MICROEJ SDK 5 Basics

For NXP i.MX RT1170 Evaluation Kit

MICROEJ®

DISCLAIMER

WHAT YOU WILL LEARN

By the end of this training, you will be able to use MICROEJ SDK to:

- Build a VEE Port.
- Build and Run a Java Application.
- Edit a Front Panel File.
- Call a C function from Java.
- Build your own Foundation Library.
- Manage Multithreading with SNI.

PREREQUISITES

ENVIRONMENT SETUP (1/8)

- Get NXP i.MX RT1170 Evaluation Kit + micro-USB cable + RK055HDMIPI4MA0 display panel

Windows 10 or 11 64-bit:

- Install JDK 11 64-bit (https://adoptopenjdk.net/?variant=openjdk8&jvmVariant=hotspot).
    - Note: select the “JavaSoft (Oracle) registry keys” feature in the installer
- Install the latest MICROEJ SDK distribution: https://docs.microej.com/en/latest/SDKUserGuide/installSDKDistributionLatest.html
- Install a serial terminal (https://www.compuphase.com/software_termite.htm).

ENVIRONMENT SETUP (2/8)

GET WEST

West is a Zephyr tool for multiple repository management systems.

It will be used to fetch the code and its dependencies.

Install West by following Installing west instructions (tested with west 1.2.0).

Check that the tool has been properly installed:

```
C:\Users\Alex>west --version
West version: v1.2.0
```

ENVIRONMENT SETUP (3/8)

FETCH VEE PORT SOURCES

On Windows, fetching the source code may trigger the following **fatal error: error: unable to create file [...]: Filename too long.**
To avoid this, git configuration needs to be updated to handle long file names:

- Start Git Bash as Administrator.
- Run following command: **git config --system core.longpaths true**

Clone the VEE Port repository with the following commands:

- **mkdir nxpvee-mimxrt1170-prj**
- **cd nxpvee-mimxrt1170-prj**
- Copy the following command (including the “.” at the end):
  west init -m https://github.com/nxp-mcuxpresso/nxp-vee-imxrt1170-evk.git --mr NXPVEE-MIMXRT1170-EVK-2.1.1 .
- **west update**

For more information see VEE Port README.

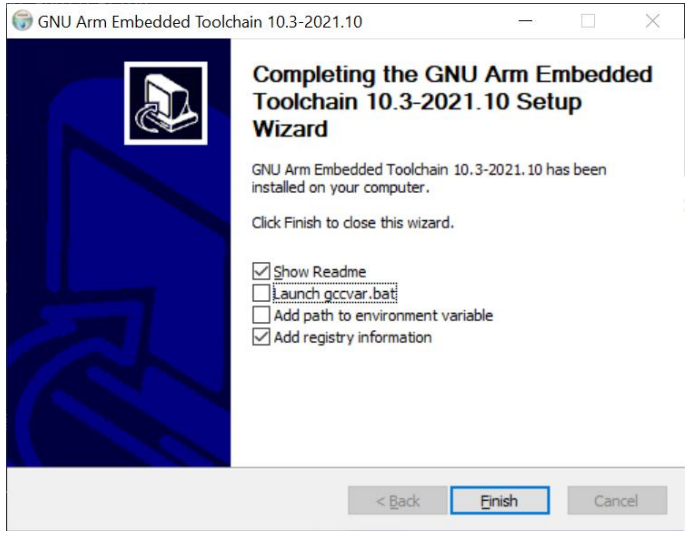
ENVIRONMENT SETUP (4/8)

INSTALL GNU ARM EMBEDDED TOOLCHAIN

Download and install GNU ARM Embedded Toolchain version 10.3 2021.10.

Windows 10 or 11 64-bit:

- At the end of the installation, it will ask you to complete the Setup of the wizard, choose the following options:



ENVIRONMENT SETUP (5/8)

INSTALL GNU ARM EMBEDDED TOOLCHAIN

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.

ENVIRONMENT SETUP (6/8)

INSTALL CMAKE

Download and install CMake (tested with CMake 3.28.3).

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.

Check that the tool has been properly installed:

```
C:\Users\Alex>cmake --version
cmake version 3.28.3

CMake suite maintained and supported by Kitware (kitware.com/cmake).
```

ENVIRONMENT SETUP (7/8)

INSTALL MAKE

Download and install Make (tested with Make 3.81).

Make is the tool that will generate the Firmware 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.

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.

Check that the tool has been properly installed:

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

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

ENVIRONMENT SETUP (8/8)

INSTALL THE FLASHING TOOL

Download and install LinkServer 1.2.45.

Windows 10 or 11 64-bit:

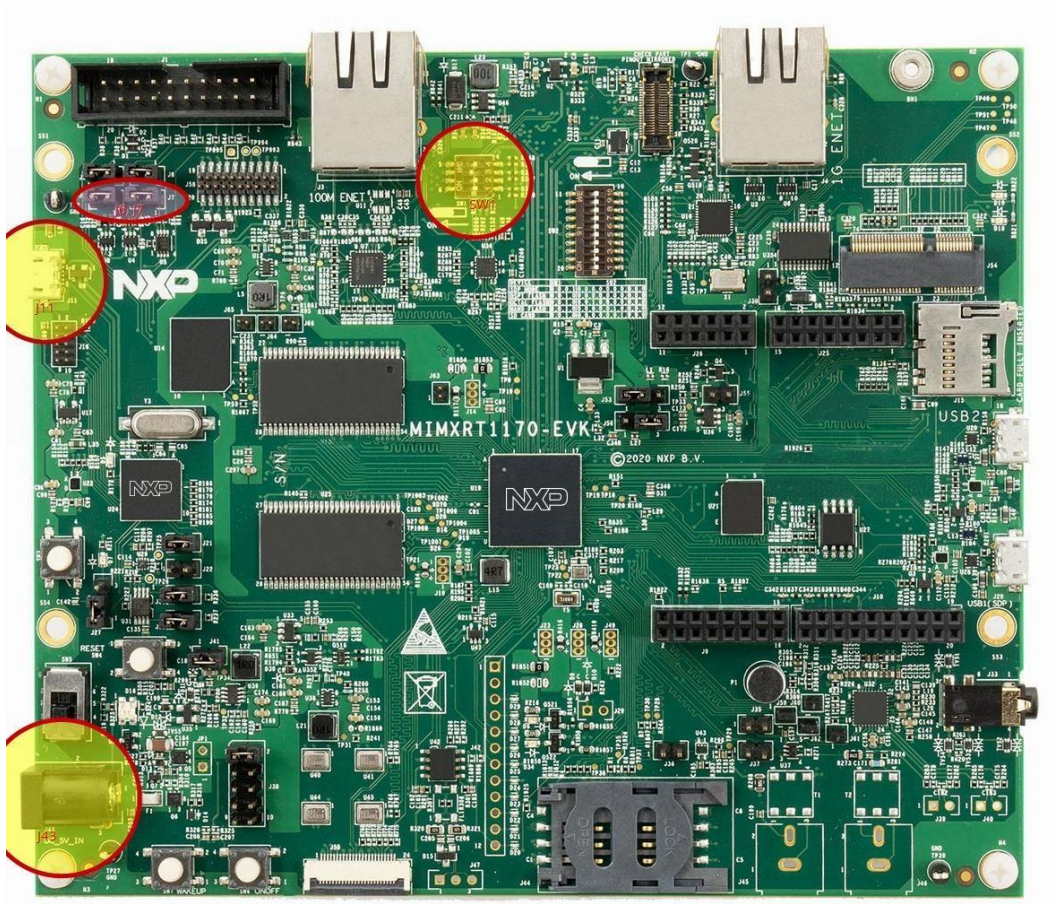
- Open the **Edit the system environment variables** application on Windows.
- Click on the **Environment Variables...** button.
- Select **Path** variable under the **User variables** section and edit it.
- Click on **New** and point to the binaries folder located where you installed LinkServer (e.g. C:\nxp\LinkServer_1.2.45\binaries).

HARDWARE SETUP

Setup the i.MX RT1170 Evaluation Kit

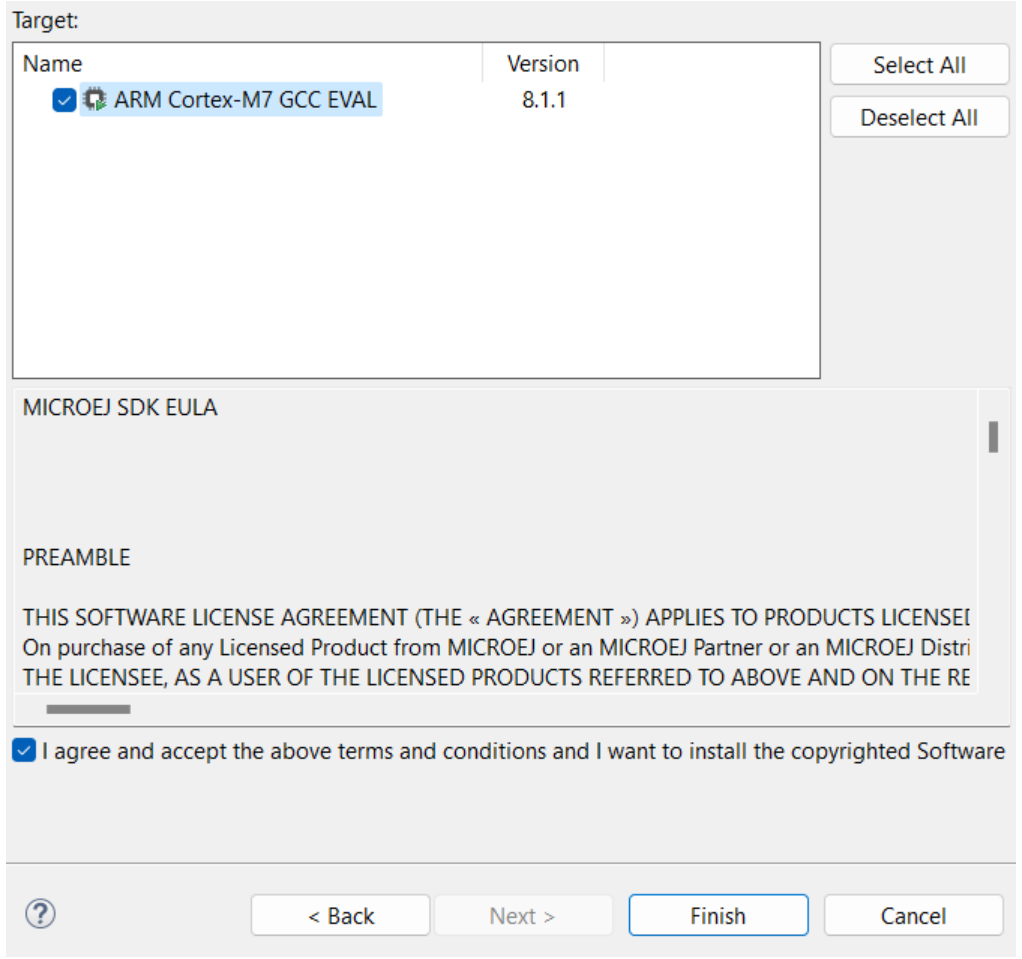
- Check that the dip switches (SW1) are set to OFF, OFF, ON and OFF.
- Ensure jumpers J6 and J7 are closed.
- Connect the micro-USB cable to J11 to power the board.
- You can connect 5 V power supply to J43 if you need to use the display

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.



INSTALL MICROEJ ARCHITECTURE

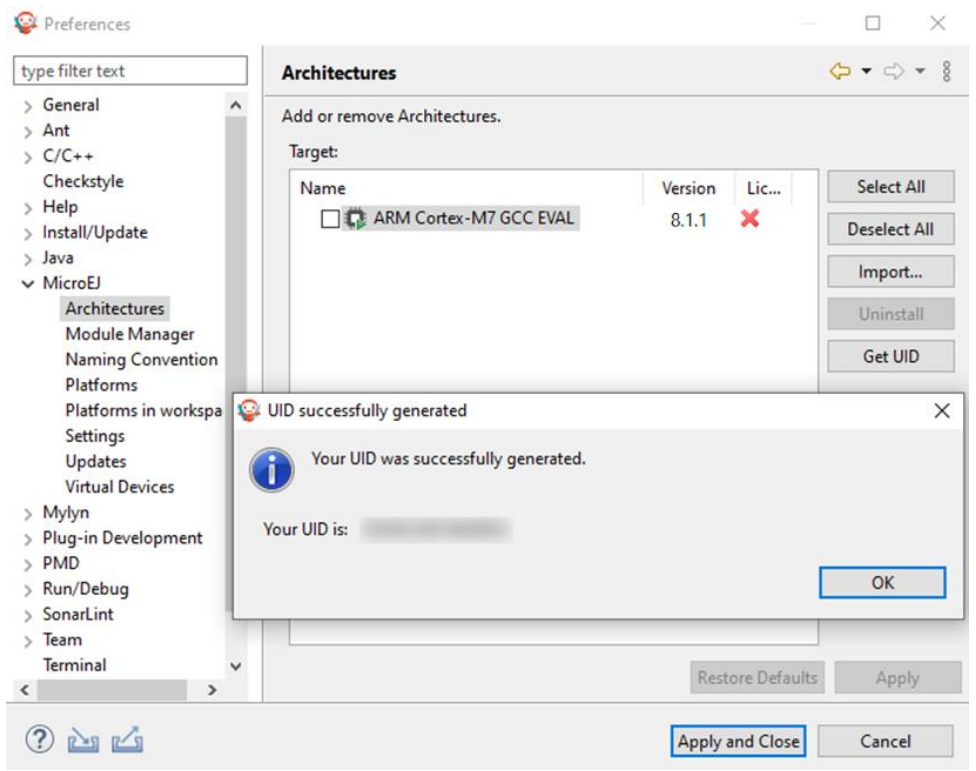
- Download the **flopi7G26-8.1.1-eval.xpf** MICROEJ Architecture for Cortex-M7 GCC (https://repository.microej.com/modules/com/microej/architecture/CM7/CM7hardfp_GCC48/flopi7G26/8.1.1/).
- Launch the MICROEJ SDK and select the default workspace.
- In MICROEJ SDK, click on **Window > Preferences > MICROEJ > Architectures > Import**.
- Select the MICROEJ Architecture previously downloaded **flopi7G26-{version}-eval.xpf**
- Accept the license terms and click on **Finish**.
- The architecture is now imported.
- Click on **Apply** and **Close** button.



ACTIVATING MICROEJ ARCHITECTURE LICENSE (1/3)

GETTING THE UID

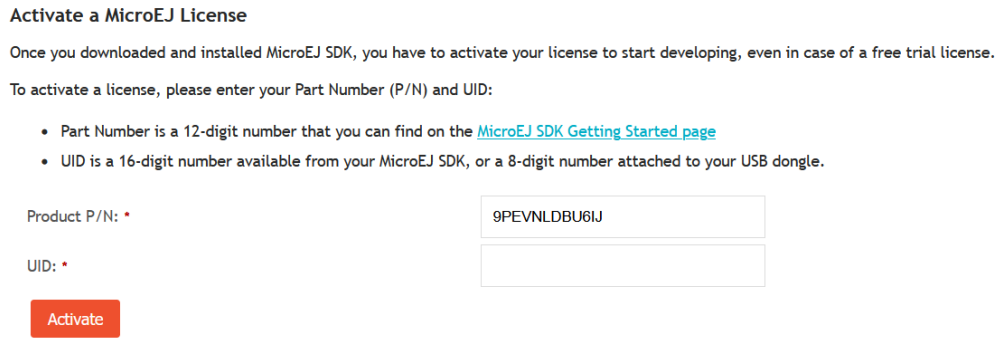
- In MICROEJ SDK, go to **Window > Preferences > MICROEJ > Architectures**.
- Select the **ARM Cortex-M7 GCC EVAL** Architecture.
- Click on **Get UID**.
- Copy the UID. It will be needed when requesting a license.



ACTIVATING MICROEJ ARCHITECTURE LICENSE (2/3)

GENERATING THE 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 **9PEVNLDBU6IJ**.
- Set UID: to the UID you generated before.
- Click on Activate.
- The license is being activated. It can be downloaded from the home page of license.microej.com.
- Once generated, download the attached zip file that contains your activation key.

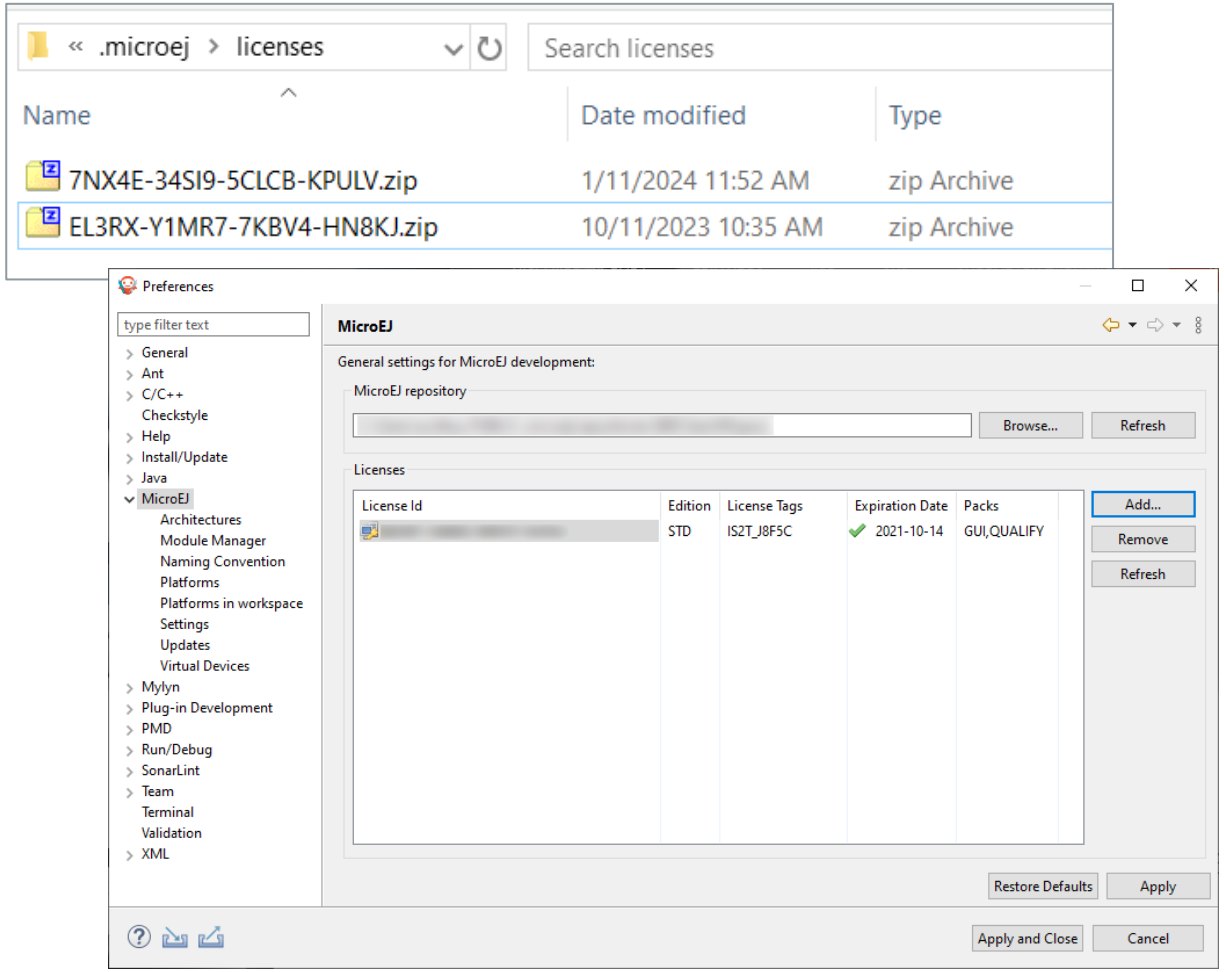


ACTIVATING MICROEJ ARCHITECTURE LICENSE (3/3)

ACTIVATING MICROEJ SDK

- Copy the zip file of the activation key in the folder **%user_home%/.microej/licenses** (create it if it does not exist).

- Then you can check if the activation key is correctly loaded in the MICROEJ SDK in **Window > Preferences > MICROEJ**. A new license is successfully installed.



VEE Port Concept

Computing platform for embedded system development

VEE PORT

- MICROEJ SDK brings the concept of **computing platform** to embedded system development
- Goals of this presentation:
  - **Why** computing platforms help to develop applications
  - **How to make a platform** with MicroEJ SDK?
- Computing platform = software platform = platform = VEE Port

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 should not we do the same for embedded devices?

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

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

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 Java 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 VEE

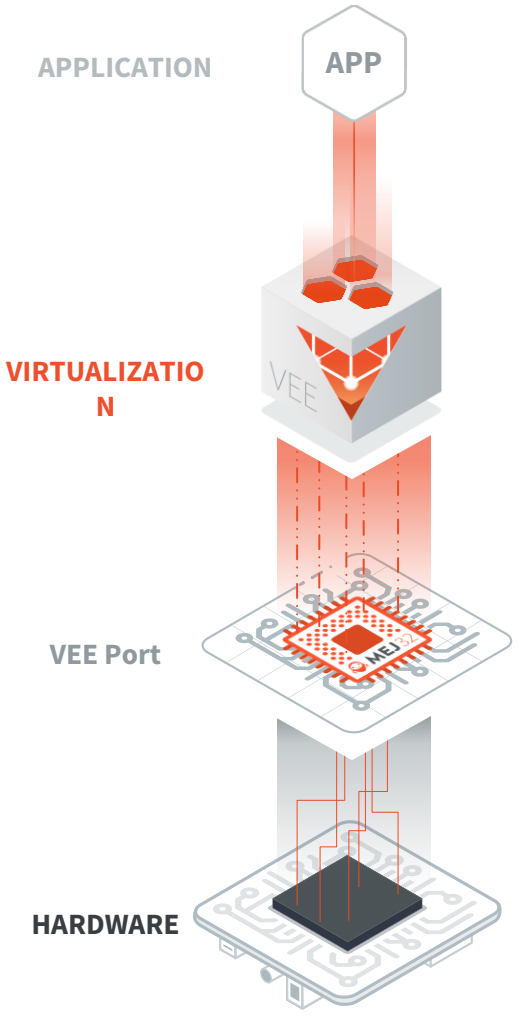
SERVICES

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 basic analog and digital IO framework.
- A sensor framework.
- A storage framework (file system).

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

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 | ECOM | BLE | …

MICROEJ VEE

VIRTUALIZATION

ABSTRACTION LAYERS

File System | Internet Protocol | Graphical Engine

MEJ32

ABSTRACTION LAYERS

I/O | Bluetooth | …

Drivers | BSP | Drivers | …

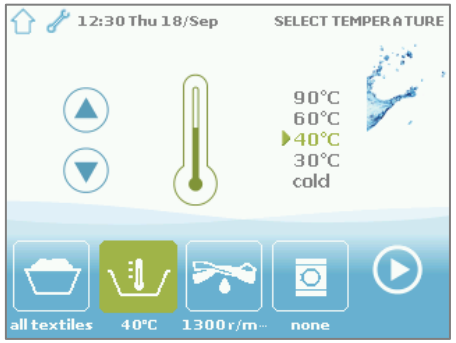
RTOS/OS

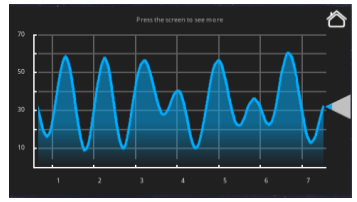
PLATFORM

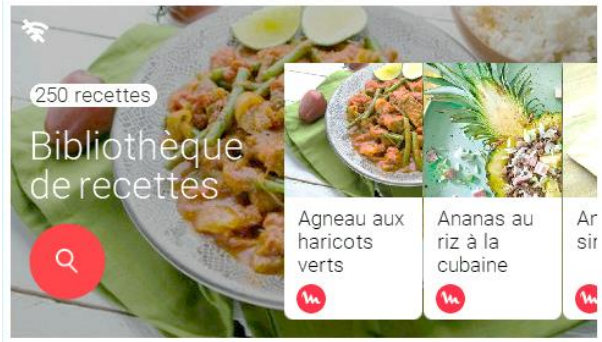
Mass Storage | Ethernet Wi-Fi / LTE | Display | PROCESSOR CORE | UART | Bluetooth | …

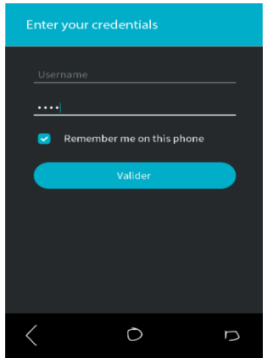
HARDWARE

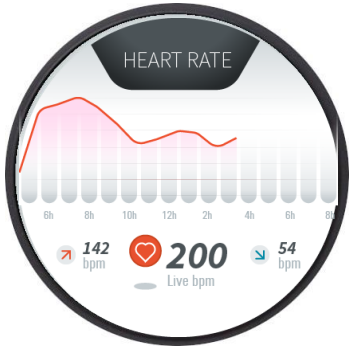
GUI EXAMPLES FOR $1 TO $5 MCU

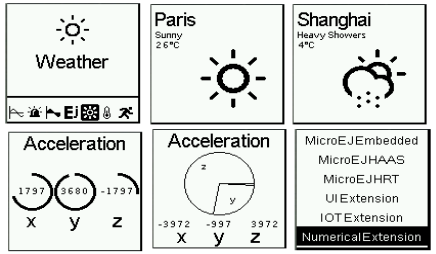










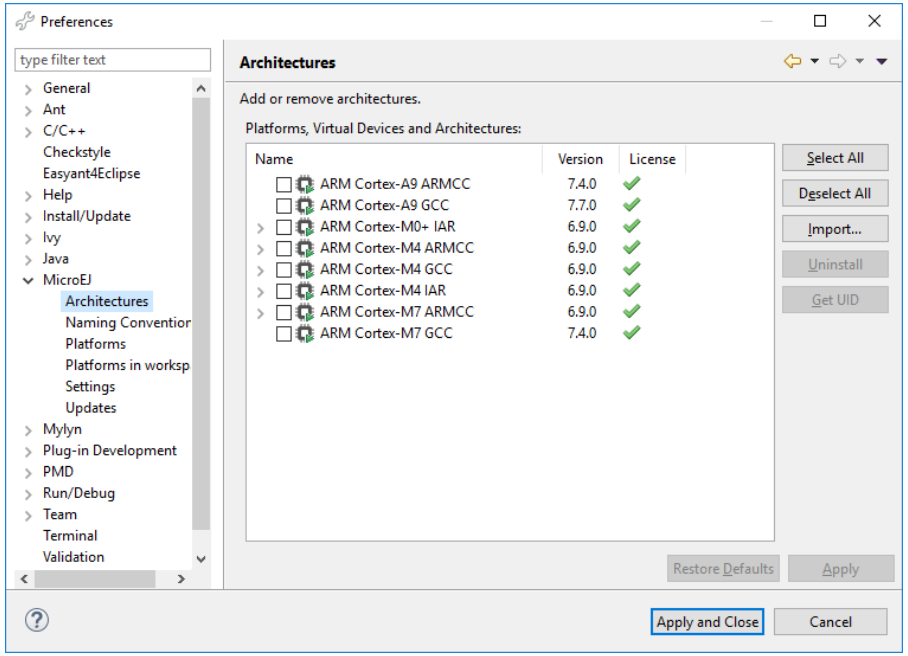


Firmware Build Flow

Build flow explained

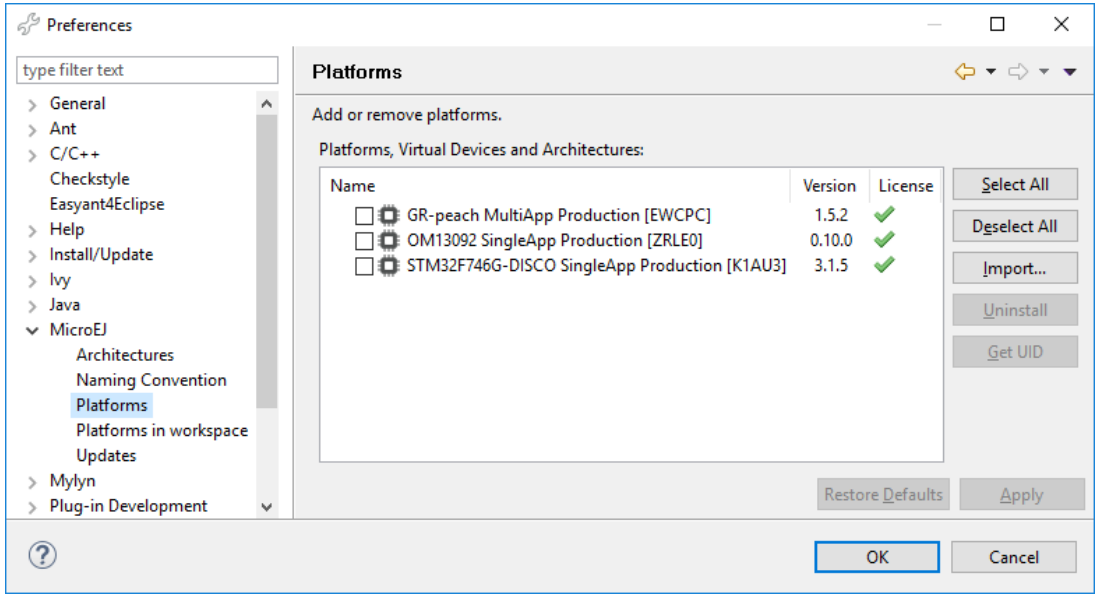
MICROEJ ARCHITECTURE

- 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.
- In SDK 5, go to menu **Window > Preferences > MicroEJ > Architectures.**
- Example of MICROEJ Architectures:
  - ARM Cortex-M4 - Keil ARM Compiler 5.
  - 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/

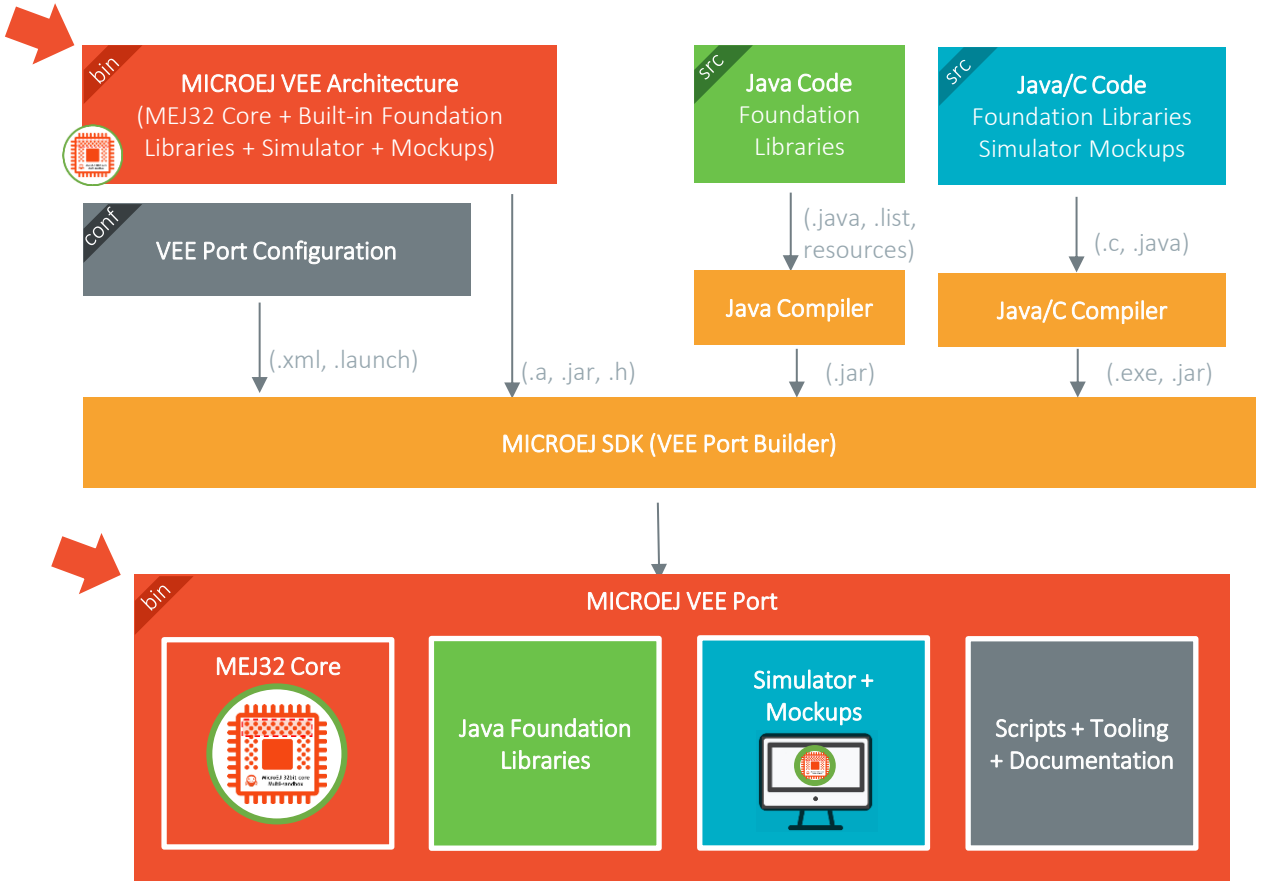


MICROEJ VEE PORT

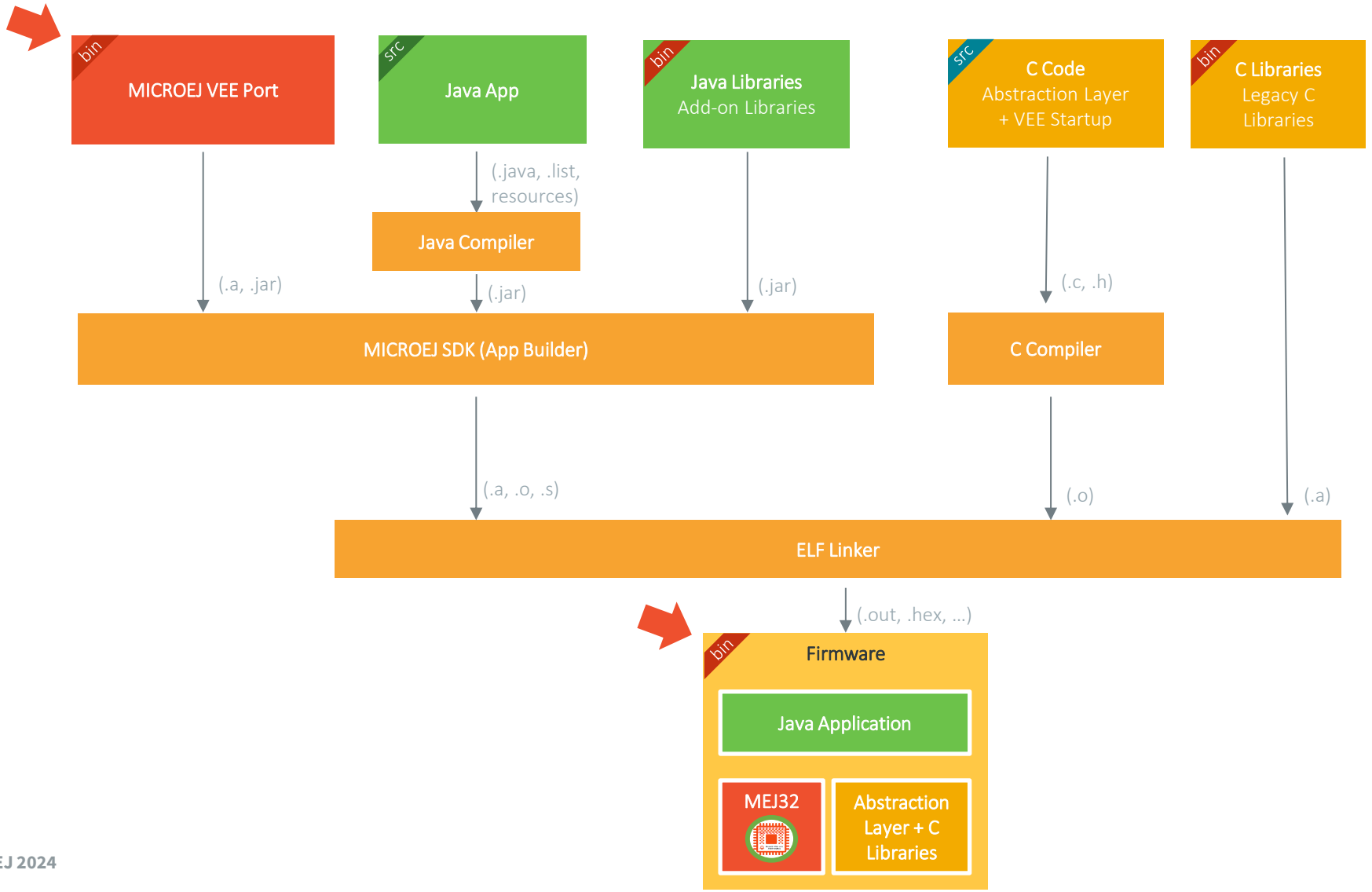
- A **MICROEJ VEE Port** is a port of a MICROEJ Architecture for a specific hardware, RTOS and BSP.
- MICROEJ VEE Ports are built using MICROEJ SDK5.
- They are distributed as source (including C sources) or binary (pre-built C BSP).
- In MICROEJ SDK5, go to menu **Window > Preferences > MicroEJ > Platforms.**
- Example of MICROEJ VEE Port:
  - Renesas S7G2-DK - ThreadX - SSP 1.3.
  - NXP OM13092 - FreeRTOS – KSDK.
  - Atmel SAMA5-Xplained – Linux.
- List of the platforms:
  - https://developer.microej.com/supported-hardware/



BUILD FLOW / VEE PORT



BUILD FLOW / FIRMWARE

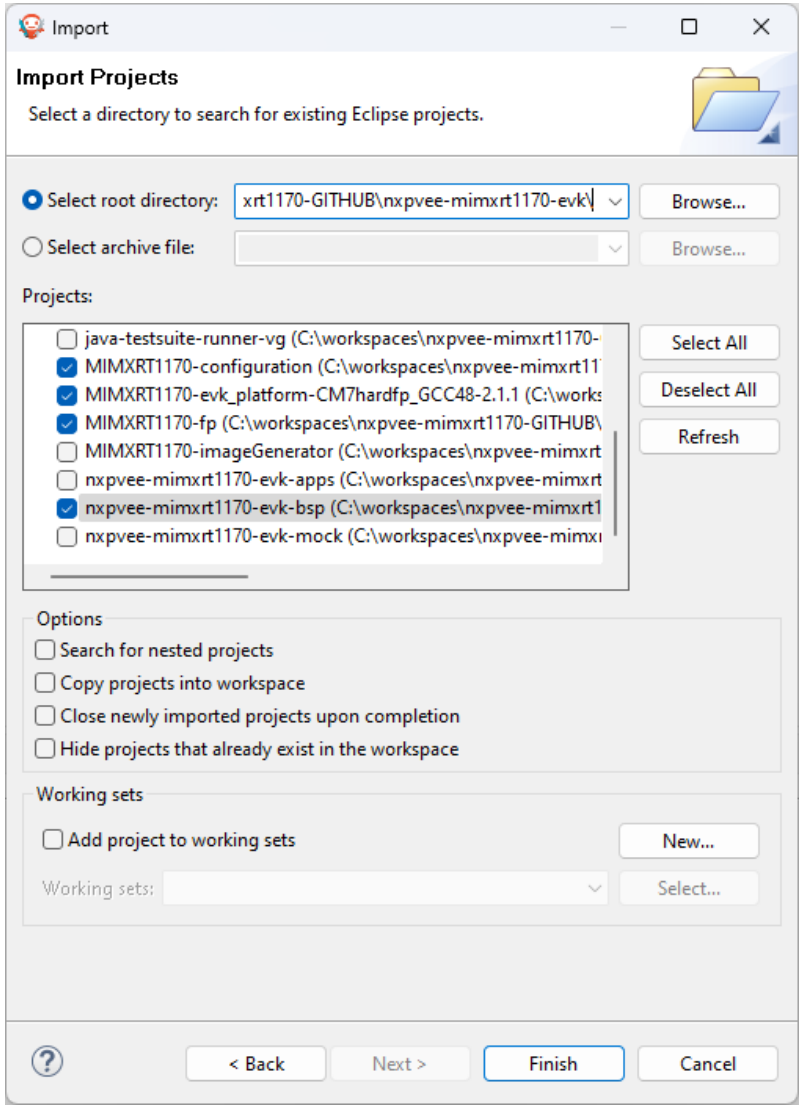


Build VEE Port

For NXP i.MX RT1170
Evaluation Kit

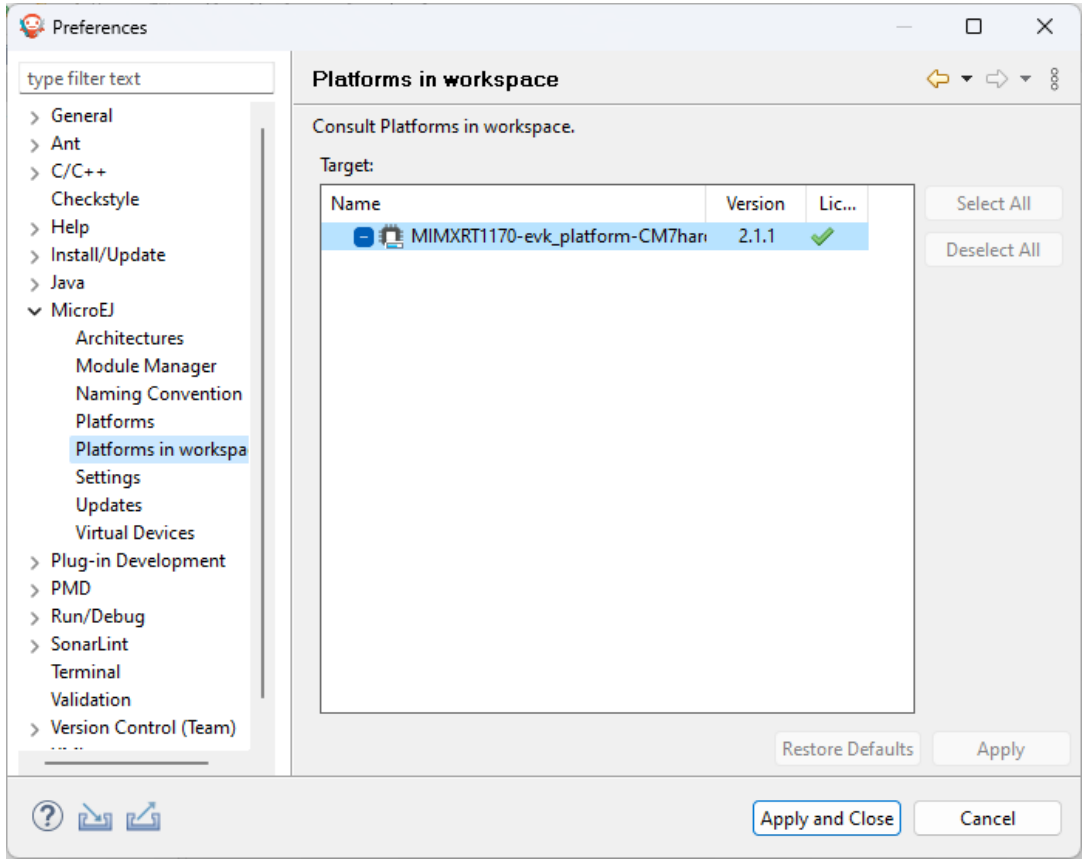
IMPORT VEE PORT SOURCES

- In MICROEJ SDK, go to **File > Import... > General > Existing Projects into Workspace**.
- Click on **Select archive file > Browse...**
- Browse to the VEE Port sources folder.
- The projects appears in the Projects list. Select the following ones:
  - **MIMXRT1170-configuration**: the configuration project used to configure the platform.
  - **MIMXRT1170-fp**: the front panel project. It describes the UI of the simulator.
  - **MIMXRT1170-bsp**: contains the Board Support Package (BSP) source code.
  - **MIMXRT1170-Platform-CM7hardfp_GCC48-{version}**: the VEE Port project (empty).
- Click on **Finish**.



BUILD THE VEE PORT

- Right-Click on **MIMXRT1170-configuration** project
- Click on **Build Module** to build the platform.
- The platform project **MIMXRT1170-Platform-CM7hardfp_GCC48-2.0.0** is now filled.
- You can see the platform in **Platforms in workspace** menu:
  - **Window > Preferences > MicroEJ > Platforms in workspace.**



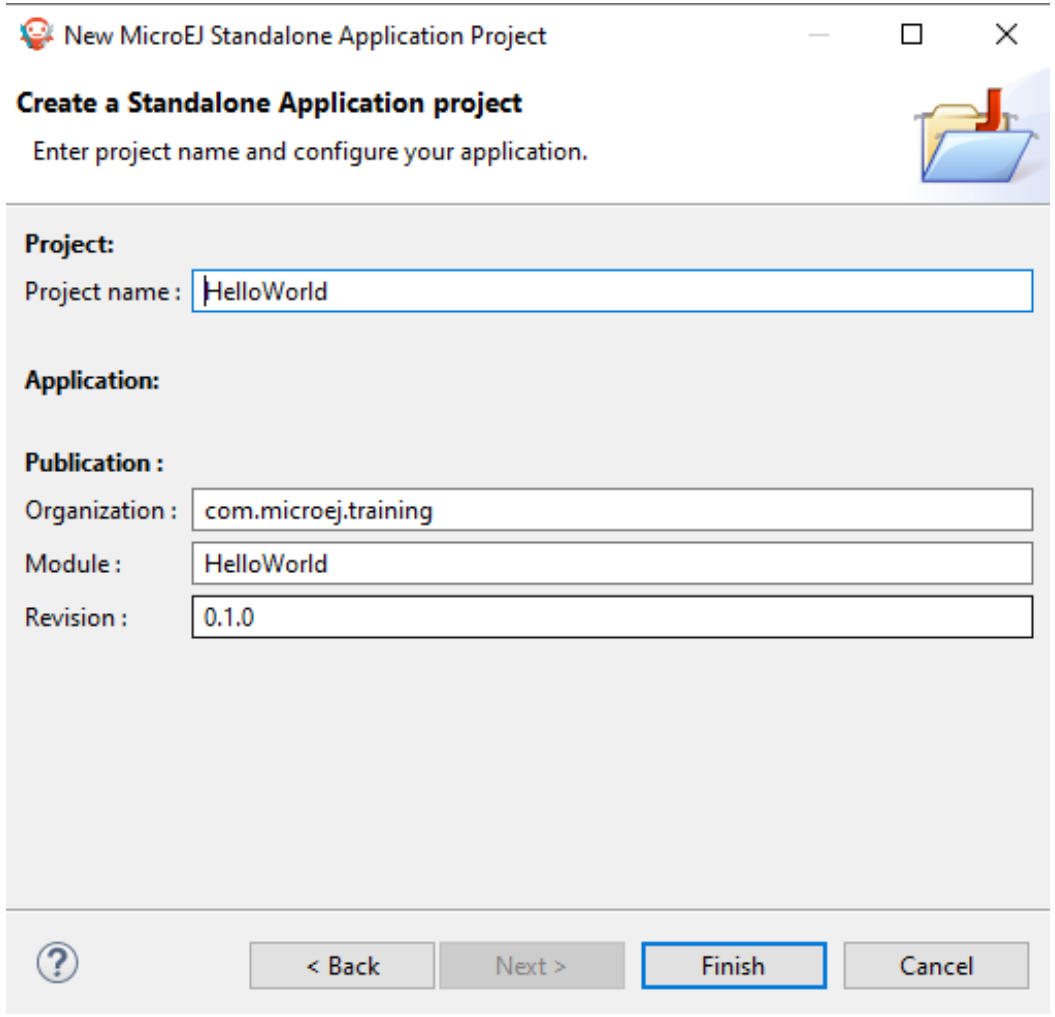
Application

Build & Run a Hello World Application.

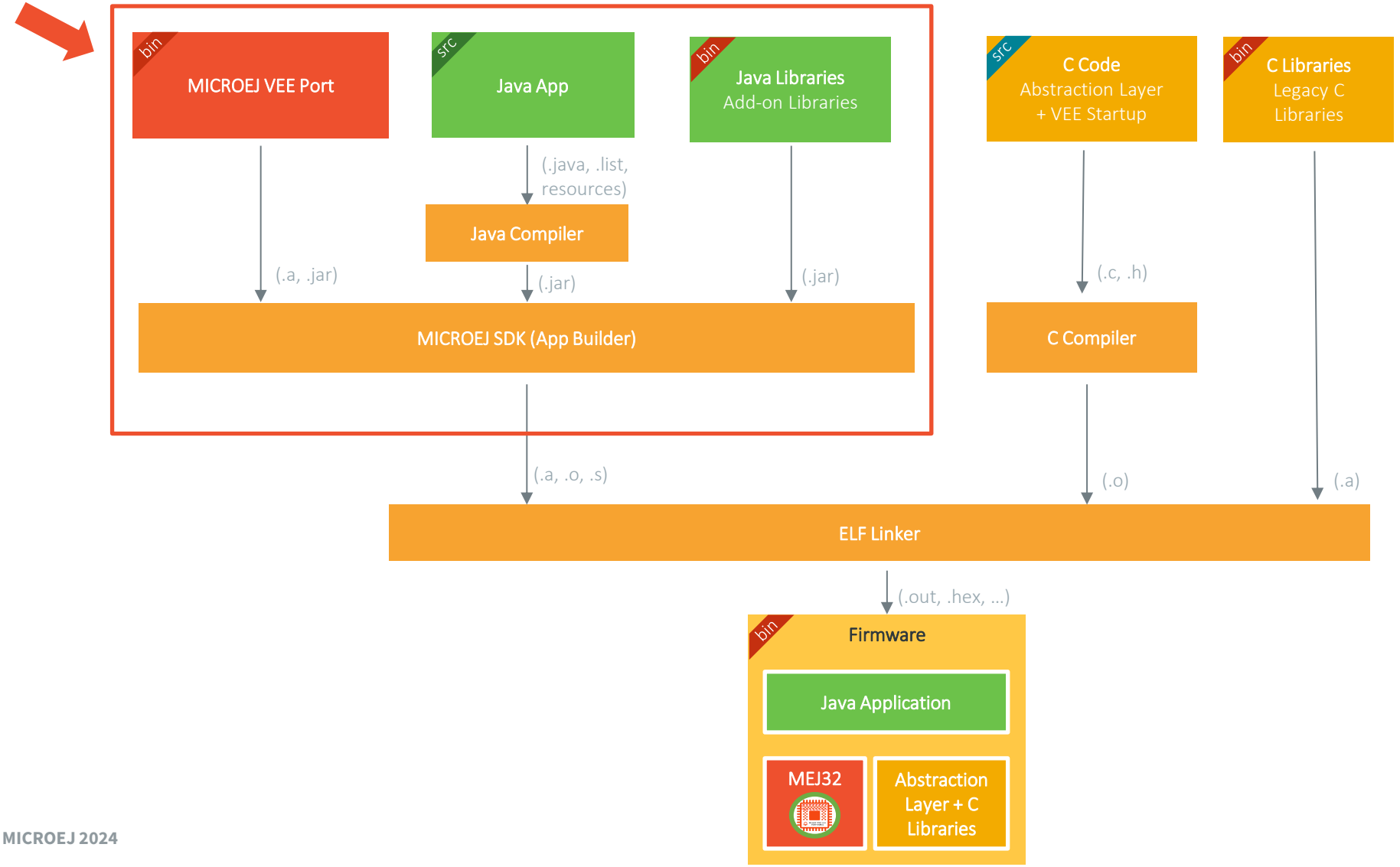
CREATE THE APPLICATION PROJECT

In MICROEJ SDK:

- Go to **File** -> **New** -> **MicroEJ Standalone Application Project**.
- Fill the input fields.



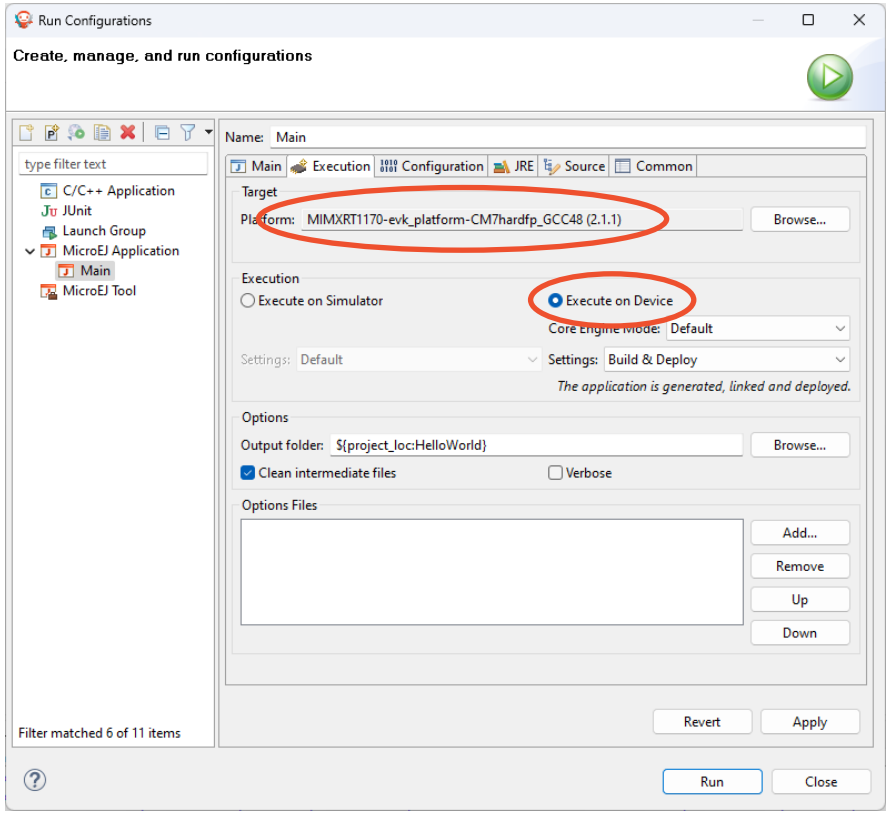
BUILD FLOW



BUILD THE APPLICATION

In MICROEJ SDK:

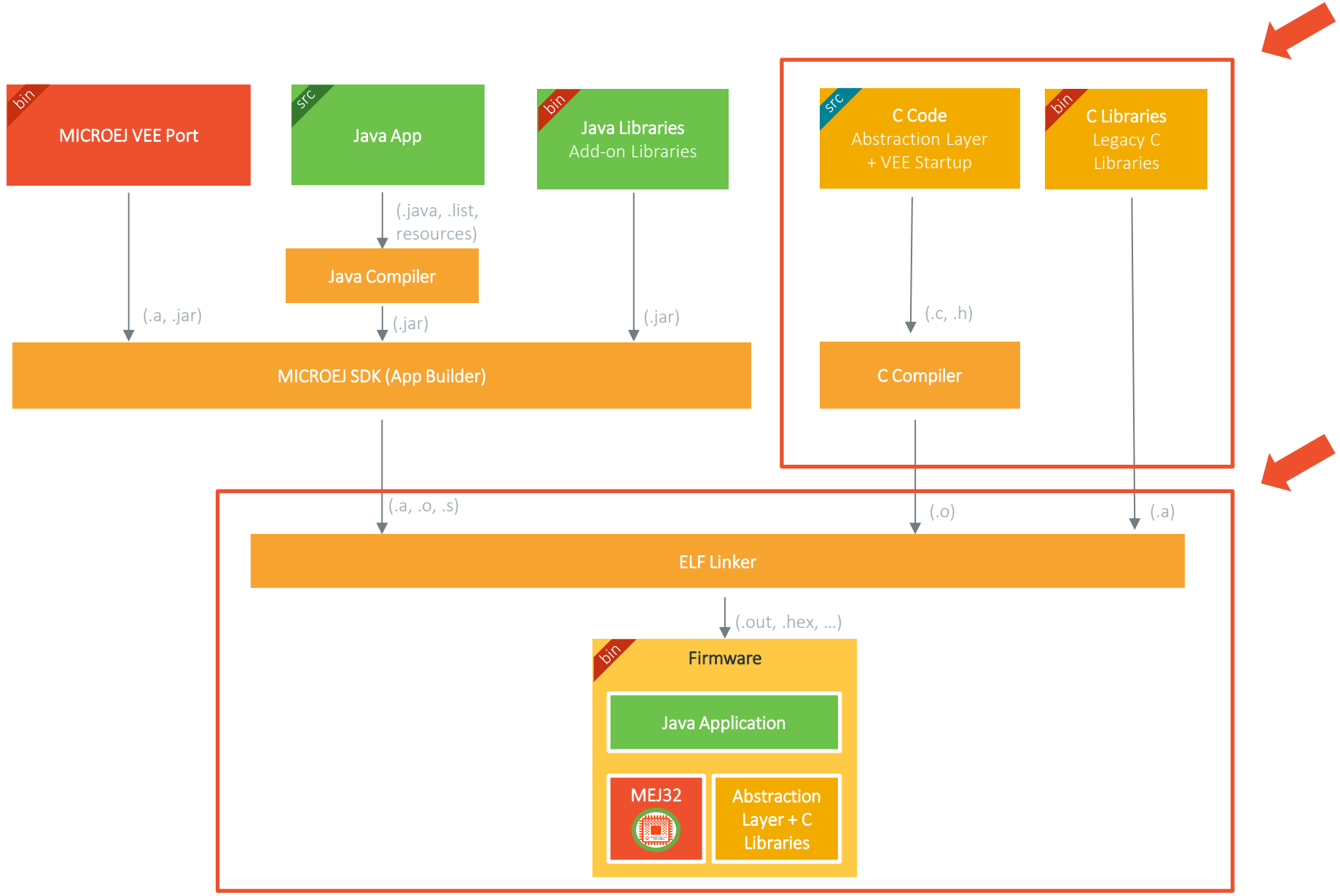
- Right-Click on the Project.
- **Run As -> Run Configuration.**
- Double click on MICROEJ Application.
- Go to **Execution** tab.
- Select **Execute on Device.**
- Click **Run**.



Expect the following output in the console:

```
=============== [ Initialization Stage ] ===============
VEE Port connected to BSP location 'C:\workspaces\nxpvee-mimxrt1170-prj\nxpvee-mimxrt1170-evk/nxpvee-mimxrt1170-evk-bsp' using platform option 'root.dir' in 'bsp/bsp.properties'.
[INFO ] Launching in Evaluation mode. Your UID is XXX.
=============== [ Launching SOAR ] ===============
=============== [ Launching Link ] ===============
=============== [ Deployment ] ===============
MicroEJ files for the 3rd-party BSP project are generated to 'C:\workspaces\HelloWorld\com.microej.training.Main\bsp'.
MicroEJ application (microejapp.o) has been deployed to: 'C:\workspaces\nxpvee-mimxrt1170-prj\nxpvee-mimxrt1170-evk\nxpvee-mimxrt1170-evk-bsp\projects\microej\platform\lib'.
MicroEJ library (microejruntime.a) has been deployed to: 'C:\workspaces\nxpvee-mimxrt1170-prj\nxpvee-mimxrt1170-evk\nxpvee-mimxrt1170-evk-bsp\projects\microej\platform\lib'.
MicroEJ header files (*.h) have been deployed to: 'C:\workspaces\nxpvee-mimxrt1170-prj\nxpvee-mimxrt1170-evk\nxpvee-mimxrt1170-evk-bsp\projects\microej\platform\inc'.
=============== [ Completed Successfully ] ===============

SUCCESS
```

BUILD FLOW



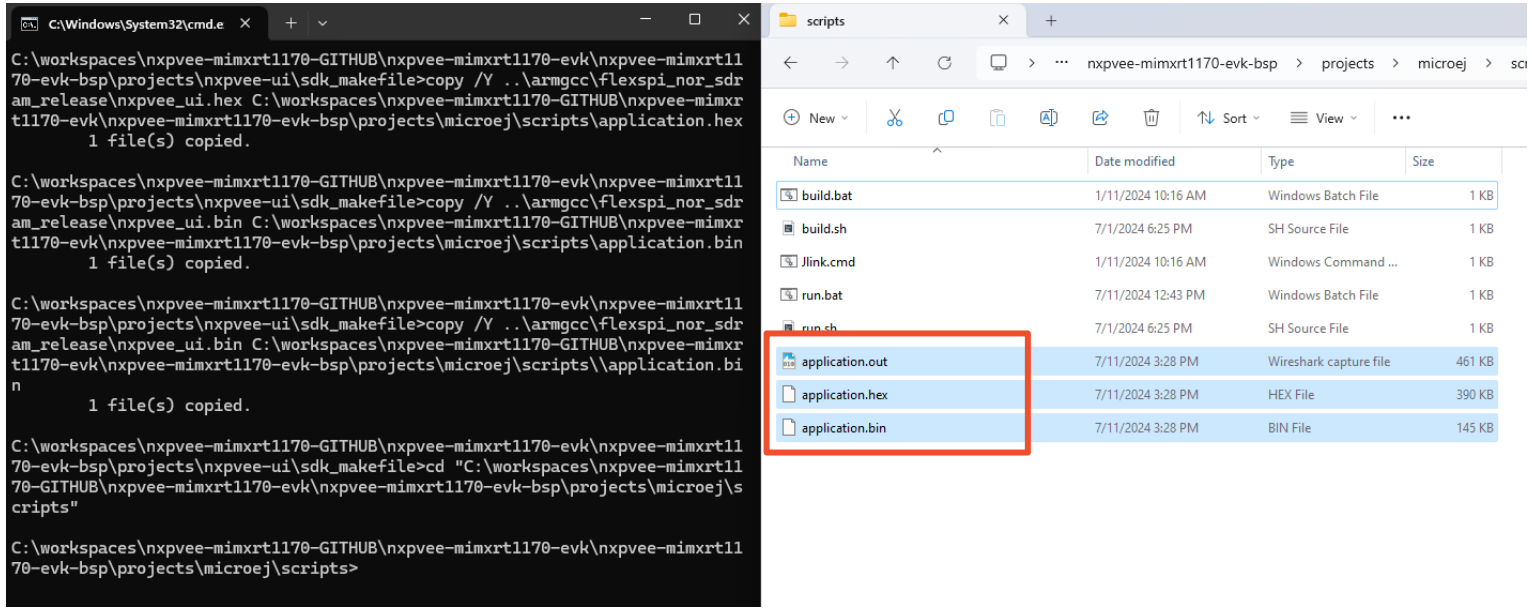
MICROEJ AND 3RD PARTY IDE

- Application must be **linked with BSP:**
  - BSP = drivers + (optional: operating system) + abstraction layer.
  - Done by a 3rd party IDE.
- 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 IDE is responsible for **compiling BSP, linking, and generating an Firmware file.**

BUILD THE FIRMWARE (1/2)

A build script is provided in the VEE Port project to build the application firmware.
This script uses a GCC Cortex-M toolchain Toolchain to build the BSP project of the VEE Port.

- Open a terminal in the **nxpvee-mimxrt1170-evk-bsp\projects\microej\scripts** folder
- Run the **build.bat** script
- Wait for the end of the build.
- Firmware binaries are generated in the **scripts/** folder:

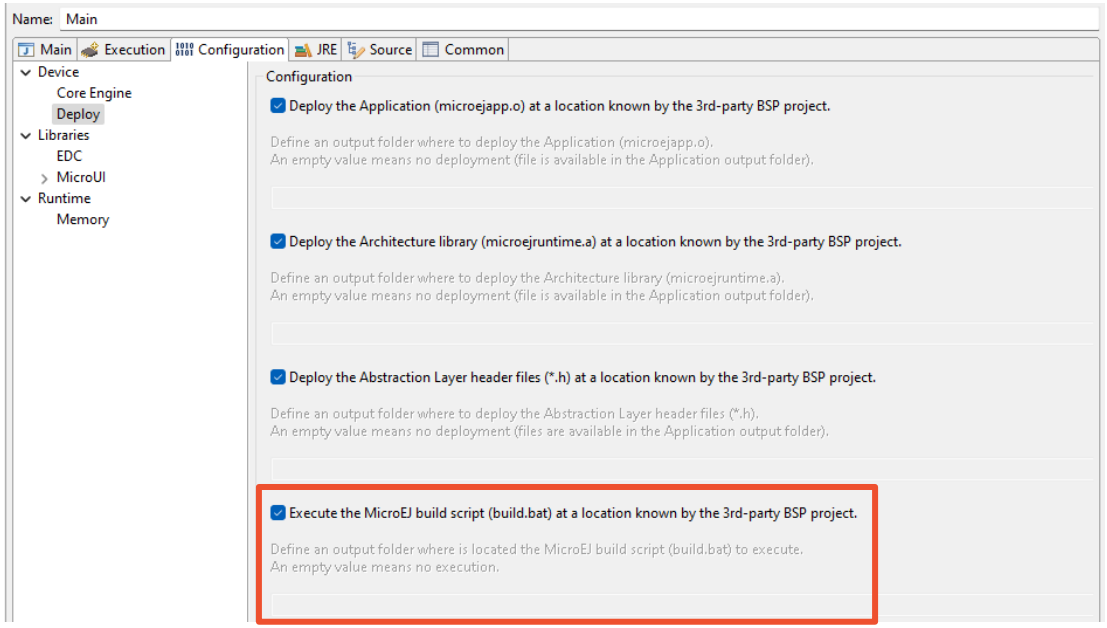


BUILD THE FIRMWARE (2/2)

The firmware build can also be triggered from MICROEJ SDK.

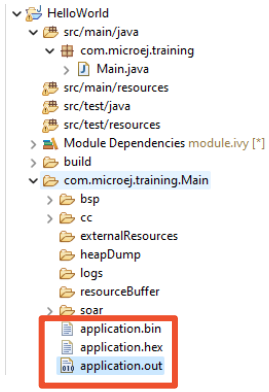
In MICROEJ SDK:

- Right-Click on the Project.
- **Run As -> Run Configuration.**
- Select the launcher corresponding to the application.
- Go to **Configuration** tab.
- Check **Execute the MicroEJ build script (build.bat)...**
- Click **Run**.



The Firmware build is triggered from MICROEJ SDK, once completed, firmware binaries are available in the application output folder:

```
C:\workspaces\nxpvee-mimxrt1170-GITHUB\nxpvee-mimxrt1170-evk\nxpvee-mimxrt1170-evk-bsp\projects\nxpvee-ui\sdk_makefile>copy /Y ..\armgcc\flexspi_nor_sdram_release\nxpvee_ui.hex
        1 file(s) copied.

C:\workspaces\nxpvee-mimxrt1170-GITHUB\nxpvee-mimxrt1170-evk\nxpvee-mimxrt1170-evk-bsp\projects\nxpvee-ui\sdk_makefile>copy /Y ..\armgcc\flexspi_nor_sdram_release\nxpvee_ui.bin
        1 file(s) copied.

C:\workspaces\nxpvee-mimxrt1170-GITHUB\nxpvee-mimxrt1170-evk\nxpvee-mimxrt1170-evk-bsp\projects\nxpvee-ui\sdk_makefile>copy /Y ..\armgcc\flexspi_nor_sdram_release\nxpvee_ui.bin
        1 file(s) copied.

C:\workspaces\nxpvee-mimxrt1170-GITHUB\nxpvee-mimxrt1170-evk\nxpvee-mimxrt1170-evk-bsp\projects\nxpvee-ui\sdk_makefile>cd "C:\workspaces\NXP_TMP\HelloWorld\com.microej.training.
Execution of script 'C:\workspaces\nxpvee-mimxrt1170-GITHUB\nxpvee-mimxrt1170-evk\nxpvee-mimxrt1170-evk-bsp\projects\microej\scripts\build.bat' done.
=============== [ Completed Successfully ] ===============

SUCCESS
```

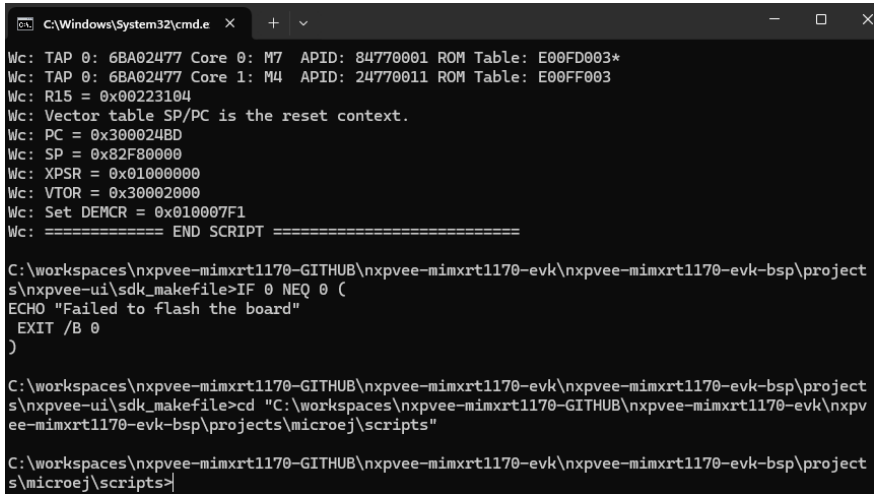


RUN THE APPLICATION ON DEVICE (1/3)

FLASH THE FIRMWARE

A run script is provided in the VEE Port project to flash the firmware.

- Open a terminal in the **nxpvee-mimxrt1170-evk-bsp\projects\microej\scripts** folder
- Run the **run.bat** script
- Wait for the end of the programming:



- Note: the “Failed to flash the board” message can be ignored, it will be fixed in the next release of the VEE Port.

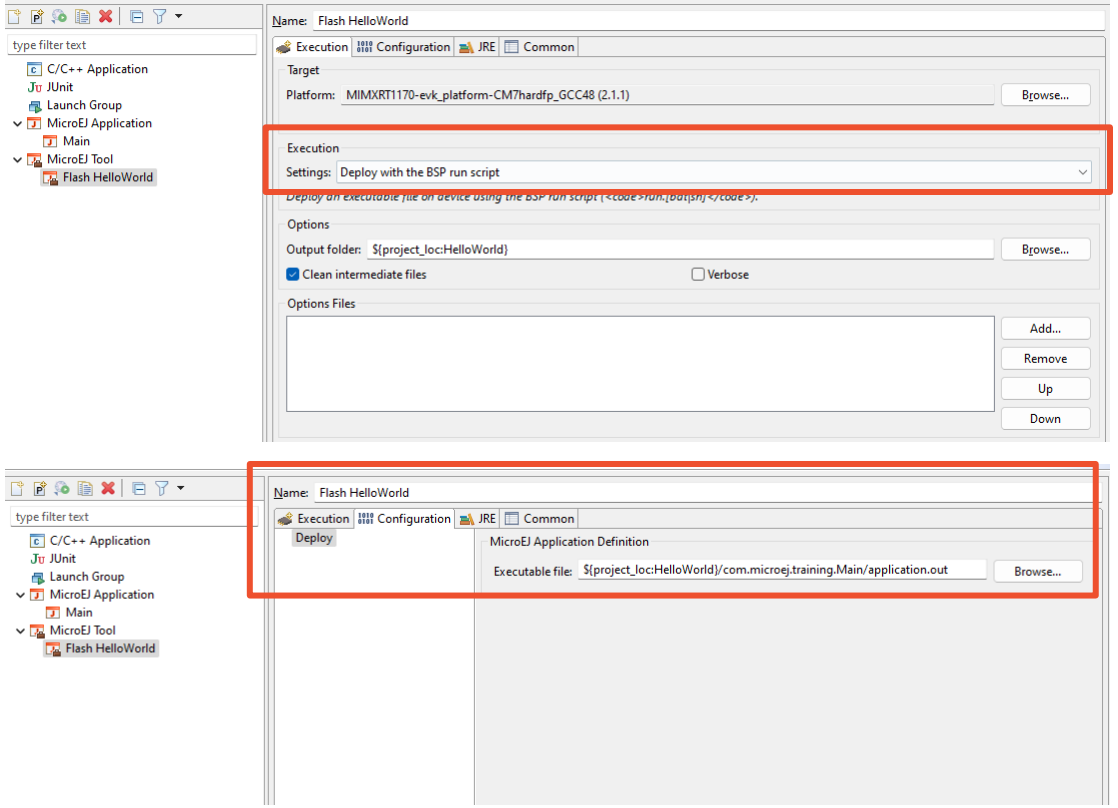
RUN THE APPLICATION ON DEVICE (2/3)

FLASH THE FIRMWARE

The firmware can also be flashed from MICROEJ SDK.

In MICROEJ SDK:

- Right-Click on the Project.
- **Run As -> Run Configuration.**
- Double Click on **MicroEJ Tool.**
- In the **Execution** section of the **Execution** tab, select: **Deploy with the BSP run script.**
- In the **Configuration** tab, select the previously generated **application.out.**
- Click **Run**.

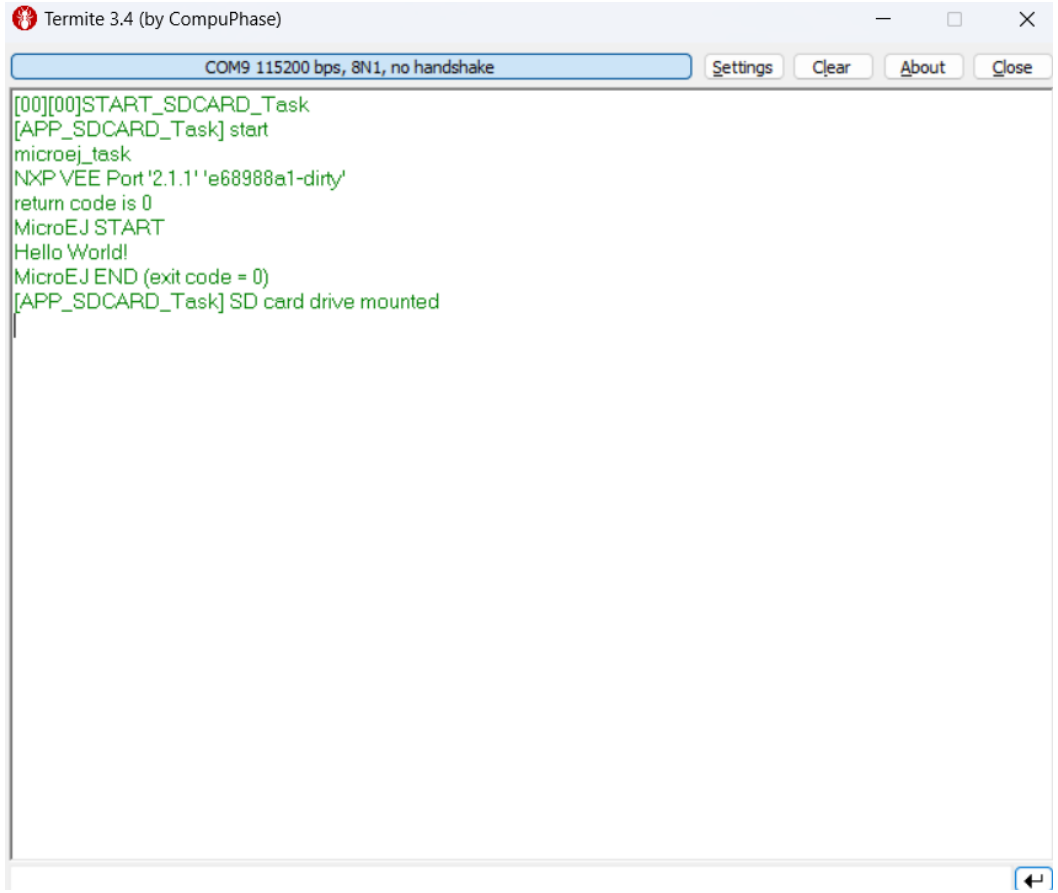



```
Wc: XPSR = 0x01000000
Wc: VTOR = 0x30002000
Wc: Set DEMCR = 0x010007F1
Wc: ============= END SCRIPT ============================

C:\workspaces\nxpvee-mimxrt1170-GITHUB\nxpvee-mimxrt1170-evk\nxpvee-mimxrt1170-evk-bsp\projects\nxpvee-ui\sdk_makefile>IF 0 NEQ 0 (
ECHO "Failed to flash the board"
 EXIT /B 0
)

C:\workspaces\nxpvee-mimxrt1170-GITHUB\nxpvee-mimxrt1170-evk\nxpvee-mimxrt1170-evk-bsp\projects\nxpvee-ui\sdk_makefile>cd "C:\workspaces\NXP_TMP\Hell
Execution of script 'C:\workspaces\nxpvee-mimxrt1170-GITHUB\nxpvee-mimxrt1170-evk\nxpvee-mimxrt1170-evk-bsp\projects\microej\scripts\run.bat' done.

SUCCESS
```

RUN THE APPLICATION ON DEVICE (3/3)

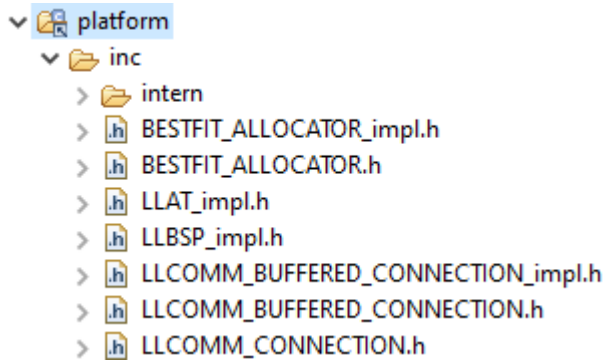
GET THE APPLICATION TRACES

- 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 and the **Hello World** message is printed in the console!

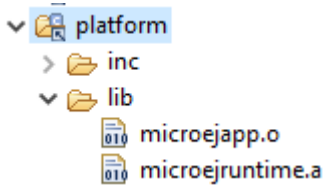


MICROEJ CORE ENGINE STARTUP

- MicroEJ header files are in: **projects/microej/platform/inc**



- MicroEJ libraries and Java application object file are used during link edition:



- MicroEJ Core Engine is invoked in: **projects/microej/core/src/microej_main.c** with **SNI_createVM():**

```
// 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

LIBRARY DEPENDENCY FILE

Contains a description of all the libraries required by the application.

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

Loaded by the MicroEJ Module Manager (MMM) to fetch automatically the dependencies using Ivy.

Available MICROEJ libraries can be found here:

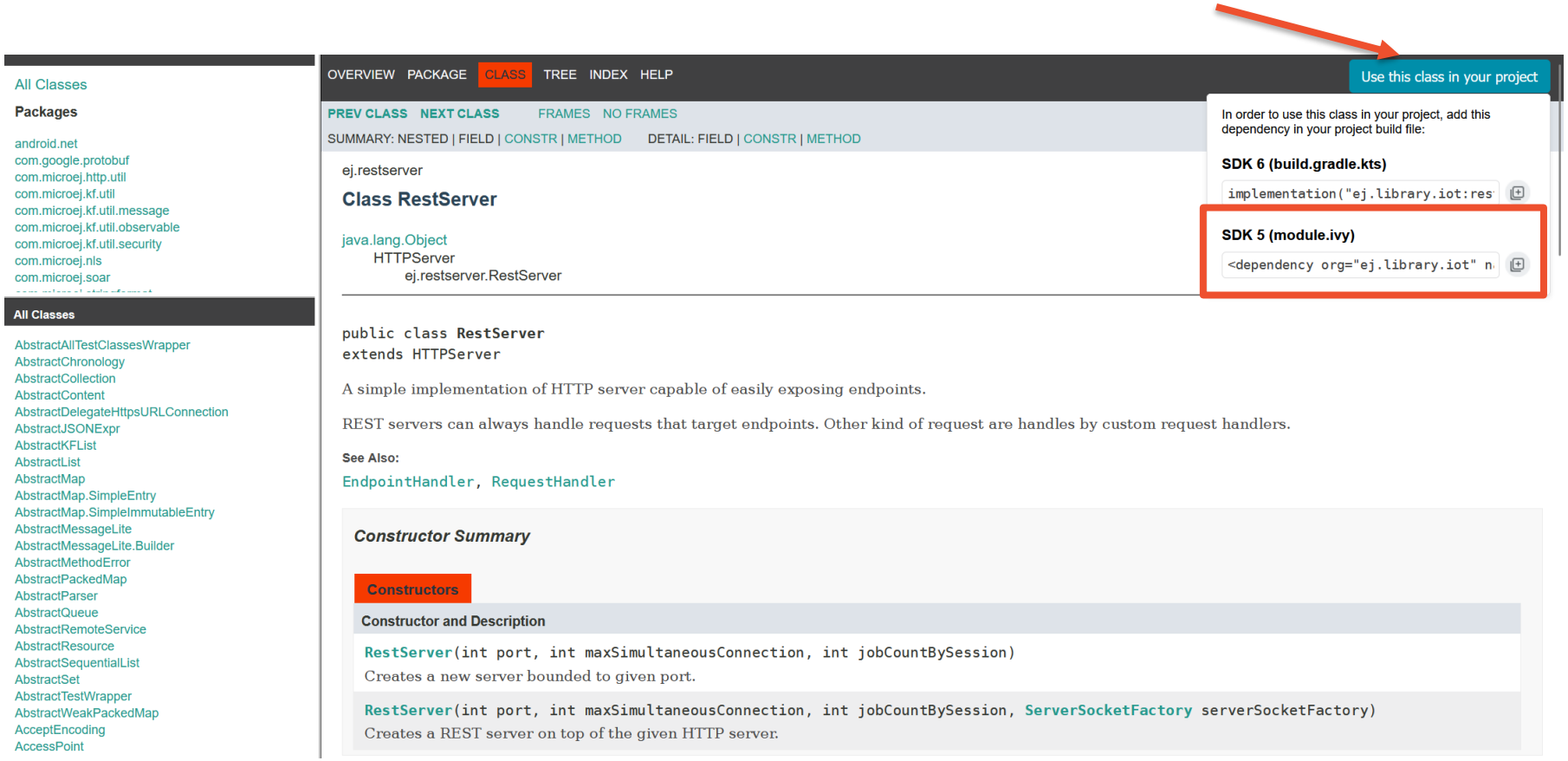
- Central Repository: https://repository.microej.com/
- Developer Repository: https://forge.microej.com/artifactory/microej-developer-repository-release/

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

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.



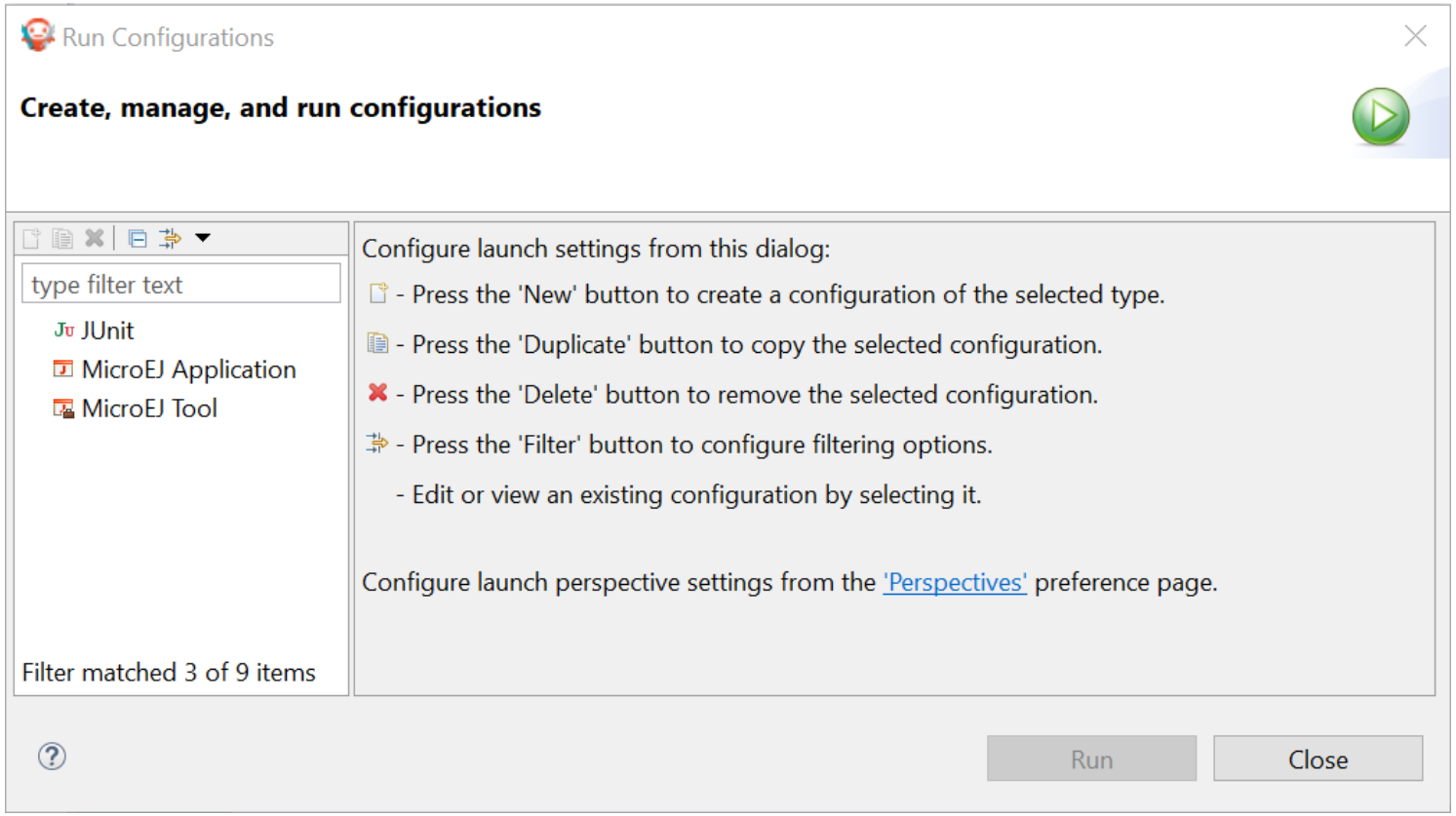
RUN CONFIGURATIONS (1/2)

- Run Configurations:
  - Eclipse provides the concept of “run configurations”
  - A run configuration tells what is executed, what is the runtime environment, what are the execution options
  - Available through the Run menu
- A Run Configuration can be executed as:
  - A Run Configuration to simply run an application 
  - A Debug Configuration to debug this application 
- External Tool Configuration to run an external program 

RUN CONFIGURATIONS (2/2)

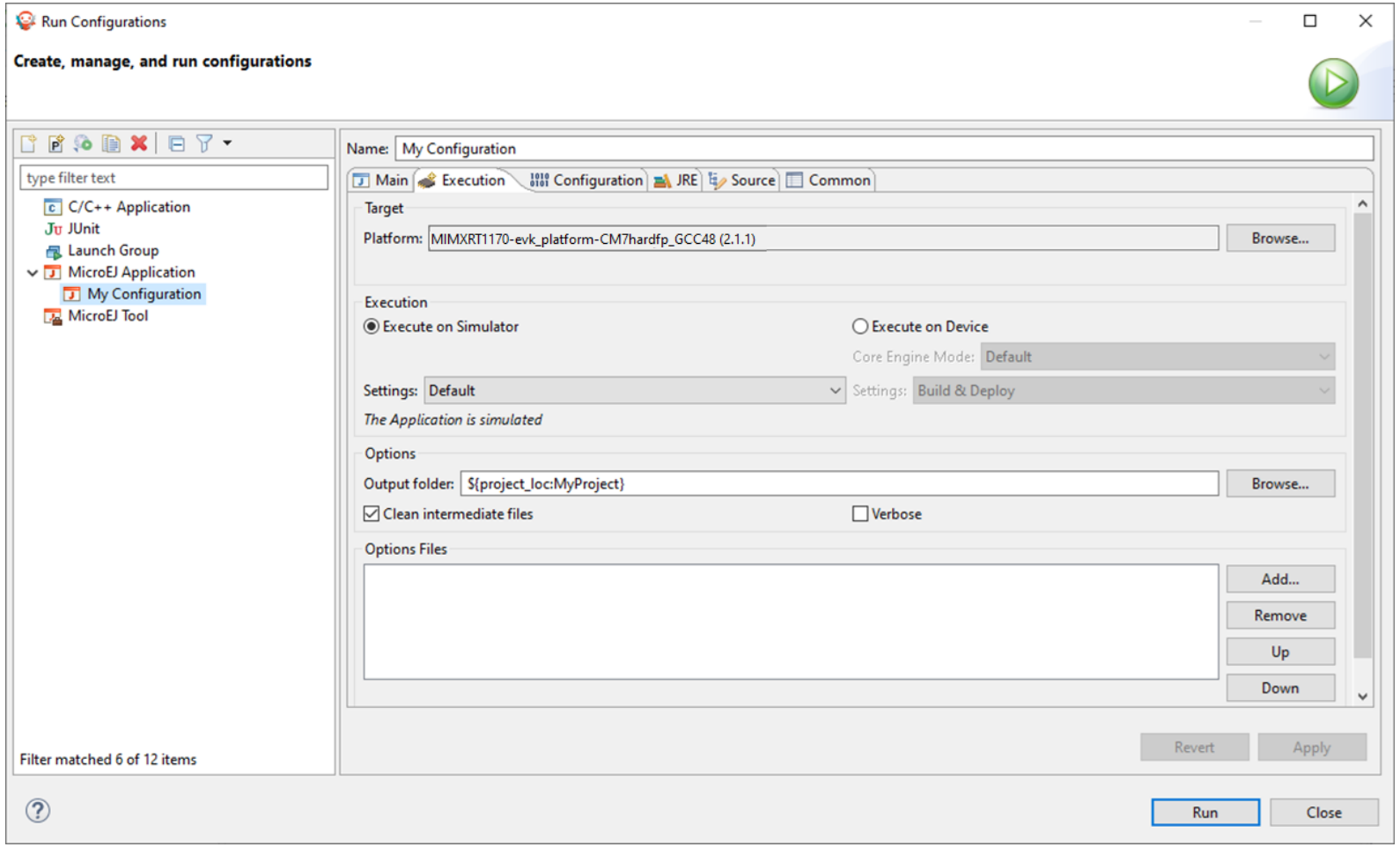
MICROEJ provides two specific run configuration types:

- MICROEJ Application
- MICROEJ Tool



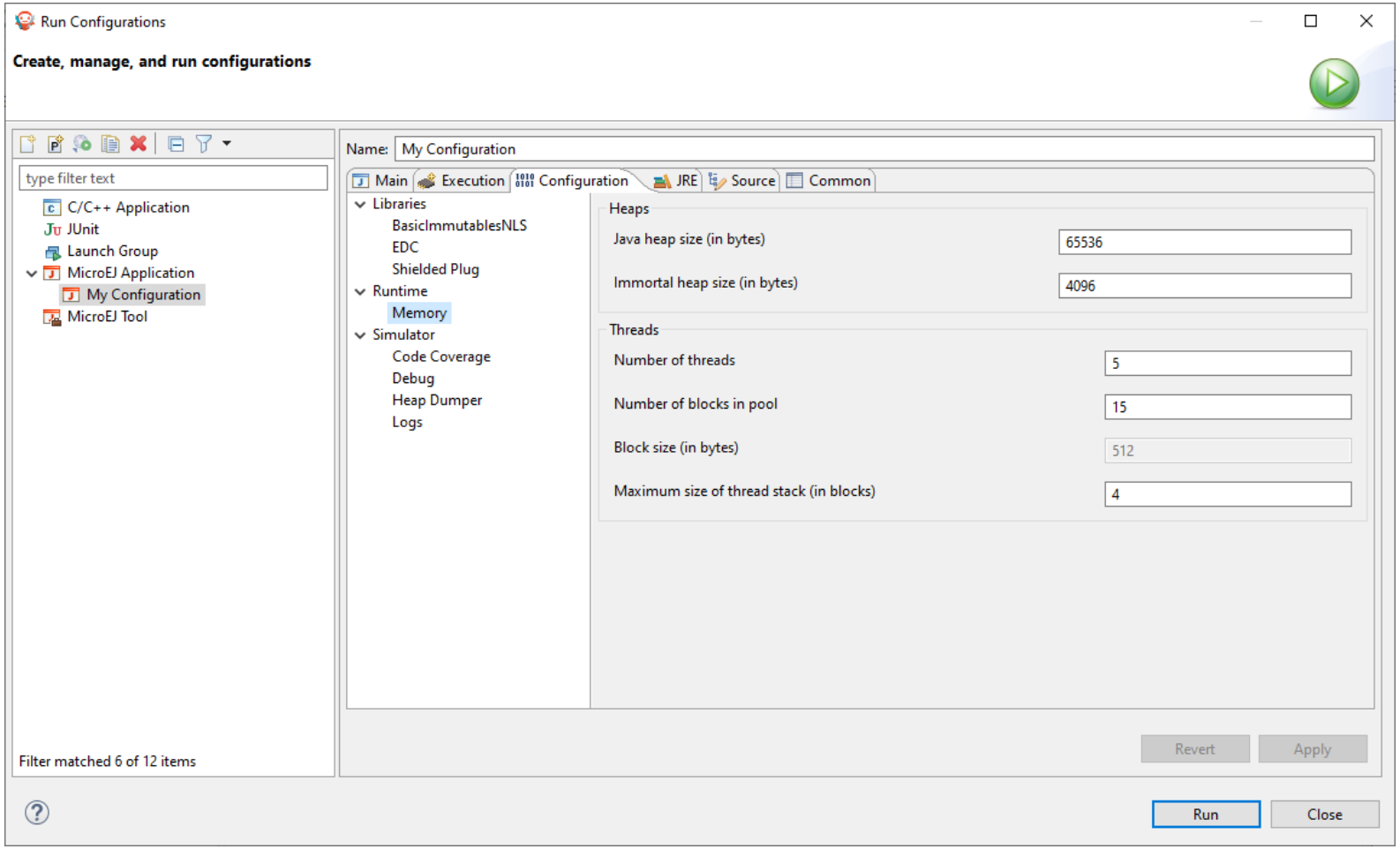
MICROEJ APPLICATION LAUNCHER (1/3)

KIND OF EXECUTION (SIMULATOR OR DEVICE)



MICROEJ APPLICATION LAUNCHER (2/3)

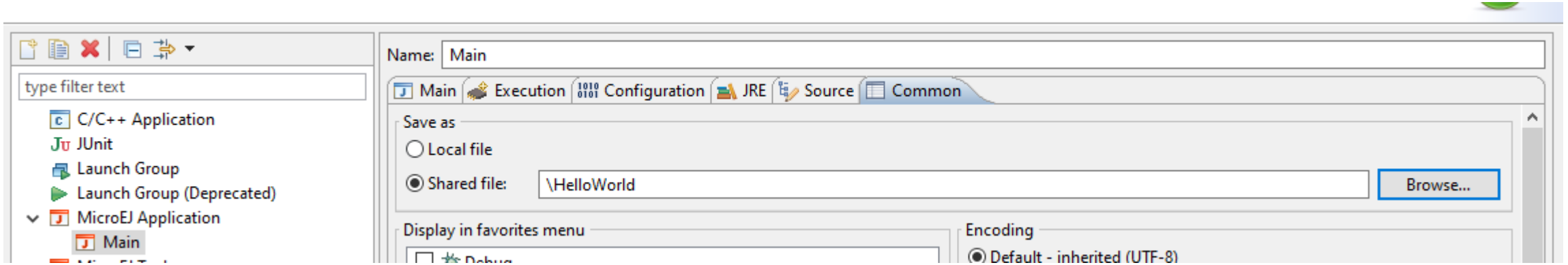
CONFIGURE LIBRARIES AND MEMORY USAGE



MICROEJ APPLICATION LAUNCHER (3/3)

SHARE RUN CONFIGURATIONS

1. Go to **Run -> Run Configurations**
2. Select a run configuration
3. In **Common** tab, select **Save as Shared file** and choose the directory where it is saved
4. You can now commit the **.launch** file in your Version Control System



Front Panel

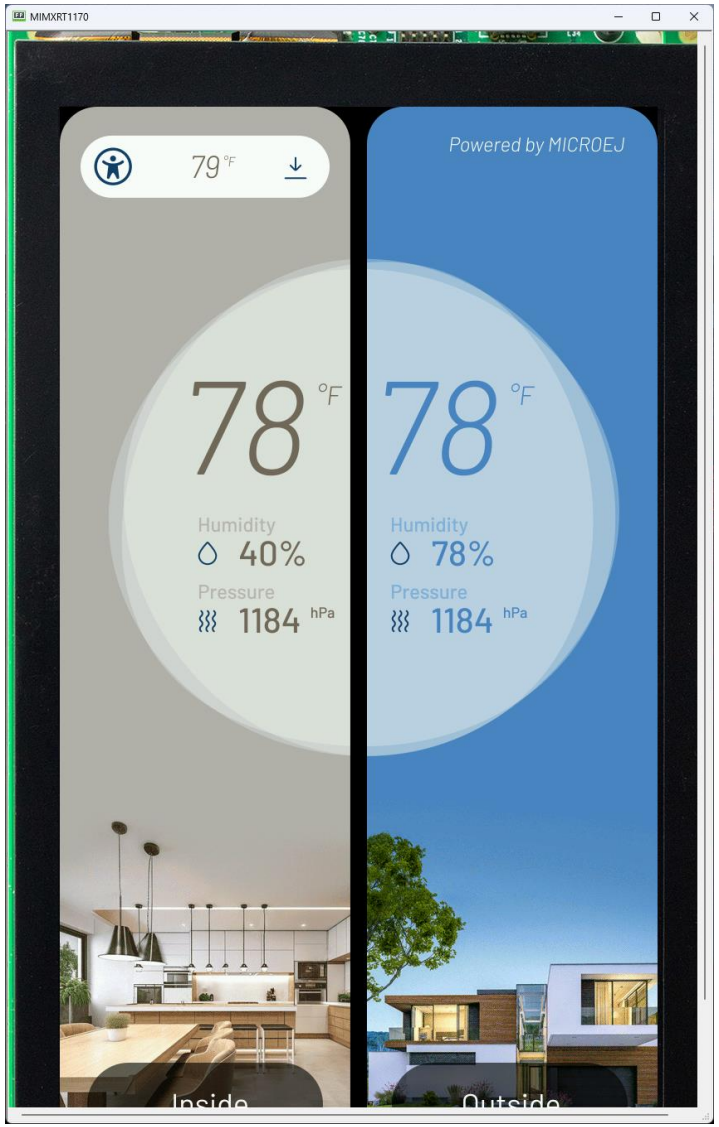
Customization

FRONT PANEL

PRINCIPLE

- MICROEJ environment 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 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 implements MicroUI. However it can be use to show a hardware device, blink an LED, interact with user without using MicroUI library.

See
https://docs.microej.com/en/latest/PlatformDeveloperGuide/frontpanel.html

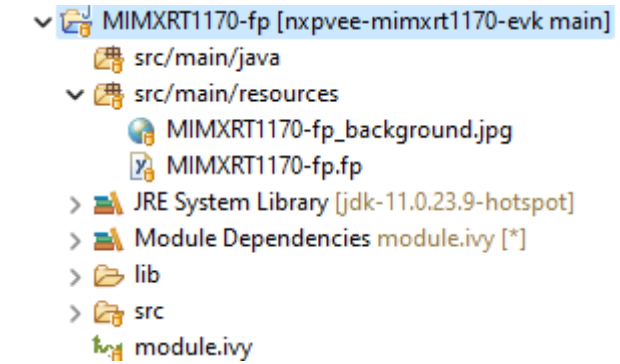


FRONT PANEL

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).
- **JRE System Library:** required to compile the custom widgets and listeners.
- **Modules Dependencies:** contains front panel framework and default widgets.
- **lib/:** contains a local copy of Modules Dependencies.



FRONT PANEL

FRONT PANEL 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

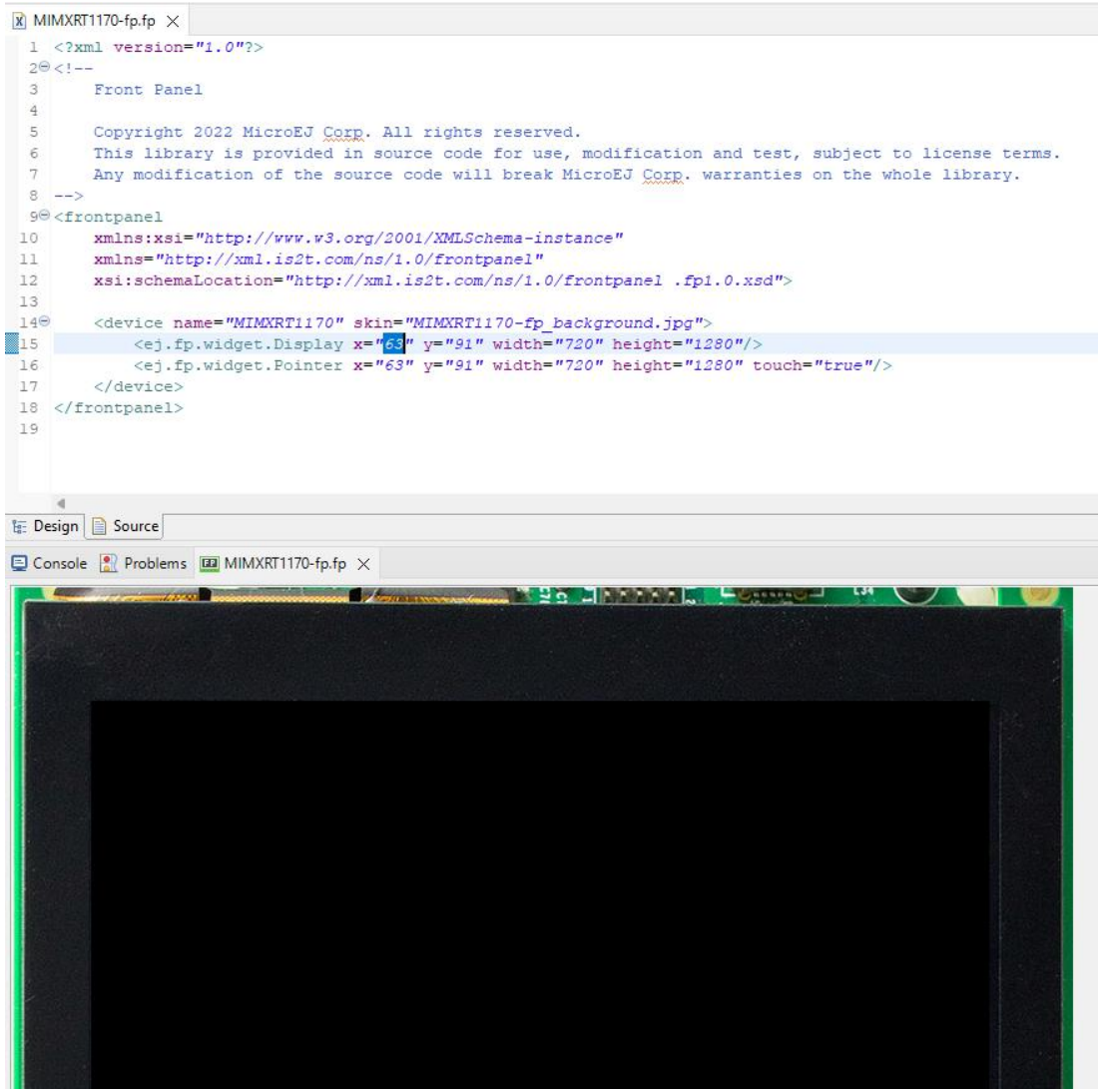
```
<?xml version="1.0"?>
<!--
    Front Panel

    Copyright 2022 MicroEJ Corp. All rights reserved.
    This library is provided in source code for use, modification and test, subject to license terms.
    Any modification of the source code will break MicroEJ Corp. warranties on the whole library.
-->
<frontpanel
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://xml.is2t.com/ns/1.0/frontpanel"
    xsi:schemaLocation="http://xml.is2t.com/ns/1.0/frontpanel .fp1.0.xsd">

    <device name="MIMXRT1170" skin="MIMXRT1170-fp_background.jpg">
        <ej.fp.widget.Display x="63" y="91" width="720" height="1280"/>
        <ej.fp.widget.Pointer x="63" y="91" width="720" height="1280" touch="true"/>
    </device>
</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.

FRONT PANEL

EDITING THE FRONT PANEL

- To edit a **.fp** file, open it using the **Eclipse XML editor:**
- Right-Click on the **.fp** file, select **Open With > XML Editor** and select the **Source** tab.
- Within the XML editor, content-assist is obtained by pressing **CTRL + SPACE** keys.
- To obtain a preview of the Front Panel, go to **Window > Show View > Other… > MICROEJ > Front Panel Preview**.
- The preview is updated each time the **.fp** file is saved.
- The VEE Port needs to be rebuilt to get the Front Panel updates.



MICROEJ SDK TOOLS

STACK TRACE READER

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:

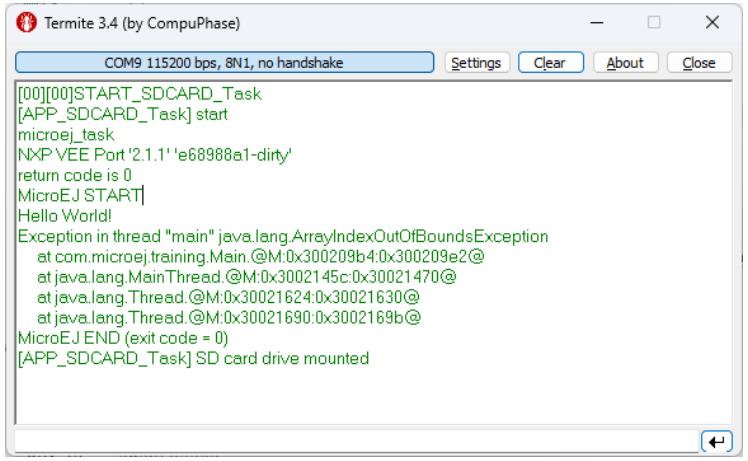
```
byte[] array = new byte[5];
array[5] = 42; // Invalid access to the array
```

- Compile the application in MICROEJ SDK:
  1. Right click on the HelloWorld MICROEJ project.
  2. **Run as -> MicroEJ Application.**
- Build the BSP Project.
- Flash the board.

STACK TRACE READER

EXCEPTION OUTPUT

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



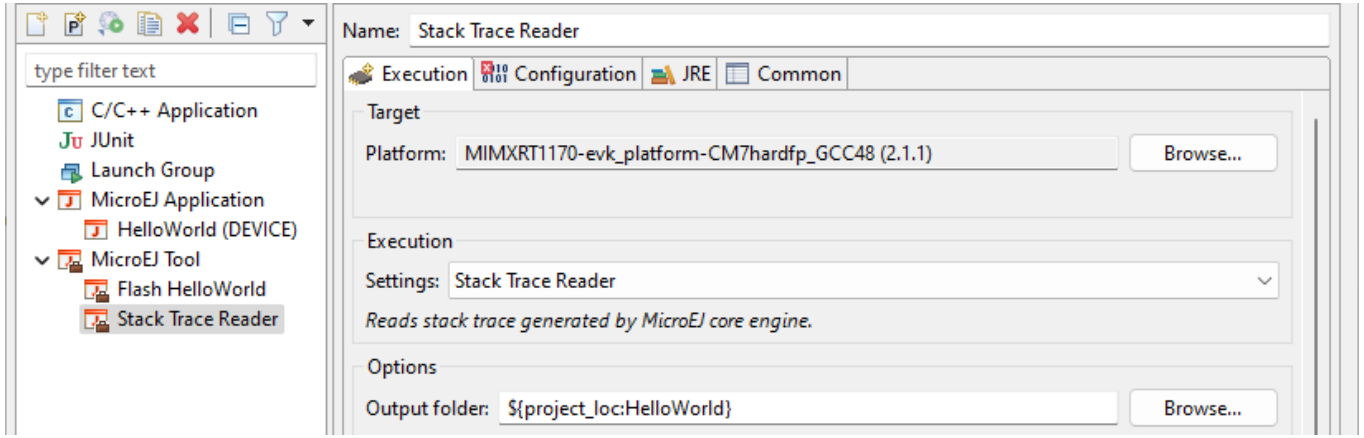
- Name of the faulty **method is not printed** directly:
  - Only the address of the method is printed
  - MICROEJ does not embed the names of the methods to limit the footprint
- To help reading the stack trace, a tool is available: **the stack trace reader**

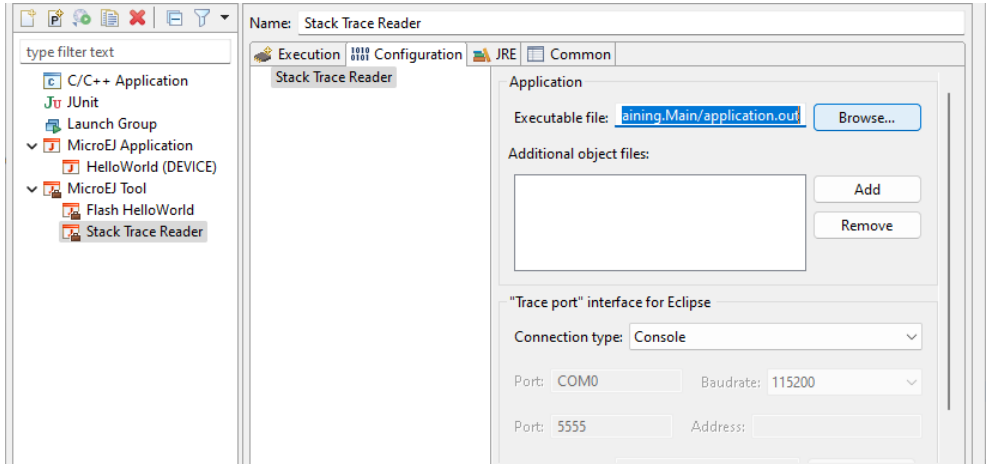
STACK TRACE READER

CONFIGURATION

In MICROEJ SDK, create the Run configuration

1. Go to Run -> Run Configurations…
2. Double-click on **MicroEJ Tool.**
3. Enter a name for the launcher.
4. Select your VEE Port.
5. Use settings: **Stack Trace Reader.**
6. Go to **Configuration** tab.
7. Use the ELF file generated by the 3rd party linker:





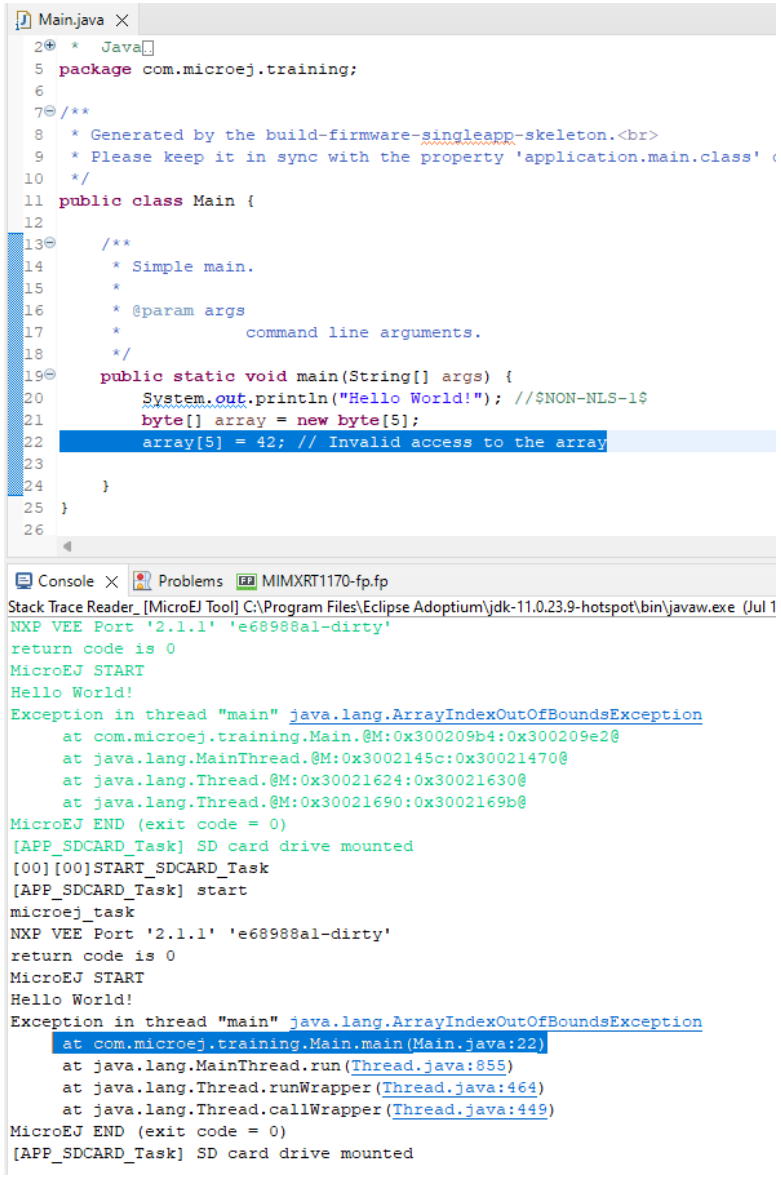
STACK TRACE READER

USAGE

1. Click **Run**
2. **Copy/Paste** the trace in your console

You can also configure it to read data directly from the com port of your device.

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



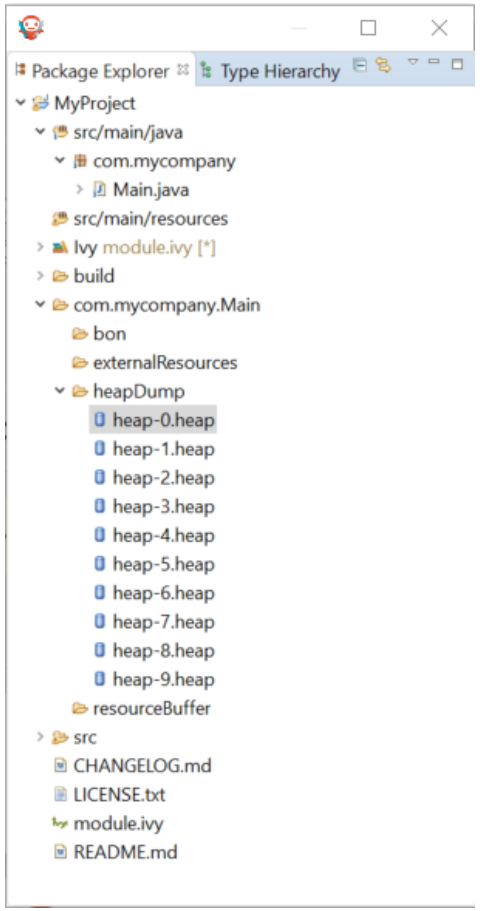
DEBUGGER

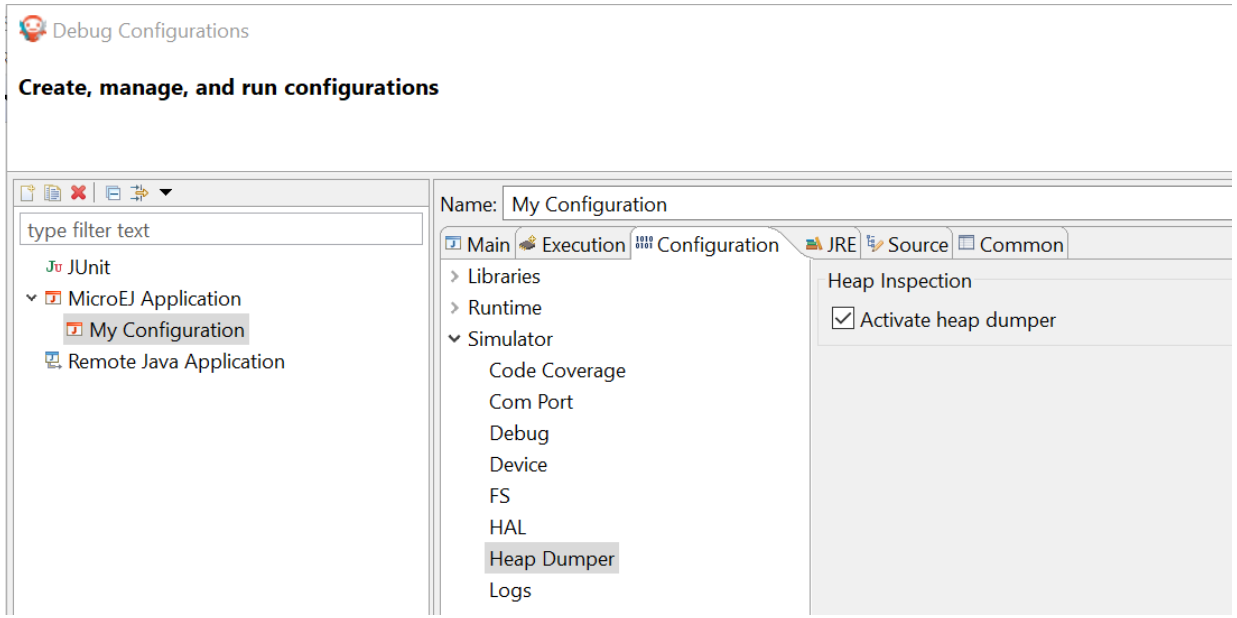
- JDWP (Java Debug Wire Protocol) to use Eclipse debugger.
- Classical debugger features:
  - Breakpoints.
  - Step-by-step execution.
  - Variables and fields value monitoring.
  - Thread execution stacks list.
- Run your Launch Configuration as a Debug Configuration:
  - Debug perspective.

HEAP DUMPER (1/2)

A heap file, describing the heap content, is created each time garbage collector is executed:

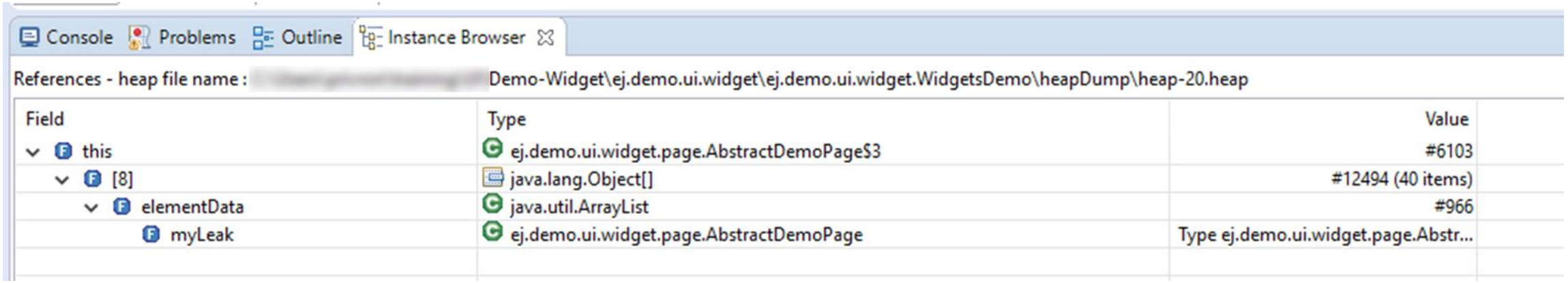
- **System.gc()** to force heap dumping:





HEAP DUMPER (2/2)

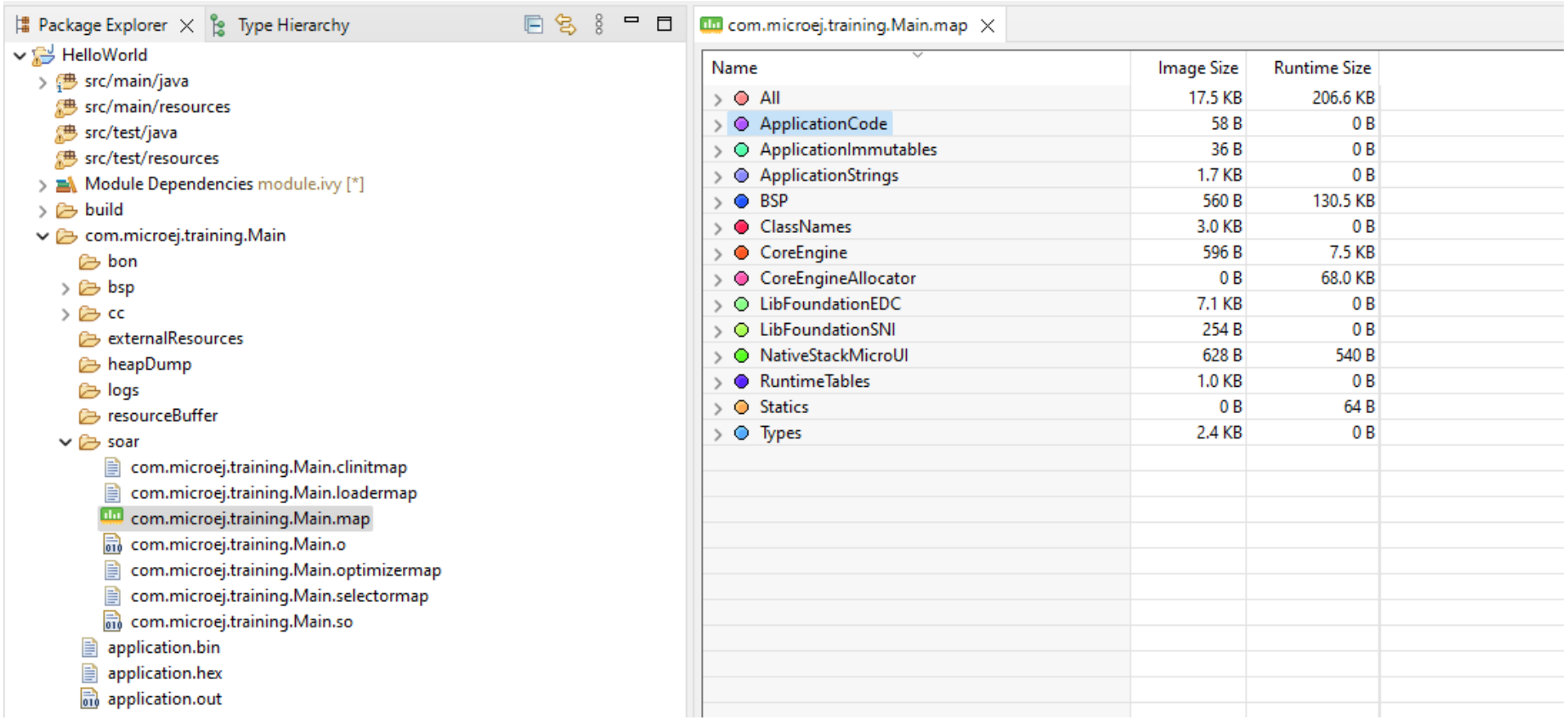
- Open .heap files with the Heap Analyzer plugin.
- Inspect objects graph.
- Detect memory leaks.
- This is an advanced feature: a good knowledge of Java and the program is required.



MEMORY MAP INSPECTOR

A **.map** file is generated when a build for device is done.
The map file analyses the memory usage of the Application and its dependencies.

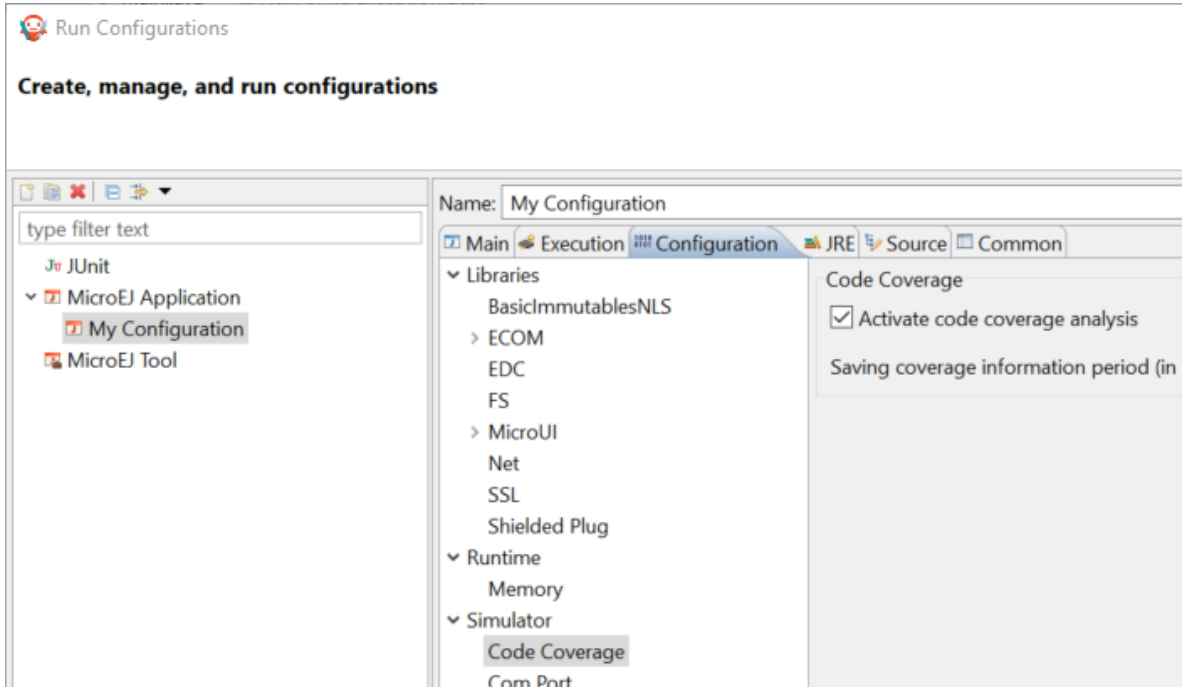
It does not include the memory usage of the BSP project.



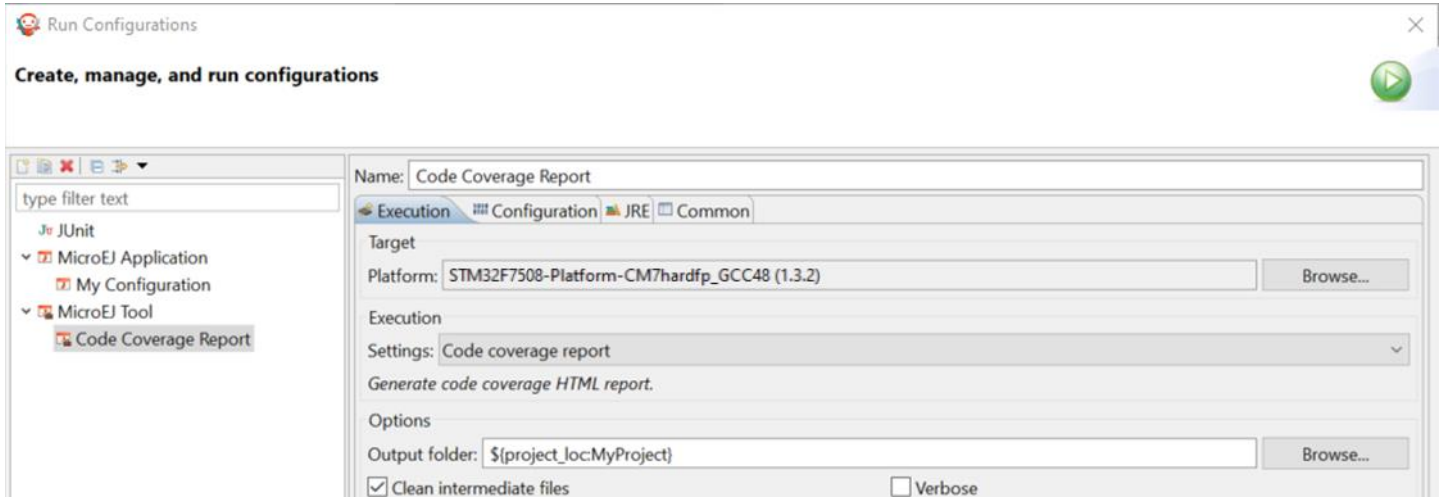
CODE COVERAGE (1/2)

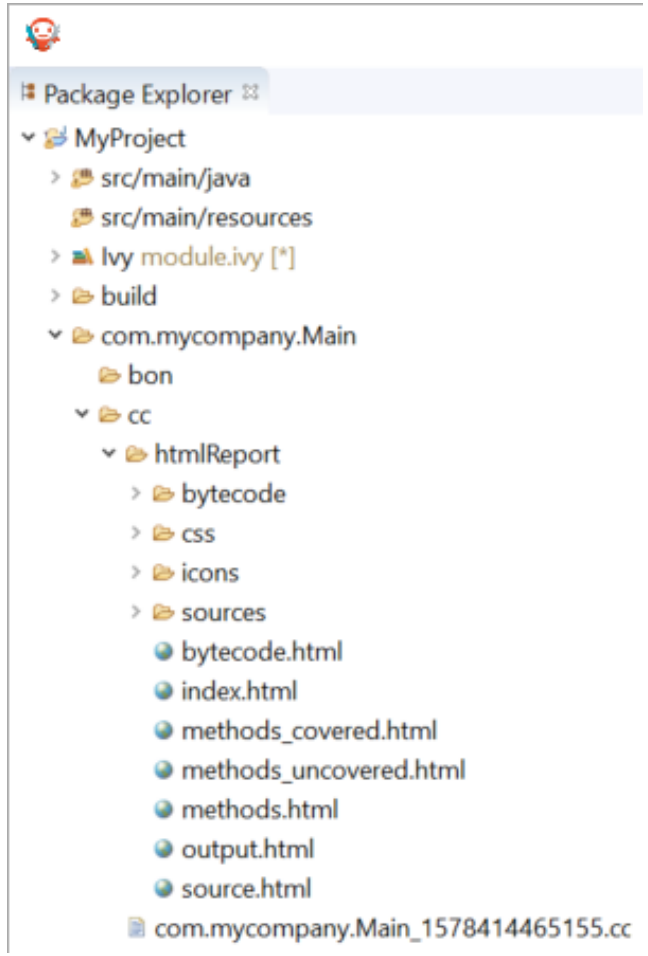
Code coverage reports:

- List used and unused source code.
- Find untested or dead code.
- HTML report generation.



CODE COVERAGE (2/2)





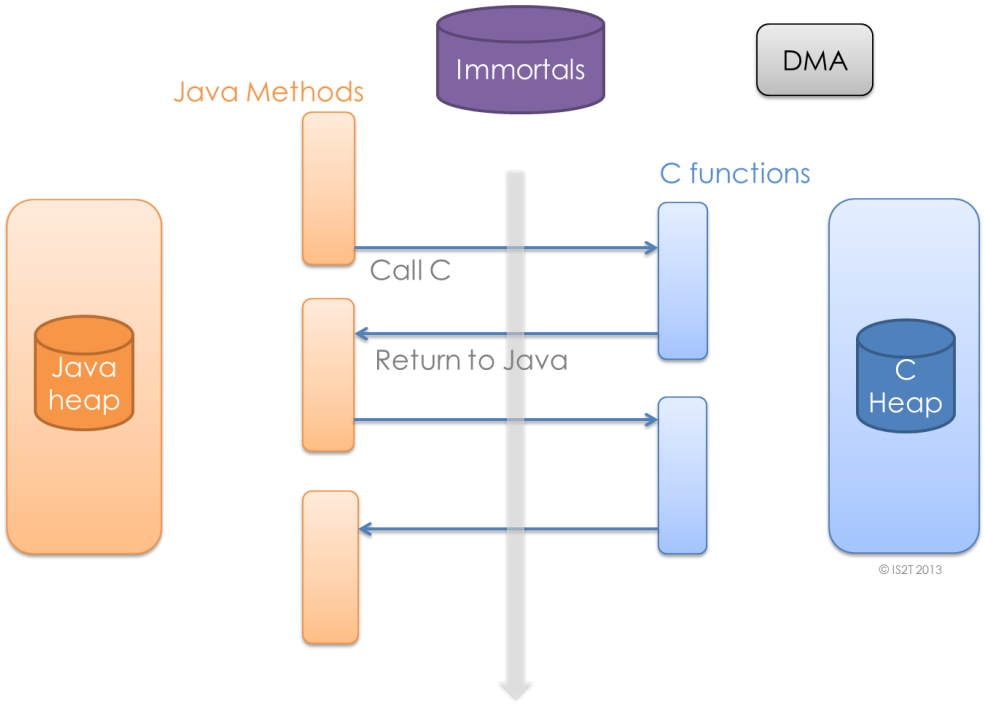
SNI

SNI (Simple Native Interface)

Call C code from Java

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

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.

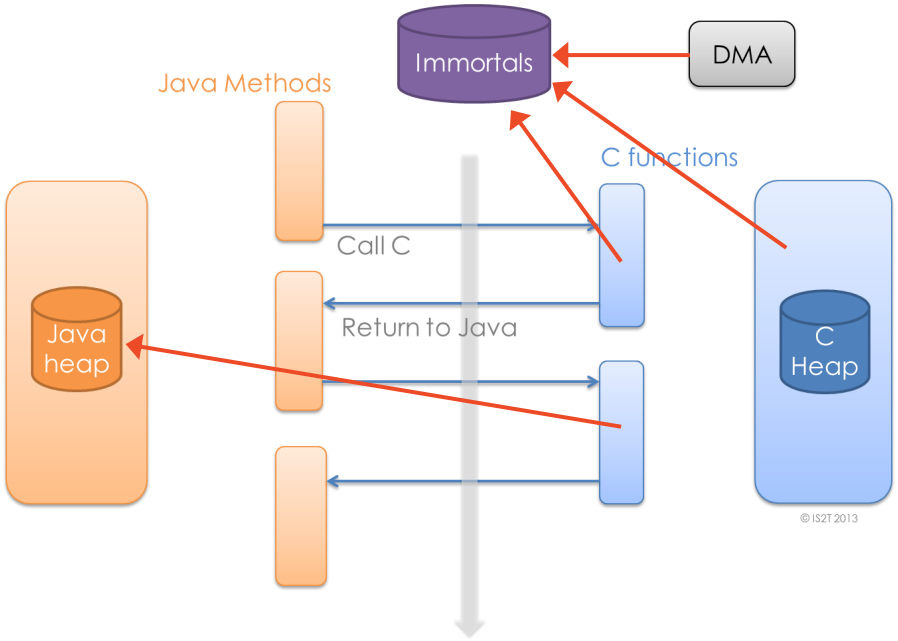
NAMING CONVENTION

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

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

```
#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 from a native function local.



Implement a Java Native Method with SNI

ADD THE JAVA NATIVE METHOD

- Modify the code of the HelloWorld main method:

```
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);
```

- Compile the application in MICROEJ SDK:
  - Right click on the HelloWorld MICROEJ project.
  - **Run as -> MicroEJ Application.**
  - Run the launcher configured to **Execute on Device.**

Note: make sure that the **Execute the MicroEJ build script (build.bat)** option is enabled in the Application launcher.

GET THE LINKER ERRORS

The following error appears in the console:

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

IMPLEMENT THE NATIVE METHOD IN THE BSP

- Open **microjvm_main.c**
- Implement the **multiplyByTwo(int value)** method, use the method signature provided by the linker error:

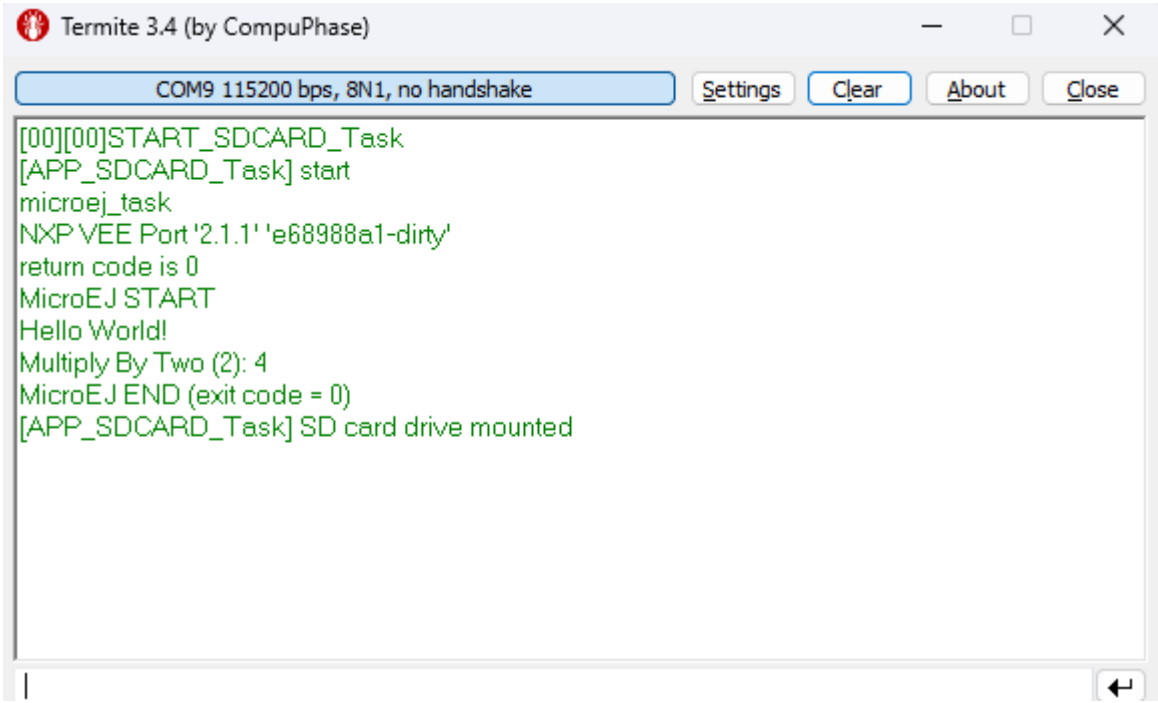
```
17 #include <stdio.h>
18 #include "microej_main.h"
19 #include "LLMJVM.h"
20 #include "sni.h"
21 #include "fsl_debug_console.h"
22
23 #ifdef __cplusplus
24     extern "C" {
25 #endif
26
27 jint Java_com_microej_training_Main_multiplyByTwo(jint value){
28     return value * 2;
29 }
```

- 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 DEVICE

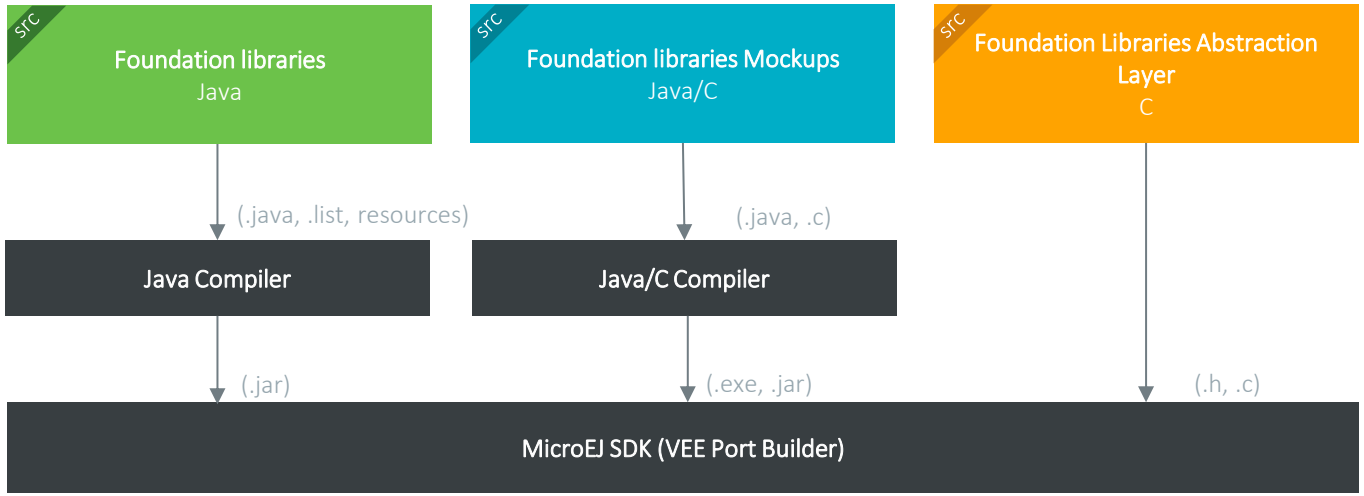
- Open the Termite serial terminal.
- Click the **Settings** button.
- Select the NXP i.MX RT1170 EVK board COM port.
- Reset the NXP i.MX RT1170 EVK board using Reset button
- The application starts: the **Hello World** message and the Multiplied by Two value is printed!



Foundation Library

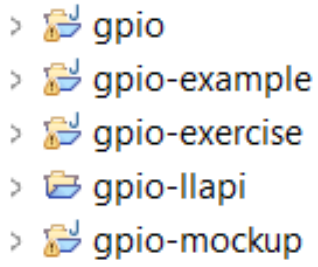
DEFINITION

- A Foundation library is a **Java library that depends on C code.**
- Composed of:
  - A main project with the **Java library source.**
  - Abstraction Layer Interface or Low Level API (LLAPI) specified in **C header files.**
  - A **mockup** of the Java library for the simulator.



FOUNDATION LIBRARY EXAMPLE

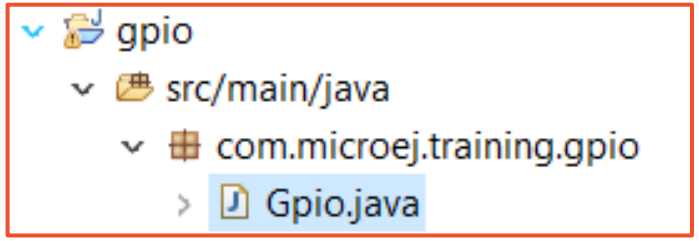
- Import the GPIO Foundation Library Example:
  - Open menu **File** > **Import...** > **General** > **Existing Projects into Workspace.**
  - Select the root directory **[training-package]/gpio-foundation-library-example-sdk5**
  - Select all the projects.
  - Click on Finish.
- If some projects don't compile click on Project > Clean... menu, select Clean all projects and click on Clean.



GPIO FOUNDATION LIBRARY

The **GPIO** class in the **gpio** project defines 2 native methods:

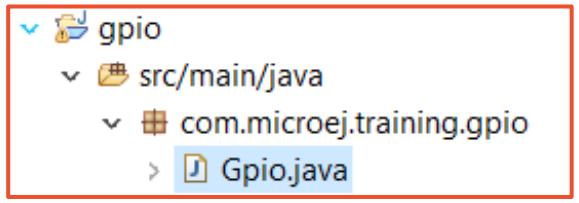
```
/**
 * GPIO management class.
 */
public class Gpio {
/**
 * Sets a value on the digital pin.
 *
 * @param pin
 *            the pin identifier
 * @param value
 *            digital pin value: true for high, false for low.
 */
native public static void set(int pin, boolean value);

/**
 * Gets the value of the digital pin.
 *
 * @param pin
 *            the pin identifier
 * @return true when the GPIO digital value is currently high, false otherwise.
 */
native public static boolean get(int pin);
}
```

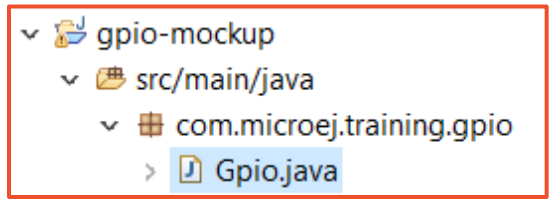


Run the Foundation Library Example on Simulator

MOCKUP IMPLEMENTATION

- The **gpio-mockup** project is a JavaSE Project.
- The implementation of the **gpio** native methods is done in a class having the same package and same name:





- Each native method is implemented, without the **native** and with the **public** modifiers:

```
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) {
		// Returns false by default
		return GPIO.getOrDefault(Integer.valueOf(pin), Boolean.FALSE).booleanValue();
	}
}
```

MOCKUP DEPLOYMENT

- Build the Mockup with MMM:
  - Right-Click on the **gpio-mockup** project and select **Build Module.**
  - A **.rip** named **gpio-mockup.rip** is generated in the **gpio-mockup\target~\artifacts** folder.
- Add it to the VEE Port:
  - Unzip the **gpio-mockup.rip**
  - Drop the content of the folder content into the project **[VEE Port]-[Version]/source/**

Warning: This folder is **overwritten** at each VEE Port build. To avoid that, add the mock module as a VEE Port dependency in the **–configuration/module.ivy**

Note: to ease the mock development phase, use the Resolve Foundation Library in workspace to retrieve mock sources in simulation → the above steps can be avoided during the development in MICROEJ SDK.

RUN ON THE SIMULATOR

- The project **gpio-example** contains an example that uses the **gpio** library:

```
private static final int PIN = 0;
private static final long DELAY = 500;

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

- The **gpio** library has been added as dependency in the module.ivy of **gpio-example:**

```
<dependency org="com.microej.training.gpio" name="gpio" rev="1.1.0"/>
```

- Right click on the MicroEJ project **gpio-example.**
- **Run as → MicroEJ Application.**

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

Run the Foundation Library Example on Device

RUN THE EXAMPLE ON DEVICE

- Build the **gpio-example** project for the device:
    - Go to **Run -> Run Configurations.**
    - Select the **gpio-example BlinkGpio** Run Configuration.
    - Go to **Execution Tab.**
    - Select **Execute on Device.**
    - Click **Run.**
- Compile, Link and Flash the Firmware.

Note: make sure that the **Execute the MicroEJ build script (build.bat)** option is enabled in the Application launcher.

GET THE LINKER ERRORS

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

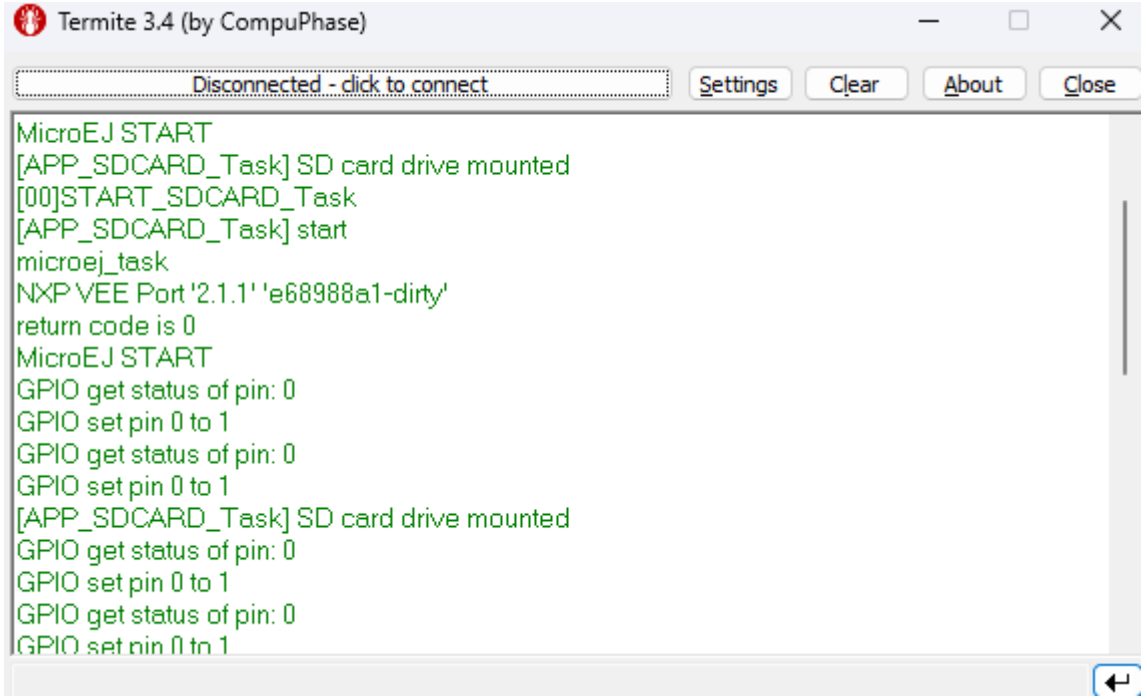
```
C:\XXX\com.microej.training.gpio.example.BlinkGpio\SOAR.o:(.text.soar+0x24dc): undefined reference to
`Java_com_microej_training_gpio_Gpio_get’

C:\XXX\com.microej.training.gpio.example.BlinkGpio\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.**
- Add a simple implementation of the 2 methods:

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

jboolean Java_com_microej_training_gpio_Gpio_get(jint pin){
         PRINTF("GPIO get status of pin: %d \n", pin);
         return 0;
}

void Java_com_microej_training_gpio_Gpio_set(jint pin, jboolean value){
          PRINTF("GPIO set pin %d to %d\n", pin, value);
}
```

RUN THE EXAMPLE ON DEVICE

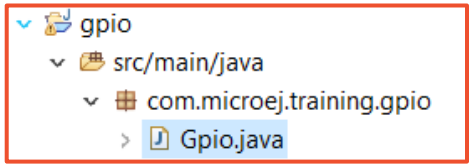
- Build the **gpio-example** project for the device:
  - Go to **Run -> Run Configurations.**
  - Select the **gpio-example BlinkGpio** Run Configuration.
  - Go to **Execution Tab.**
  - Select **Execute on Device.**
  - Click **Run.**
- Compile, Link and Flash with the 3rd party IDE.
- Open the Termite serial terminal to get execution traces.



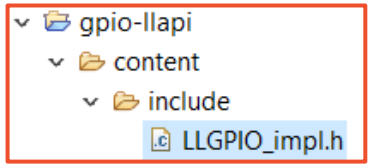
ABSTRACTION LAYER INTERFACE: LLAPI

The LLAPI project defines the natives to be implemented in the BSP project:

```
/**
 * GPIO management class.
 */
public class Gpio {
/**
 * Sets a value on the digital pin.
 *
 * @param pin
 *            the pin identifier
 * @param value
 *            digital pin value: true for high, false for low.
 */
native public static void set(int pin, boolean value);

/**
 * Gets the value of the digital pin.
 *
 * @param pin
 *            the pin identifier
 * @return true when the GPIO digital value is currently high, false
otherwise.
 */
native public static boolean get(int pin);
}
```




```
#define LLGPIO_set Java_com_microej_training_gpio_Gpio_set
#define LLGPIO_get Java_com_microej_training_gpio_Gpio_get
/**
 * Sets a value on the digital pin.
 *
 * @param pin
 *            the pin identifier
 * @param value
 *            digital pin value: JTRUE for high, JFALSE for low.
 */
void LLGPIO_set(int32_t pin, uint8_t state);

/**
 * Gets the value of the digital pin.
 *
 * @param pin
 *            the pin identifier
 * @return JTRUE when the GPIO digital value is currently high, JFALSE
otherwise.
 */
uint8_t LLGPIO_get(int32_t pin);
```



ABSTRACTION LAYER IMPLEMENTATION: LLIMPL

- The implementation is done in the BSP project.
- The **LLGPIO_get(int32_t pin)** and **LLGPIO_set(int32_t pin, uint8_t state)** functions are implemented as follow:

```
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 (pi
        GPIO_PinInit(GPIO9, 3U, &gpio9_pinM13_config);

        IOMUXC_SetPinMux(
          IOMUXC_GPIO_AD_04_GPIO9_IO03,           /* GPIO_
          0U);
    }
}
```

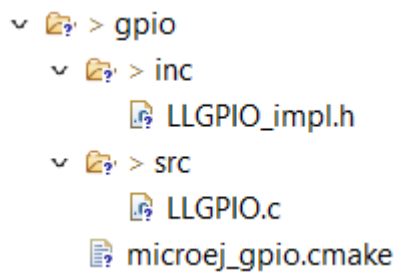
```
void LLGPIO_set(int32_t pin, uint8_t state)
{
    LLGPIO_initialize();

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

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

}
```

- The next slide describes how to add the sources code to the BSP project.

ADD SOURCES TO THE BSP PROJECT (1/2)

- Create the following folders in **nxpvee-mimxrt1170-evk-bsp/projects/microej:**
  - **gpio/**
    - **inc/**
    - **src/**
- Copy **LLGPIO_impl.h** to **gpio/inc/** (located in *[training-package]/gpio-foundation-library-example/gpio-llapi/content/include/LLGPIO_impl.h*)
- Copy **LLGPIO.c** to **gpio/src/** (located in *[training-package]/LLGPIO/NXP-i.MX_RT1170/LLGPIO.c*)
- Create a **microej_gpio.cmake** file in the **gpio/** folder
- Add the following content to **microej_gpio.cmake**:

```
include_guard()
message("microej/gpio component is included.")

target_sources(${MCUX_SDK_PROJECT_NAME} PRIVATE
    ${CMAKE_CURRENT_LIST_DIR}/src/LLGPIO.c
)

target_include_directories(${MCUX_SDK_PROJECT_NAME} PRIVATE    ${CMAKE_CURRENT_LIST_DIR}/inc)
```



gpio/ folder structure

ADD SOURCES TO THE BSP PROJECT (2/2)

Add the following lines in **nxpvee-mimxrt1170-evk-bsp/projects/nxpvee-ui/armgcc/CMakeLists.txt**:

- Add the **gpio/inc/** folder to include directories:

```
89 target_include_directories(${MCUX_SDK_PROJECT_NAME} PUBLIC
90     ${ProjDirPath}/../app
91     ${ProjDirPath}/../bsp
92     ${MicroEjRootDirPath}/
93     ${MicroEjRootDirPath}/core/inc
94     ${MicroEjRootDirPath}/cpuload/inc
95     ${MicroEjRootDirPath}/fs/inc
96     ${MicroEjRootDirPath}/gpio/inc
```

- Add the **gpio/** folder to CMake module path:

```
116 set(CMAKE_MODULE_PATH
117     ${MicroEjRootDirPath}/core
118     ${MicroEjRootDirPath}/cpuload
119     ${MicroEjRootDirPath}/fs
120     ${MicroEjRootDirPath}/gpio
121     ${MicroEjRootDirPath}/kf
122     ${MicroEjRootDirPath}/net
```

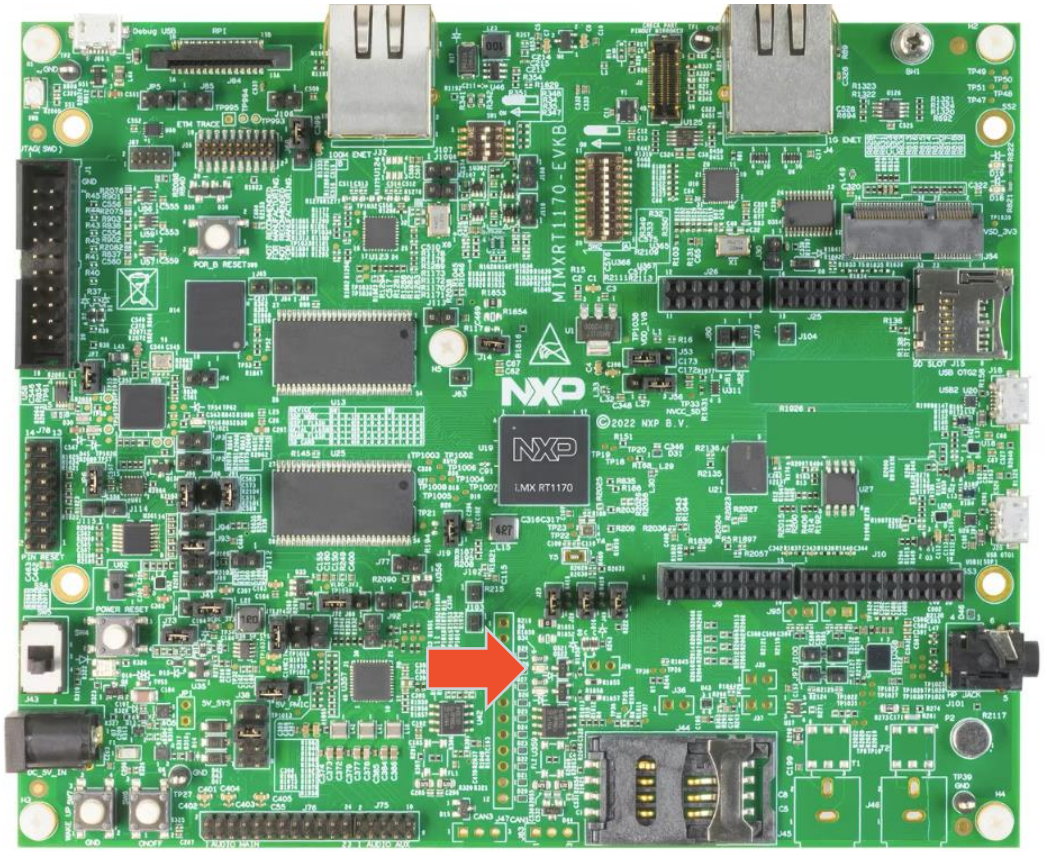
- Under the **# include modules**, add:

```
248 include(microej_core)
249 include(microej_cpuload)
250 include(microej_fs)
251 include(microej_gpio)
252 include(microej_kf)
253 include(microej_net)
254 include(microej_security)
255 include(microej_ssl)
```

RUN THE EXAMPLE ON DEVICE

- Build the **gpio-example** project for the device:
  - Go to **Run -> Run Configurations.**
  - Select the **gpio-example BlinkGpio** Run Configuration.
  - Go to **Execution Tab.**
  - Select **Execute on Device.**
  - Click **Run.**
- Compile, Link and Flash the Firmware.

The GREEN LED D6 blinks on the board.



Packaging and Tests

PACKAGING AND TESTS

The following actions will be performed when building the GPIO library:

- Generate a JAR file with the classfiles.
- Generate a zip file with the sources.
- Generate the Javadoc.
- Execute the tests (defined in src/test/java folder).
- Publish the library in an MMM repository.

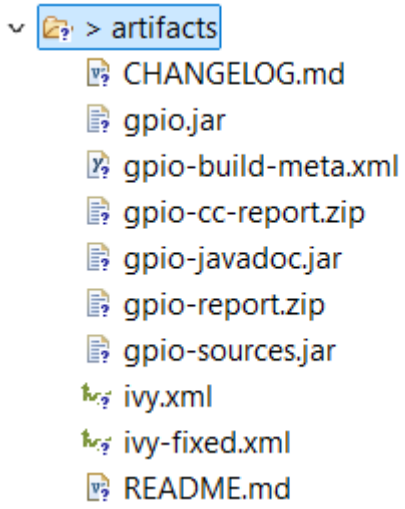
ADD VEE PORT TO TEST THE LIBRARY

A VEE Port must be referenced to run the tests of the library:

1. Right-Click on the **source/** folder of the VEE Port project.
2. Go to **Properties**.
3. Copy the location path.
4. Open the file **module.ivy** of the **gpio** project.
5. Uncomment the definition of the property **platform-loader.target.platform.dir.**
6. Paste the path previously copied.

LAUNCH THE LIBRARY BUILD

- Right-Click on the **gpio** project and select **Build Module.**
- Build result is available in the folder **target~/artifacts**:

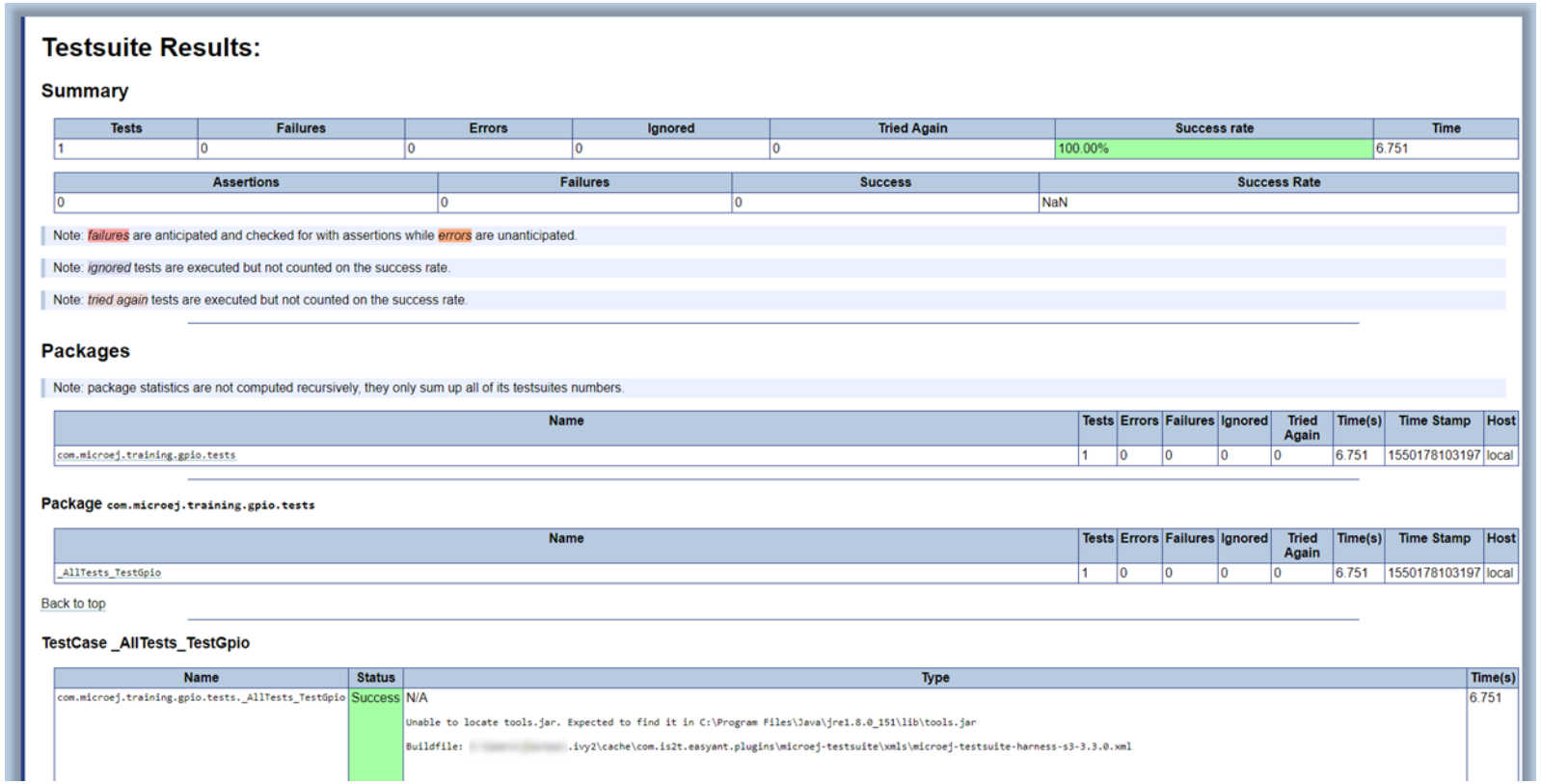


- Build result is published in a local MMM repository:

~\.ivy2\repository\com\microej\training\gpio

TESTS RESULT

A testsuite report is available in the **target~/artifacts/gpio-report-${version}.zip** file or in **target~\test\html\test:**



Testsuite Results:

Summary

Tests	Failures	Errors	Ignored	Tried Again	Success rate	Time
1	0	0	0	0	100.00%	6.751

Assertions	Failures	Success	Success Rate
0	0	0	NaN

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.training.gpio.tests	1	0	0	0	0	6.751	1550178103197	local

Package com.microej.training.gpio.tests

Name	Tests	Errors	Failures	Ignored	Tried Again	Time(s)	Time Stamp	Host
_AllTests_TestGpio	1	0	0	0	0	6.751	1550178103197	local

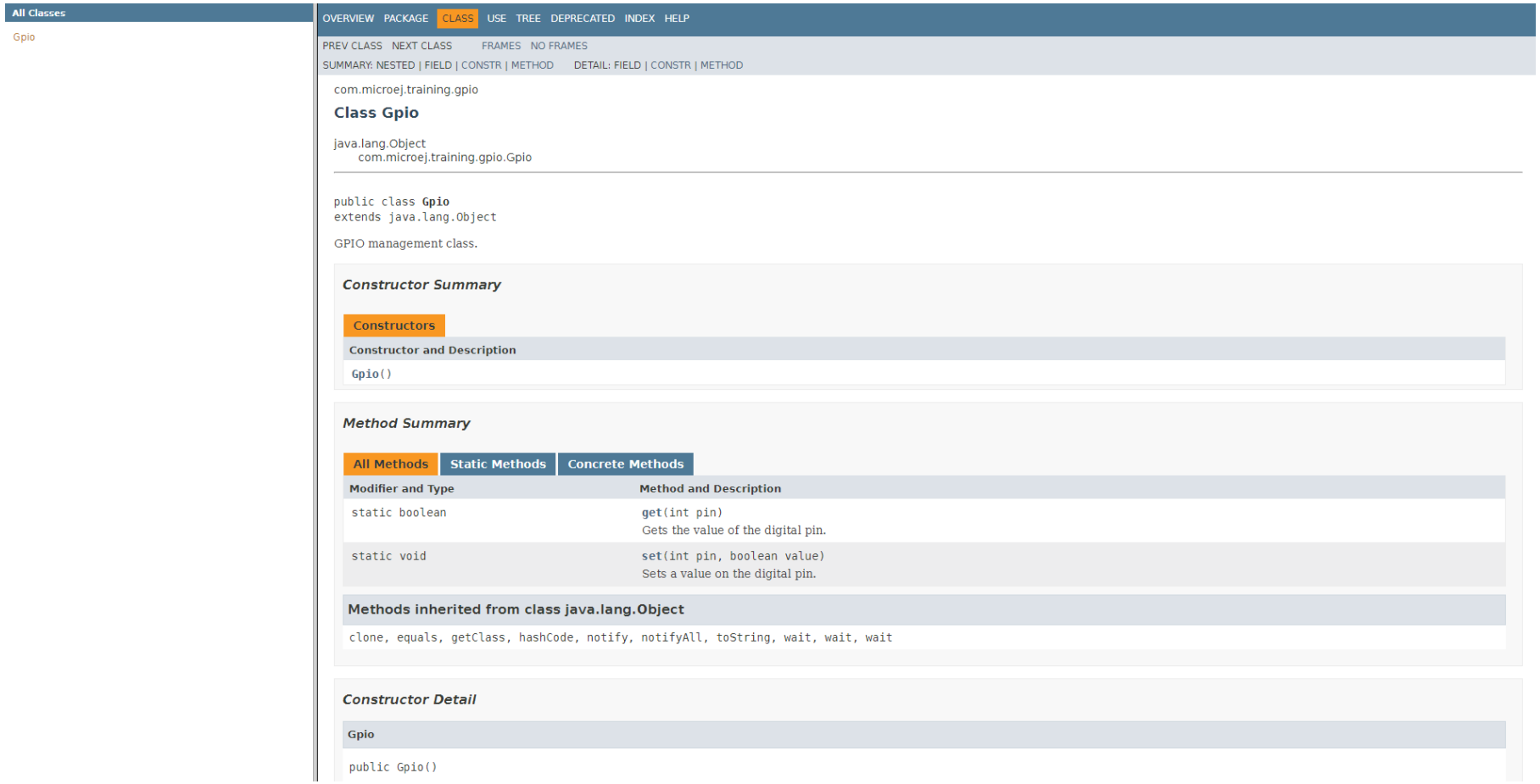
Back to top

TestCase _AllTests_TestGpio

Name	Status	Type	Time(s)
com.microej.training.gpio.tests._AllTests_TestGpio	Success	N/A Unable to locate tools.jar. Expected to find it in C:\Program Files\Java\jre1.8.0_151\lib\tools.jar Buildfile: [redacted].ivy2\cache\com.is2t.easyant.plugins\microej-testsuite\xmls\microej-testsuite-harness-s3-3.3.0.xml	6.751

JAVADOC

Javadoc is available in **target~/javadoc** folder:



All Classes

Gpio

OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

PREV CLASS NEXT CLASS FRAMES NO FRAMES

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

com.microej.training.gpio

Class Gpio

java.lang.Object
com.microej.training.gpio.Gpio

```
public class Gpio
extends java.lang.Object
```

GPIO management class.

Constructor Summary

Constructors

Constructor and Description
`Gpio()`

Method Summary

All Methods **Static Methods** **Concrete Methods**

Modifier and Type	Method and Description
`static boolean`	`get(int pin)` Gets the value of the digital pin.
`static void`	`set(int pin, boolean value)` Sets a value on the digital pin.

Methods inherited from class java.lang.Object

`clone, equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

Gpio

```
public Gpio()
```

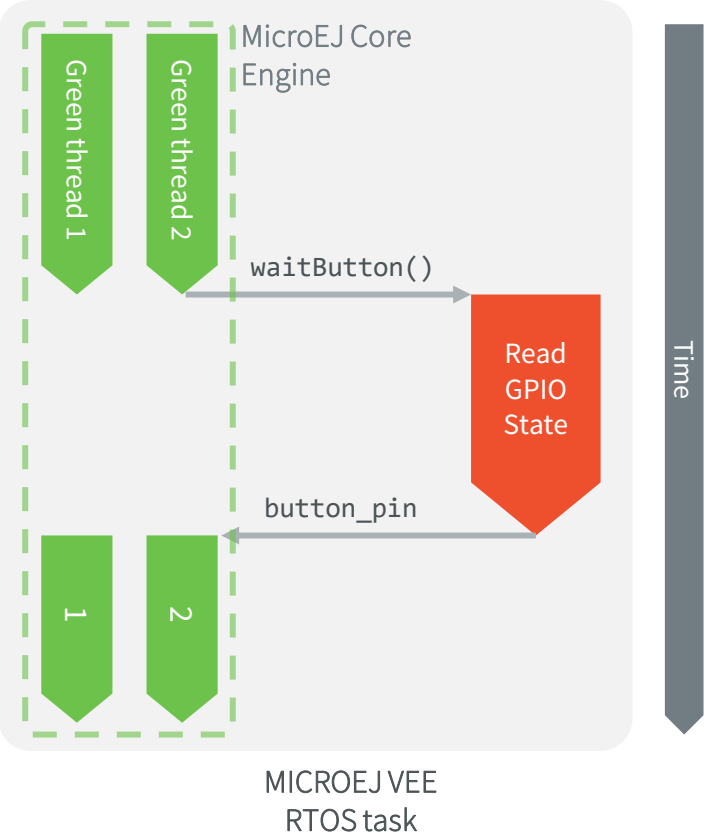
SNI

Manage Multithreading

GREEN THREAD ARCHITECTURE

- Green threads are threads that are scheduled by the virtual machine instead of natively by the underlying operating system.
- Green threads emulate multithreaded environments without relying on any native OS abilities, enabling them to work in environments that do not have native thread support.

THREAD SYNCHRONIZATION: BLOCKING CASE



- While a native method is executed, other Java threads can't be scheduled.
  - SNI functions stop the Java world.
- Usually, the actions are asynchronous on the BSP side and the result takes times to be returned (e.g., IP/USB/Bluetooth stacks).
- Goal: Execute a native in another task and wait for the result.

GPIO EXERCISE OVERVIEW

- The code of the **gpio-exercise** project does the following actions:
  - Wait for a button event and prints the index of the pressed button (User/Blue button)
  - Toggles the board LED1 each 500ms
  - Each action in performed in a dedicated thread

```
public class GpioExercise {

private static final int PIN = 0;
private static final long DELAY = 500;

public static void main(String[] args) throws InterruptedException
{

	 // This thread waits for button actions.
	Thread t = new Thread(new Runnable() {

	@Override
	public void run() {
	while (true) {
		System.out.println("Waiting for a button event...");
		int action = waitButton();
		System.out.println("Button pressed! Action ID=0x" +
		Integer.toHexString(action));
	}
	}
	});
	t.start();

	// The main thread loops indefinitely and blinks the LED.
	while (true) {
		Gpio.set(PIN, !Gpio.get(PIN));
		Thread.sleep(DELAY);
	}

}

public static native int waitButton();
}
```

Run the GPIO Exercise code

SETUP

- A C implementation is provided in the training package (**[training-package]/LLGPIO/NXP-i.MX_RT1170/LLGPIO_exercise_freertos.c**).

- Add **LLGPIO_exercise_freertos.c** to the BSP project:
  - Copy / Paste **LLGPIO_exercise_freertos.c** in the **nxpvee-mimxrt1170-evk-bsp\projects\microej\gpio\src** folder.
  - Edit **gpio/microej_gpio.cmake** to use **LLGPIO_exercise_freertos.c** instead of **LLGPIO.c**

RUN THE EXERCISE CODE (1/2)

- Compile the application in MICROEJ SDK:
  - Right click on the **GpioExercise.java** class of the **gpio-exercise** project
  - **Run as → Run Configurations..**
  - Double click on MICROEJ Application
  - Go to **Execution** tab
  - Select the VEE Port
  - Select **Execute on Device**
  - Click **Run**
- Compile, Link and Flash the Firmware.

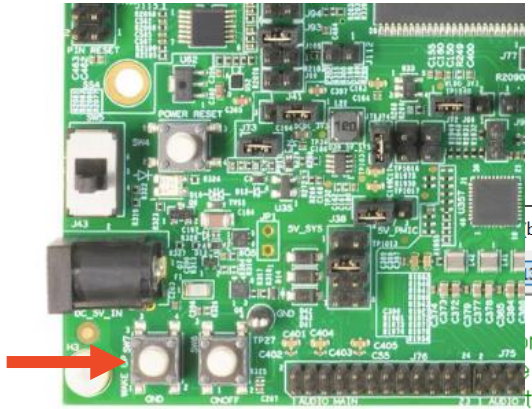
Note: make sure that the **Execute the MicroEJ build script (build.bat)** option is enabled in the Application launcher.

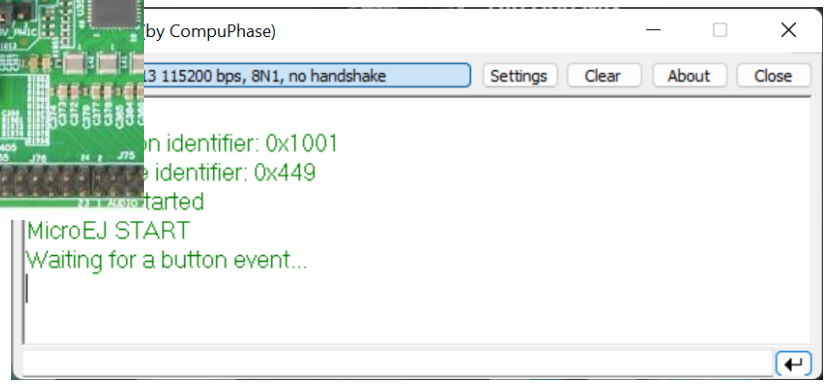
RUN THE EXERCISE CODE (2/2)

- Open the Termite serial terminal.
- Reset the NXP i.MX RT1170 EVK board using Reset button
- The application starts and waits for a button event.

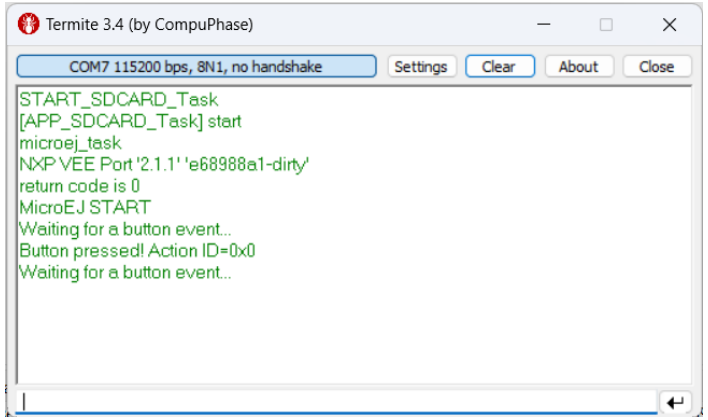
- **LED1 is not blinking each 500ms as expected. The waitButton() native blocks the execution of the other Java threads.**

- When pressing the button once:
  - The ID of the button event is printed in the console
  - The LED turns on
- When pressing again, the ID of the button event is printed and the LED turns off





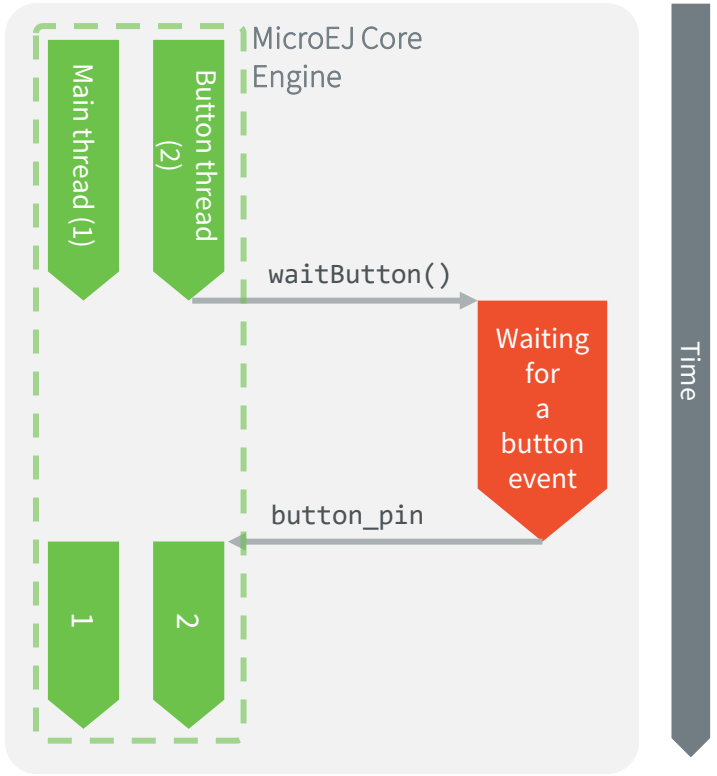
Traces when the application starts



Traces after 1 button press

GPIO EXERCISE: BLOCKING BEHAVIOR

- In this example, the execution of the **waitButton()** native method will block until the button is pressed.
  - In other words, while **Java_com_microej_training_gpio_example_GpioExercise_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:



MICROEJ VEE RTOS task

Hand's On

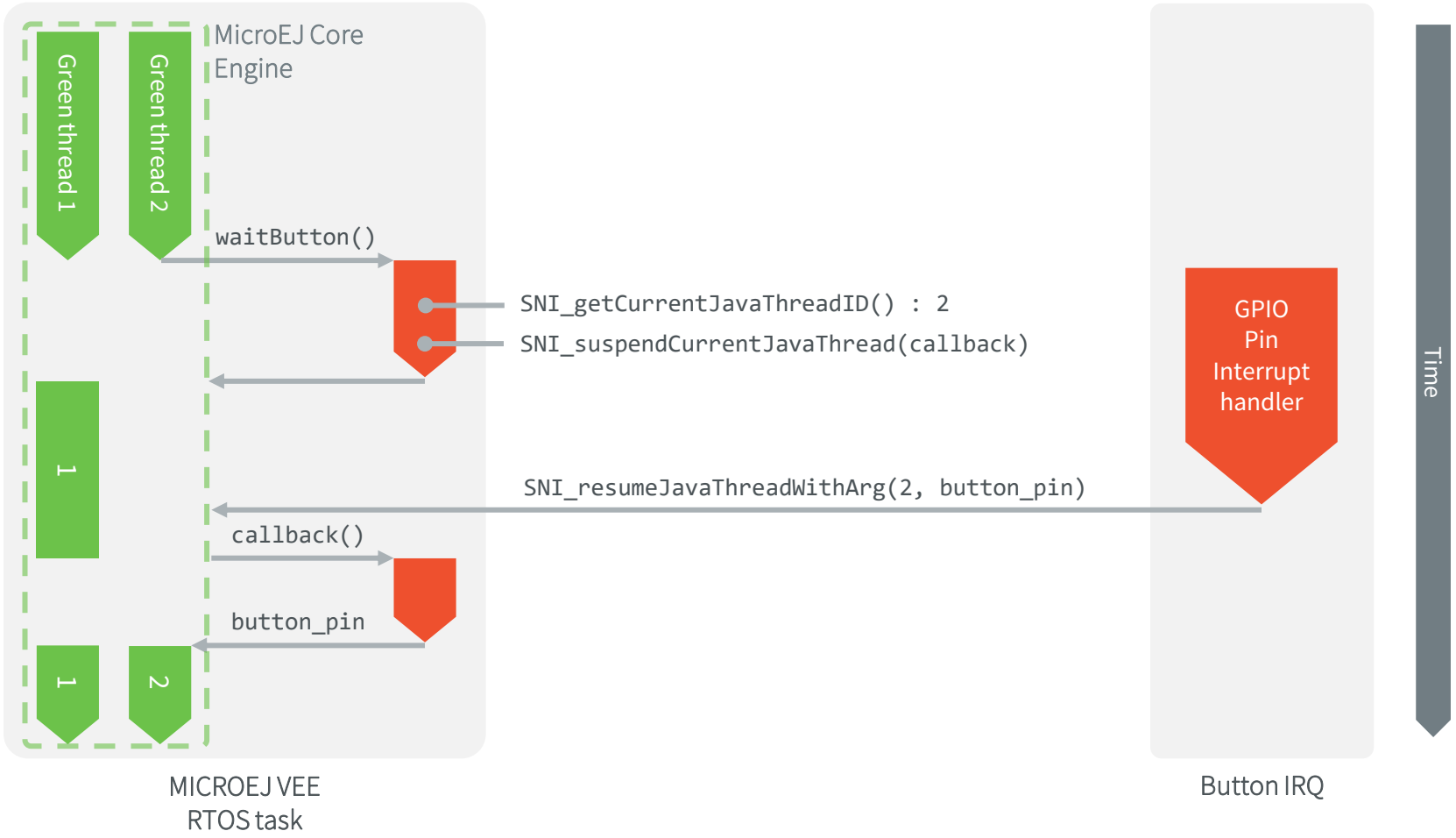
Implement a blocking Java native method without blocking the execution of other Java threads.

HAND'S ON DIRECTIVES

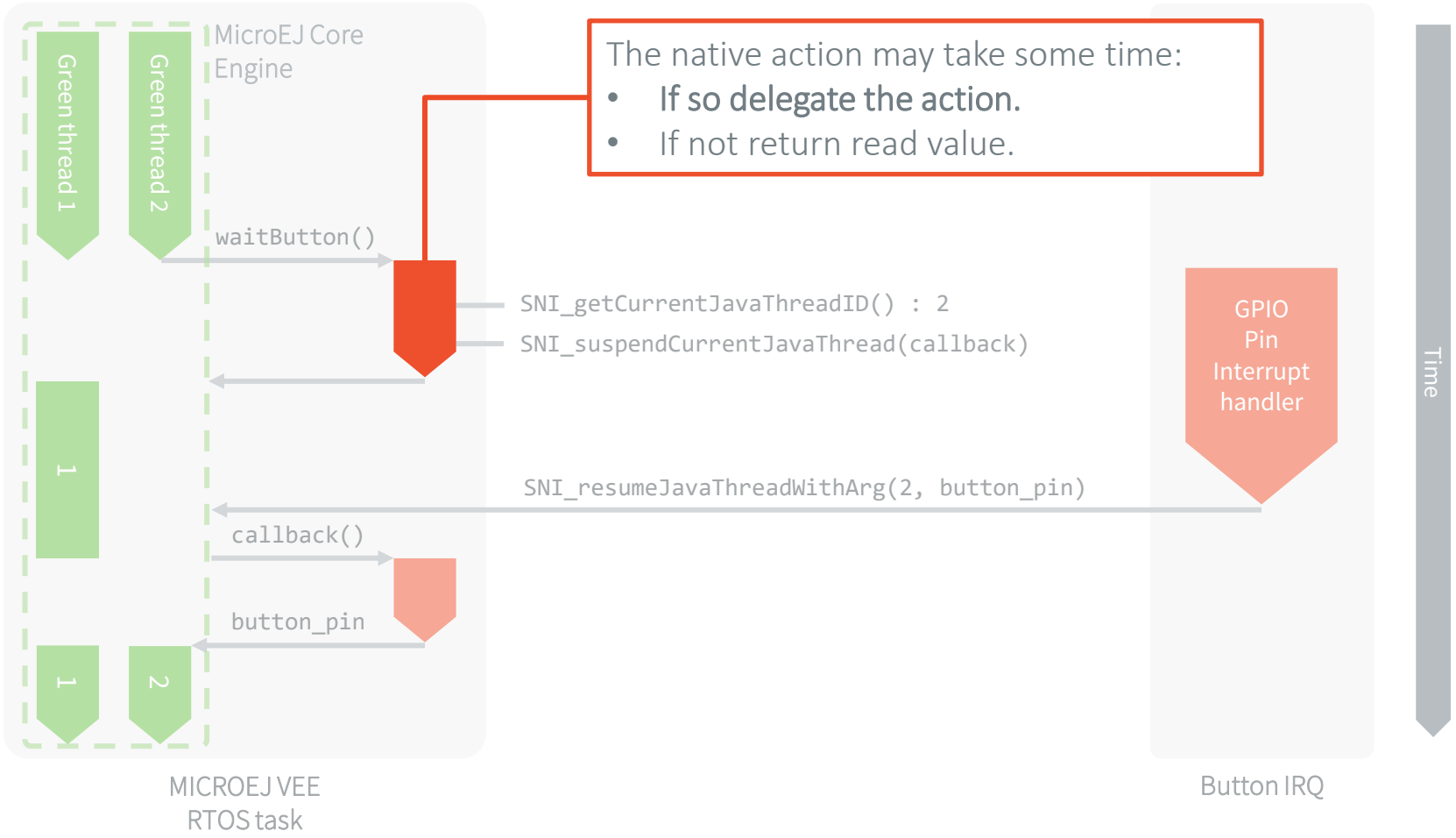
- Only the C code should be updated

- Here is a summary of what should 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.
- Tips:
  - Use the SNI functions defined in **sni.h**
  - SNI documentation: https://docs.microej.com/en/latest/PlatformDeveloperGuide/sni.html#sni

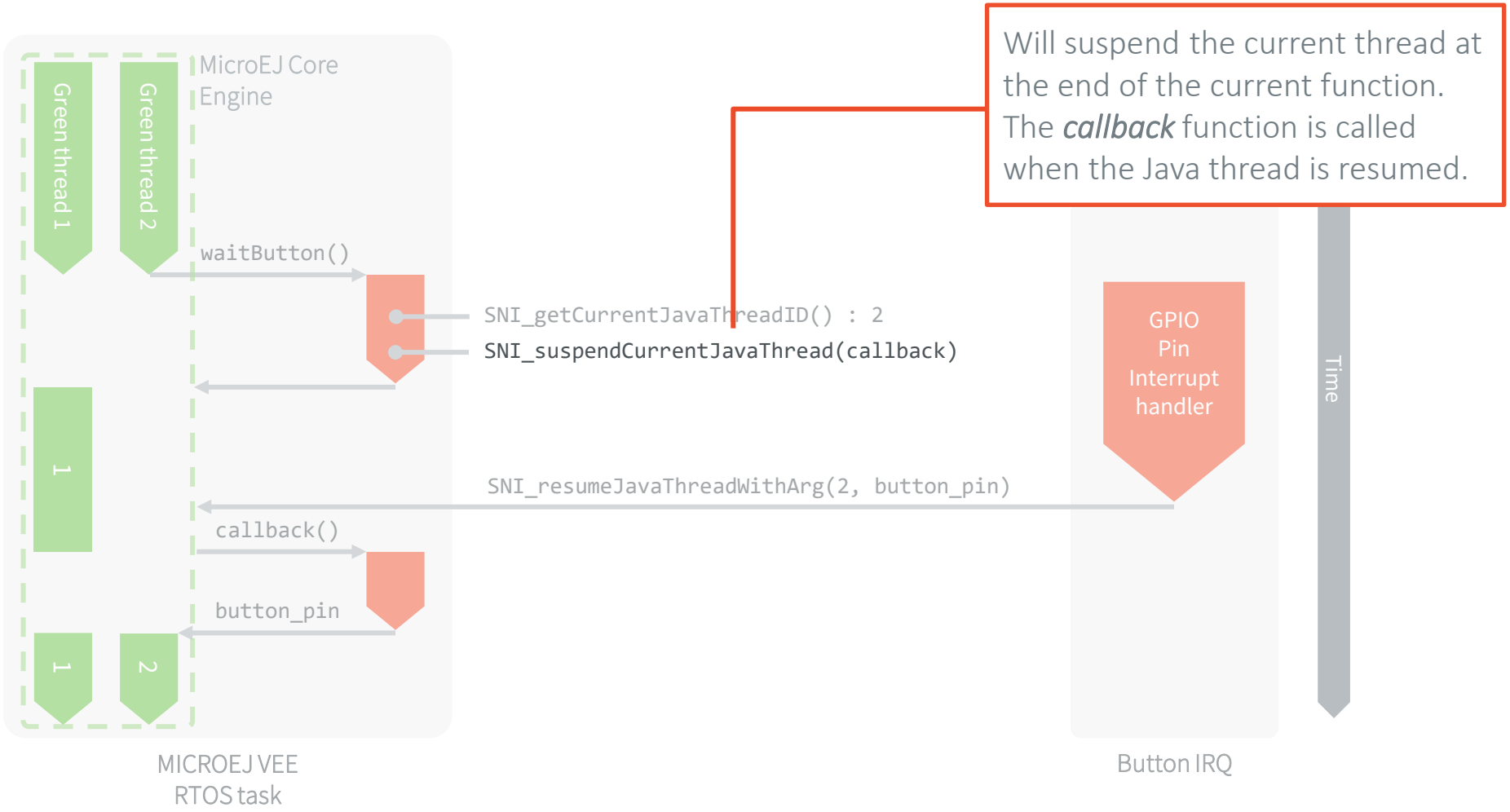
THREAD SYNCHRONIZATION: CALLBACK PATTERN



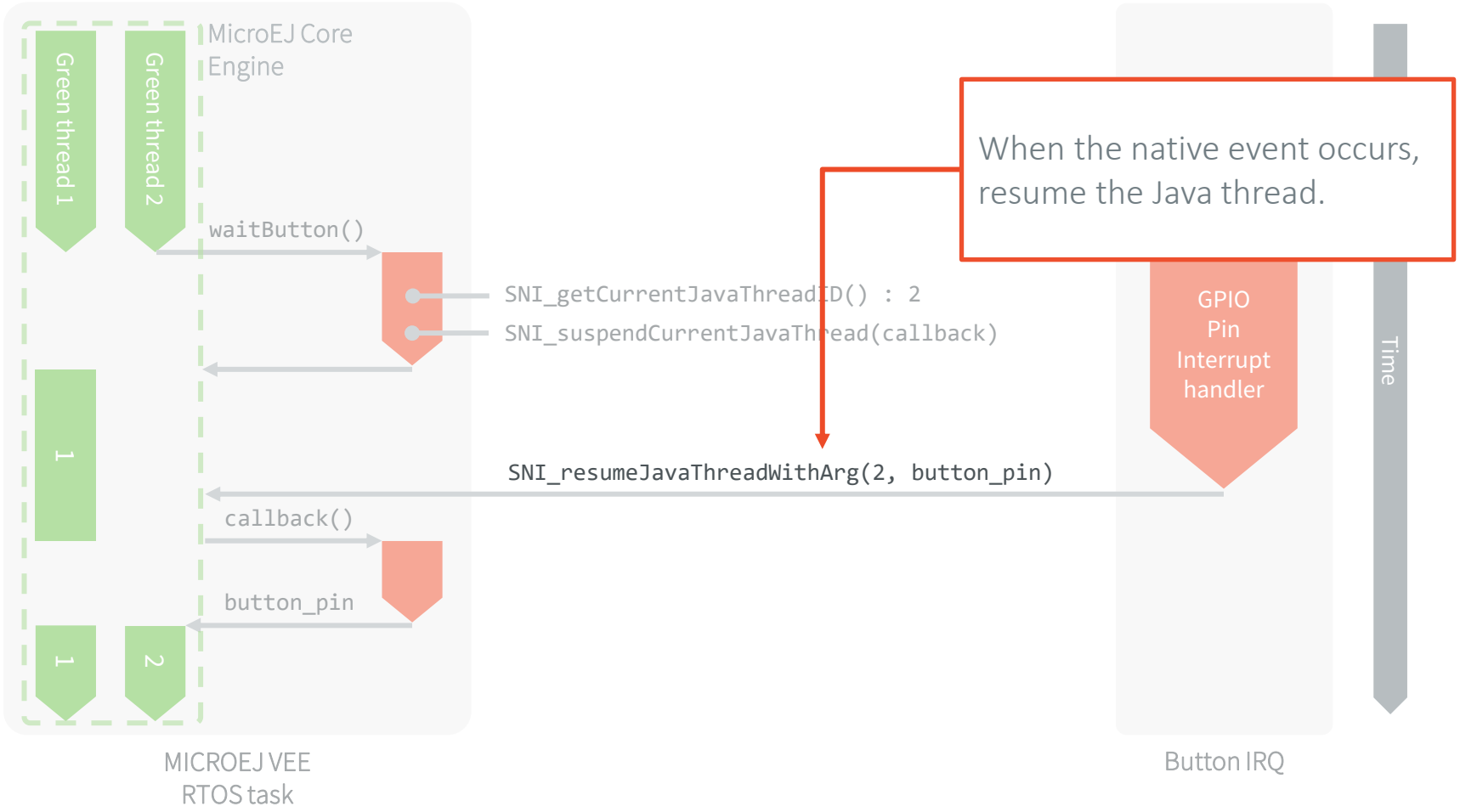
THREAD SYNCHRONIZATION: CALLBACK PATTERN



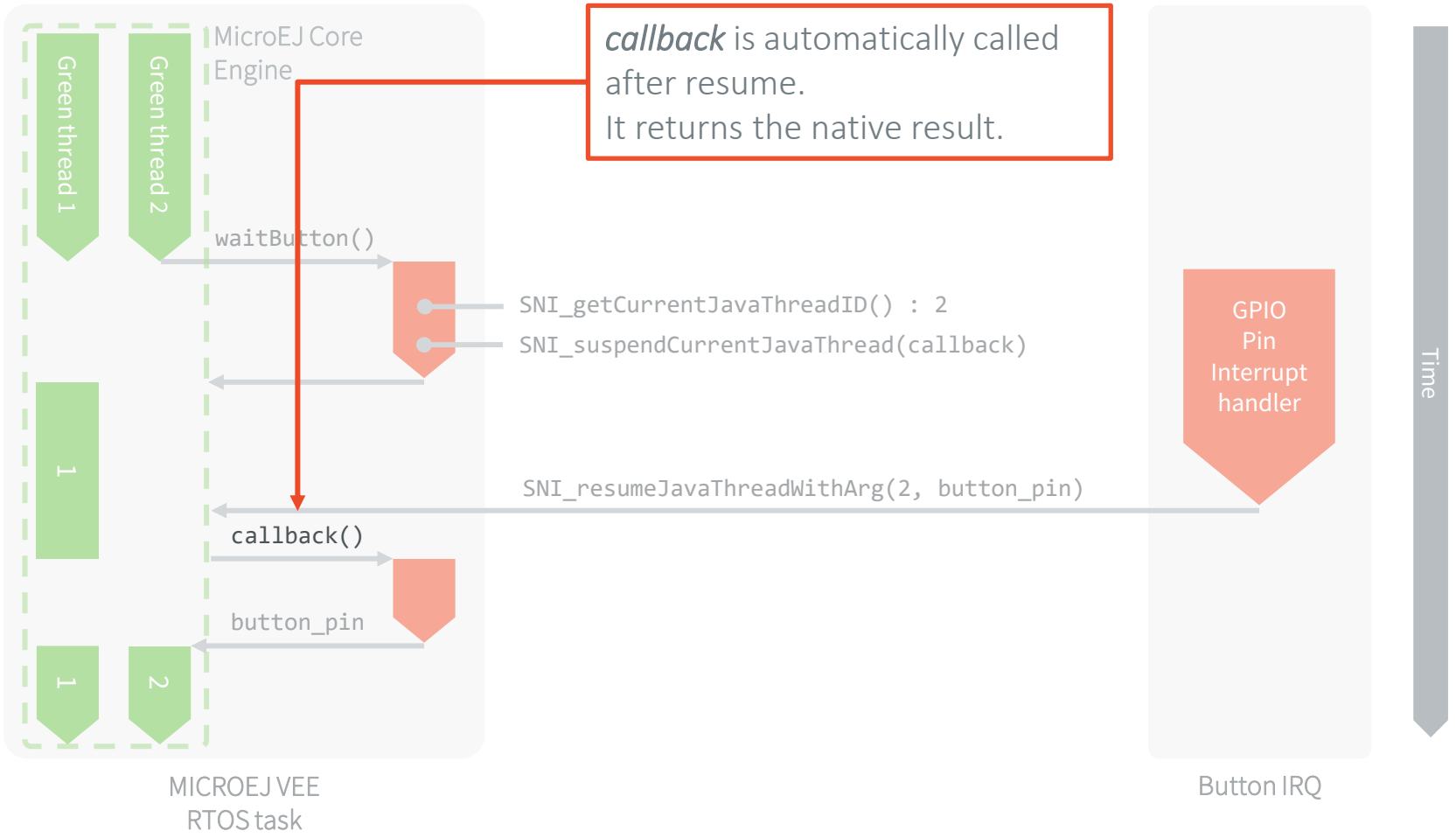
THREAD SYNCHRONIZATION: CALLBACK PATTERN



THREAD SYNCHRONIZATION: CALLBACK PATTERN



THREAD SYNCHRONIZATION: CALLBACK PATTERN



STEP 1: UPDATE THE C NATIVE FUNCTION

- The **Java_com_microej_training_gpio_example_GpioExercise_waitButton()** function will now suspend the current Java thread.
  It will also store the information required to resume it and register the callback function.
- The function **SNI_suspendCurrentJavaThreadWithCallback()** returns immediately. The current thread is actually suspended when the native function returns.
- The value returned by the **Java_com_microej_training_gpio_example_GpioExercise_waitButton()** doesn't matter anymore. The callback function will be in charge of returning the value.

```
static int32_t java_thread_id;

jint Java_com_microej_training_gpio_example_GpioExercise_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:

```
static volatile jint button_index;

/** Interrupt request handler called when the button is pressed. */
void BOARD_ButtonHandler(void* arg)
{
    button_index = (int32_t)EXAMPLE_BUTTON_GPIO_PIN;
    SNI_resumeJavaThreadWithArg(java_thread_id, (void*)&button_index); // save the button_index pointer
in the VM
}
```

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.

```
static jint waitButton_callback(){
  int32_t * button_index_addr; // will contain the pointer to button_index
  SNI_getCallbackArgs(NULL, (void**)&button_index_addr);
  return (jint)*button_index_addr; // The returned value to Java is the button_index value
}
```

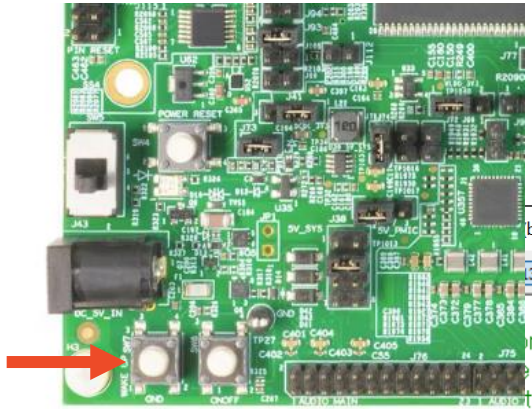
RUN THE UPDATED CODE

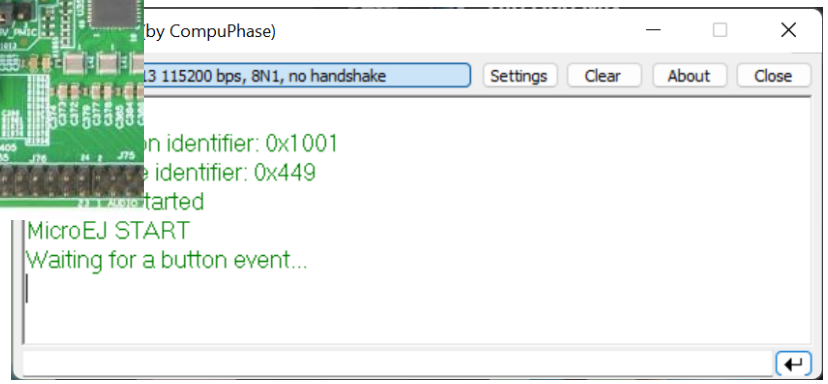
- Open the Termite serial terminal.
- Reset the NXP i.MX RT1170 EVK board using Reset button
- The application starts and waits for a button event.

- **LED1 is now blinking each 500ms.**

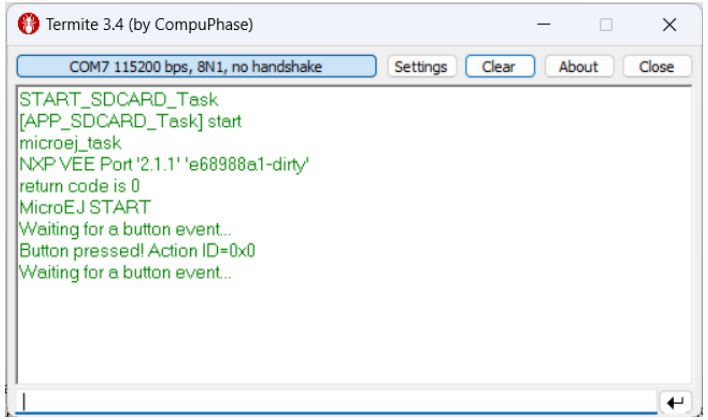
- When pressing the button once:
  - The ID of the button event is printed in the console

- When pressing again, the ID of the button event is printed and the LED turns off





Traces when the application starts



Traces after 1 button press

Resources

Online Resources

- 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)

Main Resources

- https://docs.microej.com/en/latest/ApplicationDeveloperGuide/index.html : Describes MICROEJ usage for end developers
- https://docs.microej.com/en/latest/PlatformDeveloperGuide/index.html: Describes how to interact with the platform and integrate MICROEJ to a board
- https://github.com/MICROEJ/Example-Standalone-Foundation-Libraries: Snippets of code for foundation libraries (EDC, BON, Net, MicroUI…)
- https://github.com/MICROEJ/ExampleJava-Widget: Source code for using the widget library

Shortcuts

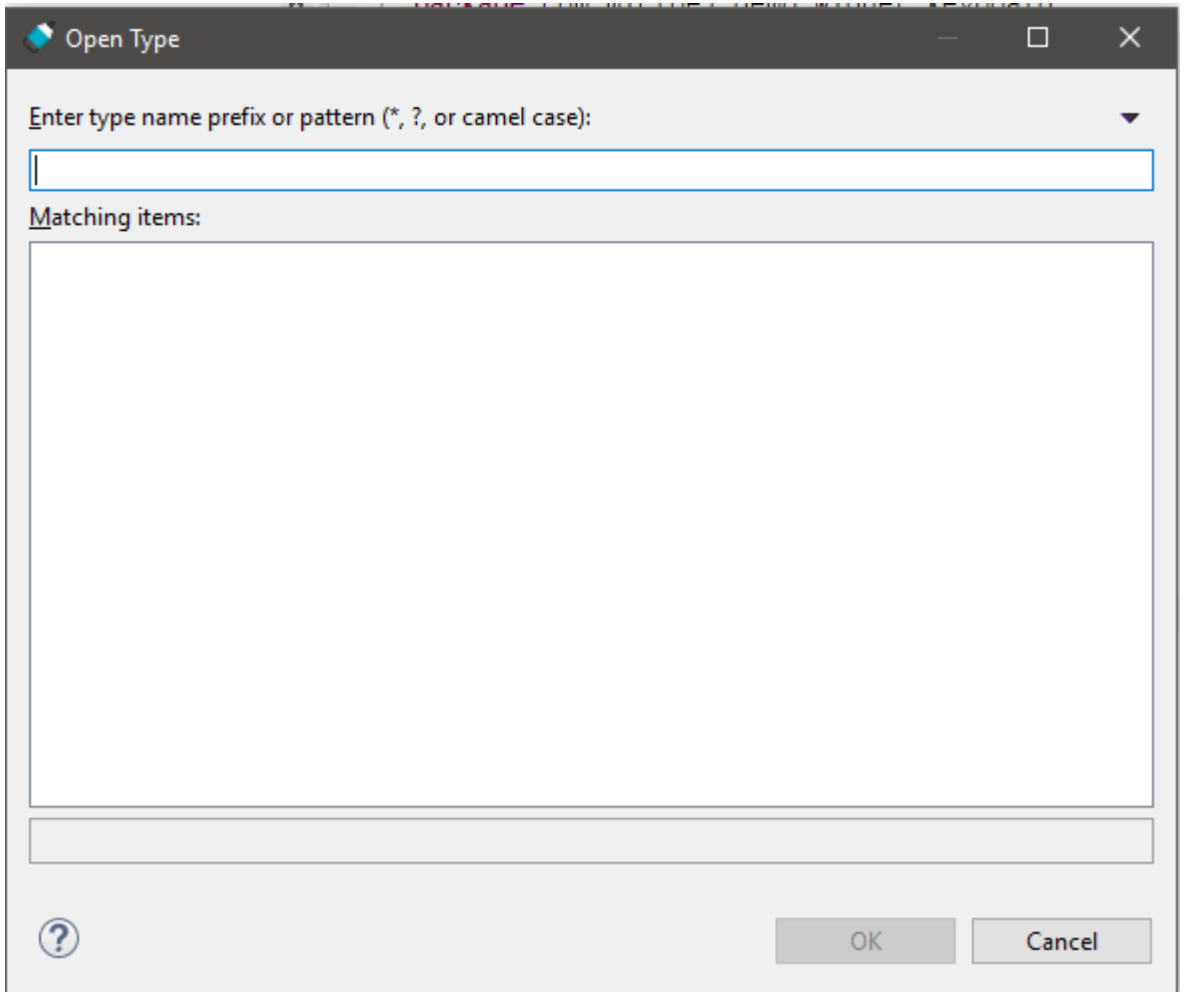
MICROEJ SDK / Studio

SHORTCUTS

- **CTRL + Space**
  - Auto completion
    - Probably the most useful one
- **CTRL + D**
  - Delete row
- **ALT + Up/Down Arrow**
  - Move the row (or the entire selection) up or down. Very useful when rearranging code
- **CTRL+SHIFT+O**
  - Organize imports.

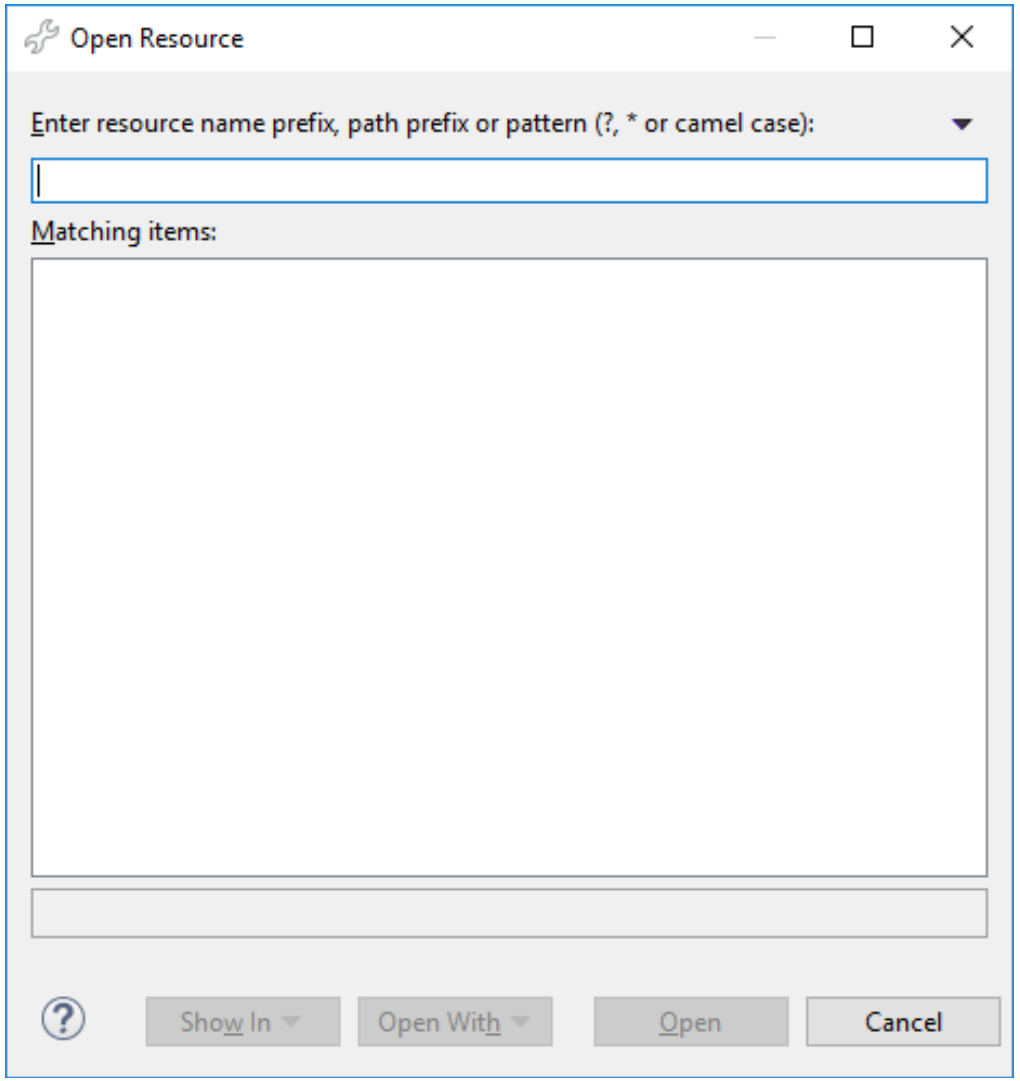
SHORTCUTS

- **CTRL+SHIFT+T**
  - Open Type.



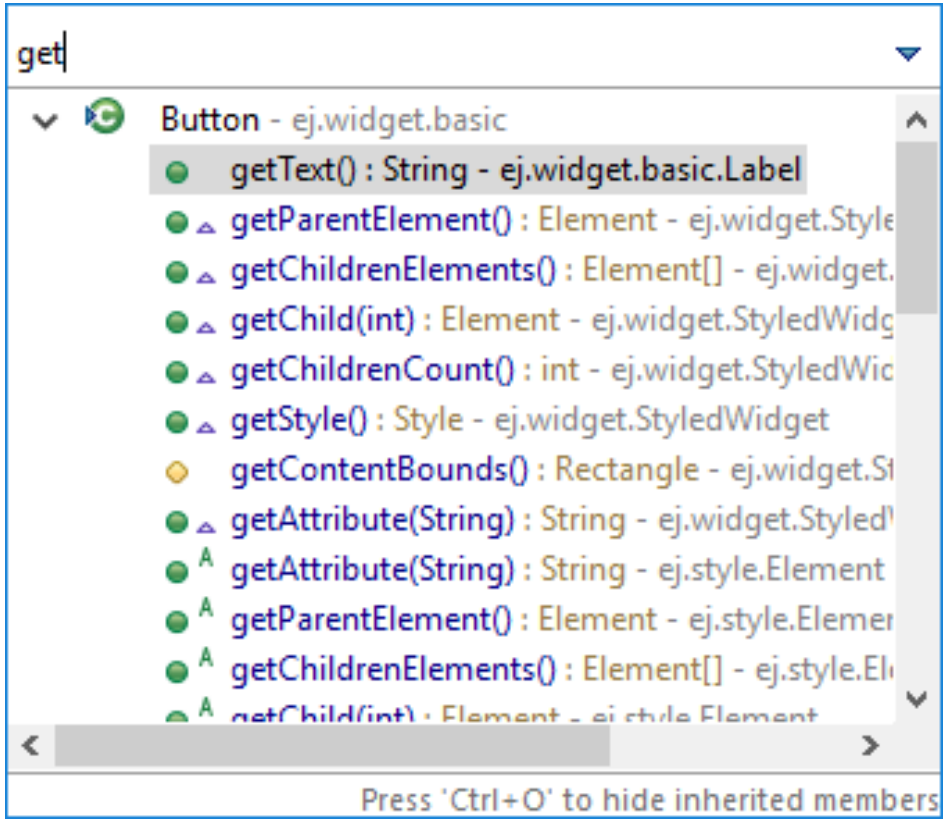
SHORTCUTS

- **CTRL+SHIFT+R**
  - Open Resource (any file)



SHORTCUTS

- **CTRL + O**
  - Open Outline (find method or field)
  - Press CTRL + O again to show methods from superclasses



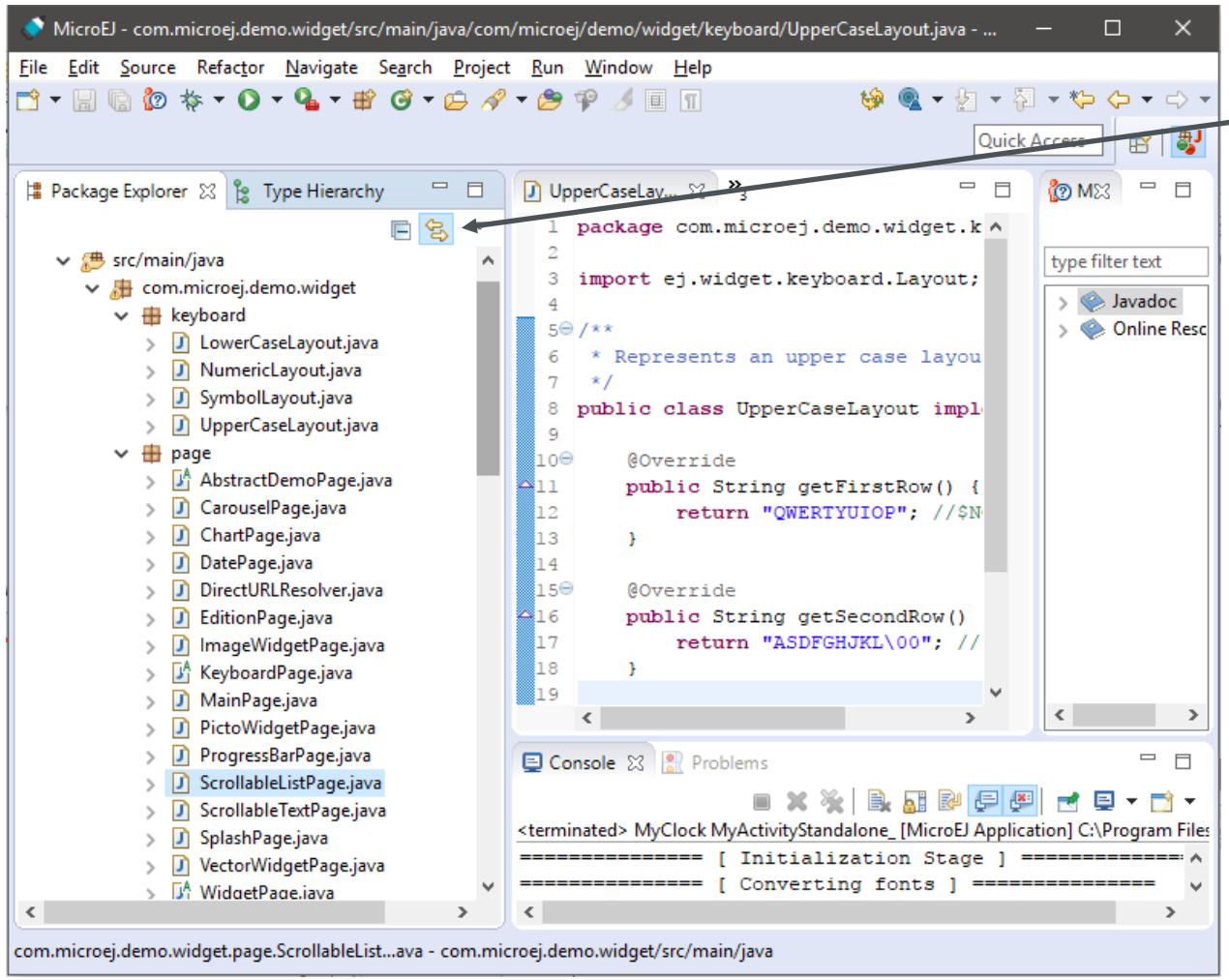
SHORTCUTS

- **F2**
  - Display the Javadoc
- **Hold CTRL + Click on class**
  - Go through the definition of class
- **CTRL + T**
  - On a method: display implementations of the method in subclasses or definitions in superclasses
  - On a class: display class hierarchy (superclasses and subclasses)
- **CRTL + 1**
  - Extract variable to
    - Local variable
    - Constant

SHORTCUTS

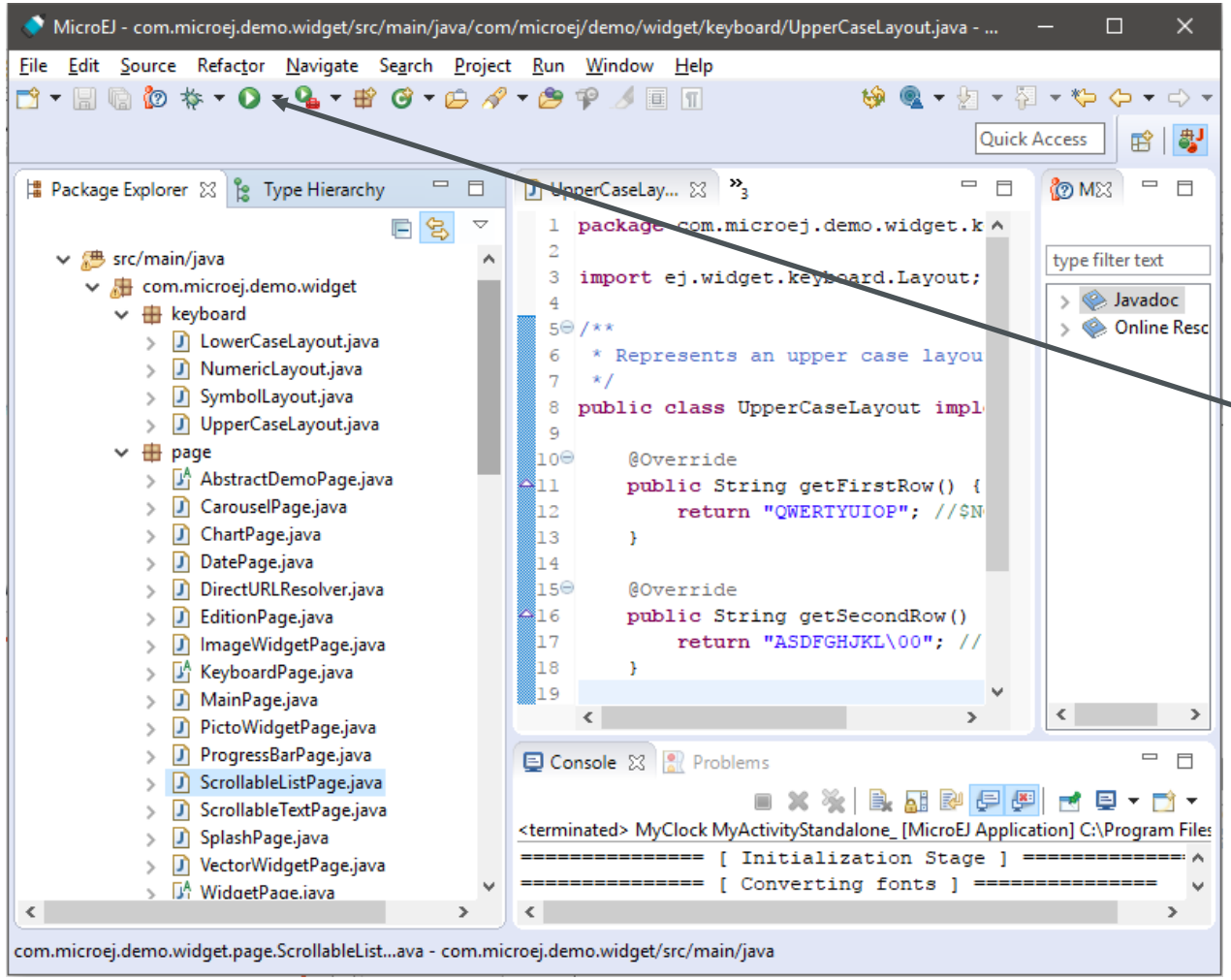
- **CTRL + I**
  - Correct indentation
- **ALT + Shift + R on a class / method / field**
  - Rename
- **CTRL + F**
  - Search in file
- **CTRL + H**
  - Search plugin of Eclipse

IDE



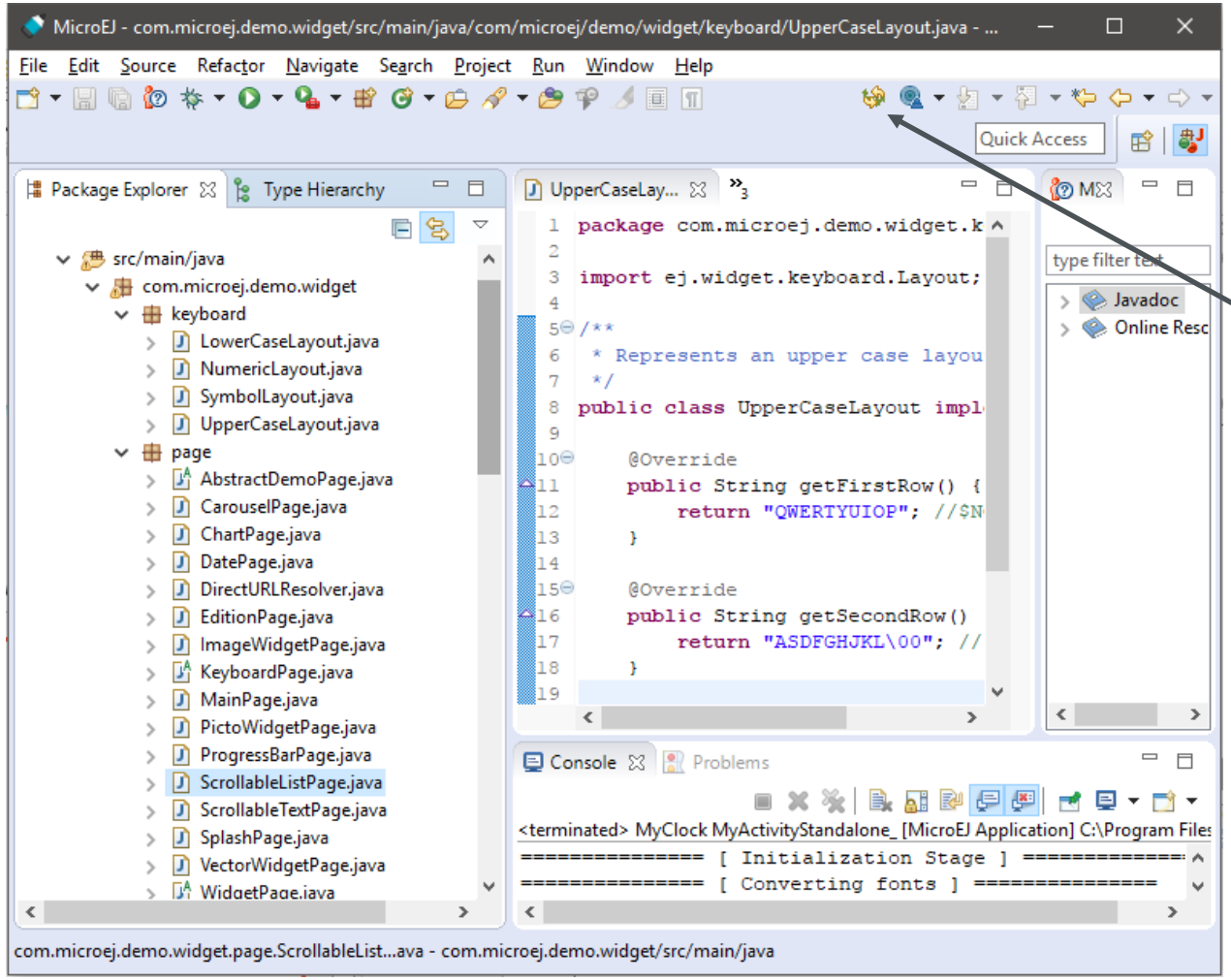
Link the explorer and the editor

IDE



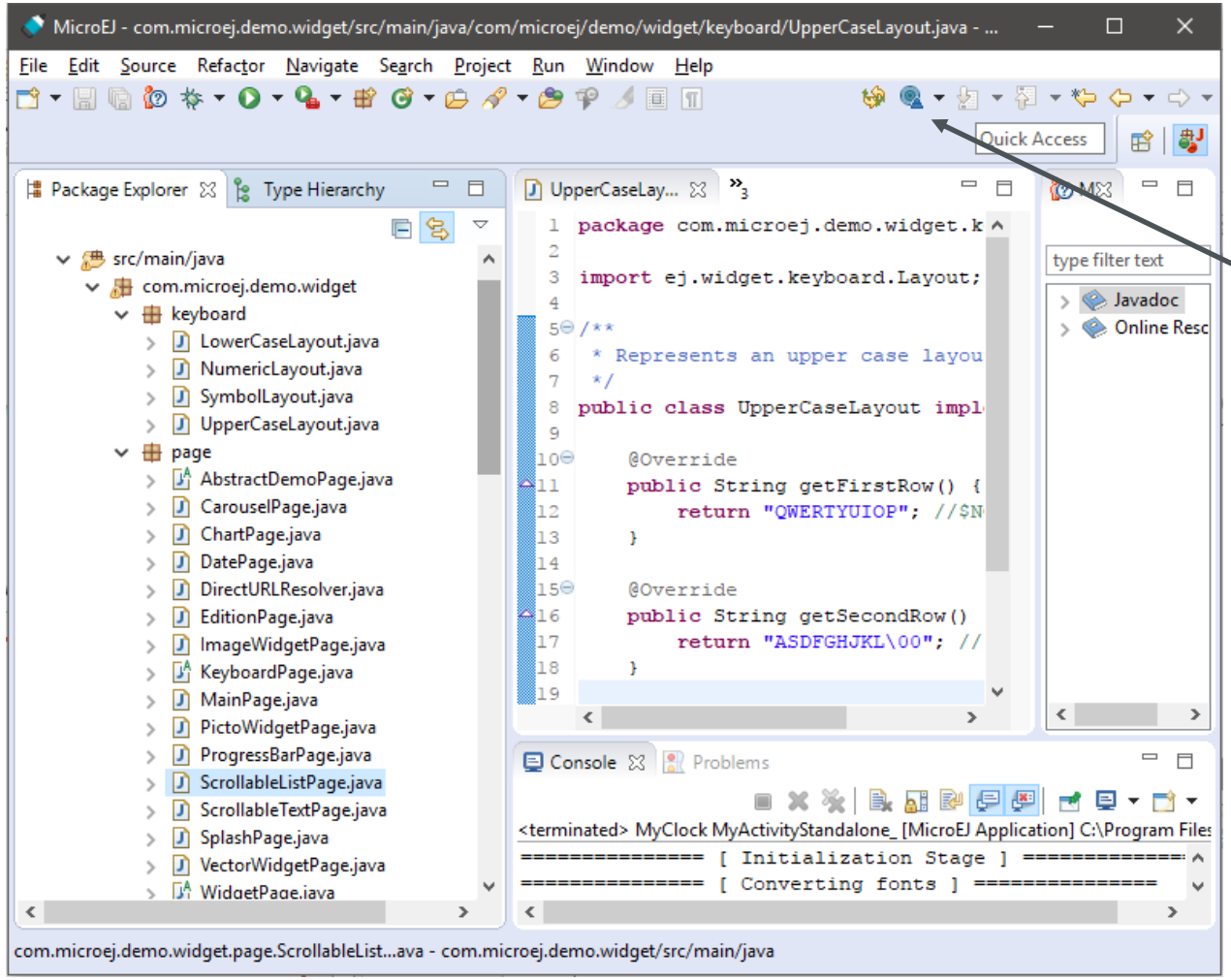
Re-Run the last
Run configuration

IDE



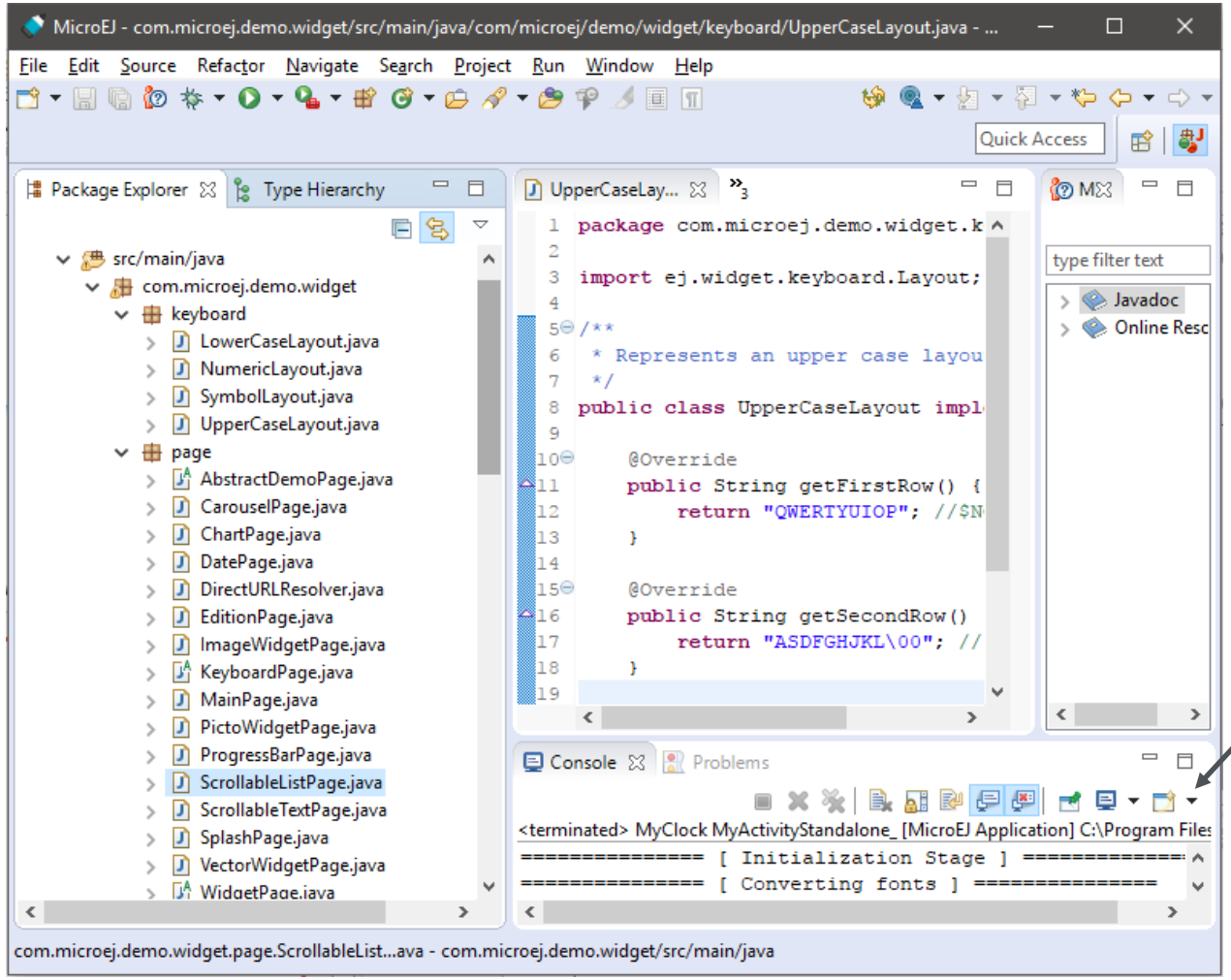
Resolve all MMM projects

IDE



Build selected module

IDE



Change console view
- Ivy console
- Addon Processor
- C/C++ Build Console

THANK YOU

for your attention !

MICROEJ®