Application Note:
TLT-0629-AN-MICROEJ-SNIGPIO

Connecting GPIOs Using SNI

In relation to: MICROEJ products

Features

This Application Note explains how to access General Purpose I/O (GPIO) pins from a Java Platform (JPF) using the Simple Native Interface (SNI) feature.

Description

This Application Note assumes the reader wishes to understand the steps involved in adding GPIO support to a JPF using SNI. It explains in detail all the steps required to build the platform and test it with a simple application that controls GPIO pins that are connected to LEDs on a STM32x0G-EVAL evaluation board.

Table of Contents

List of Figures

1 Introduction

1.1 Intended audience

The intended audience for this Application Note are developers who wish to add GPIO support to a Java Platform, using the Simple Native Interface.

1.2 Scope

This Application Note describes the steps required to add GPIO support to an existing Java Platform that was built for an STM3220G-EVAL or STM3240G-EVAL board.

1.3 Prerequisites

This Application Note assumes that the reader has already created a Java Platform, as described in the *TLT-0625-AN-CM_ARMCC-FromScratch* Application Note. All the prerequisites specified in that Application Note apply equally to this Note.

2 SNI overview

The Simple Native Interface (SNI) provides a mechanism for Java applications to call C functions. A Java method that is implemented by a C function must :

- be declared as `static`
- be declared as `native`
- only use primitive types for its parameters and return value

A Java native method has no body. It is mapped to a C function whose name is `Java_` followed by the fully qualified name of the method in which “.” is replaced by “_”. We will see examples of this later.

Note that the MicroEJ Java Platform created by following the steps described in the *TLT-0625-AN-CM_ARMCC-FromScratch* Application Note supports SNI without any further modifications ; SNI is a standard feature of any MicroEJ Java Platform.

For more information about SNI see the *Simple Native Interface for Green Thread Context Profile Specification* (ESR-SPE-0012-SNI-GT).

3 An outline of the required steps

To add GPIO support:

1. Create the Java application that will test the JPF

   This is implemented by defining and using `native` methods which will be implemented in C later.

2. Build the Java application binary file

   The Java project is built in the MicroEJ environment, targeting the embedded JPF (“embJPF”), to produce a binary object file that can be linked by µVision.

3. Write the required C source code to implement the `native` methods.

4. Configure the Keil µVision project

   The properties of the µVision project are adjusted to include the C source code that implements the `native` methods.

5. Build and deploy the C project

   The C project is built and deployed to the target board using the ST-LINK connection.

4 The steps in detail

4.1 Create the Java application that will test the JPF

To reduce typing, the Java application is provided with this Application Note as a ready-to-import MicroEJ project. Import it using :

File → **Import...** → **General** → **Existing Projects into Workspace**

click on **Next**, select **Select archive file**, click on **Browse** and select the `GPIOsExampleApp.zip` archive file, which is provided with this Note.

Three classes are provided: `GPIOsExample`, `GPIOs` and `GPIOsNatives` :

- `GPIOsNatives` defines the native methods; these will be implemented as C functions.
- `GPIOs` provides an API to use the methods defined by `GPIOsNatives`. The separation of the API from the natives means that the API class can provide a richer API than the simple native methods.
- `GPIOsExample` is the class that defines the main method. It is implemented by calling the methods provided by `GPIOs`.

In the MicroEJ **Package Explorer**, browse to the `GPIOsExample` Java class, which looks like this :

```
package com.is2t.examples.gpios;

import com.is2t.gpios.GPIOs;

/**
 * Main class of the GPIO example.
 * Creates tasks that toggle the GPIO pins, with each available pin
 * having a period twice the preceding pin.
 */
public class GPIOsExample {
    public static void main(String[] args) {
        int count = GPIOs.getGPIOsCount();

        int halfPeriodInMilliseconds = 60;
        for (int gpioId = 0; gpioId < count; gpioId++) {
            GPIOs.blinkGPIO(gpioId, halfPeriodInMilliseconds);
            halfPeriodInMilliseconds = halfPeriodInMilliseconds * 2;
        }
    }
}
```

This application calls methods on the `GPIOs` Java class (which will be shown later) to make each GPIO “blink” - that is, turn on and off repeatedly - with increasing period.

Now browse to the `GPIOs` Java class. We will discuss the `blinkGPIO` and `toggleGPIO` methods only :

```
/**
 * Blinks a GPIO at fixed rate.
 *
 * @param id the index of the GPIO to blink
 * @param delay the delay between two toggles
 * @throws IllegalArgumentException if the given id is not valid or
 *                  if the GPIO is already blinking
 */
public static void blinkGPIO(final int id, long delay) throws
 IllegalArgumentException{
    if (id < 0 || id >= getGPIOsCount()) {
        throw new IllegalArgumentException();
    }
    if (isBlinking[id]) {
        throw new IllegalArgumentException();
    }
    isBlinking[id] = true;
    TimerTask task = new TimerTask() {
        public void run() {
            toggleGPIO(id);
        }
    };
    timer.scheduleAtFixedRate(task, 0, delay);
}

private static void toggleGPIO(int id) throws IllegalArgumentException{
    int resultOfToggleGPIO = GPIOsNatives.toggleGPIO(id);
    if (resultOfToggleGPIO == GPIOsNatives.INVALID_ID) {
        throw new IllegalArgumentException(id + " is invalid");
    }
}
```

This `blinkGPIO` method is implemented using a Java `Timer` class ; it calls `toggleGPIO` to perform the GPIO toggling. The `toggleGPIO` method is implemented as a call to the native method `GPIOsNatives.toggleGPIO`.

Now browse to the `GPIOs` Java class and look at the `toggleGPIO` method :

```
/**
 * Toggles the state of a GPIO.<br>
 * Requires that {@link #initGPIOs()} is called before.
 *
 * @param id the index of the GPIO to toggle
 * @return {@link #INVALID_ID} if the given id is invalid
 */
/* package */ static native int toggleGPIO(int id);
```

This is the native method; it acts as a placeholder so that Java code can compile against this method while the implementation itself will be in C.

Part of the value added by the `GPIOs` class is to convert errors returned by the native methods to Java exceptions – native methods cannot throw exceptions.

Note that the native methods do not need to be public; in fact they could even be private (but could then only be called by methods defined in the same class). In this example, the native methods are package visible, so they can only be called by the `GPIOs` class, which provides the API for use by other Java code. The point of restricting the access to the native methods, and only providing access through the API class, is to prevent incorrect use of the native methods. For example, the `initGPIOs` method should only be called once. Restricting access to native methods to only the package that defines the API class reduces the possibility of them being called erroneously, because they cannot be called by Java code in any other package. If the API class is the only other class in the same package then, provided the API is correctly implemented, users of the API cannot call the native methods erroneously.

4.2 Build the Java application binary file

The next step is to build the Java application into a binary file that can be linked with the C parts of the platform. Note that this can be done without an implementation of the native methods – they will be needed when compiling the application in µVision but not for building the Java application.

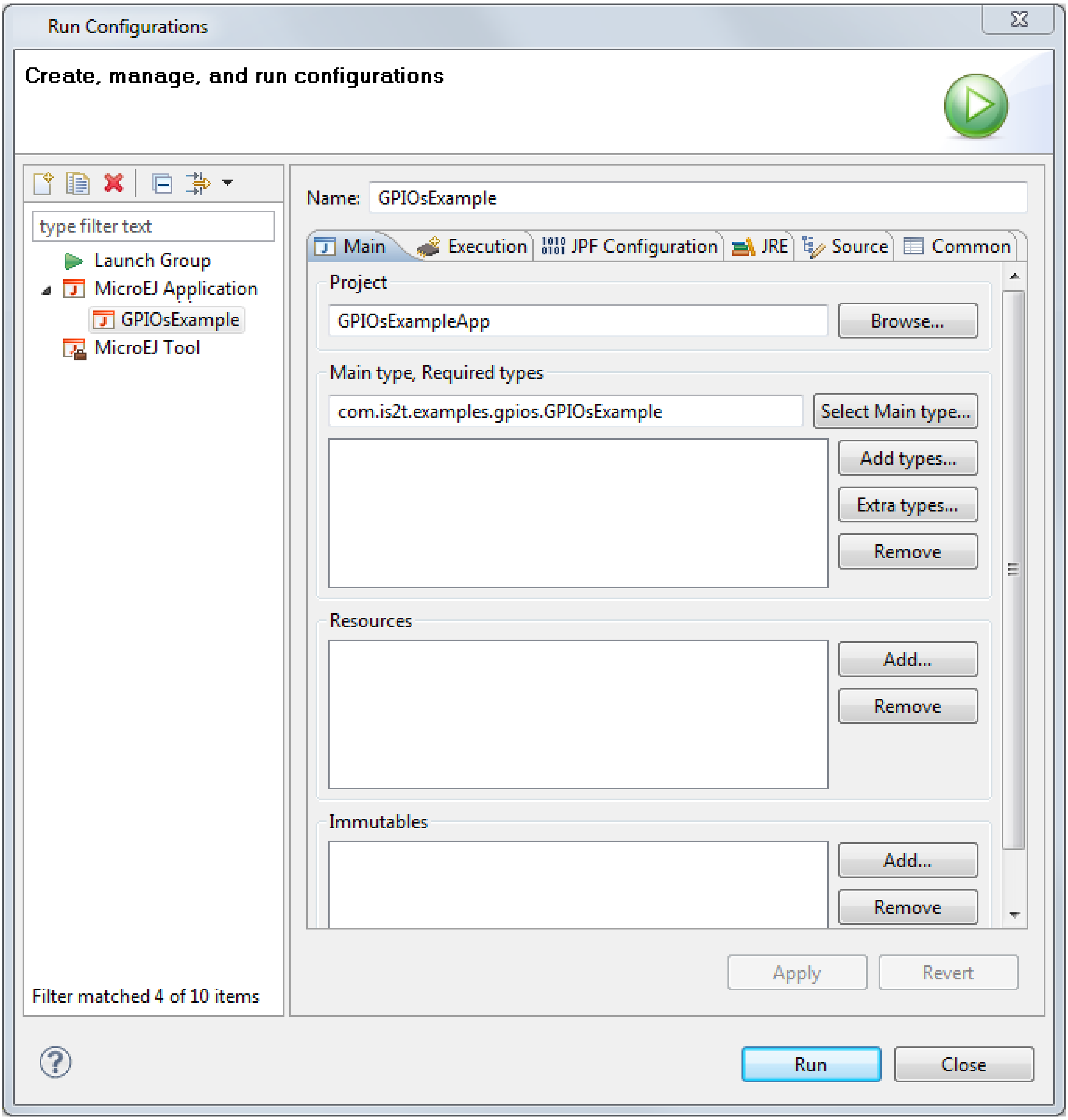
To build the application a suitable *launch configuration* must be created.

Right-click in the **Package Explorer** on the `GPIOsExample` Java class, and select :

Run As → **Run Configurations...**

The **Run Configurations** dialog will open. Double-click the **MicroEJApplication** entry in the list to the left of the dialog to create a new configuration.

The details in the **Main** tab will be entered already, and should look like this :



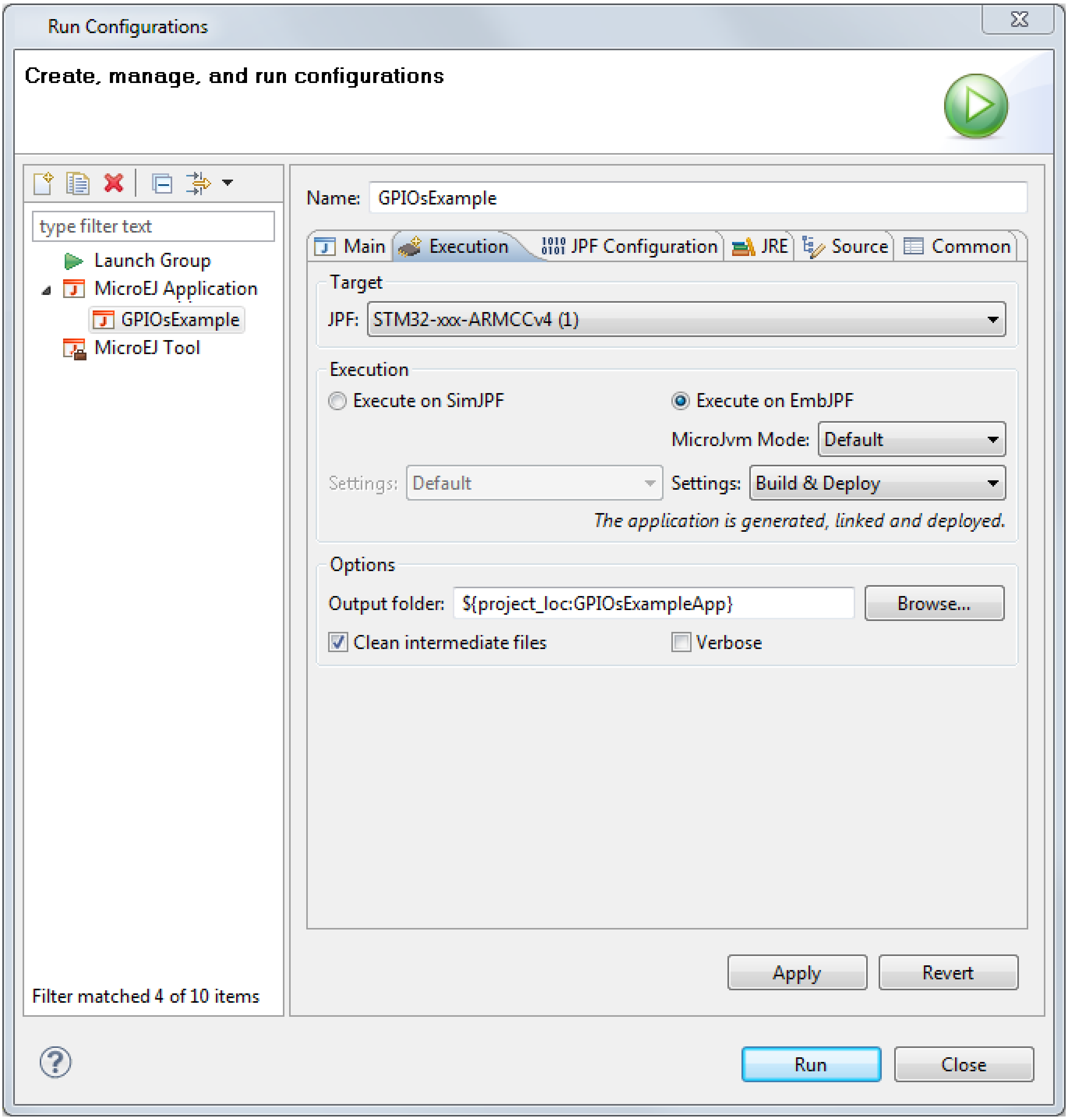
Name: GPIOsExample

Project: GPIOsExampleApp

Main type, Required types: com.is2t.examples.gpios.GPIOsExample

Figure 4.1. Launch Configuration - Main tab

Select the **Execution** tab. Check the **Execute on EmbJPF** option, as shown below :



Name: GPIOsExample

JPF: STM32-xxx-ARMCCv4 (1)

○ Execute on SimJPF

● Execute on EmbJPF

MicroJvm Mode: Default

Settings: Default

Settings: Build & Deploy

The application is generated, linked and deployed.

Output folder: ${project_loc:GPIOsExampleApp}

☑ Clean intermediate files

☐ Verbose

Figure 4.2. Launch Configuration - Execution tab

Press **Run**. The application is built into a binary file called `SOAR.o`, located in the `com.is2t.example.GPIOsExample` folder of the `GPIOsExampleApp` project.

4.3 Write the required C code

We need to implement C functions with names that correspond to the names of the native methods in Java. As mentioned earlier, the C function name is `Java_` followed by the fully qualified name of the native method in which “.” is replaced by “_”.

For example, consider the native method `initGPIOs` in `com.is2t.gpios.GPIOsNatives`. The C function that corresponds to this is called `Java_com_is2t_gpios_GPIOsNatives_initGPIOs`.

The C code to implement the native methods for this example is included in this Application Note as the file `gpio.c`, which must be copied to the `FromScratch` C project folder, alongside the other C source files. Here is the `Java_com_is2t_gpios_GPIOsNatives_toggleGPIO` function :

```
/**
 * @brief  To toggle output
 * @param  Output ID
 * @retval OK or invalid ID
 */
int Java_com_is2t_gpios_GPIOsNatives_toggleGPIO(int id) {
    if (id >=  GPIO_COUNT) return INVALID_ID;
    GPIO_ToggleBits((GPIO_TypeDef*) GPIO_PORT_ID[id], GPIO_ID[id]);
    return OK;
}
```

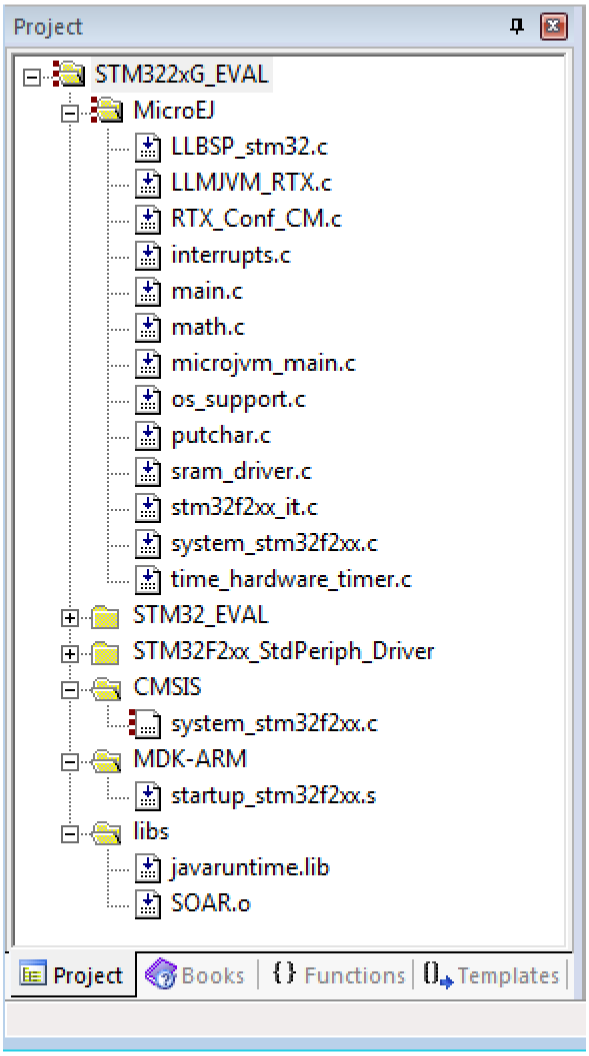
The name of this function corresponds to the native method `toggleGPIO` in `com.is2t.gpios.GPIOsNatives`. The implementation uses the `GPIO_ToggleBits` function provided by `stm32f2xx_gpio.c` (or `stm32f4xx_gpio.c` as appropriate).

Note that the GPIOs used in this implementation control the LEDs as well as pins. The GPIO `id` parameter of `toggleGPIO` maps to :

- id 0: LED1 (PG6, accessible on CN3 pin 24)
- id 1: LED2 (PG8, accessible on CN3 pin 22)
- id 2: LED3 (PI9, accessible on CN1 pin 12)
- id 3: LED4 (PC7, accessible on CN3 pin 20)

4.4 Configure the Keil µVision project

The Keil µVision project file `Project.uvproj` in the `FromScratch/MDK-ARM` folder must be changed to add the extra source files. Double-click the `Project.uvproj` file to open it in the µVision environment. The contents of the project will look like this (if it is the one created by following the steps described in the *TLT-0625-AN-CM_ARMCC-FromScratch* Application Note) :



- STM322xG_EVAL
  - MicroEJ
    - LLBSP_stm32.c
    - LLMJVM_RTX.c
    - RTX_Conf_CM.c
    - interrupts.c
    - main.c
    - math.c
    - microjvm_main.c
    - os_support.c
    - putchar.c
    - sram_driver.c
    - stm32f2xx_it.c
    - system_stm32f2xx.c
    - time_hardware_timer.c
  - STM32_EVAL
  - STM32F2xx_StdPeriph_Driver
  - CMSIS
    - system_stm32f2xx.c
  - MDK-ARM
    - startup_stm32f2xx.s
  - libs
    - javaruntime.lib
    - SOAR.o

Figure 4.3. µVision Project Content Before Configuration

The actual files in the User group may vary.

Several different configuration actions are required :

- Add the extra C file

  Right-click on the **User** group of the project, select **Add Files to Group 'User'...** and select the file `gpio.c` that was provided with this Application Note.

- Remove the FromScratch application

  Right-click on the `SOAR.o` file and select **Remove File ....**

- Add the GPIOExamples application

  Right-click on the **MicroEJ** group and select **Add Files to Group 'MicroEJ'....** Navigate to the `com.is2t.example.GPIOsExample` folder of the `GPIOsExampleApp` application, and select the `SOAR.o` file. It may be necessary to select **Files of type: All files (*.*)** in the dialog box to see the file. Press **Add**. If µVision asks for the type of the file, select **Object file**.

4.5 Build and deploy the C project

The C code should now compile cleanly. Build it using **Project** → **Build target** (**F7**). It can now be downloaded to the target board using the ST-LINK connection. Reset the board to run the application and the relevant GPIO pins will toggle (and the associated LEDs will blink).

5 Document History

Date	Revision	Description
February 19th 2013	A	First release
February 25th 2013	B	Minor improvements to the text
November 12th 2013	C	MicroEJ 2.0.0 compatibility
July 11th 2014	D	MicroEJ 3.0.0 compatibility

Headquarters
11, rue du chemin Rouge
44373 Nantes Cedex 3
FRANCE
Phone: +33 2 40 18 04 96
www.is2t.com