Application Note:
TLT-0651-AN-MICROEJ-Scheduler

Scheduler-Based Application: Mixing Java and C

In relation to: MICROEJ products

Features

This Application Note explains how to port a scheduler for a C program to a Java Platform using the MicroEJ® environment and the ST Standard Peripherals Library and how to build it using Keil µVision.

Description

This Application Note assumes the reader wishes to understand the steps involved in creating a Java Platform and reuse C code to schedule tasks in a Java application. It explains in detail how to port a simple scheduler from a C program to a mixed C and Java program.

Table of Contents

List of Figures

1 Introduction

1.1 Intended audience

The intended audience for this Application Note are developers who wish to port a simple scheduler implemented in C to a program that mixes Java and C code.

1.2 Scope

This Application Note describes the steps required to port a simple scheduler implemented in C to a program that mixes Java and C code. It is an extension of the approach described in the *TLT-0649-AN-CM_ARMCC-STM32x0G-EVAL-BareMetalJava* application note.

No operating system (OS) will be used to execute the MicroJvm® virtual machine in this Application Note. Readers who wish to use an OS should start from the *TLT-0625-AN-CM_ARMCC-STM32x0G-EVAL-FromScratch* Application Note instead of the *TLT-0649-AN-CM_ARMCC-STM32x0G-EVAL-BareMetalJava* Application Note. Whether the virtual machine runs inside the main C function or inside a OS task does not change how the Java application works or how to port the scheduler.

1.3 Prerequisites

This Application Note assumes that the reader has already created a Java Platform without an operating system, as described in the *TLT-0649-AN-CM_ARMCC-STM32x0G-EVAL-BareMetalJava* Application Note. All the prerequisites specified in that Application Note apply equally to this Note.

2 The C scheduler to be ported

This Application Note comes with a set of C files containing a simple C scheduler. These files replace some of the files of the C project created in the *TLT-0649-AN-CM_ARMCC-STM32x0G-EVAL-BareMetalJava* application note, to modify it.

Here is a brief description of these files:

- `leds.c` / `leds.h` : contain functions to use the LEDs available on a STM3220G-EVAL (or STM3240G-EVAL) board.
- `tasks.c` / `tasks.h` : contain functions that can be scheduled. They are “cooperative”, meaning that they don't do infinite loops, use blocking functions, etc. They perform a part of their job, return, and can resume their work next time they are called.
- `main.c` : contains the original C program (commented) and the new C program (uncommented). The original program has an infinite loop and calls tasks sequentially. The new program launches the virtual machine and implements native methods used by the Java application. In both versions, tasks are configured in the same `config_tasks.h` file.
- `config_tasks.h` : contains the binding of tasks to the functions that implement them, using a `#define` pre-processor command. The real functions can be in various files as long as the correct header files are included. Here, they are all gathered in tasks.c hence only `tasks.h` needs to be included.

3 An outline of the required steps

1. Create the Java application to test the JPF.

   The Java application now schedules the tasks, reusing the C code of the tasks.

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 Keil µVision.

3. Add the new files to the Keil µVisions project folder.

   The files supplied with this Application Note are added to the C project.

4. Configure the Keil µVision project.

   The properties of the µVision project are modified to include to the files and the Java application.

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

4.1.1 Principle of the Java application

Our goal is to port this scheduler, fully defined in C, to a Java application that will reuse existing C code.

To achieve our purpose, we have to launch the MicroJvm® virtual machine and to perform the scheduling inside the Java world and call our existing C implemented tasks. Hence, the infinite loop will rather be in the Java world than in the C world and this loop will access the C functions through Java methods defined as native. Native methods in Java are methods declared and used in the Java world but their implementation is in the C world. When the Java application is generated by MicroEJ's compiler and linker it contains unresolved symbols corresponding to the native methods. Keil's linker will resolve those symbols with the functions implemented in C using a defined naming convention.

The capability of calling C functions from Java code is part of the SNI library and is consequently defined in the SNI specification. To fully understand this mechanism please read that document.

The file `main.c` contains both implementations. You can switch between the C and Java versions by commenting / uncommenting the appropriate macro : `#define USE_JAVA_VERSION`.

4.1.2 Create the Java Project

Create a new MicroEJ® Java Project. Do this by selecting:

File → New → Java Project

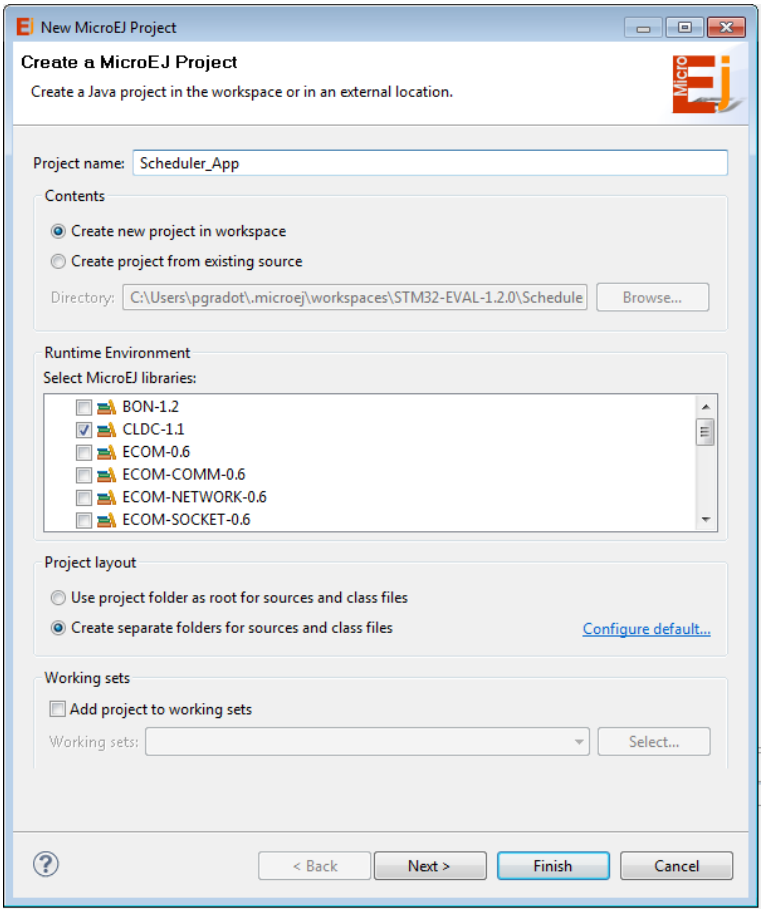

Figure 4.1. Creating the Java Project

Enter the project name **Scheduler_App** and leave all the other settings unchanged. On pressing **Finish** a new project is created.

4.1.3 Create the main class

Right-click on the newly-created `Scheduler_App` project and select:

New → Class

Create a class called **Scheduler** in the package **com.is2t.examples**, and tick the box asking for a main method to be created:

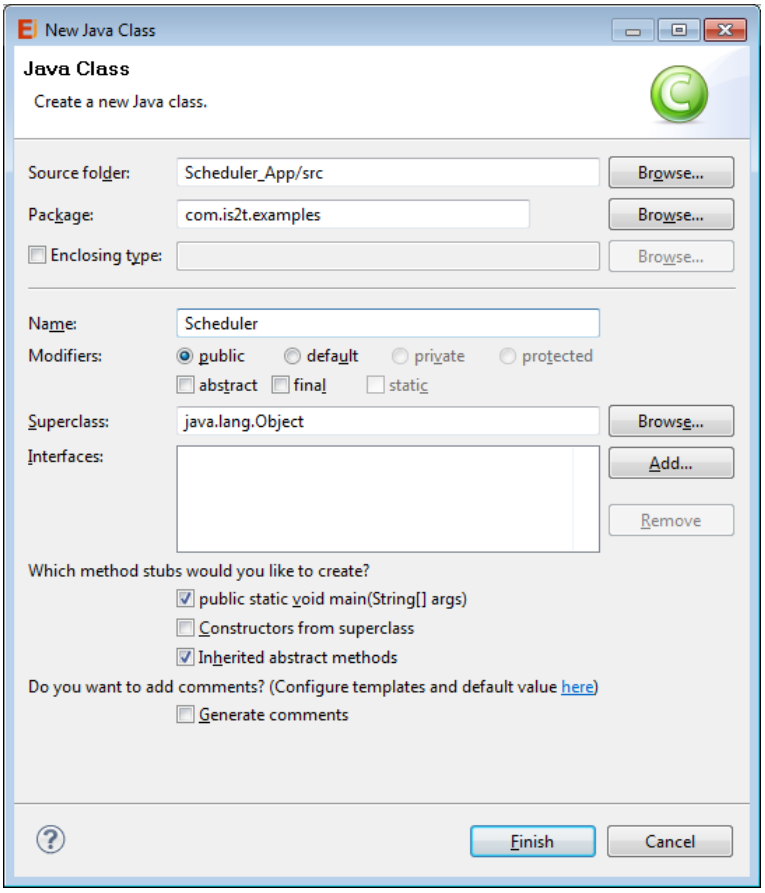

Figure 4.2. Creating the main class

Please note that the package name is important for the naming convention of SNI. Fill in the Java code of the application, either copying the code shown below or with the Java file provided with this Application Note:

```
package com.is2t.examples;

public class Scheduler {
    // Native functions --> implementation is done in C code
    private native static void Task1();
    private native static void Task2();
    private native static void Task3();
    private native static void Task4();
    private native static void Task5();


    // Java functions
    private static int Un = 15;
    private static boolean TrivialCyleReached = false;
    private static void SyracuseSequence() {

        if(TrivialCyleReached == false) {  // don't print if trivial cycle has been
 reached
            System.out.println("Syracuse : " + Un);
        }

        if(Un % 2 == 0) {
            // Even
            Un = Un /2;
            if(Un == 1) {
                TrivialCyleReached = true;
            }
        }
        else {
            // Odd
            Un = 3*Un + 1;
        }
    }

    // Main function schedules both C and Java tasks
    public static void main(String[] args) {
        System.out.println("Java scheduling starts...");

        while(true) {
            Task1();
            Task2();
            Task3();
            Task4();
            Task5();
            SyracuseSequence();
        }
    }
}
```

An extra task is added. It is fully implemented inside the Java world and can be scheduled the same way as native tasks. This shows how a ported scheduler can mix C and Java tasks.

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.

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

Right-click in the **Package Explorer** on the Java class created in the previous step, 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:

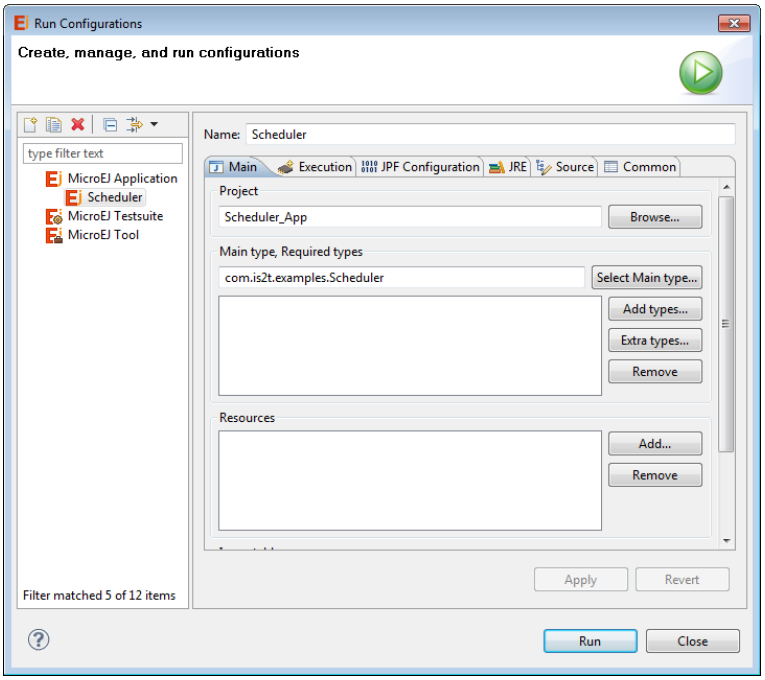

Figure 4.3. Run Configuration - Main tab

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

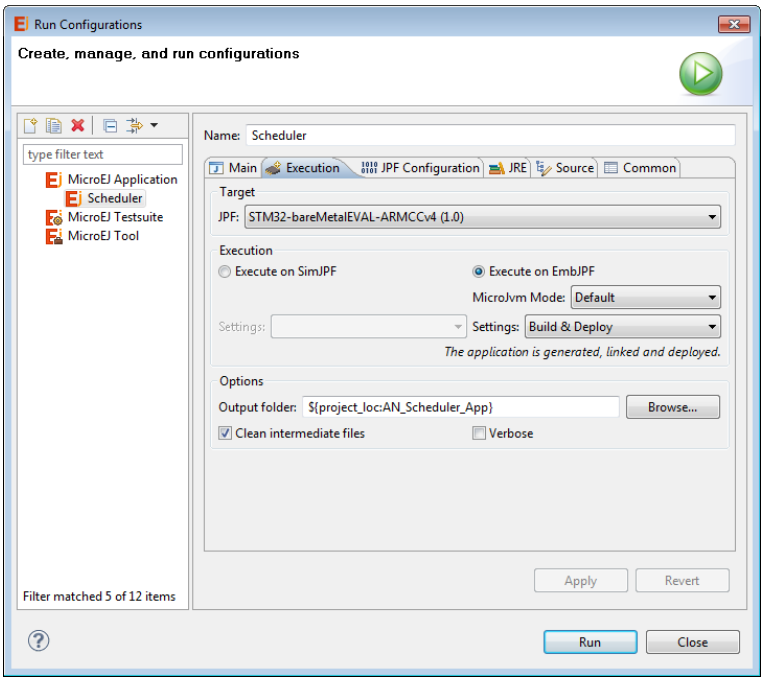

Figure 4.4. Run Configuration - Execution tab

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

4.3 Add the new files to the Keil µVision project folder

Copy and paste the project folder created in the *TLT-0649-AN-CM_ARMCC-STM32x0G-EVAL-BareMetalJava* Application Note. Rename the new folder `Scheduler`.

Copy the provided files and replace existing files with the same name (e.g, : `main.c`).

The folder content should now look like this (the actual set of files may be different):

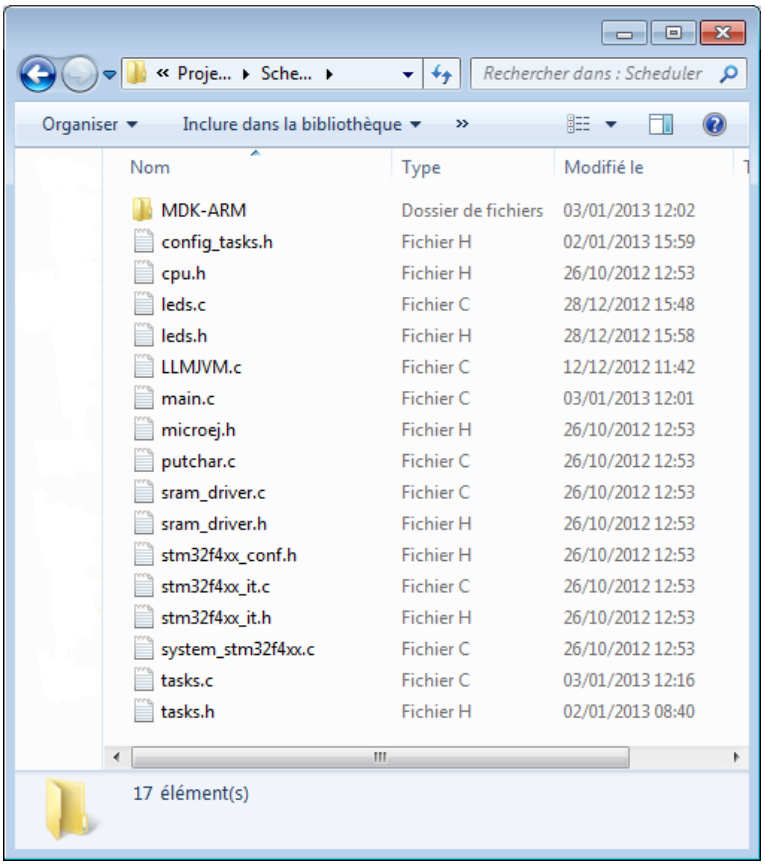

Figure 4.5. C file list

4.4 Configure the Keil µVision project

The file Project.uvproj in the `MDK-ARM` folder is a Keil µVision project definition. It has been configured to suit the original contents of the template Standard Peripherals Library project and must be changed to suit the `Scheduler` requirements.

Double-click the `Project.uvproj` file to open it in the Keil µVision environment.

We will first add the new C files to the project:

- In the **Project** window, right-click on the **User** group.
- Select **Add Files to Group 'User'....**
- Select all the new C files contained in the project folder.
- Click on **Add** and then on **Close**.

We also need to change the Java application object file:

- In the **Project** window, right-click on the file `com.is2t.example.SOAR.o` (in the **MicroEJ** group).
- Select **Remove file com.is2t.example.SOAR.o** and validate.
- In the **Project** window, right-click on the **MicroEJ** group.
- Select **Add Files to Group 'MicroEJ'....**
- Navigate to the `com.is2t.example.Scheduler` folder of the `Scheduler_App` application.

- Select the `com.is2t.example.SOAR.o` file and add it. It may be necessary to select **Files of type: All files (*.*)** in the dialog box to see the file. If Keil 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. LEDs will blink and the following output will appear in the terminal emulator connected to the serial port :

```
START
VM START
Java scheduling starts...
1th month : 1 couples
Syracuse : 15
2th month : 1 couples
Syracuse : 46
3th month : 2 couples
Syracuse : 23
4th month : 3 couples
Syracuse : 70
5th month : 5 couples
Syracuse : 35
6th month : 8 couples
Syracuse : 106
7th month : 13 couples
Syracuse : 53
8th month : 21 couples
Syracuse : 160
9th month : 34 couples
Syracuse : 80
10th month : 55 couples
Syracuse : 40
11th month : 89 couples
Syracuse : 20
12th month : 144 couples
Syracuse : 10
13th month : 233 couples
Syracuse : 5
14th month : 377 couples
Syracuse : 16
15th month : 610 couples
Syracuse : 8
16th month : 987 couples
Syracuse : 4
17th month : 1597 couples
Syracuse : 2
18th month : 2584 couples
19th month : 4181 couples

(…)

45th month : 1134903170 couples
46th month : 1836311903 couples
47th month : 2971215073 couples
```

5 Document History

Date	Revision	Description
January 2nd, 2013	A	First release
May 13th, 2013	B	Second release : correct minor typographical errors, modify `main.c` file to run C or Java version easily.

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