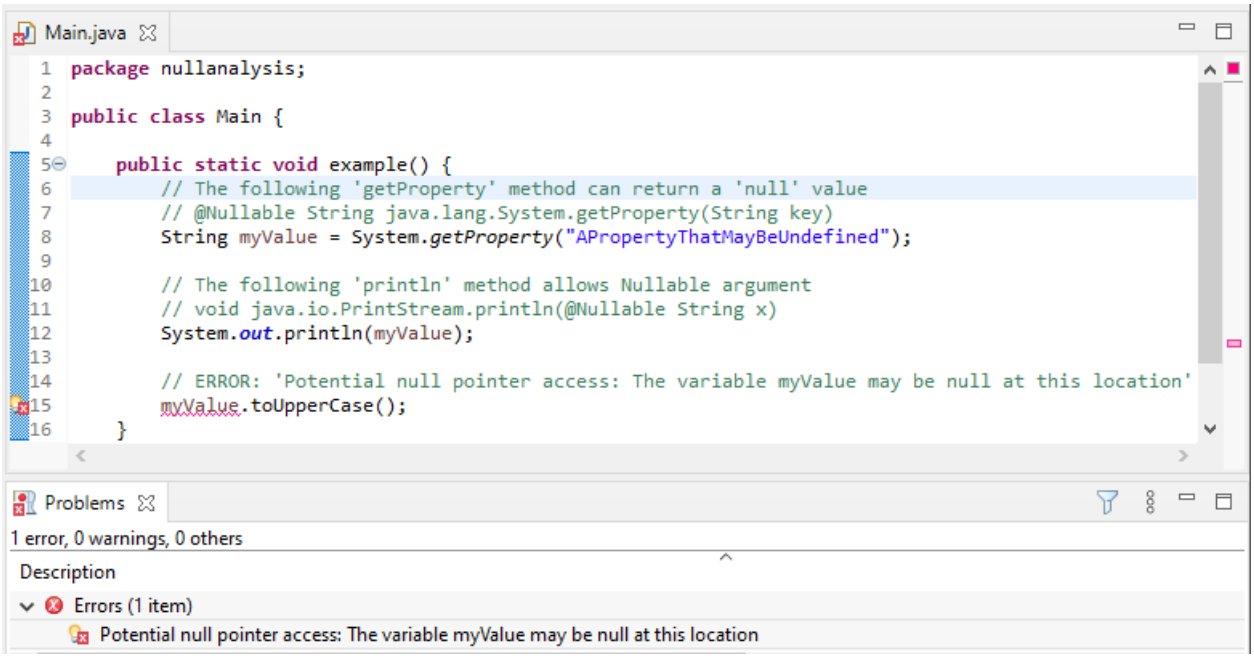


Fig. 95: Example of Null Analysis Detection

Principle

The Null Analysis tool is based on Java annotations. Each Java field, method parameter and method return value must be marked to indicate whether it can be `null` or not.

Once the Java code is annotated, *module projects* must be configured to enable Null Analysis detection in MicroEJ SDK.

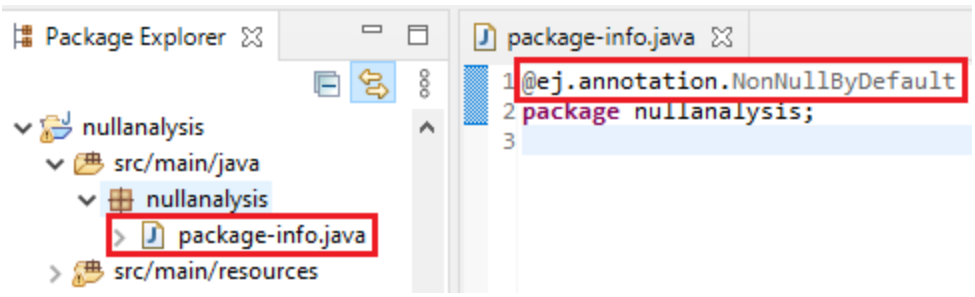
Java Code Annotation

MicroEJ defines its own annotations:

- @NonNullByDefault: Indicates that all fields, method return values or parameters can never be null in the annotated package or type. This rule can be overridden on each element by using the Nullable annotation.
- @Nullable: Indicates that a field, local variable, method return value or parameter can be null.
- @NonNull: Indicates that a field, local variable, method return value or parameter can never be null.

MicroEJ recommends to annotate the Java code as follows:

- In each Java package, create a `package-info.java` file and annotate the Java package with `@NonNullByDefault`. This is a common good practice to deal with non `null` elements by default to avoid undesired NullPointerException. It enforces the behavior which is already widely outlined in Java coding rules.



- In each Java type, annotate all fields, methods return values and parameters that can be null with `@Nullable`. Usually, this information is already available as textual information in the field or method Javadoc comment. The following example of code shows where annotations must be placed:

```
@Nullable
public Object thisFieldCanBeNull;

@Nullable
public Object thisMethodCanReturnNull() {
    return null;
}

public void thisMethodParameterCanBeNull(@Nullable Object param) {

}
```

Module Project Configuration

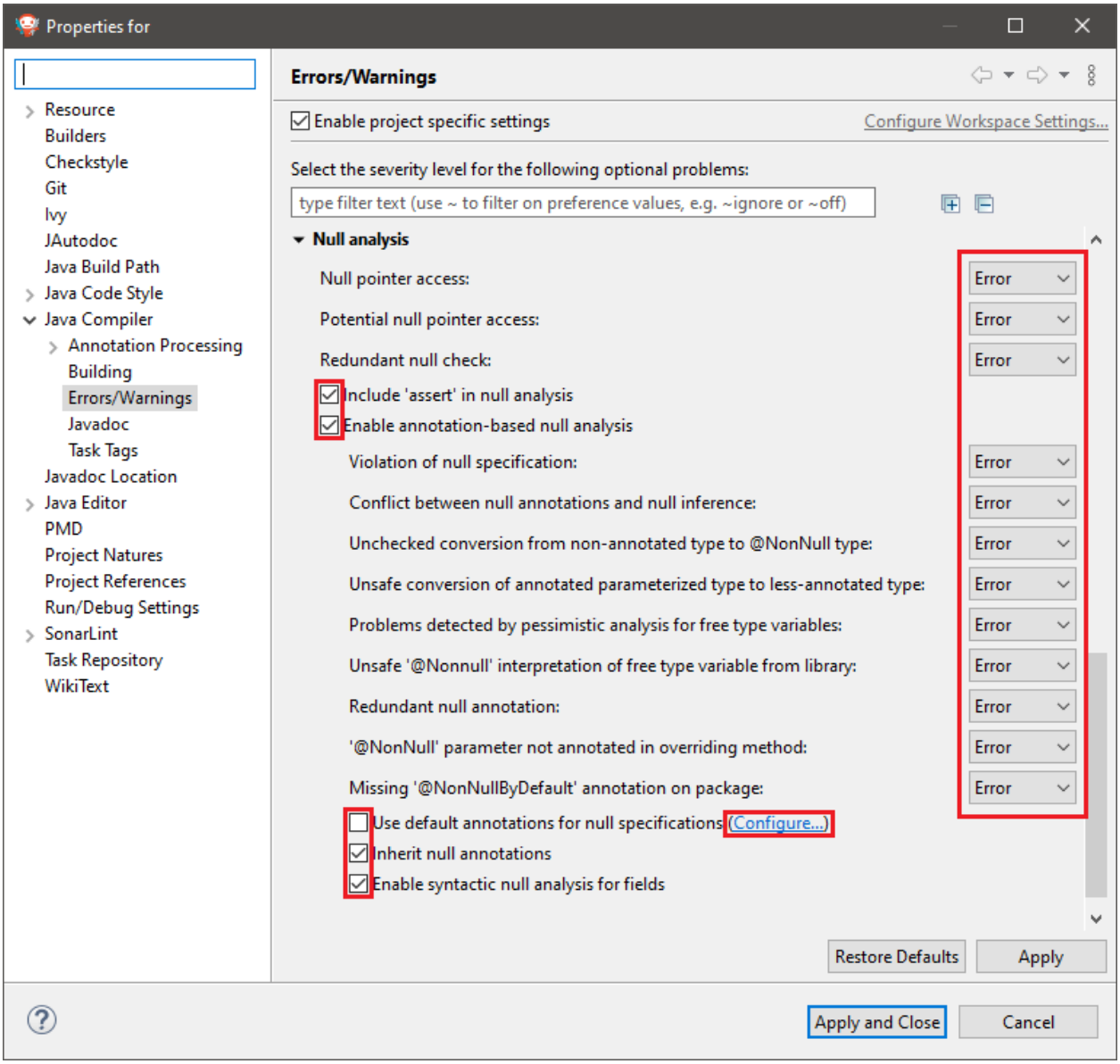
Requirements

EDC-1.3.3 or higher is required when MicroEJ SDK `5.3.0` or higher is used. See EDC 1.3.3 Changelog for more details.

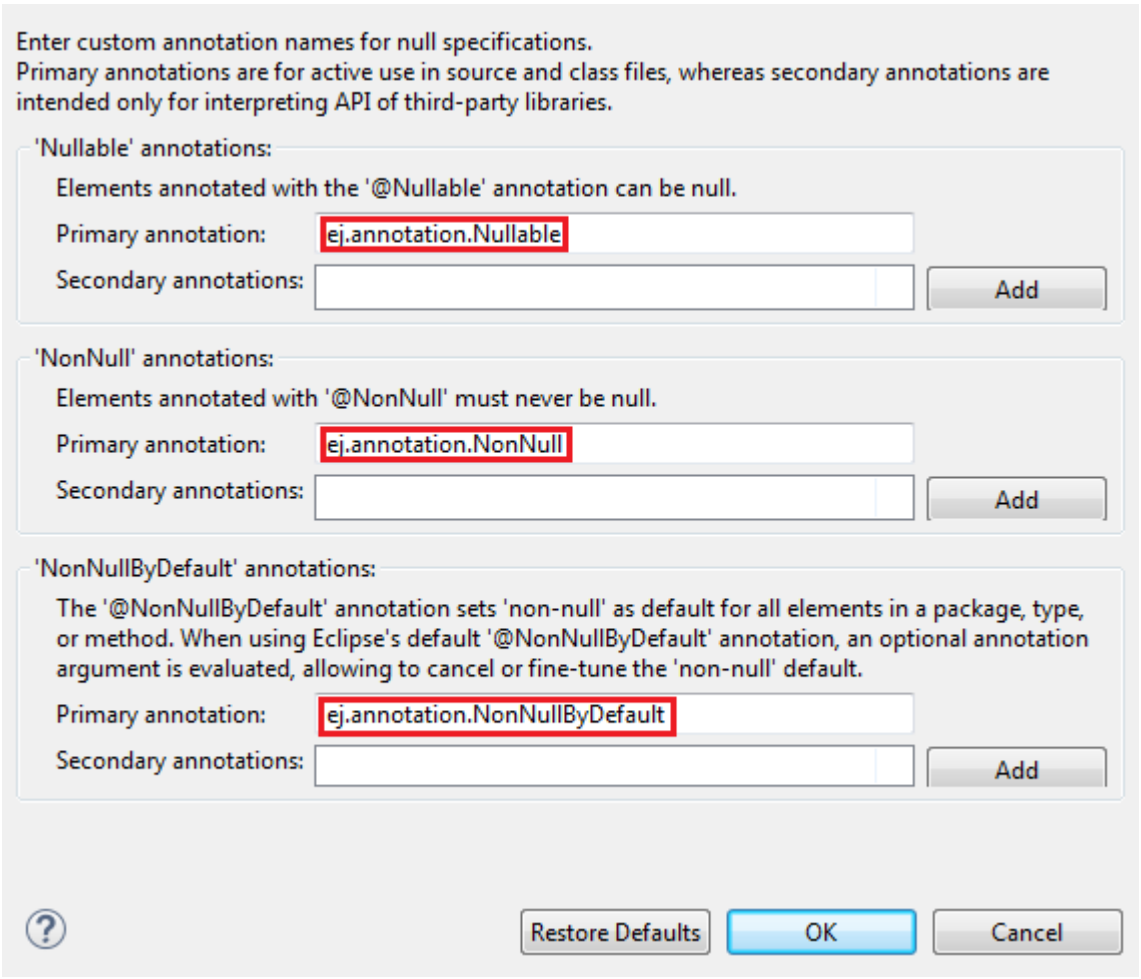
Project configuration

To enable the Null Analysis tool, a *module project* must be configured as follows:

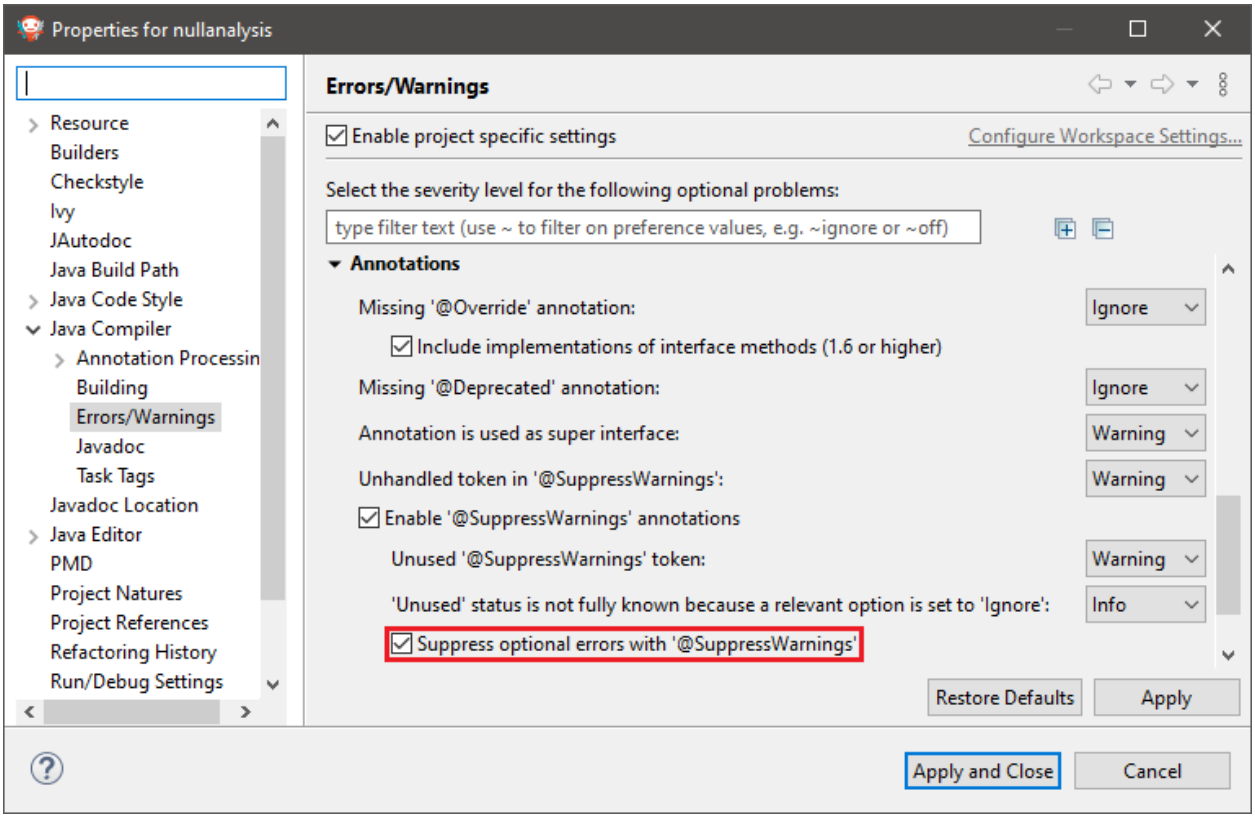
- In the Package Explorer, right-click on the module project and select `Properties`,
- Navigate to `Java Compiler` > `Errors/Warnings`,
- In the `Null analysis` section, configure options as follows:



- Click on the Configure… link to configure MicroEJ annotations:
  - `ej.annotation.Nullable`
  - `ej.annotation.NonNull`
  - `ej.annotation.NonNullByDefault`



- In the `Annotations` section, check `Suppress optional errors with '@SuppressWarnings'` option:



This option allows to fully ignore Null Analysis errors in advanced cases using `@SuppressWarnings("null")` annotation.

If you have multiple projects to configure, you can then copy the content of the `.settings` folder to an other *module project*.

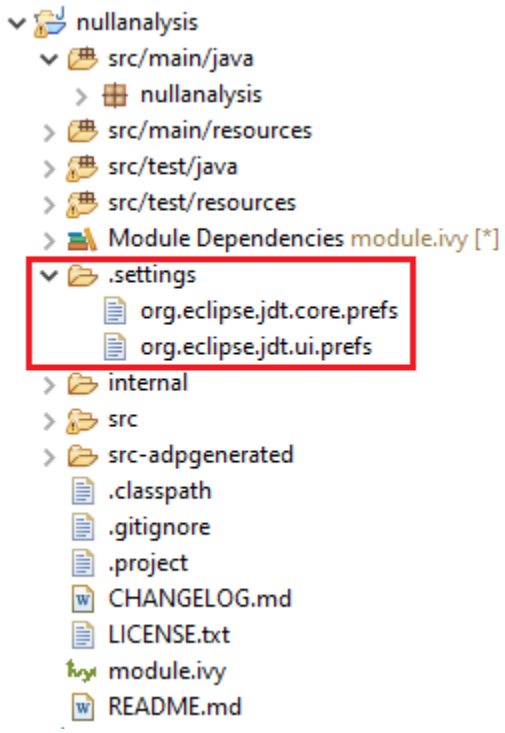


Fig. 96: Null Analysis Settings Folder

Warning: You may lose information if your target module project already has custom parameterization or if it was created with another MicroEJ SDK version. In case of any doubt, please configure the options manually or merge with a text file comparator.

MicroEJ Libraries

Many libraries available on *Central Repository* are annotated with Null Analysis. If you are using a library which is not yet annotated, please contact *our support team*.

For the benefit of Null Analysis, some APIs have been slightly constrained compared to the Javadoc description. Here are some examples to illustrate the philosophy:

- System.getProperty(String key, String def) does not accept a `null` default value, which allows to ensure the returned value is always non `null`.
- Collections of the Java Collections Framework that can hold `null` elements (e.g. HashMap) do not accept `null` elements. This allows APIs to return `null` (e.g. HashMap.get(Object)) only when an element is not contained in the collection.

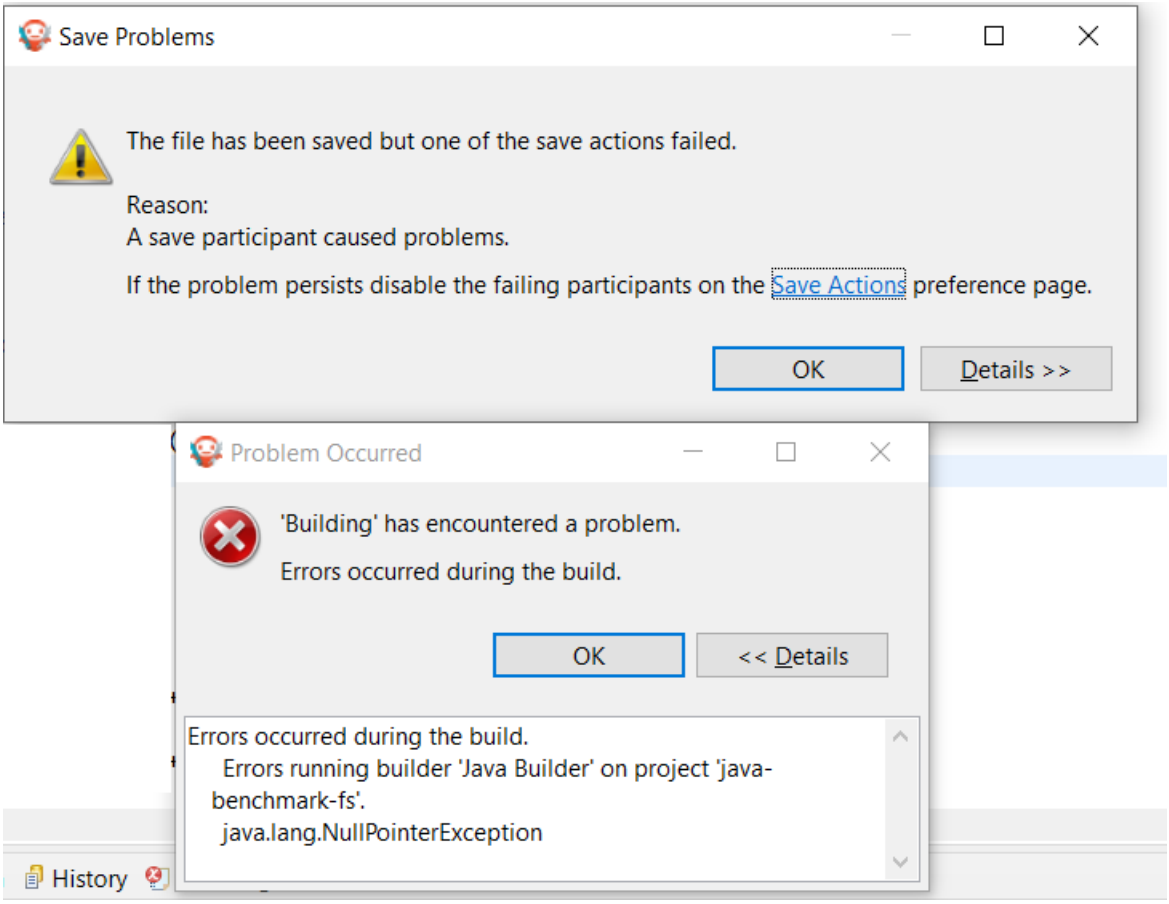
Implementations are left unchanged and still comply with the Javadoc description whether the Null Analysis is enabled or not. So if these additional constraints are not acceptable for your project, please disable Null Analysis.

Advanced Use

For more information about Null Analysis and inter-procedural analysis, please visit Eclipse JDT Null Analysis documentation.

Troubleshooting

The project cannot build anymore after Null Analysis setup




```
java.lang.NullPointerException
  at org.eclipse.jdt.internal.compiler.lookup.BinaryTypeBinding.getMethods(BinaryTypeBinding.
↪java:1348)
  at org.eclipse.jdt.internal.compiler.lookup.AnnotationBinding.
↪setMethodBindings(AnnotationBinding.java:238)
  at org.eclipse.jdt.internal.compiler.lookup.LookupEnvironment.
↪createAnnotation(LookupEnvironment.java:995)
  at org.eclipse.jdt.internal.compiler.lookup.AnnotationBinding.
↪buildTargetAnnotation(AnnotationBinding.java:191)
  at org.eclipse.jdt.internal.compiler.lookup.AnnotationBinding.
↪addStandardAnnotations(AnnotationBinding.java:79)
  at org.eclipse.jdt.internal.compiler.lookup.BinaryTypeBinding.
```

(continues on next page)

(continued from previous page)

```
↪retrieveAnnotations(BinaryTypeBinding.java:1698)
  at org.eclipse.jdt.internal.compiler.lookup.ReferenceBinding.
↪getAnnotations(ReferenceBinding.java:1054)
  at org.eclipse.jdt.internal.compiler.lookup.BinaryTypeBinding.
↪evaluateTypeQualifierDefault(BinaryTypeBinding.java:2021)
  at org.eclipse.jdt.internal.compiler.lookup.BinaryTypeBinding.
↪getNonNullByDefaultValue(BinaryTypeBinding.java:1999)
  at org.eclipse.jdt.internal.compiler.lookup.BinaryTypeBinding.
↪scanTypeForNullDefaultAnnotation(BinaryTypeBinding.java:1943)
  at org.eclipse.jdt.internal.compiler.lookup.BinaryTypeBinding.
↪cachePartsFrom(BinaryTypeBinding.java:470)
  at org.eclipse.jdt.internal.compiler.lookup.LookupEnvironment.
↪createBinaryTypeFrom(LookupEnvironment.java:1055)
  at org.eclipse.jdt.internal.compiler.lookup.LookupEnvironment.
↪createBinaryTypeFrom(LookupEnvironment.java:1036)
  at org.eclipse.jdt.internal.compiler.Compiler.accept(Compiler.java:308)
  at org.eclipse.jdt.internal.compiler.lookup.LookupEnvironment.askForType(LookupEnvironment.
↪java:326)
  at org.eclipse.jdt.internal.compiler.lookup.PackageBinding.getType(PackageBinding.java:195)
  at org.eclipse.jdt.internal.compiler.lookup.PackageBinding.
↪initDefaultNullness(PackageBinding.java:325)
  at org.eclipse.jdt.internal.compiler.lookup.PackageBinding.
↪getDefaultNullness(PackageBinding.java:339)
  at org.eclipse.jdt.internal.compiler.lookup.BinaryTypeBinding.
↪scanTypeForNullDefaultAnnotation(BinaryTypeBinding.java:1965)
  at org.eclipse.jdt.internal.compiler.lookup.BinaryTypeBinding.
↪cachePartsFrom(BinaryTypeBinding.java:470)
  at org.eclipse.jdt.internal.compiler.lookup.LookupEnvironment.
↪createBinaryTypeFrom(LookupEnvironment.java:1055)
  at org.eclipse.jdt.internal.compiler.lookup.LookupEnvironment.
↪createBinaryTypeFrom(LookupEnvironment.java:1036)
  at org.eclipse.jdt.internal.compiler.Compiler.accept(Compiler.java:308)
  at org.eclipse.jdt.internal.compiler.lookup.LookupEnvironment.askForType(LookupEnvironment.
↪java:326)
  at org.eclipse.jdt.internal.compiler.lookup.LookupEnvironment.getType(LookupEnvironment.
↪java:1705)
  at org.eclipse.jdt.internal.compiler.lookup.LookupEnvironment.
↪getResolvedType(LookupEnvironment.java:1633)
  at org.eclipse.jdt.internal.compiler.lookup.LookupEnvironment.
↪getResolvedJavaBaseType(LookupEnvironment.java:1645)
  at org.eclipse.jdt.internal.compiler.lookup.AnnotationBinding.
↪buildTargetAnnotation(AnnotationBinding.java:134)
  at org.eclipse.jdt.internal.compiler.lookup.AnnotationBinding.
↪addStandardAnnotations(AnnotationBinding.java:79)
  at org.eclipse.jdt.internal.compiler.lookup.BinaryTypeBinding.
↪retrieveAnnotations(BinaryTypeBinding.java:1698)
  at org.eclipse.jdt.internal.compiler.lookup.ReferenceBinding.
↪getAnnotations(ReferenceBinding.java:1054)
  at org.eclipse.jdt.internal.compiler.lookup.BinaryTypeBinding.
↪evaluateTypeQualifierDefault(BinaryTypeBinding.java:2021)
  at org.eclipse.jdt.internal.compiler.lookup.BinaryTypeBinding.
↪getNonNullByDefaultValue(BinaryTypeBinding.java:1999)
```

(continues on next page)

(continued from previous page)

```
  at org.eclipse.jdt.internal.compiler.lookup.BinaryTypeBinding.
↪scanTypeForNullDefaultAnnotation(BinaryTypeBinding.java:1943)
  at org.eclipse.jdt.internal.compiler.lookup.BinaryTypeBinding.
↪cachePartsFrom(BinaryTypeBinding.java:470)
  at org.eclipse.jdt.internal.compiler.lookup.LookupEnvironment.
↪createBinaryTypeFrom(LookupEnvironment.java:1055)
  at org.eclipse.jdt.internal.compiler.lookup.LookupEnvironment.
↪createBinaryTypeFrom(LookupEnvironment.java:1036)
  at org.eclipse.jdt.internal.compiler.Compiler.accept(Compiler.java:308)
  at org.eclipse.jdt.internal.compiler.lookup.LookupEnvironment.askForType(LookupEnvironment.
↪java:326)
  at org.eclipse.jdt.internal.compiler.lookup.PackageBinding.getType(PackageBinding.java:195)
  at org.eclipse.jdt.internal.compiler.lookup.PackageBinding.
↪isViewedAsDeprecated(PackageBinding.java:314)
  at org.eclipse.jdt.internal.compiler.lookup.ReferenceBinding.
↪isViewedAsDeprecated(ReferenceBinding.java:1745)
  at org.eclipse.jdt.internal.compiler.lookup.BinaryTypeBinding.
↪cachePartsFrom(BinaryTypeBinding.java:566)
  at org.eclipse.jdt.internal.compiler.lookup.LookupEnvironment.
↪createBinaryTypeFrom(LookupEnvironment.java:1055)
  at org.eclipse.jdt.internal.compiler.lookup.LookupEnvironment.
↪createBinaryTypeFrom(LookupEnvironment.java:1036)
  at org.eclipse.jdt.internal.compiler.Compiler.accept(Compiler.java:308)
  at org.eclipse.jdt.internal.compiler.lookup.LookupEnvironment.askForType(LookupEnvironment.
↪java:257)
  at org.eclipse.jdt.internal.compiler.lookup.LookupEnvironment.getType(LookupEnvironment.
↪java:1703)
  at org.eclipse.jdt.internal.compiler.lookup.BinaryTypeBinding.
↪getNonNullByDefaultValue(BinaryTypeBinding.java:1995)
  at org.eclipse.jdt.internal.compiler.lookup.BinaryTypeBinding.
↪scanTypeForNullDefaultAnnotation(BinaryTypeBinding.java:1943)
  at org.eclipse.jdt.internal.compiler.lookup.BinaryTypeBinding.
↪cachePartsFrom(BinaryTypeBinding.java:470)
  at org.eclipse.jdt.internal.compiler.lookup.LookupEnvironment.
↪createBinaryTypeFrom(LookupEnvironment.java:1055)
  at org.eclipse.jdt.internal.compiler.lookup.LookupEnvironment.
↪createBinaryTypeFrom(LookupEnvironment.java:1036)
  at org.eclipse.jdt.internal.compiler.Compiler.accept(Compiler.java:308)
  at org.eclipse.jdt.internal.compiler.lookup.LookupEnvironment.askForType(LookupEnvironment.
↪java:326)
  at org.eclipse.jdt.internal.compiler.lookup.PackageBinding.getType(PackageBinding.java:195)
  at org.eclipse.jdt.internal.compiler.lookup.PackageBinding.
↪initDefaultNullness(PackageBinding.java:325)
  at org.eclipse.jdt.internal.compiler.lookup.PackageBinding.
↪getDefaultNullness(PackageBinding.java:339)
  at org.eclipse.jdt.internal.compiler.lookup.BinaryTypeBinding.
↪scanTypeForNullDefaultAnnotation(BinaryTypeBinding.java:1965)
  at org.eclipse.jdt.internal.compiler.lookup.BinaryTypeBinding.
↪cachePartsFrom(BinaryTypeBinding.java:470)
  at org.eclipse.jdt.internal.compiler.lookup.LookupEnvironment.
↪createBinaryTypeFrom(LookupEnvironment.java:1055)
  at org.eclipse.jdt.internal.compiler.lookup.LookupEnvironment.
```

(continues on next page)

(continued from previous page)

```
↪createBinaryTypeFrom(LookupEnvironment.java:1036)
  at org.eclipse.jdt.internal.compiler.Compiler.accept(Compiler.java:308)
  at org.eclipse.jdt.internal.compiler.lookup.LookupEnvironment.askForType(LookupEnvironment.
↪java:326)
  at org.eclipse.jdt.internal.compiler.lookup.LookupEnvironment.getType(LookupEnvironment.
↪java:1705)
  at org.eclipse.jdt.internal.compiler.lookup.LookupEnvironment.
↪getResolvedType(LookupEnvironment.java:1633)
  at org.eclipse.jdt.internal.compiler.lookup.LookupEnvironment.
↪getResolvedJavaBaseType(LookupEnvironment.java:1645)
  at org.eclipse.jdt.internal.compiler.lookup.Scope.getJavaLangObject(Scope.java:2961)
  at org.eclipse.jdt.internal.compiler.lookup.ClassScope.connectSuperclass(ClassScope.
↪java:1065)
  at org.eclipse.jdt.internal.compiler.lookup.ClassScope.connectTypeHierarchy(ClassScope.
↪java:1246)
  at org.eclipse.jdt.internal.compiler.lookup.CompilationUnitScope.
↪connectTypeHierarchy(CompilationUnitScope.java:367)
  at org.eclipse.jdt.internal.compiler.lookup.LookupEnvironment.
↪completeTypeBindings(LookupEnvironment.java:518)
  at org.eclipse.jdt.internal.compiler.Compiler.internalBeginToCompile(Compiler.java:878)
  at org.eclipse.jdt.internal.compiler.Compiler.beginToCompile(Compiler.java:394)
  at org.eclipse.jdt.internal.compiler.Compiler.compile(Compiler.java:444)
  at org.eclipse.jdt.internal.compiler.Compiler.compile(Compiler.java:426)
  at org.eclipse.jdt.internal.core.builder.AbstractImageBuilder.compile(AbstractImageBuilder.
↪java:386)
  at org.eclipse.jdt.internal.core.builder.BatchImageBuilder.compile(BatchImageBuilder.
↪java:214)
  at org.eclipse.jdt.internal.core.builder.AbstractImageBuilder.compile(AbstractImageBuilder.
↪java:318)
  at org.eclipse.jdt.internal.core.builder.BatchImageBuilder.build(BatchImageBuilder.java:79)
  at org.eclipse.jdt.internal.core.builder.JavaBuilder.buildAll(JavaBuilder.java:275)
  at org.eclipse.jdt.internal.core.builder.JavaBuilder.build(JavaBuilder.java:192)
  at org.eclipse.core.internal.events.BuildManager$2.run(BuildManager.java:832)
  at org.eclipse.core.runtime.SafeRunner.run(SafeRunner.java:45)
  at org.eclipse.core.internal.events.BuildManager.basicBuild(BuildManager.java:220)
  at org.eclipse.core.internal.events.BuildManager.basicBuild(BuildManager.java:263)
  at org.eclipse.core.internal.events.BuildManager$1.run(BuildManager.java:316)
  at org.eclipse.core.runtime.SafeRunner.run(SafeRunner.java:45)
  at org.eclipse.core.internal.events.BuildManager.basicBuild(BuildManager.java:319)
  at org.eclipse.core.internal.events.BuildManager.basicBuildLoop(BuildManager.java:371)
  at org.eclipse.core.internal.events.BuildManager.build(BuildManager.java:392)
  at org.eclipse.core.internal.events.AutoBuildJob.doBuild(AutoBuildJob.java:154)
  at org.eclipse.core.internal.events.AutoBuildJob.run(AutoBuildJob.java:244)
  at org.eclipse.core.internal.jobs.Worker.run(Worker.java:63)
```

You may encounter the two popup windows and the full stack trace above when your version of `EDC` is too old. To fix this issue, please use EDC-1.3.3 or higher with MicroEJ SDK `5.3.0` or higher.

3.10 IDE

The SDK provides an Integrated Development Environment (IDE) for creating and building Applications. It is based on Eclipse Java Edition and relies on the integrated Java Compiler (JDT).

3.10.1 Startup

When starting the SDK, it prompts you to select the last used workspace or a default workspace on the first run.

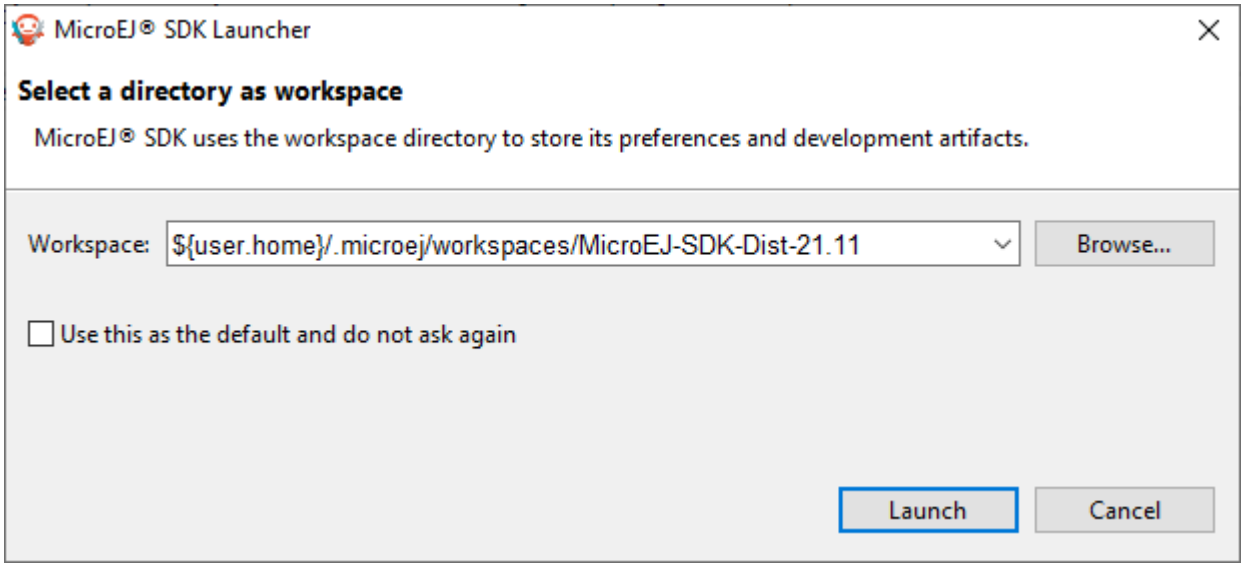


Fig. 97: Workspace selection

A workspace is the Eclipse main folder where are imported a set of projects containing the source code.

When loading a new workspace, the SDK prompts for the location of the MicroEJ repository, where Architectures, Platforms or Virtual Devices will be imported.

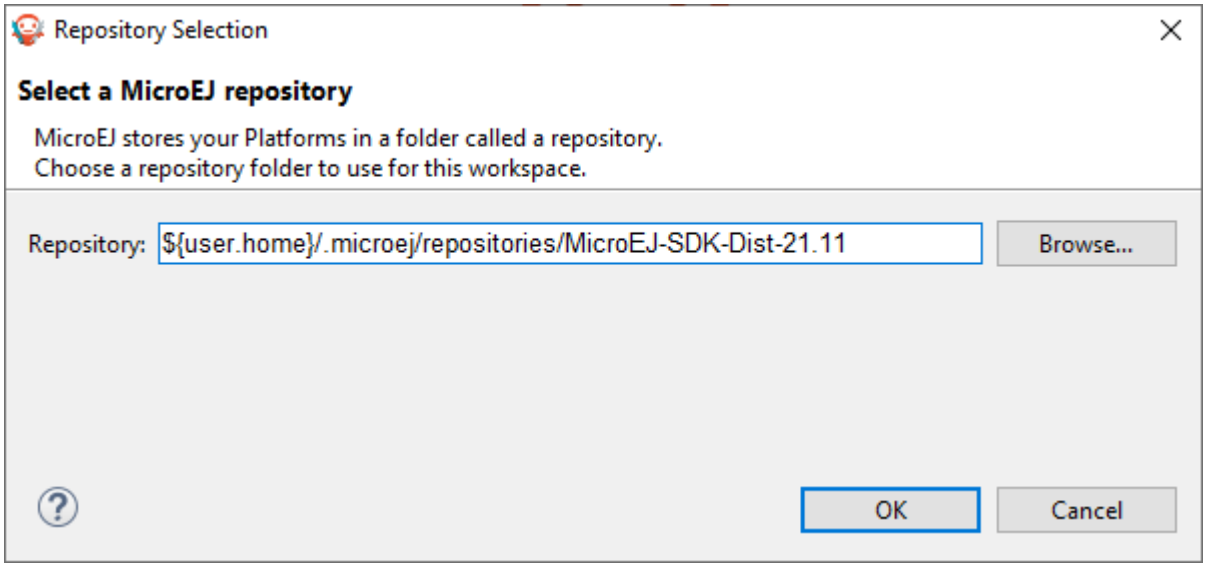


Fig. 98: Repository selection

By default, the SDK suggests to point to the default repository on your operating system, located at `${user.home}/.microej/repositories/[version]`. You can select an alternative 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.

3.10.2 Resolve Dependencies in Workspace

When resolving the modules' dependencies, if the project of a dependency is imported and opened in the same workspace as the module, the project is directly used for compilation and execution instead of using the dependency, provided that the dependency's project has the same version as the one required by the module.

For example, suppose that the workspace contains a module `myApp` and its dependency `mylib` :



Fig. 99: A module and its dependency opened in the same workspace

If the `mylib` project's version is `1.0.0`, it is used for compilation and execution. Otherwise the published artifact is downloaded from the artifact repository.

To avoid a dependency to be resolved in the workspace, you can close the corresponding project or remove it from the workspace.

Warning: If you open, close, import or remove a project, you must refresh the dependency resolution of other previously imported projects by clicking on the Resolve All button :

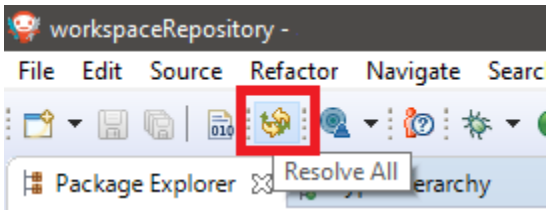


Fig. 100: Resolve all the workspace projects

3.10.3 Resolve Foundation Libraries in Workspace

A *Foundation Library* is composed of :

- An API project that contains Java classes, methods and fields used at compile time with their associated Javadoc,
- An Implementation project that contains the runtime code executed by the Platform and Low Level C header files

Beside Foundation Library projects, there is usually a *Mock* project that contains the implementation of native methods for simulation.

Note: To learn how to setup a Foundation Library, please consult the How-to available on https://github.com/MicroEJ/How-To/tree/master/FoundationLibrary-Get-Started.

When the API is set as a dependency, the Implementation project is automatically used at runtime if it is opened in the workspace.

If a Mock project or a Front Panel project is also opened in the workspace, it is automatically used for execution on Simulator.

Note: When opened in the workspace, Foundation Library Implementation projects, Mock projects and Front Panel projects are loaded, regardless of their version, prior to the ones provided by the Platform (if any).

To avoid the use of an Implementation project, a Mock project or a Front Panel project, uncheck the `Resolve Foundation Library in workspace` option in `Window` > `Preferences` > `MicroEJ` > `Settings`.

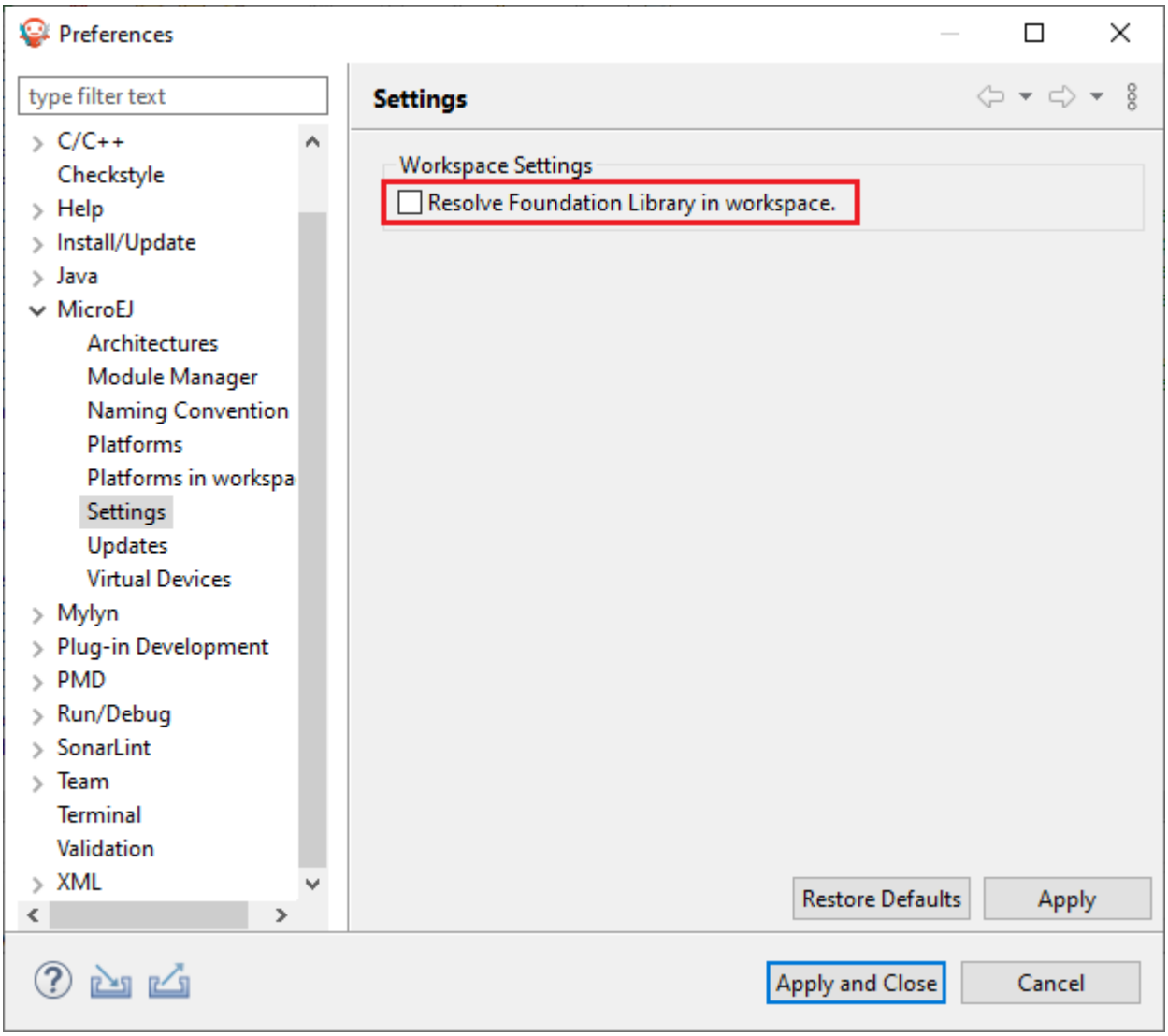


Fig. 101: Resolve Foundation Library in workspace

3.10.4 Resolve Front Panel in Workspace

A Front Panel is a “mock” of the control panel of the device. The Front Panel generates a graphical representation of the device, and is displayed in a window on the user’s development machine when the application is executed in the Simulator.

Note: To learn more about Front Panels, consult the *Front Panel section*.

When a Front Panel project is opened in the workspace, it is automatically used at runtime when launching the Simulator.

Note: This feature requires SDK version `5.7.0` or higher and Architecture version `8.0` or higher.

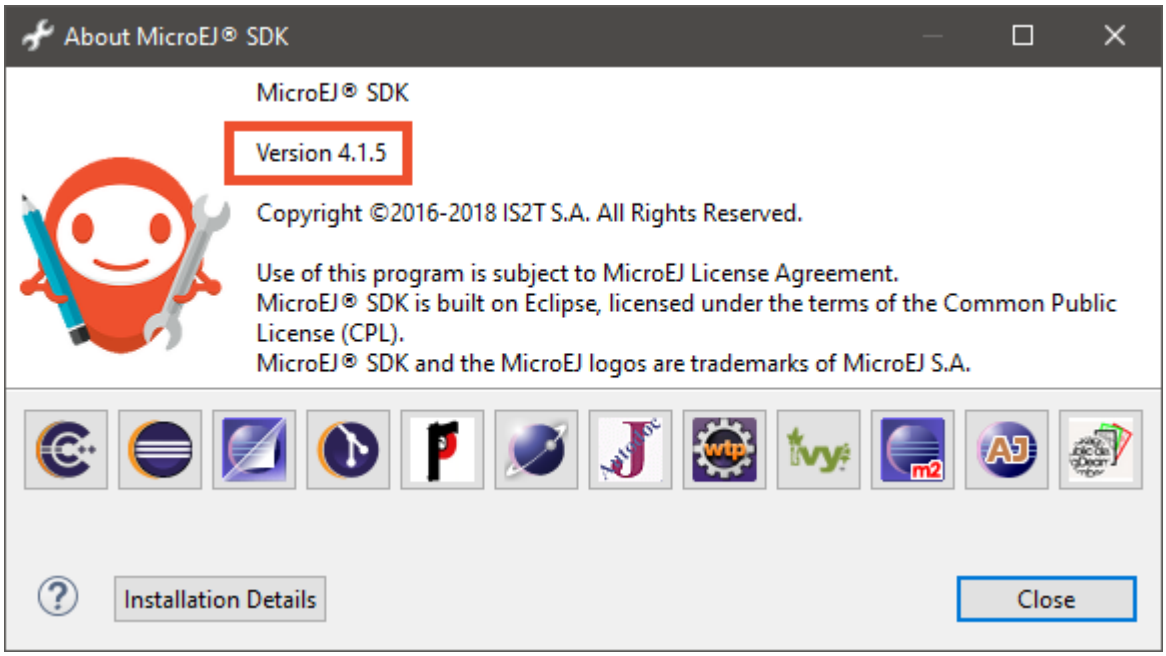
If the workspace contains several Front Panel projects, they are all automatically used by the Simulator, which can very probably causes issues. You can select the Front Panel you want to use by closing all the other Front Panel projects.

Also, a Front Panel project can contain several Front Panel descriptor files. Refer to the *Multiple Front Panel Files* section to know how to select the file you want to use.

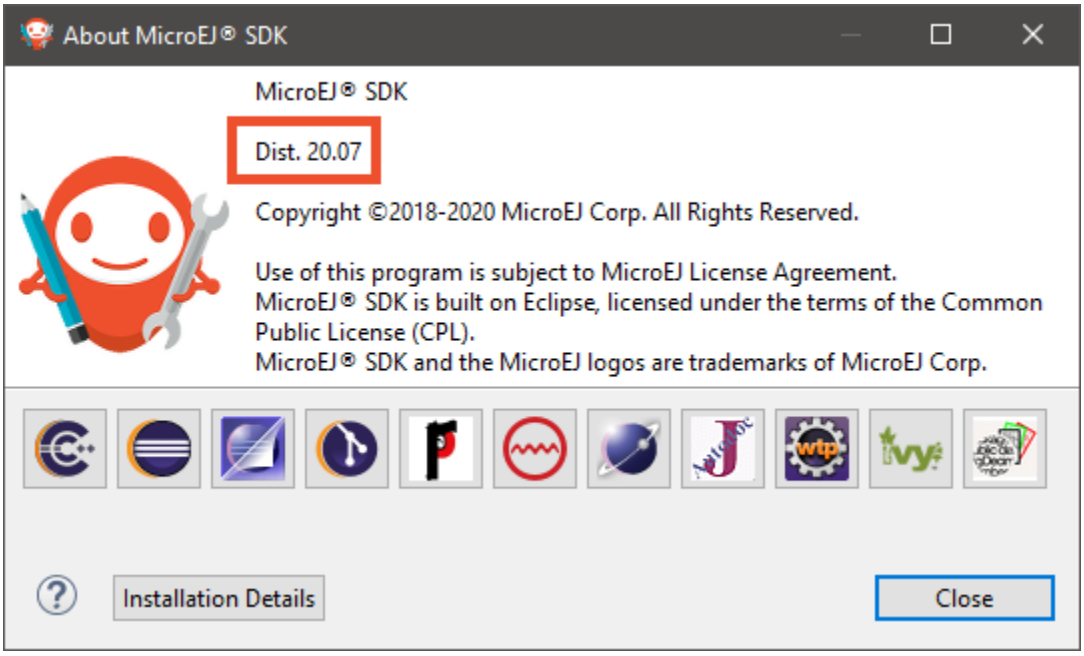
3.11 SDK Version

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

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

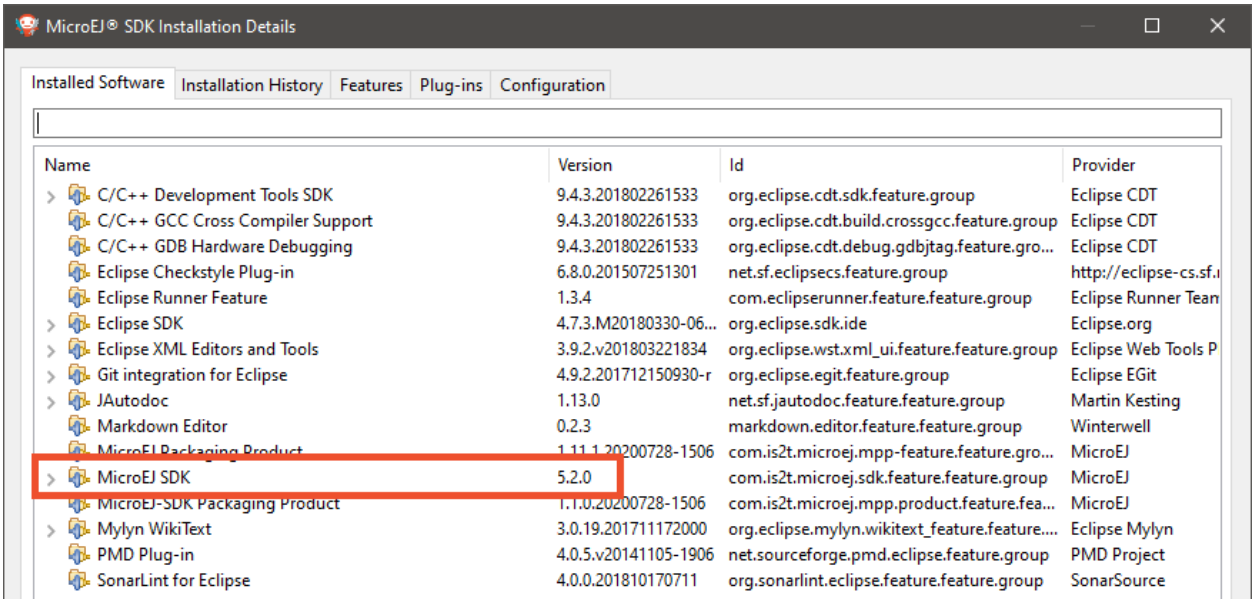


In case of SDK `5.x`, the value displayed is the SDK distribution, such as `19.05` or `20.07`:



To retrieve the SDK version that is currently installed in this distribution, proceed with the following steps:

- Click on the `Installation Details` button,
- Click on the `Installed Software` tab,
- Retrieve the version of entry named `MicroEJ SDK`.



3.12 MicroEJ Module Manager

3.12.1 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.

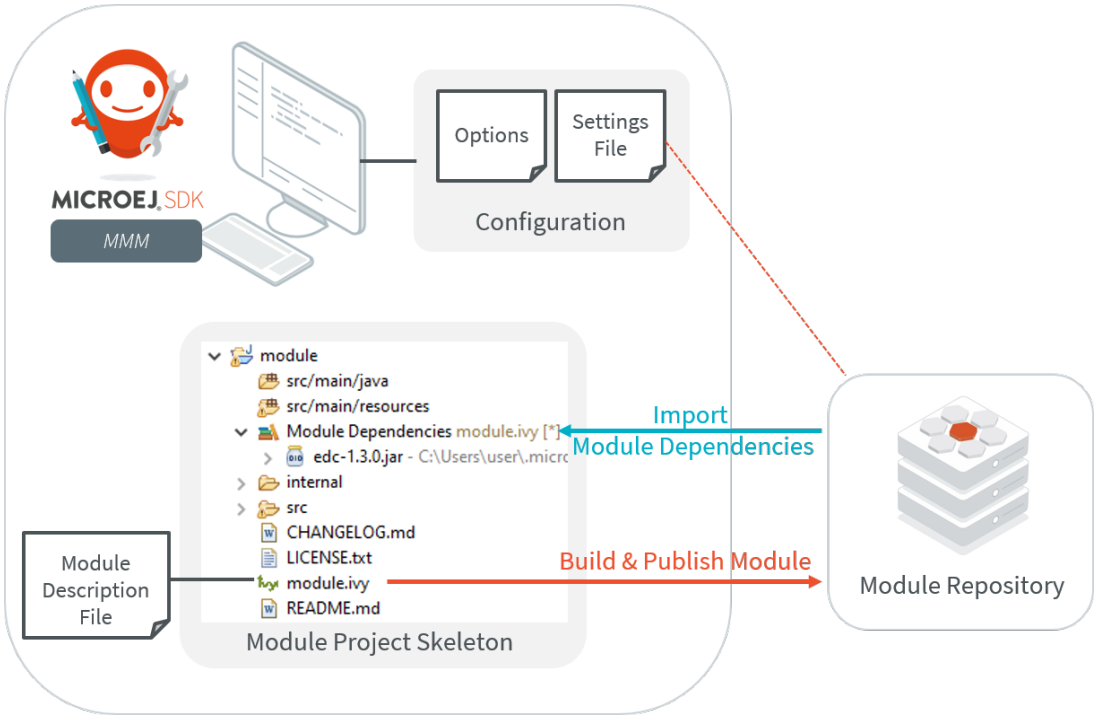


Fig. 102: MMM Overview

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.

3.12.2 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-E.pdf.

3.12.3 Module Project Skeleton

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

- Select File > New > Project... ,
- Select MicroEJ > Module Project [1],
- Fill the module information (project name, module organization, name and revision),
- Select one of the suggested skeletons depending on the desired *module nature*,
- Click on Finish .

The project is created and a set of files and directories are generated from the selected skeleton.

Note: When an empty Eclipse project already exists or when the skeleton has to be created within an existing directory, the MicroEJ module is created as follows:

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

3.12.4 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"/>
```

(continues on next page)

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

(continued from previous page)

```
        <conf name="dist" visibility="public"/>
        <conf name="test" visibility="private"/>
    </configurations>

    <publications>
    </publications>

    <dependencies>
      <dependency org="[dep_organisation]" name="[dep_name]" rev="[dep_version]"/>
    </dependencies>
</ivy-module>
```

Enable MMM Semantic

The MMM semantic is enabled in a module by adding the MicroEJ XML namespace and the `ej:version` attribute in the `ivy-module` node:

```
<ivy-module xmlns:ej="https://developer.microej.com" ej:version="2.0.0">
```

Note: Multiple namespaces can be declared in the `ivy-module` node.

MMM semantic is enabled in the module created with the *Module Project Skeleton*.

Module Dependencies

Module dependencies are added to the `dependencies` node as follow:

```
<dependencies>
  <dependency org="[dep_organisation]" name="[dep_name]" rev="[dep_version]"/>
</dependencies>
```

When no matching rule is specified, the default matching rule is `compatible` .

Dependency Matching Rule

The following matching rules are specified by MMM:

Name	Range Notation	Semantic
compatible	[M.m.p-RC, (M+1).0.0-RC[	Equal or up to next major version. Default if not set.
equivalent	[M.m.p-RC, M.(m+1).0-RC [	Equal or up to next minor version
greaterOrEqual	[M.m.p-RC, ∞[	Equal or greater versions
perfect	[M.m.p-RC, M.m.(p+1)-RC[	Exact match (strong dependency)

Set the matching rule of a given dependency with `ej:match="matching rule"` . For example:

```
<dependency org="[dep_organisation]" name="[dep_name]" rev="[dep_version]" ej:match="perfect
↪" />
```

Dependency Visibility

- A dependency declared `public` is transitively resolved by upper modules. The default when not set.
- A dependency declared `private` is only used by the module itself, typically for:
  - Bundling the content into the module
  - Testing the module

The visibility is set by the configurations declared in the `configurations` node. For example:

```
<configurations defaultconfmapping="default->default;provided->provided">
    <conf name="[conf_name]" visibility="private"/>
</configurations>
```

The configuration of a dependency is specified by setting the `conf` attribute, for example:

```
<dependency org="[dep_organisation]" name="[dep_name]" rev="[dep_version]" conf="[conf_name]-
↪>*" />
```

Build Options

MMM builds can be configured by settings options in the `module.ivy` file using the `ea:property` tag inside the `ea:build` tag:

```
<ea:build organisation="..." module="..." revision="x.y.z">
    <ea:property name="[build_option_name]" value="[build_option_value]"/>
</ea:build>
```

Refer to the documentation of *Module Natures* for the list of available build options for each Module Nature.

The options can also be defined via System Properties. If an option is defined as both System Property and `ea:property` tag, the value passed as System Property takes precedence.

Automatic Update Before Resolution

The Easyant plugin `ivy-update` can be used to automatically update the version (attribute `rev`) of every module dependencies declared.

```
<info organisation="[organisation]" module="[name]" status="integration" revision="[version]
↪">
    <ea:plugin org="com.is2t.easyant.plugins" name="ivy-update" revision="1.+" />
</info>
```

When the plugin is enabled, for each *module dependency*, MMM will check the version declared in the module file and update it to the highest version available which satisfies the matching rule of the dependency.

3.12.5 SDK Configuration

By default, when starting an empty workspace, the SDK is configured to import dependencies from *MicroEJ Central Repository* and to publish built modules to a local directory. The repository configuration is stored in a *settings file* (`ivysettings.xml`), and the default one is located at `$USER_HOME\.microej\microej-ivysettings-[VERSION].xml`

Preferences Page

The MMM preferences page in the SDK is available at Window > Preferences > MicroEJ > Module Manager Page 148, 1.

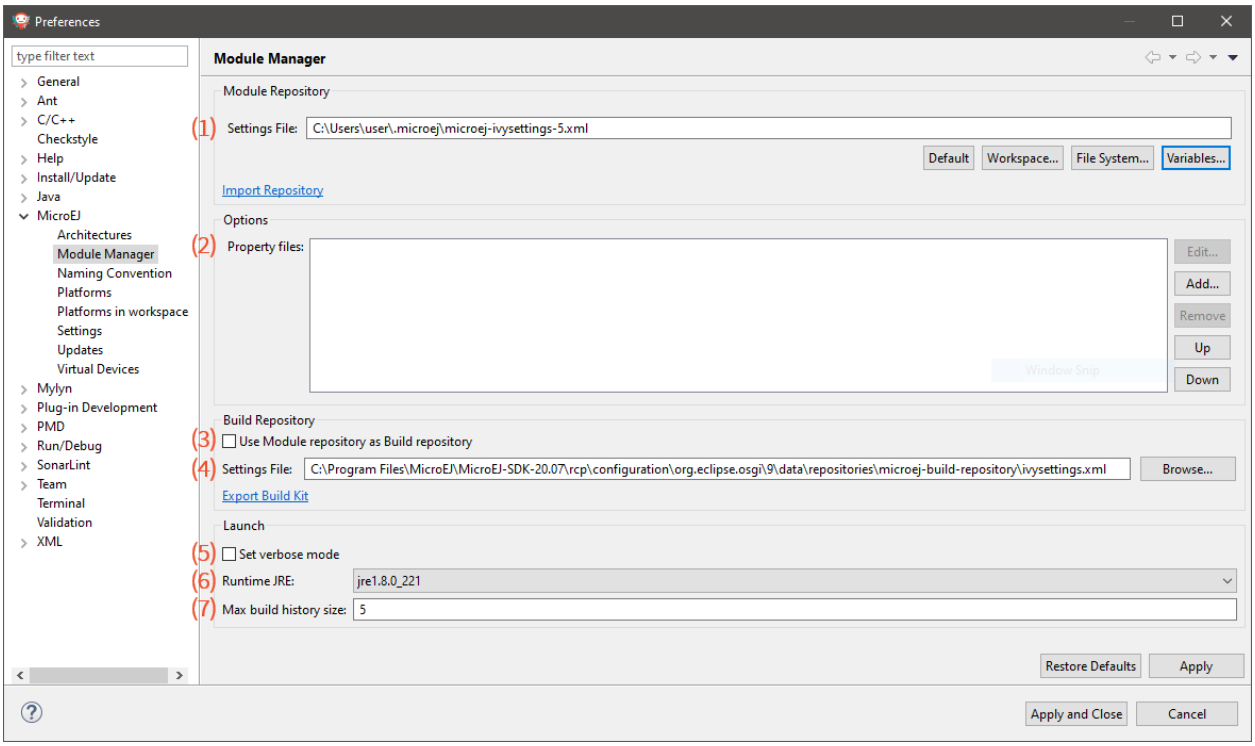
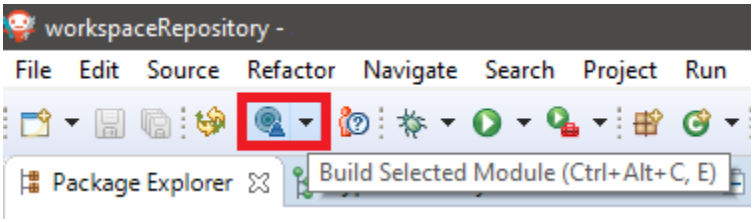


Fig. 103: MMM Preferences Page

This page allows to configure the following elements:

1. `Settings File` : the file describing how to connect *module repositories*. See the *settings file* section.
2. `Options` : files declaring MMM options. See the *Options* section.
3. `Use Module repository as Build repository` : the *settings file* for connecting the build repository in place of the one bundled in the SDK. This option shall not be enabled by default and is reserved for advanced configuration.
4. `Build repository Settings File` : the *settings file* for connecting the build repository in place of the one bundled in the SDK. This option is automatically initialized the first time the SDK is launched. It shall not be modified by default and is reserved for advanced configuration.
5. `Set verbose mode` : to enable advanced debug traces when building a module.

6. `Runtime JRE` : the Java Runtime Environment that executes the build process.
7. `Max build history size` : the maximum number of previous builds available in Build Module shortcut list:



Settings File

The settings file is an XML file that describes how MMM connects local or online *module repositories*. The file format is described in Apache Ivy documentation.

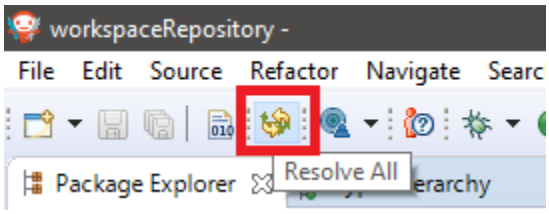
To configure MMM to a custom settings file (usually from an *offline repository*):

1. Set Settings file to a custom `ivysettings.xml` settings filePage 148, 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:



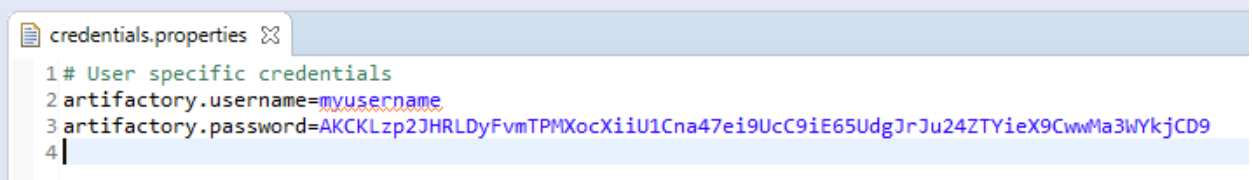
   To only resolve a subset of the workspace projects:
   - In the Package Explorer, select the desired projects,
   - Right-click on a project and select Ivy > Clean all caches .
3. Trigger Add-On Library processors for automatically generated source code
   - Select Project > Clean… ,
   - Select Clean all projects ,
   - Click on Clean button.

Options

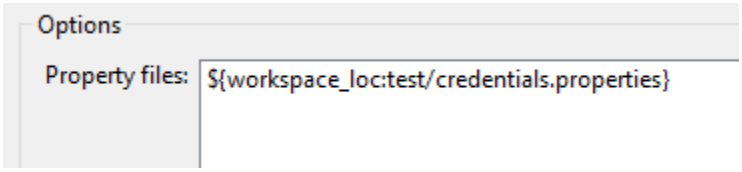
Options can be used to parameterize a *module description file* or a *settings file*. Options are declared as key/value pairs in a standard Java properties file, and are expanded using the `${my_property}` notation.

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



2. Register this property file to MMM options

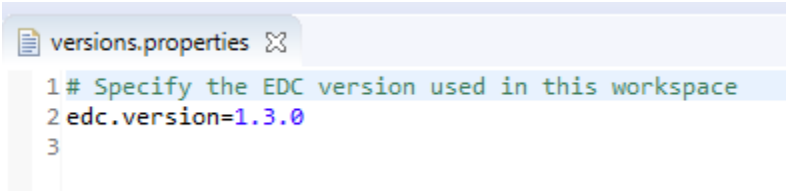


3. Use this option in a *settings file*

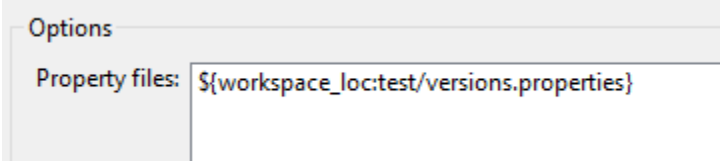
```
38
39  <credentials host="artifactory.corp" realm="Artifactory Realm" username="${artifactory.username}" passwd="${artifactory.password}" />
40
```

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



2. Register this property file to MMM options



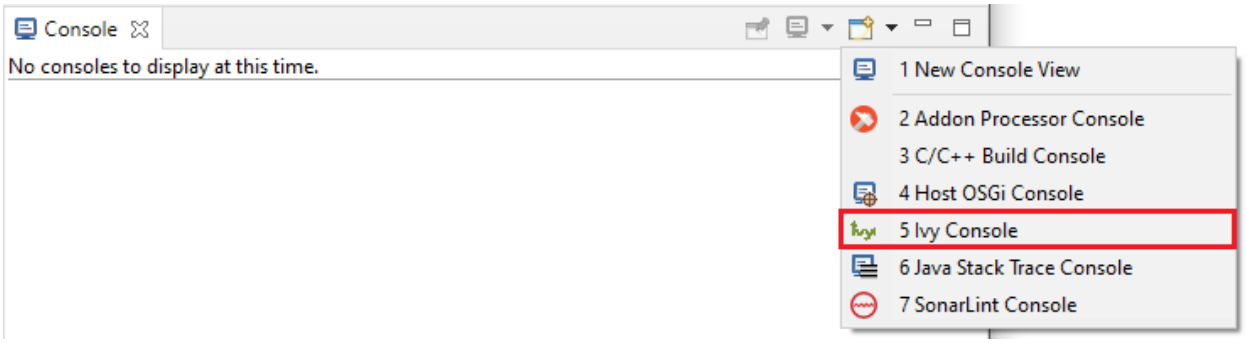
3. Use this option in a *module description file*

```
22      <dependencies>
23          <!--
24              Use the EDC version defined by MMM configuration
25          -->
26          <dependency org="ej.api" name="edc" rev="${edc.version}" />
27      </dependencies>
28  </ivy-module>
```

Resolution Logs

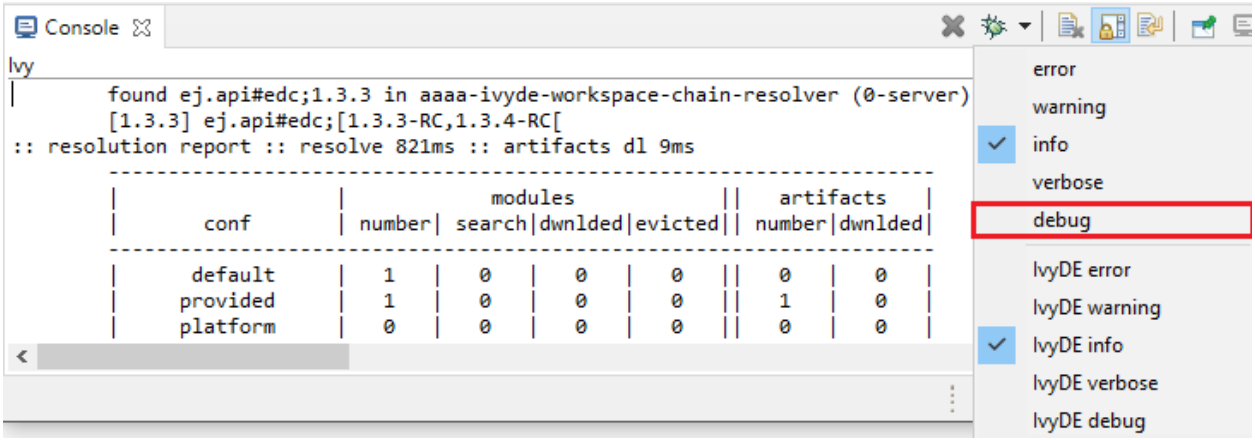
Resolution logs of module projects imported in the workspace are available from the console view:

- Select `Windows` > `Show View` > `Console` ,
- In the Console view, click on the console window icon and select `Ivy console` :



To enable the verbose mode:

- In the `Ivy console` view, click on the debug icon and select *debug* instead of *info* (defaults):



This triggers the full workspace resolution with verbose mode enabled.

3.12.6 Module Build

In the SDK, the build of a MicroEJ module project can be started as follows:

- In the *Package Explorer*, right-click on the project,
- Select Build Module .

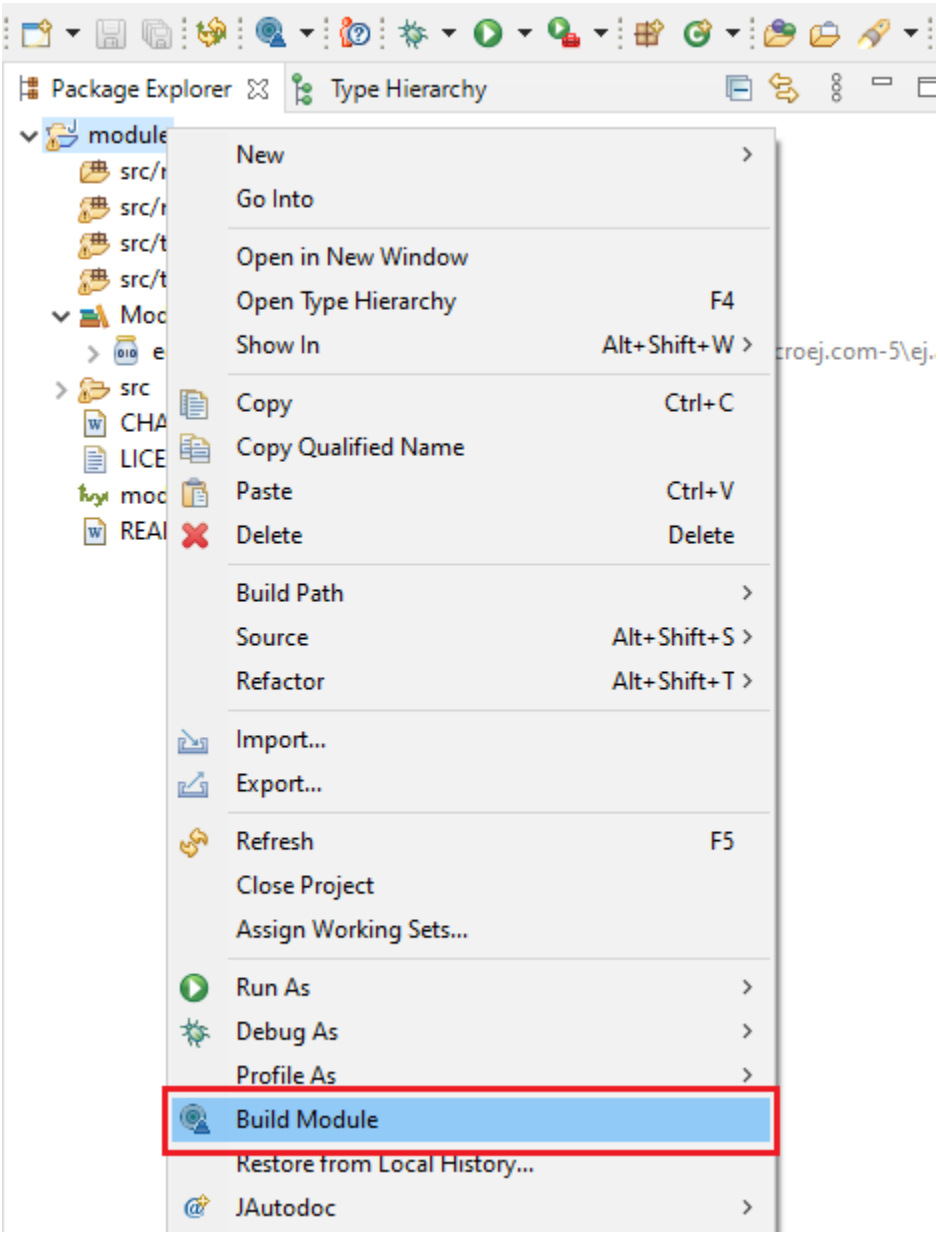


Fig. 104: Module Build

The build of a module can take time depending on

- the *module nature* to build,

- the number and the size of module dependencies to download,
- the repository connection bandwidth, ...

The module build logs are redirected to the integrated console.

Alternatively, the build of a MicroEJ module project can be started from the build history:

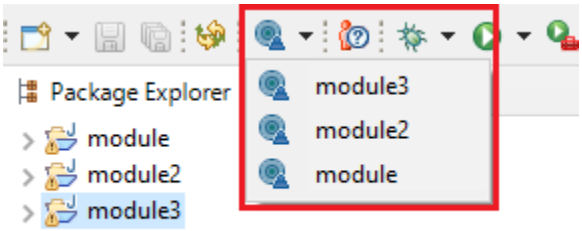


Fig. 105: Module Build History

3.12.7 Build Kit

The Module Manager Build Kit is a consistent set of tools, scripts, configuration and artifacts required for building modules in command-line mode. Starting from SDK `5.4.0`, it also contains a *Command Line Interface* (CLI). The Build Kit allows to work in headless mode (e.g. in a terminal) and to build your modules using a Continuous Integration tool.

The Build Kit is bundled with the SDK and can be exported using the following steps:[2]

- Select `File` > `Export` > `MicroEJ` > `Module Manager Build Kit`,
- Choose an empty `Target directory`,
- Click on the `Finish` button.

Once the Build Kit is fully exported, the directory content shall look like:

```
/
├─ bin
│  ├─ mmm
│  └─ mmm.bat
├─ conf
├─ lib
├─ microej-build-repository
│  ├─ ant-contrib
│  ├─ com
│  ├─ ...
│  └─ ivysettings.xml
├─ microej-module-repository
│  └─ ivysettings.xml
└─ release.properties
```

[2] If using SDK versions lower than `5.4.0`, please refer to the *following section*.

- Add the `bin` directory of the Build Kit directory to the `PATH` environment variable of your machine.
- Make sure the `JAVA_HOME` environment variable is set and points to a JRE/JDK installation or that `java` executable is in the `PATH` environment variable (Java 8 is required)
- Confirm that the installation works fine by executing the command `mmm --version`. The result should display the MMM CLI version.

The `mmm` tool can run on any supported *Operating Systems*:

- on Windows, either in the command prompt using the Windows batch script `mmm.bat` or in MinGW environments such as Git BASH using the bash script `mmm`.
- on macOS and Linux distributions using the bash script `mmm`.

The build repository (`microej-build-repository` directory) contains scripts and tools for building modules. It is specific to a SDK version and shall not be modified by default.

The module repository (`microej-module-repository` directory) contains a default *Settings File* for importing modules from *Central Repository* and this local repository (modules that are locally built will be published to this directory). You can override with custom settings or by extracting an *offline repository*.

To go further with headless builds, please consult *the next chapter* for command line builds, and this *tutorial* to setup MicroEJ modules build in continuous integration environments.

3.12.8 Command Line Interface

Starting from version `5.4.0`, the SDK provides a Command Line Interface (CLI). Please refer to the *Build Kit* section for installation details.

The following operations are supported by the MMM CLI:

- creating a module project
- cleaning a module project
- building a module project
- running a MicroEJ Application project on the Simulator
- publishing a module in a module repository

Usage

In order to use the MMM CLI for your project:

- go to the root directory of your project
- run the following command

```
mmm [COMMAND] [OPTION]...
```

where `COMMAND` is the command to execute (for example `mmm build`). The available commands are:

- `help`: display help information about the specified command
- `init`: create a new project
- `clean`: clean the project
- `build`: build the project
- `publish`: build the project and publish the module

- `run` : run the MicroEJ Application project on the Simulator

The available options are:

- `--help` (`-h`): show the help message and exit
- `--version` (`-V`): print version information and exit
- `--build-repository-settings-file` (`-b`): path of the Ivy settings file for build scripts and tools. Defaults to `${CLI_HOME}/microej-build-repository/ivysettings.xml` .
- `--module-repository-settings-file` (`-r`): path of the Ivy settings file for modules. Defaults to `${CLI_HOME}/microej-module-repository/ivysettings.xml` .
- `--ivy-file` (`-f`): path of the project's Ivy file. Defaults to `./module.ivy` .
- `--verbose` (`-v`): verbose mode. Disabled by default. Add this option to enable verbose mode.
- `-Dxxx=yyy` : any additional option passed as system properties.

When no command is specified, MMM CLI executes Easyant with custom targets using the `--targets` (`-t`) option (defaults to `clean,verify`).

Shared configuration

In order to share configuration across several projects, these parameters can be defined in the file `${user.home}/.microej/.mmmconfig` . This file uses the TOML format. Parameters names are the same than the ones passed as system properties, except the character `_` is used as a separator instead of `-` . The parameters defined in the [options] section are passed as system properties. Here is an example:

```
build_repository_settings_file = "/home/johndoe/ivy-configuration/ivysettings.xml"
module_repository_settings_file = "/home/johndoe/ivy-configuration/ivysettings.xml"
ivy_file = "ivy.xml"

[options]
my.first.property = "value1"
my.second.property = "value2"
```

Warning:

- TOML values must be surrounded with double quotes
- Backslash characters (`\`) must be doubled (for example a Windows path `C:\\Users\\johndoe\\ivysettings.xml`)

Command line options take precedence over those defined in the configuration file. So if the same option is defined in both locations, the value defined in the command line is used.

Commands

init

The command `init` creates a new project (executes Easyant with `skeleton:generate` target). The skeleton and project information must be passed with the following system properties:

- `skeleton.org` : organisation of the skeleton module. Defaults to `com.is2t.easyant.skeletons` .
- `skeleton.module` : name of the skeleton module. Mandatory, defaults to `microej-javalib` .
- `skeleton.rev` : revision of the skeleton module. Mandatory, defaults to `+` (meaning the latest released version).
- `project.org` : organisation of the project module. Mandatory, defaults to `com.mycompany` .
- `project.module` : name of the project module. Mandatory, defaults to `myproject` .
- `project.rev` : revision of the project module. Defaults to `0.1.0` .
- `skeleton.target.dir` : relative path of the project directory (created if it does not exist). Mandatory, defaults to the current directory.

For example

```
mmm init -Dskeleton.org=com.is2t.easyant.skeletons -Dskeleton.module=microej-javalib -
↪Dskeleton.rev=4.2.8 -Dproject.org=com.mycompany -Dproject.module=myproject -Dproject.rev=1.
↪0.0 -Dskeleton.target.dir=myproject
```

If one of these properties is missing, it will be asked in interactive mode:

```
$ mmm init -Dskeleton.org=com.is2t.easyant.skeletons -Dskeleton.module=microej-javalib -
↪Dskeleton.rev=4.2.8 -Dproject.org=com.mycompany -Dproject.module=myproject -Dproject.rev=1.
↪0.0

...

-skeleton:check-generate:
   [input] skipping input as property skeleton.org has already been set.
   [input] skipping input as property skeleton.module has already been set.
   [input] skipping input as property skeleton.rev has already been set.
   [input] The path where the skeleton project will be unzipped [/home/tdelhomenie/microej/
↪working/skeleton]
```

To force the non-interactive mode, the property `skeleton.interactive.mode` must be set to `false` . In non-interactive mode the default values are used for missing non-mandatory properties, and the creation fails if mandatory properties are missing.

```
$ mmm init -Dskeleton.org=com.is2t.easyant.skeletons -Dskeleton.module=microej-javalib -
↪Dskeleton.rev=4.2.8 -Dproject.org=com.mycompany -Dskeleton.target.dir=myproject -Dskeleton.
↪interactive.mode=false

...

* Problem Report:

expected property 'project.module': Module name of YOUR project
```

clean

The command `clean` cleans the project (executes Easyant with `clean` target). For example

```
mmm clean
```

cleans the project.

build

The command `build` builds the project (executes Easyant with `clean,verify` targets). For example

```
mmm build -f ivy.xml -v
```

builds the project with the Ivy file ivy.xml and in verbose mode.

publish

The command `publish` builds the project and publishes the module. This command accepts the publication target as a parameter, amongst these values:

- `local` (default value): executes the `clean,publish-local` Easyant target, which publishes the project with the resolver referenced by the property `local.resolver` in the *Settings File*.
- `shared`: executes the `clean,publish-shared` Easyant target, which publishes the project with the resolver referenced by the property `shared.resolver` in the *Settings File*.
- `release`: executes the `clean,release` Easyant target, which publishes the project with the resolver referenced by the property `release.resolver` the *Settings File*.

For example

```
mmm publish local
```

builds the project and publishes the module using the local resolver.

run

The command `run` runs the application on the Simulator (executes Easyant with `compile,simulator:run` targets). It has the following requirements:

- to run on the Simulator, the project must be configured with one of the following *Module Natures*:
  - *Sandboxed Application*
  - *Standalone Application*
  - *Add-On Library*
- the property `application.main.class` must be set to the Fully Qualified Name of the application main class (for example `com.mycompany.Main`)
- a MicroEJ Platform must be provided (see *Platform Selection* section)
- *Standalone Application Options* must be defined using properties file under in the `build` directory (see *Using a Properties File* section)
- the module must have been built once before running the Simulator. So the `mmm build` command must be executed before running the Simulator the first time or after a project clean (`mmm clean` command).

> **Note:** The next times, it is not required to rebuild the module if source code files have been modified. The content of `src/main/java` and `src/main/resources` folders are automatically compiled by `mmm run` command before running the Simulator.

For example

```
mmm run -D"platform-loader.target.platform.file"="/path/to/the/platform.zip"
```

runs the application on the given platform.

The Simulator can be launched in debug mode by setting the property `execution.mode` of the application file `build/commons.properties` to `debug`:

```
execution.mode=debug
```

The debug port can be defined with the property `debug.port`. Go to *Simulator Debug options section* for more details.

help

The command `help` displays the help for a command. For example

```
mmm help run
```

displays the help of the command `run`.

3.12.9 Build System Options

MMM allows to modify the behavior of a build via System options. These options must be passed as system properties, using *CLI* `-D` option or via the *SDK Configuration options*. MMM provides the following options:

- `easyant.debug.port` : defines the debug port and triggers the debug mode for the build execution.

3.12.10 Meta Build

A Meta Build is a module allowing to build other modules. It is typically used in a project containing multiple modules. The Meta Build module serves as an entry point to build all the modules of the project.

Meta Build creation

- In the SDK, select File > New > Module Project.

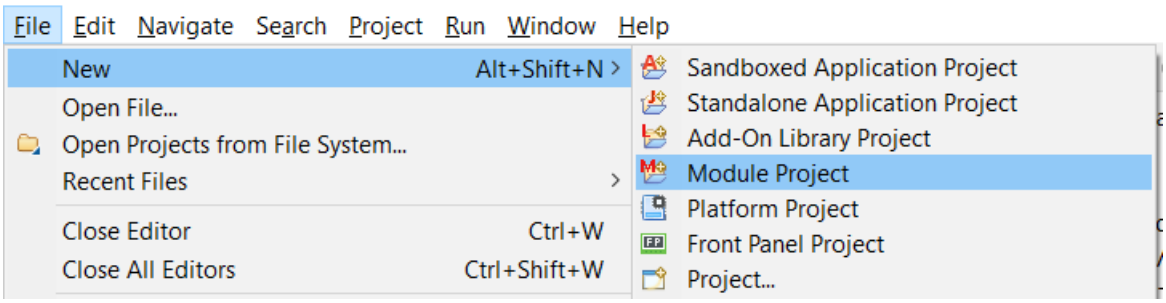


Fig. 106: New Meta Build Project

- Fill in the fields `Project name`, `Organization`, `Module` and `Revision`, then select the `Skeleton` named `microej-meta-build`

- Click on Finish . A template project is automatically created and ready to use.

Meta Build configuration

The main element to configure in a meta build is the list of modules to build. This is done in 2 files, located at the root folder:

- `public.modules.list` which contains the list of the modules relative paths to build and publish.
- `private.modules.list` which contains the list of the modules relative paths to build. These modules are not published but only stored in a private and local repository in order to be fetched by the public modules.

The format of these files is a plain text file with one module path by line, for example:

```
module1
module2
module3
```

These paths are relative to the meta build root folder, which is set by default to the parent folder of the meta build module (`..`). For this reason, a meta build module is generally created at the same level of the other modules to build. Here is a typical structure of a meta build:

```
/
├─ module1
│  ├─ ...
│  └─ module.ivy
├─ module2
│  ├─ ...
│  └─ module.ivy
├─ module3
│  ├─ ...
│  └─ module.ivy
└─ metabuild
   ├─ private.modules.list
   ├─ public.modules.list
   └─ module.ivy
```

The modules build order is calculated based on the dependency information. If a module is a dependency of another module, it is built first.

For a complete list of configuration options, please refer to *Meta Build Module Nature* section.

3.12.11 Troubleshooting

Unresolved Dependency

If the following message appears when resolving module dependencies:

```
:: problems summary ::
:::: WARNINGS
   module not found: com.mycompany#mymodule;[M.m.p-RC,M.m.(p+1)-RC[

            ::::::::::::::::::::::::::::::::::::::::::::::
```

(continues on next page)

(continued from previous page)

```
            ::          UNRESOLVED DEPENDENCIES         ::

            ::::::::::::::::::::::::::::::::::::::::::::::

            :: com.mycompany#mymodule;[M.m.p-RC,M.m.(p+1)-RC[: not found

            ::::::::::::::::::::::::::::::::::::::::::::::
```

First, check that either a released module `com.mycompany/mymodule/M.m.p` or a snapshot module `com.mycompany/mymodule/M.m.p-RCYYYYMMDD-HHMM` exists in your module repository.

- If the module does not exist,
  - if it is declared as a *direct dependency*, the module repository is not compatible with your source code. You can either check if an other module version is available in the repository or add the missing module to the repository.
  - otherwise, this is likely a missing transitive module dependency. The module repository is not consistent. Check the module repository settings file and that consistency check has been enabled during the module repository build (see *Configure Consistency Check*).
- If the module exists, this may be either a configuration issue or a network connection error. We have to find the cause in the resolution logs.

Note:

The activation of the verbose mode depends on how the resolution has been launched:

- if the error occurs during workspace resolution, configure the verbose mode of *resolution logs*,
- if the error occurs while building a module from workspace, check the verbose mode option in *preferences page*,
- if the error occurs while building a module from command line, set the verbose mode option in *command line options*.

For URL repositories, find:

```
trying https://[MY_REPOSITORY_URL]/[MY_REPOSITORY_NAME]/com.mycompany/mymodule/
tried https://[MY_REPOSITORY_URL]/[MY_REPOSITORY_NAME]/com.mycompany/mymodule/
```

For filesystem repository, find:

```
trying [MY_REPOSITORY_PATH]/com.mycompany/mymodule/
tried [MY_REPOSITORY_PATH]/com.mycompany/mymodule/
```

If your module repository URL or filesystem path does not appear, check your *settings file*. This is likely a missing resolver.

Otherwise, if your module repository is an URL, this may be a network connection error between MMM (the client) and the module repository (the server). First, check for *Invalid Certificate* issue.

Otherwise, the next step is to debug at the HTTP level:

```
HTTP response status: [RESPONSE_CODE] url=https://[MY_REPOSITORY_URL]/com.mycompany/
↪mymodule/
CLIENT ERROR: Not Found url=https://[MY_REPOSITORY_URL]/com.mycompany/mymodule/
```

Depending on the HTTP error code:

- `401 Unauthorized` : check your settings file credentials configuration.
- `404 Not Found` : add the following options to log raw HTTP traffic:

```
-Dorg.apache.commons.logging.Log=org.apache.commons.logging.impl.SimpleLog -Dorg.
↪apache.commons.logging.simplelog.showdatetime=true -Dorg.apache.commons.logging.
↪simplelog.log.org.apache.http=DEBUG -Dorg.apache.commons.logging.simplelog.log.
↪org.apache.http.wire=ERROR
```

Particularly, Ivy requires the HTTP `HEAD` request which may be disabled by some servers.

Invalid Certificate

If the following message appears when resolving module dependencies:

```
HttpClientHandler: sun.security.validator.ValidatorException: PKIX path building failed: sun.
↪security.provider.certpath.SunCertPathBuilderException: unable to find valid certification␣
↪path to requested target url=[artifactory address]
```

This can be raised in several cases, such as:

- an artifact repository configured in the MicroEJ Module Manager settings using a self-signed SSL certificate or a SSL certificate not trusted by the JDK.
- the requests to an artifact repository configured in the MicroEJ Module Manager settings are redirected to a proxy server using a SSL certificate not trusted by the JDK.

In all cases, the SSL certificate (used by the artifact repository server or the proxy) must be added to the JDK trust store that is running MicroEJ Module Manager. Ask your System Administrator, or retrieve the SSL certificate and add it to the JDK trust store:

- on Windows
  1. Install Keystore Explorer.
  2. Start Keystore Explorer, and open file `[JRE_HOME]/lib/security/cacerts` or `[JDK_HOME]/jre/lib/security/cacerts` with the password `changeit` . You may not have the right to modify this file. Edit rights if needed before opening it or open Keystore Explorer with admin rights.
  3. Click on `Tools` , then `Import Trusted Certificate` .
  4. Select your certificate.
  5. Save the `cacerts` file.
- on Linux/macOS
  1. Open a terminal.
  2. Make sure the JDK's `bin` folder is in the `PATH` environment variable.
  3. Execute the following command:

```
keytool -importcert -v -noprompt -trustcacerts -alias myAlias -file /path/to/the/
↪certificate.pem -keystore /path/to/the/truststore -storepass changeit
```

If the problem still occurs, set the `javax.net.debug` property to `all` to enable SSL protocol traces:

- when using the MMM CLI, add the property in the command line with: `-Djavax.net.debug=all`

- when using the Build Module button in the SDK, add the property in the MicroEJ Module Manager options as described in the section *Options*
- when resolving the dependencies on a project in the SDK with the button Ivy > Resolve, add the following line at the end of the file `MicroEJ-SDK.ini` located at the root of the SDK installation:

  ```
  -Djavax.net.debug=all
  ```

  and start the SDK from a terminal.

In all cases, such logs should appear in the terminal or in the SDK console:

```
...
javax.net.ssl|DEBUG|01|main|2022-09-09 18:22:20.828 CEST|SSLContextImpl.
↪java:428|System property jdk.tls.client.cipherSuites is set to 'null'
javax.net.ssl|DEBUG|01|main|2022-09-09 18:22:20.871 CEST|SSLCipher.java:464|jdk.
↪tls.keyLimits:  entry = AES/GCM/NoPadding KeyUpdate 2^37. AES/GCM/
↪NOPADDING:KEYUPDATE = 137438953472
javax.net.ssl|DEBUG|01|main|2022-09-09 18:22:20.892 CEST|SSLContextImpl.
↪java:402|Ignore disabled cipher suite: TLS_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA
...
```

There should be a trace at the beginning which indicates the path of the truststore used by the JDK:

```
javax.net.ssl|FINE|01|main|2022-09-05 14:34:38.631 CEST|TrustStoreManager.
↪java:112|trustStore is: /path/to/the/truststore
```

The error very probably occurs during the handshake phase of the SSL negotiation. There should be the following trace before the error:

```
Consuming server Certificate handshake message
```

The traces below this one indicates the SSL certificate (or the SSL certificates chain) presented by the server. This certificate or one of the root or intermediate certificates must be added in the JDK truststore as explained previously.

Target “simulator:run” does not exist

If the following message appears when executing the `mmm run` command:

```
* Problem Report:

Target "simulator:run" does not exist in the project "my-app".
```

it means that the command `run` is not supported by the build type declared by your module project. Make sure it is one of the following ones:

- `build-application`, with version `7.1.0` or higher
- `build-microej-javalib`, with version `4.2.0` or higher
- `build-firmware-singleapp`, with version `1.3.0` or higher

3.12.12 Former SDK Versions (lower than 5.2.0)

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

New MicroEJ Module Project

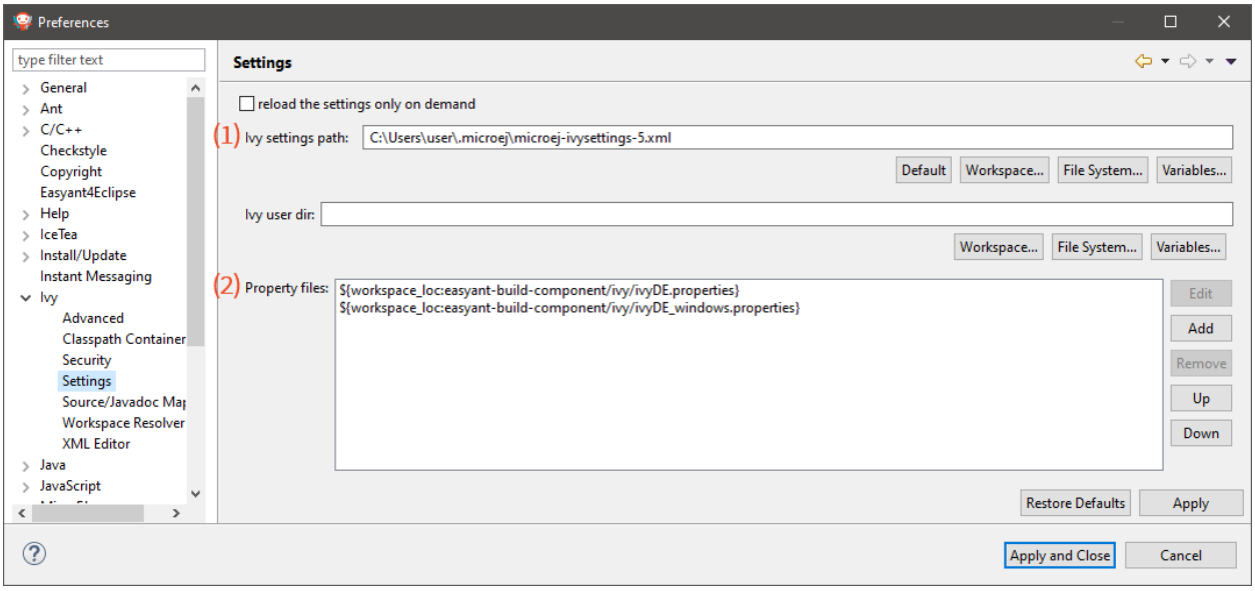
The New MicroEJ Module Project wizard is available at File > New > Project... , EasyAnt > EasyAnt Project .

Preferences Pages

MMM Preferences Pages are located in two dedicated pages. The following pictures show the options mapping using the same options numbers declared in *Preferences Page*.

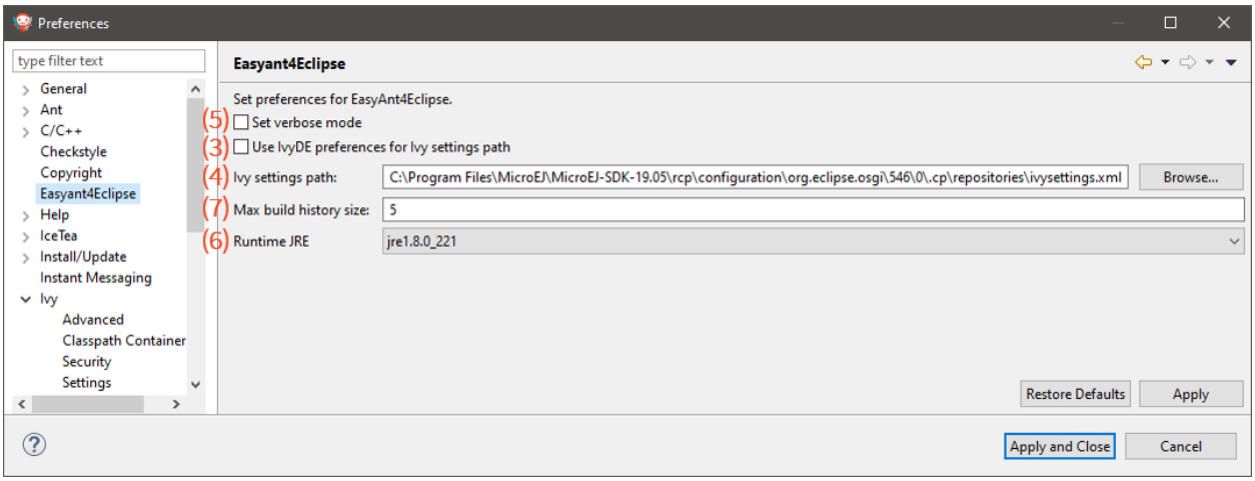
Ivy Preferences Page

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



Easyant Preferences Page

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



Build Kit

- Create an empty directory (e.g. `mmm_sdk_[version]_build_kit`),
- Locate your SDK installation plugins directory (by default, `C:\Program Files\MicroEJ\MicroEJ SDK-[version]\rcp\plugins` on Windows OS),
- Open the file `com.is2t.eclipse.plugin.easyant4e_[version].jar` with an archive manager,
- Extract the directory `lib` to the target directory,
- Open the file `com.is2t.eclipse.plugin.easyant4e.offlinerepo_[version].jar` with an archive manager,
- Navigate to directory `repositories`,
- Extract the file named `microej-build-repository.zip` for SDK `5.x` or `is2t_repo.zip` for SDK `4.1.x` to the target directory.

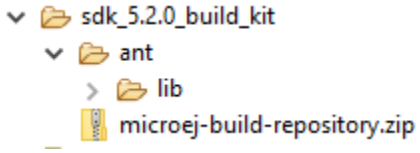
3.12.13 Former SDK Versions (from 5.2.0 to 5.3.x)

Build Kit

The Build Kit is bundled with the SDK and can be exported using the following steps:

- Select File > Export > MicroEJ > Module Manager Build Kit,
- Choose an empty Target directory,
- Click on the Finish button.

Once the Build Kit is fully exported, the directory content shall look like:



3.13 Release Notes

Starting from SDK version `5.0.0`, Architectures are distributed separately from the Integrated Development Environment. Evaluation Architectures can be downloaded from the Architectures Repository.

The SDK is now packaged into an Eclipse P2 repository (https://repository.microej.com/p2/sdk), allowing partial updates and installation on any compatible Eclipse version. The historical version (`5`) of MicroEJ is reused for the P2 repository delivery.

MicroEJ Corp. continues to regularly build all-in-one packages, called *Distributions*, including the SDK and dedicated OS installers. This distribution has a separate versioning, which follows modern convention: `[YY].[MM]` .

3.14 SDK Distribution Changelog

3.14.1 [24.01] - 2024-01-31

Note: This release requires a JDK 11 and therefore an Architecture `7.17.0` or higher. Please refer to *System Requirements* for more details.

- Included *SDK 5.8.2*.
- Enabled the “Terminate and Relaunch while launching” launcher option by default when a new Workspace is created.

3.14.2 [23.07] - 2023-07-03

Note: This release requires a JDK 11 and therefore an Architecture `7.17.0` or higher. Please refer to *System Requirements* for more details.

- Included *SDK 5.8.0*.
- Downgraded to Eclipse version `2022-03` to fix incompatibilities of components with JDK 11.
- Fixed no JDK found error when launching the installer in the case the JDK path has not been set in the Windows registry.

3.14.3 [23.02] - 2022-02-28

Note: This release requires a JDK 11 and therefore an Architecture `7.17.0` or higher. Please refer to *System Requirements* for more details.

- Included *SDK 5.7.0*.
- Updated to Eclipse version `2022-12` .

3.14.4 [22.06] - 2022-06-29

Note: This release requires a JDK 11 and therefore an Architecture `7.17.0` or higher. Please refer to *System Requirements* for more details.

- Included *SDK 5.6.0*.
- Added support for macOS aarch64 (M1 chip).
- Updated to Eclipse version `2022-03`.
- Changed required Java Runtime to JDK 11 (JRE and other versions are not supported anymore).

3.14.5 [21.11] - 2021-11-15

Note: This release prepares for a future JRE 11 support. However, the only officially supported JRE version is still JRE 8. Please refer to *System Requirements* for more details.

- Included *SDK 5.5.0*.
- Updated installer to accept both JRE 8 and JRE 11.
- Fixed error `Error while loading manipulator` when installing SDK updates on MacOS.
- Updated End User License Agreement.

3.14.6 [21.03] - 2021-03-25

- Included *SDK 5.4.0*.
- Updated End User License Agreement.

KNOWN ISSUES:

- The following error occurs when installing an SDK update on MacOS:

```
!MESSAGE Error while loading manipulator.
!STACK 0
java.lang.IllegalStateException: Error while loading manipulator.
    at org.eclipse.equinox.internal.p2.touchpoint.eclipse.LazyManipulator.
↪loadDelegate(LazyManipulator.java:64)
    at org.eclipse.equinox.internal.p2.touchpoint.eclipse.LazyManipulator.
↪getConfigData(LazyManipulator.java:117)
    at org.eclipse.equinox.internal.p2.touchpoint.eclipse.actions.
↪UninstallBundleAction.uninstallBundle(UninstallBundleAction.java:57)
    at org.eclipse.equinox.internal.p2.touchpoint.eclipse.actions.
↪UninstallBundleAction.execute(UninstallBundleAction.java:33)
    at org.eclipse.equinox.internal.p2.engine.ParameterizedProvisioningAction.
↪execute(ParameterizedProvisioningAction.java:42)
    at org.eclipse.equinox.internal.p2.engine.Phase.mainPerform(Phase.java:186)
    at org.eclipse.equinox.internal.p2.engine.Phase.perform(Phase.java:99)
    at org.eclipse.equinox.internal.p2.engine.PhaseSet.perform(PhaseSet.java:50)
    at org.eclipse.equinox.internal.p2.engine.Engine.perform(Engine.java:80)
```

(continues on next page)

(continued from previous page)

```
    at org.eclipse.equinox.internal.p2.engine.Engine.perform(Engine.java:48)
    at org.eclipse.equinox.p2.operations.ProvisioningSession.
↪performProvisioningPlan(ProvisioningSession.java:181)
    at org.eclipse.equinox.p2.operations.ProfileModificationJob.
↪runModal(ProfileModificationJob.java:76)
    at org.eclipse.equinox.p2.operations.ProvisioningJob.run(ProvisioningJob.
↪java:190)
    at org.eclipse.core.internal.jobs.Worker.run(Worker.java:63)
```

The workaround is to replace `/eclipse/plugins/` by `/Eclipse/plugins/` (capital `E`) in `MicroEJ-SDK-21.03.app\Contents\Eclipse\eclipse.ini`.

- See *SDK 5.4.0 Known Issues* section

3.14.7 [20.12] - 2020-12-11

- Included *SDK 5.3.1*
- Disabled Java version check when updating SDK (see *known issues of SDK Distribution 20.10*)

3.14.8 [20.10] - 2020-10-30

- Included *SDK 5.3.0*
- Updated to Eclipse version `2020-06`
- Fixed low quality MacOS SDK icons

Note: Starting with this release, only 64bits JRE are supported because 32bits JRE support has been removed since Eclipse version `2018-12`.

KNOWN ISSUES:

- Projects configured with Null Analysis must be updated to import EDC API 1.3.3 or higher in order to avoid an Eclipse JDT builder error (see also this link for more details).
- The default settings file for connecting MicroEJ Central Repository is not automatically installed. To connect to the MicroEJ Central Repository, follow the procedure:
  - For Windows, create the folder: `C:\Users\%USERNAME%\.microej`.
  - For Linux, create the folder: `/home/$USER/.microej`.
  - For macos, create the folder: `/Users/$USER/.microej`.
  - Download and save this file microej-ivysettings-5.xml to the previously created `.microej` folder.
- By default, a check is done on the JRE version required by the plugins on install/update. Since CDT requires JRE 11, it prevents to install/update a newer SDK version. The CDT documentation explains that this can be bypassed by disabling the option Windows > Preferences > Install/Update > Verify provisioning operation is compatible with currently running JRE.

3.14.9 [20.07] - 2020-07-28

- Included *MicroEJ SDK 5.2.0*
- Updated the default microej repository folder name (replaced SDK version by the distribution number)
- Added `Dist.` prefix in installer name (e.g. `MicroEJ SDK Dist. 20.07`) to avoid confusion between SDK distribution vs SDK version
- Updated SDK End User License Agreement
- Disabled popup window when installing a SDK update site (allow to install unsigned content by default)

3.14.10 [19.05] - 2019-05-17

- Included *SDK version 5.1.0*
- Updated MicroEJ icons (`16x16` and `32x32`)
- Updated the publisher of Windows executables (`MicroEJ` instead of `IS2T SA.`)
- Updated the JRE link to download in case the default JRE is not compatible. (`https://www.java.com` is deprecated)

3.14.11 [19.02] - 2019-02-22

- Updated to Eclipse Oxygen version `4.7.2`
- Included *SDK version 5.0.1*
- Included Sonarlint version `4.0.0`

3.15 SDK Changelog

3.15.1 [5.8.2] - 2024-01-31

General

- Added the `--keep-going` option for the MMM Command Line Interface to continue the build of the meta-build when a subproject fails.

MicroEJ Module Manager

General

- Upgraded Front Panel plugin to version 6.3.0 to use FP framework dependency only by default.

Build Types

- Set default Java compile version to 1.8 for `build-std-javalib` build type.
- Fixed hardcoded dependency line in generated javadoc of artifacts repositories.
- Fixed incompatibility of the Artifact Checker with modules published with the SDK 6.
- New build types added:
  - build-firmware-multiapp#8.2.0
  - build-firmware-singleapp#2.3.0
  - build-std-javalib#3.3.0

3.15.2 [5.8.1] - 2023-09-19

General

- Fixed unreadable tooltip because of black text on black background for the VEE Ports and Architectures views.
- Fixed wrong value for the example in the StackOverflow error message in the Memory Map Analyzer plugin.
- Fixed Configuration tab content disappearing when navigating in Run Configurations.

MicroEJ Module Manager

General

- Remove legacy configuration fields for application project wizard (Application ID, Printable Name and Description).

Build Types

- New build types added:
  - None

Skeletons

- Add section in README of the `build-addon-processor` skeleton to document how to override a generated source file.

3.15.3 [5.8.0] - 2023-07-03

General

- Added improvements in Outline view and Instance Browser view (new Owner column, new filters) of the Heap Viewer.
- Enabled on/off tags in the MicroEJ Java format profile.
- Updated Code template for Widget.handleEvent to use MWT 3 API.
- Fixed default Ivy settings file not created at startup.
- Fixed topological order in Application classpath.

MicroEJ Module Manager

General

Build Types

- Added Gradle dependency line in the generated Javadoc of an artifact repository (`build-artifact-repository` build type).
- New build types added:
  - build-artifact-repository#3.4.0
  - build-izpack#3.3.0

Skeletons

- Removed META-INF folder from `firmware-multiapp` skeleton.

3.15.4 [5.7.0] - 2023-02-27

General

- Added latest BSD license and SDK/BSD license and deprecate ESR.
- Added the capability to resolve a Front Panel dependency as a project in the workspace, as any other module type.
- Added the capability to resolve a Front Panel Mock dependency as a project in the workspace, as any other module type.
- Added the support to fetch Maven modules from MMM projects.
- Changed the error message displayed by the Memory Map Analyzer to show the real error message.
- Fixed build error when an ADP is opened in the workspace.
- Fixed slowness issue during Ivy resolution on Windows with JDK 11.
- Fixed syntaxic coloration lost in an opened module.ivy file after an SDK restart.
- Fixed inadequate colors in editors and console in Dark theme.

- Fixed failing Ivy resolution after an SDK restart.
- Fixed the freeze of the Heap Analyzer when opening a large heap file or clicking on a large byte array.
- Fixed error when building a VEE Port using the Build Platform button in the `.platform` file.
- Fixed "Resolve Foundation Library in workspace" option unchecked after closing and re-opening the workspace.
- Fixed Addon Processor modules not resolved when opened in the workspace.

MicroEJ Module Manager

General

- Fixed release version of a runtime API module.
- Fixed build of a module that uses the `obf-proguard` plugin with JDK 11.
- Upgraded ProGuard to version 7.2.1 to support JDK 11.
- Fixed Application external jars resolution at compile time.
- Fixed resolution in workspace error depending on a Mock's name.
- Fixed error message when an Easyant target is executed in a folder that does not contain a `module.ivy` file.

Build Types

- Fixed `build-std-javalib` compilation with JDK 11.
- Fixed Artifact Checker's execution on `build-std-javalib`.
- Fixed `build-artifact-repository` build type which couldn't find the previous release of the repository to merge it.
- New build types added:
  - build-addon-processor#2.2.0
  - build-application#9.2.0
  - build-artifact-repository#3.3.0
  - build-custom#2.2.0
  - build-firmware-customizer#3.2.0
  - build-firmware-multiapp#8.1.0
  - build-firmware-singleapp#2.2.0
  - build-izpack#3.2.0
  - build-microej-extension#2.2.0
  - build-microej-javaapi#5.2.0
  - build-microej-javaimpl#5.2.0
  - build-microej-javalib#6.2.0
  - build-microej-mock#2.2.0
  - build-microej-ri#3.2.0

  - build-microej-testsuite#4.2.0
  - build-product-java#2.2.0
  - build-runtime-api#4.1.0
  - build-std-javalib#3.2.0

Skeletons

- Aligned Kernel APIs dependencies between runtime-api and firmware-multiapp skeletons.
- Changed default compilation level to Java 8 for Mock projects.
- Fixed Build Executable options to make the “No BSP Connection” work.

3.15.5 [5.6.2] - 2022-08-31

General

- Fixed error when opening some heap dump files.
- Fixed error when saving a EJF file with the Font Designer.

MicroEJ Module Manager

General

- Fixed invalid module name when using spaces in the project name.

Skeletons

- Fixed wrong package name in the class generated when creating a firmware-multiapp project.

3.15.6 [5.6.1] - 2022-07-08

General

- Removed check on JRE version when opening a workspace.

3.15.7 [5.6.0] - 2022-06-29

General

- Added support for JDK 11.
- Changed Easyant targets executed by `mmm build` from `clean,verify` to `clean,package`.
- Upgraded Front Panel plugin to version 6.1.3 to remove warning on fp framework.
- Updated Workspace settings to ignore errors in Ant build files by default.
- Fixed error when opening a Heap Dump file not part of the workspace.

- Fixed error when opening a Map file not part of the workspace.
- Removed Resources Center view.

MicroEJ Module Manager

General

- Added the capability to override module organisation/name/revision with *Build System Options*.
- Added error message when using non-supported Eclipse Link Folders.
- Updated End User License Agreement.
- Fixed MMM failure when resolving a dependency with a version containing a number with 4 digits.
- Fixed error when building a meta-build project with public sub-modules and using target `verify`.

Build Types

- Added support for Kernel Runtime Environments (`build-firmware-multiapp` , `build-runtime-api` and `build-application`).
- Added option `javadoc.modules.excludes` to exclude modules from Javadoc generation when building a module repository.
- New build types added:
  - build-addon-processor#2.1.0
  - build-application#9.1.0
  - build-artifact-repository#3.2.0
  - build-custom#2.1.0
  - build-firmware-customizer#3.1.0
  - build-firmware-multiapp#8.0.0
  - build-firmware-singleapp#2.1.0
  - build-izpack#3.1.0
  - build-microej-extension#2.1.0
  - build-microej-javaapi#5.1.0
  - build-microej-javaimpl#5.1.0
  - build-microej-javalib#6.1.0
  - build-microej-mock#2.1.0
  - build-microej-ri#3.1.0
  - build-microej-testsuite#4.1.0
  - build-product-java#2.1.0
  - build-runtime-api#4.0.0
  - build-std-javalib#3.1.0

Build Plugins

- Updated `elf-utils` plugin to load the ELF related tools from the architecture/platform.

Skeletons

- Added JUnit dependency to all Java module skeletons (including default JUnit tests pattern).
- Updated `firmware-singleapp` and `firmware-multiapp` skeletons for building the executable by default.
- Updated Sandboxed Application skeleton (`application`) to be compatible with any Kernel (based on KF `FeatureEntryPoint`).

3.15.8 [5.5.3] - 2022-05-03

MicroEJ Module Manager

- Fixed error `Can't parse module descriptor` when building a Module on Windows with a JDK 8.0.331+.

3.15.9 [5.5.2] - 2021-12-22

General

- Fixed Addon Processors of a project in a workspace being applied to others projects.

MicroEJ Module Manager

Build Plugins

- Updated Log4j in Artifact Checker and Cobertura plugins to version 2.17.0.

3.15.10 [5.5.1] - 2021-12-02

General

- Fixed wrong category name in `New Project` wizard.

3.15.11 [5.5.0] - 2021-11-15

Note: This release prepares for a future JRE 11 support. However, the only officially supported JRE version is still JRE 8. Please refer to *System Requirements* for more details.

General

- Added Add-On Processor resolution in workspace.
- Updated tools for both JRE 8 and JRE 11 compatibility.
- Fixed corrupted font file created by the Font designer when importing large number of glyphs.
- Updated Architecture version check during Pack import (`greaterOrEqual` instead of `compatible`). This allows to import *Architecture Specific Pack* and *Legacy Generic Pack* on future Architecture versions `8.x` .
- Updated End User License Agreement.

MicroEJ Module Manager

- Added `bin` folder to `.gitignore` file of module natures Java project skeleton.
- Added Null Analysis configuration to `artifact-checker` . When building a module repository, null analysis configuration is only checked on the highest module version included in the repository.
- Added Eclipse Public License v2.0 to the list of default licenses allowed for `artifact-checker` .
- Clarified input messages of `mmm init` command.
- Updated `artifact-checker` plugin binding to target `verify` . This allow module checks to be executed on builds triggered by a pull request (no publication).
- Fixed missing `artifact-checker` plugin to some module natures (`custom` , `firmware-multiapp` , `firmware-singleapp` , `microej-javaimpl` , `microej-mock` , `microej-testsuite` , `product-java`).
- Fixed `mmm run` execution on a `firmware-singleapp` module (do not trigger the Firmware build).
- Fixed `kf-testsuite` plugin test project build.
- Added support of branch analysis with Sonar.
- Added ability to package private dependencies to `mock` module natures (configuration `embedded`).
- Added `testsuite` and `javadoc` plugin to `firmware-singleapp` module nature.
- Added `ssh` deployment to `microej-kf-testsuite` plugin.
- Updated `firmware-multiapp` to remove the `bsp` directory in Virtual Devices.
- Updated `firmware-multiapp` to allow Virtual Devices for launching a specific main class other than the Kernel main class. This is useful for running JUnit tests using a Virtual Device instead of a Platform.
- Updated `firmware-multiapp` to allow Virtual Devices for automatically launching a Sandboxed Application project in the SDK.
- Updated `firmware-multiapp` to automatically configure the Virtual Device Kernel UID when a Firmware is built.
- Fixed `firmware-multiapp` skeleton default dependencies with only modules available in MicroEJ Central Repository.
- Fixed `firmware-multiapp` unexpected build error when no declared pre-installed Application.
- Fixed `firmware-multiapp` build which may fail an unexpected `Unresolved Dependencies` error the first time, for Kernel APIs module dependencies (configuration `kernelapi`) or Virtual Device specific modules dependencies (configuration `default-vd`).
- Fixed `firmware-multiapp` unexpected build error when no Application (`.wpk` file) found in the dropins folder.

- Fixed `firmware-multiapp` unexpected build error when no declared pre-installed Application.
- Fixed `firmware-singleapp` and `firmware-multiapp` skeletons wrong package name generation for the default Main class.
- Fixed `artifact-repository` changelog check for modules with a snapshot version.
- New build types added:
  - build-addon-processor#2.0.0
  - build-application#9.0.0
  - build-artifact-repository#3.0.0
  - build-custom#2.0.0
  - build-firmware-customizer#3.0.0
  - build-firmware-multiapp#7.0.0
  - build-firmware-singleapp#2.0.0
  - build-izpack#3.0.0
  - build-microej-extension#2.0.0
  - build-microej-javaapi#5.0.0
  - build-microej-javaimpl#5.0.0
  - build-microej-javalib#6.0.0
  - build-microej-mock#2.0.0
  - build-microej-ri#3.0.0
  - build-microej-testsuite#4.0.0
  - build-product-java#2.0.0
  - build-runtime-api#3.0.0
  - build-std-javalib#3.0.0
  - microej-meta-build#3.0.0

3.15.12 [5.4.1] - 2021-04-16

Note: This release is both compatible with Eclipse version `2020-06` and Eclipse Oxygen, so it can still be installed on a previous SDK Distribution.

MicroEJ Module Manager

- Fixed missing `repository` configuration in `artifact-repository` skeleton (this configuration is required to include modules bundled in an other module repository)
- Fixed missing some old build types versions that were removed by error. (introduced in SDK `5.4.0`, please refer to the *Known Issues* section for more details)
- Fixed wrong version of module built in a meta-build (module was published with the module version instead of the snapshot version)
- Fixed code coverage analysis on source code (besides on bytecode) thanks to the property `cc.src.folders` (only for architectures in version 7.16.0 and beyond)
- New build types added:
  - microej-meta-build#2.0.1

3.15.13 [5.4.0] - 2021-03-25

Note: This release is both compatible with Eclipse version `2020-06` and Eclipse Oxygen, so it can still be installed on a previous SDK Distribution.

Known Issues

- Some older build types versions have been removed by error. Consequently, using SDK `5.4.0`, it may be not possible to build modules that have been created with an older SDK version (For example, MicroEJ GitHub code). The list of missing build types:
  - build-application `7.0.2`
  - build-microej-javalib `4.1.1`
  - build-firmware-singleapp `1.2.10`
  - build-microej-extension `1.3.2`

General

- Added MicroEJ Module Manager Command Line Interface in Build Kit
- Added ignore optional compilation problems in Addon Processor generated source folders
- Added logs to Standalone Application build indicating the mapping of Foundation Libraries to the Platform
- Updated End User License Agreement
- Added the latest HIL Engine API to mock-up skeleton (native resources management)
- Updated the Architecture import wizard to automatically accept Pack licenses when the Architecture license is accepted

MicroEJ Module Manager

General

- Added JSCH library to execute MicroEJ test suites on Device through ssh
- Added pre-compilation phase before executing Addon Processor to have compiled classes available
- Updated the default settings file to import modules from MicroEJ Developer repository (located at `${user.dir}\.microej\microej-ivysettings-5.4.xml`)

Build Types

- Updated all relevant build types to load the Platform using the `platform` configuration instead of the `test` configuration:
  - Sandboxed Application (`application`)
  - Foundation Library Implementation (`javaimpl`)
  - Addon Library (`javalib`)
  - MicroEJ Testsuite (`testsuite`)
- Updated Module Repository to allow to partially include an Architecture module (`eval` and/or `prod`)
- Fixed potential Addon Processor error `NoClassDefFoundError: ej/tool/addon/util/Message` depending on the resolution order
- Removed javadoc generation for `microej-extension`
- New build types added:
  - build-application#8.0.0
  - build-artifact-repository#2.3.0
  - build-firmware-singleapp#1.4.0
  - build-microej-extension#1.4.0
  - build-microej-javaimpl#4.0.0
  - build-microej-javalib#5.0.0
  - build-microej-testsuite#3.0.0

Build Plugins

- Updated Addon Processor to fail the build when an error is detected. Error messages are dumped to the build logs.
- Updated Platform Loader to handle Platform module (`.zip` file)
- Updated Platform Loader to handle Virtual Device module (`.vde` file) declared as a dependency. It worked before only by using the `dropins` folder.
- Updated Platform Loader to list the Platforms locations when too many Platform modules are detected

Skeletons

- Fixed wrong `README.md` generation for `artifact-repository` skeleton
- Removed useless files in `microej-javaapi`, `microej-javaimpl` and `microej-extension` skeletons (intern changelog and `.dbk` file)

3.15.14 [5.3.1] - 2020-12-11

Note: This release is both compatible with Eclipse version `2020-06` and Eclipse Oxygen, so it can still be installed on a previous SDK Distribution.

General

- Fixed missing default settings file for connecting MicroEJ Central Repository when starting a fresh install (introduced in `5.3.0`)

MicroEJ Module Manager

Build Plugins

- Fixed potential build error when computing Sonar classpath from dependencies (`ivy:cachepath` task was sometimes using a wrong cache location)

Skeletons

- Fixed skeleton dependency to EDC API 1.3.3 to avoid an Eclipse JDT builder error when Null Analysis is enabled (see *known issues of SDK Distribution 20.10*)

3.15.15 [5.3.0] - 2020-10-30

Note: This release is both compatible with Eclipse version `2020-06` and Eclipse Oxygen, so it can still be installed on a previous SDK Distribution.

Known Issues

- Library module build may lead to unexpected `Unresolved Dependencies` error in some cases (in `sonar:init` target / `ivy:cachepath` task). Workaround is to trigger the library build again.

General

- Fixed various plugins for Eclipse version `2020-06` compatibility (icons, project explorer menu entries)
- Fixed closed module.ivy files after an SDK restart that were opened before
- Removed license check before launching an Application on Device
- Disabled `Activate on new event` option of the Error Log view to prevent popup of this view when an internal error is thrown
- Removed license check before Platform build
- Updated filter of the Launch Group configuration (exclude the deprecated Eclipse CDT one)
- Fixed inclusion of mock project dependencies in launcher mock classpath
- Enhance error message in Platform editor (`.platform` files) when the required Architecture has not been imported (displays Architecture information)

MicroEJ Module Manager

General

- Fixed workspace default settings file when clicking on the `Default` button
- First wrong resolved dependency when ChainResolver returnFirst option is enabled and the module to resolve is already in the cache
- Fixed potential build module crash (`Not comparable` issue) when resolving module dependencies across multiple configurations

Build Types

- Exclude packs from artifact checker when building a module repository
- Merged Foundation & Add-On Libraries javadoc when building a module repository
- Added Module dependency line for each type in module repository javadoc
- Added an option to skip deprecated types, fields, methods in module repository javadoc
- Allow to include or exclude Java packages in module repository javadoc
- Added an option `skip.publish` to skip artifacts publication in `build-custom` build type
- Allow to define Application options from build option using the `platform-launcher.inject.` prefix
- Added generation and publication of code coverage report after a testsuite execution. The report generation is enabled under the following conditions:
  - at least one test is executed,
  - tests are executed on Simulator,
  - build option `s3.cc.activated` is set to `true` (default),
  - the Platform is based on an Architecture version `7.12.0` or higher
  - if testing a Foundation Library (using `microej-testsuite`), build option `microej.testsuite.cc.jars.name.regex` must be set to match the simple name of the library being covered (e.g. `edc-*.jar` or `microui-*.jar`)

- Fixed sonar false negative Null Analysis detection in some cases
- Added a better error message for Studio rebrand build when `izpack.microej.product.location` option is missing
- Deprecated `build-microej-ri` and disabled documentation generation (useless docbook toolchains have been removed to reduce the bundle size: `-150MB`)
- New build types added:
  - build-artifact-repository#2.0.1
  - build-custom#1.2.0
  - build-firmware-singleapp#1.2.10
  - build-microej-ri#2.4.0

Skeletons

- Fixed `microej-mock` content script initialization folder name

3.15.16 [5.2.0] - 2020-07-28

General

- Added `Dist.` prefix in default workspace and repository name to avoid confusion between SDK distribution vs SDK version
- Replaced `Version` by `Dist.` in Help > About MicroEJ® SDK menu. The SDK version is available in `Installation Details` view.
- Replaced `IS2T S.A.` and `MicroEJ S.A.` by `MicroEJ Corp.` in Help > About MicroEJ® SDK menu.
- Updated Front Panel plugin to version 6.1.1
- Removed MicroEJ Copyright in Java class template and skeletons files
- Fixed Stopping a MicroEJ launch in the progress view doesn't stop the launch

MicroEJ Module Manager

General

- Added a new configuration page (Window > Preferences > Module Manager). This page is a merge of formerly named `Easyant4Eclipse` preferences page and `Ivy Settings` relevant options for MicroEJ.
- Added Export > MicroEJ > Module Manager Build Kit wizard, to extract the files required for automating MicroEJ modules builds out of the IDE.
- Added New > MicroEJ > Module Project wizard (formerly named `New Easyant Project`), with module fields content assist and alphabetical sort of the skeletons list
- Added Import > MicroEJ > Module Repository wizard to automatically configure workspace with a module repository (directory or zip file)
- Added New MicroEJ Add-On Library Project wizard to simplify Add-On Library skeleton project creation

- Updated the build repository (`microej-build-repository.zip`) to be self contained with its owns `ivysettings.xml`
- Updated Virtual Device Player (`firmware-singleapp`) `launcher-windows.bat` (use `launcher-windows-verbose.bat` to get logs)
- Renamed the classpath container to `Module Dependencies` instead of `Ivy`
- Fixed Addon Processor `src-adpgenerated` folder generation when creating or importing a project with the same name than a previously deleted one
- Fixed implementation of settings ChainResolver returnFirst option
- Fixed Ivy module resolution being blocked from time to time

Build Types

- Fixed meta build to publish correct snapshot revisions for built dependencies. (Indirectly fixes ADP resolution issue when an Add-On Library and its associated Addon Processor were built together using a meta build)
- Fixed potential infinite loop when building a Modules Repository with MMM semantic enabled
- Fixed javadoc not being generated in artifactory repository build when `skip.javadoc` is set to `false`
- Added the capability to build partial modules repository, by using the user provided `ivysettings.xml` file to check the repository consistency
- Added the possibility to partially extend the build repository in a module repository. The build repository can be referenced by a file system resolver using the property `${microej-build-repository.repo.dir}`
- Added the possibility to include a module repository into an other module repository (using new configuration `repository->*`)
- Added the possibility to bundle a set of Virtual Devices when building a branded Studio. They are automatically imported to the MicroEJ repository when booting on a new workspace.
- Added the possibility to bundle a Module Repository when building a branded Studio. It is automatically imported and settings file is configured when booting on a new workspace.

Build Plugins

- Added variables `@MMM_MODULE_ORGANISATION@` , `@MMM_MODULE_NAME@` and `@MMM_MODULE_VERSION@` for README.md file
- Fixed `microej-kf-testsuite` repository access issue (introduced in SDK `5.0.0`).
- Fixed `artifact-checker` to accept revisions surrounded by brackets (as specified by https://keepachangelog.com/en/1.0.0/)

Skeletons

- Updated `module.ivy` indentation characters with tabs instead of spaces
- Updated `CHANGELOG.md` formatting
- Updated and standardized `README.md` files
- Updated dependencies in `module.ivy` to use the latest versions
- Added `.gitignore` to ignore the `target~` and `src-adpgenerated` folder where the module is built
- Added Sandboxed Application WPK dropins folder (`META-INF/wpk`)
- Removed conf `provided` in `module.ivy` for foundation libraries dependencies
- Remove MicroEJ internal site reference in `module.ant` file
- Fixed corrupted library `workbenchExtension-api.jar` in `microej-extension` skeleton
- Fixed corrupted library `HILEngine.jar` in `microej-mock` skeleton
- Fixed javadoc content issue in Main class `firmware-singleapp` skeleton

Misc

- Updated End User License Agreement
- Added support for generating Application Options in reStructured Text format

3.15.17 [5.1.2] - 2020-03-09

MicroEJ Module Manager

- Fixed potential build error when generating fixed dependencies file (`fixdeps` task was sometimes using a wrong cache location)
- Fixed topogical sort of classpath dependencies when building using `Build Module` (same as in `IvyDE` classpath sorted view)
- Fixed resolution of modules with a version `0.m.p` when transitively fetched (an error was thrown with the range `[1.m.p-RC,1.m.(p+1)-RC[`)
- Fixed missing classpath dependencies to prevent an error when building a standard JAR with JUnit tests

3.15.18 [5.1.1] - 2019-09-26

General

- Fixed files locked in `Platform in workspace` projects preventing the Platform from being deleted or rebuilt

3.15.19 [5.1.0] - 2019-05-17

General

- Updated MicroEJ icons (`16x16` and `32x32`)
- Fixed potential long-blocking operation when launching an application on a Virtual Device on Windows 10 (Windows defender performs a slow analysis on a zip file when it is open for the first time since OS startup)
- Fixed missing ADP resolution on a fresh MicroEJ installation
- Fixed ADP source folders order generation in `.classpath` (alphabetical sort of the ADP id)
- Fixed Run As... > MicroEJ Application automatic launcher creation: when selecting a `Platform in workspace` , an other platform of the repository was used instead
- Fixed `Memory Map Analyzer` load of mapping scripts from Virtual Devices
- Fixed MMM and ADP resolution when importing a zip project in a fresh MicroEJ install
- Fixed ADP crash when a project declares dependencies without a source folder
- Fixed inability to debug an application on a Virtual Device if option `execution.mode` was specified in firmware build properties (now SDK options cannot be overridden)
- Updated `Front Panel` plugin to comply with the new Front Panel engine
  - The Front Panel engine has been refactored and moved from UI Pack to Architecture (UI pack `12.0.0` requires Architecture version `7.11.0` or higher)
  - New Front Panel Project wizard now generates a project skeleton for this new Front Panel engine, based on MMM
  - Legacy Front Panel projects for UI Pack `v11.1.0` or higher are still valid
- Updated Virtual Device builder to speed-up Virtual Device boot time (pre-installed Applications are now extracted at build time)
- Fixed inability to select a `Platform in workspace` in a MicroEJ Tool launch configuration
- Fixed broken title in MicroEJ export menu (Platform Export)

MicroEJ Module Manager

Build Plugins

- Added a new option `application.project.dir` passed to launch scripts with the workspace project directory
- Updated MMM to throw a non ambiguous error message when a `module.ivy` configured for MMM declares versions with legacy Ivy range notation
- Updated MicroEJ Central Repository cache directory to `${user.dir}\.microej\caches\repository.microej.com-[version]` instead of `${user.dir}\.ivy2`
- Updated `Update Module Dependencies...` to be disabled when `module.ivy` cannot be loaded. The menu entry is now grayed when the project does not declare an IvyDE classpath container
- Fixed wrong resolution order when a module is both resolved in the repository and the workspace (the workspace module must always take precedence to the module resolved in the repository)

- Fixed useless `unknown resolver trace` when cache is used by multiple Ivy settings configurations with different resolver names.
- Fixed slow Add-On Processor generation. The classpath passed to ADP modules could contain the same entry multiple times, which leads each ADP module to process the same classpath multiple times.
- Fixed misspelled recommendation message when a build failed
- Fixed `Update Module Dependencies...` tool: wrong `ej:match="perfect"` added where it was expected to be `compatible`
- Fixed `Update Module Dependencies...` tool: parse error when `module.ivy` contains `<artifact type="rip"/>` element
- Fixed resolution and publication of a module declared with an Ivy branch
- Fixed character `'-'` rejected in module organisation (according to MMM specification `2.0-B`)
- Fixed ADP resolution error when the Add-On Processor module was only available in the cache
- Fixed potential build crash depending on the build kit classpath order (error was `This module requires easyant [0.9,+]`)
- Fixed `product-java` broken skeleton

Build Types

- Updated Platform Loader error message when the property `platform-loader.target.platform.dir` is set to an invalid directory
- Fixed meta build property substitution in `*.modules.list` files
- Fixed missing publications for `README.md` and `CHANGELOG.md` files
- Update skeletons to fetch latest libraries (Wadapps Framework `v1.10.0` and Junit `v1.5.0`)
- Updated `README.md` publication to generate MMM usage and the list of Foundation Libraries dependencies
- Added a new build nature for building platform options pages (`microej-extension`)
- Updated Virtual Device builder to speed-up Virtual Device boot time (pre-installed Applications are now extracted at build time)
- Fixed Virtual Device Player builder (dependencies were not exported into the zip file) and updated `firmware-singleapp` skeleton with missing configurations

Skeletons

- Updated `CHANGELOG.md` based on `Keep a Changelog` specification (https://keepachangelog.com/en/1.0.0/)
- Updated offline module repository skeleton to fetch in a dedicated cache directory under `${user.dir}/.microej/caches`

3.15.20 [5.0.1] - 2019-02-14

General

- Removed Wadapps Code generation (see migration notes below)
- Added support for MicroEJ Module Manager semantic (see migration notes below)
- Added a dedicated view for Virtual Devices in MicroEJ Preferences
- Removed Platform related views and menus in the SDK (Import/Export and Preferences)
- Added Studio rebranding capability (product name, icons, splash screen and installer for Windows)
- Added a new meta build version, with simplified syntax for multi-projects build (see migration notes below)
- Added a skeleton for building offline module repositories
- Added support for importing extended characters in Fonts Designer
- Allow to import Virtual Devices with `.vde` extension (`*.jpf` import still available for backward compatibility)
- Removed legacy selection for Types, Resources and Immutables in MicroEJ Launch Configuration (replaced by `*.list` files since MicroEJ `4.0`)
- Enabled IvyDE workspace dependencies resolution by default
- Enabled MicroEJ workspace Foundation Libraries resolution by default
- Added possibility for Architectures to check for a minimum required version of SDK (`sdk.min.version` property)
- Updated `New Standalone Application Project` wizard to generate a single-app firmware skeleton
- Updated Virtual Device Builder to manage Sandboxed Applications (compatible with Architectures Products `*_7.10.0` or newer)
- Updated Virtual Device Builder to include kernel options (now options are automatically filled for the application developer on Simulator)

MicroEJ Module Manager

Build Plugins

- Added IvyDE resolution from properties defined in Windows > Preferences > Ant > Runtime > Properties
- Fixed *Illegal character in path* error that may occur when running an Add-On Processor
- Fixed IvyDE crash when defining an Ant property file with Eclipse variables

Build Types

- Kept only latest build types versions (skeletons updated)
- Updated metabuild to execute tests by default for private module dependencies
- Removed remaining build dependencies to JDK (Java code compiler and Javadoc processors). All MicroEJ code is now compiled using the JDT compiler
- Introduced a new plugin for executing custom testsuite using MicroEJ testsuite engine
- Fixed *MalformedURLException* error in Easyant trace
- Fixed Easyant build crash when an Ivy settings file contains a cache definitions with a wildcard
- Updated Platform Builder to keep track in the Platform of the architecture on which it has been built (`architecture.properties`)
- Updated Virtual Device Builder to generate with `.vde` extension
- Updated Multi-app Firmware Builder to embed (Sim/Emb) specific modules (Add-On libraries and pre-installed Applications)
- Fixed `build-microej-ri` v1.2.1 missing dependencies (embedded in SDK 4.1.5)

Skeletons

- Updated all skeletons: migrated to latest build types, added more comments, copyright cleanup and configuration for MicroEJ Module Manager semantic)
- Added the latest HIL Engine API to mock-up skeleton (Start and Stop listeners hooks)

3.16 Build Types per SDK

- SDK 5.8.2
  - build-addon-processor#2.2.0
  - build-application#9.2.0
  - build-artifact-repository#3.4.0
  - build-custom#2.2.0
  - build-firmware-customizer#3.2.0
  - build-firmware-multiapp#8.2.0
  - build-firmware-singleapp#2.3.0
  - build-izpack#3.3.0
  - build-microej-extension#2.2.0
  - build-microej-javaapi#5.2.0
  - build-microej-javaimpl#5.2.0
  - build-microej-javalib#6.2.0
  - build-microej-mock#2.2.0
  - build-microej-ri#3.2.0

    - build-microej-testsuite#4.2.0
    - build-product-java#2.2.0
    - build-runtime-api#4.1.0
    - build-std-javalib#3.3.0
    - microej-meta-build#3.0.0
- SDK 5.8.0 and SDK 5.8.1
    - build-addon-processor#2.2.0
    - build-application#9.2.0
    - build-artifact-repository#3.4.0
    - build-custom#2.2.0
    - build-firmware-customizer#3.2.0
    - build-firmware-multiapp#8.1.0
    - build-firmware-singleapp#2.2.0
    - build-izpack#3.3.0
    - build-microej-extension#2.2.0
    - build-microej-javaapi#5.2.0
    - build-microej-javaimpl#5.2.0
    - build-microej-javalib#6.2.0
    - build-microej-mock#2.2.0
    - build-microej-ri#3.2.0
    - build-microej-testsuite#4.2.0
    - build-product-java#2.2.0
    - build-runtime-api#4.1.0
    - build-std-javalib#3.2.0
    - microej-meta-build#3.0.0
- SDK 5.7.0
    - build-addon-processor#2.2.0
    - build-application#9.2.0
    - build-artifact-repository#3.3.0
    - build-custom#2.2.0
    - build-firmware-customizer#3.2.0
    - build-firmware-multiapp#8.1.0
    - build-firmware-singleapp#2.2.0
    - build-izpack#3.2.0
    - build-microej-extension#2.2.0
    - build-microej-javaapi#5.2.0

  - build-microej-javaimpl#5.2.0
  - build-microej-javalib#6.2.0
  - build-microej-mock#2.2.0
  - build-microej-ri#3.2.0
  - build-microej-testsuite#4.2.0
  - build-product-java#2.2.0
  - build-runtime-api#4.1.0
  - build-std-javalib#3.2.0
  - microej-meta-build#3.0.0
- SDK 5.6.2, 5.6.1 and SDK 5.6.0
  - build-addon-processor#2.1.0
  - build-application#9.1.0
  - build-artifact-repository#3.2.0
  - build-custom#2.1.0
  - build-firmware-customizer#3.1.0
  - build-firmware-multiapp#8.0.0
  - build-firmware-singleapp#2.1.0
  - build-izpack#3.1.0
  - build-microej-extension#2.1.0
  - build-microej-javaapi#5.1.0
  - build-microej-javaimpl#5.1.0
  - build-microej-javalib#6.1.0
  - build-microej-mock#2.1.0
  - build-microej-ri#3.1.0
  - build-microej-testsuite#4.1.0
  - build-product-java#2.1.0
  - build-runtime-api#4.0.0
  - build-std-javalib#3.1.0
  - microej-meta-build#3.0.0
- SDK 5.5.3, SDK 5.5.2, SDK 5.5.1 and SDK 5.5.0
  - build-addon-processor#2.0.0
  - build-application#9.0.0
  - build-artifact-repository#3.0.0
  - build-custom#2.0.0
  - build-firmware-customizer#3.0.0
  - build-firmware-multiapp#7.0.0

  - build-firmware-singleapp#2.0.0
  - build-izpack#3.0.0
  - build-microej-extension#2.0.0
  - build-microej-javaapi#5.0.0
  - build-microej-javaimpl#5.0.0
  - build-microej-javalib#6.0.0
  - build-microej-mock#2.0.0
  - build-microej-ri#3.0.0
  - build-microej-testsuite#4.0.0
  - build-product-java#2.0.0
  - build-runtime-api#3.0.0
  - build-std-javalib#3.0.0
  - microej-meta-build#3.0.0
- SDK 5.4.1
  - build-addon-processor#1.0.3
  - build-application#8.0.0
  - build-artifact-repository#2.3.0
  - build-custom#1.2.0
  - build-firmware-customizer#2.0.1
  - build-firmware-multiapp#5.1.2
  - build-firmware-singleapp#1.4.0
  - build-izpack#2.0.1
  - build-microej-extension#1.4.0
  - build-microej-javaapi#4.0.4
  - build-microej-javaimpl#4.0.0
  - build-microej-javalib#5.0.0
  - build-microej-mock#1.0.3
  - build-microej-ri#2.4.0
  - build-microej-testsuite#3.0.0
  - build-product-java#1.2.4
  - build-runtime-api#2.0.2
  - build-std-javalib#2.0.1
  - microej-meta-build#2.0.1
- SDK 5.4.0
  - build-addon-processor#1.0.3
  - build-application#8.0.0

  - build-artifact-repository#2.3.0
  - build-custom#1.2.0
  - build-firmware-customizer#2.0.1
  - build-firmware-multiapp#5.1.2
  - build-firmware-singleapp#1.4.0
  - build-izpack#2.0.1
  - build-microej-extension#1.4.0
  - build-microej-javaapi#4.0.4
  - build-microej-javaimpl#4.0.0
  - build-microej-javalib#5.0.0
  - build-microej-mock#1.0.3
  - build-microej-ri#2.4.0
  - build-microej-testsuite#3.0.0
  - build-product-java#1.2.4
  - build-runtime-api#2.0.2
  - build-std-javalib#2.0.1
  - microej-meta-build#2.0.0
- SDK 5.3.1 and SDK 5.3.0
  - build-addon-processor#1.0.3
  - build-application#7.0.2
  - build-artifact-repository#2.0.1
  - build-custom#1.2.0
  - build-firmware-customizer#2.0.1
  - build-firmware-multiapp#5.1.2
  - build-firmware-singleapp#1.2.10
  - build-izpack#2.0.1
  - build-microej-extension#1.3.2
  - build-microej-javaapi#4.0.4
  - build-microej-javaimpl#3.2.2
  - build-microej-javalib#4.1.1
  - build-microej-mock#1.0.3
  - build-microej-ri#2.4.0
  - build-microej-testsuite#2.2.2
  - build-product-java#1.2.4
  - build-runtime-api#2.0.2
  - build-std-javalib#2.0.1

    - microej-meta-build#2.0.0
- SDK 5.2.0
    - build-addon-processor#1.0.3
    - build-application#7.0.2
    - build-artifact-repository#1.6.2
    - build-custom#1.1.3
    - build-firmware-customizer#2.0.1
    - build-firmware-multiapp#5.1.2
    - build-firmware-singleapp#1.2.9
    - build-izpack#2.0.1
    - build-microej-extension#1.3.2
    - build-microej-javaapi#4.0.4
    - build-microej-javaimpl#3.2.2
    - build-microej-javalib#4.1.1
    - build-microej-mock#1.0.3
    - build-microej-ri#2.3.1
    - build-microej-testsuite#2.2.2
    - build-product-java#1.2.4
    - build-runtime-api#2.0.2
    - build-std-javalib#2.0.1
    - microej-meta-build#2.0.0
- SDK 5.1.2, SDK 5.1.1 and SDK 5.1.0
    - build-addon-processor#1.0.3
    - build-application#7.0.2
    - build-artifact-repository#1.6.0
    - build-custom#1.1.3
    - build-firmware-customizer#2.0.1
    - build-firmware-multiapp#5.1.2
    - build-firmware-singleapp#1.2.9
    - build-izpack#2.0.1
    - build-microej-extension#1.3.2
    - build-microej-javaapi#4.0.4
    - build-microej-javaimpl#3.2.2
    - build-microej-javalib#4.1.1
    - build-microej-mock#1.0.3
    - build-microej-ri#2.3.1

- build-microej-testsuite#2.2.2
- build-product-java#1.2.4
- build-runtime-api#2.0.2
- build-std-javalib#2.0.1
- microej-meta-build#2.0.0

3.17 Migration Notes

3.17.1 From 5.2.x to 5.3.x or more

This section applies if MicroEJ SDK `5.3.x` is started on a workspace that was previously created using MicroEJ SDK `5.2.x`.

Workspace migration warning

Starting with the MicroEJ SDK Distribution 20.10, when opening a workspace which has been created with an older MicroEJ Distribution, a message is displayed with the following warning:

```
The workspace was written with an older version. Continue and update workspace which may␣
↪make it incompatible with older versions?
```

This is a generic warning from Eclipse which can be safely ignored as long as you don't intend to open it back with an older MicroEJ SDK Distribution then.

3.17.2 From 5.1.x to 5.2.x

This section applies if MicroEJ SDK `5.2.x` is started on a workspace that was previously created using MicroEJ SDK `5.1.x`.

Enable New Wizards Shortcuts in MicroEJ Perspective

Eclipse perspective settings are stored in the workspace metadata, so the new wizards shortcuts (`Add-On Library Project` and `Module Project`) are not visible in the `File` > `New` menu.

The MicroEJ perspective must be reset to its default settings as following:

- Click on `Windows` > `Perspective` > `Open Perspective` > `Other...` menu
- Select `MicroEJ` perspective
- Click on `Windows` > `Perspective` > `Reset Perspective...` menu
- Click on `Yes` button to accept to reset the MicroEJ perspective to its defaults.

The new wizards shortcuts are now visible into `File` > `New` menu.

Re-enable the Ivy Preferences Pages (Advanced Use)

The original Window > Preferences > Ivy pages can be re-enabled as following:

- Close all running instances of the SDK
- Edit `MicroEJ-SDK.ini` and add the property `-Dorg.apache.ivy.showAdvancedPrefs=true`
- Start the SDK again
- Go to Window > Preferences > Module Manager page

A new link `Ivy settings` should appear on the bottom of the page. It opens a popup window with the original Ivy preferences pages.

3.17.3 From 4.1.x to 5.x

This section applies if MicroEJ SDK `5.x` is started on a workspace that was previously created using MicroEJ SDK `4.1.x`.

Wadapps Application Update

The Wadapps code generator has been moved from IDE to an Addon Processor coming with `ej.library.wadapps.framework` module (`v1.9.0` or higher is required).

A Wadapps Application Project can be updated as follows:

- Right-click on the project, then Configure > Remove Sandboxed Application Nature
- Right-click on the project, then Configure > Add Sandboxed Application Nature
- Update `module.ivy` dependency to fetch `ej.library.wadapps.framework` version `1.9.0` (or perform MicroEJ Module Manager update as defined below)
- Delete remaining folder `src/.generated~` if any
- Check that project compiles and folder `src-adpgenerated/wadapps` is generated

MicroEJ Module Manager Update

It is highly recommended to migrate `module.ivy` to the MicroEJ Module Manager semantic, since the default Ivy resolution will be no more maintained in future versions.

The `module.ivy` can be updated as follows:

- Right-click on `module.ivy`, then Update Module Dependencies...

This has for effect to both migrate the `module.ivy` to the MicroEJ Module Manager semantic and also to update dependencies version to the latest available in the target repository.

Meta build Project Update

A project using `microej-meta-build` version `1.x` can be updated to version `2.x` as follows:

- Edit `module.ivy`
  - Replace the `microej-meta-build` version by `2.0.+`
  - Update all properties declaration to append the `metabuild.inject.` prefix (e.g. `<ea:property name="skip.test" value="true" />` must be updated to `<ea:property name="metabuild.inject.skip.test" value="true" />`)
  - Optionally remove or comment the root folder declaration as it is the default. (`<ea:property name="metabuild.root" value=".." />`)
- Delete `module.properties`. It only contains the property `easyant.fork.build=true`. This property is now automatically set by `easyant-build-component` since version `1.12.0`. Otherwise it must be explicitly injected by the build system as an Ant property: `easyant.inject.easyant.fork.build=true`
- Extract from `override.module.ant` the projects declarations lines:
  - Extract the project declarations of `local.submodule.dirs.id` into a new file named `private.modules.list` (one project per line)
  - Extract the project declarations of `submodule.dirs.id` into a new file names `public.modules.list` (one project per line)
- Delete `override.module.ant`

The new file system structure shall look like:

```
metabuild-project
    module.ivy
    private.modules.list
    public.modules.list
```

CHAPTER

FOUR

SDK 6 USER GUIDE

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

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

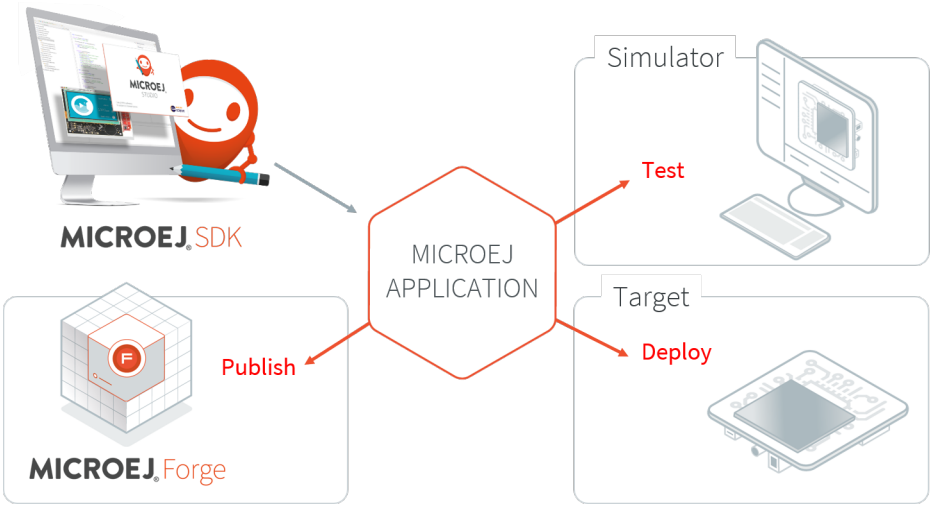


Fig. 1: MicroEJ Application Development Overview

Note: The SDK 6 is limited to the build, test and simulation of Applications and Add-On Libraries (see *Scope and Limitations* for more information). If you need other features, such as creating a VEE Port, you have to use *the SDK 5*.

The SDK is composed of the following main elements:

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

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

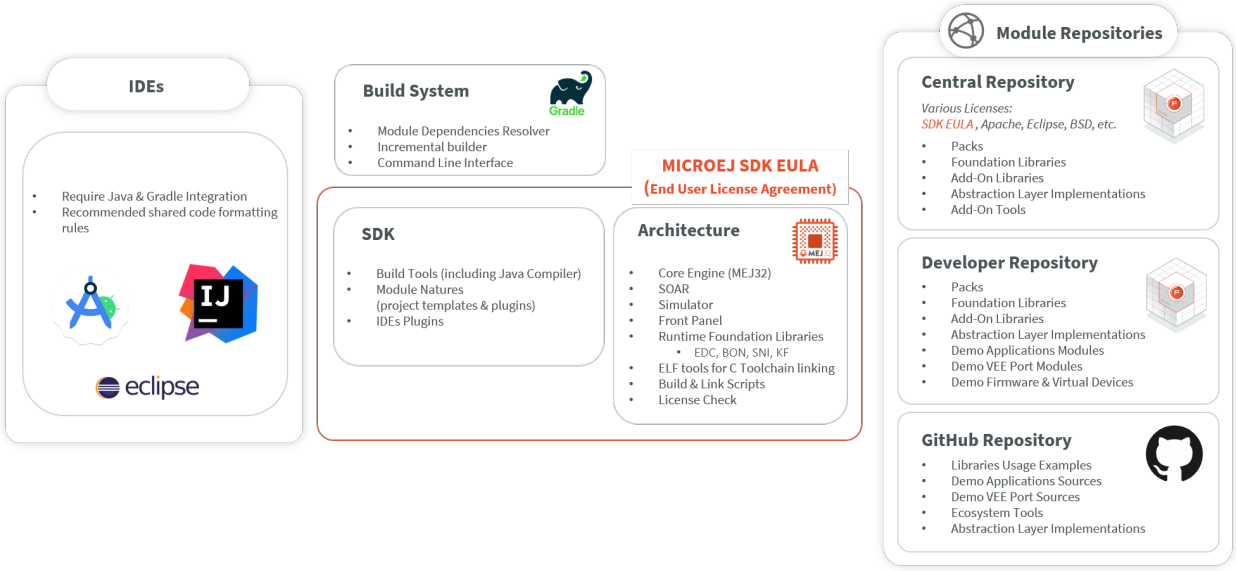


Fig. 2: SDK Detailed View

4.1 Getting Started

4.1.1 NXP

i.MX RT595 Evaluation Kit

During this Getting Started, you will learn to run:

- run an Application on the i.MX RT595 Evaluation Kit Virtual Device,
- run the same Application on your i.MX RT595 Evaluation Kit.

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

Prerequisites

Note: This Getting Started has been tested on Windows 10 & 11.

This Getting Started is separated in two main parts.

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

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

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

- i.MX RT595 Evaluation Kit, available here.
- G1120B0MIPI display panel, available here,
- A GNU ARM Embedded Toolchain, Cmake and Make are needed to build the BSP. You will be guided on how to install toolchain later.
- LinkServer tool to flash the board. You will be guided on how to install this tool later.

Environment Setup

To follow this Getting Started, you need to install:

- MICROEJ SDK 6.
- Get Demo-Wearable-VG.

Install MICROEJ SDK 6

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

Get Demo-Wearable-VG

For this Getting Started, the `Demo-Wearable-VG` Application will be use. You can download it using the following command:

```
git clone -b 2.0.0 https://github.com/MicroEJ/Demo-Wearable-VG.git
```

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

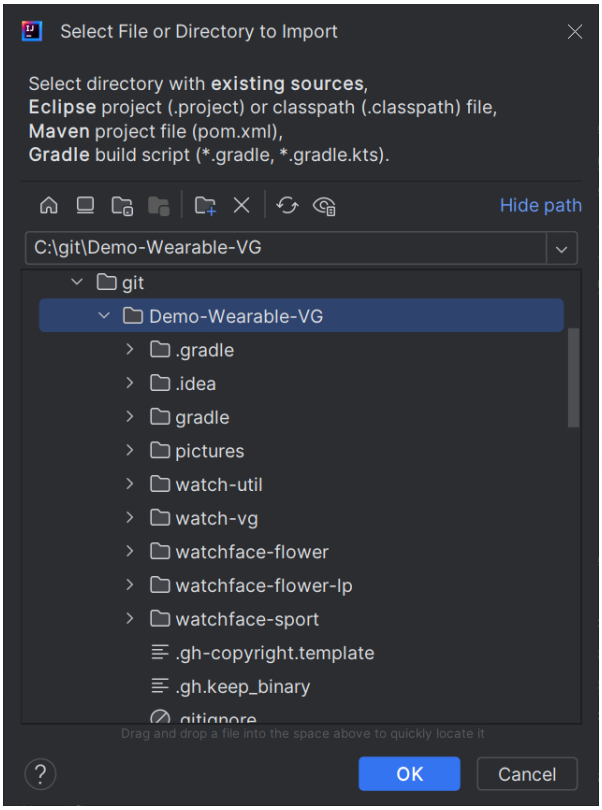
Set up the Application on your IDE

Import the Project

The first step is to import the `Demo-Wearable-VG` Application into your IDE:

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

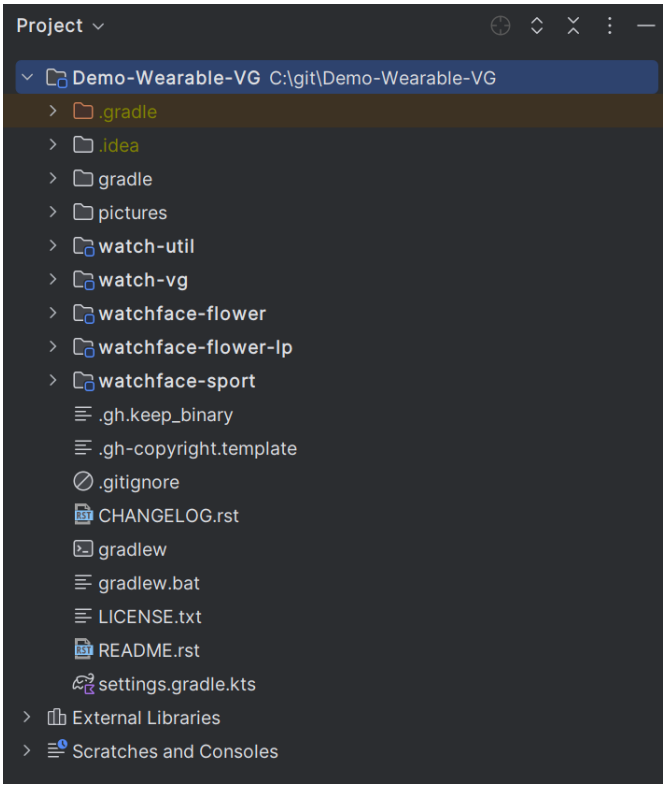
- If you are in the Welcome Screen, click on the `Open` button. Otherwise click either on `File` > `Open…` or on `File` > `New` > `Project From Existing Sources…` .
- Select the `Demo-Wearable-VG` directory located where you downloaded it and click on the `OK` button.



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

- Click on the `Create` button.

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

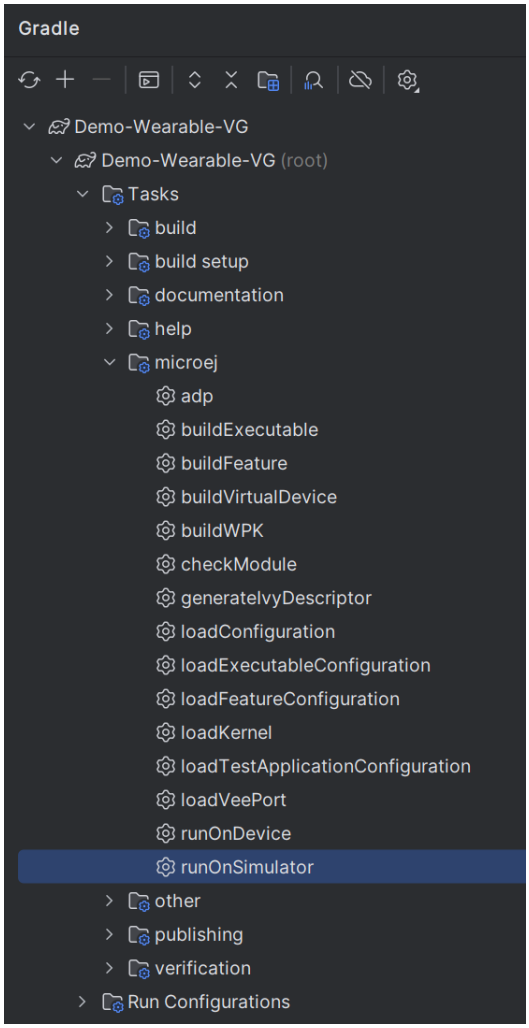


Run an Application on the Virtual Device

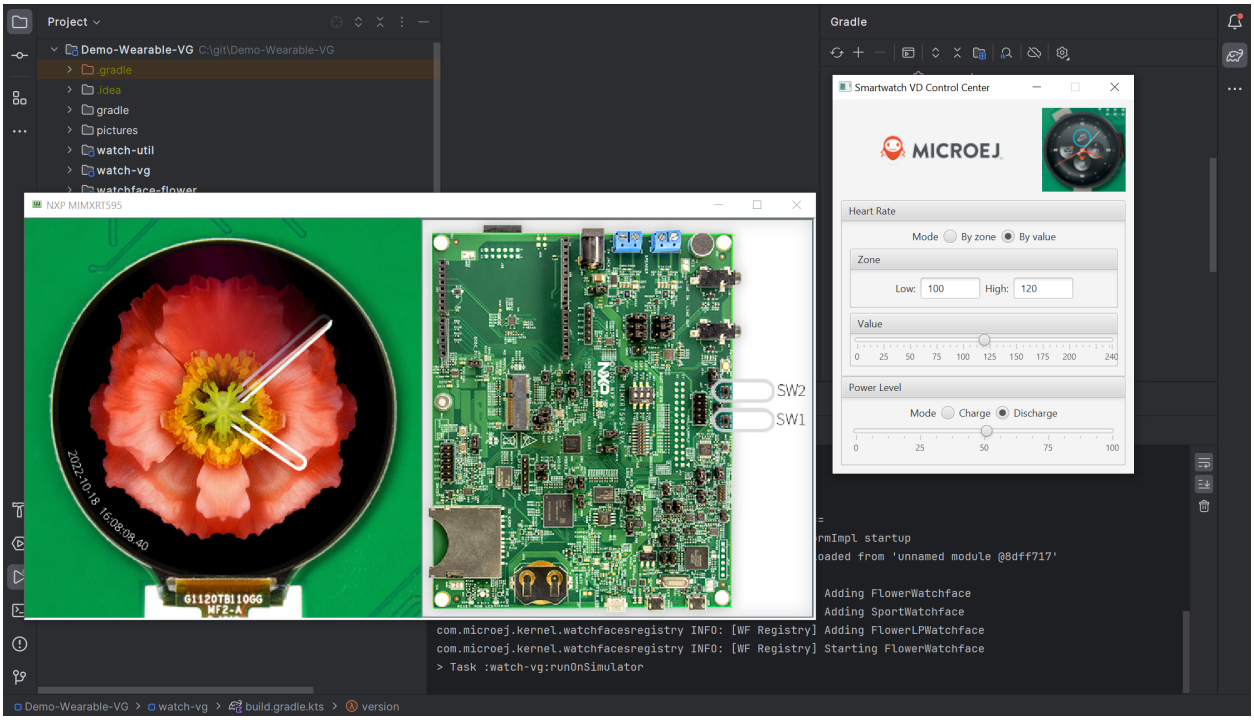
In order to execute the `Demo-Wearable-VG` Application on the Virtual Device, the SDK provides the Gradle `runOnSimulator` task.

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

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



The Virtual Device starts and executes the `Demo-Wearable-VG` application.



Note: If you want to know more about the use of the `Demo-Wearable-VG` , please have a look at its README.md file.



Well done !

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

If you want to learn how to run an application on your i.MX RT595 Evaluation Kit, you can continue this Getting Started: *Run an Application on i.MX RT595 Evaluation Kit*.

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

Run an Application on i.MX RT595 Evaluation Kit

To deploy `Demo-Wearable-VG` application on your board, you will have to:

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

Environment Setup

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

Install the C Toolchain

The C toolchain must be installed, it is composed of the GNU ARM Embedded Toolchain, CMake and Make.

Note: This Getting Started has been tested with the following configuration:

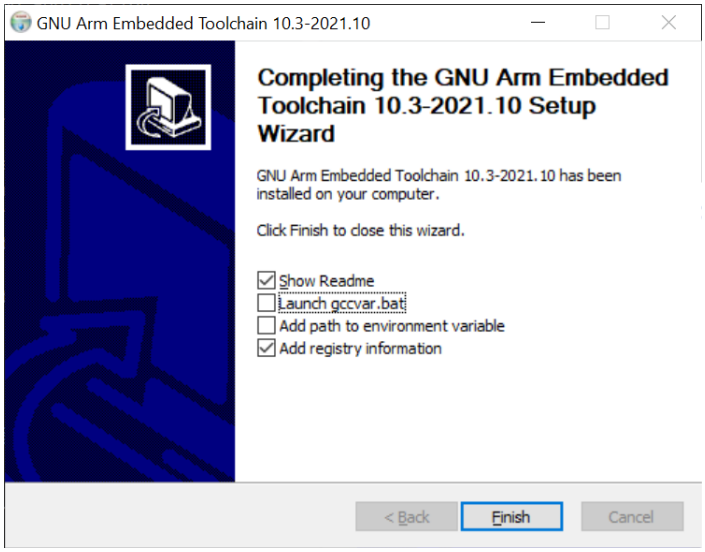
- GNU ARM Embedded Toolchain version `10.3 2021.10`.
- CMake version `3.26.5`.
- Make version `3.81`.

Later versions may or may not work, and may need modification to the Getting Started steps.

Install GNU ARM Embedded Toolchain

The toolchain is the GNU ARM Embedded Toolchain.

At the end of the installation, it will ask you to complete the Setup of the wizard, choose the following options:



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

- Open the "Edit the system environment variables" application on Windows.
- Click on the "Environment Variables..." button.
- Click on the "New..." button under the "User variables" section.
- Set "Variable Name" to `ARMGCC_DIR`.
- Set "Variable Value" to the toolchain directory (e.g. `C:\Program Files (x86)\GNU Arm Embedded Toolchain\10 2021.10`).
- Click on the "Ok" button until it closes "Edit the system environment variables" application.

Install CMake

CMake is the application used by the build system to generate the firmware.

During the installation, it will ask you if you wish to add CMake to your system Path, add it at least to the current user system path. If you missed it, you can manually add `CMake/bin` folder to your path.

Install Make

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

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

Install the Flashing Tool

Note: This Getting Started has been tested with LinkServer version `1.2.45`.

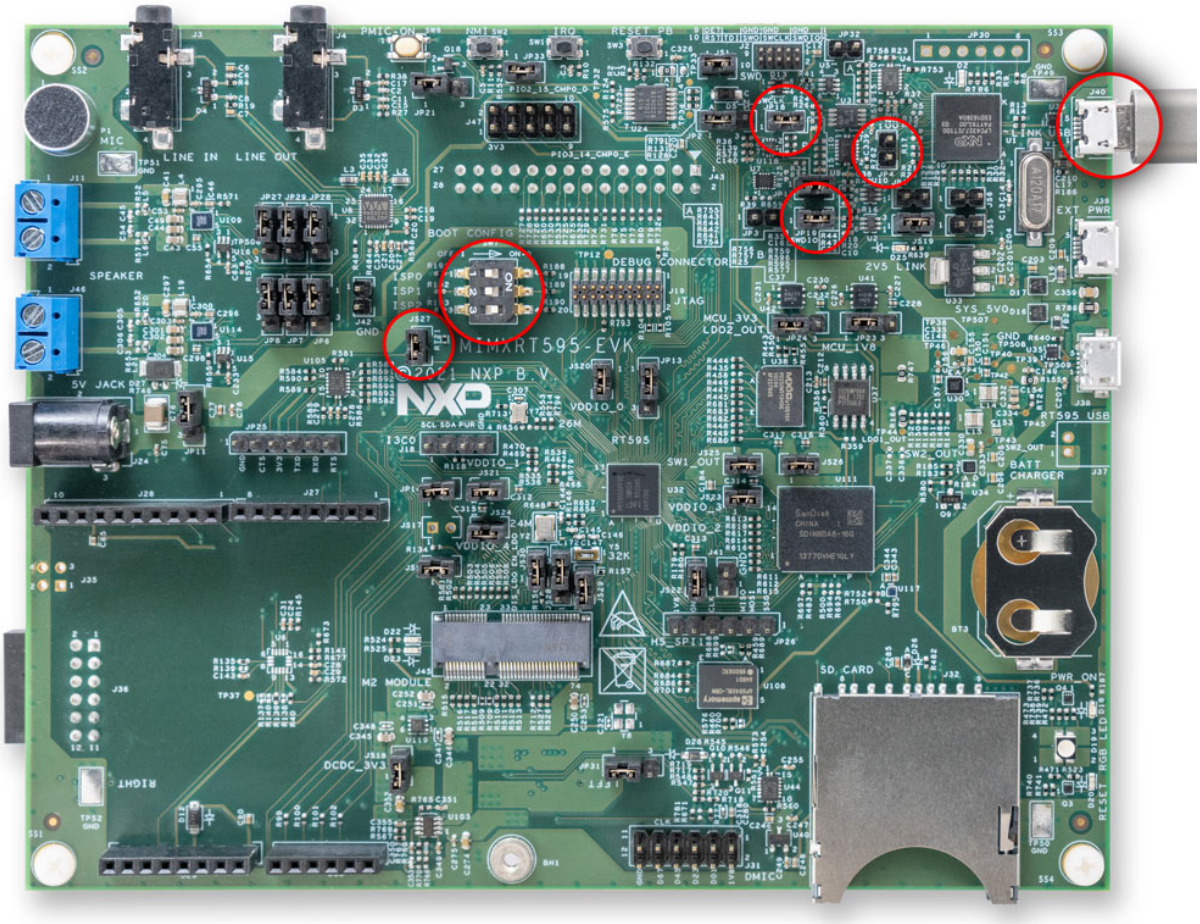
Later versions may or may not work, and may need modification to the Getting Started steps.

LinkServer is needed to flash the board.

Once installed, `LinkServer_xxx/binaries` folder must be set on your Path. To do so:

- Open the `Edit the system environment variables` application on Windows.
- Click on the `Environment Variables...` button.
- Select `Path` variable under the `User variables` section and edit it.
- Click on `New` and point to the `binaries` folder located where you installed LinkServer (e.g. `nxp/LinkServer_1.2.45/binaries`).

Hardware Setup



Setup the i.MX RT595 Evaluation Kit:

- Check that the dip switches (SW7) are set to OFF, OFF and ON (ISP0, ISP1, ISP2).
- Ensure jumpers JP18 and JP19 are closed.
- Remove jumper JP4.
- Connect the micro-USB cable to J40 to power the board.

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

A COM port is automatically mounted when the board is plugged into a computer using a USB cable. All board logs are available through this COM port.

The COM port uses the following parameters:

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

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

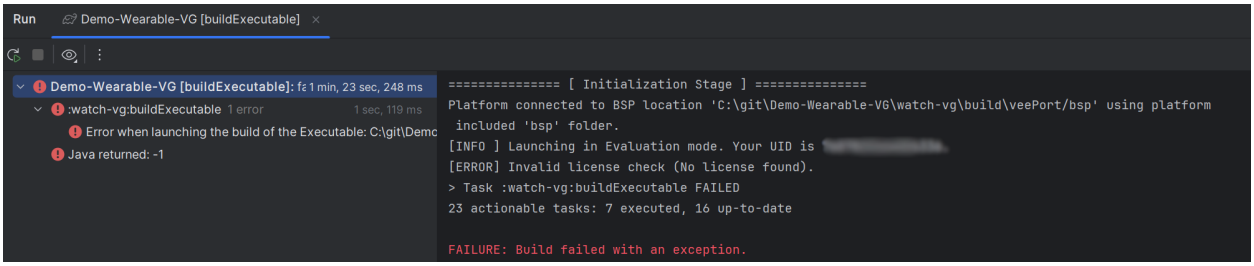
Congratulations, you have finished the setup of your environment. You are now ready to discover how to build and flash a MicroEJ application.

Build the Executable for i.MX RT595 Evaluation Kit

In order to build the Executable of the `Demo-Wearable-VG` Application, the SDK provides the Gradle `buildExecutable` task.

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

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



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

Request your Evaluation License:

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

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

The gradle task deploys the MicroEJ application in the BSP and then builds the BSP using Make.

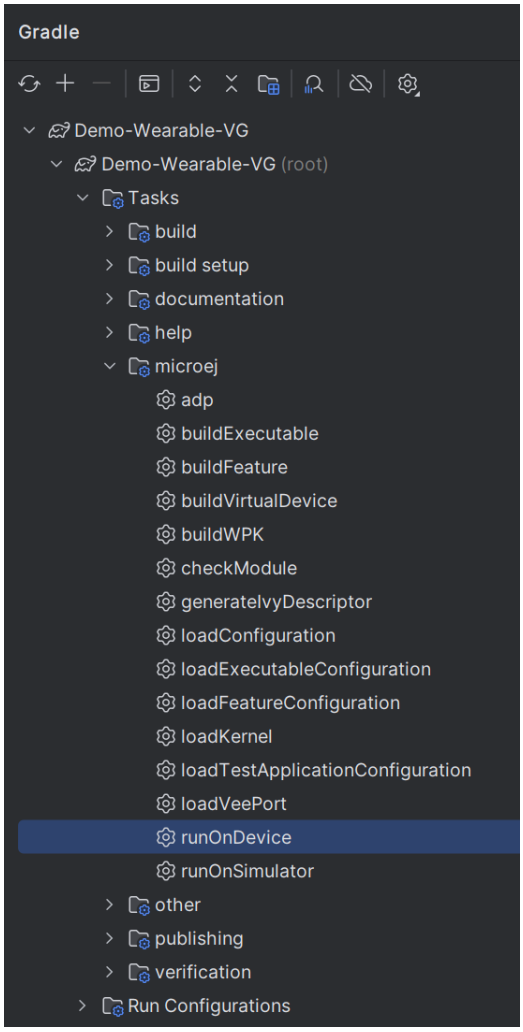
The `Demo-Wearable-VG` application is built and ready to be flashed on i.MX RT595 Evaluation Kit once the hardware setup is completed.

Flash the Application on the i.MX RT595 Evaluation Kit

In order to flash the `Demo-Wearable-VG` Application on i.MX RT595 Evaluation Kit, the application provides the Gradle `runOnDevice` task.

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

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



Once the firmware is flashed, you should see the `Demo-Wearable-VG` running on your board.

Modify the Java Application

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

For example, we could modify the color of the date on the Flower Watchface that is shown at the startup of the application.

- Open `FlowerWatchface.java` file located in the `watchface-flower/src/main/java/com/microej/demo/watch/watchface/flowe` folder.
- On the `renderDate` method, replace the following line:

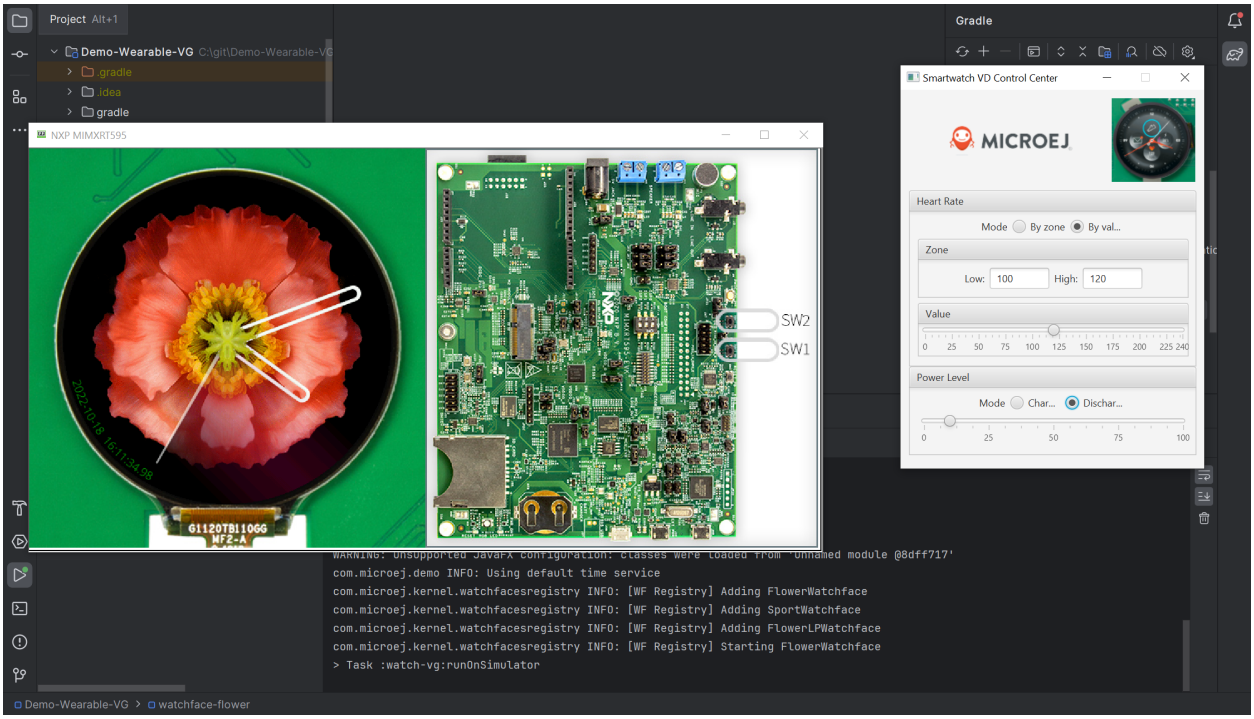
```
g.setColor(style.getColor());
```

by

```
g.setColor(Colors.GREEN);
```

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

Here is the modified application running in simulation:



i.MX RT1170 Evaluation Kit

During this Getting Started, you will learn to run:

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

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

Prerequisites

Note: This Getting Started has been tested on Windows 10 & 11.

This Getting Started is separated in two main parts.

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

- AnimatedMascot Application will be used in this Getting Started. Please contact *our support team* to get the code for this demo. It will soon be released publicly.
- An Internet connection to access Github repositories & *Module Repositories*.
- MICROEJ SDK 6 (installed during *Environment Setup*).

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

- i.MX RT1170 Evaluation Kit, available here.
- RK055HDMIPI4MA0 display panel, available here.
- A GNU ARM Embedded Toolchain, Cmake and Make are needed to build the BSP. You will be guided on how to install toolchain later.
- LinkServer tool to flash the board. You will be guided on how to install this tool later.

Environment Setup

To follow this Getting Started, you need to install:

- MICROEJ SDK 6.

Install MICROEJ SDK 6

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

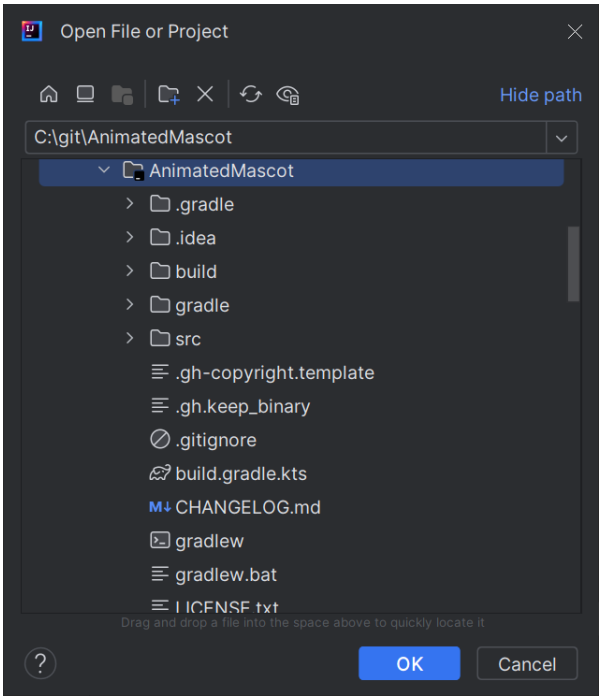
Set up the Application on your IDE

Import the Project

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

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

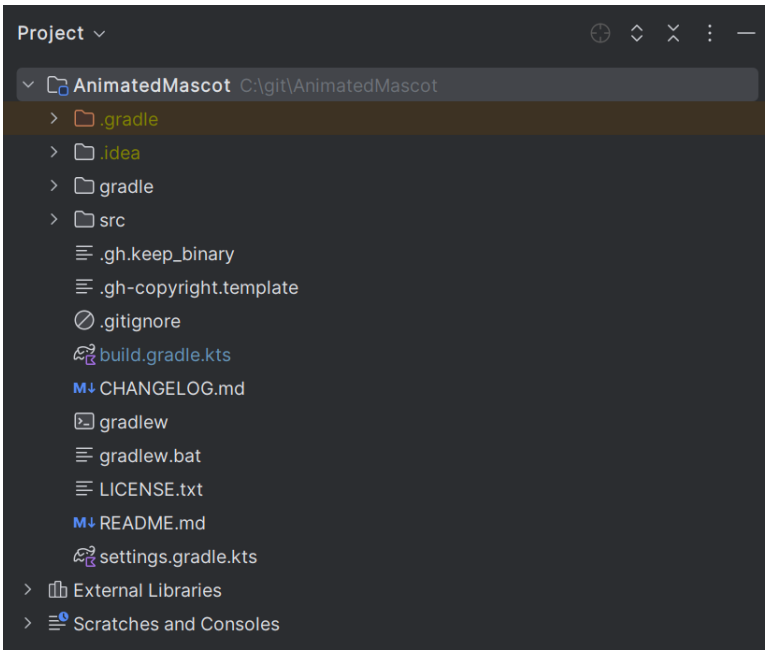
- If you are in the Welcome Screen, click on the Open button. Otherwise click either on File > Open… or on File > New > Project From Existing Sources… .
- Select the `AnimatedMascot` directory located where you downloaded it and click on the OK button.



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

- Click on the `Create` button.

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

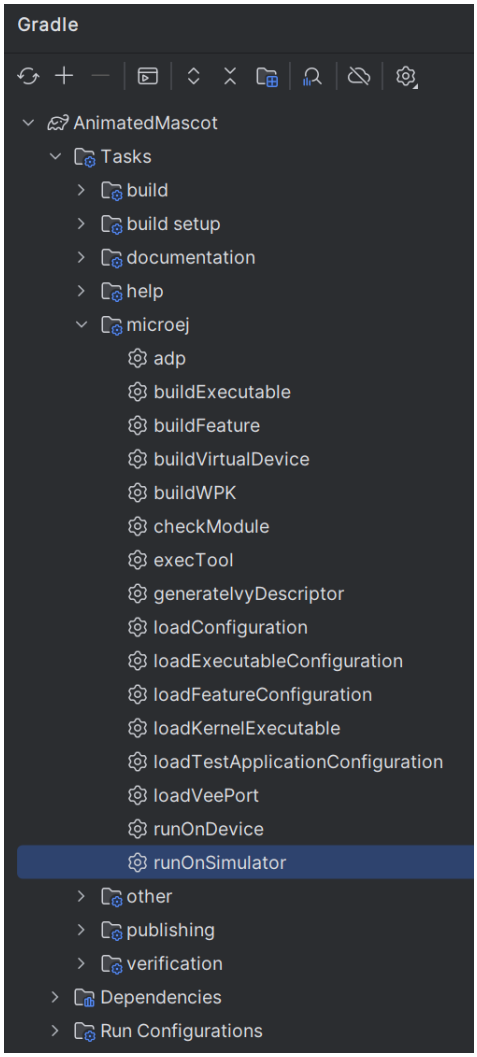


Run an Application on the Virtual Device

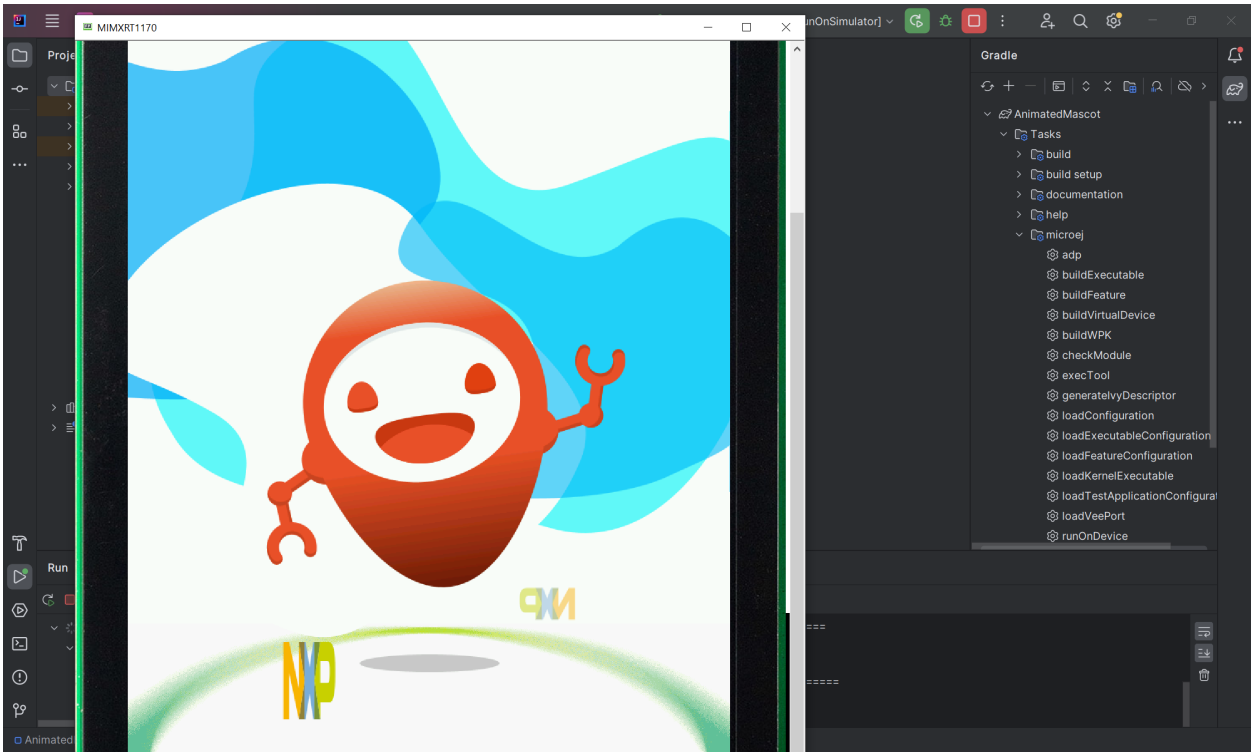
In order to execute the `AnimatedMascot` Application on the Virtual Device, the SDK provides the Gradle `runOnSimulator` task.

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

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



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



Note: The Front Panel may be too big for your screen, that is because of the RK055HDMIPI4MA0 display resolution. You can scroll down to see the bottom of the display.

If you want to know more about the use of the `AnimatedMascot` usage, please have a look at its `README.md` file.



Well done !

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

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

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

Run an Application on i.MX RT1170 Evaluation Kit

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

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

Environment Setup

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

Install the C Toolchain

The C toolchain must be installed, it is composed of the GNU ARM Embedded Toolchain, CMake and Make.

Note: This Getting Started has been tested with the following configuration:

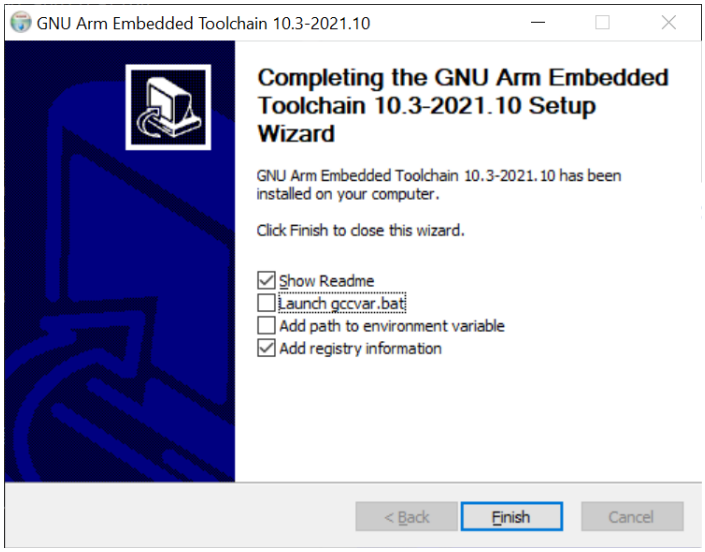
- GNU ARM Embedded Toolchain version `10.3 2021.10`.
- CMake version `3.26.5`.
- Make version `3.81`.

Later versions may or may not work, and may need modification to the Getting Started steps.

Install GNU ARM Embedded Toolchain

The toolchain is the GNU ARM Embedded Toolchain.

At the end of the installation, it will ask you to complete the Setup of the wizard, choose the following options:



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

- Open the `Edit the system environment variables` application on Windows.
- Click on the `Environment Variables...` button.
- Click on the `New...` button under the `User variables` section.
- Set `Variable Name` to `ARMGCC_DIR`.
- Set `Variable Value` to the toolchain directory (e.g. `C:\Program Files (x86)\GNU Arm Embedded Toolchain\10 2021.10`).
- Click on the `Ok` button until it closes `Edit the system environment variables` application.

Install CMake

CMake is the application used by the build system to generate the firmware.

During the installation, it will ask you if you wish to add CMake to your system Path, add it at least to the current user system path. If you missed it, you can manually add `CMake/bin` folder to your path.

Install Make

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

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

Install the Flashing Tool

Note: This Getting Started has been tested with LinkServer version `1.2.45`.

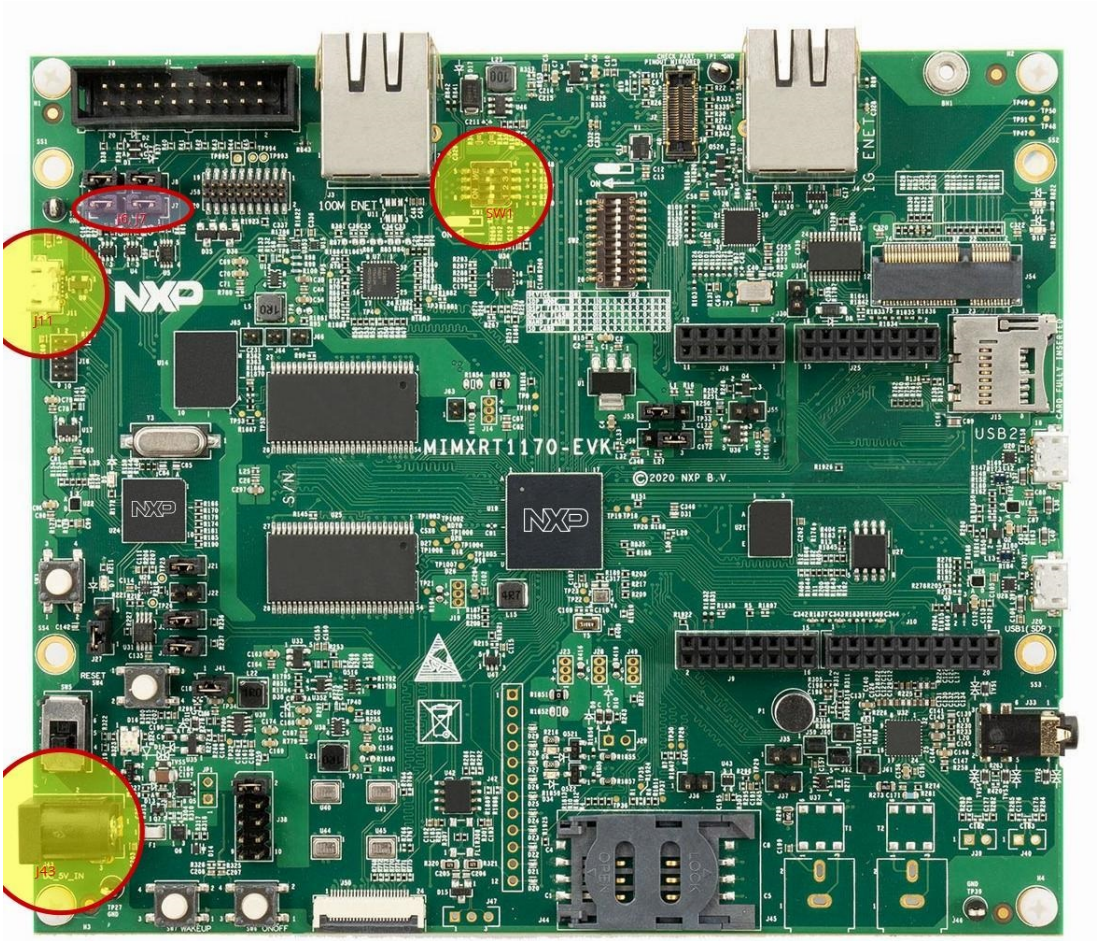
Later versions may or may not work, and may need modification to the Getting Started steps.

LinkServer is needed to flash the board.

Once installed, `LinkServer_xxx/binaries` folder must be set on your Path. To do so:

- Open the `Edit the system environment variables` application on Windows.
- Click on the `Environment Variables…` button.
- Select `Path` variable under the `User variables` section and edit it.
- Click on `New` and point to the `binaries` folder located where you installed LinkServer (e.g. `nxp/LinkServer_1.2.45/binaries`).

Hardware Setup



Setup the i.MX RT1170 Evaluation Kit

- Check that the dip switches (SW1) are set to OFF, OFF, ON and OFF.
- Ensure jumpers J6 and J7 are closed.
- Connect the micro-USB cable to J11 to power the board.
- You can connect 5 V power supply to J43 if you need to use the display

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

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

The COM port uses the following parameters:

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

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

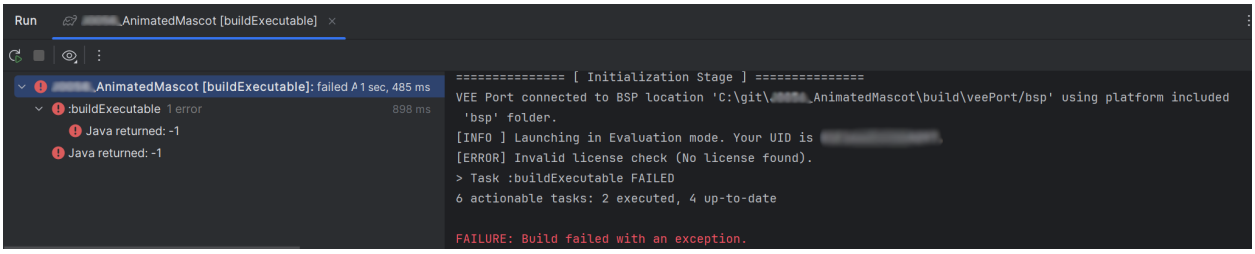
Congratulations, you have finished the setup of your environment. You are now ready to discover how to build and flash a MicroEJ application.

Build the Executable for i.MX RT1170 Evaluation Kit

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

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

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



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

Request your Evaluation License:

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

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

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

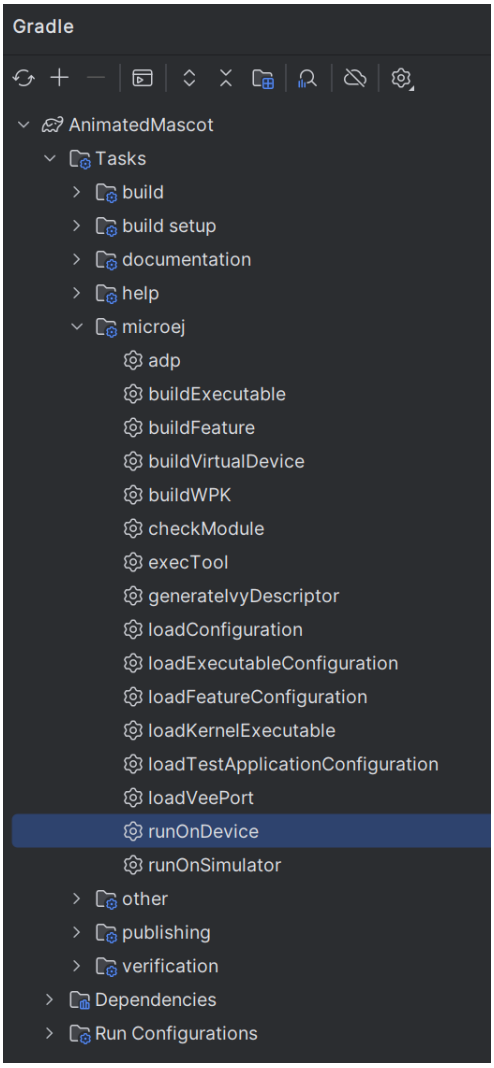
The `AnimatedMascot` application is built and ready to be flashed on i.MX RT1170 Evaluation Kit once the hardware setup is completed.

Flash the Application on the i.MX RT1170 Evaluation Kit

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

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

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



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

Modify the Java Application

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

For example, we could modify the color of the background that is shown behind the MicroEJ mascot.

- Open `AnimatedMascot.java` file located in the `src/main/java/com/microej/demo/animatedMascot` folder.
- Background color is set line 84, replace the following line:

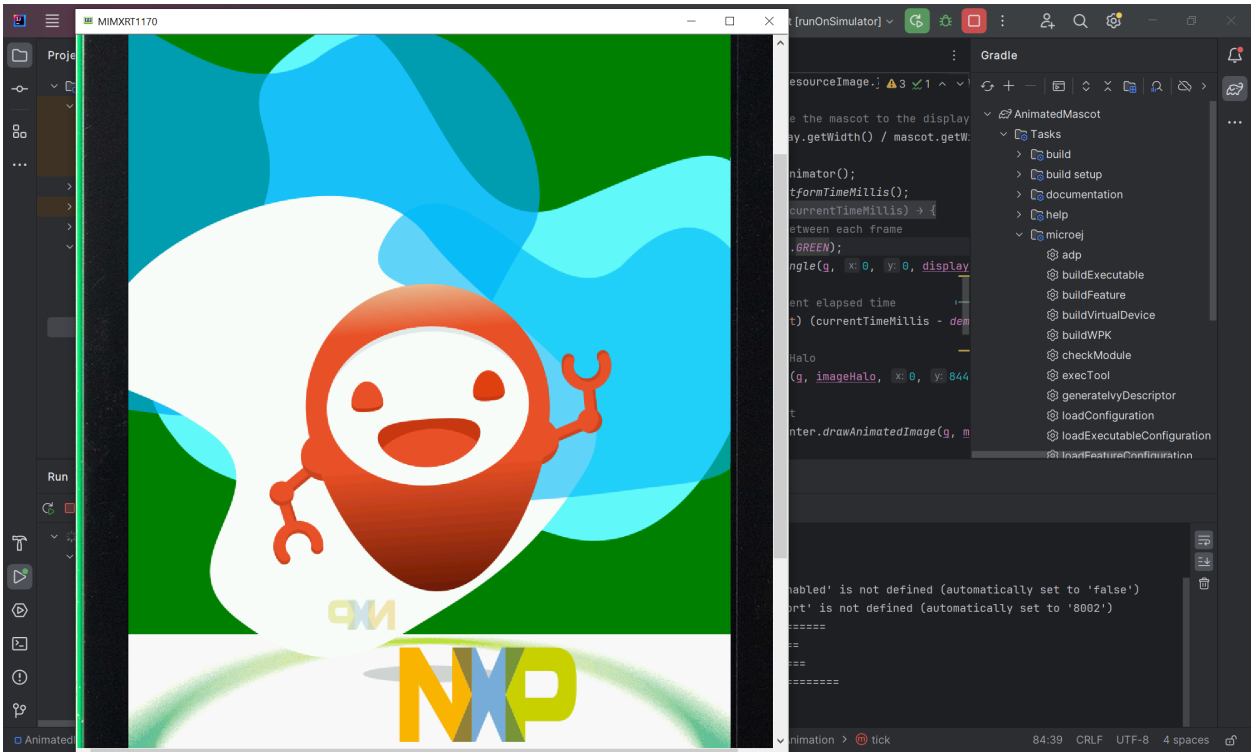
```
g.setColor(Colors.WHITE);
```

by

```
g.setColor(Colors.GREEN);
```

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

Here is the modified application running in simulation:



4.2 Installation

This chapter will guide you through the installation process of the SDK on your workstation. First check the System Requirements before proceeding.

4.2.1 System Requirements

- **Hardware**
  - Intel x64 (Dual-core i5 minimum) or macOS AArch64 (M1) processor
  - 4GB RAM (minimum)
  - 16GB Disk (minimum)
- **Operating Systems**
  - Windows 11 or Windows 10
  - Linux distributions (tested on Ubuntu 20.04 and 22.04)

  - macOS x86_64 with Intel chip
  - macOS aarch64 with M1 chip

- **Java Runtime Environment**
  - JDK 11 or 17 - Eclipse Temurin or Oracle Distributions

4.2.2 Check your JDK version

The SDK requires a JDK 11 or a higher LTS version to be installed and:

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

OR

- The `java` executable of a JDK available in the `PATH`.

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

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

```
$ java -version
openjdk version "11.0.15" 2022-04-19
OpenJDK Runtime Environment Temurin-11.0.15+10 (build 11.0.15+10)
OpenJDK 64-Bit Server VM Temurin-11.0.15+10 (build 11.0.15+10, mixed mode)
```

If you don't have a JDK installed, you can download and install one from Adoptium or Oracle.

4.2.3 Install Gradle

Once a JDK is correctly configured, the next step is to install Gradle by following the official documentation. The SDK is only compatible with the Gradle `8.0.2` and higher, so make sure to install a right version. Once done, you can verify your installation by opening a terminal and run the command `gradle -v`. It should display, amongst other information, the Gradle and the JVM versions:

```
 $ gradle -v

------------------------------------------------------------
Gradle 8.0.2
------------------------------------------------------------

Build time:   2023-03-03 16:41:37 UTC
Revision:     7d6581558e226a580d91d399f7dfb9e3095c2b1d

Kotlin:       1.8.10
Groovy:       3.0.13
Ant:          Apache Ant(TM) version 1.10.11 compiled on July 10 2021
JVM:          11.0.18 (Eclipse Adoptium 11.0.18+10)
OS:           Windows 10 10.0 amd64
```

4.2.4 Configure Repositories

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

- Create the folder `<user.home>/.gradle/init.d` if they do not exist.
- download and copy `this file` in the previously created folder.

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

4.2.5 Install the IDE

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

- Android Studio - Minimum supported version is `Hedgehog - 2023.1.1`.
- IntelliJ IDEA (Community or Ultimate edition) - Minimum supported version is `2021.2`.
- Eclipse IDE for Java Developers - Minimum supported version is `2022-03`.

Follow their respective documentation to install one of them.

These 3 IDEs come with the Gradle plugin installed by default.

4.2.6 Install the IDE Plugin

Once your favorite IDE is installed, the MicroEJ plugin must be installed.

Android Studio

IntelliJ IDEA

Eclipse

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

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

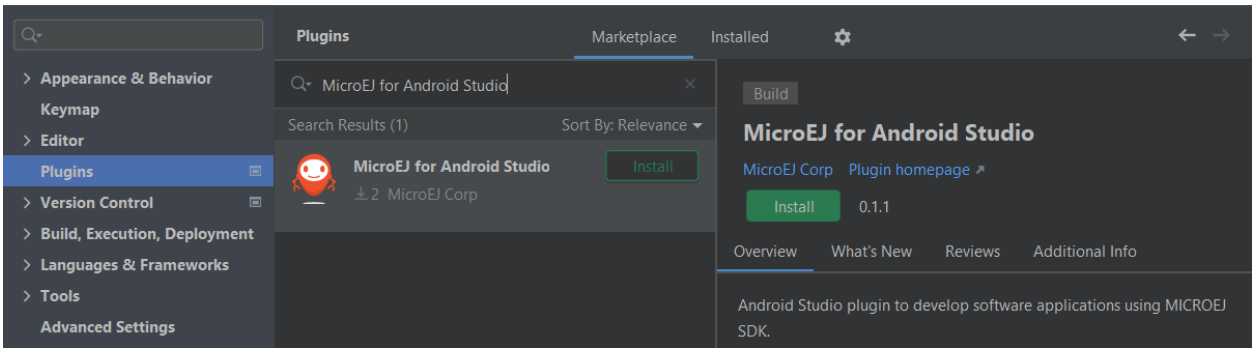


Fig. 3: Android Studio Plugin Installation

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

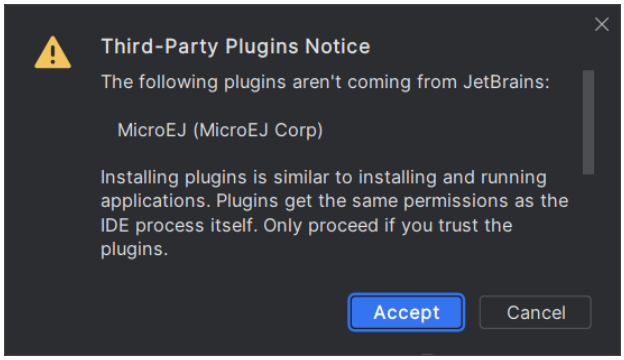


Fig. 4: Android Studio Plugin Installation - Third-Party Plugins Notice

- Click on the `Restart IDE` button.

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

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

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

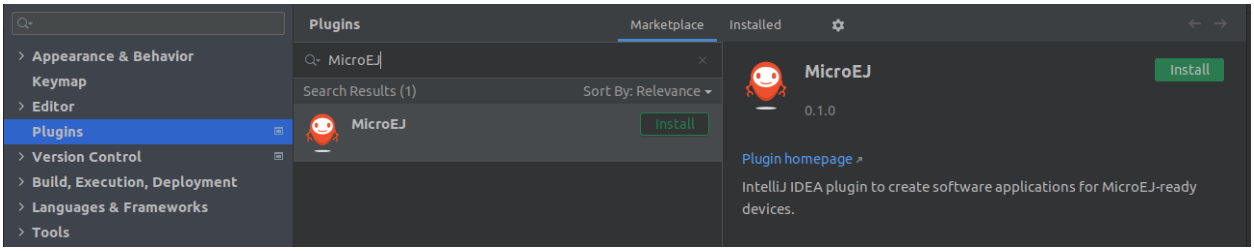


Fig. 5: IntelliJ IDEA Plugin Installation

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

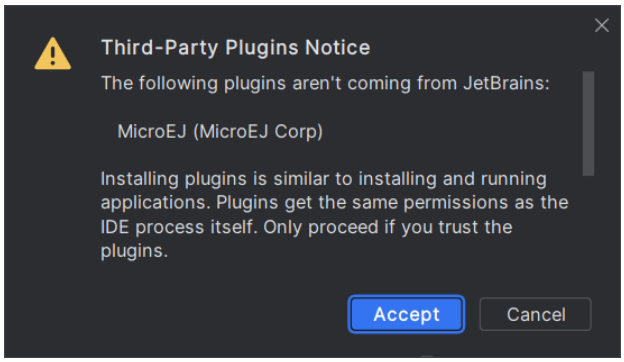


Fig. 6: IntelliJ IDEA Plugin Installation - Third-Party Plugins Notice

- Click on the `Restart IDE` button.

To install the snapshot version of the MicroEJ plugin, please refer to *How to Install MicroEJ Plugin Snapshot Version on Android Studio or IntelliJ IDEA*.

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

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

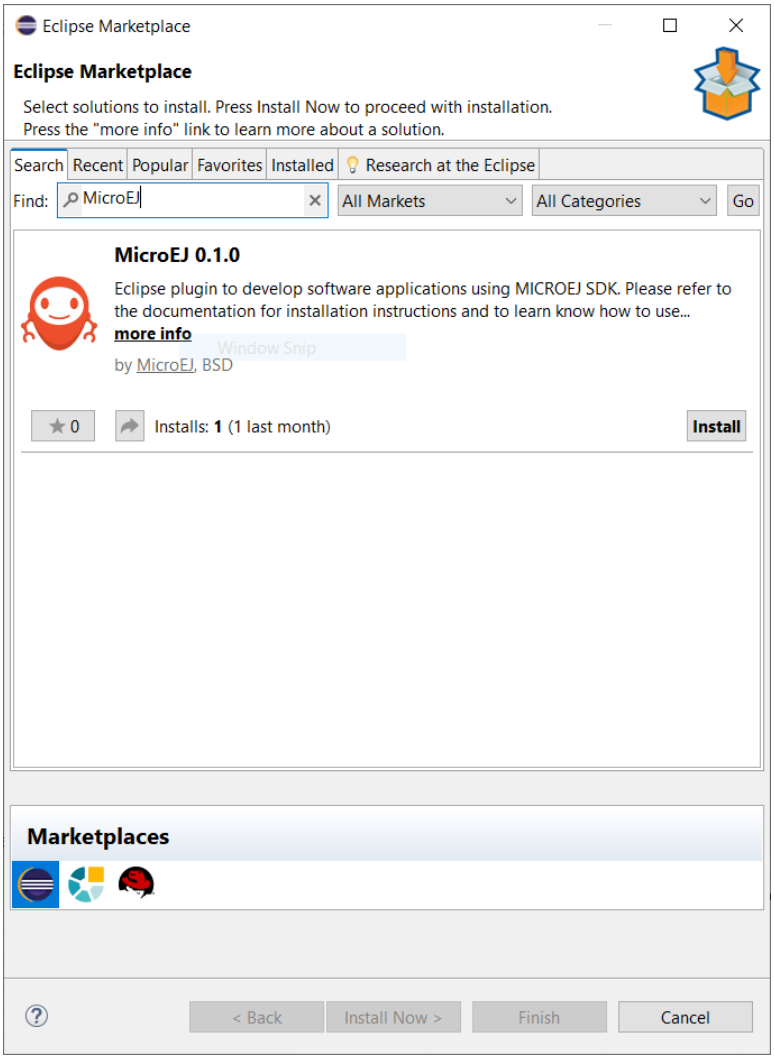


Fig. 7: Eclipse Plugin Installation - Marketplace

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

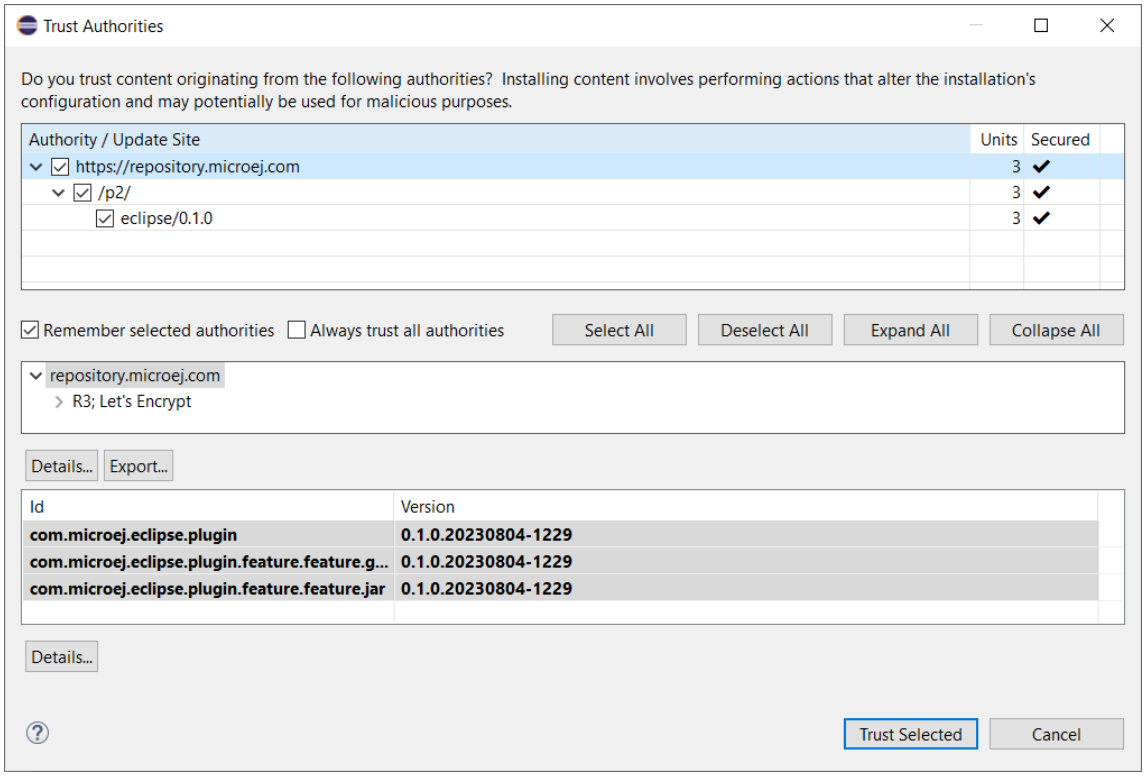


Fig. 8: Eclipse Plugin Installation - Trust Authorities

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

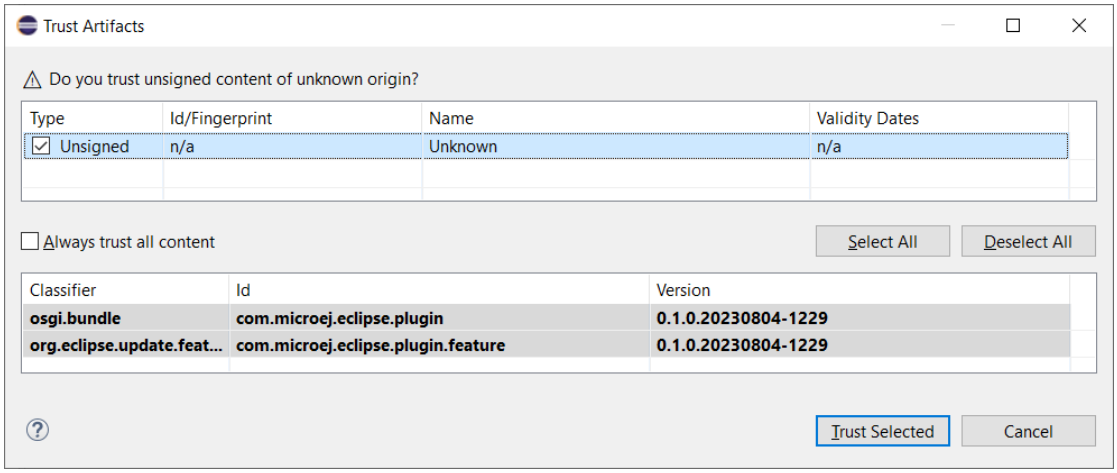


Fig. 9: Eclipse Plugin Installation - Trust Artifacts

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

4.3 Licenses

4.3.1 SDK EULA

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

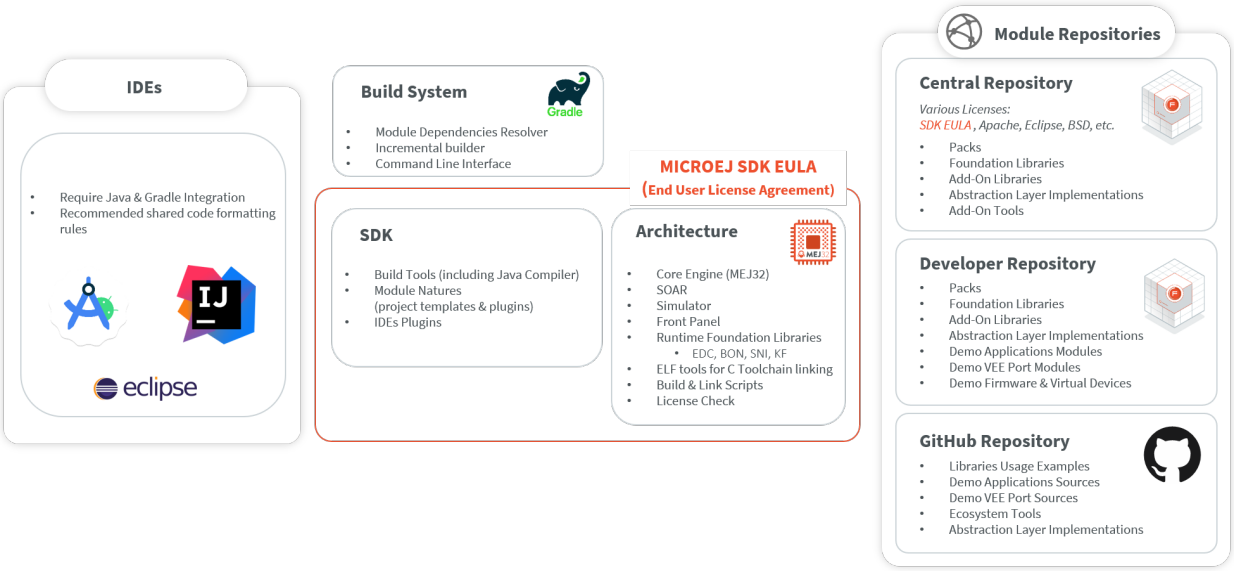


Fig. 10: SDK Detailed View

4.3.2 License Manager Overview

Architectures are distributed in two different versions:

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

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

4.3.3 License Check

The table below summarizes where the license is checked.

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

4.3.4 SDK EULA Acceptation

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

```
> The MICROEJ SDK End-User License Agreement (EULA) must be accepted before it can start.
  The license terms for this product can be downloaded from
  https://repository.microej.com/licenses/sdk/LAW-0011-LCS-MicroEJ_SDK-EULA-v3.1B.txt
  You can accept the EULA by specifying the -Daccept-microej-sdk-eula-v3-1b=YES command line␣
↪option,
  or setting the system property systemProp.accept-microej-sdk-eula-v3-1b=YES in a gradle.
↪properties file,
  or setting the ACCEPT_MICROEJ_SDK_EULA_V3_1B=YES environment variable.
```

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

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

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

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

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

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

- set the `ACCEPT_MICROEJ_SDK_EULA_V3_1B` environment variable to `YES`.

4.3.5 Evaluation Licenses

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

Get your Machine UID

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

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

Request your Activation Key

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

Install the License Key

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

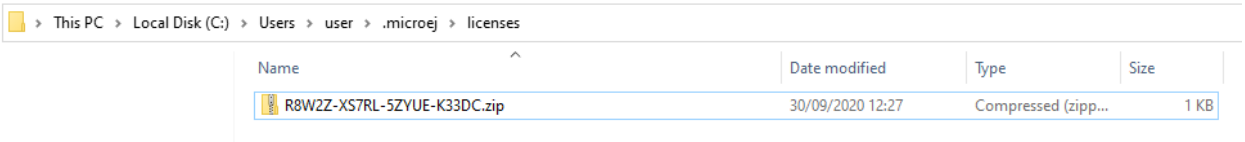


Fig. 11: MicroEJ Shared Licenses Directory

Troubleshooting

Machine UID has changed

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

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

4.3.6 Production Licenses

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



Fig. 12: MicroEJ USB Dongle

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

Request your Activation Key

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

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

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

Activate your USB Dongle

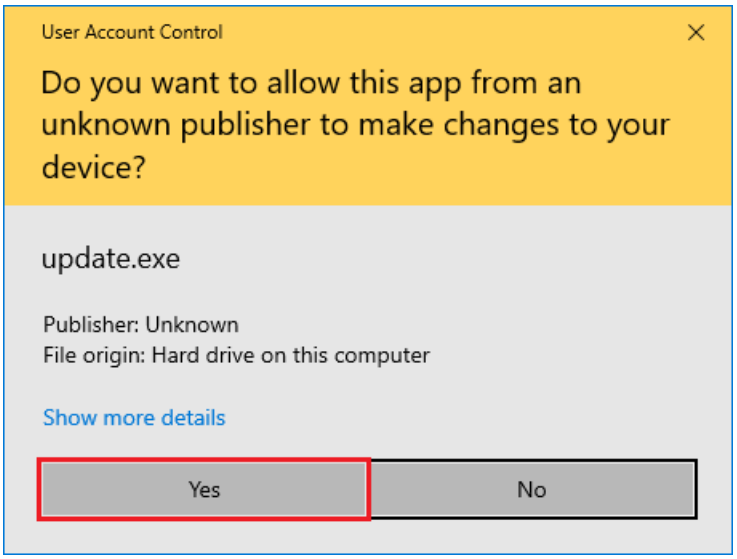
This section contains instructions that will allow you to flash your USB dongle with the proper activation key.

You shall ensure that the following prerequisites are met :

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

You can then proceed to the USB dongle update:

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



- Click on the `Update` button (no password needed)

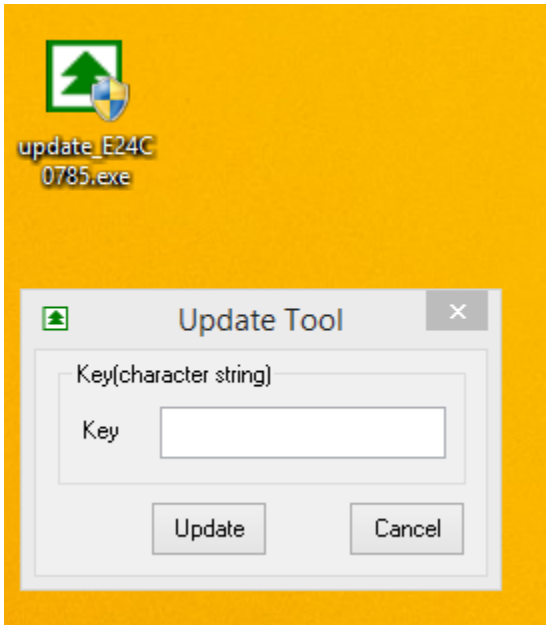


Fig. 13: Dongle Update Tool

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

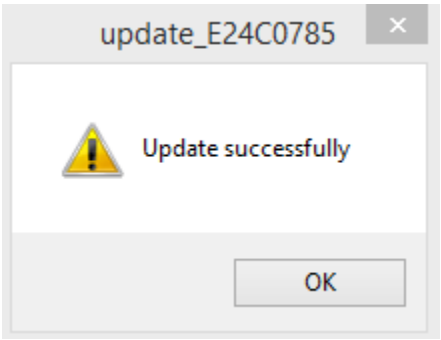


Fig. 14: Successful Dongle Update

Check Activation

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

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

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

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

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

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

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

USB Dongle on GNU/Linux

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

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

    LABEL="usbdongle_start"

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

    LABEL="usbdongle_end"
```

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

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

USB Dongle with Docker on Linux

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

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

    LABEL="usbdongle_start"

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

    LABEL="usbdongle_end"
```

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

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

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

USB Dongle with WSL

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

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

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

Then, you need install usbipd-win on Windows from usbipd-win Github repository.

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

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

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

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

You then need to unplug and plug your dongle again before attaching the dongle to WSL from powershell:

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

The `<BUSID>` can be obtainted with the following powershell command:

```
usbipd wsl list
```

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

Troubleshooting

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

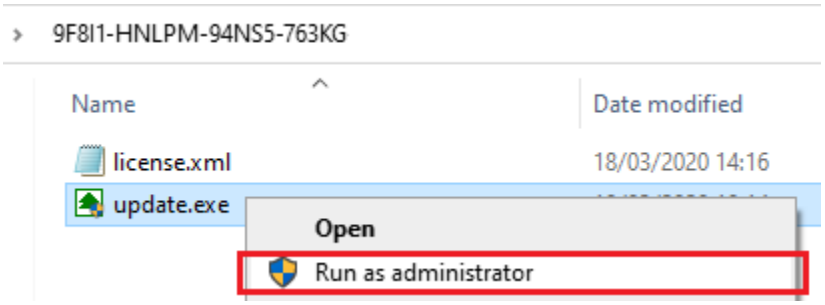
Windows Troubleshooting

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

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

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

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



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

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

VirtualBox Troubleshooting

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

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

WSL Troubleshooting

Check that your dongle is attached to WSL from Powershell:

```
usbipd wsl list
```

You should have a line saying `Attached - Ubuntu` :

```
PS C:\Users\sdkuser> usbipd.exe wsl list
BUSID  VID:PID    DEVICE                                                  ␣
↪STATE
2-1    096e:0006  USB Input Device                                         ␣
↪Attached - Ubuntu
2-6    0c45:6a10  Integrated Webcam                                       ␣
```

(continues on next page)

(continued from previous page)

```
↪Not attached
2-10   8087:0026  Intel(R) Wireless Bluetooth(R)                              ␣
↪Not attached
3-1    045e:0823  USB Input Device                                            ␣
↪Not attached
3-4    046d:c31c  USB Input Device                                            ␣
↪Not attached
```

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

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

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

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

Remote USB Dongle Connection

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

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

4.4 Scope and Limitations

The SDK 6 allows to:

- Build the Java artifact of an Application and an Add-On Library.
- Execute the tests of a project with the Simulator and on a device.
- Execute the Artifact Checker on a project.
- Run an Application with the Simulator.
- Load the VEE Port from its archive file path, its folder path or a dependency.
- Build the Executable of an Application.
- Build the WPK of an Application.
- Build the Feature file (.fo) of an Application.
- Build the Virtual Device of an Application.
- Use the Stack Trace Reader.
- Use the Code Coverage Analyzer.

- Use the Font Designer, Memory Map Analyzer, Heap Analyzer and Front Panel Designer tools.

Therefore, it does not support all the features of the SDK 5, especially:

- Build of Foundation Libraries, VEE Ports or any other component type except Applications, Add-On Libraries and Mocks.
- Launch of some MicroEJ tools, such as the Local Deploy, the Serial to Socket Transmitter or the Kernel Metadata Generator.

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

4.5 Create a Project

This chapter explains the different ways to create a new project.

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

Android Studio

IntelliJ IDEA

Eclipse

Command Line Interface

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

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

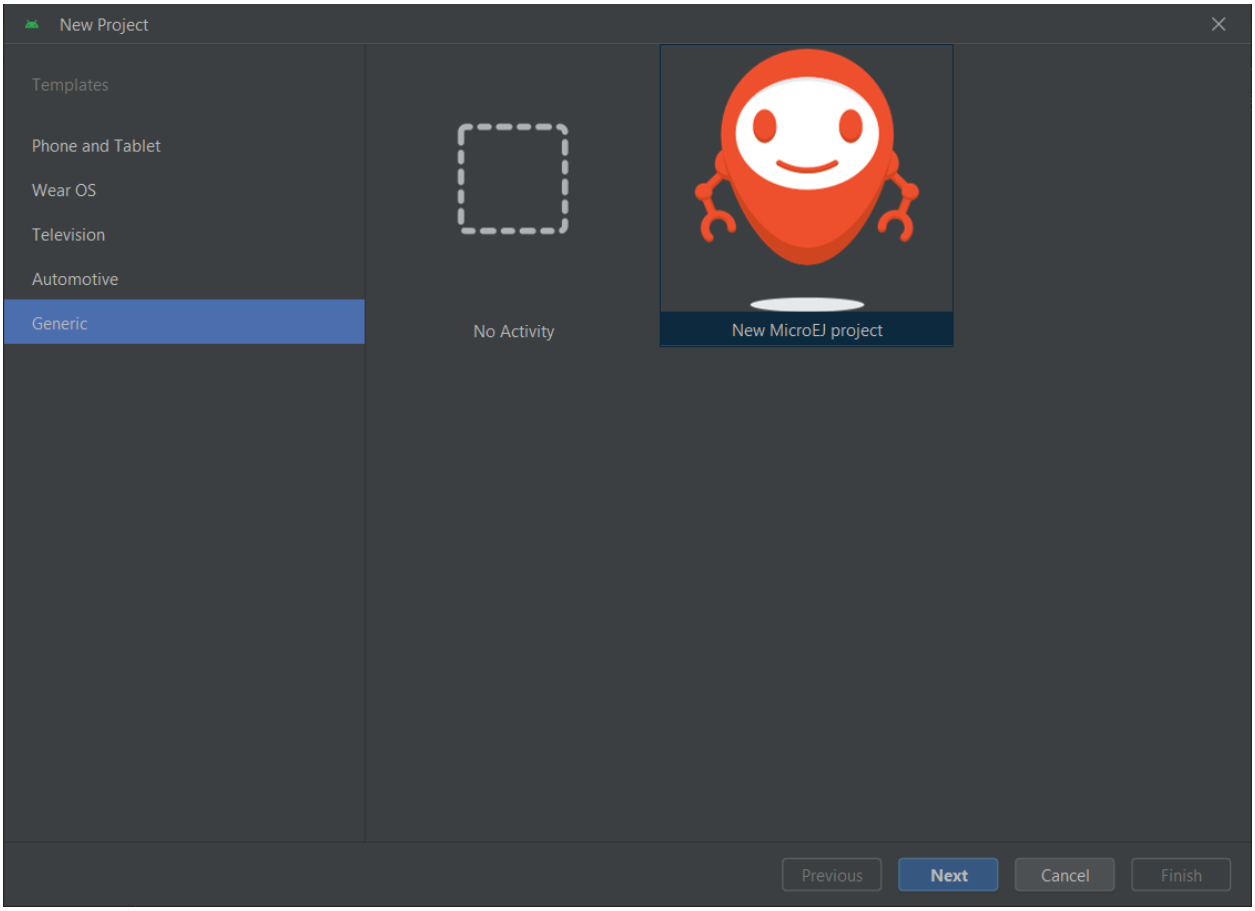


Fig. 15: Project Creation in Android Studio

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

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

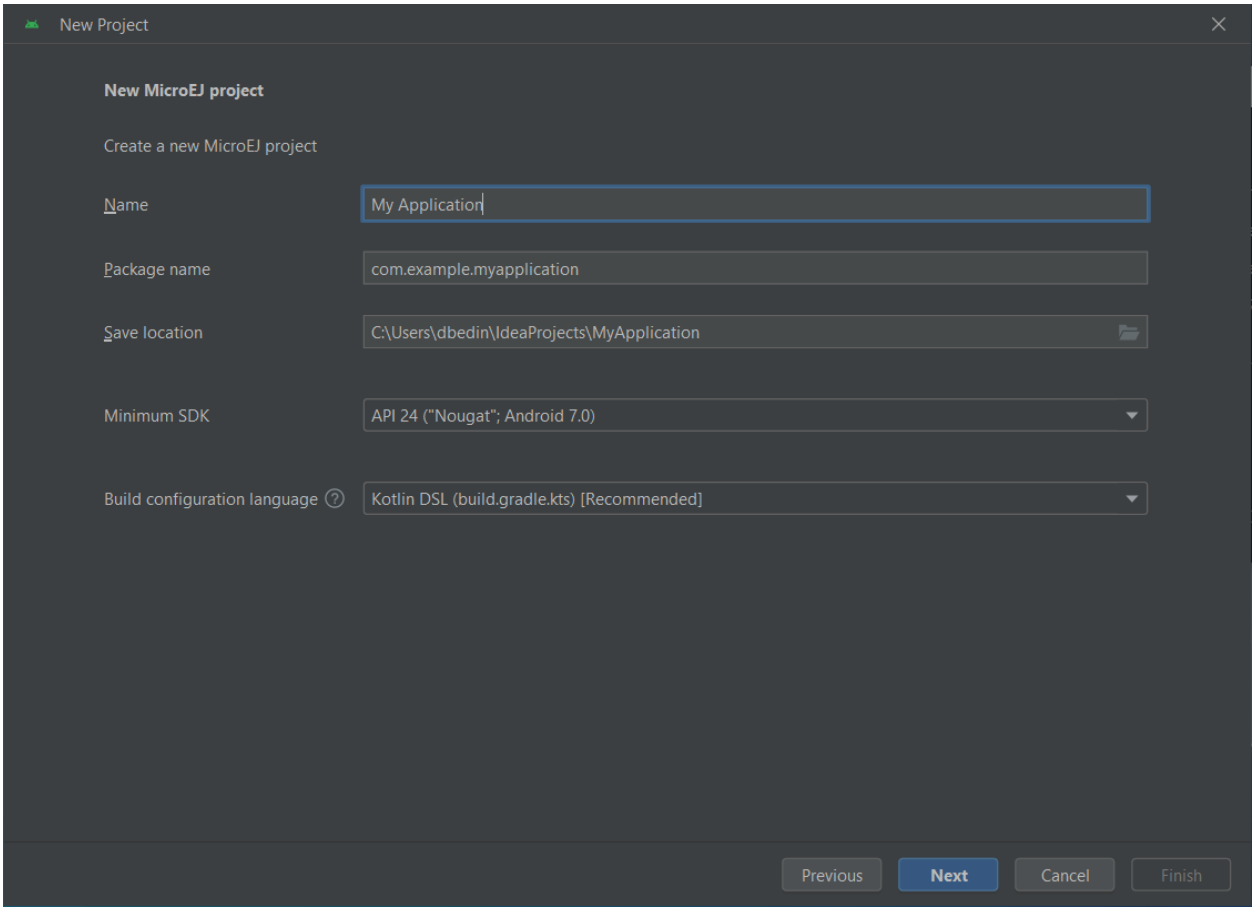


Fig. 16: Project Creation in Android Studio

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

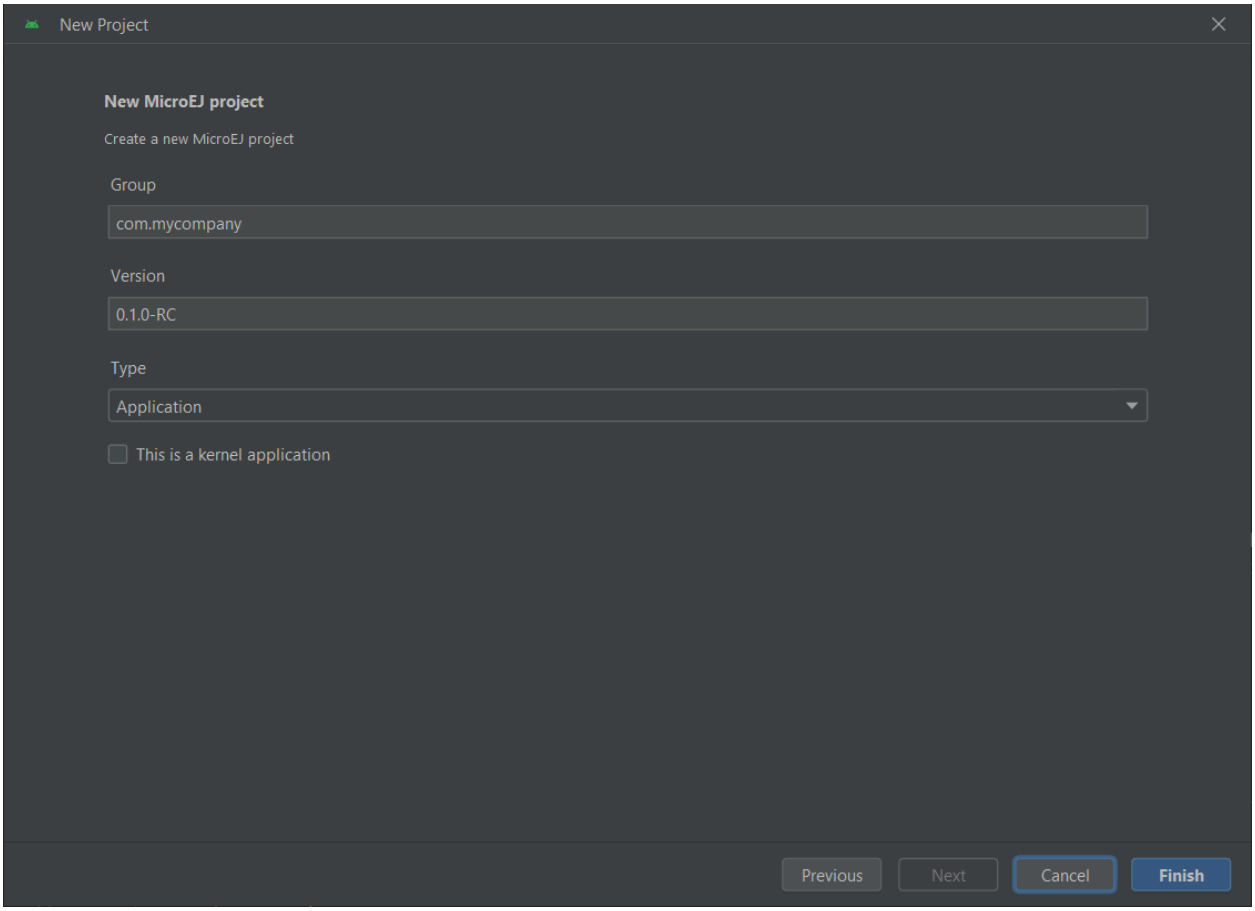


Fig. 17: Project Creation in Android Studio

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

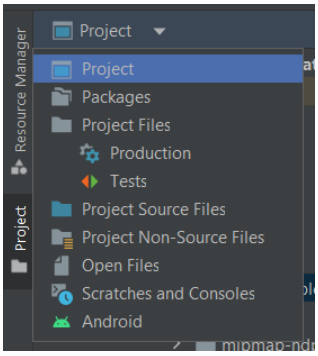


Fig. 18: Project View in Android Studio

Note: The newly created Gradle project uses Gradle Wrapper with Gradle version `8.2`. Refer to the *Gradle Wrapper* section for more information.

The project created by the wizard is a multi-project with a single subproject (named `app`). This subproject is either

an Application or an Add-On Library, depending on the module type that has been chosen.

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

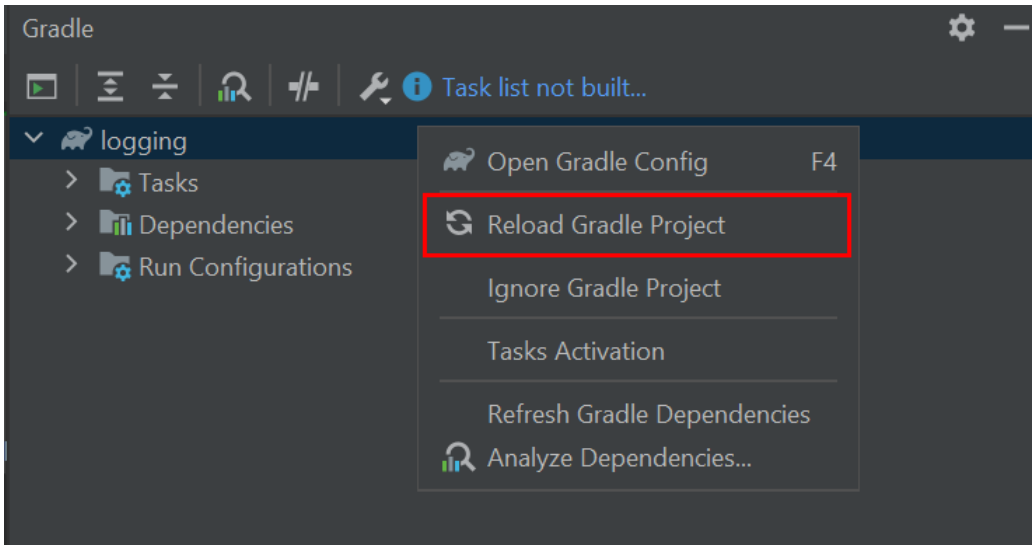


Fig. 19: Gradle Project reload in Android Studio

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

Warning: When reloading your Gradle project, the build can fail if the SDK EULA has not been accepted. In that case, you must set the `ACCEPT_MICROEJ_SDK_EULA_V3_1B` environment variable to `YES` and restart Android Studio. For more information about SDK EULA, refer to the *Licenses* chapter.

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

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

- Click on `File` > `New` > `Project...` .
- Select `MicroEJ` in `Generators` list on the left panel.
- Fill the name of the project in the `Name` field.
- Select the location of the project in the `Location` field.
- Select the module type among `Application` , `Mock` and `Addon-Library` buttons.
- If you selected `Application` module type, you can check `This is a kernel application` checkbox if your Application is a Kernel.
- Fill the version of the artifact to publish in the `Version` field.

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

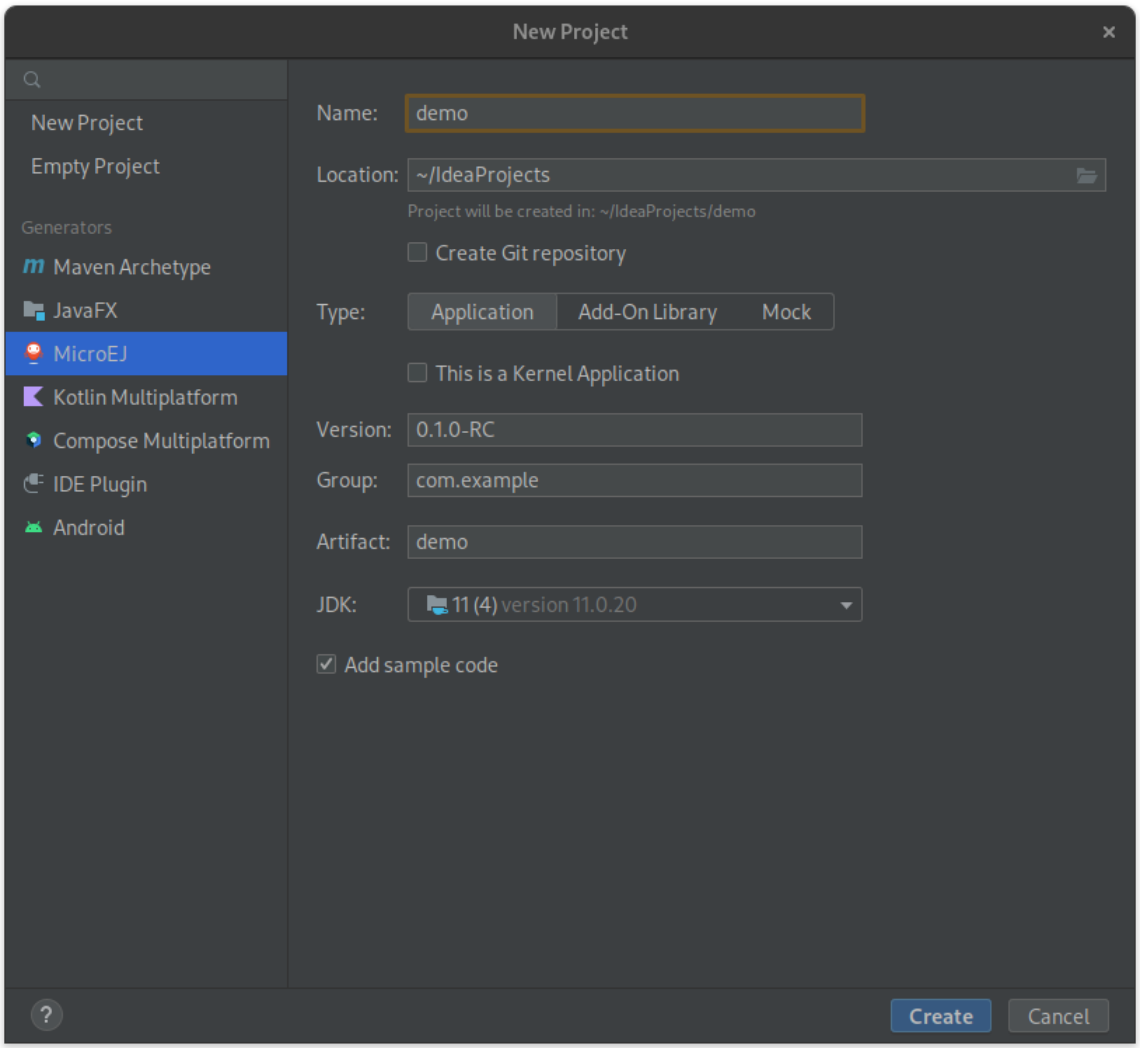


Fig. 20: Project Creation in IntelliJ IDEA

Note: The Gradle project created by the wizard uses Gradle Wrapper with Gradle version `8.5`. Refer to the *Gradle Wrapper* section for more information.

Note: By default, IntelliJ IDEA automatically saves any file change, but requires the user to explicitly trigger the reload of a Gradle project when its configuration has changed. Therefore, when the configuration of a Gradle project

has been updated, you have to click on the reload icon button which appears on the right of the editor:

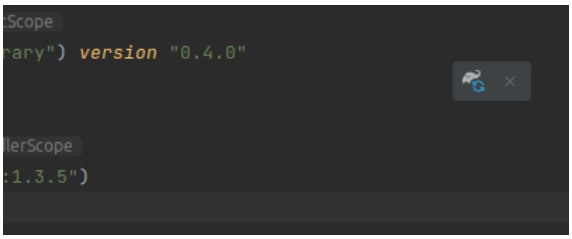


Fig. 21: Gradle Project reload in IntelliJ IDEA

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

Warning: When reloading your Gradle project, the build can fail if the SDK EULA has not been accepted. In that case, you must set the `ACCEPT_MICROEJ_SDK_EULA_V3_1B` environment variable to `YES` and restart IntelliJ IDEA. For more information about SDK EULA, refer to the *Licenses* chapter.

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

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

- Click on `File` > `New` > `Project...` .
- Select the project type `MicroEJ` > `MicroEJ Application Project` , `MicroEJ Mock` or `MicroEJ Add-onLibrary Project` and click on the `Next` button.

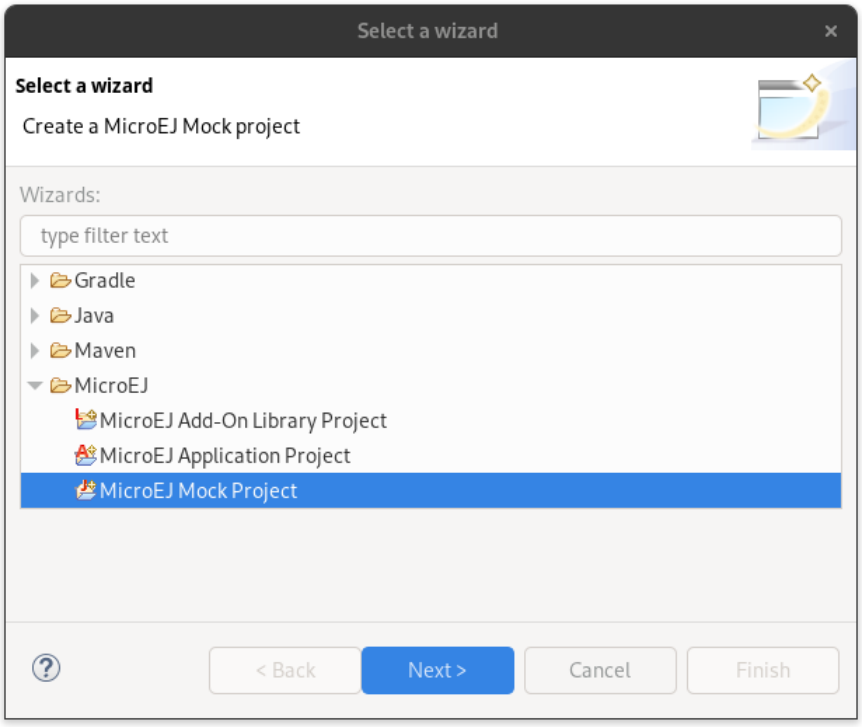


Fig. 22: Project Type Selection in Eclipse

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

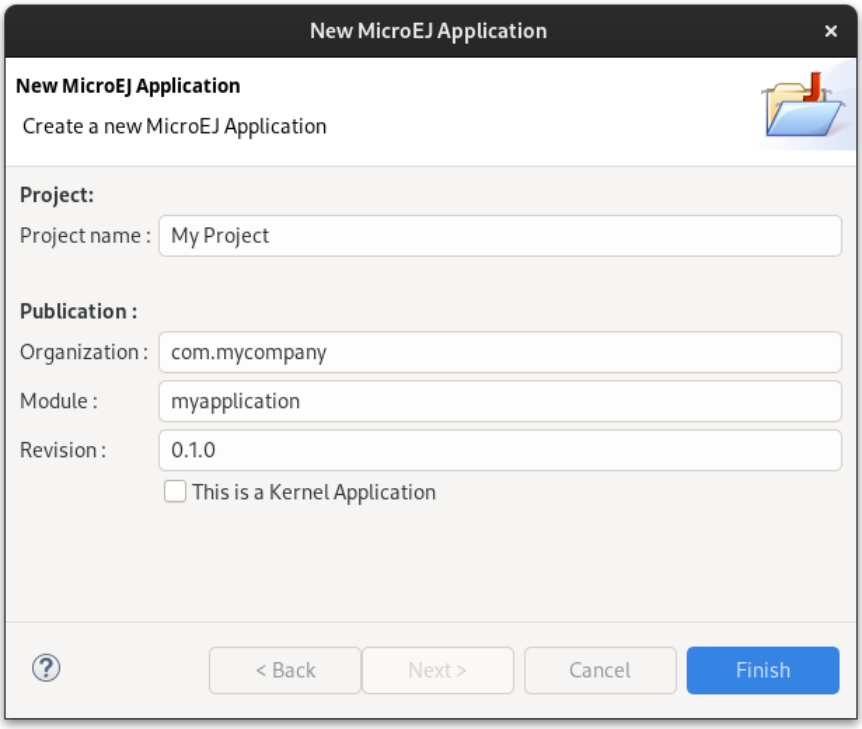


Fig. 23: Application Creation in Eclipse

Note: The Gradle project created by the wizard uses Gradle Wrapper with Gradle version `8.5`. Refer to the *Gradle Wrapper* section for more information.

Warning: When reloading your Gradle project, the build can fail if the SDK EULA has not been accepted. In that case, you must set the `ACCEPT_MICROEJ_SDK_EULA_V3_1B` environment variable to `YES` and restart Eclipse. For more information about SDK EULA, refer to the *Licenses* chapter.

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

The creation of a project can be done via the command line interface via the Gradle `init` task. This task guides you through multiple steps to configure and select the project template to use. Refer to the official documentation for the full list of templates and options.

In order to create a MicroEJ project, the best way is to use the `application` template:

- In a new empty directory, execute the command `gradle init`.
- Select the `application` project type.
- Select the `Java` implementation language.
- For the step `Generate multiple subprojects for application?`, select `no`.
- Select build script DSL `Kotlin`.

Note: The SDK uses Kotlin as the default Gradle build script DSL. The use of the Groovy build script DSL is still possible but not officially supported.

- For the test framework, select `JUnit 4`.
- Choose the name of the project (defaults to the name of the parent directory).
- Choose the package name for the source files.
- For the target version of Java, select `7`.
- Decide if you want to use Gradle new APIs and behavior in your build script. If you are new to Gradle, choose `no`.

Note: These steps are the ones proposed when creating a project with Gradle `8.2.1`. Depending on the Gradle version used, the steps to create a project can be slightly different.

The created project is a multi-project build containing a root project and a single subproject (named `app`). The `app` subproject is a standard Java Application project (Gradle `java` plugin), so it must be updated to be a MicroEJ project:

- Open the project in your favorite editor.
- Open the `app/build.gradle.kts` file.
- Erase its whole content.
- *Configure the project* depending on the module nature you want to build.
- Declare the dependencies required by your project in the `dependencies` block. For example:

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

- Delete the test class in the folder `app/src/test/java`.

4.5.1 Configure a Project

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

- *Application*
- *Add-On Library*
- *Mock*
- *J2SE Library*

Application Project

- Add the `com.microej.gradle.application` plugin in the `build.gradle.kts` file:

```
plugins {
    id("com.microej.gradle.application") version "0.15.0"
}
```

Note: The `java` plugin must not be added since it is automatically applied by the MicroEJ plugin.

- If your Application is a Standalone Application:
  - Create the Java main class in the `src/main/java` folder.
  - Define the property `applicationMainClass` in the `microej` configuration block of the `build.gradle.kts` file. It must be set to the Full Qualified Name of the Application main class, for example:

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

- If your Application is a Kernel Application:
  - Create the Java main class in the `src/main/java` folder.
  - Define the property `applicationMainClass` in the `microej` configuration block of the `build.gradle.kts` file. It must be set to the Full Qualified Name of the Application main class, for example:

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

  - Create a file named `kernel.kf` in the `src/main/resources` folder. This file is a property file which must contain at least the `version` property, and optionally the `name` property (defaults to `KERNEL`), for example:

```
version=1.0.0
name=MY-KERNEL
```

- If your Application is a *Sandboxed Application*:
  - Create the Java class of the Feature Entry Point in the `src/main/java` folder, for example:

```
package com.mycompany;

import ej.kf.FeatureEntryPoint;

public class MyApplication implements FeatureEntryPoint {

  @Override
  public void start() {
    System.out.println("Feature MyApplication started!");
  }

  @Override
```

(continues on next page)

(continued from previous page)

```
  public void stop() {
    System.out.println("Feature MyApplication stopped!");
  }
}
```

- Create a file with the extension `.kf` in the `src/main/resources` folder, for example `MyApplication.kf`. This file must at least contain the property `entryPoint` set to the Full Qualified Name of the Application Feature class, for example:

```
entryPoint=com.mycompany.MyApplication
version=0.1.0
```

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

Add-On Library Project

- Add the `com.microej.gradle.addon-library` plugin in the build script:

```
plugins {
    id("com.microej.gradle.addon-library") version "0.15.0"
}
```

Note: The `java` plugin must not be added since it is automatically applied by the MicroEJ plugin.

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

Mock

- Add the `com.microej.gradle.mock` plugin in the build script:

```
plugins {
    id("com.microej.gradle.mock") version "0.15.0"
}
```

Note: The `java` plugin must not be added since it is automatically applied by the MicroEJ plugin.

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

J2SE Library Project

- Add the `com.microej.gradle.j2se-library` plugin in the build script:

```
plugins {
    id("com.microej.gradle.j2se-library") version "0.15.0"
}
```

Note: The `java` plugin must not be added since it is automatically applied by the MicroEJ plugin.

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

4.5.2 Create a subproject in an existing project

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

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

Android Studio

IntelliJ IDEA

Eclipse

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

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

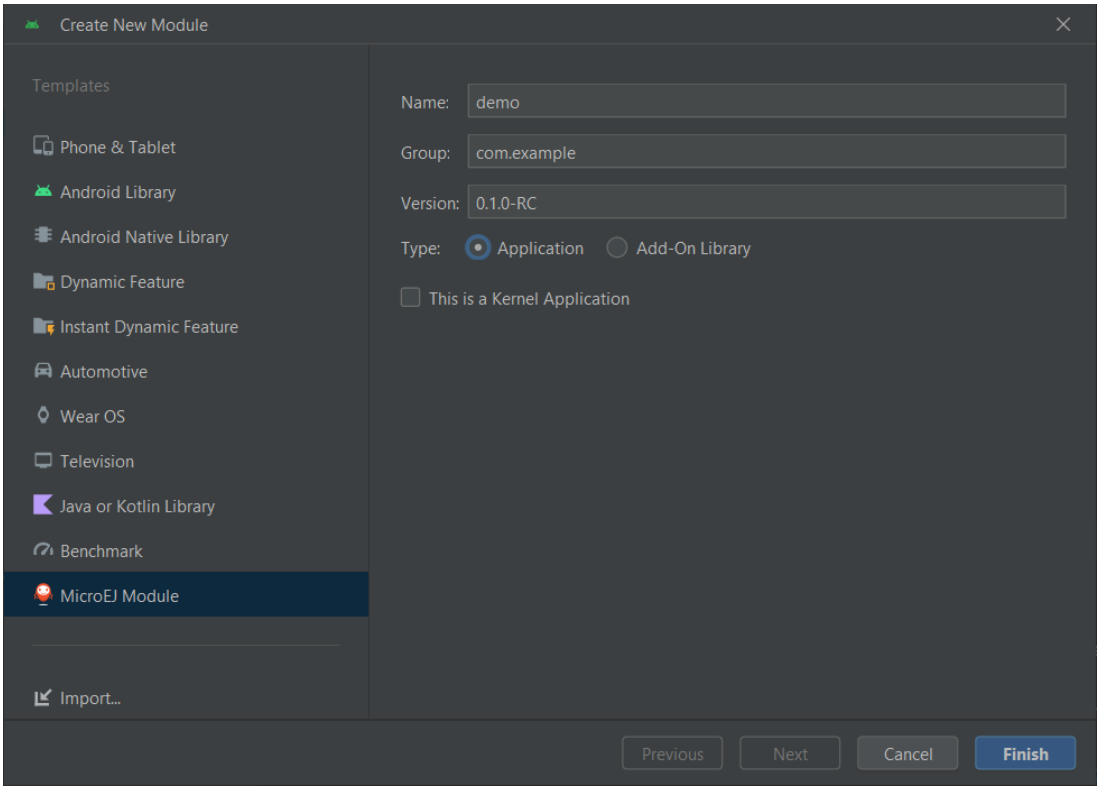


Fig. 24: Module Creation in Android Studio

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

- Click on File > New > Module... .
- Select MicroEJ in Generators list on the left panel.
- Fill the name of the module in the Name field.
- Select the location of the module in the Location field.
- Select the module type among Application and Addon-Library buttons.
- If you selected Application module type, you can check This is a kernel application checkbox if your Application is a Kernel.
- Fill the version of the artifact to publish in the Version field.
- Fill the group of the artifact to publish in the Group field.
- Fill the name of the artifact to publish in the Artifact field.
- Select the JVM used by Gradle in the JDK combobox.
- Check the Add sample code checkbox.
- Click on Create button.

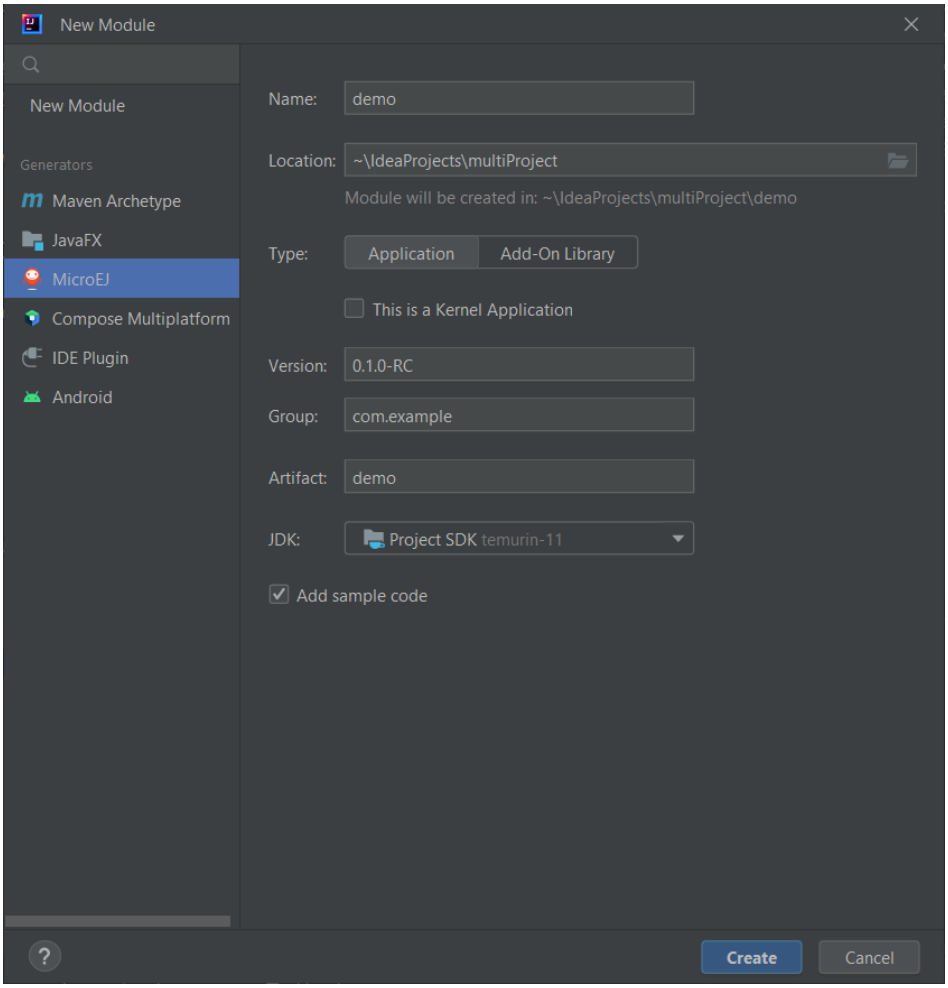


Fig. 25: Module Creation in IntelliJ IDEA

- Include the module to your project by adding the following line to the `settings.gradle.kts` file of the project:

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

- Right-click on the module name in the Gradle tasks view and click on `Unlink Gradle Project` .
- Reload of a Gradle project by clicking on the reload icon button which appears on the right of the editor:

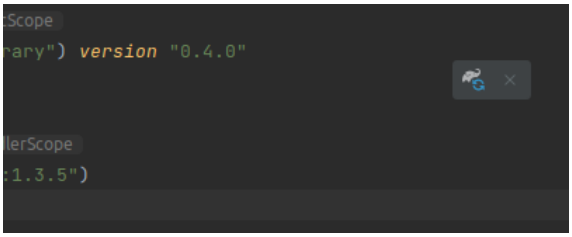


Fig. 26: Gradle Project reload in IntelliJ IDEA

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

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

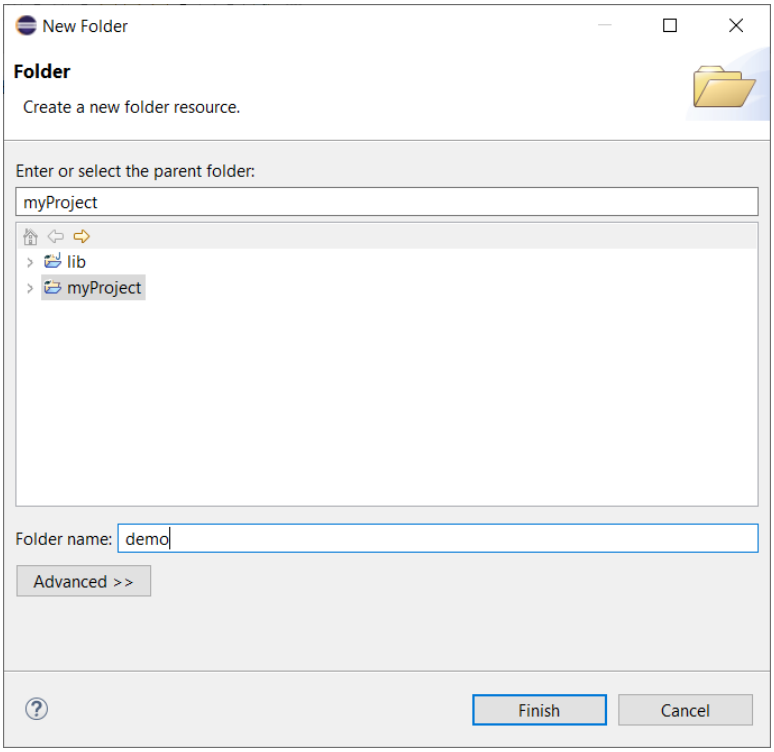


Fig. 27: Module Creation in Eclipse

- Right-click on your newly created folder and click on `New` > `File` .
- Enter `build.gradle.kts` in the `File name` field.
- Click on `Finish` button and open the `build.gradle.kts` file.
- Add the MicroEJ plugin, depending on the module nature you want to build, for example for an Add-On Library:

```
plugins {
    id("com.microej.gradle.addon-library") version "0.15.0"
}
```

or for an Application:

```
plugins {
    id("com.microej.gradle.application") version "0.15.0"
}
```

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

- Declare the dependencies required by your project in the `dependencies` block. For example:

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

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

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

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

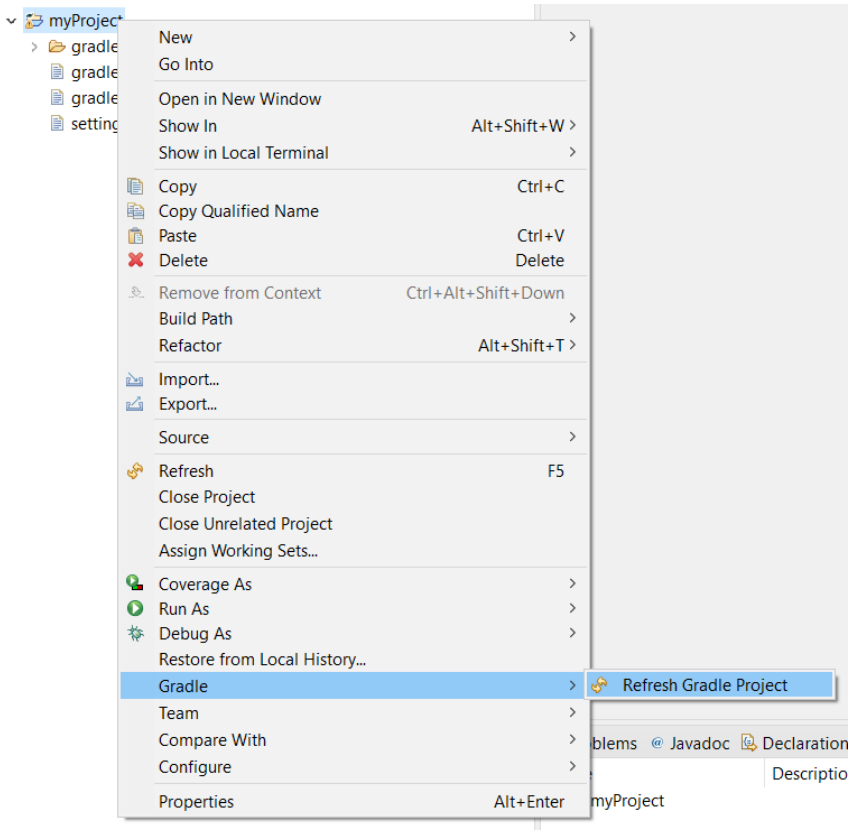


Fig. 28: Gradle Project reload in Eclipse

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

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

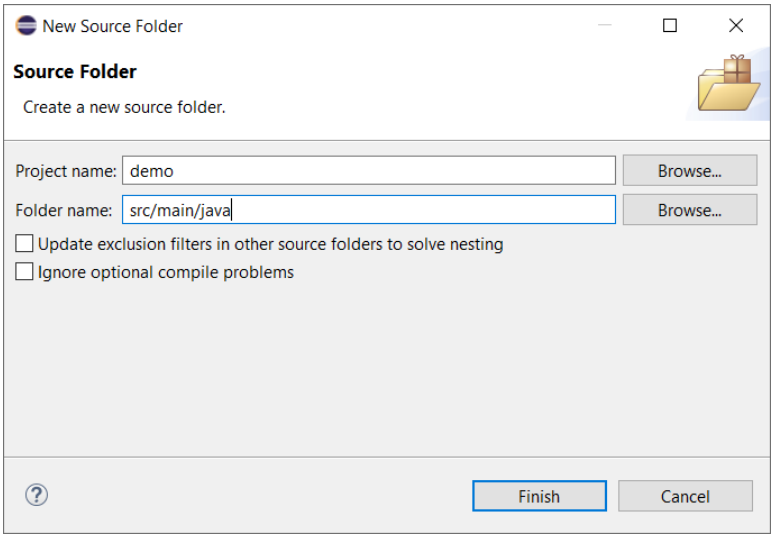


Fig. 29: Source Folder Creation in Eclipse

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

4.5.3 Gradle Wrapper

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

When creating a project following one of the project creation systems described in the *Create a Project* section, the Wrapper files are automatically generated in the `gradle/wrapper` folder of the project. It is also possible to add the Wrapper to an existing project by executing the `wrapper` task:

```
gradle wrapper
```

The Gradle version used by the project can then be updated in the `gradle/wrapper/gradle-wrapper.properties` file. The SDK requires Gradle `8.0.2` or higher:

```
distributionUrl=https\://services.gradle.org/distributions/gradle-8.0.2-bin.zip
```

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

Windows

Linux

gradlew build

./gradlew build

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

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

4.6 Import a Project

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

Android Studio

IntelliJ IDEA

Eclipse

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

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

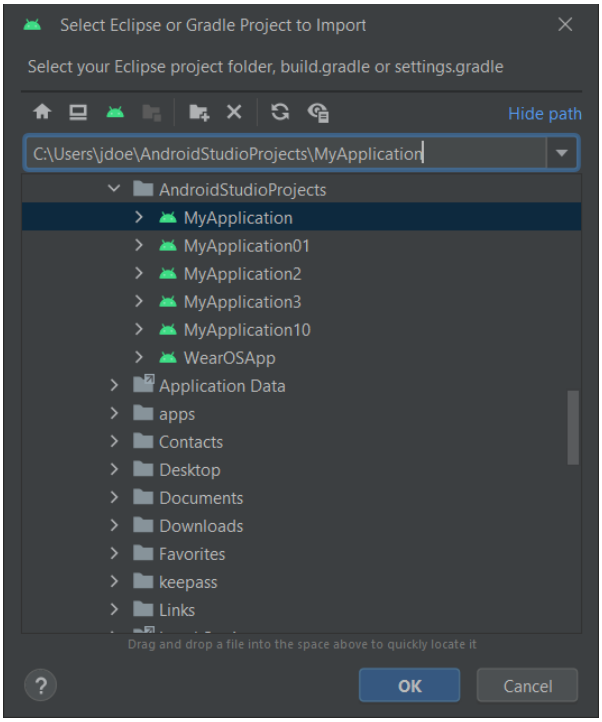


Fig. 30: Project Import in Android Studio

The Gradle project should now be opened in Android Studio.

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

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

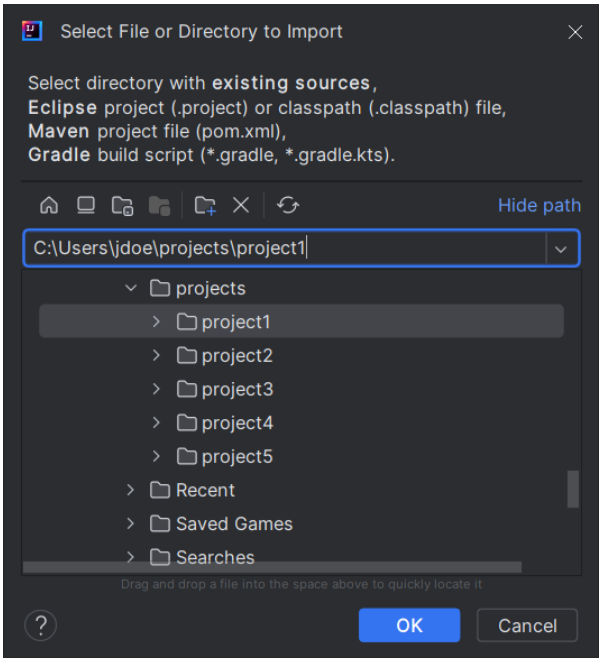


Fig. 31: Project Import in IntelliJ IDEA

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

The Gradle project should now be opened in IntelliJ IDEA.

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

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

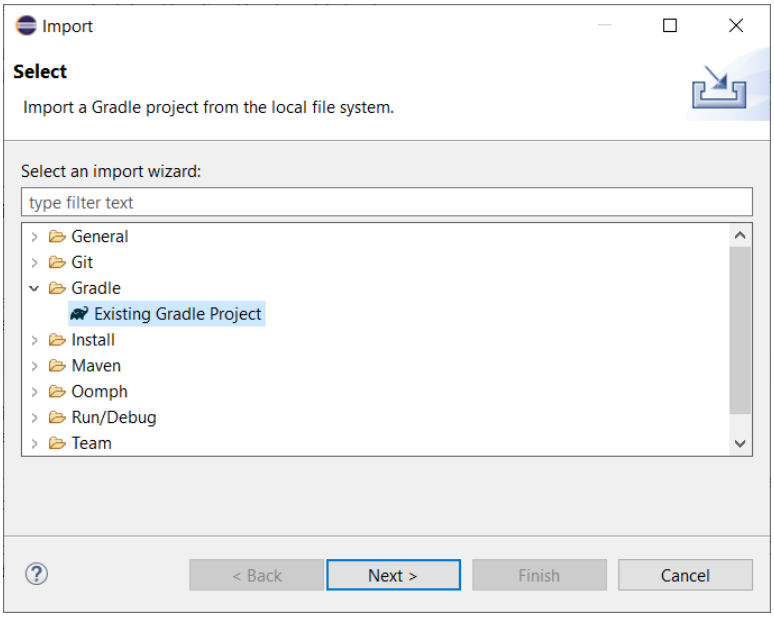


Fig. 32: Project Type Selection in Eclipse

- Select the root directory of the project.

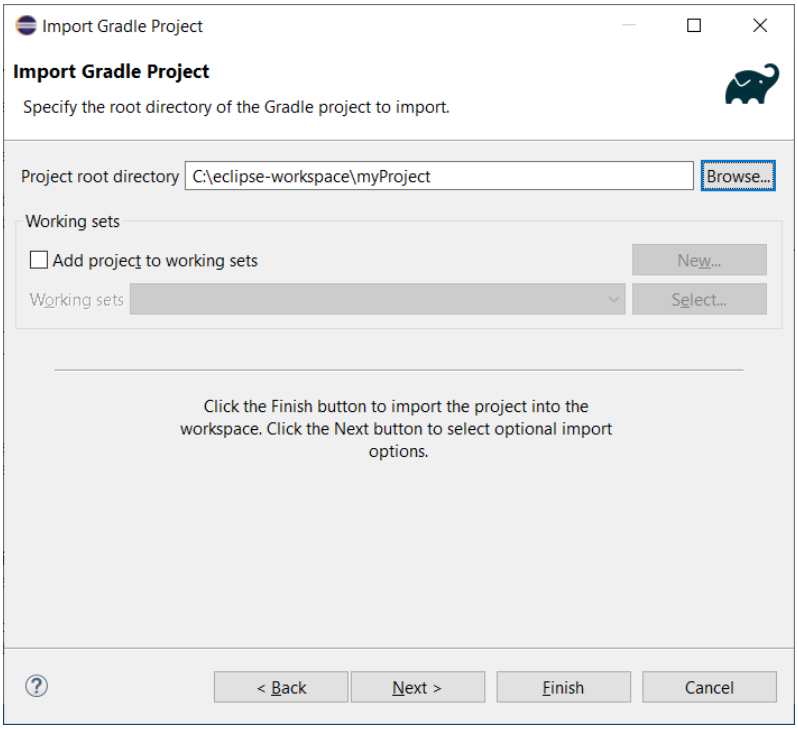


Fig. 33: Project root folder in Eclipse

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

The Gradle project should now be opened in Eclipse.

4.7 Select a VEE Port

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

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

4.7.1 Using a Module Dependency

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

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

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

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

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

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

4.7.2 Using a Local VEE Port Directory

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

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

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

4.7.3 Using a Local VEE Port Archive

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

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

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

4.8 Run on Simulator

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

- The Application main class or Feature class must be configured, as described in *Configure a Project*.
- If the Application has a main class, the target VEE Port must be defined. Refer to the *Select a VEE Port* page to know the different ways to provide a VEE Port for a module project.
- If the Application has a Feature class and a `.kf` file containing the property `entryPoint` set to the Full Qualified Name of the Feature class in the `src/main/resources` folder , the target Kernel must be defined. Refer to the *Select a Kernel* page to know the different ways to provide a Kernel for a module project.

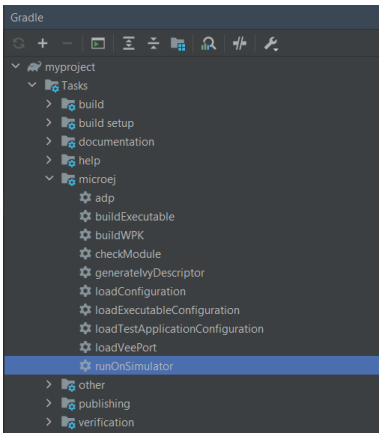
Once these prerequisites are fulfilled, the Application can be started with the Simulator:

Android Studio / IntelliJ IDEA

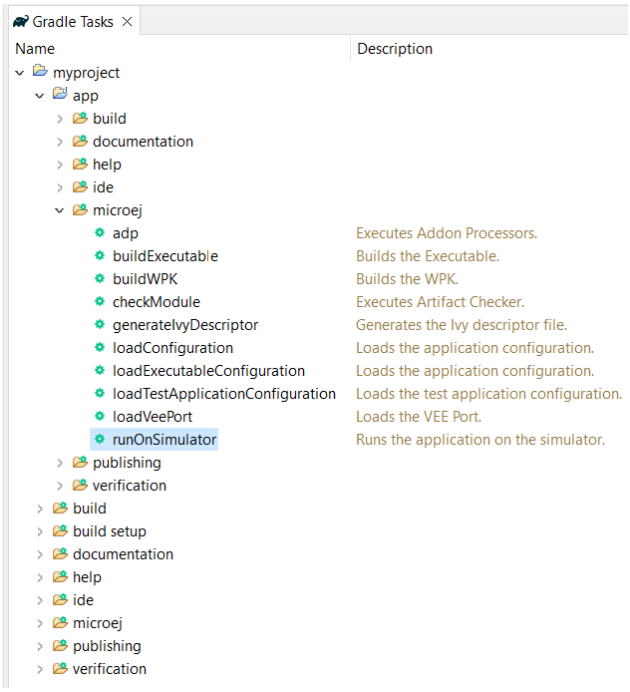
Eclipse

Command Line Interface

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



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



From the command line interface:

```
$ ./gradlew runOnSimulator
```

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

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

SUCCESS
```

Warning: The execution of the `runOnSimulator` task can fail if the SDK EULA has not been accepted. In that case, you can set the `ACCEPT_MICROEJ_SDK_EULA_V3_1B` environment variable to `YES` and restart your IDE or you can define the `accept-microej-sdk-eula-v3-1b` System property by creating a *custom configuration*. For more information about SDK EULA, refer to the *Licenses* chapter.

4.8.1 Verbose Mode

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

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

4.8.2 Debug on Simulator

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

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

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

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

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

If you want to connect the IDE debugger:

Android Studio / IntelliJ IDEA

Eclipse

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

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

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

4.8.3 Generate Code Coverage

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

```
gradle :runOnSimulator -D"s3.cc.thread.period=15" -D"s3.cc.activated=true"
```

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

Description:

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

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

Option Name: `s3.cc.activated`

Description

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

4.8.4 Generate Heap Dump

Option Name: `s3.inspect.heap`

Description:

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

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

4.9 Build an Executable

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

- The property `applicationMainClass` must be defined in the `microej` configuration block of the Gradle build file of the project. It must be set to the Full Qualified Name of the Application main class, for example:

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

- A target VEE Port that uses an Architecture version `7.17` minimum must be defined. Refer to the *Select a VEE Port* page to know the different ways to provide a VEE Port for a module project.

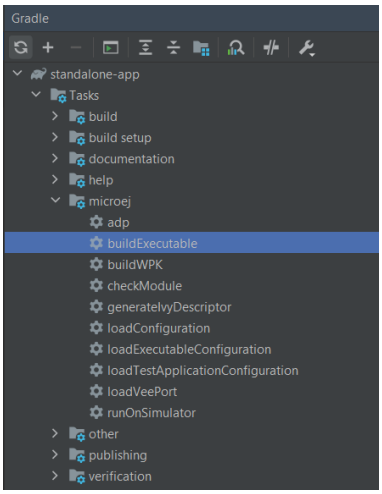
Once these prerequisites are fulfilled, the Executable can be built:

Android Studio / IntelliJ IDEA

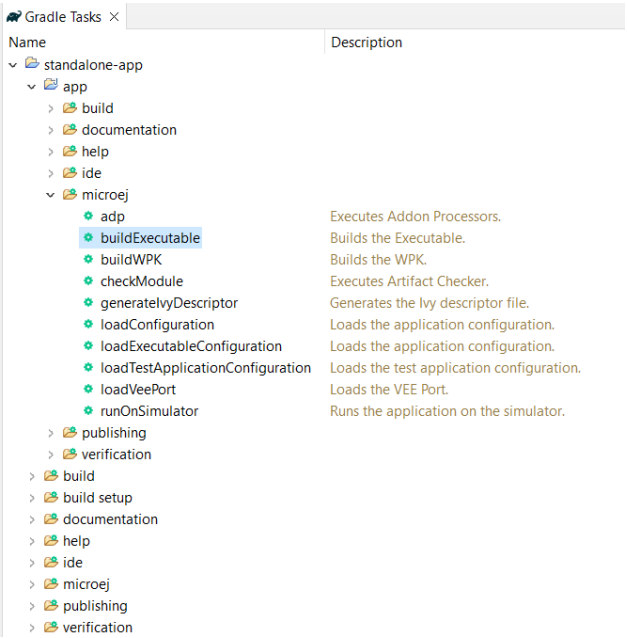
Eclipse

Command Line Interface

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



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



From the command line interface:

```
$ ./gradlew buildExecutable
```

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

4.10 Run on Device

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

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

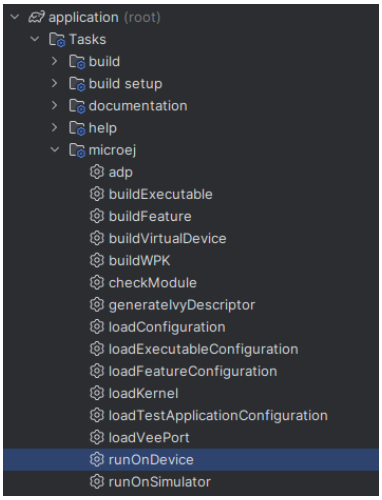
Once these prerequisites are fulfilled, the Application can be deployed on the Device:

Android Studio / IntelliJ IDEA

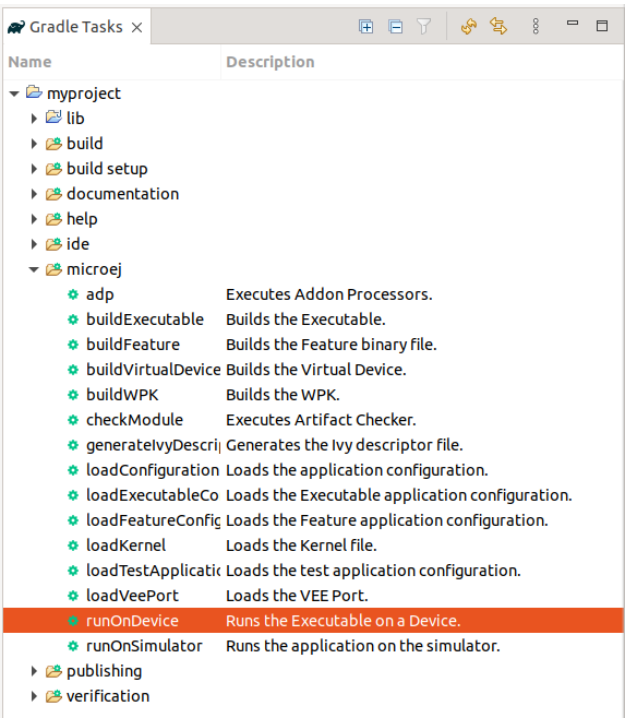
Eclipse

Command Line Interface

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



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



From the command line interface:

```
$ ./gradlew runOnDevice
```

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

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

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

The Application Executable is now deployed on the Device.

4.11 Select a Kernel

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

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

4.11.1 Using a Module Dependency

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

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

4.11.2 Using a Local Kernel

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

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

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

4.12 Build a Feature file

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

- The Application must contain a Java class implementing the `ej.kf.FeatureEntryPoint` interface.
- The Application must contain a `.kf` file in the `src/main/resources` folder containing at least the property `entryPoint` set to the Fully Qualified Name of the Application Feature class, for example:

  ```
  entryPoint=com.microej.MyFeature
  ```

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

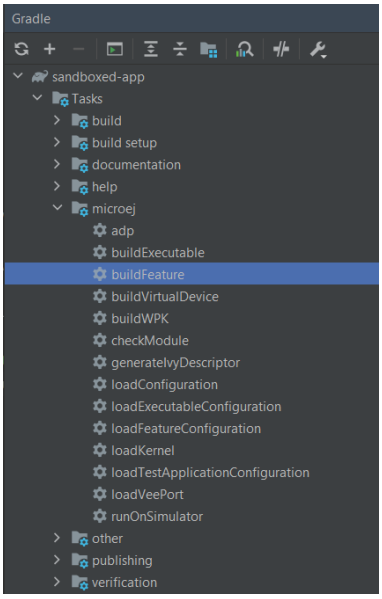
Once these prerequisites are fulfilled, the Feature file can be built:

Android Studio / IntelliJ IDEA

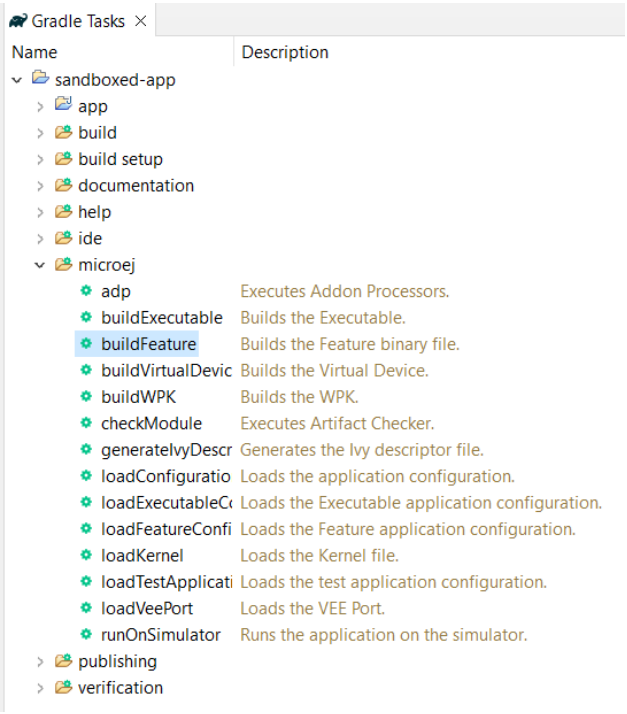
Eclipse

Command Line Interface

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



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



From the command line interface:

```
$ ./gradlew buildFeature
```

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

4.13 Build a Virtual Device

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

The prerequisites to use the `buildVirtualDevice` task are:

- The property `applicationMainClass` must be defined in the `microej` configuration block of the Gradle build file of the project. It must be set to the fully qualified name of the Application main class, for example:

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

- A target VEE Port that uses an Architecture version `7.17` minimum must be defined. Refer to the *Select a VEE Port* page to know the different ways to provide a VEE Port for a module project.

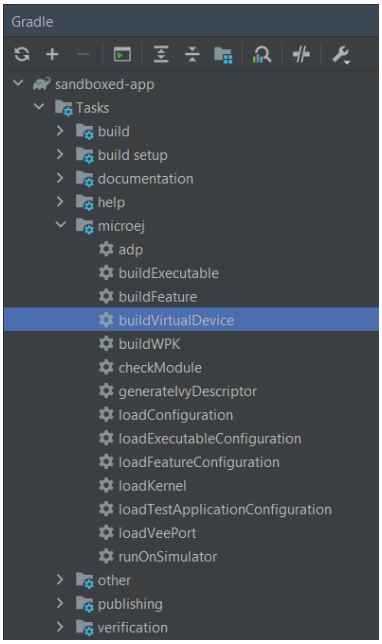
Once these prerequisites are fulfilled, the Virtual Device can be built:

Android Studio / IntelliJ IDEA

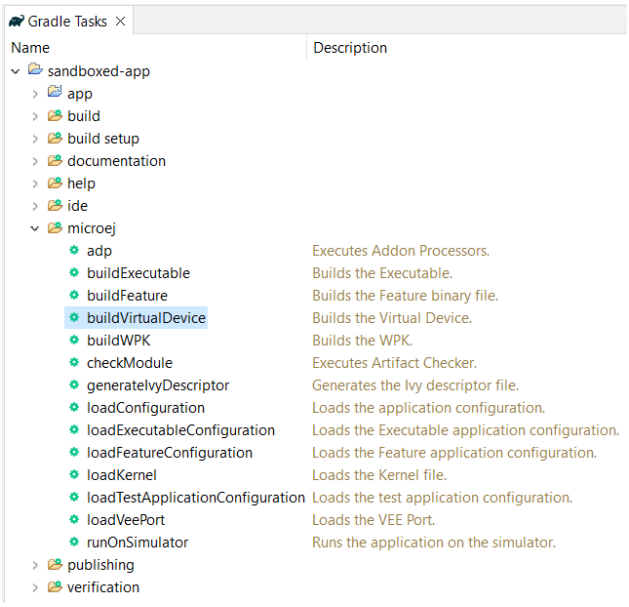
Eclipse

Command Line Interface

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



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



From the command line interface:

```
$ ./gradlew buildVirtualDevice
```

When the build is completed, the Virtual Device is available in the `build/virtualDevice` folder of the project. It can then be used to *run an Application on the Simulator* for example.

4.13.1 Add a Pre-Installed Application in a Virtual Device

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

To install an Application in a Virtual Device for a Kernel, you must declare the Application as a dependency of the project in the build file, with the `microejApplication` configuration:

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

Warning:

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

4.13.2 Add a Kernel API in a Virtual Device

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

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

4.14 Add a Dependency

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

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

4.14.1 Configurations

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

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

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

Here is an example of dependencies declaration for a project:

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

   testImplementation("ej.library.test:junit:1.7.1")

   microejVee("com.microej.platform.esp32.esp-wrover-kit-v41:HDAHT:1.8.0")
}
```

In this example, the `ej.library.runtime:basictool` module is used at compile time and runtime, the `ej.library.test:junit` module is used for the tests compilation and execution, and the `com.microej.platform.esp32.esp-wrover-kit-v41:HDAHT` module is the VEE Port used for build and test.

For an exhaustive list of the available configurations and more details on how to manage dependencies, refer to the following official documentations:

- Declaring dependencies
- Java plugin
- Java Library plugin

4.14.2 Version

The version declared in the dependencies of a build file are explicit:

- Release version: to depend on a released version of a module, the exact fixed version must be used (e.g., `1.0.0`).
- Snapshot version: to depend on a snapshot version (`-RC`) of a module, the version must be declared explicitly with the `-RC+` suffix (e.g., `1.0.0-RC+`).

Note: This is an important change compared to the SDK 5. In the SDK 5, using a fixed version (e.g., `1.0.0`) fetched the release version (e.g., `1.0.0`) if it existed, or a snapshot version (e.g., `1.0.0-RCxxx`) otherwise. This is not the case anymore in the SDK 6.

4.14.3 Dependencies Repositories

Gradle needs to know in which repositories the modules must be fetched and published. The *SDK 6 installation process* provides a Gradle Init Script to declare the *MicroEJ public repositories*. You can declare other repositories, either in the same Gradle Init Script and in any other location supported by Gradle. Refer to the official documentation for more information on repositories configuration.

It is important to note that the declaration order of the repositories matters. Gradle requests the repositories in the order they are declared and stops as soon as it finds a matching version.

4.15 Test a Project

The SDK provides the capabilities to implement and run tests for a project. It relies on the standard JUnit API.

There are different types of tests:

- Test on the Simulator
- Test on a device
- Test on a J2SE VM

Each type of test is detailed in the next sections.

4.15.1 JUnit Compliance

The SDK relies on JUnit, the most popular Java testing framework, to define and execute the tests. It supports a subset of JUnit 4, namely the annotations: `@After`, `@AfterClass`, `@Before`, `@BeforeClass`, `@Ignore`, `@Test`.

Each test case entry point must be declared using the `org.junit.Test` annotation (`@Test` before a method declaration). Refer to JUnit documentation to get details on the usage of other annotations.

4.15.2 Gradle Integration

Tests are configured and launched by Gradle. Gradle provides 2 ways to configure tests in a project:

- By using the built-in `Test` task, as described in Testing in Java & JVM projects.
- By using the new `JVM Test Suite` plugin, as described in The JVM Test Suite Plugin.

The `JVM Test Suite` plugin provides an enhanced model to declare multiple groups of automated testsuites, and is therefore the recommended way to configure your tests. The next sections use the `JVM Test Suite` plugin to explain how to configure and run tests, but the same results can be achieved with the `Test` task.

4.15.3 Test on Simulator

Tests can be executed on the Simulator. They are run on a target VEE Port and generate a JUnit XML report.

Executing tests on the Simulator allows to check the behavior of the code in an environment similar to the target device but without requiring the board. This solution is therefore less constraining and more portable than testing on the board.

Configure the Testsuite

The configuration of the testsuites of a project must be defined inside the following block in the `build.gradle.kts` file:

```
testing {
   suites { // (1)
      val test by getting(JvmTestSuite::class) { // (2)
         microej.useMicroejTestEngine(this) // (3)

         dependencies { // (4)
             implementation(project())
             implementation("ej.api:edc:1.3.5")
             implementation("ej.library.test:junit:1.7.1")
             implementation("org.junit.platform:junit-platform-launcher:1.8.2")
         }
      }
   }
}
```

This piece of configuration is the minimum configuration required to define a testsuite on the Simulator:

- `(1)` : configures all the testsuites of the project.
- `(2)` : configures the built-in `test` suite provided by Gradle. Use this testsuite to configure the tests on the Simulator.

- `(3)` : declares that this testsuite uses the MicroEJ Testsuite Engine. By default, the MicroEJ Testsuite Engine executes the tests on the Simulator.
- `(4)` : adds the dependencies required by the tests. The first line declares a dependency to the code of the project. The second line declares a dependency on the `edc` Library. The third line declares a dependency to the JUnit API used to annotate Java Test classes. Finally the fourth line declares a dependency to a required JUnit library.

Create a Test Class

The SDK provides a JUnit library containing the subset of the supported JUnit API: `ej.library.test:junit`. Before creating the Test class, make sure this library is declared in the testsuite dependencies:

```
testing {
   suites {
      val test by getting(JvmTestSuite::class) {
         ...
         dependencies {
            implementation("ej.library.test:junit:1.7.1")
         }
         ...
      }
   }
}
```

The test class can now be created in the `src/test/java` folder. This can be done manually or with IDE menu:

Android Studio / IntelliJ IDEA

Eclipse

- right-click on the `src/test/java` folder.
- select `New` > `Java Class`, then press `Alt` + `Insert` and select `Test Method`.
- right-click on the `src/test/java` folder.
- select `New` > `Other...` > `Java` > `JUnit` > `New JUnit Test Case`.

Note: Gradle allows to define alternative folders for test sources but it would require additional configuration, so it is recommended to stick with the `src/test/java` folder.

Setup a VEE Port

Before running tests, at least one target VEE Port must be configured using one of the methods described in the *Select a VEE Port* page. If several VEE Ports are defined, the testsuite is executed on each of them.

Execute the Tests

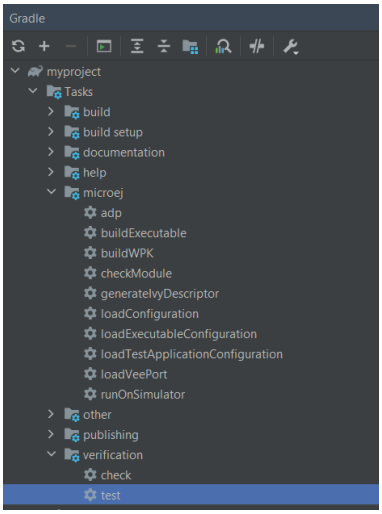
Once the testsuite is configured, it can be run thanks to the `test` Gradle task. This task is bound to the `check` and the `build` Gradle lifecycle tasks, which means that the tests are also executed when launching one of these tasks.

Android Studio / IntelliJ IDEA

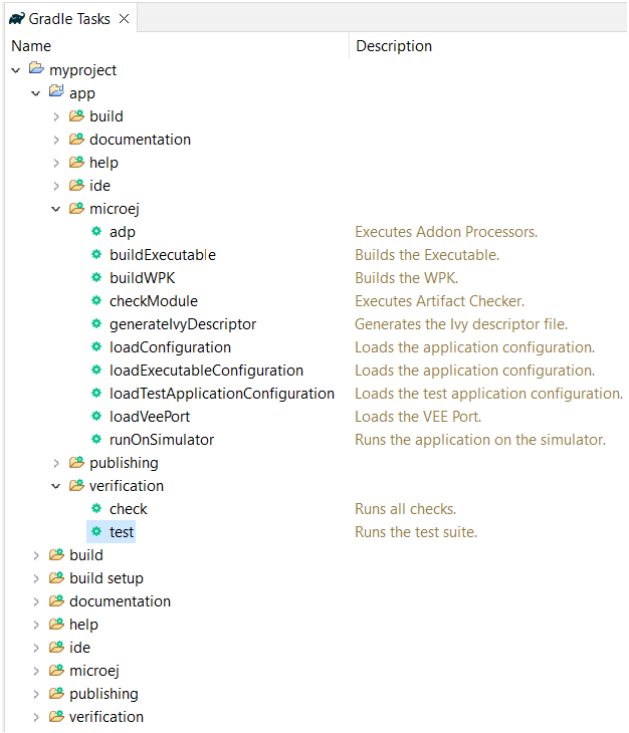
Eclipse

Command Line Interface

In order to execute the testsuite from Android Studio or IntelliJ IDEA, double-click on the task in the Gradle tasks view:



In order to execute the testsuite from Eclipse, double-click on the task in the Gradle tasks view:



In order to execute the testsuite from the Command Line Interface, execute this command:

```
$ ./gradlew test
```

Filter the Tests

Gradle automatically executes all the tests located in the test source folder. If you want to execute only a subset of these tests, Gradle provides 2 solutions:

- Filtering configuration in the build script file.
- Filtering option in the command line.

The tests filtering configuration must be done in the `filter` block of the test task:

```
testing {
   suites {
     val test by getting(JvmTestSuite::class) {

         ...

         targets {
             all {
                 testTask.configure {
                     filter {
                         includeTestsMatching("com.mycompany.*")
                     }
                 }
             }
         }
     }
   }
}
```

This example tells Gradle to run the tests located in the `com.mycompany` package only. Other methods are available for test filtering, such as `excludeTestsMatching` to exclude tests. Refer to the TestFilter documentation for the complete list of available filtering methods.

As mentionned earlier, Gradle allows to filter the tests from the command line directly, thanks to the `--tests` option:

```
./gradlew test --tests MyTestClass
```

This can be convenient to quickly execute one test for example, without requiring a change in the build script file.

Refer to the Gradle Test filtering documentation for more details on how to filter the tests and on the available patterns.

> **Warning:** At the moment, only class-level filtering is supported. This means that, for instance, it is not possible to run a single test method within a test class.

4.15.4 Test on Device

The SDK allows to execute a testsuite on a device. This requires to:

- Have a VEE Port which implements the *BSP Connection*.
- Have a device connected to your workstation both for programming the Executable and getting the output traces. Consult your VEE Port specific documentation for setup.
- Start the *Serial to Socket Transmitter* tool if the VEE Port does not redirect execution traces.

The configuration is similar to the one used to execute a testsuite on the Simulator.

1. Follow the instructions to *setup a testsuite on the Simulator*.
2. In the build script file, replace the line:

   ```
   microej.useMicroejTestEngine(this)
   ```

   by:

   ```
   microej.useMicroejTestEngine(this, TestTarget.EMB)
   ```

3. Add the `import` statement at the beginning of the file:

   ```
   import com.microej.gradle.plugins.TestTarget
   ```

4. Add the required properties as follows:

   ```
   val test by getting(JvmTestSuite::class) {
      microej.useMicroejTestEngine(this, TestTarget.EMB)

      targets {
         all {
            testTask.configure {
               doFirst {
                     systemProperties = mapOf(
                        // Enable the build of the Executable
                        "microej.testsuite.properties.deploy.bsp.microejscript" to "true",
                        "microej.testsuite.properties.microejtool.deploy.name" to
   ↪"deployToolBSPRun",
                        // Tell the testsuite engine that the VEE Port Run script␣
   ↪redirects execution traces
                        "microej.testsuite.properties.launch.test.trace.file" to "true",
                        // Configure the TCP/IP address and port if the VEE Port Run␣
   ↪script does not redirect execution traces
                        "microej.testsuite.properties.testsuite.trace.ip" to "localhost",
                        "microej.testsuite.properties.testsuite.trace.port" to "5555"
                     )
               }
            }
         }
      }
   }
   ```

The properties are:

- `microej.testsuite.properties.deploy.bsp.microejscript` : enables the build of the Executable. It is required.
- `microej.testsuite.properties.microejtool.deploy.name` : name of the tool used to deploy the Executable to the board. It is required. It is generally set to `deployToolBSPRun` .
- `microej.testsuite.properties.launch.test.trace.file` : enables the redirection of the traces in file. If the VEE Port does not have this capability, the *Serial to Socket Transmitter* tool must be used to redirect the traces to a socket.
- `microej.testsuite.properties.testsuite.trace.ip` : TCP/IP address used by the *Serial to Socket Transmitter* tool to redirect traces from the board. This property is only required if the VEE Port does not redirect execution traces.
- `microej.testsuite.properties.testsuite.trace.port` : TCP/IP port used by the *Serial to Socket Transmitter* tool to redirect traces from the board. This property is only required if the VEE Port does not redirect execution traces.

Any other property can be passed to the Test Engine by prefixing it by `microej.testsuite.properties.` . For example, to set the the Immortal heap size:

```
systemProperties = mapOf(
   "microej.testsuite.properties.core.memory.immortal.size" to "8192",
   ...
)
```

4.15.5 Test on J2SE VM

The SDK allows to run tests on a J2SE VM. This can be useful when the usage of mock libraries like Mockito is needed.

There is nothing specific related to MicroEJ to run tests on a J2SE VM. Follow the Gradle documentation to setup such tests. As an example, here is a typical configuration to execute the tests located in the `src/test/java` folder:

```
testing {
   suites {
      val test by getting(JvmTestSuite::class) {
         useJUnitJupiter()
      }
   }
}
```

4.15.6 Test Suite Reports

Once a testsuite is completed, the JUnit XML report is generated in the module project location `build/testsuite/output/<date>/testsuite-report.xml` .

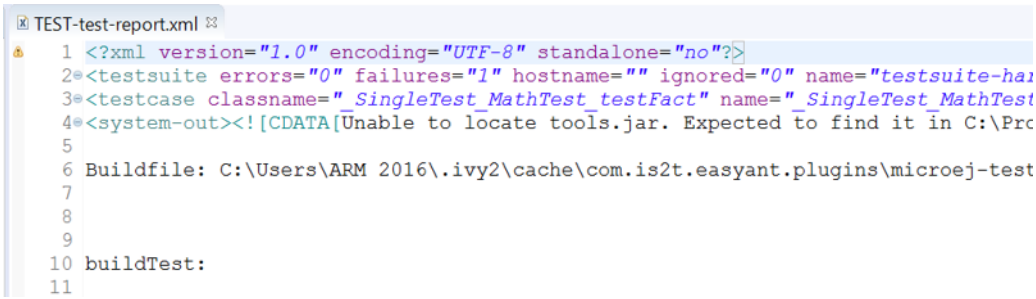


Fig. 34: Example of MicroEJ Test Suite XML Report

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

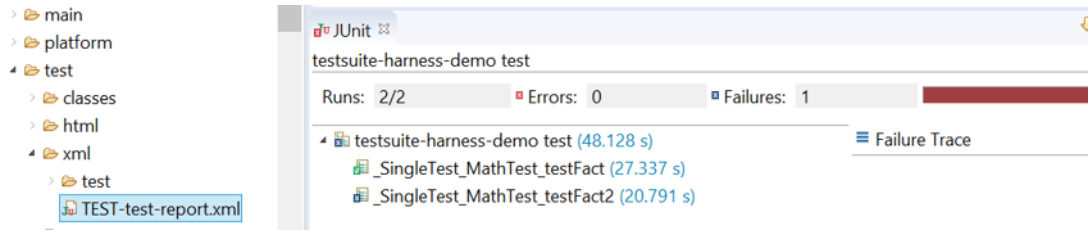


Fig. 35: Example of MicroEJ Test Suite XML Report in JUnit View

4.15.7 Mixing tests

The SDK allows to define multiple testsuites on different targets. For example, you can configure a testsuite to run tests on the Simulator and a testsuite to run tests on a device.

Configuring multiple testsuites is almost only a matter of aggregating the testsuite declarations documented in the previous sections, as described in the Gradle documentation.

Mixing tests on the Simulator and on a device

If you need to define a testsuite to run on the Simulator and a testsuite to run on a device, the only point to take care is related to the tests source location, because:

- Gradle automatically uses the testsuite name to know the tests source folder to use. For example, for a testsuite named `test` (the built-in testsuite), the folder `src/test/java` is used, and for a testsuite named `testOnDevice`, the folder `src/testOnDevice/java` is used.
- Tests classes executed by the MicroEJ Test Engine on the Simulator and on device are not directly the tests source classes. The SDK generates new tests classes, based on the original ones, but compliant with the MicroEJ Test Suite mechanism. This process assumes by default that the tests classes are located in the `src/test/java` folder.

Therefore:

- It is recommended to use the built-in `test` testsuite for either the tests on the Simulator or the tests on device. This avoids extra configuration to change the location of the tests source folder.
- The tests source folder of the other testsuite must be changed to use the `src/test/java` folder as well:

```
testing {
   suites {
      val test by getting(JvmTestSuite::class) {
         microej.useMicroejTestEngine(this)

         dependencies {
            implementation(project())
            implementation("ej.library.test:junit:1.7.1")
            implementation("org.junit.platform:junit-platform-launcher:1.8.2")
         }
      }

      val testOnDevice by registering(JvmTestSuite::class) {
         microej.useMicroejTestEngine(this, TestTarget.EMB)

         sources {
            java {
               setSrcDirs(listOf("src/test/java"))
            }
            resources {
               setSrcDirs(listOf("src/test/resources"))
            }
         }

         dependencies {
            implementation(project())
            implementation("ej.library.test:junit:1.7.1")
            implementation("org.junit.platform:junit-platform-launcher:1.8.2")
         }

         targets {
             all {
               testTask.configure {
                  doFirst {
                     systemProperties = mapOf(
                        "microej.testsuite.properties.deploy.bsp.microejscript" to "true",
                        "microej.testsuite.properties.microejtool.deploy.name" to
↪"deployToolBSPRun",
                        "microej.testsuite.properties.testsuite.trace.ip" to "localhost",
                        "microej.testsuite.properties.testsuite.trace.port" to "5555"
                     )
                  }
               }
            }
         }
      }
   }
}
```

The important part is the `sources` block of the `testOnDevice` testsuite. This allows to use the `src/test/java` and `src/test/resources` folders as the tests source folders.

With this configuration, all tests are executed on both the Simulator and the device. If you want to have different tests for each testsuite, it is recommended to separate the tests in different packages. For example the tests on the

Simulator could be in `src/test/java/com/mycompany/sim` and the tests on the device could be in `src/test/java/com/mycompany/emb`. Then you use the test filtering capabilities to configure which package to run in which testsuite:

```
testing {
   suites {
      val test by getting(JvmTestSuite::class) {
         ...

         targets {
             all {
               testTask.configure {
                  ...

                  filter {
                     includeTestsMatching("com.mycompany.sim.*")
                  }
               }
            }
         }
      }

      val testOnDevice by registering(JvmTestSuite::class) {
         ...

         targets {
             all {
               testTask.configure {
                  ...

                  filter {
                     includeTestsMatching("com.mycompany.emb.*")
                  }
               }
            }
         }
      }
   }
}
```

Mixing tests on the Simulator and on a J2SE VM

Defining tests on the Simulator and on a J2SE VM is only a matter of aggregating the configuration of each testsuite:

```
testing {
   suites {
      val test by getting(JvmTestSuite::class) {
         microej.useMicroejTestEngine(this)
         ...
      }

      val testOnJ2SE by registering(JvmTestSuite::class) {
```

(continues on next page)

(continued from previous page)

```
            useJUnitJupiter()
            ...
        }
    }
}
```

As explained in the previous section, it is recommended to use the built-in `test` testsuite for the tests on the Simulator since it avoids adding confguration to change the tests sources folder. With this configuration, tests on the Simulator are located in the `src/test/java` folder, and tests on a J2SE VM are located in the `src/testOnJ2SE/java` folder.

4.15.8 Configure the Testsuite Engine

The engine used to execute the testsuite provides a set of configuration parameters that can be defined with System Properties. For example, to set the timeout of the tests:

- In the command line with `-D`:

```
./gradlew test -Dmicroej.testsuite.timeout=120
```

- In the build script file:

```
testing {
    suites {
        val test by getting(JvmTestSuite::class) {
            ...

            targets {
                all {
                    testTask.configure {
                        ...

                        doFirst {
                            systemProperties = mapOf(
                                "microej.testsuite.timeout" to "120"
                            )
                        }
                    }
                }
            }
        }
    }
}
```

The following configuration parameters are available:

Name	Description	Default
microej.testsuite.timeout	The time in seconds before a test is considered as failed. Set it to 0 to disable the timeout.	60
microej.testsuite.jvmArgs	The arguments to pass to the Java VM started for each test.	Not set
microej.testsuite.lockPort	The localhost port used by the framework to synchronize its execution with other frameworks on same computer. Synchronization is not performed when this port is 0 or negative.	0
microej.testsuite.retry.count	A test execution may not be able to produce the success trace for an external reason, for example an unreliable harness script that may lose some trace characters or crop the end of the trace. For all these unlikely reasons, it is possible to configure the number of retries before a test is considered to have failed.	0
microej.testsuite.retry.if	Regular expression checked against the test output to retry the test. If the regular expression is found in the test output, the test is retried (up to the value of microej.testsuite.retry.count).	Not set
microej.testsuite.retry.unless	Regular expression checked against the test output to retry the test. If the regular expression is not found in the test output, the test is retried (up to the value of microej.testsuite.retry.count).	Not set
microej.testsuite.verbose.level	Verbose level of the testsuite output. Available values are error, warning, info, verbose and debug.	info

4.15.9 Inject Application Options

Standalone Application Options can be defined to configure the Application or Library being tested. They can be defined globally, to be applied on all tests, or specifically to a test.

Inject Application Options Globally

In order to define an Application Option globally, it must be prefixed by microej.testsuite.properties. and passed as a System Property, either in the command line or in the build script file. For example, to inject the property core.memory.immortal.size:

- In the command line with -D:

```
./gradlew test -Dmicroej.testsuite.properties.core.memory.immortal.size=8192
```

- In the build script file:

```
testing {
   suites {
      val test by getting(JvmTestSuite::class) {
         ...

         targets {
            all {
               testTask.configure {
```

(continues on next page)

(continued from previous page)

```
                ...

                doFirst {
                   systemProperties = mapOf(
                      "microej.testsuite.properties.core.memory.immortal.size
↪" to "8192"
                   )
                }
             }
          }
       }
    }
}
```

Inject Application Options For a Specific Test

In order to define an Application Option for a specific test, it must set in a file with the same name as the generated test case file, but with the `.properties` extension instead of the `.java` extension. The file must be put in the `src/test/resources` folder and within the same package than the test file. For example, to inject a Application Option for the test class `com.mycompany.MyTest`, it must be set in a file named `src/test/resources/com.mycompany/MyTest.properties`.

4.16 Publish a Project

Publishing is the process by which the built artifacts of a module is made available to other modules or any other systems.

The requirements to publish a module are:

- Defining the `name` of the module. It is set by default to the name of the module folder, and can be changed in the `settings.gradle.kts` file located at the root of the module, thanks to the property `rootProject.name`:

```
rootProject.name = "myModule"
```

- Defining the `group` and `version` properties. They can be set in the `build.gradle.kts` file:

```
group = "com.mycompany"
version = "1.0.0"
```

- Declaring a `maven` publication repository. This can be done in the build file for example, with:

```
publishing {
    repositories {
        maven {
            name = "mavenPublish"
            url = uri("https://my.server/repository")
        }
    }
}
```

Refer to the official documentation for more information on publication repositories.

Then the publication of a module to a repository is achieved by executing the `publish` task:

```
$ ./gradlew publish
```

The following artifacts are automatically published:

- The main artifact, which is the JAR file for Application and Add-On Library natures.
- The README.md file.
- The CHANGELOG.md file.
- The LICENSE.txt file.
- The ASSEMBLY_EXCEPTION.txt file.
- The Gradle module descriptor file.
- The Ivy descriptor file (to allow SDK 5 project to fetch it).
- The WPK file, if the project is an Application.
- The Virtual Device, if the project is an Application containing a Java main class.

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

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

If the project is an Application containing a Java class that implements the `ej.kf.FeatureEntryPoint` interface, the Feature is not built and published by default (when launching a `./gradlew build` or a `./gradlew publish` for example). This default behavior can be changed by adding the `produceFeatureDuringBuild()` method in the `microej` configuration block of the Gradle build file of the project:

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

4.17 Development Tools

MicroEJ provides a number of tools to assist with various aspects of development. These tools are either command line tools or Eclipse IDE plugins.

Command line tools

Command line tools can be executed using the gradle task `:execTool` .

The format of the task is as follow:

```
gradle :execTool --name=TOOL_NAME --toolProperty="PROPERTY=VALUE" --toolProperty=
↪"PROPERTY=VALUE" ...
```

The parameter required `--name` is used to describe the name of the tool to execute. The optional parameters `--toolProperty` are used to configure the tool's options.

In addition, the tool's options can be defined in `configuration/tools/TOOL_NAME.properties`.

The following sections describe the command line tools and their options:

4.17.1 Stack Trace Reader

Principle

Stack Trace Reader is a MicroEJ tool that reads and decodes the MicroEJ stack traces. When an exception occurs, the *Core Engine* prints the stack trace on the standard output `System.out`. The class names, non-required types names(see *Types*), and method names obtained are encoded with a MicroEJ internal format. This internal format prevents embedding all class names and method names in the executable image to save some memory space. The Stack Trace Reader tool allows you to decode the stack traces by replacing the internal class names and method names with their real names. It also retrieves the line numbers in the Application.

Functional Description

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

When *Multi-Sandbox capability* is enabled, the stack trace reader can simultaneously decode heterogeneous stack traces with lines owned by different Sandboxed Applications and the Kernel. Lines owned by the Kernel can be decoded with the Executable debug information file (optionally made available by your Kernel provider).

Use (Standalone Application)

For example, write the following new line to dump the currently executed stack trace on the standard output.

```
package com.mycompany;

public class Test {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
        new Exception().printStackTrace();
    }
}
```

To decode an application stack trace, the stack trace reader tool requires the application executable ELF file. In the case of a platform with full BSP connection (see *BSP Connection Cases*), the file is `application.out` in the output folder. In the other cases, the ELF file is generated by the C toolchain when building the BSP project (usually a `.out` or `.axf` file).

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

```
VM START
Hello World from Gradle!
Exception in thread "main" @C:0x8070c30@
     at @C:0x8070c60@.@M:0x8075850:0x807585a@
     at @C:0x8070c30@.@M:0x80769a4:0x80769ba@
```

(continues on next page)

(continued from previous page)

```
    at @C:0x8070c30@.@M:0x807857c:0x8078596@
    at @C:0x8070c00@.@M:0x8074e04:0x8074e1a@
    at @C:0x8070ce0@.@M:0x807601c:0x807603c@
    at @C:0x806fe10@.@M:0x807779c:0x80777b0@
    at @C:0x8070c00@.@M:0x8077b40:0x8077b4c@
    at @C:0x8070c00@.@M:0x80779b0:0x80779bb@
```

To decode the trace, execute the `:execTool` task as followed:

```
gradle :execTool --name=stackTraceDecrypter \
  --toolProperty="proxy.connection.connection.type=console" \
  --toolProperty="application.file=../../executable/application/application.out" \
  --toolProperty="additional.application.files=" \
  --console plain
```

Paste the previous trace dump into the console. The output of the Stack Trace Reader is the following:

```
=============== [ MicroEJ Core Engine Trace ] ===============
console:
[INFO] Paste the MicroEJ core engine stack trace here.
    VM START
    Hello World from Gradle!
    Exception in thread "main" @C:0x8070c30@
         at @C:0x8070c60@.@M:0x8075850:0x807585a@
         at @C:0x8070c30@.@M:0x80769a4:0x80769ba@
         at @C:0x8070c30@.@M:0x807857c:0x8078596@
         at @C:0x8070c00@.@M:0x8074e04:0x8074e1a@
         at @C:0x8070ce0@.@M:0x807601c:0x807603c@
         at @C:0x806fe10@.@M:0x807779c:0x80777b0@
         at @C:0x8070c00@.@M:0x8077b40:0x8077b4c@
         at @C:0x8070c00@.@M:0x80779b0:0x80779bb@
    VM START
    Hello World from Gradle!
    Exception in thread "main" java.lang.Throwable
         at java.lang.System.getStackTrace(Unknown Source)
         at java.lang.Throwable.fillInStackTrace(Throwable.java:82)
         at java.lang.Throwable.<init>(Throwable.java:32)
         at java.lang.Thread.dumpStack(Thread.java:573)
         at com.microej.Main.main(Main.java:22)
         at java.lang.MainThread.run(Thread.java:855)
         at java.lang.Thread.runWrapper(Thread.java:464)
         at java.lang.Thread.callWrapper(Thread.java:449)
```

Options

Option: Executable file

Option Name: `application.file`

Required?: Yes

Description:

Specify the full path of a full linked elf file.

Option: Additional object files

Option Name: `additional.application.files`

Required?: Yes

Option: Connection type

Option Name: `proxy.connection.connection.type`

Required?: Yes

Available values:

- `console`
- `file`
- `uart`
- `socket`

Description:

Specify the connection type between the device and PC.

Option: Port

Option Name: `pcboardconnection.usart.pc.port`

Required?: (For `uart` Connection Type)

Description:

Format: `port name`

Specifies the PC COM port:

- Windows - `COM1` , `COM2` , `...` , `COM*n*`
- Linux - `/dev/ttyS0` , `/dev/ttyS1` , `...` , `/dev/ttyS*n*`

Option: Baudrate

Option Name: `pcboardconnection.usart.pc.baudrate`

Required?: (For `uart` Connection Type)

Available values:

- `9600`
- `38400`
- `57600`
- `115200`

Description:

Defines the COM baudrate for PC-Device communication.

Option: Port

Option Name: `pcboardconnection.socket.port`

Required?: (For `socket` Connection Type)

Description:

IP port.

Option: Address

Option Name: `pcboardconnection.socket.address`

Required?: (For `socket` Connection Type)

Description:

IP address, on the form A.B.C.D. or empty.

Option: Stack trace file

Option Name: `pcboardconnection.file.path`

Required?:

Description:

Path to a stack trace file or empty.

4.17.2 Code Coverage Analyzer

Principle

The Simulator features an option to output `.cc` (Code Coverage) files that represent the use rate of functions of an application. It traces how the opcodes are really executed.

Functional Description

The Code Coverage Analyzer scans the output `.cc` files, and outputs an HTML report to ease the analysis of methods coverage. The HTML report is available in a folder named htmlReport in the same folder as the .cc files.

Dependencies

In order to work properly, the Code Coverage Analyzer should input the `.cc` files (See *Generate Code Coverage*). The `.cc` files rely on the classpath used during the execution of the Simulator to the Code Coverage Analyzer. Therefore the classpath is considered to be a dependency of the Code Coverage Analyzer.

Installation

This tool is a built-in Architecture tool.

Use

A MicroEJ tool is available to launch the Code Coverage Analyzer tool. The tool name is Code Coverage Analyzer.

Two levels of code analysis are provided, the Java level and the bytecode level. Also provided is a view of the fully or partially covered classes and methods. From the HTML report index, just use hyperlinks to navigate into the report and source / bytecode level code.

```
gradle :execTool --name=codeCoverageAnalyzer \
    --toolProperty="cc.dir=/MODULE_PATH/build/output/com.microej.Main/cc/" \
    --toolProperty="cc.includes=" \
    --toolProperty="cc.excludes=" \
    --toolProperty="cc.src.folders=/MODULE_PATH/src" \
    --toolProperty="cc.html.dir=/MODULE_PATH/cc"
```

Options

Option: *.cc files folder

Option Name: `cc.dir`

Description:

Specify a folder which contains the cc files to process (*.cc).

Option: Source Folders

Option Name: `cc.src.folders`

Description:

A list of folders to the source files.

Option: HTML Dir

Option Name: `cc.html.dir`

Description:

The output directory for the HTML report.

Option: Includes

Option Name: `cc.includes`

Description:

List packages and classes to include to code coverage report. If no package/class is specified, all classes found in the project classpath will be analyzed.

Examples:

`packageA.packageB.*` : includes all classes which are in package `packageA.packageB`

`packageA.packageB.className` : includes the class `packageA.packageB.className`

Option: Excludes

Option Name: `cc.excludes`

Description:

List packages and classes to exclude to code coverage report. If no package/class is specified, all classes found in the project classpath will be analyzed.

Examples:

`packageA.packageB.*` : excludes all classes which are in package `packageA.packageB`

`packageA.packageB.className` : excludes the class `packageA.packageB.className`

IDE tools

Eclipse IDE tools are graphical tools which are available as Eclipse plugins: Memory Map Analyzer, Heap Analyzer and Font Designer.

Follow these steps to install the latest stable version of these tools:

- Install Eclipse IDE for Java Developers - Minimum supported version is `2022-03` .
- In Eclipse, go to `Help` > `Eclipse Marketplace...` .
- In the `Find` field, type `MicroEJ Tools` , then press `Enter` .
- Click on the `Install` button of the `MicroEJ Tools` plugin.

- Accept the license, then click on the `Finish` button.
- In the upcoming `Trust Artifacts` window, check the `Unsigned` item and click on `Trust Selected` button.

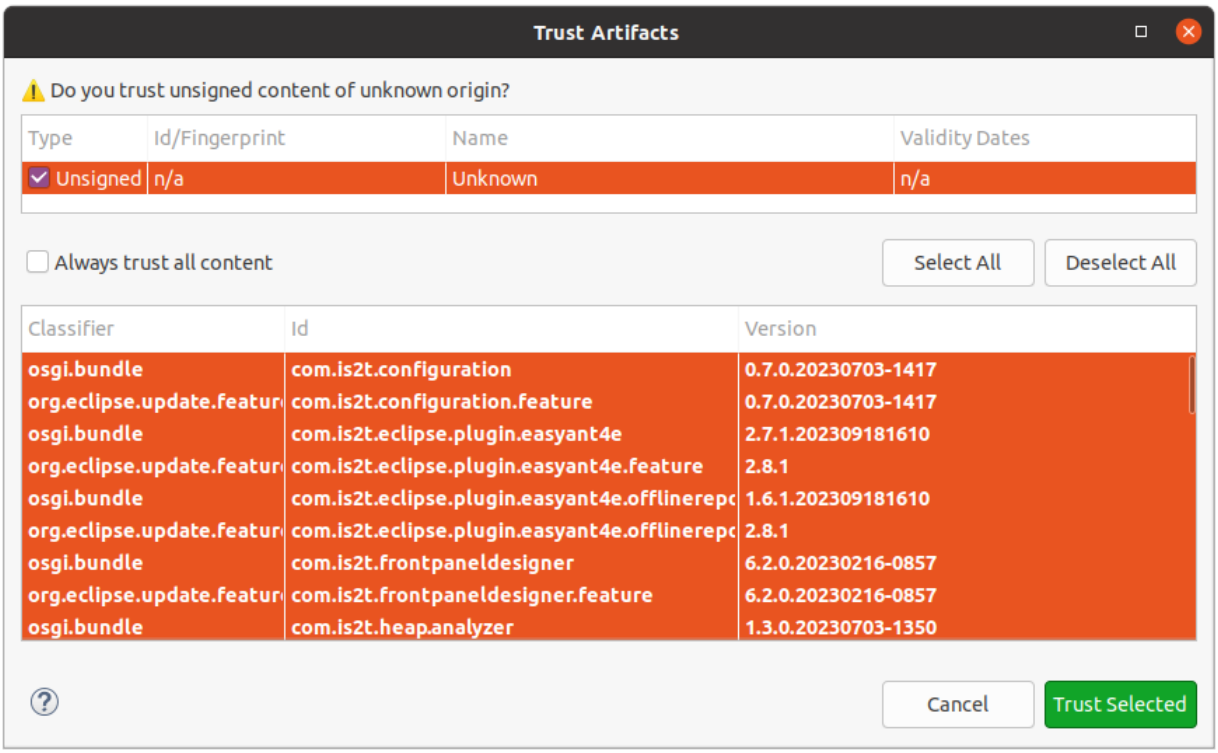


Fig. 36: Eclipse Plugin Installation - Trust Artifacts

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

The following sections describe the IDE tools and their options:

4.17.3 Memory Map Analyzer

Principle

When the Executable of an Application is built, a Memory Map file is generated. This file can be visualized with the Memory Map Analyzer, an Eclipse IDE plug-in. It displays the memory consumption of different features in the RAM and ROM.

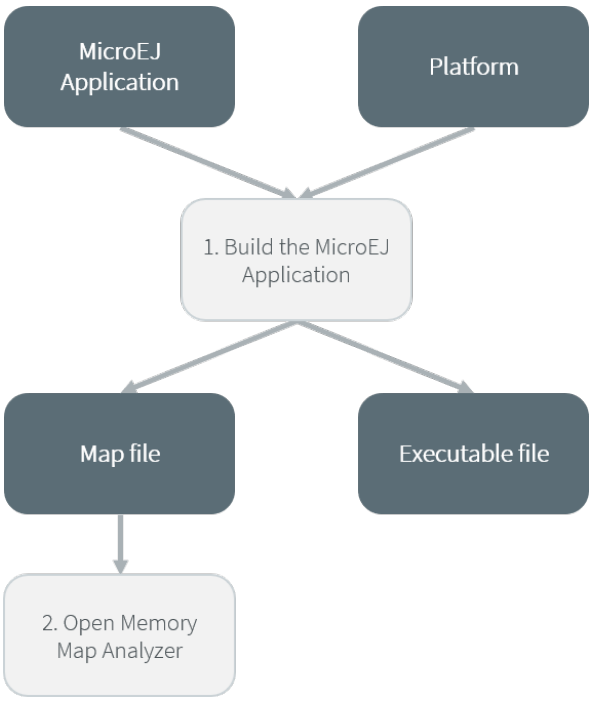


Fig. 37: Memory Map Analyzer Process

Use

When the Executable file of an Application has been produced, the Memory Map file is available at `build/executable/application/SOAR.map`.



Fig. 38: Memory Map File

You can visualize it by following these steps:

- Make sure *the Eclipse IDE is installed with the required plugin*, then launch it.
- Click on File > Open File… .
- Select the Memory Map file.

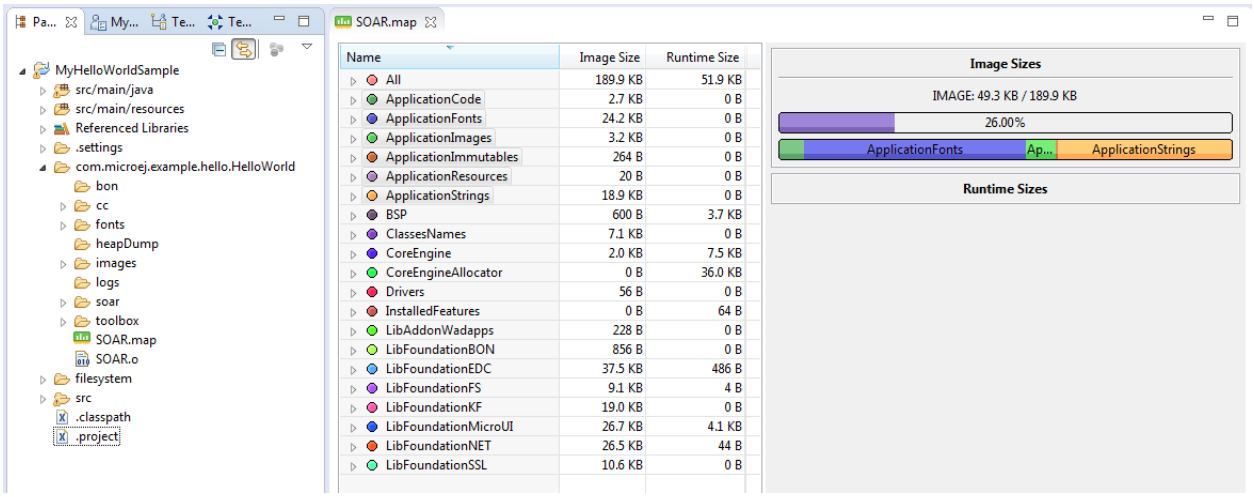


Fig. 39: Consult Full Memory

You can select an item (or several) to show the memory used by this item(s) on the right, or select `All` to show the memory used by all items. This special item performs the same action as selecting all items in the list.

You can also select an item in the list, and expand it to see all symbols used by the item. This view is useful in understanding why a symbol is embedded.

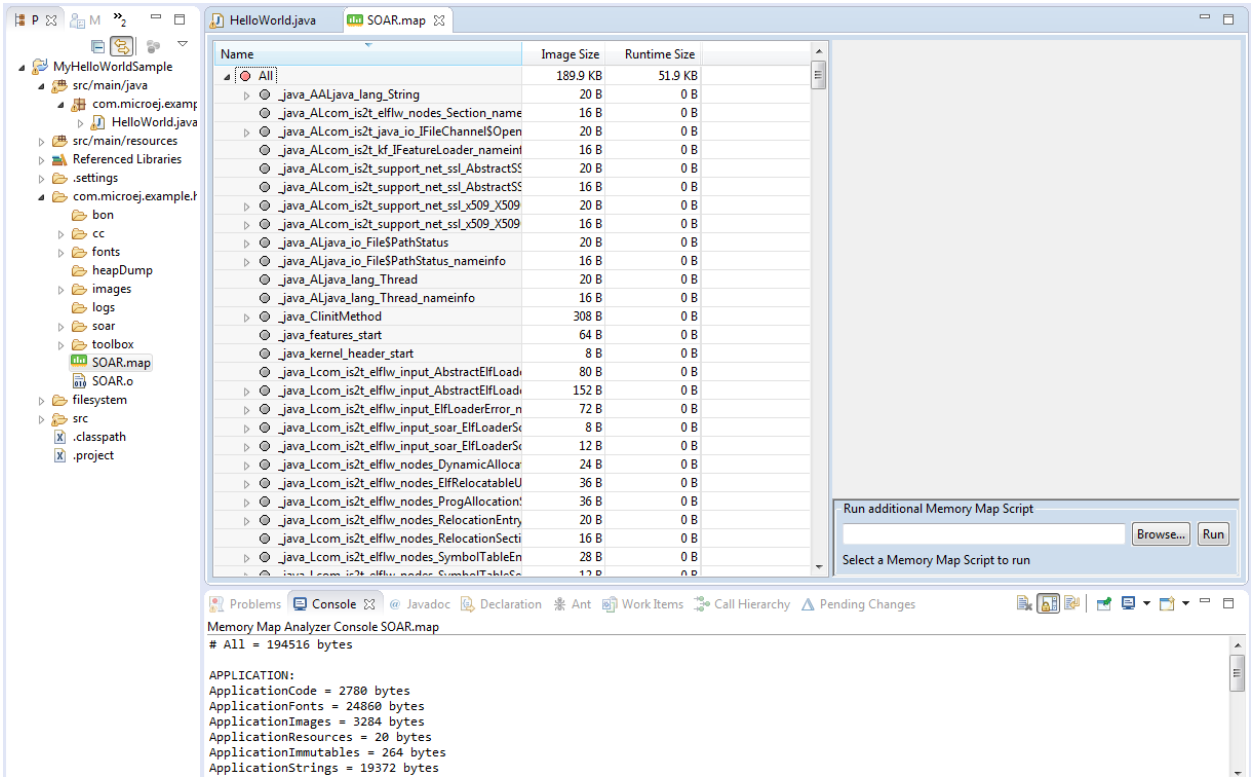


Fig. 40: Detailed View

4.17.4 Heap Dumper & Heap Analyzer

Introduction

Heap Dumper is a tool that allows to get a snapshot of the heap of an Application running on the Simulator or on a device.

The Heap Analyzer is a set of tools to help developers understand the contents of the Java heap and find problems such as memory leaks. For its part, the Heap Analyzer IDE plugin is able to visualize dump files. It helps you analyze their contents thanks to the following features:

- memory leaks detection
- objects instances browse
- heap usage optimization (using immortal or immutable objects)

The Heap

The heap is a memory area used to hold Java objects created at runtime. Objects persist in the heap until they are garbage collected. An object becomes eligible for garbage collection when there are no longer any references to it from other objects.

Heap Dump

A heap dump is an XML file (with the `.heap` extension) that provides a snapshot of the heap contents at the moment the file is created. It contains a list of all the instances of both class and array types that exist in the heap. For each instance, it records:

- The time at which the instance was created
- The thread that created it
- The method that created it

For instances of class types, it also records:

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

For instances of array types, it also records:

- The type of the contents of the array
- The contents of the array

For each referenced class type, it records the values in the static fields of the class.

Heap Analyzer Tools

The Heap Analyzer is an Eclipse IDE plugin that adds three tools to the MicroEJ environment.

Tool name	Number of input files	Purpose
Heap Viewer	1	Shows what instances are in the heap, when they were created, and attempts to identify problem areas
Progressive Heap Usage	1 or more	Shows how the number of instances in the heap has changed over time
Compare	2	Compares two heap dumps, showing which objects were created, or garbage collected, or have changed values

Heap Dumper

When the Heap Dumper option is enabled, the garbage collector process ends by performing a dump that represents a snapshot of the heap at this moment. To generate such dump, you must explicitly call the `System.gc()` method in your code.

This tool can be used with the Simulator and with a device.

Simulator

In order to generate a Heap dump of an Application running on the Simulator:

- Set the `s3.inspect.heap` Application properties to `true`.
- Update your Application code to call the `System.gc()` method where you need a Heap dump.
- run the Application on the Simulator.

When the `System.gc()` method is called, a `.heap` file is generated in the `build/output/application/heapDump/` folder of the Application project.

Device

In order to generate a Heap dump of an Application running on a device:

- Update your Application code to call the `System.gc()` method where you need a Heap dump.
- Build the Executable and deploy it on the device.

When the Application is executed on the device and the `System.gc()` method is called, a Heap dump is performed and stored in the device memory. You then have to:

- *Retrieve the hex file from the device*
- *Extract the Heap dump from the hex file*

Retrieve the `.hex` file from the device

Retrieving the `.hex` file from the device can be done with Eclipse CDT and GDB:

- Run your debug configuration.
- Open the `Disassembly` view.
- Type `LLMJVM_on_Runtime_gc_done` in location field.
- Add a breakpoint by double-clicking on the first line.
- Resume execution until the debugger stops on the breakpoint.
- Open your debug configuration and copy the host name and the port in the *Debugger* tab.
- Run a GDB console.
- Connect to the GDB server, for example: `target remote localhost:2331`.
- Dump the memory of the Java heap section by executing the following command line:

```
dump ihex memory heap.hex &_java_heap_start &_java_heap_end
```

You now have the `.hex` file and need to extract the Heap dump.

Extract the Heap dump from the `.hex` file

In order to extract the Heap dump from an `.hex` file, run the `execTool` Gradle task with the tool name `heapDumperPlatform`:

```
gradle :execTool --name=heapDumperPlatform \
  --toolProperty="application.file=../../executable/application/application.out" \
  --toolProperty="heap.filename=/path/to/memory.hex" \
  --toolProperty="additional.application.files=" \
  --console plain
```

You can find the list of available options below:

Name	Description	Default
`application.file`	Specify the full path of the Executable file, a full linked ELF file.	Not set
`additional.application.filenames`	Specify the full path of Feature files with debug information (`.fodbg` files).	Not set
`heap.filename`	Specify the full path of heap memory dump, in Intel Hex format.	Not set
`additional.memory.filenames`	Specify the full path of additional memory files in Intel Hex format (Installed Feature areas, Dynamic Features table, ...).	Not set
`output.name`	Name of the extracted Heap dump file.	`application.heap`

Heap Viewer

To open the Heap Viewer tool, 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.

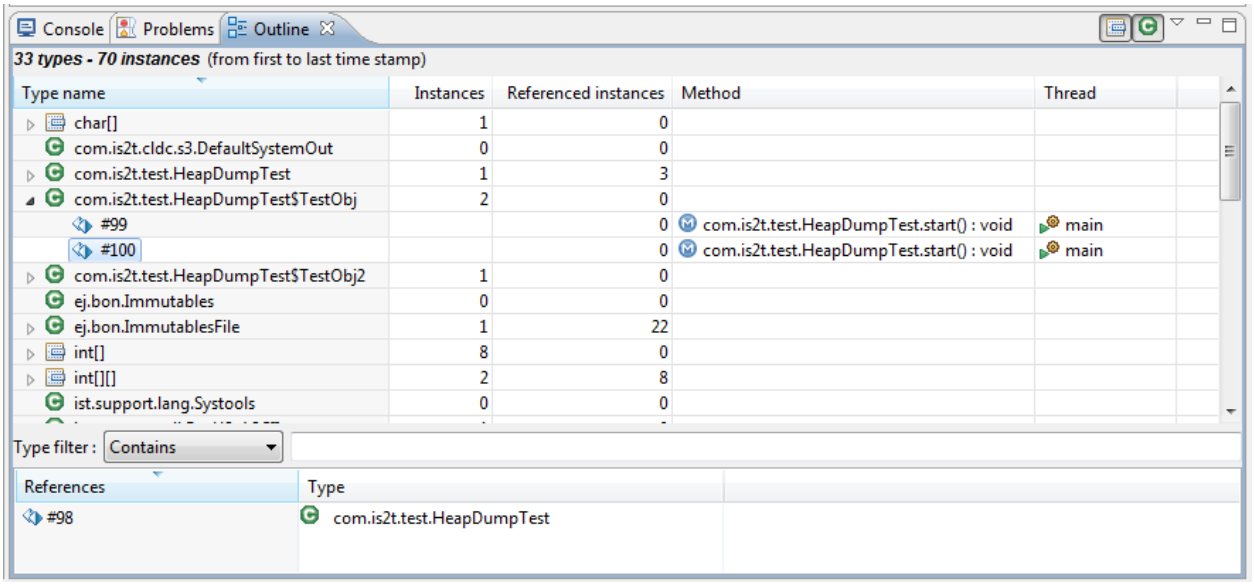


Fig. 41: Outline View

[1] Although this is an Eclipse editor, it is not possible to edit the contents of the heap dump.

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.

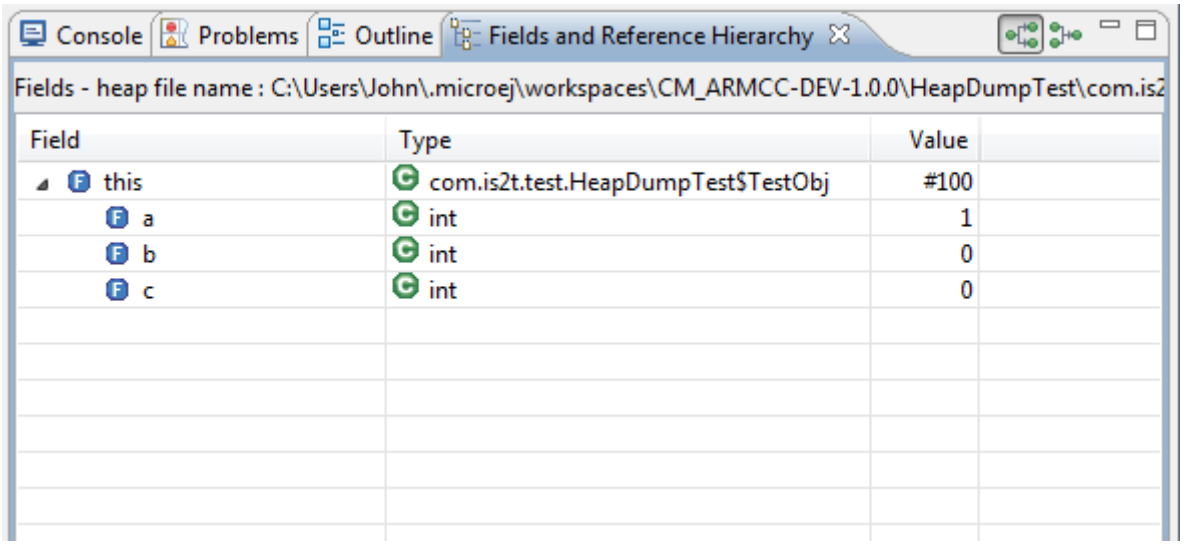


Fig. 42: Instance Browser View - Fields mode

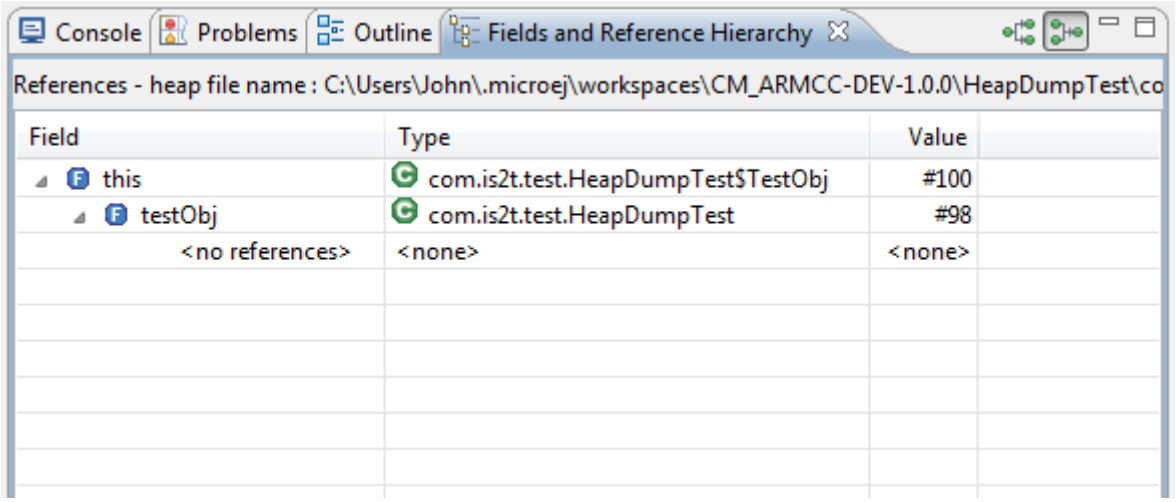


Fig. 43: Instance Browser View - References mode

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.

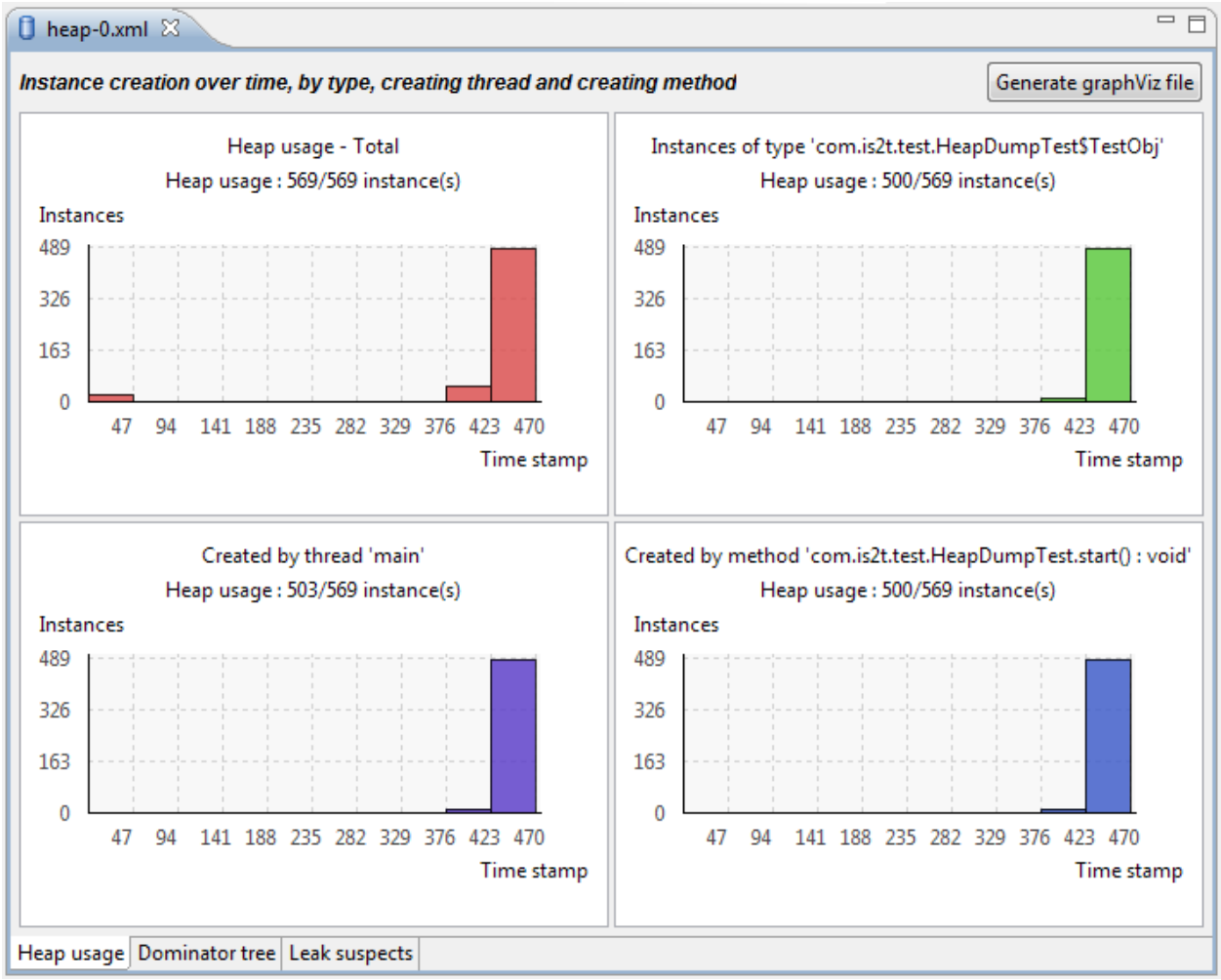


Fig. 44: Heap Viewer - Heap Usage Tab

Clicking on the graph area in a chart restricts the Outline view to just the types and instances that were created during the selected time period. Clicking on a chart but outside of the graph area restores the Outline view to

showing all types and instances[2].

The button Generate graphViz file in the top-right corner of the Heap Usage page generates a file compatible with graphviz (www.graphviz.org).

The section *Heap Usage Monitoring* shows how to compute the maximum heap usage.

Dominator Tree Tab

The Dominator tree page of the Heap Viewer allows the user to browse the instance reference tree which contains the greatest number of instances. This can be useful when investigating a memory leak because this tree is likely to contain the instances that should have been garbage collected.

The page contains two tree viewers. The top viewer shows the instances that make up the tree, starting with the root. The left column shows the ids of the instances – initially just the root instance is shown. The Shallow instances column shows the number of instances directly referenced by the instance, and the Referenced instances column shows the total number of instances below this point in the tree (all descendants).

The bottom viewer groups the instances that make up the tree either according to their type, the thread that created them, or the method that created them.

Double-clicking an instance in either viewer opens the Instance Browser view (if not already open) and shows details of the instance in that view.

[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.

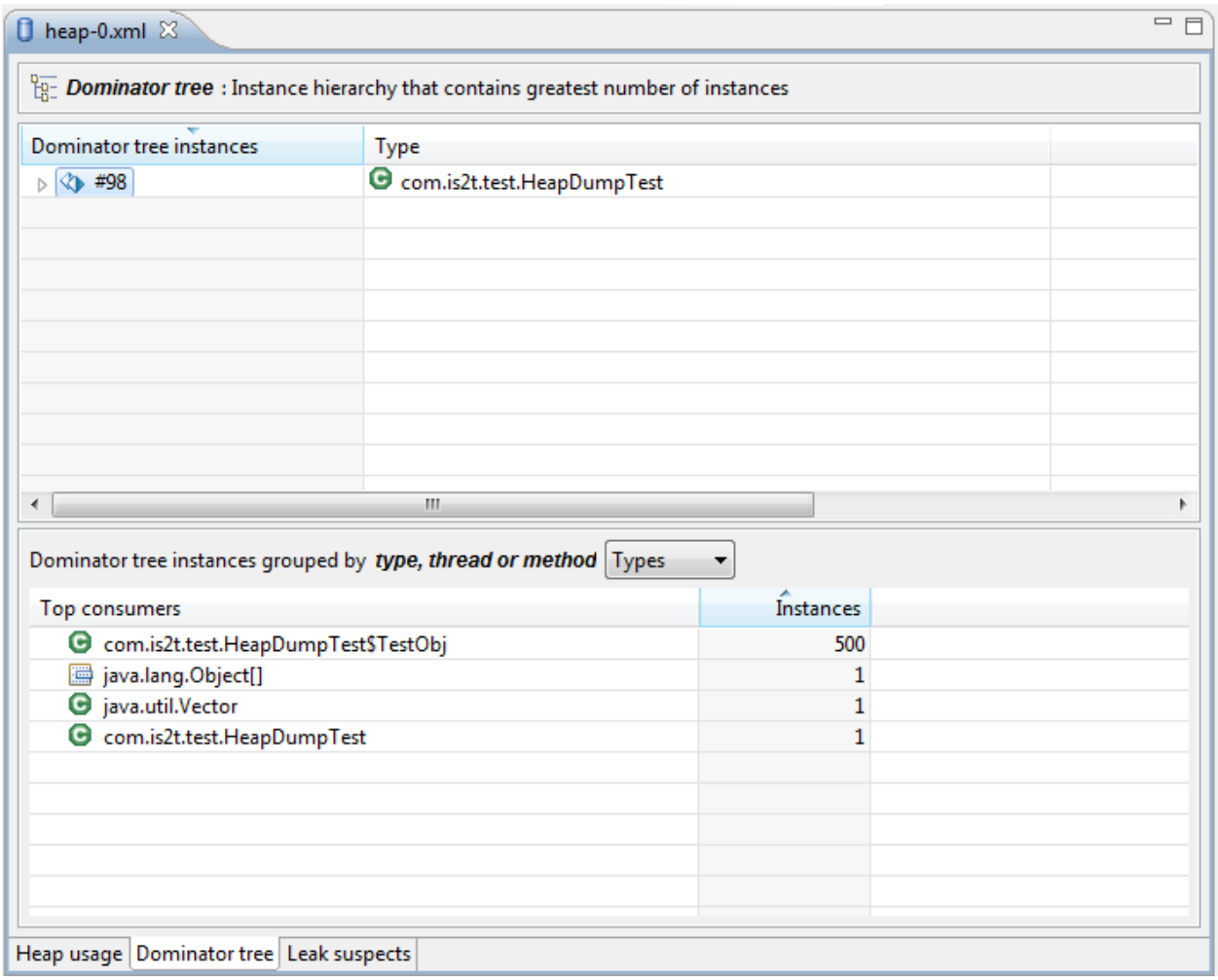


Fig. 45: Heap Viewer - Dominator Tree Tab

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.

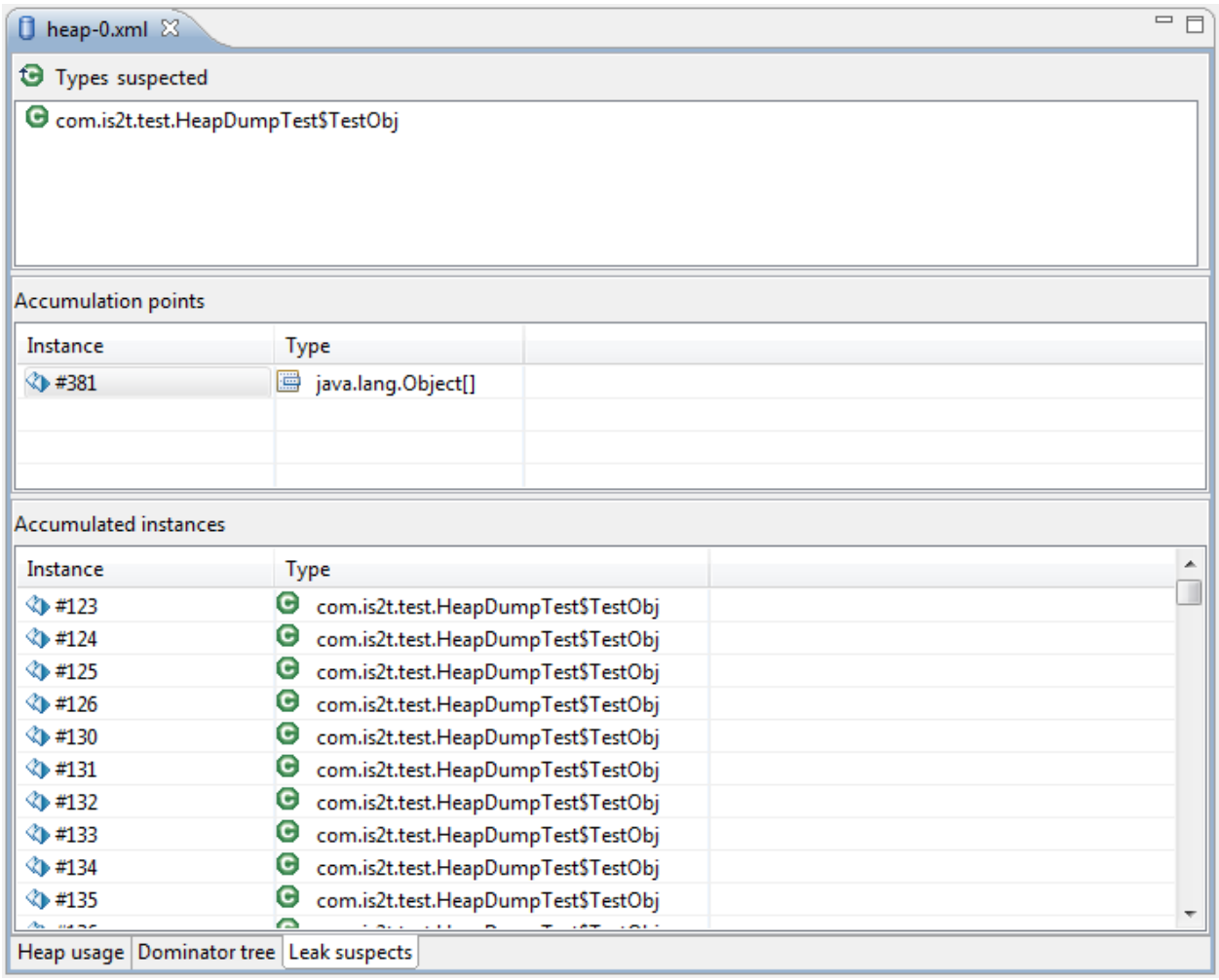


Fig. 46: Heap Viewer - Leak Suspects Tab

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.

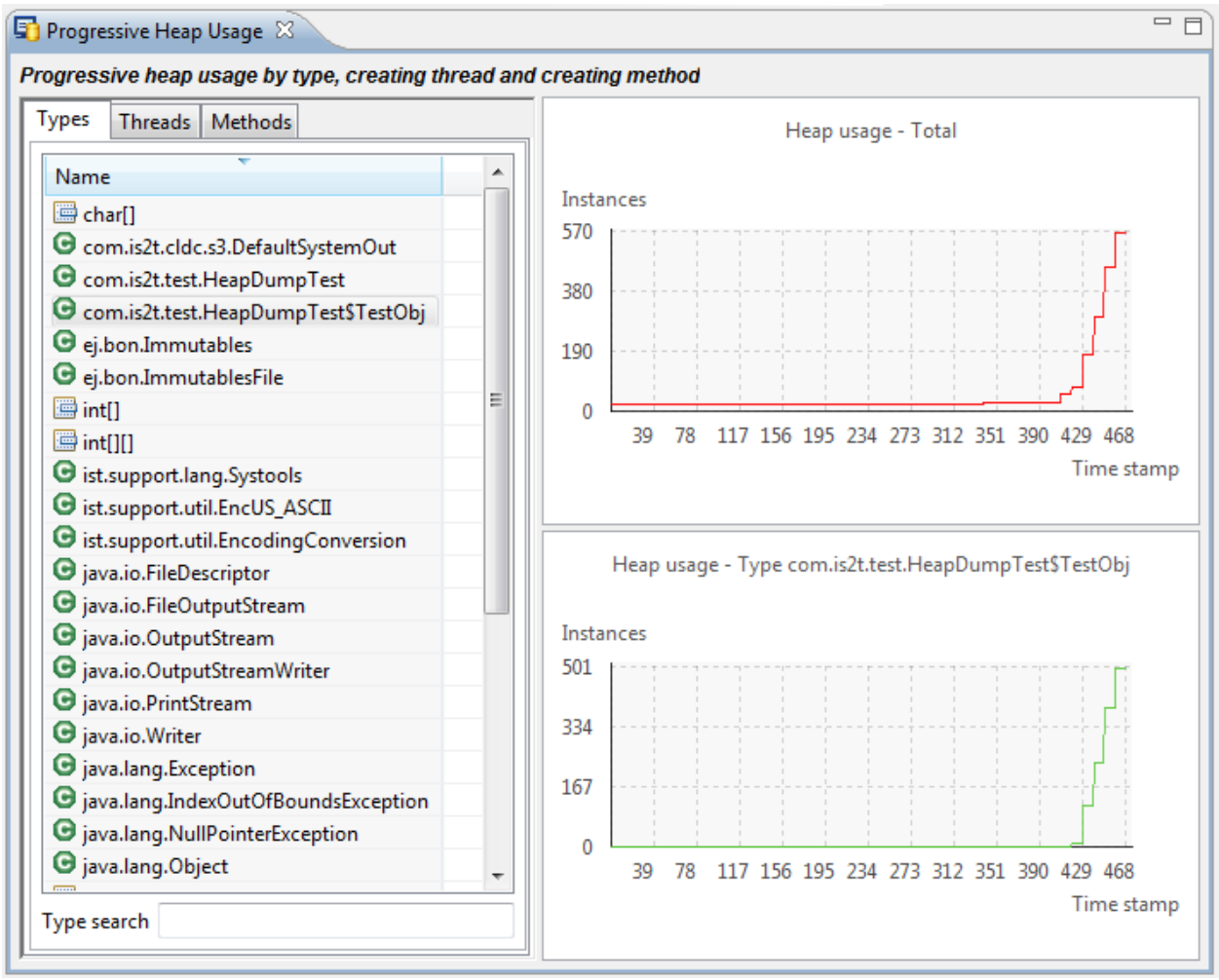


Fig. 47: Progressive Heap Usage

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.

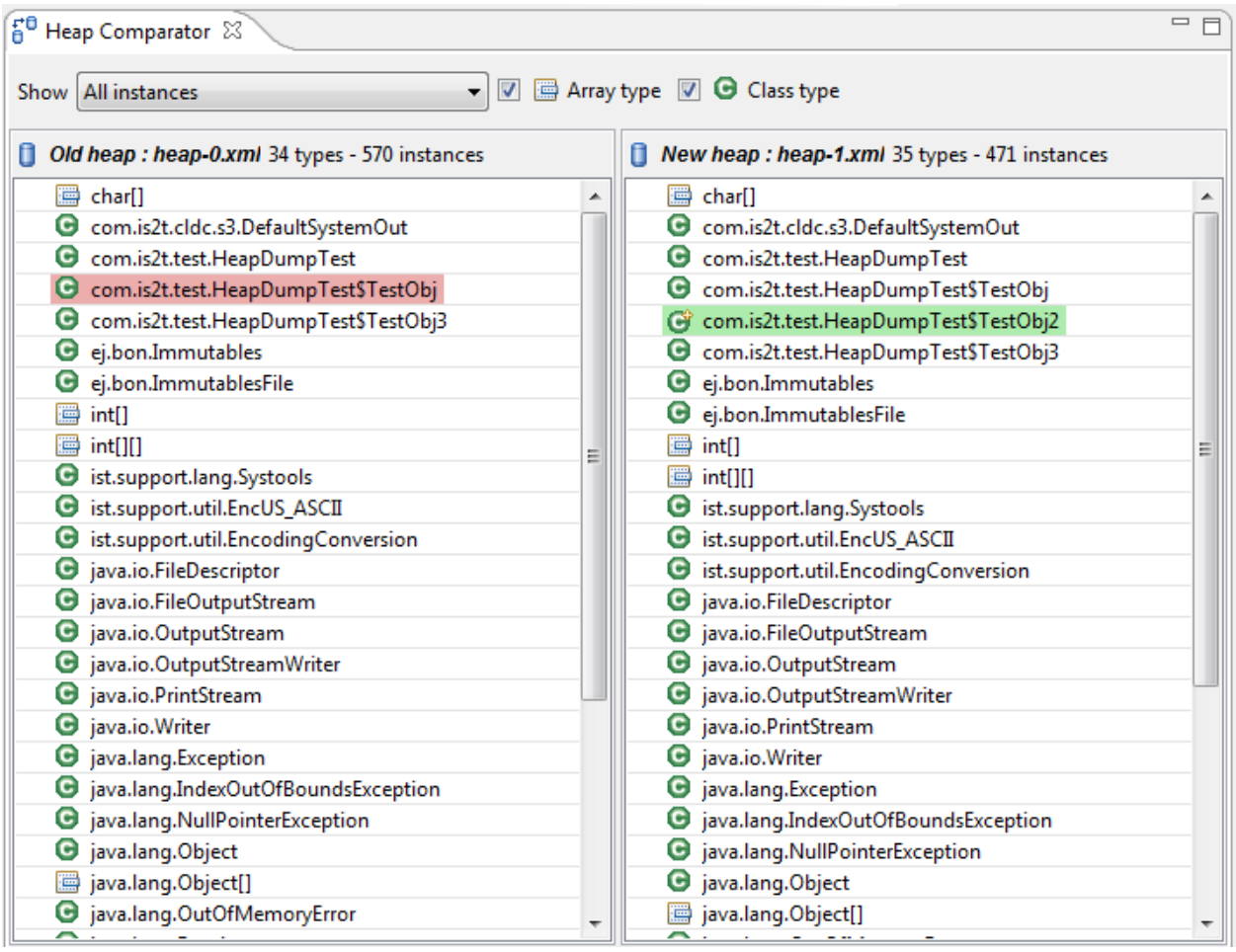


Fig. 48: Compare Heap Dumps

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.

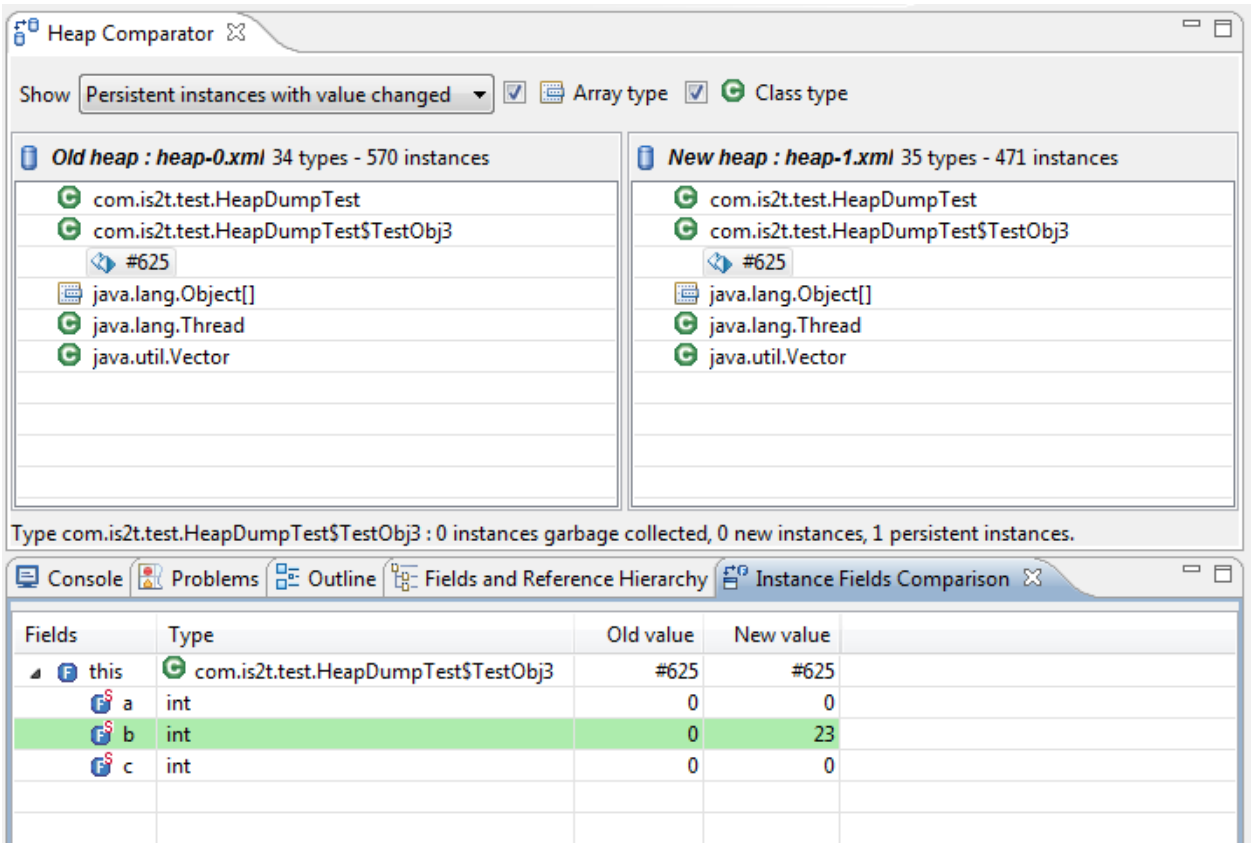


Fig. 49: Instance Fields Comparison view

4.17.5 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

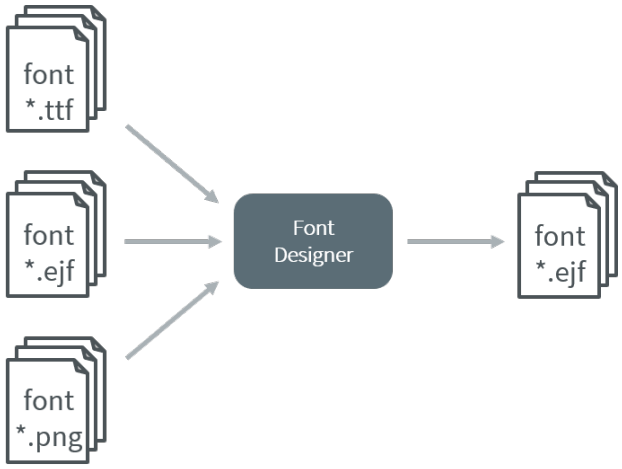


Fig. 50: Font Generation

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.

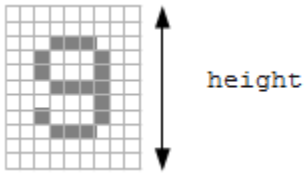


Fig. 51: Font Height

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 '}'.

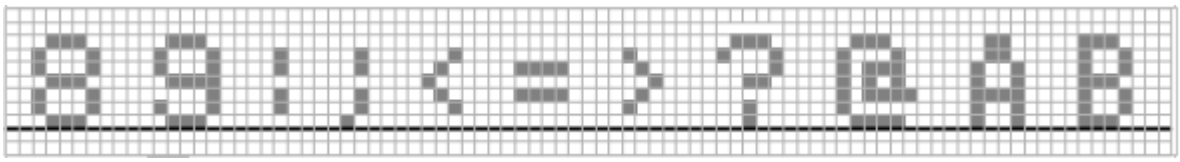


Fig. 52: The Baseline

Space Character

The Space character (0x20) is a specific character because it has no filled pixels. From the Main Properties Menu you can fix the space character size in pixels.

Note: When the font is monospace, the space size is equal to the font width.

Styles

Font Designer allows creating a font file that holds several combinations of built-in styles (styles hardcoded in pixels map) and runtime styles (styles rendered dynamically at runtime). However, since MicroUI 3, a MicroUI font holds only one style: `PLAIN`, `BOLD`, `ITALIC` or `BOLD + ITALIC`.

Font Designer features three drop-downs, one for each of `BOLD`, `ITALIC`, and `UNDERLINED`. Each drop-down has three options: `None`, `Built-in` and `Dynamic`. The font options must be adjusted to be compatible with MicroUI 3:

- The style option `Dynamic` (that targets the runtime style) is forbidden; select `None` instead.
- The syle `UNDERLINED` is forbidden; select `None` instead.

The styles options `Built-in` tag the font as bold, italic, or bold and italic. This style can be retrieved by the MicroEJ Application thanks the methods Font.isBold() and Font.isItalic(). Adjust the styles options according to the font:

- The font is a *plain* font: select `None` option for each style.
- The font is a *bold* font: select `Built-in` for the style *bold* and `None` for the other styles.
- The font is an *italic* font: select `Built-in` for the style *italic* and `None` for the other styles.
- The font is a *bold* and *italic* font: select `Built-in` for the styles *bold* and *italic* and `None` for `UNDERLINED`.

Warning: When a font holds a dynamic style or when the style `UNDERLINED` is not `None`, an error at MicroEJ application compile-time is thrown (incompatible font file).

Identifiers

A number of identifiers can be attached to 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.

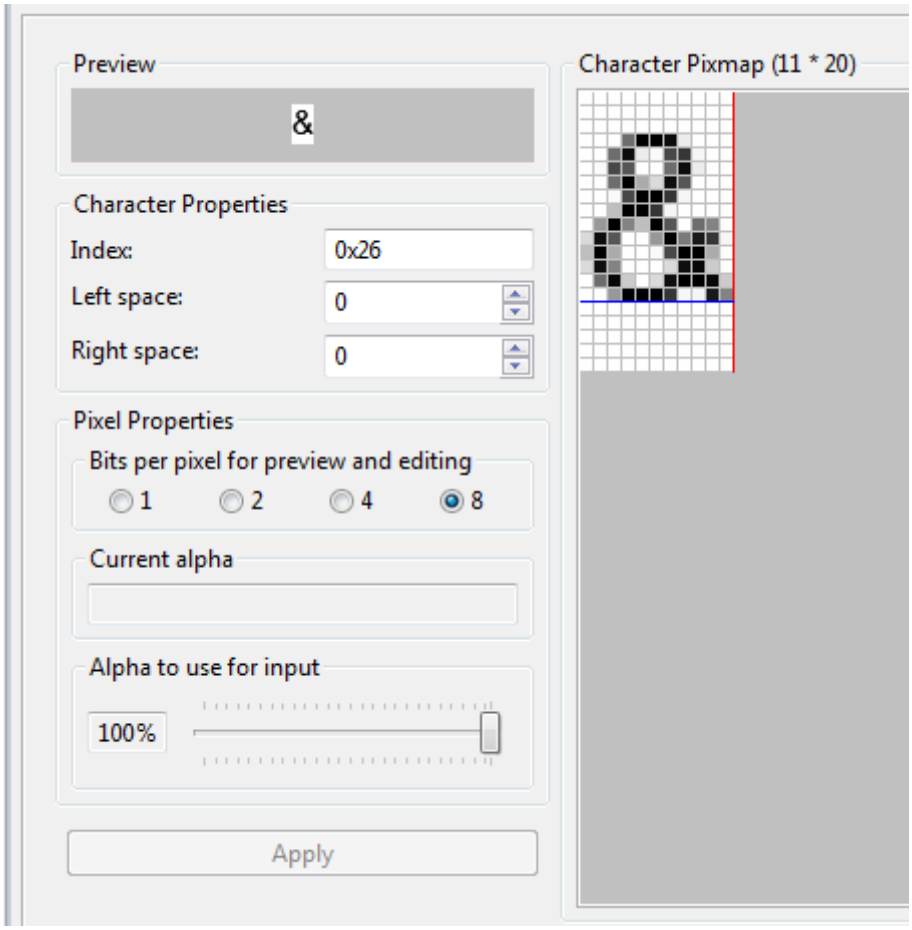


Fig. 53: Character Editor

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.

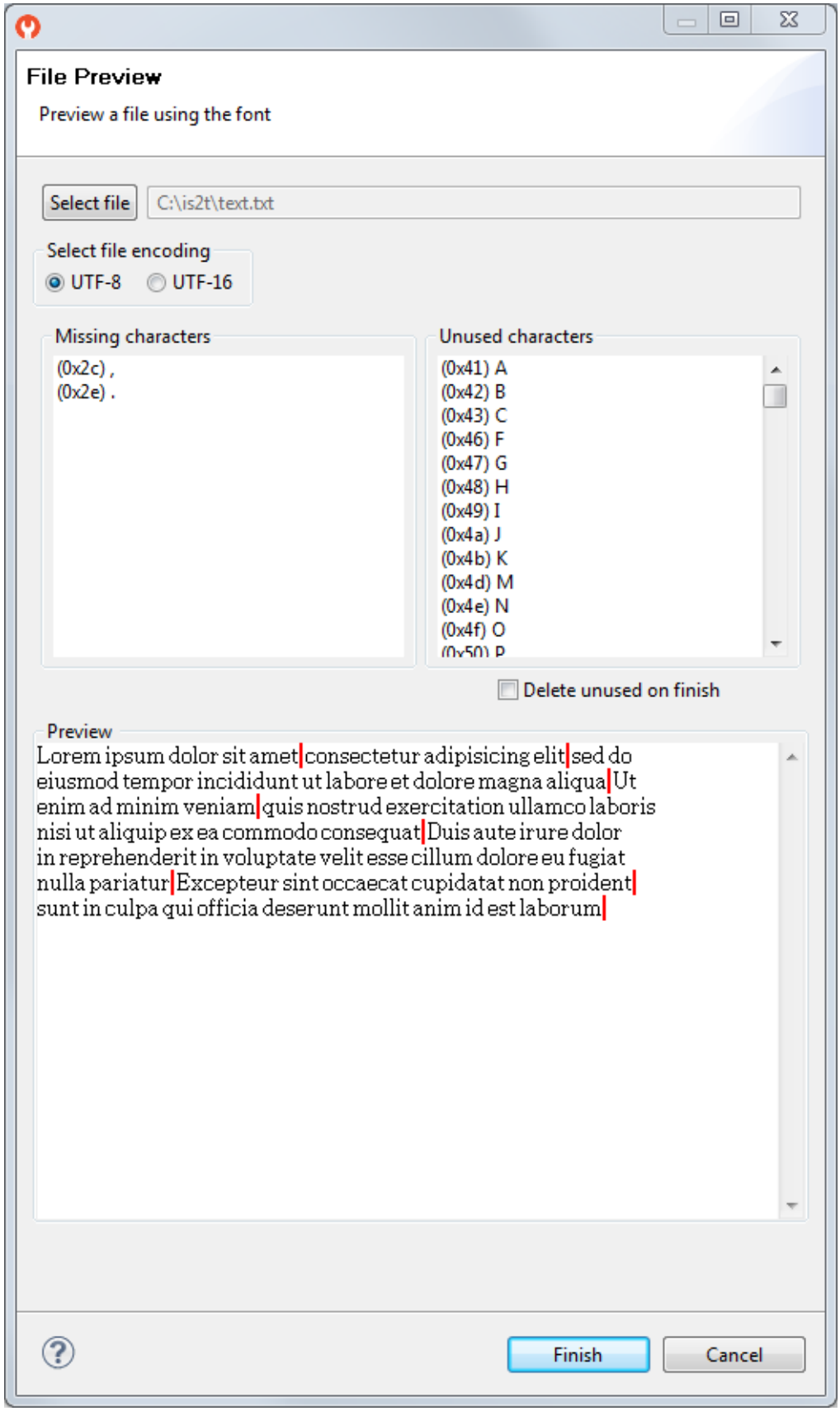


Fig. 54: Font Preview

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.

4.17.6 Local Deployment Socket

Principle

The Local Deployment Socket is a tool that allows to transfer a Sandboxed Application on the device over a network connection.

Functional Description

The Local Deployment Socket builds the Sandboxed Application `.fo` and upload it on the device identified by its IP address. On the device, it is the job of the Kernel Application that receives the `.fo` to install and to run the Sandboxed Application.

Use

```
gradle :execTool --name=localDeploymentSocket \
  --toolProperty="application.main.class=com.mycompany.MyFeature" \
  --toolProperty="board.server.host=10.0.0.171" \
  --toolProperty="board.server.port=4000" \
  --toolProperty="board.timeout=120000" \
  --toolProperty="use.storage=true"
```

Options

Option: Application Feature Class

Option Name: `application.main.class`

Required?: Yes

Description:

Specify the entry-point as the full qualified name of the Feature Application to deploy.

Option: Server Host

Option Name: `board.server.host`

Required?: Yes

Description:

The IP of the target device.

Option: Server Port

Option Name: `board.server.port`

Required?: Yes

Description:

The TCP port on which the Kernel listens (usually 4000).

Option: Timeout

Option Name: `board.timeout`

Required?: Yes

Description:

If there is no activity within the defined timeout period (in seconds), the tool will disconnect from the device.

Option: Use Storage

Option Name: `use.storage`

Required?: Yes

Description:

A boolean describing whether to use the storage to store the Application or not. Refer to the Kernel documentation to find out the correct setting.

4.17.7 Null Analysis

NullPointerException thrown at runtime is one of the most common causes for failure of Java programs. All modern IDEs provide a Null Analysis tool which can detect such programming errors (misuse of potential `null` Java values) at compile-time.

Principle

The Null Analysis tool is based on Java annotations. Each Java field, method parameter and method return value must be marked to indicate whether it can be `null` or not.

Once the Java code is annotated, the IDE must be configured to enable Null Analysis detection.

Java Code Annotation

MicroEJ defines its own annotations:

- @NonNullByDefault: Indicates that all fields, method return values or parameters can never be null in the annotated package or type. This rule can be overridden on each element by using the `@Nullable` annotation.
- @Nullable: Indicates that a field, local variable, method return value or parameter can be null.
- @NonNull: Indicates that a field, local variable, method return value or parameter can never be null.

MicroEJ recommends to annotate the Java code as follows:

- In each Java package, create a `package-info.java` file and annotate the Java package with `@NonNullByDefault` if you use Eclipse or with your custom annotation if you use Android Studio or IntelliJ IDEA (see next section on IDEs configuration). This is a common good practice to deal with non `null` elements by default to avoid undesired NullPointerException. It enforces the behavior which is already widely outlined in Java coding rules.

```
@ej.annotation.NonNullByDefault
package com.mycompany;
```

- In each Java type, annotate all fields, methods return values and parameters that can be null with `@Nullable`. Usually, this information is already available as textual information in the field or method Javadoc comment. The following example of code shows where annotations must be placed:

```
@Nullable
public Object thisFieldCanBeNull;

@Nullable
public Object thisMethodCanReturnNull() {
```

(continues on next page)

(continued from previous page)

```
  return null;
}

public void thisMethodParameterCanBeNull(@Nullable Object param) {

}
```

IDE Configuration

Requirements

The project must depend at least on the version `1.3.3` of the `ej.api:edc` module:

```
dependencies {
  implementation("ej.api:edc:1.3.3")
}
```

Project configuration

Android Studio / IntelliJ IDEA

Eclipse

To enable the Null Analysis tool in Android Studio and IntelliJ IDEA, refer to the official documentation on Configure nullability annotations.

Both IDEs support custom annotations for `Nullable` and `NotNull` annotations, but not for `NonNullByDefault`. Here are the solutions to be able to define all fields, methods return values and parameters of a whole class or package as non null by default:

- create a custom annotation in your project using the `@TypeQualifierDefault` annotation, for example `NonNullByDefault`:

```
import javax.annotation.Nonnull;
import javax.annotation.meta.TypeQualifierDefault;
import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

/**
* This annotation can be applied to a package, class or method to indicate that the␣
↪class fields,
* method return types and parameters in that element are not null by default.
*/
@Documented
@Nonnull
@TypeQualifierDefault(
        {
                ElementType.ANNOTATION_TYPE,
                ElementType.CONSTRUCTOR,
```

(continues on next page)

(continued from previous page)

```
                ElementType.FIELD,
                ElementType.LOCAL_VARIABLE,
                ElementType.METHOD,
                ElementType.PACKAGE,
                ElementType.PARAMETER,
                ElementType.TYPE
        })
@Retention(RetentionPolicy.RUNTIME)
public @interface NonNullByDefault {
}
```

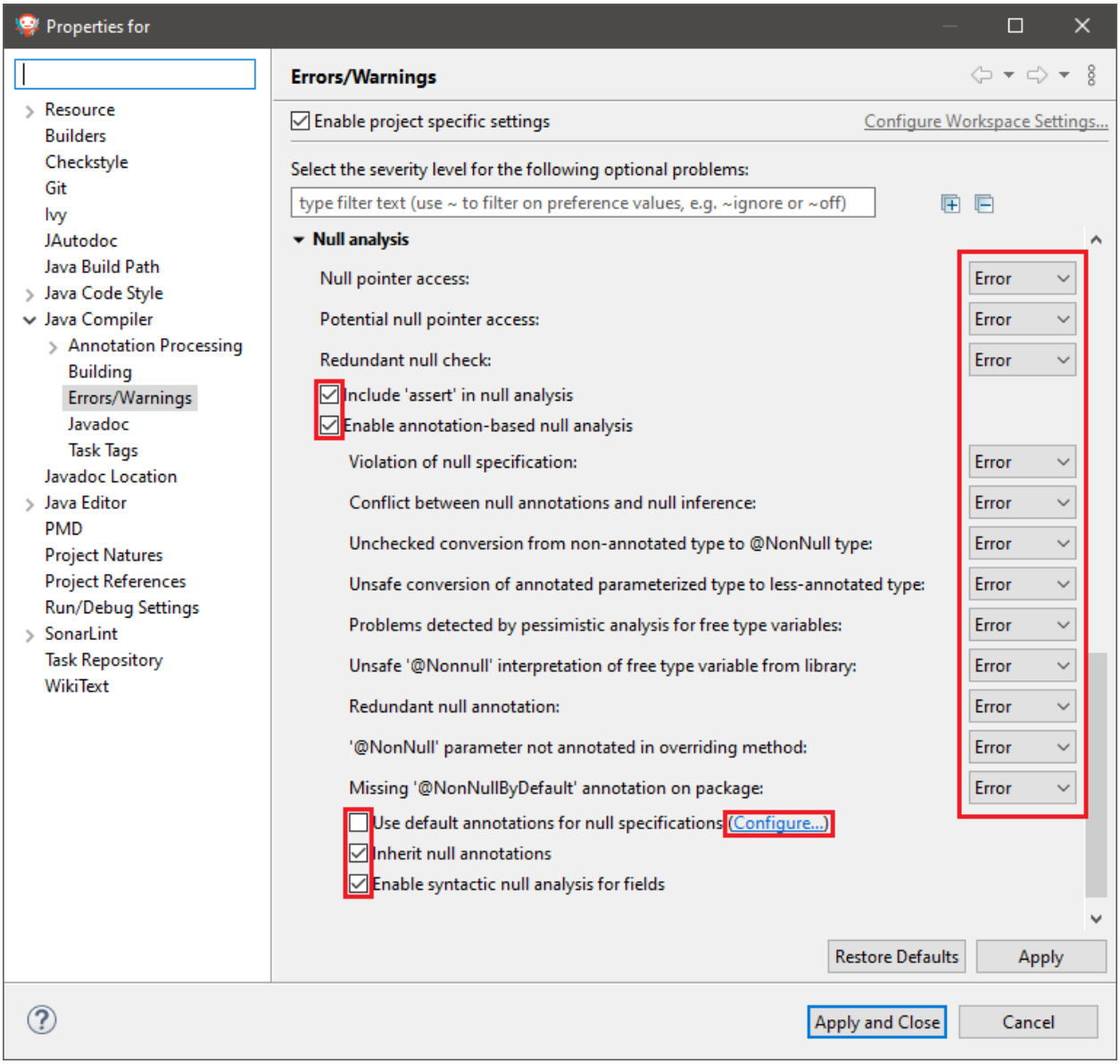
This requires to add the following dependency in your project:

```
compileOnly("com.google.code.findbugs:jsr305:3.0.2")
```

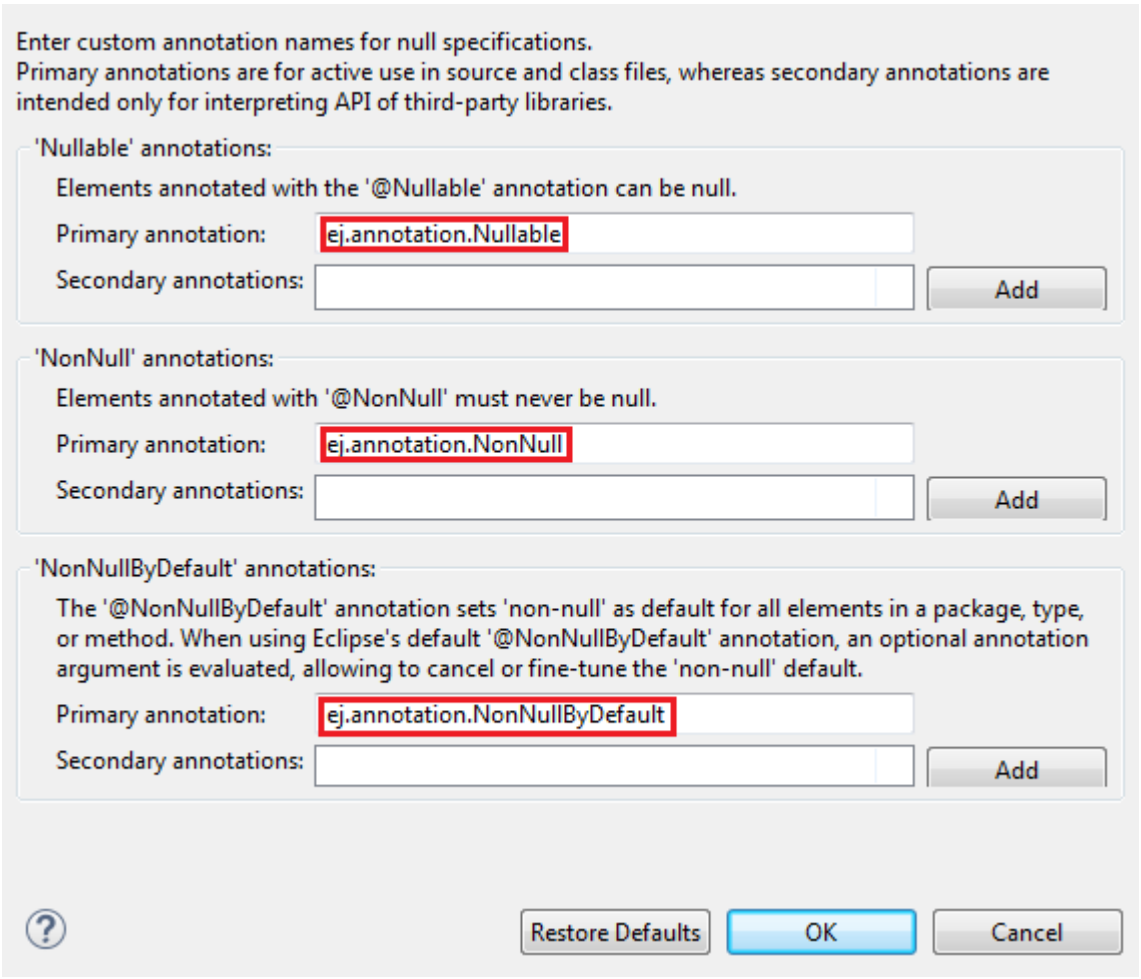
- add the `@NonNull` annotation explicitly on each field, method return value or parameter.

To enable the Null Analysis tool in Eclipse, a project must be configured as follows:

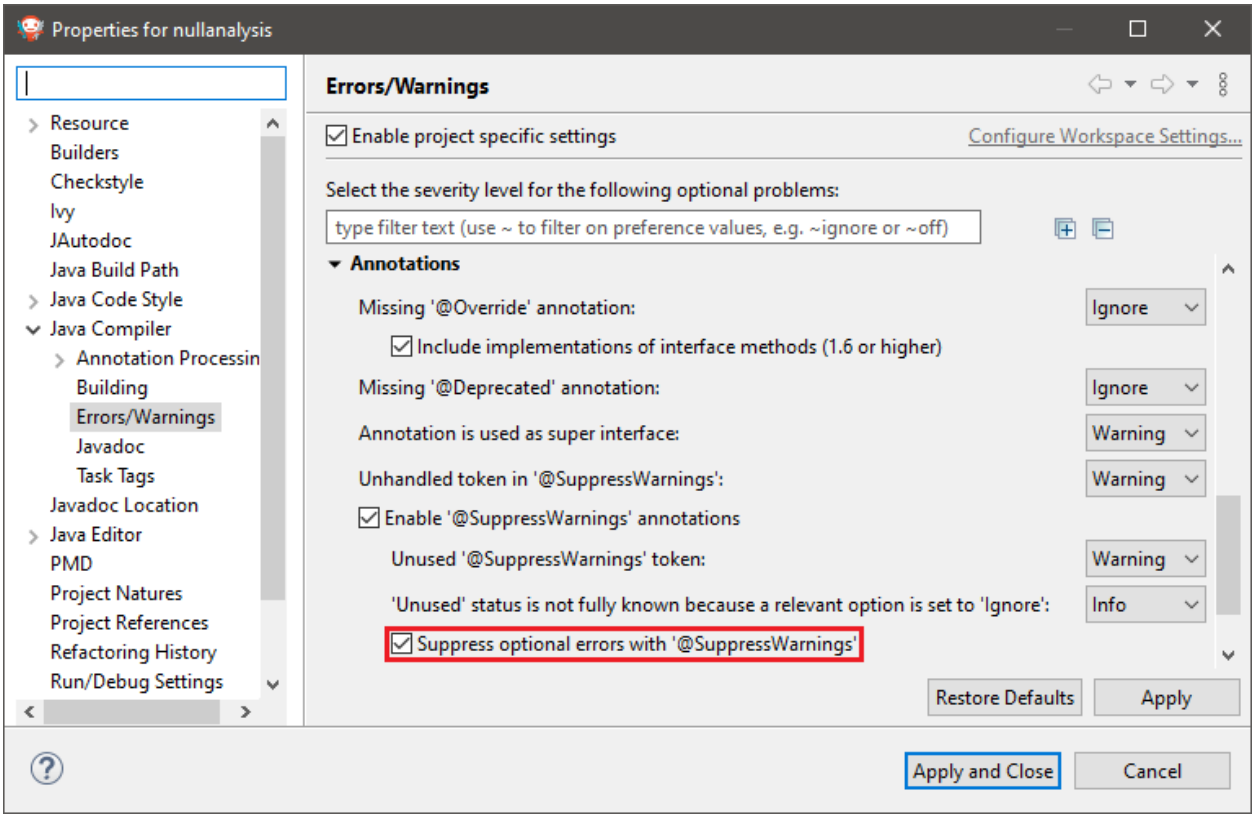
- In the Package Explorer, right-click on the module project and select `Properties` ,
- Navigate to `Java Compiler` > `Errors/Warnings` ,
- In the `Null analysis` section, configure options as follows:



- Click on the Configure… link to configure MicroEJ annotations:
  - `ej.annotation.Nullable`
  - `ej.annotation.NonNull`
  - `ej.annotation.NonNullByDefault`



- In the `Annotations` section, check `Suppress optional errors with '@SuppressWarnings'` option:



This option allows to fully ignore Null Analysis errors in advanced cases using `@SuppressWarnings("null")` annotation.

If you have multiple projects to configure, you can then copy the content of the `.settings` folder to an other module project.

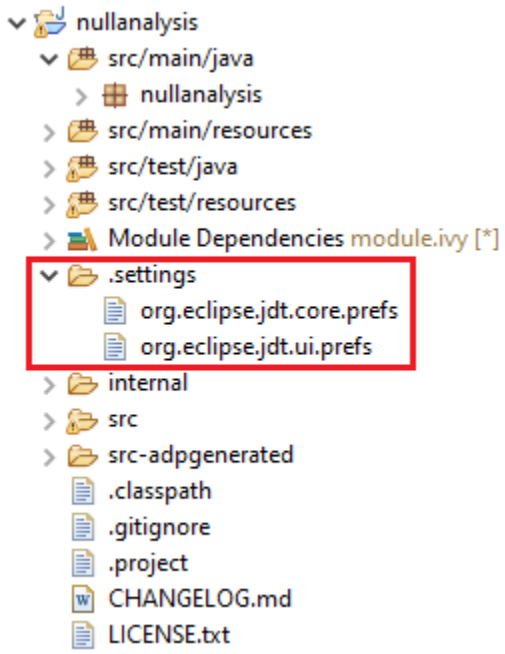


Fig. 55: Null Analysis Settings Folder

Warning: You may lose information if your target module project already has custom parameterization or if it was created with another SDK version. In case of any doubt, please configure the options manually or merge with a text file comparator.

MicroEJ Libraries

Many libraries available on *Central Repository* are annotated with Null Analysis. If you are using a library which is not yet annotated, please contact *our support team*.

For the benefit of Null Analysis, some APIs have been slightly constrained compared to the Javadoc description. Here are some examples to illustrate the philosophy:

- System.getProperty(String key, String def) does not accept a `null` default value, which allows to ensure the returned value is always non `null`.
- Collections of the Java Collections Framework that can hold `null` elements (e.g. HashMap) do not accept `null` elements. This allows APIs to return `null` (e.g. HashMap.get(Object)) only when an element is not contained in the collection.

Implementations are left unchanged and still comply with the Javadoc description whether the Null Analysis is enabled or not. So if these additional constraints are not acceptable for your project, please disable Null Analysis.

4.18 Manage Versioning

The SDK 5 used a specific notation for the snapshot versions. Instead of using the `-SNAPSHOT` prefix (e.g.,``1.0.0-SNAPSHOT``), it used the `-RCxxx` prefix, where `xxx` is the timestamp (e.g., `1.0.0-RC202212021535`).

In order to be able to transition from SDK 5 to SDK 6 smoothly, it is recommended to continue to publish snapshot versions with the `-RCxxx` prefix. This can be done by setting the Gradle module version with the `-RC` prefix. For example:

```
version = "1.0.0-RC"
```

The SDK will automatically append the timestamp to the version to keep the same notation than MMM. This way, SDK 5 projects will be able to fetch modules published by the SDK 6.

Note: You are free to use any version number notation you want, but you have to be aware that SDK 5 projects will not be able to depend on snapshot modules published without the `-RCxxx` prefix.

4.19 Manage Resolution Conflicts

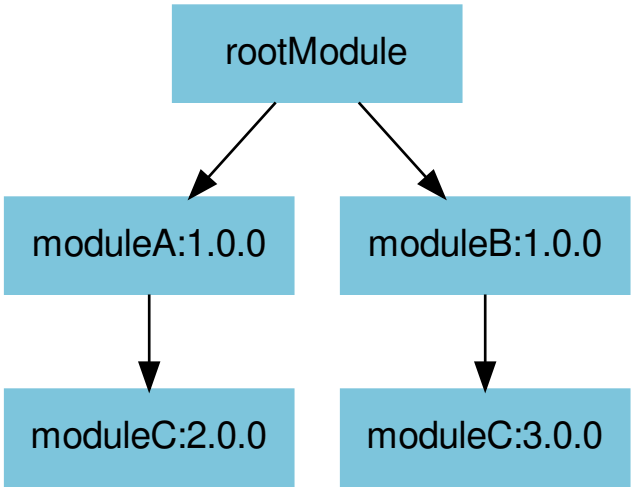
The MicroEJ Gradle plugin adds specific rules for compilation, building, resolving dependencies, versioning, and publishing.

Gradle comes with a powerful dependency manager. One of its job is to resolve the conflicts in the dependency graph, to determine which version should be added to the graph. By default, Gradle selects the highest version amongst all the versions requested for a dependency. There are ways to influence the dependencies resolution, but we believe additional rules should be added to provide a better and safer conflict resolution.

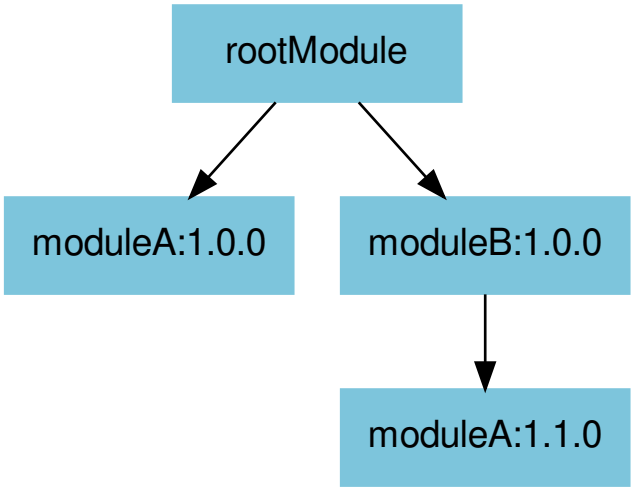
Note: You can learn more on the Gradle conflicts resolution and the way to configure it in the official documentation.

The MicroEJ Gradle plugin adds the 2 following rules:

- The resolution fails when a dependency is requested with 2 incompatible versions in the graph, according to the Semantic Versioning specification. So, it means that if 2 versions do not have the same major version, the build fails. For example, this dependency graph makes the build fail because the `moduleC` dependency is requested in 2 incompatible versions:



- The resolution fails when a direct dependency is resolved with a higher minor version than the one declared. For example, this dependency graph makes the build fails because the `moduleA` dependency is resolved in version `1.1.0` (the highest one), which is higher than the direct declared version (`1.0.0`):



If you want to come back to the Gradle default behavior, these 2 rules can be disabled by setting the

`microejConflictResolutionRulesEnabled` property of the `microej` configuration block to `false` in the project build file:

```
microej {
  microejConflictResolutionRulesEnabled = false
}
```

4.20 Migrate an MMM Project

This page explains how to migrate a project created with the SDK 5 or lower to the SDK 6. It covers the following items:

- Project structure
- Build descriptor file
- Build scripts

4.20.1 Project structure

The structure of an SDK 6 Gradle project is similar to an MMM project. The differences are:

- The `module.ivy` file is replaced by a `build.gradle.kts` file and a `settings.gradle.kts` file (see *Build Descriptor File*).
- The `module.ant` and `override.module.ant` files are removed (see *Build Scripts*).
- The `build` folder located at the root of the project and containing the Application configuration properties is replaced by the `configuration` folder. This change is required since Gradle uses the `build` folder to store the generated files and artifacts (equivalent of the MMM `target~` folder).

Therefore, here are the 2 project structures side by side:

MMM Project	Gradle Project

`|- src`
`|  |- main`
`|  |  |- java`
`|  |  |- resources`
`|  |- test`
`|  |- java`
`|  |- resources`
`|- build`
`|  |- common.properties`
`|- module.ivy`
`|- module.ant`
	`|- src`
	`|  |- main`
	`|  |  |- java`
	`|  |  |- resources`
	`|  |- test`
	`|  |- java`
	`|  |- resources`
	`|- configuration`
	`|  |- common.properties`
	`|- build.gradle.kts`
	`|- settings.gradle.kts`

4.20.2 Build Descriptor File

The `module.ivy` file of the MMM project must be replaced by a `build.gradle.kts` file and a `settings.gradle.kts` file. The `settings.gradle.kts` contains the name of the project, whereas the `build.gradle.kts` file contains all the other information (module type, group, version, ...).

Build Type

The MMM build type defined in the `module.ivy` file with the `ea:build` tag is replaced by a plugin in the `build.gradle.kts` file. For example, here is the block to add at the beginning of the file to migrate a `build-microej-javalib` MMM module:

```
plugins {
    id("com.microej.gradle.addon-library") version "0.15.0"
}
```

The mapping between MMM build types and Gradle plugins is:

MMM Build Type	Gradle Plugin
`build-microej-javalib`	`com.microej.gradle.addon-library`
`build-application`	`com.microej.gradle.application`
`build-firmware-singleapp`	`com.microej.gradle.application`
`build-firmware-multiapp`	`com.microej.gradle.application`
`build-std-javalib`	`com.microej.gradle.j2se-library`

Module Information

The module information defined by the `info` tag in the `module.ivy` file are split in the 2 following descriptor files:

- `settings.gradle.kts`
  - The property `rootProject.name` replaces the `module` attribute.
- `build.gradle.kts`
  - The property `group` replaces the `organisation` attribute.
  - The property `version` replaces the `revision` attribute.

So for example, the following `info` tag:

```
<info organisation="com.mycompany" module="myProject" status="integration" revision="0.1.0">
```

will be converted to:

Listing 1: settings.gradle.kts

```
rootProject.name = "myProject"
```

Listing 2: build.gradle.kts

```
group = "com.mycompany"
version = "0.1.0"
```

Note: Refer to *Manage Versioning* section for more information on the way to define the module version.

Configuration

The configuration of an MMM build is only done with `ea:property` tags in the `module.ivy` file, whereas it can take multiple form in Gradle. You can refer to the *Module Natures* page for a complete list of configurations.

As a first example, the main class is defined in MMM with the property `application.main.class`:

```
<ea:property name="application.main.class" value="com.mycompany.Main"/>
```

whereas it is defined by the `applicationMainClass` property of the `microej` block in Gradle:

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

As a second example, the pattern of the executed tests is defined in MMM with the property `test.run.includes.pattern`:

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

whereas it is defined by the `filter` object of the `test` task in Gradle:

```
testing {
   suites {
     val test by getting(JvmTestSuite::class) {

         ...

         targets {
             all {
                 testTask.configure {
                     filter {
                         includeTestsMatching("MyTest")
                     }
                 }
             }
         }
     }
   }
}
```

Dependencies

The `dependencies` tag in the `module.ivy` file is replaced by the `dependencies` block in the `build.gradle.kts` file. Each dependency is tight to a Gradle configuration. For example, migrating a dependency used at compile time and runtime should use the `implementation` configuration, so the following dependency:

```
<dependency org="ej.api" name="edc" rev="1.3.5" />
```

will be converted to:

```
implementation("ej.api:edc:1.3.5")
```

wheras a dependency used for the tests only should use the `testIplementation` configuration, so the following dependency:

```
<dependency conf="test->*" org="ej.library.test" name="junit" rev="1.7.1"/>
```

will be converted to:

```
testImplementation("ej.library.test:junit:1.7.1")
```

Also note that this will not resolve snapshot builds since versions are explicit in SDK 6, see *this chapter* for more details. To resolve both snapshot and release versions, use `[1.0.0-RC,1.0.0]` instead of `1.0.0`.

Note: If the dependency relates to another module of the same project, you may use a multi-project structure instead (see Multi-Project Build Basics).

Refer to the *Add a Dependency* page to go further on the Gradle dependencies and configurations.

Example

This section gives an example of migration from a `module.ivy` file to a `build.gradle.kts` file and a `settings.gradle.kts` file.

SDK 5 and lower

Listing 3: module.ivy

```
<ivy-module version="2.0" xmlns:ea="http://www.easyant.org" xmlns:m="http://ant.apache.org/
↪ivy/extra" xmlns:ej="https://developer.microej.com" ej:version="2.0.0">
   <info organisation="com.mycompany" module="myProject" status="integration" revision="0.1.0
↪">
      <ea:build organisation="com.is2t.easyant.buildtypes" module="build-application"␣
↪revision="9.2.+">
         <ea:property name="test.run.includes.pattern" value="**/_AllTests_*.class"/>
      </ea:build>
   </info>

   <configurations defaultconfmapping="default->default;provided->provided">
      <conf name="default" visibility="public" description="Runtime dependencies to other␣
↪artifacts"/>
      <conf name="provided" visibility="public" description="Compile-time dependencies to␣
↪APIs provided by the platform"/>
```

(continues on next page)

(continued from previous page)

```
      <conf name="platform" visibility="private" description="Build-time dependency, specify␣
↪the platform to use"/>
      <conf name="documentation" visibility="public" description="Documentation related to␣
↪the artifact (javadoc, PDF)"/>
      <conf name="source" visibility="public" description="Source code"/>
      <conf name="dist" visibility="public" description="Contains extra files like README.md,
↪ licenses"/>
      <conf name="test" visibility="private" description="Dependencies for test execution.␣
↪It is not required for normal use of the application, and is only available for the test␣
↪compilation and execution phases."/>
      <conf name="microej.launch.standalone" visibility="private" description="Dependencies␣
↪for standalone application. It is not required for normal use of the application, and is␣
↪only available when launching the main entry point on a standalone MicroEJ launch."/>
   </configurations>

   <publications>
      <!-- keep this empty if no specific artifact to publish -->
      <!-- must be here in order to avoid all configurations for the default artifact -->
   </publications>

   <dependencies>
      <!--
         Put your custom Runtime Environment dependency here. For example:

         <dependency org="com.company" name="my-runtime-api" rev="1.0.0" conf="provided->
↪runtimeapi" />
      -->
      <!--
         Or put direct dependencies to MicroEJ libraries if your Application is not intended␣
↪to run on a specific custom Runtime Environment.
      -->
      <dependency org="ej.api" name="edc" rev="1.3.5" />
      <dependency org="ej.api" name="kf" rev="1.6.1" />

      <dependency conf="test->*" org="ej.library.test" name="junit" rev="1.7.1"/>

      <dependency org="com.microej.platform.esp32.esp-wrover-kit-v41" name="HDAHT" rev="1.8.0
↪" conf="platform->default" transitive="false"/>
   </dependencies>
</ivy-module>
```

SDK 6

Listing 4: settings.gradle.kts

```
rootProject.name = "myProject"
```

Listing 5: build.gradle.kts

```
plugins {
    id("com.microej.gradle.application") version "0.15.0"
}
```

(continues on next page)

(continued from previous page)

```
group = "com.mycompany"
version = "0.1.0"

dependencies {
   implementation("ej.api:edc:1.3.3")
   implementation("ej.api:kf:1.6.1")

   testImplementation("ej.library.test:junit:1.7.1")

   microejVee("com.microej.platform.esp32.esp-wrover-kit-v41:HDAHT:1.8.2")
}
```

4.20.3 Build Scripts

MMM supports the use of the `module.ant` and `override.module.ant` to customize the build process. These files are not supported anymore with Gradle. Instead, since Gradle build files are code, customizations can be applied directly in the build files.

As an example, defining a property conditionnaly is done as follows in a `module.ant` file:

```
<target name="my-project:define-properties" extensionOf="compile">
   <condition property="myProperty" value="myValue">
      <not><equals arg1="${anotherProperty}" arg2="anotherValue"/></not>
   </condition>
</target>
```

and as follows in a `build.gradle.kts` file:

```
var myProperty = ""
tasks.register("defineProperties") {
   if(project.properties["anotherProperty"] == "anotherValue") {
      myProperty = "myValue"
   }
}

tasks.compileJava {
   dependsOn("defineProperties")
}
```

4.21 Module Natures

This page describes the most common module natures as follows:

- **Plugin Name**: the build type name, derived from the module nature name: `com.microej.gradle.[NATURE_NAME]`.
- **Documentation**: a link to the documentation.
- **Tasks**: tasks available from the module nature, with the graph of their relationships.

- **Configuration**: properties that can be defined to configure the module. Properties are defined inside the `microej` block of the `build.gradle.kts` file.

4.21.1 Add-On Library

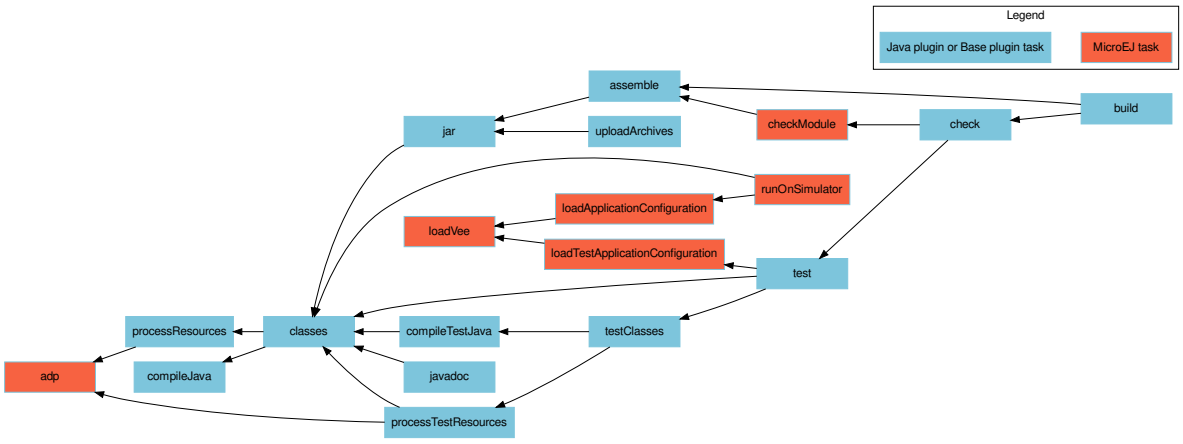
Plugin Name: `com.microej.gradle.addon-library`

Documentation: *Libraries*

Tasks:

This plugin adds the following tasks to your project:

- tasks of the Gradle Java plugin
- *adp*
- *loadVee*
- *loadApplicationConfiguration*
- *runOnSimulator*
- *loadTestApplicationConfiguration*
- *checkModule*



Configuration:

This module nature inherits from the configuration of all its tasks.

4.21.2 Application

Plugin Name: `com.microej.gradle.application`

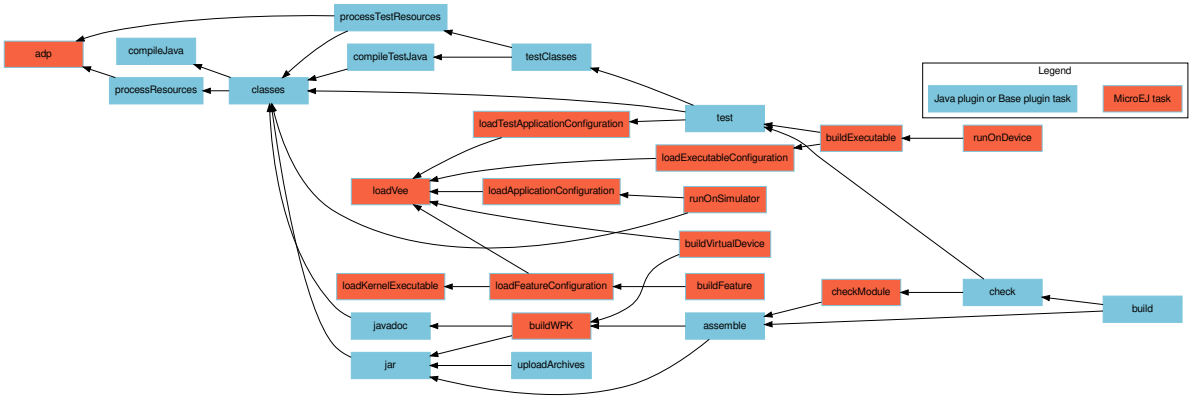
Documentation: *Standalone Application*

Tasks:

This plugin adds the following tasks to your project:

- tasks of the Gradle Java plugin
- *adp*

- *loadVee*
- *loadApplicationConfiguration*
- *runOnSimulator*
- *loadTestApplicationConfiguration*
- *checkModule*
- *loadExecutableConfiguration*
- *buildExecutable*
- *buildWPK*
- *buildVirtualDevice*
- *loadKernelExecutable*
- *loadFeatureConfiguration*
- *buildFeature*
- *runOnDevice*



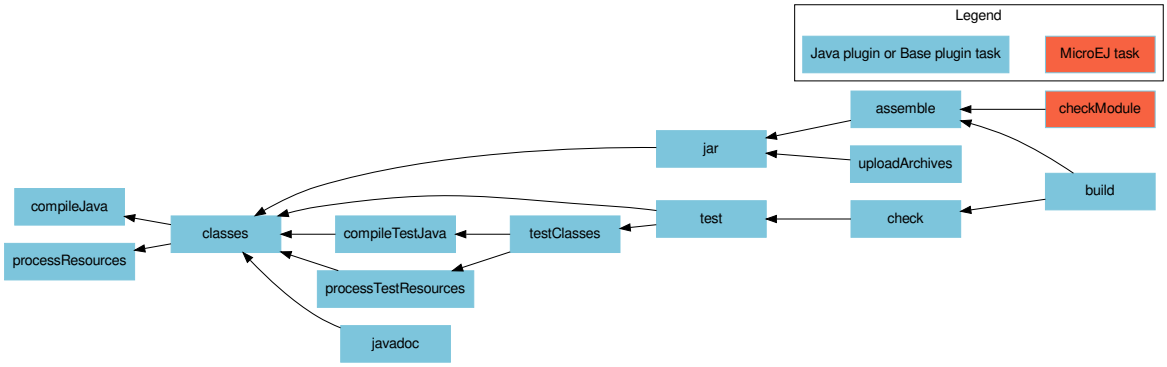
4.21.3 J2SE Library

Plugin Name: `com.microej.gradle.j2se-library`

Tasks:

This plugin adds the following tasks to your project:

- tasks of the Gradle Java plugin
- *checkModule*



Configuration:

This module nature inherits from the configuration of all its tasks.

4.21.4 Tasks

This page describes the module nature tasks as follows:

- **Description**: description and link to the related documentation.
- **Module Natures**: list of *Module Natures* using this task.
- **Configuration**: properties that can be defined to configure the task.

adp

Description: Executes the Addon Processors.

Inputs:

- The project directory

Outputs:

- The directory for each ADP output type (`build/adp/all/main/java` , `build/adp/all/main/resources` , `build/adp/all/test/java` , `build/adp/all/test/resources`)

Module Natures:

This task is used by the following module natures:

- *Add-On Library*
- *Application*

loadVee

Description: Loads the VEE.

Inputs:

- The list of VEE archive files or folders.

Outputs:

- The directory where the VEE is copied/extracted (`build/vee`)

Module Natures:

This task is used by the following module natures:

- *Add-On Library*
- *Application*

loadApplicationConfiguration

Description: Loads the configuration for the Application to execute.

Inputs:

- The extracted VEE Port folder
- The project classpath which contains the MicroEJ dependent application classes and resources
- The Full Qualified Name of the Application main class
- The folder containing the application configuration (`configuration`)
- The System properties
- The debug mode
- The debug port

Outputs:

- The configuration file with all the properties set to launch the application (`build/properties/target.properties`)

Module Natures:

This task is used by the following module natures:

- *Add-On Library*
- *Application*

Configuration:

This task provides the following properties that can be defined in the `microej` extension:

Name	Description	Default
`applicationMainClass`	Full Qualified Name of the main class of the application. This option is required.	Not set

For example:

```
microej {
  applicationMainClass = "com.company.Main"
}
```

runOnSimulator

Description: Executes the Application with the Simulator.

Inputs:

- The extracted VEE Port folder
- The configuration file with all the properties set to launch the application (`build/properties/target.properties`)

Module Natures:

This task is used by the following module natures:

- *Add-On Library*
- *Application*

loadTestApplicationConfiguration

Description: Loads the configuration for the Test Application to execute.

Inputs:

- The extracted VEE Port folder

Outputs:

- The directory containing the configuration file with all the properties set to launch the test application (`build/testsuite/properties/`)

Module Natures:

This task is used by the following module natures:

- *Add-On Library*
- *Application*

checkModule

Description: Checks the compliance of the module.

Inputs:

- The list of the checkers to execute, separated by comas. If not set, all the checkers are executed.
- The list of the checkers to skip, separated by comas.

Module Natures:

This task is used by the following module natures:

- *Add-On Library*
- *Application*

Configuration:

This task is not bound by default on any lifecycle task, which means that it should be called explicitly if it must be executed.

This task provides the following properties that can be defined in the `microej` extension:

Name	Description	Default
`checkers`	Comma-separated list of the names of the checkers to execute. An empty list means that all checkers are executed.	`""`
`skippedCheckers`	Comma-separated list of the names of the checkers to exclude. Only one property of `checkers` and `skippedCheckers` can be defined.	`""`

For example:

```
microej {
  checkers = "readme,license"
}
```

loadExecutableConfiguration

Description: Loads the configuration to build the Executable of an Application.

Inputs:

- The extracted VEE Port folder
- The project classpath which contains the MicroEJ dependent application classes and resources
- The Full Qualified Name of the Application main class
- The folder containing the application configuration (`configuration`)

Outputs:

- The configuration file with all the properties set to launch the build of the Executable (`build/properties/target.properties`)

Module Natures:

This task is used by the following module natures:

- *Application*

buildExecutable

Description: Builds the Executable of an Application.

Inputs:

- The extracted VEE Port folder
- The configuration file with all the properties set to launch the build of the Executable (`build/properties/target.properties`)
- The project build classpath

Outputs:

- The directory in which the Executable file and the build files are generated (`build/executable/application`)

- The Zip file containing the generated build files (`build/executable/buildFiles.zip`)

Module Natures:

This task is used by the following module natures:

- *Application*

buildWPK

Description: Builds the WPK of the Application.

Inputs:

- The Application name
- The Application version
- The Full Qualified Name of the Application main class
- The Application JAR file
- The Application Javadoc
- The Jar files of the Application classpath
- The folder containing the application configuration (`configuration`)

Outputs:

- The WPK of the Application (`build/libs/<application_name>.wpk`)

Module Natures:

This task is used by the following module natures:

- *Application*

buildVirtualDevice

Inputs:

- The extracted VEE Port folder
- The WPK of the Application
- The project build classpath
- The WPK of the Applications that must be pre-installed in the Virtual Device

Outputs:

- The Zip file of the Virtual Device (`build/libs/<application_name>-virtualDevice.zip`)

Description: Build the Virtual Device of an Application.

Module Natures:

This task is used by the following module natures:

- *Application*

loadKernelExecutable

Description: Loads the Kernel Executable file.

Inputs:

- The list of Kernel Executable files.

Outputs:

- The loaded Kernel Executable file is copied (`build/kernelExecutable/kernel.out`)

Module Natures:

This task is used by the following module natures:

- *Application*

loadFeatureConfiguration

Description: Loads the configuration to build the Feature file of an Application.

Inputs:

- The Kernel Virtual Device
- The folder containing the Kernel Executable file (`build/kernelExecutable`)
- The project classpath
- The path of the folder where the Feature file must be generated (`build/feature`)

Outputs:

- The configuration file with all the properties set to launch the build of the Feature file (`build/properties/target.properties`)

Module Natures:

This task is used by the following module natures:

- *Application*

buildFeature

Description: Build the Feature file of an Application.

Inputs:

- The Kernel Virtual Device
- The folder containing the Kernel Executable file (`build/kernelExecutable`)
- The project classpath

Outputs:

- The folder in which the Feature file is generated (`build/feature`)

Module Natures:

This task is used by the following module natures:

- *Application*

runOnDevice

Description: Runs the Executable on a Device.

Inputs:

- The extracted VEE Port folder
- The folder containing the Executable file (`build/executable/application`)
- The configuration file with all the properties set to launch the build of the Executable (`build/properties/target.properties`)

Module Natures:

This task is used by the following module natures:

- *Application*

4.21.5 Global Properties

The following properties are available in any module:

Name	Description	Default
`microejConflictResolutionRulesEnabled`	Boolean to enabled or disabled the MicroEJ conflict resolution rules.	`true`

For example:

```
microej {
  microejConflictResolutionRulesEnabled = false
}
```

4.22 Troubleshooting

4.22.1 Java Compiler Version Issue

The SDK requires a JDK 11, so when a JDK 8 is used, the following kind of errors are raised:

- When fetching the MicroEJ Gradle plugin:

```
A problem occurred configuring root project 'myProject'.
> Could not resolve all files for configuration ':classpath'.
> Could not resolve com.microej.gradle.plugins:plugins:0.3.0.
        Required by:
                project : > com.microej.gradle.addon-library:com.microej.gradle.addon-
↪library.gradle.plugin:0.3.0:20221118.151454-1
        > No matching variant of com.microej.gradle.plugins:plugins:0.3.0:20221118.
↪151454-1 was found. The consumer was configured to find a runtime of a library␣
↪compatible with Java 8, packaged as a jar, and its dependencies declared externally,␣
↪as well as attribute 'org.gradle.plugin.api-version' with value '7.4' but:
                - Variant 'apiElements' capability com.microej.gradle.plugins:plugins:0.
↪3.0 declares a library, packaged as a jar, and its dependencies declared externally:
                        - Incompatible because this component declares an API of a␣
```

(continues on next page)

(continued from previous page)

```
↪component compatible with Java 11 and the consumer needed a runtime of a component␣
↪compatible with Java 8
                        - Other compatible attribute:
                                - Doesn't say anything about org.gradle.plugin.api-
↪version (required '7.4')
                - Variant 'javadocElements' capability com.microej.gradle.
↪plugins:plugins:0.3.0 declares a runtime of a component, and its dependencies␣
↪declared externally:
                        - Incompatible because this component declares documentation␣
↪and the consumer needed a library
                        - Other compatible attributes:
                                - Doesn't say anything about its target Java version␣
↪(required compatibility with Java 8)
                                - Doesn't say anything about its elements (required␣
↪them packaged as a jar)
                                - Doesn't say anything about org.gradle.plugin.api-
↪version (required '7.4')
                - Variant 'runtimeElements' capability com.microej.gradle.
↪plugins:plugins:0.3.0 declares a runtime of a library, packaged as a jar, and its␣
↪dependencies declared externally:
                        - Incompatible because this component declares a component␣
↪compatible with Java 11 and the consumer needed a component compatible with Java 8
                        - Other compatible attribute:
                                - Doesn't say anything about org.gradle.plugin.api-
↪version (required '7.4')
                - Variant 'sourcesElements' capability com.microej.gradle.
↪plugins:plugins:0.3.0 declares a runtime of a component, and its dependencies␣
↪declared externally:
                        - Incompatible because this component declares documentation␣
↪and the consumer needed a library
                        - Other compatible attributes:
                                - Doesn't say anything about its target Java version␣
↪(required compatibility with Java 8)
                                - Doesn't say anything about its elements (required␣
↪them packaged as a jar)
                                - Doesn't say anything about org.gradle.plugin.api-
↪version (required '7.4')
```

- When using the MicroEJ Gradle plugin:

```
Cause: com/microej/gradle/plugins/MicroejApplicationGradlePlugin has been compiled by a␣
↪more recent version of the Java Runtime (class file version 55.0), this version of␣
↪the Java Runtime only recognizes class file versions up to 52.0
```

The solution is to use a JDK 11 or a higher LTS version (`11` , `17` or `21`) to fix this error:

- For the command line interface, make sure that a supported JDK version is defined in the `PATH` environment. To check, run `java -version` . You should see something like this:

```
$ java -version
openjdk version "11.0.14.1" 2022-02-08
OpenJDK Runtime Environment Temurin-11.0.14.1+1 (build 11.0.14.1+1)
OpenJDK 64-Bit Server VM Temurin-11.0.14.1+1 (build 11.0.14.1+1, mixed mode)
```

Alternatively, you can set the `JAVA_HOME` environment variable to point to the installation directory of the JDK.

- For Android Studio and IntelliJ IDEA, go to File > Settings... > Build, Execution, Deployment > Build Tools > Gradle, and make sure the selected Gradle JVM is a supported JDK version:

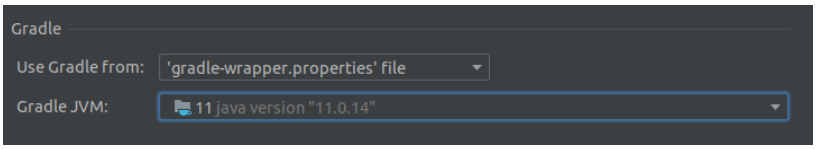


Fig. 56: Project JDK in Android Studio and IntelliJ IDEA

4.22.2 Unresolved Dependency

If this kind of message appears when resolving plugins or modules dependencies:

```
* What went wrong:
Plugin [id: 'com.microej.gradle.application', version: '0.15.0'] was not found in any of the␣
↪following sources:
```

or this kind:

```
* What went wrong:
Execution failed for task ':compileJava'.
> Could not resolve all files for configuration ':compileClasspath'.
  > Could not find com.mycompany:mymodule:M.m.p.
     Searched in the following locations:
       - https://my-company-first-repository/com/mycompany/mymodule/M.m.p/kf-M.m.p.pom
       - https://my-company-first-repository/com/mycompany/mymodule/M.m.p/ivy-M.m.p.xml
       - https://my-company-second-repository/com/mycompany/mymodule/M.m.p/kf-M.m.p.pom
       - https://my-company-second-repository/com/mycompany/mymodule/M.m.p/ivy-M.m.p.xml
     Required by:
         project :
```

First, check that either the requested plugin or module exists in your repository.

- If the plugin or module does not exist,
  - if it is declared as a *direct dependency*, the module repository is not compatible with your source code. You can either check if another module version is available in the repository or add the missing module to the repository.
  - otherwise, this is likely a missing transitive module dependency. The module repository is not consistent. Check the module repository and make sure all the transitive dependencies exist.
- If the module exists, this may be due to a missing repository in the configuration. Check that your repository appears in the list of URLs below the error line:

  ```
  Searched in the following locations:
  ```

  If the URL of your repository is not listed, add it to *the list of the repositories*.

- If the repository is correctly configured, this may be a network connection error. We can check in the debug logs, by adding the `--debug` arguments in the Gradle command line.

Otherwise, if your module repository is an URL, check for an *Invalid SSL Certificate* issue.

4.22.3 Invalid SSL Certificate

If a dependency cannot be retrieved from a remote repository, this may be due to a missing or incorrect SSL certificate. It can be checked in the debug logs, by adding the `--debug` and `-Djavax.net.debug=all` arguments in the Gradle command line, for example:

```
./gradlew build --debug -Djavax.net.debug=all
```

If the SSL certificate is missing or incorrect, the following line should appear:

```
PKIX path building failed: sun.security.provider.certpath.
↪SunCertPathBuilderException: unable to find valid certification path to␣
↪requested target
```

This can be raised in several cases, such as:

- an artifact repository configured in the MicroEJ Module Manager settings using a self-signed SSL certificate or a SSL certificate not trusted by the JDK.
- the requests to an artifact repository configured in the MicroEJ Module Manager settings are redirected to a proxy server using a SSL certificate not trusted by the JDK.

In all cases, the SSL certificate (used by the artifact repository server or the proxy) must be added to the JDK trust store that is running Gradle. Ask your System Administrator, or retrieve the SSL certificate and add it to the JDK trust store:

- on Windows
  1. Install Keystore Explorer.
  2. Start Keystore Explorer, and open file `[JRE_HOME]/lib/security/cacerts` or `[JDK_HOME]/jre/lib/security/cacerts` with the password `changeit`. You may not have the right to modify this file. Edit rights if needed before opening it or open Keystore Explorer with admin rights.
  3. Click on `Tools`, then `Import Trusted Certificate`.
  4. Select your certificate.
  5. Save the `cacerts` file.
- on Linux/macOS
  1. Open a terminal.
  2. Make sure the JDK's `bin` folder is in the `PATH` environment variable.
  3. Execute the following command:

     ```
     keytool -importcert -v -noprompt -trustcacerts -alias myAlias -file /path/to/the/
     ↪certificate.pem -keystore /path/to/the/truststore -storepass changeit
     ```

If the problem still occurs, there should be a trace which indicates the beggining of the handshake phase of the SSL negotiation:

```
2023-12-15T17:32:47.442+0100 [DEBUG] [org.apache.http.conn.ssl.SSLConnectionSocketFactory]␣
↪Starting handshake
```

The error very probably occurs during this phase. There should be the following trace before the error:

```
Consuming server Certificate handshake message
```

The traces below this one indicates the SSL certificate (or the SSL certificates chain) presented by the server. This certificate or one of the root or intermediate certificates must be added in the JDK truststore as explained previously.

4.22.4 Failing Resolution in `adp` Task

During the build of a project, the error `Cannot locate module version for non-maven layout` may be raised:

```
Execution failed for task ':adp'.
> Could not resolve all files for configuration ':addonProcessorClasspath'.
        > Could not download binary-nls-processor-2.4.2.adp (com.microej.tool.addon.
↪runtime:binary-nls-processor:2.4.2)
                > Cannot locate module version for non-maven layout.
        > Could not download js-processor-0.13.0.adp (com.microej.tool.addon.runtime:js-
↪processor:0.13.0)
                > Cannot locate module version for non-maven layout.
        > Could not download junit-processor-1.7.1.adp (ej.tool.addon.test:junit-processor:1.
↪7.1)
                > Cannot locate module version for non-maven layout.
```

This is due to a wrong pattern in the declaration of the Ivy repositories. Check your Ivy repositories and make sure the value of the `artifact` of the `patternLayout` block is set to `[organisation]/[module]/[revision]/[artifact]-[revision](-[classifier])(.[ext])`. For example:

```
ivy {
        url = uri("https://repository.microej.com/5/artifacts/")
        patternLayout {
                artifact("[organisation]/[module]/[revision]/[artifact]-[revision](-
↪[classifier])(.[ext])")
                ivy("[organisation]/[module]/[revision]/ivy-[revision].xml")
                setM2compatible(true)
        }
}
```

4.22.5 Missing Version for Publication

If the following message is displayed when publishing a module:

```
The project version must be defined.
```

It means the `version` property is missing and should be defined in the module build file. See *Publish a Project* for more information.

4.22.6 Fail to load a VEE Port as dependency

When a VEE Port is defined as a dependency, the build of the project can fail with the following message:

```
      > No 'release.properties' and 'architecture.properties' files found.
The given file <path/to/file> is not a VEE Port archive.
```

If the dependency is a valid VEE Port, this error probably means that several artifacts of the VEE Port have been published with the `default` Ivy configuration. To fix this issue, you can select the right artifact by adding information on the one to fetch in the `artifact` block, for example:

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

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

4.22.7 Slow Build because of File System Watching

In some cases, Gradle may take a lot of time to execute its build. One of the possible reasons is the file system watching feature which allows Gradle to track any change on the file system. Depending on your environment, this feature can impact the build execution time significantly. For example, when network drives are mapped and the network connection experiences instability.

This feature can be disabled for a build by passing the `--no-watch-fs` option in the command line, for example:

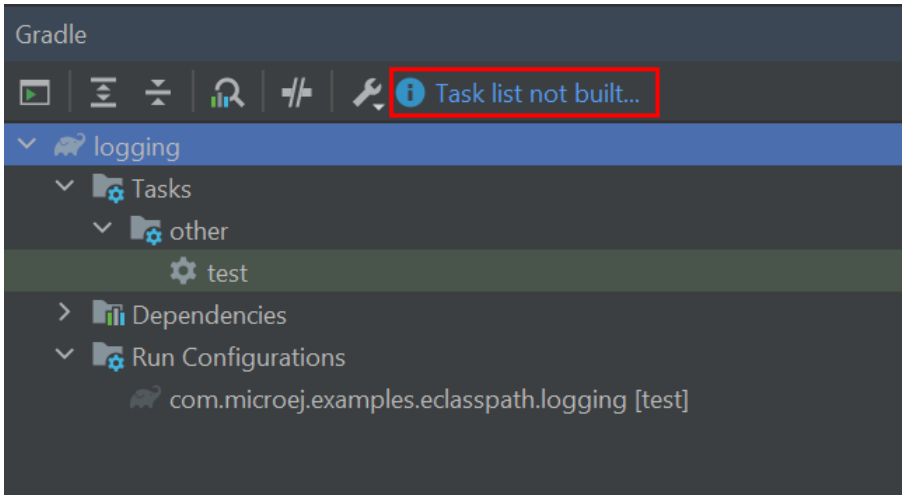
```
./gradlew build --no-watch-fs
```

or for all builds by setting the following property in the `$USER_HOME/.gradle/gradle.properties` file:

```
org.gradle.vfs.watch=false
```

4.22.8 Missing Tasks in the Gradle view of Android Studio

In some cases, Android Studio may not build all the Gradle tasks, the `Task list not built...` message is displayed:

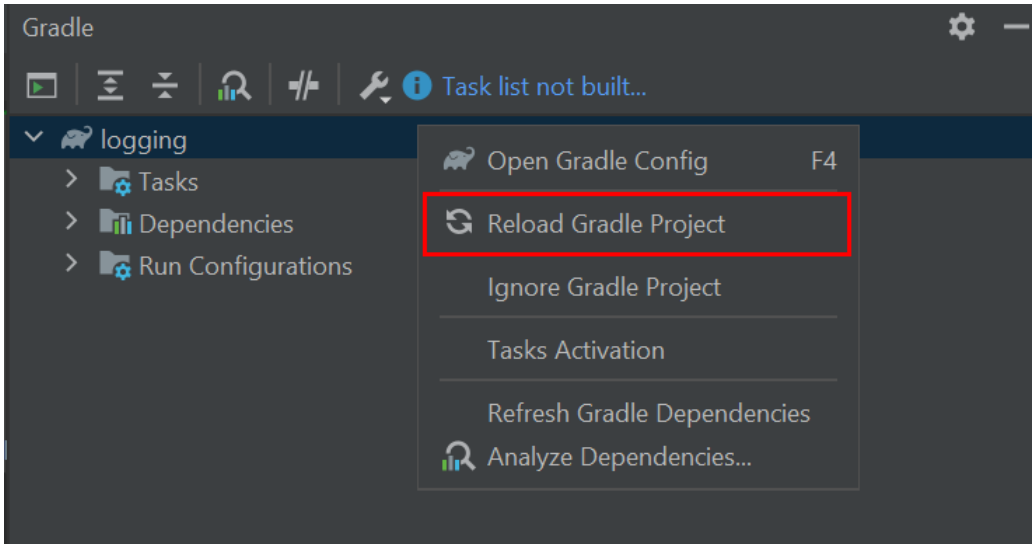


To build all the Gradle tasks in Android Studio:

- Go to `File` > `Settings` > `Experimental` ,
- Enable the option: `Configure all Gradle tasks during Gradle Sync (...)` .

Back in the Gradle task view:

- Right-click on the project name,
- Select `Reload Gradle Project` .



Note: By default, all supported IDEs require the user to explicitly trigger the reload of a Gradle project when its configuration has changed. However you can configure your IDE to automatically reload your project. Refer to the *How To Automatically reload a Gradle project* section for more information.

4.23 Tutorials

4.23.1 Branding an Eclipse IDE

Eclipse IDE allows to create custom versions of its distribution. This can be very convenient if you need to redistribute your own unique Eclise IDE version, customized to your brand.

This tutorial will guide you through the steps to create such a branded Eclipse IDE.

Install Eclipse and the MicroEJ Plugin

- Download the Eclipse IDE for RCP and RAP Developers Edition.
- Install and launch it.
- Install the MicroEJ plugin as described in the `Eclipse` tab of the *Install the IDE Plugin* chapter.

Create the Project

Once everything is installed, the first step is to create the project:

- Click on `File` > `New` > `Plug-in Project` .
- Fill the `Project name` field, for example `my-branded-eclipse` .
- Click on the `Next` button.
- Change the version, the name and the vendor if necessary.

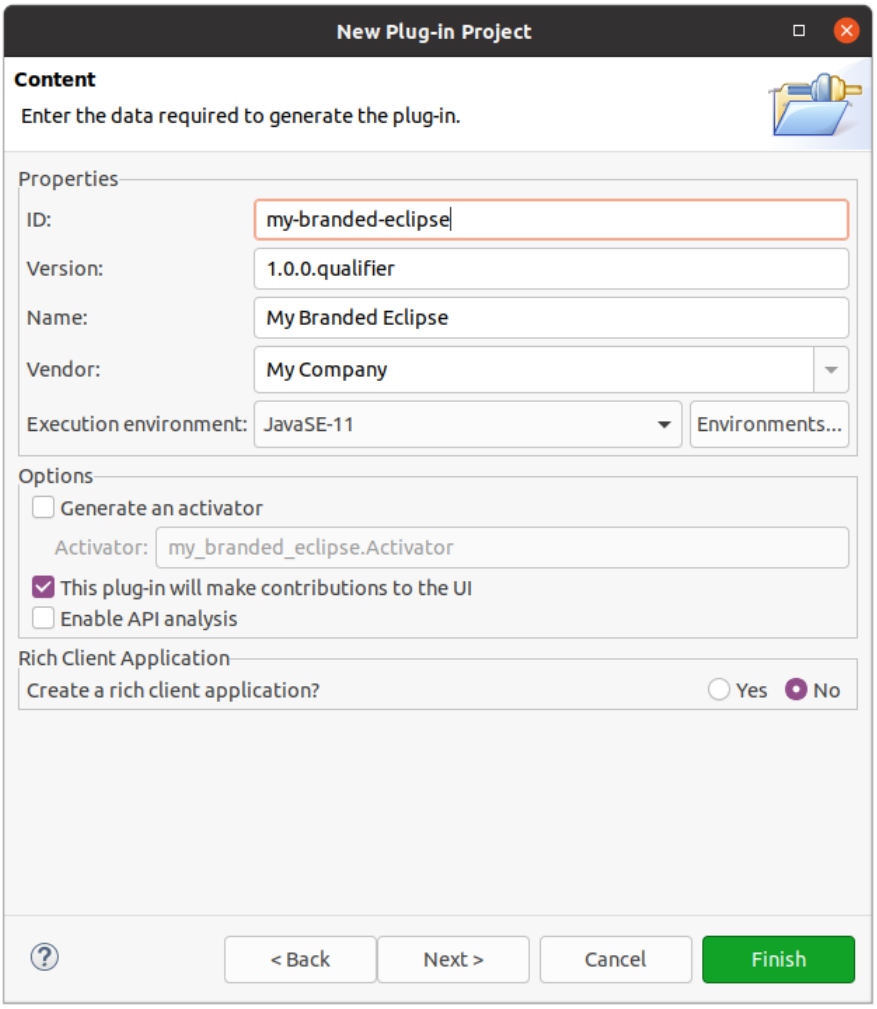


Fig. 57: Creation of a Branding Eclipse Project

- Click on the Finish button.
- Create a folder named `images` at the root of the project. It will contain the branding resources described later in this tutorial.

Configure the Product

- Right-click on the project, then click on New > Other... .
- Select Plug-in Development > Product Configuration .
- Click on the Next button.
- Set a file name, for example `eclipse` .
- Click on the Finish button. The Product configuration file should be created and opened now.
- In the Overview tab
  - In the Product field, click on New... .

  - Set a value in the Product Name field, for example `My Branded Eclipse`.
  - Select the project in the Defining Plug-in field.
  - Set a value in the Product ID field, for example `product`.
  - In the Application field, select `org.eclipse.ui.ide.workbench`.
  - Click on the Finish button.
- In the Contents tab
  - For each of the following terms, click on Add..., type the term in the field, then select all the items in the list and click on the Add button:

```
jdt
microej
egit
buildship
mpc
mylyn
org.eclipse.ui.ide.application
<plugin-name> (``my-branded-eclipse`` for the example values used previously)
```

  - Click on Add Required Plug-ins.
- In the Configuration tab
  - Click on the Add Recommended... button.
  - Click on the OK button.
- In the Launching tab
  - If you want to change the default name of the Eclipse launch executable (defaults to `eclipse`), set the Launcher Name field with the new name.
  - If you want to change the icon files of the Eclipse launch executable file,
    * Copy the image file(s) of the IDE launcher in the `images` folder. The image format depends on the Operating System:
      · `icon.xpm` for Linux
      · `icon.icns` for macOS
      · `icon.ico` file or `icon.bmp` files for Windows.

      For Windows, if `bmp` files are used, it is required to provide one `bmp` file for each one of the following resolutions: 16x16 (8-bit), 16x16 (32-bit), 32x32 (8-bit), 32x32 (32-bit), 48x48 (8-bit), 48x48 (32-bit), 256x256 (32-bit).
    * Select the icon files for the targeted Operating Systems. Make sure the paths are the relative paths from the project root folder.
- In the Splash tab
  - If you want to change the default splash screen displayed at startup,
    * Copy the image file of the splash screen at the root of the project. The following name and image types are supported:

        - `splash.png`
        - `splash.jpg`
        - `splash.jpeg`
        - `splash.gif`
        - `splash.bmp`

      The recommended size for the splash screen is 455x295.

    * In the Plug-in field, click on the Browse... button.
    * Select the plugin of the project (`my-branded-eclipse` in our example).

  - If you want to change the splash screen behavior, adapt the other options in the Customization section to your need. For example you may want to add a progress bar in the splash screen by checking the option Add a progress bar .

- In the Branding tab (make sure the image paths are the relative paths from the project root folder)
  - If you want to change the default application window icon (visible in the Windows dock for example),
    * Copy the image files associated with the application window in the `images` folder. The image format must be `png` , with one `png` file for each one of the following resolutions: 16x16, 32x32, 48x48, 64x64, 128x128, 256x256.
    * For each field in the Window Images section, select the corresponding image.
  - If you want to content of the `About` dialog (visible in Help > About...),
    * Copy the `About` dialog image in the `images` folder. This image must be in `png` format and should not exceed 500x330.
    * In the About Dialog section, select the image and fill the text. The text is not shown if the image exceeds 250x330.

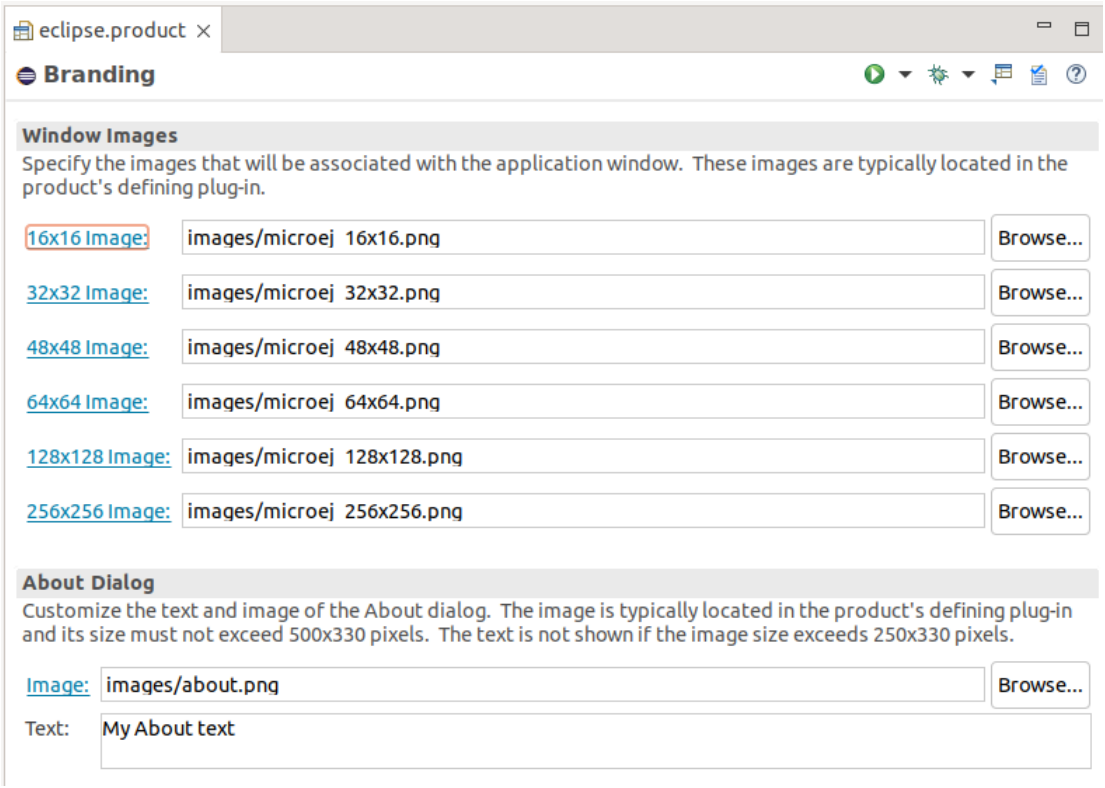


Fig. 58: Branding Tab of a Branding Eclipse Project

- Save the Product file.
- Go back to the `Overview` tab and click on `Synchronize` in the `Testing` section.

Your project should look like this at this stage:

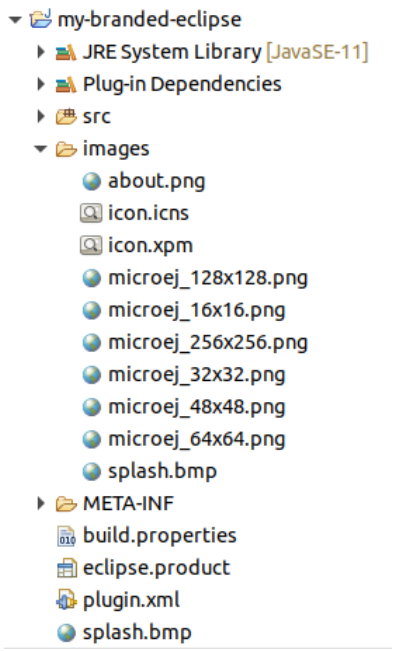


Fig. 59: Structure of a Branding Eclipse Project

Advanced Options

Eclipse provides several other options to customize an Eclipse Product that can be defined in the `plugin_customization.ini` file located at the root of the project. Create this file if it does not exist in your project.

Then you can define any option, for example to set the default perspective to the Java perspective:

```
org.eclipse.ui/defaultPerspectiveId=org.eclipse.jdt.ui.JavaPerspective
```

Here is a list of interesting options:

Name	Description	Default
org.eclipse.ui/SHOW_PROGRESS_ON_STARTUP	Show progress bar in the splash screen.	`false`
org.eclipse.ui/defaultPerspectiveId	Perspective that the workbench opens initially.	`org.eclipse.ui.resourcePerspective`

Export the Product

The final step is to export the project as an Eclipse Product:

- Open the `build.properties` file, and make sure to select the `Build` tab.
- In the `Binary Build` section, select:
  - `META-INF` folder
  - `plugin.xml` file
  - `splash.bmp` file
  - `images` folder
  - `plugin_customization.ini` file (if exists)
- Save your changes in the `build.properties` file.
- Right-click on the project, then click on `Export...` .
- Select `Plug-in Developement` > `Eclipse product` , then click on the `Next` button.
- In the `Configuration` field, select the `.product` file.
- In the `Synchronization` section, make sure the `Synchronize before exporting` option is checked.
- In the `Directory` field of the `Destination` section, select the destination folder.
- Click on the `Finish` button.

Once the process is done, you should find the new branded Eclipse IDE in the destination folder.

4.23.2 Creating and Using an Offline Repository

Developing MicroEJ projects requires the Gradle plugins used for the build, as well as the modules (Add-On Libraries, Foundation Libraries, ...) used by the project code. All these artifacts must be available in artifact repositories.

MicroEJ provides them as online repositories which can be used directly, thanks to the configuration described in the *Configure Repositories* section. However, it is not always possible to rely on these online repositories. Gradle allows to use repositories packaged as a set of local folders and files, called Offline Repositories.

This tutorial explains how to create and use Offline Repositories for your MicroEJ project.

Offline Repository for the Gradle Plugins

The first step is to create an Offline Repository for the Gradle plugins. The artifacts of the Gradle plugins are available in the SDK 6 Forge repository.

- Go to the SDK 6 repository.
- Click on the `Download` button:

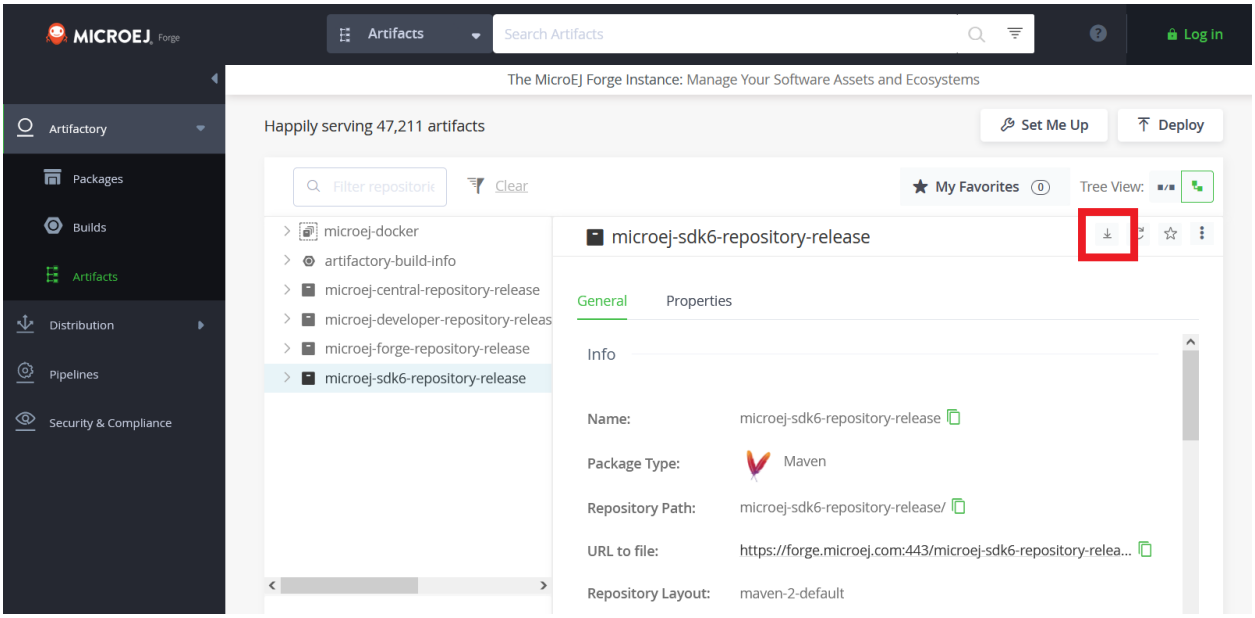


Fig. 60: Download SDK 6 Gradle Plugins Repository

- In the upcoming popup, check the Include Checksum Files checkbox.
- Click on Download .

Now that the Offline Repository of the Gradle plugins has been retrieved, you can configure your projects to use it:

- Unzip the downloaded archive at the location of your choice, for example in the `C:\sdk6-repository` folder.
- Add the following repositories declaration in *your repositories configuration script*, inside the `pluginManagement > repositories` block:

```
pluginManagement {
  repositories {

    ...

    maven {
        name = "offlineSDKRepositoryMaven"
        url = uri("C:\\sdk6-repository")
    }
    ivy {
        name = "offlineSDKRepositoryIvy"
        url = uri("C:\\sdk6-repository")
        patternLayout {
            artifact("[organisation]/[module]/[revision]/[artifact]-[revision](-
↪[classifier])(.[ext])")
            ivy("[organisation]/[module]/[revision]/ivy-[revision].xml")
            setM2compatible(true)
        }
    }

    ...
```

(continues on next page)

(continued from previous page)

```
  }
}
```

Offline Repository for the Modules

There are 2 ways to create an Offline Repository containing the required modules:

- download an existing online repository.
- create a SDK 5 offline repository project to create a custom repository.

Download an existing online repository

A quick way to get an Offline Repository for the modules is to download an existing online repository. MicroEJ provides several *module repositories*, the main one being the *Central Repository*.

If this online repository, or another one, contains all the module required for your project, download it. For example for the Central Repository, go to its location and click on the Download button.

Now go to *this section* to configure your project to use it.

Custom Offline Repository

If you need a custom Offline Repository (for example because the available online repositories does not contain all the modules required by your project, or you want to control exactly what contains the repository), you can create your own. This can be done only with SDK 5 for the moment, so refer to *this page*.

Once done, go to *this section* to configure your project to use it.

Use an Offline Modules Repository

When the Offline Repository of the modules has been retrieved or created, you can configure your projects to use it:

- Unzip the Offline Repository archive at the location of your choice, for example in the `C:\modules-repository` folder.
- Add the following repositories declaration in *your repositories configuration script*, inside the `repositories` block:

```
repositories {

  ...

  maven {
      name = "offlineModulesRepositoryMaven"
      url = uri("C:\\modules-repository")
  }
  ivy {
      name = "offlineModulesRepositoryIvy"
      url = uri("C:\\modules-repository")
```

(continues on next page)

(continued from previous page)

```
      patternLayout {
          artifact("[organisation]/[module]/[revision]/[artifact]-[revision](-[classifier])(.
↪[ext])")
          ivy("[organisation]/[module]/[revision]/ivy-[revision].xml")
          setM2compatible(true)
      }
  }

  ...

}
```

4.24 How-to Guides

4.24.1 How To Define a Specific Java Home for Gradle

By default, Gradle uses the JDK defined in the `JAVA_HOME` environment variable or in the `PATH`. If you want to use a different JDK without changing the default JDK of your system, you can define the property `org.gradle.java.home` in the Gradle Properties. Gradle Properties can be defined in the following locations, sorted by the highest priority:

- command line, as set using `-D`.
- `gradle.properties` in the `GRADLE_USER_HOME` directory (defaults to `$USER_HOME/.gradle`).
- `gradle.properties` in the project directory, then its parent project directory up to the build root directory.
- `gradle.properties` in the Gradle installation directory.

If an option is configured in multiple locations, the first one found in any of these locations wins. Therefore, if you want all your Gradle project to use a different JDK than the system default JDK, you can add the following property in the file `$USER_HOME/.gradle/gradle.properties`:

```
org.gradle.java.home="C:\\path\\to\\the\\jdk"
```

4.24.2 How To Pass a Property to Project Build Script

It is sometimes needed to use properties to pass values to a project build script. This avoids to have hardcoded values in the project sources.

Gradle allows to define System Properties with the command line thanks to the `-D` prefix:

```
$ ./gradlew build -DmyPropertyName="myPropertyValue"
```

and use them with the API `providers.systemProperty("myPropertyName").get()`.

For example to define a local VEE Port directory, the project can be configured with:

```
dependencies {
   microejVee(files(providers.systemProperty("myVeePortPath").get()))
}
```

and built with:

```
$ ./gradlew build -DmyVeePortPath="C:\\path\\to\\my\\veePort\\directory"
```

The `providers.systemProperty("myPropertyName")` API returns a `org.gradle.api.provider.Provider` object, which provides other capabilities like:

- defining a default value if the System Property does not exist: `providers.systemProperty("myPropertyName").getOrElse("myDefaultValue")`
- returning `null` if the value does not exist: `providers.systemProperty("myPropertyName").getOrNull()`.

4.24.3 How To Skip a Gradle Task

When a task is executed, it is possible to skip one or more of the tasks on which the called task depends. For example, you can skip the `test` task if you want to build a project without executing the tests.

If you want to skip a task, one of the following ways can be used :

- Add the `-x` or `--exclude-task` option in the command line:

  ```
  ./gradlew build -x test
  ```

  The task is skipped for this execution only.

- Exclude the task in the build script of the project

  ```
  project.gradle.startParameter.excludedTaskNames.add("test")
  ```

  The task is never executed.

When one of these two ways is used, not only the task but also all the tasks on which it depends are skipped. For example, if you choose to skip the `test` task, all the tasks which are used to produce the test runtime classpath are also skipped.

Skip the task only

It is possible to skip a task but still execute the tasks on which it depends using one of the following ways :

- Disable the task in the build script of the project:

  ```
  tasks.test {
    enabled = false
  }
  ```

  The task is never executed.

- Define a predicate in the build script of the project:

  ```
  tasks.test {
    val skipProvider = providers.gradleProperty("skipTest")
    onlyIf {
      !skipProvider.isPresent()
    }
  }
  ```

  The task is skipped each time the predicate evaluates to `false`. In this example, the `test` task is not executed if the `skipTest` property is added in the command line:

```
./gradlew build -PskipTest
```

4.24.4 How To Automatically reload a Gradle project

By default, regardless of the IDE that you are using (Android Studio, IntelliJ IDEA or Eclipse), the reload of a Gradle project must be explicitly triggered by the user when the configuration of the project has changed. This allows to avoid reloading the project too frequently, but the user must not forget to manually reload the project to apply changes.

It is also possible to configure your IDE to automatically reload your Gradle project:

Android Studio / IntelliJ IDEA

Eclipse

The auto-reload of a Gradle project with Android Studio / IntelliJ IDEA can be enabled as follows:

- Click on File > Settings… .
- Go to Build, Execution, Deployment > Build Tools .
- Check the Reload changes in the build scripts option and check the Any changes option.

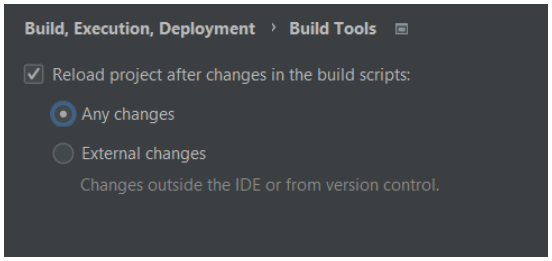


Fig. 61: Auto-reload option in Android Studio / IntelliJ IDEA

- Go to Languages & Frameworks > Kotlin > Kotlin Scripting .
- Check all the Auto Reload options.

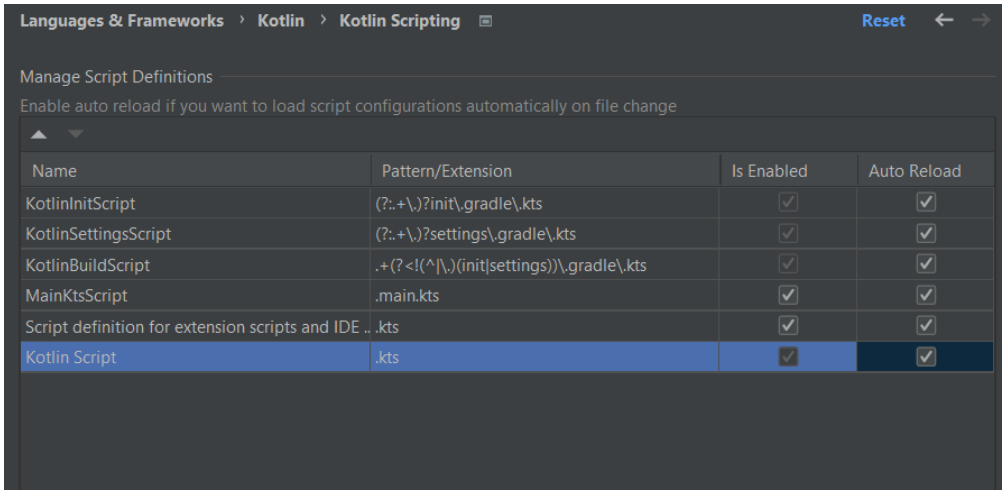


Fig. 62: Auto-reload Kotlin option in Android Studio / IntelliJ IDEA

The auto-reload of a Gradle project with Eclipse can be enabled as follows:

- Click on `Window` > `Preferences` > `Gradle` .
- Check the `Automatic Project Synchronization` option.

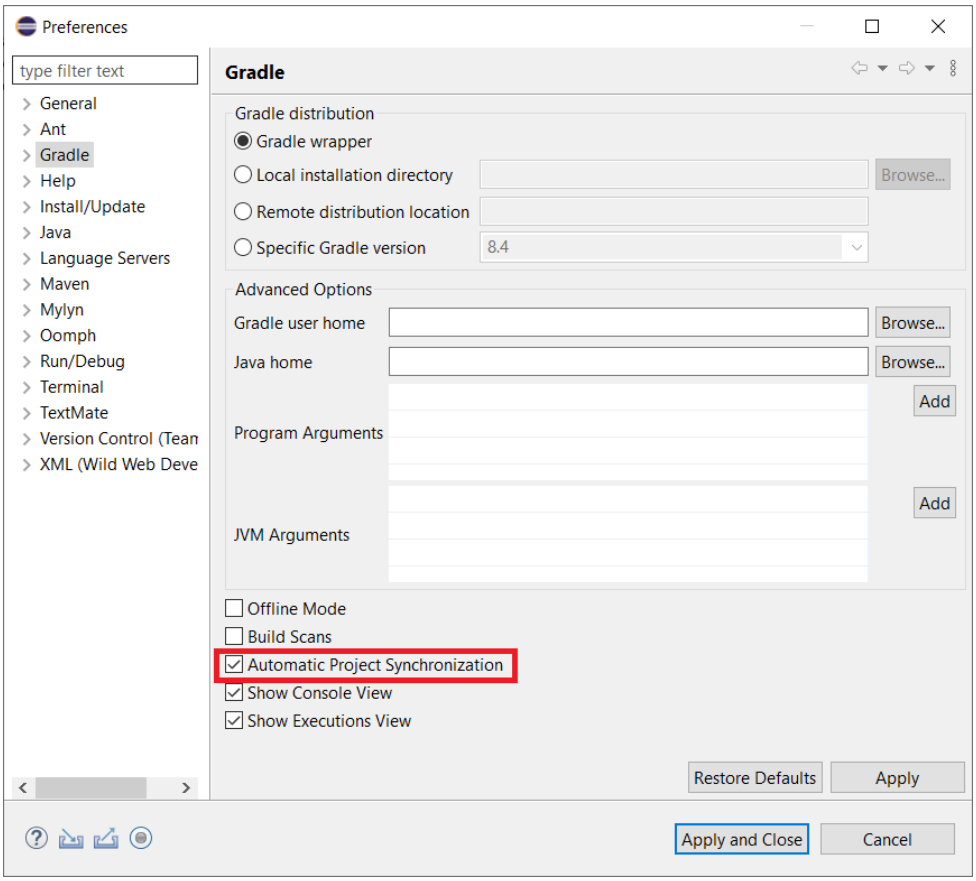


Fig. 63: Auto-reload option in Eclipse

4.24.5 How To Add a Repository

The *SDK 6 installation process* asks to create a Gradle Init Script file to declare modules and plugins repositories. You may need to use additional repositories or replace the default ones, for example to fetch a module only available in your company's repository. This page presents the different options to do that.

If you need more details on this topic, refer to the official Gradle documentation on repository declaration.

How To Add a Modules Repository

The different ways to add a modules repository are:

- add a `repositories` block in the `build.gradle.kts` file of the project:

```
repositories {
  maven {
    name = "myModulesRepository"
    url = uri("https://my.company/my-modules-repository")
  }
}
```

  The repositories defined here are fetched **after** the ones defined in the Gradle init script.

  For a multi-project, the repositories must be declared in a `build.gradle.kts` file located in the root folder to make them available in all the subprojects.

- update the Gradle Init Script to add, replace or delete a repository. The version of this script provided in the installation process is a recommended version to be applied to quickly setup an environment. However, it can be modified to adapt it to your need, especially for the list of repositories. The modules repositories are defined in the block `settingsEvaluated > allprojects > repositories`, and are applied to all the Gradle builds executed on the machine.

How To Add a Plugins Repository

The different ways to add a plugins repository are:

- add a `pluginManagement > repositories` block in the `settings.gradle.kts` file of the project or the multi-project:

```
pluginManagement {
  repositories {
    maven {
      name = "myPluginsRepository"
      url = uri("https://my.company/my-plugind-repository")
    }
  }
}
```

  The repositories defined here are fetched **before** the ones defines in the init script.

- update the Gradle Init Script to add, replace or delete a repository. The version of this script provided in the installation process is a recommended version to be applied to quickly setup an environment. However, it can be modified to adapt it to your need, especially for the list of repositories. The plugins repositories are defined in the block `settingsEvaluated > allprojects > pluginManagement > repositories`, and are applied to all the Gradle builds executed on the machine.

4.24.6 How To Resolve Dependencies in the IDE

When contributing to multiple interdependent projects, it is very convenient and more productive to test a change without rebuilding and publishing manually the updated projects.

Gradle allows to consider a local project as a dependency thanks to the Composite Build feature. For example, if you have a project named `myApplication`, with the coordinates `com.mycompany:myApplication:1.0.0`, and a project named `myLibrary`, with the coordinates `com.mycompany:myLibrary:1.0.0`, structured as follows:

```
|- myApplication
|   |- src
|   |- build.gradle.kts
|   |- settings.gradle.kts
|- myLibrary
|   |- src
|   |- build.gradle.kts
|   |- settings.gradle.kts
```

And the `build.gradle.kts` file of the `myApplication` project declaring a dependency to the `myLibrary` module:

```
dependencies {
   implementation("com.mycompany:myLibrary:1.0.0")
}
```

Without any additional configuration, Gradle will try to fetch the `com.mycompany:myLibrary:1.0.0` dependency from the declared repositories. This means that when you do a change in the `myLibrary` project, it would require to build and publish it, then refresh dependencies on the `myApplication` project to get the update. This is painful and time consuming.

In order to avoid this, Gradle allows to consider the `myLibrary` build as part of the `myApplication` build, meaning that when the `myApplication` project is built, the `myLibrary` project is also rebuilt if it has been changed, and is used as the dependency. This can be configured by adding the following line in the `settings.gradle.kts` file of the `myApplication` project:

```
includeBuild("../myLibrary")
```

The path given to the `includeBuild` method is the relative path of the project to include.

> **Warning:** The `includeBuild` method should be used to declare a dependency between two autonomous projects. To declare a dependency between two subprojects of a multi-project, a Project dependency must be used. Refer to the *Dependencies Between Subprojects of a Multi-Project* section for more information.

Refer to the Official Gradle documentation on the Composite Build feature for more details.

Dependencies Between Subprojects of a Multi-Project

Gradle allows to declare dependencies between subprojects of a multi-project build by declaring a Project dependency.

For example, if you have a multi-project named `myProject` composed of two subprojects `myApplication` and `myLibrary`:

```
|- myProject
|   |- myApplication
|   |   |- src
|   |   |- build.gradle.kts
|   |   |- settings.gradle.kts
|   |- myLibrary
|   |   |- src
|   |   |- build.gradle.kts
|   |   |- settings.gradle.kts
|   |- settings.gradle.kts
```

You can declare a Project dependency in the `build.gradle.kts` file of the `myApplication` subproject to make it depend on the `myLibrary` subproject:

```
dependencies {
   implementation(project(":myLibrary"))
}
```

When building the `myApplication` subproject, the `myLibrary` subproject is also rebuilt if it has been changed, so contrary to a Module dependency (e.g. `implementation("com.mycompany:myLibrary:1.0.0")`), you don't have to manually build and publish it, and then refresh dependencies on the `myApplication` project to get the update.

Refer to the Official Gradle documentation about the different kinds of dependencies for more details.

4.24.7 How to Install MicroEJ Plugin Snapshot Version on Android Studio or IntelliJ IDEA

If you want to test the version under development, the latest snapshot version of the plugin can be installed:

- In Android Studio or IntelliJ IDEA, go to `File > Settings...`.
- Go to `Plugins` menu.
- Click on the icon at the right of the `Installed` tab, then click on `Manage Plugin Repositories`.

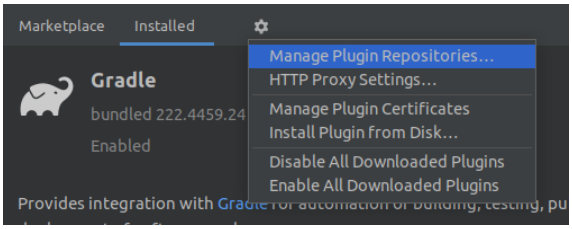


Fig. 64: Android Studio and IntelliJ IDEA Plugin Repository

- Click on the `+` icon.
- Set the URL `https://repository.microej.com/intellij-plugins/snapshots/updatePlugins.xml`.
- Click on the `OK` button.

- Click on the `Marketplace` tab.
- In the search field, type `MicroEJ`:

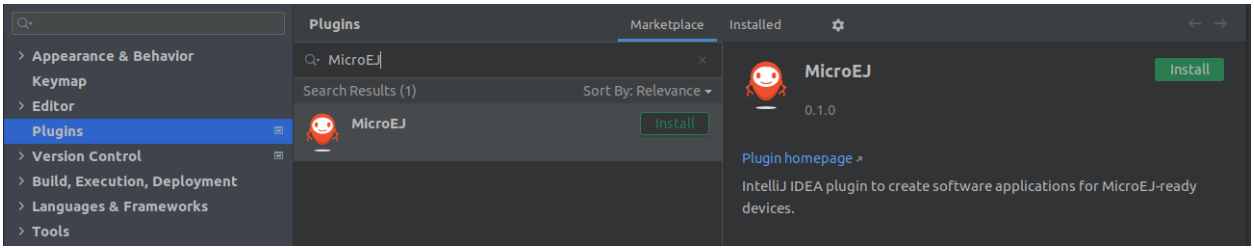


Fig. 65: Android Studio and IntelliJ IDEA Snapshot Plugin Installation

- Click on the `Install` button.
- Click on the `Restart IDE` button.

4.24.8 How To Build a Project

Generally speaking, building a project means compiling the source files, executing the tests and generating the module artifact. Depending on the nature of the project, the build can include other specific phases. Refer to the *Module Natures* page for a complete description of the build phases.

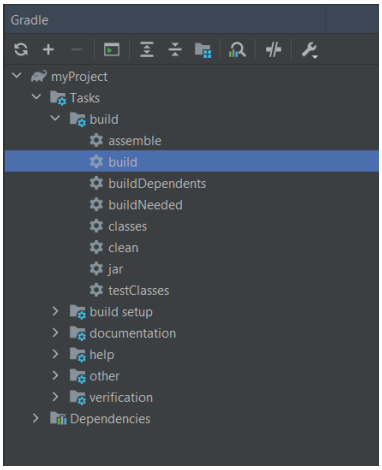
The build of a project is done by executing the Gradle `build` task.

Android Studio / IntelliJ IDEA

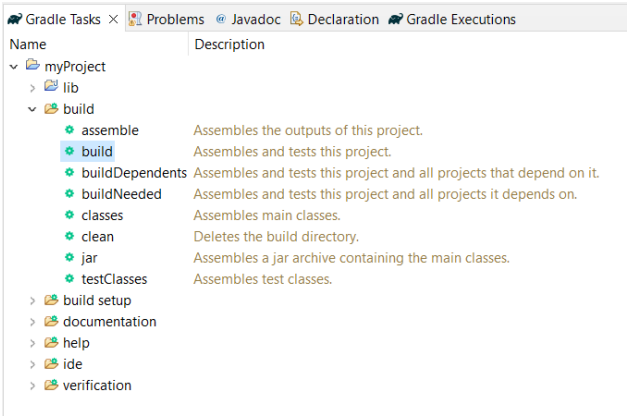
Eclipse

Command Line Interface

It can be executed from Android Studio or IntelliJ IDEA by double-clicking on the `build` task in the Gradle tasks view:



It can be executed from Eclipse by double-clicking on the `build` task in the Gradle tasks view:



It can be executed with the command line interface:

```
$ ./gradlew build
```

Gradle stores the artifacts produced by the build in the `build/libs` folder.

Note: If the build fails with a message related to the `Artifact Checker` such as:

```
The Artifact Checker found the following problems:
```

Fix the listed problems or skip the Artifact Checker by adding the following line in the build script file:

```
project.gradle.startParameter.excludedTaskNames.add("checkModule")
```

4.24.9 How To Build an Executable With Multiple VEE Ports

When creating an Application, only one VEE Port must be defined to build an Executable. However, it is possible to build an Executable for a list of VEE Ports by using a Gradle multi-project.

For example, if you want to build an Executable for two VEE Ports, you can create a multi-project composed of two subprojects:

```
|- rootProject
|   |- myApplicationVeePort1
|   |   |- src/main/java
|   |   |- src/main/resources
|   |   |- build.gradle.kts
|   |- myApplicationVeePort2
|   |   |- build.gradle.kts
|   |- settings.gradle.kts
|   |- build.gradle.kts
```

- The `myApplicationVeePort1` subproject is the Application project in which the first VEE Port is defined and the `applicationMainClass` property is set to the Fully Qualified Name of your main class.
- The `myApplicationVeePort2` subproject is an Application project that only contains a `build.gradle.kts` file in which the second VEE Port is defined and the `applicationMainClass` property is set to the Fully Qualified Name of your main class. To avoid having to duplicate the Source code of the Application, a SourceSet must be defined to use the Source code of the `myApplicationVeePort1` subproject:

```
sourceSets {
        main {
                java {
                        srcDirs(project(":myApplicationVeePort1").layout.
↪projectDirectory.dir("src/main/java"),
                            project(":myApplicationVeePort1").layout.projectDirectory.
↪dir("src/main/resources"))
                }
        }
}
```

- In the `build.gradle.kts` file of the multi-project, you can define the Gradle configuration that is common to all subprojects to avoid duplicates, for example:

```
plugins {
    id("com.microej.gradle.application") version "0.15.0" apply false
}

subprojects {
    apply(plugin = "com.microej.gradle.application")

    val implementation by configurations

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

- For each VEE Port, the Application configuration properties can be defined in the `configuration` folder of the corresponding Application project.

The Executable of the Application can now be built for a VEE Port by executing the `buildExecutable` task on the corresponding subproject:

```
./gradlew myApplicationVeePort1:buildExecutable
```

To build the Executable for all VEE Ports, a new task can be created in the `build.gradle.kts` file of the multi-project:

```
tasks.create("buildAllExecutables")
subprojects {
    rootProject.tasks.getByName("buildAllExecutables").dependsOn("$path:buildExecutable")
}
```

All Executables can now be built by executing the `buildAllExecutables` task:

```
./gradlew buildAllExecutables
```

For each VEE Port, the Executable is generated in the `build/output/application` folder of the corresponding subproject.

4.24.10 How To Create a Custom Configuration in the IDE

This chapter explains how to create a new Configuration in all the supported IDEs.

Android Studio / IntelliJ IDEA

Eclipse

The creation of a new Configuration with Android Studio / IntelliJ IDEA is done as follows:

- Click on `Run` > `Edit Configurations...` .
- Click on `+` and Select `Gradle` .
- Fill the name of the new Configuration in the `Name` field.
- Add a task and a property if needed in the `Run` dialog, for example `runOnSimulator -Pdebug.mode=true` .

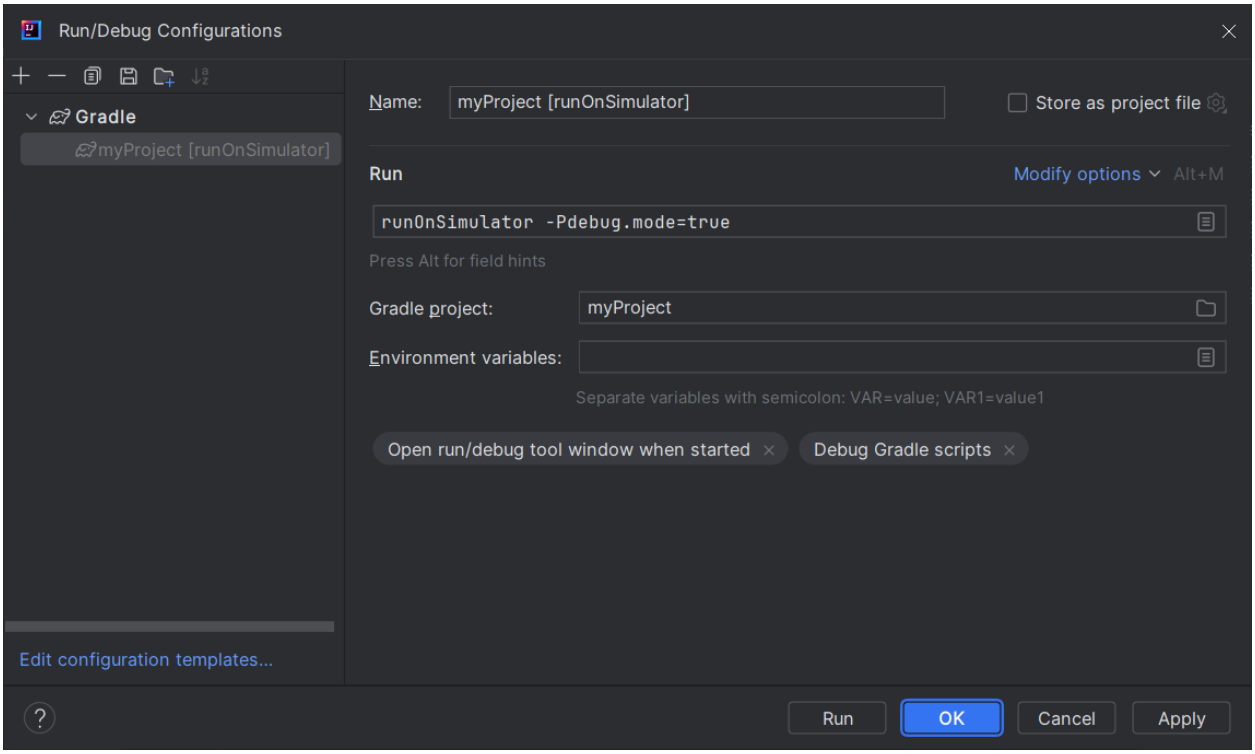


Fig. 66: Configuration Creation in IntelliJ IDEA

- Click on `OK` .
- Select the newly created Configuration in the drop-down list in the Navigation bar and click on the `run` button to launch it.

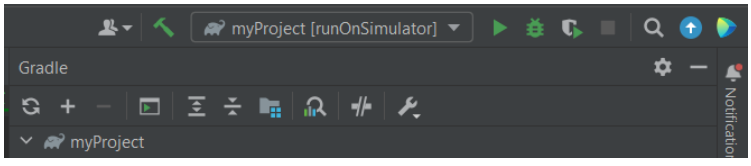


Fig. 67: Navigation bar in IntelliJ IDEA

- The Configuration can also be launched by double-clicking on it in the Gradle tasks view.

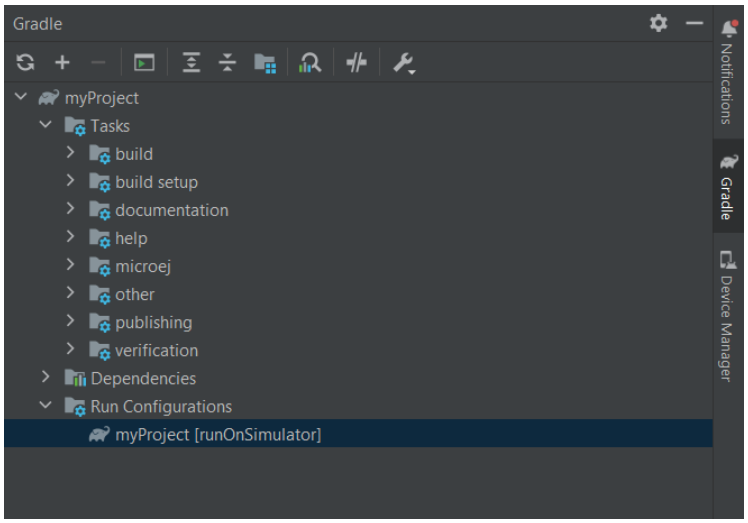


Fig. 68: Gradle view in IntelliJ IDEA

The creation of a new Configuration with Eclipse is done as follows:

- Click on `Run` > `Run Configurations...` .
- Right-click on `Gradle Task` and click on `New Configuration` .
- Fill the name of the new Configuration in the `Name` field.
- Add a task's name in the `Gradle Tasks` tab, for example `runOnSimulator` .

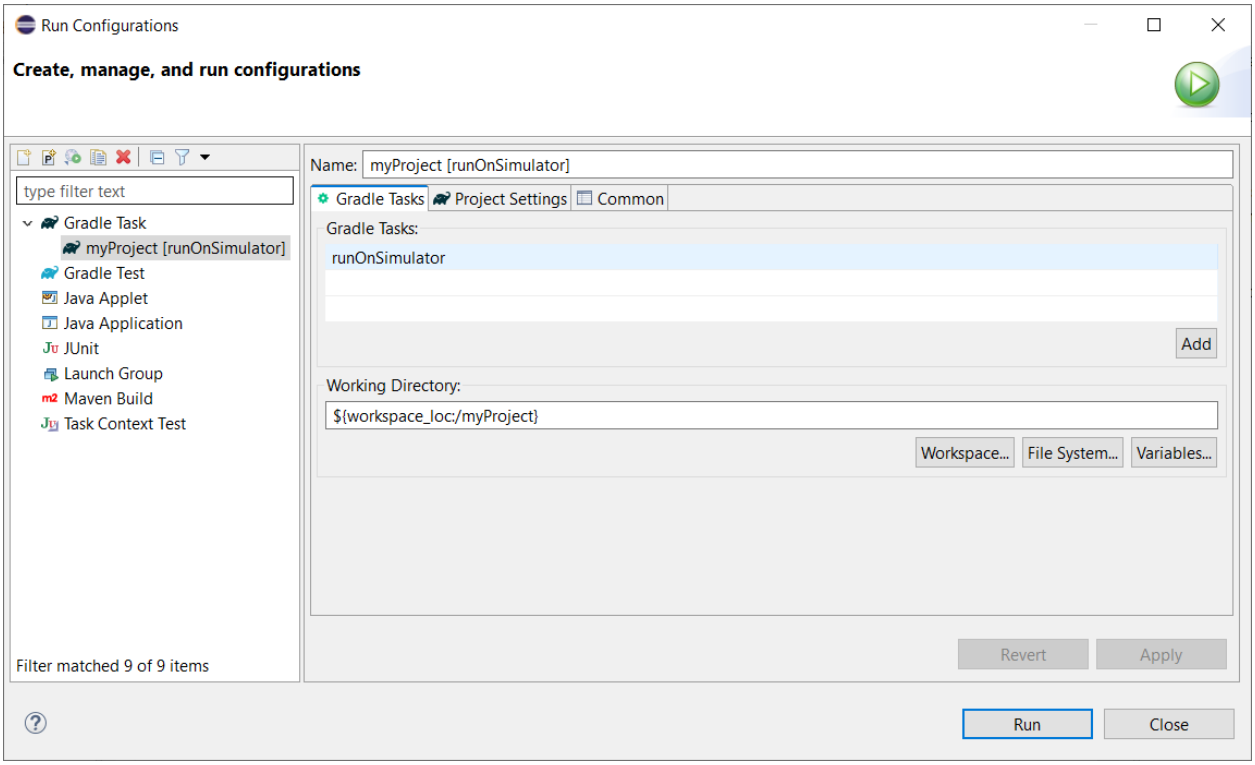


Fig. 69: Configuration Gradle Tasks tab in Eclipse

- Go to the `Project Settings` tab.
- Check `Override project settings`.
- Add a property as a Program Argument if needed, for example `-Pdebug.mode=true`.

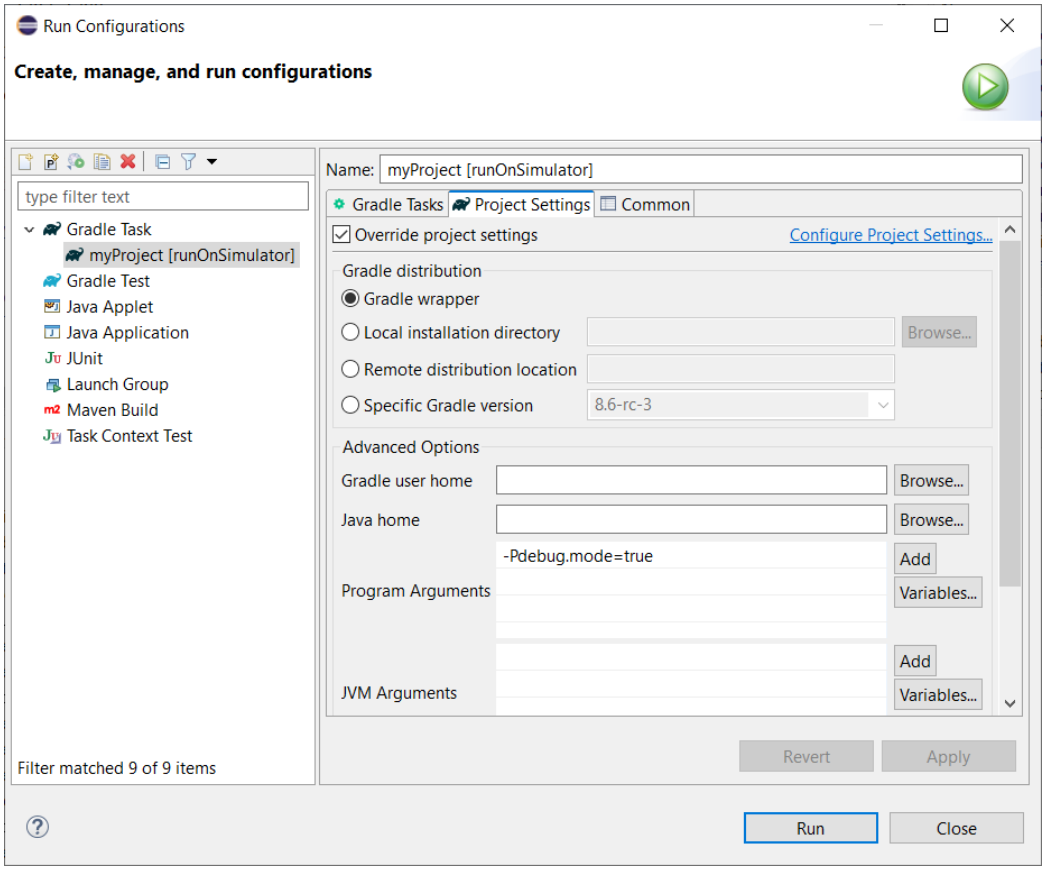


Fig. 70: Configuration Project Settings tab in Eclipse

- Click on Run to launch the Configuration.

4.25 Appendices

4.25.1 Virtual Device

This chapter describes the structure of a *Virtual Device*.

Structure

A Virtual Device is structured as follows:

```
|- virtualDevice
|   |- installed-applications
|   |- javaLibs
|   |- MICROJVM
|   |- main-application
|   |- mocks
|   |- options
|   |   |- target.properties
|   |- resources
```

(continues on next page)

(continued from previous page)

```
|   |- S3
|   |- scripts
|   |   |- init-vd
|   |   |   |- vd-init.xml
|   |- tools
|   |- architecture.properties
|   |- release.properties
|   |- veePort.properties
|   |- workbenchExtension*.jar
```

The Virtual Device contains the Simulation part files of the VEE Port used to build it:

- the `javaLibs/` folder, that contains the Foundation Libraries which are common to MICROJVM and S3
- the `linker/` folder, that contains the Linker jar files. This folder is not embedded in the Virtual Device if an Architecture `8.0.0` is used
- the `MICROJVM/` folder, that contains the VEE Port's files required to build a Feature file (`.fo`)
- the `mocks/` folder, that contains the Jar files of the mocks for Foundation Libraries
- the `resources/` folder, that contains the OS specific libraries
- the `S3/` folder, that contains the Simulator, HIL and Foundation Libraries specific to the Simulator
- the `scripts/` folder, that contains launch and initialization scripts
- the `tools/` folder
- the `workbenchExtension*.jar` files

The following elements are also embedded in the Virtual Device:

- a Kernel Application, whose WPK file is extracted in the `main-application` folder of the Virtual Device
- the WPK files of pre-installed Applications that are extracted in their own folder, in the `installed-applications/` folder that is empty by default

---

Note: Applications can only be pre-installed in a Multi-Sandbox Virtual Device. In case of a Mono-Sandbox Virtual Device, the `installed-applications/` folder is always empty.

- the `options/target.properties` , that contains the properties of the VEE Port used to build the Virtual Device
- the `scripts/init-vd/vd-init.xml` script, that allows to enable or not the Virtual Device. If the Virtual Device is not enabled, the Application main class specified by the user is launched on the VEE Port

You can refer to the *Build a Virtual Device* page to know how to build a Virtual Device.

4.25.2 Dependencies Configurations

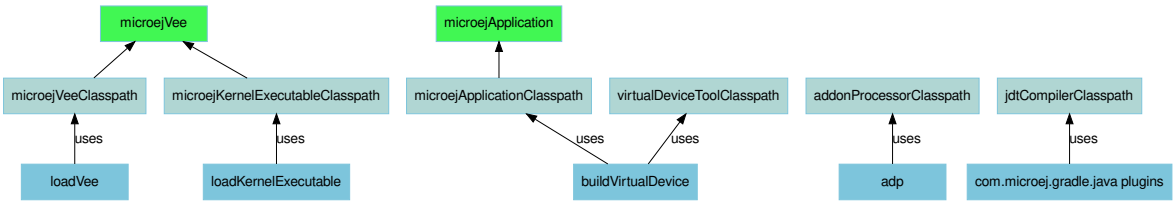
This chapter describes all the dependency configurations added to your project by the MicroEJ Gradle plugins.

Note: The MicroEJ Gradle plugins extend the Gradle Java and Java Library plugins. For more information about the configurations inherited from those plugins, refer to the official documentation :

- Java plugin
- Java Library plugin

The following diagrams show the dependency configurations. Use this legend to interpret the colors:

- Green background : Dependencies can be declared against this configuration
- Gray background : This configuration is for consumption by tasks only
- Blue background : A task or plugin



Publication Variants

The *Application* plugin defines a list of variants that are used during the publication of an Application.

microejWPK

This variant is used to publish the WPK of an Application, that can be fetched by declaring a dependency with the `microejApplication` configuration. The `LibraryElement` attribute of the variant is set to `microej-wpk`.

microejExecutable

This variant is used to publish the Executable of an Application, that can be fetched by declaring a dependency with the `microejVee` configuration. The `LibraryElement` attribute of the variant is set to `microej-executable`.

microejExecutableBuildFiles

This variant is used to publish the files generated when building the Executable of an Application. The `LibraryElement` attribute of the variant is set to `microej-build-files`.

microejVirtualDevice

This variant is used to publish the Virtual Device of an Application, that can be fetched by declaring a dependency with the `microejVee` configuration. The `LibraryElement` attribute of the variant is set to `microej-vee-port`.

4.26 Changelog

4.26.1 [0.15.0] - 2024-01-26

Added

- Unify `microejVeePort` and `microejKernel` configurations into *microejVee*.
- Add verification of dependencies checksums during build.
- Add the plugin `com.microej.gradle.mock` to build a Mock.
- Mention the system property to accept SDK EULA in error message.

Changed

- Task `:execTool` looks for a script named after the argument NAME with the following patterns in that order: NAME, NAME.microejTool, NAME.microejLaunch.
- Align the behavior of the `:buildFeature` task with the `localDeploymentSocket.microejLaunch` script.
  - output files are derived after “application” instead of “feature” (for example “application.fo”).
  - the application.main.class is set to the entryPoint defined in the .kf of the application.

Fixed

- Upgrade to junit-test-engine 0.2.2 to fix failing tests using fonts.
- Handle Security Manager removal from JDK 18+ when executing MicroEJ VEE scripts.
- Support all MicroEJ VEE (VEE Ports & Kernel) for the task `:execTool`.

Removed

- Remove support of dropIns folder for MicroEJ VEE (VEE Port or Kernel) selection.

4.26.2 [0.14.0] - 2024-01-03

Added

- Add Jenkinsfile files to build and test with a JDK 17 and a JDK 21 (LTS versions).

Changed

- Do not build/publish an Executable or a Feature by default and add the `produceExecutableDuringBuild()` and `produceFeatureDuringBuild()` methods to build them if needed.
- Set group and version for all projects, including the root project, in order to generate correctly the release tag.
- Use version 2.1.0 of the microej-licenses library to check with the new SDK EULA 3.1-B.

Fixed

- Fix the override behavior of the Application main class that was not consistent when `-Dapplication.main.class` is used.
- Fix the Custom Ant Logger to display build errors without having to enable the verbose mode.
- Follow Gradle recommendation on resolvable and consumable configurations.
- Move the Custom Logger to a dedicated module and use its jar instead of fetching the plugin when executing VEE Port scripts to fix the tests failure during a release.
- Set Java Compiler encoding to UTF-8.
- Set Java Compliance level to 1.7 in JavaPluginExtension to fix the `Cannot find the class file for java.lang.invoke.MethodHandles` error when opening a Gradle project in Eclipse.
- Fix wrong generated Virtual Device of an Application when the VEE Port changed.
- Bump source level for javadoc task to 1.8 to support JDK 21.
- Fix classpaths when using a Virtual Device to remove warnings about kf files not found.
- Make sure to close all opened streams.
- Fix `Wrong java/lang/Object` error when running an Application on the Simulator with a local repository.
- Fix `No .kf file found for this feature classpath` error message in logs when running VD with launcher script.

4.26.3 [0.13.0] - 2023-11-10

Added

- Add a check on EULA acceptation when using the MicroEJ Gradle plugin.
- Automatically publish the ASSEMBLY_EXCEPTION.txt file if it exists at the root of the project.
- Allow to publish the Feature file of an Application.
- Add `:execTool` task to execute Stack Trace Reader and Code Coverage Analyzer Tools provided by the selected VEE Port or Kernel.

Fixed

- Fix warning during compilation because of non-existing file or folder (`incorrect classpath: C:\\Users\\user\\...\myProject\\build\\resources\\main`).
- Fix warning in SOAR when building an Executable with Architecture 8.0.0 (`[M59] - Classpath file [C:\Users\user\...\myProject\build\resources\main] does not exist`).
- Enable Ant verbose mode for VEE Port scripts when Gradle debug log level is enabled.
- Fix the build of a Feature when the provided Virtual Device does not contain the `dynamicFeatureLink.microejLaunch` build script (Virtual Device built with SDK 5).

4.26.4 [0.12.1] - 2023-10-16

Fixed

- Fix the issue with the `microejKernel` configuration that prevented IDEs from loading a project.

4.26.5 [0.12.0] - 2023-10-13

Added

- Allow to publish the Virtual Device of an Application.
- Allow to fetch a Virtual Device and an Executable with the `microejKernel` configuration.

Changed

- Add README, CHANGELOG and License files as publication artifacts in the generated ivy.xml file.
- Publish test report in Jenkins job.
- Set `deploy.bsp.microejscript` property to `true` by default to build the executable.
- Publish the Executable file as a variant.
- Rename the `kernelFile` property to `kernelExecutableFile` .
- Use File dependency instead of the `veePortPath` property to load a local VEE Port.
- Use File dependency instead of the `kernelExecutableFile` property to load a local Kernel Executable.

Fixed

- Fix unexpected fetch of the transitive dependencies of a VEE Port dependency (`microejVeePort` configuration).
- Fix System properties defined in `gradle.properties` are ignored.
- Fix VEE Port launcher: temporary configuration file could prevent to launch a second time.
- Remove usage of deprecated API Project.getBuildDir().
- Fix the message when no executable are found by the `runOnDevice` task.
- Fix Executable not updated after a project change and a call to the `buildExecutable` task.
- Fix wrong order of tests classes and resources folder in the test classpath.
- Call VEE Port Ant script from a separate temporary directory to satisfy MicroEJ Architecture. This fixes spurious HIL timeouts when calling the `runOnSimulator` task.
- Fix Java process still running when Simulator is interrupted.
- Fix missing Nashorn dependencies when running a testsuite and when launching the launcher scripts to make it work with JDK 17 and higher.

4.26.6 [0.11.1] - 2023-09-22

Fixed

- Fix usage of a SNAPSHOT version of the junit-test-engine dependency.

4.26.7 [0.11.0] - 2023-09-22

Changed

- Use Gradle standard mechanism to support Multi-VEEPort instead of relying on an in-house feature.

4.26.8 [0.10.0] - 2023-09-13

Added

- Add a task `runOnDevice` to run the Executable on a Device.
- Support all JDK LTS versions higher or equals to version 11.

Fixed

- Allow to build a Feature file of an Application with a Virtual Device.
- Fix javadoc failure when the project contains a JDK class.

4.26.9 [0.9.0] - 2023-09-01

Added

- Allow to depend on local Application project (dependency with `microejApplication` configuration).
- Implement properties loading chain.
- Add launcher scripts to the Virtual Device.

Changed

- Move the `vd-init.xml` script in the template file instead of hardcoding it in the class.
- Remove the Application properties from `options/application.properties` file and rename file to `target.properties` in Virtual Device.
- Merge `veePortFiles` and `veePortDirs` properties into the `veePortPaths` property.
- Add missing Javadoc and clean the project.

Fixed

- Fix resources generated by Addon Processors of type FolderKind.MainResources not processed.
- Fix root path used for relative VEE port path: use the project root directory.
- Fix the content of a WPK to remove the Foundation Libraries.
- Make sure *.a* and *.o* files of an Application are correct by always executing the *buildExecutable* task.

4.26.10 [0.8.0] - 2023-07-13

Added

- Allow to build the Virtual Device of an Application.
- Add checks to ensure that a Virtual Device can be used or not depending on the called task.
- Allow to build the Feature binary file of an Application.

Fixed

- Add the Application properties defined in the `configuration folder` to the WPK file.

4.26.11 [0.7.0] - 2023-06-26

Added

- Add Standard Java Library plugin (`com.microej.gradle.j2se-library`).
- Rename `com.microej.gradle.library` plugin to `com.microej.gradle.addon-library` .

Changed

- Unbind the checkModule task from the build task.
- Use version `0.1.1` of the MicroEJ JUnit Test Engine to fix error when test classes are not in a package.
- Use version `2.0.0` of the microej-licenses library to check with the new authorized licenses.

4.26.12 [0.6.0] - 2023-05-30

Added

- Allow to publish WPK file artifact.
- Allow to publish files generated by the `buildExecutable` task.
- Allow to define multiple testsuites in different environments (sim or J2SE).
- Allow to define a testsuite for tests on device.

Changed

- Use Ivy descriptor content to know if a dependency is a Foundation Library or an Addon Processor Library.
- Optimize the `loadVeePort` task to reduce the time to load a VEE Port.
- Use a smaller VEE Port as dependency in tests to reduce the time to build.
- Remove `JPF` support.
- Check that the given file/directory is a VEE Port.
- Move Application properties to `configuration` folder instead of `src/main/resources`.
- Clean the Jenkins workspace after a successful build.
- Improve the checker on changelog files to support "-SNAPSHOT" suffix and "Unreleased" label.
- Remove the `debugOnSimulator` task and use a property to run an Application in debug mode.

Fixed

- Fix multiple VEE Ports error message in `loadConfiguration` task.
- Fix connection to a debugger and debug.port property.
- Fix StackOverflow error when building a project with cyclic dependencies.

4.26.13 [0.5.0] - 2023-03-24

Added

- Add Xlint checking.
- Add verification of using java 11 by user's project.
- Allow to build the Executable file of an Application.
- Allow to build the WPK file of an Application.

- Allow to define multiple VEE Ports.
- Check that the project uses at least Gradle 8.0.
- Add more tests on topological order in the Application classpath.

Changed

- Make the plugin compatible with Gradle `8.0`.

4.26.14 [0.4.0] - 2023-01-27

Added

- Apply the Java Library Plugin to user's project.
- Allow to load a VEE Port by dropping it in the `dropIns` folder.

Changed

- Optimise memory used by project.
- Remove the `runArtifactChecker` property, the Artifact Checker task must be executed explicitly.
- Hide compilation warnings in the adp and compileJava tasks.

Fixed

- Disable the warning on non-compatible version for Maven client.
- Fix loading new dependency when the `build.gradle.kts` file is updated.
- Fix too long classpath error when running the simulator on Windows.

4.26.15 [0.3.0] - 2022-12-09

Added

- Add feature to avoid loading the VEE Port when there is no test.
- Add the auto assembling project for `runOnSimulator` and `debugOnSimulator` tasks.
- Add the opportunity disable custom conflict resolution rules.
- Add the plugin `com.microej.gradle.library` to build an Addon Library.
- Generate and publish the Java sources jar.
- Generate and publish the Javadoc jar.
- Publish `README.md`, `CHANGELOG.md` and `LICENSE.txt` files if they exist in the project.
- Suffix version with timestamp when it ends with "-RC".
- Make the build fail if a direct dependency is resolved with a higher minor version than the one declared.
- Add the `checkModule` task to check compliance of the module with MicroEJ rules.

- Add the execution of tests on the simulator.
- Add support for Mac M1.
- Build the plugin in Java 11.
- Add test to ensure that the dependencies are topologically sorted.

Changed

- Remove automatic version conversion.
- Rename the Application plugin to `com.microej.gradle.application`.
- Change the publication plugin to publish Maven modules instead of Ivy modules.
- Use Ant Java API to launch the simulator to avoid requiring an Ant installation.
- Rename the `runOnSim` and `debugOnSim` tasks to `runOnSimulator` and `debugOnSimulator`.
- Use JDT compiler instead of javac.
- Isolate functional tests to keep a quick build.

Fixed

- VEE Port not reloaded when referenced by `veePortDirPath` and the VEE Port source folder is updated.
- Set Java source and target version to be recognized by IDEs.
- Make `processResources` task implicitly depend on ADP task to fix failures during `runOnSimulator`.

4.26.16 [0.2.0] - 2022-05-17

Changed

- Make the build fails when an ADP raises errors.
- Convert build scripts from `Groovy` to `Kotlin`.

4.26.17 [0.1.0] - 2022-05-03

Added

- Add the capability to load the platform from dependencies.
- Add the task `debugOnSim` to execute the application in debug mode in the simulator.
- Publish the sources jar of the plugin.

Fixed

- Extract ADP classpath JAR files into OS temp dir to avoid error on cleaning because of locks.

4.27 Migration Notes

4.27.1 From 0.14.0 to 0.15.0

This section applies if MicroEJ SDK 6 `0.15.0` is used on a project that was created using MicroEJ SDK 6 `0.14.0` or lower.

Unification of VEE dependency declaration

The `microejVeePort` configuration, used to define a VEE Port, and the `microejKernel` configuration, used to define a Kernel, have been unified into the `microejVee` configuration.

- To use a VEE Port or a Kernel published in an artifact repository, declare a Module dependency in the `build.gradle.kts` file:

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

- To use a VEE Port directory available locally, declare a file dependency in the `build.gradle.kts` file:

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

- To use a VEE Port archive available locally, declare a file dependency in the `build.gradle.kts` file:

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

- To use a Kernel Virtual Device and Executable available locally, declare a file dependency in the `build.gradle.kts`:

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

4.27.2 From 0.11.1 to 0.12.0

This section applies if SDK 6 `0.12.0` is used on a project that was created using SDK 6 `0.11.1` or lower.

Use of File Dependencies to Define a Local VEE Port or a Kernel Executable

The `veePortPath` and the `kernelFile` properties have been replaced by file dependencies.

- To use a VEE Port archive available locally, declare a file dependency in the `build.gradle.kts` file, with the `microejVeePort` configuration:

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

- To use a VEE Port directory available locally, declare a file dependency in the `build.gradle.kts` file, with the `microejVeePort` configuration:

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

- To use a kernel Virtual Device and Executable available locally, declare a file dependency in the `build.gradle.kts` file, with the `microejKernel` configuration:

```
dependencies {
   microejKernel(files("C:\\path\\to\\my\\kernel\\executable.out", "C:\\path\\to\\my\\
↪kernel\\virtual\\device"))
}
```

4.27.3 From 0.10.0 to 0.11.0

This section applies if SDK 6 `0.11.0` is used on a project that was created using SDK 6 `0.10.0` or lower.

Gradle mechanism usage for Multiple VEE Ports Support

Using multiple VEE Ports in a project uses Gradle mechanism now instead of relying on in-house feature. This implies: - the `veePortPaths` property has been renamed to `veePortPath` and accepts a String value:

```
microej {
   veePortPath = "C:\\path\\to\\my\\veePort\\source"
}
```

- the `kernelFiles` property has been renamed to `kernelFile` and accepts a String value:

```
microej {
   kernelFile = "C:\\path\\to\\my\\kernel\\file"
}
```

Refer to the *How To Build an Executable With Multiple VEE Ports* section to learn how to support multiple VEE Ports using the Gradle mechanisms.

4.27.4 From 0.8.0 to 0.9.0

This section applies if SDK 6 `0.9.0` is used on a project that was created using SDK 6 `0.8.0` or lower.

Merge of the veePortDirs and veePortFiles properties

The build properties `veePortDirs` and `veePortFiles` have been merged into a single property `veePortPaths`. To define a local VEE Port, set the build property `veePortPaths` in the `microej` configuration block to the path of the VEE Port file (`.zip` or `.vde`) or to the source folder of the VEE Port:

```
microej {
  veePortPaths = listOf("C:\\path\\to\\my\\veePort\\source")
}
```

The `veePortPaths` property is defined as a list in order to provide multiple VEE Port files or source folders if it is needed:

```
microej {
  veePortPaths = listOf("C:\\path\\to\\my\\veePort1\\source", "C:\\path\\to\\my\\veePort2\\
↪file.zip")
}
```

CHAPTER

FIVE

APPLICATION DEVELOPER GUIDE

5.1 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.

A MicroEJ Sandboxed Application is a MicroEJ Application that can run over a Multi-Sandbox Executable. 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.

5.2 MicroEJ Runtime

5.2.1 Language

MicroEJ allows to develop Applications in the Java® Language Specification version 7 with *some limitations*, and supports code extensions written in *JavaScript*.

Basically, Java source code is compiled by the Java compiler[1] into the binary format specified in the JVM specification[2]. This binary code is linked by a tool named *SOAR* before execution: `.class` files and some other application-related files (see *Classpath* chapter) are linked to produce the final binary file that the *Core Engine* will execute.

[1] The JDT compiler from the Eclipse IDE.
[2] Tim Lindholm & Frank Yellin, The Java™ Virtual Machine Specification, Second Edition, 1999

Note: When opened in the SDK, make sure that the Compiler Compliance Level of your project is set to 1.7 to ensure the bytecode produced by the Java compiler is compatible with MicroEJ. The Compliance Level can be changed from the menu: Window > Preferences > Java > Compiler .

5.2.2 Core Libraries

This section describes the core libraries which make up the runtime. Theses Foundation Libraries are tightly coupled with the Core Engine.

Embedded Device Configuration (EDC)

The Embedded Device Configuration specification defines the minimal standard runtime environment for embedded devices.

This module is always required in the build path of an Application project; and all others libraries depend on it. This library provides a set of options. Refer to the chapter *Standalone Application Options* which lists all available options.

It defines all default API packages:

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

Documentation	Link
Java APIs	https://repository.microej.com/javadoc/microej_5.x/libraries/edc-1.3-api/
Module	https://repository.microej.com/modules/ej/api/edc/

Use

The EDC API Module must be added to the project build file of the Application Project:

Gradle (build.gradle.kts)

MMM (module.ivy)

```
implementation("ej.api:edc:1.3.5")
```

```
<dependency org="ej.api" name="edc" rev="1.3.5"/>
```

Beyond Profile (BON)

This profile 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.

Documentation	Link
Java APIs	https://repository.microej.com/javadoc/microej_5.x/apis/ej/bon/package-summary.html
Specification	https://repository.microej.com/packages/ESR/ESR-SPE-0001-BON-1.2-G.pdf
Module	https://repository.microej.com/modules/ej/api/bon/

Use

Add the following dependency to the project build file of the Application Project to use the BON API Module:

Gradle (build.gradle.kts)

MMM (module.ivy)

```
implementation("ej.api:edc:1.3.5")
```

```
<dependency org="ej.api" name="edc" rev="1.3.5"/>
```

Simple Native Interface (SNI)

SNI provides a simple mechanism for implementing native Java methods in the C language.

SNI allows you to:

- Call a C function from a Java method.
- Access an Immortal array in a C function (see the *Beyond Profile (BON)* to learn about immortal objects).

SNI does not allow you to:

- Access or create a Java object in a C function (except byte arrays).
- Access Java static variables in a C function.
- Call Java methods from a C function.

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

Documentation	Link
Java APIs	https://repository.microej.com/javadoc/microej_5.x/apis/ej/sni/package-summary.html
Specification	https://repository.microej.com/packages/ESR/ESR-SPE-0012-SNI_GT-1.2-I.pdf
Module	https://repository.microej.com/modules/ej/api/sni/

Please refer to *Simple Native Interface (SNI)* section for more details.

Kernel & Features (KF)

The Kernel & Features semantic (KF) extends the runtime for managing Multi-Sandboxed Applications.

Please refer to the *Kernel & Features Specification* for more details, the *Multi-Sandbox capability* of the Core Engine and more generally the *Kernel Developer Guide* chapter.

5.2.3 Scheduler

The Core Engine features a Green Threads model. The semantic is as follows:

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

Threads stacks automatically adapt their sizes according to the thread requirements: once a thread terminates, its associated stack is reclaimed, freeing the corresponding RAM memory.

5.2.4 Garbage Collector

The Core Engine includes a state-of-the-art memory management system, the Garbage Collector (GC). It manages a bounded piece of RAM memory, devoted to the 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 Application keep running.

See also *Garbage Collector options* for more details.

5.2.5 Limitations

Primitive Types

Getting a Class instance of a primitive type is not supported:

- `boolean.class`,
- `byte.class`,
- `char.class`,
- `short.class`,
- `int.class`,
- `long.class`,
- `float.class`,
- `double.class`.

On Architecture `8.x`, you will get the following dedicated error message:

```
Unsupported access to the Class instance of a primitive type (found 'boolean.class' in␣
↪method 'com.mycompany.MyClass.myMethod()void')
```

On Architecture `7.x` you will get the following default error message:

[3] This protocol raises the priority of a thread that is holding a monitor needed by a higher-priority thread, to the priority of that higher-priority thread (until exiting the monitor).

```
No such field TYPE at com/mycompany/MyClass.myMethod()V.
```

5.3 SOAR

This chapter describes SOAR capabilities and optimizations from the Application developer's point of view. To get more details on its internal structure, please refer to *SOAR Build Phases* section.

5.3.1 Java Symbols Encoding

Java symbols are any of package, type, method, field, or local names. In `.class` files, they are encoded in `UTF-8`. However, SOAR only supports Java symbols composed of characters that can be stored on 8 bits (unsigned byte).

This is typically the case of ISO-8859-X encoding family.

If you try to build an Application that includes an unsupported Java symbol you will get the following error:

```
Unsupported Java symbol XXX in file YYY. A character cannot be stored on an unsigned byte.
```

Note: Classpath `*.list` files are standard Java properties files that are encoded in `ISO-8859-1` (Latin-1). If you need to refer to a Java Symbol that contains a character out of this charset, you must declare the character using the `\uHHHH` notation where HHHH is the hexadecimal index of the character in the Unicode character set.

5.3.2 Class Initialization Code

SOAR complies with the deterministic class initialization (`<clinit>`) order specified in *[BON] specification*. The application is statically analyzed from its entry points in order to generate a clinit dependency graph. The computed clinit sequence is the result of the topological sort of the dependency graph. An error is thrown if the clinit dependency graph contains cycles.

A clinit map file (ending with extension `.clinitmap`) is generated beside the SOAR object file. It describes for each clinit dependency:

- the types involved
- the kind of dependency
- the stack calls between the two types

In case of complex clinit dependencies graph, the SOAR may detect static cycles (circular dependencies) and fail with an error. In such case, you have to manually cut-off the cycles, by providing the explicit clinit dependencies.

Explicit clinit dependencies are declared in XML files ending with the `.clinitdesc` extension, at the root of a library or application classpath.

The file has the following format:

```
<?xml version='1.0' encoding='UTF-8'?>
<clinit>
    <type name="T1" depends="T2"/>
</clinit>
```

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

5.3.3 Method Devirtualization

Method devirtualization consists of transforming a virtual method call to a direct method call when possible. A virtual method call is a call to a non-private instance method declared either in an interface or in a class. The Core Engine determines the proper method to call at runtime depending on the actual class of the object. A call to a constructor or a private method is already optimized as a direct method call by the Java compiler.

SOAR automatically optimizes a virtual method call to a direct method call if there is one and only one embedded implementation method.

Note: SOAR generates the list of the embedded methods in the *SOAR Information File*.

5.3.4 Method Inlining

Method inlining consists of replacing a direct method call with the content of the method. This avoids the creation of a new stack frame context, which can be slower than executing the code itself. Method inlining is transitively applied from leaf to root methods.

The following method code patterns are inlined:

- empty code after processing *assertions* and *if code removal*.
- call to a constructor with no parameters.
- call to a private method with no parameters.
- call to a static method with no parameters, if and only if the caller is also a static method.

Note: Method inlining is performed after *method devirtualization*, so a virtual method call will be inlined if there is a unique embedded implementation method that matches one of the inlined method code patterns.

5.3.5 Binary Code Verifier

The Binary Code Verifier is the tool that scrutinizes the bytecode instructions for adherence to strict rules and constraints. This process is crucial for preventing runtime errors, security vulnerabilities, and unexpected behavior. It ensures that code loaded by the SOAR is in a consistent state before being linked. Consequently, this guarantees the safe execution of the code by the Core Engine.

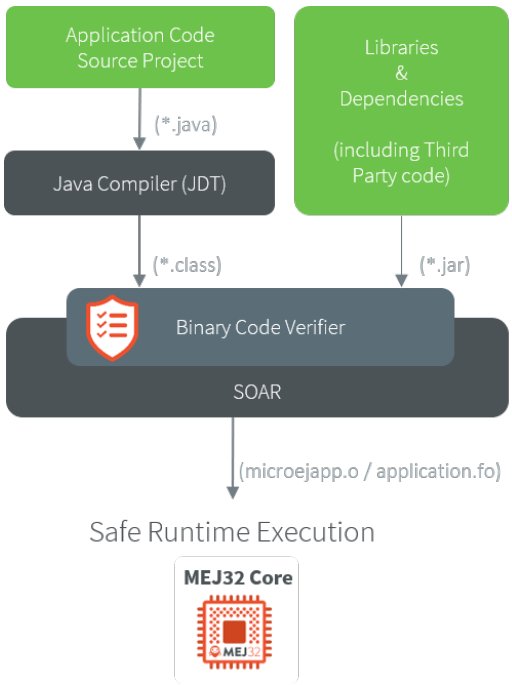


Fig. 1: Application Build Flow with Binary Code Verifier

The Binary Code Verifier performs tasks including:

- Type Checking: Verifying that variables and operands are used in a manner consistent with their declared data types, preventing type-related errors at runtime.
- Bytecode Structure: Ensuring the bytecode is well-formed and follows the structure required by the JVM, which helps prevent memory corruption and crashes.
- Stack Management: Checking that the operand stack used for calculations and evaluations is properly managed to prevent stack overflows or underflows.
- Access Control: Verifying that class accesses and method invocations adhere to Java's access control rules, ensuring data encapsulation and security.
- Exception Handling: Validating that exception handlers are correctly defined and that exceptions are caught and handled appropriately.
- Control Flow: Analyzing the flow of control within bytecode to detect anomalies in loops, branches, and jumps that could lead to program instability.

A default implementation, derived from the Apache BCEL Project, is included in the SOAR. If you wish to integrate an alternative implementation, contact our support team for access to the SOAR interface API and integration instructions.

Note: The Binary Code Verifier is enabled by default when building a Sandboxed Application, and disabled by

default when building a Standalone Application. See *Option(checkbox): Enable Bytecode Verifier* for more details.

5.4 SOAR Output Files

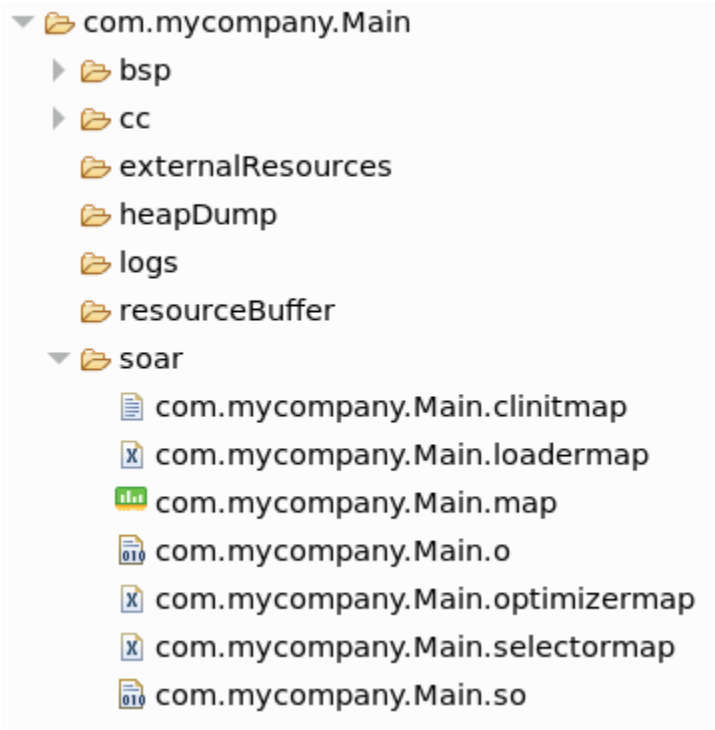
When building a Standalone Application, multiple files are generated next to the ELF executable file.

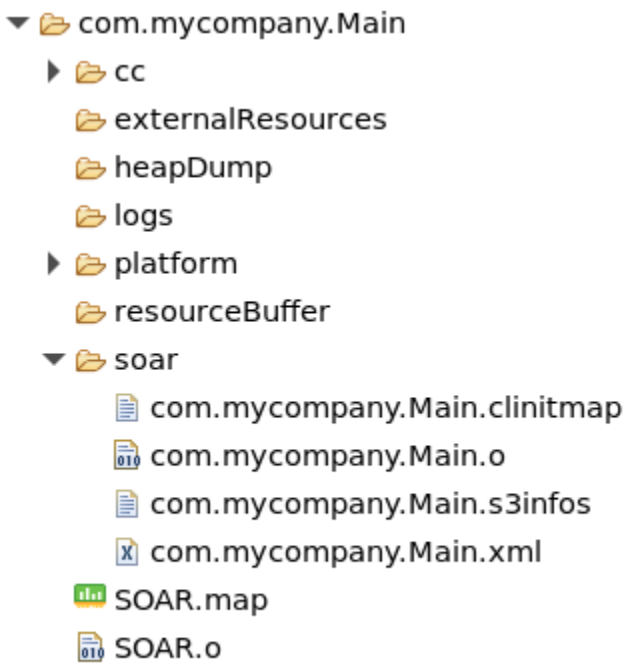
5.4.1 Launch Output Folder

Using a *MicroEJ Application Launch*, the files are generated in a folder which is named like the main type and which is located in the output folder specified in the run configuration.

Build Output Files (Architecture `8.x`)

Build Output Files (Architecture `7.x`)





5.4.2 Published Module Files

After *building* the Standalone Application, the published module contains the following main files:

- `[name]-[version].out` : Firmware (ELF Executable)
- `[name]-[version].zip` : Virtual Device
- `[name]-[version]-workingEnv.zip` : Build intermediate files, including the content of the launch output Folder

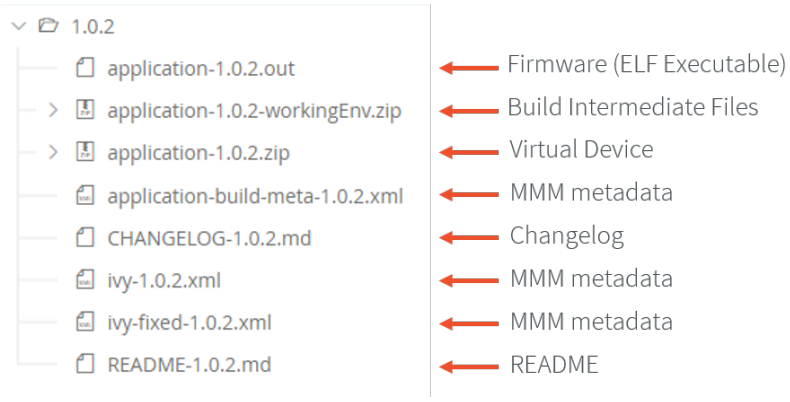


Fig. 2: Published Standalone Application Module Files

5.4.3 The SOAR Map File

The `.map` file lists every embedded symbol of the application (section, Java class or method, etc.) and its size in ROM or RAM. Since Architecture `8.x`, this file is called `<main class>.map`. It was formerly named `SOAR.map` for Architecture `7.x`. This file can be opened using the *Memory Map Analyzer*.

The embedded symbols are grouped into multiple categories. For example, the `Object` class and its methods are grouped in the `LibFoundationEDC` category. For each symbol or each category, you can see its size in ROM (`Image Size`) and RAM (`Runtime Size`).

The SOAR groups all the Java strings in the same section, which appears in the `ApplicationStrings` category. The same applies to the static fields (`Statics` category), the types (`Types` category), and the class names (`ClassNames` category).

5.4.4 The SOAR Information File

The SOAR information file contains details on the embedded elements of an application.

Since Architecture `8.x`, information are dispatched in separate files which are related to *SOAR build phases*:

- `soar/<main class>.loadermap`: generated by the SOAR Resolver. It provides details on files and resources that have been loaded from the *Application Classpath*.
- `soar/<main class>.selectormap`: generated by the SOAR Resolver. It provides details about the elements that have been included in the application.
- `soar/<main class>.optimizermap`: generated by the SOAR Optimizer. It provides details about the elements that have been linked in the application.

Each of these files can be opened with an XML editor. The following table describes the information that can be retrieved with their file location.

The SOAR Information File (Architecture `8.x`)

The SOAR Information File (Architecture `7.x`)

Information	XML Location (tag > subtag [attribute=value])	File Location
Classpath	classpath	soar/<main class>.loadermap
Resources	resources	soar/<main class>.loadermap
External resources	external_resources	soar/<main class>.loadermap
System properties	properties	soar/<main class>.loadermap
Constants	constants	soar/<main class>.loadermap
Immutables	N/A	N/A
Interned strings	strings	soar/<main class>.selectormap
Class initialization order	clinit	soar/<main class>.selectormap
Types	types	soar/<main class>.selectormap
Number of types	types>[nb]	soar/<main class>.selectormap
Number of concrete classes	types[nbConcreteClasses]	soar/<main class>.selectormap
Number of abstract classes	types[nbAbstractClasses]	soar/<main class>.selectormap
Number of interfaces	types[nbInterfaces]	soar/<main class>.selectormap
Number of arrays	types[nbArrays]	soar/<main class>.selectormap
Class instance size (in bytes)	types>type[instanceSize]	soar/<main class>.optimizermap
Type *embeds its name*	types>type[hasRuntimeName = true]	soar/<main class>.selectormap
Type *is exposed as Kernel API*	types>type[api=true]	soar/<main class>.selectormap
Number of reference fields in a class	types>type[nbReferenceFields]	soar/<main class>.optimizermap
Methods	methods	soar/<main class>.selectormap
Method code size (in bytes)	methods>method[codesize]	soar/<main class>.optimizermap
Method *is inlined*	methods>method[inlined=true]	soar/<main class>.optimizermap
Method *is exposed as Kernel API*	methods>method[api=true]	soar/<main class>.selectormap
Statics fields	statics	soar/<main class>.selectormap

Information	XML tag>subtag[attribute=value]	File
Classpath	classpath	soar/<main class>.xml
Resources	selected_resources	soar/<main class>.xml
External resources	external_resources	soar/<main class>.xml
System properties	java_properties	soar/<main class>.xml
Constants	constants	soar/<main class>.xml
Immutables	selected_immutables	soar/<main class>.xml
Interned strings	selected_internStrings	soar/<main class>.xml
Class initialization order	clinit_order	soar/<main class>.xml
Types	selected_types	soar/<main class>.xml
Number of types	selected_types[nb]	soar/<main class>.xml
Number of concrete classes	selected_types[nbConcreteClasses]	soar/<main class>.xml
Number of abstract classes	selected_types[nbAbstractClasses]	soar/<main class>.xml
Number of interfaces	selected_types[nbInterfaces]	soar/<main class>.xml
Number of arrays	selected_types[nbArrays]	soar/<main class>.xml
Class instance size (in bytes)	selected_types>type[instanceSize]	soar/<main class>.xml
Type *embeds its name*	required_types	soar/<main class>.xml
Type *is exposed as Kernel API*	selected_types>type[api=true]	soar/<main class>.xml
Number of reference fields in a class	selected_types>type[nbReferenceFields]	soar/<main class>.xml
Methods	selected_methods	soar/<main class>.xml
Method code size (in bytes)	selected_methods>method[codeSize]	soar/<main class>.xml
Method *is inlined*	selected_methods>method[inlined=true]	soar/<main class>.xml
Method *is exposed as Kernel API*	selected_methods>method[api=true]	soar/<main class>.xml
Statics fields	selected_static_fields	soar/<main class>.xml

5.5 Virtual Device

The Virtual Device includes the same custom MicroEJ Core, libraries, and pre-installed Applications as the real device. The Virtual Device allows developers to run their applications either on the Simulator, or directly on the real device through local deployment.

The Simulator runs a mockup board support package (BSP Mock) that mimics the hardware functionality. An application on the Simulator is run as a Standalone Application.

Before an application is locally deployed on device, the SDK ensures that it does not depend on any API that is unavailable on the device.

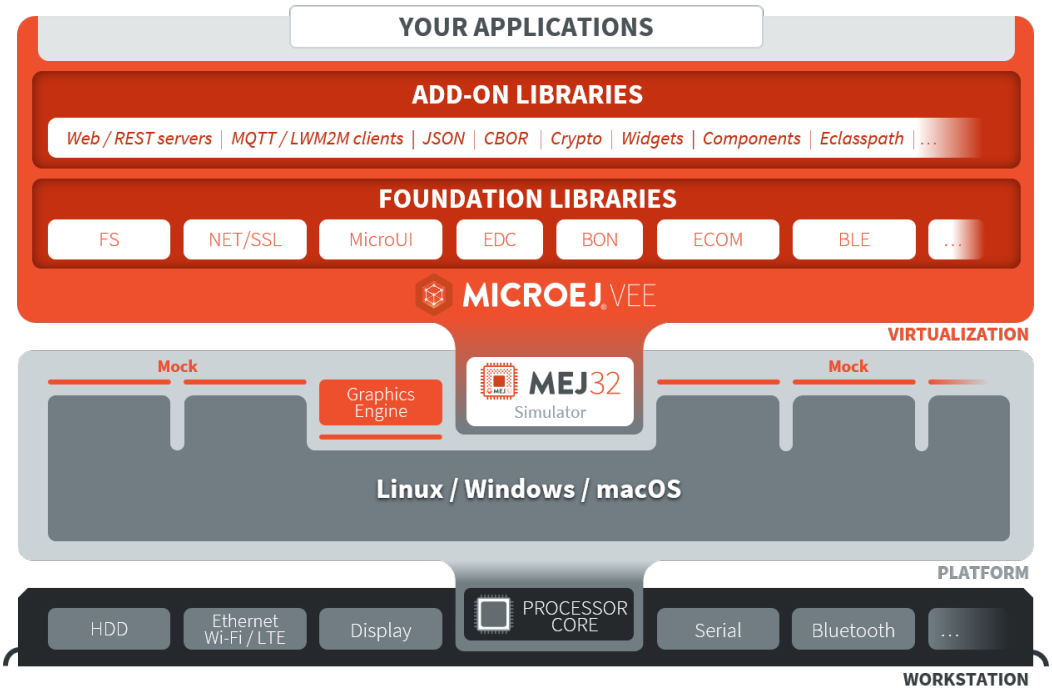


Fig. 3: MicroEJ Virtual Device Architecture

5.6 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.

5.6.1 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).

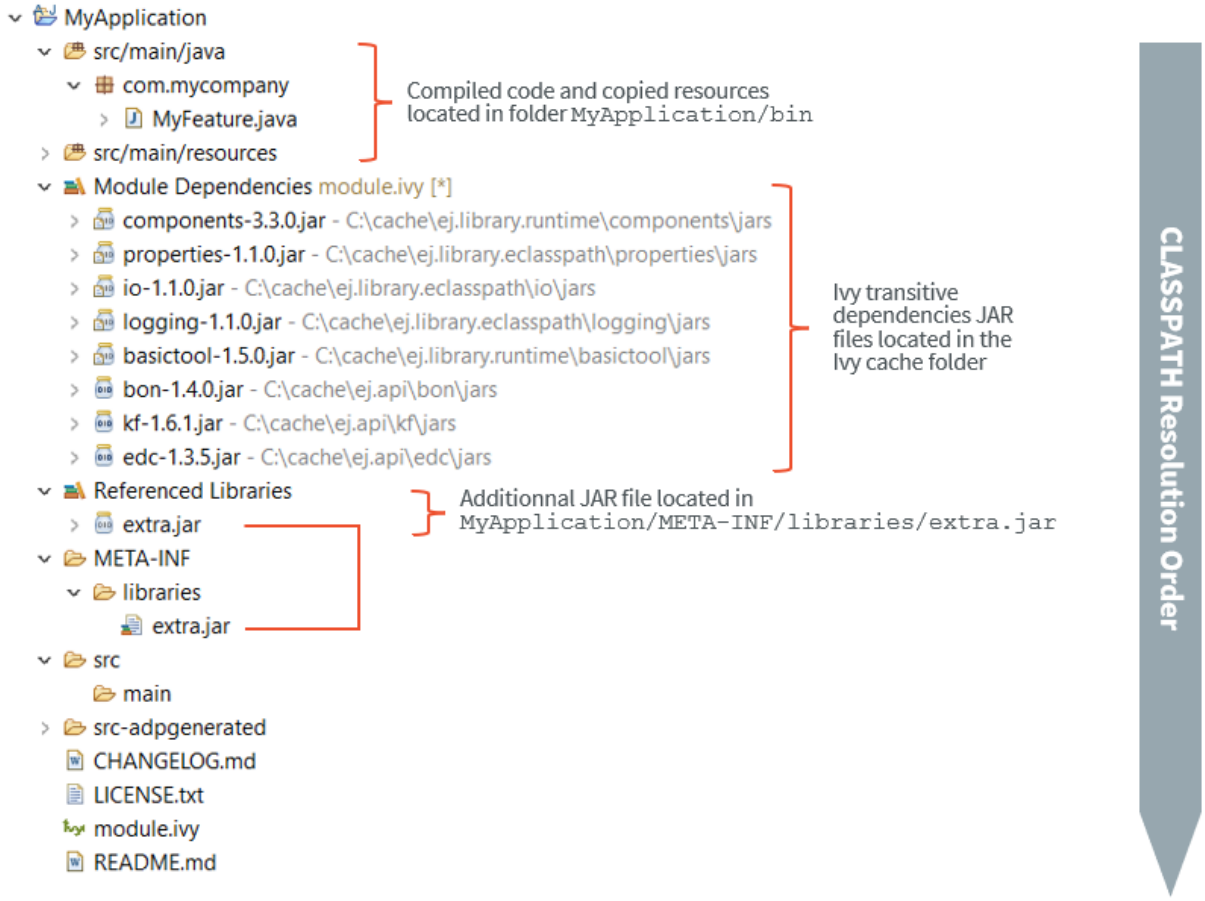


Fig. 4: MicroEJ Application Classpath Mapping

Note: For Sandboxed Applications, when a library cannot be *added as a dependency* (because it is not available in a repository for example), its JAR file can be directly added in the `META-INF/libraries` folder of the Application project. It is then automatically added in the compilation classpath and is available for the Application.

5.6.2 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.

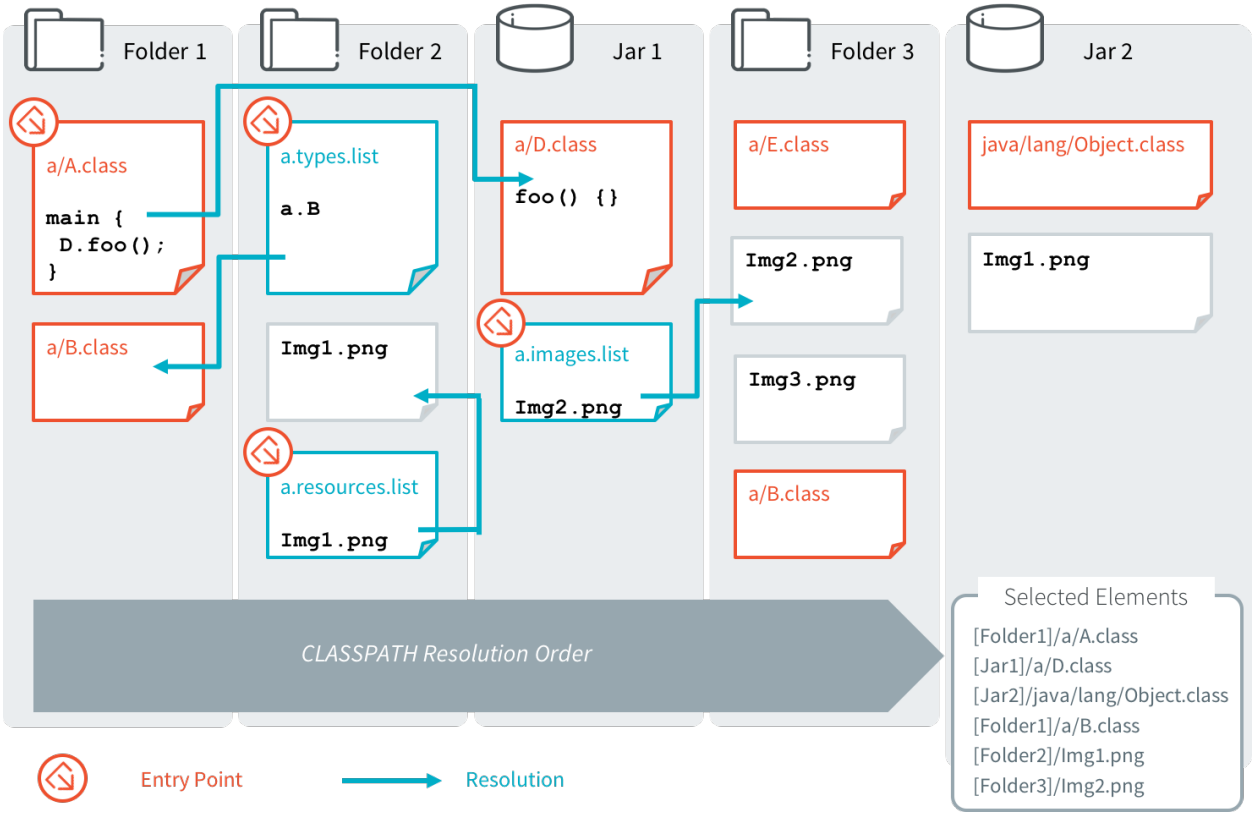


Fig. 5: Classpath Load Principle

5.6.3 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*;
- Immutables Object data files, described in Section *Immutable Objects*;
- Raw Resources, Images, Fonts and Native Language Support (NLS) described in *Application Resources*;
- `*.[extension].list` files, declaring contents to load. Supported list file extensions and format is specific to declared application contents and is described in the appropriate section.

At source level, Java types are stored in `src/main/java` folder of the *module project*, any other kind of resources and list files are stored in the `src/main/resources` folder.

Application Entry Points

MicroEJ Application entry point declaration differs depending on the application kind:

- In case of a Standalone Application, it is a class that contains a `public static void main(String[])` method, declared using the option `application.main.class`.
- In case of a Sandboxed Application, it is a class that implements ej.kf.FeatureEntryPoint, declared using the `entryPoint` property in the `.kf` file in the `src/main/resources/` folder.

Types

MicroEJ types (classes, interfaces) are compiled from source code (`.java`) to classfiles (`.class`). When a type is loaded, all types dependencies found in the classfile are loaded (transitively).

A type can be declared as a *Required type* in order to enable the following usages:

- to be dynamically loaded from its name (with a call to Class.forName(String));
- to retrieve its fully qualified name (with a call to Class.getName()).
- when *Tiny-Sandbox* capability is enabled, to retrieve its package (with a call to Class.getPackage()).

A type that is not declared as a *Required type* may not have its fully qualified name (FQN) embedded. Its FQN can be retrieved using the stack trace reader tool (see *Stack Trace Reader*).

Required Types are declared in MicroEJ Classpath using `*.types.list` files. The file format is a standard Java properties file, each line listing the fully qualified name of a type. Example:

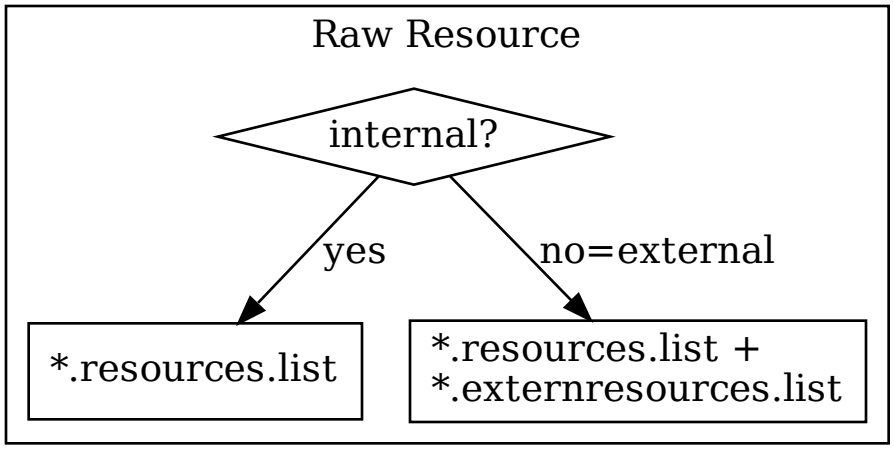
```
# The following types are marked as MicroEJ Required Types
com.mycompany.MyImplementation
java.util.Vector
```

Resources

Resources are binary files that need to be embedded by the application.

Note: For more details on all supported resources types, please refer to *Application Resources* chapter.

Raw resources are resources that can be dynamically retrieved with a call to java.lang.Class.getResourceAsStream(String). Raw Resources are declared in `*.resources.list` files (**and** in `*.externresources.list` for external resources, see *Application Resources*).



The file format is a standard Java properties file, each line is a relative `/` separated name of a file in MicroEJ Classpath to be embedded as a resource. Example:

```
# The following resource is embedded as a raw resource
com/mycompany/MyResource.txt
```

A resource is 4-bytes aligned in memory by default. Starting from *Architecture 8.0.0*, it is possible to modify the alignment constraint. Example:

```
# The following resource is linked to a 32-bytes aligned address in memory.
com/mycompany/MyResource.txt:32
```

Note: If a Resource is declared multiple times in the classpath, the alignment constraint with the highest value is used. If the alignment constraints are specific to the target, it is recommended to only declare them in the Application project instead of libraries.

Immutable Objects

Immutables objects are regular read-only objects that can be retrieved with a call to ej.bon.Immutables.get(String). Immutables objects are declared in files called *immutable objects data files*, which format is described in the *[BON] specification*. Immutables objects data files are declared in MicroEJ Classpath using `*.immutables.list` files. The file format is a standard Java properties file, each line is a `/` separated name of a relative file in MicroEJ Classpath to be loaded as an Immutable objects data file. Example:

```
# The following file is loaded as an Immutable objects data files
com/mycompany/MyImmutables.data
```

System Properties

System Properties are key/value string pairs that can be accessed with a call to System.getProperty(String).

System Properties are defined when building a *Standalone Application*, by declaring `*.properties.list` files in MicroEJ Classpath.

The file format is a standard Java properties file. Example:

Listing 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 *Standalone Application 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 System Property Definition as Application Option

```
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`).

Note: When building a *Sandboxed Application*, `*.properties.list` files found in MicroEJ Classpath are silently skipped.

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'")
}
```

Please mind that `Constants.getXXX` must be inlined in the `if` condition to take effect. The following piece of code will not remove the code:

```
static final boolean MY_CONSTANT = Constants.getBoolean("com.mycompany.myconstantkey");

...

if(MY_CONSTANT){
   System.out.println("this code will not be removed when MY_CONSTANT is resolved to 'false'
↪")
}
```

Note: In *Multi-Sandbox* environment, constants are processed locally within each context. In particular, constants defined in the Kernel are not propagated to *Sandboxed Applications*.

5.7 Application Resources

An Application resource is the contents of a file identified its relative path from the Application *classpath*.

An Application resource is one of the following type:

- Raw Resource,
- Image,
- Font,
- Internationalized Message (Native Language Support).

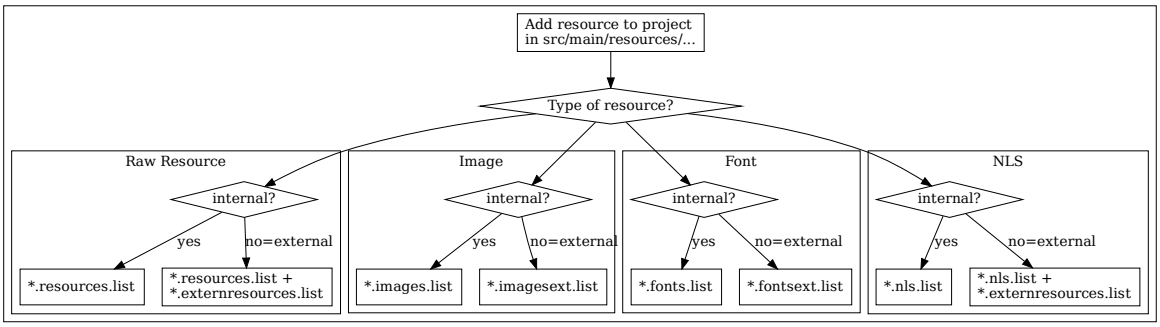
The resource may be stored in RAM, flash, or external flash; and it is the responsibility of the Core Engine and/or the BSP to retrieve and load it.

There are two ways to store resources:

- Internal resource: The resource is taken into consideration during the Application build. The SOAR step loads the resource and copies it into the same C library as the Application. Like the Application, the resource is linked into the CPU address space range (internal device memories, external parallel memories, etc.).
- External resource: The resource is not taken into consideration during the Application build. It is the responsibility of the BSP project to manage external resources. The resource is often programmed outside the CPU address space range (storage media like SD card, serial NOR flash, EEPROM, etc.).

  The BSP must implement the proper Low Level API (LLAPI) C functions: `LLEXT_RES_impl.h`. See *External Resources Loader* for more information on the implementation.

All resources must be added in the project, usually in `src/main/resources/...` folder. All resources must be declared in the appropriate `*.list` files depending on the type (raw, image, font, NLS) and the storage location (internal or external). The following figure summarized how to declare resources:



For more details on how to use Application resources, refer to the following dedicated sections:

- *Raw Resource*
- *Image*
- *Font*
- *Internationalized String (Native Language Support)*

5.8 Standalone Application

5.8.1 Introduction

A Standalone Application is a Java Application directly linked to the C code to produce an Executable. Such an application must define a main entry point (i.e., a class containing a `public static void main(String[])` method).

The following figure shows the general process of building a Standalone Application to an Executable.

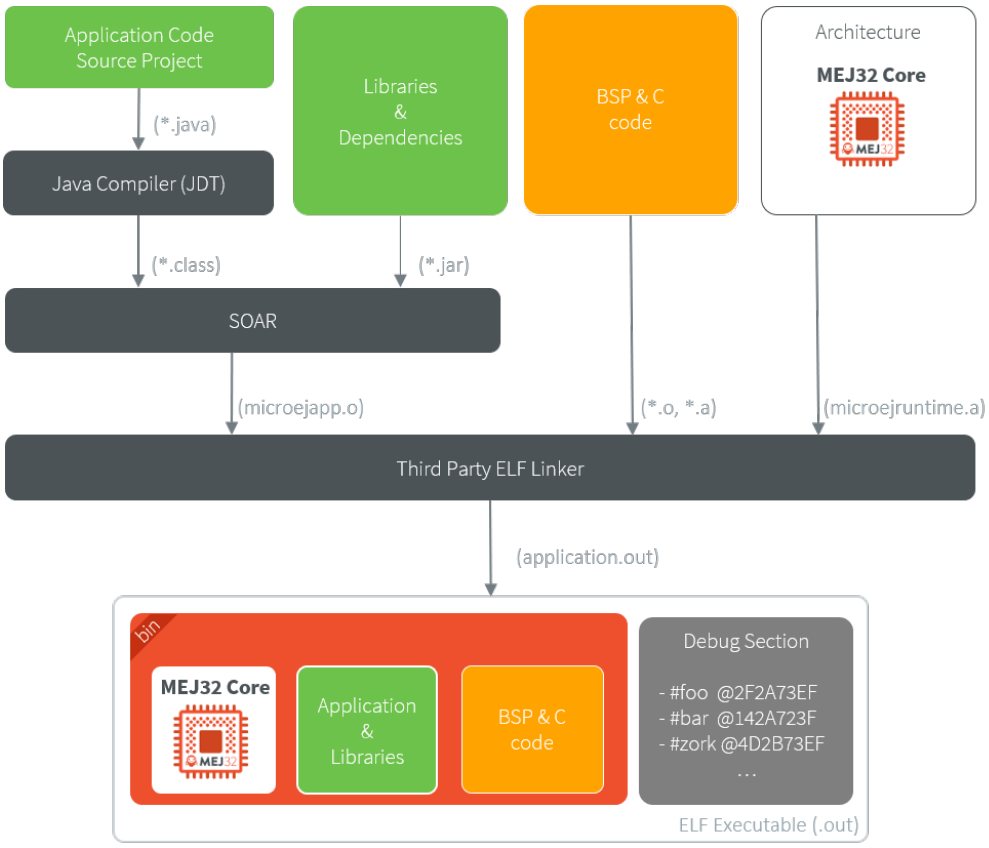


Fig. 6: Standalone Application Link Flow

5.8.2 Standalone Application Options

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

- Memory Allocation options (e.g., set the 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 the latest Architecture version. Please consult the appropriate Pack(s) documentation for options related to other Foundation Libraries (MicroUI, NET, SSL, FS, ...) integrated to your VEE Port.

Notice that some options may not be supported by your VEE Port, in the following cases:

- Option is specific to the Core Engine capability (*tiny*/*mono*/*multi*).
- Option is specific to the target (Core Engine or Simulator).
- Option has been introduced in a newer Architecture version.

5.8.3 Defining an Option with SDK 6

With the SDK 6, the Applications options can be defined in a properties file located in the `configuration` folder of the project. Usually, the options are defined in a file named `common.properties`, but all properties files located in this folder are loaded, no matter what their name is.

To set an option in a properties file, open the file in a text editor and add a line to set the desired option to the desired value, for example:

```
soar.generate.classnames=false
```

5.8.4 Defining an Option with SDK 5 or lower

With the SDK 5 or lower, a Standalone Application option can be defined either from a launcher or from a properties file. It is also possible to use both together. Each MicroEJ Architecture and MicroEJ Pack option comes with a default value, which is used if the option has not been set by the user.

Using a Launcher

To set an option in a launcher, perform the following steps:

1. In the SDK, select `Run` > `Run Configurations...` menu,
2. Select the launcher of the application under `MicroEJ Application` or create a new one,
3. Select the `Configuration` tab,
4. Find the desired option and set it to the desired value. If the option does not appear in the page, there are two cases: - the option has been introduced in a newer Architecture version, - the option is an advanced option. It is set using a system property in the `JRE Tab`. See the *JRE Tab* section for more details.

It is recommended to index the launcher configuration to your version control system. To export launcher options to the filesystem, perform the following steps:

1. Select the `Common` tab,
2. Select the `Shared file:` option and browse the desired export folder,
3. Press the `Apply` button. A file named `[launcher_configuration_name].launch` is generated in the export folder.

Using a Properties File

Options can be also be defined in properties files.

When a Standalone Application is built using the `firmware-singleapp` *skeleton*, options are loaded from properties files located in the `build` folder at the root of the project.

The properties files are loaded in the following order:

1. Every file matching `build/sim/*.properties`, for Simulator options only (Virtual Device build). These files are optional.
2. Every file matching `build/emb/*.properties`, for Device options only (Firmware build). These files are optional.
3. Every file matching `build/*.properties`, both for Simulator and Device options. At least one file is required.

Usually, the `build` folder contains a single file named `common.properties`.

In case an option is defined in multiple properties files, the option of the first loaded file is taken into account and the same option defined in the other files is ignored (a loaded option cannot be overridden).

The figure below shows the expected tree of the `build` folder:

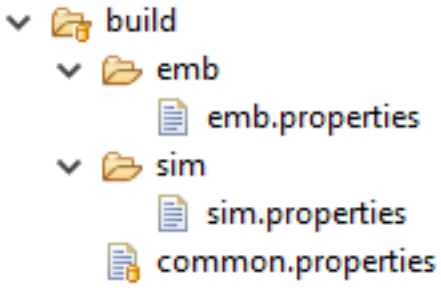


Fig. 7: Build Options Folder

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 the SDK, select `Run` > `Run Configurations...`,
2. Select the launcher of the application,
3. Select the `Execution` tab,
4. Under `Option Files`, press the `Add...` button,
5. Browse the `sim.properties` file for Simulator or the `emb.properties` file for Device (if any) and press `Open` button,
6. Add the `common.properties` file and press the `Open` button.

Note: An option set in a properties file can not be modified in the `Configuration` tab. Options are loaded in the order the properties files are added (you can use `Up` and `Down` buttons to change the file order). In `Configuration`

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

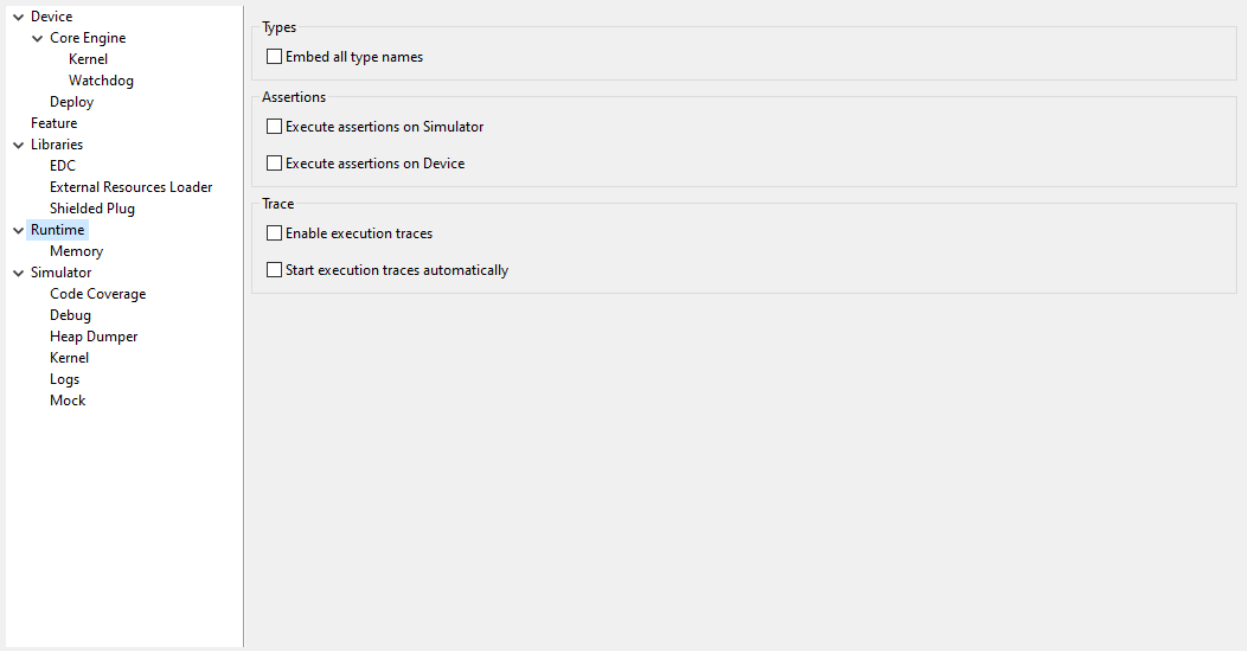
Generating a Properties File

In order to export options defined in a `.launch` file to a properties file, perform the following steps:

1. Select the `[launcher_configuration_name].launch` file,
2. Select File > Export > MicroEJ > Launcher as Properties File ,
3. Browse the desired output `.properties` file,
4. Press the Finish button.

Warning: The Simulator uses some system properties to configure internal memory limits. See *Group: Advanced Simulation Options* for more information.

5.8.5 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`

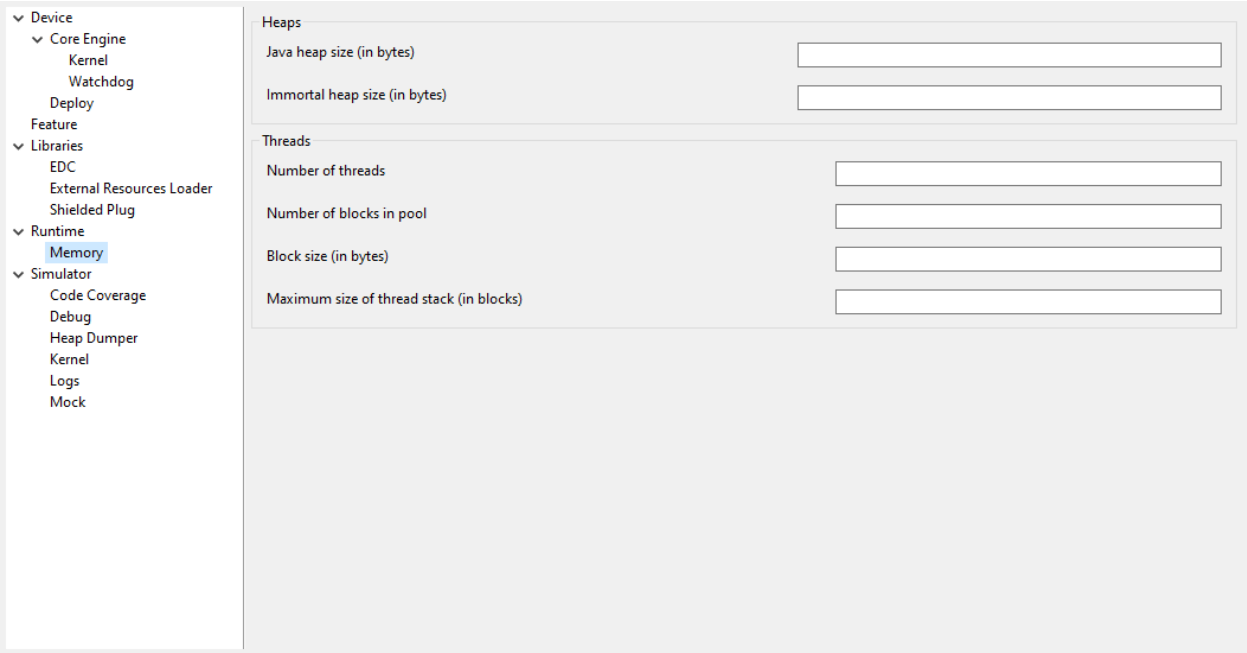
Default value: `false`

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)

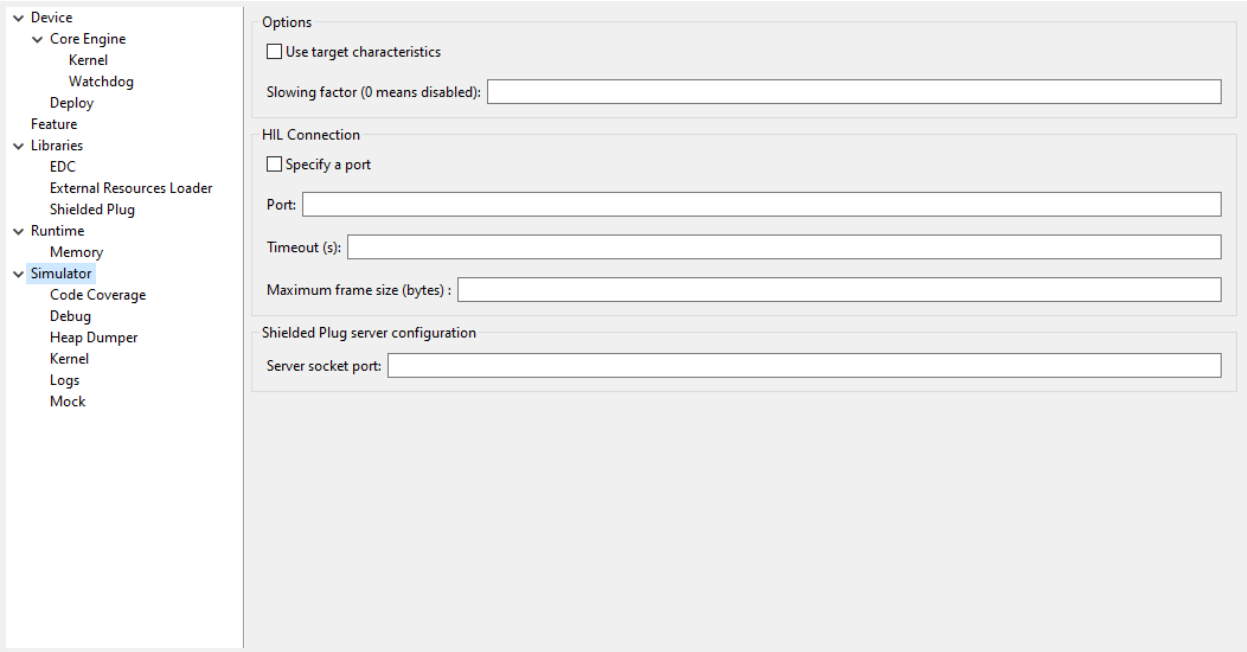
Option Name: `core.memory.thread.max.size`

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.

5.8.6 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): Port

Option Name: `s3.hil.port`

Default value: `8001`

Description:

Format: Positive `integer`

Values: [1024-65535]

It specifies the port used by the MicroEJ Simulator to accept HIL connections.

Option(text): Timeout (s)

Option Name: `s3.hil.timeout`

Default value: `10`

Description:

Format: Positive `integer`

It specifies the time the MicroEJ Simulator should wait before failing when it invokes native methods.

Option(text): Maximum frame size (bytes)

Option Name: `com.microej.simulator.hil.frame.size`

Default value: `262144`

Description:

Maximum frame size in bytes. Must be increased to transfer large arrays to native side.

Group: Shielded Plug server configuration

Description:

This group allows configuration of the Shielded Plug database.

Option(text): Server socket port

Option Name: `sp.server.port`

Default value: `10082`

Description:

Set the Shielded Plug server socket port.

Group: Advanced Simulation Options

When running large applications, the Simulator can abruptly reach a memory limit with the following trace:

```
[...] An error message [...]
"Internal limits reached. Please contact support@microej.com"
See error log file: /tmp/microej/s3/s3_1616489929186.log
```

Depending on the error message, one of the following options must be set to increase the size of the memory area which is full.

Option: Objects Heap Size

Error Message: `java.lang.OutOfMemoryError` exception thrown

Option Name: `S3.JavaMemory.HeapSize`

Default value: `4096` (kilobytes)

Description:

This memory area contains any kind of objects (regular, immortal and immutable objects). If you get a `java.lang.OutOfMemoryError` exception but your Java Heap is not full, most likely you should augment this option. It must be greater than the sum of *Java Heap* and *Immortal Heap*.

Option: System Chars Size

Error Message: `Failed to allocate internString.`

Option Name: `S3.JavaMemory.SystemCharsSize`

Default value: `1024` (kilobytes)

Description:

This memory area contains system interned strings. System interned strings are likely allocated by the debugger. If you get a `Failed to allocate internString.` message while debugging an Application, most likely you should augment this option.

Option: Application Chars Size

Error Message: `Failed to allocate internString.`

Option Name: `S3.JavaMemory.ApplicationCharsSize`

Default value: `4096` (kilobytes)

Description:

This memory area contains Application interned strings (String literals). If you get a `Failed to allocate internString.` message while the Simulator is starting the Application, most likely you should augment this option.

Option: Methods Size

Error Message: `Failed to allocate method's code.`

Option Name: `S3.JavaMemory.MethodsSize`

Default value: `10000` (kilobytes)

Description:

This memory area contains loaded methods code.

Option: Thread Stack Size

Error Message: `The simulator has encountered a stack overflow error while analyzing method [...]`

Option Name: `S3.JavaMemory.ThreadStackSize`

Default value: `300` (kilobytes)

Description:

This memory area contains all Application threads stacks.

Option: Icetea Heap End

Error Message: `S3 internal heap is full.`

Option Name: `IceteaRuntimeSupport.S3.HeapEnd`

Default value: `64000000` (bytes)

Description:

This is the overall Simulator memory limit. It includes fixed sizes internal structures and all memory areas. The value must be greater than the size of the memory areas that can be parameterized above.

Option: Symbol Table Size

Error Message: `Symbols table area is full.`

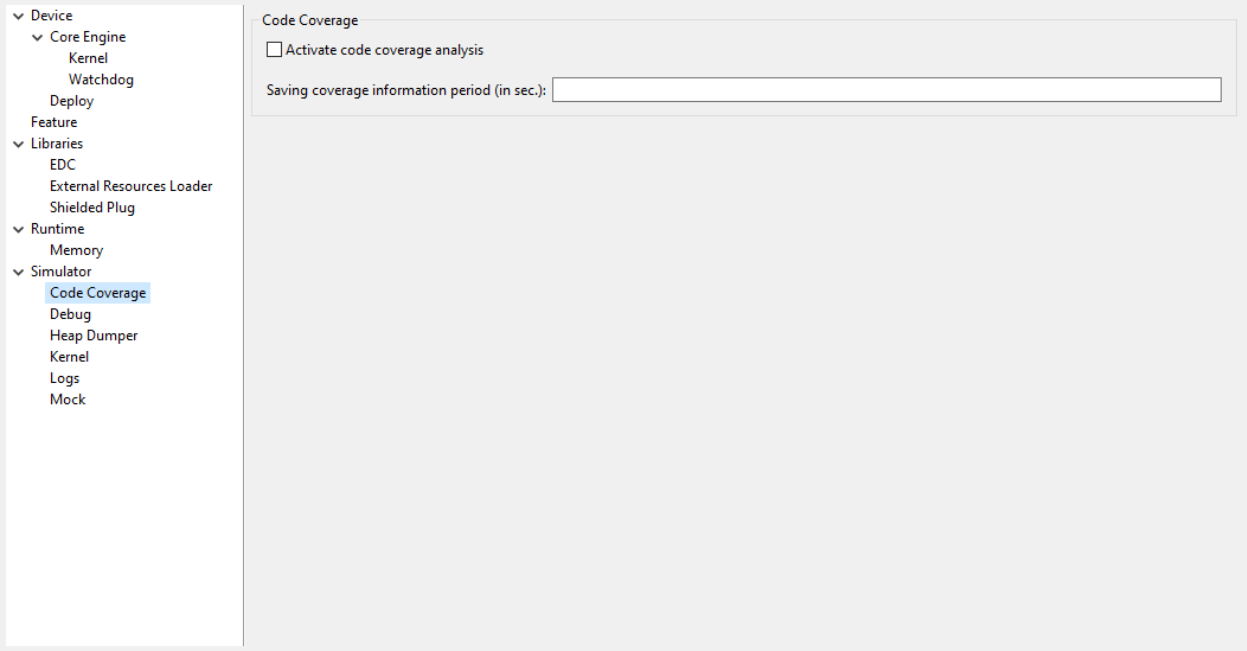
Option Name: `S3.SymbolTable.MaxNbState`

Default value: `500000`

Description:

This is the number of symbols that can be handled by the internal symbol table (any kind of names: class names, method names, …).

Category: Code Coverage



Group: Code Coverage

Description:

This group is used to set parameters of the code coverage analysis tool.

Option(checkbox): Activate code coverage analysis

Option Name: `s3.cc.activated`

Default value: `false`

Description:

When selected it enables the code coverage analysis by the MicroEJ Simulator. Resulting files are output in the cc directory inside the output directory. You can process these files to an HTML report afterward with the built-in *Code Coverage Analyzer* .

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

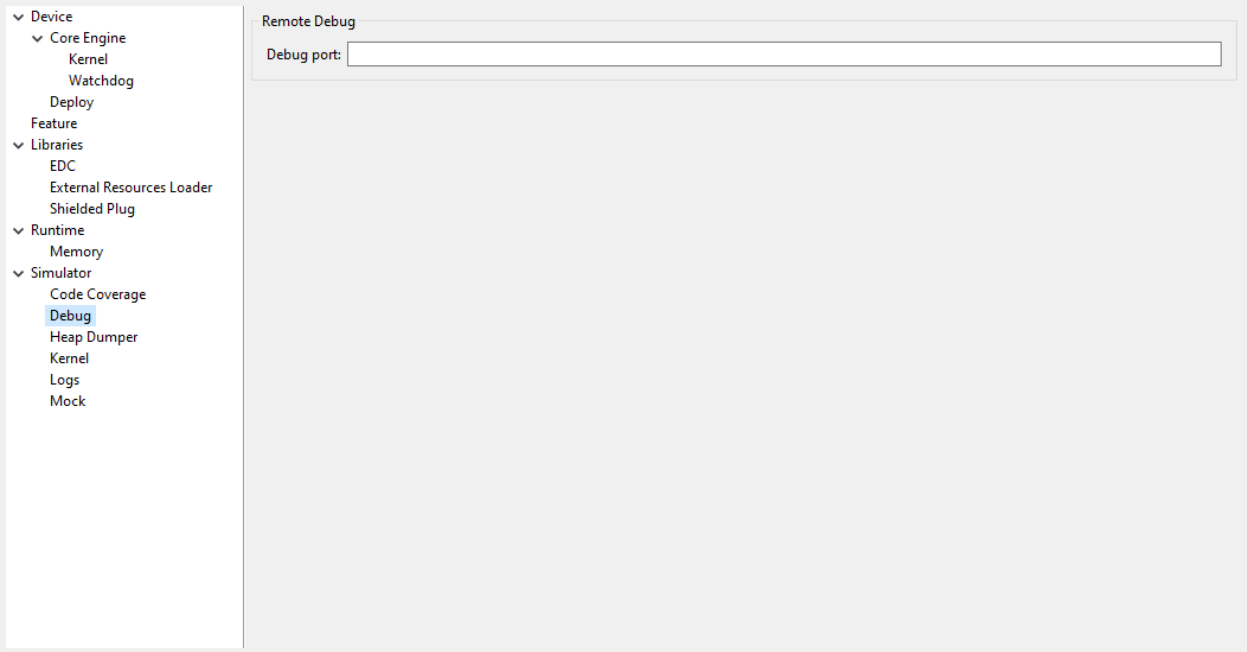
Option Name: `s3.cc.thread.period`

Default value: `15`

Description:

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

Category: Debug



Group: Remote Debug

Option(text): Debug port

Option Name: `debug.port`

Default value: `12000`

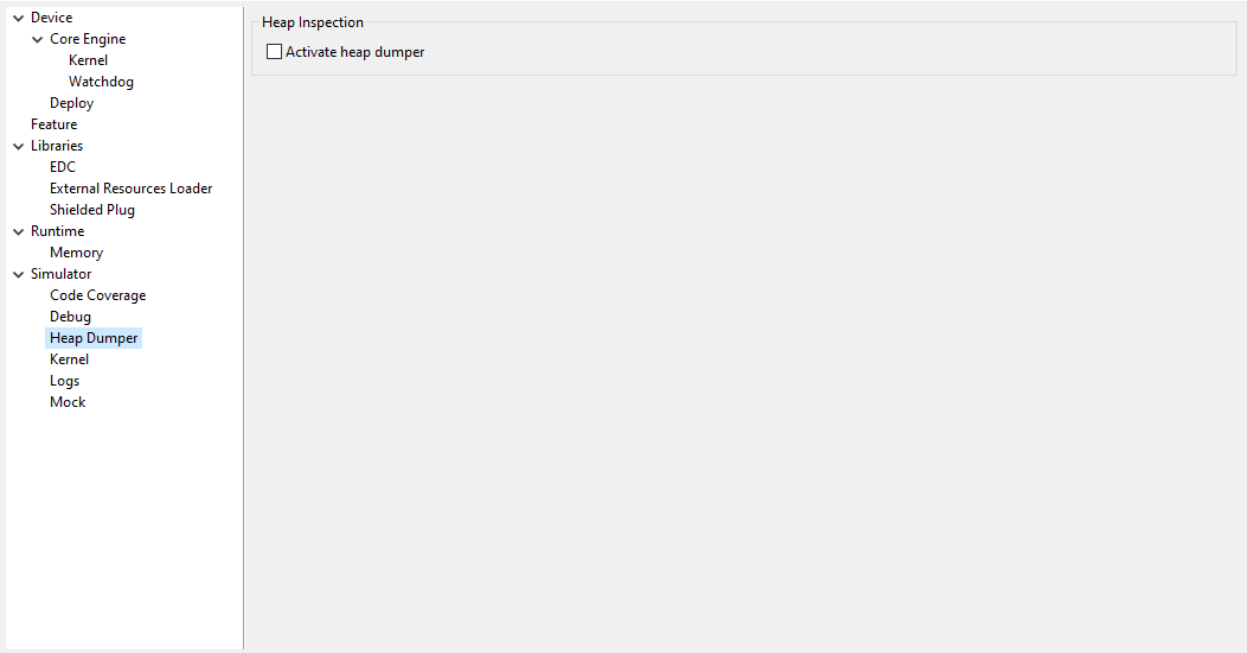
Description:

Configures the JDWP debug port.

Format: Positive `integer`

Values: [1024-65535]

Category: Heap Dumper



Group: Heap Inspection

Description:

This group is used to specify heap inspection properties.

Option(checkbox): Activate heap dumper

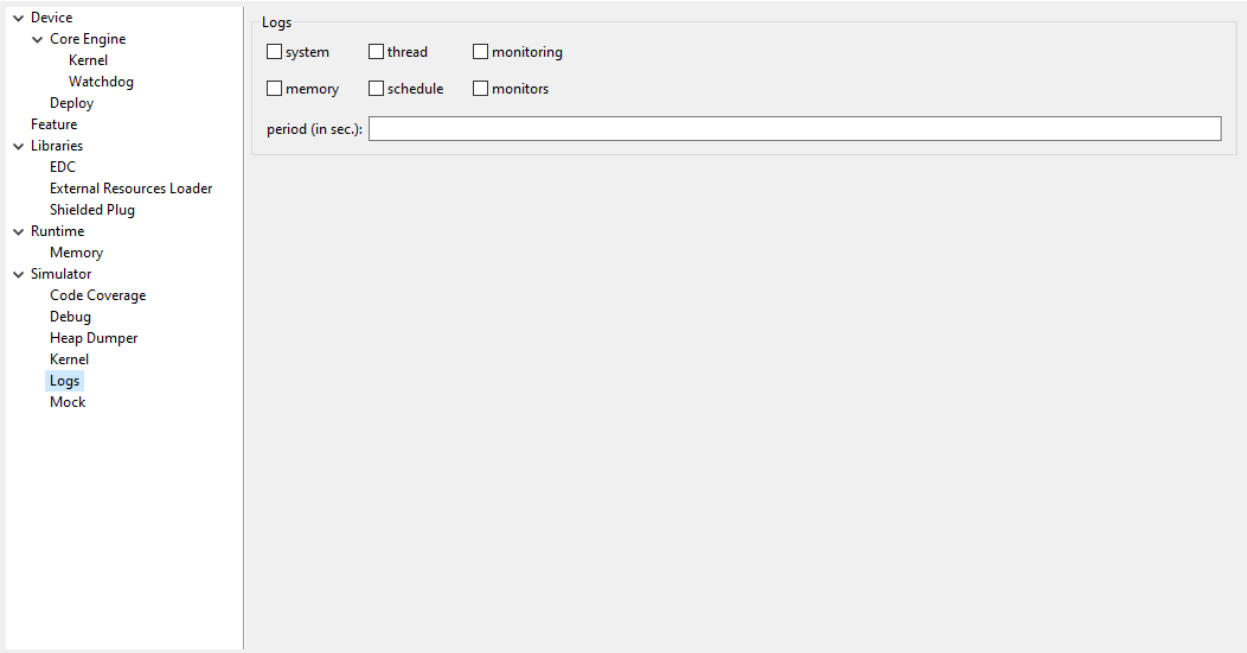
Option Name: `s3.inspect.heap`

Default value: `false`

Description:

When selected, this option enables a dump of the heap each time the System.gc() method is called by the MicroEJ Application.

Category: Logs



Group: Logs

Description:

This group defines parameters for MicroEJ Simulator log activity. Note that logs can only be generated if the `Simulator > Use target characteristics` option is selected.

Some logs are sent when the Simulator executes some specific action (such as start thread, start GC, etc), other logs are sent periodically (according to defined log level and the log periodicity).

Option(checkbox): system

Option Name: `console.logs.level.low`

Default value: `false`

Description:

When selected, System logs are sent when the Simulator executes the following actions:

start and terminate a thread

start and terminate a GC

exit

Option(checkbox): thread

Option Name: `console.logs.level.thread`

Default value: `false`

Description:

When selected, thread information is sent periodically. It gives information about alive threads (status, memory allocation, stack size).

Option(checkbox): monitoring

Option Name: `console.logs.level.monitoring`

Default value: `false`

Description:

When selected, thread monitoring logs are sent periodically. It gives information about time execution of threads.

Option(checkbox): memory

Option Name: `console.logs.level.memory`

Default value: `false`

Description:

When selected, memory allocation logs are sent periodically. This level allows to supervise memory allocation.

Option(checkbox): schedule

Option Name: `console.logs.level.schedule`

Default value: `false`

Description:

When selected, a log is sent when the Simulator schedules a thread.

Option(checkbox): monitors

Option Name: `console.logs.level.monitors`

Default value: `false`

Description:

When selected, monitors information is sent periodically. This level permits tracing of all thread state by tracing monitor operations.

Option(text): period (in sec.)

Option Name: `console.logs.period`

Default value: `2`

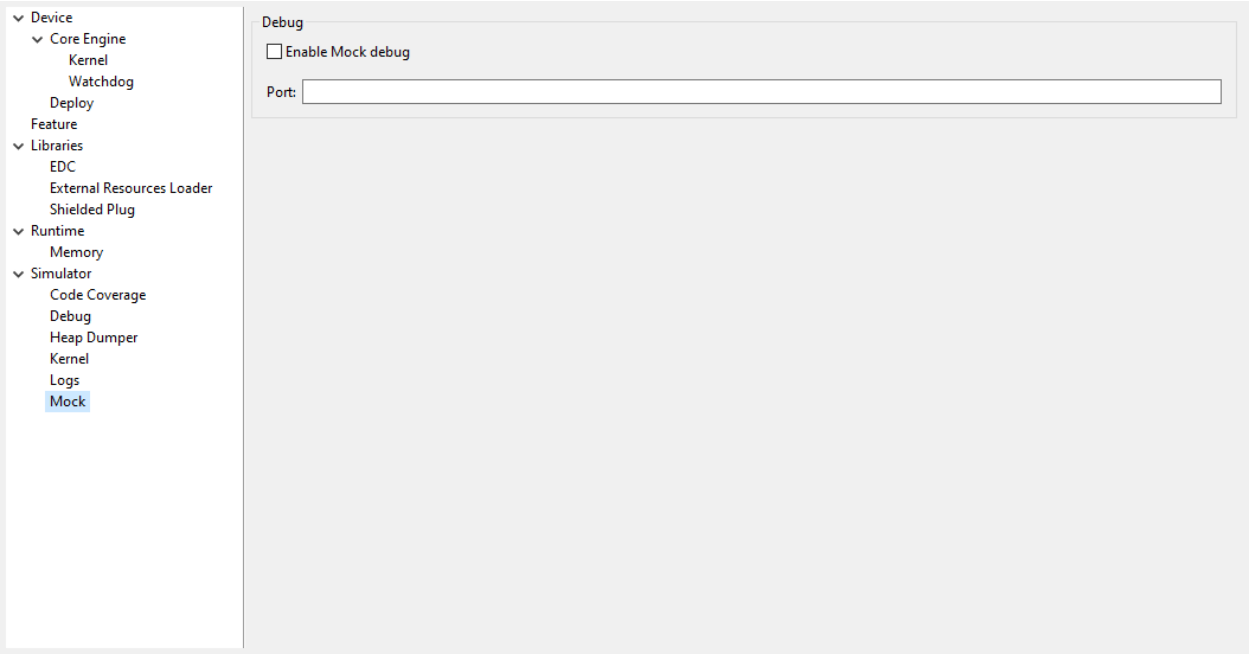
Description:

Format: Positive `integer`

Values: [0-60]

Defines the periodicity of periodical logs.

Category: Mock



Description:

Specify Hardware In the Loop Mock client options

Group: Debug

Option(checkbox): Enable Mock debug

Option Name: `com.microej.simulator.hil.debug.enabled`

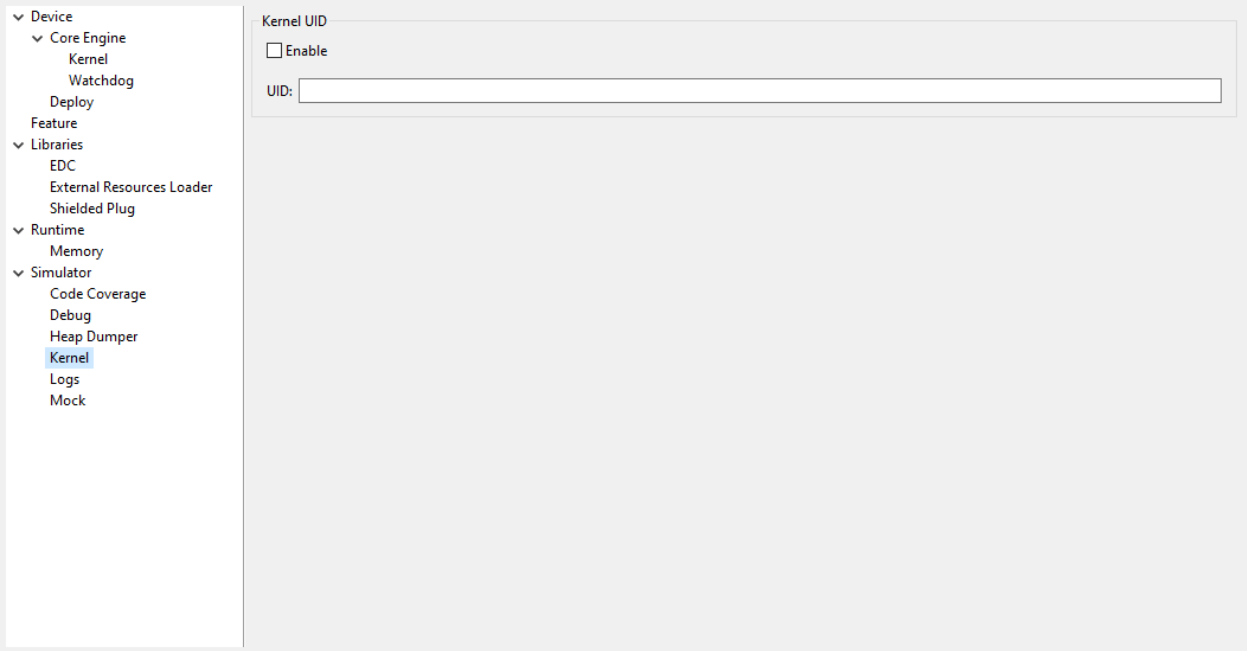
Default value: `false`

Option(text): Port

Option Name: `com.microej.simulator.hil.debug.port`

Default value: `8002`

Category: Kernel



Group: Kernel UID

Option(checkbox): Enable

Option Name: `com.microej.simulator.kf.kernel.uid.enabled`

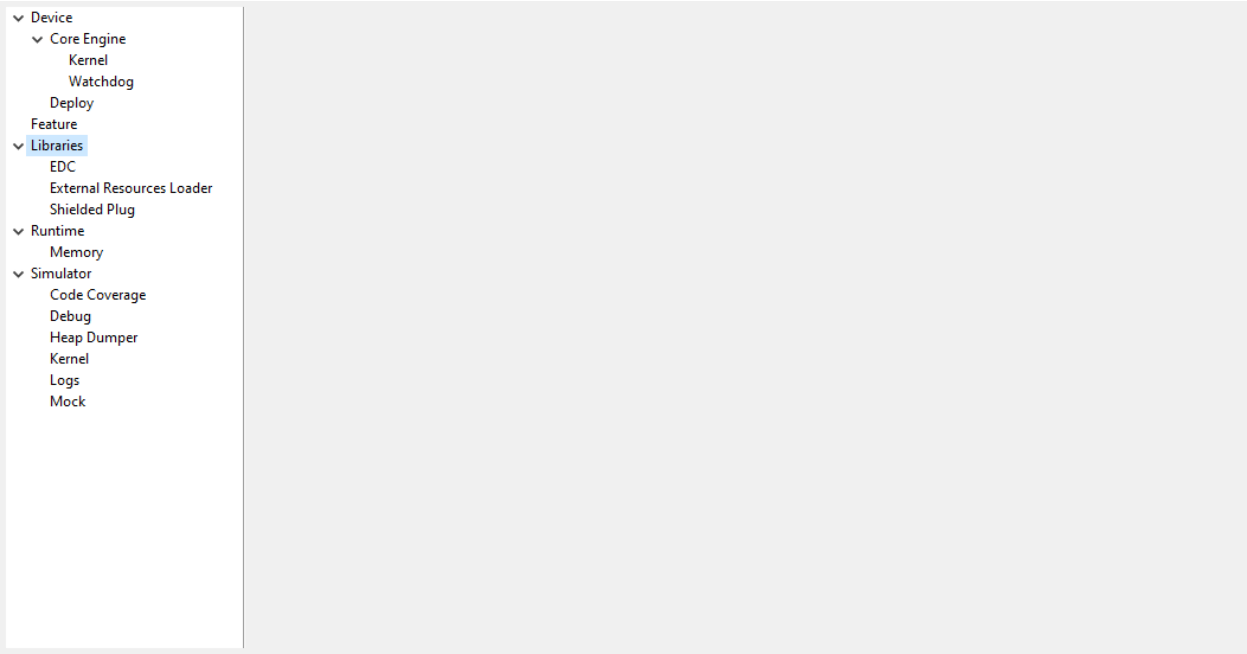
Default value: `false`

Option(text): UID

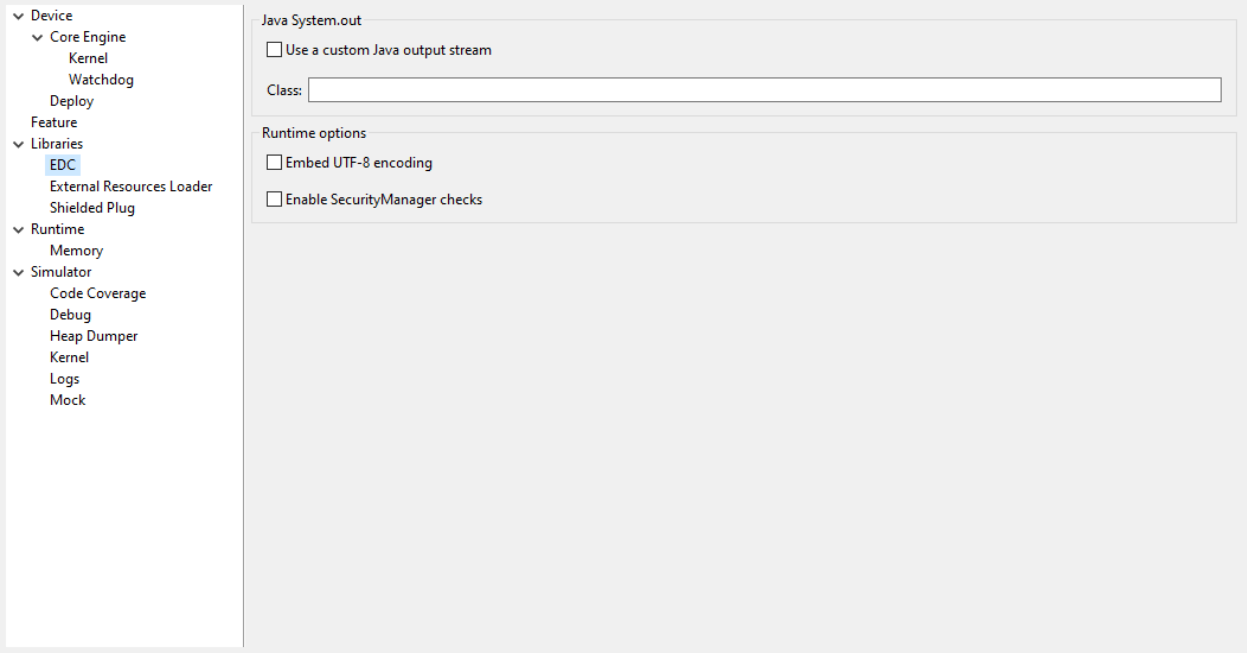
Option Name: `com.microej.simulator.kf.kernel.uid`

Default value: `(empty)`

5.8.7 Category: Libraries



Category: EDC



Group: Java System.out

Option(checkbox): Use a custom Java output stream

Option Name: `core.outputstream.disable.uart`

Default value: `false`

Description:

Select this option to specify another Java `System.out` print stream.

If selected, the default Java output stream is not used by the Java application. The Core Engine will not use the default Java output stream at startup.

Option(text): Class

Option Name: `core.outputstream.class`

Default value: `(empty)`

Description:

Format: `Java` class like `packageA.packageB.className`

Defines the Java class used to manage `System.out` .

At startup the Core Engine will try to load this class using the `Class.forName()` method. If the given class is not available, it will use the default Java output stream as usual. The specified class must be available in the application classpath.

Group: Runtime options

Description:

Specifies the additional classes to embed at runtime.

Option(checkbox): Embed UTF-8 encoding

Option Name: `cldc.encoding.utf8.included`

Default value: `true`

Description:

Embed UTF-8 encoding.

Option(checkbox): Enable SecurityManager checks

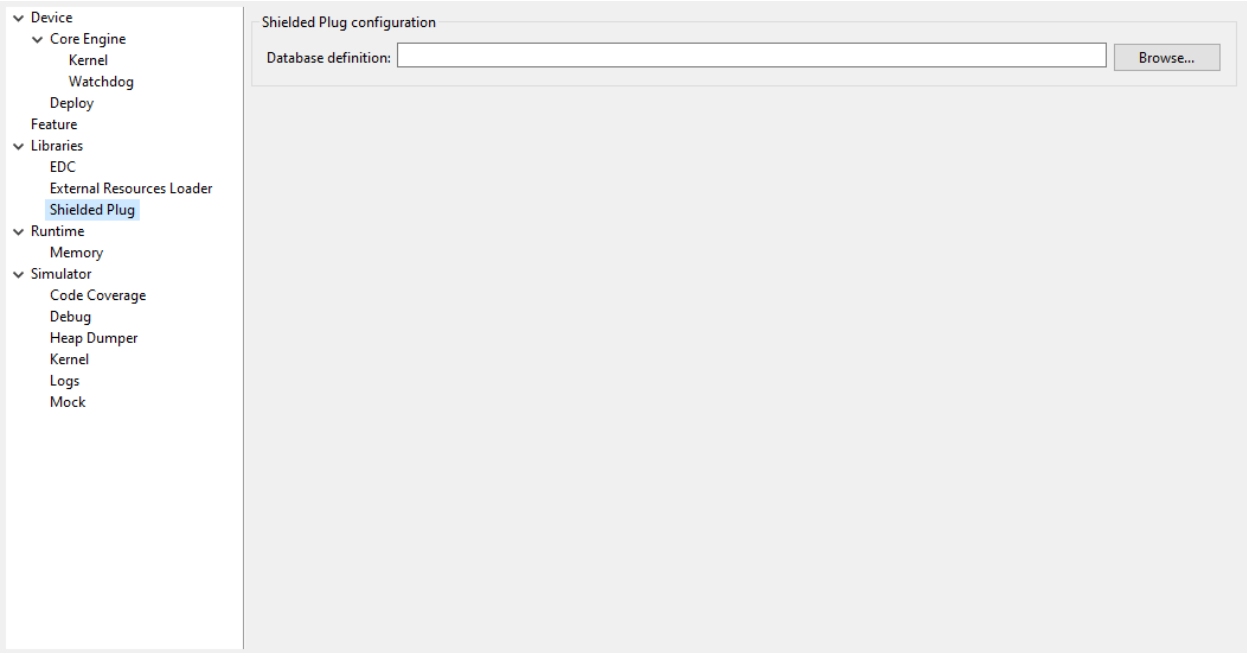
Option Name: `com.microej.library.edc.securitymanager.enabled`

Default value: `false`

Description:

Enable the security manager Permission checks.

Category: Shielded Plug



Group: Shielded Plug configuration

Description:

Choose the database XML definition.

Option(browse): Database definition

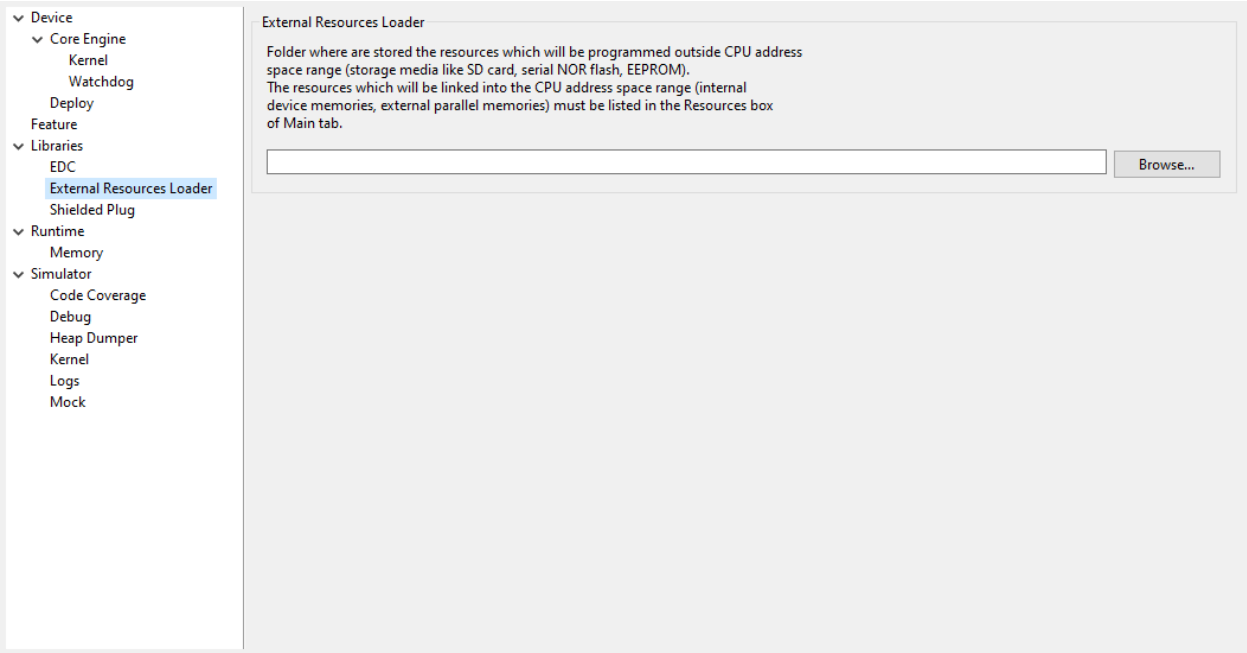
Option Name: `sp.database.definition`

Default value: `(empty)`

Description:

Choose the database XML definition.

Category: External Resources Loader



Group: External Resources Loader

Description:

This group allows to specify the external resources input folder. The content of this folder will be copied in an application output folder and used by SOAR and the Simulator. If empty, the default location will be [output folder]/externalResources, where [output folder] is the location defined in Execution tab.

Option(browse):

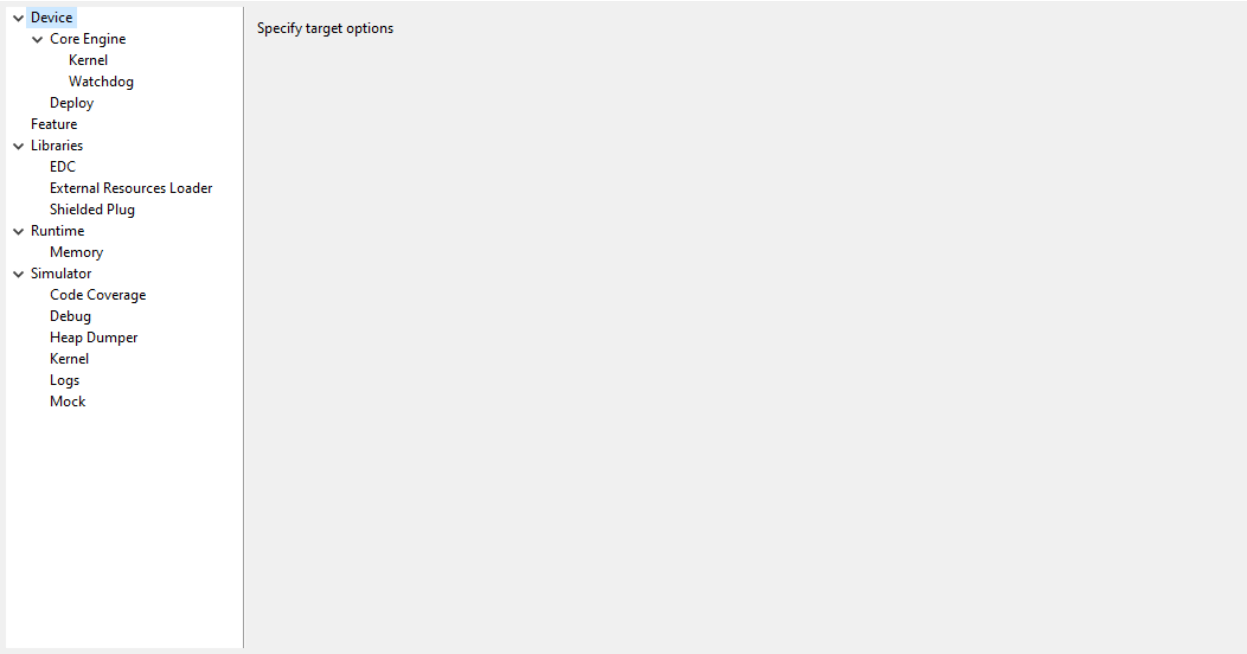
Option Name: `ej.externalResources.input.dir`

Default value: `(empty)`

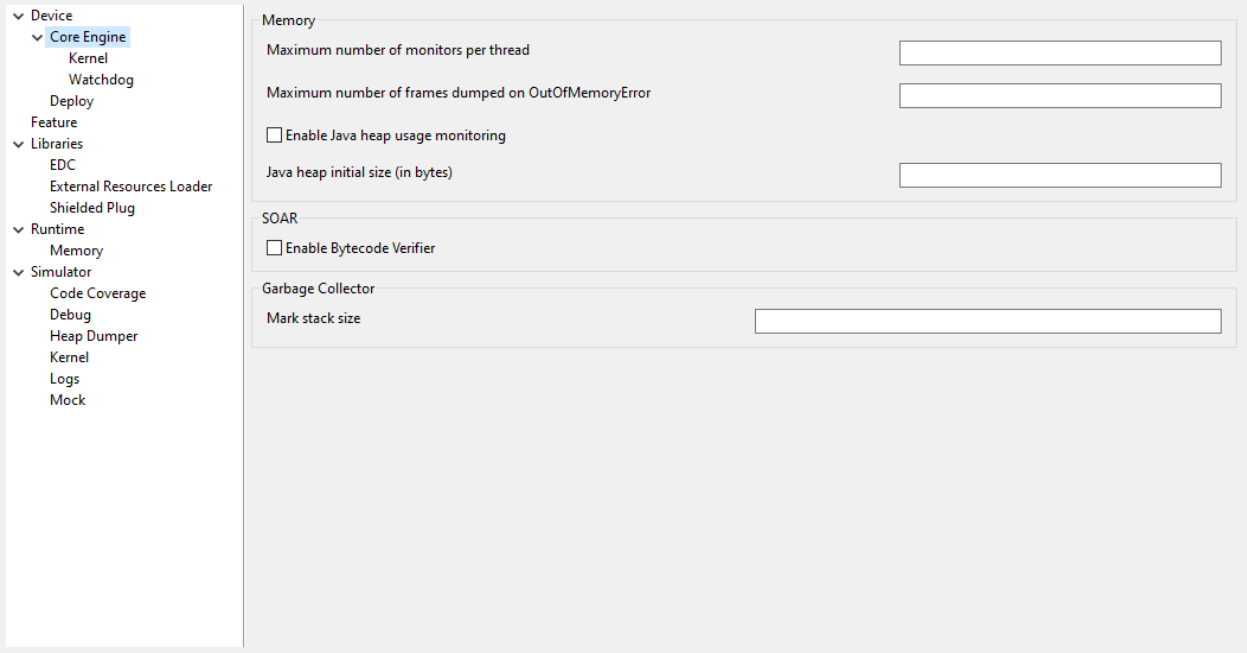
Description:

Browse to specify the external resources folder..

5.8.8 Category: Device



Category: Core Engine



Group: Memory

Option(text): Maximum number of monitors per thread

Option Name: `core.memory.thread.max.nb.monitors`

Default value: `8`

Description:

Specifies the maximum number of monitors a thread can own at the same time.

Option(text): Maximum number of frames dumpers on OutOfMemoryError

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

Default value: `5`

Description:

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

Option(checkbox): Enable Java heap usage monitoring

Option Name: `com.microej.runtime.debug.heap.monitoring.enabled`

Default value: `false`

Option(text): Java heap initial size

Option Name: `com.microej.runtime.debug.heap.monitoring.init.size`

Default value: `0`

Description:

Specify the initial size (in bytes) of the Java Heap.

Group: SOAR

Option(checkbox): Enable Bytecode Verifier

Option Name: `soar.bytecode.verifier`

Default value: Standalone Application: `false` , Sandboxed Application: `true`

Description:

Enables *Binary Code Verifier* during application build.

In the context of building a Standalone Application, the bytecode verifier is, by default, disabled to prioritize performance. In this case, the code is considered trusted. Conversely, when building a Sandboxed Application, the bytecode verifier is automatically enabled by default. This is particularly important when dealing with untrusted third-party code.

Group: Garbage Collector

Option(text): GC mark stack size

Option Name: `com.microej.runtime.core.gc.markstack.levels.max`

Default value: `32`

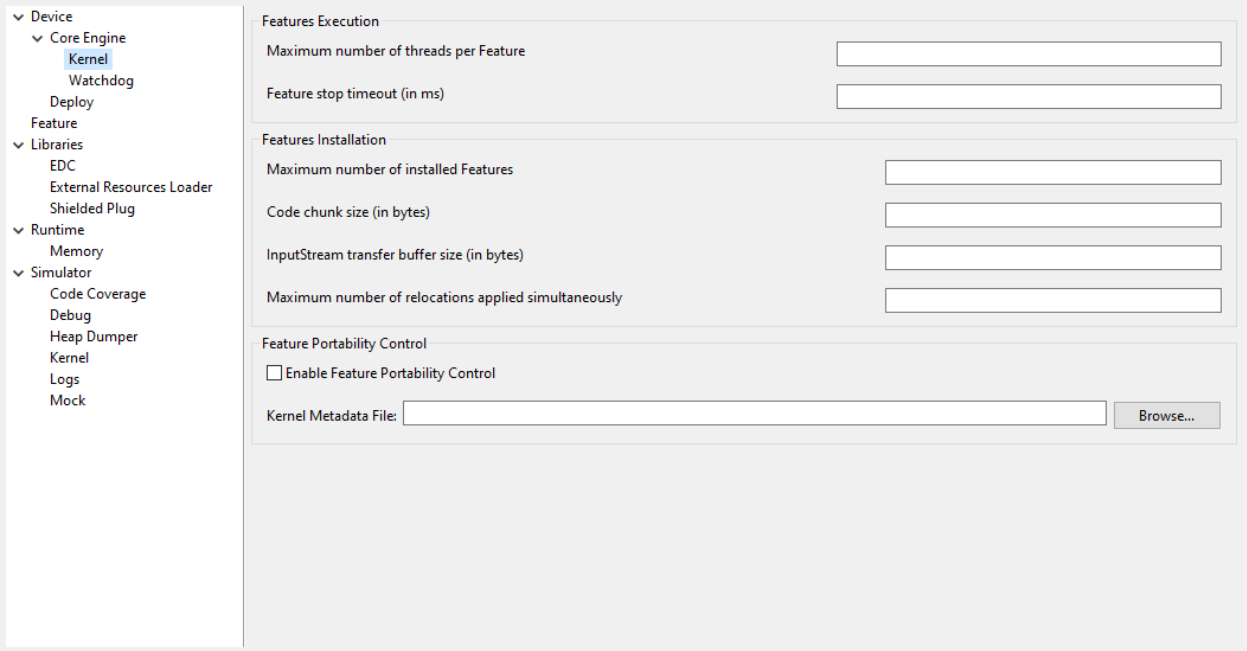
Description:

Indicates the quantity of items in the *Garbage Collector*'s mark stack. The mark stack is used by the Garbage Collector for identifying live objects within the heap through a depth-first search approach. Once the mark stack reaches its capacity, the Garbage Collector proceeds to inspect heap memory, which may slow down garbage collection performance.

You can receive a notification when the mark stack limit is reached by implementing the following hook:

```
void LLMJVM_on_GC_MarkStackOverflow_reached(void) {
   // When entering here, the GC mark stack is undersized, which may affect GC performance.
   // It is recommended to either increase the GC mark stack size or reduce the object graph␣
↪depth.
}
```

Category: Kernel



Group: Threads

Option(text): Maximum number of threads per Feature

Option Name: `core.memory.feature.max.threads`

Default value: `5`

Description:

Specifies the maximum number of threads a Feature is allowed to use at the same time.

Option(text): Feature stop timeout

Option Name: `com.microej.runtime.kf.waitstop.delay`

Default value: `2000`

Description:

Specifies the maximum time allowed for the FeatureEntryPoint.stop() method to return (value in milliseconds).

Group: Features Installation

Option(text): Maximum number of installed Features

Option Name: `com.microej.runtime.kernel.dynamicfeatures.max`

Default value: `16`

Description:

Specifies the maximum number of Features that can be installed to this Kernel (see Kernel.install() method).

Option(text): Code chunk size

Option Name: `com.microej.soar.kernel.featurecodechunk.size`

Default value: `65536`

Description:

Specifies the size in bytes of the code chunk in RAM. See *Code Chunk Size* section for more details.

Option(text): InputStream transfer buffer size

Option Name: `com.microej.runtime.kf.link.transferbuffer.size`

Default value: `512`

Description:

Specifies the size in bytes of the temporary byte array for reading in the Feature InputStream. See *InputStream Transfer Buffer Size* section for more details.

Option(text): Maximum number of relocations applied simultaneously

Option Name: `com.microej.runtime.kf.link.chunk.relocations.count`

Default value: `128`

Group: Feature Portability Control

Option(checkbox): Enable Feature Portability Control

Option Name: `com.microej.soar.kernel.featureportabilitycontrol.enabled`

Default value: `false`

Option(browse): Kernel Metadata File

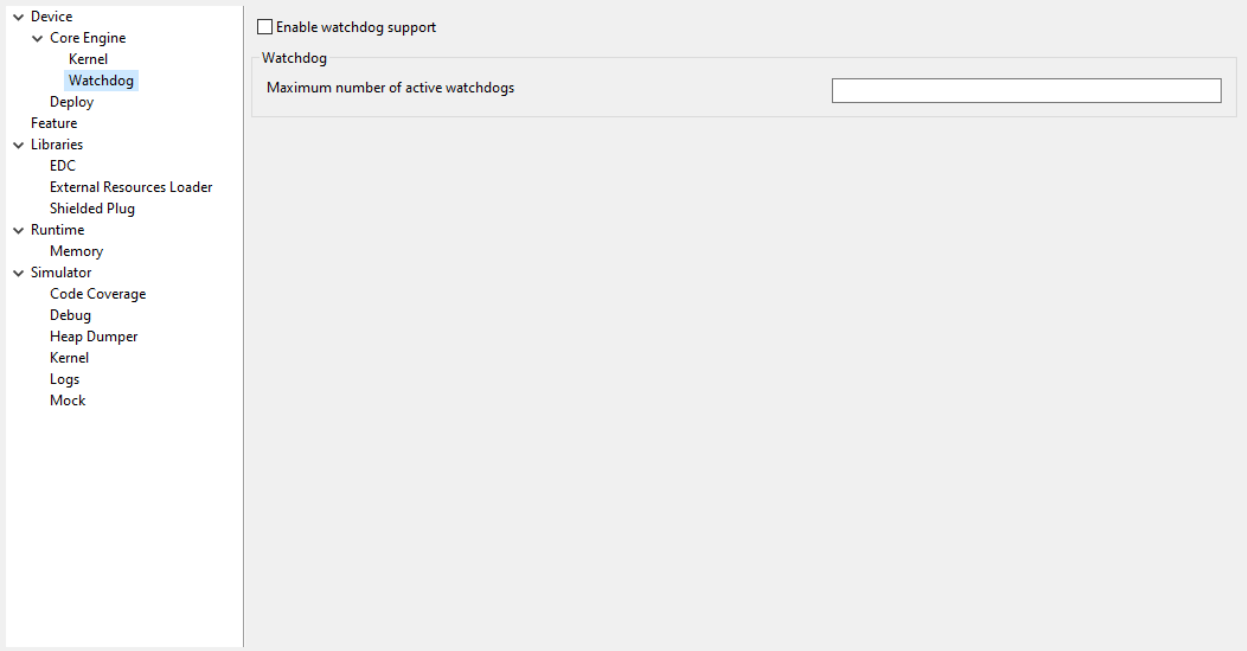
Option Name: `com.microej.soar.kernel.featureportabilitycontrol.metadata.path`

Default value: `(empty)`

Description:

Specifies the path to the Kernel metadata file for Feature Portability Control.

Category: Watchdog



Option(checkbox): Enable watchdog support

Option Name: `enable.watchdog.support`

Default value: `true`

Group: Watchdog

Option(text):

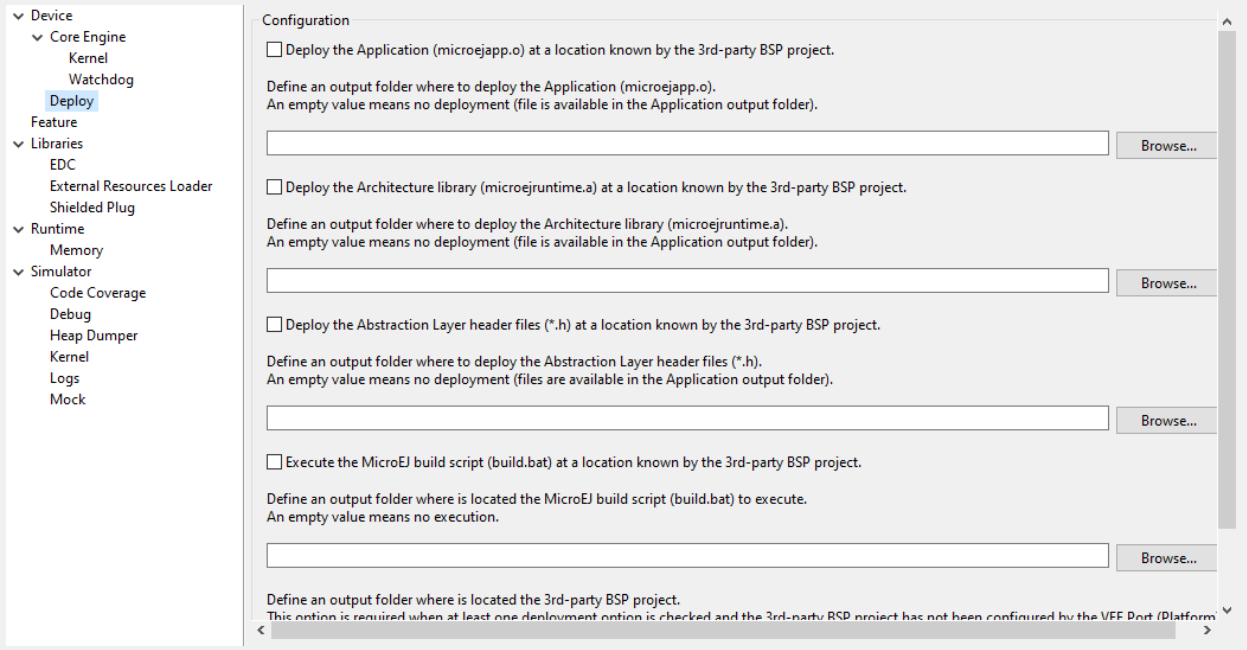
Option Name: `maximum.active.watchdogs`

Default value: `4`

Description:

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

Category: Deploy



Description:

Configures the output locations where store the Application, the Architecture libraries and Abstraction Layer header files.

See *Board Support Package (BSP) connection* chapter for more details.

Group: Configuration

Option(checkbox): Deploy the Application (microejapp.o) at a location known by the 3rd-party BSP project.

Option Name: `deploy.bsp.microejapp`

Default value: `true`

Description:

Deploy the Application (microejapp.o) at a location known by the 3rd-party BSP project.

Option(browse):

Option Name: `deploy.dir.microejapp`

Default value: `(empty)`

Description:

Choose an output folder where to deploy the Application. An empty value means no deployment (file is available in the Application output folder).

Option(checkbox): Deploy the Architecture library (microejruntime.a) at a location known by the 3rd-party BSP project.

Option Name: `deploy.bsp.microejlib`

Default value: `true`

Description:

Deploy the Architecture library (microejruntime.a) at a location known by the 3rd-party BSP project.

Option(browse):

Option Name: `deploy.dir.microejlib`

Default value: `(empty)`

Description:

Choose an output folder where to deploy the Architecture library. An empty value means no deployment (file is available in the Application output folder).

Option(checkbox): Deploy the Abstraction Layer header files (`*.h`) at a location known by the 3rd-party BSP project.

Option Name: `deploy.bsp.microejinc`

Default value: `true`

Description:

Deploy the Abstraction Layer header files (`*.h`) at a location known by the 3rd-party BSP project.

Option(browse):

Option Name: `deploy.dir.microejinc`

Default value: `(empty)`

Description:

Choose an output folder where to deploy the Architecture library. An empty value means no deployment (file is available in the Application output folder).

Option(checkbox): Execute the MicroEJ build script (build.bat) at a location known by the 3rd-party BSP project.

Option Name: `deploy.bsp.microejscript`

Default value: `false`

Description:

Execute the MicroEJ build script (build.bat) at a location known by the 3rd-party BSP project.

Option(browse):

Option Name: `deploy.dir.microejscript`

Default value: `(empty)`

Description:

Choose an output folder where is located the MicroEJ build script (build.bat) to execute. An empty value means no execution.

Option(browse):

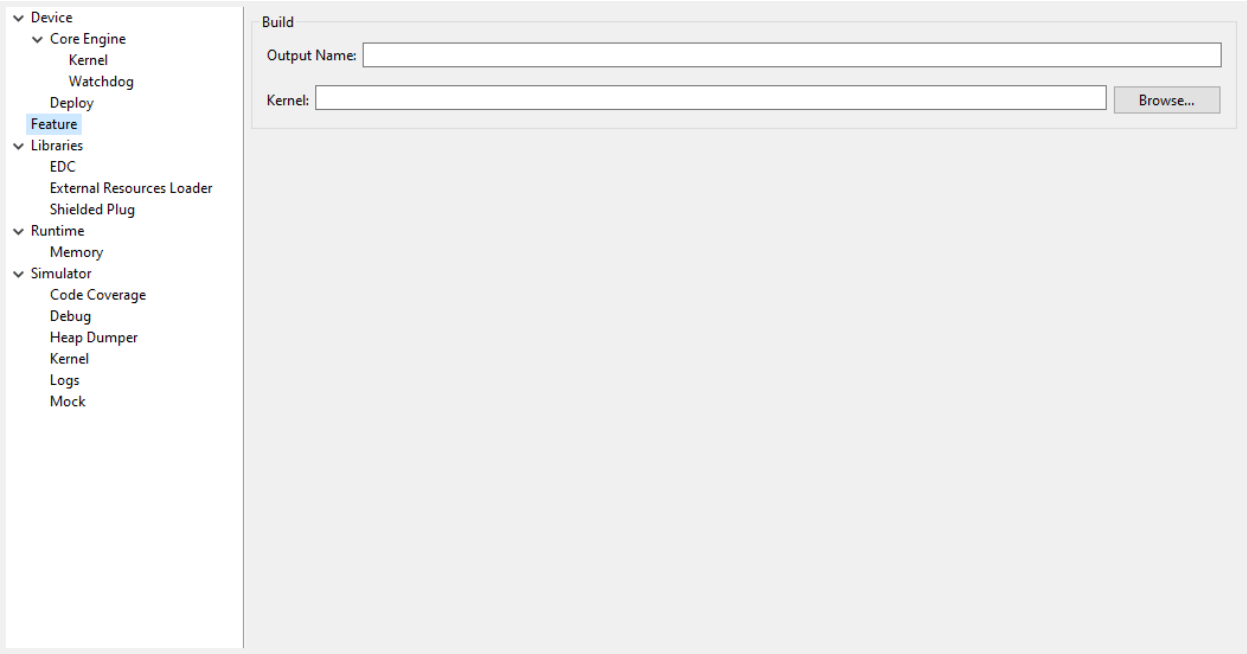
Option Name: `deploy.bsp.root.dir`

Default value: `(empty)`

Description:

Choose an output folder where is located the 3rd-party BSP project. An empty value means not set (3rd-party BSP project location may have been configured by the VEE Port).

5.8.9 Category: Feature



Description:

Specify Feature options

Group: Build

Option(text): Output Name

Option Name: `feature.output.basename`

Default value: `application`

Option(browse): Kernel

Option Name: `kernel.filename`

Default value: `(empty)`

5.9 Sandboxed Application

5.9.1 Fundamental Concepts

Multi-Sandboxing is based on the *the Kernel & Features Specification* (KF).

It allows an application code to be split between multiples parts:

- the main application, called the Kernel,

- zero or more applications called Features.

Therefore, a Kernel Application relates to the *Kernel* concept and a Sandboxed Application relates to the *Feature* concept.

Some fundamental points:

- The Kernel is mandatory. It is assumed to be reliable, trusted and cannot be modified.
- A Feature is an application "extension" managed by the Kernel.
- A Feature is fully controlled by the Kernel: it can be installed (dynamically or statically pre-installed), started, stopped and uninstalled at any time independent of the system state (particularly, a Feature never depends on another Feature to be stopped).
- A Feature is optional, potentially not-trusted, maybe unreliable and can be executed without jeopardizing the safety of the Kernel execution and other Features.
- Resources accesses (RAM, hardware peripherals, CPU time, ...) are under control of the Kernel.

Note: You can go further by reading *the Kernel & Features Specification*.

5.9.2 Shared Interfaces

Principle

The Shared Interface mechanism provided by the Core Engine is an object communication bus based on plain Java interfaces where method calls are allowed to cross Sandboxed Applications boundaries without relying on Kernel APIs.

The Shared Interface mechanism is the cornerstone for designing reliable Service Oriented Architectures. Communication is based on the sharing of interfaces defining APIs (Contract Oriented Programming).

The basic schema:

- A provider application publishes an implementation for a shared interface into a system registry.
- A user application retrieves the implementation from the system registry and directly calls the methods defined by the shared interface.

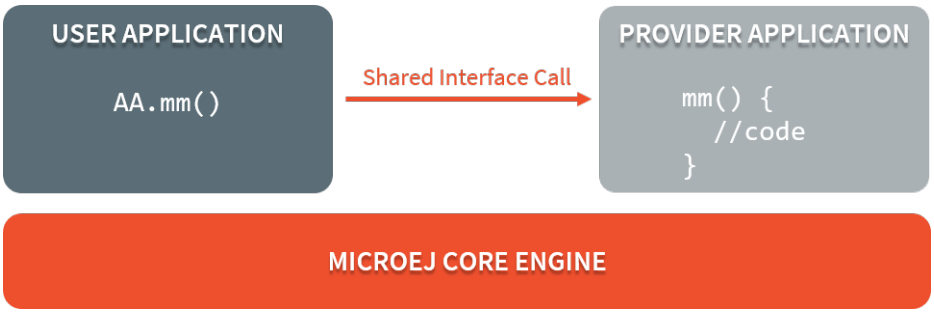


Fig. 8: Shared Interface Call Mechanism

The Shared Interface mechanism is based on automatic proxy objects created by the Core Engine. This offers a reliable way for users to handle broken links in case the provider application has been stopped or uninstalled.

Applications with a Shared Interface must provide a dedicated implementation (called the Proxy class implementation). Its main goal is to perform the remote invocation and provide a reliable implementation regarding the interface contract even if the remote application fails to fulfill its contract (unexpected exceptions, application killed, ...). The Core Engine will allocate instances of this Proxy class when an implementation (of the Shared Interface) owned by another application is being transferred to this application.

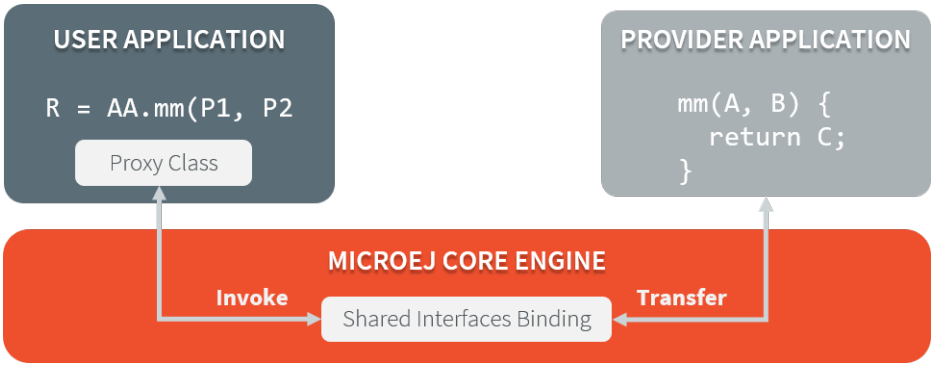


Fig. 9: Shared Interfaces Proxy Overview

This mecanism is formally specified in the *[KF] specification*.

Shared Interface Usage

Usage of a Shared Interface follows these steps:

1. Define the Shared Interface:
    1. Define the Java interface
    2. Implement the proxy for the interface
    3. Register the interface as a Shared Interface
2. From the provider application,
    1. Create an instance of this Shared Interface
    2. Register the instance to a KF service registry
3. From the consumer application,
    1. Retrieve a proxy of the instance from the KF service registry
    2. Call methods of the instance proxy.

Define the Shared Interface

Define the Java Interface

The definition of a Shared Interface starts by defining a standard Java interface. For example:

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

Some restrictions apply to Shared Interfaces compared to standard Java interfaces:

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

Implement the Proxy Class

A proxy class is implemented and executed on the client side, 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(); // handle errors
        }
    }
}
```

Each implemented method of the proxy class is responsible for performing the remote call and catching all errors from the server side and to provide an appropriate answer to the client application call according to the interface method specification (contract).

The *Proxy class implementation* section documents how to perform the remote invocation.

Register the Shared Interface

To declare an interface as a Shared Interface, it must be registered in a Shared Interfaces identification file. A Shared Interface identification file is an XML file with the `.si` filename extension and the following format:

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

Shared Interface identification files must be placed at the root of the application classpath, typically it is defined in the `src/main/resources` folder.

Use the Shared Interface at Runtime

Projects Structure

Both the consumer and the provider applications must have the Java interface, the proxy class and the identification file on the classpath in order to be able to use the Shared Interface.

Typically, the 3 files can be defined in an Add-On Library that both application projects depend on.

Create and Share an instance of a Shared Interface

The provider application can instantiate the Java interface. For example:

```
MyInterface myInstance = new MyInterface() {
    @Override
    public void foo() {
        System.out.println("Hello world!");
    }
};
```

In order to share the instance with other applications, the provider application must register the instance with some registry owned by the Kernel (see *Communication between Kernel and Feature* for details) like so:

```
ServiceFactory.getServiceRegistry().register(MyInterface.class, myInstance);
```

Retrieve and Use a Proxy of a Shared Interface Instance

The consumer application can then retrieve the instance from the Kernel registry like so:

```
MyInterface otherAppInstance = ServiceFactory.getServiceRegistry().getService(MyInterface.
↪class);
// otherAppInstance is actually an instance of the proxy class owned by the consumer␣
↪application
```

Then it can call the interface methods transparently:

```
otherAppInstance.foo(); // remote invocation through the proxy
```

Transferable Types

In the process of a cross-application method call, parameters and return value of methods declared in a Shared Interface must be transferred back and forth between application boundaries.

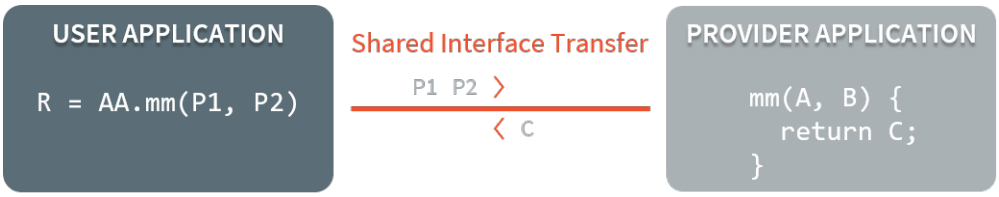


Fig. 10: Shared Interface Parameters Transfer

The following table 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 type is owned by the Kernel can be transferred to another Application provided this has been authorized by the Kernel. The list of Kernel types that can be transferred is Kernel specific, so you have to consult your Kernel specification. When an argument transfer is forbidden, the call is abruptly stopped and an java.lang.IllegalAccessError is thrown by the *Core Engine*.

Note: For these types to be transferable, a dedicated *Kernel Type Converter* must have been registered in the Kernel.

The table below lists typical Kernel types allowed to be transferred through a Shared Interface call on *Evaluation Kernels <https://repository.microej.com/old_index.php?resource=FIRM>* distributed by MicroEJ Corp.

Table 2: MicroEJ Evaluation Kernels Rules for Transferable Types

Type	Rule
java.lang.Boolean	Clone by copy
java.lang.Byte	Clone by copy
java.lang.Character	Clone by copy
java.lang.Short	Clone by copy
java.lang.Integer	Clone by copy
java.lang.Float	Clone by copy
java.lang.Long	Clone by copy
java.lang.Double	Clone by copy
java.lang.String	Clone by copy
java.io.InputStream	Create a Proxy reference
java.util.Date	Clone by copy
java.util.List<T>	Clone by copy with recursive element conversion
java.util.Map<K,V>	Clone by copy with recursive keys and values conversion

Implementing the Proxy Class

Remote invocation methods are defined in the super class ej.kf.Proxy and are named `invokeXXX()` where `XXX` is the kind of return type.

Table 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

As this class is part of the Application, the developer has the full control on the Proxy implementation and is free to insert additional code such as logging calls and errors for example. It is also possible to have different proxy implementations for the same Shared Interface in different applications.

A Sandboxed Application is an Application that can run over a Multi-Sandbox Executable. Sandboxed Applications can be linked statically to the Multi-Sandbox Executable or installed dynamically on the device.

Typical use cases for a Sandboxed Application are:

- over the air provisioning: the Application is dynamically installed or updated on a fleet of heterogenous devices.
- modularization: a monolithic application is split into multiple Sandboxed Applications; each of them can be started or stopped separately.

The following figure shows the general process of building a Sandboxed Application.

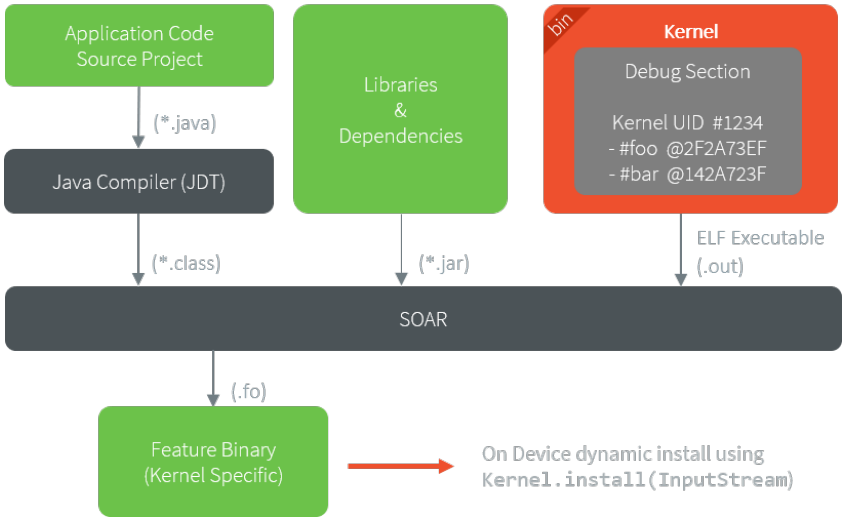


Fig. 11: Sandboxed Application Link Flow

Please refer to the *Kernel Developer Guide* to learn more on writing Kernel Applications and building Multi-Sandbox Executable and Virtual Devices.

5.10 Character Encoding

5.10.1 Default Encoding

The default character encoding is `ISO-8859-1`. It is thus the encoding used when:

- creating a new string from a byte array without specifying the encoding (String(byte[]) constructor),
- getting the byte array from a string without specifying the encoding (String.getBytes() method),
- printing a string to standard output stream (System.out),
- creating a new PrintStream without specifying the encoding.

5.10.2 UTF-8 Encoding

EDC provides an implementation of the `UTF-8` character encoding. It can be embedded using the *Embed UTF-8 encoding option* (otherwise a java.io.UnsupportedEncodingException exception will be thrown).

This implementation also supports Unicode code points as supplementary characters, by setting the *constant* `com.microej.library.edc.supplementarycharacter.enabled` to `true`.

5.10.3 Custom Encoding

It is possible to connect additional custom encodings. Please contact *our support team* for more details.

5.10.4 Console Output

By default, the standard output stream (System.out) uses `ISO-8859-1` encoding to print strings. If you want to print a string with a different encoding, you can create a new PrintStream:

```
PrintStream outUtf8 = new PrintStream(System.out, true, "UTF-8");
outUtf8.println("");
```

> **Warning:** Make sure you embed the `UTF-8` encoder (see *UTF-8 Encoding*)

The print methods write the raw byte array with the encoding used by the PrintStream to the console. The console must then be configured with the same encoding to display characters properly.

Set Encoding in MicroEJ SDK Console

The default encoding for Eclipse consoles is `UTF-8`. If your application prints non-ASCII characters, they may not be displayed properly.

The encoding used by a console for a given application can be set in the application launcher options: `Run` > `Run Configurations...`, and then `Common` tab > `Encoding` radio buttons.

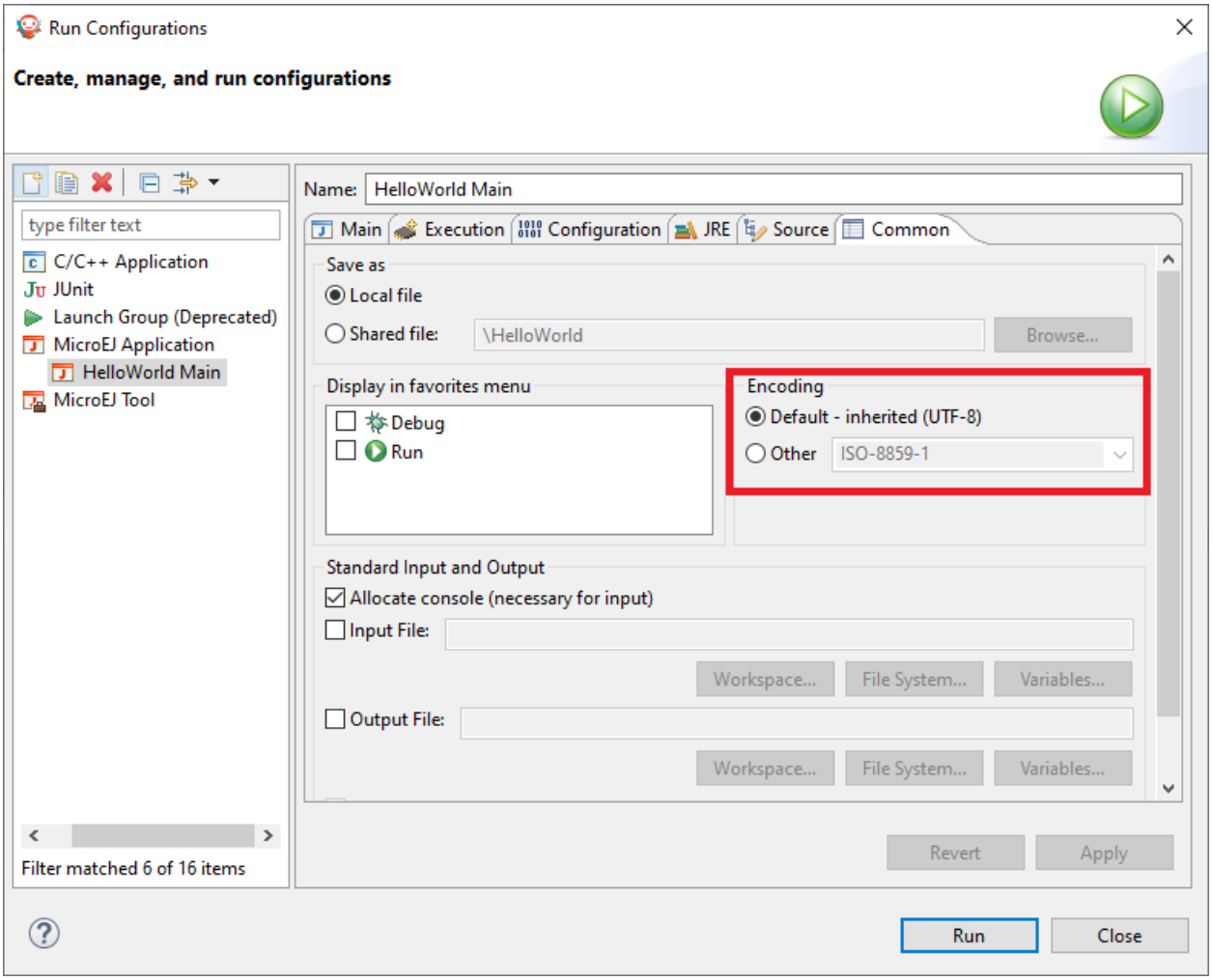


Fig. 12: Eclipse Launcher Console Encoding Options

5.11 Limitations

The following table lists the limitations of MicroEJ Architectures version `7.14.0` or higher, for both Evaluation and Production usage. Please consult *Architectures Changelog* for limitations changes on former versions.

Note: The term *unlimited* means there is no Architecture specific limitation. However, there may be limitations driven by device memory layout. Please refer to Platform specific documentation to get the memory mapping of *MicroEJ Core Engine sections*.

Table 4: Architecture Limitations

Item	EVAL	PROD
[Mono-Sandbox] Number of concrete types[1]	8192	8192
[Multi-Sandbox] Number of concrete types per context[Page 444, 1]	4096	4096
Number of abstract classes and interfaces	unlimited	unlimited
Class or Interface hierarchy depth	127	127
Number of methods	unlimited	unlimited
Method size in bytes	65536	65536
Numbers of exception handlers per method	63	63
Number of parameters for an SNI method	15	15
Number of instance fields[2] (Base type)	4096	4096
Number of instance fields[2] (References)	31	31
Number of static fields (boolean + byte)	65536	65536
Number of static fields (short + char)	65536	65536
Number of static fields (int + float)	65536	65536
Number of static fields (long + double)	65536	65536
Number of static fields (References)	65536	65536
Number of threads	63	63
Number of held monitors[3]	63	63
Time limit	60 minutes	unlimited
Number of methods and constructors calls	500000000	unlimited
Number of Java heap Garbage Collection	3000[4]	unlimited

5.12 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.

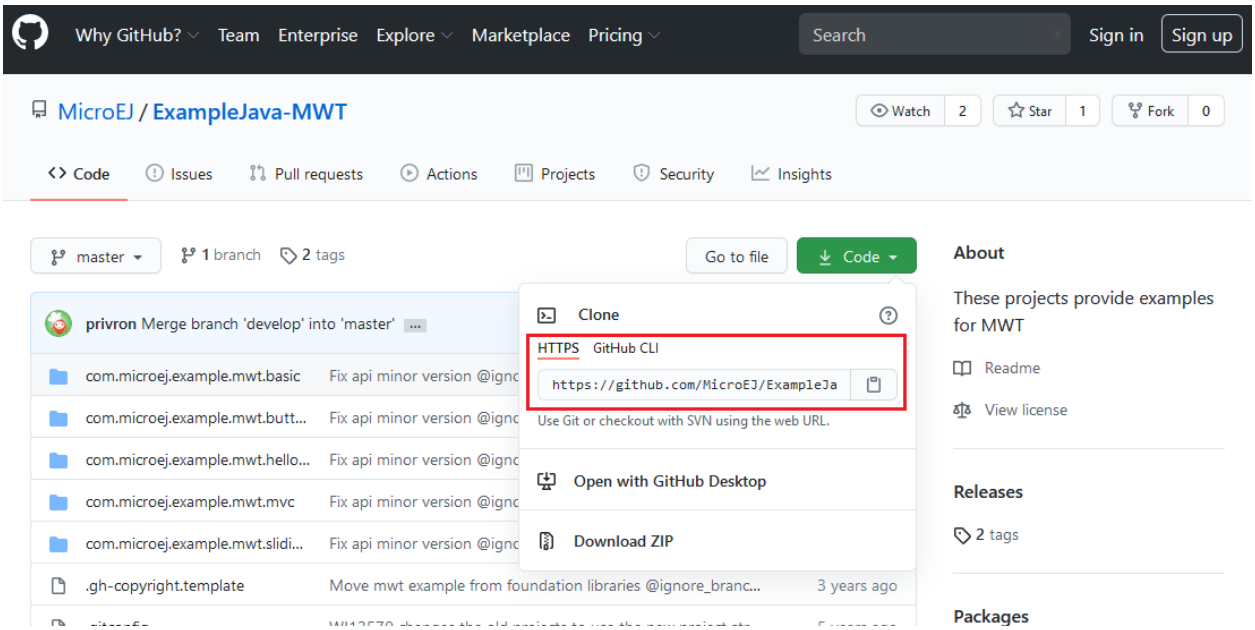
5.12.1 Repository Import

This section explains the steps to import a Github repository in MicroEJ SDK, illustrated with the MWT Examples repository.

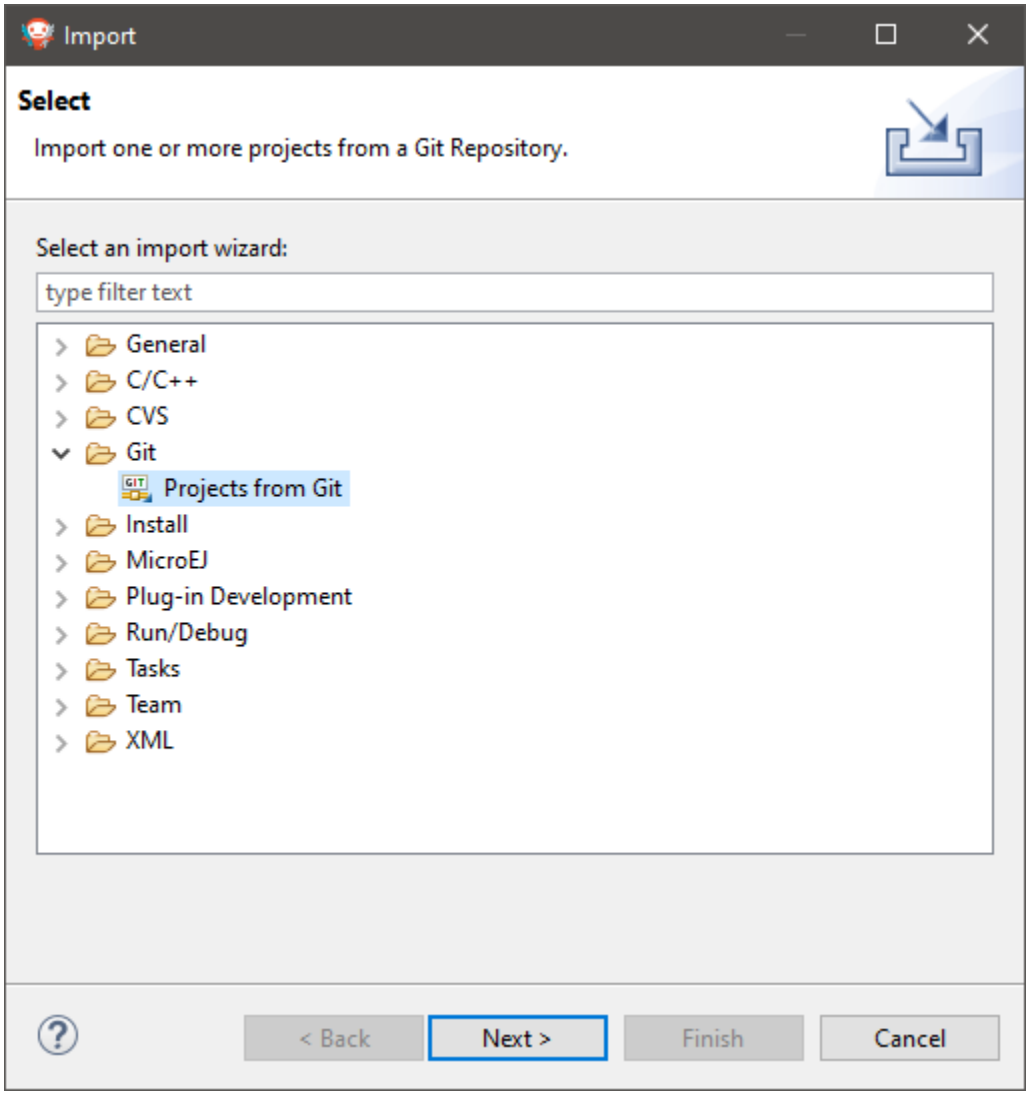
Note: MicroEJ SDK Distribution includes the Eclipse plugin for Git.

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

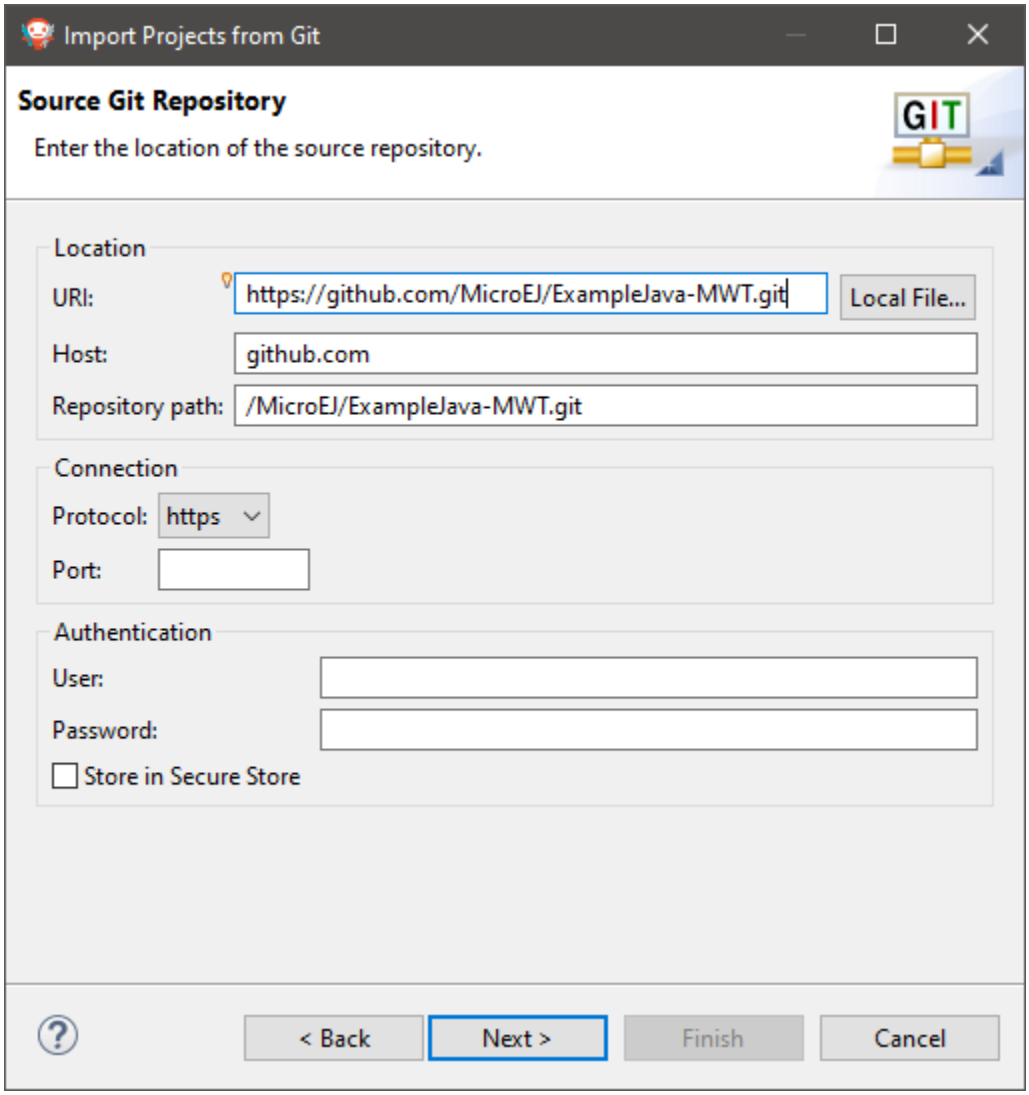
[1] Concrete types are classes and arrays that can be instantiated.
[2] All instance fields declared in the class and its super classes.
[3] The maximum number of different monitors that can be held by one thread at any time is defined by the *maximum number of monitors per thread Application option*.
[4] The Java heap Garbage Collection limit may throw unexpected cascading java.lang.OutOfMemoryError exceptions before the MicroEJ Core Engine exits.



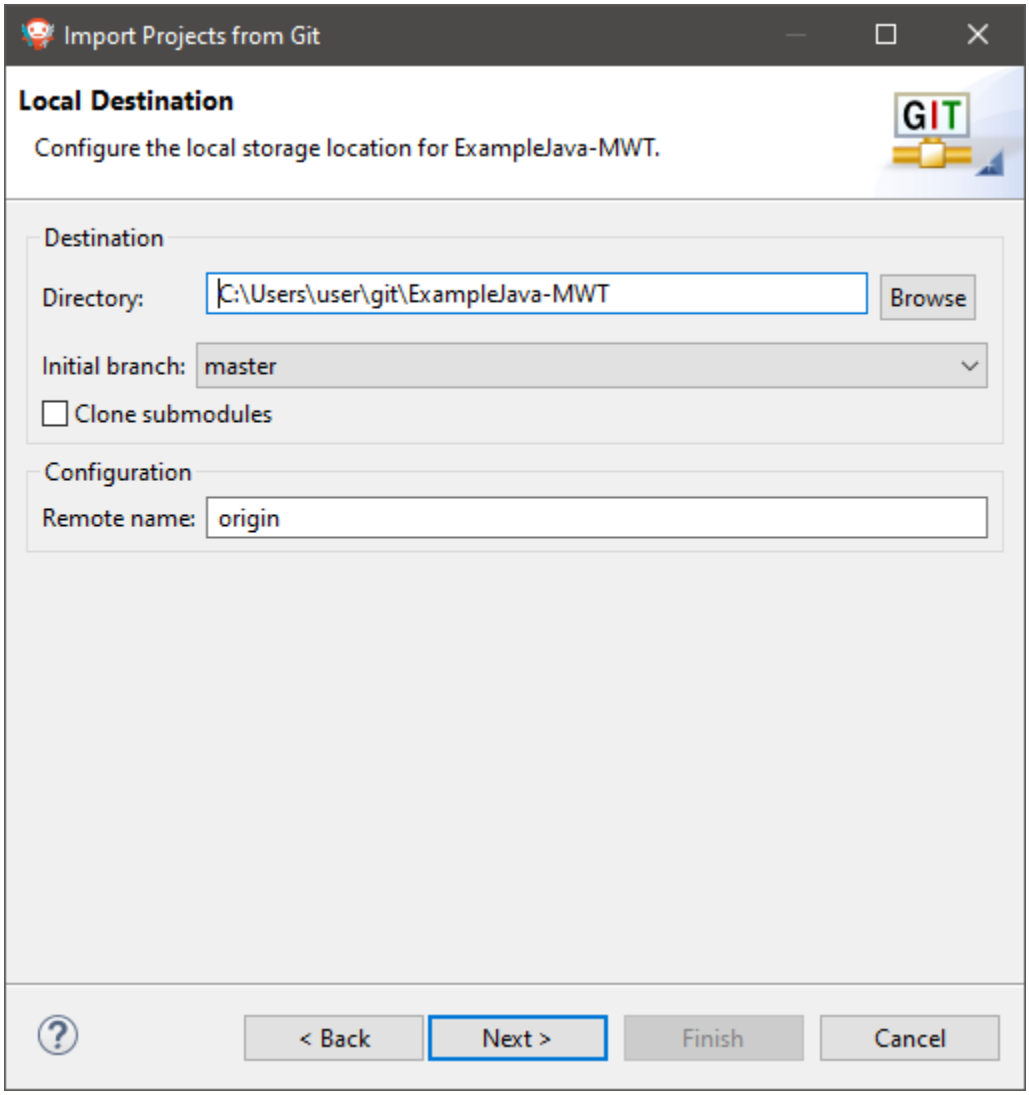
In the SDK, to clone and import the project from the remote Git repository into the MicroEJ workspace, select File > Import > Git > Projects from Git wizard.



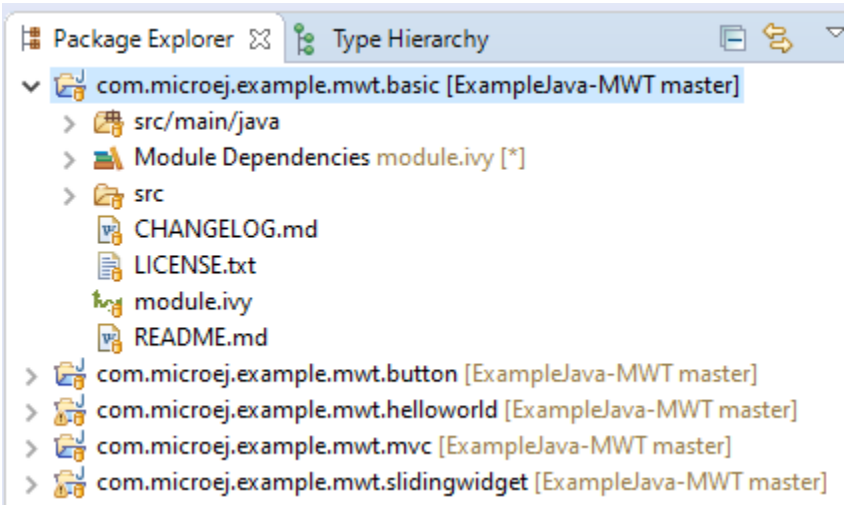
Click Next , select Clone URI , click Next and paste the remote repository address in the URI field. For this repository, the address is https://github.com/MicroEJ/ExampleJava-MWT.git. If the HTTP address is a valid repository, the other fields are filed automatically.



Click Next , select the `master` branch, click Next and accept the proposed *Local Destination* by clicking Next once again.



Click Next once more and finally Finish . The Package Explorer view now contains the imported projects.



If you want to import projects from another (GitHub) repository, you simply have to do the same procedure using the Git URL of the desired repository.

5.12.2 MicroEJ GitHub Badges

MicroEJ GitHub Badges are badges embedded in a README at the root of the repository. They highlight the compatibilities of the repository at a quick glance with MicroEJ Architecture, MicroEJ SDK and Graphical User Interface versions.

The color of the badge has the following meaning:

- Green means a current supported version: 

- Orange means a still supported version that will be deprecated in the future: 

- Gray means a deprecated version: 

5.13 Module Repositories

This chapter describes the *module repositories* provided by MicroEJ Corp.

5.13.1 Central Repository

The Central Repository is the module repository distributed and maintained by MicroEJ Corp. It contains a selection of production-grade modules such as *Libraries* and *Packs*.

Use

By default, the SDK is configured to import modules from the online Central Repository.

You can manually browse the repository at https://repository.microej.com/modules/.

Before starting to develop production code, it is strongly recommended to import the repository to your local environment. Please follow the steps described at https://developer.microej.com/central-repository/.

Licensing

The Central Repository is a set of modules distributed under various software licenses, including the *SDK EULA* for some of them. Please consult the `LICENSE.txt` file attached to each module.

Changelog

The Central Repository content is versioned. The overall changelog is available at https://repository.microej.com/ and describes modules additions or removals. For module content changes, please consult the `CHANGELOG.md` file attached to each module.

Javadoc

To consult the APIs documentation (Javadoc) of all *libraries* available in the repository, please visit https://repository.microej.com/javadoc/microej_5.x/apis/.

5.13.2 Developer Repository

The developer repository is an online repository hosted by MicroEJ Corp., contains community modules provided "as-is". It is similar to what Maven Central Repository are for hosting Java standard modules.

MicroEJ Corp. contributes to the developer repository in the following cases:

- Demos (Platforms, Firmware, Virtual Devices, Applications),
- Incubating Libraries,
- Former Central Repository versions,
- Hardware specific modules.

Use

By default, the SDK is configured to import modules from the developer repository[1].

You can also manually browse the repository at https://forge.microej.com/artifactory/microej-developer-repository-release/.

Before starting to develop production code, it is strongly recommended to transfer the desired modules to your local environment by creating your own *module repository* copy.

Licensing

The developer repository is a set of modules distributed under various software licenses. Please consult the `LICENSE.txt` file attached to each module.

Changelog

The developer repository is populated from multiple sources, thus there is no changelog for the whole repository content as it is the case of the Central Repository.

Please consult the `CHANGELOG.md` file attached to each module.

Javadoc

To consult the APIs documentation (Javadoc) of *libraries* available in the developer repository, please consult the javadoc attached to each module.

[1] Require SDK version `5.4.0` or higher.

Community

The developer repository can host modules developed by the community. If your organization plan to develop such module, please contact *our support team* to get dedicated credentials for publication.

5.13.3 Content Organization

The following table describes how are organized the *modules natures* within the repository.

Table 5: Modules Organization

Organization	Module Nature
`ej.api` , `com.microej.api`	Foundation Library API
`com.microej.architecture`	*Architecture*
`com.microej.pack`	*Pack*
`ej.tool` , `com.microej.tool`	Tool or Add-On processor
Any other	Add-On Library

5.14 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*).

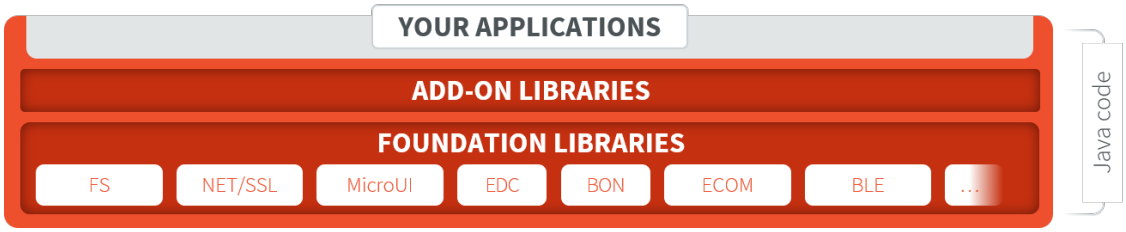


Fig. 13: MicroEJ Foundation Libraries and Add-On Libraries

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/.

5.14.1 Graphical User Interface

This section presents libraries relative to the user interface.

The following schema shows the overall architecture and modules:

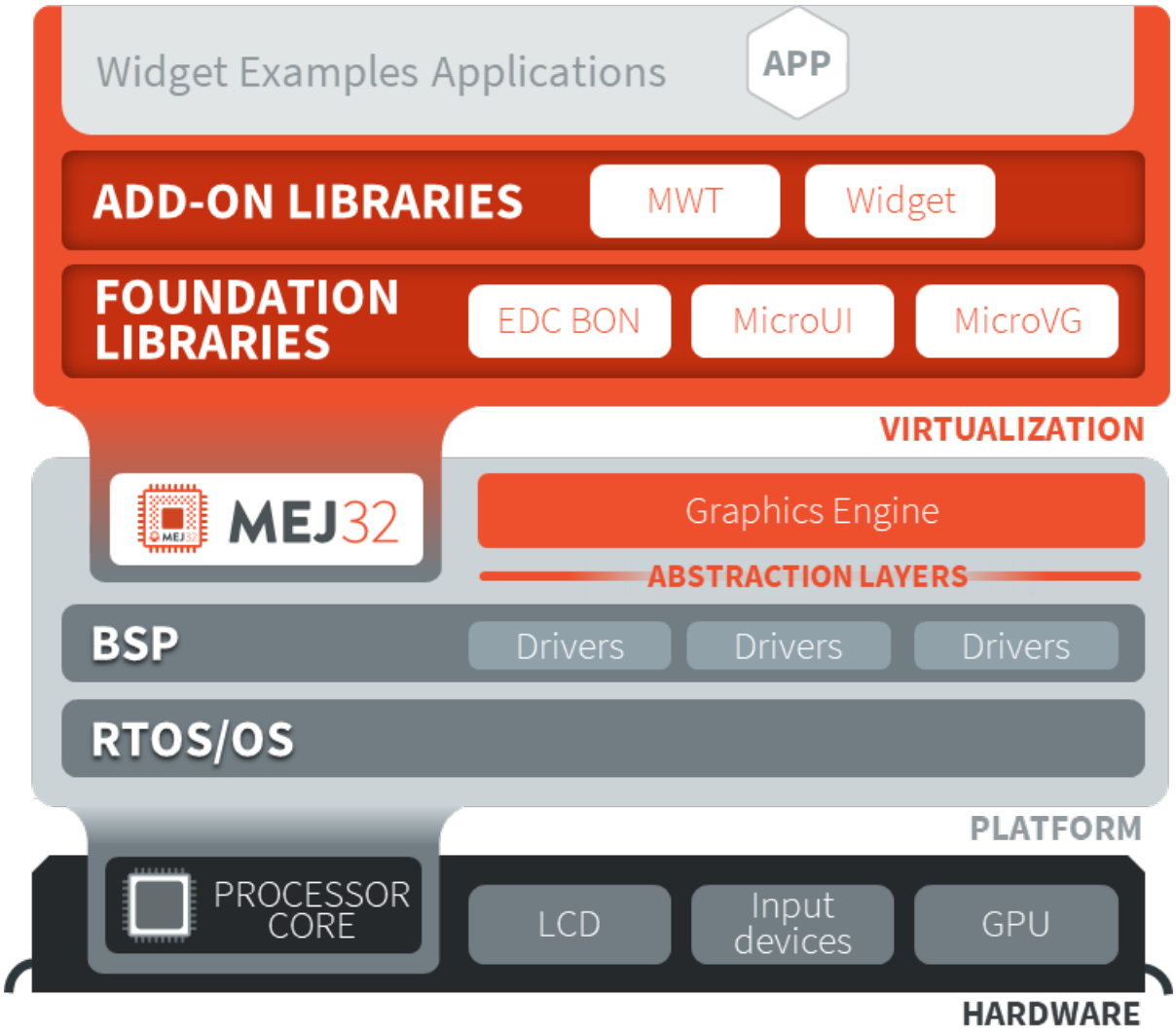


Fig. 14: Graphical User Interface Overview

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.

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 the project build file:

Gradle (build.gradle.kts)

MMM (module.ivy)

```
implementation("ej.api:microui:3.1.0")
```

```
<dependency org="ej.api" name="microui" rev="3.1.0"/>
```

Drawing Foundation Library extends MicroUI drawing APIs[1] with more complex ones such as:

- thick line, arc, circle and ellipse
- polygon
- image deformation and rotation

To use the Drawing Foundation Library, add Drawing API module to the project build file:

Gradle (build.gradle.kts)

MMM (module.ivy)

```
implementation("ej.api:drawing:1.0.3")
```

```
<dependency org="ej.api" name="drawing" rev="1.0.3"/>
```

Drawing Logs

When performing drawing operations, the VEE port may report incidents that occurred during a drawing to the application. Graphics contexts enable this by holding flags that can be set by the VEE port and read by the application.

Usage Overview

When the VEE port needs to report an incident, it will set *drawing log flags* in the graphics context describing its nature. The application will then be able to read the flag values to know if an incident occurred. This mechanism is meant to help the developer to debug the application if it does not display what is expected. See *Drawing Logs* for more information on setting drawing log flags in the VEE port.

Incidents are split into two categories:

- *Warnings* are *non-critical* incidents that the application developer may ignore. When such an incident is reported, the flags are set in the graphics context so that the application can read them. However, if they are not explicitly read, the incident will be ignored silently.

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

- *Errors* are *critical* incidents that the application developer should not ignore when developing. As with warnings, *drawing log flags* will be set in the graphics context. Additionally, an exception will be thrown when the display is flushed so that the developer is aware of the incident.

Any incident may be either a warning or an error, depending on how the VEE port reported it. The distinction is made through the value of the flag `DRAWING_LOG_ERROR`.

Default Behavior

When the VEE port reports an incident, it sets drawing log flags in the graphics context. Additionally, if the incident was an error, it sets the special flag `DRAWING_LOG_ERROR`.

Every time the display is flushed, the flags contained in its graphics context will be checked. If the flag `DRAWING_LOG_ERROR` is set — which means an error has been reported — the `flush` function will throw a `MicroUIException` with the code `DRAWING_ERROR`, and the values of the drawing log flags in its message. Afterward, the flags will be reset.

> **Warning:** This behavior can be disabled at build time. In this case, the flags will keep their values after the display is flushed, and no exceptions will be thrown.
>
> Therefore, the developer should **not** rely on the drawing log flags in the application workflow. They are meant to be used as a **debugging hint**.

If an exception is thrown, the application developer should use the flag values to find the cause of the error and fix it accordingly.

Explicit Checks

MicroUI only checks the drawing log flags automatically during a display flush. The developer may want to read the flag values between drawing operations to investigate the cause of an error. Two functions are provided to do so:

- `GraphicsContext.getAndClearDrawingLogFlags` will return the current values of the flags and reset them.
- `GraphicsContext.checkDrawingLogFlags` behaves like `GraphicsContext.getAndClearDrawingLogFlags`. However, it will also throw an exception if the flag `DRAWING_LOG_ERROR` is set, like it is done when the display is flushed.

For example, if a VEE port with no implementation to draw circles reports an error with the flag `DRAWING_LOG_NOT_IMPLEMENTED`, the application would behave as below.

```
// The VEE port has not implemented this function.
Painter.drawCircle(gc, 1, 2, 3);

// This throws a MicroUIException with the error code -13 (DRAWING_ERROR).
Display.getDisplay().flush();
```

The application developer could force a check of the drawing log flags:

```
// The VEE port has not implemented this function.
Painter.drawCircle(gc, 1, 2, 3);

// This throws a MicroUIException with the error code -13 (DRAWING_ERROR).
int flags = gc.checkDrawingLogFlags();
```

Or the developer could explicitly retrieve the value of the flags:

```
// The VEE port has not implemented this function.
Painter.drawCircle(gc, 1, 2, 3);

// This retrieves the values of drawing log flags.
int flags = gc.getAndClearDrawingLogFlags();
// This prints "80000001" (DRAWING_LOG_ERROR | DRAWING_LOG_NOT_IMPLEMENTED == 1 << 31 | 1 <<␣
↪0).
System.out.println(Integer.toHexString(flags));
```

Configuration

When releasing an application, the developer should disable the automatic check of drawing log flags performed when the display is flushed. Doing so will prevent exceptions from being thrown, which would cause an unexpected crash. It will also not clear the drawing log flags when the display is flushed.

Disabling this check can be done by setting the *constant* `com.microej.library.microui.impl.check-drawing-errors-on-flush` to `false` when building the application. If it is not set, it defaults to `true`.

Available Constants

MicroUI provides a set of constants to describe reported incidents. They are defined and documented in the class `GraphicsContext`.

Constant	Value	Description
`DRAWING_LOG_NOT_IMPLEMENTED`	`1 << 0`	This function is not implemented.
`DRAWING_LOG_FORBIDDEN`	`1 << 1`	This function must not be called in this situation.
`DRAWING_LOG_OUT_OF_MEMORY`	`1 << 2`	The system ran out of memory.
`DRAWING_LOG_CLIP_MODIFIED`	`1 << 3`	The VEE port modified clip values in the graphics context.
`DRAWING_LOG_LIBRARY_INCIDENT`	`1 << 29`	An incident occurred in an underlying library.
`DRAWING_LOG_UNKNOWN_INCIDENT`	`1 << 30`	An incident that does not match other flags occurred.
`DRAWING_LOG_ERROR`	`1 << 31`	Special flag denoting critical incidents.

The special value `DRAWING_SUCCESS` (defined as `0`) represents a state where no drawing log flags are set, so encountering this value means no incident was reported.

New flag constants may be added in future versions of MicroUI. Also, their actual values may change, and the developer should not rely on them.

Images

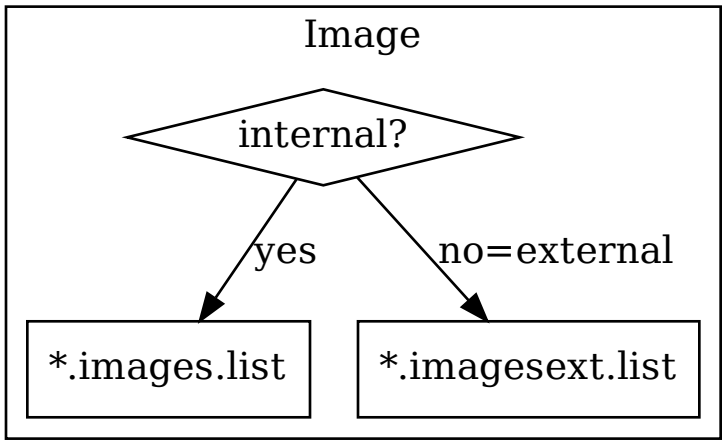
Immutable Images

Overview

Immutable images are graphical resources that can be accessed with a call to ej.microui.display.Image.getImage() or ej.microui.display.ResourceImage.loadImage(). As their name suggests, immutable images can not be modified. Therefore, there is no way to get a Graphics Context to draw into these images. To be displayed, these images have to be converted from their source format to a RAW format. The conversion can either be done:

- At build-time, using the Image Generator.
- At run-time, using the relevant decoder library.

Immutable images are declared in *Classpath* `*.images.list` files (**or** in `*.imagesext.list` for an external resource, see *External Images*).



The file format is a standard Java properties file. Each line contains a `/` -separated resource path relative to the Classpath root referring to a standard image file (e.g. `.png`, `.jpg`). The resource may be followed by an optional parameter (separated by a `:`) which defines and/or describes the image output file format (RAW format). When no option is specified, the image is embedded as-is and will be decoded at run-time. Example:

```
# The following image is embedded as
# a PNG resource (decoded at run-time)
com/mycompany/MyImage1.png

# The following image is embedded as
# a 16-bit encoding without transparency (decoded at build-time)
com/mycompany/MyImage2.png:RGB565

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

Configuration File

Here is the format of the `*.images.list` files.

```
ConfigFile          ::= Line [ 'EOL' Line ]*
Line                ::= ImagePath [ ':' ImageOption ]*
ImagePath           ::= Identifier [ '/' Identifier ]*
ImageOption         ::= [^:]*
Identifier          ::= Letter [ LetterOrDigit ]*
Letter              ::= 'a-zA-Z_$'
LetterOrDigit       ::= 'a-zA-Z_$0-9'
```

Unspecified Output Format

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

Refer to the platform specification to retrieve the list of runtime decoders.

Advantages

- Preserves the image characteristics.
- Preserves the original image compression.

Disadvantages

- Requires an image runtime decoder.
- Requires some RAM in which to store the decoded image.
- Requires execution time to decode the image.

```
image1
```

Display Output Format

It encodes the image into the exact display memory representation. If the image to encode contains some transparent pixels, the output file will embed the transparency according to the display's implementation capacity. When all pixels are fully opaque, no extra information will be stored in the output file in order to free up some memory space.

Note: When the display memory representation is standard, the display output format is automatically replaced by a standard format.

Advantages

- Drawing an image is very fast because no pixel conversion is required at runtime.
- Supports alpha encoding when the display pixel format allows it.

Disadvantages

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

```
image1:display
```

Standard Output Formats

Some image formats are well known and commonly implemented by GPUs.

Refer to the platform specification to retrieve the list of natively supported formats.

Advantages

- The pixel layout and bit format are standard, so it is easy to manipulate these images on the C-side.
- Drawing an image is very fast when the display driver recognizes the format (with or without transparency).

Disadvantages

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

Here is the list of the standard formats:

- Transparent images:
  - ARGB8888: 32-bit format, 8 bits for transparency, 8 per color,
  - ARGB4444: 16-bit format, 4 bits for transparency, 4 per color,
  - ARGB1555: 16-bit format, 1 bit for transparency, 5 per color.
- Transparent images with premultiplied alpha (RGB and alpha are linked)
  - ARGB8888_PRE: 32-bit format, 8 bits for transparency, 8 per color,
  - ARGB4444_PRE: 16-bit format, 4 bits for transparency, 4 per color,
  - ARGB1555_PRE: 16-bit format, 1 bit for transparency, 5 per color.
- Opaque images:
  - RGB888: 24-bit format, 8 per color,
  - RGB565: 16-bit format, 5 for red, 6 for green, 5 for blue.
- Alpha images, only transparency is encoded (the color applied when drawing the image is the current GraphicsContext color):
  - A8: 8-bit format,
  - A4: 4-bit format,
  - A2: 2-bit format,
  - A1: 1-bit format.

Examples:

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

Grayscale Output Formats

Some grayscale formats may be useful on grayscale or black and white displays.

Advantages

- Reduced footprint with less bits per pixels.

Disadvantages

- No compression: the image size in bytes is proportional to the number of pixels.
- Slower: a pixel conversion is required at runtime.

Here is the list of the grayscale formats:

- With transparency:
  - AC44: 4 bits for transparency, 4 bits with grayscale conversion,
  - AC22: 2 bits for transparency, 2 bits with grayscale conversion,
  - AC11: 1 bit for transparency, 1 bit with grayscale conversion.
- Without transparency:
  - C4: 4 bits with grayscale conversion,
  - C2: 2 bits with grayscale conversion,
  - C1: 1 bit with grayscale conversion.

Examples:

```
image1:AC44
image2:C2
```

Compressed Output Formats

Some image formats are compressed using run-length encoding. This compression is lossless. The principle is that identical consecutive pixels are stored as one entry (value and count). The more the consecutive pixels are identical, the more the compression is efficient.

Advantages

- Good compression when there are a lot of identical consecutive pixels.

Disadvantages

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

Here is the list of the compressed formats:

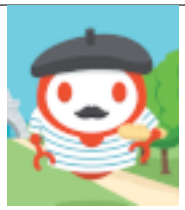
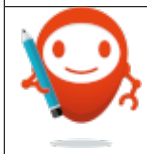
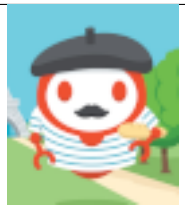
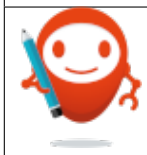
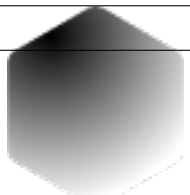
- ARGB1565_RLE: 16-bit format, 1 bit for transparency, 5 for red, 6 for green, 5 for blue. (Formerly named RLE1 up to UI Pack 13.3.X.)
- A8_RLE: similar to A8.

```
image1:ARGB1565_RLE
image2:RLE1 # Deprecated
image3:A8_RLE
```

Expected Result

The following table summarizes the usage of the different formats and the actual result on a white background.

Table 6: Image Output Formats Usage

Format	Source	Result
ARGB8888		
ARGB4444		
ARGB1555		
ARGB8888_PRE		
ARGB4444_PRE		
ARGB1555_PRE		
RGB888		
		
RGB565		
		
A8		

Usage Advice

- When the image is rarely used, or when there is little Flash and enough RAM: embed the image in its original compressed format (PNG or JPG for instance).
- For an opaque image: *RGB565* is usually sufficient.
- For a transparent image: *ARGB4444* is usually sufficient.
- For a transparent image that contains only shape(s) with horizontal or vertical edges:
  - *ARGB1555* may be interesting to have more colors,
  - for a smaller footprint if the image matches the RLE rule, *ARGB1565_RLE* is best.
- For a pictogram to colorize:
  - *A4* is usually sufficient,
  - *A8* may be necessary for pictograms with long gradients,
  - for a smaller footprint if the image matches the RLE rule, *A8_RLE* is best.
- For BSP with a GPU, choose a format compatible with the GPU (all formats may not be available),
  - *ARGB* formats: choose between non-premultiplied formats and premultiplied formats (suffixed with *_PRE*),
  - *Ax* formats (pictogram): all bits-per-pixel values may not be available.
  - be careful about the color components position (*A-R-G-B* versus *R-G-B-A* for instance),
  - avoid formats *Cx*, *ACxx* and *xxx_RLE*, which are not compatible with a GPU.

Caching Generated Images

Images converted using the Image Generator can be cached so that they are not rebuilt every time the application is launched. Doing so can significantly speed up the application build phase.

The cache is enabled by default. It may be disabled by setting the *Application option* `ej.microui.imageConverter.disableCache` to `true`.

The Image Generator obeys several rules when choosing whether an image should be converted.

- If the cache is disabled, all images are generated every time the application is launched.
- All images will be regenerated if the application is launched using another VEE port and the new VEE port uses a different Image Generator or another set of Image Generator plugins.
- If the generated image does not exist, it will be generated.
- If the source image has been modified since the last time it was converted, the image will be regenerated.
- The image will be regenerated if the destination format has been modified in the *images.list* file.

Cached images are stored in `.cache/images`, which is located in the *application output folder*. You may delete this directory to force the generation of all images in your application. An image that was previously generated but is no longer listed in the `*.images.list` files when the application is launched will be deleted from the cache directory.

> **Warning:** When *testing an Image Generator extension project*, the image cache is automatically disabled.

External Images

To fetch immutable images from external memory, the application must pre-register the *external Image resources*. The management of this kind of image may be different than the internal images and may require some allocations in the *Images Heap*. For more details about the external image management, refers to the VEE Port Guide chapter *External Resource*.

Image Generator Error Messages

These errors can occur while preprocessing images.

Table 7: Static Image Generator Error Messages

ID	Type	Description
0	Error	The image generator has encountered an unexpected internal error.
1	Error	The images list file has not been specified.
2	Error	The image generator cannot create the final, raw file.
3	Error	The image generator cannot read the images list file. Make sure the system allows reading of this file.
4	Warning	The image generator has found no image to generate.
5	Error	The image generator cannot load the images list file.
6	Warning	The specified image path is invalid: The image will be not converted.
7	Warning	There are too many or too few options for the desired format.
8	Error	The display format is not generic; a MicroUIRawImageGeneratorExtension implementation is required to generate the MicroUI raw image.
9	Error	The image cannot be read.
10	Error	The image generator has encountered an unexpected internal error (invalid endianness).
11	Error	The image generator has encountered an unexpected internal error (invalid bpp).
12	Error	The image generator has encountered an unexpected internal error (invalid display format).
13	Error	The image generator has encountered an unexpected internal error (invalid pixel layout).
14	Error	The image generator has encountered an unexpected internal error (invalid output folder).
15	Error	The image generator has encountered an unexpected internal error (invalid memory alignment).
16	Error	The input image format and / or the ouput format are not managed by the image generator.
17	Error	The image has been already loaded with another output format.

Mutable Images

Overview

Unlike immutable images, mutable images are graphical resources that can be created and modified at runtime. The application can draw into such images using the Painter classes with the image's Graphics Context as the destination. Mutable images can be created with a call to constructor ej.microui.display.BufferedImage().

```
BufferedImage image = new BufferedImage(320,  240);
GraphicsContext g = image.getGraphicsContext();
g.setColor(Colors.BLACK);
Painter.fillRectangle(g, 0, 0, 320, 240);
g.setColor(Colors.RED);
Painter.drawHorizontalLine(g, 50, 50, 100);
image.close();
```

Display Format

By default, the output format of a BufferedImage matches the display's pixel organization (layout, depth, etc.). The algorithms used to draw in such an image are the same as those used on the display (for footprint purposes). The algorithm cannot draw transparent pixels since the display buffer is opaque.

In addition, GraphicsContext.setColor() does not consider the alpha channel and only accepts RGB values. The given color value is interpreted as a 24-bit RGB color, where the high-order byte is ignored, and the remaining bytes contain the red, green, and blue channels, respectively.

Other Formats

It is also possible to create a buffered image with another format using the constructor with the format parameter.

The other formats than the display one are not supported by MicroUI. But a VEE port can manage one or more formats (see *Destination Format*).

Depending on the format, the transparency may be supported.

Images Heap

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

- Mutable images (i.e. BufferedImage instances).
- Immutable images decoded at runtime, typically a PNG: the heap is used to store the decoded image **and** the runtime decoder's temporary buffers, required during the decoding step. After the decoding step, all the temporary buffers are freed. Note that the size of the temporary buffers depends on the decoder **and** on the original image itself (compression level, pixel encoding, etc.).
- Immutable images which are not byte-addressable, such as images opened with an input stream (i.e. ResourceImage instances).
- Immutable images which are byte-addressable but converted to a different output format (i.e. ResourceImage instances).

In other words, every image which cannot be retrieved using ej.microui.display.Image.getImage() is saved on the image heap.

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

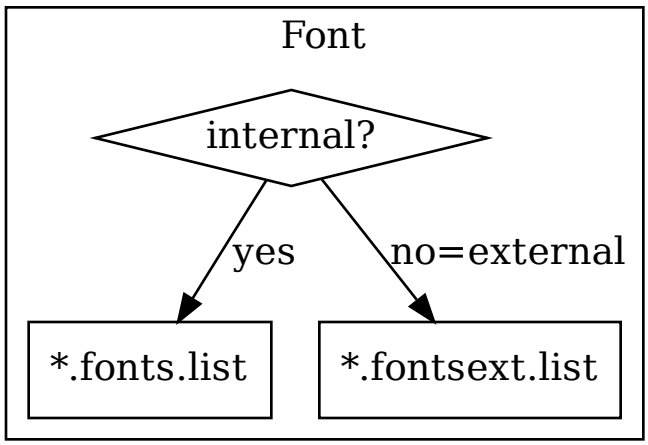
> **Warning:** A ResourceImage allocated on the images heap must be closed manually by the application (ResourceImage.close()); otherwise, a memory leak will occur on the images heap.

For more details about the images heap implementation, refers to *this chapter* in the VEE Port Guide.

Fonts

Overview

Fonts are graphical resources that can be accessed with a call to ej.microui.display.Font.getFont(). Fonts are declared in *Classpath* `*.fonts.list` files (**or** in `*.fontsext.list` for an external resource, see *External Fonts*).



The file format is a standard Java properties file, each line representing a `/` separated resource path relative to the Classpath root referring to a Font file (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
```

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://home.unicode.org/).

Table 8: Ranges

Name	Tag	Start	End
Basic Latin	basic_latin	0x0	0x7f
Latin-1 Supplement	latin-1_supplement	0x80	0xff
Latin Extended-A	latin_extended-a	0x100	0x17f
Latin Extended-B	latin_extended-b	0x180	0x24f
IPA Extensions	ipa_extensions	0x250	0x2af
Spacing Modifier Letters	spacing_modifier_letters	0x2b0	0x2ff
Combining Diacritical Marks	combining_diacritical_marks	0x300	0x36f
Greek and Coptic	greek_and_coptic	0x370	0x3ff
Cyrillic	cyrillic	0x400	0x4ff
Cyrillic Supplement	cyrillic_supplement	0x500	0x52f
Armenian	armenian	0x530	0x58f
Hebrew	hebrew	0x590	0x5ff
Arabic	arabic	0x600	0x6ff
Syriac	syriac	0x700	0x74f
Arabic Supplement	arabic_supplement	0x750	0x77f
Thaana	thaana	0x780	0x7bf
NKo	nko	0x7c0	0x7ff
Samaritan	samaritan	0x800	0x83f
Mandaic	mandaic	0x840	0x85f
Arabic Extended-A	arabic_extended-a	0x8a0	0x8ff
Devanagari	devanagari	0x900	0x97f
Bengali	bengali	0x980	0x9ff
Gurmukhi	gurmukhi	0xa00	0xa7f
Gujarati	gujarati	0xa80	0xaff
Oriya	oriya	0xb00	0xb7f
Tamil	tamil	0xb80	0xbff
Telugu	telugu	0xc00	0xc7f
Kannada	kannada	0xc80	0xcff
Malayalam	malayalam	0xd00	0xd7f
Sinhala	sinhala	0xd80	0xdff
Thai	thai	0xe00	0xe7f
Lao	lao	0xe80	0xeff
Tibetan	tibetan	0xf00	0xfff
Myanmar	myanmar	0x1000	0x109f
Georgian	georgian	0x10a0	0x10ff
Hangul Jamo	hangul_jamo	0x1100	0x11ff
Ethiopic	ethiopic	0x1200	0x137f
Ethiopic Supplement	ethiopic_supplement	0x1380	0x139f

continues on next page

Table 8 – continued from previous page

Name	Tag	Start	End
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
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

continues on next page

Table 8 – continued from previous page

Name	Tag	Start	End
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
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

continues on next page

Table 8 – continued from previous page

Name	Tag	Start	End
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
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

External Fonts

To fetch fonts from non-byte addressable external memory, the application must pre-register the *external Font resources*. The management of this kind of font may be different than the internal fonts and may require a dedicated heap. For more details about the external font management, refers to the VEE Port Guide chapter *External Resources*.

Font Generator Error Messages

Table 9: Static Font Generator Error Messages

ID	Type	Description
0	Error	The font generator has encountered an unexpected internal error.
1	Error	The Fonts list file has not been specified.
2	Error	The font generator cannot create the final, raw file.
3	Error	The font generator cannot read the fonts list file.
4	Warning	The font generator has found no font to generate.
5	Error	The font generator cannot load the fonts list file.
6	Warning	The specified font path is invalid: The font will be not converted.
7	Warning	There are too many arguments on a line: the current entry is ignored.
8	Error	The font generator has encountered an unexpected internal error (invalid output format).
9	Error	The font generator has encountered an unexpected internal error (invalid endianness).
10	Error	The specified entry is invalid.
11	Error	The specified entry does not contain a list of characters.
12	Error	The specified entry does not contain a list of identifiers.
13	Error	The specified entry is an invalid width.
14	Error	The specified entry is an invalid height.
15	Error	The specified entry does not contain the characters' addresses.
16	Error	The specified entry does not contain the characters' bitmaps.
17	Error	The specified entry bits-per-pixel value is invalid.
18	Error	The specified range is invalid.
19	Error	There are too many identifiers. The output RAW format cannot store all identifiers.
20	Error	The font's name is too long. The output RAW format cannot store all name characters.
21	Error	There are too many ranges. The output RAW format cannot store all ranges.
22	Error	Output list files cannot be created.
23	Error	Dynamic styles are not supported. Only a PLAIN font can be encoded.
24	Error	Underlined style is not supported. Only a BOLD and ITALIC font can be set.

Default Character

The application may request the rendering of a string where some characters are not available in the selected font. In that case, a default character is drawn instead: it is the first available character in the font. For example, the first available character for a font where the range matches the ASCII printable characters (`0x21-0x7E`) would be the exclamation mark (`0x21`).

The characters of a font are referenced by their Unicode value. For a given *font range*, the default character is the first character of the first range. Consequently, the default character may not be the same for two given fonts of an application: it depends on the specified character range for each font.

To help developers identify quickly why a string is rendered with unexpected characters, it is recommended that the font maker sets a default character that is easy to recognize (a symbol, for example, a rectangle). This character must have the first character index (index `0` is allowed).

Caching Generated Fonts

Fonts converted using the Font Generator can be cached so that they are not rebuilt every time the application is launched. Doing so can significantly speed up the application build phase.

The cache is enabled by default. It may be disabled by setting the *Application option* `ej.microui.fontConverter.disableCache` to `true`.

The Font Generator obeys several rules when choosing whether a font should be converted.

- If the cache is disabled, all fonts are generated every time the application is launched.
- All fonts will be regenerated if the application is launched using another VEE port and the new VEE port uses a different Font Generator.
- If the generated font does not exist, it will be generated.
- If the source font has been modified since the last time it was converted, the font will be regenerated.
- The font will be regenerated if the destination format or the range has been modified in the *fonts.list* file.

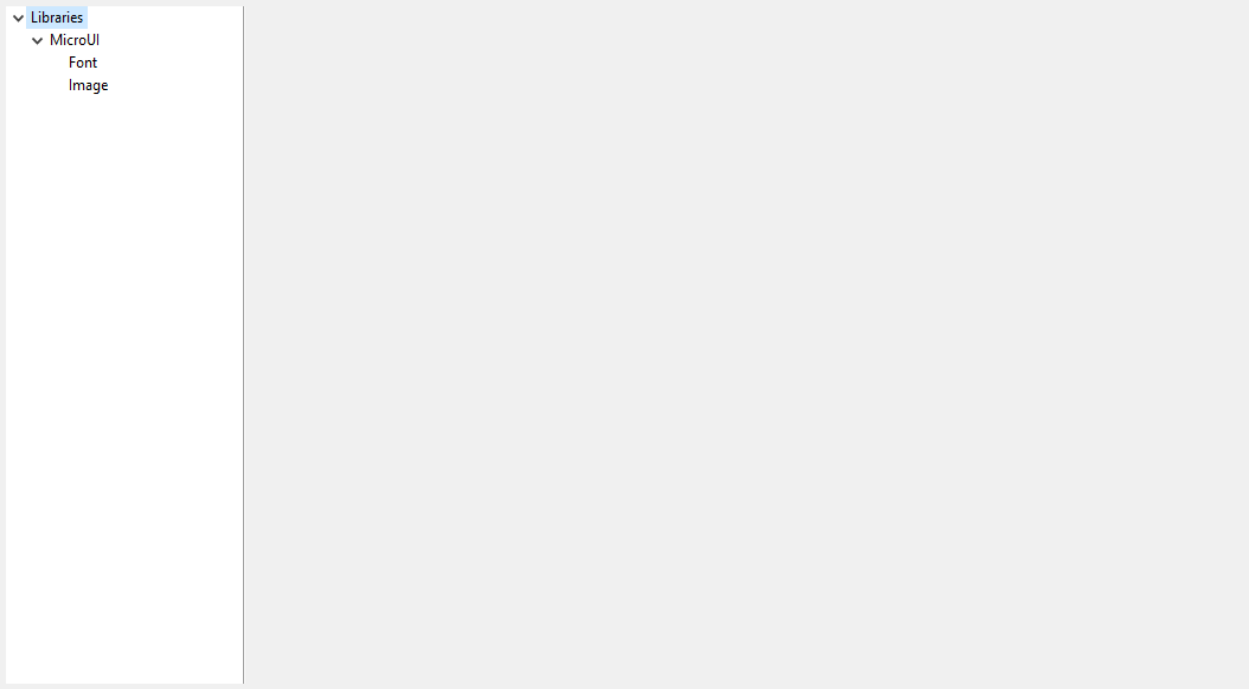
Cached fonts are stored in `.cache/fonts`, which is located in the *application output folder*. You may delete this directory to force the generation of all fonts in your application. A font that was previously generated but is no longer listed in the `*.fonts.list` files when the application is launched will be deleted from the cache directory.

Application Options

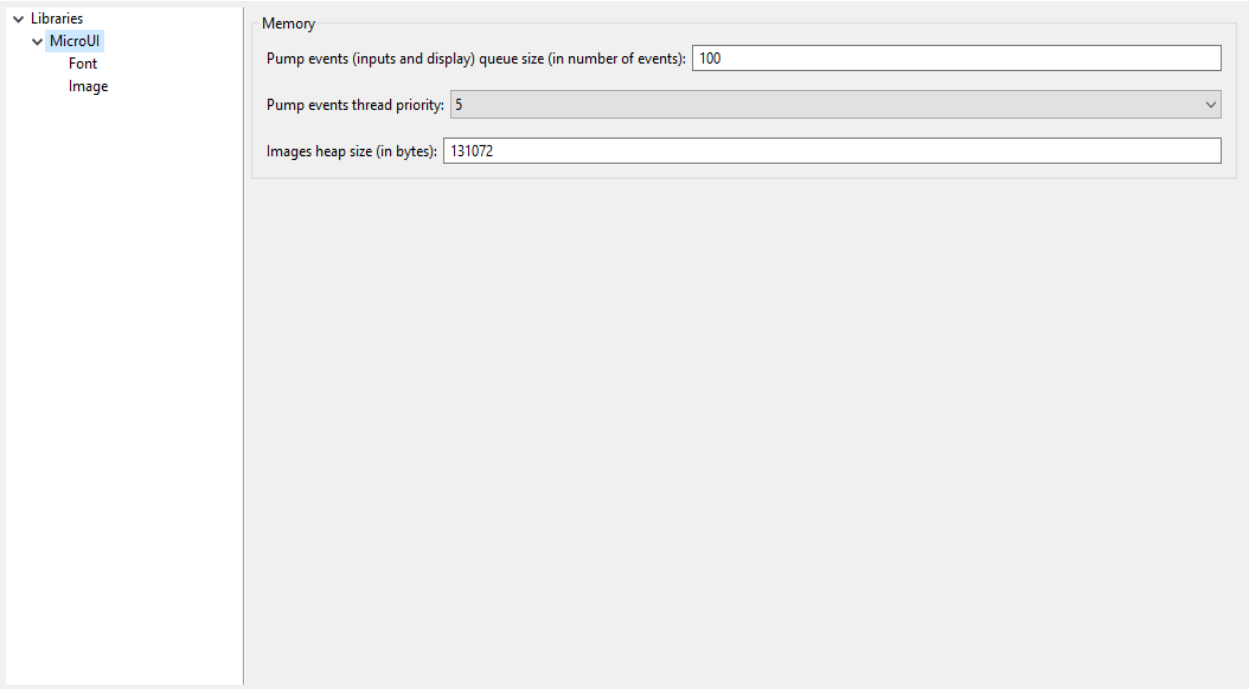
MicroUI libraries and its tools provide a set of options. See *Standalone Application Options* to have more information about the application options.

Note: MicroUI implementation requires one thread (MicroUI Pump) and at least 100 bytes in the *immortals heap*.

Category: Libraries



Category: MicroUI



Group: Memory

Option(text): Pump events (inputs and display) queue size (in number of events)

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

Default value: `100`

Description:

Specifies the size of the pump events queue.

Option(combo): Pump events thread priority

Option Name: `com.microej.library.microui.pump.priority`

Default value: `5`

Available values: `1` to `10`

Description:

Specifies the priority of the pump events queue.

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

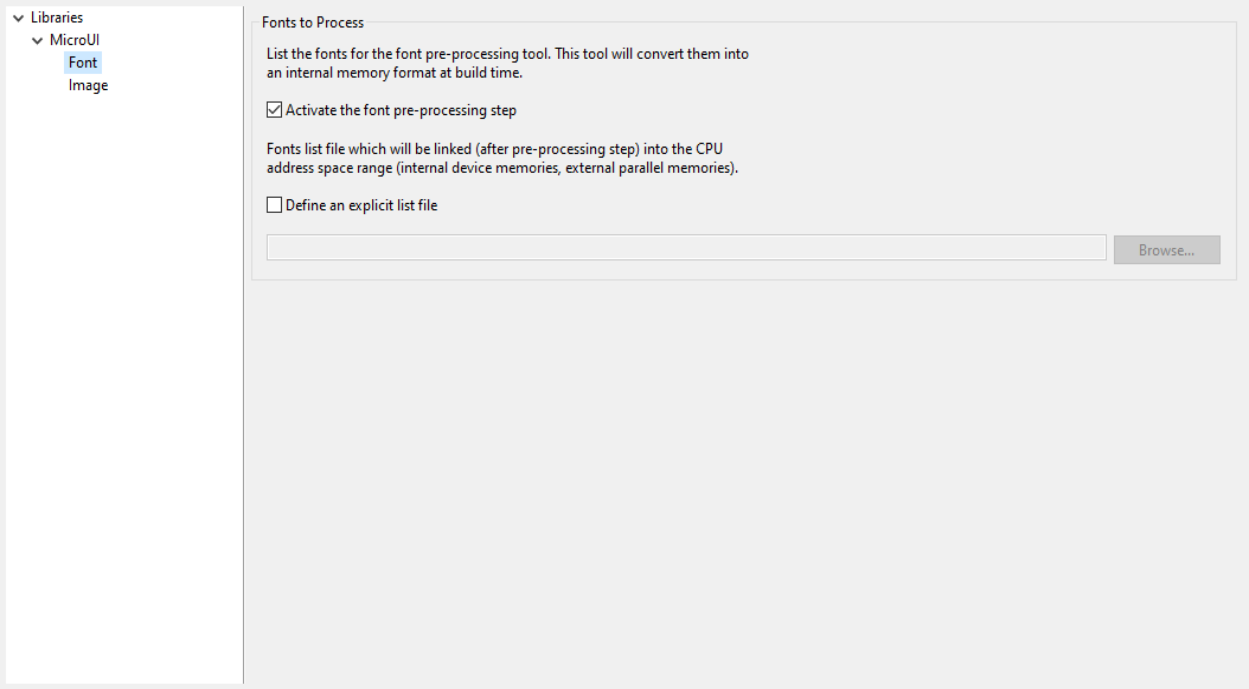
Option Name: `ej.microui.memory.imagesheap.size`

Default value: `131072`

Description:

Specifies the size of the images heap. This heap is used to store the dynamic user images, the decoded images and the working buffers of embedded image decoders (for instance the PNG decoder). A too small value can cause OutOfMemory errors and incomplete drawings.

Category: Font



Group: Fonts to Process

Description:

This group allows to select a file describing the font files which need to be converted into a RAW format. At MicroUI runtime, the pre-generated fonts will be read from the flash memory without any modifications (see MicroUI specification).

Option(checkbox): Activate the font pre-processing step

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

Default value: `true`

Description:

When checked, enables the next option `Fonts list` file. When the next option is disabled, there is no check on the file path validity.

Option(checkbox): Define an explicit list file

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

Default value: `false`

Description:

By default, list files are loaded from the classpath. When checked, only the next option `Fonts list` file is processed.

Option(browse):

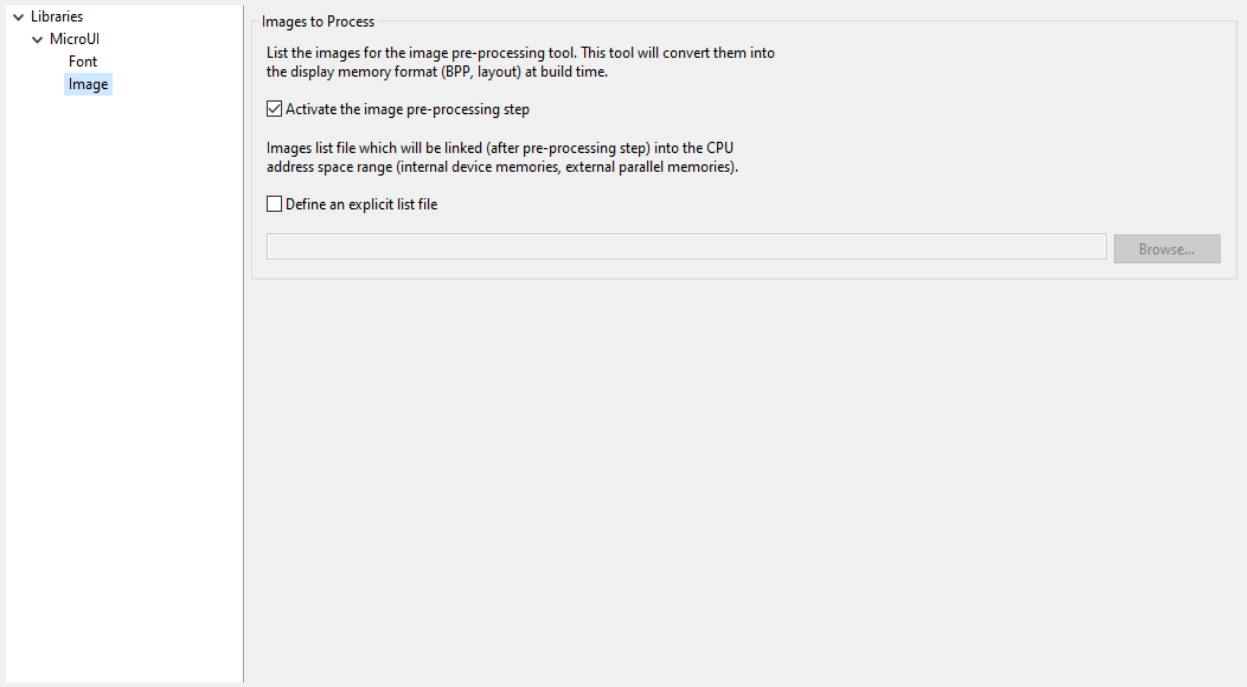
Option Name: `ej.microui.fontConverter.file`

Default value: `(empty)`

Description:

Browse to select a font list file. Refer to Font Generator chapter for more information about the font list file format.

Category: Image



Group: Images to Process

Description:

This group allows to select a file describing the image files which need to be converted into a RAW format. At MicroUI runtime, the pre-generated images will be read from the flash memory without any modifications (see MicroUI specification).

Option(checkbox): Activate the image pre-processing step

Option Name: `ej.microui.imageConverter.useIt`

Default value: `true`

Description:

When checked, enables the next option `Images list` file. When the next option is disabled, there is no check on the file path validity.

Option(checkbox): Define an explicit list file

Option Name: `ej.microui.imageConverter.file.enabled`

Default value: `false`

Description:

By default, list files are loaded from the classpath. When checked, only the next option `Images list` file is processed.

Option(browse):

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

Default value: `(empty)`

Description:

Browse to select an image list file. Refer to Image Generator chapter for more information about the image list file format.

Debug Traces

MicroUI logs several actions when traces are enabled. This chapter explains the trace identifiers.

Note: Most of the logs are only available on the Embedded VEE Port (not on the Simulator).

Trace format

The trace output format is the following:

```
[TRACE: MicroUI] Event AA(BB[CC],DD[EE])
```

where:

- AA is the event identifier. See next table.
- BB is the first event data.
- CC is the index of the first event data (0x0).
- DD is the second event data.
- EE is the index of the second event data (0x1).

- etc.

For example, given the following trace output:

```
[TRACE: MicroUI] Event 0x2(1[0x0],2[0x1],117571586[0x2])
```

- 0x2 -> Execute native input event
- 1 -> Event “Button” (index 0x0)
- 2 -> Generator Id (index 0x1)
- 117571586 -> event data (index 0x2)

Trace identifiers

The following tables describe some events data.

Table 10: 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 11: 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 12: Create Image

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

Table 13: 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 14: 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

continues on next page

Table 14 – 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)

SystemView Integration

The traces are *SystemView* compatible.

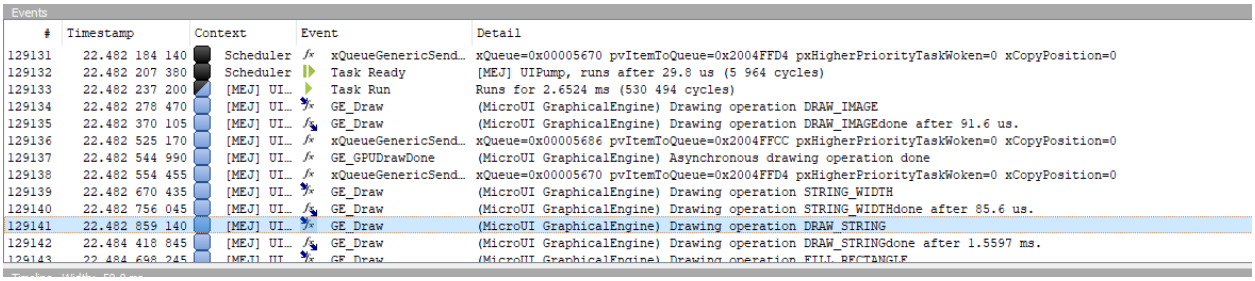


Fig. 15: MicroUI Traces displayed in SystemView

The following text can be copied in a file called `SYSVIEW_MicroUI.txt` and copied in SystemView installation folder (e.g. `SEGGER/SystemView_V252a/Description/`).

```
NamedType UIEvent 0=COMMAND
NamedType UIEvent 1=BUTTON
NamedType UIEvent 2=POINTER
NamedType UIEvent 3=STATE
NamedType UIEvent 4=UNKNOWN
NamedType UIEvent 5=CALLSERIALLY
NamedType UIEvent 6=STOP
NamedType UIEvent 7=INPUT
NamedType UIEvent 8=SHOW_DISPLAYABLE
NamedType UIEvent 9=HIDE_DISPLAYABLE
NamedType UIEvent 11=PENDING_FLUSH
NamedType UIEvent 12=FORCE_FLUSH
NamedType UIEvent 13=REPAINT_DISPLAYABLE
NamedType UIEvent 14=REPAINT_CURRENT_DISPLAYABLE
NamedType UIEvent 15=KF_STOP_FEATURE

NamedType 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
```

(continues on next page)

(continued from previous page)

```
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
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
```

(continues on next page)

(continued from previous page)

```
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 15: 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).
-12	The maximum number of event generators in the pool (254) has been reached.

Migration Guide

The MicroUI implementation is provided by the MicroEJ UI Pack. According the MicroEJ UI Pack used to build the MicroEJ Platform, the application has to be updated.

- Refer to the *table* that illustrates the implemented MicroUI API for each MicroEJ UI Pack.
- Refer to the latest MicroUI API Changelog.
- Refer to the latest Drawing API Changelog.

The following chapters describe the changes to perform in the application according the MicroEJ UI Pack used to build the MicroEJ Platform.

From 12.x to 13.x

- Update ej.api#microui dependency to the latest available version `3.x`.
- Add ej.api#drawing dependency.

Gradle (build.gradle.kts)

MMM (module.ivy)

```
implementation("ej.api:microui:3.1.0")
implementation("ej.api:drawing:1.0.3")
```

```
<dependencies>
   <dependency org="ej.api" name="microui" rev="3.1.0"/>
   <dependency org="ej.api" name="drawing" rev="1.0.3"/>
</dependencies>
```

From 10.x to 12.x

- In MicroEJ application launcher > `Configuration` tab > *MicroUI*: check `Use Flying Images` when the application is using the flying images (property `com.microej.library.microui.flyingimage.enabled`).
- In MicroEJ application launcher, increase the *Java heap*: it now contains MicroUI images metadata (size, format, clip etc.). The iceatea heap has been automatically decreased.

From 9.x to 10.x

- In MicroEJ application launcher > `Configuration` tab > *MicroUI*: set the image heap size (property *ej.microui.memory.imagesheap.size*).

MicroVG

Introduction

MicroVG Foundation Library provides vector drawing capabilities.

To use the MicroVG Foundation Library, add MicroVG API module to the project build file:

Gradle (build.gradle.kts)

MMM (module.ivy)

```
implementation("ej.api:microvg:1.2.0")
```

```
<dependency org="ej.api" name="microvg" rev="1.2.0"/>
```

The MicroVG Library brings the following features:

- the creation and drawing of paths with color or linear gradient.
- the drawing of texts using vector fonts with color or linear gradient.
- the drawing of vector images.

- the transformation of paths, texts, images with affine transformation matrices.

Note: The MicroVG library natives use different drawing engines, font rendering and layout engines for embedded and simulator implementations.

This can lead to some slightly drawing differences, like for instance in the antialiasing processing of font glyphs.

Path

Path Creation

The MicroVG library enables the creation of vector paths composed of the following commands:

- Move
- Line
- Cubic Bezier Curve
- Quadratic Bezier Curve
- Close

The coordinates of the points associated with these commands can be absolute or relative.

```
Path path = new Path();

path.moveTo(70, 20);
path.cubicTo(0, 0, 10, 50, 80, 90);
path.lineTo(95, 75);
path.quadTo(12, 40, 80, 50);
path.close();
```

```
Path path = new Path();

path.moveTo(70, 20);
path.cubicToRelative(-70, -20, -60, 30, 10, 70);
path.lineToRelative(15, -15);
path.quadToRelative(-83, -35, -15, -25);
path.close();
```

Path Drawing

A path can be drawn with a call to ej.microvg.VectorGraphicsPainter.fillPath().

The drawn path will be filled with the graphic context color or with a linear gradient.

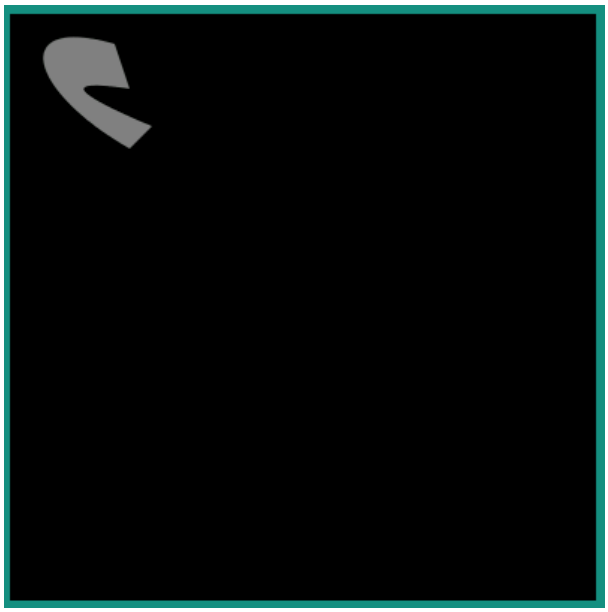
The path can be transformed by a transformation matrix (this concept is explained in *Matrix* section) before drawing.

A *FillType* and an *Alpha Blending Mode* can be applied during the drawing.

Fill Path With Graphics Context Color

The default alpha channel value of the drawing is `0xFF` (opaque opacity).

```
g.setColor(Colors.GRAY);
VectorGraphicsPainter.fillPath(g, path, 0, 0);
```

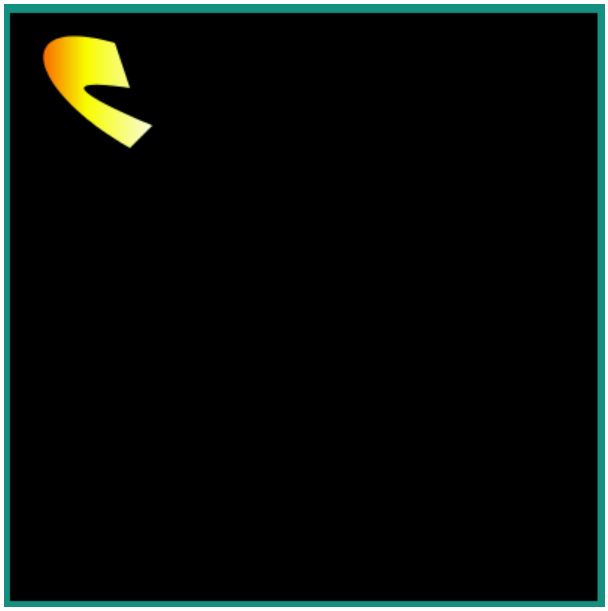


Fill Path With a Linear Gradient

Refer to *Linear Gradient* section for more details about the definition of a linear gradient.

The opacity value of the drawing is defined by the Alpha channel of the ARGB color values of the each linear gradient stop point.

```
LinearGradient gradient = new LinearGradient(0, 0, 100, 0, new int[] { 0xffff0000,␣
↪0xffffff00, 0xffffffff });
VectorGraphicsPainter.fillPath(g, path, new Matrix(), gradient);
```



Fill Type

A path can be drawn with a *FillType* argument. This argument defines the way a path will be filled.

The following values are a available:

- FillType.Winding: Specifies that “inside” is computed by a non-zero sum of signed edge crossings.
- FillType.EVEN_ODD: Specifies that “inside” is computed by an odd number of edge crossings.

```
Path path = new Path();

path.moveTo(50, 0);
path.lineTo(21, 90);
path.lineTo(98, 35);
path.lineTo(2, 35);
path.lineTo(79, 90);
path.close();
```



Opacity and Blending Mode

The opacity of the drawing can be provided to the *fillPath* method with a blending mode.

When the drawing is done with graphic context color, the given alpha value replaces the default value (`0xFF`).

When the drawing is done with a linear gradient, the given alpha is applied above each gradient colors alpha channel values(0x80 alpha value on `#80FFFFFF` ARGB color leads to `#40FFFFFF` color).

The supported blending modes are:

- `SRC` : The source pixels replace the destination pixels.
- `SRC_OVER` : The source pixels are drawn over the destination pixels.
- `DST_OVER` : The source pixels are drawn behind the destination pixels.
- `SRC_IN` : Keeps the source pixels that cover the destination pixels, discards the remaining source and destination pixels.
- `DST_IN` : Keeps the destination pixels that cover source pixels, discards the remaining source and destination pixels.
- `DST_OUT` : Keeps the destination pixels that are not covered by source pixels. Discards destination pixels that are covered by source pixels. Discards all source pixels.
- `PLUS` : Adds the source pixels to the destination pixels and saturates the result.
- `SCREEN` : Adds the source and destination pixels, then subtracts the source pixels multiplied by the destination.
- `MULTIPLY` : Multiplies the source and destination pixels.



Fig. 16: SRC



Fig. 17: SRC_OVER



Fig. 18: DST_OVER



Fig. 19: SRC_IN



Fig. 20: DST_IN



Fig. 21: DST_OUT



Fig. 22: PLUS



Fig. 23: SCREEN

Matrix



g. 24: ULTIPLY

A *Matrix* is composed of an array of numbers with three rows and three columns. It is used to apply an affine transformations to *Path* points. (Refer to https://en.wikipedia.org/wiki/Transformation_matrix#Affine_transformations to get more information about affine transformations).

The available transformations are:

- translation
- rotation

- scaling

  Scaling and rotation are always performed around the (0,0) pivot point. In order to rotate or scale a *Path* with a pivot point, the matrix must be translated before and after the rotation/scaling.

  A *Matrix* is created as an identity matrix, which means that a *Path* resulting of a transformation with this matrix is identical to the original *Path*.

  The *Matrix* can be initialized with a transformation with set methods:

- *setTranslate(translateX, translateY)*
- *setRotate(angle)*
- *setScale(scaleX, scaleY)*

  A transformation can be prepended to a *Matrix* with the prepend methods:

- *preTranslate(translateX, translateY)*
- *preRotate(angle)*
- *preScale(scaleX, scaleY)*

  A transformation can be appended to a *Matrix* with the append methods:

- *postTranslate(translateX, translateY)*
- *postRotate(angle)*
- *postScale(scaleX, scaleY)*

  A *Matrix* can also get transformations from an other *Matrix* with the concatenate and set methods:

- *preConcat(matrix)*
- *postConcat(matrix)*
- *set(matrix)*
- *setConcat(matrix0, matrix1)*

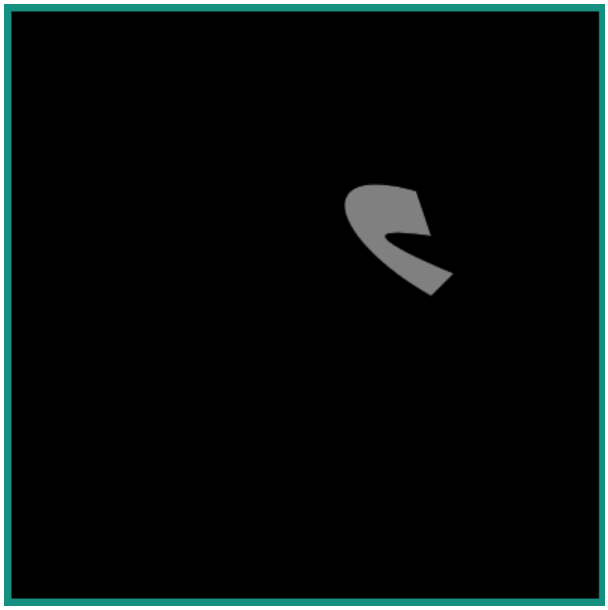
  Once a *Matrix* has been computed, it can be used to draw an object (*Path*, *String*, *VectorImage*). All the points of the drawn object will be transformed by the *Matrix*.

  When a *Matrix* has been computed with multiple type of transformation, the sequence order of the transformation is important. Chaining the transformations in a different order will not provide the same *Matrix*. The result of the previous transformation is the input to the next transformation.

  The following examples use the *Path* created in the section *Path Creation* with different transformations.

Translation

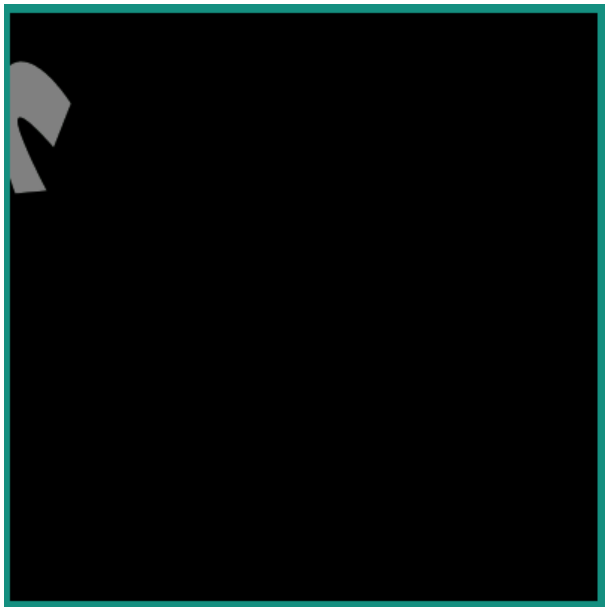
```
Matrix matrix = new Matrix();
matrix.setTranslate(200, 150);
```



Rotation

Around point (0,0).

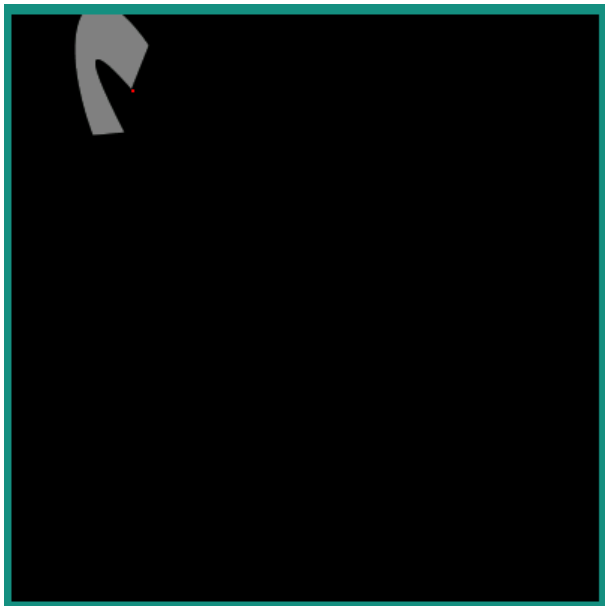
```
Matrix matrix = new Matrix();
matrix.setRotate(40);
```



Around a pivot point (80,50).

```
Matrix matrix = new Matrix();
matrix.setRotate(40);

float pivotX = 80;
float pivotY = 50;
matrix.preTranslate(-pivotX, -pivotY);
matrix.postTranslate(pivotX, pivotY);
```



Scale

From point (0,0).

```
Matrix matrix = new Matrix();
matrix.setScale(2,3);
```



Concatenate Matrixes

Sequence order has an incidence on the rendering.

```
Matrix matrix0 = new Matrix();
matrix0.setScale(2, 3);

Matrix matrix1 = new Matrix();
matrix1.setTranslate(100, 40);

Matrix matrix2 = new Matrix();
matrix2.setConcat(matrix0, matrix1);

g.setColor(Colors.GRAY);
VectorGraphicsPainter.fillPath(g, path, matrix2);
```

(continues on next page)

(continued from previous page)

```
matrix2.setConcat(matrix1, matrix0);

g.setColor(Colors.YELLOW);
VectorGraphicsPainter.fillPath(g, path, matrix2);
```



Linear Gradient

The MicroVG library supports the drawing of shapes with a linear gradient of color.

A linear gradient is specified by a linear segment and a set of ARGB colors associated with points on that segment.

The colors along the segment between those points are calculated using linear interpolation, then extended perpendicular to that line.

The position of the color points on the segment are given from `0.0f` (start of point) to `1.0f` (end of the segment).

There are two ways to create a gradient:

- with a start point, an end point and a color table: the first color will be applied to the start point, the second color to the end point and other colors distributed evenly along the gradient segment.

```
Path path = new Path();
path.moveTo(0, 0);
path.lineTo(100, 0);
```

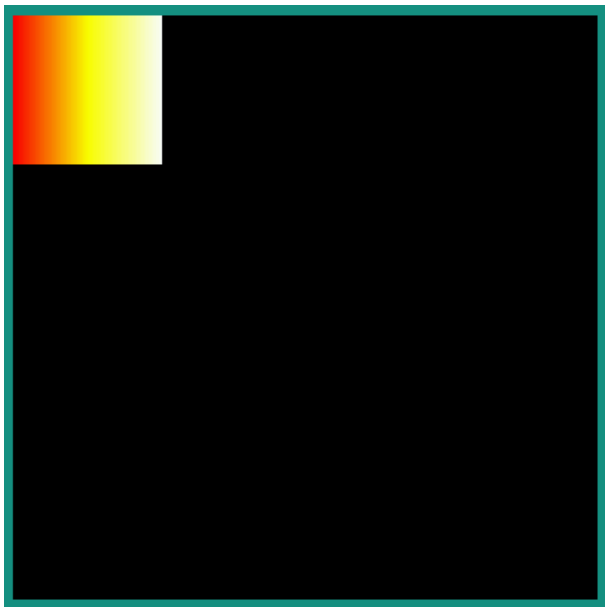
(continues on next page)

(continued from previous page)

```
path.lineTo(100, 100);
path.lineTo(0, 100);
path.close();

LinearGradient gradient = new LinearGradient(0, 0, 99, 0,
                                                                    ␣
↪                                   new int[] { 0xffff0000, 0xffffff00, 0xffffffff });

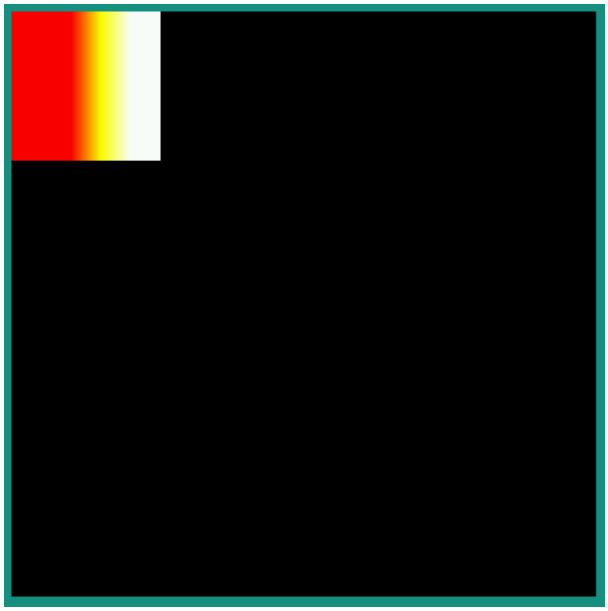
VectorGraphicsPainter.fillPath(g, path, new Matrix(), gradient);
```



- with a start point, an end point, a color table and a position table: the colors are applied to their corresponding relative positions on the segment. If the first point is not the start point of the segment, then first color is applied from the start of the segment to the first point. If the last point is not the end point of the segment, then last color is applied from the last point to the end of the segment.

```
LinearGradient gradient = new LinearGradient(0, 0, 99, 0,
                                                                    ␣
↪                                   new int[] { 0xffff0000, 0xffffff00, 0xffffffff },
                                                                  ␣
↪                                                 new float[] { 0.4f, 0.6f, 0.8f });

VectorGraphicsPainter.fillPath(g, path, new Matrix(), gradient);
```

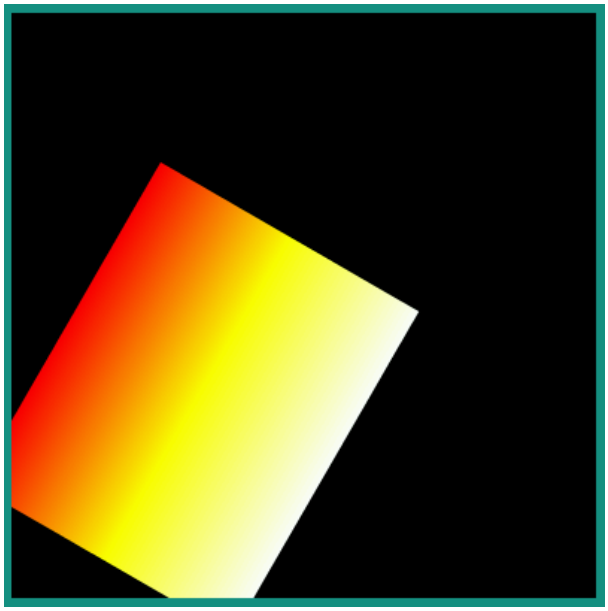


The transformation applied to the object (*Path* or *String*) to draw with a gradient is also applied to that gradient. The *LinearGradient* is not updated after the drawing.

```
LinearGradient gradient = new LinearGradient(0, 0, 99, 0,
                                                ␣
↪                                new int[] { 0xffff0000, 0xffffff00, 0xffffffff });

Matrix matrix = new Matrix();
matrix.setScale(2, 2.5f);
matrix.postRotate(30);
matrix.postTranslate(100, 100);

VectorGraphicsPainter.fillPath(g, path, matrix, gradient);
```



Vector Fonts

Overview

The MicroVG library enables the usage of Vector Fonts.

Compared to MicroUI *Fonts*, Vector Fonts brings the following features:

- the text strings are scalable and can be transformed using a Matrix object.
- the TTF/OTF font files don't need to be preprocessed.
- the text strings can be drawn with opacity, a color or a linear gradient.

The library also considers the *Kerning* space described in the font file kerning table, and allows a fine adjustement of the inter-letters spacing.

It also provides metrics measurement methods to correctly place the text within the surrounding drawing elements (i.e. in a label).

Loading a Font File

A Vector Font has to be loaded in a VectorFont object with a call to ej.microvg.VectorFont.loadFont(). This VectorFont object can then be used to draw text strings.

The fonts are decoded at runtime. They don't need to be pre-processed by some generator tool like *MicroUI Fonts* Vector Font files must be declared as resources in a `.resources.list` file available in the classpath (*Application Resources*). To declare them as *external resources*, the font files must be declared too in a `.externresources.list` file.

Text String Drawing

A string can be drawn in the graphics context with a call to ej.microvg.VectorGraphicsPainter.drawString().

The text string height is scalable, and multiple font files can be used in parrallel.

```
VectorFont font0 = VectorFont.loadFont("/fonts/Arial.ttf");
VectorFont font1 = VectorFont.loadFont("/fonts/RAVIE.ttf");

int x = 20;
int y = 30;
int yOffset = 150;

g.setColor(Colors.LIME);
VectorGraphicsPainter.drawString(g, "Hello MicroEJ", font0, 20, x, y);
VectorGraphicsPainter.
↪drawString(g, "Hello MicroEJ", font1, 20, x, y + yOffset);

g.setColor(Colors.RED);
y += 20;
VectorGraphicsPainter.drawString(g, "Hello MicroEJ", font0, 30, x, y);
VectorGraphicsPainter.
↪drawString(g, "Hello MicroEJ", font1, 30, x, y + yOffset);

g.setColor(Colors.WHITE);
y += 30;
VectorGraphicsPainter.drawString(g, "Hello MicroEJ", font0, 40, x, y);
VectorGraphicsPainter.
↪drawString(g, "Hello MicroEJ", font1, 40, x, y + yOffset);

g.setColor(Colors.YELLOW);
y += 40;
VectorGraphicsPainter.drawString(g, "Hello MicroEJ", font0, 50, x, y);
VectorGraphicsPainter.
↪drawString(g, "Hello MicroEJ", font1, 50, x, y + yOffset);

display.flush();
```



Text Color

The text string can be colored with the graphics context color or a with a linear gradient(*Linear Gradient*).

FillType and *Alpha Blending Mode* are also managed similarly to *Path* drawing (refer to *Fill Type* and *Opacity and Blending Mode*).

```
g.setColor(Colors.LIME);
VectorGraphicsPainter.drawString(g, "Hello MicroEJ", font, 50, x, y);

LinearGradient gradient = new LinearGradient(0, 0, 250, 50,
      ␣
↪                          new int[] { 0xffff0000, 0xffffff00, 0xffffffff });

Matrix matrix = new Matrix();
matrix.setTranslate(x, y + 60);
VectorGraphicsPainter.
↪drawGradientString(g, "Hello MicroEJ", font, 50, matrix, gradient, 0xff,
                BlendMode.SRC_OVER, 0);
```



Text Transformations

The text string can also be transformed with a Matrix to translate, rotate, scale the drawing.

```
Matrix matrix0 = new Matrix();

matrix0.setTranslate(20, 60);
VectorGraphicsPainter.drawString(g,
↪ "Hello MicroEJ", font, 50, matrix0, 0xff, BlendMode.SRC_OVER, 0);

matrix0.preRotate(180);
matrix0.postTranslate(300, 120);
VectorGraphicsPainter.drawString(g,
↪ "Hello MicroEJ", font, 50, matrix0, 0xff, BlendMode.SRC_OVER, 0);

Matrix matrix1 = new Matrix();
matrix1.setScale(0.5f, 1.2f);
matrix1.postRotate(45);
matrix1.postTranslate(80, 200);

VectorGraphicsPainter.drawString(g,
↪ "Hello MicroEJ", font, 50, matrix1, 0xff, BlendMode.SRC_OVER, 0);

matrix1.setScale(0.5f, 1.2f);
matrix1.postRotate(-45);
matrix1.postTranslate(200, 300);
VectorGraphicsPainter.drawString(g,
↪ "Hello MicroEJ", font, 50, matrix1, 0xff, BlendMode.SRC_OVER, 0);
```



Letter Spacing

The inter character distance can be adjusted for each string drawing. By default, the inter character distance is computed from the font file metrics, considering kerning, if the font file includes a kerning table. It can be adjusted with the `letterSpacing` parameter of drawString(). Its default value is 0 pixel, a positive/negative value will increase/reduce the inter space distance by the corresponding pixel value.

```
Matrix matrix = new Matrix();

matrix.setTranslate(20, 60);
VectorGraphicsPainter.drawString(g,
↪ "Hello MicroEJ", font, 50, matrix, 0xff, BlendMode.SRC_OVER, 0);

matrix.postTranslate(0, 60);
VectorGraphicsPainter.drawString(g,
↪ "Hello MicroEJ", font, 50, matrix, 0xff, BlendMode.SRC_OVER, 5f);

matrix.postTranslate(0, 60);
VectorGraphicsPainter.drawString(g,
↪ "Hello MicroEJ", font, 50, matrix, 0xff, BlendMode.SRC_OVER, -2);
```



Colored Emojis

The library supports the drawing of colored multilayer glyphs, but only for the embedded implementation. The simulator implementation draws the full emoji glyph with the color of the graphics context.

Only font files with CPAL/COLR tables are supported.

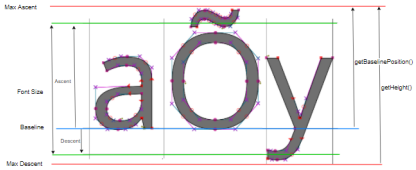
Font files with CBDT/CBLC tables are not supported.

To add colored emojis to a font, see the tutorial *How to Add Emojis to a Vector Font*.

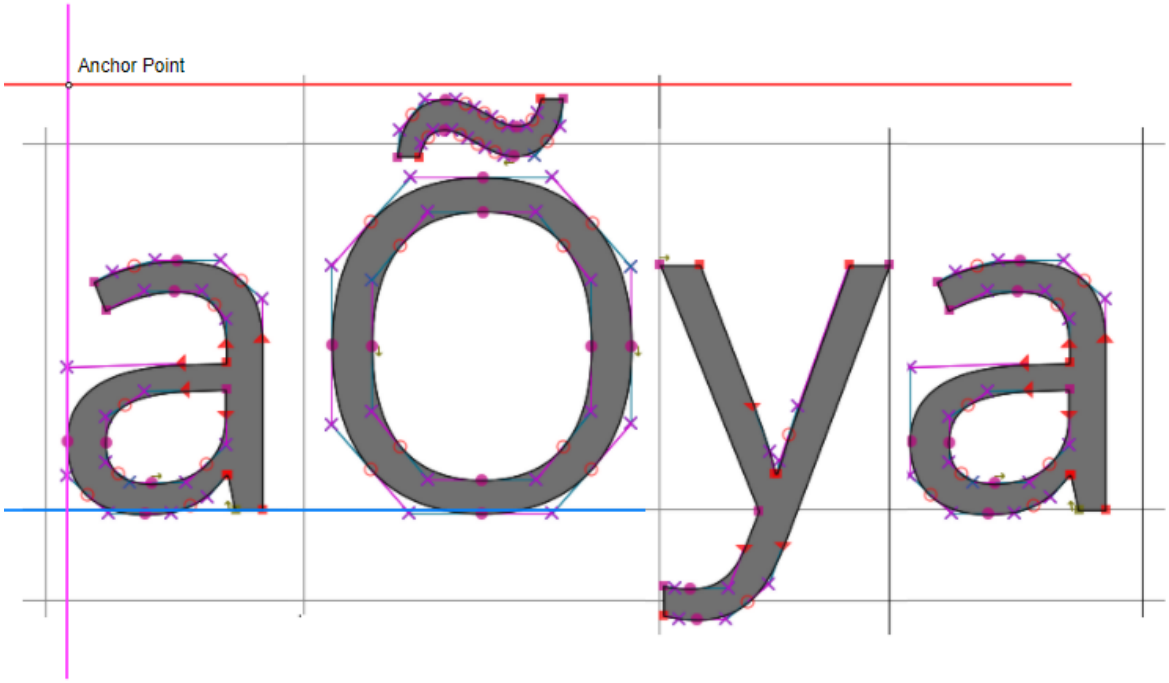
Metrics and Text Positioning

All metrics provided by the ej.microvg.VectorFont class are given for a specific font size. The font size defines the height to which each character bounding box will be scaled.

The following figure presents some concepts of font metrics standarts:



When a string is drawn with a call to ej.microvg.VectorGraphicsPainter.drawString() or ej.microvg.VectorGraphicsPainter.drawGradientString(), the anchor point of the string is the top left corner of the text rendering box. This anchor point is located horizontally on the first pixel of the first drawn glyph and vertically on the max ascent line.



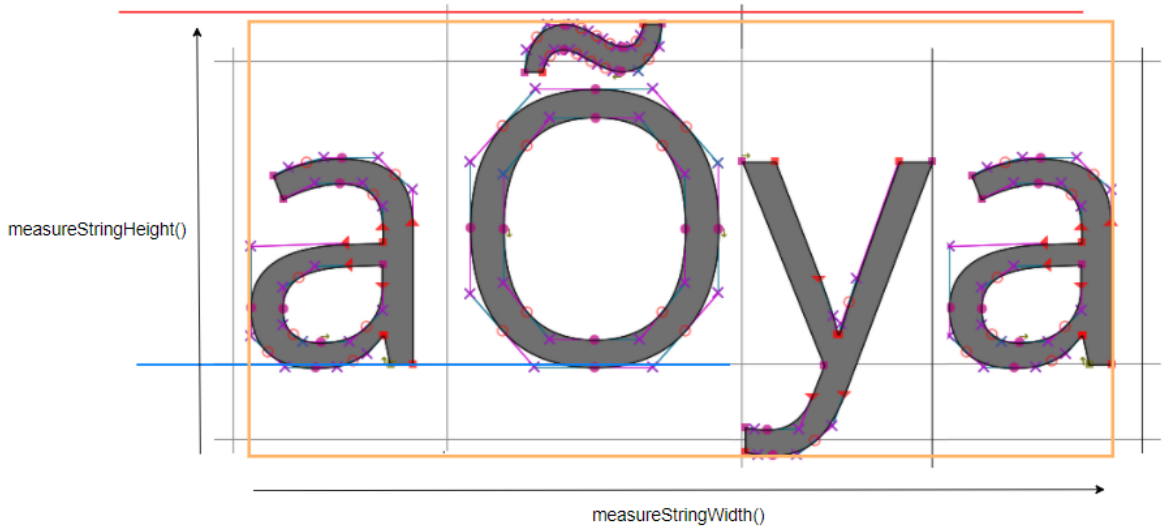
The ej.microvg.VectorFont.getBaselinePosition() method can be used to position the text baseline on a horizontal line.

The ej.microvg.VectorFont.getHeight() method can be used to center a text inside a label, by positionning the anchor point in order to have the same space above and below the text string.

Two other methods are available to position a known text in a label:

- ej.microvg.VectorFont.measureStringHeight()
- ej.microvg.VectorFont.measureStringWidth()

These methods return the width and height of a string drawing. They are computed from the width and height of the glyphs composing the string.



These methods can measure a specific glyph width and height using a one character string.

Note: The metrics are extracted from the character glyph metrics without considering the antialiasing introduced by the glyphs rasterizer.

Drawing a Text on a Circle

The library proposes the drawing of a text on a circle by a call to ej.microvg.VectorGraphicsPainter.drawStringOnCircle(). The string is rendered as if the baseline of the string was a circle arc.

The string direction can be either clockwise or counter clockwise.

All the features described above are still available (linear gradient, transformations, letter spacing, kerning, colored emojis).

```
int x = 196;
int y = 196;
int diameter = 250;

g.setColor(Colors.YELLOW);

Painter.drawCircle(g, x - diameter / 2, y - diameter / 2, diameter);

g.setColor(Colors.PURPLE);
Matrix matrix = new Matrix();

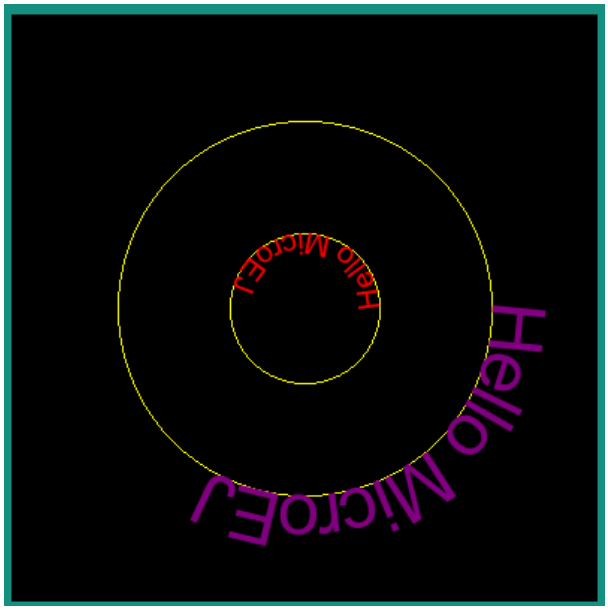
matrix.setTranslate(x, y);

VectorGraphicsPainter.
↪drawStringOnCircle(g, "Hello MicroEJ", font, 50, matrix, diameter / 2,
                Direction.CLOCKWISE);

diameter = 100;

g.setColor(Colors.YELLOW);
Painter.drawCircle(g, x - diameter / 2, y - diameter / 2, diameter);

g.setColor(Colors.RED);
VectorGraphicsPainter.
↪drawStringOnCircle(g, "Hello MicroEJ", font, 20, matrix, diameter / 2,
                Direction.COUNTER_CLOCKWISE);
```



The anchor point of the drawing is the center of the circle.

The position where the text starts along the circle is the 3 o'clock position (positive X axis). This starting position can be modified by specifying a rotation into the transformation *Matrix*.

```
g.setColor(Colors.PURPLE);
Matrix matrix = new Matrix();

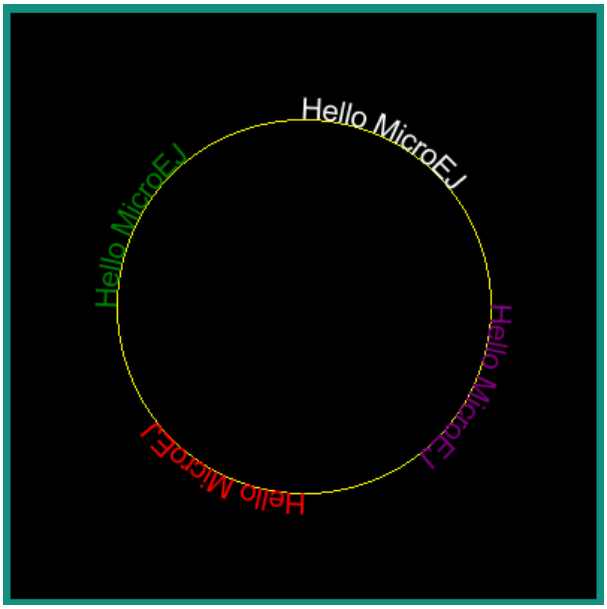
matrix.setTranslate(x, y);

VectorGraphicsPainter.
↪drawStringOnCircle(g, "Hello MicroEJ", font, 20, matrix, diameter / 2,
                Direction.CLOCKWISE);

matrix.preRotate(90);
g.setColor(Colors.RED);
VectorGraphicsPainter.
↪drawStringOnCircle(g, "Hello MicroEJ", font, 20, matrix, diameter / 2,
                Direction.CLOCKWISE);

matrix.preRotate(90);
g.setColor(Colors.GREEN);
VectorGraphicsPainter.
↪drawStringOnCircle(g, "Hello MicroEJ", font, 20, matrix, diameter / 2,
                Direction.CLOCKWISE);

matrix.preRotate(90);
g.setColor(Colors.WHITE);
VectorGraphicsPainter.
↪drawStringOnCircle(g, "Hello MicroEJ", font, 20, matrix, diameter / 2,
                Direction.CLOCKWISE);
```



Complex Text Layout

Some scripts like Arabic or Thai scripts request a specific text layout mode where the shape or positioning of a grapheme depends on its relation to other graphemes (Refer to https://en.wikipedia.org/wiki/Complex_text_layout).

The MicroVG library provides two different layout modes:

- the simple layout mode for latin scripts and other scripts where character unicodes and glyphs are one-to-one associated.
- the complex layout mode for complex text layout scripts like arabic or thai.

The simple layout mode draws the text character as described in the previous sections. It uses the font **Kerning** table and the glyphs advanceX parameter to position the glyphs one after the other.

The complex layout mode uses the **GPOS** and **GSUB** font tables to substitute and position the character glyph.

The complex layout mode can be selected while loading the glyph with *ej.microvg.VectorFont.loadFont* by passing a supplementary boolean argument with value *true*.

Next example shows the same arabic string drawn with the same font but with simple (in white) and complex layout(in RED).

```
VectorFont font0 = VectorFont.loadFont(FONT_NAME, false);
VectorFont font1 = VectorFont.loadFont(FONT_NAME, true);
```

(continues on next page)

(continued from previous page)

```
String s = "";

g.setColor(Colors.WHITE);
VectorGraphicsPainter.drawString(g, s, font0, 20, 50, 50);

g.setColor(Colors.RED);
VectorGraphicsPainter.drawString(g, s, font1, 20, 50, 100);
```



Text Measurement and Positioning

The measurement of string in complex layout mode respects the requirements presented in *Metrics and Text Positioning*.

Strings from script where text is read from right to left, like arabic, are still drawn with the anchor point located on the top left of the string.

Bidirectional Text

The complex layout mode does not support bidirectional text. A bidirectional text has to be splitted in multiple strings and each string has to be drawn to the correct location.

Limitations

The simulator rendering of complex layout mode for *Drawing a Text on a Circle* feature is done with many approximations. This rendering can still be used to have an overview of the text positionning on the display.

The letterSpacing feature is not supported by the simulator implementation. Texts will be displayed with a letterspacing value of 0.

External Fonts

To fetch fonts from external memory, the application must pre-register the *external Font resources*. The management of this kind of font may be different than the *internal* images and may require some allocations in the runtime memory. For more details about the external font management, refers to the VEE Port Guide chapter *External Memory*.

Vector Images

Overview

Vector Images are graphical resources that can be accessed with a call to ej.microvg.VectorImage.getImage(). The images are converted at build-time (using the image generator tool) to binary resources.

Images that must be processed by the image generator tool are declared in `*.vectorimages.list` files (**or** in `*.externvectorimages.list` for an external resource, see *External Images*). The file format is a standard Java properties file, each line representing a `/` separated resource path relative to the MicroEJ classpath root referring to a vector image file (e.g. `.svg` , `.xml`). The resource must be followed by a parameter (separated by a `:`) which defines and/or describes the image output file format (raw format).

Currently accepted formats are :

- `:VGF` : vglite compatible format with coordinates encoded as float numbers (32 bits).
- `:VG32` : vglite compatible format with coordinates encoded as signed int numbers (32 bits).
- `:VG16` : vglite compatible format with coordinates encoded as signed short numbers (16 bits).
- `:VG8` : vglite compatible format with coordinates encoded as signed char numbers (8 bits).

Example:

```
/com/mycompany/MyImage1.svg:VGF
/com/mycompany/androidVectorDrawable.xml:VG8
```

Supported Input Files

The image generator tool supports the following input file formats:

- Android Vector Drawable
- SVG

Refer to the *Limitations / Supported Features* section for the list of supported features for these file formats.

The vector image objects are extracted and converted to paths made of `Move`, `Line` and `Curve` commands.

Each path is associated with either a fill color or a linear gradient. All object strokes are converted to filled paths at build-time.

Objects group transformations are also extracted from the input file and applied at run-time.

Drawing Images

Drawing and Transforming Images

Once an image has been loaded it can be drawn in the graphic context with a call to ej.microvg.VectorGraphicsPainter.drawImage().

The image is associated with a transformation *Matrix* that will be applied in order to translate, scale and/or rotate the image.

The application can get the width and the height of the image with ej.microvg.VectorImage.getWidth() and ej.microvg.VectorImage.getHeight() to correctly scale and position the image in the application window.

The following example describes how an Android Vector Drawable file can be drawn and positioned on the display.

- Android Vector Drawable file:

```
<vector xmlns:android="http://schemas.
↪android.com/apk/res/android" xmlns:aapt="http://schemas.android.com/aapt"
   android:width="100dp" android:height=
↪"100dp" android:viewportWidth="100" android:viewportHeight="100">
  ␣
↪<path android:pathData="M 0 0 h50 v50 h-50 z" android:fillColor="#FFFFAA"/>
   <path android:pathData="M 50 50 h50 v50 h-50 z">
      <aapt:attr name="android:fillColor">
            <gradient
       ␣
↪       android:startColor="#0000ff" android:startX="50" android:startY="50"
   ␣
↪           android:endColor="#ff00ff" android:endX="100" android:endY="100"
             android:type="linear">
            </gradient>
      </aapt:attr>
   </path>
</vector>
```

```
public static void main(String[] args) {

   MicroUI.start();

   Display display = Display.getDisplay();
   GraphicsContext g = display.getGraphicsContext();

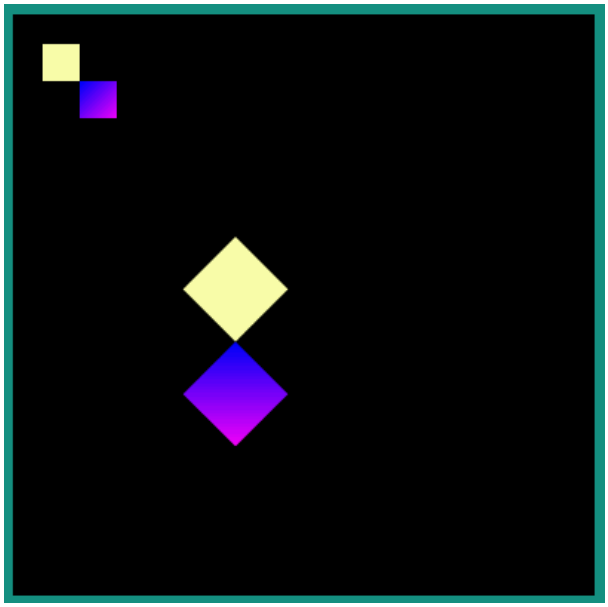
   VectorImage␣
↪image = VectorImage.getImage("/images/myImage.xml"); //$NON-NLS-1$

   Matrix matrix0 = new Matrix();
   matrix0.setTranslate(20, 20);
   matrix0.preScale(50 / image.getWidth(), 50 / image.getHeight());

   Matrix matrix1 = new Matrix();
   matrix1.setTranslate(150, 150);
   matrix1.preRotate(45);

   VectorGraphicsPainter.drawImage(g, image, matrix0);
   VectorGraphicsPainter.drawImage(g, image, matrix1);

   display.flush();
}
```



Drawing With Opacity

The vector image can be drawn with a global opacity level.

```
VectorImage␣
↪image = VectorImage.getImage("/images/myImage.xml"); //$NON-NLS-1$

// the␣
↪global opacity rendering value, between 0 (transparent) and 255 (opaque)
int opacity = 0x80;

VectorGraphicsPainter.drawImage(g, image, new Matrix(), opacity);
```



Warning: As paths are drawn one after the other, images that contain overlapping paths are not correctly colored when a global opacity is applied. The rendering of these images will throw an exception. The images must be reworked to suppress overlapping.

Color Filtering

A VectorImage object can be derived from another VectorImage object, keeping the paths and transformations but updating the colors using a color matrix.

This color matrix is a 4x5 float matrix. It is organized like that:

- Each line is used to compute a component of the resulting color, in this order: red, green, blue, alpha.
- The four first columns are multipliers applied to a component of the initial color, in this order: red, green, blue, alpha.
- The last column is a constant value.

Let A, R, G, B be the components of the initial color and the following array a color matrix:

```
{ rR, rG, rB, rA, rC,  // red
  gR, gG, gB, gA, gC,  // green
  bR, bG, bB, bA, bC,  // blue
  aR, aG, aB, aA, aC } // alpha
```

The resulting color components are computed as:

```
resultRed = rR * R + rG * G + rB * B + rA * A + rC
resultGreen = gR * R + gG * G + gB * B + gA * A + gC
resultBlue = bR * R + bG * G + bB * B + bA * A + bC
resultAlpha = aR * R + aG * G + aB * B + aA * A + aC
```

If the resulting component value is below 0 or above 255, the component value is clamped to these limits.

Note: The new image is a ResourceVectorImage. The image buffer is allocated in the MicroUI image heap. The application must manage the image cycle life and close the image to free the image buffer.

A VectorImage object can also be drawn associated to a color matrix by a call to ej.microvg.VectorGraphicsPainter.drawFilteredImage().

The following example illustrates this feature.

```
VectorImage␣
↪image = VectorImage.getImage("/images/myImage.xml"); //$NON-NLS-1$

 // Derive a new VectorImage
 float[] colorMatrix0 = new float[] { //
       1f, 0, 0, 0, 0, // red
       0, 0, 0, 0, 0, // green
       0, 0, 1f, 0, 0, // blue
       0, 0, 0, 1f, 0, // alpha
 };

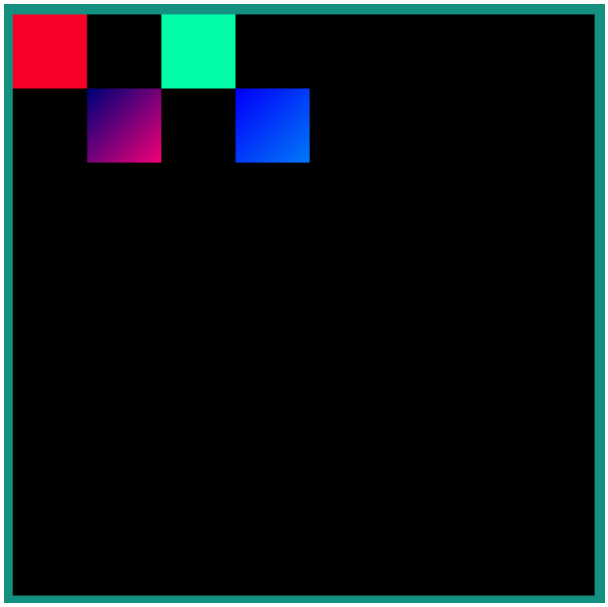
 VectorImage imageFiltered = image.filterImage(colorMatrix0);
 VectorGraphicsPainter.drawImage(g, imageFiltered, new Matrix());
```

(continues on next page)

(continued from previous page)

```
float[] colorMatrix1 = new float[] { //
      0f, 0, 0, 0, 0, // red
      0.5f, 0.5f, 0, 0, 0, // green
      0, 0, 1f, -0.5f, 0, // blue
      0, 0, 0, 1f, 0, // alpha
};
Matrix matrix1 = new Matrix();
matrix1.setTranslate(image.getWidth(), 0);

VectorGraphicsPainter.drawFilteredImage(g, image, matrix1, colorMatrix1);
```



Animated Vector Images

The Android Vector Drawable format provides the ability to change the properties of vector graphics over time, in order to create animated effects.

The transformations of the objects over the time are embedded in the Vector image file and a call to ej.microvg.VectorGraphicsPainter.drawAnimatedImage() or ej.microvg.VectorGraphicsPainter.drawFilteredAnimatedImage() will draw the image for a specific time frame.

The application can get the duration of the image animation with a call to ej.microvg.VectorImage.getDuration().

Every image object that is animated outside the image viewbox is clipped at the image boundary. In any cases, especially when the image is rotated, the image boundary is the rectangle that contains all the corners of the original image.

The supported file format is an Animated Vector Drawable xml file with animations and vector definition in the same file as described in Android API.

The SVG format also supports the animation of vector graphics objects, but this feature is not yet implemented in the MicroVG library for this file format.

SVG files that need to be animated should be converted to Android Vector Drawable format with the Android Vector Asset tool and then animated manually or with a tool like Shapeshifter.

Warning: A flaw in Eclipse Temurin™ JDK 8 causes animated vector images to render incorrectly on the Simulator. You should upgrade to Eclipse Temurin™ JDK 11 or use the JDK from Oracle instead.

Supported animations

This section will present the different available animations with an example.

For each example, this simple java code will be used.

```
VectorImage␣
↪image = VectorImage.getImage("/images/myImage.xml"); //$NON-NLS-1$
Matrix matrix = new Matrix();
matrix.setTranslate(100,100);
matrix.preScale(2,2);

long elapsed = 0;
long step = 10;
while (true) {
   // Clear Screen
   g.setColor(Colors.BLACK);
   Painter.fillRectangle(g, 0, 0, display.getWidth(), display.getHeight());

   VectorGraphicsPainter.drawAnimatedImage(g, image, matrix, elapsed);

   display.flush();

   // Pause the current thread
   try {
      Thread.sleep(step);
   } catch (InterruptedException e) {
      e.printStackTrace();
   }

   // Update current image time
   if (elapsed < image.getDuration()) {
      elapsed += step;
   } else {
      elapsed = 0;
   }
}
```

TranslateX and TranslateY

Any group in the Android Vector Drawable can be translated in X or Y direction with an object animator.

```
<animated-vector xmlns:android="http://schemas.android.com/apk/res/android"
      xmlns:aapt="http://schemas.android.com/aapt">
<aapt:attr name="android:drawable">
   <vector android:width="100dp" android:height="100dp"
           android:viewportWidth="100" android:viewportHeight="100">
      <group android:name="yellow_group">
            ␣
↪<path android:pathData="M 0 0 h50 v50 h-50 z" android:fillColor="#FFFFAA"/>
      </group>
      <group android:name="gradient_group">
      <path android:pathData="M 50 50 h50 v50 h-50 z">
            <aapt:attr name="android:fillColor">
               <gradient
          ␣
↪       android:startColor="#0000ff" android:startX="50" android:startY="50"
      ␣
↪           android:endColor="#ff00ff" android:endX="100" android:endY="100"
                android:type="linear">
               </gradient>
            </aapt:attr>
      </path>
      </group>
   </vector>
</aapt:attr>
<target android:name="yellow_group">
   <aapt:attr name="android:animation">
      <set android:ordering="together">
            <objectAnimator␣
↪android:propertyName="translateX" android:valueType="floatType"
               android:duration=
↪"1000" android:startOffset="0" android:valueFrom="0" android:valueTo="50"/>
            <objectAnimator␣
↪android:propertyName="translateX" android:valueType="floatType"
               android:duration="1000
↪" android:startOffset="1500" android:valueFrom="50" android:valueTo="0"/>
      </set>
   </aapt:attr>
</target>
<target android:name="gradient_group">
   <aapt:attr name="android:animation">
      <set android:ordering="together">
            <objectAnimator␣
↪android:propertyName="translateX" android:valueType="floatType"
               android:duration="1000
↪" android:startOffset="0" android:valueFrom="0" android:valueTo="-50"/>
            <objectAnimator␣
```

(continues on next page)

(continued from previous page)

```
↪android:propertyName="translateX" android:valueType="floatType"
                android:duration="1000
↪" android:startOffset="1500" android:valueFrom="-50" android:valueTo="0"/>
            <objectAnimator␣
↪android:propertyName="translateY" android:valueType="floatType"
                android:duration="1000
↪" android:startOffset="0" android:valueFrom="0" android:valueTo="-50"/>
            <objectAnimator␣
↪android:propertyName="translateY" android:valueType="floatType"
                android:duration="1000
↪" android:startOffset="1500" android:valueFrom="-50" android:valueTo="0"/>
      </set>
   </aapt:attr>
</target>
</animated-vector>
```

TranslateXY over a path

Any group in the Android Vector Drawable can be translated over a path.

```
<animated-vector xmlns:android="http://schemas.android.com/apk/res/android"
                 xmlns:aapt="http://schemas.android.com/aapt">
<aapt:attr name="android:drawable">
   <vector android:width="100dp" android:height="100dp"
           android:viewportWidth="100" android:viewportHeight="100">
     ... same as previous example
   </vector>
</aapt:attr>
<target android:name="gradient_group">
   <aapt:attr name="android:animation">
      <set android:ordering="together">
           <objectAnimator
                android:propertyName="translateXY" android:duration="5000"
          ␣
↪      android:propertyXName="translateX" android:propertyYName="translateY"
                android:pathData=
↪"M -0.143 0.479 C -30.355 28.02 -153.405 -111.8 -39.441 -70.818
        ␣
↪                       C -48.423 -63.52 70.593 -18.608 -91.09 -15.802 Z"/>
      </set>
   </aapt:attr>
</target>
</animated-vector>
```

ScaleX and ScaleY

A group in the Android Vector Drawable can be scaled on X or Y direction. The scaling pivot point is the one defined in the group attributes. By default, the pivot point is (0,0).

```
<animated-vector xmlns:android="http://schemas.android.com/apk/res/android"
                 xmlns:aapt="http://schemas.android.com/aapt">
<aapt:attr name="android:drawable">
   <vector android:width="100dp" android:height="100dp"
           android:viewportWidth="100" android:viewportHeight="100">
      ␣
↪<group android:name="yellow_group" android:pivotX="25" android:pivotY="25">
         ␣
↪<path android:pathData="M 0 0 h50 v50 h-50 z" android:fillColor="#FFFFAA"/>
      </group>
      <group android:name="gradient_group" >
      <path android:pathData="M 50 50 h50 v50 h-50 z">
            <aapt:attr name="android:fillColor">
               <gradient
          ␣
↪       android:startColor="#0000ff" android:startX="50" android:startY="50"
      ␣
↪           android:endColor="#ff00ff" android:endX="100" android:endY="100"
                android:type="linear">
               </gradient>
            </aapt:attr>
      </path>
      </group>
   </vector>
</aapt:attr>
<target android:name="yellow_group">
   <aapt:attr name="android:animation">
      <set android:ordering="together">
         ␣
↪<objectAnimator android:propertyName="scaleX" android:valueType="floatType"
        ␣
↪      android:duration="1000" android:startOffset="0" android:valueFrom="1"
            android:valueTo="0.5"/>
         ␣
↪<objectAnimator android:propertyName="scaleX" android:valueType="floatType"
           ␣
↪ android:duration="1000" android:startOffset="1500" android:valueFrom="0.5"
            android:valueTo="1"/>
      </set>
```

(continues on next page)

(continued from previous page)

```
   </aapt:attr>
</target>
<target android:name="gradient_group">
   <aapt:attr name="android:animation">
      <set android:ordering="together">
           ␣
↪<objectAnimator android:propertyName="scaleX" android:valueType="floatType"
              android:duration="1000" android:startOffset="0"
              android:valueFrom="0.2" android:valueTo="1"/>
           ␣
↪<objectAnimator android:propertyName="scaleX" android:valueType="floatType"
              android:duration="1000" android:startOffset="1500"
              android:valueFrom="1" android:valueTo="0.2"/>
           ␣
↪<objectAnimator android:propertyName="scaleY" android:valueType="floatType"
              android:duration="1000" android:startOffset="0"
              android:valueFrom="0.2" android:valueTo="1"/>
           ␣
↪<objectAnimator android:propertyName="scaleY" android:valueType="floatType"
              android:duration="1000" android:startOffset="1500"
              android:valueFrom="1" android:valueTo="0.2"/>
      </set>
   </aapt:attr>
</target>
</animated-vector>
```

Rotate

A group in the Android Vector Drawable can be rotated around a pivot point. The pivot point is the one defined in the group attributes. By default, the pivot point is (0,0).

```
<animated-vector xmlns:android="http://schemas.android.com/apk/res/android"
                 xmlns:aapt="http://schemas.android.com/aapt">
<aapt:attr name="android:drawable">
   <vector android:width="100dp" android:height="100dp"
           android:viewportWidth="100" android:viewportHeight="100">
      ... same as previous example
   </vector>
</aapt:attr>
<target android:name="yellow_group">
   <aapt:attr name="android:animation">
      <set android:ordering="together">
           <objectAnimator␣
```

(continues on next page)

(continued from previous page)

```
↪android:propertyName="rotation" android:valueType="floatType"
                android:duration="1000" android:startOffset="0"
                android:valueFrom="0" android:valueTo="720"/>
            <objectAnimator␣
↪android:propertyName="rotation" android:valueType="floatType"
                android:duration="1000" android:startOffset="1500"
                android:valueFrom="720" android:valueTo="0"/>
        </set>
    </aapt:attr>
</target>
</animated-vector>
```

Morphing

The Android Vector Drawable format supports the animation of the `pathData` attribute of a path. With this type of animation a shape can be transformed to a totally different other shape. The only constraint is that the origin and destination `pathData` must have the same commands format.

Lets take, for instance, the morphing of a rectangle to a circle which have the following commands.

```
Circle: M 11.9 9.8 C 11.9 8.1 13.3 6.7 14.9 6.7 C 16.6␣
↪6.7 18 8.1 18 9.8 C 18 11.6 16.6 13 14.9 13 C 13.3 13 11.9 11.6 11.9 9.8 Z

Rectangle: M 11.9 6.7 H 18 V 13 H 11.9 Z
```

The rectangle path has to be reworked to match with the sequence of commands of the circle path.

The following tools can be used to manipulate the paths to create the wanted animation effect:

- Shapeshifter
- SVGPathEditor

There is an infinity of possibilities to create the new path, and the association of each points of the paths will induce a specific morphing animation. As an example, let's define two rectangles very similar visually but with different definitions:

```
New Rectangle path1: M 11.9 9.8 C 11.
↪897 7.735 11.906 7.995 11.906 6.697 C 16.6 6.7 16.601 6.706 17.995 6.697 C␣
↪18 11.6 17.995 11.587 18.004 13.006 C 13.3 13 13.852 13.006 11.897 13.006 Z

New Rectangle path2: M 11.906 6.697 C 11.953 6.698 12.
↪993 6.698 17.995 6.697 C 17.999 8.331 17.997 9.93 18.002 13.004 C 16.239␣
↪13.007 16.009 13.001 11.893 13.007 C 13.3 13 13.852 13.006 11.893 13.007 Z
```

```
<animated-vector xmlns:android="http://schemas.android.com/apk/res/android"
                 xmlns:aapt="http://schemas.android.com/aapt">
 <aapt:attr name="android:drawable">
     <vector android:width="20dp" android:height="20dp"
              android:viewportWidth="20" android:viewportHeight="20">
      <path android:fillColor="#FF0000" android:pathData="M 0 0 h40 v40 h-40"/>
      <path android:fillColor="#FF0000" android:pathData="M 0 0 h40 v40 h-40"/>
         <group android:name="group1" android:translateX="-10">
             <path
                 android:name="circle1"
                 android:pathData="M 11.9 9.8 C 11.9 8.1 13.3 6.7 14.9 6.7
                                   C 16.6 6.7 18 8.1 18 9.8
                                   C 18 11.6 16.6 13 14.9 13
                                   C 13.3 13 11.9 11.6 11.9 9.8 Z"
                 android:fillColor="#FFFFAA"/>
         </group>
         <group android:name="group2">
             <path android:name="circle2"
                 android:pathData="M 11.9 9.8 C 11.9 8.1 13.3 6.7 14.9 6.7
                                   C 16.6 6.7 18 8.1 18 9.8
                                   C 18 11.6 16.6 13 14.9 13
                                   C 13.3 13 11.9 11.6 11.9 9.8 Z"
                 android:fillColor="#00FFAA" />
         </group>
     </vector>
 </aapt:attr>

 <target android:name="circle1">
 <aapt:attr name="android:animation">
     <set>
         <objectAnimator
             android:propertyName="pathData"
             android:duration="2000"
             android:valueFrom="M 11.9 9.8 C 11.9 8.1 13.3 6.7 14.9 6.7
                                C 16.6 6.7 18 8.1 18 9.8
                                C 18 11.6 16.6 13 14.9 13
                                C 13.3 13 11.9 11.6 11.9 9.8 Z"
      android:valueTo="M 11.9 9.8 C 11.897 7.735 11.906 7.995 11.906 6.697
                               C 16.6 6.7 16.601 6.706 17.995 6.697
                               C 18 11.6 17.995 11.587 18.004 13.006
                               C 13.3 13 13.852 13.006 11.897 13.006 Z"
             android:valueType="pathType"/>
     </set>
 </aapt:attr>
 </target>
 <target android:name="circle2">
     <aapt:attr name="android:animation">
         <set>
```

(continues on next page)

(continued from previous page)

```
            <objectAnimator
                android:propertyName="pathData"
                android:duration="2000"
                android:valueFrom="M 11.9 9.8 C 11.9 8.1 13.3 6.7 14.9 6.7
                                   C 16.6 6.7 18 8.1 18 9.8
                                   C 18 11.6 16.6 13 14.9 13
                                   C 13.3 13 11.9 11.6 11.9 9.8 Z"
↪   android:valueTo="M 11.906 6.697 C 11.953 6.698 12.993 6.698 17.995 6.697
                                  C 17.999 8.331 17.997 9.93 18.002 13.004
                                  C 16.239 13.007 16.009 13.001 11.893 13.007
                                  C 13.3 13 13.852 13.006 11.893 13.007 Z"
                android:valueType="pathType"/>
        </set>
    </aapt:attr>
</target>
</animated-vector>
```

> **Warning:** As path strokes are converted at build-time to filled path, the morphing of stroked paths is not supported. Any image with a path morphing animation on a stroked path will be rejected. Path strokes must be manually converted to filled path and the morphing of these new filled paths must be created.

Color and Opacity

Any path fillColor, strokeColor, fillAlpha and strokeAlpha attributes in the Android Vector Drawable can be animated.

```
<animated-vector xmlns:android="http://schemas.android.com/apk/res/android"
                 xmlns:aapt="http://schemas.android.com/aapt">
 <aapt:attr name="android:drawable">
    <vector android:width="55dp" android:height="55dp"
            android:viewportWidth="55" android:viewportHeight="55">
       <group android:translateX="5">
        <path android:name="fillColor" android:fillColor="#FF00FF"
              android:pathData="M 0 0 h20 v20 h-20 Z"/>
        <path android:name="fillAlpha" android:fillColor="#FF0000"
              android:pathData="M 25 0 h20 v20 h-20 Z"/>
        <path android:name="strokeColor" android:strokeWidth="5"
↪   android:strokeColor="#FFFF00" android:pathData="M 0 25 h20 v20 h-20 Z"/>
        <path android:name=
```

(continues on next page)

(continued from previous page)

```
↪"strokeAlpha" android:strokeWidth="5" android:strokeColor="#00FF00"
              android:pathData="M 25 25 h20 v20 h-20 Z"/>
       </group>
    </vector>
</aapt:attr>

<target android:name="fillColor">
<aapt:attr name="android:animation">
    <set><objectAnimator
            android:propertyName="fillColor"
            android:duration="3000"
            android:valueFrom="#FF00FF"
            android:valueTo="#FFFF00"/>
    </set>
</aapt:attr>
</target>
<target android:name="strokeColor">
    <aapt:attr name="android:animation">
        <set><objectAnimator
                android:propertyName="strokeColor"
                android:duration="3000"
                android:valueFrom="#FFFF00"
                android:valueTo="#FF00FF"/>
        </set>
    </aapt:attr>
</target>

<target android:name="fillAlpha">
    <aapt:attr name="android:animation">
        <set> <objectAnimator
                android:propertyName="fillAlpha"
                android:duration="3000"
                android:valueFrom="0.2"
                android:valueTo="1"
                android:valueType="floatType"/>
        </set>
    </aapt:attr>
</target>
<target android:name="strokeAlpha">
    <aapt:attr name="android:animation">
        <set> <objectAnimator
                android:propertyName="strokeAlpha"
                android:duration="3000"
                android:valueFrom="1"
                android:valueTo="0.2"
                android:valueType="floatType"/>
        </set>
    </aapt:attr>
</target>
</animated-vector>
```

Warning: The color of paths colored with a linear gradient can not be animated.

Easing Interpolators

Every animation is associated with an easing interpolator. By default, the animation transition is linear, but the rate of change in the animation can be defined by an interpolator. This allows the existing animation effects to be accelerated, decelerated, repeated, bounced, etc.

The supported Android interpolators are:

- accelerate_cubic
- accelerate_decelerate
- accelerate_quad
- anticipate
- anticipate_overshoot
- bounce
- cycle
- decelerate_cubic
- decelerate_quad
- decelerate_quint
- fast_out_extra_slow_in
- fast_out_linear_in
- fast_out_slow_in
- linear
- linear_out_slow_in
- overshoot

Any other vectorial path can also be used as the interpolator easing function.

Following examples show the behavior of some of the interpolators for a simple translation animation.

- Image:

```
<animated-vector xmlns:android="http://schemas.
↪android.com/apk/res/android" xmlns:aapt="http://schemas.android.com/aapt">
 <aapt:attr name="android:drawable">
     <vector android:width="100dp" android:height=
↪"100dp" android:viewportWidth="100" android:viewportHeight="100">
     <path android:pathData="M 0␣
↪0 h100 v20 h-100 Z" android:strokeColor="#FFFFFF" android:strokeWidth="1"/>
```

(continues on next page)

(continued from previous page)

```
        <group android:name="translate">
        ␣
↪<path android:pathData="M 0 0 h20 v20 h-20 Z" android:fillColor="#335566"/>
      </group>
     </vector>
 </aapt:attr>

 <target android:name="translate">
 <aapt:attr name="android:animation">
     <set><objectAnimator
             android:propertyName="translateX"
             android:duration="2000"
             android:valueFrom="0"
             android:valueTo="80"
             android:interpolator = "@android:interpolator/linear" />
     </set>
 </aapt:attr>
 </target>
</animated-vector>
```

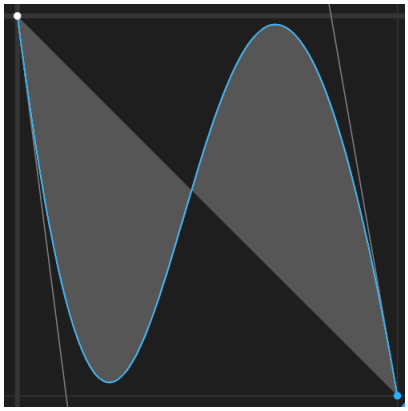
```
android:interpolator = "@android:interpolator/linear"
```

```
android:interpolator = "@android:interpolator/accelerate_cubic"
```

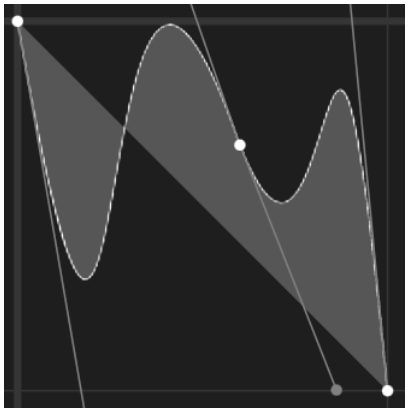
```
android:interpolator = "@android:interpolator/bounce"
```

```
android:interpolator = "@android:interpolator/fast_out_slow_in"
```

```
<aapt:attr name="android:interpolator">
   <pathInterpolator android:pathData="M 0 0 C 0.371 2.888 0.492 -1.91 1 1"/>
</aapt:attr>
```



```
<aapt:attr name="android:interpolator">
   <pathInterpolator android:pathData="M␣
↪0 0 C 0.333 1.939 0.171 -0.906 0.601 0.335 C 0.862 0.998 0.83 -0.771 1 1"/>
</aapt:attr>
```



External Images

To fetch images from external memory, the application must pre-register the *external Image resources*. The management of this kind of image may be different than the compile-time images and may require some allocations in the MicroUI *Images Heap*. For more details about the external image management, refers to the VEE Port Guide chapter *External Memory*.

Caching Generated Images

Images converted using the Image Generator can be cached so that they are not rebuilt every time the application is launched. Doing so can significantly speed up the application build phase.

See *Caching Generated Images* to have more details.

Note: The cache is available from version 13.6 of the UI Pack.

Limitations / Supported Features

Android Vector Drawable

The MicroVG library supports most of the Android Vector Drawable features with the following limitations:

- `clip-path` feature is only supported for static images.
- `trim-path` animation is not supported.
- morphing animations are not supported for paths with stroke.
- usage of path opacity is limited
  - `drawImage` with alpha is not supported if the image contains overlapping paths.
  - images with global alpha(`android:alpha` attribute of `vector` element) and overlapping paths are not supported.
  - Beware that using `android:fillColor` and `android:strokeColor` attributes on the same path leads to overlapping paths.
- `radial` and `sweep` gradient types are not supported.
- `tint` , `tintMode` and `autoMirrored` features are not supported.
- `trimPath` feature is not supported.

SVG

The MicroVG library supports a subset of SVGTiny: https://www.w3.org/TR/SVGTiny12/ including:

- Path
- Basic shape
- Painting filling
- Painting stroking
- Painting gradient (only linear gradient with one pattern)
- Painting color formats : #RRGGBB, #RGB, rgb(r,g,b), keywords
- Transforms
- Text
- Fonts (the text fonts used in the SVG file has to be installed on the operating system)

Debug Traces

MicroVG logs several actions when traces are enabled. This chapter explains the trace identifiers.

Note: The logs are only available on the Embedded VEE Port (not on the Simulator).

Trace format

The trace output format is the following:

```
[TRACE: MicroVG] Event AA(BB[CC])
```

where:

- AA is the event identifier. See next table.
- BB is the event data.
- CC is the index of the event data (0x0).

For example, given the following trace output:

```
[TRACE: MicroVG] Event 0x2(2[0x0])
```

- 0x2 -> Execute drawing event
- 2 -> Event “Draw String” (index 0x0)

Trace identifiers

The following tables describe some events data.

Table 16: MicroVG Traces

Event ID	Description	End of event
0x0 (0)	Image event `%0%` (see Image Type).	End of `%0%` (see Image Type).
0x1 (1)	Font event `%0%` (see Font Type).	End of `%0%` (see Font Type).
0x2 (2)	Drawing event `%0%` (see Drawing Type).	End of `%0%` (see Drawing Type).

Table 17: Image Type

Event ID	Description
0x0 (0)	Get or load image from RAW file
0x1 (1)	Create BufferedVectorImage
0x2 (2)	Close image

Table 18: Font Type

Event ID	Description
0x0 (0)	Load font from TTF / OTF file
0x1 (1)	Retrieve font baseline
0x2 (2)	Retrieve font height
0x3 (3)	Measure string width
0x4 (4)	Measure string height

Table 19: Drawing Type

Event ID	Description
0x0 (0)	Fill path with a color
0x1 (1)	Fill path with a linear gradient
0x2 (2)	Draw string with a color
0x3 (3)	Draw string with a linear gradient
0x4 (4)	Draw string on a circle with a color
0x5 (5)	Draw string on a circle with a gradient
0x6 (6)	Draw image

SystemView Integration

The traces are *SystemView* compatible.

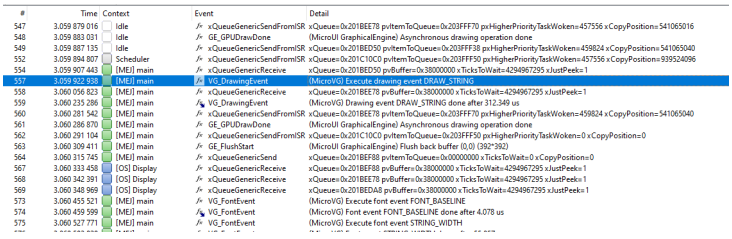


Fig. 25: MicroVG Traces displayed in SystemView

The following text can be copied in a file called `SYSVIEW_MicroVG.txt` and copied in SystemView installation folder (e.g. `SEGGER/SystemView_V252a/Description/`).

```
NamedType VGImage 0=LOAD_IMAGE
NamedType VGImage 1=CREATE_IMAGE
NamedType VGImage 2=CLOSE_IMAGE

NamedType VGFont 0=LOAD_FONT
NamedType VGFont 1=FONT_BASELINE
NamedType VGFont 2=FONT_HEIGHT
NamedType VGFont 3=STRING_WIDTH
NamedType VGFont 4=STRING_HEIGHT

NamedType VGDraw 0=DRAW_PATH
NamedType VGDraw 1=DRAW_PATH_GRADIENT
NamedType VGDraw 2=DRAW_STRING
NamedType VGDraw 3=DRAW_STRING_GRADIENT
NamedType VGDraw 4=DRAW_STRING_ON_CIRCLE
NamedType VGDraw 5=DRAW_STRING_ON_CIRCLE_GRADIENT
NamedType VGDraw 6=DRAW_IMAGE
NamedType VGDraw 7=DRAW_VGLITE_PATH
NamedType VGDraw 8=UPLOAD_VGLITE_PATH

0        VG_ImageEvent      (MicroVG)␣
↪Execute image event %VGImage  | (MicroVG) Image event %VGImage done
1        VG_FontEvent    ␣
↪  (MicroVG) Execute font event %VGFont  | (MicroVG) Font event %VGFont done
2        VG_DrawingEvent    (MicroVG)␣
↪Execute drawing event %VGDraw  | (MicroVG) Drawing event %VGDraw done
```

Android Vector Drawable Loader

Overview

The AVD Loader is an Add-On Library that can load vector images from Android Vector Drawable XML files. Unlike the vector images that are loaded using a raw output file format (see *Vector Images*), the XML parsing and interpreting is done at runtime. This is useful for loading a vector image as an external resource, especially when the resource has to be loaded dynamically (i.e., not known at build-time).

To use the AVD Loader library, add the following dependency to the project build file:

Gradle (build.gradle.kts)

MMM (module.ivy)

```
implementation("ej.library.ui:vectorimage-loader:1.1.0")
```

```
<dependency org="ej.library.ui" name="vectorimage-loader" rev="1.1.0"/>
```

Note: The AVD Loader library requires the VG Pack 1.2 and above.

Supported Format

The library supports the vector drawables with the following elements (in that order):

<vector>
Used to define a vector drawable

android:viewportWidth The width of the image (must be a positive value).

android:viewportHeight The height of the image (must be a positive value).

<path>
Defines a path.

android:fillColor (optional)
The color used to fill the path. Color is specified as a 32-bit ARGB value in hexadecimal format (#AARRGGBB). This attribute is optional when a gradient color is specified (see below).

android:fillType The fill-Type for the path, can be either evenOdd or nonZero .

android:pathData The path data, using the commands in { M , L , C , Q , Z } (match upper-case).

A linear gradient can also be used as color fill for a <path> . This element is optional if a solid color fill has been specified.

<gradient>
Used to define a linear gradient

android:endX The x-coordinate for the end of the gradient vector.

android:endY The y-coordinate for the end of the gradient vector.

android:startX The x-coordinate for the start of the gradient vector.

android:startY The y-coordinate for the start of the gradient vector.

<item> Defines an item of the gradient (minimum two items for a gradient).

android:color The color of the item. Color is specified as a 32-bit ARGB value in hexadecimal format (#AARRGGBB).

android:offset The position of the item inside the gradient (value in [0..1]).

Here is an example of a Vector Drawable myImage.xml that complies with that format. It defines a 100 x 100 image with two paths: the first one with a solid color fill, the second one with a linear gradient.

```
<vector xmlns:aapt=
↪"http://schemas.android.com/aapt" xmlns:android="http://schemas.android.
↪com/apk/res/android" android:height="100.0dp" android:viewportHeight=
↪"100.0" android:viewportWidth="100.0" android:width="100.0dp">
```

(continues on next page)

(continued from previous page)

```
   <path android:fillColor="#FFFFFFAA
↪" android:fillType="nonZero" android:pathData="M0,0L50,0L50,50L0,50Z " />
   <path android:fillType=
↪"nonZero" android:pathData="M50,50L100,50L100,100L50,100Z ">
      <aapt:attr name="android:fillColor">
         <gradient android:endX="100.0" android:endY="100.0"␣
↪android:startX="50.0" android:startY="50.0" android:type="android:linear">
            <item android:color="#FF0000FF" android:offset="0.0" />
            <item android:color="#FFFF00FF" android:offset="1.0" />
         </gradient>
      </aapt:attr>
   </path>
</vector>
```

The library only supports a subset of the Vector Drawable specification, to optimize the CPU time and memory needed for parsing and interpreting Vector Drawables in resource-constrained embedded devices. If the input Vector Drawable does not comply with this format, the library will throw an exception.

Note: The image generator tool provides a way to make a Vector Drawable compatible with the library. See *this section* for more information.

Loading a Vector Drawable

The following code loads the Vector Drawable `myImage.xml` with the `AvdImageLoader.loadImage()` method. This method has one parameter which is the path to the Vector Drawable file, provided as *a raw resource of the application*. The resulting vector image can then be drawn on the display:

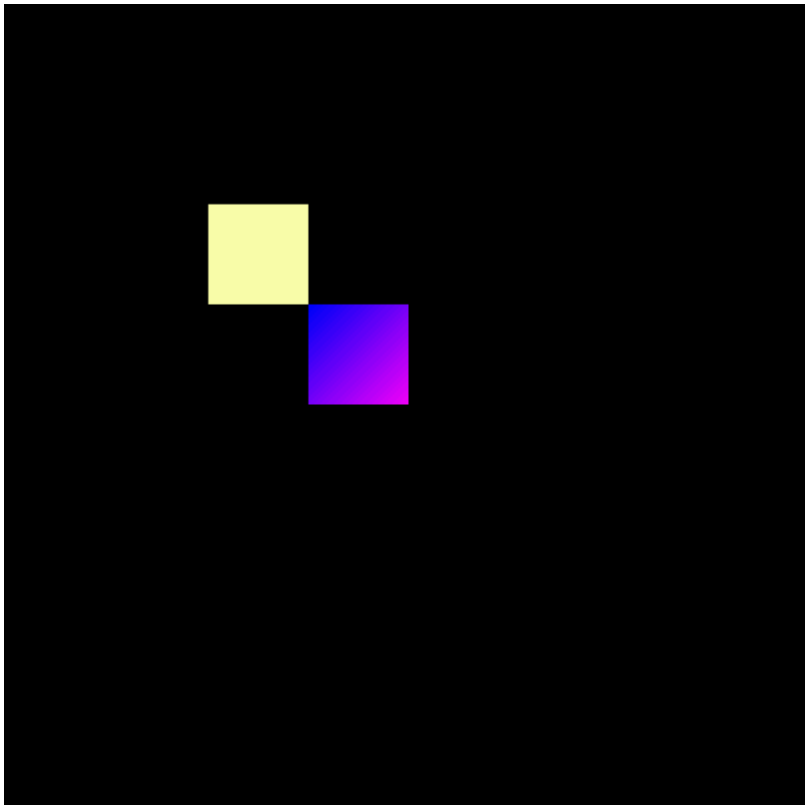
```
public static void main(String[] args) {
   MicroUI.start();

   Display display = Display.getDisplay();
   GraphicsContext g = display.getGraphicsContext();

   try (ResourceVectorImage␣
↪image = AvdImageLoader.loadImage("/images/myImage.xml")) {
      VectorGraphicsPainter.drawImage(g, image, 100, 100);
      display.requestFlush();
   }
}
```

Listing 5: Declaration of the resource in a `*.resources.list` file.

```
/images/myImage.xml
```



Note: The image must be provided as a raw resource of the application, either *internal or external*. For external resource loading, the BSP must implement the proper Abstraction Layer API (LLAPI), see *External Resources Loader* for more information on the implementation.

Warning: The new image is a `ResourceVectorImage`. In the current implementation, an image loaded with the `AvdImageLoader` is allocated in the Java heap. To release memory, the application must close the image and remove any references to it.

Limitations

The AVD Loader can only load static images (i.e., no animations). The other limitations are *the same as for vector images*.

Advanced

Make a Vector Drawable compatible with the library

To ensure that a Vector Drawable can be loaded by the AVD Loader library at runtime, the image generator tool can generate a compatible version of the drawable.

The tool comes with the VG pack installed in the platform, use the following command line to run it:

```
java -cp␣
↪[path_to_platform]/source/tools/imagegenerator-vectorimage.jar com.microej.
↪converter.vectorimage.Main --input originalImage.xml --avd myImage.xml
```

This processes the input Vector Drawable `originalImage.xml` and outputs a Vector Drawable `myImage.xml` which is compliant with the library and optimized for runtime loading.

The processing does the following:

- Normalize the output
- Limit the size of the XML file (e.g., minification)
- Pre-process the resource-consuming operations (e.g., transformations, stroking)

Convert a SVG into a compatible Vector Drawable

It is possible to convert a SVG into a compatible Vector Drawable using the platform tooling. Use the following command:

```
java -cp␣
↪[path_to_platform]/source/tools/imagegenerator-vectorimage.jar com.microej.
↪converter.vectorimage.Main --input originalImage.svg --avd myImage.xml
```

This processes the input SVG `originalImage.svg` and outputs a Vector Drawable `myImage.xml`.

Memory Usage

The loading of a Vector Drawable at runtime uses Java heap:

- for the working buffers and intermediate objects used during the loading phase. The XML parser is optimized to stream the data and uses as few heap as possible.
- for the image data.

Simplify the Path Data

The loading time and heap usage grow linearly with the number of path commands in the Vector Drawable. To achieve optimal performances, it is recommended to reduce the number of path commands, by "simplifying" the paths. The simplification algorithm will determine the optimal amount of anchor points to use in the artwork. Most of the modern Graphic Design Software have an option to simplify a path (check this article for Adobe Illustrator for example).

Monitor the Number of Path Commands

To print the number of paths and path commands declared in a Vector Drawable, set the *constant* `ej.vectorimage.loader.debug.enabled` to `true`. This will output the numbers in the console when loading a file.

Output example:

```
avdimageloader INFO: Parsed a path data with a number of 5 commands
avdimageloader INFO: Parsed a path data with a number of 5 commands
avdimageloader INFO: Parsed a path data with a number of 28 commands
avdimageloader INFO: Number of paths in loaded image: 3
```

Troubleshooting

The Image Cannot Be Parsed

A error can be raised when the parsing fails:

```
Exception in thread "main" ej.microvg.
↪VectorGraphicsException: MicroVG: The image cannot be parsed. The image␣
↪must be a valid AVD image, converted with the platform's image generator.
```

This error indicates that the file is not a compatible Vector Drawable, as specified in *this section*.

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 the project build file:

Gradle (build.gradle.kts)

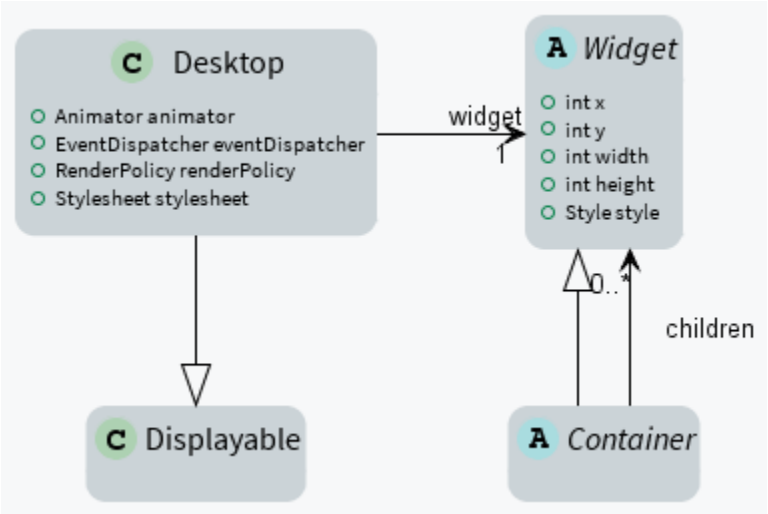
MMM (module.ivy)

```
implementation("ej.library.ui:mwt:3.3.0")
```

```
<dependency org="ej.library.ui" name="mwt" rev="3.3.0"/>
```

Concepts

Graphical Elements



Widget

A widget is an object that is intended to be displayed on a screen. A widget occupies a specific region of the display and holds a state. A user may interact with a widget (using a touch screen or a button for example).

Widgets are arranged on a desktop. A widget can be part of only one desktop hierarchy, and can appear only once on that desktop.

Container

A container follows the composite pattern: it is a widget composed of other widgets. It also defines the layout policy of its children (defining their bounds). The children's positions are relative to the position of their parent. Containers can be nested to design elaborate user interfaces.

By default, the children are rendered in the order in which they have been added in the container. And thus if the container allows overlapping, the widgets added last will be on top of the widgets added first. A container can also modify how its children are rendered.

Desktop

A desktop is a displayable intended to be shown on a display (cf. MicroUI). At any time, only one desktop can be displayed per display.

A desktop contains a widget (or a container). When the desktop is shown, its widget (and all its hierarchy for a container) is drawn on the display.

Rendering

A new rendering of a widget on the display can be requested by calling its requestRender() method. The rendering is done asynchronously in the MicroUI thread.

When a container is rendered, all its children are also rendered.

A widget can be transparent, meaning that it does not draw every pixel within its bounds. In this case, when this widget is asked to be rendered, its parent is asked to be rendered in the area of the widget (recursively if the parent is also transparent). Usually a widget is transparent when its background (from the style) is transparent.

A widget can also be rendered directly in a specific graphics context by calling its render(GraphicsContext) method. It can be useful to render a widget (and its children) in an image for example.

Render Policy

A render policy is a strategy that MWT uses in order to repaint the entire desktop or to repaint a specific widget.

The most naive render policy would be to render the whole hierarchy of the desktop whenever a widget has changed. However DefaultRenderPolicy is smarter than that: it only repaints the widget, and its ancestors if the widget is transparent. The result is correct only if there is no overlapping widget, in which case OverlapRenderPolicy should be used instead. This policy repaints the widget (or its non-transparent ancestor), then it repaints all the widgets that overlap it.

When using a *partial buffer*, these render policies can not be used because they render the entire screen in a single pass. Instead, a custom render policy which renders the screen in multiple passes has to be used. Refer to the partial buffer demo for more information on how to implement this render policy and how to use it.

The render policy can be changed by overridding Desktop.createRenderPolicy().

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.

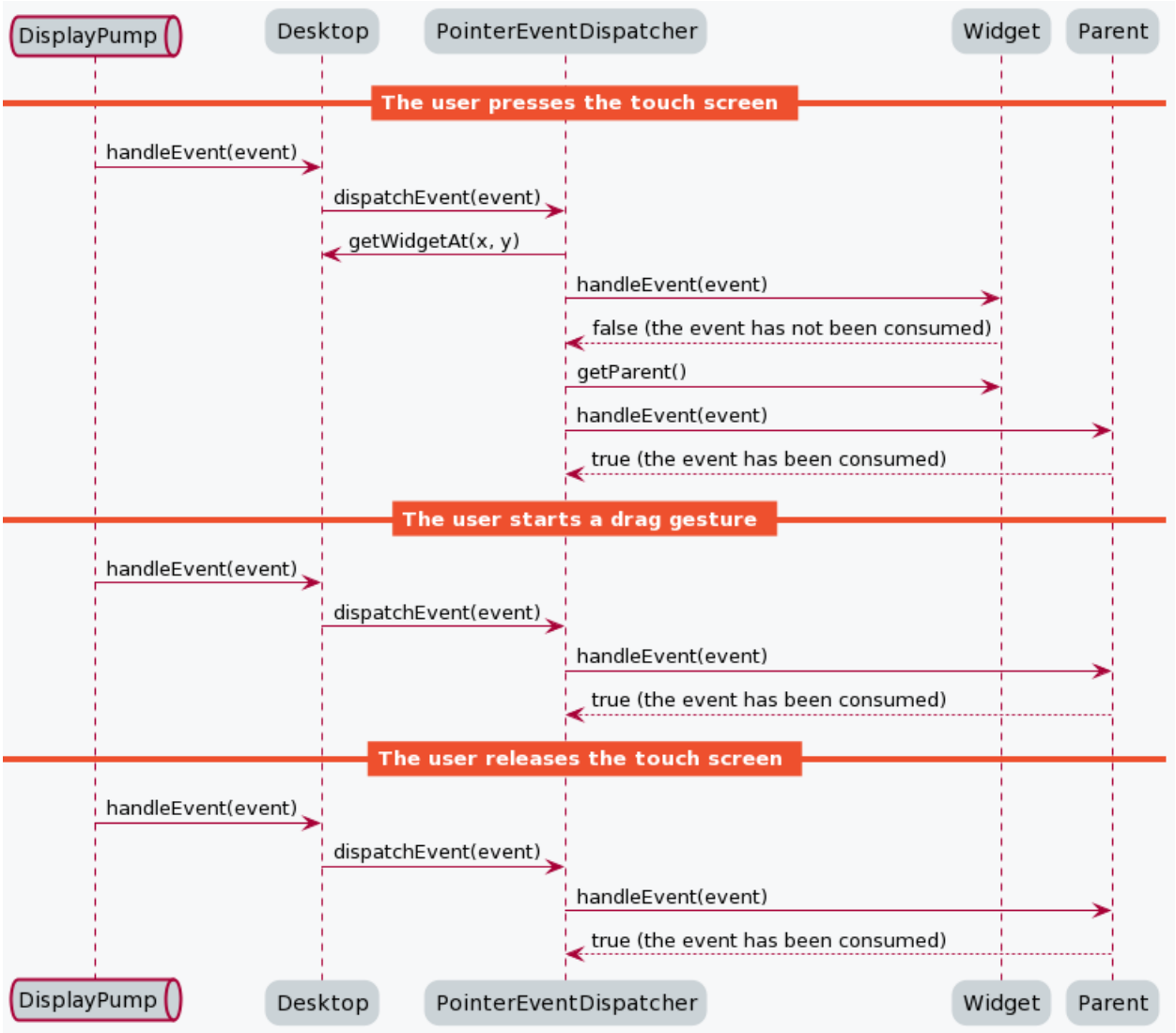
Widgets are disabled by default and don't receive the events.

Pointer Event Dispatcher

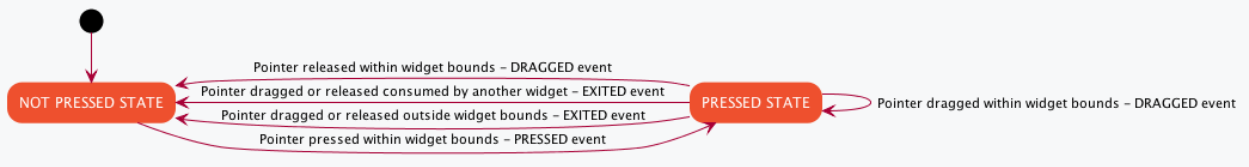
By default, the desktop proposes an event dispatcher that handles only pointer events.

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

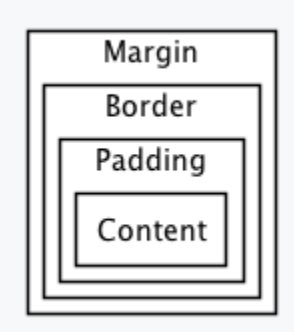
The horizontal and vertical alignments are used to position the content of the widget within its bounds.

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));

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.

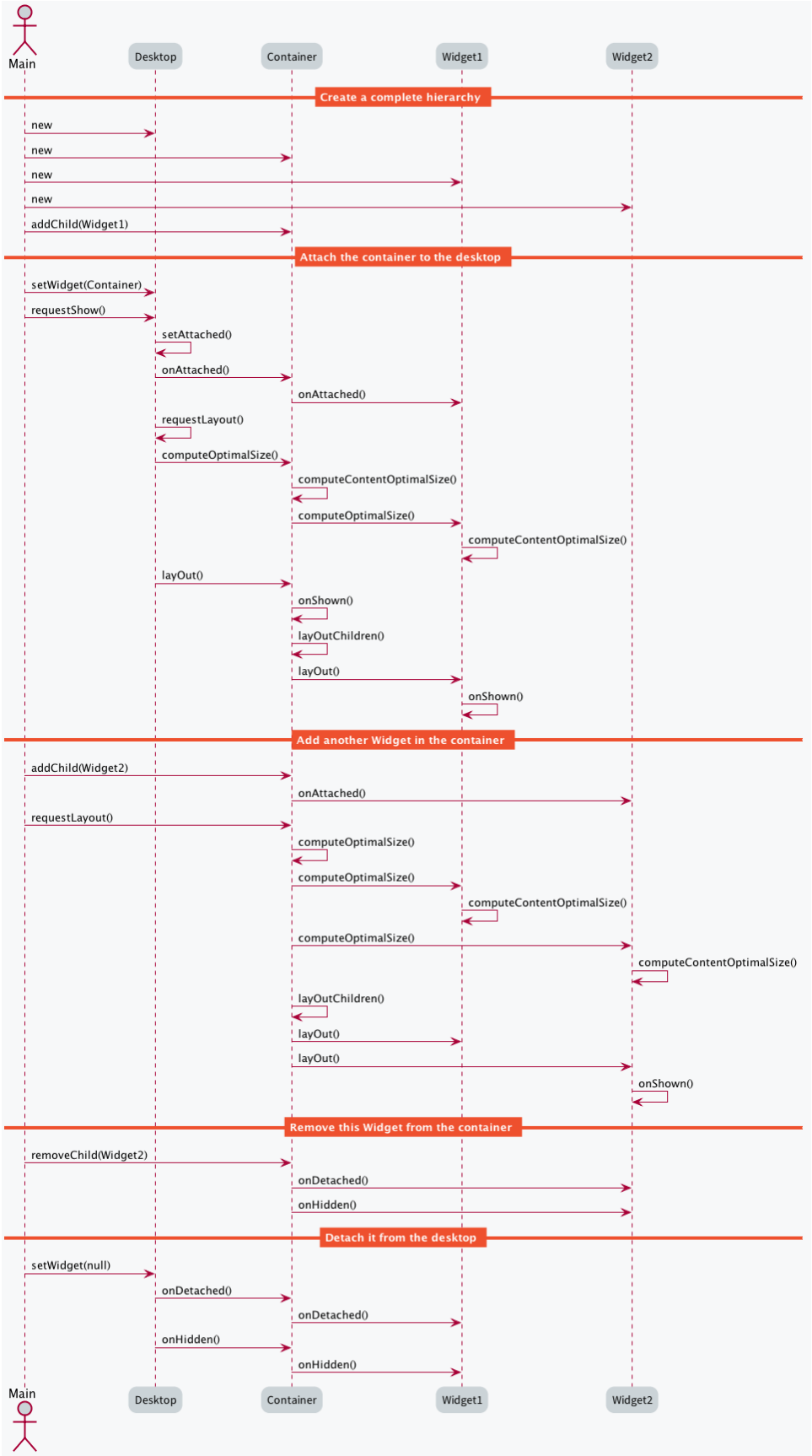
A desktop is attached automatically as soon as it is shown on the display. It can also be attached manually by calling Desktop.setAttached(). It could be used to render the desktop (and its widgets) on an image for example.

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 is the size of the smallest possible area which would still allow to represent the widget. Unless the widget is using an OptimalDimension in its style, the actual size of the widget will most likely be bigger than the optimal size returned in this method.

The `size` parameter of the computeContentOptimalSize() method initially contains the size available for the widget. An available width or height equal to Widget.NO_CONSTRAINT means that the optimal size should be computed without considering any restriction on the respective axis. Before the method returns, the size object should be set to the optimal size of the widget.

When implementing this method, the getStyle() method may be called in order to retrieve the style of the widget.

For example, the following snippet computes the optimal size of an image widget:

```
@Override
protected void computeContentOptimalSize(Size size) {
        size.setSize(this.image.getWidth(), this.image.getHeight());
}
```

Rendering the content of the widget

The renderContent() method is called by the MWT framework in order to render the content of the widget.

The `g` parameter is used to draw the content of the widget. It is already configured with the translation and clipping area which match the widget's bounds. The `contentWidth` and `contentHeight` parameters indicate the actual size of the content of the widget (excluding its outlines). Unless the widget is using an OptimalDimension in its style, the given content size will most likely be bigger than the optimal size returned in computeContentOptimalSize(). If the drawings do not take the complete content area, the position of the drawings should be computed using the horizontal and vertical alignment values set in the widget's style.

When implementing this method, the getStyle() method may be called in order to retrieve the style of the widget.

For example, the following snippet renders the content of an image widget:

```
@Override
protected void␣
↪renderContent(GraphicsContext g, int contentWidth, int contentHeight) {
        Style style = getStyle();
        int imageX = Alignment.computeLeftX(this.
↪image.getWidth(), 0, contentWidth, style.getHorizontalAlignment());
        int imageY = Alignment.computeTopY(this.
↪image.getHeight(), 0, contentHeight, style.getVerticalAlignment());
        Painter.drawImage(g, this.image, imageX, imageY);
}
```

Handling events

When a widget is created, it is disabled and it will not receive any event. A widget may be enabled or disabled by calling setEnabled(). A common practice is to enable the widget in its constructor.

Enabled widgets can handle events by overriding handleEvent(). MicroUI event APIs may be used in order to know more information on the event, such as its type. The `handleEvent()` method should return whether or not the event was consumed by the widget.

For example, the following snippet prints a message when the widget receives an event:

```
@Override
public boolean handleEvent(int event) {
        System.out.println("Event type: " + Event.getType(event));
        return false;
}
```

Consuming events

To indicate that an event was consumed by a widget, handleEvent() should return `true` . Usually, once an event is consumed, it is not dispatched to other widgets (this behavior is controlled by the event dispatcher). The widget that consumed the event is the last one to receive it.

The following guidelines are recommended to decide when to consume an event and when not to consume an event:

- If the widget triggers an action when receiving the event, it consumes the event.
- If the widget does not trigger an action when receiving the event, it does not consume the event.

Note: If the event is Pointer.PRESSED, do not consume the event unless it is required that the subsequent widgets in the hierarchy do not receive it. The Pointer.PRESSED event is special because pressing a widget is usually not the deciding factor to trigger an action. The user has to release or to drag the widget to trigger an action. If the user presses a widget and then drags the pointer (e.g. their finger or a stylus) out of the widget before releasing it, the action is not triggered.

Listening to the life-cycle hooks

Widget subclasses may override the following methods in order to allocate and free the necessary resources:

- onAttached()
- onDetached()
- onLaidOut()
- onShown()
- onHidden()

For example, the `onAttached()` method may be overridden to load an image:

```
@Override
protected void onAttached() {
        this.image = ResourceImage.loadImage(this.imagePath);
}
```

Likewise, the `onDetached()` method may be overridden to close the image:

```
@Override
protected void onDetached() {
        this.image.close();
}
```

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

```
@Override
protected void onShown() {
        Animator animator = getDesktop().getAnimator();
        animator.startAnimation(this);
}
```

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

```
@Override
protected void onHidden() {
        Animator animator = getDesktop().getAnimator();
        animator.stopAnimation(this);
}
```

How to Create a Container

To create a custom container, a new class should be created, extending the Container class. This new class may define a constructor and setter methods in order to provide a way for the user to configure the container, such as its orientation. Container subclasses have to implement two methods and may override optional methods, as explained in the following sections.

Implementing the mandatory methods

This section explains how to implement the two mandatory methods of a container subclass.

Computing the optimal size of the container

The computeContentOptimalSize() method is called by the MWT framework in order to know the optimal size of the container. The optimal size of the container should be big enough so that each child can be laid out with a size at least as big as its own optimal size.

The container is responsible for computing the optimal size of every child. To do so, the computeChildOptimalSize() method should be called for every child. After this method is called, the optimal size of the child can be retrieved by calling getWidth() and getHeight() on the child widget.

The `Size` parameter of the computeContentOptimalSize() method initially contains the size available for the container. An available width or height equal to Widget.NO_CONSTRAINT means that the optimal size should be computed without considering any restriction on the respective axis. Before the method returns, the size object should be set to the optimal size of the container.

For example, the following snippet computes the optimal size of a simple wrapper:

```
@Override
protected void computeContentOptimalSize(Size size) {
        Widget child = getChild(0);
        computeChildOptimalSize(child, size.getWidth(), size.getHeight());
        size.setSize(child.getWidth(), child.getHeight());
}
```

Laying out the children of the container

The layOutChildren() method is called by the MWT framework in order to lay out every child of the container, i.e. to set the position and size of the children. If a child is laid out outside the bounds of the container (partially or fully), only the part of the widget which is within the container bounds will be visible.

The container is responsible for laying out each child. To do so, the layOutChild() method should be called for every child. Before this method is called, the optimal size of the child can be retrieved by calling getWidth() and getHeight() on the child widget.

When laying out a child, its bounds have to be passed as parameter. The position will be interpreted as relative to the position of the container content. This means that the position should not include the outlines of the container. This means that the `(0, 0)` coordinates represent the top-left pixel of the container content and the `(contentWidth-1, contentHeight-1)` coordinates represent the bottom-right pixel of the container content.

For example, the following snippet lays out the children of a simple wrapper:

```
@Override
protected void layOutChildren(int contentWidth, int contentHeight) {
        Widget child = getChild(0);
        layOutChild(child, 0, 0, contentWidth, contentHeight);
}
```

Managing the visibility of the children of the container

By default, when a container is shown, each of its children is shown too. This behavior can be changed by overriding the setShownChildren() method of Container. When implementing this method, the setShownChild() method should be called for each child which should be shown when the container is shown.

At any time while the container is visible, children may be shown or hidden by calling setShownChild() or setHiddenChild().

When a container is hidden, each of its children is hidden too (unless it is already hidden). It is not necessary to override setHiddenChildren(), except for optimization.

Providing APIs to change the children list of the container

The Container class introduces `protected` APIs in order to manipulate the list of children of the container. These methods may be overridden in the container subclass and set as `public` in order to make these APIs available for the user.

Each of the following methods may be overridden individually:

- addChild()
- removeChild()
- removeAllChildren()
- insertChild()
- replaceChild()
- changeChildIndex()

For example, the following snippet allows the user to call the `addChild()` method on the container:

```
@Override
public void addChild(Widget child) {
        super.addChild(child);
}
```

How to Animate a Widget

Starting and stopping the animation

To animate a widget, an Animator instance is required. This instance can be retrieved from the desktop of the widget by calling Desktop.getAnimator(). Make sure that your widget subclass implements the Animation interface so that it can be used with an Animator.

An animation can be started at any moment, provided that the widget is shown. For example, the animation can start on a click event. Likewise, an animation can be stopped at any moment, for example a few seconds after the animation has started. Once the widget is hidden, its animation should always be stopped to avoid memory leaks and unnecessary operations.

To start the animation of the widget, call the startAnimation() method of the Animator instance. To stop it, call the stopAnimation() method of the same Animator instance.

For example, the following snippet starts the animation as soon as the widget is shown and stops it once the widget is hidden:

```
public class MyAnimatedWidget extends Widget implements Animation {

        private long startTime;
        private long elapsedTime;

        @Override
        protected void onShown() {
                // start animation
                getDesktop().getAnimator().startAnimation(this);
                // save start time
                this.startTime = Util.platformTimeMillis();
                // set widget initial state
                this.elapsedTime = 0;
        }

        @Override
        protected void onHidden() {
                // stop animation
                getDesktop().getAnimator().stopAnimation(this);
        }
}
```

Performing an animation step

The tick() method is called by the animator in order to update the widget. It is called in the UI thread once the display has been flushed. This method should not render the widget but should update its state and request a new render. The tick() method should return whether or not the animation should continue after this increment.

For example, the following snippet updates the state of the widget when it is ticked, requests a new render and keeps the animation going until 5 seconds have passed:

```
@Override
public boolean tick(long platformTimeMillis) {
        // update widget state
        this.elapsedTime = platformTimeMillis - this.startTime;
        // request new render
        requestRender();
        // return whether to continue or to stop the animation
        return (this.elapsedTime < 5_000);
}
```

The renderContent() method should render the widget by using its current state (saved in the fields of the widget). This method should not call methods such as Util.platformTimeMillis() because the widget could be rendered in multiple passes, for example if a *partial buffer* is used.

For example, the following snippet renders the current state of the widget by displaying the time elapsed since the start of the animation:

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

Note: Since an animator ticks its animations as often as possible, the animator may take 100% CPU usage if none of its animations requests a render. For more information on how to debug animators, see the *How to Debug Animators* section.

How to Define an Outline or Border

To create a custom outline or border, a new class should be created, extending the Outline class. Outline subclasses have to implement two methods, as explained in the following sections.

Applying the outline on an outlineable object

The apply(Outlineable) method is called by the MWT framework in order to subtract the outline from a Size or Rectangle object.

The Outlineable parameter of the method initially contains the size or bounds of the box, including the outline. Before the method returns, the outlineable object should be modified by subtracting the outline. In order to remove the outline from the object, the removeOutline() method of Outlineable should be used, passing as argument the thickness on each side.

For example, the following snippet applies an outline of 1 pixel on every side:

```
@Override
public void apply(Outlineable outlineable) {
        outlineable.removeOutline(1, 1, 1, 1);
}
```

Applying the outline on a graphics context

The apply(GraphicsContext, Size) method is called by the MWT framework in order to render the outline (only relevant if it is a border) and to update the translation and clip of a graphics context.

The `Size` parameter of the method initially contains the size of the box, including the outline. Before the method returns, the size object should be modified by subtracting the outline. In order to remove the outline from the object, the removeOutline() method of Outlineable should be used, passing as argument the thickness on each side.

For example, the following snippet applies an outline of 1 pixel on every side:

```
@Override
public void apply(GraphicsContext g, Size size) {
        size.removeOutline(1, 1, 1, 1);
        g.translate(1, 1);
        g.setClip(0, 0, size.getWidth(), size.getHeight());
}
```

How to Define a Background

To create a custom background, a new class should be created, extending the Background class. Background subclasses have to implement two methods, as explained in the following sections.

Informing whether the background is transparent

The isTransparent() method is called by the MWT framework in order to know whether or not the background is transparent. A background is considered as transparent if it does not draw every pixel with maximal opacity when it is applied.

For example, the following snippet informs that the background is completely opaque regardless of its size:

```
@Override
public boolean isTransparent(int width, int height) {
        return false;
}
```

Applying the background on a graphics context

The apply(GraphicsContext g, int width, int height) method is called by the MWT framework in order to render the background and to set or remove the background color of subsequent drawings.

For example, the following snippet applies a white background:

```
@Override
public void apply(GraphicsContext g, int width, int height) {
        g.setColor(Colors.WHITE);
        Painter.fillRectangle(g, 0, 0, width, height);
        g.setBackgroundColor(Colors.WHITE);
}
```

How to Create a Desktop Event Dispatcher

Creating a custom event dispatcher can help you address two use cases:

- [Dispatch] Extending an EventDispatcher is used to dispatch the events. For example, the FocusEventDispatcher will send the events to the widget owning the focus.
- [Handle] Overriding the desktop is used to directly trigger a behavior. For example "BACK" command shows the previous page.

To create a custom event dispatcher, a new class should be created, extending the EventDispatcher class. Event dispatcher subclasses have to implement a method and may override optional methods, as explained in the following sections.

Dispatching the events to the widgets

The dispatchEvent() method is called by the MWT framework in order to dispatch a MicroUI event to the widgets of the desktop. The getDesktop() method may be called in order to retrieve the desktop with which the event dispatcher is associated. This is useful in order to browse the widget hierarchy of the desktop, for example by using the getWidget() and getWidgetAt() methods of Desktop.

In order to send an event to one of the widgets of the hierarchy, the sendEventToWidget() method should be used. The dispatchEvent() method should return whether or not the event was dispatched and consumed by a widget.

For example, the following snippet dispatches every event to the widget of the desktop:

```
@Override
public boolean dispatchEvent(int event) {
        Widget desktopWidget = getDesktop().getWidget();
        if (desktopWidget != null) {
                return sendEventToWidget(desktopWidget, event);
        } else {
                return false;
        }
}
```

In addition to dispatching the provided events, an event dispatcher may generate custom events. This may be done by using a DesktopEventGenerator. Its buildEvent() method allows to build an event which may be sent to a widget using the sendEventToWidget() method.

Initializing and disposing the dispatcher

EventDispatcher subclasses may override the initialize() and dispose() methods in order to allocate and free the necessary resources.

For example, the initialize() method may be overridden to create an event generator and to add it to the system pool of MicroUI:

```
@Override
public void initialize() {
        this.eventGenerator = new DesktopEventGenerator();
        this.eventGenerator.addToSystemPool();
}
```

Likewise, the dispose() method may be overridden to remove the event generator from the system pool of MicroUI:

```
@Override
public void dispose() {
        this.eventGenerator.removeFromSystemPool();
}
```

How to Define an Extra Style Field

Extra style fields allow to customize a widget by configuring graphical elements of the widget from the stylesheet. Extra fields are only relevant to a specific widget type and its subtypes. A widget type can support up to `7` extra fields. The value of an extra field may be represented as an `int`, a `float` or any object, and it can not be inherited from parent widgets.

Defining an extra field ID

The recommended practice is to add a public constant for the ID of the new extra field in the widget subtype. This ID should be an integer with a value between `0` and `6`.

Every extra field ID has to be unique within the widget type. However, two unrelated widget types may define an extra field with the same ID.

For example, the following snippet defines an extra field for a secondary color:

```
public static final int SECONDARY_COLOR_FIELD = 0;
```

Setting an extra field in the stylesheet

The value of an extra field may be set in the stylesheet in a similar fashion to built-in style fields, using one of the `setExtraXXX()` methods of EditableStyle.

For example, the following snippet sets the value of an extra field for all the instances of a widget subtype:

```
EditableStyle␣
↪style = stylesheet.getSelectorStyle(new TypeSelector(MyWidget.class));
style.setExtraInt(MyWidget.SECONDARY_COLOR_FIELD, Colors.RED);
```

Getting an extra field during rendering

The value of an extra field may be retrieved from the style of a widget in a similar fashion to built-in style fields, using one of the `getExtraXXX()` methods of Style. When calling one of these methods, a default value has to be given in case the extra field is not set for this widget.

For example, the following snippet gets the value of an extra field of the widget:

```
Style style = getStyle();
int secondaryColor = style.getExtraInt(SECONDARY_COLOR_FIELD, Colors.BLACK);
```

How to Use the Overlap Render Policy

The MWT library implements two *render policies*: the DefaultRenderPolicy and the OverlapRenderPolicy:

- DefaultRenderPolicy: renders the specified widget. If the widget is transparent, it renders its parent (and recursively).
- OverlapRenderPolicy: renders the specified widget but also the other widgets that overlap with it.

While the DefaultRenderPolicy will be fine for most GUIs, it will not handle properly the case where widgets overlap. In this case, the OverlapRenderPolicy will be the best match.

Making Widgets Overlap

A widget is said to overlap with another when:

- their boundaries intersect
- it comes after in the widget tree (depth-first search)

The following snippet displays two widgets that overlap:

```
public static void main(String[] args) {
    MicroUI.start();

    Desktop desktop = new Desktop();

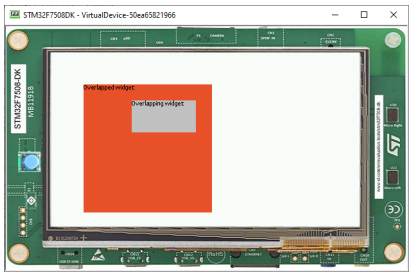
    // make two widgets overlap in a Canvas container
    Canvas rootWidget = new Canvas();
    final Button overlapped = new Button("Overlapped widget");
    rootWidget.addChild(overlapped, 50, 50, 200, 200);
    final Label overlapping = new Label("Overlapping widget");
    rootWidget.addChild(overlapping, 125, 75, 100, 50);
    desktop.setWidget(rootWidget);

    // the overlapping widget is silver
    CascadingStylesheet stylesheet = new CascadingStylesheet();
    EditableStyle␣
↪style = stylesheet.getSelectorStyle(new TypeSelector(Label.class));
    style.setBackground(new RectangularBackground(Colors.SILVER));

    // the overlapped widget is orange
    style = stylesheet.getSelectorStyle(new TypeSelector(Button.class));
    style.setBackground(new RectangularBackground(0xee502e));
    desktop.setStylesheet(stylesheet);

    desktop.requestShow();
}
```

As expected from the addChild() sequence, the widget `overlapping` overlaps the widget `overlapped`:



So far, the DefaultRenderPolicy is being used and it seems to look fine: the widgets of the desktop are rendered successively in depth-first order after the call to desktop.requestShow().

Requesting a New Render

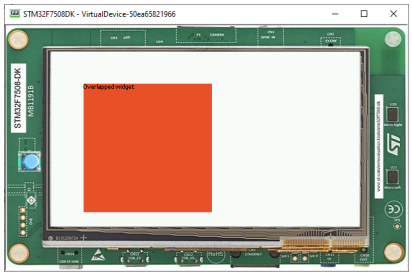
Let's see how the DefaultRenderPolicy performs when the widget `overlapped` is requested to render again. In most cases, a widget is requested to render when its content has been updated (e.g. the value displayed has changed). For demonstration purposes, let's introduce a mean to trigger a new render: each time the user clicks on the widget `overlapped`, it will request the widget to render.

The snippet above shows how to do that, by adding an OnClickListener to the `overlapped` widget:

```
overlapped.setOnClickListener(new OnClickListener() {

    @Override
    public void onClick() {
        overlapped.requestRender();
    }
});
```

When the user clicks on the widget `overlapped`, the widget is rendered again but not the widget `overlapping`. As a consequence, the widget that overlaps is not displayed anymore:



When using the DefaultRenderPolicy, widgets are rendered regardless of their order in the widget hierarchy. However, the OverlapRenderPolicy will take account of the relative order of the other widgets: widgets that come after in the widget tree will be rendered if their boundaries intersect those of the widget.

Using the OverlapRenderPolicy

Overriding the method createRenderPolicy() of the Desktop, as follows, will cause the OverlapRenderPolicy to be used when rendering widgets:

```
Desktop desktop = new Desktop() {
    @Override
    protected RenderPolicy createRenderPolicy() {
        return new OverlapRenderPolicy(this);
    }
};
```

Now, both widgets will be displayed correctly when they are requested to render.

As a conclusion, favor the OverlapRenderPolicy when a GUI uses overlapping elements. Note that this render policy is slightly more time-consuming because it traverses the widget tree to determine which widgets are overlapping with each other.

How to Debug

Highlighting the Bounds of the Widgets

When designing a UI, it can be pretty convenient to highlight the bounds of each widget. Here are some cases where it helps:

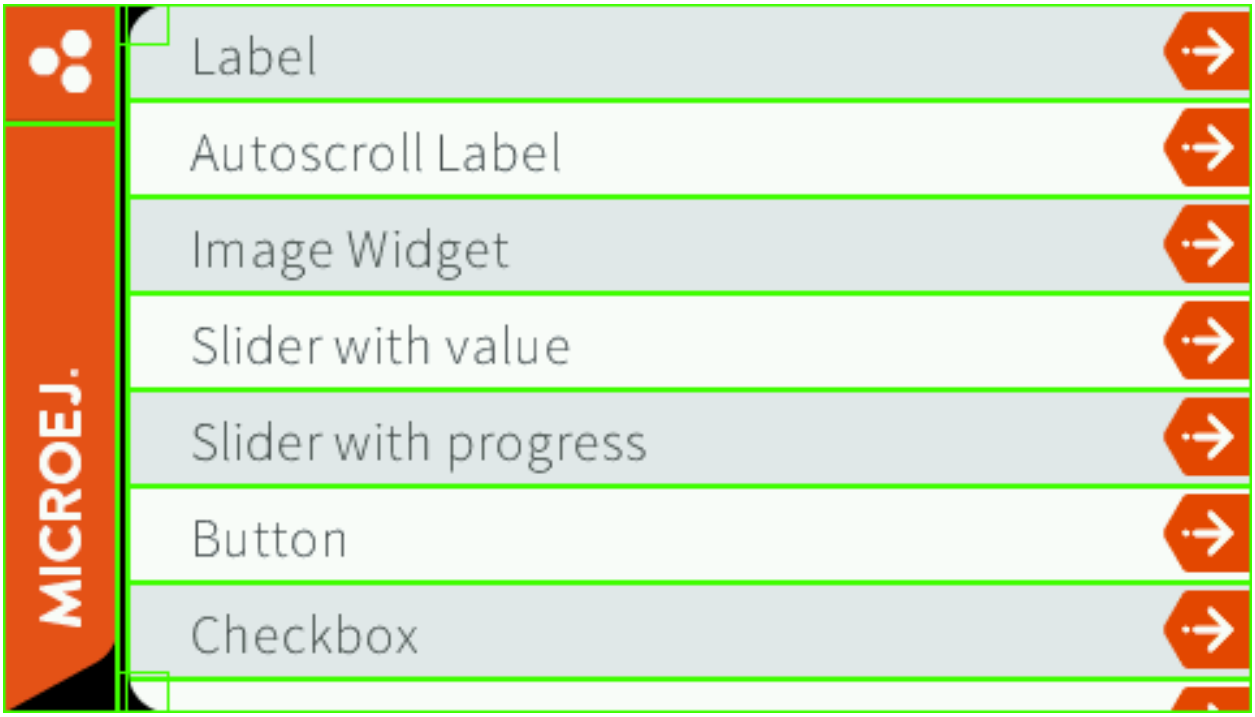
- Verify if the layout fits the expected design.
- Set the outlines (margin, padding, border).
- Check the alignment of the widget content inside its bounds.

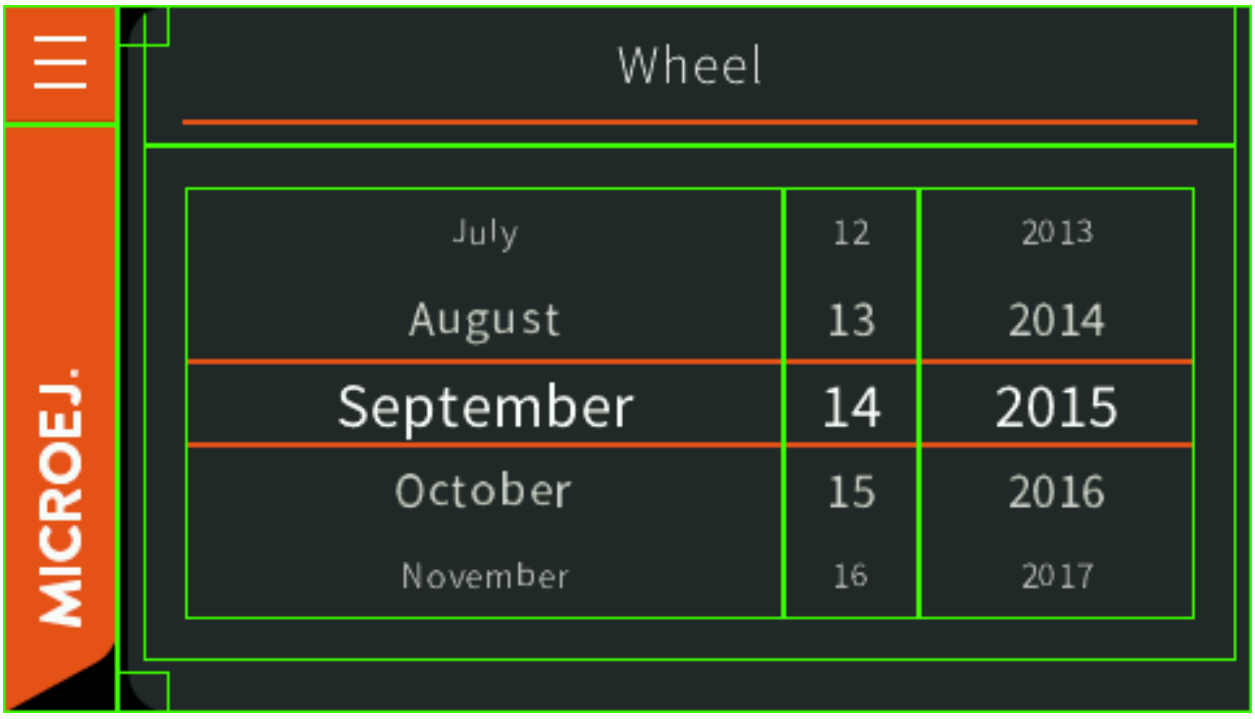
Setting the `ej.mwt.debug.bounds.enabled` constant to `true` will add a rectangle overlay over each widget and container. For more information about constants, see the *Constants* section.

By default, the rectangles around the widgets are magenta. But their color can be adjusted by modifying the `ej.mwt.debug.bounds.color` constant.

Here is an example of a `xxx.constants.list` file with the result in an application:

```
ej.mwt.debug.bounds.enabled=true
ej.mwt.debug.bounds.color=0x00ff00
```





Note: Available since MWT 3.3.0.

Monitoring the render operations

When developing a GUI application, it may not be obvious what/how exactly the UI is rendered. Especially, when a widget can be re-rendered from a distant part of the application code. Or simply because of the `RenderPolicy` used.

MWT provides a way to inject a monitor for the following render operations:

- Render requests done by the Application.
- Successive render executions triggered by the `RenderPolicy`.

Setting the `ej.mwt.debug.render.enabled` constant to `true` will enable the monitoring of above render operations. For more information about the monitoring mechanism, see RenderPolicy Javadoc. For more information about constants, see the *Constants* section.

The Widget library provides a default monitor implementation that prints the operations on the standard output. The logs produced also contain information about what is rendered (widget and area) and what code requested the rendering. For more information about this monitor implementation, see RenderMonitor Javadoc.

To use a different implementation (and if Widget is not in the classpath), set the `ej.mwt.debug.render.monitor` constant to the FQN of the monitor implementation class.

Here is an example of a `xxx.constants.list` file with the result in an application:

```
ej.mwt.debug.render.enabled=true
ej.mwt.debug.render.monitor=ej.widget.debug.RenderMonitor
```

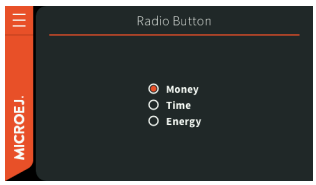


Fig. 26: Screenshot before click

Listing 6: Application logs after click

```
rendermonitor@ INFO: Render requested on com.microej.demo.widget.
↪common.PageHelper$2 > SimpleDock > OverlapContainer > SimpleDock > List␣
↪> RadioButton at {0,0 87x25} of {221,116 87x25} by com.microej.demo.widget.
↪radiobutton.widget.RadioButtonGroup.setChecked(RadioButtonGroup.java:47)
rendermonitor@ INFO: Render requested on com.microej.demo.widget.
↪common.PageHelper$2 > SimpleDock > OverlapContainer > SimpleDock > List␣
↪> RadioButton at {0,0 87x25} of {221,166 87x25} by com.microej.demo.widget.
↪radiobutton.widget.RadioButtonGroup.setChecked(RadioButtonGroup.java:50)
rendermonitor@ INFO: Render executed on  com.
↪microej.demo.widget.common.PageHelper$2 > SimpleDock > OverlapContainer␣
↪> SimpleDock > List > RadioButton at {-221,-116 87x25} of {221,116 87x25}
rendermonitor@ INFO: Render executed on  com.
↪microej.demo.widget.common.PageHelper$2 > SimpleDock > OverlapContainer␣
↪> SimpleDock > List > RadioButton at {-221,-141 87x25} of {221,141 87x25}
rendermonitor@ INFO: Render executed on  com.
↪microej.demo.widget.common.PageHelper$2 > SimpleDock > OverlapContainer␣
```

(continues on next page)

(continued from previous page)

```
↪> SimpleDock > List > RadioButton at {-221,-166 87x25} of {221,166 87x25}
rendermonitor@ INFO: Render␣
↪executed on  com.microej.demo.widget.common.PageHelper$2 > SimpleDock␣
↪> OverlapContainer > ImageWidget at {133,116 87x25} of {44,0 20x16}
rendermonitor@ INFO: Render␣
↪executed on  com.microej.demo.widget.common.PageHelper$2 > SimpleDock␣
↪> OverlapContainer > ImageWidget at {133,-140 87x25} of {44,256 20x16}
rendermonitor@ INFO: Render executed on  com.
↪microej.demo.widget.common.PageHelper$2 > SimpleDock > OverlapContainer␣
↪> SimpleDock > List > RadioButton at {-221,-116 87x25} of {221,166 87x25}
rendermonitor@ INFO: Render␣
↪executed on  com.microej.demo.widget.common.PageHelper$2 > SimpleDock␣
↪> OverlapContainer > ImageWidget at {133,166 87x25} of {44,0 20x16}
rendermonitor@ INFO: Render␣
↪executed on  com.microej.demo.widget.common.PageHelper$2 > SimpleDock␣
↪> OverlapContainer > ImageWidget at {133,-90 87x25} of {44,256 20x16}
```

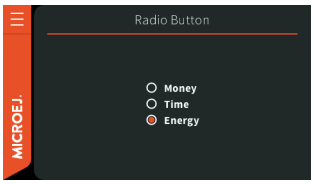


Fig. 27: Screenshot after click

Note: Available since MWT 3.5.0 & Widget 5.0.0.

Monitoring the Animators

Since an animator ticks its animations as often as possible, the animator may take 100% CPU usage if none of its animations requests a render.

MWT provides a way to inject a monitor to be notified when none of the animations has requested a render during an animator tick.

Setting the `ej.mwt.debug.animator.enabled` constant to `true` will enable animator monitoring. For more information about constants, see the *Constants* section.

The Widget library provides a default monitor implementation which logs warning messages. The logs produced also contain information about the animations running on the animator. The Animation instances are logged using their `toString()` method, so it can be a good idea to override this method in the Animation subclasses to be able to identify them.

To use a different implementation (and if Widget is not in the classpath), set the `ej.mwt.debug.animator.monitor` constant to the FQN of the monitor implementation class.

Here is an example of a `xxx.constants.list` file with the result in an application:

```
ej.mwt.debug.animator.enabled=true
```

Listing 7: Application logs when the watchface update animation is started but it doesn't request a render

```
animatormonitor WARNING: No render requested␣
↪during animator tick. Animations list: [Watchface update animation]
```

Note: Available since MWT 3.5.0 & Widget 5.0.0.

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 the project build file:

Gradle (build.gradle.kts)

MMM (module.ivy)

```
implementation("ej.library.ui:widget:4.1.0")
```

```
<dependency org="ej.library.ui" name="widget" rev="4.1.0"/>
```

To fork one of the provided widgets, duplicate the associated Java class from the widget library JAR into the source code of your application. It is recommended to move the duplicated class to an other package and to rename the class in order to avoid confusion between your forked class and the original class.

Provided Widgets

Widgets:

- Label: displays a text.
- ImageWidget: displays an image which is loaded from a resource.
- Button: displays a text and reacts to click events.
- ImageButton: displays an image which is loaded from a resource and reacts to click events.

Containers:

- List: lays out any number of children horizontally or vertically.

- Flow: lays out any number of children horizontally or vertically, using multiple rows if necessary.
- Grid: lays out any number of children in a grid.
- Dock: lays out any number of children by docking each child one by one on a side.
- SimpleDock: lays out three children horizontally or vertically.
- OverlapContainer: lays out any number of children by stacking them.
- Canvas: lays out any number of children freely.

Color Utilities

The widget library offers some color utilities.

The ColorHelper is helpful for decomposing colors into components (alpha, red, blue, green) and building back a color from components.

The GradientHelper can blend two colors and create a gradient from two colors.

The resulting gradient is a list of distinct colors from the start color to the end color. The colors are truncated to the display color depth. As a consequence, for the same start and end colors, a gradient created for a 32-bit display will contain more colors than on a 16-bit display.

The LightHelper proposes several primitives concerning the luminance of the colors. The luminance of a color is computed from the luminance and the quantity of each of its components. The green being the brighter, then the red and finally the blue.

Debug Utilities

A few utilities useful for debugging are available in the package `ej.widget.util.debug` of the widget library.

Print the Hierarchy of Widgets

The method HierarchyInspector.hierarchyToString(Widget) returns a String representing the hierarchy of a widget. In other words, it prints the widget and its children recursively in a tree format.

It may be used to analyse the content of a page and have a quick estimation of the number of widgets and the depth of the hierarchy.

For example:

```
Scroll
+--ScrollableList
|  +--Label
|  +--Dock
|  |  +--ImageWidget
|  |  +--Label
|  +--Label
```

Print the Path to a Widget

The method HierarchyInspector.pathToWidgetToString(Widget) returns a String representing the list of ancestors of the widget. For example: `Desktop > Scroll > ScrollableList > Label`.

It may be used to identify a widget in a trace.

It is also possible to choose the separator by using HierarchyInspector.pathToWidgetToString(Widget, char) method. For example: `Desktop ; Scroll ; ScrollableList ; Label`.

Count the Number of Widgets or Containers

The methods HierarchyInspector.countNumberOfWidgets(Widget) and HierarchyInspector.countNumberOfContainers(Widget) respectively count the number of widgets and containers in a hierarchy.

It may be used to evaluate the complexity of a hierarchy of widgets.

Count the Maximum Depth of a Hierarchy

The method HierarchyInspector.countMaxDepth(Widget) counts the maximum depth of a hierarchy. In other words, the depth of the widget with the biggest number of parents recursively.

It may be used to evaluate the complexity of a hierarchy of widgets.

Print the Bounds of a Widget

The method BoundsInspector.boundsToString(Widget) returns a String with the widget type and its bounds. The returned String contains:

- the simple name of the class of the widget,
- its position relative to its parent,
- its size,
- its absolute position.

  For example: `Label: 0,0 7x25 (absolute: 75,75)`

Print the bounds of all the widgets in a hierarchy

The method BoundsInspector.boundsRecursiveToString(Widget) returns a String representing the type and bounds of each widget in the hierarchy of a widget.

For example:

```
Scroll: 0,0 480x272 (absolute: 0,0)
+--ScrollableList: 0,0 480x272 (absolute: 0,0)
|  +--Label: 0,0 480x50 (absolute: 0,0)
|  +--Dock: 0,50 480x50 (absolute: 0,50)
|  |  +--ImageWidget: 0,0 70x50 (absolute: 0,50)
```

(continues on next page)

(continued from previous page)

```
|  |  +--Label: 70,0 202x50 (absolute: 70,50)
|  +--Label: 0,100 480x50 (absolute: 0,100)
```

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

The showcased widgets are listed in the README of the project.

MWT Examples

The MWT Examples are code samples that show how to implement various use cases with MWT.

Because the MWT toolkit is designed to be compact and customizable, it allows for many possibilities when developing a GUI. Thus, the examples can be used, with or without modifications, to extend and customize the MWT framework for your specific needs. They also help to learn the best practices for the development of graphic interfaces with MWT.

Source

To get the source code of these examples, clone the following GitHub repository: https://github.com/MicroEJ/ExampleJava-MWT.

The repository contains several projects (one project for each example). You can import the projects in MicroEJ SDK to browse the source code and run the examples.

- Go to `File` > `Import...`.
- In `General`, select `Existing Projects into Workspace`.
- Check `Select root directory` and browse to the cloned MWT Examples repository.
- Select the projects to import and click `Finish`.

  For each project, please refer to its `README.md` file for more details about the example and its usage.

Run the Examples

Make sure to have a valid VEE port in the workspace. The examples can be tested with the Platform STM32F7508-DK.

To run the examples in the simulator:

- Right-click on a project.
- Select Run As > MicroEJ Application .

Note: Some of the examples may require extra configuration, please refer to their README for more details.

To run the examples on a device, *adapt the Run Configuration*, then refer to the VEE port documentation.

Provided Examples

Attribute Selectors

This example shows how to customize the style of widgets using attribute selectors, similar to CSS Attribute Selectors.

It provides several types of attribute selectors, any of which can be used in a stylesheet to select widgets based on custom attributes. In this case, the background color of a label switches depending on the value of an attribute of the label.

Buffered Image Pool

This example shows how to use a pool of BufferedImages to share them across an application.

In this demo, there is one image in the pool, which is shared between the histogram widget and the transition container.

Context-Sensitive Container

This example shows a smartwatch application that looks different depending on whether the user is wearing the device on the left arm or on the right arm.

It demonstrates how a container can adapt to the context by changing how its children are laid out: in this case, depending on the wrist mode, the widgets are displayed on either the left or right side. For demonstration purposes, the example displays a virtual watch to simulate the device flip.

Drag'n'Drop

This example shows how to implement drag'n'drop support in a grid.

Focus

This example shows how to introduce focus management in a project when using peripherals like buttons or a joystick.

The virtual joystick (on the right) is used to simulate a hardware joystick. When the joystick directions (up, down, left, right) are pressed, the focus changes on the items in the same way as when using the touch pointer.

Immutable Stylesheet

This example shows how to create and use an immutable stylesheet.

The immutable stylesheet resolves the style for a widget with the same algorithm as the cascading stylesheet. The difference is that the immutable stylesheet is described in an immutable

file instead of Java code. Therefore, the style objects are allocated in flash instead of the Java heap.

Lazy Stylesheet

This example shows how to create and use a “lazy” stylesheet.

The lazy stylesheet resolves the style for a widget with the same algorithm as the cascading stylesheet. The difference is that the lazy stylesheet associates style factories with selectors (rather than style instances). As a result, the style elements are allocated “on demand” when a rule’s selector applies to a widget.

Masking Grid

This example shows how to mask a widget temporarily.

The grid is a custom container (`MaskingGrid`) that exposes an API to change the visibility of its children (visible or invisible). When requested to render, the grid only renders the children marked as visible.

MVC

This example shows how to create and use an MVC design pattern (Model, View, Controller).

The value of the model can be changed by clicking on the physical button.

It is also possible to resize all the widgets at once.

Popup

This example shows how to show a popup in an application.

Two types of popups are illustrated. The information popup can be dismissed by clicking outside of its bounds. The action popup needs the user to click on a button to close it.

Remove Widget

This example shows how to add and remove widgets in a widget hierarchy.

The layout adapts automatically to the number of items because `requestLayout()` is called for each addition/deletion on the container.

Slide Container

This example shows a slide container. This is a container that shows only its last child.

An animation is done when adding/removing a child by translating the widgets from/to the right.

Stack Container

This example shows a stack container. This is a container that stacks its children on top of each other.

An animation is done when adding/removing a child by translating the widget from/to the right.

Stashing Grid

This example shows how to stash a widget temporarily.

The grid is a custom container (`StashingGrid`) that exposes an API to change the visibility of its children (visible or invisible). When requested to lay out, the grid only lays out the children marked as visible. When requested to render, the grid only renders the children marked as visible.

Theming and Branding

This example shows how to create theming and branding for your project.

The application contains only one page.

There are two different types of theming shown:

1. Changing from normal to condensed by passing a *Theme* when building the stylesheet.
2. Changing the styling (including padding, margin, background, etc.) itself with a *StyleTheme*.

Transition

This example shows a container that triggers effects during page transitions.

The effect applied to the transition container can be changed dynamically. New effects can be developed easily.

Virtual Watch

This example shows how to simulate the skin and inputs of a device with a different device (e.g., an evaluation board).

This can be a convenient option when the target hardware is not yet available.

Here, it simulates a watch with a round screen and 3 buttons. The actual application is shown in a round area of the screen and receives events from the virtual buttons. The virtual buttons send commands when clicked, the same way a target device would have sent events from the native world. The goal is to be able to migrate the application on the target device without modifying the application code.

Advanced

How to Detect Text Overflow

Widgets that display a text may experience text overflow when the strings are too long to fit into the available area. It can be the case, for example, in applications that support multiple languages because widgets have to deal with texts of different lengths.

Instrumenting the Widget

The goal is to check whether the text to be displayed is within the content bounds of the widget. A way to test this is to extend or modify the widget. In this article, the widget `MyLabel` will extend the type Label from the Widget library, which displays a text:

```
import ej.widget.basic.Label;

public class MyLabel extends Label {

    public MyLabel(String text) {
        super(text);
    }
}
```

Overriding the onLaidOut() Method

Once the position and size of a wigdet are set during the lay out process, the onLaidOut() method is called to notify the widget. Overriding onLaidOut() of class `MyLabel` is a good place to check whether the text overflows or not.

For example, the following snippet compares the text width with the available width: it will print a message if an overflow is detected.

```
@Override
protected void onLaidOut() {
    super.onLaidOut();

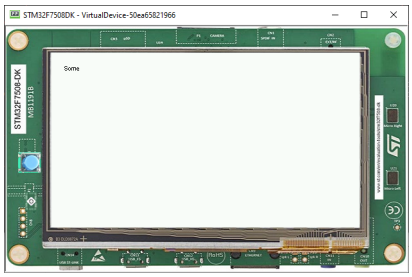
    // compute the width of the text with the specified font
    final Font font = getStyle().getFont();
    final String text = getText();
    final int textWidth = font.stringWidth(text);

    // compare to the width available for the content of the widget
    final int contentWidth = getContentBounds().getWidth();
    if (textWidth > contentWidth) {
        System.out.println("Overflow detected:\n > Text: \"" + text + "\
↪"\n > Width = " + textWidth + " px (available: " + contentWidth + " px)");
    }
}
```

Testing

Here is a case where the widget size is set manually to be a little shorter than the text width:

```
public static void main(String[] args) {
    MicroUI.start();
    Desktop desktop = new Desktop();
    Canvas canvas = new Canvas();
    //␣
↪add a label with an arbitrary fixed width of 25 pixels (which is too short)
    canvas.addChild(new MyLabel("Some text"), 20, 20, 25, 10);
    desktop.setWidget(canvas);
    desktop.requestShow();
}
```



The text is cropped and the console logs that a text overflow has been detected:

```
============== [ Initialization Stage ] ==============
============== [ Converting fonts ] ===============
============== [ Converting images ] ===============
============== [ Launching on Simulator ] ===============
Overflow detected:
 > Text: "Some text"
 > Width = 47 px (available: 25 px)
```

Improving the Detection

To ease the correction process, it is best to add some additional debug information to locate the issue. Let's extract the text overflow detection into a helper class, so that it is available for all classes across the application.

The following snippet:

- extracts the text overflow detection into the class `MyTextHelper`.
- prints the part of the text that is displayed.
- prints the path to the widget in the widget tree to help the tester locate the affected widget in the GUI.

```
public class MyLabel extends Label {

    public MyLabel(String text) {
        super(text);
    }

    @Override
    protected void onLaidOut() {
        super.onLaidOut();

        final Font font = getStyle().getFont();
        final String text = getText();
        MyTextHelper.checkTextOverflow(this, text, font);
    }
}

public class MyTextHelper {

    /**
    * Checks whether the given text overflows␣
↪for the specified widget and font. In the case where an overflow is
    * detected, the method prints a message that details the error.
    *
    * @param widget
    *            the widget that displays the text.
    * @param text
    *            the text to display.
    * @param font
    *            the font used for drawing the text.
    */
```

(continues on next page)

(continued from previous page)

```
    public static void checkTextOverflow(final␣
↪Widget widget, final String text, final Font font) {
        final int textWidth = font.stringWidth(text);
        final int contentWidth = widget.getContentBounds().getWidth();

        if (textWidth > contentWidth) {
    ␣
↪       String displayedText = buildDisplayedText(text, font, contentWidth);
            String widgetPath = buildWidgetPath(widget);
            System.out.println(
                    "Overflow detected:\
↪n > Text: \"" + text + "\"\n > Width = " + textWidth + " px (available: "
                            + contentWidth +␣
↪" px) \n > Displayed: \"" + displayedText + "\"\n > Path : " + widgetPath);
        }
    }

    private␣
↪static String buildDisplayedText(String text, Font font, int width) {
        for (int i = text.length() - 1; i > 0; i--) {
            if (font.substringWidth(text, 0, i) <= width) {
                return text.substring(0, i);
            }
        }

        return "";
    }

    private static String buildWidgetPath(Widget widget) {
        StringBuilder builder = new StringBuilder();

        Widget ancestor = widget;
        do {
            builder.insert(0, " > " + ancestor.getClass().getSimpleName());
            ancestor = ancestor.getParent();
        } while (ancestor != null);
        builder.insert(0, widget.getDesktop().getClass().getSimpleName());

        return builder.toString();
    }
}
```

When the application is launched again, the console shows more information about the text overflow:

```
=============== [ Initialization Stage ] ===============
=============== [ Converting fonts ] ===============
=============== [ Converting images ] ===============
=============== [ Launching on Simulator ] ===============
Overflow detected:
 > Text: "Some text"
 > Width = 47 px (available: 25 px)
```

(continues on next page)

(continued from previous page)

```
 > Displayed: "Some"
 > Path : Desktop > Canvas > MyLabel
```

To keep control over the extra verbosity and code size, one option is to use *BON constants* to enable/disable this debug code at will. In the following snippet, when the constant `com.mycompany.check.text.overflow` is set to `false`, the debug code will not be embedded in the application.

```
public static void checkTextOverflow(final␣
↪Widget widget, final String text, final Font font) {
    if (Constants.getBoolean("com.mycompany.check.text.overflow")) {
        final int textWidth = font.stringWidth(text);
        final int contentWidth = widget.getContentBounds().getWidth();

        if (textWidth > contentWidth) {
    ␣
↪       String displayedText = buildDisplayedText(text, font, contentWidth);
           String widgetPath = buildWidgetPath(widget);
           System.out.println(
                   "Overflow detected:\
↪n > Text: \"" + text + "\"\n > Width = " + textWidth + " px (available: "
                           + contentWidth +␣
↪" px) \n > Displayed: \"" + displayedText + "\"\n > Path : " + widgetPath);
        }
    }
}
```

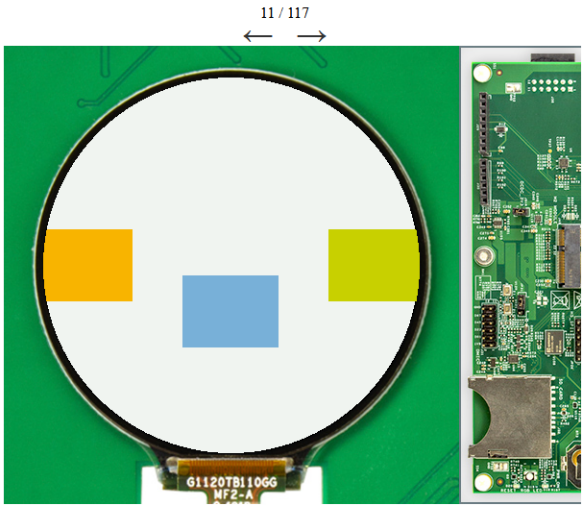
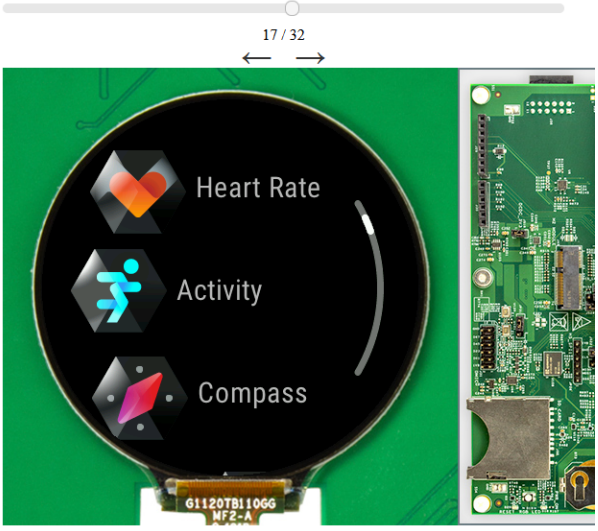
MicroUI Flush Visualizer

Building smooth and visually appealing UI applications requires a keen focus on performance. To achieve efficient UI rendering, minimizing unnecessary work that consumes valuable CPU time is essential.

This article introduces the MicroUI Flush Visualizer, a tool designed to investigate potential performance bottlenecks in UI applications running on the Simulator by showing the pixel surface drawn between two MicroUI frame buffer flushes.

- Requirements: VEE Port with UI Pack 13.0.0 or higher.

Here are examples of the MicroUI Flush Visualizer in action:

Simple GFX	Demo Wearable
	

`fillCircle x=0 y=0 diameter=390 area=119459.06065275188`
`fillRectangle x1=-7 y1=158 x2=92 y2=232 area=7326`
`fillRectangle x1=145 y1=206 x2=244 y2=280 area=7326`
`fillRectangle x1=297 y1=158 x2=396 y2=232 area=7326`
`11# Area drawn: 117%`
	`fillRectangle x1=0 y1=0 x2=391 y2=391 area=152881`
	`drawImage regionX=0 regionY=0 width=100 height=87 x=50 y=53 alpha=255 area=8700`
	`drawImage regionX=0 regionY=0 width=131 height=35 x=160 y=74 alpha=255 area=4585`
	`drawImage regionX=0 regionY=0 width=100 height=87 x=30 y=156 alpha=255 area=8700`
	`drawImage regionX=0 regionY=0 width=90 height=35 x=140 y=181 alpha=255 area=3150`
	`drawImage regionX=0 regionY=0 width=100 height=87 x=52 y=268 alpha=255 area=8700`
	`drawImage regionX=0 regionY=0 width=115 height=35 x=162 y=288 alpha=255 area=4025`
	`drawImage regionX=0 regionY=0 width=100 height=4 x=166 y=388 alpha=255 area=400`
	`17# Area drawn: 131%`

Understanding the Rendering Pipeline

One key element to successful smooth UI applications is performance. An efficient UI application must only draw what is necessary to achieve the visual effects desired. For example, assuming the application targets 60 FPS to perform a transition between two screens, that means the application has 1/60s ~= 16ms in total to execute the *Rendering Pipeline*.

The *Rendering Pipeline* of a UI application consists of three main phases: Layout, Rendering, and Flush.

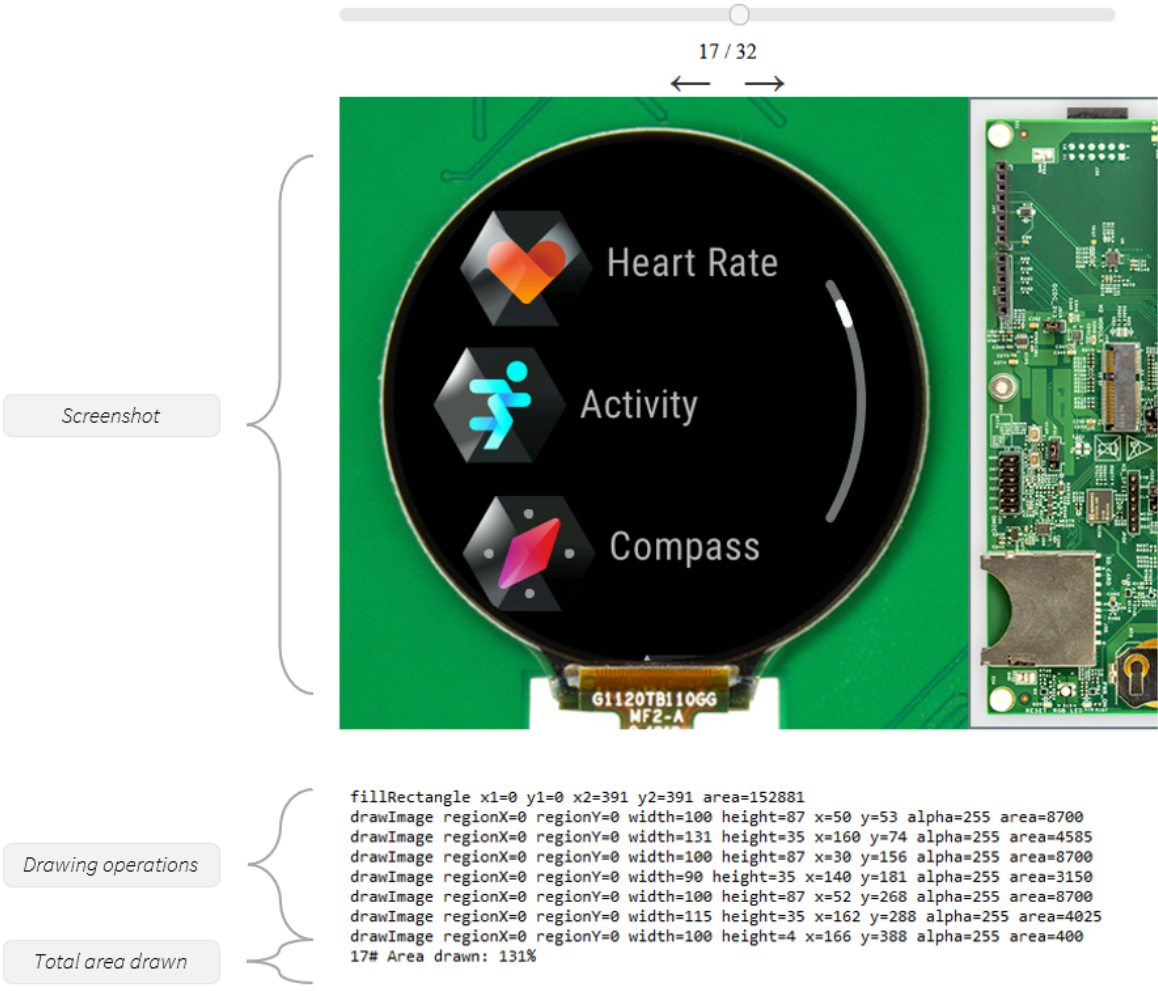
1. *Layout*: This phase determines which widgets should be displayed on the screen and the positions of the widgets. It is typically triggered when widgets are added or removed from the widget hierarchy. An application should only modify the widget hierarchy when necessary and avoid doing so during animation to ensure efficiency.
2. *Rendering*: During this phase, each widget executes its rendering code to perform the necessary drawing operations. The widgets must render only what is needed and minimize overlapping with other widgets to ensure optimal performance.
3. *Flush*: This phase involves copying the UI working buffer to the screen buffer. The VEE Port performs this operation, and it is the responsibility of the Firmware developer to optimize this process, for example, by utilizing a GPU.

   The MicroUI Flush Visualizer is a tool designed to visualize the drawing operations requested by widgets during the *Rendering* phase between each flush.

   The MicroUI Flush Visualizer provides the following information:

- A screenshot of the result of the Rendering Pipeline (what was shown on the screen).
- A list of drawing operations that were performed during the *Rendering* phase.

- The total area covered by the sum of the area drawn by the drawing operations as a percentage. A value of 100% indicates that the area drawn is equivalent to the surface of the entire display; a value of 200% indicates the area drawn is equivalent to twice the surface of the entire display.



This tool helps identify potential bottlenecks in the application's rendering by looking for flushes that exceed 200% of the display area.

A perfect application has 100% of its display area drawn. A total area drawn between 100% to 200% is the norm in practice because widgets often overlap. However, if the total area drawn is bigger than 200%, that means that the total surface of the display was drawn more than twice, meaning that a lot of time could be spent drawing things that are never shown.

As always, when conducting a performance study, measure. Use *SystemView* to identify the bottlenecks in your application on the embedded target. A total area drawn over 200% is inefficient, but your application may have bigger bottlenecks. Confirm it by measuring the time spent drawing vs. the time spent elsewhere between flushes.

Installation

1. Open the Front Panel file `.fp`.
2. Replace the `ej.fp.widget.Display` widget by `ej.fp.widget.FlushVisualizerDisplay`.

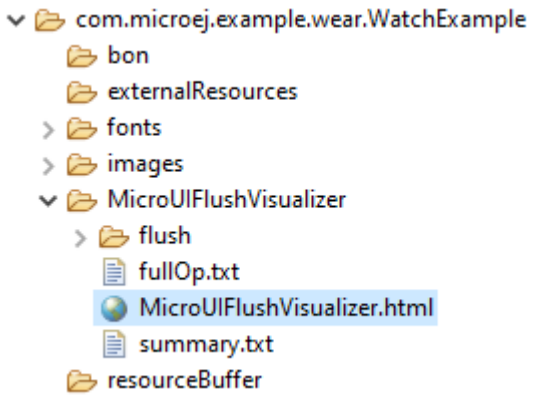
   For example:

```
<ej.fp.widget.FlushVisualizerDisplay␣
↪x="10" y="10" width="200" height="200" filter="f.png" />
```

The `FlushVisualizerDisplay` widget is available in the frontpanel widget module version 2.2.+ for UI Pack 13.4.0 or later and version 3.+ for UI Pack 13.5.1 or later.

Usage

1. Run the application in the Simulator.
2. Open the file `MicroUIFlushVisualizer/MicroUIFlushVisualizer.html` that was generated in the *application output folder*.



Limitations

Please refer to the javadoc of the `FlushVisualizerDisplay` widget.

Front Panel: Advanced Usage

The Front Panel is provided by the VEE Ports.

It is especially useful for those exposing a MicroUI display, LEDs and input devices. It then provides an interactive window for the Application simulation. See *Front Panel Mock*.

The following sections provides a non-exhaustive list of tips to make the best use of the Front Panel for Application developers.

Pixel Accurate Display: Window scaling

The Front Panel Display is designed for a pixel-accurate simulation of the GUI application. It uses MicroUI's graphics engine, not the system native engine. Therefore, the Front Panel does not support scaling of the window. For accurate graphics, the window must not be resized by the system or the application that launched the simulation (typically the IDE). Check your settings:

System (Windows)

Application (Windows)

On Windows, check your Display settings:

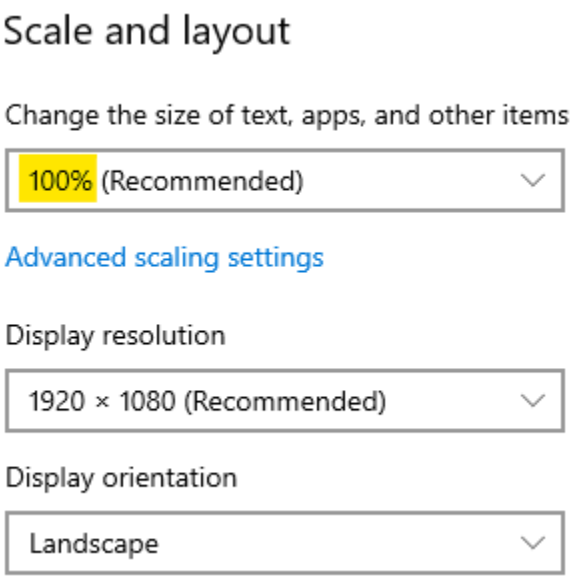


Fig. 28: Windows Display "Scale & Layout" settings

If needed, override the application auto scaling with the system's in Windows Explorer:

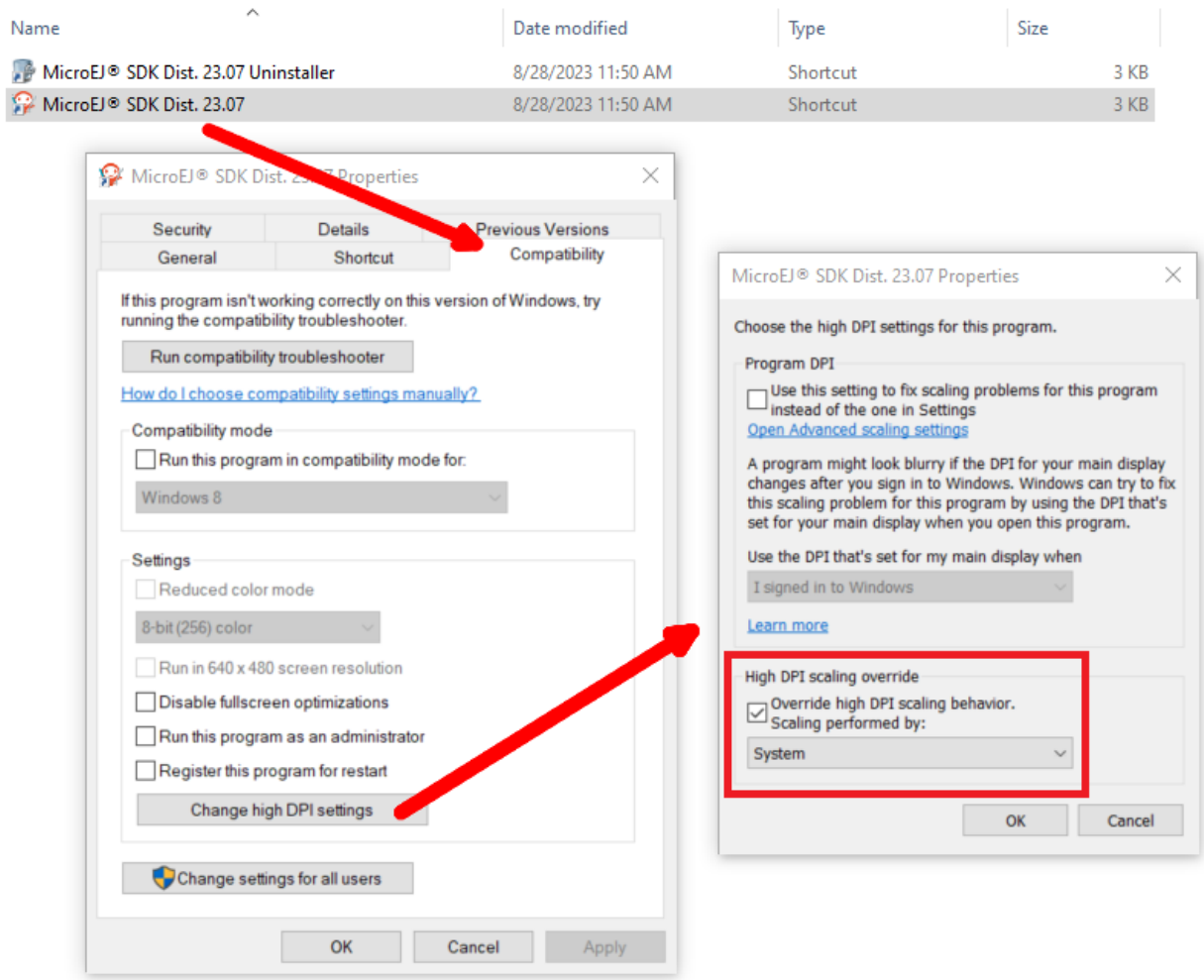


Fig. 29: Windows Application “High DPI scaling override” setting

Zoom on pixelated view for checking custom drawings

Assuming a pixel accurate simulated display (see *Window scaling*), use a screen magnifier tool to zoom on portions of the GUI. It is especially useful to check custom drawings as well as MicroUI Fonts (EJF) & Images. Also, make sure the tool does not apply a filter to smooth when scaling.

Windows Magnifier

Windows Magnifier Settings

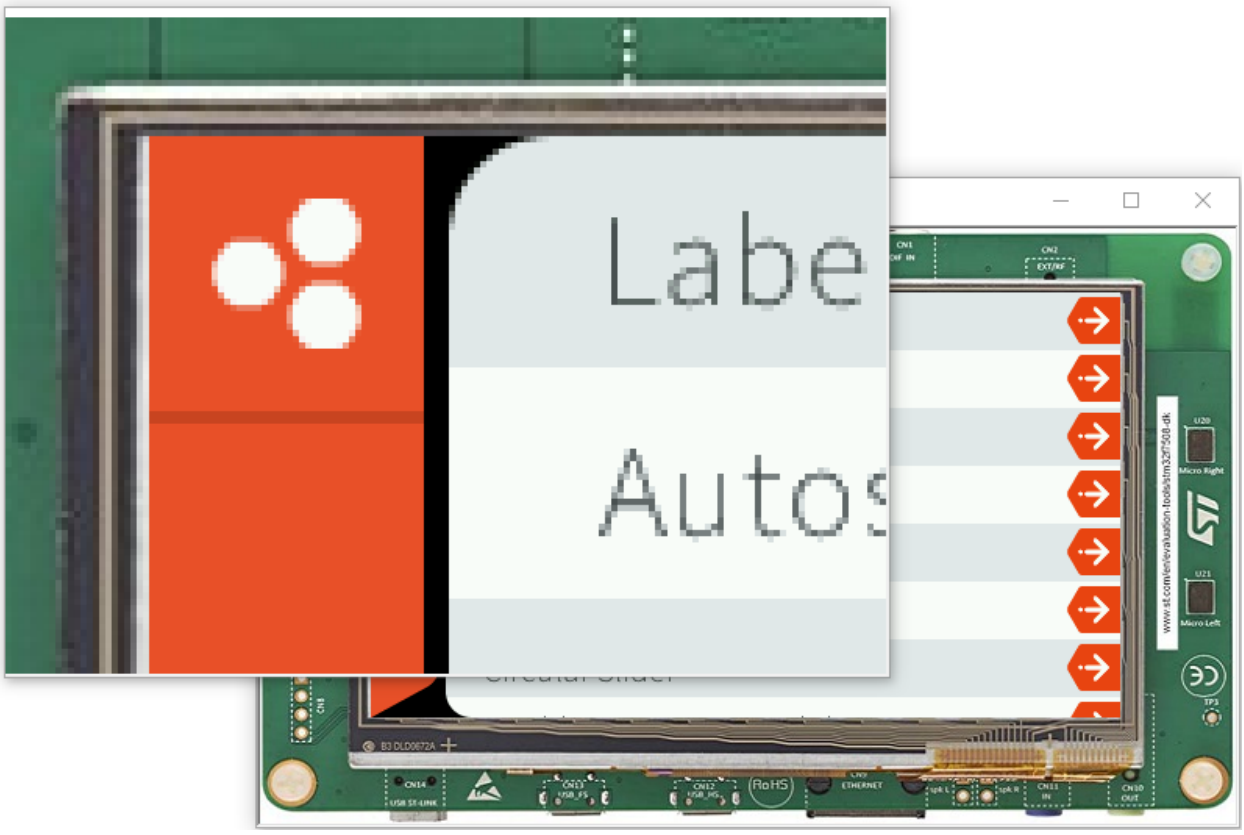


Fig. 30: Windows Magnifier Example

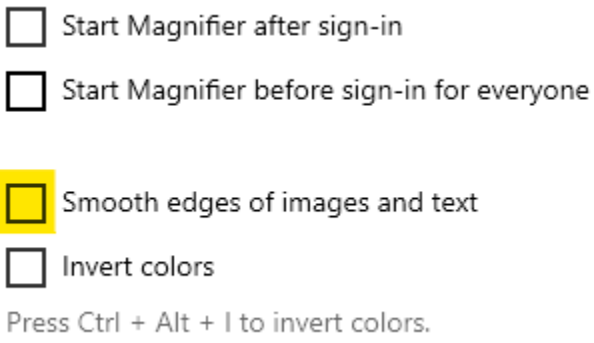


Fig. 31: Windows Magnifier “Smooth edges of images and text” setting

Take screenshots of the simulated display

An alternative is to make a screenshot and zooming with an image viewer/editor. For that, use a screenshot tool or programmatically extend the Front Panel with:

```
// Use ej.fp.widget package to access ej.fp.widget.Display.visibleBuffer
package ej.fp.widget;

import java.awt.Graphics;
import java.awt.Image;
import java.awt.Toolkit;
import java.awt.datatransfer.DataFlavor;
import java.awt.datatransfer.Transferable;
import java.awt.datatransfer.UnsupportedFlavorException;
import java.awt.image.BufferedImage;
import java.io.IOException;

import ej.fp.Device;
import ej.fp.widget.Button.ButtonListener;

public class SceenshotOnClick implements ButtonListener {

    @Override
    public void press(Button widget) {
        copyImageToClipboard(copyToType(takeScreenshot(),
↪ BufferedImage.TYPE_INT_RGB));
        System.out.println("Screenshot copied to clipboard");
    }

    @Override
    public void release(Button widget) {
        // do nothing
    }

    private static BufferedImage takeScreenshot() {
        Display display = Device.getDevice().getWidget(Display.class, null);
        return (BufferedImage) display.visibleBuffer.getRAWImage();
    }

    private static Image copyToType(BufferedImage src, int imageType) {
        BufferedImage␣
↪dst = new BufferedImage(src.getWidth(), src.getHeight(), imageType);
        Graphics g = dst.createGraphics();
        g.drawImage(src, 0, 0, null);
        g.dispose();
        return dst;
    }

    private static void copyImageToClipboard(Image image) {
        Toolkit.
↪getDefaultToolkit().getSystemClipboard().setContents(new Transferable() {

            @Override
```

(continues on next page)

(continued from previous page)

```
            public boolean isDataFlavorSupported(DataFlavor flavor) {
                return DataFlavor.imageFlavor.equals(flavor);
            }

            @Override
            public DataFlavor[] getTransferDataFlavors() {
                return new DataFlavor[] { DataFlavor.imageFlavor };
            }

            @Override
            public Object getTransferData(DataFlavor␣
↪flavor) throws UnsupportedFlavorException, IOException {
                if (!DataFlavor.imageFlavor.equals(flavor)) {
                    throw new UnsupportedFlavorException(flavor);
                }
                return image;
            }

        }, null);
    }

}
```

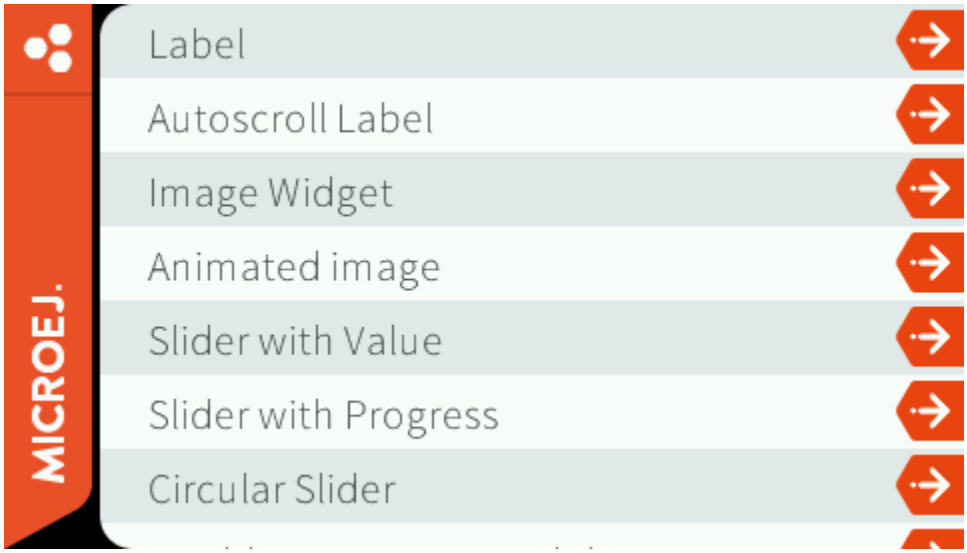
Visual Testing

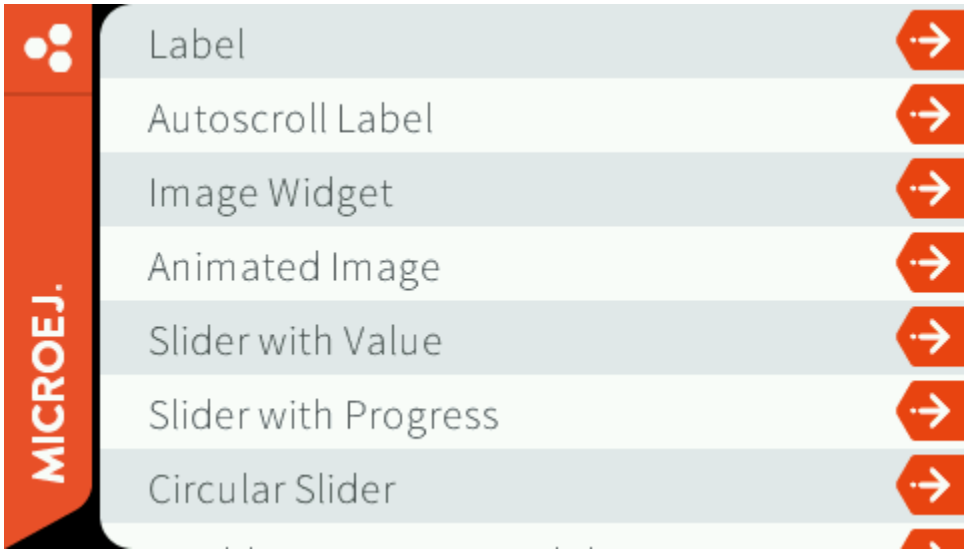
Such screenshots simplifies visual testing. The screenshot can be compared against a made-up image from design specification (typically exported from design tools), or against another screenshot taken from a different version of the application. To go further (and possibly automate such tests), use tools like ImageMagick:

Before

After

Compare

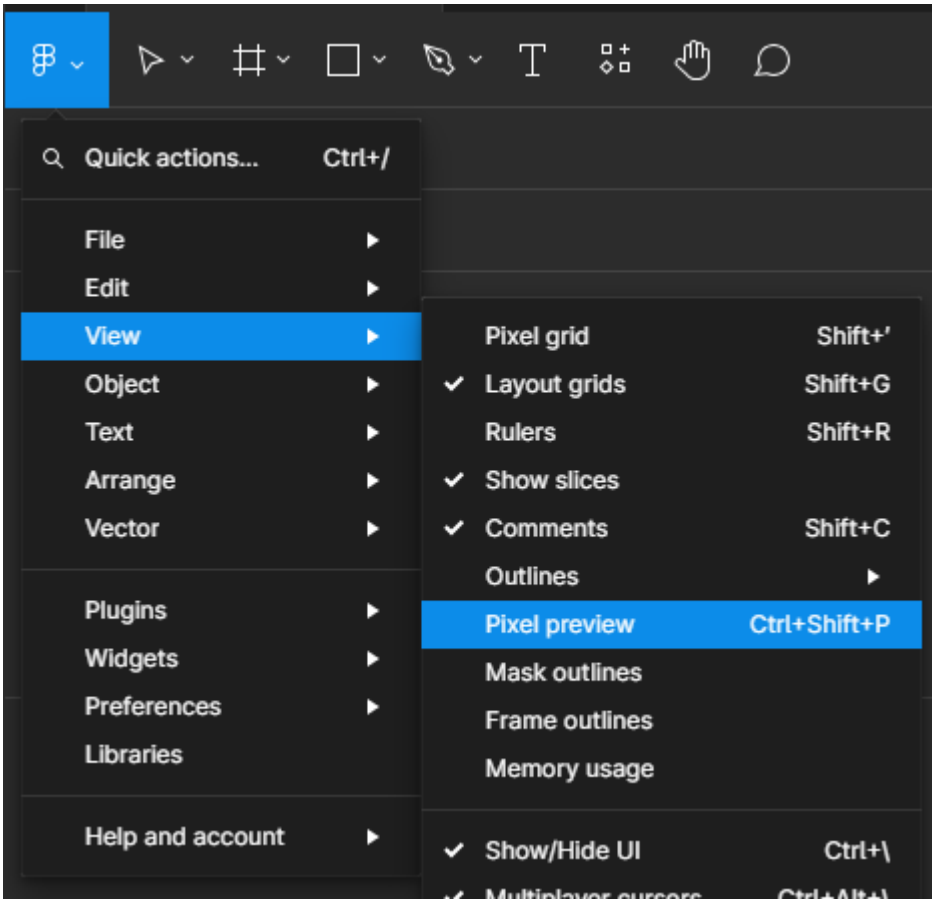





```
$ compare before.png after.png compare.png
```

Compare screenshots with Figma frames

In Figma, frames can be easily exported to PNG images. But it may need more processing before the comparison with the screenshot. First (optional), within Figma, enable Pixel Preview (`View` > `Pixel Preview (Ctrl+Shift+P)`):



Then, if the exported frame does not contain only the display, the image can be cropped with:

```
$ convert figma.png -crop 480x480+45+45 figma-cropped.png
```

Then, if the *MicroUI bpp setting* is not `RGB888`, the image can be filtered to match the supported colors. For example, for `bpp=RGB565`, apply the following filter:

```
$ convert figma.png -channel red,blue␣
↪-evaluate AND 63743 -channel green -evaluate AND 64767 figma-rgb565.png
```

Keep the Front Panel always on top

To keep the front panel visible while developing the application, use multiple displays and/or use tools like Microsoft PowerToys' Always on Top utility.

5.14.2 Native Language Support

Introduction

Native Language Support (NLS) allows the application to facilitate internationalization. It provides support to manipulate messages and translate them in different languages. Each message to be internationalized is referenced by a key, which can be used in the application code instead of using the message directly.

Principle

NLS is distributed as an add-on library containing a single Java interface: NLS.

In addition to that, the binary-nls library provides a factory for implementations of this interface: it uses an *add-on processor* which processes, offboard, the Localization Source Files into one BON resource buffer file for compactness.

During the *clinit* phase, this resource file is opened and the list of locales is parsed. After that, the resource remains opened for the rest of the Application execution and is directly used to retrieve messages translations for the supported locales.

Usage of this binary-nls implementation is documented below (see current *limitations*).

Localization Source Files

Messages must be defined in localization source files, located in the Classpath of the application (i.e. in the `src/main/resources` folder).

Localization source files can be either PO files or Android String resources.

Here is an example of a PO file:

```
msgid "Label1"
msgstr "My label 1"

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

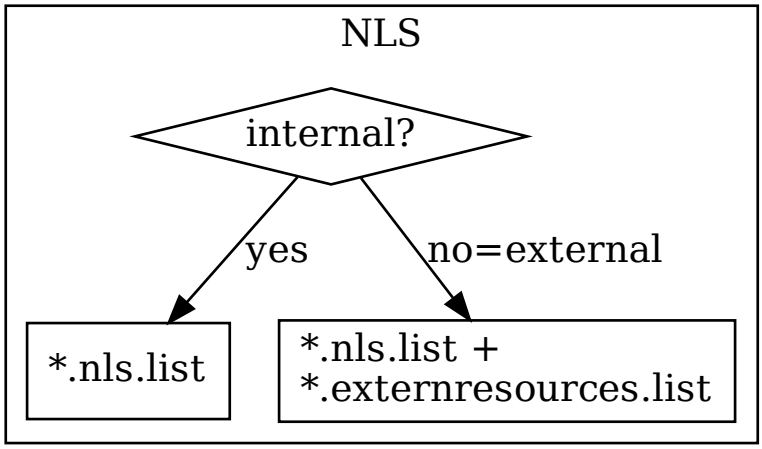
And here is an example of an Android String resource:

```
<resources>
   <string name="Label1">My label 1</string>
   <string name="Label2">My label 2</string>
</resources>
```

Note: When using Android String resources, string arrays are also supported. However, plurals are not supported.

NLS List Files

Localization source files are declared in *Classpath* `*.nls.list` files (**and** to `*.externresources.list` for an external resource, see *Application Resources* and *Loading Translations as an External Resource*).



The file format is a standard Java properties file, each line represents the Full Qualified Name of a Java interface that will be generated and used in the application. Example:

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

Usage

The binary-nls module must be added to the Application project build file:

Gradle (build.gradle.kts)

MMM (module.ivy)

```
implementation("com.microej.library.runtime:binary-nls:2.5.0")
```

```
<dependency org="com.microej.library.runtime" name="binary-nls" rev="2.5.0"/>
```

This module includes an Add-On Processor which parses the localization source files. For each interface declared in the NLS list files, all the localization source files whose names start with the interface name are used to generate:

- a Java interface with the given FQN, containing a field for each message of the localization source files
- a NLS binary file containing the translations

So, in the example, the generated interface `com.mycompany.myapp.Labels` will gather all the translations from files named `Labels*` and located in any package of the Classpath. The names of the localization source files should be suffixed by their locale (for example `Labels_en_US.po`).

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

Besides the message fields, the generated interface declares an NLS instance which is automatically created in the clinit of the interface.

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

```
String label = Labels.NLS.getMessage(Labels.Label1);
```

For the application to know which language to use among those made available and when, you can set it and change it at any point using the setCurrentLocale(locale) method. If no locale has been set yet when getting a message, the translation for the first locale available in alphabetical order will be used by default. However, you can also pick this locale to default to yourself, by adding a `com.microej.binarynls.defaultLocale` property followed by a locale name in a `.properties.list` file.

Plural Forms

Starting with version 4.0.0 of the NLS module and version 3.0.0 of the binary-nls module is introduced support of GNU gettext's plural form feature in PO files. This allows usage of `Plural-Forms` header entries and several `msgstr` 's per `msgid` (referred to as plural forms) as specified by gettext; you can then retrieve the correct message in a locale for a given count of things by using the ej.nls.NLS.getMessage() methods that take in this count value as an argument.

If a message for a given `msgid` has a `msgid_plural` and plural forms in a PO file for an interface declared in an NLS list file, it must also have plural forms in all other PO files for this interface.

Warning: Please note that one significant difference with gettext's implementation is that the expression described in the `plural` field of the `Plural-Forms` header must be a valid **Java** expression returning an `int`, as opposed to a C expression. A usual case in which this makes a difference is for expressions that rely on boolean values being evaluated as zero or one in C, such as in:

```
"Plural-Forms: nplurals=2; plural=n != 1;\n"
```

This expression will not work with our implementation as Java does not interpret booleans as integers. An easy way to convert this expression would be:

```
"Plural-Forms: nplurals=2; plural=n != 1 ? 1 : 0;\n"
```

Also note that the validity of these provided expressions is not entirely checked. Providing an expression that is not valid Java or that would return an invalid plural form index would cause errors at runtime or even in the Java files generated by the Add-On Processor.

Dealing With Missing Translations

By default, if a translation is missing for a given `msgid` in a PO file in a given language, the message returned by the ej.nls.NLS.getMessage() method with the locale set to this language will simply be the `msgid` itself. In the case of an XML Android String resource, the `name` attribute of a missing `string` element will be returned. However if returning this identifier is not a suitable solution, you might want to set a fallback locale parameter for an interface. This parameter corresponds to a language to print the translation for a message in, in case it is not available in the current language.

Starting with version 2.5.0 of the binary-nls module, you can set this fallback locale by specifying a locale name in a `.nls.list` file, after the name of the interface you want this locale to be the fallback for, separated by a colon `:`. For example, with the following `.nls.list` file, if a translation is missing in a language for a message in the `Labels` and `Messages` PO/XML files, the message will be translated to `en_US` instead of just returning its `msgid` / `name`.

```
# Missing translations for Labels and Messages will fall back to en_US
com.mycompany.myapp.Labels:en_US
com.mycompany.myapp.Messages:en_US
```

As such, you can specify a different fallback locale for each interface in a `.nls.list` file. For example, with the following `.nls.list` file, the messages in `Labels` will not have a fallback language set and will only return the `msgid` / `name` if a translation is missing, while missing translations will default to `en_US` for the messages in `Messages`, and to `ja_JP` for the messages in `Content`:

```
# Missing translations for Labels will fall back to their msgid/name
com.mycompany.myapp.Labels

# Missing translations for Messages will fall back to en_US
com.mycompany.myapp.Messages:en_US

# Missing translations for Content will fall back to ja_JP
com.mycompany.myapp.Content:ja_JP
```

In the case of a message with plural forms in PO files, this works much the same way, using the messages and forms in the fallback locale if available. If no fallback locale is specified or if the

requested message is not specified in it, then the `msgid` will be used for a count value of 1, and the `msgid_plural` will be used for any other value, as gettext would function.

BinaryNLS Resource Generation

If the classpath of the Application contains `.po` / `.xml` files and `.nls.list` files, the `binary-nls` Add-On Processor will generate the following source files for each NLS interface:

- a `.resourcebuffer`
- a `.resourcebuffer.list` which references the `.resourcebuffer`
- a `.resources.list` which references the resource (this resource does not exist yet but it will be generated later)

When building the Application or running it on Simulator, the Resource Buffer Generator is first executed. Based on the `.resourcebuffer` and the `.resourcebuffer.list`, it will generate a resource.

Since the generated resource is referenced by the `.resources.list` generated by the `binary-nls` ADP, the SOAR will embed the resource in the Application binary. Unless it is also referenced by an `.externresources.list` in which case the SOAR will output the resource in the *External Resources Folder* instead.

This resource is loaded as soon as the BinaryNLS instance is created, in the clinit of the generated NLS interface (see *Principle*).

Limitations

The latest BinaryNLS implementation does not support:

- to dynamicly add a new locale
- to dynamicly modify messages translations

even when the resource is external (see *External resource loader*).

For any addition / modification, the Application must be restarted and, typically, the full resource buffer must be updated (not only the part of the added/modified locale).

Also, there is no API to close the resource buffer. If it is external, the Application must be stopped to close this resource, before it can potentially be modified depending on the external resource loader.

NLS External Loader Tool

The NLS External Loader tool allows to update the PO files of an application executed on a Virtual Device without rebuilding it. PO files can be dropped in a given location in the Virtual Device folders to dynamically replace the language strings packaged in the application.

This is typically useful when testing or translating an application in order to have a quick feedback when changing the PO files. Once the PO files are updated, a simple restart of the Virtual Device allows to immediately see the result.

Installation

To enable the NLS External Loader in the Virtual Device, add the following dependency to the Firmware project:

Gradle (build.gradle.kts)

MMM (module.ivy)

```
implementation("com.microej.tool:nls-po-external-loader:2.3.0")
```

```
<dependency org="com.microej.
↪tool" name="nls-po-external-loader" rev="2.3.0" transitive="false"/>
```

Then rebuild the Firmware project to produce the Virtual Device.

Usage

Once the project built:

- unzip the Virtual Device and create a folder named `translations` in the root folder.
- copy all the PO files from the project into the `translations` folder. All PO files found in this folder are processed, no matter their folder level.
- start the Virtual Device with the launcher. The following logs should be printed if the NLS External Loader has been executed and has found the PO files:

```
externalPoLoaderInit:init:

externalPoLoaderInit:loadPo:
   [mkdir] Created dir: <PATH>\tmp\microejlaunch1307817858\resourcebuffer
[po-to-nls] *.nls files found in <PATH>\output\<FIRMWARE>\resourceBuffer :
[po-to-nls]   - com.mycompany.Messages1
[po-to-nls]   - com.mycompany.Messages2
[po-to-nls] Loading *.po files for NLS interface com.mycompany.Messages1
[po-to-nls]   => loaded locales : fr_FR,de_DE,ja_JP,en_US
[po-to-nls] Loading *.po files for NLS interface com.mycompany.Messages2
[po-to-nls]   => loaded locales : fr_FR,de_DE,ja_JP,en_US
```

- update the languages strings in the PO files of the Virtual Device (the files in the *translations/* folder).
- restart the Virtual Device and check the changes.

It is important to know the following rules about the NLS External Loader:

- the external PO files names must match with the default PO files names of the application to be processed.
- when PO files with a given name are loaded, the default translations for these PO files are replaced, there is no merge. It means that:
  - if messages are missing in the new PO files, they are not available anymore for the application and may very probably make it crash.
  - if languages are missing (the application has 3 PO files for English, French and Spanish, and only PO files for English and French are available in the translations folder), the messages of

the missing languages are not available anymore for the application and may very probably make it crash.

- if new messages are added in the PO files, it has no impact, they are ignored by the application.

- External PO files are loaded at Virtual Device startup, so any change requires a restart of the Virtual Device to be considered

Troubleshooting

java.io.IOException: NLS-PO:S=4

The following error occurs when at least 1 PO file is missing for a language:

```
[parallel2] NLS-PO:I=6
[parallel2] Exception␣
↪in thread "main" java.io.IOException: NLS-PO:S=4 323463627 -1948548092
[parallel2]     at java.lang.Throwable.fillInStackTrace(Throwable.java:79)
[parallel2]     at java.lang.Throwable.<init>(Throwable.java:30)
[parallel2]     at java.lang.Exception.<init>(Exception.java:10)
[parallel2]     at java.io.IOException.<init>(IOException.java:16)
[parallel2]     at com.microej.nls.BinaryNLS.loadBinFile(BinaryNLS.java:310)
[parallel2]     at com.microej.nls.BinaryNLS.<init>(BinaryNLS.java:157)
[parallel2]     at com.microej.nls.BinaryNLS.newBinaryNLS(BinaryNLS.java:118)
```

Make sure that all PO files are copied in the `translations` folder.

Crowdin

Crowdin is a cloud-based localization platform which allows to manage multilingual content. The NLS External Loader can fetch translations directly from Crowdin to make the translation process even easier. Translators can then contribute and validate their translations in Crowdin and apply them automatically in the Virtual Device.

A new dependency must be added to Firmware project dependencies to enable this integration:

Gradle (build.gradle.kts)

MMM (module.ivy)

```
implementation("com.microej.tool:nls-po-crowdin:1.0.0")
```

```
<dependency org=
↪"com.microej.tool" name="nls-po-crowdin" rev="1.0.0" transitive="false"/>
```

Once the module has been built, edit the file `platform/tools/crowdin/crowdin.properties` to configure the Crowdin connection:

- set `crowdin.token` to the Crowdin API token. A token can be generated in the Crowdin in `Settings` > `API` > click on `New Token`.
- set `crowdin.projectsIds` to the id of the Crowdin project. The project id can be found in the `Details` section on a project page. Multiple projects can be set by separating their id with a comma (for example `crowdin.projectsIds=12,586,874`).

When the configuration is done, the fetch of the Crowdin translations can be done by executing the script `crowdin.bat` or `crowdin.sh` located in the folder `platform/tools/crowdin/`. The PO files retrieved from Crowdin are automatically pasted in the folder `translations`, therefore the new translations are applied after the next Virtual Device restart.

5.14.3 Networking

Foundation Libraries

Name	Description	Module Link	API Link	Use
Bluetooth	Bluetooth Low Energy (BLE) based on the Generic Attribute Profile (GATT).	bluetooth	ej.bluetooth package	• Bluetooth utility Library
ECOM-Network	Network interfaces management and IP configurations.	ecom-network	NetworkInterfaceManager class	
ECOM-WIFI	Wi-Fi connectivity.	ecom-wifi	WifiManager class	• Wi-Fi setup Example • Wi-Fi utility Library
NET	Client and Server raw TCP/IP sockets.	net	java.net package	• NET Example • NET utility Library
Security	Cryptographic operations.	security	javax.crypto package	
SSL	Client and Server secure sockets layer using Transport Layer Security (TLS) protocols.	ssl	java.net.ssl package	• SSL mutual client Example • SSL mutual server Example • SSL utility Library

Add-On Libraries

IoT Libraries

Name	Description	Module Link	API Link	Use
Android Connectivity	Network connection state and notifications.	android-connectivity	ConnectivityManager class	• Connectivity Example
HTTP Client	OpenJDK HTTP client.	http-client, http-sclient	HttpURLConnection class	• HTTP client README
Web Server (HOKA)	Tiny footprint yet extensible web server.	HOKA	HttpServer class	• *HOKA User Manual* • HOKA Examples
MQTT Client (MicroPaho)	Tiny footprint MQTT 3.1.1 client based on Eclipse Paho Java APIs.	micropaho	MqttClient class	• MicroPaho README • MQTT publish Example • MQTT subscribe Example
REST Client	REpresentational State Tranfer (REST) client.	restclient	Resty class	• REST client README
SNTP Client	Simple Network Time Protocol (SNTP) client, used to retrieve the current time from an NTP server.	sntp-client	SntpClient class	• SNTP client README
WebSocket Client	WebSocket client (RFC 6455).	web-socket, websocket-secure	WebSocket class	• WebSocket client README • WebSocket client Example

Data Serialization Libraries

Name	Description	Module Link	API Link	Use
CBOR	Concise Binary Object Representation (CBOR) encoder and decoder (RFC 7049).	cbor	• CborEncoder class • CborDecoder class	• *CBOR Tutorial*
JSON	JavaScript Object Notation (JSON) encoder and decoder.	json	• JSONObject class (decoder) • JSONWriter class (encoder)	• README • *JSON Tutorial*
Protocol Buffers	Google Protocol Buffers 3 encoder and decoder, supporting files compiled by `protoc` with lite plugin.	protobuf3	• CodedInputStream class (decoder) • CodedOutputStream class (encoder)	• Protobuf3 Example
XML	eXtensible Markup Language encoder and decoder (kXML 3).	kxml2	• XmlPullParser class (decoder) • XmlSerializer class (encoder)	• *XML Tutorial*

Cloud Agent Libraries

Name	Description	Module Link	Use
AWS IoT Core	AWS IoT Core client, providing publish/subscribe functionalities.	aws-iot	• AWS IoT Core README • AWS IoT Core Example
Google Cloud Platform Iot Core	Google Cloud Platform Iot Core client.	gcp-iotcore	• Google Cloud Platform Getting Started

HOKA Web Server

HOKA is a tiny extensible Java web server for embedded applications.

It comes with the support of HTTP, HTTPS, Server session, and routing for REST API.

Note: This is the documentation of the latest version of HOKA library 8.X.X

Intended Audience

The intended audience for this document is Java developers who are familiar with socket communication, the HTTP 1.1 protocol, and web server concepts.

Getting Started

Create a new MicroEJ application and add the HOKA library dependency to your MicroEJ application

Gradle (build.gradle.kts)

MMM (module.ivy)

```
implementation("ej.library.iot:hoka:8.4.0")
```

```
<dependency org="ej.library.iot" name="hoka" rev="8.4.0"/>
```

```
public class MyServer {

  public static void main(String[] args) throws IOException {

      HttpServer http = HttpServer.builder().port(8080).build();

      http.get("/hello", new RequestHandler() {

        @Override
        public void process(HttpRequest request, HttpResponse response) {
          response.setData("Hello world!");
        }
      });

      http.start();

  }
}
```

Run the application and check the result at **http://localhost:8080/hello**

Routes Mapping

In HOKA, an HTTP request is a combination of 4 elements:

- **Verb**: The HTTP verbs, GET, POST, PUT or DELETE…
- **Path**: The request path or URI. `/hello/:username`
- **Handler**: The request handler process the request and respond to the client.
- **content type**: (optional) the route supported content type

Note: Paths are matched in the order of their creation. The handler of the first matching path will be invoked. All the paths need to be registered before calling the `start()` method of the `HttpServer` instance. If no path matches the incoming request, the server will return a 404 Not Found response.

```
HttpServer http = HttpServer.builder().port(8080).build();

http.get("/", new RequestHandler() {

  @Override
  public void process(HttpRequest request, HttpResponse response) {
    // read a resource

  }
});

http.post("/", new RequestHandler() {

  @Override
  public void process(HttpRequest request, HttpResponse response) {
    // write a resource
  }
});

http.put("/", new RequestHandler() {

  @Override
  public void process(HttpRequest request, HttpResponse response) {
    // update a resource
  }
});

http.delete("/", new RequestHandler() {

  @Override
  public void process(HttpRequest request, HttpResponse response) {
    // delete a resource
  }
});
```

A path can be registered with **one single** specific content type in different request handlers on the same path.

For example, to map two content types on the same path, do the following:

```
HttpServer http = HttpServer.builder().port(8080).build();

http.get("/", "application/json", new RequestHandler() {

  @Override
  public void process(HttpRequest request, HttpResponse response) {
    // read a resource and return a json formatted response.
  }
});

http.get("/", "application/xml", new RequestHandler() {

  @Override
  public void process(HttpRequest request, HttpResponse response) {
    // read a resource and return a json formatted response.
  }
});
```

Path Parameters

The request path can contain named parameters called path parameters. Those parameters are made available through the `request` instance of the `process()` method of the `RequestHandler`. The path parameter can be accessed by calling `HttpRequest#getPathParam(String param)`

```
HttpServer http = HttpServer.builder().port(8080).build();

http.get("/hello/:name", new RequestHandler() {

  @Override
  public void process(HttpRequest request, HttpResponse response) {
    String name = request.getPathParam("name");
    response.setData("Hello " + name);
  }
});
```

Splat Parameters

The request path also supports splat parameters using wildcard '*'. Those parameters are made available through the `request` instance of the `process()` method of the `RequestHandler`. The splat parameters array can be accessed by calling `HttpRequest#getSsplatParams()`

```
HttpServer http = HttpServer.builder().port(8080).build();

http.get("/greet/*/by/*", new RequestHandler() {

  @Override
  public void process(HttpRequest request, HttpResponse response) {
    String name = request.getSplatParams().get(0);
    String greeting = request.getSplatParams().get(1);
```

(continues on next page)

(continued from previous page)

```
    response.setData(greeting + " " + name);
  }
});
```

Request

- `HttpRequest#getMethod()` : returns the request method (1 for `POST` , 2 for `GET` , 3 for `PUT` and 4 for `DELETE` . see HttpRequest for the full list).
- `HttpRequest#getURI()` : returns the requested URI.
- `HttpRequest#getQueryParams()` : returns the request query parameters map.
- `HttpRequest#getQueryParam(String)` : returns the query parameter by the given name from the query parameters map.
- `HttpRequest#getPathParam(String)` : returns the request path parameter by the given name.
- `HttpRequest#getSplatParams()` : returns the list of splat parameters.
- `HttpRequest#setAttribute(String, Object)` : set a server-side request attribute. can be used to passe data between handlers.
- `HttpRequest#getAttribute(String)` : get a server-side request attribute.
- `HttpRequest#getVersion()` : returns the HTTP protocol version of the request.
- `HttpRequest#getHeaders()` : returns the request headers, all header field names are converted to lowercase.
- `HttpRequest#getHeader(String)` : returns the value of the header with the given name.
- `HttpRequest#parseBody(BodyParser)` : parses the body of the request with the given parser.
- `HttpRequest#getRequestBody()` : return the request InputStream to be used for any custom request handling.

Body Parsers

The `HttpRequest#parseBody(BodyParser)` is used to read the body (data) of a request.

HOKA library provides 4 implementations of `BodyParser` :

- `StringBodyParser` : returns the full request body as a string.
- `MultipartStringsParser` : parse a `multipart/*` request body, each part is returned as a string.
- `MultiPartBodyParser` : parse a `multipart/*` body, and parse each part as header fields and an `InputStream` body.
- `ParameterParser` : parse an `application/x-www-form-urlencoded` request body.

Cookies

The cookies are lazily parsed the first time they are accessed.

- `HttpRequest#getCookies()` : returns the list of cookies.
- `HttpRequest#getCookie(String)` : returns the value of the cookie by the given name.

Response

Build a `HttpResponse` based on the request with the following data :

- `data` : the body of the response as a `String`, `byte[]` or as an `InputStream`.
- `status` : the status of the response to send. HTTP response code.
- `mimeType` : the value of the `content-type` header.
- `HttpResponse#addHeader(String name, String value)` : adds a header with given name and value.
- `HttpResponse#addCookie(Cookie)` : adds a cookie to the response. USe ej.hoka.http.Cookie.Builder() to create a cookie instance.

```
// Use the cookie builder to create a cookie instance.
Cookie cookie = Cookie.builder().name("cookieName").value("cookieValue")
                                .expires(expirationDate)
                                .maxAge(900)
                                .domain("www.example.com")
                                .path("/api")
                                .sameSite(SameSite.Strict)
                                .secure()
                                .httpOnly()
                                .build();
```

MIME Types

The `Mime` class provides constant values for commonly used MIME types and utility methods to return the MIME type of a resource name based on file extensions.

The predefined MIME types are :

- MIME_PLAINTEXT = “text/plain”
- MIME_HTML = “text/html”
- MIME_XML = “text/xml”
- MIME_APP_JSON = “application/json”
- MIME_DEFAULT_BINARY = “application/octet-stream”
- MIME_CSS = “text/css”
- MIME_PNG = “image/png”
- MIME_JPEG = “image/jpeg”
- MIME_GIF = “image/gif”

- MIME_JS = “application/x-javascript”
- MIME_FORM_ENCODED_DATA = “application/x-www-form-urlencoded”
- MIME_MULTIPART_FORM_ENCODED_DATA = “multipart/form-data”

The method `Mime#getMIMEType(String URI)` returns the MIME type of the given URI, assuming that the file extension in the URI was previously registered with the `Mime#mapFileExtensionToMIMEType(String fileExtension, String mimeType)`. Only lower case file extensions are recognized.

For example, calling `getMIMEType("/images/logo.png")` will return the string `"image/png"`.

The following table shows the predefined assignments between file extensions and MIME types:

Extension	MIME type
“.png”	`MIME_PNG`
“.css”	`MIME_CSS`
“.gif”	`MIME_GIF`
“.jpeg”	`MIME_JPEG`
“.jpg”	`MIME_JPEG`
“.html”	`MIME_HTML`
“.htm”	`MIME_HTML`
“.js”	`MIME_JS`
“.txt”	`MIME_PLAINTEXT`
“.xml”	`MIME_XML`

Halt Request Processing Chain

to stop a request processing and return immediately. The following static methods form `HttpServer` class should be used.

This will cause the request handler to stop immediately and the response will be returned to the client without executing other filters.

This is useful for error handling for example.

```
halt();  <--- return a 200 OK response.
halt(HTTPConstants.HTTP_STATUS_UNAUTHORIZED);
halt(HTTPConstants.HTTP_STATUS_UNAUTHORIZED, "login required!");
```

Filters

A filter is also a request handler that is executed before or after a registered request.

It needs to be registered before calling the `start()` method on the server instance.

It can be used to pre-process or post-process a request.

Multiple filters can be registered. They will be executed in the order they were added in.

HOKA supports 4 types of filters.

- **before all requests**: runs before any registered path.
- **before a specific path**: runs before a specific registered path.

- **after a specific path**: runs after a specific registered path.
- **after all requests**: runs after any registered path.

Before

Example of adding a filter that will be executed before any registered path.

Multiple before filters can be added by calling `before()` multiple times. They will be executed in their registration order.

```
HttpServer http = HttpServer.builder().port(8080).build();

http.before(new RequestHandler() {

  @Override
  public void process(HttpRequest request, HttpResponse response) {
    boolean authenticated = false;
    // check if authenticated ...
    if (!authenticated) {
      halt(HTTPConstants.
↪HTTP_STATUS_UNAUTHORIZED); // stop the processing and return an error.
    }
  }
});
```

Example of adding a filter that will be executed before a specific registered path.

Unlike global before filters, only one before filter by path can be registered.

```
HttpServer http = HttpServer.builder().port(8080).build();

http.before("/private/*", new RequestHandler() {

  @Override
  public void process(HttpRequest request, HttpResponse response) {
    // check access privilege ...
    halt(HTTPConstants.
↪HTTP_STATUS_FORBIDDEN); // stop the processing and return an error.
  }
});
```

After

Example of adding a filter that will be executed after any registered path.

Multiple global after filters can be added by calling `after()` multiple times. They will be executed in their registration order.

```
HttpServer http = HttpServer.builder().port(8080).build();

http.after(new RequestHandler() {
```

(continues on next page)

(continued from previous page)

```
  @Override
  public void process(HttpRequest request, HttpResponse response) {
    // do some post processing on the request/response
    response.addHeader("common header key", "common header value");
  }
});
```

Example of adding a filter that will be executed after a specific registered path.

Unlike global after filters, only one after filter by path can be registered.

```
HttpServer http = HttpServer.builder().port(8080).build();

http.after("/private/*", new RequestHandler() {

  @Override
  public void process(HttpRequest request, HttpResponse response) {
    // do some post processing on the request/response
    response.addHeader("special header key", "special header value");
  }
});
```

Error Handling

Not Found Error

The 404 not found error can be customized by using the `HttpServer#notFoundError()` method.

```
HttpServer http = HttpServer.builder().port(8080).build();

// html, The html page can be loaded form a file
http.
↪notFoundError("<html><body><h1>404 Page doesn't exist</h1></body></html>");

// json format
http.notFoundError(
↪"{\"message\":\"404 Page doesn't exist\"}", "application/json");
```

Internal Server Error

The 500 Internal Server Error can also be customized.

```
// html, The html page can be loaded form a file
http.internalServerError(
↪"<html><body><h1>505 Something went wrong!</h1></body></html>");

// json format
http.internalServerError(
↪"{\"message\":\"505 Something went wrong!\"}", "application/json");
```

Exception Mapping

An exception can be mapped to a custom handler to return specific errors.

```
HttpServer http = HttpServer.builder().port(8080).build();

http.get("/throwerror", new RequestHandler() {

  @Override
  public void process(HttpRequest request, HttpResponse response) {
    throw new MyCustomError();
  }
});

http.exception(MyCustomError.class, new RequestHandler() {

  @Override
  public void process(HttpRequest request, HttpResponse response) {
    // handle the custom error here.
  }
});
```

Static Files

A specific static file handler can be set to serve files from the application classpath by using `ClasspathFilesHandler` class.

```
HttpServer http = HttpServer.builder() //
    .port(8080) //
    .staticFilesHandler(ClasspathFilesHandler.
↪builder() // set the static file handler
        .rootDirectory(
↪"/public") // set the static file folder form src/main/resources
        .build())
    .build();
```

Note that the public directory name is not included in the request URL. to access a file in `src/main/resources/public/css/main.css` the url is `http://localhost:8080/css/main.css`

An external file location can be used by providing your own implementation of `StaticFilesHandler` interface and adding the `fs` foundation library to work with File* classes from `java.io`.

Web Server Configuration

`HttpServer` class builder has the following options :

```
HttpServer http = HttpServer.builder()
    .port(8080) //␣
↪setup the port number to bind the server socket on. Use 0 for a random port
    .simultaneousConnections(3)␣
↪// setup the max simultaneous connections accepted by the server
    .workerCount(3)␣
↪// setup the number of threads to handle incoming connections
    .connectionTimeout(60 * 1000) // setup connection timeout
    .encodingRegistry(new EncodingRegistry())␣
↪// register a custom the content encoding & transfer-coding registry
    .secure(/**SSLContext#getServerSocketFactory()*/) // setup SSL / HTTPS
    .apiBase("/api/v1/") // setup a common URI base for all␣
↪relative registered path. relative means, the path do not starting with a /
    .staticFilesHandler(staticFilesHandler) // setup the static files handler
    .withTrailingSlashSupport()␣
↪// process route with trailing slash as different routes
    .withStrictAcceptContentEncoding() // activate strict␣
↪content acceptance. return 406 Not Acceptable for unknown content-encoding
    .developmentMode()// enable␣
↪development mode, send error stack trace to the client side as in html
    .build();
```

Trailing Slash Matching

By default, the HOKA server ignores the trailing forward slash at the ends of the request URI.

For example:

- `GET | host/hello`
- `GET | host/hello/`

Will link to the same request handler.

This behavior can be deactivated by calling the method `HttpServer#builder()#withTrailingSlashSupport()` on the server builder.

Note that `host` and `host/` will link to the same request handler whatever the Trailing Slash Match is activated or not.

Development Mode

Development mode can be activated by calling `HttpServer#builder()#developmentMode()`` .

This will tell the HOKA server to send the exception stack trace to the client.

The stack trace is sent in a plain text response. This is useful when developing the web application; otherwise, a “500 Internal Error” response is sent.

Note: when development mode is active, internal error page customization is deactivated. The development mode page is returned instead.

Generate Server Self Signed Key and Certificate for HOKA WebServer TLS

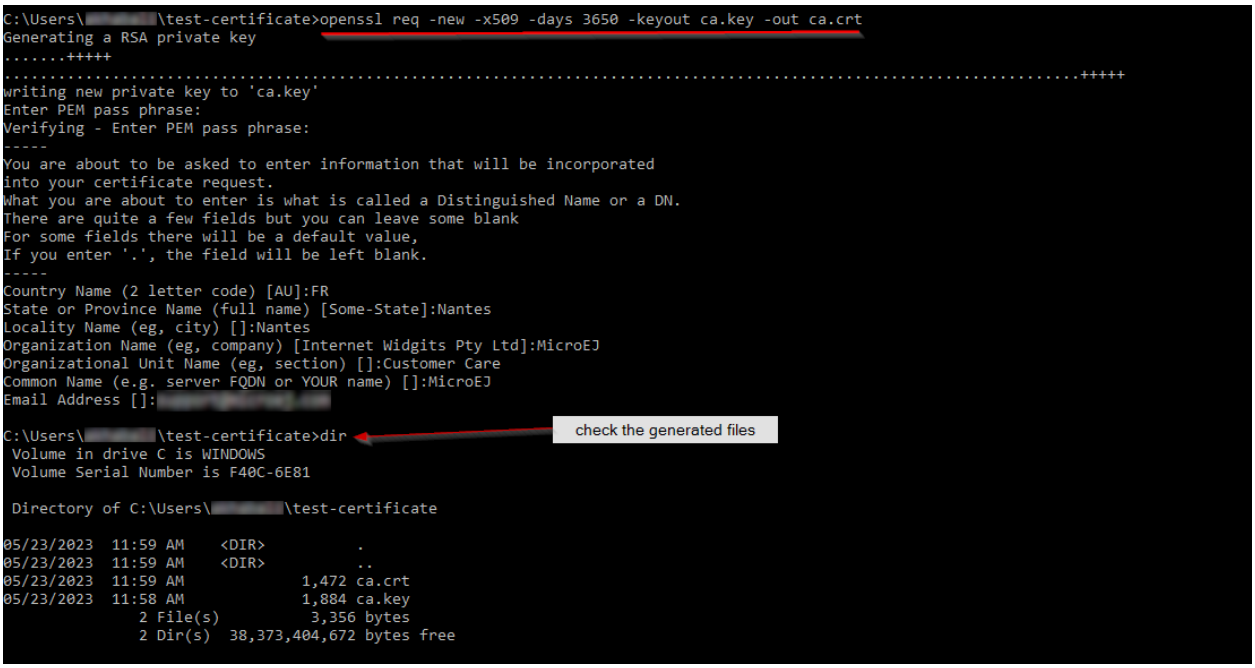
This section details the commands and steps to generate a self signed certificate and a DER formatted key for HOKA server to enable TLS.

Generate Root CA Key & Certificate

To generate a root certificate authority (CA) using openssl, execute the following command and follow the instructions by filling the certificate information:

```
openssl req -new -x509 -days 3650 -keyout ca.key -out ca.crt
```

- `ca.key` : is the name of the generated root private key in PEM format.
- `ca.crt` : is the name of the generated root certificate in PEM format.



Generate HOKA Server Private Key

To generate a private key using openssl, execute the following command:

```
openssl genrsa -out hoka.key 4096
```

- `hoka.key` : is the name of the generated private key.
- `4096` : is the length of the private key.

Generate HOKA Server Self Signed Public Key

To generate a Self signed public key:

1. Generate a certificate signing request (CSR) using openssl, for that execute the following command and fill in the information: `openssl req -new -sha256 -key hoka.key -out hoka-csr.pem`

```
C:\Users\[redacted]\test-certificate>openssl req -new -sha256 -key hoka.key -out hoka-csr.pem
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:FR
State or Province Name (full name) [Some-State]:Nantes
Locality Name (eg, city) []:Nantes
Organization Name (eg, company) [Internet Widgits Pty Ltd]:MicroEJ
Organizational Unit Name (eg, section) []:Customer Care
Common Name (e.g. server FQDN or YOUR name) []:MicroEJ
Email Address []:[redacted]

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:MicroEJ

C:\Users\[redacted]\test-certificate>dir
 Volume in drive C is WINDOWS
 Volume Serial Number is F40C-6E81

 Directory of C:\Users\[redacted]\test-certificate

05/23/2023  12:03 PM    <DIR>          .
05/23/2023  12:03 PM    <DIR>          ..
05/23/2023  11:59 AM             1,472 ca.crt
05/23/2023  11:58 AM             1,884 ca.key
05/23/2023  12:03 PM             1,846 hoka-csr.pem
05/23/2023  12:01 PM             3,294 hoka.key
               4 File(s)          8,496 bytes
               2 Dir(s)  38,371,045,376 bytes free
```

2. Use the CSR to generate a self signed certificate using openssl by executing the following command: `openssl x509 -req -days 365 -in hoka-csr.pem -CA ca.crt -CAkey ca.key -CAcreateserial -out hoka.crt`

Convert HOKA Private Key to DER Format

To convert the private key to DER format using openssl execute the following command:

```
openssl pkcs8 -inform PEM -in hoka.key -topk8 -outform DER -out hoka.der -v1 PBE-SHA1-3DES -passout pass:changeit
```

Note: In the HOKA SSL example, `hoka.key` corresponds to the above `hoka.der`.

Handle Encoding

Content And Transfer Encoding

The HTTP protocol specifies how to send the request/response payload (the body) with a specific encoding. To guarantee that the receiver can understand the encoded stream, HTTP has specified headers for encoding : `content-encoding` , `transfer-encoding` and `accept-encoding` . The `HttpRequest` and `HttpResponse` classes uses encoding handlers stored in the `EncodingRegistry` to, respectively, decode and encode the payloads with the relevant handler (`ContentEncoding` or `TransferEnCoding`). For the response, the `accept-encoding` the header value is used to determine the available encoding with the highest quality (acceptance value).

By default, the registry contains the “identity” encoding handler and the “chunked” transfer-coding handlers.

Request And Response Encoding

When parsing the request, `HttpRequest` wraps the body with the appropriate decoder or, if not found, sends a “406 Not Acceptable” response. The body-parser will receive the wrapped (decoded) stream as input to not have to deal with encodings. The same for `HttpResponse` uses the encoder wrapper to write the response into the encoded stream sent to the socket. Also, when using an input stream with unknown length as the response’s data, the transfer encoding used to send the response is “chunked”; otherwise, it is “identity”. When using a String as the response data, use the `HttpResponse#setData(String, String)` to specify the encoding of the string (by default, `ISO-8859-1` is used).

URL Encoding

The percent-encoded special characters in the URI and in the query (parameters) are automatically decoded at parsing.

Session

HOKA provides tools to enable session management on the HTTP server.

Here is an example of how to use it.

```
// create a new session and store the user data in a session
final SessionHandler sessionHandler = new SessionHandler(new SecureRandom());
final Session session = this.sessionHandler.newSession();

//  for example from a login request handler
// ...  authenticate a user and store it user name into a session attribute
session.setAttribute("username", username);
// add a session cookie to the HttpResponse
response.addCookie("jsessionid", session.getId(), 0, false, true);


// from a protected request handler
// Get the session if from the cookie
```

(continues on next page)

(continued from previous page)

```
String sessionId = request.getCookie("jsessionid");
Session␣
↪session = this.session.getSession(sessionId); // get the session by it's id
// check if the user exists in the server session.
String username = (String)␣
↪session.getAttribute("username"); // access the username for example.
```

HOKA Configuration

The server can be configured by creating a property file in *src/main/resources* named *hoka.properties*

```
# Copyright 2021 MicroEJ Corp. All rights reserved.
# Use of this source code␣
↪is governed by a BSD-style license that can be found with this software.

# HOKA Server properties

# Use this property to set the logging level of the server.
# TRACE, DEBUG, INFO, WARN, ERROR, NONE
# the lower level activate all the others.
hoka.logger.level=INFO

# use this property to set a custom␣
↪logger. The custom logger must implement the interface ej.hoka.log.Logger
# if not␣
↪set HOKA use a SimpleLogger implementation that logs to the standard output
# Ensure that your logger is kept by␣
↪the Soar by adding it to *.types.list properties file in the app resources.
#hoka.logger.class=

# I/O buffer size used to read/write data from/to request/response
#hoka.buffer.size=4096
```

This section presents networking libraries.

The following schema shows the overall architecture and modules:

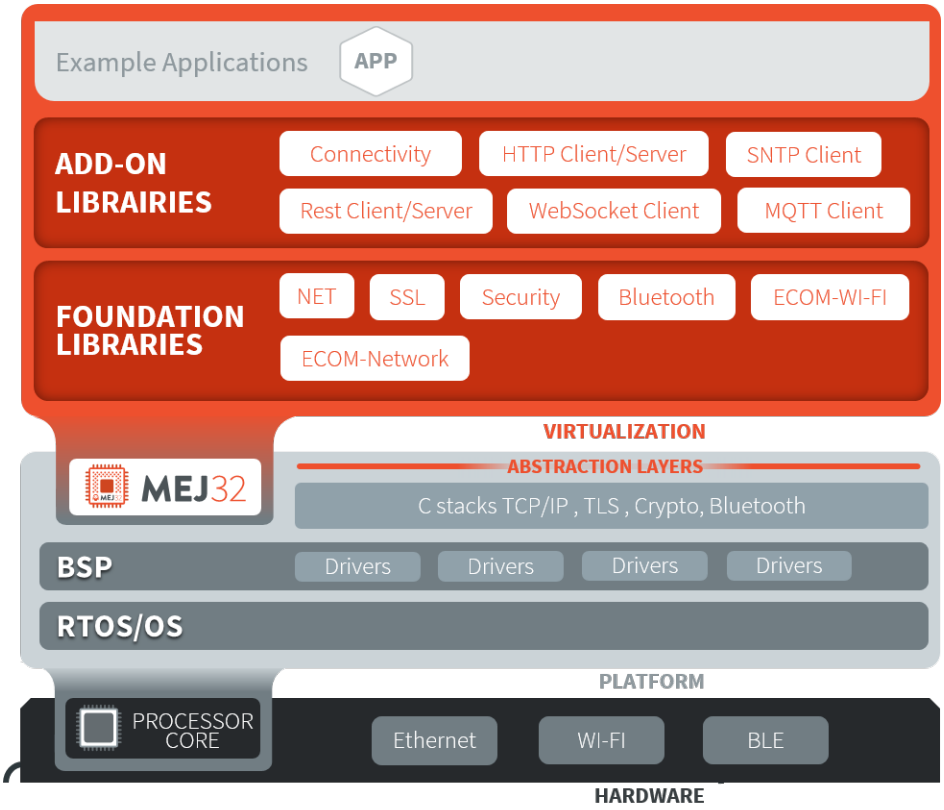


Fig. 32: Network Libraries Overview

- *Foundation Libraries*
- *Add-On Libraries*

5.14.4 Date and Time

Introduction

Java developers have long used the Date, Calendar and TimeZone classes for handling date and time. Java SE 8 introduced a more advanced and comprehensive Date and Time API that goes beyond simply replacing Date or Calendar. It provides a complete time model for applications.

There are many benefits of using the latest:

- Immutability: types are immutable, making thread-safe code easier to write and less prone to bugs (due to no mutable state).
- Improved API design: it offers an intuitive and developer-friendly design that better addresses the challenges of date and time manipulation. Application code is also easier to read and understand.
- Simplified date and time arithmetic: the API introduces methods for common date and time operations, simplifying tasks like adding or subtracting days, months, or years.
- Precision: it provides more precise representations for date and time values, including support for nanoseconds, which is important for applications requiring high precision.

- Comprehensive time model: it introduces new classes that deal with different concepts of time such as date without a time or time without a date, durations or periods.

  In general, it's a good practice to use the `Java Time API` when dealing with date, time, and time zone-related operations because of its convenient features and extensive capabilities. Yet, for straightforward timestamp handling or lightweight applications, `System.currentTimeMillis()` can be adequate. One aspect to keep in mind is that the Time API offers better readability and advanced operations, which might be missing when using timestamp manipulation or older APIs.

Overview

The library introduces different classes for date, time, date-time, and variations for offset and time zone. While this may seem like a lot of classes, most applications can start with only these types:

- Instant: an instantaneous point on the timeline. It can be used to store timestamps of application events.
- LocalDate: stores a date without a specific time or time zone, like `2023-09-26`.
- LocalTime: stores a time without a specific date or time zone, like `15:30`.
- LocalDateTime: stores both a date and time without a specific time zone, like `2023-09-26T15:30`. It combines LocalDate and LocalTime.
- ZonedDateTime: stores both a date and time, including a time zone. This is handy for performing precise date and time calculations while considering the time zone.
- Duration: a duration of time, measured in hours, minutes, seconds, and nanoseconds.
- Period: a duration of time in terms of years, months, and days.

Note: Working with a time zone can make calculations more complex. In many cases, the application can only work with LocalDate, LocalTime, and Instant, and then add the time zone at the user interface (UI) level.

The API has many methods, but it remains easy to handle because it sticks to consistent method prefixes:

- `of` : static factory method.
- `get` : gets a value.
- `is` : checks if some condition is true.
- `with` : equivalent to a setter for immutable objects, returns a copy with the specified argument set.
- `plus` : adds an amount to an object.
- `minus` : subtracts an amount from an object.
- `to` : converts this object to another type.
- `at` : combines this object with another.

Usage

The Date and Time API is provided as an Add-on Library.

To use the time library, add the following to the project build file:

Gradle (build.gradle.kts)

MMM (module.ivy)

```
implementation("ej.library.eclasspath:time:1.0.0")
```

```
<dependency org="ej.library.eclasspath" name="time" rev="1.0.0"/>
```

Examples

This section presents a series of small, focused examples that demonstrate various aspects of the Java Date and Time API.

Instant

The Instant class is the closest equivalent of Date. It represents a specific instant in time.

```
// Creating instants
Instant now = Instant.now(); // now
Instant someInstant␣
↪= Instant.ofEpochSecond(1695732445L); // September 26, 2023 12:47:25 PM

// Displaying
System.out.println("Seconds elapsed since epoch " + now.getEpochSecond());

// Chaining operations on instants
long␣
↪secondsUntil = someInstant.plusSeconds(10).until(now, ChronoUnit.SECONDS);
System.out.println(
↪"Amount of time until another instant in seconds: " + secondsUntil);
```

LocalDate

LocalDate stores a date without a time. It is called “local” because it isn’t associated with any specific time zone, similar to a wall clock. It simplifies date operations by dealing only with dates, making it suitable for scenarios not requiring time zone concerns (e.g., booking systems, calendars, date validation, etc.).

```
// Creating LocalDate instances
LocalDate today = LocalDate.now(); // Current date
LocalDate specificDate = LocalDate.of(2023, Month.JULY, 15); // July 15, 2023

// Displaying LocalDate instances
System.out.println("Today's Date: " + today);
System.out.println("Specific Date: " + specificDate);
```

(continues on next page)

(continued from previous page)

```
// Performing operations
LocalDate futureDate = today.plusDays(30); // Adding 30 days to today
LocalDate pastDate = today.minusMonths(2); // Subtracting 2 months from today

// Displaying the results of operations
System.out.println("Date 30 days from today: " + futureDate);
System.out.println("Date 2 months ago from today: " + pastDate);

// Comparing LocalDate instances
boolean isAfter␣
↪= specificDate.isAfter(today); // Check if specificDate is after today

// Displaying comparison results
System.out.println("Is specificDate after today? " + isAfter);
```

LocalTime

LocalTime stores a particular time of day, focusing only on the time (hour, minute, second, nanosecond), and doesn't include date or time zone details. Useful when you only need to handle time values without dates or time zones (e.g., scheduling events like alarms, stopwatch and timers, event timing, etc.).

```
// Creating LocalTime instances
LocalTime now = LocalTime.now(); // Current time
LocalTime specificTime = LocalTime.of(14, 30); // 2:30 PM

// Displaying LocalTime instances
System.out.println("Current Time: " + now);
System.out.println("Specific Time: " + specificTime);

// Performing operations
LocalTime␣
↪futureTime = now.plusHours(3); // Adding 3 hours to the current time
LocalTime pastTime = now.minusMinutes(15).minusSeconds(29);
↪ // Subtracting 15 minutes and 29 seconds from the current time

// Displaying the results of operations
System.out.println("Time 3 hours from now: " + futureTime);
System.out.println("Time 15 minutes ago: " + pastTime);

// Displaying time fields
System.out.println("Hour: " + now.getHour());
System.out.println("Minute: " + now.getMinute());
System.out.println("Second: " + now.getSecond());
```

LocalDateTime

LocalDateTime combines both date and time components and provides a precise timestamp. This makes it suitable for scenarios where you need to work with both date and time information, but without considering time zone conversions (e.g., timestamping, user interfaces, etc.).

```
// Creating LocalDateTime instances
LocalDateTime now = LocalDateTime.now(); // Current date and time
LocalDateTime specificDateTime␣
↪= LocalDateTime.of(2023, Month.JULY, 15, 14, 30); // July 15, 2023, 2:30 PM

// Displaying LocalDateTime instances
System.out.println("Current Date and Time: " + now);
System.out.println("Specific Date and Time: " + specificDateTime);

// Performing operations
LocalDateTime futureDateTime␣
↪= now.plusDays(30).plusHours(3); // Adding 30 days and 3 hours to now
LocalDateTime pastDateTime = now.minusMonths(2).
↪minusMinutes(15); // Subtracting 2 months and 15 minutes from
  ␣
↪                                                                    // now
// Displaying the results of operations
System.out.
↪println("Date and Time 30 days and 3 hours from now: " + futureDateTime);
System.out.println(
↪"Date and Time 2 months and 15 minutes ago from now: " + pastDateTime);

// Displaying date and time fields
System.out.println("Year: " + now.getYear());
System.out.println("Month: " + now.getMonth());
System.out.println("Day of Month: " + now.getDayOfMonth());
System.out.println("Hour: " + now.getHour());
System.out.println("Minute: " + now.getMinute());
System.out.println("Second: " + now.getSecond());
System.out.println("Day of Year: " + now.get(ChronoField.DAY_OF_YEAR));
System.out.println("Day of Week: " + now.get(ChronoField.DAY_OF_WEEK));

// Displaying comparison results
System.out.println("Is specificDateTime␣
↪after current date and time? " + specificDateTime.isAfter(now));
```

Duration

Duration represents a duration of time, typically measured in hours, minutes, seconds, and nanoseconds. It is used to calculate and work with time intervals, such as the amount of time between two points in time or the duration of an event. It is suitable for tasks involving precise timing, such as measuring time elapsed or setting timeouts.

```
// Creating Duration instances
Duration fiveHours = Duration.ofHours(5); // Duration of 5 hours
Duration thirtyMinutes = Duration.ofMinutes(30); // Duration of 30 minutes
```

(continues on next page)

(continued from previous page)

```
Duration twoSeconds = Duration.ofSeconds(2); // Duration of 2 seconds

// Displaying Duration instances
System.out.println("5 Hours: " + fiveHours);
System.out.println("30 Minutes: " + thirtyMinutes);
System.out.println("2 Seconds: " + twoSeconds);

// Performing operations
Duration combinedDuration␣
↪= fiveHours.plus(thirtyMinutes).plusSeconds(10); // Adding durations
Duration␣
↪subtractedDuration = fiveHours.minus(twoSeconds); // Subtracting durations

// Displaying the results of operations
System.out.println("Combined Duration: " + combinedDuration);
System.out.println("Subtracted Duration: " + subtractedDuration);

// Displaying duration fields
System.out.println("Hours: " + combinedDuration.toHours());
System.out.println("Minutes: " + combinedDuration.toMinutes());
System.out.println("Seconds: " + combinedDuration.getSeconds());

// Comparing Duration instances
boolean isLonger = fiveHours.compareTo(thirtyMinutes)␣
↪> 0; // Check if fiveHours is longer than thirtyMinutes
boolean isEqual = fiveHours.
↪equals(Duration.ofHours(5)); // Check if fiveHours is equal to 5 hours

// Displaying comparison results
System.out.println("Is fiveHours longer than thirtyMinutes? " + isLonger);
System.out.println("Is fiveHours equal to 5 hours? " + isEqual);
```

Period

Period represents a duration of time in terms of years, months, and days. It is primarily concerned with human-centric time measurements, like the length of a month or a year. It is well-suited for measuring time intervals within a calendar context. For example, it can represent periods such as 2 years, 3 months, and 5 days.

```
// Creating LocalDate instances
LocalDate date1 = LocalDate.of(2021, 6, 15); // June 15, 2021
LocalDate date2 = LocalDate.of(2023, 9, 25); // September 25, 2023

// Calculating the period between two dates
Period period = Period.between(date1, date2);

// Displaying the period
System.
↪out.println("Period between " + date1 + " and " + date2 + ": " + period);

// Displaying period fields
```

(continues on next page)

(continued from previous page)

```
System.out.println("Years: " + period.getYears());
System.out.println("Months: " + period.getMonths());
System.out.println("Days: " + period.getDays());

// Creating Period instances using factory methods
Period customPeriod = Period.of(2, 3, 5); // 2 years, 3 months, and 5 days

// Displaying the custom period
System.out.println("Custom Period: " + customPeriod);

// Performing operations on periods
Period addedPeriod = period.plus(customPeriod); // Adding periods
Period subtractedPeriod = period.minus(customPeriod); // Subtracting periods

// Displaying the results of operations
System.out.println("Added Period: " + addedPeriod);
System.out.println("Subtracted Period: " + subtractedPeriod);

// Comparing Period instances
boolean isEqual = customPeriod.equals(Period.of(2,
↪ 3, 5)); // Check if customPeriod is equal to 2 years, 3 months, and 5 days

// Displaying comparison results
System.out.println(
↪"Is customPeriod equal to 2 years, 3 months, and 5 days? " + isEqual);
```

Time Zone Support

The library relies on a time zone rules provider to supply the rules and data required for managing time zones. The zone rules provider offers information about how time zones are defined, including their offsets from Coordinated Universal Time (UTC), daylight saving time (DST) rules and historical changes.

The Time API introduces multiple types for time zone management:

- ZoneId : represents a time zone identifier (e.g., `Africa/Johannesburg`).
- ZoneOffset : represents a fixed time zone offset from Coordinated Universal Time (UTC).
- ZonedDateTime : represents the local time for a specific location.
- ZoneRulesProvider : foundation for supplying time zone rules and data and implementing custom time zone rules providers.

All the zone-aware classes of the library rely on the underlying time zone rules provider to supply accurate information about the time zone.

Java SE 8 and higher have a default provider that delivers zone rules for the time zones defined by IANADatabase. The `time` library does not use this provider as the default (see *Restrictions*). Instead, the library comes with a default provider which is very lightweight and designed to handle only the time zone rules for the “GMT” (Greenwich Mean Time) zone. This is suitable for operations on dates and times that do not depend on time zone considerations. Any attempt to use another zone ID will throw a ZoneRulesException because the ID is unknown. For example,

```
// Displaying available time zones - will list a single item: "GMT"
Set<String> timeZones = ZoneId.getAvailableZoneIds();
for (String timeZone : timeZones) {
    System.out.println(timeZone);
}

// Creating ZonedDateTime instance - will throw a ZoneRulesException
ZonedDateTime specificDateTime = ZonedDateTime.of(2023, 7, 15, 14,
↪ 30, 0, 0, ZoneId.of("Europe/Dublin")); // July 15, 2023, 2:30 PM in Dublin

//␣
↪Creating ZoneId instance from a region ID - will throw a ZoneRulesException
ZoneId tokyoTimeZone = ZoneId.of("Asia/Tokyo");
```

However, you can define a custom default provider for loading time zone rules. First, create a class that extends ZoneRulesProvider and defines custom zone rules like in the example after:

```
public class CustomZoneRulesProvider extends ZoneRulesProvider {

    @Override
    protected Set<String> provideZoneIds() {
        Set<String> set = new HashSet<>(1);
        set.add("CustomZone");
        return set;
    }

    @Override
    protected ZoneRules provideRules(String zoneId, boolean forCaching) {
        if ("CustomZone".equals(zoneId)) {
            // this custom zone has a fixed offset (+02:00)
            return ZoneRules.of(ZoneOffset.ofHours(2));
        }
        throw new ZoneRulesException("Unknown zone ID");
    }

    @Override
  ␣
↪ protected NavigableMap<String, ZoneRules> provideVersions(String zoneId) {
        throw new ZoneRulesException(
↪"No version history available for this zone ID " + zoneId);
    }
}
```

To make this class the default provider, set the constant `java.time.zone.DefaultZoneRulesProvider` to be the Full Qualified name of the custom provider class.

Here is an example of a `xxx.constants.list` file with the constant in an application:

```
java.time.zone.DefaultZoneRulesProvider=com.mycompany.CustomZoneRulesProvider
```

Note: Custom time zone rules providers are usually made for specific needs or to work with non-standard data sources.

Migration Guide

If you're using the legacy date and time classes (Date, Calendar), it's a great time to consider migrating to the new API. This small migration guide will help you transition from the old time API to the Java Date and Time API (`java.time`). It covers some common date and time operations and demonstrates how to perform them using both approaches.

Displaying the Current Date

Legacy Time API

New Time API

```
// Create a Calendar instance representing the current date and time
Calendar calendar = Calendar.getInstance();

// Get date components from the Calendar
int year = calendar.get(Calendar.YEAR);
int month = calendar.get(Calendar.MONTH) + 1; // Months are 0-based
int day = calendar.get(Calendar.DAY_OF_MONTH);

// Display the date
System.out.println("Current Date: " + year + "-" + month + "-" + day);
```

```
// Get the current date using LocalDate
LocalDate currentDate = LocalDate.now();

// Display the date
System.out.println("Current Date: " + currentDate);
```

Calculating a Timestamp from a Date

Legacy Time API

New Time API

```
// Create a Calendar instance
Calendar calendar = Calendar.getInstance();
calendar.set(2023, 10, 06, 15, 27, 30);     // November 06, 2023 3:27:30 PM
long timeInMillis = calendar.getTimeInMillis();
```

```
// Create a LocalDateTime instance with the desired date and time
LocalDateTime localDateTime = LocalDateTime.of(2023, 10, 06, 15, 27, 30);

// Convert LocalDateTime to a timestamp from Epoch
long timeInMillis = localDateTime.toInstant(ZoneOffset.UTC).toEpochMilli();
```

Calculating Date and Time Differences

Legacy Time API

New Time API

```
public long computeDifference(Date date1, Date date2){
    return date1.getTime() - date2.getTime();
}
```

```
public long computeDifference(LocalDateTime date1, LocalDateTime date2){
    return Duration.between(date1, date2).toMillis();
}
```

Calculating the Day of the Week

Legacy Time API

New Time API

```
// Create a Calendar instance
Calendar calendar = Calendar.getInstance();

// Set a date (e.g., October 15, 2023)
calendar.set(2023, Calendar.OCTOBER, 15);

// Get the␣
↪day of the week as an integer (1 = Sunday, 2 = Monday, ..., 7 = Saturday)
int dayOfWeek = calendar.get(Calendar.DAY_OF_WEEK);
```

```
// Create a LocalDate instance for a specific date (October 15, 2023)
LocalDate date = LocalDate.of(2023, 10, 15);

// Get the day of the week as an enum value (DayOfWeek)
DayOfWeek dayOfWeek = date.getDayOfWeek();
```

Handling Time Zones

Legacy Time API

New Time API

```
TimeZone timeZone = TimeZone.getTimeZone("America/New_York");
Calendar calendar = Calendar.getInstance(timeZone);
Date dateInNewYork = calendar.getTime();
```

```
ZoneId zoneId = ZoneId.of("America/New_York");
ZonedDateTime zonedDateTime = ZonedDateTime.now(zoneId);
```

Restrictions

The library's goal is to offer Application developers an API that closely mirrors the one found in Java SE 8. However, we had to make the library compatible with both pre-Java 8 features and the constraints found in embedded devices. Here are the items where the backport differs from its Java 8 counterpart:

- Non-ISO chronologies are not present (*Hijrah*, *Japanese*, *Minguo*, *ThaiBuddhist*). The overwhelming majority of applications use the ISO calendar system. Applications still have the option to introduce their own chronologies.
- No formatting or parsing methods (methods `parse`, `format`, `getDisplayName`, `ofLocale`).
- The default zone-rules provider does not use IANADatabase. This provider loads zone rules from a local TZDB database and it consumes a significant amount of RAM. We plan to add this support shortly.
- Removed the method `ZoneRulesProvider.registerProvider(ZoneRulesProvider provider)`. The unique provider is defined with the constant `java.time.zone.DefaultZoneRulesProvider`.
- Static methods in interfaces are not supported and were removed or moved (see below).
- Default methods in interfaces are not supported and were removed (pulled down in concrete types).
- Removed static methods `TemporalAdjusters.ofDateAdjuster(UnaryOperator<LocalDate> dateBasedAdjuster)` and `WeekFields.of(Locale locale)`.
- No overflow checks on calculations (removed `throws ArithmeticException` when relevant). Excessively checking for overflow in all calculations can impact performance negatively.
- No null checks on method arguments. Developers are encouraged to use the *Null Analysis* tool to detect null access and adhere to the API javadoc specifications.

Note: For a comprehensive list of restrictions, refer to the `README` of the module. If some of the restrictions listed above are highly limiting and necessary for your application, please contact your MicroEJ sales representative or *our support team*.

Static Interface Methods

- `ChronoLocalDate.from(TemporalAccessor)` : *removed*
- `ChronoLocalDate.timeLineOrder()` : use `LocalDate.timeLineOrder()` instead
- `ChronoLocalDateTime.from(TemporalAccessor)` : *removed*
- `ChronoLocalDateTime.timeLineOrder()` : use `LocalDateTime.timeLineOrder()` instead
- `ChronoZonedDateTime.from(TemporalAccessor)` : *removed*
- `ChronoZonedDateTime.timeLineOrder()` : use `ZonedDateTime.timeLineOrder()` instead
- `ChronoPeriod.between(ChronoLocalDate, ChronoLocalDate)` : *removed*
- `Chronology.from(TemporalAccessor)` : use `AbstractChronology.from(TemporalAccessor)` instead
- `Chronology.getAvailableChronologies()` : use `AbstractChronology.getAvailableChronologies()` instead

- `Chronology.of(String)` : use `AbstractChronology.of(String)` instead
- `Chronology.ofLocale(Locale)` : *removed*

5.14.5 Event Queue

Principle

The Event Queue Foundation Library provides an asynchronous communication interface between the native world and the Java world based on events.

Functional Description

Overview

The Event Queue Foundation Library allows users to send events from the native world to the Java world. It is composed of a Java API that provides mechanisms to register specific event notifications and a C API that allows someone to send events in the queue.

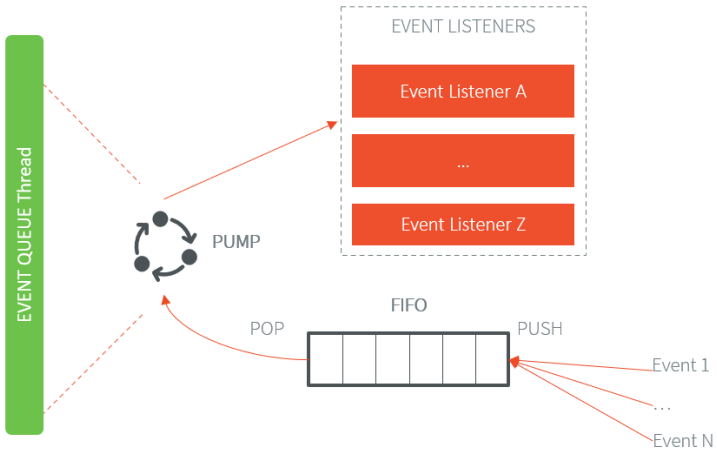


Fig. 33: Event Queue Overview

A FIFO mechanism is implemented on the native side and is system specific. The user can offer events to this FIFO by using the C or the Java API.

Event notifications are handled using event listeners (Observer design pattern). The application code has to register event listeners to be notified when new events are coming.

Then the Event Pump automatically retrieves new events pushed in the FIFO and notifies the event listeners.

Architecture

The Event Queue Foundation Library uses a dedicated Java thread to forward and process events. Application event listener's calls are done in the context of the Event Queue thread.

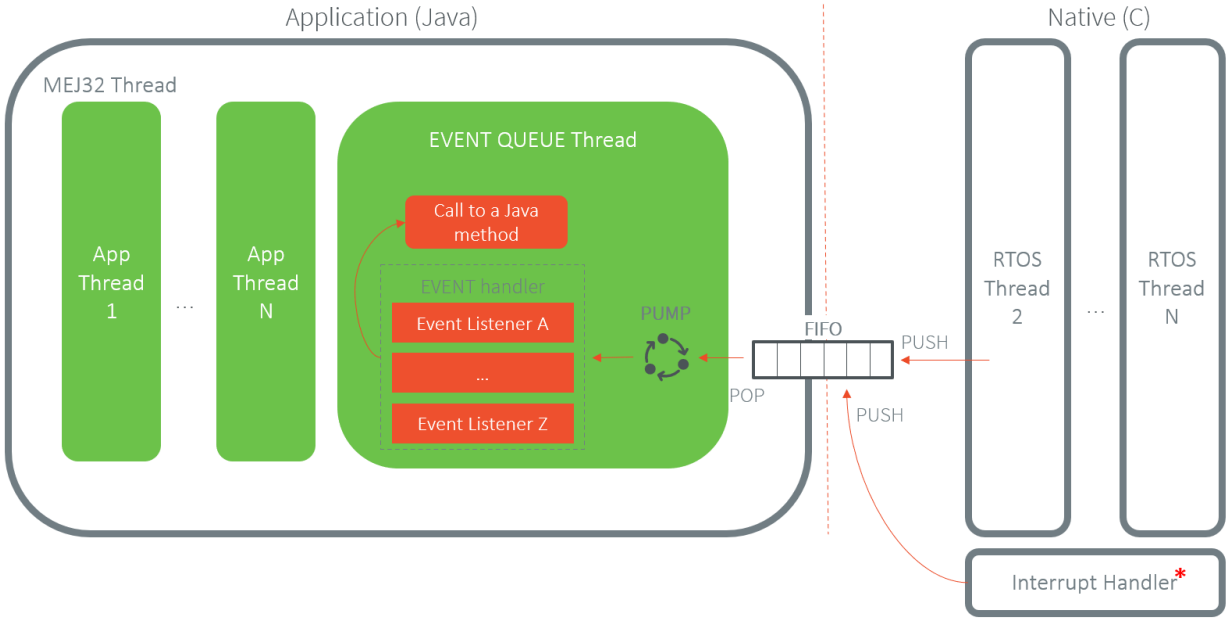


Fig. 34: Event Queue Architecture

Events reading operations are done using the SNI mechanism. Event Queue Java thread is suspended when the events FIFO is empty and resumed when a new event is sent.

Note: To support sending events from the Interrupt Handler, the VEE Port must provide a compatible implementation.

Event format

An event is composed of a type and, optionally, data. The type identifies the listener that will handle the event. The data is application specific and passed to the listener as a raw byte array.

The items stored in the FIFO buffer are integers (4 bytes). There are two kinds of events that can be sent over the Event Queue:

- Standard event: an event with data that fits on 24 bits. The event is stored in the FIFO as a single 4 bytes item.
- Extended event: an event with data that does not fit on 24 bits. The event is stored in the FIFO as multiple 4 bytes items.

```
+---
↪-----------+----------+-----------------------------------------------------+
| Extended␣
```

(continues on next page)

(continued from previous page)

```
↪(1) | Type (7) | Data (if Extended==0), Length (if Extended==1) (24) |
+---
↪-----------+----------+-----------------------------------------------------+
...
+---
↪-------------------------------------------------------------------------+
|                  ␣
↪Extended Data for extended events (32)                    | (Length bytes)
+---
↪-------------------------------------------------------------------------+
```

Format explanation:

- *Extended* (1 bit): event kind flag (0 for standard event, 1 for extended event).
- *Type* (7 bits): event type, which allows to find the corresponding event queue listener.
- *Length* (24 bits): length of the data in bytes (for extended events only).
- *Data* (24 bits): standard event data (for standard events only).
- *Extended data* (`Length` bytes): extended event data (for extended events only).

Event Queue listener

An application can register listeners to the Event Queue. Each listener is registered for a specific event type. The same listener can be registered several times for different event types, but each event type can only have one listener.

When the queue receives an event from the FIFO, it will get the event type and check if it is an extended event. Then it will check if a listener is registered for this event type. If so, it will call its handle method depending on the extended event flag. It will call the default listener if no listener corresponds to the event type.

You can create your Event Queue listener by implementing the `EventQueueListener` interface. It contains two methods that are used to handle standard and extended events.

Before registering your listener, you must get a valid unique type using the `getNewType()` method from the `EventQueue` class. Then you can register your listener using the `registerListener(EventQueueListener listener, int type)` method from the `EventQueue` class.

The unique type your listener uses could be stored on the Java world and passed/stored to the C world. One way to do it is to create a native method that sends the event type to the C world during the initialization phase.

To set the default listener, you must use `EventQueue.setDefaultListener(EventQueueListener listener)`.

For example:

```
public static int eventType;

public static void main(String[] args) throws InterruptedException {
   EventQueue eventQueue = EventQueue.getInstance();
```

(continues on next page)

(continued from previous page)

```
   // Get the unique type to register your listener.
   /
↪/ eventType must be stored if you want to offer an event from the Java API.
   eventType = eventQueue.getNewType();

   // Create and register a listener.
   eventQueue.registerListener(new ExampleListener(), eventType);

   // Send eventType to the C world.
   initialize(eventType);
}

/**
* This native␣
↪method will take the event type as an entry and store it in the C world.
*/
public static native void initialize(int type);
```

Standard event

Standard events are events with data that can be stored on 24 bits.

```
+-------+----------+-----------+
| 0 (1) | Type (7) | Data (24) |
+-------+----------+-----------+
```

The first bit equals 0, indicating that this is a standard event.

Then there is the event type stored on 7 bits.

To finish, there is the data that you want to send to the application event listener. It is stored on 24 bits.

Offer the event

There are two ways to send a standard event through the Event Queue: from the C API or the Java API.

From C API

To send a standard event through the Event Queue using the C API, you must use the `LLEVENT_offerEvent(int32_t type, int32_t data)` method from `LLEVENT.h`.

For example:

```
// Assuming that event_type has been passed from␣
↪the Java world through a native method after registering your listener.
int type = event_type;
int data = 12;

LLEVENT_offerEvent(type, data);
```

From Java API

To send a standard event through the Event Queue using the Java API, you must use the `offerEvent(int type, int data)` method from the `EventQueue` class.

For example:

```
EventQueue eventQueue = EventQueue.getInstance();

// Assuming that eventType␣
↪has been stored in the Java world when you registered the listener.
int type = eventType;
int data = 12;

eventQueue.offerEvent(type, data);
```

Handle the event

To handle a standard event, you must implement your listener `handleEvent(int type, int data)` method. You can process the data received by the Event Queue in this method.

First, you have to register your listener as explained *Event Queue listener* in section.

For example:

```
EventQueue queue = EventQueue.getInstance();
int type = queue.getNewType();
initialize(type);
queue.registerListener(type, new EventQueueListener() {
   @Override
   public void handleEvent(int type, int data) {
      System.out.println("My data is equal to: " + data);
   }
   @Override
   public␣
↪void handleExtendedEvent(int type, EventDataReader eventDataReader) {
      throw new RuntimeException();
   }
});
```

Extended event

Extended events are events with data that can not be stored on 24 bits.

```
+-------+----------+-------------+
| 1 (1) | Type (7) | Length (24) |
+-------+----------+-------------+
...
+--------------------------------+
|       Extended Data  (32)      | (Length bytes)
+--------------------------------+
```

On the first 32 bits of the events, you will have:

- First bit is equal to 1, saying that this is an extended event,
- The event type stored on 7 bits,
- The length of the data following the header in bytes stored on 24 bits.

  Then you will have the data. The number of bytes of the data depends on the length.

Data Alignment

To process the data from an extended event, you will use an `EventDataReader` object. You will see it more in detail in the *Handle the event* section.

With `EventDataReader` API, there are two ways to read an event:

- Read the data with `read(byte[] b, int off, int len)` or `readFully(byte[] b)` methods.
  - You will get the data in a byte array and can process it on your own in your `handleExtendedEvent(int type, EventDataReader eventDataReader)` method.
- Read the data with the methods related to the primitive types such as `readBoolean()` or `readByte()`.
  - The reader is designed to parse C-struct data.
  - To use the methods, **your fields must follow this alignment:**
    * A **boolean** (1 byte) will be 1-byte aligned.
    * A **byte** (1 byte) will be 1-byte aligned.
    * A **char** (2 bytes) will be 2-byte aligned.
    * A **double** (8 bytes) will be 8-byte aligned.
    * A **float** (4 bytes) will be 4-byte aligned.
    * An **int** (4 bytes) will be 4-byte aligned.
    * A **long** (8 bytes) will be 8-byte aligned.
    * A **short** (2 bytes) will be 2-byte aligned.
    * An **unsigned byte** (1 byte) will be 1-byte aligned.
    * A **unsigned short** (2 bytes) will be 2-byte aligned.

Offer the event

There are two ways to send an extended event through the Event Queue: from the C API or the Java API.

From C API

To send an extended event through the Event Queue using the C API, you have to use the `LLEVENT_offerExtendedEvent(int32_t type, void* data, int32_t data_length)` method from `LLEVENT.h`.

For example:

```
struct accelerometer_data {
   int x;
   int y;
   int z;
}

// Assuming that event_type has been passed from␣
↪the Java world through a native method after registering your listener.
int type = event_type;

struct accelerometer_data data;
data.x = 42;
data.y = 72;
data.z = 21;

LLEVENT_offerExtendedEvent(type, (void*)&data, sizeof(data));
```

From Java API

To send an extended event through the Event Queue using the Java API, you must use the `offerExtendedEvent(int type, byte[] data)` method from the EventQueue API.

For example:

```
EventQueue eventQueue = EventQueue.getInstance();

// Assuming that eventType␣
↪has been stored in the Java world when you registered the listener.
int type = eventType;

// Array of 3 integers. Each integer is stored in 4 bytes.
byte[] accelerometerData = new byte[3*4];

// Write integers into the byte array using ByteArray API.
ByteArray.writeInt(accelerometerData, 0, 42);
ByteArray.writeInt(accelerometerData, 4, 72);
ByteArray.writeInt(accelerometerData, 8, 21);

eventQueue.offerExtendedEvent(type, accelerometerData);
```

Handle the event

To handle an extended event, you must implement your listener's `handleExtendedEvent(int type, EventDataReader eventDataReader)` method. You can process the data received by the Event Queue on this method.

It provides an EventDataReader that contains the methods needed to read the data of an extended event.

First, you have to register your listener as explained *Event Queue listener* in section.

For example:

```
EventQueue queue = EventQueue.getInstance();
int type = queue.getNewType();
initialize(type);
queue.registerListener(type, new EventQueueListener() {
   @Override
   public void handleEvent(int type, int data) {
      throw new RuntimeException();
   }
   @Override
   public␣
↪void handleExtendedEvent(int type, EventDataReader eventDataReader) {
      int x = 0;
      int y = 0;
      int z = 0;
      try {
         x = eventDataReader.readInt();
         y = eventDataReader.readInt();
         z = eventDataReader.readInt();
      } catch (IOException e) {
         System.out.println("IOException␣
↪while reading accelerometer values from the EventDataReader.");
      }
      System.out.println(
↪"Accelerometer values: X = " + x + ", Y = " + y + ", Z = " + z + ".");
   }
});
```

Mock the Event Queue

To simulate event that are normally sent through the C API, use the Event Queue Mock API from your mock.

The Event Queue Mock API dependency must be added to the project build file of your MicroEJ Mock project.

```
<dependency org="com.microej.
↪pack.event" name="event-pack" rev="2.0.0" conf="provided->mockAPI"/>
```

It provides two methods:

- `EventQueueMock.offerEvent(int type, int data)` is the equivalent of `LLEVENT_offerEvent(int32_t type, int32_t data)` method from `LLEVENT.h`.

- `EventQueueMock.offerExtendedEvent(int type, byte[] data, int dataLength)` is the equivalent of `LLEVENT_offerExtendedEvent(int32_t type, void* data, int32_t data_length)` method from `LLEVENT.h`.

  Example of use:

```
// Assuming that event_type has been passed from␣
↪your Application through a native method after registering your listener.
int type = event_type;
int data = 12;

EventQueueMock.offerEvent(type, data);
```

Use

The Event Queue API Module must be added to the project build file of the MicroEJ Application project to use the Event Queue Foundation Library.

Gradle (build.gradle.kts)

MMM (module.ivy)

```
implementation("ej.api:event:2.0.0")
```

```
<dependency org="ej.api" name="event" rev="2.0.0"/>
```

To use this API, your VEE Port must implement a compatible version. Please refer to the *VEE Porting Guide* to port the Event Queue for your project.

5.14.6 JavaScript

MicroEJ allows to develop parts of an application in JavaScript. Basically, a MicroEJ Application boots in Java, then it initializes the JavaScript runtime to run a mix of Java and JavaScript code.

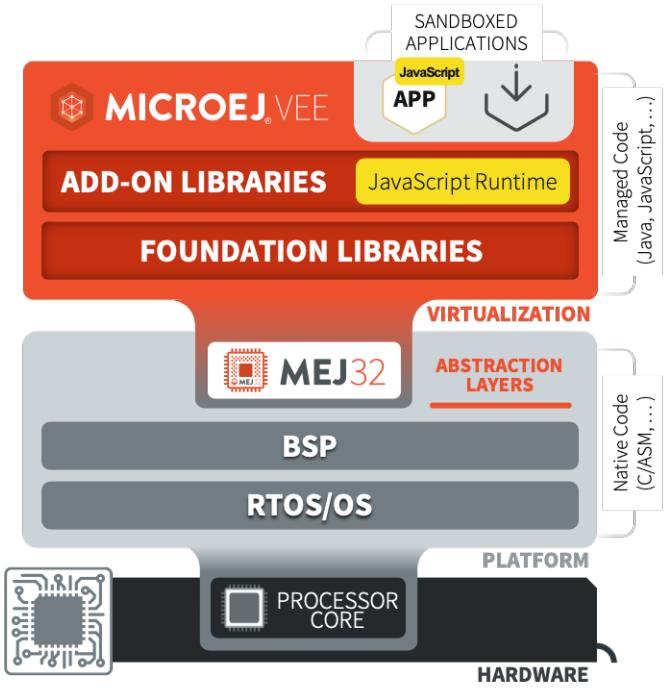


Fig. 35: MicroEJ JavaScript Overview

It supports the ECMAScript 5.1 specification, with *some limitations*. You can start playing with it by following the *Getting Started* page.

Getting Started

Let's walk through the steps required to use Javascript in your MicroEJ application:

- install the *MMM CLI (Command Line Interface)*
- create your Standalone Application project with the `init` command:

```
mmm␣
↪init -Dskeleton.org=com.is2t.easyant.skeletons -Dskeleton.module=firmware-
↪singleapp -Dskeleton.rev=1.1.12 -Dproject.org=com.mycompany -Dproject.
↪module=myproject -Dproject.rev=1.0.0 -Dskeleton.target.dir=myproject
```

Adapt the properties values to your need. See the *MMM CLI init command documentation* for more details.

Javascript is supported in the following *Module Natures page*: - *Add-On Library*, - *Standalone Application*, - *Sandboxed Application*.

- add the js dependency in the build file:

Gradle (build.gradle.kts)

MMM (module.ivy)

```
implementation("com.microej.library.runtime:js:0.13.0")
```

```
<dependency org="com.microej.library.runtime" name="js" rev="0.13.0"/>
```

- add the following lines in your application main class:

```
import com.microej.js.JsErrorWrapper;
import com.microej.js.JsCode;
import com.microej.js.JsRuntime;

...

JsCode.initJs();
JsRuntime.ENGINE.runOneJob();
JsRuntime.stop();
```

- create a file named `hello.js` in the folder `src/main/js` with the following content:

```
function hello() {
    var message = "MicroEJ Javascript application!";
    print("My first", message);
}

hello()
```

- follow the steps described in the *run command documentation*
- in a terminal, go to the folder containing the `module.ivy` file and build the project with the command:

```
mmm build
```

You should see the following message at the end of the build:

```
BUILD SUCCESSFUL

Total time: 20 seconds
```

- now that your application is built, you can run it in the simulator with the command:

```
mmm run
```

You should see the following output:

```
My first  MicroEJ Javascript application!
```

You can now go further by exploring the *capabilities of the MicroEJ Javascript engine* and discovering the *commands available in the CLI*.

Sources Management

JavaScript Sources Location

The JavaScript sources of an application must be located in the project folder `src/main/js`. All JavaScript files (`*.js`) found in this folder, at any level, are processed.

JavaScript Sources Load Order

When several JavaScript files are found in the sources folder, they are loaded in alphabetical order of their relative path. For example, the following source files:

```
src
└── main
    └── js
        ├── components
        │   ├── component1.js
        │   └── component2.js
        ├── ui
        │   └── widgets.js
        ├── app.js
        ├── feature1.js
        └── feature2.js
```

are loaded in this order:

1. app.js
2. components/component1.js
3. components/component2.js
4. feature1.js
5. feature2.js
6. ui/widgets.js

JavaScript Sources Load Scope

All the code of the JavaScript source files are loaded in the same scope. It means a variable or function defined in a source file can be used in another one if it has been loaded first. In this example:

Listing 8: src/main/js/lib.js

```
function sum(a, b) {
    return a + b;
}
```

Listing 9: src/main/js/main.js

```
print("5 + 3 = " + sum(5, 3));
```

the file `src/main/js/lib.js` is loaded before `src/main/js/main.js` so the function `sum` can be used in `src/main/js/main.js`.

JavaScript Sources Processing

JavaScript sources need to be processed before being executed. This processing is done in the following cases:

- when building the project with *MMM*.
- when developing the project in MicroEJ SDK. The MicroEJ SDK detects any change in JavaScript sources folder (addition/update/deletion) to trigger the processing.

Examples

This section is intended to provide a set of examples to cover most of the use cases when developing JavaScript applications with MicroEJ:

Simple Application

Note: Before trying this example, make sure you have the *MMM CLI (Command Line Interface)* installed.

This example shows the minimal code for a MicroEJ JavaScript application:

- create an *Add-On Library* project or a *Sandboxed Application* project
- add the MicroEJ JavaScript dependency in the build file of your project:

  Gradle (build.gradle.kts)

  MMM (module.ivy)

```
implementation("com.microej.library.runtime:js:0.13.0")
```

```
<dependency org="com.microej.library.runtime" name="js" rev="0.13.0"/>
```

- init the JavaScript code in your Java application with:

```
import com.microej.js.JsCode;

...

JsCode.init();
```

The class `com.microej.js.JsCode` is the Java class generated from the JavaScript sources.

- ask the MicroEJ JavaScript engine to start processing the job queue with:

```
import com.microej.js.JsRuntime;

...

JsRuntime.ENGINE.run();
```

This makes the JavaScript engine process the job queue forever until the program is stopped.

- create a file with the `js` extension in the `src/main/js` folder (for example `app.js`) with the following content:

```
print("My Simple Application");
```

- build and execute the application with the *MMM CLI*:

```
$ mmm build
$ mmm run
```

The message `My Simple Application` should be displayed.

Use a Java API in JavaScript

Note: Before trying this example, make sure you have the *MMM CLI (Command Line Interface)* installed.

It is also recommended to follow the *Getting Started* page and/or the *Simple Application* example before.

In this example the JavaScript code calls a Java API. The Java API can come from the application or from any library used by the application. Let's create it in the project for this example, in a class `Calculator` (`src/main/java/com/mycompany/Calculator.java`):

```
public class Calculator {
    public int sum(int x, int y) {
        return x + y;
    }

    public int mul(int x, int y) {
        return x * y;
    }
}
```

Then in the Java Main class of the application, add the glue to expose the *Calculator* Java API to the JavaScript code and init the JavaScript engine:

```
public static void main(String[] args) throws Exception {
    // Add the "getCalculator" function in the JavaScript global object
    JsRuntime.JS_GLOBAL_
↪OBJECT.put("getCalculator", JsRuntime.createFunction(new JsClosure() {
        @Override
        @Nullable
        public␣
↪Object invoke(Object thisBinding, int argsLength, Object... arguments) {
```

(continues on next page)

(continued from previous page)

```
            return new Calculator();
        }
    }), false);

    // Init the JavaScript code
    JsCode.initJs();
    // Start the JavaScript engine
    JsRuntime.ENGINE.run();
}
```

You can now call the API from the JavaScript code:

```
var calc = getCalculator();
print(calc.sum(1, 2));
print(calc.mul(5, 3));
```

As you can see, the methods of the Java API *Calculator* can be used directly from the JavaScript code.

Finally, build and execute the application with the *MMM CLI*:

```
$ mmm build
$ mmm run
```

The sum and multiply results should be displayed.

For more information about communication between Java and JavaScript please refer to the *Communication Between Java and JS* page.

Create a JavaScript API from Java

Note: Before trying this example, make sure you have the *MMM CLI (Command Line Interface)* installed.

It is also recommended to follow the *Getting Started* page and/or the *Simple Application* example before.

In this example a JavaScript API is exposed from Java. This can be useful when a specific API must be defined in JavaScript or when adapting an existing Java API to a JavaScript API.

Create a class `MyApiHostObject` (`src/main/java/com/mycompany/MyApiHostObject.java`):

```
public class MyApiHostObject extends JsObject {

        public MyApiHostObject(Object thisBinding) {

                this.put("count
↪", new DataPropertyDescriptor(JsRuntime.createFunction(new JsClosure() {
                        @Override
                        @Nullable
                        public Object␣
↪invoke(@Nullable Object thisBinding, int argsLength, Object... arguments) {
                                String data = (String) arguments[0];
```

(continues on next page)

(continued from previous page)

```
                                return Integer.valueOf(data.length());
                        }
                }));

        }
}
```

This class defines a JavaScript object using the MicroEJ JavaScript API by extending the class `JsObject`. It also defines a `count` method which accepts a String parameter and returns its length.

Then in the Java Main class of the application, add the glue to expose the *MyApi* object to the JavaScript code and init the JavaScript engine:

```
public static void main(String[] args) throws Exception {
        // Add the "MyApi" function in the JavaScript global object
        JsRuntime.
↪JS_GLOBAL_OBJECT.put("MyApi", JsRuntime.createFunction(new JsClosure() {
                @Override
                @Nullable
                public␣
↪Object invoke(Object thisBinding, int argsLength, Object... arguments) {
                        return new MyApiHostObject(thisBinding);
                }
        }), false);

        // Init the JavaScript code
        JsCode.initJs();
        // Start the JavaScript engine
        JsRuntime.ENGINE.run();
}
```

You can now call the new API from the JavaScript code:

```
var myApi = new MyApi();
print(myApi.count("Hello World!"));
```

Finally, build and execute the application with the *MMM CLI*:

```
$ mmm build
$ mmm run
```

The length of the string `Hello World!` (12) should be displayed.

For more information about communication between Java and JavaScript please refer to the *Communication Between Java and JS* page.

API

This page lists the API provided by the MicroEJ JavaScript engine.

Built-in Objects

The built-in objects are the API objects defined by the ECMAScript specification. This section lists all the JavaScript built-in objects and their support status in the MicroEJ JavaScript engine. For the complete reference about these built-in objects, consult the ECMAScript 5.1 specification.

For a description and usage examples of each method or property, consult a JavaScript documentation such as Mozilla Developer Reference.

Array

- Array (len)
- isArray (arg)
- toString ( )
- **[excluded]** toLocaleString ( )
- concat ([item1 [, item2 [, …]]])
- join (separator)
- pop ( )
- push ([item1 [, item2 [, …]]])
- reverse ( )
- shift ( )
- slice (start, end)
- sort (comparefn)
- **[excluded]** splice (start, deleteCount [, item1 [, item2 [, …]]])
- unshift ([item1 [, item2 [, …]]])
- indexOf (searchElement [, fromIndex])
- lastIndexOf (searchElement [, fromIndex])
- every (callbackfn [, thisArg])
- some (callbackfn [, thisArg])
- forEach (callbackfn [, thisArg])
- map (callbackfn [, thisArg])
- filter (callbackfn [, thisArg])
- **[excluded]** reduce (callbackfn [, initialValue])
- **[excluded]** reduceRight (callbackfn [, initialValue])
- length

Boolean

- Boolean (value)
- Boolean.prototype.toString ( )
- Boolean.prototype.valueOf ( )

Date

- **[excluded]**

Error

- **[excluded]**

Function

- **[excluded]** Function (p1, p2, … , pn, body)
- length
- **[excluded]** toString ( )
- apply (thisArg, argArray)
- call (thisArg [, arg1 [, arg2, …]])
- **[excluded]** bind (thisArg [, arg1 [, arg2, …]])
- [[Call]]
- [[Construct]]

Global

- NaN
- Infinity
- undefined
- **[excluded]** eval (x)
- parseInt (string , radix)
- parseFloat (string)
- isNaN (number)
- isFinite (number)
- **[excluded]** escape (string)
- **[excluded]** unescape (string)
- **[excluded]** decodeURI (encodedURI)
- **[excluded]** decodeURIComponent (encodedURIComponent)

- **[excluded]** encodeURI (uri)
- **[excluded]** encodeURIComponent (uriComponent)

JSON

- parse (text [, reviver])
- stringify (value , [replacer [, space]])

Math

- E
- LN10
- LN2
- LOG2E
- LOG10E
- PI
- SQRT1_2
- SQRT2
- abs (x)
- acos (x)
- asin (x)
- atan (x)
- atan2 (y, x)
- ceil (x)
- cos (x)
- exp (x)
- floor (x)
- log (x)
- max ([value1 [, value2 [, …]]])
- min ([value1 [, value2 [, …]]])
- pow (x, y)
- random ( )
- round (x)
- sin (x)
- sqrt (x)
- tan (x)

Number

- Number (value)
- MAX_VALUE
- MIN_VALUE
- NaN
- NEGATIVE_INFINITY
- POSITIVE_INFINITY
- **[excluded]** toString ([radix])
- **[excluded]** toLocaleString()
- valueOf ( )
- **[excluded]** toFixed (fractionDigits)
- **[excluded]** toExponential (fractionDigits)
- **[excluded]** toPrecision (precision)

Object

- Object ([value])
- Object.getPrototypeOf (O)
- Object.getOwnPropertyDescriptor (O, P)
- Object.getOwnPropertyNames (O)
- Object.create (O [, Properties])
- Object.defineProperty (O, P, Attributes)
- Object.defineProperties (O, Properties)
- **[excluded]** Object.seal (O)
- **[excluded]** Object.freeze (O)
- **[excluded]** Object.preventExtensions (O)
- Object.isSealed (O)
- Object.isFrozen (O)
- Object.isExtensible (O)
- Object.keys (O)
- toString ( )
- **[excluded]** toLocaleString ( )
- valueOf ( )
- hasOwnProperty (V)
- isPrototypeOf (V)
- propertyIsEnumerable (V)

Regex

- RegExp (pattern, flags)
- exec (string)
- test (string)
- toString ()

String

- String (value)
- fromCharCode ([char0 [, char1 [, …]]])
- toString ( )
- valueOf ( )
- charAt (pos)
- charCodeAt (pos)
- concat ([string1 [, string2 [, …]]])
- indexOf (searchString, position)
- lastIndexOf (searchString, position)
- **[excluded]** localeCompare (that)
- match (regexp)
- replace (searchValue, replaceValue)
- **[excluded]** search (regexp)
- slice (start, end)
- split (separator, limit)
- **[excluded]** substr (start [, length])
- substring (start, end)
- toLowerCase ( )
- **[excluded]** toLocaleLowerCase ( )
- toUpperCase ( )
- **[excluded]** toLocaleUpperCase ( )
- trim ( )
- length
- [[GetOwnProperty]] (P)

Host Objects

Host objects are not part of the ECMAScript specification, they are additional API provided by the MicroEJ JavaScript engine.

Global

setTimeout(function[, delay, arg1, arg2, ...])

- **description:** sets a timer which executes a function once the timer expires.
- **arguments:**
  - `function` : the function to execute when the delay expires.
  - `delay` (optional): the time in milliseconds that the timer must wait before executing the given function.
  - `arg1, arg2, ...` (optional): additional arguments passed to the given function.
- **returns:** the timer object. This object can be passed to the function `clearTimeout` to cancel the timer.

setInterval(function[, delay, arg1, arg2, ...])

- **description:** repeatedly calls a function, with a fixed time delay between each call.
- **arguments:**
  - `function` : the function to execute when the delay expires.
  - `delay` (optional): the time in milliseconds that the timer must wait between each execution of the given function.
  - `arg1, arg2, ...` (optional): additional arguments passed to the given function.
- **returns:** the timer object. This object can be passed to the function `clearInterval` to cancel the timer.

clearTimeout(timer)

- **description:** cancels the given timer created by a call to `setTimeout` .
- **arguments:**
  - `timer` : the timer to cancel.

clearInterval(timer)

- **description:** cancels the given timer created by a call to `setInterval`.
- **arguments:**
  - `timer`: the timer to cancel.

print([arg1, arg2, ...])

- **description:** prints the given arguments in the standard output. The arguments are concatenated and separated by a space. A new line is added at the end.
- **arguments:**
  - `arg1, arg2, ...`: the list of elements to print.

Communication Between Java and JS

The MicroEJ engine allows to communicate between Java and JavaScript: Java API can be used from JavaScript code and vice-versa.

JavaScript Engine

The JavaScript code is executed in a single-threaded engine, which means only one JavaScript statement is executed at a given time. Each piece of JavaScript code that must be executed is pushed in a job queue. It is up to the engine to manage the job queue and execute the jobs.

One consequence of this design is that Java code called from a JavaScript code must not be blocker. When calling a Java API from a Javascript code, in order to avoid blocking the JavaScript engine, the Java code must return as quick as possible. Otherwise the JavaScript engine is stuck and cannot execute other JavaScript jobs. It is especially harmfull when the Java operation takes time, for example for network or IO operations. In such a case, it is therefore recommended to execute it in a new thread and return immediately.

Another consequence of the JavaScript engine design is that JavaScript code must always be executed by the engine, by the single thread. Therefore, any call to a JavaScript code from a Java code must create a job and add it to the job queue.

Calling Java from JavaScript

The MicroEJ engine allows to expose Java objects or methods to the JavaScript code by using the engine API and creating the adequate JavaScript object.

Import Java Types from JavaScript

Java objects can be exposed to JavaScript using the `JavaImport` mechanism. It takes a Java fully qualified name as argument and returns an object that gives access to the constructors, static methods and static fields. All the classes from the project's classpath can be imported (project's own classes and its dependencies).

For instance, the following code imports java.lang.System and prints a string calling System.out.println():

```
var System = JavaImport("java.lang.System")
System.out.println("foo");
```

Here we instantiate a Java `File` object and check that it exists:

```
var File = JavaImport("java.io.File")
var myFile = new File("myFile.txt")

if (myFile.exists()) {
        print("myFile.txt exists")
} else {
        print("myFile.txt does not exist")
}
```

> **Warning:** You cannot instantiate an anonymous class from an interface or an abstract class with the `new` keyword and `JavaImport`. Nevertheless, you can still access to static fields and methods.

Implement JavaScript Functions in Java

We can also implement JavaScript functions in Java by adding their implementation to the global object from Java. For example, here is the code to create a JavaScript function named `javaPrint` in the global scope:

```
JsRuntime.JS_
↪GLOBAL_OBJECT.put("javaPrint", JsRuntime.createFunction(new JsClosure() {
            @Override
            public Object invoke(Object thisBinding, Object... arguments) {
                    System.out.println("Print from Java: " + arguments[0]);
                    return null;
            }
    }), false);
```

The function is created with a `com.microej.js.objects.JsObjectFunction` object created with the API `JsRuntime.createFunction(JsClosure jsClosure)`, and injected in the object `JsRuntime.JS_GLOBAL_OBJECT` which maps to the JavaScript global scope.

The function `javaPrint` can then be used in JS:

```
javaPrint("foo")
```

This technique can also be used to share any Java object to JavaScript. It is achieved by returning the Java object in the `invoke` method of the `JsClosure` object. For example, a Java Date object can be exposed as follows:

```
JsRuntime.JS_GLOBAL_
↪OBJECT.put("getCurrentDate", JsRuntime.createFunction(new JsClosure() {
        @Override
        public Object invoke(Object thisBinding, Object... arguments) {
                return Calendar.getInstance().getTime();
        }
}), false);
```

When a Java object is exposed in JavaScript, all its public methods can be called, therefore the JavaScript code can then use this Date object and get the time:

```
var date = getCurrentDate()
var time = date.getTime()
print("Current time: ", time)
```

for more information on how these called are managed by the MicroEJ JavaScript engine, please go to the *Foreign Function Interface* section.

Java objects can also be shared using one of the other Java JS adapter objects. With this solution, the code of the Java object is executed at engine initialisation, contrary to the previous solution where it is executed only when the JavaScript code is called. For example, here is the code to expose a Java string named `javaString` in the JavaScript global scope:

```
JsRuntime.JS_GLOBAL_OBJECT.put("javaString", "Hello World!", false);
```

The string `javaString` can then be used in JS:

```
var myString = javaString;
```

The available Java JS adapter objects are:

- `com.microej.js.objects.JsObject` : exposes a Java object as a JavaScript object
- `com.microej.js.objects.JsObjectFunction` : exposes a Java "process" as a JavaScript function (using a JsClosure object)
- `com.microej.js.objects.JsObjectString` : exposes a Java String as a JavaScript String
- `com.microej.js.objects.JsObjectArray` : exposes a Java items collection as a JavaScript Array
- `com.microej.js.objects.JsObjectBoolean` : exposes a Java Boolean as a JavaScript Boolean
- `com.microej.js.objects.JsObjectNumber` : exposes a Java Number as a JavaScript Number

Calling JavaScript from Java

The MicroEJ JavaScript engine API allows to call JavaScript code from Java code. For example, given the following JavaScript function in a file in `src/main/js`:

```
function sum(a, b) {
    print(a + " + " + b + " = " + (a+b));
}
```

it can be called from a Java piece of code with:

```
JsObjectFunction
↪functionObject = (JsObjectFunction) JsRuntime.JS_GLOBAL_OBJECT.get("sum");
JsRuntime.ENGINE.addJob(functionObject,
↪ JsRuntime.JS_GLOBAL_OBJECT, new Integer(5), new Integer(3));
```

The first line gets the JavaScript function from the global scope. The second line adds a job in the JavaScript engine queue to execute the function, in the global scope, with the arguments `5` and `3`.

Passing Values Between JavaScript and Java

JavaScript base types are represented by Java objects and not Java base types. The following table shows the mapping between types in both languages:

JavaScript	Java
Number	java.lang.Integer or java.lang.Double
Boolean	java.lang.Boolean
String	java.lang.String
Null	`null` value
Undefined	`JsRuntime.JS_UNDEFINED_OBJECT` singleton

In JavaScript, a `Number` type is a 64-bits floating-point value. Nevertheless, Kifaru may use integer values (Integer Java type) when possible for performance reasons. Otherwise, Double type will be used.

Note: Prefer passing Integer values as argument to a job added to the JavaScript execution queue, or return `Integer` values when implementing a `JsClosure` instead of Double when possible.

It is not possible to retrieve the returned value of a JavaScript function from Java. For instance, consider the following JavaScript function:

```
function sum(a, b) {
    return a + b;
}
```

When calling this function from Java, we have no way to get the result back:

```
JsObjectFunction
↪functionObject = (JsObjectFunction) JsRuntime.JS_GLOBAL_OBJECT.get("sum");
JsRuntime.ENGINE.addJob(functionObject,
↪ JsRuntime.JS_GLOBAL_OBJECT, new Integer(5), new Integer(3));
```

A workaround is to modify the JavaScript function so it takes a callback object as argument:

```
function sum(a, b, callback) {
    callback.returnValue(a + b);
}
```

Here is a possible implementation of the callback object:

```
public class Callback<T> {

    @Nullable
    private T value;

    private boolean returned;

    /**
     * Gets the value returned by this callback function when ready.
     * <p>
     * A call to this method waits for the value to be ready.
     *
     * @return the value return by the callback
     */
        @Nullable
    public T getValue() {
        synchronized (this) {
            while (!this.returned) {
                try {
                    wait();
                } catch (InterruptedException e) {
                    throw new JsErrorWrapper(""); //$NON-NLS-1$
                }
            }
        }

         return this.value;
    }

    /**
     * Sets the value to return by this callback function.
     *
     * @param value
     *            the value to return
     */
    public synchronized void returnValue(@Nullable T value) {
        this.value = value;
        this.returned = true;
        notify();
    }
}
```

We can now pass the callback to the job. The Java code will wait on the `callback.getValue()` until the result is ready.

```
JsObjectFunction␣
↪functionObject = (JsObjectFunction) JsRuntime.JS_GLOBAL_OBJECT.get("sum");
Callback<Integer> callback = new Callback<>();
JsRuntime.ENGINE.addJob(functionObject,
↪ JsRuntime.JS_GLOBAL_OBJECT, new Integer(5), new Integer(3), callback);
Integer returnedValue = callback.getValue();
System.out.println("Result is " + returnedValue);
```

Tests

JavaScript applications can be tested with tests written in JavaScript. The JavaScript test files must be located in the project folder `src/test/js`. All JavaScript files (`*.js`) found in this folder, at any level, are considered as test files.

In order to setup JavaScript tests for your application, follow these steps:

- create an *Add-On Library* project or a *Standalone Application* project
- define the following configuration in the build file of the project:

  Gradle (build.gradle.kts)

  MMM (module.ivy)

```
tasks.test {
   filter {
      includeTestsMatching("*._JsTest_*Code")
   }
}
```

Add these properties nside the `ea:build` tag (if the properties already exist, replace them):

```
<ea:property␣
↪name="test.run.includes.pattern" value="**/_JsTest_*Code.class"/>
<ea:property␣
↪name="target.main.classes" value="${basedir}/target~/test/classes"/>
```

- add the MicroEJ JavaScript dependency in the build file of the project:

  Gradle (build.gradle.kts)

  MMM (module.ivy)

```
implementation("com.microej.library.runtime:js:0.13.0")
```

```
<dependency org="com.microej.library.runtime" name="js" rev="0.13.0"/>
```

- define the platform to use to run the tests with one of the options described in *Platform Selection* section
- create a file `assert.js` in the folder `src/test/resources` with the following content:

```
var assertionCount = 0;

function assert(value) {
        assertionCount++;
```

(continues on next page)

(continued from previous page)

```
        if (value == 0) {
                print("assert " + assertionCount + " - FAILED");
        } else {
                print("assert " + assertionCount + " - PASSED");
        }
}
```

This method `assert` will be available in all tests to do assertions.

- create a file `test.js` in the folder `src/test/js` and write your first test:

```
var a = 5;
var b = 3;
var sum = a + b;
assert(sum === 8);
```

- build the application in the SDK or in command line with the *MMM CLI*

  The execution of the tests produces a report available in the folder `target~/test/html` for the project.

Limitations

The MicroEJ engine supports the version 5.1 of the ECMAScript specification, with the limitations described in this page.

Unsupported Directives

Directives, such as `'use strict'` , are not supported and are considered as literal statements. Literal statements are just ignored.

Unsupported Statements

The following syntaxes are not supported by the MicroEJ JavaScript engine:

- `with (x) { }` : the `with` statement is not supported in MicroEJ since its usage is not recommended. See the reference documentation for more information.

Unsupported Built-in Objects

The unsupported built-in objects are listed in the *API section*.

Troubleshooting

Compilation error `cannot be resolved to a type` in FFI class

A compilation error can be raised when the classpath contains unexpected classes, for example:

```
Exception in thread "main" java.lang.Error: Unresolved compilation problems:
        ArrayComparisonFailure cannot be resolved to a type
        ArrayComparisonFailure cannot be resolved to a type

        at java.lang.Throwable.fillInStackTrace(Throwable.java:82)
        at java.lang.Throwable.<init>(Throwable.java:37)
        at java.lang.Error.<init>(Error.java:18)
        at com.microej.js.JsFfi.ffi_toString_0(JsFfi.java:54)
        at com.microej.js.JsCode$1$1.invoke(JsCode.java:50)
```

As described in *the FFI section*, in order to call Java methods from JavaScript code, all the methods with the given names are searched in the classpath. Since the classpath can contain test dependencies which are not available at compile time, the generated FFI can contain classes from these dependencies and therefore fail to compile. The following classes are excluded by default:

- `ej.junit.*`
- `org.junit.*`
- `junit.*`
- `org.hamcrest.*`
- `java.lang.String`
- `java.lang.Number`

This list can be changed by setting the system property `js.ffi.excludes.classes` to a comma-separated list of FQN patterns. For example:

```
js.ffi.excludes.classes=ej.junit.*,org.junit.*,junit.
↪*,org.hamcrest.*,java.lang.String,java.lang.Number,com.mycompany.test.*
```

Warning: Defining this property overwrites the default value, so do not forget to keep the default excluded classes (unless you know what you are doing).

Internals

JavaScript Sources Processing

The JavaScript code is not executed directly, it is first translated in Java code and compiled with the Java application code. This transpilation is done by the JavaScript Add-On Processor. This processor uses the OpenJDK Nashorn library (extracted from *jre1.8.0_92*) to parse the Javascript files.

The operations performed by this processor are summarized in this diagram:



- **Parsing**: all JavaScript source files located in the folder `src/main/js` and `src/test/js` are parsed by the Nashorn library to provide a JavaScript AST.
- **JS Validation**: validation on the JavaScript AST to detect unsupported language features (for example `eval`).
- **Conversion preparation**: before actually converting the JavaScript AST to a Java AST, a preparation operation is done to initialize all the lexical environments (done by `JsIrVisitor`).
- **Conversion**: conversion of the JavaScript AST to a Java AST.
- **Java AST cleanup/optim**: post-conversion step to cleanup and optimize the Java AST. The following operations are done: - fix imports - remove dead code - remove literal statements
- **Java sources generation**: generation of the Java sources from the Java AST.

Foreign Function Interface

As said in the section *Calling Java from JavaScript*, a JavaScript code can manipulate Java objects and call methods on Java objects. This chapter describes how does the call to methods on Java objets work.

Let `getValue()` a Java method called from JavaScript on a Java object. As long as the type of the object is not known at compile-time in the JavaScript code, all the types containing a method with the same signature are searched in the classpath. Then the JavaScript preprocessor generates a `JsFfi` class and a method that dynamically tries to find the type of the receiver object. So, when the `getValue()` method is called from JavaScript, this generated method is called.

> **Warning:** Calling a method whose name is very common could result in a delay while calling it, and some useless methods embedded.

This example shares a Java Date of the current time:

```
JsRuntime.JS_GLOBAL_
↪OBJECT.put("getCurrentDate", JsRuntime.createFunction(new JsClosure() {
        @Override
        public Object invoke(Object thisBinding, Object... arguments) {
                return Calendar.getInstance().getTime();
        }
}), false);
```

The JavaScript can then use this Date to print the current time:

```
var date = getCurrentDate()
var time = date.getTime()
print("Current time: ", time)
```

In this case, the generated method in *JsFfi* looks like:

```
public static Object␣
↪ffi_getTime_0(Object function, @ej.annotation.Nullable Object this_) {
        try {
                if (this_ instanceof JsObject || this_ instanceof String)
                    ␣
↪   return JsRuntime.functionCall(((Reference) function).getValue(), this_);
                if (this_ instanceof Calendar) {
                        return ((Calendar) this_).getTime();
                }
                if (this_ instanceof Date) {
                        return new Double(((Date) this_).getTime());
                }
        } catch (JsErrorWrapper e) {
                throw e;
        } catch (Throwable t) {
                throw new JsErrorWrapper(new JsObjectError.TypeError(
↪"A Java exception has been thrown in generated FFI code of getTime"), t);
        }
        throw new JsErrorWrapper(new JsObjectError.TypeError("getTime"));
}
```

5.15 Development Tools

5.15.1 Event Tracing

Description

Event Tracing allows to record integer based events for debugging and monitoring purposes without affecting execution performance too heavily. Basically, it gives access to `Tracer` objects that are named and can produce a limited number of different event types.

A record is an event type identified by an `eventID` and can have a list of values. It can be a single event or a period of time with a start and an end.

Event Tracing can be accessed from two APIs:

- A Java API, provided by the Trace API module. The following dependency must be added to the build file of the MicroEJ Application project:

  Gradle (build.gradle.kts)

  MMM (module.ivy)

```
implementation("ej.api:trace:1.1.0")
```

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

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

Event Recording

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 eventID) methods. The `recordEventEnd(...)` method can also have another integer parameter for a custom value for the event end. One can use it to trace the returned value of a method.

The Trace API also provides a String constant Tracer.TRACE_ENABLED_CONSTANT_PROPERTY representing the *Constant* value of `core.trace.enabled` option. This constant can be used to *remove at build time* portions of code when the trace system is disabled. To do that, just surround tracer record calls with a if statement that checks the constant's state. When the constant is set to `false`, the code inside the if statement will not be embedded with the application and thus will not impact the performances.

```
if(Constants.getBoolean(Tracer.TRACE_ENABLED_CONSTANT_PROPERTY)) {
  // This␣
↪code is not embedded if TRACE_ENABLED_CONSTANT_PROPERTY is set to false.
  tracer.recordEventEnd(0);
}
```

Examples:

- Trace a single event:

```
private static final Tracer tracer = new Tracer("Application", 100);

public static void main(String[] args) {
  Tracer.startTrace();
  tracer.recordEvent(0);
}
```

Standard Output:

```
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.

A Platform can connect its own implementation by overriding the functions defined in the `LLTRACE_impl.h` file.

MicroEJ Corp. provides an implementation that redirects the events to *SystemView* tool, the real-time recording and visualization tool from Segger. It is perfect for a finer understanding of the runtime behavior by showing events sequence and duration.

A implementation example for the NXP OM13098 development board with SystemView support is available here.

Please contact *our support team* for more information about how to integrate this Platform module.

Advanced Event Tracing

Method invocation can be *profiled*.

Note: This feature requires Architecture version `7.17.0` or higher and is only available on MicroEJ Core Engine, not on Simulator.

MicroEJ Corp. provides an implementation on Linux targets to profile an Application and generate a flamegraph for the Trace Compass tool.

Please contact *our support team* for more information about how to generate flamegraph.

5.15.2 VEE Debugger Proxy

Principle

The VEE debugger proxy is an implementation of the Java Debug Wire protocol (JDWP) for debugging Applications executed by MICROEJ VEE. It consists of a TCP server implementing the JDWP protocol and acting as a proxy between the IDE (debugger) and the Executable (debuggee) running on the device.

The debugger proxy allows a postmortem debug from a snapshot of the memory (core dump file for Linux/QNX targets and Intel Hex file for MCU targets) of a running Executable binary.

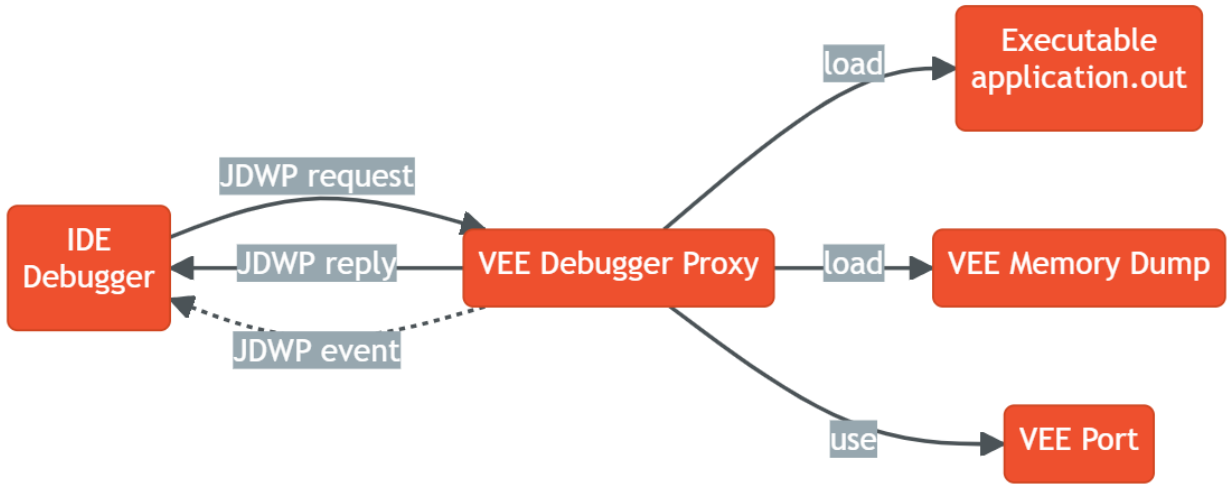


Fig. 36: Debugger Proxy Principle

Warning: The snapshot of the memory (core dump or Intel Hex files) should only be generated when the Core Engine task is stopped on one of the Core Engine hooks (`LLMJVM_on_OutOfMemoryError_thrown` , `LLMJVM_on_Runtime_gc_done` etc.) or in a native function. Otherwise, the Core Engine memory dump is not guarranted to be consistent, which may cause the VEE Debugger to crash abruptly.

Note: This feature requires Architecture version `8.1.0` or higher and works for both Mono-Sandbox and Multi-Sandbox Executables.

Please contact *our support team* to get the VEE Debugger Proxy tool `microej-debugger-proxy.jar` compatible with your Architecture version.

Debugging Executable for Linux or QNX target

In order to debug an Executable for Linux or QNX target, you need to dump the memory of the running Executable and then run the VEE Debugger Proxy.

For Linux and QNX target, the memory dump must be a core dump file.

Generate a Core Dump File using GDB

Open a shell terminal on the device and enter the following commands:

```
# Instruct the Linux kernel to Dump file-backed private mappings.
echo 0x37 > /proc/self/coredump_filter
# Start GDB
gdb ./application.out
```

The following GDB script can be used to generate a core dump file:

- when the signal `SIGUSR1` is received
- or when an out of memory error occurs
- or when an explicit garbage collection (GC) is done.

You can run the script directly in the GDB console.

```
## From GBD documentation:
## generate-core-file [file]
##   Produce a core dump of the inferior process.
##   The␣
↪optional argument file specifies the file name where to put the core dump.
##   If not specified,␣
↪the file name defaults to 'core.pid', where pid is the inferior process ID.

## Generate a core dump when the signal SIGUSR1 is received
catch signal SIGUSR1
commands
silent
generate-core-file
```

(continues on next page)

(continued from previous page)

```
cont
end

## Generate a core dump when an out of memory error occurs
break LLMJVM_on_OutOfMemoryError_thrown
commands
silent
generate-core-file
cont
end

## Generate a core dump when an explicit garbage collection (GC) is done
break LLMJVM_on_Runtime_gc_done
commands
silent
generate-core-file
cont
end
```

Starts executing the Mono-Sandbox Executable under GDB:

```
# In the GDB console:
run
```

A core dump file will be generated once the Executable reach one of the breaking conditions described previously.

Run the VEE Debugger Proxy

Open a shell terminal on your workstation and run the following command:

```
java -DveePortDir=<path to VEE Port directory> \
    -Ddebugger.port=<8000> \
    -Ddebugger.out.path=<path to the Executable file (application.out)> \
    -Ddebugger.features.out.path=<comma-
↪separated list of the Feature files with debug information (*.fodbg files).
↪ To be used if you want to debug an installed Sandboxed Application> \
    -Ddebugger.out.coredump.path=<path to the core dump file> \
    -jar microej-debugger-proxy.jar
```

Open the SDK and run a *Remote Java Application Launch* to debug your code.

Debugging Executable for MCU target

The VEE Debugger Proxy for MCU target requires a memory dump of the running Executable in Intel Hex format. It provides a tool to generate a script for IAR (IAR8 or IAR9) or GDB debugger, that contains the needed commands to dump the required memory regions in Intel Hex format.

Generate VEE memory dump script for IAR (IAR8 or IAR9) or GDB debugger

Open a shell terminal on your workstation and run the following command:

```
java -DveePortDir=<path to VEE Port directory> \
    -Ddebugger.out.path=<path to the Executable file (application.out)> \
    -cp microej-debugger-proxy.jar com.microej.jdwp.VeeDebuggerCli \
    --debugger=IAR8|IAR9|GDB \
    --output=<Output directory where the script file will be generated>
```

A script file named `vee-memory-dump.mac` (for IAR) or `vee-memory-dump.gdb` (for GDB) is generated into the specified output directory.

You can now use this script to dump the memory of the running Executable.

Dump the memory of the running Executable

With IAR Debugger

Note: You must use a version of IAR Workbench for which the `vee-memory-dump.mac` script file is generated.

A script file generated for IAR8 will not work on IAR Workbench 9.x.x and vice versa.

In IAR Embedded Workbench:

- Register the generated `vee-memory-dump.mac` script file in the debugger project option:

1. Open the Debugger Project option window by clicking on `Project > Options... > Debugger > Setup`
2. Check the option `Use macro file(s)` and browse to the generated `vee-memory-dump.mac` file.
3. Click on `OK` to confirm.

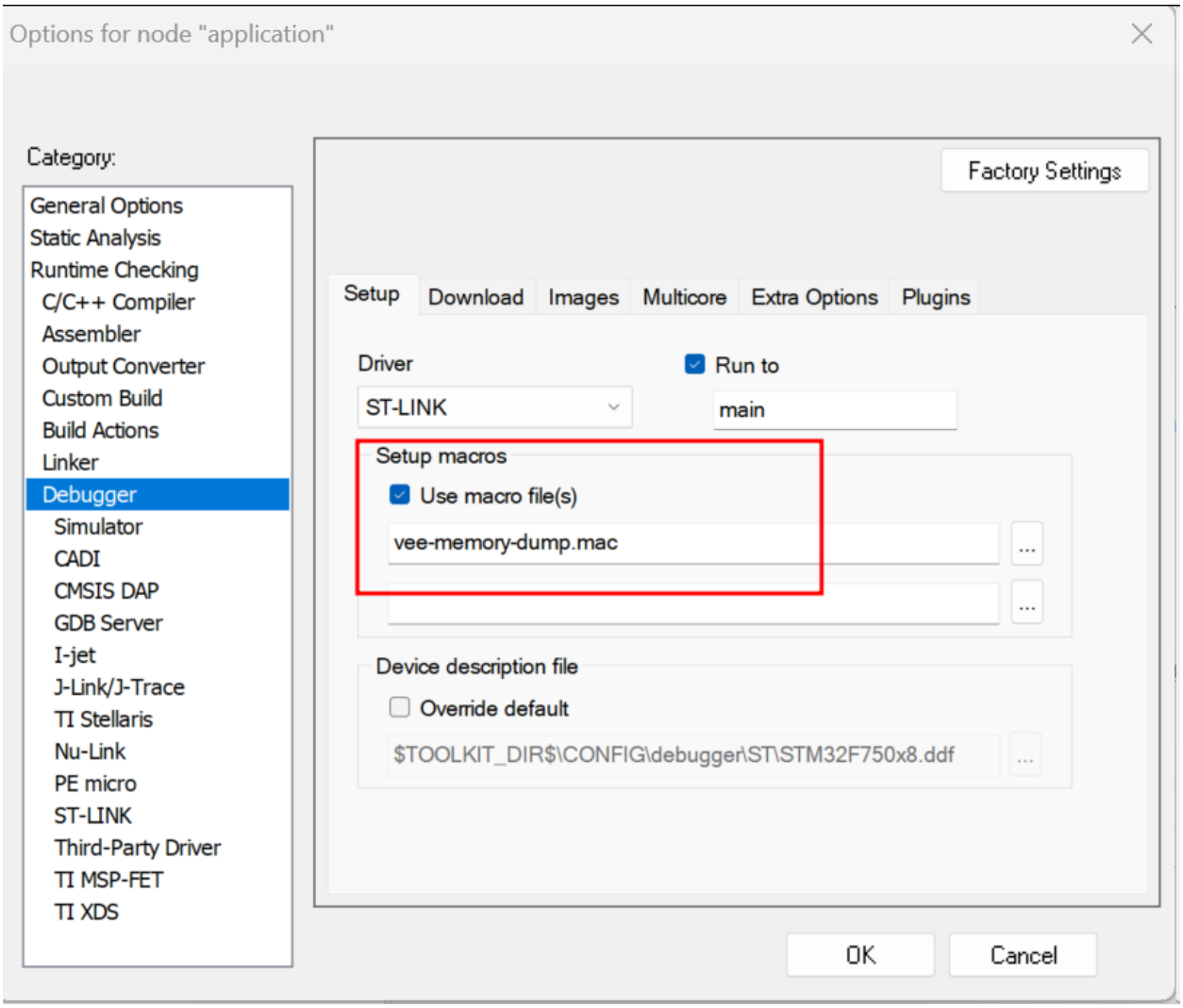


Fig. 37: IAR Debugger Project Option

- Add the macro `dumpMemories()` as an action expression to a code breakpoint:

1. Open IAR Breakpoints window by clicking on `View > Breakpoints`
2. Right click on IAR Breakpoints window and select `New Breakpoint > Code`
3. In the `Expression` text field, enter `dumpMemories()` and click on `OK`

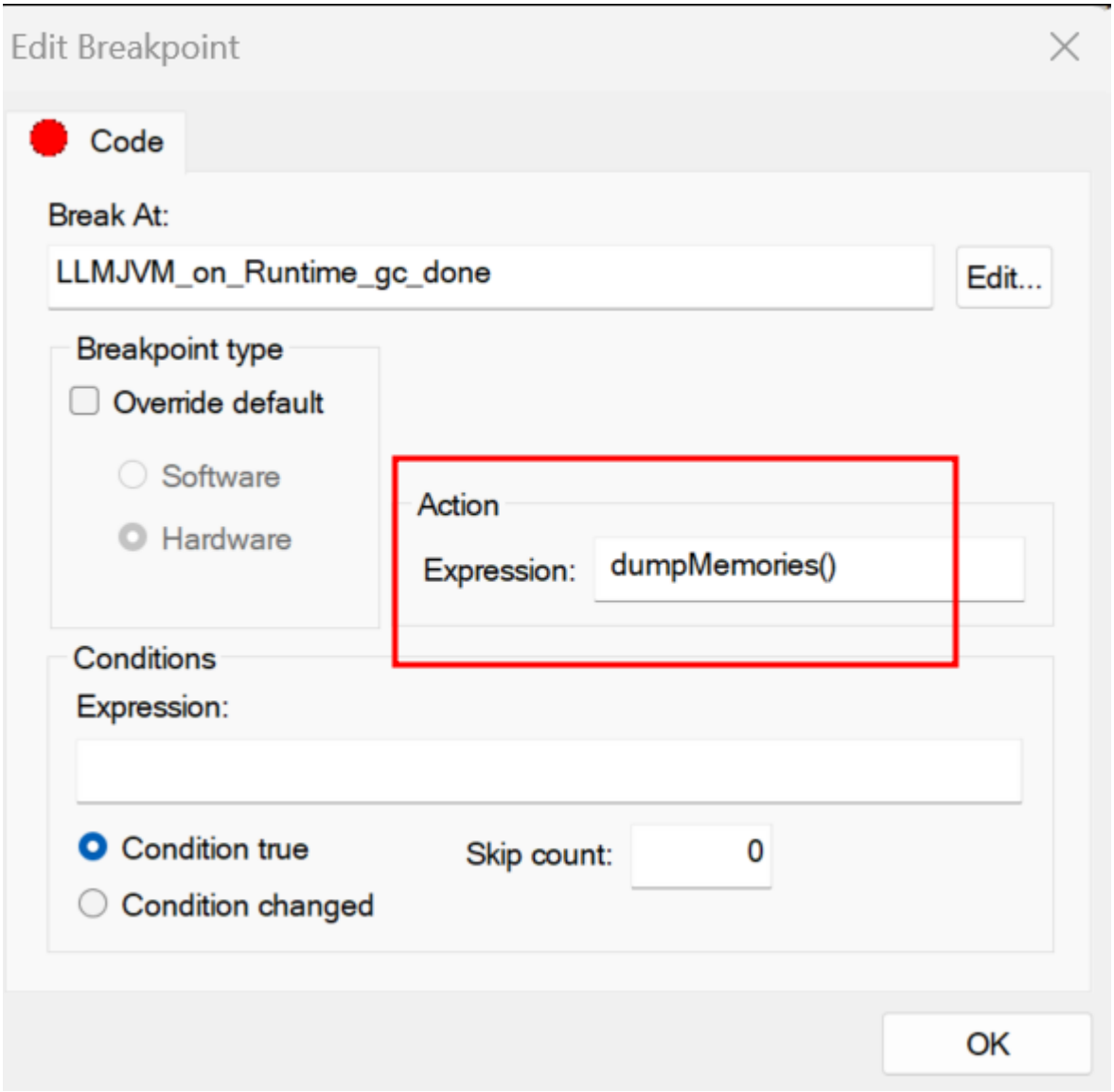


Fig. 38: IAR Breakpoint editor

When the IAR Debugger hits the specified breakpoint, the `dumpMemories()` macro function is executed and the memory is dumped into `*.hex` files.

The `*.hex` files are generated in the same directory as the `vee-memory-dump.mac` file.

With GNU Debugger (GDB)

In your GDB console:

- Create a breakpoint at a specific safe point (Core Engine hooks or native function)

```
# E.g. Add breakpoint at LLMJVM_on_Runtime_gc_done hook
break LLMJVM_on_Runtime_gc_done
run
```

- When the running Executable stops at the Breakpoint, run the `vee-memory-dump.gdb` script file to dump the memory.

```
# E.g. Run the GDB memory dump script
source [/path/to]/vee-memory-dump.gdb
```

The memory is dumped into `*.hex` files in the same directory as the `vee-memory-dump.gdb` file.

Start the VEE Debugger Proxy

Open a shell terminal on your workstation and run the following command:

```
java -DveePortDir=<path to VEE Port directory> \
    -Ddebugger.port=<8000> \
    -Ddebugger.out.path=<path to the Executable file (application.out)> \
    -Ddebugger.features.out.path=<comma-
↪separated list of the Feature files with debug information (*.fodbg files).
↪ To be used if you want to debug an installed Sandboxed Application> \
    -Ddebugger.out.hex.path=<comma-separated list of the memory dump files␣
↪in Intel Hex format or a single file containg all the dumped memory> \
    -jar microej-debugger-proxy.jar
```

Open the SDK and run a *Remote Java Application Launch* to debug your code.

Note: If you have multiple `*.hex` files generated in the previous step, you can if you want merge them into a single `*.hex` file.

It will be easier to use a single `*.hex` file than multiple files in the Debugger Proxy command line.

You can run the following shell script to merge all the `*.hex` files into a single file called `all.hex` for example.

Make sure to move to the directory where `*.hex` files are generated before running the script.

- On Windows workstation

```
set ALL_HEX="all.hex"
rem delete all.hex file if it exists
if exist "%ALL_HEX%" (del /f %ALL_HEX%)
rem merge all the *.hex files
copy /b *.hex %ALL_HEX%
```

- On Linux workstation

```
#!/usr/bin/bash
ALL_HEX="all.hex"
#delete all.hex file if it exists
test -f $ALL_HEX && rm $ALL_HEX
#merge all the *.hex files
cat *.hex > $ALL_HEX
```

Now, use this single `all.hex` file as value to the Debugger Proxy option `-Ddebugger.out.hex.path`

VEE Debugger Proxy Options Summary

- **veePortDir**: The path to the VEE Port directory (must point to the *source* folder of the VEE Port.).
- **debugger.port**: The TCP server port, defaults to `8000`.
- **debugger.out.path**: The Path to the Executable file to debug (`application.out`).
- **debugger.features.out.path**: comma-separated list of the Feature files with debug information (`*.fodbg files`). This option must be used if you want to debug an installed Sandboxed Application. In this case, note that the specified Executable in `debugger.out.path` option must be the Multi-Sandbox Executable.
- **debugger.out.coredump.path**: The Path to the core dump file (conflict with **debugger.out.hex.path** option).
- **debugger.out.hex.path**: The Path to the memory dump files in Intel Hex format (conflict with **debugger.out.coredump.path** option). If you have multiple Intel Hex files, you can either merge them into a single file or list them with a comma separator, such as `[/path/to]/java_heap.hex,[/path/to]/java_stacks.hex,[/path/to]/vm_instance.hex`.

Troubleshooting

You may encounter some command line issues if you try to run the proxy on Windows Powershell.

On Windows workstation, we recommend using `CMD` Command Prompt instead.

5.15.3 Dependency Discoverer

Introduction

Dependency Discoverer is a tool that lists unresolved dependencies (types, methods and fields) of a set of Java ARchive (JAR) files and .class files. It is a versatile tool and can be used in other contexts, for instance, to list every dependency of a JAR file.

It can be used through a command-line interface, with the possibility to output the result in JSON or XML format, allowing an easy scripting process.

Installation

This tool is available at https://github.com/MicroEJ/Tool-ApiDependencyDiscoverer. A JAR and Windows executable version can be downloaded from the release page. It is also possible to clone and import the project in the SDK and use it from sources.

Use

For usage information, see https://github.com/MicroEJ/Tool-ApiDependencyDiscoverer/blob/master/README.md.

5.15.4 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 10: 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 11: 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 20: 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.

continues on next page

Table 20 – continued from previous page

Tags	Attributes	Description
	name	Name of the section. The section name may not be unique. However, it is recommended that you define a unique name if the section must be referred separately for memory positioning.
	start	Optional. Expression defining the absolute start address of the section. Must be resolved to a constant after the full load of the lsc file.
	end	Optional. Expression defining the absolute end address of the section. Must be resolved to a constant after the full load of the lsc file.
	size	Optional. Expression defining the size in bytes of the section. Invariant: `(end-start)+1=size`. Must be resolved to a constant after the full load of the lsc file.
	align	Optional. Expression defining the alignment in bytes of the section.
	rootSection	Optional. Boolean value. Sets this section as a root section to be embedded even if it is not targeted by any embedded symbol. See also `rootSection` tag.
	symbolPrefix	Optional. Used in collaboration with `symbolTags`. Prefix of symbols embedded in the auto-generated section. See *Auto-generated Sections*.
	symbolTags	Optional. Used in collaboration with `symbolPrefix`. Comma separated list of tags of symbols embedded in the auto-generated section. See *Auto-generated Sections*.
defSymbol		Defines a new global symbol. Symbol name must be unique in the linker context
	name	Name of the symbol.
	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`

continues on next page

Table 20 – continued from previous page

Tags	Attributes	Description
lscFragment		Root tag for an lsc file fragment. Lsc fragments are loaded from the linker path option, or included from a master file using the include tag.
memoryLayout		Describes the organization of a set of memory intervals. The memory layouts are processed in the order in which they are declared in the file. The same interval may be organized in several layouts. Each layout starts at the value of the cursor the previous layout ended. The following tags are allowed within a memoryLayout directive: defSymbol (under certain conditions), group, memoryLayoutRef, padding, and sectionRef.
	ranges	Exclusive with default. Comma-separated ordered list of fixed sections to which the layout is applied. Sections represent memory segments.
	image	Optional. Boolean value. false if not set. If true, the layout describes a part of the binary image: Only image sections can be embedded. If false, only runtime sections can be embedded.
	relocation	Optional. Name of the section to which this layout is relocated.
	name	Exclusive with ranges. Defines a named memoryLayout directive instead of specifying a concrete memory location. May be included in a parent memoryLayout using memoryLayoutRef.
memoryLayoutRef		memoryLayout directive. Provides an extension-point mechanism to include memoryLayout directives defined outside the current one.
	name	All directives of memoryLayout defined with the same name are included in an undefined order.
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.

continues on next page

Table 20 – continued from previous page

Tags	Attributes	Description
	name	Name of the sections to embed. When the name ends with *, all sections starting with the given name are embedded (name completion), except sections that are embedded in another sectionRef using the exact name (without completion).
	symbol	Optional. Only embeds the section targeted by the given symbol. This is the only way at link level to embed a specific section whose name is not unique.
	force	Optional. Deprecated. Replaced by the rootSection tag. The only allowed value is true. By default, for compaction, the linker embeds only what is needed. Setting this attribute will force the linker to embed all sections that appear in all loaded relocatable files, even sections that are not targeted by a symbol.
	sort	Optional. Specifies that the sections must be sorted in memory. The value can be: - order : the sections will be in the same order as the input files - name : the sections are sorted by their file names - unit : the sections declared in an object file are grouped and sorted in the order they are declared in the object file
u4		Binary section statement. Describes the four next raw bytes of the section. Bytes are organized in the endianness of the target ELF executable.
	value	Expression must be resolved to a constant after the full load of the lsc file (32 bits value).
fill		Binary section statement. Fills the section with the given expression. Bytes are organized in the endianness of the target ELF executable.
	size	Expression defining the number of bytes to be filled.
	value	Expression must be resolved to a constant after the full load of the lsc file (32 bits value).

Expressions

An attribute expression is a value resulting from the computation of an arithmetical and logical expression. Supported operators are the same operators supported in the Java language, and follow Java semantics:

- Unary operators: `+ , - , ~ , !`
- Binary operators: `+ , - , * , / , % , << , >>> , >> , < , > , <= , >= , == , != , &, | , ^ , && , ||`
- Ternary operator: `cond ? ifTrue : ifFalse`
- Built-in macros:
  - `START(name)` : Get the start address of a section or a group of sections
  - `END(name)` : Get the end address of a section or a group of sections
  - `SIZE(name)` : Get the size of a section or a group of sections. Equivalent to `END(name)-START(name)`
  - `TSTAMPH()` , `TSTAMPL()` : Get 32 bits linker time stamp (high/low part of system time in milliseconds)
  - `SUM(name,tag)` : Get the sum of an auto-generated section (*Auto-generated Sections*) column. The column is specified by its tag name.

An operand is either a sub expression, a constant, or a symbol name. Constants may be written in decimal (`127`) or hexadecimal form (`0x7F`). There are no boolean constants. Constant value `0` means `false` , and other constants' values mean `true` . Examples of use:

```
value="symbol+3"
value="((symbol1*4)-(symbol2*3)"
```

Note: Ternary expressions can be used to define selective linking because they are the only expressions that may remain partially unresolved without generating an error. Example:

```
<defSymbol name="myFunction" value="condition ? symb1 : symb2"/>
```

No error will be thrown if the condition is `true` and `symb1` is defined, or the condition is `false` and `symb2` is defined, even if the other symbol is undefined.

Auto-generated Sections

The MicroEJ Linker allows you to define sections that are automatically generated with symbol values. This is commonly used to generate tables whose contents depends on the linked symbols. Symbols eligible to be embedded in an auto-generated section are of the form: `prefix_tag_suffix` . An auto-generated section is viewed as a table composed of lines and columns that organize symbols sharing the same prefix. On the same column appear symbols that share the same tag. On the same line appear symbols that share the same suffix. Lines are sorted in the lexical order of the symbol name. The next line defines a section which will embed symbols starting with `zeroinit` . The first column refers to symbols starting with `zeroinit_start_` ; the second column refers to symbols starting with `zeroinit_end_` .

```
<defSection
        name=".zeroinit"
        symbolPrefix="zeroInit"
        symbolTags="start,end"
    />
```

Consider there are four defined symbols named `zeroinit_start_xxx` , `zeroinit_end_xxx` , `zeroinit_start_yyy` and `zeroinit_end_yyy` . The generated section is of the form:

```
0x00: zeroinit_start_xxx
0x04: zeroinit_end_xxx
0x08: zeroinit_start_yyy
0x0C: zeroinit_end_yyy
```

If there are missing symbols to fill a line of an auto-generated section, an error is thrown.

Execution

MicroEJ Linker can be invoked through an ANT task. The task is installed by inserting the following code in an ANT script

```
<taskdef
    name="linker"
    classname="com.is2t.linker.GenericLinkerTask"
    classpath="[LINKER_CLASSPATH]"
/>
```

`[LINKER_CLASSPATH]` is a list of path-separated jar files, including the linker and all architecture-specific library loaders.

The following code shows a linker ANT task invocation and available options.

```
<linker
    doNotLoadAlreadyDefinedSymbol="[true|false]"
    endianness="[little|big|none]"
    generateMapFile="[true|false]"
    ignoreWrongPositioningForEmptySection="[true|false]"
    lsc="[filename]"
    linkPath="[path1:...pathN]"
    mergeSegmentSections="[true|false]"
    noWarning="[true|false]"
    outputArchitecture="[tag]"
    outputName="[name]"
    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 21: 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 22: Linker-Specific Configuration Tags

Message ID	Description
0	The linker has encountered an unexpected internal error. Please contact the support hotline.
1	A library cannot be loaded with this linker. Try verbose to check installed loaders.

continues on next page

Table 22 – continued from previous page

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.

continues on next page

Table 22 – 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.

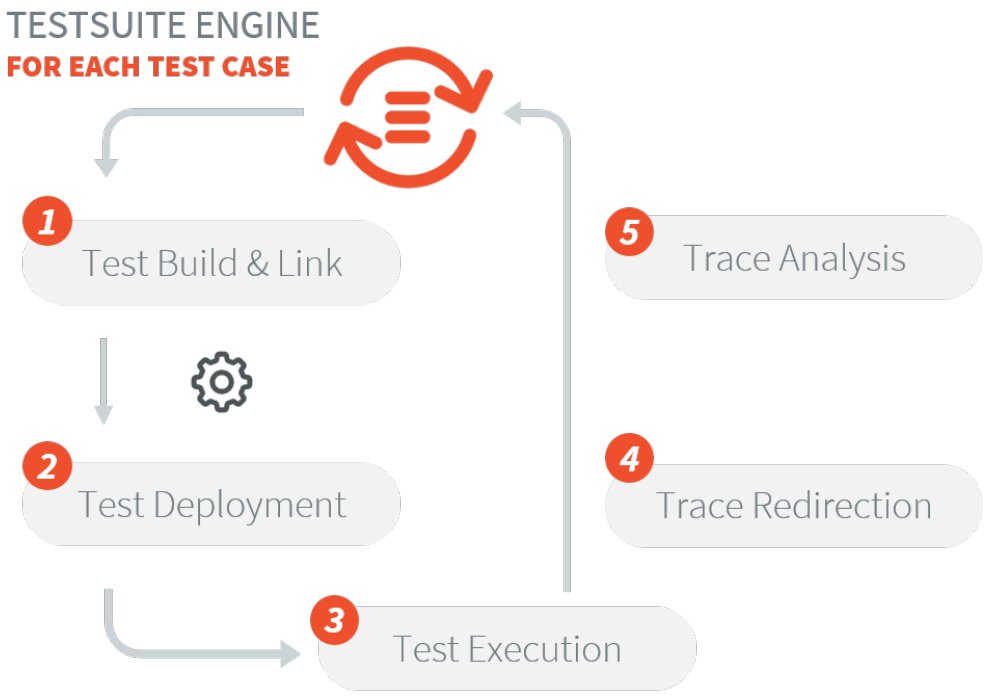
5.15.5 MicroEJ Test Suite Engine

Introduction

The MicroEJ Test Suite 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 test suites in their build flow.

The MicroEJ Test Suite Engine allows the user to test any kind of projects within the configuration of a generic Ant file.



The MicroEJ Test Suite Engine is already pre-configured for running test suites on a MicroEJ Platform (either on Simulator or on Device).

- For Application and Libraries, refer to *Test Suite with JUnit* section.
- For Foundation Libraries Test Suites, refer to *VEE Port Test Suite* section.

Using the MicroEJ Test Suite 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 test suite and to retrieve an XML report document in a JUnit format.
- `javaTestsuite` is a subtask of the `testsuite` task, used to run a specialized test suite 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 23: `testsuite` task mandatory attributes

Attribute Name	Description
`outputDir`	The output folder of the test suite. 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 test suite engine. It is called with a `basedir` located at output location of the current test.

The test suite engine provides the following properties to the harness script giving all the informations to start the test:

Table 24: `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 25: `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 test suite. 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 test suite. 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 test suite was invoked.
`productName`	The name of the current tested product.	`TestSuite`
`jvm`	The location of your Java VM to start the test suite (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 test suite (`true` or `false`), depending if every tests successfully passed the test suite or not. Ignored tests do not affect this result.	None

Finally, you have to give as nested element the path containing the tests.

Table 26: `testsuite` task nested elements

Element Name	Description
`testPath`	Containing all the file of the tests which will be launched by the test suite.
`testIgnoredPath` (optional)	Any test in the intersection between `testIgnoredPath` and `testPath` will be executed by the test suite, 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 12: Example of test suite 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 test suite 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 27: `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 13: 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}"
```

(continues on next page)

(continued from previous page)

```
>
    <testPath refid="target.@{prefix}.path"/>
    <testIgnoredPath refid="tests.@{prefix}.ignored.path" />
</testsuite:javaTestsuite>
```

The `htmlReport` Task

This task allow the user to transform a given path containing a sample of JUnit reports to an HTML detailed report. Here is the attributes to fill:

- A nested `fileset` element containing all the JUnit reports of each test. Take care to exclude the final JUnit report generated by the test suite.
- 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 test suite. If the format is set to `frames` , with a non-Java MicroEJ Test Suite, the name of the links will not be relevant because of the non-existency of packages.

Listing 14: 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 Test Suite 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 test suite engine.

Specific Custom Properties

Some custom properties are specifics and retrieved from the test suite 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 Test Suite is a Java test suite, 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 Test Suite is used.

5.15.6 Heap Usage Monitoring

Introduction

When building a *Standalone Application*, the Java heap size must be specified as an *Application Option* (see *Option(text): Java heap size (in bytes)*). The value to set in this option depends on the maximum heap usage, and the developer can estimate it by running the application.

The Core Engine provides a Java API to introspect the heap usage at runtime. Additionally, heap usage monitoring can be enabled to compute the maximum heap usage automatically.

Here are the descriptions of the different notions related to heap usage:

- **Heap:** memory area used to store the objects allocated by the application.
- **Heap Size:** current size of the heap.
- **Maximum Heap Size:** maximum size of the heap. The heap size cannot exceed this value. See *Option(text): Java heap size (in bytes)*.
- **Heap Usage:** the amount of the heap currently being used to store alive objects.
- **Garbage Collector (GC):** a memory manager in charge of recycling unused objects to increase free memory.

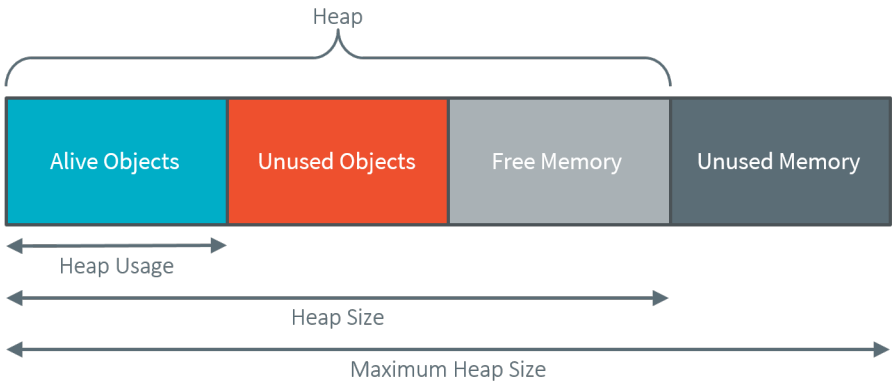


Fig. 39: Heap Structure Summary

The Java class java.lang.Runtime defines the following methods:

- gc(): Runs the garbage collector. System.gc() is an alternative means of invoking this method.
- freeMemory(): Returns the amount of free memory in the heap. This value does not include unused objects eligible for garbage collection. Calling the gc() method may result in increasing the value returned by this method.
- totalMemory(): Returns the current size of the heap. The value returned by this method may vary over time.
- maxMemory(): Returns the maximum size of the heap.

Heap Usage Introspection

The methods provided by the Runtime class allow introspecting the heap usage by comparing the heap size and the free memory size. A garbage collection must be executed before computing the heap usage to recycle all the unused objects and count only alive objects.

The application can compute the current heap usage by executing the following code:

```
Runtime runtime = Runtime.getRuntime(); // get Runtime instance
runtime.gc(); // Ensure unused objects are recycled
long heapUsage = runtime.totalMemory() - runtime.freeMemory();
```

This example gives the heap usage at a given point but not the maximum heap usage of the application.

Note: When heap usage monitoring is disabled, the heap size is fixed, and so totalMemory() and maxMemory() return the same value.

Automatic Heap Usage Monitoring

The maximum heap usage of an application's execution can be computed automatically by enabling heap usage monitoring.

Note: This feature is available in the Architecture versions 7.16.0 or higher for the Applications deployed on hardware devices (not on Simulator).

When this option is activated, an initial size for the heap must be specified, and the Core Engine increases the heap size dynamically. The value returned by totalMemory() is the current heap size. maxMemory() returns the maximum size of the heap. A call to gc() decreases the heap size to the higher value of either the heap usage or the initial heap size.

At any moment, totalMemory() returns the maximum heap usage of the current execution (assuming the maximum heap usage is higher than the initial heap size, and gc() has not been called).

See the section *Option(checkbox): Enable Java heap usage monitoring* to enable this option and configure the initial heap size.

Even if the heap size can vary during time, a memory section of maxMemory() bytes is allocated at link time or during the Core Engine startup. No dynamic allocation is performed when increasing the heap size.

Warning: A small initial heap size will impact the performances as the GC will be executed every time the heap size needs to be increased.

Furthermore, the smaller the heap size is, the more frequent the GC will occur. This feature should be used only for heap usage benchmarking.

Heap Usage Analysis

To analyze heap usage and see what objects are alive in the application, use the Heap Dumper & Heap Analyzer tools (*on SDK 6*, *on SDK 5*).

CHAPTER

SIX

VEE PORTING GUIDE

6.1 Introduction

6.1.1 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."

6.1.2 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.

6.2 MicroEJ Platform

6.2.1 Introduction

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.

6.2.2 Build Process

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.

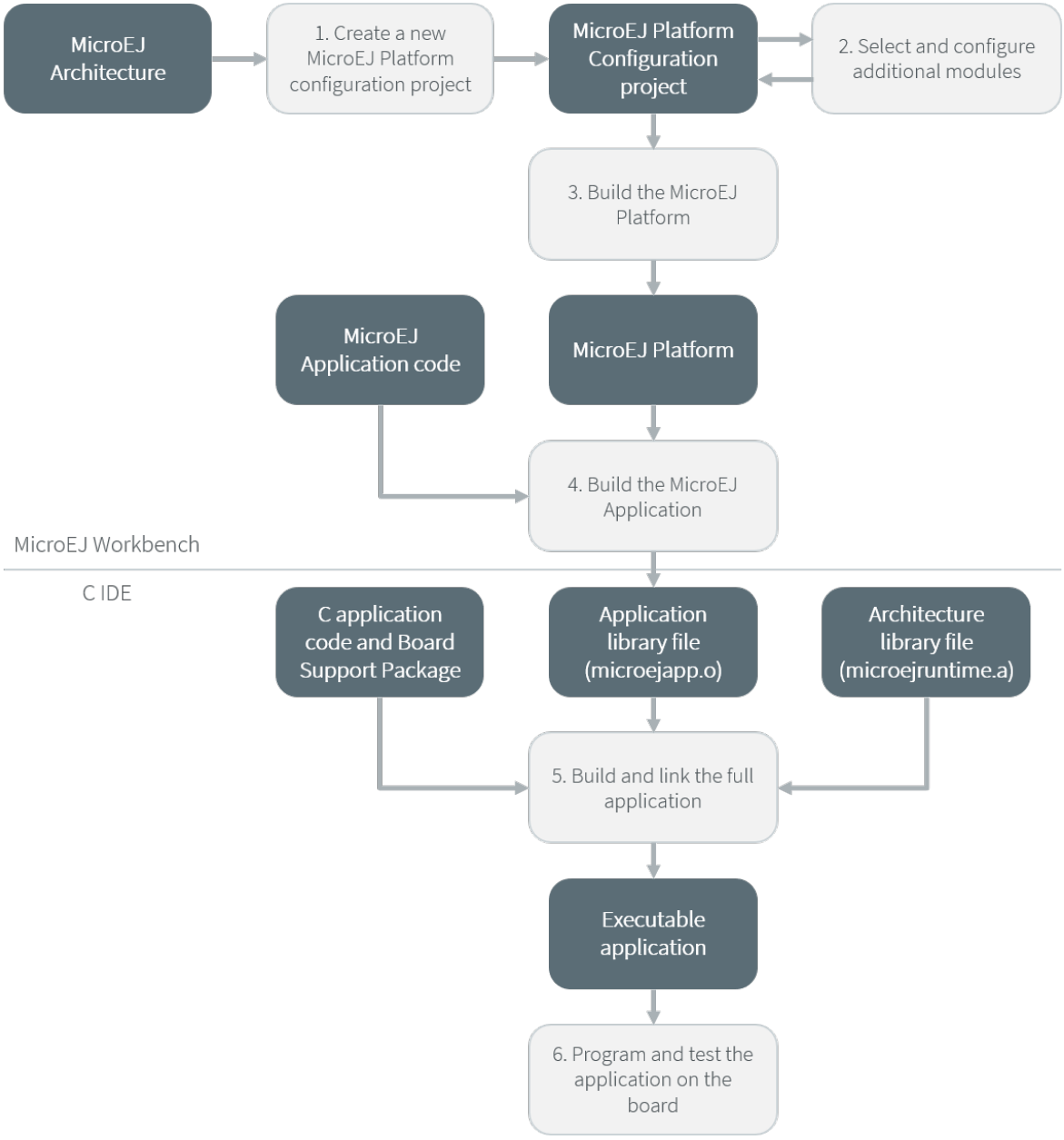


Fig. 1: Overall Process

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.

6.2.3 Concepts

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:

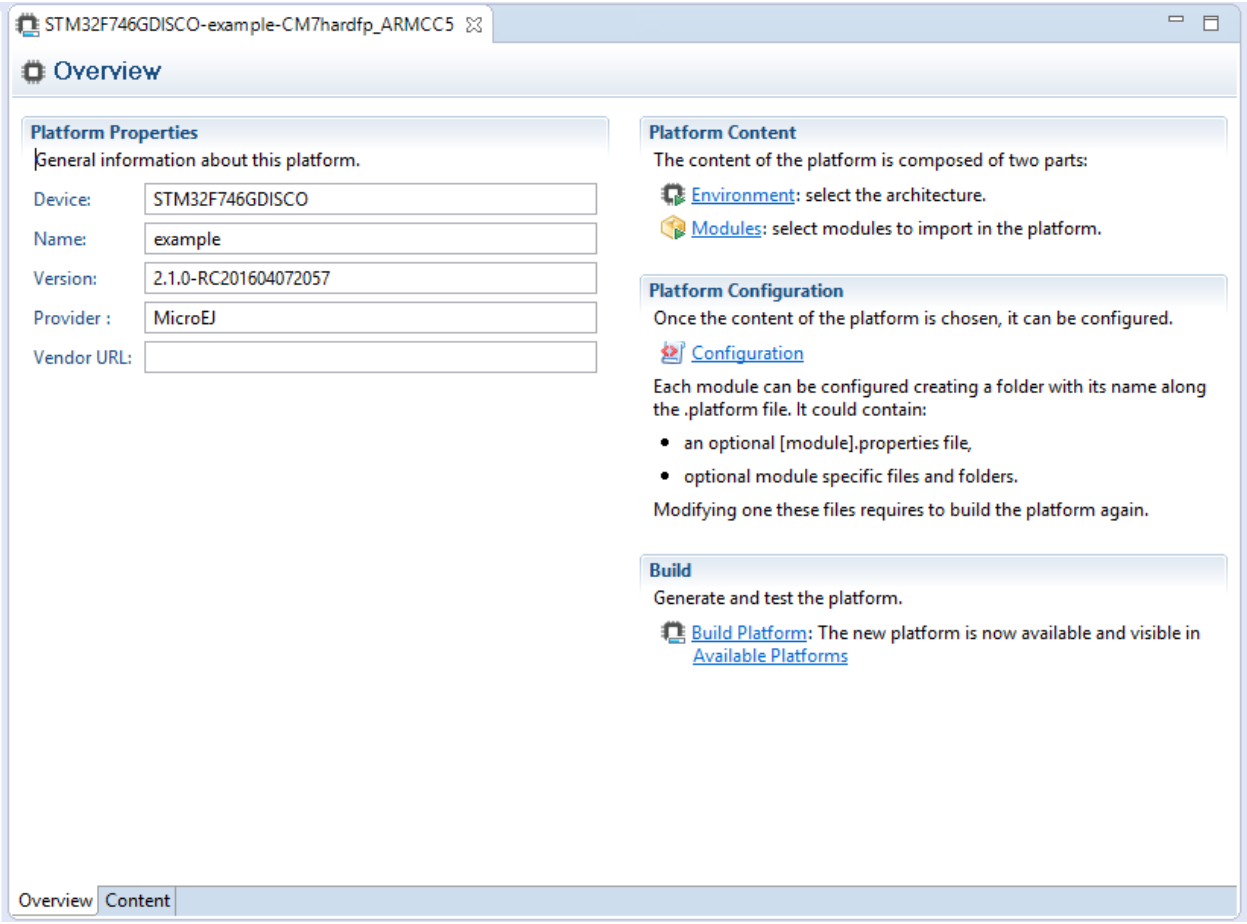


Fig. 2: MicroEJ Platform Configuration Overview Tab

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

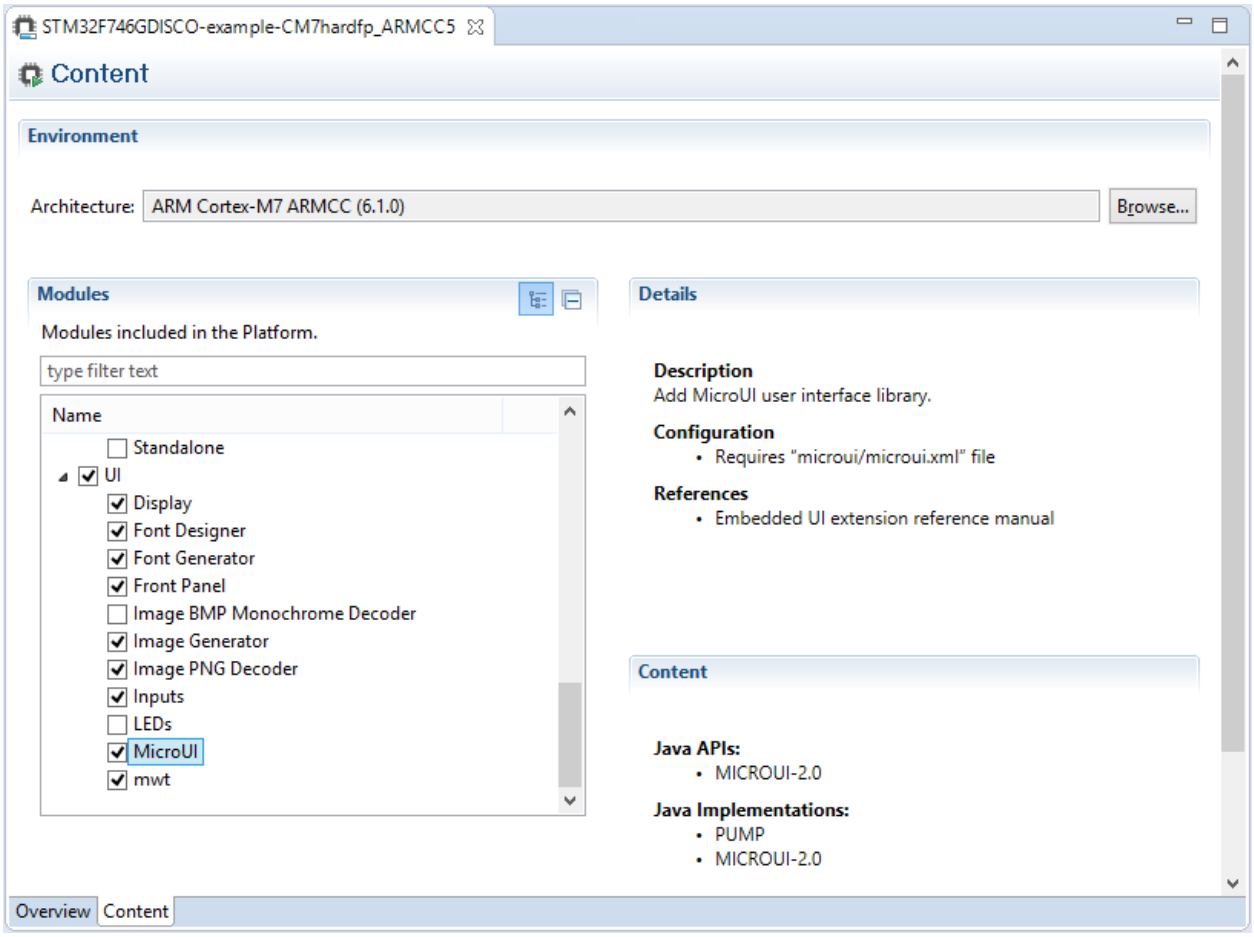


Fig. 3: MicroEJ Platform Configuration Content Tab

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 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 has to supply the C code that links a platform component to the target hardware, a *Low Level API* is defined. There is a standard pattern for the definition and implementation of these APIs. Each interface has a name and is specified by two header files:

- `[INTERFACE_NAME].h` specifies the functions that make up the public API of the implementation. In some cases the user code will never act as a client of the API, and so will never use this file.
- `[INTERFACE_NAME]_impl.h` specifies the functions that must be coded by the user in the implementation.

The user creates *implementations* of the interfaces, each captured in a separate C source file. In the simplest form of this pattern, only one implementation is permitted, as shown in the illustration below.

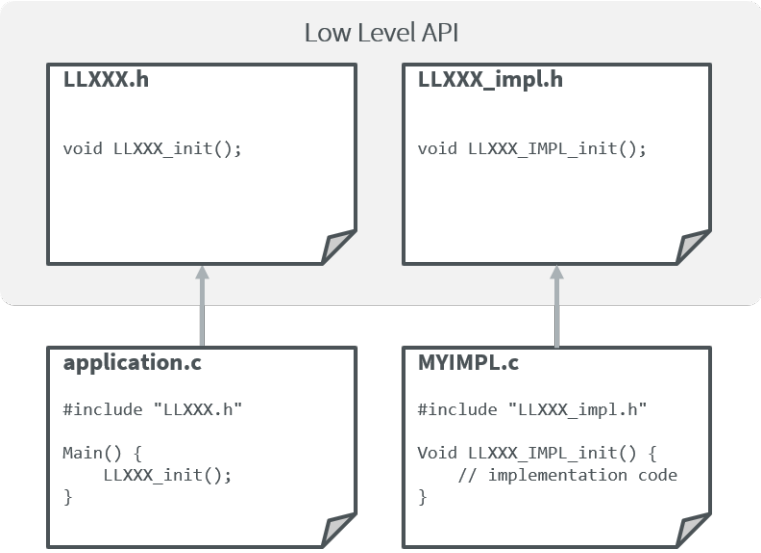


Fig. 4: Low Level API Pattern (single implementation)

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.

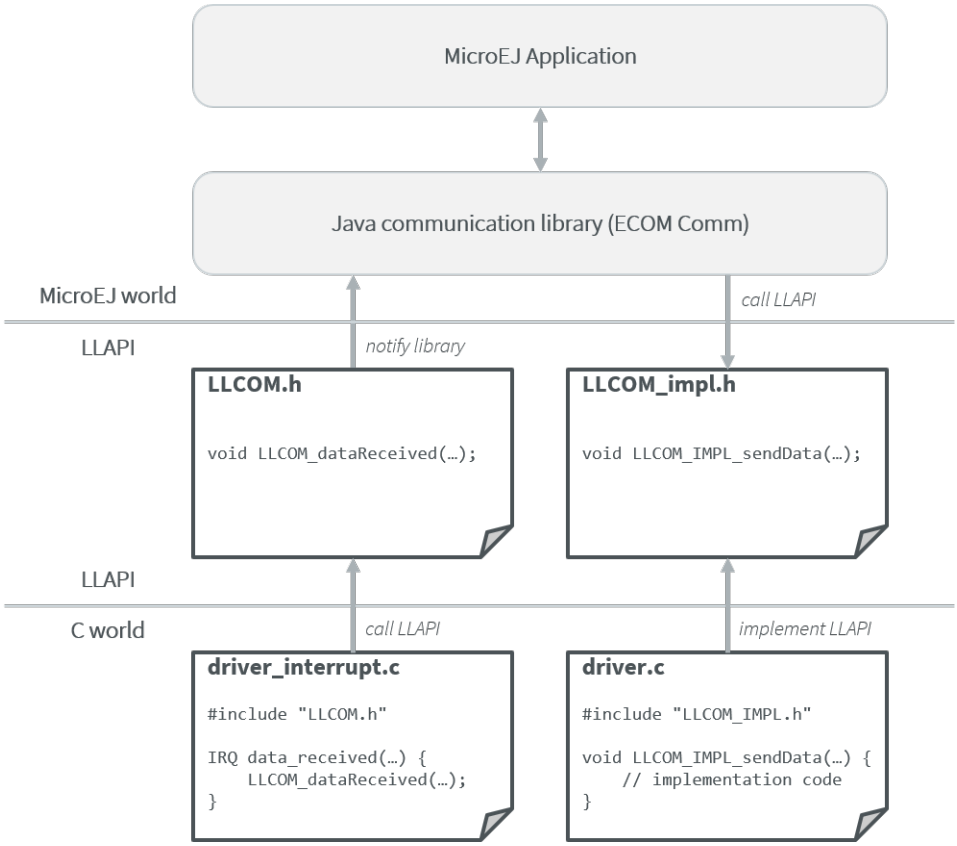


Fig. 5: Low Level API Example

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.

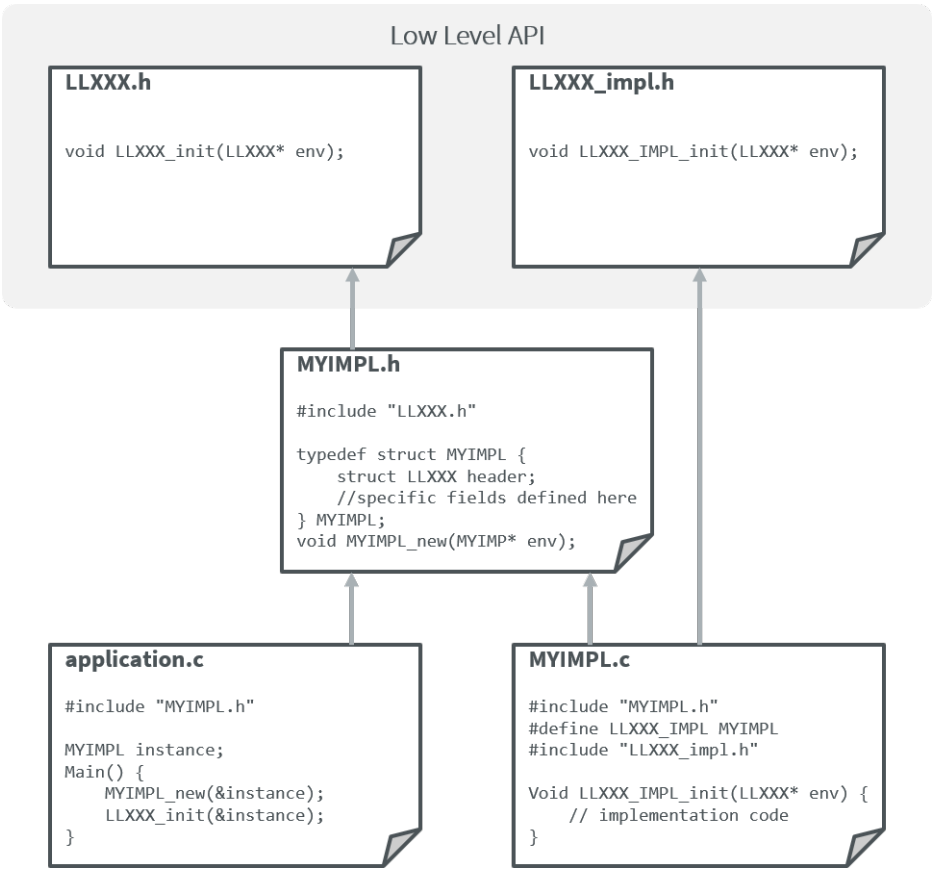


Fig. 6: Low Level API Pattern (multiple implementations/instances)

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

6.3 MicroEJ Architecture

MicroEJ Architecture features the MicroEJ Core Engine built for a specific instructions set (ISA) and compiler.

The MicroEJ Core Engine is a tiny and fast runtime associated with a Scheduler and a Garbage Collector.

MicroEJ Architecture provides implementations of the following Foundation Libraries :

- Embedded Device Configuration (see *[EDC]*).
- Beyond Profile (see *[BON]*).
- Simple Native Interface (see *[SNI]*).
- Kernel & Features (see *[KF]*).
- Shielded Plug (see *[SP]*).

The following figure shows the components involved.

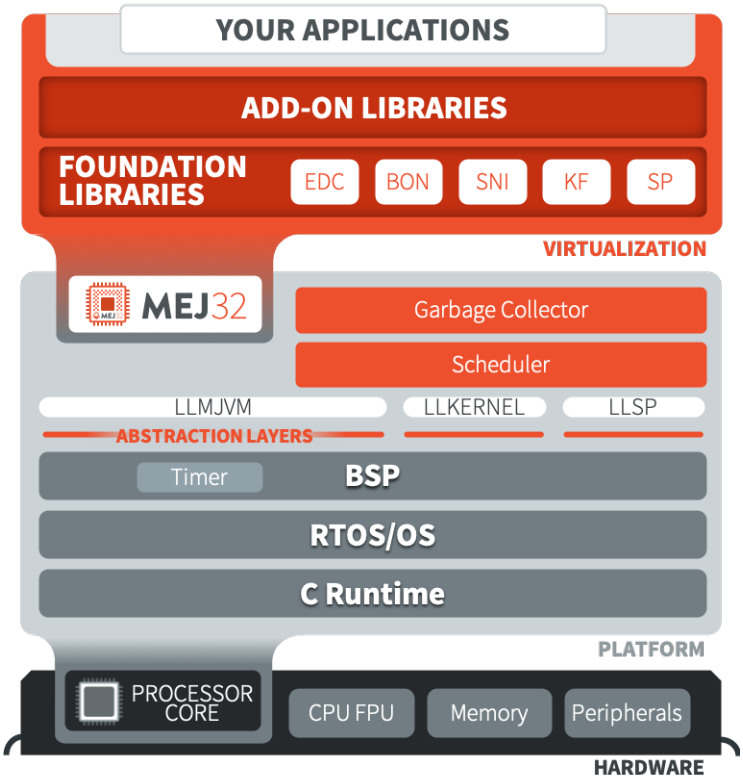


Fig. 7: MicroEJ Architecture Modules

Three Low Level APIs allow the MicroEJ Architecture to link with (and port to) external code, such as any kind of RTOS or legacy C libraries:

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

For further information on Architecture installation and releases, you can check these chapters:

6.3.1 Naming Convention

MicroEJ Architecture files ends with the `.xpf` extension, and are classified using the following naming convention:

```
com/microej/architecture/[ISA]/[TOOLCHAIN]/[UID]/[VERSION]/[UID]-[VERSION]-[USAGE].xpf
```

- `ISA` : instruction set architecture (e.g. `CM4` for Arm® Cortex®-M4, `ESP32` for Espressif ESP32, ...).
- `TOOLCHAIN` : C compilation toolchain (e.g. `CM4hardfp_GCC48`).
- `UID` : Architecture unique ID (e.g. `flopi4G25`).
- `VERSION` : module version (e.g. `7.12.0`).
- `USAGE` = `eval` for evaluation Architectures, `prod` for production Architectures.

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

See *Platform Configuration* for usage.

6.3.2 Architectures Changelog

Notation

A line prefixed by `[]` describes a change that only applies on a specific configuration: `[Core Engine Capability/Instruction Set/C Compiler]`:

- Core Engine Capability
  - `Mono` : Mono-Sandbox (default)
  - `Tiny` : Tiny-Sandbox
  - `Multi` : Multi-Sandbox
- Instruction Set
  - `ARM9` : ARM ARM9
  - `Cortex-A` : ARM Cortex-A
  - `Cortex-M` : ARM Cortex-M
  - `ESP32` : Espressif ESP32
  - `RX` : Renesas RX
  - `x86` : Intel x86
- C Compilation Toolchain
  - `ARMCC5` : Keil ARMCC uVision v5. See also *ARM Linker Specific Options*.
  - `Clang` : Clang
  - `GCC` : GNU GCC Compiler. See also *GNU LD Specific Options*.
  - `IAR` : IAR Embedded Workbench for ARM. See also *IAR Linker Specific Options*.
  - `QNX65` : BlackBerry QNX 6.5
  - `QNX70` : BlackBerry QNX 7.0

[8.1.0] - 2023-12-22

This Architecture version update introduces the following main features:

- Updated *Feature installation* flow to support Code chunks. A Feature can now be installed to ROM without the need of the Code size in RAM.
- Support for debugging ASLR Executables
- Support for debugging MCU targets
- Support for debugging Multi-Sandbox Executables
- Updated the options to select the *Core Engine capability*. See *Migrate Core Engine Capability Configuration*.
  - Added the VEE Port option `com.microej.runtime.capability`
  - Removed the `Multi Applications` module from the platform configuration file

  - Value of the `BON` constant `com.microej.architecture.capability` is now `mono` instead of `single` when the Core Engine capability is Mono-Sandbox.

- Support of THALES Sentinel License Manager
- Added a default application for early-stage VEE Port integration without the need of a SDK license.

If you plan to migrate a VEE Port from Architecture `8.0.0` to Architecture `8.1.0`, consider the *Architecture 8.0.0 Migration* chapter.

Core Engine

- Added option *com.microej.runtime.core.gc.markstack.levels.max* to configure the maximum number of elements of the Garbage Collector's mark stack.
- In `sni.h`, clarified the behavior of `SNI_createVM()`, `SNI_startVM()`, and `SNI_destroyVM()` when restarting the Core Engine. See also the *Core Engine implementation* section.
- Fixed missing default initialization of the options *core.memory.javaheap.size* and *core.memory.immortal.size*.
- [Multi] - Added a check when `LLKERNEL_IMPL_getFeatureHandle()` returns `0`. Corresponding error code is `LLKERNEL_FEATURE_INIT_ERROR_NULL_HANDLE`.
- [Multi] - Removed Feature installation in RAM (legacy *In-Place Installation mode*). See *Migrate Your LLKERNEL Implementation*.
- [Multi] - Updated *Feature installation boot sequence*: all Feature handles are now retrieved prior to initializing them.
- [Multi] - Updated check of *Kernel UID* at the beginning of Kernel.install(java.io.InputStream), before allocating Feature sections.
- [Multi] - Updated the specification for `LLKERNEL_IMPL_allocateFeature()` function to return the handle `0` if the Feature could not be allocated.
- [Multi] - Updated the specification for `LLKERNEL_IMPL_getAllocatedFeaturesCount()` function to ensure that it returns a valid result at any time, even if it is only called by the Core Engine during startup.
- [Multi] - Updated the specifications for `LLKERNEL_IMPL_getFeatureAddressRAM()` and `LLKERNEL_IMPL_getFeatureAddressROM()` functions to return `NULL` when an incorrect index is provided. This change is only for `LLKERNEL` TCK purposes, as the Core Engine only invokes these methods with valid indices.
- [Multi] - Added an option to enable *RAM Control* at VEE Port build (disabled by default).

Foundation Libraries

- Fixed, in `BON`, ResourceBuffer.readString() which does not increment correctly the position in the buffer.
- Fixed, in `BON`, `-1` returned by ResourceBuffer.available() instead of `0` when the end of the buffer is reached.
- Fixed, in `BON`, invalid value returned by ResourceBuffer.available() on the Simulator.
- Fixed, in `BON`, potential crash when calling ResourceBuffer.close() several times on a `ResourceBuffer` loaded with the *External Resources Loader*.

Integration

- Updated Architecture End User License Agreement to version `SDK 3.1-B`.
- Removed warning messages related to missing KF options when running the SOAR or the Simulator in Mono-Sandbox.

Simulator

- Added compatibility with IntelliJ IDEA IDE to debug applications.
- Added message when waiting for a connection in debug mode.
- Fixed debugger verbose mode.
- Removed bootstrap thread from the debugger vision.
- Fixed debugger suspend count on threads handling.
- Fixed stop issue on static method entry breakpoint.

SOAR

- Fixed trimming of leading or trailing spaces in immutable strings
- [Multi] - Fixed integration of the *bytecode verifier* in Feature mode.
- [Multi] - Improved the error message thrown when no Feature definition file is found and displayed the class-path to better guide developers in identifying potential causes.

Tools

- Updated SOAR and VM Model Readers
  - Added support to retrieve the Core Engine memory regions (used by the VEE Debugger Proxy to generate a memory dump script (see *Generate VEE memory dump script*))
  - Added an API to relink the SOAR Model objects, i.e. change their associated addresses (used by the VEE Debugger Proxy to support ASLR Executables debug)
  - Added new APIs to load Kernel and Features SOAR Model objects (used by the VEE Debugger Proxy to support Multi-Sandbox Executable debug)
- [ARMCC5] - Fixed *SOAR Debug Infos Post Linker* tool to throw a dedicated error when the SOAR object file does not contain the debug section.
- [Multi] - Fixed missing first null entry in the symbol table generated by the Firmware Stripper.

[8.0.0] - 2023-06-27

Note: This Architecture requires SDK version `5.7.0` or higher (see *SDK Version*).

This major Architecture version update introduces the following main features:

- Added compatibility with dynamic linkers enabling Address Space Layout Randomization (ASLR).
- Added *Feature build on device*. For that, the SOAR has been deeply redesigned and split into multiple phases. The most noticeable change is about the *SOAR Information File* that is now composed of 3 files.
- Added Feature portability. The same `.fo` file can now be installed:
  - On any Executable built from the same Kernel Application (`microejapp.o`). The VEE Port C code can be modified and relinked without requiring to rebuild the `.fo` file anymore.
  - On different Kernel Applications provided some conditions are met. Basically, a `.fo` built on Kernel 1 can be installed on Kernel 2 if the exposed Kernel APIs are left unchanged. See *Feature Portability Control* for more details.
- Redesigned Feature installation flow. A Feature can now be installed in any byte-addressable memory mapped to the CPU's address space, including ROM. For that, `LLKERNEL` Low Level APIs have been fully rewritten. See *Feature installation* for more details. Former Feature installation in RAM is preserved and is now called *In-Place Installation*. Former static Feature installed by the SDK (using the Firmware Linker tool) is removed in favor of *Feature persistency* at boot.

If you plan to migrate a VEE Port from Architecture `7.x` to Architecture `8.x` , consider the *Architecture 7.x Migration* chapter.

Core Engine

- Renamed *Core Engine sections* to fully respect the ELF standard naming convention.
- Removed check when passing a non-immortal array in SNI if VEE Port option `core.sni.nonimmortal.access` was set to `false` .
- Removed `LLBSP_isInReadOnlyMemory` in Core Engine Abstraction Layer (`LLBSP.h` file).
- Clarified `LLMJVM_IMPL_getCurrentTime` API contract in Core Engine Abstraction Layer (`LLMJVM_impl.h` file).
- Updated `Trace` C library from version `1.0.0` to `2.0.0` . See *Migrate Trace C Library Usage*.
  - Renamed header file `trace.h` into `LLTRACE.h` to avoid filename conflicts.
  - Renamed C functions `TRACE_xxx` into `LLTRACE_xxx` .
- Fixed potential crash when Core Engine is restarted after a call to System.exit(int).
- [Multi] - Added option *com.microej.runtime.kernel.dynamicfeatures.max* to configure the maximum number of Features that can be dynamically installed.
- [Multi] - Added option *com.microej.runtime.kf.waitstop.delay* to configure the maximum time allowed for a Feature to stop.
- [Multi] - Fixed missing release of allocated Feature buffers after Core Engine exits (*In-Place Installation* mode).

Foundation Libraries

- Updated KF to version 1.7 :
  - Added heap memory control: Module.getAllocatedMemory(), Kernel.setReservedMemory() and Feature.setMemoryLimit() methods.
  - Added load of a Feature resource (Feature.getResourceAsStream() method).
- Updated KF dynamic loader to support the new *Feature installation* flow.
- Removed Foundation Libraries API Jars and Javadoc.
- Removed *Unknown product - Unknown version* comment in auto-generated Low Level API header files.
- Removed the Serial Communication modules group, including the Foundation Libraries ECOM and ECOM-COMM . See *Migrate ECOM-COMM Module*.
- Removed the deprecated Device Information module group, including the Foundation Library Device . See *Migrate Device Module*.
- Fixed *Option(checkbox): Embed UTF-8 encoding* defaults to true when building a Standalone Application using MMM.
- Fixed KF to call the registered Thread.UncaughtExceptionHandler when an exception is thrown in FeatureEntryPoint.stop().
- Fixed unexpected java.lang.NullPointerException thrown by the skip method of an InputStream returned by Class.getResourceAsStream(). This error only occurs with a resource loaded by the External Resource Loader.
- Fixed the behavior of available , read , skip , mark , reset and close methods of an InputStream returned by Class.getResourceAsStream() and previously closed.
- Fixed the LLEXT_RES_read() Low Level API specification (the buffer passed cannot be null).
- [Mono] Fixed an unexpected FeatureFinalizer exception or infinite loop when a Standalone Application touches a KF API in some cases.
- [Tiny] Fixed an unexpected SOAR error when a Standalone Application touches a KF API.
- [Multi] Fixed exception thrown when calling Kernel.removeConverter().
- [Multi] Fixed an unexpected NullPointerException thrown by ej.kf.Kernel.<clinit> method in some cases.
- [Multi] Fixed KF watchdogs not triggered correctly when several expire at the same time.

Integration

- Added support for resolving *Front Panel in Workspace* before the included Front Panel.
- Added Memory Map Scripts for Eclasspath Math , Formatter and DateFormat .
- Updated default value of VEE Port configuration option vendorURL .
- Updated Memory Map Scripts for MicroVG library.
- Updated Memory Map Scripts for Eclasspath Executor library.
- Updated output Map file location to soar/[application_main_class].map (formerly named SOAR.map).
- Removed unused SOAR.o file. It is available at bsp/microejapp.o .
- Renamed MicroEJ launch Build dynamic Feature to Build Feature .

- [Multi] Fixed the SOAR output files from being deleted when the `Clean intermediate files` option is enabled.

Simulator

- Added *Mock debug* mode.
- Added missing default values for the properties `s3.slow`, `console.logs.period`, and `s3.hil.timeout` when launching the Simulator from the command line.
- Added a check for unsupported access to the Class instance of a primitive type (e.g. `byte.class`).
- Added HIL Engine debug logs when verbose option is enabled.
- Added log of the Mock classpath when verbose option is enabled.
- Added log of Mock resolution errors (class or method not found).
- Added support for mark/reset on an InputStream returned by Class.getResourceAsStream().
- Fixed "Internal limits" error in HIL engine when too many array arguments are used at the same time by one or several native methods.
- Fixed slow reading with an array of bytes of the input stream returned by Class.getResourceAsStream(String).
- Fixed configuration of the Java heap size using *Option(text): Java heap size (in bytes)*. The legacy `core.memory.javaheapsum.size` option is not more supported.
- Fixed *Option(text): Immortal heap size (in bytes)* default value when running a Standalone Application using MMM.
- Fixed stop of the HIL Engine if Simulator was terminated before the connection is established.
- Fixed load of the Mock classes in the classpath order (left-to-right).
- Fixed the missing error check when loading an immutable file referencing an external object id (the `importObject` directive is required).
- Fixed initialization of transparent images in the Front Panel when the initial color is not fully opaque. (introduced in version *7.11.0*)
- [Multi] Fixed the computation of object sizes. The 4-byte KF header was missing.

SOAR

- Added support for *Resource* alignment constraint.
- Added a check for legacy `.system.properties` files in the *Application Classpath*. The build process is stopped and an error is reported. See *Migrate Legacy System Properties Files*.
- Added a check for unsupported access to the Class instance of a primitive type (e.g. `byte.class`).

Tools

- Updated the serial PC connector to JSSC `2.9.4`, including support for macOS aarch64 (M1 chip).
- Removed *Test Suite Engine*. If needed, the Test Suite Engine is available in the *Build Kit*.
- Removed Immutables NLS library. Use *Binary NLS* add-on library instead.
- Fixed an incorrect generation of a debug file beside the memory file when launching the Heap Dumper.
- [Multi] Added Heap Dumper support for dynamically installed Features.

[7.20.1] - 2023-04-10

Foundation Libraries

- Fixed `Float.parseFloat(...)` and `Double.parseDouble(...)` that don't throw a `NumberFormatException` when the given string is empty.
- Fixed float and double to string conversions that contain an unecessary `+` sign in the exponent.

[7.20.0] - 2023-04-04

Known Issues

- `Float.parseFloat(...)` and `Double.parseDouble(...)` don't throw a `NumberFormatException` when the given string is empty.
- Float and double to string conversions contain an unecessary `+` sign in the exponent.

Core Engine

- Added the capability to customize implementation of the function that performs an atomic exchange operation.
- [ESP32] - Remove default implementation of the function that performs an atomic exchange operation. The Core Engine abstraction layer implementation has to implement the C function `int32_t LLBSP_IMPL_atomic_exchange(int32_t* ptr, int32_t value)`.

Foundation Libraries

- Fixed uninitialized pointer access in the *External Resources Loader*, which can cause a system crash when reading data from a resource.

[7.19.0] - 2023-02-16

Known Issues

- Float.parseFloat(...) and Double.parseDouble(...) don't throw a NumberFormatException when the given string is empty.
- Float and double to string conversions contain an unecessary `+` sign in the exponent.

Core Engine

- Added the capability to customize implementation of the functions that convert strings to float/double values and vice-versa.
- [Cortex-A/Clang] - Fixed wrong float/double arguments passed to the SNI natives.

Tools

- Removed dependency on GNU `ar` program to create `microejruntime.a` archive file.

[7.18.1] - 2022-10-26

Integration

- Fixed License Manager issue with JDK 8u351 or higher (`[M65] - License check failed [tampered (3)].`).

[7.18.0] - 2022-09-14

Integration

- Added support for Windows 11.
- Added License Manager support for macOS aarch64 (M1 chip).
- Removed warning when launching Applications or Tools with JDK 11 (*Warning: Nashorn engine is planned to be removed from a future JDK release*).

SOAR

- Added grouping of all immutables objects in a single ELF section.

[7.17.0] - 2022-06-13

Core Engine

- Fixed potential premature evaluation timeout when Core Engine is not started at the same time as the device.
- Fixed potential crash during the call of `LLMJVM_dump` when printing information about the Garbage Collector.
- Added new functions to Low Level API `LLMJVM_MONITOR_impl.h` (see *Advanced Event Tracing*):
  - `void LLMJVM_MONITOR_IMPL_on_invoke_method(void* method)` : called by the Core Engine when an method is invoked.
  - `void LLMJVM_MONITOR_IMPL_on_return_method(void* method)` : called by the Core Engine when a method returns.
- [Cortex-M] - Added support for MCU configuration with unaligned access traps enabled (`UNALIGN_TRP` bit set in `CCR` register).

Foundation Libraries

- Updated `KF` to version `1.6` :
  - Added Kernel.canUninstall() method.

Integration

- Fixed some Architecture tools compatibility issues with SDKs running on JDK 11.
- Fixed missing default value for ShieldedPlug server port when running it with MMM (`10082`).
- Updated Memory Map Scripts for `ej.microvg` library.
- Updated Architecture End User License Agreement to version `SDK 3.1-A` .

Simulator

- Added class file major version check (<=51). Classes must be compiled for Java 7 or lower. Set the options property `S3.DisableClassFileVersionCheck` to `false` to disable this verification.
- Added native method signature in the stack trace of the UnsatisfiedLinkError thrown when a native method is missing.
- Fixed HIL engine method `NativeInterface.getResourceContent()` that generates a runtime error in the Simulator.
- Fixed error “Internal limits reached … S3 internal heap is full” when repeatedly loading a resource that is available in the classpath but not referenced in a `.resources.list` file.
- Fixed OutOfMemoryError when loading a large resource with Class.getResourceAsStream().
- Fixed `A[].class.isAssignableFrom(B[].class)` returning `false` instead of `true` when `B` is a subclass of `A` .
- Fixed potential “Internal limits reached” error when an OutOfMemoryError is thrown.
- Fixed error “Cannot pin objects anymore” when passing repeatedly immutable objects to a native method.

- Fixed properties not passed correctly to the mocks when the Virtual Device is executed from a path that contains spaces.
- [Multi] - Fixed an unexpected error when `kernel.kf` file is missing and KF library is used: "Please specify a 'kernel.kf' file to enable Kernel & Features semantics."
- [Multi] - Fixed type `double[]` not recognized in `kernel.api` file.

SOAR

- Fixed internal error when using a BON constant in an if statement at the end of a `try` block.
- Fixed internal error when a `try` block ends with an `assert` expression while assertions are disabled.
- [Multi] - Raise a warning instead of an error when duplicated `.kf` files are detected in the Kernel classpath. Usual classpath resolution order is used to load the file (see *MicroEJ Classpath*).
- [Multi] - Fixed SOAR error when building a Feature that uses an array of basetypes that is not explicitly declared in the `kernel.api` file of the Kernel.
- [Multi] - Optimized "Build Dynamic Feature" scripts speed by removing unnecessary steps.

[7.16.3] - 2022-04-06

Core Engine

- [Cortex-M/IAR] Fix unaligned stack pointer when calling SNI native functions in ARM IAR architectures.

[7.16.2] - 2021-11-10

Core Engine

- [Cortex-M/GCC/ARMCC5] Fix unaligned stack pointer when calling SNI native functions in ARM GCC and ARMCC architectures with non-ASM Core Engines.

[7.16.1] - 2021-07-16

Core Engine

- [GCC] Fixed wrong inlined extern symbol access (affects only some GCC architectures until version `6.x`). This produces an unexpected `java.lang.OutOfMemoryError: Stacks space` exception at boot time.

[7.16.0] - 2021-06-24

Known Issues

- [Multi] - SOAR may fail to build a Feature with the following message:

```
1 : KERNEL/FEATURE ERROR
    [M25] - Type double[] is expected to be owned by the Kernel but is not embedded.
```

  Workaround is to explicitly declare each array of basetypes in your `kernel.api` file:

```
<type name="int[]"/>
<type name="long[]"/>
<type name="short[]"/>
<type name="double[]"/>
<type name="float[]"/>
<type name="byte[]"/>
<type name="char[]"/>
<type name="boolean[]"/>
```

Notes

The `Device` module provided by the Architecture is deprecated and will be removed in a future version. It has been moved to the Device Pack. Please update your VEE Ports.

Core Engine

- Added a dedicated error code `LLMJVM_E_INITIALIZE_ERROR (-23)` when `LLMJVM_IMPL_initialize()`, `LLMJVM_IMPL_vmTaskStarted()`, or `LLMJVM_IMPL_shutdown()` fails. Previously the generic error code `LLMJVM_E_MAIN_THREAD_ALLOC (-5)` was returned.
- Added automatic heap consumption fing when option `com.microej.runtime.debug.heap.monitoring.enabled` is set to `true`
- Fixed some parts of `LLMJVM_checkIntegrity()` code were embedded even if not called
- [Multi] - Fixed potential crash during the call of `LLMJVM_checkIntegrity()` when analyzing a corrupted Java stack (make this function robust to object references with an invalid memory address)

Foundation Libraries

- Added source code for `KF`, `SCHEDCONTROL`, `SNI`, `SP` implementations
- Updated `KF` API with annotations for Null analysis
- Updated `SNI` API with annotations for Null analysis
- Updated `SP` API with annotations for Null analysis
- Updated `ResourceManager` implementation with annotations for Null analysis
- Updated `KF` implementation:
  - Added missing Kernel.getAllFeatureStateListeners() method
  - Updated code for correct Null analysis detection
  - Fixed Feature.getCriticality() to throw IllegalStateException if it is in state `UNINSTALLED` (instead of returning `NORM_CRITICALITY`)
  - Fixed potential race condition between Kernel.addResourceControlListener() and Kernel.removeResourceControlListener(). Adding a new listener may not register it if another one is removed at the same time.

Integration

- Added a new task in ELF Utils library allowing to update the content of an ELF section:
  - Declaration:

```
<taskdef classpath="${platform.dir}/tools/elfutils.jar" classname="com.is2t.elf.
↪utils.AddSectionTask" name="addSection" />
```

  - Usage:

```
<addSection file="${executable.file}" sectionFile="${section.file}" sectionName="$
↪{section.name}" sectionAlignment="${section.alignment}" outputDir="${output.dir}"␣
↪outputName="${output.name}" />
```

- Updated Architecture End User License Agreement to version `SDK 3.0-C`
- Updated copyright notice of Low Level APIs header files to latest SDK default license
- Updated Architecture module with required files and configurations for correct publication in a module repository (`README.md` , `LICENSE.txt` , and `CHANGELOG.md`)

Simulator

- Added an option (`com.microej.simulator.hil.frame.size`) to configure the HIL engine max frame size
- Fixed load of an immutable byte field (sign extension)
- Fixed java.lang.String constructors `String(byte[] bytes, ...)` when passing characters in the range `[0x80,0xFF]` using default `ISO-8859-1` encoding
- Fixed potential crash in debug mode when a breakpoint is set on a field access (introduced in version `7.13.0`)
- Fixed wrong garbage collection of an object only referenced by an immortal object

SOAR

- Fixed the following compilation issues in `if` statement with BON constant:
  - too many code may be removed when the block contains a `while` loop
  - potential `Stacks merging coherence error` may be thrown when the block contains a nested `try-catch` statement
  - potential `Stacks merging coherence error` when declaring a ternary expression with Constants.getBoolean() in condition expression
- Fixed `assert` statement removal when it is located at the end of a `then` block: the `else` block may be executed instead of jumping over
- Removed names of arrays of basetype unless `soar.generate.classnames` option is set to `true`
- [Multi] - Fixed potential link exception when a Feature use one of the `ej_bon_ByteArray` methods (e.g. `ej.kf.InvalidFormatException: code=51:ON_ej_bon_ByteArray_method_readUnsignedByte_AB_I_I`)
- [Multi] - Fixed SOAR error (`Invalid SNI method`) when one of the ej.bon.Constants.getXXX() methods is declared in a `kernel.api` file. This issue was preventing from using BON Constants in Feature code.

Tools

- Updated Code Coverage Analyzer report generation:
  - Automatically configure `src/main/java` source directory beside a `/bin` directory if available
  - Added an option (`cc.src.folders`) to specify the source directory (require SDK `5.4.1` or higher)
  - Removed the analysis of generated code for `synchronized` statements
  - Fixed crash when loading source code with annotations
- Fixed Memory Map scripts: `ClassNames` group may contain duplicate sections with `Types` group
- Fixed load of an ELF executable when a section overlaps a segment (updated ELF Utils, Kernel Packager and Firmware Linker)
- Fixed Firmware Linker to generate output executable file at the same location than the input executable file

[7.15.1] - 2021-02-19

SOAR

- [Multi] - Fixed potential VM crash when declaring a Proxy class which is `abstract` .

[7.15.0] - 2020-12-17

Core Engine

- Added support for applying Feature relocations

Foundation Libraries

- Updated `KF` implementation to apply Feature relocations using the Core Engine. The former Java implementation is deprecated but can still be enabled using the option `com.microej.runtime.kf.link.relocations.java.enabled` .

Integration

- Updated the Architecture naming convention: the usage level is `prod` instead of `dev` .
- Fixed generation of temporary properties file with a `.properties.list` extension instead of deprecated `.system.properties` extension.

SOAR

- Fixed crash when declaring a clinit dependency rule on a class that is loaded but not embedded.

Tools

- Fixed Memory Map Script `All` graph creation to prevent slow opening of large `.map` file in Memory Map Analyzer.

[7.14.1] - 2020-11-30

Core Engine

- [Multi/x86/QNX7] - Fixed missing multi-sandbox version

Tools

- Fixed categories for class names and SNI library in Memory Map Scripts

[7.14.0] - 2020-09-25

Notes

The following set of Architecture properties are automatically provided as `BON` constants:

- `com.microej.architecture.capability=[tiny|single|multi]`
- `com.microej.architecture.name=[architecture_uid]`
- `com.microej.architecture.level=[eval|prod]`
- `com.microej.architecture.toolchain=[toolchain_uid]`
- `com.microej.architecture.version=7.14.0`

Note: Starting from *Architecture 8.1.0*, `com.microej.architecture.capability` constant is set to `mono` instead of `single` when the Core Engine capability is Mono-Sandbox.

The following set of VEE Port properties (customer defined) are automatically provided as `BON` constants:

- `com.microej.platform.hardwarePartNumber`
- `com.microej.platform.name`
- `com.microej.platform.provider`
- `com.microej.platform.version`
- `com.microej.platform.buildLabel`

Foundation Libraries

- Updated `EDC` UTF-8 encoder to support Unicode code points as supplementary characters
- Fixed java.lang.NullPointerException thrown when java.util.WeakHashMap.put() method is called with a `null` key (introduced in version *7.11.0*)

Integration

- Added all options starting with `com.microej.` prefix as `BON` constants
- Added all properties defined in `architecture.properties` as options prefixed by `com.microej.architecture.`
- Added all properties defined in `release.properties` as options prefixed by `com.microej.platform.`
- Added all properties defined in `script/mjvm.properties` as options prefixed by `com.microej.architecture.`
- Added an option (`com.microej.library.edc.supplementarycharacter.enabled`) to enable support for supplementary characters (enabled by default)
- Updated Memory Map Scripts to extract Java static fields in a dedicated group named `Statics`
- Updated Memory Map Scripts to extract Java types in a dedicated group named `Types`
- Fixed generated Feature filename (unexpanded `${feature.output.basename}` variable, introduced in version *7.13.0*)
- Fixed definition of missing default values for memory options (same values than launcher default ones)
- [Tiny,Multi] - Added display of the Core Engine capability when launching SOAR

SOAR

- [Multi] - Added a new attribute named `api` in Kernel `soar.xml` file indicating which types, methods and static fields are exposed as Kernel APIs
- [Multi] - Fixed potential link error when calling Object.clone() method on an array in Feature mode

Tools

- Updated the serial PC connector to JSSC `2.9.2` (COM port could not be open on Windows 10 using a JRE `8u261` or higher)

[7.13.3] - 2020-09-18

Core Engine

- [QNX70] - Embed method names and line numbers information in the application
- [Cortex-A/QNX70] - Fixed wrong float/double arguments passed to the SNI natives (introduced in version *7.12.0*)

Simulator

- Fixed unnecessary stacktrace dump on Long.parseLong(...) error
- Fixed UTF-8 encoded Strings not correctly printed

Tools

- Updated Memory Map Scripts for `ej.library.runtime.basictool` library

[7.13.2] - 2020-08-14

Core Engine

- [ARM9/QNX65] - Fixed custom convention call
- [x86/QNX70] - Fixed SIGFPE raised when overflow occurs on division
- [x86/QNX70] - Fixed issue with NaN conversion to int or long

Tools

- Fixed Feature build script for SDK 5.x (introduced in version *7.13.0*)
- Updated Memory Map Scripts for MicroUI 3 and Service libraries

[7.13.1] - 2020-07-20

Core Engine

- [ESP32] - Fixed potential PSRAM access faults by rebuilding using esp-idf v3.3.0 toolchain (`simikou2`)

[7.13.0] - 2020-07-03

Core Engine

- Added `SNI-1.4` support, with the following new `LLSNI.h` Low Level APIs:
  - Added function `SNI_registerResource()`
  - Added function `SNI_unregisterResource()`
  - Added function `SNI_registerScopedResource()`
  - Added function `SNI_unregisterScopedResource()`
  - Added function `SNI_getScopedResource()`
  - Added function `SNI_retrieveArrayElements()`
  - Added function `SNI_flushArrayElements()`
  - Added function `SNI_isResumePending()`
  - Added function `SNI_clearCurrentJavaThreadPendingResumeFlag()`

  - Added define `SNI_VERSION`
  - Added define `SNI_IGNORED_RETURNED_VALUE`
  - Added define `SNI_ILLEGAL_ARGUMENT`
  - Updated the documentation of some functions to clarify the behavior
- Added a message to IllegalArgumentException thrown in an SNI call when passing a non-immortal array in SNI (only in case the VEE Port is configured to disallow the use of non-immortal arrays in SNI native calls)
- Added function `LLMJVM_CheckIntegrity()` to `LLMJVM.h` Low Level API to perform heap and internal structures integrity check
- Updated `KF` implementation to use `SNI-1.4` to close native resources when the Feature is stopped (`ej.lang.ResourceManager` is now deprecated)
- Updated `LLMJVM_dump()` output with the following new information related to `SNI-1.4` native resource management:
  - Last native method called (per thread)
  - Current native method being invoked (per thread)
  - Last native resource close hook called (per thread)
  - Current native resource close hook being invoked (per thread)
  - Pending Native Exception (per thread)
  - Pending `SNI` Scoped Resource to close (per thread)
  - Current Garbage Collector state: (running or not, last scanned object address, last scanned object class)
  - `LLMJVM` schedule request (global and per thread)
- Updated non-immortal array access from SNI default behavior (now allowed by default)
- Fixed thread state displayed by `LLMJVM_dump` for threads in `SLEEP` state
- Fixed `sni.h` header file function prototypes using the `SNI_callback` typedef
- Fixed crash when an OutOfMemoryError is thrown while creating a native exception in SNI
- [Multi] - Fixed runtime exceptions that can be implicitly thrown (such as NullPointerException) which were not automatically exposed by the Kernel
- [Multi] - Fixed passing Kernel array parameters through a shared interface method call. These parameters were passed by copy instead of by reference as specified by `KF` specification
- [Multi] - Fixed execution context when jumping in a catch block of a ej.kf.Proxy method (the catch block was executed in the Kernel context instead of the Feature context)
- [ARMCC5] - Fixed link error `Undefined symbol _java_Ljava_lang_OutOfMemoryError_field_OOMEMethodAddr_I` with ARM Compiler 5 linker (introduced in version *7.12.0*)

Foundation Libraries

- Updated `SNI` to version `1.4`
- Updated internal library `Resource-Manager-1.0` as deprecated. Use `SNI-1.4` native resources instead
- Updated Thread.getId() method implementation to return the same value than `SNI_getCurrentJavaThreadID()` function
- Optimized SNI.toCString() method by removing a useless temporary buffer copy
- Fixed `EDC` implementation of String(byte[],int,int) constructor which could allocate a too large temporary buffer
- Fixed `EDC` implementation of Thread.interrupt() method to throw a java.lang.SecurityException when the interrupted thread cannot be modified by the the current thread
- Fixed `EDC` implementation to remove remaining references to java.util.SecurityManager class when it is disabled
- Fixed `EDC` implementation of Thread.interrupt() method that was declared `final`
- Fixed `EDC` API of Thread.interrupt() to clarify the behavior of the method
- Fixed `EDC` API of java.util.Calendar method to specify that non-lenient mode is not supported
- Fixed `EDC` API of java.io.FilterInputStream.in field to be marked `@Nullable`

Integration

- Updated Architecture End User License Agreement to version `SDK 3.0-B`

Simulator

- Added `SNI-1.4` support, with the following new HIL engine APIs:
  - Added methods `NativeInterface.suspendStart()` and `NativeInterface.suspendStop()` to notify the simulator that a native is suspended so that it can schedule a thread with a lower priority
- Added `KF` support to dynamically install Features (`.fs3` files)
- Added the capability to specify the Kernel UID from an option (see options in `Simulator` > `Kernel` > `Kernel UID`)
- Added object size in generated `.heap` dump files
- Optimized file accesses from the Application
- Fixed crash in debug mode when paused on a breakpoint in SDK and hovering a Java variable with the mouse
- Fixed potential crash in debug mode when putting a breakpoint in the SDK on a line of code declared in an inner class
- Fixed potential crash in debug mode (java.lang.NullPointerException) when a breakpoint set on a field access is hit
- Fixed potential crash in debug mode (ArrayIndexOutOfBoundsException)
- Added support for JDWP commands `DisableCollection` / `EnableCollection` in the debugger
- Fixed invalid heap dump generation in debug mode.

- Fixed crash when a Mockup implements `com.is2t.hil.StartListener` and this implementation throws an uncaught exception in the clinit
- Fixed verbose of missing resource only when a resource is available in the classpath but not declared in a `.resources.list` file
- Fixed heap consumption simulation for objects instances of classes declaring fields of type `float` or `double`
- Fixed Device UID not displayed in the Front Panel window title (introduced in version *7.11.0*)
- Fixed loading of a resource from a JAR when the path starts with `/`
- Fixed potential deadlock on Front Panel startup in some cases
- Fixed Thread.getState() returning `TERMINATED` whereas the thread is running
- Fixed Simulator which may not stop properly when closing the Front Panel window
- Fixed Front Panel which stops sending widget events when dragging out of a widget
- [Multi] - Fixed monitor that may not be released when an exception occurs in a synchronized block (introduced in version `7.10.0`)
- [Multi] - Fixed invalid heap dump generation that causes heap analyzer crash
- [Multi] - Fixed potential crash (java.lang.NullPointerException) in debug mode when debugging an Application (introduced in version *7.10.0*)
- [Multi] - Fixed error when using `KF` library without defining a `kernel.kf` file in the Kernel (introduced in version *7.10.0*)

SOAR

- Added an option (`soar.bytecode.verifier`) to enable or disable the bytecode verifier (disabled by default)
- Removed size related limits in Architecture Evaluation version

Tools

- Added `SNI-1.4` support to HIL engine
- Updated Heap Dumper to verbose information about the memory section when an overlap is detected in the HEX file
- Updated Memory Map Scripts (Security, DTLS, Device)
- Fixed License Manager (Evaluation) random crash on Windows 10 when a VEE Port is built using `Build Module` button
- Fixed License Manager (Evaluation) wrong UID computation after reboot when Windows 10 Hyper-V feature is enabled
- Fixed HIL engine to exit as soon as the Simulator is disconnected (avoid remaining detached processes)
- Fixed ELF to Map generating symbol addresses different from the ELF symbol addresses (introduced in version *7.11.0*)
- Fixed Heap Dumper crash when a wrong object header is encountered
- Fixed Heap Dumper failure when a memory dump is larger than the heap section
- Fixed Heap Dumper crash when loading an Intel HEX file that contains lines of type `02`

[7.12.0] - 2019-10-16

Core Engine

- Updated implementation of internal OutOfMemoryError thrown with the maximum number of frames that can be dumped
- Updated `LLMJVM_dump()` output with the following new information:
  - Maximum number of alive threads
  - Total number of created threads
  - Maximum number of stack blocks used
  - Current number of stack blocks used
  - Objects referenced by each stack frame: address, type, length (in case of arrays), string content (in case of String objects)
  - [Multi] - Kernel stale references with the name of the Feature stopped

Foundation Libraries

- Fixed `EDC` implementation of Throwable.getStackTrace() when called on a OutOfMemoryError thrown by Core Engine or Simulator (either the returned stack trace array was empty or a java.lang.NullPointerException was thrown)
- [Tiny] - Fixed `EDC` implementation of StackTraceElement.toString() (removed the character `.` before the type)
- [Multi] - Fixed `KF` implementation of Feature.start() to throw an ExceptionInInitializerError when an exception is thrown in a Feature clinit method

Simulator

- Updated implementation of internal OutOfMemoryError thrown with more than one frames dumped per thread
  - By default the `20` top frames per thread are dumped. This can be modified using `S3.OutOfMemoryErrorNbFrames` system property
- Fixed wrong parsing of an array of `long` when an element is declared with only 2 digits (e.g. `25` was parsed as `2`)
- Fixed error parsing of an array of `byte` when an element is declared with the unsigned hexadecimal notation (e.g. `0xFF`) (introduced in version *7.10.0*)
- Fixed crash when ResourceBuffer.readString() is called on a String greater than `63` characters (introduced in version *7.10.0*)
- Fixed code coverage `.cc` generation of classpath directories
- Fixed crash during a GC when computing the references map of a complex method (an error message is dumped with the involved method name and suggest to increase the internal stack using `S3.JavaMemory.ThreadStackSize` system property)
- [Multi] - Added validity check of Shared Interface declaration files (`.si`) according to `KF` specification
- [Multi] - Fixed processing of Resource Buffers declared in Feature classpath

SOAR

- Added a new option `core.memory.oome.nb.frames` to configure the maximum number of stack frames that can be dumped when an internal OutOfMemoryError is thrown by Core Engine

Tools

- Updated Heap Dumper to verbose detected object references that are outside the heap
- Updated Heap Dumper to throw a dedicated error when an object reference does not target the beginning of an object (most likely a corrupted heap)
- Updated Heap Dumper to dump `.heap.error` partial file when a crash occurred during heap processing
- Fixed Heap Dumper crash when processing an object owned by a Feature which type is also owned by the Feature (was working before only when the type is owned by the Kernel)
- Fixed Firmware Linker potential negative offset generation when some sections do not appear in the same order in the ELF file than in their associated LOAD segment
- Fixed Code Coverage Analyzer potential generated empty report (wrong load of classfiles from JAR files)

[7.11.0] - 2019-06-24

Important Notes

- Java assertions execution is now disabled by default. If you experience any runtime trouble when migrating from a previous Architecture, please enable Java assertions execution both on Simulator and on Device (maybe the application code requires Java assertions to be executed).
- Calls to Security Manager are now disabled by default. If you are using the Security Manager, it must be explicitly enabled using the option described below (likely the case when building a Multi-Sandbox Firmware and its associated Virtual Device).
- Front Panel framework is now provided by the Architecture instead of the UI Pack. This allow to build a VEE Port with a Front Panel (splash screen, basic I/O, ...), even if it does not provide a MicroUI port. Moreover, the Front Panel framework API has been redesigned and is now distributed using the `ej.tool.frontpanel.framework` module instead of the legacy Eclipse classpath variable.

Known Issues

- SOAR `Internal SOAR error` or `Stacks merging coherence error` thrown when an `if` statement (being removed) is declared at the end of a `try` block:

```
try {
   ...
   if (Constants.getBoolean(XXX)) { // constant resolved to false
      ... // code being removed
   }
} catch (Exception e) {
        ...
}
```

Core Engine

- Added `EDC-1.3` support for daemon threads
- Added `BON` support for ej.bon.Util.newArray(T[],int)
- [Multi/ARMCC5] - Fixed unused undefined symbol that prevent Keil MDK-ARM to link properly

Foundation Libraries

- Updated `EDC` to version `1.3` (see EDC-1.3 API Changelog)
  - Updated the implementation code for correct Null analysis detection (added assertions, extracted multiple field accesses into a local)
  - Fixed PrintStream.PrintStream(OutputStream, boolean) writer initialization
  - Removed useless String literals in java.lang.Throwable
- Updated UTF-8 decoder to support Unicode code points
- Updated `BON` to version `1.4` (see BON-1.4 API Changelog)
- Updated `TRACE` to version `1.1`
  - Added ej.trace.Tracer.getGroupID()
  - Added a BON Constant (`core.trace.enabled`) to remove trace related code when tracing is disabled
- Fixed `KF` to call the registered Thread.UncaughtExceptionHandler when an exception is thrown by the first Feature thread

Integration

- Added new options for Java assertions execution in category `Runtime` (`core.assertions.sim.enabled` and `core.assertions.emb.enabled`). By default, Java assertions execution is disabled both on Simulator and on Device.
- Updated options categories (options property names left unchanged)
  - Added a new category named `Runtime`
  - Renamed `Target` to `Device`
  - Moved `Embed All type names` option from `Core Engine` to `Runtime`
  - Moved `Core Engine` under `Device`
  - Removed category `Target > Debug` and moved `Trace` options to `Runtime`
  - Removed category `Debug` and moved all sub categories under `Simulator`
  - Renamed category `JDWP` to `Debug`
- Added an option (`com.microej.library.edc.securitymanager.enabled`) to enable Security Manager runtime checks (disabled by default)

Simulator

- Added a cache to speed-up classfile loading in JARs
- Added `EDC-1.3` support for daemon threads
- Added `BON-1.4` support for compile-time constants (load of `.constants.list` resources)
- Added `BON-1.4` support for ej.bon.Util.newArray()
- Added Front Panel framework
- Updated error message when reaching Simulator limits
- Removed the `Bootstrapping a Smart Software Simulator` message when verbose mode in enabled
- Fixed Object.clone() on an immutable object to return a new (mutable) object instead of an immutable one
- Fixed Object.clone() crash when an OutOfMemory occurs
- Fixed potential crash when calling an abstract method (some interfaces of the hierarchy were not taken into account - introduced in version *7.10.0*)
- Fixed `OutOfMemory` errors even if the heap is not full (resources loaded from Class.getResourceAsStream() and ResourceBuffer creation were taken into account in simulated heap memory - introduced in version *7.10.0*)
- Fixed potential crash when a GC occurs while a ResourceBuffer is opened (introduced in version *7.10.0*)
- Fixed potential debugger hangs when an exception was thrown but not caught in the same method
- [Multi] - Fixed wrong class loading in some cases
- [Multi] - Fixed wrong immutable loading in some cases

SOAR

- Added `BON-1.4` support for compile-time constants (load of `.constants.list` resources)
- Added bytecode removal for Java assertions (when option is disabled)
- Added bytecode removal for `if(ej.bon.Constants.getBoolean())` pattern
  - `then` or `else` block is removed depending on the boolean condition
  - *WARNING: Current limitation: the ``if`` statement cannot wrap or be nested in a ``try-catch-finally`` statement*
- Added an option for grouping all the methods by type in a single ELF section
  - `com.microej.soar.groupMethodsByType.enabled` (`false` by default)
  - *WARNING: this option avoids to reach the maximum number of ELF sections (65536) when building a large application, but affects the application code size (especially inline methods are embedded even if they are not used)*
- Added an error message when `microejapp.o` cannot be generated because the maximum number of ELF sections (65536) is reached

Tools

- Updated License Manager (Production) to debug dongle recognition issues from command line (see *Check Activation with the Command Line Tool*).
- Updated License Manager (Production) to support dongle recognition on macOS `10.14` (Mojave)
- Fixed ELF To Map to produce correct sizes from an executable generated by IAR Embedded Workbench for ARM
- Fixed Firmware Linker `.ARM.exidx` section generation (missing section link content)
- Updated deployment files policy for VEE Ports in Workspace, in order to be more flexible depending on the C project layout. This also allows to deploy to the same C project different Applications built with different VEE Ports
  - VEE Port configuration: in `bsp/bsp.properties`, a new option `output.dir` indicates where the files are deployed by default
    * Application (`microejapp.o`) and Runtime library (`microejruntime.a`) are deployed to `${output.dir}/lib`. Architecture header files (`*.h`) are deployed to `${output.dir}/inc/`
    * When this option is not set, the legacy behavior is left unchanged (`project.file` option in collaboration with `augmentCProject` scripts)
  - Launch configuration: `Device > Deploy` options allow to override the default VEE Port configuration in order to deploy each file into a separate folder.
- Fixed wrong ELF file generation when a section included in a LOAD segment was generated before one of the sections included in a LOAD segment declared before the first one (integrated in ELF Utils and Firmware Linker)
- Fixed wrong ELF file generation when a section included in a LOAD segment had an address which was outside its LOAD segment virtual address space (integrated in ELF Utils and Firmware Linker)

[7.10.1] - 2019-04-03

Simulator

- Fixed Object.getClass() may return a Class instance owned by a Feature for type owned by the Kernel

[7.10.0] - 2019-03-29

Core Engine

- Added internal memories checks at startup: heaps and statics memories are not allowed to overlap with `LLBSP_IMPL_isInReadOnlyMemory()`
- [Multi] - Updated Feature Kill implementation to prepare future RAM Control (fully managed by Core Engine)
- [Multi] - Updated implementation of ej.kf.Kernel: all APIs taking a Feature argument now will throw a java.lang.IllegalStateException when the Feature is not started

Foundation Libraries

- Updated `KF` library in sync with Core Engine Kill related fixes and Simulator with Kernel & Features semantic
- Updated `BON` library on Simulator (now uses the same implementation than the one used by the Core Engine)

Integration

- Added generation of `architecture.properties` file when building a VEE Port. (Used by SDK `5.x` when manipulating VEE Ports & Virtual Devices)

Simulator

- Added `Embed all types names` option for Simulation
- Added memory size simulation for Java Heap and Immortal Heap (Enabling `Use target characteristics` option is no more required)
- Added Kernel & Features semantic, as defined in the `KF-1.4` specification
  - Fully implemented:
    * Ownership for types, object and thread execution context
    * Kernel mode
    * Context Local Static Field References
  - Partially implemented:
    * Kernel API (Type grained only)
    * Shared Interfaces are binded using direct reference links (no Proxy execution)
    * Feature.stop() does not perform the safe kill. The application cannot be stopped unless it has correctly removed all its shared references.
  - Not implemented:
    * Dynamic Feature installation from Kernel.install(java.io.InputStream)
    * Execution Rules Runtime checks

Tools

- Updated Memory Map Scripts (Bluetooth, MWT, NLS, Rcommand and AllJoyn libraries)
- Fixed `Kernel Packager` internal limits error when the ELF executable does not contains a `.debug.soar` section
- Fixed wrong ELF file generation when segment file size is different than the mem size (integrated in `ELF Utils` and `Firmware Linker`)
- Fixed Simulator COM port mapping default value (set to `disabled` instead of `UART<->UART` in order to avoid an error when launch configuration is just created)
- Fix ELF To Map: the total sections size were not equal to the segments size

[7.9.1] - 2019-01-08

Tools

- Fixed ELF objcopy generation when ELF executable file contains `0` size segments
- Fixed `Stack Trace Reader` error when ELF executable file contains relocation sections

[7.9.0] - 2018-09-20

Core Engine

- Fixed OutOfMemoryError thrown when allocating an object of the size of free memory in immortals heap

SOAR

- Optimized SOAR processing (up to 50% faster on applications with tens of classpath entries)

[7.8.0] - 2018-08-01

Tools

- [ARMCC5] - Updated *SOAR Debug Infos Post Linker* tool to generate the full ELF executable file

[7.7.0] - 2018-07-19

Core Engine

- Added a permanent hook `LLMJVM_on_Runtime_gc_done` called after an explicit java.lang.Runtime.gc()
- Updated internal heap header for memory dump

SOAR

- Added check for the maximum number of allowed concrete types (avoids a Core Engine link error)

Tools

- Added `Heap Dumper` tool

[7.6.0] - 2018-06-29

Foundation Libraries

- [Multi] - Updated `BON` library: a Timer owned by the Kernel can execute a TimerTask owned by a Feature

[7.5.0] - 2018-06-15

Internal Release - COTS Architecture left unchanged.

[7.4.0] - 2018-06-13

Core Engine

- Removed partial support of `ej.bon.Util.throwExceptionInThread()` (deprecated)
- [Multi/Linux] - Updated default configuration to always embed method names
- [Multi/Cortex-M] - Optimized KF checks execution for array & field accesses

Foundation Libraries

- Updated ej.bon.Timer to schedule ej.bon.TimerTask owned by multiple Features

Simulator

- Fixed implementation of Class.getResourceAsStream() to throw an IOException when the stream is closed

SOAR

- [GCC] - Fixed `microejapp.o` link with GCC 6.3

Tools

- Added a retry mechanism in the Testsuite Engine
- Added a message to suggest increasing the JVM heap when an OutOfMemoryError occurs in the `Firmware Linker` tool
- Fixed generation of LL header files for all cross compilation toolchains (file separator for included paths is `/`)
- [Cortex-A/ARMCC5] - Fixed SNI convention call issue
- [ESP32,RX] - Fixed `Firmware Linker` tool internal limit

[7.3.0] - 2018-03-07

Simulator

- Added an option for the IDE to customize the mockups classpath
- Fixed Deadlock in Shielded Plug remote client when interrupting a thread that waits for block modification

[7.2.0] - 2018-03-02

Core Engine

- [Multi] - Enabled quantum counter computation only when Feature quota is set
- [Cortex-M/IAR] - Updated compilation flags to `-Oh`

Simulator

- Added a hook in the mockup that is automatically called during the HIL engine startup
- Added dump of loaded classes when `verbose` option is enabled
- Fixed Runtime.freeMemory() call freeze when `Emb Characteristics` option is enabled
- Fixed ShieldedPlug server error after interrupting a thread that is waiting for a database block
- Fixed crash `Access to a wrong reference` in some cases
- Fixed java.lang.NullPointerException when interrupting a thread that has not been started
- Fixed crash when closing an HIL engine connection in some cases
- [Multi] - Fixed KF & Watchdog library link when `Emb Characteristics` option is enabled
- [Multi] - Fixed XML Parsing error when `Emb Characteristics` option is enabled

[7.1.2] - 2018-02-02

SOAR

- Fixed SNI library was added in the classpath in some cases

[maintenance/6.18.0] - 2017-12-15

Core Engine

- [Multi] - Enabled quantum counter computation only when Feature quota is set
- [Cortex-M/IAR] - Updated compilation flags to `-Oh`

Simulator

- Fixed Runtime.freeMemory() call freeze when `Emb Characteristics` option is enabled
- [Multi] - Fixed KF & Watchdog library link when `Emb Characteristics` option is enabled
- [Multi] - Fixed XML Parsing error when `Emb Characteristics` option is enabled

Tools

- Updated `Kernel API Generator` tool with classes filtering

[7.1.1] - 2017-12-08

Tools

- [Multi/RX] - Fixed `Firmware Linker` tool

[7.1.0] - 2017-12-08

Core Engine

- [Multi/RX] - Added KF support

Integration

- Fixed `SNI-1.3` library name

SOAR

- [RX] - Added support for ELF symbol prefix `_`

Tools

- Updated `Kernel API generator` tool with classes filtering

[7.0.0] - 2017-11-07

Core Engine

- Added SNI-1.3 support
- `SNI_suspendCurrentJavaThread()` is not interruptible via Thread.interrupt() anymore

Foundation Libraries

- Updated to `SNI-1.3`

[6.17.2] - 2017-10-26

Simulator

- Fixed deadlock during bootstrap in some cases

[6.17.1] - 2017-10-25

Core Engine

- Fixed conversion of `-0.0` into a positive value

[6.17.0] - 2017-10-10

Tools

- Updated Memory Map Scripts for TRACE library

[6.16.0] - 2017-09-27

Core Engine

- Fixed External Resource Loader link error (introduced in version *6.13.0*)

[6.15.0] - 2017-09-12

Core Engine

- Added a new option to configure the maximum number of monitors that can be owned per thread (8 per thread by default, as it was fixed before)

Foundation Libraries

- Fixed ECOM-COMM internal heap calibration

SOAR

- Added log of the class loading cause

[6.14.2] - 2017-08-24

Tools

- Fixed `Firmware Linker` tool script (load `activity.xml` from the wrong folder)
- Fixed load of symbol `_java_Ljava_io_EOFException` that can be required by some linkers even if this symbol is not touched

[6.14.1] - 2017-08-02

Simulator

- Fixed Device Mockup too long initialization that may block the Front Panel Mockup

Foundation Libraries

- Fixed BON `.types.list` potential conflicts with KF

Tools

- Modified `Firmware Linker` internal scripts structure for new Virtual Devices tools

[6.13.0] - 2017-07-21

Core Engine

- Added support for ej.bon.ResourceBuffer

Foundation Libraries

- Updated to `BON-1.3`

SOAR

- Added support for `*.resourcesext.list` (resources excluded from the firmware)

Tools

- Added BON Resource Buffer generator

[6.12.0] - 2017-07-07

Core Engine

- Added a trace when IllegalMonitorStateException is thrown on a `monitorexit`

Tools

- Added property `skip.mergeLibraries` for Platform Builder.
- Updated the serial PC connector to JSSC `2.8.0`.

Simulator

- Fixed unexpexted java.lang.NullPointerException in some cases

[6.11.0] - 2017-06-13

Integration

- Fixed useless watchdog library copied in root folder

[6.11.0-beta1] - 2017-06-02

Core Engine

- Added an option to enable execution traces
- Added Low Level API `LLMJVM_MONITOR_impl.h`
- Added Low Level API `LLTRACE_impl.h`

Foundation Libraries

- Added `TRACE-1.0`

[6.10.0] - 2017-06-02

Core Engine

- Optimized java.lang.Runtime.gc() (removed useless heap compaction in some cases)

[6.9.2] - 2017-06-02

Integration

- Fixed missing properties in `release.properties` (introduced in version *v6.9.1*)
- Fixed artifacts build dependencies to private dependencies

[6.9.1] - 2017-05-29

SOAR

- [Multi] - Fixed selected methods list in report generation (removed Kernel related method)

[6.9.0] - 2017-03-15

Base version, included into SDK 4.1.

6.3.3 Release Notes

Foundation Libraries

The following table describes Foundation Libraries API versions implemented in MicroEJ Architectures.

Table 1: Architecture API Implementation

Architecture Range	EDC	BON	KF	SNI	SP	Trace	Device	ECOM-COMM
8.0.0	1.3	1.4	1.7	1.4	2.0	1.1	N/A[1]	N/A[2]
[7.17.0-7.20.1]	1.3	1.4	1.6	1.4	2.0	1.1	1.0	1.1
[7.13.0-7.16.0]	1.3	1.4	1.5	1.4	2.0	1.1	1.0	1.1
[7.11.0-7.12.0]	1.3	1.4	1.5	1.3	2.0	1.1	1.0	1.1
[7.10.0-7.10.1]	1.2	1.3	1.5	1.3	2.0	1.0	1.0	1.1
[7.0.0-7.9.1]	1.2	1.3	1.4	1.3	2.0	1.0	1.0	1.1
[6.13.0-6.18.0]	1.2	1.3	1.4	1.2	2.0	1.0	1.0	1.1
[6.11.0-6.12.0]	1.2	1.2	1.4	1.2	2.0	1.0	1.0	1.1
[6.9.0-6.10.0]	1.2	1.2	1.4	1.2	2.0	N/A	1.0	1.1

[1] See *Migrate Device Module*.
[2] See *Migrate ECOM-COMM Module*.

6.4 MicroEJ Packs

6.4.1 Overview

On top of a MicroEJ Architecture can be imported MicroEJ Packs which provide additional features such as:

- *Serial Communications*,
- *Graphical User Interface*,
- *Networking*,
- *File System*,
- etc.

Each MicroEJ Pack is optional and can be selected on demand during the *MicroEJ Platform configuration* step.

6.4.2 Naming Convention

MicroEJ Packs are distributed in two packages:

- MicroEJ Architecture Specific Pack under the com/microej/architecture/* organization.
- MicroEJ Generic Pack under the com/microej/pack/* organization.

See *Pack Import* for usage.

Architecture Specific Pack

MicroEJ Architecture Specific Packs contain compiled libraries archives and are thus dependent on the MicroEJ Architecture and toolchain used in the MicroEJ Platform.

MicroEJ Architecture Specific Packs files ends with the `.xpfp` extension and are classified using the following naming convention:

```
com/microej/architecture/[ISA]/[TOOLCHAIN]/[UID]-[NAME]-pack/[VERSION]/[UID]-[NAME]-pack-
↪[VERSION].xpfp
```

- `ISA` : instruction set architecture (e.g. `CM4` for Arm® Cortex®-M4, `ESP32` for Espressif ESP32, ...).
- `TOOLCHAIN` : C compilation toolchain (e.g. `CM4hardfp_GCC48`).
- `UID` : Architecture unique ID (e.g. `flopi4G25`).
- `NAME` : pack name (e.g. `ui`).
- `VERSION` : pack version (e.g. `13.0.4`).

For example, MicroEJ Architecture Specific Pack UI versions for Arm® Cortex®-M4 microcontrollers compiled with GNU CC toolchain are available at https://repository.microej.com/modules/com/microej/architecture/CM4/CM4hardfp_GCC48/flopi4G25-ui-pack/.

Generic Pack

MicroEJ Generic Packs can be imported on top of any MicroEJ Architecture.

They are classified using the following naming convention:

```
com/microej/pack/[NAME]/[NAME]-pack/[VERSION]/
```

- `NAME` : pack name (e.g. `bluetooth`).
- `VERSION` : pack version (e.g. `2.1.0`).

For example, MicroEJ Generic Pack Bluetooth versions are available at https://repository.microej.com/modules/com/microej/pack/bluetooth/bluetooth-pack/.

Legacy Generic Pack

Legacy MicroEJ Generic Packs files end with the `.xpfp` extension. These Packs contain one or more Platform modules. See *Platform Module Configuration* for their configuration. They are classified using the following naming convention:

```
com/microej/pack/[NAME]/[VERSION]/[NAME]-[VERSION].xpfp
```

- `NAME` : pack name (e.g. `net`).
- `VERSION` : pack version (e.g. `9.2.3`).

For example, the Legacy MicroEJ Generic Pack NET version `9.2.3` is available at https://repository.microej.com/modules/com/microej/pack/net/9.2.3/net-9.2.3.xpfp.

6.5 Platform Creation

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

Note: The creation of a Platform with this guide requires at least the version `5.4.0` of the SDK.

Note: If you own a legacy Platform, you can either create your Platform again from scratch, or follow the *Former Platform Migration* chapter.

6.5.1 Architecture Selection

The first step is to select a *MicroEJ Architecture* compatible with your device instructions set and C compiler.

MicroEJ Corp. provides MicroEJ Evaluation Architectures for most common instructions sets and compilers at https://repository.microej.com/modules/com/microej/architecture.

Please refer to the chapter *Architectures MCU / Compiler* for the details of ABI and compiler options.

If the requested MicroEJ Architecture is not available for evaluation or to get a MicroEJ Production Architecture, please contact your MicroEJ sales representative or *our support team*.

6.5.2 Platform Configuration

The next step is to create a MicroEJ Platform configuration project:

- Select File > New > Project... > General > Project ,
- Enter a Project name . The name is arbitrary and can be changed later. The usual convention is `[PLATFORM_NAME]-configuration` ,
- Click on Finish button. A new empty project is created,
- Install the latest Platform Configuration Additions by following instructions described at https://github.com/MicroEJ/VEEPortQualificationTools/blob/master/framework/platform/README.rst.
  - Files within the `content-sdk-5` folder must be copied to the configuration project folder.
  - Files within the `content-architecture-7` must be copied to the configuration project folder only if you are using an Architecture version `7.x` . If you are using an Architecture version `8.x` , the files are already included and **must not** be copied.

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

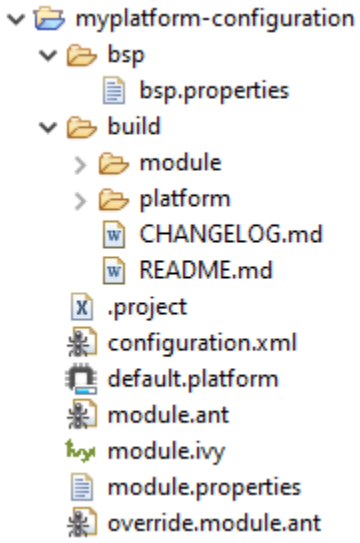


Fig. 8: MicroEJ Platform Configuration Project Skeleton

Note: The version of installed Platform Configuration Additions is indicated in the CHANGELOG file.

- Edit the *Module Description File* `module.ivy` to declare the MicroEJ Architecture dependency:

```
<dependencies>

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

</dependencies>
```

The name of the module dependency needed for your Platform can be found in the chapter *Architectures MCU / Compiler*. Check the table of your corresponding Architecture and follow the link in the Module Name column.

For example, to declare the MicroEJ Evaluation Architecture version `7.14.0` for Arm® Cortex®-M4 microcontrollers compiled with GNU CC toolchain:

```
<dependencies>

   <dependency org="com.microej.architecture.CM4.CM4hardfp_GCC48" name="flopi4G25" rev=
↪"7.14.0">
     <artifact name="flopi4G25" m:classifier="eval" ext="xpf"/>
   </dependency>

</dependencies>
```

And the module for this Architecture is located in the *Central Repository* at https://repository.microej.com/modules/com/microej/architecture/CM4/CM4hardfp_GCC48/flopi4G25/7.14.0/.

> **Note:** The Platform Configuration Additions allow to select the Architecture `USAGE` using the option `com.microej.platformbuilder.architecture.usage`. Edit the file `module.properties` to set the property to `prod` to use a Production Architecture and to `eval` to use an Evaluation Architecture.

6.5.3 Pack Import

MicroEJ Pack provides additional features on top of the MicroEJ Architecture such as Graphical User Interface or Networking.

Note: MicroEJ Packs are optional. You can skip this section if you intend to integrate MicroEJ runtime only with custom libraries.

To declare a MicroEJ Pack dependency, edit the *Module Description File* `module.ivy` as follows:

```
  <dependencies>
    <!-- MicroEJ Architecture Specific Pack -->
    <dependency org="com.microej.architecture.[ISA].[TOOLCHAIN]" name="[UID]-[NAME]-pack"␣
↪rev="[VERSION]"/>

    <!-- MicroEJ Generic Pack -->
    <dependency org="com.microej.pack.[NAME]" name="[NAME]-pack" rev="[VERSION]"/>

    <!-- Legacy MicroEJ Generic Pack -->
    <dependency org="com.microej.pack" name="[NAME]" rev="[VERSION]"/>

  </dependencies>
```

For example, to declare the MicroEJ Architecture Specific Pack UI version 13.0.4 for MicroEJ Architecture `flopi4G25` on Arm® Cortex®-M4 microcontrollers compiled with GNU CC toolchain:

```
<dependencies>
    <!-- MicroEJ Architecture Specific Pack -->
    <dependency org="com.microej.architecture.CM4.CM4hardfp_GCC48" name="flopi4G25-ui-pack"␣
↪rev="13.0.4"/>

</dependencies>
```

To declare the MicroEJ Generic Pack Bluetooth version 2.1.0:

```
<dependencies>
    <!-- MicroEJ Generic Pack  -->
    <dependency org="com.microej.pack.bluetooth" name="bluetooth-pack" rev="2.1.0"/>

</dependencies>
```

And to declare the Legacy MicroEJ Generic Pack Net version 9.2.3:

```
<dependencies>
    <!-- Legacy MicroEJ Generic Pack -->
    <dependency org="com.microej.pack" name="net" rev="9.2.3"/>

</dependencies>
```

Warning: *MicroEJ Architecture Specific Packs* and *Legacy MicroEJ Generic Packs* provide Platform modules that are **not installed** by default. See *Platform Module Configuration* section for more details.

6.5.4 Platform Build

The MicroEJ Platform can be built either from the SDK or from the *MMM CLI*. To build the MicroEJ Platform from the SDK, perform a regular *Module Build*:

- Right-click on the Platform Configuration project,
- Select Build Module .

To build the MicroEJ Platform from the MMM CLI:

- Set the `eclipse.home` property to the path of your SDK, using `-Declipse.home=<path>` in the command line or using the *Shared configuration*.

  By default, the SDK's path is one of the following directories:

  - on Windows: `C:\Program Files\MicroEJ\MicroEJ-SDK-<YY.MM>\rcp`
  - on Linux: `/home/<user>/MicroEJ/MicroEJ-SDK-<YY.MM>/rcp`
  - on macOS: `/Applications/MicroEJ/MicroEJ-SDK-<YY.MM>/rcp/MicroEJ-SDK-<YY.MM>.app/Contents/Eclipse`
- From the Platform Configuration project, execute the command: `mmm`

In both cases, the build starts and the build logs are redirected to the integrated console. Once the build is terminated, you should get the following message:

```
module-platform:report:
  [echo]    ␣
↪==========================================================================================================
  [echo]     Platform has been built in this directory 'C:\tmp\mydevice-Platform-
↪[TOOLCHAIN]-0.1.0'.
  [echo]     To import this project in your MicroEJ SDK workspace (if not already␣
↪available):
  [echo]      - Select 'File' > 'Import...' > 'General' > 'Existing Projects into␣
↪Workspace' > 'Next'
  [echo]      - Check 'Select root directory' and browse 'C:\tmp\mydevice-Platform-
↪[TOOLCHAIN]-0.1.0' > 'Finish'
  [echo]    ␣
↪==========================================================================================================

BUILD SUCCESSFUL

Total time: 43 seconds
```

Then, import the Platform directory to your SDK workspace as mentioned in the report. You should get a ready-to-use MicroEJ Platform project in the workspace available for the MicroEJ Application project to run on. You can also check the MicroEJ Platform availability in: Window > Preferences > MicroEJ > Platforms in workspace .

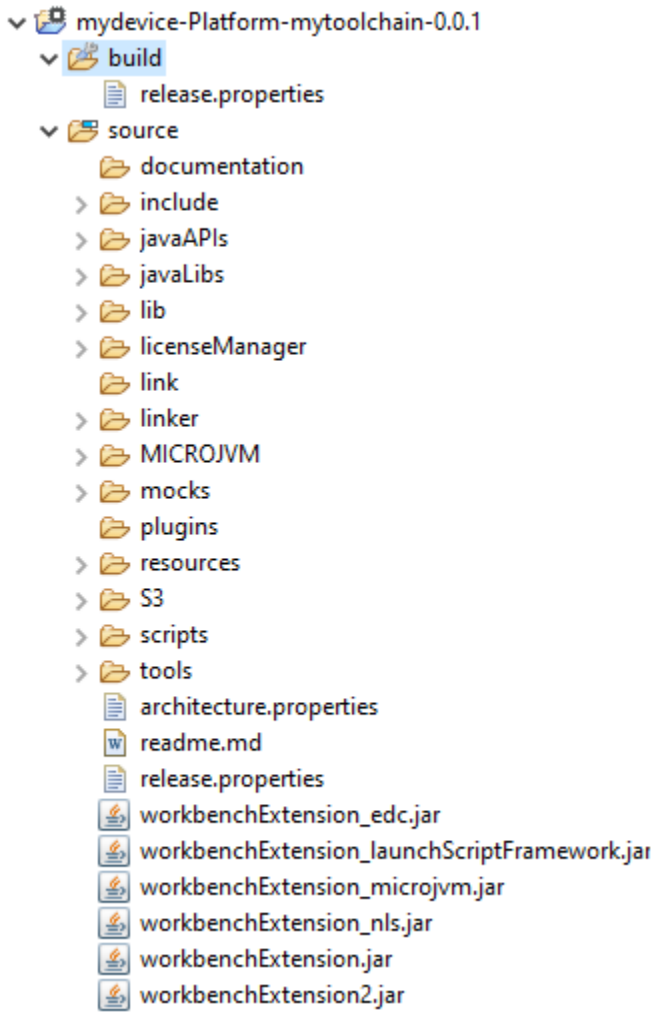


Fig. 9: MicroEJ Platform Project

This step is only required the first time the Platform is built, or if the Platform properties have changed (i.e, name, version). When the same Platform is built again, the Platform project should be automatically refreshed after a few seconds. In case of any doubt, right-click on the Platform project and select Refresh to get the new content.

6.5.5 Platform Module Configuration

A Platform module is the minimal unit that can extend a MicroEJ Architecture with additional features such as:

- Runtime Capability (e.g. *Multi-Sandbox*, *External Resources Loader*) ,
- Foundation Library Implementation (e.g. *MicroUI*, *NET*),
- Simulator (e.g. *Front Panel Mock*),
- Tool (e.g. *MicroEJ Java H*).

Platform modules provided by *MicroEJ Generic Packs* are automatically installed during the *Platform build* and do not require extra configuration. They are not displayed in the Platform Editor.

Platform modules provided by *MicroEJ Architectures*, *MicroEJ Architecture Specific Packs* and *Legacy MicroEJ Generic Packs* are **not installed** by default. They must be enabled and configured using the Platform Editor.

Before opening the Platform Editor, the Platform must have been built once to let *MicroEJ Module Manager* resolve and download MicroEJ Architecture and Packs locally. Then import them in the SDK as follows:

- Select File > Import > MicroEJ > Architectures ,
- Browse myplatform-configuration/target~/dependencies folder (contains `.xpf` and `.xpfp` files once the Platform is built),
- Check the I agree and accept the above terms and conditions… box to accept the license,
- Click on Finish button. This may take some time.

Once imported, double-click on the default.platform file to open the Platform Editor.

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

Platform modules are organized in groups. When a group is selected, by default all its modules are selected. To view all the modules making up a group, click on the Expand All icon on the top-right of the frame. This will let you select/deselect on a per-module basis. Note that individual module selection is not recommended and that it is only available when the module has been imported.

The description and contents of an item (group or module) are displayed next to the list when an item is selected.

All the selected Platform modules will be installed in the Platform.

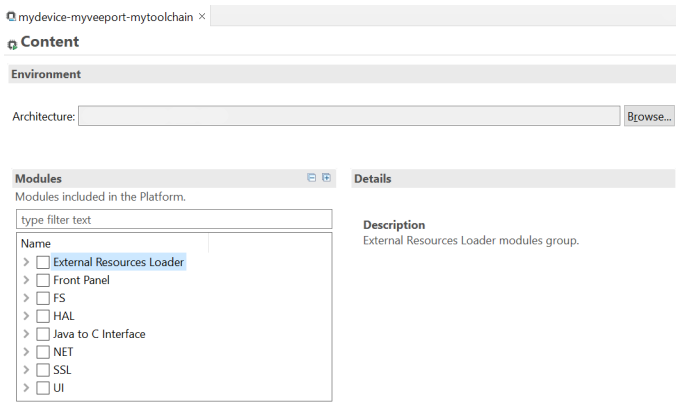


Fig. 10: MicroEJ Platform Configuration Modules Selection

Each selected Platform module can be customized by creating a [module] folder (named after the module name), next to the .platform file definition. It may contain:

- A [module].properties file named after the module name. These properties will be injected in the execution context prefixed by the module name. Some properties might be needed for the configuration of some modules. Please refer to the modules documentation for more information.
- A bsp.xml file which provides additional information about the BSP implementation of Low Level APIs.

  This file must start with the node `<bsp>` . It can contain several lines like this one: `<nativeName="A_LLAPI_NAME" nativeImplementation name="AN_IMPLEMENTATION_NAME"/>`

where:

- `A_LLAPI_NAME` refers to a Low Level API native name. It is specific to the MicroEJ C library which provides the Low Level API.
- `AN_IMPLEMENTATION_NAME` refers to the implementation name of the Low Level API. It is specific to the BSP; and more specifically, to the C file which does the link between the MicroEJ C library and the C driver.

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

- Optional module specific files and folders

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

Note: It is possible to quickly rebuild the Platform from the Platform Editor if only the Platform module configuration has changed. Click on the `Build Platform` link on the `Overview` tab of the Platform Editor.

6.5.6 Platform Customization

The configuration project (the project which contains the `.platform` file) can contain an optional `dropins` folder. The full content of this folder will be copied in the Platform during the build. This feature allows to add or overwrite libraries, tools, etc. into the Platform.

The dropins folder organization should respect the 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 are organized. Then fill the dropins folder with additional features and build again the Platform to get a customized Platform.

Files in the dropins folder have priority. If one file has the same path and name as a file already installed in the Platform, the file from the dropins folder will be selected first.

Platform build can also be customized by updating the `configuration.xml` file next to the `.platform` file. This Ant script can extend one or several of the extension points available. By default, you should not have to change the default configuration script.

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

6.5.7 Platform Publication

The publication of the built Platform to a *module repository* is disabled by default. It can be enabled by setting the `skip.publish` property to `false` in the `module.properties` file of the Platform configuration project .

The publication is kept disabled by default in the project sources because developers usually use the locally built platform in the workspace. However, the publication is required in a Continuous Integration environment. This can be done by leaving the `skip.publish` property to `true` in the project sources and by overwriting it in the command launched by the Continuous Integration environment, for example:

```
mmm publish shared -Dskip.publish=false
```

If the Platform is configured with *Full BSP connection*, the build script can be launched to validate that the BSP successfully compiles and links before the Platform is published. It can be enabled by setting the `com.microej.platformbuilder.bsp.build.enabled` property to `true` in the `module.properties` file of the Platform configuration project (defaults to `false` if not set).

6.5.8 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 (the Executable). For more information, please consult the *MicroEJ build process overview*.

The BSP connection can be configured by defining 4 folders where the following files are located:

- 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 Executable (`application.out`).

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

BSP connection configuration is only required in the following cases:

- Use the SDK to produce the Executable of a Mono-Sandbox Application (recommended).
- Use the SDK to run a *MicroEJ Test Suite* on device.
- Build a the Executable of a Multi-Sandbox Application.

MicroEJ provides a flexible way to configure the BSP connection to target any kind of projects, teams organizations and company build flows. To achieve this, the BSP connection can be configured either at 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).

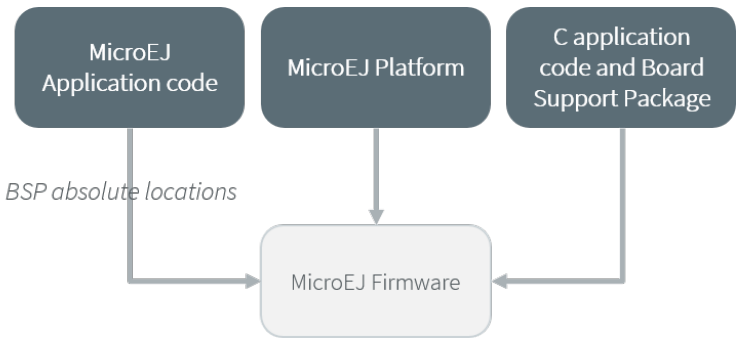


Fig. 11: MicroEJ Platform with no BSP connection

This case is recommended when:

- the Executable is built outside the 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).

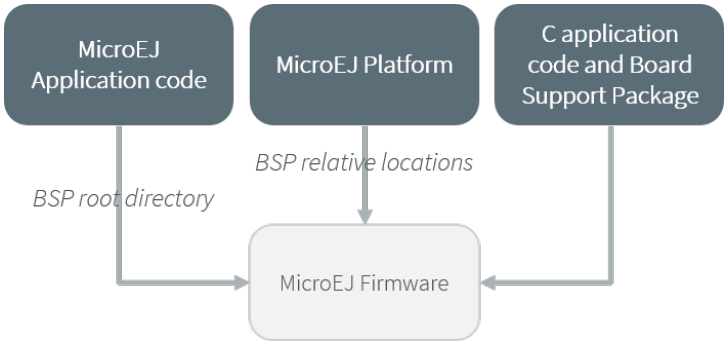


Fig. 12: MicroEJ Platform with partial BSP connection

  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 requires to build the Executable again.

- Case 3: Full BSP connection

  The MicroEJ Platform includes the BSP.

  BSP connection is configured when building the 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.

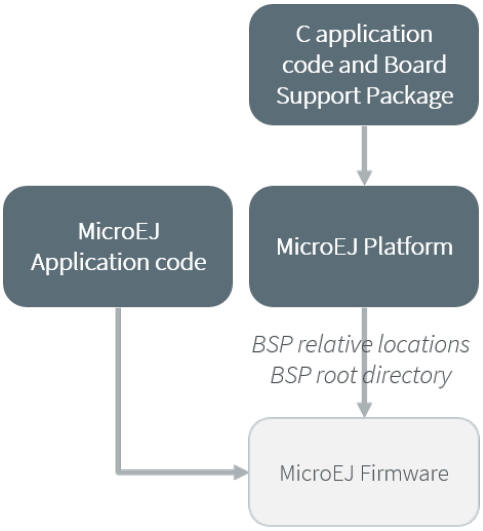


Fig. 13: MicroEJ Platform with full BSP connection

This case is recommended when:

- the MicroEJ Platform is used to build various MicroEJ Applications.
- the MicroEJ Platform is validated using MicroEJ test suites.
- 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 the Platform options, which can be set in the `bsp/bsp.properties` file of the Platform configuration project.

Table 2: 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 the Application options, which can be set as regular *MicroEJ Application Options*.

Table 3: 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`) at the location specified 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 include the BSP.
`deploy.dir.microejapp`	Absolute path to the directory where to deploy the MicroEJ Application file (`microejapp.o`). An empty value means no deployment.
`deploy.dir.microejlib`	Absolute path to the directory where to deploy the MicroEJ Platform runtime file (`microejruntime.a`) to this absolute directory. An empty value means no deployment.
`deploy.dir.microejinc`	Absolute path to the directory where to deploy the MicroEJ Platform header files (`*.h`) to this absolute directory. An empty value means no deployment.
`deploy.dir.microejscript`	Absolute path to the directory that contains the BSP build script file (`build.bat` or `build.sh`). An empty value means no build script execution.

Note: It is also possible to configure the BSP root directory by setting the *build option* `toolchain.dir` , instead of the application option `deploy.bsp.root.dir` . This allows to build the Executable by specifying both the 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 built outside the SDK

```
Platform Options:
  [NONE]

Application Options:
  [NONE]
```

- No BSP connection, Executable built using the SDK

```
Platform Options:
  [NONE]

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

- Partial BSP connection, Executable built outside the SDK

```
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 built using the 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 built using the 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]

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

Build Script File

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

The build script must comply with the following specification:

- On Windows operating system, it is a Windows batch file named `build.bat` .
- On macOS or Linux operating systems, it is a shell script named `build.sh` , with execution permission enabled.
- On error, the script must end with a non zero exit code.
- On success
  - The Executable must be copied to a file named `application.out` in the directory from where the script has been executed.
  - The script must end with zero exit code.

Many build script templates are available for most commonly used C toolchains in the Platform Qualification Tools repository.

The build script can also be launched before the Platform publication, see *Platform Publication* for more details.

Note: The Executable must be an ELF executable file. On Unix, the command `file(1)` can be use to check the format of a file. For example:

```
~$ file application.out
ELF 32-bit LSB executable
```

Run Script File

This script is required only for Platforms intended to run a *MicroEJ Testsuite* on device.

The BSP run script is used to invoke a third-party tool to upload and start the Executable on device.

The run script must comply with the following specification:

- On Windows operating system, it is a Windows batch file named `run.bat`.
- On macOS or Linux operating systems, it is a shell script named `run.sh`, with execution permission enabled.
- The Executable filename is passed as first script parameter if there is one, otherwise it is the `application.out` file located in the directory from where the script has been executed.
- On error, the script must end with a non zero exit code.
- On success:
  - The Executable (`application.out`) has been uploaded and started on the device
  - The script must end with zero exit code.

The run script can optionally redirect execution traces. If it does not implement execution traces redirection, the testsuite must be configured with the following *Standalone Application Options* in order to take its input from a TCP/IP socket server, such as *Serial to Socket Transmitter*.

```
testsuite.trace.ip=localhost
testsuite.trace.port=5555
```

6.5.9 Platform API Documentation

The Platform API documentation provides a comprehensive HTML Javadoc that combines all the Foundation Library APIs.

It can be built using the following steps:

- Create a new *module repository project*.
- Enable module repository javadoc generation (see *Generate Javadoc*).
- Go to your Platform build directory and browse `source/javaLibs` and `source/MICROJVM/javaLibs` directories. You will find Foundation Libraries implementations JAR files in the following pattern: `<module_name>-<major>.<minor>.jar`.

  Example: `EDC-1.3.jar`: `module_name` = `edc`, `major` = `1`, `minor` = `3`.
- For each Foundation Library your want to include,
  - Retrieve its api module in either the *Central Repository*, *Developer Repository* or your custom repository. Most of the Foundation Library APIs provided by MicroEJ are available under the `ej.api` organization.

    Example: EDC is on the Central Repository (https://repository.microej.com/modules/ej/api/edc/)

- Get the latest available patch version corresponding to your `<major>.<minor>` version. This allows to benefit from the latest javadoc fixes and updates for the corresponding version.

  Example: `ej.api#edc#1.3.5`: `patch``=``5`

- Declare a dependency line in the module repository.

```
<dependency conf="artifacts->*"  transitive="false" org="<org>" name="<module_name>
↪" rev="<major>.<minor>.<patch>" />
```

  Example:

```
<dependency conf="artifacts->*"  transitive="false" org="ej.api" name="edc" rev="1.
↪3.5" />
```

- Build the module repository.

The Platform API documentation is available in `<module_repository_project>/target~/artifacts/<module_repository_name>-javadoc.zip`.

6.5.10 Link-Time Option

It is possible to define custom *Application options* that can be passed to the BSP through an ELF symbol defined at link-time, hence the term *link-time option*. This allows to provide configuration options to the Application developer without the need to rebuild the BSP source code.

To define a link-time option, first choose an option name with only alphanumeric characters (`[a-zA-Z][a-zA-Z0-9]*` without spaces).

Proceed with the following steps by replacing `[my_option]` with your option name everywhere:

- Create a folder inside your *Platform Customization* part (e.g: `[platform]-configuration/dropins/scripts/init-[my_option]`)
- Create an init script file and put it inside (e.g: `[platform]-configuration/dropins/scripts/init-[my_option]/init-[my_option].xml` file). Here is the init script file template content:

```
<project name="[my_option]-init">
  <target name="init/execution/[my_option]" extensionOf="init/execution" if="onBoard">
    <!-- Set option default value -->
    <property name="[my_option]" value="0"/>

    <!-- Create tmp dir -->
    <local name="link.files.dir"/>
    <microejtempfile deleteonexit="true" prefix="link[my_option]" property="link.files.
↪dir"/>
    <mkdir dir="${link.files.dir}"/>
    <!-- Get tmp link file name -->
    <local name="link.[my_option]"/>
    <property name="link.[my_option]" value="${link.files.dir}/[my_option].lscf" />
    <echoxml file="${link.[my_option]}" append="false">
      <lscFragment>
        <defSymbol name="[my_option]" value="${[my_option]}" rootSymbol="true"/>
      </lscFragment>
    </echoxml>
    <!-- Add link file in linker's link files path -->
```

(continues on next page)

(continued from previous page)

```
    <augment id="partialLink.lscf.path">
      <path location="${link.files.dir}"/>
      <path location="${jpf.dir}/link"/>
    </augment>
  </target>
</project>
```

- In your BSP source code, define an ELF symbol `[my_option]` can then be used inside C files in your BSP with:

```
// Declare the symbol as an extern global
extern int [my_option];

void my_func(void){
   // Get the symbol value
   int [my_option]_value = ((int)(&[my_option]));

   // Get the symbol value
   if([my_option]_value == 1){
     ...
   }
   else{
     ...
   }
}
```

Warning: A Link-time option should avoid to be set to `0`. Some third-party linkers consider such symbols as undefined, even if they are declared.

6.6 VEE Port Qualification

6.6.1 Introduction

A VEE Port integrates one or more Foundation Libraries with their respective Abstraction Layers.

VEE Port Qualification is the process of validating the conformance of the Abstraction Layer that implements the *Low Level APIs* of a Foundation Library.

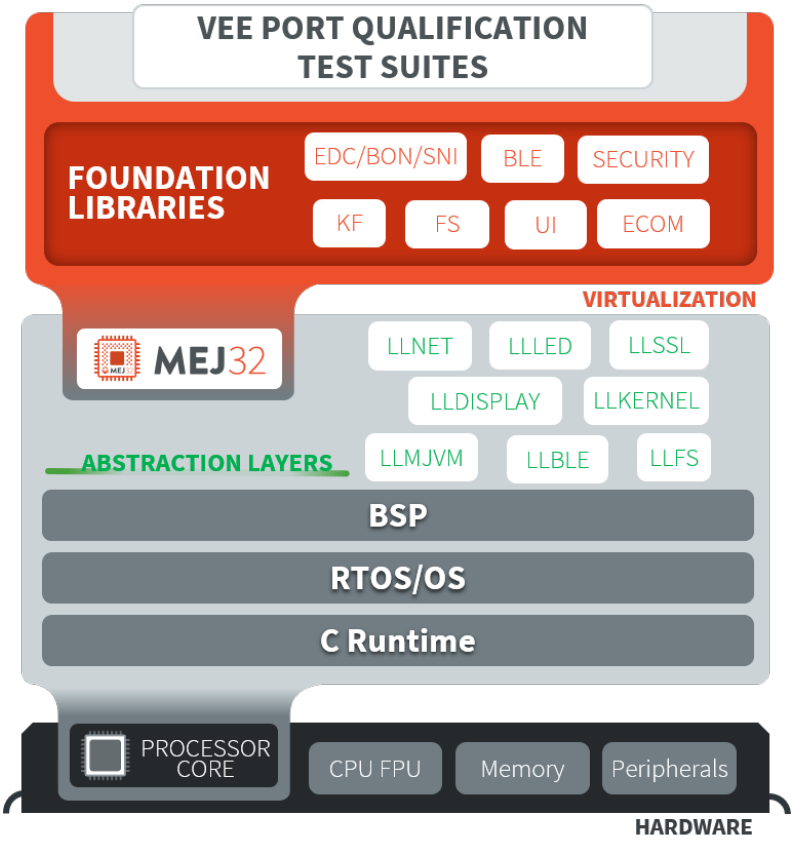


Fig. 14: VEE Port Qualification Overwiew

For each Low Level API, an Abstraction Layer implementation is required. The validation of the Abstraction Layer implementation is performed by running tests at two-levels:

- In C, by calling Low Level APIs (usually manually).
- In Java, by calling Foundation Library APIs (usually automatically using *VEE Port Test Suite*).

The following figure depicts an example for the FS Pack:

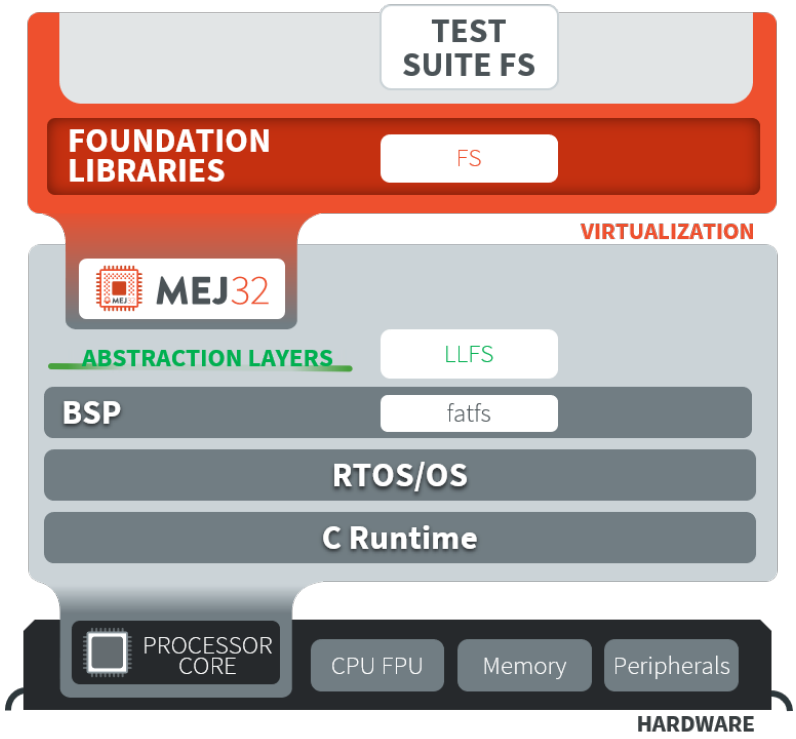


Fig. 15: VEE Port Qualification Example for FS Pack

MicroEJ provides a set of tools and pre-defined projects aimed at simplifying the steps for validating VEE Ports in the form of the VEE Port Qualification Tools (PQT).

6.6.2 VEE Port Qualification Tools Overview

The VEE Port Qualification Tools provide the following components:

- Platform Configuration Additions (PCA):
  - Used to:
    * Manage Architecture, Packs dependencies and the VEE Port build with the MicroEJ Module Manager.
    * Configure the BSP connection to call the build and run scripts.
  - Added when creating a VEE Port (see *Platform Creation* or check the tutorial *Create a MicroEJ Firmware From Scratch*).
- Build and Run Scripts examples:
  - Used to generate and deploy an Executable on a device by invoking a third-party toolchain for the BSP.
  - Added when integrating the BSP to the VEE Port (see *Build Script File* and *Run Script File* or check the tutorial *Create MicroEJ Platform Build and Run Scripts*).
- C and Java Test Suites:
  - Used to validate the Low Level APIs implementations.

- Validated during the BSP development and whenever an Abstraction Layer implementation is added or changed (see *VEE Port Test Suite* or check the tutorial *Run a Test Suite on a Device*).

Please refer to the VEE Port Qualification Tools README for more details and the location of the components.

6.6.3 VEE Port Test Suite

The purpose of a VEE Port Test Suite is to validate the Abstraction Layer that implements the *Low Level APIs* of a Foundation Libraries by automatically running Java tests on the device.

The *MicroEJ Test Suite Engine* is used for building, running a Test Suite, and providing a report.

A Test Suite contains one or more tests. For each test, the Test Suite Engine will:

1. Build an Executable for the test.
2. Run the Executable onto the device.
3. Retrieve the execution traces.
4. Analyze the traces to determine whether the test has `PASSED` or `FAILED`.
5. Append the result to the Test Report.
6. Repeat until all tests of the Test Suite have been executed.

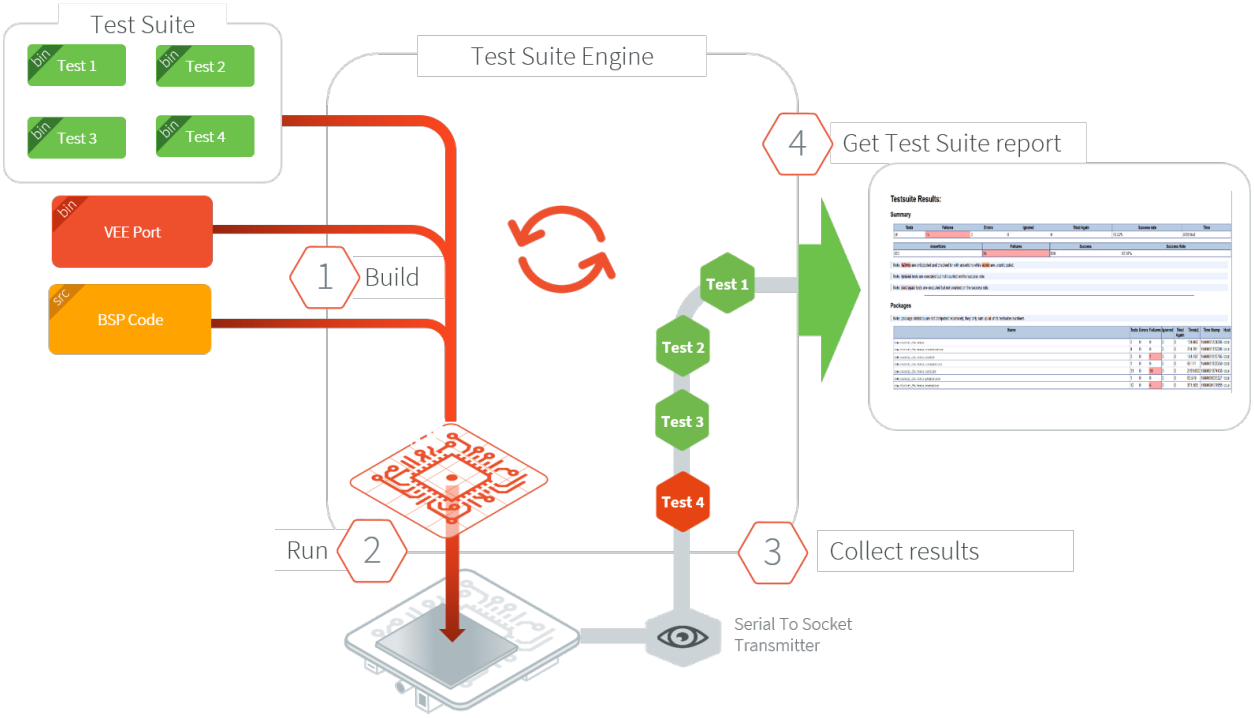


Fig. 16: VEE Port Test Suite on Device Overview

6.6.4 Create a VEE Port Test Suite

A VEE Port Test Suite is composed of two projects:

- the Test Suite module: the project that contains test cases. Test cases are written in *Junit*. When this project is built, it produces a versionned library. See *Test Suite Versioning* for available Test Suite modules for the most common Packs provided by MicroEJ Corp.
- the Test Suite runner: the project that contains the configuration for its execution on a VEE Port. When this project is built, it runs the Test Suite on a Device and generates the Test Suite report.

Note: Creating a VEE Port Test Suite requires SDK `5.6.0` or higher.

Create the Test Suite Module

The Test Suite module contains the tests of the Foundation Library to be qualified.

Create the Test Suite Module Project

A new Test Suite module is created using the `microej-javaimpl` Skeleton (see *Foundation Library Implementation*).

To create the Test Suite module, click on: File > New > Project... then select MicroEJ > Module Project

Fill up the following fields of the form:

- Project name (e.g: `myFoundationLib-testsuite`).
- Organization (e.g: `com.mycompany`).
- Module (e.g: `myFoundationLib-testsuite`).
- Revision (version of your Test Suite module).
- Select the Skeleton: `microej-javaimpl`.

Then, create two test source folders:

- Right-click on your project.
- Click on: New > Source Folder.
- Fill up the `Folder name` field of the form with: `src/test/java` and for the second folder: `src/test/resources`.

You should get a Foundation Library Test Suite project that looks like:

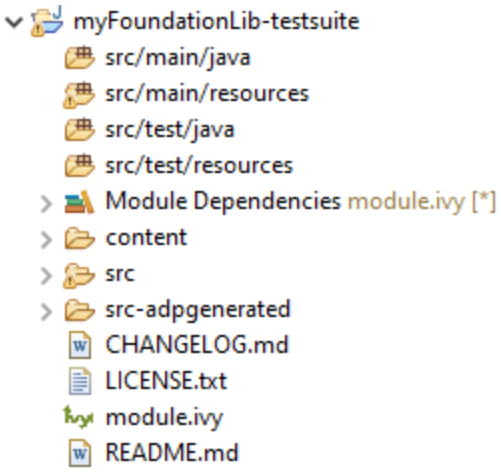


Fig. 17: Foundation Library Test Suite Project Skeleton

Your Test Suite module project is created and ready to be setup.

Configure the Test Suite Module Project

Open the `module.ivy` file and follow steps below:

- Edit the module `ivy-module > info > ea:build` node to update `rip.printableName` :

```
<ea:build organisation="com.is2t.easyant.buildtypes" module="build-microej-javaimpl"␣
↪microej.lib.name="myFoundationLib-testsuite-1.0" rip.printableName="myFoundationLib␣
↪Test Suite Impl" revision="5.2.+">
```

- Add the following properties in the `ivy-module > info` node:

```
<ea:property name="skip.test" value="set"/>
<ea:property name="target.main.classes" value="${basedir}/target~/test/classes"/>
<ea:property name="addon-processor.src.test.java.path.ref.name" value="src.java.path"/>
```

- Update the JUnit dependency to:

```
<dependency org="ej.library.test" name="junit" rev="1.7.1" conf="default;test->*"/>
```

- Add a `module.ant` file at the root of the Test Suite project with the following content:

```
<project>
        <target name="BuildTestTarget" extensionOf="abstract-compile:compile-ready"␣
↪depends="resources-std:copy-test-resources">
                <augment id="src.java.path">
                        <path location="${basedir}/src/test/java" />
                        <path location="${target}/adpgenerated/src-adpgenerated/junit/
↪java"/>
                </augment>
        </target>
</project>
```

Note: An error on `module.ant` file can occurred with message `Target resources-std:copy-test-resources does not exist in this project`. Please ignore it.

Create a New Test Case

Right click on `src/test/java`, then click on `New` > `Class`. Fill `Name:` with the `MyTest` and then click on `Finish`. Copy/paste the following example in `MyTest.java` file:

```
import org.junit.Assert;
import org.junit.Test;

public class MyTest {

	@Test
	public static void Test() {
		Assert.assertTrue(true);
	}
}
```

The console output on the Simulator for this test should be:

```
=============== [ Initialization Stage ] ===============
=============== [ Launching on Simulator ] ===============
OK: Test
PASSED: 1
=============== [ Completed Successfully ] ===============

SUCCESS
```

Build the Test Suite Module

Once the test cases are implemented, you can *build the module*. The next step is to create a Test Suite Runner. The Test Suite Runner will fetch the Test Suite Module dependency.

Create the Test Suite Runner

The Test Suite runner project contains configuration files for running a Test Suite module on a Device using a VEE Port.

Create the Test Suite Runner Project

- To create the Test Suite runner project, click on: File > New > Other... > MicroEJ > Module Project .
- Fill up the following fields of the form:
  - Project name
  - Organization
  - Module
  - Revision (version of your Test Suite module)
  - Select the Skeleton: `microej-testsuite`
- Inside the `module.ivy` file, add the dependency to the Test Suite module as following:

```
<dependency org="com.mycompany" name="myFoundationLib-testsuite" rev="0.1.0" conf="test-
↪>default;provided->provided"/>
```

- Inside the `module.ant` , add the following ANT target to configure trace redirection options :

```
<target name="tracefile:init" extensionOf="abstract-test:test-ready">
        <!-- Set the launch.test.trace.file when the testsuite.trace.ip properties is␣
↪not set -->
        <condition property="microej.testsuite.properties.launch.test.trace.file">
                <not>
                        <isset property="microej.testsuite.properties.testsuite.trace.ip
↪" />
                </not>
        </condition>
</target>
```

- Create the file `override.module.ant` at the root of the project. Add the following content to configure the load of testsuite options:

```
<project name="myFoundationlib.testsuite.override" xmlns:ac="antlib:net.sf.antcontrib">
        <!-- Load options from 'local.properties' beside this file -->
        <ac:if>
                <available file="local.properties" type="file"/>
                <ac:then>
                        <property file="local.properties"/>
                </ac:then>
        </ac:if>
        <!-- Load options from 'config.properties' beside this file -->
        <property file="config.properties"/>
</project>
```

- Create the following `.properties` files:

- `{PROJECT_LOC}/validation/microej-testsuite-common.properties` : see microej-testsuite-common.properties template.
- `{PROJECT_LOC}/config.properties` : see config.properties template.

Note: `{PROJECT_LOC}` refers here to the location of your Test Suite runner project.

Configure and Run the Test Suite

Follow the *Run a Test Suite on a Device* tutorial to configure your VEE Port and run the Test Suite on your Device.

6.6.5 Test Suite Versioning

Foundation Libraries are integrated in a VEE Port using Packs (see *Pack Import*). Use the Test Suite version compliant with the API version provided by the Foundation Library to validate the Abstraction Layer implementation. For example, the Test Suite FS module 3.0.3 should be used to validate the Abstraction Layer implementation of the *Low Level API FS* provided by the FS Pack 5.1.2.

Note: A Pack can provide several Foundation Libraries.

Core Engine

Table 4: Core Engine Validation

Architecture	Test Suite
7.0.0 or higher	Core Engine Test Suite

UI Pack

Table 5: UI Validation

UI Pack	C Test Suite
13.0.0 or higher (UI3)	Graphical User Interface Test Suite
[6.0.0-12.1.5] (UI2)	Graphical User Interface Test Suite

FS Pack

Table 6: FS API Implementation and Validation

FS Pack	FS API	Java Test Suite
[6.0.0-6.1.0[	2.1.1	3.0.8
[5.1.2-5.2.0[	2.0.6	3.0.3
[4.0.0-4.1.0[	2.0.6	On demand[1]

[1] Test Suite available on demand, please contact *MicroEJ Support*.

BLUETOOTH Pack

Table 7: BLUETOOTH API Implementation and Validation

BLUETOOTH Pack	BLUETOOTH API	Java Test Suite
2.1.0	2.1.0	2.0.0
2.0.1	2.0.0	2.0.0

NET Pack

Table 8: NET, SSL and SECURITY APIs Implementations and Validations

NET Pack	NET API	SSL API	SECURITY API	NET Java Test Suite	SSL Java Test Suite	SECURITY Java Test Suite
[8.1.2-8.2.0]	1.1.0	2.1.0	N/A	3.4.0 (On demandPage 750, 1)	3.0.1 (On demandPage 750, 1)	N/A
9.0.0	1.1.0	2.2.0	1.3.1	3.4.0 (On demandPage 750, 1)	3.1.4 (On demandPage 750, 1)	1.1.0 (On demandPage 750, 1)
[9.0.1-9.4.1]	1.1.1	2.2.0	1.3.1	3.5.2 (On demandPage 750, 1)	3.1.4 (On demandPage 750, 1)	1.1.0 (On demandPage 750, 1)
[10.0.0-10.5.0]	1.1.4	2.2.3	1.4.2	4.1.1	4.0.1	1.3.1
[11.0.1-11.0.2]	1.1.4	2.2.3	1.4.2	4.1.1	4.0.2	1.3.1

EVENT QUEUE Pack

Table 9: EVENT QUEUE API Implementation and Validation

EVENT QUEUE Pack	EVENT QUEUE API	Java Test Suite
2.0.1	2.0.0	2.0.0

6.7 Core Engine

The Core Engine is the core component of the Architecture. It executes at runtime the Application code.

6.7.1 Block Diagram

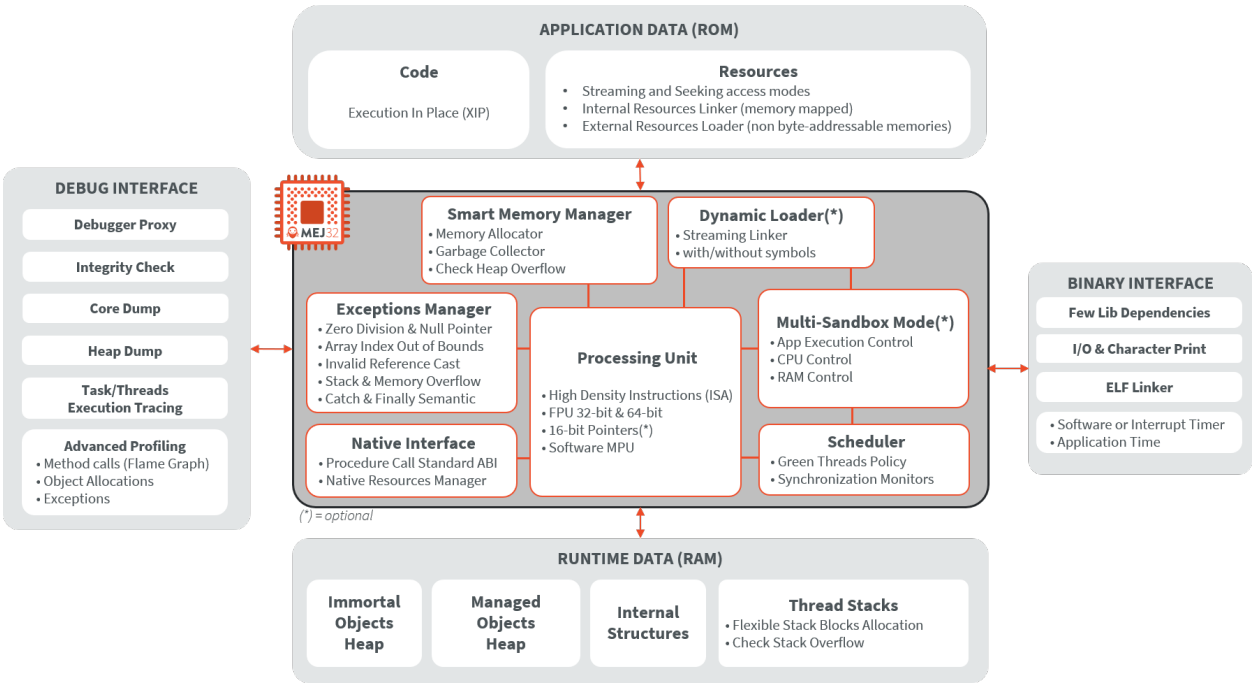


Fig. 18: Core Engine Block Diagram

6.7.2 Link Flow

The following diagram shows the overall build flow. Application development is performed within MICROEJ SDK. The remaining steps are performed within the C third-party IDE.

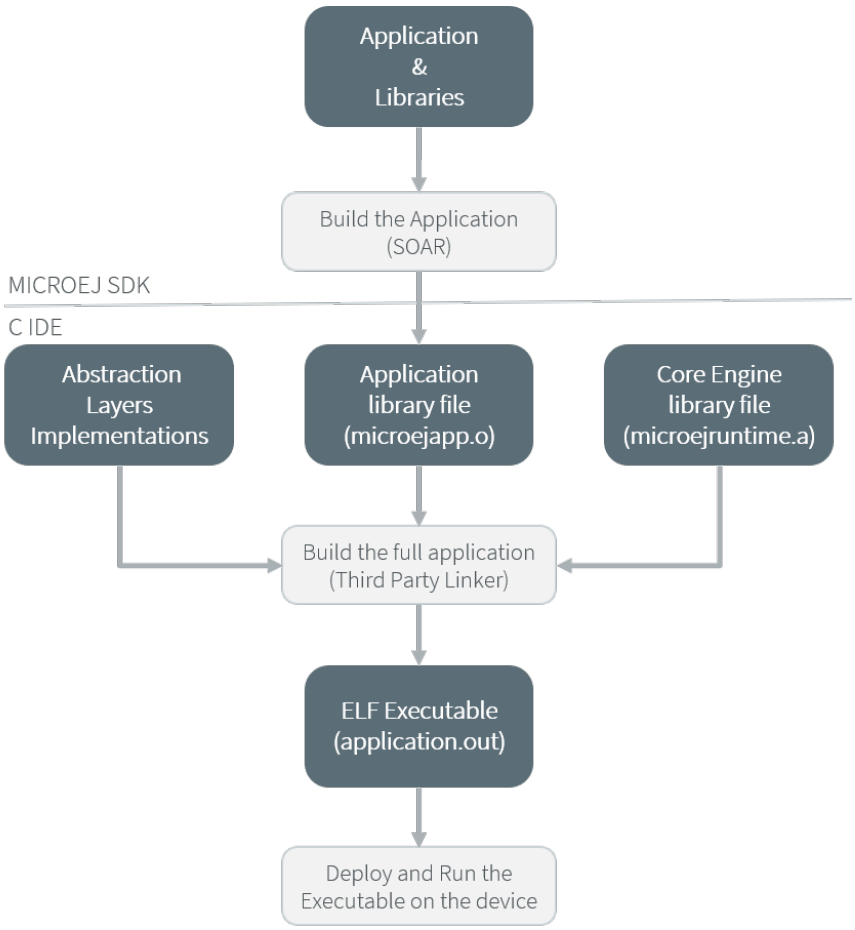


Fig. 19: Core Engine Flow

1. Step 1 consists in writing an Application against a set of Foundation Libraries available in the VEE Port.
2. Step 2 consists in compiling the Application code and the required libraries in an ELF library, using the SOAR.
3. Step 3 consists in linking the previous ELF file with the Core Engine library and a third-party BSP (OS, drivers, etc.). This step requires a third-party linker provided by a C toolchain.

6.7.3 Architecture

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

The Core Engine implements a *green thread architecture*. It runs in a single RTOS task.

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 Core Engine.

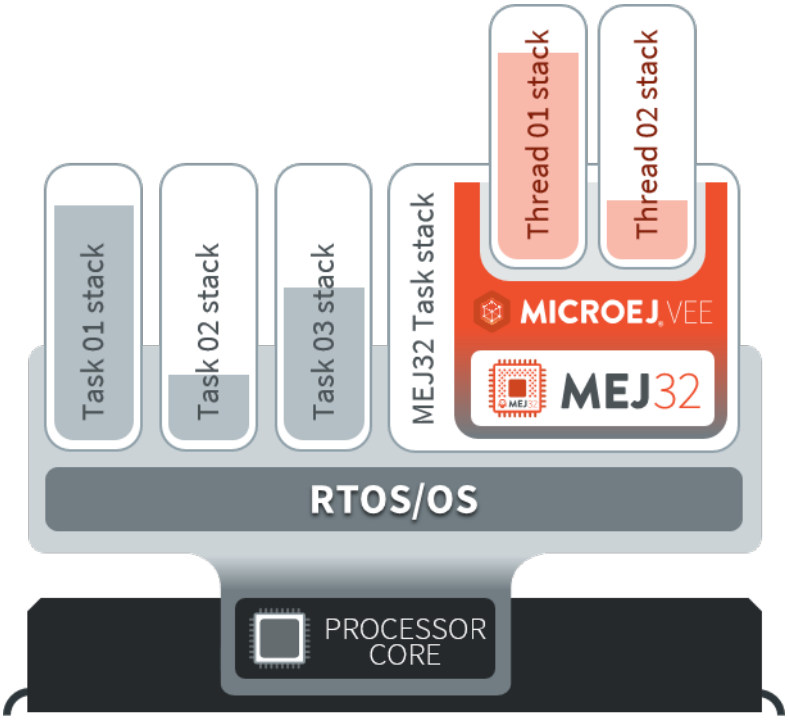


Fig. 20: A Green Threads Architecture Example

The activity of the Core Engine is defined by the Application. When the Application is blocked (i.e., when all the MicroEJ threads sleep), the RTOS task running the Core Engine sleeps.

6.7.4 Capabilities

The Core Engine defines 3 exclusive capabilities:

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

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

To select the Core Engine capability, create the property file `mjvm/mjvm.properties` in the Platform configuration project and define the property `com.microej.runtime.capability` with one of the following values:

- `mono` for Mono-Sandbox (default value)
- `multi` for Multi-Sandbox
- `tiny` for Tiny-Sandbox

If the property `com.microej.runtime.capability` is not defined, the Mono-Sandbox Core Engine capability is used.

6.7.5 Implementation

The Core Engine implements the *[SNI] specification*. It is created and initialized with the C function `SNI_createVM`. Then it is started and executed in the current RTOS task by calling `SNI_startVM`. The function `SNI_startVM` returns when the Application exits or if an error occurs (see section *Error Codes*). The function `SNI_destroyVM` handles the Core Engine termination and must be called after the return of the function `SNI_startVM`.

Only one instance of the Core Engine can be created in the system, and both `SNI_createVM` and `SNI_destroyVM` should only be called once. When restarting the Core Engine, don't call `SNI_createVM` or `SNI_destroyVM` before calling `SNI_startVM` again. For more information, refer to the *Restart the Core Engine* section.

The file `LLMJVM_impl.h` that comes with the Architecture defines the API to be implemented. See section *LLMJVM: MicroEJ Core Engine*.

Initialization

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

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

Scheduling

To support the green thread round-robin policy, the Core Engine assumes there is an RTOS timer or some other mechanism that counts (down) and fires a call-back when it reaches a specified value. The Core Engine initializes the timer using the `LLMJVM_IMPL_scheduleRequest` function with one argument: the absolute time at which the timer should fire. When the timer fires, it must call the `LLMJVM_schedule` function, which tells the Core Engine to execute a green thread context switch (which gives another MicroEJ thread a chance to run).

When several MicroEJ threads with the same priority are eligible for execution, the round-robin algorithm will automatically switch between these threads after a certain amount of time, called the *time slice*. The time slice is expressed in milliseconds, and its default value is `20` ms. It can be configured at link time with the symbol `_java_round_robin_period`, defined in the *linker configuration file* `linkVMConfiguration.lscf` located in the VEE Port folder `/MICROJVM/link/`. To override the content of this file, create, in the VEE Port configuration project, a folder named `/dropins/MICROJVM/link/`, and copy into this folder the file `linkVMConfiguration.lscf` retrieved from an existing VEE Port. Since a symbol cannot be null, the actual time slice value in milliseconds is `_java_round_robin_period - 1`. Set the symbol to 1 (i.e., time slice to 0) to disable the round-robin scheduling.

> **Warning:** Modifying the time slice value is an advanced configuration that can impact the performances.
>
> Decreasing the time slice will increase the number of context switches. Therefore scheduler will use more CPU time.
>
> Increasing the time slice can create a latency with intensive threads monopolizing the CPU.

Idle Mode

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

Time

The Core Engine defines two different times:

- the application time: the difference, measured in milliseconds, between the current time and midnight, January 1, 1970, UTC.
- the monotonic time: this time always moves forward and is not impacted by application time modifications (NTP or Daylight Savings Time updates). It can be implemented by returning the running time since the start of the device.

The Core Engine relies on the following C functions to provide those times to the Application:

- `LLMJVM_IMPL_getCurrentTime` : must return the monotonic time in milliseconds if the given parameter is `1` , otherwise must return the application time in milliseconds. This function is called by the method java.lang.System.currentTimeMillis() It is also used by the Core Engine scheduler, and should be implemented efficiently.
- `LLMJVM_IMPL_getTimeNanos` : must return a monotonic time in nanoseconds.
- `LLMJVM_IMPL_setApplicationTime` : must set the difference between the current time and midnight, January 1, 1970, UTC. Implementations may apply this time to the whole underlying system or only to the Core Engine (i.e., the value returned by `LLMJVM_IMPL_getCurrentTime(0)`).

Error Codes

The C function `SNI_createVM` returns a negative value if an error occurred during the Core Engine initialization or execution. The file `LLMJVM.h` defines the Core Engine error code constants. The following table describes these error codes.

Table 10: Core Engine Error Codes

Error Code	Meaning
0	The Application ended normally (i.e., all the non-daemon threads are terminated or `System.exit(exitCode)` has been called). See section *Exit Codes*.
-1	The `microejapp.o` produced by SOAR is not compatible with the Core Engine (`microejruntime.a`). The object file has been built from another Architecture.
-2	Internal error. Invalid link configuration in the Architecture or the VEE Port.
-3	Evaluation version limitations reached: termination of the application. See section *Limitations*.
-5	Not enough resources to start the very first MicroEJ thread that executes `main` method. See section *Option(text): Java heap size (in bytes)*.
-12	Number of threads limitation reached. See sections *Limitations* and *Option(text): Number of threads*.
-13	Fail to start the Application because the specified managed heap is too large or too small. See section *Option(text): Java heap size (in bytes)*.
-14	Invalid Application stack configuration. The stack start or end is not eight-byte aligned, or stack block size is too small. See section *Option(text): Number of blocks in pool*.
-16	The Core Engine cannot be restarted.
-17	The Core Engine is not in a valid state because of one of the following situations: • `SNI_startVM` called before `SNI_createVM`. • `SNI_startVM` called while the Appplication is running. • `SNI_createVM` called several times.
-18	The memory used for the managed heap or immortal heap does not work properly. Read/Write memory checks failed. This may be caused by an invalid external RAM configuration. Verify `_java_heap` and `_java_immortals` sections locations.
-19	The memory used for the Application static fields does not work properly. Read/Write memory checks failed. This may be caused by an invalid external RAM configuration. Verify `.bss.soar` section location.
-20	KF configuration internal error. Invalid link configuration in the Architecture or the VEE Port.
-21	Number of monitors per thread limitation reached. See sections *Limitations* and *Options* .
-22	Internal error. Invalid FPU configuration in the Architecture.
-23	The function `LLMJVM_IMPL_initialize` defined in the Abstraction Layer implementation returns an error.
-24	The function `LLMJVM_IMPL_vmTaskStarted` defined in the Abstraction Layer implementation returns an error.
-25	The function `LLMJVM_IMPL_shutdown` defined in the Abstraction Layer implementation returns an error.

Example

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

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

#ifdef __cplusplus
   extern "C" {
#endif

/**
 * @brief Creates and starts a MicroEJ instance. This function returns when the MicroEJ␣
↪execution ends.
 * @param argc arguments count
 * @param argv arguments vector
 * @param app_exit_code_ptr pointer where this function stores the application exit code or␣
↪0 in case of error in the Core Engine. May be null.
 * @return the Core Engine error code in case of error, or 0 if the execution ends without␣
↪error.
 */
int microej_main(int argc, char **argv, int* app_exit_code_ptr) {
        void* vm;
        int core_engine_error_code = -1;
        int32_t app_exit_code = 0;
        // create Core Engine
        vm = SNI_createVM();

        if (vm == NULL) {
                printf("MicroEJ initialization error.\n");
        } else {
                printf("MicroEJ START\n");

                // Error codes documentation is available in LLMJVM.h
                core_engine_error_code = (int)SNI_startVM(vm, argc, argv);

                if (core_engine_error_code < 0) {
                        // Error occurred
                        if (core_engine_error_code == LLMJVM_E_EVAL_LIMIT) {
                                printf("Evaluation limits reached.\n");
                        } else {
                                printf("MicroEJ execution error (err = %d).\n", (int) core_
↪engine_error_code);
                        }
                } else {
                        // Core Engine execution ends normally
                        app_exit_code = SNI_getExitCode(vm);
                        printf("MicroEJ END (exit code = %d)\n", (int) app_exit_code);
                }
```

(continues on next page)

(continued from previous page)

```
                // delete Core Engine
                SNI_destroyVM(vm);
        }

        if(app_exit_code_ptr != NULL){
                *app_exit_code_ptr = (int)app_exit_code;
        }

        return core_engine_error_code;
}

#ifdef __cplusplus
   }
#endif
```

Restart the Core Engine

The Core Engine supports the restart of the Application after the end of its execution. The application stops when all non-daemon threads are terminated or when `System.exit(exitCode)` is called. When the application ends, the C function `SNI_startVM` returns.

To restart the application, call again the `SNI_startVM` function (see the following pattern).

```
// create Core Engine (called only once)
vm = SNI_createVM();
...
// start a new execution of the Application at each iteration of the loop
while(...){
        ...
        core_engine_error_code = SNI_startVM(vm, argc, argv);
        ...
        // Get exit status passed to System.exit()
        app_exit_code = SNI_getExitCode(vm);
        ...
}
...
// delete Core Engine (called before stopping the whole system)
SNI_destroyVM(vm);
```

Note: Please note that while the Core Engine supports restart, *MicroUI* does not. Attempting to restart the Application on a VEE Port with UI support may result in undefined behavior.

Dump the States of the Core Engine

The internal Core Engine function called `LLMJVM_dump` allows you to dump the state of all MicroEJ threads: name, priority, stack trace, etc. This function must only be called from the MicroJvm virtual machine thread context and only from a native function or callback. Calling this function from another context may lead to undefined behavior and should be done only for debug purpose.

This is an example of a dump:

```
================================== VM Dump ===================================
Java threads count: 3
Peak java threads count: 3
Total created java threads: 3
Last executed native function: 0x90035E3D
Last executed external hook function: 0x00000000
State: running
------------------------------------------------------------------------------
Java Thread[1026]
name="main" prio=5 state=RUNNING max_java_stack=456 current_java_stack=184

java.lang.MainThread@0xC0083C7C:
    at (native) [0x90003F65]
    at com.microej.demo.widget.main.MainPage.getContentWidget(MainPage.java:95)
        Object References:
            - com.microej.demo.widget.main.MainPage@0xC00834E0
            - com.microej.demo.widget.main.MainPage$1@0xC0082184
            - java.lang.Thread@0xC0082194
            - java.lang.Thread@0xC0082194
    at com.microej.demo.widget.common.Navigation.createRootWidget(Navigation.java:104)
        Object References:
            - com.microej.demo.widget.main.MainPage@0xC00834E0
    at com.microej.demo.widget.common.Navigation.createDesktop(Navigation.java:88)
        Object References:
            - com.microej.demo.widget.main.MainPage@0xC00834E0
            - ej.mwt.stylesheet.CachedStylesheet@0xC00821DC
    at com.microej.demo.widget.common.Navigation.main(Navigation.java:40)
        Object References:
            - com.microej.demo.widget.main.MainPage@0xC00834E0
    at java.lang.MainThread.run(Thread.java:855)
        Object References:
            - java.lang.MainThread@0xC0083C7C
    at java.lang.Thread.runWrapper(Thread.java:464)
        Object References:
            - java.lang.MainThread@0xC0083C7C
    at java.lang.Thread.callWrapper(Thread.java:449)
------------------------------------------------------------------------------
Java Thread[1281]
name="UIPump" prio=5 state=WAITING timeout(ms)=INF max_java_stack=120 current_java_stack=117
external event: status=waiting

java.lang.Thread@0xC0083628:
    at ej.microui.MicroUIPump.read(Unknown Source)
        Object References:
            - ej.microui.display.DisplayPump@0xC0083640
```

(continues on next page)

(continued from previous page)

```
    at ej.microui.MicroUIPump.run(MicroUIPump.java:176)
        Object References:
            - ej.microui.display.DisplayPump@0xC0083640
    at java.lang.Thread.run(Thread.java:311)
        Object References:
            - java.lang.Thread@0xC0083628
    at java.lang.Thread.runWrapper(Thread.java:464)
        Object References:
            - java.lang.Thread@0xC0083628
    at java.lang.Thread.callWrapper(Thread.java:449)
--------------------------------------------------------------------------------
Java Thread[1536]
name="Thread1" prio=5 state=READY max_java_stack=60 current_java_stack=57

java.lang.Thread@0xC0082194:
    at java.lang.Thread.runWrapper(Unknown Source)
        Object References:
            - java.lang.Thread@0xC0082194
    at java.lang.Thread.callWrapper(Thread.java:449)
================================================================================

============================== Garbage Collector ===============================
State: Stopped
Last analyzed object: null
Total memory: 15500
Current allocated memory: 7068
Current free memory: 8432
Allocated memory after last GC: 0
Free memory after last GC: 15500
================================================================================

============================== Native Resources ================================
Id         CloseFunc  Owner            Description
--------------------------------------------------------------------------------
================================================================================
```

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

Dump The State Of All MicroEJ Threads From A Fault Handler

It is recommended to call the `LLMJVM_dump` API as a last resort in a fault handler. Calling `LLMJVM_dump` is undefined if the VM is not paused. The call to `LLMJVM_dump` MUST be done last in the fault handler.

Trigger VM Dump From Debugger

To trigger a VM dump from the debugger, set the PC register to the `LLMJVM_dump` physical memory address.

The symbol for the `LLMJVM_dump` API is defined in the header file `LLMJVM.h`. Search for this symbol in the appropriate C toolchain `.map` file.

Note: `LLMJVM_dump` (in `LLMJVM.h`) needs to be called explicitly. A linker optimization may remove the symbol if it is not used anywhere in the code.

Requirements:

- Embedded debugger is attached and the processor is halted in an exception handler.
- A way to read stdout (usually UART).

Check Internal Structure Integrity

The internal Core Engine function called `LLMJVM_checkIntegrity` checks the internal structure integrity of the MicroJvm virtual machine and returns its checksum.

- If an integrity error is detected, the `LLMJVM_on_CheckIntegrity_error` hook is called and this method returns `0`.
- If no integrity error is detected, a non-zero checksum is returned.

This function must only be called from the MicroJvm virtual machine thread context and only from a native function or callback. Calling this function multiple times in a native function must always produce the same checksum. If the checksums returned are different, a corruption must have occurred.

Please note that returning a non-zero checksum does not mean the MicroJvm virtual machine data has not been corrupted, as it is not possible for the MicroJvm virtual machine to detect the complete memory integrity.

MicroJvm virtual machine internal structures allowed to be modified by a native function are not taken into account for the checksum computation. The internal structures allowed are:

- basetype fields in Java objects or content of Java arrays of base type,
- internal structures modified by a `LLMJVM` function call (e.g. set a pending Java exception, suspend or resume the Java thread, register a resource, ...).

This function affects performance and should only be used for debug purpose. A typical use of this API is to verify that a native implementation does not corrupt the internal structures:

```
void Java_com_mycompany_MyClass_myNativeFunction(void) {
            int32_t crcBefore = LLMJVM_checkIntegrity();
            myNativeFunctionDo();
    int32_t crcAfter = LLMJVM_checkIntegrity();
    if(crcBefore != crcAfter){
            // Corrupted MicroJVM virtual machine internal structures
            while(1);
    }
}
```

6.7.6 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.

6.7.7 Link

Several sections are defined by the Core Engine. Each section must be linked by the third-party linker. Read-Only (RO) sections can be placed in writable memories. In such cases, it is the responsibility of the BSP to prevent these sections from being written.

Starting from *Architecture 8.0.0*, sections have been renamed to follow the standard ELF naming convention.

Linker Sections (Architecture `8.x`)

Linker Sections (Architecture `7.x`)

Section name	Aim	Location	Alignment (in bytes)
`.bss.microej.heap`	Application heap	RW	4
`.bss.microej.immortals`	Application immortal heap	RW	4
`.bss.microej.stacks`	Application threads stack blocks	RW[1]	8
`.bss.microej.statics`	Application static fields	RW	8
`.rodata.microej.resource.*`	Application resources (one section per resource)	RO	16
`.rodata.microej.soar`	Application and library code	RO	16
`.bss.microej.runtime`	Core Engine internal structures	RW	8
`.text.__icetea__*`	Core Engine generated code	RX	ISA Specific
`.bss.microej.kernel`	Core Engine Multi-Sandbox section (Feature code chunk)	RW	4

Note: During its startup, the Core Engine automatically zero-initializes the sections `.bss.microej.runtime`, `.bss.microej.heap`, and `.bss.microej.immortals`.

[1] Among all RW sections, those should be always placed into internal RAM for performance purpose.

Section name	Aim	Location	Alignment (in bytes)
`_java_heap`	Application heap	RW	4
`_java_immortals`	Application immortal heap	RW	4
`.bss.vm.stacks.java`	Application threads stack blocks	RW[1]	8
`.bss.soar`	Application static fields	RW	8
`.rodata.resources`	Application resources	RO	16
`.text.soar`	Application and library code	RO	16
`ICETEA_HEAP`	Core Engine internal structures	RW	8
`.text.__icetea__*`	Core Engine generated code	RX	ISA Specific

Page 763, 1

Note: During its startup, the Core Engine automatically zero-initializes the sections `ICETEA_HEAP`, `_java_heap`, and `_java_immortals`.

6.7.8 Dependencies

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

6.7.9 Installation

The Core Engine and its components are mandatory. By default, it is configured with Mono-Sandbox capability. See the *Capabilities* section to update the Core Engine with Multi-Sandbox or Tiny-Sandbox capability.

6.7.10 Abstraction Layer

Core Engine Abstraction Layer implementations can be found on MicroEJ Github for several RTOS.

6.7.11 Memory Considerations

The memory consumption of main Core Engine runtime elements are described in *the table below*.

Table 11: Memory Considerations

Runtime element	Memory	Size in bytes (Mono-sandbox)	Size in bytes (Multi-Sandbox)	Size in bytes (Tiny-Sandbox)
Object Header	RW	4	8 (+4)	4
Thread	RW	168	192 (+24)	168
Stack Frame Header	RW	12	20 (+8)	12
Class Type	RO	32	36 (+4)	32
Interface Type	RO	16	24 (+8)	16

Note: To get the full size of an Object, search for the type in the *SOAR Information File* and read the attribute `instancesize` (this includes the Object header).

Note: To get the full size of a Stack Frame, search for the method in the *SOAR Information File* and read the attribute `stacksize` (this includes the Stack Frame header).

6.7.12 Use

Refer to the *MicroEJ Runtime* documentation.

6.8 Advanced Event Tracing

6.8.1 Principle

MicroEJ Core Engine allows method execution to be profiled. The following two new hooks functions are used for that:

- `LLMJVM_MONITOR_IMPL_on_invoke_method` called at the start of the method invocation.
- `LLMJVM_MONITOR_IMPL_on_return_method` called when returning from the invoked method.

Calling these functions each time a method is invoked will slow down the application execution, so these functions are not called by default when event tracing is enabled and started.

Note: This feature requires Architecture version `7.17.0` or higher and is only available on MicroEJ Core Engine, not on Simulator.

To activate them, you need to follow these steps:

- Enable and start the trace *see here*
- Tell the third-party linker program to redirect all calls to `LLMJVM_invoke_method` and `LLMJVM_return_method` symbols to respectively `LLMJVM_invoke_method_with_trace` and `LLMJVM_return_method_with_trace` symbols.

6.8.2 Platforms using GNU LD linker

Add the following options to the LD linker command line:

```
--require-defined=LLMJVM_invoke_method_with_trace
--defsym=LLMJVM_invoke_method=LLMJVM_invoke_method_with_trace
--require-defined=LLMJVM_return_method_with_trace
--defsym=LLMJVM_return_method=LLMJVM_return_method_with_trace
```

6.8.3 Platforms using IAR ILINK linker

Pass the following options to the ILINK linker program

```
--redirect LLMJVM_invoke_method=LLMJVM_invoke_method_with_trace
--redirect LLMJVM_return_method=LLMJVM_return_method_with_trace
```

6.9 Multi-Sandbox

6.9.1 Principle

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

The Core Engine implements the *[KF] specification*. A Kernel is a Standalone Application generated on a Multi-Sandbox-enabled VEE Port. A Feature is a Sandboxed Application generated against a specific Kernel.

6.9.2 Functional Description

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

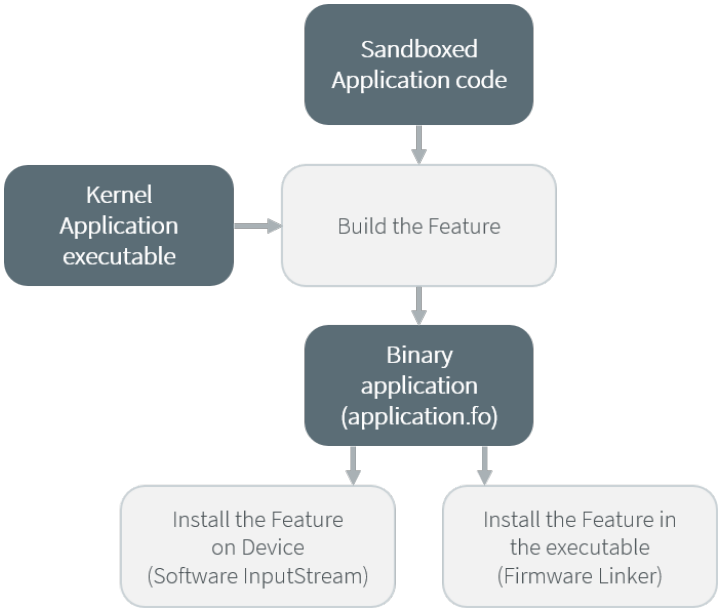


Fig. 21: Multi-Sandbox Process

Once a Kernel has been generated, additional Sandboxed Application code (Feature) can be built against the Kernel. The binary file produced (the `.fo` file) can be installed on the Kernel on which it was generated.

For more details on the build flow, please refer to *Multi-Sandbox Kernel link* and *Sandboxed Application link* sections.

6.9.3 Memory Considerations

Multi-Sandbox memory overhead of Core Engine runtime elements are described in *Memory Considerations table*.

6.9.4 Dependencies

- `LLKERNEL_impl.h` implementation (see *Feature Installation* section).

6.9.5 Installation

Multi-Sandbox is an option disabled by default. To enable the Multi-Sandbox capability of the Core Engine, set the property `com.microej.runtime.capability` to `multi` in `mjvm/mjvm.properties` file. See the example below:

```
com.microej.runtime.capability=multi
```

Note: Before *Architecture 8.1.0*, to enable the Multi-Sandbox capability of the Core Engine, select the Multi Applications module in the platform configuration file.

6.9.6 Use

The KF API Module must be added to the *module.ivy* of the Application project to use *[KF]* library.

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

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

6.9.7 Feature Installation

Introduction

Feature installation is triggered by a call to the Kernel.install(InputStream) method. It consists of the following steps:

- loading Feature's content from `.fo` file,
- linking Feature's code with the Kernel,
- storing Feature's content into the target memory.

A Feature `.fo` file is composed of the following elements:

- Code: Application code (methods, types, ...) as well as built-in objects (strings and immutables),
- RO Data: *Application Resources* that do not require content modification,
- RW Data: Reserved memory for Feature execution (Application static fields and Feature internal structures),
- Metadata: Temporary information required during the installation phase, such as code relocations.

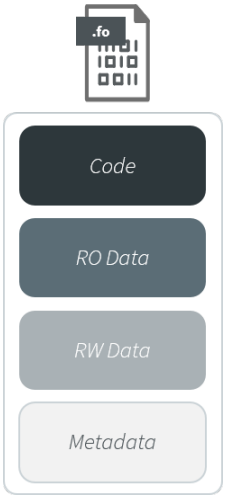


Fig. 22: Feature `.fo` File Content

Feature installation flow allows to install Features in any byte-addressable memory mapped to the CPU's address space. The Feature content is read chunk-by-chunk from the InputStream and progressively transferred to the target memory. Only a small amount of RAM is required. The `LLKERNEL_impl.h` Abstraction Layer interface provides Low Level APIs for allocating and transferring Feature content in different memory areas, including ROM.

Installation Flow

The RO Data (Application Resources) is directly transferred to the target location. The Code is divided into chunks. Each chunk is temporarily copied to RAM to be relocated. Then it is transferred to the target location.

A minimum amount of RAM is required:

- A temporary buffer is allocated in the Java heap for reading bytes from the InputStream,
- Metadata is allocated in the Java heap,
- Code chunk is temporarily copied in a memory area to be relocated (see more details below).

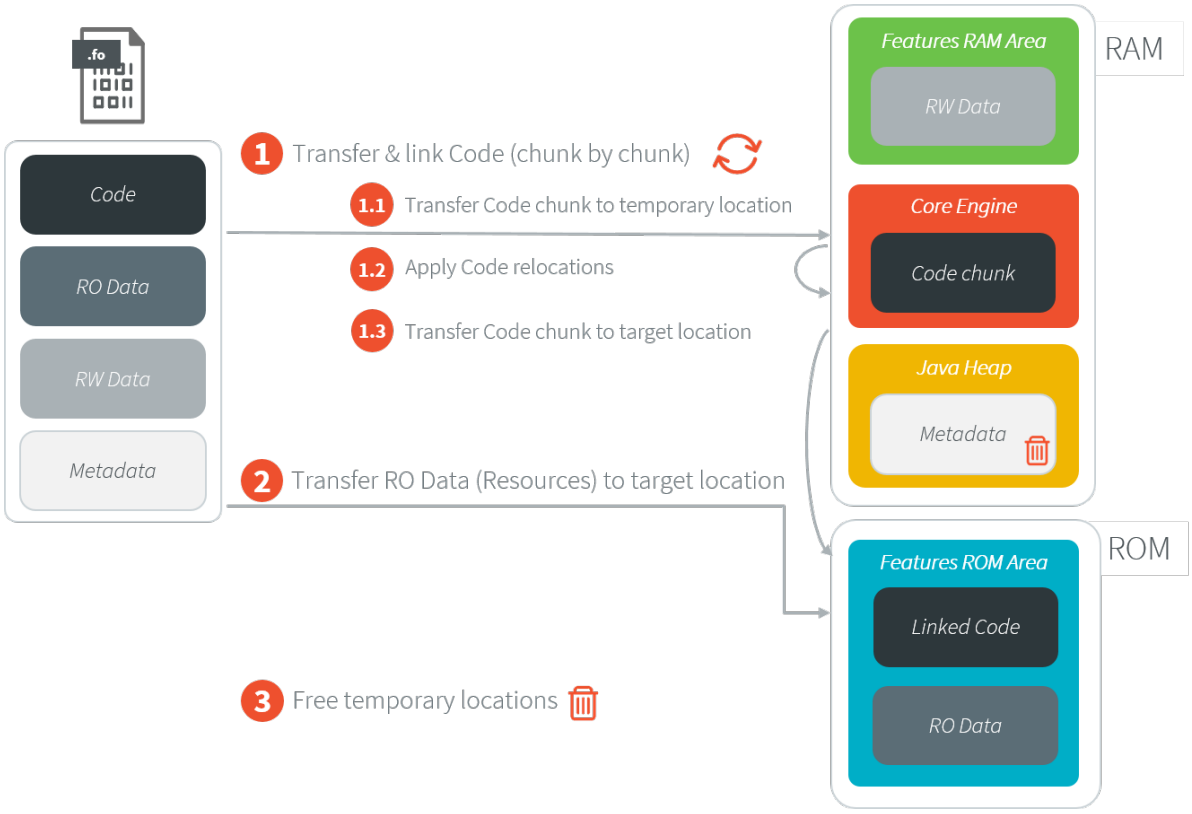


Fig. 23: Feature Installation Steps

The Abstraction Layer implementation is responsible for providing the following elements:

- the location where the Feature will be installed,
- the implementation to copy a chunk of bytes to the target location.

The detailed installation flow is described in the following sequence diagram:

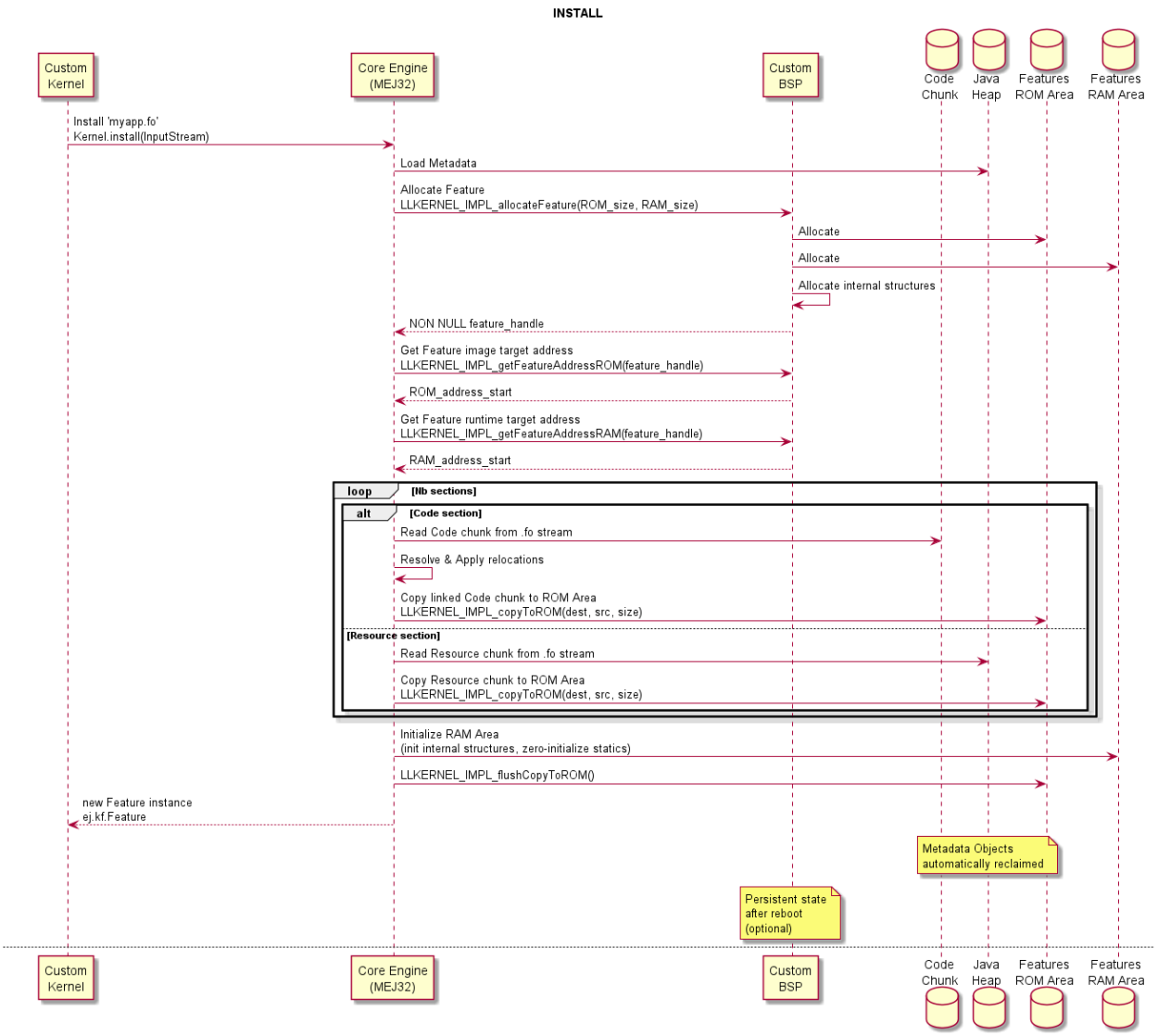


Fig. 24: Feature Installation Flow

The detailed uninstallation flow is described in the following sequence diagram:

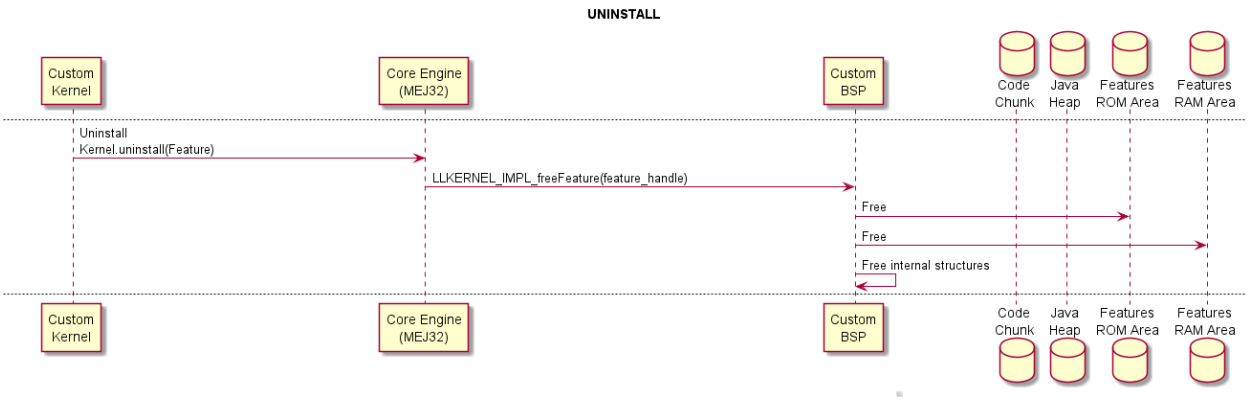


Fig. 25: Feature Uninstallation Flow

Feature Persistency

Feature Persistency is the ability of the Core Engine to gather installed Features from prior executions of the Kernel upon start up. This means that the Kernel will boot with a set of available Features that were already installed. To ensure that the Features remain available even after the device restarts, you will have to implement an Abstraction Layer that stores the Features into a Read-Only memory.

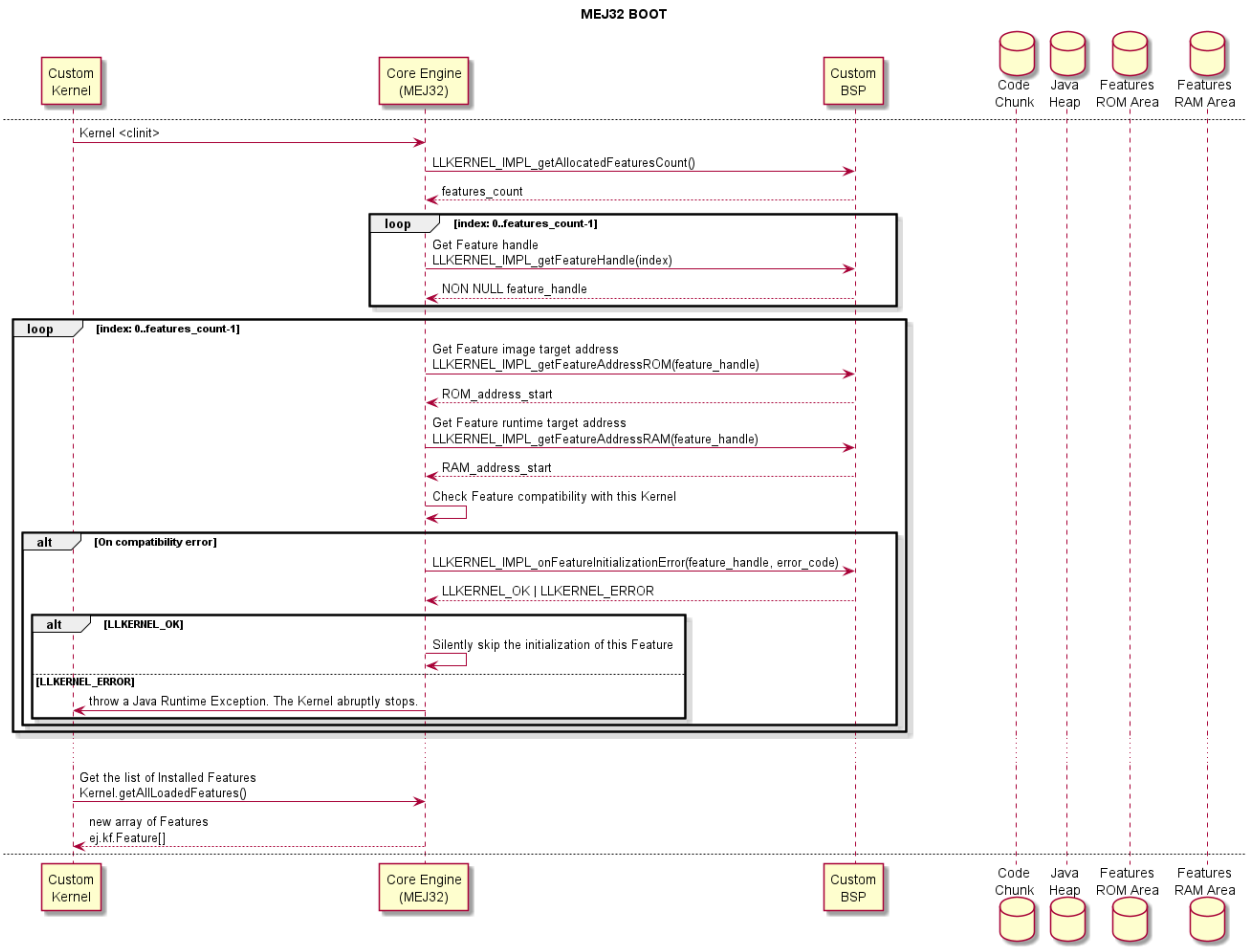


Fig. 26: Feature Installation Boot Flow

Note: Features are available in the *INSTALLED* state. It is the responsibility of the Kernel to manually start the desired Features.

Advanced Options

Code Chunk Size

Feature `.fo` Code section is divided into chunks that are temporary copied to RAM to be relocated. The Code chunk size can be configured with the following option:

Option Name: `com.microej.soar.kernel.featurecodechunk.size`

Default Value: `65536` (bytes)

A small number will reduce the RAM consumption but will increase the `.fo` size and will affect the installation time.

InputStream Transfer Buffer Size

When calling the Kernel.install(InputStream) method, the Feature `.fo` bytes are read from the InputStream using a temporary byte array allocated in the Java Heap. The size of this array can be configured with the following option:

Option Name: `com.microej.runtime.kf.link.transferbuffer.size`

Default Value: `512` (bytes)

Relocation Process Yield

When a Feature file has a large amount of code, it may appear that the Core Engine blocks while applying relocations during the Feature installation. The number of relocations to apply in batch can be configured with the following option:

Option Name: `com.microej.runtime.kf.link.chunk.relocations.count`

Default Value: `128`

Once the Core Engine has processed the given number of relocations, the thread that called the Kernel.install(InputStream) method yields the execution to other threads. A small number will give more smooth execution for threads but a slowest installation execution. A large number will make the Core Engine block for applying relocations but a faster installation execution.

Determining the Amount of Required Memory

The amount of memory required for installing a `.fo` file is determined by analyzing the sizes of the ELF sections.

Sections can be dumped using the standard binutils `readelf` tool:

```
readelf -WS application.fo
There are 8 section headers, starting at offset 0x34:

Section Headers:
[Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
[ 0]                   NULL            00000000 000000 000000 00      0   0  0
[ 1] .soar.rel         LOPROC+0        00000000 000174 000bcc 00      6   0  4
[ 2] .strtab           STRTAB          00000000 000d40 000063 00      0   0  1
[ 3] .symtab           SYMTAB          00000000 000da4 000050 10      2   1  4
[ 4] .bss.soar.feature NOBITS          00000000 000df4 000050 00   A  0   0  4
[ 5] .rodata.microej.resources PROGBITS        00000000 000e00 079080 00   A  0   0 64
[ 6] .rodata           PROGBITS        00000000 079e80 001974 00   A  0   0 16
[ 7] .shstrtab         STRTAB          00000000 07b7f4 000059 00      0   0  1
```

The following table summarizes the sections and their content:

Section	Description	Temporary Memory Location	Target Memory Location
`.soar.rel`	Metadata	Java Heap	None
`.strtab`	Metadata	Java Heap	None
`.symbtab`	Metadata	Java Heap	None
`.bss.soar.feature`	RW Data	None	Features RAM area
`.rodata.microej.resources`	RO Data	None	Features ROM area
`.rodata`	Code chunk	RAM	Features ROM area
`.shstrtab`	Metadata	Java Heap	None

In-Place Installation

Note: This section describes the legacy Feature installation flow, based on a `malloc/free` implementation in RAM. It is deprecated and available up to *Architecture 8.0.0*.

See *Migrate Your LLKERNEL Implementation* for migrating to the latest installation flow.

Feature content is installed in RAM. The required memory is allocated in the Kernel Working Buffer. This includes code, resources, static fields, and internal structures. When the Feature is uninstalled, allocated memory is reclaimed. When the Core Engine or the device restarts, the Kernel Working Buffer is reset; thus there is no persistent Feature.

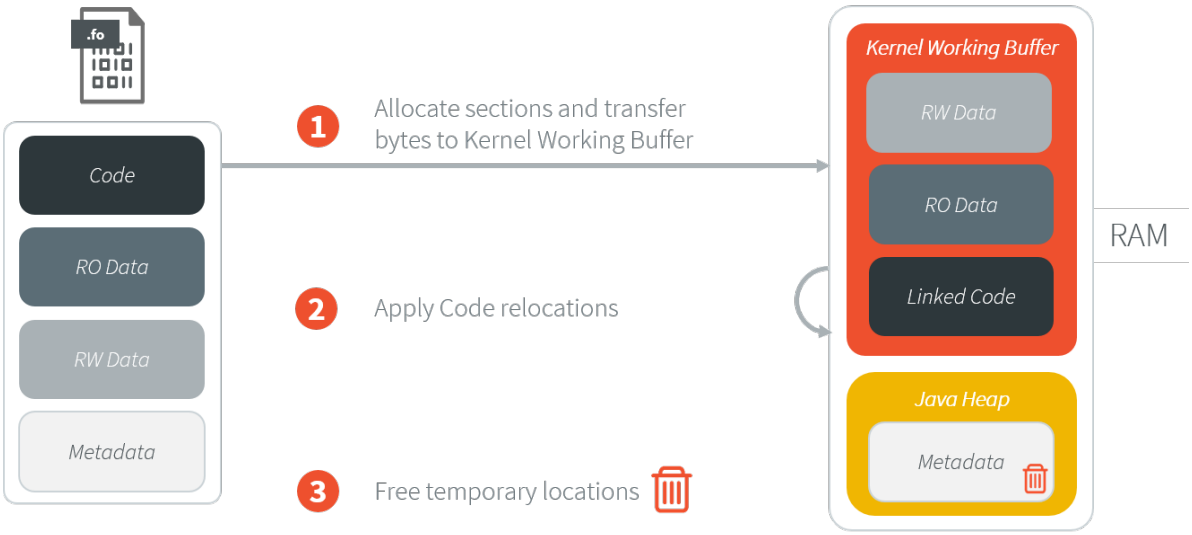


Fig. 27: In-Place Feature Installation Overview

The In-Place installation flow is described in the following sequence diagram:

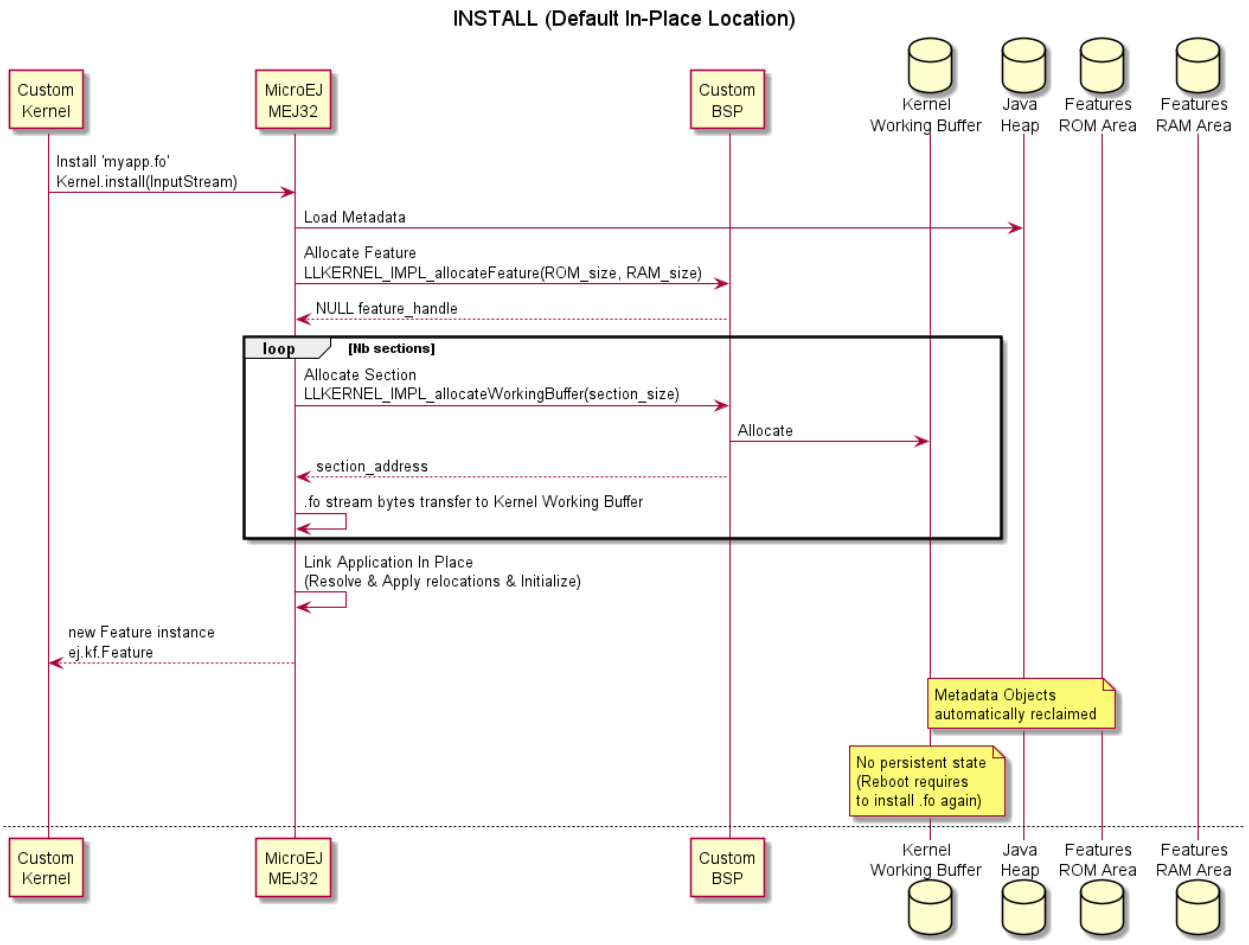


Fig. 28: In-Place Feature Installation Flow

The In-Place uninstallation flow is described in the following sequence diagram:

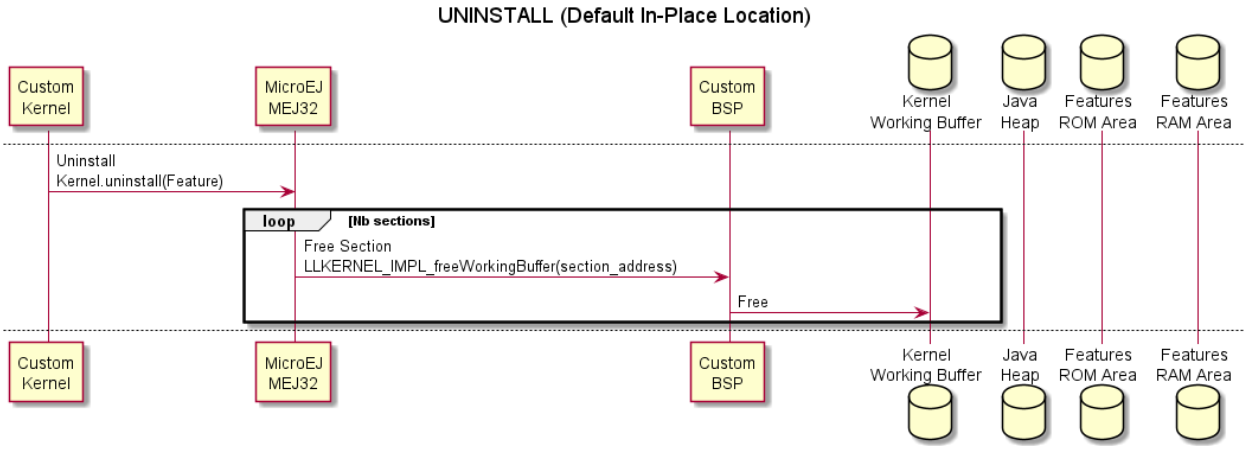


Fig. 29: In-Place Feature Uninstallation Flow

6.9.8 RAM Control

Note: This feature requires Architecture *8.1.0* or higher.

In a Multi-Sandbox environment, RAM Control automatically stops less critical Features when a more critical Feature cannot allocate new objects. See the *RAM Control: Feature Criticality* section of the *Kernel & Features Specification* for more details.

By default, RAM Control is disabled in the Core Engine. To enable it, set the property `com.microej.runtime.kf.ramcontrol.enabled` to `true` when building the VEE Port. This can be done by defining this property in the file `mjvm/mjvm.properties` of your VEE Port configuration project:

```
com.microej.runtime.kf.ramcontrol.enabled=true
```

When RAM Control is enabled, all Foundation Libraries must declare their native resources using SNI (see `sni.h` header file). This is necessary for the automatic release of native resources when the Core Engine abruptly stops a Feature to recover heap memory. Foundation Libraries can no longer register native resources using the deprecated class `ej.lang.ResourceManager`. Attempting to do so will result in an exception being thrown.

6.10 Tiny-Sandbox

6.10.1 Principle

The Tiny-Sandbox capability of the Core Engine allows to build a Standalone Application optimized for size. This capability is suitable for environments requiring a small memory footprint.

6.10.2 Installation

Tiny-Sandbox is an option disabled by default. To enable the Tiny-Sandbox capability of the Core Engine, set the property `com.microej.runtime.capability` to `tiny` in `mjvm/mjvm.properties` file. See the example below:

```
com.microej.runtime.capability=tiny
```

Note: Before *Architecture 8.1.0*, enabling the Tiny-Sandbox capability was done by setting the property `mjvm.standalone.configuration` in the `configuration.xml` file as follows:

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

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

6.10.3 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.
- Incompatible with dynamic linkers enabling Address Space Layout Randomization (ASLR).

6.11 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.

6.11.1 Simple Native Interface (SNI)

Principle

[SNI] specification 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.

Functional Description

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

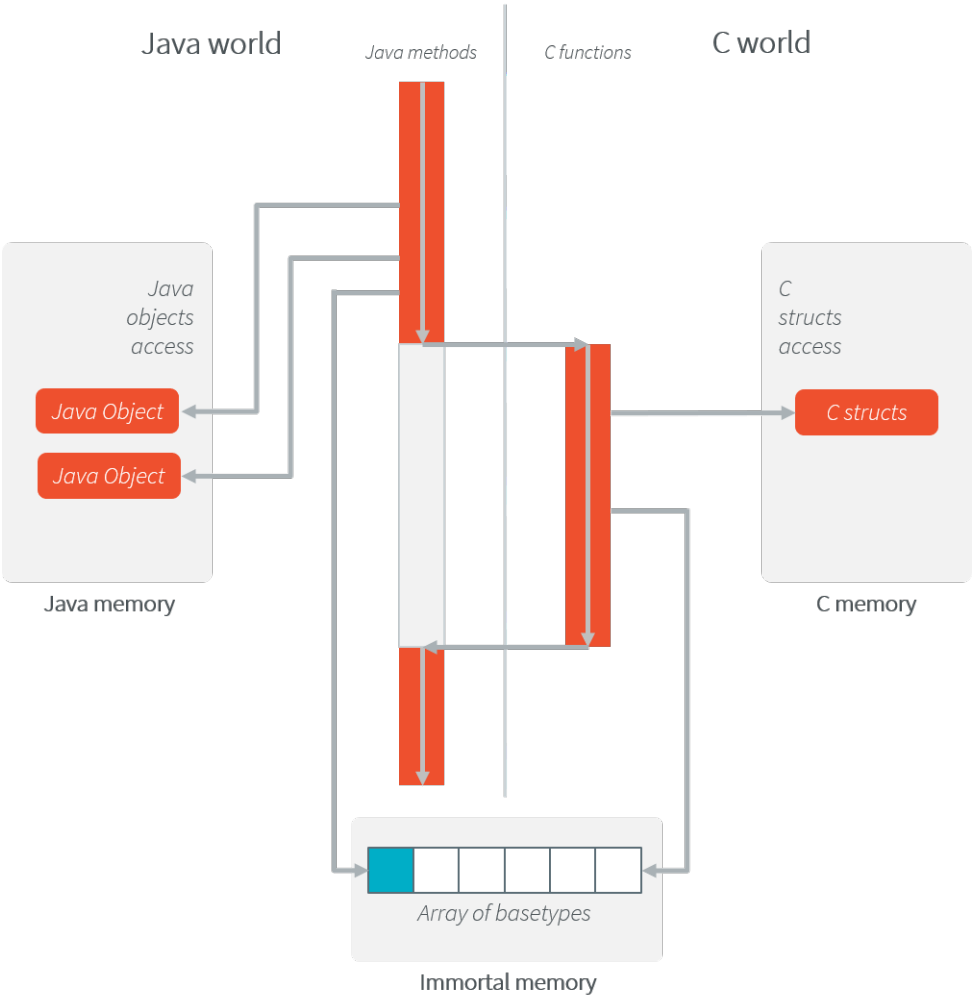


Fig. 30: *[SNI]* Processing

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 {
```

(continues on next page)

(continued from previous page)

```
    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.

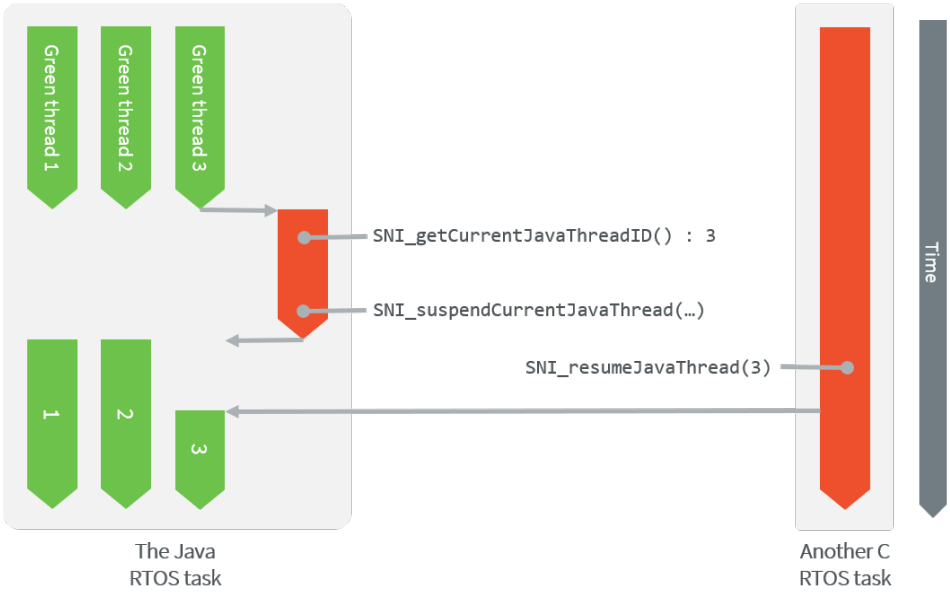


Fig. 31: Green Threads and RTOS Task Synchronization

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 Architecture, 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 Application project to use the *[SNI]* library.

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

6.11.2 Shielded Plug (SP)

Principle

The Shielded Plug (SP) library provides data segregation with a clear publish-subscribe API. The data-sharing between modules uses the concept of shared memory blocks, with introspection. The database is made of blocks: chunks of RAM.

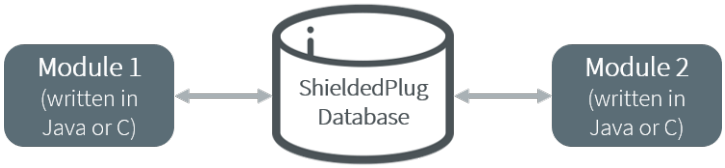


Fig. 32: A Shielded Plug Between Two Application (Java/C) Modules.

Documentation	Link
Java APIs	https://repository.microej.com/javadoc/microej_5.x/apis/ej/sp/package-summary.html
Specification	https://repository.microej.com/packages/ESR/ESR-SPE-0014-SP-2.0-B.pdf
Module	https://repository.microej.com/modules/ej/api/sp/

Functional Description

The usage of the Shielded Plug (SP) starts with the definition of a database. The implementation uses an XML file description to describe the database; the syntax follows the one proposed by the [SP] specification.

Once this database is defined, it can be accessed within the MicroEJ Application or the C application. The `SP` Foundation Library is accessible from the *[SP] API Module*. This library contains the classes and methods to read and write data in the database. The C header file `sp.h` available in the MicroEJ Platform `source/include` folder contains the C functions for accessing the database.

To embed the database in your binary file, the XML file description must be processed by the *SP compiler*. This compiler generates a binary file (`.o`) that will be linked to the overall application by the linker. It also generates two descriptions of the block ID constants, one in Java 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 compiler. The tool name is `Shielded Plug Compiler` . It outputs:

- A description of the requested resources of the database as a binary file (`.o`) that will be linked to the overall application by the linker. It is an ELF format description that reserves both the necessary RAM and the necessary Flash memory for the Shielded Plug database.
- Two descriptions, one in Java and one in C, of the block ID constants to be used by either Java or C application modules.

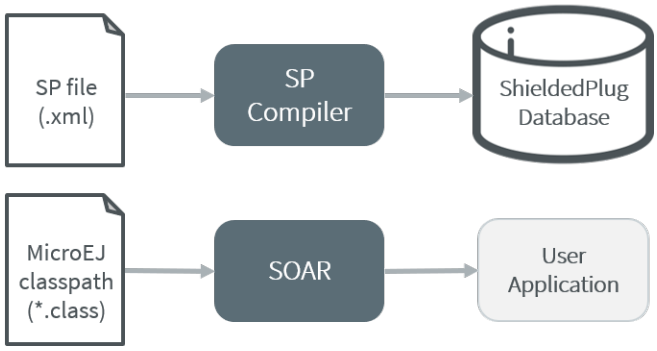


Fig. 33: Shielded Plug Compiler Process Overview

Example

Below is an example of using a database. The code that publishes the data is written in C, and the code that receives the data is written in Java. 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);
```

(continues on next page)

(continued from previous page)

```
            print(temp);
            //update the thermostat status
            database.writeInt(Forecast.THERMOSTAT,temp>tempLimit ? 0 : 1);
        }
        catch(EmptyBlockException e){
            print("Temperature not available");
        }
        sleep(30000);
    }
}
```

C Code

Here is a C header that declares the constants defined in the XML description of the database.

```
#define Forecast_ID 0
#define Forecast_TEMP 1
#define Forecast_THERMOSTAT 2
```

Below, the code shows the publication of the temperature and thermostat controller task.

```
void temperaturePublication() {
   ShieldedPlug database = SP_getDatabase(Forecast_ID);
   int32_t temp = temperature();
   SP_write(database, Forecast_TEMP, &temp);
}

void thermostatTask(){
   int32_t thermostatOrder;
   ShieldedPlug database = SP_getDatabase(Forecast_ID);
   while(1){
      SP_waitFor(database, Forecast_THERMOSTAT);
      SP_read(database, Forecast_THERMOSTAT, &thermostatOrder);
      if(thermostatOrder == 0) {
         thermostatOFF();
      }
      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 Application project.

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

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

6.11.3 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).

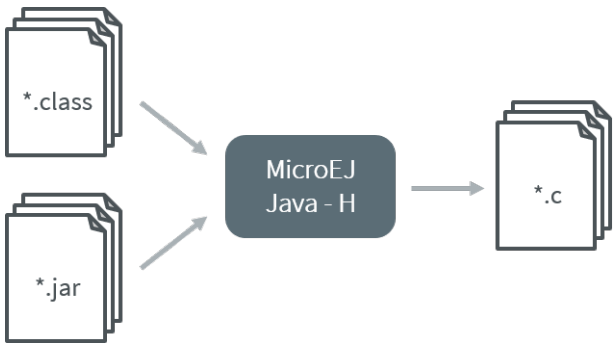


Fig. 34: MicroEJ Java H Process

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.

6.12 External Resources Loader

6.12.1 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).

See *Application Resources* for more information on how to declare external resources depending on its kind (raw resources, images, fonts, NLS).

6.12.2 Implementations

External Resources Loader module provides some Low Level API (LLEXT_RES) to let the BSP manage the external resources.

Open a Resource

The LLAPI to implement in the BSP are listed in the header file `LLEXT_RES_impl.h`. First, the framework tries to open an external resource using the `open` function. This function receives the resources path as a parameter. This path is the absolute path of the resource from the MicroEJ Application classpath (the MicroEJ Application source base directory). For example, when the resource is located here: `com.mycompany.myapplication.resource.MyResource.txt`, the given path is: `com/mycompany/myapplication/resource/MyResource.txt`.

The external resources loader implementation should, when possible, lock the resource when it is opened. Any modification of an opened resource may not be properly handled by the application.

Resource Identifier

This `open` function has to return a unique ID (positive value) for the external resource, or returns an error code (negative value). This ID will be used by the framework to manipulate the resource (read, seek, close, etc.).

Several resources can be opened at the same time. The BSP does not have to return the same identifier for two resources living at the same time. However, it can return this ID for a new resource as soon as the old resource is closed.

Resource Offset

The BSP must hold an offset for each opened resource. This offset must be updated after each call to `read` and `seek`.

Resource Inside the CPU Address Space Range

An external resource can be programmed inside the CPU address space range. This memory (or a part of memory) is not managed by the SOAR and so the resources inside are considered as external.

Most of 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.

6.12.3 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`.

6.12.4 Dependencies

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

6.12.5 Installation

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

6.12.6 Use

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

6.13 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.

6.13.1 ECOM

> **Warning:** This chapter describes the Foundation Library `ECOM-1.1`.
>
> `ECOM-1.1` is discontinued since *Architecture 8.0.0*.

Principle

The Embedded COMmunication Foundation Library (ECOM) is a generic communication library with abstract communication stream support (a communication framework only). It allows you to open and use streams on communication devices such as a COMM port.

This library also provides a device manager, including a generic device registry and a notification mechanism, which allows plug&play-based applications.

This library does not provide APIs to manipulate some specific options for each communication method, but it does provide some generic APIs which abstract the communication method. After the opening step, the MicroEJ Application can use every communications method (COMM, USB etc.) as generic communication in order to easily change the communication method if needed.

Functional Description

The diagram below shows the overall process to open a connection on a hardware device.

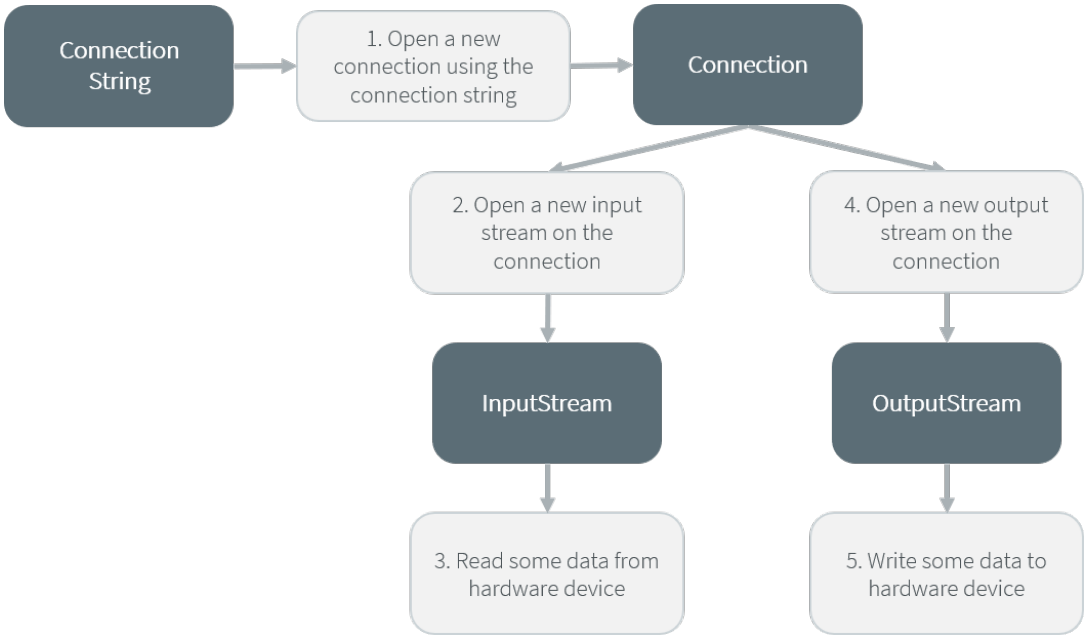


Fig. 35: ECOM Flow

1. Step 1 consists of opening a connection on a hardware device. The connection kind and its configuration are fixed by the String parameter of the method Connector.open(String).
2. Step 2 consists of opening an InputStream on the connection. This stream allows the MicroEJ Application to access the “RX” feature of the hardware device.
3. Step 3 consists of using the InputStream APIs to receive in the MicroEJ Application all hardware device data.
4. Step 4 consists of opening an OutputStream on the connection. This stream allows the MicroEJ Application to access the “TX” feature of the hardware device.
5. Step 5 consists of using the OutputStream APIs to transmit some data from the MicroEJ Application to the hardware device.

Note that steps 2 and 4 may be performed in parallel, and do not depend on each other.

Device Management API

A device is defined by implementing ej.ecom.Device. It is identified by a name and a descriptor (ej.ecom.HardwareDescriptor), which is composed of a set of MicroEJ properties. A device can be registered/unregistered in the ej.ecom.DeviceManager.

A device registration listener is defined by implementing ej.ecom.RegistrationListener. When a device is registered to or unregistered from the device manager, listeners registered for the device type are notified. The notification mechanism is done in a dedicated Java thread. The mechanism can be enabled or disabled (see *Standalone 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 to use 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.

6.13.2 ECOM Comm

Warning: This chapter describes the Foundation Library `ECOM-COMM-1.1`.

`ECOM-COMM-1.1` is deprecated in favor of `ECOM-COMM-2.0` and has been removed from *Architecture 8.0.0*. See *Migrate ECOM-COMM Module* for more details.

Principle

The ECOM Comm Java library provides support for serial communication. ECOM Comm extends ECOM to allow stream communication via serial communication ports (typically UARTs). In the MicroEJ Application, the connection is established using the Connector.open() method. The returned connection is a ej.ecom.io.CommConnection, and the input and output streams can be used for full duplex communication.

The use of ECOM Comm in a custom platform requires the implementation of an UART driver. There are two different modes of communication:

- In Buffered mode, ECOM Comm manages software FIFO buffers for transmission and reception of data. The driver copies data between the buffers and the UART device.
- In Custom mode, the buffering of characters is not managed by ECOM Comm. The driver has to manage its own buffers to make sure no data is lost in serial communications because of buffer overruns.

This ECOM Comm implementation also allows dynamic add or remove of a connection to the pool of available connections (typically hot-plug of a USB Comm port).

Functional Description

The ECOM Comm process respects the ECOM process. Please refer to the illustration *"ECOM flow"*.

Component Architecture

The ECOM Comm C module relies on a native driver to perform actual communication on the serial ports. Each port can be bound to a different driver implementation, but most of the time, it is possible to use the same implementation (i.e. same code) for multiple ports. Exceptions are the use of different hardware UART types, or the need for different behaviors.

Five C header files are provided:

- `LLCOMM_impl.h`

  Defines the set of functions that the driver must implement for the global ECOM comm stack, such as synchronization of accesses to the connections pool.

- `LLCOMM_BUFFERED_CONNECTION_impl.h`

  Defines the set of functions that the driver must implement to provide a Buffered connection

- `LLCOMM_BUFFERED_CONNECTION.h`

  Defines the set of functions provided by ECOM Comm that can be called by the driver (or other C code) when using a Buffered connection

- `LLCOMM_CUSTOM_CONNECTION_impl.h`

  Defines the set of functions that the driver must implement to provide a Custom connection

- `LLCOMM_CUSTOM_CONNECTION.h`

  Defines the set of functions provided by ECOM Comm that can be called by the driver (or other C code) when using a Custom connection

The ECOM Comm drivers are implemented using standard LLAPI features. The diagram below shows an example of the objects (both Java and C) that exist to support a Buffered connection.

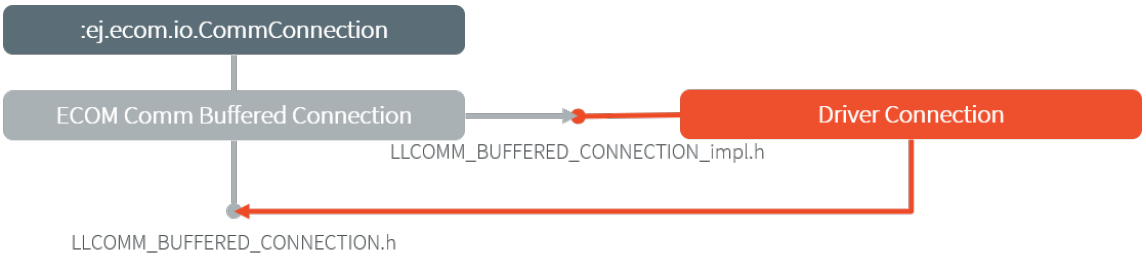


Fig. 36: ECOM Comm components

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 a hardware device[2].
- A platform port name. This is mostly used for dynamic connections or on platforms having a file-system based device mapping.

When the Comm Port is identified by a number, its string identifier is the concatenation of “com” and the number (e.g. com11).

Application Port Mapping

The mapping from application port numbers to platform ports is done in the application launch configuration. This way, the application can refer only to the application port number, and the data stream can be directed to the matching I/O port on different versions of the hardware.

Ultimately, the application port number is only visible to the application. The platform identifier will be sent to the driver.

Opening Sequence

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

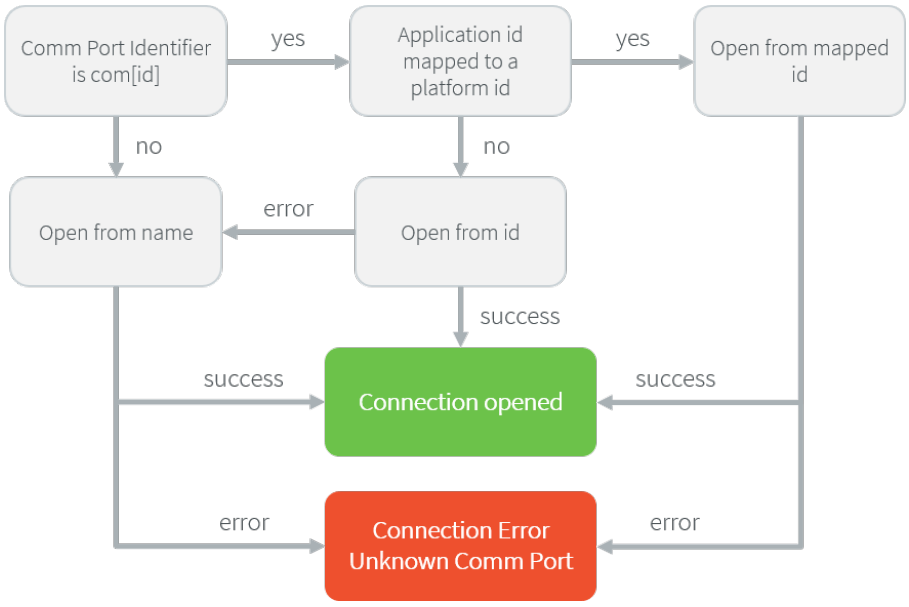


Fig. 37: Comm Port Open Sequence

[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 *Standalone Application Options*.

A removed connection is alive until it is closed by the application and, if enabled, unregistered from ECOM device manager. A connection is effectively uninstalled (and thus eligible to be reused) only when it is released by the stack.

The following sequence diagram shows the lifecycle of a dynamic connection with ECOM registration mechanism enabled.

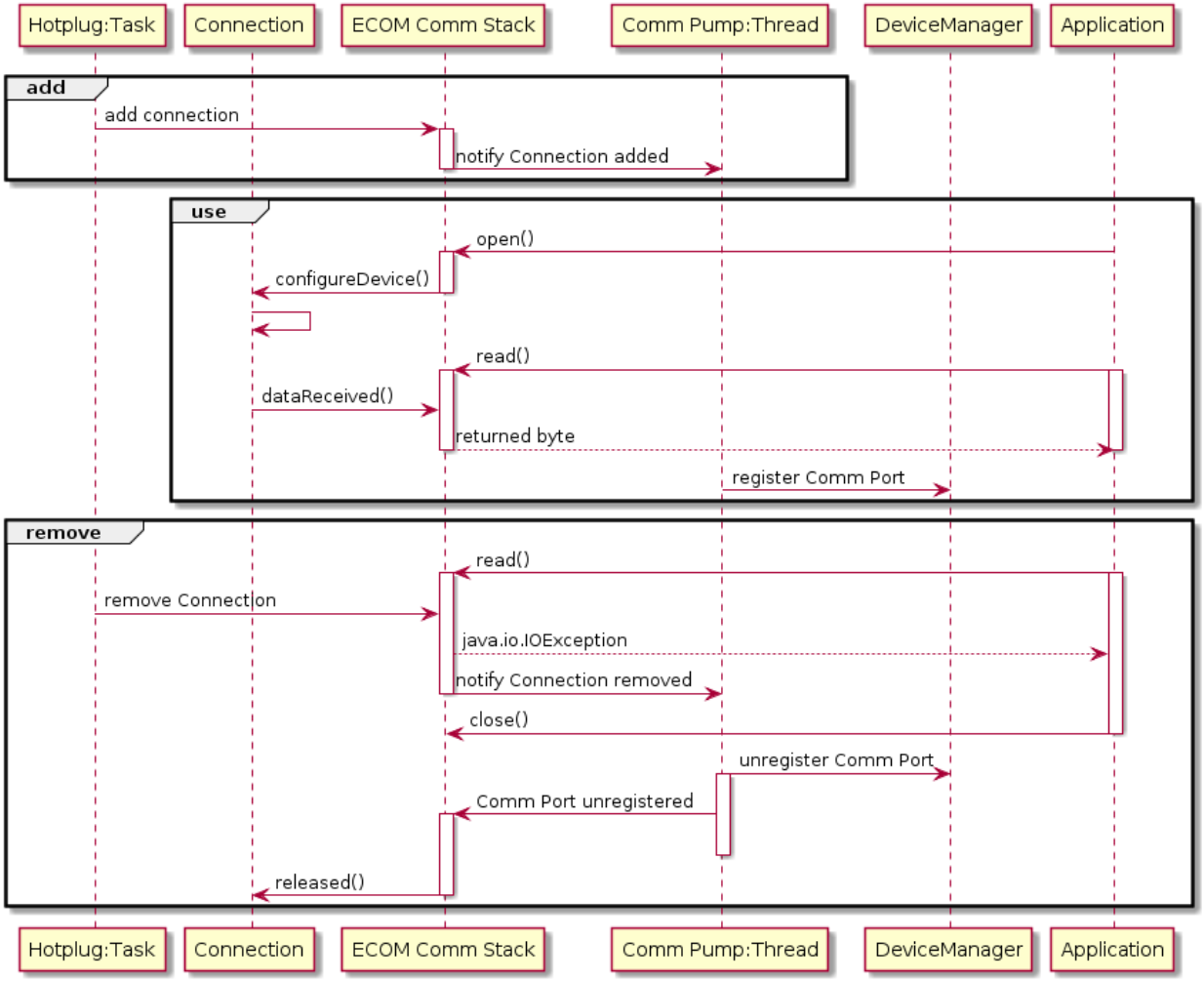


Fig. 38: Dynamic Connection Lifecycle

Java API

Opening a connection is done using ej.ecom.io.Connector.open(String url). The connection string (the `url` parameter) must start with "comm:", followed by the Comm port identifier, and a semicolon-separated list of options. Options are the baudrate, the parity, the number of bits per character, and the number of stop bits:

- baudrate=n (9600 by default)
- bitsperchar=n where n is in the range 5 to 9 (8 by default)
- stopbits=n where n is 1, 2, or 1.5 (1 by default)
- parity=x where x is odd, even or none (none by default)

All of these are optional. Illegal or unrecognized parameters cause an IllegalArgumentException.

Driver API

The ECOM Comm Low Level API is designed to allow multiple implementations (e.g. drivers that support different UART hardware) and connection instances (see Low Level API Pattern chapter). Each ECOM Comm driver defines a data structure that holds information about a connection, and functions take an instance of this data structure as the first parameter.

The name of the implementation must be set at the top of the driver C file, for example[3]:

```
#define LLCOMM_BUFFERED_CONNECTION MY_LLCOMM
```

This defines the name of this implementation of the `LLCOMM_BUFFERED_CONNECTION` interface to be `MY_LLCOMM`.

The data structure managed by the implementation must look like this:

```
typedef struct MY_LLCOMM{
    struct LLCOMM_BUFFERED_CONNECTION header;
    // extra data goes here
} MY_LLCOMM;

void MY_LLCOMM_new(MY_LLCOMM* env);
```

In this example the structure contains only the default data, in the header field. Note that the header must be the first field in the structure. The name of this structure must be the same as the implementation name (`MY_LLCOMM` in this example).

The driver must also declare the "new" function used to initialize connection instances. The name of this function must be the implementation name with `_new` appended, and it takes as its sole argument a pointer to an instance of the connection data structure, as shown above.

The driver needs to implement the functions specified in the `LLCOMM_impl.h` file and for each kind of connection, the `LLCOMM_BUFFERED_CONNECTION_impl.h` (or `LLCOMM_CUSTOM_CONNECTION_impl.h`) file.

The driver defines the connections it provides by adding connection objects using `LLCOMM_addConnection`. Connections can be added to the stack as soon as the `LLCOMM_initialize` function is called. Connections added during the call of the `LLCOMM_impl_initialize` function are static connections. A static connection is registered to the ECOM registry and cannot be removed. When a connection is dynamically added outside the 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 *Standalone Application Options*).

A XML file is so required to configure the Java platform. The name of this file must be `ecom-comm.xml` . It has to be stored in the module configuration folder (see *Installation*).

This file must start with the node `<ecom>` and the sub node `<comms>` . It can contain several time this kind of line: `<comm platformId="A_COMM_PORT_NUMBER" nickname="A_NICKNAME"/>` where:

- `A_COMM_PORT_NUMBER` refers the Comm port the Java platform user will be able to use (see *Application Port Mapping*).
- `A_NICKNAME` is optional. It allows to fix a printable name of the Comm port.

The `maxConnections` attribute indicates the maximum number of connections allowed, including static and dynamic connections. This attribute is optional. By default, it is the number of declared Comm Ports.

Example:

Listing 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 to use the ECOM Comm library.

```
<dependency org="ej.api" name="ecom-comm" rev="1.1.4"/>
```

This Foundation Library is always required when developing a MicroEJ Application which communicates with some external devices using the serial communication mode.

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

6.14 Graphical User Interface

Note: This chapter describes the current Graphical User Interface version `3`, provided by UI Pack version `13.0.0` or higher. The UI Pack *Changelog* and a *Migration Guide* are provided at the end of this chapter. If you are using the former Graphical User Interface version `2` (provided by MicroEJ UI Pack version up to `12.1.x`), please refer to this MicroEJ Documentation Archive.

6.14.1 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, Abstraction Layer APIs `LLUI_xxx`, etc.

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

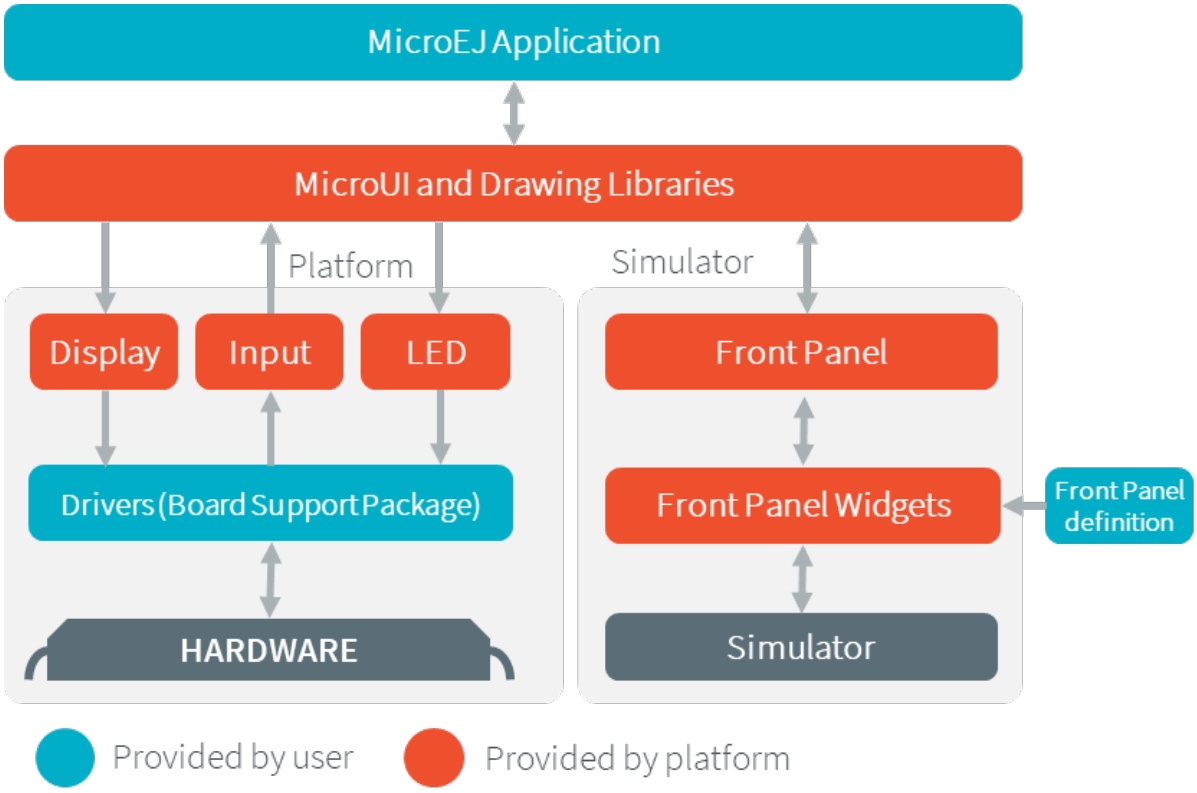


Fig. 39: Overview

The modules responsible to manage the Display, the Input and the LED are respectively called *Display module*, *Input module* and *LED module*. These three low-level parts connect MicroUI library to the user-supplied drivers code (coded in C). The drivers can use hardware accelerators like DMA and GPU to perform specific actions (buffers copy, drawings, etc.).

The MicroEJ Simulator provides all features of MicroUI library. The three modules are grouped together in a module called *Front Panel*. The Front Panel is supplied with a set of software widgets that generically support a range of input devices such as buttons, joysticks and touchscreens, and output devices such as displays and LEDs. With the help of the Front Panel Designer tool that forms part of the MicroEJ Workbench the user must define a Front Panel mock-up using these widgets.

The Display module also manages fonts and images. The fonts and images are pre-processed before compiling the application. The following diagram depicts the components involved in its design, along with the provided tools:

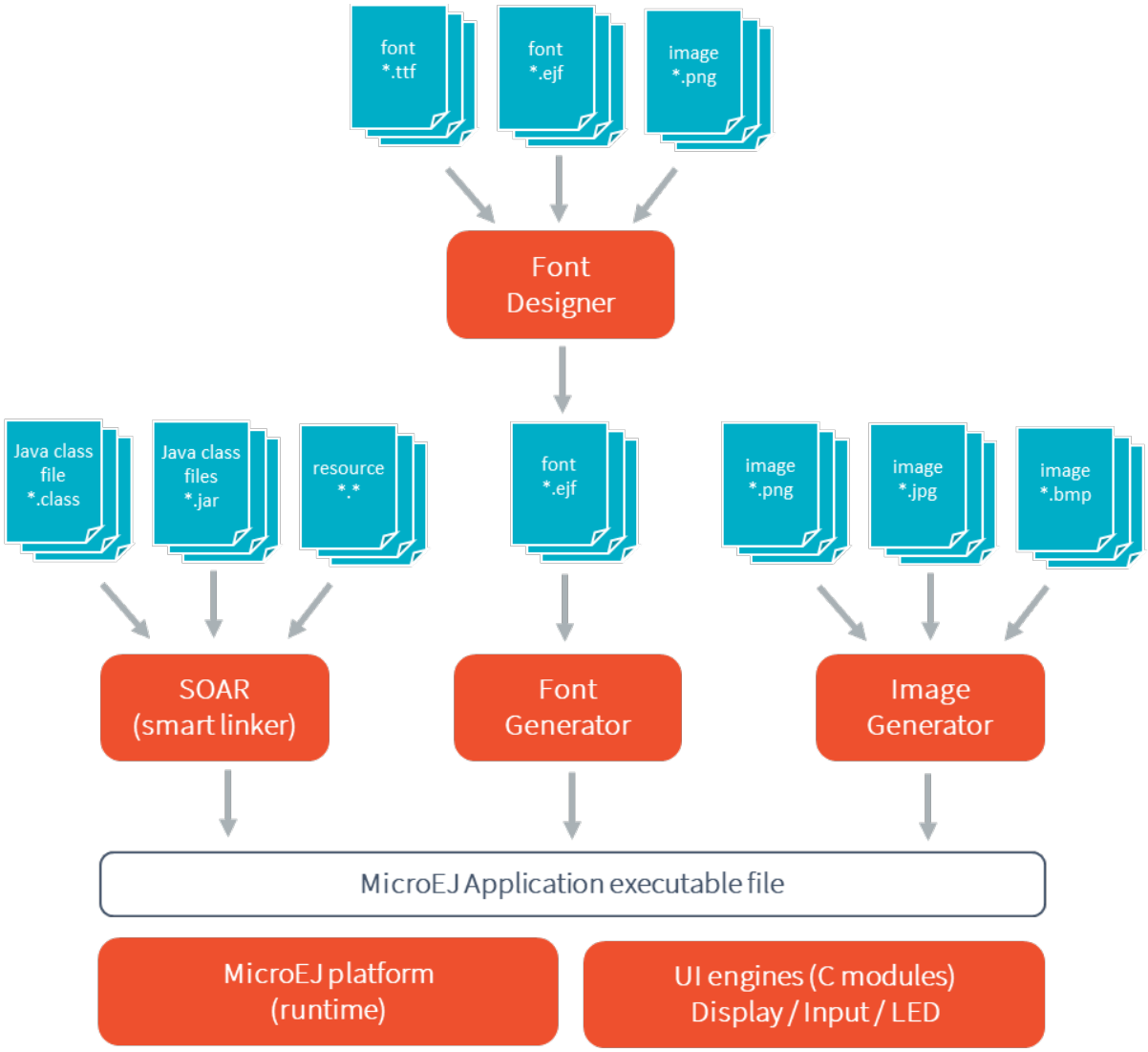


Fig. 40: The User Interface Extension Components along with a VEE Port

6.14.2 UI Port

This chapter summarizes all the steps to port the UI Pack: from the VEE Port Configuration project to more advanced features like using a GPU. This chapter only introduces the concepts and references the following chapters. The concepts are overviewed and incomplete (only the typical case is described).

It is recommended to follow the steps in this order:

1. Edit the VEE Port Configuration project to add the UI Pack dependency and configuration,
2. Create the Simulator extension project,
3. Port the minimal implementation of the BSP,
4. Extend the implementation by connecting a GPU.

UI Port Configuration

Principle

The first step is to update the *VEE Port Configuration project* (often named `xxx-configuration`): this project holds the *Module Description File* (`module.ivy`). This update is done in several steps, described in the sections below. Some steps are optional, depending on the capabilities of the hardware.

> **Warning:** This chapter assumes that a valid VEE Port has been created (as described in the chapter *Platform Creation*).

UI Pack Selection

The UI Pack bundles several modules, including the Graphics Engine. The Graphics Engine is a library already compiled for an MCU and a C compiler. The *MicroEJ Central Repository* provides UI Packs for a set of MCU/Compiler pairs (like for MicroEJ Architectures).

Refer to the chapter *Pack Import* to add the required UI Pack. As an example, the module dependency to add for a Cortex-M4 and GCC toolchain would be:

```
<dependencies>
    <!-- MicroEJ Architecture Specific Pack -->
    <dependency org="com.microej.architecture.CM4.CM4hardfp_GCC48" name="flopi4G25-ui-pack"␣
↪rev="13.5.1"/>
</dependencies>
```

UI Pack Modules

The following sections describe each module that comes with the UI Pack (purpose and configuration).

The modules provided by the UI Pack are **not installed** by default. When a module is required, it has to be enabled and configured using the VEE Port Editor.

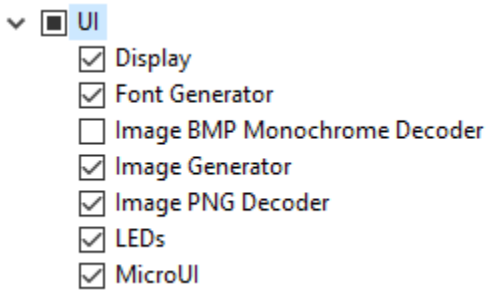


Fig. 41: UI Pack Modules

Refer to the chapter *Platform Module Configuration* to add the UI Pack modules.

Module MicroUI

MicroUI is a Foundation Library that defines a Low Level UI framework (refer to the chapter *MicroUI* for more information). The **mandatory** module MicroUI (it must be checked in the VEE Port configuration file) provides the MicroUI implementation library. It requires a static initialization step to specify what MicroUI features are available for the application layer:

1. Create the file `[VEE Port Configuration project]/microui/microui.xml`
2. Edit the file as described here: *Static Initialization*.

```
<microui>

    <display name="DISPLAY"/>

    <eventgenerators>
        <command name="COMMANDS"/>
        <buttons name="BUTTONS" extended="3"/>
        <buttons name="JOYSTICK" extended="5"/>
        <touch name="TOUCH" display="DISPLAY"/>
    </eventgenerators>

    <fonts>
        <font file="resources\fonts\myfont.ejf"/>
    </fonts>

</microui>
```

Module LEDs

MicroUI provides some API to manipulate the LEDs. This module allows the UI Port to drive the LEDs. Refer to the chapter *LED* to have more information.

This module is optional: when not selected, a stub implementation is used, and the UI Port does not need to provide one.

Modules Image Decoders

Note: This chapter only applies when the device has a display.

This module adds an internal image decoder: it allows the application to embed an encoded image (e.g., PNG or BMP Monochrom) and let the Graphics Engine decode it at runtime. Both decoders (PNG and BMP Monochrom) are optional and can be selected (or not) independently. Refer to the chapter *Encoded Image* to have more information.

This module is optional: when no image decoder is embedded, the Graphics Engine relies on the UI Port (thanks to Abstraction Layer API) to decode the images.

Module Image Generator

Note: This chapter only applies when the device has a display.

This module allows decoding the application's images at compile-time. The application's images are decoded and stored in a binary format compatible with the Graphics Engine. The memory footprint of the application is higher, but the image loading time at runtime is very low. Refer to the chapter *Image Generator* to have more information.

This module is optional: when not selected, the application cannot embed generated images compatible with the Graphics Engine.

Module Font Generator

Note: This chapter only applies when the device has a display.

This module allows for embedding the MicroEJ bitmap fonts of the application. The application's fonts (EJF files) are decoded and stored in a binary format compatible with the Graphics Engine. Refer to the chapter *Font Generator* to have more information.

This module is optional: when not selected, the application cannot embed fonts compatible with the Graphics Engine.

Module Display

Note: This chapter only applies when the device has a display.

This chapter takes the concepts described in chapter *Display*. The first step is determining the kind of display: size, pixel format, and constraints. This information will be used later by the UI Port configuration project, the Simulator extension project, and the BSP.

Size

The size is expressed in *pixels*, often 320x240 or 480x272. This size defines the area the application can target; it can retrieve this size by calling `Display.getWidth()` and `Display.getHeight()`. It is always a rectangular area, even for the rounded displays (a square area frames a rounded display).

The display size is fixed for a display: retrieve this size in the board's datasheet.

Pixel Format

The display pixel format (or pixel structure) gives two notions: the number of bits-per-pixel and the organization of color components in these bits.

The number of bits-per-pixel (bpp) is an integer value among this list: 1, 2, 4, 8, 16, 24, or 32.

The color components organization defines how the color components (Red, Green, and Blue) are distributed in the pixel. The greater the display pixel format (in bits), the better is the definition. This format also indicates the number of bits-per-pixel. For instance, the format RGB565 is a 16-BPP format, indicating that the five MSB bits are for the Red color component, the six next bits are for the Green component, and the five LSB bits are for the Blue component. This pixel format can be symbolized by `RRRRRGGGGGGBBBBB` or `RRRR RGGG GGGB BBBB`.

The display pixel format is often fixed by the display itself (its capabilities) and by the memory bus between the MCU and the LCD module. However, the display pixel format is often configurable by the LCD controller. Note that the number of bits-per-pixel and the display size fix the required memory to allocate: `memory_size = width x height x bpp / 8`. Consequently, the pixel format may be less precise than the display capabilities depending on the memory available on the device. For instance, the RGB565 format may be used whereas the display is a 24-bit display (RGB888).

Constraints

The hardware constraints (display, bus, memory, etc.) may drive the configuration:

- The pixel format: Some hardware cannot use another pixel format other than the one of the display. This format may be standard or custom. See *Pixel Structure*.
- The size of the buffers: The available memory may be limited. This limitation can drive the chosen pixel format.
- Memory alignment: Some LCD controllers require a memory alignment on the display buffer (alignment on 64 bits, for instance).
- Buffer width alignment: Some LCD controllers also require an alignment for each line. The line size (in pixels) in memory may be larger than the display line size (width): this is the stride. The alignment constraint may be expressed in pixels or bytes. The required memory to allocate becomes: `memory_size = stride (in pixels) x height x bpp / 8`.

Configuration

In the VEE Port Configuration project, create and fill the file `display.properties`:

1. Create the file `[VEE Port Configuration project]/display/display.properties`
2. Fill the file as described here: *Installation*, according to the pixel format and the display constraints.

```
bpp=rgb565
imageBuffer.memoryAlignment=32
memoryLayout=line
byteLayout=line
```

VEE Port Build

Once modules are selected and configured, the VEE Port can be built again; see *Platform Build*.

Simulation

Principle

The simulation part of the UI port requires the creation (or extension) of a Front Panel project which is compatible with the UI Pack.

First, if no Front Panel project exists, follow the steps described here: *Front Panel Mock*. Then, follow the next chapters to extend the Front Panel project with UI Pack notions.

Project Extension

The Front Panel project must depend on the UI Pack. Add the following dependency to the Front Panel ivy file:

```
<dependency org="com.microej.pack.ui" name="ui-pack" rev="[UI Pack version]">
   <artifact name="frontpanel" type="jar"/>
</dependency>
```

See *Simulation* for more information about the Front Panel project dependencies.

LEDs

When the VEE Port Configuration project LEDs module is checked, the Front Panel project should add a widget `LED` for each led.

1. With an image editor, create an image for the LED off and an image for the LED on. Both images must have the same size.
2. Create a couple of images for each LED.
3. In the Front Panel description file, add this line for each LED:

```
<ej.fp.widget.LED label="0" x="170" y="753" ledOff="Led-0.png" ledOn="Led-GREEN.png" overlay=
↪"false"/>
```

The label must have an integer value from 0 to `NUMBER_OF_LEDS - 1`. The `ej.microui.led.Leds` class uses this value as the LED identifier in `setLedOff(int ledId)`, `setLedOn(int ledId)`, and other methods of the class.

Buttons

The widget `Button` can simulate any hardware button.

1. With an image editor, create an image for the button released and an image for the button pressed. Both images must have the same size.
2. Create a couple of images for each button.
3. In the Front Panel description file, add this line for each button:

```
<ej.fp.widget.Button label="0" x="316" y="769" skin="W-U-0.png" pushedSkin="W-U-1.png"/>
```

The label must have an integer value from 0 to `NUMBER_OF_BUTTONS - 1`. The label is used by the application to listen to the button.

By default, the widget sends a MicroUI Button event to the Buttons Event Generator whose name is `BUTTONS` and whose identifier is the button's label. To target another Buttons Event Generator, refer to the chapter *Inputs Extensions*.

Widget Button Code

```
public static class ButtonListenerToButtonEvents implements ButtonListener {

@Override
public void press(Button widget) {
   EventButton.sendPressedEvent(getMicroUIGeneratorTag(), widget.getID());
}

@Override
public void release(Button widget) {
   EventButton.sendReleasedEvent(getMicroUIGeneratorTag(), widget.getID());
}

/**
 * Gets the MicroUI Buttons events generator tag. This generator has to match the generator␣
↪set during the
 * VEE Port build in <code>microui/microui.xml</code>
 *
 * @return a MicroUI Buttons events generator tag
 */
protected String getMicroUIGeneratorTag() {
   return EventButton.COMMON_MICROUI_GENERATOR_TAG;
}
```

Application Code

To listen to the button, two ways are possible:

- By default, the current `Displayable` receives all events. The subclass has to implement the method `boolean handleEvent(int event);`.
- A class must extend the interface `EventHandler`, and this class must be set as the handler of the event generators Buttons:

```
Buttons[] buttonsHandlers = (Buttons[]) EventGenerator.get(Buttons.class);
for (EventGenerator buttonsHandler : generators) {
   buttonsHandler.setEventHandler(this);
}
```

Here is an example of a handler:

```
@Override
public boolean handleEvent(int event) {

   // get the event's data
   int data = Event.getData(event);

   String state = null;

   // print its state(s)
   if (Buttons.isPressed(data)) {
      state = "pressed ";
   }
   if (Buttons.isReleased(data)) {
      state = "released ";
   }
   if (Buttons.isRepeated(data)) {
      state = "repeated ";
   }
   if (Buttons.isLong(data)) {
      state = "long ";
   }
   if (Buttons.isClicked(data)) {
      state = "clicked ";
   }
   if (Buttons.isDoubleClicked(data)) {
      state = "double-clicked ";
   }

   if (state != null) {
      System.out.print("button\t\t");

      // get the button's id
      int id = Buttons.getButtonId(data);
      System.out.print(id+" ");
      System.out.println(state);
   }

   return true;
}
```

Button to Command Event

A recommended approach is to favor Command events over Buttons events. MicroUI Command events are more generic because not they are not tied to a hardware component like a physical button. Command events make the application code more flexible to hardware changes. For instance, instead of reacting to Button event 0, the application will respond to Command event `Enter` or `Up`. The application does not care about the source of the Command event: it may be the button 0, 1, 10, or any other input device.

To map a MicroUI Command on the widget Button:

1. Update the widget description by adding a `listenerClass`.

```
<ej.fp.widget.Button label="0" x="316" y="769" skin="W-U-0.png" pushedSkin="W-U-1.png"␣
↪listenerClass="com.is2t.microej.fp.Button2Command"/>
```

2. In the Front Panel project, create the class `com.is2t.microej.fp.Button2Command`, for instance:

```
public class Button2Command implements ej.fp.widget.Button.ButtonListener {

   public int getCommand(int buttonId) {
      // same command as EmbJPF (see buttons_listener.c)
      switch (buttonId) {
      default:
      case 0:
         return EventCommand.ESC;
      case 1:
         return EventCommand.MENU;
      }
   }

   @Override
   public void press(Button widget) {
      EventCommand.sendEvent(getCommand(widget.getID()));
   }

   @Override
   public void release(Button widget) {
      // nothing to send
   }
}
```

The application code becomes:

```
// [...]

   Command[] commandHandlers = (Command[]) EventGenerator.get(Command.class);
   for (EventGenerator commandHandler : generators) {
      commandHandler.setEventHandler(this);
   }

// [...]

@Override
public boolean handleEvent(int event) {

   // get the event's data
   int data = Event.getData(event);

   switch (data) {
   case Command.ESC:
      System.out.println("ESC");
      break;
   case Command.BACK:
      System.out.println("BACK");
      break;
```

(continues on next page)

(continued from previous page)

```
   // [...]
}
```

Touch Panel

Contrary to the other input devices, no image is required because a touch panel covers the display area.

1. Retrieve the display size in pixels.
2. In the Front Panel description file, add this line:

```
<ej.fp.widget.Pointer x="185" y="395" width="480" height="272" touch="true"/>
```

By default, the widget sends a MicroUI Pointer event to the Pointer Event Generator, whose name is `TOUCH` (a touch panel is considered a Pointer with only dragged events). To target another Pointer Event Generator, refer to the chapter *Inputs Extensions*.

A snippet of application code that handles Pointer events:

```
// [...]

   Pointer[] pointerHandlers = (Pointer[]) EventGenerator.get(Pointer.class);
   for (EventGenerator pointerHandler : generators) {
      pointerHandler.setEventHandler(this);
   }

// [...]

@Override
public boolean handleEvent(int event) {
   Pointer pointer = (Pointer) Event.getGenerator(event);
   int x = pointer.getX();
   int y = pointer.getY();
   System.out.println("(" + x + "," + y + ")");
}
```

Display

The widget Display features a lot of options to simulate the hardware specificities.

1. Retrieve the display size in pixels.
2. In the Front Panel description file, add this line:

```
<ej.fp.widget.Display x="185" y="395" width="480" height="272"/>
```

For more information, refer to the java-doc of the widget Display and the chapter *Widget Display*.

Build

Once the Front Panel project is created or modified, the VEE Port must be built again (the front panel is built simultaneously with the VEE Port; see *Platform Build*).

BSP Port

Principle

The BSP Port (or Embedded Port) involves implementing some Abstraction Layer APIs (low-level APIs: LLAPI). There are several kinds of LLAPI:

- The mandatory LLAPI to manipulate the LEDs,
- The mandatory LLAPI to send the input events,
- The mandatory LLAPI to initialize, use and flush the drawings to the display,
- The optional LLAPI to customize the Graphics Engine to be compatible with the display constraints,
- The optional LLAPI to manipulate the optional display features (backlight, contrast, etc.),
- The optional LLAPI to add some features as new image decoders,
- The optional LLAPI to use a GPU.

The following chapters describe each group of Abstraction Layer APIs, except the GPU acceleration (see the dedicated section *GPU Port*).

MicroUI C Module

The UI Pack **requires** the *MicroUI C module*. This C module

- implements some MicroUI native functions,
- manages the drawings synchronization with the Graphics Engine,
- features an image heap allocator,
- features an input events decoder.

Before all, install the MicroUI C Module:

1. Find the correct version of the C module according to the UI Pack version; see *C Modules*.
2. Unzip it in the BSP project.
3. Add the mandatory files to the list of the BSP project's compiled files: `LLDW_PAINTER_impl.c`, `LLUI_PAINTER_impl.c`, `ui_drawing_stub.c`, `ui_drawing.c` and `ui_image_drawing.c`.
4. Add the optional files in the BSP project (if their associated feature is used/needed):
   - `LLUI_DISPLAY_HEAP_impl.c`: to use another image heap allocator,
   - `LLUI_INPUT_LOG_impl.c` and `microui_event_decoder.c`: to decode the MicroUI event (input events and MicroUI internal events).
5. Add the C Module's include folder to the BSP project's include directories list.

LEDs

As soon as the VEE Port Configuration project LEDs module is checked, the VEE Port features the header file LLAPI `LLUI_LED_impl.h`. This header must be implemented. The mandatory functions to implement are:

- `LLUI_LED_IMPL_initialize`: initialize the LED driver (if required) and return the available number of LEDs.
- `LLUI_LED_IMPL_getIntensity`: return, if possible, the LED intensity.
- `LLUI_LED_IMPL_setIntensity`: set the LED intensity.

Refer to *Abstraction Layer API* to have more information. Refer too to the C-doc in the header file itself.

Inputs

The VEE Port always features the header file LLAPI `LLUI_INPUT_impl.h`. This header must be implemented even if there is no input device: the critical section management is required by the MicroUI library itself. The mandatory functions to implement are:

- `LLUI_INPUT_IMPL_initialize`: can be empty if nothing is to initialize.
- `LLUI_INPUT_IMPL_getInitialStateValue`: empty if there is no State Event Generator.
- `LLUI_INPUT_IMPL_enterCriticalSection`: disable all input events (disable input devices interrupts and/or disable the OS scheduling).
- `LLUI_INPUT_IMPL_leaveCriticalSection`: re-enable all inputs events.

Refer to *Abstraction Layer API* to have more information. Refer too to the C-doc in the header file itself.

Display

As soon as the VEE Port Configuration project Display module is checked, the VEE Port features the header file LLAPI `LLUI_DISPLAY_impl.h`. This header must be implemented. The mandatory functions to implement are:

- `LLUI_DISPLAY_IMPL_initialize`: fill the given structure `LLUI_DISPLAY_SInitData` (display size, buffer address, etc.).
- `LLUI_DISPLAY_IMPL_binarySemaphoreTake`: *takes* the given semaphore.
- `LLUI_DISPLAY_IMPL_binarySemaphoreGive`: *gives* the given semaphore.
- `LLUI_DISPLAY_IMPL_flush`: copy/send the buffer content to the LCD.

Refer to *Abstraction Layer API* to have more information. Refer too to the C-doc in the header file itself.

Display: LCD Constraints

According to the LCD constraints (see *UI Port Configuration*), some additional LLAPI must be implemented:

- `LLUI_DISPLAY_IMPL_convertARGBColorToDisplayColor` and `LLUI_DISPLAY_IMPL_convertDisplayColorToARGBColor`: required when the pixel format is custom (not standard, see *Dependencies*).
- `LLUI_DISPLAY_IMPL_prepareBlendingOfIndexedColors`: required when the display buffer is a LUT buffer, not a pixel buffer.
- `LLUI_DISPLAY_IMPL_isDoubleBuffered`: the default implementation returns always `true`; only useful as information for the application.

- `LLUI_DISPLAY_IMPL_isColor` : the default implementation always returns `true` when the BPP is higher than 8; only useful as information for the application.
- `LLUI_DISPLAY_IMPL_getNumberOfColors` : the default implementation returns always `1 << BPP` ; only useful as information for the application.

Display: Optional Features

Several kinds of features can be implemented.

Hardware features:

- `LLUI_DISPLAY_IMPL_setContrast` and `LLUI_DISPLAY_IMPL_getContrast` : to configure the display contrast.
- `LLUI_DISPLAY_IMPL_hasBacklight` , `LLUI_DISPLAY_IMPL_setBacklight` and `LLUI_DISPLAY_IMPL_getBacklight` : to turn on or off the display backlight.

Runtime Image Decoders

The BSP can add some runtime image decoders with the runtime decoders selected in the VEE Port configuration project (modules PNG and BMP Monochrom decoders).

- `LLUI_DISPLAY_IMPL_decodeImage` : called by MicroUI to decode an image whose format is unknown by the internal runtime image decoders.

Image Heap Management

By default, a best-fit allocator manages the image heap. To add another allocator, implement these functions:

- `LLUI_DISPLAY_IMPL_image_heap_initialize` : initialize the allocator.
- `LLUI_DISPLAY_IMPL_image_heap_allocate` : allocates the expected buffer.
- `LLUI_DISPLAY_IMPL_image_heap_free` : frees the given buffer.

MicroUI Image Management

These three functions are only helpful for compatibility with a GPU; see *GPU Port*.

- `LLUI_DISPLAY_IMPL_getNewImageStrideInBytes`
- `LLUI_DISPLAY_IMPL_adjustNewImageCharacteristics`
- `LLUI_DISPLAY_IMPL_initializeNewImage`

Test Suite

The Port Qualification Toolkit (PQT) provides a UI test suite to validate the UI Port (see *VEE Port Test Suite* to have more information). This test suite **must** be executed to validate the UI Port and after each modification on this UI Port (for instance, after changes to improve performances).

The UI Port test suite is available here: https://github.com/MicroEJ/VEEPortQualificationTools/tree/master/tests/ui/ui3.

The test suite is constituted of two blocks:

- The minimal *Display* test suite: a simple application test (with source code) to validate the mandatory functions to implement to target a Display.
- An extended *Display* test suite: a library that tests several MicroUI drawings. This test suite only applies when the BSP uses a GPU to perform the drawings. See *GPU Port*.

The test suite does not check all UI Port features. However, some example projects are available in MicroEJ GitHub:

- LED: refer to the application https://github.com/MicroEJ/Example-Standalone-Foundation-Libraries/tree/master/microui.led.
- Input: refer to the application https://github.com/MicroEJ/Example-Standalone-Foundation-Libraries/tree/master/microui.input

Some other example projects are also available in MicroEJ GitHub and can be used to check if the UI Port is valid:

- Hello World: https://github.com/MicroEJ/Example-Standalone-Foundation-Libraries/tree/master/microui.helloworld
- Use of images: https://github.com/MicroEJ/Example-Standalone-Foundation-Libraries/tree/master/microui.image

GPU Port

Principle

MicroUI and *MicroUI C module* are designed to be extended using a GPU hardware drawing acceleration. This acceleration is optional and should be performed after the mandatory operations (see *BSP Port*).

A GPU can be used to draw shapes and/or images. Most of the time, the minimal implementation consists of filling the rectangles and drawing the images. The MicroUI C module is designed to let the BSP implement only the GPU features. When a drawing is not implemented over a GPU, the software implementation is automatically used instead. No extra code should be added to the BSP to use the software algorithms.

The main advantages of using a GPU are:

- the drawing is rendered faster than using the software algorithms,
- the drawing is performed asynchronously, allowing the MCU to perform other actions (no need to wait until the end of the drawing).

Existing C Modules

Some *C Modules* are available on the MicroEJ Repository. These C modules already implement compatible features with a GPU. Add the mandatory files to the list of the BSP project's compiled files to use the associated GPU (and add the C Module's include folder to the BSP project's include directories list). Refer to *C Modules* to have more information.

Port a GPU

Drawing Function

As described in *Painter Abstraction Layer API*, the idea of the implementation of `LLUI_PAINTER_impl.h` (and `LLDW_PAINTER_impl.h`) is first to manage the synchronization with the Graphics Engine and then, to dispatch the drawing itself to a third party implementation through the header file `ui_drawing.h` . The files `LLUI_PAINTER_impl.c` and `LLDW_PAINTER_impl.c` available in the MicroUI C module already perform this operation for all MicroUI drawings. Consequently, only the drawing itself should be implemented in the BSP.

For example:

```
DRAWING_Status UI_DRAWING_fillRectangle(MICROUI_GraphicsContext* gc, jint x1, jint y1, jint␣
↪x2, jint y2) {
   // TODO
}
```

The drawing function has to take into account these properties:

- the color: the structure `MICROUI_GraphicsContext` gives the shape color (always fully opaque),
- the clip: the `LLUI_DISPLAY.h` file provides some functions to retrieve the current `MICROUI_GraphicsContext`'s clip,
- the buffer destination address by calling the `LLUI_DISPLAY_getBufferAddress` function,
- the shape bounds: the drawing function parameters.

The drawing function must return the drawing status. This status indicates to the Graphics Engine the kind of drawing:

- synchronous drawing: the drawing is performed by the GPU and entirely performed before returning. In that case, the drawing function has to return `DRAWING_DONE`.
- asynchronous drawing: the drawing is started, maybe processed by the GPU before returning. In that case, the drawing function has to return `DRAWING_RUNNING`.

As explained above, the second case should be the rule. That means that the Graphics Engine cannot ask for another drawing (accelerated or not) before the end of the drawing currently performed by the GPU. To unlock the Graphics Engine, the GPU interrupt routine must call the Graphics Engine function `LLUI_DISPLAY_notifyAsynchronousDrawingEnd` to notify the end of the drawing. The Graphics Engine manages the synchronization alone; no extra support in the BSP is mandatory.

Note: The end of the asynchronous drawing may occur before the end of the drawing function execution (before returning). The Graphics Engine also manages this use case, and the BSP implementation does not need to check this use case.

Fallback

A GPU may not be able to manage all the drawing functions. For instance, it cannot manage all image formats, or it cannot manage all rotation angles, etc. In that case, the drawing function can call the software drawing function instead.

```
DRAWING_Status UI_DRAWING_fillRectangle(MICROUI_GraphicsContext* gc, jint x1, jint y1, jint␣
↪x2, jint y2) {
   DRAWING_Status ret;
   if (!compatible_drawing(gc, x1, y1, x2, y2)) {
      UI_DRAWING_SOFT_fillRectangle(gc, x1, y1, x2, y2);
      ret = DRAWING_DONE;
   }
   else {
      gpu_fill_rect(LLUI_DISPLAY_getBufferAddress(&gc->image), x1, y1, x2, y2);
      ret = DRAWING_RUNNING;
   }
   return ret;
}
```

Image Constraints

The GPU may have strong requirements on the images:

- the pixels buffer start address alignment,
- an image stride different than the image width.

These constraints affect the compile-time images (Image Generator) and the runtime images (decoded images and MicroUI BufferedImage).

Address Alignment

In the VEE Port Configuration project, specify the property `imageBuffer.memoryAlignment` in the `display.properties` file. The value is the alignment in bits. This value will be taken into account by the compile-time images (Image Generator) and the runtime images.

Note: For the runtime images, this alignment value may be customized thanks to the function `LLUI_DISPLAY_IMPL_adjustNewImageCharacteristics`.

Stride (Compile-time Images)

The stride is dynamic, often depending on the image format and width. Consequently, the stride cannot be set as a property in the `display.properties` file for example.

For the compile-time images (Image Generator), a specific extension of the ImageGenerator is required.

1. See *Extended Mode* to create the ImageGenerator extension project.
2. Create a class that implements `BufferedImageLoader`. The value to be returned is expressed in pixels.

```
public class MicroUIGeneratorExtension extends BufferedImageLoader{

   private static final int ALIGNMENT_PIXELS = 16;

   @Override
   public int getStride(int defaultStride) {
      return (getWidth() + ALIGNMENT_PIXELS - 1) & ~(ALIGNMENT_PIXELS - 1);
   }
}
```

3. Create the file `/META-INF/services/com.microej.tool.ui.generator.MicroUIRawImageGeneratorExtension`
4. Fill it with the class name:

```
my.package.MicroUIGeneratorExtension
```

5. Build the project and copy the result in the VEE Port Configuration project, subfolder `dropins/tools`.
6. Rebuild the VEE Port.

Stride (Runtime Images)

For the compile-time images, the BSP has to implement the LLAPI `LLUI_DISPLAY_IMPL_getNewImageStrideInBytes` (the value to be returned is expressed in bytes):

```
uint32_t UI_DRAWING_getNewImageStrideInBytes(jbyte image_format, uint32_t image_width,␣
↪uint32_t image_height, uint32_t default_stride) {
     uint32_t bpp = DISPLAY_UTILS_get_bpp((MICROUI_ImageFormat)image_format);
     return (bpp >= (uint32_t)8) ? ALIGN(image_width, (uint32_t)16) * (bpp / (uint32_t)8) :␣
↪ALIGN(image_width, (uint32_t)8);
}
```

Test Suite

As described *here*, the Port Qualification Toolkit (PQT) provides a UI test suite to validate the UI Port. The second block of the UI test suite (extended *Display* test suite) uses a library that tests several MicroUI drawings. This test suite **must** be executed to validate the UI Port over a GPU and after each modification on this UI Port (for instance, after changes to improve performances).

6.14.3 MicroUI

Principle

MicroUI library defines a Low Level UI framework for embedded devices. This module allows the creation of basic Human-Machine-Interfaces (HMI), with output on a pixel-based screen.

Architecture

MicroUI library is the entry point to perform some drawings on a display and to interact with user input events. This library contains only a minimal set of basic APIs. High-level libraries can be used to have more expressive power, such as *MWT (Micro Widget Toolkit)*. In addition to this restricted set of APIs, the MicroUI implementation has been designed so that the EDC and BON footprint is minimal.

At application startup all MicroUI objects relative to the I/O devices are created and accessible. The following MicroUI methods allow you to access these objects:

- Display.getDisplay() : returns the instance of the display which drives the main display screen.
- Leds.getNumberOfLeds(): returns the numbers of available LEDs.

MicroUI is not a standalone library. It requires a configuration step and several extensions to drive I/O devices (display, inputs, LEDs).

First, MicroUI requires a configuration step in order to create these internal objects before the call to the `main()` method. The chapter *Static Initialization* explains how to perform the configuration step.

Note: This configuration step is the same for both embedded and simulated VEE Ports.

The embedded VEE Port requires some additional C libraries to drive the I/O devices. Each C library is dedicated to a specific kind of I/O device. A specific chapter is available to explain each kind of I/O device.

Table 12: MicroUI C libraries

I/O devices	Extension Name	Chapter
Graphical / pixel-based display	Display	*Display*
Inputs (buttons, joystick, touch, pointers, etc.)	Input	*Input*
LEDs	LED	*LED*

The simulation VEE Port uses a mock which simulates all I/O devices. Refer to the chapter *Simulation*.

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 *Standalone Application Options*).

Exceptions

The thread cannot be stopped with a Java exception: the exceptions are always checked by the framework.

When an exception occurs in a user method called by the internal thread (for instance render()), the current `UncaughtExceptionHandler` receives the exception. When no exception handler is set, a default handler prints the stack trace.

Native Calls

The MicroUI implementation for MicroEJ uses native methods to perform some actions (read input devices events, perform drawings, turn on LEDs, etc.). The library implementation has been designed to not use blocking native methods (wait input devices, wait end of drawing, etc.) which can lock the full 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:

- α*FG* is the alpha level of the foreground pixel (layer pixel),
- α*BG* is the alpha level of the background pixel (working buffer pixel),
- *Cxx* is a color component of a pixel (Red, Green or Blue),
- α*Out* 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 VEE Port configuration file, check UI > MicroUI to install the library. When checked, the XML file `microui/microui.xml` is required during VEE Port 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.

6.14.4 Static Initialization

Principle

The MicroUI implementation for MicroEJ requires a configuration step (also called extension step) to customize itself before application startup (see *Architecture*). This configuration step uses an XML file. In order to save both runtime execution time and flash memory, the file is processed by the Static MicroUI Initializer tool, avoiding the need to process the XML configuration file at runtime. The tool generates appropriate initialized objects directly within the MicroUI library, as well as Java and C constants files for sharing MicroUI event generator IDs.

This XML file (also called the initialization file) defines:

- The MicroUI event generators that will exist in the application in relation to low-level drivers that provide data to these event generators (see *Input*).
- Whether the application has a display; and if so, it provides its logical name.
- Which fonts will be provided to the application.

The next chapters describe succinctly the XML file. For more information about grammar, please consult appendix *MicroUI Static Initializer*.

Functional Description

The Static MicroUI Initializer tool takes as entry point the initialization file which describes the MicroUI library extension. This tool is automatically launched during the VEE Port build (see *Installation*).

The Static MicroUI Initializer tool is able to generate two files:

- A Java library which extends MicroUI library. This library is automatically added to the *MicroEJ Application classpath* when MicroUI API library is fetched. This library is used at MicroUI startup to create all instances of I/O devices (Display, EventGenerator, etc.) and contains the fonts described into the configuration file (these fonts are also called “system fonts”).

Warning: This MicroUI extension library is always generated and MicroUI library cannot run without this extension.

- A C header file (`*.h`). This header file contains some IDs which are used to make a link between an input device (buttons, touch) and its MicroUI event generator (see *Input*).

Note: The Front Panel project does not need a configuration file (like C header file for embedded VEE Port).

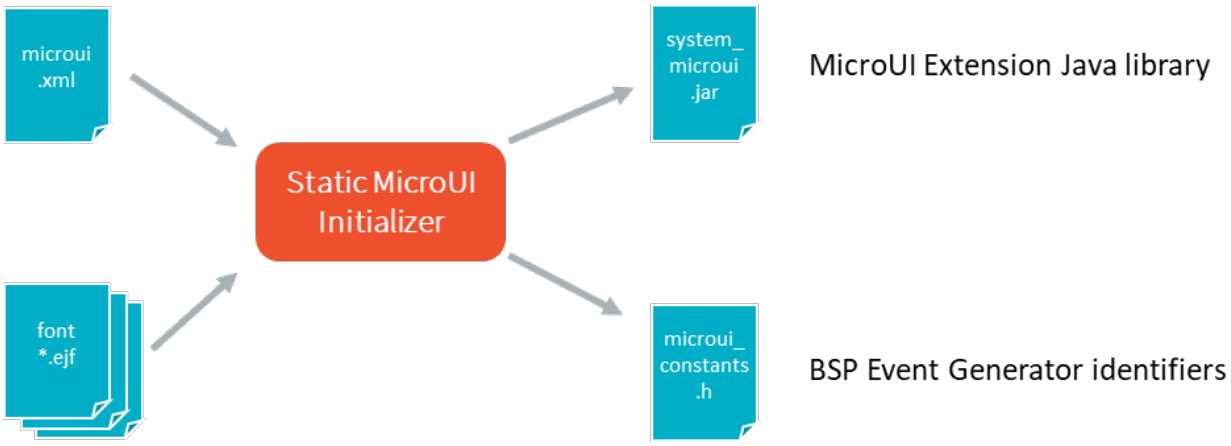


Fig. 42: Static MicroUI Initializer Process

XML File

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

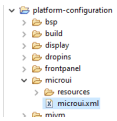


Fig. 43: Static MicroUI Initializer XML File

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"/>
```

(continues on next page)

(continued from previous page)

```
        <customrange start="0x21" end="0x3f"/>
    </font>
    <font file="C:\data\myfont.ejf"/>
</fonts>
```

- Event generators element: The `eventgenerators` element augments the initialization file with:
  - the configuration of the predefined MicroUI EventGenerator: Command, Buttons, States, Pointer and Touch.
  - the configuration of the generic MicroUI EventGenerator.

  The following snippet is an example of `eventgenerators` element:

```
<eventgenerators>
    <!-- Generic Event Generators -->
    <eventgenerator name="GENERIC" class="foo.bar.Zork">
        <property name="PROP1" value="3"/>
        <property name="PROP2" value="aaa"/>
    </eventgenerator>

    <!-- Predefined Event Generators -->
    <command name="COMMANDS"/>
    <buttons name="BUTTONS" extended="3"/>
    <buttons name="JOYSTICK" extended="5"/>
    <pointer name="POINTER" width="1200" height="1200"/>
    <touch name="TOUCH" display="DISPLAY"/>
    <states name="STATES" numbers="NUMBERS" values="VALUES"/>

</eventgenerators>

<array name="NUMBERS">
    <elem value="3"/>
    <elem value="2"/>
    <elem value="5"/>
</array>

<array name="VALUES">
    <elem value="2"/>
    <elem value="0"/>
    <elem value="1"/>
</array>
```

XML File Example

This common MicroUI initialization file initializes MicroUI with:

- a Display,
- a Command event generator,
- a Buttons event generator which targets *n* buttons (3 first buttons having extended features),
- a Buttons event generator which targets the buttons of a joystick,
- a Pointer event generator which targets a touch panel,

- a Font whose path is relative to this file.

```
<microui>

    <display name="DISPLAY"/>

    <eventgenerators>
        <command name="COMMANDS"/>
        <buttons name="BUTTONS" extended="3"/>
        <buttons name="JOYSTICK" extended="5"/>
        <touch name="TOUCH" display="DISPLAY"/>
    </eventgenerators>

    <fonts>
        <font file="resources\fonts\myfont.ejf"/>
    </fonts>

</microui>
```

Dependencies

No dependency.

Installation

The Static Initialization tool is part of the MicroUI module (see *MicroUI*). Install the MicroUI module to install the Static Initialization tool and fill all properties in MicroUI module configuration file (which must specify the name of the initialization file).

Use

The Static MicroUI Initializer tool is automatically launched during the VEE Port build.

6.14.5 Abstraction Layer API

Principle

The MicroUI implementation for MicroEJ requires an Abstraction Layer implementation. This Abstraction Layer implementation finalizes the MicroUI implementation started with the static initialization step (see *Static Initialization*) for a given VEE Port.

The Abstraction Layer implementation consists in a set of headers files to implement in C to target the hardware drivers. Some functions are mandatory, others are not. Some other headers files are also available to call UI engines internal functions.

For the simulator, some Front Panel interfaces and classes allow to specify the simulated VEE Port characteristics.

Embedded VEE Port

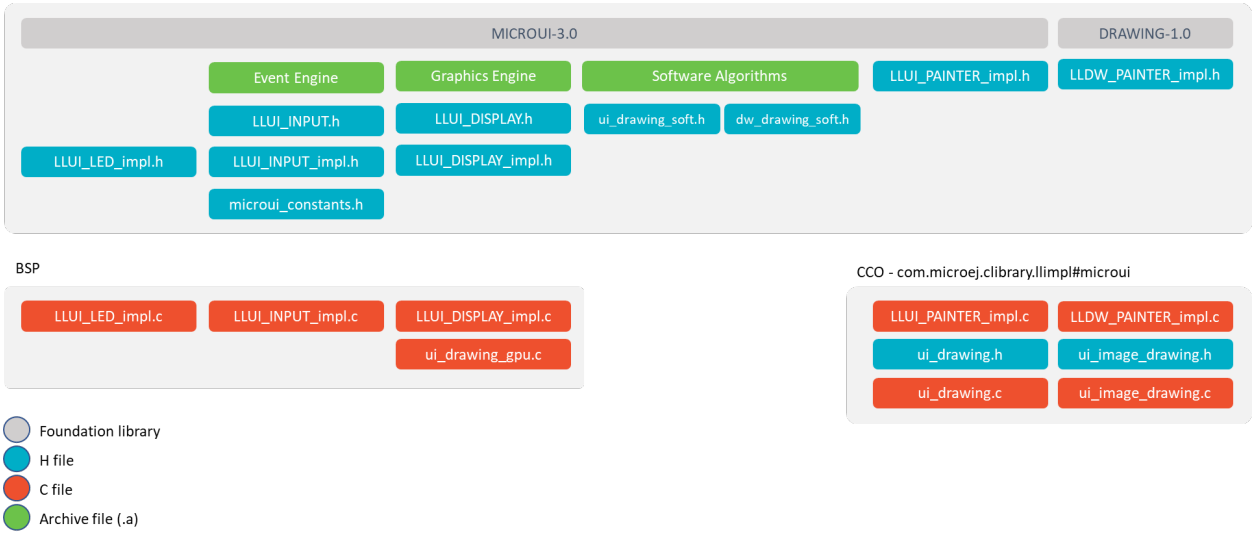


Fig. 44: MicroUI Embedded Abstraction Layer API

The specification of header files names is:

- Name starts with `LLUI_`.
- Second part name refers the UI engine: `DISPLAY`, `INPUT`, `LED`.
- Files whose name ends with `_impl` list functions to implement over hardware.
- Files whose name has no suffix list internal UI engines functions.

There are some exceptions :

- `LLUI_PAINTER_impl.h` and `LLDW_PAINTER_impl.h` list a subpart of UI Graphics Engine functions to implement (all MicroUI native drawing methods).
- `ui_drawing_soft.h` and `dw_drawing_soft.h` list all drawing methods implemented by the Graphics Engine.
- `microui_constants.h` is the file generated by the MicroUI Static Initializer (see *Static Initialization*).

The *MicroUI C module* provides a default implementation of the UI Pack Abstraction Layer API:

- `LLUI_PAINTER_impl.c` and `LLDW_PAINTER_impl.c` manage the synchronization with the Graphics Engine and redirect all drawings to `ui_drawing.h` and `ui_image_drawing.h`.
- `ui_drawing.h` and `ui_image_drawing.h` list all drawing methods the VEE Port can implement.
- `ui_drawing.c` and `ui_image_drawing.c` are the default implementation of `ui_drawing.h` and `ui_image_drawing.h` that redirects all drawings to `ui_drawing_soft.h` and `dw_drawing_soft.h`.

The BSP has to implement `LLUI_xxx` header files and optionally `ui_drawing.h` and `ui_image_drawing.h` (to draw using a GPU and/or to draw in a *custom BufferedImage*).

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

Simulator

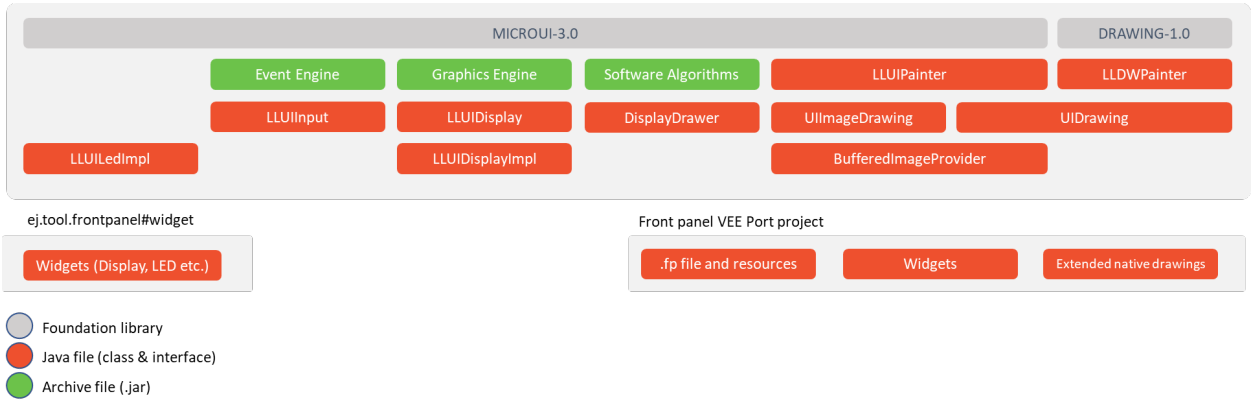


Fig. 45: MicroUI Simulator Abstraction Layer API

In the simulator the three UI engines are grouped in a mock called Front Panel. The Front Panel comes with a set of classes and interfaces which are the equivalent of headers file (`*.h`) of Embedded VEE Port.

The specification of class names is:

- Package are the same than the MicroUI library (ej.microui.display, ej.microui.event, ej.microui.led).
- Name start with `LLUI` .
- The second part of the name refers the UI engine: `Display` , `Input` , `Led` .
- Files whose name ends with `Impl` list methods to implement like in the embedded VEE Port.
- Files whose name has no suffix list internal UI engine functions.

There are some exceptions :

- `LLUIPainter.java` and `LLDWPainter.java` list a subpart of UI Graphics Engine functions (all MicroUI native drawing methods).
- `UIDrawing.java` and `DWDrawing.java` list all drawing methods the VEE Port can implement (and already implemented by the Graphics Engine).
- `EventXXX` list methods to create input events compatible with MicroUI implementation.

All files and their aims are described in *Simulation*.

6.14.6 LED

Principle

The LED module contains the C part of the MicroUI implementation which manages LED devices. This module is composed of only one element: an implementation of the Abstraction Layer APIs for the LEDs which must be provided by the BSP (see *LLUI_LED: LEDs*).

Functional Description

The LED module implements the MicroUI Leds framework. `LLUI_LED` specifies the Abstraction Layer APIs that receive orders from the Java world.

The Abstraction Layer APIs are the same for the LED which is connected to a `GPIO` (`0` or `1`), to a `PWM`, to a bus (`I2C`, `SPI`), etc. The BSP has the responsibility of interpreting the application parameter `intensity`.

Typically, when the LED is connected to a `GPIO`, the `intensity` "0" means "OFF", and all other values "ON". When the LED is connected via a `PWM`, the `intensity` "0" means "OFF", and all other values must configure the `PWM` duty cycle signal.

The BSP should be able to return the state of an LED. If it is not able to do so (for example `GPIO` is not accessible in read mode), the BSP has to save the LED state in a global variable. If not, the returned value may be wrong and the application may not be able to know the LEDs states.

Abstraction Layer API

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

The Abstraction Layer APIs to implement are listed in the header file `LLUI_LEDS_impl.h`. First, in the initialization function, the BSP must return the available number of LEDs the board provides. The other functions are used to turn the LEDs on and off.

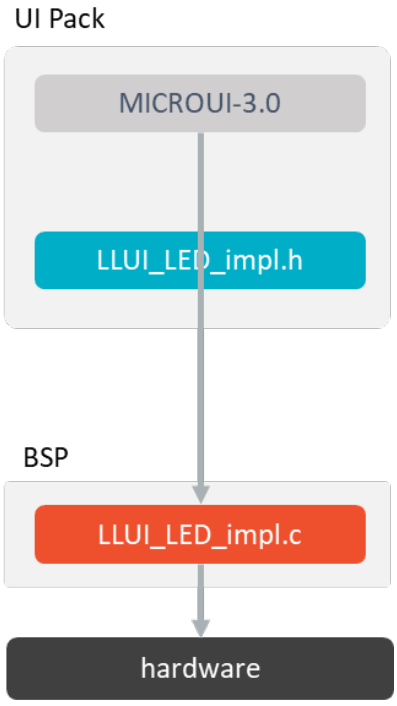


Fig. 46: Led Abstraction Layer API

When there is no LED on the board, a *stub* implementation of C library is available. This C library must be linked by the third-party C IDE when the MicroUI module is installed in the VEE Port. This stub library does not provide any Abstraction Layer API files.

Typical Implementation

This chapter helps to write a basic `LLUI_LEDS_impl.h` implementation. This implementation manages some two-state LEDs: on or off.

The pseudo-code calls external functions such as `LEDS_DRIVER_xxx` to symbolize the use of external drivers.

```
static void get_led_port_and_pin(int32_t ledID, int32_t* port, int32_t* pin)
{
   switch(ledID)
   {
      /* TODO */
      *port = ...;
      *pin = ...;
   }
}

jint LLUI_LED_IMPL_getIntensity(jint ledID)
{
   int32_t port;
   int32_t pin;
   get_led_port_and_pin(ledID, &port, &pin);

   return GPIO_ReadPin(port, pin) == GPIO_PIN_RESET ? LLUI_LED_MAX_INTENSITY : LLUI_LED_MIN_
↪INTENSITY;
}

jint LLUI_LED_IMPL_initialize(void)
{
   return DRIVER_LEDS_Init();  // return the available number of leds
}

void LLUI_LED_IMPL_setIntensity(jint ledID, jint intensity)
{
   int32_t port;
   int32_t pin;
   get_led_port_and_pin(ledID, &port, &pin);

   GPIO_WritePin(port, pin, 0 == intensity ? GPIO_PIN_RESET : GPIO_PIN_SET);
}
```

Dependencies

- MicroUI module (see *MicroUI*).
- `LLUI_LED_impl.h` implementation if standard implementation is chosen (see *Functional Description* and *LLUI_LED: LEDs*).

Installation

LEDs is a sub-part of MicroUI library. When the MicroUI module is installed, the LED module must be installed in order to be able to connect physical LEDs with VEE Port. If not installed, the *stub* module will be used.

In the VEE Port configuration file, check `UI` > `LEDs` to install LEDs.

Use

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

6.14.7 Input

Principle

The Input module contains the C part of the MicroUI implementation which manages input devices. This module is composed of two elements:

- the C part of MicroUI input API (a built-in C archive) called Input Engine,
- an implementation of Abstraction Layer APIs for the input devices that must be provided by the BSP (see *LLUI_INPUT: Input*).

Functional Description

The Input module implements the MicroUI `int` -based event generators' framework. `LLUI_INPUT` specifies the Abstraction Layer APIs that send events to the Java world.

Drivers for input devices must generate events that are sent, via a MicroUI Event Generator, to the application. An event generator accepts notifications from devices, and generates an event in a standard format that can be handled by the application. Depending on the MicroUI configuration, there can be several different types of event generator in the system, and one or more instances of each type.

Each MicroUI Event Generator represents one side of a pair of collaborative components that communicate using a shared buffer:

- The producer: the C driver connected to the hardware. As a producer, it sends its data into the communication buffer.
- The consumer: the MicroUI Event Generator. As a consumer, it reads (and removes) the data from the communication buffer.

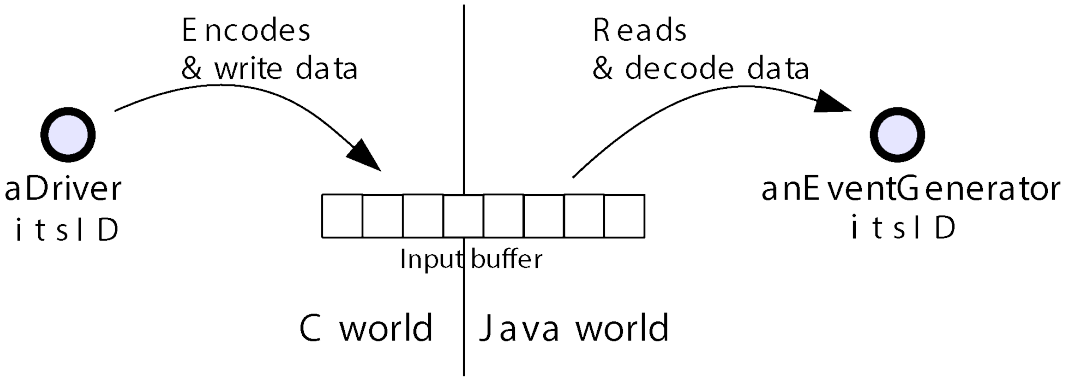


Fig. 47: Drivers and MicroUI Event Generators Communication

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.

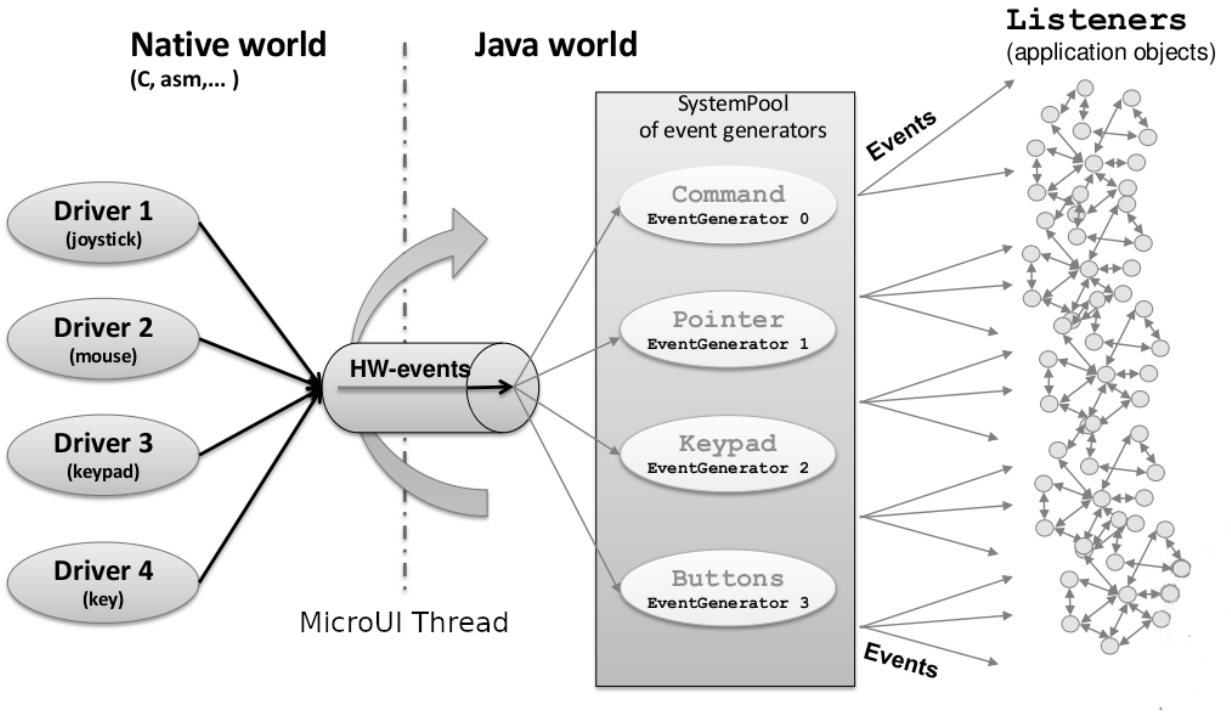


Fig. 48: MicroUI Events Framework

Driver Listener

Drivers may either interface directly with event generators, or they can send their notifications to a *Listener*, also written in C, and the listener passes the notifications to the event generator. This decoupling has two major benefits:

- The drivers are isolated from the MicroEJ libraries – they can even be existing code.
- The listener can translate the notification; so, for example, a joystick could generate pointer events.

Static Initialization

The event generators available on MicroUI startup (after the call to MicroUI.start()) are the event generators listed in the MicroUI description file (XML file). This file is a part of the MicroUI Static Initialization step (*Static Initialization*).

The order of event generators defines the unique identifier for each event generator. These identifiers are generated in a header file called `microui_constants.h`. The input driver (or its listener) has to use these identifiers to target a specific event generator.

If an unknown identifier is used or if two identifiers are swapped, the associated event may be never received by the application or may be misinterpreted.

Standard Event Generators

MicroUI provides a set of standard event generators: Command, Buttons, Pointer and States. For each standard generator, the Input Engine proposes a set of functions to create and send an event to this generator.

Static Initialization proposes an additional event generator: `Touch`. A touch event generator is a Pointer event generator whose area size is the display size where the touch panel is placed. Furthermore, contrary to a pointer, a *press* action is required to be able to have a *move* action (and so a *drag* action). The Input Engine proposes a set of functions to target a touch event generator (equal to a pointer event generator but with some constraints). The touch event generator is identified as a standard Pointer event generator, by consequence the Java application has to use the Pointer API to deal with a touch event generator.

According to the event generator, one or several parameters are required. The parameter format is event generator dependant. For instance a `Pointer` X-coordinate is encoded on 16 bits (0-65535 pixels).

Note: `Pointer` and `Touch` origin (point `0,0`) is the top-left point.

Generic Event Generators

MicroUI provides an abstract class GenericEventGenerator (package `ej.microui.event`). The aim of a generic event generator is to be able to send custom events from native world to the application. These events may be constituted by only one 32-bit word or by several 32-bit words (maximum 255).

On the application side, a subclass must be implemented by clients who want to define their own event generators. Two abstract methods must be implemented by subclasses:

- `eventReceived`: The event generator received an event from a C driver through the Abstraction Layer API `sendEvent` function.
- `eventsReceived`: The event generator received an event made of several `int` s.

The event generator is responsible for converting incoming data into a MicroUI event and sending the event to its listener. It should be defined during MicroUI Static Initialization step (in the XML file, see *Static Initialization*). This allows the MicroUI implementation to instantiate the event generator on startup.

If the event generator is not available in the application classpath, a warning is thrown (with a stack trace) and the application continues. In this case, all events sent by BSP to this event generator are ignored because no event generator is able to decode them.

Abstraction Layer API

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

The Abstraction Layer APIs to implement are listed in the header file `LLUI_INPUT_impl.h`. It allows events to be sent to the MicroUI implementation. The input drivers are allowed to add events directly using the event generator's unique ID (see *Static Initialization*). The drivers are fully dependent on the MicroEJ framework (a driver or a driver listener cannot be developed without MicroEJ because it uses the header file generated during the MicroUI initialization step).

To send an event to the application, the driver (or its listener) has to call one of the event engine function, listed in `LLUI_INPUT.h`. These functions take as parameter the MicroUI 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.

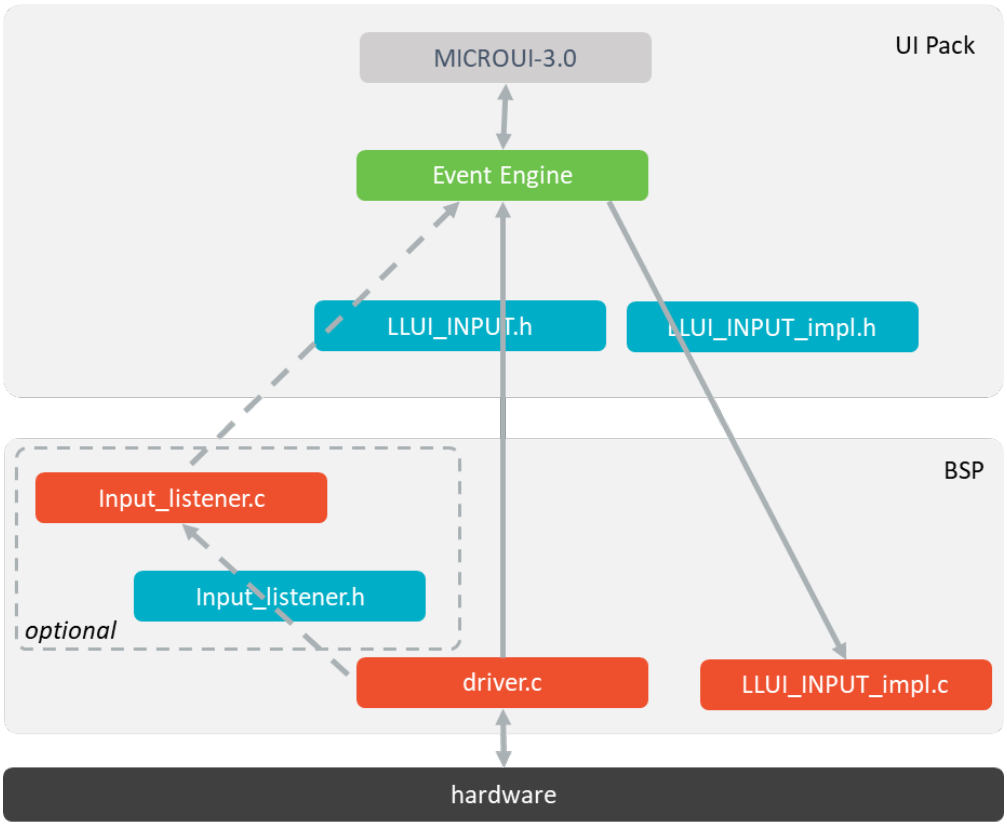


Fig. 49: Input Abstraction Layer API

When there is no input device on the board, a *stub* implementation of C library is available. This C library must be linked by the third-party C IDE when the MicroUI module is installed in the VEE Port. This stub library does not provide any Abstraction Layer API files.

Typical Implementation

This chapter helps to write a basic `LLUI_INPUT_impl.h` implementation. This implementation should be divided into several files:

- `LLUI_INPUT_impl.c` : implements `LLUI_INPUT_imp.h` and *receives* the input devices interrupts / callbacks (button press, touch move, etc.).
- `xxx_helper.c` : one helper per kind of input device (group of buttons, touch, etc.). It links the input device hardware status and the software status (MicroUI event status).
- `event_generator.c` : converts the input device hardware events in MicroUI events.

The pseudo-code calls external functions such as `BUTTONS_DRIVER_xxx` or `TOUCH_DRIVER_xxx` to symbolize the use of external drivers.

LLUI_INPUT_impl.c

Its main aim is to synchronize the Input Engine with the input devices. The Input Engine holds a circular FIFO to store the input devices' events. The use of this FIFO must be performed under the critical section. The concurrent actions "an input device adds a new event in the Input Engine" and "the Input Engine reads an event from the FIFO" must not be performed simultaneously. The implementation does not need to manage the concurrency: the Input Engine automatically calls the functions `LLUI_INPUT_IMPL_enterCriticalSection` and `LLUI_INPUT_IMPL_leaveCriticalSection` when an event is added or read.

- If the input devices add events under interrupt, the critical section must disable and re-enable the input devices' interrupts.
- If the input devices add events from an OS task, the critical section must use a semaphore to prevent scheduling.
- If both modes are used (typical use case), the critical section must be designed in consequence.

The following pseudo-code shows a typical implementation with:

- buttons under interrupt.
- touch panel from an OS task.

```
static xSemaphoreHandle _sem_input;

void LLUI_INPUT_IMPL_initialize(void)
{
   _sem_input = xSemaphoreCreateBinary();
   xSemaphoreGive(g_sem_input); // first take must pass

   BUTTONS_DRIVER_initialize();
   TOUCH_DRIVER_initialize();
}

jint LLUI_INPUT_IMPL_getInitialStateValue(jint stateMachinesID, jint stateID)
{
   // no state on this BSP
```

(continues on next page)

(continued from previous page)

```
   return 0;
}

void LLUI_INPUT_IMPL_enterCriticalSection()
{
   if (MICROEJ_FALSE == interrupt_is_in())
   {
      xSemaphoreTake(_sem_input, portMAX_DELAY);
      BUTTONS_DRIVER_disable_interrupts();
   }
   // else: already in secure state (under interrupt)
}

void LLUI_INPUT_IMPL_leaveCriticalSection()
{
   if (MICROEJ_FALSE == interrupt_is_in())
   {
      BUTTONS_DRIVER_enable_interrupts();
      xSemaphoreGive(_sem_input);
   }
   // else: already in secure state (under interrupt)
}
```

The other aim of this implementation is to *receive* the input devices' hardware events and to redirect these events to the dedicated helper.

```
// called by the touch panel dedicated task
void TOUCH_DRIVER_callback(uint8_t pressed, int32_t x, int32_t y)
{
   if (pressed)
   {
      // here, pen is down for sure
      TOUCH_HELPER_pressed(x, y);
   }
   else
   {
      // here, pen is up for sure
      TOUCH_HELPER_released();
   }
}

void GPIO_IRQHandler(int32_t button, uint32_t port, uint32_t pin)
{
   if (GPIO_PIN_SET == GPIO_ReadPin(port, pin))
   {
      // GPIO == 1 means "pressed"
      BUTTONS_HELPER_pressed(button);
   }
   else
   {
      // GPIO == 0 means "released"
      BUTTONS_HELPER_released(button);
```

(continues on next page)

(continued from previous page)

```
   }
}
```

buttons_helper.c

The Input Engine's FIFO might be full. In such a case, a new input device event cannot be added. Consequently, a button *release* event should not be added to the FIFO if the previous button *press* event had not been added. This helper keeps the *software* state: the input device's state seen by the application.

Note: This helper does not convert the hardware event into a MicroUI event. It lets `event_generator.c` performs this job.

```
static uint8_t buttons_pressed[NUMBER_OF_BUTTONS];

void BUTTONS_HELPER_initialize(void)
{
   for(uint32_t i = 0; i < NUMBER_OF_BUTTONS; i++)
   {
      buttons_pressed[i] = MICROEJ_FALSE;
   }
}

void BUTTONS_HELPER_pressed(int32_t buttonId)
{
   // button is pressed

   if (MICROEJ_TRUE == buttons_pressed[buttonId])
   {
      // button was pressed => repeat event  (don't care if event is lost)
      EVENT_GENERATOR_button_repeated(buttonId);
   }
   else
   {
      // button was released => press event
      if (LLUI_INPUT_OK == EVENT_GENERATOR_button_pressed(buttonId) )
      {
         // the event has been managed: we can store the new button state
         // button is pressed now
         buttons_pressed[buttonId] = MICROEJ_TRUE;
      }
      // else: event has been lost: stay in "release" state
   }
}

void BUTTONS_HELPER_repeated(int32_t buttonId)
{
   // manage this repeat event like a press event to check "software" button state
   BUTTONS_HELPER_pressed(buttonId);
}
```

(continues on next page)

(continued from previous page)

```
void BUTTONS_HELPER_released(int32_t buttonId)
{
   // button is now released

   if (MICROEJ_TRUE == buttons_pressed[buttonId])
   {
      // button was pressed => release event
      if (LLUI_INPUT_OK == EVENT_GENERATOR_button_released(buttonId) )
      {
         // the event has been managed: we can store the new button state
         // button is released now
         buttons_pressed[buttonId] = MICROEJ_FALSE;
      }
      // else: event has been lost: stay in "press" state
   }
   // else: already released
}
```

touch_helper.c

The Input Engine's FIFO might be full. In such a case, a new input device event cannot be added. Consequently, a touch *move* / *drag* event should not be added to the FIFO if the previous touch *press* event had not been added. This helper keeps the *software* state: the input device's state seen by the application.

This helper also filters the touch panel events. It uses two defines `FIRST_MOVE_PIXEL_LIMIT` and `MOVE_PIXEL_LIMIT` to reduce the number of events sent to the application (values are expressed in pixels).

Note: This helper does not convert the hardware event in the MicroUI event. It lets `event_generator.c` performs this job.

```
// Number of pixels to generate a move after a press
#ifndef FIRST_MOVE_PIXEL_LIMIT
#error "Please set the define FIRST_MOVE_PIXEL_LIMIT (in pixels)"
#endif

// Number of pixels to generate a move after a move
#ifndef MOVE_PIXEL_LIMIT
#error "Please set the define MOVE_PIXEL_LIMIT (in pixels)"
#endif

#define DIFF(a,b)             ((a) < (b) ? (b-a) : (a-b))
#define KEEP_COORD(p,n,limit)    (DIFF(p,n) <= limit ? MICROEJ_FALSE : MICROEJ_TRUE)
#define KEEP_PIXEL(px,x,py,y,limit) (KEEP_COORD(px,x,limit) || KEEP_COORD(py,y,limit))
#define KEEP_FIRST_MOVE(px,x,py,y)  (KEEP_PIXEL(px,x,py,y, FIRST_MOVE_PIXEL_LIMIT))
#define KEEP_MOVE(px,x,py,y)     (KEEP_PIXEL(px,x,py,y, MOVE_PIXEL_LIMIT))

static uint8_t touch_pressed = MICROEJ_FALSE;
static uint8_t touch_moved = MICROEJ_FALSE;
static uint16_t previous_touch_x, previous_touch_y;
```

(continues on next page)

(continued from previous page)

```
void TOUCH_HELPER_pressed(int32_t x, int32_t y)
{
   // here, the pen is down for sure

   if (MICROEJ_TRUE == touch_pressed)
   {
      // pen was down => move event

      // keep pixel according first "move" event or not
      int keep_pixel;
      if(MICROEJ_TRUE == touch_moved)
      {
         keep_pixel = KEEP_MOVE(previous_touch_x, x, previous_touch_y, y);
      }
      else
      {
         keep_pixel = KEEP_FIRST_MOVE(previous_touch_x, x, previous_touch_y, y);
      }

      if (MICROEJ_TRUE == keep_pixel)
      {
         // store the new pixel
         previous_touch_x = x;
         previous_touch_y = y;
         touch_moved = MICROEJ_TRUE;

         // send a MicroUI touch event (don't care if event is lost)
         EVENT_GENERATOR_touch_moved(x, y);
      }
      // else: same position; no need to send an event
   }
   else
   {
      // pen was up => press event
      if (LLUI_INPUT_OK == EVENT_GENERATOR_touch_pressed(x, y))
      {
         // the event has been managed: we can store the new touch state
         // touch is pressed now
         previous_touch_x = x;
         previous_touch_y = y;
         touch_pressed = MICROEJ_TRUE;
         touch_moved = MICROEJ_FALSE;
      }
      // else: event has been lost: stay in "release" state
   }
}

void TOUCH_HELPER_moved(int32_t x, int32_t y)
{
   // manage this move like a press event to check "software" touch state
   TOUCH_HELPER_pressed(x, y);
```

(continues on next page)

(continued from previous page)

```
}

void TOUCH_HELPER_released(void)
{
   // here, the pen is up for sure

   if (MICROEJ_TRUE == touch_pressed)
   {
      // pen was down => release event
      if (LLUI_INPUT_OK == EVENT_GENERATOR_touch_released())
      {
         // the event has been managed: we can store the new touch state
         // touch is released now
         touch_pressed = MICROEJ_FALSE;
      }
      // else: event has been lost: stay in "press | move" state
   }
   // else: the pen was already up
}
```

event_generator.c

This file aims to convert the events (received by `LLUI_INPUT_impl.c` and then filtered by `xxx_helper.c`) to the application through the Input Engine.

This C file should be the only C file to include the header file `microui_constants.h`. This header file has been generated during the VEE Port build (see *Static Initialization*). It holds some defines that describe the available list of MicroUI Event Generators. Each MicroUI Event Generator has its identifier: 0 to *n-1*.

A button event is often converted in the MicroUI Command event. That allows the application to be button-independent: the application is not waiting for button 0 or button 1 events but MicroUI Command `ESC` or `LEFT` for instance. The following pseudo-code converts the buttons events in MicroUI Command events.

Note: Each hardware event can be converted into another kind of MicroUI event. For instance, a joystick can simulate a MicroUI Pointer; a touch panel can be reduced to a set of MicroUI Commands (left, right, top, left), etc.

```
#include "microui_constants.h"

static uint32_t _get_button_command(int32_t button_id)
{
   switch (button_id)
   {
   default:
   case BUTTON_WAKEUP_ID:
      return LLUI_INPUT_COMMAND_ESC;
   case BUTTON_TAMPER_ID:
      return LLUI_INPUT_COMMAND_MENU;
   }
}
```

(continues on next page)

(continued from previous page)

```
int32_t EVENT_GENERATOR_button_pressed(int32_t buttonId)
{
   return LLUI_INPUT_sendCommandEvent(MICROUI_EVENTGEN_COMMANDS, _get_button_
↪command(buttonId));
}

int32_t EVENT_GENERATOR_button_repeated(int32_t buttonId)
{
   return LLUI_INPUT_sendCommandEvent(MICROUI_EVENTGEN_COMMANDS, _get_button_
↪command(buttonId));
}

int32_t EVENT_GENERATOR_button_released(int32_t buttonId)
{
   // do not send a Command event on the release event
   return LLUI_INPUT_OK; // the event has been managed
}

int32_t EVENT_GENERATOR_touch_pressed(int32_t x, int32_t y)
{
   return LLUI_INPUT_sendTouchPressedEvent(MICROUI_EVENTGEN_TOUCH, x, y);
}

int32_t EVENT_GENERATOR_touch_moved(int32_t x, int32_t y)
{
   return LLUI_INPUT_sendTouchMovedEvent(MICROUI_EVENTGEN_TOUCH, x, y);
}

int32_t EVENT_GENERATOR_touch_released(void)
{
   return LLUI_INPUT_sendTouchReleasedEvent(MICROUI_EVENTGEN_TOUCH);
}
```

Event Buffer

MicroUI is using a circular buffer to manage the input events. As soon as an event is added, removed, or replaced in the queue, the event engine calls the associated Abstraction Layer API (LLAPI) `LLUI_INPUT_IMPL_log_queue_xxx()`. This LLAPI allows the BSP to log this event to dump it later thanks to a call to `LLUI_INPUT_dump()`.

Note: When the functions `LLUI_INPUT_IMPL_log_queue_xxx()` are not implemented, a call to `LLUI_INPUT_dump()` has no effect (there is no default logger).

The following steps describe how the logger is called:

1. On startup, MicroUI calls `LLUI_INPUT_IMPL_log_queue_init()`: it gives the event buffer. The implementation should prepare its logger.
2. The BSP adds or replaces an event in the queue, the event engine calls `LLUI_INPUT_IMPL_log_queue_add()` or `LLUI_INPUT_IMPL_log_queue_replace()`. The implementation should store the event metadata: buffer index, event size, etc.

3. If the event cannot be added because the queue is full, the event engine calls `LLUI_INPUT_IMPL_log_queue_full()`. The implementation can print a warning, throw an error, etc.
4. MicroUI reads an event, the event engine calls `LLUI_INPUT_IMPL_log_queue_read()`. The implementation has to update its metadata (if required).

The following steps describe how the dump is performed:

1. The BSP calls `LLUI_INPUT_dump()`: the event engine starts a dump of the event buffer.
2. First, the event engine dumps the older events. It calls `LLUI_INPUT_IMPL_log_dump()` for each old event. The log type value is `0`; it means that all logs are the events or events' data already consumed (*past* events), and the first log is the latest event or data stored in the queue.
3. Then, the event engine dumps the *future* events (events not consumed yet by MicroUI). It calls `LLUI_INPUT_IMPL_log_dump()` for each new event. The log type value is `1`; it means that all logs are the events or data not consumed yet (*future* events).
4. The *future* events can target a MicroUI object (a *Displayable* for a *requestRender* event, a *Runnable* for a *callSerially* event, etc.). The event engine notifies the logger to print the MicroUI objects by calling `LLUI_INPUT_IMPL_log_dump()` with `2` as log type value.
5. Finally, the event engine notifies the logger about the end of the dump by calling `LLUI_INPUT_IMPL_log_dump()` with `3` as log type value.

Warning: The dump of MicroUI objects linked to the *future* events is only available with the MicroEJ Architectures 7.16 and higher. With older MicroEJ Architectures, nothing is dumped.

An implementation is available on the *MicroUI C module*. This logger is constituted with two files:

- `LLUI_INPUT_LOG_impl.c`: this file holds some metadata for each event. When the event engine calls `LLUI_INPUT_IMPL_log_dump()`, the logger retrieves the event metadata and calls `microui_event_decoder.c` functions. To enable this logger, set the define `MICROUIEVENTDECODER_ENABLED` in `microui_event_decoder_conf.h`.
- `microui_event_decoder.c`: this file describes the MicroUI events. It has to be customized with the MicroUI event generators identifiers. See `microui_event_decoder_conf.h`.

Example of a dump:

```
============================== MicroUI FIFO Dump ===============================
--------------------------------- Old Events ---------------------------------
[27: 0x00000000] garbage
[28: 0x00000000] garbage
[...]
[99: 0x00000000] garbage
[00: 0x08000000] Display SHOW Displayable (Displayable index = 0)
[01: 0x00000008] Command HELP (event generator 0)
[02: 0x0d000000] Display REPAINT Displayable (Displayable index = 0)
[03: 0x07030000] Input event: Pointer pressed (event generator 3)
[04: 0x009f0063]    at 159,99 (absolute)
[05: 0x07030600] Input event: Pointer moved (event generator 3)
[06: 0x00aa0064]    at 170,100 (absolute)
[07: 0x02030700] Pointer dragged (event generator 3)
[08: 0x0d000000] Display REPAINT Displayable (Displayable index = 0)
[09: 0x07030600] Input event: Pointer moved (event generator 3)
[10: 0x00b30066]    at 179,102 (absolute)
```

(continues on next page)

(continued from previous page)

```
[11: 0x02030700] Pointer dragged (event generator 3)
[12: 0x0d000000] Display REPAINT Displayable (Displayable index = 0)
[13: 0x07030600] Input event: Pointer moved (event generator 3)
[14: 0x00c50067]    at 197,103 (absolute)
[15: 0x02030700] Pointer dragged (event generator 3)
[16: 0x0d000000] Display REPAINT Displayable (Displayable index = 0)
[17: 0x07030600] Input event: Pointer moved (event generator 3)
[18: 0x00d00066]    at 208,102 (absolute)
[19: 0x02030700] Pointer dragged (event generator 3)
[20: 0x0d000000] Display REPAINT Displayable (Displayable index = 0)
[21: 0x07030100] Input event: Pointer released (event generator 3)
[22: 0x00000000]    at 0,0 (absolute)
[23: 0x00000008] Command HELP (event generator 0)
--------------------------------- New Events ---------------------------------
[24: 0x0d000000] Display REPAINT Displayable (Displayable index = 0)
[25: 0x07030000] Input event: Pointer pressed (event generator 3)
[26: 0x002a0029]    at 42,41 (absolute)
-------------------------- New Events' Java objects --------------------------
[java/lang/Object[2]@0xC000FD1C
     [0] com/microej/examples/microui/mvc/MVCDisplayable@0xC000BAC0
     [1] null
================================================================================
```

Notes:

- The event `24` holds an object in the events objects array (a `Displayable`); its object index is `0`.
- An object is `null` when the memory slot has been used during the application execution but freed at the dump time.
- The object array' size is the maximum of non-null objects reached during application execution.
- The indices of old events are out-of-date: the memory slot is now null or reused by a newer event.
- The event `25` targets the event generator `3`; the identifier is available in `microui_constants.h` (created during the VEE Port build, see *Static Initialization*).
- The events `27` to `99` cannot be identified (no metadata or partial event content due to circular queue management).
- Refers to the implementation on the *MicroUI C module* to have more information about the format of the event; this implementation is always up-to-date with the MicroUI implementation.

Dependencies

- MicroUI module (see *MicroUI*)
- Static MicroUI initialization step (see *Static Initialization*). This step generates a header file which contains some unique event generator IDs. These IDs must be used in the BSP to make the link between the input devices drivers and the MicroUI `Event Generator`s.
- `LLUI_INPUT_impl.h` implementation (see *LLUI_INPUT: Input*).
- The *MicroUI C module* to optionally use the *default input logger*.

Installation

Input module is a sub-part of the MicroUI library. The Input module is installed at same time than MicroUI module.

Use

The MicroUI Input APIs are available in the classes of packages `ej.microui.event` and `ej.microui.event.generator`.

6.14.8 Display

Principle

The Display module contains the C part of the MicroUI implementation which manages graphical displays. This module is composed of three elements:

- the C part of MicroUI Display API (a built-in C archive) called Graphics Engine,
- an implementation of Abstraction Layer APIs for the displays (LLUI_DISPLAY) that the BSP must provide (see *LLUI_DISPLAY: Display*),
- an implementation of Abstraction Layer APIs for MicroUI drawings.

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 Abstraction Layer.

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.

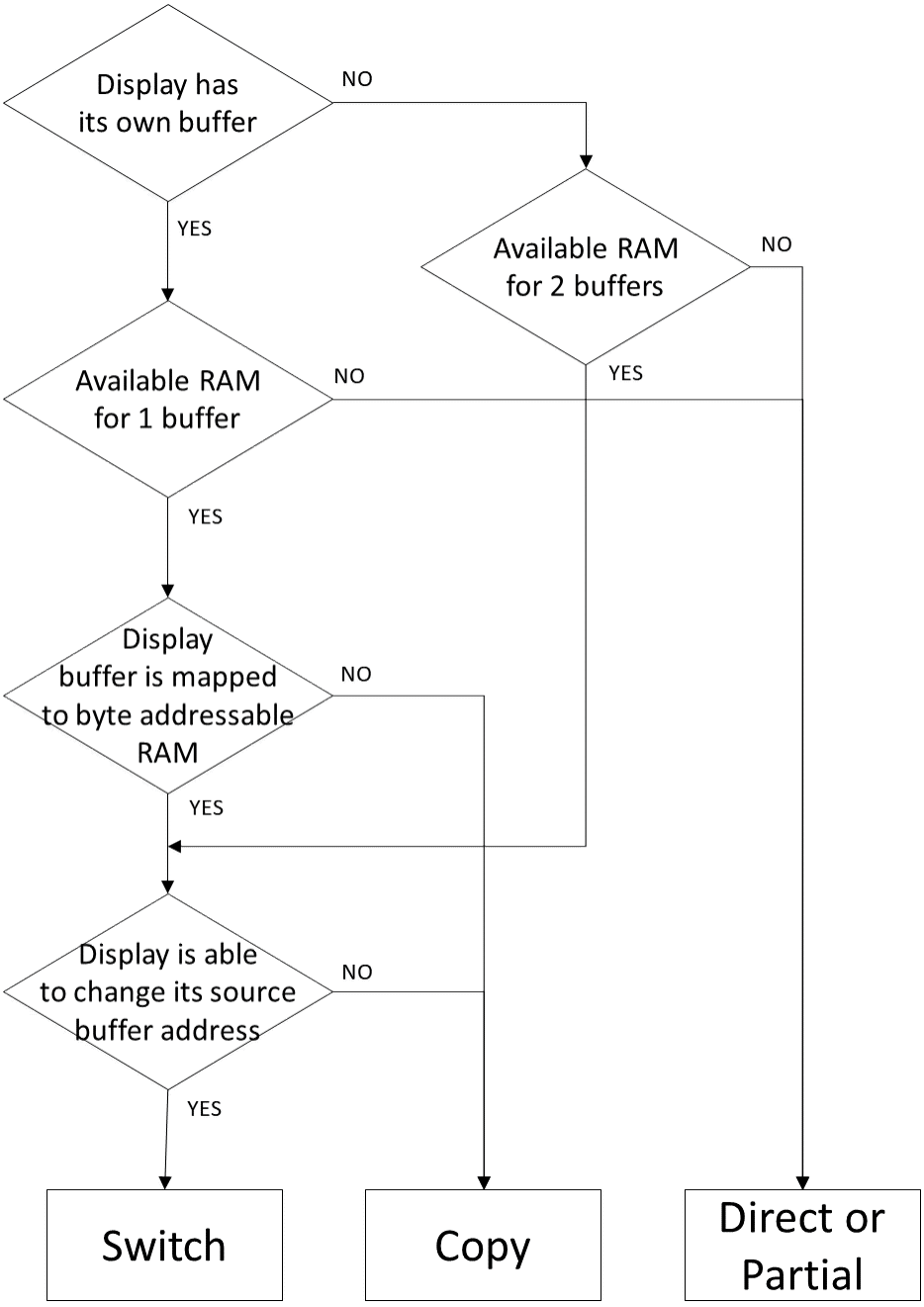


Fig. 50: Buffer Modes

Implementation

The Graphics Engine does not depend on the type of buffer mode. The implementation of Display.flush() calls the Abstraction Layer 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.

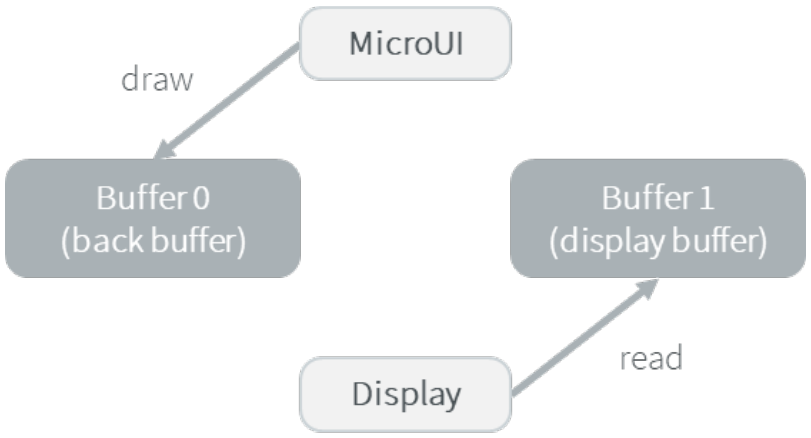
The next sections describe the work to do for each mode.

Switch

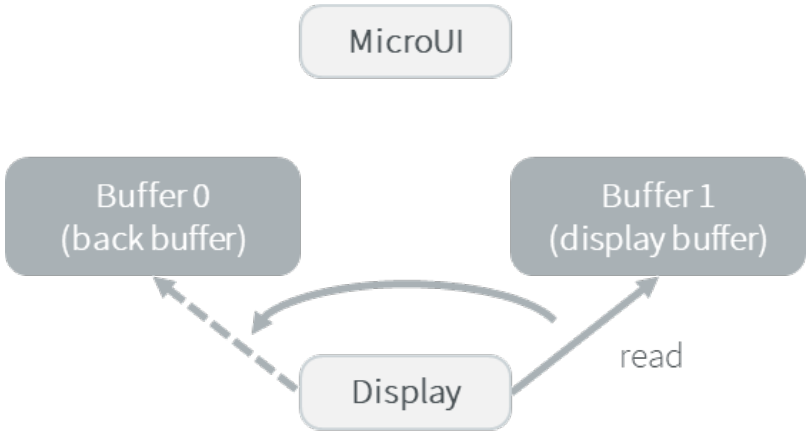
The switch mode is a double-buffered mode where two buffers in RAM alternately play the role of the back buffer and the display buffer. The display source is alternatively changed from one buffer to the other. Switching the source address may be done asynchronously. The synchronize function is called 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*.

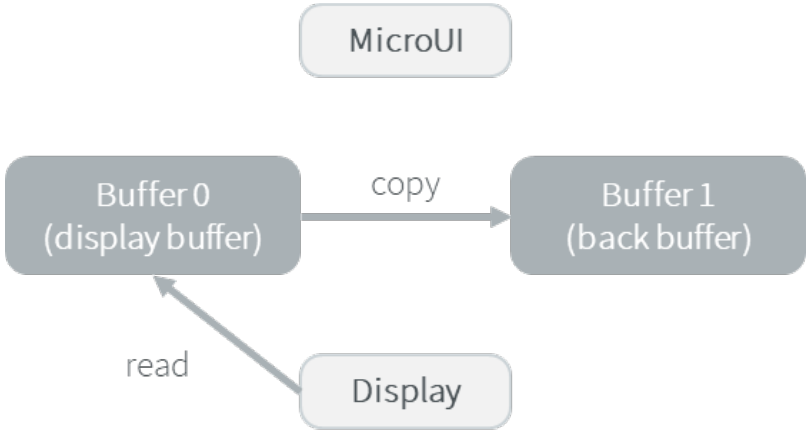
- *Step 1:* Drawing
  MicroUI is drawing in buffer 0 (back buffer) and the display is reading its contents from buffer 1 (display buffer).



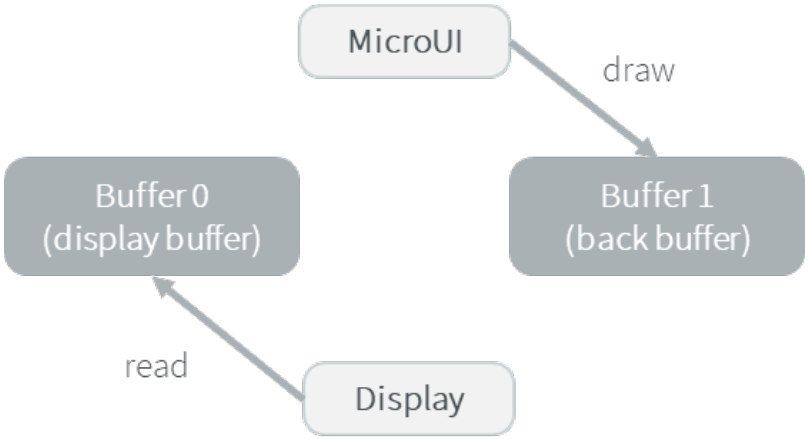
- *Step 2:* Switch
  The drawing is done. Set that the next read will be done from buffer 0.
  Note that the display "hardware component" asynchronously continues to read data from buffer 1.



- *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.

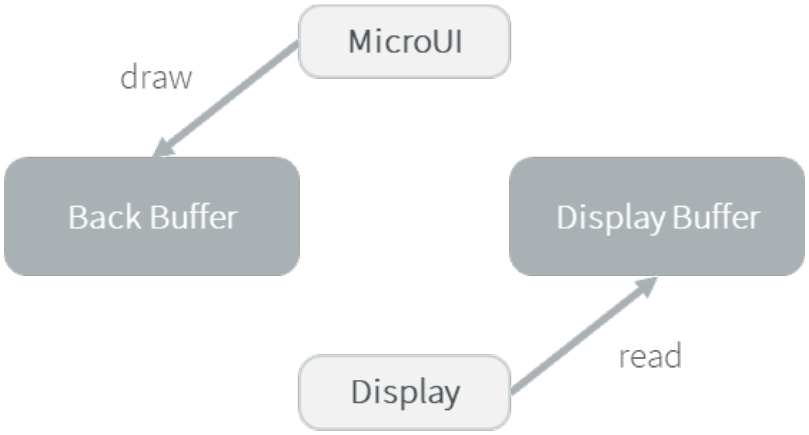


Copy

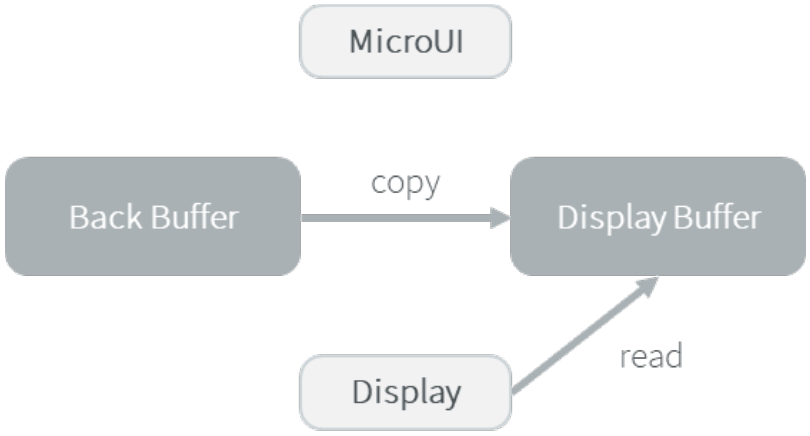
The copy mode is a double-buffered mode where the back buffer is in RAM and has a fixed address. To update the display, data is sent to the display buffer. This can be done either by a memory copy or by sending bytes using a bus, such as SPI or I2C.

Synchronization steps are described *below*.

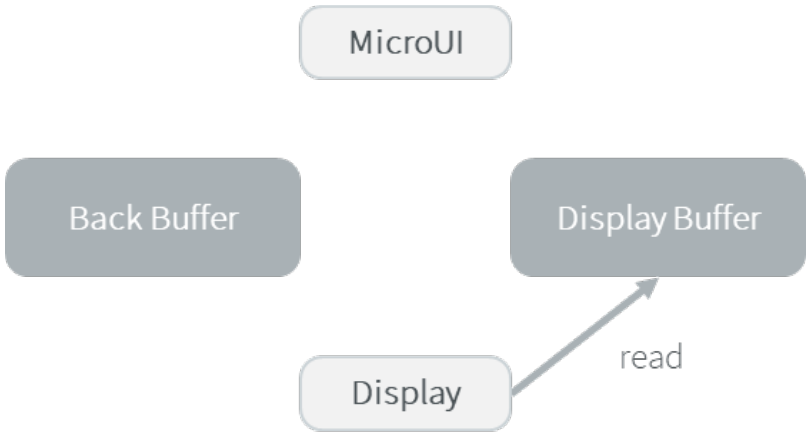
- *Step 1:* Drawing
  MicroUI is drawing in the back buffer and the display is reading its content from the display buffer.



- *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.



Switch VS Copy

Where *Switch* mode is possible, *Copy* can also be used:

- In *Switch* mode, the copy from the new frame buffer to the new back buffer consists of restoring the past: after this copy, the application retrieves its previous drawings in the back buffer.
- In *Copy* mode, the copy from the back buffer to the frame buffer consists of copying the application drawings into the display buffer.

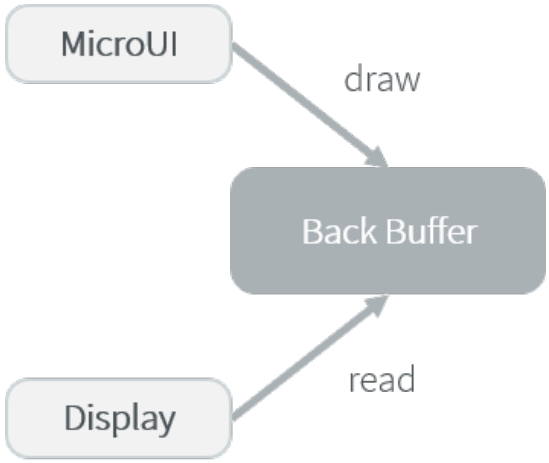
However, when possible, the *Switch* mode should be implemented:

- The new frame buffer data is available instantly. As soon as the LCD controller has updated its frame buffer address, the data is ready to be sent to the LCD. In *Copy* mode, the process of copying the data to the display buffer occurs while the LCD controller is reading them. Therefore, the buffer copy has to be faster than the LCD controller reading. If this requirement is not met, the LCD controller will send a mix of new and old data (because the buffer copy is not completely finished).
- The synchronization with the LCD controller is more effortless. An interrupt is thrown as soon as the LCD controller has updated its frame buffer address. Then, the copy buffer process can start. In *Copy* mode, the copy buffer process should be synchronized with the LCD tearing signal.
- During the copy, the same buffer is used as source by the copy buffer process (DMA, memcopy, etc.) and by the LCD controller. Both masters are using the same RAM section in *reading*. In *Copy* mode, the same RAM

section switches in *Write* mode (copy buffer process) and *Read* mode (LCD controller).

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 13: Byte Layout: line

BPP	MSB							LSB
4	pixel 1				pixel 0			
2	pixel 3		pixel 2		pixel 1		pixel 0	
1	pixel 7	pixel 6	pixel 5	pixel 4	pixel 3	pixel 2	pixel 1	pixel 0

Table 14: 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 15: 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 16: 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 17: 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 18: 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 19: 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

Principle

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*).

Standard

When the value is one among this list: `ARGB8888 | RGB888 | RGB565 | ARGB1555 | ARGB4444 | C4 | C2 | C1`, the Display module considers the pixels representation as **standard**. All standard representations are internally managed by the Display module, by the *Front Panel* and by the *Image Generator*. No specific support is required as long as a VEE Port is using a standard representation. It can:

- generate at compile-time RAW images in the same format than display pixel format,
- convert at runtime MicroUI 32-bit colors in display pixel format,
- simulate at runtime the display pixel format.

Note: The custom implementations of the image generator, some Abstraction Layer APIs, and Front Panel APIs are ignored by the Display module when a standard pixel representation is selected.

According to the chosen format, some color data can be lost or cropped.

- ARGB8888: the pixel uses 32 bits-per-pixel (alpha[8], red[8], green[8] and blue[8]).

```
u32 convertARGB8888toLCDPixel(u32 c){
    return c;
}

u32 convertLCDPixeltoARGB8888(u32 c){
    return c;
}
```

- RGB888: the pixel uses 24 bits-per-pixel (alpha[0], red[8], green[8] and blue[8]).

```
u32 convertARGB8888toLCDPixel(u32 c){
    return c & 0xffffff;
}

u32 convertLCDPixeltoARGB8888(u32 c){
    return 0
            | 0xff000000
```

(continues on next page)

(continued from previous page)

```
            | c
            ;
}
```

- RGB565: the pixel uses 16 bits-per-pixel (alpha[0], red[5], green[6] and blue[5]).

```
u32 convertARGB8888toLCDPixel(u32 c){
    return 0
            | ((c & 0xf80000) >> 8)
            | ((c & 0x00fc00) >> 5)
            | ((c & 0x0000f8) >> 3)
            ;
}

u32 convertLCDPixeltoARGB8888(u32 c){
    return 0
            | 0xff000000
            | ((c & 0xf800) << 8)
            | ((c & 0x07e0) << 5)
            | ((c & 0x001f) << 3)
            ;
}
```

- ARGB1555: the pixel uses 16 bits-per-pixel (alpha[1], red[5], green[5] and blue[5]).

```
u32 convertARGB8888toLCDPixel(u32 c){
    return 0
            | (((c & 0xff000000) == 0xff000000) ? 0x8000 : 0)
            | ((c & 0xf80000) >> 9)
            | ((c & 0x00f800) >> 6)
            | ((c & 0x0000f8) >> 3)
            ;
}

u32 convertLCDPixeltoARGB8888(u32 c){
    return 0
            | ((c & 0x8000) == 0x8000 ? 0xff000000 : 0x00000000)
            | ((c & 0x7c00) << 9)
            | ((c & 0x03e0) << 6)
            | ((c & 0x001f) << 3)
            ;
}
```

- 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)
            ;
```

(continues on next page)

(continued from previous page)

```
}

u32 convertLCDPixeltoARGB8888(u32 c){
    return 0
            | ((c & 0xf000) << 16)
            | ((c & 0xf000) << 12)
            | ((c & 0x0f00) << 12)
            | ((c & 0x0f00) << 8)
            | ((c & 0x00f0) << 8)
            | ((c & 0x00f0) << 4)
            | ((c & 0x000f) << 4)
            | ((c & 0x000f) << 0)
            ;
}
```

- C4: the pixel uses 4 bits-per-pixel (grayscale[4]).

```
u32 convertARGB8888toLCDPixel(u32 c){
    return (toGrayscale(c) & 0xff) / 0x11;
}

u32 convertLCDPixeltoARGB8888(u32 c){
    return 0xff000000 | (c * 0x111111);
}
```

- C2: the pixel uses 2 bits-per-pixel (grayscale[2]).

```
u32 convertARGB8888toLCDPixel(u32 c){
    return (toGrayscale(c) & 0xff) / 0x55;
}

u32 convertLCDPixeltoARGB8888(u32 c){
    return 0xff000000 | (c * 0x555555);
}
```

- C1: the pixel uses 1 bit-per-pixel (grayscale[1]).

```
u32 convertARGB8888toLCDPixel(u32 c){
    return (toGrayscale(c) & 0xff) / 0xff;
}

u32 convertLCDPixeltoARGB8888(u32 c){
    return 0xff000000 | (c * 0xffffff);
}
```

Driver-Specific

The Display module considers the pixel representation as **driver-specific** when the value is one among this list: `1 | 2 | 4 | 8 | 16 | 24 | 32`. This mode is often used when the pixel representation is not `ARGB` or `RGB` but `BGRA` or `BGR` instead. This mode can also be used when the number of bits for a color component (alpha, red, green, or blue) is not standard or when the value does not represent a color but an index in a *CLUT*. This mode requires some specific support in the VEE Port:

- An extension of the image generator is mandatory: see *Extended Mode* to convert MicroUI's standard 32-bit ARGB colors to display pixel format.
- The Front Panel widget `Display` requires an extension to convert the MicroUI 32-bit colors in display pixel format and vice-versa, see *Widget Display*.
- The driver must implement functions that convert MicroUI's standard 32-bit ARGB colors to display pixel format and vice-versa: see *Color Conversions*.

The following example illustrates the use of specific format BGR565 (the pixel uses 16 bits-per-pixel (alpha[0], red[5], green[6] and blue[5]):

1. Configure the VEE Port:
   - Create or open the VEE Port configuration project file `display/display.properties`:

```
bpp=16
```

2. Image Generator:
   - Create a project as described *here*.
   - Create the class `com.microej.graphicalengine.generator.MicroUIGeneratorExtension` that extends the class `com.microej.tool.ui.generator.BufferedImageLoader`.
   - Fill the method `convertARGBColorToDisplayColor()`:

```
public class MicroUIGeneratorExtension extends BufferedImageLoader {
  @Override
  public int convertARGBColorToDisplayColor(int color) {
     return ((color & 0xf80000) >> 19) | ((color & 0x00fc00) >> 5) | ((color &␣
↪0x0000f8) << 8);
  }
}
```

   - Configure the Image Generator' service loader: add the file `/META-INF/services/com.microej.tool.ui.generator.MicroUIRawImageGeneratorExtension`:

```
com.microej.graphicalengine.generator.MicroUIGeneratorExtension
```

   - Build the module (click on the blue button).
   - Copy the generated jar file (`imageGeneratorMyPlatform.jar`) in the VEE Port configuration project: `/dropins/tools/`.

2. Simulator (Front Panel):
   - Create the class `com.microej.fp.MyDisplayExtension` that implements the interface `ej.fp.widget.Display.DisplayExtension`:

```
public class MyDisplayExtension implements DisplayExtension {

   @Override
   public int convertARGBColorToDisplayColor(Display display, int color) {
      return ((color & 0xf80000) >> 19) | ((color & 0x00fc00) >> 5) | ((color &␣
↪0x0000f8) << 8);
   }

   @Override
   public int convertDisplayColorToARGBColor(Display display, int color) {
      return ((color & 0x001f) << 19) | ((color & 0x7e00) << 5) | ((color & 0xf800) >>␣
↪8) | 0xff000000;
   }

   @Override
   public boolean isColor(Display display) {
      return true;
   }

   @Override
   public int getNumberOfColors(Display display) {
      return 1 << 16;
   }
}
```

- Configure the widget `Display` in the `.fp` file by referencing the display extension:

```
<ej.fp.widget.Display x="41" y="33" width="320" height="240" extensionClass="com.
↪microej.fp.MyDisplayExtension"/>
```

3. Build the VEE Port as usual
4. Update the `LLUI_DISPLAY` implementation by adding the following functions:

```
uint32_t LLUI_DISPLAY_IMPL_convertARGBColorToDisplayColor(uint32_t color)
{
   return ((color & 0xf80000) >> 19) | ((color & 0x00fc00) >> 5) | ((color & 0x0000f8) <
↪< 8);
}

uint32_t LLUI_DISPLAY_IMPL_convertDisplayColorToARGBColor(uint32_t color)
{
  return ((color & 0x001f) << 19) | ((color & 0x7e00) << 5) | ((color & 0xf800) >> 8) |␣
↪0xff000000;
}
```

Abstraction Layer API

Overview

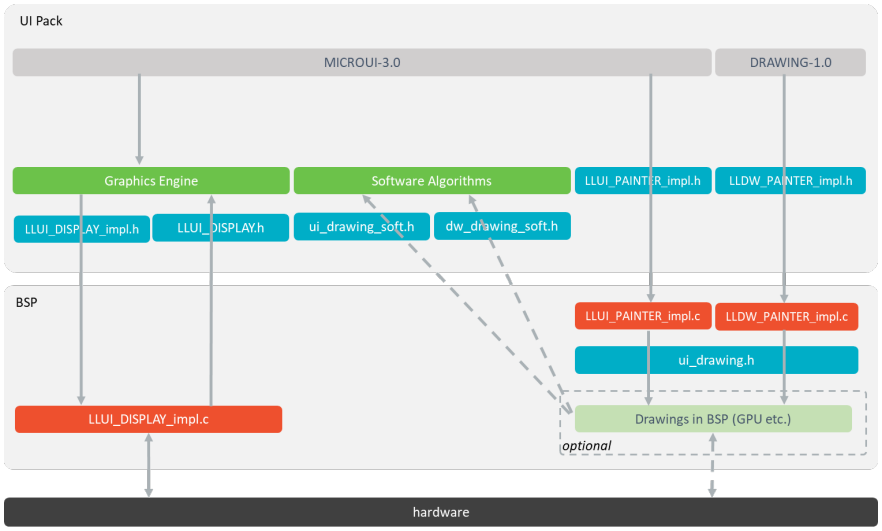


Fig. 51: Display Abstraction Layer API

- MicroUI library calls the BSP functions through the Graphics Engine and header file `LLUI_DISPLAY_impl.h`.
- Implementation of `LLUI_DISPLAY_impl.h` can call Graphics Engine functions through `LLUI_DISPLAY.h`.
- To perform some drawings, MicroUI uses `LLUI_PAINTER_impl.h` functions.
- The *MicroUI C module* provides a default implementation of the drawing native functions of `LLUI_PAINTER_impl.h` and `LLDW_PAINTER_impl.h`:
  - It implements the synchronization layer, then redirects drawings implementations to `ui_drawing.h`.
  - `ui_drawing.h` is already implemented by built-in software algorithms (library provided by the UI Pack).
  - It is possible to implement some of the `ui_drawing.h` functions in the BSP to provide a custom implementation (for instance, a GPU).
  - Custom implementation is still allowed to call software algorithms declared in `ui_drawing_soft.h` and `dw_drawing_soft.h`.

Display Size

The Abstraction Layer distinguishes between the display *virtual* size and the display *physical* size (in pixels).

- The display *virtual* size is the size of the area where the drawings are visible. Virtual memory size is: `lcd_width * lcd_height * bpp / 8`.
- The display *physical* size is the required memory size where the *virtual* area is located. On some devices, the memory width (in pixels) is higher than the virtual width. In this way, the graphics buffer memory size is: `memory_width * memory_height * bpp / 8`.

Note: The *physical* size may not be configured; in that case, the Graphics Engine considers the *virtual* size os *physical* size.

Semaphores

The Graphics Engine requires two binary semaphores to synchronize its internal states. These semaphores are reserved for the Graphics Engine. The `LLUI_DISPLAY_impl.h` implementation is not allowed to use these semaphores to synchronize the function `LLUI_DISPLAY_IMPL_flush()` with the display driver (or for any other synchronization actions). The implementation must create its semaphores in addition to these dedicated Graphics Engine's semaphores.

The binary semaphores must be configured in a state such that the semaphore must first be *given* before it can be *taken* (this initialization must be performed in `LLUI_DISPLAY_IMPL_initialize` function).

Required Abstraction Layer API

Four Abstraction Layer APIs are required to connect the Graphics Engine to the display driver. The functions are listed in `LLUI_DISPLAY_impl.h`.

- `LLUI_DISPLAY_IMPL_initialize` : The initialization function is called when the application is calling MicroUI.start(). Before this call, the display is useless and 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 the two binary semaphores, the back buffer address (see *Display Configurations*), the display *virtual* size in pixels (`lcd_width` and `lcd_height`) and optionally the display *physical* size in pixels (`memory_width` and `memory_height`).
- `LLUI_DISPLAY_IMPL_binarySemaphoreTake` and `LLUI_DISPLAY_IMPL_binarySemaphoreGive` : Two distinct functions have to be implemented to *take* and *give* a binary semaphore.
- `LLUI_DISPLAY_IMPL_flush` : According 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 Abstraction Layer API

Several optional Abstraction Layer 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 Abstraction Layer 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 principle of implementing a MicroUI drawing function is described in the chapter *Drawings*.

Graphics Engine API

The Graphics Engine provides a set of functions to interact with the C archive. The functions allow 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`.

Typical Implementations

This chapter helps to write some basic `LLUI_DISPLAY_impl.h` implementations according the display buffer mode (see *Display Configurations*). The pseudo-code calls external function such as `LCD_DRIVER_xxx` or `DMA_DRIVER_xxx` to symbolize the use of external drivers.

Note: The pseudo code does not use the dirty area bounds (xmin, ymax, etc.) to simplify the reading.

Common Functions

The three functions `LLUI_DISPLAY_IMPL_initialize`, `LLUI_DISPLAY_IMPL_binarySemaphoreTake` and `LLUI_DISPLAY_IMPL_binarySemaphoreGive` are often the same. The following example shows an implementation over FreeRTOS.

```
void LLUI_DISPLAY_IMPL_initialize(LLUI_DISPLAY_SInitData* init_data)
{
   // create the Graphics Engine's binary semaphores
   g_sem_copyLaunch = xSemaphoreCreateBinary();
   g_sem_taskTest = xSemaphoreCreateBinary();

   // fill the LLUI_DISPLAY_SInitData structure
   init_data->binary_semaphore_0 = (void*)xSemaphoreCreateBinary();
   init_data->binary_semaphore_1 = (void*)xSemaphoreCreateBinary();
   init_data->lcd_width = LCD_DRIVER_get_width();
   init_data->lcd_height = LCD_DRIVER_get_height();
}

void LLUI_DISPLAY_IMPL_binarySemaphoreTake(void* sem)
{
   xSemaphoreTake((xSemaphoreHandle)sem, portMAX_DELAY);
}

void LLUI_DISPLAY_IMPL_binarySemaphoreGive(void* sem, bool under_isr)
{
   if (under_isr)
   {
      portBASE_TYPE xHigherPriorityTaskWoken = pdFALSE;
      xSemaphoreGiveFromISR((xSemaphoreHandle)sem, &xHigherPriorityTaskWoken);
      if(xHigherPriorityTaskWoken != pdFALSE )
      {
         // Force a context switch here.
         portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
      }
```

(continues on next page)

(continued from previous page)

```
   }
   else
   {
      xSemaphoreGive((xSemaphoreHandle)sem);
   }
}
```

Direct Mode

This mode considers the application and the LCD driver share the same buffer. In other words, all drawings made by the application are immediately shown on the display. This particular case is the easiest to write because the `flush()` stays empty:

```
void LLUI_DISPLAY_IMPL_initialize(LLUI_DISPLAY_SInitData* init_data)
{
   // [...]

   // use same buffer between the LCD driver and the Graphics Engine
   LCD_DRIVER_initialize(lcd_buffer);
   init_data->back_buffer_address = lcd_buffer;
}

uint8_t* LLUI_DISPLAY_IMPL_flush(MICROUI_GraphicsContext* gc, uint8_t* srcAddr, uint32_t␣
↪xmin, uint32_t ymin, uint32_t xmax, uint32_t ymax)
{
   // nothing to send to the LCD, just have to return the same buffer address
   return srcAddr;
}
```

Serial Display

A display connected to the CPU through a serial bus (I2C, SPI, etc.) requires the *copy* mode: the application uses a buffer to perform its drawings and the buffer's content has to be sent to the display when the Graphics Engine is calling the `flush()` function.

The specification of the `flush()` function is to be **not** blocker (atomic). Its aim is to prepare / configure the serial bus and data to send and then, to start the asynchronous copy (data sent). The `flush()` function has to return as soon as possible.

Before executing the next application drawing after a flush, the Graphics Engine automatically waits the end of the serial data sent: the drawing buffer (currently used by the serial device) is not updated until the end of data sent. The serial device driver has the responsibility to unlock the Graphics Engine by calling the function `LLUI_DISPLAY_flushDone()` at the end of the copy.

There are two use cases:

Hardware

The serial data sent is performed in hardware. In that case, the serial driver must configure an interrupt to be notified about the end of the copy.

```
void LLUI_DISPLAY_IMPL_initialize(LLUI_DISPLAY_SInitData* init_data)
{
   // [...]

   LCD_DRIVER_initialize();
   init_data->back_buffer_address = back_buffer;

   // initialize the serial driver & device: GPIO, etc.
   SERIAL_DRIVER_initialize();
}

uint8_t* LLUI_DISPLAY_IMPL_flush(MICROUI_GraphicsContext* gc, uint8_t* srcAddr, uint32_t␣
↪xmin, uint32_t ymin, uint32_t xmax, uint32_t ymax)
{
   // configure the serial device to send n bytes
   // srcAddr == back_buffer
   SERIAL_DRIVER_prepare_sent(srcAddr, LCD_WIDTH * LCD_HEIGHT * LCD_BPP / 8);

   // configure the "end of copy" interrupt
   SERIAL_DRIVER_enable_interrupt(END_OF_COPY);

   // start the copy
   SERIAL_DRIVER_start();

   // return the same buffer address for next drawings
   return srcAddr;
}

void SERIAL_DEVICE_IRQHandler(void)
{
   SERIAL_DRIVER_clear_interrupt();
   SERIAL_DRIVER_disable_interrupt(END_OF_COPY);

   // end of copy, unlock the Graphics Engine
   LLUI_DISPLAY_flushDone(true); // true: called under interrupt
}
```

Software

The copy (serial data sent) cannot be performed in hardware or require a software loop to send all data. This sent must not be performed in the `flush()` function. A dedicated OS task is required to perform this sent.

```
static void* _copy_task_semaphore;

static void _task_flush(void *p_arg)
{
   while(1)
   {
      // wait until the Graphics Engine gives the order to copy
      LLUI_DISPLAY_IMPL_binarySemaphoreTake(_copy_task_semaphore);

      // send data
      SERIAL_DRIVER_send_data(back_buffer, LCD_WIDTH * LCD_HEIGHT * LCD_BPP / 8);
```

(continues on next page)

(continued from previous page)

```
      // end of copy, unlock the Graphics Engine
      LLUI_DISPLAY_flushDone(false); // true: called outside interrupt
   }
}

void LLUI_DISPLAY_IMPL_initialize(LLUI_DISPLAY_SInitData* init_data)
{
   // [...]

   LCD_DRIVER_initialize();
   init_data->back_buffer_address = back_buffer;

   // create a "flush" task and a dedicated semaphore
   _copy_task_semaphore = (void*)xSemaphoreCreateBinary();
   xTaskCreate(_task_flush, "FlushTask", 1024, NULL, 12, NULL);
}

uint8_t* LLUI_DISPLAY_IMPL_flush(MICROUI_GraphicsContext* gc, uint8_t* srcAddr, uint32_t␣
↪xmin, uint32_t ymin, uint32_t xmax, uint32_t ymax)
{
   // unlock the copy task
   LLUI_DISPLAY_IMPL_binarySemaphoreGive(_copy_task_semaphore, false);

   // return the same buffer address for next drawings
   return srcAddr;
}
```

Parallel Display: Copy Mode (Tearing Disabled)

Note: This mode should synchronize the copy buffer process with the LCD tearing signal. However, this notion is sometimes not available. This chapter describes the copy buffer process without using the tearing signal (see *next chapter*).

This kind of configuration requires two buffers in RAM. The first buffer is used by the application (back buffer) and the second buffer is used by the LCD controller to send data to the display (frame buffer).

The content of the frame buffer must be updated with the content of the back buffer when the Graphics Engine is calling the `flush()` function.

The specification of the `flush()` function is to be **not** blocker (atomic). Its aim is to prepare / configure the copy buffer process and then, to start the asynchronous copy. The `flush()` function has to return as soon as possible.

Before executing the next application drawing after a flush, the Graphics Engine automatically waits the end of the copy buffer process: the back buffer (currently used by the copy buffer process) is not updated until the end of the copy. The copy driver has the responsibility to unlock the Graphics Engine by calling the function `LLUI_DISPLAY_flushDone()` at the end of the copy.

There are two use cases:

Hardware

The copy buffer process is performed in hardware (DMA). In that case, the DMA driver must configure an interrupt to be notified about the end of the copy.

```
void LLUI_DISPLAY_IMPL_initialize(LLUI_DISPLAY_SInitData* init_data)
{
   // [...]

   // use two distinct buffers between the LCD driver and the Graphics Engine
   LCD_DRIVER_initialize(frame_buffer);
   init_data->back_buffer_address = back_buffer;

   // initialize the DMA driver: GPIO, etc.
   DMA_DRIVER_initialize();
}

uint8_t* LLUI_DISPLAY_IMPL_flush(MICROUI_GraphicsContext* gc, uint8_t* srcAddr, uint32_t␣
↪xmin, uint32_t ymin, uint32_t xmax, uint32_t ymax)
{
   // configure the DMA to send n bytes
   // srcAddr == back_buffer
   DMA_DRIVER_prepare_sent(frame_buffer, srcAddr, LCD_WIDTH * LCD_HEIGHT * LCD_BPP / 8); //␣
↪dest / src / size

   // configure the "end of copy" interrupt
   DMA_DRIVER_enable_interrupt(END_OF_COPY);

   // start the copy
   DMA_DRIVER_start();

   // return the same buffer address for next drawings
   return srcAddr;
}

void DMA_IRQHandler(void)
{
   DMA_DRIVER_clear_interrupt();
   DMA_DRIVER_disable_interrupt(END_OF_COPY);

   // end of copy, unlock the Graphics Engine
   LLUI_DISPLAY_flushDone(true); // true: called under interrupt
}
```

Software

The copy buffer process cannot be performed in hardware or require a software loop to send all data (DMA linked list). This copy buffer process must not be performed in the `flush()` function. A dedicated OS task is required to perform this copy.

```
static void* _copy_task_semaphore;

static void _task_flush(void *p_arg)
{
   while(1)
   {
```

(continues on next page)

(continued from previous page)

```
      int32_t size = LCD_WIDTH * LCD_HEIGHT * LCD_BPP / 8;
      uint8_t* dest = frame_buffer;
      uint8_t* src = back_buffer;

      // wait until the Graphics Engine gives the order to copy
      LLUI_DISPLAY_IMPL_binarySemaphoreTake(_copy_task_semaphore);

      // copy data
      while(size)
      {
         int32_t s = min(DMA_MAX_SIZE, size);
         DMA_DRIVER_send_data(dest, src, s); // dest / src / size
         dest += s;
         src += s;
         size -= s;
      }

      // end of copy, unlock the Graphics Engine
      LLUI_DISPLAY_flushDone(false); // true: called outside interrupt
   }
}

void LLUI_DISPLAY_IMPL_initialize(LLUI_DISPLAY_SInitData* init_data)
{
   // [...]

   // use two distinct buffers between the LCD driver and the Graphics Engine
   LCD_DRIVER_initialize(frame_buffer);
   init_data->back_buffer_address = back_buffer;

   // create a "flush" task and a dedicated semaphore
   _copy_task_semaphore = (void*)xSemaphoreCreateBinary();
   xTaskCreate(_task_flush, "FlushTask", 1024, NULL, 12, NULL);
}

uint8_t* LLUI_DISPLAY_IMPL_flush(MICROUI_GraphicsContext* gc, uint8_t* srcAddr, uint32_t␣
↪xmin, uint32_t ymin, uint32_t xmax, uint32_t ymax)
{
   // unlock the copy task
   LLUI_DISPLAY_IMPL_binarySemaphoreGive(_copy_task_semaphore, false);

   // return the same buffer address for next drawings
   return srcAddr;
}
```

Parallel Display: Copy Mode (Tearing Enabled)

The configuration is the same than previous chapter but it uses the LCD tearing signal to synchronize the LCD refresh rate with the copy buffer process.

The copy buffer process should not start during the call of `flush()` but should wait the next tearing signal to start the copy.

There are two use cases:

Hardware

```
static uint8_t _start_DMA;

void LLUI_DISPLAY_IMPL_initialize(LLUI_DISPLAY_SInitData* init_data)
{
   // [...]

   // use two distinct buffers between the LCD driver and the Graphics Engine
   LCD_DRIVER_initialize(frame_buffer);
   init_data->back_buffer_address = back_buffer;

   // enable the tearing interrupt
   _start_DMA = 0;
   TE_enable_interrupt();

   // initialize the DMA driver: GPIO, etc.
   DMA_DRIVER_initialize();
}

uint8_t* LLUI_DISPLAY_IMPL_flush(MICROUI_GraphicsContext* gc, uint8_t* srcAddr, uint32_t␣
↪xmin, uint32_t ymin, uint32_t xmax, uint32_t ymax)
{
   // configure the DMA to send n bytes
   // srcAddr == back_buffer
   DMA_DRIVER_prepare_sent(frame_buffer, srcAddr, LCD_WIDTH * LCD_HEIGHT * LCD_BPP / 8); //␣
↪dest / src / size

   // configure the "end of copy" interrupt
   DMA_DRIVER_enable_interrupt(END_OF_COPY);

   // unlock the job of the tearing interrupt
   _start_DMA = 1;

   // return the same buffer address for next drawings
   return srcAddr;
}

void TE_IRQHandler(void)
{
   TE_clear_interrupt();

   if (_start_DMA)
   {
```

(continues on next page)

(continued from previous page)

```
      _start_DMA = 0;

      // start the copy
      DMA_DRIVER_start();
   }
}

void DMA_IRQHandler(void)
{
   DMA_DRIVER_clear_interrupt();
   DMA_DRIVER_disable_interrupt(END_OF_COPY);

   // end of copy, unlock the Graphics Engine
   LLUI_DISPLAY_flushDone(true); // true: called under interrupt
}
```

Software

```
static void* _copy_task_semaphore;
static uint8_t _start_copy;

static void _task_flush(void *p_arg)
{
   while(1)
   {
      // wait until the Graphics Engine gives the order to copy
      LLUI_DISPLAY_IMPL_binarySemaphoreTake(_copy_task_semaphore);

      int32_t size = LCD_WIDTH * LCD_HEIGHT * LCD_BPP / 8;
      uint8_t* dest = frame_buffer;
      uint8_t* src = back_buffer;

      // copy data
      while(size)
      {
         int32_t s = min(DMA_MAX_SIZE, size);
         DMA_DRIVER_send_data(dest, src, s); // dest / src / size
         dest += s;
         src += s;
         size -= s;
      }

      // end of copy, unlock the Graphics Engine
      LLUI_DISPLAY_flushDone(false); // true: called outside interrupt
   }
}

void LLUI_DISPLAY_IMPL_initialize(LLUI_DISPLAY_SInitData* init_data)
{
   // [...]

   // use two distinct buffers between the LCD driver and the Graphics Engine
```

(continues on next page)

(continued from previous page)

```
   LCD_DRIVER_initialize(frame_buffer);
   init_data->back_buffer_address = back_buffer;

   // create a "flush" task and a dedicated semaphore
   _copy_task_semaphore = (void*)xSemaphoreCreateBinary();
   xTaskCreate(_task_flush, "FlushTask", 1024, NULL, 12, NULL);

   // enable the tearing interrupt
   _start_copy = 0;
   TE_enable_interrupt();
}

uint8_t* LLUI_DISPLAY_IMPL_flush(MICROUI_GraphicsContext* gc, uint8_t* srcAddr, uint32_t␣
↪xmin, uint32_t ymin, uint32_t xmax, uint32_t ymax)
{
   // unlock the job of the tearing interrupt
   _start_copy = 1;

   // return the same buffer address for next drawings
   return srcAddr;
}

void TE_IRQHandler(void)
{
   TE_clear_interrupt();

   if (_start_copy)
   {
      _start_copy = 0;

      // unlock the copy task
      LLUI_DISPLAY_IMPL_binarySemaphoreGive(_copy_task_semaphore, true);
   }
}
```

Parallel Display: Switch Mode

This kind of configuration requires two buffers in RAM. The first buffer is used by the application (buffer A) and the second buffer is used by the LCD controller to send data to the display (buffer B).

The LCD controller is reconfigured to use the buffer A when the Graphics Engine is calling the `flush()` function. A copy buffer process is required to restore the content of the application buffer (buffer B after the flush).

Before executing the next application drawing after a flush, the Graphics Engine automatically waits the end of the copy buffer process: the buffer B (currently used by the LDC controller or by the copy buffer process) is not updated until the end of the copy. The copy driver has the responsibility to unlock the Graphics Engine by calling the function `LLUI_DISPLAY_flushDone()` at the end of the copy.

There are two use cases:

Hardware

The copy buffer process is performed in hardware (DMA). In that case, the DMA driver must configure an interrupt to be notified about the end of the copy.

```
static uint8_t* buffer_A;
static uint8_t* buffer_B;

void LLUI_DISPLAY_IMPL_initialize(LLUI_DISPLAY_SInitData* init_data)
{
   // [...]

   // use two distinct buffers between the LCD driver and the Graphics Engine
   LCD_DRIVER_initialize(buffer_B);
   init_data->back_buffer_address = buffer_A;

   // initialize the DMA driver: GPIO, etc.
   DMA_DRIVER_initialize();
}

uint8_t* LLUI_DISPLAY_IMPL_flush(MICROUI_GraphicsContext* gc, uint8_t* srcAddr, uint32_t␣
↪xmin, uint32_t ymin, uint32_t xmax, uint32_t ymax)
{
   // the copy destination is the future back buffer (current frame buffer)
   uint8_t* dest = LCDC_get_address();

   // configure the DMA to send n bytes
   DMA_DRIVER_prepare_sent(dest, srcAddr, LCD_WIDTH * LCD_HEIGHT * LCD_BPP / 8); // dest /␣
↪src / size

   // configure the "end of copy" interrupt
   DMA_DRIVER_enable_interrupt(END_OF_COPY);

   // change the LCDC address (executed at next LCD refresh loop)
   LCDC_set_address(srcAddr);

   // return the new buffer address for next drawings
   return dest;
}

// only called when reloading a new LCDC address
void LCDC_RELOAD_IRQHandler(void)
{
   LCDC_DRIVER_clear_interrupt();

   // start the copy
   DMA_DRIVER_start();
}

void DMA_IRQHandler(void)
{
   DMA_DRIVER_clear_interrupt();
   DMA_DRIVER_disable_interrupt(END_OF_COPY);

   // end of copy, unlock the Graphics Engine
   LLUI_DISPLAY_flushDone(true); // true: called under interrupt
}
```

Software

The copy buffer process cannot be performed in hardware or require a software loop to send all data (DMA linked list). This copy buffer process must not be performed in the `flush()` function. A dedicated OS task is required to perform this copy.

```
static void* _copy_task_semaphore;
static uint8_t* buffer_A;
static uint8_t* buffer_B;

static void _task_flush(void *p_arg)
{
   while(1)
   {
      // wait until the Graphics Engine gives the order to copy
      LLUI_DISPLAY_IMPL_binarySemaphoreTake(_copy_task_semaphore);

      int32_t size = LCD_WIDTH * LCD_HEIGHT * LCD_BPP / 8;
      uint8_t* src = LCDC_get_address();
      uint8_t* dest = src == buffer_A ? buffer_B : buffer_A;

      // copy data
      while(size)
      {
         int32_t s = min(DMA_MAX_SIZE, size);
         DMA_DRIVER_send_data(dest, src, s);
         src += s;
         dest += s;
         size -= s;
      }

      // end of copy, unlock the Graphics Engine
      LLUI_DISPLAY_flushDone(false); // true: called outside interrupt
   }
}

void LLUI_DISPLAY_IMPL_initialize(LLUI_DISPLAY_SInitData* init_data)
{
   // [...]

   // use two distinct buffers between the LCD driver and the Graphics Engine
   LCD_DRIVER_initialize(buffer_B);
   init_data->back_buffer_address = buffer_A;

   // initialize the DMA driver: GPIO, etc.
   DMA_DRIVER_initialize();

   // create a "flush" task and a dedicated semaphore
   _copy_task_semaphore = (void*)xSemaphoreCreateBinary();
   xTaskCreate(_task_flush, "FlushTask", 1024, NULL, 12, NULL);
}

uint8_t* LLUI_DISPLAY_IMPL_flush(MICROUI_GraphicsContext* gc, uint8_t* srcAddr, uint32_t␣
↪xmin, uint32_t ymin, uint32_t xmax, uint32_t ymax)
```

(continues on next page)

(continued from previous page)

```
{
   // the copy destination is the future back buffer (current frame buffer)
   uint8_t* dest = LCDC_get_address();

   // change the LCDC address (executed at next LCD refresh loop)
   LCDC_set_address(srcAddr);

   // return the new buffer address for next drawings
   return dest;
}

// only called when reloading a new LCDC address
void LCDC_RELOAD_IRQHandler(void)
{
   LCDC_DRIVER_clear_interrupt();

   // unlock the copy task
   LLUI_DISPLAY_IMPL_binarySemaphoreGive(_copy_task_semaphore, true);
}
```

Display Synchronization

Note: This chapter is mainly helpful when the *copy* mode is used: the aim consists of synchronizing the update of the LCD frame buffer with the LCD refresh rate.

Overview

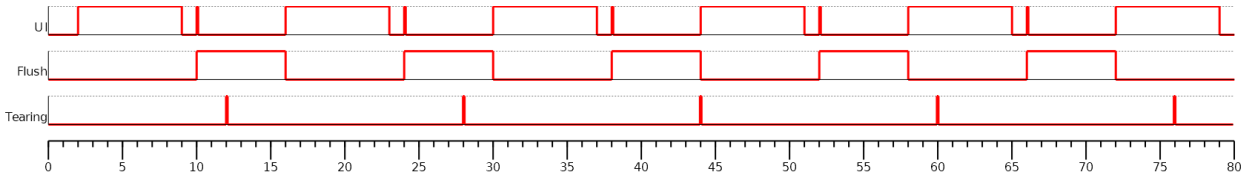
The Graphics Engine is designed to be synchronized with the display refresh rate by defining some points in the rendering timeline. It is optional; however it is mainly recommended. This chapter explains why to use display tearing signal and its consequences. Some chronograms describe several use cases: with and without display tearing signal, long drawings, long flush time, etc. Times are in milliseconds. To simplify chronograms views, the display refresh rate is every 16ms (62.5Hz).

Captions definition:

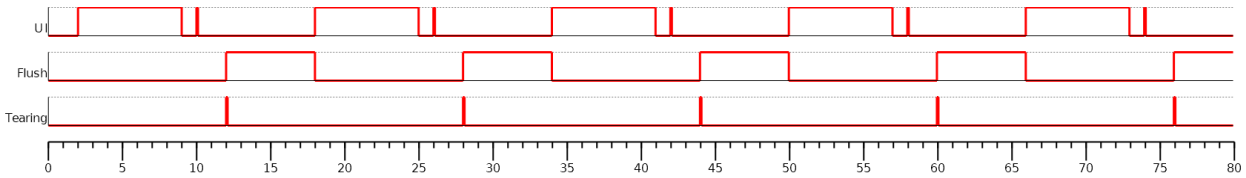
- UI: It is the UI task which performs the drawings in the back buffer. At the end of the drawings, the examples consider that the UI thread calls Display.flush() 1 millisecond after the end of the drawings. At this moment, a flush can start (the call to Display.flush() is symbolized by a simple *peak* 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

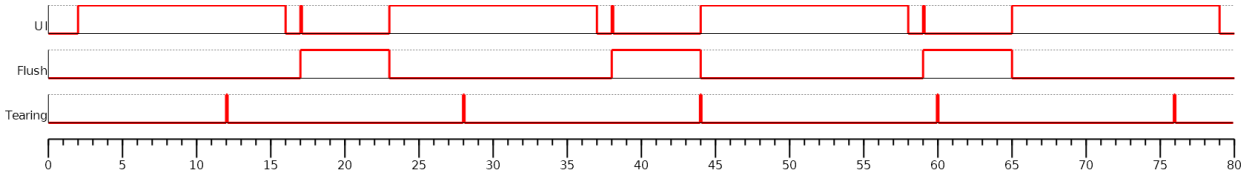
In this example, the drawing time is 7ms, the time between the end of drawing and the call to Display.flush() is 1ms and the flush time is 6ms. So the expected rendering frequency is 7 + 1 + 6 = 14ms (71.4Hz). Flush starts just after the call to Display.flush() and the next drawing starts just after 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.



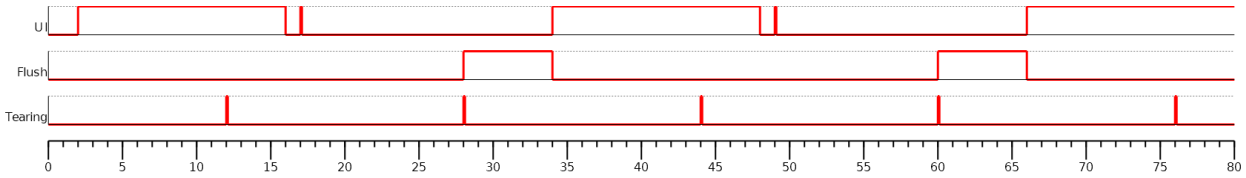
In this example, the times are identical to previous example. The tearing signal is used to start the flush in respecting the display refreshing time. The rendering frequency becomes smaller: it is cadenced on the tearing signal, every 16ms (62.5Hz). During 2ms, the CPU can schedule other tasks or goes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



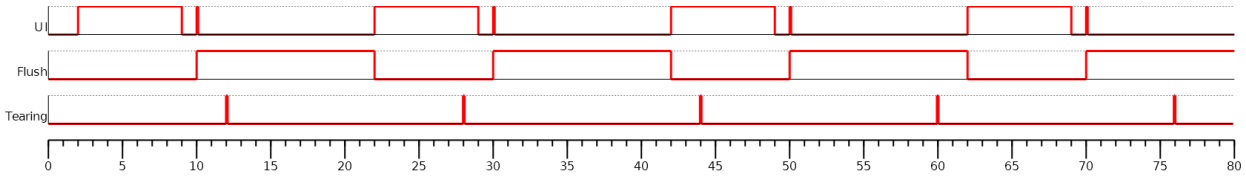
In this example, the times are identical to previous example. The tearing signal is used to start the flush in respecting the display refreshing time. The drawing time + flush time is higher than display tearing signal period. So the flush cannot start at every tearing peak: it is cadenced on two tearing 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.



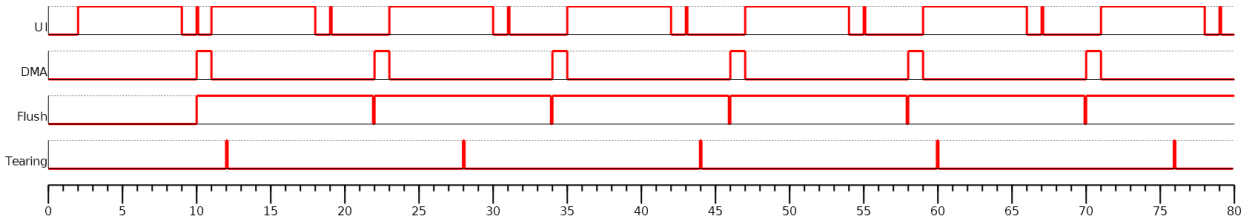
Additional Buffer

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.

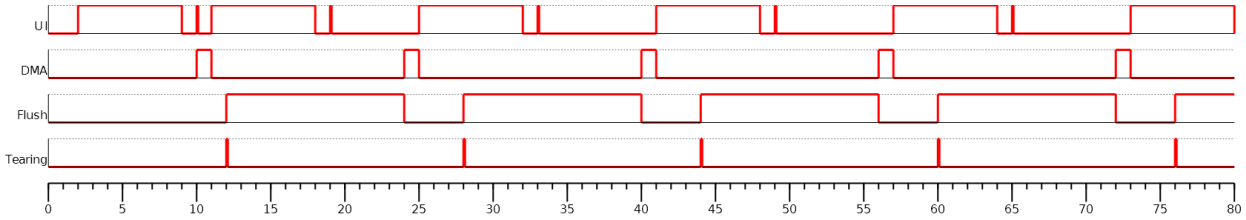
In this example, the drawing time is 7ms, the time between the end of drawing and the call to Display.flush() is 1ms and the flush time is 12ms. So the expected rendering frequency is 7 + 1 + 12 = 20ms (50Hz). Flush starts just after the call to Display.flush() and the next drawing starts just after the end of flush. Tearing signal is not taken in consideration. The rendering frequency is cadenced on drawing time + flush time.



As mentioned 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 the 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 the 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, i.e. 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 application to perform other operations during the GPU rendering. However, as soon as the application is trying to perform another drawing, the previous drawing made by the GPU must be done. The Graphics Engine is designed to be synchronized with the GPU asynchronous drawings by defining some points in the rendering timeline. It is not optional: MicroUI 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_notifyAsynchronousDrawingEnd()` .

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 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-bit ARGB colors (see *LLUI_DISPLAY: Display*) to display color representation. For each application ARGB8888 color, the display driver has to find the corresponding color in the table. The Graphics Engine will store the index of the color in the table instead of using the color itself.

When an application color is not available in the display driver table (CLUT), the display driver can try to find the closest color or return a default color. 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 Abstraction Layer API `LLUI_DISPLAY_IMPL_prepareBlendingOfIndexedColors(void* foreground, void* background)`.

This API is only used when a blending is required and when the background color is enabled. The Graphics Engine calls the API just before the blending and gives as 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 VEE Port designer should give to the application developer the available list of colors the CLUT manages.
- The CLUT must provide a set of blending ranges the application can use. Each range can have its own size (different number of colors between two colors). Each range is independent. For instance if the foreground color `RED` (`0xFFFF0000`) can be blended with two background colors `WHITE` (`0xFFFFFFFF`) and `BLACK` (`0xFF000000`), two ranges must be provided. Both the ranges have to contain the same index for the color `RED`.
- Application can only use blending ranges provided by the CLUT. Otherwise the display driver is not able to find the range and the default color will be used to perform the blending.
- Rendering of dynamic images (images decoded at runtime) may be wrong because the ARGB colors may be out of 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 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 VEE Port can be built with a reduced set of pixel support.

All pixel format manipulations (read, write, copy) are using dedicated functions. It is possible to remove some functions or to use generic functions. The advantage is to reduce the memory footprint. The inconvenient is that some features are removed (the application should not use them) or some features are slower (generic functions are slower than the dedicated functions).

Functions

There are five pixel *conversion* modes:

- Draw an image without transformation and without global alpha blending: copy a pixel from a format to the destination format (display format).
- Draw an image without transformation and with global alpha blending: copy a pixel with alpha blending from a format to the destination format (display format).
- Draw an image with transformation and with or without alpha blending: draw an ARGB8888 pixel in destination format (display format).
- Load a ResourceImage with an output format: convert an ARGB8888 pixel to the output format.
- Read a pixel from an image (Image.readPixel() or to draw an image with transformation or to convert an image): read any pixel formats and convert it in ARGB8888.

Table 20: Pixel Conversion

	Nb input formats	Nb output formats	Number of combinations
Draw image without global alpha	22	1	22
Draw image with global alpha	22	1	22
Draw image with transformation	2	1	2
Load a `ResourceImage`	1	6	6
Read an image	22	1	22

There are `22x1 + 22x1 + 2x1 + 1x6 + 22x1 = 74` functions. Each function takes between 50 and 200 bytes depending on its complexity and the C compiler.

Linker File

All pixel functions are listed in a VEE Port linker file. It is possible to edit this file to remove some features or to share some functions (using generic function).

How to get the file:

1. Build VEE Port as usual.
2. Copy VEE Port file `[platform]/source/link/display_image_x.lscf` in VEE Port configuration project: `[VEE Port 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 VEE Port: the *dropins* file is copied in the VEE Port instead of the original one.

Warning: When the display format in `[VEE Port 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 Foundation Library provides additional drawing APIs. This library is fully integrated in Display module. It requires an implementation of its Abstraction Layer API: `LLDW_PAINTER_impl.h`. These functions are implemented in the Abstraction Layer implementation module com.microej.clibrary.llimpl#microui. 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*).
- The *MicroUI C module*.

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 VEE Port. If not installed, the *stub* module will be used.

In the VEE Port configuration file, check `UI` > `Display` to install the Display module. When checked, the properties file `display/display.properties` is required during VEE Port 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 *GPU Format Support* and *Customize MicroEJ Standard 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 VEE Port heap size when building a firmware in command line. When using a MicroEJ launcher, the size set in this launcher has priority over the VEE Port value.

Use

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

6.14.9 Drawings

Abstraction Layer

All MicroUI drawings (available in the Painter class) call a native function. These native functions are already implemented (in the *MicroUI C Module* for the Embedded VEE Port and in the *Front Panel* for the Simulator). These implementations use the Graphics Engine's software algorithms to perform the drawings.

Each drawing can be overwritten independently in the VEE Port:

- to use another software algorithm (custom algorithm, no third-party library, etc.),
- to use a GPU to perform the operation,
- to target a destination whose format is different from the display buffer format,
- etc.

The MicroUI native drawing functions are listed in `LLUI_PAINTER_impl.h` and `LLDW_PAINTER_impl.h` (for the Drawing library) for the Embedded VEE Port and, respectively, `LUIPainter.java` and `LLDWPainter.java` for the Simulation VEE Port.

The implementation must handle many constraints: synchronization between drawings, Graphics Engine notification, MicroUI GraphicsContext clip and colors, dirty flush area, etc. The principle of implementing a MicroUI drawing function is described in the chapter *Custom Drawing*.

Destination Format

Since MicroUI 3.2, the destination buffer of the drawings can be different than the display buffer format (see *MicroEJ Format: Display*). This destination buffer format can be a *standard format* (ARGB8888, A8, etc.) or a *custom format*.

See *Buffered Image* for more information about how to create buffered images with another format than the display format and how to draw in them.

Graphics Engine Software Algorithms

The Graphics Engine features a software implementation for each MicroUI and Drawing libraries drawing. These software algorithms:

- respect the MicroUI GraphicsContext clip,
- use the current MicroUI GraphicsContext foreground color and optional background color,
- update the next Display.flush() dirty area.

The Graphics Engine provides a header file `ui_drawing_soft.h` (emb), and an implementation instance of `UIDrawing` that can be retrieved with `ej.microui.display.LLUIDisplay.getUIDrawerSoftware()` (sim) to let the VEE Port use these algorithms. For instance, a GPU may be able to draw an image whose format is RGB565 but not ARGB1555. For this image format, BSP implementation can call the `UI_DRAWING_SOFT_drawImage` function.

Warning: These software algorithms only target buffers whose format is the display buffer format.

MicroUI C Module

Principle

An implementation of `LLUI_PAINTER_impl.h` is already available on the *MicroUI C module*. This implementation respects the synchronization between drawings and the Graphics Engine notification, reduces (when possible) the MicroUI GraphicsContext clip constraints, and updates (when possible) the dirty flush area.

This implementation does not perform the drawings; it only calls the equivalent of drawing available in `ui_drawing.h`. This allows simplifying how to use a GPU (or a third-party library) to perform a drawing: the `ui_drawing.h` implementation just has to take into consideration the MicroUI GraphicsContext clip and colors and Display.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 *MicroUI C module* (in *weak* mode). This allows to implement only the functions the GPU can perform. For a given drawing, the weak function implementation is calling the equivalent of the drawing available in `ui_drawing_soft.h` (this file lists all drawing functions implemented by the Graphics Engine in software).

Note: More details are available in `LLUI_PAINTER_impl.h`, `ui_drawing.h`, `LLUI_Display.h`, and `LLUI_Display_impl.h` classes.

Default Implementation

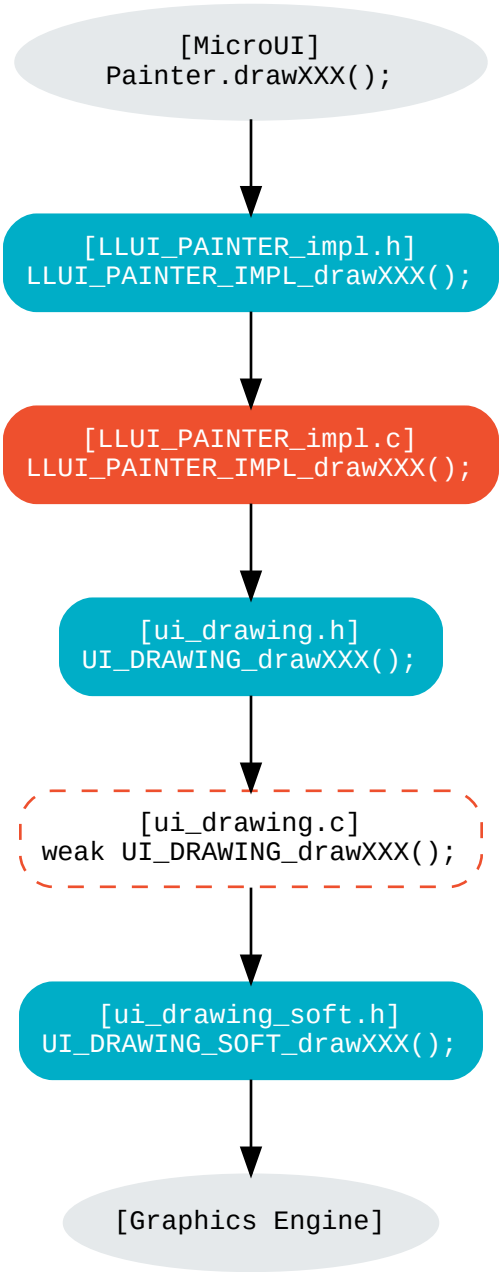
The default implementation is the most used. It takes into account:

- there is only one destination format (the display buffer format),
- no drawing is overwritten in the BSP (no GPU, third-party library, etc.),
- *non-standard images* cannot be used as a source.

The *MicroUI C module* is designed to simplify the UI VEE Port:

- just need to add the C module in the BSP (no extra code is needed),
- flash footprint is reduced (no extra table to manage several destination formats and several sources),
- functions indirections are limited (the software drawing algorithm is called as faster as possible).

The following graph illustrates the steps to perform a shape drawing (not an image):



LLUI_PAINTER_IMPL_drawLine (available in MicroUI C Module)

```
void LLUI_PAINTER_IMPL_drawLine(MICROUI_GraphicsContext* gc, jint startX, jint startY, jint␣
↪endX, jint endY) {
   // Synchronize the native function of MicroUI Painter.drawLine() with the Graphics Engine
   if (LLUI_DISPLAY_requestDrawing(gc, (SNI_callback)&LLUI_PAINTER_IMPL_drawLine)) {
      // Call ui_drawing.h function
      DRAWING_Status status = UI_DRAWING_drawLine(gc, startX, startY, endX, endY);
      // Update the status of the Graphics Engine
      LLUI_DISPLAY_setDrawingStatus(status);
   }
}
```

The Graphics Engine requires synchronization between the drawings. Doing that requires a call to `LLUI_DISPLAY_requestDrawing` at the beginning of native function implementation. This function takes as a parameter the MicroUI GraphicsContext and the pointer on the native function itself. This pointer must be cast in a `SNI_callback`.

UI_DRAWING_drawLine (available in MicroUI C Module)

```
#define UI_DRAWING_DEFAULT_drawLine UI_DRAWING_drawLine
```

The function name is set thanks to a `define`. This name redirection is useful when the VEE Port features multiple destination formats (not the use-case here).

UI_DRAWING_DEFAULT_drawLine (available in MicroUI C Module)

```
// Use the preprocessor 'weak'
__weak DRAWING_Status UI_DRAWING_DEFAULT_drawLine(MICROUI_GraphicsContext* gc, jint startX,␣
↪jint startY, jint endX, jint endY) {
   // Default behavior: call the Graphics Engine's software algorithm
   return UI_DRAWING_SOFT_drawLine(gc, startX, startY, endX, endY);
}
```

Implementing the weak function only consists in calling the Graphics Engine's software algorithm. This software algorithm will respect the GraphicsContext color and clip and update the Display.flush() dirty area.

Custom Implementation

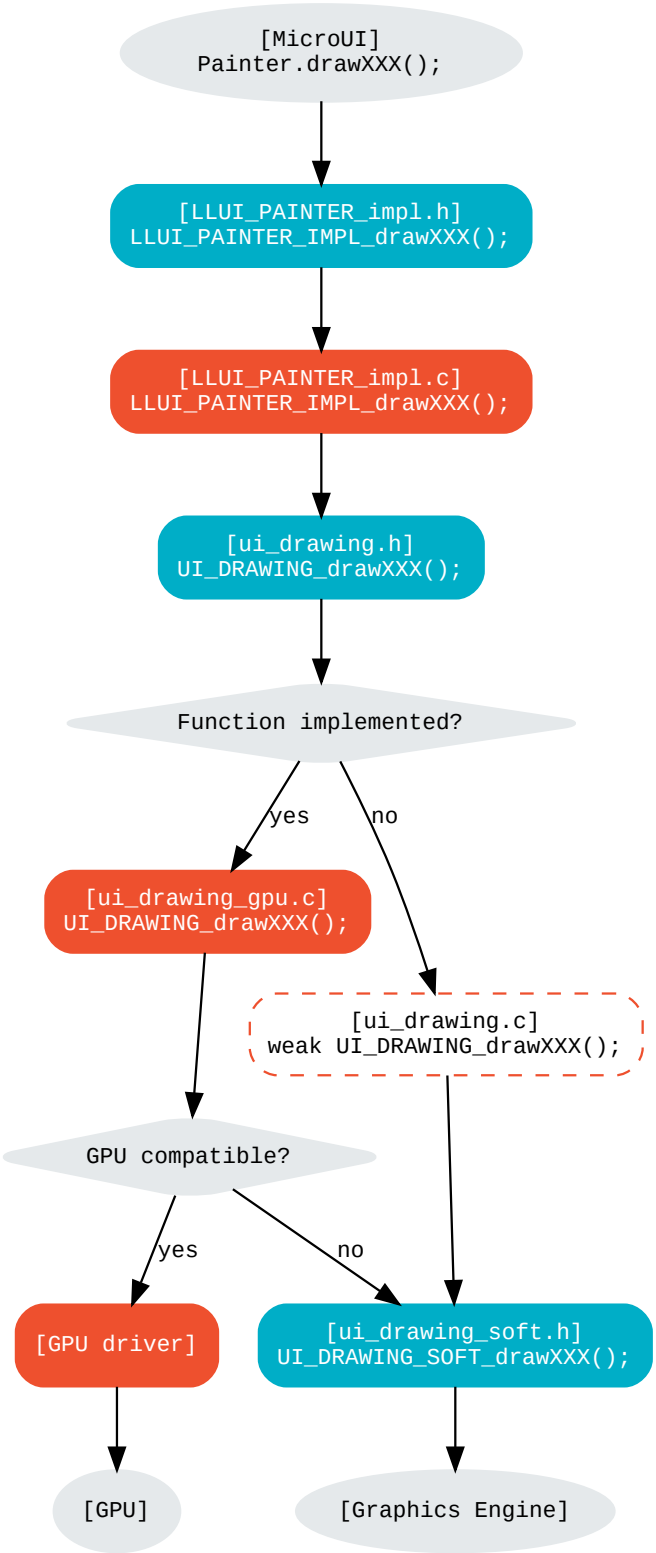
The custom implementation helps connect a GPU or a third-party library. It takes into account:

- there is only one destination format (the display buffer format),
- *non-standard images* cannot be used as a source.

The *MicroUI C module* is designed to simplify the adding of third-party drawers:

- just need to add the C module in the BSP,
- overwrite only the expected drawing(s),
- a drawing implementation has just to respect the clip and color (synchronization with the Graphics Engine already done),
- flash footprint is reduced (no extra table to manage several destination formats and several sources),
- functions indirections are limited (the drawing algorithm is called as faster as possible).

The following graph illustrates the steps to perform a shape drawing (not an image):


[MicroUI]
Painter.drawXXX();
[LLUI_PAINTER_impl.h]
LLUI_PAINTER_IMPL_drawXXX();
[LLUI_PAINTER_impl.c]
LLUI_PAINTER_IMPL_drawXXX();
[ui_drawing.h]
UI_DRAWING_drawXXX();
Function implemented?
yes
no
[ui_drawing_gpu.c]
UI_DRAWING_drawXXX();
[ui_drawing.c]
weak UI_DRAWING_drawXXX();
GPU compatible?
yes
no
[GPU driver]
[ui_drawing_soft.h]
UI_DRAWING_SOFT_drawXXX();
[GPU]
[Graphics Engine]

Take the same example as the default implementation (draw a line): the BSP just has to overwrite the weak function `UI_DRAWING_drawLine` :

UI_DRAWING_drawLine (to write in the BSP)

```
#define UI_DRAWING_GPU_drawLine UI_DRAWING_drawLine
```

The function name should be set thanks to a `define` . This name redirection is useful when the VEE Port features multiple destination formats (not the use-case here).

UI_DRAWING_GPU_drawLine (to write in the BSP)

```
// Contrary to the MicroUI C Module, this function is not "weak"
DRAWING_Status UI_DRAWING_GPU_drawLine(MICROUI_GraphicsContext* gc, jint startX, jint startY,
↪ jint endX, jint endY) {

   DRAWING_Status status;

   if (is_gpu_compatible(xxx)) {
      // Can use the GPU to draw the line

      // Retrieve the destination buffer address
      uint8_t* destination_address = LLUI_DISPLAY_getBufferAddress(&gc->image);

      // Update the next "flush"'s dirty area
     LLUI_DISPLAY_setDrawingLimits(startX, startY, endX, endY);

      // Configure the GPU clip
      gpu_set_clip(startX, startY, endX, endY);

      // Draw the line
      gpu_draw_line(destination_address, startX, startY, endX, endY, gc->foreground_color);

      // GPU is running: set the right status for the Graphics Engine
      status = DRAWING_RUNNING;
   }
   else {
      // Default behavior: call the Graphics Engine's software algorithm (like "weak"␣
↪function)
      status = UI_DRAWING_SOFT_drawLine(gc, startX, startY, endX, endY);
   }
   return status;
}
```

First, the drawing function must ensure the GPU can render the expected drawing. If not, the drawing function must perform the same thing as the default weak function: calls the Graphics Engine software algorithm.

The GPU drawing function usually requires the destination buffer address: the drawing function calls `LLUI_DISPLAY_getBufferAddress(&gc->image);` .

The drawing function must update the next Display.flush() area (dirty area) by calling `LLUI_DISPLAY_setDrawingLimits()` .

The drawing function has to respect the GraphicsContext clip The `MICROUI_GraphicsContext` structure holds

the clip, and the drawer cannot perform a drawing outside this clip (otherwise, the behavior is unknown). Note the bottom-right coordinates might be smaller than the top-left (in x and/or y) when the clip width and/or height is null. The clip may be disabled (when the current drawing fits the clip); this allows to reduce runtime. See `LLUI_DISPLAY_isClipEnabled()`.

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

Finally, after the drawing, the drawing function has to return the drawing status. Most of the time, the GPU performs *asynchronous* drawings: the drawing is started but not completed. To notify the Graphics Engine, the status to return is `DRAWING_RUNNING`. In case of the drawing is done after the call to `gpu_draw_line()`, the status to return is `DRAWING_DONE`.

Warning:

- If the update of the dirty area is not performed, the next call to Display.flush() will not call the `LLUI_DISPLAY_IMPL_flush()` function.
- If the drawing status is not set to the Graphics Engine, the global VEE execution is locked: the Graphics Engine waits indefinitely for the status and cannot perform the next drawing.
- In case of the drawing is *asynchronous*, the GPU interrupt routine (or an OS task) has to notify the Graphics Engine of the end of the drawing by calling `LLUI_DISPLAY_notifyAsynchronousDrawingEnd`.

Extended C Modules

Several *C Modules* are available on the MicroEJ Repositories. These modules are compatible with the MicroUI C module (they follow the rules described above) and use one GPU (a C Module per GPU). These C Modules should be fetched in the VEE Port in addition to the MicroUI C Module; it avoids re-writing the GPU port.

Simulation

Principle

This is the same principle as *MicroUI C Module* for the Embedded side:

- The drawing primitive natives called the matching method in `LLUIPainter`.
- The `LLUIPainter` synchronizes the drawings with the Graphics Engine and dispatches the drawing itself to an implementation of the interface `UIDrawing`.
- The Front Panel provides a software implementation of `UIDrawing` available by calling `ej.microui.display.LLUIDisplay.getUIDrawerSoftware()`.
- The `DisplayDrawer` implements `UIDrawing` and is used to draw in the display buffer and the images with the same format.

These classes are available in the *UI Pack extension* of the Front Panel Mock.

Note: More details are available in `LLUIPainter`, `UIDrawing`, `LLUIDisplay`, and `LLUIDisplayImpl` classes.

Default Implementation

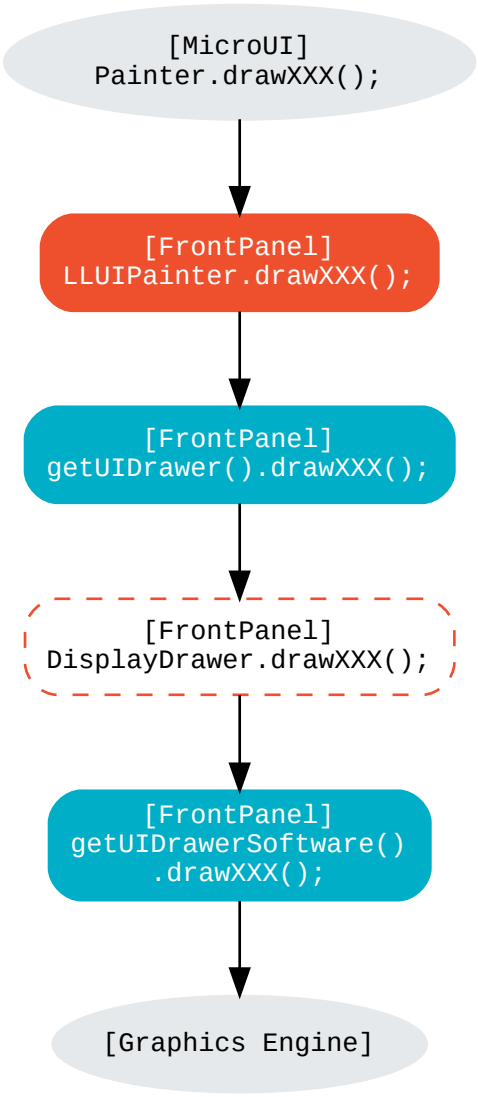
The default implementation is the most used. It considers that:

- there is only one destination format (the display buffer format),
- no drawing is overwritten in the BSP (no third-party library),
- *non-standard images* cannot be used as a source.

The *UI Pack extension* is designed to simplify the UI VEE Port:

- Simply add the dependency to the UI Pack extension in the VEE Port Front Panel project.
- Function indirections are limited (the software drawing algorithm is called as fast as possible).

The following graph illustrates the steps to perform a shape drawing (not an image):



LLUIPainter.drawLine (available in UI Pack extension)

```
public static void drawLine(byte[] target, int x1, int y1, int x2, int y2) {

   // Retrieve the Graphics Engine instance
   LLUIDisplay graphicalEngine = LLUIDisplay.Instance;
```

(continues on next page)

(continued from previous page)

```
   // Synchronize the native function of MicroUI Painter.drawLine() with the Graphics Engine
   synchronized (graphicalEngine) {

      // Retrieve the Front Panel instance of the MicroUI GraphicsContext (the destination)
      MicroUIGraphicsContext gc = graphicalEngine.mapMicroUIGraphicsContext(target);

      // Ask to the Graphics Engine if a drawing can be performed on the target
      if (gc.requestDrawing()) {

         // Retrieve the drawer for the GraphicsContext (by default: DisplayDrawer)
         UIDrawing drawer = getUIDrawer(gc);

         // Call UIDrawing function
         drawer.drawLine(gc, x1, y1, x2, y2);
      }
   }
}
```

The Graphics Engine requires synchronization between the drawings. To do that, the drawing is synchronized on the instance of the Graphics Engine itself.

The target (the Front Panel object that maps the MicroUI GraphicsContext) is retrieved in the native drawing method by asking the Graphics Engine to map the byte array (returned by `GraphicsContext.getSNIContext()`). Like the embedded side, this object holds a clip, and the drawer cannot perform a drawing outside of this clip (otherwise, the behavior is unknown).

DisplayDrawer.drawLine (available in UI Pack extension)

```
@Override
public void drawLine(MicroUIGraphicsContext gc, int x1, int y1, int x2, int y2) {
        LLUIDisplay.Instance.getUIDrawerSoftware().drawLine(gc, x1, y1, x2, y2);
}
```

The implementation of `DisplayDrawer` simply calls the Graphics Engine's software algorithm. This software algorithm will use the GraphicsContext color and clip and update the Display.flush() dirty area.

Custom Implementation

The custom implementation helps connect a third-party library or to simulate the same constraints as the embedded side (the same GPU constraints). It considers that:

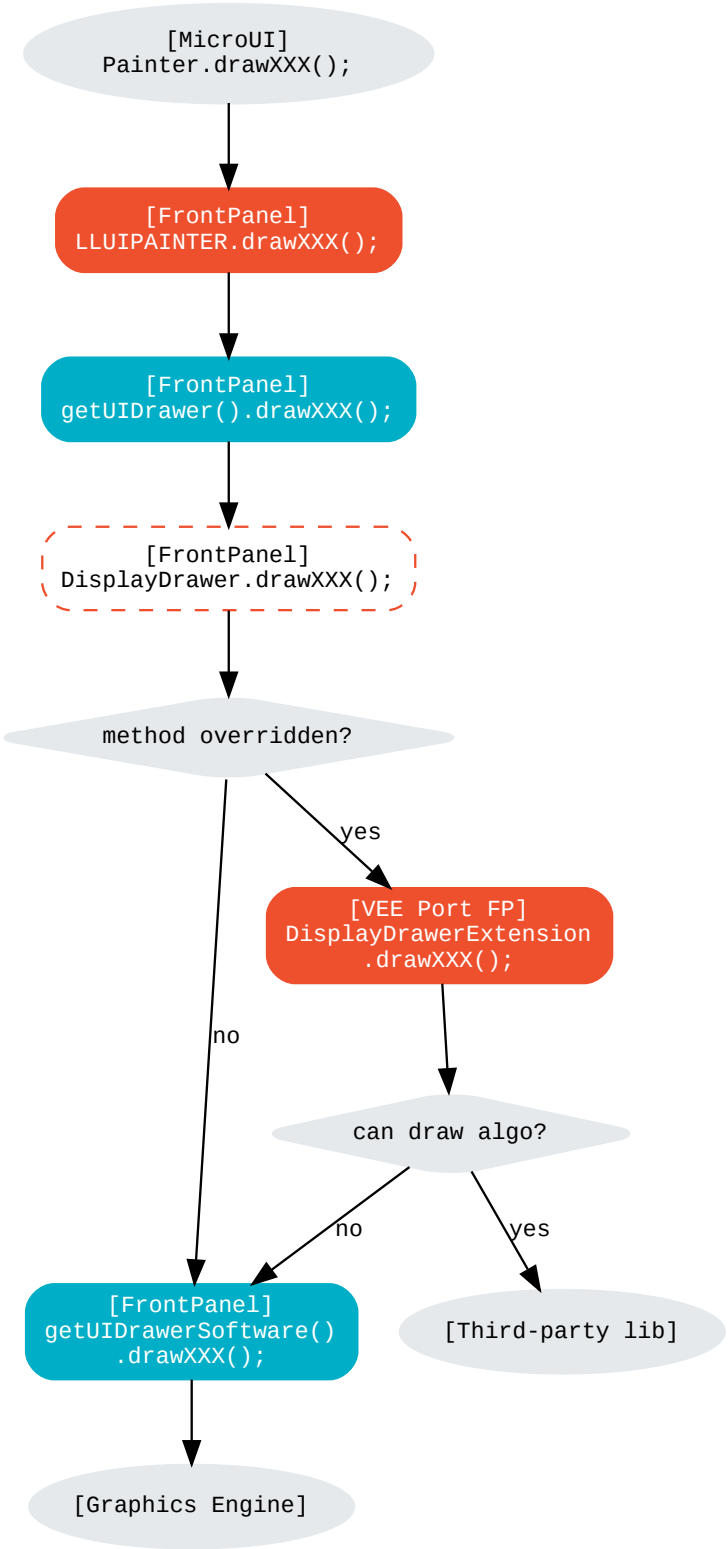
- there is only one destination format (the display buffer format),
- *non-standard images* cannot be used as a source.

The *UI Pack extension* is designed to simplify the adding of third-party drawers:

- Add the dependency to the UI Pack extension in the VEE Port Front Panel project.
- Create a subclass of `DisplayDrawer` (implementation of the interface `UIDrawing`).
- **Overwrite only the desired drawing(s).**
  - Each drawing implementation must comply with the clip and color (synchronization with the Graphics Engine already done).
  - Function indirections are limited (the drawing algorithm is called as fast as possible).

- Register this drawer in place of the default display drawer.

The following graph illustrates the steps to perform a shape drawing (not an image):


[MicroUI]
Painter.drawXXX();
[FrontPanel]
LLUIPAINTER.drawXXX();
[FrontPanel]
getUIDrawer().drawXXX();
[FrontPanel]
DisplayDrawer.drawXXX();
method overridden?
yes
no
[VEE Port FP]
DisplayDrawerExtension
.drawXXX();
can draw algo?
no
yes
[FrontPanel]
getUIDrawerSoftware()
.drawXXX();
[Third-party lib]
[Graphics Engine]

Let's use the same example as the previous section (draw line function): the Front Panel project has to create its drawer based on the default drawer:

MyDrawer (to write in the Front Panel project)

```
public class MyDrawer extends DisplayDrawer {

   @Override
   public void drawLine(MicroUIGraphicsContext gc, int x1, int y1, int x2, int y2) {

      if (isCompatible(xxx)) {
         // Can use the GPU to draw the line on the embedded side: can use another algorithm␣
↪than the software algorithm

         // Retrieve the AWT Graphics2D
         Graphics2D src = (Graphics2D)((BufferedImage)gc.getImage().getRAWImage()).
↪getGraphics();

         // Draw the line using AWT (have to respect clip & color)
         src.setColor(new Color(gc.getRenderingColor()));
         src.drawLine(x1, y1, x2, x2);

         // Update the next "flush"'s dirty area
         gc.setDrawingLimits(x1, y1, x2, x2);
      }
      else {
         // Default behavior: call the Graphics Engine's software algorithm
         super.drawLine(gc, x1, y1, x2, y2);
      }
   }
}
```

The Front Panel framework is running over AWT. The method `gc.getImage()` returns a `ej.fp.Image`. It is the representation of a MicroUI Image in the Front Panel framework. The method `gc.getImage().getRAWImage()` returns the implementation of the Front Panel image on the J2SE framework: an AWT BufferedImage. The AWT graphics 2D can be retrieved from this buffered image.

The MicroUI color (`gc.getRenderingColor()`) is converted to an AWT color. After the drawing, the implementation updates the Graphics Engine dirty area by calling `gc.setDrawingLimits()`.

The method behavior is exactly the same as the embedded side; see:ref:*section_drawings_cco_custom*.

This newly created drawer must now replace the default display drawer. There are two possible ways to register it:

- Declare it as a UIDrawing service.
- Declare it programmatically.

UIDrawing Service

- Create a new file in the resources of the Front Panel project named `META-INF/services/ej.microui.display.UIDrawing` and write the fully qualified name of the previously created drawer:

```
com.mycompany.MyDrawer
```

Programmatically

- Create an empty widget to invoke the new implementation:

```
@WidgetDescription(attributes = { })
public class Init extends Widget{
   @Override
   public void start() {
      super.start();
      LLUIDisplay.Instance.registerUIDrawer(new MyDrawer());
   }
}
```

- Invoke this widget in the .fp file:

```
<frontpanel xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="https://developer.
↪microej.com" xsi:schemaLocation="https://developer.microej.com .widget.xsd">
   <device name="STM32429IEVAL" skin="Board-480-272.png">
      <com.is2t.microej.fp.Init/>
      [...]
   </device>
</frontpanel>
```

Custom Drawing

Principle

MicroUI allows adding some custom drawings (drawings not listed in the MicroUI Painter classes). A custom drawing has to respect the same rules as the MicroUI drawings to avoid corrupting the MicroUI execution (flickering, memory corruption, unknown behavior, etc.).

As explained above, MicroUI implementation provides an Abstraction Layer that lists all MicroUI Painter drawing native functions and their implementations (*MicroUI C Module* and *Simulation*). The implementation of MicroUI Painter drawings should be used as a model to implement the custom drawings.

Application Method

```
// Application drawing method
protected void render(GraphicsContext gc) {

   // [...]

   // Set the GraphicsContext color
   gc.setColor(Colors.RED);
   // Draw a red line
   Painter.drawLine(gc, 0, 0, 10, 10);
   // Draw a red custom drawing
   drawCustom(gc.getSNIContext(), 5, 5);

   // [...]
}
```

(continues on next page)

(continued from previous page)

```
// Custom drawing native method
private static native void drawCustom(byte[] graphicsContext, int x, int y);
```

All native functions must have a MicroUI GraphicsContext as a parameter (often the first parameter) that identifies the destination target. The application retrieves this target by calling the method `GraphicsContext.getSNIContext()`. This method returns a byte array to give as-is to the drawing native method.

BSP Implementation

The native drawing function implementation pattern is:

```
void Java_com_mycompany_MyPainterClass_drawCustom(MICROUI_GraphicsContext* gc, jint x, jint␣
↪y) {

   // Tell the Graphics Engine if the drawing can be performed
   if (LLUI_DISPLAY_requestDrawing(gc, (SNI_callback)&Java_com_mycompany_MyPainterClass_
↪drawCustom)) {
      DRAWING_Status status;

      // Perform the drawing (respecting clip if not disabled)
      status = custom_drawing(LLUI_DISPLAY_getBufferAddress(&gc->image), x, y);

      // Set drawing status
      LLUI_DISPLAY_setDrawingStatus(status);
   }
   // Else: refused drawing
}
```

The target (the MicroUI GraphicsContext) is retrieved in the native drawing function by mapping the `MICROUI_GraphicsContext` structure in MicroUI native drawing function declaration.

This implementation has to follow the same rules as the custom MicroUI drawings implementation: see *Custom Implementation*.

Simulation

Note: This chapter considers the VEE Port Front Panel project already features a custom drawer that replaces the default drawer `DisplayDrawer`. See *Custom Implementation*.

The native drawing function implementation pattern is as follows (see below for the explanations):

```
public static void drawCustom(byte[] target, int x, int y) {

   // Retrieve the Graphics Engine instance
   LLUIDisplay graphicalEngine = LLUIDisplay.Instance;

   // Synchronize the native function with the Graphics Engine
   synchronized (graphicalEngine) {
```

(continues on next page)

(continued from previous page)

```
      // Retrieve the Front Panel instance of the MicroUI GraphicsContext (the destination)
      MicroUIGraphicsContext gc = graphicalEngine.mapMicroUIGraphicsContext(target);

      // Ask to the Graphics Engine if a drawing can be performed on the target
      if (gc.requestDrawing()) {

         // Retrieve the drawer for the GraphicsContext (by default: DisplayDrawer)
         UIDrawing drawer = getUIDrawer(gc);

         // Call UIDrawing function
         MyDrawer.Instance.drawSomething(gc, x, y);
      }
   }
}
```

This implementation has to follow the same rules as the custom MicroUI drawings implementation: see *Custom Implementation*.

Drawing Logs

When performing drawing operations, the program may fail or encounter an incident of some kind. MicroUI offers a mechanism allowing the VEE Port to report such incidents to the application through the use of flags.

Usage Overview

When an incident occurs, the VEE Port can report it to the application by setting the *drawing log flags* stored in the graphics context. The flags will then be made available to the application. See *Drawing Logs* for more information on reading the flags in the application.

Without an intervention from the application, the drawing log flags retain their values through every call to drawing functions and are cleared when a flush is performed.

Note: The clearing of drawing log flags can be disabled at build time by the application developer.

Incidents are split into two categories:

- *Non-critical* incidents, or *warnings*, are incidents that the application developer may ignore. The flags are made available for the application to check, but without an explicit statement in the application, these incidents will be ignored silently.
- *Critical* incidents, or *errors*, are failures significant enough that the application developer should not ignore them. As for warnings, the application may check the drawing log flags explicitly. However, when flushing the display, the application checks the flags and throws an exception if an error has been reported.

Warning: As this behavior can be disabled at build time, the drawing log flags are meant to be used as a **debugging hint** when the application does not display what the developer expects. The VEE Port must **not** rely on applications throwing an exception if an error was reported or on the drawing log flags being reset after the display is flushed.

Note: Any incident may be either a *warning* or an *error*. They are differentiated with the special flag `DRAWING_LOG_ERROR`.

Available Constants

MicroUI offers a set of flag constants to report incidents. They are defined and documented in `LLUI_PAINTER_impl.h` (for embedded targets) and `LLUIPainter` (for front panels).

Constant	Value	Description
`DRAWING_LOG_NOT_IMPLEMENTED`	`1 << 0`	This function is not implemented.
`DRAWING_LOG_FORBIDDEN`	`1 << 1`	This function must not be called in this situation.
`DRAWING_LOG_OUT_OF_MEMORY`	`1 << 2`	The system ran out of memory.
`DRAWING_LOG_CLIP_MODIFIED`	`1 << 3`	The VEE Port modified clip values in the graphics context.
`DRAWING_LOG_LIBRARY_INCIDENT`	`1 << 29`	An incident occurred in an underlying library.
`DRAWING_LOG_UNKNOWN_INCIDENT`	`1 << 30`	An incident that does not match other flags occurred.
`DRAWING_LOG_ERROR`	`1 << 31`	Special flag denoting critical incidents.

The special value `DRAWING_SUCCESS` (defined as `0`) represents a state where no drawing log flags are set, so encountering this value means no incident was reported.

New flag constants may be added in future versions of MicroUI. Also, their actual values may change, and the developer should not rely on them.

Hint: Sometimes, incidents may match more than one flag constant. In such cases, the VEE Port may report the incident with multiple flags by combining them with the bitwise OR operator (`|`), just like any other flags. For example, an out-of-memory incident occurring in an underlying drawing library may be reported with the value `DRAWING_LIBRARY_INCIDENT | DRAWING_OUT_OF_MEMORY`.

Embedded Targets

MicroUI exposes two functions to be used in the VEE Port. Both functions are declared in `LLUI_DISPLAY.h`, and their documentation is available in that file.

- `LLUI_DISPLAY_reportWarning` reports a warning to the application. It will set the flags passed as an argument in the graphics context. It will *not* reset the previous flag values, thus retaining all reported incidents until the application clears the flags.
- `LLUI_DISPLAY_reportError` reports an error to the application. It behaves similarly to `LLUI_DISPLAY_reportWarning`, except it will additionally set the flag `DRAWING_LOG_ERROR`. This special flag will cause an exception to be thrown in the application the next time the application checks the flags.

For example, if the VEE Port contains a custom implementation to draw a line that may cause an out-of-memory error, it could report this error this way:

```
void LLUI_PAINTER_IMPL_drawLine(MICROUI_GraphicsContext* gc, jint startX, jint startY, jint␣
↪endX, jint endY) {
   // This could cause an out-of-memory error.
   unsigned int result = custom_line_drawing();
```

(continues on next page)

(continued from previous page)

```
   // Check if an error occurred.
   if (result == OUT_OF_MEMORY) {
      // If an error occurred, set the corresponding flag.
      LLUI_DISPLAY_reportError(gc, DRAWING_LOG_OUT_OF_MEMORY);
   }
}
```

Simulator

Similarly, MicroUI exposes two functions to set drawing log flags in the front panel implementation. Both functions are declared as methods of the interface `MicroUIGraphicsContext` and are documented there. The graphics engine provides an implementation for these methods.

- `MicroUIGraphicsContext.reportWarning` behaves like `LLUI_DISPLAY_reportWarning` and reports a warning to the application.
- `MicroUIGraphicsContext.reportError` behaves like `LLUI_DISPLAY_reportError` and reports an error to the application.

The front panel version of the previous example that reported an out-of-memory error would look like this:

```
public static void drawLine(byte[] target, int startX, int startY, int endX, int endY) {
   LLUIDisplay engine = LLUIDisplay.Instance;

   synchronized (engine) {
      MicroUIGraphicsContext gc = engine.mapMicroUIGraphicsContext(target);

      // This could cause an out-of-memory error.
      int result = CustomDrawings.drawLine();

      // Check if an error occurred.
      if (result == Constants.OUT_OF_MEMORY) {
         // If an error occurred, set the corresponding flag.
         gc.reportError(gc, DRAWING_LOG_OUT_OF_MEMORY);
      }
   }
}
```

6.14.10 Images

Overview

Principle

The Image Engine is designed to make the distinction between three kinds of MicroUI images:

- the images which can be used by the application without a loading step: class Image,
- the images which requires a loading step before being usable by the application: class ResourceImage,
- the buffered images where the application can draw into: class BufferedImage.

The first kind of image requires the Image Engine to be able to use (get, read and draw) an image referenced by its path without any loading step. The *open* step should be very fast: just have to find the image in the application resources list and create an Image object which targets the resource. No RAM memory to store the image pixels is required: the Image Engine directly uses the resource address (often in FLASH memory). And finally, *closing* step is useless because there is nothing to free (except Image object itself, via the garbage collector).

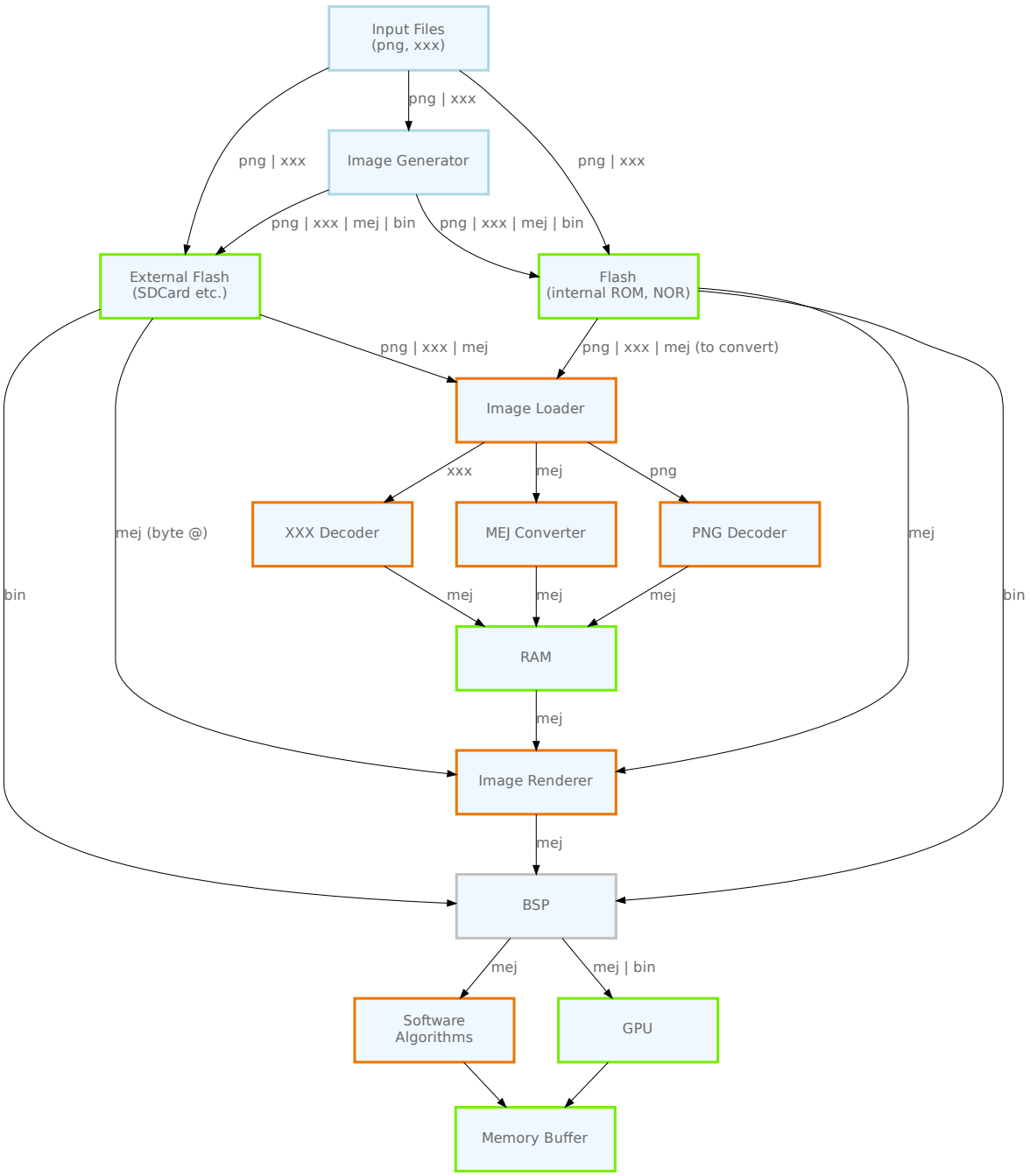
The second kind of image requires the Image Engine to be able to use (load, read and draw) an image referenced by its path with or without any loading step. When the image is understandable by the Image Engine without any loading step, the image is considered like the first kind of image (fast *open* step, no RAM memory, useless *closing* step). When a loading step is required (dynamic decoding, external resource loading, image format conversion), the *open* state becomes longer and a buffer in RAM is required to store the image pixels. By consequence a *closing* step is required to free the buffer when image becomes useless.

The third kind of image requires, by definition, a buffer to store the image data. Image Engine must be able to use (create, read and draw) this kind of image. The *open* state consists in creating a buffer. By consequence a *closing* step is required to free the buffer when the image becomes useless. Contrary to the other kinds of images, the application will be able to draw into this image.

Functional Description

The Image Engine is composed of:

- An “Image Generator” module, for converting images into a MicroEJ format (known by the Image Engine Renderer) or into a VEE Port binary format (cannot be used by the Image Engine Renderer), before runtime (pre-generated images).
- The “Image Loader” module, for loading, converting and closing the images.
- A set of “Image Decoder” modules, for converting standard image formats into a MicroEJ format (known by the Image Renderer) at runtime. Each Image Decoder is an additional module of the main module “Image Loader”.
- The “Image Renderer” module, for reading and drawing the images in MicroEJ format.



- **Colors:**
  - blue: off-board elements (tools, files).
  - green: hardware elements (memory, processor).

  - orange: on-board Graphics Engine elements.
  - gray: BSP.

- **Line labels:**
  - `png` : symbolizes all standard image input formats (PNG, JPG, etc.).
  - `xxx` : symbolizes a non-standard input format.
  - `mej` : symbolizes the MicroEJ output format (*MicroEJ Format: Standard*).
  - `bin` : symbolizes a VEE Port binary format (*Binary Format*).

Process overview:

1. The user specifies the pre-generated images to embed (see *Image Generator*) and / or the images to embed as regular resources (see *Encoded Image*).
2. The files are embedded as resources with the application. The files' data are linked into the FLASH memory.
3. When the application creates a MicroUI Image object, the Image Loader loads the image, calling the right sub Image Engine module (see *Image Generator* and *Encoded Image*) to decode the specified image.
4. When the application draws this MicroUI Image on the display (or on buffered image), the decoded image data is used, and no more decoding is required, so the decoding is done only once.
5. When the MicroUI Image is no longer needed, it must be closed explicitly by the application. The Image Engine Core asks the right sub Image Engine module (see *Image Generator* and *Encoded Image*) to free the image working area.

Dependencies

- MicroUI module (see *MicroUI*),
- Display module (see *Display*): the characteristics of the target display are used to configure the *Image Generator*.

Image Format

The Image Engine makes the distinction between:

- The *input format*: the format of the original image.
- The *output format*: the image format used by the Image Renderer.

Several formats are managed in input: PNG, JPEG, BMP, etc. A specific VEE Port can support additional input formats.

Several formats are managed in output: the MicroEJ formats and the binary format. The output format can be:

- Generated from the input format using the off-board tool *Image Generator* at application compile-time.
- Generated from the input format by using a *runtime decoder* of the *Image Loader* at application run-time.
- Dynamically created when using a BufferedImage.

The Image Renderer manages only the MicroEJ formats (*MicroEJ Format: Standard*, *MicroEJ Format: Display*, and *MicroEJ Format: Custom*).

The following table lists all the formats and their usage.

Format	Input	Output	BufferedImage
Display	no	yes	yes
Standard	no	yes	yes[1]
Grayscale	no	yes	yes[Page 897, 1]
RLE	no	yes	no
Custom	no	not yet	yes[1]
Binary	no	yes	no
Original	yes	no	no

The following sections list all the formats and their usage.

MicroEJ Format: Display

The display buffer holds a pixel encoding which is:

- standard: see *Standard Output Formats*,
- grayscale: see *Grayscale Output Formats*,
- non-standard: see *Display Output Format* and *Pixel Structure*.

The non-standard display format can be customized to encode the pixel in the same encoding as the display. The number of bits per pixel and the pixel bit organization is asked during the MicroEJ format generation and when the `drawImage` algorithms are running. If the image to encode contains some transparent pixels, the output file will embed the transparency according to the display's implementation capacity. When all pixels are fully opaque, no extra information will be stored in the output file to free up some memory space.

Notes:

- From the Image Engine point of view, the non-standard display format stays a MicroEJ format, readable by the Image Renderer.
- The required memory to encode an image with a non-standard display format is similar to *MicroEJ Format: Standard*.
- When the display format is standard or grayscale, the encoded image format is replaced by the related standard format.
- The *Graphics Engine's drawing software algorithms* only target (are only compatible with) the buffered images whose format is the same as the display format (standard or non-standard).

MicroEJ Format: Standard

See *Standard Output Formats*.

This format requires a small header (around 20 bytes) to store the image size (width, height), format, flags (is_transparent, etc.), row stride, etc. The required memory also depends on the number of bits per pixel of the MicroEJ format:

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

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

[1] Need some support in the VEE Port to support formats different than the display one (see *Buffered Image*).



Here are the conversions of 32-bit to each format:

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

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

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

```
u32 convertARGB8888toRAWFormat(u32 c){
    return 0
            | ((c & 0xf0000000) >> 16)
            | ((c & 0x00f00000) >> 12)
            | ((c & 0x0000f000) >> 8)
            | ((c & 0x000000f0) >> 4)
            ;
}
```

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

```
u32 convertARGB8888toRAWFormat(u32 c){
    return 0
            | (((c & 0xff000000) == 0xff000000) ? 0x8000 : 0)
            | ((c & 0xf80000) >> 9)
            | ((c & 0x00f800) >> 6)
            | ((c & 0x0000f8) >> 3)
            ;
}
```

- RGB888: 24-bit format, 8 per color.

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

- RGB565: 16-bit format, 5 for red, 6 for green, 5 for blue.

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

- A8: 8-bit format, only transparency is encoded.

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

- A4: 4-bit format, only transparency is encoded.

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

- A2: 2-bit format, only transparency is encoded.

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

- A1: 1-bit format, only transparency is encoded.

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

The pixel order follows this rule:

```
pixel_offset = (pixel_Y * image_width + pixel_X) * bpp / 8;
```

MicroEJ Format: Grayscale

See *Grayscale Output Formats*.

This format requires a small header (around 20 bytes) to store the image size (width, height), format, flags (is_transparent, etc.), row stride, etc. The required memory also depends on the number of bits per pixel of the MicroEJ format:

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

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

```
u32 convertARGB8888toRAWFormat(u32 c){
    return 0
        | ((color >> 24) & 0xf0)
        | ((toGrayscale(color) & 0xff) / 0x11)
        ;
}
```

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

```
u32 convertARGB8888toRAWFormat(u32 c){
    return 0
        | ((color >> 28) & 0xc0)
        | ((toGrayscale(color) & 0xff) / 0x55)
        ;
}
```

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

```
u32 convertARGB8888toRAWFormat(u32 c){
    return 0
        | ((c & 0xff000000) == 0xff000000 ? 0x2 : 0x0)
```

(continues on next page)

(continued from previous page)

```
        | ((toGrayscale(color) & 0xff) / 0xff)
        ;
}
```

- C4: 4 bits with grayscale conversion.

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

- C2: 2 bits with grayscale conversion.

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

- C1: 1 bit with grayscale conversion.

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

The pixel order follows this rule:

```
pixel_offset = (pixel_Y * image_width + pixel_X) * bpp / 8;
```

MicroEJ Format: RLE Compressed

See *Compressed Output Formats*.

MicroEJ Format: Custom

A custom format embeds a buffer whose data are VEE Port specific. This data may be:

- a pixel buffer whose encoding is different than the formats proposed before,
- a buffer that is not a pixel buffer.

This format is identified by a specific format value between 0 and 7: see custom formats.

Images with a custom format can be used as any other image. For that, it requires some support at different levels depending on their usage:

- To convert an image to this format at compile-time and embed it, an extension of the image generator is necessary; see *VEE Port MicroEJ Custom Format*.
- To create a new one at runtime, some native extension is necessary; see *Buffered Image*.
- To use it as a source (to draw the image in another buffer), some native extension is necessary; see *Custom Format Support*.
- To use it as a destination (to draw into the image), some native extension is necessary; see *Buffered Image*.

Binary Format

This format is not compatible with the Image Renderer and MicroUI. It can be used by MicroUI addon libraries which provide their image management procedures.

Advantages:

- Encoding is known by VEE Port.
- Compression is inherent to the format itself.

Disadvantages:

- This format cannot target a MicroUI Image (unsupported format).

Original Input Format

See *Unspecified Output Format*.

An image can be embedded without any conversion/compression. This allows embedding the resource as it is to keep the source image characteristics (compression, bpp, etc.). This option produces the same result as specifying an image as a resource in the MicroEJ launcher.

The following table lists the original formats that can be decoded at run-time and/or compile-time:

- Image Generator: the off-board tool that converts an image into an output format. All AWT ImageIO default formats are supported and always enabled.
- Front Panel: the decoders embedded by the simulator part. All AWT ImageIO default formats are supported but disabled by default.
- Runtime Decoders: the decoders embedded by the embedded part.

Table 21: Original Image Formats

Type	Image Generator	Front Panel	Runtime Decoders
Graphics Interchange Format (GIF)	yes	yes[2]	no[7]
Joint Photographic Experts Group (JPEG)	yes	yes[2]	no[7]
Portable Network Graphics (PNG)	yes	yes[3]	yes[3]
Windows bitmap (BMP)	yes	yes[4]	yes/no[4]
Web Picture (WebP)	yes[5]	yes[5]	yes[6]

[2] The formats are disabled by default; see:ref:*fp_ui_decoder*.
[7] The UI-pack does not provide some runtime decoders for these formats, but a BSP can add its decoders (see *Encoded Image*).
[3] The PNG format is supported when the module `PNG` is selected in the VEE Port configuration file (see *Encoded Image*).
[4] The Monochrome BMP is supported when the module `BMPM` is selected in the VEE Port configuration file (see *Encoded Image*); the *colored* BMP format is only supported by the Front Panel (disabled by default, see *Image Decoders*).
[5] Install the tool `com.microej.tool#imageio-webp-1.0.1` from the *Developer Repository* in the VEE Port to support the WEBP format (see *Service Image Loader* and *Image Decoders*).
[6] Install the C component `com.microej.clibrary.thirdparty#libwebp-1.0.1` in the BSP to support the WEBP format at runtime.

GPU Format Support

The MicroEJ formats *display*, *standard* and *grayscale* may be customized to be compatible with the hardware (usually GPU). It can be extended by one or several restrictions on the pixels array:

- Its start address has to be aligned on a higher value than the number of bits-per-pixels.
- A padding has to be added after each line (row stride).
- The MicroEJ format can hold a VEE Port-dependent header between the MicroEJ format header (start of file) and the pixel array. The MicroEJ format is designed to let the VEE Port encode and decode this additional header. This header is unnecessary and never used for Image Engine software algorithms.

Note: From the Image Engine point of view, the format stays a MicroEJ format, readable by the Image Engine Renderer.

Advantages:

- The GPU recognizes encoding.
- Drawing an image is often very fast.
- Supports opacity encoding.

Disadvantages:

- No compression: the image size in bytes is proportional to the number of pixels. The required memory is similar to *MicroEJ Format: Standard* when no custom header exists.

When the MicroEJ format holds another header (called `custom_header`), the required memory is:

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

The row stride allows adding some padding at the end of each line to start the next line at an address with a specific memory alignment; it is often required by hardware accelerators (GPU). The row stride is, by default, a value in relation to the image width: `row_stride_in_bytes = image_width * bpp / 8`. Thanks to the Abstraction Layer API `LLUI_DISPLAY_IMPL_getNewImageStrideInBytes`, it can be customized at image buffer creation. The required memory becomes:

```
required_memory = header + custom_header + row_stride * image_height;
```



Image Generator

Principle

The Image Generator module is an off-board tool that generates image data that is ready to be displayed without needing additional runtime memory. The two main advantages of this module are:

- A pre-generated image is already encoded in the format known by the Image Renderer (MicroEJ format) or by the VEE Port (custom binary format). The time to create an image is very fast and does not require any RAM (Image Loader is not used).
- No extra support is needed (no runtime Image Decoder).

Functional Description

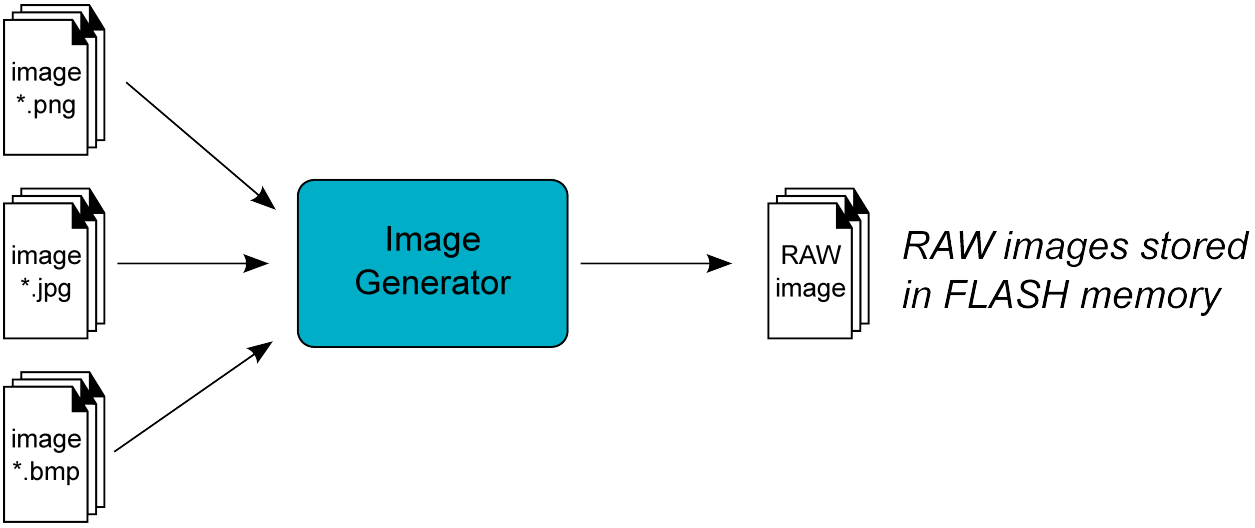


Fig. 52: Image Generator Principle

Process overview (see too *Functional Description*)

1. The user defines, in a text file, the images to load.
2. The Image Generator outputs a binary file for each image to convert.
3. The raw files are embedded as (hidden) resources within the MicroEJ Application. The binary files' data are linked into the FLASH memory.
4. When the application creates a MicroUI Image object which targets a pre-generated image, the Image Engine has only to create a link from the MicroUI image object to the data in the FLASH memory. Therefore, the loading is very fast; only the image data from the FLASH memory is used: no copy of the image data is sent to the RAM first.
5. When the MicroUI Image is no longer needed, it is garbage-collected by the VEE Port, which just deletes the useless link to the FLASH memory.

The image generator can run in two modes:

- Standalone mode: the image to convert (input files) are standard (PNG, JPEG, etc.), the generated binary files are in MicroEJ format and do not depend on VEE Port characteristics or restrictions (see *MicroEJ Format: Standard*).
- Extended mode: the image to convert (input files) may be custom, the generated binary files can be encoded in customized MicroEJ format (can depend on several VEE Port characteristics and restrictions, see *MicroEJ Format: Display*, *GPU Format Support* and *MicroEJ Format: Custom*) or the generated files are encoded in another format than MicroEJ format (binary format, see *Binary Format*).

Structure

The Image Generator module is constituted from several parts, the core part and services parts:

- "Core" part: it takes an images list file as entry point and generates a binary file (no specific format) for each file. To read a file, it redirects the reading to the available service loaders. To generate a binary file, it redirects the encoding to the available service encoders.
- "Service API" part: it provides some APIs used by the core part to load input files and to encode binary files. It also provides some APIs to customize the MicroEJ format.
- "Standard input format loader" part: this service loads standard image files (PNG, JPEG, etc.).
- "MicroEJ format generator" part: this service encodes an image in MicroEJ format.

Standalone Mode

The standalone Image Generator embeds all parts described above. By consequence, once installed in a VEE Port, the standalone image generator does not need any extended module to generate MicroEJ files from standard images files.

Extended Mode

To increase the capabilities of Image Generator, the extension must be built and added in the VEE Port. As described above this extension will be able to:

- read more input image file formats,
- extend the MicroEJ format with VEE Port characteristics,
- encode images in a third-party binary format.

To do that the Image Generator provides some services to implement. This chapter explain how to create and include this extension in the VEE Port. Next chapters explain the aim of each service.

1. Create a `std-javalib` project. The module name must start with the prefix `imageGenerator` (for instance `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 VEE Port (minimum `13.0.0`). The `module.ivy` should look like:

   ```
   <ivy-module version="2.0" xmlns:ea="http://www.easyant.org" xmlns:m="http://www.easyant.
   ↪org/ivy/maven" xmlns:ej="https://developer.microej.com" ej:version="2.0.0">

      <info organisation="com.microej.microui" module="imageGeneratorMyPlatform" status=
   ↪"integration" revision="1.0.0">
         <ea:build organisation="com.is2t.easyant.buildtypes" module="build-std-javalib"␣
   ↪revision="2.+"/>
      </info>
   ```

(continues on next page)

(continued from previous page)

```
   <configurations defaultconfmapping="default->default;provided->provided">
      <conf name="default" visibility="public" description="Runtime dependencies to␣
↪other artifacts"/>
      <conf name="provided" visibility="public" description="Compile-time dependencies␣
↪to APIs provided by the VEE Port"/>
      <conf name="documentation" visibility="public" description="Documentation related␣
↪to the artifact (javadoc, PDF)"/>
      <conf name="source" visibility="public" description="Source code"/>
      <conf name="dist" visibility="public" description="Contains extra files like␣
↪README.md, licenses"/>
      <conf name="test" visibility="private" description="Dependencies for test␣
↪execution. It is not required for normal use of the application, and is only␣
↪available for the test compilation and execution phases."/>
   </configurations>

   <publications/>

   <dependencies>
      <dependency org="com.microej.pack.ui" name="ui-pack" rev="[UI Pack version]">
         <artifact name="imageGenerator" type="jar"/>
      </dependency>
   </dependencies>
</ivy-module>
```

3. Create the folder `META-INF/services` in source folder `src/main/resources` (this folder will be filled in later).
4. When a service is added (see next chapters), build the easyant project.
5. Copy the generated jar: `target~/artifacts/imageGeneratorMyPlatform.jar` in the VEE Port configuration project folder: `MyPlatform-configuration/dropins/tools/`
6. Rebuild the platform.

Advanced: Test the Extension Project

To quickly test an extension project without rebuilding the VEE Port or manually exporting the project, add the *Application Option* `ej.imagegenerator.extension.project` to the absolute path of an Image Generator Extension project (e.g. `c:\mycompany\myimagegeneratorextension`). The Image Generator will use the specified Image Generator Extension project instead of the one included in the VEE Port. This feature is useful for locally testing certain changes in the Image Generator Extension project.

```
-Dej.imagegenerator.extension.project=${project_loc:myimagegeneratorextension}
```

> **Warning:** This feature only works if the VEE Port has been built with the Image Generator module enabled.
>
> The VEE Port will not actually contain the changes until a new VEE Port is built: the VEE Port dropins folder must be updated after any changes to the Image Generator Extension project.

Warning: Using this feature automatically disables the *image cache*.

Service Image Loader

The standalone Image Generator is not able to load all images formats, for instance SVG format. The service loader can be used to add this feature in order to generate an image file in MicroEJ format. There are two ways to populate the service loader: create a custom implementation of `com.microej.tool.ui.generator.MicroUIRawImageGeneratorExtension` or `javax.imageio.spi.ImageReaderSpi`.

MicroUIRawImageGeneratorExtension

This service allows to add a custom image reader.

1. Open image generator extension project.
2. Create an implementation of interface `com.microej.tool.ui.generator.MicroUIRawImageGeneratorExtension`.
3. Create the file `META-INF/services/com.microej.tool.ui.generator.MicroUIRawImageGeneratorExtension` and open it.
4. Note down the name of created class, with its package and classname.
5. Rebuild the image generator extension, copy it in VEE Port configuration project (`dropins/tools/`) and rebuild the VEE Port (see above).

Note: The class `com.microej.tool.ui.generator.BufferedImageLoader` already implements the interface. This implementation is used to load standard images. It can be sub-classed to add some behavior.

ImageReaderSpi

This extension is part of AWT ImageIO. By default, the ImageIO class only manages the standard image formats JPG, PNG, BMP and GIF. It allows to add some image readers by adding some implementations of the service *javax.imageio.spi.ImageReaderSpi*.

Since UI Pack 13.2.0, the Image Generator automatically includes new image decoders (new ImageIO services, see the class `com.microej.tool.ui.generator.BufferedImageLoader`), compiled in JAR files that follow this convention:

1. The JAR contains the service declaration `/META-INF/services/javax.imageio.spi.ImageReaderSpi`,
2. The JAR filename's prefix is *imageio-*,
3. The JAR location is the VEE Port configuration project's `dropins/tools/` directory.

Note: The same JAR is used by the Image Generator and by the *Front Panel*.

Customize MicroEJ Standard Format

As mentioned above (*MicroEJ Format: Display* and *GPU Format Support*), the MicroEJ format can be extended by notions specific to the VEE Port (and often to the GPU the VEE Port is using). The generated file stays a MicroEJ file format, usable by the Image Renderer. Additionally, the file becomes compatible with the VEE Port constraints.

1. Open image generator extension project.
2. Create a subclass of `com.microej.tool.ui.generator.BufferedImageLoader` (to be able to load standard images) or create an implementation of interface `com.microej.tool.ui.generator.MicroUIRawImageGeneratorExtension` (to load custom images).
3. Override method `convertARGBColorToDisplayColor(int)` if the VEE Port's display pixel encoding is not standard (see *Pixel Structure*).
4. Override method `getStride(int)` if a padding must be added after each line.
5. Override method `getOptionalHeader()` if an additional header must be added between the MicroEJ file header and pixels array. The header size is also used to align image memory address (custom header is aligned on its size).
6. Create the file `META-INF/services/com.microej.tool.ui.generator.MicroUIRawImageGeneratorExtension` and open it.
7. Note down the name of created class, with its package and classname.
8. Rebuild the image generator extension, copy it in VEE Port configuration project and rebuild the VEE Port (see above).

If the only constraint is the pixels array alignment, the Image Generator extension is not useful:

1. Open VEE Port configuration file `display/display.properties`.
2. Add the property `imageBuffer.memoryAlignment`.
3. Build again the VEE Port.

This alignment will be used by the Image Generator and also by the Image Loader.

VEE Port MicroEJ Custom Format

The Image Generator does not yet provide a service for generating the *MicroEJ Format: Custom*. A custom image can only be created at runtime, see *Buffered Image*.

VEE Port Binary Format

The Image Generator can generate a *binary file* compatible with the VEE Port (and not with the Image Renderer). This is very useful when a VEE Port features a foundation library that can use other kinds of images than the MicroUI library. The binary file can be encoded according to the user's options in the images list file.

1. Open image generator extension project.
2. Create an implementation of the interface `com.microej.tool.ui.generator.ImageConverter`.
3. Create the file `META-INF/services/com.microej.tool.ui.generator.ImageConverter` and open it.
4. Note the name of the created class, with its package and class name.
5. Rebuild the image generator extension, copy it into the VEE Port configuration project, and rebuild the VEE Port (see above).

The binary file can have two kinds of formats (see the API `OutputFileType getType()`):

- A simple resource: the binary output file is embedded as a resource; the application (or the library) can retrieve the file by using an API like `getResourceAsStream()`.
- An immutable file: the output file contains one or several *immutable objects*; the application (or the library) can retrieve the objects by using the *Beyond Profile (BON)* library.

Configuration File

The Image Generator uses a configuration file (also called the "list file") for describing images that need to be processed. The list file is a text file in which each line describes an image to convert. The image is described as a resource path, and should be available from the application classpath.

Note: The list file must be specified in the application launcher (see *Standalone Application Options*). However, all the files in the application classpath with suffix `.images.list` are automatically parsed by the Image Generator tool.

Each line can add optional parameters (separated by a ':') which define and/or describe the output file format (raw format). When no option is specified, the image is not converted and embedded as well.

Note: See *Configuration File* to understand the list file grammar.

- MicroEJ standard output format: to encode the image in a standard MicroEJ format, specify the MicroEJ format:

Listing 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 ARGB1565_RLE output format (formerly RLE1): to encode the image in ARGB1565_RLE format, specify `ARGB1565_RLE` as output format:

Listing 6: ARGB1565_RLE Output Format Example

```
image1:ARGB1565_RLE
image1:RLE1 # Deprecated
```

- 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 VEE Port, refer to the VEE Port 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 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.

- Refer to the chapter *Images* to have more details how to use this kind of resources.
- Refer to the chapter *External Resource* to have more details how the Image Engine manages this kind of resources.

Installation

The Image Generator is an additional module for the MicroUI library. When the MicroUI module is installed, also install this module in order to be able to target pre-generated images.

In the VEE Port configuration file, check UI > Image Generator to install the Image Generator module. When checked, the properties file `imageGenerator/imageGenerator.properties` is required to specify the Image Generator extension project. When no extension is required (standalone mode only), this property is useless.

Use

The MicroUI Image APIs are available in the class ej.microui.display.Image ant its subclasses. There are no specific APIs that use a pre-generated image. When an image has been pre-processed, the MicroUI Image APIs `getImage` and `loadImage` will get/load the images.

Refer to the chapter *Standalone Application Options* (Libraries > MicroUI > Image) for more information about specifying the image configuration file.

Image Loader

Principle

The Image Loader is a module of the MicroUI runtime that:

- retrieves image data that is ready to be displayed without needing additional runtime memory,
- retrieves image data that is required to be converted into the format known by the Image Renderer (MicroEJ format),
- retrieves image in external memories (*External Resource* loader),
- converts images in MicroEJ format,
- creates a runtime buffer to manage MicroUI *Buffered Image*,
- manages dynamic images life cycle.

Note: The Image Loader is managing images to be compatible with Image Renderer. It does manage image in custom format (see *Binary Format*)

Functional Description

1. The application is using one of three ways to create a MicroUI Image object.
2. The Image Loader creates the image according the MicroUI API, image location, image input format and image output format to be compatible with Image Renderer.
3. When the application closes the image, the Image Loader frees the RAM memory.

Images Heap

There are several ways to create a MicroUI Image. Except few specific cases, the Image Loader requires some RAM memory to store the image content in MicroEJ format. This format requires a small header as explained here: *MicroEJ Format: Standard*. It can be GPU compatible as explained here: *GPU Format Support*.

The heap size is application dependant. In the application launcher, set its size in Libraries > MicroUI > Images heap size (in bytes) . It will declare a section whose name is `.bss.microui.display.imagesHeap` .

By default, the Image Loader uses an internal best fit allocator to allocate the image buffers (internal Graphics Engine's allocator). Some specific *Abstraction Layer API* (LLAPI) are available to override this default implementation. These LLAPIs may be helpful to control the buffers allocation, retrieve the remaining space, etc. When not implemented by the BSP, the default internal Graphics Engine's allocator is used.

External Resource

Principle

An image is retrieved by its path (except for BufferedImage). The path describes a location in the application classpath. The resource may be generated at the same time as the application (internal resource) or be external (external resource). The Image Loader can load some images located outside the CPU addresses' space range. It uses the External Resource Loader.

When an image is located in such memory, the Image Loader copies it into RAM (into the CPU addresses' space range). Then it considers the image as an internal resource: it can continue to load the image (see following chapters). The RAM section used to load the external image is automatically freed when the Image Loader does not need it again.

The image may be located in external memory but be available in CPU addresses' space ranges (byte-addressable). In this case, the Image Loader considers the image as *internal* and does not need to copy its content in RAM.

Configuration File

Like internal resources, the Image Generator uses a *configuration file* (also called the "list file") for describing images that need to be processed. The list file must be specified in the application launcher (see *Standalone Application Options*). However, all the files in the application classpath with the suffix `.imagesext.list` are automatically parsed by the Image Generator tool.

Process

This chapter describes the steps to setup the loading of an external resource from the application:

1. Add the image to the application project resources (typically in the source folder `src/main/resources` and in the package `images`).
2. Create / open the configuration file (e.g. `application.imagesext.list`).
3. Add the relative path of the image and its output format (e.g. `/images/myImage.png:RGB565` see *Images*).
4. Build the application: the Image Generator converts the image in RAW format in the external resources folder (`[application_output_folder]/externalResources`).
5. Deploy the external resources to the external memory (SDCard, flash, etc.) of the device.
6. (optional) Configure the *External Resources Loader* to load from this source.
7. Build the application and run it on the device.
8. The application loads the external resource using ResourceImage.loadImage(String).
9. The image loader looks for the image and copies it in the *images heap* (no copy if the external memory is byte-addressable).
10. (optional) The image may be decoded (for instance: PNG), and the source image is removed from the images heap.
11. The external resource is immediately closed: the image's bytes have been copied in the images heap, or the image's bytes are always available (byte-addressable memory).
12. The application can use the image.
13. The application closes the image: the image is removed from the image heap.

Simulation

The Simulator automatically manages the external resources like internal resources. All images listed in `*.imagesext.list` files are copied in the external resources folder, and this folder is added to the Simulator's class-path.

Image in MicroEJ Format

An image may be pre-processed (*Image Generator*) and so already in the format compatible with Image Renderer: MicroEJ format.

- When application is loading an image which is in such format and without specifying another output format, the Image Loader has just to make a link between the MicroUI Image object and the resource location. No more runtime decoder or converter is required, and so no more RAM memory.
- When application specifies another output format than MicroEJ format encoded in the image, Image Loader has to allocate a buffer in RAM. It will convert the image in the expected MicroEJ format.
- When application is loading an image in MicroEJ format stored as *External Resource*, the Image Loader has to copy the image into RAM memory to be usable by Image Renderer.

Encoded Image

An image can be encoded (PNG, JPEG, etc.). In this case Image Loader asks to its Image Decoders module if a decoder is able to decode the image. The source image is not copied in RAM (expect for images stored as *External Resource*). Image Decoder allocates the decoded image buffer in RAM first and then inflates the image. The image is encoded in MicroEJ format specified by the application, when specified. When not specified, the image in encoded in the default MicroEJ format specified by the Image Decoder itself.

The UI extension provides two internal Image Decoders modules:

- PNG Decoder: a full PNG decoder that implements the PNG format (`https://www.w3.org/Graphics/PNG`). Regular, interlaced, indexed (palette) compressions are handled.
- BMP Monochrome Decoder: .bmp format files that embed only 1 bit per pixel can be decoded by this decoder.

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.

Installation

The Image Decoders modules are some additional modules to the Display module. The decoders belong to distinct modules, and either or several may be installed.

In the VEE Port configuration file, check UI > Image PNG Decoder to install the runtime PNG decoder. Check UI > Image BMP Monochrome Decoder to install the runtime BMP monochrom decoder.

Use

The MicroUI Image APIs are available in the class ej.microui.display.Image. There is no specific API that uses a runtime image. When an image has not been pre-processed (see *Image Generator*), the MicroUI Image APIs `createImage*` will load this image.

Image Renderer

Principle

The Image Renderer is a module of the MicroUI runtime that reads and draws the images (see *Image Format*). It calls Abstraction Layer APIs to draw and transform the images (rotation, scaling, deformation, etc.). It also includes software algorithms to perform the rendering.

Functional Description

All MicroUI image drawings are redirected to a set of Abstraction Layer APIs. All Abstraction Layer APIs are implemented by weak functions, which call software algorithms. The BSP can override this default behavior for each Abstraction Layer API independently. Furthermore, the BSP can override an Abstraction Layer API for a specific MicroEJ format (for instance `ARGB8888`) and call the software algorithms for all other formats.

Destination Format

Since MicroUI 3.2, the destination buffer of the drawings can be different from the display buffer format (see *MicroEJ Format: Display*). This destination buffer format can be a *standard format* (ARGB8888, A8, etc.) or a *custom format*.

See *Buffered Image* for more information about how to create buffered images with another format than the display format and how to draw in them.

Input Formats

Standard

The Image Renderer is by default able to draw all *standard formats*. No extra support in the VEE Port is required to draw this kind of image.

The image drawing resembles a *shape drawing*. The drawing is performed by default by the *Graphics Engine Software Algorithms* and can be overridden to use a third-party library or a GPU.

Custom

A *MicroEJ Format: Custom* image can be:

- an image with a pixel buffer but whose pixel organization is not standard,
- an image with a data buffer: an image encoded with a third-party encoder (proprietary format or not),
- an image with a "command" buffer: instead of performing the drawings on pixels, the image *stores* the drawing actions to replay them later,
- etc.

The VEE Port must extend the Image Renderer to support the drawing of these images. This extension can consist in:

- decoding the image at runtime to draw it,
- using a compatible GPU to draw it,
- using a command interpreter to perform some *shape drawings*,
- etc.

To draw the custom images, the Image Renderer introduces the notion of *custom image drawer*. This drawer is an engine that has the responsibility to draw the image. Each custom image format (`0` to `7`) has its own image drawer.

Each drawing of a custom image is redirected to the associated image drawer.

Note: A custom image drawer can call the UI Shapes Drawing API to draw its elements in the destination.

The implementation is not the same between the Embedded side and the Simulation. However, the concepts are the same and are described in dedicated chapters.

MicroUI C Module

Principle

As described above, an *image drawer* allows drawing the images whose format is *custom*. The *MicroUI C module* is designed to manage the notion of drawers: it does not *support* the custom formats but allows adding some additional drawers.

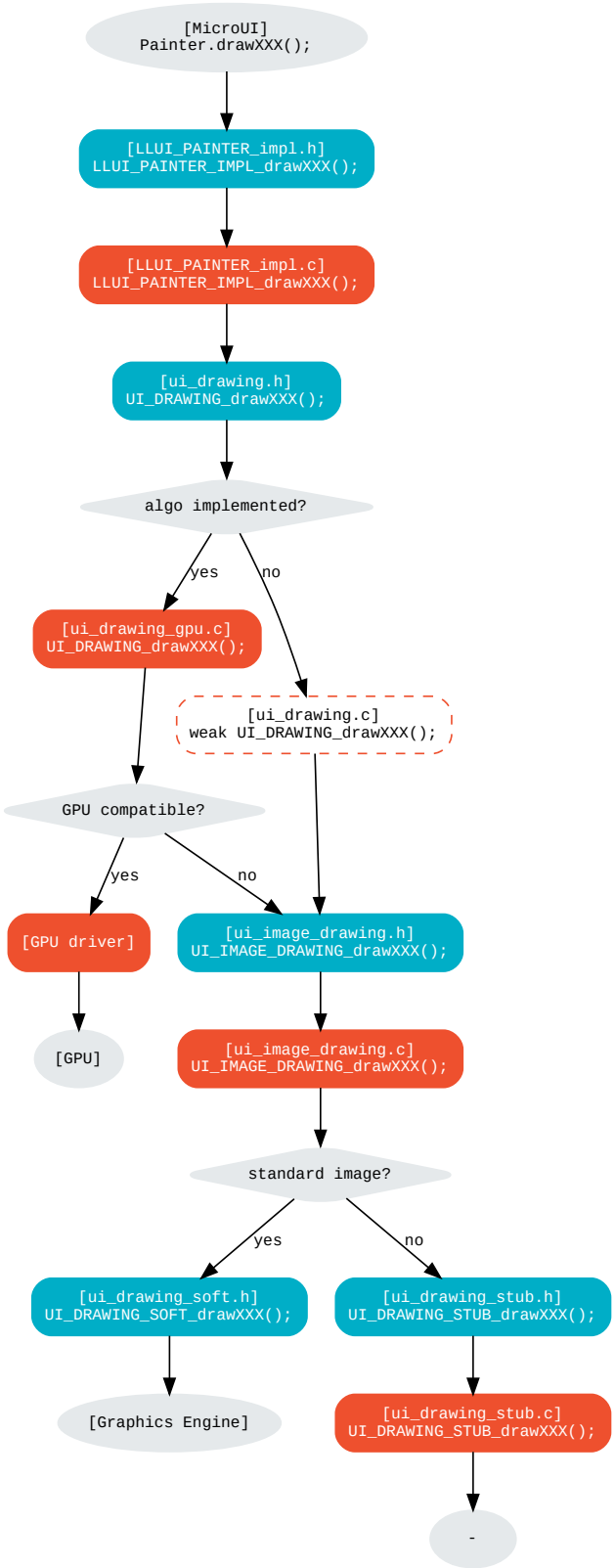
This support uses several weak functions and tables to redirect the image drawings. When this support is not used (when the VEE Port does not need to support *custom* images), this support can be removed to reduce the footprint (by removing the indirection tables) and improve the performances (by reducing the number of runtime function calls).

Standard Formats Only (Default)

The default implementation can only draw images with a *standard format*. In other words, the application cannot draw a custom image. This is the most frequent use case, the only one available with MicroUI before version 3.2.

Hint: To select this implementation (to disable the custom format support), the define `LLUI_IMAGE_CUSTOM_FORMATS` must be unset.

The following graph illustrates the drawing of an image:


[MicroUI]
Painter.drawXXX();
[LLUI_PAINTER_impl.h]
LLUI_PAINTER_IMPL_drawXXX();
[LLUI_PAINTER_impl.c]
LLUI_PAINTER_IMPL_drawXXX();
[ui_drawing.h]
UI_DRAWING_drawXXX();
algo implemented?
yes
no
[ui_drawing_gpu.c]
UI_DRAWING_drawXXX();
[ui_drawing.c]
weak UI_DRAWING_drawXXX();
GPU compatible?
yes
no
[GPU driver]
[GPU]
[ui_image_drawing.h]
UI_IMAGE_DRAWING_drawXXX();
[ui_image_drawing.c]
UI_IMAGE_DRAWING_drawXXX();
standard image?
yes
no
[ui_drawing_soft.h]
UI_DRAWING_SOFT_drawXXX();
[Graphics Engine]
[ui_drawing_stub.h]
UI_DRAWING_STUB_drawXXX();
[ui_drawing_stub.c]
UI_DRAWING_STUB_drawXXX();
-

LLUI_PAINTER_IMPL_drawImage (available in MicroUI C Module)

Similar to `LLUI_PAINTER_IMPL_drawLine`, see *MicroUI C Module*.

UI_DRAWING_drawImage

```
// Available in MicroUI C Module
#define UI_DRAWING_DEFAULT_drawImage UI_DRAWING_drawImage

// To write in the BSP (optional)
#define UI_DRAWING_GPU_drawImage UI_DRAWING_drawImage
```

The function names are set thanks to some `define`. These name redirections are helpful when the VEE Port features more than one destination format (not the use-case here).

UI_DRAWING_GPU_drawImage (to write in the BSP)

Similar to `UI_DRAWING_GPU_drawLine` (see *MicroUI C Module*), but lets the image drawer manage the image instead of calling the software drawer directly.

```
// Unlike the MicroUI C Module, this function is not "weak".
DRAWING_Status UI_DRAWING_GPU_drawImage(MICROUI_GraphicsContext* gc, MICROUI_Image* img,␣
↪jint regionX, jint regionY, jint width, jint height, jint x, jint y, jint alpha) {

   DRAWING_Status status;

   if (is_gpu_compatible(xxx)) {

      // See chapter "Drawings"
      // [...]
   }
   else {
      // Let the image drawer manages the image (available in the C module)
      status = UI_IMAGE_DRAWING_draw(gc, img, regionX, regionY, width, height, x, y, alpha);
   }
   return status;
}
```

UI_DRAWING_DEFAULT_drawImage (available in MicroUI C Module)

```
// Use the preprocessor 'weak'
__weak DRAWING_Status UI_DRAWING_DEFAULT_drawImage(MICROUI_GraphicsContext* gc, MICROUI_
↪Image* img, jint regionX, jint regionY, jint width, jint height, jint x, jint y, jint␣
↪alpha) {
#if !defined(LLUI_IMAGE_CUSTOM_FORMATS)
   return UI_DRAWING_SOFT_drawImage(gc, img, regionX, regionY, width, height, x, y, alpha);
#else
   return UI_IMAGE_DRAWING_draw(gc, img, regionX, regionY, width, height, x, y, alpha);
#endif
}
```

The define `LLUI_IMAGE_CUSTOM_FORMATS` is not set, so the implementation of the weak function only consists in calling the Graphics Engine's software algorithm.

Custom Format Support

In addition to the *standard formats*, this implementation allows drawing images with a *custom format*. This advanced use case is available only with MicroUI 3.2 or higher.

Hint: To select this implementation, the define `LLUI_IMAGE_CUSTOM_FORMATS` must be set (no specific value).

The MicroUI C module uses some tables to redirect the image management to the expected extension. There is one table per Image Abstraction Layer API (draw, copy, region, rotate, scale, flip) to embed only necessary algorithms (a table and its functions are only embedded in the final binary file if and only if the MicroUI drawing method is called).

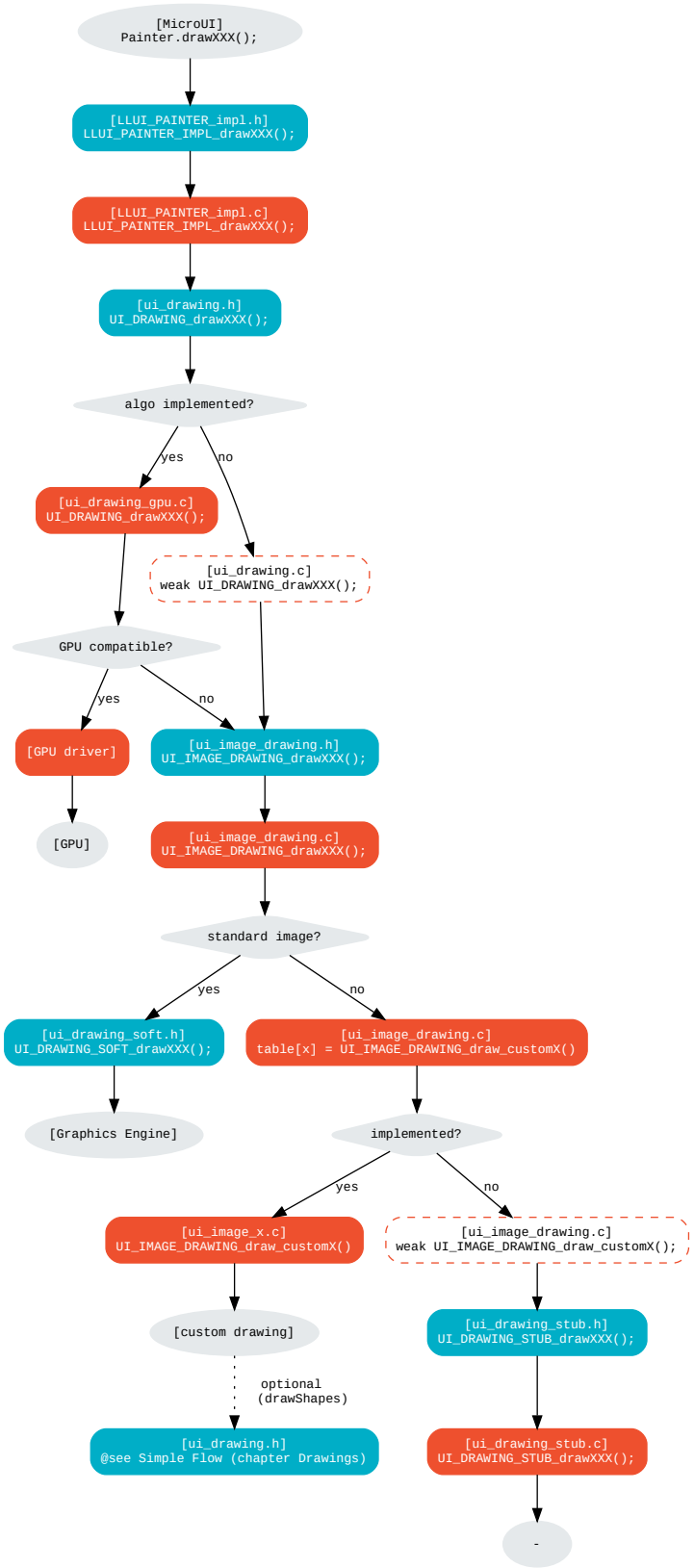
Each table contains ten elements:

```
static const UI_IMAGE_DRAWING_draw_t UI_IMAGE_DRAWING_draw_custom[] = {
      &UI_DRAWING_STUB_drawImage,
      &UI_DRAWING_SOFT_drawImage,
      &UI_IMAGE_DRAWING_draw_custom0,
      &UI_IMAGE_DRAWING_draw_custom1,
      &UI_IMAGE_DRAWING_draw_custom2,
      &UI_IMAGE_DRAWING_draw_custom3,
      &UI_IMAGE_DRAWING_draw_custom4,
      &UI_IMAGE_DRAWING_draw_custom5,
      &UI_IMAGE_DRAWING_draw_custom6,
      &UI_IMAGE_DRAWING_draw_custom7,
};
```

- `UI_DRAWING_STUB_drawImage` is the drawing function called when the drawing function is not implemented,
- `UI_DRAWING_SOFT_drawImage` is the drawing function that redirects the drawing to the *Graphics Engine Software Algorithms*,
- `UI_IMAGE_DRAWING_draw_customX` (`0` to `7`) are the drawing functions for each custom format.

The MicroUI C Module retrieves the table index according to the image format.

The following graph illustrates the drawing of an image:


[MicroUI]
Painter.drawXXX();
[LLUI_PAINTER_impl.h]
LLUI_PAINTER_IMPL_drawXXX();
[LLUI_PAINTER_impl.c]
LLUI_PAINTER_IMPL_drawXXX();
[ui_drawing.h]
UI_DRAWING_drawXXX();
algo implemented?
yes
no
[ui_drawing_gpu.c]
UI_DRAWING_drawXXX();
[ui_drawing.c]
weak UI_DRAWING_drawXXX();
GPU compatible?
yes
no
[GPU driver]
[GPU]
[ui_image_drawing.h]
UI_IMAGE_DRAWING_drawXXX();
[ui_image_drawing.c]
UI_IMAGE_DRAWING_drawXXX();
standard image?
yes
no
[ui_drawing_soft.h]
UI_DRAWING_SOFT_drawXXX();
[Graphics Engine]
[ui_image_drawing.c]
table[x] = UI_IMAGE_DRAWING_draw_customX()
implemented?
yes
no
[ui_image_x.c]
UI_IMAGE_DRAWING_draw_customX()
[ui_image_drawing.c]
weak UI_IMAGE_DRAWING_draw_customX();
[custom drawing]
optional
(drawShapes)
[ui_drawing.h]
@see Simple Flow (chapter Drawings)
[ui_drawing_stub.h]
UI_DRAWING_STUB_drawXXX();
[ui_drawing_stub.c]
UI_DRAWING_STUB_drawXXX();
-

Take the same example as the *Standard Formats Only* implementation (draw an image):

UI_DRAWING_DEFAULT_drawImage (available in MicroUI C Module)

```
// Use the preprocessor 'weak'
__weak DRAWING_Status UI_DRAWING_DEFAULT_drawImage(MICROUI_GraphicsContext* gc, MICROUI_
↪Image* img, jint regionX, jint regionY, jint width, jint height, jint x, jint y, jint␣
↪alpha) {
#if !defined(LLUI_IMAGE_CUSTOM_FORMATS)
   return UI_DRAWING_SOFT_drawImage(gc, img, regionX, regionY, width, height, x, y, alpha);
#else
   return UI_IMAGE_DRAWING_draw(gc, img, regionX, regionY, width, height, x, y, alpha);
#endif
}
```

The define `LLUI_IMAGE_CUSTOM_FORMATS` is set so the implementation of the weak function redirects the image drawing to the image drawers manager (`ui_image_drawing.h`).

UI_IMAGE_DRAWING_draw (available in MicroUI C Module)

```
static const UI_IMAGE_DRAWING_draw_t UI_IMAGE_DRAWING_draw_custom[] = {
   &UI_DRAWING_STUB_drawImage,
   &UI_DRAWING_SOFT_drawImage,
   &UI_IMAGE_DRAWING_draw_custom0,
   &UI_IMAGE_DRAWING_draw_custom1,
   &UI_IMAGE_DRAWING_draw_custom2,
   &UI_IMAGE_DRAWING_draw_custom3,
   &UI_IMAGE_DRAWING_draw_custom4,
   &UI_IMAGE_DRAWING_draw_custom5,
   &UI_IMAGE_DRAWING_draw_custom6,
   &UI_IMAGE_DRAWING_draw_custom7,
};

DRAWING_Status UI_IMAGE_DRAWING_draw(MICROUI_GraphicsContext* gc, MICROUI_Image* img, jint␣
↪regionX, jint regionY, jint width, jint height, jint x, jint y, jint alpha){
   return (*UI_IMAGE_DRAWING_draw_custom[_get_table_index(gc, img)])(gc, img, regionX,␣
↪regionY, width, height, x, y, alpha);
}
```

The implementation in the MicroUI C module redirects the drawing to the expected drawer. The drawer is retrieved thanks to its format (function `_get_table_index()`):

- the format is standard but the destination is not the *display* format: index `0` is returned,
- the format is standard and the destination is the *display* format: index `1` is returned,
- the format is custom: index `2` to `9` is returned,

UI_IMAGE_DRAWING_draw_custom0 (available in MicroUI C Module)

```
// Use the preprocessor 'weak'
__weak DRAWING_Status UI_IMAGE_DRAWING_draw_custom0(MICROUI_GraphicsContext* gc, MICROUI_
↪Image* img, jint regionX, jint regionY, jint width, jint height, jint x, jint y, jint␣
↪alpha){
```

(continues on next page)

(continued from previous page)

```
   return UI_DRAWING_STUB_drawImage(gc, img, regionX, regionY, width, height, x, y, alpha);
}
```

The default implementation of `UI_IMAGE_DRAWING_draw_custom0` (same behavior for `0` to `7`) consists in calling the stub implementation.

UI_DRAWING_STUB_drawImage (available in MicroUI C Module)

```
DRAWING_Status UI_DRAWING_STUB_drawImage(MICROUI_GraphicsContext* gc, MICROUI_Image* img,␣
↪jint regionX, jint regionY, jint width, jint height, jint x, jint y, jint alpha){
  // Set the drawing log flag "not implemented"
  LLUI_DISPLAY_reportError(gc, DRAWING_LOG_NOT_IMPLEMENTED);
  return DRAWING_DONE;
}
```

The implementation only consists in setting the *Drawing log flag* `DRAWING_LOG_NOT_IMPLEMENTED` to notify the application that the drawing has not been performed.

Simulation

Principle

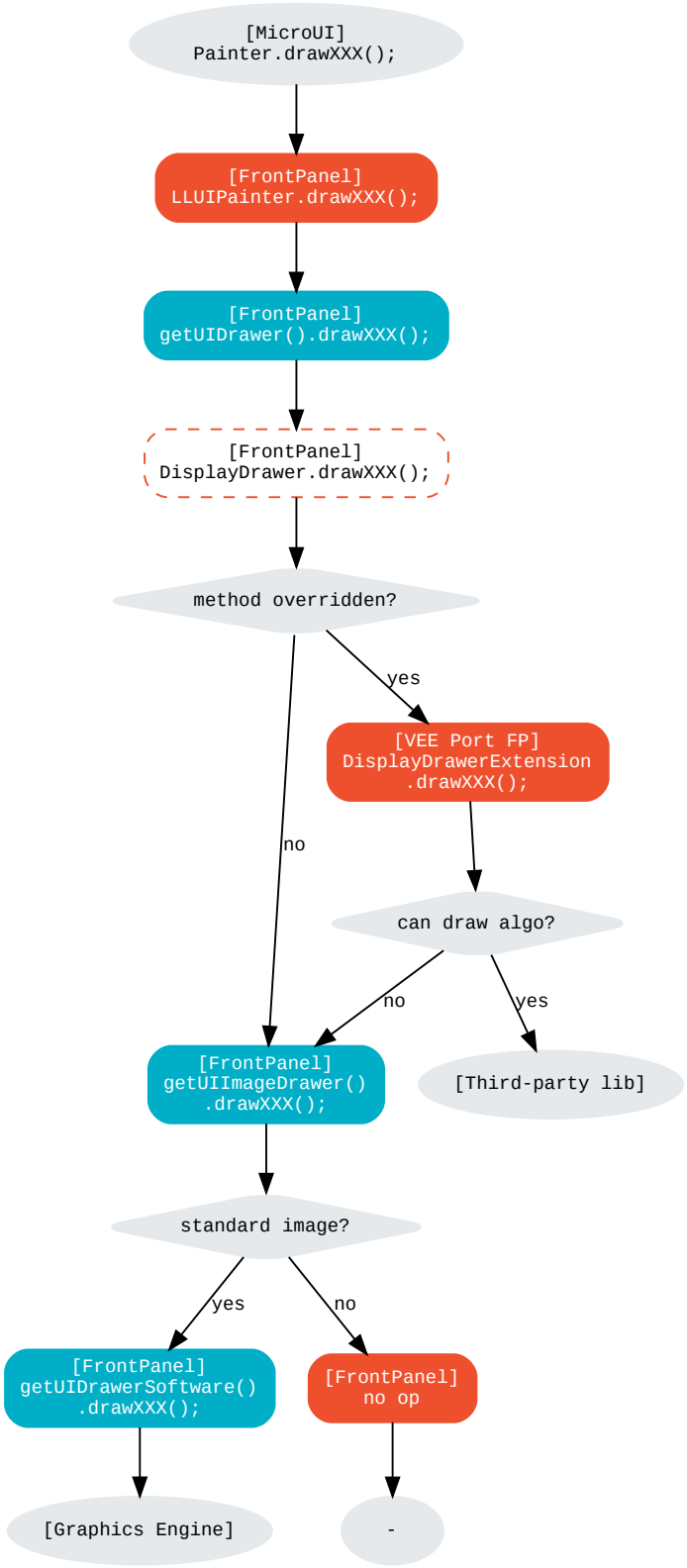
The simulation behavior is similar to the *MicroUI C Module* for the Embedded side.

The *Front Panel* defines support of the drawers based on Java service loader.

Standard Formats Only (Default Implementation)

The default implementation can draw images with a standard format.

The following graph illustrates the drawing of an image:


[MicroUI]
Painter.drawXXX();
[FrontPanel]
LLUIPainter.drawXXX();
[FrontPanel]
getUIDrawer().drawXXX();
[FrontPanel]
DisplayDrawer.drawXXX();
method overridden?
yes
no
[VEE Port FP]
DisplayDrawerExtension
.drawXXX();
can draw algo?
no
yes
[FrontPanel]
getUIImageDrawer()
.drawXXX();
[Third-party lib]
standard image?
yes
no
[FrontPanel]
getUIDrawerSoftware()
.drawXXX();
[FrontPanel]
no op
[Graphics Engine]
-

It is possible to override the image drawers for the standard format in the same way as the custom formats.

Custom Format Support

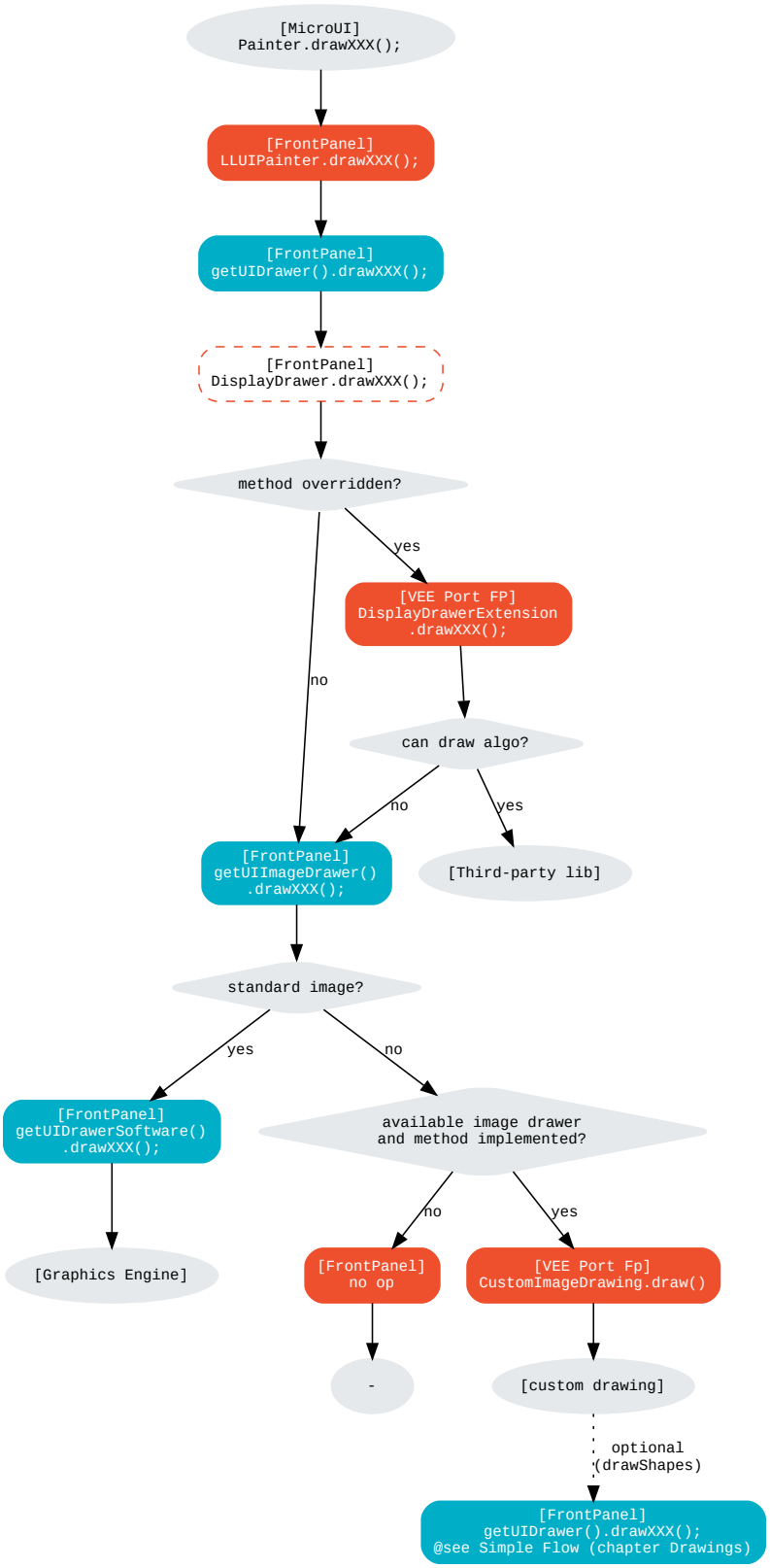
It is possible to draw images with a custom format by implementing the `UIImageDrawing` interface. This advanced use case is available only with MicroUI 3.2 or higher.

The `UIImageDrawing` interface contains one method for each image drawing primitive (draw, copy, region, rotate, scale, flip). Only the necessary methods can be implemented. Each non-implemented method will result in calling the stub implementation.

The method `handledFormat()` must be implemented and returns the managed format.

Once created, the `UIImageDrawing` implementation must be registered as a service.

The following graph illustrates the drawing of an image:


[MicroUI]
Painter.drawXXX();
[FrontPanel]
LLUIPainter.drawXXX();
[FrontPanel]
getUIDrawer().drawXXX();
[FrontPanel]
DisplayDrawer.drawXXX();
method overridden?
yes
no
[VEE Port FP]
DisplayDrawerExtension
.drawXXX();
can draw algo?
no
yes
[FrontPanel]
getUIImageDrawer()
.drawXXX();
[Third-party lib]
standard image?
yes
no
[FrontPanel]
getUIDrawerSoftware()
.drawXXX();
available image drawer
and method implemented?
[Graphics Engine]
no
yes
[FrontPanel]
no op
[VEE Port Fp]
CustomImageDrawing.draw()
-
[custom drawing]
optional
(drawShapes)
[FrontPanel]
getUIDrawer().drawXXX();
@see Simple Flow (chapter Drawings)

Let's implement the image drawer for the *CUSTOM_0* format.

```
public class MyCustomImageDrawer implements UIImageDrawing {

   @Override
   public MicroUIImageFormat handledFormat() {
      return MicroUIImageFormat.MICROUI_IMAGE_FORMAT_CUSTOM_0;
   }

   @Override
   public void draw(MicroUIGraphicsContext gc, MicroUIImage img, int regionX, int regionY,␣
↪int width, int height,
         int x, int y, int alpha) {
      MyCustomImage customImage = (MyCustomImage) img.getImage().getRAWImage();
      customImage.drawOn(gc, regionX, regionY, width, height, x, y, alpha);
   }

}
```

Now, this drawer needs to be registered as a service. This can be achieved by creating a file in the resources of the Front Panel project named `META-INF/services/ej.microui.display.UIImageDrawing`. And its content containing the fully qualified name of the previously created image drawer.

```
com.mycompany.MyCustomImageDrawer
```

It is also possible to declare it programmatically (see where a drawer is registered in the *drawing custom* section):

```
LLUIDisplay.Instance.registerUIImageDrawer(new MyCustomImageDrawer());
```

Installation

The Image Renderer module is part of the MicroUI module and Display module. Install them to be able to use some images.

Use

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

Buffered Image

Overview

MicroUI application can create an image it can draw into: the MicroUI ej.microui.display.BufferedImage. The format of this kind of image is Display (default), Standard, or Custom (see following chapters).

> **Warning:** The output format Standard and Custom depends on the VEE Port capabilities.

To create this kind of image, the Image Loader has to create a buffer in the *images heap* whose size depends on the image data size (see *Image Creation*).

Drawer

A buffered image requires a *drawer*. A drawer is an engine that has the responsibility to:

- allow the application to create Standard and Custom buffered images,
- draw into these images.

The implementation is not the same between the Embedded side and the Simulation. However, the concepts are the same and are described in dedicated chapters.

Formats

Display

This is the format used by default when no format is specified when creating a MicroUI `BufferedImage`.

The image format is the same as the display buffer format; in other words, its number of bits-per-pixel and its pixel bits organization are the same (see chapter *MicroEJ Format: Display*).

- Image creation: the Graphics Engine provides the capacity to create this kind of image; no specific support is required in the VEE Port.
- Draw into the image: the rules to draw into this kind of buffered image are the same as in the display buffer; see:ref:*section_drawings*.
- Draw the image: the rules to draw this kind of buffered image are described in the chapter *image renderer standard*.

Standard

A MicroUI `BufferedImage` can be created specifying a *MicroEJ Format: Standard* or *MicroEJ Format: Grayscale* format.

Note: When the display format is the same as the standard format used to create the buffered image, the rules to create the image, to draw into it and to draw it are the same as the Display format. This chapter describes the use case when the format differs from the *display* format.

Unlike the display format, the VEE Port must feature a *drawer* for each standard format.

- Image creation: the drawer allows the creation of this kind of buffered image; if the VEE Port does not feature a drawer for a specific format, the MicroUI `BufferedImage` cannot be created, and an exception is thrown at runtime.
- Draw into the image: the drawer can implement all MicroUI drawings or just a reduced set; when a drawing is not implemented, a stub implementation (that does nothing) is used.
- Draw the image: the image is *standard*, so its rendering is *standard* also; the rules to draw this kind of buffered image are described in the chapter *image renderer standard* (no extra support needed in the VEE Port).

Custom

A MicroUI `BufferedImage` can be created specifying a *MicroEJ Format: Custom* formats.

Like standard formats, the VEE Port must feature a *drawer* for each custom format. It must also feature an image allocator.

- Image creation: the allocator and drawer allow to create of this kind of buffered image; if the VEE Port does not feature an allocator and a drawer for a specific format, the MicroUI `BufferedImage` cannot be created, and an exception is thrown at runtime.
- Draw into the image: the drawer can implement all MicroUI drawings or just a reduced set; when a drawing is not implemented, a stub implementation (that does nothing) is used.
- Draw the image: the image is *custom*, so its rendering is *custom* also; the rules to draw this kind of buffered image are described in the chapter *image renderer custom*.

MicroUI C Module

Drawer

As described above, a *drawer* allows to create and draw into buffered images whose format differs from the display format. The *MicroUI C module* is designed to manage the notion of drawers: it does not *support* the other formats than display format, but it allows to add some additional drawers.

This support uses several weak functions and tables to redirect the image creation and drawings. When this support is not used (when the VEE Port does not need to support *extra* images), this support can be removed to reduce the footprint (by removing the indirection tables) and increase the performances (by reducing the number of runtime function calls).

In addition to the Display, Standard, and Custom formats, the MicroUI C module implementation introduces the notion of *Single* and *Multiple* formats, more specifically *Single Format Implementation* and *Multiple Formats Implementation*.

Single Format Implementation (Default Implementation)

This MicroUI BufferedImage implementation can only target images with the display format. In other words, the application cannot create a MicroUI BufferedImage with a format different than the display format. This is the most frequent use case, the only one available with MicroUI before version 3.2.

Hint: To select this implementation (to disable the multi formats support), the define `LLUI_GC_SUPPORTED_FORMATS` must be unset or lower than `2`.

This is the default implementation.

Multiple Formats Implementation

This MicroUI BufferedImage implementation allows the creation of a MicroUI BufferedImage whose format differs from the display format. This advanced use case is available only with MicroUI 3.2 or higher.

Hint: To select this implementation, the define `LLUI_GC_SUPPORTED_FORMATS` must be set to `2` or more. Its value defines the available number of *extra* formats the VEE Port features.

The MicroUI C module uses some tables to redirect the image management to the expected *drawer*. There is one table per Abstraction Layer API not to embed all algorithms (a table and its functions are only embedded in the final binary file if and only if the MicroUI drawing method is called). The tables size is dimensioned according to the define value.

To manipulate the tables, the C module uses 0-based index whose value is different from the image format value. For instance, according to the VEE Port capabilities, the support image format ARGB8888 can have the index `1` for a given VEE Port and `2` for another one. This differentiation reduces the size of the tables: when the VEE Port does not support a format, no extra size in the tables is used (no empty cell).

Note: The index `0` is reserved for the *display* format.

A table holds a list of functions for a given algorithm. For instance, the following table allows redirecting the drawing `writePixel` to the drawers `0` to `2`:

```
static const UI_DRAWING_writePixel_t UI_DRAWER_writePixel[] = {
  &UI_DRAWING_writePixel_0,
  &UI_DRAWING_writePixel_1,
#if (LLUI_GC_SUPPORTED_FORMATS > 2)
  &UI_DRAWING_writePixel_2,
#endif
};
```

- `UI_DRAWING_writePixel_0` is the drawing function called when the image format is the display format,
- `UI_DRAWING_writePixel_1` and `UI_DRAWING_writePixel_2` are the drawing functions called for the images whose format are respectively identified by the index `1` and `2` (see *Image Creation* below).

By default, the C module only manages up to 3 formats: the *display* format (index `0`) and two other formats. To add another format, the C module must be customized: look for everywhere the define `LLUI_GC_SUPPORTED_FORMATS` is used and add a new cell in the tables.

Custom Format

A MicroUI BufferedImage can have a *custom* format once the Multiple Formats Implementation is selected. However, third-party support is required to render this kind of image.

Hint: In addition to the `#define LLUI_GC_SUPPORTED_FORMATS`, the `#define LLUI_IMAGE_CUSTOM_FORMATS` must be set. This is the same `define` used to render custom RAW images: see *Custom Format Support*.

Image Creation

Overview

Creating an image consists of several steps. The Graphics Engine manages these steps, which calls four Abstraction Layer APIs. The MicroUI C Module already implements these four LLAPI.

According to the support of multiple drawers, the C module redirects or not these LLAPI to some `ui_drawing.h` functions. The image creation steps are briefly described below; refer to the following chapters for more details.

1. The application asks for the creation of a buffered image.
2. The Graphics Engine calls the LLAPI `LLUI_DISPLAY_IMPL_getDrawerIdentifier()` : this function allows to get a drawer index related to the image format. The index `0` indicates to use the default drawer: the *display* drawer. A positive value indicates a drawer index for all other formats than the display format. A negative index indicates that the VEE Port does not support the image format (in that case, the image creation is refused, and an exception is thrown in the application).
3. Depending on the image format, the Graphics Engine calculates the minimal stride of the image. This stride can be customized to fit the GPU constraint (see *GPU Format Support*) by implementing the LLAPI `LLUI_DISPLAY_IMPL_getNewImageStrideInBytes()` .
4. The Graphics Engine determines the image buffer size according to the image format, size (width and height), and stride (see previous step). This size and the buffer alignment can be adjusted thanks to the LLAPI `LLUI_DISPLAY_IMPL_adjustNewImageCharacteristics()` . The buffer size should be larger or equal to that calculated by the Graphics Engine. If smaller, the Graphics Engine will use the initial value. For a *custom* image, the initial value is 0: the VEE Port must set a positive value; otherwise, the image creation is refused, and an exception is thrown in the application.
5. The Graphics Engine allocates the image buffer according to the values adjusted before (size and alignment).
6. Finally, the Graphics Engine calls the LLAPI `LLUI_DISPLAY_IMPL_initializeNewImage()` that allows the VEE Port to initialize the image buffer (often only useful for custom images).

Single Format Implementation

The MicroUI C module implements the four LLAPI to create only MicroUI BufferedImages with the *display* format.

- `LLUI_DISPLAY_IMPL_getDrawerIdentifier()` : the C module checks if the image format is the *display* format. If yes, it returns the index `0` indicating the Graphics Engine to use the default drawer. If not, it returns a negative index: the image creation is refused.
- It redirects the three last LLAPI to some `ui_drawing.h` functions. These `ui_drawing.h` functions are already implemented as `weak` functions, which allows the VEE Port to implement only the required functions:
  - Implementation of `LLUI_DISPLAY_IMPL_getNewImageStrideInBytes()` calls `UI_DRAWING_getNewImageStrideInBytes()` , the weak function returns the stride given as parameter.
  - Implementation of `LLUI_DISPLAY_IMPL_adjustNewImageCharacteristics()` calls `UI_DRAWING_adjustNewImageCharacteristics()` , the weak function does nothing.
  - Implementation of `LLUI_DISPLAY_IMPL_initializeNewImage()` calls `UI_DRAWING_initializeNewImage()` , the weak function does nothing.

Multiple Formats Implementation

The MicroUI C module implements the four LLAPI to create a MicroUI BufferedImage with any format.

- `LLUI_DISPLAY_IMPL_getDrawerIdentifier()` : the C module checks if the image format is the *display* format. If yes, it returns the index `0``indicating the Graphics Engine to use the default drawer. If not, it calls the function ``UI_DRAWING_is_drawer_1()` and then `UI_DRAWING_is_drawer_2()` . The VEE Port has the responsibility to implement at least one function. The index `1` or `2` will be assigned to the image format according to the VEE Port capabilities. The image creation is refused if no drawer is found for the given format.
- It redirects the three last LLAPI to the associated tables:
  - Implementation of `LLUI_DISPLAY_IMPL_getNewImageStrideInBytes()` calls the functions of the table `UI_DRAWER_getNewImageStrideInBytes[]` , the weak functions return the stride given as parameter.
  - Implementation of `LLUI_DISPLAY_IMPL_adjustNewImageCharacteristics()` calls the functions of the table `UI_DRAWER_adjustNewImageCharacteristics[]` , the weak functions do nothing.
  - Implementation of `LLUI_DISPLAY_IMPL_initializeNewImage()` calls the functions of the table `UI_DRAWER_initializeNewImage[]` , the weak functions do nothing.

Display and Standard Image

For this kind of image, the implementation of the functions `getNewImageStrideInBytes` , `adjustNewImageCharacteristics` and `initializeNewImage` is optional: it mainly depend on the *GPU support*.

Custom Image

For the *custom* images, the implementation of the function `getNewImageStrideInBytes` is optional but the implementation of the functions `adjustNewImageCharacteristics` and `initializeNewImage` is mandatory:

- `adjustNewImageCharacteristics` has to set the image buffer size (the default value is `0` , which is an invalid size); the Graphics Engine will use this value to allocate the image buffer.
- `initializeNewImage` must initialize the custom image buffer.

Image Closing

The BSP has the responsibility to free the third-party resources associated with an image. Most of the time, the resources are allocated and initialized in the implementation of `LLUI_DISPLAY_IMPL_initializeNewImage()` (see above). When the Graphics Engine closes an image, it calls the function `LLUI_DISPLAY_IMPL_freeImageResources()` . Depending on whether multiple drawers are supported, the C module may redirect this LLAPI to some `ui_drawing.h` functions.

Single Format Implementation

The MicroUI C module provides an implementation of the LLAPI. By default, no third-party resources are associated with buffered images. Therefore, `LLUI_DISPLAY_IMPL_freeImageResources()` calls the weak function `UI_DRAWING_freeImageResources()` that does nothing.

If the function `UI_DRAWING_initializeNewImage()` has been implemented in the BSP, the function `UI_DRAWING_freeImageResources()` should be implemented too.

Multiple Formats Implementation

The MicroUI C module implements the LLAPI to let each image manager close the image resources. The implementation of `LLUI_DISPLAY_IMPL_freeImageResources()` calls the functions of the table `UI_DRAWER_freeImageResources[]`, which have default weak implementations that do nothing.

Display and Standard Image

For this kind of image, implementing the function `freeImageResources` is optional: it mainly depends on the *GPU support*.

Custom Image

For the *custom* images, the implementation of the function `freeImageResources` is optional, but often required to free the third-party resources.

Draw into the Image: Display Format

Overview

To draw into a buffered image with the display format, the same concepts to draw in the display buffer are used: the MicroUI Abstraction Layer drawings are redirected to the `ui_drawing.h` functions (see *Drawings* for more details).

The MicroUI C module already implements all `ui_drawing.h` functions, and the drawings are redirected to the *Graphics Engine Software Algorithms*. However the function names are `UI_DRAWING_DEFAULT_drawX()` and not `UI_DRAWING_drawX()`. Thanks to the define `LLUI_GC_SUPPORTED_FORMATS`, the function names are redefined with C macros. This compile-time redirection allows using the same implementation (`UI_DRAWING_DEFAULT_drawX()`) when the multiple formats support is disabled or enabled (when the target is an image with the same format as the display).

The weak implementation of the function `UI_DRAWING_DEFAULT_drawX()` calls *Graphics Engine Software Algorithms*. This implementation allows a GPU or a third-party drawer to perform the rendering (see *Drawings* for more details).

Single Format Implementation

The define `LLUI_GC_SUPPORTED_FORMATS` is unset or lower than `2`; the compile-time redirection is:

```
#define UI_DRAWING_DEFAULT_writePixel UI_DRAWING_writePixel
```

Multiple Formats Implementation

For the images whose format is the display format (index `0`, see *Multiple Formats Implementation*), the compile-time redirection is:

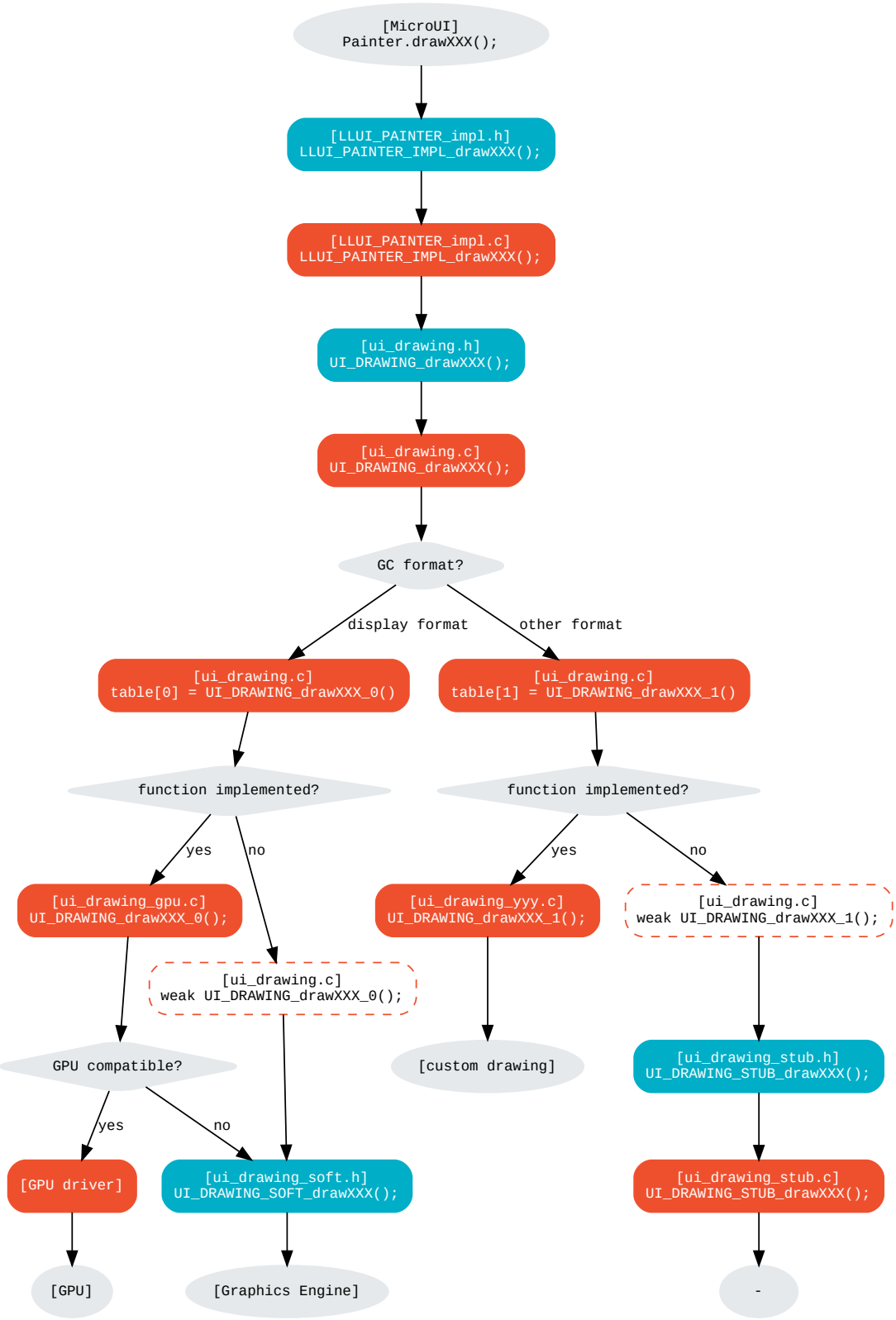
```
#define UI_DRAWING_DEFAULT_writePixel UI_DRAWING_writePixel_0
```

Draw into the Image: Non-Display Format

To draw into a buffered image with a format different than the display format, the *Multiple Formats Implementation* must be selected.

For the images whose format is not the display format (index `1` and `2`), the C module provides weak implementations that do nothing.

The following graph illustrates the drawing of a shape (not an image, see *Draw the Image: Multiple Formats Implementation*):


[MicroUI]
Painter.drawXXX();
[LLUI_PAINTER_impl.h]
LLUI_PAINTER_IMPL_drawXXX();
[LLUI_PAINTER_impl.c]
LLUI_PAINTER_IMPL_drawXXX();
[ui_drawing.h]
UI_DRAWING_drawXXX();
[ui_drawing.c]
UI_DRAWING_drawXXX();
GC format?
display format
other format
[ui_drawing.c]
table[0] = UI_DRAWING_drawXXX_0()
[ui_drawing.c]
table[1] = UI_DRAWING_drawXXX_1()
function implemented?
function implemented?
yes
no
yes
no
[ui_drawing_gpu.c]
UI_DRAWING_drawXXX_0();
[ui_drawing_yyy.c]
UI_DRAWING_drawXXX_1();
[ui_drawing.c]
weak UI_DRAWING_drawXXX_1();
[ui_drawing.c]
weak UI_DRAWING_drawXXX_0();
GPU compatible?
[custom drawing]
[ui_drawing_stub.h]
UI_DRAWING_STUB_drawXXX();
yes
no
[GPU driver]
[ui_drawing_soft.h]
UI_DRAWING_SOFT_drawXXX();
[ui_drawing_stub.c]
UI_DRAWING_STUB_drawXXX();
[GPU]
[Graphics Engine]
-

LLUI_PAINTER_IMPL_drawLine (available in MicroUI C Module)

See *MicroUI C Module*.

UI_DRAWING_drawLine (available in MicroUI C Module)

```
static const UI_DRAWING_drawLine_t UI_DRAWER_drawLine[] = {
  &UI_DRAWING_drawLine_0,
  &UI_DRAWING_drawLine_1,
#if (LLUI_GC_SUPPORTED_FORMATS > 2)
  &UI_DRAWING_drawLine_2,
#endif
};

DRAWING_Status UI_DRAWING_drawLine(MICROUI_GraphicsContext* gc, jint startX, jint startY,␣
↪jint endX, jint endY){
  // Table redirection according to the drawer index
  return (*UI_DRAWER_drawLine[gc->drawer])(gc, startX, startY, endX, endY);
}
```

The implementation in the MicroUI C module redirects the drawing to the expected drawer. The drawer is identified by the index stored in the `MICROUI_GraphicsContext` (index fixed during the image creation).

UI_DRAWING_drawLine_0 (available in MicroUI C Module)

```
#define UI_DRAWING_DEFAULT_drawLine UI_DRAWING_drawLine_0
```

The index `0` is reserved for drawing into the image whose format is the display format (see above). The function name is set thanks to a `define` to reuse the same code between Single and Multiple Formats Implementations.

The behavior after this function is similar to *Custom Implementation*.

UI_DRAWING_drawLine_1 (available in MicroUI C Module)

```
 // use the preprocessor 'weak'
__weak DRAWING_Status UI_DRAWING_drawLine_1(MICROUI_GraphicsContext* gc, jint startX, jint␣
↪startY, jint endX, jint endY){
    // Default behavior: call the stub implementation
  return UI_DRAWING_STUB_drawLine(gc, startX, startY, endX, endY);
}
```

The implementation of the weak function only consists in calling the stub implementation.

UI_DRAWING_STUB_drawLine (available in MicroUI C Module)

```
DRAWING_Status UI_DRAWING_STUB_drawLine(MICROUI_GraphicsContext* gc, jint startX, jint␣
↪startY, jint endX, jint endY){
  // Set the drawing log flag "not implemented"
  LLUI_DISPLAY_reportError(gc, DRAWING_LOG_NOT_IMPLEMENTED);
  return DRAWING_DONE;
}
```

The implementation only consists in setting the *Drawing log* `DRAWING_LOG_NOT_IMPLEMENTED` to notify the application that the drawing has not been performed.

UI_DRAWING_drawLine_1 (to write in the BSP)

```
// this drawer has the index 1
#define UI_DRAWING_IDENTIFIER_A8_FORMAT 1
#define UI_DRAWING_A8_is_drawer CONCAT(UI_DRAWING_is_drawer_, UI_DRAWING_IDENTIFIER_A8_
↪FORMAT)
#define UI_DRAWING_A8_drawLine CONCAT(UI_DRAWING_drawLine_, UI_DRAWING_IDENTIFIER_A8_FORMAT)
```

This example illustrates how to implement the `drawLine` function for an image with the format `A8`. The drawer should be written in its file. However, the MicroUI C module advises not to use directly the name `UI_DRAWING_drawLine_1` but to use this mechanism to redirect at compile-time the call to `UI_DRAWING_A8_drawLine`.

- The define `UI_DRAWING_IDENTIFIER_A8_FORMAT` assignes the index to the A8 drawer, here `1`.
- The define `UI_DRAWING_A8_is_drawer` sets at compile-time the name of the `is_drawer` function, here: `UI_DRAWING_is_drawer_1`.
- The define `UI_DRAWING_A8_drawLine` sets at compile-time the name of the `drawLine` function, here: `UI_DRAWING_drawLine_1`.

UI_DRAWING_A8_is_drawer (to write in the BSP)

```
bool UI_DRAWING_A8_is_drawer(jbyte image_format) {
  return MICROUI_IMAGE_FORMAT_A8 == (MICROUI_ImageFormat)image_format;
}
```

This function (actually `UI_DRAWING_is_drawer_1` thanks to the define, see above) answers `true` when the application tries to open a MicroUI BufferedImage with the format `A8`.

UI_DRAWING_A8_drawLine (to write in the BSP)

```
DRAWING_Status UI_DRAWING_A8_drawLine(MICROUI_GraphicsContext* gc, jint startX, jint startY,␣
↪jint endX, jint endY){

  // Retrieve the destination buffer address
  uint8_t* destination_address = LLUI_DISPLAY_getBufferAddress(&gc->image);

  // Update the next "flush"'s dirty area
  LLUI_DISPLAY_setDrawingLimits(startX, startY, endX, endY);

  // Configure the GPU clip
  THIRD_PARTY_DRAWER_set_clip(startX, startY, endX, endY);

  // Draw the line
  THIRD_PARTY_DRAWER_draw_line(destination_address, startX, startY, endX, endY, (gc->
↪foreground_color & 0xff) /* Use the blue component as opacity level */),

  // Here, consider the drawing as done (not an asynchronous drawing).
  return DRAWING_DONE;
}
```

This function (actually `UI_DRAWING_drawLine_1` thanks to the define, see above) performs the drawing. It is very similar to *Custom Implementation*.

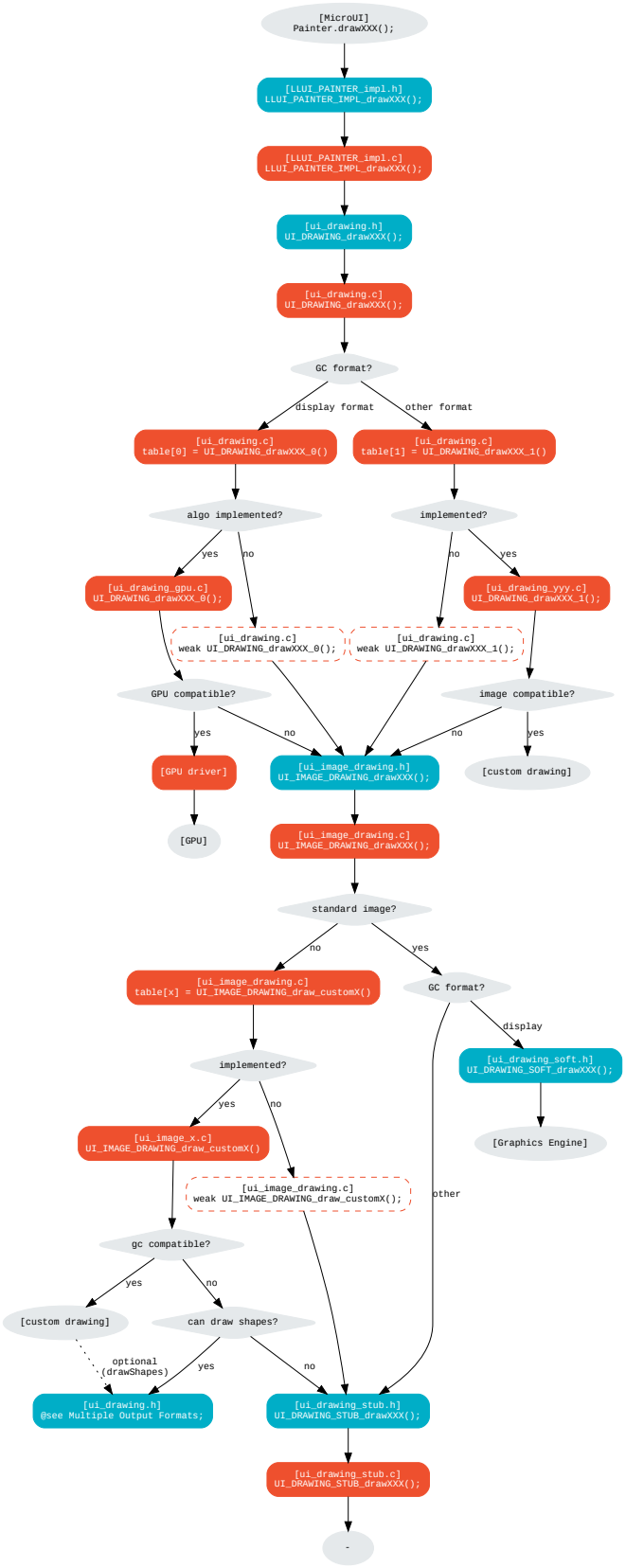
Draw the Image: Single Format Implementation

By definition, the image is a standard image (only display format is allowed), so its drawing is redirected to `ui_image_drawing.h` , see *Standard Formats Only (Default)*.

Draw the Image: Multiple Formats Implementation

Unlike the Single Format Implementation, the destination may be another format than the display format. Consequently, the drawer must check the image format **and** the destination format.

The following graph illustrates the drawing of an image (draw, rotate, or scale) in another image or display buffer (to draw a shape, see *Draw into the Image: Non-Display Format*). This graph gathers both *draw in a custom image* and *render a custom image*.


[MicroUI]
Painter.drawXXX();
[LLUI_PAINTER_impl.h]
LLUI_PAINTER_IMPL_drawXXX();
[LLUI_PAINTER_impl.c]
LLUI_PAINTER_IMPL_drawXXX();
[ui_drawing.h]
UI_DRAWING_drawXXX();
[ui_drawing.c]
UI_DRAWING_drawXXX();
GC format?
display format
other format
[ui_drawing.c]
table[0] = UI_DRAWING_drawXXX_0()
[ui_drawing.c]
table[1] = UI_DRAWING_drawXXX_1()
algo implemented?
implemented?
yes
no
no
yes
[ui_drawing_gpu.c]
UI_DRAWING_drawXXX_0();
[ui_drawing_yyy.c]
UI_DRAWING_drawXXX_1();
[ui_drawing.c]
weak UI_DRAWING_drawXXX_0();
[ui_drawing.c]
weak UI_DRAWING_drawXXX_1();
GPU compatible?
image compatible?
yes
no
no
yes
[GPU driver]
[ui_image_drawing.h]
UI_IMAGE_DRAWING_drawXXX();
[custom drawing]
[GPU]
[ui_image_drawing.c]
UI_IMAGE_DRAWING_drawXXX();
standard image?
no
yes
[ui_image_drawing.c]
table[x] = UI_IMAGE_DRAWING_draw_customX()
GC format?
display
[ui_drawing_soft.h]
UI_DRAWING_SOFT_drawXXX();
implemented?
yes
no
[ui_image_x.c]
UI_IMAGE_DRAWING_draw_customX()
[Graphics Engine]
[ui_image_drawing.c]
weak UI_IMAGE_DRAWING_draw_customX();
other
gc compatible?
yes
no
[custom drawing]
can draw shapes?
optional
(drawShapes)
yes
no
[ui_drawing.h]
@see Multiple Output Formats;
[ui_drawing_stub.h]
UI_DRAWING_STUB_drawXXX();
[ui_drawing_stub.c]
UI_DRAWING_STUB_drawXXX();
-

The following description considers that both previous graphs (*draw in a custom image* and *render a custom image*) have been read and understood. It only describes the *final* use-case: draw a custom image in an unknown destination (unknown destination format):

UI_IMAGE_DRAWING_draw_custom4 (to write in the BSP)

```
// This image drawer manages the custom format 4
#define UI_IMAGE_IDENTIFIER_CMD_FORMAT 4
#define UI_IMAGE_DRAWING_CMD_draw CONCAT(UI_IMAGE_DRAWING_draw_custom_, UI_IMAGE_IDENTIFIER_
↪CMD_FORMAT)

// Macro to map a custom struct "cmd_image_t*" on the MicroUI Image buffer
#define MAP_CMD_ON_IMAGE(image) ((cmd_image_t*) LLUI_DISPLAY_getBufferAddress(image))

DRAWING_Status UI_IMAGE_DRAWING_CMD_draw(MICROUI_GraphicsContext* gc, MICROUI_Image* img,␣
↪jint regionX, jint regionY, jint width, jint height, jint x, jint y, jint alpha){

  // Retrieve the commands list
  cmd_image_t* cmd = MAP_CMD_ON_IMAGE(img);

  for(int i = 0; i < cmd->size; i++) {
    switch (cmd->list[i].kind) {

      case COMMAND_LINE: {

        // Change the graphics context color
        gc->foreground_color = cmd->list[i].color;

        // Draw a line as usual
        UI_DRAWING_drawLine(gc, x + cmd->list[i].args[0], y + cmd->list[i].args[1], x + cmd->
↪list[i].args[2], y + cmd->list[i].args[3]);

        break;
      }

      // All others commands
      // [...]
    }
  }

  // Restore the original color
  gc->foreground_color = original_color;

  return DRAWING_DONE;
}
```

This drawer manages a custom image with a commands buffer (a list of drawings). The image drawing consists in decoding the commands list and calling the standard shapes drawings. This drawer does not need to *recognize* the destination: the drawing of the shapes will do it.

Thanks to the define `UI_IMAGE_IDENTIFIER_CMD_FORMAT`, this drawer uses the custom format `4`.

UI_IMAGE_DRAWING_draw_custom6 (to write in the BSP)

```
// This image drawer manages the custom format 6
#define UI_IMAGE_IDENTIFIER_PROPRIETARY_FORMAT 6
#define UI_IMAGE_DRAWING_PROPRIETARY_draw CONCAT(UI_IMAGE_DRAWING_draw_custom_, UI_IMAGE_
↪IDENTIFIER_PROPRIETARY_FORMAT)

DRAWING_Status UI_IMAGE_DRAWING_PROPRIETARY_draw(MICROUI_GraphicsContext* gc, MICROUI_Image*␣
↪img, jint regionX, jint regionY, jint width, jint height, jint x, jint y, jint alpha){

  DRAWING_Status ret;

  // Can only draw in an image with the same format as display
  if (LLUI_DISPLAY_isDisplayFormat(gc->image.format)) {
    // Call a third-party library
    THIRD_PARTY_LIB_draw_image([...]);
    ret = DRAWING_DONE; // or DRAWING_RUNNING
  }
  else {
    // Cannot draw the image: call stub implementation
    ret = UI_DRAWING_STUB_drawImage(gc, img, regionX, regionY, width, height, x, y, alpha);
  }

  return ret;
}
```

This drawer manages an image whose format is *proprietary* . This example considers that the third-party library can only draw the image in a buffer with the display format. Otherwise, the drawing is canceled, and the stub implementation is used.

Thanks to the define `UI_IMAGE_IDENTIFIER_PROPRIETARY_FORMAT` , this drawer uses the custom format `6` .

Extended C Modules

MicroVG enables a custom format for the Buffered Vector Image. It uses the mechanisms described above and can be used as an example. See *C Modules*.

The drawings in the custom format *BVI* are implemented into the file `ui_drawing_bvi.c` .

Simulation

The simulation behavior is similar to the *MicroUI C Module* for the Embedded side.

Drawer

It is possible to draw in images with a format different than the display one by implementing the `UIDrawing` interface.

This interface contains one method for each drawing primitive. Only the necessary methods need be implemented. Each non-implemented method will result in calling the stub implementation.

The method `handledFormat()` must be implemented and returns the managed format.

Once created, the `UIDrawing` implementation must be registered as a service.

Creating an image with a standard format (different from the display one) is supported in the Front Panel as long as a `UIDrawing` is defined for this format.

Creating an image with a custom format also requires implementing the *image creation* in the VEE Port.

Image Creation

Creating images with a custom format is possible by implementing the `BufferedImageProvider` interface.

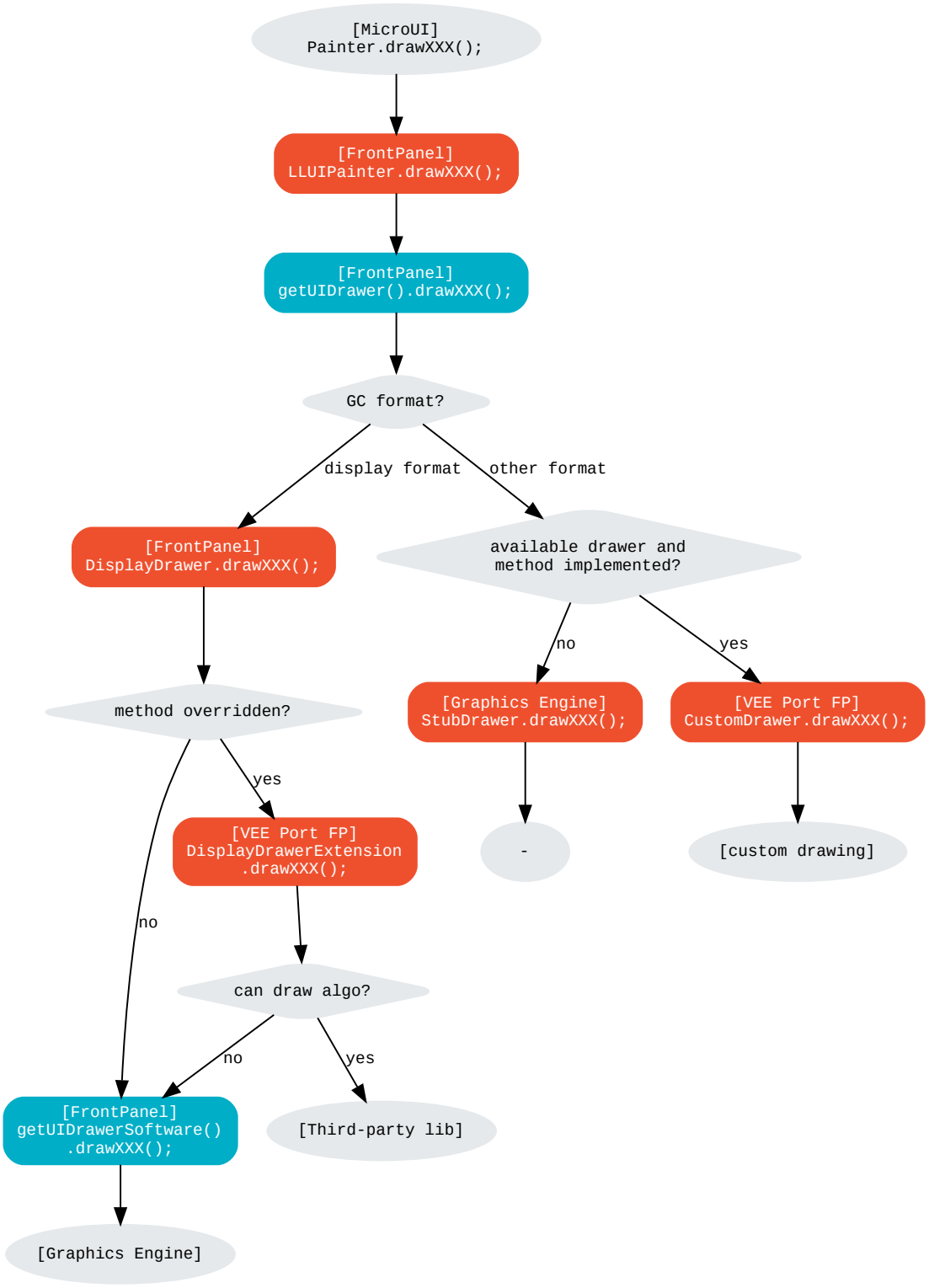
This interface extends `UIDrawing` and `UIImageDrawing` and contains a method `newBufferedImage()`. This method needs to be implemented to create the custom image. It must return an object representing the image. This object will be available in the drawing methods (*Drawer*).

The method `handledFormat()` must be implemented and returns the managed format.

Once created, the `BufferedImageProvider` implementation must be registered as a service.

Draw into the Image: Non-Display Format

The following graph illustrates the drawing of a shape (not an image, see *Draw the Image: Multiple Formats Implementation*):


[MicroUI]
Painter.drawXXX();
[FrontPanel]
LLUIPainter.drawXXX();
[FrontPanel]
getUIDrawer().drawXXX();
GC format?
display format
other format
[FrontPanel]
DisplayDrawer.drawXXX();
available drawer and
method implemented?
no
yes
method overridden?
[Graphics Engine]
StubDrawer.drawXXX();
[VEE Port FP]
CustomDrawer.drawXXX();
yes
[VEE Port FP]
DisplayDrawerExtension
.drawXXX();
-
[custom drawing]
no
can draw algo?
no
yes
[FrontPanel]
getUIDrawerSoftware()
.drawXXX();
[Third-party lib]
[Graphics Engine]

Standard Format

Let's implement the drawer for the *ARGB8888* format (with only the draw line primitive).

```
public class MyARGB8888ImageDrawer implements UIDrawing {

   @Override
   public MicroUIImageFormat handledFormat() {
      return MicroUIImageFormat.MICROUI_IMAGE_FORMAT_ARGB8888;
   }

   @Override
   public void drawLine(MicroUIGraphicsContext gc, int x1, int y1, int x2, int y2) {
     Image image = gc.getImage();
     image.drawLine(x1, y1, x2, y2, gc.getMicroUIColor());
   }

}
```

Now, this drawer needs to be registered as a service. This can be achieved by creating a file in the resources of the Front Panel project named `META-INF/services/ej.microui.display.UIDrawing`. And its content containing the fully qualified name of the previously created image drawer.

```
com.mycompany.MyARGB8888ImageDrawer
```

It is also possible to declare it programmatically (see where a drawer is registered in the *drawing custom* section):

```
LLUIDisplay.Instance.registerUIDrawer(new MyARGB8888ImageDrawer());
```

Custom Format

Let's implement the buffered image provider for the *CUSTOM_0* format (with only the draw line primitive).

```
public class MyCustom0ImageProvider implements BufferedImageProvider {

   @Override
   public MicroUIImageFormat handledFormat() {
      return MicroUIImageFormat.MICROUI_IMAGE_FORMAT_CUSTOM_0;
   }

   @Override
   public Object newBufferedImage(int width, int height)
     // Create the image.
     return new CustomImage(width, height);
   }

   @Override
   public void drawLine(MicroUIGraphicsContext gc, int x1, int y1, int x2, int y2) {
     // Draw in the image.
     CustomImage customImage = (CustomImage) gc.getImage().getRAWImage();
     customImage.drawLine(x1, y1, x2, y2, gc.getMicroUIColor());
   }
```

(continues on next page)

(continued from previous page)

```
   @Override
   public void draw(MicroUIGraphicsContext gc, MicroUIImage img, int regionX, int regionY,␣
↪int width, int height,
         int x, int y, int alpha) {
     // Draw the image in another buffer.
     MyCustomImage customImage = (MyCustomImage) img.getImage().getRAWImage();
     customImage.drawOn(gc, regionX, regionY, width, height, x, y, alpha);
   }
}
```

Now, this buffered image provider needs to be registered as a service. This can be achieved by creating a file in the resources of the Front Panel project named `META-INF/services/ej.microui.display.BufferedImageProvider`. And its content containing the fully qualified name of the previously created buffered image provider.

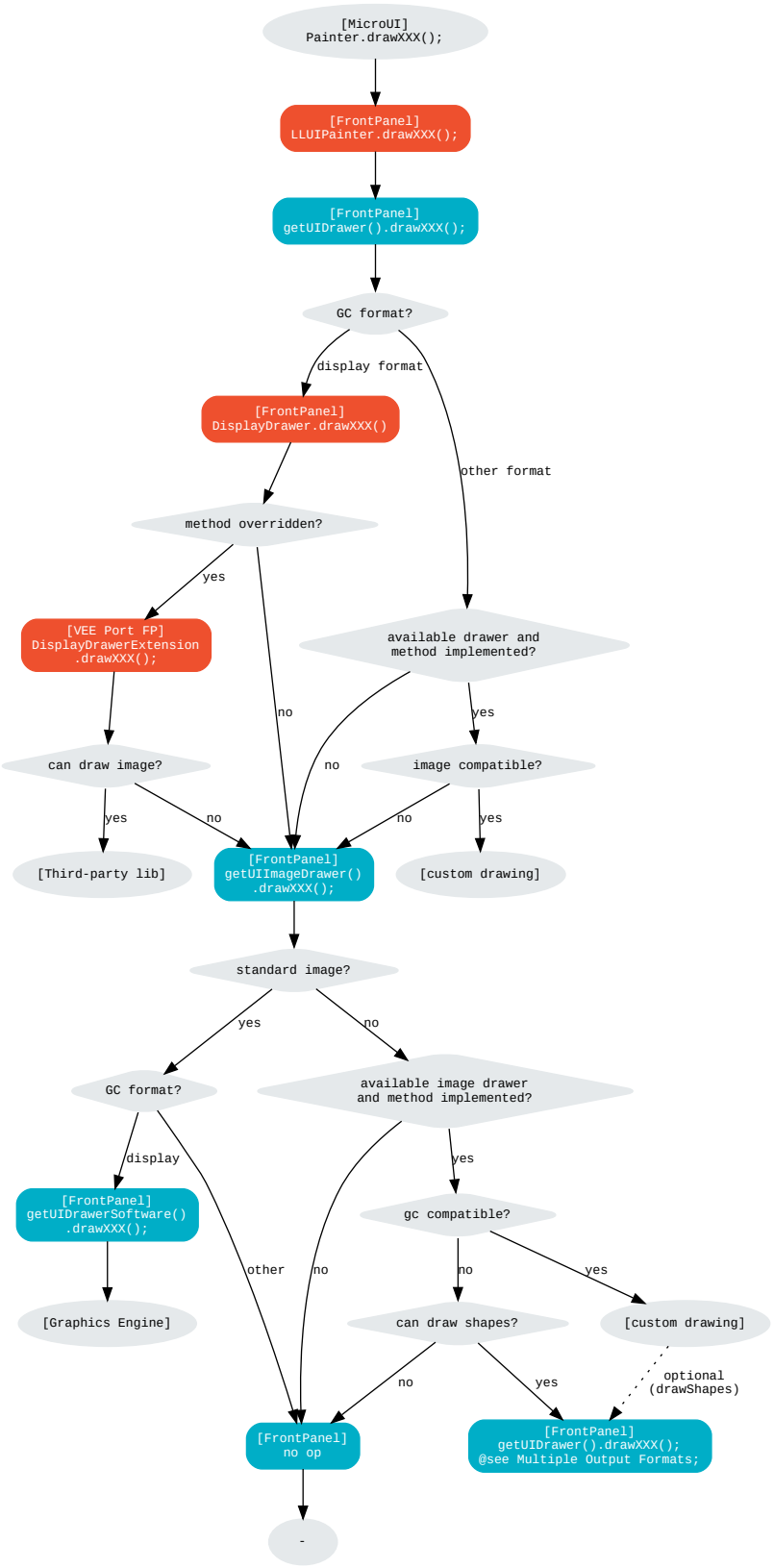
```
com.mycompany.MyCustom0ImageProvider
```

It is also possible to declare it programmatically (see where a drawer is registered in the *drawing custom* section):

```
LLUIDisplay.Instance.registerBufferedImageProvider(new MyCustom0ImageProvider());
```

Draw the Image: Multiple Formats Implementation

The following graph illustrates the drawing of an image (draw, rotate, or scale) in another image or display buffer (to draw a shape, see *Draw into the Image: Non-Display Format*). This graph gathers both graphs *draw in a custom image* and *render a custom image*.


[MicroUI]
Painter.drawXXX();
[FrontPanel]
LLUIPainter.drawXXX();
[FrontPanel]
getUIDrawer().drawXXX();
GC format?
display format
other format
[FrontPanel]
DisplayDrawer.drawXXX()
method overridden?
yes
no
[VEE Port FP]
DisplayDrawerExtension
.drawXXX();
available drawer and
method implemented?
yes
no
can draw image?
image compatible?
yes
no
no
yes
[Third-party lib]
[FrontPanel]
getUIImageDrawer()
.drawXXX();
[custom drawing]
standard image?
yes
no
GC format?
available image drawer
and method implemented?
display
other
no
yes
[FrontPanel]
getUIDrawerSoftware()
.drawXXX();
gc compatible?
no
yes
[Graphics Engine]
can draw shapes?
[custom drawing]
no
yes
optional
(drawShapes)
[FrontPanel]
no op
[FrontPanel]
getUIDrawer().drawXXX();
@see Multiple Output Formats;
-

Dependencies

- MicroUI module (see *MicroUI*),
- Display module (see *Display*).

Installation

The BufferedImage module is part of the MicroUI module and Display module. Install them to be able to use some buffered images.

Use

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

6.14.11 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 VEE Port-independent format. See *Font Designer*.
- A “Font Generator” module, for converting fonts from the VEE Port-independent format into a VEE Port-dependent format.
- The “Font Renderer” module which decodes and renders at application runtime the VEE Port-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

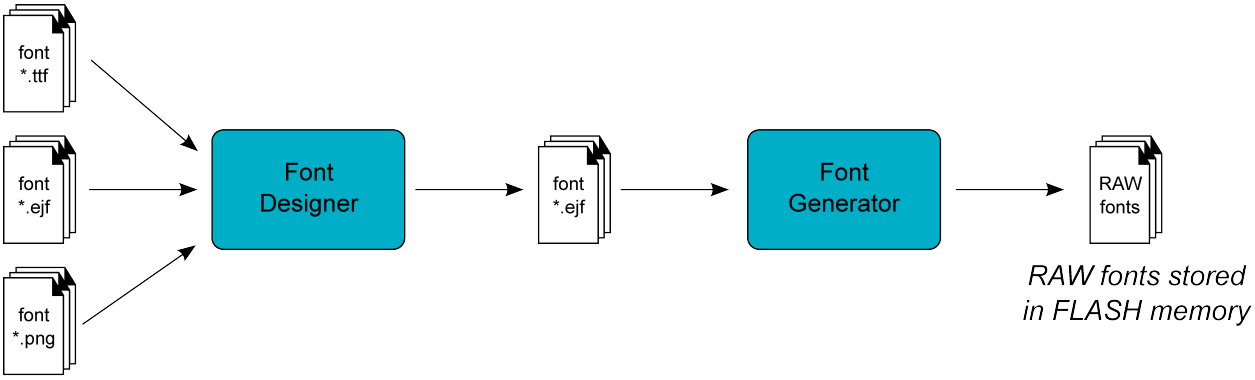


Fig. 53: Font Generation

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 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.

Dependencies

- MicroUI module (see *MicroUI*),
- Display module (see *Display*).

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 (see official Unicode website) 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.

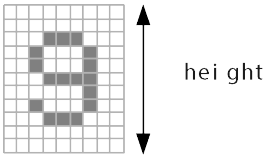


Fig. 54: Font Height

- 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.

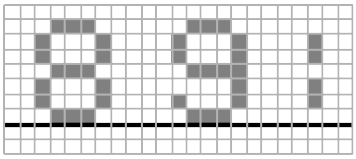


Fig. 55: Font baseline

- 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:

  مكتب البحرم

  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

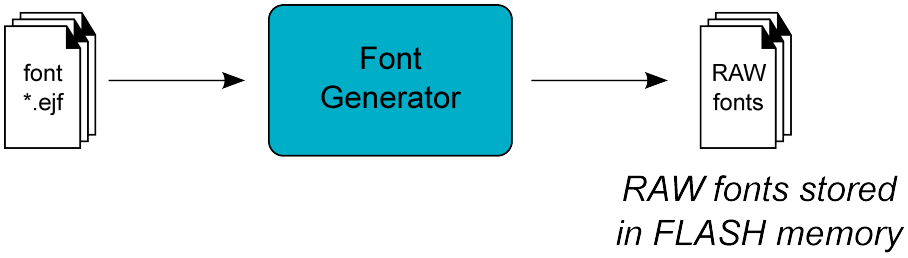


Fig. 56: Font Generator Principle

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 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 22: Font 1-BPP RAW Conversion

Grayscale Ranges	Transparency Levels
0x00 to 0x7f	0x1
0x80 to 0xff	0x0

Table 23: 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 24: 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 application launcher (see *Standalone Application Options*). However, all files in application classpath with suffix `.fonts.list` are automatically parsed by the Font Generator tool.

Each line can have optional parameters (separated by a ':') which define some ranges of characters to embed in the final raw file, and the required pixel depth. By default, all characters available in the input font file are embedded, and the pixel depth is 1 (i.e 1 bit-per-pixel).

Note: See *Configuration File* to understand the list file grammar.

Selecting only a specific set of characters to embed reduces the memory footprint. There are two ways to specify a character range: the custom range and the known range. Several ranges can be specified, separated by ";".

Below is an example of a list file for the Font Generator:

Listing 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.

- Refer to the chapter *Fonts* to have more details how to use this kind of resources.
- Refer to the chapter *External Resources* to have more details how the Font Engine manages this kind of resources.

Installation

The Font Generator module is an additional tool for MicroUI library. When the MicroUI module is installed, install this module in order to be able to embed some additional fonts with the application.

If the module is not installed, the platform user will not be able to embed a new font with his/her application. He/she will be only able to use the system fonts specified during the MicroUI initialization step (see *Static Initialization*).

In the VEE Port configuration file, check `UI` > `Font Generator` to install the Font Generator module.

Use

In order to be able to embed ready-to-be-displayed fonts, you must activate the fonts conversion feature and specify the fonts configuration file.

Refer to the chapter *Standalone Application Options* (`Libraries` > `MicroUI` > `Font`) for more information about specifying the fonts configuration file.

Font Loader

Principle

The Font Loader is a module of the MicroUI runtime that loads font data (precomputed bitmaps of glyphs) ready to be displayed. The font data must be stored as a resource (in EJF raw format). Typically, these resources are generated by the *Font Generator* and embedded as internal resources or loaded from external memories (*External Resources* loader).

External Resources

Memory Management

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 the application is using the font. As soon as the application uses another external font, new font replaces the old one. Then, on 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 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.

Configuration File

Like internal resources, the Font Generator uses a *configuration file* (also called the "list file") for describing fonts that need to be processed. The list file must be specified in the application launcher (see *Standalone Application Options*). However, all the files in the application classpath with the suffix `.fontsext.list` are automatically parsed by the Font Generator tool.

Process

This chapter describes the steps to setup the loading of an external resource from the application:

1. Add the font to the application project resources (typically in the source folder `src/main/resources` and in the package `fonts`).
2. Create / open the configuration file (e.g. `application.fontsext.list`).
3. Add the relative path of the font and, at least, its output format (e.g. `/fonts/myFont.ejf::4`, see *Fonts*).
4. Build the application: the Font Generator converts the font in RAW format in the external resources folder (`[application_output_folder]/externalResources`).
5. Deploy the external resources to the external memory (SDCard, flash, etc.) of the device.
6. (optional) Configure the *External Resources Loader* to load from this source.
7. Build the application and run it on the device.
8. The application loads the external resource using Font.getFont(String).
9. The font loader looks for the font and only reads the font header.
10. (optional) The external resource is closed if the external resource is inside the CPU addresses' space range.
11. The application can use the font.
12. The external resource is never closed: the font's bytes are copied in RAM on demand (drawString, etc.).

Simulation

The Simulator automatically manages the external resources like internal resources. All images listed in `*.fontsext.list` files are copied in the external resources folder, and this folder is added to the Simulator's classpath.

Backward Compatibility

As explained *here*, the notion of `Dynamic` styles and the style `UNDERLINED` are not supported anymore by MicroUI 3. However, an external font may have been generated with an older version of the Font Generator; consequently, the generated file can hold the `Dynamic` style. The Font Renderer can load these old versions of fonts. However, there are some runtime limitations:

- The `Dynamic` styles are ignored. The font is drawn without any dynamic algorithm.
- The font style (the style returned by Font.isBold() and Font.isItalic()) is the `Dynamic` style. For instance, when a font holds the style *bold* as dynamic style and the style *italic* as built-in style, the font is considered as *bold* + *italic*; even if the style *bold* is not rendered.

Installation

The Font 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.

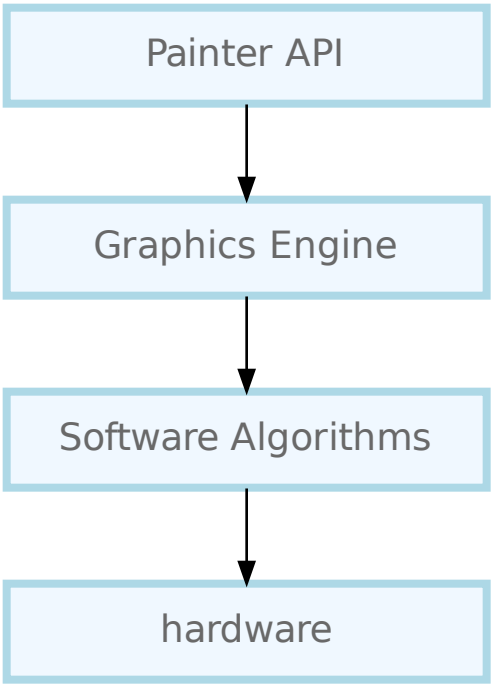
Font Renderer

Principle

The Font Renderer is a module of the MicroUI runtime that reads and draws the fonts.

Functional Description

The Graphics Engine redirects all MicroUI font drawings to the internal software algorithms. There is no indirection to a set of Abstraction Layer API.



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.

6.14.12 C Modules

Principle

Several C modules implement the UI Pack's Abstraction Layer APIs. Some are generic, and some are VEE Port dependent (more precisely: GPU-dependent). The generic modules provide header files to be extended by the specific modules. The generic C modules are available on the *Central Repository* and the specific C modules on the *Developer Repository*.

The following picture illustrates the available C modules, and the following chapters explain the aim and relations of each C module.

Note: It is a simplified view: all sources and headers files of each C module are not visible.

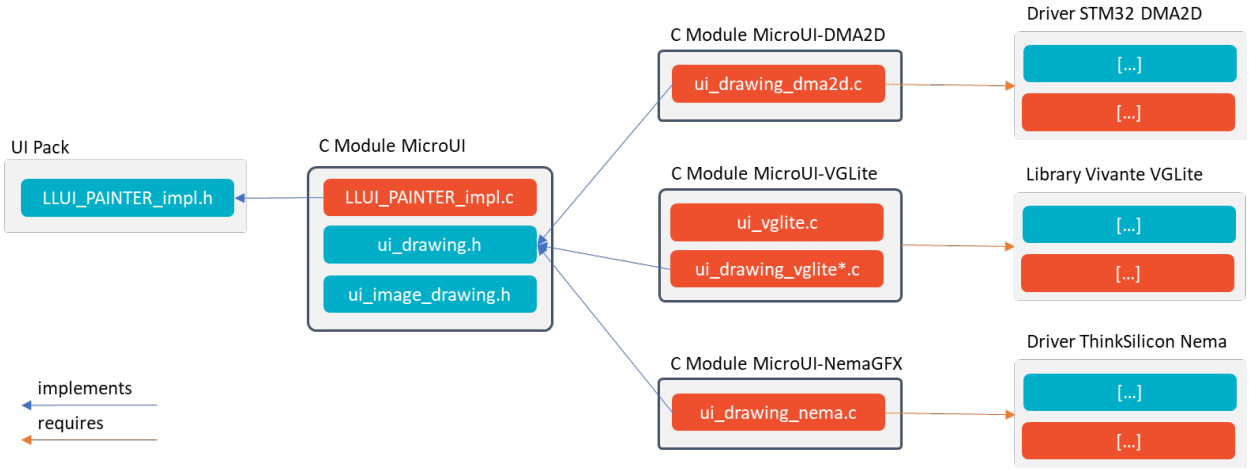


Fig. 57: MicroUI C Modules

UI Pack

The UI Pack provides a header file to implement the MicroUI drawings: `LLUI_PAINTER_impl.h`. See *UI Pack* chapter to have more information.

The UI Pack and its header files are available on the Central Repository: https://repository.microej.com/modules/com/microej/pack/ui/ui-pack/.

C Module: MicroUI

This generic C module provides an implementation of all MicroUI Abstraction Layer APIs. This C module is **mandatory** and provides default implementations for:

- `LLUI_PAINTER_impl.h` and `LLDW_PAINTER_impl.h` : see *Abstraction Layer API* (it manages the synchronization with the Graphics Engine and redirects all drawings to an implementation of `ui_drawing.h`),
- Images heap allocator: see *Images Heap*,
- `LLUI_INPUT_IMPL_log_queue_xxx()` : see *Event Buffer*.

See *UI Pack* chapter to have more information. This C module is available on the *Central Repository*: com.microej.clibrary.llimpl#microui.

C Module: MicroUI Over DMA2D

This C module is a specific implementation of the C module MicroUI over STM32 DMA2D (Chrom-ART Graphics Accelerator). It implements a set of drawings over the official Chrom-ART Graphics Accelerator API: `ui_drawing_dma2d.c` .

Accelerated Drawings

The following table describes the accelerated features:

Feature	Comment
Fill rectangle	
Draw image	ARGB8888, RGB888, RGB565, ARGB1555, ARGB4444, A8, A4
Flush (copy)	Copy of data from back buffer to frame buffer

This C module is available on the *Central Repository*: com.microej.clibrary.llimpl#display-dma2d.

C Module: MicroUI Over VGLite

This C module is a specific implementation of the C module MicroUI over Vivante VGLite. It implements a set of drawings over the official Vivante VGLite library (that targets some GPU with vector graphics acceleration): `ui_drawing_vglite.c` .

This C module also provides a set of header files (and their implementations) to manipulate some MicroUI concepts over the VGLite library: image management, path format, etc.: `ui_vglite.h` and `ui_drawing_vglite_path.h` .

Options

This C module provides some drawing algorithms that are disabled by default.

- The rendering time of a simple shape with the GPU (time in the VG-Lite library + GPU setup time + rendering time) is longer than with software rendering. To enable the hardware rendering for simple shapes, uncomment the definition of `VGLITE_USE_GPU_FOR_SIMPLE_DRAWINGS` in `ui_vglite_configuration.h` .
- The rendering time of an RGB565 image into an RGB565 buffer without applying an opacity (alpha == 0xff) is longer than with software rendering (as this kind of drawing consists in performing a mere memory copy). To enable the hardware rendering for RGB565 images, uncomment the definition of `VGLITE_USE_GPU_FOR_RGB565_IMAGES` in `ui_vglite_configuration.h` .
- ARGB8888, ARGB1555, and ARGB4444 transparent images may not be compatible with some revisions of the VG-Lite GPU. Older GPU revisions do not render transparent images correctly because the pre-multiplication of the pixel opacity is not propagated to the pixel color components. To force the hardware rendering for non-premultiplied transparent images when the VG-Lite GPU is not compatible, uncomment the definition of `VGLITE_USE_GPU_FOR_TRANSPARENT_IMAGES` in `ui_vglite_configuration.h` . Note that this limitation does not concern the VG-Lite GPU compatible with non-premultiplied transparent images and the A8/A4 formats.

Accelerated Drawings

The following table describes the accelerated features:

Feature	Comment
Draw line	Disabled by default (see above)
Fill rectangle	Disabled by default (see above)
Draw rounded rectangle	Disabled by default (see above)
Fill rounded rectangle	
Draw circle arc	Disabled by default (see above)
Fill circle arc	
Draw ellipse arc	Disabled by default (see above)
Fill ellipse arc	
Draw ellipse arc	Disabled by default (see above)
Fill ellipse arc	
Draw circle	Disabled by default (see above)
Fill circle	
Draw image	ARGB8888_PRE, ARGB1555_PRE, ARGB4444_PRE, RGB565, A8, A4 ARGB8888, ARGB1555, ARGB4444 (see above)
Draw thick faded point	Only with fade <= 1
Draw thick faded line	Only with fade <= 1
Draw thick faded circle	Only with fade <= 1
Draw thick faded circle arc	Only with fade <= 1
Draw thick faded ellipse	Only with fade <= 1
Draw thick line	
Draw thick circle	
Draw thick circle arc	
Draw thick ellipse	
Draw flipped image	See draw image
Draw rotated image	See draw image
Draw scaled image	See draw image

This C module is available on the *Developer Repository*: com.microej.clibrary.llimpl#microui-vglite.

Compatibility With MCU i.MX RT595

The previous versions of the C Module Over VGLite included an implementation of the Low-Level API `LLUI_DISPLAY_impl.h`. This support has been extracted into a dedicated C Module since the version `7.0.0`. The dedicated C Module is available on the *Developer Repository*: com.microej.clibrary.llimpl#microui-mimxrt595-evk.

Only the C Module com.microej.clibrary.llimpl#microui-vglite is useful to target the Vivante VG-Lite GPU to perform the MicroUI and MicroVG drawings. The C Module com.microej.clibrary.llimpl#microui-mimxrt595-evk only gives an example of an implementation compatible with the MCU i.MX RT595 MCU.

Note: For more information, see the *migration notes*.

C Module: MicroUI Over NemaGFX

This C module is a specific implementation of the C module MicroUI over Think Silicon NemaGFX. It implements a set of drawings over the official NemaGFX library (that targets some GPU with vector graphics acceleration): `ui_drawing_nema.c`.

Implementation

The MicroUI Graphics Engine waits the end of the asynchronous drawings (performed by the GPU). The VEE Port must unlock this waiting by using one of these two solutions:

- Mode *interrupt*: the GPU interrupt routine has to call the function `UI_DRAWING_NEMA_post_operation()` (the GPU interrupt routine is often written in the same file than the implementation of `nema_sys_init()`).
- Mode *task*: the VEE Port has to add a dedicated task that will wait the end of the drawings.

The mode *interrupt* is enabled by default. To use the mode *task*, comment the define `NEMA_INTERRUPT_MODE` in `ui_drawing_nema_configuration.h`

Note: Retrieve more details in the `#define NEMA_INTERRUPT_MODE` documentation.

Options

This C module provides some drawing algorithms that are disabled by default.

- The rendering time of a simple shape with the GPU (time in the NemaGFX library + GPU setup time + rendering time) is longer than with software rendering. To enable the hardware rendering for simple shapes, uncomment the definition of `ENABLE_SIMPLE_LINES` in `ui_drawing_nema_configuration.h`.
- To draw a shape, the GPU uses the commands list. For rectangular shapes (draw/fill rectangles and images), the maximum list size is fixed (around 300 bytes). For the other shapes (circle, etc.) the list increases according the shape size (dynamic shape): several blocks of 1024 bytes and 40 bytes are allocated and never freed. By default, the dynamic shapes are disabled and the software algorithms are used instead. To enable the hardware rendering for dynamic shapes, uncomment the definition of `ENABLE_DYNAMIC_SHAPES` in `ui_drawing_nema_configuration.h`.
- Some GPU might not be able to render the images in specific memories. Comment the define `ENABLE_IMAGE_ROTATION` in `ui_drawing_nema_configuration.h` to not use the GPU to render the rotated images.

Accelerated Drawings

The following table describes the accelerated features:

Feature	Comment
Draw line	
Draw horizontal line	Disabled by default (see above: ENABLE_SIMPLE_LINES)
Draw vertical line	Disabled by default (see above: ENABLE_SIMPLE_LINES)
Draw rectangle	Disabled by default (see above: ENABLE_SIMPLE_LINES)
Fill rectangle	
Draw rounded rectangle	Disabled by default (see above: ENABLE_DYNAMIC_SHAPES)
Fill rounded rectangle	Disabled by default (see above: ENABLE_DYNAMIC_SHAPES)
Draw circle	Disabled by default (see above: ENABLE_DYNAMIC_SHAPES)
Fill circle	Disabled by default (see above: ENABLE_DYNAMIC_SHAPES)
Draw image	ARGB8888, RGB565, A8
Draw thick faded line	Only with fade <= 1
Draw thick faded circle	Only with fade <= 1, disabled by default (see above: ENABLE_DYNAMIC_SHAPES)
Draw thick line	
Draw thick circle	Disabled by default (see above: ENABLE_DYNAMIC_SHAPES)
Draw rotated image	See draw image
Draw scaled image	See draw image

This C module is available on the *Developer Repository*: com.microej.clibrary.llimpl#microui-nemagfx.

Compatibility

The compatibility between the components (Packs, C modules, and Libraries) is described in the *C Modules*.

6.14.13 Simulation

Principle

The graphical user interface uses the Front Panel mock (see *Front Panel Mock*) and some extensions (widgets) to simulate the user interactions. It is the equivalent of the three embedded modules (Display, Input and LED) of the VEE Port (see *MicroUI*).

The Front Panel enhances the development environment by allowing User Interface applications to be designed and tested on the computer rather than on the target device (which may not yet be built). The mock interacts with the user's computer in two ways:

- output: LEDs, graphical displays
- input: buttons, joystick, touch, haptic sensors

Note: This chapter completes the notions described in *Front Panel Mock* chapter.

Module Dependencies

The Front Panel project is a regular MicroEJ Module project. Its module.ivy file should look like this example:

```
<ivy-module version="2.0" xmlns:ea="http://www.easyant.org" xmlns:ej="https://developer.
↪microej.com" ej:version="2.0.0">
   <info organisation="com.mycompany" module="examplePanel" status="integration" revision="1.
↪0.0"/>

   <configurations defaultconfmapping="default->default;provided->provided">
      <conf name="default" visibility="public" description="Runtime dependencies to other␣
↪artifacts"/>
      <conf name="provided" visibility="public" description="Compile-time dependencies to␣
↪APIs provided by the platform"/>
   </configurations>

   <dependencies>
      <dependency org="ej.tool.frontpanel" name="framework" rev="1.1.1"/>
      <dependency org="ej.tool.frontpanel" name="widget" rev="3.0.0"/>
   </dependencies>
</ivy-module>
```

By default, the project depends on the Front Panel Framework which only contains the Front Panel core classes and which does not provide any Front Panel Widgets (see *Module Dependencies*). To add interactive Front Panel Widgets (typically a simulated display and input devices), add the library that provides compatible Front Panel Widgets with the Graphics Engine:

```
<dependency org="ej.tool.frontpanel" name="widget" rev="3.0.0"/>
```

Note: The life cycle of this library is different than the UI pack's one, see *Front Panel API*.

Source code for Front Panel Widgets is available by expanding the library from the project view.



Fig. 58: Front Panel Widgets

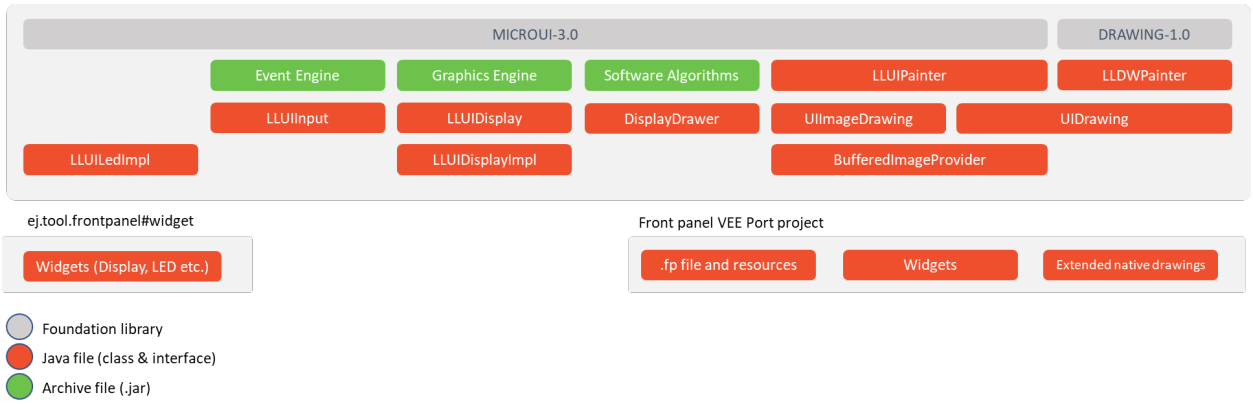
To implement UI Pack extensions for the simulator (custom widgets compatible with the Graphics Engine, custom drawings, etc.), add the Front Panel extension API from the UI Pack (set the version used by the VEE Port):

```
<dependencies>
   <dependency org="com.microej.pack.ui" name="ui-pack" rev="[UI Pack version]">
      <artifact name="frontpanel" type="jar"/>
   </dependency>
</dependencies>
```

Warning: This extension is built for each UI Pack version. By consequence, a Front Panel project is done for a VEE Port built with the same UI Pack. When the UI Pack mismatch, some errors may occur during the Front Panel project export step, during the VEE Port build, and/or during the application runtime. The latest current pack version is 13.7.2.

MicroUI Implementation

As described *here*, the Front Panel uses an equivalent of embedded side's header files that implement MicroUI native methods.



This set of classes and interfaces is available in the module com.microej.pack.ui#ui-pack. It offers the same capacity to override some built-in drawing algorithms (internal Graphics Engine drawing algorithms), to add some custom drawing algorithms, to manipulate the MicroUI concepts (GraphicsContext, Image, etc.) in the Front Panel project, etc.

- The interface `ej.microui.display.LLUIDisplay` represents the MicroUI Graphics Engine (MicroUI framework). It provides methods to map MicroUI byte arrays in MicroUI Graphics Context objects, manipulate MicroUI colors, clip, etc. An instance of this framework is available via the field `Instance`.
- The interface `ej.microui.display.LLUIDisplayImpl` all methods required by MicroUI implementation to be compatible with the MicroUI Display class implementation. See *Widget Display*.
- The class `ej.microui.display.LLUIPainter` implements all MicroUI drawing natives. It defines some interfaces and classes to manipulate the MicroUI concepts (GraphicsContext, Image, etc.) in the Front Panel project. Like the embedded side, this class manages the synchronization with the Graphics Engine and delegates the drawing to the interface `ej.microui.display.UIDrawing`.
- The interface `ej.microui.display.UIDrawing` defines all the drawing methods available in MicroUI. The default implementation of the methods involving images calls the matching method in `ej.microui.display.UIImageDrawing`. The default implementation of the other methods reports the error that the drawing is not done.
- The interface `ej.microui.display.UIImageDrawing` defines all the methods that draw an image. The default implementation of the methods reports the error that the drawing is not done.
- The class `ej.microui.display.DisplayDrawer` implements `ej.microui.display.UIDrawing` that draws using the Graphics Engine software algorithms.
- The classes in the package `ej.drawing` implement the native of the MicroUI extended library: Drawing
- The classes in the package `ej.microui.event` manage the input events, see *Inputs Extensions*.

- The classes in the package `ej.microui.led` manage the LEDs.

Widget Display

The widget Display implements the interface `ej.microui.display.LLUIDisplayImpl` to be compatible with the implementation of the MicroUI class Display.

Features

- Simple or double buffering (default value): `doubleBufferFeature=true|false`.
- *LCD refresh rate*: simulates the time between two visible frames on the hardware device.
- *LCD flush time*: simulates the time to send the frame content to the hardware device.
- Backlight (enabled by default): `backlightFeature=true|false`.
- *Non-rectangular displays*: `filter="xxx.png"`. Some displays can have another appearance (for instance: circular).
- *Standard* pixel formats.
- *Driver-specific* pixel formats: `extensionClass="xxx"`. This class must be added in the Front Panel project and implement the interface `ej.fp.widget.Display.DisplayExtension`.

Refresh Rate

Usually a LCD is cadenced around 50-60Hz. That means the LCD can display a new frame every 16-20ms. By default this widget displays a new frame as soon as possible. It can be configured to reduce this time to simulate the hardware device.

In the widget declaration, set the attribute `refreshRate="xxx"` with a value in Hertz. A zero or negative value disables the feature.

Note: This feature is only available when double buffering mode is enabled.

The application can substitute the VEE Port's value by setting the property `-Dej.fp.widget.display.refreshRate=xxx` in the application launcher.

Flush Time

On a hardware device, the time to send the frame data from the back buffer memory to the LCD is not null. According to the hardware device technology, this time varies between 3-4 ms to 10-15ms. In SPI mode, this time may be higher, around 50ms, even more. By default this widget copies the content of back buffer as faster as possible. It can be configured to reduce this time to simulate the hardware device.

In the widget declaration, set the attribute `flushTime="xxx"` with a value in milliseconds. A zero or negative value disables the feature.

Note: This feature is only available when double buffering mode is enabled.

The application can substitute the VEE Port's value by setting the property `-Dej.fp.widget.display.flushTime=xxx` in the application launcher.

Non-rectangular Display

The Front Panel can simulate using a filter (see *Widget*). This filter defines the pixels inside and outside the whole display area. The filter image must have the same size as the rectangular display area. A display pixel at a given position will not be rendered if the pixel at the same position in the mask is fully transparent.

Note: Usually the touch panel over the display uses the same filter to reduce the touch panel area.

Example of non-rectangular display and touch:

```
<ej.fp.widget.Display x="41" y="33" width="392" height="392" filter="mask_392.png" />
<ej.fp.widget.Pointer x="41" y="33" width="392" height="392" filter="mask_392.png" touch=
↪"true"/>
```

Inputs Extensions

The input device widgets (button, joystick, touch, etc.) require a listener to know how to react on input events (press, release, move, etc.). The aim of this listener is to generate an event compatible with MicroUI Event Generator. Thereby, a button press action can become a MicroUI Buttons press event or a Command event or anything else.

A MicroUI Event Generator is known by its name. This name is fixed during the MicroUI static initialization (see *Static Initialization*). To generate an event to a specific event generator, the widget has to use the event generator name as identifier.

A Front Panel widget can:

- Force the behavior of an input action: the associated MicroUI Event Generator type is hardcoded (Buttons, Pointer, etc.), the event is hardcoded (for instance: widget button press action may be hardcoded on event generator Buttons and on the event *pressed*). Only the event generator name (identifier) should be editable by the Front Panel extension project.
- Propose a default behavior of an input action: contrary to first point, the Front Panel extension project is able to change the default behavior. For instance a joystick can simulate a MicroUI Pointer.
- Do nothing: the widget requires the Front Panel extension project to give a listener. This listener will receive all widgets action (press, release, etc.) and will have to react on it. The action should be converted on a MicroUI Event Generator event or might be dropped.

This choice of behavior is widget dependant. Please refer to the widget documentation to have more information about the chosen behavior.

Heap Simulation

Graphics Engine is using two dedicated heaps: for the images (see *Images Heap*) and the external fonts (see *External Resources*). Front Panel partly simulates the heaps usage.

- Images heap: Front Panel simulates the heap usage when the application is creating a BufferedImage, when it loads and decodes an image (PNG, BMP, etc.) which is not a raw resource and when it converts an image in MicroEJ format in another MicroEJ format. However it does not simulate the external image copy in heap (see *External Resource*).
- External fonts heap: Front Panel does not simulate this heap (see *External Resources*). There is no rendering limitation when application is using a font which is located outside CPU addresses ranges.

Image Decoders

Front Panel uses its own internal image decoders when the associated modules have been selected (see *internal image decoders*). Some additional decoders can be added like the C-side for the embedded VEE Port (see *external image decoders*). Front Panel uses the Java AWT ImageIO API to load the encoded images.

Generic Image Decoders

The Java AWT ImageIO class holds a limited list of additional decoders. To be compliant with the embedded side, these decoders are disabled by default. To add an additional decoder, specify the property `hardwareImageDecoders.list` in Front Panel configuration properties file (see *Installation*) with one or several property values:

Table 25: Front Panel Additional Image Decoders

Type	Property value
Graphics Interchange Format (GIF)	gif
Joint Photographic Experts Group (JPEG)	jpeg or jpg
Portable Network Graphics (PNG)	png
Windows bitmap (BMP)	bmp

The decoders list is comma (,) separated. Example:

```
hardwareImageDecoders.list=jpg,bmp
```

Custom Image Decoders

Additionally, the Java AWT ImageIO class offers the possibility to add some custom image decoders by using the service `javax.imageio.spi.ImageReaderSpi` .

Since UI Pack 13.2.0, Front Panel automatically includes new image decoders (new ImageIO services, see the method `LLUIDisplayImpl.decode()`), compiled in JAR files that follow this convention:

1. The JAR contains the service declaration `/META-INF/services/javax.imageio.spi.ImageReaderSpi` ,
2. The JAR filename's prefix is *imageio-*,
3. The JAR location is the VEE Port configuration project's `dropins/tools/` directory.

Note: The same JAR is used by the Front Panel and by the *Image Generator*.

Drawings

Front Panel is designed to modify the default behavior for performing *drawings*.

Image Rendering

Front Panel is designed to add the support of *custom images*.

Buffered Image

Front Panel is designed to add the support of *MicroUI BufferedImage* with a format different from the display format.

Classpath

A standard mock is running on the same JVM than the HIL Engine (see *Mock* chapter). It shares the same classpath. When the application is not using the MicroUI library (i.e., it is not an UI application, whether the VEE Port holds the MicroEJ Graphics Engine or not), the Front Panel mock runs a standard mock. When the application is using the MicroUI library, the Front Panel UI mock runs on the same JVM than the MicroEJ Simulator. In this case, the other mocks don't share the same classpath than the Front Panel mock. As a consequence, an other mock than the Front Panel mock can not send input events to MicroUI, the object created in the standard mocks's class loader are not available in the Front Panel UI's class loader (and vice versa), etc.

Since the UI Pack 13.2.0, it is possible to force to run the Front Panel UI mock in the same classpath than the HIL Engine by adding the property `-Dej.fp.hil=true` in the application JRE tab. Note that this option only works when the version of the MicroEJ Architecture used to build the VEE Port is 7.17.0 or higher.

Dependencies

- MicroUI module (see *MicroUI*),
- Display module (see *Display*): This module gives the characteristics of the graphical display that are useful for configuring the Front Panel.

Installation

Front Panel is an additional module for MicroUI library. When the MicroUI module is installed, install this module in order to be able to simulate UI drawings on the Simulator. See *Installation* to install the module.

The properties file can additional properties:

- `hardwareImageDecoders.list` [optional, default value is "" (*empty*)]: Defines the available list of additional image decoders provided by the hardware (see *Image Decoders*). Use comma (',') to specify several decoders among this list: bmp, jpg, jpeg, gif, png. If empty or unspecified, no image decoder is added.

Use

Launch a MicroUI application on the Simulator to run the Front Panel.

6.14.14 Release Notes

MicroEJ Architecture Compatibility Version

The current UI Pack version is 13.7.2. The following tables describe the compatibility ranges between MicroEJ UI Packs and MicroEJ Architectures.

Standard Versions

UI Pack Range	Architecture Range	Comment
[13.5.0-13.7.2]	[7.13.0-9.0.0[	Compatibility with Architecture 8
[13.0.0-13.4.1]	[7.13.0-8.0.0[	SNI 1.3
[12.0.0-12.1.5]	[7.11.0-8.0.0[	Move Front Panel in MicroEJ Architecture
[11.0.0-11.2.0]	[7.0.0-8.0.0[	SNI Callback feature
[9.3.1-10.0.2]	[6.13.0-7.0.0[	`LLEXT` link error with Architecture 6.13+ and UI 9+
[9.2.0-9.3.0]	[6.12.0-6.13.0[	SOAR can exclude some resources
[9.1.0-9.1.2]	[6.8.0-6.12.0[	Internal scripts
[8.0.0-9.0.2]	[6.4.0-6.12.0[	Manage external memories like byte addressable memories
[6.0.0-7.4.7]	[6.1.0-6.12.0[	

Maintenance Versions

UI Pack Version	UI Pack Base Version	Architecture Range	Comment
(maint) 8.0.0	7.4.7	[7.0.0-8.0.0[	SNI Callback feature

Foundation Libraries

The following table describes Foundation Libraries API versions implemented in MicroEJ UI Packs.

Table 26: MicroUI API Implementation

UI Pack Range	MicroUI	Drawing
[13.7.0-13.7.2]	3.4.0	1.0.4
[13.6.0-13.6.2]	3.3.0	1.0.4
[13.5.0-13.5.1]	3.2.0	1.0.4
[13.2.0-13.4.1]	3.1.1	1.0.4
13.1.0	3.1.0	1.0.3
[13.0.4-13.0.7]	3.0.3	1.0.2
13.0.3	3.0.2	1.0.1
[13.0.1-13.0.2]	3.0.1	1.0.0
13.0.0	3.0.0	1.0.0
[12.1.0-12.1.5]	2.4.0	
[11.1.0-11.2.0]	2.3.0	
[9.2.0-11.0.1]	2.2.0	
[9.1.1-9.1.2]	2.1.3	
9.1.0	2.1.2	
[9.0.0-9.0.2]	2.0.6	
[6.0.0-8.1.0]	2.0.0	

Abstraction Layer Interface

The following sections briefly describes Abstraction Layer interface changes. For more details, refer to the *Migration Guide*.

Display

UI Pack Range	Changes
[13.0.0-13.7.2]	*UI3* format: implement `LLUI_DISPLAY_impl.h`: • `void LLUI_DISPLAY_IMPL_initialize([...]);` • `void LLUI_DISPLAY_IMPL_binarySemaphoreTake([...]);` • `void LLUI_DISPLAY_IMPL_binarySemaphoreGive([...]);` • `uint8_t* LLUI_DISPLAY_IMPL_flush([...]);`
[10.0.0-12.1.5]	Remove: • `int32_t LLDISPLAY_IMPL_getWorkingBufferStartAddress([...]);` • `int32_t LLDISPLAY_IMPL_getWorkingBufferEndAddress([...]);`
[8.0.0-9.4.1]	Merge in `LLDISPLAY_impl.h`: • `LLDISPLAY_SWITCH_impl.h` • `LLDISPLAY_COPY_impl.h` • `LLDISPLAY_DIRECT_impl.h`
[6.0.0-7.4.7]	*UI2* format: implement one of header file: • `LLDISPLAY_SWITCH_impl.h` • `LLDISPLAY_COPY_impl.h` • `LLDISPLAY_DIRECT_impl.h`

Input

UI Pack Range	Changes
[13.0.0-13.7.2]	*UI3* format: implement `LLUI_INPUT_impl.h`: • `void LLUI_INPUT_IMPL_initialize([...]);` • `jint LLUI_INPUT_IMPL_getInitialStateValue([...]);` • `void LLUI_INPUT_IMPL_enterCriticalSection([...]);` • `void LLUI_INPUT_IMPL_leaveCriticalSection([...]);`
[6.0.0-12.1.5]	*UI2* format: implement `LLINPUT_impl.h` • `void LLINPUT_IMPL_initialize([...]);` • `int32_t LLINPUT_IMPL_getInitialStateValue([...]);` • `void LLINPUT_IMPL_enterCriticalSection([...]);` • `void LLINPUT_IMPL_leaveCriticalSection([...]);`

LED

UI Pack Range	Changes
[13.0.0-13.7.2]	*UI3* format: implement `LLUI_LED_impl.h`: • `jint LLUI_LED_IMPL_initialize([...]);` • `jint LLUI_LED_IMPL_getIntensity([...]);` • `void LLUI_LED_IMPL_setIntensity([...]);`
[6.0.0-12.1.5]	*UI2* format: implement `LLLEDS_impl.h` • `int32_t LLLEDS_IMPL_initialize([...]);` • `int32_t LLLEDS_IMPL_getIntensity([...]);` • `void LLLEDS_IMPL_setIntensity([...]);`

Front Panel API

The Front Panel project must fetch the widgets compatible with the MicroEJ UI Pack fetched in the VEE Port configuration project:

- Before MicroEJ UI Pack `12.0.0`, the Front Panel project must depend on the classpath variable `FRONTPANEL_WIDGETS_HOME`.
- For the UI Packs `12.x.x`, the Front Panel project must fetch the module ej.tool.frontpanel.widget-microui.
- Since MicroEJ UI Pack `13.0.0`, the Front Panel project must depend on the module com.microej.pack.ui.ui-pack(frontpanel) (the module version is the MicroEJ Generic UI Pack version, that is always aligned with the MicroEJ UI Packs specific for MCUs).

UI Pack Range	Module	Version
[13.0.0-13.7.2]	com.microej.pack.ui.ui-pack(frontpanel)	[13.0.0-13.7.2]
[12.0.0-12.1.5]	ej.tool.frontpanel.widget-microui	1.0.0
[6.0.0-11.2.0]	n/a	n/a

The widget module ej.tool.frontpanel#widget provides some widgets compatible with the Graphics Engine. This module fetches by transitivity the module com.microej.pack.ui.ui-pack(frontpanel). When the Front Panel project does not require/use the latest Front Panel UI API, it can only fetch the widget module.

Note: This module has been moved from the MicroEJ Central Repository to the MicroEJ Developer Repository.

Widget Module Range	UI Pack Compatibility Range	Repository
3.0.0	[13.5.1-10-13.7.2]	Developer
2.2.0	[13.1.0-13.7.2]	Developer
[2.1.0-2.1.1]	[13.1.0-13.7.2]	Central
2.0.0	[13.0.0-13.7.2]	Central
1.0.1	[12.0.0-12.1.5]	Developer

To use the latest functionalities provided by the UI Pack `13.0.0` and higher, the Front Panel project must depend on the same version of the UI Pack as the VEE Port configuration project. However, if the Front Panel project does not require/use the latest Front Panel UI API, it can fetch a version of the UI Pack older than the version fetched in the VEE Port configuration project.

Image Generator API

Since MicroEJ UI Pack `13.0.0`, the Image Generator extension project must depend on module com.microej.pack.ui.ui-pack(imagegenerator). The module version is the MicroEJ Generic UI Pack version, that is always aligned with the MicroEJ UI Packs specific for MCUs.

UI Pack Range	Module	Version
[13.0.0-13.7.2]	com.microej.pack.ui.ui-pack(imagegenerator)	[13.0.0-13.7.2]

Note: Before MicroEJ UI Pack `13.0.0`, the Image Generator extension project must depend on classpath variable `IMAGE-GENERATOR-x.x`.

C Modules

MicroUI C Module

The MicroUI C module com.microej.clibrary.llimpl(microui) is available on MicroEJ Central Repository, see *C Modules*. The following table describes the compatibility versions between the MicroEJ UI Packs and the C modules:

UI Pack Range	C Module Range	Comment
[13.7.0-13.7.2]	3.1.0	free image resources
[13.5.0-13.6.2]	3.0.0	multiple Graphics Context output formats
[13.3.0-13.4.1]	[2.0.0-2.0.1]	copy and draw image
[13.1.0-13.2.0]	[1.1.0-1.1.1]	image heap, events queue, drawing limits
[13.0.0-13.1.0]	[1.0.0-1.0.3]	

Extended C Modules

Some C modules extend the main MicroUI C module. They override the default implementation to use a GPU to perform some drawings. Contrary to the main MicroUI C module, they are optional: when they are not available, the default implementation of drawings is used. The default implementations use the Graphics Engine software algorithms.

STM32 Chrom-ART

The *DMA2D C module* targets the STM32 CPU that provides the Chrom-ART accelerator.

The following table describes the version compatibility between the MicroEJ UI Packs and the C modules:

UI Pack Range	C Module Range	Comment
[13.7.0-13.7.2]	4.1.0	free image resources
[13.5.0-13.6.2]	4.0.0	multiple Graphics Context output formats
[13.3.0-13.4.1]	[3.0.0-3.0.2]	copy and draw image
[13.1.0-13.2.0]	[2.0.0-2.1.0]	drawing limits
[13.0.0-13.0.7]	[1.0.6-1.0.8]	

Vivante VG-Lite

The *VG-Lite C module* targets the NXP CPU that provides the Vivante VG-Lite accelerator.

The following table describes the version compatibility between the MicroEJ UI Packs and the C modules:

UI Pack Range	C module Range	Comment
[13.7.0-13.7.2]	7.2.0	free image resources
[13.5.0-13.6.2]	[6.0.0-7.1.0]	multiple Graphics Context output formats
[13.3.0-13.4.1]	[3.0.0-5.0.1]	copy and draw image
[13.1.0-13.2.0]	[1.0.0-2.0.0]	

The following table describes the version compatibility between the C module and the VG-Lite libraries (officially supported):

C Module Range	VG-Lite Libraries Range
[7.1.0-7.2.0]	3.0.15_rev4 and 3.0.15_rev7
[4.0.0-7.0.0]	3.0.15_rev4
[2.0.0-3.0.0]	3.0.11_rev3
1.0.0	3.0.4_rev2 and 3.0.4_rev4

Think Silicon NemaGFX

The *NemaGFX C module* targets the CPU that provides the NemaGFX accelerator.

The following table describes the version compatibility between the MicroEJ UI Packs and the C modules:

UI Pack Range	C module Range	Comment
[13.7.0-13.7.2]	[1.1.0-1.2.0]	free image resources
[13.5.0-13.6.2]	1.0.0	

6.14.15 Changelog

13.7.2 (2023-12-21)

MicroUI

Fixed

- Fix the drawing of thick faded circle arcs.

Simulator

Fixed

- Fix the management of KF feature's fonts.

13.7.0 (2023-10-23)

MicroUI

Added

- Add the pre-multiplied image formats *ARGB8888_PRE*, *ARGB1555_PRE* and *ARGB4444_PRE*.
- Add the possibility to free third-party resources associated with images.
- Add some traces when debugging the SNI resources.

Image Generator

Changed

- Do not enable the cache when generating external resources.

Fixed

- Do not use cached images when there is no *.images.list* file.
- Do not use cached images when a VEE Port property has changed.
- Fix the handling of backslashes in list files.
- Remove debug log in script.

Font Generator

Changed

- Do not enable the cache when generating external resources.

Fixed

- Do not use cached fonts when a VEE Port property has changed.
- Fix the handling of backslashes in list files.

[13.6.2] (2023-09-20)

Image Generator

Fixed

- Fix handling zip/jar file entries in the cache.

Font Generator

Fixed

- Fix handling zip/jar file entries in the cache.

[13.6.1] (2023-07-26)

MicroUI

Fixed

- Fix creating a BufferedImage when traces are enabled.

[13.6.0] (2023-07-17)

MicroUI

Added

- Add a flag stating that an undefined character was drawn.

Fixed

- Fix the Java compiler version used to build the MicroUI extension class to be compatible with the JDK 11.
- Fix the drawing of faded arcs and ellipses.

Image Generator

Changed

- Use a cache to avoid generating images for each launch.

Font Generator

Changed

- Use a cache to avoid generating fonts for each launch.

[13.5.1] (2023-06-08)

MicroUI

Fixed

- Fix the compatibility with MicroEJ Architecture 8 (SOAR error with internal MicroUI system properties file).

FrontPanel

Fixed

- Fix consecutive calls to `LLUIDisplay.newMicroUIImage()` throwing an exception.
- Allow overriding the display drawer with a service or in a front panel widget.

[13.5.0] (2023-05-03)

- Compatible with Architecture 7.13.0 or higher.

MicroUI Implementation

Changed

- Implement MicroUI 3.2 (multi-mutable image formats, drawing log flags, etc.).

Fixed

- Fix ellipse fading.

Drawing Implementation

Fixed

- Fix the position of arc caps.

LLAPIs

Added

- Add some functions in *LLUI_DISPLAY.h* to manage the MicroUI Drawing Log flags.
- Add some functions in *LLUI_DISPLAY.h* to change the MicroUI clip and colors.
- Add the notion of “drawer” to identify the available drawer for a given MicroUI Image format.

Changed

- Change the MicroUI image format *MICROUI_IMAGE_FORMAT_LCD* by *MICROUI_IMAGE_FORMAT_DISPLAY*.
- Change the signature of *xx_drawing_soft.h*: all functions return a drawing status.

Removed

- Remove *ui_drawing.h* and *dw_drawing.h* (move them in MicroUI C Module).

FrontPanel

Added

- Add a service to decode immutable images with a custom format.
- Add a service to create mutable images with a custom format.
- Add a service to draw into mutable images with a format different than the display format.
- Add some methods to manage the MicroUI Drawing Log flags.
- Add some methods to change the MicroUI clip and colors.

Changed

- Merge *DWDrawing* in *UIDrawing*.
- Turn *UIDrawing* as a service to handle drawings for a specific format.
- Change the mechanism to get the software drawer.
- Change the MicroUI image format *MICROUI_IMAGE_FORMAT_LCD* by *MICROUI_IMAGE_FORMAT_DISPLAY*.

Removed

- Remove the interfaces *UIDrawingDefault* and *DWDrawingDefault* (implement the interface *UIDrawing* instead).

ImageGenerator

Added

- Add compatibility with Architecture 8.

[13.4.1] (2023-02-06)

Drawing Implementation

Fixed

- Fix thick lines drawing (when thickness is larger than length).
- Fix circle and ellipse drawing (when the diameter/axis has an even length).

FrontPanel

Changed

- Increase the speed of RAW image decoding step.

ImageGenerator

Fixed

- Fix the VEE Port's memory alignment constraint.

[13.4.0] - 2022-12-13

- Compatible with Architecture 7.13.0 or higher.

MicroUI Implementation

Fixed

- Fix the unexpected resuming of the pump Java thread when a new event is added to the queue if it is an other component than the MicroUI queue that has suspended the pump Java thread.
- Fix the flush bounds of drawCircleArc and drawEllipseArc.

FrontPanel

Added

- Add some checks to not perform a drawing when it is unnecessary.

Fixed

- Fix the Front Panel representation of a BufferedImage: it is always opaque.

ImageGenerator

Added

- Add the image format A8_RLE.

Changed

- Rename RLE1 format in ARGB1565_RLE (keep RLE1 for backward compatibility).

Fixed

- Fix the non-generation of external images for the features.

FontGenerator

Fixed

- Fix the external fonts output folder for the features.

[13.3.1] - 2022-09-09

- Compatible with Architecture 7.13.0 or higher.

Image Generator

Added

- Add an *Application Option* to quickly test an Image Generator Extension project.

Changed

- Increase logs when application verbosity is enabled.
- Check the stride defined by the Image Generator Extension project (throw an error if the value is incompatible with the memory alignment).

Fixed

- Fix the external resource generation: they were no longer generated (UI pack 13.3.0 regression).
- Fix the duplicate generation (as internal and external resources) of the custom `.list` file images (consider only custom `.list` file images as external resources when the prefix of the list file extension starts with `extern`).
- Fix the internal limit error when converting images with BPP lower than 8 bits (for platforms that define a rule for the image stride through an Image Generator Extension project).

[13.3.0] - 2022-09-02

- Compatible with Architecture 7.13.0 or higher.

MicroUI Implementation

Fixed

- Fix the Cx (x == 1 | 2 | 4) Graphics Engine's when memory layout is "column".
- Fix the consistency between Image.getImage() and Font.getFont() about starting MicroUI.

Front Panel

Added

- Add custom image formats and a service to prepare for future MicroUI functionality.

Image Generator

Fixed

- Fix the stride stored in the image when the Graphics Engine's memory layout is "column".

LLAPIs

Added

- Add custom image formats to prepare for future MicroUI functionality.
- Add LLAPI to adjust new image characteristics (size and alignment).
- Add API: `UI_DRAWING_copyImage` and `UI_DRAWING_drawRegion`.
- Add the LLUI version (== UI Pack version) in header files.

Changed

- Use type `jbyte` to identify an image format instead of `MICROUI_ImageFormat` (prevent C compiler optimization).

Removed

- Remove the MicroUI's native functions declaration with macros *(not backward compatible)*.

[13.2.0] - 2022-05-05

- Compatible with Architecture 7.16.0 or higher.

Integration

Changed

- Update to the latest SDK license notice.

MicroUI Implementation

Changed

- Use `.rodata` sections instead of `.text` sections.

Fixed

- Clean KF stale references when killing a feature without display context switch.
- Make sure to wait the end of an asynchronous drawing before killing a KF feature.
- Redirect the events sent to the pump to the pump's handler instead of to the event generator's handler.
- Fix the drawing of antialiased arc: caps are drawn over the arc itself (rendering issue when the GraphicsContext's background color is set).
- Fix the drawing of antialiased arc: arc is not fully drawn when (int)startAngle == (int)((startAngle + arcAngle) % 360)).
- Fix the input queue size when not already set by the application launcher.

- Fix the use of a negative `scanLength` in GraphicsContext.readPixels() and Image.readPixels().

Front Panel

Added

- Add the property `-Dej.fp.hil=true` in the application launcher to force to run the FrontPanel with the Graphics Engine as a standard HIL mock (requires MicroEJ Architecture 7.17.0 or higher).
- Add `LLUIDisplayImpl.decode()` : the Front Panel project is able to read encoded image like the embedded side.
- Include automatically the AWT ImageIO services.
- Add `MicroUIImage.readPixel()` to read an image's pixel color.

Fixed

- Fix the "display context switch" and the loading of feature's font.
- Fix OOM (Java heap space) when opening/closing several hundreds of big RAW Images.
- Fix the synchronization with the Graphics Engine when calling GraphicsContext.setColor() or GraphicsContext.enableEllipsis().

Image Generator

Added

- Include automatically the AWT ImageIO services.
- Allow to a custom image converter to generate a file other than a binary resource.
- Allow to a custom image converter to specify the supported `.list` files.

LLAPIs

Added

- Add `LLUI_DISPLAY_readPixel` to read an image's pixel color.

BSP

Fixed

- Fix the IAR Embedded Workbench warnings during debug session.

[13.1.0] - 2021-08-03

- Compatible with Architecture 7.16.0 or higher.

MicroUI API

Removed

- Remove MicroUI and Drawing API from UI pack.

MicroUI Implementation

Changed

- Compatible with MicroUI API 3.1.0.
- Check Immortals heap minimal size required by MicroUI implementation.
- Change the EventGenerator Pointer event format.
- Do no systematically use the GPU to draw intermediate steps of a shape.

Fixed

- EventGenerator's event has not to be sent to the Display's handler when EventGenerator's handler is null.
- Fill rounded rectangle: fix rendering when corner radius is higher than rectangle height.
- An external image is closed twice when the application only checks if the image is available.
- RLE1 image rendering when platform requires image pixels address alignment.
- Manage the system fonts when the font generator is not embedded in the platform.
- Have to wait the end of current drawing before closing an image.

Drawing Implementation

Changed

- Compatible with Drawing API 1.0.3.

LLAPIs

Added

- Add `LLUI_DISPLAY_convertDisplayColorToARGBColor()` .
- Add LLAPI to manage the *MicroUI Image heap*.
- Add LLAPI to dump the *MicroUI Events queue*.

Changed

- Change signature of `LLUI_DISPLAY_setDrawingLimits()` : remove `MICROUI_GraphicsContext*` to be able to call this function from GPU callback method.

Simulator

Added

- Add `MicroUIImage.getImage(int)` : apply a rendering color on Ax images.
- Add `LLUIDisplay.convertRegion()` : convert a region according image format restrictions.

Changed

- Compatible with new EventGenerator Pointer event format.

Fixed

- Fix OutputFormat A8 when loading an image (path or stream) or converting a RAW image.
- Fix OOM (Java heap space) when opening/closing several hundreds of MicroUI Images.
- Simulates the image data alignment.

[13.0.7] - 2021-07-30

- Compatible with Architecture 7.16.0 or higher.

MicroUI Implementation

Fixed

- Allow to open a font in format made with UI Pack 12.x (but cannot manage `Dynamic` styles).
- Display.flush() method is called once when MicroUI pump thread has a higher priority than the caller of Display.requestFlush().
- Display.requestFlush() is only executed once from a feature (UI deadlock).

Misc

Fixed

- Fix MMM dependencies: do not fetch the MicroEJ Architecture.

[13.0.6] - 2021-03-29

- Compatible with Architecture 7.16.0 or higher.

LLAPIs

Fixed

- Size of the typedef `MICROUI_Image` : do not depend on the size of the enumeration `MICROUI_ImageFormat` (`LLUI_PAINTER_impl.h`).

[13.0.5] - 2021-03-08

- Compatible with Architecture 7.16.0 or higher.

MicroUI Implementation

Removed

- Remove ResourceManager dependency.

Fixed

- A feature was not able to call Display.callOnFlushCompleted().
- Stop feature: prevent NullPointerException when a kernel's EventGenerator is removed from event generators pool.
- Filter DeadFeatureException in MicroUI pump.
- Drawing of thick arcs which represent an almost full circle.
- Drawing of thick faded arcs which pass by 0° angle.

Simulator

Fixed

- Front panel memory management: reduce simulation time.

[13.0.4] - 2021-01-15

- Compatible with Architecture 7.16.0 or higher.

MicroUI API

Changed

- [Changed] Include MicroUI API 3.0.3.
- [Changed] Include MicroUI Drawing API 1.0.2.

MicroUI Implementation

Fixed

- Fix each circle arc cap being drawn on both sides of an angle.
- Fix drawing of rounded caps of circle arcs when fade is 0.
- Cap thickness and fade in thick drawing algorithms.
- Clip is not checked when filling arcs, circles and ellipsis.
- Image path when loading an external image (`LLEXT`).
- `InternalLimitsError` when calling MicroUI.callSerially() from a feature.

Drawing Implementation

Fixed

- Draw deformed image is not rendered.

ImageGenerator

Changed

- Compatible with com.microej.pack.ui#ui-pack(imageGenerator)#13.0.4.

Fixed

- NullPointerException when trying to convert an unknown image.
- Restore external resources option in MicroEJ launcher.

[13.0.3] - 2020-12-03

- Compatible with Architecture 7.16.0 or higher.

MicroUI API

Changed

- [Changed] Include MicroUI API 3.0.2.
- [Changed] Include MicroUI Drawing API 1.0.1.

MicroUI Implementation

Fixed

- Reduce Java heap usage.
- Fix empty images heap.
- Draw image algorithm does not respect image stride in certain circumstances.
- Fix flush limits of drawThickFadedLine, drawThickEllipse and drawThickFadedEllipse.

[13.0.2] - 2020-10-02

- Compatible with Architecture 7.16.0 or higher.
- Use new naming convention: `com.microej.architecture.[toolchain].[architecture]-ui-pack`.

Fixed

- [ESP32] - Potential `PSRAM` access faults by rebuilding using esp-idf v3.3.0 toolchain - `simikou2`.

[13.0.1] - 2020-09-22

- Compatible with Architecture 7.16.0 or higher.

MicroUI API

Changed

- Include MicroUI API 3.0.1.

MicroUI Implementation

Fixed

- Throw an exception when there is no display.
- Antialiased circle may be cropped.
- FillRoundedRectangle can give invalid arguments to FillRectangle.
- Flush bounds may be invalid.
- Reduce memory footprint (java heap and immortal heap).
- No font is loaded when an external font is not available.
- A8 color is cropped to display limitation too earlier on simulator.

LLAPIs

Fixed

- Missing a LLAPI to check the overlapping between source and destination areas.

Simulator

Fixed

- Cannot use an external image decoder on front panel.
- Missing an API to check the overlapping between source and destination areas.

ImageGenerator

Fixed

- Cannot build a platform with image generator and without front panel.

[13.0.0] - 2020-07-30

- Compatible with Architecture 7.16.0 or higher.
- Integrate SDK 3.0-B license.

MicroUI API

Changed

- [Changed] Include MicroUI API 3.0.0.
- [Changed] Include MicroUI Drawing API 1.0.0.

MicroUI Implementation

Added

- Manage image data (pixels) address alignment (not more fixed to 32-bits word alignment).

Changed

- Reduce EDC dependency.
- Merge `DisplayPump` and `InputPump` : only one thread is required by MicroUI.
- Use a `bss` section to load characters from an external font instead of using java heap.

Removed

- Dynamic fonts (dynamic bold, italic, underline and ratios).

Fixed

- Lock only current thread when waiting end of flush or end of drawing (and not all threads).
- Draw anti-aliased ellipse issue (vertical line is sometimes drawn).
- Screenshot on platform whose *physical* size is higher than *virtual* size.

Known issue

- Render of draw/fill arc/circle/ellipse with an even diameter/edge is one pixel too high (center is 1/2 pixel too high).

LLAPIs

Added

- Some new functions are mandatory: see header files list, tag *mandatory*.
- Some new functions are optional: see header files list, tag *optional*.
- Some header files list the libraries `ej.api.microui` and `ej.api.drawing` natives. Provided by Abstraction Layer implementation module com.microej.clibrary.llimpl#microui.
- Some header files list the drawing algorithms the platform can implement; all algorithms are optional.
- Some header files list the internal Graphics Engine software algorithms the platform can call.

Changed

- All old header files and functions have been renamed or shared.
- See *Migration notes* that describe the available changes in LLAPI.

Simulator

Added

- Able to override MicroUI drawings algorithms like embedded platform.

Changed

- Compatible with com.microej.pack.ui#ui-pack(frontpanel)#13.0.0.
- See *Migration notes* that describe the available changes in Front Panel API.

Removed

- `ej.tool.frontpanel#widget-microui` has been replaced by `com.microej.pack.ui#ui-pack(frontpanel)`.

ImageGenerator

Added

- Redirects source image reading to the image generator extension project in order to increase the number of supported image formats in input.
- Redirects destination image generation to the image generator extension project in order to be able to encode an image in a custom RAW format.
- Generates a linker file in order to always link the resources in same order between two launches.

Changed

- Compatible with com.microej.pack.ui#ui-pack(imageGenerator)#13.0.0.
- See *Migration notes* that describe the available changes in Image Generator API.
- Uses a service loader to loads the image generator extension classes.
- Manages image data (pixels) address alignment.

Removed

- Classpath variable `IMAGE-GENERATOR-x.x` : Image generator extension project has to use ivy dependency `com.microej.pack.ui#ui-pack(imageGenerator)` instead.

FontGenerator

Changed

- Used a dedicated `bss` section to load characters from an external font instead of using the java heap.

[12.1.5] - 2020-10-02

- Compatible with Architecture 7.11.0 or higher.
- Use new naming convention: `com.microej.architecture.[toolchain].[architecture]-ui-pack`.

Fixed

- [ESP32] - Potential `PSRAM` access faults by rebuilding using esp-idf v3.3.0 toolchain - `simikou2`.

[12.1.4] - 2020-03-10

- Compatible with Architecture 7.11.0 or higher.

MicroUI Implementation

Fixed

- Obsolete references on Java heap are used (since MicroEJ UI Pack 12.0.0).

[12.1.3] - 2020-02-24

- Compatible with Architecture 7.11.0 or higher.

MicroUI Implementation

Fixed

- Caps are not used when drawing an anti-aliased line.

[12.1.2] - 2019-12-09

- Compatible with Architecture 7.11.0 or higher.

MicroUI Implementation

Fixed

- Fix Graphics Engine empty clip (empty clip had got a size of 1 pixel).
- Clip not respected when clip is set “just after or before” graphics context drawable area: first (or last) line (or column) of graphics context was rendered.

[12.1.1] - 2019-10-29

- Compatible with Architecture 7.11.0 or higher.

MicroUI Implementation

Fixed

- Fix Graphics Engine clip (cannot be outside graphics context).

[(maint) 8.0.0] - 2019-10-18

- Compatible with Architecture 7.0.0 or higher.
- Based on 7.4.7.

MicroUI Implementation

Fixed

- Pending flush cannot be added after an `OutOfEventException`.

[12.1.0] - 2019-10-16

- Compatible with Architecture 7.11.0 or higher.

MicroUI API

Changed

- Include MicroUI API 2.4.0.

MicroUI Implementation

Changed

- Prepare inlining of get X/Y/W/H methods.
- Reduce number of strings embedded by MicroUI library.

Fixed

- Pending flush cannot be added after an `OutOfEventException`.
- Display.isColor() returns an invalid value.
- Draw/fill circle/ellipse arc is not drawn when angle is negative.

[12.0.2] - 2019-09-23

- Compatible with Architecture 7.11.0 or higher.

MicroUI Implementation

Changed

- Change `CM4hardfp_IAR83` compiler flags.
- Remove RAW images from cache as soon as possible to reduce java heap usage.
- Do not cache RAW images with their paths to reduce java heap usage.

Fixed

- Remove useless exception in SystemInputPump.

[12.0.1] - 2019-07-25

- Compatible with Architecture 7.11.0 or higher.

MicroUI Implementation

Fixed

- Physical size is not taken in consideration.

Simulator

Fixed

- Increase native implementation execution time.

[12.0.0] - 2019-06-24

- Compatible with Architecture 7.11.0 or higher.

MicroUI Implementation

Added

- Trace MicroUI events and log them on SystemView.

Changed

- Manage the Graphics Context clip on native side.
- Use java heap to store images metadata instead of using icetea heap (remove option “max offscreen”).
- Optimize retrieval of all fonts.
- Ensure user buffer size is larger than LCD size.
- Use java heap to store flying images metadata instead of using icetea heap (remove option “max flying images”).

- Use java heap to store fill polygon algorithm's objects instead of using icetea heap (remove option "max edges").
- `SecurityManager` enabled as a boolean constant option (footprint removal by default).
- Remove `FlyingImage` feature using BON constants (option to enable it).

Fixed

- Wrong rendering of a fill polygon on emb.
- Wrong rendering of image overlaping on C1/2/4 platforms.
- Wrong rendering of a LUT image with more than 127 colors on emb.
- Wrong rendering of an antialiased arc with 360 angle.
- Debug option com.is2t.microui.log=true fails when there is a flying image.
- Gray scale between gray and white makes magenta.
- Minimal size of some buffers set by user is never checked.
- The format of a RAW image using "display" format is wrong.
- Dynamic image width for platform C1/2/4 may be wrong.
- Wrong pixel address when reading from a C2/4 display.
- getDisplayColor() can return a color with transparency (spec is `0x00RRGGBB`).
- A fully opaque image is tagged as transparent (ARGB8888 platform).

Simulator

Added

- Simulate flush time (add JRE property `-Dfrontpanel.flush.time=8`).

Fixed

- A pixel read on an image is always truncated.

FrontPanel Plugin

Removed

- FrontPanel version 5: Move front panel from MicroEJ UI Pack to Architecture *(not backward compatible)*; Architecture contains now Front Panel version 6.

[11.2.0] - 2019-02-01

- Compatible with Architecture 7.0.0 or higher.

MicroUI Implementation

Added

- Manage extended UTF16 characters (> 0xffff).

Fixed

- IOException thrown instead of an OutOfMemory when using external resource loader.

Tools

Removed

- Remove Font Designer from pack (useless).

[11.1.2] - 2018-08-10

- Compatible with Architecture 7.0.0 or higher.

MicroUI Implementation

Fixed

- Fix drawing bug in thick circle arcs.

[11.1.1] - 2018-08-02

- Compatible with Architecture 7.0.0 or higher.
- Internal release.

[11.1.0] - 2018-07-27

- Compatible with Architecture 7.0.0 or higher.
- Merge 10.0.2 and 11.0.1.

MicroUI API

Changed

- Include MicroUI API 2.3.0.

MicroUI Implementation

Added

- `LLDisplay` : prepare round LCD.

Fixed

- `Fillrect` throws a hardfault on 8bpp platform.
- Rendering of a LUT image is wrong when using software algorithm.

[11.0.1] - 2018-06-05

- Compatible with Architecture 7.0.0 or higher.
- Based on 11.0.0.

MicroUI Implementation

Fixed

- Image rendering may be invalid on custom display.
- Render a dynamic image on custom display is too slow.
- LRGB888 image format is always fully opaque.
- Number of colors returned when it is a custom display may be wrong.

[10.0.2] - 2018-02-15

- Compatible with Architecture 6.13.0 or higher.
- Based on 10.0.1.

MicroUI Implementation

Fixed

- Number of colors returned when it is a custom display may be wrong.
- LRGB888 image format is always fully opaque.
- Render a dynamic image on custom display is too slow.
- Image rendering may be invalid on custom display.

[11.0.0] - 2018-02-02

- Compatible with Architecture 7.0.0 or higher.
- Based on 10.0.1.

MicroUI Implementation

Changed

- SNI Callback feature in the VM to remove the SNI retry pattern *(not backward compatible)*.

[10.0.1] - 2018-01-03

- Compatible with Architecture 6.13.0 or higher.

MicroUI Implementation

Fixed

- Hard fault when using custom display stack.

[10.0.0] - 2017-12-22

- Compatible with Architecture 6.13.0 or higher.

MicroUI Implementation

Changed

- Improve `TOP-LEFT` anchor checks.

Fixed

- Subsequent renderings may not be correctly flushed.
- Rendering of display on display was not optimized.

Simulator

Changed

- Check the allocated memory when creating a dynamic image *(not backward compatible)*.

Misc

Added

- Option in platform builder to images heap size.

[9.4.1] - 2017-11-24

- Compatible with Architecture 6.12.0 or higher.

ImageGenerator

Fixed

- Missing some files in image generator module.

[9.4.0] - 2017-11-23

- Compatible with Architecture 6.12.0 or higher.
- Deprecated: use 9.4.1 instead.

MicroUI Implementation

Added

- LUT image management.

Changed

- Optimize character encoding removing first vertical line when possible.

Fixed

- Memory leak when an `OutOfEventException` is thrown.
- A null Java object is not checked when using a font.

[9.3.1] - 2017-09-28

- Compatible with Architecture 6.12.0 or higher.

MicroUI Implementation

Fixed

- Returned X coordinates when drawing a string was considered as an error code.
- Exception when loading a font from an application.
- `LLEXT` link error with Architecture 6.13+ and UI 9+.

[9.3.0] - 2017-08-24

- Compatible with Architecture 6.12.0 or higher.

MicroUI Implementation

Fixed

- Ellipsis must not drawn when text anchor is a “manual” `TOP-RIGHT`.

Simulator

Fixed

- Do not create an AWT window for each image.
- Error when trying to play with an unknown led.

[9.2.1] - 2017-08-14

- Compatible with Architecture 6.12.0 or higher.

Simulator

Added

- Provide function to send a Long Button event.
- “flush” debug option.

Fixed

- Mock startup is too long.

[9.2.0] - 2017-07-21

- Compatible with Architecture 6.12.0 or higher.
- Merge 9.1.2 and 9.0.2.

MicroUI API

Changed

- Include MicroUI API 2.2.0.

MicroUI Implementation

Added

- Provide function to send a Long Button event (emb only).

Changed

- Use font format v5.
- A signature on RAW files.
- Allow to open a raw image with `Image.createImage(stream)`.
- Improve `Image.createImage(stream)` when stream is a memory input stream.

Fixed

- Draw region of the display on the display does not support overlap.
- Unspecified exception while loading an image with an empty name.
- Display.flush(): ymax can be higher than display.height.

ImageGenerator

Fixed

- Generic displays must be able to generate standard images.

Misc

Changed

- SOAR can exclude some resources (update llext output folder).

Fixed

- RI build: reduce frontpanel dependency.

[9.0.2] - 2017-04-21

- Compatible with Architecture 6.4.0 or higher.
- Based on 9.0.1.

MicroUI Implementation

Fixed

- Rendering of a RAW image on grayscale display is wrong.

ImageGenerator

Fixed

- An Ax image may be fully opaque.

[9.1.2] - 2017-03-16

- Compatible with Architecture 6.8.0 or higher.
- Based on 9.1.1.

MicroUI API

Changed

- Include MicroUI API 2.1.3.

MicroUI Implementation

Added

- Renderable strings.

Changed

- Draw string: improve time to perform it.
- Optimize antialiased circle arc drawing when fade=0.

Fixed

- ImageScale bugs.
- Draw string: some errors are not thrown.
- `Font.getWidth()` and getHeight() don't use ratio factor.
- Draw antialiased circle arc render issue.
- Draw antialiased circle arc render bug with 45° angles.
- MicroUI lib expects the dynamic image decoder default format.
- Wrong error code is returned when converting an image.

ImageGenerator

Fixed

- Use the application classpath.
- An Ax image may be fully opaque.

[9.0.1] - 2017-03-13

- Compatible with Architecture 6.4.0 or higher.
- Based on 9.0.0.

MicroUI Implementation

Fixed

- Hardfault when filling a rectangle on an odd image.
- Pixel rendering on non-standard LCD is wrong.
- RZ hardware accelerator: RAW images have to respect an aligned size.
- Use the classpath when invoking the fonts and images generators.

Simulator

Fixed

- Wrong rendering of A8 images.

FrontPanel Plugin

Fixed

- Manage display mask on preview.
- Respect initial background color set by user on preview.
- Preview does not respect the real size of display.

[9.1.1] - 2017-02-14

- Compatible with Architecture 6.8.0 or higher.
- Based on 9.1.0.

Misc

Fixed

- RI build: Several custom event generators in same `microui.xml` file are not embedded.

[9.1.0] - 2017-02-13

- Compatible with Architecture 6.8.0 or higher.
- Based on 9.0.0.

MicroUI API

Changed

- Include MicroUI API 2.1.2.

MicroUI Implementation

Added

- G2D hardware accelerator.
- Hardware accelerator: add flip feature.

Fixed

- Hardfault when filling a rectangle on an odd image.
- Pixel rendering on non-standard LCD is wrong.
- RZ hardware accelerator: RAW images have to respect an aligned size.
- Use the classpath when invoking the fonts and images generators.
- Exception when flipping an image out of display bounds.
- Flipped image is translated when clip is modified.

Simulator

Fixed

- Wrong rendering of A8 images.

FrontPanel Plugin

Fixed

- Manage display mask on preview.
- Respect initial background color set by user on preview.
- Preview does not respect the real size of display.

[9.0.0] - 2017-02-02

- Compatible with Architecture 6.4.0 or higher.

MicroUI API

Changed

- Include MicroUI API 2.0.6.

MicroUI Implementation

Changed

- Update MicroUI to use watchdogs in KF implementation.

Fixed

- Display linker file is required even if there is no display on platform.
- MicroUI on KF: NPE when changing app quickly (in several threads).
- MicroUI on KF: NPE when stopping a Feature and there's no eventHandler in a generator.
- MicroUI on KF: Remaining K->F link when there is no default event handler registered by the Kernel.

MWT

Removed

- Remove MWT from MicroEJ UI Pack *(not backward compatible)*.

Simulator

Added

- Optional mask on display.

Changed

- Display Device UID if available in the window title.

Tools

Changed

- FrontPanel plugin: Update icons.
- FontDesigner plugin: Update icons.
- Font Designer and Generator: use Unicode 9.0.0 specification.

Misc

Fixed

- Remove obsolete documentations from FrontPanel And FontDesigner plugins.

[8.1.0] - 2016-12-24

- Compatible with Architecture 6.4.0 or higher.

MicroUI Implementation

Changed

- Improve image drawing timings.
- Runtime decoders can force the output RAW image's fully opacity.

MWT

Fixed

- With two panels, the paint is done but the screen is not refreshed.
- Widget show notify method is called before the panel is set.
- Widget still linked to panel when `lostFocus()` is called.

Simulator

Added

- Can add an additional screen on simulator.

[8.0.0] - 2016-11-17

- Compatible with Architecture 6.4.0 or higher.

MicroUI Implementation

Added

- RZ UI acceleration.
- External image decoders.
- Manage external memories like internal memories.
- Custom display stacks (hardware acceleration).

Changed

- Merge stacks `DIRECT/COPY/SWITCH` *(not backward compatible)*.

Fixed

- add KF rule: a thread cannot enter in a feature code while it owns a kernel monitor.
- automatic flush is not waiting the end of previous flush.
- Invalid image rotation rendering.
- Do not embed Images & Fonts.list of kernel API classpath in app mode.
- Invalid icetea heap allocation.
- microui image: invalid “defaultformat” and “format” fields values.

MWT

Fixed

- possible to create an inconsistent hierarchy.

Simulator

Added

- Can decode additional image formats.

Fixed

- Cannot set initial value of StateEventGenerator.

[7.4.7] - 2016-06-14

- Compatible with Architecture 6.1.0 or higher.

MicroUI Implementation

Fixed

- Do not create all fonts derivations of built-in styles.
- A bold font is not flagged as bold font.
- Wrong A4 image rendering.

Simulator

Fixed

- Cannot convert an image.

[7.4.2] - 2016-05-25

- Compatible with Architecture 6.1.0 or higher.

MicroUI Implementation

Fixed

- invalid image drawing for *column* display.

[7.4.1] - 2016-05-10

- Compatible with Architecture 6.1.0 or higher.

MicroUI Implementation

Fixed

- Restore stack 1, 2 and 4 BPP.

[7.4.0] - 2016-04-29

- Compatible with Architecture 6.1.0 or higher.

MicroUI Implementation

Fixed

- image A1's width is sometimes invalid.

Simulator

Added

- Restore stack 1, 2 and 4 BPP.

[7.3.0] - 2016-04-25

- Compatible with Architecture 6.1.0 or higher.

MicroUI Implementation

Added

- Stack 8BPP with LUT support.

[7.2.1] - 2016-04-18

- Compatible with Architecture 6.1.0 or higher.

Misc

Fixed

- Remove `java` keyword in workbench extension.

[7.2.0] - 2016-04-05

- Compatible with Architecture 6.1.0 or higher.

Tools

Added

- Preprocess `*.xxx.list` files.

[7.1.0] - 2016-03-02

- Compatible with Architecture 6.1.0 or higher.

MicroUI Implementation

Added

- Manage several images RAW formats.

[7.0.0] - 2016-01-20

- Compatible with Architecture 6.1.0 or higher.

Misc

Changed

- Remove `@jpf.property.header@` prefix to Application options *(not backward compatible)*.

[6.0.1] - 2015-12-17

MicroUI Implementation

Fixed

- A negative clip throws an exception on simulator.

[6.0.0] - 2015-11-12

MicroUI Implementation

Changed

- LLDisplay for UIv2 *(not backward compatible)*.

6.14.16 Migration Guide

From 13.6.x to 13.7.x

Front Panel

- Set the explicit dependency to the UI Pack 13.7.2:

```
<dependency org="com.microej.pack.ui" name="ui-pack" rev="13.7.2">
      <artifact name="frontpanel" type="jar"/>
</dependency>
```

BSP

- *[VEE Port configuration project]*
  - Fetch C Module MicroUI 3.1.0.
- *[BSP project]*
  - Optionally, implement `UI_DRAWING_freeImageResources(MICROUI_Image* image)` (single-output buffered image format) or `UI_DRAWING_freeImageResources_X(MICROUI_Image* image)` (mulitple-output buffered image formats, where *X* is the image format identifier) to free the resources associated with a buffered image when it is closed.

BSP with DMA2D

- *[VEE Port configuration project]*
  - Fetch C Module Display-DMA2D 4.1.0.

BSP with VG-Lite

- *[VEE Port configuration project]*
  - Fetch C Module MicroUI-VGLite 7.2.0.

BSP with NemaGFX

- *[VEE Port configuration project]*
  - Fetch C Module MicroUI-VGLite 1.1.0.

From 13.5.x to 13.6.x

Front Panel

- Set the explicit dependency to the UI Pack 13.6.2:

```
<dependency org="com.microej.pack.ui" name="ui-pack" rev="13.6.2">
       <artifact name="frontpanel" type="jar"/>
</dependency>
```

BSP with VG-Lite

These steps are for a VEE Port that manages its own implementation of `LLUI_DISPLAY_impl.h` (that did not use the old implementation which was available in this C Module):

- *[VEE Port configuration project]*
  - Fetch C Module MicroUI-VGLite 7.1.0.
  - (optional) Fetch C Module MicroUI-Mimxrt595-EVK 7.0.0.
- *[BSP project]*
  - Delete the properties file `cco_microui-vglite.properties`.
  - Delete the following files from the file-system and from the C project configuration:
    * `inc/display_utils.h`
    * `inc/display_vglite.h`
    * `inc/drawing_vglite.h`
    * `inc/vglite_path.h`
    * `src/display_stub.c`
    * `src/display_utils.c`

    * src/display_vglite.c
    * src/drawing_vglite.c
    * src/vglite_path.c
  - Add the new files to the C project configuration:
    * src/ui_drawing_vglite_path.c
    * src/ui_drawing_vglite_process.c
    * src/ui_vglite.c
  - Review all imports of the removed header files.
  - In the implementation of LLUI_DISPLAY_impl.h, call UI_VGLITE_init() during the initialization step.
  - In the GPU interrupt rountine, call UI_VGLITE_IRQHandler().
  - Review all options of ui_vglite_configuration.h.
  - Implement UI_VGLITE_IMPL_notify_gpu_xxx() instead of DISPLAY_IMPL_notify_gpu_xxx().
  - Implement UI_VGLITE_IMPL_error() instead of DISPLAY_IMPL_error().
  - Change all calls to DISPLAY_VGLITE_xxx() functions to UI_VGLITE_xxx() functions.
  - Change all calls to DRAWING_VGLITE_xxx() functions to UI_DRAWING_VGLITE_PROCESS_xxx() functions.
  - Change all calls to VGLITE_PATH_xxx() functions to UI_DRAWING_VGLITE_PATH_xxx() functions.
  - Change all calls to DISPLAY_UTILS_xxx() functions to UI_VGLITE_xxx() functions.
- Build the VEE Port and the BSP.

BSP With MCU i.MX RT595

These steps are for a VEE Port that uses the implementation of LLUI_DISPLAY_impl.h which was available in the C Module *VGLite*:

- *[VEE Port configuration project]*
  - Fetch C Module MicroUI-VGLite 7.1.0.
  - Fetch C Module MicroUI-Mimxrt595-EVK 7.0.0.
- *[BSP project]*
  - Follow the steps of *BSP with VG-Lite* (described above) except the calls to UI_VGLITE_init() and UI_VGLITE_IRQHandler().
  - Implement DISPLAY_DMA_IMPL_notify_dma_xxx() instead of DISPLAY_IMPL_notify_dma_xxx().
- Build the VEE Port and the BSP.

From 13.4.x to 13.5.x

Front Panel

- Set the explicit dependency to the UI Pack 13.5.1:

```
<dependency org="com.microej.pack.ui" name="ui-pack" rev="13.5.1">
        <artifact name="frontpanel" type="jar"/>
</dependency>
```

- Replace any calls to `LLUIPainter.setDrawer()` and `LLDWPainter.setDrawer()` to `LLUIDisplay.Instance.registerUIDrawer()`.
- Replace any calls to `LLUIPainter.getDrawer()` and `LLDWPainter.getDrawer()` to `LLUIDisplay.Instance.getUIDrawer()`.
- Replace any calls to `LLUIDisplay.getDWDrawerSoftware()` to `LLUIDisplay.Instance.getUIDrawerSoftware()`.
- Implementation of the interface `UIDrawingDefault`: implement the interface `UIDrawing` instead.
- Implementation of the interfaces `DWDrawing` and `DWDrawingDefault`: implement the interface `UIDrawing` instead.
- Implementation of the service `BufferedImageProvider`: implement `handledFormat()` and remove the parameter `format` from `newBufferedImage()`.
- Replace any occurrences of `MICROUI_IMAGE_FORMAT_LCD` by `MICROUI_IMAGE_FORMAT_DISPLAY`.

BSP without GPU

- *[VEE Port configuration project]*
  - Fetch UI Pack 13.5.1 and C Module MicroUI 3.0.0.
- *[BSP project]*
  - Delete the VEE Port `include` folder (often `/platform/inc`).
  - Delete the properties file `cco_microui.properties`.
  - In the C project configuration, include the new C files `ui_drawing.c`, `ui_image_drawing.c` and `ui_drawing_stub.c`.
- Build the VEE Port and the BSP.

BSP with DMA2D

- Follow the migration steps of "BSP without GPU".
- *[VEE Port configuration project]*
  - Fetch C Module Display-DMA2D 4.0.0.
- *[BSP project]*
  - Delete the properties file `cco_display-dma2d.properties`.
  - Delete the C files `drawing_dma2d.h` and `drawing_dma2d.c` and remove them from the C project configuration.

  - In the C project configuration, include the new C file `ui_drawing_dma2d.c`.
  - Replace the import `drawing_dma2d.h` by `ui_drawing_dma2d.h`.
  - Replace the calls to functions `DRAWING_DMA2D_xxx()` by `UI_DRAWING_DMA2D_xxx()`.
- Build the VEE Port and the BSP.

BSP with VG-Lite

- Follow the migration steps of "BSP without GPU".
- *[VEE Port configuration project]*
  - Fetch C Module MicroUI-VGLite 6.0.1.
- *[BSP project]*
  - Delete the properties file `cco_microui-vglite.properties`.
  - Delete the C files `vg_drawer.h` and `vg_drawer.c` and remove them from the C project configuration.
  - Verify the options in `display_configuration.h`.
  - In the C project configuration, include the new C file `ui_drawing_vglite.c`.
- Build the VEE Port and the BSP.

From 13.3.x to 13.4.x

Front Panel

- Set the explicit dependency to the UI Pack 13.4.1:

```
<dependency org="com.microej.pack.ui" name="ui-pack" rev="13.4.1">
       <artifact name="frontpanel" type="jar"/>
</dependency>
```

BSP

- The signatures of `LLUI_DISPLAY_IMPL_decodeImage` and `LLUI_DISPLAY_IMPL_getNewImageStrideInBytes` have changed:
  - use `jbyte` in function signatures.
  - add a cast to `MICROUI_Image` in implementation.
- Use the MicroUI C module 2.0.1.

From 13.2.x to 13.3.x

Front Panel

- Set the explicit dependency to the UI Pack 13.3.1:

```
<dependency org="com.microej.pack.ui" name="ui-pack" rev="13.3.1">
	<artifact name="frontpanel" type="jar"/>
</dependency>
```

BSP

- Install the *MicroUI C module 2.0.0*.
- The signatures of `LLUI_DISPLAY_IMPL_decodeImage` and `LLUI_DISPLAY_IMPL_getNewImageStrideInBytes` have changed:
  - use `jbyte` in functions signatures
  - add a cast to `MICROUI_Image` in the implementation.

From 13.1.x to 13.2.x

Front Panel

- Set the explicit dependency to the UI Pack 13.2.0:

```
<dependency org="com.microej.pack.ui" name="ui-pack" rev="13.2.0">
	<artifact name="frontpanel" type="jar"/>
</dependency>
```

From 13.0.x to 13.1.x

Front Panel

- Set the explicit dependency to the UI Pack 13.1.0:

```
<dependency org="com.microej.pack.ui" name="ui-pack" rev="13.1.0">
	<artifact name="frontpanel" type="jar"/>
</dependency>
```

BSP

- Add a cast when using `MICROUI_Image*` object: `(MICROUI_ImageFormat)image->format` .
- Remove parameter `MICROUI_GraphicsContext*` when calling `LLUI_DISPLAY_setDrawingLimits()` .
- Ensure to call `LLUI_DISPLAY_setDrawingLimits()` before calling `LLUI_DISPLAY_setDrawingStatus()` or `LLUI_DISPLAY_notifyAsynchronousDrawingEnd()` .

- (optional) Add an implementation of `LLUI_DISPLAY_IMPL_image_heap_xxx` to control the *images heap allocation*; by default the internal Graphics Engine's allocator is used. Another implementation is also available on the *MicroUI C module*.

From 12.x to 13.x

VEE Port Configuration Project

- Update Architecture version: 7.16.0 or higher.
- Add the following module in the *module description file*:

```
<dependency org="com.microej.clibrary.llimpl" name="microui" rev="1.0.3"/>
```

- If not already set, set the `ea:property bsp.project.microej.dir` in the module ivy file to configure the BSP output folder where is extracted the module.

Hardware Accelerator

- Open `-configuration` project > `display` > `display.properties`
- Remove optional property `hardwareAccelerator`. If old value was `dma2d`, add the following module in the *module description file*:

```
<dependency org="com.microej.clibrary.llimpl" name="display-dma2d" rev="1.0.6"/>``
```

- For the hardware accelerator DMA2D, please consult STM32F7Discovery board updates. Add the file `lldisplay_dma2d.c`, the global defines `DRAWING_DMA2D_BPP=16` (or another value) and `STM32F4XX` or `STM32F7XX`
- For the others hardware accelerators, please contact MicroEJ support.

Front Panel

This chapter resumes the changes to perform. The available changes in Front Panel API are described in *next chapter*.

- If not already done, follow the Front Panel version 6 migration procedure detailled in chapter *From 11.x to 12.x*.
- Update the fp project dependency: `<dependency org="ej.tool.frontpanel" name="widget" rev="2.0.0"/>`
- `ej.fp.event.MicroUIButtons` has been renamed in `ej.microui.event.EventButton`, and all others `ej.fp.event.MicroUIxxx` in `ej.microui.event.Eventxxx`
- Display abstract class `AbstractDisplayExtension` (class to extend widget Display when targetting a custom display) has been converted on the interface `DisplayExtension`. Some methods names have changed and now take in parameter the display widget.

Front Panel API

- ej.drawing.DWDrawing
  - [Added] Equivalent of dw_drawing.h and dw_drawing_soft.h** : allows to implement some drawing algorithms and/or to use the ones provided by the Graphics Engine. The drawing methods are related to the library ej.api.drawing .
  - [Added] Interface DWDrawingDefault : default implementation of DWDrawing which calls the Graphics Engine algorithms.
- ej.drawing.LLDWPainter
  - [Added] Equivalent of module com.microej.clibrary.llimpl#microui (LLDW_PAINTER_impl.c): implements all ej.api.drawing natives and redirect them to the interface DWDrawing .
  - [Added] setDrawer(DWDrawing) : allows to configure the implementation of DWDrawing the LLDWPainter has to use. When no drawer is configured, LLDWPainter redirects all drawings to the internal Graphics Engine software algorithms.
- ej.fp.event.MicroUIButtons
  - [Removed] Replaced by EventButton .
- ej.fp.event.MicroUICommand
  - [Removed] Replaced by EventCommand .
- ej.fp.event.MicroUIEventGenerator
  - [Removed] Replaced by LLUIInput .
- ej.fp.event.MicroUIGeneric
  - [Removed] Replaced by EventGeneric .
- ej.fp.event.MicroUIPointer
  - [Removed] Replaced by EventPointer .
- ej.fp.event.MicroUIStates
  - [Removed] Replaced by EventState .
- ej.fp.event.MicroUITouch
  - [Removed] Replaced by EventTouch .
- ej.fp.widget.MicroUIDisplay
  - [Removed] Replaced by LLUIDisplayImpl . Abstract widget display class has been replaced by an interface that a widget (which should simulate a display) has to implement to be compatible with the Graphics Engine.
  - [Removed] AbstractDisplayExtension , all available implementations and setExtensionClass(String) : the standard display formats (RGB565, etc.) are internally managed by the Graphics Engine. For generic formats, some APIs are available in LLUIDisplayImpl .
  - [Removed] finalizeConfiguration() , getDisplayHeight() , getDisplayWidth() , getDrawingBuffer() , setDisplayWidth(int) , setDisplayHeight(int) , start() : LLUIDisplayImpl is not an abstract widget anymore, these notions are widget dependent.
  - [Removed] flush() .
  - [Removed] getNbBitsPerPixel() .

  - [Removed] `switchBacklight(boolean)` .

- `ej.fp.widget.MicroUILED`
  - [Removed] Replaced by `LLUILedImpl` . Abstract widget LED class has been replaced by an interface that a widget (which should simulate a LED) has to implement to be compatible with the Graphics Engine.
  - [Removed] `finalizeConfiguration()` : `LLUILedImpl` is not an abstract widget anymore, this notion is widget dependent.
  - [Removed] `getID()` : MicroUI uses the widget (which implements the interface `LLUILedImpl`)'s label to retrieve the LED. The LED labels must be integers from 0 to `n-1` .
- `ej.microui.display.LLUIDisplay`
  - [Added] Equivalent of `LLUI_DISPLAY.h` : several functions to interact with the Graphics Engine.
  - [Added] `blend(int,int,int)` : blends two ARGB colors and opacity level.
  - [Added] `convertARGBColorToColorToDraw(int)` : crops given color to display capacities.
  - [Added] `getDisplayPixelDepth()` : replaces `MicroUIDisplay.getNbBitsPerPixel()` .
  - [Added] `getDWDrawerSoftware()` : gives the unique instance of Graphics Engine's internal software drawer (instance of `DWDrawing`).
  - [Added] `getUIDrawerSoftware()` : gives the unique instance of Graphics Engine's internal software drawer (instance of `UIDrawing`).
  - [Added] `mapMicroUIGraphicsContext(byte[])` and `newMicroUIGraphicsContext(byte[])` : maps the graphics context byte array (GraphicsContext.getSNIContext()) on an object which represents the graphics context in front panel.
  - [Added] `mapMicroUIImage(byte[])` and `newMicroUIImage(byte[])` : maps the image byte array (Image.getSNIContext()) on an object which represents the image in front panel.
  - [Added] `requestFlush(boolean)` : requests a call to `LLUIDisplayImpl.flush()` .
  - [Added] `requestRender(void)` : requests a call to `Displayable.render()` .
- `ej.microui.display.LLUIDisplayImpl`
  - [Added] Replaces `MicroUIDisplay` , equivalent of `LLUI_DISPLAY_impl.h` .
  - [Added] `initialize()` : asks to initialize the widget and to return a front panel image where the Graphics Engine will perform the MicroUI drawings.
  - [Changed] `flush(MicroUIGraphicsContext, Image, int, int, int, int)` : asks to flush the graphics context drawn by MicroUI in image returned by `initialize()` .
- `ej.microui.display.LLUIPainter`
  - [Added] Equivalent of module com.microej.clibrary.llimpl#microui (`LLUI_PAINTER_impl.c`): implements all `ej.api.microui` natives and redirect them to the interface `UIDrawing` .
  - [Added] `MicroUIGraphicsContext` : representation of a MicroUI GraphicsContext in front panel. This interface (implemented by the Graphics Engine) provides several function to get information on graphics context, clip, etc.
  - [Added] `MicroUIGraphicsContext#requestDrawing()` : allows to take the hand on the drawing buffer.
  - [Added] `MicroUIImage` : representation of a MicroUI Image in front panel. This interface (implemented by the Graphics Engine) provides several function to get information on image.

  - [Added] `setDrawer(UIDrawing)` : allows to configure the implementation of `UIDrawing` the `LLUIPainter` has to use. When no drawer is configured, `LLUIPainter` redirects all drawings to the internal Graphics Engine software algorithms.
  -

- `ej.microui.display.UIDrawing`
  - [Added] Equivalent of `ui_drawing.h` and `ui_drawing_soft.h**` : allows to implement some drawing algorithms and/or to use the ones provided by the Graphics Engine. The drawing methods are related to the library `ej.api.microui` .
  - [Added] Interface `UIDrawingDefault` : default implementation of `UIDrawing` which calls the Graphics Engine algorithms.
- `ej.microui.event.EventButton`
  - [Added] Replaces `MicroUIButton` .
- `ej.microui.event.EventCommand`
  - [Added] Replaces `MicroUICommand` .
- `ej.microui.event.EventGeneric`
  - [Added] Replaces `MicroUIGeneric` .
- `ej.microui.event.EventPointer`
  - [Added] Replaces `MicroUIPointer` .
- `ej.microui.event.EventQueue`
  - [Added] Dedicated events queue used by MicroUI.
- `ej.microui.event.EventState`
  - [Added] Replaces `MicroUIState` .
- `ej.microui.event.EventTouch`
  - [Added] Replaces `MicroUITouch` .
- `ej.microui.event.LLUIInput`
  - [Added] Replaces `MicroUIEventGenerator` .
- `ej.microui.led.LLUILedImpl`
  - [Added] Replaces `MicroUILED` .

Image Generator

This chapter resumes the changes to perform. The available changes in Image Generator API are described in *next chapter*.

This chapter only concerns VEE Port with a custom display. In this case a dedicated image generator extension project is available. This project must be updated.

- Reorganize project to use source folders `src/main/java` and `src/main/resources`
- Add new `module.ivy` file:

```
<ivy-module version="2.0" xmlns:ea="http://www.easyant.org" xmlns:m="http://www.
↪easyant.org/ivy/maven" xmlns:ej="https://developer.microej.com" ej:version="2.
↪0.0">

   <info organisation="com.is2t.microui" module="imageGenerator-xxx" status=
↪"integration" revision="1.0.0">
      <ea:build organisation="com.is2t.easyant.buildtypes" module="build-std-
↪javalib" revision="2.+"/>
   </info>

   <configurations defaultconfmapping="default->default;provided->provided">
      <conf name="default" visibility="public" description="Runtime␣
↪dependencies to other artifacts"/>
      <conf name="provided" visibility="public" description="Compile-time␣
↪dependencies to APIs provided by the VEE Port"/>
      <conf name="documentation" visibility="public" description="Documentation␣
↪related to the artifact (javadoc, PDF)"/>
      <conf name="source" visibility="public" description="Source code"/>
      <conf name="dist" visibility="public" description="Contains extra files␣
↪like README.md, licenses"/>
      <conf name="test" visibility="private" description="Dependencies for test␣
↪execution. It is not required for normal use of the application, and is only␣
↪available for the test compilation and execution phases."/>
   </configurations>

   <publications/>

   <dependencies>
      <dependency org="com.microej.pack.ui" name="ui-pack" rev="[UI Pack␣
↪version]">
         <artifact name="imageGenerator" type="jar"/>
      </dependency>
   </dependencies>
</ivy-module>
```

The artifact name prefix must be `imageGenerator-`.

- Update project classpath: remove classpath variable `IMAGE-GENERATOR-x.x` and add ivy file dependency
- Instead of implementing `GenericDisplayExtension`, the extension class must extend `BufferedImageLoader` class; check class methods to override.
- Add the file `src/main/resources/META-INF/services/com.microej.tool.ui.generator.MicroUIRawImageGeneratorExtension`; this file has to specify the class which extends the `BufferedImageLoader` class, for instance:

  ```
  com.microej.generator.MyImageGeneratoExtension
  ```

- Build the easyant project
- Copy the jar in the VEE Port `-configuration` project > `dropins` folder
- Rebuild the VEE Port after any changes

Image Generator API

- `com.is2t.microej.microui.image.CustomDisplayExtension`
  - [Removed] Replaced by `ImageConverter` and `MicroUIRawImageGeneratorExtension`.
- `com.is2t.microej.microui.image.DisplayExtension`
  - [Removed]
- `com.is2t.microej.microui.image.GenericDisplayExtension`
  - [Removed] Replaced by `ImageConverter` and `MicroUIRawImageGeneratorExtension`.
- `com.microej.tool.ui.generator.BufferedImageLoader`
  - [Added] Pixelated image loader (PNG, JPEG etc.).
- `com.microej.tool.ui.generator.Image`
  - [Added] Representation of an image listed in a `images.list` file.
- `com.microej.tool.ui.generator.ImageConverter`
  - [Added] Generic converter to convert an image in an output stream.
- `com.microej.tool.ui.generator.MicroUIRawImageGeneratorExtension`
  - [Added] Graphics Engine RAW image converter: used when the image (listed in `images.list`) targets a RAW format known by the Graphics Engine.

Font

- Open optional font(s) in `-configuration` project > `microui/**/*.ejf`
- Remove all `Dynamic` styles (select `None` or `Built-in` for bold, italic and underline); the number of generated fonts must be `1` (the feature to render `Dynamic` styles at runtime have been removed)
- Save the file(s)

BSP

This chapter resumes the changes to perform. The available changes in LLAPI are described in *next chapter*.

- Delete all VEE Port header files (folder should be set in `-configuration` project > `bsp` > `bsp.properties` > property `output.dir`)
- If not possible to delete this folder, delete all UI headers files:
  - `intern/LLDISPLAY*`
  - `intern/LLINPUT*`
  - `intern/LLLEDS*`
  - `LLDISPLAY*`
  - `LLINPUT*`
  - `LLLEDS*`
- Replace all `#include "LLDISPLAY.h"`, `#include "LLDISPLAY_EXTRA.h"` and `#include "LLDISPLAY_UTILS.h"` by `#include "LLUI_DISPLAY.h"`

- Replace all `#include "LLDISPLAY_impl.h"`, `#include "LLDISPLAY_EXTRA_drawing.h"` and `#include "LLDISPLAY_EXTRA_impl.h"` by `#include "LLUI_DISPLAY_impl.h"`
- Replace all `LLDISPLAY_EXTRA_IMAGE_xxx` by `MICROUI_IMAGE_FORMAT_xxx`
- All `LLDISPLAY_IMPL_xxx` functions have been renamed in `LLUI_DISPLAY_IMPL_xxx`
- `LLUI_DISPLAY_IMPL_initialize` has now the paremeter `LLUI_DISPLAY_SInitData* init_data`; fill it as explained in C doc.
- Implement new functions `void LLUI_DISPLAY_IMPL_binarySemaphoreTake(void* sem)` and `void LLUI_DISPLAY_IMPL_binarySemaphoreGive(void* sem, bool under_isr)`
- Signature of `LLUI_DISPLAY_IMPL_flush` has changed
- All `LLDISPLAY_EXTRA_IMPL_xxx` functions have been renamed in `LLUI_DISPLAY_IMPL_xxx`
- Fix some functions signatures (`LLUI_DISPLAY_IMPL_hasBacklight()`, etc)
- Remove the functions `LLDISPLAY_IMPL_getGraphicsBufferAddress`, `LLDISPLAY_IMPL_getHeight`, `LLDISPLAY_IMPL_getWidth`, `LLDISPLAY_IMPL_synchronize`, `LLDISPLAY_EXTRA_IMPL_waitPreviousDrawing`, `LLDISPLAY_EXTRA_IMPL_error`
- Add the end of asynchronous flush copy, call `LLUI_DISPLAY_flushDone`
- Add the files `LLUI_PAINTER_impl.c` and `LLDW_PAINTER_impl.c` in your C configuration project
- Replace the prefix `LLINPUT` in all header files, functions and defines by the new prefix `LLUI_INPUT`
- Replace the prefix `LLLEDS` in all header files, functions and defines by the new prefix `LLUI_LED`
- Replace the prefix `LLDISPLAY` in all header files, functions and defines by the new prefix `LLUI_DISPLAY`

LLAPI

- `dw_drawing_soft.h`
  - [Added] List of internal Graphics Engine software algorithms to perform some drawings (related to library `ej.api.drawing`).
- `dw_drawing.h`
  - [Added] List of `ej.api.drawing` library's drawing functions to optionally implement in VEE Port.
- `LLDISPLAY.h` and `intern/LLDISPLAY.h`
  - [Removed]
- `LLDISPLAY_DECODER.h` and `intern/LLDISPLAY_DECODER.h`
  - [Removed]
- `LLDISPLAY_EXTRA.h` and `intern/LLDISPLAY_EXTRA.h` merged in `LLUI_PAINTER_impl.h` and `LLDW_PAINTER_impl.h`
  - [Changed] `LLDISPLAY_SImage`: replaced by `MICROUI_Image`.
  - [Removed] `LLDISPLAY_SRectangle`, `LLDISPLAY_SDecoderImageData`, `LLDISPLAY_SDrawImage`, `LLDISPLAY_SFlipImage`, `LLDISPLAY_SScaleImage` and `LLDISPLAY_SRotateImage`
- `LLDISPLAY_EXTRA_drawing.h`
  - [Removed]
- `LLDISPLAY_EXTRA_impl.h` and `intern/LLDISPLAY_EXTRA_impl.h` merged in `LLUI_DISPLAY_impl.h`, `ui_drawing.h` and `dw_drawing.h`

- [Changed] `LLDISPLAY_EXTRA_IMPL_setContrast(int32_t)` : replaced by `LLUI_DISPLAY_IMPL_setContrast(uint32_t)` (_optional_).
- [Changed] `LLDISPLAY_EXTRA_IMPL_getContrast(void)` : replaced by `LLUI_DISPLAY_IMPL_getContrast(void)` (_optional_).
- [Changed] `LLDISPLAY_EXTRA_IMPL_hasBackLight(void)` : replaced by `LLUI_DISPLAY_IMPL_hasBacklight(void)` (_optional_).
- [Changed] `LLDISPLAY_EXTRA_IMPL_setBacklight(int32_t)` : replaced by `LLUI_DISPLAY_IMPL_setBacklight(uint32_t)` (_optional_).
- [Changed] `LLDISPLAY_EXTRA_IMPL_getBacklight(void)` : replaced by `LLUI_DISPLAY_IMPL_getBacklight(void)` (_optional_).
- [Changed] `LLDISPLAY_EXTRA_IMPL_isColor(void)` : replaced by `LLUI_DISPLAY_IMPL_isColor(void)` (_optional_).
- [Changed] `LLDISPLAY_EXTRA_IMPL_getNumberOfColors(void)` : replaced by `LLUI_DISPLAY_IMPL_getNumberOfColors(void)` (_optional_).
- [Changed] `LLDISPLAY_EXTRA_IMPL_isDoubleBuffered(void)` : replaced by `LLUI_DISPLAY_IMPL_isDoubleBuffered(void)` (_optional_).
- [Changed] `LLDISPLAY_EXTRA_IMPL_getBacklight(void)` : replaced by `LLUI_DISPLAY_IMPL_getBacklight(void)` (_optional_).
- [Changed] `LLDISPLAY_EXTRA_IMPL_fillRect(void*,int32_t,void*,int32_t)` : replaced by `UI_DRAWING_fillRectangle(MICROUI_GraphicsContext*,jint,jint,jint,jint)` (_optional_).
- [Changed] `LLDISPLAY_EXTRA_IMPL_drawImage(void*,int32_t,void*,int32_t,void*)` : replaced by `UI_DRAWING_drawImage(MICROUI_GraphicsContext*,MICROUI_Image*,jint,jint,jint,jint,jint,jint,jint)` (_optional_).
- [Changed] `LLDISPLAY_EXTRA_IMPL_flipImage(void*,int32_t,void*,int32_t,void*)` : replaced by `DW_DRAWING_drawFlippedImage(MICROUI_GraphicsContext*,MICROUI_Image*,jint,jint,jint,jint,jint,jint,DRAWING_Flip,jint)` (_optional_).
- [Changed] `LLDISPLAY_EXTRA_IMPL_scaleImage(void*,int32_t,void*,int32_t,void*)` : replaced by `DW_DRAWING_drawScaledImageNearestNeighbor(MICROUI_GraphicsContext*,MICROUI_Image*,jint,jint,jfloat,jfloat,jint)` and `DW_DRAWING_drawScaledImageBilinear(MICROUI_Graphics MICROUI_Image*,jint,jint,jfloat,jfloat,jint)` (_optional_).
- [Changed] `LLDISPLAY_EXTRA_IMPL_rotateImage(void*,int32_t,void*,int32_t,void*)` : replaced by `DW_DRAWING_drawRotatedImageNearestNeighbor(MICROUI_GraphicsContext*,MICROUI_Image*,jint,jint,jint,jint,jfloat,jint)` and `DW_DRAWING_drawRotatedImageBilinear(MICROUI_Grap MICROUI_Image*,jint,jint,jint,jint,jfloat,jint)` (_optional_).
- [Changed] `LLDISPLAY_EXTRA_IMPL_convertARGBColorToDisplayColor(int32_t)` and `LLDISPLAY_EXTRA_IMPL_convertDisplayColorToARGBColor(int32_t)` : replaced respectively by `LLUI_DISPLAY_IMPL_convertARGBColorToDisplayColor(uint32_t)` and `LLUI_DISPLAY_IMPL_convertDisplayColorToARGBColor(uint32_t)` (_optional_).
- [Changed] `LLDISPLAY_EXTRA_IMPL_prepareBlendingOfIndexedColors(void*,void*)` : replaced by `LLUI_DISPLAY_IMPL_prepareBlendingOfIndexedColors(uint32_t*,uint32_t*)` (_optional_).
- [Changed] `LLDISPLAY_EXTRA_IMPL_decodeImage(int32_t,int32_t,int32_t,void*)` : replaced by `LLUI_DISPLAY_IMPL_decodeImage(uint8_t*,uint32_t,MICROUI_ImageFormat,MICROUI_Image*,bool*)` (_optional_).

  - [Removed] LLDISPLAY_EXTRA_IMPL_getGraphicsBufferMemoryWidth(void) and LLDISPLAY_EXTRA_IMPL_getGraphicsBufferMemoryHeight(void) : replaced by elements in structure LLUI_DISPLAY_SInitData (_optional_).
  - [Removed] LLDISPLAY_EXTRA_IMPL_backlightOn(void) and LLDISPLAY_EXTRA_IMPL_backlightOff(void) .
  - [Removed] LLDISPLAY_EXTRA_IMPL_enterDrawingMode(void) and LLDISPLAY_EXTRA_IMPL_exitDrawingMode(void) .
  - [Removed] LLDISPLAY_EXTRA_IMPL_error(int32_t) .
  - [Removed] LLDISPLAY_EXTRA_IMPL_waitPreviousDrawing(void) : implementation has to call LLUI_DISPLAY_notifyAsynchronousDrawingEnd(bool) instead.
- LLDISPLAY_impl.h and intern/LLDISPLAY_impl.h merged in LLUI_DISPLAY_impl.h
  - [Changed] LLDISPLAY_IMPL_initialize(void) : replaced by LLUI_DISPLAY_IMPL_initialize(LLUI_DISPLAY_SInitD (_mandatory_).
  - [Changed] LLDISPLAY_IMPL_flush(int32_t,int32_t,int32_t,int32_t,int32_t) : replaced by LLUI_DISPLAY_IMPL_flush(MICROUI_GraphicsContext*,uint8_t*, uint32_t,uint32_t, uint32_t,uint32_t) (_mandatory_).
  - [Removed] LLDISPLAY_IMPL_getWidth(void) , LLDISPLAY_IMPL_getHeight(void) and LLDISPLAY_IMPL_getGraphicsBufferAddress(void) : replaced by elements in structure LLUI_DISPLAY_SInitData .
  - [Removed] LLDISPLAY_IMPL_synchronize(void) : implementation has to call LLUI_DISPLAY_flushDone(bool) instead.
- LLDISPLAY_UTILS.h and intern/LLDISPLAY_UTILS.h merged in LLUI_DISPLAY.h
  - [Changed] LLDISPLAY_UTILS_getBufferAddress(int32_t) : replaced by LLUI_DISPLAY_getBufferAddress(MICROUI_Image*) .
  - [Changed] LLDISPLAY_UTILS_setDrawingLimits(int32_t,int32_t,int32_t,int32_t,int32_t) : replaced by LLUI_DISPLAY_setDrawingLimits(MICROUI_GraphicsContext*,jint,jint,jint, jint) .
  - [Changed] LLDISPLAY_UTILS_blend(int32_t,int32_t,int32_t) : replaced by LLUI_DISPLAY_blend(uint32_t,uint32_t,uint32_t) .
  - [Changed] LLDISPLAY_UTILS_allocateDecoderImage(void*) : replaced by LLUI_DISPLAY_allocateImageBuffer(MICROUI_Image*,uint8_t) .
  - [Changed] LLDISPLAY_UTILS_flushDone(void) : replaced by LLUI_DISPLAY_flushDone(bool) .
  - [Changed] LLDISPLAY_UTILS_drawingDone(void) : replaced by LLUI_DISPLAY_notifyAsynchronousDrawingEnd(bool) .
  - [Removed] LLDISPLAY_UTILS_getWidth(int32_t) , LLDISPLAY_UTILS_getHeight(int32_t) and LLDISPLAY_UTILS_getFormat(int32_t) : use MICROUI_Image elements instead.
  - [Removed] LLDISPLAY_UTILS_enterDrawingMode(void) and LLDISPLAY_UTILS_exitDrawingMode(void) .
  - [Removed] LLDISPLAY_UTILS_setClip(int32_t,int32_t,int32_t,int32_t,int32_t) .
  - [Removed] LLDISPLAY_UTILS_getClipX1/X2/Y1/Y2(int32_t) : use MICROUI_GraphicsContext elements instead.
  - [Removed] LLDISPLAY_UTILS_drawPixel(int32_t,int32_t,int32_t) and LLDISPLAY_UTILS_readPixel(int32_t,int32_t,int32_t) .

- LLDW_PAINTER_impl.h
  - [Added] List of ej.api.drawing library's native functions implemented in module com.microej.clibrary.llimpl#microui.
- LLLEDS_impl.h and intern/LLLEDS_impl.h merged in LLUI_LED_impl.h
  - [Changed] LLLEDS_MIN_INTENSITY and LLLEDS_MAX_INTENSITY : replaced respectively by LLUI_LED_MIN_INTENSITY and LLUI_LED_MAX_INTENSITY.
  - [Changed] LLLEDS_IMPL_initialize(void) : replaced by LLUI_LED_IMPL_initialize(void).
  - [Changed] LLLEDS_IMPL_getIntensity(int32_t) : replaced by LLUI_LED_IMPL_getIntensity(jint).
  - [Changed] LLLEDS_IMPL_setIntensity(int32_t,int32_t) : replaced by LLUI_LED_IMPL_setIntensity(jint,jint).
- LLINPUT.h and intern/LLINPUT.h merged in LLUI_INPUT.h
  - [Changed] LLINPUT_sendEvent(int32_t,int32_t) : replaced by LLUI_INPUT_sendEvent(jint,jint).
  - [Changed] LLINPUT_sendEvents(int32_t,int32_t*,int32_t) : replaced by LLUI_INPUT_sendEvents(jint,jint*,jint).
  - [Changed] LLINPUT_sendCommandEvent(int32_t,int32_t) : replaced by LLUI_INPUT_sendCommandEvent(jint,jint).
  - [Changed] LLINPUT_sendButtonPressedEvent(int32_t,int32_t) : replaced by LLUI_INPUT_sendButtonPressedEvent(jint,jint).
  - [Changed] LLINPUT_sendButtonReleasedEvent(int32_t,int32_t) : replaced by LLUI_INPUT_sendButtonReleasedEvent()jint,jint.
  - [Changed] LLINPUT_sendButtonRepeatedEvent(int32_t,int32_t) : replaced by LLUI_INPUT_sendButtonRepeatedEvent(jint,jint).
  - [Changed] LLINPUT_sendButtonLongEvent(int32_t,int32_t) : replaced by LLUI_INPUT_sendButtonLongEvent(jint,jint).
  - [Changed] LLINPUT_sendPointerPressedEvent(int32_t,int32_t,int32_t,int32_t,int32_t) : replaced by LLUI_INPUT_sendPointerPressedEvent(jint,jint,jint,jint,LLUI_INPUT_Pointer).
  - [Changed] LLINPUT_sendPointerReleasedEvent(int32_t,int32_t) : replaced by LLUI_INPUT_sendPointerReleasedEvent(jint,jint).
  - [Changed] LLINPUT_sendPointerMovedEvent(int32_t,int32_t,int32_t,int32_t) : replaced by LLUI_INPUT_sendPointerMovedEvent(jint,jint,jint,LLUI_INPUT_Pointer).
  - [Changed] LLINPUT_sendTouchPressedEvent(int32_t,int32_t,int32_t) : replaced by LLUI_INPUT_sendTouchPressedEvent(jint,jint,jint).
  - [Changed] LLINPUT_sendTouchReleasedEvent(int32_t) : replaced by LLUI_INPUT_sendTouchReleasedEvent(jint).
  - [Changed] LLINPUT_sendTouchMovedEvent(int32_t,int32_t,int32_t) : replaced by LLUI_INPUT_sendTouchMovedEvent(jint,jint,jint).
  - [Changed] LLINPUT_sendStateEvent(int32_t,int32_t,int32_t) : replaced by LLUI_INPUT_sendStateEvent(jint,jint,jint).
  - [Changed] LLINPUT_getMaxEventsBufferUsage(void) : replaced by LLUI_INPUT_getMaxEventsBufferUsage(void).

- LLINPUT_impl.h and intern/LLINPUT_impl.h merged in LLUI_INPUT_impl.h
  - [Changed] LLINPUT_IMPL_initialize(void) : replaced by LLUI_INPUT_IMPL_initialize(void) (_mandatory_).
  - [Changed] LLINPUT_IMPL_getInitialStateValue(int32_t,int32_t) : replaced by LLUI_INPUT_IMPL_getInitialStateValue(jint,jint) (_mandatory_).
  - [Changed] LLINPUT_IMPL_enterCriticalSection(void) : replaced by LLUI_INPUT_IMPL_enterCriticalSection(void) (_mandatory_).
  - [Changed] LLINPUT_IMPL_leaveCriticalSection(void) : replaced by LLUI_INPUT_IMPL_leaveCriticalSection(void) (_mandatory_).
- LLUI_DISPLAY.h
  - [Added] Renaming of LLDISPLAY_UTILS.h .
  - [Added] Several functions to interact with the Graphics Engine and to get information on images, graphics context, clip, etc.
  - [Added] LLUI_DISPLAY_requestFlush(bool) : requests a call to LLUI_DISPLAY_IMPL_flush() .
  - [Added] LLUI_DISPLAY_requestRender(void) : requests a call to Displayable.render() .
  - [Added] LLUI_DISPLAY_freeImageBuffer(MICROUI_Image*) : frees an image previously allocated by LLUI_DISPLAY_allocateImageBuffer(MICROUI_Image*,uint8_t) .
  - [Added] LLUI_DISPLAY_requestDrawing(MICROUI_GraphicsContext*,SNI_callback) : allows to take the hand on the shared drawing buffer.
  - [Added] LLUI_DISPLAY_setDrawingStatus(DRAWING_Status) : specifies the drawing status to the Graphics Engine.
- LLUI_DISPLAY_impl.h
  - [Added] Merge of LLDISPLAY_EXTRA_impl.h and LLDISPLAY_impl.h .
  - [Added] Structure LLUI_DISPLAY_SInitData : implementation has to fill it in LLUI_DISPLAY_IMPL_initialize(LLUI_DISPLAY_SInitData*) .
  - [Added] LLUI_DISPLAY_IMPL_binarySemaphoreTake(void*) and LLUI_DISPLAY_IMPL_binarySemaphoreGive(void*,bool) : implementation has to manage a binary semaphore (_mandatory_).
  - [Added] LLUI_DISPLAY_IMPL_getNewImageStrideInBytes(MICROUI_ImageFormat,uint32_t,uint32_t,uint32_t) : allows to set an image stride different than image side (_optional_).
- LLUI_PAINTER_impl.h
  - [Added] List of ej.api.microui library's native functions implemented in module com.microej.clibrary.llimpl#microui.
  - [Added] MICROUI_ImageFormat : MicroUI Image pixel format.
  - [Added] MICROUI_Image : MicroUI Image representation.
  - [Added] MICROUI_GraphicsContext : MicroUI GraphicsContext representation.
- ui_drawing_soft.h
  - [Added] List of internal Graphics Engine software algorithms to perform some drawings (related to library ej.api.microui).
- ui_drawing.h
  - [Added] List of ej.api.microui library's drawing functions to optionally implement in VEE Port.

Custom Native Drawing Functions

- In custom UI native methods, replace `LLDISPLAY_UTILS_getBufferAddress(xxx);` by `(uint32_t)LLUI_DISPLAY_getBufferAddress(xxx)` (new function returns `uint8_t*`), where `uint32_t xxx` is replaced by `MICROUI_Image* xxx` or by `MICROUI_GraphicsContext* xxx`.
- Replace `LLDISPLAY_UTILS_getFormat(xxx)` by `xxx->format`, where `uint32_t xxx` is replaced by `MICROUI_Image* xxx` or by `MICROUI_GraphicsContext* xxx`.
- Replace call to `LLDISPLAY_allocateDecoderImage` by a call to `LLUI_DISPLAY_allocateImageBuffer`
- Optional: implement drawing functions listed in `ui_drawing.h` following the available examples in `LLUI_PAINTER_impl.c` and `LLDW_PAINTER_impl.c` files comments.

Application

- See application *Migration Guide*.

From 11.x to 12.x

VEE Port Configuration Project

- Update Architecture version: 7.11.0 or higher.

Front Panel

- Create a new Front Panel Project (next sections explain how to update each widget):
  1. Verify that FrontPanelDesigner is at least version 6: Help > About > Installations Details > Plug-ins.
  2. Create a new front panel project: File > New > Project... > MicroEJ > MicroEJ Front Panel Project, choose a name and press Finish.
  3. Move files from `[old project]/src` to `[new project]/src/main/java`.
  4. Move files from `[old project]/resources` to `[new project]/src/main/resources`.
  5. Move files from `[old project]/definitions` to `[new project]/src/main/resources`, **except** your `xxx.fp` file.
  6. If existing delete file `[new project]/src/main/java/microui.properties`.
  7. Delete file `[new project]/src/main/resources/.fp.xsd`.
  8. Delete file `[new project]/src/main/resources/.fp1.0.xsd`.
  9. Delete file `[new project]/src/main/resources/widgets.desc`.
  10. Open `[old project]/definitions/xxx.fp`.
  11. Copy `device` attributes (`name` and `skin`) from `[old project]/definitions/xxx.fp` to `[new project]/src/main/resources/xxx.fp`.
  12. Copy content of `body` (not `body` tag itself) from `[old project]/definitions/xxx.fp` under `device` group of `[new project]/src/main/resources/xxx.fp`.

- Widget “led2states”:
  1. Rename `led2states` by `ej.fp.widget.LED`.
  2. Rename the attribute `id` by `label`.
- Widget “pixelatedDisplay”:
  1. Rename `pixelatedDisplay` by `ej.fp.widget.Display`.
  2. Remove the attribute `id`.
  3. (*if set*) Remove the attribute `initialColor` if its value is `0`
  4. (*if set*) Rename the attribute `mask` by `filter`; this image must have the same size in pixels than display itself (`width * height`).
  5. (*if set*) Rename the attribute `realWidth` by `displayWidth`.
  6. (*if set*) Rename the attribute `realHeight` by `displayHeight`.
  7. (*if set*) Rename the attribute `transparencyLevel` by `alpha`; change the value: `newValue = 255 - oldValue`.
  8. (*if set*) Remove the attribute `residualFactor` (not supported).
  9. (*if set*) If `extensionClass` is specified: follow next notes.
- Widget “pixelatedDisplay”: `ej.fp.widget.Display` Extension Class:
  1. Open the class
  2. Extends `ej.fp.widget.MicroUIDisplay.AbstractDisplayExtension` instead of `com.is2t.microej.frontpanel.display.DisplayExtension`.
  3. Rename method `convertDisplayColorToRGBColor` to `convertDisplayColorToARGBColor`.
  4. Rename method `convertRGBColorToDisplayColor` to `convertARGBColorToDisplayColor`.
- Widget “pointer”:
  1. Rename `pointer` by `ej.fp.widget.Pointer`.
  2. Remove the attribute `id`.
  3. (*if set*) Rename the attribute `realWidth` by `areaWidth`.
  4. (*if set*) Rename the attribute `realHeight` by `areaHeight`.
  5. Keep or remove the attribute `listenerClass` according next notes.
- Widget “pointer”: `ej.fp.widget.Pointer` Listener Class:

  This extension class is useless if the implementation respects these rules:

  - *(a)* `press` method is sending a `press` MicroUI Pointer event.
  - *(b)* `release` method is sending a `release` MicroUI Pointer event.
  - *(c)* `move` method is sending a `move` MicroUI Pointer event.
  - *(d)* The MicroUI Pointer event generator name is `POINTER` when `ej.fp.widget.Pointer`'s `touch` attribute is `false` (or not set).
  - *(e)* The MicroUI Pointer event generator name is `TOUCH` when `ej.fp.widget.Pointer`'s `touch` attribute is `true`.

  If only *(d)* or *(e)* is different:

  1. Open the listener class.

2. Extends the class `ej.fp.widget.Pointer.PointerListenerToPointerEvents` instead of implementing the interface `com.is2t.microej.frontpanel.input.listener.PointerListener`.
3. Implements the method `getMicroUIGeneratorTag()`.

In all other cases:

1. Open the listener class.
2. Implements the interface `ej.fp.widget.Pointer.PointerListener` instead of `com.is2t.microej.frontpanel.input.listener.PointerListener`.

- Widget "push":
  1. Rename `push` by `ej.fp.widget.Button`.
  2. Rename the attribute `id` by `label`.
  3. (*if set*) Review `filter` image: this image must have the same size in pixels than the button `skin`.
  4. (*if set*) Remove the attribute `hotkey` (not supported).
  5. Keep or remove the attribute `listenerClass` according next notes.
- Widget "push": `ej.fp.widget.Button` Listener Class:

  This extension class is useless if the implementation respects these rules:

  - *(a)* `press` method is sending a `press` MicroUI Buttons event with button `label` (equals to old button `id`) as button index.
  - *(b)* `release` method is sending a `release` MicroUI Buttons event with button `label` (equals to old button `id`) as button index.
  - *(c)* The MicroUI Buttons event generator name is `BUTTONS`.

  If only *(c)* is different:

  1. Open the listener class.
  2. Extends the class `ej.fp.widget.Button.ButtonListenerToButtonEvents` instead of implementing the interface `com.is2t.microej.frontpanel.input.listener.ButtonListener`.
  3. Overrides the method `getMicroUIGeneratorTag()`.

  In all other cases:

  1. Open the listener class.
  2. Implements the interface `ej.fp.widget.Button.ButtonListener` instead of `com.is2t.microej.frontpanel.input.listener.ButtonListener`.
- Widget "repeatPush":
  1. Rename `repeatPush` by `ej.fp.widget.RepeatButton`.
  2. (*if set*) Remove the attribute `sendPressRelease` (not supported).
  3. Same rules than widget *push*.
- Widget "longPush":
  1. Rename `longPush` by `ej.fp.widget.LongButton`.
  2. Same rules than widget *push*.
- Widget "joystick":

1. Rename `joystick` by `ej.fp.widget.Joystick`.
2. Remove the attribute `id`.
3. (*if set*) Rename the attribute `mask` by `filter`; this image must have the same size in pixels than joystick `skin`.
4. (*if set*) Remove the attribute `hotkeys` (not supported).
5. Keep or remove the attribute `listenerClass` according next notes.

- Widget "joystick": `ej.fp.widget.Joystick` Listener Class:

  This extension class is useless if the implementation respects these rules:

  - *(a)* `press` methods are sending some MicroUI Command events `UP`, `DOWN`, `LEFT`, `RIGHT` and `SELECT`.
  - *(b)* `repeat` methods are sending same MicroUI Command events `UP`, `DOWN`, `LEFT`, `RIGHT` and `SELECT`.
  - *(c)* `release` methods are sending nothing.
  - *(d)* The MicroUI Command event generator name is `JOYSTICK`.

  If only *(d)* is different:

  1. Open the listener class
  2. Extends the class `ej.fp.widget.Joystick.JoystickListenerToCommandEvents` instead of implementing the interface `com.is2t.microej.frontpanel.input.listener.JoystickListener`.
  3. Overrides the method `getMicroUIGeneratorTag()`.

  In all other cases:

  1. Open the listener class.
  2. Implements the interface `ej.fp.widget.Joystick.JoystickListener` instead of `com.is2t.microej.frontpanel.input.listener.JoystickListener`.

- Others Widgets:

  These widgets may have not been migrated. Check in `ej.tool.frontpanel.widget` library if some widgets are compatible or write your own widgets.

Application

- See application *Migration Guide*.

From 10.x to 11.x

VEE Port Configuration Project

- Update Architecture version: 7.0.0 or higher.

From 9.x to 10.x

VEE Port Configuration Project

- Update Architecture version: 6.13.0 or higher.
- Edit `display/display.properties`
- Add property `imagesHeap.size=xxx` ; this value fixes the images heap size when using the VEE Port in command line (to build a firmware)
- In VEE Port linker file (standalone mode with MicroEJ linker): remove the image heap reserved section and put the section `.bss.microui.display.imagesHeap` instead.

BSP

- In BSP linker file: remove the image heap reserved section and put the section `.bss.microui.display.imagesHeap` instead
- Edit `LLDISPLAY*.c` : remove the functions `LLDISPLAY_IMPL_getWorkingBufferStartAddress` and `LLDISPLAY_IMPL_getWorkingBufferEndAddress`

Application

- See application *Migration Guide*.

From 8.x to 9.x

Application

- See application *Migration Guide*.

From 7.x to 8.x

VEE Port Configuration Project

- Update Architecture version: 6.4.0 or higher.
- Edit `display/display.properties` : remove property `mode=xxx`

BSP

- Edit `LLDISPLAY*.c`
- For LLDISPLAY `SWITCH`
  - Remove the function `LLDISPLAY_SWITCH_IMPL_getDisplayBufferAddress()`
  - Replace the function `void LLDISPLAY_SWITCH_IMPL_getDisplayBufferAddress()` by `int32_t LLDISPLAY_IMPL_flush()`
  - In this function, return the old LCD frame buffer address

  - Replace the function `LLDISPLAY_COPY_IMPL_getBackBufferAddress()` by `LLDISPLAY_IMPL_getGraphicsBufferAddress()`

- For LLDISPLAY `COPY`
  - Replace the function `void LLDISPLAY_COPY_IMPL_copyBuffer()` by `int32_t LLDISPLAY_IMPL_flush()`
  - In this function, return the back buffer address (given in argument)
  - Replace the function `LLDISPLAY_COPY_IMPL_getBackBufferAddress()` by `LLDISPLAY_IMPL_getGraphicsBufferAddress()`
- For LLDISPLAY `DIRECT`
  - Add the function `void LLDISPLAY_IMPL_synchorize(void)` (do nothing)
  - Add the function `int32_t LLDISPLAY_IMPL_flush()`
  - In this function, just return the back buffer address (given in argument)
- Replace h file `LLDISPLAY_SWITCH_IMPL.h`, `LLDISPLAY_COPY_IMPL.h` or `LLDISPLAY_DIRECT_IMPL.h` by `LLDISPLAY_IMPL.h`
- Replace all functions `LLDISPLAY_SWITCH_IMPL_xxx`, `LLDISPLAY_COPY_IMPL_xxx` and `LLDISPLAY_DIRECT_IMPL_xxx` by `LLDISPLAY_IMPL_xxx`
- Remove the argument `int32_t type` from `getWidth` and `getHeight`

STM32 VEE Ports with DMA2D only

- In VEE Port configuration project, edit `display/display.properties`
- Add property `hardwareAccelerator=dma2d`
- In BSP project, edit `LLDISPLAY*.c`
- simplify following functions (see STM32F7Discovery board implementation)

```
LLDISPLAY_EXTRA_IMPL_fillRect
LLDISPLAY_EXTRA_IMPL_drawImage
LLDISPLAY_EXTRA_IMPL_waitPreviousDrawing
```

- Add the following function

```
void LLDISPLAY_EXTRA_IMPL_error(int32_t errorCode)
{
   printf("lldisplay error: %d\n", errorCode);
   while(1);
}
```

- Launch an application with images and fillrect
- Compile, link and debug the BSP
- Set some breakpoints on three functions
- Ensure the functions are called

6.15 Vector Graphics

Note: This chapter describes the VG Pack implementation. The current VG Pack only targets the i.MX RT595 MCU (it is part of the NXP i.MX RT500 crossover MCU product line that embeds the Vivante GCNanoLiteV IP from Verisilicon). Please contact our support team for other hardware accelerators (GPU with vector graphics acceleration).

6.15.1 Principle

The Vector Graphics Pack features an extension of the *User Interface Pack* that implements the *MicroVG API*.

The diagram below shows a simplified view of the components involved in the provisioning of Vector Graphics Extension.

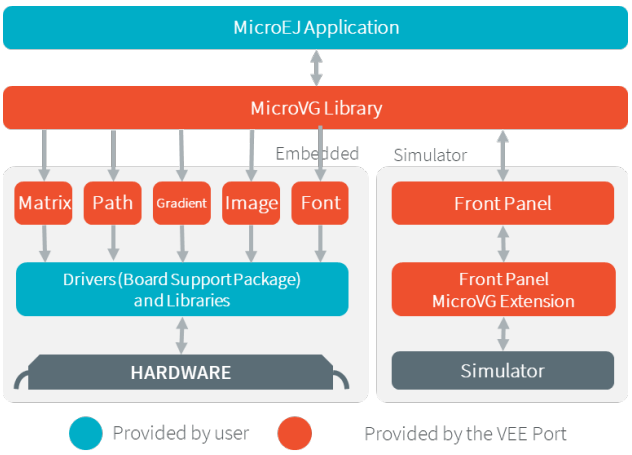


Fig. 59: Overview

The modules responsible to manage the Matrix, the Path, the Gradient, the Image and the Font are respectively called *Matrix module*, *Path module*, *Gradient module*, *Image module* and *Font module*. These five low-level parts connect the MicroVG library to the user-supplied drivers code (coded in C). The drivers can use hardware accelerators like GPU to perform specific actions (matrix computations, path rendering, font decoding, etc.).

The MicroEJ Simulator provides all features of the MicroVG library. The five modules are grouped in a module called *Front Panel*. The Front Panel is an extension of the *UI Pack's Front Panel mock*.

6.15.2 MicroVG

Principle

MicroVG library is an extension of the MicroUI library and provides vector drawing capabilities.

Architecture

MicroVG library is the entry point to perform some vectorial drawings on display. This library contains only a minimal set of basic APIs. As a result, high-level libraries can be used to have more expressive power. In addition to this restricted set of APIs, the MicroVG implementation has been designed to minimize the EDC, BON, and MicroUI footprint.

Native Calls

Like MicroUI, the MicroVG implementation for MicroEJ uses native methods to perform some actions (manipulate matrices, perform drawings, decode and render fonts, etc.). The library implementation has been designed not to block native methods (wait until the end of the drawing, etc.), which can lock the complete MicroEJ Core Engine execution.

Refer to the *MicroUI implementation* to have more details about the native calls.

Installation

The MicroVG library is an additional module. In the VEE Port configuration's *module description file*, add the VG Pack dependency:

```
<dependency org="com.microej.pack.vg" name="vg-pack" rev="[VG Pack version]" conf="default->
↪default"/>
```

Note: The latest current pack version is 1.4.2.

The VG Pack will be automatically available after a VEE Port rebuild.

Use

See *MicroVG* chapter in Application Developer Guide.

6.15.3 Abstraction Layer API

Principle

The MicroVG implementation for MicroEJ requires an Abstraction Layer implementation. The Abstraction Layer implementation consists of a set of header files to implement in C to target the hardware drivers.

The VG Pack's embedded Front Panel extension implements all MicroVG features for the simulator.

Embedded VEE Port

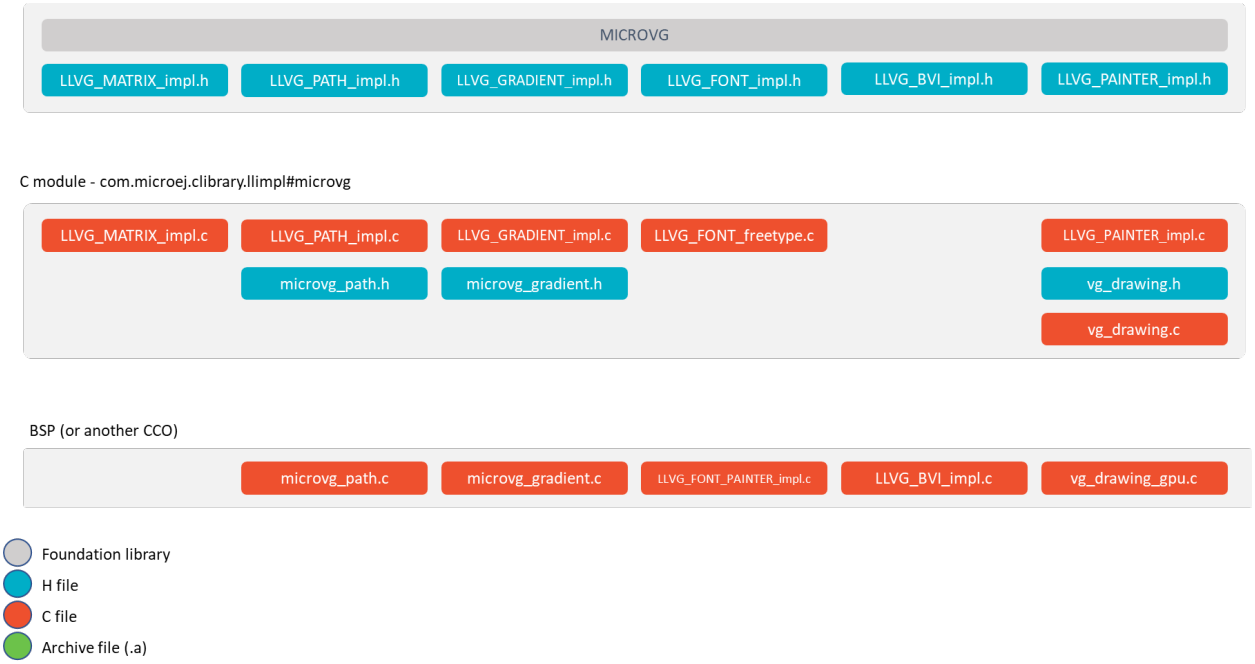


Fig. 60: MicroVG Embedded Abstraction Layer API

The specification of header files names is:

- Name starts with `LLVG_`.
- Second part's name refers to the VG engine: `MATRIX`, `PATH`, `GRADIENT`, `BVI` (image), `FONT`.
- All file's name ends with `_impl`: all functions must be implemented over hardware or in software.

A *master* header file initializes the native Vector Graphics engine: see *LLVG: VectorGraphics*. All other header files and their aims are described in next VG engines chapters: *Matrix*, *Path*, *Gradient*, *Image* and *Font*.

Simulator

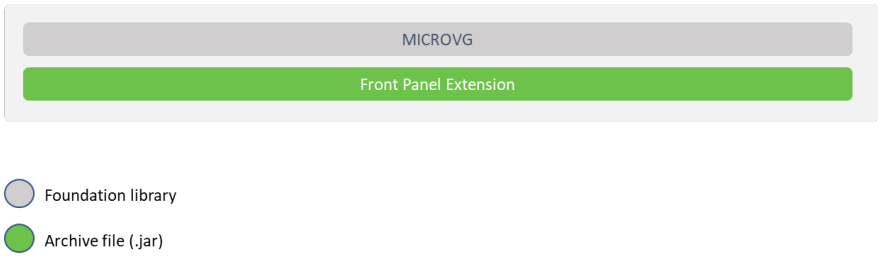


Fig. 61: MicroVG Simulator Abstraction Layer API

The Simulator's five VG engines are grouped in a *Front Panel extension*.

Note: The current implementation is built-in in the VG Pack and is only compatible with the i.MX RT595 MCU (see *VG Pack note*).

6.15.4 Matrix

Principle

The Matrix module contains the C part of the MicroVG implementation, which manages arithmetics matrices. This module is composed of only one element: an implementation of Abstraction Layer APIs to create and manipulate the matrices.

Functional Description

The Matrix module implements the framework of the MicroVG Matrix. It provides Abstraction Layer APIs that manipulate the matrices: fill an identity matrix, do a translation, a rotation, or a scaling and concatenate two matrices.

A matrix is a 3x3 matrix, and its elements are encoded in `float` (32-bit values):

- matrix_memory[0] = matrix[0][0];
- matrix_memory[1] = matrix[0][1];
- matrix_memory[2] = matrix[0][2];
- matrix_memory[3] = matrix[1][0];
- matrix_memory[4] = matrix[1][1];
- matrix_memory[5] = matrix[1][2];
- matrix_memory[6] = matrix[2][0];
- matrix_memory[7] = matrix[2][1];
- matrix_memory[8] = matrix[2][2];

The buffer where the matrix is encoded is stored in the Java heap.

Abstraction Layer API

The Abstraction Layer APIs that have to be implemented are listed in the header file `LLVG_MATRIX_impl.h` (see *LLVG_MATRIX: Matrix*):

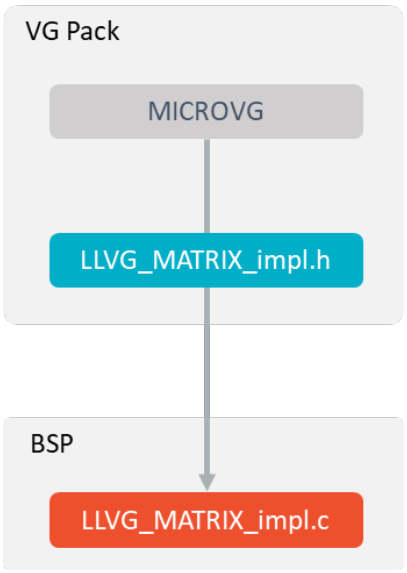


Fig. 62: Matrix Abstraction Layer API

- MicroVG library calls the BSP functions through the header file `LLVG_MATRIX_impl.h`.
- The *C module MicroVG* provides a default implementation of this header file: `LLVG_MATRIX_impl.c`.
- This file is automatically copied in the BSP project when fetching the C module during the VEE Port build.

Use

The MicroVG Matrix APIs are available in the class `ej.microvg.Matrix`.

6.15.5 Path

Principle

The Path module contains the C part of the MicroVG implementation, which manages vector paths. This module is composed of two elements:

- an implementation of Abstraction Layer APIs to create path elements compatible with the hardware,
- an implementation of Abstraction Layer APIs for MicroVG drawings.

Functional Description

The Path module implements the framework of the MicroVG Path. It provides Abstraction Layer APIs that create and merge some paths in a VEE Port-specific format. After the path creation and encoding, the path data should not change when the application draws it: the encoded format should be used by the VEE Port-specific implementation (generally GPU).

A path is a succession of commands. The command encoding is implementation specific; however, the `float` format is recommended.

List of commands:

- `LLVG_PATH_CMD_CLOSE` : MicroVG “CLOSE” command.

- `LLVG_PATH_CMD_MOVE` : MicroVG “MOVE ABS” command.
- `LLVG_PATH_CMD_MOVE_REL` : MicroVG “MOVE REL” command.
- `LLVG_PATH_CMD_LINE` : MicroVG “LINE ABS” command.
- `LLVG_PATH_CMD_LINE_REL` : MicroVG “LINE REL” command.
- `LLVG_PATH_CMD_QUAD` : MicroVG “QUAD ABS” command.
- `LLVG_PATH_CMD_QUAD_REL` : MicroVG “QUAD REL” command.
- `LLVG_PATH_CMD_CUBIC` : MicroVG “CUBIC ABS” command.
- `LLVG_PATH_CMD_CUBIC_REL` : MicroVG “CUBIC REL” command.

The buffer where the commands are encoded is stored in the Java heap. The buffer size is automatically increased by the MicroVG implementation when no more commands can be added.

A path is drawn with a color or with a *linear gradient*.

Abstraction Layer API

There are two separate Abstraction Layer API header files (see *LLVG_PATH: Vector Path*):

- `LLVG_PATH_impl.h` specifies the Abstraction Layer APIs used to create and encode the path.
- `LLVG_PAINTER_impl.h` lists the Abstraction Layer APIs called by VectorGraphicsPainter to draw the path.

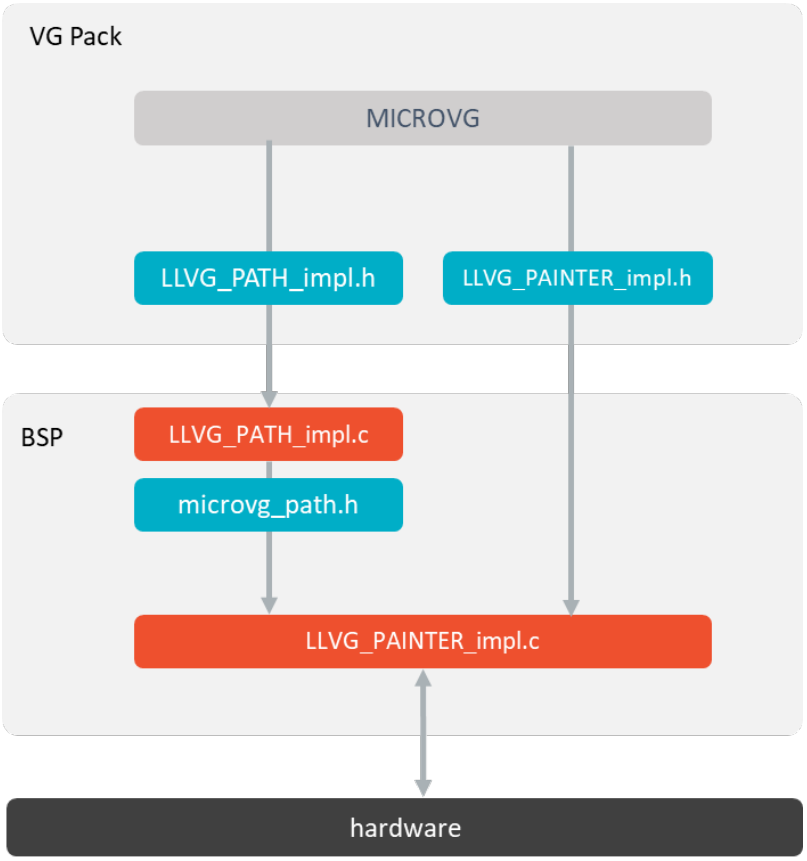


Fig. 63: Path Abstraction Layer API

- MicroVG library calls the BSP functions through the header files `LLVG_PATH_impl.h` and `LLVG_PAINTER_impl.h`.
- The *C module MicroVG* provides a default implementation of `LLVG_PATH_impl.h`: it manages the path buffer creation and filling, then redirect the command encoding to `microvg_path.h`.
- This C module also provides an implementation of `LLVG_PAINTER_impl.c` that synchronizes the drawing with the MicroUI Graphics Engine and redirects the drawing itself to a third-party drawer.
- A C module dedicated to a GPU provides an implementation of this drawer and `microvg_path.h`: it encodes the path commands and implements the drawings over the GPU library.
- The drawer also manages the *Gradient*.
- These files are automatically copied in the BSP project when fetching the C modules during the VEE Port build.

Use

The MicroVG Path APIs are available in the class `ej.microvg.Path`.

6.15.6 Gradient

Principle

The Gradient module contains the C part of the MicroVG implementation, which manages linear gradients. This module is composed of only one element: an implementation of the Abstraction Layer APIs to create gradient elements compatible with the hardware.

Functional Description

The Gradient module implements the framework of the MicroVG LinearGradient. It provides Abstraction Layer APIs that consist in creating a linear gradient in a VEE Port-specific format. After the gradient creation and encoding, the gradient data should not change when the application draws it: the encoded format should be used by the VEE Port-specific implementation (generally GPU).

A linear gradient is a succession of colors at different positions. The colors from the MicroVG library implementation are encoded in the 32-bit format: ARGB8888. The color encoding in the gradient is a VEE Port-specific implementation.

The buffer where the gradient is encoded is stored in the Java heap. The MicroVG implementation on demand automatically increases the buffer size.

Abstraction Layer API

There are two separate Abstraction Layer API header files (see *LLVG_GRADIENT: Vector Linear Gradient*):

- `LLVG_GRADIENT_impl.h` specifies the Abstraction Layer APIs used to create and encode the gradient.
- `LLVG_PAINTER_impl.h` lists the Abstraction Layer APIs called by VectorGraphicsPainter to draw the path.

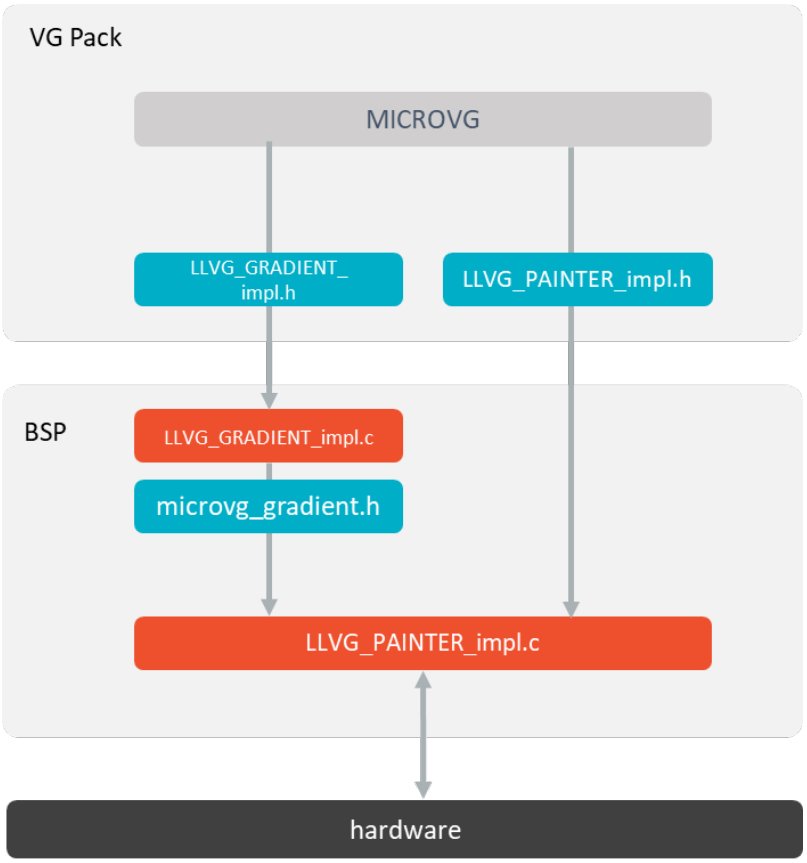


Fig. 64: Gradient Abstraction Layer API

- MicroVG library calls the BSP functions through the header files `LLVG_GRADIENT_impl.h` and `LLVG_PAINTER_impl.h`.
- The *C module MicroVG* provides a default implementation of `LLVG_GRADIENT_impl.h`: it manages the gradient buffer creation and filling, then redirect the gradient encoding to `microvg_gradient.h`.
- This C module also provides an implementation of `LLVG_PAINTER_impl.c` that synchronizes the drawing with the MicroUI Graphics Engine and redirects the drawing itself to a third-party drawer.
- A C module dedicated to a GPU provides an implementation of `LLVG_PAINTER_impl.h` and `microvg_gradient.h`: it encodes the gradient and implements the drawings over the GPU library.
- These files are automatically copied in the BSP project when fetching the C modules during the VEE Port build.

Use

The MicroVG Gradient APIs are available in the class `ej.microvg. LinearGradient`.

6.15.7 Image

Principle

The Image module contains the part of the MicroVG implementation which manages vectorial images. This module is composed of several elements:

- an *offline tool* that converts standard vector images in a binary format compatible with the *Rendering Engine*,
- an implementation of Abstraction Layer APIs to manipulate image files,
- an implementation of Abstraction Layer APIs for MicroVG drawings.

Compile-time Image

The Image module implements the MicroVG VectorImage framework. It provides an offline tool that consists in opening and decoding an image file and some Abstraction Layer APIs that manipulate the image at runtime.

A compile-time image file:

- is either an AVD (Android Vector Drawable) or a Scalable Vector Graphics (SVG),
- is identified by the resource name,
- is encoded in a binary format compatible with the *image renderer*,
- can be stored as an internal resource or an external one (see *External Memory*),
- is an immutable image: the application cannot draw into it.

Image Generator

The offline tool is an extension of the MicroUI *Image Generator*. This tool is automatically installed during the VEE Port build.

The tool converts :

- The Android Vector Drawable (AVD): this kind of image can hold linear gradients, animations on colors, opacity, path transformations, etc.
- The Scalable Vector Graphics (SVG): this kind of image is partially supported: linear gradients but no animations. It is advised to convert the SVG files into AVD files before using the Image Converter tool.

The tool generates a binary (RAW) file compatible with the *Rendering Engine*. The RAW file consists in a series of vector paths and animations.

To list the images to convert, the tool uses the application list files whose extension is `.vectorimage.list`. The generator provides an option to encode the path data (the path's points): it can be stored on signed 8, 16, 32-bit words or in `float` format. Respectively, the options are `VG8`, `VG16`, `VG32` and `VGF`.

This is an example of a `vectorimage.list` file:

```
# Convert an AVD in float format
/avd_image_1.xml:VGF
# Convert an AVD in signed 16-bit format
/path/to/avd_image_2.xml:VG16
# Convert an SVG in signed 8-bit format
/svg_image.svg:VG8
```

MicroVG Library

To load this kind of image, the application has to call VectorImage.getImage(). This API takes the image relative path: `/avd_image_1.xml` or `/path/to/avd_image_2.xml` or `/svg_image.svg` .

The implementation uses the Abstraction Layer API to retrieve the image. No data is stored in the Java heap (except the VectorImage object's instance).

Resource Vector Image

The Image module implements the MicroVG ResourceVectorImage framework.

Filtered Image

MicroVG VectorImage.filterImage() API allows to transform an image using a 4x5 color matrix. The result of the image transformation is stored in the MicroUI *Images Heap*. MicroVG ports for dedicated GPU (Low Level implementation) are responsible of the deallocation of this generated image. An implementation is available for *MicroVG Over VGLite*.

External Memory

Principle

MicroVG provides the API ResourceVectorImage.loadImage(). This is an extension of the compile-time images (the concepts are exactly the same), but it allows a load of a RAW image stored in an external memory that is not byte-addressable.

An external image loaded from byte-addressable memory is processed the same way than any compile-time image. For an image loaded from an external memory which is not byte-addressable, its data must be copied into byte-addressable memory before the image can be used for drawings. By default (see *C Modules*), the image data is copied into MicroUI *Images Heap*. The implementation is responsible for the image's lifecycle: allocation and release (already implemented in the *C Modules*).

Configuration File

Like compile-time images, the *Image Generator* uses a list file whose extension is `.externvectorimages.list`. The rules are exactly the sames than the compile-time images.

Process

The process to open a Vector Image from an external memory is exactly the same than the loading of *an external MicroUI Image*.

The following steps describe how to setup the loading of an external resource from the application:

1. Add the image to the application project resources (typically in the source folder `src/main/resources` and in the package `images`).
2. Create / open the configuration file (e.g. `application.externvectorimages.list`).
3. Add the relative path of the image and its output format (e.g. `/images/myImage.avd:VGF` see *Image Generator*).
4. Build the application: the Image Generator converts the image in RAW format in the external resources folder (`[application_output_folder]/externalResources`).
5. Deploy the external resources to the external memory (SDCard, flash, etc.) of the device.
6. (optional) Configure the *External Resources Loader* to load from this source.
7. Build the application and run it on the device.
8. The application loads the external resource using ResourceVectorImage.loadImage().
9. The image loader looks for the image and copies it in the *images heap* (no copy if the external memory is byte-addressable).
10. The external resource is immediately closed: the image's bytes have been copied in the images heap, or the image's bytes are always available (byte-addressable memory).
11. The application can use the image.
12. The application closes the image: the image is removed from the image heap.

Simulation

The Simulator automatically manages the external resources like internal resources. All images listed in `*.externvectorimages.list` files are copied in the external resources folder, and this folder is added to the Simulator's classpath.

Buffered Vector Image

This image is a ResourceVectorImage that the application can draw into. More specifically, the drawings are not *performed* but *stored*.

The concept consists in storing the compatible MicroUI drawings[1] and all MicroVG drawings into a command list. The application can then play this list of commands applying (or not) a global transformation.

[1] The compatible MicroUI drawings depend on the GPU Port; see:ref:*section_vg_cco*.

Note: The implementation uses the concept of MicroUI *custom* format (the custom `Format.CUSTOM_7`).

The way to register the drawing commands is strongly linked to the targeted GPU:

- The paths and gradients are stored to be used directly by the GPU to render the image (prevent runtime modifications before the image rendering).
- Depending on the GPU capabilities (a GPU may be able to draw a MicroUI anti-aliased line but not an aliased line), some MicroUI drawing API may be implemented (see *Buffered Image*).

As a consequence, the implementation is dedicated to the GPU. The *C Modules* provide some implementations, and the Front Panel (for the Simulation) features the same limitations as the embedded side (it is not possible to store a MicroUI drawing in the simulator if the embedded side is not able to perform it).

Runtime Image

The third-party library *VectorImageLoader* features an API to load an Android Vector Drawable (AVD) at runtime. This API creates a ResourceVectorImage

This library uses a simple XML parser (for performance and footprint convenience) that limits compatibility with the AVD specification. For instance, this loader does not manage the animations.

The *Vector Image Generator* can generate a compatible AVD file in the `.vectorimage.list`, using `AVD` as output format.

```
# Convert an AVD into a compatible AVD format
/avd_image.xml:AVD
# Convert an SVG into a compatible AVD format
/svg_image.svg:AVD
```

Rendering Engine

The Vector Image Rendering Engine has the responsibility of drawing the vector images. The destination is the display buffer, a MicroUI *BufferedImage* or a MicroVG BufferedVectorImage.

Three transformations can be applied when drawing a vector image:

- a global path transformation (3x3 matrix)
- a color transformation (4x5 color matrix)
- an opacity (value between 0 and 255)

The *C Modules* and the Front Panel already implement this engine.

Abstraction Layer API

There are two separate Abstraction Layer API header files:

- `LLVG_BVI_impl.h` specifies the Abstraction Layer APIs used to open and manage the BufferedVectorImage cycle-life.
- `LLVG_PAINTER_impl.h` lists the Abstraction Layer APIs called by VectorGraphicsPainter to draw an image (compile-time, runtime, or buffered vector image).

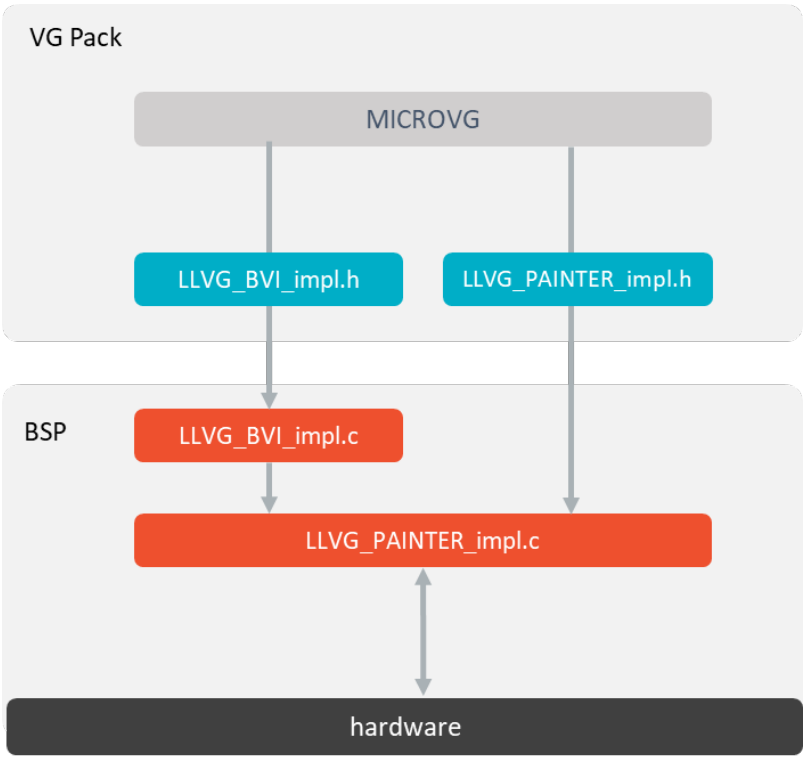


Fig. 65: Image Abstraction Layer API

- MicroVG library calls the BSP functions through the header files `LLVG_BVI_impl.h` and `LLVG_PAINTER_impl.h`.
- A C module dedicated to a GPU provides an implementation of `LLVG_BVI_impl.h` and `LLVG_PATH_PAINTER_impl.h`: the implementation is specific to the target (the GPU): the format of the RAW paths, gradients, and animations are GPU compliant.
- These files are automatically copied in the BSP project when fetching the C modules during the VEE Port build.

Simulation

The implementation of the MicroVG library is included in the VG Pack. No specific support is required to retrieve and use the images.

Use

The MicroVG Font APIs are available in the class `ej.microvg.` VectorImage.

6.15.8 Font

Principle

The Font module contains the C part of the MicroVG implementation, which manages vectorial fonts. This module is composed of two elements:

- an implementation of Abstraction Layer APIs to manipulate font files,
- an implementation of Abstraction Layer APIs for MicroVG drawings.

Functional Description

The Font module implements the MicroVG VectorFont framework. It provides Abstraction Layer APIs that consist of opening and decoding a font file and getting the font's characteristics.

A font file:

- is either a TTF or an OTF,
- is identified by the resource name,
- can be stored as internal resource or external (see *External Fonts*).

No data is stored in the Java heap. The implementation is responsible for the font's cycle life: allocation and release.

A font is used to draw a string with a color or with a *linear gradient*.

Abstraction Layer API

There are two separate Abstraction Layer API header files (see *LLVG_FONT: Vector Font*):

- `LLVG_FONT_impl.h` specifies the Abstraction Layer APIs used to open and retrieve the font's characteristics.
- `LLVG_PAINTER_impl.h` lists the Abstraction Layer APIs called by VectorGraphicsPainter to draw a string with the font.

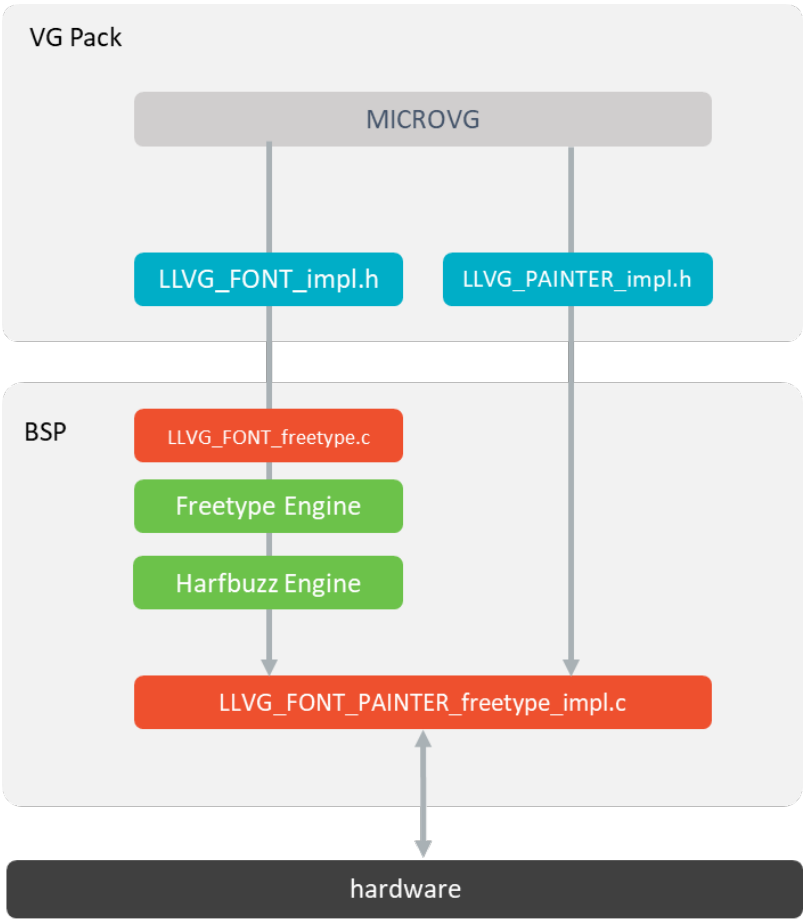


Fig. 66: Font Abstraction Layer API

- MicroVG library calls the BSP functions through the header files `LLVG_FONT_impl.h` and `LLVG_PAINTER_impl.h`.
- The *C module MicroVG* provides a default implementation of `LLVG_FONT_impl.h` over FreeType.
- This C module also provides an implementation of `LLVG_PAINTER_impl.c` that synchronizes the drawing with the MicroUI Graphics Engine and redirects the drawing itself to a third-party drawer.
- A C module dedicated to a GPU provides an implementation of this drawer. It also redirects the *complex layout* to a third party C module.
- The drawer also manages the *Gradient*.
- The *C module Harfbuzz* provides an implementation of *complex layout*.
- These files are automatically copied in the BSP project when fetching the C modules during the VEE Port build.

External Memory

Principle

MicroVG does not provide some Low Level API to make the distinction between a font loaded from different kind of memories (internal or external, byte-addressable or not). The Low Level implementation (*C Modules MicroVG and FreeType*) features the font management from an external memory which is not byte-addressable when the VEE Port provides an implementation of the *External Resources Loader*.

Configuration File

A Vector Font file is a *simple* resource. To specify this resource as an external resource, the font file path must be listed in a `.externresources.list` file in addition with the `.resources.list` file (see *Application Resources*).

Process

The following steps describe how to setup the loading of an external resource from the application:

1. Add the font to the application project resources (typically in the source folder `src/main/resources` and in the package `fonts`).
2. Create / open the configuration files (e.g. `application.resources.list` and `application.externresources.list`).
3. In both files, add the relative path of the font (e.g. `/fonts/myFont.ttf`).
4. Build the application: the processed external resources are copied into the external resources folder (`[application_output_folder]/externalResources`).
5. Deploy the external resources to the external memory (SDCard, flash, etc.) of the device.
6. (optional) Configure the *External Resources Loader* to load from this source.
7. Build the application and run it on the device.
8. The application loads the external resource using ej.microvg.VectorFont.loadFont().
9. FreeType (*C Modules*) recognizes this resource as external resource; it configures itself to manage this resource differently than an internal resource (see *Library: FreeType* to have more details).
10. The application can use the font.

Simulation

The Simulator automatically manages the external resources like internal resources. All images listed in `*.externresources.list` files are copied in the external resources folder, and this folder is added to the Simulator's classpath.

Use

The MicroVG Font APIs are available in the class `ej.microvg. VectorFont`.

6.15.9 C Modules

Principle

Several C modules implement the VG Pack's Abstraction Layer APIs. Some are generic, and some are VEE Port dependent (more precisely: GPU dependent). The generic modules provide header files to be extended by the specific modules. The generic C modules are available on the *Central Repository* and the specific C modules on the *Developer Repository*.

The following picture illustrates the available C modules and their relations for an implementation that uses:

- FreeType library for the font renderer and the font layouter in simple layout mode.
- Harfbuzz library for the font layouter in complex layout mode.
- Vivante VGLite library for the drawing of vector paths

The following chapters explain the aim and relations of each C module.

Note: It is a simplified view: all sources and headers files of each C module are not visible.

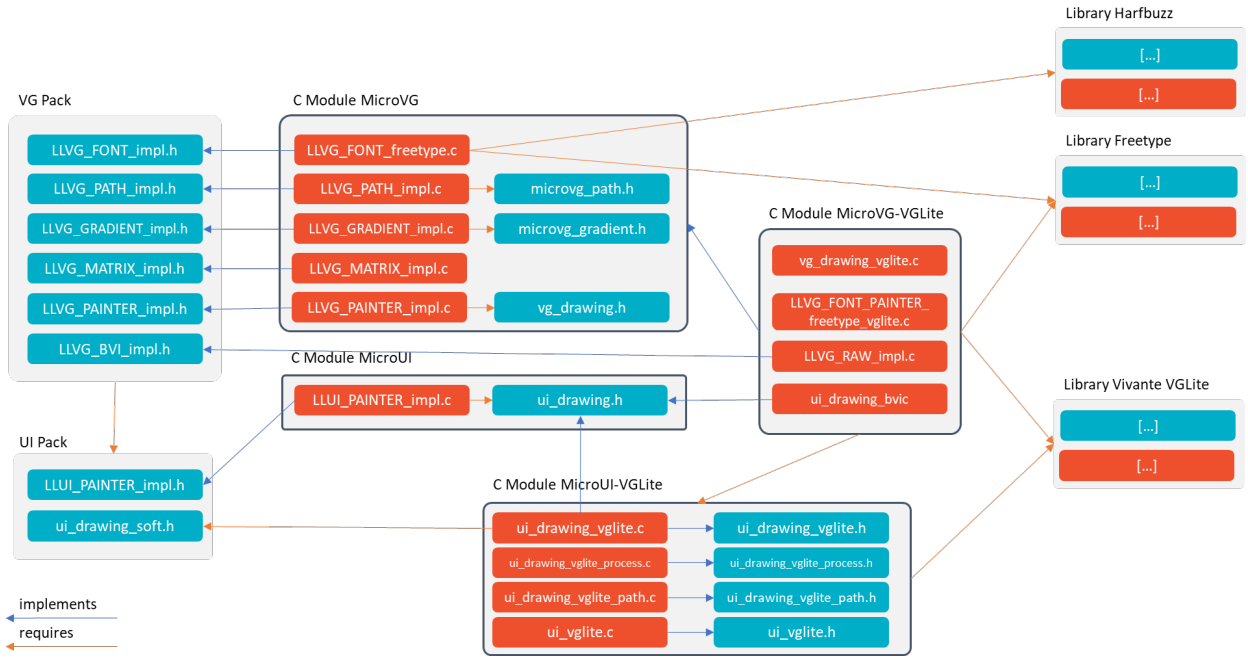


Fig. 67: MicroVG C Modules

UI Pack & MicroUI C Modules

The UI Pack provides a header file to implement the MicroUI drawings: `LLUI_PAINTER_impl.h`. See *C Modules* chapter to have more information.

Library: Vivante VGLite

This library is the official Vivante VGLite library. The C modules use its header files to target the GPU.

Note: The library must be patched to be compatible with the C module "MicroUI over VGLite". Consult the C module's ReadMe file for more information.

VG Pack

The VG Pack provides a set of header files to implement the MicroVG concepts. The header files are described in the dedicated chapters: *Matrix module*, *Path module*, *Gradient module*, *Image module* and *Font module*.

The VG Pack is an extension of the UI Pack. The VG Pack's header files require the UI Pack's header files to manipulate the MicroUI concepts. Consequently, the VG Pack must be installed on a VEE Port that fetches a UI Pack.

The VG Pack and its header files are available on the *Central Repository*: com.microej.pack.vg#vg-pack.

C Module: MicroVG

Description

This generic C module provides an implementation of MicroVG concepts: matrix, path, linear gradient and font; respectively `LLVG_MATRIX_impl.c`, `LLVG_PATH_impl.c`, `LLVG_GRADIENT_impl.c` and `LLVG_FONT_freetype.c`.

- Matrix (see Matrix module's *Abstraction Layer API*): a basic software implementation.
- Path (see Path module's *Abstraction Layer API*): a generic implementation that manages the command buffer's life cycle and dispatches the command encoding to a 3rd-party header file `microvg_path.h`.
- Gradient (see Gradient module's *Abstraction Layer API*): a generic implementation that manages the gradient buffer's life cycle and dispatches the gradient encoding to a 3rd-party header file `microvg_gradient.h`.
- Font (see Font module's *Abstraction Layer API*): an implementation of vector font over FreeType: open font file and retrieve font's characteristics.
- The MicroVG painter native functions are implemented in `LLVG_PAINTER_impl.c` and the drawings are redirected to `vg_drawing.h`.
- Image management is too specific to the GPU and is not implemented in this C module.

This C module is available on the *Central Repository*: com.microej.clibrary.llimpl#microvg.

Dependencies

This generic C module requires some specific modules:

- Path and Gradient require a C module specific to a VEE Port (to a GPU format).
- Font requires the FreeType library and optionally the Harfbuzz library to manage the *complex layout*.

Configuration

This C module uses a configuration file. This file (a header file with some C defines) enables (or disables) and configures some options:

- `VG_FEATURE_PATH` : set this define to embed the full implementation of `Path` feature. Otherwise, a stub implementation is used, and all `Path` drawings are dropped.
- `VG_FEATURE_GRADIENT` : configure this define to embed the full implementation of `LinearGradient` or a stub implementation that only manages one color (linear gradient's first color). The respective options are `VG_FEATURE_GRADIENT_FULL` and `VG_FEATURE_GRADIENT_FIRST_COLOR` .
- `VG_FEATURE_FONT` : configure this define to specify the Font Engine and the Font Engine's backend. Two options are currently available: the FreeType engine with a vectorial backend and the FreeType engine with a bitmap backend. The respective options are `VG_FEATURE_FONT_FREETYPE_VECTOR` and `VG_FEATURE_FONT_FREETYPE_BITMAP` .
- `VG_FEATURE_FREETYPE_TTF` : set this define to enable the support of TTF font files in FreeType.
- `VG_FEATURE_FREETYPE_OTF` : set this define to enable the support of OTF font files in FreeType.
- `VG_FEATURE_FREETYPE_COLORED_EMOJI` : set this define to enable the support of colored emoji in FreeType.
- `VG_FEATURE_FONT_COMPLEX_LAYOUT` : set this define to enable the support of *complex layout*. This option is managed by the C module `Harfbuzz` (see upper).
- `VG_FEATURE_FONT_EXTERNAL` : set this define to allow loading of external font files (outside the application classpath). See *External Fonts*.
- `VG_FEATURE_FREETYPE_HEAP_SIZE` : specify the FreeType engine's heap size.
- `VG_FEATURE_FONT_COMPLEX_LAYOUT_HEAP_SIZE` : specify the Harfbuzz engine's heap size.

Note: This options list is not exhaustive. Please consult the C module's readme file for more information.

Library: FreeType

Description

The FreeType library compatible with MicroEJ is packaged in a C module on the *Developer Repository*: com.microej.clibrary.thirdparty#freetype.

This C module provides a fork of FreeType 2.11.0.

Memory Heap Configuration

The FreeType library requires a memory Heap for FreeType internal objects allocated when a font file is loaded (see https://freetype.org/freetype2/docs/design/design-4.html). The size of this heap depends on the number of fonts loaded in parallel and on the fonts themselves. This size is defined by `VG_FEATURE_FREETYPE_HEAP_SIZE` in `microvg_configuration.h`.

All fonts do not require the same heap size. FreeType heap usage can be monitored using the following configurations:

- `MICROVG_MONITOR_HEAP` defined in `microvg_helper.h`
- `MEJ_LOG_MICROVG` and `MEJ_LOG_INFO_LEVEL` defined in `mej_log.h`

Principle

1. The Application loads a font with ej.microvg.VectorFont.loadFont().
   - If the resource is internal or external from byte-addressable memory, the FreeType library is configured to read directly from that resource memory section.
   - Else, if the resource is external from non-byte-addressable memory, the FreeType library is configured to use the *external loader* to read from that memory.
   - At this point, the font resources are allocated and the font generic data (including baseline & height metrics) is loaded on the FreeType dedicated heap.
2. The Application requests metrics.
   - For generic metrics, already loaded data is directly used (and scaled to the font size used).
   - For text-dependent metrics: computed by loading metrics of every glyph required by the input string (the glyphs bitmaps are not actually rendered here).
3. The Application requests drawings.
   - For every character to draw:
     - the associated glyph is loaded,
     - the bitmap is rendered for the given font size and
     - the character is drawn in the given graphic context.
4. The Application unloads the font with ej.microvg.VectorFont.close().
   - Any resource associated with the font is released.
   - At this point, any attempt to use the font will result in an exception.

Library: Harfbuzz

The library Harfbuzz compatible with MicroEJ is packaged in a C module on the *Developer Repository*: com.microej.clibrary.thirdparty#harfbuzz.

This C module provides a fork of Harfbuzz 4.2.1.

The Harfbuzz library requires a memory Heap for Harfbuzz internal objects allocated when a font file is loaded. The size of this heap depends on the number of fonts loaded in parallel and on the fonts themselves. This size is defined by `VG_FEATURE_HARFBUZZ_HEAP_SIZE_HEAP` in `microvg_configuration.h`.

All fonts do not require the same heap size. The `MICROVG_MONITOR_HEAP` define in `microvg_helper.h` and `MEJ_LOG_MICROVG` and `MEJ_LOG_INFO_LEVEL` defines in `mej_log.h` can be used to monitor the Harfbuzz heap evolution.

FreeType and Harfbuzz libraries are not sharing the same heap, but this could easilly be done by updating `ft_system.c` and `hb-alloc.c` files.

C Module: MicroVG Over VGLite

This C module is a specific implementation of the VG Pack drawings over the official Vivante VGLite library (that targets some GPU with vector graphics acceleration):

- It implements the MicroVG API `vg_drawing.h` in `vg_drawing_vglite.c` and `LLVG_PAINTER_FONT_freetype_vglite.c`.
- It implements the MicroVG Image management (draw a compile-time image, create a BufferedVectorImage, etc.): `LLVG_RAW_impl.c`.
- It provides an implementation of MicroVG drawings to the MicroVG BufferedVectorImage: `vg_drawing_bvi.c`.
- It also implements MicroUI drawings to the MicroVG BufferedVectorImage: `ui_drawing_bvi.c`.

The implementation requires:

- the concepts of the C module MicroVG,
- the concepts of the C module MicroUI over VGLite,
- the FreeType library,
- the Vivante VGLite library.

This C module is available on the *Developer Repository*: com.microej.clibrary.llimpl#microvg-vglite.

Compatibility

The compatibility between the components (Packs, C modules, and Libraries) is described in the *Release Notes*.

6.15.10 Simulation

Principle

The VG Pack embeds an extension of *UI Pack's Front Panel mock* to implement the equivalent of the five embedded modules (Matrix, Path, Gradient, Image and Font).

The implementation simulates the same characteristics and limitations as the embedded modules.

Installation

No action is required in the VEE Port's Front Panel project: the MicroVG simulation part is automatically used when an application uses MicroVG APIs on the simulator.

Use

Launch a MicroVG application on the Simulator to run the Front Panel extension.

6.15.11 Release Notes

UI Pack Compatibility Version

The current VG Pack version is 1.4.2. The following table describes the compatibility ranges between VG and UI Packs.

VG Pack Range	UI Pack Range	Comment
[1.3.0-1.4.2]	[13.5.0-14.0.0[	BufferedImage with custom format
[1.1.0-1.2.1]	[13.3.0-14.0.0[	Internal feature
[1.0.0-1.0.1]	[13.2.0-14.0.0[	

Foundation Libraries

The following table describes Foundation Libraries API versions implemented in MicroEJ VG Packs.

Table 27: MicroVG API Implementation

VG Pack Range	MicroVG
[1.4.0-1.4.2]	1.4.0
1.3.0	1.3.0
1.2.1	1.2.0
1.1.0	1.1.0
[1.0.0-1.0.1]	1.0.0

C Modules Compatibility Version

The C modules are described *here*.

Several generic C modules are available for a given version of the VG Pack. In addition to generic C modules, the specific implementation of the VG Pack over Vivante VGLite depends on:

- the UI Pack (see upper),
- the UI Pack C module: see *UI Pack*,
- and by consequence, the specific C module MicroUI over VGLite: see *C Module: MicroUI Over VGLite*.

The following table describes the compatibility ranges between the VG Packs and the C modules (generic and specific):

VG Pack	MicroVG	FreeType	Harfbuzz	MicroUI-VGLite	MicroVG-VGLite
1.4.2	4.0.0	2.0.2	1.0.2	7.2.0	6.1.1
[1.4.0-1.4.1]	3.0.1	2.0.2	1.0.2	[7.0.0-7.1.0]	[6.0.0-6.1.0]
1.3.0	3.0.0	2.0.2	1.0.2	6.0.1	5.0.1
1.2.1	2.1.0	2.0.2	1.0.2	5.0.1	4.0.3
1.1.0	2.0.0	2.0.2	1.0.2	3.0.0	3.0.2
[1.0.0-1.1.0[	n/a	n/a	n/a	n/a	n/a

Note: The C module `MicroVG over VGLite` fetches automatically by transitivity the other C modules. No need to fetch explicitly the different modules (except the C module `Harfbuzz`). An update of this C module also updates (if necessary) the other C modules.

6.15.12 Changelog

[1.4.2] - 2023-11-13

MicroVG Implementation

Added

- Add some traces when debugging the SNI resources.

Fixed

- Fix dynamic paths larger than 64 KB.

Front Panel

Fixed

- Fix dynamic paths larger than 64 KB.

[1.4.1] - 2023-09-21

Fixed

- Fix the path command “move relative”.

[1.4.0] - 2023-07-21

Fixed

- Fix the UI Pack minimal compatible version (13.5.0).

MicroVG Implementation

Added

- Add SystemView event logs (feature available with C Module MicroVG 3.0.1).

Changed

- Compatible with MicroVG API 1.4.

Fixed

- Fix path bounds computation.

[1.3.0] - 2023-05-10

- Compatible with UI Pack 13.5.0 or higher.

MicroVG Implementation

Changed

- Compatible with MicroVG API 1.3.

Front Panel

Fixed

- Simplify pixel data conversion after drawing.

[1.2.1] - 2023-02-06

Front Panel

Fixed

- Fix the cropped images when using GraphicsContext clip and translation.

[1.2.0] - 2022-12-30

- Compatible with UI Pack 13.3.0 or higher.

MicroVG Implementation

Changed

- Compatible with MicroVG API 1.2.
- Change the VectorImage internal format: *raw* format instead of *immutables* format.

Front Panel

Fixed

- Fix the redirection of fillEllipseArc to the right software algorithm.

Vector Image Converter

Added

- Add “fill alpha” animations to gradient elements.

[1.1.1] - 2022-09-05

- Compatible with UI Pack 13.3.0 or higher.

MicroVG Implementation

Changed

- Compatible with MicroVG API 1.1.
- Allow loading a VectorFont from external resources.
- Change color animation interpolation (match Android formula).

Fixed

- Fix NullPointerException while sorting TranslateXY VectorDrawableObjectAnimator in vectorimage-converter.

LLAPIs

Added

- Add LLAPI to close a font: `LLVG_FONT_IMPL_dispose()` .

Changed

- Manage the font *complex layout*.
- Returns an error code when drawing something.

[1.0.1] - 2022-05-16

- Compatible with UI Pack 13.2.0 or higher.

MicroVG Implementation

Fixed

- Fix incorrect transformation of animated paths while creating a filtered image.

[1.0.0] - 2022-05-13

- Initial release.
- Compatible with UI Pack 13.2.0 or higher.
- Compatible with MicroVG API 1.0.0.

6.15.13 Migration Guide

From 1.3.x to 1.4.x

BSP with VG-Lite

- Follow the migration steps of *C Module MicroUI-VGLite 7.1.0*.
- *[VEE Port configuration project]*
  - Fetch VG Pack 1.4.2, C Modules MicroVG 4.0.0 and MicroVG-VGLite 6.1.1.
  - Delete the content of `dropins/include` folder.
- *[BSP project]*
  - Delete the properties files `cco_microvg.properties` and `cco_microvg-vglite.properties`.
- Build the VEE Port, the FreeType library (in case of a dedicated project), and the BSP.

From 1.2.x to 1.3.x

BSP with VG-Lite

- Follow the migration steps of *C Module MicroUI-VGLite 6.0.1*.
- *[VEE Port configuration project]*
  - Fetch VG Pack 1.3.0, C Modules MicroVG 3.0.0 and MicroVG-VGLite 5.0.1.
  - Delete the content of `dropins/include` folder.
- *[BSP project]*
  - Delete the properties files `cco_microvg.properties` and `cco_microvg-vglite.properties`.
  - Delete the C files `freetype_bitmap_helper.h`, `freetype_bitmap_helper.c`, `LLVG_BVI_impl.c`, `LLVG_FONT_PAINTER_freetype_bitmap.c` and `LLVG_PATH_PAINTER_vglite.c` and remove them from the C project configuration.
  - In the C project configuration, include the new C files `ui_drawing_bvi.c`, `LLVG_BVI_stub.c`, `LLVG_PAINTER_impl.c`, `vg_drawing_bvi.c`, `vg_drawing_stub.c`, `vg_drawing_vglite.c` and `vg_drawing.c`.

- In the C project configuration, set the define `LLUI_GC_SUPPORTED_FORMATS=2` to enable the Buffered-VectorImage support.
- Verify the options in `microvg_configuration.h`.

- Build the VEE Port, the FreeType library (in case of a dedicated project), and the BSP.

6.16 Networking

6.16.1 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.

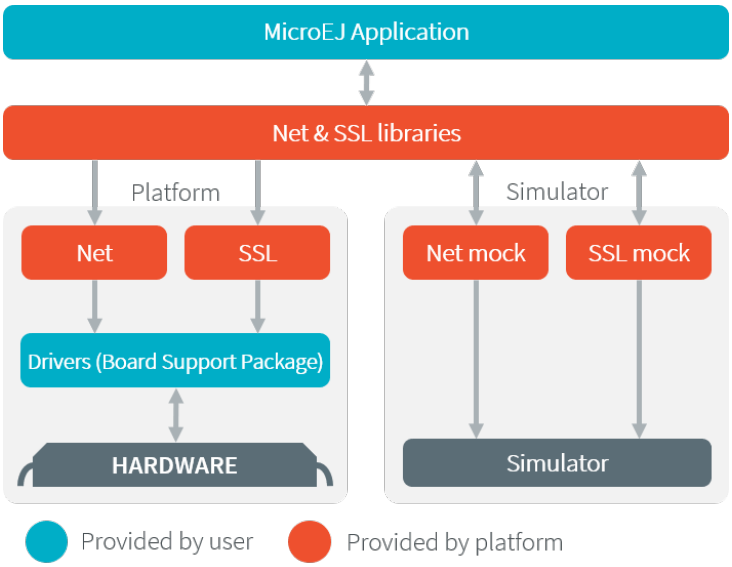


Fig. 68: Overview

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.

6.16.2 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

The Net Pack bundles several libraries: Net, SSL & Security.

Refer to the chapter *Pack Import* to integrate a specific version of the Net Pack:

```
<dependencies>
    <dependency org="com.microej.pack.net" name="net-pack" rev="11.0.2"/>
</dependencies>
```

Then, using the VEE Port Editor (see *Platform Module Configuration*), enable the Net library (API, Impl & Mock):

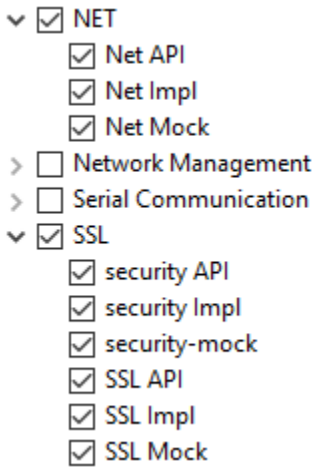


Fig. 69: Net Pack Modules

Use

The Net API Module must be added to the *module.ivy* of the MicroEJ Application project to use the Net library.

```
<dependency org="ej.api" name="net" rev="1.1.4"/>
```

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

6.16.3 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

The Net Pack bundles several libraries: Net, SSL & Security.

Refer to the chapter *Pack Import* to integrate a specific version of the Net Pack:

```
<dependencies>
    <dependency org="com.microej.pack.net" name="net-pack" rev="11.0.2"/>
</dependencies>
```

Then, using the VEE Port Editor (see *Platform Module Configuration*), enable the SSL library (API, Impl & Mock):

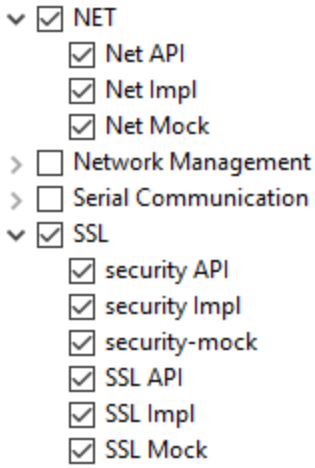


Fig. 70: Net Pack Modules

Use

The SSL API module must be added to the *module.ivy* of the MicroEJ Application project to use the SSL library.

```
<dependency org="ej.api" name="ssl" rev="2.2.3"/>
```

6.16.4 Network Interfaces Management

Overview

The Network Foundation Library provides a way to manage and configure TCP/IP network interfaces.

Dependencies

- Network core module (see *Network Core Engine*).
- `LLECOM_NETWORK_impl.h` implementation (see *LLECOM_NETWORK: Network Interfaces*).

Installation

The Network *Pack* module must be installed in your VEE Port.

In the Platform configuration project, (`-configuration` suffix), add the following dependency to the *module.ivy* file:

```
<dependency org="com.microej.pack.ecom-network" name="ecom-network-pack" rev="1.0.0" />
```

The Platform project must be rebuilt (*Platform Build*).

Use

The Network API Module must be added to the *module.ivy* file of the Application project:

```
<dependency org="ej.api" name="ecom-network" rev="2.1.1"/>
```

6.16.5 Wi-Fi

Overview

The Wi-Fi Foundation Library provides a way to manage and configure Wi-Fi access points.

Dependencies

- Network core module (see *Network Core Engine*).
- `LLECOM_WIFI_impl.h` implementation (see *LLECOM_WIFI: Wi-Fi Management*).

Installation

The Wi-Fi *Pack* module must be installed in your VEE Port.

In the Platform configuration project, (`-configuration` suffix), add the following dependency to the *module.ivy* file:

```
<dependency org="com.microej.pack.ecom-wifi" name="ecom-wifi-pack" rev="1.0.0" />
```

The Platform project must be rebuilt (*Platform Build*).

Use

The Wi-Fi API Module must be added to the *module.ivy* file of the Application project:

```
<dependency org="ej.api" name="ecom-wifi" rev="2.2.2"/>
```

6.17 Bluetooth

6.17.1 Principle

The Bluetooth Foundation Library defines a low-level Bluetooth framework for embedded devices. It allows you to manage abstract Bluetooth connections without worrying about the native underlying Bluetooth kind.

6.17.2 Functional Description

The MicroEJ Application manages Bluetooth elements using Adapter/Connection/Service/Characteristic/Descriptor/etc abstraction. The Bluetooth implementation made for each MicroEJ Platform is responsible for surfacing the native Bluetooth specific behavior.

6.17.3 Overview

The Bluetooth Foundation Library provides a way to manage and configure Bluetooth module.

6.17.4 Dependencies

- `LLBLUETOOTH_impl.h` implementation (see *LLBLUETOOTH: Bluetooth*).

6.17.5 Installation

The Bluetooth *Pack* module must be installed in your VEE Port.

In the Platform configuration project, (`-configuration` suffix), add the following dependency to the *module.ivy* file:

```
<dependency org="com.microej.pack.bluetooth" name="bluetooth-pack" rev="2.2.1" />
```

The Platform project must be rebuilt (*Platform Build*).

6.17.6 Use

The Bluetooth API Module must be added to the *module.ivy* file of the Application project:

```
<dependency org="ej.api" name="bluetooth" rev="2.1.1"/>
```

6.18 Event Queue

6.18.1 Principle

The Event Queue Foundation Library provides an asynchronous communication interface between the native world and the Java world based on events. Its functional architecture and usage are documented in the *Application Developer Guide*.

6.18.2 Dependencies

- `LLEVENT_impl.h` and `LLEVENT.h` implementations (see *LLEVENT: Event Queue*).

6.18.3 Installation

The Event Queue *Pack* module must be installed in your VEE Port.

In the VEE Port configuration project, add the following dependency to the *module.ivy* file:

```
<dependency org="com.microej.pack.event" name="event-pack" rev="2.0.1" transitive="false"/>
```

6.19 File System

6.19.1 Principle

The FS Foundation Library defines a low-level File System framework for embedded devices. It allows you to manage abstract files and directories without worrying about the native underlying File System kind.

6.19.2 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.

6.19.3 Dependencies

- `LLFS_impl.h` and `LLFS_File_impl.h` implementations (see *LLFS: File System*).

6.19.4 Installation

FS is an additional module. In the platform configuration file, check `FS` to install it. When checked, the properties file `fs/fs.properties` is required during platform creation in order to configure the module. This properties file specifies the characteristics of the File System used in the C project (case sensitivity, root directory, file separator, etc.).

The FS module defines two pre-configured File System types: `Unix` and `FatFS`. Some characteristics don't need to be specified for these File System types, but they can be overridden if needed. For example, specifying a `Unix` File System type will automatically set the file separator to `/`.

If none of the pre-configured File System types correspond to the File System used in the C project, the `Custom` type can be used. When this type is selected, all the File System characteristics must be specified in the properties file.

The list below describes the properties that can be defined in the file `fs/fs.properties`:

- `fs`: Defines the type of File System used in the C project (optional, the default value is `Unix`). This property can have one of the following values:
  - `Unix`: select this configuration when using a Unix-like File System (case-sensitive, file separator is `/`).
  - `FatFS`: select this configuration when using FatFS File System (case-insensitive, file separator is `/`).

  - `Custom` : select this configuration when using another type of File System.
- `root.dir` : Defines the File System root volume. This property is optional for `Unix` and `FatFS` (`/` by default for both).
- `user.dir` : Defines the File System user directory. This property is optional for `FatFS` (`/usr/` by default).
- `java.io.tmpdir` : Defines the File System temporary directory. This property is optional for `Unix` and `FatFS` (`/tmp/` by default for both).
- `file.separator` : Defines the File System file separator. This property is optional for `Unix` and `FatFS` (`/` by default for both).
- `path.separator` : Defines the File System path separator. This property is optional for `Unix` and `FatFS` (`:` by default for both).
- `case.sensitivity` : Defines the case sensitivity of the File System. This property is optional for `Unix` (`caseSensitive` by default) and `FatFS` (`caseInsensitive` by default). This property can have one of the following values:
  - `caseSensitive` : the File System is case-sensitive.
  - `caseInsensitive` : the File System is case-insensitive.

Properties File Template

The following snippet can be used as a template for `fs.properties` file:

```
# Defines the type of File System used in the C project.
# Possible values are:
#   - FatFs
#   - Unix
#   - Custom
# @optional, default value is "Unix"
#fs=

# Defines the File System root volume.
# @optional for the following File System types:
#   - FatFs (default value is "/")
#   - Unix (default value is "/")
# @mandatory for the following File System type:
#   - Custom
#root.dir=

# Defines the File System user directory.
# @optional for the following File System type:
#   - FatFs (default value is "/usr")
# @mandatory for the following File System types:
#   - Unix
#   - Custom
#user.dir=

# Defines the File System temporary directory.
# @optional for the following File System types:
#   - FatFs (default value is "/tmp")
#   - Unix (default value is "/tmp")
# @mandatory for the following File System type:
```

(continues on next page)

(continued from previous page)

```
#   - Custom
#java.io.tmpdir=

# Defines the File System file separator.
# @optional for the following File System types:
#   - FatFs (default value is "/")
#   - Unix (default value is "/")
# @mandatory for the following File System type:
#   - Custom
#file.separator=

# Defines the File System path separator.
# @optional for the following File System types:
#   - FatFs (default value is ":")
#   - Unix (default value is ":")
# @mandatory for the following File System type:
#   - Custom
#path.separator=

# Defines the case sensitivity of the File System.
# Valid values are "caseInsensitive" and "caseSensitive".
# @optional for the following File System types:
#   - FatFs (default value is "caseInsensitive")
#   - Unix (default value is "caseSensitive")
# @mandatory for the following File System type:
#   - Custom
#case.sensitivity=
```

6.19.5 Use

The FS API Module must be added to the *module.ivy* of the MicroEJ Application project to use the FS library.

```
<dependency org="ej.api" name="fs" rev="2.0.6"/>
```

6.20 Hardware Abstraction Layer

6.20.1 Principle

The Hardware Abstraction Layer (HAL) Foundation Library features API that target IO devices, such as GPIOs, analog to/from digital converters (ADC / DAC), etc. The API are very basic in order to be as similar as possible to the BSP drivers.

6.20.2 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.

6.20.3 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;
```

6.20.4 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.

6.20.5 Dependencies

- `LLHAL_impl.h` implementation (see *LLHAL: Hardware Abstraction Layer*).

6.20.6 Installation

HAL is an additional module. In the platform configuration file, check `HAL` to install the module.

6.20.7 Use

The HAL API Module must be added to the *module.ivy* of the MicroEJ Application project to use the HAL library.

```
<dependency org="ej.api" name="hal" rev="1.0.4"/>
```

6.21 Device Information

6.21.1 Principle

The Device Foundation Library provides access to the device information. This includes the architecture name and a unique identifier of the device for this architecture.

6.21.2 Dependencies

- `LLDEVICE_impl.h` implementation (see *LLDEVICE: Device Information*).

6.21.3 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.

6.21.4 Use

The Device API Module must be added to the *module.ivy* of the MicroEJ Application project to use the Device library.

```
<dependency org="ej.api" name="device" rev="1.0.2"/>
```

6.22 Security

6.22.1 Principle

The Security Foundation Library provides standard Java API (part of the Java Cryptography Architecture) for cryptographic operations: cipher, digest, MAC, signature, secure random & key/certificate management. It relies on a native crypto engine (such as Mbed TLS, OpenSSL or wolfSSL).

6.22.2 Dependencies

- The `LLSEC_*.h` implementations (see *LLSEC: Security*).

6.22.3 Installation

The Net Pack bundles several libraries: Net, SSL & Security.

Refer to the chapter *Pack Import* to integrate a specific version of the Net Pack:

```
<dependencies>
    <dependency org="com.microej.pack.net" name="net-pack" rev="11.0.2"/>
</dependencies>
```

Then, using the VEE Port Editor (see *Platform Module Configuration*), enable the Security library (API, Impl & Mock):

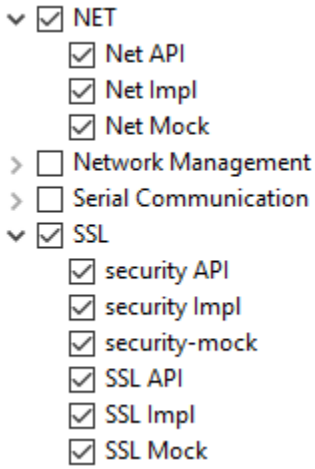


Fig. 71: Net Pack Modules

6.22.4 Use

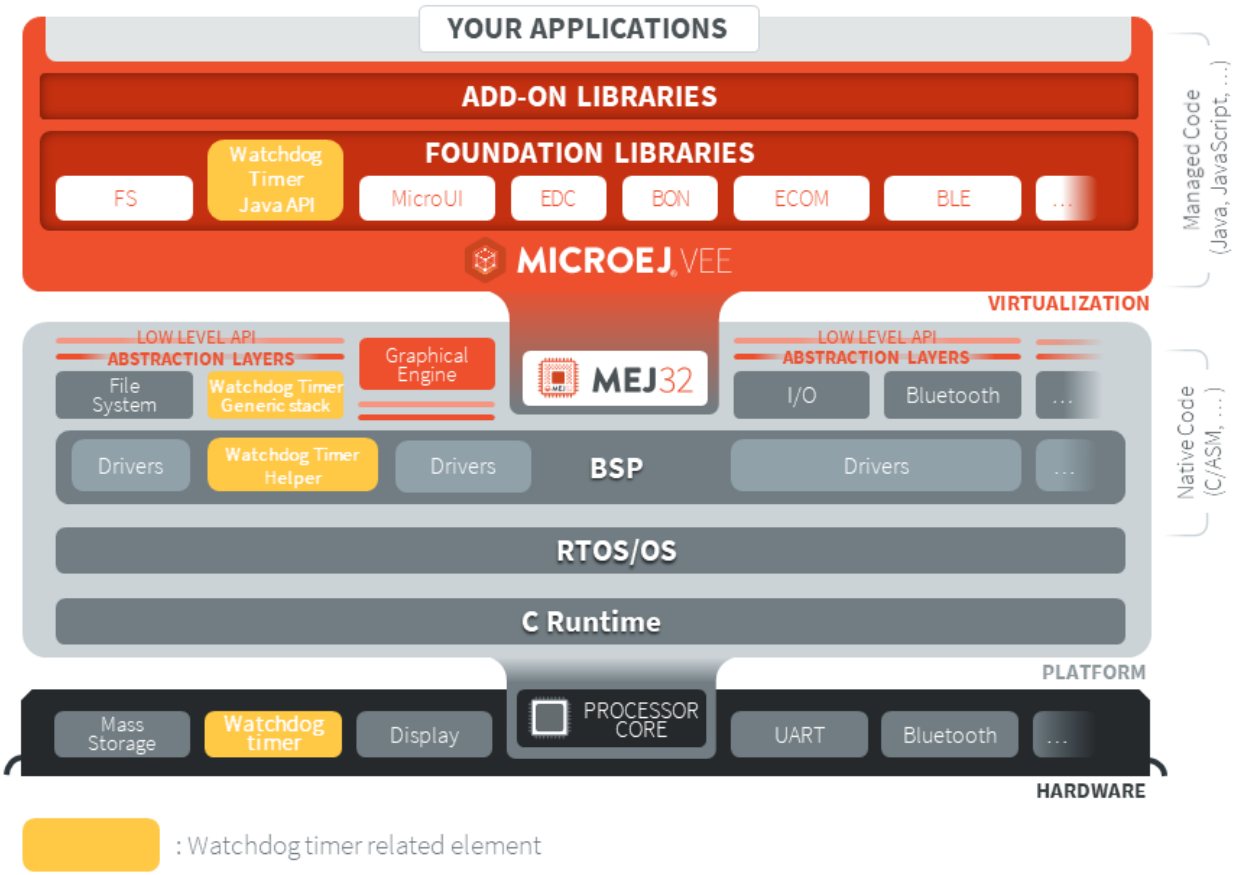
The Security API module must be added to the *module.ivy* of the MicroEJ Application project to use the Security library.

```
<dependency org="ej.api" name="security" rev="1.6.0"/>
```

6.23 Watchdog Timer

6.23.1 Overview

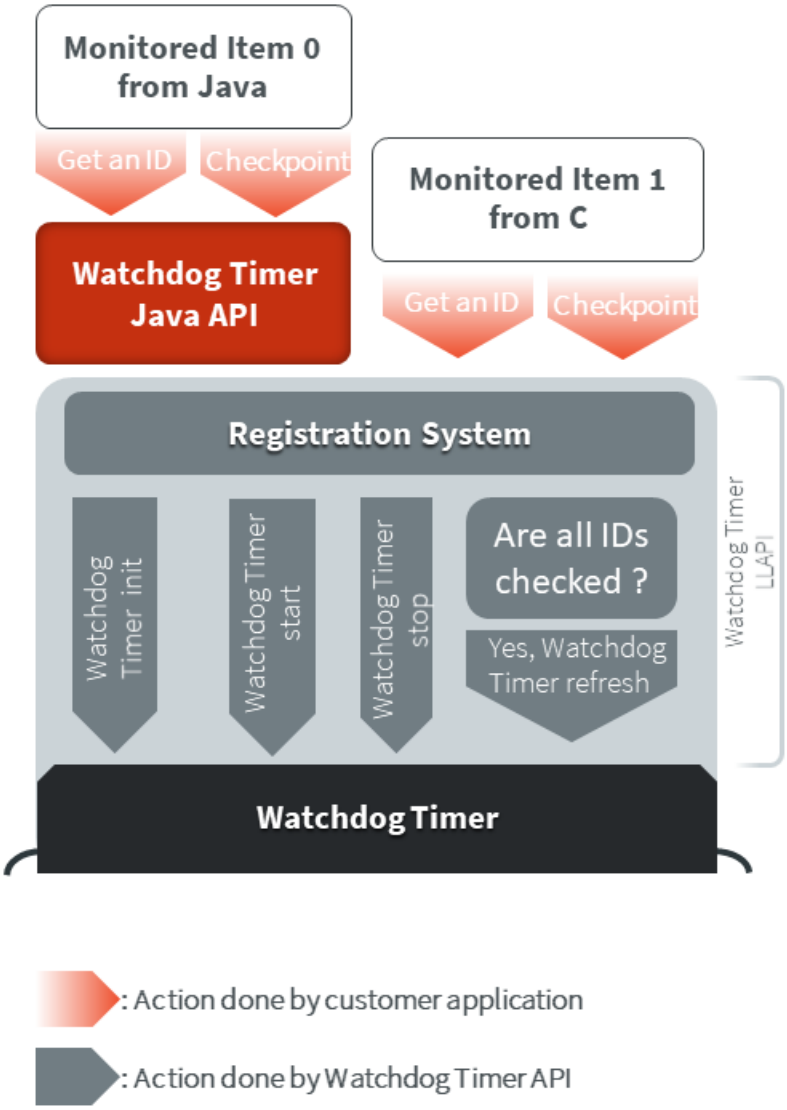
The WatchdogTimer Foundation Library provides a way to handle hardware watchdog timer. A watchdog is particularly useful if you want to monitor different items of your software system during the runtime. The figure below shows watchdog elements at each level of a MicroEJ project:



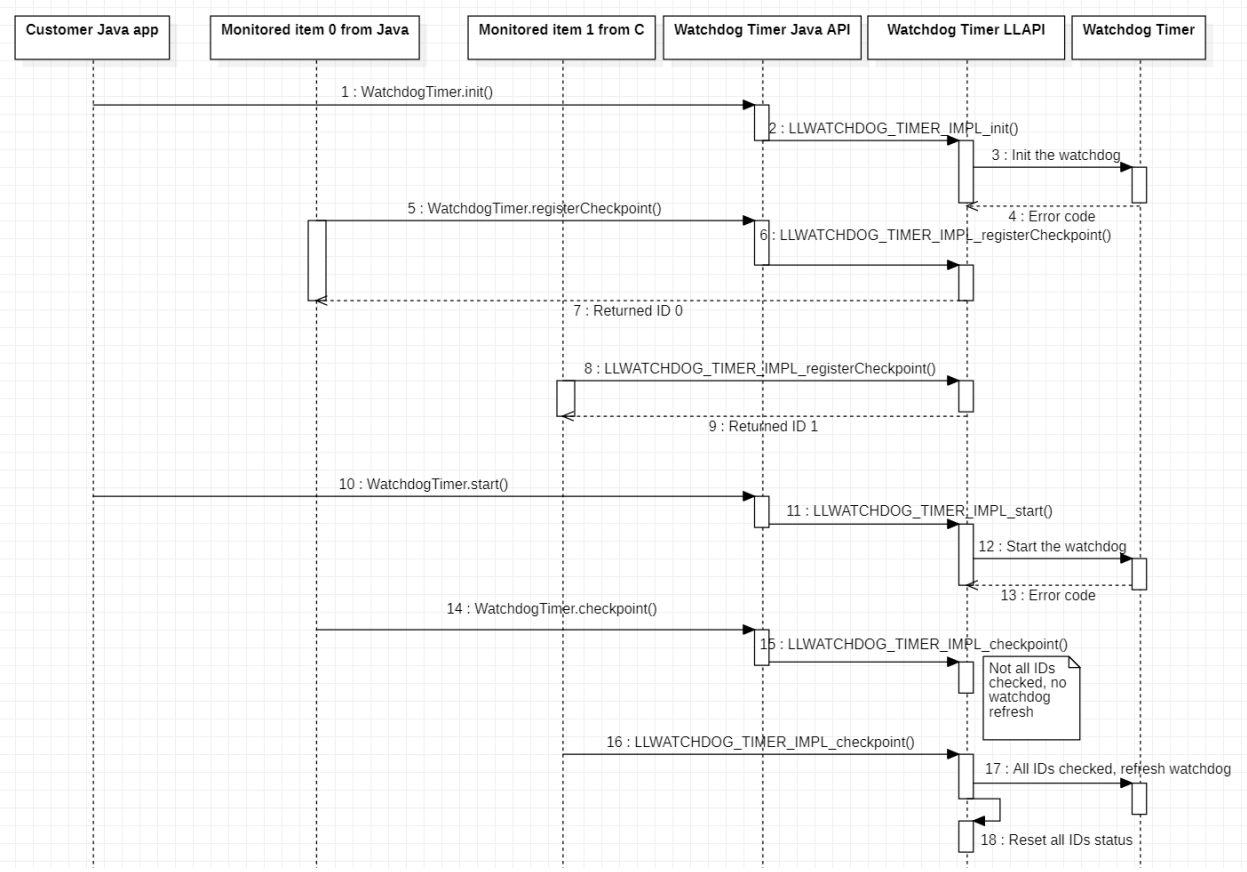
6.23.2 Principle

The Watchdog Timer API is in **two parts**, the first part drives the **watchdog timer** itself. The second part of the API implements a **checkpoint registration system** linked to the watchdog timer.

The checkpoint registration system allows the user to add checkpoints monitored by the hardware watchdog timer. Each checkpoint registered by the Watchdog Timer API must attest their activity before the watchdog timeout, otherwise a **hardware reset** is performed. The high level diagram below summarizes interactions between the user application, the Watchdog Timer API and the Watchdog timer.



The particularity of this library is that it can be either used in Java, in C inside the BSP or even both of them. The use of this library in the BSP in C is relevant when the user needs to monitor an item of the software system which is outside of the MicroEJ Virtual Machine. The sequence diagram below shows a standard use of the Watchdog API in Java and in C.



6.23.3 Mock Implementation

When you run your Application on the Simulator, the watchdog timer must be emulated. To do so, a Java Timer Task is used which emulates the watchdog timer.

The Mock implementation does not perform a hardware reset when the false watchdog timer triggers.

6.23.4 Dependencies

- This library needs to be used with the Watchdog Timer Generic C implementation developed for this purpose, its module name is `watchdog-timer-generic`.
- `LLWATCHDOG_TIMER_impl.h` implemented by the Watchdog Timer C implementation (see *LLWATCHDOG_TIMER: Watchdog Timer*).
- `watchdog_timer_helper.h` implementation needed by the Watchdog Timer C implementation (see *LLWATCHDOG_TIMER: Watchdog Timer*).

6.23.5 Installation

Watchdog Timer is an API composed of a *Pack* module and a C component module. You need both of them in your VEE Port to install the API.

In the Platform configuration project, (`-configuration` suffix), add the following dependencies to *module.ivy* file:

```
<dependency org="com.microej.pack.watchdog-timer" name="watchdog-timer-pack" rev="2.0.1" />
<dependency org="com.microej.clibrary.llimpl" name="watchdog-timer-generic" rev="3.0.1"/>
```

The Platform project must be rebuilt (*Platform Build*).

Then, you have to implement functions that match the `LLWATCHDOG_TIMER_IMPL_*_action` pattern which is required by the Watchdog C implementation.

6.23.6 Use in an Application

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

```
<dependency org="ej.api" name="watchdog-timer" rev="2.0.0"/>
```

6.23.7 Code example in Java

Here is an example that summarizes all features in a simple use case. The checkpoint is performed in a TimerTask scheduled to run every 5 seconds. To use TimerTask in your Java application, add the following BON API dependency:

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

Then, you can use this example code:

```
// Test a simple watchdog timer use case
public static void main(String[] args) {

        if (WatchdogTimer.isResetCause()) {
                System.out.println("Watchdog timer triggered the last board reset!"); //$NON-
↪NLS-1$
        } else {
                System.out.println("Watchdog timer DID NOT triggered the last board reset!");
↪ //$NON-NLS-1$
        }

        WatchdogTimer.init();
        System.out.println("Watchdog timer initialized to trigger after " + WatchdogTimer.
↪getWatchdogTimeoutMs() + " ms."); //$NON-NLS-1$

        TimerTask checkpointTask = new TimerTask() {

                private final int checkpointId = WatchdogTimer.registerCheckpoint();

                @Override
                public void run() {
```

(continues on next page)

(continued from previous page)

```
                        // We attest our task activity using the checkpoint method.
                        // Since this is our only checkpoint registered, the watchdog timer␣
↪is refreshed.
                        WatchdogTimer.checkpoint(this.checkpointId);
                        System.out.println("Task performed watchdog checkpoint with the ID "␣
↪+ this.checkpointId); //$NON-NLS-1$
                }
        };

        // We schedule our task to be executed every 5 seconds.
        Timer timer = new Timer();
        final int DELAY = 0;
        final int PERIOD = 5000; // We assume that the watchdog timeout period is higher␣
↪than 5000 milliseconds.
        timer.schedule(checkpointTask, DELAY, PERIOD);

        // Everything is ready, we launch the watchdog
        WatchdogTimer.start();
        System.out.println("Watchdog started!");

        // Let the checkpointTask runs for a minute.

        final int WAIT_A_MINUTE = 60000; // 60 000 milliseconds to wait a minute
        try {
                Thread.sleep(WAIT_A_MINUTE);
        } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
        }

        // Our program is finished. Now we stop the checkpointTask and the watchdog.
        timer.cancel();
        WatchdogTimer.stop(); // This method also unregisters all checkpoints.
        System.out.println("Monitored task stopped and Watchdog timer stopped.");
}
```

6.23.8 Use in C inside the BSP

Once the Platform is configured to use the Watchdog Timer API as explained in `Installation` section, you can use functions defined in `LLWATCHDOG_TIMER_impl.h`.

Note that compared to the Java API, you have to get error codes returned by functions to check if the function is executed correctly since you have no access to exceptions generated for the Java.

The Watchdog Timer Low Level API provides a set of functions with the same usage as in Java. Here is the list of the watchdog Low Level API functions:

```
LLWATCHDOG_TIMER_IMPL_init()                    // refer to ej.hal.WatchdogTimer.init()
LLWATCHDOG_TIMER_IMPL_start()                   // refer to ej.hal.WatchdogTimer.start()
LLWATCHDOG_TIMER_IMPL_stop()                    // refer to ej.hal.WatchdogTimer.stop()
LLWATCHDOG_TIMER_IMPL_registerCheckpoint()      // refer to ej.hal.WatchdogTimer.
↪registerCheckpoint()
```

(continues on next page)

(continued from previous page)

```
LLWATCHDOG_TIMER_IMPL_unregisterCheckpoint()      // refer to ej.hal.WatchdogTimer.
↪unregisterCheckpoint()
LLWATCHDOG_TIMER_IMPL_checkpoint()                // refer to ej.hal.WatchdogTimer.
↪checkpoint()
LLWATCHDOG_TIMER_IMPL_isResetCause()              // refer to ej.hal.WatchdogTimer.
↪isResetCause()
LLWATCHDOG_TIMER_IMPL_getWatchdogTimeoutMs()      // refer to ej.hal.WatchdogTimer.
↪getWatchdogTimeoutMs()
```

There is an additional function in `LLWATCHDOG_TIMER_impl.h` compared to the Java API. This is `LLWATCHDOG_TIMER_IMPL_refresh` , because a low level implementation of this function is required for the library. However, the user does not need and should not use this function on his own.

6.23.9 Code example in C

Here is an example that summarizes main features in a simple use case. The checkpoint is performed in a FreeRTOS task scheduled to attest its activity to the watchdog every 5 seconds.

```
#include <stdio.h>
#include <stdint.h>

#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "semphr.h"

#include "LLWATCHDOG_TIMER_impl.h"

#define MONITORED_TASK_STACK_SIZE 1024
#define TASK_SLEEP_TIME_MS 5000 // We sleep for 5 seconds, assuming that the watchdog␣
↪timeout is higher.

/*-----------------------------------------------------------*/

static void my_monitored_task( void *pvParameters ){
        // We get an ID from watchdog registration system for this new checkpoint
        int32_t checkpoint_id = LLWATCHDOG_TIMER_IMPL_registerCheckpoint();

        for(;;){
                vTaskDelay( TASK_SLEEP_TIME_MS / portTICK_PERIOD_MS);
                // Since this is our only checkpoint registered, the watchdog timer is␣
↪refreshed.
                LLWATCHDOG_TIMER_IMPL_checkpoint(checkpoint_id);
                printf("MonitoredTask with ID = %d did watchdog checkpoint!\n", checkpoint_
↪id);
        }
}

/*-----------------------------------------------------------*/

int main( void ){
```

(continues on next page)

(continued from previous page)

```
        xTaskHandle handle_monitored_task;

        /* Check if last reset was done by the Watchdog timer. */
        if(LLWATCHDOG_TIMER_IMPL_isResetCause()){
                printf("Watchdog timer triggered the last reset, we stop the program now! \n
↪");
                return -1;
        }

        /* Setup the Watchdog Timer*/
        if(WATCHDOG_TIMER_ERROR == LLWATCHDOG_TIMER_IMPL_init()){
                printf("Failed to init watchdog timer in main. \n");
        } else{
                printf("Watchdog timer initialized to trigger after %d ms \n", LLWATCHDOG_
↪TIMER_IMPL_getWatchdogTimeoutMs());
        }

        /* Start the Watchdog Timer*/
        if(WATCHDOG_TIMER_ERROR == LLWATCHDOG_TIMER_IMPL_start()){
                printf("Failed to start watchdog timer in main. \n");
        } else{
                printf("Watchdog started!\n");
        }

        /* Create the monitored task. */
        xTaskCreate( my_monitored_task, "MonitoredTask", MONITORED_TASK_STACK_SIZE, NULL,␣
↪tskIDLE_PRIORITY, &handle_monitored_task);

        /* Start the scheduler. */
        printf("Starting scheduler...\n");
        vTaskStartScheduler();

        return 0;
}
```

6.24 SystemView

6.24.1 Principle

SystemView is a real-time recording and visualization tool for embedded systems that reveals the actual runtime behavior of an application, going far deeper than the system insights provided by debuggers. This is particularly effective when developing and working with complex embedded systems comprising multiple threads and interrupts: 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.

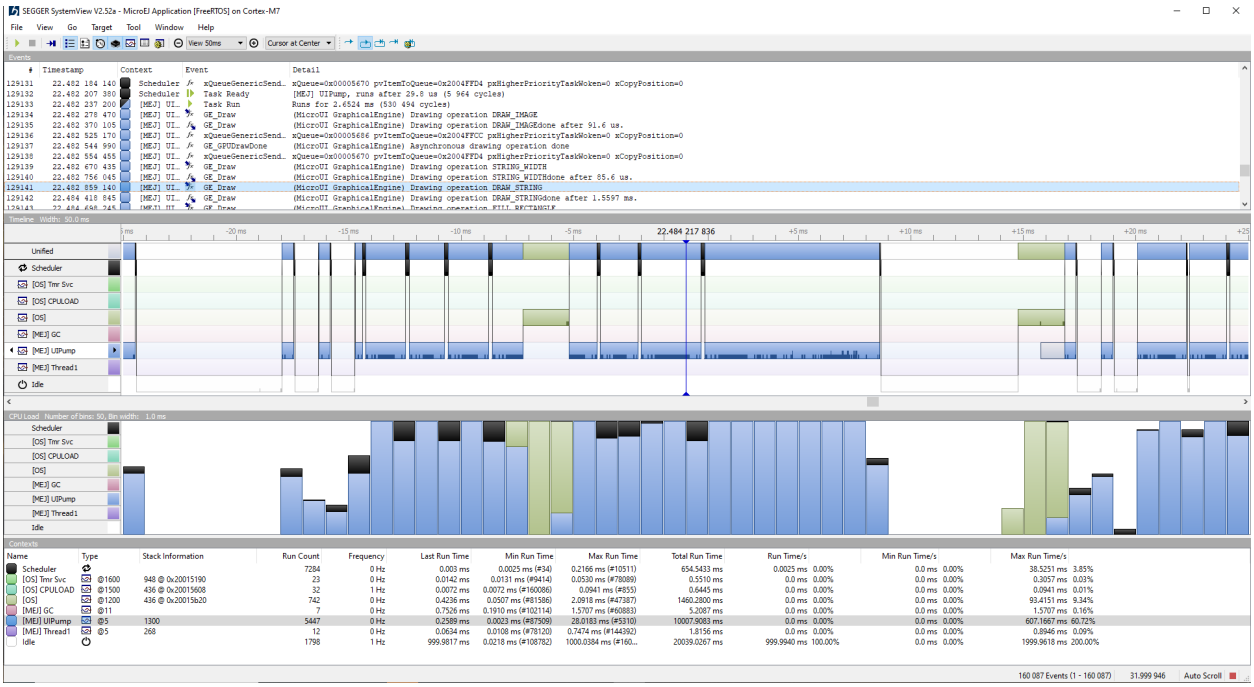
A specific SystemView extension made by MicroEJ allows to trace the OS tasks and the MicroEJ Java threads at the same time. This chapter explains how to add SystemView feature to a platform and set it up.

A SystemView support is provided to use the software with a MicroEJ system. This documentation shows how to set up your BSP and your Java application.

Note: SystemView support for MicroEJ is compatible with FreeRTOS 9 and FreeRTOS 10.

Note: This SystemView section has been written for SystemView version V2.52a. Later versions may or may not work, and may need modification to the following steps.

Here is an example when analyzing the Demo Widget running on the STM32F7508-DK platform.



6.24.2 References

- https://www.segger.com/products/development-tools/systemview/
- https://www.segger.com/downloads/jlink/UM08027

6.24.3 Installation

SystemView installation consists of adding several items in the BSP. The following steps describe them, and they must be performed in the right order. If SystemView support 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 V2.52a: http://segger.com/downloads/systemview/.
2. Apply SystemView for FreeRTOS patch as described in the documentation (https://www.segger.com/downloads/jlink/UM08027); the patch is available in the installation folder `SEGGER\SystemView\Src\Sample\FreeRTOSVxx`.

Note: If you are using FreeRTOS V10.2.0, use the patch located here: https://forum.segger.com/index.php/Thread/6158-SOLVED-SystemView-Kernelpatch-for-FreeRTOS-10-2-0/?s=add3b0f6a33159b9c4b602da0082475afeceb89a

3. Check if the patch disabled SystemView systick events in `port.c`, if not remove these lines manually:



4. Add `SEGGER\SystemView\Src\Sample\FreeRTOSVxx\Config\SEGGER_SYSVIEW_Config_FreeRTOS.c` in your BSP.

This file can be modified to fit your system configuration:

- Update `SYSVIEW_APP_NAME`, `SYSVIEW_DEVICE_NAME`, and `SYSVIEW_RAM_BASE` defines to fit your system information.
- To add MicroEJ Java threads management in SystemView tasks initialization:
  - Add these includes `#include "task.h"`, `#include "LLMJVM_MONITOR_SYSVIEW.h"`, `#include "LLTRACE_SYSVIEW_configuration.h"`, `#include "SEGGER_SYSVIEW_configuration.h"` and the include that declares the external variable `pvMEJCoreEngineTask`. `pvMEJCoreEngineTask` must be the FreeRTOS task handle used to create the MicroEJ Core Engine task. Initializes this variable at `NULL` before the call of the FreeRTOS scheduler.

- In function `_cbSendSystemDesc(void)`, add this instruction: `SEGGER_SYSVIEW_SendSysDesc("N="SYSVIEW_APP_NAME",D="SYSVIEW_DEVICE_NAME", O=FreeRTOS");` before `SEGGER_SYSVIEW_SendSysDesc("I#15=SysTick");`.
- Replace the `Global function` section with this code:

```
/*********************************************************************
*
*       Global functions
*
**********************************************************************
*/

SEGGER_SYSVIEW_OS_API SYSVIEW_MICROEJ_X_OS_TraceAPI;

static void SYSVIEW_MICROEJ_X_OS_SendTaskList(void){
   SYSVIEW_X_OS_TraceAPI.pfSendTaskList();

// The strategy to send tasks info is different in post mortem and live␣
↪analysis.
#if (1 == SEGGER_SYSVIEW_POST_MORTEM_MODE)
   /**
   * POST MORTEM analysis
   *
   * Using the post mortem analysis, FreeRTOS tasks regularly call the SYSVIEW_
↪MICROEJ_X_OS_SendTaskList() function when
   * a packet (systemview event) is sent to the SEGGER circular buffer. It is␣
↪necessary because the information of tasks
   * must be regularly uploaded in the circular buffer in order to provide a␣
↪valid analysis at any moment.
   * Consequently, we only allow to call LLMJVM_MONITOR_SYSTEMVIEW_send_task_
↪list() when the current task is the MicroEJ Core Engine.
   */

   /* Obtain the handle of the current task. */
   TaskHandle_t xHandle = xTaskGetCurrentTaskHandle();
   configASSERT( xHandle ); // Check the handle is not NULL.

   // Check if the current task handle is the MicroEJ Core Engine task handle.␣
↪pvMEJCoreEngineTask is an external variable.
   if( xHandle == pvMEJCoreEngineTask){
      // Launched by the MicroEJ Core Engine, we execute LLMJVM_MONITOR_
↪SYSTEMVIEW_send_task_list()
      LLMJVM_MONITOR_SYSTEMVIEW_send_task_list();
   }
#else
   /**
   * LIVE analysis
   *
   * Using the live analysis, the call of SYSVIEW_MICROEJ_X_OS_SendTaskList()␣
↪is triggered by
   * the SystemView Software through the J-Link probe. Consequently, the␣
↪MicroEJ Core Engine task will never call
   * the function LLMJVM_MONITOR_SYSTEMVIEW_send_task_list(). However, if the␣
```

(continues on next page)

(continued from previous page)

```
↪MicroEJ Core Engine task is created,
   * the function must be called LLMJVM_MONITOR_SYSTEMVIEW_send_task_list().
   */
   // Check if the MicroEJ Core Engine task handle is not NULL.␣
↪pvMEJCoreEngineTask is an external variable.
   if( NULL != pvMEJCoreEngineTask){
      // The MicroEJ Core Engine task is running, we execute LLMJVM_MONITOR_
↪SYSTEMVIEW_send_task_list()
      LLMJVM_MONITOR_SYSTEMVIEW_send_task_list();
   }
#endif
}

void SEGGER_SYSVIEW_Conf(void) {
   SYSVIEW_MICROEJ_X_OS_TraceAPI.pfGetTime = SYSVIEW_X_OS_TraceAPI.pfGetTime;
   SYSVIEW_MICROEJ_X_OS_TraceAPI.pfSendTaskList = SYSVIEW_MICROEJ_X_OS_
↪SendTaskList;

   SEGGER_SYSVIEW_Init(SYSVIEW_TIMESTAMP_FREQ, SYSVIEW_CPU_FREQ, &SYSVIEW_
↪MICROEJ_X_OS_TraceAPI, _cbSendSystemDesc);
   SEGGER_SYSVIEW_SetRAMBase(SYSVIEW_RAM_BASE);
}
```

5. Add in your BSP the MicroEJ C module files for SystemView: com.microej.clibrary.thirdparty#systemview (or check the differences between pre-installed SystemView and C files provided by this module)
6. Add in your BSP the MicroEJ C module files for SystemView FreeRTOS support (or check the differences between pre-installed SystemView and C files provided by this module)
   - FreeRTOS 10: com.microej.clibrary.thirdparty#systemview-freertos10
   - FreeRTOS 9: please contact *our support team* to get the latest maintenance version of `com.microej.clibrary.thirdparty#systemview-freertos9` module.
7. Install the Abstraction Layer implementation of the *Java Trace API* for SystemView by adding C module files in your BSP: com.microej.clibrary.llimpl#trace-systemview
8. Make FreeRTOS compatible with SystemView: open `FreeRTOSConfig.h` and:
   - add `#define INCLUDE_xTaskGetIdleTaskHandle 1`
   - add `#define INCLUDE_pxTaskGetStackStart 1`
   - add `#define INCLUDE_uxTaskPriorityGet 1`
   - comment the line `#define traceTASK_SWITCHED_OUT()` if defined
   - comment the line `#define traceTASK_SWITCHED_IN()` if defined
   - add `#include "SEGGER_SYSVIEW_FreeRTOS.h"` at the end of the file
9. Enable SystemView on startup (before creating the first OS task): call `SEGGER_SYSVIEW_Conf();`. The following include directive is required: `#include "SEGGER_SYSVIEW.h"`.
10. Print the RTT block address to the serial port on startup: `printf("SEGGER_RTT block address: %p\n", &(_SEGGER_RTT));`. The following include directive is required: `#include "SEGGER_RTT.h"`.

Note: This is useful if SystemView does not automatically find the RTT block address. See section *RTT Control Block*

Not Found for more details.

Note: You may also find the RTT block address in RAM by searching `_SEGGER_RTT` in the .map file generated with the firmware binary.

11. Add a call to `SEGGER_SYSVIEW_setMicroJVMTask((U32)pvCreatedTask);` just after creating the OS task to register the MicroEJ Core Engine OS task. The handler to give is the one filled by the `xTaskCreate` function.
12. Copy the file `/YourPlatformProject-bsp/projects/microej/trace/systemview/SYSVIEW_MicroEJ.txt` to the SystemView installation path, such as `SEGGER/SystemView_V252a/Description/`. If you use MicroUI traces, you can also copy the file in the section *Debug Traces*

6.24.4 MicroEJ Core Engine OS Task

The *MicroEJ Core Engine* task is the OS task that executes MicroEJ Java threads. Once it is *started* (by calling `SNI_startVM`), it executes the initialization code and rapidly starts to execute the MicroEJ Application main thread. At that time, the events produced by this OS task (context switch, semaphores, etc.) are dispatched to the current MicroEJ Java thread. Consequently, this OS task is useless when the MicroEJ Application is running.

SystemView for MicroEJ disables the visibility of this OS task when the MicroEJ Application is running. It simplifies the SystemView client debugging.

6.24.5 OS Tasks and Java Threads Names

To make a distinction between the OS tasks and the MicroEJ Java threads, a prefix is added to the OS tasks names (`[OS]`) and the Java threads names (`[MEJ]`).

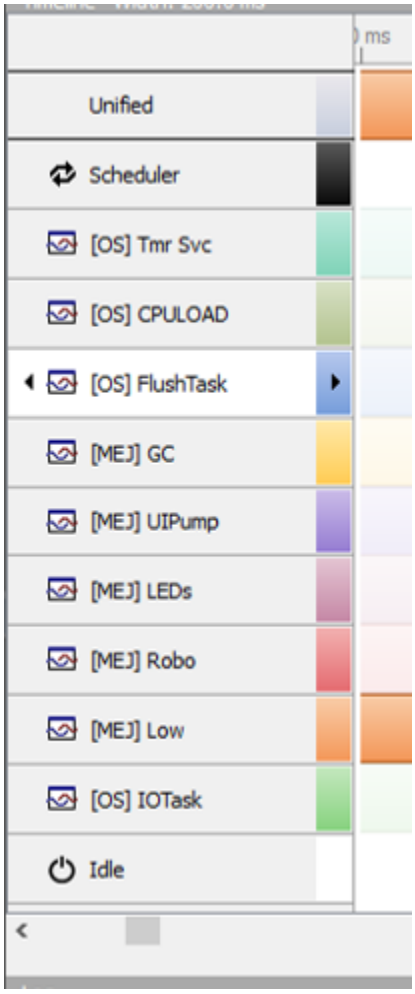


Fig. 72: OS Tasks and Java Threads Names

Note: SystemView limits the number of characters to 32. The prefix length is included in these 32 characters; consequently, the end of the original OS task or Java thread name can be cropped.

6.24.6 OS Tasks and Java Threads Priorities

SystemView lists the OS tasks and Java threads according to their priorities. However, the priority notion does not have the same signification when talking about OS tasks or Java threads: a Java thread priority depends on the MicroEJ Core Engine OS task priority.

As a consequence, a Java thread with the priority `5` may not appear between an OS task with the priority `4` and another OS task with priority `6`:

- if the MicroEJ Core Engine OS task priority is `3`, the Java thread must appear below an OS task with priority `4`.
- if the MicroEJ Core Engine OS task priority is `7`, the Java thread must appear above an OS task with priority `6`.

To keep a consistent line ordering in SystemView, the priorities sent to the SystemView client respect the following rules:

- OS task: `priority_sent = task_priority * 100`.
- MicroEJ Java thread: `priority_sent = MicroJvm_task_priority * 100 + thread_priority`.

6.24.7 Use

MicroEJ Architecture can generate specific events that allow monitoring of current Java thread, Java exceptions, Java allocations, … as well as custom application events. Please refer to the *Event Tracing* section.

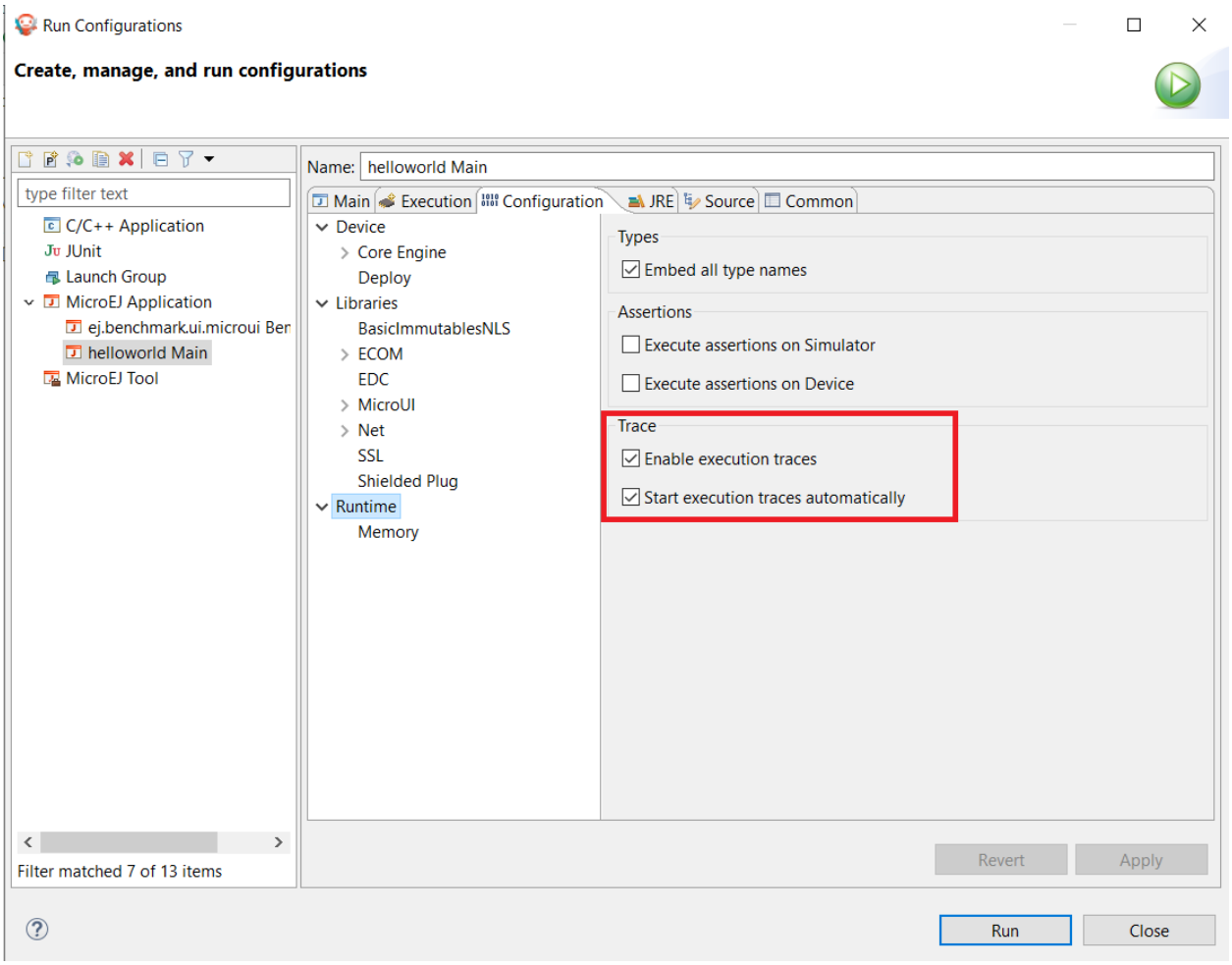
To enable events recording, refer to the *Event Recording* section to configure the required *Application Options*.

6.24.8 Troubleshooting

SystemView doesn't see any activity in MicroEJ Tasks

You have to enable runtime traces of your Java application.

- In Run > Run configuration, select your Java application launcher.
- Then, go to Configuration tab > Runtime > Trace.
- Finally, check checkboxes Enable execution traces and Start execution traces automatically as shown in the picture below.
- Rebuild your firmware with the new Java application version, which should fix the issue.

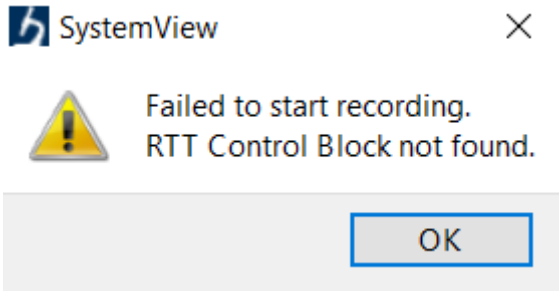


You may only check the first checkbox when you know when you want to start the trace recording. For more information, please refer to the *Event Recording* section to configure the required *Application Options*.

OVERFLOW Events in SystemView

Depending on the application, OVERFLOW events can be seen in System View. To mitigate this problem, the default *SEGGER_SYSVIEW_RTT_BUFFER_SIZE* can be increased from the default 1kB to a more appropriate size of 4kB. Still, if OVERFLOW events are still visible, the user can further increase this configuration found in `/YourPlatformProject-bsp/projects/microej/thirdparty/systemview/inc/SEGGER_SYSVIEW_configuration.h`.

RTT Control Block Not Found



- Get the RTT block address from the standard output by resetting the board (it is printed at the beginning of the firmware program),
- In SystemView, select `Target` > `Start recording` ,
- In `RTT Control Block Detection` , select `Address` and put the address retrieved. You can also try with `Search Range` option.

6.24.9 RTT block found by SystemView but no traces displayed

- Be sure that your MCU is running. The BSP may use semi-hosting traces that block the MCU execution if the application is running out of a Debug session.
- You can check the state of the MCU using J-Link tools such as `J-Link Commander` and `Ozone` to start a Debug session.

6.24.10 Bus hardfault when running SystemView without Java Virtual Machine (JVM)

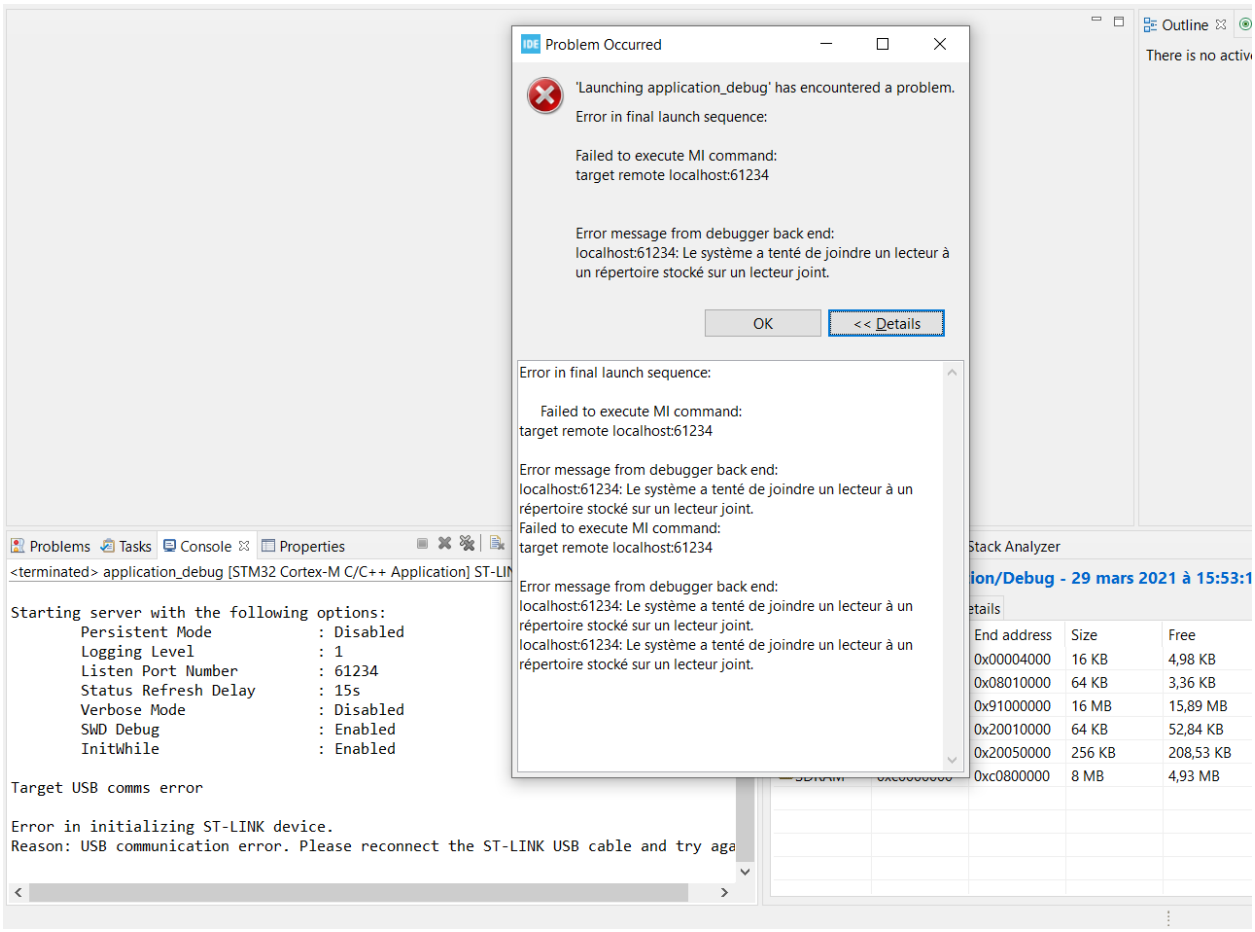
The function `LLMJVM_MONITOR_SYSTEMVIEW_send_task_list();` triggers a `Bus Hardfault` when no JVM is launched. To solve this issue, comment this function call out in `SEGGER_SYSVIEW_Config_FreeRTOS.c` when you run SystemView without launching the JVM.

6.24.11 SystemView for STM32 ST-Link Probe

SystemView software requires a J-Link probe. If your target board uses an ST-Link probe, it is possible to re-flash the ST-LINK on board with a J-Link firmware. See instructions provided by SEGGER Microcontroller https://www.segger.com/products/debug-probes/j-link/models/other-j-links/st-link-on-board/ for more details.

If you cannot flash a firmware for an STM32 device after replacing the J-Link firmware with the ST-Link original one:

- Use ST_Link utility program to update the ST_Link firmware, go to `ST-LINK` > `Firmware update` .
- Then, try to flash again.



6.25 Simulation

6.25.1 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]

6.25.2 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.

[1] Only the execution speed is not accurate. The Simulator speed can be set to match the average MicroEJ Platform speed in order to adapt the Simulator speed to the desktop speed.

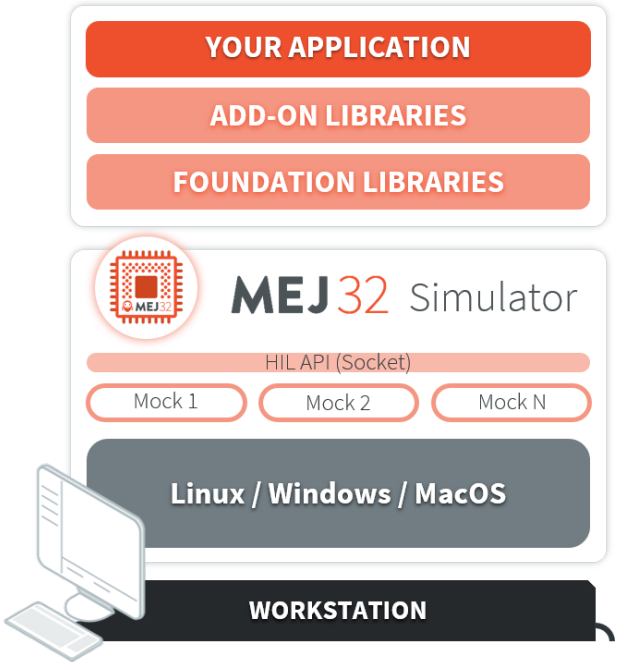


Fig. 73: The HIL Connects the MicroEJ Simulator to the Workstation.

The “simulated C world” is made of Mocks that simulate native code (such as drivers and any other kind of C libraries), so that the MicroEJ Application can behave the same as the device using the MicroEJ Platform.

The MicroEJ Simulator and the HIL are two processes that run in parallel: the communication between them is through a socket connection. Mocks run inside the process that runs the HIL engine.

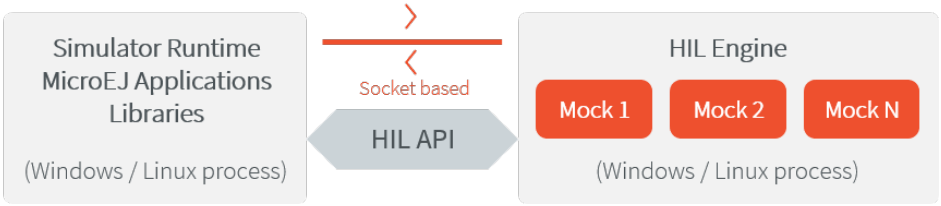


Fig. 74: A MicroEJ Simulator connected to its HIL Engine via a socket.

6.25.3 Dependencies

No dependency.

6.25.4 Installation

The Simulator is a built-in feature of MicroEJ Platform architecture.

6.25.5 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.

6.25.6 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 Virtual Device while still appearing to interact with native code.

Functional Description

As with *SNI*, HIL is responsible for finding the method to execute as a replacement for the native Java method that the MicroEJ Simulator tries to run. Following the SNI specification philosophy, the matching algorithm uses a naming convention. When a native method is called in the MicroEJ Simulator, it requests that the HIL engine execute it. The corresponding Mock executes the method and provides the result back to the MicroEJ Simulator.

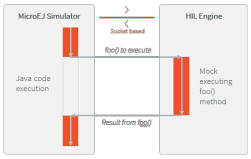


Fig. 75: The MicroEJ Simulator Executes a Native Java Method `foo()` .

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.
 */
```

(continues on next page)

(continued from previous page)

```
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 mock project containing the same `Sensor` class on the same `example` package.

To create a new MicroEJ mock project:

SDK 6

SDK 5

Follow the steps described in *SDK 6 User Guide - Create a Project* depending on your IDE.

- Select File > New > Module Project,
- Fill the module information (project name, module organization, name and revision),
- Select the microej-mock skeleton,
- Click on Finish.

The following code is the required `Sensor` class of the created Mock project:

```
package 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 {
```

(continues on next page)

(continued from previous page)

```
    /**
     * 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.

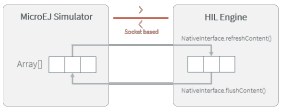


Fig. 76: An Array and Its Counterpart in the HIL Engine.

Below is a typical usage.

```
public static void foo(char[] chars, int offset, int length){
        NativeInterface ni = HIL.getInstance();
        //inside the Mock
        ni.refreshContent(chars, offset, length);
        chars[offset] = 'A';
        ni.flushContent(chars, offset, 1);
}
```

Blocking Native Methods

Some native methods block until an event has arrived *[SNI]*. Such behavior is implemented in native using the following three functions:

- `int32_t SNI_suspendCurrentJavaThread(int64_t timeout)`
- `int32_t SNI_getCurrentJavaThreadID(void)`
- `int32_t SNI_resumeJavaThread(int32_t id)`

This behavior is implemented in a Mock using the following methods on a `lock` object:

- Object.wait(long timeout): Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.
- Object.notifyAll(): Wakes up all the threads that are waiting on this object's monitor.
- `NativeInterface.notifySuspendStart()` : Notifies the Simulator that the current native is suspended so it can schedule a thread with a lower priority.
- `NativeInterface.notifySuspendEnd()` : Notifies the Simulator that the current native is no more suspended. Lower priority threads in the Simulator will not be scheduled anymore.

```
public static byte[] data = new byte[BUFFER_SIZE];
public static int dataLength = 0;
private static Object lock = new Object();

    // Mock native method
    public static void waitForData() {
            NativeInterface ni = HIL.getInstance();
            // inside the Mock
            // wait until the data is received
            synchronized (lock) {
                    while (dataLength == 0) {
                            try {
                                    ni.notifySuspendStart();
```

(continues on next page)

(continued from previous page)

```
                                lock.wait(); // equivalent to lock.wait(0)
                            } catch (InterruptedException e) {
                                // Use the error code specific to your library
                                throw new NativeException(-1, "InterruptedException",␣
↪e);
                            } finally {
                                ni.notifySuspendEnd();
                            }
                        }
                }
        }

// Mock data reader thread
public static void notifyDataReception() {
      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

SDK 6

SDK 5

- Copy the `HILEngine.jar` from the VEE Port into a project folder, for example in `libs`.
- Add a dependency to this local library in the `build.gradle.kts` file:

```
implementation(files("libs/HILEngine.jar"))
```

The HIL Engine API is automatically provided by the `microej-mock` project skeleton.

Installation

SDK 6

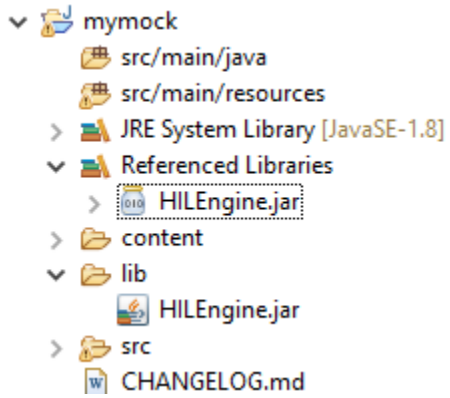
SDK 5

- *Create a J2SE Library project*,
- In the `build.gradle.kts` file, change the `com.microej.gradle.j2se-library` plugin to `com.microej.gradle.mock`.
- Build and publish the Mock by executing the Gradle `publish` task.

Once the module is built, the mock can be installed in a VEE Port in one of the two ways:

- by adding the mock module as a regular VEE Port *module dependency* (if your VEE Port configuration project contains a `module.ivy` file),
- or by manually copying the JAR file `[mock_project]/build/libs/[mock_name]-[mock_version].jar` to the *VEE Port configuration* mock dropins folder `dropins/mocks/dropins/`.

First create a new *module project* using the `microej-mock` skeleton.



Once implemented, right-click on the repository project and select `Build Module`.

Once the module is built, the mock can be installed in a VEE Port in one of the two ways:

- by adding the mock module as a regular VEE Port *module dependency* (if your VEE Port configuration project contains a `module.ivy` file),
- or by manually copying the JAR file `[mock_project]/target~/rip/mocks/[mock_name].jar` to the *VEE Port configuration* mock dropins folder `dropins/mocks/dropins/`.

Make sure the option *Resolve Foundation Libraries in Workspace* is enabled to use the mock without having to install it after each modification during development.

Use

Once installed, a Mock is used automatically by the Simulator when the MicroEJ Application calls a native method which is implemented into the Mock.

JavaFX

JavaFX is an open-source library for creating modern Java user interfaces that is highly portable. It can be used to quickly create graphical Mocks for your VEE Port.

- If your SDK is running on JDK 8, the Oracle JDK contains JavaFX, so this version allows you to use it right now in your project.
- If your SDK is running on JDK 11, JavaFX must be added as an additional dependency to your Mock and VEE Port project. For that, MicroEJ Corp. provides a ready-to-use packaged module for all supported OS versions.

```
<dependency org="com.microej.tool" name="javafx" rev="1.2.0" />
```

The Module serves two purposes, depending on whether it is added to a Mock or a VEE Port project:

- In a Mock project, JavaFX is added as a compile-time dependency, its content is not included in the Mock.
- If your VEE Port contains at least one Mock, JavaFX must be added to the VEE Port project in order to embed its content in the VEE Port.

6.25.7 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.

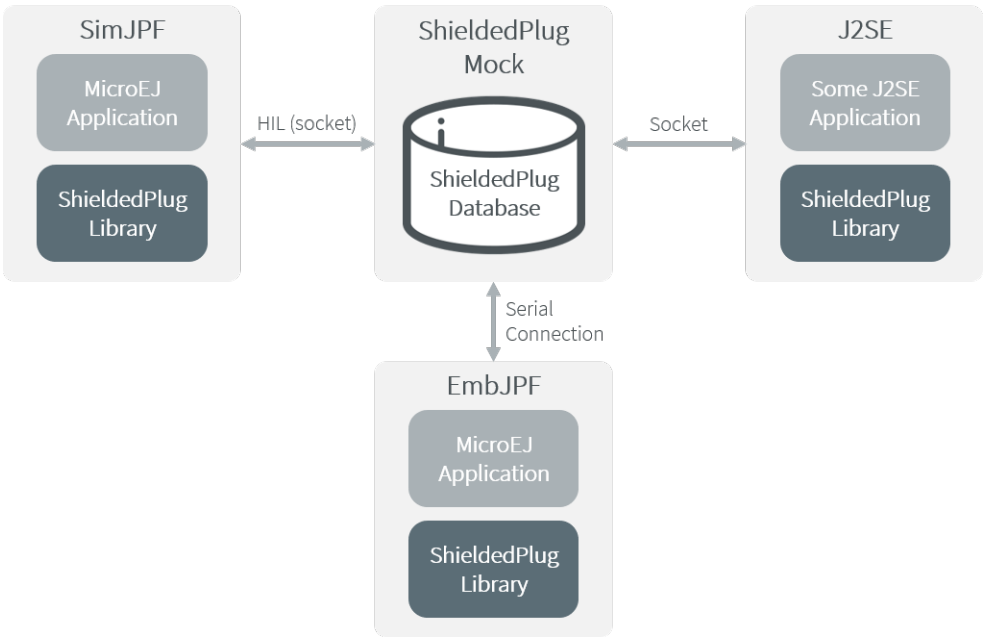


Fig. 77: Shielded Plug Mock General Architecture

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`

6.25.8 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 Front Panel *Widgets* that make the link between the application and the user interactions.
6. Previews the Front Panel to check the layout of controls and the events they create, etc.
7. Exports the Front Panel project into a MicroEJ 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:

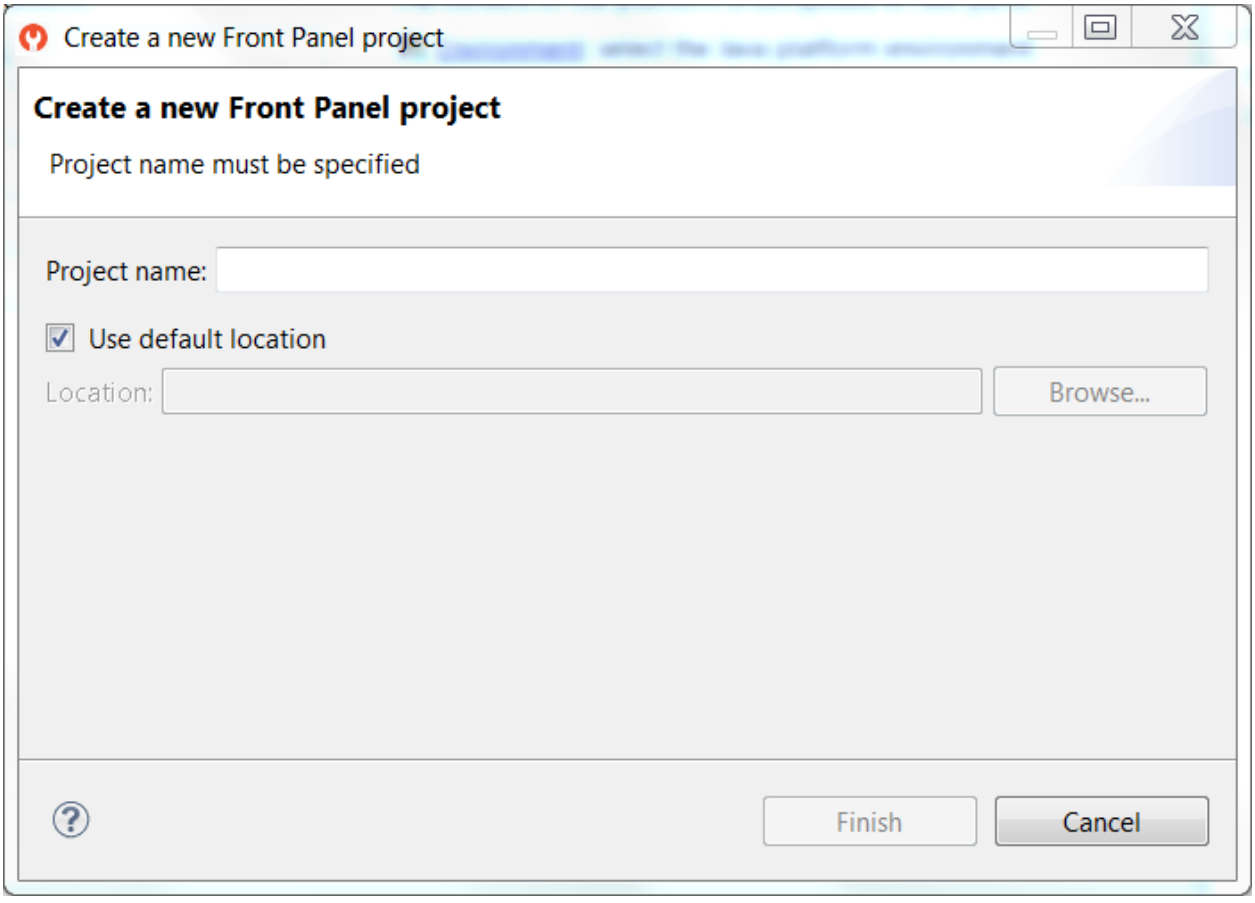


Fig. 78: New Front Panel Project Wizard

Enter the name for the new project.

Project Contents

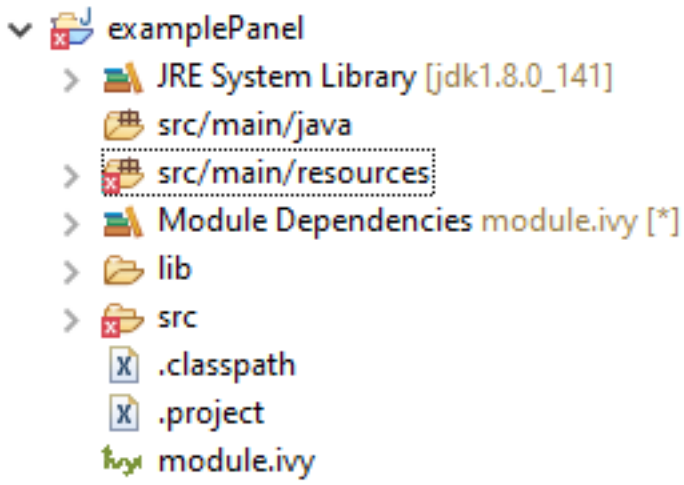


Fig. 79: Project Contents

A Front Panel project has the following structure and contents:

- The `src/main/java` folder is provided for the definition of Front Panel `Widgets`. It is initially empty. The creation of these classes will be explained later.
- The `src/main/resources` folder holds the file or files that define the contents and layout of the Front Panel, with a `.fp` extension (the fp file or files), plus images used to create the Front Panel. A newly created project will have a single fp file with the same name as the project, as shown above. The contents of fp files are detailed later in this document.
- The `JRE System Library` is referenced, because a Front Panel project needs to support the writing of Java for the `Listeners` (and `DisplayExtensions`).
- The `Modules Dependencies` contains the libraries for the Front Panel simulation, the widgets it supports and the types needed to implement `Listeners` (and `DisplayExtensions`).
- The `lib` contains a local copy of `Modules Dependencies`.

Module Dependencies

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"/>
```

(continues on next page)

(continued from previous page)

```
      <conf name="provided" visibility="public" description="Compile-time dependencies to␣
↪APIs provided by the platform"/>
   </configurations>

   <dependencies>
        <dependency org="ej.tool.frontpanel" name="framework" rev="1.1.1"/>
   </dependencies>
</ivy-module>
```

The Front Panel Framework contains the Front Panel core classes, mainly the ability to create your own Front Panel *Widget* to simulate user interactions.

Note: Some Front Panel Widgets are available to interact with the MicroUI devices (display, input devices, etc.), see *Simulation*.

Front Panel File

File Content

The Front Panel engine takes an XML file (the `.fp` file) as input. It describes the panel using widgets: they simulate the drivers, sensors and actuators of the real device. The Front Panel engine generates the graphical representation of the real device, and is displayed in a window on the user's development machine when the application is executed in the Simulator.

The following example file describes a simple board with one LED:

```
<?xml version="1.0"?>
<frontpanel
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns="https://developer.microej.com"
   xsi:schemaLocation="https://developer.microej.com .widget.xsd">

   <device name="MyBoard" skin="myboard.png">
      <ej.fp.widget.LED x="131" y="127" skin="box_led.png"/>
   </device>
</frontpanel>
```

The `device skin` must refer to a `png` file in the `src/main/resources` folder. This image is used to render the background of the Front Panel. The widgets are drawn on top of this background.

The `device` contains the elements that define the widgets that make up the Front Panel. The name of the widget element defines the type of widget. The set of valid types is determined by the Front Panel Designer. Every widget element defines a `label`, which must be unique for widgets of this type (optional or not), and the `x` and `y` coordinates of the position of the widget within the Front Panel (0,0 is top left). There may be other attributes depending on the type of the widget.

The file and tags specifications are available in chapter *Front Panel*.

Note: The `.fp` file grammar has changed since the UI Pack version `12.0.0` (Front Panel core has been moved to MicroEJ Architecture starting from version `7.11.0`). A quick migration guide is available: open Platform con-

figuration 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.

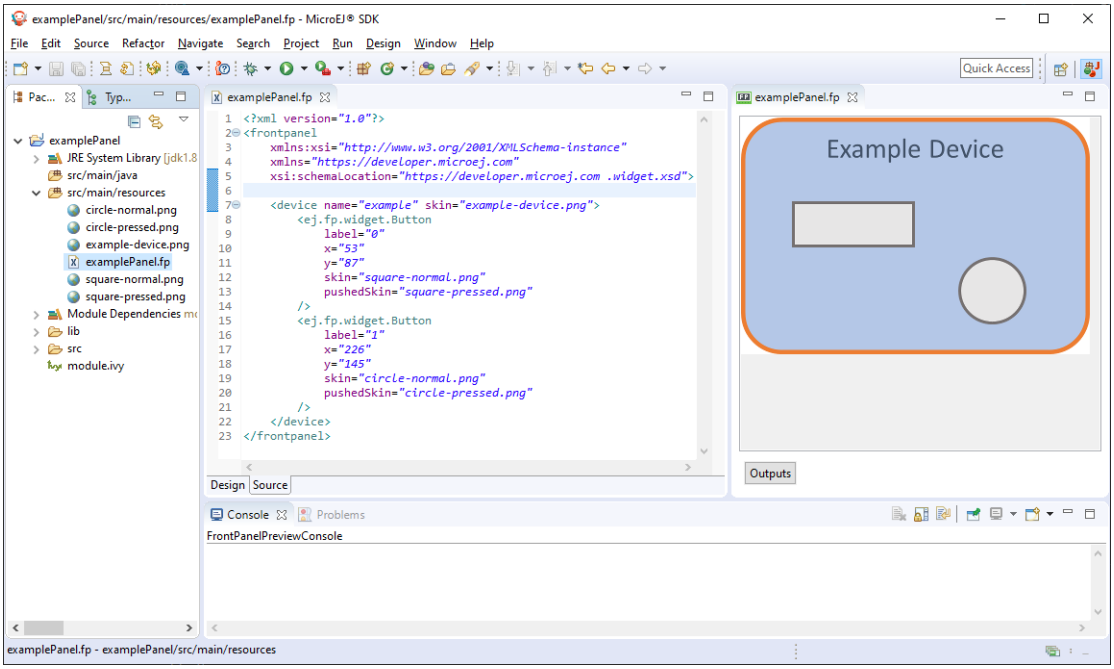


Fig. 80: Working Layout Example

Within the XML editor, content-assist is obtained by pressing `CTRL + SPACE` keys. The editor will list all the elements valid at the cursor position, and insert a template for the selected element.

Multiple Front Panel Files

A Front Panel project can contain multiple `.fp` files. All those files are compiled when exporting the Front Panel project to a VEE Port (or during VEE Port build). It may be useful to have two or more representations of a board (skin, device layout, display size, etc…). When running the simulator, by default, the `.fp` file declared by the *VEE Port configuration*, is used (or a random one if no default is configured). To pick a specific one, set the *Application Option* `frontpanel.file` to a Front Panel simple file name included in the VEE Port (e.g. `myproduct.fp`).

Widget

Description

A widget is a subclass of Front Panel Framework class `ej.fp.Widget`. The library `ej.tool.frontpanel#widget` provides a set of widgets which are Graphics Engine compatible (see *Simulation*). To create a new widget (or a subclass of an existing widget), have a look on available widgets in this library.

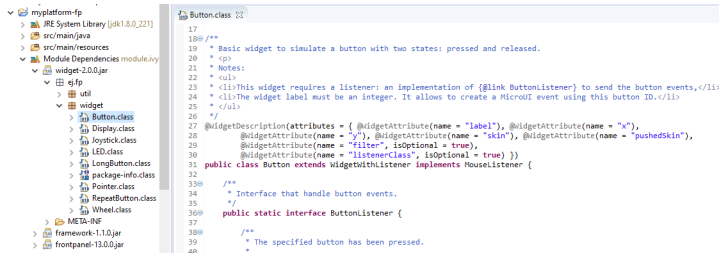


Fig. 81: Front Panel Widgets

A widget is recognized by the `fp` file as soon as its class contains a `@WidgetDescription` annotation. The annotation contains several `@WidgetAttribute`. An attribute has got a name and tells if it is an optional attribute of widget (by default an attribute is mandatory).

This is the description of the widget `LED`:

```
@WidgetDescription(attributes = { @WidgetAttribute(name = "x"),
      @WidgetAttribute(name = "y"), @WidgetAttribute(name = "skin")})
```

As soon as a widget is created (with its description) in Front Panel project, the `fp` file can use it. Close and reopen `fp` file after creating a new widget. In `device` group, press `CTRL + SPACE` keys to visualize the available widgets: the new widget can be added.

```
<ej.fp.widget.LED x="170" y="753" skin="box_led.png" />
```

Each attribute requires the `set` methods in the widget source code. For instance, the widget LED (or its hierarchy) contains the following methods for sure:

- `setX(int)`,
- `setY(int)`,
- `setskin(Image)`.

The `set` method parameter's type fixes the expected value in `fp` file. If the attribute cannot match the expected type, an error is throw when editing `fp` file. Widget master class already provides a set of standard attributes:

- `setFilter(Image)`: apply a filtering image which allows to crop input area (*Input Device Filters*).

- `setWidth(int)` and `setHeight(int)` : limits the widget size.
- `setLabel(String)` : specifies an identifier to the widget.
- `setOverlay(boolean)` : draws widget skin with transparency or not.
- `setSkin(Image)` : specifies the widget skin.
- `setX(int)` and `setY(int)` : specifies widget position.

Notes:

- Widget class does not specify if an attribute is optional or not. It it the responsability to the subclass.
- The label is often used as identifier. It also allows to retrieve a widget calling `Device.getDevice().getWidget(Class<T>, String)` . Some widgets are using this identifier as an integer label. It is the responsability to the widget to fix the signification of the label.
- The widget size is often fixed by the its skin (which is an image). See `Widget.finalizeConfiguration()` : it sets the widget size according the skin if the skin has been set; even if methods `setWidth()` and `setHeight()` have been called before.

Runtime

The Front Panel engine parsing the `fp` file at application runtime. The widget methods are called in two times. First, engine creates widget by widget:

1. widget's constructor: Widget should initialize its own fields which not depend on widget attributes (not valorized yet).
2. `setXXX()` : Widget should check if given attribute value matches the expected behavior (the type has been already checked by caller). For instance if a width is not negative. On error, implementation can throw an `IllegalArgumentException` . These checks must not depend on other attributes because they may have not already valorized.
3. `finalizeConfiguration()` : Widget should check the coherence between all attributes: they are now valorized.

During these three calls, all widgets are not created yet. And so, by definition, the main device (which is a widget) not more. By consequence, the implementation must not try to get the instance of device by calling `Device.getDevice()` . Furthermore, a widget cannot try to get another widget by calling `Device.getDevice().getWidget(s)` . If a widget depend on another widget for any reason, the last checks can be performed in `start()` method. This method is called when all widgets and main device are created. Call to `Device.getDevice()` is allowed.

The method `showYourself()` is only useful when visualizing the `fp` file during its editing (use Eclipse view `Front Panel Preview`). This method is called when clicking on button `Outputs` .

Example

The following code is a simple widget LED. MicroEJ Application can interact with it using native methods `on()` and `off()` of class `ej.fp.widget.LED` :

```
package ej.fp.widget;

import ej.fp.Device;
import ej.fp.Image;
import ej.fp.Widget;
```

(continues on next page)

(continued from previous page)

```
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.
   */
   @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
```

(continues on next page)

(continued from previous page)

```
      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.

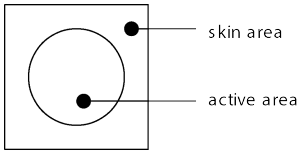


Fig. 82: Active Area

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.

Advanced: Test the Front Panel Project

Note: Starting from SDK 5.7.0 and *Architecture 8.0.0*, the Front Panel projects are automatically resolved in the workspace, so this section and the property `ej.fp.project` are obsolete since. See *Resolve Foundation Libraries in Workspace* for more details.

If the Front Panel project has been created with a SDK version lower than 5.7.0, a project option must be updated:

- right-click on the `Module Dependencies` entry.
- click on `Properties`.
- go to the `Classpath` tab.
- check the `Resolve dependencies in workspace` option.

To quickly test a Front Panel project without rebuilding the Platform or manually exporting the project, add the *Application Option* `ej.fp.project` to the absolute path of a Front Panel project (e.g. `c:\mycompany\myfrontpanel-fp`). The Simulator will use the Front Panel project specified instead of the one included in the Platform. This feature is useful for locally testing some changes in the Front Panel project.

```
-Dej.fp.project=${project_loc:myfrontpanel-fp}
```

Warning: This feature only works if the Platform has been built with the Front Panel module enabled and the Platform does not contain the changes until a new Platform is built.

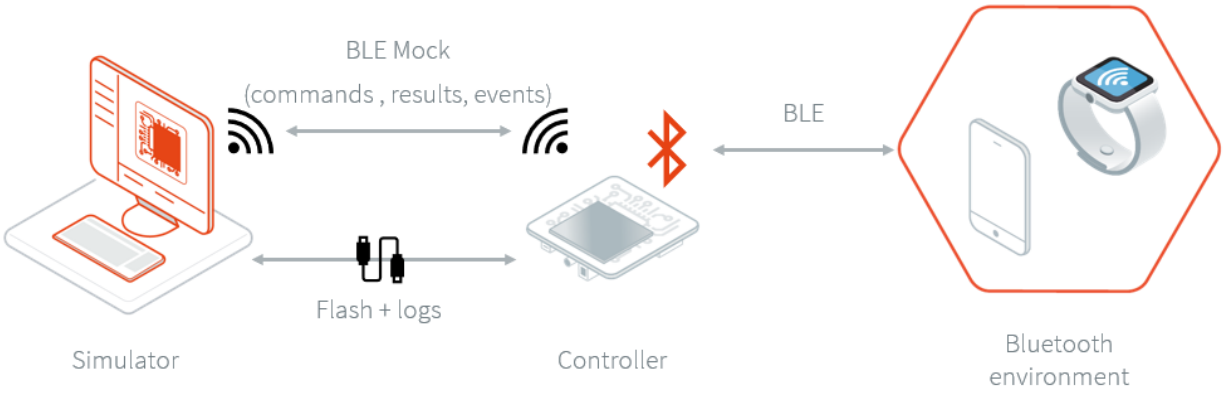
Use

Launch an application on the Simulator to run the Front Panel.

6.25.9 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-S3-DevKitC-1 board reference. Other implementations or sources can be provided on request.

Requirements

- A ESP32-S3-DevKitC-1 board.
- A Bluetooth LE mock controller firmware (this Bluetooth controller executable only works with Bluetooth pack v2).
- A tool to flash the firmware like `https://www.espressif.com/en/support/download/other-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

Unzip `Executable-Bluetooth-Mock-Controller-ESP32-S3-1.0.0.zip`. Inside it you will find the firmware file: `Executable-Bluetooth-Mock-Controller-ESP32-S3-1.0.0.bin`.

To set up the controller, follow these steps:

- Plug-in the ESP32-S3-DevKitC-1 board to your computer,
- Find the associated COM port,
- In the flash tool:
  - select the chip “ESP32-S3”
  - browse for the firmware file
  - set the offset to 0x000000
  - set the SPI speed to 80 Mhz
  - set the SPI mode to DIO
  - set the COM port
  - set the baudrate to 460 800
  - start the flash download

With the flash download tool from Espressif, you should end with something similar to this :

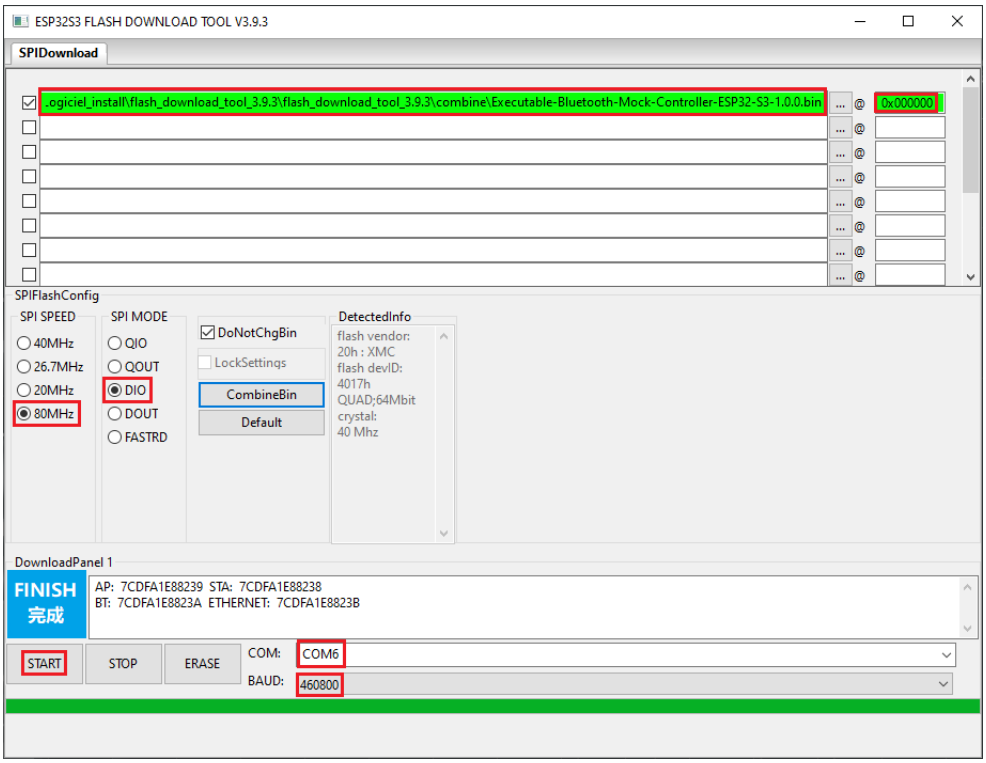
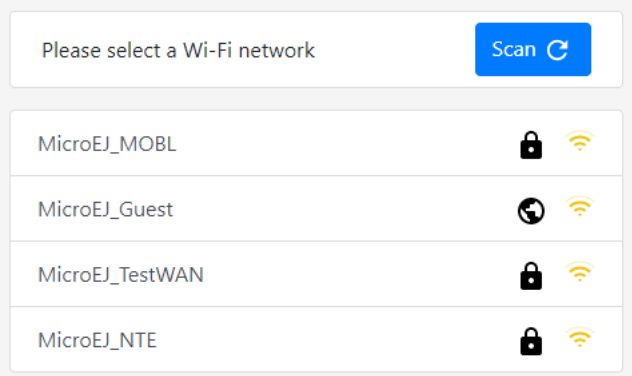


Fig. 83: Bluetooth LE Flash Download Tool Configuration

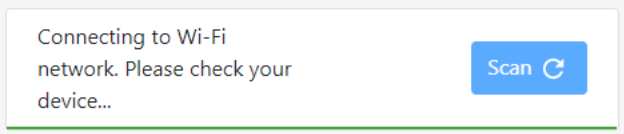
Network Setup

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.
6. Reboot the ESP32-S3-DevKitC-1 board.

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 :

```
ej.bluetooth.bluetoothwificontroller INFO: Joining access point MicroEJ_NTE...
ej.bluetooth.bluetoothwificontroller INFO: Starting server at 192.168.2.108 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.

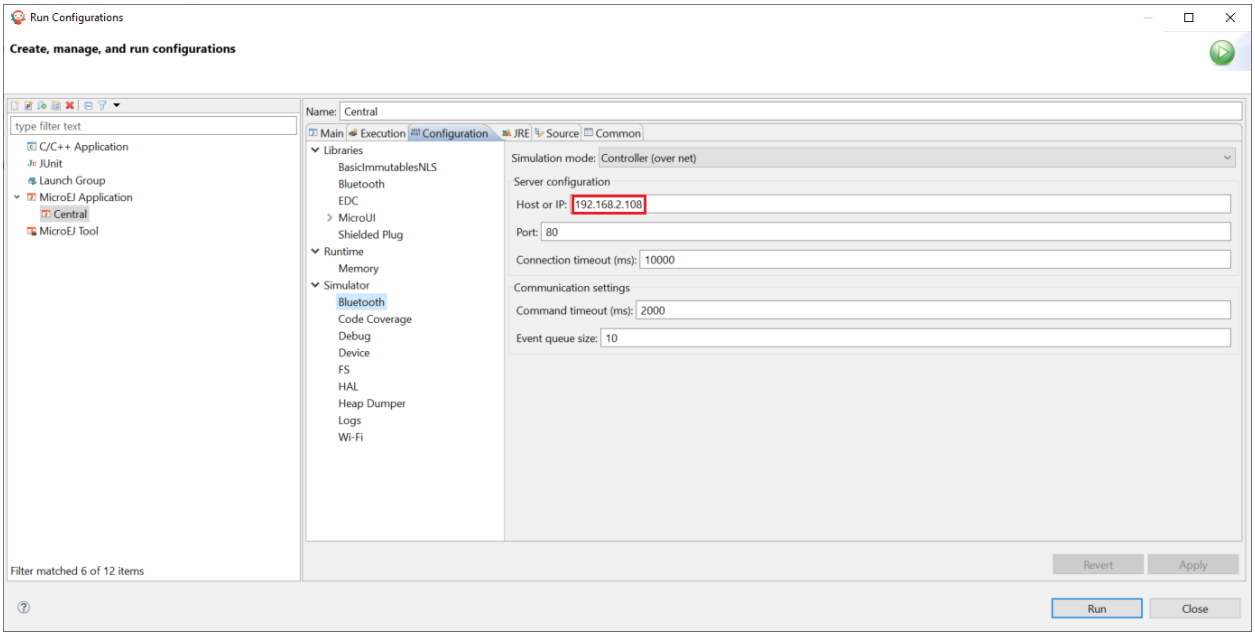


Fig. 84: Bluetooth LE Mock Configuration

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.

"Invalid parameter type: 0x47 expected 0x53" error

Reboot the ESP32-S3-DevKitC-1 board. The controller restarts and connects to the Wi-Fi.

Simulation Errors

Error during the simulation: mock could not connect to controller

This error means the mock process (Simulator) could not initialize the connection with the controller. Please check that the device is connected to the network (see logs in the serial port output) and that your computer is in the same network.

6.26 Appendices

6.26.1 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:

- 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 Shielded Plug for the 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 *Generic Event Generators*).

The UI extension provides an implementation for each of MicroUI's built-in event generators. Each one has dedicated functions that allows a driver to send them structured data without needing to understand the underlying protocol to encode/decode the data. *The following table* shows the functions provided to send structured events to the predefined event generators:

Table 28: LLUI_INPUT API for predefined event generators

Function name	Default event generator kind[Page 1109, 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

Functions LLUI_INPUT_IMPL_log_xxx allow logging the use of event buffer. Implementation of these LLAPIs is already available on the MicroEJ Central Repository (LLUI_INPUT_LOG_impl.c). This implementation is using an array to add some metadata to each event. This metadata is used when the BSP is calling LLUI_INPUT_dump() . When no implementation is included in the BSP, the call to LLUI_INPUT_dump() has no effect (no available logger).

[1] The implementation class is a subclass of the MicroUI class of the column.

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`
- `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 to *give* two binary semaphores.

Image Heap

The display driver must reserve a runtime memory buffer for creating dynamic images when using MicroUI ResourceImage and BufferedImage classes methods. The display driver may choose to reserve an empty buffer. Thus, calling MicroUI methods will result in a MicroUIException exception.

The section name is `.bss.microui.display.imagesHeap` .

Functions `LLUI_DISPLAY_IMPL_image_heap_xxx` allow to control the image buffers allocation in the image heap. Implementation of these LLAPIs is already available on the MicroEJ Central Repository (`LLUI_DISPLAY_HEAP_impl.c`). This implementation is using a best fit allocator. It can be updated to log the allocations, the remaining space, etc. When no implementation is included in the BSP, the default Graphics Engine'a allocator (a best fit allocator) is used.

External Font Heap

The display driver must reserve a runtime memory buffer for loading external fonts (fonts located outside CPU addresses ranges). The display driver may choose to reserve an empty buffer. Thus, calling MicroUI Font methods will result in empty drawings of some characters.

The section name is `.bss.microui.display.externalFontsHeap` .

Flush and Synchronization

The back buffer (graphics buffer) address 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.

LLVG: VectorGraphics

Principle

The *VG Pack* provides a Low Level API for initializing the Vector Graphics engine. The file `LLVG_impl.h`, which comes with the VG Pack, defines the API headers to be implemented.

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 function `LLVG_IMPL_initialize` is the first native function called by the MicroVG implementation. It allows to initialize all C components: GPU initialization, Font engine, heap management, etc.

LLVG_MATRIX: Matrix

Principle

The *Matrix module* provides Low Level APIs for manipulating matrices. The file `LLVG_MATRIX_impl.h`, which comes with the Matrix module, defines the API headers to be implemented.

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.

Implementation

The matrix functions are divided in four groups:

1. identity and copy: fill an identity matrix or copy a matrix to another one.
2. setXXX: erase the content of the matrix by an operation (translate, rotation, scaling, concatenate).
3. xxx (no prefix): perform an operation with the matrix as first argument: `M' = M * xxx(x, y)` where `xxx` is the operation (translate, rotation, scaling, concatenate).
4. postXXX: perform an operation with the matrix as second argument: `M' = xxx(x, y) * M` where `xxx` is the operation (translate, rotation, scaling, concatenate).

LLVG_PATH: Vector Path

Principle

The *Path module* provides Low Level APIs for creating paths in platform specific format. The file `LLVG_PATH_impl.h`, which comes with the Path module, defines the API headers to be implemented. The file `LLVG_PAINTER_impl.h` defines the API headers to be implemented to draw the paths (with a color or a gradient).

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.

Creation

The header file `LLVG_PATH_impl.h` allows to convert a MicroVG library format path in a buffer that represents the same vectorial path in the platform specific format (generally GPU format).

The first function called is `LLVG_PATH_IMPL_initializePath`, which allows the implementation to initialize the path buffer. The buffer is allocated in the Java heap and its size is fixed by the MicroVG implementation. When the buffer is too small for the platform specific format, the implementation has to return the expected buffer size instead of the keyword `LLVG_SUCCESS`.

The next steps consist in appending some commands in the path buffer. The command encoding depends on the platform specific format. When the buffer is too small to add the new command, the implementation has to return a value that indicates the number of bytes the array must be enlarged with.

List of commands:

- `LLVG_PATH_CMD_CLOSE` : MicroVG “CLOSE” command.
- `LLVG_PATH_CMD_MOVE` : MicroVG “MOVE ABS” command.
- `LLVG_PATH_CMD_MOVE_REL` : MicroVG “MOVE REL” command.
- `LLVG_PATH_CMD_LINE` : MicroVG “LINE ABS” command.
- `LLVG_PATH_CMD_LINE_REL` : MicroVG “LINE REL” command.
- `LLVG_PATH_CMD_QUAD` : MicroVG “QUAD ABS” command.
- `LLVG_PATH_CMD_QUAD_REL` : MicroVG “QUAD REL” command.
- `LLVG_PATH_CMD_CUBIC` : MicroVG “CUBIC ABS” command.
- `LLVG_PATH_CMD_CUBIC_REL` : MicroVG “CUBIC REL” command.

List of operations:

- `LLVG_PATH_IMPL_appendPathCommand1` : Adds a command with 1 point parameter in the array.
- `LLVG_PATH_IMPL_appendPathCommand2` : Adds a command with 2 points parameter in the array.
- `LLVG_PATH_IMPL_appendPathCommand3` : Adds a command with 3 points parameter in the array.

A path is automatically closed by the MicroVG implementation (by adding the command `LLVG_PATH_CMD_CLOSE`). A path can be reopened (function `LLVG_PATH_IMPL_reopenPath`), that consists in removing the last added command (`LLVG_PATH_CMD_CLOSE` command) from the buffer.

Drawing

The header file `LLVG_PAINTER_impl.h` provides the functions called by the application via VectorGraphicsPainter to draw a path.

- A path can be drawn with a 32-bit color (ARGB8888): `LLVG_PAINTER_IMPL_drawPath`.
- A path can be drawn with a *linear gradient*: `LLVG_PAINTER_IMPL_drawGradient`.

The drawing destination is symbolized by a MicroUI GraphicsContext: a pointer to a `MICROUI_GraphicsContext` instance. Like MicroUI Painter natives, the implementation has to *synchronize the drawings* with the MicroUI Graphics Engine.

LLVG_GRADIENT: Vector Linear Gradient

Principle

The *Gradient module* provides Low Level APIs for creating linear gradients in platform specific format. The file `LLVG_GRADIENT_impl.h`, which comes with the Gradient module, 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.

Implementation

Only one function has to be implemented: `LLVG_GRADIENT_IMPL_initializeGradient`. It consists in encoding the MicroVG LinearGradient in a buffer that represents the linear gradient in platform specific format (generally GPU format).

This function allows the implementation to initialize the gradient buffer. The buffer is allocated in the Java heap and its size is fixed by the MicroVG implementation. When the buffer is too small for the platform specific format, the implementation has to return the expected buffer size instead of the keyword `LLVG_SUCCESS`.

LLVG_FONT: Vector Font

Principle

The *Font module* provides Low Level APIs for decoding fonts (`LLVG_FONT_impl.h`) and rendering texts (`LLVG_PAINTER_impl.h`). Both header files, which come with the Font module, define the API headers to be implemented.

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 `LLVG_FONT_IMPL_load_font`, which allows the driver to open a font file from its name. This function takes a parameter to configure the text rendering engine:

- Simple layout: uses the glyph advance metrics and the font kerning table.
- Complex layout: uses the font GPOS and GSUB tables.

See VectorFont for more information.

The implementation must manage its own heap to keep the font opened. The font's data are disposed by a call to `LLVG_FONT_IMPL_dispose`.

Font Characteristics

The other functions in `LLVG_FONT_impl.h` consist in retrieving some font characteristics according a text and a font size: string width, string height, baseline, etc.

See VectorFont for more information.

Drawing

The header file `LLVG_PAINTER_impl.h` provides the functions called by the application via VectorGraphicsPainter to draw a path.

- A string can be drawn with a 32-bit color (ARGB8888): `LLVG_PAINTER_IMPL_drawString`.
- A string can be drawn with a *linear gradient*: `LLVG_PAINTER_IMPL_drawStringGradient`.
- A string can be draw on a circle: `LLVG_PAINTER_IMPL_drawStringOnCircle` and `LLVG_FONT_PAINTER_IMPL_drawStringOnCircleGradient`.

The drawing destination is symbolized by a MicroUI GraphicsContext: a pointer to a `MICROUI_GraphicsContext` instance. Like MicroUI Painter natives, the implementation has to *synchronize the drawings* with the MicroUI Graphics Engine.

LLNET: Network

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).

LLECOM_NETWORK: Network Interfaces

Naming Convention

The Low Level Network Interfaces API (LLECOM_NETWORK), relies on functions that need to be implemented by engineers in a driver. The names of these functions match the `LLECOM_NETWORK_IMPL_*` pattern.

Header Files

One header file is provided:

- LLECOM_NETWORK_impl.h

  Defines the set of functions that the BSP must implement to manage and configure and TCP/IP network interfaces.

LLECOM_WIFI: Wi-Fi Management

Naming Convention

The Low Level Wi-FI API (LLECOM_WIFI), relies on functions that need to be implemented by engineers in a driver. The names of these functions match the `LLECOM_WIFI_IMPL_*` pattern.

Header Files

One header file is provided:

- LLECOM_WIFI_impl.h

  Defines the set of functions that the BSP must implement to manage and configure Wi-FI access points.

LLBLUETOOTH: Bluetooth

Naming Convention

The Low Level Bluetooth API (LLBLUETOOTH), relies on functions that need to be implemented by engineers in a driver. The names of these functions match the `LLBLUETOOTH_IMPL_*` pattern.

Header Files

One header file is provided:

- LLBLUETOOTH_impl.h

  Defines the set of functions that the BSP must implement to manage and configure and Bluetooth module.

LLEVENT: Event Queue

Naming Convention

The Low Level Event Queue API (LLEVENT), relies on functions that need to be implemented by engineers in a driver. The names of these functions match the `LLEVENT_IMPL_*` or `LLEVENT_*` pattern.

Header Files

Two header files are provided:

- LLEVENT_impl.h

  Defines the set of functions that the BSP must implement to manage, offer/handle events from the Event Queue.

- LLEVENT.h

  Defines the set of functions that the BSP must implement to use the Event Queue from the native side.

LLFS: File System

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.

LLWATCHDOG_TIMER: Watchdog Timer

Naming Convention

The Low Level Watchdog Timer API (LLWATCHDOG_TIMER), provides functions that allow the use of this API at the BSP level in C. The names of these functions match the `LLWATCHDOG_TIMER_IMPL_*` pattern.

The Watchdog API is delivered with a Generic C implementation on which the platform must depend. This implementation relies on functions that need to be implemented by engineers in a driver. The name of these functions match the `LLWATCHDOG_TIMER_IMPL_*_action` pattern.

Header Files

One C header file is provided:

- LLWATCHDOG_TIMER_impl.h

  Defines a set of functions that can be used at BSP level if required.

This C header file contains functions to implement:

- watchdog_timer_helper.h

  Defines a set of functions that the BSP must implement to link the platform watchdog timer to the Watchdog Timer library.

LLSEC: Security

Naming Convention

The Low Level Security API (LLSEC) provides functions that allow the use of this API at the BSP level in C. The names of these functions match the `LLSEC_*_IMPL_*` pattern.

Header Files

Several C header files are provided:

- LLSEC_CIPHER_impl.h

  Defines a set of functions that must be implemented by the BSP in order to decrypt and encrypt data using cryptographic ciphers.

- LLSEC_CONSTANTS.h

  Defines constants for certificates encoding formats.

- LLSEC_DIGEST_impl.h

  Defines a set of functions that must be implemented by the BSP in order to support message digest algorithms such as SHA-1 or SHA-256.

- LLSEC_ERRORS.h

  Defines the Security API error return codes.

- LLSEC_KEY_FACTORY_impl.h

  Defines a set of functions that must be implemented by the BSP in order to get keys informations such as algorithm or encoded form.

- LLSEC_KEY_PAIR_GENERATOR_impl.h

  Defines a set of functions that must be implemented by the BSP in order to generate private/public key pairs.

- LLSEC_MAC_impl.h

  Defines a set of functions that must be implemented by the BSP in order to support MAC algorithms.

- LLSEC_PRIVATE_KEY_impl.h

  Defines a set of functions that must be implemented by the BSP in order to encode private keys in DER format.

- LLSEC_PUBLIC_KEY_impl.h

  Defines a set of functions that must be implemented by the BSP in order to encode public keys.

- LLSEC_RANDOM_impl.h

  Defines a set of functions that must be implemented by the BSP in order to generate random data.

- LLSEC_SIG_impl.h

  Defines a set of functions that must be implemented by the BSP in order to support signatures functionalities.

- LLSEC_X509_CERT_impl.h

  Defines a set of functions that must be implemented by the BSP in order to manage X509 certificates operations like getting the public key, extracting the issuer, etc.

6.26.2 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 29: Generic Error Messages

Message ID	Description
1	Negative offset.
2	Negative length.
3	Offset + length > object length.

When an exception is thrown by the implementation of the EDC API, the error message

`EDC-1.2:E=<messageId>`

is issued, where `<messageId>` meaning is defined in the following table:

Table 30: 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.
7	Operation Reader.reset() is not supported.
8	String is empty.
9	Start index is out of range.
10	End index is out of range.
11	A throwable cannot suppress itself in Throwable.addSuppressed(Throwable exception).
12	Given exception is null in Throwable.addSuppressed(Throwable exception).

Exit Codes

The MicroEJ Application can stop its execution by calling the method System.exit(). To retrieve the appplication exit code (or exit status), use the C function `SNI_getExitCode()` after the end of `SNI_startVM()` (see `sni.h` header file). If the MicroEJ Application ended without calling System.exit() then `SNI_getExitCode()` returns `0`.

The error codes returned by `SNI_startVM()` are defined in the section *Error Codes*.

SNI

Error Messages

The following error messages are issued at runtime.

Table 31: [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 32: Feature definition file properties

Key	Usage	Description
entryPoint	Mandatory	The fully qualified name of the class that implements ej.kf.FeatureEntryPoint
immutables	Optional	Semicolon separated list of paths to *[BON]* immutable files owned by the Feature. `[BON]` immutable file is defined by a `/` separated path relative to application classpath
resources	Optional	Semicolon separated list of resource names owned by the Feature. Resource name is defined by Class.getResourceAsStream(String)
requiredTypes	Optional	Comma separated list of fully qualified names of required types. (Types that may be dynamically loaded using Class.forName()).
types	Optional	Comma separated list of fully qualified names of types owned by the Feature. A wildcard is allowed as terminal character to embed all types starting with the given qualified name (`a.b.C,x.y.*`)
version	Mandatory	String version, that can retrieved using ej.kf.Module.getVersion()

Kernel Definition Files

Kernel definition files are mandatory if one or more Feature definition file is loaded and are named `kernel.kf` and `kernel.cert` . `kernel.kf` must only define the `version` key. All types, resources and immutables are automatically owned by the Kernel if not explicitly set to be owned by a Feature.

Kernel API Files

Kernel API file definition is explained here: *Kernel API*.

Access Error Codes

When an instruction is executed that will break a *[KF] specification* 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 33: 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 34: 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

ECOM

Warning: This chapter describes the Foundation Library `ECOM-1.1`.

`ECOM-1.1` is discontinued since *Architecture 8.0.0*.

Error Messages

When an exception is thrown by the implementation of the ECOM API, the error message

`ECOM-1.1:E=<messageId>`

is issued, where `<messageId>` meaning is defined in the next table:

Table 35: 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 36: 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 37: 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 38: 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 39: SSL Error Messages

Message ID	Description
-2	Connection reset by the peer.
-3	Connection timed out.

continues on next page

Table 39 – continued from previous page

Message ID	Description
-5	Dispatch blocking request queue is full, and a new request cannot be added now.
-6	Certificate parsing error.
-7	The certificate data size bigger than the immortal buffer used to process certificate.
-8	No trusted certificate found.
-9	Basic constraints check failed: Intermediate certificate is not a CA certificate.
-10	Subject/issuer name chaining error.
-21	Wrong block type for RSA function.
-22	RSA buffer error: Output is too small, or input is too large.
-23	Output buffer is too small, or input is too large.
-24	Certificate AlogID setting error.
-25	Certificate public-key setting error.
-26	Certificate date validity setting error.
-27	Certificate subject name setting error.
-28	Certificate issuer name setting error.
-29	CA basic constraint setting error.
-30	Extensions setting error.
-31	Invalid ASN version number.
-32	ASN get int error: invalid data.
-33	ASN key init error: invalid input.
-34	Invalid ASN object id.
-35	Not null ASN tag.
-36	ASN parsing error: zero expected.
-37	ASN bit string error: wrong id.
-38	ASN OID error: unknown sum id.
-39	ASN date error: bad size.
-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.

continues on next page

Table 39 – continued from previous page

Message ID	Description
-71	AES-CCM Authentication check fail.
-80	Cavium Init type error.
-81	Bad alignment error, no alloc help.
-82	Bad ECC encrypt state operation.
-83	Bad padding: message wrong length.
-84	Certificate request attributes setting error.
-85	PKCS#7 error: mismatched OID value.
-86	PKCS#7 error: no matching recipient found.
-87	FIPS mode not allowed error.
-88	Name constraint error.
-89	Random Number Generator failed.
-90	FIPS Mode HMAC minimum key length error.
-91	RSA Padding error.
-92	Export public ECC key in ANSI format error: Output length only set.
-93	In Core Integrity check FIPS error.
-94	AES Known Answer Test check FIPS error.
-95	DES3 Known Answer Test check FIPS error.
-96	HMAC Known Answer Test check FIPS error.
-97	RSA Known Answer Test check FIPS error.
-98	DRBG Known Answer Test check FIPS error.
-99	DRBG Continuous Test FIPS error.
-100	AESGCM Known Answer Test check FIPS error.
-101	Process input state error.
-102	Bad index to key rounds.
-103	Out of memory.
-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.

continues on next page

Table 39 – continued from previous page

Message ID	Description
-129	Verify problem on certificate and check date/time on your device.
-130	Verify problem based on signature.
-131	PSK client identity error.
-132	PSK server hint error.
-133	PSK key callback error.
-134	Record layer length error.
-135	Can't decode peer key.
-136	The peer sent close notify alert.
-137	Wrong client/server type.
-138	The peer didn't send the certificate.
-140	NTRU key error.
-141	NTRU DRBG error.
-142	NTRU encrypt error.
-143	NTRU decrypt error.
-150	Bad ECC Curve Type or unsupported.
-151	Bad ECC Curve or unsupported.
-152	Bad ECC Peer Key.
-153	ECC Make Key failure.
-154	ECC Export Key failure.
-155	ECC DHE shared failure.
-157	Not a CA by basic constraint.
-159	Bad Certificate Manager error.
-160	OCSP Certificate revoked.
-161	CRL Certificate revoked.
-162	CRL missing, not loaded.
-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.

continues on next page

Table 39 – continued from previous page

Message ID	Description
-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.

6.26.3 Tools Options and Error Codes

Immutable Files Related Error Messages

The following error messages are issued at SOAR time (link phase) and not at runtime.

Table 40: 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 41: [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 42: 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 43: 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 44: 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 10: 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"/>
```

(continues on next page)

(continued from previous page)

```
</array>

<array name="VALUES">
    <elem value="2"/>
    <elem value="0"/>
    <elem value="1"/>
</array>
```

Table 45: Event Generators Static Definition

Tag	Attributes	Description
eventgenerators		The list of event generators.
	priority	*Optional.* An integer value. Defines the internal display thread priority. Default value is 5.
eventgenerator		Describes a generic event generator. See also *Generic Event Generators*.
	name	The logical name.
	class	The event generator class (must extend the ej.microui.event.generator.GenericEventGenerator class). This class must be available in the MicroEJ Application classpath.
	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.

continues on next page

Table 45 – continued from previous page

Tag	Attributes	Description
	name	The logical name.
	numbers	The logical name of the array which defines the number of state machines for this States generator, and their range of state values. The IDs of the state machines start at 0. The number of state machines managed by the States generator is equal to the size of the numbers array, and the value of each entry in the array is the number of different values supported for that state machine. State machine values for state machine i can be in the range 0 to numbers[i] -1.
	values	*Optional.* The logical name of the array which defines the initial state values of the state machines for this States generator. The values array must be the same size as the numbers array. If initial state values are specified using a values array, then the LLUI_INPUT_IMPL_getInitialStateValue function is not called; otherwise that function is used to establish the initial values[1]
	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 46: 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 47: 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 48: 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 49: 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>
```

continues on next page

Table 49 – 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 11: 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 50: 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.

6.26.4 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 51: Supported MicroEJ Core Engine Capabilities by MicroEJ Architecture Matrix

MicroEJ Core Engine Architectures		Capabilities		
MCU	Compiler	Mono- Sandbox	Tiny- Sandbox	Multi- Sandbox
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
ARMv7A	GCC	YES	YES	YES
ARMv7VE	GCC	YES	YES	YES
ESP32	ESP-IDF	YES	NO	YES

ARM Cortex-M0

Table 52: 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 53: ARM Cortex-M4 Compilers

Compiler	Build Version	Known Compatible Versions	Flags and Options	Module
Keil uVision	5.18.0.0	5.x	`--cpu Cortex-M4.fp --apcs=/hardfp --fpmode=ieee_no_fenv`	flopi4A20
GCC	4.8	4.x, 5.x, 6.x, 7.x, 8.x, 9.x	`-mabi=aapcs -mcpu=cortex-m4 -mlittle-endian -mfpu=fpv4-sp-d16 -mfloat-abi=hard -mthumb`	flopi4G25
IAR Embedded Workbench for ARM	8.32.1.18638	8.x, 9.x	`--cpu Cortex-M4F --fpu VFPv4_sp`	flopi4I35

Note: Since MicroEJ 4.0, Cortex-M4 architectures are compiled using `hardfp` convention call.

ARM Cortex-M7

Table 54: ARM Cortex-M7 Compilers

Compiler	Build Version	Known Compatible Versions	Flags and Options	Module
Keil uVision	5.18.0.0	5.x	`--cpu Cortex-M7.fp.sp --apcs=/hardfp --fpmode=ieee_no_fenv`	flopi7A21
GCC	4.8	4.x, 5.x, 6.x, 7.x, 8.x, 9.x	`-mabi=aapcs -mcpu=cortex-m7 -mlittle-endian -mfpu=fpv5-sp-d16 -mfloat-abi=hard -mthumbb`	flopi7G26
IAR Embedded Workbench for ARM	8.32.1.18638	8.x, 9.x	`--cpu Cortex-M7 --fpu VFPv5_sp`	flopi7I36

ARMv7A (ARMv7-A without integer division extension: Cortex-A5/Cortex-A8/Cortex-A9)

Table 55: ARMv7A Compilers

Compiler	Build Version	Known Compatible Versions	Flags and Options	Module
GCC	10.3	4.x, 5.x, 6.x, 7.x, 8.x, 9.x, 10.x	`-mabi=aapcs-linux -march=armv7-a -mlittle-endian -mfpu=vfp -mfloat-abi=hard -mthumb`	`oliveARMv7A_2`

ARMv7VE (ARMv7-A with integer division extension: Cortex-A7/Cortex-A15)

Table 56: ARMv7VE Compilers

Compiler	Build Version	Known Compatible Versions	Flags and Options	Module
GCC	10.3	4.x, 5.x, 6.x, 7.x, 8.x, 9.x, 10.x	`-mabi=aapcs-linux -march=armv7ve -mlittle-endian -mfpu=vfp -mfloat-abi=hard -mthumb`	`oliveARMv7VE_1`

ESP32

Table 57: Espressif ESP32 Compilers

Compiler	Version	Flags and Options	Module Name	Module Version
GCC (ESP-IDF)	5.2.0 (crosstool-ng-1.22.0-80-g6c4433a)	`-mlongcalls`	simikou1	Any
GCC (ESP-IDF)	5.2.0 (crosstool-ng-1.22.0-80-g6c4433a)	`-mlongcalls -mfix-esp32-psram-cache-issue`	simikou2	Up to `7.13.0` (included)
GCC (ESP-IDF)	5.2.0 (crosstool-ng-1.22.0-96-g2852398)	`-mlongcalls -mfix-esp32-psram-cache-issue`	simikou2	`7.12.2` or higher
GCC (ESP-IDF)	8.2.0 (crosstool-NG esp-2019r2)	`-mlongcalls`	simikou3	`7.16.0` or higher
GCC (ESP-IDF)	5.2.0 (crosstool-ng-1.22.0-97-gc752ad5)	`-mlongcalls -mfix-esp32-psram-cache-issue`	simikou4	`7.12.2` or higher
GCC (ESP-IDF)	8.4.0 (crosstool-NG esp-2021r1)	`-mlongcalls`	simikou5	`7.16.1` or higher
GCC (ESP-IDF)	8.4.0 (crosstool-NG esp-2021r1)	`-mlongcalls -mfix-esp32-psram-cache-issue -mfix-esp32-psram-cache-strategy=memw`	simikou6	`7.16.1` or higher
GCC (ESP-IDF)	11.2.0 (crosstool-NG esp-2022r1)	`-mlongcalls`	simikou7	`7.20.1` or higher

IAR Linker Specific Options

This section lists options that must be passed to IAR linker for correctly linking the MicroEJ object file (`microejapp.o`) generated by the SOAR.

`--no_range_reservations`

MicroEJ SOAR generates ELF absolute symbols to define some *Link-Time Option* (0 based values). By default, IAR linker allocates a 1 byte section on the fly, which may cause silent sections placement side effects or a section overlap error when multiple symbols are generated with the same absolute value:

```
Error[Lp023]: absolute placement (in [0x00000000-0x000000db]) overlaps with absolute symbol
[...]
```

The option `--no_range_reservations` tells IAR linker to manage an absolute symbol as described by the ELF specification.

`--diag_suppress=Lp029`

MicroEJ SOAR generates internal veneers that may be interpreted as illegal code by IAR linker, causing the following error:

```
Error[Lp029]: instruction validation failure in section "C:\xxx\microejapp.o[.text.
__icetea__virtual___1xxx#1126]": nested IT blocks. Code in wrong mode?
```

The option `--diag_suppress=Lp029` tells IAR linker to ignore instructions validation errors.

GNU LD Specific Options

`--start-group --end-group`

By default the GNU linker does not search unresolved symbols in previously loaded files and can cause undefined reference errors. To solve this issue, either change the load order of libraries (put `microejapp.o` first) or guard the libraries with the options `--start-group` and `--end-group` .

ARM Linker Specific Options

ARM linker (`armlink`) is the linker included in ARM Compiler and Keil MDK-ARM development tools.

Fix Unexpected Undefined Symbol

The ARM linker requires to resolve all symbols before detecting some that are not transitively required for linking the Executable. This typically happen when linking ELF object files containing dead code or debug functions that are compiled but not intended to be linked. If such functions refer to unresolved symbols, you may need to define a fake symbol to make the linker happy. You can declare it in your BSP project or directly in your VEE Port as following:

- Create a file `link/armlink-weak.lscf` in the *dropins* directory of your VEE Port configuration project.
- Edit the file and declare as many symbols as required. See also the *MicroEJ Linker* chapter for more details on the MicroEJ linker file syntax.

```
<lscFragment>
  <defSymbol name="[symbolName]" value="0" rootSymbol="true" weak="true"/>
</lscFragment>
```

The weak symbol(s) will be directly defined in the application object file (`microejapp.o`).

Link the SOAR Debug Section

When building an Application, the *SOAR* generates a dedicated ELF debug section named `.debug.soar` in the application object file (`microejapp.o`). This section is used by debug tools such as the *Stack Trace Reader* or the *Heap Dumper*. It is also used by the SOAR itself for *building Features* on a Kernel.

Unfortunately, the ARM linker does not link this section in the output ELF executable, even with debug mode enabled. If you try to load the raw executable produced by the ARM linker, the tools will fail with a *no debug section* error. Here is an example with the *Stack Trace Reader*:

```
=============== [ MicroEJ Core Engine Trace ] ===============
[INFO] Paste the MicroEJ core engine stack trace here.
1 : PROXY ERROR
  [M8] - The file XXX is not a valid image file or has no debug informations (can't read␣
↪file: XXX (no debug section)).
```

To be able to use debug tools, the debug section must be manually linked and injected in the Executable. This is done using the *SOAR debug infos post-linker tool*.

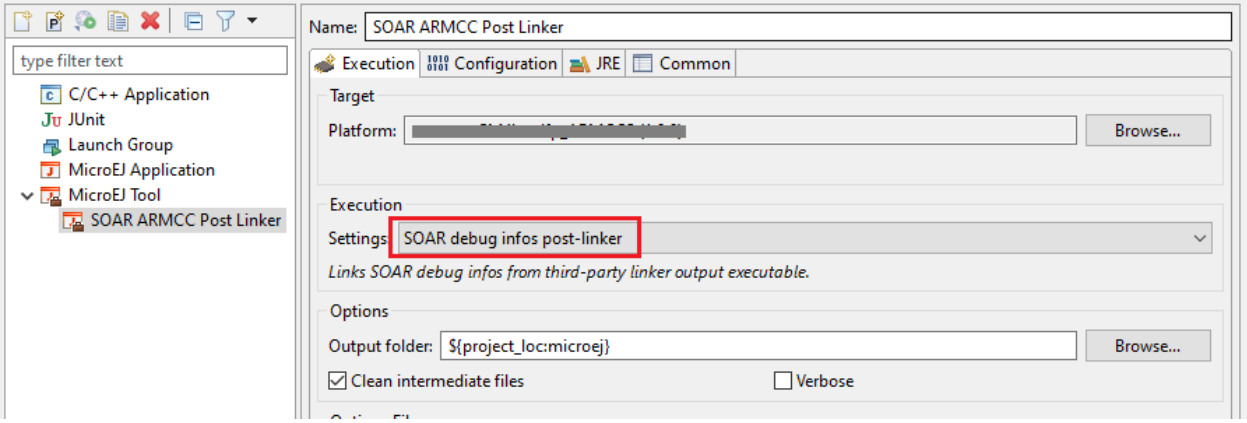


Fig. 85: SOAR debug infos post-linker tool Selection

This tool takes two file options:

- `soar.object.file` : the internal object file produced by the SOAR when building the Application. It can be found in the *Launch Output Folder* at `soar/[application_main_class].o` .
- `output.executable.file` : the Executable file produced by the ARM linker that includes the linked Application.

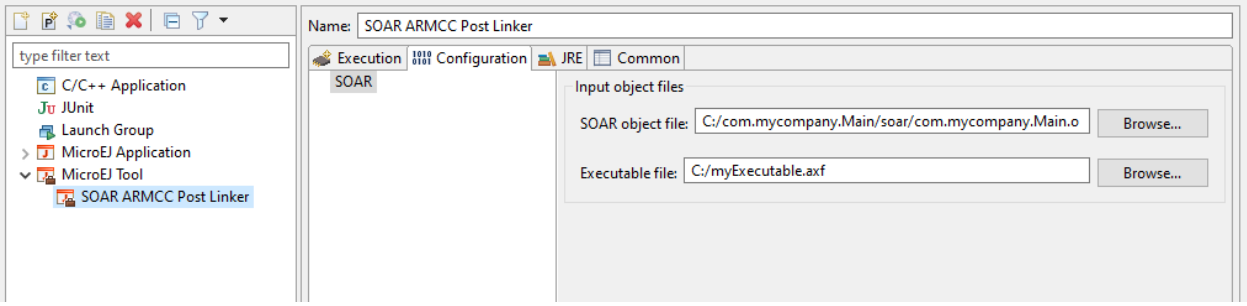


Fig. 86: SOAR debug infos post-linker tool Configuration

Once executed, it produces a new Executable file beside the original one with the `.microej` extension suffix

```
=============== [ SOARDebugInfosPostLinker ] ===============
Successfully generated c:\myExecutable.axf.microej.

SUCCESS
```

This file now contains the linked `.debug.soar` section so that it can be used by the debug tools.

6.26.5 Former Platform Migration

This chapter describes the steps to migrate a former MicroEJ Platform in its latest form described in *Platform Creation* chapter.

As a reminder, this new form brings two main features:

- Both Platform *build* and *dependencies declaration* are managed by *MicroEJ Module Manager*. This allows a fully automated build and continuous integration.
- The configuration of the target Board Support Package (BSP) has been revisited to support any *BSP Connection cases*.

Former MicroEJ Platforms were usually distributed by MicroEJ Corp. in an all-in-one ZIP file also called *fullPackaging*.

In this document, the MicroEJ Platform for STMicroelectronics STM32F746G-DISCO board will be used as an example.

The following figure shows the *fullPackaging* structure once extracted.

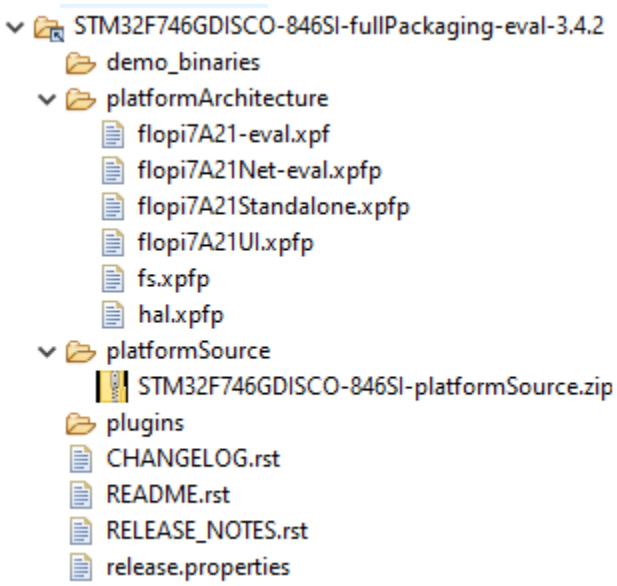


Fig. 87: STM32F746G-DISCO Platform Full Packaging Structure

The migration steps are:

1. Create a *Module Repository* to store the *MicroEJ Architecture* and *MicroEJ Packs* used by the Platform.
2. Import the Platform Configuration Additions into the Platform Configuration project.
3. Update the Front Panel project configuration.
4. Configure the *BSP Connection*.
5. Add the *Build Script* and *Run Script*.

Note: The migration of a Platform requires at least the version `5.4.0` of the SDK.

Create an Architecture Repository

The first step is to create an Architecture Repository containing the Architecture and Packs provided in the `platformArchitecture` directory of the *fullPackaging* package.

Note: If the Architecture and Packs used by the Platform are already stored in the module repositories provided by MicroEJ Corp (*Central Repository*, *Developer Repository*), or in your organization's repositories, then move to the next step.

By default, we provide the steps to extend the default *MicroEJ SDK settings file configuration* with local Architecture and Packs modules. The following steps can be adapted to custom *settings file*.

- Create a new empty project named `architecture-repository`
- Create a new file named `ivysettings.xml` with the following content and update the included settings file according to your MicroEJ SDK version (see *SDK Version*)

```
<?xml version="1.0" encoding="UTF-8"?>
<ivysettings>
  <property name="local.repo.url" value="${ivy.settings.dir}" override="false"/>

  <!--
      Include default settings file for MicroEJ SDK version:
      - MICROEJ SDK 5.4.0 or higher: ${user.home}/.microej/microej-ivysettings-5.4.xml
      - MICROEJ SDK 5.0.0 to 5.3.1: ${user.home}/.microej/microej-ivysettings-5.xml
      - MICROEJ SDK 4.1.x: ${user.home}/.ivy2/microej-ivysettings-4.1.xml
  -->
  <include file="${user.home}/.microej/microej-ivysettings-5.xml"/>

  <settings defaultResolver="ArchitectureResolver"/>

  <resolvers>
    <chain name="ArchitectureResolver">
      <filesystem m2compatible="true">
        <artifact pattern="${local.repo.url}/${microej.artifact.pattern}" />
        <ivy pattern="${local.repo.url}/${microej.ivy.pattern}" />
      </filesystem>
      <resolver ref="${microej.default.resolver}"/>
    </chain>
  </resolvers>
</ivysettings>
```

- Copy the Architecture file (`.xpf`) into the correct directory following its *naming convention*).
  - Open or extract the Architecture file (`.xpf`)
  - Open the `release.properties` file to retrieve the naming convention mapping:
    * `architecture` is the `ISA` (e.g. `CM7`)
    * `toolchain` is the `TOOLCHAIN` (e.g. `CM7hardfp_ARMCC5`)
    * `name` is the `UID` (e.g. `flopi7A21`)
    * `version` is the `VERSION` (e.g. `7.11.0`)

  For example, in the STM32F746G-DISCO Platform, the Architecture file `flopi7A21-eval.xpf` shall be copied and renamed to `architecture-repository/com/microej/architecture/CM7/CM7hardfp_ARMCC5/flopi7A21/7.11.0/flopi7A21-7.11.0-eval.xpf` .
- Copy the Architecture Specific Packs files (`.xpfp`) into the correct directory following MicroEJ Naming Convention (see *Pack Import*) with the exception of the Standalone pack that should not be imported (e.g. named `flopi7A21Standalone.xpfp`).
  - Open or extract the Architecture Specific Pack (`.xpfp`).

    Note: The Architecture Specific Packs have the `UID` of the Architecture in their name (e.g. `flopi7A21UI.xpfp`) and their `release_pack.properties` file contains the information of the Architecture.

  - Open the `release_pack.properties` file to retrieve the naming convention mapping:
    * `architecture` is the `ISA` (e.g. `CM7`)
    * `toolchain` is the `TOOLCHAIN` (e.g. `CM7hardfp_ARMCC5`)

    * `name` is the `UID` (e.g. `flopi7A21`)
    * `packName` is the `NAME` (e.g. `ui`)
    * `packVersion` is the `VERSION` (e.g. `12.0.1`)

  For example, in the STM32F746G-DISCO Platform, the Architecture Specific Pack UI `flopi7A21UI.xpfp` shall be copied and renamed to `architecture-repository/com/microej/architecture/CM7/CM7hardfp_ARMCC5/flopi7A21-ui-pack/12.0.1/flopi7A21-ui-pack-12.0.1.xpfp`.

- Copy the Legacy Generic Packs (`.xpfp` files) into the correct directory following MicroEJ Naming Convention (see *Pack Import*).
  - Open or extract the Generic Pack (`.xpfp`).

    Note: The `release_pack.properties` of Legacy Generic Packs does not contain information about Architecture.

  - Open the `release_pack.properties` file:
    * `packName` is the `NAME` (e.g. `fs`)
    * `packVersion` is the `VERSION` (e.g. `4.0.2`)

  For example, in the STM32F746G-DISCO Platform, the Legacy Generic Pack FS `fs.xpfp` shall be copied and renamed to `architecture-repository/com/microej/pack/fs/4.0.2/fs-4.0.2.xpfp`.

- Configure MicroEJ Module Manager to use the Architecture Repository:
  - Go to Window > Preferences > MicroEJ > Module Manager
  - In Module Repository set Settings File: to `${workspace_loc:architecture-repository/ivysettings.xml}`.
  - Apply and Close

Here is the layout of the Architecture Repository for the STM32F746G-DISCO Platform.

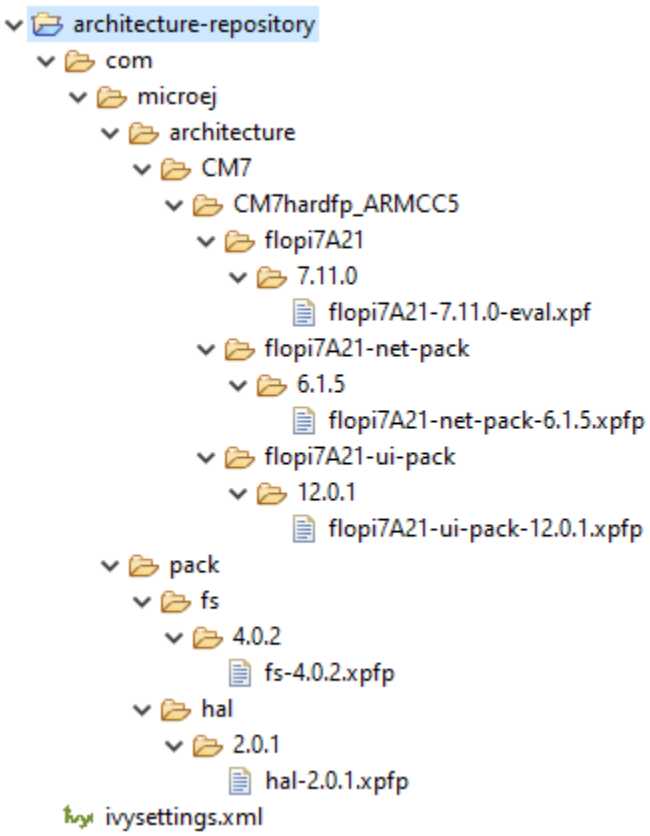
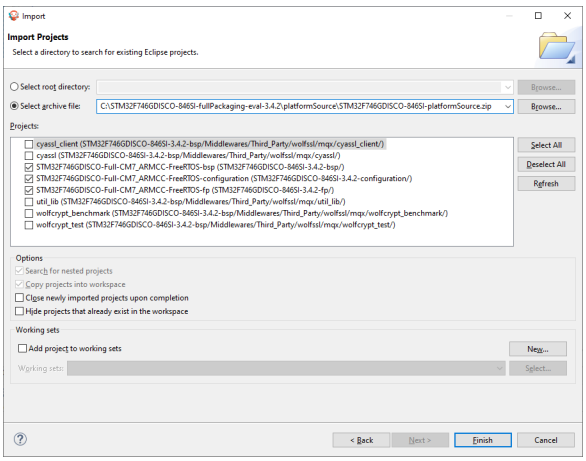


Fig. 88: Architecture Repository for STM32F746G-DISCO *fullPackaging*

Import the Former Platform Sources

- Go to File > Import... > General > Existing Projects into Workspace .
- Browse to the archive file that contains the platform sources, like in the example below.



- Select the `-configuration` , `-fp` and `-bsp` projects prefixed with the Platform name (e.g., `STM32F746GDISCO-Full-CM7_ARMCC-FreeRTOS`).

- Click `Finish`.

Install the Platform Configuration Additions

- Rename the file `bsp.properties` located in the Platform Configuration Project to `bsp2.properties` (save it for later).
- Install Platform Configuration Additions, by following instructions described at https://github.com/MicroEJ/VEEPortQualificationTools/blob/master/framework/platform/README.rst.
  Files within the `content` folder have to be copied to the `-configuration` project (e.g. `STM32F746GDISCO-Full-CM7_ARMCC-FreeRTOS-configuration`).
- Edit the `module.properties` file and set `com.microej.platformbuilder.platform.filename` to the name of the platform configuration file (e.g. `STM32F746GDISCO.platform`).
- Update the default name of the Platform module in the `module.ivy`. Replace with `<info organisation="com.microej.platform.st.stm32f746g-disco" module="Platform" status="integration" revision="1.0.0">`.
- Update the `module.ivy` with the Architecture and Packs dependencies.

Here is the module dependencies declared for the STM32F746G-DISCO Platform.

Listing 12: STM32F746GDISCO-Full-CM7_ARMCC-FreeRTOS-configuration/module.ivy

```
<dependencies>
  <!-- MicroEJ Architecture -->
  <dependency org="com.microej.architecture.CM7.CM7hardfp_ARMCC5" name="flopi7A21" rev="7.11.
↪0">
    <artifact name="flopi7A21" m:classifier="${com.microej.platformbuilder.architecture.
↪usage}" ext="xpf"/>
  </dependency>

  <!-- MicroEJ Architecture Specific Packs  -->
  <dependency org="com.microej.architecture.CM7.CM7hardfp_ARMCC5" name="flopi7A21-ui-pack"␣
↪rev="12.0.1">
    <artifact name="flopi7A21-ui-pack" ext="xpfp"/>
  </dependency>
  <dependency org="com.microej.architecture.CM7.CM7hardfp_ARMCC5" name="flopi7A21-net-pack"␣
↪rev="6.1.5">
    <artifact name="flopi7A21-net-pack" ext="xpfp"/>
  </dependency>

  <!-- Legacy MicroEJ Generic Packs -->
  <dependency org="com.microej.pack" name="fs" rev="4.0.2">
    <artifact name="fs" ext="xpfp"/>
  </dependency>
  <dependency org="com.microej.pack" name="hal" rev="2.0.1">
    <artifact name="hal" ext="xpfp"/>
  </dependency>

</dependencies>
```

Update the Front Panel Configuration

- In `-configuration/frontpanel/frontpanel.properties` set the `project.name` to the folder name that contains the front-panel (e.g. `project.name=STM32F746GDISCO-Full-CM7_ARMCC-FreeRTOS-fp`).

At this state, the Platform is not connected to the BSP yet, but you can check that everything is properly configured so far by building it:

- Right-click on the `-configuration` project and select Build Module
- Import the Platform built into the workspace by following the instructions available at the end of the build logs (see logs example below).

```
module-platform:report:
    [echo]    ␣
↪==========================================================================================================
    [echo]     Platform has been built in this directory 'C:\STM32F746GDISCO-Platform-
↪CM7hardfp_ARMCC5-0.1.0'.
    [echo]     To import this project in your MicroEJ SDK workspace (if not already␣
↪available):
    [echo]      - Select 'File' > 'Import...' > 'General' > 'Existing Projects into Workspace
↪' > 'Next'
    [echo]      - Check 'Select root directory' and browse 'C:\STM32F746GDISCO-Platform-
↪CM7hardfp_ARMCC5-0.1.0' > 'Finish'
    [echo]    ␣
↪==========================================================================================================
```

At this stage the Platform is built and imported in the workspace, so you can create a Standalone Application and run it on the Simulator (see *Create a MicroEJ Standalone Application*).

Note: If the build failed, it might be because the Architecture and Packs can not be retrieved from the Architecture Repository. Ensure that the Architecture Repository is correctly configured and that it contains the required artifacts (as described in *the first step*).

Configure the BSP Connection

This section explains how to configure a full BSP Connection on the STM32F746G-DISCO Platform. See *BSP Connection* for more information.

- Open `-configuration/bsp/bsp.properties` .
- Comment out and set the following variables:
  - `root.dir`
  - `microejapp.relative.dir`
  - `microejlib.relative.dir`
  - `microejinc.relative.dir`
  - `microejscript.relative.dir`

  For example:

```
# Specify the MicroEJ Application file ('microejapp.o') parent directory.
# This is a '/' separated directory relative to 'bsp.root.dir'.
microejapp.relative.dir=Projects/STM32746G-Discovery/Applications/MicroEJ/platform/lib

# Specify the MicroEJ Platform runtime file ('microejruntime.a') parent directory.
# This is a '/' separated directory relative to 'bsp.root.dir'.
microejlib.relative.dir=Projects/STM32746G-Discovery/Applications/MicroEJ/platform/lib

# Specify MicroEJ Platform header files ('*.h') parent directory.
# This is a '/' separated directory relative to 'bsp.root.dir'.
microejinc.relative.dir=Projects/STM32746G-Discovery/Applications/MicroEJ/platform/inc

# Specify BSP external scripts files ('build.bat' and 'run.bat') parent directory.
# This is a '/' separated directory relative to 'bsp.root.dir'.
microejscript.relative.dir=Projects/STM32746G-Discovery/Applications/MicroEJ/scripts

# Specify the BSP root directory. Can use ${project.parent.dir} which target the parent␣
↪of platform configuration project
# For example, '${project.parent.dir}/PROJECT-NAME-bsp' specifies a BSP project beside␣
↪the '-configuration' project
root.dir=${project.parent.dir}/STM32F746GDISCO-Full-CM7_ARMCC-FreeRTOS-bsp/
```

The paths to `microejXXX.relative.dir` can be inferred by looking at the `output.dir` value in `bsp2.properties` saved earlier. For example on the STM32F746G-DISCO project, its value is `${workspace}/${project.prefix}-bsp/Projects/STM32746G-Discovery/Applications/MicroEJ/platform`.

- The BSP project path `${workspace}/${project.prefix}-bsp` becomes `${project.parent.dir}/STM32F746GDISCO-Full-CM7_ARMCC-FreeRTOS-bsp/`.
- `Projects/STM32746G-Discovery/Applications/MicroEJ/platform` is the path to the Application file, Platform header and runtime files. MicroEJ convention is to put the Application file and Platform runtime files to `platform/lib/` and the Platform header files to `platform/inc/`.
- *Build Script File* and *Run Script File* are PCA-specific and did not exist before. By convention we put them in a `scripts/` directory.

The paths to `microejXXX.relative.dir` can be also be checked by looking at the C TOOLCHAIN configuration of the BSP. For example on the STM32F746G-DISCO project, the BSP configuration is located at `STM32F746GDISCO-Full-CM7_ARMCC-FreeRTOS-bsp/Projects/STM32746G-Discovery/Applications/MicroEJ/MDK-ARM/Project.uvprojx`.

- In Project > Options for Target 'standalone'... > C/C++ > Include Paths contains `../platform/inc`. This corresponds to the `microejinc.relative.dir` relative the TOOLCHAIN project's file.
- In the Project pane, there is a folder `MicroEJ/Libs` that contains `microejruntime.lib` and `microejapp.o`.
  - Right-click on `microejruntime.lib` > Options for File 'XXX'... . The Path is `../platform/lib/microejruntime.lib`. This corresponds to the `microejlib.relative.dir`.
  - Right-click on `microejapp.o` > Options for File 'XXX'... . The Path is `../platform/lib/microejapp.o`. This corresponds to the `microejapp.relative.dir`.
- Rebuild the platform (Right-click on the `-configuration` project and select Build Module)

At this stage the Platform is connected to the BSP so you can build and program a Firmware (see *Run on the Device*).

Add the Build and Run Scripts

The final stage consists of adding the *Build Script*, to automate the build of a Firmware, and the *Run Script*, to automate the programming of a MicroEJ Firmware onto the device.

The Platform Qualification Tools provides examples of Build Script and Run Script for various C TOOLCHAIN here. *This tutorial* also describes the steps to create and use these scripts.

On the STM32F746G-DISCO, the C TOOLCHAIN used is Keil uVision.

- Create the directory pointed by `microejscript.relative.dir` (e.g. `STM32F746GDISCO-Full-CM7_ARMCC-FreeRTOS-bsp\Projects\STM32746G-Discovery\Applications\MicroEJ\scripts`).
- Copy the example scripts from the Platform Qualification Tools for the C TOOLCHAIN of the BSP (e.g. `PlatformQualificationTools/framework/platform/scripts/KEILuV5/`)
- Configure the scripts. Refer to the documentation in the scripts comments for this step.
- Enable the execution of the build script:
  - Go to Run > Run Configurations…
  - Select the launch configuration
  - Go to Configuration > Device > Deploy
  - Ensure Execute the MicroEJ build script (build.bat) at a location known by the 3rd-party BSP project. is checked.

Use the Platform in Module Projects

Module projects may require the Platform, for example to build an Application or to run a Test Suite. One way of selecting the Platform in a module project is to declare it as a module dependency (see *Platform Selection*).

In case a former Platform is loaded this way in your existing module projects, the dependency has to be updated. In this example, the Platform would now be selected like this:

```
<dependency org="com.microej.platform.st.stm32f746g-disco" name="Platform" rev="1.
↪0.0" conf="platform->default" transitive="false"/>
```

This also requires that your module projects use a compatible version of the associated build type (the build type relates to the *Module Natures*). As stated before, loading a Platform in its latest form requires at least the version `5.4.0` of the SDK. Therefore, make sure to use versions of the build types that come with the SDK `5.4.0` and above. Here is a brief summary of the minimum version for the most common module natures:

- *Add-On Library*: build type `com.is2t.easyant.buildtypes#build-microej-javalib` version `5.0.0` and above.
- *Standalone Application*: build type `com.is2t.easyant.buildtypes#build-firmware-singleapp` version `1.4.0` and above.
- *Sandboxed Application*: build type `com.is2t.easyant.buildtypes#build-application` version `8.0.0` and above.

Going further

Now that the Platform is connected to the BSP it can leverage the Java Test Suites provided by the Platform Qualification Tools. See *Run a Test Suite on a Device* for a step by step explanation on how to do so.

6.26.6 Architecture 8.0.0 Migration

This chapter describes the steps to migrate a VEE Port from Architecture `8.0.0` to Architecture `8.1.0`.

As a reminder, refer to the *Architecture 8.1.0 Changelog* section for the complete list of changes and updates.

Migrate Core Engine Capability Configuration

The selection of the *Core Engine capability* is now done via the property `com.microej.runtime.capability`. Refer to one of the sections below depending on your desired capability.

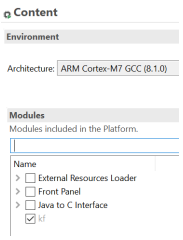
If you use the property `com.microej.platformbuilder.module.multi.enabled`, update your Platform Configuration Additions to the version `2.1.0` or higher. It is also recommended to delete the property `com.microej.platformbuilder.module.multi.enabled` and to use the property `com.microej.runtime.capability` instead.

Mono-Sandbox

Mono-Sandbox remains the default capability and no changes are required to your VEE Port configuration.

Multi-Sandbox

In the Platform Editor, the `Multi Applications` (`kf`) module now appears in gray:



Unselect the `kf` module and follow the instructions from the *Multi-Sandbox installation* section.

Tiny-Sandbox

The property `mjvm.standalone.configuration` used to select the Tiny-Sandbox capability is now deprecated. It is recommended to remove the definition of this property from the `configuration.xml` file and follow the instructions from the *Tiny-Sandbox installation* section.

Migrate Your LLKERNEL Implementation

This section only applies if your LLKERNEL was based on legacy *In-Place Installation mode*. The Kernel Working Buffer no longer exists. The functions `LLKERNEL_IMPL_allocateWorkingBuffer()` and `LLKERNEL_IMPL_freeWorkingBuffer()` are no more called and can be simply removed from your implementation.

Memory allocation for the Features will now use the function `LLKERNEL_IMPL_allocateFeature()`. The following code is a `LLKERNEL_impl.c` template for migrating your current implementation using this API. The code logic based on a `malloc/free` implementation does not need to be changed.

```
#include <stdlib.h>
#include <string.h>

#include "LLKERNEL_impl.h"

// Your implementation of malloc()
#define KERNEL_MALLOC(size) malloc((size_t)(size))

// Your implementation of free()
#define KERNEL_FREE(addr) free((void*)(addr))

// Your implementation of 'ASSERT(0)'
#define KERNEL_ASSERT_FAIL() while(1)

// Utility macros for allocating RAM and ROM areas with required alignment constraints
#define KERNEL_AREA_GET_MAX_SIZE(size, alignment) ((size)+((alignment)-1))
#define KERNEL_AREA_GET_START_ADDRESS(addr, alignment) ((void*)((((int32_
↪t)(addr))+(alignment)-1)&~((alignment)-1)))

typedef struct installed_feature{
   void* ROM_area;
   void* RAM_area;
} installed_feature_t;

int32_t LLKERNEL_IMPL_allocateFeature(int32_t size_ROM, int32_t size_RAM) {
   int32_t ret = 0;
   int total_size = sizeof(installed_feature_t);
   total_size += KERNEL_AREA_GET_MAX_SIZE(size_ROM, LLKERNEL_ROM_AREA_ALIGNMENT);
   total_size += KERNEL_AREA_GET_MAX_SIZE(size_RAM, LLKERNEL_RAM_AREA_ALIGNMENT);

   void* total_area = KERNEL_MALLOC(total_size);
   if(NULL != total_area){
      installed_feature_t* f = (installed_feature_t*)total_area;
      f->ROM_area = KERNEL_AREA_GET_START_ADDRESS((void*)(((int32_t)f)+((int32_
↪t)sizeof(installed_feature_t))), LLKERNEL_ROM_AREA_ALIGNMENT);
      f->RAM_area = KERNEL_AREA_GET_START_ADDRESS((void*)(((int32_t)f->ROM_area)+size_ROM),␣
↪LLKERNEL_RAM_AREA_ALIGNMENT);
      ret = (int32_t)f;
   } // else out of memory

   return ret;
}
```

(continues on next page)

(continued from previous page)

```
void LLKERNEL_IMPL_freeFeature(int32_t handle) {
   KERNEL_FREE(handle);
}

int32_t LLKERNEL_IMPL_getAllocatedFeaturesCount(void) {
   // No persistency support
   return 0;
}

int32_t LLKERNEL_IMPL_getFeatureHandle(int32_t allocation_index) {
   // No persistency support
   KERNEL_ASSERT_FAIL();
}

void* LLKERNEL_IMPL_getFeatureAddressRAM(int32_t handle) {
   return ((installed_feature_t*)handle)->RAM_area;
}

void* LLKERNEL_IMPL_getFeatureAddressROM(int32_t handle) {
   return ((installed_feature_t*)handle)->ROM_area;
}

int32_t LLKERNEL_IMPL_copyToROM(void* dest_address_ROM, void* src_address, int32_t size) {
   memcpy(dest_address_ROM, src_address, size);
   return LLKERNEL_OK;
}

int32_t LLKERNEL_IMPL_flushCopyToROM(void) {
   return LLKERNEL_OK;
}

int32_t LLKERNEL_IMPL_onFeatureInitializationError(int32_t handle, int32_t error_code) {
   // No persistency support
   KERNEL_ASSERT_FAIL();
   return 0;
}
```

6.26.7 Architecture 7.x Migration

This chapter describes the steps to migrate a VEE Port from Architecture `7.x` to Architecture `8.0.0`.

As a reminder, refer to the *Architecture 8.0.0 Changelog* section for the complete list of changes and updates.

Update Platform Configuration Additions

Architecture `8.0.0` now directly integrates the *BSP Connection* mechanism. Consequently, Platform Configuration Additions files have been separated in two directories:

- `content-sdk-5` : files required for building the VEE Port using SDK 5.x (MMM)
- `content-architecture-7` : files required for building the Executable using Architecture 7.x.

See https://github.com/MicroEJ/VEEPortQualificationTools/blob/master/framework/platform/README.rst for more details.

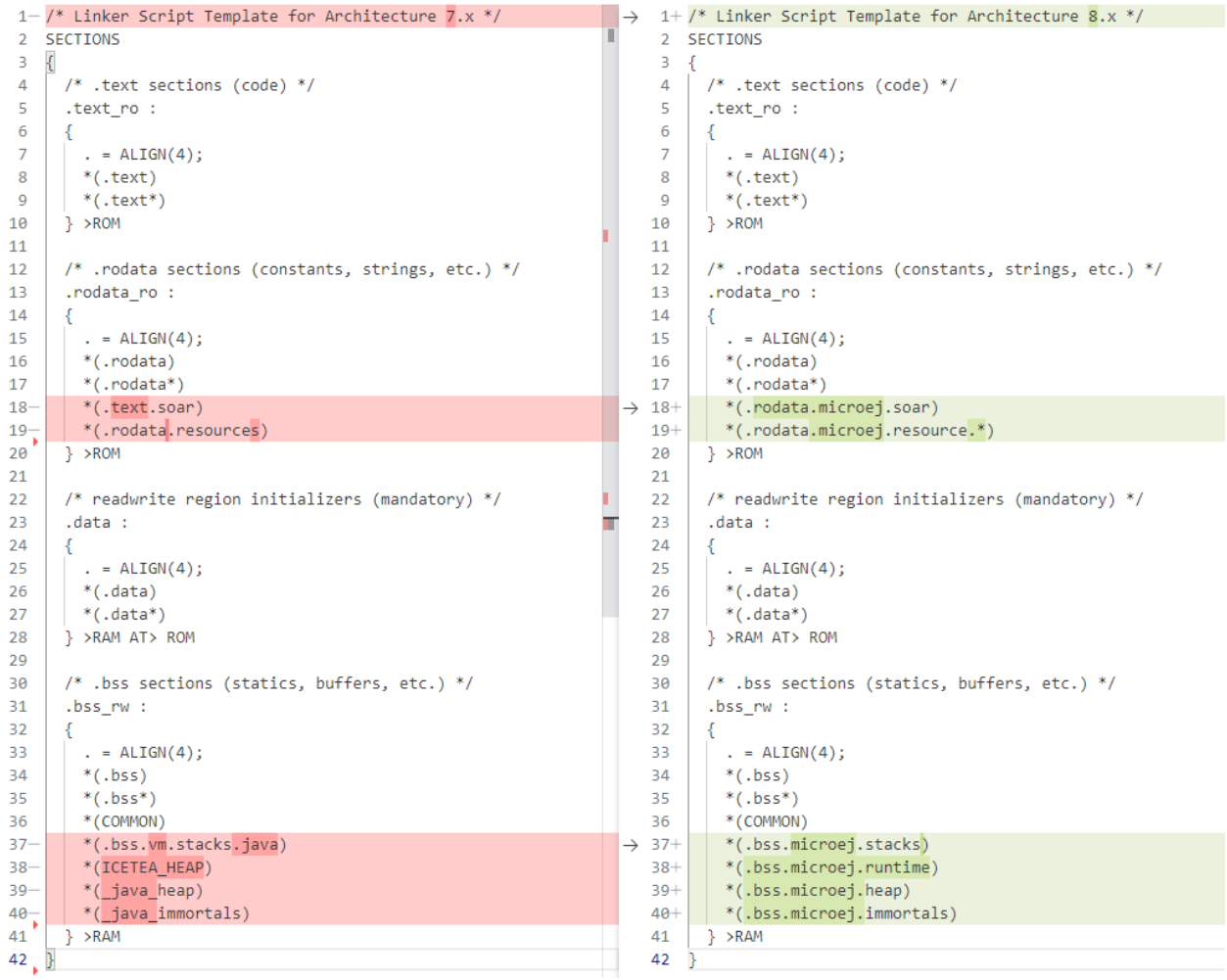
Your VEE Port must be updated to remove files that are now included in Architecture 8:

- Delete `[name]-configuration/build/module/module-dropins` directory.
- Delete `[name]-configuration/build/module/module-dropins.ant` file.
- Delete `[name]-configuration/build/platform/platform-deploy.ant` file.
- Delete `[name]-configuration/build/platform/platform-kf.ant` file.
- Download the latest `content-sdk-5` directory. Your local files must be overridden.
- Edit your *module.ivy* and put back your module *name*, *version*, *organisation* and <*dependencies*> content.
- Edit your *module.properties* and put back your options (if they have changed from default ones).
- Delete the following files from your `[name]-configuration/dropins` directory:
  - `scripts/init-bsp/*`
  - `scripts/init-license-checker/*`
  - `scripts/checkOS.xml`
  - `scripts/deployInBSP.xml`
  - `scripts/deployInBSPCommon.xml`
  - `scripts/deployToolBSPRun*`
  - `scripts/fullLink*`
  - `tools/license-checker.jar`
  - `workbenchExtension-launchScriptFramework.jar`
- Rebuild your VEE Port.
- Rebuild your Executable.

Update BSP with new Sections Names

The Core Engine sections have been renamed to respect the standard ELF convention. See *Core Engine Link* section for further details.

All references to section names in your BSP must be updated. This is usually only used in your linker script file, but section names are sometimes also hardcoded in the C Code. Here is an example of a GNU LD script highlighting the typical changes that must be made:



```
/* Linker Script Template for Architecture 7.x */
SECTIONS
{
  /* .text sections (code) */
  .text_ro :
  {
    . = ALIGN(4);
    *(.text)
    *(.text*)
  } >ROM

  /* .rodata sections (constants, strings, etc.) */
  .rodata_ro :
  {
    . = ALIGN(4);
    *(.rodata)
    *(.rodata*)
    *(.text.soar)
    *(.rodata.resources)
  } >ROM

  /* readwrite region initializers (mandatory) */
  .data :
  {
    . = ALIGN(4);
    *(.data)
    *(.data*)
  } >RAM AT> ROM

  /* .bss sections (statics, buffers, etc.) */
  .bss_rw :
  {
    . = ALIGN(4);
    *(.bss)
    *(.bss*)
    *(COMMON)
    *(.bss.vm.stacks.java)
    *(ICETEA_HEAP)
    *(_java_heap)
    *(_java_immortals)
  } >RAM
}
```

```
/* Linker Script Template for Architecture 8.x */
SECTIONS
{
  /* .text sections (code) */
  .text_ro :
  {
    . = ALIGN(4);
    *(.text)
    *(.text*)
  } >ROM

  /* .rodata sections (constants, strings, etc.) */
  .rodata_ro :
  {
    . = ALIGN(4);
    *(.rodata)
    *(.rodata*)
    *(.rodata.microej.soar)
    *(.rodata.microej.resource.*)
  } >ROM

  /* readwrite region initializers (mandatory) */
  .data :
  {
    . = ALIGN(4);
    *(.data)
    *(.data*)
  } >RAM AT> ROM

  /* .bss sections (statics, buffers, etc.) */
  .bss_rw :
  {
    . = ALIGN(4);
    *(.bss)
    *(.bss*)
    *(COMMON)
    *(.bss.microej.stacks)
    *(.bss.microej.runtime)
    *(.bss.microej.heap)
    *(.bss.microej.immortals)
  } >RAM
}
```

Fig. 89: Example of LD Script File Migration

Remove LLBSP_IMPL_isInReadOnlyMemory

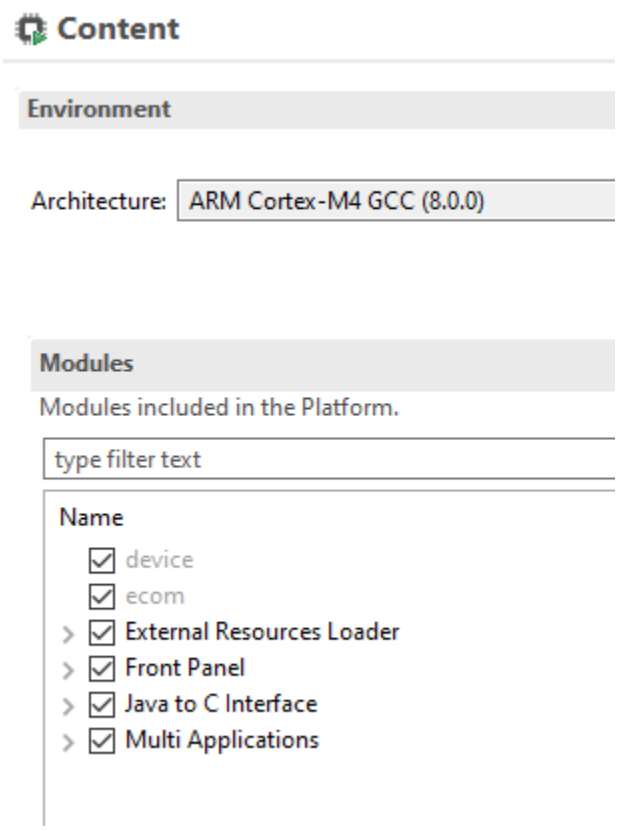
The `LLBSP_IMPL_isInReadOnlyMemory` function has been removed since it is no more called by the Core Engine. You can simply remove your implementation function.

Migrate Built-in Modules

The following built-in legacy modules have been removed from the Architecture:

- Device
- ECOM-COMM

In the Platform Editor, these modules now appear in gray with Architecture `8.x` :



To remove these modules, open the `.platform` file using a text editor and remove the following XML elements:

```
<group name="device"/>
<group name="ecom"/>
```

Migrate Device Module

The latest Device Pack available on the *Central Repository* is backward compatible with the built-in Architecture module.

The following dependency must be added to the *module.ivy* of the VEE Port configuration project:

```
<dependency org="com.microej.pack.device" name="device-pack" rev="1.1.1" />
```

Migrate ECOM-COMM Module

The Foundation Library `ECOM-COMM-1.1` has been removed from *Architecture 8.0.0*. It is now replaced by `ECOM-COMM-2.0` which is distributed in its own Pack.

There are two migration options:

- either migrate to the latest `ECOM-COMM-2.0` Pack,
- or integrate the legacy `ECOM-COMM-1.1` Pack files as-is into your VEE Port `dropins` directory.

Contact our support team to get the best migration strategy and detailed instructions.

Migrate Your LLKERNEL Implementation

The following code is a `LLKERNEL_impl.c` template for migrating your current implementation of Feature installation in RAM. This is now called *In-Place Installation*. Your code logic for managing allocated blocks does not need to be changed. As there is no installation in ROM, most of the new functions do not need to be implemented.

```
#include "LLKERNEL_impl.h"

void* LLKERNEL_IMPL_allocateWorkingBuffer(int32_t size) {
   // Paste here the code of your former 'LLKERNEL_IMPL_allocate' function
}

void LLKERNEL_IMPL_freeWorkingBuffer(void* chunk_address) {
   // Paste here the code of your former 'LLKERNEL_IMPL_free' function
}

int32_t LLKERNEL_IMPL_allocateFeature(int32_t size_ROM, int32_t size_RAM) {
   return 0;
}

int32_t LLKERNEL_IMPL_getAllocatedFeaturesCount(void) {
   return 0;
}

void LLKERNEL_IMPL_freeFeature(int32_t handle) {
   // Paste here your implementation of 'ASSERT(0)'
}

int32_t LLKERNEL_IMPL_getFeatureHandle(int32_t allocation_index) {
   // Paste here your implementation of 'ASSERT(0)'
   return 0;
}

void* LLKERNEL_IMPL_getFeatureAddressRAM(int32_t handle) {
   // Paste here your implementation of 'ASSERT(0)'
   return 0;
}

void* LLKERNEL_IMPL_getFeatureAddressROM(int32_t handle) {
   // Paste here your implementation of 'ASSERT(0)'
   return 0;
}

int32_t LLKERNEL_IMPL_copyToROM(void* dest_address_ROM, void* src_address, int32_t size) {
   // Paste here your implementation of 'ASSERT(0)'
   return 0;
}

int32_t LLKERNEL_IMPL_flushCopyToROM(void) {
   // Paste here your implementation of 'ASSERT(0)'
   return 0;
}
```

(continues on next page)

(continued from previous page)

```
int32_t LLKERNEL_IMPL_onFeatureInitializationError(int32_t handle, int32_t error_code) {
   // Paste here your implementation of 'ASSERT(0)'
   return 0;
}
```

Migrate Trace C Library Usage

In Architecture `8.0.0`, the `Trace` C library's version has been updated from `1.0.0` to `2.0.0`. This new version introduces the following backward incompatible changes:

- C header file `trace.h` has been renamed into `LLTRACE.h`.
- The functions declared in this header have been renamed from `TRACE_xxx` to `LLTRACE_xxx`.

If you have included `trace.h` in a C file, the compilation will fail with an error message similar to one of the following messages:

- `fatal error: trace.h: No such file or directory`
- `Fatal Error[Pe1696]: cannot open source file "trace.h"`

To fix this issue, you can either migrate to version `2.0.0` of the `Trace` library or provide a backward compatibility layer.

To migrate to version `2.0.0`, you need to make the following changes:

- Replace the directives `#include "trace.h"` with `#include "LLTRACE.h"`.
- Replace any references to the `TRACE_xxx` functions (e.g., `TRACE_record_event_void`) with references to the corresponding `LLTRACE_xxx` function (e.g., `LLTRACE_record_event_void`).

If you decide not to modify existing code, you can create and add to your project a `trace.h` file with the following content:

```
#ifndef TRACE_H
#define TRACE_H

/**
 * Trace library API backward compatibility layer.
 * Allows to use Trace API 1.0.0 (Architecture 7.x) in a VEE Port
 * that includes Trace API 2.0.0 (Architecture 8.x).
 */

#include "LLTRACE.h"

#ifdef __cplusplus
        extern "C" {
#endif

#define TRACE_start LLTRACE_start
#define TRACE_start LLTRACE_start
#define TRACE_stop LLTRACE_stop
#define TRACE_is_started LLTRACE_is_started
#define TRACE_declare_event_group LLTRACE_declare_event_group
#define TRACE_record_event_void LLTRACE_record_event_void
#define TRACE_record_event_u32 LLTRACE_record_event_u32
```

(continues on next page)

(continued from previous page)

```
#define TRACE_record_event_u32x2 LLTRACE_record_event_u32x2
#define TRACE_record_event_u32x3 LLTRACE_record_event_u32x3
#define TRACE_record_event_u32x4 LLTRACE_record_event_u32x4
#define TRACE_record_event_u32x5 LLTRACE_record_event_u32x5
#define TRACE_record_event_u32x6 LLTRACE_record_event_u32x6
#define TRACE_record_event_u32x7 LLTRACE_record_event_u32x7
#define TRACE_record_event_u32x8 LLTRACE_record_event_u32x8
#define TRACE_record_event_u32x9 LLTRACE_record_event_u32x9
#define TRACE_record_event_u32x10 LLTRACE_record_event_u32x10
#define TRACE_record_event_end LLTRACE_record_event_end
#define TRACE_record_event_end_u32 LLTRACE_record_event_end_u32

#ifdef __cplusplus
        }
#endif

#endif //TRACE_H
```

Migrate Legacy System Properties Files

Legacy System Properties files (`*.system.properties`) are no more supported by Architecture `8.0.0` . These files must be renamed to `*.properties.list` files (see *System Properties* for more details).

To facilitate the migration, legacy System Properties files are detected by SOAR and the following error is thrown:

```
1 : SOAR-L ERROR :
[M78] - System properties file [properties/xxx.system.properties] in classpath entry [...]␣
↪must be renamed to [properties/xxx.properties.list].
```

The following modules declare legacy System Properties files in older versions. Make sure to update the module to the specified version or a newer release in your projects.

- Pack `NET` version `9.4.2` .
- Add-On library `eclasspath-logging` version `1.2.1` .
- Testsuite `FS` version `3.0.7` .

CHAPTER

SEVEN

KERNEL DEVELOPER GUIDE

7.1 Overview

7.1.1 Introduction

The Kernel Developer's Guide describes how to create a Kernel Application. A Kernel Application is a *Standalone Application* that can be extended (statically or dynamically) to install, run, and control the execution of new applications called *Sandboxed Applications*.

The intended audience of this document are Java developers and system architects who plan to design and build their own Kernel.

Here is a non-exhaustive list of the activities to be done by Kernel Developers:

- Integrating the Kernel Application with a VEE Port to produce a Multi-Sandbox Executable and Virtual Device
- *Defining the set of APIs* that will be exposed to Applications, optionally by maintaining a custom *Runtime Environment*
- Managing lifecycles of applications (deciding when to install, start, stop and uninstall them)
- Defining and applying permissions on system resources (rules & policies)
- Managing connectivity
- Controlling and monitoring resources

This document takes as prerequisite that a VEE Port is available for the target device (see *VEE Porting Guide*). This document also assumes that the reader is familiar with the development and deployment of Applications (see *Application Developer Guide*) and specifics of developing Sandboxed Applications (see *Sandboxed Application*).

7.1.2 Terms and Definitions

A *Multi-Sandbox VEE Port* is a VEE Port with the Multi-Sandbox capability of the Core Engine enabled (see the chapter *Multi-Sandbox* of the *VEE Porting Guide*). A Multi-Sandbox Executable can only be built with a Multi-Sandbox VEE Port.

A *Virtual Device* is the Multi-Sandbox Executable counterpart for developing a Sandboxed Application. It provides the Kernel functional simulation part. Usually it also provides a mean to directly deploy a Sandboxed Application on the target device running the Multi-Sandbox Executable (this is called *Local Deployment*).

7.1.3 Overall Architecture

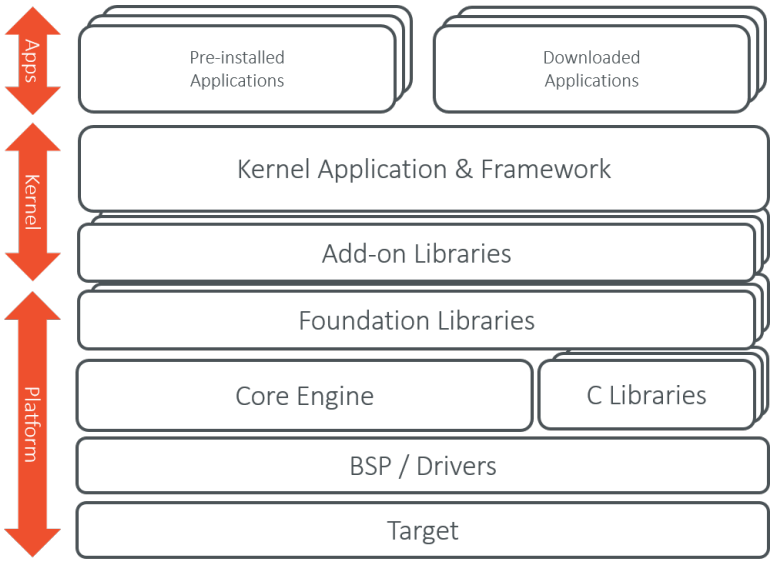


Fig. 1: Kernel Boundary Overview

7.1.4 Input and Output Artifacts

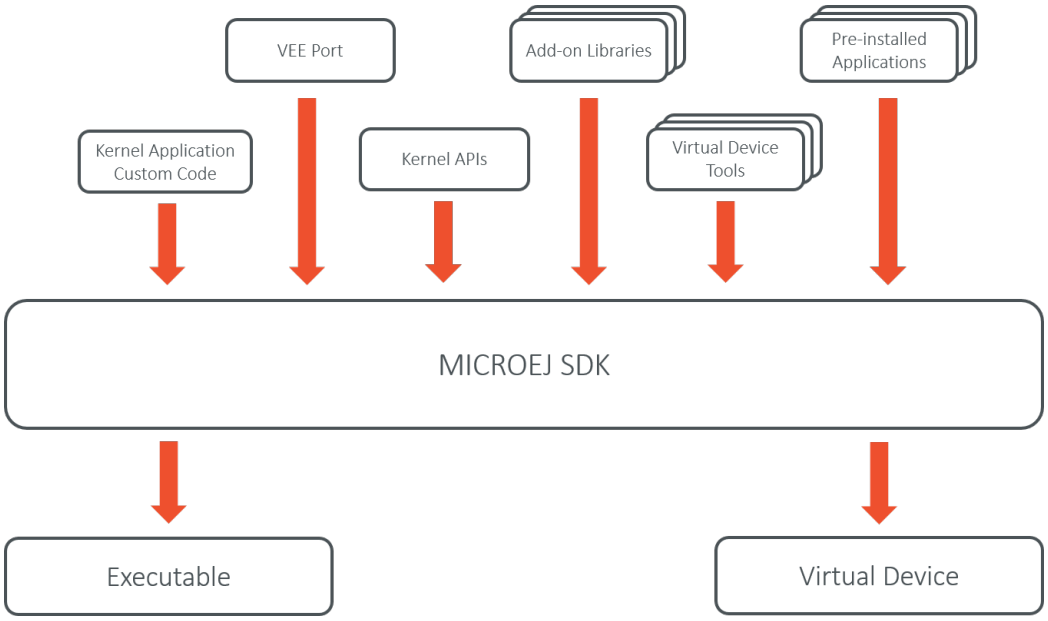


Fig. 2: Kernel Input and Output Artifacts

7.1.5 Kernel Build Flow

The following describes the Kernel build flow.

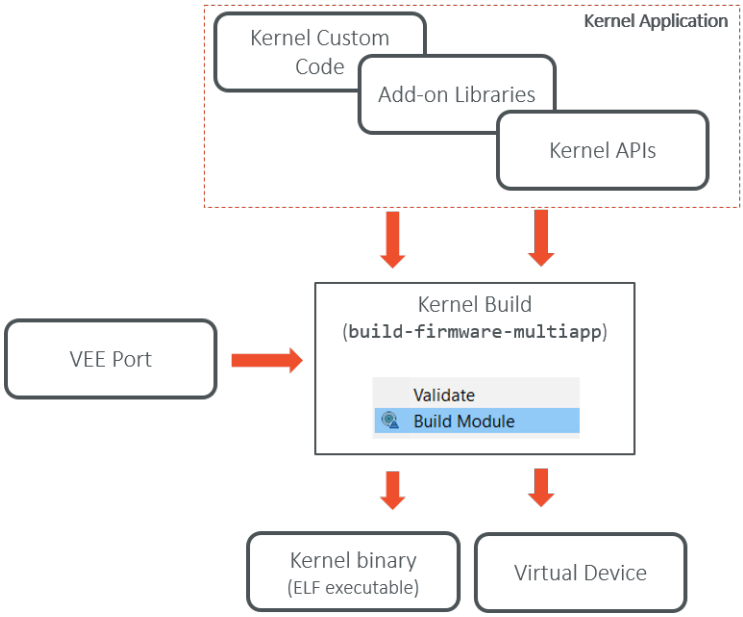


Fig. 3: Kernel Build Flow

The Virtual Device builder performs the following steps:

- Remove the embedded part of the VEE Port (including MEJ32).
- Append Add-On Libraries and *pre-installed Applications* into the runtime classpath. See *Kernel Module Configuration* section for specifying the dependencies.
- Add a custom license allowing Virtual Device redistribution.
- Generate the Runtime Environment from the Kernel APIs.

7.1.6 Kernel Implementation Libraries

Kernel implementations must cover the following topics:

- The kernel entry point implementation, that deals with configuring the different policies, registering kernel services and converters, and starting applications.
- The storage infrastructure implementation: mapping the `Storage` service on an actual data storage implementation. There are multiple implementations of the data storage, provided in different artifacts that will be detailed in dedicated sections.
- The applications management infrastructure: how application code is stored in memory and how the lifecycle of the code is implemented. Again, this has multiple alternative implementations, and the right module must be selected at build time to cover the specific Kernel needs.
- The simulation support: how the Virtual Device implementation reflects the Executable implementation, with the help of specific artifacts.
- The Kernel API definition: not all the classes and methods used to implement the Kernel Application are actually exposed to the Sandboxed Applications. There are some artifacts available that expose some of the libraries to the applications, these ones can be picked when the Kernel is assembled.

- The Kernel types conversion and other KF-related utilities: Kernel types instances owned by one application can be transferred to another application through a Shared Interface. For that to be possible, a conversion proxy must be registered for this kernel type.
- Tools libraries: tools that plug into the SDK, extending them with features that are specific to the Kernel, like deployment of an application, a management console, ...

7.2 Kernel & Features Specification

Multi-Sandboxing is based on the Kernel & Features specification (KF). The fundamental concepts are introduced in the *Sandboxed Application chapter*.

The following table provides links to the complete KF APIs & specification.

Documentation	Link
Java APIs	https://repository.microej.com/javadoc/microej_5.x/apis/ej/kf/package-summary.html
Specification	https://repository.microej.com/packages/ESR/ESR-SPE-0020-KF-1.4-H.pdf
Module	https://repository.microej.com/modules/ej/api/kf/

7.3 Getting Started

MicroEJ provides a ready-to-use Kernel template called *Kernel GREEN* to get familiar with Kernel development.

This Kernel is available in two different formats:

- in binary: pre-built Executables and Virtual Devices for evaluation boards available at https://repository.microej.com/packages/green/. It allows the Application developer to write its first Sandboxed Application quickly and then dynamically deploy it through a TCP/IP connection. See the Get Started with Multi-Sandbox for STM32F7508-DK Discovery Kit.
- in source: the Kernel GREEN repository is available at https://github.com/MicroEJ/Kernel-GREEN. It allows the Kernel developer to explore Multi-Sandboxing capabilities and start to adapt them to its needs. See the README file from the repository. It contains a step-by-step guide to build the Kernel GREEN on any compatible VEE Port.

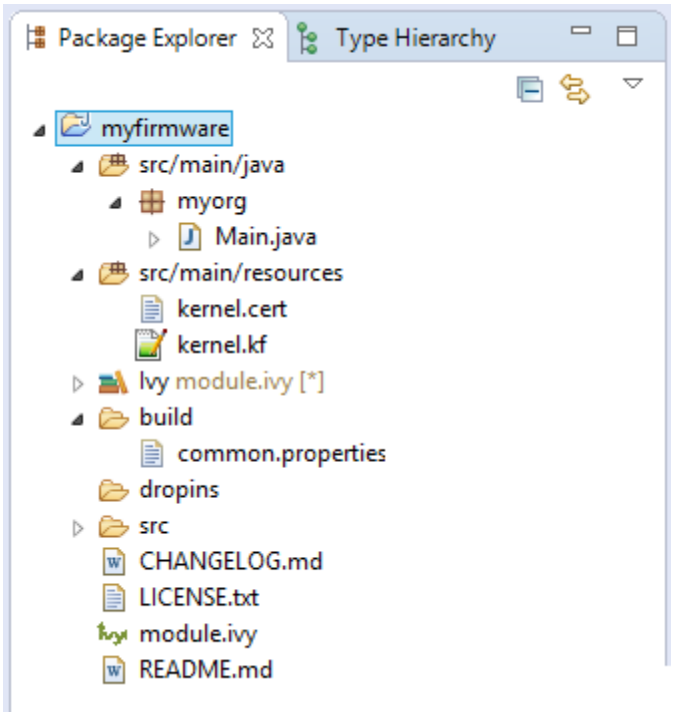
7.4 Kernel Creation

This chapter requires a minimum understanding of *MicroEJ Module Manager* and *Module Natures*.

7.4.1 Create a new Project

First create a new *Kernel Application*.

A new project is generated into the workspace:



7.4.2 Configure a VEE Port

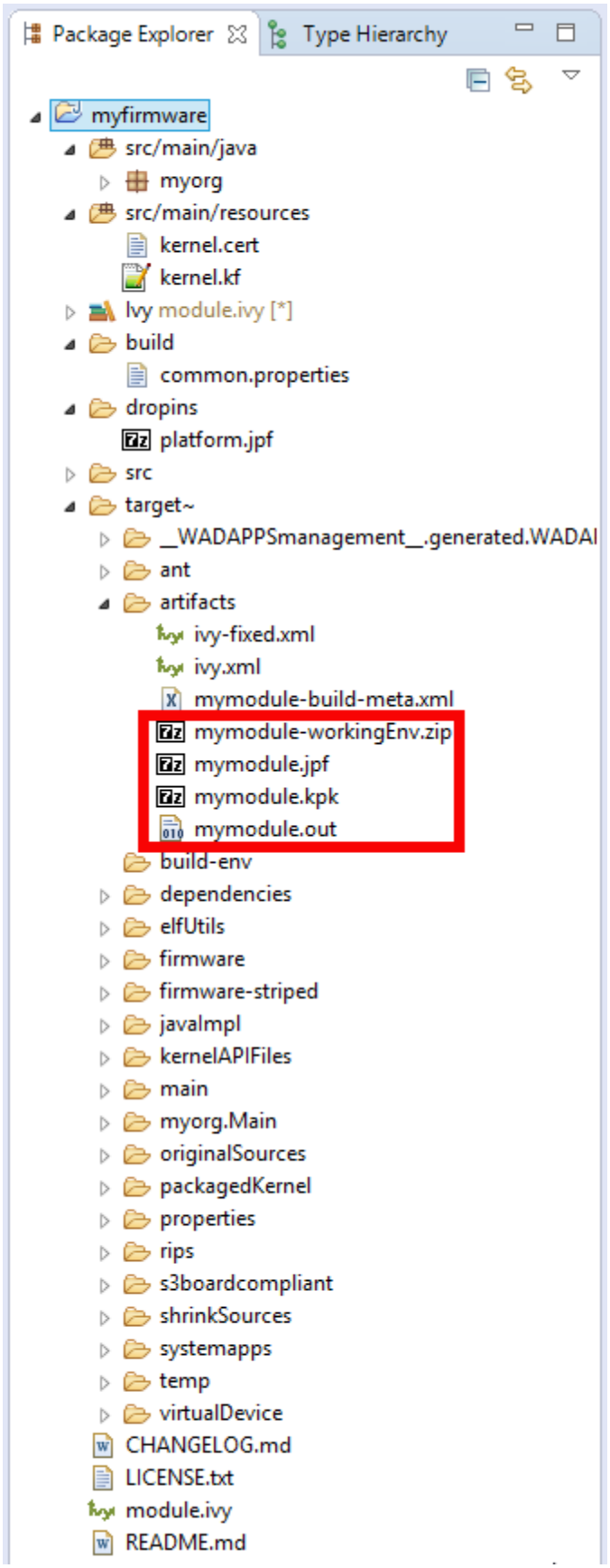
Before building the Kernel, you need to build a VEE Port with Multi-Sandbox capability. To enable the Multi-Sandbox capability in your VEE Port configuration, follow the instructions from the *Multi-Sandbox* section.

Once the VEE Port is built, configure the target VEE Port in your Kernel project. See *Platform Selection*.

7.4.3 Build the Executable and Virtual Device

In the Package Explorer, right-click on the project and select `Build Module` . The build of the Executable and Virtual Device may take several minutes. Once the build has succeeded, the folder `myfirmware` > `target~` > `artifacts` contains the firmware output artifacts (see *Input and Output Artifacts*) :

- `mymodule.out` : The Executable to be programmed on device.
- `mymodule.kpk` : The Kernel package to be imported in a MicroEJ Forge instance.
- `mymodule.vde` : The Virtual Device to be imported in the SDK.
- `mymodule-workingEnv.zip` : This file contains all files produced by the build phase (intermediate, debug and report files).


Package Explorer
Type Hierarchy
myfirmware
src/main/java
myorg
src/main/resources
kernel.cert
kernel.kf
Ivy module.ivy [*]
build
common.properties
dropins
platform.jpf
src
target~
__WADAPPSmanagement__.generated.WADAI
ant
artifacts
ivy-fixed.xml
ivy.xml
mymodule-build-meta.xml
mymodule-workingEnv.zip
mymodule.jpf
mymodule.kpk
mymodule.out
build-env
dependencies
elfUtils
firmware
firmware-striped
javaImpl
kernelAPIFiles
main
myorg.Main
originalSources
packagedKernel
properties
rips
s3boardcompliant
shrinkSources
systemapps
temp
virtualDevice
CHANGELOG.md
LICENSE.txt
module.ivy
README.md

7.4.4 Expose APIs

A Kernel must define the set of classes, methods and static fields all applications are allowed to use.

Note: According to the *Kernel and Features specification*, no API is open by default to Sandboxed Applications.

This can be done either by declaring *Kernel APIs* or by defininig a *Runtime Environment*.

The main difference is from the Application development point of view. In the first case, the Application project still declares standard module dependencies. This is the good starting point for quickly building a Kernel with Applications based on the MicroEJ modules as-is. In the second case, the Application project declares the runtime environment dependency. This is the preferred way in case you intend to build and maintain a dedicated Applications ecosystem.

A Kernel API or a Runtime Environment module is added as a dependency with the configuration `kernelapi->default` .

```
<dependency org="com.microej.kernelapi" name="edc" rev="1.0.6" conf="kernelapi->default"/>
```

7.4.5 Implement a Security Policy

The Kernel can restrict sensitive or possibly unsafe operations performed by Sandboxed Applications, thus defining a security policy. Implementing a security policy is achieved by enabling support for Security Management system-wide and by registering to the Kernel a custom SecurityManager that will handle the Permission checks.

Note: An API controlled by the Security Manager must be guarded by a *Permission check*. The usual API documentation convention is to declare to throw a SecurityException with details about the requested Permission.

Enable the Security Management

For the sake of ROM footprint optimization, calls to Permission checks are disabled by default. In order to activate this feature the *Option(checkbox): Enable SecurityManager checks* option must be set.

Implement your Security Policy

This can be achieved by subclassing the base SecurityManager class, overriding its SecurityManager.checkPermission(Permission) method, and registering an instance of this class to the Kernel by a call to System.setSecurityManager(SecurityManager).

```
// create a new Security Manager
SecurityManager sm = new SecurityManager() {
   @Override
   public void checkPermission(java.security.Permission perm) {
      // here implement your Kernel Security Policy
   };
};
// register the Security Manager
System.setSecurityManager(sm);
```

Then you have to implement your own Security Policy.

Implementation of a Security Policy is demonstrated in the Kernel-GREEN project. This Kernel implements a logging-only Security Policy using the utility class FeaturePermissionCheckDelegate that helps in implementing Permission checks in a Multi-Sandbox environment.

7.4.6 Add Pre-installed Applications

Your device may come with pre-installed applications, also known as applications that are already available when the Kernel starts. These applications are installed during the manufacturing process, such as in ROM alongside the Kernel executable.

To mimic this behavior on a Virtual Device, add a new dependency with the configuration `systemapp-vd->application`.

```
<dependency org="com.mycompany" name="myapp" rev="0.1.0" conf="systemapp-vd->application"/>
```

7.4.7 Build the Executable in the Workspace

It is possible to build the Executable using a *MicroEJ Launch* rather than the regular module build. This speeds-up the build time thanks to MicroEJ Module Manager workspace resolution and Eclipse incremental compilation.

- Import the Kernel project and all Sandboxed Application projects in the same workspace,
- Prepare a MicroEJ Application launch for the Kernel as a regular *Standalone Application*,
- Prepare a MicroEJ Application launch for each Sandboxed Application using *Build Dynamic Feature* settings.

The following figure shows the overall build flow:

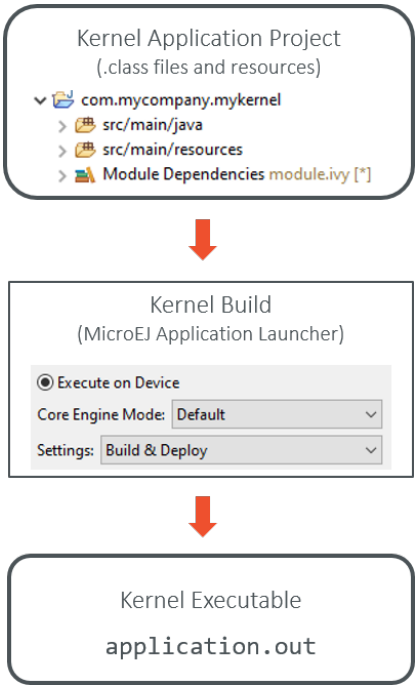


Fig. 4: Kernel Build Flow using MicroEJ Launches

7.4.8 Kernel Application Configuration

Module Configuration

The `build-firmware-multiapp` build type defines additional configurations, used to specify the different kind of firmware inputs (see *Input and Output Artifacts*) as dependencies.

The following table lists the different configuration mapping usage where a dependency line is declared:

```
<dependency org="..." name="..." rev="..." conf="[Configuration Mapping]"/>
```

Table 1: Configurations Mapping for `build-firmware-multiapp` Build Type

Configuration Mapping	Dependency Kind	Usage
`vdruntime->default`	Add-On Library (`JAR`)	Embedded in the Virtual Device only, not in the Executable
`default->default; vdruntime->default`	Add-On Library (`JAR`)	Embedded in both the Executable and the Virtual Device
`platform->default`	VEE Port	VEE Port dependency used to build the Executable and the Virtual Device. There are other ways to select the VEE Port (see *Platform Selection*)
`kernelapi->default`	Runtime Environment (`JAR`)	See *Runtime Environment*
`systemapp-vd->application`	Application (`WPK`)	Included to the Virtual Device as pre-installed Application.

Example of minimal firmware dependencies.

The following example defines a Kernel that exposes all APIs of `EDC` library.

```
<dependencies>
    <dependency org="ej.api" name="edc" rev="1.2.0" conf="provided" />
    <!-- Runtime API (set of Kernel API files) -->
    <dependency org="com.microej.kernelapi" name="edc" rev="1.0.0" conf="kernelapi->default"/
↪>
</dependencies>
```

Build Options

The *Kernel Application module nature* section describes all the options available for building a Kernel module.

Build only a Virtual Device with a pre-existing Kernel

Copy/Paste the `.kpk` file into the folder `dropins`

7.5 Kernel APIs

Kernel API files (`kernel.api`) specify among all types owned by the Kernel which ones **must** be used by Features, and for those types which members (method, and static fields) are allowed to be accessed by Features. When a type is not declared in a Kernel API, the Kernel and each Feature **can** have their own version of that type, but if a type is declared in a Kernel API file only the Kernel version will be used by the Kernel and all the Features.

For mode details refer to the *Class Spaces* chapter of the *Kernel & Features Specification*.

7.5.1 Kernel API Definition

A Kernel API file is an XML file named `kernel.api` declared at the root of one or more path composing the *Application classpath*.

Listing 1: Kernel API Example for exposing System.out.println API

```
<require>
    <type name="java.io.PrintStream"/>
    <type name="java.lang.String"/>
    <type name="java.lang.System"/>
    <field name="java.lang.System.out"/>
    <method name="java.io.PrintStream.println(java.lang.String)void"/>
</require>
```

The table below describes the format of the XML elements. The full XML schema is available in the *Kernel & Features Specification*.

Table 2: 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]`

7.5.2 Writing Kernel APIs

This section lists different ways to help to write `kernel.api` files.

Default Kernel APIs Derivation

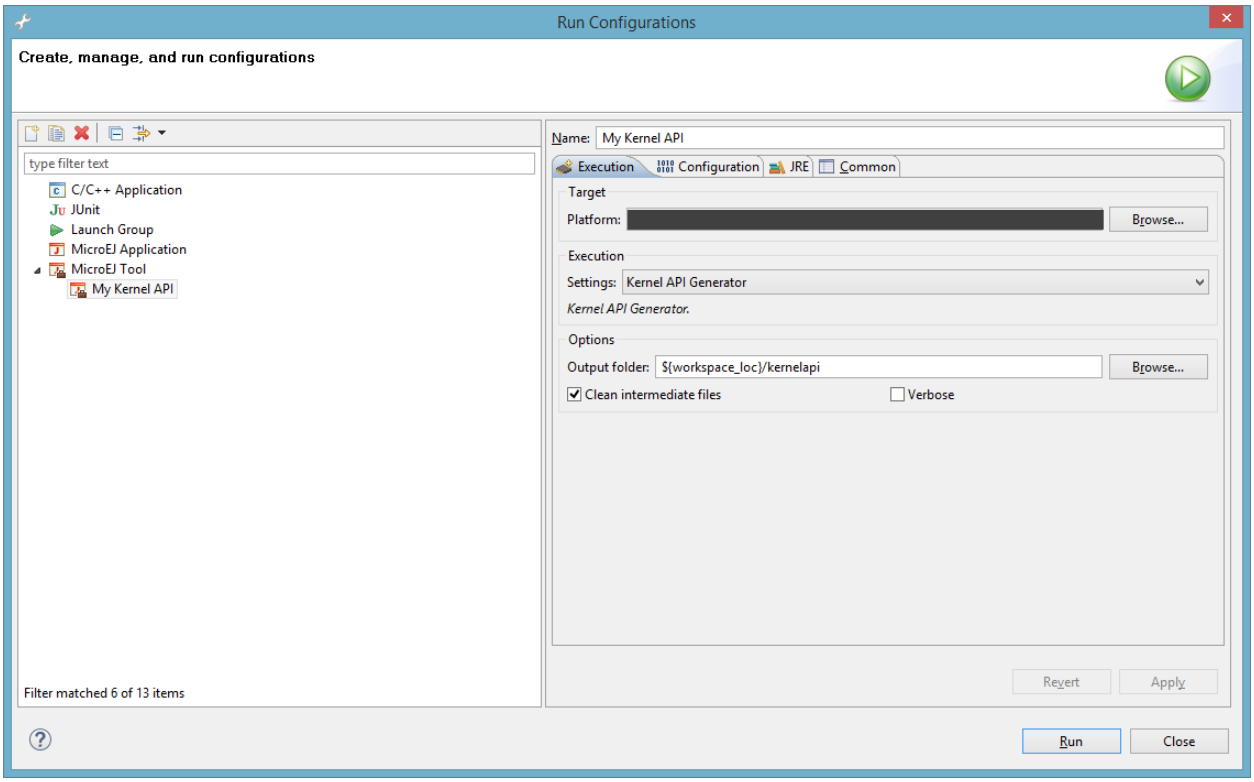
MicroEJ Corp. provides predefined kernel API files for the most common libraries provided by a Kernel. These files are packaged as MicroEJ modules in the *Developer Repository* under the com/microej/kernelapi organisation.

The packaged file `kernel.api` can be extracted from the JAR file and edited in order to keep only desired types, methods and fields.

Kernel API Generator

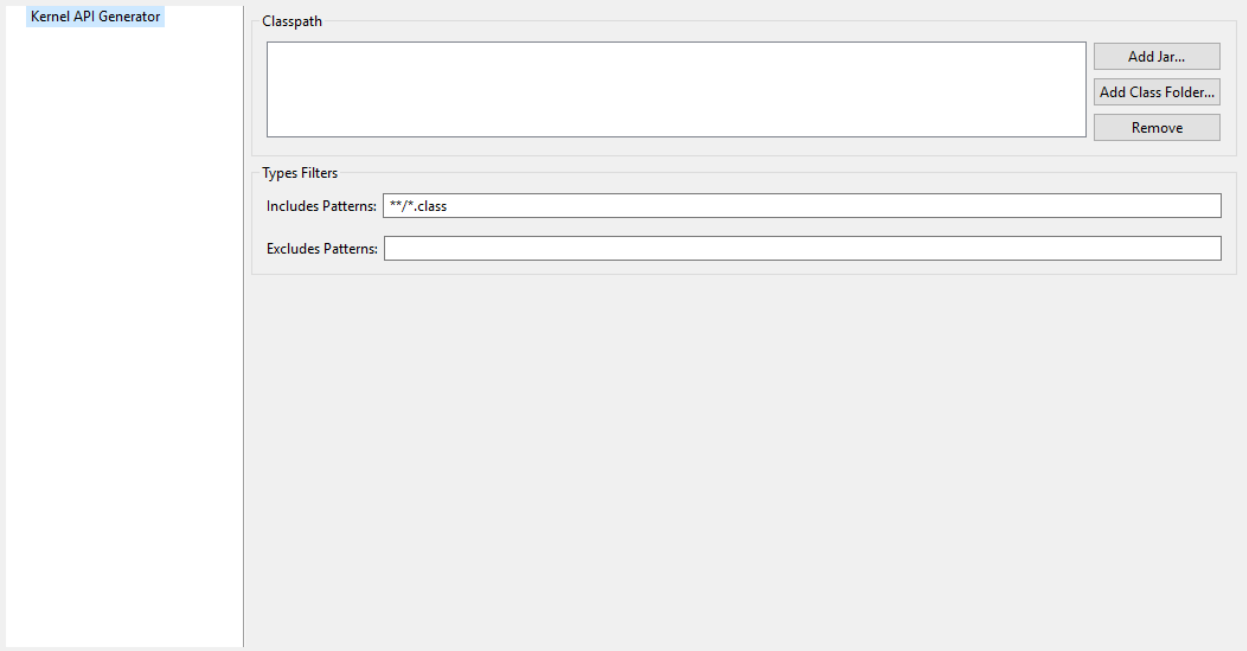
MicroEJ Kernel API Generator is a tool that help to generate a `kernel.api` file based on a Java classpath.

In the 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.

7.6 Runtime Environment

7.6.1 Principle

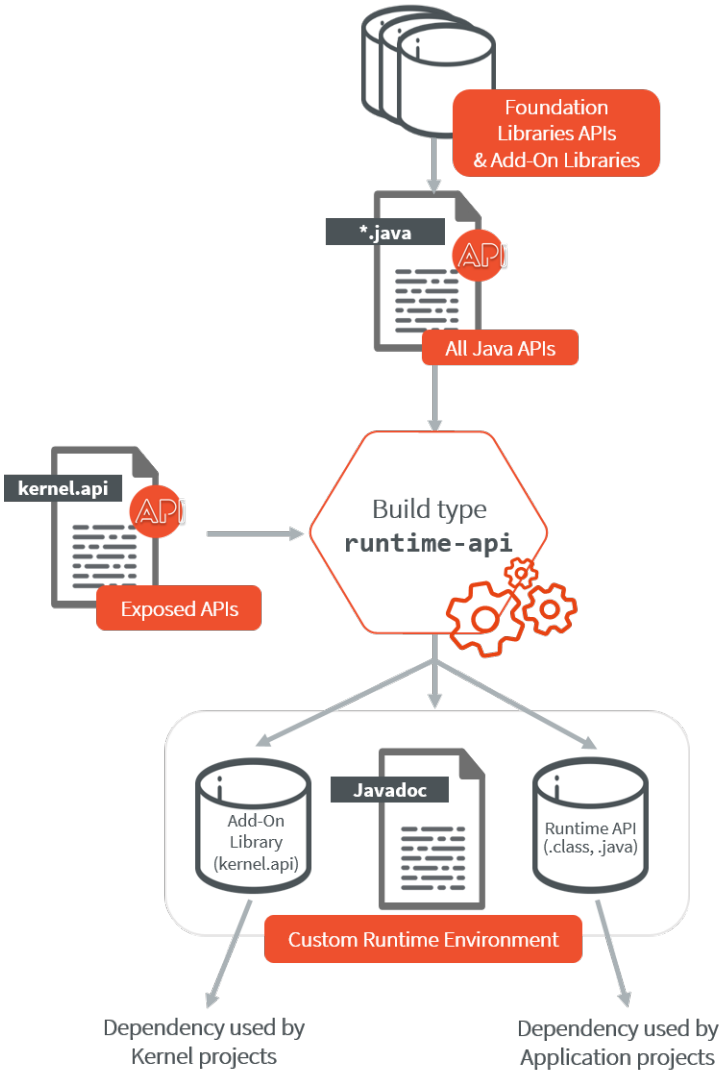
A Runtime Environment is a *module nature* for defining the set of APIs available to an Application developer on a Kernel. It is built by aggregating a set of *Kernel APIs*.

Building a Runtime Environment is one of the 2 solutions to define the APIs of a Kernel, as described in the section *Expose APIs*. Having the set of APIs named and versioned in a Runtime Environment allows to maintain, share and document it outside of a specific Kernel implementation.

Once built, a Runtime Environment module contains the following elements:

- A JAR file with the whole library of APIs (`.class` files and `.java`), used by Application projects to compile Application code;
- A JAR file with the `kernel.api` file defined in the module (if any), used by Kernel projects to fetch all the `kernel.api` files (by transitivity) to expose APIs when building the Firmware and the Virtual Device;
- A JAR file with the Javadoc of the APIs for documentation.

The following figure shows the overall build flow:



7.6.2 Create a new Runtime Environment Module

A Runtime Environment *module project* is created with the `runtime-api` skeleton.

```
<info organisation="com.mycompany" module="myruntimeapi" status="integration" revision="1.0.0
↪">
   <ea:build organisation="com.is2t.easyant.buildtypes" module="build-runtime-api" revision=
↪"4.0.+">
   </ea:build>
</info>
```

Kernel APIs as Dependencies

The Kernel APIs can be declared as dependencies of the module. For example, the following dependencies declare a Runtime Environment that aggregates all classes, methods and fields defined by `EDC`, `KF`, `BON`, `MicroUI` Kernel APIs modules.

```
<dependencies>
   <dependency org="com.microej.kernelapi" name="edc" rev="1.0.6"/>
   <dependency org="com.microej.kernelapi" name="kf" rev="2.0.3"/>
   <dependency org="com.microej.kernelapi" name="bon" rev="1.1.1"/>
   <dependency org="com.microej.kernelapi" name="microui" rev="3.1.0"/>
</dependencies>
```

The libraries modules are fetched transitively from the Kernel APIs dependencies. For example, the dependency `com.microej.kernelapi#edc;1.0.6` fetches the library ej.api#edc;1.2.3.

It is also possible to force the version of the libraries to use by declaring them as direct dependencies. This is typically used to get a latest version of the library with improvements such as Javadoc fixes or Null Analysis annotations. In this example:

```
<dependencies>
   <dependency org="com.microej.kernelapi" name="edc" rev="1.0.6"/>

   <dependency org="ej.api" name="edc" rev="1.3.4"/>
</dependencies>
```

The Runtime Environment uses the version `1.3.4` of the EDC library instead of the version `1.2.3` fetched transitively by the dependency `com.microej.kernelapi#edc;1.0.6`.

Kernel APIs as Project File

The Kernel APIs can also be defined in a file in the Runtime Environment directly. The file must be named `kernel.api` and stored in the `src/main/resources` folder.

Add Add-On Processors

When the Runtime Environment includes an Add-On Library which uses an Add-On Processor, this Add-On Processor must be declared as a direct dependency in the Runtime Environment.

The Add-On Processor dependency line can be retrieved as follows:

- In your target *module repository*, go to the Add-On Library folder,
- Open the `ivy-[version].xml` file,
- Search for the dependency line with `conf="addon-processor->addon-processor"`

```
<ivy-module xmlns:ea="http://www.easyant.org" xmlns:ej="https://developer.
↪microej.com" xmlns:m="http://ant.apache.org/ivy/maven" version="2.0"␣
↪ej:version="2.0.0">
   <info organisation="com.mycompany" module="mylibrary" revision="M.m.p"␣
↪status="release" publication="20220523165033">
      ...
   </info>
   <configurations>
```

(continues on next page)

(continued from previous page)

```
      ...
      <conf name="addon-processor" visibility="public" description="Addon␣
↪processors dependencies."/>
   </configurations>
   <publications>
      ...
   </publications>
   <dependencies>
      <dependency org="ej.api" name="edc" rev="1.3.3" conf="default->default;
↪provided->provided"/>
      ...
      <dependency org="com.mycompany.addon" name="mylibrary-processor" rev="x.y.
↪z" conf="addon-processor->addon-processor"/>
      ...
   </dependencies>
</ivy-module>
```

- In the Runtime Environment *module description file*, declare the `addon-processor` configuration in the list of `configurations`

```
<conf name="addon-processor" visibility="public" description="Add-On Processors␣
↪dependencies."/>
```

- Paste the Add-On Processor dependency line

Warning: If the Add-On library version is changed, the Add-On Processor version must be updated.

Here is a list of known libraries using an Add-On Processor:

- NLS:

```
<dependency org="com.microej.tool.addon.runtime" name="binary-nls-processor" rev="
↪<version>" conf="addon-processor->addon-processor"/>
```

- Wadapps:

```
<dependency org="ej.tool.addon.wadapps" name="wadapps-processor" rev="<version>" conf=
↪"addon-processor->addon-processor"/>
```

- JavaScript:

```
<dependency org="com.microej.tool.addon.runtime" name="js-processor" rev="<version>"␣
↪conf="addon-processor->addon-processor"/>
```

7.6.3 Use a Runtime Environment in an Application

The Runtime Environment dependency must be declared in the Application project as following:

```
<dependency org="com.mycompany" name="myruntimeapi" rev="1.0.0" conf="provided->runtimeapi"/>
```

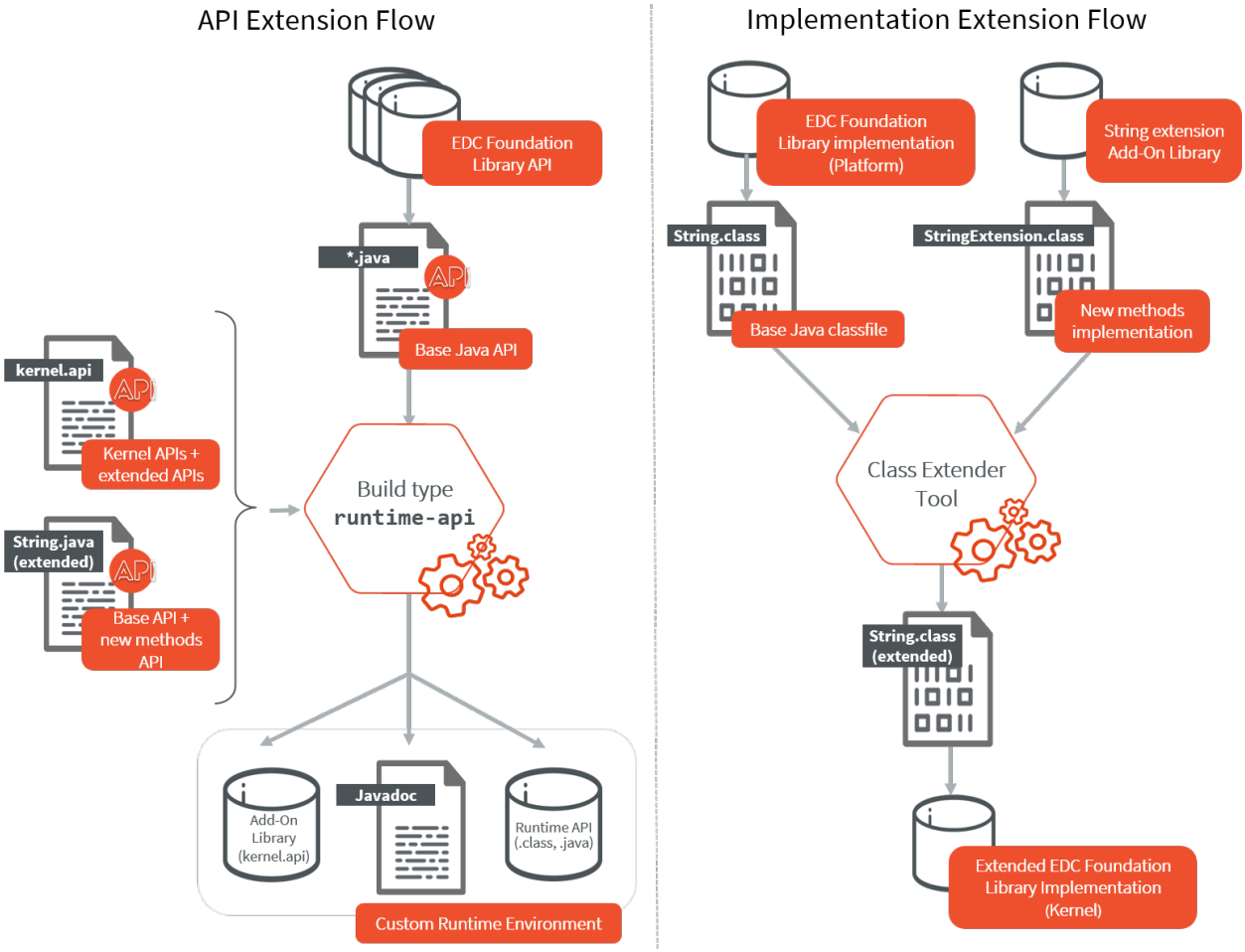
Note: If you want to add an other library dependency, make sure it is has been built on this Runtime Environment. Otherwise this could lead to inconsistent situations, for example by using an API not available at runtime. An other approach is to add it to the Runtime Environment.

7.6.4 Extend a Runtime Environment

In a Kernel, Foundation and Add-On libraries can be extended by adding new methods to their existing classes. For example, it allows to add new methods to the class java.lang.String of the module ej.api#edc. This is done thanks to the Class Extender tool. This tool works at binary level and is able to inject methods from one class to another. Extensions can thus be independently compiled and be retrieved by the Kernel and applied during a Multi-Sandbox Executable build.

To make the extensions available to Application developers, the Runtime Environment has to be extended too.

The following diagram illustrates the process of extending the default java.lang.String class from *[EDC]* from a Kernel developer point of view:



The extension must be applied in 2 locations:

1. In the Runtime Environment. This ensures that Applications developers can see and use the new methods. The custom Runtime Environment must contain the following element:

   - the API to extend, as a dependency. Here this is the EDC Foundation Library API, which contains the java.lang.String class we want to extend. We can add it transitively through its kernelapi:

   ```
   <dependency org="com.microej.kernelapi" name="edc" rev="1.0.6"/>
   ```

   - a *Kernel API* file definition in the `src/main/resources` folder which includes the new methods. For example:

   ```
   <?xml version="1.0" encoding="UTF-8"?>
   <require>
      <method name="java.lang.String.myNewMethod(int)java.lang.String"/>
      <method name="java.lang.String.myOtherNewMethod()void"/>
   </require>
   ```

   - the new version of the Java source of the API to extend. This class overrides the original class fetched from the dependency. Therefore it must include all the methods, the ones existing in the original class as well as the new methods, with their Javadoc specification. In our example, we must add a new `String.java` source file in the `src/main/java/java/lang` folder, and add the new methods:

```
public String myNewMethod(int number) {
   return "My number is " + number;
 }

public void myOtherNewMethod() {
   System.out.println("Hello!");
}
```

This class overrides the java.lang.String class fetched from the EDC dependency.

Once built, the custom Runtime Environment contains the new methods and can be used in the Applications projects.

2. In the Kernel. The EDC implementation is extended during the Kernel build thanks to the Class Extender tool. Refer to the Class Extender tool README and especially to the chapter `Include Class Extender During Firmware Project Build` to learn how to integrate it in a Kernel build.

   MicroEJ Corp. provides some ready-to-use extension modules:

   - com.microej.library.runtime#string-regex: String methods based on Regular Expressions (e.g. `String.split()`, `String.replaceAll()`)
   - com.microej.library.runtime#string-format: String formatting utility methods (e.g. `String.format()`)

7.7 Kernel UID

The Kernel UID is a sequence of bytes that uniquely identifies the Kernel. This UID is generated by SOAR from Java code content, Platform characteristics and a timestamp. Two Kernels built from the same Kernel Application code will not share the same UID.

The Kernel UID is used by *Core Engine* to check if an Application can be installed on a Kernel. During the *Application build*, the resulting `.fo` file embeds the Kernel UID on which it has been built.

During Kernel.install(), the UID embedded in the `.fo` is compared with the Kernel UID. By default, if both UIDs are equal the Application installation continues. Otherwise it is stopped. See also *Feature Portability Control* for `.fo` installation on different Kernels.

The Kernel UID can be retrieved at runtime using Kernel.getInstance() .getUID().

7.8 Sandboxed Application Lifecycle

The lifecycle of an Sandboxed Application is managed by the Kernel.

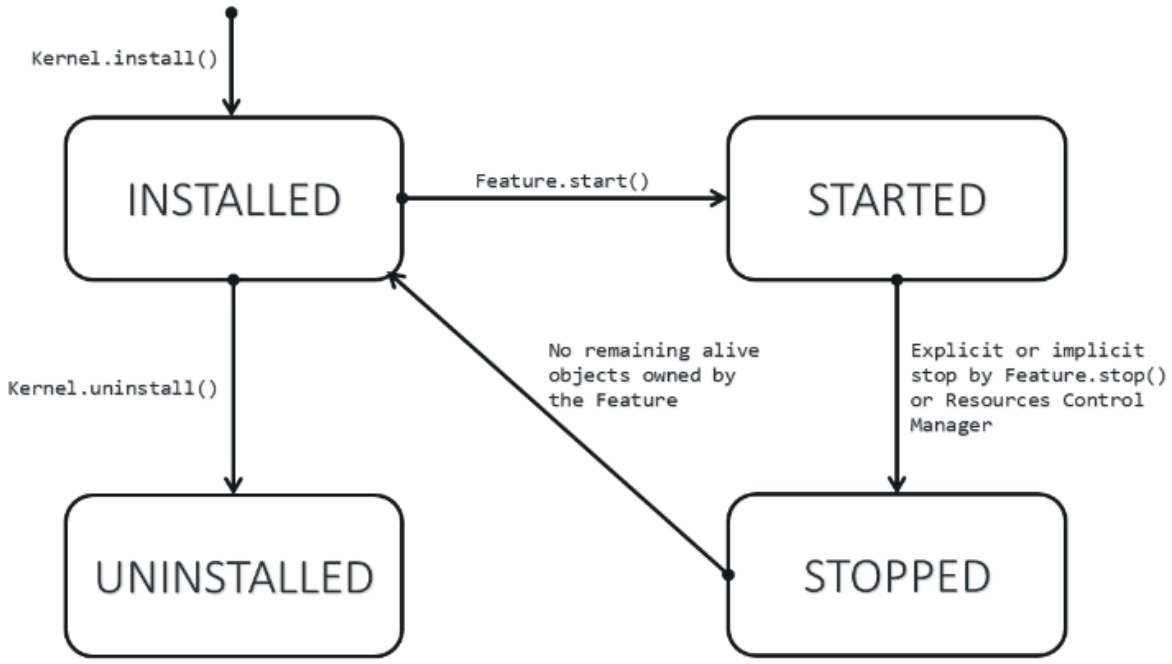


Fig. 5: Sandboxed Application Lifecycle

An Application is in one of the following states:

- **INSTALLED**: the Application has been successfully linked to the Kernel and is not running. There are no references from the Kernel to objects owned by this Application.
- **STARTED**: the Application has been started and is running.
- **STOPPED**: the Application has been stopped and all its owned threads and execution contexts are terminated. The memory and resources are not yet reclaimed.
- **UNINSTALLED**: the Application has been unlinked from the Kernel.

7.9 Kernel and Features Communication

Kernel and Features can communicate with each other by sharing interface implementation instances at runtime.

In this section you will learn:

- How two (or more) Feature(s) can communicate with each other.
- How the Kernel can communicate with a Feature.

Below are defined several terms that will be used throughout this page:

- *Shared Interface* is a mechanism specific to MicroEJ Multi-Sandbox that allows exchanging object instances between Features.
- Service represents an object instance (i.e an interface implementation)
- Shared Services is a MicroEJ helper module that eases sharing services within a Multi-Sandbox context; it provides generic APIs that can be re-implemented as needed.

- Registry or Service Registry represents the actual implementation of Shared Services APIs. MicroEJ provides such registries for KF but custom registries can be implemented as needed.

7.9.1 Shared Services

Services can be shared by means of the ej.Service library.

The Shared Services mechanism relies on a registry system that mostly consists in a Java map of class types to object instances (`Map<Class<?>, Object>`).

Each Feature owns a local registry in which it can register and get services within its own context; services registered in a local context cannot be retrieved by the Kernel nor any other Feature.

The Kernel also has a local registry in which it can register services that can be used within its own context but not from the context of Features.

Finally there exists a unique shared service registry contains all the registered shared services; this registry is available to all Features and to the Kernel as well.

Security policies can be implemented to restrict the usage of certain services by certain Features.

Note: The following sections relate to the existing KF implementation of the ej.Service library available in the KF-Util module ; you can however do your own custom implementation depending as needed.

7.9.2 Communication between Features

The KF specification does not allow Features to access object instances from other Features directly: access can only be done by means of a proxy of the target object instance.

This is made possible through the Shared Interfaces mechanism. More information about proxies can be found in the *Shared Interfaces* section.

In a nutshell Shared Interfaces and Shared Services are two complementary notions: the Shared Interfaces mechanism is responsible for setting up the capability of sharing an instance between Features whereas Shared Services offer a way to get, store and retrieve these instances once correctly set up.

Register a Service

The following line of code allows you to easily register a Service instance.

```
ServiceFactory.register(MyInterface.class,myInterface)
```

When registering a service from a Feature there are two possible options:

- The registered service is not a Shared Interface; in this case the service instance will be registered in a local service registry and only available from the Feature itself.
- The registered service is a Shared Interface; in this case the service instance will be registered in the Shared Service Registry and therefore available to any other Features that has a proxy for this instance.

For Features to use the Shared Interfaces mechanism, a Kernel must provide:

- an API for a first Feature to register its Shared Interface, and for a second Feature to get a proxy on it (by means of the ej.Service library)
- a set of registered Kernel types converters (see below)

Get a Service

The following line of code allows you to easily get a Service instance.

```
MyInterface myInterface = ServiceFactory.getService(MyInterface.class)
```

When getting a service instance from a Feature, the service instance is searched in the following order:

1. In the Local Registry, check for an instance registered by the Feature.
2. In the Shared Registry, check for an instance registered by the Feature itself.
3. In the Shared Registry, check for an instance registered (publicly) by the Kernel.
4. In the Shared Registry, check for an instance registered as a Shared Interface by an other Feature.

7.9.3 Communication between Kernel and Feature

The Kernel can also communicate with Features using Shared Services, by exposing object instances to Features in the shared registry.

Register a Service

From the Kernel side two distinct APIs may be used to register a Service, depending on whether the service must be registered locally or not.

You can use the generic ej.Service API that will automatically register the service instance in the **local** Kernel service registry.

```
ServiceFactory.register(MyInterface.class,myInterface) //accessible within the Kernel␣
↪context only
```

Or you can specify in which registry the Kernel should register the service by using the ServiceRegistryKF API from the KF-Util module as depicted below.

By doing so the service instance is exposed in the Shared Registry.

Note: To allow the usage of Kernel APIs by Features, you must make sure that the Kernel registers the necessary Kernel APIs. Learn more about Kernel API. Use of extra APIs from ServiceRegistryKF to specify the registry is reserved for the Kernel and will throw an exception if used from a Feature context.

Get a Service

The following line of code allows you to easily get a Service instance.

```
MyInterface myInterface = ServiceFactory.getService(MyInterface.class)
```

When getting a service instance from the Kernel, the service instance is searched in the following order:

1. In the Local Registry, check for an instance registered by the Kernel.
2. In the Shared Registry, check for an instance registered by the Kernel.
3. In the Shared Registry, check for an instance registered as Shared Interface by an other Feature.

If no instance was found, an attempt is made to create a new instance of the provided type from *System Properties*.

This property binds the service type (the property key) to the actual service implementation type (the property value) that will be used for instantiation.

For example, in order to allow an instance of the `ej.bon.Timer` service to be created automatically if not present, the following property must be set:

```
ej.bon.Timer=ej.bon.Timer
```

Note: Since the service type and the implementation type are dynamically bound using class reflection, both types must be declared as *Required Types*.

7.9.4 Implement a Registry

In case the existing KF implementation of Shared Services does not fit your needs, you can implement your own registry system classes using the Kernel.bind() KF API.

This API allows a consumer Feature for remote use of an instance which type is owned by another Feature or the Kernel. In case the type is owned by another Feature, the returned instance is a Proxy of the shared instance. In case the type is owned by the Kernel, the returned instance is the conversion result of the shared instance to the Kernel type; for this to happen a suitable *Converter* must be registered.

As an example the steps below describe how to implement a generic Shared Interface service that relies on the Kernel.bind() API.

1. Declare the following class in your Kernel

```
package com.microej.example;

import ej.kf.Feature;
import ej.kf.Feature.State;
import ej.kf.FeatureStateListener;
import ej.kf.Kernel;
import ej.kf.Module;

/**
* Example of Kernel APIs for registering a generic Shared Interface service.
*/
public class GlobalService {

   private static Object GLOBAL_SERVICE;
   static {
      // automatically unregister the global service when the Feature is stopped.
      Kernel.addFeatureStateListener(new FeatureStateListener() {

         @Override
         public void stateChanged(Feature feature, State previousState) {
            synchronized (GlobalService.class) {
               if (GLOBAL_SERVICE != null && Kernel.getOwner(GLOBAL_SERVICE) == feature
                     && previousState == State.STARTED) {
                  GLOBAL_SERVICE = null;
               }
```

(continues on next page)

(continued from previous page)

```
            }
         }
      });
   }

   /**
   * Basic API to register a Feature service. <br>
   * The service is automatically unregistered when the Feature is stopped.
   *
   * @param service
   *            the service being registered. It must implement a shared interface.
   */
   public synchronized static void registerService(Object service) {
      Kernel.enter();
      GLOBAL_SERVICE = service;
   }

   /**
   * Basic API to retrieve a Feature service. <br>
   *
   * @param <T>
   *            the interface type
   *
   * @param serviceClass
   *            the interface of the service being retrieved. It must implement a shared␣
↪interface.
   * @return the binded service or <code>null</code> if no registered service
   */
   @SuppressWarnings("unchecked")
   public synchronized static <T> T getService(Class<T> serviceClass) {
      Module contextOwner = Kernel.getContextOwner();
      Kernel.enter();
      if (GLOBAL_SERVICE == null) {
         return null;
      }
      return Kernel.bind((T) GLOBAL_SERVICE, serviceClass, (Feature) contextOwner);
   }
}
```

1. Declare the following exposed APIs in your `kernel.api` file (refer to Kernel API Definition for details)

```
<method name="com.microej.example.GlobalService.registerService(java.lang.Object)void" />
<method name="com.microej.example.GlobalService.getService(java.lang.Class)java.lang.Object"␣
↪/>
```

1. Your App1 is ready to register a Shared Interface as a service

```
MySharedInterface service = new MySharedInterface();
GlobalService.registerService(service);
```

1. Your App2 is ready to retrieve a Shared Interface as a service

```
MySharedInterface service = GlobalService.getService(MySharedInterface.class))
service.use();
```

7.9.5 Kernel Types Converter

The Shared Interface mechanism allows to transfer an object instance of a Kernel type from one Feature to an other (see *Transferable Types* section).

To do that, the Kernel must register a new Kernel type converter. See the Converter class and Kernel.addConverter() method for more details.

The table below shows some converters defined in the com.microej.library.util#kf-util library.

Table 3: Example of Available Kernel Types Converters

Type	Converter Class	Conversion Rule
java.lang.Boolean	BooleanConverter	Clone by copy
java.lang.Byte	ByteConverter	Clone by copy
java.lang.Character	CharacterConverter	Clone by copy
java.lang.Short	ShortConverter	Clone by copy
java.lang.Integer	IntegerConverter	Clone by copy
java.lang.Float	FloatConverter	Clone by copy
java.lang.Long	LongConverter	Clone by copy
java.lang.Double	DoubleConverter	Clone by copy
java.lang.String	StringConverter	Clone by copy
java.io.InputStream	InputStreamConverter	Create a Proxy reference
java.util.Date	DateConverter	Clone by copy
java.util.List<T>	ListConverter	Clone by copy with recursive element conversion
java.util.Map<K,V>	MapConverter	Clone by copy with recursive keys and values conversion

7.10 Multi-Sandbox Enabled Libraries

A Multi-Sandbox enabled library is a Foundation Library or an Add-On Library that can be embedded by a Kernel with its APIs exposed to Features.

A library requires specific code for enabling Multi-Sandbox in the following cases:

- it implements an internal global state: lazy initialization of a singleton, registry of callbacks, internal cache, ...,
- it provides access to native resources that must be controlled using a Security Manager.

Otherwise, the library is called a stateless library. A stateless library is Multi-Sandbox enabled by default: it can be embedded by the Kernel, and its APIs are directly exposed to Features without code modification.

Note: This chapter generally applies to any Kernel code, not just libraries.

7.10.1 Manage Internal Global State

A library may define code that performs modifications of its internal state, for example:

- lazy initialization of a singleton,
- registering/un-registering a callback,
- maintaining an internal global cache, ...

By default, calling one of these APIs from a Feature context will throw one of the following errors:

```
java.lang.IllegalAccessError: KF:E=S1
   at <Kernel Method>
   ...
   at <Feature Method>

java.lang.IllegalAccessError: KF:E=F1
   at <Kernel Method>
   ...
   at <Feature Method>
```

The reason is that the Core Engine rejects assigning a Feature object in a static field or an instance field owned by the Kernel. See the *KF library access error codes* for more details. This prevents unwanted object links from the Kernel to the Feature, which would lead to stale references when stopping the Feature.

The library code must be adapted to implement the desired behavior when the code is called from a Feature context. The following sections describe the most common strategies applied on a concrete example:

- declaring a static field local to the Feature,
- allowing a field assignment in Kernel mode,
- using existing Multi-Sandbox enabled data structures.

Declare a Static Field Local to the Feature

The *Kernel & Features Specification* defines Context Local Storage for static fields. This implies that the Core Engine allocates a dedicated memory slot to store the static field for each execution context (the Kernel and each Feature).

Context Local Storage for static fields is typically used when the library defines a lazy initialized singleton. A lazy initialized singleton is a singleton that is only allocated the first time it is required. This is how is implemented the well-known Math.random() method:

```
public class Math{
   private static Random RandomGenerator;

   public static double random() {
      if(RandomGenerator == null) {
         RandomGenerator = new Random();
      }
      return RandomGenerator.nextDouble();
   }
}
```

To enable this code for Multi-Sandbox, you can simply declare the static field local to the context. For that, create a `kernel.intern` file at the root of the library or Kernel classpath (e.g., in the `src/main/resources` directory) with the following content:

```
<kernel>
   <contextLocalStorage name="java.lang.Math.RandomGenerator"/>
</kernel>
```

When the method is called in a new context, the static field is read to `null`, and then a new object will be allocated and assigned to the local static field. Thus, each context will create its own instance of the `Random` singleton on demand.

Note: By default, reading a static field for the first time in a new context returns `null`. However, it is possible to write dedicated code to initialize the static field before its first read access. See section *§4.3.3 Context Local Static Field References* of the *Kernel & Features Specification* for more details.

Allow a Field Assignment in Kernel Mode

It is possible to assign a Feature object in a static field or an instance field owned by the Kernel only if the Kernel owns the current context. Such an assignment must be removed before stopping the Feature. The common way is to register a FeatureStateListener at Kernel boot. This gives a hook to remove Kernel links to Feature objects when a Feature moves to the `STOPPED` state.

```
Kernel.addFeatureStateListener(new FeatureStateListener() {

   @Override
   public synchronized void stateChanged(Feature feature, State previousState) {
      if (feature.getState() == State.STOPPED) {
         // Here, remove Kernel->Feature references
      }
   }
};
```

Without this, the Feature will remain in the `STOPPED` state. Therefore, it will not be possible to uninstall it or start it again until the link is removed. The remaining Feature objects referenced by the Kernel are called Kernel stale references.

Note: To help debug your Kernel, Kernel stale references are displayed by the *Core Engine dump*.

Use Existing Multi-Sandbox Enabled Data Structures

MicroEJ Corp. provides ready-to-use classes on the shelf that are Multi-Sandbox enabled. Among them, we can cite the following:

- `KernelObservable` : Implementation of Observable that can handle observers from any Module.
- `KFList` : Implementation of a Collection with multi-context support.
- `SharedPropertyRegistry` : Map of key/value properties.
- `SharedServiceRegistry` : Map of API/implementation services.

Please contact *our support team* for more details on usage.

7.10.2 Implement a Security Manager Permission Check

A Multi-Sandbox enabled Foundation Library should protect Feature from accessing native resources. This is done by requesting a check to the current SecurityManager defined by the Kernel.

The following code is the typical code that must be written at the beginning of API methods.

```
void myAPIThatOpensAccessToANativeResource(){

   if (Constants.getBoolean("com.microej.library.edc.securitymanager.enabled")) {
      // Here, the Security Manager support is enabled.

      SecurityManager securityManager = System.getSecurityManager();
      if (securityManager != null) {
         // Here, the Kernel has registered a Security Manager

         // Create a Permission with relevant parameters for the Security Manager to render␣
↪the permission
         MyResourcePermission p = new MyResourcePermission();

         // Request the permission check.
         // If the Kernel rejects the permission, it will throw a SecurityException
         securityManager.checkPermission(p);
      }
   }

   // Implementation code
   // ...

}
```

Note: The code is wrapped by a static check of the *Option(checkbox): Enable SecurityManager checks*. By default, this option is disabled, so the SOAR automatically removes the code. This allows you to use your library in a Mono-Sandbox environment where ROM footprint matters. Your Kernel shall enable this option to trigger the Security Manager checks. See *Implement a Security Policy* for more details.

7.10.3 Known Foundation Libraries Behavior

This section details the Multi-Sandbox semantic that has been added to Foundation Libraries in order to be Multi-Sandbox enabled. Most of the Foundation Libraries provided by MicroEJ Corp. are Multi-Sandbox enabled unless the library documentation (e.g., `README.md`) mentions specific limitations.

MicroUI

Note: This chapter describes the current MicroUI version `3`, provided by UI Pack version `13.0.0` or higher. If you are using the former MicroUI version `2` (provided by MicroEJ UI Pack version up to `12.1.x`), please refer to this MicroEJ Documentation Archive.

Physical Display Ownership

The physical display is owned by only one context at a time (the Kernel or one Feature). The following cases may trigger a physical display owner switch:

- during a call to Display.requestShow(Displayable), Display.requestHide(Displayable), Display.requestRender() or Display.requestFlush(): after the successful permission check, it is assigned to the context owner.
- during a call to MicroUI.callSerially(Runnable): after the successful permission check it is assigned to owner of the `Runnable` instance.

The physical display switch performs the following actions:

- If a Displayable instance is currently shown on the `Display`, the method Displayable.onHidden() is called,
- All pending events (input events, display flushes, call serially runnable instances) are removed from the display event serializer,
- System Event Generators handlers are reset to their default EventHandler instance,
- The pending event created by calling Display.callOnFlushCompleted(Runnable) is removed and will be never added to the display event serializer.

Warning: The display switch is performed immediately when the current thread is the MicroUI thread itself (during a MicroUI event such as a MicroUI.callSerially(Runnable)). The caller looses the display and its next requests during same MicroUI event will throw a new display switch. Caller should call future display owner's code (which will ask a display switch) in a dedicated MicroUI.callSerially(Runnable) event.

The call to Display.callOnFlushCompleted(Runnable) has no effect when the display is not assigned to the context owner.

Automatically Reclaimed Resources

Instances of ResourceImage and Font are automatically reclaimed when a Feature is stopped.

BON

Kernel Timer

A Kernel Timer instance can handle TimerTask instances owned by the Kernel or any Features.

It should not be created in *clinit code*, otherwise you may have to manually declare *explicit clinit dependencies*.

Automatically Reclaimed Resources

TimerTask instances are automatically canceled when a Feature is stopped.

ECOM

The ej.ecom.DeviceManager registry allows to share devices across Features. Instances of ej.ecom.Device that are registered with a Shared Interface type are made accessible through a Proxy to all other Features that embed the same Shared Interface (or an upper one of the hierarchy).

ECOM-COMM

Instances of ej.ecom.io.CommConnection are automatically reclaimed when a Feature is stopped.

FS

Instances of java.io.FileInputStream, java.io.FileOutputStream are automatically reclaimed when a Feature is stopped.

NET

Instances of java.net.Socket, java.net.ServerSocket, java.net.DatagramSocket are automatically reclaimed when a Feature is stopped.

SSL

Instances of javax.net.ssl.SSLSocket are automatically reclaimed when a Feature is stopped.

7.11 Setup a KF Test Suite

A KF test suite can be executed when building a Foundation Library or an Add-On library, and usually extends the tests written for the *default library test suite* to verify the behavior of this library when its APIs are exposed by a Kernel.

A KF test suite is composed of a set of KF tests, each KF test itself is a minimal Multi-Sandbox Executable composed of a Kernel and zero or more Features.

7.11.1 Enable the Test Suite

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="+" />
```

- Configure the option `artifacts.resolver` to the name of the resolver used to import KF test dependencies. The name must be one of the resolver names defined in your *settings file*. If you are using the default settings file, set the option to `MicroEJChainResolver` . This option is usually set as a global *MMM option*.

7.11.2 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 directory for the Kernel project and initialize it using the `microej-javalib` *skeleton*,
  - Create a new directory for the Feature project and initialize it using the `application` *skeleton*,
  - Create a new directory for the Firmware project and initialize it using the `firmware-multiapp` *skeleton*.

The names of the project directories are free, however MicroEJ suggests the following naming convention, assuming the KF test directory is `[TestName]` :

- `[TestName]-kernel` for the Kernel project,
- `[TestName]-app[1..N]` for Feature projects,
- `[TestName]-firmware` for the Firmware project.

The KF Test Suite structure shall be similar to the following figure:

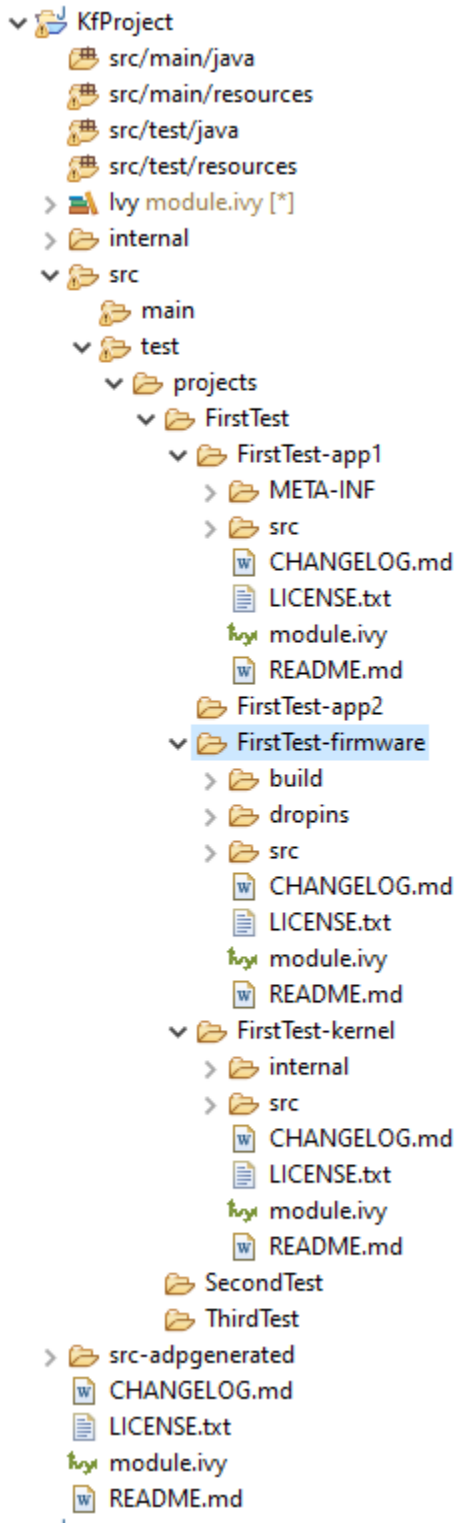


Fig. 6: KF Test Suite Overall Structure

All the projects will be built automatically in the right order based on their dependencies.

7.11.3 KF Test Suite Options

It is possible to configure the same options defined by *Test Suite Options* for the KF test suite, by using the prefix `microej.kf.testsuite.properties` instead of `microej.testsuite.properties`.

7.12 Kernel Linking

This chapter describes how a Kernel Application is linked.

Basically, a Kernel Application is linked as a Standalone Application. The main difference is that a Kernel Application defines *Kernel APIs*, and requires to embed additional information that will be used later to build a Sandboxed Application against this Kernel (by linking with the Kernel APIs). Such additional information is called the Kernel Metadata.

7.12.1 Link Flow

The following figure shows the general process of linking an Executable, applied to a Kernel Application.

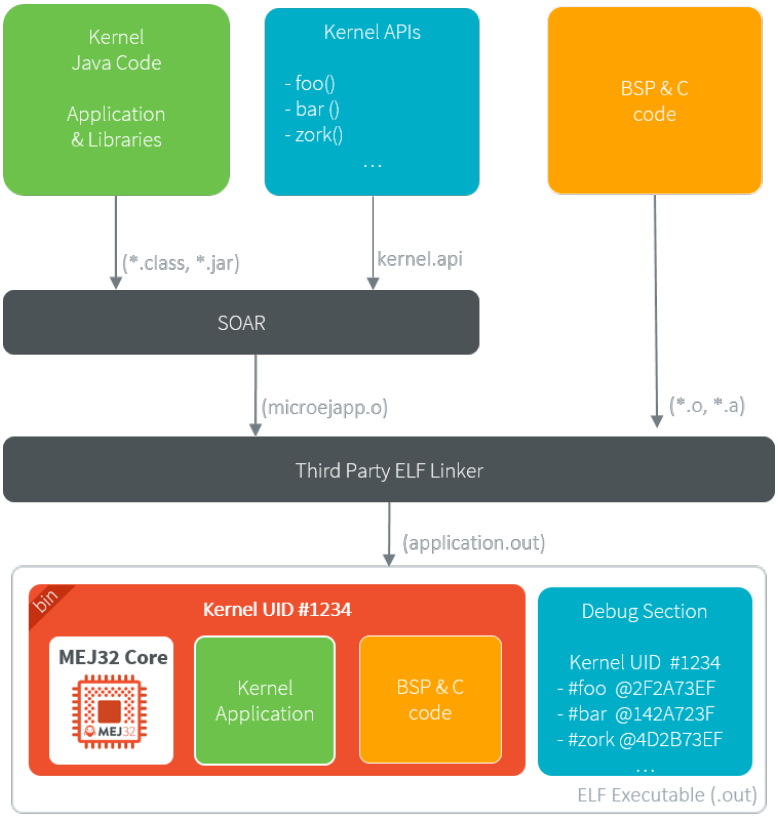


Fig. 7: Kernel Link Flow

The Platform must be configured with *Multi-Sandbox capability*.

By default, the Kernel Metadata is included in the `.debug.soar` section which also serves for debug purpose (*Stack Trace Reader*, *Heap Dumper*). Particularly, it contains resolved absolute addresses of Kernel APIs.

7.12.2 Kernel Metadata Generation

To *build a Sandboxed Application on Device*, the Kernel Metadata must be exported after the Firmware link from the `.debug.soar` section of the executable. This step is not necessary to *build a Sandboxed Application Off Board*.

The Kernel Metadata can be exported from an existing Firmware executable file by using the Kernel Metadata Generator tool. It produces a `.kdat` file that will be used to link the Sandboxed Applications on device.

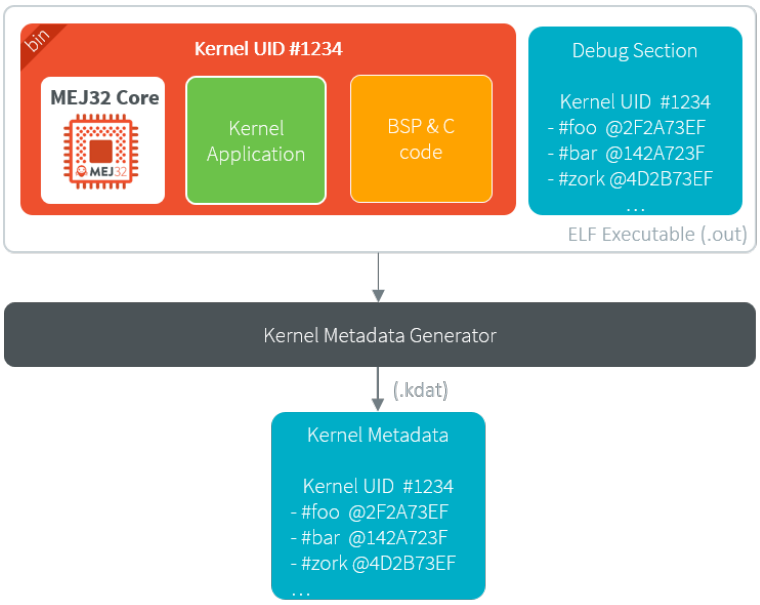


Fig. 8: Kernel Metadata Generator

The `.kdat` file is optimized for size. When linking a Sandboxed Application `.fso` file, only the required metadata will be loaded in Java heap. It will be loaded from a standard InputStream, so that it can be stored to a memory that is not accessible from the CPU's address space.

Note: The Kernel Metadata `.kdat` file can also be integrated in a Firmware executable file using post-link tools such as binutils objcopy, provided a dedicated section has been reserved by the third-party linker.

7.12.3 Feature Portability Control

A Kernel can install `.fo` files that have been built on other Kernels, provided this Kernel complies with other Kernels according to a set of rules declared hereafter. This is called *Feature Portability Control*, as the verification is performed during the new Kernel build, with no impact on the Feature dynamic installation.

Principle

During a Kernel build, SOAR can verify this Kernel preserves the portability of any `.fo` files built on a previous Kernel using *the Kernel metadata file*. If the portability is preserved, the *UID* of the previous Kernel is embedded in the new Kernel, allowing `.fo` files built on the previous Kernel to be installed as well. Otherwise, SOAR fails with an error indicating the broken rule(s).

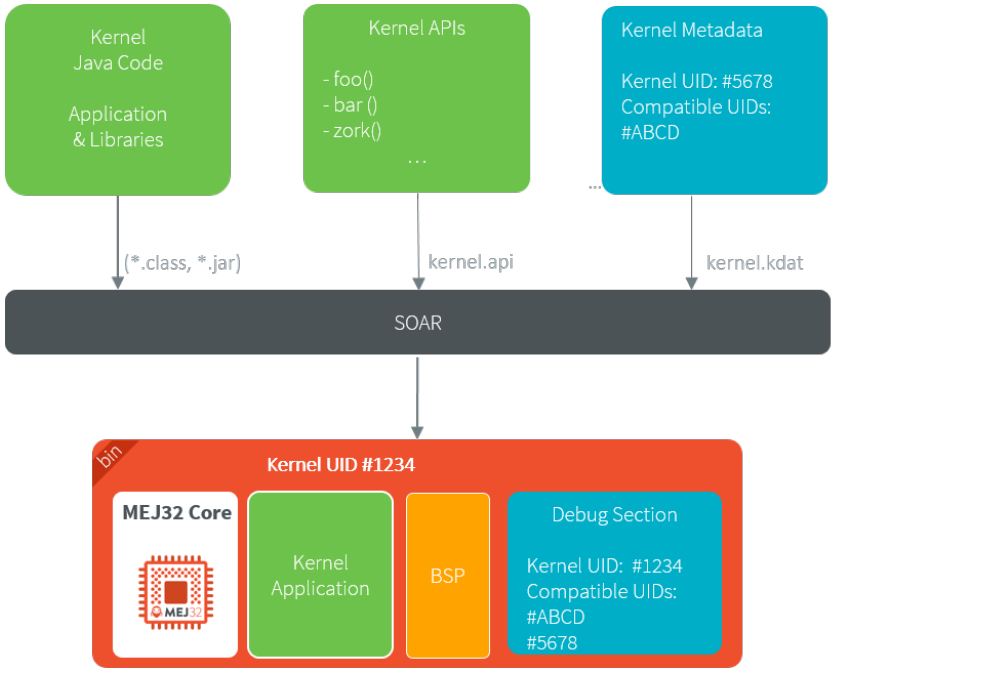


Fig. 9: Feature Portability Control Principle

Enable

Note: This is a new functionality that requires Architecture *8.0.0* or higher.

Add the following *Application Options* to your Kernel project:

- `com.microej.soar.kernel.featureportabilitycontrol.enabled` : `true` to enable Feature Portability Control. Any other value disables Feature Portability Control (the following option is ignored).
- `com.microej.soar.kernel.featureportabilitycontrol.metadata.path` : Path to the Kernel Metadata file (`.kdat` file).

Portability Rules

A Kernel Application can install a `.fo` file that has been built against another Kernel Application if the Kernel Application code has not changed or if the modifications respect the portability rules. Here is the list of the modifications that can be done while preserving the portability:

- Modify method code, except if *Method Devirtualization* or *Method Inlining* has changed.
- Add a new type (including declared as Kernel API),
- Add a new static method (including declared as Kernel API),
- Add a new instance method in a type **not declared** as Kernel API,
- Add a new instance method with `private` visibility in a type **declared** as Kernel API,
- Add a new static field (including declared as Kernel API),
- Add a new instance field in a type **not declared** as Kernel API,
- Rename an instance field with `private` visibility in a type **declared** as Kernel API,
- Modify a Java type, method, or static field **not declared** as Kernel API (code, signature, hierarchy)
- Remove a Java type, method, or static field **not declared** as Kernel API

Both Kernel Applications must be built from Platforms based on the same Architecture version.

Any other modifications will break the Feature portability. For example, the following modifications will not preserve the portability:

- Remove a Java type, method or static field **declared** as Kernel API,
- Add or remove an instance method in a type **declared** as Kernel API, even if the method is **not declared** as Kernel API,
- Add or remove an instance field in a type **declared** as Kernel API,
- Modify method or field signature **declared** as Kernel API (name, declaring type, static vs instance member, ...),
- Modify hierarchy of a type **declared** as Kernel API.

7.13 Application Linking

This chapter describes how a Sandboxed Application is built so that it can be (dynamically) installed on a Kernel. The build output file of a Sandboxed Application against a Kernel is called a Feature, hence the `f` letter used in the extension name of the related files (`.fso` and `.fo` files).

7.13.1 SOAR Build Phases

When building a Sandboxed Application to a Feature, SOAR processing is divided in two phases:

1. **SOAR Resolver**: loads the set of application `.class` files and resources. Among the various steps, mention may be made of:
   - Computing the transitive closure from the application entry points of all required elements (types, methods, fields, strings, immutables, resources, system properties),
   - Computing the *clinit order*.

The result is an object file that ends with `.fso` extension. The `.fso` file is a portable file that can be linked on any compatible Kernel (see *FSO Compatibility*).

2. **SOAR Optimizer**: links a `.fso` file against a specific Kernel. Among the various steps, mention may be made of:

   - Linking to the expected Kernel APIs (types, methods, fields) according to the JVM specification[1],
   - Generating the MEJ32 instructions,
   - Building the virtualization tables.

   The result is an object file that ends with the `.fo` extension. By default, the `.fo` file is specific to a Kernel: it can only be installed on the Kernel it has been linked to. Rebuilding a Kernel implies to run this phase again, unless the application has been built for the previous Kernel (see *Feature Portability*).

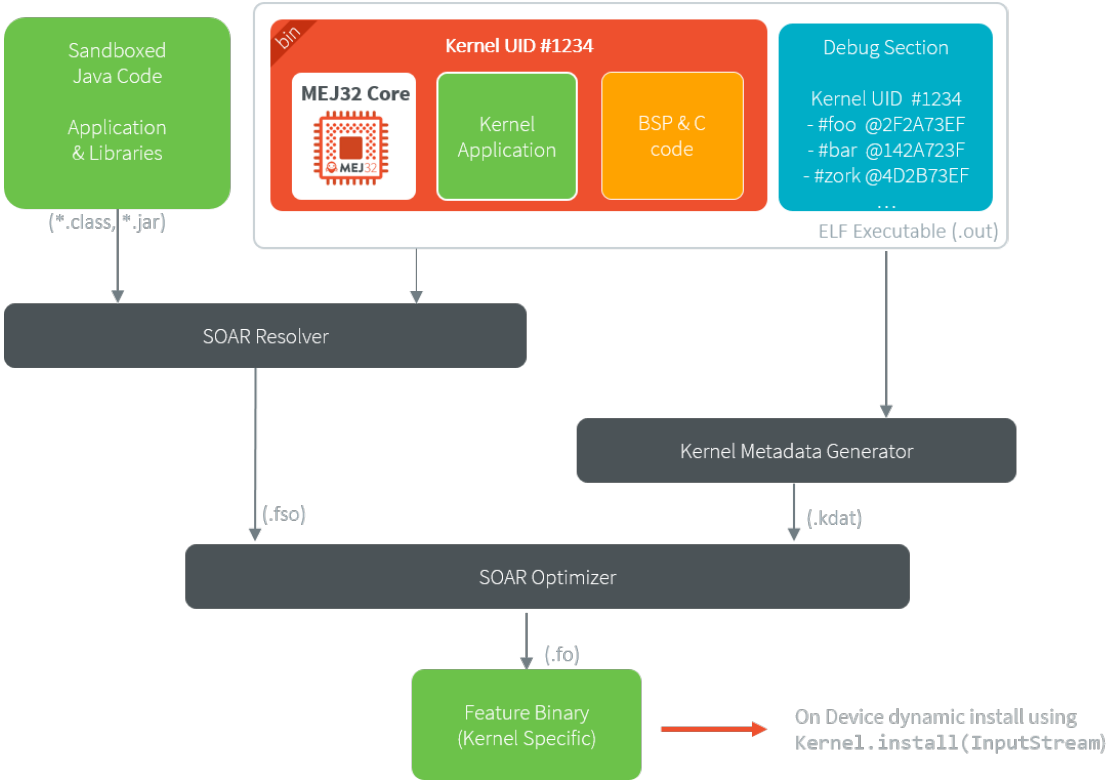


Fig. 10: Sandboxed Application Build Flow

The Feature `.fo` file can be deployed to the Device using Kernel.install() method.

[1] Tim Lindholm & Frank Yellin, The Java™ Virtual Machine Specification, Second Edition, 1999

7.13.2 Feature Build Off Board

A Sandboxed Application can be built to a Feature (`.fo` file) using a *MicroEJ Application Launch* configured as follows:

- Set the `Settings` field in the `Execution` tab to `Build Dynamic Feature` .
- Set the `Kernel` field in the `Configuration` tab to a Multi-Sandboxed Firmware (`.out` ELF executable file).

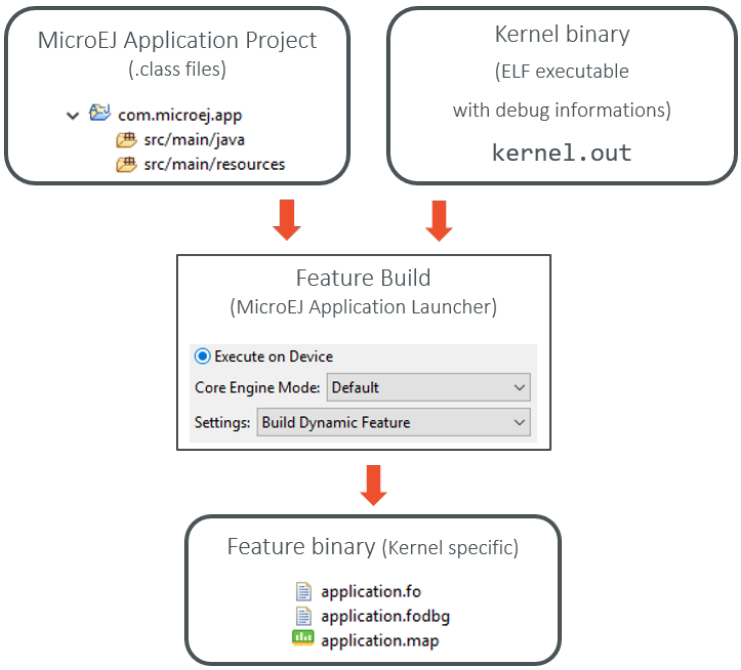


Fig. 11: Feature Build Flow using MicroEJ Launch

7.13.3 Feature Build On Device

Note: This is a new functionality that requires a custom Architecture configuration. Please contact *our support team* for more details.

The SOAR Optimizer is packaged to a Foundation Library named `SOAR` , thus this phase can be executed directly on Device.

General Workflow

Here are the typical steps to achieve:

- Build the Sandboxed Application on any compatible Kernel to get the `.fso` file,
- Transfer the `.fso` file on Device by any mean,
- Generate the *Kernel Metadata* for the Kernel on which the `.fso` file is being linked,
- Transfer the `.kdat` file on Device by any mean,
- Write a MicroEJ Standalone Application for building the `.fso` file:

- implement a `com.microej.soar.KernelMetadataProvider` to provide an InputStream to load the `.kdat` file,
- provide an InputStream to load the `.fso` file,
- provide an OutputStream to store the `.fo` file,
- call `FeatureOptimizer.build()` method.

Then the `.fo` file can be dynamically installed using Kernel.install().

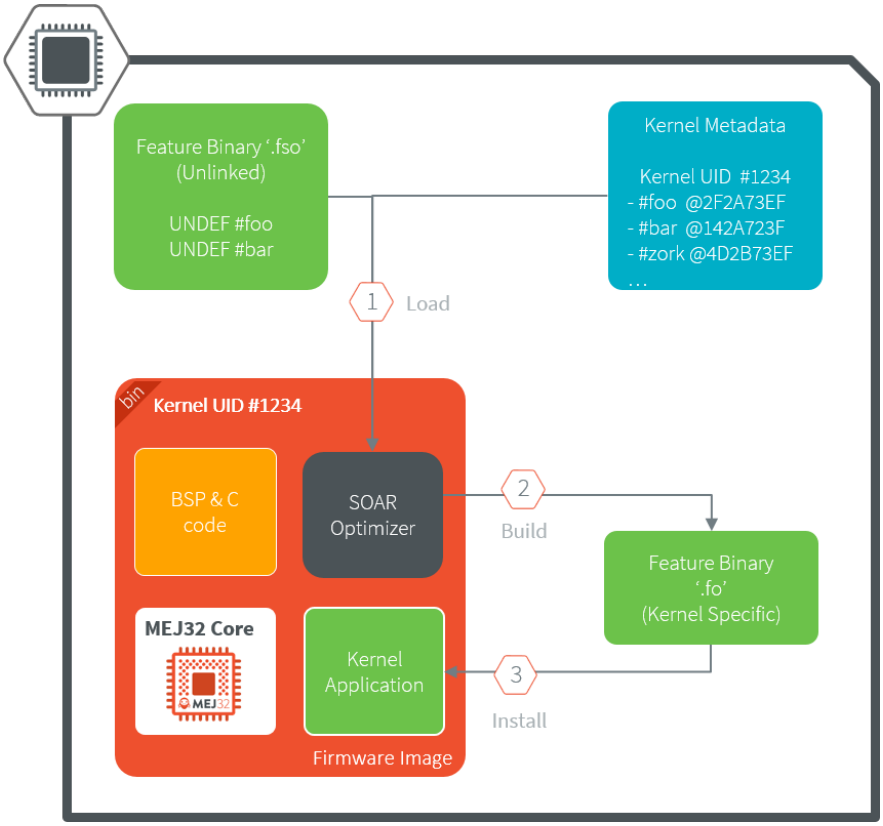


Fig. 12: Sandboxed Application Build on Device

Note: Although this is common, it is not required to run the SOAR Optimizer phase on the Kernel that will dynamically install the `.fo`. There is no relationship between `SOAR` and `KF` Foundation Libraries.

Implement the Kernel

SOAR Optimizer can be integrated on any Standalone Application providing the following *module dependencies*:

```
<dependency org="ej.api" name="edc" rev="1.3.3" />
<dependency org="com.microej.api" name="soar" rev="1.0.0" />
<dependency org="ej.library.eclasspath" name="collections" rev="1.4.0" />
```

The following code template illustrates the usage of the `SOAR` Foundation Library:

```
package com.microej.example;

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

import com.microej.soar.FeatureOptimizer;
import com.microej.soar.FeatureOptimizerException;
import com.microej.soar.KernelMetadataProvider;

/**
* This is a template code that shows the typical steps to follow for building a
* .fo file from a .fso file on Device.
*/
public class TemplateFSOBuild {

   /**
   * Your Platform specific {@link KernelMetadataProvider} implementation.
   */
   private static final class MyKernelMetadataProvider implements KernelMetadataProvider {

      @Override
      public InputStream openInputStream(int offset) throws IOException {
         // Return an InputStream to the Kernel Metadata resource (.kdat file) at the given␣
↪offset in bytes.
         return null; // TODO
      }

      @Override
      public String toString() {
         // Here, return a printable representation of this Kernel Metadata Provider (for␣
↪debug purpose only)
         return "Kernel Metadata loaded from ..."; // TODO
      }
   }

   /**
   * A method that builds a .fso file to a .fo file.
   */
   public static void build() {
      // Create the KernelMetadataProvider instance
      KernelMetadataProvider kernelMetadataProvider = new MyKernelMetadataProvider();

      // Load the .fso InputStream
      InputStream fsoInputStream = null; // TODO

      // Prepare the target OutputStream where to store the .fo
      OutputStream foOutputStream = null; // TODO

      // Create the FeatureOptimizer instance
      FeatureOptimizer featureOptimizer;
      try {
```

(continues on next page)

(continued from previous page)

```
            featureOptimizer = new FeatureOptimizer(kernelMetadataProvider);
        } catch (FeatureOptimizerException e) {
            // Handle Kernel Metadata cannot be loaded
            e.printStackTrace(); // TODO
            return;
        }

        // Build
        try {
            featureOptimizer.build(fsoInputStream, foOutputStream);
        } catch (FeatureOptimizerException e) {
            // Handle .fso cannot be built to .fo
            e.printStackTrace(); // TODO
        }
    }
}
```

7.13.4 FSO Compatibility

A `.fso` file can be linked on any Kernel providing all the following conditions:

- its Architecture has the same endianness than the Architecture on which the `.fso` file has been produced,
- its Architecture version is compatible[2] with the Architecture version on which the `.fso` file has been produced,
- it provides the required APIs according to the JVM specification[Page 1201, 1].

A current limitation is that if the Sandboxed Application declares an immutable object, SOAR Optimizer will resolve fields within the same class rather than considering the entire class hierarchy.

7.13.5 Feature Portability

By default, a `.fo` file can only be installed on the Kernel on which it has been linked.

Starting from *Architecture 8.0.0*, the same Feature file can be installed on different Kernels. This is called *Feature Portability*. Thus it is not required to rebuild the `.fo` file in the following cases:

- Relinking the executable (memory layout changes),
- Recompiling the C code,
- Rebuilding the Kernel Application, if *Feature Portability Control* has been enabled.

[2] New version is greater than or equals to the old one within the same major version.

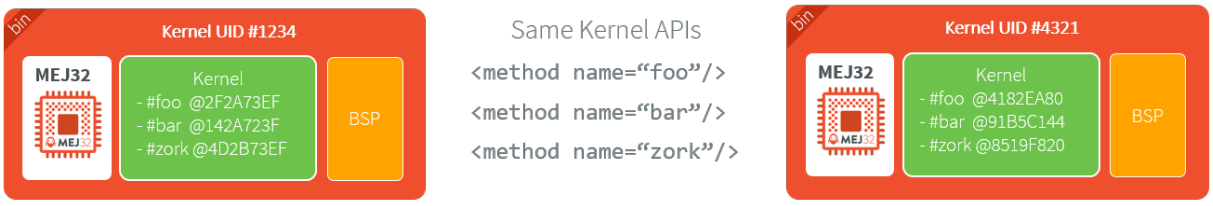


Fig. 13: Feature Portability Overview

CHAPTER

EIGHT

VEE WEAR USER GUIDE

VEE Wear is a specialized application framework tailor-made for next-gen smartwatches. Engineered with a highly optimized memory footprint, VEE Wear ensures efficient performance on low-power MCUs and MPUs while offering robust features comparable to larger operating systems.

VEE Wear includes all graphical features from MicroEJ runtime, including:

- A thread-safe 2D graphics engine
- A widget framework
- Vector drawing support
- And more. Refer to the *Graphical User Interface* section for more info

Android Compatibility Kit

The *Android Compatibility Kit* comprises two technologies:

1. Support for Android development tools, including Android Studio and Gradle, is provided by MICROEJ SDK 6.
2. Support for MicroEJ applications on the Android OS (Android Runtime).

The support provided by Android development tools and Android runtime is especially beneficial when running the same application on both an Android processor and an MCU. This scenario occurs, for example, when implementing Android off-loading with a big-little architecture.

Moreover, the Android runtime enables the execution of the same application on an Android smartphone. For instance, in scenarios such as building a watch face or app picker within a companion smartphone app, the MicroEJ application code can be directly utilized to display the app or watch face. This eliminates the necessity to develop a similar version specifically for Android.

iOS Compatibility Kit

The *iOS Compatibility Kit* feature provides support for MicroEJ applications on iOS (iOS Runtime). This runtime enables the execution of the same application on an iOS smartphone. For example, when creating a watch face or app picker within a companion smartphone app, the MicroEJ application code can be directly used to display the app or watch face. This eliminates the necessity to develop a separate version for iOS.

MicroEJ's offloading framework

VEE Wear incorporates an *offloading framework* designed to optimize power consumption in a big-little architecture, where an application processor runs Android, and a companion MCU operates MICROEJ VEE. By alternating the execution of applications between low-consumption MCUs and powerful MPUs, this setup guarantees maximum power efficiency, thereby conserving battery life. The offloading framework encompasses an inter-processor communication framework and efficient low-power profile management.

Low Power Facer Engine

VEE Wear supports the Facer Engine, expanding Facer's extensive watch face catalog of 500,000 faces across all smartwatches, including low-power RTOS watches, enriching the user experience for all users. For further information about the MicroEJ and Facer partnership, please contact your MicroEJ sales representative.

8.1 Android Compatibility Kit

MicroEJ provides a set of tools and libraries to run applications powered by MicroEJ on Android and Wear OS. This allows for the same application to be developed, simulated, tested, and executed on MicroEJ VEE and Android alike.

Having the same code ready for both Android and MicroEJ VEE opens up a wide range of use cases, including but not limited to:

- Develop derivative products based on small MCUs or low-cost MPUs with limited resources where Android cannot be used as it is inherently resource-intensive.
- Reduce energy consumption by enabling two processors to coexist to distribute tasks between a very powerful processor powered by Android and a low-power processor powered by MicroEJ.

Below are some examples from the wearable segment that illustrate these use cases:

- Watch faces can be developed once and deployed on both a smartwatch (MicroEJ VEE) and its companion smartphone app (Android), enabling consistent functionality and appearance across both devices. This provides a good user experience for the user while minimizing code duplication and maintenance for the developer.
- Power efficiency is a significant concern with wearable as sophisticated features often come at a high cost in terms of power consumption. An *offloading* framework can reduce power usage by executing the same application alternately on a low-consumption MCU and a powerful MPU. Operating in standby mode as much as possible on the MCU is one of the strategies to achieve energy efficiency.

8.1.1 Overview

The Android Compatibility Kit is composed of two main components:

- A runtime: applications developed on MicroEJ can run on the Android platform thanks to the Android-based implementation of the MicroEJ Foundation libraries and dedicated support libraries.
- A developer kit: the MICROEJ SDK 6 and a Gradle plugin provide the necessary support for developing applications in Android Studio using Gradle.

Workflow

Below is a general overview of the workflow when developing a product that targets both MicroEJ and Android-powered devices.

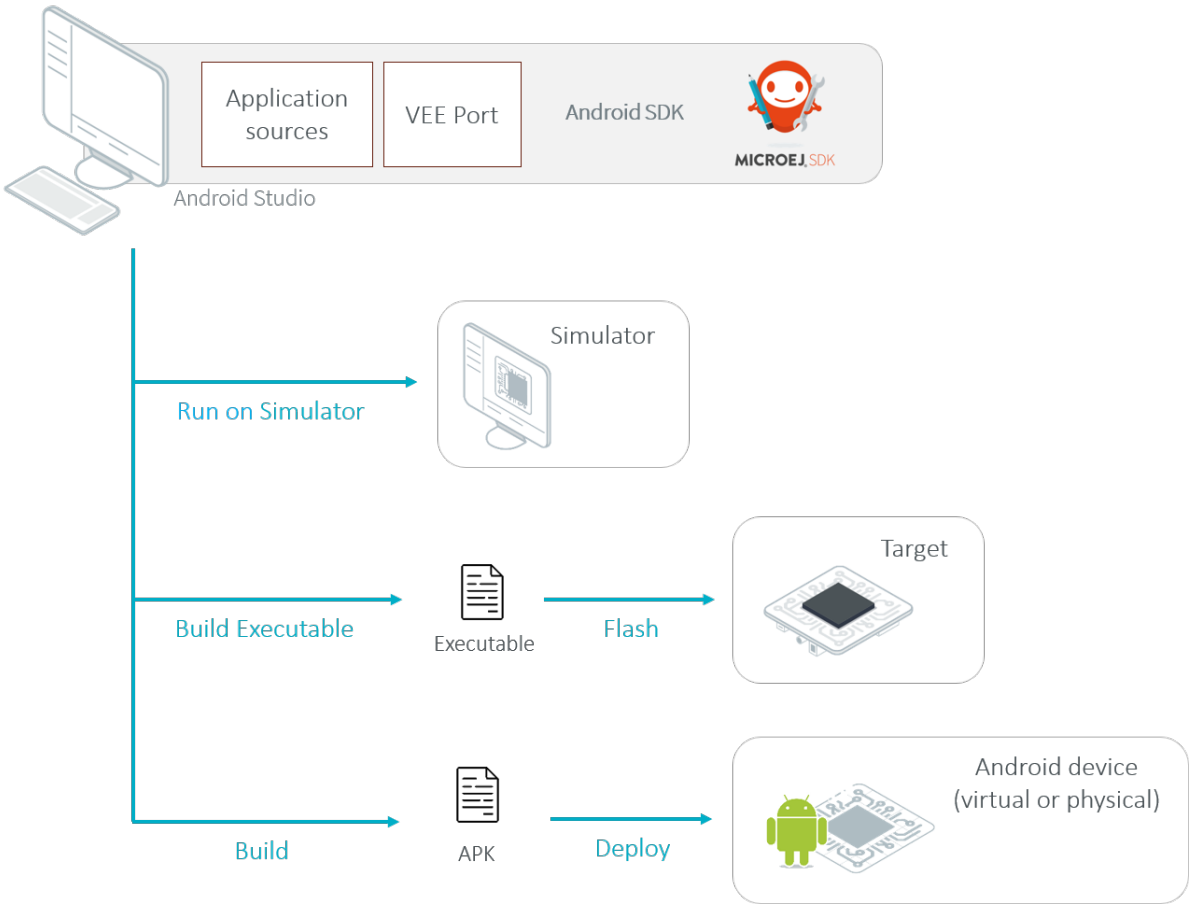


Fig. 1: Workflow Diagram

Software Architecture

Applications designed to run in MicroEJ VEE can also run on Android, thanks to a specific implementation of MicroEJ Foundation libraries based on Android libraries.

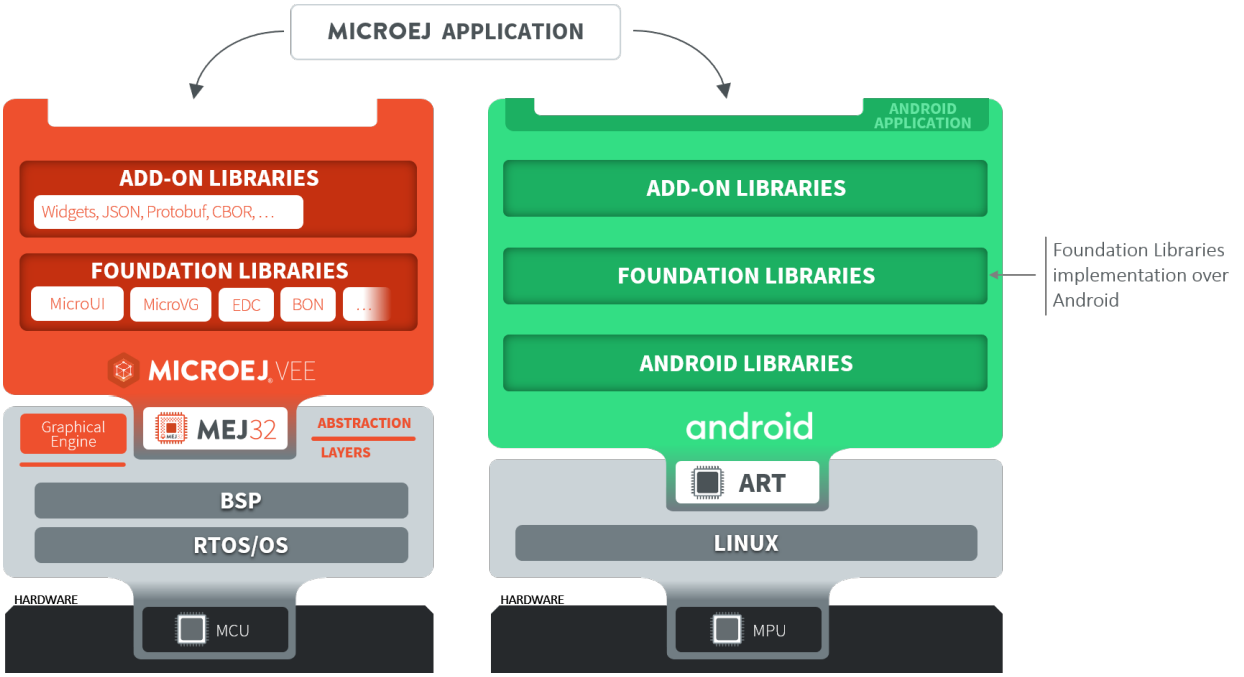


Fig. 2: Software Architecture

Available APIs and Features

- Foundation Libraries
  - EDC
  - BON
  - SNI
  - MICROUI
  - MICROVG
  - TRACE
- All compatible Add-on Libraries
- Supported Resources
  - Images
  - Fonts (EJF, TTF, OTF)
  - Android Vector Drawables (AVD)
  - SVG
  - NLS (including Android XML and PO formats)
  - Constants

  - Properties
- Custom native APIs can be implemented over Android libraries to make them executable on Android devices.

8.1.2 Installation

To develop applications compatible with both Android and MicroEJ VEE, it is required to use *MICROEJ SDK 6*. The MICROEJ SDK provides Gradle plugins that allow for seamless integration in Android Studio.

For comprehensive installation instructions, read the *SDK 6 Installation Guide*. Follow the steps related to Android Studio when relevant.

In addition to installing MICROEJ SDK 6, make sure to comply with the requirements listed below.

JDK Version

The Android Compatibility Kit, like MICROEJ SDK 6, is compatible with a JDK 11 or higher LTS version. The JDK version to use will depend on the Android Gradle Plugin (AGP) being used by your Android project. Starting from version 8, AGP requires a minimum JDK 17. If you intend to use JDK 11, you will need to specify a compatible version of AGP (e.g., 7.4.2). Refer to the Android Gradle plugin release notes for more information.

Configure Repositories

The *SDK 6 repositories configuration* references the MicroEJ module repositories which are required for resolving the SDK Gradle plugins and modules. Working with Android plugins and modules involves extending this configuration to include additional repositories that are essential for Android development.

- download and copy `this file` in `<user.home>/.gradle/init.d/`.

8.1.3 Project Setup

This chapter will guide you through the process of creating a project for having an application compatible with both Android and MicroEJ VEE.

The recommended project structure to get started is to have a basic multi-project build that contains a root project and two subprojects: one subproject for the Android/Wear OS application and one subproject for the MicroEJ Application. The MicroEJ Application defines code that will run on both MicroEJ VEE and Android, while the Android application includes wrapper code and logic specific to Android.

What follows is the directory and file structure of a typical project:

```
├── android-app/
│   ├── src
│   ├── microej.properties       # MicroEJ Application Options for Android/Wear OS
│   └── build.gradle.kts
├── microej-app/
│   ├── src
│   ├── configuration/
│   │   └── common.properties    # MicroEJ Application Options for MicroEJ VEE
│   └── build.gradle.kts
├── build.gradle.kts
└── settings.gradle.kts
```

Create or Import an Android project

The Android documentation covers the process of creating apps for diverse form factors, including smartphones and wearable devices. Read Create a Project and follow the guidelines before proceeding. If you are creating a project from scratch, we recommend using the `Empty Activity` template.

Note: The project template in Android Studio defines a default `repositories` configuration in the `settings.gradle.kts` file of the project like below:

```
pluginManagement {
   repositories {
      google()
      mavenCentral()
      gradlePluginPortal()
   }
}
dependencyResolutionManagement {
   repositoriesMode.set(RepositoriesMode.FAIL_ON_PROJECT_REPOS)
   repositories {
      google()
      mavenCentral()
   }
}
```

Please note that this will override the `repositories` configuration defined during the *installation process*, based on Gradle initialization scripts. Set the configuration according to your preference, but we suggest removing these lines at first from the settings file to get started.

Assuming that a Gradle project with an Android application is now opened in Android Studio, do the following:

- Open the `build.gradle.kts` file at the root of the project.
- Add these lines to the `plugins` block:

```
id("com.microej.gradle.application") version "0.15.0" apply false
id("com.microej.android.gradle.plugins.android") version "0.3.6" apply false
```

Create or Import a MicroEJ Application

The next step is adding the module that contains the MicroEJ Application to the Gradle project.

Create a MicroEJ Application

Import an existing MicroEJ Application

- Click on File > New > New Module... .
- Select Java or Kotlin Library .
- Set the name of the module in the Library Name field.
- Set the package name of the module in the Package name field.
- Enter a name for the main Java class of the application in the Class name field.
- Select the language `Java` in the Language field.

- Select `Kotlin DSL` in the `Build configuration language` field.
- Click on `Finish` .

The module created by Android Studio is a standard Java module (Gradle `java-library` plugin). The `build.gradle.kts` file has to be updated to make it a MicroEJ Application module:

- Open the `build.gradle.kts` file.
- Erase its whole content.
- Add the `com.microej.gradle.application` plugin in the `build.gradle.kts` file:

```
plugins {
   id("com.microej.gradle.application")
}
```

- Add the following `microej` block in the `build.gradle.kts` file:

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

  where the property `applicationMainClass` is set to the Full Qualified Name of the main class of the application. This class must define a `main()` method and is the entry point of the application.
- Declare the dependencies required by your application in the `dependencies` block of the `build.gradle.kts` file. The EDC library is always required in the build path of an Application project, as it defines the minimal runtime environment for embedded devices:

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

If you have already developed a MicroEJ Application, you can import it in the project.

Note: If the MicroEJ Application has been created with the *SDK 5* or lower, it is required to first migrate it to *SDK 6*. Read the comprehensive *Migration Guide* before proceeding.

- Click on `File` > `New` > `Import Module...` .
- Browse to the source directory of the Gradle project.
- Set the module name.
- Click on `Finish` .

  Note: Android Studio may use the Groovy DSL to include the imported module. The result is the creation of a `setting.gradle` file that shadows the configuration in the `settings.gradle.kts` file. If that occurs, merge the relevant content of the `setting.gradle` file into the existing `settings.gradle.kts` and remove the `setting.gradle` .

- Set the `microejConflictResolutionRulesEnabled` property to `false` in the `build.gradle.kts` file:

```
microej {
   microejConflictResolutionRulesEnabled = false
   ...
}
```

Note: The MicroEJ Gradle plugin comes with additional conflict resolution rules compared to Gradle's default behavior. This can make the build fail when working with Android dependencies, so it is recommended to use Gradle's default conflict management in this case. These extra rules can be disabled by setting the `microejConflictResolutionRulesEnabled` property to `false` in the `microej` configuration block. Read *Manage Resolution Conflicts* for more details.

- Ensure that the Gradle settings file includes the Android and MicroEJ modules, like in this example:

```
include(":android-app")
include(":microej-app")
```

- To synchronize your project files, select Sync Now from the notification bar that appears after making changes.

When the Gradle project has been reloaded, it should compile successfully, without any error.

Configure the Android Application

The next steps show how to configure the Android or Wear OS application to declare the MicroEJ Application.

- Open the `build.gradle.kts` file of the Android application.
- Add the `com.microej.android.gradle.plugins.android` plugin:

```
plugins {
   id("com.android.application")
   id("com.microej.android.gradle.plugins.android")
   ...
}
```

- Add a dependency to the MicroEJ support library depending on the target (Android or Wear OS).

Android

Wear OS

```
dependencies {
   implementation("com.microej.android.support:microej-application:2.0.1")
   ...
}
```

The support library `microej-application` allows running a MicroEJ Application in an Android Activity using the MicroEJ support engine.

```
dependencies {
   implementation("com.microej.android.support:microej-wearos:2.0.1")
   implementation("androidx.wear.watchface:watchface:1.1.1")
   implementation("androidx.wear.watchface:watchface-guava:1.1.1")
   ...
}
```

The support library `microej-wearos` allows running a MicroEJ Application in a Wear OS WatchFaceService using the MicroEJ support engine.

- Add a dependency to the MicroEJ Application using the `microejApp` configuration, for example:

```
dependencies {
   microejApp(project(":microej-app"))
   ...
}
```

  where `microej-app` is the name of the subproject that contains the MicroEJ Application.

- Add a dependency to a VEE Port, for example:

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

  There are multiple options for providing a VEE Port in your project. Read *Select a VEE Port* to explore the available options.

  Note: It is required to select a VEE Port that's configured to build MicroEJ Applications for Android. Read the *VEE Port section* to learn how to configure a VEE Port for this purpose.

- Add a file named `microej.properties` at the root of the Android application. This file sets the MicroEJ Application Options when running on Android. It is similar in principle to *defining Application Options* for the embedded device. Depending on the target device (Android or embedded device), the content may differ.
- Select Sync Now from the notification bar to synchronize your project files.

Start the MicroEJ Application

The final step involves calling the entry point of the MicroEJ Application from within the Android or Wear OS application.

Android

Wear OS

Assuming that the Android application declares an activity in the `AndroidManifest.xml` :

- Open the corresponding activity Java/Kotlin file.
- Make `MicroEJActivity` the superclass of this class.
- Override the method `getApplicationMainClass()` and make it return the Full Qualified Name of the main class of the MicroEJ Application.

This is an example of a simple activity:

Kotlin

Java

```
class MainActivity : MicroEJActivity() {
   override fun getApplicationMainClass(): String {
      return "com.mycompany.Main";
```

(continues on next page)

(continued from previous page)

```
    }
}
```

```
public class MainActivity extends MicroEJActivity {
    @Override
    protected String getApplicationMainClass() {
        return "com.mycompany.Main";
    }
}
```

When the activity is created, it instantiates the main class of the MicroEJ Application and invokes its `main()` method.

Assuming that the Wear OS application declares a watch face service in the `AndroidManifest.xml` :

- Open the corresponding watch face service Java/Kotlin file.
- Make `MicroEJWatchFaceService` the superclass of this class.
- Override the method `getApplicationMainClass()` and make it return the Full Qualified Name of the main class of the MicroEJ Application.

This is an example of a simple activity:

Kotlin

Java

```
class MyWatchFaceService : MicroEJWatchFaceService() {
    override fun getApplicationMainClass(): String {
        return "com.mycompany.Main";
    }
}
```

```
public class MyWatchFaceService extends MicroEJWatchFaceService {
    @Override
    protected String getApplicationMainClass() {
        return "com.mycompany.Main";
    }
}
```

When the watch face service is created, it instantiates the main class of the MicroEJ Application and invokes its `main()` method.

Select Sync Now from the notification bar to synchronize your project files.

Run on MicroEJ VEE and Android

The application can now be deployed to both MicroEJ VEE and Android environments.

The deployment of an application designed to use the Android Compatibility Kit has nothing specific compared to other MicroEJ or Android applications. This means that you can refer to the dedicated documentation for this matter:

- for MicroEJ VEE: refer to sections *Run On Simulator*, *Build Executable* and *Run On Device*.
- for Android: refer to the official Android documentation.
- for Wear OS: refer to the official Wear OS documentation.

8.1.4 VEE Port

This section explains how to configure a VEE Port so that it provides the capability to build a MicroEJ Application for Android.

Once it is configured, the VEE Port can thus be used to build a MicroEJ Application for Android, in addition to standard features such as building a MicroEJ Application for an Embedded Device and running it on the Simulator.

VEE Port Configuration

The configuration steps ensure that the VEE Port provides build scripts and implementations of Foundation Libraries which are specific to Android.

These files are gathered in Android Packs. Each Android Pack provides support for one or multiple Foundation Libraries. The Core Android Pack is absolutely necessary to be able to build any MicroEJ Application for Android. Additional Android Packs should be included depending on the Foundation Libraries provided by the VEE Port.

To declare an Android Pack dependency, edit the `module.ivy` file of the VEE Port and add the following line within the `<dependencies>` element:

```
<dependency org="com.microej.android.pack" name="[NAME]-android-pack" rev=
↪"[VERSION]"/>
```

MicroEJ Android Packs

MicroEJ provides Android Packs for some Foundation Libraries:

Name	Module	Implemented Libraries
Core Android Pack	com.microej.android.pack#core-android-pack	EDC, BON, SNI, Trace
UI Android Pack	com.microej.android.pack#ui-android-pack	MicroUI, Drawing
VG Android Pack	com.microej.android.pack#vg-android-pack	MicroVG

Note: Some Foundation Libraries such as FS and NET do not require an Android Pack as their APIs are already implemented by the Android SDK.

For more information on the usage and limitations of each Android Pack, refer to its README.

Custom Android Packs

A MicroEJ Application may call native methods, which require a different implementation on each execution target (embedded device, Simulator or Android). Therefore if an Application is executed on Android, the VEE Port should provide an implementation of these native methods for Android. This dedicated implementation is called an Android mock, and it is usually packaged in an Android Pack. This section explains how to develop a custom Android Pack including an Android mock.

Note: Currently, VEE Ports and their components can not be developed in *MicroEJ SDK 6*. This means that Android Packs must be developed with *SDK 5* and MMM, and cannot be developed with Android Studio and Gradle.

Setting Android SDK Environment Variable

Since the Android mock will be compiled using Android SDK, you should have Android Studio and Android SDK installed. If it is not set already on your system, you should set the ANDROID_HOME environment variable.

You can follow these steps to find the Android SDK location on your system:

- In Android Studio, select File > Settings... .
- In the settings dialog, find Android SDK and copy the path set as Android SDK Location .

On Windows, this path is typically `C:\Users\[USER]\AppData\Local\Android\Sdk` .

Make sure to restart MicroEJ SDK after setting the environment variable.

Creating the Android Pack Module

The first step is to create the `custom-android-pack` project:

- In MicroEJ SDK, select File > New > Project... .
- In the wizard dialog, select MicroEJ > Module Project and click on Next > .
- In the new module dialog, type `custom-android-pack` as Project Name and as Module , choose the Organization and Revision of your choice, select `product-java` as Skeleton and click on Finish .

By default, the library built by the module is not packaged as an Android mock. To make sure that the library is added to the list of Android mocks, edit the `module.ivy` file of the project and add the following lines within the `<ea:build>` element:

```
<ea:property name="target.main.artifact.rip.relativedir" value="android/mocks/dropins"/>
```

Compiling against Android SDK

By default, the library is compiled against the JRE. Both the Eclipse project and the MMM build must be configured to compile against Android SDK rather than the JRE.

First, the JRE must be replaced by Android SDK in the build path of the Eclipse project:

- Right-click on the project, select `Build Path` > `Configure Build Path...` .
- In the properties dialog, open the `Libraries` tab, select `JRE System Library` , click on `Remove` and click on `Add Variable...` .
- In the classpath entry dialog, click on `Configure Variables...` .
- In the variables dialog, click on `New...` .
- In the new variable dialog, type `ANDROID_HOME` as `Name` , type the Android SDK location as `Path` and click on `OK` .
- Back to the variables dialog, click on `Apply and Close` .
- Back to the classpath entry dialog, select the `ANDROID_HOME` variable and click on `Extend...` .
- In the variable extension dialog, browse the `platforms/android-[VERSION]/android.jar` file and click on `OK` .
- Back to the properties dialog, click on `Apply and Close` .

Finally, the JRE must be replaced by Android SDK in the build path of the MMM module:

- Edit the `module.ivy` file and add the following lines within the `<ea:build>` element:

```
<ea:property name="include.java.runtime" value="false"/>
<ea:property name="no.obfuscation" value="true"/>
```

- Create a file named `module.ant` at the root of the project with the following content:

```
<project name="custom-android-pack" xmlns:ea="antlib:org.apache.easyant">
        <target name="-custom-android-pack:augment-classpath" extensionOf=
↪"abstract-compile:compile-ready">
                <property environment="env"/>
                <ea:path pathid="compile.main.classpath" overwrite="prepend">
                        <fileset file="${env.ANDROID_HOME}/platforms/android-
↪[VERSION]/android.jar"/>
                </ea:path>
        </target>
</project>
```

- In this `module.ant` , replace `[VERSION]` in the `<fileset>` element by the minimum Android SDK version required by your Android mock.

Implementing the Android mock

You can add the Java source code of your Android mock into the `src/main/java` folder of the project. At runtime, the Android mock will be added to the classpath before the code of the Application and before its dependencies. This allows you to replace the implementation of any Java class in an Android mock. The recommended practice is to replace only the classes which include native methods.

Using the Android Pack in the VEE Port

To build the Android Pack, right-click on the project and select Build Module .

The Android Pack can be included in a VEE Port by declaring a dependency in the `module.ivy` of the VEE Port as explained in the first subsection:

```
<dependency org="[ORGANIZATION]" name="custom-android-pack" rev="[VERSION]"/>
```

8.2 iOS Compatibility Kit

MicroEJ provides a set of tools and libraries to run applications powered by MicroEJ on iOS. This allows for the same application to be developed, simulated, tested, and executed on MicroEJ VEE and iOS alike.

Thanks to the iOS Compatibility Kit, watch faces can be developed once and deployed on both a smartwatch (MicroEJ VEE) and its companion smartphone app (iOS), enabling consistent functionality and appearance across both devices. This provides a good user experience for the user while minimizing code duplication and maintenance for the developer.

8.2.1 Software Architecture

The iOS Compatibility Kit provides a JDK which can be used to compile and run Java code on iOS. The code of the MicroEJ Application and of the libraries it depends on is executed on a Java VM started by the iOS app.

The JDK runtime includes JavaFX to be able to display Applications which use *MicroUI* or *MicroVG*.

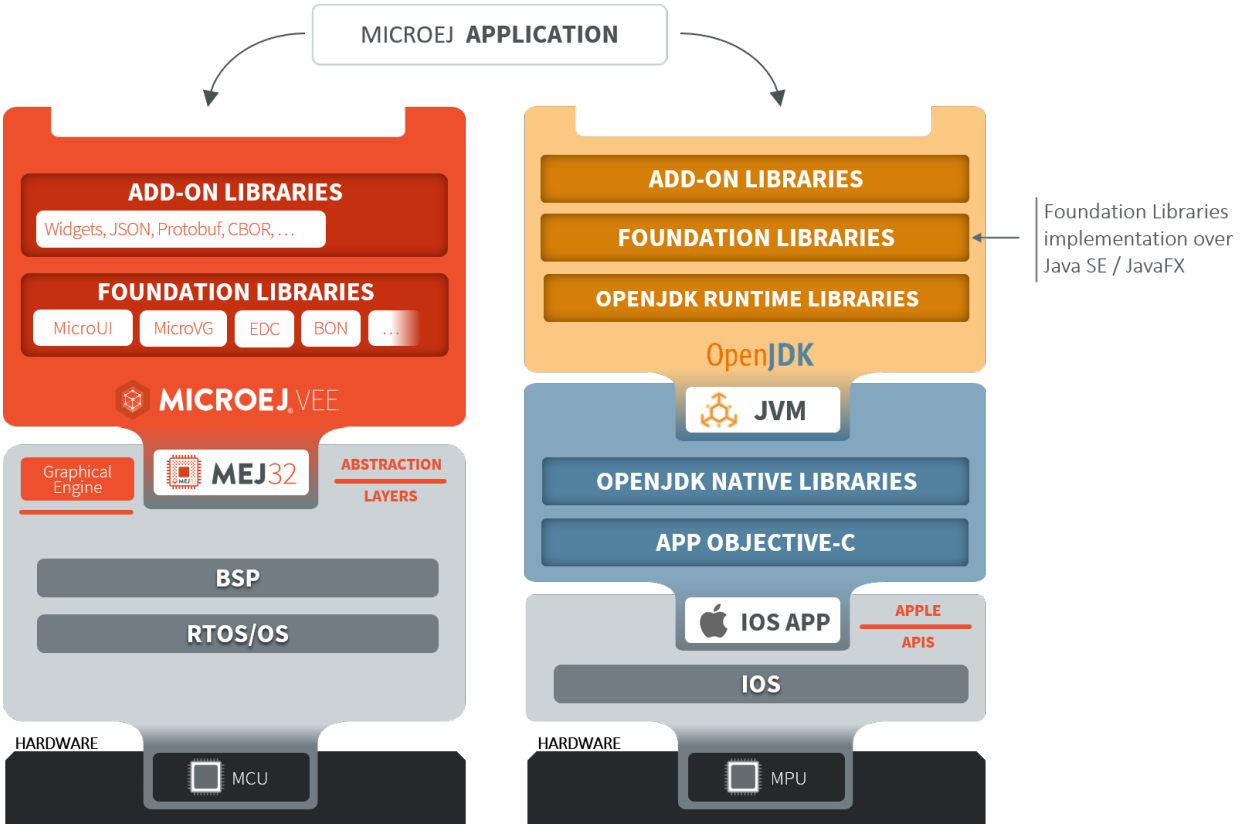


Fig. 3: Software Architecture

8.2.2 Workflow

The iOS app can be developed and built on Xcode as a regular iOS application. To run a MicroEJ Application, the Xcode project is configured to embed:

- the runtime libraries of the JDK (JARs and native libraries),
- the implementation of the MicroEJ Foundation Libraries compiled against Java SE and JavaFX APIs,
- the code and resources of the MicroEJ Application and of the Add-On Libraries that it depends on.

8.2.3 Evaluation

The iOS Compatibility Kit is available on demand. You can contact *MicroEJ Support* to evaluate this solution.

8.3 Offloading

Many high-end smartwatches rely on a microprocessor running Android. The power consumption of these devices is fairly high and show an average battery life of one or two days. Integrating an extra low-power microcontroller into the watch's hardware enables the delegation of specific tasks from the main microprocessor, resulting in an increased battery life. Keeping a powerful microprocessor on the hardware ensures the ability to display high-performance animations and access the Android ecosystem.

8.3.1 Solution

VEE Wear offers a comprehensive solution for software development on this dual architecture:

- MicroEJ VEE enables the execution of applications written in high-level code through virtualization on the microcontroller.
- the *Android Compatibility Kit* allows the execution of the same application on the microprocessor without the need for re-implementation.
- the Offloading Framework provides the ability to switch the application context between the two processors depending on their capabilities and on the application flow.

One Code, Two Targets

MicroEJ Application development shares the same programming language as Android Application development. Thanks to this similarity, any MicroEJ Application code is compatible with the Android runtime environment. The *Android Compatibility Kit* provides the tools and libraries to execute a MicroEJ Application on Android.

Using this solution, the application code can be programmed once and executed both on the low-power microcontroller and the high-power microprocessor.

Offloading Framework

The offloading strategy is a set of rules defined by the design of the watch.

Here is a non-exhaustive list of common offloading rules:

- switching to the low-power microcontroller after a few seconds of user inactivity (often called ambient mode)
- switching to the high-power microprocessor when starting a high-performance animation
- switching to the high-power microprocessor when navigating to a menu which is only available on Android

Switching from a processor to the other may require to synchronize the state of the software and to provide the necessary data for the other processor to take over. The time to wake-up the processor and to synchronize the data has to be taken into account when designing the software architecture.

Once these rules have been decided, the Offloading Framework APIs can be used to wake up the other processor, to synchronize data, to be notified when the other processor is ready, to hand over the control of the display, to put the processor to sleep, etc.

8.3.2 Evaluation

The Android Compatibility Kit can be evaluated by following *its documentation*.

An demonstration with a sample offloading framework is available on demand. You can contact *MicroEJ Support* to test this demonstration.

CHAPTER

NINE

TUTORIALS

9.1 Understand How to Build a Firmware and its Dependencies

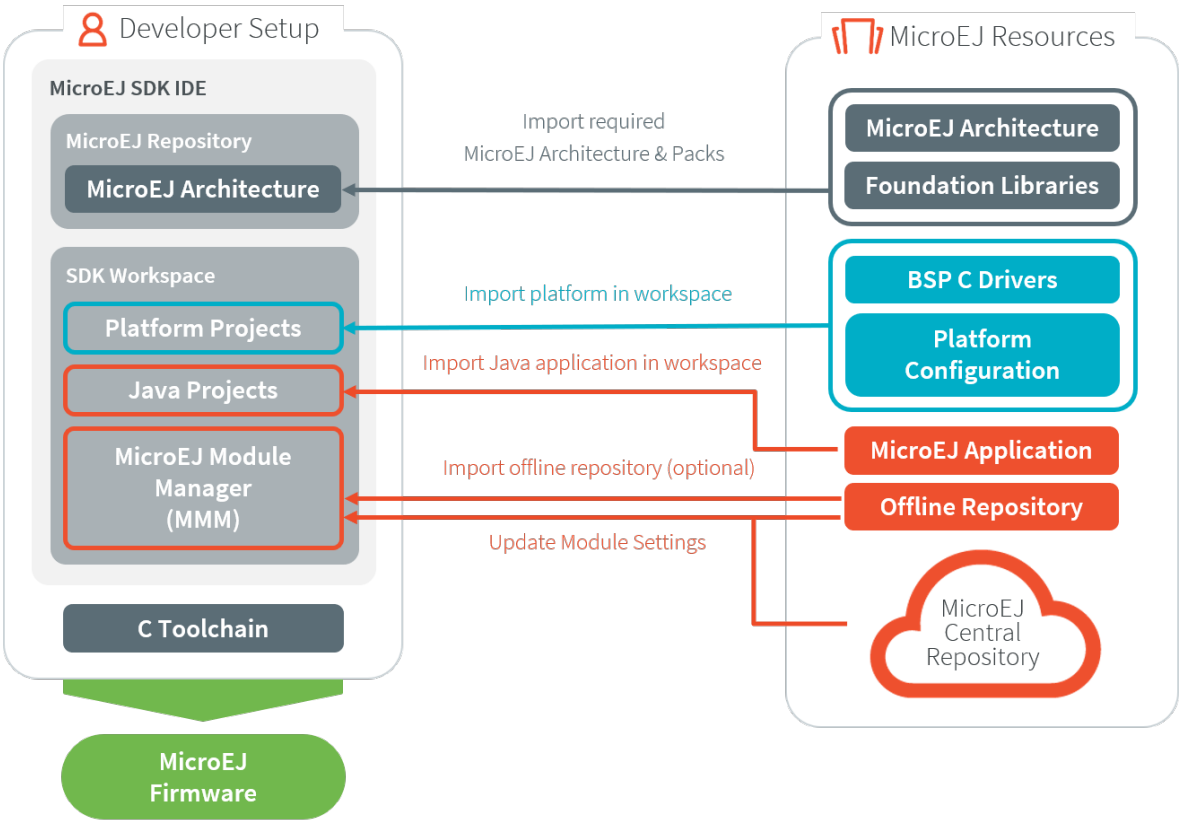
A Firmware is built from several input resources and tools. Each component has dependencies and requirements that must be carefully respected in order to build a Firmware.

This document describes the components, their dependencies and the process involved in the build of a Firmware.

Good knowledge of the *MicroEJ Glossary* is required.

9.1.1 The Components

As depicted in the following image, several resources and tools are used to build a Firmware.



Architecture

A *MicroEJ Architecture* contains the runtime port to a target instruction set (ISA), a C compiler (CC) and Foundation Libraries.

Architectures are distributed in two versions:

- *Evaluation Architectures*: license with runtime limitations (explained in the *Application Developer Guide*).
- *Production Architectures*: license suitable for production.

A selection of supported embedded Architectures can be found here: https://developer.microej.com/mej32-embedded-runtime-architectures/

The Architecture is either provided from:

- *MicroEJ Central Repository*, for Evaluation Architectures only.
- *MicroEJ Support team* or your MicroEJ sales representative, for Production Architectures only.

Note: Ask MicroEJ sales or support team if the requested architecture is not listed as available.

Platform Sources

A *Platform* includes development tools and a runtime environment:

- the Architecture and *MicroEJ Packs*,
- the Abstraction Layers implementations,
- the Simulator and its associated Mocks,
- a C Board Support Package (BSP) with C drivers and an optional RTOS.

The Platform sources contains the following projects:

- `<platform>-configuration` : The Platform Configuration project.
- `<platform>-bsp` : The C code for the board-specific files (drivers).
- `<platform>-fp` : Front Panel mockup for the simulator.

See *Platform Import* to learn how to import an existing Platform, and *Platform Creation* to learn how to create a Platform.

Depending on the project's requirements, the Platform can be connected in various ways to the BSP; see *BSP Connection* for more information on how to do it.

Application

An Application is a Java project that can be configured (in the Run configurations … properties):

1. to either run on:
   - the 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 Platform. It generates a `microejapp.o` (native object code) linked with the `<platform>-bsp` project.

To import an existing Application as a zipped project in the SDK:

- Go to File > Import... > General > Existing Projects into Workspace > Select archive file > Browse... .
- Select the zip file of the project.
- And select Finish import.

See *Create a MicroEJ Standalone Application* for more information on how to create, configure, and develop a Standalone Application.

C Toolchain (GCC, KEIL, IAR, ...)

Used to compile and link the following files into the final executable (binary, hex, elf, ... that will be programmed on the hardware):

- the `microejapp.o` (application),
- the `microejruntime.lib` or `microejruntime.a` (Platform runtime),
- the BSP C files (C application files and Board Support Package).

Module Repository

A Module Repository provides the modules required to build Platforms and 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

- An Architecture targets a specific instruction set (ISA) and a specific C compiler (CC).
  - The C toolchain used for the Architecture must be the same as the one used to compile and link the BSP project.
- A Platform consists of the aggregation of both an Architecture and a BSP with a C toolchain.
  - Changing either the Architecture or the C toolchain results in a change of the Platform.
- An Application is independent of the Architecture.
  - It can run on any Platform as long the Platform provides the required APIs.
  - To run an Application on a new device, create a new Platform for this device with the exact same features. The Application will not require any change.

9.1.2 How to Build

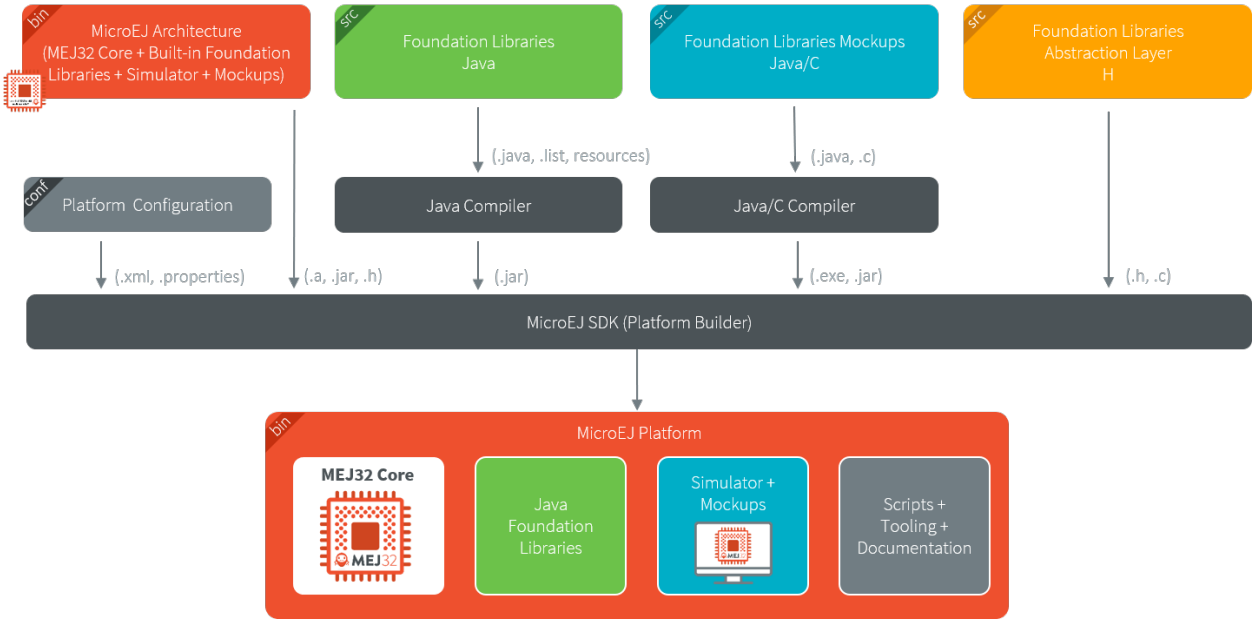
The process of building a Firmware is two-fold:

1. Build a Platform,
2. Compile/link the application and BSP using the C toolchain.

Note: The Application will also run on the Simulator using the mocks provided by the Platform.

Build a Platform

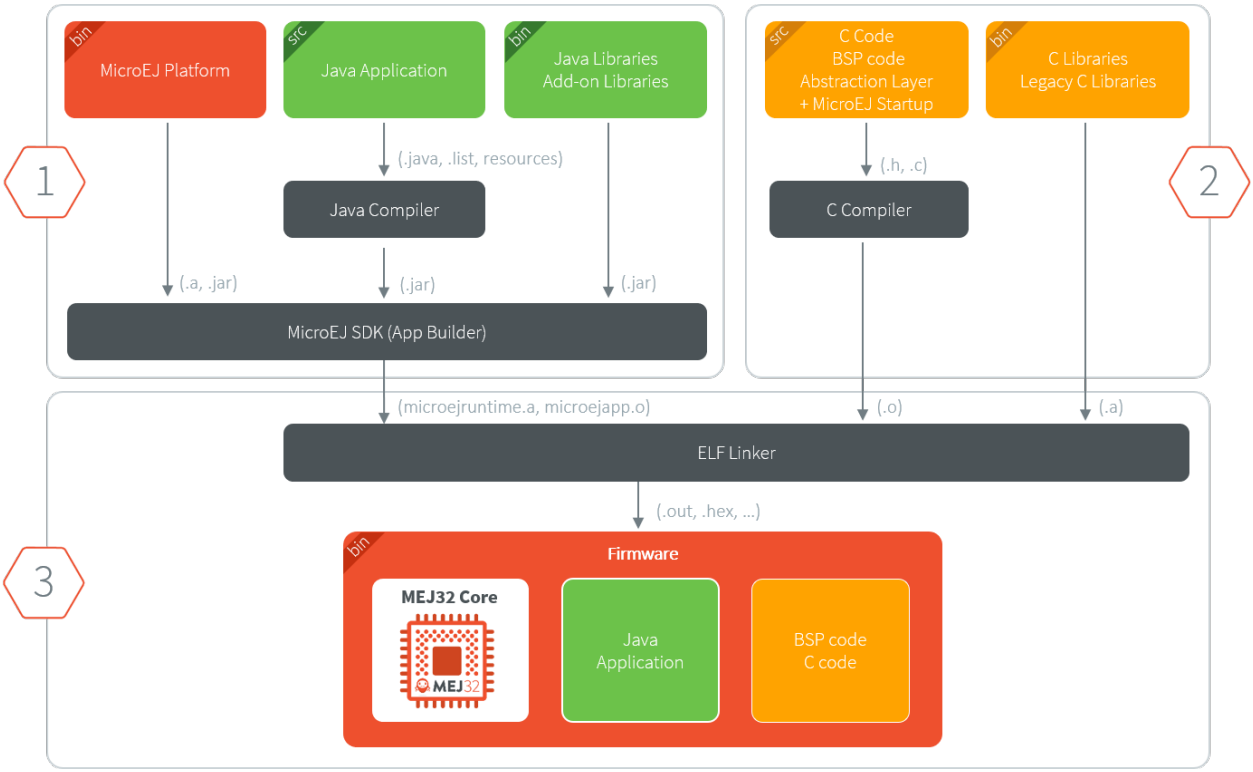
The next schema presents the components and process to build a Platform.



Build a Firmware

The next schema presents the build flow of a Mono-Sandbox Firmware (previously known as a MicroEJ Single-app Firmware). The steps are:

1. Build the Application using the SDK (generates a `microejapp.o` file).
2. Compile the BSP C sources using the C toolchain (generates `.o` files).
3. Link the BSP files (`.o`), the Application (`microejapp.o`) and the Platform runtime library (`microejruntime.a`) using the C toolchain to produce the final executable (ELF or binary, for example `application.out`).



See *BSP Connection* for more information on how to connect a Platform to a BSP.

Dependencies Between Processes

- Rebuild the Platform:
  - When the Architecture (`.xpf`) changes.
  - When a *Pack* provided by MicroEJ (`.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 Platform is not required:
  - When the implementation (`.c`) of a Foundation Library changes.
  - When the BSP (`.c`) changes.
  - When the Application changes.
- Rebuild the Application:
  - When its code changes.
  - When the Platform changes.
- Rebuild the BSP:
  - When its code changes.
  - When the Platform changes.
- Rebuild the Firmware:

- When the Application (`microejapp.o`) changes.
- When the BSP (`*.o`) changes.
- When the Platform (`microejruntime.a`) changes.

9.2 Create a MicroEJ Platform for a Custom Device

9.2.1 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/modules/, and MicroEJ Platform demo projects for various evaluation boards at https://repository.microej.com/index.php?resource=JPF.

We recommend reading the *MICROEJ VEE* section to get an overview of MicroEJ Firmware build flow.

The following document assumes the reader is familiar with the *VEE Porting 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.

9.2.2 A MicroEJ Platform Project is already available for the same MCU/RTOS/C Compiler

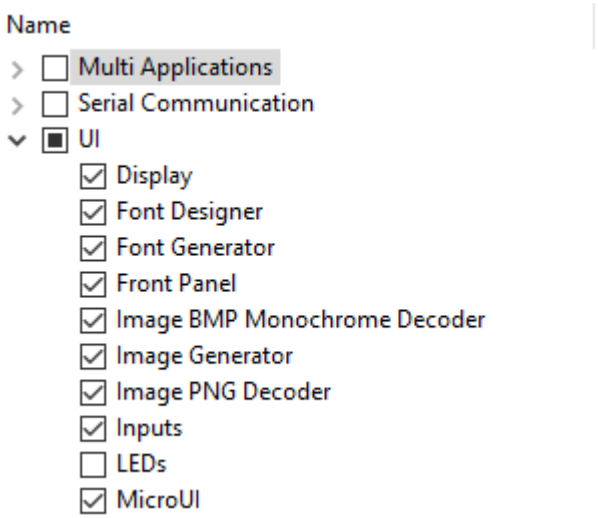
This is the fastest way: the MicroEJ Platform is usually provided for a silicon vendor evaluation board. Import this platform in MicroEJ SDK.

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 the 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 *VEE Porting Guide*.

BSP

Required actions:

- modify the BSP C project to match the device specification
  - edit the scatter file/link options
  - edit the compilation options
- create/review/change the platform Low Level C drivers. They must match the device components and the MCU IO pin assignment

---

Note: A number of `LL*.h` files are referenced from the project. Implement the function prototypes declared there so that the JVM can delegate the relevant operations to the provided BSP C functions.

Simulator

In the SDK

- modify the existing Simulator Front Panel `xxx-fp` project

9.2.3 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 the 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 the SDK: create the MicroEJ Platform and refer to the selected MicroEJ Platform as a model for implementation. (refer to *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.

9.2.4 Platform Validation

Use the Platform Qualification Tools to qualify the MicroEJ Platform built.

9.2.5 Further Assistance Needed

Please note that porting MicroEJ to a new device is also something that is part of our engineering services. Consider contacting our sales team to request a quote.

9.3 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)).

9.3.1 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*.

9.3.2 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.

9.3.3 Prerequisites

- MicroEJ SDK version 5.3.0 or higher (distribution 20.10). Can be downloaded from https://repository.microej.com/packages/SDK (tested on MicroEJ SDK distribution 20.10)
- Windows 10 or higher with Windows Subsystem for Linux (WSL). See the installation guide.
- A Linux distribution installed on WSL (Tested on Ubuntu 19.10 eoan and Ubuntu 20.04 focal).

> **Note:** In WSL, use the command `lsb_release -a` to print the current Ubuntu version.

A code editor such as Visual Studio Code is also recommended to edit BSP files.

9.3.4 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.

9.3.5 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.

9.3.6 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.

Listing 1: https://github.com/MicroEJ/FreeRTOS/commit/48248eae13baebf7df9638cd8da6fbfe1a735a9c

```
diff --git a/FreeRTOS/Demo/CORTEX_LM3S811_GCC/standalone.ld b/FreeRTOS/Demo/CORTEX_
↪LM3S811_GCC/standalone.ld
--- a/FreeRTOS/Demo/CORTEX_LM3S811_GCC/standalone.ld
+++ b/FreeRTOS/Demo/CORTEX_LM3S811_GCC/standalone.ld
@@ -42,7 +42,15 @@ SECTIONS
         _etext = .;
```

(continues on next page)

(continued from previous page)

```
     } > FLASH

-    .data : AT (ADDR(.text) + SIZEOF(.text))
+    .ARM.exidx :
+    {
+        *(.ARM.exidx*)
+        *(.gnu.linkonce.armexidx.*)
+    } > FLASH
+
+    _begin_data = .;
+
+    .data : AT ( _begin_data )
     {
         _data = .;
         *(vtable)
```

This is the output of the `git diff` command. Lines starting with a `-` should be removed. Lines starting with a `+` should be added.

Note: The `patch(1)` can be used to apply the patch. Assuming WSL shell is in `FreeRTOS/Demo/CORTEX_LM3S811_GCC` directory:

1. Install dos2unix utility: `sudo apt install dos2unix`
2. Convert all files to unix line-ending: `find -type f -exec dos2unix {} \;`
3. Copy the content of the code block in a file named `linker.patch` (every lines of the code block must be copied in the file).
4. Apply the patch: `patch -l -p4 < linker.patch`.

It is also possible to paste the diff directly into the console:

1. In WSL, invoke `patch -l -p4`. The command starts, waiting for input on stdin (the standard input).
2. Copy the diff and paste it in WSL
3. Press enter
4. Press `Ctrl-d Ctrl-d` (press the `Control` key + the letter `d` twice).

---

3. Run the build again: `make`
4. Run the emulator with the generated kernel: `qemu-system-arm -M lm3s811evb -nographic -kernel gcc/RTOSDemo.bin`

   The following error appears and then nothing:

```
ssd0303: error: Unknown command: 0x80
ssd0303: error: Unexpected byte 0xe3
ssd0303: error: Unknown command: 0x80
ssd0303: error: Unexpected byte 0xe3
ssd0303: error: Unknown command: 0x80
ssd0303: error: Unexpected byte 0xe3
ssd0303: error: Unknown command: 0x80
ssd0303: error: Unexpected byte 0xe3
```

(continues on next page)

(continued from previous page)

```
ssd0303: error: Unknown command: 0x80
ssd0303: error: Unexpected byte 0xe3
ssd0303: error: Unknown command: 0x80
ssd0303: error: Unexpected byte 0xe3
ssd0303: error: Unknown command: 0x80
ssd0303: error: Unexpected byte 0xe3
ssd0303: error: Unknown command: 0x80
ssd0303: error: Unexpected byte 0xe3
ssd0303: error: Unknown command: 0x80
ssd0303: error: Unexpected byte 0xe3
```

5. Press `Ctrl-a x` (press `Control` + the letter `a`, release, press `x`) to the end the QEMU session. The session ends with `QEMU: Terminated`.

Note: The errors can be safely ignored. They occur because the OLED controller emulated receive incorrect commands.

At this point, the target device is successfully booted with the FreeRTOS kernel.

9.3.7 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).

Here is the patch that implements `putchar(3)` and `puts(3)` and prints `Hello World`.

Listing 2: https://github.com/MicroEJ/FreeRTOS/commit/d09ec0f5cbdf69ca97a5ac15f8b905538aa4c61e

```
diff --git a/FreeRTOS/Demo/CORTEX_LM3S811_GCC/main.c b/FreeRTOS/Demo/CORTEX_LM3S811_GCC/main.
↪c
--- a/FreeRTOS/Demo/CORTEX_LM3S811_GCC/main.c
+++ b/FreeRTOS/Demo/CORTEX_LM3S811_GCC/main.c
@@ -134,9 +134,25 @@ SemaphoreHandle_t xButtonSemaphore;
 QueueHandle_t xPrintQueue;

 /*-----------------------------------------------------------*/
+#define UART0BASE ((volatile int*) 0x4000C000)
+
+int putchar (int c){
+    (*UART0BASE) = c;
+    return c;
+}
+
+int puts(const char *s) {
+    while (*s) {
+        putchar(*s);
+        s++;
+    }
```

(continues on next page)

(continued from previous page)

```
+    return putchar('\n');
+}

 int main( void )
 {
+    puts("Hello, World! puts function is working.");
+
    /* Configure the clocks, UART and GPIO. */
    prvSetupHardware();
```

Rebuild and run the newly generated kernel: `make && qemu-system-arm -M lm3s811evb -nographic -kernel gcc/RTOSDemo.bin` (press `Ctrl-a x` to interrupt the emulator).

```
make: Nothing to be done for 'all'.
Hello, World! puts function is working.
ssd0303: error: Unknown command: 0x80
ssd0303: error: Unexpected byte 0xe3
ssd0303: error: Unknown command: 0x80
ssd0303: error: Unexpected byte 0xe3
ssd0303: error: Unknown command: 0x80
ssd0303: error: Unexpected byte 0xe3
ssd0303: error: Unknown command: 0x80
ssd0303: error: Unexpected byte 0xe3
ssd0303: error: Unknown command: 0x80
ssd0303: error: Unexpected byte 0xe3
ssd0303: error: Unknown command: 0x80
ssd0303: error: Unexpected byte 0xe3
ssd0303: error: Unknown command: 0x80
ssd0303: error: Unexpected byte 0xe3
ssd0303: error: Unknown command: 0x80
ssd0303: error: Unexpected byte 0xe3
ssd0303: error: Unknown command: 0x80
ssd0303: error: Unexpected byte 0xe3
QEMU: Terminated
```

With this two functions implemented, `printf(3)` is also available.

Listing 3: https://github.com/MicroEJ/FreeRTOS/commit/1f7e7ee014754a4dcb4f6c5a470205e02f6ac3c8

```
diff --git a/FreeRTOS/Demo/CORTEX_LM3S811_GCC/main.c b/FreeRTOS/Demo/CORTEX_LM3S811_GCC/main.
↪c
--- a/FreeRTOS/Demo/CORTEX_LM3S811_GCC/main.c
+++ b/FreeRTOS/Demo/CORTEX_LM3S811_GCC/main.c
@@ -149,9 +149,11 @@ int puts(const char *s) {
        return putchar('\n');
 }

+#include <stdio.h>
+
 int main( void )
 {
```

(continues on next page)

(continued from previous page)

```
-       puts("Hello, World! puts function is working.");
+       printf("Hello, World! printf function is working.\n");

        /* Configure the clocks, UART and GPIO. */
        prvSetupHardware();
```

At this point, the character output on the UART is implemented in the FreeRTOS BSP. The next step is to create the MicroEJ Platform and MicroEJ Application.

9.3.8 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.

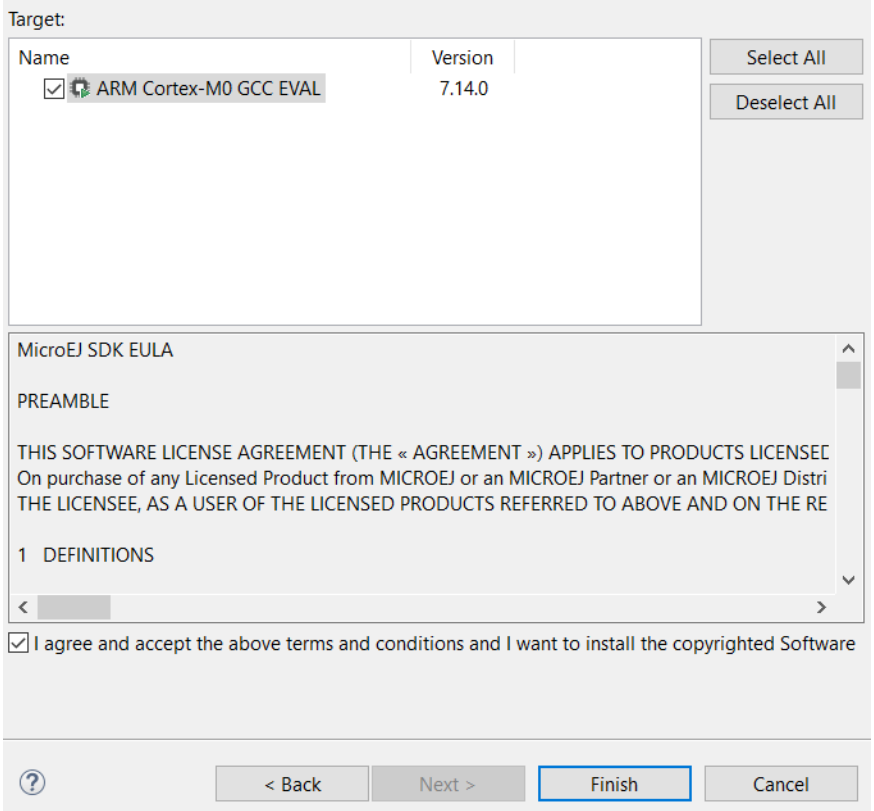
The list of MicroEJ Architectures supported is listed *here*. MicroEJ Evaluation Architectures provided by MicroEJ Corp. can be downloaded from MicroEJ Architectures Repository.

There is no `CM3` in MicroEJ Architectures Repository and the Arm® Cortex®-M3 MCU is not mentioned in the *capabilities matrix*. This means that the MicroEJ Architectures for Arm® Cortex®-M3 MCUs are no longer distributed for evaluation. Download the latest MicroEJ Architecture for Arm® Cortex®-M0 instead (the Arm® architectures are binary upward compatible from Arm®v6-M (Cortex®-M0) to Arm®v7-M (Cortex®-M3)).

Import the MicroEJ Architecture

This step describes how to import a *MicroEJ Architecture*.

1. Start MicroEJ SDK on an empty workspace. For example, create an empty folder `workspace` next to the `FreeRTOS` git folder and select it.
2. Keep the default MicroEJ Repository
3. Download the latest MicroEJ Architecture for Arm® Cortex®-M0 instead: https://repository.microej.com/modules/com/microej/architecture/CM0/CM0_GCC48/flopi0G22/7.14.0/flopi0G22-7.14.0-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

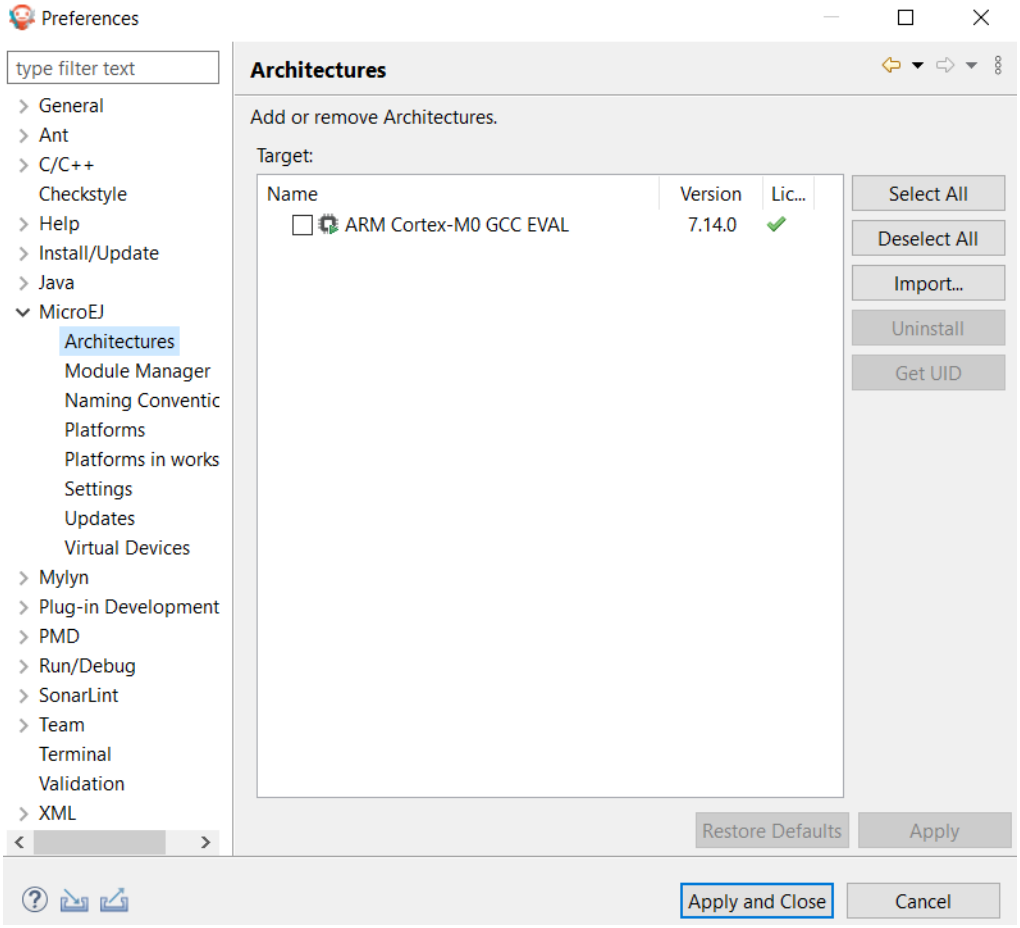


Install an Evaluation License

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 contact *our support team*.
    - Once received by email, save the attached zip file that contains the activation key.
11. Go back to Microej SDK.

12. Select the `Window` > `Preferences` > `MicroEJ` menu.
13. Press `Add...` .
14. Browse the previously downloaded activation key archive file.
15. Press `OK` . A new license is successfully installed.
16. Go to `Architectures` sub-menu and check that all architectures are now activated (green check).
17. Microej SDK is successfully activated.

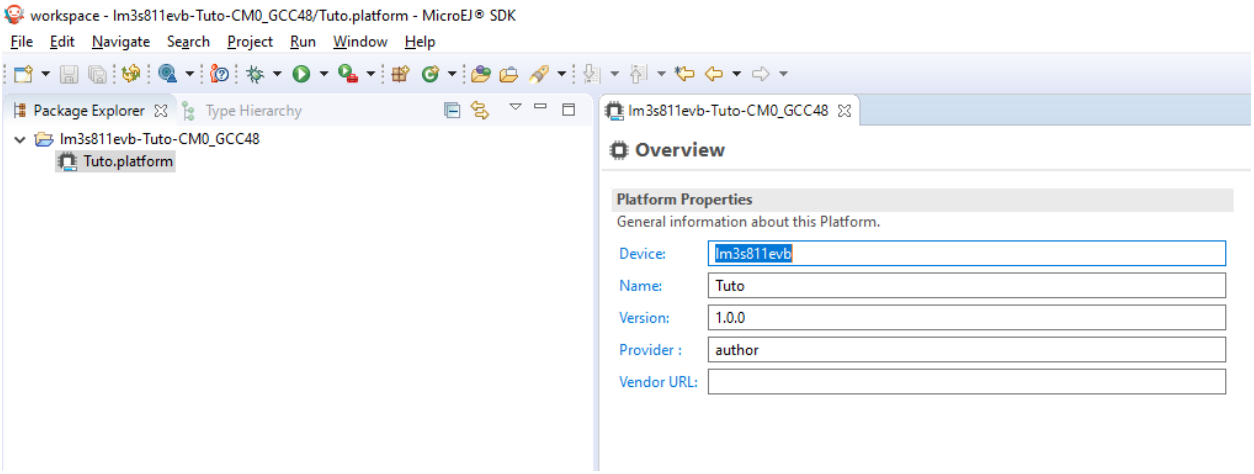


Create the MicroEJ Platform

This step describes how to create a new *MicroEJ Platform* using the MicroEJ Architecture previously imported.

1. Select `File` > `New` > `Platform Project` .
2. Ensure the `Architecture` selected is the MicroEJ Architecture previously imported.
3. Ensure the `Create from a platform reference implementation` box is unchecked.
4. Click on `Next` button.
5. Fill the fields:
   - Set `Device:` to `lm3s811evb`

- Set `Name:` to `Tuto`



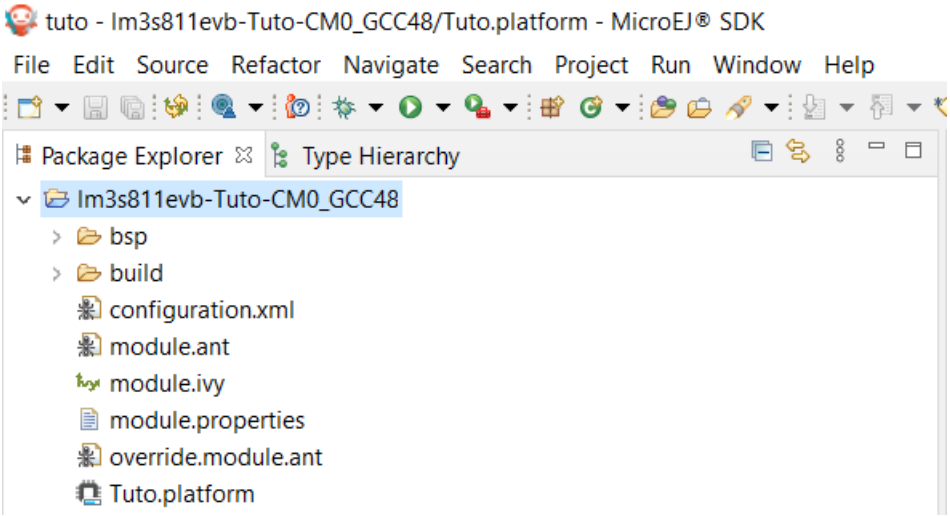
Setup the MicroEJ Platform

This step describes how to configure the MicroEJ Platform previously created. For more information on this topic, please refer to *Platform Configuration*.

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 direc-

tory (the directory that contains the `.platform` file). It can be automatically downloaded using the following command line:

```
svn export --force https://github.com/MicroEJ/VEEPortQualificationTools/tags/2.6.0/
↪framework/platform/content [path_to_platform_configuration_directory]
```

---

2. Edit the file `bsp/bsp.properties` as follow:

```
# Specify the MicroEJ Application file ('microejapp.o') parent directory.
# This is a '/' separated directory relative to 'bsp.root.dir'.
microejapp.relative.dir=microej/lib

# Specify the MicroEJ Platform runtime file ('microejruntime.a') parent directory.
# This is a '/' separated directory relative to 'bsp.root.dir'.
microejlib.relative.dir=microej/lib

# Specify MicroEJ Platform header files ('*.h') parent directory.
# This is a '/' separated directory relative to 'bsp.root.dir'.
microejinc.relative.dir=microej/inc
```

3. Edit the file `modules.ivy` and add the MicroEJ Architecture as a dependency:

```
<dependencies>
      <dependency org="com.microej.architecture.CM0.CM0_GCC48" name="flopi0G22" rev="7.
↪14.0">
            <artifact name="flopi0G22" m:classifier="${com.microej.platformbuilder.
↪architecture.usage}" ext="xpf"/>
      </dependency>
</dependencies>
```

4. Edit the file `modules.properties` and set the MicroEJ platform filename:

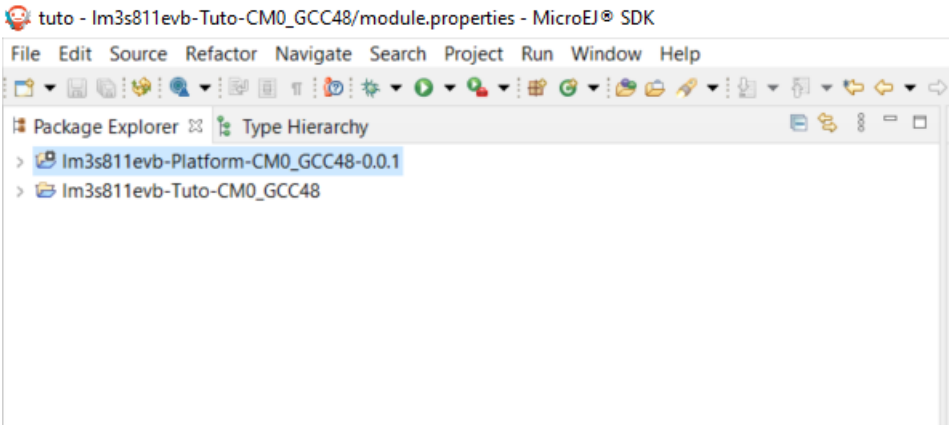
```
# Platform configuration file (relative to this project).
com.microej.platformbuilder.platform.filename=Tuto.platform
```

5. Right-click on the platform project and click on `Build Module`.
6. The following message appears in the console:

```
module-platform:report:
      [echo]    ␣
↪=============================================================================================================
      [echo]     Platform has been built in this directory 'C:\Users\user\src\tuto-
↪from-scratch\workspace/lm3s811evb-Platform-CM0_GCC48-0.0.1'.
      [echo]     To import this project in your MicroEJ SDK workspace (if not already␣
↪available):
      [echo]      - Select 'File' > 'Import...' > 'General' > 'Existing Projects into␣
↪Workspace' > 'Next'
      [echo]      - Check 'Select root directory' and browse 'C:\Users\user\src\tuto-
↪from-scratch\workspace/lm3s811evb-Platform-CM0_GCC48-0.0.1' > 'Finish'
      [echo]    ␣
↪=============================================================================================================

BUILD SUCCESSFUL
```

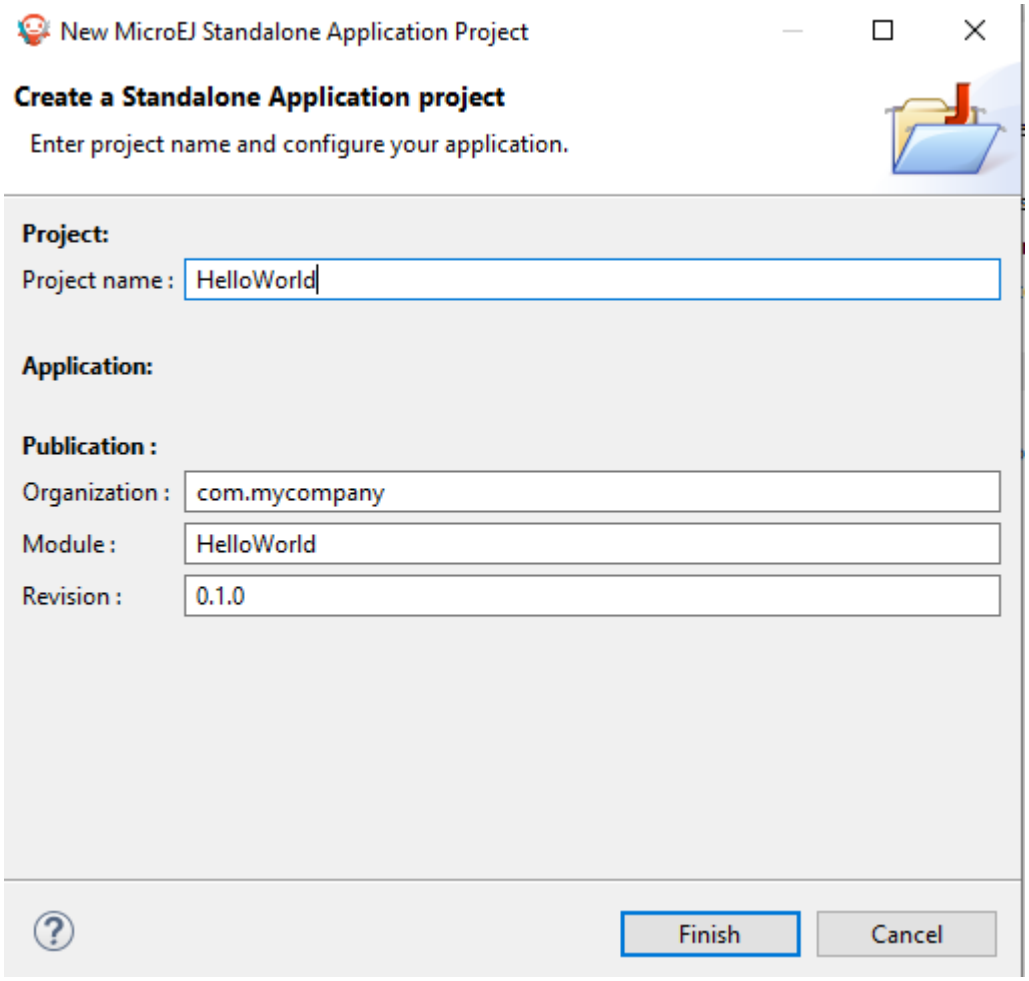
7. Follow the instructions to import the generated platform in the workspace:



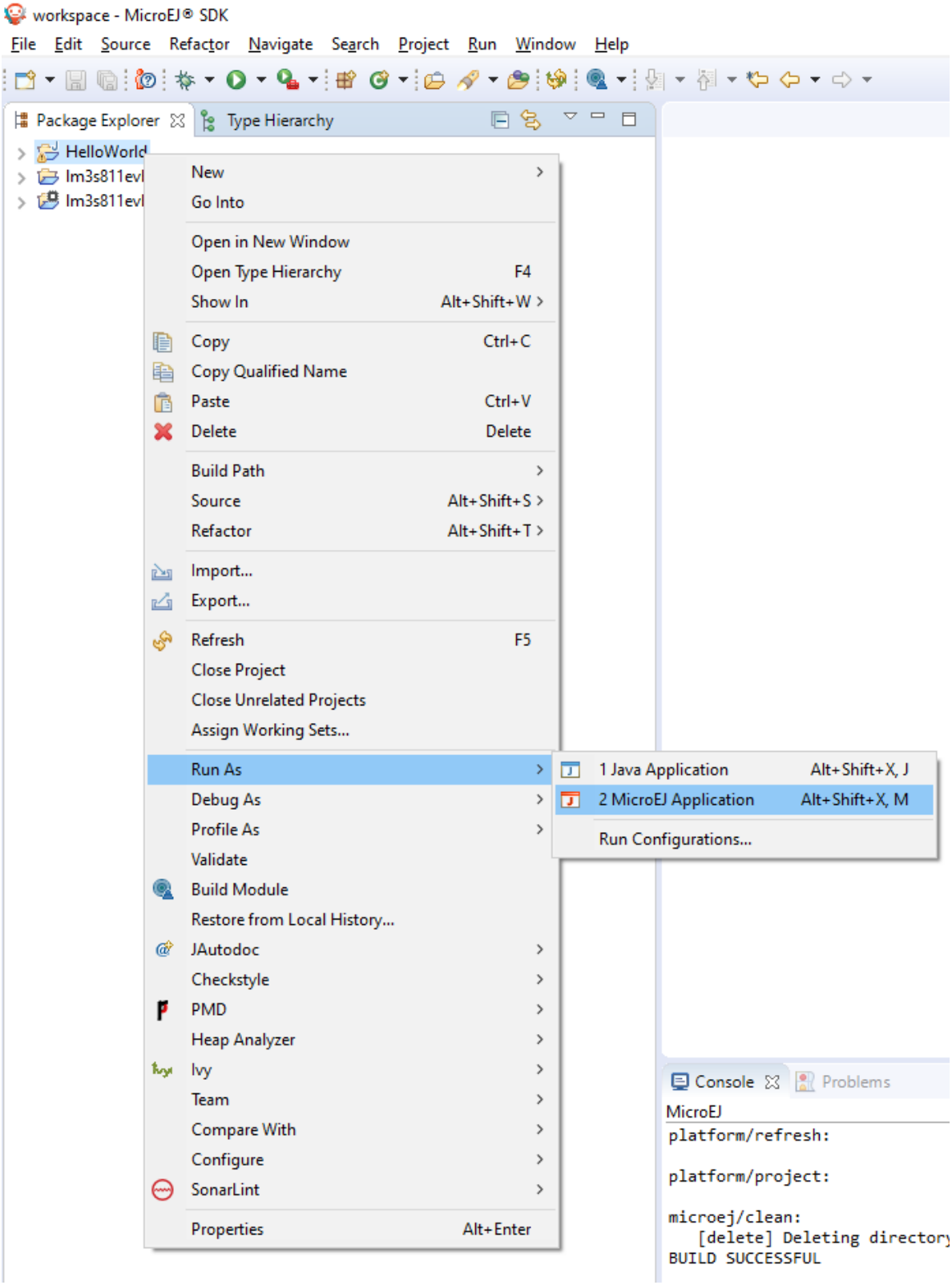
At this point, the MicroEJ Platform is ready to be used to build MicroEJ Applications.

9.3.9 Create MicroEJ Application HelloWorld

1. Select File > New > Standalone Application Project .
2. Set the name to `HelloWorld` and click on Finish



3. Run the application in Simulator to ensure it is working properly. Right-click on HelloWorld project > Run As > MicroEJ Application



The following message appears in the console:

```
============== [ Initialization Stage ] ==============
============== [ Launching on Simulator ] ===============
Hello World!
============== [ Completed Successfully ] ===============

SUCCESS
```

9.3.10 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/emb.properties` file.

1. Create a file `HelloWorld/build/emb.properties` with the following content:

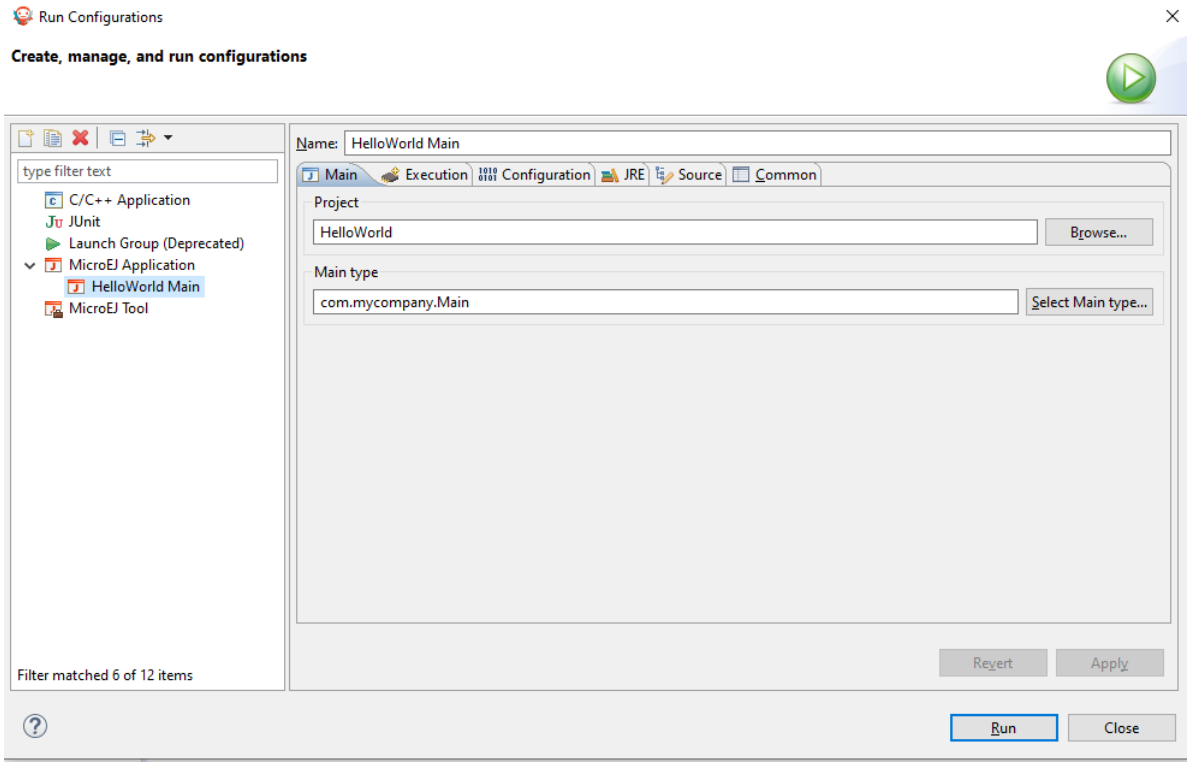
```
core.memory.immortal.size=0
core.memory.javaheap.size=1024
core.memory.threads.pool.size=4
core.memory.threads.size=1
core.memory.thread.max.size=4
deploy.bsp.microejapp=true
deploy.bsp.microejlib=true
deploy.bsp.microejinc=true
deploy.bsp.root.dir=[absolute_path] to FreeRTOS\\FreeRTOS\\Demo\\CORTEX_LM3S811_GCC
```

Note: Assuming the WSL current directory is `FreeRTOS/FreeRTOS/Demo/CORTEX_LM3S811_GCC`, use the following command to find the `deploy.bsp.root.dir` path with proper escaping:

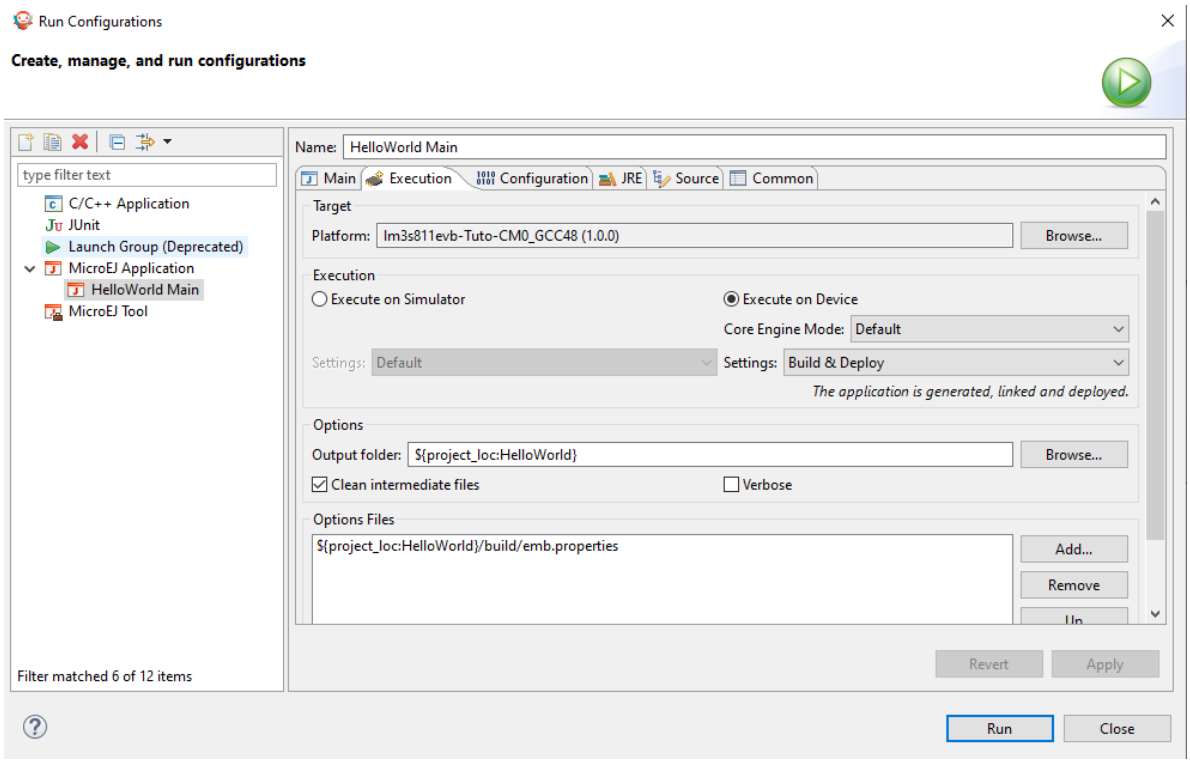
```
pwd | sed -e 's|/mnt/c/|C:\\\\|' -e 's|/|\\\\|g'
```

---

2. Open Run > Run configurations...
3. Select the HelloWorld launcher configuration



4. Select `Execution` tab.
5. Change the execution mode from `Execute on Simulator` to `Execute on Device` .
6. Add the file `build/emb.properties` to the options files



7. Click on Run

```
=============== [ Initialization Stage ] ===============
Platform connected to BSP location 'C:\Users\user\src\tuto-from-scratch\FreeRTOS\FreeRTOS\
↪Demo\CORTEX_LM3S811_GCC' using application option 'deploy.bsp.root.dir'.
=============== [ Launching SOAR ] ===============
=============== [ Launching Link ] ===============
=============== [ Deployment ] ===============
MicroEJ files for the 3rd-party BSP project are generated to 'C:\Users\user\src\tuto-from-
↪scratch\workspace\HelloWorld\com.mycompany.Main\platform'.
The MicroEJ application (microejapp.o) has been deployed to: 'C:\Users\user\src\tuto-from-
↪scratch\FreeRTOS\FreeRTOS\Demo\CORTEX_LM3S811_GCC\microej\lib'.
The MicroEJ platform library (microejruntime.a) has been deployed to: 'C:\Users\user\src\
↪tuto-from-scratch\FreeRTOS\FreeRTOS\Demo\CORTEX_LM3S811_GCC\microej\lib'.
The MicroEJ platform header files (*.h) have been deployed to: 'C:\Users\user\src\tuto-from-
↪scratch\FreeRTOS\FreeRTOS\Demo\CORTEX_LM3S811_GCC\microej\inc'.
=============== [ Completed Successfully ] ===============

SUCCESS
```

At this point, the HelloWorld MicroEJ Application is built and deployed in the FreeRTOS BSP.

9.3.11 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
```

(continues on next page)

(continued from previous page)

```
└── lib
    ├── microejapp.o
    └── microejruntime.a

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 4: 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 5: 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()
```

(continues on next page)

(continued from previous page)

```
{
        return 0;
}

int32_t LLMJVM_IMPL_shutdown(void)
{
        return LLMJVM_OK;
}
```

The `microej` folder in the BSP has the following structure:

```
$ pwd
/mnt/c/Users/user/src/tuto-from-scratch/FreeRTOS/FreeRTOS/Demo/CORTEX_LM3S811_GCC
$ tree microej/
microej/
├── inc
│   ├── BESTFIT_ALLOCATOR.h
│   ├── BESTFIT_ALLOCATOR_impl.h
│   ├── LLBSP_impl.h
│   ├── LLMJVM.h
│   ├── LLMJVM_MONITOR_impl.h
│   ├── LLMJVM_impl.h
│   ├── LLTRACE_impl.h
│   ├── MJVM_MONITOR.h
│   ├── MJVM_MONITOR_types.h
│   ├── intern
│   │   ├── BESTFIT_ALLOCATOR.h
│   │   ├── BESTFIT_ALLOCATOR_impl.h
│   │   ├── LLBSP_impl.h
│   │   ├── LLMJVM.h
│   │   ├── LLMJVM_impl.h
│   │   └── trace_intern.h
│   ├── sni.h
│   └── trace.h
├── lib
│   ├── microejapp.o
│   └── microejruntime.a
└── src
    ├── LLBSP.c
    └── LLMJVM_stub.c

4 directories, 21 files
```

Invoke 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`.

Listing 6: https://github.com/MicroEJ/FreeRTOS/commit/7ae8e79f9c811621569ccb90c46b1dcda91da35d

```
diff --git a/FreeRTOS/Demo/CORTEX_LM3S811_GCC/main.c b/FreeRTOS/Demo/CORTEX_LM3S811_GCC/main.
↪c
--- a/FreeRTOS/Demo/CORTEX_LM3S811_GCC/main.c
+++ b/FreeRTOS/Demo/CORTEX_LM3S811_GCC/main.c
@@ -150,11 +150,14 @@ int puts(const char *s) {
 }

 #include <stdio.h>
+#include "sni.h"

 int main( void )
 {
        printf("Hello, World! printf function is working.\n");

+       SNI_startVM(SNI_createVM(), 0, NULL);
+
    /* Configure the clocks, UART and GPIO. */
    prvSetupHardware();
```

Build and Link the Firmware with the MicroEJ Runtime and MicroEJ Application

To build and link the firmware with the MicroEJ Runtime and MicroEJ Application, the BSP port must be modified to:

1. Use the MicroEJ header files in folder `microej/inc`
2. Use the source files folder `microej/src` that contains the Low Level API implementation `LLBSP.c` and `LLMJVM_stub.c`
3. Compile and link `LLBSP.o` and `LLMJVM_stub.o`
4. Link with MicroEJ Application (`microej/lib/microejapp.o`) and MicroEJ Runtime (`microej/lib/microejruntime.a`)

The following patch updates the BSP port `Makefile` to do it:

Listing 7: https://github.com/FreeRTOS/FreeRTOS/commit/257d9e1d123be0342029e2930c0073dd5a4a2b2d

```
--- a/FreeRTOS/Demo/CORTEX_LM3S811_GCC/Makefile
+++ b/FreeRTOS/Demo/CORTEX_LM3S811_GCC/Makefile
@@ -29,8 +29,10 @@ RTOS_SOURCE_DIR=../../Source
 DEMO_SOURCE_DIR=../Common/Minimal

 CFLAGS+=-I hw_include -I . -I ${RTOS_SOURCE_DIR}/include -I ${RTOS_SOURCE_DIR}/portable/GCC/
↪ARM_CM3 -I ../Common/include -D GCC_ARMCM3_LM3S102 -D inline=
```

(continues on next page)

(continued from previous page)

```
+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
arm-none-eabi-ld: failed to merge target specific data of file microej/lib/microejruntime.
↪a(sni_vm_startup_greenthread.o)
arm-none-eabi-ld: gcc/RTOSDemo.axf section `ICETEA_HEAP' will not fit in region `SRAM'
arm-none-eabi-ld: region `SRAM' overflowed by 4016 bytes
microej/lib/microejapp.o: In function `_java_internStrings_end':
```

The RAM requirements of the BSP (with printf), FreeRTOS, the MicroEJ Application and MicroEJ Runtime do not fit in the 8k of SRAM. It is possible to link within 8k of RAM by customizing a *MicroEJ Tiny-Sandbox* on a baremetal device (without a RTOS) but this is not the purpose of this 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`:

Listing 8: https://github.com/MicroEJ/FreeRTOS/commit/0e2e31d8a510d37178c340051bab636902471eea

```
diff --git a/FreeRTOS/Demo/CORTEX_LM3S6965_GCC/standalone.ld b/FreeRTOS/Demo/CORTEX_LM3S6965_
↪GCC/standalone.ld
--- a/FreeRTOS/Demo/CORTEX_LM3S6965_GCC/standalone.ld
+++ b/FreeRTOS/Demo/CORTEX_LM3S6965_GCC/standalone.ld
@@ -28,8 +28,8 @@

 MEMORY
 {
-    FLASH (rx) : ORIGIN = 0x00000000, LENGTH = 64K
-    SRAM (rwx) : ORIGIN = 0x20000000, LENGTH = 8K
+    FLASH (rx) : ORIGIN = 0x00000000, LENGTH = 256K
+    SRAM (rwx) : ORIGIN = 0x20000000, LENGTH = 64K
 }

 SECTIONS
```

The new command to run the firmware with QEMU is: `qemu-system-arm -M lm3s6965evb -nographic -kernel gcc/RTOSDemo.bin`.

Rebuild the firmware with `make`. The following error occurs:

```
  CC    microej/src/LLMJVM_stub.c
  LD    gcc/RTOSDemo.axf
↪                                                                                ␣
↪                                                                                ␣
↪                                                    microej/lib/microejapp.o: In function␣
↪`_java_internStrings_end':
C:\Users\user\src\tuto-from-scratch\workspace\HelloWorld\com.mycompany.Main\SOAR.o:(.text.
↪soar+0x1b3e): undefined reference to `ist_mowana_vm_GenericNativesPool___com_1is2t_1vm_
↪1support_1lang_1SupportNumber_1parseLong'
C:\Users\user\src\tuto-from-scratch\workspace\HelloWorld\com.mycompany.Main\SOAR.o:(.text.
↪soar+0x1cea): undefined reference to `ist_mowana_vm_GenericNativesPool___com_1is2t_1vm_
↪1support_1lang_1SupportNumber_1toStringLongNative'         C:\Users\user\src\tuto-from-
↪scratch\workspace\HelloWorld\com.mycompany.Main\SOAR.o:(.text.soar+0x1e3e): undefined␣
↪reference to `ist_mowana_vm_GenericNativesPool___com_1is2t_1vm_1support_1lang_1Systools_
↪1appendInteger'
C:\Users\user\src\tuto-from-scratch\workspace\HelloWorld\com.mycompany.Main\SOAR.o:(.text.
↪soar+0x1f2a): undefined reference to `ist_mowana_vm_GenericNativesPool___java_1lang_
↪1System_1getMethodClass'
C:\Users\user\src\tuto-from-scratch\workspace\HelloWorld\com.mycompany.Main\SOAR.o:(.text.
↪soar+0x1e3e): undefined reference to `ist_mowana_vm_GenericNativesPool___com_1is2t_1vm_
↪1support_1lang_1Systools_1appen
... skip ...
C:\Users\user\src\tuto-from-scratch\workspace\HelloWorld\com.mycompany.Main\SOAR.o:(.text.
↪soar+0x31d6): undefined reference to `ist_mowana_vm_GenericNativesPool___java_1lang_
↪1System_1initializeProperties'
C:\Users\user\src\tuto-from-scratch\workspace\HelloWorld\com.mycompany.Main\SOAR.o:(.text.
↪soar+0x37b6): undefined reference to `ist_mowana_vm_GenericNativesPool___java_1lang_
↪1Thread_1storeException'
C:\Users\user\src\tuto-from-scratch\workspace\HelloWorld\com.mycompany.Main\SOAR.o:(.text.
↪soar+0x37c8): undefined reference to `ist_microjvm_NativesPool___java_1lang_1Thread_
```

(continues on next page)

(continued from previous page)

```
↪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.

Listing 9: https://github.com/MicroEJ/FreeRTOS/commit/4b23ea2e77112f053368718d299ff8db826ddde1

```
diff --git a/FreeRTOS/Demo/CORTEX_LM3S6965_GCC/makedefs b/FreeRTOS/Demo/CORTEX_LM3S6965_GCC/
↪makedefs
--- a/FreeRTOS/Demo/CORTEX_LM3S6965_GCC/makedefs
+++ b/FreeRTOS/Demo/CORTEX_LM3S6965_GCC/makedefs
@@ -196,13 +196,13 @@ ifeq (${COMPILER}, gcc)
             echo ${LD} -T ${SCATTER_${notdir ${@:.axf=}}}    \
                        --entry ${ENTRY_${notdir ${@:.axf=}}} \
                        ${LDFLAGSgcc_${notdir ${@:.axf=}}}    \
-                       ${LDFLAGS} -o ${@} ${^}               \
-                       '${LIBC}' '${LIBGCC}';                \
+                       ${LDFLAGS} -o ${@} --start-group ${^} \
+                       '${LIBC}' '${LIBGCC}' --end-group;    \
         fi
         @${LD} -T ${SCATTER_${notdir ${@:.axf=}}}    \
                --entry ${ENTRY_${notdir ${@:.axf=}}} \
                ${LDFLAGSgcc_${notdir ${@:.axf=}}}    \
-               ${LDFLAGS} -o ${@} ${^}               \
-               '${LIBC}' '${LIBGCC}'
+               ${LDFLAGS} -o ${@} --start-group ${^} \
+               '${LIBC}' '${LIBGCC}' --end-group
         @${OBJCOPY} -O binary ${@} ${@:.axf=.bin}
 endif
```

Rebuild with `make`. The following error occurs:

```
 LD    gcc/RTOSDemo.axf
microej/lib/microejruntime.a(VMCOREMicroJvm__131.o): In function `VMCOREMicroJvm__1131____1_
↪11046':
_131.c:(.text.VMCOREMicroJvm__1131____1_11046+0x20): undefined reference to `fmodf'
microej/lib/microejruntime.a(VMCOREMicroJvm__131.o): In function `VMCOREMicroJvm__1131____1_
↪11045':
_131.c:(.text.VMCOREMicroJvm__1131____1_11045+0x2c): undefined reference to `fmod'
microej/lib/microejruntime.a(iceTea_lang_Math.o): In function `iceTea_lang_Math___cos':
Math.c:(.text.iceTea_lang_Math___cos+0x2a): undefined reference to `cos'
microej/lib/microejruntime.a(iceTea_lang_Math.o): In function `iceTea_lang_Math___sin':
Math.c:(.text.iceTea_lang_Math___sin+0x2a): undefined reference to `sin'
microej/lib/microejruntime.a(iceTea_lang_Math.o): In function `iceTea_lang_Math___tan':
Math.c:(.text.iceTea_lang_Math___tan+0x2a): undefined reference to `tan'
microej/lib/microejruntime.a(iceTea_lang_Math.o): In function `iceTea_lang_Math___acos__D':
Math.c:(.text.iceTea_lang_Math___acos__D+0x18): undefined reference to `acos'
microej/lib/microejruntime.a(iceTea_lang_Math.o): In function `iceTea_lang_Math___acos(void)
↪':
Math.c:(.text.iceTea_lang_Math___acos__F+0x12): undefined reference to `acosf'
microej/lib/microejruntime.a(iceTea_lang_Math.o): In function `iceTea_lang_Math___asin':
Math.c:(.text.iceTea_lang_Math___asin+0x18): undefined reference to `asin'
microej/lib/microejruntime.a(iceTea_lang_Math.o): In function `iceTea_lang_Math___atan':
Math.c:(.text.iceTea_lang_Math___atan+0x2): undefined reference to `atan'
microej/lib/microejruntime.a(iceTea_lang_Math.o): In function `iceTea_lang_Math___atan2':
Math.c:(.text.iceTea_lang_Math___atan2+0x2): undefined reference to `atan2'
microej/lib/microejruntime.a(iceTea_lang_Math.o): In function `iceTea_lang_Math___log':
Math.c:(.text.iceTea_lang_Math___log+0x2): undefined reference to `log'
microej/lib/microejruntime.a(iceTea_lang_Math.o): In function `iceTea_lang_Math_(...)(long␣
↪long,  *)':
Math.c:(.text.iceTea_lang_Math___exp+0x2): undefined reference to `exp'
microej/lib/microejruntime.a(iceTea_lang_Math.o): In function `iceTea_lang_Math_(char,...
↪)(int, long)':
Math.c:(.text.iceTea_lang_Math___ceil+0x2): undefined reference to `ceil'
microej/lib/microejruntime.a(iceTea_lang_Math.o): In function `iceTea_lang_Math___floor':
... skip ...
```

This error occurs because the Math library is missing. The rule for linking the firmware is defined in the file `makedefs`. Replicating how the libc is managed, the following patch finds the `libm.a` library and add it at link time:

Listing 10: https://github.com/MicroEJ/FreeRTOS/commit/a202f43948c41b848ebfbc8c53610028c454b66f

```
diff --git a/FreeRTOS/Demo/CORTEX_LM3S6965_GCC/makedefs b/FreeRTOS/Demo/CORTEX_LM3S6965_GCC/
↪makedefs
--- a/FreeRTOS/Demo/CORTEX_LM3S6965_GCC/makedefs
+++ b/FreeRTOS/Demo/CORTEX_LM3S6965_GCC/makedefs
@@ -102,6 +102,11 @@ LIBGCC=${shell ${CC} -mthumb -march=armv6t2 -print-libgcc-file-name}
 #
 LIBC=${shell ${CC} -mthumb -march=armv6t2 -print-file-name=libc.a}

+#
+# Get the location of libm.a from the GCC front-end.
+#
+LIBM=${shell ${CC} -mthumb -march=armv6t2 -print-file-name=libm.a}
```

(continues on next page)

(continued from previous page)

```
+
 #
 # The command for extracting images from the linked executables.
 #
@@ -197,12 +202,12 @@ ifeq (${COMPILER}, gcc)
                         --entry ${ENTRY_${notdir ${@:.axf=}}} \
                         ${LDFLAGSgcc_${notdir ${@:.axf=}}}    \
                         ${LDFLAGS} -o ${@} --start-group ${^} \
-                        '${LIBC}' '${LIBGCC}' --end-group;    \
+                        '${LIBM}' '${LIBC}' '${LIBGCC}' --end-group; \
          fi
         @${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.

Listing 11: https://github.com/MicroEJ/FreeRTOS/commit/eb208d846f52c0695c06456b540e412ba96e640a

```
diff --git a/FreeRTOS/Demo/CORTEX_LM3S6965_GCC/makedefs b/FreeRTOS/Demo/CORTEX_LM3S6965_GCC/
↪makedefs
--- a/FreeRTOS/Demo/CORTEX_LM3S6965_GCC/makedefs
+++ b/FreeRTOS/Demo/CORTEX_LM3S6965_GCC/makedefs
@@ -107,6 +107,11 @@ LIBC=${shell ${CC} -mthumb -march=armv6t2 -print-file-name=libc.a}
 #
 LIBM=${shell ${CC} -mthumb -march=armv6t2 -print-file-name=libm.a}

+#
+# Get the location of libnosys.a from the GCC front-end.
+#
+LIBNOSYS=${shell ${CC} -mthumb -march=armv6t2 -print-file-name=libnosys.a}
+
 #
 # The command for extracting images from the linked executables.
 #
```

(continues on next page)

(continued from previous page)

```
@@ -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.

Listing 12: https://github.com/MicroEJ/FreeRTOS/commit/898f2e6cd492616b4ccaabc136cafa76ef038690

```
diff --git a/FreeRTOS/Demo/CORTEX_LM3S6965_GCC/standalone.ld b/FreeRTOS/Demo/CORTEX_LM3S6965_
↪GCC/standalone.ld
--- a/FreeRTOS/Demo/CORTEX_LM3S6965_GCC/standalone.ld
+++ b/FreeRTOS/Demo/CORTEX_LM3S6965_GCC/standalone.ld
@@ -64,5 +64,6 @@ SECTIONS
         *(.bss)
         *(COMMON)
         _ebss = .;
+        end = .;
     } > SRAM
 }
```

Then rebuild with `make`. There should be no error. Finally, run the firmware in QEMU with the following command:

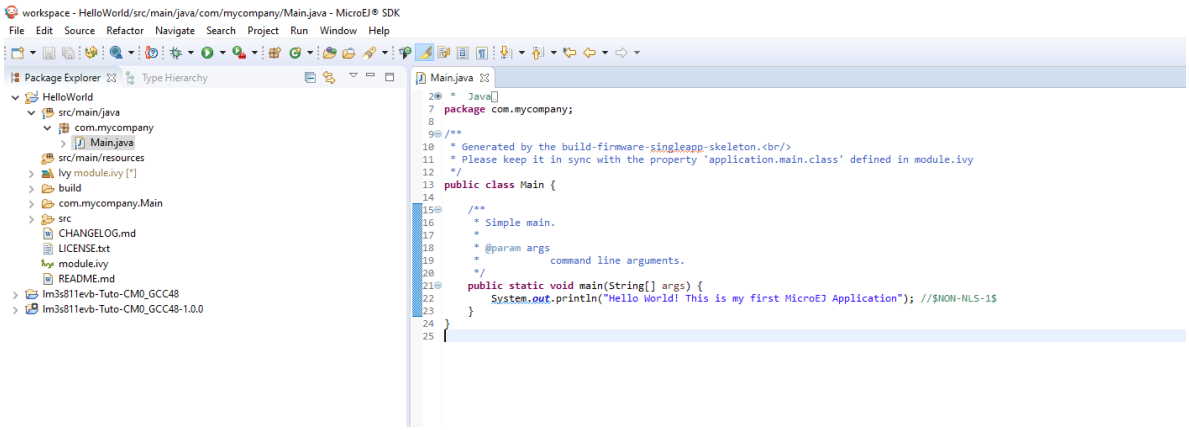
```
qemu-system-arm -M lm3s6965evb -nographic -kernel gcc/RTOSDemo.bin
```

```
Hello, World! printf function is working.
Hello World!
QEMU: Terminated // press Ctrl-a x to end the QEMU session
```

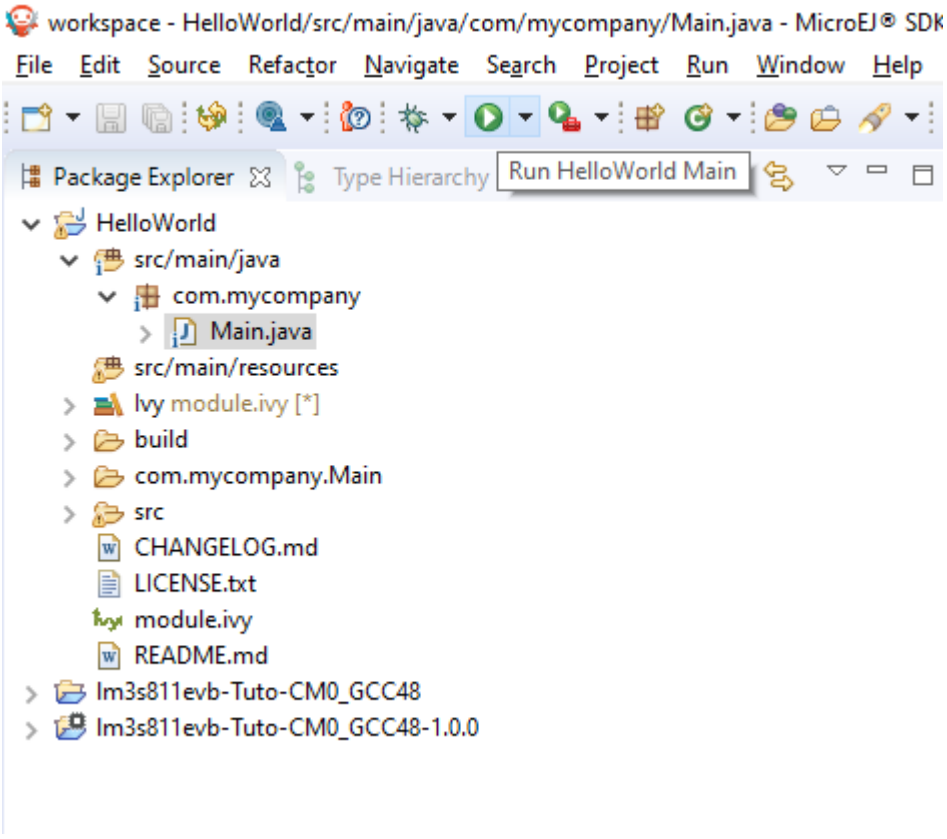
The first `Hello, World!` is from the `main.c` and the second one from the MicroEJ Application.

To make this more obvious:

1. Update the MicroEJ Application to print `Hello World! This is my first MicroEJ Application`



2. Rebuild the MicroEJ Application



On success, the following message appears in the console:

```
=============== [ Initialization Stage ] ===============
Platform connected to BSP location 'C:\Users\user\src\tuto-from-scratch\FreeRTOS\
↪FreeRTOS\Demo\CORTEX_LM3S6965_GCC' using application option 'deploy.bsp.root.dir'.
=============== [ Launching SOAR ] ===============
```

(continues on next page)

(continued from previous page)

```
=============== [ 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.
- Follow the *Create MicroEJ Platform Build and Run Scripts* tutorial to get this MicroEJ Platform fully automated for build and execution.

9.4 Add IAR to MicroEJ SDK Docker Image

This document presents how to create a Dockerfile with MicroEJ SDK version 5.x and Cross-platform Build Tools for Arm to build a MicroEJ application. You can use this image in your automated CI.

9.4.1 Prerequisites

- A recent version of IAR BXARM and its user licence.

This tutorial was tested with MicroEJ SDK `5.8.1-jdk11`, IAR `9.30.1`, and Docker `24.0.6`.

9.4.2 Create the Dockerfile

Here is our final Dockerfile. We will explain each specific step below.

```
FROM microej/sdk:5.8.1-jdk11

USER root
SHELL ["/bin/bash", "-c"]

ARG IAR_BXARM_VERSION=9.30.1
ARG IAR_BXARM_PACKAGE="bxarm-$IAR_BXARM_VERSION.deb"

COPY ressources/$IAR_BXARM_PACKAGE /tmp/$IAR_BXARM_PACKAGE
RUN apt-get update && apt-get install sudo libsqlite3-0 libxml2 tzdata dos2unix /
↪tmp/$IAR_BXARM_PACKAGE -y && \
    apt-get clean autoclean autoremove && rm -rf /var/lib/apt/lists/* /tmp/*.deb

ENV PATH="/opt/iarsystems/bxarm/arm/bin/:/opt/iarsystems/bxarm/common/bin/:$PATH"
ENV IAR_LICENSE_SERVER=$IAR_LICENSE_SERVER_IP

# Set workdir
WORKDIR ${HOME}

ADD run.sh /run.sh
RUN chmod a+x /run.sh

# Good practice, switch back to user.
USER ${user}

ENTRYPOINT ["/run.sh"]
```

1. In a new directory create a file named `Dockerfile`.
2. We use MicroEJ SDK base image, they are available on docker hub. In your Dockerfile add this code:

   ```
   FROM microej/sdk:5.8.1-jdk11
   ```

3. Add IAR BXARM deb package in a directory named `resources`.
4. Add the package info to your Dockerfile (update the version with the one you want to use):

```
ARG IAR_BXARM_VERSION=9.30.1
ARG IAR_BXARM_PACKAGE="bxarm-$IAR_BXARM_VERSION.deb"
```

5. Copy the package to a temporary directory.

```
COPY ressources/$IAR_BXARM_PACKAGE /tmp/$IAR_BXARM_PACKAGE
```

6. Install this package along with any others required packages.

```
RUN apt-get update && apt-get install sudo libsqlite3-0 libxml2 tzdata dos2unix␣
↪/tmp/$IAR_BXARM_PACKAGE -y && \
apt-get clean autoclean autoremove && rm -rf /var/lib/apt/lists/* /tmp/*.deb
```

7. Set IAR path and license server address:

```
ENV PATH="/opt/iarsystems/bxarm/arm/bin/:/opt/iarsystems/bxarm/common/bin/:$PATH
↪"
ENV IAR_LICENSE_SERVER=$IAR_LICENSE_SERVER_IP
```

8. Finally create a `run.sh` script with the following content:

```
lightlicensemanager setup -s license.iar.public
exec "$@"
```

9.5 Create MicroEJ Platform Build and Run Scripts

This tutorial describes all the steps to create *MicroEJ Platform* build and run scripts and shows how to use them.

9.5.1 Intended Audience

The audience for this document is Platform engineers who want to

- validate their MicroEJ Platform using automated *MicroEJ test suites*.
- prepare their MicroEJ Platform for automated builds and continuous integration using *MicroEJ Module Manager*.
- ease *MicroEJ Standalone Application* development by simplifying the Firmware build for Java developers.
- configure their MicroEJ Platform with full *BSP connection*.

9.5.2 Prerequisites

This tutorial is a direct continuation of *Create a MicroEJ Firmware From Scratch* tutorial. It should have been completed before starting this one.

9.5.3 Introduction

Build and Run scripts are normalized entry points to

- build a MicroEJ Firmware linked to the Board Support Package,
- deploy and run the Firmware on a device.

External tools only need to run these scripts without additional knowledge about the toolchain or deployment tools.

See *Build Script File* and *Run Script File* sections for more information about these scripts. Script examples are provided in Platform Qualification Tools repository.

9.5.4 Overview

In the previous *Create a MicroEJ Firmware From Scratch* tutorial, the final binary is produced by invoking `make` in the FreeRTOS BSP. The command to type is dependant of the toolchain used. The Firmware is then executed in QEMU but could have been instead flashed to a device with another specific command. This tutorial explain how to write *build* and *run* scripts for these two tasks.

The next sections will

- describe step-by-step how to create the build and run scripts both for unix-like systems (Bash scripts) and Windows systems (batch files). These scripts automate Firmware build and execution in QEMU as presented in *Create a MicroEJ Firmware From Scratch* tutorial.
- show a practical usage of these scripts in a MicroEJ development flow. This will allow to configure a MicroEJ Standalone Application to build the Firmware in MicroEJ SDK.

Finally, this tutorial describes how to convert the MicroEJ Platform from partial BSP connection to full BSP connection.

9.5.5 Create Build and Run Scripts

This section describes how to write build and run scripts.

There are two scripts:

1. `build.[sh|bat]` which calls the C toolchain to build and link the Firmware file. It also ensures that the output file is called `application.out` and is located in the directory from where the script was called.
2. `run.[sh|bat]` which deploys and runs `application.out` on the device. In this tutorial, it will only run the Firmware with QEMU instead of flashing it on real hardware.

Each of these scripts come in two flavors: `.sh` for unix-like systems, and `.bat` for Windows systems.

First, create a `microej/scripts` directory in BSP project:

```
$ pwd
/mnt/c/Users/user/src/tuto-from-scratch/FreeRTOS/FreeRTOS/Demo/CORTEX_LM3S6965_GCC
$ mkdir microej/scripts
```

Note: The scripts created in the next sections will be put in this directory.

Create *build.sh* and *run.sh* Scripts

Warning: Make sure the build and run scripts have the execution permission.

1. Create a file called `build.sh` in the `microej/scripts` directory with the following content:

```
#!/bin/bash

# Save application current directory and jump one level above scripts
CURRENT_DIRECTORY=$(pwd)

# Move to the directory where the Makefile is
cd $(dirname "$0")/../..

# Build the firmware
make

# Copy output the the current directory while renaming it
cp gcc/RTOSDemo.bin  $CURRENT_DIRECTORY/application.out

# Restore application directory
cd $CURRENT_DIRECTORY/
```

2. Verify that the script successfully built your Firmware and put it in the current directory with the name `application.out`.

```
$ pwd
/mnt/c/Users/user/src/tuto-from-scratch/FreeRTOS/FreeRTOS/Demo/CORTEX_LM3S6965_GCC
$ make clean
$ microej/scripts/build.sh
  CC    init/startup.c
  CC    main.c
  CC    ../../Source/list.c
  CC    ../../Source/queue.c
  CC    ../../Source/tasks.c
[..]
  130 |  __attribute__( ( always_inline ) ) static inline uint8_t␣
↪ucPortCountLeadingZeros( uint32_t ulBitmap )
      |                                                          ^~~~~~~~~~~~~~~~~
↪~~~~~~
  LD    gcc/RTOSDemo.axf
$ ls *.out
application.out
```

3. Check that `application.out` successfully runs with QEMU:

```
$ qemu-system-arm -M lm3s6965evb -nographic -kernel application.out
Hello, World! printf function is working.
Hello World!
QEMU: Terminated // press Ctrl-a x to end the QEMU session
```

4. Create a file called `run.sh` in the `microej/scripts` directory with the following content:

```
#!/bin/bash

# Add some text to the console before launch
echo -e "\033[0;32m## Start application in QEMU."
echo -e "## Use 'Ctrl-a x' to quit.\e[0m"

# Launch application with QEMU
qemu-system-arm -M lm3s6965evb -nographic -kernel application.out
```

5. We can now run the Firmware we just built with the `run.sh` script:

```
$ pwd
/mnt/c/Users/user/src/tuto-from-scratch/FreeRTOS/FreeRTOS/Demo/CORTEX_LM3S6965_GCC
$ microej/scripts/run.sh
## Start application in QEMU.
## Use 'Ctrl-a x' to quit.
Hello, World! printf function is working.
Hello World!
```

Note: This script is very simple because our Firmware is just run with QEMU instead of real hardware. To deploy the Firmware on a device, the script would have to setup and call a flash tool. See for instance the build and run scripts for Espressif-ESP-WROVER-KIT-V4.1.

Create *build.bat* and *run.bat* Scripts

As our toolchain has only be configured for Linux in WSL, we create wrappers that call shell scripts through WSL. We could also decide to directly invoke QEMU for Windows instead. This is just a implementation choice for this Platform.

1. Create a file called `build.bat` in the `microej/scripts` directory with the following content:

```
@echo off
SETLOCAL ENABLEEXTENSIONS

REM Reset ERRORLEVEL between multiple run in the same shell
SET ERRORLEVEL=0

REM Save application current directory and jump to scripts directory
SET CURRENT_DIRECTORY=%CD%
CD "%~dp0"

REM Get the script directory in a Unix path format
FOR /F %%i in ('WSL pwd') DO SET SCRIPT_DIRECTORY=%%i

REM Restore application directory
CD %CURRENT_DIRECTORY%

REM Run the bash build script with WSL
WSL %SCRIPT_DIRECTORY%/build.sh

IF %ERRORLEVEL% NEQ 0 (
```

(continues on next page)

(continued from previous page)

```
      EXIT /B %ERRORLEVEL%
)
```

2. Calling this script in PowerShell should produce the following result:

```
PS C:\Users\user\src\tuto-from-scratch\FreeRTOS\FreeRTOS\Demo\CORTEX_LM3S6965_GCC>␣
↪microej\scripts\build.bat
  CC    init/startup.c
  CC    main.c
  CC    ../../Source/list.c
  CC    ../../Source/queue.c
  CC    ../../Source/tasks.c
 [...]
  CC    microej/src/LLMJVM_stub.c
  LD    gcc/RTOSDemo.axf
Current DIR /mnt/c/Users/user/src/tuto-from-scratch/FreeRTOS/FreeRTOS/Demo/CORTEX_
↪LM3S6965_GCC/microej/scripts
        1 file(s) moved.
```

Note: This prints the full build output if it is the first build (or after a `make clean`) otherwise it prints `make: Nothing to be done for 'all'`.

3. Create a file called `run.bat` in the `microej/scripts` directory with the following content:

```
@echo off
SETLOCAL ENABLEEXTENSIONS

REM Reset ERRORLEVEL between multiple run in the same shell
SET ERRORLEVEL=0

REM Save application current directory and jump to scripts directory
SET CURRENT_DIRECTORY=%CD%
CD "%~dp0"

REM Get the script directory in a Unix path format
FOR /F %%i in ('WSL pwd') DO SET SCRIPT_DIRECTORY=%%i

REM Restore application directory
CD %CURRENT_DIRECTORY%

REM Run the bash run script with WSL
WSL %SCRIPT_DIRECTORY%/run.sh

IF %ERRORLEVEL% NEQ 0 (
    EXIT /B %ERRORLEVEL%
)
```

4. Calling this script in PowerShell should produce the following result:

```
C:\Users\user\src\tuto-from-scratch\FreeRTOS\FreeRTOS\Demo\CORTEX_LM3S6965_GCC\
↪application.out
```

(continues on next page)

(continued from previous page)

```
1 file(s) copied.
## Start application in QEMU.
## Use 'Ctrl-a x' to quit.
Hello, World! printf function is working.
Hello World!
```

9.5.6 Use Build Script in MicroEJ SDK

In this section, we illustrate how build script is used in practice to ease the Firmware build for Java developers in MicroEJ SDK.

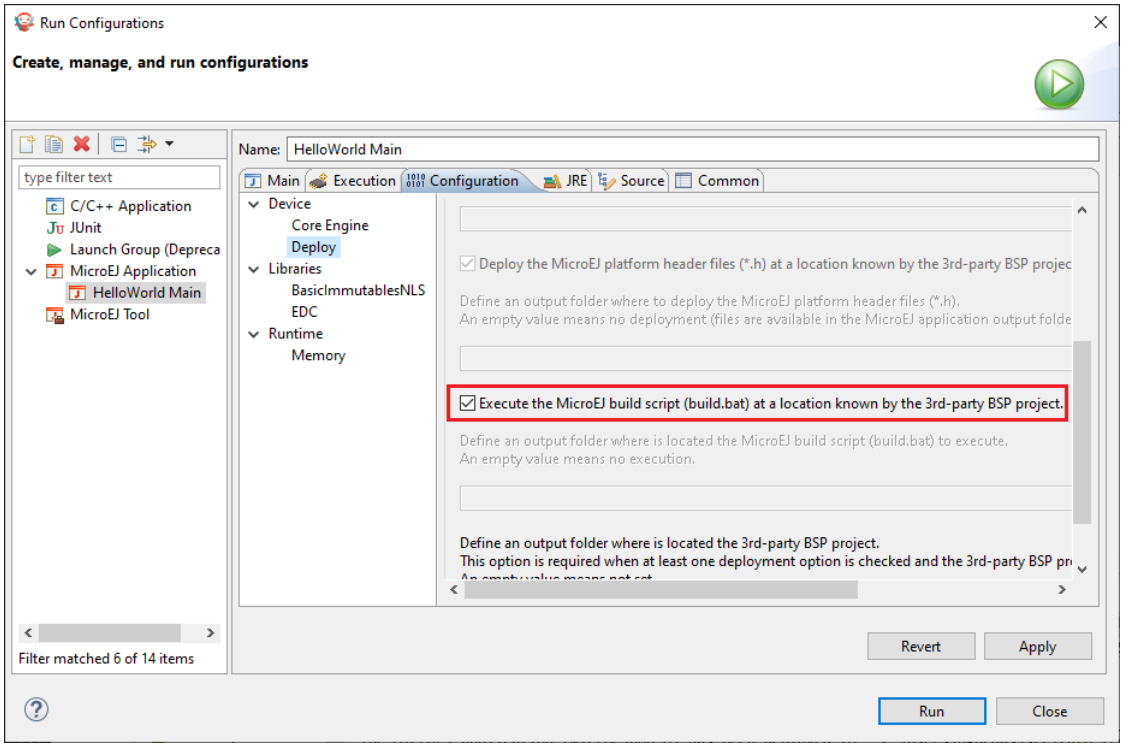
We will configure a MicroEJ Standalone Application to enable full Firmware build (application + BSP + link) when building the *HelloWorld* application.

We will then configure a full BSP connection. This will remove the need to configure the path of the BSP root directory as a MicroEJ Standalone Application option. Please refer to *BSP connection cases* section and *BSP connection options* for more details.

Note: Build and run scripts do not require to configure a full BSP connection. This last part has only be added to allow a MicroEJ Standalone Application project to be built independently from the BSP.

Build Firmware from MicroEJ SDK

1. Right click on the `HelloWorld` application project
2. In the menu, select `Run As` > `Run Configurations...`
3. Select the `Configuration` tab
4. Select `Device` > `Deploy` entry in the configurations menu
5. Check `Execute the MicroEJ script (build.bat) at the location known by the 3rd-party BSP project` checkbox



6. Click on the `Run` button. It should print the following:

```
=============== [ Initialization Stage ] ===============
Platform connected to BSP location 'C:\Users\user\src\tuto-from-scratch\FreeRTOS\
↪FreeRTOS\Demo\CORTEX_LM3S6965_GCC' using application option 'deploy.bsp.root.dir
↪'.
[INFO ] Launching in Evaluation mode. Your UID is 0120202834374C4A.
=============== [ Launching SOAR ] ===============
=============== [ Launching Link ] ===============
=============== [ Deployment ] ===============
MicroEJ files for the 3rd-party BSP project are generated to 'C:\Users\user\src\
↪tuto-from-scratch\workspace\HelloWorld\com.mycompany.Main\platform'.

FAIL
The following error occurred while executing this line:
C:\Users\user\src\tuto-from-scratch\workspace\lm3s811evb-Platform-CM0_GCC48-0.0.1\
↪source\scripts\deploy.xml:30: The following error occurred while executing this␣
↪line:
C:\Users\user\src\tuto-from-scratch\workspace\lm3s811evb-Platform-CM0_GCC48-0.0.1\
↪source\scripts\deployInBSP.xml:97: The following error occurred while executing␣
↪this line:
C:\Users\user\src\tuto-from-scratch\workspace\lm3s811evb-Platform-CM0_GCC48-0.0.1\
↪source\scripts\deployInBSP.xml:260: Option 'deploy.bsp.microejscript' is enabled␣
↪but this Platform does no define a well-known location. Either update the␣
↪Platform configuration (option 'deploy.bsp.microejscript.relative.dir' in 'bsp/
↪bsp.properties') or disable this option.
```

7. Edit the file `bsp/bsp.properties` as follow:

```
# Specify BSP external scripts files ('build.bat' and 'run.bat') parent directory.
# This is a '/' separated directory relative to 'bsp.root.dir'.
microejscript.relative.dir=microej/scripts
```

8. Rebuild your Platform (right-click on the platform configuration project and click on Build Module)
9. Run the *HelloWorld* launcher once again. This should print the following result:

```
=============== [ Initialization Stage ] ===============
Platform connected to BSP location 'C:\Users\user\src\tuto-from-scratch\FreeRTOS\
↪FreeRTOS\Demo\CORTEX_LM3S6965_GCC' using platform option 'deploy.bsp.root.dir'.
[INFO ] Launching in Evaluation mode. Your UID is 0120202834374C4A.===============␣
↪[ Launching SOAR ] ===============
=============== [ Launching Link ] ===============
=============== [ Deployment ] ===============
MicroEJ files for the 3rd-party BSP project are generated to 'C:\Users\user\
↪Workspaces\_test_fw_tuto\HelloWorld\com.mycompany.Main\platform'.
The MicroEJ application (microejapp.o) has been deployed to: 'C:\Users\user\src\
↪tuto-from-scratch\FreeRTOS\FreeRTOS\Demo\CORTEX_LM3S6965_GCC\microej\lib'.
The MicroEJ platform library (microejruntime.a) has been deployed to: 'C:\Users\
↪user\src\tuto-from-scratch\FreeRTOS\FreeRTOS\Demo\CORTEX_LM3S6965_GCC\microej\lib
↪'.
The MicroEJ platform header files (*.h) have been deployed to: 'C:\Users\user\src\
↪tuto-from-scratch\FreeRTOS\FreeRTOS\Demo\CORTEX_LM3S6965_GCC\microej\inc'.
Execution of script 'C:\Users\user\src\tuto-from-scratch\FreeRTOS\FreeRTOS\Demo\
↪CORTEX_LM3S6965_GCC\microej\scripts\build.bat' started...
LD    gcc/RTOSDemo.axf
Current DIR /mnt/c/Users/user/Workspaces/_test_fw_tuto/HelloWorld/com.mycompany.
↪Main
Execution of script 'C:\Users\user\src\tuto-from-scratch\FreeRTOS\FreeRTOS\Demo\
↪CORTEX_LM3S6965_GCC\microej\scripts\build.bat' done.
=============== [ Completed Successfully ] ===============

SUCCESS
```

Reading the traces, we see that the *HelloWorld* application (*microejapp.o*) and the MicroEJ Platform library (*microejruntime.a*) have been deployed to the suitable BSP location. Then the `build.bat` script has been executed to rebuild the BSP and link the Firmware. The output is the `application.out` binary that can be flashed on the device (or run on QEMU).

Convert from partial BSP connection to full BSP connection (optional)

In this section, we configure the BSP root directory in the Platform. Such configuration is called *full BSP connection*: the MicroEJ Platform includes the BSP, and any MicroEJ Standalone Application can be built against this MicroEJ Platform without extra configuration.

When launching the `HelloWorld` application from MicroEJ SDK, the launcher knows how to find the BSP because we have configured its path in `HelloWorld/build/emb.properties` file which is imported in the launcher (this file has been configured in *Create a MicroEJ Firmware From Scratch* tutorial).

1. Cut `deploy.bsp.root.dir` property value from `HelloWorld/build/emb.properties` file
2. Paste the value in `bsp/bsp.properties` as follow:

```
# Specify the BSP root directory. Can use ${project.parent.dir} which target the␣
↪parent of platform configuration project
# For example, '${workspace}/${project.prefix}-bsp' specifies a BSP project beside␣
↪the '-configuration' project
root.dir=[absolute_path] to FreeRTOS\\FreeRTOS\\Demo\\CORTEX_LM3S811_GCC
```

3. Rebuild your MicroEJ Platform (right-click on the platform configuration project and click on Build Module)

   The MicroEJ Platform is now fully connected to the BSP.

4. Launch `HelloWorld` project from Eclipse launcher, it should print the following result:

```
=============== [ Initialization Stage ] ===============
Platform connected to BSP location 'C:\Users\user\src\tuto-from-scratch\
↪FreeRTOS\FreeRTOS\Demo\CORTEX_LM3S6965_GCC' using platform option 'root.
↪dir' in 'bsp/bsp.properties'.
[INFO ] Launching in Evaluation mode. Your UID is 0120202834374C4A.
=============== [ Launching SOAR ] ===============
=============== [ Launching Link ] ===============
=============== [ Deployment ] ===============
MicroEJ files for the 3rd-party BSP project are generated to 'C:\Users\
↪user\src\tuto-from-scratch\workspace\HelloWorld\com.mycompany.Main\
↪platform'.
The MicroEJ application (microejapp.o) has been deployed to: 'C:\Users\
↪user\src\tuto-from-scratch\FreeRTOS\FreeRTOS\Demo\CORTEX_LM3S6965_GCC\
↪microej\lib'.
The MicroEJ platform library (microejruntime.a) has been deployed to: 'C:\
↪Users\user\src\tuto-from-scratch\FreeRTOS\FreeRTOS\Demo\CORTEX_LM3S6965_
↪GCC\microej\lib'.
The MicroEJ platform header files (*.h) have been deployed to: 'C:\Users\
↪user\src\tuto-from-scratch\FreeRTOS\FreeRTOS\Demo\CORTEX_LM3S6965_GCC\
↪microej\inc'.
Execution of script 'C:\Users\user\src\tuto-from-scratch\FreeRTOS\FreeRTOS\
↪Demo\CORTEX_LM3S6965_GCC\microej\scripts\build.bat' started...
  LD    gcc/RTOSDemo.axf
Current DIR /mnt/c/Users/user/src/tuto-from-scratch/FreeRTOS/FreeRTOS/Demo/
↪CORTEX_LM3S6965_GCC/microej/scripts
Execution of script 'C:\Users\user\src\tuto-from-scratch\FreeRTOS\FreeRTOS\
↪Demo\CORTEX_LM3S6965_GC  C\microej\scripts\build.bat' done.
=============== [ Completed Successfully ] ===============

SUCCESS
```

Note: You can notice the difference in the second line of the trace that now says that the connection is `using platform option root.dir' in 'bsp/bsp.properties'` instead of `using platform option 'deploy.bsp.root.dir'` in the previous launch.

The application launcher does not know anymore where the BSP is located. Nevertheless it still builds a Firmware ready to be flashed. The link to the BSP is now configured in the MicroEJ Platform. Any MicroEJ Standalone Application can be built against this MicroEJ Platform with no BSP specific option.

9.5.7 Going Further

- More about build and run scripts in *Build Script File* and *Run Script File* sections
- Some build scripts examples from Platform Qualification Tools
- Perform the *Run a Test Suite on a Device* tutorial to learn how to run an automated testsuite
- Perform the *Setup an Automated Build using Jenkins and Artifactory* tutorial to learn how to automate the build of a MicroEJ Platform module

9.6 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, JFrog Artifactory and a Git plateform (you can also use Gitlab or Github for example).

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 1 hour to complete.

9.6.1 Intended Audience

The audience for this document is engineers who are in charge of integrating *MicroEJ Module Manager (MMM)* to their continuous integration environment.

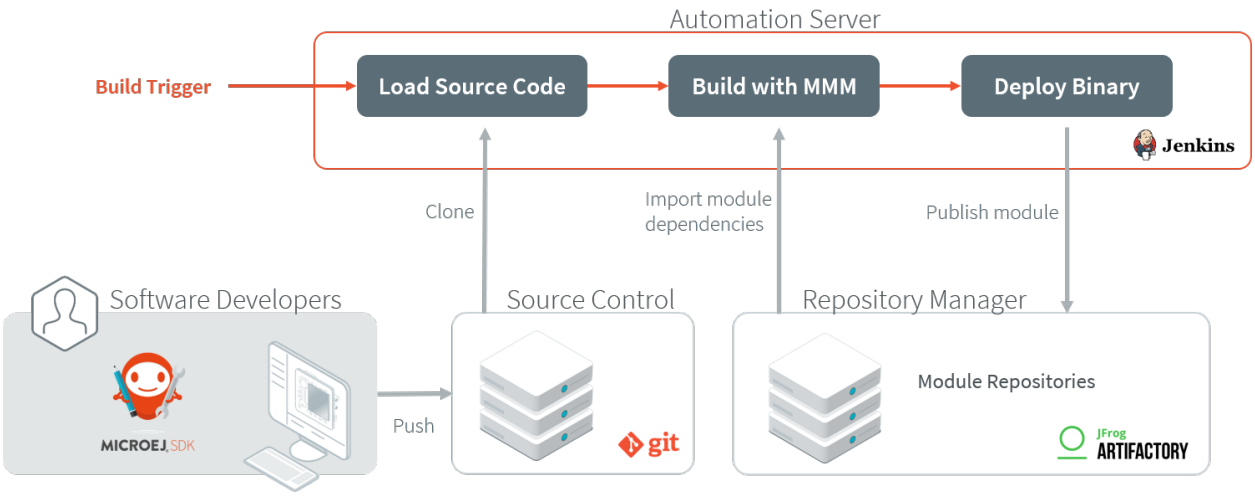
In addition, this tutorial should be of interest to all developers wishing to understand how MicroEJ works with headless module builds.

For those who are only interested by command line module build, consider using the *MMM Command Line Interface*.

9.6.2 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



9.6.3 Prerequisites

- MicroEJ SDK 5 `5.8.1` or higher.
- Docker and Docker Compose V2 on Linux, Windows or Mac
- Git `2.x` installed, with Git executable in path. We recommend installing Git Bash if your operating system is Windows (https://gitforwindows.org/).

This tutorial was tested with Jenkins `2.426.1`, Artifactory `7.71.5` and Gitea `1.21.1`.

Note: For SDK versions before 5.4.0, please refer to this MicroEJ Documentation Archive.

9.6.4 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. Run and setup Jenkins, Artifactory and Gitea
2. Create a simple MicroEJ module (Hello World)
3. Create a new Jenkins job for the Hello World module
4. Build the module

In order to simplify the steps, this tutorial will be performed locally on a single machine.

Artifactory will host MicroEJ modules in 3 repositories:

- `microej-module-repository`: repository initialized with pre-built MicroEJ modules, a mirror of the *Central Repository*
- `custom-modules-snapshot`: repository where custom snapshot modules will be published
- `custom-modules-release`: repository where custom release modules will be published

9.6.5 Prepare your Docker environment

This section assumes the prerequisites have been properly installed.

1. Create a new directory, inside create a file named `docker-compose.yaml` and copy this content:

```yaml
version: '3'
services:
  artifactory:
    image: releases-docker.jfrog.io/jfrog/artifactory-oss:7.71.5
    container_name: artifactory
    environment:
      - JF_ROUTER_ENTRYPOINTS_EXTERNALPORT=8082
    ports:
      - 8082:8082 # for router communication
      - 8081:8081 # for artifactory communication
      - 8085:8085 # for artifactory federation communication
    volumes:
      - artifactory:/var/opt/jfrog/artifactory
      - /etc/localtime:/etc/localtime:ro
    restart: always
    logging:
      driver: json-file
      options:
        max-size: "50m"
        max-file: "10"
    ulimits:
      nproc: 65535
      nofile:
        soft: 32000
        hard: 40000

  gitea:
    image: gitea/gitea:1.21.1
    container_name: gitea
    environment:
      - USER_UID=1000
      - USER_GID=1000
    restart: always
    volumes:
      - gitea:/data
      - /etc/timezone:/etc/timezone:ro
      - /etc/localtime:/etc/localtime:ro
    ports:
      - "3000:3000"
      - "222:22"

  jenkins:
    image: jenkins_master
    container_name: jenkins
    build:
      dockerfile: Dockerfile
    restart: always
    ports:
```

(continues on next page)

(continued from previous page)

```
      - 50000:50000
      - 8080:8080
    volumes:
      - jenkins:/var/jenkins_home
      - /var/run/docker.sock:/var/run/docker.sock
    links:
      - gitea
      - artifactory

volumes:
  gitea:
  artifactory:
  jenkins:
```

2. Create another file named `Dockerfile` and copy this content:

```
FROM jenkins/jenkins:2.426.1-lts
USER root
RUN apt-get update -qq \
    && apt-get install -qqy apt-transport-https ca-certificates curl gnupg2␣
↪software-properties-common
RUN curl -fsSL https://download.docker.com/linux/debian/gpg | apt-key add -
RUN add-apt-repository \
   "deb [arch=amd64] https://download.docker.com/linux/debian \
   $(lsb_release -cs) \
   stable"
RUN apt-get update  -qq \
    && apt-get -y install docker-ce
RUN usermod -aG docker jenkins
```

3. In this directory, launch the command `docker compose up -d`. After a few moments you should have three running containers (named jenkins, gitea and artifactory).

Using `docker compose ps` will show if containers started properly. Logs can be viewed with `docker compose logs`.

9.6.6 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.

This tutorial uses the MicroEJ Central Repository, which is the Module Repository used by MicroEJ SDK to fetch dependencies when starting an empty workspace. It bundles Foundation Library APIs and numerous Add-On Libraries.

Next step is to download a local copy of this repository:

1. Visit the Central Repository on the MicroEJ Developer website.
2. Navigate to the Production Setup section.
3. Click on the offline repository link. This will download the Central Repository as a ZIP file.

9.6.7 Setup Artifactory

Configure Artifactory

For demonstration purposes we will allow anonymous users to deploy modules in the repositories:

1. Once Artifactory container is started, go to `http://localhost:8082/`.
2. Login to Artifactory for the first time using the default `admin` account (Username: `admin`, Password: `password`).
3. Skip the installation wizard if it appears.
4. Go to `Administration` > `User Management` > `Settings`.
5. In the `User Security Configuration` section, check `Allow Anonymous Access`.
6. Click on `Save`.
7. Go to `Administration` > `User Management` > `Permissions`.
8. Click on `Anything` entry (do not check the line), then go to `Users` tab
9. Click on `anonymous` and check `Deploy/Cache` permission in the `Selected Users Repositories` category.
10. Click on `Save`.

Next steps will involve uploading large files, so we have to increase the file upload maximum size accordingly:

1. Go to `Administration` > `Artifactory` > `General` > `Settings`.
2. In the `General Settings` section, change the value of `File Upload In UI Max Size (MB)` to `1024` then click on `Save`.

Create Repositories

We will now create and configure the repositories. Let's start with the repository for the future built snapshot modules:

1. Go to `Administration` > `Repositories` > `Repositories` in the left menu.
2. Click on `Add Repositories` > `Local Repository`
3. Select `Maven`.
4. Set `Repository Key` field to `custom-modules-snapshot` and click on `Create Local Repository`.

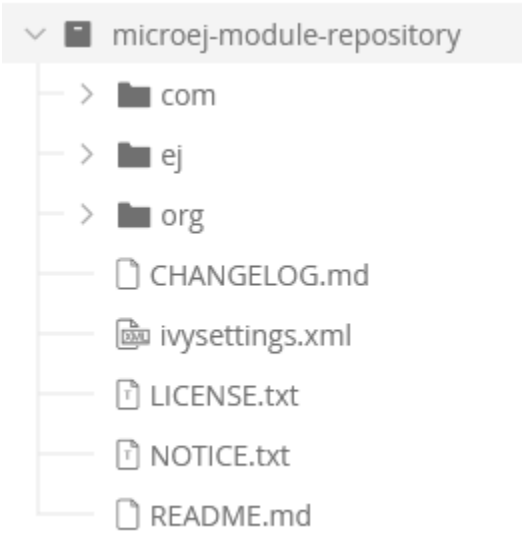
Repeat the same steps for the other repositories with the `Repository Key` field set to `custom-modules-release` and `microej-module-repository`.

Import MicroEJ Repositories

In this section, we will import MicroEJ repositories into Artifactory repositories to make them available to the build server.

1. Go to Administration > Artifactory > Import & Export > Repositories .
2. Scroll to the Import Repository from Zip section.
3. As Target Local Repository , select `microej-module-repository` in the list.
4. Click on Select file and select the MicroEJ module repository zip file (`central-repository-[version].zip`) that you downloaded earlier (please refer to section *Get a Module Repository*).
5. Click Upload . At the end of upload, click on Import . Upload and import may take some time.

Artifactory is now hosting all required MicroEJ modules. Go to Application > Artifactory > Artifacts and check that the repository `microej-module-repository` does contain modules as shown in the figure below.



9.6.8 Setup Gitea

Install Gitea

1. Once the Gitea container is started, go to `http://localhost:3000/` .
2. Don't change anything on the `Initial Configuration` , click on Install Gitea
3. Click on Register account and create one. The first created user become the administrator.

9.6.9 Configure Gitea

1. At the top right click on the arrow then `New Repository`
2. As `Repository Name` set `helloworld`, leave the other options as default.
3. Click `Create Repository`.

9.6.10 Setup Jenkins

Install Jenkins

1. Once Jenkins container is started, go to `http://localhost:8080/`.
2. To unlock Jenkins, copy/paste the generated password that has been written in the container log. Click on `Continue`.
3. Select option `Install suggested plugins` and wait for plugins installation.
4. Fill in the `Create First Admin User` form. Click `Save and continue`.
5. Click on `Save and finish`, then on `Start using Jenkins`.

Configure Jenkins

1. Go to `Manage Jenkins` > `Plugins`.
2. **Add Docker Pipeline plugin:**
    1. Go to `Available plugins` section.
    2. Search *Docker Pipeline*.
    3. Install it and restart Jenkins

9.6.11 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

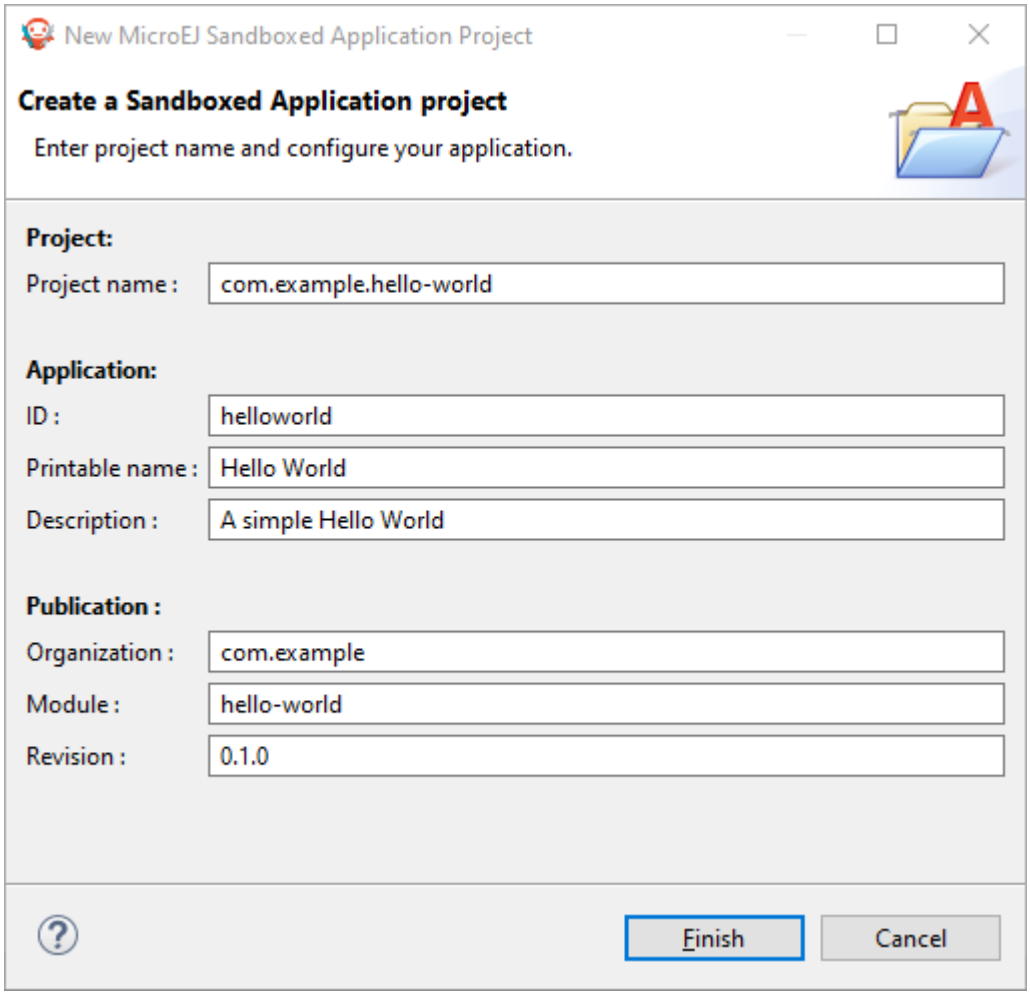
In this example, we will create a very simple module using the Sandbox Application buildtype (`build-application`) that we'll push to a Git repository.

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.

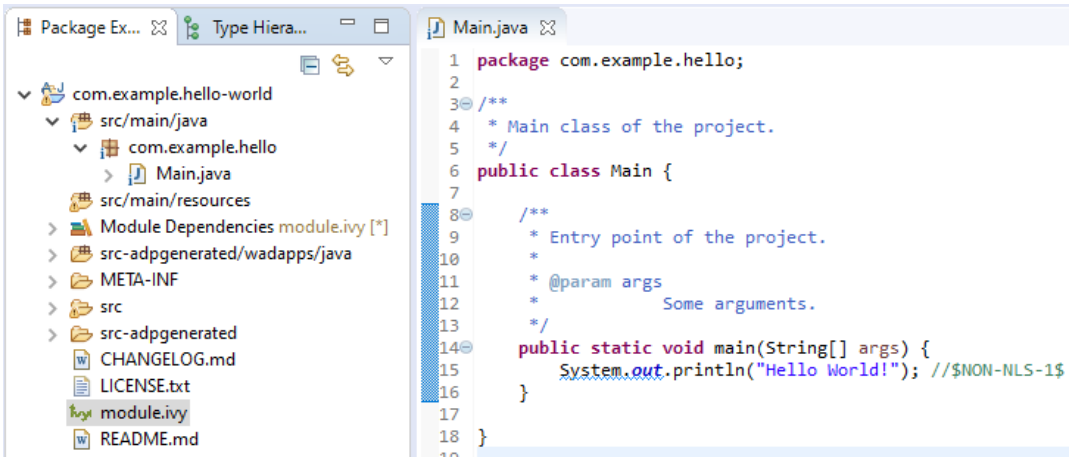
---

1. Start MicroEJ SDK.
2. Go to `File` > `New` > `Sandboxed Application Project`.

3. Fill in the template fields, set `Project name` to `com.example.hello-world`.

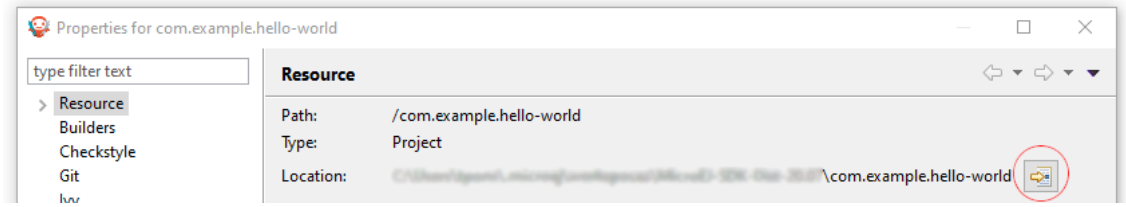


4. Click `Finish`. This will create the project files and structure.
5. Right-click on source folder `src/main/java` and select `New` > `Package`. Set a name to the package and click `Finish`.
6. Right-click on the new package and select `New` > `Class`. Set `Main` as name for the class and check `public static void main(String[] args)`, then click `Finish`.
7. Add the line `System.out.println("Hello World!");` to the method and save it.



8. **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.



Note: For more details about MicroEJ Applications development, refer to the *Application Developer Guide*.

Upload to your Git repository

Note: We need the IP address of the Docker Bridge Network, here we consider that it's `172.17.0.1` but you can check with the command `ip addr show docker0` on the Docker host.

1. Open the project directory, create a file named `Jenkinsfile` and copy this content inside:

```
pipeline {
    agent {
        docker {
            image 'microej/sdk:5.8.1'
            args '-e ACCEPT_MICROEJ_SDK_EULA_V3_1B=YES'
        }
    }
    stages {
        stage('Publish') {
            steps {
                sh 'mmm publish shared -r ivy/ivysettings-artifactory.xml'
```

(continues on next page)

(continued from previous page)

```
            }
        }
    }
}
```

2. Create a directory named `ivy`, create a file named `ivysettings-artifactory.xml` and copy this content inside:

```
<?xml version="1.0" encoding="UTF-8"?>
<ivy-settings>

        <property name="artifactory.repository.url" value="http://172.17.0.
↪1:8082/artifactory" override="false"/>
        <property name="local.repository.dir" value="${user.home}/.ivy2/
↪repository/" override="false"/>

        <!--
                Map MMM resolvers (*.resolver) to custom resolver

                Kinds of repositories:
                - release: used when publishing a released module.
                - snapshot: used when publishing a snapshot module.
                - local: used when publishing a snapshot module locally.
        -->
        <property name="release.resolver" value="modulesReleaseRepository"␣
↪override="false"/>
        <property name="shared.resolver" value="modulesSnapshotRepository"␣
↪override="false"/>
        <property name="local.resolver" value="localRepository" override="false
↪"/>

        <property name="modules.resolver" value="fetchAll" override="false" />
        <property name="request.cache.dir" value="${user.home}/.ivy2/cache"␣
↪override="false"/>
        <property name="default.conflict.manager" value="latest-compatible"␣
↪override="false"/>

        <settings defaultResolver="${modules.resolver}" defaultConflictManager="
↪${default.conflict.manager}" defaultResolveMode="dynamic"/>
        <caches defaultCacheDir="${request.cache.dir}"/>

          <resolvers>

                <url name="modulesReleaseRepository" m2compatible="true">
                        <artifact pattern="${artifactory.repository.url}/custom-
↪modules-release/[organization]/[module]/[branch]/[revision]/[artifact]-
↪[revision](-[classifier]).[ext]" />
                        <ivy pattern="${artifactory.repository.url}/custom-
↪modules-release/[organization]/[module]/[branch]/[revision]/ivy-[revision].xml
↪" />
                </url>
```

(continues on next page)

(continued from previous page)

```
                <url name="modulesSnapshotRepository" m2compatible="true"␣
↪checkmodified="true">
                        <artifact pattern="${artifactory.repository.url}/custom-
↪modules-snapshot/[organization]/[module]/[branch]/[revision]/[artifact]-
↪[revision](-[classifier]).[ext]" />
                        <ivy pattern="${artifactory.repository.url}/custom-
↪modules-snapshot/[organization]/[module]/[branch]/[revision]/ivy-[revision].
↪xml" />
                </url>

                <url name="microejModulesRepository" m2compatible="true">
                        <artifact pattern="${artifactory.repository.url}/
↪microej-module-repository/[organization]/[module]/[branch]/[revision]/
↪[artifact]-[revision](-[classifier]).[ext]" />
                        <ivy pattern="${artifactory.repository.url}/microej-
↪module-repository/[organization]/[module]/[branch]/[revision]/ivy-[revision].
↪xml" />
                </url>

                <filesystem name="localRepository" m2compatible="true"␣
↪checkmodified="true">
                        <artifact pattern="${local.repository.dir}/
↪[organization]/[module]/[branch]/[revision]/[artifact]-[revision](-
↪[classifier]).[ext]" />
                        <ivy pattern="${local.repository.dir}/[organization]/
↪[module]/[branch]/[revision]/ivy-[revision].xml" />
                </filesystem>

                <chain name="fetchRelease">
                        <resolver ref="modulesReleaseRepository"/>
                        <resolver ref="microejModulesRepository"/>
                </chain>

                <chain name="fetchSnapshot">
                        <resolver ref="modulesSnapshotRepository"/>
                        <resolver ref="fetchRelease"/>
                </chain>

                <chain name="fetchLocal">
                        <resolver ref="localRepository"/>
                        <resolver ref="fetchSnapshot"/>
                </chain>

                <chain name="fetchAll">
                        <resolver ref="fetchLocal"/>
                </chain>

        </resolvers>
</ivy-settings>
```

This file configures the MicroEJ Module Manager to import and publish modules from the Artifactory repositories described in this tutorial. Please refer to the *Settings File* section for more details.

Note: At this point, the content of the directory `com.example.hello-world` should look like the following:

```
com.example.hello-world
├── bin
│   └── ...
├── ivy
│   └── ivysettings-artifactory.xml
├── src
│   └── ...
├── src-adpgenerated/
│   └── ...
├── CHANGELOG.md
├── Jenkinsfile
├── LICENSE.txt
├── README.md
└── module.ivy
```

1. Open a terminal from the directory `com.example.hello-world` and type the following commands:

```
git init
git checkout -b main
git add *
git commit -m "Add Hello World application"
git remote add origin http://localhost:3000/<admin_user>/helloworld.git
git push -u origin main
```

Create a New Jenkins Job

Start by creating a new job for building our application.

1. Go to Jenkins dashboard.
2. Click on `New Item` .
3. Set item name to `Hello World` .
4. Select `Multibranch Pipeline` .
5. Validate with `Ok` button.
6. In `General` tab set `Display Name` to `Hello World`
7. In `Branch Sources` , click on `Add Source` > `Git` .
8. Add `Project Repository` http://172.17.0.1:3000/<admin_user>/helloworld.git



9. Click on `Save` .

Build the "Hello World" Application

Let's run the job!

In Jenkins `Hello World` dashboard, click on `main` branch, then click on `Build Now` .

Note: You can check the build progress by clicking on the build progress bar and showing the `Console Output` .

At the end of the build, the module is published to `http://localhost:8082/artifactory/list/custom-modules-snapshot/com/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 recommended step is to adapt MMM/Jenkins/Artifactory configuration to your ecosystem and development flow.

9.6.12 Appendix

This section discusses some of the customization options.

Customize Jenkins

Jenkins jobs are highly configurable, following options and values are recommended by MicroEJ, but they can be customized at your convenience.

In `General` tab:

1. Check `Discard old builds` and set `Max # of builds to keep` value to `15` .
2. Click on `Advanced` button, and check `Block build when upstream project is building` .

In `Build triggers` tab:

1. Check Poll SCM , and set a CRON-like value (for example `H/30 * * * *` to poll SCM for changes every 30 minutes).

In Post-build actions tab:

1. Add post-build action Publish JUnit test result report :
2. Set Test report XMLs to `**/target~/test/xml/**/test-report.xml, **/target~/test/xml/**/*Test.xml` .

Note: The error message `'**/target~/test/xml/**/test-report.xml' doesn't match anything: '**' exists but not '**/target~/test/xml/**/test-report.xml'` will be displayed since no build has been executed yet. These folders will be generated during the build.

---

3. Check Retain long standard output/error .
4. Check Do not fail the build on empty test results

Customize `target~` path

Some systems and toolchains don't handle long path properly. A workaround for this issue is to move the build directory (that is, the `target~` directory) closer to the root directory.

To change the `target~` directory path, set the *build option* `target` .

In Advanced , expand Properties text field and set the `target` property to the path of your choice. For example:

```
target=C:/tmp/
```

9.7 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.

9.7.1 Intended Audience

The audience for this document is engineers who are developing any kind of Java code (application or library).

9.7.2 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.
- Interfaces are named in the same way as classes (see below).
- Method and instance field names are written in lower camel case.
- Static field names are written in lower camel case.
- Constant names are written in fully upper case with underscore as word separator.
- Enum constant names are written in fully upper case with underscores as word separators.
- Local (and parameter) names are written in lower camel case.
- When a name contains an acronym, capitalize only the first letter of the acronym (e.g. for a local with the `HTTP` acronym, use `myHttpContext` instead of `myHTTPContext`).

It is also recommended to use full words instead of abbreviations (e.g. `MyProxyReference` instead of `MyProxyRef`).

Interfaces and Subclasses Naming Convention

An Interface is named after the feature it exposes. It does not have a `I` prefix because it hurts readability and may cause naming issues when potentially converted to/from an abstract class.

The classes implementing an interface are named after the interface and how they implement it. Using `Impl` suffix is not recommended because it does not express the implementation specificity. If there is no specificity, maybe there is no need to have an interface.

Example: an interface `Storage` (that allows to load/store data) may have several implementations, such as `StorageFs` (on a file system), `StorageDb` (on a database), `StorageRam` (volatile storage in RAM).

Visibility

Here is a list of the usage of each Java element visibility:

- `public`: API.
- `protected`: API for subclasses.
- `package`: component intern API (collaboration inside a package).
- `private`: internal structure, cache, lazy, etc.

By default, all instance fields must be private.

Package visibility can be used by writing the comment `/* package */` in place of the modifier.

Javadoc

Javadoc comments convention is based on the official documentation.

Note: Javadoc is written in HTML format and doesn't accept XHTML format: tags must not be closed. For example, use only a `<p>` between two paragraphs.

Here is a list of the rules to follow when writing Javadoc:

- All APIs (see *Visibility*) must have a full Javadoc (classes, methods, and fields).
- Add a dot at the end of all phrases.
- Add `@since` tag when introducing a new API.
- Do not hesitate to use links to help the user to navigate in the API (`@see` , `@link`).
- Use the `@code` tag in the following cases:
  - For keywords (e.g. `{@code null}` or `{@code true}`).
  - For names and types (e.g. `{@code x}` or `{@code Integer}`).
  - For code example (e.g. `{@code new Integer(Integer.parseInt(s))}`).

Here is a list of additional rules for methods:

- The first sentence starts with the third person (as if there is *This method* before).
- All parameters and returned values must be described.
- Put as much as possible information in the description, keep `@param` and `@return` minimal.
- Start `@param` with a lower case and usually with *the* or *a*.
- Start `@return` with a lower case as if the sentence starts with *Returns*.
- Avoid naming parameters anywhere other than in `@param` . If the parameter is renamed afterward, the comment is not changed automatically. Prefer using *the given xxx*.

Code Convention

Official documentation: https://www.oracle.com/java/technologies/javase/codeconventions-introduction.html

Class Declaration

The parts of a class or interface declaration must appear in the order suggested by the Code Convention for the Java Programming Language:

- Constants.
- Class (static) fields.
- Instance fields.
- Constructors
- Methods

Fields Order

For a better readability, the fields (class and instance) must be ordered by scope:

1. `public`
2. `protected`
3. `package`
4. `private`

Methods Order

It is recommended to group related methods together. It helps for maintenance:

- when searching for a bug on a specific feature,
- when refactoring a class into several ones.

Modifiers Order

Class and member modifiers, when present, appear in the order recommended by the Java Language Specification:

```
public protected private abstract default static final transient volatile synchronized native
strictfp
```

Code Style and Formatting

MicroEJ defines a formatting profile for `.java` files, which is automatically set up when creating a new *Module Project Skeleton*.

Note: MicroEJ SDK automatically applies formatting when a `.java` file is saved. It is also possible to manually apply formatting on specific files:

- In `Package Explorer`, select the desired files, folders or projects,
- then go to Source > Format. The processed files must not have any warning or error.

Here is the list of formatting rules included in this profile:

- Indentation is done with 1 tab.
- Braces are mandatory with `if`, `else`, `for`, `do`, and `while` statements, even when the body is empty or contains only a single statement.
- Braces follow the Kernighan and Ritchie style (Egyptian brackets) described below:
  - No line break before the opening brace.
  - Line break after the opening brace.
  - Line break before the closing brace.
  - Line break after the closing brace, only if that brace terminates a statement or terminates the body of a method, constructor, or named class. For example, there is no line break after the brace if it is followed by else or a comma.

- One statement per line.
- Let the formatter automatically wraps your code when a statement needs to be wrapped.

Here is a list of additional formatting rules that are not automatically applied:

- Avoid committing commented code (other than to explain an optimization).
- All methods of an interface are public. There is no need to specify the visibility (easier to read).

Note: Most of these rules are checked by *Code Analysis with SonarQube™*.

9.7.3 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() {
  int result = PRIME;
  result = PRIME * result + field1;
  result = PRIME * result + (field2 == null ? 0 : field2.hashCode());
  return result;
}
```

9.7.4 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 *Test Suite 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.

9.8 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.

9.8.1 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.

9.8.2 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.

9.8.3 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 *SOAR 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.

9.8.4 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.

Images

Encoding

If you are tight on ROM but have enough RAM and CPU power to decode PNG images on the fly, consider storing your images as PNG resources. If you are in the opposite configuration (lots of ROM, but little RAM and CPU power), consider storing your images in raw format.

See *Image Generator* documentation for more information on how to achieve that.

Color Depth (BPP)

Make sure to use images with a color depth not exceeding the one of your display to avoid the following issues:

- Waste of memory.
- Differences between the rendering on the target device and the original image resource.

External Storage

To save storage on internal flash, the application can access the images from an external storage device.

See *External Resources Loader* documentation for more information on how to achieve that.

Application Code

The following application code guidelines are recommended in order to minimize the size of the application:

- Check libraries versions and changelogs regularly. Latest versions may be more optimized.
- Avoid manipulating String objects:
  - For example, prefer using integers to represent IDs.
  - Avoid overriding Object.toString() for debugging purposes. This method will always be embedded even if it is not called explicitly.

  - Avoid using the *logging library* or `System.out.println()`, use the *trace library* or the *message library* instead. The logging library uses strings, while the trace and message libraries use integer-based error codes.
  - Avoid using the string concatenation operator (`+`), use an explicit StringBuilder instead. The code generated by the `+` operator is not optimal and is bigger than when using manual `StringBuilder` operations.
- Avoid manipulating wrappers such as Integer and Long objects, use primitive types instead. Such objects have to be allocated in Java heap memory and require additional code for boxing and unboxing.
- Avoid declaring Java Enumerations (`enum`), declare compile-time constants of primitives types instead (e.g. `static final int I = 0;`). The Java compiler creates an Enum object in the Java heap for each enumeration item, as well as complex class initialization code.
- Avoid using the service library, use singletons or Constants.getClass() instead. The service library requires embedding class reflection methods and the type names of both interfaces and implementations.
- Avoid using the Java Collections Framework. This OpenJDK standard library has not been designed for memory constrained devices.
  - Use raw arrays instead of List objects. The ArrayTools class provides utility methods for common array operations.
  - Use PackedMap objects instead of Map objects. It provides similar APIs and features with lower 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.
- Avoid accessing a private field of a nested class. The Java compiler will generate a dedicated method instead of a direct field access. This method is called *synthetic*, and is identified by its name prefix: `access$`.

- Replace constant arrays and objects initialization in `static final` fields by *immutables objects*. Indeed, initializing objects dynamically generates code which takes significant ROM and requires execution time.
- Check if some features available in software libraries are not already provided by the device hardware. For example, avoid using java.util.Calendar (full Gregorian calendar implementation) if the application only requires basic date manipulation provided by the internal real-time clock (RTC).

MicroEJ Platform Configuration

The following configuration guidelines are recommended in order to minimize the size of the application:

- Check MicroEJ Architecture and Packs versions and changelogs regularly. Latest versions may be more optimized.
- Configure the Platform to use the *tiny* capability of the 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 with SDK 5 or lower* 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 with SDK 5 or lower* section.

Listing Required Class Names

Some class names may be required by an application to work properly. These class names must be explicitly specified in a `*.types.list` file.

The code of the application must be checked for all uses of the Class.forName(), Class.getName() and Class.getSimpleName() methods. For each of these method calls, if the class name if absolutely required and can not be known at compile-time, add it to a `*.types.list` file. Otherwise, remove the use of the class name.

The following sections illustrates this on concrete use cases.

Case of Service Library

The ej.service.ServiceLoader class of the service library is a dependency injection facility. It can be used to dynamically retrieve the implementation of a service.

The assignment between a service API and its implementation is done in `*.properties.list` files. Both the service class name and the implementation class name must be embedded (i.e., added in a `*.types.list` file).

For example:

```
# example.properties.list
com.example.MyService=com.example.MyServiceImpl
```

```
# example.types.list
com.example.MyService
com.example.MyServiceImpl
```

Case of Properties Loading

Some properties may be loaded by using the name of a class to determine the full name of the property. For example:

```
Integer.getInteger(MyClass.class.getName() + ".myproperty");
```

In this case, it can be replaced with the actual string. For example:

```
Integer.getInteger("com.example.MyClass.myproperty");
```

Case of Logger and Other Debugging Facilities

Logging mechanisms usually display the name of the classes in traces. It is not necessary to embed these class names. The *Stack Trace Reader* can decipher the output.

9.8.5 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 (ej.bon.Timer, Executor, MicroUI.callSerially(Runnable), ...).
- When designing Graphical User Interface:
  - Avoid creating mutable images (BufferedImage instances) to draw in them and render them later, render graphics directly on the display instead. Mutable images require allocating a lot of memory from the images heap.
  - Make sure that your Widget hierarchy is as flat as possible (avoid any unnecessary Container). Deep widget hierarchies take more memory and can reduce performance.

MicroEJ Platform Configuration

The following configuration guidelines are recommended in order to minimize the runtime size of the application:

- Check the size of the stack of each RTOS task. For example, 1.0KB may be enough for the 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 with SDK 5 or lower* 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 it by using the *Heap Usage Monitoring Tool*.

Thread Stacks

- Configure the *maximum number of threads* option. This number can be known accurately by counting in the code how many `Thread` and `Timer` objects may run concurrently. You can call Thread.getAllStackTraces() or Thread.activeCount() to know what threads are running at a given moment.
- Configure the *number of allocated thread stack blocks* option. This can be done empirically by starting with a low number of blocks and increasing this number as long as the application throws a `StackOverflowError`.
- Configure the *maximum number of blocks per thread* option. The best choice is to set it to the number of blocks required by the most greedy thread. Another acceptable option is to set it to the same value as the total number of allocated blocks.
- Configure the *maximum number of monitors per thread* option. This number can be known accurately by counting the number of concurrent `synchronized` blocks. This can also be done empirically by starting with a low number of monitors and increasing this number as long as no exception occurs. Either way, it is recommended to set a slightly higher value than calculated.

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 *Dump the States of the Core Engine* section for usage.

More specifically, the `Peak java threads count` value printed in the dump can be used to configure the maximum number of threads. The `max_java_stack` and `current_java_stack` values printed for each thread can be used to configure the number of stack blocks.

MicroUI Images Heap

- Configure the *images heap* to be as small as possible. You can compute the optimal size empirically. It can also be calculated accurately by adding the size of every image that may be stored in the images heap at a given moment. One way of doing this is to inspect every occurrence of BufferedImage() allocations and ResourceImage usage of `loadImage()` methods.

9.9 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.

9.9.1 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.

9.9.2 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 Module

The XML Module must be added to the *module.ivy* of the MicroEJ Application project to use the KXML library.

```
<dependency org="org.kxml2" name="kxml2" rev="2.3.2"/>
```

Example Of Use

An example is available at https://github.com/MicroEJ/Example-XML. It presents how to use XML data exchange for your 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"
```

(continues on next page)

(continued from previous page)

```
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
```

9.9.3 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 Module

The JSON Module must be added to the *module.ivy* of the MicroEJ Application project to use 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 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:

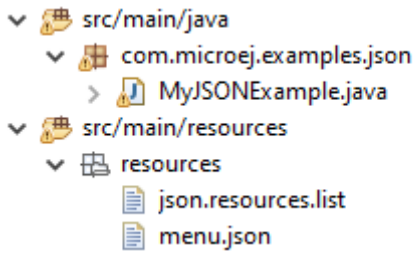
```
{
        "menu": {
                "id": "file",
                "value": "File",
                "popup": {
                        "menuitem": [
                                {"value": "New", "onclick": "CreateNewDoc()"},
                                {"value": "Open", "onclick": "OpenDoc()"},
                                {"value": "Close", "onclick": "CloseDoc()"}
                        ]
```

(continues on next page)

(continued from previous page)

```
                }
        }
}
```

First, we need to include this file in our project by adding it to the `src/main/resources` folder and creating a `.resources.list` properties file to declare this resource for our application to be able to retrieve it (see *Resources* for more details).



This `.resources.list` file (here named `json.resources.list`) should contain the path to our JSON file as such :

```
resources/menu.json
```

The example below will parse the file, browse the resulting data structure (`org.json.me.JSONObject`) and print the value of the `menuitem` JSON array.

```
package com.microej.examples.json;

import java.io.DataInputStream;
import java.io.IOException;

import org.json.me.JSONArray;
import org.json.me.JSONException;
import org.json.me.JSONObject;

/**
* This example uses the org.json.me parser provided by json.org to parse and
* browse a JSON content.
*
* The JSON content is simple abstraction of a file menu as provided here:
* http://www.json.org/example.html
*
* The example then tries to list all the 'menuitem's available in the popup
* menu. It is assumed the user knows the menu JSON file structure.
*
*/
public class MyJSONExample {

        public static void main(String[] args) {

                // get back an input stream from the resource that represents the JSON
                // content
                DataInputStream dis = new DataInputStream(
```

(continues on next page)

(continued from previous page)

```
                                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 ] ===============
=============== [ Launching Simulator ] ===============
The menuitem content of popup menu is:
[{"value":"New","onclick":"CreateNewDoc()"},{"value":"Open","onclick":"OpenDoc()"},{"value":
↪"Close","onclick":"CloseDoc()"}]
=============== [ Completed Successfully ] ===============
```

(continues on next page)

(continued from previous page)

```
SUCCESS
```

9.9.4 CBOR

The CBOR (Concise Binary Object Representation) binary data serialization format is a lightweight data-interchange format similar to JSON but with a smaller footprint, making it very practical for embedded applications, though its messages are often less easily readable by humans.

CBOR Module

The CBOR Module must be added to the *module.ivy* of the MicroEJ Application project to use the CBOR library.

```
<dependency org="ej.library.iot" name="cbor" rev="1.1.0"/>
```

Example Of Use

An example is available at https://github.com/MicroEJ/Example-Sandboxed-IOT/tree/master/cbor. It shows how to use the CBOR library in your MicroEJ Application by encoding some data and reading it back, printing it on the standard output both as a raw byte string and in a JSON-like format. You can use tools like cbor.me to convert the byte string output to a JSON format and check that it matches the encoded data. The project can run on any MicroEJ Platform (no external dependencies).

The execution of this example on the MicroEJ Simulator should print the following trace:

```
=============== [ Initialization Stage ] ===============
=============== [ Launching on Simulator ] ===============
CBOR data string :␣
↪a1646d656e75a36269646466696c656576616c75656446696c6565706f707570a1686d656e756974656d83a26576616c7565634e65
Data content :
{
        "menu" : {
                "id" : "file",
                "value" : "File",
                "popup" : {
                        "menuitem" : [ {
                                        "value" : "New",
                                        "onclick" : "CreateNewDoc()"
                                }, {
                                        "value" : "Open",
                                        "onclick" : "OpenDoc()"
                                }, {
                                        "value" : "Close",
                                        "onclick" : "CloseDoc()"
                                } ]
                }
        }
}
=============== [ Completed Successfully ] ===============
```

Another example showing how to use the *JSON* module along with the *CBOR* module to convert data from JSON to CBOR is available here : https://github.com/MicroEJ/Example-Sandboxed-IOT/tree/master/cbor.json.

The execution of this example on the MicroEJ Simulator should print the following trace:

```
Initial data (271 bytes) = {"menu":{"value":"File","id":"file","popup":{"menuitem":[{"value":
↪"New","onclick":"CreateNewDoc()"},{"value":"Open","onclick":"OpenDoc()"},{"value":"Close",
↪"onclick":"CloseDoc()"}]}}}
Data serialized (139 bytes)
Data deserialized = {menu={value=File, id=file, popup={menuitem=[{value=New,␣
↪onclick=CreateNewDoc()}, {value=Open, onclick=OpenDoc()}, {value=Close, onclick=CloseDoc()}
↪]}}}
```

9.10 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.

9.10.1 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.

9.10.2 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

9.10.3 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
```

(continues on next page)

(continued from previous page)

```
   }

   private static ApplicationState currentState;
   private static ApplicationState previousState;

   public static void main(String[] args) {
      currentState = ApplicationState.UNINSTALLED;
      switchState(ApplicationState.INSTALLED);
   }

   public static void switchState(ApplicationState newState) {
      previousState = currentState;
      currentState = newState;
   }
}
```

Finally, the last section describes some techniques to remove logging related code in order to reduce the memory footprint.

9.10.4 Log with the Trace Library

The `ej.api.trace` Trace library provides a way of tracing integer events. Its features and principles are described in the *Event Tracing* section.

Here is a short example of how to use this library to log the entry/exit of the `switchState()` method:

1. 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);
   ```

(continues on next page)

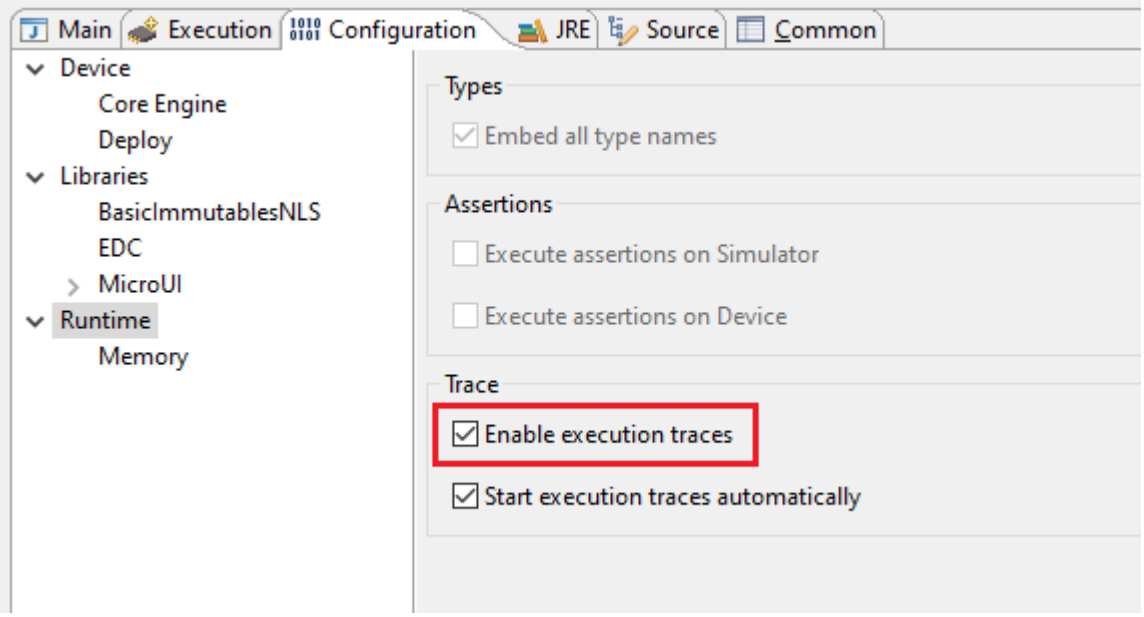
(continued from previous page)

```
   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.

9.10.5 Log with the Message Library

The `ej.library.runtime.message` Message library was designed to enable logging while minimizing RAM/ROM footprint and CPU usage. For that, logs are based on message identifiers, which are stored on integers instead of using of constant Strings. In addition to a message identifier, the category of the message allows the user to find the corresponding error/warning/info description. An external documentation must be maintained to describe all message identifiers and their expected arguments for each category.

Principles:

- The MessageLogger type allows for logging messages solely based on integers that identify the message content.

- Log a message by using methods MessageLogger.log(...) methods, by specifying the log level, the message category, and the message identifier. Use optional arguments to add any useful information to the log, such as a Throwable or contextual data.
- Log levels are very similar to those of the `Logging` library. The class ej.util.message.Level lists the available levels.
- Loggers rely on the MessageBuilder type for message creation. The messages built by the BasicMessageBuilder follow this pattern: `[category]:[LEVEL]=[id]`. The builder appends the specified Object arguments (if any) separated by spaces, then the full stack trace of the Throwable argument (if any).
- The FilterMessageLogger allows to filter messages actually logged based on a threshold level (defaults to `INFO`). The threshold level can be modified dynamically with FilterMessageLogger.setLevel(). Use the system `FilterMessageLogger.INSTANCE` or create a new logger to configure the level of logged messages per instance.

Here is a short example of how to use this library to log the entry/exit of the `switchState()` method:

1. To use this library, add 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;

   FilterMessageLogger.INSTANCE.log(Level.INFO, LOG_CATEGORY, LOG_ID, previousState,␣
↪currentState);
}
```

This produces the following output:

```
Application:I=2 UNINSTALLED INSTALLED
```

9.10.6 Log with the Logging Library

The `ej.library.eclasspath.logging` Logging library implements a subset of the standard Java java.util.logging package and follows the same principles:

- There is one instance of LogManager by application that manages the hierarchy of loggers.
- Find or create Logger objects using the method Logger.getLogger(String). If a logger has already been created with the same name, this logger is returned, otherwise a new logger is created.
- Each Logger created with this method is registered in the `LogManager` and can be retrieved using its String `ID`.
- A minimum level can be set to a Logger so that only messages that have at least this level are logged. The class java.util.logging.Level lists the available standard levels.
- The Logger API provides multiple methods for logging:

  - log(...) methods that send a LogRecord to the registered Handler instances. The LogRecord object wraps the String message and the log level.
  - Log level-specific methods, like severe(String msg), that call the aforementioned `log(...)` method with the correct level.
- The library defines a default Handler implementation, called DefaultHandler, that prints the message of the LogRecord on the standard error output stream. It also prints the stack trace of the Throwable associated with the LogRecord if there is one.

Here is a short example of how to use this library to log the entry/exit of the `switchState()` method:

1. 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.

9.10.7 Remove Logging Related Code

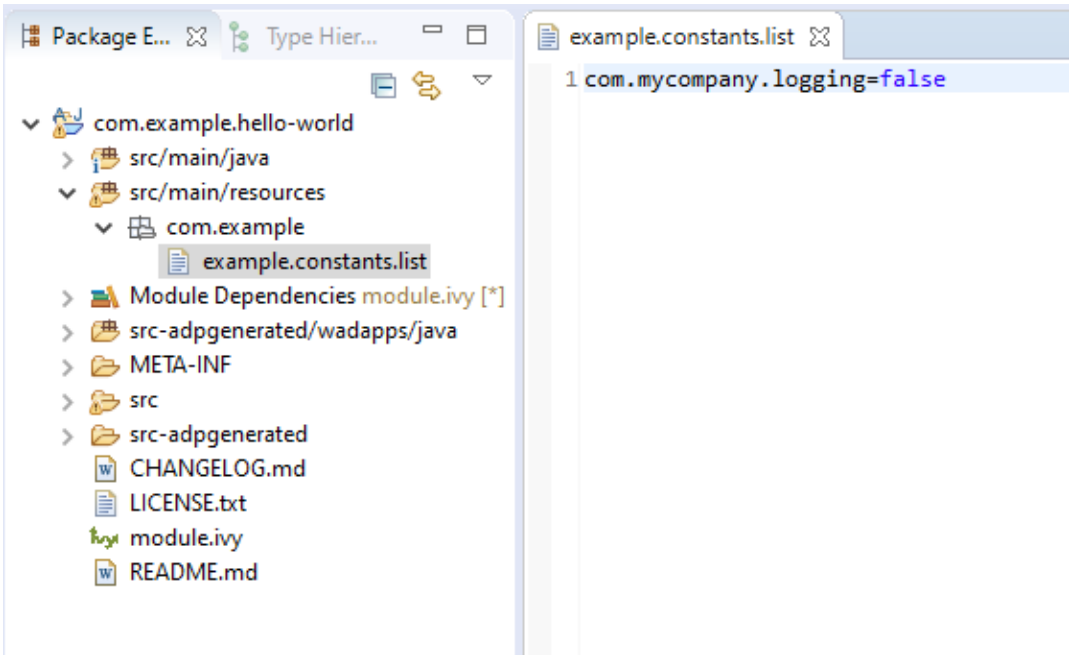
This section describes some techniques to remove logging related code, which saves memory footprint when logging is disabled at runtime. This is typically useful when building two Firmware flavors: one for production and one for debug.

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 (`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);
```

(continues on next page)

(continued from previous page)

```
    }
}
```

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.

9.11 Run a Test Suite on a Device

This tutorial describes all the steps to configure and run a *VEE Port Test Suite* on a device using the Platform Qualification Tools.

In this tutorial, the target device is the Espressif ESP32-WROVER-KIT V4.1 board and the Filesystem Test Suite for *FS* module will be used as an example.

The tutorial should take 1 hour to complete (excluding the Platform Getting Started setup).

9.11.1 Intended Audience and Scope

The audience for this document is software engineers who want to validate an Abstraction Layer implementation or understand how to automatically run a MicroEJ Test Suite on their device.

The following topics are out of the scope of this tutorial:

- How to write test cases and package a Test Suite module. See *Test Suite with JUnit* for this topic.
- How to create a new Foundation Library. See the Foundation Library Getting Started to learn more about creating custom Foundation Library.

9.11.2 Prerequisites

This tutorial assumes the following:

- Good knowledge of the *MicroEJ Glossary*.
- Tutorial *Understand How to Build a Firmware and its Dependencies* has been followed.
- MicroEJ SDK distribution 20.07 or more (see *SDK Version*).
- The WROVER Platform has been properly setup (i.e., it can be used to generate a Mono-Sandbox Executable).

The explanation can be adapted to run the test suite on any other MicroEJ Platform providing:

- An implementation of *LLFS: File System* version 1.0.2 in com.microej.pack#fs-4.0.3.
- A partial or full *BSP Connection*.

Note: This tutorial can also be adapted to run other test suites in addition to the Filesystem Test Suite presented here.

9.11.3 Introduction

This tutorial presents a local setup of the *VEE Port Test Suite* for the *FS* Foundation Library on a concrete device (not on Simulator).

In essence, a Foundation Library provides an API to be used by an Application or an Add-On Library.

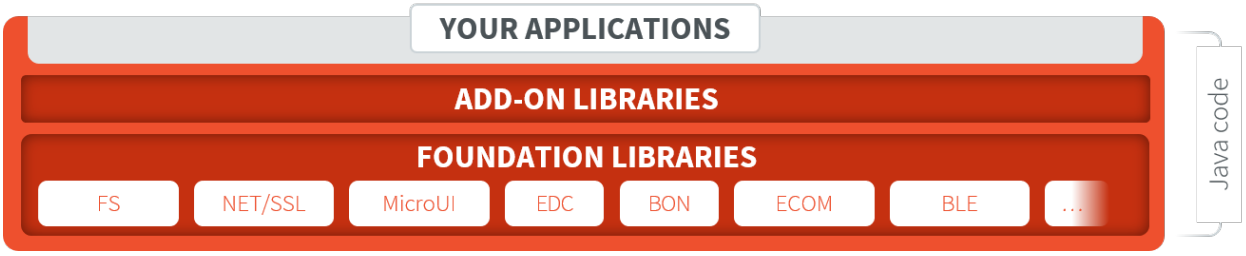


Fig. 1: MicroEJ Foundation Libraries, Add-On Libraries and MicroEJ Application

For example, the Java file system API java.io.File is provided by the MicroEJ Foundation Library named FS. The Abstraction Layer of each Foundation API must be implemented in C in the Board Support Package. The Test Suite is used to validate the C code implementation of the Abstraction Layer.

9.11.4 Import the Test Suite

Follow these steps to import the Filesystem Test Suite into the workspace from the Platform Qualification Tools:

- Clone or download the Platform Qualitification Tools project 2.3.0.
- Select File > Import... .
- Select Existing Projects into Workspace .
- Set Select the root directory to the directory `tests/fs` in the Platform Qualification Tools fetched in the previous step.
- Ensure Copy projects into workspace is checked.
- Click on Finish .

The project `java-testsuite-fs` should now be available in the workspace.

9.11.5 Configure the Test Suite

Select the Test Suite Version

For a given Foundation Library version, a specific Test Suite version should be used to validate the Abstraction Layer implementation. Please refer to *Test Suite Versioning* to determine the correct Test Suite version to use.

On the WROVER Platform, the FS Test Suite version to use is specified in `{PLATFORM}-configuration/testsuites/fs/README.md`. The Test Suite version must be set in the `module.ivy` of the `java-testsuite-fs` project (e.g. `java-testsuite-fs/module.ivy`). For example:

```
<dependency org="com.microej.pack.fs" name="fs-testsuite" rev="3.0.3"/>
```

Configure the Platform BSP Connection

Several properties must be defined depending on the type of BSP Connection used by the MicroEJ Platform.

For a MicroEJ Application, these properties are set using the launcher of the application. For a Test Suite, the properties are defined in a file named `config.properties` in the root folder of the Test Suite. For example, see this example of config.properties file.

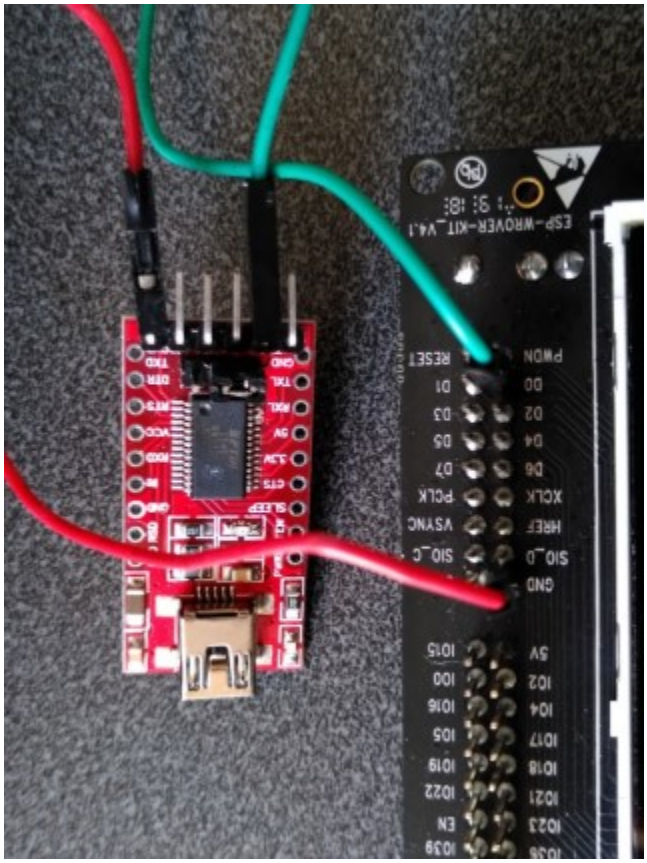
See *BSP Connection* for an explanation of the properties. See the comments in the file for a details description of each properties. The `microej.testsuite.properties.deploy.*` and `target.platform.dir` properties are required.

Configure Execution Trace Redirection

When the Test Suite is executed, the Test Suite Engine must read the trace to determine the result of the execution. To do that, we will use the *Serial to Socket Transmitter* tool to redirect the execution traces dumped to a COM port.

The WROVER platform used in this tutorial is particular because the UART port is already used to flash the device. Thus, a separate UART port must be used for the trace output.

This platform defines the option `microej.testsuite.properties.debug.traces.uart` to redirect traces from standard input to UART.



See the Testsuite Configuration section of the WROVER Platform documentation for more details.

Start Serial To Socket

The *Serial to Socket Transmitter* tool can be configured to listen on a particular COM port and redirect the output on a local socket. The properties `microej.testsuite.properties.testsuite.trace.ip` and `microej.testsuite.properties.testsuite.trace.port` must be configured.

Follow these steps to create a launcher for Serial To Socket Transmitter:

- Select `Run` > `Run Configurations...` .
- Right-click on `MicroEJ Tool` > `New` .
- In the `Execution` tab:
  - Set `Name` to `Serial To Socket Transmitter` .
  - Select a MicroEJ Platform available in the workspace in `Target` > `Platform` .
  - Select `Serial To Socket Transmitter` in `Execution` > `Settings` .
  - Set the `Output folder` to the workspace.
- In the `Configuration` tab:
  - Set the correct COM port and baudrate for the device in `Serial Options` .

- Set a valid port number in Server Options > Port . This port is the same as the one set in `config.properties` as `microej.testsuite.properties.testsuite.trace.port` .

Configure the Test Suite Specific Options

Depending on the Test Suite and the specificities of the device, various properties may be required and adjusted. See the file `validation/microej-testsuite-common.properties` (for example https://github.com/MicroEJ/VEEPortQualificationTools/blob/2.3.0/tests/fs/java/java-testsuite-fs/validation/microej-testsuite-common.properties) and the README of the Test Suite for a description of each property.

On the WROVER Platform, the configuration files `config.properties` and `microej-testsuite-common.properties` are provided in `{PLATFORM}-configuration/testsuites/fs/` .

In `config.properties` , the property `target.platform.dir` must be set to the absolute path to the platform. For example `C:/P0065_ESP32-WROVER-Platform/ESP32-WROVER-Xtensa-FreeRTOS-platform/source` .

9.11.6 Run the Test Suite

To run the Test Suite, right click on the Test Suite module and select `Build Module` .

9.11.7 Configure the Tests to Run

It is possible to exclude some tests from being executed by the Test Suite Engine.

To speed-up the execution, let's configure it to run only a small set of tests. In the following example, only the classes that match `TestFilePermission` are executed. This configuration goes into the file `config.properties` in the folder of the test suite.

```
# Comma separated list of patterns of files that must be included
# test.run.includes.pattern=**/Test*.class
test.run.includes.pattern=**/TestFilePermission*.class
# Comma separated list of patterns of files that must be excluded (defaults to inner classes)
test.run.excludes.pattern=**/*$*.class
```

Several reasons might explain why to exclude some tests:

- **Iterative development**. Test only the Abstraction Layer that is currently being developed. The full Test Suite must still be executed to validate the complete implementation.
- **Known bugs in the Foundation Library**. The latest version of the Test Suite for a given Foundation Library might contain regression tests or tests for new features. If the MicroEJ Platform doesn't use the latest Foundation Library, then it can be necessary to exclude the new tests.
- **Known bugs in the Foundation Library implementation**. The project might have specific requirements that prevent a fully compliant implementation of the Foundation Library.

9.11.8 Examine the Test Suite Report

Once the Test Suite is completed, open the HTML *Test Suite Report* stored in `java-testsuite-fs/target~/test/html/test/junit-noframes.html` .

At the beginning of the file, a summary is displayed. Below, all execution traces for each test executed are available.

If necessary, the binaries produced and ran on the device by the Test Suite Engine are available in `target~/test/xml/<TIMESTAMP>/bin/<FULLY-QUALIFIED-CLASSNAME>/application.out` .

The following image shows the test suite report fully passed:

Testsuite Results:

Summary

Tests	Failures	Errors	Ignored	Tried Again	Success rate	Time
41	0	0	0	0	100.00%	2683.271

Assertions	Failures	Success	Success Rate
3310	0	3310	100.00%

Note: *failures* are anticipated and checked for with assertions while *errors* are unanticipated.

Note: *ignored* tests are executed but not counted on the success rate.

Note: *tried again* tests are executed but not counted on the success rate.

Packages

Note: package statistics are not computed recursively, they only sum up all of its testsuites numbers.

Name	Tests	Errors	Failures	Ignored	Tried Again	Time(s)	Time Stamp	Host
com.microej.fs.tests	2	0	0	0	0	132.951	1599225360034	local
com.microej.fs.tests.constructors	4	0	0	0	0	254.649	1599225493041	local
com.microej.fs.tests.fields	2	0	0	0	0	129.236	1599225747722	local
com.microej.fs.tests.integration	1	0	0	0	0	62.722	1599225876968	local
com.microej.fs.tests.methods	22	0	0	0	0	1406.581	1599225939694	local
com.microej.fs.tests.properties	1	0	0	0	0	63.021	1599227346480	local
com.microej.fs.tests.scenarios	9	0	0	0	0	634.111	1599227409508	local

9.12 Implement a Blocking Java Native Method with SNI

This tutorial describes the good practices to follow when implementing a blocking native method in C. A native method is a method declared in Java with the `native` keyword and implemented in C using the *Simple Native Interface (SNI)*.

9.12.1 Intended Audience

The audience for this document is Platform developers who want to implement Abstraction Layer interfaces.

9.12.2 Prerequisites

The following document assumes the reader already has a setup ready to run a MicroEJ Standalone Application on a target device.

The following document also assumes the reader is familiar with the *Simple Native Interface (SNI)* mechanism. If not, the CallingCFromJava GitHub example shows the minimum steps required to create a Java program that makes a call to a C function via SNI.

9.12.3 Overview

The MicroEJ Core Engine implements a green thread architecture with all the Java threads executed within one single RTOS/OS task. Thus, it embeds its own scheduler that controls the execution of the Java threads. With such an architecture, the MicroEJ Core Engine cannot preempt a Java thread that executes a native method. Therefore a blocking native method will prevent the execution of other Java threads. To mitigate the contention, a native method must explicitly yield its current use of the processor.

This tutorial will explain how to use SNI to implement a blocking Java native method without blocking the execution of other Java threads.

9.12.4 Requirements

A MicroEJ Platform with (at least):

- EDC-1.3.
- SNI-1.4.

9.12.5 Example Code

Let's start with a MicroEJ Standalone Application that calls a blocking Java method implemented in C.

The following example waits for a button event and prints the index of the pressed button.

The MicroEJ Application code:

```
package example;

public class NativeCCallExample {

    public static void main(String[] args) {

        while (true) {
            System.out.println("Waiting for a button event...");
            int buttonIndex = waitButton();
            System.out.println("Button pressed: " + buttonIndex);
        }
    }

    public static native int waitButton();
}
```

The C implementation of the `waitButton()` native has been written in pseudo-code. It should be adapted according to the BSP of the target board.

```
#include "semaphore.h"
#include "sni.h"

static int pressed_button_index;
static Semaphore button_semaphore;

void button_init(){
    button_semaphore = SemaphoreCreateBinary();
```

(continues on next page)

(continued from previous page)

```
}

jint Java_example_NativeCCallExample_waitButton(){
    // Wait for a button event
    semaphoreTake(button_semaphore);
    return pressed_button_index;
}

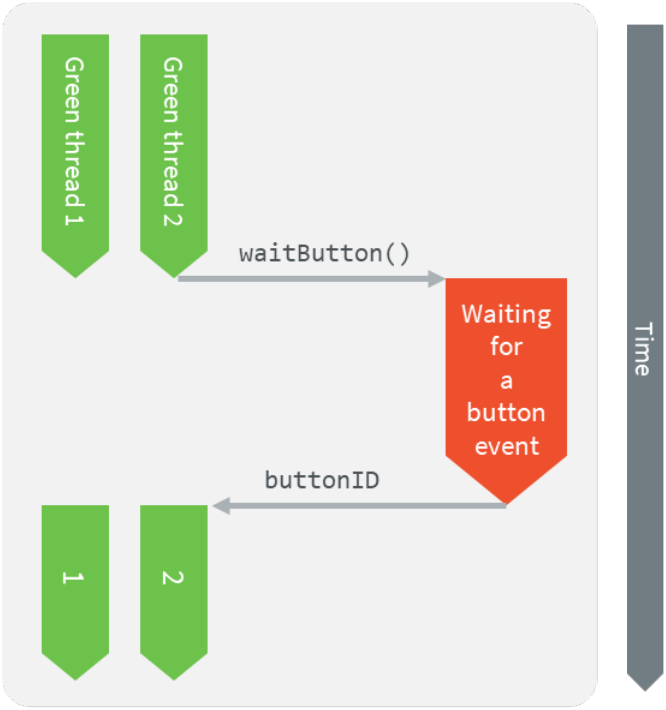
/** Interrupt request handler called when a button is pressed. */
int buttonIRQ(int button_index){
    pressed_button_index = button_index;
    semaphoreGiveFromISR(button_semaphore);
}
```

Application Behavior

In this example, the execution of the `waitButton()` native method will block until a button is pressed. In other words, while `Java_example_NativeCCallExample_waitButton()` has not returned, no other Java thread can be scheduled.

This is because the native function is called in the same RTOS/OS task as the Java application.

This schematic explains what is going on:



The MicroEJ VEE RTOS task

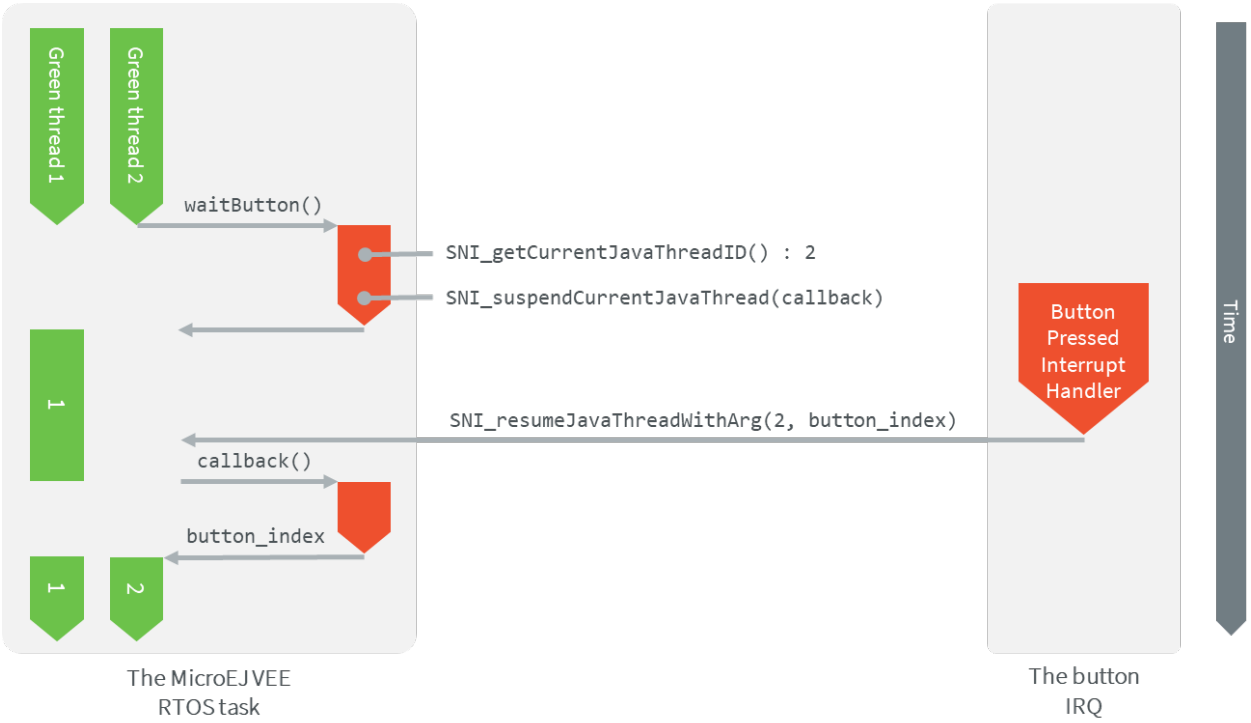
9.12.6 Implement a Non-Blocking Method

This section will explain how to update the example code to make a non-blocking method.

Here is a summary of what will be done in C:

- Signal the MicroEJ Core Engine to suspend the current thread when the native function returns.
- Remove the blocking operations from the native function so that it returns immediately.
- Implement a callback function that returns the index of the pressed button.
- Register this callback function in the MicroEJ Core Engine to call it when the Java thread is resumed.
- Resume the Java thread when a button is pressed.

This schematic summarizes the steps described above:



Update the C Native Function Implementation

Step 1: Update the C Native Function

The `Java_example_NativeCCallExample_waitButton()` function will now suspend the current Java thread. It will also store the information required to resume it and return the index of the pressed button.

The SNI functions used in this example are defined in `sni.h`. See this header file for a more detailed description of the API.

- Store the ID of the Java thread that called the function. This ID should be stored in a global variable. It is used to resume the Java thread when a button is pressed.

```
java_thread_id = SNI_getCurrentJavaThreadID();
```

- Signal the MicroEJ Core Engine to suspend the current Java thread and specify the callback function to be called when the thread is resumed. Let's call the callback function `waitButton_callback()`.

```
SNI_suspendCurrentJavaThreadWithCallback(0, (SNI_callback)waitButton_callback, NULL);
```

The function `SNI_suspendCurrentJavaThreadWithCallback()` returns immediately. The current thread is actually suspended when the native function returns.

The value returned by the `Java_example_NativeCCallExample_waitButton()` doesn't matter anymore. The callback function will be in charge of returning the value.

The updated `Java_example_NativeCCallExample_waitButton()` function should look like this:

```
static int32_t java_thread_id;

jint Java_example_NativeCCallExample_waitButton(){

    java_thread_id = SNI_getCurrentJavaThreadID();

    SNI_suspendCurrentJavaThreadWithCallback(0, (SNI_callback)waitButton_callback, NULL);

    return SNI_IGNORED_RETURNED_VALUE; // Returned value not used
}
```

Step 2: Update the Button Interrupt Function

The role of the button interrupt is now to resume the Java thread when a button event occurs. Update it this way:

```
int buttonIRQ(int button_index){
    SNI_resumeJavaThreadWithArg(java_thread_id, (void*)button_index);
}
```

The button's index is passed to the function `SNI_resumeJavaThreadWithArg()` so that the callback retrieves it when the thread is resumed.

Step 3: Implement the Callback Function

The callback function must have the same signature as the SNI native (same parameters and return type): `jint waitButton_callback()`.

The callback function is automatically called by the Java thread when it is resumed. Use the `SNI_getCallbackArgs()` function to retrieve the arguments that was previously given to the `SNI_suspendCurrentJavaThreadWithCallback()` or `SNI_resumeJavaThreadWithArg()` functions.

```
jint waitButton_callback()
{
    int button_index;
    SNI_getCallbackArgs(NULL, (void*)&button_index);
    return (jint)button_index; // Actual value returned to Java
}
```

Application Behavior

In this configuration, calling the native method `waitButton()` will still return only when a button is pressed, but it will not prevent other Java threads from being scheduled.

9.13 Get Started With GUI

9.13.1 Setup your Environment

Prerequisites

The *MICROEJ SDK* must be installed. Please check the *MICROEJ SDK requirements*.

Download and Install

1. Download the installer package corresponding to your host computer OS: Download MicroEJ SDK.
2. Unzip the downloaded installer package if needed and execute the installer.

Start the IDE for the First Time

1. Start MICROEJ SDK and select a workspace.

   Note: If you are not familiar with Eclipse workspaces, select the default and press OK.

2. Select the MICROEJ repository.

   Note: If you are not familiar with MICROEJ repositories, select the default and press OK.

Prepare VEE Port Sources

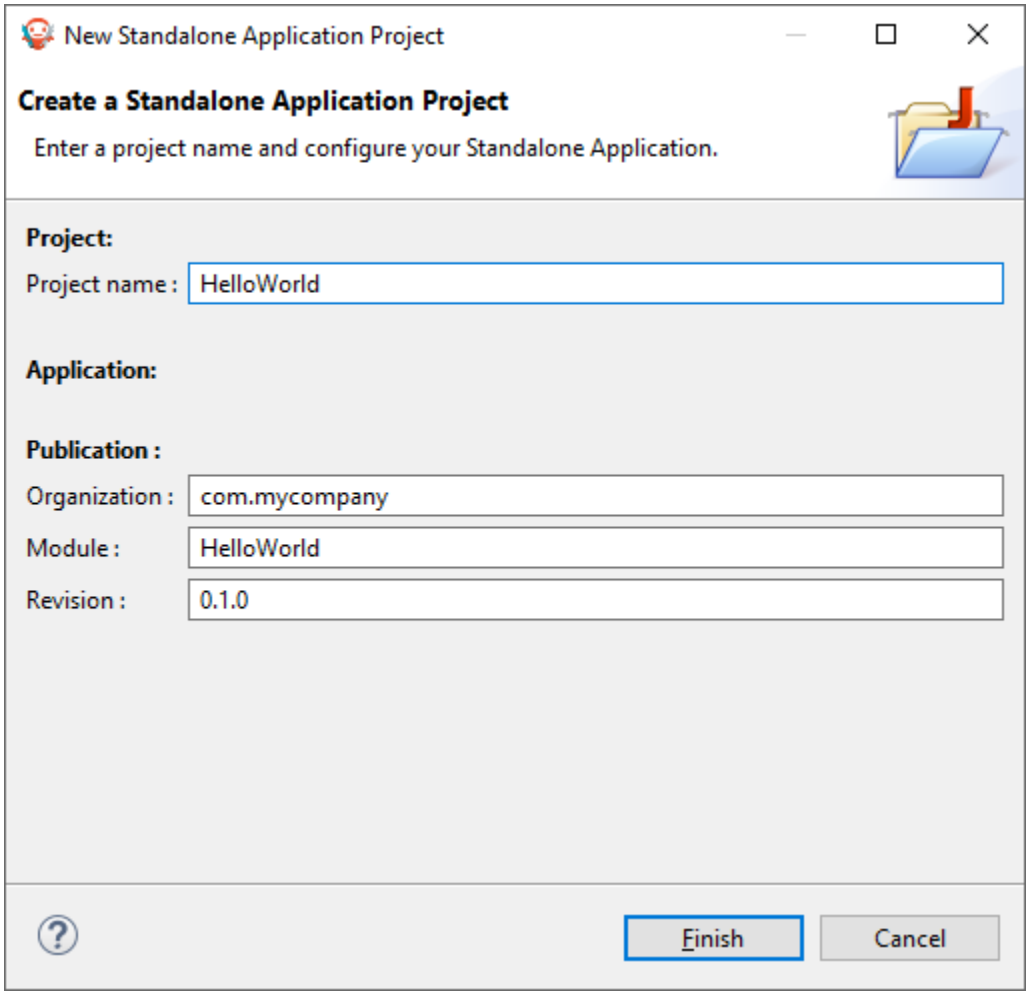
1. Get the VEE Port sources from GitHub for STM32F7508-DK, open a terminal on your workstation and run the following commands:

```
git clone --recursive https://github.com/MicroEJ/VEEPort-STMicroelectronics-
↪STM32F7508-DK.git
cd VEEPort-STMicroelectronics-STM32F7508-DK
git checkout tags/2.2.0
```

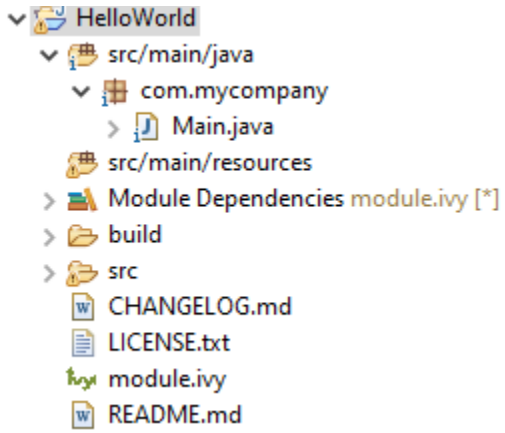
2. Follow the README to import the VEE Port sources, activate your license and build your VEE Port, in the VEE Port Setup section.
3. Once all the steps of the VEE Port setup are done, a new Java project can be created.

Create a New Project

Go to File > New > Standalone Application Project :



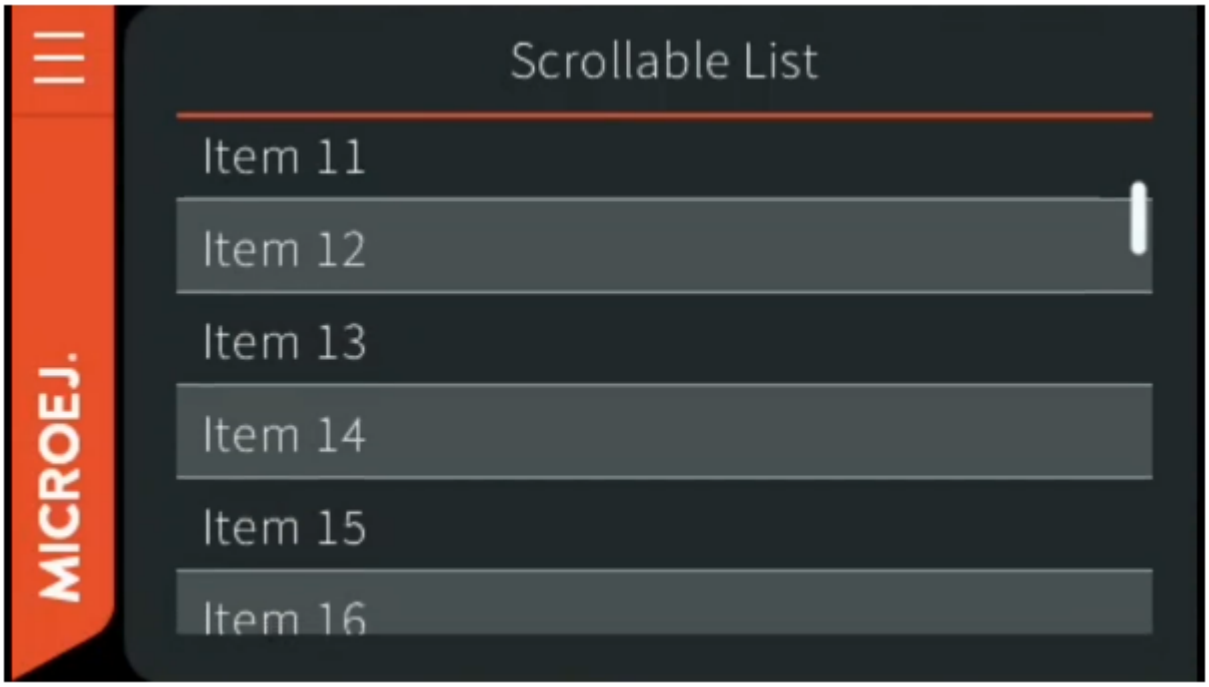
The project structure should look like this:



Featured Project: Widget Demo

You can have a look at the widget demo project, which contains multiple samples of widgets and usage.

- Widget Demo GitHub Repository



Next step: *Starting MicroUI*

9.13.2 Starting MicroUI

1. To get started, first we need to add MicroUI, a Foundation Library that provides an Abstraction Layer to access the low-level UI inputs and outputs.
2. Look for `module.ivy` , and replace dependencies with the following:

```
<dependencies>
   <dependency org="ej.api" name="microui" rev="3.4.1"/>
</dependencies>
```

> **Note:** There's no need to add EDC as a dependency. It will be automatically resolved with the correct version (as a dependency of the MicroUI library).

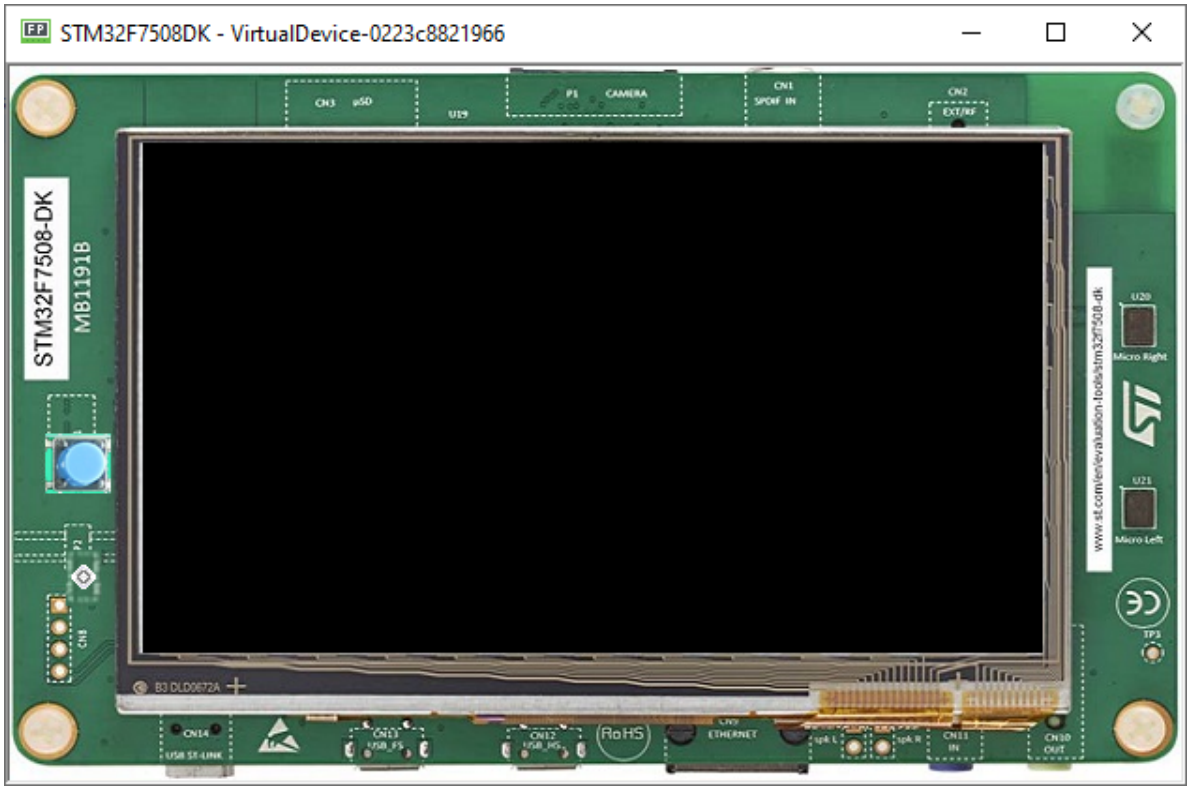
3. This call initializes the MicroUI framework and starts the UI Thread, which manages the user input and display events.

```
public static void main(String[] args) {
   MicroUI.start();
}
```

Note: MicroUI has to be started before any UI operations.

4. To run your code on the Simulator, left click on the Project Go To `Run` > `Run As` > `MicroEJ Application` .

Note: If you have several VEE Ports you will be asked which to use.



Widgets

1. The widget library provides a collection of common widgets and containers. It is based on MWT, a base library that defines core type graphical elements for designing rich graphical user interface embedded applications.
2. Look for `module.ivy` , and replace dependencies with the following:

```
<dependencies>
   <dependency org="ej.library.ui" name="widget" rev="5.0.0" />
</dependencies>
```

Note: There's no need to add MWT or MicroUI, as both are dependencies of the Widget library. They will be automatically resolved with the correct version.

Desktop Usage

1. A desktop is the top-level object that can be displayed on a Display. It may contain a widget, and at most one desktop is shown on a Display at any given time.
2. Desktop automatically triggers the layout and rendering phases for itself and its children.

```
public static void main(String[] args) {
   MicroUI.start();

   Desktop desktop = new Desktop();
   desktop.requestShow();
}
```

Displaying a Label

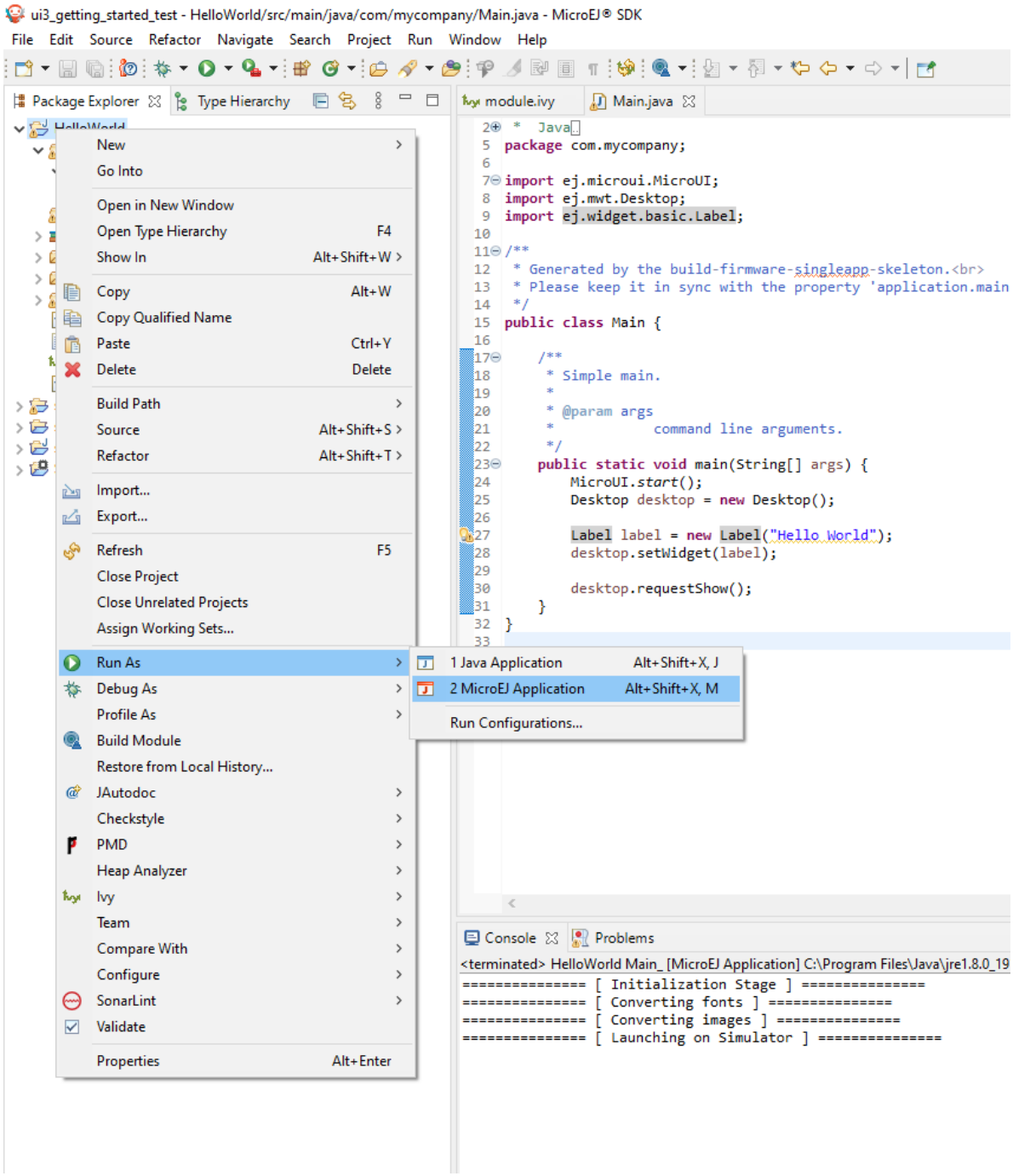
1. To add a label, just instantiate a Label object and add it to the desktop as the root widget.

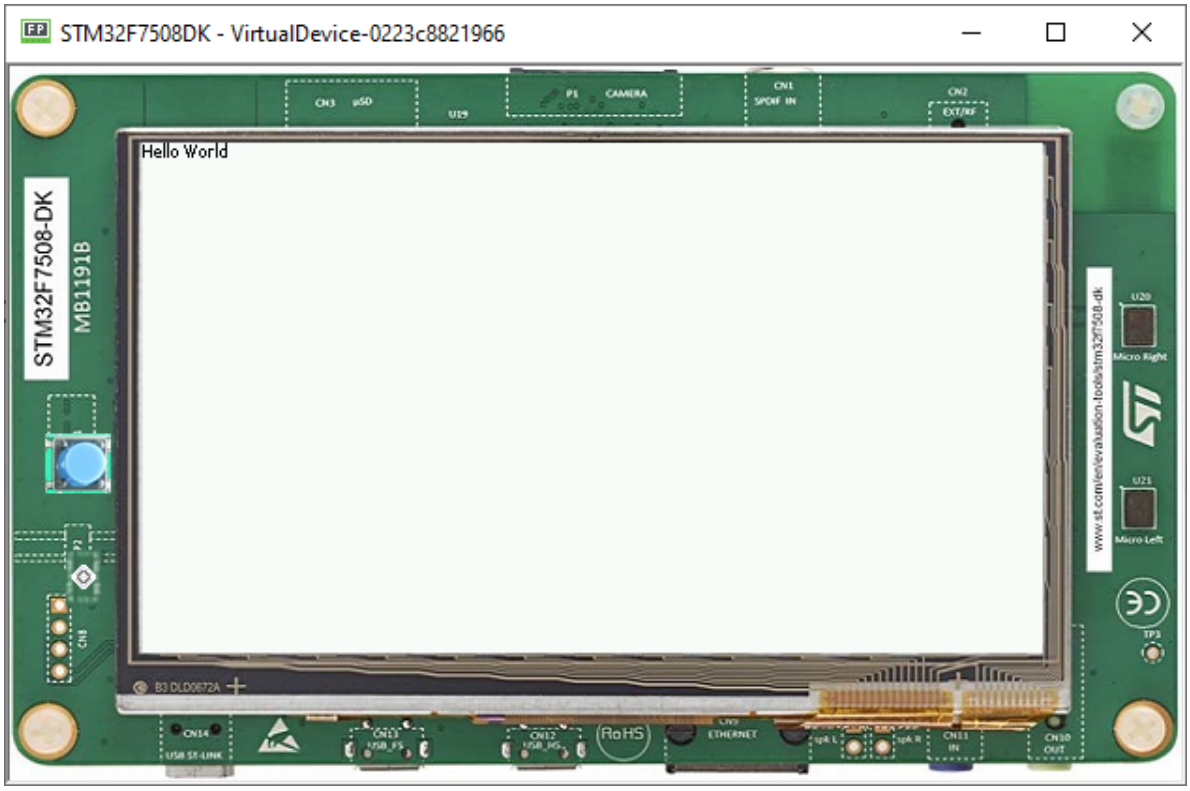
```
public static void main(String[] args) {
   MicroUI.start();
   Desktop desktop = new Desktop();

   Label label = new Label("Hello World");
   desktop.setWidget(label);

   desktop.requestShow();
}
```

2. To run the code go to the **Main.java** file and right click it, hover over Run As and select MicroEJ Application .


ui3_getting_started_test - HelloWorld/src/main/java/com/mycompany/Main.java - MicroEJ® SDK
File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer
Type Hierarchy
module.ivy
Main.java
New
Go Into
Open in New Window
Open Type Hierarchy F4
Show In Alt+Shift+W
Copy Alt+W
Copy Qualified Name
Paste Ctrl+Y
Delete Delete
Build Path
Source Alt+Shift+S
Refactor Alt+Shift+T
Import...
Export...
Refresh F5
Close Project
Close Unrelated Projects
Assign Working Sets...
Run As
Debug As
Profile As
Build Module
Restore from Local History...
JAutodoc
Checkstyle
PMD
Heap Analyzer
Ivy
Team
Compare With
Configure
SonarLint
Validate
Properties Alt+Enter
1 Java Application Alt+Shift+X, J
2 MicroEJ Application Alt+Shift+X, M
Run Configurations...
package com.mycompany;
import ej.microui.MicroUI;
import ej.mwt.Desktop;
import ej.widget.basic.Label;
public class Main {
public static void main(String[] args) {
MicroUI.start();
Desktop desktop = new Desktop();
Label label = new Label("Hello World");
desktop.setWidget(label);
desktop.requestShow();
}
}
Console
Problems
<terminated> HelloWorld Main_ [MicroEJ Application] C:\Program Files\Java\jre1.8.0_19
[Initialization Stage]
[Converting fonts]
[Converting images]
[Launching on Simulator]



Next step: *Basic Drawing on Screen*

9.13.3 Basic Drawing on Screen

- We have seen a basic use of the MWT and widgets libraries. Before going further let's see how to write directly on a display using both Displayable and GraphicsContext classes.
- A Displayable represents what can be shown on a screen, a GraphicsContext provides access to a modifiable (readable and writable) pixel buffer to be associated with an Image or a Displayable.
- It is then possible to have access to a drawable interface that represents a pixelated version of the Display.

```
public static void main(String[] args){
   MicroUI.start();
   Displayable myDisplayable = new Displayable() {

      @Override
      protected void render(GraphicsContext g) {
         // Draws a yellow line.
         g.setColor(Colors.YELLOW);
         Painter.drawLine(g, 0, 0, 100, 50);
      }

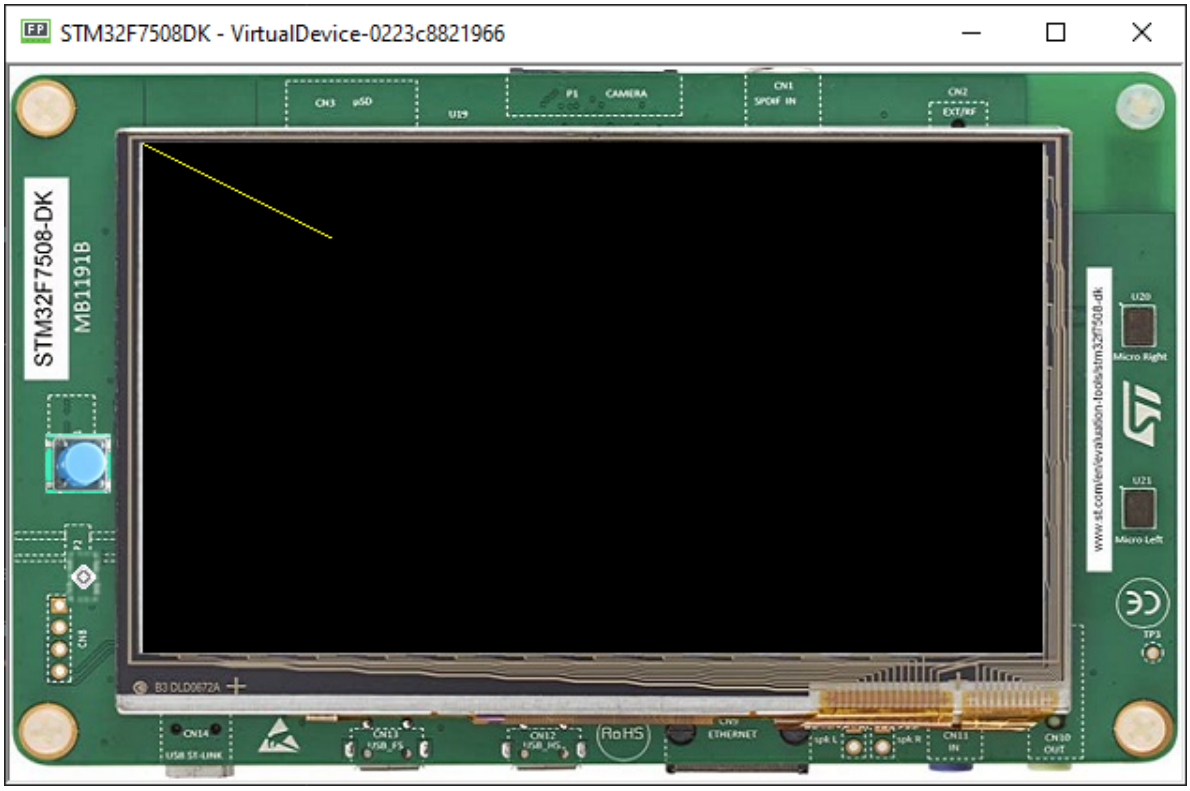
      @Override
      public boolean handleEvent(int event) {
         return false;
      }
   };
```

(continues on next page)

(continued from previous page)

```
   Display.getDisplay().requestShow(myDisplayable);
}
```

- This draws a line from the coordinates of the display `(0,0)` to `(100,50)`.



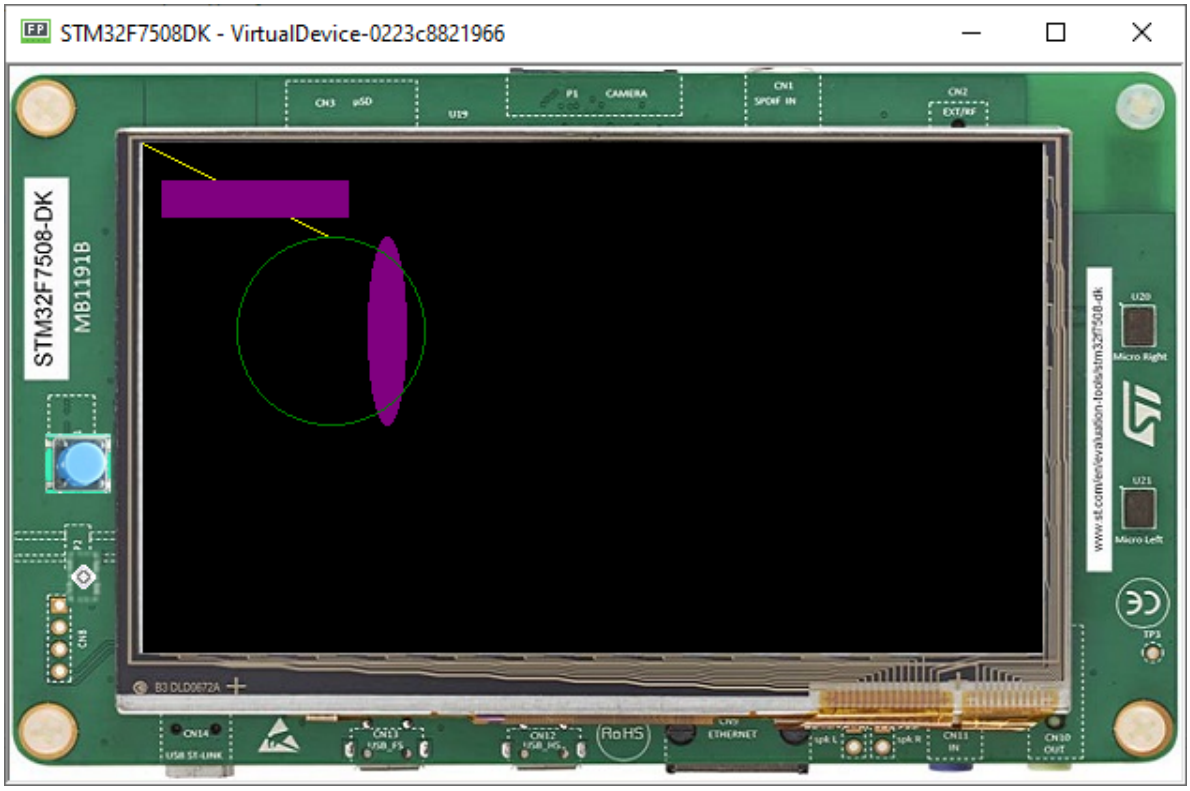
Drawing Basic Shapes

- The Painter class contains several primitives to draw geometric objects.
- The code below draws each component with the selected color (yellow, purple, green).
- The drawLine() method uses the starting and finishing point with x and y coordinates.
- Fill rectangle and ellipse methods use x and y coordinates and also width and height.
- Draw circle uses x and y and a diameter.

```
g.setColor(Colors.YELLOW);
Painter.drawLine(g, 0, 0, 100, 50);

g.setColor(Colors.PURPLE);
Painter.fillRectangle(g, 10, 20, 100, 20);
Painter.fillEllipse(g, 120, 50, 20, 100);

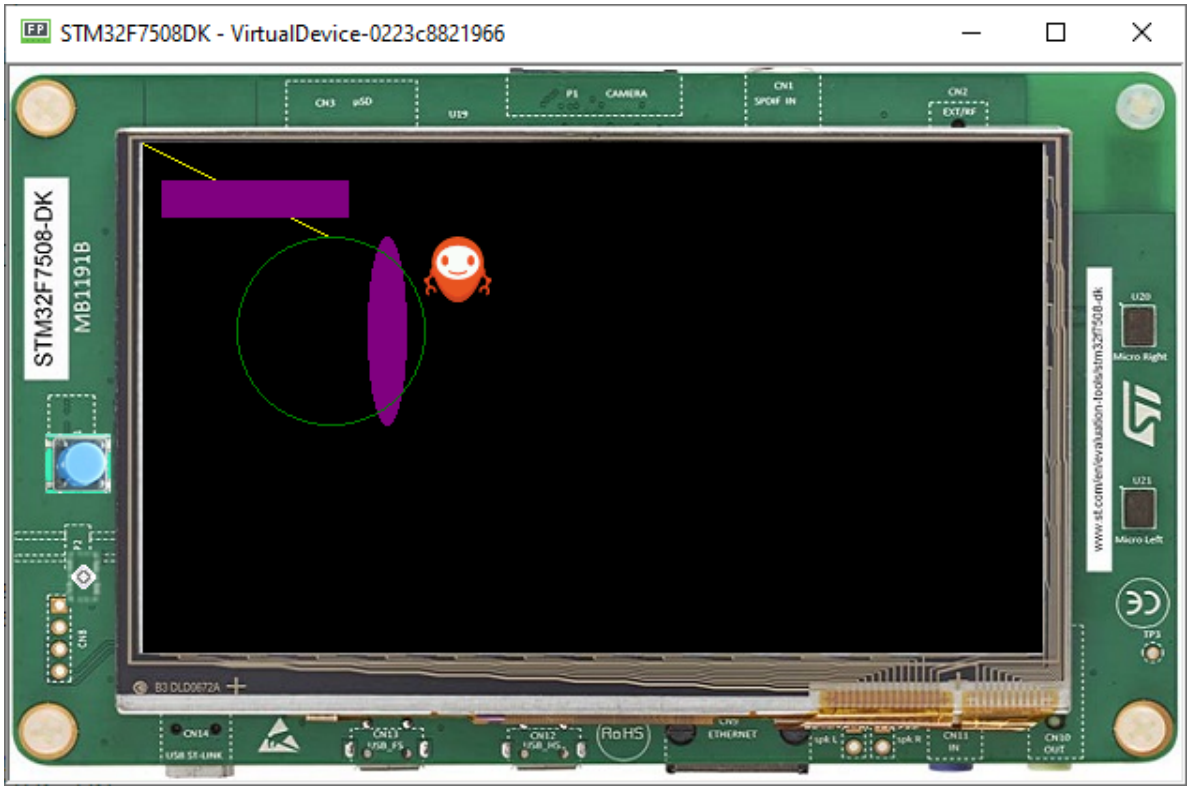
g.setColor(Colors.GREEN);
Painter.drawCircle(g, 50, 50, 100);
```



Drawing Images

- The Painter class contains several primitives to draw images.

```
Image image = Image.getImage("/images/microej_logo.png");
// Draws the image at x,y coordinates (150, 50).
Painter.drawImage(g, image, 150, 50);
```

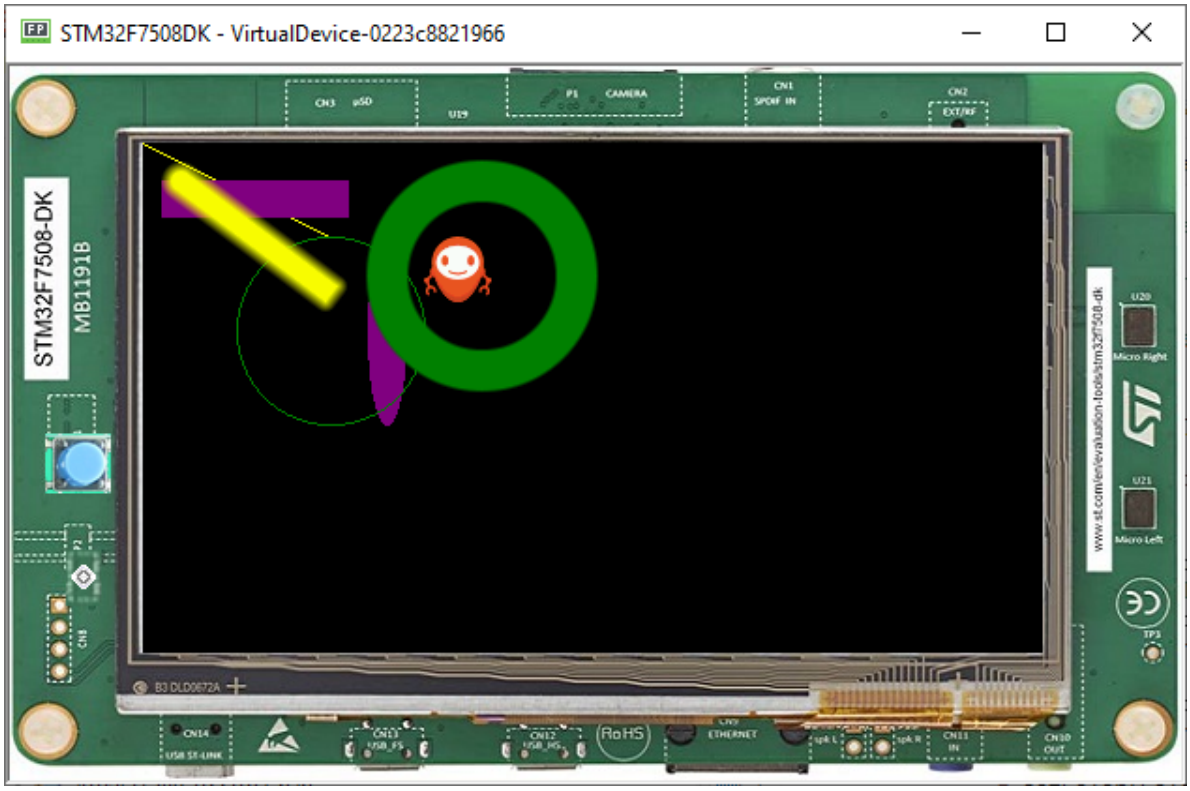


Drawing Thick Shapes

- The ShapePainter class offers a set of primitives to render thick shapes with or without anti-aliasing.
- The code below shows how to draw a thick faded line.

```
// Draws a thick yellow line.
g.setColor(Colors.YELLOW);
ShapePainter.drawThickFadedLine(g, 20, 20, 100, 80, 10, 6, Cap.ROUNDED, Cap.
↪PERPENDICULAR);

// Draws a thick green circle.
g.setColor(Colors.GREEN);
ShapePainter.drawThickFadedCircle(g, 130, 20, 100, 20, 2);
```



Next step: *Animation*

9.13.4 Animation

Animations can be used to make the GUI more appealing and more lively.

MWT provides a framework to create fluid animations. The principle is as follow:

- make a step of all the running animations (with a probable new rendering of some widgets),
- wait for the display to be flushed,
- do it again.

The goals are:

- doing animations as fast as possible (considering the complexity of the drawings and the hardware capabilities),
- synchronizing all the running animations and avoiding glitches.

Usage

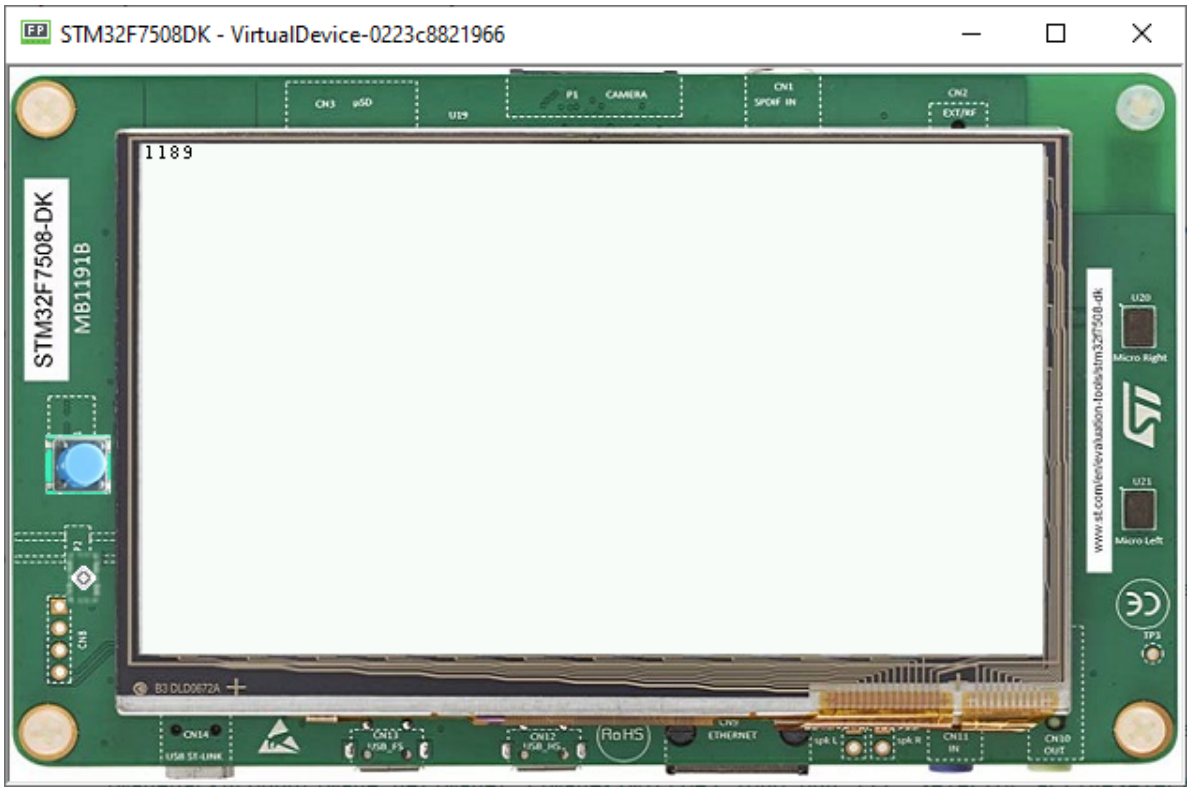
- An animation can be created by implementing the Animation interface and its tick() method.
- The tick() method is called for each step of the animation.
- Every time the method is called, the widget should be re-rendered.
- The animation can be stopped by returning `false`.

```
Animation labelAnimation = new Animation() {

   int tick = 0;

   @Override
   public boolean tick(long currentTimeMillis) {
      label.setText(Integer.toString(tick++));
      label.requestRender();
      return true;
   }
};
Animator animator = new Animator();
animator.startAnimation(labelAnimation);
```

- The code above updates the label text everytime it is called:



- The final code looks like this:

```
public static void main(String[] args) {
   MicroUI.start();
```

(continues on next page)

(continued from previous page)

```
   Desktop desktop = new Desktop();
   final Label label = new Label("hello");

   Flow flow = new Flow(LayoutOrientation.VERTICAL);
   flow.addChild(label);

   Animation labelAnimation = new Animation() {

      int tick = 0;

      @Override
      public boolean tick(long currentTimeMillis) {
         label.setText(Integer.toString(this.tick++));
         label.requestRender();
         return true;
      }
   };
   Animator animator = new Animator();
   animator.startAnimation(labelAnimation);

   desktop.setWidget(flow);
   desktop.requestShow();
}
```

Next step: *Creating Widgets*

9.13.5 Creating Widgets

- To create a widget, we need to create a class that extends the Widget superclass.
- In this example, we are going to create a simple progress bar.
- So create a MyProgressBarWidget class extending Widget.

Note: The computeContentOptimalSize() and renderContent() methods must be overridden:

```
public class MyProgressBarWidget extends Widget {
   @Override
   protected void computeContentOptimalSize(Size size) {
      // TODO Auto-generated method stub
   }

   @Override
   protected void renderContent(GraphicsContext g, int contentWidth, int contentHeight) {
      // TODO Auto-generated method stub
   }
}
```

Setting Up

- Let's use a progress bar with a fixed size:

```
protected void computeContentOptimalSize(Size size) {
   size.setSize(200,50);
}
```

- Then, let's create the progress bar, first, it is important to add a progress value:

```
private float progressValue;
```

- Now, let's render the progress bar:

```
protected void renderContent(GraphicsContext g, int contentWidth, int contentHeight) {
   // Draws the remaining bar: a 1 px thick grey line, with 1px of fading.
   g.setColor(Colors.SILVER);
   int halfHeight = contentHeight / 2;
   ShapePainter.drawThickFadedLine(g, 0, halfHeight, contentWidth, halfHeight, 1, 1,␣
↪Cap.ROUNDED, Cap.ROUNDED);

   // Draws the progress bar: a 3 px thick blue line, with 1px of fading.
   g.setColor(Colors.NAVY);
   int barWidth = (int) (contentWidth * this.progressValue);
   ShapePainter.drawThickFadedLine(g, 0, halfHeight, barWidth, halfHeight, 3, 1, Cap.
↪ROUNDED, Cap.ROUNDED);
}
```

- Finally, let's create a method to set the progress on the progress bar:

```
public void setProgress(float progress) {
   this.progressValue = progress;
}
```

Using with Animator

- Using the code made in the previous Animation tutorial, doing the modifications below, it is now possible to see the progress bar animated:

```
public static void main(String[] args) {
   MicroUI.start();
   Desktop desktop = new Desktop();
   final MyProgressBarWidget progressBar = new MyProgressBarWidget();
   Flow flow = new Flow(LayoutOrientation.VERTICAL);
   flow.addChild(progressBar);

   Animation progressBarAnimation = new Animation() {

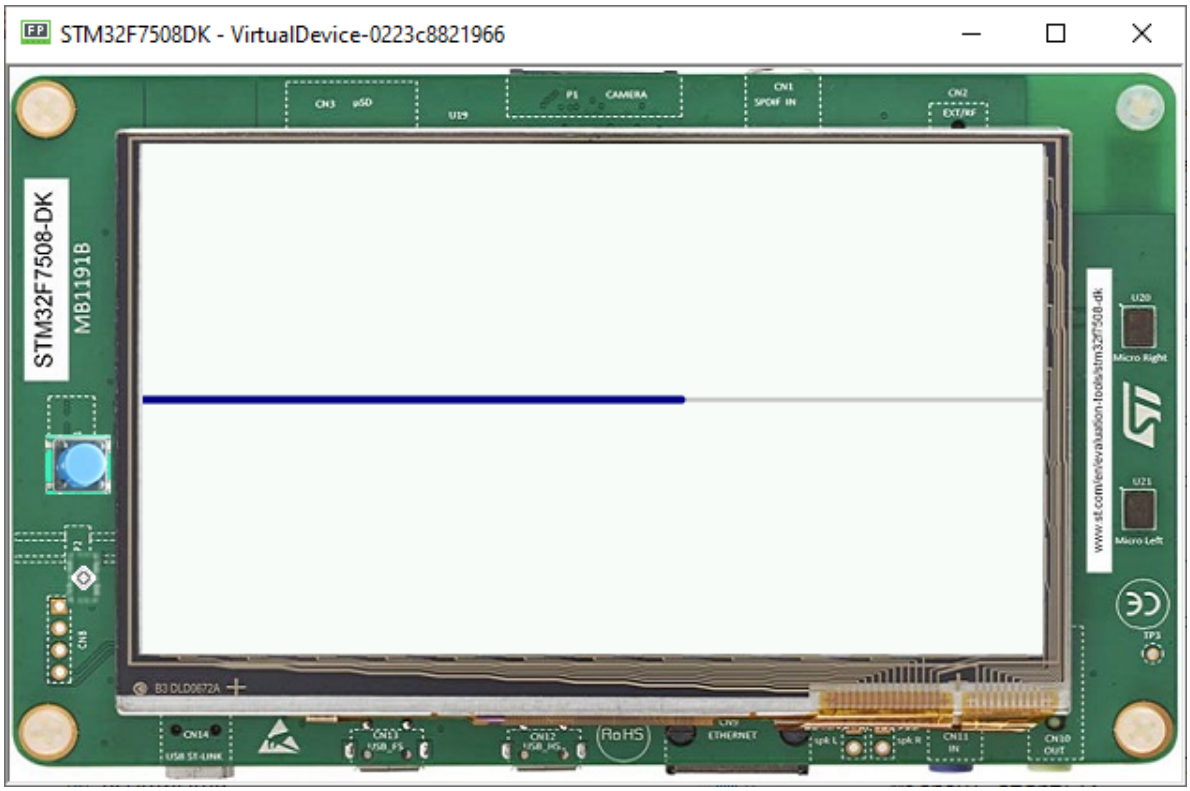
      float progress;

      @Override
      public boolean tick(long currentTimeMillis) {
         this.progress += 0.001f;
```

(continues on next page)

(continued from previous page)

```
            progressBar.setProgress(this.progress);
            progressBar.requestRender();
            return true;
        }
    };
    Animator animator = desktop.getAnimator();
    animator.startAnimation(progressBarAnimation);
    desktop.setWidget(flow);
    desktop.requestShow();
}
```



Next step: *Using Layouts*

9.13.6 Using Layouts

The lay out process determines the position and size of the widgets. It depends on:

- The layout of the containers: how the children are arranged within the containers.
- The widgets content size: the size needed by the widgets for optimal display.

This process is started automatically when the desktop is shown. It can also be triggered programmatically.

Using a Flow Layout

The flow layout lays out any number of children horizontally or vertically, using multiple rows if necessary depending on the size of each child widget.

Creating a flow layout:

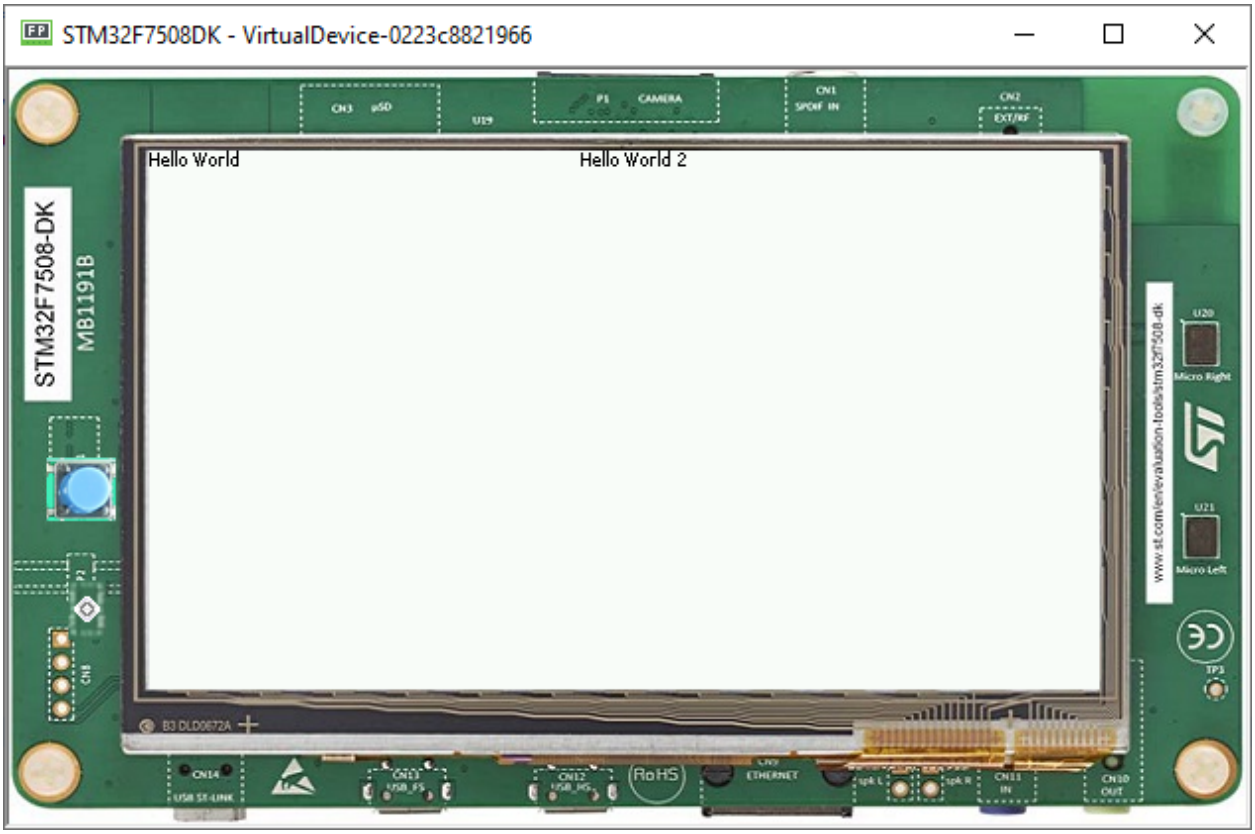
- First, instantiate a Flow container, then add two Label objets to this container.
- Finally, add the Flow container to the Desktop.

```
public static void main(String[] args) {
   MicroUI.start();
   Desktop desktop = new Desktop();
   Label label = new Label("Hello World");
   Label secondLabel = new Label("Hello World 2");

   Flow flowContainer = new Flow(LayoutOrientation.HORIZONTAL);
   flowContainer.addChild(label);
   flowContainer.addChild(secondLabel);

   desktop.setWidget(flowContainer);
   desktop.requestShow();
}
```

Both of the labels will share the screen:



Using a Canvas

A canvas lays out any number of children freely.

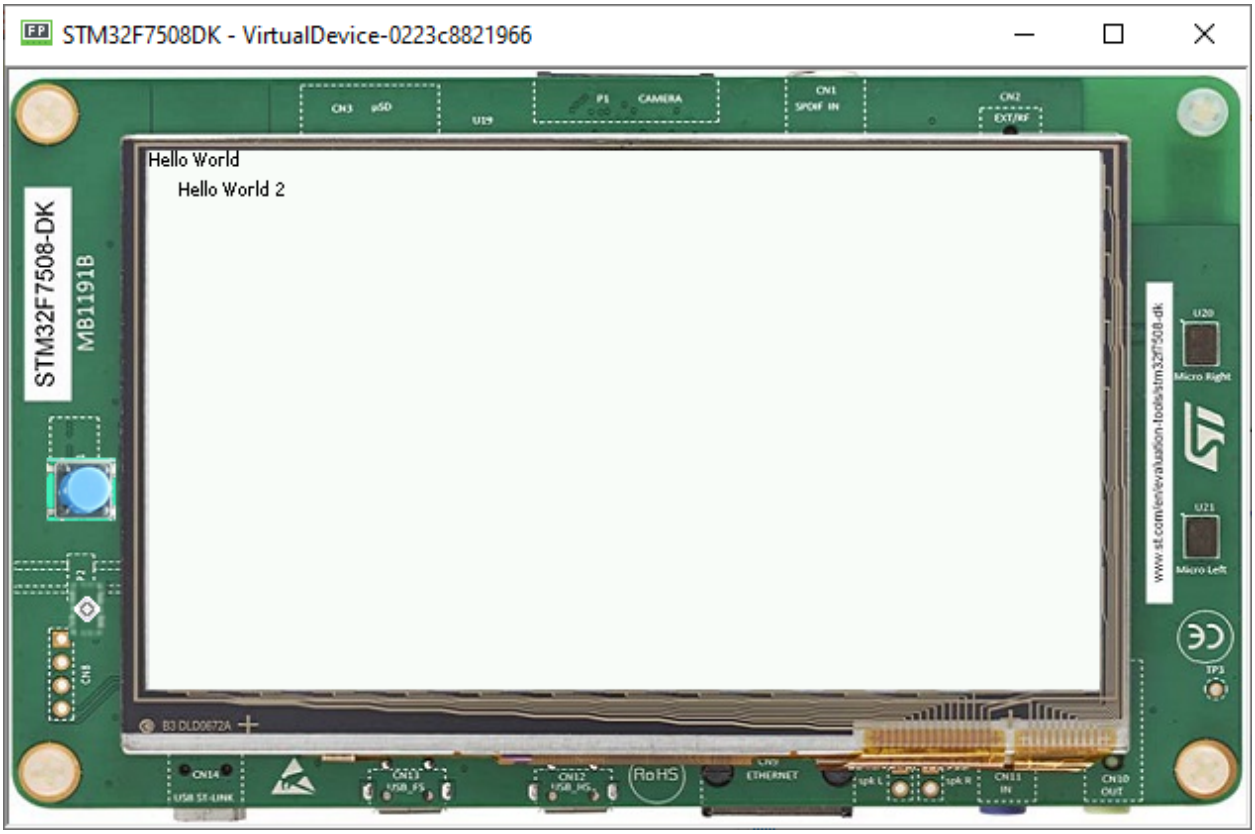
To add a widget to a Canvas, specify its position and size.

Note: Using Widget.NO_CONSTRAINT sets the width and height to the optimal size of the widget.

```
public static void main(String[] args) {
   MicroUI.start();
   Desktop desktop = new Desktop();
   Label label = new Label("Hello World");
   Label label2 = new Label("Hello World 2");

   Canvas canvas = new Canvas();
   canvas.addChild(label, 0, 0, Widget.NO_CONSTRAINT, Widget.NO_CONSTRAINT);
   canvas.addChild(label2, 15, 15, Widget.NO_CONSTRAINT, Widget.NO_CONSTRAINT);

   desktop.setWidget(canvas);
   desktop.requestShow();
}
```



Next step: *Style*

9.13.7 Style

Instances of Desktop, Widget, and Container classes are semantic elements of the GUI, describing the structure and meaning of the content.

The Style API (ej.mwt.style) defines style options for widgets, allowing for a clear separation of the core structure (content) and the design aspects (colors, fonts, spacing, background, etc.).

Note: Some of the attributes are inspired by CSS, like Background, Border, Color, Dimension, Font, Alignment, Margin/Padding. And the CascadingStylesheet manages the order of the selectors (with their specificity), the cascading, etc.

Selectors

Selectors determine the widget(s) to which a style applies. There are three main types of selectors:

- Simple selectors (based on type or class),
- State Selectors (based on state or position),
- Combinators (based on relationships or conditions).

Note: More of this will be presented in the *Advanced Styling* step.

Usage

- With a CascadingStylesheet, we can define a style for all labels using a TypeSelector:

```
CascadingStylesheet stylesheet = new CascadingStylesheet();
EditableStyle style = stylesheet.getSelectorStyle(new TypeSelector(Label.class));
```

- We can now change the style object options. In this sample, we change the base color to red and adding a black rectangular border of 1px thickness.

```
style.setColor(Colors.RED);
style.setBorder(new RectangularBorder(Colors.BLACK, 1));
```

- For these options to take effect, bind the stylesheet to the desktop.

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

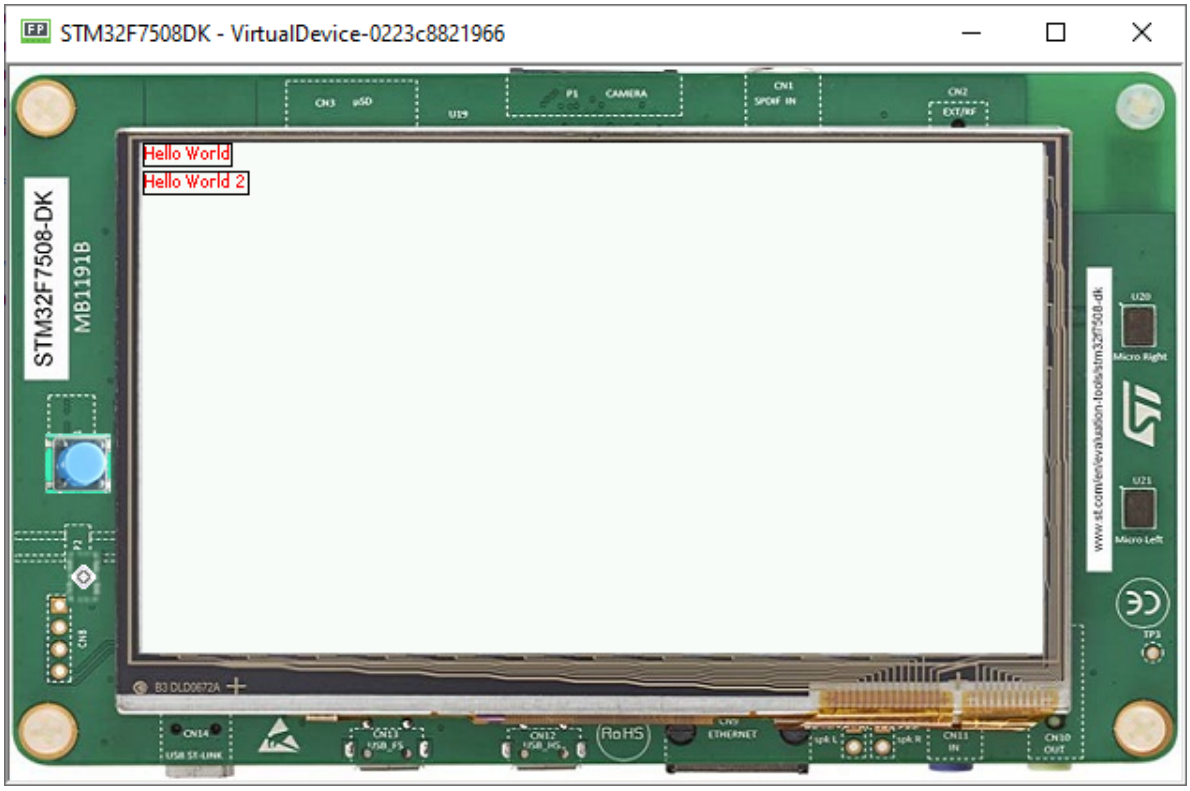
- The final code looks like this:

```
public static void main(String[] args) {
   MicroUI.start();
   Desktop desktop = new Desktop();
   Label label = new Label("Hello World");
   Label label2 = new Label("Hello World 2");

   Canvas canvas = new Canvas();
   canvas.addChild(label, 0, 0, Widget.NO_CONSTRAINT, Widget.NO_CONSTRAINT);
   canvas.addChild(label2, 0, 15, Widget.NO_CONSTRAINT, Widget.NO_CONSTRAINT);

   CascadingStylesheet stylesheet = new CascadingStylesheet();
   EditableStyle style = stylesheet.getSelectorStyle(new TypeSelector(Label.class));
   style.setColor(Colors.RED);
   style.setBorder(new RectangularBorder(Colors.BLACK, 1));

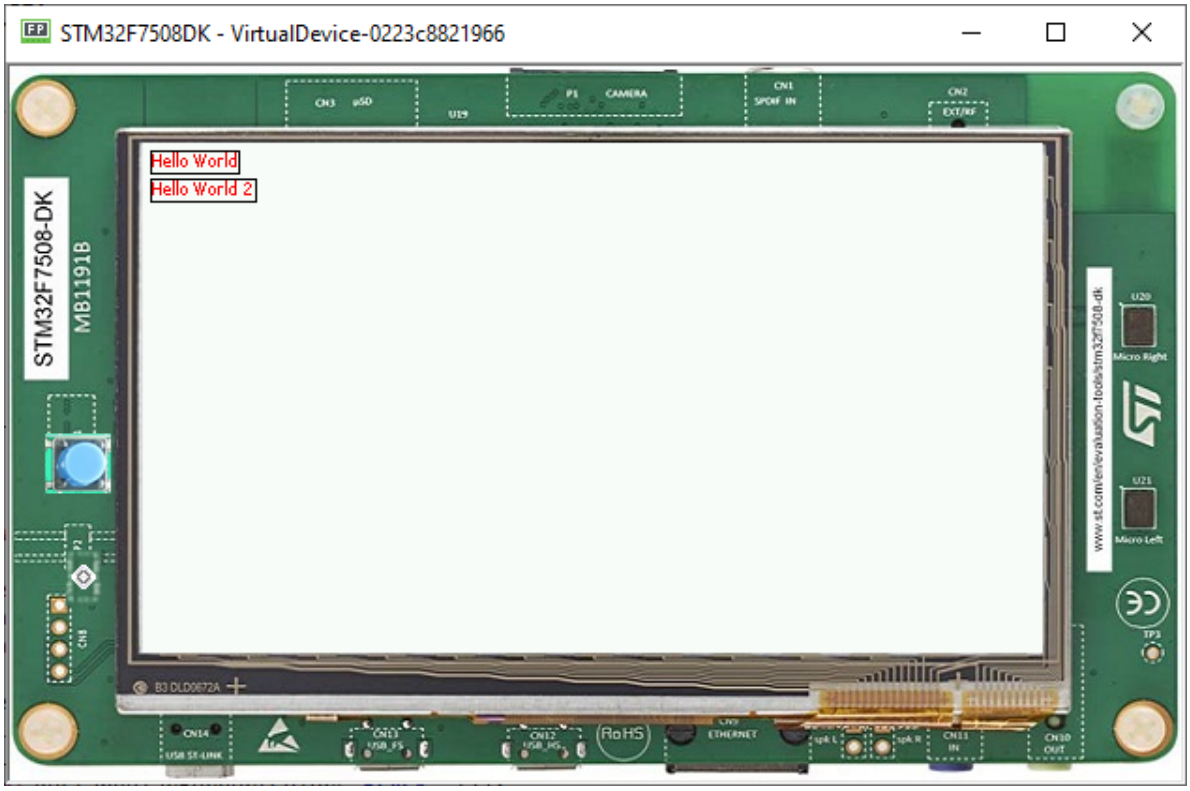
   desktop.setStylesheet(stylesheet);
   desktop.setWidget(canvas);
   desktop.requestShow();
}
```



Padding and Margin

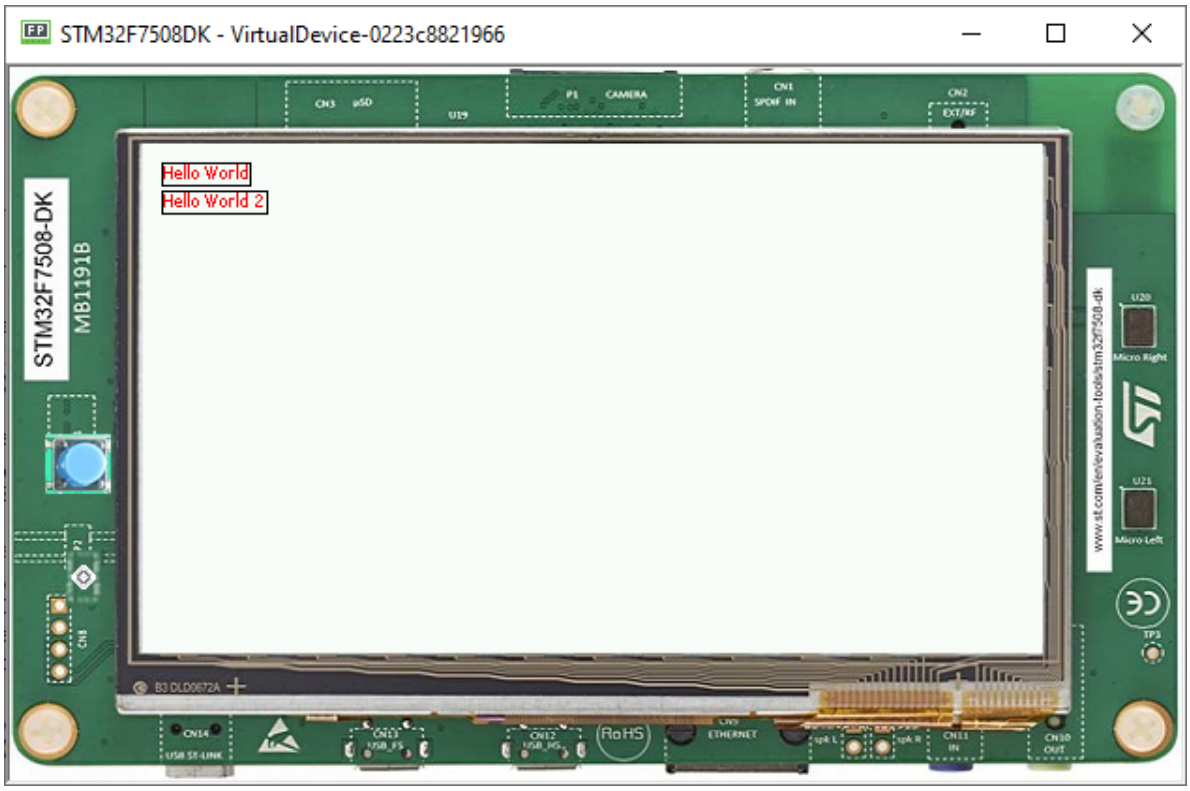
- Using margin and padding is pretty simple. Adding margin is as follows:

```
style.setMargin(new UniformOutline(4));
```



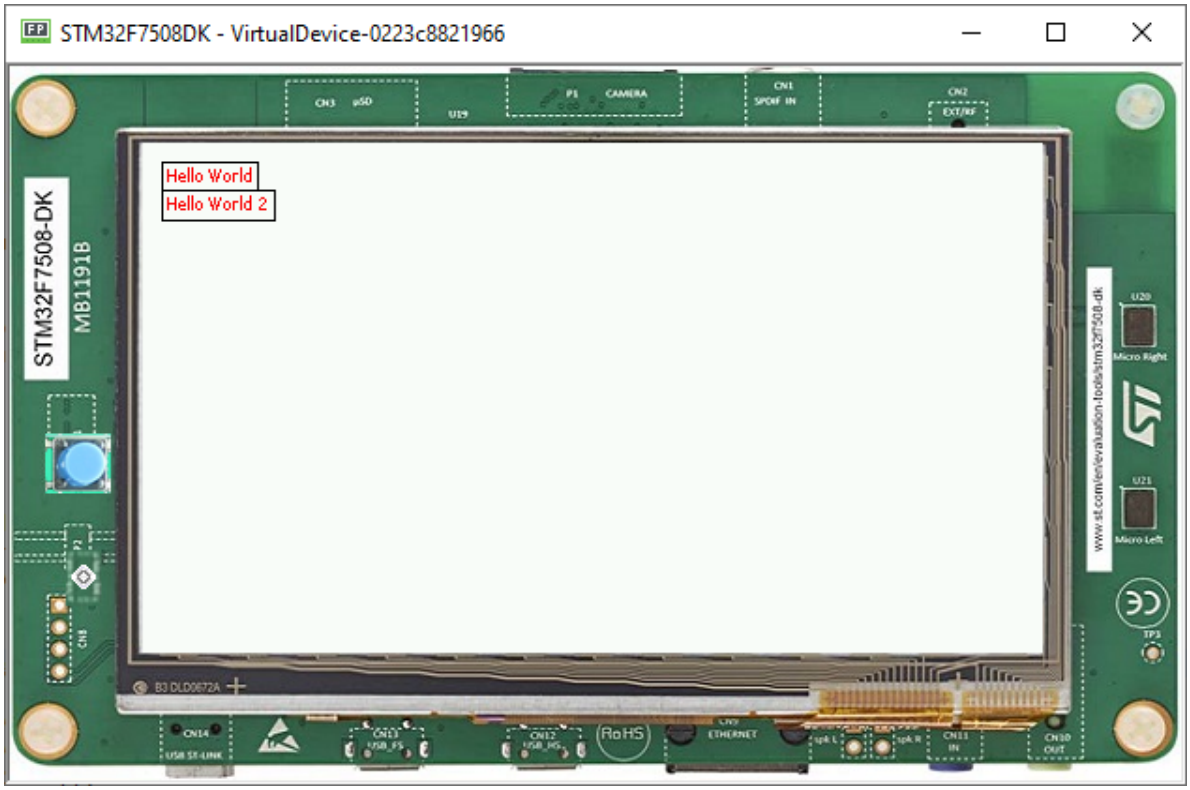
- Setting an oversized margin looks like this:

```
style.setMargin(new UniformOutline(10));
```



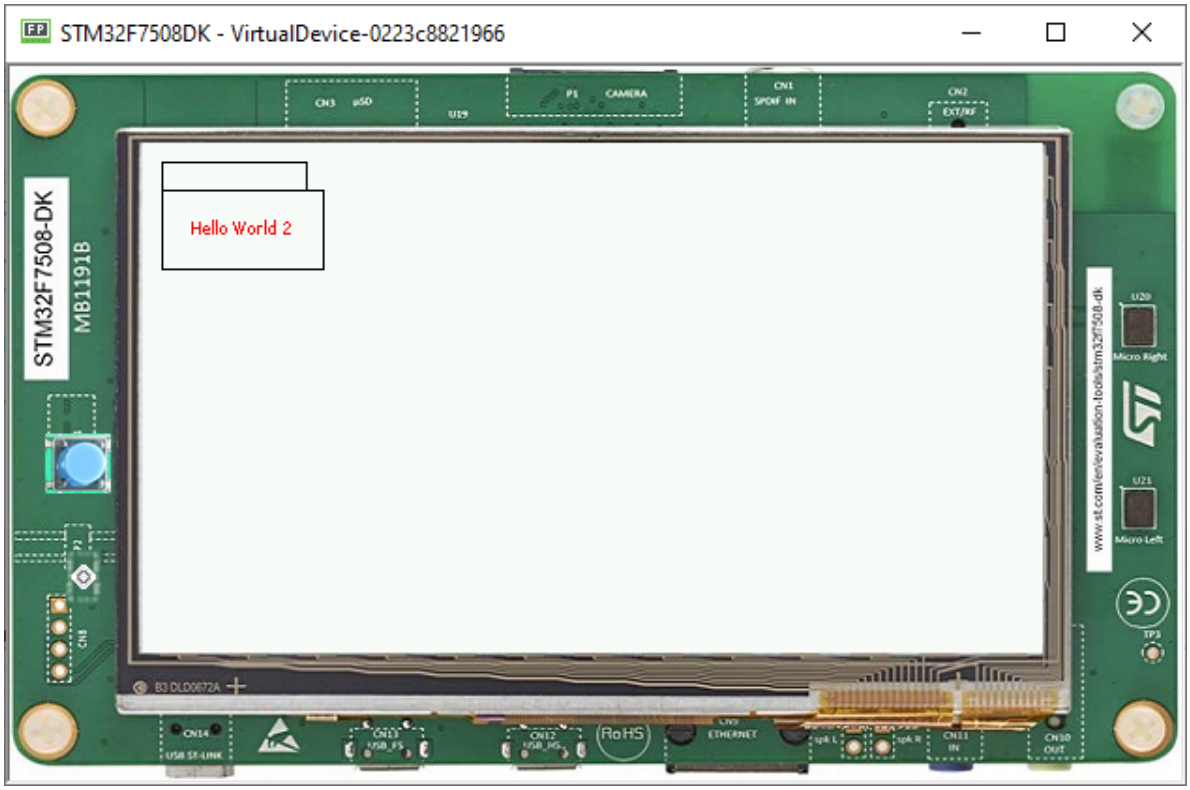
- Adding padding:

```
style.setPadding(new UniformOutline(2));
```



- Oversizing the padding (the widgets ovelap each other because we use a canvas with fixed positions):

```
style.setPadding(new UniformOutline(15));
```

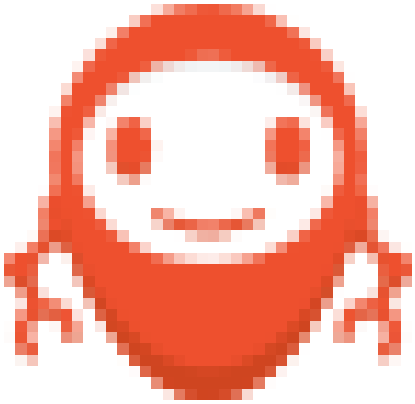


Next step: *Images*

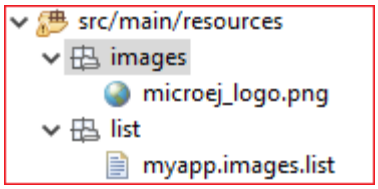
9.13.8 Images

Adding Images

- Create two packages in the Resources folder, one named `list` and another named `images`.
- Create an images list file, and add it to the `list` package (`myapp.images.list`).
- Save the following image to the `images` package:



- The structure looks like this:



- Then go to the `myapp.images.list` and add the image file:

```
/images/microej_logo.png:ARGB4444
```

- The image declaration in the .list file follows this pattern:

```
path:format
```

- `path` is the path to the image file, relative to the `resources` folder.
- `format` specifies how the image will be embedded in the application.

Note: The ARGB4444 mode is used here because the image has transparency, more info in the *Images* section.

Displaying an Image

- To display this image, first create an instance of the widget ImageWidget, specifying the path to the image in the constructor:

```
ImageWidget image = new ImageWidget("/images/microej_logo.png");
```

- Add the widget to the canvas container by adding this line:

```
canvas.addChild(image, 0, 30, Widget.NO_CONSTRAINT, Widget.NO_CONSTRAINT);
```

- The final code looks like this:

```
public static void main(String[] args) {
   MicroUI.start();
   Desktop desktop = new Desktop();
   Label label = new Label("Hello World");
   Label label2 = new Label("Hello World 2");

   Canvas canvas = new Canvas();
   canvas.addChild(label, 0, 0, Widget.NO_CONSTRAINT, Widget.NO_CONSTRAINT);
   canvas.addChild(label2, 0, 15, Widget.NO_CONSTRAINT, Widget.NO_CONSTRAINT);

   ImageWidget image = new ImageWidget("/images/microej_logo.png");
   canvas.addChild(image, 0, 30, Widget.NO_CONSTRAINT, Widget.NO_CONSTRAINT);

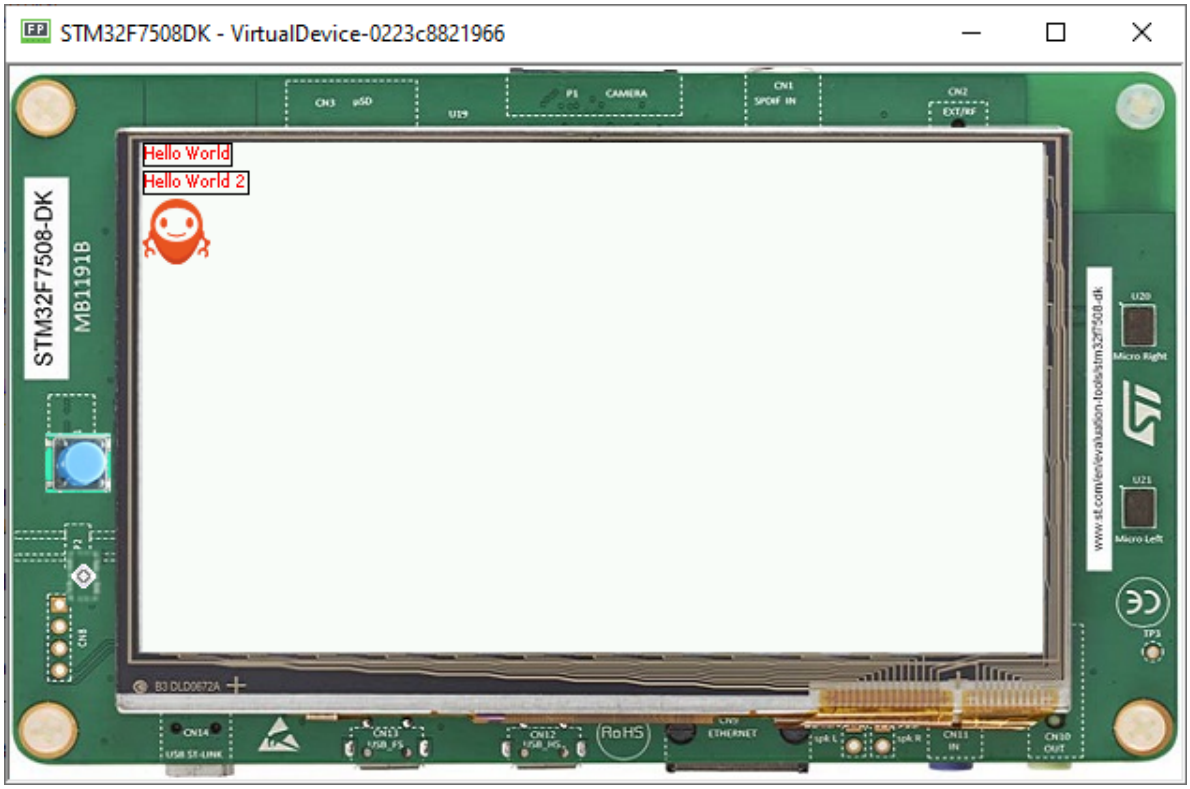
   CascadingStylesheet css = new CascadingStylesheet();
   EditableStyle style = css.getSelectorStyle(new TypeSelector(Label.class));
   style.setColor(Colors.RED);
   style.setBorder(new RectangularBorder(Colors.BLACK, 1));

   desktop.setStylesheet(css);
```

(continues on next page)

(continued from previous page)

```
    desktop.setWidget(canvas);
    desktop.requestShow();
}
```



Next step: *Advanced Styling*

9.13.9 Advanced Styling

Using Images in Stylesheet

- Let's add a button to the application, with the MicroEJ logo as background.
- Since this background will apply to a specific button, introduce a new class selector that will select this button.

Class Selector

- Just like a class in CSS, it associates to every element that is from the same class.
- Define a class for the button as follows:

```
private static final int BUTTON = 600;
```

- Bind the class `BUTTON` to the button widget:

```
Button button = new Button("Click ME");
button.addClassSelector(BUTTON);
```

- Retrieve the style for this class from the stylesheet and edit the attributes:

```
EditableStyle style = css.getSelectorStyle(new ClassSelector(BUTTON));
```

- Finally, lets add an Image Background to this Button:

```
style.setBackground(new ImageBackground(Image.getImage("/images/microej_logo.png")));
```

- And the result should be as follows:

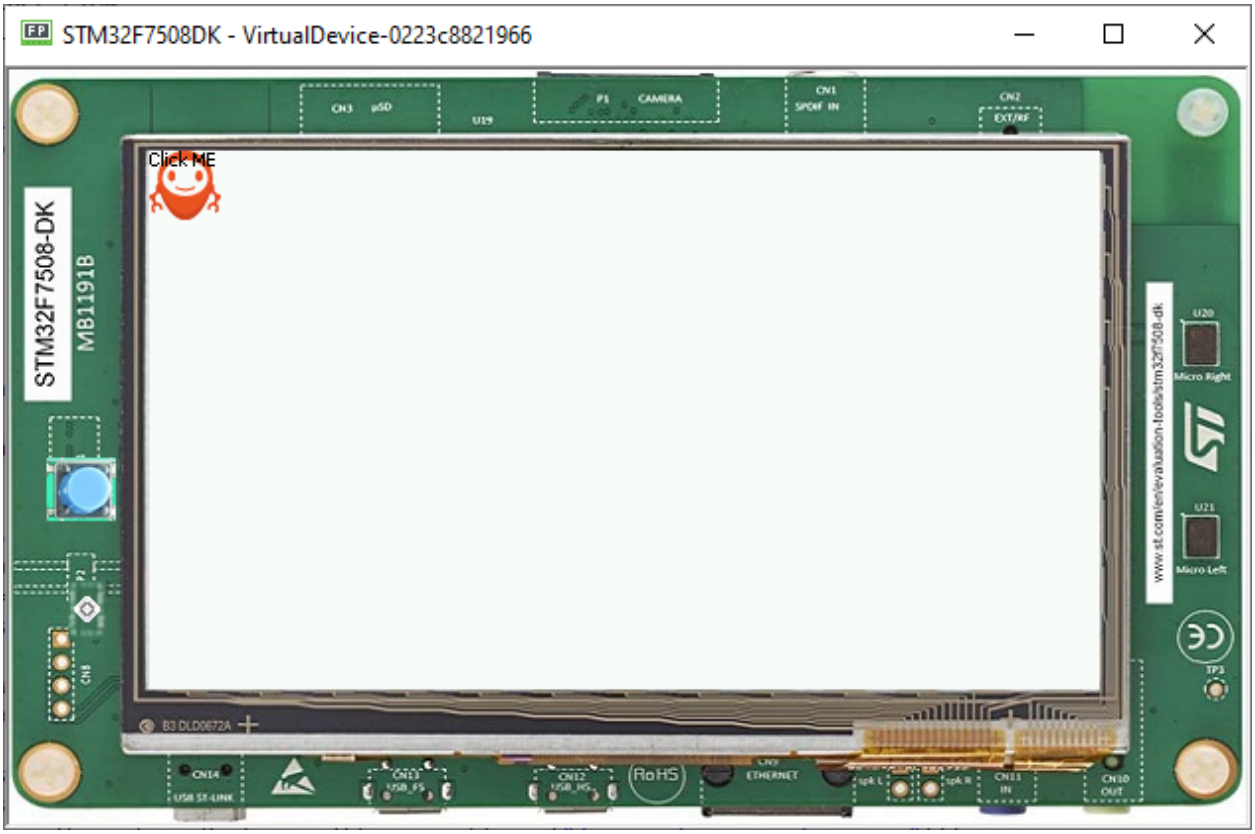
```
public class Main {
   private static final int BUTTON = 600;

   public static void main(String[] args) {
      MicroUI.start();
      Desktop desktop = new Desktop();
      Button button = new Button("Click ME");
      button.addClassSelector(BUTTON);

      Flow flow = new Flow(LayoutOrientation.VERTICAL);
      flow.addChild(button);

      CascadingStylesheet css = new CascadingStylesheet();
      EditableStyle style = css.getSelectorStyle(new ClassSelector(BUTTON));
      style.setBackground(new ImageBackground(Image.getImage("/images/microej_logo.png
↪")));

      desktop.setStylesheet(css);
      desktop.setWidget(flow);
      desktop.requestShow();
   }
}
```

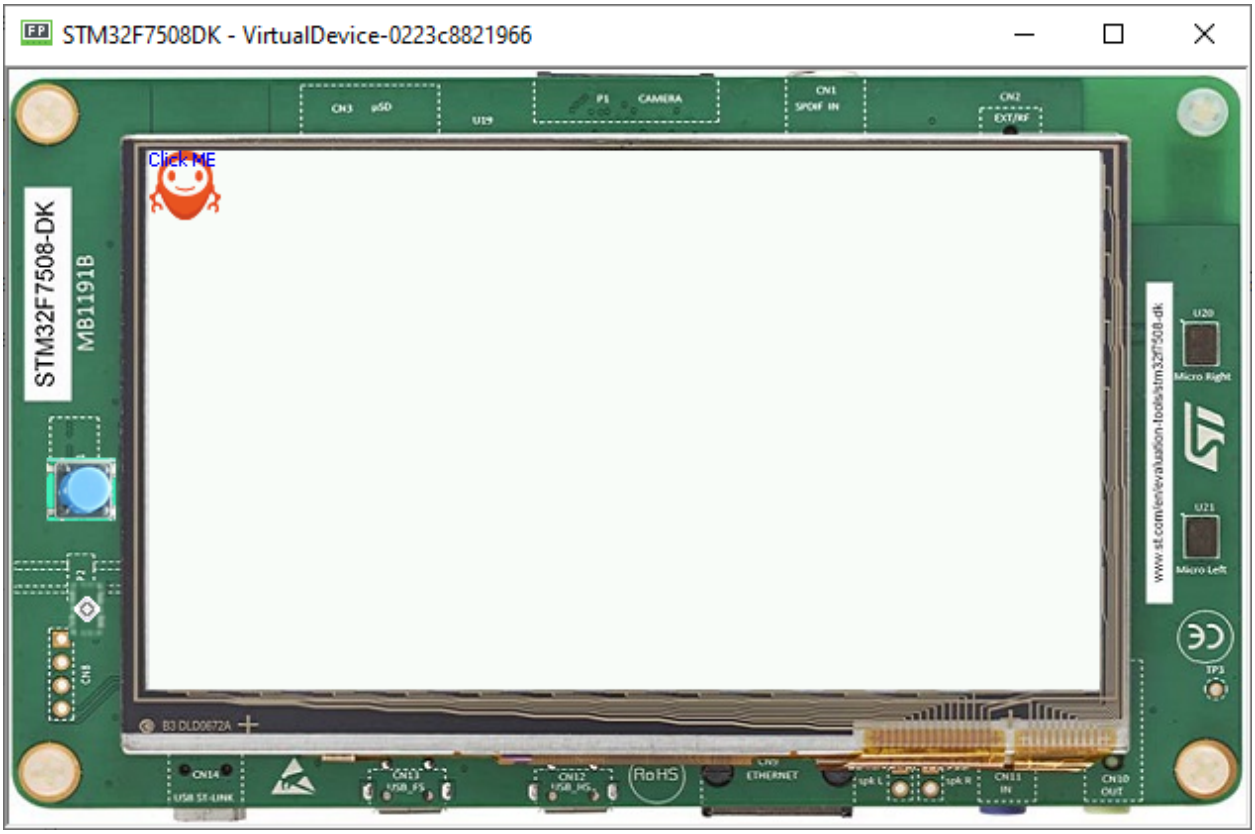


Combinator and Conditional Style

- It is possible to combine two or more selectors using combinators.
- In this example the active state of the button will turn the text blue.

```
CascadingStylesheet css = new CascadingStylesheet();
Selector buttonSelector = new ClassSelector(BUTTON);
EditableStyle style = css.getSelectorStyle(buttonSelector);
style.setBackground(new ImageBackground(Image.getImage("/images/microej_logo.png")));
Selector activeSelector = new StateSelector(Button.ACTIVE);
EditableStyle styleActive = css.getSelectorStyle(new AndCombinator(buttonSelector,␣
↪activeSelector));
styleActive.setColor(Colors.BLUE);
```

- The class selector for the button has been extracted as a locale to be combined with the button active state selector.



Next step: *Event Handling*

9.13.10 Event Handling

MicroUI generates integer-based events that encode the low-level input type and some data. The application can handle these events in the handleEvent() method.

The handleEvent Method

- Every class that extends Widget inherits the handleEvent() method.
- Add custom event handling by overriding the handleEvent() method of a widget.
- As an example, here is the event handling of the Button class:

```
@Override
public boolean handleEvent(int event) {
        int type = Event.getType(event);
        if (type == Pointer.EVENT_TYPE) {
                int action = Pointer.getAction(event);
                if (action == Pointer.PRESSED) {
                        setPressed(true);
                } else if (action == Pointer.RELEASED) {
                        setPressed(false);
                        handleClick();
                        return true;
```

(continues on next page)

(continued from previous page)

```
                }
        } else if (type == DesktopEventGenerator.EVENT_TYPE) {
                int action = DesktopEventGenerator.getAction(event);
                if (action == PointerEventDispatcher.EXITED) {
                        setPressed(false);
                }
        }
        return super.handleEvent(event);
}
```

- It's important to note that only widgets that are "enabled" will receive input events. One can enable a widget by calling setEnabled(true).
- In the Button class, the click triggers an action defined by the registered OnClickListener.

Using Events with Buttons

As an example of usage of the Button class we reuse the code created in the previous step, and add a simple action to the button by adding a OnClickListener.

```
button.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick() {
        System.out.println("Clicked!");
    }
});
```

When running the modified sample, this is shown in the console:

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

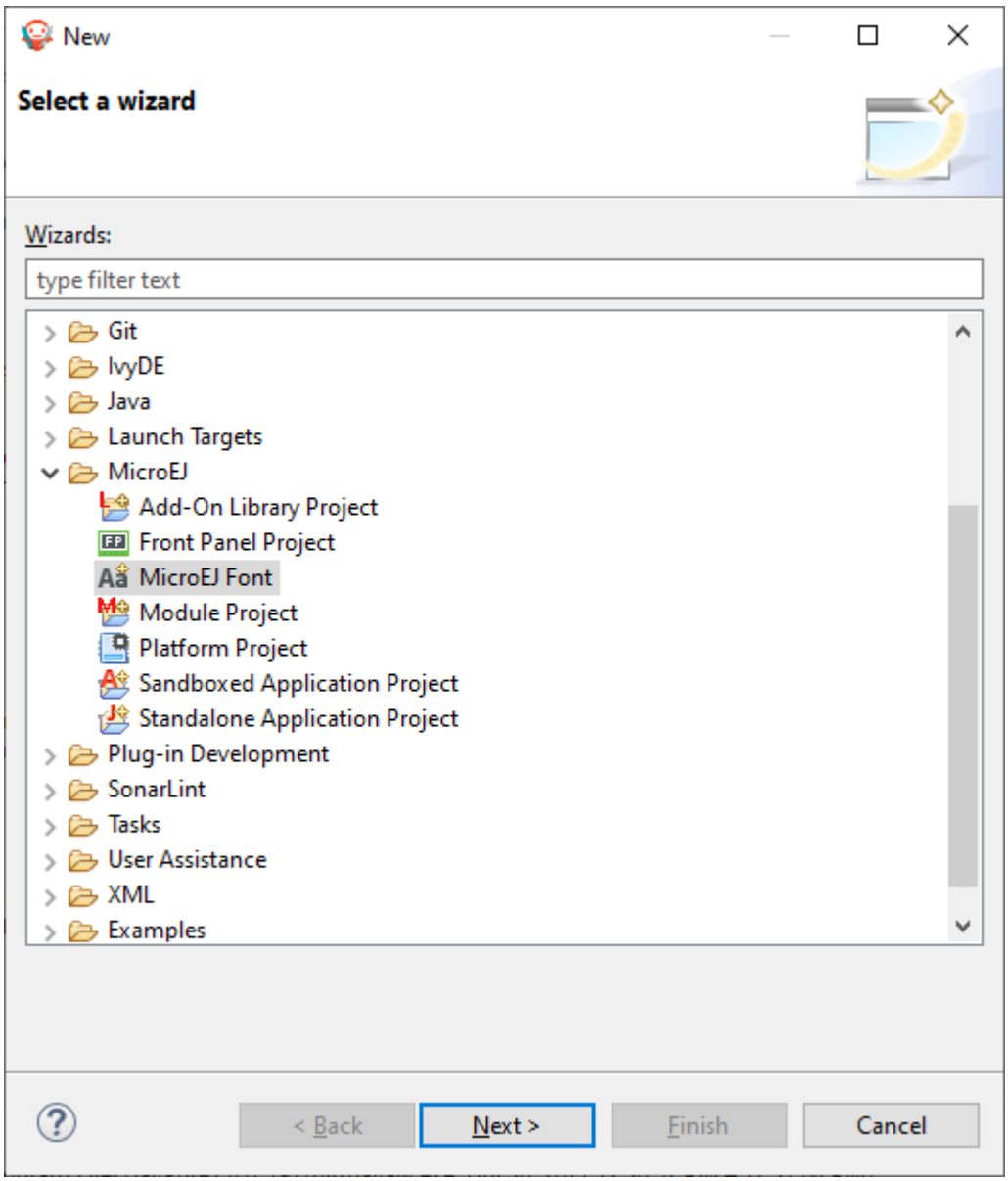
Next step: *Fonts*

9.13.11 Fonts

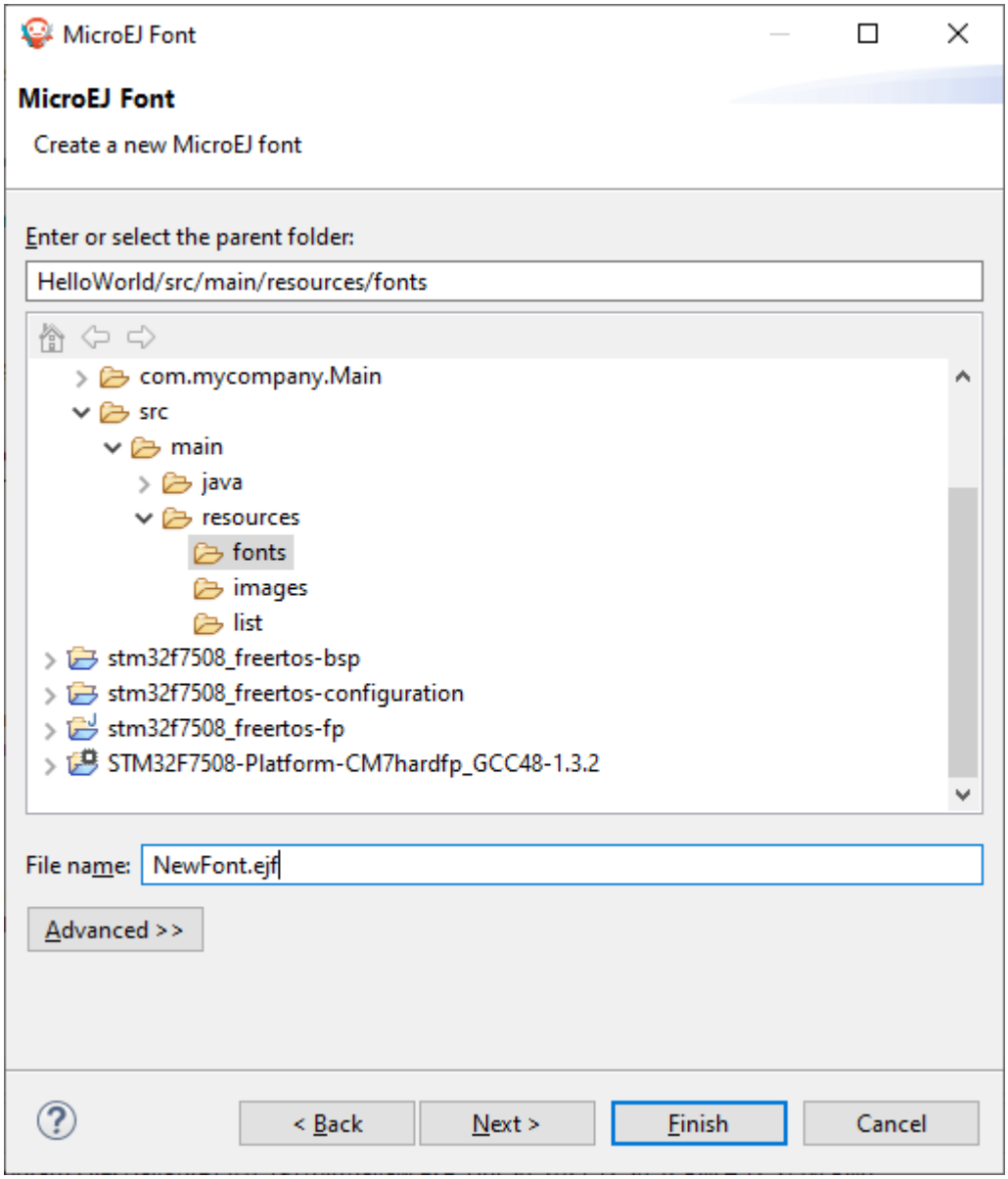
- Fonts are graphical resources that can be accessed with a call to ej.microui.display.Font.getFont(). To be displayed, these fonts have to be converted at build-time from their source format to the display raw format by the font generator tool.
- Fonts, just like images, must be declared in a `*.fonts.list`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

Creating a font

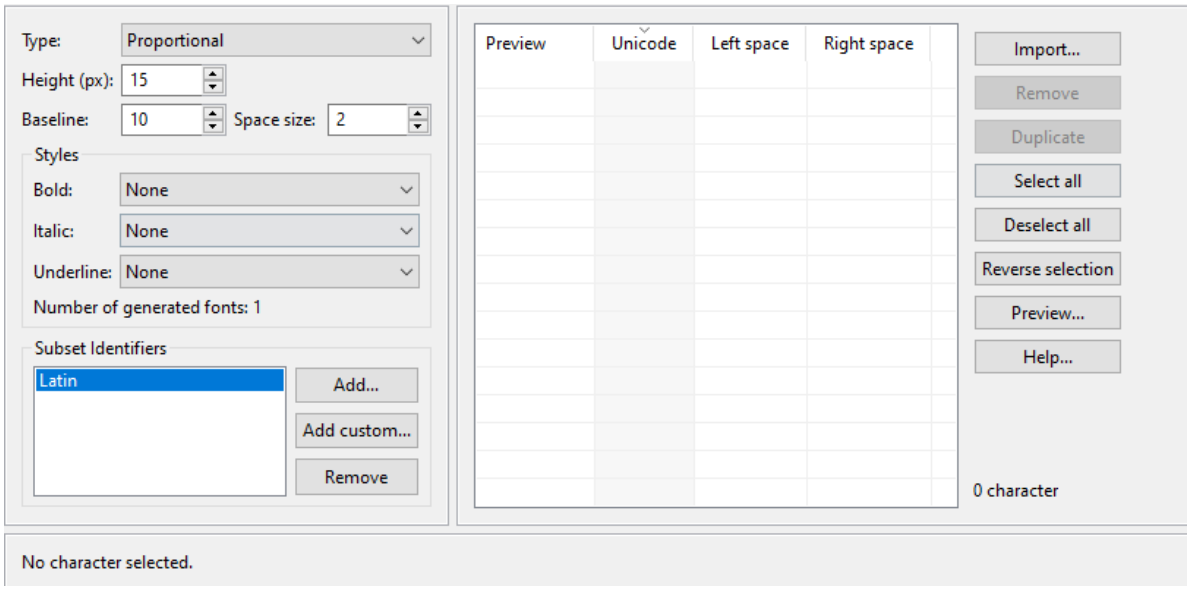
- To create a font, go to the package you want to store your fonts in, usually `Resources` > `fonts` .
- Then `Right-Click` > `New` > `Other` > `MicroEJ` > `MicroEJ Font` :



- Then, type the name of the font:

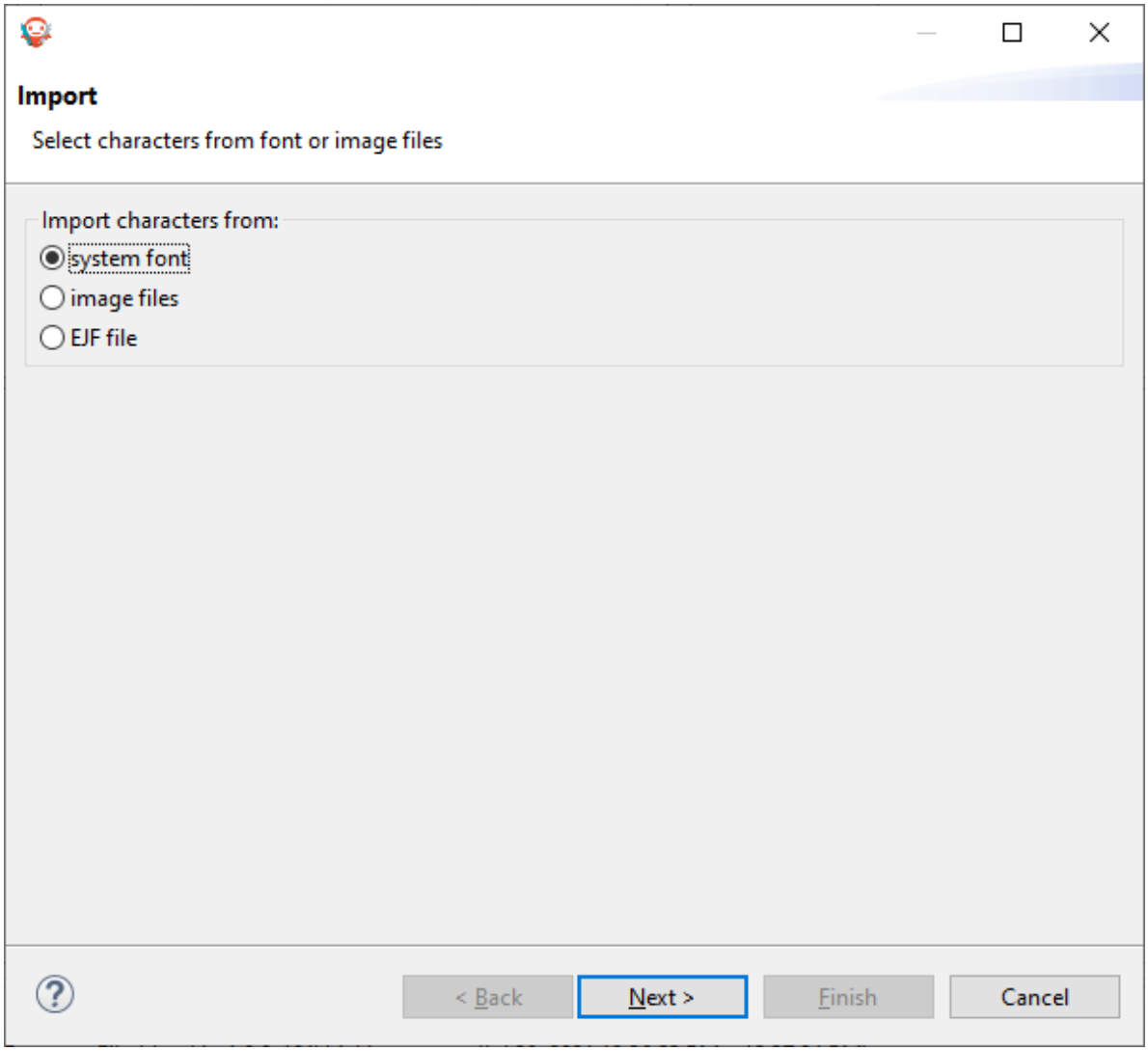


- Click on Finish , the following window opens:

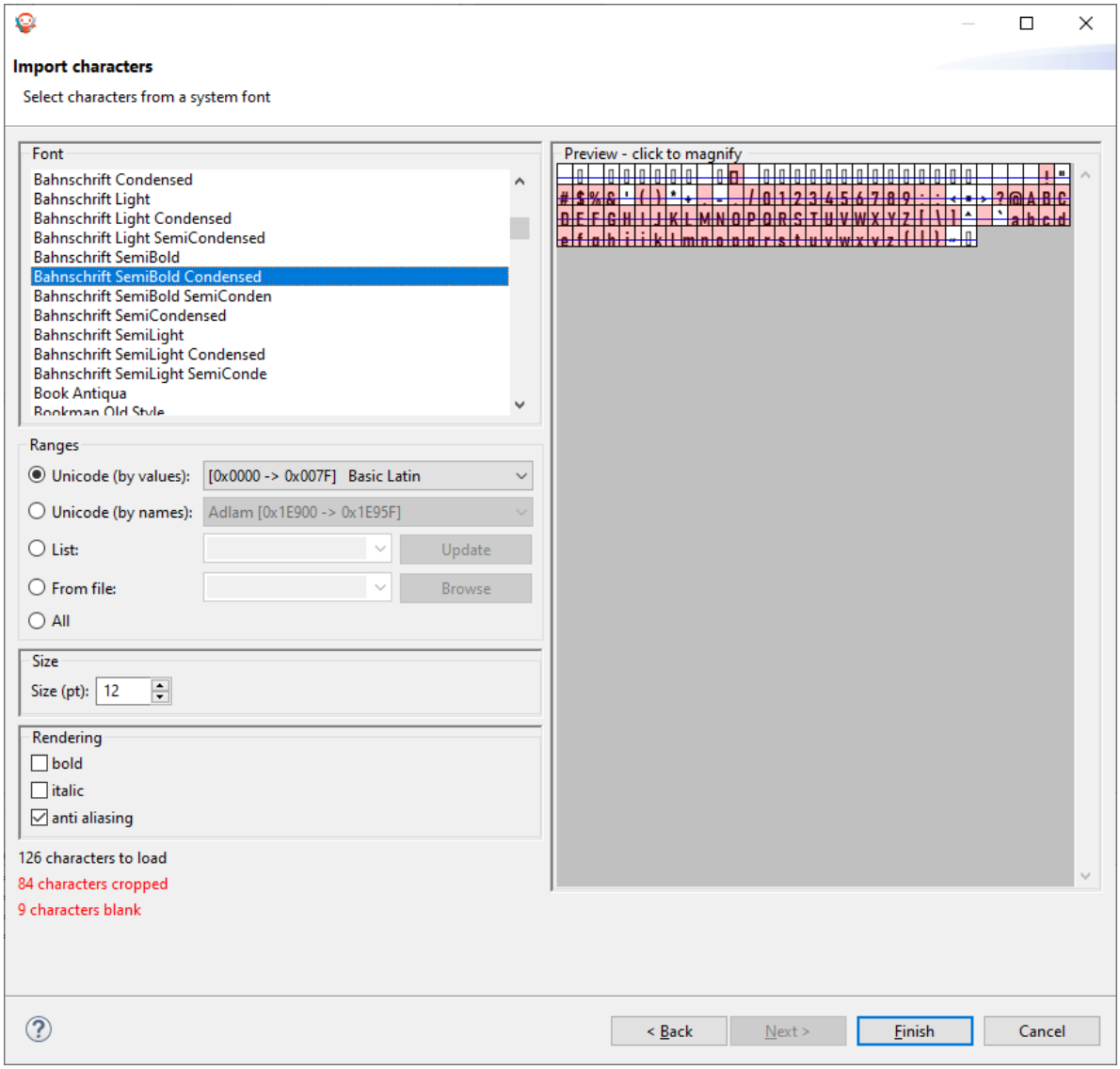


Note: It is important to have the font that you want already installed on your system.

- To import characters from a system font, click on Import… and the following opens:



- Click on Next and then select the font to use as shown below:



Note: If using a latin based alphabet, just leave the settings as they are and click on Finish , don't forget to adjust the height and baseline of the font.

- Click Finish and save the file. The font is imported in the .ejf file.
- Then just add the font to a `myapp.fonts.list` file in the `src/main/resources` source folder of your application:

```
/fonts/NewFont.ejf
```

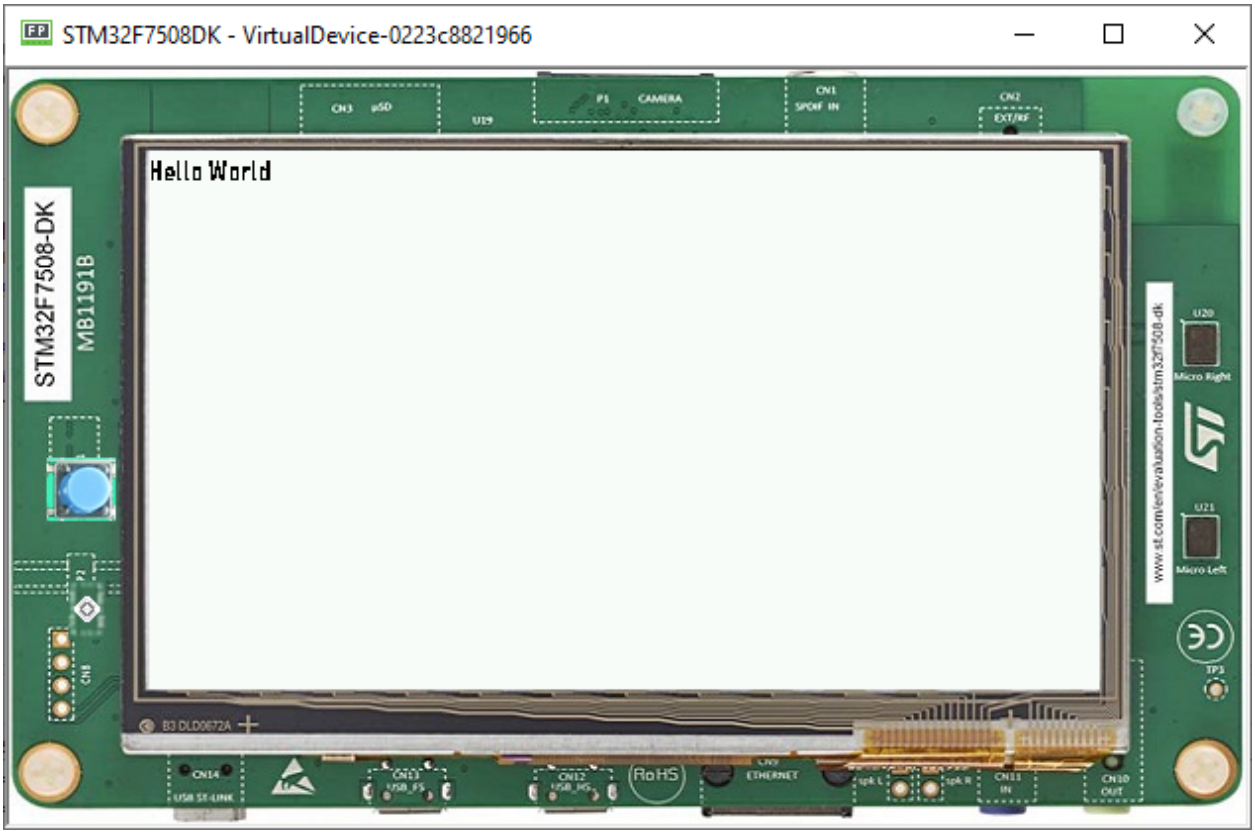
- More info in the *Fonts* section.

Adding the Font to a Label

To add the font, choose the font in the StyleSheet:

```
public static void main(String[] args) {
   MicroUI.start();
   Desktop desktop = new Desktop();
   Flow flow = new Flow(LayoutOrientation.VERTICAL);
   Label label = new Label("Hello World");
   Font font = Font.getFont("/fonts/NewFont.ejf");
   CascadingStylesheet css = new CascadingStylesheet();
   EditableStyle style = css.getSelectorStyle(new ClassSelector(BUTTON));
   flow.addChild(label);
   style.setFont(font);
   desktop.setStylesheet(css);
   desktop.setWidget(flow);
   desktop.requestShow();
}
```

Note: Don't forget to set the stylesheet to the desktop.



Next step: *Scroll List*

9.13.12 Scroll List

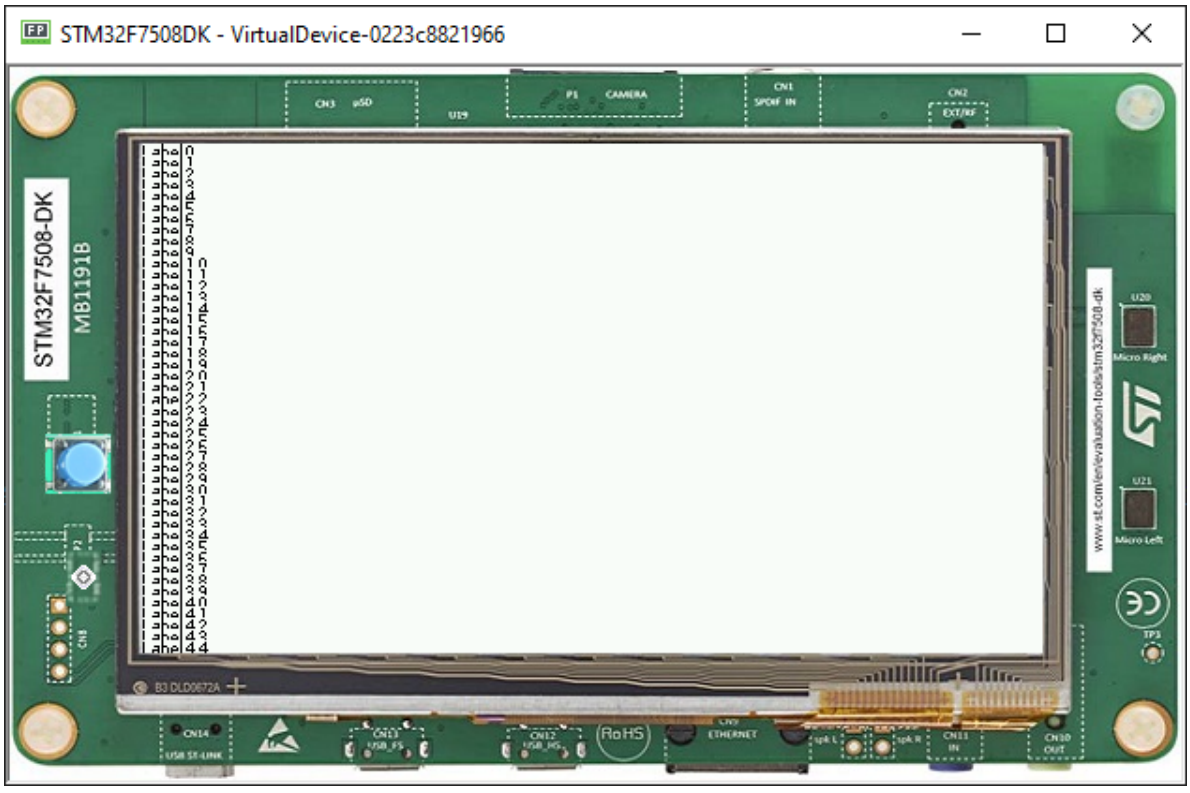
List

- A list is a container that lays out its children one after the other in a single column or row depending on its orientation.
- The size of each widget is set proportionally to its optimal size and the available size.
- Naturally, it shows some issues if the list contains too many components.
- Adding 45 items to a list shows the following result:

```
public static void main(String[] args) {
   MicroUI.start();
   Desktop desktop = new Desktop();

   List list = new List(LayoutOrientation.VERTICAL);
   for (int i = 0; i < 45; i++) {
      Label label = new Label("Label" + i);
      list.addChild(label);
   }

   desktop.setWidget(list);
   desktop.requestShow();
}
```

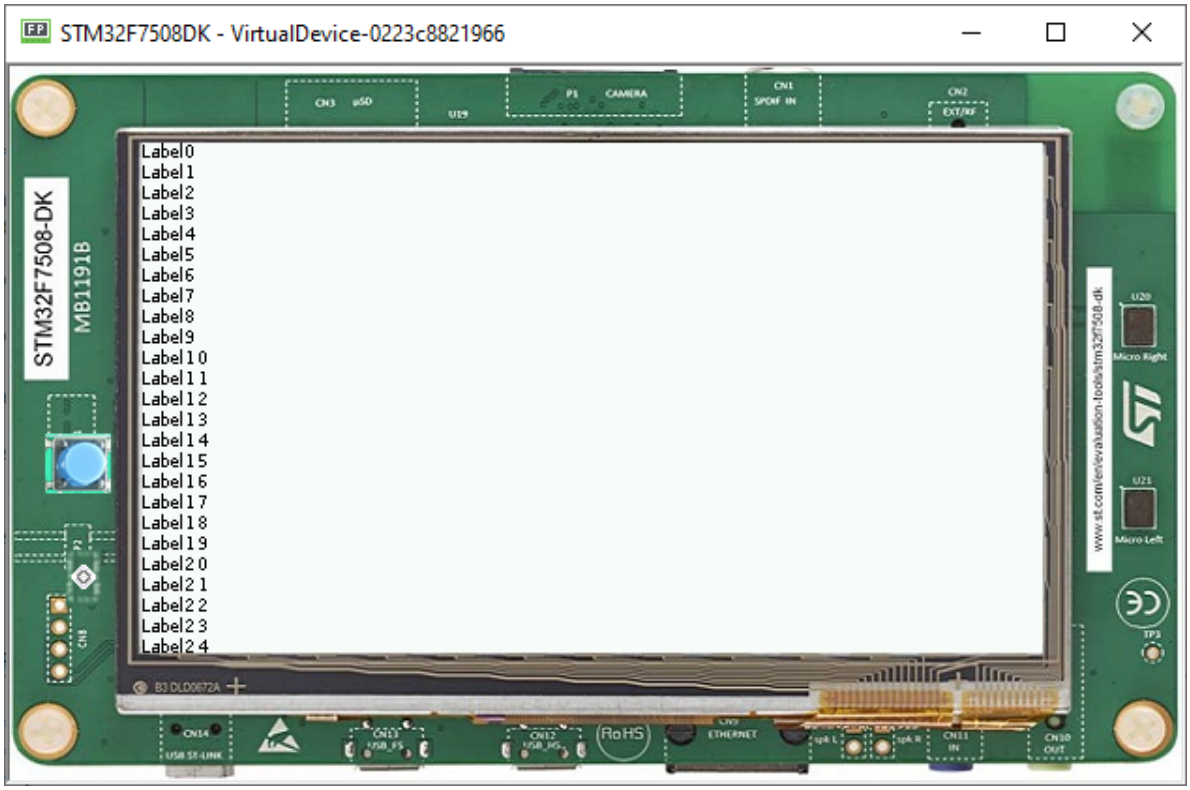


- To be able to see all the items, the list must be bigger than the display. So it needs to be included in another container that is able to scroll it.

Scrollable List

- A simple way to see all the items correctly and scroll the list is to include it in a `Scroll` container:

```
public static void main(String[] args) {
   MicroUI.start();
   Desktop desktop = new Desktop();
   List list = new List(LayoutOrientation.VERTICAL);
   for (int i = 0; i < 45; i++) {
      Label item = new Label("Label" + i);
      list.addChild(item);
   }
   CascadingStylesheet css = new CascadingStylesheet();
   Scroll scroll = new Scroll(LayoutOrientation.VERTICAL);
   scroll.setChild(list);
   desktop.setStylesheet(css);
   desktop.setWidget(scroll);
   desktop.requestShow();
}
```



- The list can be optimized for the scroll (to exclude the items that are outside the visible area). A scrollable list is available at Widget Demo.
- It can be copied in the project and replace the list:

```
public static void main(String[] args) {
   MicroUI.start();
   Desktop desktop = new Desktop();
   ScrollableList list = new ScrollableList(LayoutOrientation.VERTICAL);
```

(continues on next page)

(continued from previous page)

```
   for (int i = 0; i < 45; i++) {
      Label label = new Label("Label" + i);
      list.addChild(label);
   }
   CascadingStylesheet css = new CascadingStylesheet();
   Scroll scroll = new Scroll(LayoutOrientation.VERTICAL);
   scroll.setChild(list);
   desktop.setStylesheet(css);
   desktop.setWidget(scroll);
   desktop.requestShow();
}
```

Next step: *Creating a Contact List using Scroll List*

9.13.13 Creating a Contact List using Scroll List

Creating the Contact Widget

- As explained in *Creating Widgets*, it is possible to create our own widget by just extending the Widget class.
- First, let's create a constructor with all the things that we are going to need for this.

```
public ContactWidget(String contactName, Image image) {
   this.contactName = contactName;
   this.image = image;
}
```

- Then, overriding the two abstract methods of `Widget`

```
@Override
protected void computeContentOptimalSize(Size size) {
   Font f = getStyle().getFont();
   int height = Math.max(f.getHeight(), this.image.getHeight());
   int stringWidth = f.stringWidth(this.contactName);
   int width = stringWidth + this.image.getWidth();
   size.setSize(width, height);
}
```

```
@Override
protected void renderContent(GraphicsContext g, int contentWidth, int contentHeight) {
   g.setColor(Colors.BLACK);
   Painter.drawImage(g, this.image, 0, 0);
   int cornerEllipseSize = contentHeight / 2;
   int imageWidth = this.image.getWidth();
   int imageHeight = this.image.getHeight();
   Painter.drawRoundedRectangle(g, 0, 0, imageWidth, imageHeight, cornerEllipseSize,␣
↪cornerEllipseSize);
   Painter.drawString(g, this.contactName, getStyle().getFont(), imageWidth + 2,␣
↪contentHeight / 3);
}
```

- Then, simply replace the children in the List used in the last step:

```
for (int i = 0; i < 45; i++) {
   list.addChild(new ContactWidget("Label" + i, Image.getImage("/images/microej_logo.png
↪")));
}
```

- The class is as follows:

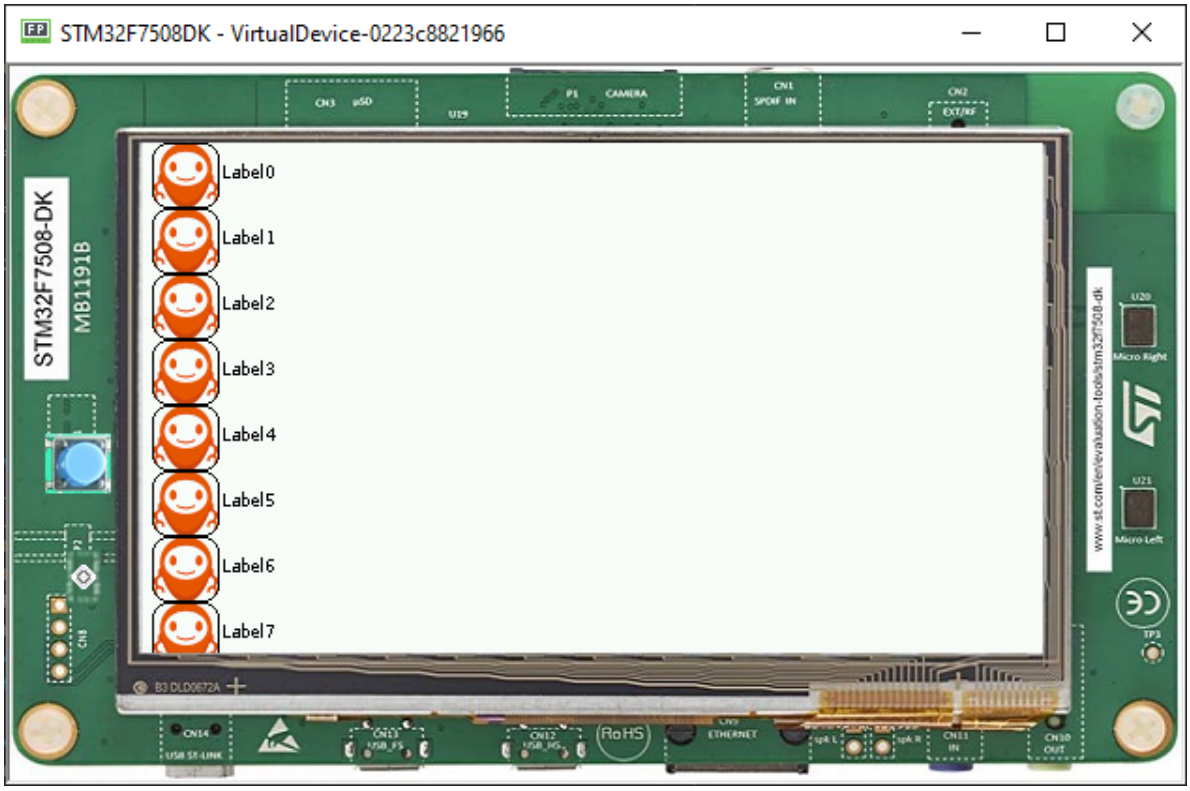
```
public class ContactWidget extends Widget {

   private String contactName;
   private Image image;

   public ContactWidget(String contactName, Image image) {
      this.contactName = contactName;
      this.image = image;
   }

   @Override
   protected void computeContentOptimalSize(Size size) {
      Font f = getStyle().getFont();
      int height = Math.max(f.getHeight(), this.image.getHeight());
      int stringWidth = f.stringWidth(this.contactName);
      int width = stringWidth + this.image.getWidth();
      size.setSize(width, height);
   }

   @Override
   protected void renderContent(GraphicsContext g, int contentWidth, int contentHeight)
↪{
      g.setColor(Colors.BLACK);
      Painter.drawImage(g, this.image, 0, 0);
      int cornerEllipseSize = contentHeight / 2;
      int imageWidth = this.image.getWidth();
      int imageHeight = this.image.getHeight();
      Painter.drawRoundedRectangle(g, 0, 0, imageWidth, imageHeight, cornerEllipseSize,␣
↪cornerEllipseSize);
      Painter.drawString(g, this.contactName, getStyle().getFont(), imageWidth + 2,␣
↪contentHeight / 3);
   }
}
```



Next step: *Internationalization*

9.13.14 Internationalization

Using PO Files

- PO files are a good way to handle Internationalization.
- Documentation is available here.
- In this example, let's create two PO files for two different languages(English and Portuguese) and add them to **resources/nls**.

  Labels_en_US.po:

```
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"
```

  Labels_pt_BR.po:

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

msgid "Label1"
msgstr "Minha label 1"

msgid "Label2"
msgstr "Minha 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 using a `myapp.nls.list` file.

Configuring NLS in MicroEJ

- First add those two dependencies to the module.ivy of your projet:

```
<dependency org="ej.library.runtime" name="nls" rev="4.0.0"/>
<dependency org="com.microej.library.runtime" name="binary-nls" rev="3.0.0"/>
```

- Then, let's create a `myapp.nls.list` file, and put it in the **src/main/resources/list** folder. The file looks like this:

```
com.mycompany.myapp.generated.Labels
```

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

- a Java interface with the given fully qualified name, containing a field for each `msgid` of the PO files.
- a NLS binary file containing the translations.

Usage

- Import the interface set in the `myapp.nls.list` file:

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

- Print the available locales:

```
for (String locale : Labels.NLS.getAvailableLocales()) {
    System.out.println(locale);
}
```

- Print the current locale:

```
System.out.println(Labels.NLS.getCurrentLocale());
```

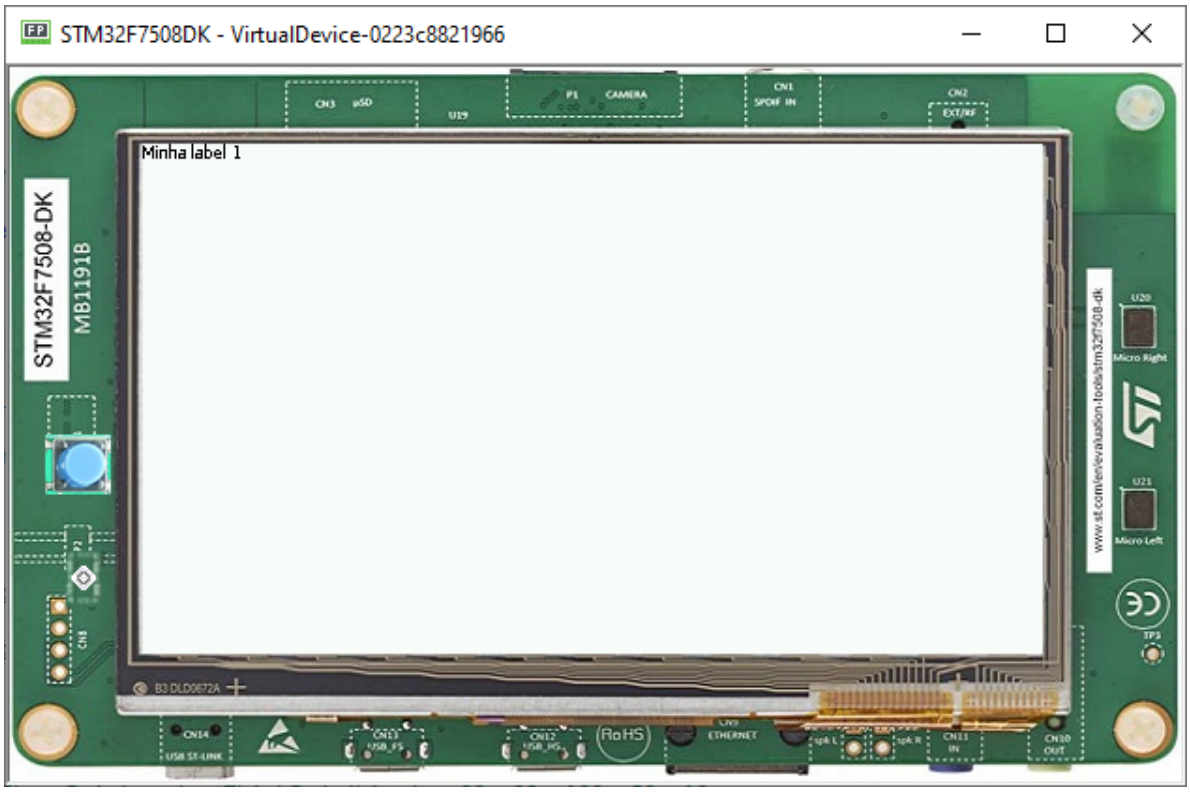
- Change the current locale:

```
Labels.NLS.setCurrentLocale("pt_BR");
```

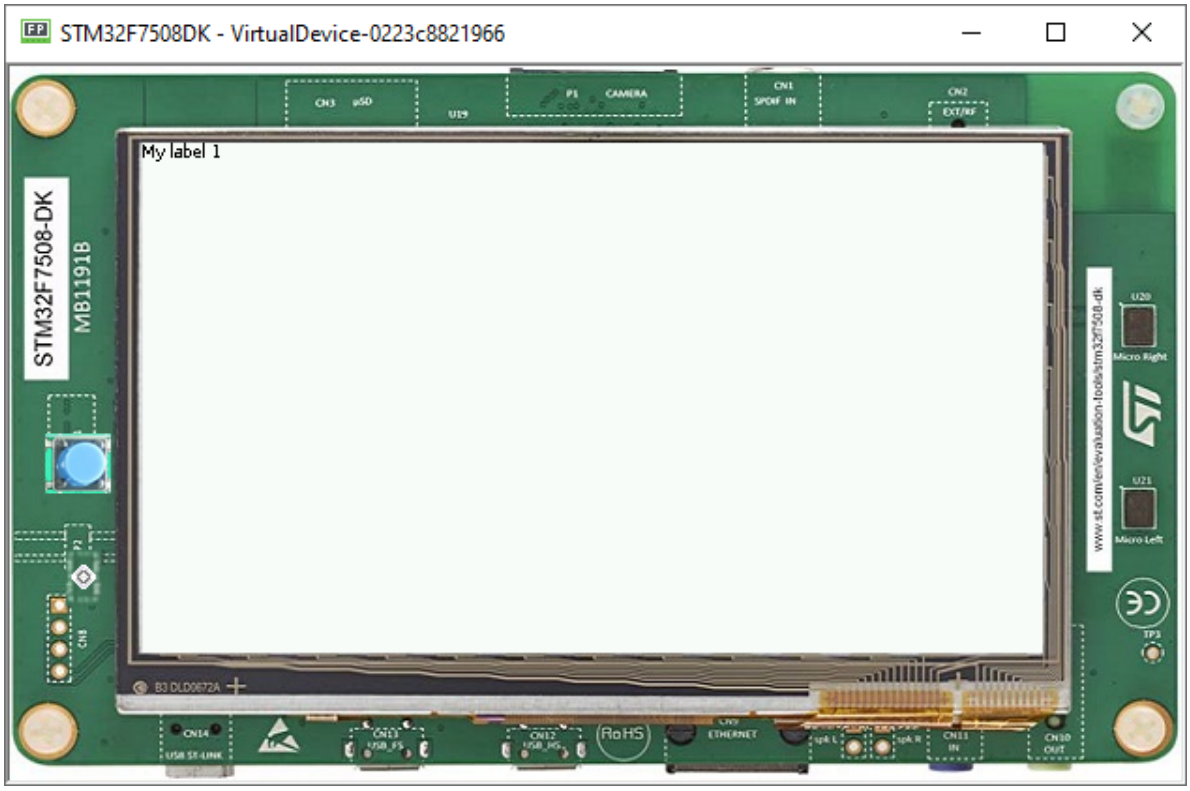
- Finally, put a message from NLS in a label. The code looks like this:

```
public static void main(String[] args) {
    MicroUI.start();
    Desktop desktop = new Desktop();
    Labels.NLS.setCurrentLocale("pt_BR");
    // For english locale uncomment the line below and comment the pt_BR locale setter␣
↪call.
    // Labels.NLS.setCurrentLocale("en_US");
    Label label = new Label(Labels.NLS.getMessage(Labels.Label1));
    desktop.setWidget(label);
    desktop.requestShow();
}
```

- The result looks like this:



- Setting the locale to “en_US” the result is as follows:



Loading Translations as an External Resource

When building the Application or running it on Simulator, the Resource Buffer Generator is executed.

A resource containing translations is generated. This resource can be loaded as external resource in order to be loaded from an external memory (e.g. from a FileSystem).

Note: This mode requires to setup the *External Resources Loader* in the VEE Port.

Follow the steps below to declare translations as external resources:

- Add a `myapp.nls.externresources.list` file in the **src/main/resources/list** folder,
- Add the following path inside the file:

```
/com/mycompany/myapp/generated/Labels.nls
```

  This path can be found in `src-adpgenerated/binarynls/java/com/mycompany/myapp/generated/Labels.nls.resources.list`
- Build the application for the target,
- Open the `SOAR.map` file to check that the resource is not embedded anymore in the application binary. The `xxx_Labels.nls` line should not appear anymore in the `ApplicationResources` section.
- The resource containing translations is now located in the `com.mycompany.myapp.Main/externalResources` folder. This resource must be embedded on the target and loaded using the External Resources Loader.

A simple implementation of the External Resources Loader is available on GitHub: Example-ExternalResourceLoader.

9.14 Discover Embedded Debugging Techniques

This tutorial describes the available tools provided to developers to debug an application. It also presents debugging methods applied to two concrete uses cases (GUI application freeze and HardFault).

1. *Debugging Tools*
2. *Use Case 1: Debugging a GUI Application Freeze*
3. *Use Case 2: Debugging a HardFault*

9.14.1 Intended Audience

The audience for this document is engineers who want to learn about the tools available to debug embedded applications.

In addition, the *Use Case 1: Debugging a GUI Application Freeze* is particularly relevant for Application engineers. Whereas the *Use Case 2: Debugging a HardFault* is more relevant for Firmware engineers.

9.14.2 Debugging Tools

This section presents an overview of the main tools available to debug an embedded application. Please refer to the developer guides for the complete reference (*Application Developer Guide*, *VEE Porting Guide*, *Kernel Developer Guide*).

- *Events Tracing and Logging*
- *Runtime State Dump*
- *Memory Inspection*
- *Platform Qualification*
- *Simulator Debugger*
- *Static Analysis Tools*
- *GUI Debugging Tools*

Events Tracing and Logging

When an application has issues, the first step is to understand what is happening inside the system.

- The Trace Library is a real-time event recording library. Use it to trace the beginning and the ending of events.

```
private static final int EVENT_ID = 0;

public static void switchState(ApplicationState newState) {
   tracer.recordEvent(EVENT_ID);

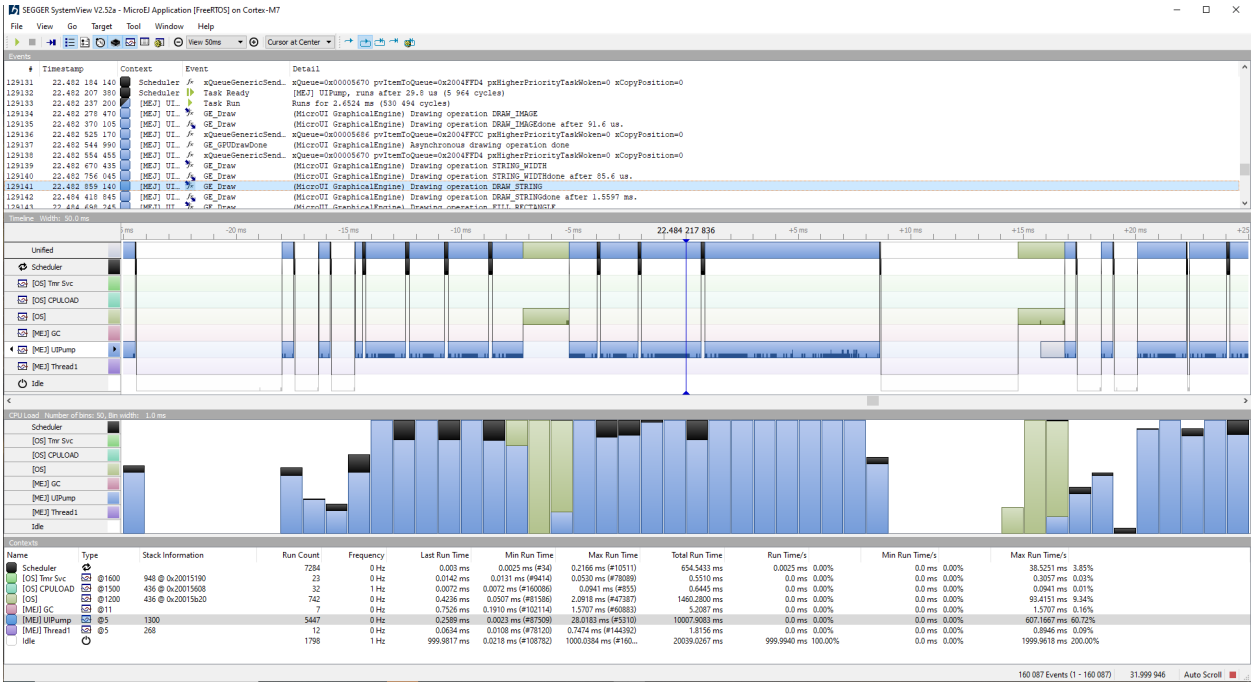
   previousState = currentState;
   currentState = newState;
```

(continues on next page)

(continued from previous page)

```
    tracer.recordEventEnd(EVENT_ID);
}
```

This API is most useful with the *SystemView Event tracer* to visualize the timeline of events.



- The Message Library is a small RAM/ROM/CPU footprint API to log errors, warnings, and misc information.

```
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);
}
```

- The Logging Library implements a subset of the standard Java `java.util.logging`.

```
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 " +␣
```

(continues on next page)

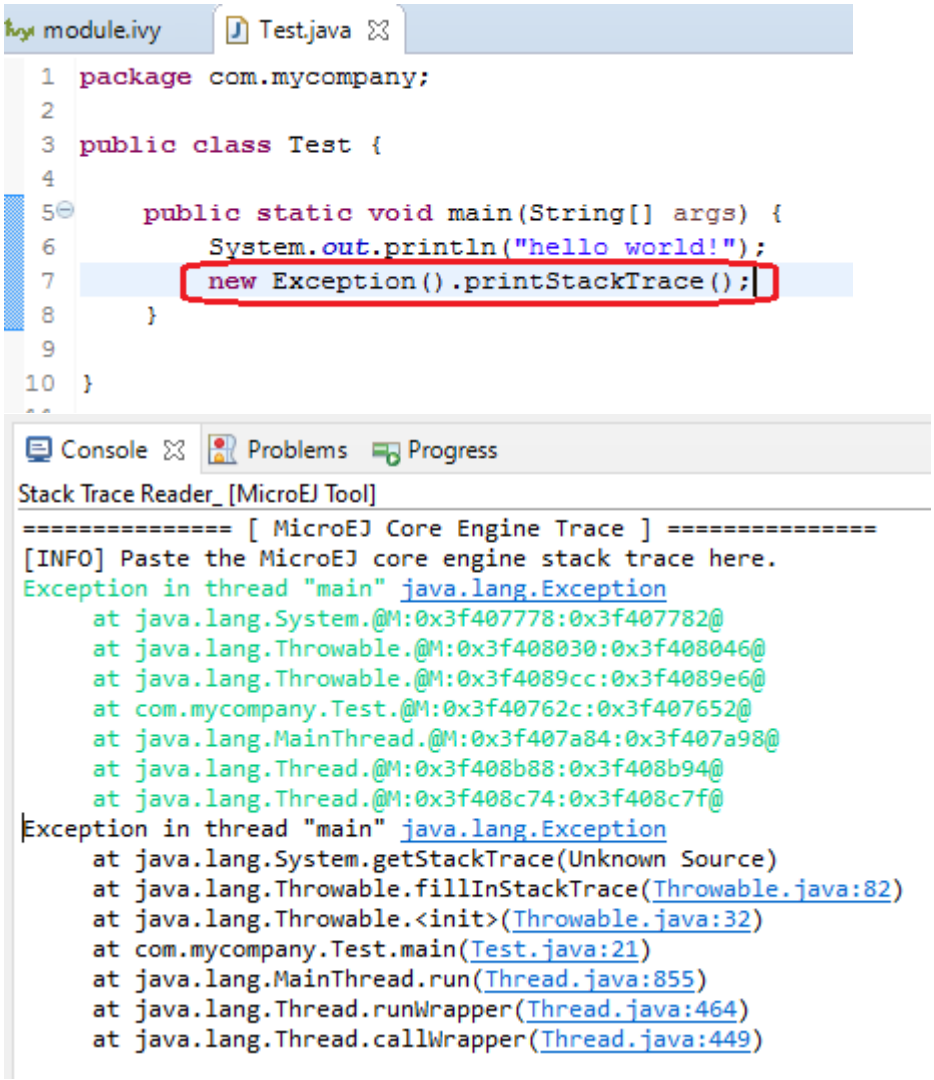
(continued from previous page)

```
↪previousState.toString() + " to "
        + currentState.toString() + ".");
}
```

Please refer to the tutorial *Instrument Java Code for Logging* for a comparison of these libraries.

Runtime State Dump

- Output information on the standard output System.out and use the *Stack Trace Reader* to read and decode the MicroEJ stack traces.



- The *Core Engine VM dump* is a low-level API to display the state of the MicroEJ Runtime and the MicroEJ threads (name, priority, stack trace, etc.)

```
================================== VM Dump ==================================
Java threads count: 3
Peak java threads count: 3
```

(continues on next page)

(continued from previous page)

```
Total created java threads: 3
Last executed native function: 0x90035E3D
Last executed external hook function: 0x00000000
State: running
--------------------------------------------------------------------------------
Java Thread[1026]
name="main" prio=5 state=RUNNING max_java_stack=456 current_java_stack=184

java.lang.MainThread@0xC0083C7C:
    at (native) [0x90003F65]
    at com.microej.demo.widget.main.MainPage.getContentWidget(MainPage.java:95)
        Object References:
            - com.microej.demo.widget.main.MainPage@0xC00834E0
            - com.microej.demo.widget.main.MainPage$1@0xC0082184
            - java.lang.Thread@0xC0082194
            - java.lang.Thread@0xC0082194
    at com.microej.demo.widget.common.Navigation.createRootWidget(Navigation.
↪java:104)
        Object References:
            - com.microej.demo.widget.main.MainPage@0xC00834E0
    at com.microej.demo.widget.common.Navigation.createDesktop(Navigation.
↪java:88)
        Object References:
            - com.microej.demo.widget.main.MainPage@0xC00834E0
            - ej.mwt.stylesheet.CachedStylesheet@0xC00821DC
    at com.microej.demo.widget.common.Navigation.main(Navigation.java:40)
        Object References:
            - com.microej.demo.widget.main.MainPage@0xC00834E0
    at java.lang.MainThread.run(Thread.java:855)
        Object References:
            - java.lang.MainThread@0xC0083C7C
    at java.lang.Thread.runWrapper(Thread.java:464)
        Object References:
            - java.lang.MainThread@0xC0083C7C
    at java.lang.Thread.callWrapper(Thread.java:449)
--------------------------------------------------------------------------------
Java Thread[1281]
name="UIPump" prio=5 state=WAITING timeout(ms)=INF max_java_stack=120 current_
↪java_stack=117
external event: status=waiting

java.lang.Thread@0xC0083628:
    at ej.microui.MicroUIPump.read(Unknown Source)
        Object References:
            - ej.microui.display.DisplayPump@0xC0083640
    at ej.microui.MicroUIPump.run(MicroUIPump.java:176)
        Object References:
            - ej.microui.display.DisplayPump@0xC0083640
    at java.lang.Thread.run(Thread.java:311)
        Object References:
            - java.lang.Thread@0xC0083628
    at java.lang.Thread.runWrapper(Thread.java:464)
```

(continues on next page)

(continued from previous page)

```
        Object References:
            - java.lang.Thread@0xC0083628
    at java.lang.Thread.callWrapper(Thread.java:449)
--------------------------------------------------------------------------------
Java Thread[1536]
name="Thread1" prio=5 state=READY max_java_stack=60 current_java_stack=57

java.lang.Thread@0xC0082194:
    at java.lang.Thread.runWrapper(Unknown Source)
        Object References:
            - java.lang.Thread@0xC0082194
    at java.lang.Thread.callWrapper(Thread.java:449)
================================================================================

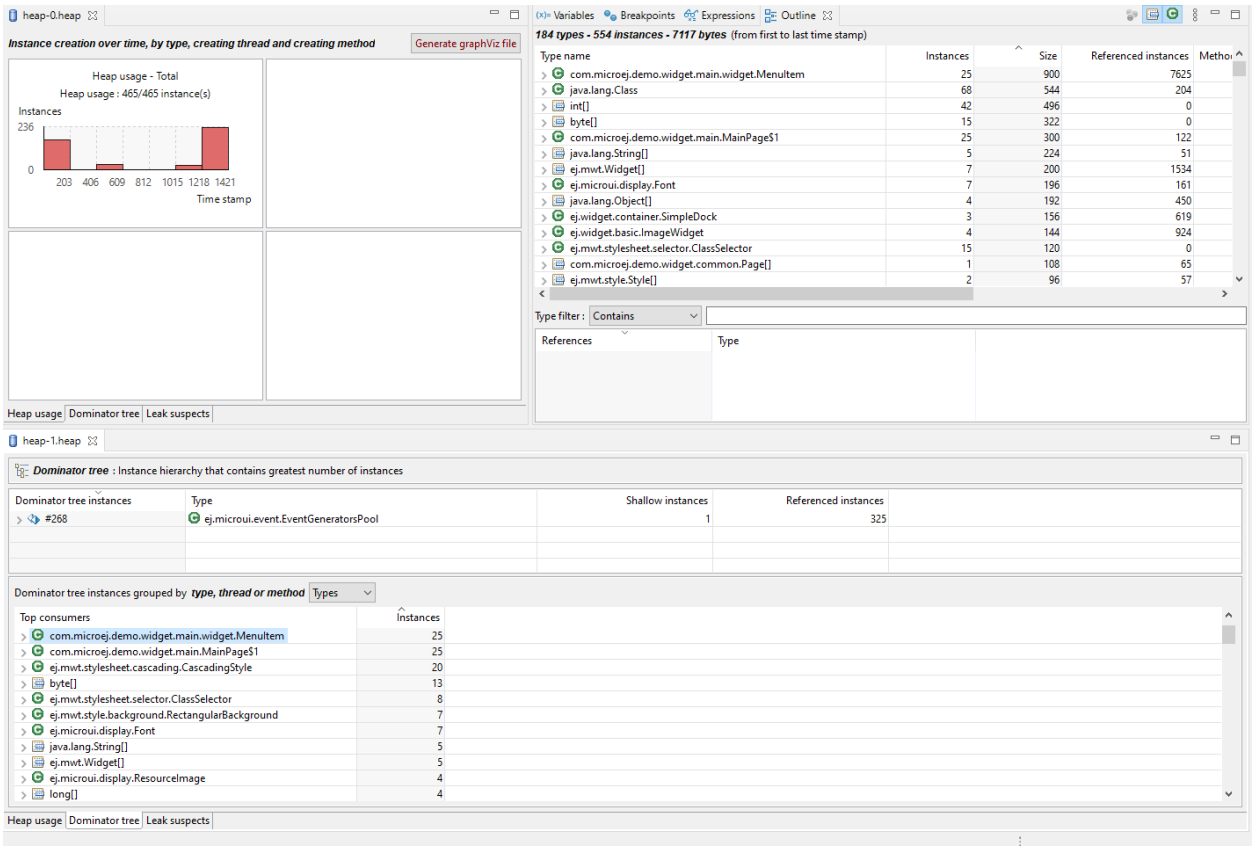
============================== Garbage Collector ===============================
State: Stopped
Last analyzed object: null
Total memory: 15500
Current allocated memory: 7068
Current free memory: 8432
Allocated memory after last GC: 0
Free memory after last GC: 15500
================================================================================

============================== Native Resources ================================
Id         CloseFunc  Owner            Description
--------------------------------------------------------------------------------
================================================================================
```

Memory Inspection

Memory issues such as memory corruption and memory leaks can be hard to troubleshoot. However, the following tools are available to address these issues:

- Check the internal structure integrity of the MicroJvm virtual machine with the *LLMJVM_checkIntegrity API* to detect memory corruptions in native functions.
- Use the *Heap Usage Monitoring Tool* to estimate the heap requirements of an application.
- The *Heap Dumper & Heap Analyzer* tools analyze the content of the heap. They are helpful to detect memory leaks and look for optimization of the heap usage.



Platform Qualification

The Platform Qualification Tools (PQT) project provides the tools required to validate each component of a MicroEJ Platform. After porting or adding a feature to a MicroEJ Platform, it is necessary to validate its integration.

The project is available on GitHub: https://github.com/MicroEJ/VEEPortQualificationTools

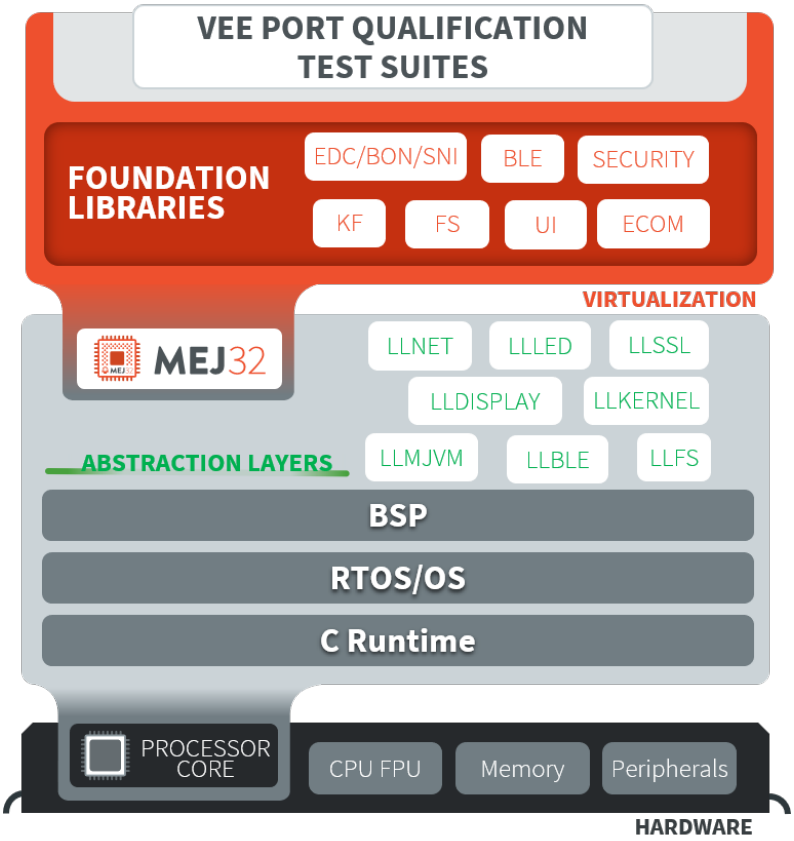
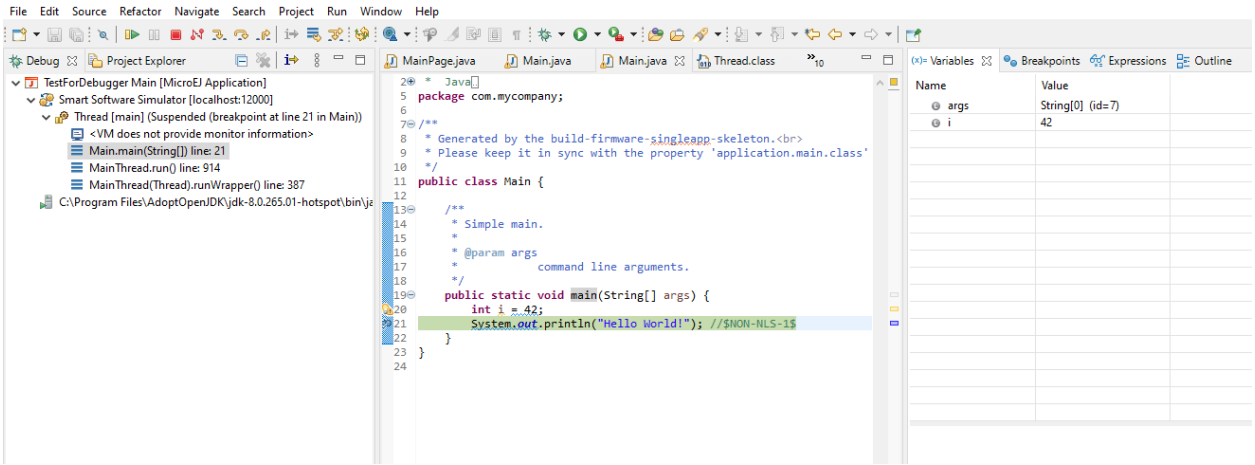


Fig. 2: Platform Qualification Overview

Please refer to the *VEE Port Qualification* documentation for more information.

Simulator Debugger

- *Debug an Application* on Simulator, add breakpoints, inspect stack frame, use step-by-step, etc.

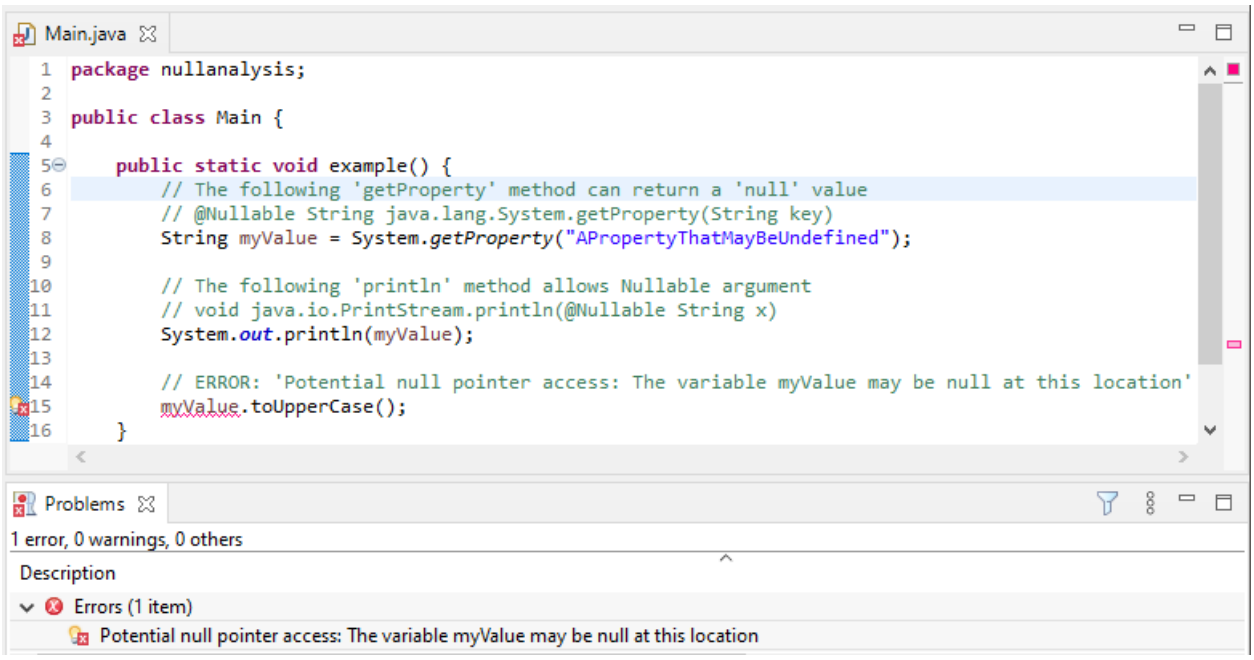


- Configure the libraries' sources location to *View library as sources* in the debugger.

Static Analysis Tools

Static analysis tools are helpful allies to prevent several classes of bugs.

- *SonarQube™* provides reports on duplicated code, coding standards, unit tests, code coverage, code complexity, potential bugs, comments, and architecture.
- Use the *Null Analysis tool* to detect and prevent NullPointerException, one of the most common causes of runtime failure of Java programs.



GUI Debugging Tools

- The Widget Library provides several *Debug Utilities* to investigate and troubleshoot GUI applications. For example, it is possible to print the type and bounds of each widget in the hierarchy of a widget:

```
Scroll: 0,0 480x272 (absolute: 0,0)
+--ScrollableList: 0,0 480x272 (absolute: 0,0)
|  +--Label: 0,0 480x50 (absolute: 0,0)
|  +--Dock: 0,50 480x50 (absolute: 0,50)
|  |  +--ImageWidget: 0,0 70x50 (absolute: 0,50)
|  |  +--Label: 70,0 202x50 (absolute: 70,50)
|  +--Label: 0,100 480x50 (absolute: 0,100)
```

- *MicroUI Event Buffer* provides an API to store and dump the events received:

```
============================== MicroUI FIFO Dump ===============================
--------------------------------- Old Events ---------------------------------
```

(continues on next page)

(continued from previous page)

```
[27: 0x00000000] garbage
[28: 0x00000000] garbage
[...]
[99: 0x00000000] garbage
[00: 0x08000000] Display SHOW Displayable (Displayable index = 0)
[01: 0x00000008] Command HELP (event generator 0)
[02: 0x0d000000] Display REPAINT Displayable (Displayable index = 0)
[03: 0x07030000] Input event: Pointer pressed (event generator 3)
[04: 0x009f0063]    at 159,99 (absolute)
[05: 0x07030600] Input event: Pointer moved (event generator 3)
[06: 0x00aa0064]    at 170,100 (absolute)
[07: 0x02030700] Pointer dragged (event generator 3)
[08: 0x0d000000] Display REPAINT Displayable (Displayable index = 0)
[09: 0x07030600] Input event: Pointer moved (event generator 3)
[10: 0x00b30066]    at 179,102 (absolute)
[11: 0x02030700] Pointer dragged (event generator 3)
[12: 0x0d000000] Display REPAINT Displayable (Displayable index = 0)
[13: 0x07030600] Input event: Pointer moved (event generator 3)
[14: 0x00c50067]    at 197,103 (absolute)
[15: 0x02030700] Pointer dragged (event generator 3)
[16: 0x0d000000] Display REPAINT Displayable (Displayable index = 0)
[17: 0x07030600] Input event: Pointer moved (event generator 3)
[18: 0x00d00066]    at 208,102 (absolute)
[19: 0x02030700] Pointer dragged (event generator 3)
[20: 0x0d000000] Display REPAINT Displayable (Displayable index = 0)
[21: 0x07030100] Input event: Pointer released (event generator 3)
[22: 0x00000000]    at 0,0 (absolute)
[23: 0x00000008] Command HELP (event generator 0)
---------------------------------- New Events -----------------------------------
[24: 0x0d000000] Display REPAINT Displayable (Displayable index = 0)
[25: 0x07030000] Input event: Pointer pressed (event generator 3)
[26: 0x002a0029]    at 42,41 (absolute)
-------------------------- New Events' Java objects --------------------------
[java/lang/Object[2]@0xC000FD1C
 [0] com/microej/examples/microui/mvc/MVCDisplayable@0xC000BAC0
 [1] null
================================================================================
```

- MicroUI can log several actions, which can be viewed in SystemView. Please refer to *Debug Traces* for more information.

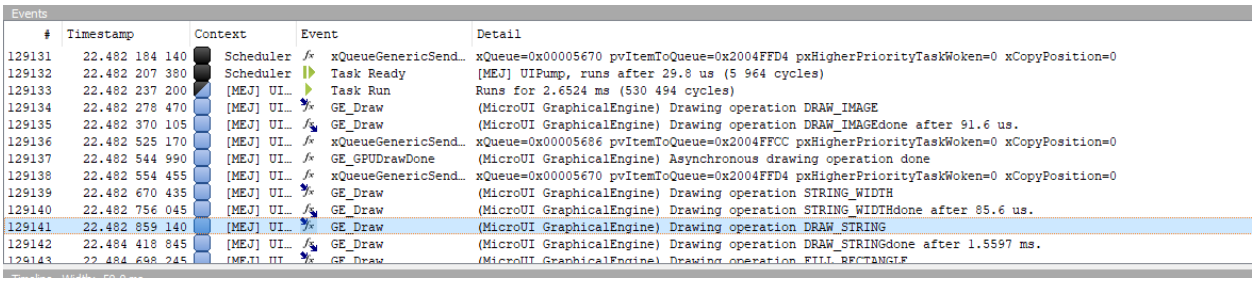


Fig. 3: MicroUI Traces displayed in SystemView

- Make sure to understand *MWT Concepts*, especially the relations between the rendering, the lay-out the event dispatch and the states of desktop and widget.
- For UI2 and former versions, please refer to MicroUI and multithreading for a description of the threading model.

9.14.3 Use Case 1: Debugging a GUI Application Freeze

When an application User Interface freezes and becomes unresponsive, in most cases, one of the following conditions applies:

- An unrecoverable system failure occurred, like a HardFault, and the RTOS tasks are not scheduled anymore.
- The RTOS task that runs the Core Engine is never given CPU time (suspended or blocked).
- The RTOS task that runs the Core Engine is executing never-ending native code (infinite loop in native implementation for example).
- A Java method executes a long-running operation in the MicroUI thread (also called Display Pump thread).
- The application code is unable to receive or process user input events.

The following sections explain how to instrument the code to locate the issue when the UI freeze occurs. The steps followed are:

1. Check if the RTOS properly schedules the Core Engine task.
2. Check if the Core Engine properly schedules all Java threads.
3. Check if the Core Engine properly schedules the MicroUI thread.
4. Check if Input Events are properly processed.

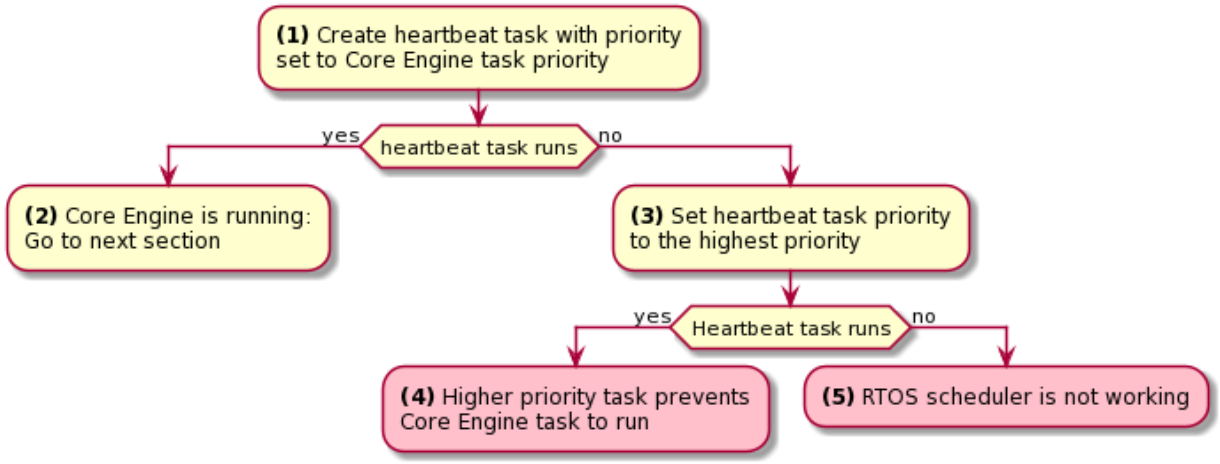
Note:

- The checks of the schedulers are possible with *SystemView* and *MicroUI Debug Traces*.
- The Input Events check is possible with the *LLUI_INPUT_dump API*.

Check RTOS Tasks Scheduling

Let's start at low level by figuring out if the RTOS is scheduling tasks correctly. If possible, use a debugger or *SystemView*; if not, use the heartbeat task described below.

The following flow chart summarizes the investigation steps with a heartbeat task:



(1) Make one of the RTOS tasks acts like a heartbeat: create a dedicated task and make it report in some way at a regular pace (print a message on standard output, blink an LED, use SystemView, etc.). Set the heartbeat task priority to the same priority as the Core Engine task.

(2) In this configuration, if the heartbeat is still running when the UI freeze occurs, we can go a step further and check whether the Core Engine is still scheduling Java threads or not. See next section *Check Java Threads Scheduling*.

(3) If the heartbeat doesn't run when the UI freeze occurs, set the heartbeat task priority to the maximum priority.

> **Warning:** Some RTOS use a task to schedule the RTOS timers. The heartbeat task priority must be lower than the RTOS timers priority.

(4) In this configuration, if the heartbeat is still running when the UI freeze occurs, then an RTOS task with a priority higher than the Core Engine task keeps using the CPU. Use the RTOS specific tools to identify what is the faulty task.

(5) If the heartbeat doesn't run when the UI freeze occurs, then the RTOS scheduler is not scheduling anything. This can be caused by an RTOS timer task or an interrupt handler that never returns, or a crash of the RTOS scheduler.

Check Java Threads Scheduling

As a reminder, the threading model implemented by Core Engine is called green thread: it defines a multi-threaded environment without relying on any native RTOS capabilities. Therefore, all Java threads run in a single RTOS task. For more details, please refer to the *MicroEJ Core Engine* section. A quick way to check if the Java threads are scheduled correctly is, here again, to make one of the threads print a heartbeat message. Copy/paste the following snippet in the `main()` method of the application:

```
TimerTask task = new TimerTask() {

    @Override
    public void run() {
        System.out.println("Alive");
    }
};
Timer timer = new Timer();
timer.schedule(task, 10_000, 10_000);
```

This code creates a new Java thread that will print the message `Alive` on the standard output every 10 seconds.

Assuming no one canceled the `Timer`, if the `Alive` printouts stop when the UI freeze occurs, then it can mean that:

- The Core Engine stopped scheduling the Java threads.
- Or that one or more threads with a higher priority prevent the threads with a lower priority from running.

Here are a few suggestions:

- Ensure no Java threads with a high priority prevent the scheduling of the other Java threads. For example, convert the above example with a dedicated thread with the highest priority:

```
Thread thread = new Thread(new Runnable() {

   @Override
   public void run() {
           while (true) {
                   try {
                           Thread.sleep(10_000);
                           System.out.println("Alive");
                   } catch (InterruptedException e) {
                           e.printStackTrace();
                   }
           }
   }
});
thread.setPriority(Thread.MAX_PRIORITY);
thread.start();
```

- The RTOS task that runs the Core Engine might be suspended or blocked. Check if some API call is suspending the task or if a shared resource could be blocking it.
- When a Java native method is called, it calls its C counterpart function in the RTOS task that runs the Core Engine. While the C function is running, no other Java methods can run because the Core Engine waits for the C function to finish. Consequently, no Java thread can ever run again if the C function never returns. Therefore, spot any suspect native functions and trace every entry/exit to detect faulty code.

Please refer to *Implementation Details* if you encounter issues when implementing the heartbeat.

Check UI Thread Liveness

Now, what if the `Alive` heartbeat runs while the UI is frozen? Java threads are getting scheduled, but the UI thread (also called Display Pump thread) does not process display events.

Let's make the heartbeat snippet above execute in the UI thread. Simply wraps the `System.out.println("Alive")` with a callSerially():

```
TimerTask task = new TimerTask() {

    @Override
    public void run() {
        System.out.println("TimerTask Alive");
        MicroUI.callSerially(new Runnable() {

            @Override
            public void run() {
```

(continues on next page)

(continued from previous page)

```
                System.out.println("UI Alive");
            }
        });
    }

    @Override
    public void uncaughtException(Timer timer, Throwable e) {
        // Default implementation of this method would cancel the task.
        // Let's just ignore uncaught exceptions for debug purposes.
        e.printStackTrace();
    }
};
Timer timer = new Timer();
timer.schedule(task, 10_000, 10_000);
```

In case this snippet prints `TimerTask Alive` but not `UI alive` when the freeze occurs, then there are a few options:

- The application might be processing a long operation in the UI thread, for example:
  - infinite/indeterminate loops
  - network/database access
  - heavy computations
  - Thread.sleep()/Object.wait()
  - `SNI_suspendCurrentJavaThread()` in native call

  When doing so, any other UI-related operation will not be processed until completion, leading the display to be unresponsive. Any code that runs in the UI thread might be responsible. Look for code executed as a result of calls to:
  - `repaint()` : code in renderContent()
  - `revalidate()` / `revalidateSubTree()` : code in `validateContent()` and `setBoundsContent()`
  - handleEvent()
  - callSerially(): code wrapped in such calls will be executed in the UI thread
- The UI thread has terminated.

As a general rule, avoid running extended operations in the UI thread, follow the general pattern and use a dedicated thread/executor instead:

```
ExecutorService executorService = ServiceLoaderFactory.getServiceLoader().
↪getService(ExecutorService.class, SingleThreadExecutor.class);
executorService.execute(new Runnable() {

    @Override
    public void run() {

        // (... long non-UI operation ...)

        // optional: update the UI upon completion
        Display.getDefaultDisplay().callSerially(new Runnable() {
```

(continues on next page)

(continued from previous page)

```
        @Override
        public void run() {
                    // update display code (will be executed in UI thread)
        }
    });
    }
});
```

Check Input Events Processing

Another case worth looking at is whether the application is processing user input events as it should. The UI may look "frozen" only because it doesn't react to input events. Replace the desktop instance with the one below to log all user inputs.

```
Desktop desktop = new Desktop() {

    @Override
    public EventHandler getController() {
        EventHandler controller = super.getController();
        return new EventHandler() {
        @Override
            public boolean handleEvent(int event) {
            System.out.println("Desktop.handleEvent() received event of type " + Event.
↪getType(event));
            return controller.handleEvent(event);
            }
        };
    }
};
```

It is also possible to display the content of MicroUI Event Buffer with the `LLUI_INPUT_IMPL_log_XXX` API. Please refer to *the Event Buffer documentation* for more information.

Implementation Details

Java Threads Creation

The number of threads in the MicroEJ Application must be sufficient to support the creation of additional threads when using Timer and Thread. The number of available threads can be updated in the launch configuration of the application (see *Option(text): Number of threads*).

If it is not possible to increase the number of available threads (for example, because the memory is full), try to reuse another thread but not the UI thread.

UART Not Available

If the UART output is not available, use another method to signal that the heartbeat task is running (e.g., blink an LED, use SystemView).

9.14.4 Use Case 2: Debugging a HardFault

When the application crashes, it can result from a HardFault triggered by the MCU.

The following sections explain:

1. What are exceptions, HardFaults, and the exception handler.
2. What to do in case of Memory Corruptions.
3. What to do when a HardFault occurs.

Useful Resources

- IAR System: Debugging a HardFault on Cortex-M https://www.iar.com/knowledge/support/technical-notes/debugger/debugging-a-hardfault-on-cortex-m/
- ESP-IDF Programming Guide: Fatal Errors https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/fatal-errors.html
- Using Cortex-M3/M4/M7 Fault Exceptions MDK Tutorial http://www.keil.com/appnotes/files/apnt209.pdf

Exceptions, HardFaults And Exception Handler

From ARM Architecture Reference Manual

> An exception causes the processor to suspend program execution to handle an event, such as an externally generated interrupt or an attempt to execute an undefined instruction. Exceptions can be generated by internal and external sources. Normally, when an exception is taken, the processor state is preserved immediately, before handling the exception. This means that, when the event has been handled, the original state can be restored and program execution resumed from the point where the exception was taken.

For example, an *IRQ request* is an exception that can be recovered by handling the hardware request properly. On the other hand, an *Undefined Instruction* exception suggests a more severe system failure that might not be recoverable.

The exceptions that cannot be recovered are named **HardFaults**.

From ARM Architecture Reference Manual

> When an exception is taken, processor execution is forced to an address that corresponds to the type of exception. This address is called the **exception vector** for that exception.

The code pointed by the exception vector is named **exception handler**. Therefore, a dedicated exception handler can be configured for all exceptions, including HardFaults.

Possible exceptions can be:

- Data Abort exception (access to unknown address)
- Undefined Instruction exception (execute code that is not valid)
- …

Check the hardware documentation for the complete list of exceptions.

What To Do In Exception Handlers?

For all HardFault handlers, the following data are available and must be printed:

- Name and value of all registers available
- Name of the handler
- Address of the failing instruction

Optionally:

- Content of the stack
- Call function `LLMJVM_dump` (from `LLMJVM.h`) to display the VM state (see *Dump the States of the Core Engine*)

Refer to the architecture documentation for how to configure the exception interrupt vector.

Memory Protection Unit (MPU)

A Memory Protection Unit (MPU) is a hardware unit that provides memory protection. An MPU allows privileged software to define memory regions and their policy. The policy describes who can access the memory.

For example, configure the heap and stack of a task to be accessible from the task itself only. The MPU generates an exception if another task or a device driver attempts to access the memory region.

If applicable, configure the MPU should to protect the application.

- Check the RTOS documentation if it supports MPU.

  For example, FreeRTOS includes FreeRTOS-MPU https://www.freertos.org/FreeRTOS-MPU-memory-protection-unit.html.

- Configure the MPU to configure the access to the JVM heap and stack to prevent any other native threads from altering this area. Refer to *this section* for the list of section names defined by the MicroEJ Core Engine.

Memory Corruption

Memory corruption can result in the following symptoms:

- The address of the failing instruction is in a data section.
- The trace is incomplete or incorrect.
- The address of the failing instruction is located in the Garbage Collector (GC).

The cause(s) of a memory corruption can be:

- A native (C) function has a bug and writes to an incorrect memory location
- A native stack overflow
- A native heap overflow
- A device incorrectly initialized or misconfigured.
- …

When the HardFault occurs in the MicroJVM task, the VM task heap or stack may be corrupted. Add `LLMJVM_checkIntegrity` call in checkpoints of the BSP code to identify the timeslot of the memory corruption. Typically, you can check a native with:

```
void Java_com_mycompany_MyClass_myNativeFunction(void) {
     int32_t crcBefore = LLMJVM_checkIntegrity();
     myNativeFunctionDo();
     int32_t crcAfter = LLMJVM_checkIntegrity();
     if(crcBefore != crcAfter){
             // Corrupted memory in MicroJVM virtual machine internal structures
             while(1);
     }
}
```

Investigation

Determine which memory regions are affected and determine which components are responsible for the corruption.

- List all the memories available and their specifics:
  - Access mode (addressable, DMA, ...)
  - Cache mechanism? L1, L2
- Is low-power enabled for CPU and peripherals? Is the memory disabled/changed to save power?
- Get the memory layout of the project:
  - What are the code sections for the BSP and the Application?
  - Where are the BSP stack and heap? What about the Application stack and heap?
  - Where is the Java immortals heap?
  - Where are the Java strings?
  - Where is the MicroEJ UI buffer?
  - Besides the Java immortals, what are the other intersection point between the Java application and the BSP? (e.g., a temporary RAM buffer for JPEG decoder).
  - Please refer to the *Core Engine Link section* to locate the Application sections, and to the *Standalone Application Options* for their sizes.
- Implement a CRC of the *hot sections* when entering/leaving all natives. *Hot Sections* are memory sections used by both Java code and native code (e.g., C or ASM).
- Move the C stack at the beginning of the memory to trigger a crash when it overflows (instead of corrupting the memory).

When a HardFault Occurs

Extract Information and Coredump

Attach an embedded debugger and get the following information:

- stack traces and registers information for each stack frame
- memory information
  - the whole memory, if possible
  - otherwise, get the *hot sections*

    * BSP and Java heap and stack
    * UI buffer
    * immortals heap
    * sections where the Java application and BSP are working together
- *Trigger VM Dump From Debugger*
- Check which function is located at the address inside the PC register.
  - It can be done either in Debug mode or by searching inside the generated .map file.

Memory Dump Analysis

- Run the Heap Dumper to check the application heap has not been corrupted.
- Make sure the native stack is not full (usually, there shall have the remaining initialization patterns in memory on top of the stack, such as `0xDEADBEEF`)

Trigger a VM Dump

`LLMJVM_dump` function is provided by `LLMJVM.h`. This function prints the VM state. Data printed in the VM state are:

- List of Java threads
- Stack trace for each thread

See *this section* to learn more about `LLMJVM_dump`.

9.15 How to Add Emojis to a Vector Font

MicroVG supports the drawing of multicolor fonts that use the COLR/CPAL tables to define multi-layered glyphs. Multicolor fonts are mainly used for providing a set of colorful emojis in messaging applications. However, emojis fonts usually do not contain many characters other than emojis, which requires applications to use multiple fonts to handle all use cases.

One solution to minimize the number of fonts used by an application is to add emojis to another font (i.e., combine fonts into one). This article shows how to achieve this using FontLab, a third-party font editor.

Note: FontLab is not a free software (it has a 30-days trial period). Tests with other tools, including free solutions, were unsuccessful in this very specific task (e.g. FontTools, FontForge which are great tools for font editing).

9.15.1 Intended Audience

The audience for this document is Application engineers who want to use *Vector Fonts* in their applications.

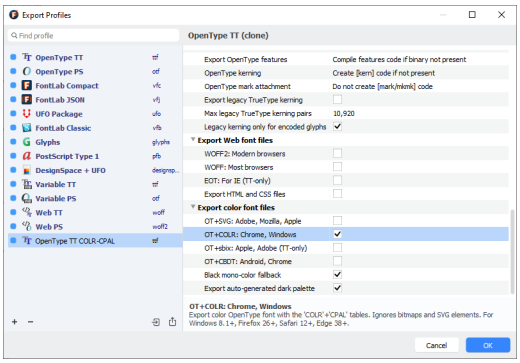
In addition, this tutorial should be of interest to all developers wishing to familiarize themselves with the vector features of *MicroVG*.

9.15.2 Prerequisites

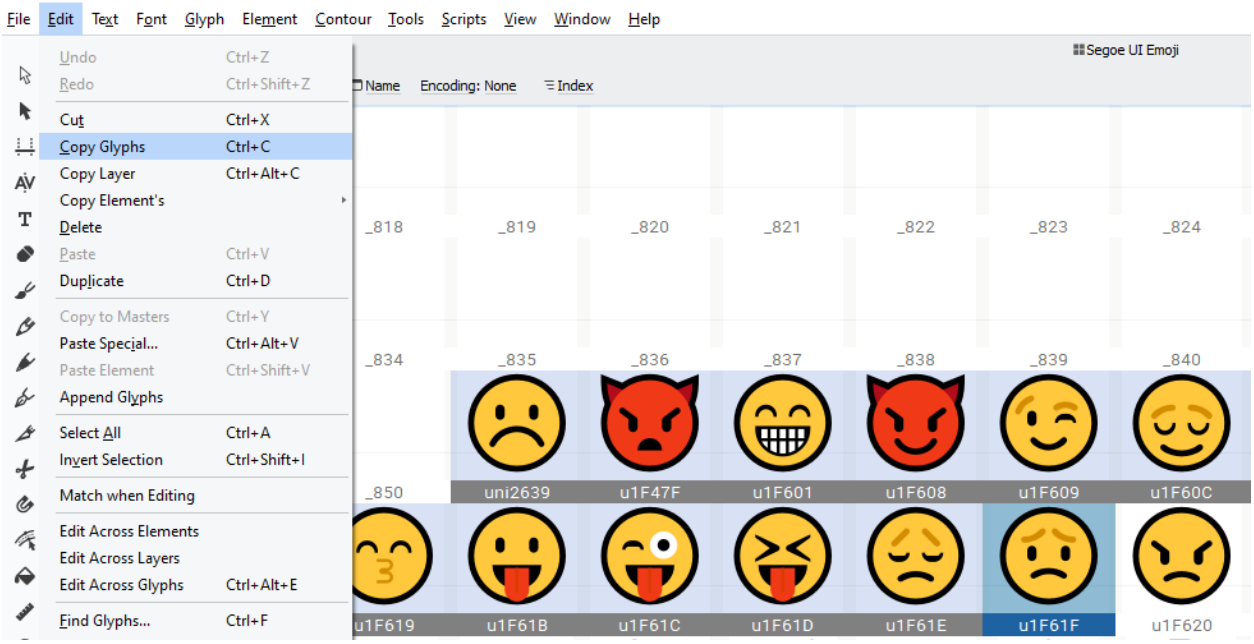
- Windows 10 (and higher) or macOS 10.14 (and higher),
- An COLR/CPAL emoji font (e.g., Segoe UI Emoji),
- A target font (i.e., a TTF/OTF font to append emojis to).

9.15.3 Append the Emoji Glyphs

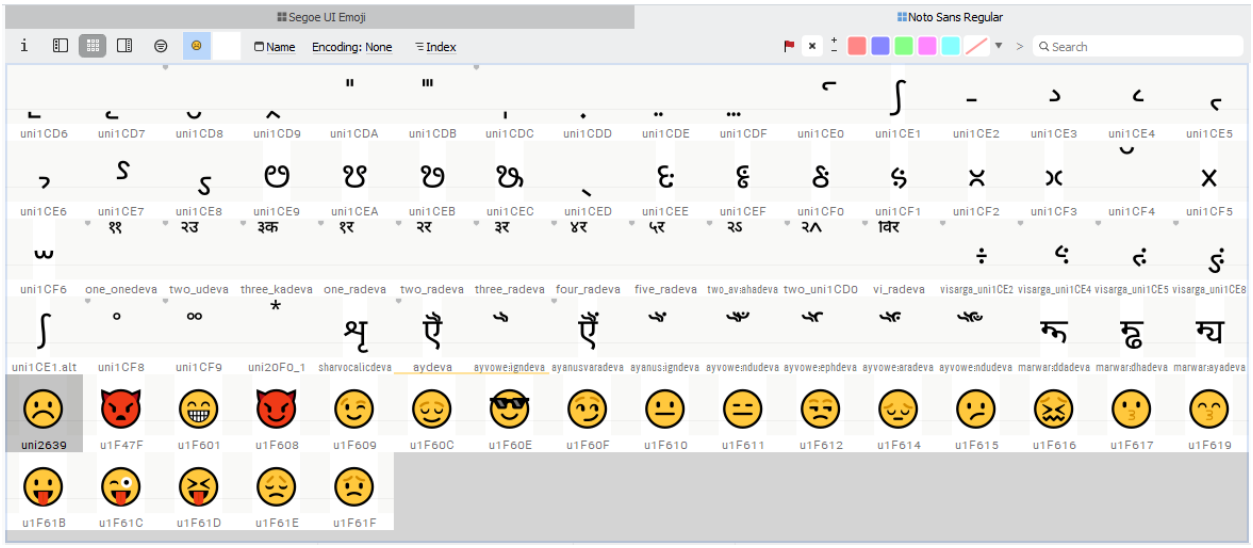
1. Download and install FontLab.
2. In FontLab, go to `File` > `Export Profiles…` .
3. Create a new Export Profile (the `+` button on the bottom-left).
4. Edit the new profile to match the configuration below in menu `Export color font files` :



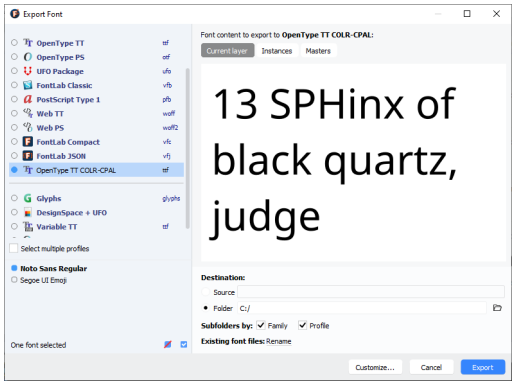
5. Click on `OK` to close the `Export Profiles` window.
6. Open the emoji font: go to `File` > `Open Fonts…` and browse to the font file.
7. Open the target font.
8. Select a range of glyphs in the emoji font and select `Edit` > `Copy Glyphs` :



9. Select the target font and append the copied glyphs: go to Edit > Append Glyphs .
10. Check that the glyphs have been added to the target font:



11. To save the changes, go to File > Export Font As… .
12. In the Export Font dialog, select the new Export Profile (with COLR/CPAL support) and Destination .



13. Click Export .

The exported font can then be used in an application, as described in the *Application Developer Guide*.

This procedure can also be used to add non-emoji glyphs to a font.

Warning: There are multiple ways of implementing emojis in fonts. The four main formats are COLR/CPAL (Microsoft), CBDT/CBLC (Google), SVG (Adobe/Firefox) and sbix (Apple). Each format uses custom tables in fonts to describe the emoji glyphs. MicroVG supports COLR/CPAL tables and this article only applies to this case. See *this section* for more details about color emojis support with MicroVG.

9.16 How to Validate GUIs

This document explains how to validate Graphical User Interfaces. It describes common pitfalls that can affect GUI performances, provides tools that allow to detect performance issues and how to solve them, and finally offers ways to test GUIs automatically.

9.16.1 Implementing GUIs Efficiently

Before using more advanced UI debugging techniques, the global application code quality should be reviewed. An overall good code quality will help to get good UI performances. It will help to get more efficient code and allow easier debugging and maintenance.

Documents and Tools to Improve Application Code Quality

Here is a list of documents or tools that help to improve the quality of application code:

- The *Improve the Quality of Java Code* tutorial explains how to improve the Quality of Java Code.
- The *Get Started With GUI* tutorial provides guidelines to start developing an efficient GUI.
- The SonarQube™ source code quality analyzer allows to analyze an Application or Library code quality.

Using Recent Versions of UI Libraries

Using the latest UI libraries (MicroUI, MWT, Widget, etc.) available may solve some performance issues. The most recent UI libraries fix some bugs that may affect performance and they provide tools and libraries that allow to implement more performant UIs.

Memory Management

The Java management of memory may affect UI performances:

- Too much memory allocation/deallocation for UI resources (Images, Fonts).
- Too much object instantiation will lead to a big Java heap size. In some use cases, the garbage collection may lead to the UI slowing down.

To avoid those pitfalls:

- Calibrate the memories (Java heap, Images heap, etc.).
- Uses memory debugging tools:
  - *Optimize Memory Footprint* tutorial.
  - *Memory inspection tools*.
  - *Heap Analyzer*.

Format of UI Resources

One crucial aspect of optimizing an application is choosing the right image formats. Images can have a significant impact on an app's performance and memory usage. Therefore, selecting the best image format is essential. It helps reduce memory usage, speed up the app, and improve its overall performance.

MicroUI manages two kind of images, mutable and immutable images.

Mutable images are graphical resources that can be created and modified at runtime. The application can draw into such images. More information about mutable images can be found *here*.

As their name suggests, immutable images can not be modified. They are the most commonly used kind of images, this section will focus on them.

Decoding Immutable Images

Immutable images can be converted for display either during the build-time process, using the Image Generator, or at run-time, utilizing the appropriate decoder library.

The decision between these two approaches depends on the project's specific requirements. **Decoding at run-time** is a good choice when storage space is limited and offers greater flexibility. However, it may require more processing power and result in slower performance. Conversely, **decoding at build time** reduces the computational workload during run-time and is well-suited for devices with stringent performance demands, though it usually require more storage and it may sacrifice some flexibility in the process.

Format of Immutable Images

There are multiple output formats that can be used to convert the images, you can find them here: *Output Formats*.

Choosing the right output format is important to get the best performance:

- For opaque images, choose a format that has no transparency, RGB565 is usually sufficient.
- For a pictogram to colorize A4 is usually sufficient. The image can be colorized at runtime.
- The image format can be compressed, see *Compressed Output Formats*

The expected result of each format can be seen here: *Formats expected result*

Images Heap

Mutable images and immutable images decoded at runtime require some memory to be used. Please go to the *Images Heap* section for more information.

9.16.2 Benchmarking GUIs

The process of rendering a frame of the UI consists of several parts:

- Drawing of the UI:
  - MWT processing (layout of widgets and widget rendering process).
  - Drawing of the UI (MicroUI drawing method execution).
- Display flush.
- (depends on the UI port) Backbuffer copy, see *Display Buffer Modes*.

Some tools are available to identify which part of this process affect the most the GUI performance.

SystemView

The SystemView tool can be used to trace the UI actions (drawings, flush, etc.) and detect which ones are the most time-consuming. The documentation of SystemView is available *here*. The MicroUI traces should be configured in SystemView in order to see the UI actions performed, it can be done by following *this documentation*. Custom traces can be added and logged from the Java application to record specific actions.

MicroUI Flush Visualizer

A perfect application has 100% of its display area drawn. This is the total area covered by the sum of the area drawn by the drawing operations. A value of 200% indicates the area drawn is equivalent to twice the surface of the entire display. A total area drawn between 100% to 200% is the norm in practice because widgets often overlap. However, if the total area drawn is bigger than 200%, that means that the total surface of the display was drawn more than twice, meaning that a lot of time could be spent drawing things that are never shown.

The MicroUI Flush Visualizer tool can be used to investigate potential performance bottlenecks in UI applications running on the Simulator by showing the pixel surface drawn between two MicroUI frame buffer flushes.

The documentation of MicroUI Flush Visualizer is available *here*.

9.16.3 Debugging GUIs

High-level Debugging and Optimizations

This section provides insights into common issues affecting performances on the high-level side. The following advices will help reduce the MWT processing and drawing time.

Widget Hierarchy and Layout

Keeping the widget hierarchy as simple as possible will help to reduce the "MWT processing" part time. Improving the widget hierarchy design may help reduce the number of widgets or the number of them that are rendered when a certain part of the UI is updated.

Here are tools that allow to detect issues with the widget hierarchy:

- *Widget debug utilities* provides tools to visualize the widget tree, count the numbers of widgets or see their bounds.
- *MWT bounds highlighting* allows to visualize the bounds of the widgets, it is useful to detect overlapping widgets.

Bad Use of requestRender and requestLayout

The `requestRender` method requests a render of the widget on the display.

The `requestLayout` method requests a layout of all the widgets in the sub-hierarchy of this widget. It will compute the size and position of the widgets as setting their styles. `requestLayout` will trigger a render request after the layout.

A common mistake is to call `requestRender` just after a `requestLayout`. This will trigger two renders and thus affect the UI performances.

Another common issue is to request a layout where a render request would have been enough. If the size, position or style of the widgets didn't change `requestRender` is enough, `requestLayout` would have a longer processing time. This is especially true for animation where we want each frame to be processed as fast as possible.

Documentation about rendering and layout is available *here*.

Animations Implementation

There are a few implementations possible for animations with MicroEJ. The way widgets are animated should be chosen according to the use case and the limitation of the hardware.

Animator

The MWT's Animator allows to execute animations as fast as possible, it waits for the low-level screen flush to be done and directly triggers a new render. Thus the Animator will give the best framerate possible but will also consume a lot of CPU processing time.

TimerTask

A TimerTask can be used to execute an animation at a fixed framerate. This technique is very useful to set a fixed period for the animation but will cause issues if the time to render a frame is longer than that period, this will lead to missed frames. Some frames can take longer to render if their content is more complex or if the CPU is already used by another non-UI thread.

The framerate set when using a TimerTask for animation should be defined wisely, the time to render frames and the CPU utilization should be taken into consideration.

Animator and TimerTask mix

A mix of the Animator and TimeTask approaches could be implemented in order to set a fixed framerate but also to rely on the screen flush.

Hardware and Low-level Debugging and Optimizations

This section provides insights into the main spots to check regarding the low-level and the hardware.

Please see the VEE Porting Guide *Graphical User Interface* documentation for more information about the UI port.

At Project Level

Compiling Optimization Options

The project should be configured to bring the best performances with compiling optimization options correctly set up.

RTOS Tasks Environment

The priority of the UI task should be set high enough to avoid too many preemptions that may induce bad UI performances.

Another point that should be taken into consideration is the amount of other tasks that are running at the same time as the UI task. The total workload may be too high for the CPU, therefore, the UI task cannot get access to the required amount of computing power.

At Hardware Level

Hardware Capabilities

MCUs and SoCs may have access to various hardware IPs to speed up the UI. The UI port should exploit all of them to get the best performance. First of all, the GPU should be used if it is available on the system. Then, driving a display implies intensive memory usage, a DMA should be used whenever it's possible.

For example, during the back copy if the flush policy is in switch mode or during your flush if your display is driven through SPI (if there is a DMA dedicated to the SPI port). For more information about the flush policy, please read our documentation about *Display*.

Hardware Configuration

Each of the hardware components such as SPI, DMA or LCD controller must be configured to bring the best performances achievable. This implies to read carefully the datasheet of the MCU and the display and determine for example the best frequency and communication mode possible.

Another example of configuration with DMAs, a DMA has often a burst mode to transfer data, the UI port should use this mode to maximize performance.

Buffers Location in Memory

An important step during the development of the UI integration is the memory location of the buffers that will use the GUI to draw to the display. In an MCU, there may be different types of RAM available that have different properties in terms of quantity and speed. The fastest RAM should be chosen for the buffers if its size allows it.

Flush Policy

As described in the *Display* page, there are several flush policies that can be implemented. The best flush policy should be selected according to the hardware capabilities. Generally, the best flush policy is the switch mode.

9.16.4 Testing GUIs

Before applying UI debugging or optimization techniques, the application behaviour should be tested. There are different ways of doing this.

Test a GUI Application with a Software Robot

It is possible to test the GUI of an application via robotic process automation (RPA). Robot tests mimic the human user behavior in the GUI and can help detect various errors by automating behavior which otherwise would cost too much effort and/or time to execute manually.

Here are the steps required to use a robot in the MicroEJ environment:

- Record the robot input events
  - For this, you need a simple EventHandler which intercepts incoming events, for example from a Pointer, then passes them on to the real event handler.
- Start the usage of the new 'Watcher' logic after the UI has started
  - With this, the watching of the Pointer envents is initiated for the whole application.
- Create a Robot
  - The robot is a simple class which uses its own Pointer to move and press at the coordinates it has been instructed.
  - The robot should have a method which starts a series of instructions to move the Pointer.
- Execute the Robot method containing the instructions
  - The intercepting Event Handling will record and for example log the input.

This simple way of automating GUI actions can be used to carry out real use cases and evaluate the results.

The *How to test a GUI application with a (software) robot* tutorial provides detailed insight into this topic.

Test a GUI Application with the Test Automation Tool

To execute regression tests automatically and monitor minor changes in a GUI, you can use the Test Automation Tool. The Test Automation Tool allows to automatically test UIs.

The tool comparison functionality can be integrated with JUnit tests.

For detailed information about the tool usage, please check the README in the repository.

9.17 How to Test a GUI Application with a (Software) Robot

This document presents how to test a GUI application with a software robot for robotic process automation (RPA).

Robot tests and traditional unit tests are different but both are useful. Traditional unit tests validate the systems through calls to the API (internal or external). Robot tests validate the systems by mimicking the human user behavior directly in the GUI. The robot implementation proposed here targets the following errors detection:

- OutOfMemory
- StackOverflow
- MEJ32 and platform libraries error
- Widget sequence validation

The following document covers:

- Recording human touch events on the simulator or on the embedded platform
- Running recorded events on the simulator or on the embedded platform

The following document does not cover:

- The display rendering validation (this can be done using the Test Automation Tool)
- Integration of the robot into an automatic JUnit test suite

We will now present the basic architecture and code required to create and to run a robot within a MicroEJ application on the simulator and embedded platform.

In the following sections, we assume the MicroEJ VEE Port has a display interface and a touch controller.

9.17.1 Overview

The robot creation process is twofold. First, we have to record and store the human user events. Second, we have to play them back with the robot.

9.17.2 Record the Scenario

The first step is to record the human user events.

Here is the code of the `EventRecorder` class that should be added to our application's project:

```
import ej.annotation.Nullable;
import ej.microui.event.Event;
import ej.microui.event.generator.Buttons;
import ej.microui.event.generator.Pointer;
```

(continues on next page)

(continued from previous page)

```
/**
* Records events.
*/
public class EventRecorder {

  private long lastEventTime;

  /**
  * Creates an event recorder.
  */
  public EventRecorder() {
    this.lastEventTime = -1;
  }

  /**
  * Records an event.
  *
  * @param event
  *            the event to record.
  */
  public void recordEvent(int event) {
    String command = getEventCommand(event);
    if (command != null) {
      long currentTime = System.currentTimeMillis();
      if (this.lastEventTime == -1) {
        this.lastEventTime = currentTime;
      }

      long delta = currentTime - this.lastEventTime;
      if (delta > 0) {
        System.out.println(getPauseCommand(delta));
      }

      System.out.println(command);

      this.lastEventTime = currentTime;
    }
  }

  @SuppressWarnings("nls")
  private static @Nullable String getEventCommand(int event) {
    if (Event.getType(event) == Pointer.EVENT_TYPE) {
      Pointer pointer = (Pointer) Event.getGenerator(event);
      switch (Pointer.getAction(event)) {
      case Pointer.PRESSED:
        return "robot.press(" + pointer.getX() + ", " + pointer.getY() + ");";
      case Pointer.MOVED:
      case Pointer.DRAGGED:
        return "robot.move(" + pointer.getX() + ", " + pointer.getY() + ");";
      case Buttons.RELEASED:
        return "robot.release(" + pointer.getX() + ", " + pointer.getY() + ");";
      default:
```

(continues on next page)

(continued from previous page)

```
        return null;
      }
    } else if (Event.getType(event) == Buttons.EVENT_TYPE) {
      if (Buttons.getAction(event) == Buttons.RELEASED) {
        return "robot.button();";
      } else {
        return null;
      }
    } else {
      return null;
    }
  }

  @SuppressWarnings("nls")
  private static @Nullable String getPauseCommand(long delay) {
    return "robot.pause(" + delay + ");";
  }
}
```

This code records all pressed, moved, dragged and released events as well as the time between each event (we want to play our robot at the same speed as the human). `EventRecorder` outputs the commands on the standard output. More on this a bit later.

Set Up the Event Recorder

The events have to be recorded from the application's desktop's `EventDispatcher`. Here is how to override it:

```
final EventRecorder eventRecorder = new EventRecorder();

Desktop desktop = new Desktop() {

  @Override
  protected EventDispatcher createEventDispatcher() {
    return new PointerEventDispatcher(this) {

      @Override
      public boolean dispatchEvent(int event) {
        eventRecorder.recordEvent(event);

        return super.dispatchEvent(event);
      }
    };
  }
};
```

When runnning the application, the `EventDispatcher` will now record the events and then redirect them to its parent `dispatchEvent` so they can be managed normally by the application.

9.17.3 Set Up the Scenario Player

As we now have recorded our scenario we have to play it. For that we have to add the `EventPlayer` to our project:

```
/**
 * Plays events.
 */
public class EventPlayer {

  @Nullable
  private final Pointer pointer;
  @Nullable
  private final Buttons buttons;

  /**
  * Creates a robot.
  */
  public EventPlayer() {
    this.pointer = EventGenerator.get(Pointer.class, 0);
    this.buttons = EventGenerator.get(Buttons.class, 1);
  }

  /**
  * Pauses before the next action.
  *
  * @param delay
  *            the delay to pause.
  */
  public void pause(long delay) {
    ThreadUtils.sleep(delay);
  }

  /**
  * Generates a press event.
  *
  * @param x
  *            the x coordinate of the pointer.
  * @param y
  *            the y coordinate of the pointer.
  */
  public void press(int x, int y) {
    if (null != this.pointer) {
      this.pointer.reset(x, y);
    }
    if (null != this.pointer) {
      this.pointer.send(Pointer.PRESSED, 0);
    }
  }

  /**
  * Generates a move event.
  *
  * @param x
```

(continues on next page)

(continued from previous page)

```
  *            the x coordinate of the pointer.
  * @param y
  *            the y coordinate of the pointer.
  */
  public void move(int x, int y) {
    if (null != this.pointer) {
      this.pointer.move(x, y);
    }
  }

  /**
  * Generates a release event.
  *
  * @param x
  *            the x coordinate of the pointer.
  * @param y
  *            the y coordinate of the pointer.
  */
  public void release(int x, int y) {
    if (null != this.pointer) {
      this.pointer.reset(x, y);
    }
    if (null != this.pointer) {
      this.pointer.send(Pointer.RELEASED, 0);
    }
  }

  /**
  * Generates a button event.
  */
  public void button() {
    if (null != this.buttons) {
      this.buttons.send(Buttons.RELEASED, 0);
    }
  }
}
```

`EventPlayer` will play events using the `EventGenerator`.

We will now extend `EventPlayer` in order to play a specific scenario:

```
/**
* Robot scenario which reproduces the recorded human user events .
*/
public class NavigationScenario extends EventPlayer implements Runnable {

  @Override
  public void run() {
    press(344, 177);
    pause(885);
    release(344, 177);
    pause(359);
```

(continues on next page)

(continued from previous page)

```
    press(184, 192);
    pause(34);
    move(185, 192);
    pause(24);
    move(188, 192);
    pause(23);
    move(191, 192);
    pause(24);
    move(196, 192);
    pause(21);
    move(206, 191);

  }
}
```

The `run` method from the code above already contains recorded events, you will have to replace it by the `EventRecorder` output you get when recording the events.

9.17.4 Run the Scenario

We will now create a task that will run the scenario:

```
  /**
* A robot task is able to run a given scenario.
*/
public class RobotTask {

  private boolean running;

  /**
  * Creates a demo robot.
  */
  public RobotTask() {
    this.running = false;
  }

  /**
  * Starts the given scenario.
  *
  * @param scenario
  *            the scenario to run.
  */
  public void startScenario(final Runnable scenario) {
    if (!this.running) {
      this.running = true;

      new Thread() {
        @Override
        public void run() {
          scenario.run();
          RobotTask.this.running = false;
```

(continues on next page)

(continued from previous page)

```
        }
      }.start();
    }
  }

  /**
  * Returns whether the robot is currently running.
  *
  * @return <code>true</code> if the robot is running, false otherwise</code>.
  */
  public boolean isRunning() {
    return this.running;
  }
}
```

You can now start the `RobotTask` in your application:

```
RobotTask robot = new RobotTask();
                  robot.startScenario(new NavigationScenario());
```

Then, launch your application: the recorded scenario is now re-played, well done!

CHAPTER

TEN

GET SUPPORT

If any questions, the best starting point is to consult the MicroEJ Forum. Feel free to create a new topic if there is no relevant content for your issue. MicroEJ Corp. engineers are listening to the forum activity, so you can expect to get a quick reply.

Otherwise, you can contact our support team.

In both cases, please provide as much information as possible on your installed environment (the table below is an example):

Delivery	**Name**
MicroEJ SDK	Distribution 20.07 / Version 5.2.0 (see *SDK Version*)
MicroEJ Architecture	ARM Cortex-M4 / IAR / Evaluation | Production (see *MicroEJ Architecture*) version *XYZ*
Module Repository	https://repository.microej.com/packages/repository/2.5.0/microej-5_0-2.5.0.zip (see *Central Repository*)
C compiler	IAR 8.40.1
Host Operating System	Windows 10 (see *System Requirements*)

CHAPTER

ELEVEN

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

Abstraction Layer, 2
Add-On Library, 2
Application, 2
Architecture, 2

C

Core Engine, also named "MEJ32", 2

E

Executable, 2

F

Foundation Library, 3

M

MICROEJ SDK, 3
MICROEJ VEE, 3
Mock, 3
Module Manager, 3

S

Simulator, 3

V

VEE Port, 3
Virtual Device, 3