MICROEJ SDK 5 Basics

For STM32F7508-DK

MICROEJ®

DISCLAIMER

AGENDA

WHAT YOU WILL LEARN

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

- Build a MICROEJ VEE Port.
- Build and Run a Java Application.
- Edit a Front Panel File.
- Create your own Foundation Libraries.
- Call a C function from Java.

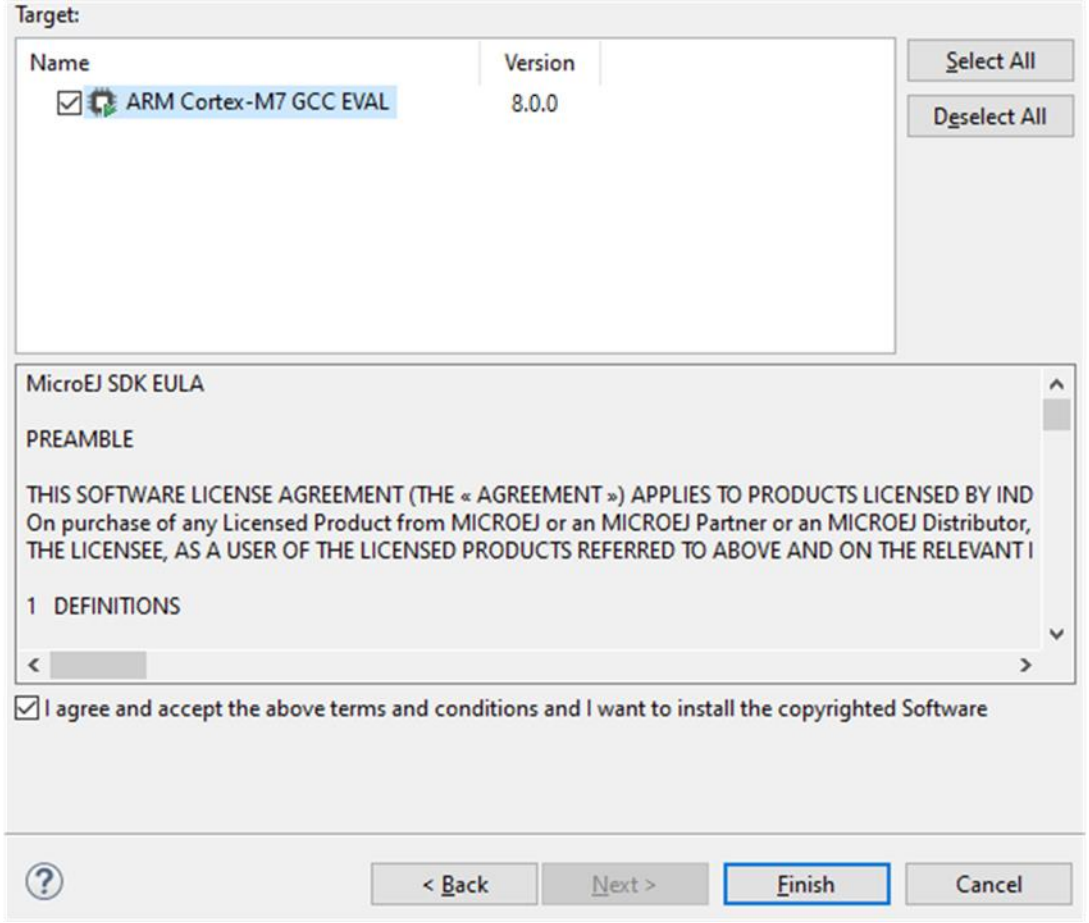
REQUIREMENTS

- STM32F7508-DK Board and a mini-USB cable.
    - Clone the STM32F7508 2.1.2 VEE Port repository using the Git --recursive option to get the submodules.
- Download the **flopi7G26-8.0.0-eval.xpf** MICROEJ Architecture for Cortex M7 GCC (https://repository.microej.com/modules/com/microej/architecture/CM7/CM7hardfp_GCC48/flopi7G26/8.0.0/).
- 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 MICROEJ 23.07 SDK (https://repository.microej.com/packages/SDK/23.07/MicroEJ-SDK-Installer-Win64-23.07.exe).
    - Install a serial terminal (https://www.compuphase.com/software_termite.htm).
    - Install **STM32CubeIDE 1.9.0** (https://www.st.com/en/development-tools/stm32cubeide.html).
    - Access to internet and the MICROEJ Central Repository (https://repository.microej.com/).

INSTALLING MICROEJ ARCHITECTURE

IMPORTING ARM CORTEX-M7 GCC ARCHITECTURE

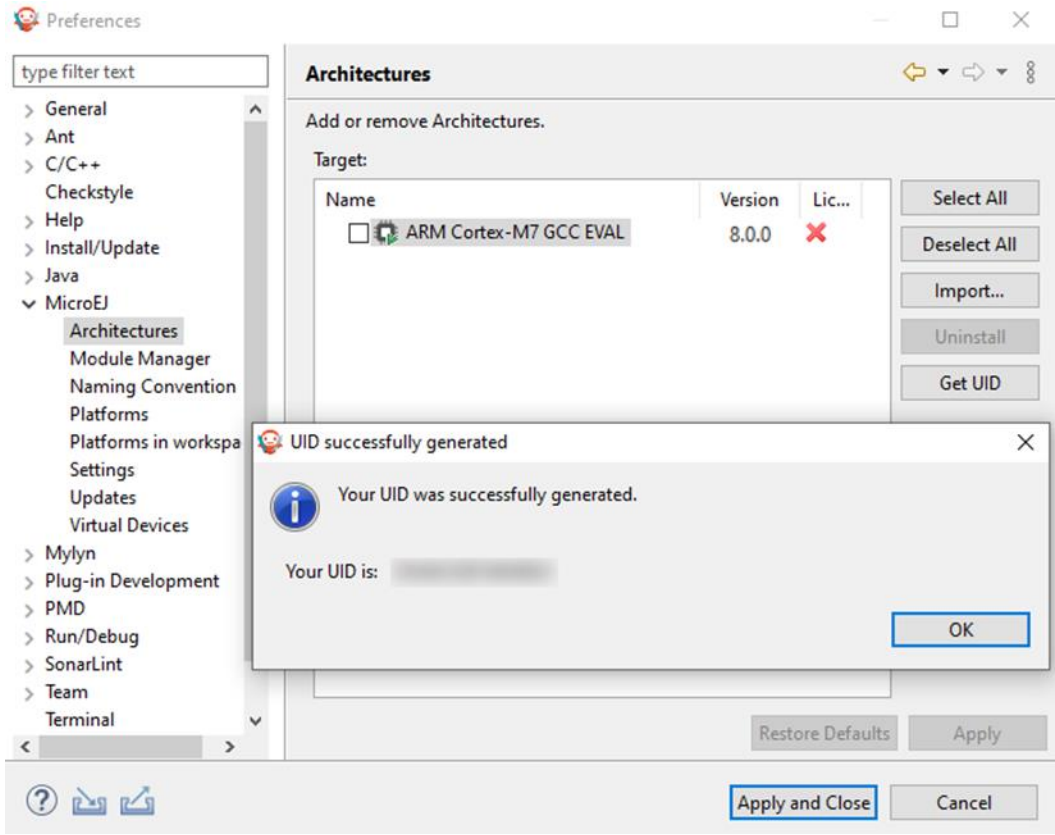
- Download & Install MICROEJ SDK (see download link in slide 5).
- Once installed, launch the MICROEJ SDK and select the default workspace.
- In MICROEJ SDK, click on **Window > Preferences > MICROEJ > Architectures > Import**.
- Select the MICROEJ Architecture previously downloaded **flopi7G26-{version}-eval.xpf** (see download link slide 5).
- Accept the license terms and click on **Finish.**
- The architecture is now imported.
- Click on **Apply and Close** button.



ACTIVATING MICROEJ ARCHITECTURE LICENSE

GETTING THE UID

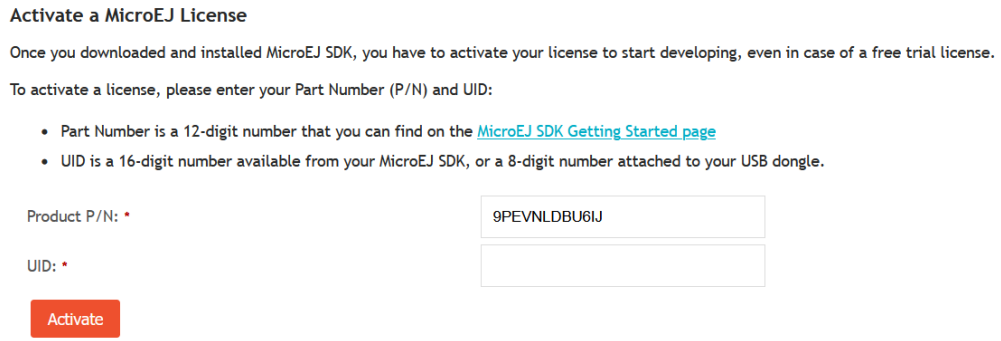
- In MICROEJ SDK, go to **Window > Preferences > MICROEJ > Architectures.**
- Select the **ARM Cortex-M7 GCC EVAL** Architecture.
- Click on Get UID.
- Copy the UID. It will be needed when requesting a license.



ACTIVATING MICROEJ ARCHITECTURE LICENSE

GENERATING THE ACTIVATION KEY

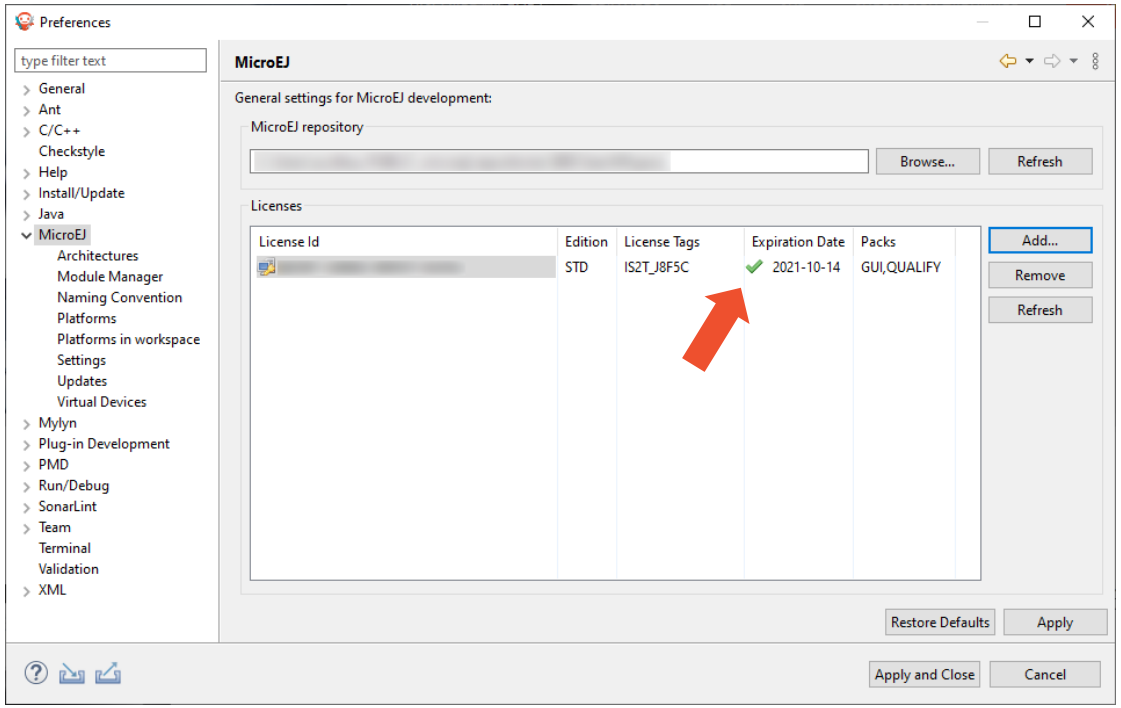
- Go to license.microej.com.
- Click on Create a new account link.
- Create your account with a valid email address. You will receive a confirmation email a few minutes after. Click on the confirmation link in the email and login with your new account.
- Click on Activate a License.
- Set Product P/N: 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

ACTIVATING MICROEJ SDK

- In MICROEJ SDK, go to Window > Preferences > MICROEJ.
- Press **Add..**
- Browse the previously downloaded activation key archive file.
- Press **OK**. 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

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

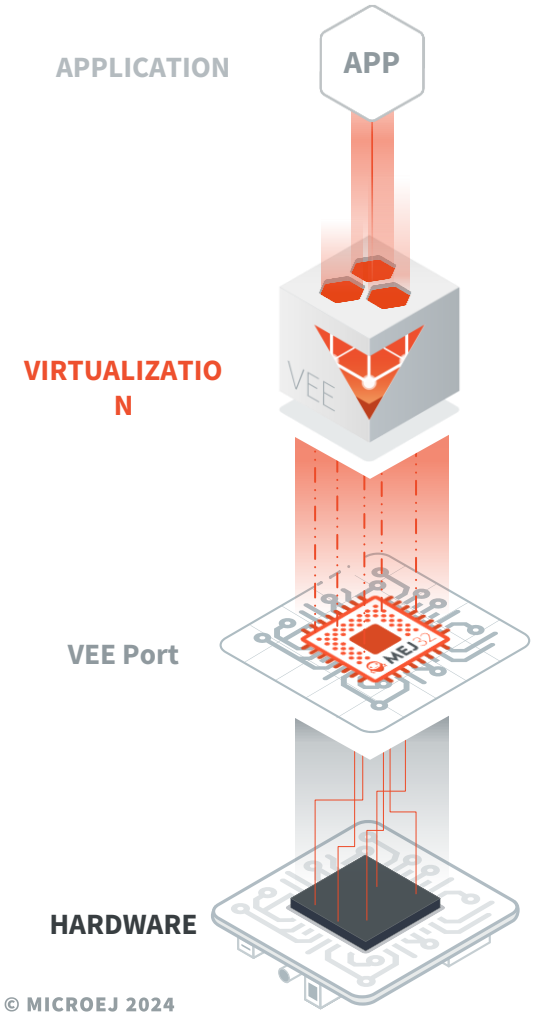
SERVICES

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

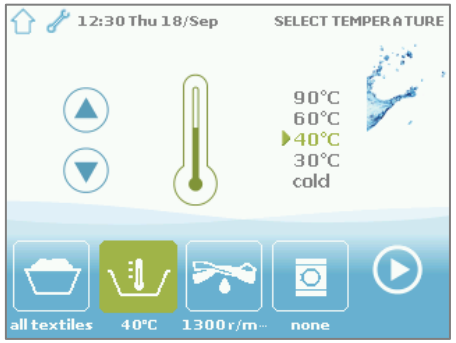
Mass Storage | Ethernet Wi-Fi / LTE | Display | PROCESSOR CORE | UART | Bluetooth | …

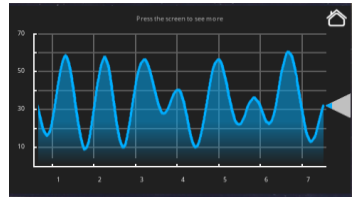
HARDWARE

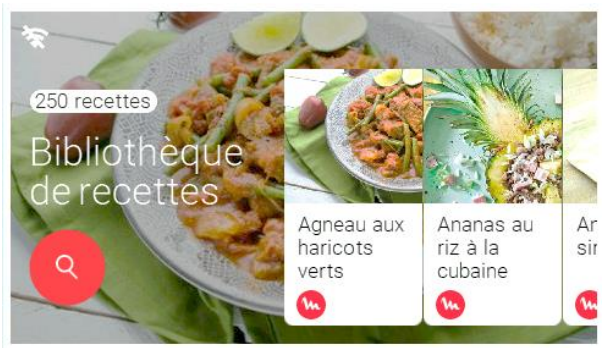
Managed Code (Java, JavaScript, …)

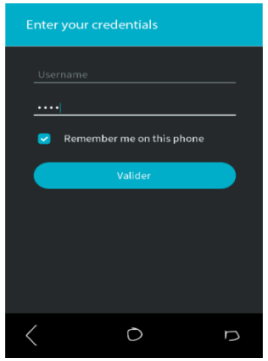
Native Code (C/ASM, …)

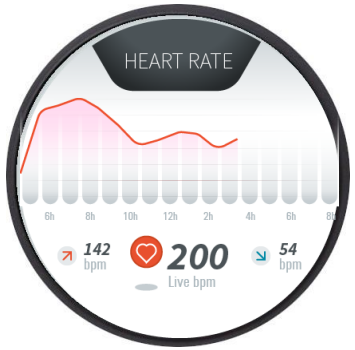
GUI EXAMPLES FOR $1 TO $5 MCU

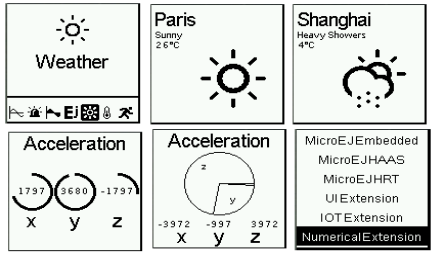












Build Flow

Build flow explained

OVERVIEW

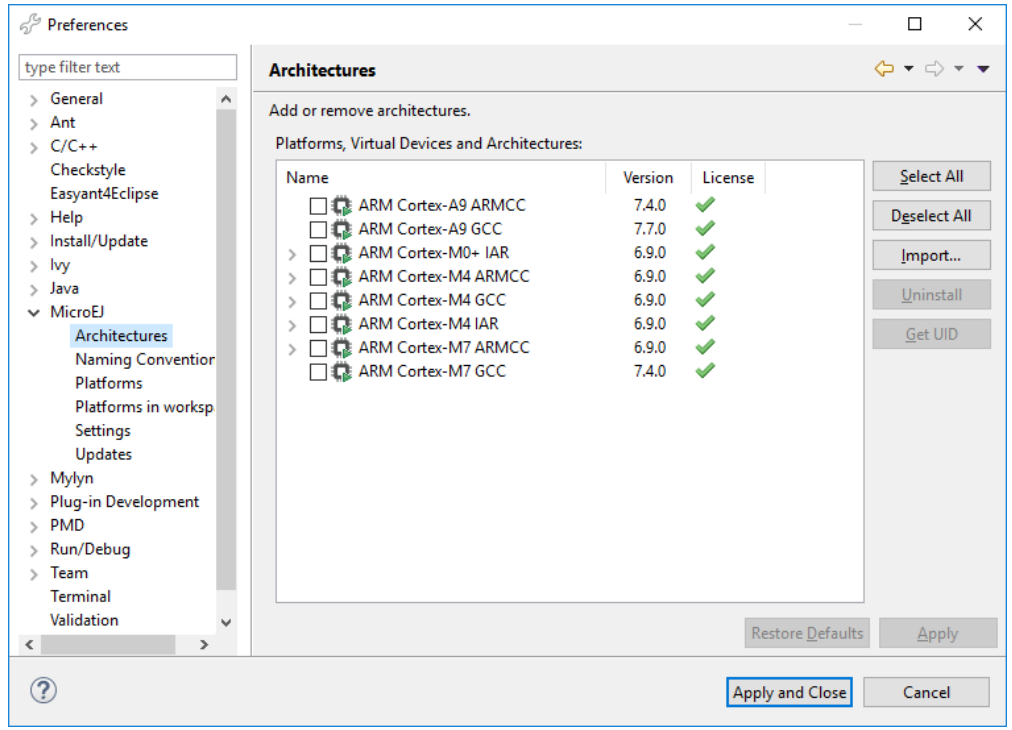
WORKBENCH

MICROEJ SDK includes:

- **MICROEJ IDE**, based on Eclipse.
- Tools to **build VEE Ports.**
- Tools to **build applications.**
- **Add-on libraries** to code with Java as high-level language.
- Native libraries and mechanism to allow developers to use C and to create **interactions between C and Java** features.
- Support for Eclipse plugins.

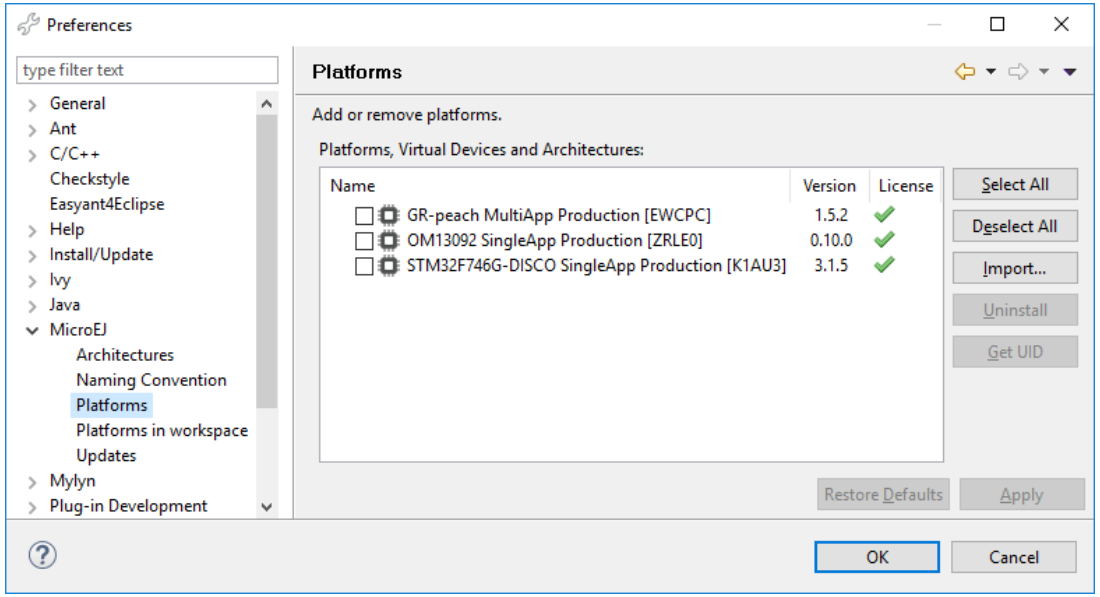
MICROEJ ARCHITECTURE

- 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 and distributed within MICROEJ SDK.
- 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/

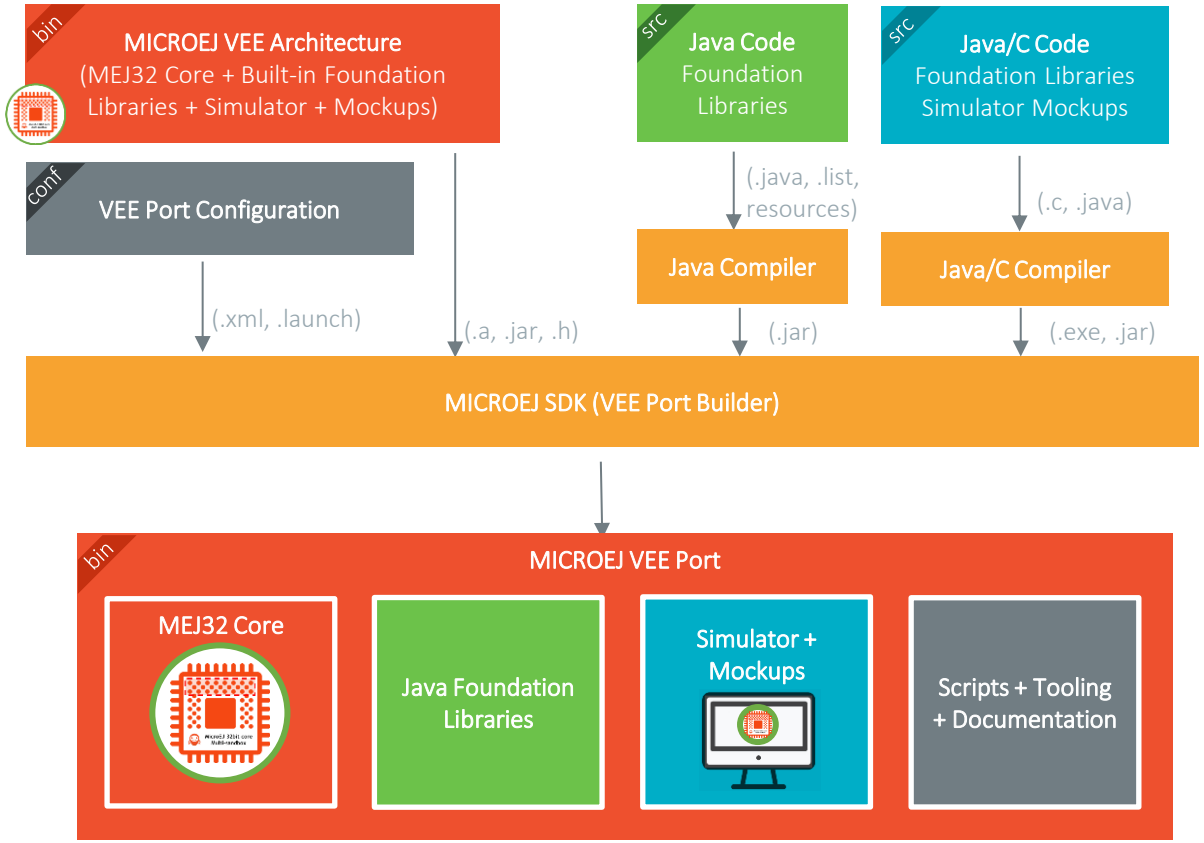


MICROEJ VEE PORT

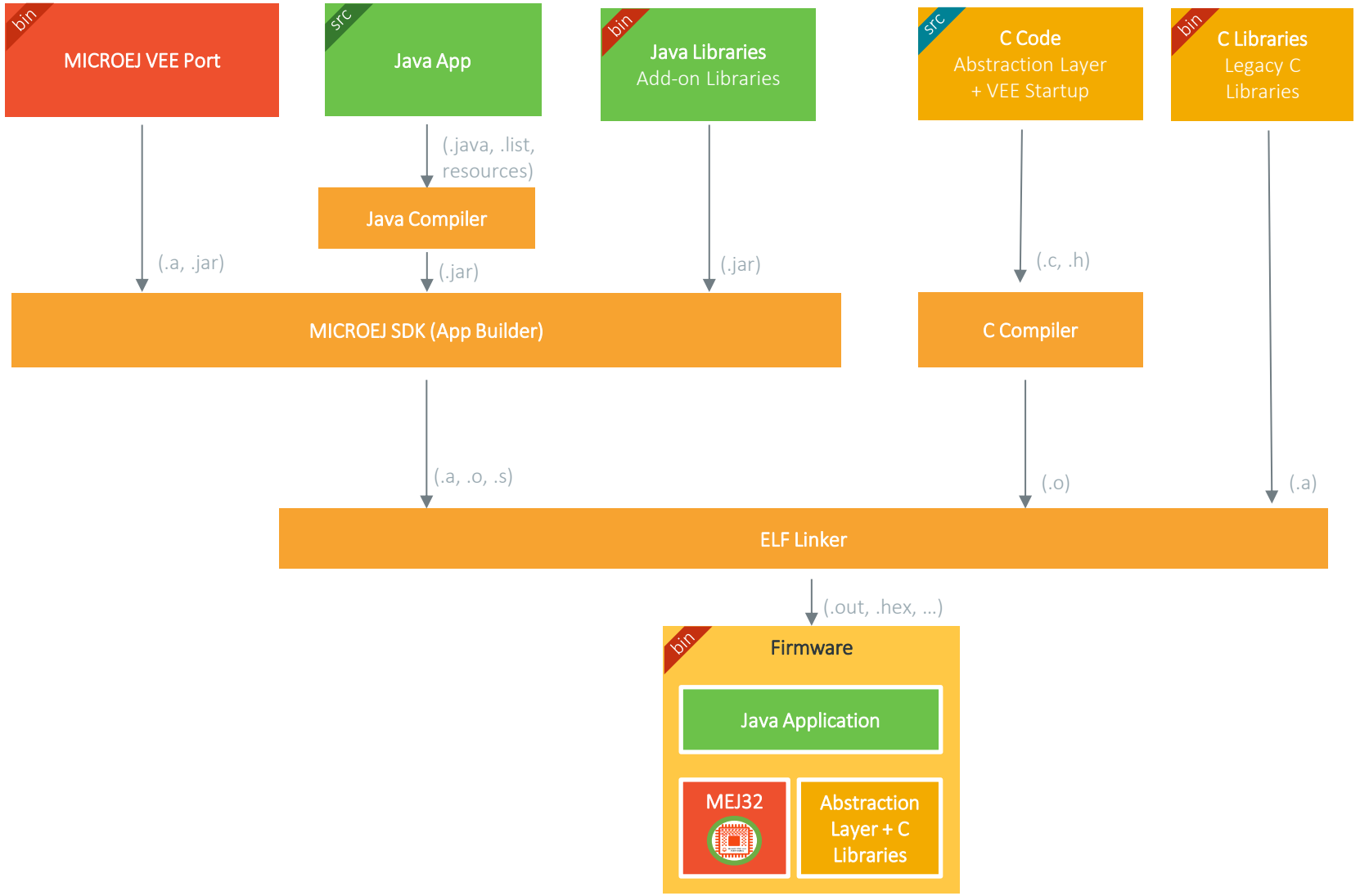
- A **MICROEJ VEE Port** is a port of a MICROEJ Architecture for a specific hardware, RTOS and BSP.
- MICROEJ VEE Ports are built using MICROEJ SDK.
- They are distributed as source (including C sources) or binary (pre-built C BSP).
- Menu **Window > Preferences > MicroEJ > Platforms.**
- Example of 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

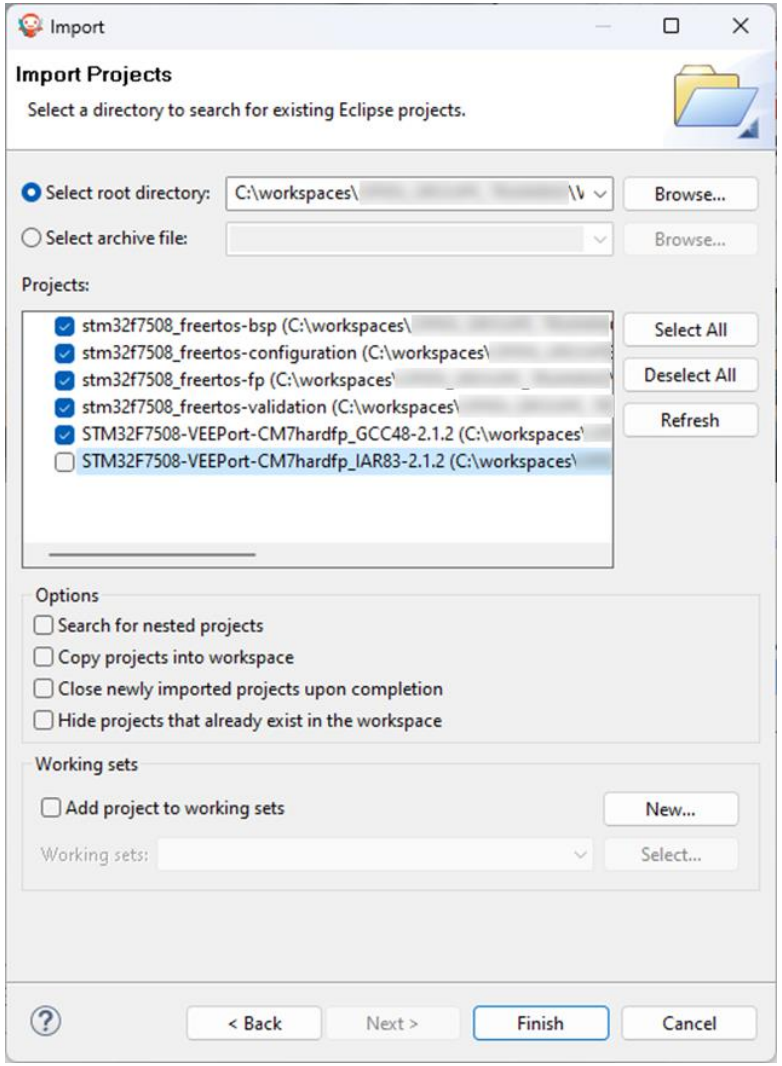


Build VEE Port

For STM32F7508-DK

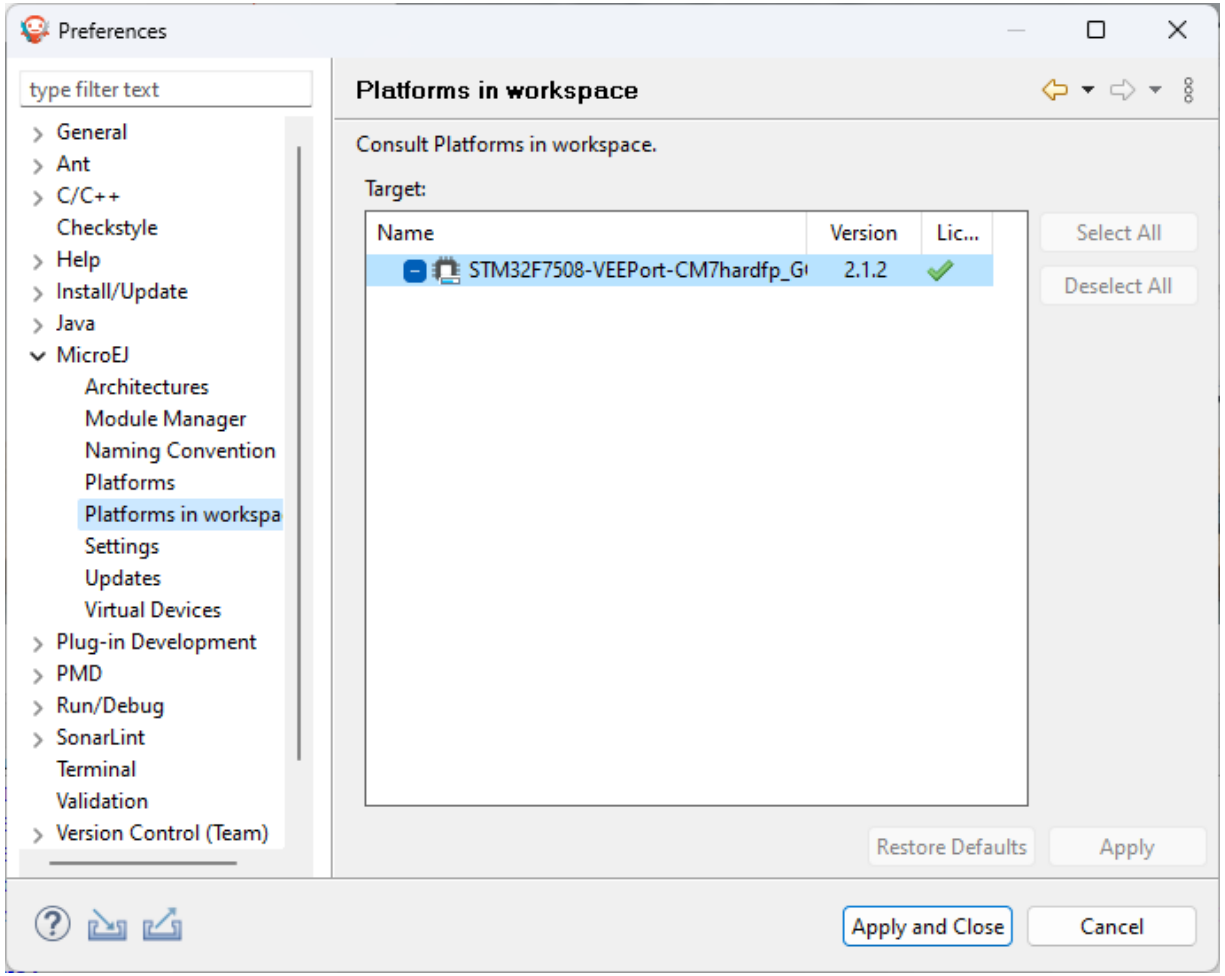
IMPORT VEE PORT SOURCES

- **File > Import... > General > Existing Projects into Workspace**.
- Click the directory where **VEEPort-STMicroelectronics-STM32F7508-DK** has been cloned.
- The projects appears in the Projects list. Select the following ones:
  - **stm32f7508_freertos-configuration**: the configuration project used to configure the VEE Port
  - **stm32f7508_freertos-fp**: the front panel project. It describes the UI of the simulator
  - **stm32f7508_freertos-bsp**: contains the Board Support Package (BSP) source code
  - **STM32F7508-Platform-CM7hardfp_GCC48-{version}**: the VEE Port project (empty)
- Click on **Finish**.



BUILD STM32F7508 VEE PORT

- Right click on **stm32f7508_freertos-configuration** project
- Click on **Build Module** to build the VEE Port.
- The VEE Port project **STM32F7508-Platform-CM7hardfp_GCC48** is now filled.
- You can see the VEE Port in **Platforms in workspace** menu:
  - **Window > Preferences > MicroEJ > Platforms in workspace**



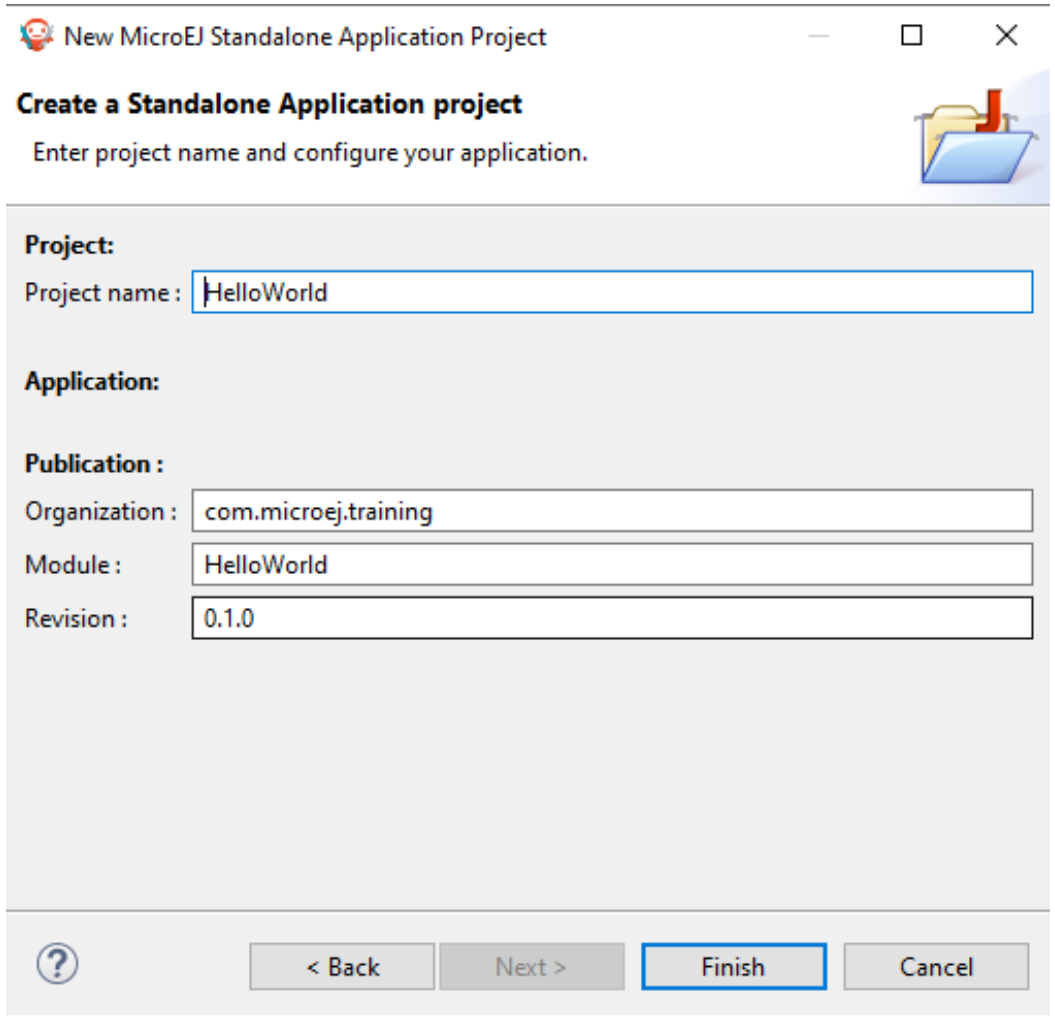
Application

Build & Run

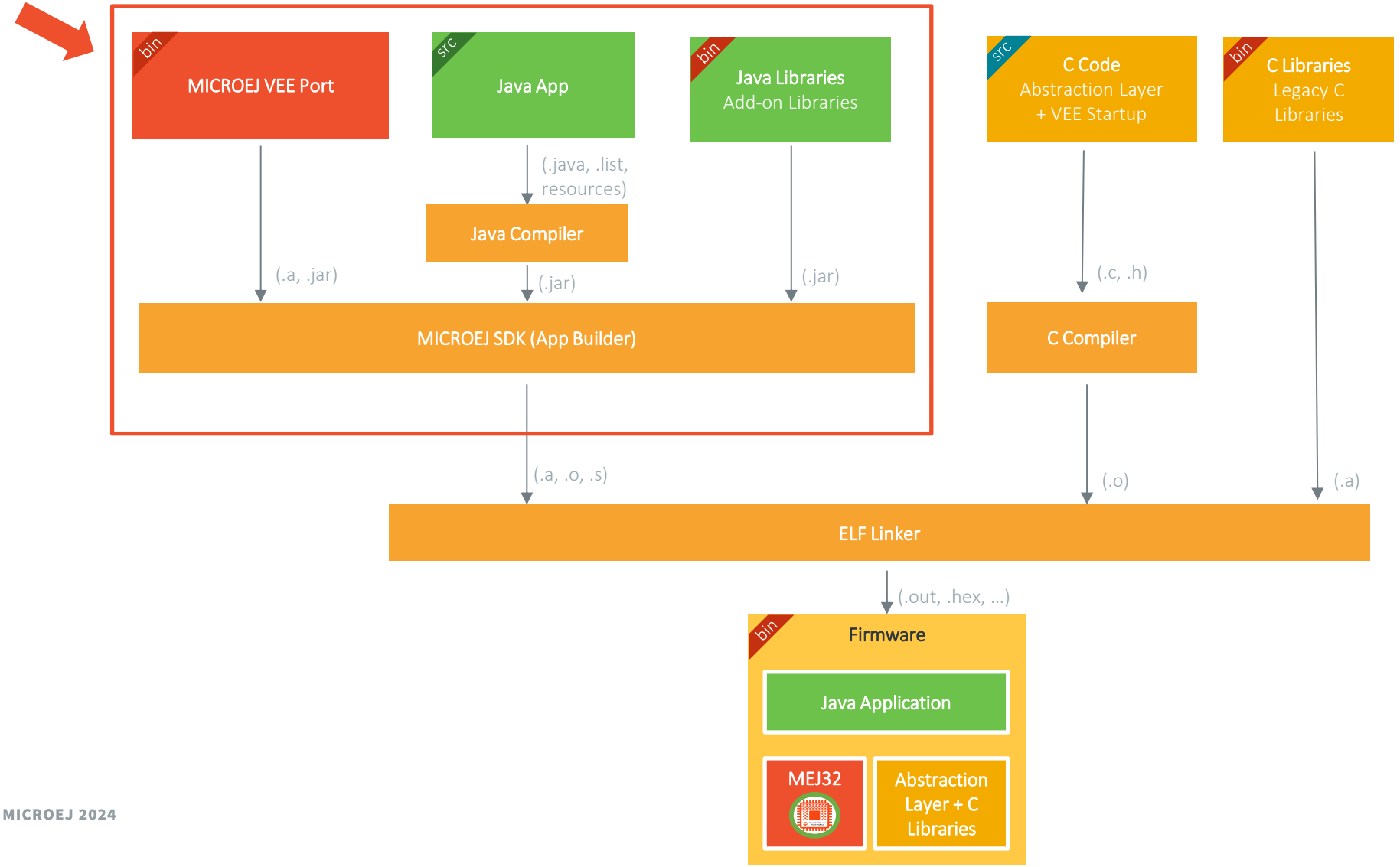
Application Creation

JAVA PROJECT CREATION

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



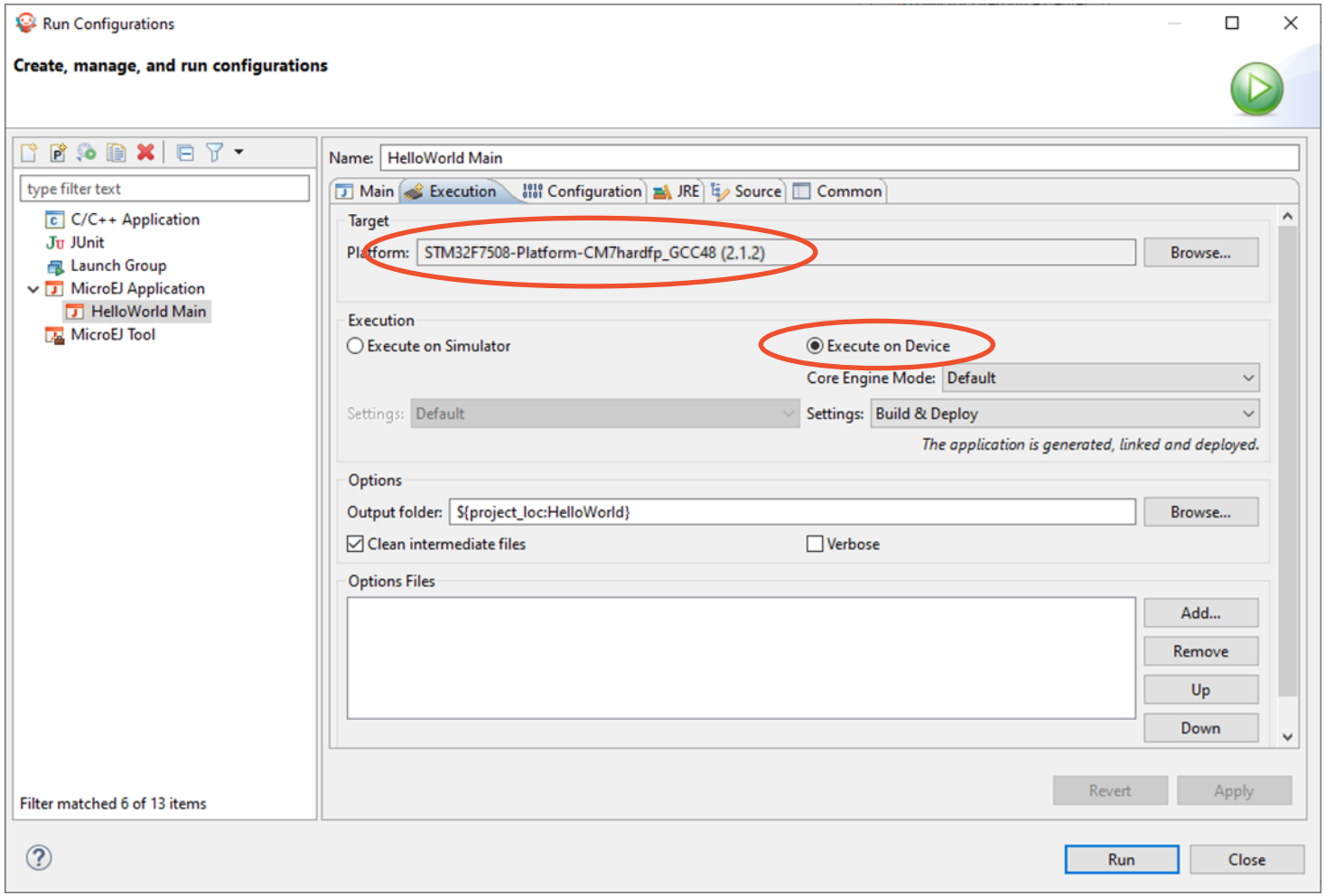
BUILD FLOW



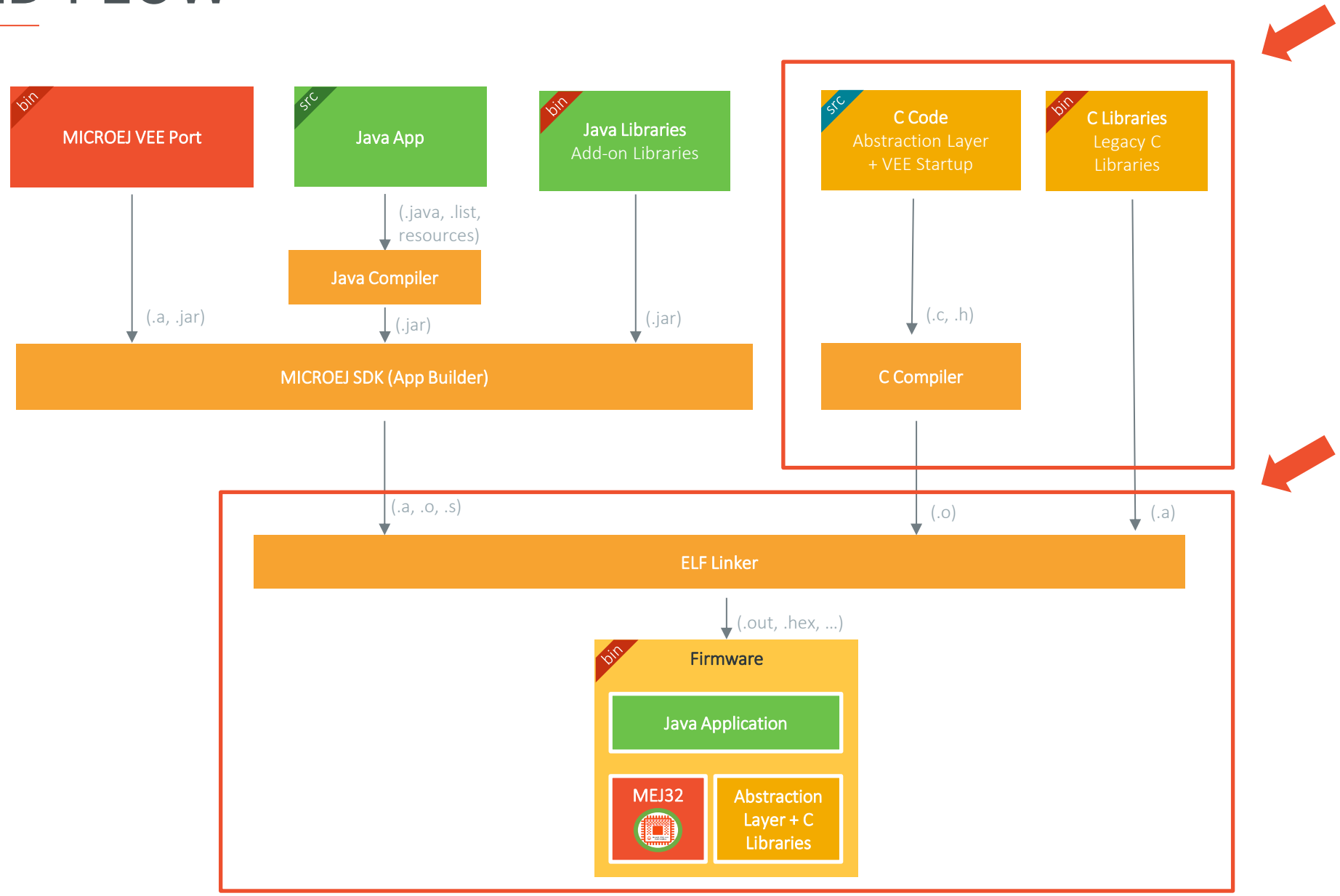
Application Creation

JAVA PROJECT LAUNCHER

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



BUILD FLOW



MICROEJ AND 3RD PARTY IDE

- Java application must be **linked with BSP:**
  - BSP = drivers + (optional: operating system) + abstraction layer.
  - Done by a 3rd party IDE.
- MicroEJ provides:
  - Java application as an **object file (microejapp.o).**
  - Java 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 executable file.**

Run the Java Application on Device

IMPORT THE BSP PROJECT

- Open STM32CubeIDE in an empty workspace.
- Select **File > Import...**
- Select **General > Existing Projects into Workspace**.
- Press **Next**.
- Next to the **Select root directory** field, press **Browse...**
- Navigate to the **stm32f7508_freertos-bsp/projects/microej/SW4STM32** folder.
- Select the **application** project.
- Press **Finish**.

Run the Java Application on Device

BUILD AND FLASH THE BSP

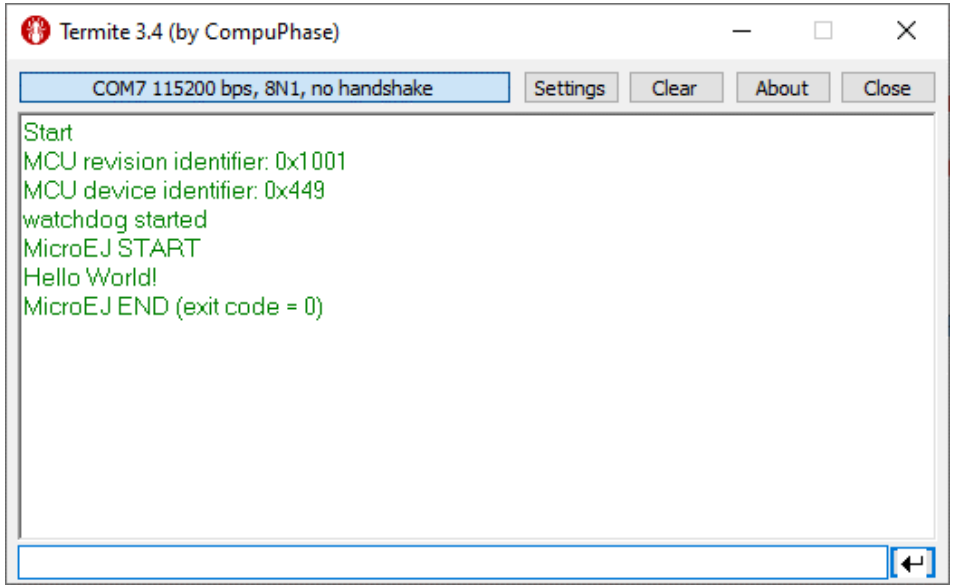
- In STM32CubeIDE, right-Click on the **application** project.
- Press **Build Project**.
- Wait for the end of the build.

- Plug the STM32F7508-DK board to the PC thanks to a mini-USB cable (CN14 - USB ST-Link connector).
- In STM32CubeIDE, select **Run > Run Configurations...**
- Under **STM32 Cortex-M C/C++ Application**, select the **application_debug** run configuration.
- Press **Run**.
- The firmware will be downloaded on the STM32F7508-DK.

Run the Java Application on Device

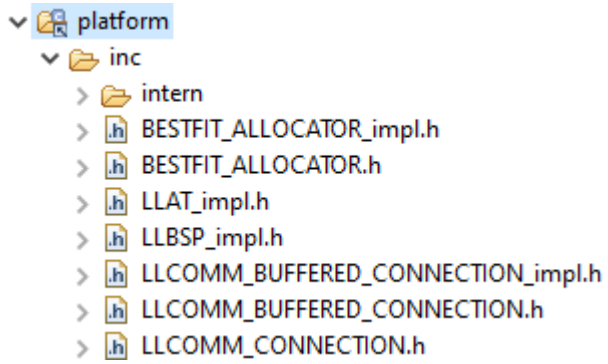
GET THE APPLICATION TRACES

- Open the Termite serial terminal.
- Click the **Settings** button.
- Select the STM32F7508-DK board COM port.
- Reset the STM32F7508-DK board pressing the **black** button.
- The application starts and the **Hello World** message is printed in the console!

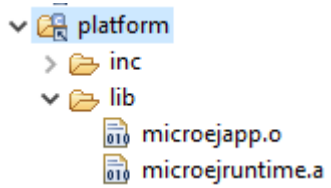


MicroEJ Core Engine Startup

- MicroEJ header files are in: **projects/microej/SW4STM32/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 STM32F7508 VEE Port, **microej_main()** is called from a FreeRTOS task in **main.c.**
It is also possible to run MicroEJ Core Engine on a baremetal device (no RTOS).

Application

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:

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

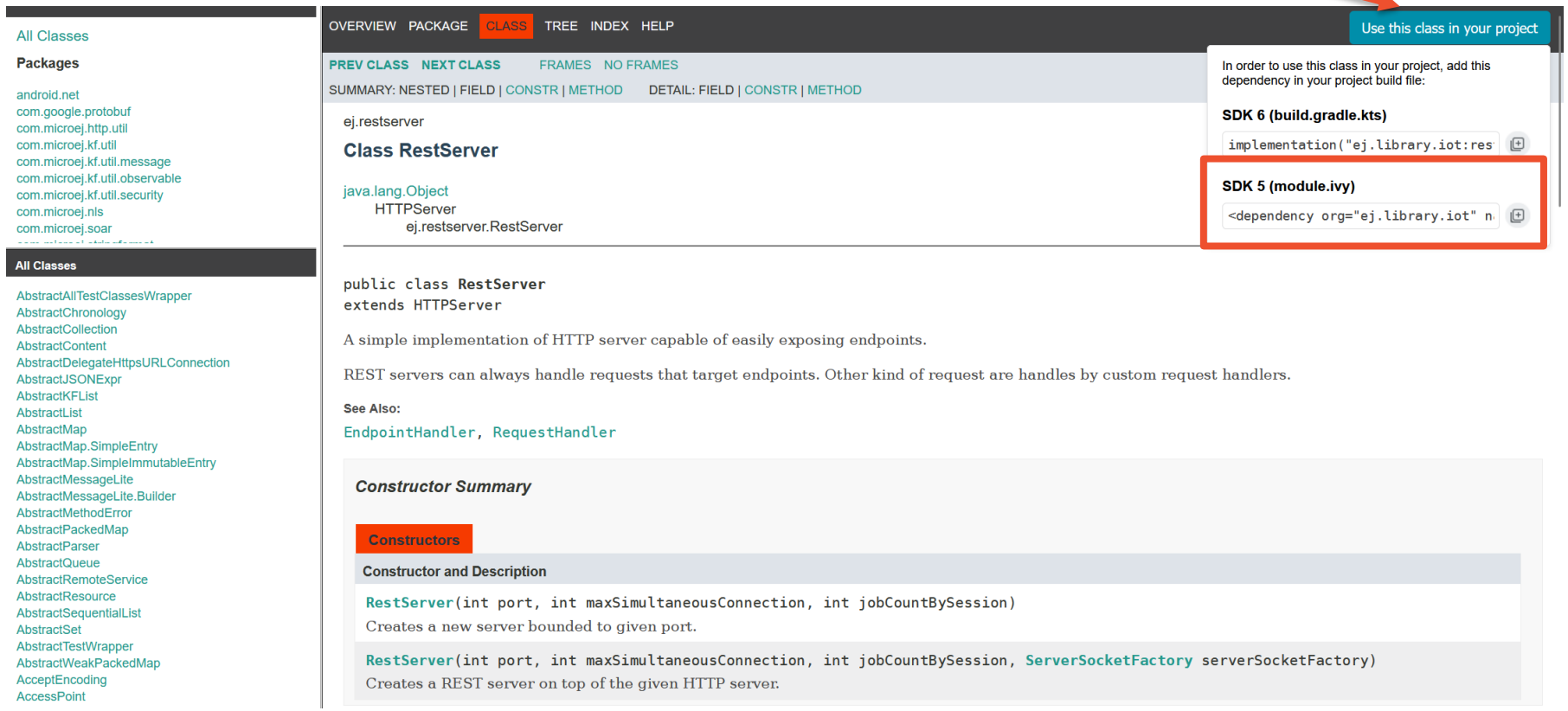
From the MICROEJ Javadoc you can search for a Class and get the MMM dependency that provides it by visiting https://repository.microej.com/javadoc/microej_5.x/apis/index.html

LIBRARY DEPENDENCY FILE

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.



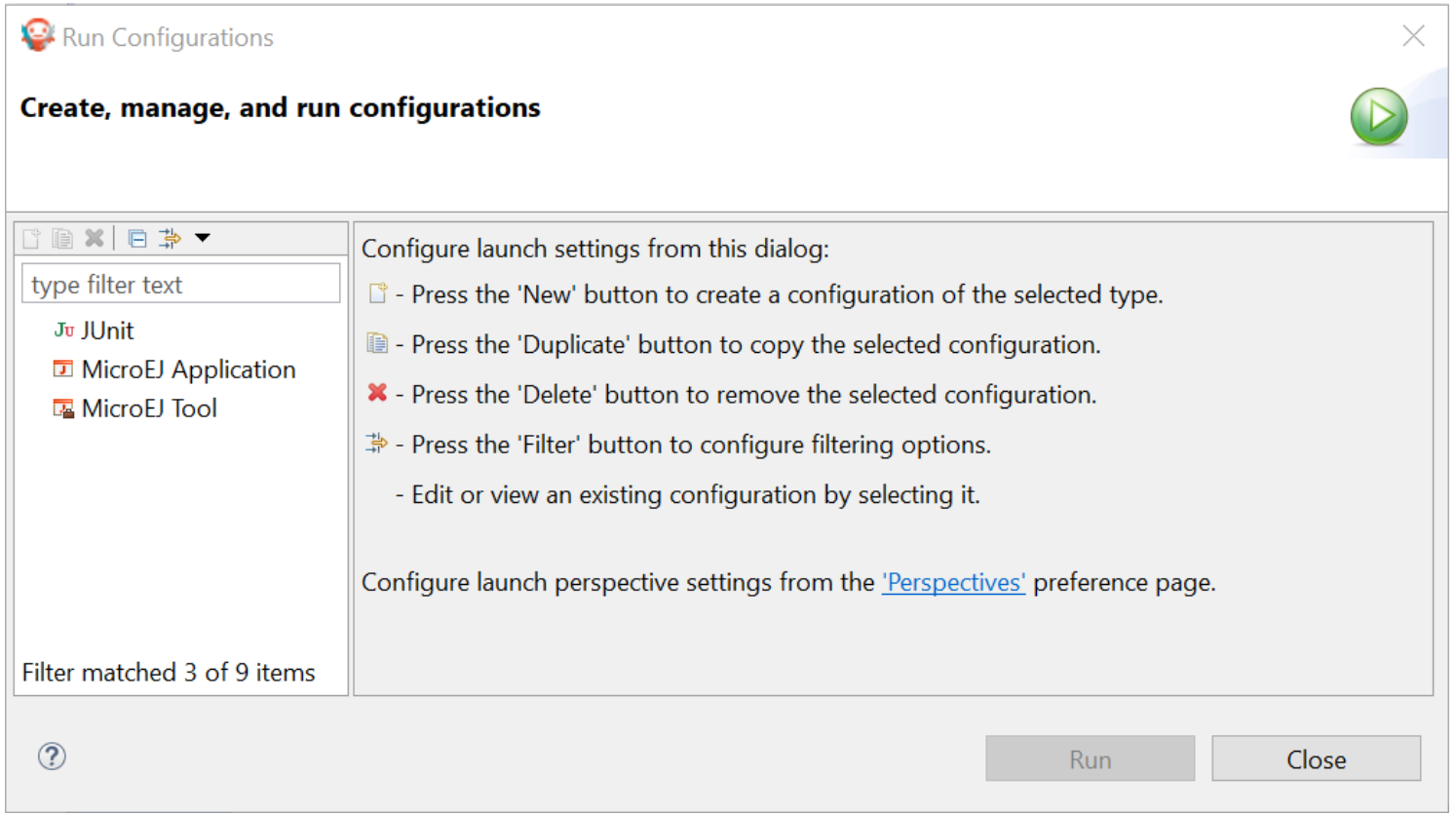
Run Configurations

- 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

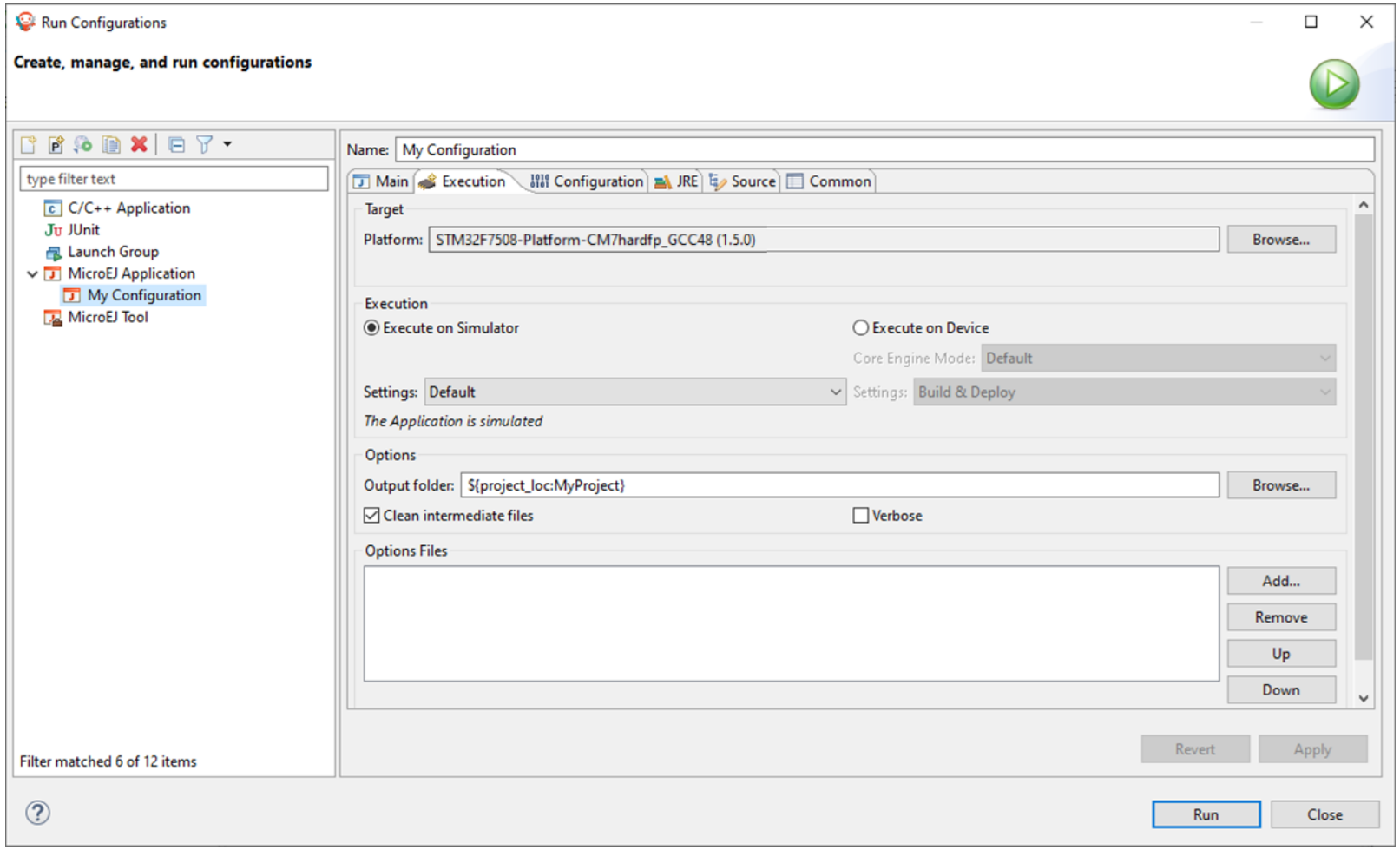
MICROEJ provides two specific run configuration types:

- MICROEJ Application
- MICROEJ Tool



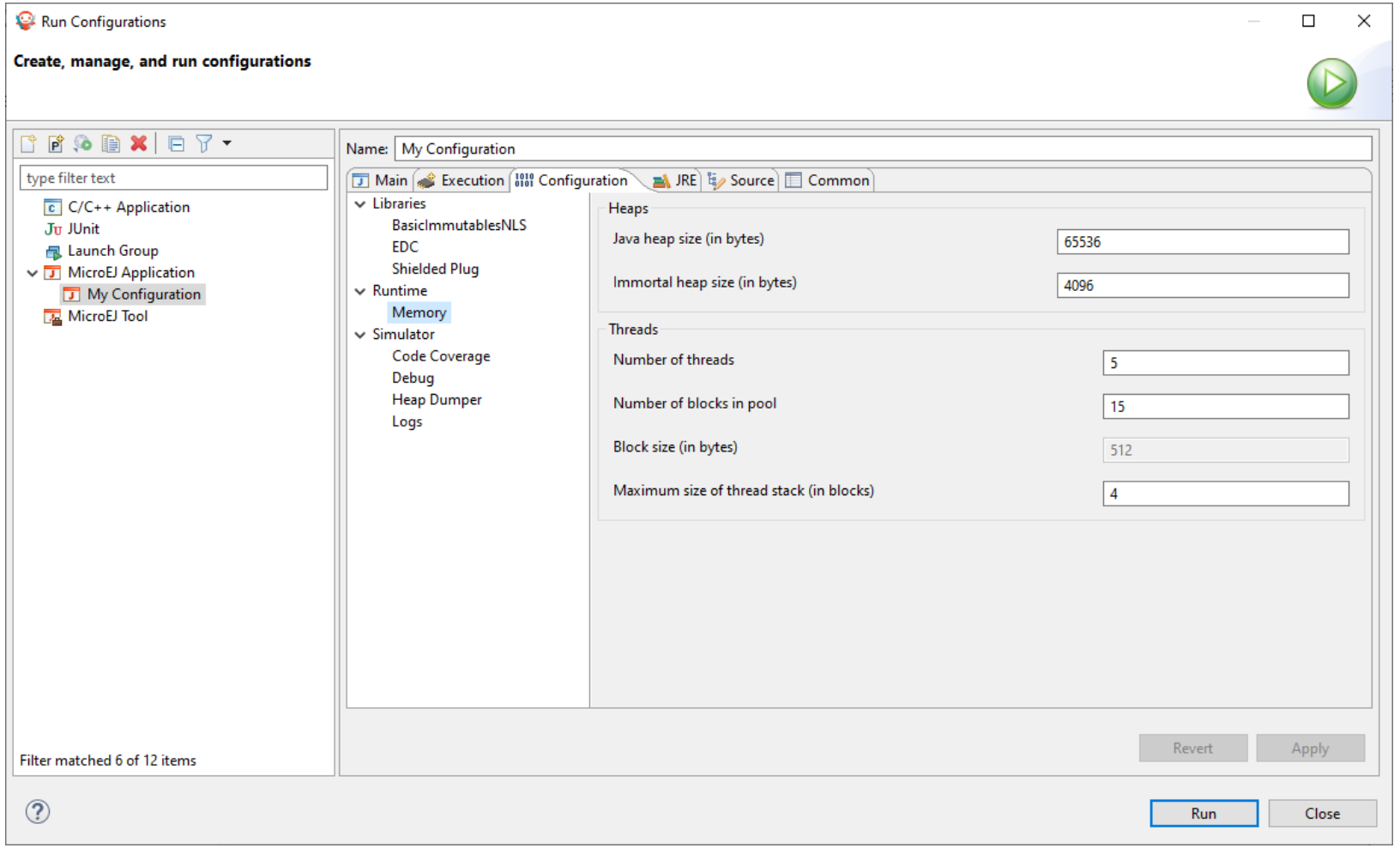
MICROEJ APPLICATION

KIND OF EXECUTION (SIMULATOR OR DEVICE)



MICROEJ APPLICATION

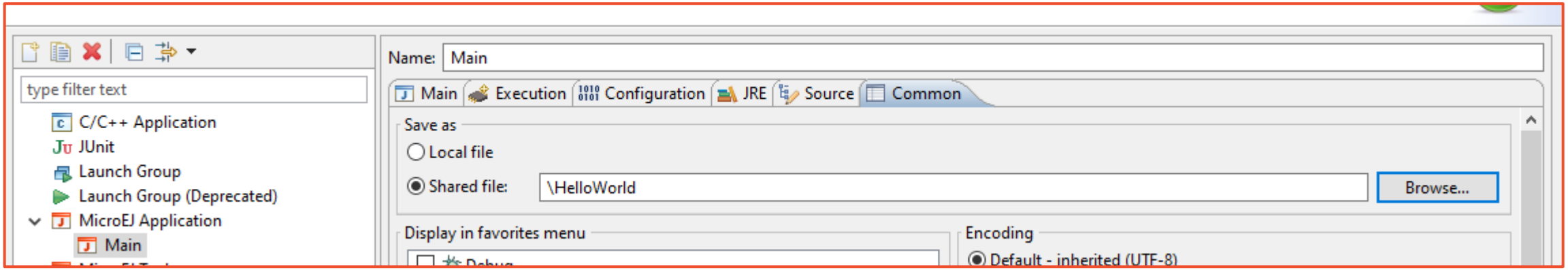
CONFIGURE LIBRARIES AND MEMORY USAGE



RUN CONFIGURATION

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



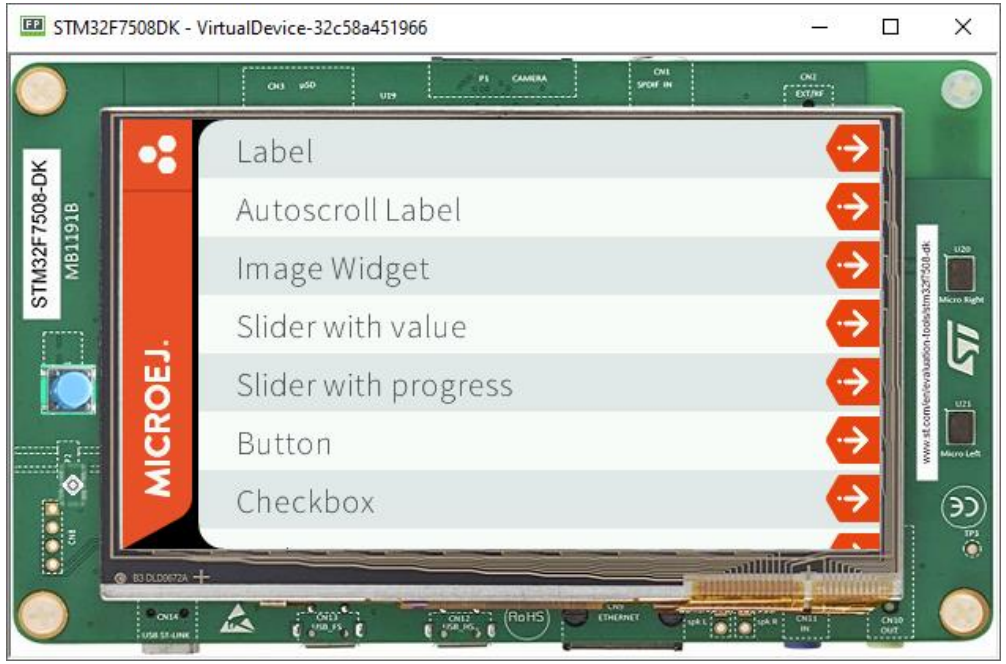
Front Panel

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.



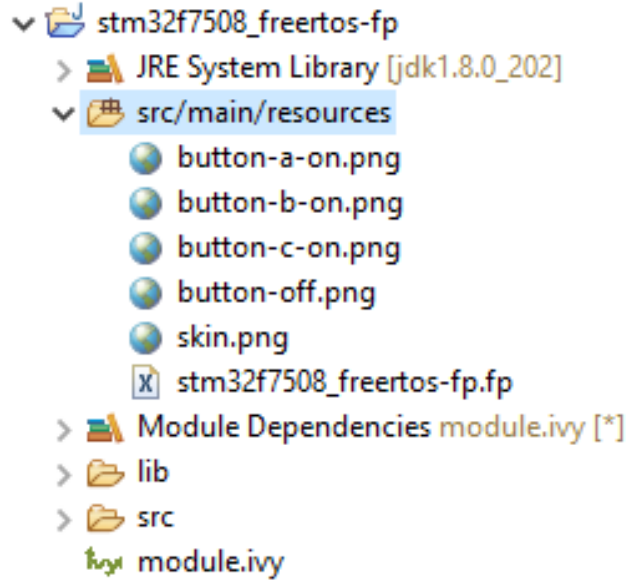
See https://docs.microej.com/en/latest/PlatformDeveloperGuide/frontpanel.html

FRONT PANEL

PROJECT CONTENT

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.

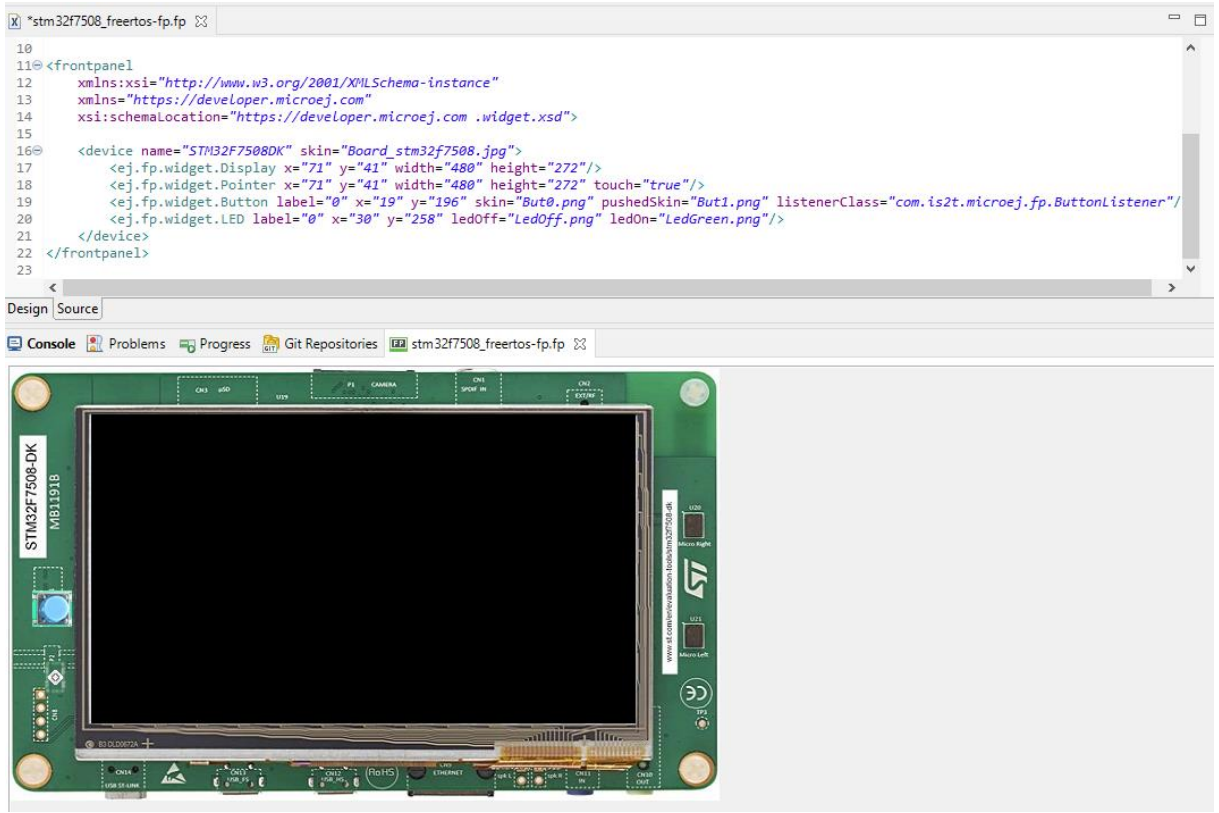
```
<frontpanel
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="https://developer.microej.com"
    xsi:schemaLocation="https://developer.microej.com .widget.xsd">

    <device name="STM32F7508DK" skin="Board_stm32f7508.jpg">
        <ej.fp.widget.Display x="71" y="41" width="480" height="272"/>
        <ej.fp.widget.Pointer x="71" y="41" width="480" height="272" touch="true"/>
        <ej.fp.widget.Button label="0" x="19" y="196" skin="But0.png" pushedSkin="But1.png" listenerClass="com.is2t.microej.fp.ButtonListener"/>
        <ej.fp.widget.LED label="0" x="30" y="258" ledOff="LedOff.png" ledOn="LedGreen.png"/>
    </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:

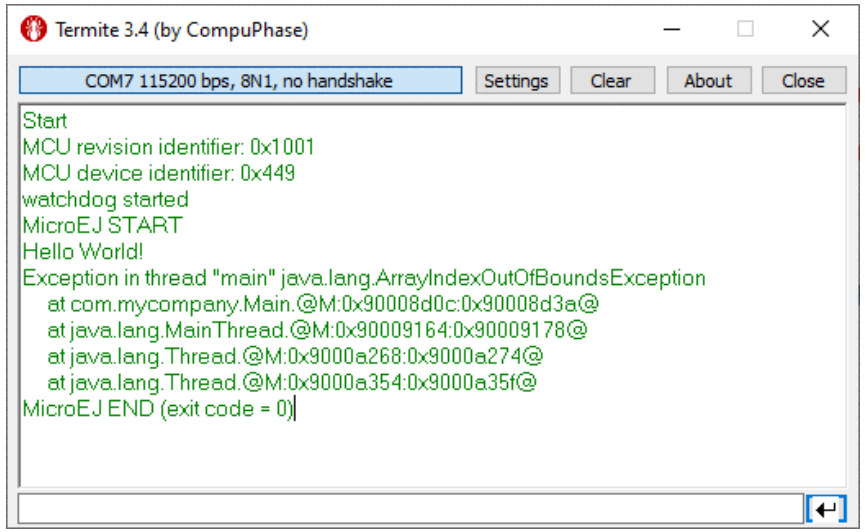
```
byte[] array = new byte[5];
array[5] = 42; // Invalid access to the array
```

- 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:



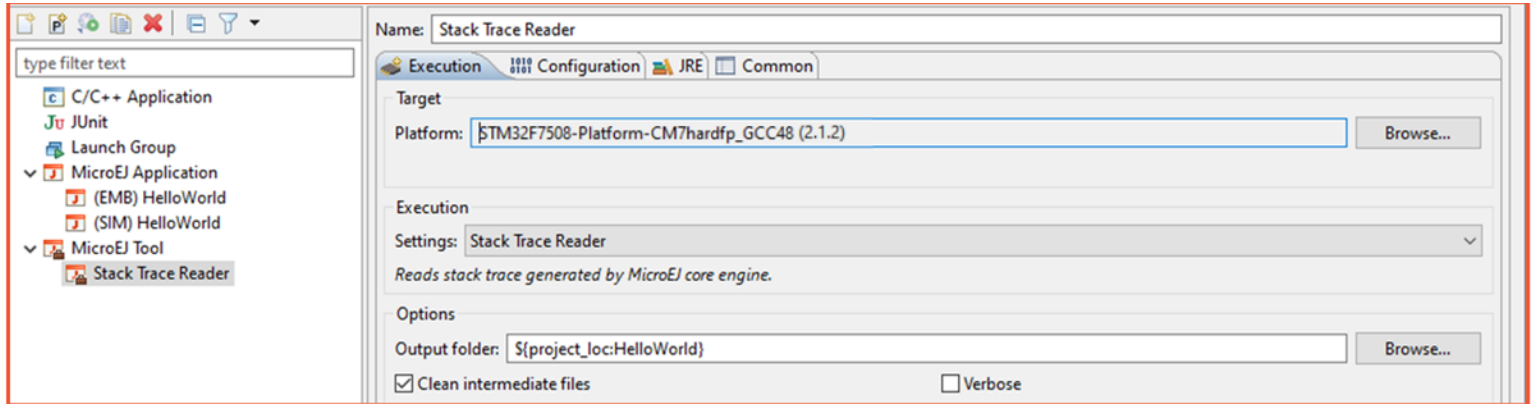
- Name of the faulty **method is not printed** directly:
  - Only the address of the method is printed
  - MICROEJ does not embed the names of the methods to limit the footprint
- To help reading the stack trace, a tool is available: **the stack trace reader**

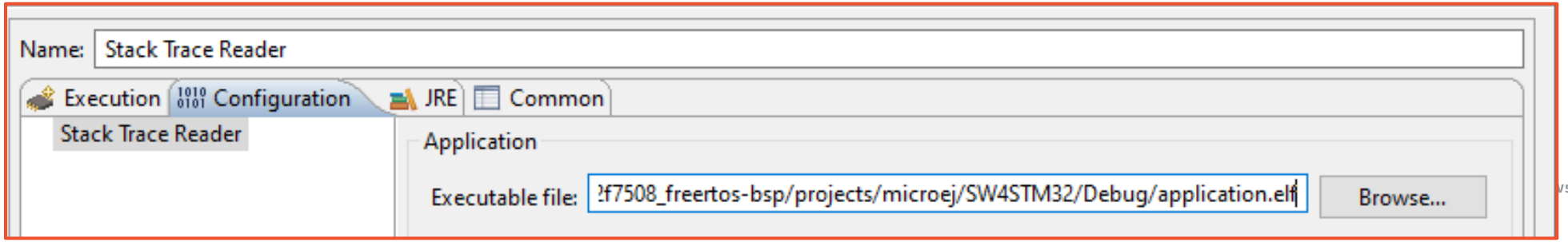
STACK TRACE READER

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:



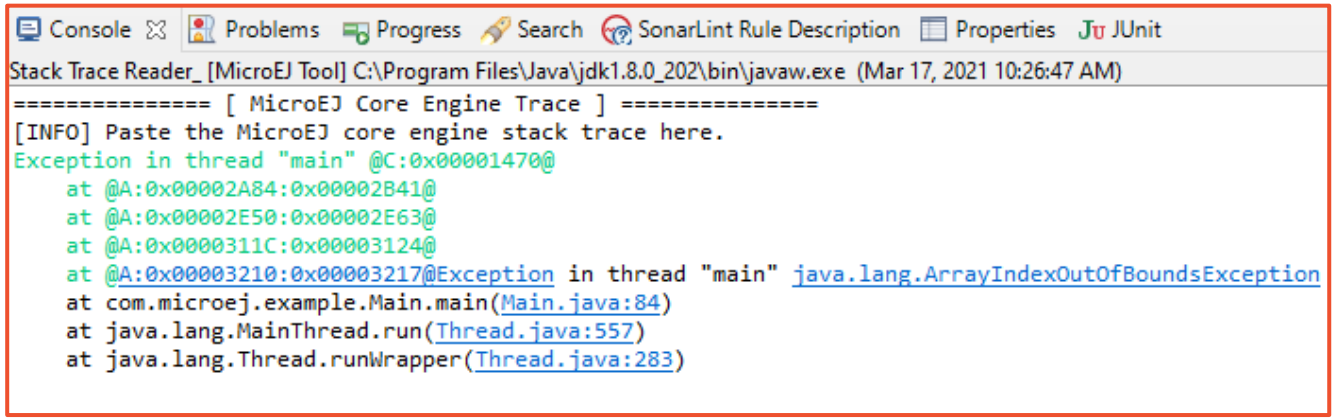


STACK TRACE READER

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

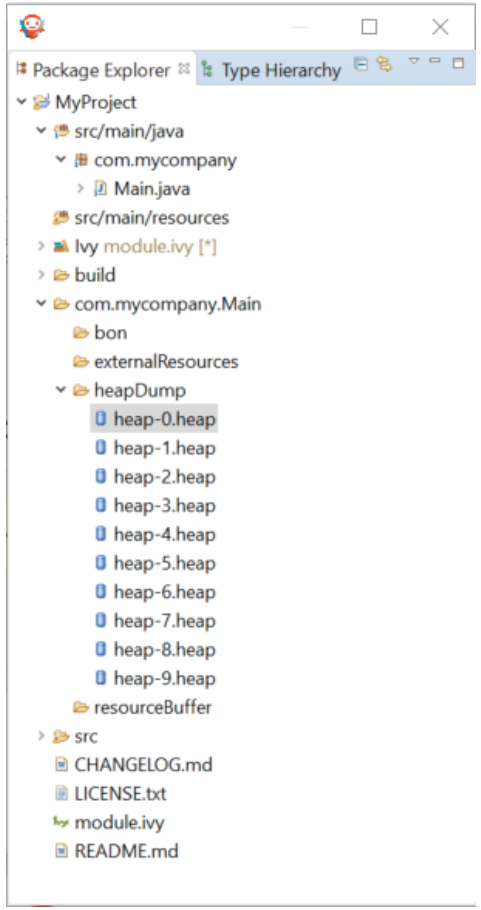
DEBUGGER

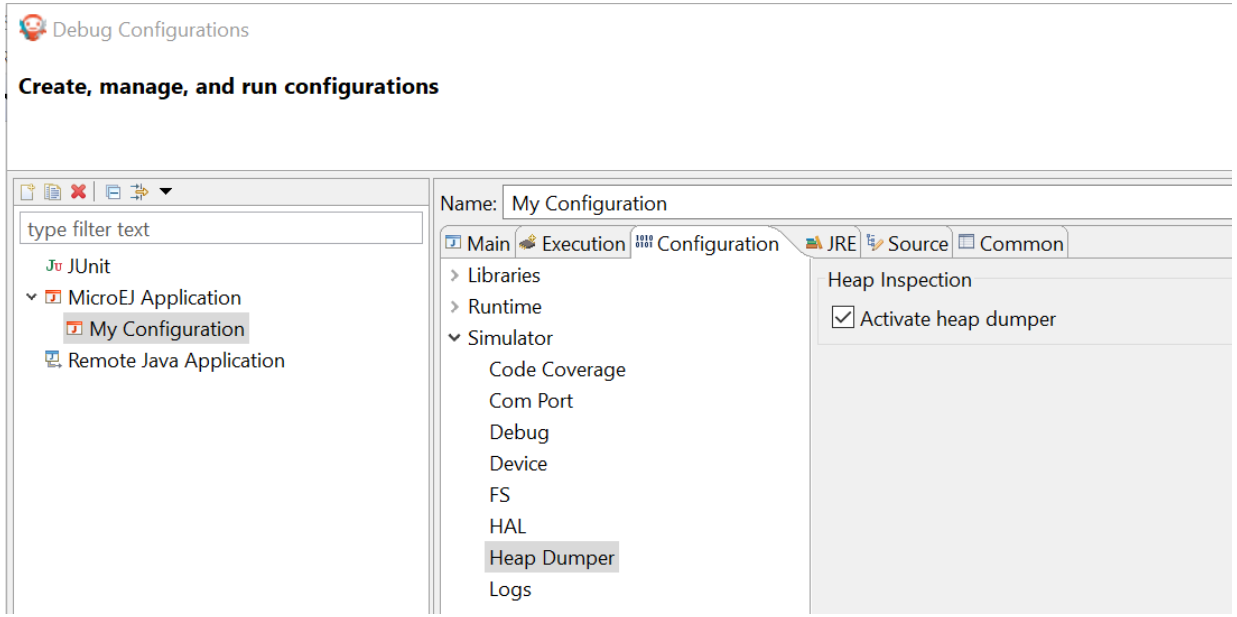
- JDWP (Java Debug Wire Protocol) to use Eclipse debugger.
- Classical debugger features:
  - Breakpoints.
  - Step-by-step execution.
  - Variables and fields value monitoring.
  - Thread execution stacks list.
- Run your Launch Configuration as a Debug Configuration:
  - Debug perspective.

Heap Dumper

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

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

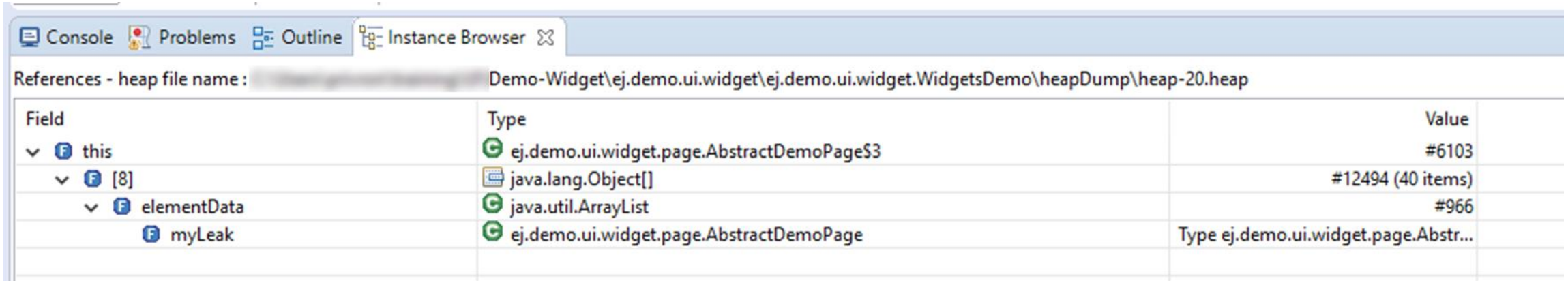




Heap Dumper

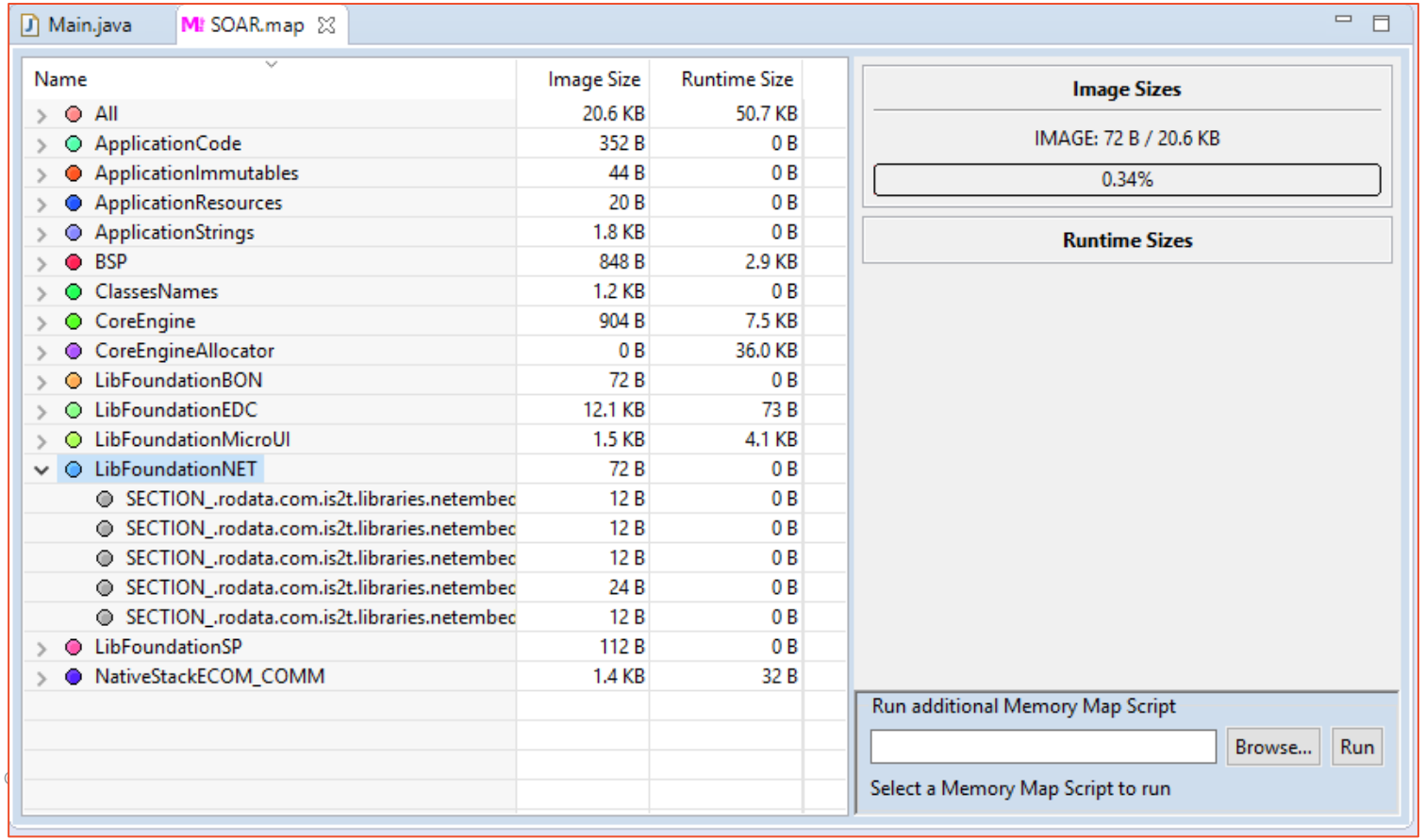
- Open .heap files with the Heap Analyzer plugin.
- Inspect objects graph.
- Detect memory leaks.
- This is an advanced feature: a good knowledge of Java and the program is required.

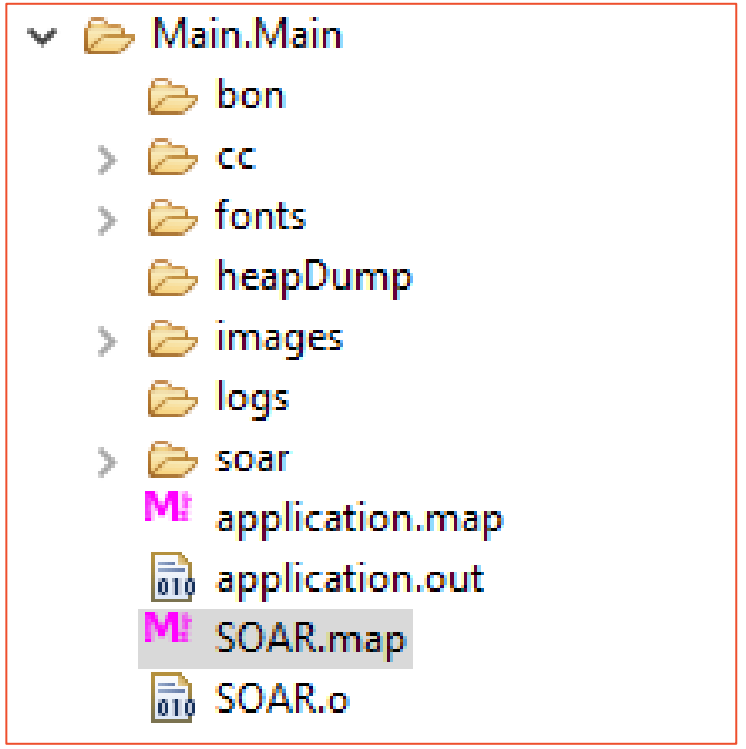
Heap Dumper



Memory Map Inspector

A *SOAR.map* file is generated when a build for device is done.
The map file maps Java and MICROEJ memory usage (no BSP).

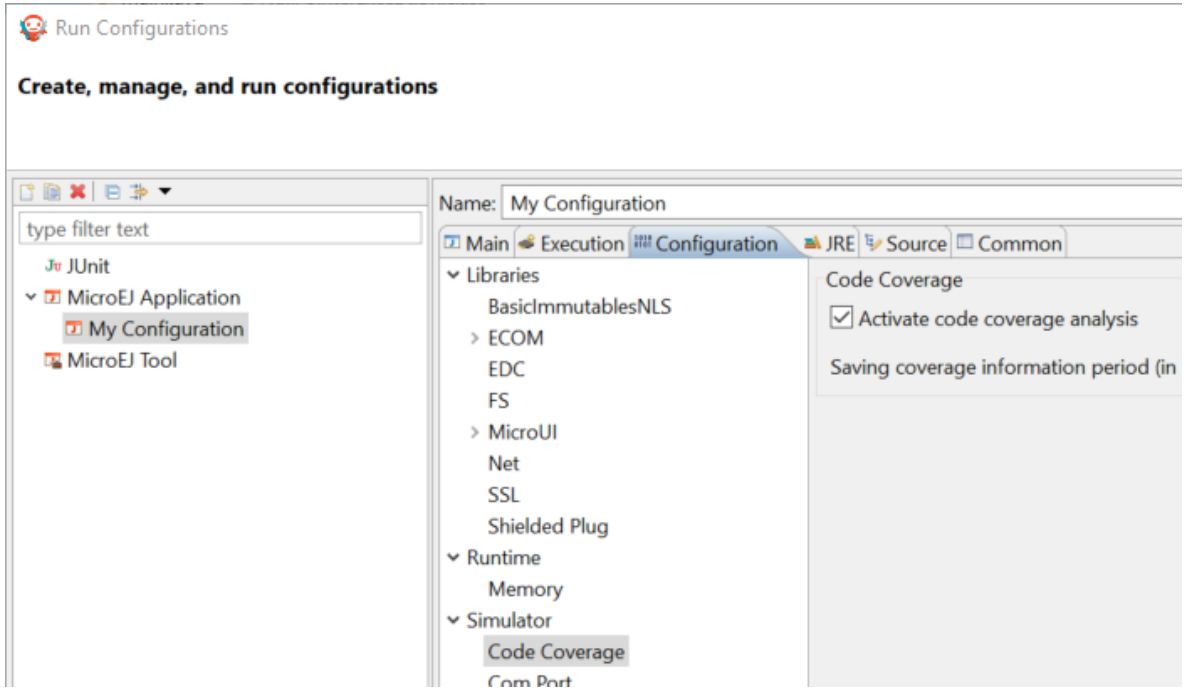




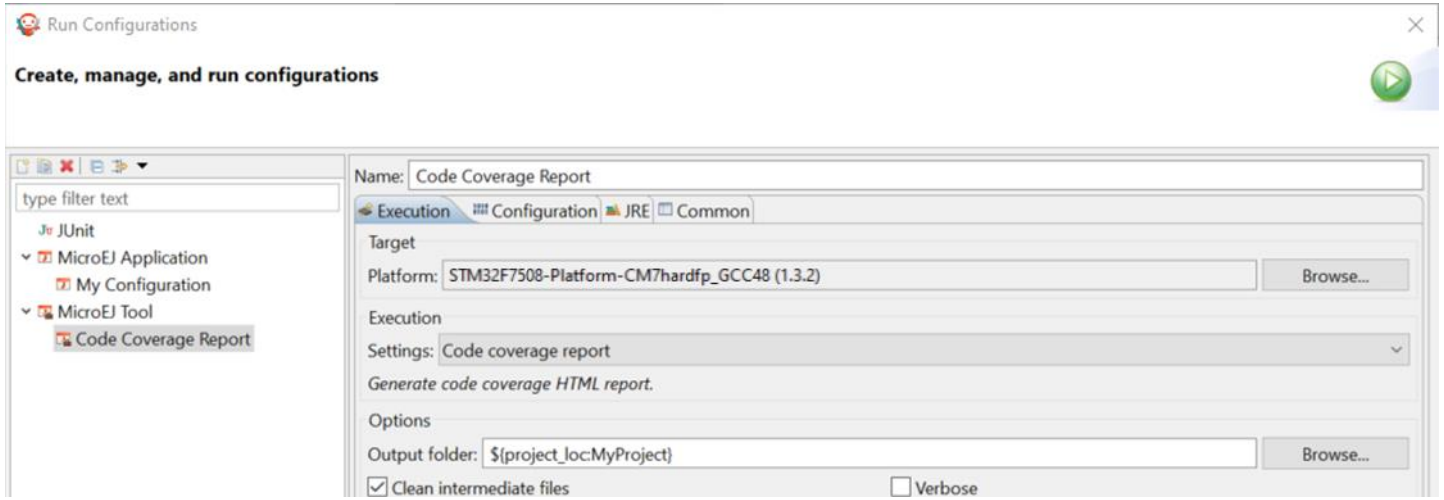
Code Coverage

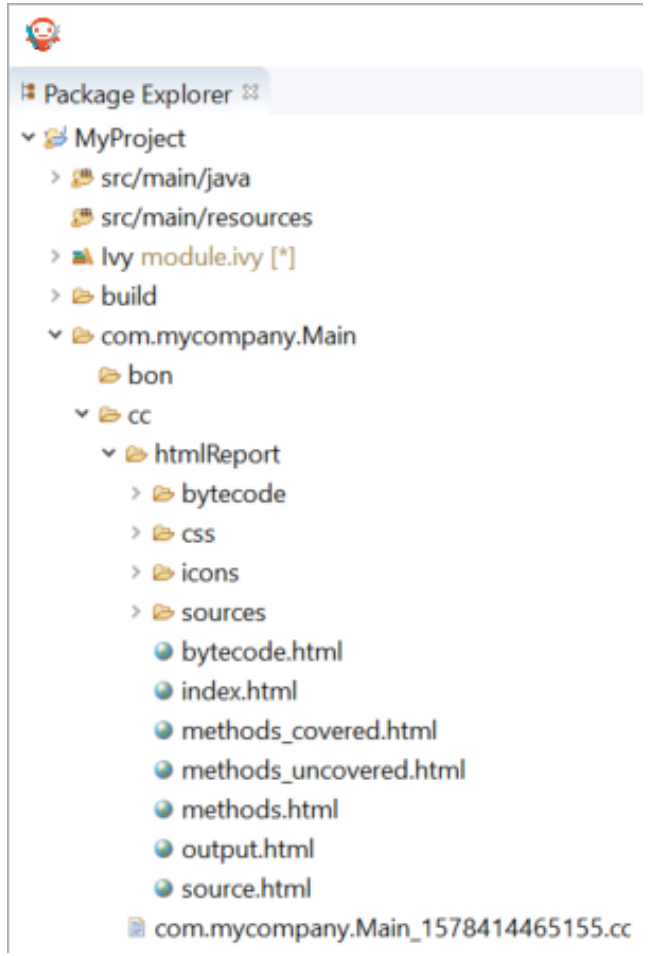
Code coverage reports:

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



Code Coverage





Dump the States of the Core Engine

LLMJVM Dump

- What?
  - Prints the state of the MicroEJ Core Engine to the standard output stream.
  - For each Java thread, the Java stack trace, the name, the state and the priority are printed.
- How-To?
  - Call the C function **LLMJVM_dump()**.
  - It is provided by **LLMJVM.h**.
- When?
  - Call the **LLMJVM_dump** as a last resort in a fault handler to get a snapshot of the Core Engine, to check if the issue comes from a LLAPI or the underlying C code.
  - Call the **LLMJVM_dump** in the Core Engine task at runtime to diagnose unexpected behavior (ex: UI freeze).
- Requirements:
  - A way to read stdout (usually UART).

LLMJVM Dump Example

```
void HardFault_Handler(void)
{
      uint32_t hfsr;
    print_stacked_registers();
    puts(__func__);

    hfsr = SCB->HFSR;
    printf("Hard Fault Status Register =\t%lX\n", hfsr);

    if(hfsr & SCB_HFSR_FORCED_Msk)
    {
        printf("FORCED");
    }

    LLMJVM_dump();

    INFINITE_LOOP();
}
```

```
================================== VM Dump ==================================
Java threads count: 3
Peak java threads count: 3
Total created java threads: 3
Last executed native function: 0x90035E3D
Last executed external hook function: 0x00000000
State: running
-----------------------------------------------------------------------------
Java Thread[1026]
name="main" prio=5 state=RUNNING max_java_stack=456 current_java_stack=184

java.lang.MainThread@0xC0083C7C:
    at (native) [0x90003F65]
    at com.microej.demo.widget.main.MainPage.getContentWidget(MainPage.java:95)
        Object References:
            ...
-----------------------------------------------------------------------------
Java Thread[1536]
name="Thread1" prio=5 state=READY max_java_stack=60 current_java_stack=57

java.lang.Thread@0xC0082194:
    at java.lang.Thread.runWrapper(Unknown Source)
        Object References:
            - java.lang.Thread@0xC0082194
    at java.lang.Thread.callWrapper(Thread.java:449)
...
=============================================================================

============================== Garbage Collector ============================
State: Stopped
Last analyzed object: null
Total memory: 15500
Current allocated memory: 7068
Current free memory: 8432
Allocated memory after last GC: 0
Free memory after last GC: 15500
=============================================================================

============================== Native Resources =============================
Id         CloseFunc  Owner            Description
-----------------------------------------------------------------------------
```

- Note: the **Stack Trace Reader** can be used to decode the trace of the **LLMJVM_dump()**.

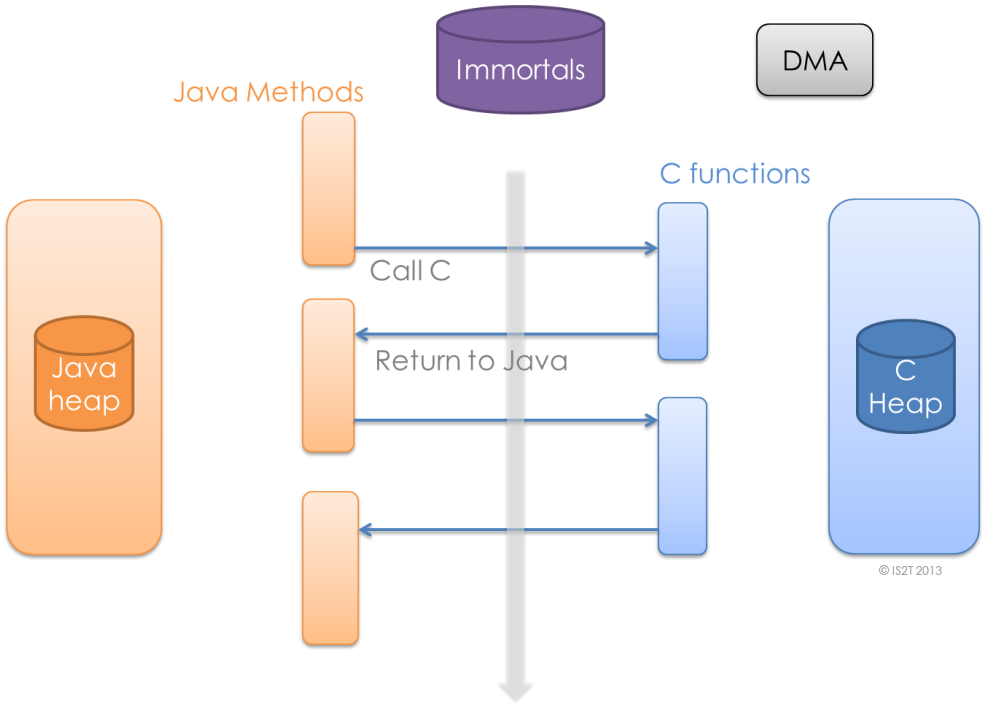
SNI

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.

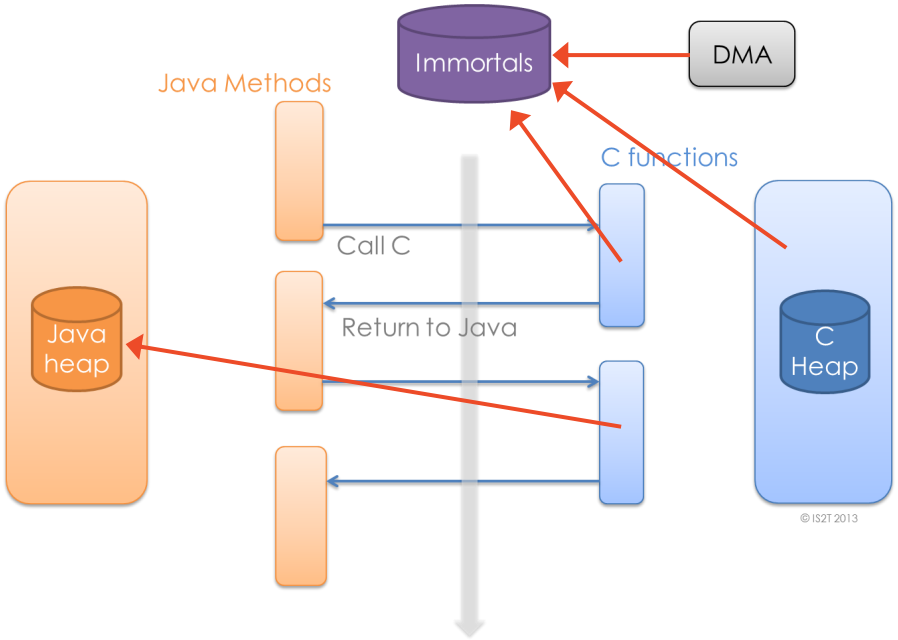
NAMING CONVENTION

```
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!");
    printHelloNative();
}
public static native void printHelloNative();
```

- 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.**

GET THE LINKER ERRORS

- In STM32CubeIDE, click on your project once to select it.
- Go to **Project** > **Build Project**.
- Wait for the end of the build. The following error appears:

```
C:\workspaces\HelloWorld\com.microej.training.Main\SOAR.o:(.text.soar+0x1f78):
undefined reference to `Java_com_microej_training_Main_printHelloNative'
```

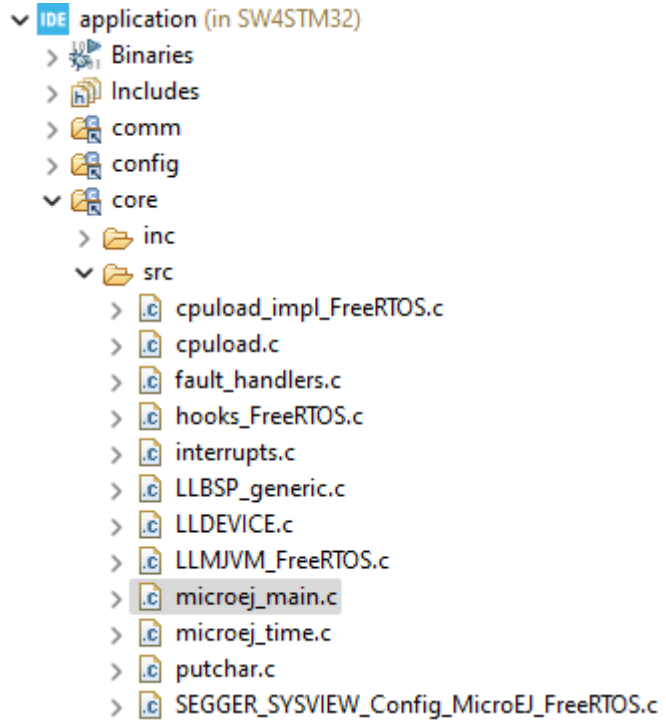
- The **printHelloNative()** method is a native method. **It must be implemented in the BSP.**

IMPLEMENT THE NATIVE METHOD IN THE BSP

- In STM32CubeIDE, open **microjvm_main.c**
- Implement the **printHelloNative()** method, use the method signature provided by the linker error:

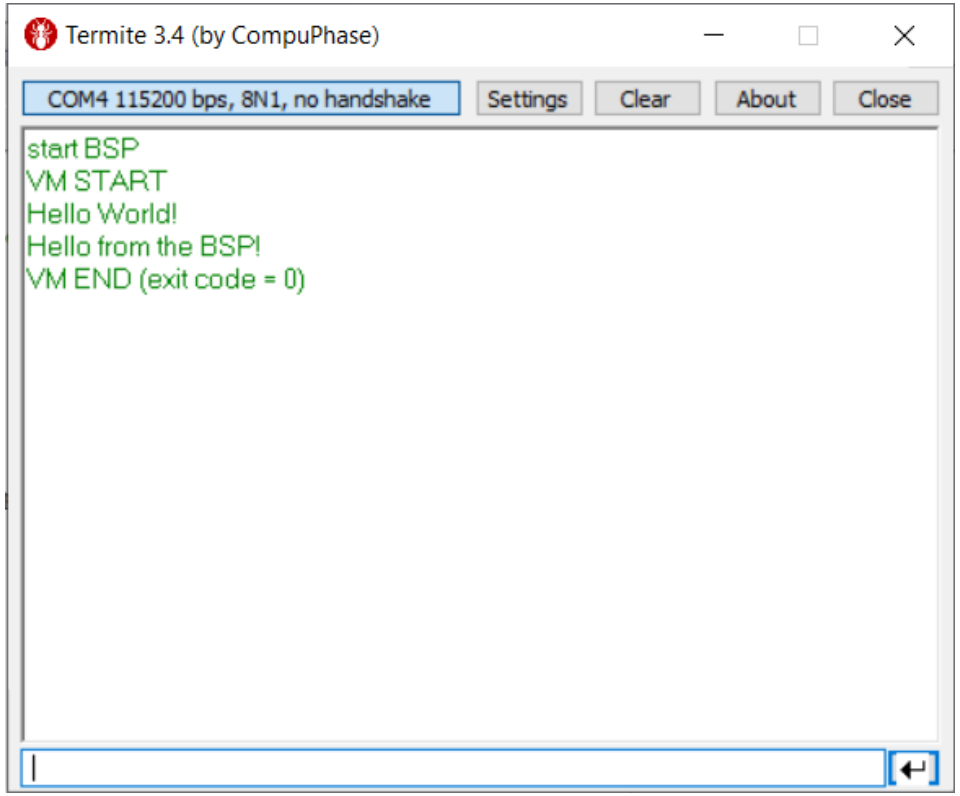
```
#include <stdio.h>
#include "microej_main.h"
#include "LLMJVM.h"
#include "sni.h"

void Java_com_microej_training_Main_printHelloNative(){
    printf("Hello from BSP!\n");
}
```

- Go to **Project** > **Build Project**.
- The build is successful.
- Flash the firmware:
    - **Run > Run Configurations > STM32 C/C++ Application > application_debug > Run.**



Run the example on Device

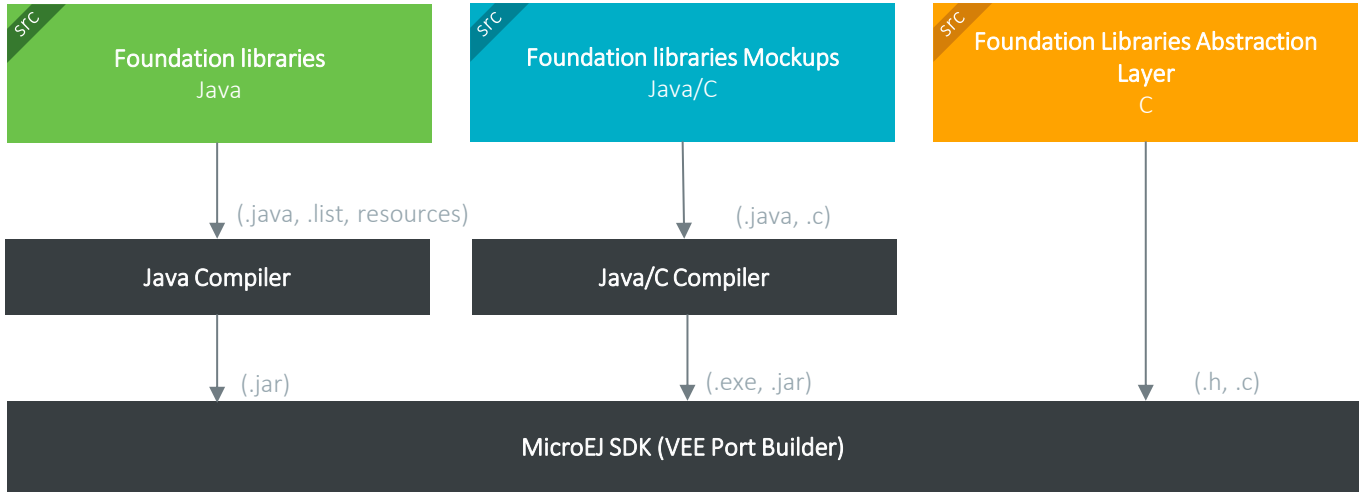
- Open the Termite serial terminal.
- Click the **Settings** button.
- Select the STM32F7508-DISCO board COM port.
- Reset the STM32F7508-DISCO board by pressing the **black** button near to the screen.
- The application starts and the **Hello World** and **Hello from BSP** messages are printed in the console!



Foundation Library

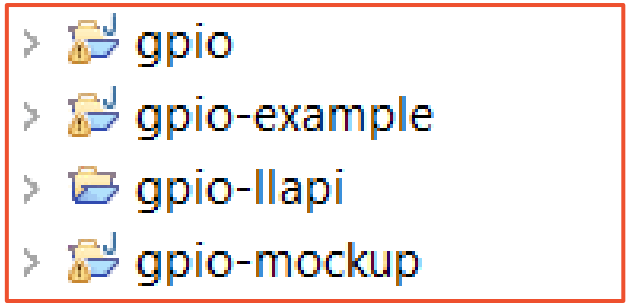
DEFINITION

- 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

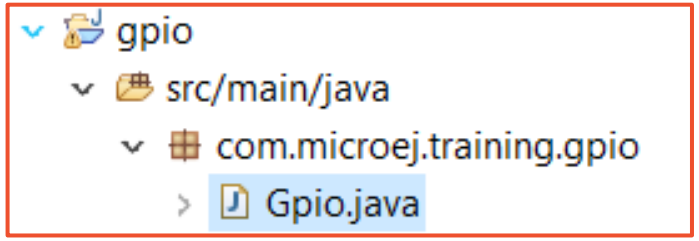
- Import the GPIO Foundation Library Example:
  - Open menu **File** > **Import...** > **General** > **Existing Projects into Workspace.**
  - Select the archive file **[training-package]/gpio_foundation_library_example-{version}.zip.**
  - Select all the projects.
  - Click on Finish.
- If some projects 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:

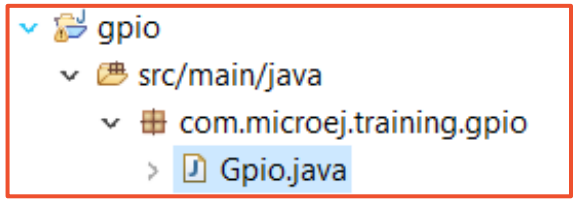
```
/**
 * GPIO management class.
 */
public class Gpio {
/**
 * Sets a value on the digital pin.
 *
 * @param pin
 *            the pin identifier
 * @param value
 *            digital pin value: true for high, false for low.
 */
native public static void set(int pin, boolean value);

/**
 * 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);
}
```

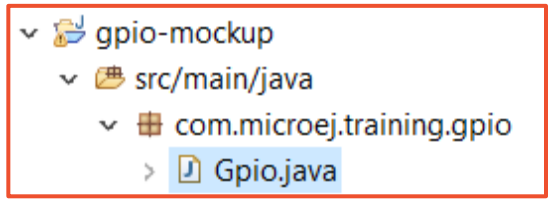


Run the Foundation Library Example on Simulator

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 **[platform]-[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 with the 3rd party IDE.

GET THE LINKER ERRORS

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

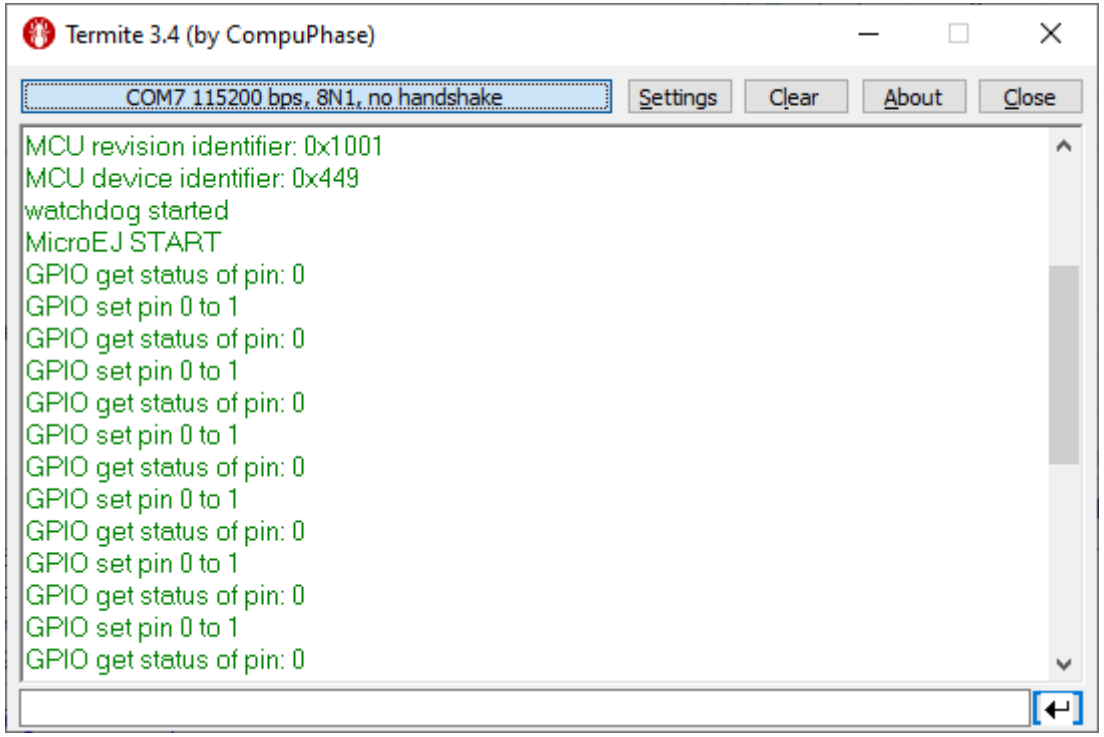
```
C:\XXX\com.microej.training.gpio.example.BlinkGpio\SOAR.o:(.text.soar+0x24dc): undefined reference to
`Java_com_microej_training_gpio_Gpio_get’

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"

jint 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

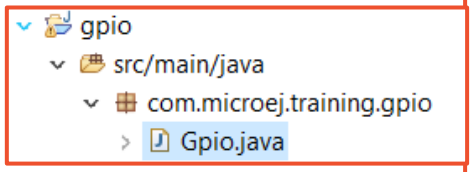
- Build the **gpio-example** project for the device:
  - Go to **Run -> Run Configurations.**
  - Select the **gpio-example BlinkGpio** Run Configuration.
  - Go to **Execution Tab.**
  - Select **Execute on Device.**
  - Click **Run.**
- Compile, Link and Flash with the 3rd party IDE.
- Open the Termite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



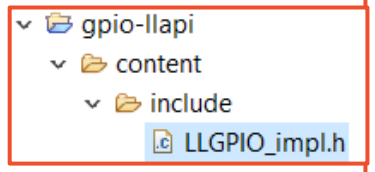
ABSTRACTION LAYER INTERFACE: LLAPI

- 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 **stm32f7508_freertos-bsp/** project.
- Add the **LLGPIO_impl.h** header file to the compiler path in the 3rd party IDE.
- Implement the **LLGPIO_get(int32_t pin)** and **LLGPIO_set(int32_t pin, uint8_t state)** functions in the BSP.

```
static uint8_t GPIO_initialized = 0;

static void LLGPIO_initialize(void)
{
    if(!GPIO_initialized)
    {
        GPIO_initialized = 1;
        __GPIOI_CLK_ENABLE();
        GPIO_InitTypeDef GPIO_InitStruct;

        /* Configure LED pin as output */
        GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
        GPIO_InitStruct.Pull = GPIO_NOPULL;
        GPIO_InitStruct.Speed = GPIO_SPEED_LOW;
        GPIO_InitStruct.Pin = GPIO_PIN_1;

        HAL_GPIO_Init(GPIOI, &GPIO_InitStruct);
    }
}
```

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

    if( state == JFALSE)
    {
        value = GPIO_PIN_RESET;
    }
    else
    {
        value = GPIO_PIN_SET;
    }

    HAL_GPIO_WritePin(GPIOI, GPIO_PIN_1, value);
}

uint8_t LLGPIO_get(int32_t pin)
{
    LLGPIO_initialize();
    GPIO_PinState state = HAL_GPIO_ReadPin(GPIOI, GPIO_PIN_1);

    return (state == GPIO_PIN_RESET ? JFALSE : JTRUE);
}
```

Implementation available in LLGPIO_STM32F7508-{version}.zip

Packaging and Tests

PACKAGING AND TESTS

BUILD A LIBRARY WITH MICROEJ MODULE MANAGER (MMM)

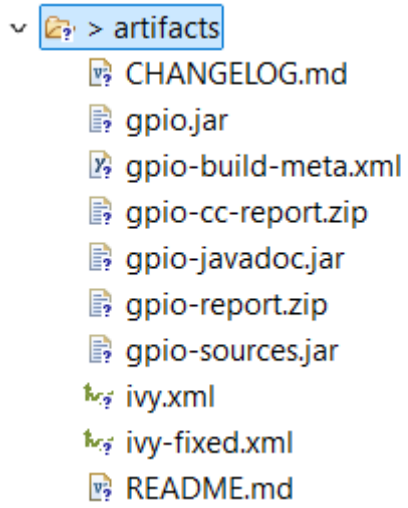
- Generate a JAR file with the classfiles.
- Generate a zip file with the sources.
- Generate the Javadoc.
- Execute the tests (defined in src/test/java folder).
- Publish the library in an MMM repository.

Configure the Testsuite

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 MMM Build

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

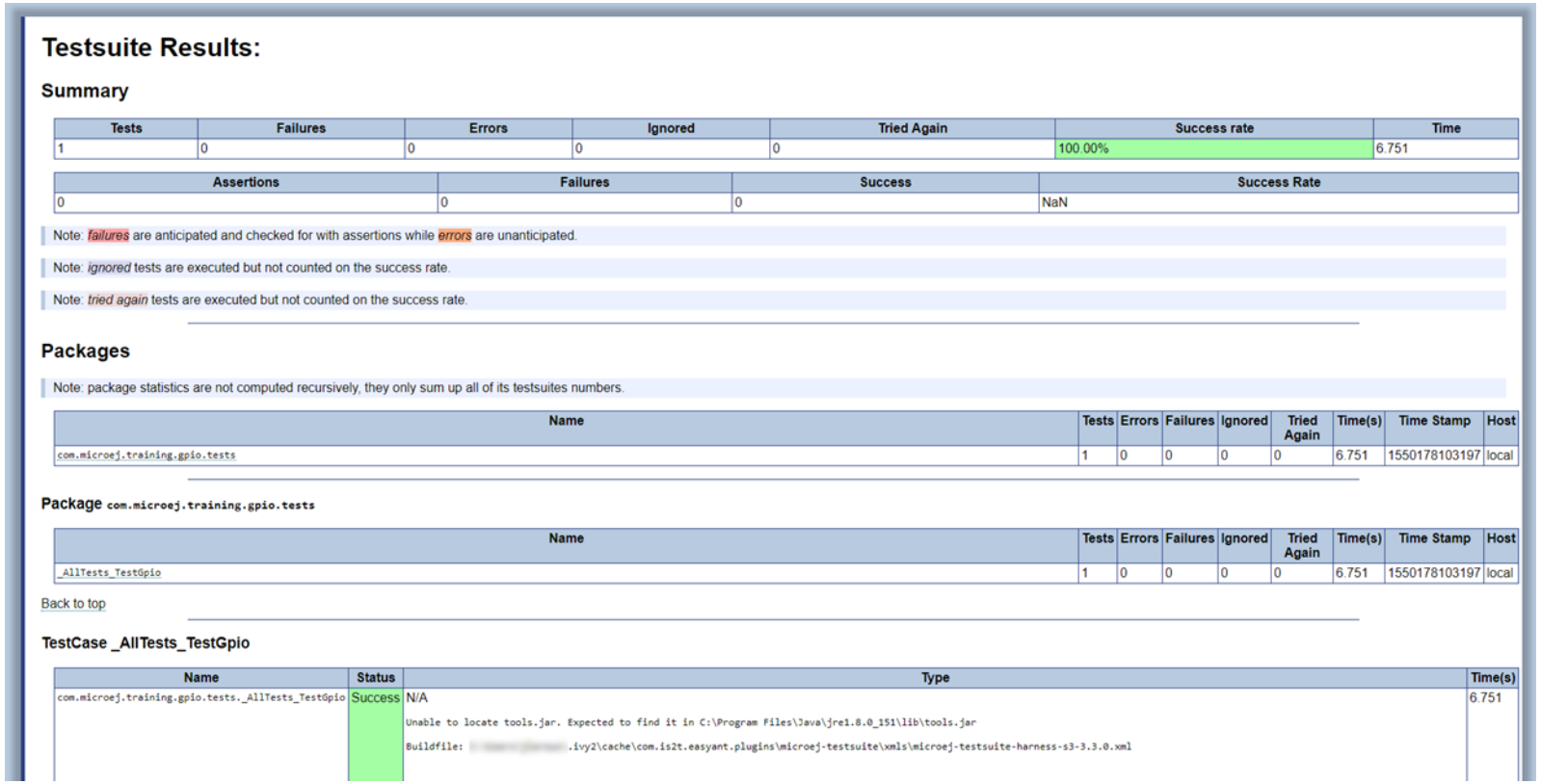


- Build result is published in a local MMM repository:

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

Tests Result

Testsuite report is available in the **target~/artifacts/myfoundation-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

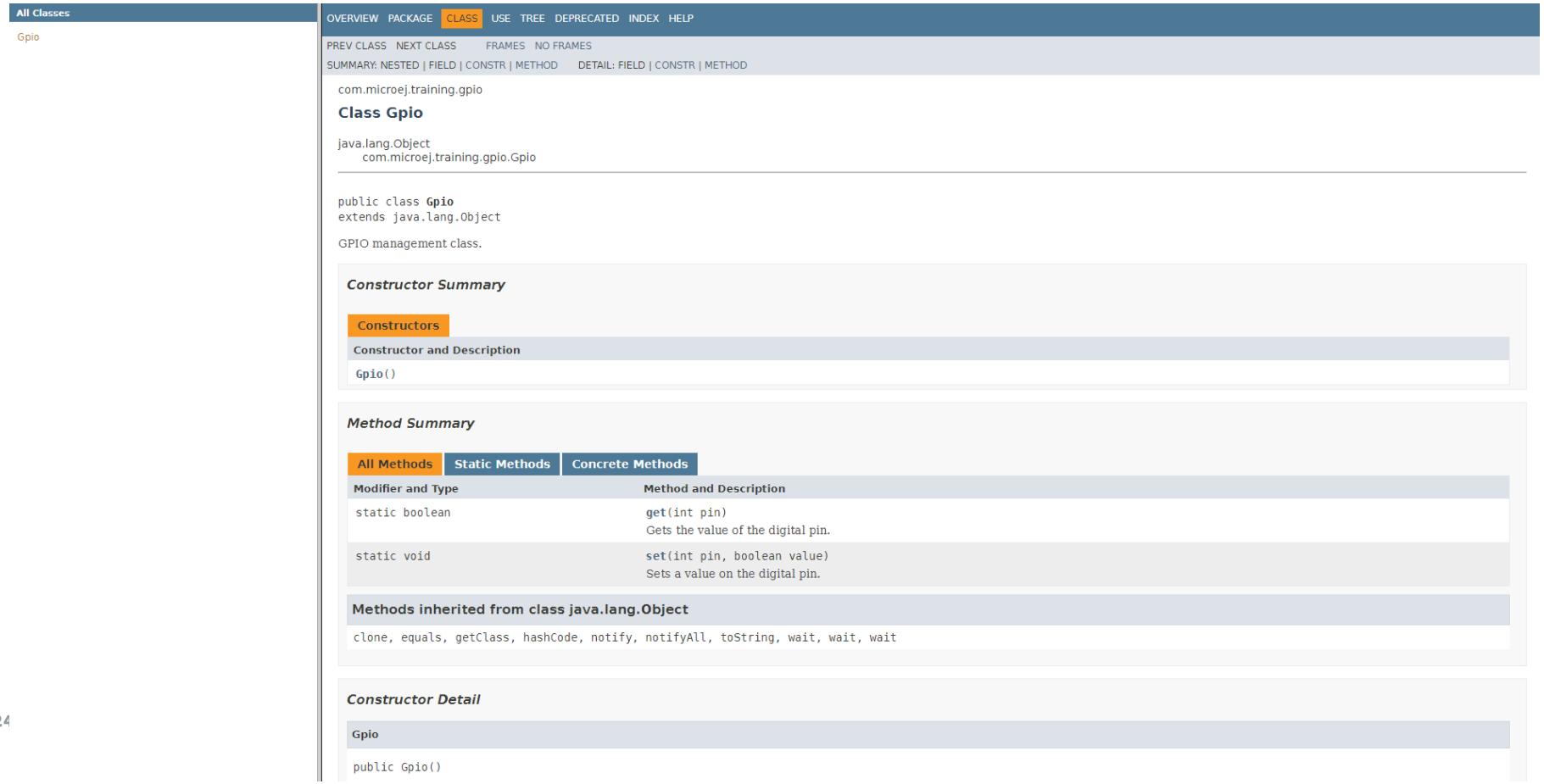
Back to top

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()
```

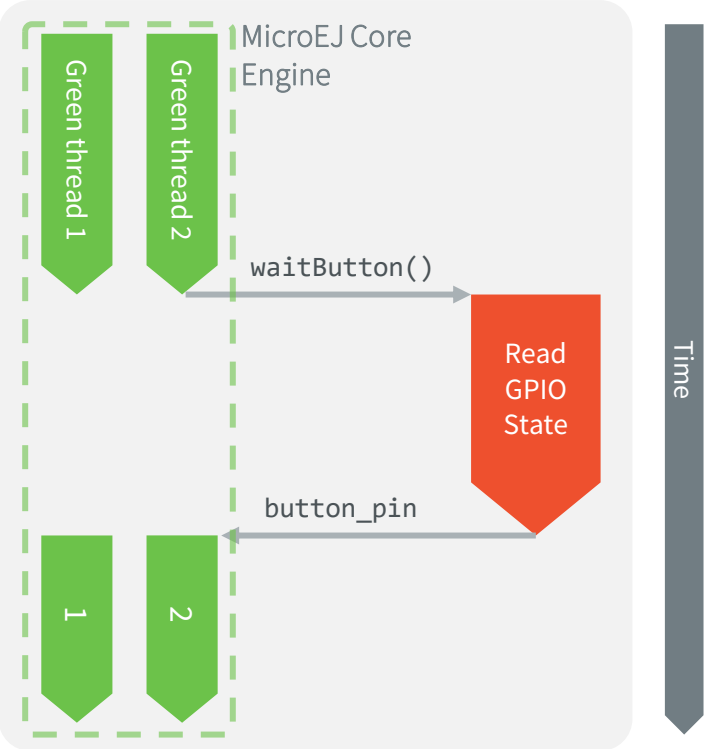
SNI

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

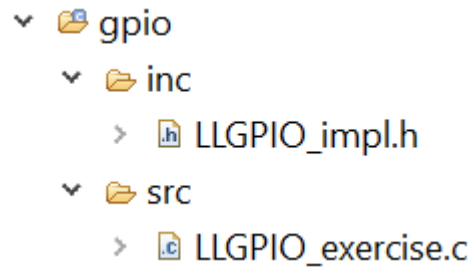
- The code of the **gpio-exercise** project does the following actions:
  - Wait for a button event and prints the index of the pressed button (User/Blue button)
  - Toggles the board LED1 each 500ms
  - Each action in performed in a dedicated thread

```
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

- **Important note:**
  - This exercise uses the User button (blue button) of the STM32F7508 board to demonstrate how to implement a blocking Java native method without blocking the execution of other Java threads.
  - The STM32F7508 platform already implements the Button events management in the MicroUI stack (see button_manager.c)
  - It is recommended to use the MicroUI library to get button events in the application code (e.g. MicroUI Input Example)
  - The next slides are showing how to run the exercise with STM32CubeIDE, it is also possible to use IAR.
- A C implementation is provided in the **LLGPIO_STM32F7508-{version}.zip** package.
- Add **LLGPIO_exercise.c** to the BSP project:
  - Copy / Paste **LLGPIO_exercise.c** in the **stm32f7508_freertos-bsp\projects\microej\gpio\src** folder.
  - Remove the previous **LLGPIO.c** implementation.
  - **LLGPIO_exercise.c** redefines an interrupt handler defined in the **ui** folder. The folder needs to be excluded from BSP build to run this sample.
  - Exclude the **ui** folder from the BSP build:
    - In STM32CubeIDE, right-click on the **ui** folder
    - Click on **Properties**
    - Click on **Exclude resource from build**



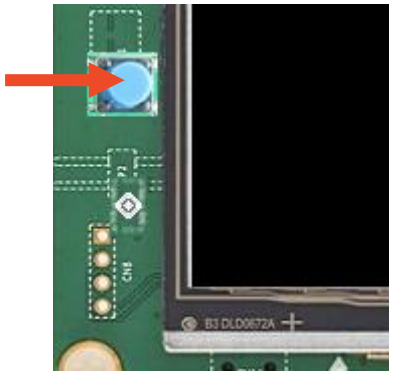
gpio/ folder in STM32CubeIDE

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 **STM32F7508** platform
  - Select **Execute on Device**
  - Click **Run**
- In STM32CubeIDE, click on your project once to select it.
  - Go to **Project** > **Build Project.**
  - Wait for the end of the build.
- Plug the STM32F7508-DK board to the PC
- In STM32CubeIDE, select **Run > Run Configurations > STM32 C/C++ Application > application_debug > Run**.

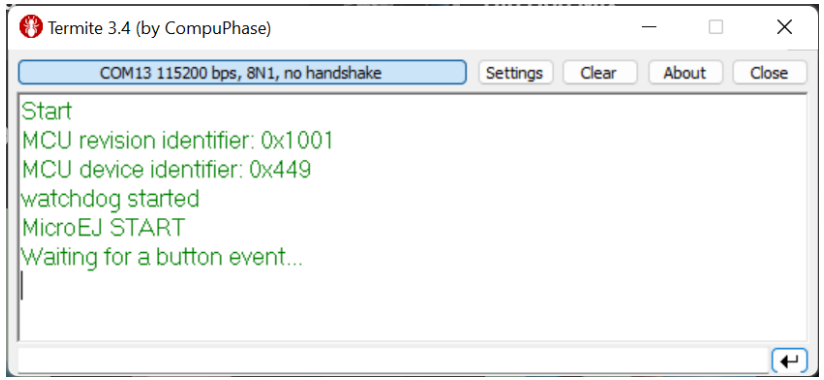
RUN THE EXERCISE CODE (2/2)

- Open the Termite serial terminal.
- Reset the STM32F7508-DK board by pressing the **black** button near to the screen.
- The application starts and waits for a button event.

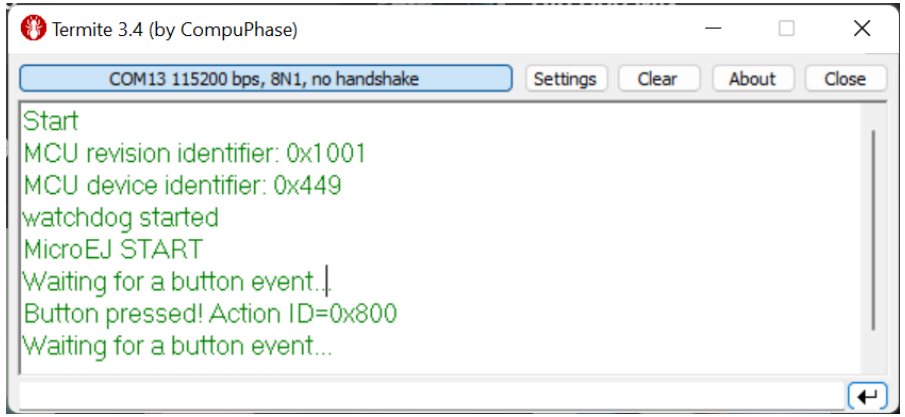


- **LED1 is not blinking each 500ms as expected.**
  The waitButton() native blocks the execution of the other Java threads.

- 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



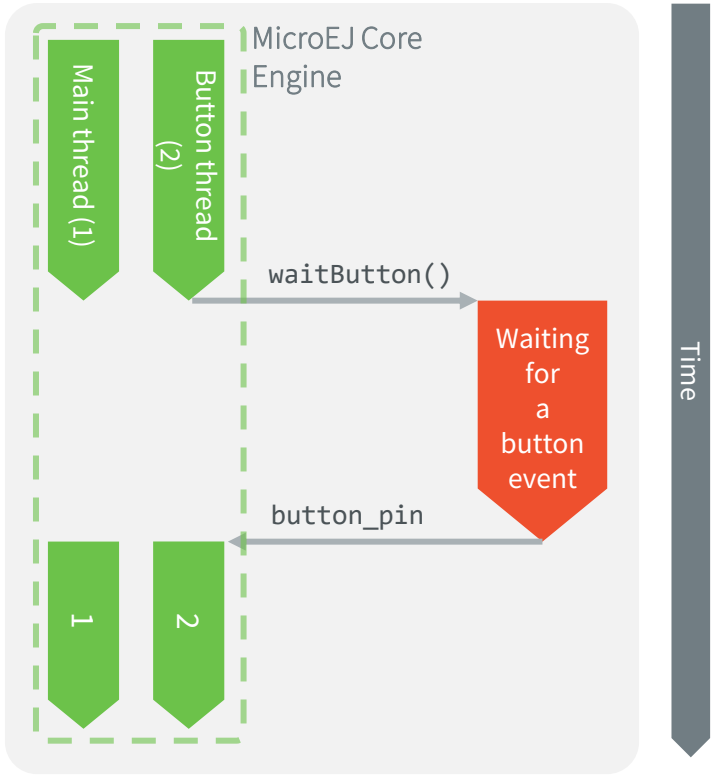
Traces when the application starts



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

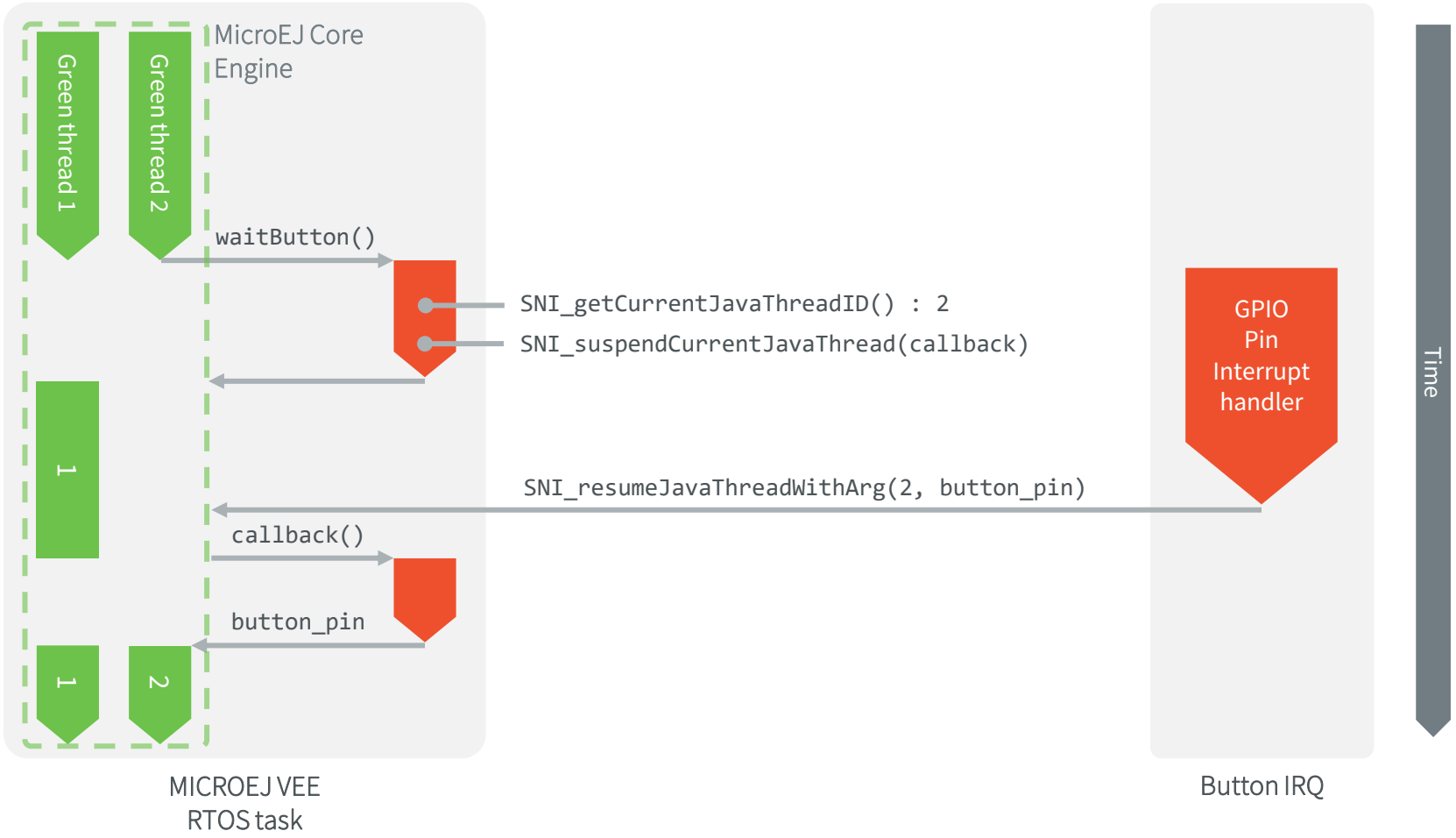
Implement a blocking Java native method without blocking the execution of other Java threads.

HAND'S ON DIRECTIVES

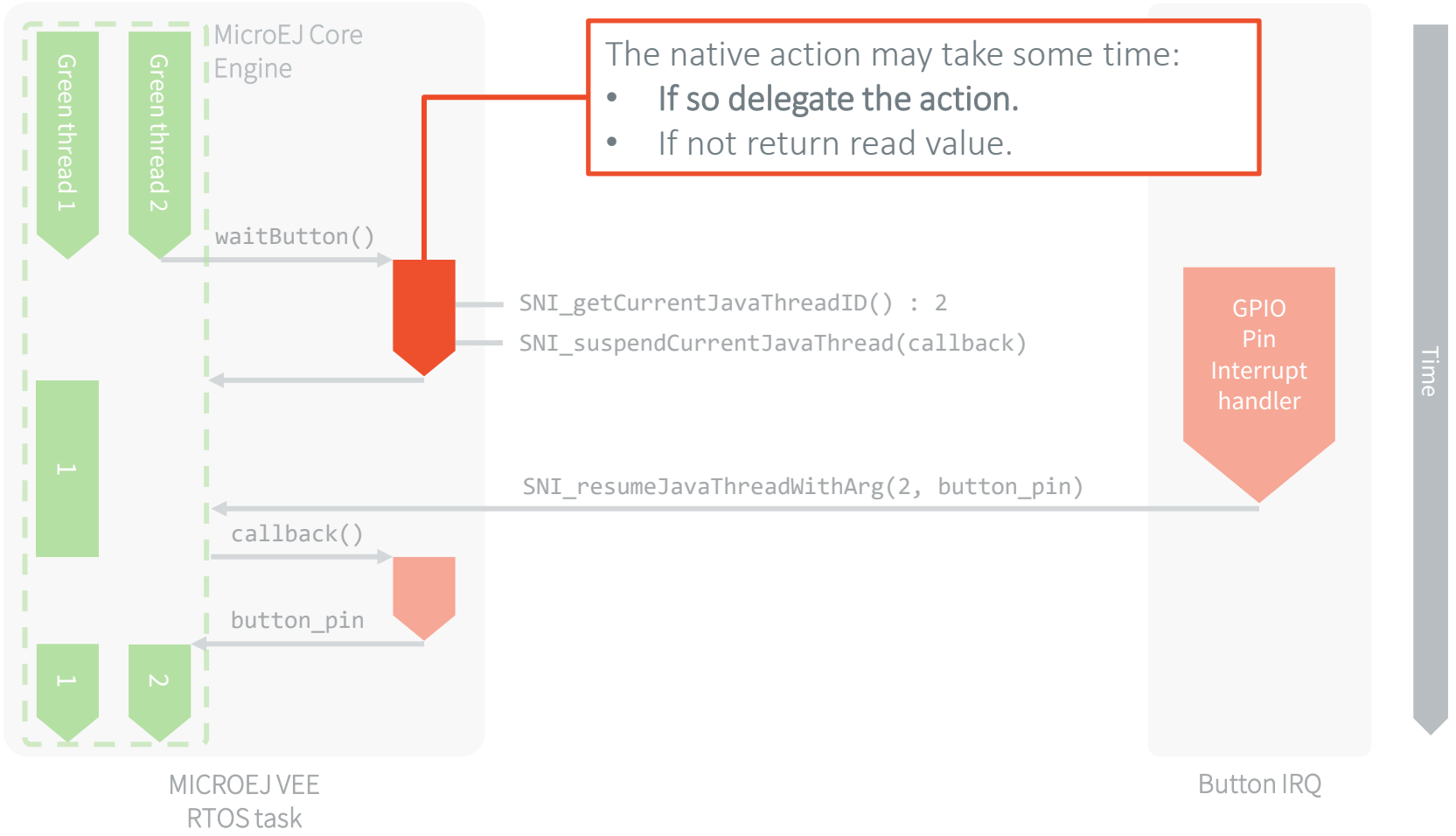
- Only the C code should be updated

- 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

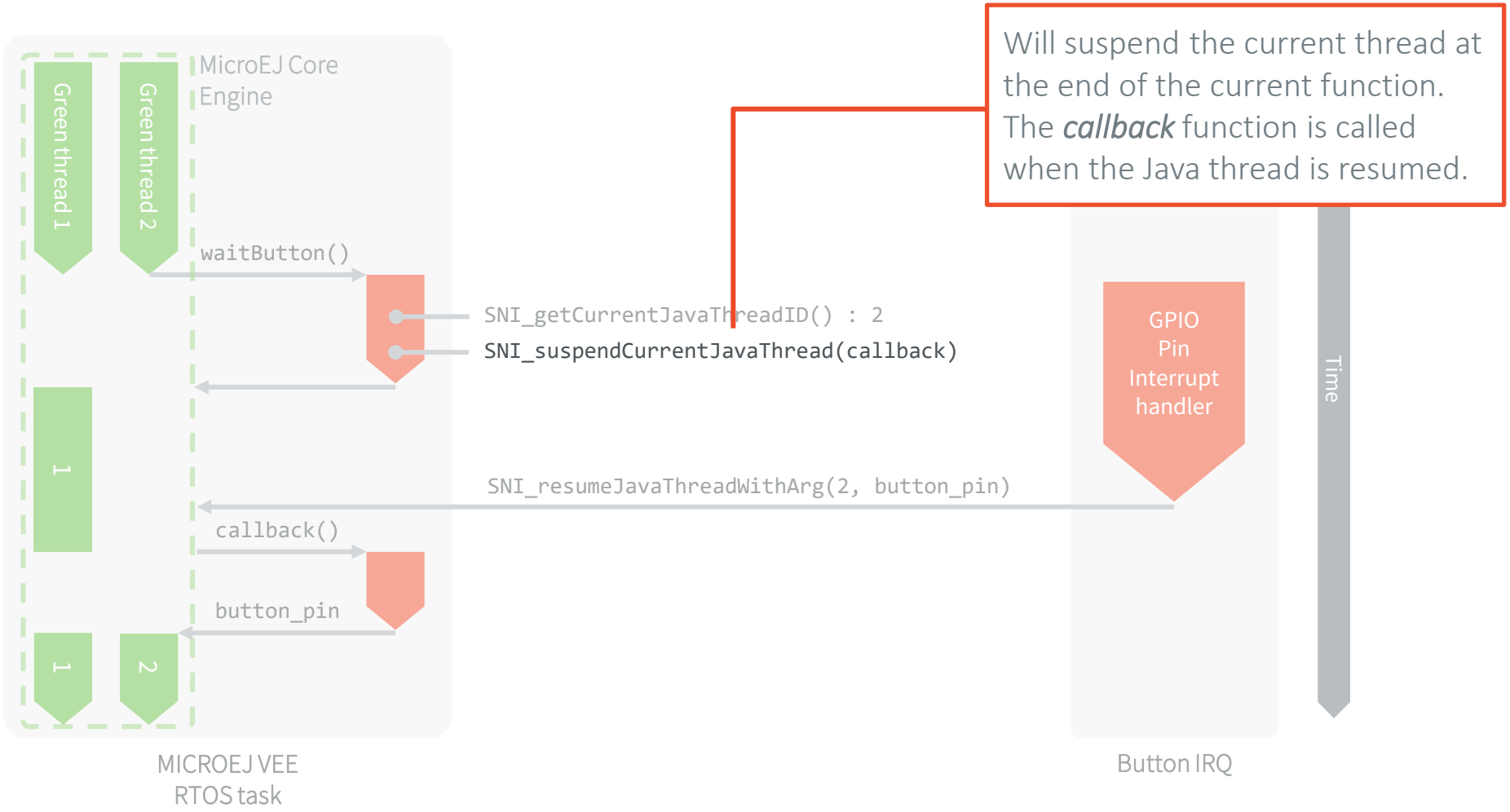
THREAD SYNCHRONIZATION: CALLBACK PATTERN



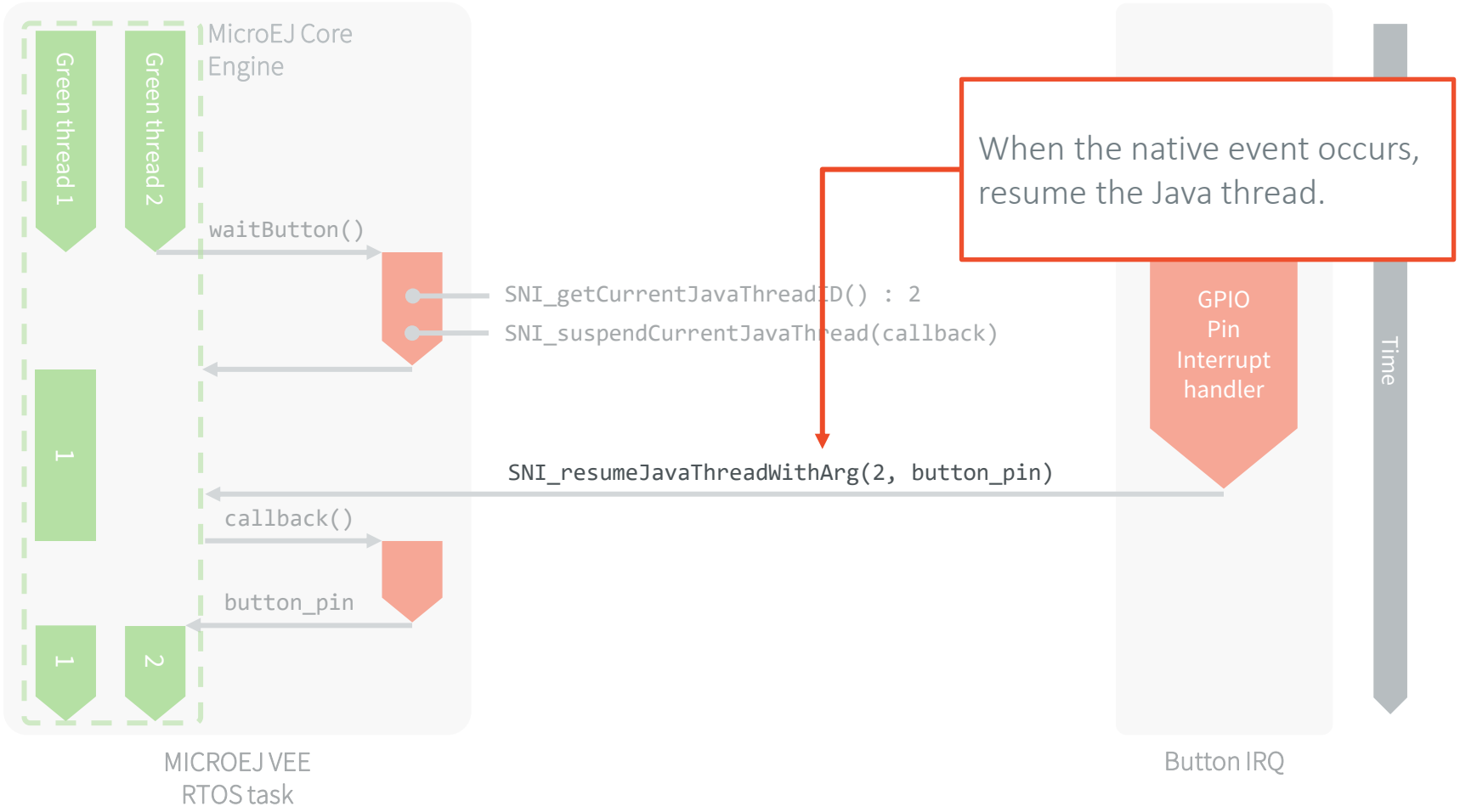
THREAD SYNCHRONIZATION: CALLBACK PATTERN



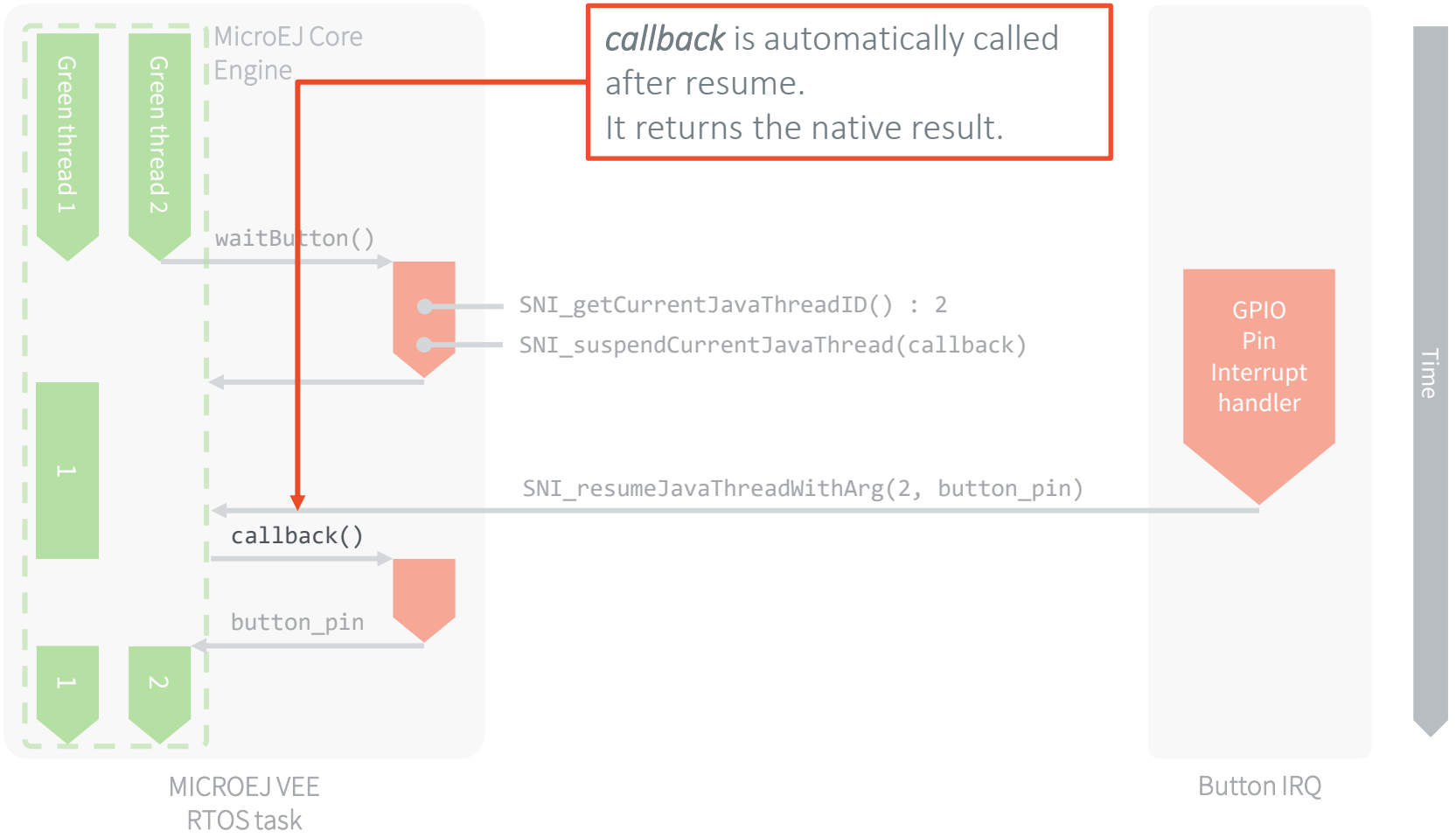
THREAD SYNCHRONIZATION: CALLBACK PATTERN



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()
{
  // Initialize the GPIOs
  LLGPIO_initialize();

  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 int32_t button_index;

void BUTTON_MANAGER_interrupt(void)
{
        uint32_t intStat = GPIO_PortGetInterruptStatus(GPIO, GPIO_PORT, kGPIO_InterruptA);
         if (intStat & (1UL << BUTTON_INTERRUPT_PIN))
         {
                  GPIO_PortClearInterruptFlags(GPIO, GPIO_PORT, kGPIO_InterruptA,
                                               (1UL << BUTTON_INTERRUPT_PIN));

                button_index = (int32_t)BUTTON_INTERRUPT_PIN;
                SNI_resumeJavaThreadWithArg(java_thread_id, (void*)&button_index);
         }
}
```

STEP 3: IMPLEMENT THE CALLBACK FUNCTION

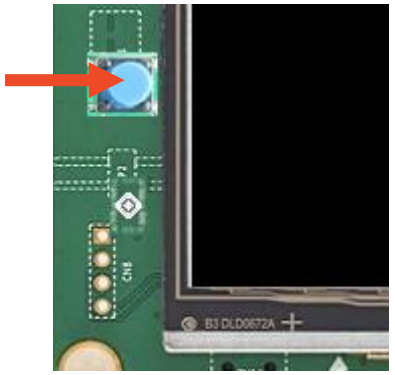
- The callback function must have the same signature as the SNI native (same parameters and return type): **jint waitButton_callback()**

- 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.

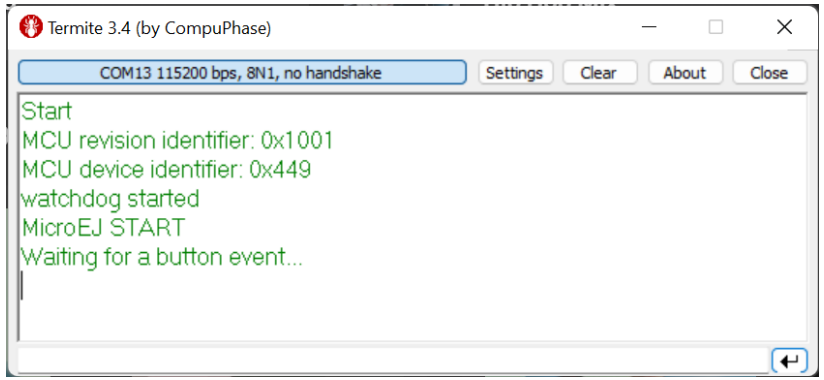
```
static jint waitButton_callback(){
  int32_t *button_index_addr;
  SNI_getCallbackArgs(NULL, (void**)&button_index_addr);
  return (jint)*button_index_addr; // Actual value returned to Java
}
```

RUN THE UPDATED CODE

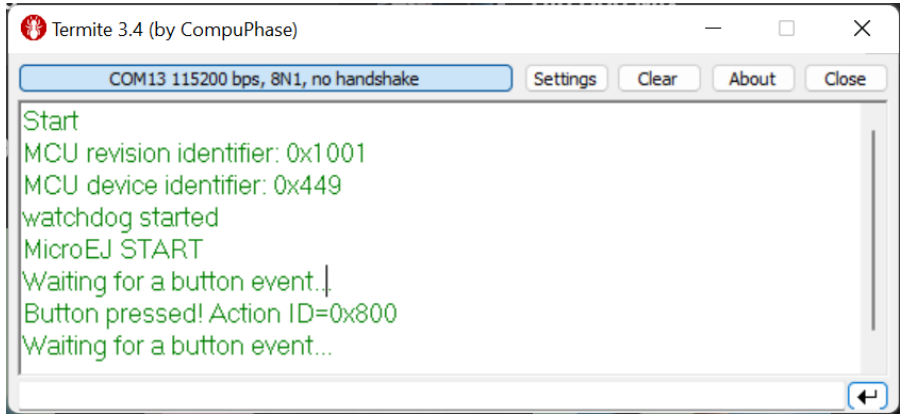
- Open the Termite serial terminal.
- Reset the STM32F7508-DK board by pressing the **black** button near to the screen.
- The application starts and waits for a button event.



- **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

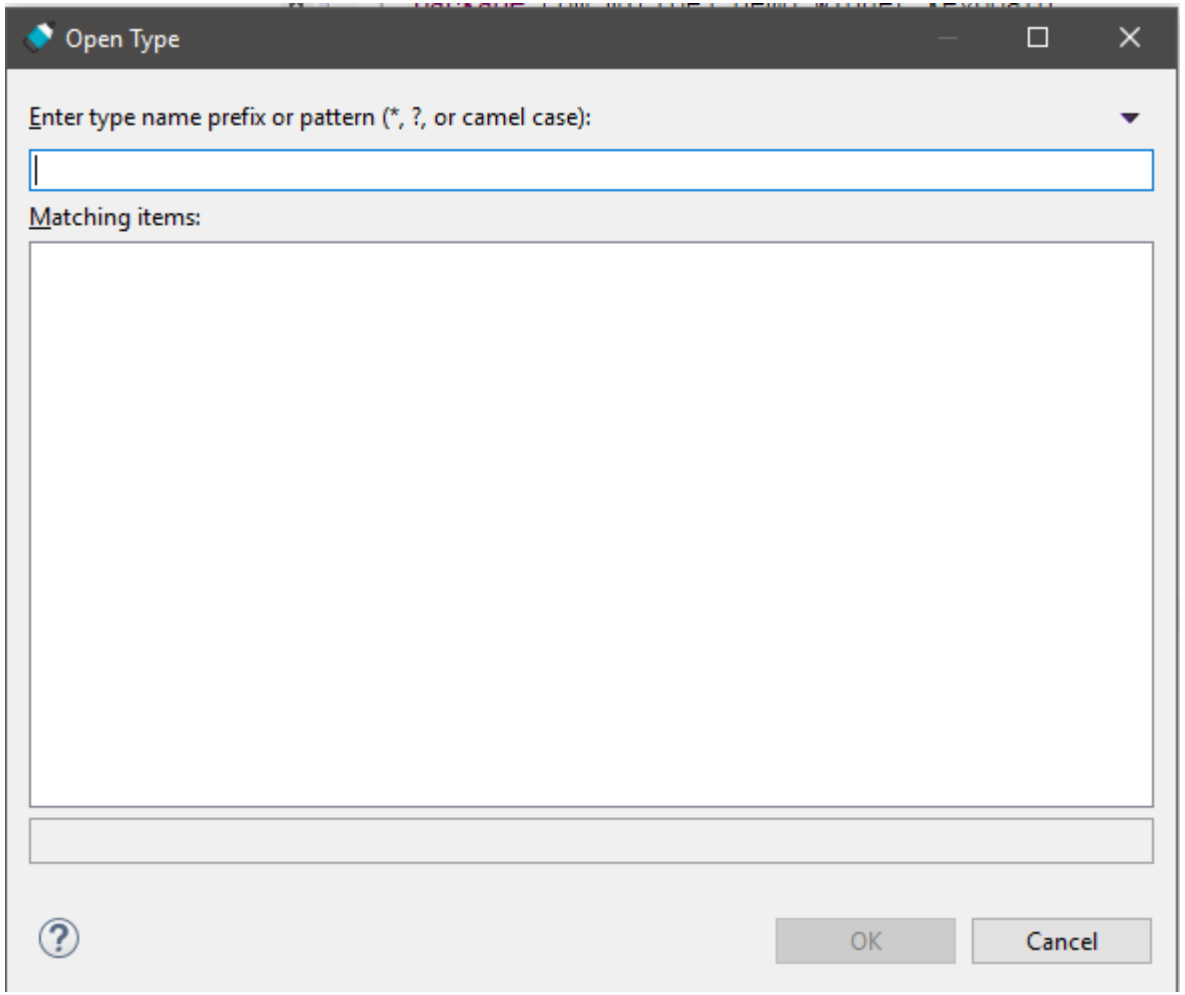
MICROEJ SDK / Studio

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.

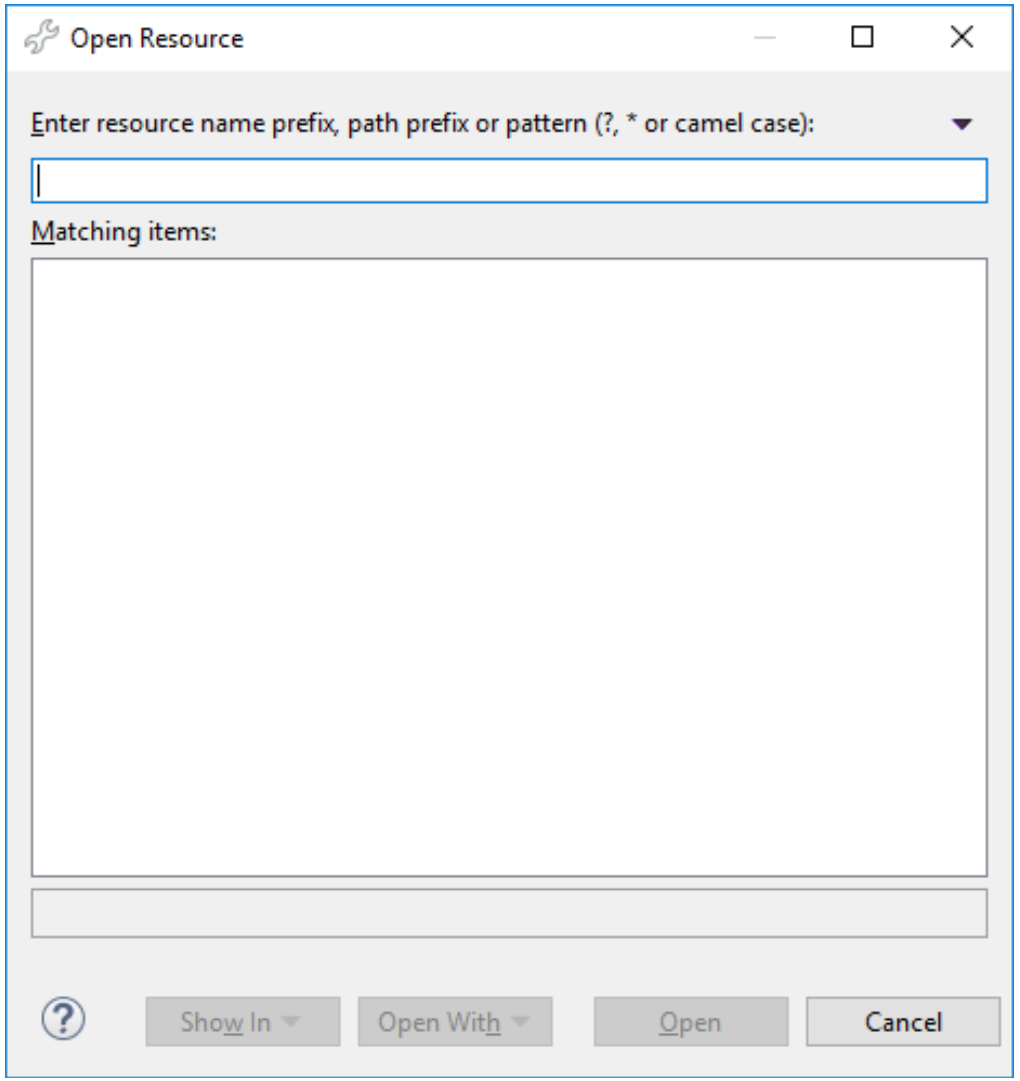
SHORTCUTS

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



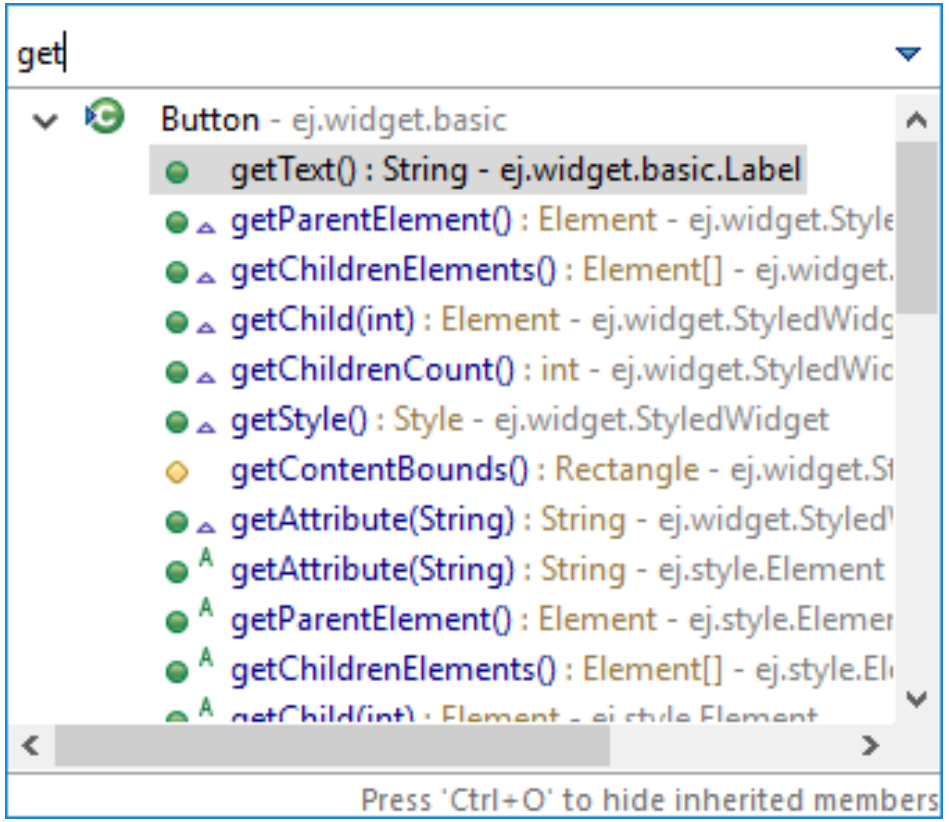
SHORTCUTS

- **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



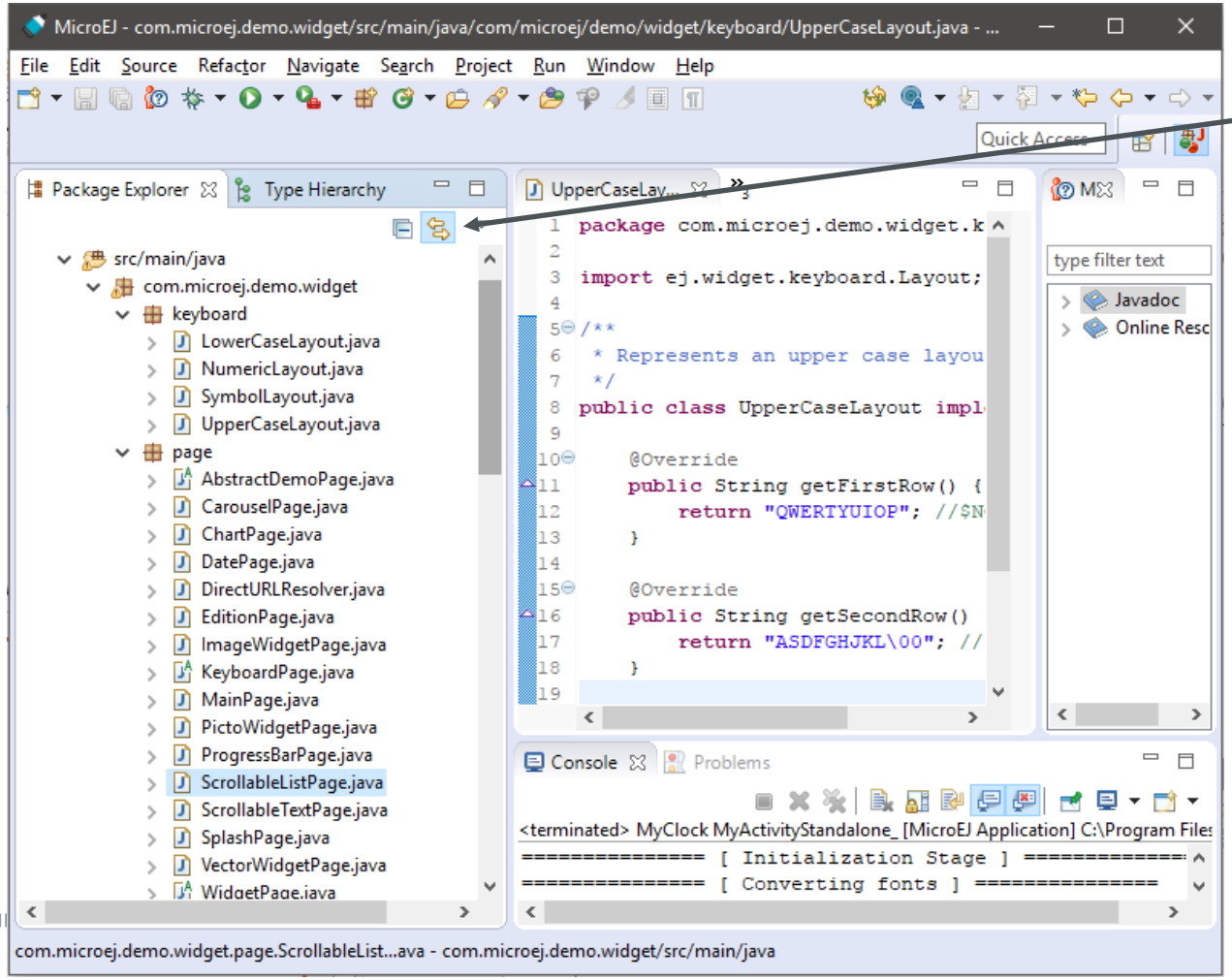
SHORTCUTS

- **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

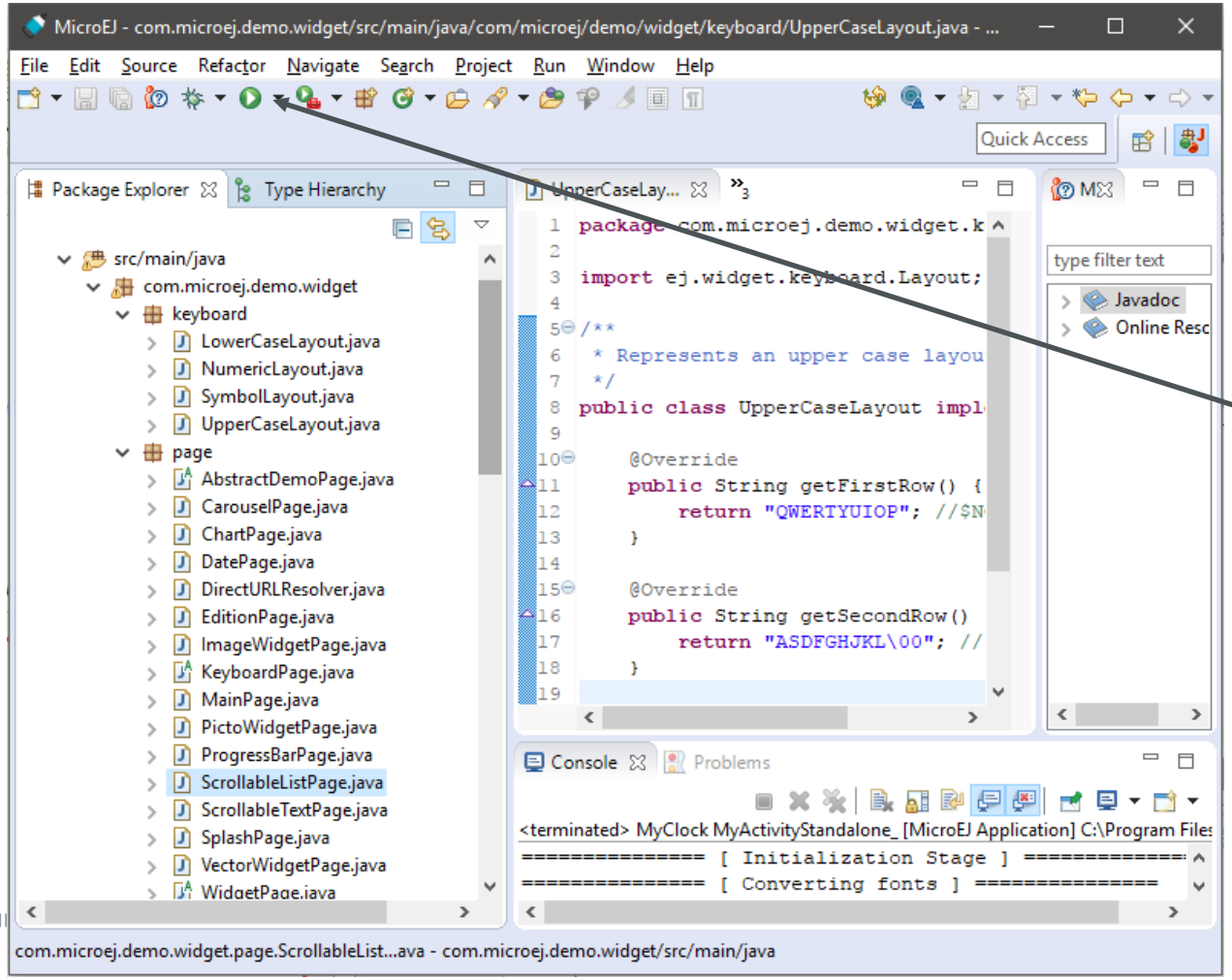
- **CTRL + I**
  - Correct indentation
- **ALT + Shift + R on a class / method / field**
  - Rename
- **CTRL + F**
  - Search in file
- **CTRL + H**
  - Search plugin of Eclipse

IDE



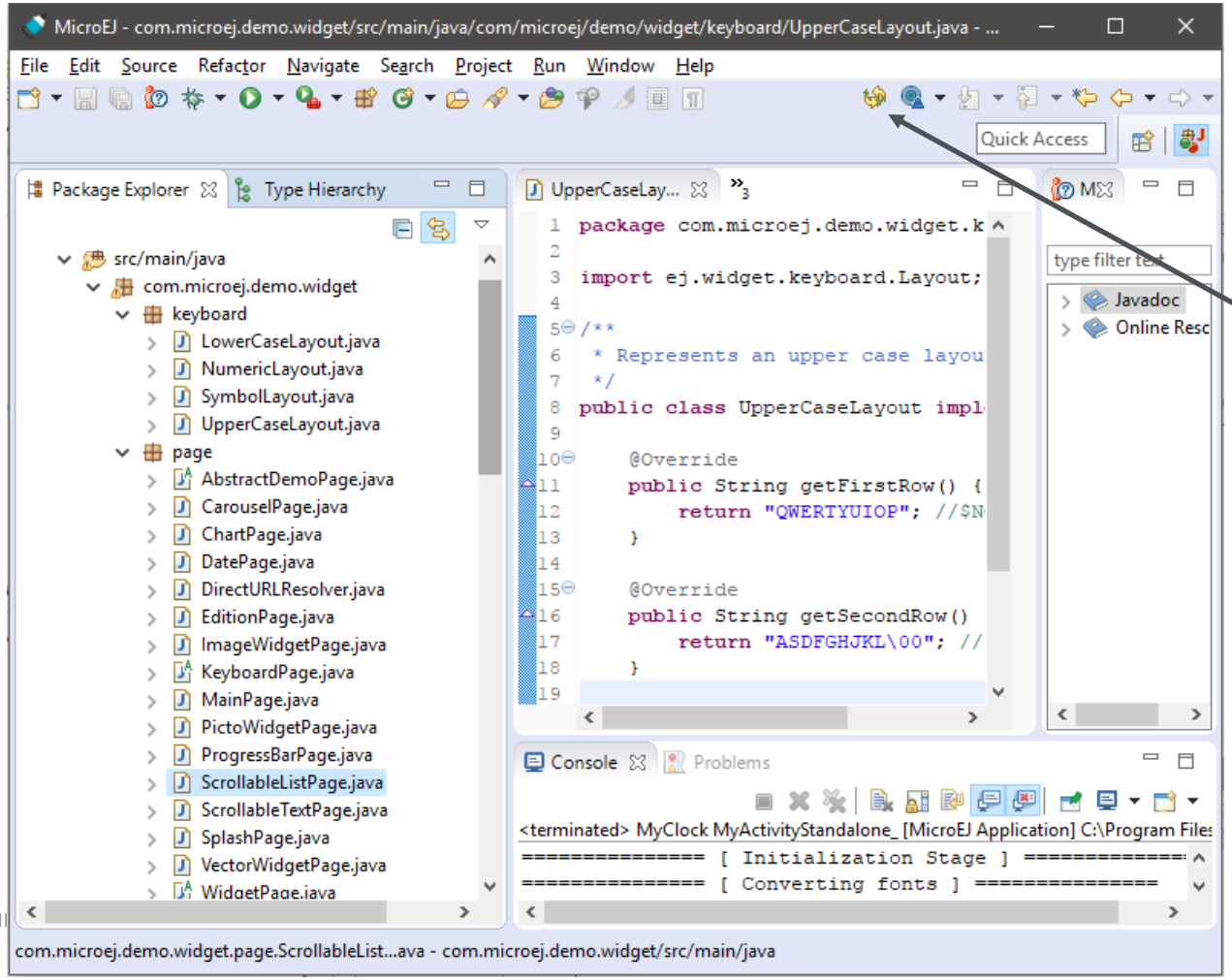
Link the explorer and the editor

IDE



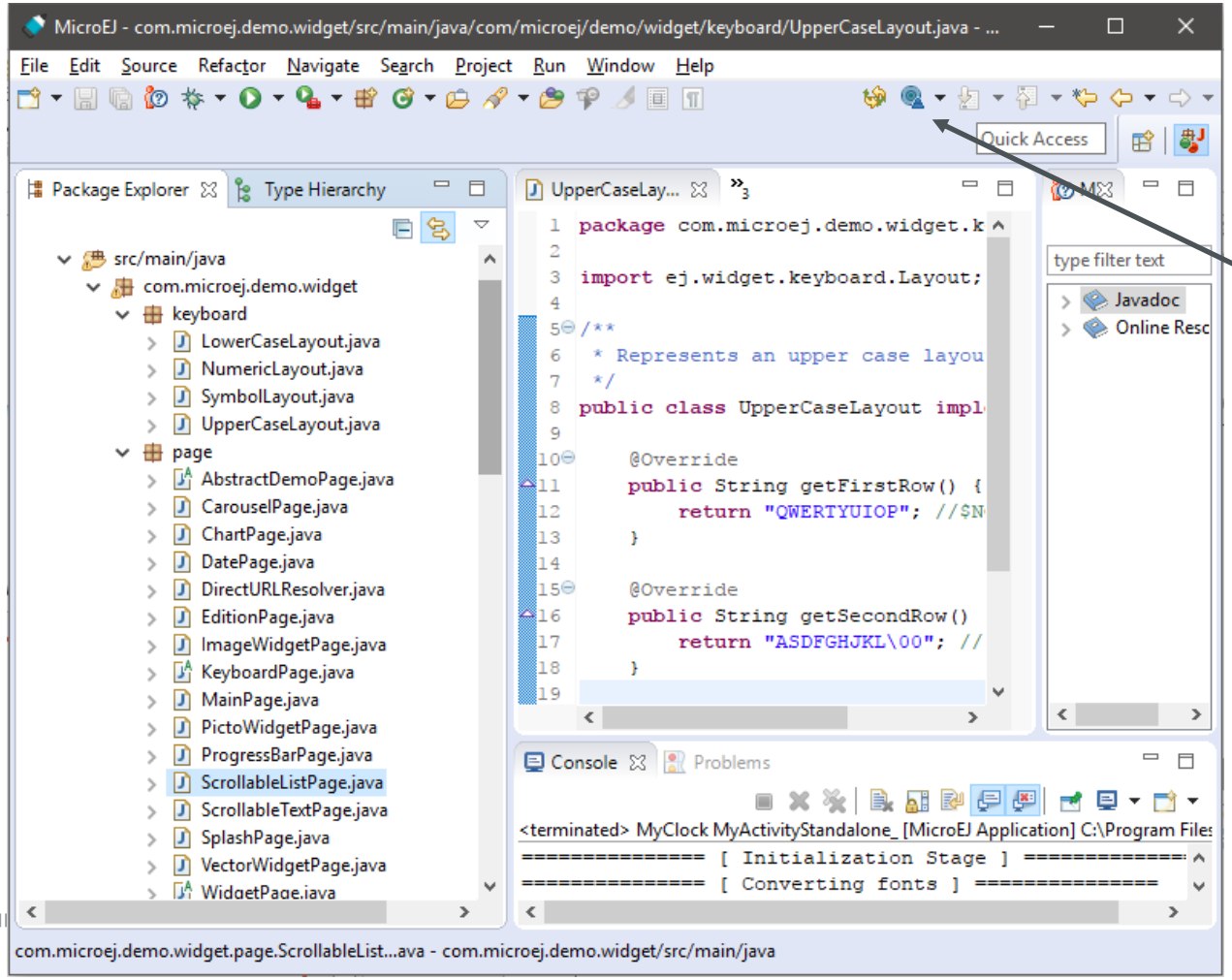
Re-Run the last
Run configuration

IDE



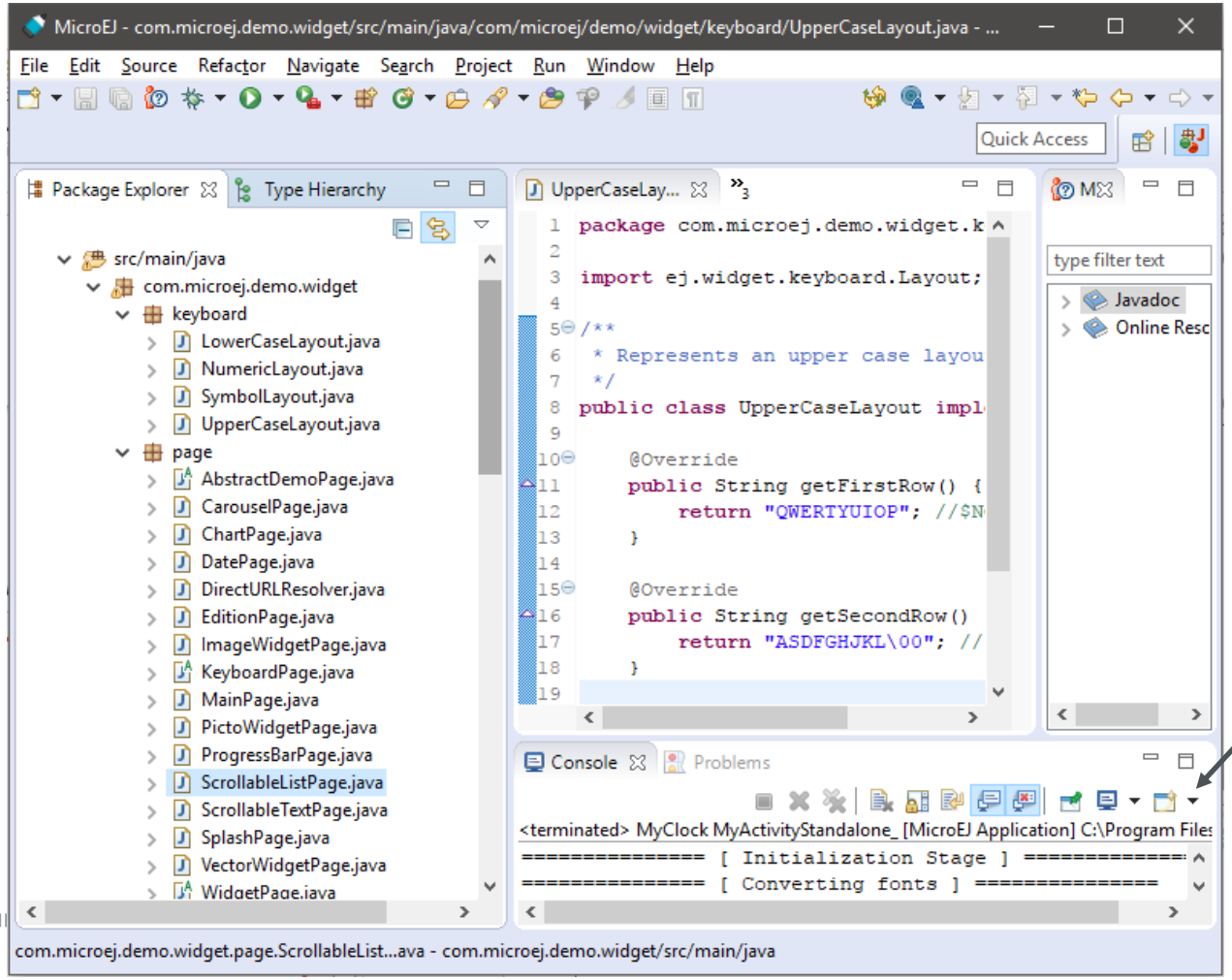
Resolve all MMM projects

IDE



Build selected module

IDE



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

THANK YOU

for your attention !

MICROEJ®