MICROEJ SDK 6 Basics

For NXP i.MX RT1170 Evaluation Kit

© MicroEJ 2025

MICROEJ®

DISCLAIMER

All rights reserved. Information, technical data and tutorials contained in this document are proprietary under copyright law of MicroEJ S.A. Without written permission from MicroEJ S.A., copying or sending parts of the document or the entire document by any means to third parties is not permitted. Granted authorizations for using parts of the document or the entire document do not mean MicroEJ S.A. gives public full access rights.

The information contained herein is not warranted to be error-free.

MicroEJ® and all relative logos are trademarks or registered trademarks of MicroEJ S.A. in France and other Countries.

Other trademarks are proprietary of their respective owners.

Java™ is Sun Microsystems' trademark for a technology for developing application software and deploying it in cross-platform, networked environments. When it is used in this site without adding the "™" symbol, it includes implementations of the technology by companies other than Sun. Java™, all Java-based marks and all related logos are trademarks or registered trademarks of Sun Microsystems Inc, in the United States and other Countries.

WHAT YOU WILL LEARN

- Understand the VEE Port concepts
- Create and run your first application project with MICROEJ SDK
- Learn how to configure your Application Project
- Use a Front Panel project
- Get an overview of MICROEJ SDK Development Tools
- Call a C function from Managed Code (Java)
- Blink an LED from Managed Code (Java)

PREREQUISITES

© MicroEJ 2025

PREREQUISITES

Hardware required:

- NXP i.MX RT1170 Evaluation Kit (EVKB) + micro-USB cable + RK055HDMIPI4MA0 display panel
- More information about the Evaluation Kit: NXP i.MX RT1170 User Manual

Environment Setup:

- Follow the NXP i.MX RT1170 Evaluation Kit Getting Started to setup your environment and run a demo application on the Virtual Device and on the i.MX RT1170 Evaluation Kit.
- Note: the next slides are using **IntelliJ IDEA** with **MicroEJ plugin for IntelliJ IDEA 1.1.0**. This training supports all other available IDEs (Android Studio, VS Code, …)

This training requires the Getting Started to be completed until the
Run an Application on the i.MX RT1170 Evaluation Kit section (included).

⚠️ The path to the NXP i.MX RT1170 VEE Port sources should be as short as possible and contain **no whitespace** or **non-ASCII character.**

© MicroEJ 2025

Virtual Execution Environment

MICROEJ VEE Overview

MICROEJ VEE

MICROEJ VEE is a scalable Virtual Execution Environment for **resource-constrained** embedded and IoT devices running on 32-bit microcontrollers or microprocessors.

MICROEJ VEE allows devices to **run multiple and mixed Managed Code and C** software applications.

Key Figures:

- Boots in 2 ms on a Cortex-M4 @180MHz.
- Optimized for low-power.
- Compact (< 30 KB footprint).
- Runs from Cortex-M0 with 128 KB flash and 32 KB RAM, to Cortex-A7.

© MicroEJ 2025

MICROEJ VEE

SERVICES

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

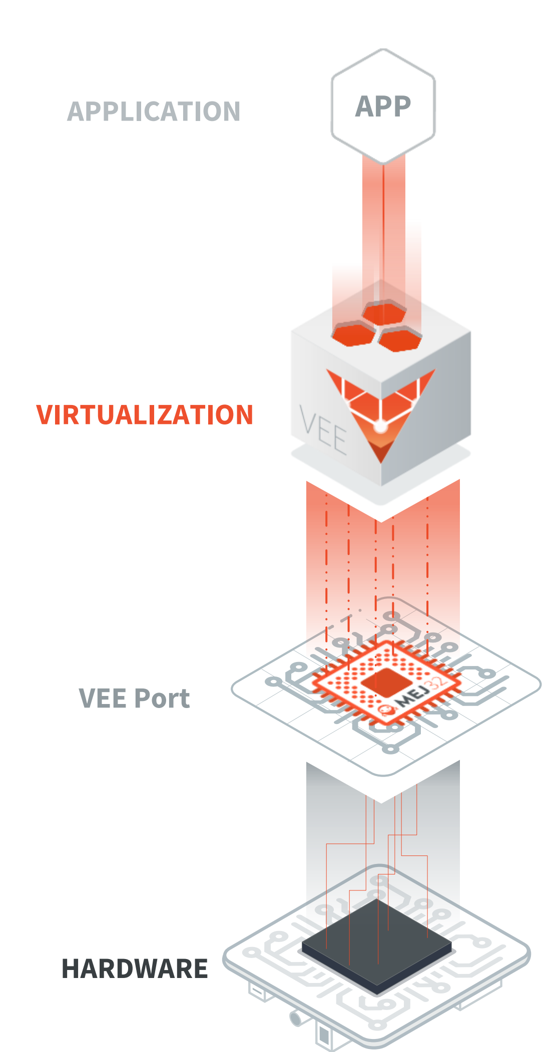
- A secure **multi-application** framework.
- A **network connection with security** (SSL/TLS, HTTPS, REST, MQTT, CoAP,…).
- A **GUI framework** (includes widgets).
- A storage framework (file system).
- VEE Wear: software solution dedicated to the development of wearable software.
- VEE Energy: software solution dedicated to the development of energy devices, including smart meters, gateways, and connected grid infrastructure.

As it runs standard languages such as Java code, MICROEJ supports all security, networking and IoT communication protocols and frameworks such as MQTT, CoAP, etc.

Refer to the Application Developer Guide to learn more about MICROEJ VEE Services.

© MicroEJ 2025

MICROEJ VEE – DETAILED VIEW



SANDBOXED APPLICATIONS

APPLICATIONS for MICROEJ VEE

ADD-ON LIBRARIES

Web / REST servers | MQTT / LWM2M clients | JSON | CBOR | Crypto | Widgets | Components | Eclasspath | ...

FOUNDATION LIBRARIES

FS | NET/SSL | MicroUI | EDC/BON (java.lang) | ECOM | BLE | ...

MICROEJ VEE

Managed Code (Java, JavaScript, ...)

VIRTUALIZATION

ABSTRACTION LAYERS

File System | Internet Protocol | Graphical Engine

MEJ32

ABSTRACTION LAYERS

I/O | Bluetooth | ...

BSP (RTOS/OS, Drivers, Third party stacks)

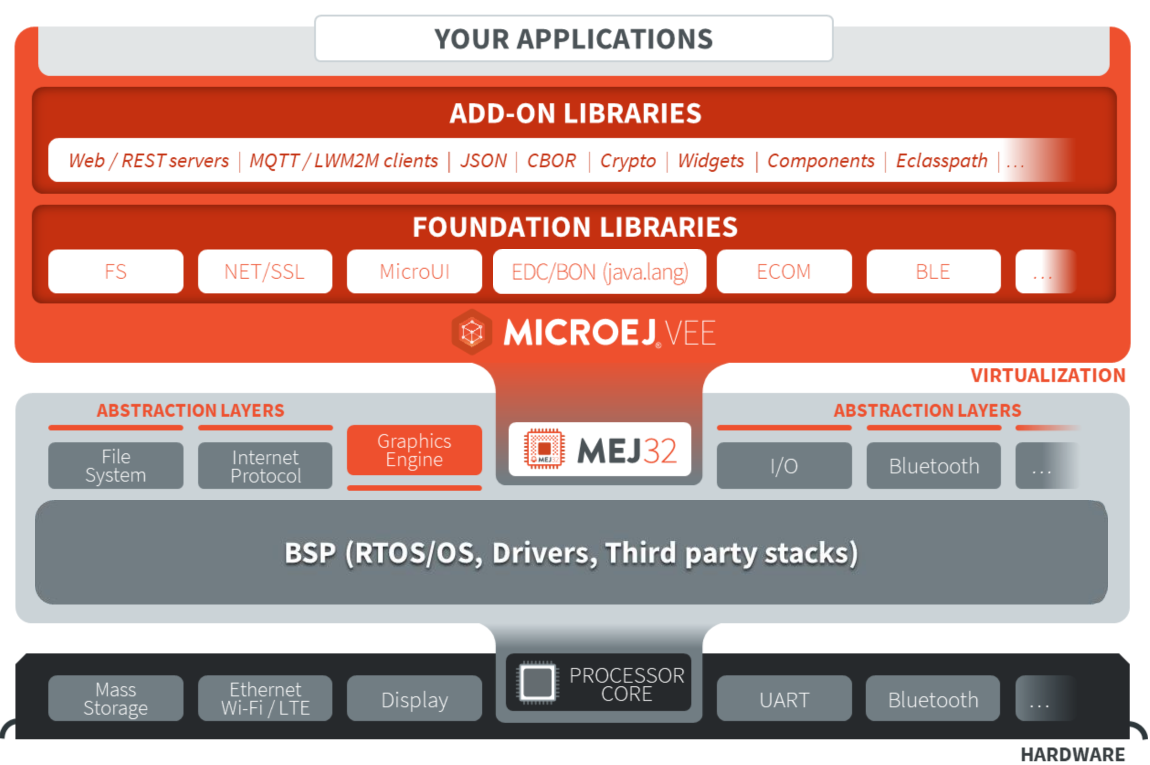
Native Code (C/ASM, ...)

VEE Port

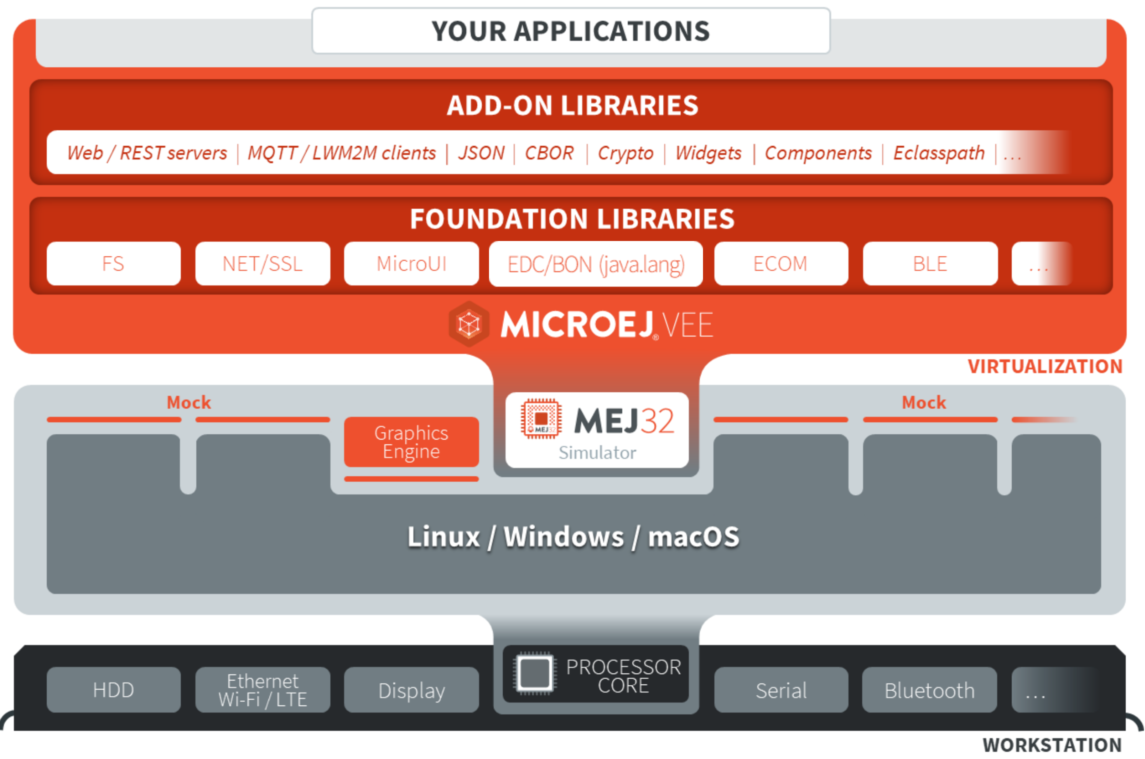
Mass Storage | Ethernet Wi-Fi / LTE | Display | PROCESSOR CORE | UART | Bluetooth | ...

HARDWARE

MANAGED CODE ON HARDWARE & VIRTUAL DEVICE



REAL DEVICE



VIRTUAL DEVICE

© MicroEJ 2025

CORE ENGINE BLOCK DIAGRAM

APPLICATION DATA (ROM)

Code

Execution In Place (XIP)

Resources

- Streaming and Seeking access modes
- Internal Resources Linker (memory mapped)
- External Resources Loader (non byte-addressable memories)

DEBUG INTERFACE

- **Debugger Proxy**
- **Integrity Check**
- **Core Dump**
- **Heap Dump**
- **Task/Threads Execution Tracing**
- **Advanced Profiling**
  - Method calls (Flame Graph)
  - Object Allocations
  - Exceptions

MEJ32

Smart Memory Manager

- Memory Allocator
- Garbage Collector
- Check Heap Overflow

Dynamic Loader(*)

- Streaming Linker
- with/without symbols

Exceptions Manager

- Zero Division & Null Pointer
- Array Index Out of Bounds
- Invalid Reference Cast
- Stack & Memory Overflow
- Catch & Finally Semantic

Processing Unit

- High Density Instructions (ISA)
- FPU 32-bit & 64-bit
- 16-bit Pointers(*)
- Software MPU

Multi-Sandbox Mode(*)

- App Execution Control
- CPU Control
- RAM Control

Native Interface

- Procedure Call Standard ABI
- Native Resources Manager

Scheduler

- Green Threads Policy
- Synchronization Monitors

(*) = optional

BINARY INTERFACE

- **Few Lib Dependencies**
- **I/O & Character Print**
- **ELF Linker**
- Software or Interrupt Timer
- Application Time

RUNTIME DATA (RAM)

Immortal Objects Heap

Managed Objects Heap

Internal Structures

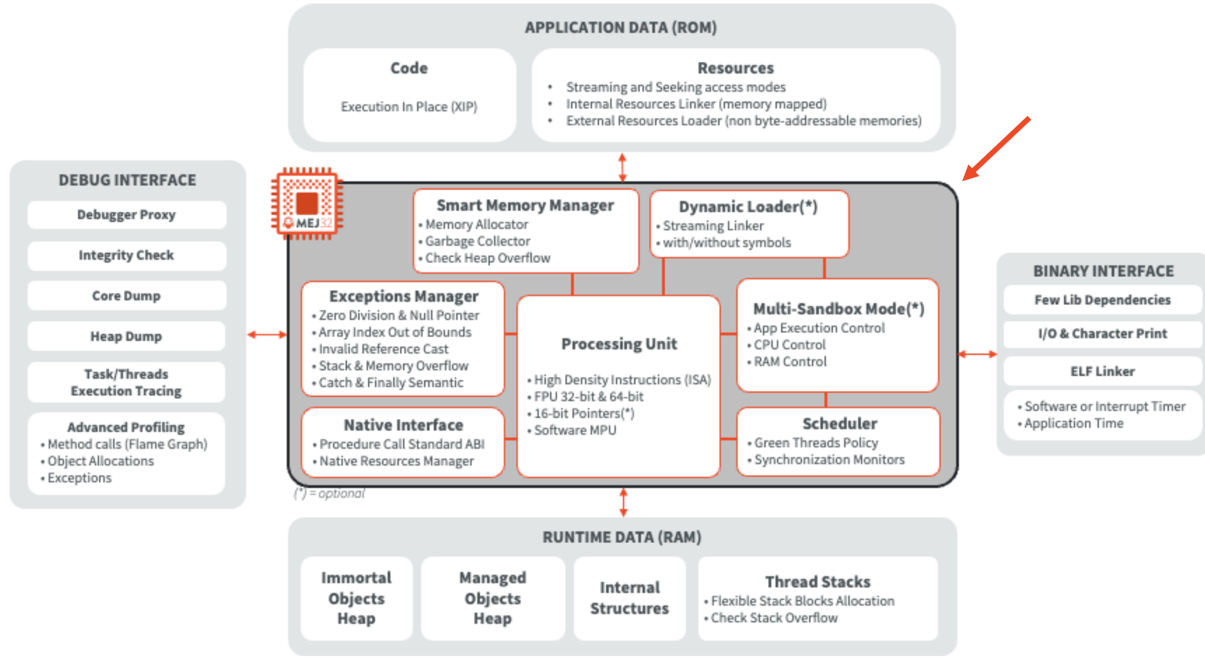
Thread Stacks

- Flexible Stack Blocks Allocation
- Check Stack Overflow

© MicroEJ 2025

CORE ENGINE KEY POINTS (1/3)

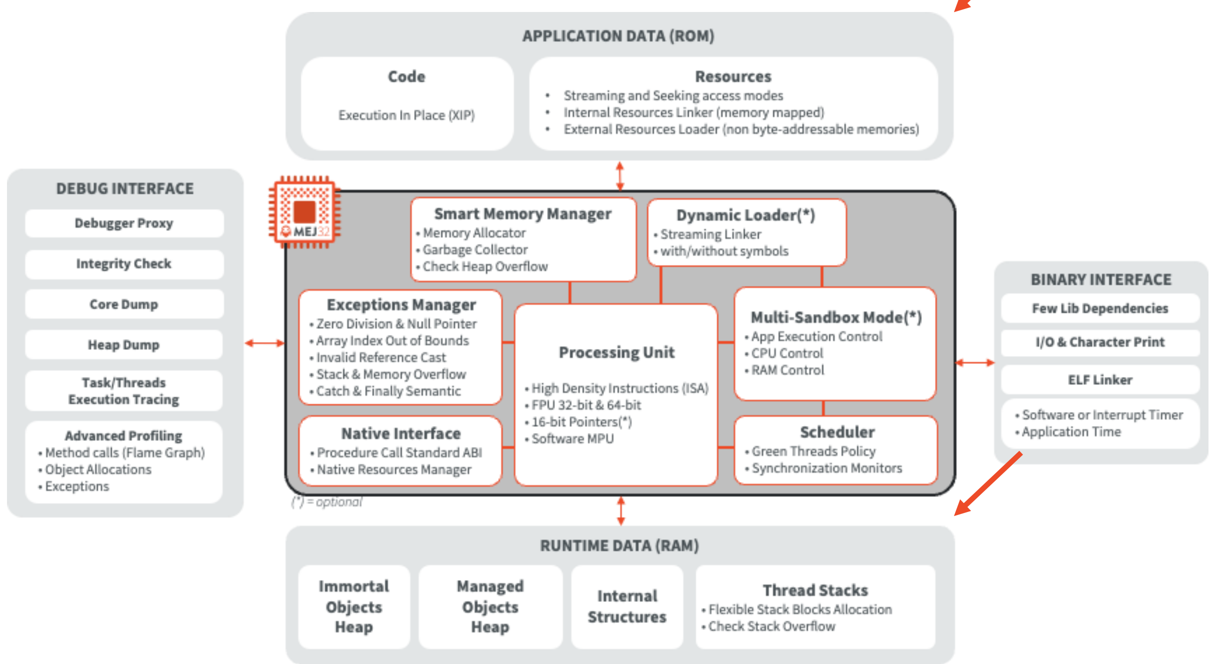
- MEJ32
  - The Core Engine, also named MEJ32, is a scalable 32-bit core for resource-constrained embedded devices. It is delivered in various flavors, mostly as a binary software package. The Core Engine allows applications written in various languages to run in a safe container called MICROEJ VEE.



© MicroEJ 2025

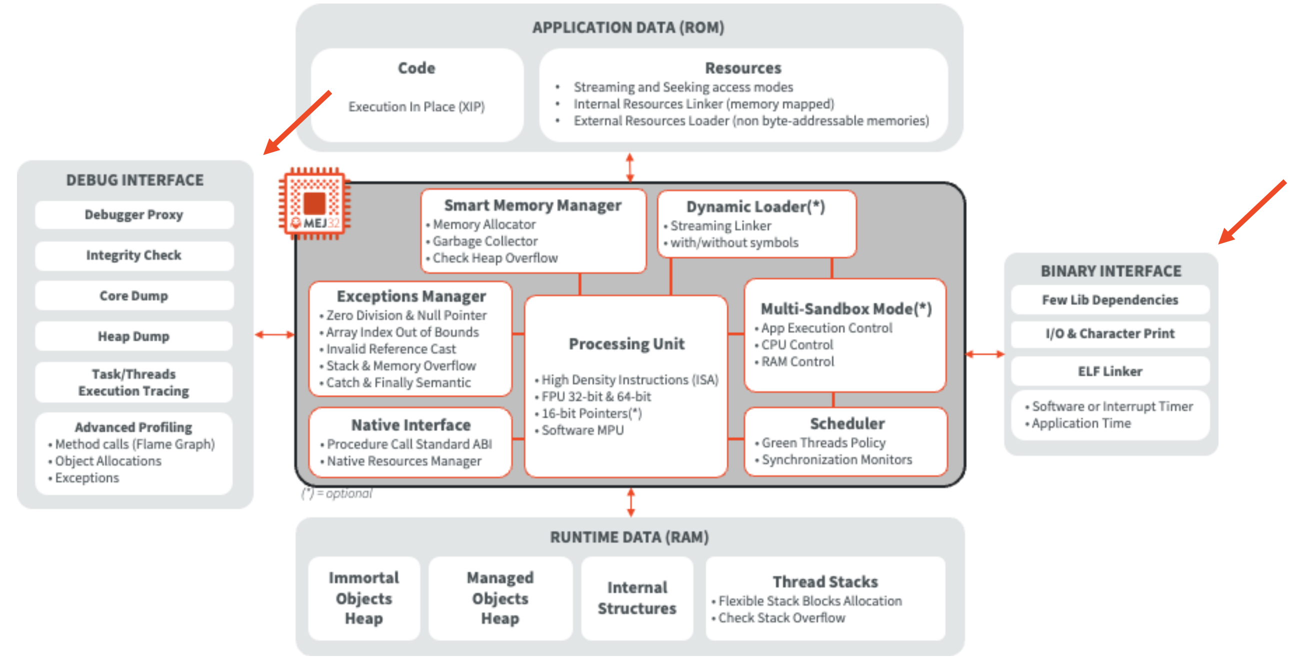
CORE ENGINE KEY POINTS (2/3)

- Application Data
  - The Application Data consists in the code itself as well as the resources (fonts, images, translations, etc.) stored in persistent and read-only memory (most of the time a device's flash memory).
- Runtime Data
  - The Runtime Data is mainly allocated in volatile memory spaces for the Core Engine to execute the Application properly (stacks, heaps, and internal structures in RAM).



CORE ENGINE KEY POINTS (3/3)

- Binary Interface
  - An Interface to different IOs, system, and external libraries calls. Most notably it includes, if necessary, an embedded ELF Linker that allows dynamic code linking at runtime. It is also at this level that calls to the OS and hardware are reified.
- Debug Interface
  - An Interface for profiling, tracing, and debugging the execution behavior.



VEE Port Concept

Computing platform for embedded system development

© MicroEJ 2025

STATE OF PLAY

- Programs made for workstations and servers are portable to Linux / OS X / Windows.
- iOS or Android let you run the **same application on several hardware** targets.
- Developers use **high-level languages** and tools.
- Low-level actions are delegated to the operating system (OS).
    - Why shouldn’t we do the same for embedded devices?

© MicroEJ 2025

VEE PORT

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

© MicroEJ 2025

VEE PORT AND ABSTRACTION

APPLICATION FEATURES ARE SPLIT IN 2 CATEGORIES

1. Hardware dependent features (ex: screen): into the VEE Port, hiding details of what **might change**
2. Hardware-independent features:
   - Mathematical algorithms
   - Software using the VEE Port functionalities
   - UI
   - Connectivity protocols
   - Business logic

© MicroEJ 2025

PURPOSE OF ABSTRACTION

- Hardware abstracted software is the key point for **portability**
- Portability is needed when
  - You want to **reuse** the same code for several projects
  - Your hardware platform becomes **obsolete**
  - You target several hardware platforms with the same application
- When switching to a new hardware platform
  - You only change the hardware specific parts
  - You re-create an **iso-functional** computing VEE Port
  - Your software runs identically on this new VEE Port

© MicroEJ 2025

Firmware Build Flow

Build flow explained

© MicroEJ 2025

OVERVIEW

MICROEJ SDK6 includes:

- **MICROEJ IDE plugin**, available for IntelliJ IDEA/Android Studio and Eclipse.
- Gradle plugins and tools to **build applications.**
- **Add-on libraries** to code with Managed Code languages such as Java.
- Tools to **build VEE Ports, Foundation Libraries and Mocks.**
- Native libraries and mechanism to allow developers to use C and to create **interactions between C and Java** features.

© MicroEJ 2025

MICROEJ ARCHITECTURE

- A **MICROEJ Architecture** is a software package that includes the MEJ32 port to a target instruction set and a C compiler, MICROEJ Foundation Libraries and the MEJ32 Simulator.
- MICROEJ Architectures are provided by MICROEJ.
- Example of MICROEJ Architectures:
  - ARM Cortex-M4 – IAR 9.0
  - ARM Cortex-M7 – GCC 8.4
  - Renesas RXv2 - IAR 8.0.
  - ARM Cortex-A7 - GCC 5.3 Linaro Linux HardFP.
- List of the Architectures:
  - https://developer.microej.com/mej32-embedded-runtime-architectures/
- Available evaluation Architectures can be found here:
  - https://repository.microej.com/modules/com/microej/architecture/

© MicroEJ 2025

MICROEJ VEE PORT

- A **MICROEJ VEE Port** is a port of a MICROEJ Architecture for a specific hardware, RTOS and BSP.
- They are distributed as source (including C sources) or binary (pre-built C BSP).
- Example of MICROEJ VEE Port:
  - NXP i.MX RT1170 – FreeRTOS – MCUXPresso SDK.
  - STM32F746-DK – Zephyr – STM32CubeIDE.
  - NXP i.MXRT1170 – Linux.
- List of VEE Port examples:
  - https://developer.microej.com/supported-hardware/

© MicroEJ 2025

BUILD FLOW / VEE PORT

Legend

- Major Input or Output
- Managed Code
- Native Code
- Configuration
- Build Tool
- Sim Tool

bin MICROEJ VEE Architecture (MEJ32 Core + Built-in Foundation Libraries + Simulator + Mockups)

M N

src Managed Code

src Native Code

src Simulator Mocks

(e.g., .java, .list, resources)

(e.g., .c, .h)

(e.g., mock*.jar)

conf VEE Port Configuration

conf Test Suite Configuration

Managed Code Compiler (e.g., JDT)

Managed/Native Code Compilers

(.xml, .launch)

(runtime.a, .jar, .h)

(e.g., .jar)

(e.g., .o, .jar)

MICROEJ SDK (Gradle Plugin)

(built VEE Port)

bin **MICROEJ VEE Port**

- MEJ32 Core
- Foundation Libraries
- Simulator + Mocks
- Scripts + Tooling + Documentation

BUILD FLOW / EXECUTABLE

bin — MICROEJ VEE Port

src — Application

bin — Libraries

src — C Code Abstraction Layer + VEE Startup

bin — C Libraries Legacy C Libraries

(e.g., .java, .list, resources)

Managed Code Compiler (e.g., JDT)

(.a, .jar)

(e.g., .jar)

(e.g., .jar)

(.c, .h)

MICROEJ SDK (SOAR + Gradle Plugin)

C Compiler

(runtime.a, microejapp.o, .s)

(.o)

(.a)

Third Party ELF Linker

(application.out, application.hex, …)

bin — **Executable**

MEJ32 Core

Application

Abstraction Layer + C Libraries

Legend

- Major Input or Output
- Managed Code
- Native Code
- Build Tool

VEE Port Project Overview

For NXP i.MX RT1170 Evaluation Kit

© MicroEJ 2025

VEE PORT SOURCE PROJECT STRUCTURE

```
.
├── apps
│
├── bsp
│   ...
│   ├── mcux-sdk
│   └── vee
│
├── vee-port
│   ├── build.gradle.kts
│   ├── configuration.properties
│   │
│   ├── extensions
│   │   ├── front-panel
│   │   ├── image-generator
│   │   └── microui
│   │
│   ├── mock
│   │
│   └── validation
│       ├── core
│       ├── ...
```

Ready to use demo applications

Board Support Package (BSP) source code:
- mcux-sdk: NXP I.MXRT1170 SDK
- vee: implementation of MicroEJ VEE abstraction layers for NXP I.MXRT1170

Project used to configure and build the VEE Port:
- build.gradle.kts: declares VEE Port components (Networking, GUI, FS, …)
- configuration.properties: configures VEE Port components.

Extensions: related to VEE Port components used
- front-panel: describes the UI of the Simulator
- image-generator: off-board tool generating images compatible with NXP i.MX RT1170 GPU
- microui: MicroUI configuration files

Mock project specific to NXP i.MX RT1170 (allows to mock native code)

Validation project used to validate the integration of MICROEJ VEE on NXP i.MX RT1170

© MicroEJ 2025

Application

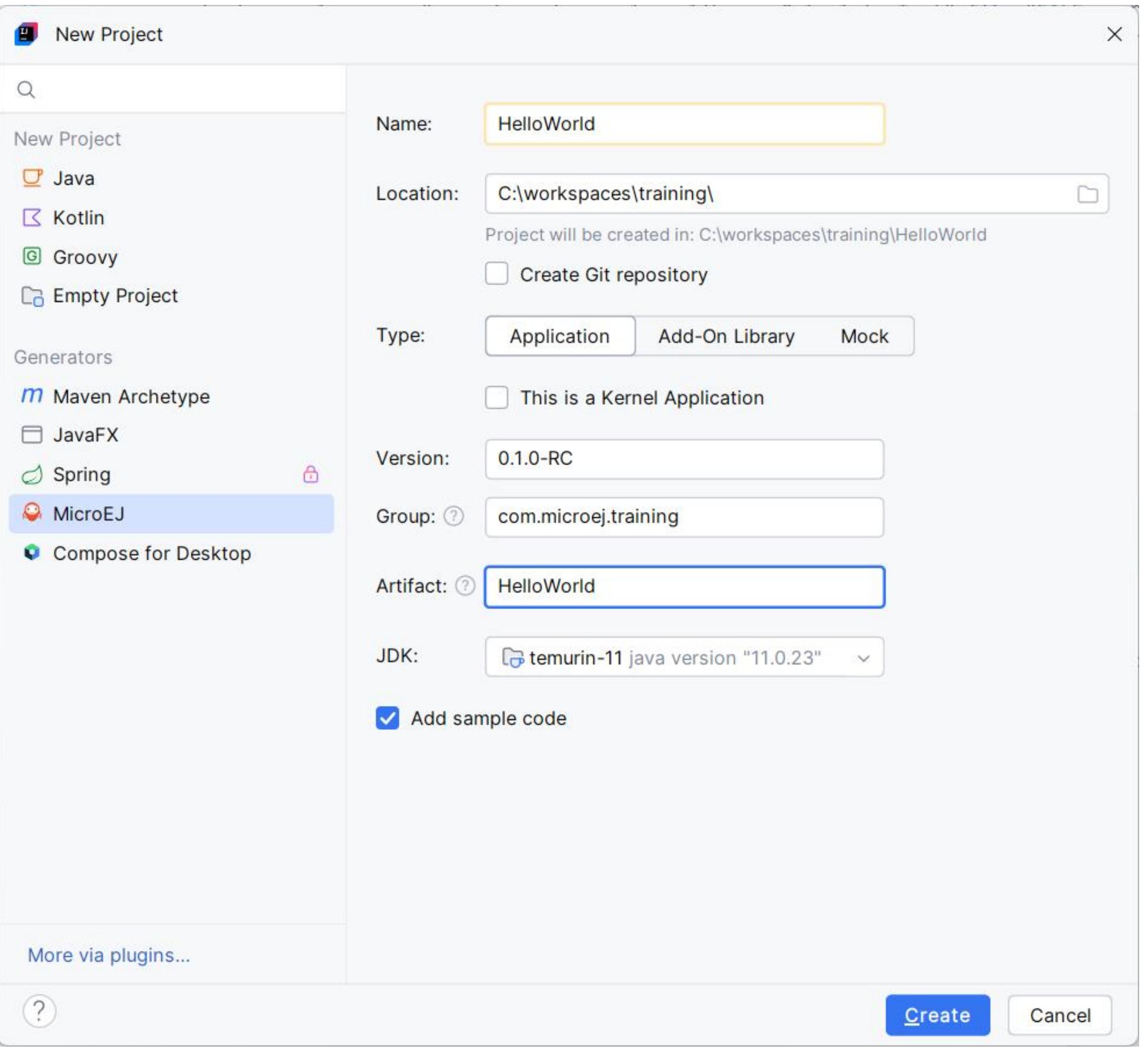
Build & Run a Hello World Application

Project Creation

Create a Hello World Application

APPLICATION PROJECT CREATION

- Go to **File** -> **New** -> **Project…**
- Go to **MicroEJ** category,
- Change the **Name** and the **Project Location** if needed,
  then click on **Create**.
- The project opens in a new IntelliJ IDEA window.



PROJECT STRUCTURE

```
HelloWorld  C:\workspaces\training\HelloWorld
> .gradle
> .idea
> build                                  <- Build outputs directory
v configuration
    common.properties                    <- Application options
> gradle                                 <- Gradle configuration directory
v src                                    <- Source directory
  v main
    v java
      v com.microej.training.helloworld
          Main                           <- Application Main class
      resources
  > test
  .gitignore
  build.gradle.kts                       <- Gradle build script file
  CHANGELOG.md
  gradlew                                ] <- Gradle Wrapper launcher scripts
  gradlew.bat                            ]
  LICENSE.txt
  README.md
  settings.gradle.kts                    <- Gradle settings script file
```

© MicroEJ 2025

VEE PORT SELECTION (1/3)

Building or running an Application / Library or Test Suite requires a VEE Port.

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

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

- Get the path to the **NXP i.MX RT1170 VEE Port** (e.g. C:\workspaces\training\nxpvee-mimxrt1170-evk\nxpvee-mimxrt1170-evk)
- Add the path to the VEE Port in the **settings.gradle.kts** file of the **HelloWorld** project:

```
rootProject.name = "HelloWorld"

includeBuild("C:\\workspaces\\training\\nxpvee-mimxrt1170-evk\\nxpvee-mimxrt1170-evk")
```

- In the **build.gradle.kts** file of the **HelloWorld** project, add the dependency to the VEE Port:

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

    //Uncomment the microejVee dependency to set the VEE Port or Kernel to use
    microejVee("com.nxp.vee.mimxrt1170:vee-port:3.0.0")
}
```

VEE PORT SELECTION (2/3)

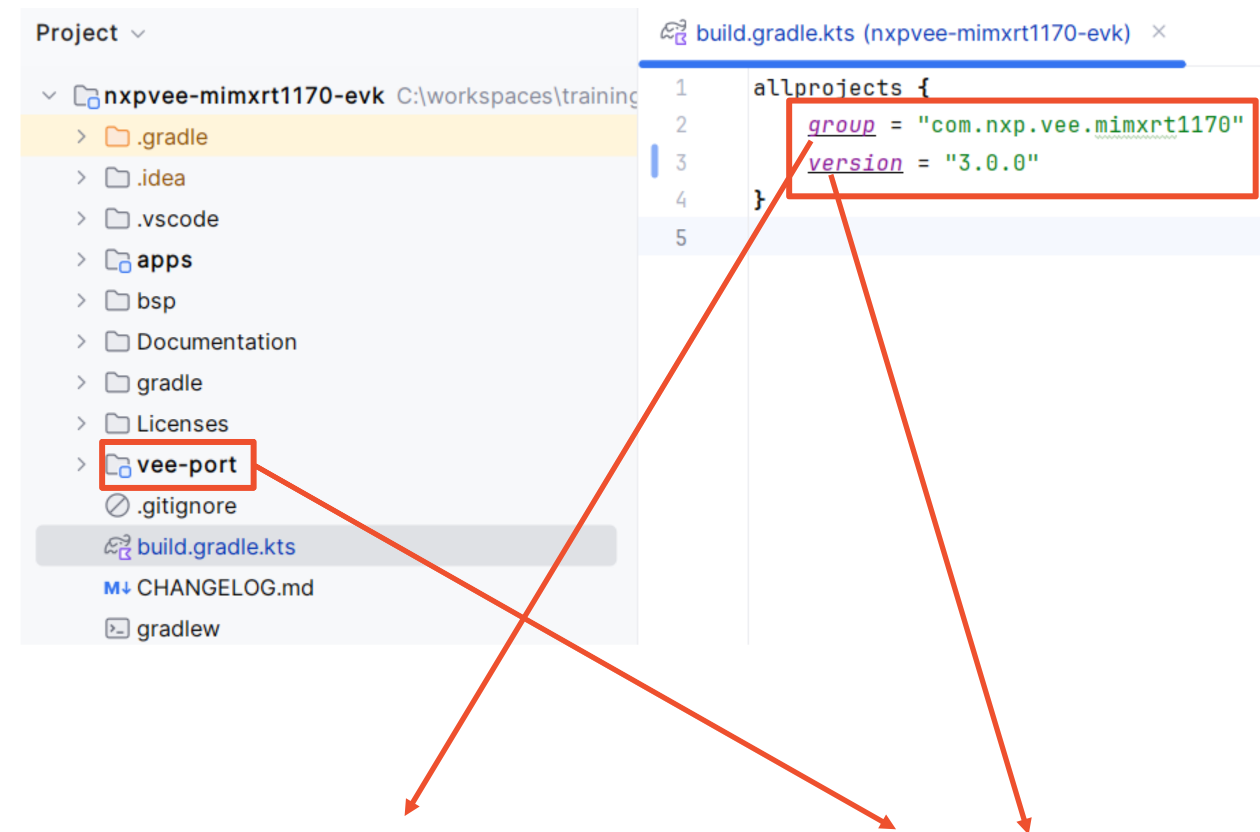
HOW TO KNOW THE MICROEJVEE MODULE NAME?

A Gradle module notation is composed of:

- A Group name (e.g. com.microej.example)
- A Module name (e.g. HelloWorld)
- A Version (e.g. 3.0.0)

The VEE Port **Group and Version** properties are defined in the **nxpvee-mimxrt1170-evk/build.gradle.kts** file.

By default, the Module name is the subproject folder name (here **vee-port**).
It can also be overridden in the **settings.gradle.kts** file of the VEE Port.

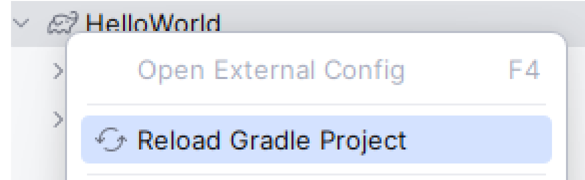


microejVee("com.nxp.vee.mimxrt1170:vee-port:3.0.0")

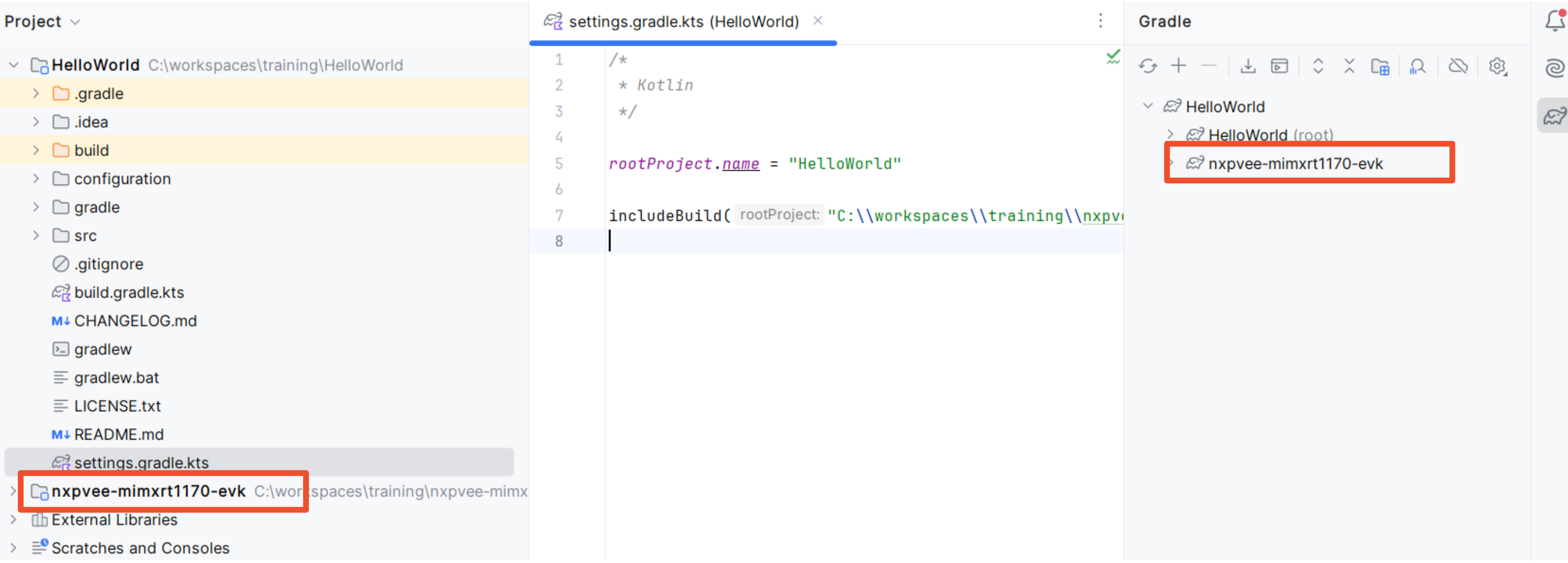
Note here that we display the VEE Port in an IDE window, the VEE port is not available yet in your IDE window (the next slide will take care of that). You could find the same information by browsing the file system.

VEE PORT SELECTION (3/3)

- Right-click on the **Hello World** project in the Gradle task view
- Click on Reload Gradle Project:



- The **NXP i.MXRT1170 VEE Port** project sources are now available in the Hello World project window:



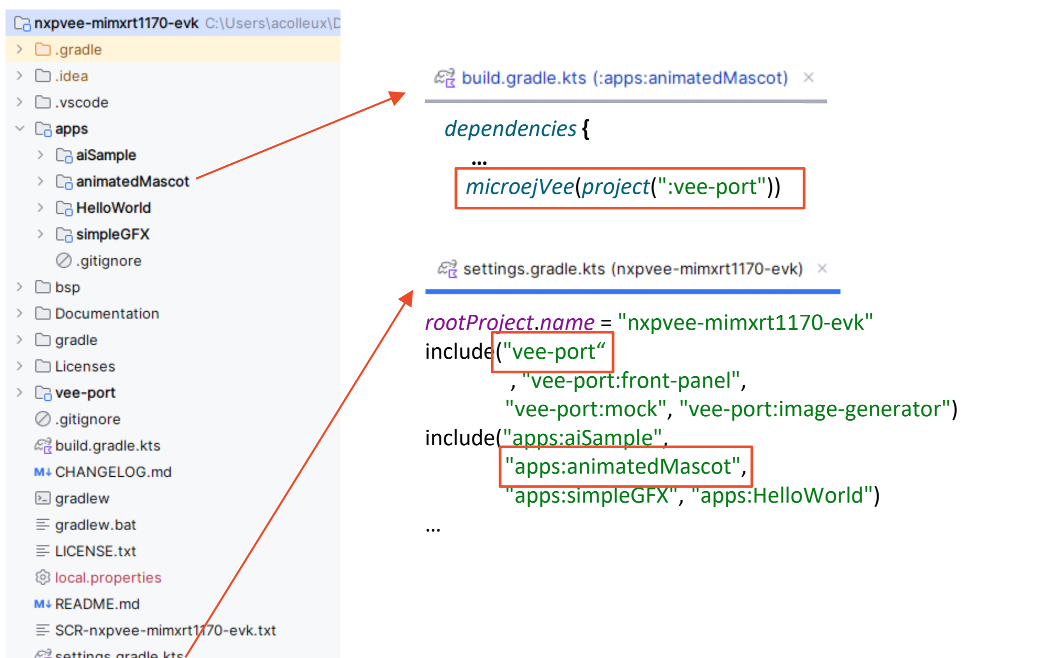
VEE PORT SELECTION WAYS

The Select a VEE Port documentation describes the different ways of including a VEE Port.

This training demonstrates 2 ways of including a VEE Port:

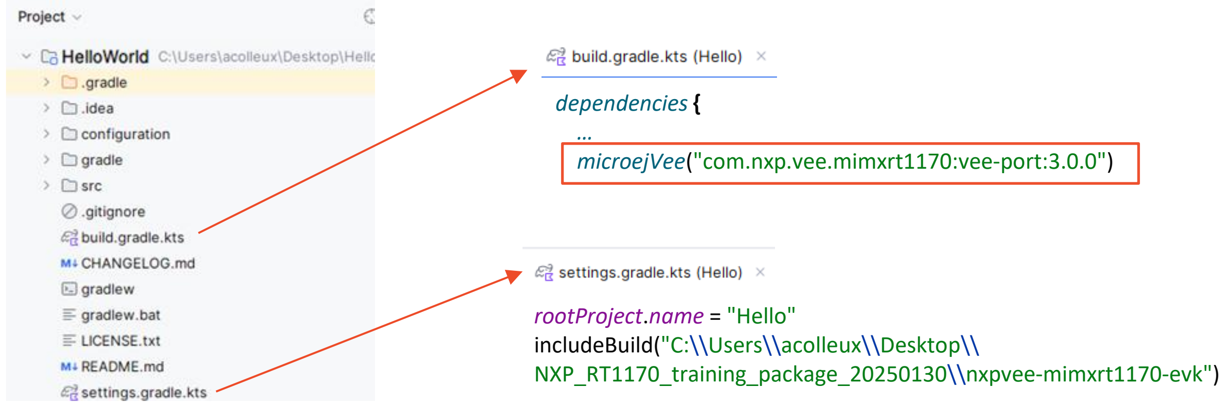
VEE Port project inside a multi-project

Proceed as follows when the Application and the VEE Port are in the **same project / folder**.
Example with the **animatedMascot** application provided **inside** the VEE Port project:



VEE Port project outside a multi-project

Proceed as follows when the Application and the VEE Port are in **separate projects / folders**.
Example with the **HelloWorld** application located **outside** of the VEE Port project:



A Gradle **multi-project** is a project that contains several projects within a single Gradle build.
A root project contains and configures subprojects.
A VEE Port project is multi-project as it contains multiple projects (see the root settings.gradle.kts).

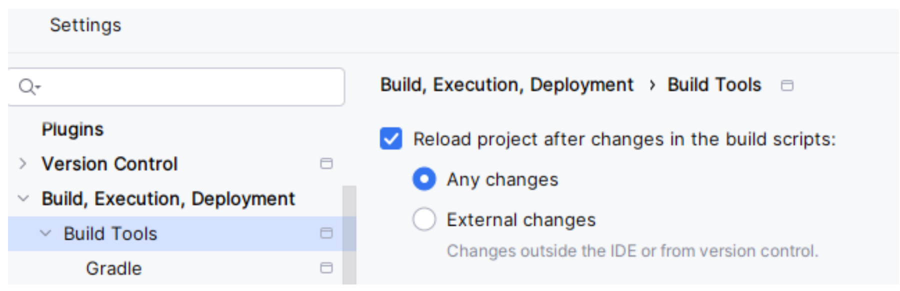
AUTOMATICALLY RELOAD A GRADLE PROJECT

By default, regardless of the IDE that you are using (IntelliJ IDEA, Android Studio or Eclipse), the reload of a Gradle project must be explicitly triggered by the user when the configuration of the project has changed.

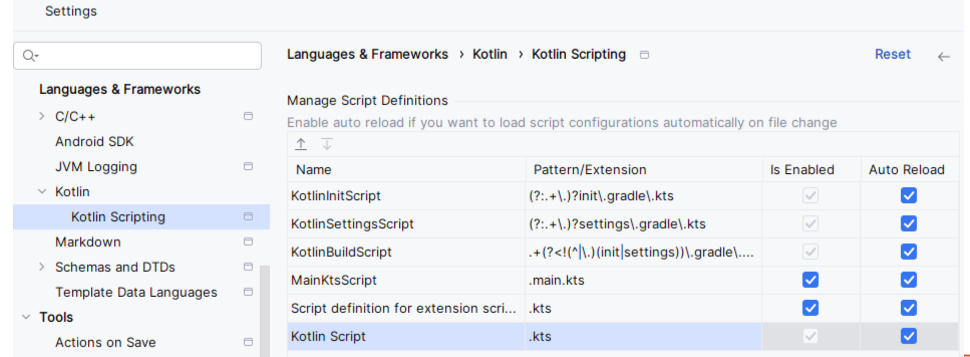
This allows to avoid reloading the project too frequently, but the user must not forget to manually reload the project to apply changes.

The auto-reload of a Gradle project with IntelliJ IDEA / Android Studio can be enabled as follows:

- Click on **File > Settings...**.
- Go to **Build, Execution, Deployment > Build Tools**.
- Check the **Reload changes in the build scripts** option and check the **Any changes** option.



- Go to **Languages & Frameworks > Kotlin > Kotlin Scripting**.
- Check all the **Auto Reload** options.



Run on the Simulator

Run the Hello World
Application on the Simulator

© MicroEJ 2025

RUN THE APPLICATION ON THE SIMULATOR

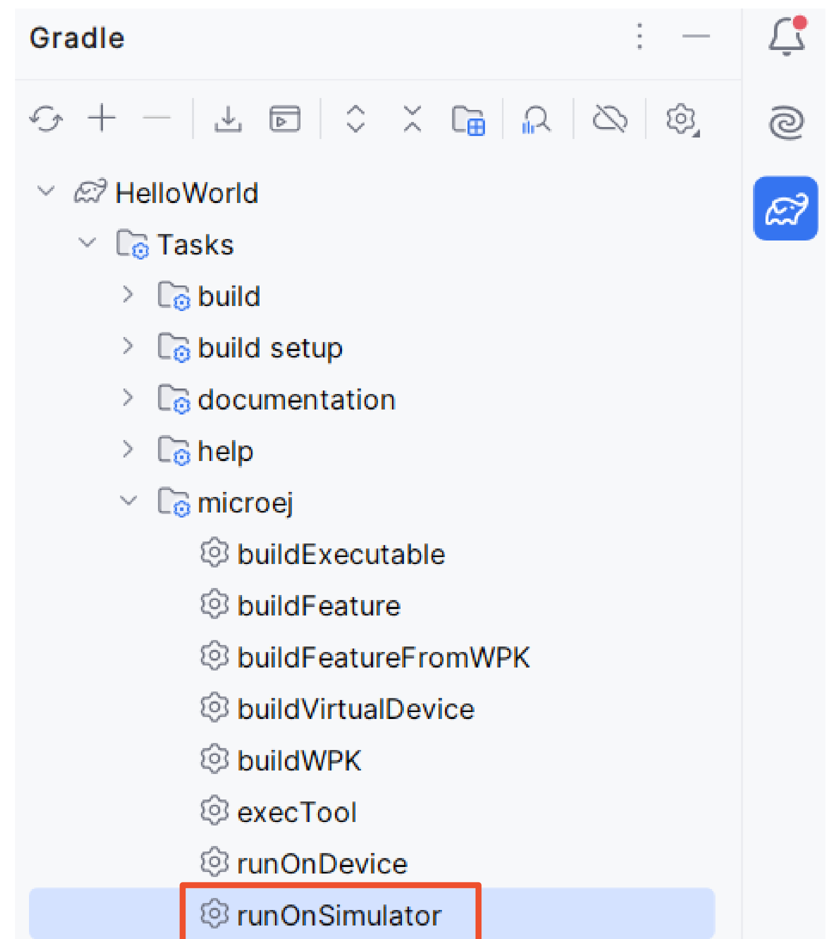
In the IntelliJ IDEA **Hello World project** window:

- Open the **Gradle** view.
- Open **Tasks > microej** and double-click on **runOnSimulator**.
- Or use the following command line:
  `./gradlew runOnSimulator`

The application will be run in the simulator. The following traces will be printed in the console:

```
=============== [ Launching on Simulator ] ===============
Hello World!
=============== [ Completed Successfully ] ===============
> Task :myapplication:runOnSimulator

BUILD SUCCESSFUL in 4s
5 actionable tasks: 5 executed

Build Analyzer results available
6:22:15 PM: Execution finished 'runOnSimulator'.
```

Note: the VEE Port is automatically built when building or running the application.



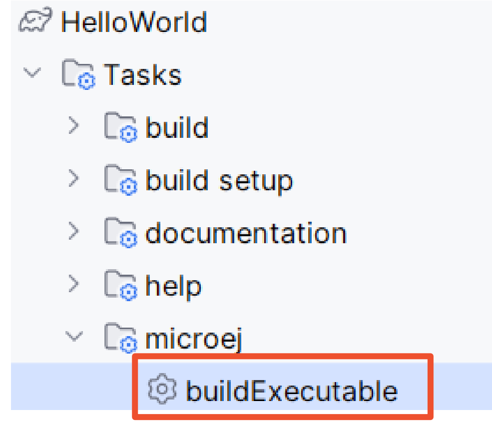
Run on the Device

Run the Hello World
Application on the Device

© MicroEJ 2025

BUILD THE APPLICATION EXECUTABLE

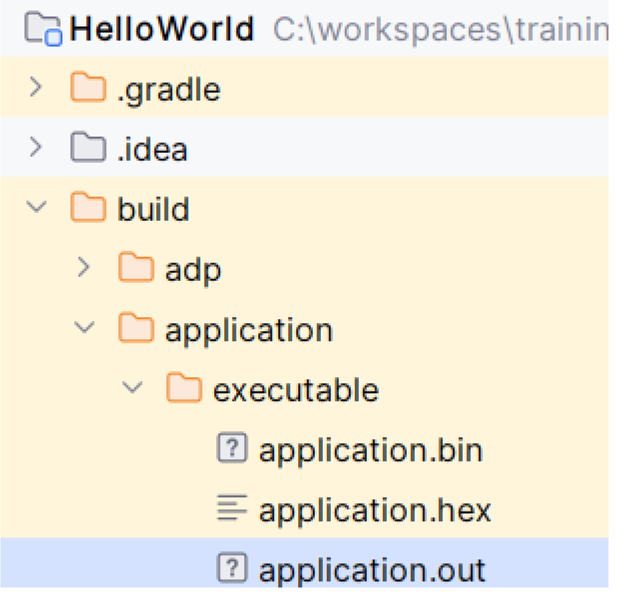
- Open the **Gradle** view.
- Open **Tasks > microej** and double-click on **buildExecutable:**



- Or use the following command line: `./gradlew buildExecutable`
- The following output can be seen in the console:

```
C:\workspaces\training\nxpvee-mimxrt1170-evk\nxpvee-mimxrt1170-evk\bsp\projects\nxpvee-ui\sdk_makefile>IF
C:\workspaces\training\nxpvee-mimxrt1170-evk\nxpvee-mimxrt1170-evk\bsp\projects\nxpvee-ui\sdk_makefile>cd
Execution of script 'C:\workspaces\training\nxpvee-mimxrt1170-evk\nxpvee-mimxrt1170-evk\bsp\projects\micr
============== [ Completed Successfully ] ==============
> Task :app:buildExecutable

BUILD SUCCESSFUL in 3s
7 actionable tasks: 1 executed, 6 up-to-date
```

- The Application Executable is available in the **build/application/executable** folder of the Application project:



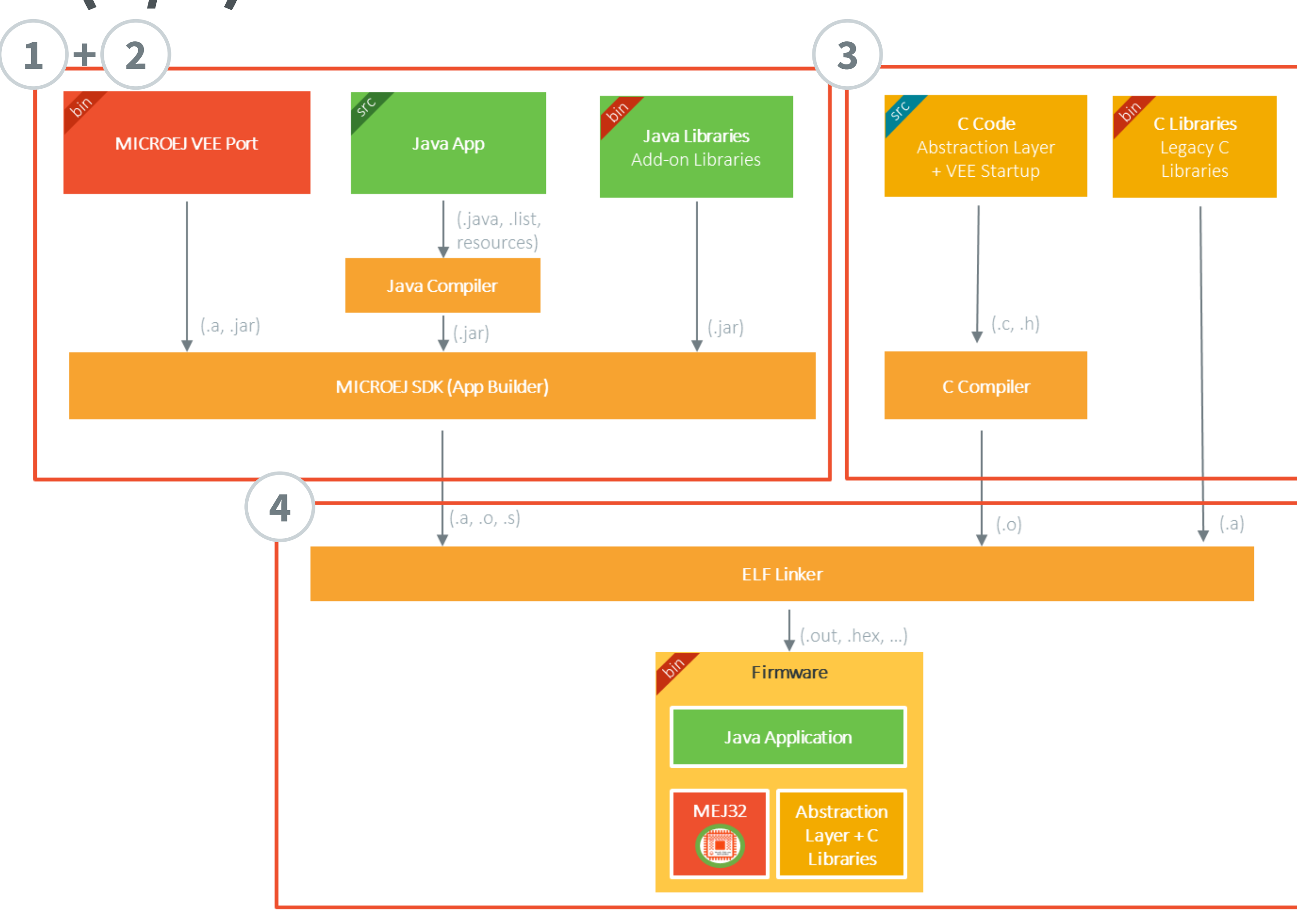
BUILD FLOW EXPLAINED (1/2)

- The Application must be **linked with the BSP:**
  - BSP = drivers + (optional: operating system) + abstraction layer.
  - Done by a 3rd party toolchain (Arm GNU, IAR, …).
- MicroEJ provides:
  - Application as an **object file (microejapp.o).**
  - MICROEJ VEE runtime environment as a **library file (microejruntime.a).**
  - **Header files** with types and functions provided by this library (.h).
  - Abstraction layer interface (.h).
  - Abstraction layer implementation (.c, .cpp).
- 3rd party toolchain is responsible for **compiling the BSP, linking, and generating the Executable file.**

© MicroEJ 2025

BUILD FLOW EXPLAINED (2/2)

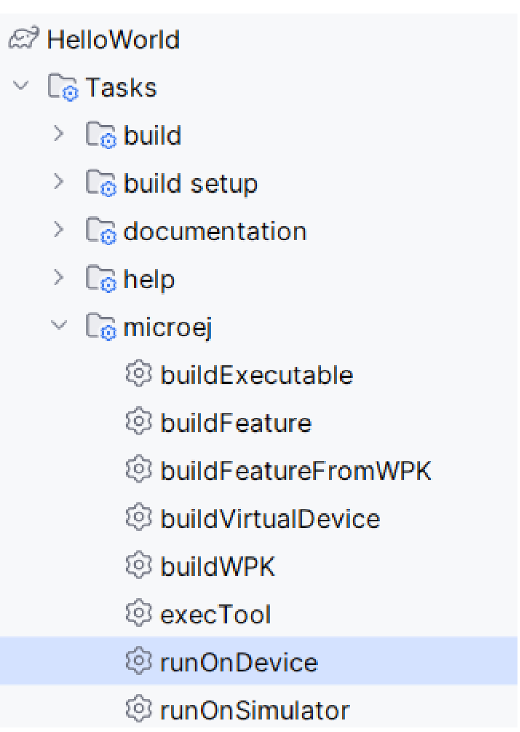
- The following steps are performed when running the **buildExecutable** task:
  1. The Application is built as an object file (**microejapp.o**).
  2. MICROEJ VEE runtime environment (**microejruntime.a**) and header files (***.h**) are deployed in the BSP project.
  3. A build script compiles the BSP using the BSP 3rd party toolchain (**Arm GNU, IAR, …**)
  4. The 3rd party toolchain is used to link the BSP sources with **microejapp.o** and **microejruntime.a.**



RUN THE APPLICATION ON THE DEVICE (1/2)

FLASH THE FIRMWARE

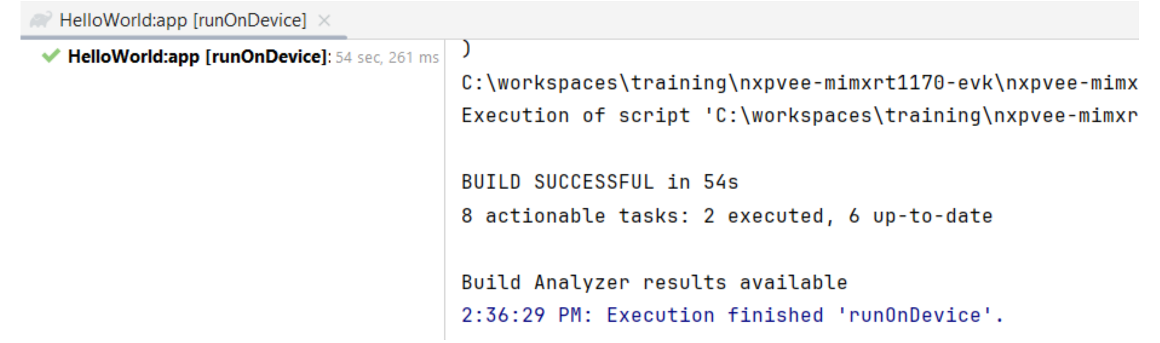
- Open the **Gradle** view.
- Open **Tasks > microej** and double-click on **runOnDevice:**



- Or use the following command line:

```
./gradlew runOnDevice
```

- The following output can be seen in the console:

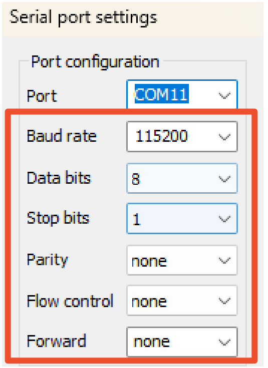


ℹ️ The **runOnDevice** task automatically triggers the **buildExecutable** task.

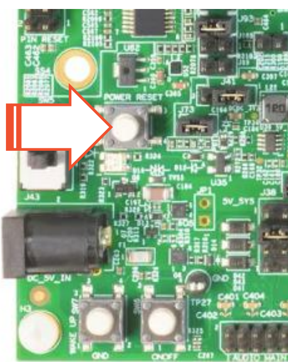
RUN THE APPLICATION ON THE DEVICE (2/2)

GET THE APPLICATION TRACES

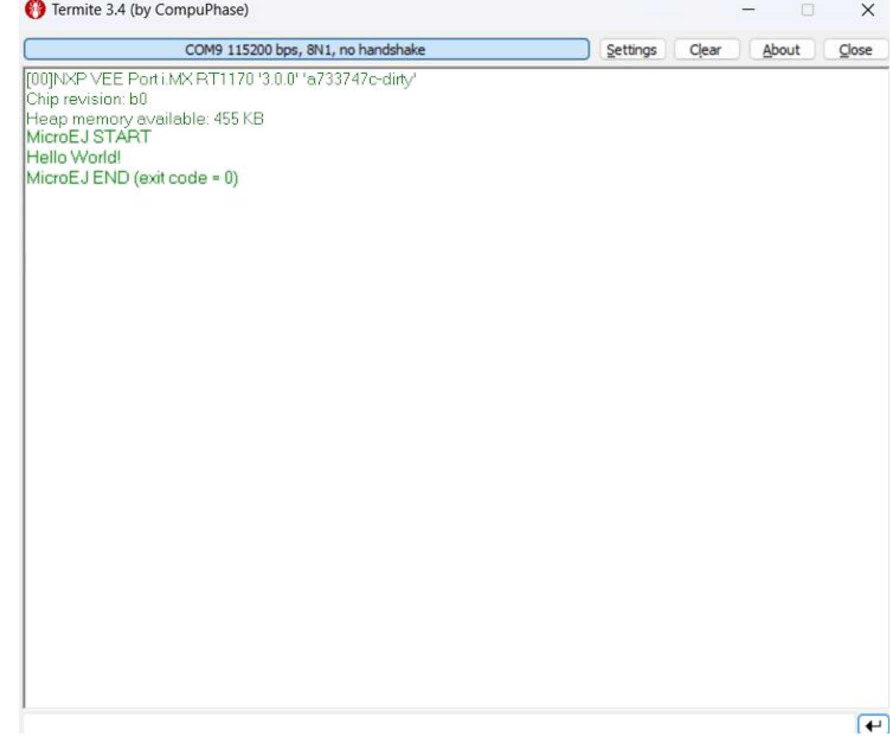
- Open the Termite serial terminal.
- Click the **Settings** button:
  - Select the NXP i.MX RT1170 EVK board COM port.
  - Set the following port parameters:



- Reset the NXP i.MX RT1170 EVK board using Reset button:



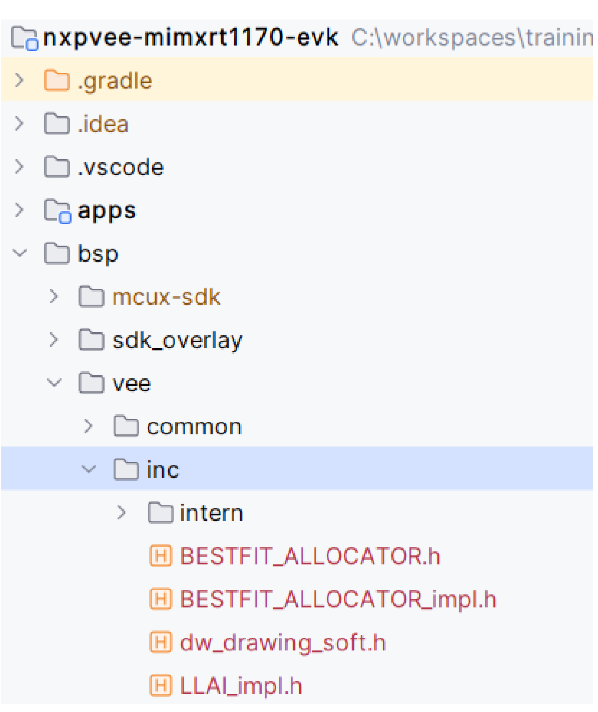
The application starts and the **Hello World** message is printed in the console!



© MicroEJ 2025

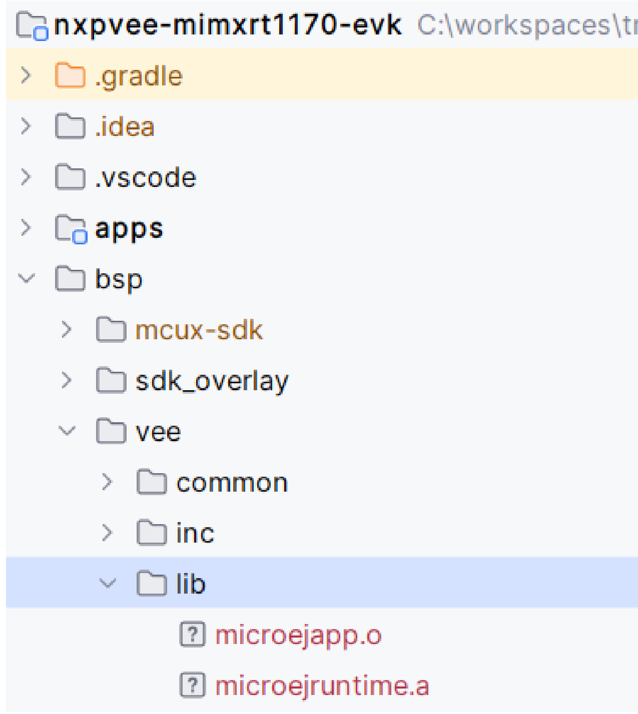
MICROEJ CORE ENGINE STARTUP

MicroEJ header files are located in: **bsp/vee/inc**



MicroEJ libraries and Managed Code application object file are used during link edition.

Those files are located in: **bsp/vee/lib**



MicroEJ Core Engine is invoked in: **bsp/vee/port/core/src/microej_main.c** with **SNI_createVM():**

```
void microej_main(int argc, char **argv) {
    void* vm;
    int32_t err;
    int32_t exitcode;

    // create VM
    vm = SNI_createVM();
```

Note: in the NXP i.MX RT1170 VEE Port, **microej_main()** is called from a FreeRTOS task in **main.c.**
It is also possible to run MicroEJ Core Engine on a bare metal device (no RTOS).

Application Project Configuration

© MicroEJ 2025

PROJECT BUILD FILE

build.gradle.kts

```
plugins {
    id("com.microej.gradle.application") version “1.0.0"
}

group = “com.example"
version = "0.1.0-RC"

microej {
    applicationEntryPoint = "com.example.helloworld.Main"
}

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

    microejVee("com.nxp.vee.mimxrt1170:vee-port:3.0.0")
}
```

MicroEJ Gradle plugin for an Application

Project group and version

MicroEJ Configuration block

Application main class

Application dependencies:

- “ej.api:edc”: provides a subset of java.lang APIs
- VEE Port to use (e.g. NXP i.MXRT1170 VEE Port)

LIBRARY DEPENDENCIES

The **build.gradle.kts** file of the project contains a description of all the libraries required by the application:

```
dependencies {
    implementation("ej.api:edc:1.3.5")
    implementation("ej.api:microui:3.4.0")
}
```

These dependencies are automatically fetched by Gradle during the build process.

Available MICROEJ libraries can be found here:

- Central Repository (https://repository.microej.com/): 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.
- Developer Repository (https://forge.microej.com/artifactory/microej-developer-repository-release/): 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.

From the MICROEJ Javadoc you can search for a Class and get the Gradle dependency that provides it by visiting https://repository.microej.com/javadoc/microej_5.x/apis/index.html

© MicroEJ 2025

GET LIBRARY DEPENDENCY

Example :

https://repository.microej.com/javadoc/microej_5.x/apis/index.html?ej/restserver/RestServer.html This button let you copy the MMM dependency directly into the clipboard.

All Classes

Packages

android.net
com.google.protobuf
com.microej.http.util
com.microej.kf.util
com.microej.kf.util.message
com.microej.kf.util.observable
com.microej.kf.util.security
com.microej.nls
com.microej.soar

All Classes

AbstractAllTestClassesWrapper
AbstractChronology
AbstractCollection
AbstractContent
AbstractDelegateHttpsURLConnection
AbstractJSONExpr
AbstractKFList
AbstractList
AbstractMap
AbstractMap.SimpleEntry
AbstractMap.SimpleImmutableEntry
AbstractMessageLite
AbstractMessageLite.Builder
AbstractMethodError
AbstractPackedMap
AbstractParser
AbstractQueue
AbstractRemoteService
AbstractResource
AbstractSequentialList
AbstractSet
AbstractTestWrapper
AbstractWeakPackedMap
AcceptEncoding
AccessPoint

OVERVIEW PACKAGE CLASS TREE INDEX HELP

Use this class in your project

PREV CLASS NEXT CLASS FRAMES NO FRAMES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

In order to use this class in your project, add this dependency in your project build file:

SDK 6 (build.gradle.kts)

```
implementation("ej.library.iot:res
```

SDK 5 (module.ivy)

```
<dependency org="ej.library.iot" n
```

ej.restserver

Class RestServer

java.lang.Object
HTTPServer
ej.restserver.RestServer

```
public class RestServer
extends HTTPServer
```

A simple implementation of HTTP server capable of easily exposing endpoints.

REST servers can always handle requests that target endpoints. Other kind of request are handles by custom request handlers.

See Also:
EndpointHandler, RequestHandler

Constructor Summary

Constructors

Constructor and Description
RestServer(int port, int maxSimultaneousConnection, int jobCountBySession) Creates a new server bounded to given port.
RestServer(int port, int maxSimultaneousConnection, int jobCountBySession, ServerSocketFactory serverSocketFactory) Creates a REST server on top of the given HTTP server.

CHECK DEPENDENCIES UPDATE

To check if Gradle dependencies are up-to-date, you can use the Gradle Versions Plugin.

This Gradle plugin lists all the dependencies declared in the **build.gradle.kts** file, and tells whether they are up-to-date or if a new version is available.

Here is an example of report:

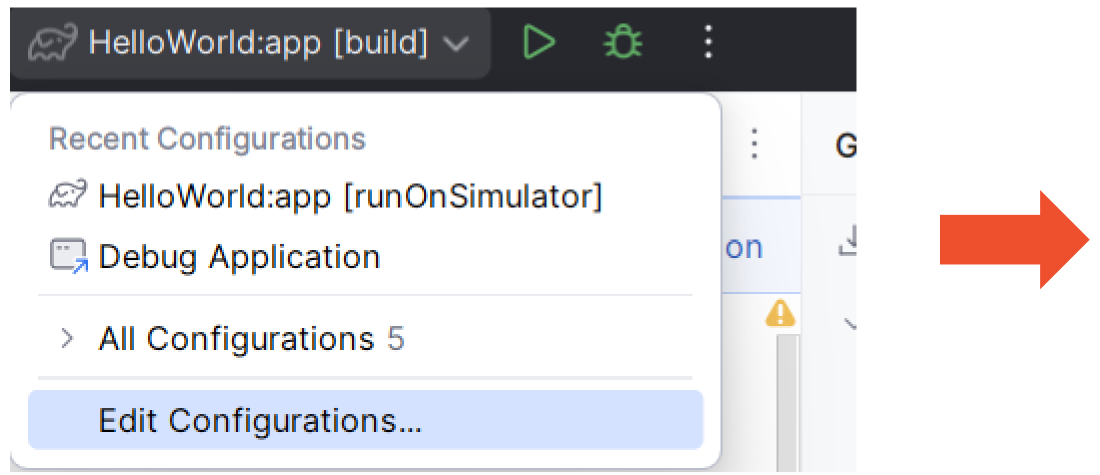
```
------------------------------------------------------------
: Project Dependency Updates (report to plain text file)
------------------------------------------------------------

The following dependencies are using the latest milestone version:
- com.github.ben-manes.versions:com.github.ben-manes.versions.gradle.plugin:0.51.0
- com.is2t.tools:application-repository:2.2.0



The following dependencies exceed the version found at the milestone revision level:
- ej.library.ui:widget [5.2.0 <- 4.2.0]

The following dependencies have later milestone versions:
- com.microej.gradle.application:com.microej.gradle.application.gradle.plugin [0.15.0 -> 0.19.0]
- com.microej.test:junit-test-engine [0.2.2 -> 0.3.0]
- ej.api:drawing [1.0.2 -> 1.0.5]
- ej.api:edc [1.3.5 -> 1.3.7]
- ej.api:microui [3.1.0 -> 3.5.0]
- ej.library.eclasspath:collections [1.4.0 -> 1.4.2]
- ej.library.runtime:basictool [1.5.0 -> 1.7.0]
- ej.library.runtime:service [1.1.1 -> 1.2.0]
- ej.library.test:junit [1.7.1 -> 1.10.0]

Gradle release-candidate updates:
- Gradle: [8.3 -> 8.10.2]
```

Follow the How To Check Dependencies Updates documentation to setup the plugin in your project.

RUN CONFIGURATIONS

When a Gradle has been executed once, a Run Configuration is created. It can be used for subsequent calls to the same task by clicking on the green arrow next to it: HelloWorld:app [runOnSimulator]

It can also be customized, for example to change the log level (--*info*) when executing the **runonSimulator** task:

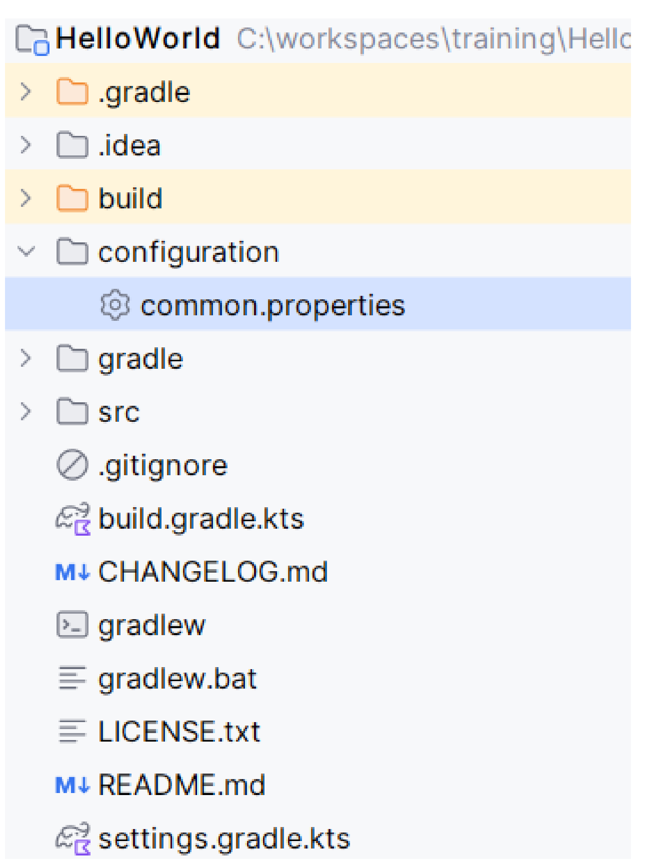


Or use the following command line:

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

APPLICATION CONFIGURATION

A MICROEJ Application can be configured by settings option properties in a properties file in a **configuration** folder in the project.



All the **.properties** files of this folder are considered, whatever their names are.

As an example, the Managed Code Heap size and the maximum number of threads can be defined with these properties:

```
core.memory.javaheap.size=8192
core.memory.threads.size=3
```

All the available options are described in the documentation.

© MicroEJ 2025

Front Panel

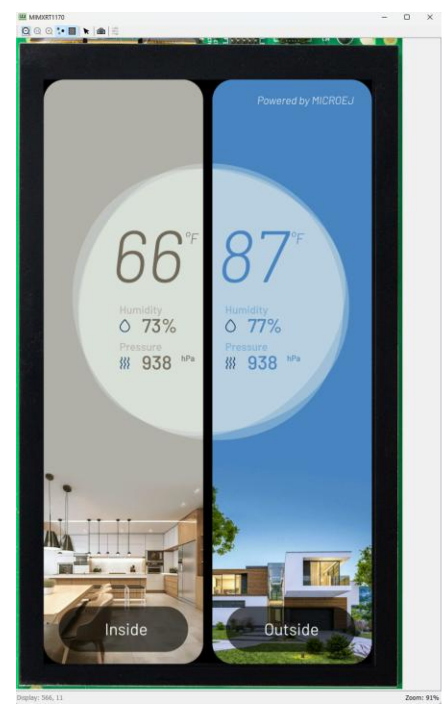
Customization

© MicroEJ 2025

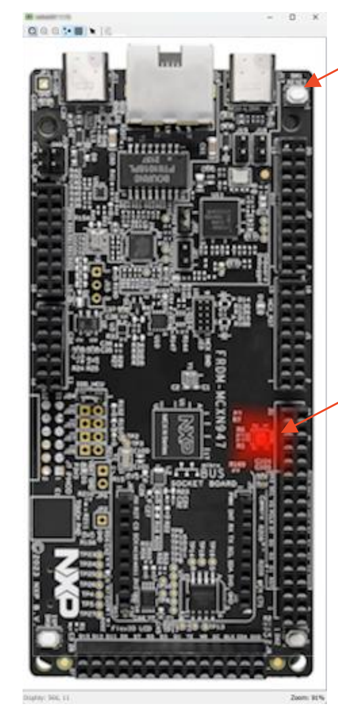
FRONT PANEL PRINCIPLE

- 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
- To make this possible for devices operated by the user, the Simulator must connect to a "mock" of the device's control panel (the "Front Panel")
- 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 is not necessarily showing a display.
  It can also be used to show a hardware device and simulate its peripherals (LEDs, buttons, …).

For more information, see Front Panel Overview.



Front Panel with a display



Front Panel with LEDs and Buttons, no display

© MicroEJ 2025

KEY FUNCTIONALITIES (1/4)

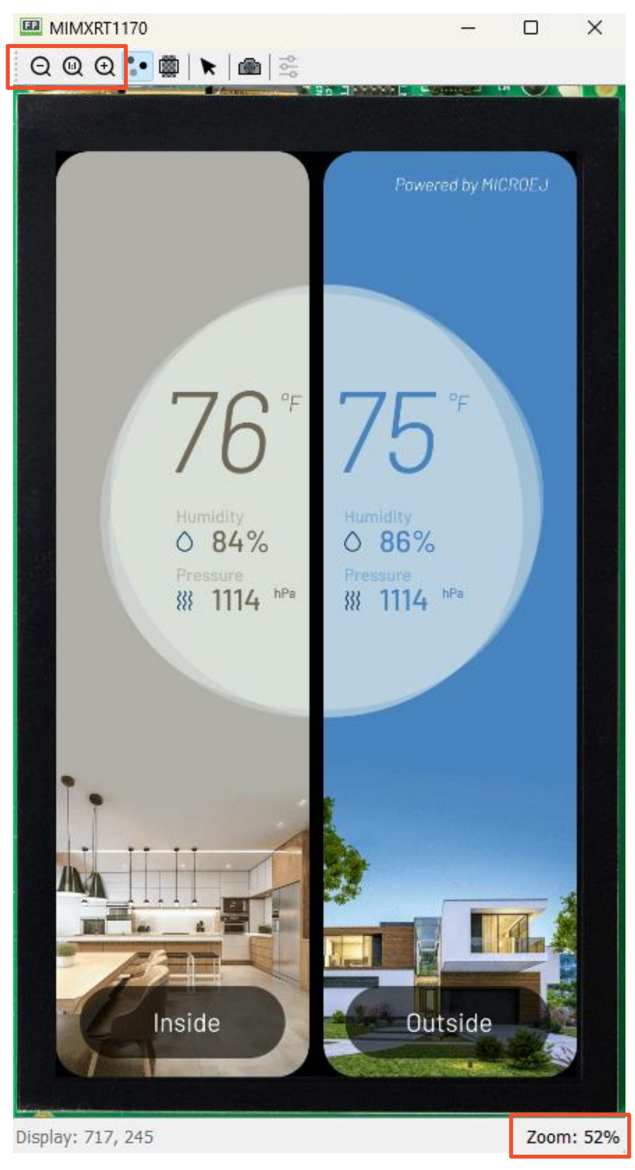
ZOOM

The Front Panel is able to zoom in or out the represented device.

The current value of the zoom is printed in the status bar.

There are three buttons in the toolbar to change the zoom:

- Zoom out by increment of 10%.
- Reset the zoom to 100%.
- Zoom in by increment of 10%.



© MicroEJ 2025

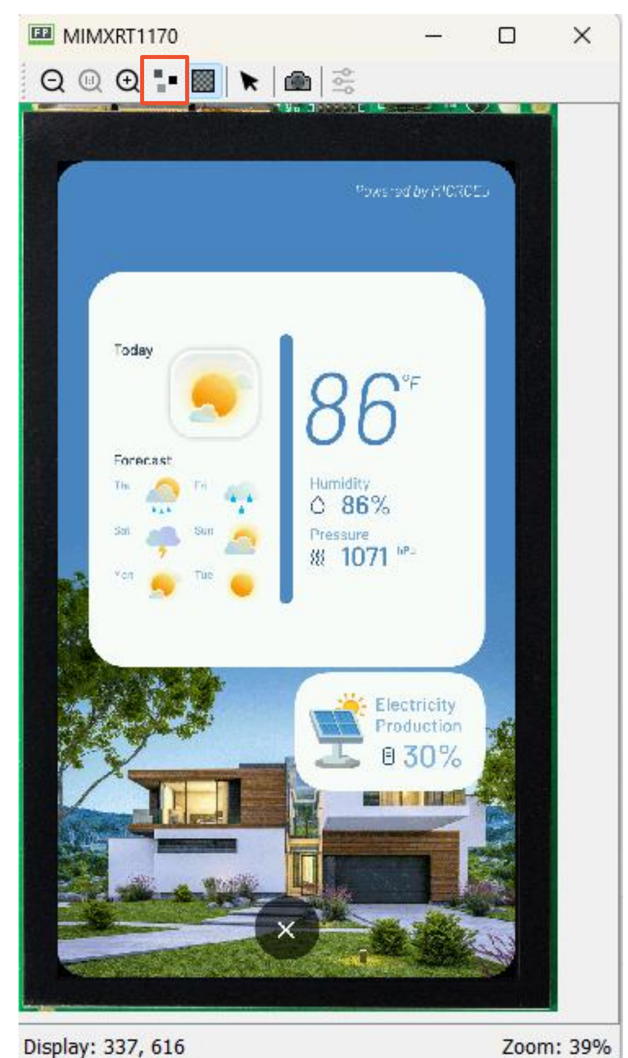
KEY FUNCTIONALITIES (2/4)

INTERPOLATION

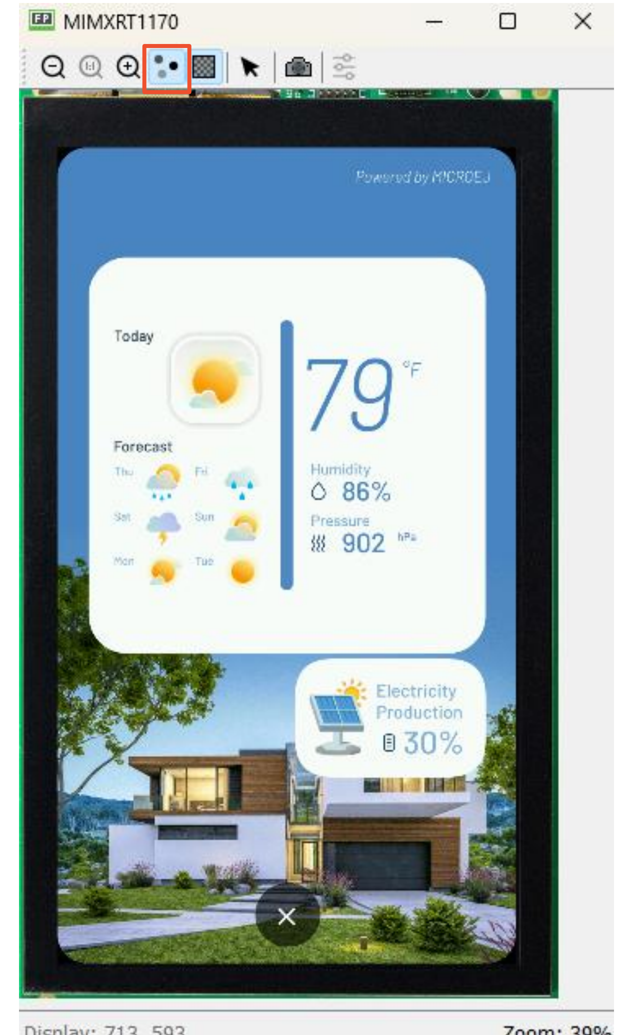
By default, the zoom is done without interpolation to ease the reading of the pixels drawn on the screen.

But it could be convenient to enable the interpolation when a great or small zoom is applied, to better read the strings for instance.

ℹ The interpolation is enabled by default on the NXP i.MX RT1170 Front Panel.



Interpolation disabled



Interpolation enabled

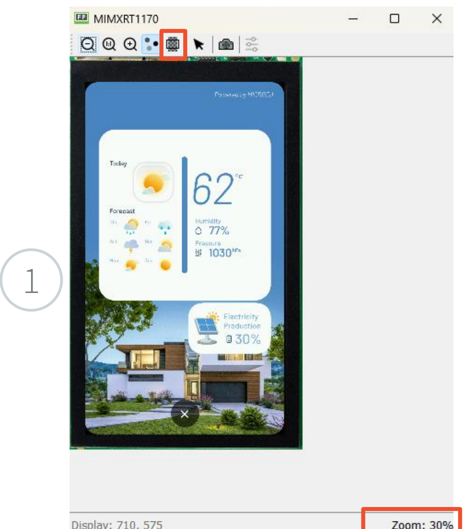
© MicroEJ 2025

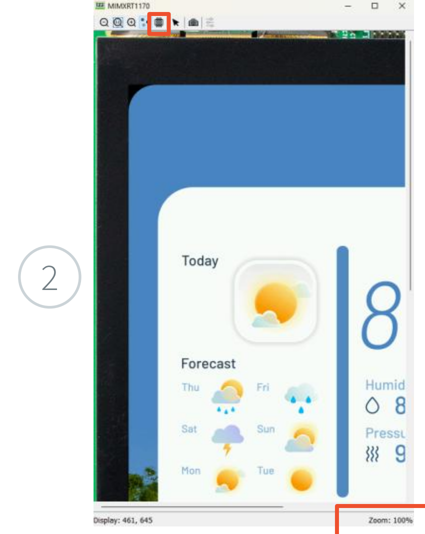
KEY FUNCTIONALITIES (3/4)

FIT

By default, the zoom and the window size are not related (▩)

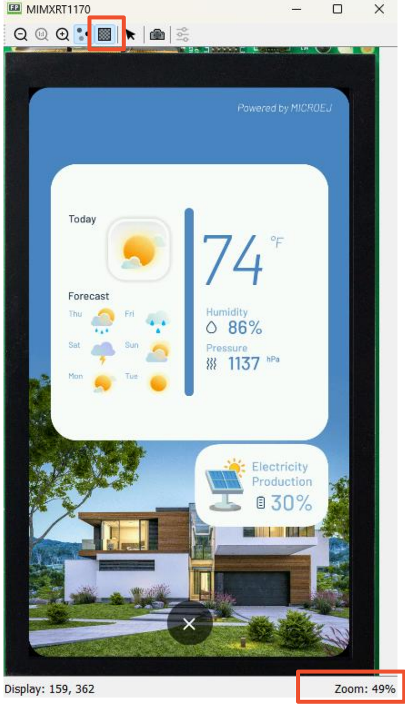
1. When the zoom is changed the window size does not change and scrollbars may appear to navigate in the device.
2. When the window size is changed, the zoom does not change.





In contrast, the zoom and the window can be linked together (▩)

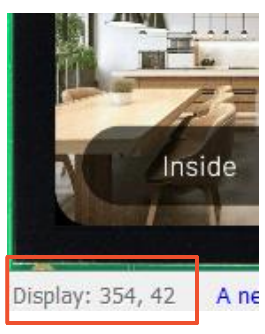
In this case, a modification of the zoom or the window size have an impact on the other.



KEY FUNCTIONALITIES (4/4)

DISPLAY COORDINATES

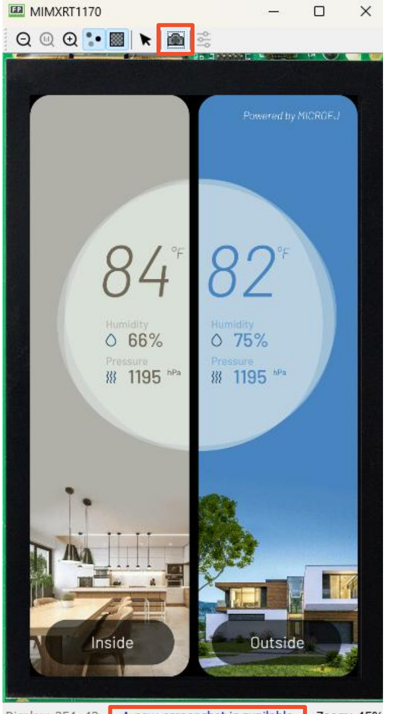
When the Front Panel contains a display, the display coordinates under the cursor are printed in the status bar:

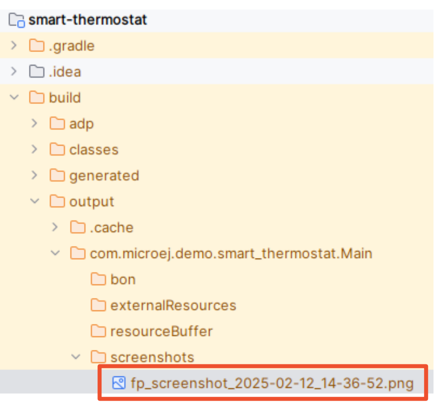


DISPLAY SCREENSHOT

When the Front Panel contains a display, a screenshot button allows to make a screenshot of the current content of the display (📷).

The screenshots are saved in the subfolder screenshots of the application output folder:





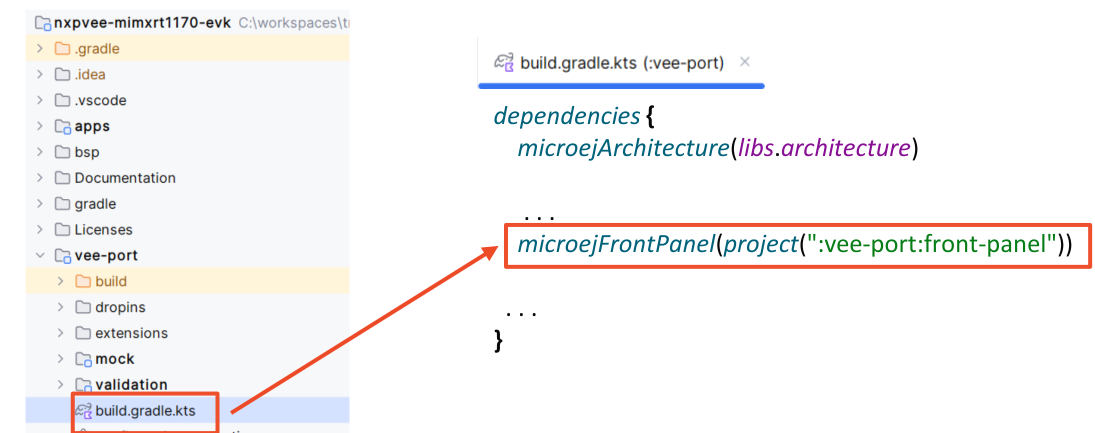
© MicroEJ 2025

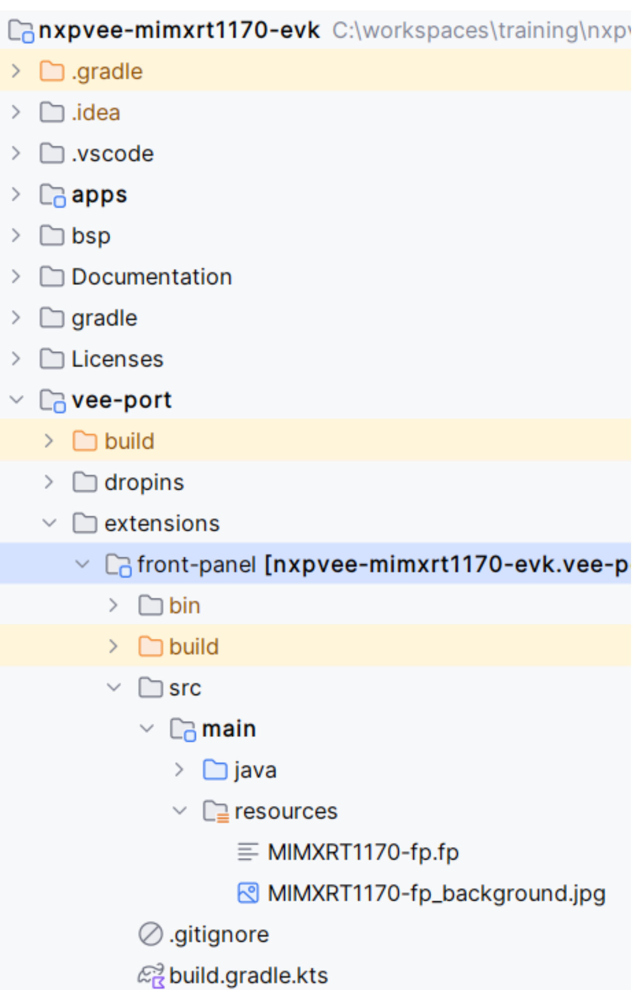
PROJECT STRUCTURE

A Front Panel project has the following structure and contents:

- **src/main/java (optional):** contains custom widgets and button event listeners.
- **src/main/resources:** holds files that define the contents and layout of the Front Panel (**.fp** file and images).

To use a Front Panel project in a VEE Port, include it in the VEE Port project dependencies:





FRONT PANEL DESCRIPTION FILE

- Description written in XML (.fp file): <**device** ...> element contains the elements that define the widgets that make up the Front Panel.
- Loaded by the Front Panel Engine to build the graphical representation of the real device.
- Declare the widgets that simulate the drivers, sensors, and actuators of the real device.

MIMXRT1170-fp.fp

```
1  <?xml version="1.0"?>
2  <!--
3      Front Panel
4  
5      Copyright 2022 MicroEJ Corp. All rights reserved.
6      This library is provided in source code for use, modification and test, subject to license terms.
7      Any modification of the source code will break MicroEJ Corp. warranties on the whole library.
8  -->
9  <frontpanel
10     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
11     xmlns="http://xml.is2t.com/ns/1.0/frontpanel"
12     xsi:schemaLocation="http://xml.is2t.com/ns/1.0/frontpanel .fp1.0.xsd">
13 
14     <device name="MIMXRT1170" skin="MIMXRT1170-fp_background.jpg">
15         <ej.fp.widget.Display x="63" y="91" width="720" height="1280"/>
16         <ej.fp.widget.Pointer x="63" y="91" width="720" height="1280" touch="true"/>
17     </device>
18 </frontpanel>
```

- Widgets:
    - The name of the widget element references the Java class of the widget (see widget-x.y.z.jar in Module Dependencies).
    - A widget can be identified by a label, which must be unique for the widgets of the same type.
    - Position specified with x and y attributes.

MICROEJ SDK DEVELOPMENT TOOLS

© MicroEJ 2025

MICROEJ SDK DEVELOPMENT TOOLS

MICROEJ SDK provides a large panel of development tools to accelerate product development.

Development tools are split in categories:

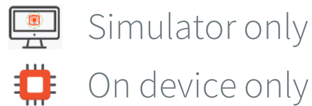
- Runtime & Post-Mortem Debugging Tools
- Memory Inspection Tools (debug memory corruption, leaks)
- Static Analysis Tools
- GUI Application Debugging Tools (bottlenecks identification, rendering issues)

In this training, only the **Stack Trace Reader** tool will be introduced.

Refer to the Development Tools training to learn more about them.

© MICROEJ 2025

DEVELOPMENT TOOLS OVERVIEW



TOOLS	RUNTIME & POST-MORTEM	MEMORY INSPECTION	STATIC ANALYSIS TOOLS	GUI DEBUGGING TOOLS
Core Engine VM Dump	×			
Debug on Device	×			
Debug on Simulator	×			
Port Qualification Tool (PQT)	×			
SystemView	×	×		×
Logging & Message Libraries	×			
Code Coverage	×			
Memory Map Analyzer		×		
Heap Dumper / Analyzer		×		
Heap Usage Monitoring		×		
Core Engine MEMORY integrity check		×		
SonarQube / Klocwork (Java/C)			×	
Null Analysis			×	
UI Flush Visualizer				×
UI MWT & Widget Debug Utilities				×

STACK TRACE READER (1/4)

EXCEPTION GENERATION

- By default, on error, the stack trace of the exception thrown is printed on the **serial console**.
- Let's generate an error. Add the following code in your HelloWorld main method:

```
byte[] array = new byte[5];
array[5] = 42; // Invalid access to the array
```

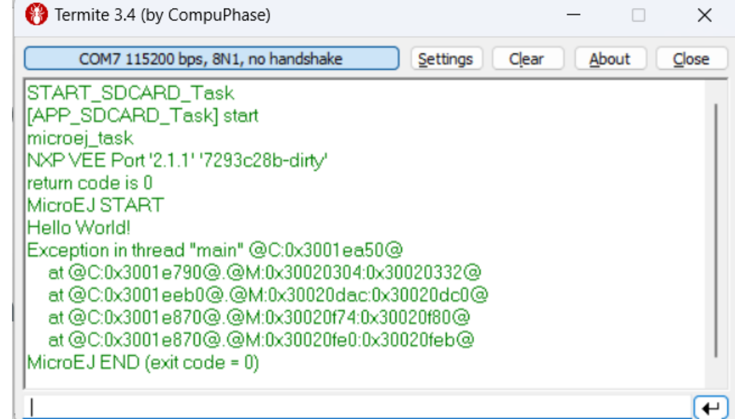
Build and Run the Application on the device:

- Open the **Gradle** view.
- Open **Tasks > microej** and double-click on **runOnDevice**.
- Or use the following command line:
  `./gradlew runOnDevice`

STACK TRACE READER (2/4)

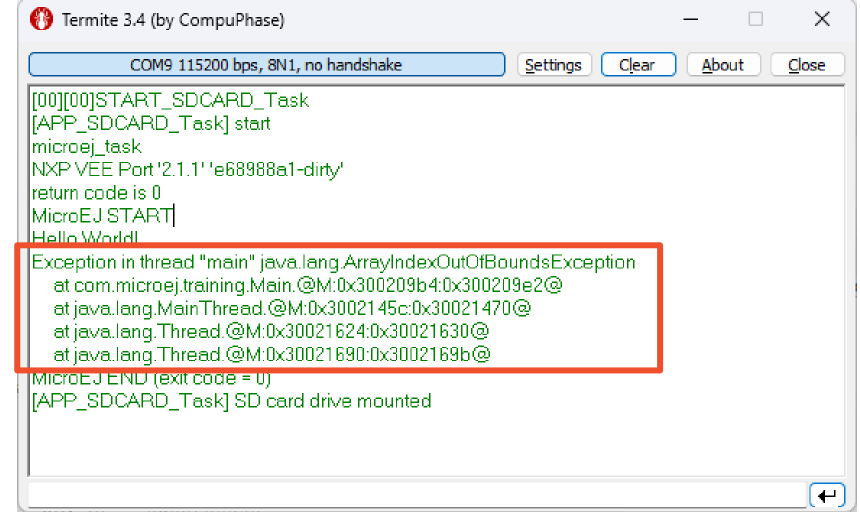
EXCEPTION OUTPUT

- In the console, we can see the stack trace:



- Name of the faulty **method is not printed** directly:
  - Only the address of the method is printed
  - MICROEJ does not embed the names of the methods to limit the footprint
- To help reading the stack trace, a tool is available: **the stack trace reader**

- **Tips:** the soar.generate.classnames can be set to get the name of the faulty class and a more detailed exception:



To set the option:

- Add the property **soar.generate.classnames=true** in the **configuration/common.properties** file.

Note: This option requires additional ROM space. **It is recommended to disable it when going to production**.

© MicroEJ 2025

STACK TRACE READER (3/4)

RUN THE STACK TRACE READER TOOL

- Open **Tasks > microej** and double-click on **stackTraceReader**
- Or use the following command line: `./gradlew stackTraceReader`

The Stack Trace Reader console opens:

```
> Task :nxpvee-mimxrt1170-evk:vee-port:mock:buildMockRip
> Task :nxpvee-mimxrt1170-evk:vee-port:buildVeePortConfiguration
> Task :loadVee

> Task :stackTraceReader
[INFO] Paste the MicroEJ core engine stack trace here.
```

© MicroEJ 2025

STACK TRACE READER (4/4)

USAGE

1. **Copy/Paste** the trace in your console.
2. The decoded trace appears.

The Stack Trace Reader can also be configured to read the data directly from the COM port of the device.

Note: restart the Stack Trace Reader tool **each time the application is rebuilt / flashed** on the device in order to properly decode the stack trace.

Online documentation:
https://docs.microej.com/en/latest/SDK6UserGuide/stackTraceReader.html

```
[INFO] Paste the MicroEJ core engine stack trace here.
Exception in thread "main" @C:0x30018a20@
    at @C:0x30018760@.@M:0x3001a3ac:0x3001a3dc@
    at @C:0x30018ea0@.@M:0x3001aeb8:0x3001aece@
    at @C:0x30018840@.@M:0x3001b084:0x3001b090@
    at @C:0x30018840@.@M:0x3001b0f0:0x3001b0fb@
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException Create breakpoint
    at com.microej.example.mylibrary.helloworld.Main.main(Main.java:19)
    at java.lang.MainThread.run(Thread.java:856)
    at java.lang.Thread.runWrapper(Thread.java:465)
```

```
14      public static void main(String[] args) {
15
16          System.out.println("Hello World!"); //$NON-NLS-1$
17
18          byte[] array = new byte[5];
19          array[5] = 42; // Invalid access to the array
20
21      }
```

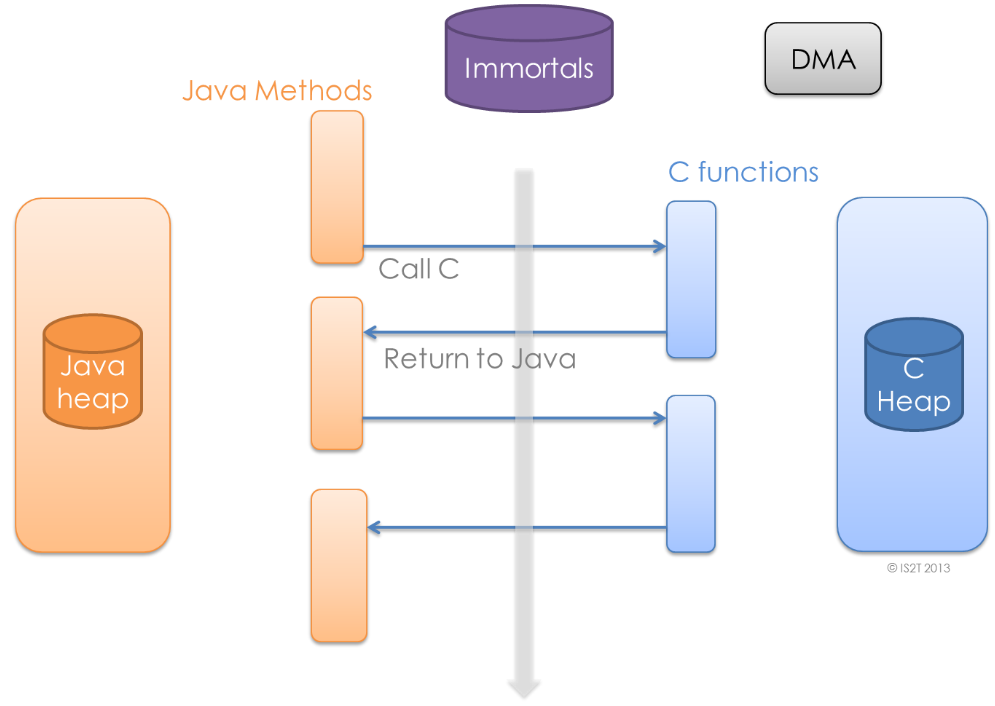
Call C code from Managed Code (Java)

Introduction to SNI mechanism
(Simple Native Interface)

© MicroEJ 2025

PRINCIPLE (1/2)

SNI Resolves native calls by executing them in another language (most of the time in C language).



Online documentation: https://docs.microej.com/en/latest/PlatformDeveloperGuide/sni.html

© MicroEJ 2025

PRINCIPLE (2/2)

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 a Java array from a native method written in C.
- Access a Java Immortal array from another RTOS task, an interrupt handler, or a DMA (see the BON specification to learn about immortal objects).

SNI does not allow you to:

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

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

© MicroEJ 2025

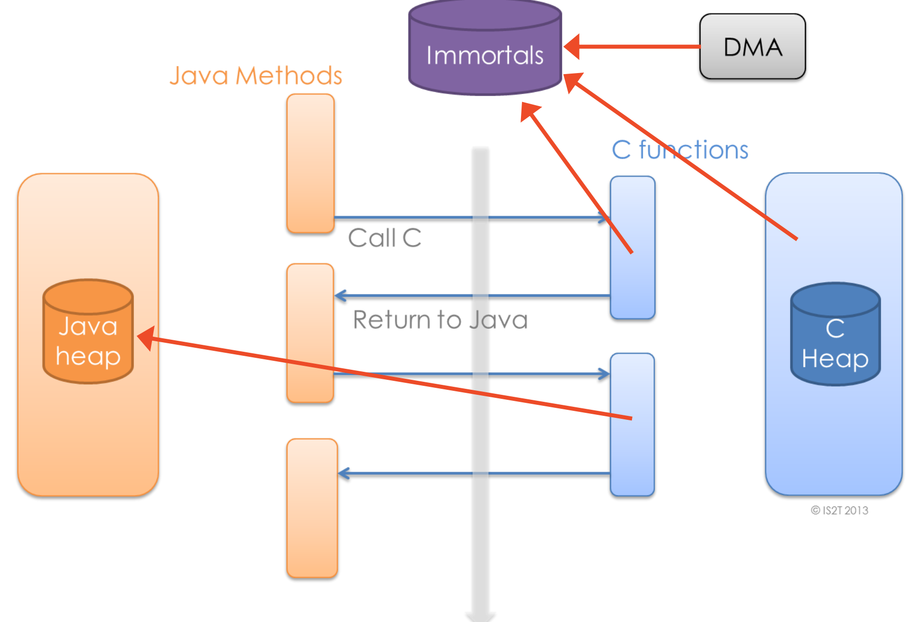
NAMING CONVENTION

```java
package com.corp.examples;
public class Hello {

    public static void main(String[] args){
        int i = printHelloNbTimes(3);
    }
    public static native int printHelloNbTimes(int times);
}
```

```c
#include <sni.h>
#include <stdio.h>

jint Java_com_corp_examples_Hello_printHelloNbTimes(jint times){
    while (--times){
        printf(“Hello world!\n”) ;
    }
    return 0 ;
}
```

DATA TYPES

- Primitive data type can be manipulated through SNI (return value and parameter):
  - byte, short, int, long, float, double, boolean, char.
- Arrays of primitive data type are managed by SNI with some limitations:
  - C globals, C Heap, DMA, RTOS tasks can reference only Immortal arrays.
  - Non-immortal arrays can be referenced only in the native method they are passed to.
  - Non-immortal arrays are passed to the native method as a reference.



Implement a Java Native Method with SNI

© MicroEJ 2025

ADD THE JAVA NATIVE METHOD

- Modify the code of the HelloWorld main method:

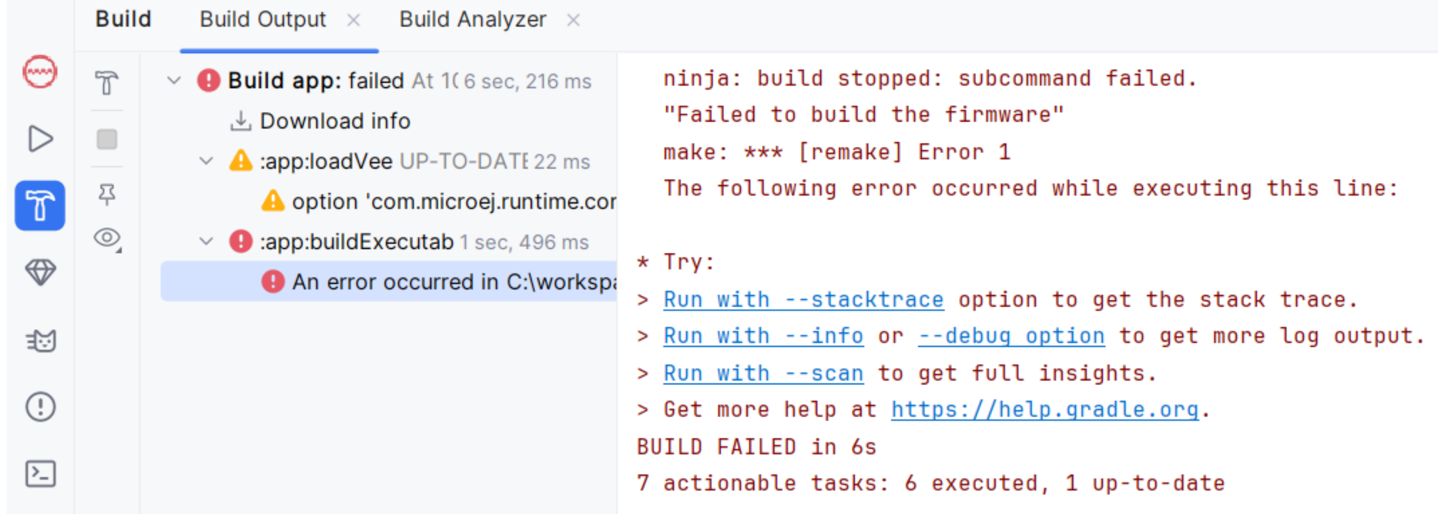
```
public class Main {

    public static void main(String[] args) {
        System.out.println("Hello World!");
        System.out.println("Multiply By Two (2): " + multiplyByTwo(2));
    }
    public static native int multiplyByTwo(int value);

}
```

Build the Application executable:

- Open the **Gradle** view.
- Open **Tasks > microej** and double-click on **buildExecutable**.
- Or use the following command line:
  `./gradlew buildExecutable`

GET THE LINKER ERRORS

The build fails with the following error message:



The detailed error can be found by scrolling up in the logs. It comes from a link issue:

```
tering directory `C:/workspaces/training/nxpvee-mimxrt1170-evk/nxpvee-mimxrt1170-evk/bsp/projects/nxpvee-ui/armgcc'
tering directory `C:/workspaces/training/nxpvee-mimxrt1170-evk/nxpvee-mimxrt1170-evk/bsp/projects/nxpvee-ui/armgcc'
tering directory `C:/workspaces/training/nxpvee-mimxrt1170-evk/nxpvee-mimxrt1170-evk/bsp/projects/nxpvee-ui/armgcc'
aving directory `C:/workspaces/training/nxpvee-mimxrt1170-evk/nxpvee-mimxrt1170-evk/bsp/projects/nxpvee-ui/armgcc'
tering directory `C:/workspaces/training/nxpvee-mimxrt1170-evk/nxpvee-mimxrt1170-evk/bsp/projects/nxpvee-ui/armgcc'
ing CXX executable flexspi_nor_sdram_debug/nxpvee_ui.elf
files (x86)/gnu arm embedded toolchain/10 2021.10/bin/../lib/gcc/arm-none-eabi/10.3.1/../../../../arm-none-eabi/bin/ld.exe: ../../microej/platform/lib/microejapp.o: in fu
es\training\HelloWorld\app\build\application\object\SOAR.o:(.rodata.microej.soar+0x1bb8): undefined reference to `Java_com_microej_example_helloworld_Main_multiplyByTwo'
on          Used Size  Region Size  %age Used
onfig:         512 B         3 KB     16.67%
```

The **multiplyByTwo(int value)** method is a native method. **It must be implemented in the BSP.**

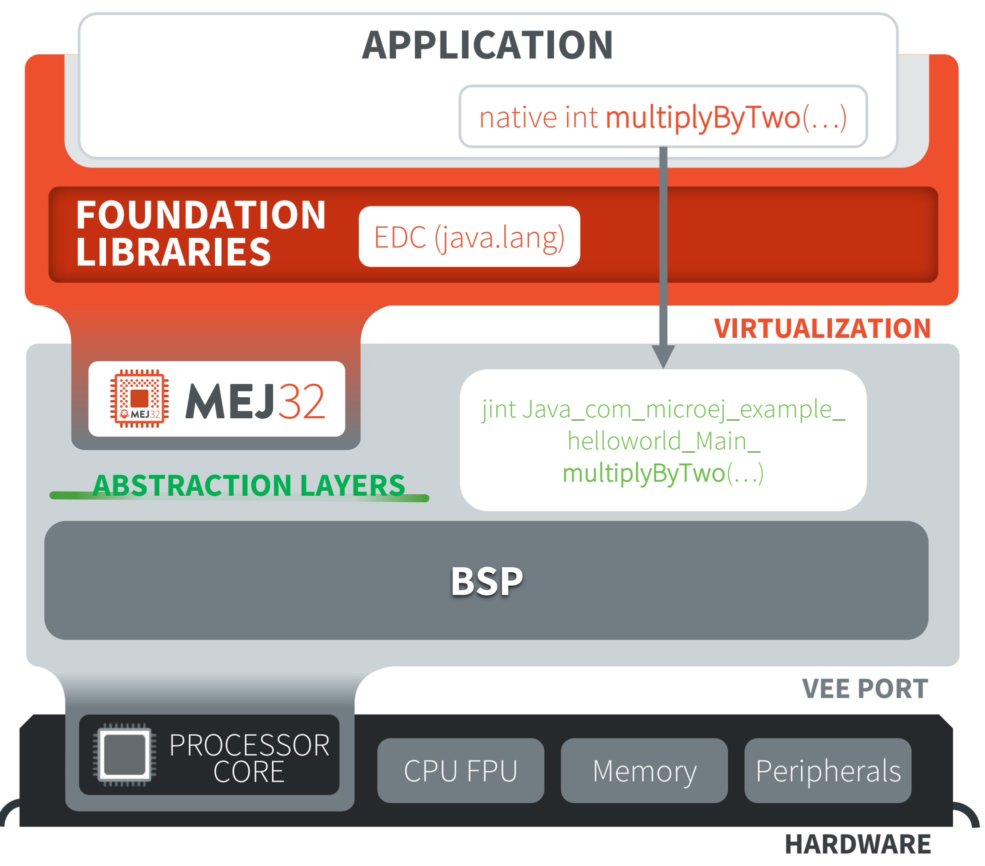
IMPLEMENT THE NATIVE METHOD IN THE BSP

- Go back to the IDE window where the VEE Port is opened.
- Open a **.c** source file from the BSP (e.g. bsp/vee/port/core/src/microej_main.c).
- Implement the **multiplyByTwo(int value)** method, use the method signature provided by the linker error:

```
#include "sni.h"

#ifdef __cplusplus
   extern "C" {
#endif

jint Java_com_microej_example_helloworld_Main_multiplyByTwo(jint value){
   return value*2;
}
```

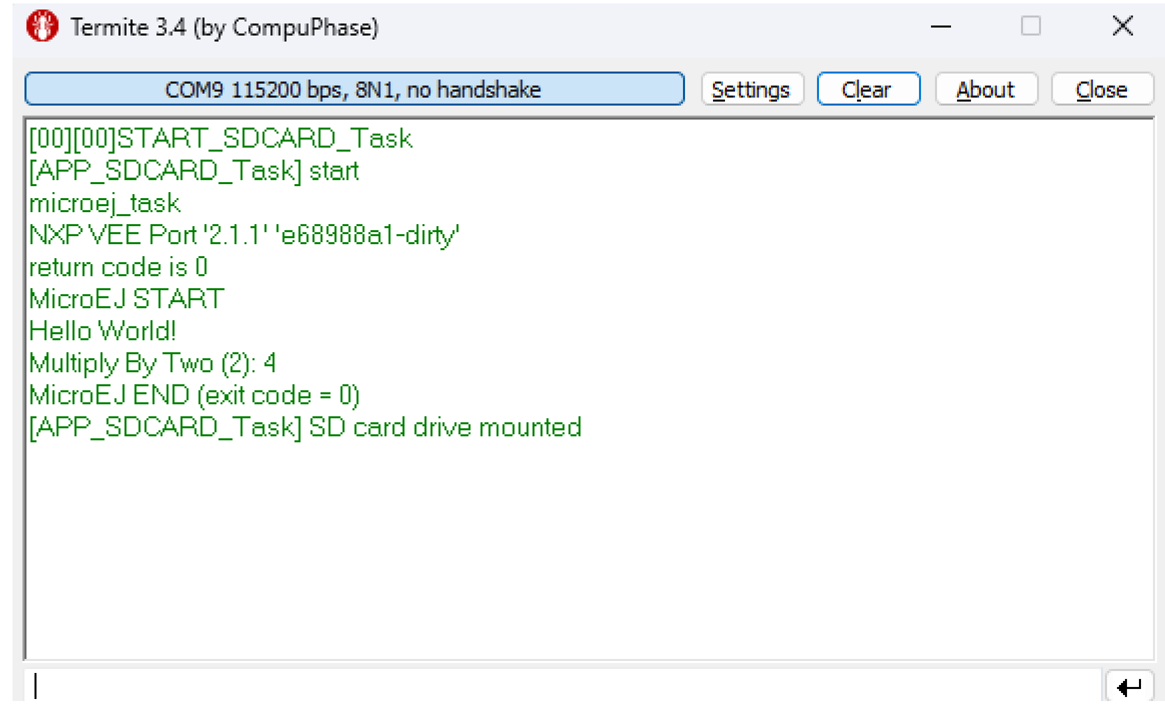
- Build the project again.
- The build is successful.
- Flash the firmware (see previous slides).

Note: sni.h provides java data types mapped on C base types (jint, jshort, jchar, jboolean, …).



RUN THE EXAMPLE ON THE DEVICE

- Open the Termite serial terminal.
- Click the **Settings** button.
- Select the NXP i.MX RT1170 EVK board COM port.
- Reset the NXP i.MX RT1170 EVK board using Reset button
- The application starts: the **Hello World** message and the Multiplied by Two value is printed!



© MicroEJ 2025

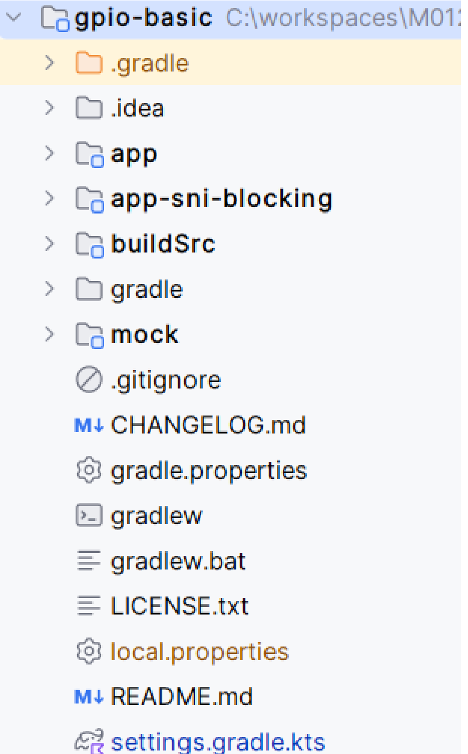
Blink an LED from Managed Code (Java)

Basic interaction with a GPIO using SNI

© MicroEJ 2025

OPEN THE PROJECT

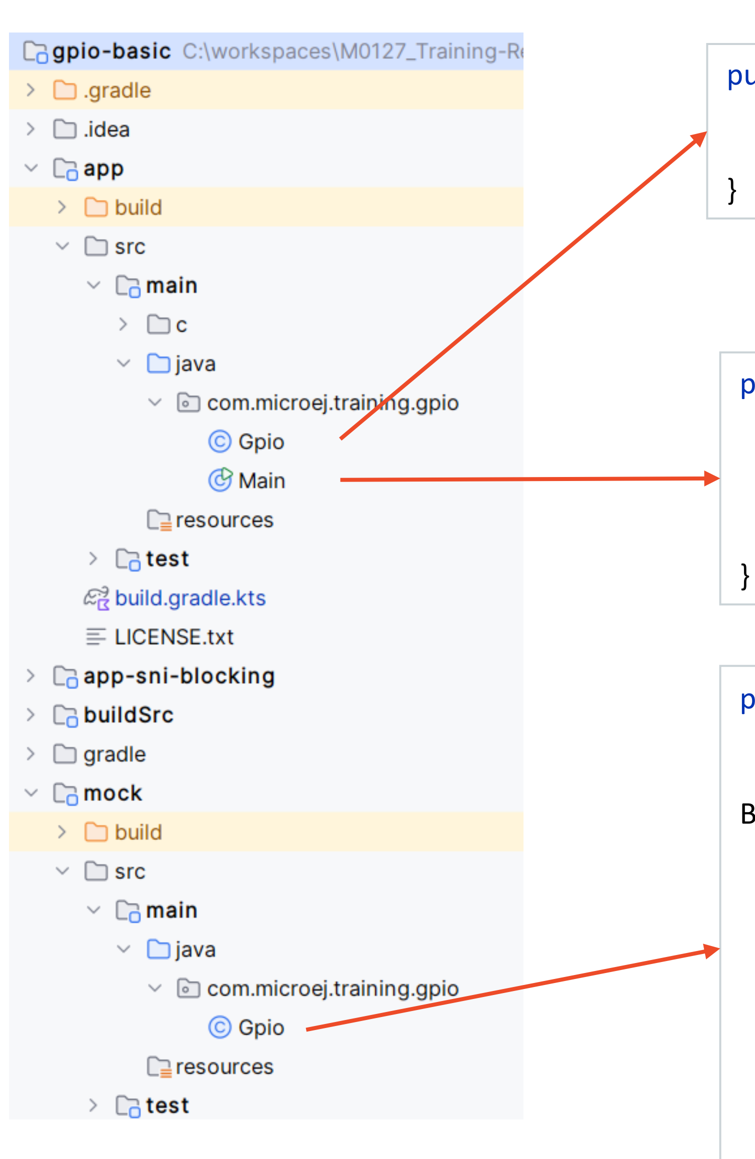
- Get the **gpio-basic** sample from the training package.
- In IntelliJ IDEA, open the **gpio-basic** example:
  - Open the project using the menu **File** > **Open...**
  - Click on OK.
  - Select to open in a new Window.
- The project appears in the IDE:



You can now close the IDE window corresponding to the **HelloWorld** project.

PROJECT STRUCTURE

MICROEJ




```java
public class Gpio {
    native public static void set(int pin, boolean value);
    native public static boolean get(int pin);
}
```

A **Gpio** class defines 2 native methods that should be implemented in the BSP to toggle the device GPIO.

```java
public static void main(String[] args) throws InterruptedException {
    while (true) {
        Gpio.set(PIN, !Gpio.get(PIN));
        Thread.sleep(DELAY);
    }
}
```

The **Main** class toggles the GPIO every 500 ms.

Application Project

```java
public class Gpio {

    private static final Map<Integer, Boolean> GPIO = new HashMap<Integer,
Boolean>();

    public static void set(int pin, boolean state) {
      System.out.println("Set GPIO "+pin+" to "+(state?"on":"off"));
      GPIO.put(Integer.valueOf(pin), Boolean.valueOf(state));
    }

    public static boolean get(int pin) {
      return GPIO.getOrDefault(Integer.valueOf(pin), Boolean.FALSE).booleanValue();
    }
}
```

The **Gpio** class is implemented in a **mock** project to run the code in simulation

Mock Project

© MicroEJ 2025

Run on the Simulator

VEE PORT SELECTION

- Get the path to the **NXP i.MX RT1170 VEE Port** (e.g. C:\workspaces\training\nxpvee-mimxrt1170-evk\nxpvee-mimxrt1170-evk)
- Add the path to the VEE Port in the **settings.gradle.kts** file of the application project:

```
rootProject.name = "gpio-basic"
include(":app")
include(":app-sni-blocking")
include(":mock")
includeBuild("C:\\workspaces\\training\\nxpvee-mimxrt1170-evk\\nxpvee-mimxrt1170-evk")
```

- In the **build.gradle.kts** file of the application project, add the dependency to the VEE Port:

```
dependencies {
    . . .

    //Uncomment the microejVee dependency to set the VEE Port or Kernel to use
    microejVee("com.nxp.vee.mimxrt1170:vee-port:3.0.0")
}
```

© MicroEJ 2025

RUN ON THE SIMULATOR

- Open the **Gradle** tasks view.
- Open **gpio-basic > Tasks > app > microej** and double-click on **runOnSimulator**.
- Or use the following command line:
  `./gradlew :app:runOnSimulator`

- The following traces can be seen in the console:

```
=============== [ Initialization Stage ] ===============
=============== [ Launching on Simulator ] ===============
Set GPIO 0 to on
Set GPIO 0 to off
Set GPIO 0 to on
…
```

Run on the Device

© MicroEJ 2025

RUN THE EXAMPLE ON THE DEVICE

- Open the **Gradle** view.
- Open **gpio-basic > Tasks > app > microej** and double-click on **runOnDevice**.
- Or use the following command line:
  `./gradlew :app:runOnDevice`

© MicroEJ 2025

GET THE LINKER ERRORS

- The following errors show up during the link step of the BSP:

```
C:\XXX\application\object\SOAR.o:(.text.soar+0x24dc): undefined reference to
`Java_com_microej_training_gpio_Gpio_get’

C:\XXX\application\object\SOAR.o:(.text.soar+0x24f0): undefined reference to
`Java_com_microej_training_gpio_Gpio_set'
```

- The GPIO **set()** and **get()** methods are native methods. **They must be implemented in the BSP.**

- The GPIO natives implementation is available in the **gpio-basic** folder: **gpio-basic-{version}/app/src/main/c/LLGPIO_NXP-i.MX_RT1170.c**
- Copy / Paste the **LLGPIO_NXP-i.MX_RT1170.c** source file in a source folder of the BSP project (e.g. bsp\vee\src\main)
- Add **LLGPIO_NXP-i.MX_RT1170.c** to **CMakeLists.txt** (bsp/vee/scripts/armgcc/CMakeLists.txt):

armgcc\CMakeLists.txt

```
81  add_executable(${MCUX_SDK_PROJECT_NAME}
82      "${MicroEjRootDirPath}/src/main/LLGPIO_NXP-i.MX_RT1170.c"
83      "${MicroEjRootDirPath}/src/main/npavee.c"
84      "${MicroEjRootDirPath}/src/main/npavee.h"
```

GPIO NATIVES IMPLEMENTATION

- The **LLGPIO_get(int32_t pin)** and **LLGPIO_set(int32_t pin, uint8_t state)** functions are implemented as follows:

```
static void LLGPIO_initialize(void)
{
    if(!GPIO_initialized)
    {
        GPIO_initialized = 1;

        // LED initialization

        /* GPIO configuration on GPIO_AD_04 (pin M13) */
        gpio_pin_config_t gpio9_pinM13_config = {
          .direction = kGPIO_DigitalOutput,
          .outputLogic = 0U,
          .interruptMode = kGPIO_NoIntmode
        };
        /* Initialize GPIO functionality on GPIO_AD_04 (pin M13) */
        GPIO_PinInit(EXAMPLE_LED_GPIO_PORT, EXAMPLE_LED_GPIO_PIN, &gpio9_pinM13_config);

        IOMUXC_SetPinMux(
          IOMUXC_GPIO_AD_04_GPIO9_IO03,           /* GPIO_AD_04 is configured as GPIO9_IO03 */
          0U);
    }
}
```

```
void LLGPIO_set(int32_t pin, uint8_t state)
{
    LLGPIO_initialize();

    if( state == JFALSE)
    {
        GPIO_PinWrite(EXAMPLE_LED_GPIO_PORT, EXAMPLE_LED_GPIO_PIN, 0U);
    }
    else
    {
        GPIO_PinWrite(EXAMPLE_LED_GPIO_PORT, EXAMPLE_LED_GPIO_PIN, 1U);
    }
}

uint8_t LLGPIO_get(int32_t pin)
{
    LLGPIO_initialize();
    return (GPIO_ReadPinInput(EXAMPLE_LED_GPIO_PORT, EXAMPLE_LED_GPIO_PIN))  == 0 ? JFALSE : JTRUE;

}
```

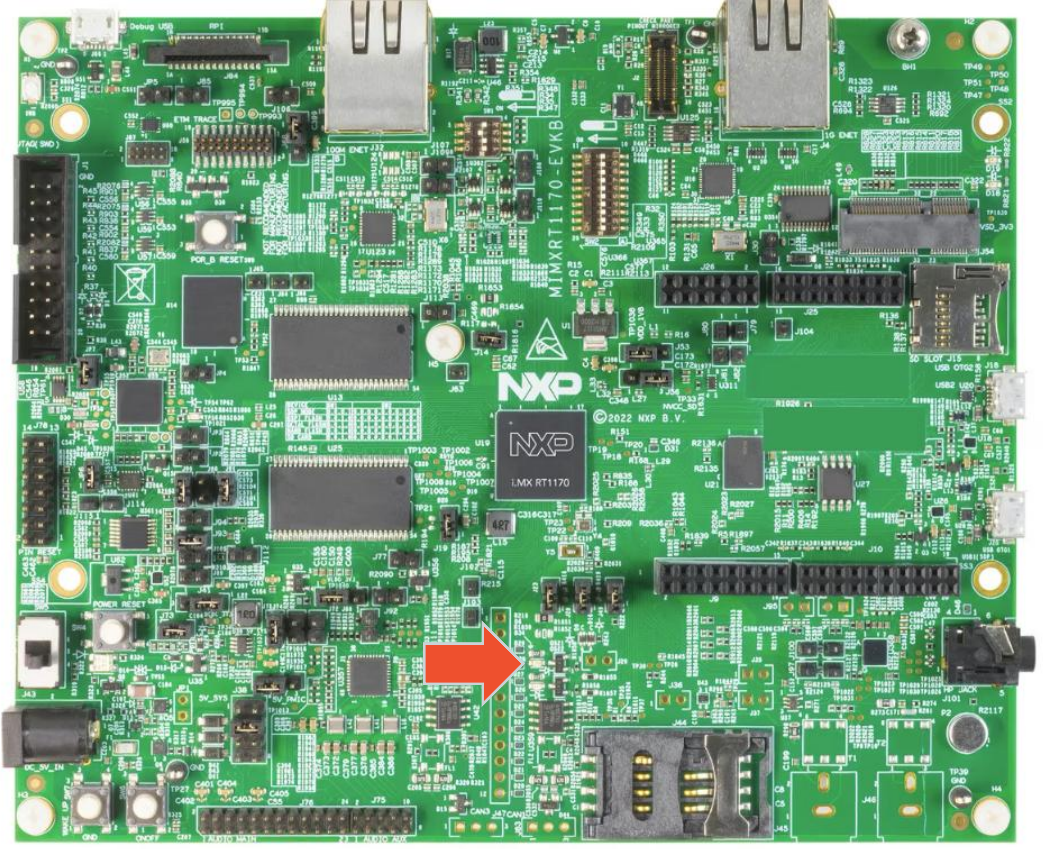
- *Note: this is a simple implementation that does not take care of GPIO pin number.*

© MicroEJ 2025

RUN THE EXAMPLE ON THE DEVICE

- Open the **Gradle** view.
- Open **gpio-basic > Tasks > app > microej** and double-click on **runOnDevice**.
- Or use the following command line:
  `./gradlew :app:runOnDevice`
- The application is built and the device is flashed.

- **The GREEN LED D6 is now blinking every 500ms.**



© MicroEJ 2025

Resources

USEFUL LINKS

- https://developer.microej.com/
  - Examples, platforms, libraries, user guides, application notes…
  - Javadocs (Java API)
  - Addon tools
- https://docs.microej.com
- https://github.com/MICROEJ/
  - Source code repository
- https://forum.microej.com/
- https://repository.microej.com/
  - MICROEJ Central Repository (modules repository)

THANK YOU

for your attention !

MICROEJ®