C / Managed Code Communication

With MicroEJ VEE

MICROEJ®

DISCLAIMER

CHAPTERS

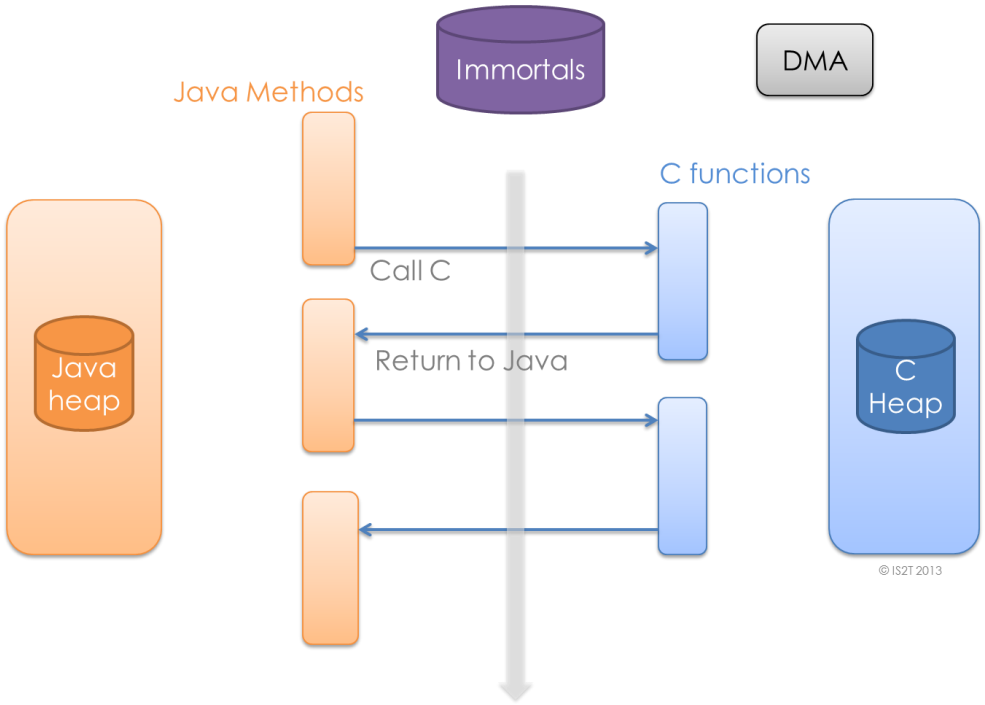
SNI

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.

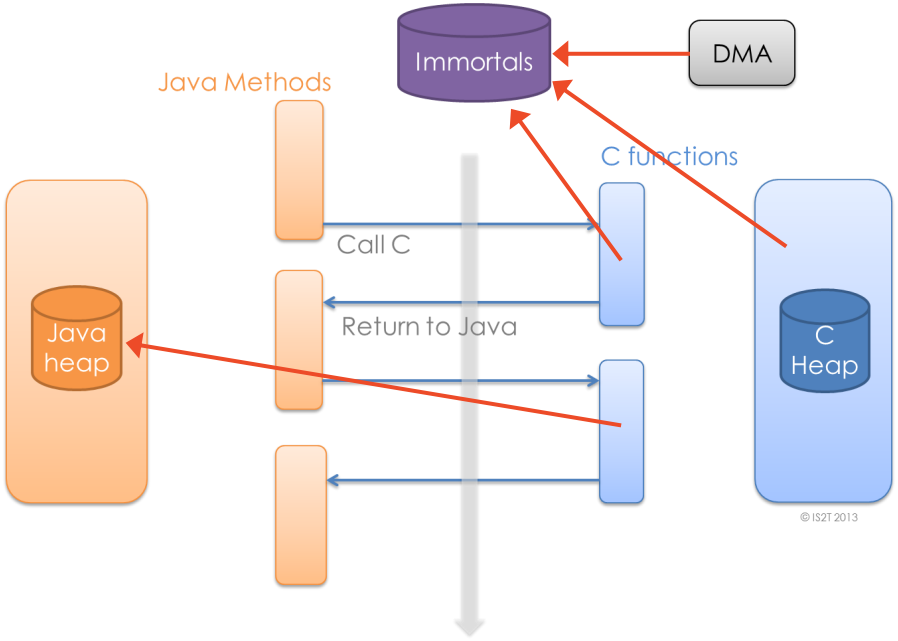
SNI – NAMING CONVENTION

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

 public static void main(String[] args) {
   int i = print(3);
 }

 public static native int print(int i);
}
```

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

jint Java_com_corp_examples_Hello_print(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.



SNI.H

- See `sni.h` header file for a full description of the SNI C API
- Located under `<vee_port>/source/include`
- **`#include <sni.h>`**

```
/**
 * Checks if the given Java array is immortal.
 *
 * An immortal Java array remains at a fixed memory location and will not be moved or changed by
 * Java's garbage collection routine.
 *
 * An immortal Java array can be used safely from another native task, an interrupt
 * handler or can be used as a DMA buffer.
 *
 * A non-immortal Java array can be used only during the execution of an SNI native or an
 * SNI callback.
 *
 * A reference to a non-immortal Java array must not be kept in C between two SNI native executions.
 *
 * @param javaArray is a Java array retrieved from an SNI native
 * arguments.
 *
 * @return <code>true</code> if the given Java array is immortal or null, otherwise
 * returns <code>false</code>.
 */
bool SNI_isImmortalArray(void* javaArray);

/**
 * Throws a new <code>ej.sni.NativeException</code> after the end of the current native
 * method or SNI callback. <code>NativeException</code> class is a subclass of
 * <code>java.lang.RuntimeException</code>. <code>NativeException</code> instances are
 * unchecked exceptions, they do not need to be declared in the native method throws
 * clause.
 *
 * The virtual machine keeps a reference to the message until the end of the current native
 * and does a copy of the message only when the current native returns. The message pointer
 * must not reference a memory allocated locally in the stack.
 *
 * If the current Java thread is suspended this function returns <code>SNI_ERROR</code>
 * and no exception is thrown.
 *
 * Calling this function while an exception is already pending will replace the previous
 * exception.
 *
 * If there is not enough space in the Java heap to allocate the exception, then an
 * <code>OutOfMemoryError</code> will be thrown in Java.
 *
 * @param errorCode value that can be retrieved in Java using the
 * <code>NativeException.getErrorCode()</code> method.
 *
 * @param message is a null-terminated string that will be used to
 * generate the message returned by <code>NativeException.getMessage()</code>. May be
 * null.
 *
 * @return <code>SNI_OK</code> on success. Returns <code>SNI_ERROR</code> if this function
 * is not called within the virtual machine task or if the current thread is suspended.
 */
int32_t SNI_throwNativeException(int32_t errorCode, const char* message);

/**
 * Throws a new <code>ej.sni.NativeIOException</code> after the end of the current native
 * method or SNI callback. <code>NativeIOException</code> class is a subclass of
 * <code>java.io.IOException</code>. <code>NativeIOException</code> instances are checked
 * exceptions, they need to be declared in the native method throws clause (e.g.,
 * <code>throws IOException</code>).
 *
 * If the native method declaration is not compatible with <code>NativeIOException</code>
 * (i.e., the throws clause does not specify <code>NativeIOException</code> or one of its
 * superclasses) then a code>NativeException</code> is thrown instead.
 *
 * The virtual machine keeps a reference to the message until the end of the current native
 * and does a copy of the message only when the current native returns. The message pointer
 * must not reference a memory allocated locally in the stack.
 *
 * If the current Java thread is suspended this function returns <code>SNI_ERROR</code>
 * and no exception is thrown.
 *
 * Calling this function while an exception is already pending will replace the previous
 * exception.
 *
 * If there is not enough space in the Java heap to allocate the exception, then an
 * <code>OutOfMemoryError</code> will be thrown in Java.
 *
 * @param errorCode is a value that can be retrieved in Java using the
 * <code>NativeIOException.getErrorCode()</code> method.
 *
 * @param message a null-terminated string that will be used to generate the message
 * returned by <code>NativeIOException.getMessage()</code> method. May be null.
```

Expose a C API in Java

JAVA CODING STYLE

- Try to stick to the Java standard naming:
  - Package names: only lower case (e.g., `com.example.bluetooth`)
  - Class names: UpperCamelCase (e.g., `BluetoothConnection`)
  - Method names: lowerCamelCase (e.g., `sendMessage`)
  - Constant names: UPPER_SNAKE_CASE (e.g., `BIG_ENDIAN`)

- Method overloading: two methods in a class with the same name and different arguments
  - `write(byte[] data)`
  - `write(byte[] data, int offset, int length)`

NATIVE METHOD DECLARATION

- Use the `native` keyword to make a method native
- The `static` keyword is required when declaring a native method
- No need for an object to call it: `<ClassName>.<methodName>()`

```
public class Example {
  public static native void foo();
}

public static void main(String[] args) {
  Example.foo();
}
```

PRIMITIVE TYPES

C	Java	sni.h typedefs
int8_t	byte	jbyte
uint8_t	-	-
bool	boolean	jboolean
int16_t	short	jshort
uint16_t	char ⚠	jchar
int32_t	int	jint
uint32_t	-	-
int64_t	long	jlong
uint64_t	-	-
float	float	jfloat
double	double	jdouble

⚠ Java `char` is 16-bit, C `char` is 8-bit

For **unsigned** types 8-bit, 32-bit:

- Apply mask and store the value on a larger type in Java
- Cast it to the right type when passing it to C

C function declarations:

```
uint8_t foo();
void bar(uint8_t b);
```

Java usage:

```
native public static byte foo();
native public static void bar(byte b);

int i = foo() & 0xFF;
bar((byte) i);
```

PRIMITIVE TYPES EXAMPLE

Expose the following C functions in Java:

```
int64_t foo(int x, int y);
void print(char c);
```

Java :

```
package quizz;

public class PrimitiveTypes {

  public static void main(String[] args) {
    long l = foo(42, 13);
    print((byte) 'x');
  }

  native public static long foo(int x, int y);
  native public static void print(byte b);
}
```

C :

```
#include <sni.h>

jlong Java_quizz_PrimitiveTypes_foo(jint x, jint y){
  return foo(x, y);
}

void Java_quizz_PrimitiveTypes_print(char c){
  print(c);
}
```

ARRAYS – JAVA STRUCTURE AND USAGE

- In C, arrays are basically just pointers
- In Java, arrays are objects with some meta-data including the length and the type of the elements

Typical Java array structure

Header: type of the elements, garbage collector related info, mutex info, …

Length: number of elements

Usage examples:

```
byte[] arrayOfByte = new byte[256]; // Arrays are allocated in the heap
int length = arrayOfByte.length;    // We can access their length
arrayOfByte[offset] = 42;           // Write access
byte value = arrayOfByte[offset];   // Read access
// Copy several elements from one array to another one.
// Similar to C function memmove().
System.arraycopy(source, srcPos, dest, destPos, length);
```

ARRAYS – SLICE

- In C, we can pass a reference to an element or a slice of an array
- In Java, it is not possible. We need to specify the offset within the array

C Example:

```
// Need to always specify the length
void write(char* data, int length);

char data[64];
write(&data[16], 8);
write(&data[0], 64);
```

Java Example:

```
// Need to specify offset and length
// to reference a slice of an array
void write(byte[] data, int offset, int length);
// No need to specify the length
// to reference the whole array
void write(byte[] data);

byte[] data = new byte[64];
write(data, 16, 8);
write(data);
```

ARRAYS – SNI AND JAVA ARRAYS

- Only arrays of primitive data type can be referenced (e.g., `int[]`, `char[]`, etc.)
- Passed as pointers to the 1st element of the array
- Length of the array can be retrieved with `SNI_getArrayLength()` macro

Java	SNI
byte[]	jbyte*
boolean[]	jboolean*
short[]	jshort*
char[]	jchar*
int[]	Jint*
long[]	jlong*
float[]	jfloat*
double[]	jdouble*

Java:

```
native public static void foo(byte[] array , int off, int len);
```

SNI C function :

```
void Java_pack_class_foo(jbyte* array, jint off, jint len){
  int length = SNI_getArrayLength(array);
  char* slice_start = &array[off];
}
```

ARRAYS – BOUNDS CHECK

- In Java, array accesses are checked and an exception is thrown in case of error:
  - `ArrayIndexOufOfBoundsException` if an array has been accessed with an illegal offset
  - `NullPointerException` if a reference to `null` is accessed

```
int[] array = new int[256];
array[-1] = 42; // -> ArrayIndexOufOfBoundsException
int b = array[256]; // -> ArrayIndexOufOfBoundsException

array = null;
int length = array.length; // -> NullPointerException
```

ARRAYS – BOUNDS CHECK AND SNI

- To ensure the reliability of the container and avoid any memory corruption, it is crucial to check the bounds of an array and the validity of a reference before using it in C
- Use the Java method `ArrayTools.checkBounds(byte[] bytes, int offset, int length)` before calling the native

Java:

```
native public static void foo(byte[] array , int off, int len);

// Throws NullPointerException if array is null.
// Throws IndexOutOfBoundsException if the slice [off, off+len[ is not valid.
ArrayTools.checkBounds(array, off, len);
foo(array, off, len);
```

Note:

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

SDK 6 (build.gradle.kts): `implementation("ej.library.runtime:basictool:1.7.0")`

SDK 5 (module.ivy): `<dependency org="ej.library.runtime" name="basictool" rev="1.7.0" />`

ARRAY AS A STRUCT

- In C, you can map a struct on an array

```c
typedef struct point{
    int32_t x;
    int32_t y;
} point_t;

void Java_com_example_Main_foo(int8_t* array){
    point_t* point = (point_t*)array;
    point->x = 32;
    point->y = 24;
}
```

- In Java use the `ByteArray` utility class defined in B-ON

```java
final int sizeOfPointStruct = ByteArray.INT_SIZE * 2;
final int offsetOfX = 0;
final int offsetOfY = ByteArray.INT_SIZE;
byte[] point = new byte[sizeOfPointStruct];
foo(point);
int x = ByteArray.readInt(point, offsetOfX);
int y = ByteArray.readInt(point, offsetOfY);
```

Note: Be careful on the alignment constraints of the fields and the potential padding added by the C compiler when the structure is not packed.

Assuming little-endian sytem

PRIMITIVE TYPES BY REFERENCE

- Use arrays of 1 element to pass primitive types by reference

C function declarations:

```
void foo(int32_t input, int32_t* output);
```

Java usage:

```
native public static void foo(int input, int[] output);

int[] outputRef = new int[1];
foo(42, outputRef);
int output = outputRef[0];
```

Note:

Avoid allocating a new array with each call to prevent creating pressure on the garbage collector. Try to reuse a global array as much as possible.

STRINGS

- In C, a string is a sequence of characters terminated with a null character `'\0'`
- In Java, a string is an object which references an array of char (`uint16_t`)

- `byte[] SNI.toCString(String javaString)`: Java String → C String

Java:

```
byte[] cString = SNI.toCString("Hello from Java");
print(cString);

native public static void print(byte[] cString);
```

C:

```
void Java_pack_Class_print(char* cString){
  int length = strlen(cString);
  printf("len=%d string=%s\n", length, cString);
}
```

- `String SNI.toJavaString(byte[] cString)`: C String → Java String

Java:

```
byte[] buffer = new byte[128];
getString(buffer);
String javaString = SNI.toJavaString(buffer);

native public static void getString(byte[] buffer);
```

C:

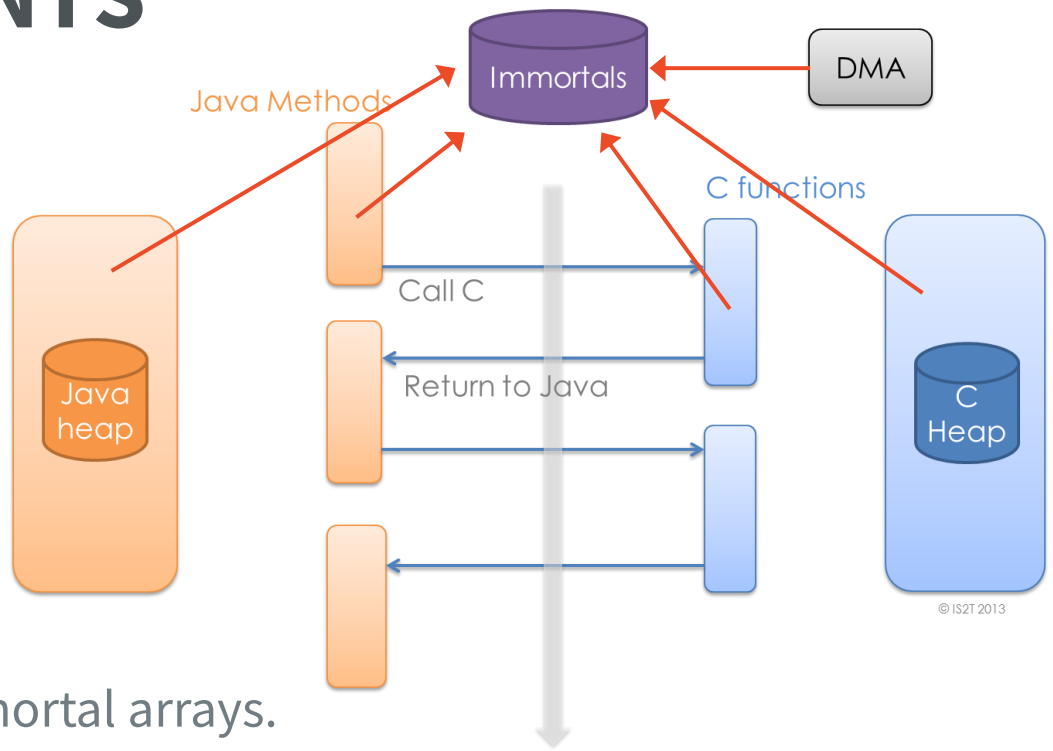
```
void Java_pack_Class_getString(char* buffer){
  int size = SNI_getArrayLength(buffer);
  strlcpy(buffer, "Hello from C", size);
}
```

Immortal Arrays

IMMORTAL ARRAYS – CONSTRAINTS

Two kinds of Java arrays:

- Reclaimable (default):
  - Can be garbage collected
  - Can be moved during heap compaction
- Immortal:
  - Fixed address
  - No free



- C globals, C Heap, DMA, RTOS tasks can reference only Immortal arrays.
- Reclaimable arrays can be referenced only from a native function local.
- Specified by the B-ON Library:
  - SDK5: `<dependency org="ej.api" name="bon" rev="1.4.3"/>`
  - SDK6: `implementation("ej.api:bon:1.4.3")`

IMMORTAL ARRAYS – CREATION

- Turns an array into an immortal array:

```
byte[] immortalArray = new byte[1024]; // Allocate a standard array
Immortals.setImmortal(immortalArray); // Turn it into an immortal array
```

- Allocate an immortal array directly:

```
static byte[] immortal1;
static float[] immortal2;
...
Immortals.run(new Runnable() {
        @Override
        public void run() {
                // All the objects allocated in this method are immortal.
                immortal1 = new byte[256];
                immortal2 = new float[512];
        }
});
```

IMMORTAL ARRAYS – USAGE

- In Java, use them as standard Java arrays:

```
immortalArray[42] = 12; // Write access
byte b = immortalArray[44]; // Read access
// Copy immortal array content into a standard array
System.arraycopy(immortalArraySrc, 0, arrayDest, 0, immortalArraySrc.length);
```

- Pass an immortal array to a native method:

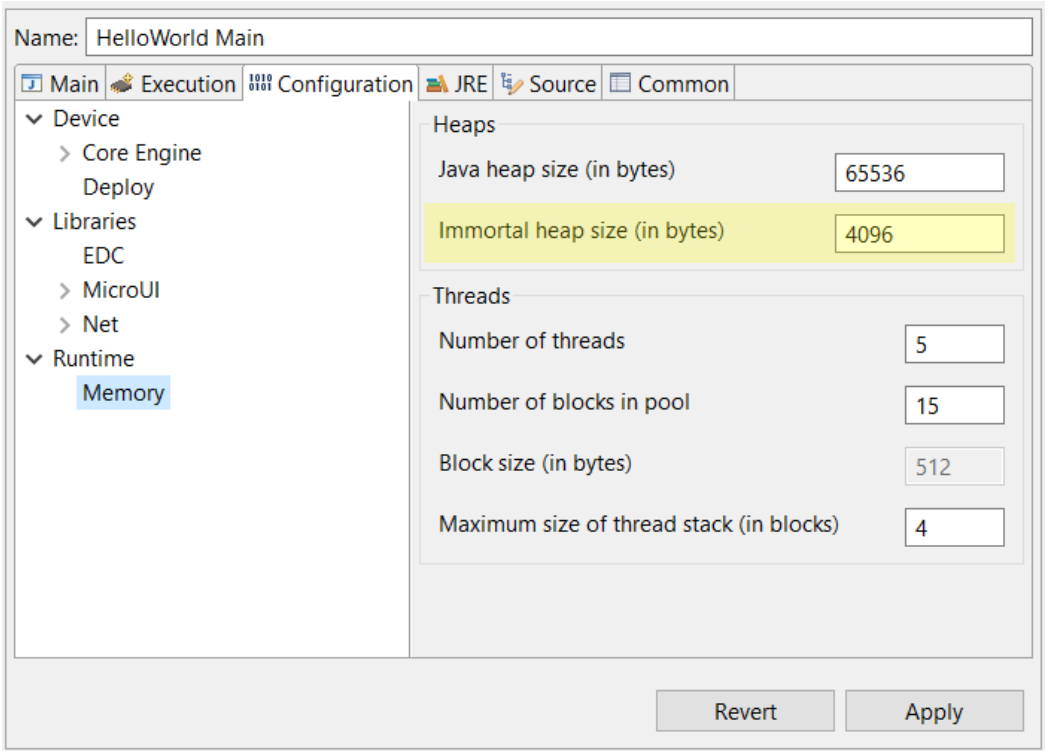
```
native private static void foo(byte[] array);
foo(immortalArray);
```

- Use it in C:

```
int8_t* global_immortal_array;
void Java_com_example_Main_foo(int8_t* array){
        int32_t array_length = SNI_getArrayLength(array);
        if(true == SNI_isImmortalArray(array)){
                global_immortal_array = array; // Keep a reference to the immortal array
        }
}
```

IMMORTAL HEAP

- Immortal heap size is defined at build-time via the launch configuration property `core.memory.immortal.size`



- At runtime, the immortal heap consumption can be retrieved:

```
long totalMemory = Immortals.totalMemory();
long freeMemory = Immortals.freeMemory();
long allocatedMemory = totalMemory - freeMemory;
System.out.println("Allocated immortal: " + allocatedMemory + " bytes");
```

Error Management

ERROR MANAGEMENT – JAVA VS C

C uses error code:

```
int socket = socket_open();
if(socket != -1){
  int res = socket_send(socket, data, len);
  if(res != -1){
    res = socket_read(socket, data, len);
    if(res != -1){
      ...
    }
  }
}
```

Java uses exception:

```
try {
  Socket socket = openSocket();
  socket.read(data);
  socket.write(data);
} catch (IOException e) {
  System.err.println(e.getMessage());
}
```

When exposing C API in Java, use exceptions instead of error code for error management

JAVA EXCEPTIONS

Java exceptions fall into two categories:

- Checked exceptions:
  - Used when the error can be recovered (e.g., failure when writing on a socket)
  - Must be caught with a try-catch block
  - Example: `IOException`, `FileNotFoundException`
- Unchecked exceptions:
  - Used when the error is actually a bug in the code (e.g., access null)
  - Extend the class `RuntimeException`
  - Example: `NullPointerException`, `ArrayIndexOutOfBoundsException`

SNI AND EXCEPTIONS

SNI offers two functions to create exceptions from C:

- `int32_t SNI_throwNativeIOException(int32_t errorCode, const char* message);`
  - Creates a `NativeIOException` subclass of `IOException`
  - Checked exception
- `int32_t SNI_throwNativeException(int32_t errorCode, const char* message);`
  - Creates a `NativeException` subclass of `RuntimeException`
  - Unchecked exception

- Java can access the error code and the message of the exception
- The exception is thrown when returning from the native function, and not when calling `SNI_throwNative(IO)Exception()`

SNI AND EXCEPTIONS – EXAMPLE

Java:

```
native static int open(byte[] path) throws NativeIOException;

try {
  int fd = open(SNI.toCString("/tmp/test.txt"));
} catch (NativeIOException e) {
  String message = e.getMessage();
  int errorCode = e.getErrorCode();
}
```

C:

```
int Java_pack_class_open(char* path){
  int fd = open(path, O_APPEND);
  if(-1 == fd){
    char* error_message = strerror(errno);
    SNI_throwNativeIOException(errno, error_message);
    …
  }
  return fd;
}
```

throws clause needs to be specified for checked exceptions. If this clause is not used, then the a `NativeException` will be thrown instead.

Code execution continues after this call.

The exception will be created and thrown only when returning from the native function.

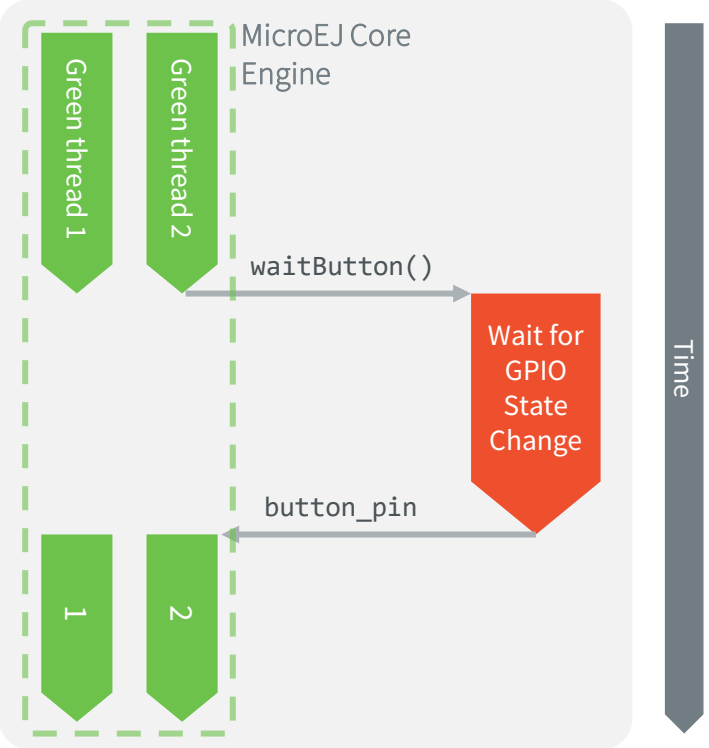
In case of exception, the returned value is ignored.

SNI and Blocking Functions

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



The MicroEJ VEE RTOS task

- While a native method is executed, other VEE threads can't be scheduled.
  - SNI functions stop the Managed Code 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 the native in another context and wait for the result without bocking the VEE.

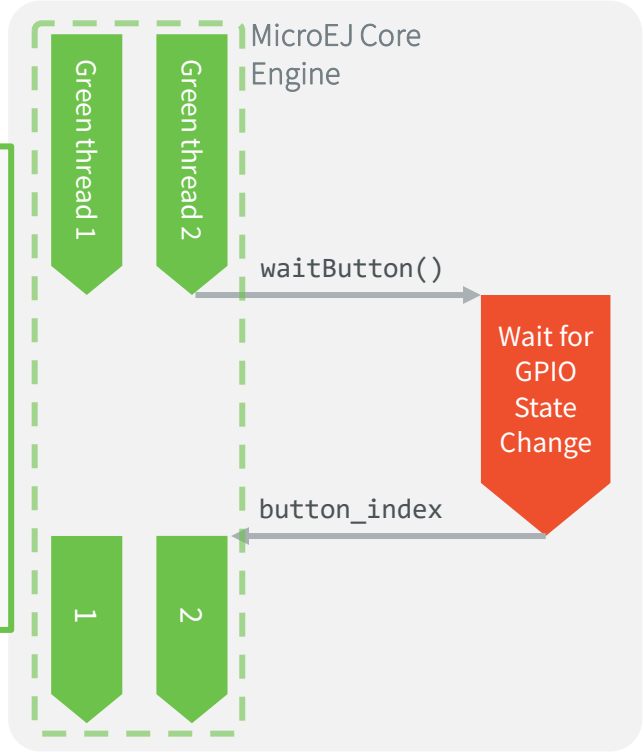
THREAD SYNCHRONIZATION: BLOCKING CASE

EXAMPLE OF NATIVE METHOD IMPLEMENTATION

In this example, the execution of the **waitButton()** native method will block until a button is pressed.
In other words, while **Java_example_NativeCCallExample_waitButton()** has not returned, no other Java thread can be scheduled.

This is because the native function is called in the **same RTOS/OS task as the Java application**:

```java
package example;

public class NativeCCallExample {

    public static void main(String[] args) {

        while (true) {
            System.out.println("Waiting for a button event...");
            int buttonIndex = waitButton();
            System.out.println("Button pressed: " + buttonIndex);
        }
    }

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



The MicroEJ VEE RTOS task

```c
#include "semaphore.h"
#include "sni.h"

static int pressed_button_index;
static Semaphore button_semaphore;

void button_init(){
    button_semaphore = SemaphoreCreateBinary();
}

jint Java_example_NativeCCallExample_waitButton(){
    // Wait for a button event
    semaphoreTake(button_semaphore);
    return pressed_button_index;
}

/** Interrupt request handler called when a button is pressed. */
int buttonIRQ(int button_index){
    pressed_button_index = button_index;
    semaphoreGiveFromISR(button_semaphore);
}
```

Hands-On

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

PREREQUISITES

Hardware required:

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

Environment Setup:

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

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

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

HANDS-ON OVERVIEW (1/3)

INTRODUCTION

The next slides present a concrete use case of blocking SNI function implementation.

The goal of the hands-on is to make a **non-blocking** implementation of this function.

A ready to use application project (**gpio-basic**) is provided in the training package associated with those slides.

The hands-on is done **on device only** (no Simulator), using the **NXP i.MXRT1170 EVK** board.

Note that the mechanisms presented in those slides are also applicable in simulation. Refer to the Mock documentation for more information.

All the steps are also described in the slides in case you are not able to run the sample on the device.

HANDS-ON OVERVIEW (2/3)

APPLICATION PROJECT OVERVIEW

The **gpio-basic** project provides the **app-sni-blocking** application that defines 2 threads:

- Main thread: toggles the board LED1 every 500 ms
- Input thread: waits for a button event and prints the index of the pressed button (User/Blue button)

This hands-on focuses on the implementation of the **waitButton()** native method.

The goal of the hands-on is to make a **non-blocking** implementation of the **waitButton()** native method.

Gpio.java

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

Main.java

```
public class Main {

   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();
}
```

HANDS-ON OVERVIEW (3/3)

WAITBUTTON() NATIVE METHOD IMPLEMENTATION

- The execution of the **LLGPIO_waitButton()** function will block until a button is pressed. It is waiting for a semaphore to be released.

- The semaphore is released when a button press (interrupt) occurs.

LLGPIO_blocking_NXP-i.MX_RT1170.c

```
#define LLGPIO_waitButton Java_com_microej_training_gpio_Main_waitButton

static uint8_t GPIO_initialized = 0;
static volatile jint button_index;

static SemaphoreHandle_t button_semaphore;


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

 // Wait for the interrupt
 xSemaphoreTake(button_semaphore, portMAX_DELAY);
 return button_index;
}


/** Interrupt request handler called when the button is pressed. */
void BOARD_ButtonHandler(void* arg)
{
   portBASE_TYPE xHigherPriorityTaskWoken = pdFALSE;
   xSemaphoreGiveFromISR(button_semaphore, &xHigherPriorityTaskWoken);
   if(xHigherPriorityTaskWoken != pdFALSE )
   {
      // Force a context switch here.
      portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
   }
}
```

RUN THE EXAMPLE CODE (1/5)

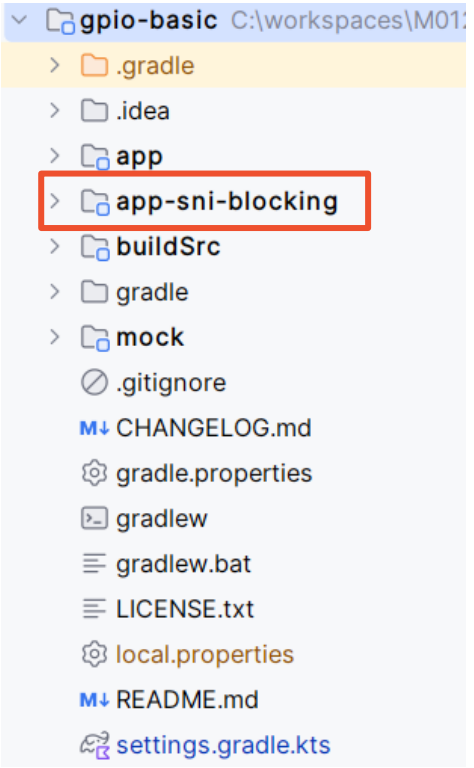
GET THE TRAINING RESOURCES

- Download and extract the training resources provided with this training, it should contain at least:
    - **gpio-basic/**

RUN THE EXAMPLE CODE (2/5)

OPEN THE PROJECT

- In your IDE, open the **gpio-basic** example:
  - Open the project using the menu **File** > **Open...**
  - Click on OK.
  - Select to open in a new Window.
- The project appears in the IDE.
- The hands-on focuses on the **app-sni-blocking** application.



RUN THE EXAMPLE CODE (3/5)

VEE PORT SELECTION

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

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

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

```
dependencies {
    . . .

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

RUN THE EXAMPLE CODE (4/5)

ADD NATIVE METHODS IMPLEMENTATION TO THE BSP

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

CMakeLists.txt

```
80
81  add_executable(${MCUX_SDK_PROJECT_NAME}
82  "${MicroEjRootDirPath}/src/main/LLGPIO_non_blocking_NXP-i.MX_RT1170.c"
83  "${MicroEjRootDirPath}/src/main/nxpvee_ui.c"
84  "${MicroEjRootDirPath}/src/main/nxpvee_ui.h"
```

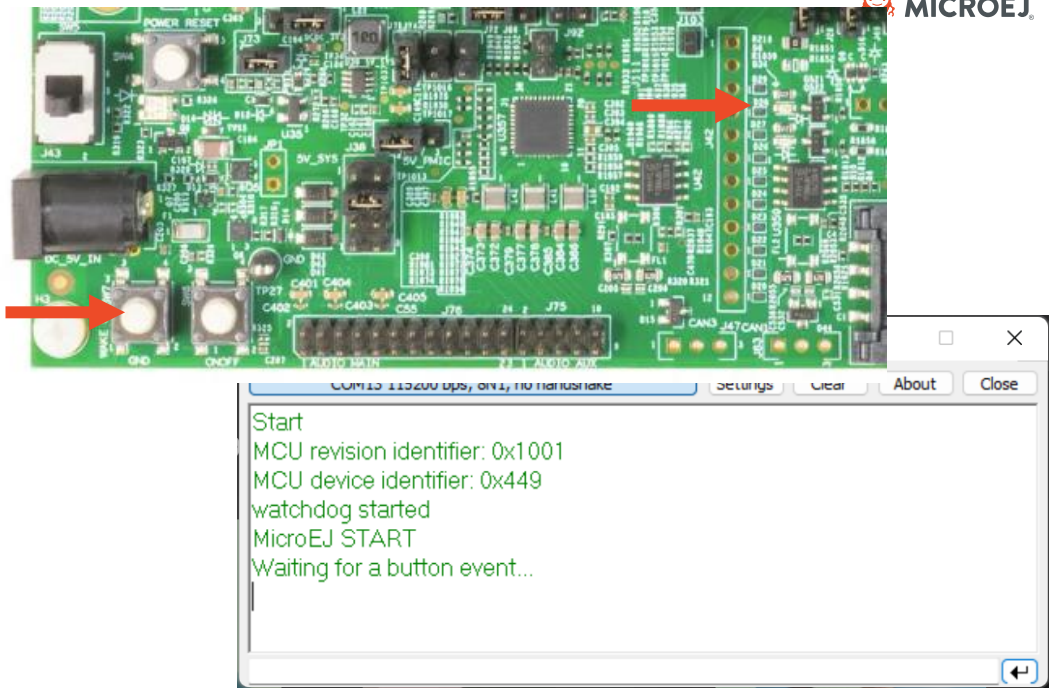
RUN THE EXAMPLE CODE (5/5)

- Open the **Gradle tasks** view.
- Open **Tasks > gpio-basic > app-sni-blocking > microej** and double-click on **runOnDevice**.

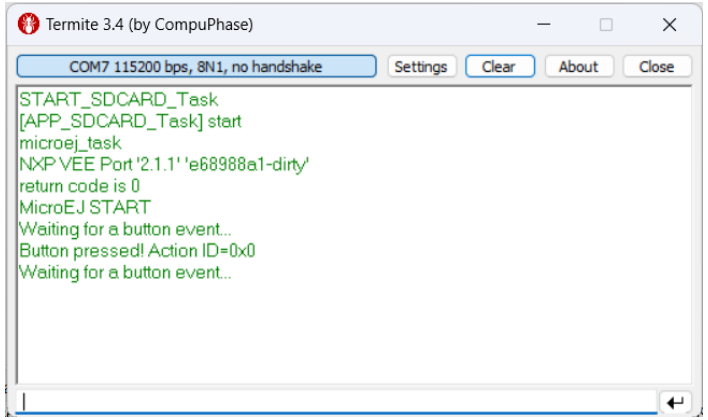
- Open the Termite serial terminal.
- Reset the NXP i.MX RT1170 EVK board using the Reset button.
- The application starts and waits for a button event.

- **LED1 is not blinking every 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

HANDS-ON DIRECTIVES

- The goal is to make a **non-blocking** implementation of the **waitButton()** native method.
- 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

SUSPEND AND RESUME A THREAD (1/3)

SNI APIS OVERVIEW

SNI offers several functions to control the execution of a VEE thread:

```
int32_t SNI_suspendCurrentJavaThread(int64_t timeout);
```

- Suspends the execution of the current VEE thread
- This is not a C blocking function (i.e., it returns immediately)
- Suspension takes effect when the native method returns

```
int32_t SNI_resumeJavaThread(int32_t javaThreadID);
```

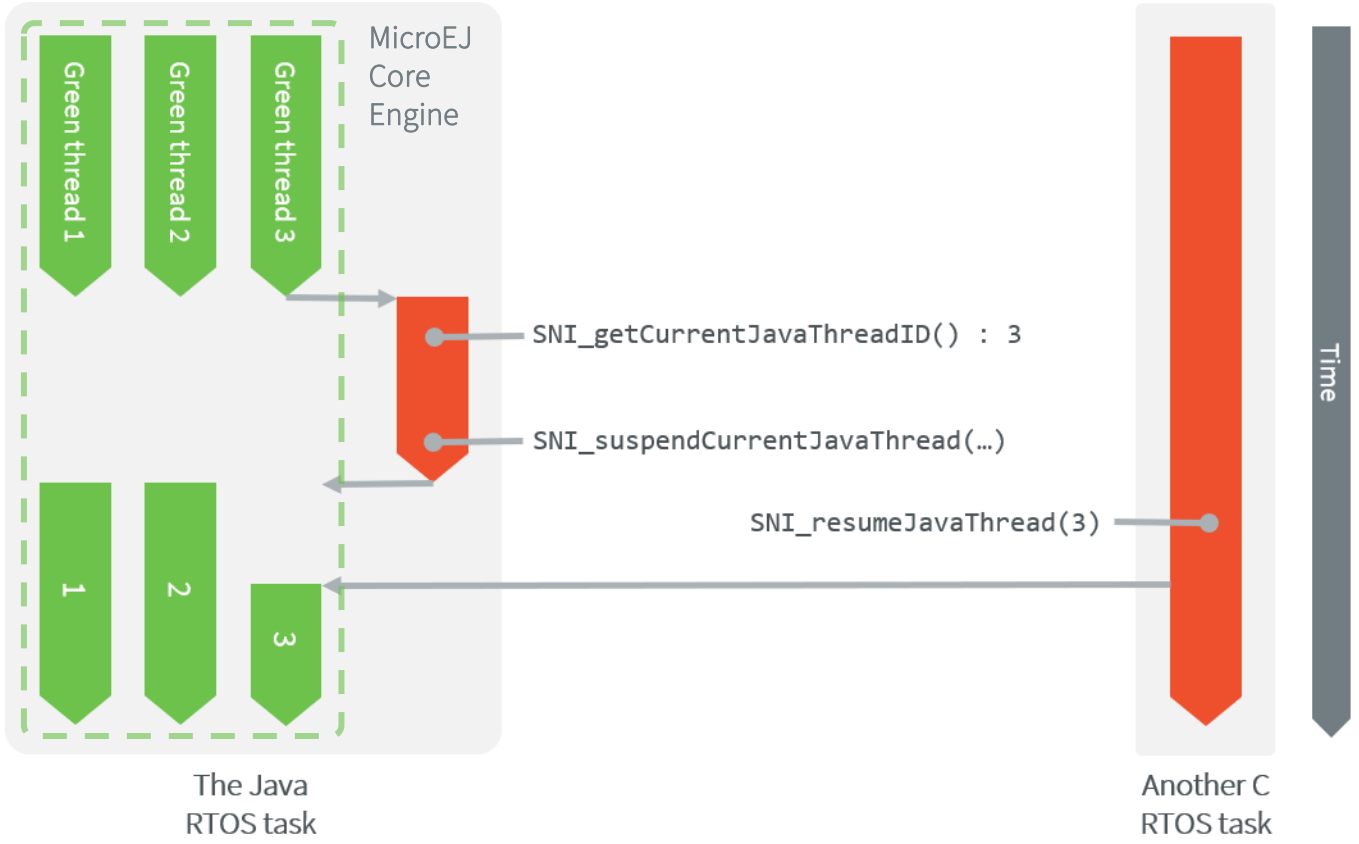
- Resumes the given VEE thread

```
int32_t SNI_getCurrentJavaThreadID(void);
```

- Returns the ID which can be used to resume the VEE thread

SUSPEND AND RESUME A THREAD (2/3)

EXAMPLE OF FLOW



SUSPEND AND RESUME A THREAD (3/3)

EXAMPLE USING SNI APIS

C:

```
int32_t thread_id;

void Java_com_ex_Example_blockingNative(){
  thread_id = SNI_getCurrentJavaThreadID();
  SNI_suspendCurrentJavaThread(0);
  printf("Thread is suspended\n");
}

void anotherTask(){
  SNI_resumeJavaThread(thread_id);
}
```

Java:

```
public static native void blockingNative();

public static void main(String[] args) {
  println("Thread will be suspended...");
  blockingNative();
  println("Thread resumed");
}
```

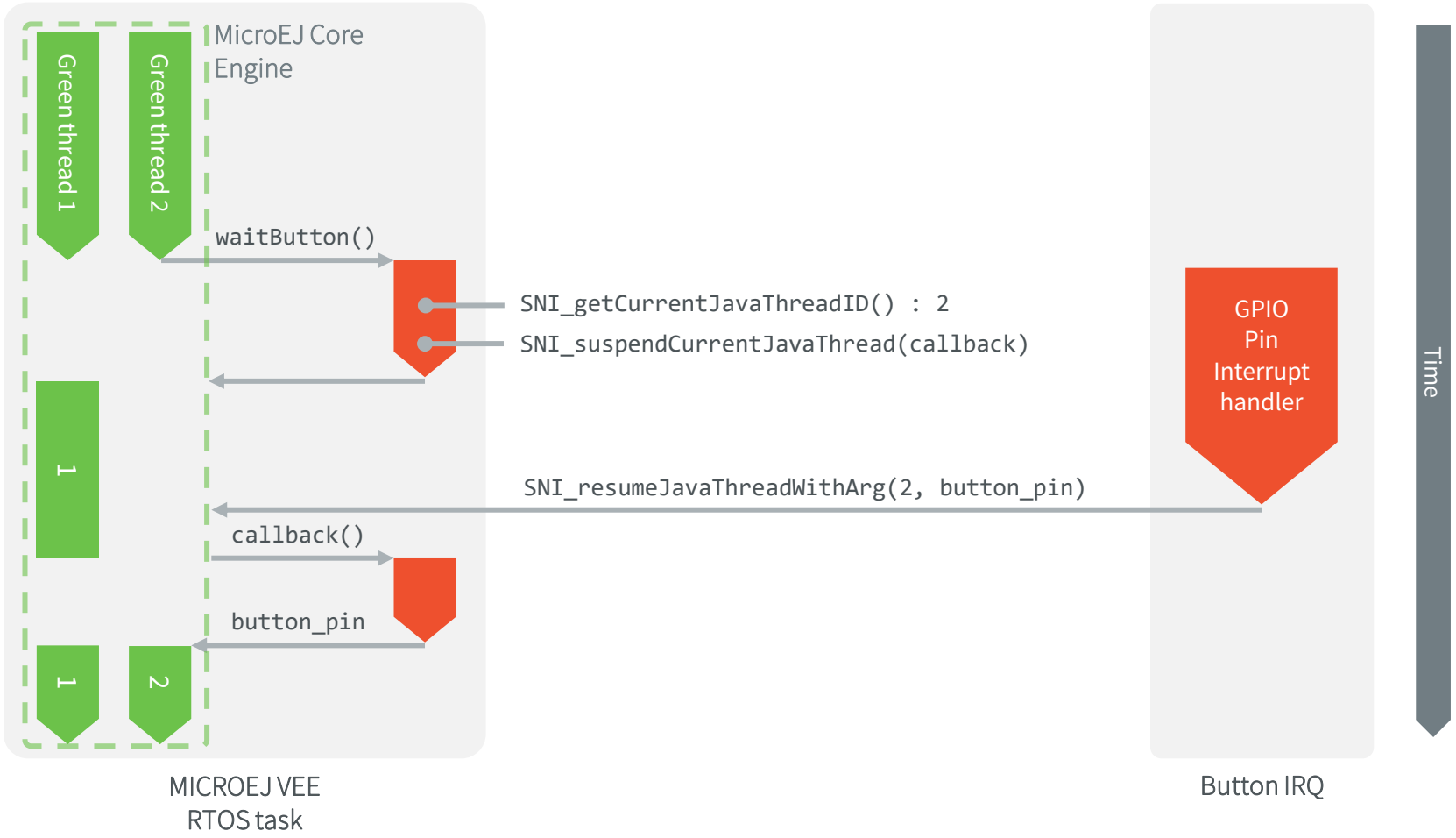
SUSPEND AND RESUME A THREAD WITH CALLBACK

- When using **`SNI_suspendCurrentJavaThread()`** with **`SNI_resumeJavaThread()`** the thread cannot execute C code before returning to the managed code.
    - This requires the application to call a new native method to retrieve the result of an asynchronous operation.

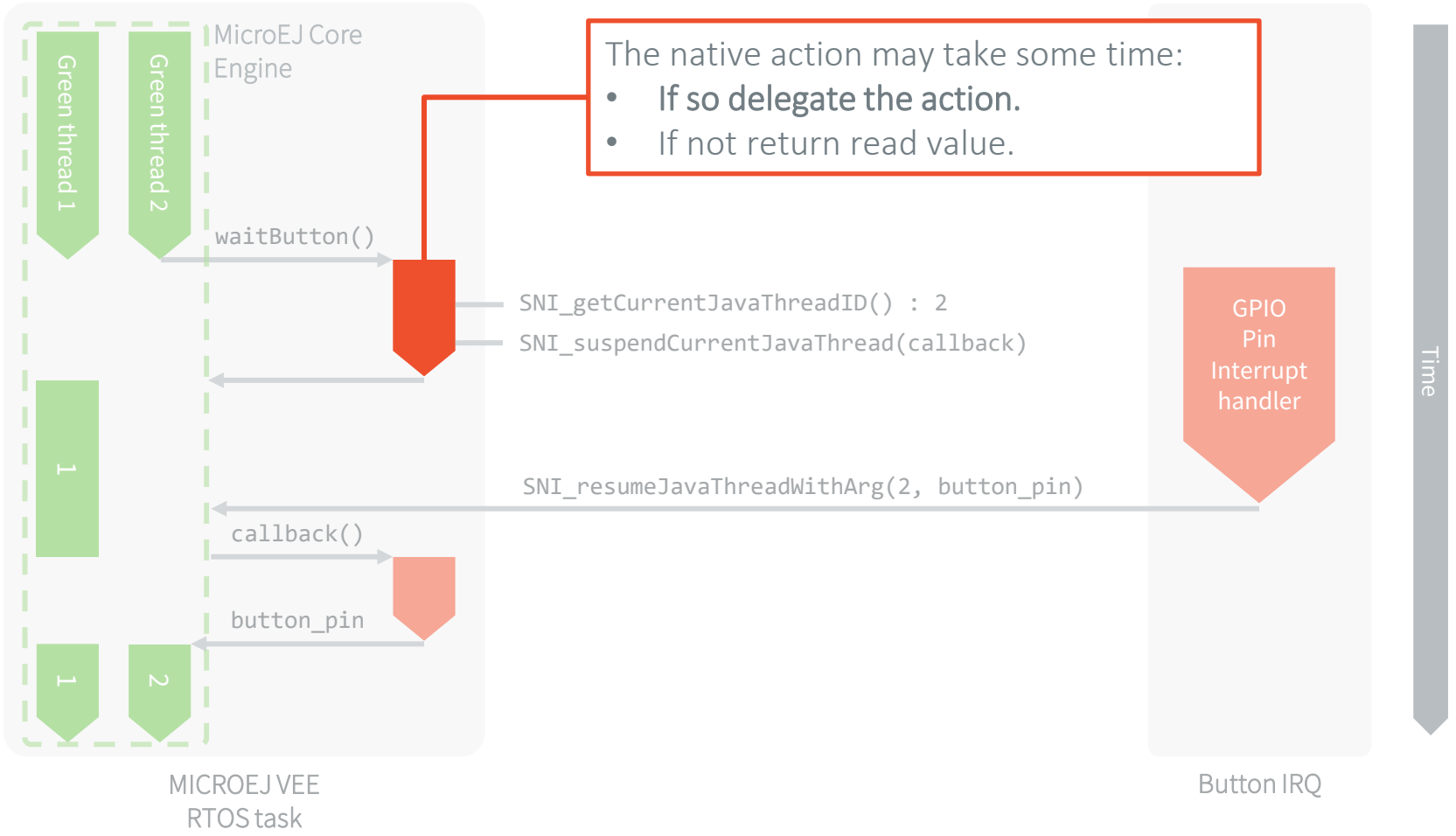
- **`SNI_suspendCurrentJavaThreadWithCallback()`** allows to specify a C function called when the Java thread is resumed.

→ The **`SNI_suspendCurrentJavaThreadWithCallback()`** method will be used in this hands-on in order to retrieve the **button index**.

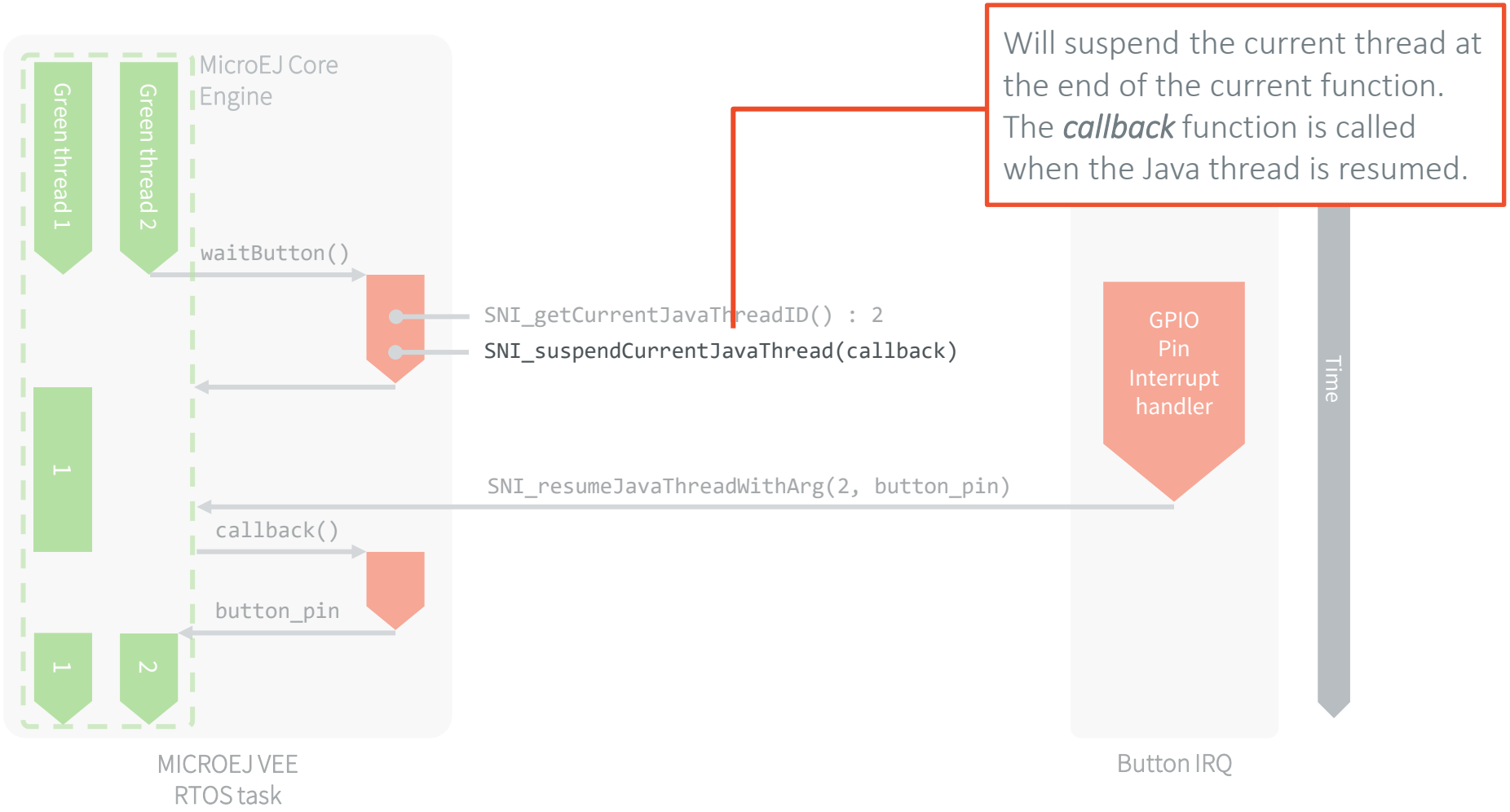
THREAD SYNCHRONIZATION: CALLBACK PATTERN



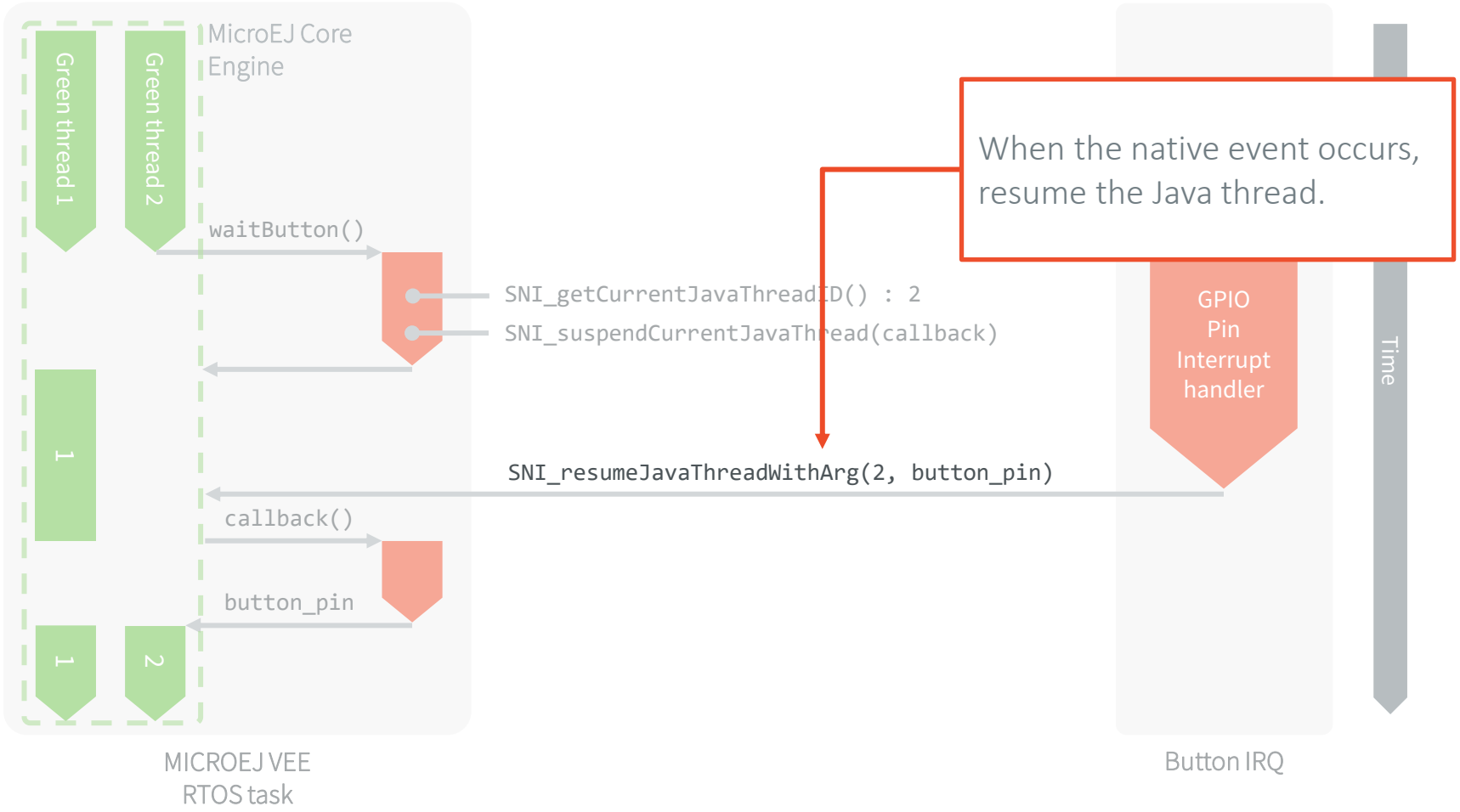
THREAD SYNCHRONIZATION: CALLBACK PATTERN



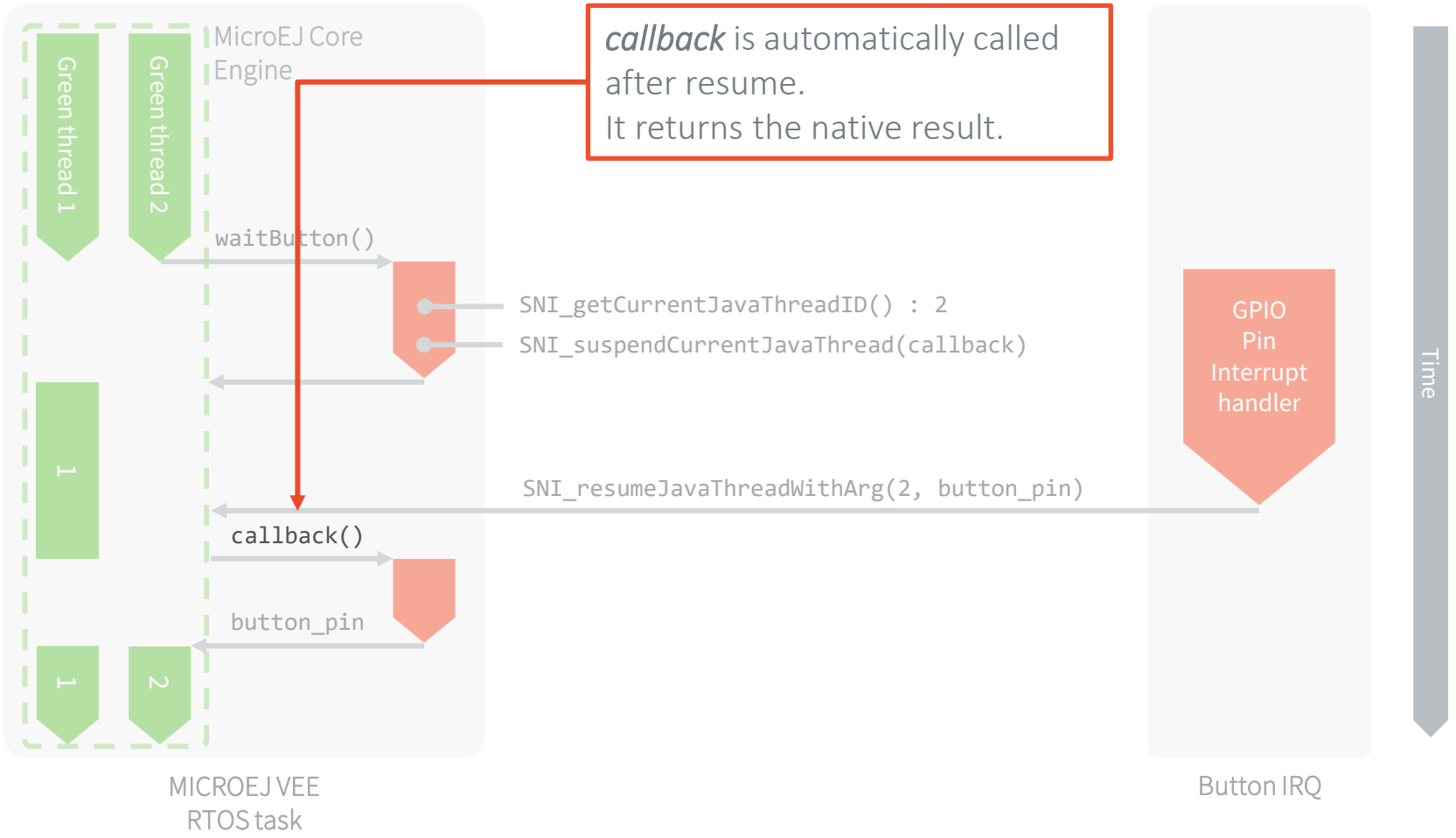
THREAD SYNCHRONIZATION: CALLBACK PATTERN



THREAD SYNCHRONIZATION: CALLBACK PATTERN



THREAD SYNCHRONIZATION: CALLBACK PATTERN



STEP 1: UPDATE THE NATIVE IMPLEMENTATION

- The **LLGPIO_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 **LLGPIO_waitButton()** doesn't matter anymore. The callback function will be in charge of returning the value.

```
static int32_t java_thread_id = -1;

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

    int32_t result = SNI_OK;
    java_thread_id = SNI_getCurrentJavaThreadID();
    result = SNI_suspendCurrentJavaThreadWithCallback(0, (SNI_callback)waitButton_callback, NULL);
    if(SNI_OK != result){
        java_thread_id = -1;
    }

    return SNI_IGNORED_RETURNED_VALUE; // Returned value not used
}
```

Note: the non-blocking implementation is available in gpio-basic-{version}/app-sni-blocking/src/main/c/LLGPIO_non_blocking_NXP-i.MX_RT1170.c

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 that 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;

    if(java_thread_id != -1){
        SNI_resumeJavaThreadWithArg(java_thread_id, (void*)&button_index); // save the button_index
pointer in the VM
        java_thread_id = -1;
    }
}
```

Note: the non-blocking implementation is available in gpio-basic-{version}/app-sni-blocking/src/main/c/LLGPIO_non_blocking_NXP-i.MX_RT1170.c

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
}
```

Note: the non-blocking implementation is available in gpio-basic-{version}/app-sni-blocking/src/main/c/LLGPIO_non_blocking_NXP-i.MX_RT1170.c

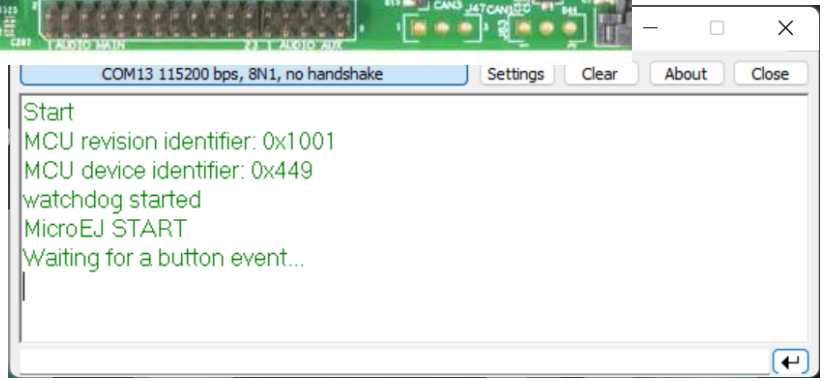
RUN THE UPDATED CODE

- Open the **Gradle** view.
- Open **Tasks > gpio-basic > app-sni-blocking > microej** and double-click on **runOnDevice**.

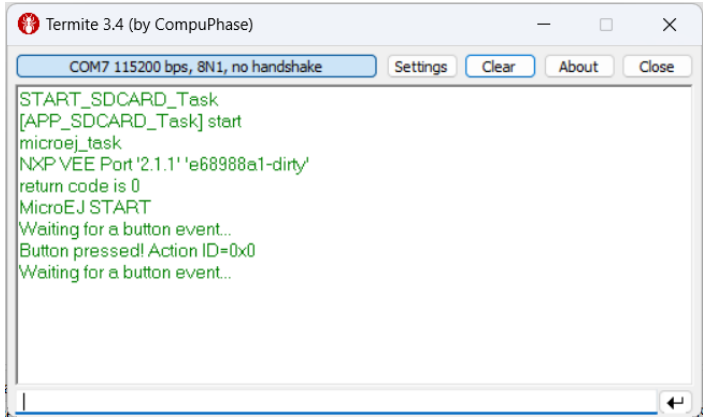
- Open the Termite serial terminal.
- Reset the NXP i.MX RT1170 EVK board using the Reset button.
- The application starts and waits for a button event.

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

- When pressing the button, the ID of the button event is printed in the console.



Traces when the application starts



Traces after 1 button press

Async Worker

ASYNCHRONOUS WORKER API

- C Library that helps writing natives that must be executed asynchronously
- Goal
  - Execute long natives in a dedicated RTOS task
- Examples
  - Filesystem accesses
  - Cryptographic algorithm
  - BLE communication

ASYNCHRONOUS WORKER API

- Typical usage consists in declaring:
  - for each SNI function, a structure that contains the parameters of the function,
  - an union of all the previously declared structures,
  - the async worker using `MICROEJ_ASYNC_WORKER_worker_declare()` macro.

EXAMPLE

- Natives:

```
int foo(int i, int j);
void bar(int i);
```

- Declare a structure for each native:

```
// foo() parameters structure
typedef struct {
	int i;
	int j;
	int result;
} foo_param_t;
```

```
// bar() parameters structure
typedef struct {
	int i;
} bar_param_t;
```

EXAMPLE

- Union of all the previously declared structures:

```
// union of all the parameters structures
typedef union {
      foo_param_t foo;
      bar_param_t bar;
      ...
} my_worker_param_t;
```

EXAMPLE

- Declare and initialize the async worker

```
MICROEJ_ASYNC_WORKER_worker_declare(my_worker, MY_WORKER_JOB_COUNT,
                                    my_worker_param_t, MY_WORKER_WAITING_LIST_SIZE);

void initialize(){
      MICROEJ_ASYNC_WORKER_initialize(&my_worker, "My Worker", my_worker_stack,
                                      MY_WORKER_PRIORITY);
      ...
}
```

EXAMPLE

- Implement the native:

```
int foo(int i, int j){
        MICROEJ_ASYNC_WORKER_job_t* job =
                    MICROEJ_ASYNC_WORKER_allocate_job(&my_worker, (SNI_callback)foo);
        foo_param_t* params = (foo_param_t*)job->params;
        params->i = i;
        params->j = j;
        MICROEJ_ASYNC_WORKER_async_exec(&my_worker, job, foo_action,
                                        (SNI_callback)foo_on_done);
        // Returned value is not used.
        return -1;
}
```

EXAMPLE

- Implement the action to execute asynchronously:

```
void foo_action(MICROEJ_ASYNC_WORKER_job_t* job){
        foo_param_t* params = (foo_param_t*)job->params;
        int i = params->i;
        int j = params->j;
        ...
        params->result = result;
}
```

EXAMPLE

- Implement the callback called when the action is done:

```
int foo_on_done(int i, int j){
        MICROEJ_ASYNC_WORKER_job_t* job = MICROEJ_ASYNC_WORKER_get_job_done();
        foo_param_t* params = (foo_param_t*)job->params;
        int result = params->result;
        MICROEJ_ASYNC_WORKER_free_job(&my_worker, job);
        return result;
}
```

GOING FURTHER WITH ASYNC WORKER

See the Delegate Blocking Operations using Async Worker training to get an example on how to manage filesystem accesses using Async Worker.

Arrays and Asynchronous Execution

ARRAYS AND ASYNCHRONOUS EXECUTION

- When the array is immortal
  - It can be used directly by other C tasks, DMA, interrupts, …
  - You don’t need to copy the data
- When the object is **NOT** immortal and data is used out of the native
  - Java array must be copied into a C buffer
  - C buffer data must be copied back to the Java array

HANDLE IMMORTAL AND NON-IMMORTAL ARRAYS

- In a native, if you want to manage both immortals and non immortals:

```
char buffer[1024];
void write(int fd, char* data, int len){
        char* actual_data;
        int actual_length;
        if(SNI_isImmortal(data)){
                actual_data = data;
                actual_length = len;
        }
        else {
                actual_data = buffer;
                actual_length = len < sizeof(buffer) ? len : sizeof(buffer);
                memcpy(buffer, data, actual_length);
        }
        async_write(fd, actual_data, actual_length);
}
```

If the given array is immortal, you can use it in another thread.

If the given array is NOT immortal, you need an intermediate buffer to use it in another thread.

You also need to copy the data to this buffer.

UTILITY FUNCTIONS FOR ARRAYS

- Use the following functions to automatically get the right buffer and copy the content if needed:

```
int32_t SNI_retrieveArrayElements(jbyte* java_array, jint java_start, jint java_length, int8_t* buffer, uint32_t buffer_length, int8_t** out_buffer, uint32_t* out_length, bool refresh_content);
```

```
int32_t SNI_flushArrayElements(jbyte* java_array, jint java_start, jint java_length, int8_t* buffer, uint32_t buffer_length);
```

SEND OR WRITE DATA

```
char buffer[1024];
void write(int fd, char* data, int len){
      char* actual_data;
      int actual_length;
      SNI_retrieveArrayElements(data, 0, len,
                        buffer, sizeof(buffer),
                        &actual_data, &actual_length,
                        true);
      async_write(fd, actual_data, actual_length);
}
```

Copy the content if data is not immortal

RECEIVE OR READ DATA

```
char buffer[1024];
void read(int fd, char* data, int len){
    char* actual_buffer;
    int actual_length;
    SNI_retrieveArrayElements(data, 0, len, buffer, sizeof(buffer),
                              &actual_buffer, &actual_length, false);
    async_read(fd, actual_buffer, actual_length);
    SNI_suspendCurrentJavaThreadWithCallback(0, (SNI_callback)read_done, NULL);
}
void read_done(int fd, char* data, int len){
      // Copy back the read data only if the Java object is not immortal
      SNI_flushArrayElements(data, 0, len, buffer, sizeof(buffer));
}
```

No need to copy the content even if data is not immortal

Reference C Resources

REFERENCE C RESOURCES

- Sometimes you need to reference C resources from Java
- Examples:
    - Socket
    - File Descriptor
    - Dynamically allocated structure
    - Peripheral description structure

REFERENCE C RESOURCES – USE CASE

- How to expose safely this API?

```
FILE* fopen(char* path);
int fclose(FILE *file);
int fread(void* ptr, int size, FILE* file);
int fwrite(void* ptr, int size, FILE* file);
```

- Basic implementation:

```
public static class File {
  native public static int fopen(byte[] path);
  native public static int fclose(int file);
  native public static int fread(byte[] data, int offset, int length, int file);
  native public static int fwrite(byte[] data, int offset, int length, int file);
}
```

- What's wrong with this implementation?

REFERENCE C RESOURCES

- Error 1: Use an `int` (32-bit) to handle a pointer

```
public static class File {
   native public static int fopen(byte[] path);
   native public static int fclose(int file);
   native public static int fread(byte[] data, int offset, int length, int file);
   native public static int fwrite(byte[] data, int offset, int length, int file);
}
```

- What if I run on a 64-bit architecture?

REFERENCE C RESOURCES

- Error 1: Use an `int` (32-bit) to handle a pointer

```
public static class File {
   native public static long fopen(byte[] path);
   native public static int fclose(long file);
   native public static int fread(byte[] data, int offset, int length, long file);
   native public static int fwrite(byte[] data, int offset, int length, long file);
}
```

Use longs (64-bit) instead of ints (32-bit) to be 64-bit compatible

REFERENCE C RESOURCES

- Error 2: Give the application direct access to a pointer

```
public static class File {
   native public static long fopen(byte[] path);
   native public static int fclose(long file);
   native public static int fread(byte[] data, int offset, int length, long file);
   native public static int fwrite(byte[] data, int offset, int length, long file);
}
```

- What if the application calls `fclose()` with a random pointer?

REFERENCE C RESOURCES

Benefit from Java encapsulation to make your code more robust.

- Error 2: Give the application direct access to a pointer

```
static class File {

  private long filePtr;

  private File(long filePtr) {
    this.filePtr = filePtr;
  }

  public static File openFile(byte[] path) {
    long filePtr = fopen(path);
    if (filePtr == 0) {
      return null;
    } else {
      return new File(filePtr);
    }
  }

  public int read(byte[] data, int offset, int length) {
    return fread(data, offset, length, this.filePtr);
  }

  public int write(byte[] data, int offset, int length){
    return fwrite(data, offset, length, this.filePtr);
  }

  public int close() {
    int result = fclose(this.filePtr);
    this.filePtr = 0;
    return result;
  }

  native private static long fopen(…);

  native private static int fclose(…);

  …
}
```

REFERENCE C RESOURCES

- Error 3: Expect a null-terminated String when opening a file

```
public static File openFile(byte[] path) {
  long filePtr = fopen(path);
  if (filePtr == 0) {
    return null;
  } else {
    return new File(filePtr);
  }
}
```

- What if the application calls `openFile()` with a non-null-terminated String?

REFERENCE C RESOURCES

- Error 3: Expect a null-terminated String when opening a file

```
public static File openFile(String path) {
  long filePtr = fopen(SNI.toCString(path));
  if (filePtr == 0) {
    return null;
  } else {
    return new File(filePtr);
  }
}
```

Use Java Strings in the public API.

This is safer and more user-friendly.

REFERENCE C RESOURCES

- Bad practice: use error codes

```
private File(long filePtr) {
  this.filePtr = filePtr;
}

public static File openFile(String path) {
  long filePtr = fopen(SNI.toCString(path));
  if (filePtr == 0) {
    return null;
  } else {
    return new File(filePtr);
  }
}

public int read(byte[] data, int offset, int length) {
  return fread(data, offset, length, this.filePtr);
}

native private static long fopen(…);
native private static int fread(…);
```

REFERENCE C RESOURCES

- Bad practice: use error codes

```
public static File openFile(String path) throws NativeIOException {
  long filePtr = fopen(SNI.toCString(path));
  return new File(filePtr);
}

public int read(byte[] data, int offset, int length) throws NativeIOException {
  return fread(data, offset, length, this.filePtr);
}

native private static long fopen(…) throws NativeIOException;
native private static int fread(…) throws NativeIOException;
```

No need to check the result of fopen( ) anymore

Use exceptions to create a more Java-like public API.

REFERENCE C RESOURCES

- Simplify even more:

```
public File(String path) throws NativeIOException {
  this.filePtr = fopen(SNI.toCString(path));
}

public int read(byte[] data, int offset, int length) throws NativeIOException {
  return fread(data, offset, length, this.filePtr);
}

native private static long fopen(…) throws NativeIOException;
native private static int fread(…) throws NativeIOException;
```

Expose directly a public constructor

REFERENCE C RESOURCES

- Error 4: array bounds are not checked

```
public int read(byte[] data, int offset, int length) throws NativeIOException {
  return fread(data, offset, length, this.filePtr);
}

public int write(byte[] data, int offset, int length) throws NativeIOException {
  return fwrite(data, offset, length, this.filePtr);
}
```

- What if the application calls `read()` with a negative offset or a null array?

REFERENCE C RESOURCES

- Error 4: array bounds are not checked

```
public int read(byte[] data, int offset, int length) throws NativeIOException {
  ArrayTools.checkBounds(data, offset, length);
  return fread(data, offset, length, this.filePtr);
}

public int write(byte[] data, int offset, int length) throws NativeIOException {
  ArrayTools.checkBounds(data, offset, length);
  return fwrite(data, offset, length, this.filePtr);
}
```

Use `ArrayTools.checkBounds()`

Don’t rely on C developers 😁

SNI Native Resources

SNI NATIVE RESOURCES

- Some data created in a native and referenced by a Java object
- Examples:
    - Socket
    - File Descriptor
    - Cryptographic algorithm data structure
    - Event structure executed asynchronously
- These resources needs to be released when:
    - close() method is called on the Java object
    - The linked Java object is garbage collected
    - The VEE stops its execution
    - An application that creates a resource is stopped (in multi-sandbox VEE)
    - The native method and the callbacks chain return

WHEN DO I NEED SNI NATIVE RESOURCES?

You are handling a native resource in a native function every time you:

- Allocate/Free some data (malloc/free)
- Keep/Release a lock on a logical or physical resource (UART driver, …)
- Open/Close a resource (socket, file, …)
- Use a temporary structure during callbacks chain (message in message queue, …)

HOW TO USE SNI NATIVE RESOURCES?

- sni.h defines:

```
int32_t SNI_registerResource(void* resource, SNI_closeFunction close,
SNI_getDescriptionFunction getDescription);
```

```
int32_t SNI_unregisterResource(void* resource, SNI_closeFunction close);
```

- A resource is uniquely identified by a pointer to a resource + a close function
- The `close` function is called when:
  - The application that has created the native resource is stopped (in multi-sandbox VEE)
  - The VEE is stopped
- The `close` function is **NOT** called by `SNI_unregisterResource()`

SNI NATIVE RESOURCE DESCRIPTION

- When declaring a native resource, a `getDescription` pointer can be passed:

```
typedef void (*SNI_getDescriptionFunction)(void* resource, char* buffer,
uint32_t bufferLength);
```

- The `getDescription` function is called:
  - In C by the function `LLMJVM_dump()` (see `LLMJVM.h`)
  - In Java by the methods
    - `static void NativeResource.printRegisteredNativeResources(PrintStream out)`
    - `String NativeResource.getDescription()`

AUTOMATIC CLOSE ON GC

- A Native Resource can be linked to a Java Object
- When the Java Object is garbage collected the close function is called automatically
- In Java, use the methods:

```
static void NativeResource.closeOnGC(long resource, long closeFunction,
Object obj)
```

SCOPED NATIVE RESOURCES

- When using SNI suspend or yield, some data may be:
  - Allocated in the native
  - Used and released in the callback
- In a multi-sandbox VEE, if an application is stopped while the thread is suspended:
  - The callback is not called
  - So the data is not released.

SCOPED NATIVE RESOURCES – USE CASE

```
void fill_rect(int x, int y, int width, int height){
	VG_draw* draw_command = (VG_draw*)malloc(sizeof(VG_draw));
	draw_command->command = FILL_RECT;
	draw_command->fillRect.x = x;
	draw_command->fillRect.y = y;
	draw_command->fillRect.width = width;
	draw_command->fillRect.height = height;
	vg_engine_execute(draw_command);
	SNI_suspendCurrentJavaThreadWithCallback(0, (SNI_callback)fill_rect_done,
draw_command);
}

void fill_rect_done(int x, int y, int width, int height){
	VG_draw* draw_command;
	SNI_getCallbackArgs(&draw_command, NULL);
	free(draw_command);
}
```

SCOPED NATIVE RESOURCES – USAGE

- Using `SNI_registerResource/unregisterResource()` takes too much time and can lower the performances
- For such kind of native resources use:

```
int32_t SNI_registerScopedResource(void* resource, SNI_closeFunction
close, SNI_getDescriptionFunction getDescription);
```

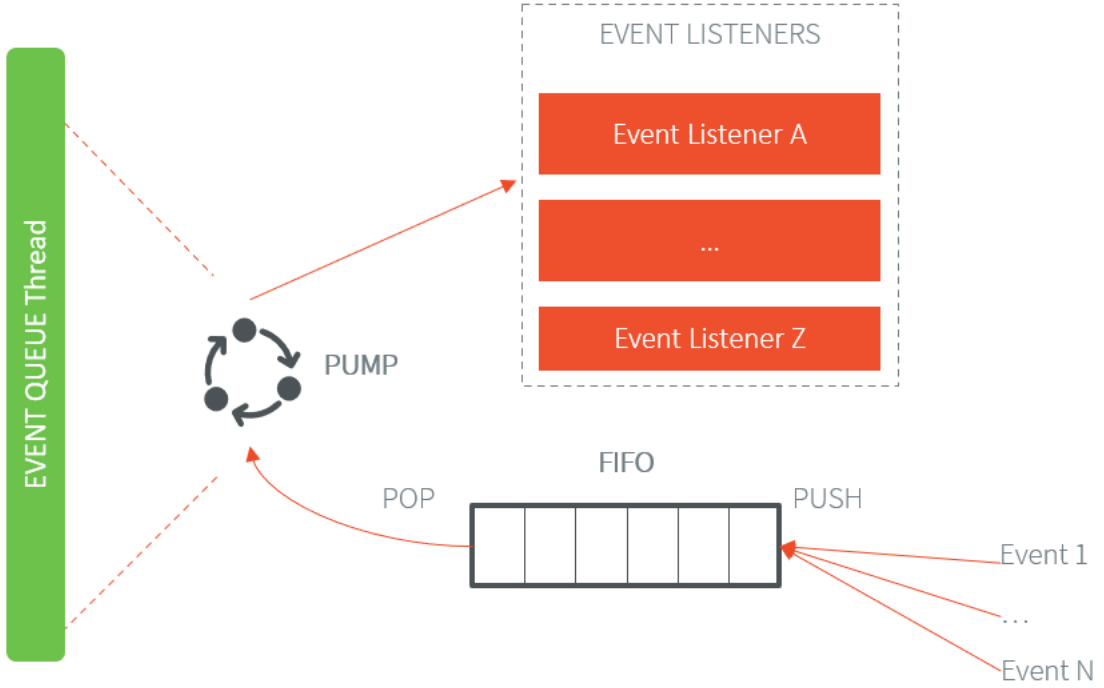
- The scoped native resource is automatically closed and unregistered when:
  - The native and all its callbacks return
  - The application that has created the native resource is stopped (in multi-sandbox VEE)
  - The VEE is stopped

Event Queue

OVERVIEW

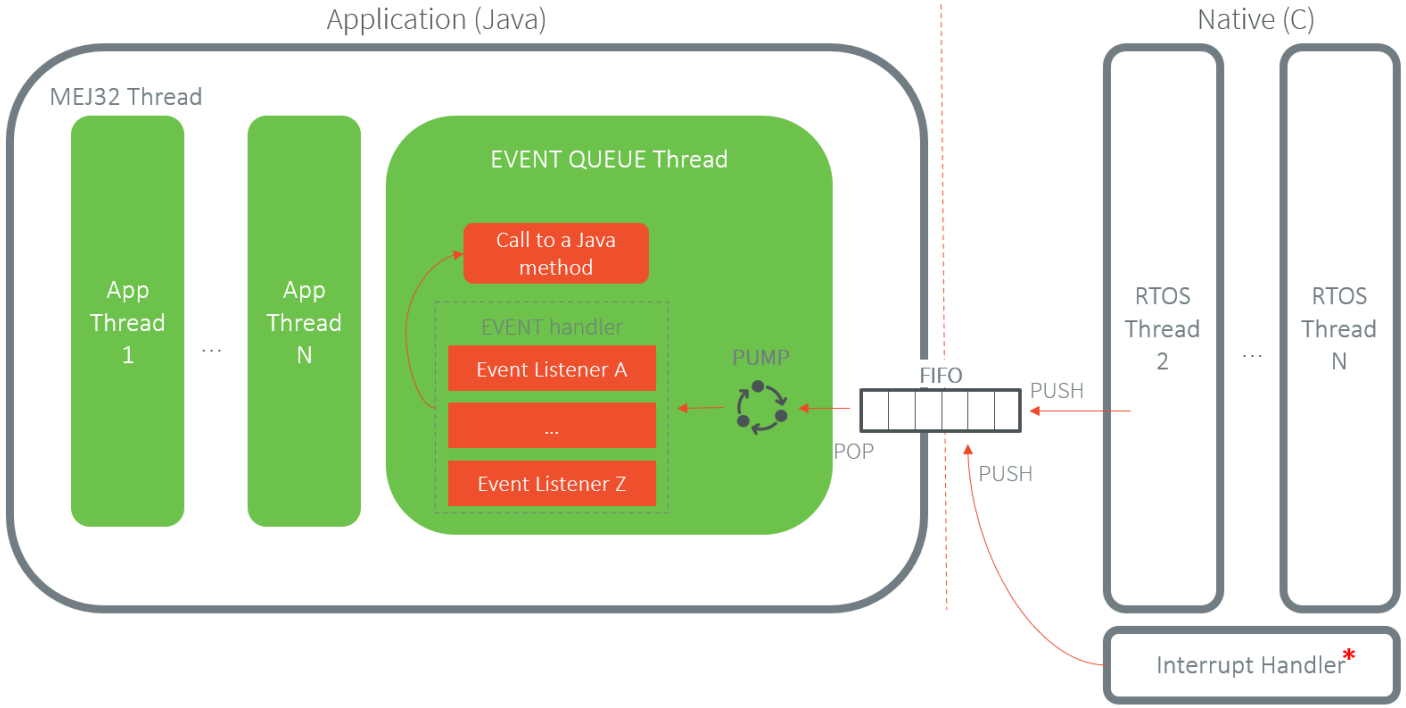
- At a glance
  - Provides an asynchronous communication interface between the native world and the Java world
  - Event-based
  - One way communication: native event producers → VEE event consumers.

- Principles
  - Implemented using a native FIFO queue
  - The Java API allows the caller code to register an event listener for subscribing to events with specific event types and handling them as they are triggered
  - The C API allows the caller code to trigger events
  - The Java API allows the caller code to trigger events as well



ARCHITECTURE

- The Event Queue Foundation Library uses a dedicated Java thread to forward and process events. Application event listener's calls are done in the context of the Event Queue thread.

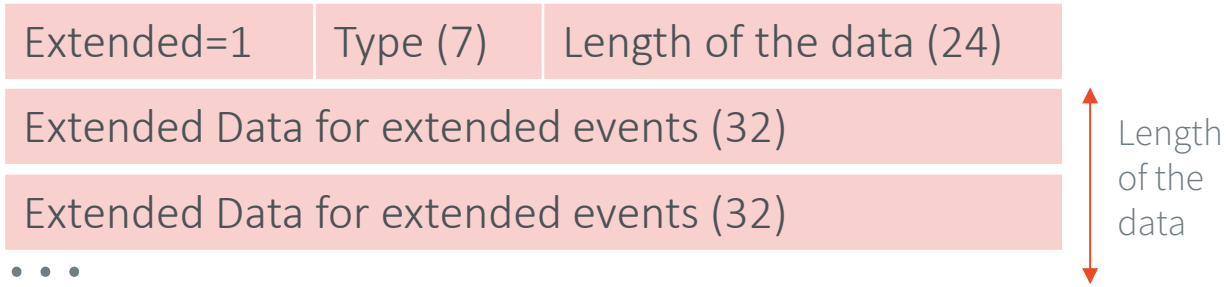


- Event Queue Java thread is suspended when the events FIFO is empty and resumed when a new event is sent.

EVENTS

- Two kinds of events:
  - Standard: 32-bit including 24-bit for the payload
  - Extended: variable size payload

Non-extended Event format



Extended Event format

- Event Format:
  - Extended: extended event flag (1 bit)
  - Type: event type which allows to find the corresponding event listener (7 bits)
  - Data: data of non-extended events (24 bits)
  - Length: length of the data in bytes for extended events (24 bits)
  - Extended Data: data of extended events (32 bits)

EVENT TYPES

- Event type allows to identify which listener will handle the event
- Number of event types is limited: 128 (7 bits)
- Good practice:
    - Ensure all events managed by the same software component share the same event type
    - Use the payload to differentiate between various subtypes of events within the same type
    - This approach maintains consistency and simplifies event handling by limiting the number of listeners
    - Example: Bluetooth events (pairing, receiving messages, etc.) should all have the same event type
- Bad practice:
    - Having numerous event types for closely related functions
    - Example: Define types 'BluetoothPairing’ for pairing events and 'BluetoothMessage’ for receiving messages

EVENT QUEUE LISTENER (1/2)

- An application can register listeners to the Event Queue.
- Each listener is registered for a specific event type. The same listener can be registered several times for different event types, but each event type can only have one listener.

```java
public static int eventType;

public static void main(String[] args) throws InterruptedException {
   EventQueue eventQueue = EventQueue.getInstance();

   // Get the unique type to register your listener.
   // eventType must be stored if you want to offer an event from the Java API.
   eventType = eventQueue.getNewType();

   // Create and register a listener.
   eventQueue.registerListener(new ExampleListener(), eventType);

   // Send eventType to the C world.
   initialize(eventType);
}

/**
* This native method will take the event type as an entry and store it in the C world.
*/
public static native void initialize(int type);
```

Example of Event Queue initialization

STANDARD EVENT EXAMPLE

OFFER THE EVENT

- From the C API:

```c
#include "LLEVENT.h"

// Assuming that event_type has been passed from
the Java world through a native method after
registering your listener.
int type = event_type;
int data = 12;

LLEVENT_offerEvent(type, data);
```

- From the Java API:

```java
EventQueue eventQueue = EventQueue.getInstance();

// Assuming that eventType has been stored in the
Java world when you registered the listener.
int type = eventType;
int data = 12;

eventQueue.offerEvent(type, data);
```

HANDLE THE EVENT

- The data received by the Event Queue is processed in the **handleEvent(int type, int data)** method.

```java
EventQueue queue = EventQueue.getInstance();
int type = queue.getNewType();
initialize(type);
queue.registerListener(type, new EventQueueListener() {
   @Override
   public void handleEvent(int type, int data) {
      System.out.println("My data is equal to: " + data);
   }
   @Override
   public void handleExtendedEvent(int type, EventDataReader
eventDataReader) {
      throw new RuntimeException();
   }
});
```

EXTENDED EVENT EXAMPLE (1/2)

OFFER THE EVENT FROM THE C API

- The C-struct must respect a specific Data Alignement.

```c
#include "LLEVENT.h"

struct accelerometer_data {
   int x;
   int y;
   int z;
}

// Assuming that event_type has been passed from
the Java world through a native method after
registering your listener.
int type = event_type;

struct accelerometer_data data;
data.x = 42;
data.y = 72;
data.z = 21;

LLEVENT_offerExtendedEvent(type, (void*)&data,
sizeof(data));
```

OFFER THE EVENT FROM THE JAVA API

```java
EventQueue eventQueue = EventQueue.getInstance();

// Assuming that eventType has been stored in the Java
world when you registered the listener.
int type = eventType;

// Array of 3 integers. Each integer is stored in 4 bytes.
byte[] accelerometerData = new byte[3*4];

// Write integers into the byte array using ByteArray API.
ByteArray.writeInt(accelerometerData, 0, 42);
ByteArray.writeInt(accelerometerData, 4, 72);
ByteArray.writeInt(accelerometerData, 8, 21);

eventQueue.offerExtendedEvent(type, accelerometerData);
```

EXTENDED EVENT EXAMPLE (2/2)

HANDLE THE EVENT

- Implement the **handleExtendedEvent(int type, EventDataReader eventDataReader)** method to process the received data.
- The data is retrieved in a byte array.
- Use **EventDataReader** to read the data.

```
EventQueue queue = EventQueue.getInstance();
int type = queue.getNewType();
initialize(type);
queue.registerListener(type, new EventQueueListener() {
   @Override
   public void handleEvent(int type, int data) {
      throw new RuntimeException();
   }
   @Override
   public void handleExtendedEvent(int type, EventDataReader eventDataReader) {
      int x = 0;
      int y = 0;
      int z = 0;
      try {
         x = eventDataReader.readInt();
         y = eventDataReader.readInt();
         z = eventDataReader.readInt();
      } catch (IOException e) {
         System.out.println("IOException while reading accelerometer values from the
EventDataReader.");
      }
      System.out.println("Accelerometer values: X = " + x + ", Y = " + y + ", Z = " +
z + ".");
   }
});
```

MOCK THE EVENT QUEUE

- To simulate event that are normally sent through the C API, use the Event Queue Mock API from your mock.
- The Event Queue Mock API dependency must be added to the module.ivy of your MicroEJ Mock project:

```
<dependency org="com.microej.pack.event" name="event-pack" rev="2.0.0" conf="provided->mockAPI"/>
```

- It provides two methods:
  - **EventQueueMock.offerEvent(int type, int data)** is the equivalent of **LLEVENT_offerEvent(int32_t type, int32_t data)** method from **LLEVENT.h**.
  - **EventQueueMock.offerExtendedEvent(int type, byte[] data, int dataLength)** is the equivalent of **LLEVENT_offerExtendedEvent(int32_t type, void* data, int32_t data_length)** method from **LLEVENT.h**.
- Example of use:

```
// Assuming that event_type has been passed from your Application
through a native method after registering your listener.
int type = event_type;
int data = 12;

EventQueueMock.offerEvent(type, data);
```

DOCUMENTATION

- Application Developer Guide documentation:
  https://docs.microej.com/en/latest/ApplicationDeveloperGuide/eventQueue.html#event-queue

- VEE Porting guide documentation:
  https://docs.microej.com/en/latest/VEEPortingGuide/packEventQueue.html

UI Pump

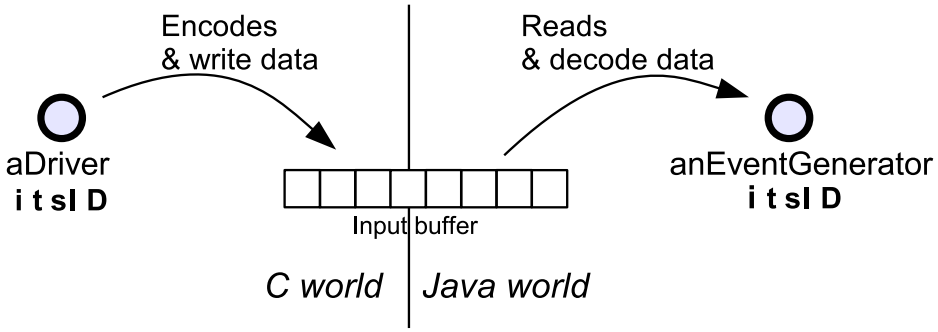
PRINCIPLE

The Input module contains the C part of the MicroUI implementation which manages input devices. This module is composed of two elements:

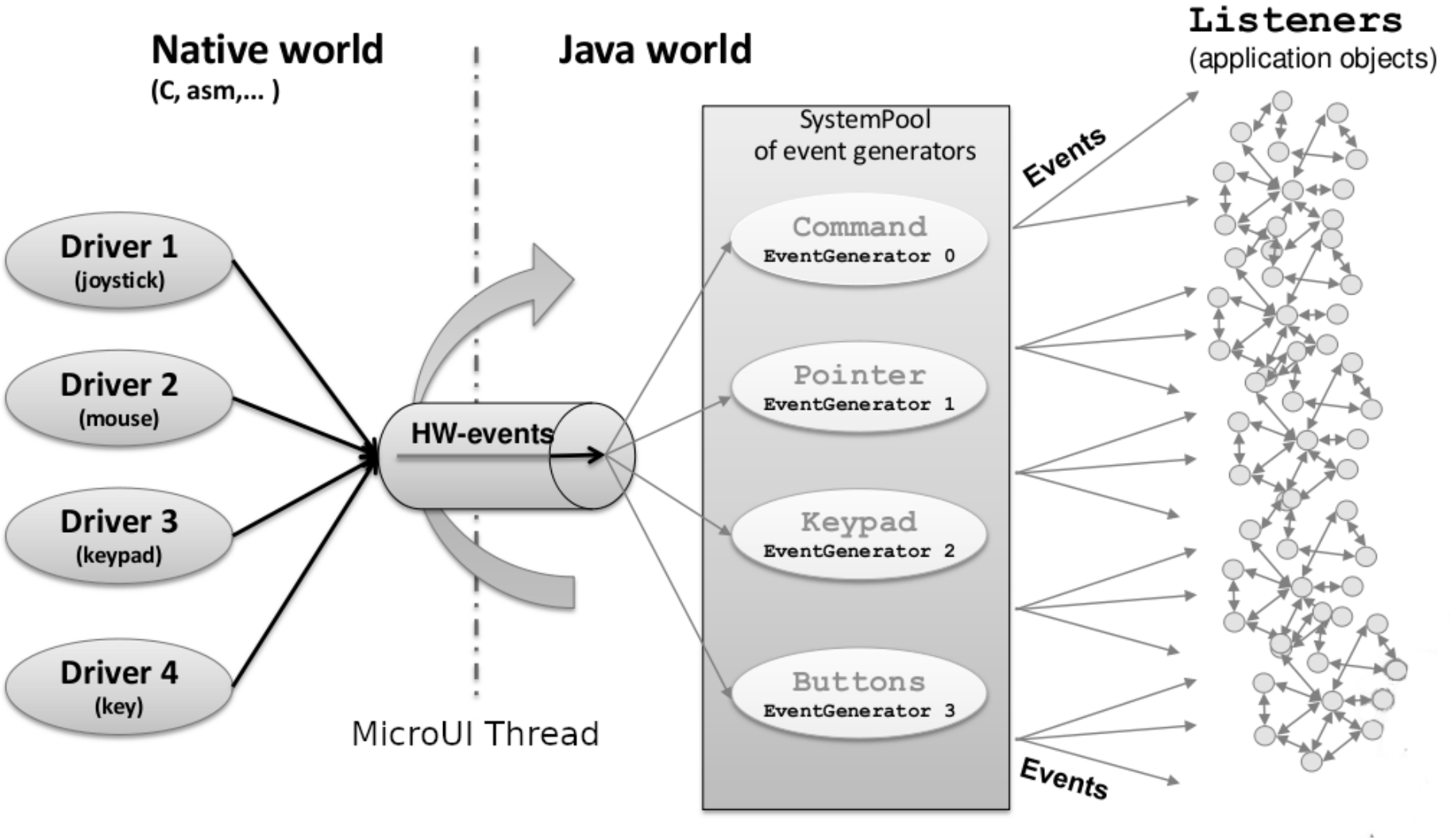
- the C part of MicroUI input API (a built-in C archive) called Input Engine,
- an implementation of a Low Level APIs for the input devices that must be provided by the BSP

Each MicroUI Event Generator represents one side of a pair of collaborative components that communicate using a shared buffer:

- The producer: the C driver connected to the hardware. As a producer, it sends its data into the communication buffer.
- The consumer: the MicroUI Event Generator. As a consumer, it reads (and removes) the data from the communication buffer.



MICROUI EVENTS FRAMEWORK



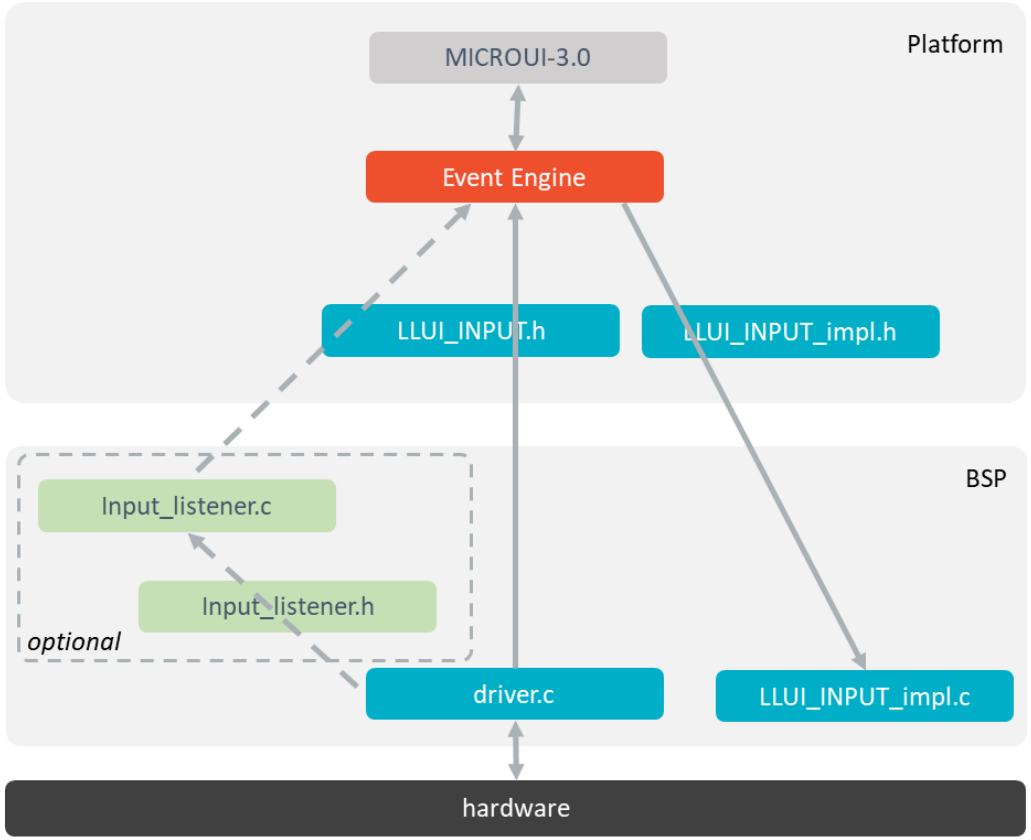
EVENT GENERATORS

- MicroUI provides a set of standard event generators: Command, Buttons, Pointer and States. For each standard generator, the Input Engine proposes a set of functions to create and send an event to this generator.

- MicroUI provides an abstract class GenericEventGenerator (package ej.microui.event). The aim of a generic event generator is to be able to send custom events from native world to the application. These events may be constituted by only one 32-bit word or by several 32-bit words (maximum 255).

LOW LEVEL API

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



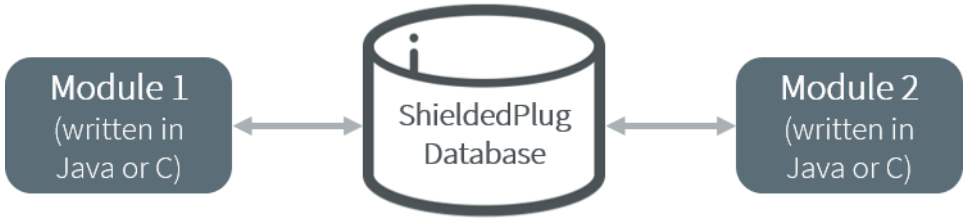
USEFUL LINKS

- Documentation:
  - https://docs.microej.com/en/latest/PlatformDeveloperGuide/uiInput.html
- Javadoc:
  - https://repository.microej.com/javadoc/microej_5.x/apis/ej/microui/event/EventGenerator.html
- C implementation example:
  - https://github.com/MicroEJ/Platform-STMicroelectronics-STM32L4R9-DISCO/blob/master/STM32L4R9DISCO-bsp/projects/microej/ui/inc/event_generator.h
  - https://github.com/MicroEJ/Platform-STMicroelectronics-STM32L4R9-DISCO/blob/master/STM32L4R9DISCO-bsp/projects/microej/ui/src/LLUI_INPUT_impl.c
  - https://github.com/MicroEJ/Platform-STMicroelectronics-STM32L4R9-DISCO/blob/master/STM32L4R9DISCO-bsp/projects/microej/ui/src/event_generator.c

Shielded Plug

PRINCIPLE

The Shielded Plug (SP) library provides data segregation with a clear publish-subscribe API. The data-sharing between modules uses the concept of shared memory blocks, with introspection. The database is made of blocks: chunks of RAM.



SHIELDED PLUG DATABASE

The usage of the Shielded Plug (SP) starts with the definition of a database. The implementation uses an XML file description to describe the database; the syntax follows the one proposed by the [SP] specification.

Example:

```
<shieldedPlug>
    <database name="Forecast" id="0" immutable="true" version="1.0.0">
        <block id="1" name="TEMP" length="4" maxTasks="1"/>
        <block id="2" name="THERMOSTAT" length="4" maxTasks="1"/>
    </database>
</shieldedPlug>
```

SHIELDED PLUG API

Once this database is defined, it can be accessed within the MicroEJ Application or the C application.

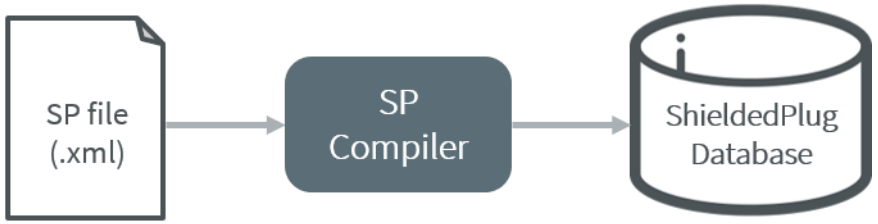
The SP Java Foundation Library is accessible from the SP API Module. This library contains the classes and methods to read and write data in the database.

The C header file sp.h available in the MicroEJ Platform source/include folder contains the C functions for accessing the database.

SHIELDED PLUG COMPILER

A MicroEJ tool is available to launch the compiler. The tool name is Shielded Plug Compiler. It outputs:

- A description of the requested resources of the database as a binary file (.o) that will be linked to the overall application by the linker. It is an ELF format description that reserves both the necessary RAM and the necessary Flash memory for the Shielded Plug database.
- Two descriptions, one in Java and one in C, of the block ID constants to be used by either Java or C application modules.



C EXAMPLE

Here is a C header that declares the constants defined in the XML description of the database.

```
#define Forecast_ID 0
#define Forecast_TEMP 1
#define Forecast_THERMOSTAT 2
```

Below, the code shows the publication of the temperature and thermostat controller task.

```
void temperaturePublication() {
   ShieldedPlug database = SP_getDatabase(Forecast_ID);
   int32_t temp = temperature();
   SP_write(database, Forecast_TEMP, &temp);
}

void thermostatTask(){
   int32_t thermostatOrder;
   ShieldedPlug database = SP_getDatabase(Forecast_ID);
   while(1){
      SP_waitFor(database, Forecast_THERMOSTAT);
      SP_read(database, Forecast_THERMOSTAT, &thermostatOrder);
      if(thermostatOrder == 0) {
         thermostatOFF();
      }
      else {
         thermostatON();
      }
   }
}
```

JAVA EXAMPLE

From the database description we can create an interface.

```
public interface Forecast {
   public static final int ID = 0;
   public static final int TEMP = 1;
   public static final int THERMOSTAT = 2;
}
```

Below is the task that reads the published temperature and controls the thermostat.

```
public void run(){
   ShieldedPlug database = ShieldedPlug.getDatabase(Forecast.ID);
   while (isRunning) {
      //reading the temperature every 30 seconds
      //and update thermostat status
      try {
         int temp = database.readInt(Forecast.TEMP);
         print(temp);
         //update the thermostat status
         database.writeInt(Forecast.THERMOSTAT,temp>tempLimit ? 0 : 1);
      }
      catch(EmptyBlockException e){
         print("Temperature not available");
      }
      sleep(30000);
   }
}
```

USEFUL LINKS

- Documentation:
  - https://docs.microej.com/en/latest/PlatformDeveloperGuide/sp.html
- Specification:
  - http://e-s-r.net/download/specification/ESR-SPE-0014-SP-2.0-A.pdf
- Javadoc:
  - https://repository.microej.com/javadoc/microej_5.x/apis/ej/sp/package-summary.html
- Shielded Plug Example:
  - https://github.com/MicroEJ/Example-Standalone-Java-C-Interface/tree/master/ProducerConsumerUsingShieldedPlug

API SELECTION CRITERIA

Implementation	Pros	Cons
UI Pump (C → Managed Code)	• Framework already implemented (thread and events already existing).	• Too much events in the UI Pump affects UI responsiveness.
Event Queue (C → Managed Code)	• Alternative to UI pump when many events are received (runs in a dedicated thread)	
Shielded Plug (SP) (Managed Code → C & C → Managed Code)	• Data oriented pump (can be generated from XML description).	• No buffering of data, more suitable for sampling scenarios where losing a value is not critical (e.g. sensors data). • Extra data copy compared to SNI-based solution relying on a shared memory area. In case of SP, the data is copied to the SP database block.
Custom SNI implementation (Managed Code → C & C → Managed Code)	• Completely customized to specific use cases.	• Has to be implemented from scratch.

Appendix

FAQ

- Is the custom SNI implementation the fastest to communicate from Managed Code to C world?
  - Yes, SNI is a generic purpose mechanism, a custom implementation would allow to stick to the use case and get the better performances.
- When calling 2 SNI functions one after another, whether the 2nd one will be executed after the 1st one completes or will it be ignored?
  - The SNI functions will be called one after another.
  - Note: MicroEJ Core Engine cannot preempt a thread that executes a native method. Therefore a blocking native method will prevent the execution of other threads / SNI functions. To mitigate the contention, a native method must explicitly yield its current use of the processor. See Implement a Blocking Java Native Method with SNI for more information.
- Is there a recommendation in terms of number of SNI functions used in a project?
  - No, it is project dependent.
  - The software architecture of the project should avoid too many back and forth between Managed Code and C, to limit the overhead brought by SNI calls.

THANK YOU

for your attention !

MICROEJ®