Application Note:
TLT-0649-AN-MICROEJ-BareMetalJava

Building a Bare Metal Java Platform

In relation to: MICROEJ products

Features

This Application Note explains how to create a Java Platform (JPF) to run a multi-threaded Java application without an Real Time Operating System (RTOS). Software development tools involved in this Application Note are the MicroEJ® environment, the ST Standard Peripherals Library, and the Keil µVision SDK.

Description

This Application Note explains in detail all the steps required to build the platform and to test it with a simple application using several threads that output text strings over a serial port.

The build process of the JPF and of the Java application are similar to the one described in the *TLT-0625-AN-MICROEJ_FromScratch* Application Note. The main difference is that this Note highlights the way the MicroJvm® virtual machine is handled by the native code (C language).

Table of Contents

List of Figures

1 Introduction

1.1 Intended audience

The intended audience for this Application Note are developers who wish to interface a JPF to an existing native board support package. In this document, we will use the STM32Fxxx Standard Peripherals Library as board support package.

1.2 Scope

This Application Note describes the steps required to build a full custom JPF for an STM3220G-EVAL or STM3240G-EVAL board “from scratch” – meaning that the required steps are performed manually rather than using one of the scripts supplied with the environment.

No RTOS will be used to execute the MicroJvm® virtual machine and application threads. Instead, the virtual machine is able to run and schedule multiple threads, called "Java threads": this configuration is often called a “green thread” architecture.

This Application Note does not explain the contents of the C files in the ST Standard Peripherals Library nor the details of starting the MicroJvm virtual machine. Please consult the SNI Specification for complete information. Nonetheless, a discussion on the implementation of the Low Level MicroJvm API and the C main function are included.

1.3 Prerequisites

The environment for MicroEJ® (MICROEJ-PKG-STD-MicroEJ-3.1.0 or later) must be installed and the required license activated.

Keil µVision version 4.54 or later must be installed, and the appropriate license activated.

An STM3220G-EVAL or STM3240G-EVAL board with the ST-LINK USB port connected to a desktop PC is required to be able to run the application note examples. An RS-232 cable needs to be connected from the 9-pin connector CN16 to a suitable terminal emulator so that output from the UART can be captured and displayed on a terminal emulator.

The appropriate Standard Peripherals Library must be downloaded from the ST web site. For the STM3220G-EVAL board the appropriate library can be found from ST Microelectronics web site.

This document is based on version 1.1.0 of the stm32f2 library, and version 1.0.1 of the stm32f4 library.

2 An outline of the required steps

The following steps give an overview of the JPF build process :

1. Create the Java Platform (JPF).

   A JPF is created within the environment.

2. Create the Java application to test the JPF.

   A Java project is created within the environment and the demonstration code is written.

3. Build the Java application binary file.

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

4. Create a file structure to hold the C code.

   The template project provided within the STM32Fxxx Standard Peripherals Library (SPL) is used to create a project to hold the C code for the application.

5. Add the MicroJvm support files to the C project.

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

6. Configure the Keil µVision project.

   The properties of the µVision project supplied with the SPL are adjusted so that it includes all the required sources and libraries, and refers to the correct header files.

7. Build and deploy the C project.

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

3 The steps in detail

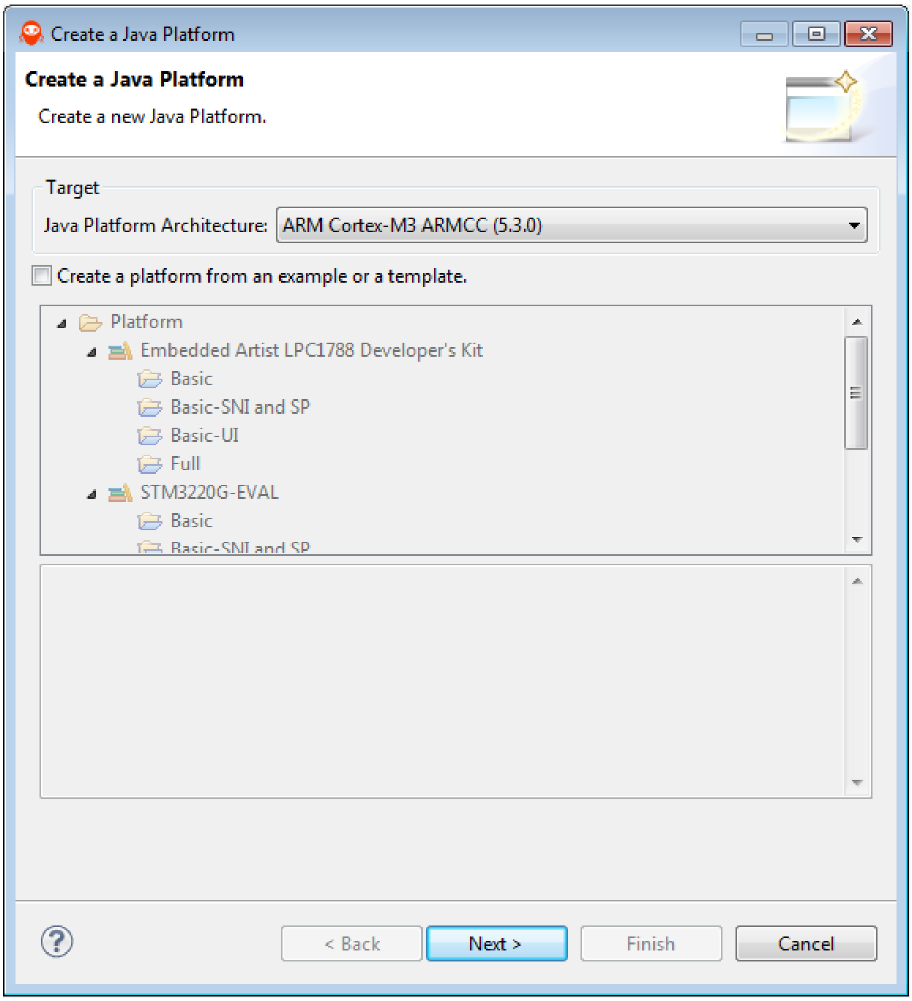
3.1 *Create the Java Platform*

A Java platform (JPF) comprises the MicroJvm virtual machine itself plus supporting libraries and tools. A JPF is suitable for use on a specific core and toolchain depending on the chosen architecture.

3.1.1 Create the Java Platform configuration

The first step is to create a JPF Configuration project that will be used to parameter the JPF. In this application note, we will build a JPF compatible with Cortex-M3 MCU. Create the JPF Configuration by selecting, in the MicroEJ environment : **File** → **New** → **Java Platform**

A dialog box appears in order to choose the JPF architecture and to suggest to start from an JPF example. For the application note, select the CORTEX-M3-based JPF architecture. To create a JPF "from scratch", uncheck the option "Create a platform from an example or a template".



Create a Java Platform

Create a new Java Platform.

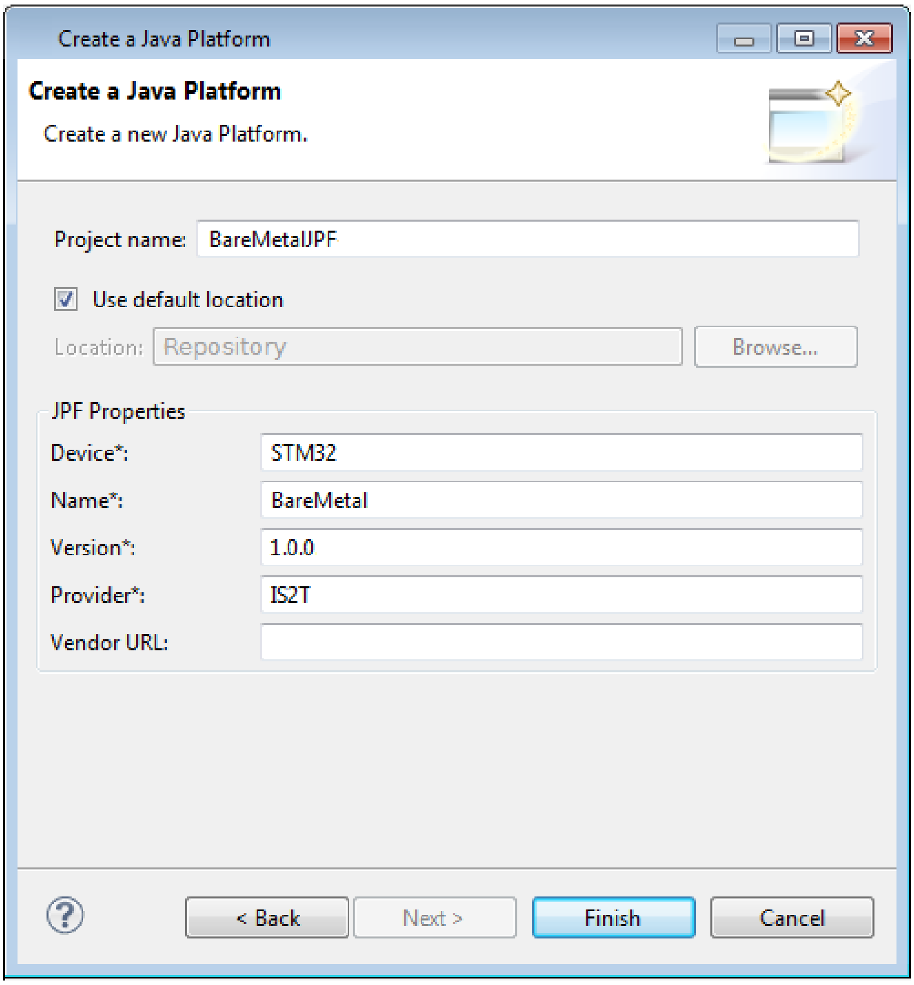
Target

Java Platform Architecture: ARM Cortex-M3 ARMCC (5.3.0)

☐ Create a platform from an example or a template.

Figure 3.1. Java Platform Configuration (architecture selection)

Click on **Next**. JPF creation wizard continues asking to set a name for the future project and properties for the created JPF. Please fill the form like following:



Create a Java Platform

Create a new Java Platform.

Project name: BareMetalJPF

☑ Use default location

Location: Repository

JPF Properties

Device*: STM32

Name*: BareMetal

Version*: 1.0.0

Provider*: IS2T

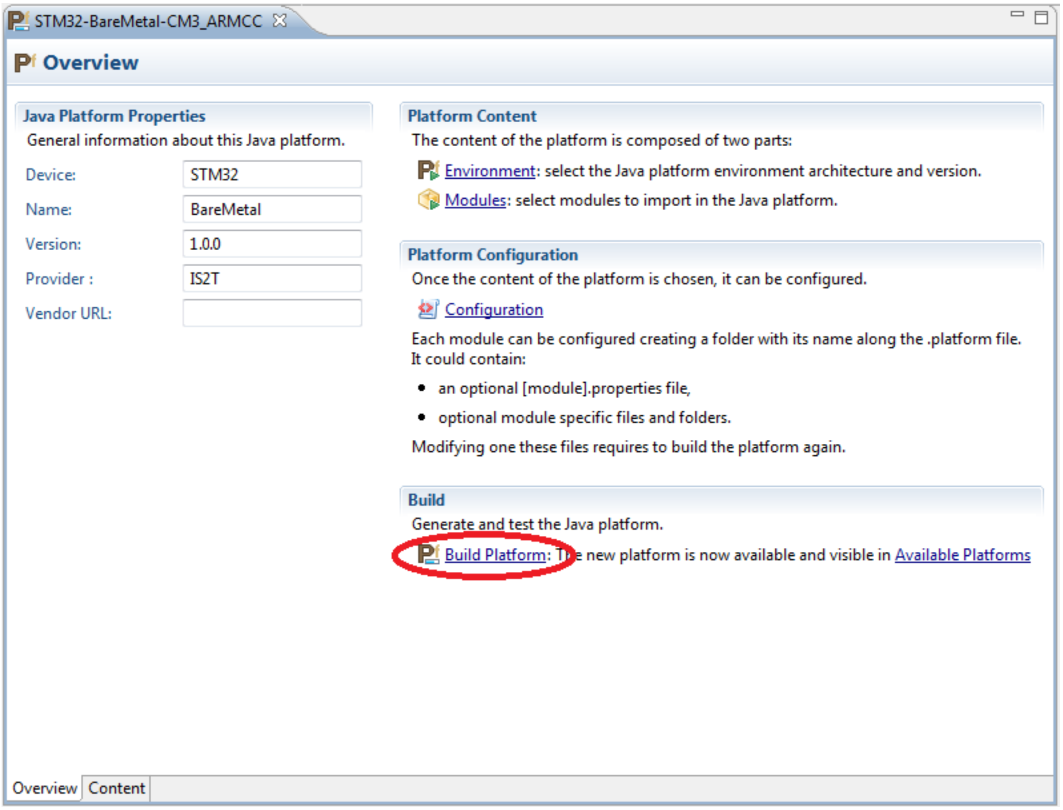
Vendor URL:

Figure 3.2. Java Platform Configuration (properties set)

The provider can be any name you wish. On pressing **Finish**, a new project is created containing the JPF configuration.

3.1.2 Build the Java Platform

The second step is to build the JPF based on the JPF Configuration. Build the JPF by opening the **BareMetal.platform** file created on the JPF Configuration project and selecting the **Build Platform** hyperlink available on right side of the panel:



Java Platform Properties

General information about this Java platform.

Device: STM32

Name: BareMetal

Version: 1.0.0

Provider : IS2T

Vendor URL:

Platform Content

The content of the platform is composed of two parts:

Environment: select the Java platform environment architecture and version.

Modules: select modules to import in the Java platform.

Platform Configuration

Once the content of the platform is chosen, it can be configured.

Configuration

Each module can be configured creating a folder with its name along the .platform file. It could contain:

- an optional [module].properties file,
- optional module specific files and folders.

Modifying one these files requires to build the platform again.

Build

Generate and test the Java platform.

Build Platform: The new platform is now available and visible in Available Platforms

Figure 3.3. Java Platform Configuration (build platform)

The JPF name will be STM32-BareMatel-ARMCCv4[1]. Your workspace should look like the one shown below :

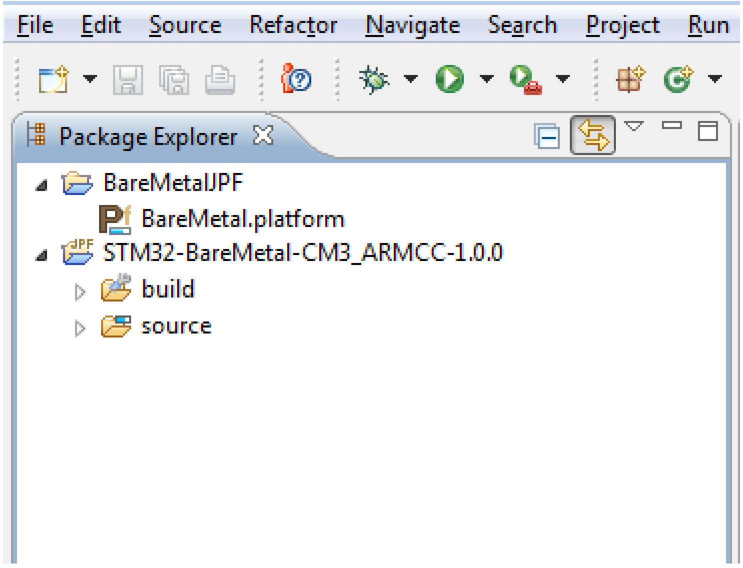


Figure 3.4. MicroEJ workspace

STM32-BareMetal-CM3_ARMCC-1.0.0 project has been created during the build operation ; it is notable by a JPF icon and contains JPF source.

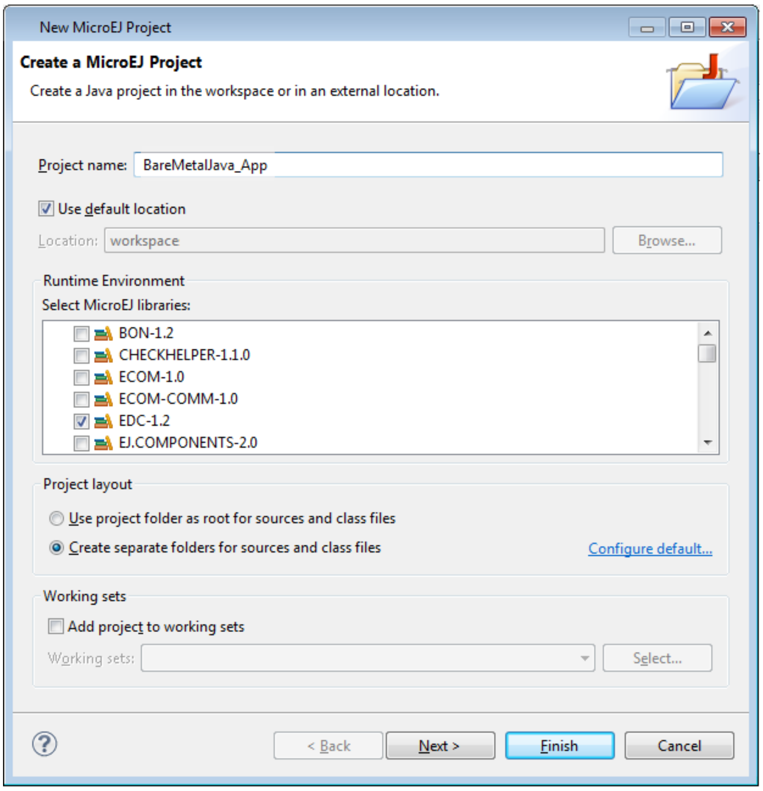
---

[1] This name is created by concatenating the Device and Name entered in the dialog with the name of the toolchain being used.

3.2 Create the Java application that will test the JPF

3.2.1 Create the Java Project

Create a new MicroEJ Java Project. Do this by selecting : **File** → **New** → **Java Project**



Create a MicroEJ Project

Create a Java project in the workspace or in an external location.

Project name: BareMetalJava_App

☑ Use default location

Location: workspace

Runtime Environment

Select MicroEJ libraries:

- ☐ BON-1.2
- ☐ CHECKHELPER-1.1.0
- ☐ ECOM-1.0
- ☐ ECOM-COMM-1.0
- ☑ EDC-1.2
- ☐ EJ.COMPONENTS-2.0

Project layout

- ○ Use project folder as root for sources and class files
- ● Create separate folders for sources and class files

Working sets

☐ Add project to working sets

Working sets:

Figure 3.5. *Create Java Project*

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

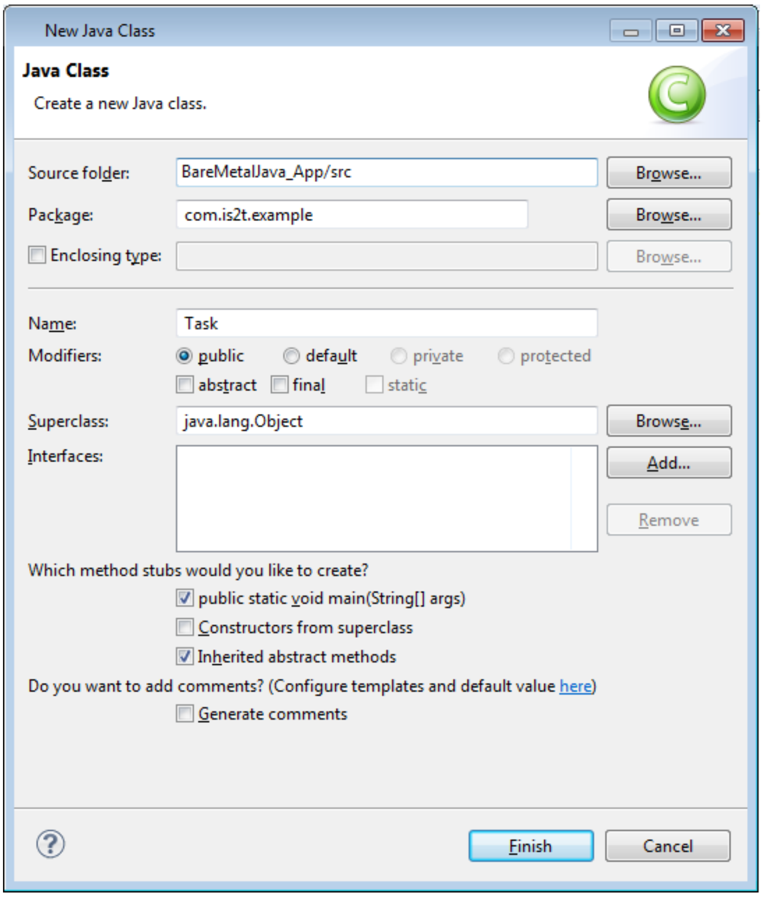
3.2.2 Create the main class

Right-click on the newly-created `BareMetalJava_App` project and select : **New** → **Class**

Create a class called **Task** in the package **com.is2t.example** [2], and tick the box asking for a main method to be created :

---

[2] The name of the package is not relevant. You can name it as you want and adapt the following.



Java Class

Create a new Java class.

Source folder: BareMetalJava_App/src

Package: com.is2t.example

☐ Enclosing type:

Name: Task

Modifiers: ● public ○ default ○ private ○ protected

☐ abstract ☐ final ☐ static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?

☑ public static void main(String[] args)

☐ Constructors from superclass

☑ Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

☐ Generate comments

Figure 3.6. Create Main Class

Fill in the Java code of the application. You can copy the code shown below or use the Java file provided with the Application Note :

```
package com.is2t.example;

public class Task implements Runnable {
    // We need to implement the Runnable interface to create a thread

    private final String text;
    private final int countTo;
    private final int delay; // in milliseconds

    // Constructor
    public Task(String text, int countTo, int delay) {
        this.text = text;
        this.countTo = countTo;
        this.delay = delay;
    }

    // The thread will execute this method
    public void run() {
        System.out.println("Thread " + text + " starts...");

        for(int i=1; i<=countTo; i++ ) {
            System.out.println(text + " : " + i + "/" + countTo);
            try {
                Thread.sleep(delay);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }

        System.out.println("Thread " + text + " ends");
    }

    // Main function, to test the multithreading capability
    public static void main(String[] args) {
        System.out.println("This is a simple application");

        // Create and start threads
        Thread t1 = new Thread( new Task("A", 15, 900) );
        Thread t2 = new Thread( new Task("B", 8, 1600) );
        Thread t3 = new Thread( new Task("C", 30, 200) );

        t1.start();
        t2.start();
        t3.start();

        // Wait for the threads to finish
        try {
            t1.join();
            t2.join();
            t3.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Now, the end");
    }
}
```

3.3 Build the Java application binary file

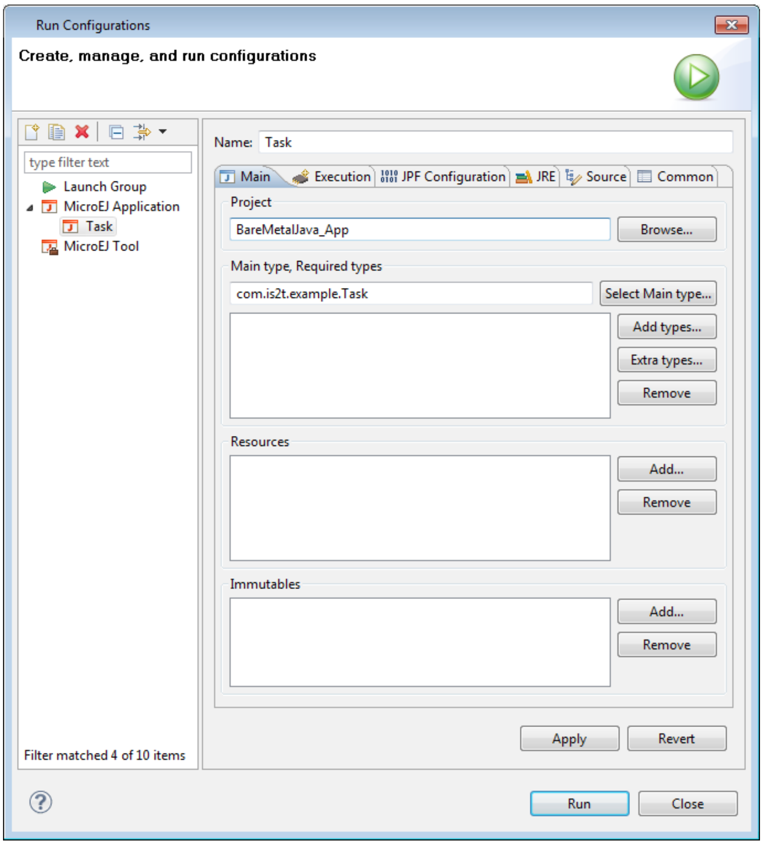
The next step is to build the Java application into a binary file that can be linked with the C parts of the platform.

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 :



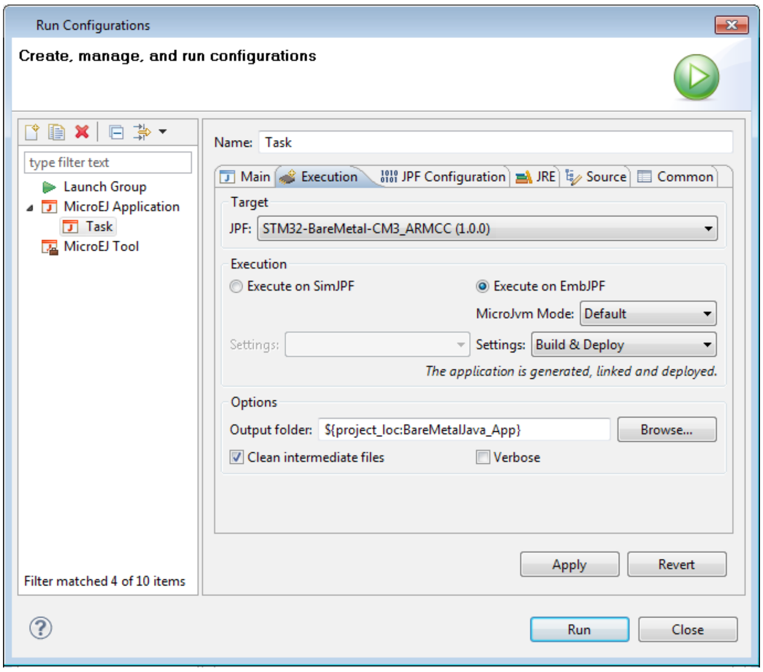
Name: Task

Project: BareMetalJava_App

Main type, Required types: com.is2t.example.Task

Figure 3.7. Run Configuration - Main tab

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



Name: Task

JPF: STM32-BareMetal-CM3_ARMCC (1.0.0)

● Execute on EmbJPF

MicroJvm Mode: Default

Settings: Build & Deploy

The application is generated, linked and deployed.

Output folder: ${project_loc:BareMetalJava_App}

☑ Clean intermediate files

☐ Verbose

Figure 3.8. Run Configuration - Execution tab

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

3.4 Create a file structure to hold the C code

The next step is to create the file structure for the C code project. This structure is based on the STM32Fxxx Standard Peripherals Library (SPL).

Unzip the SPL into a suitable location, where it can be edited in-place – much of it can be deleted for our purposes.

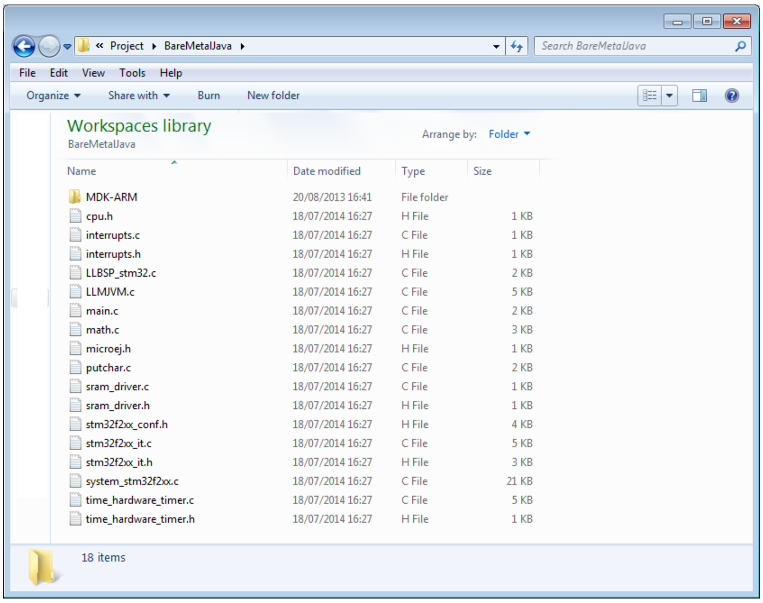
The Project folder of the library contains a template project folder, called `STM32Fxxx_StdPeriph_Templates`, that will form the basis of the project for this example. Copy this folder and rename it to be called `BareMetalJava`. This will become the project folder for the C files.

The template project contains support for several different C development environments. The files for use with Keil µVision are in the `MDK-ARM` folder. Delete all the other sub-folders and files in the folder `BareMetalJava` – they are going to be replaced in the next step.

3.5 Add the MicroJvm support files to the C project

The C project requires a number of files that provide platform-specific support to the MicroJvm virtual machine. The content of some of these files is discussed below. The required files are supplied with this Application Note. Two sets of files are provided; one set for the STM3220G-EVAL and another for the STM3240G-EVAL. Copy all the files from the relevant set into the `STM32Fxxx_StdPeriph_Lib_Vxxx/Project/BareMetalJava` folder (which will become the C project).

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



Name	Date modified	Type	Size
MDK-ARM	20/08/2013 16:41	File folder	
cpu.h	18/07/2014 16:27	H File	1 KB
interrupts.c	18/07/2014 16:27	C File	1 KB
interrupts.h	18/07/2014 16:27	H File	1 KB
LLBSP_stm32.c	18/07/2014 16:27	C File	2 KB
LLMJVM.c	18/07/2014 16:27	C File	5 KB
main.c	18/07/2014 16:27	C File	2 KB
math.c	18/07/2014 16:27	C File	3 KB
microej.h	18/07/2014 16:27	H File	1 KB
putchar.c	18/07/2014 16:27	C File	2 KB
sram_driver.c	18/07/2014 16:27	C File	1 KB
sram_driver.h	18/07/2014 16:27	H File	1 KB
stm32f2xx_conf.h	18/07/2014 16:27	H File	4 KB
stm32f2xx_it.c	18/07/2014 16:27	C File	5 KB
stm32f2xx_it.h	18/07/2014 16:27	H File	3 KB
system_stm32f2xx.c	18/07/2014 16:27	C File	21 KB
time_hardware_timer.c	18/07/2014 16:27	C File	5 KB
time_hardware_timer.h	18/07/2014 16:27	H File	1 KB

Figure 3.9. C file list

3.6 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 `BareMetalJava` requirements.

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

Step1 - We will first add the C files to the project :

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

Step 2 - We also need to add files created with the MicroEJ environment :

- In the **Project** window, right-click on the root of the project.
- Select **Add group....**
- Name it **MicroEJ**.
- Right-click on the **MicroEJ** group.
- Select **Add Files to Group 'MicroEJ'....**
- Navigate to the `com.is2t.example.Task` folder of the `BareMetalJava_App` application.
- Select the `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**.
- Navigate to the `source\lib` folder of the `STM32-BareMetal-CM3_ARMCC-1.0.0`.
- Select the `javaruntime.lib` file, add it and close the window.

Step 3 - Now that all the needed files have been added to the project, we have to configure the include path so that it can compile properly.

- In the Project window, right-click on the root of the project.
- Select **Options for Target 'STM322xG_EVAL'....**
- The **Target** tab is displayed and we can see that **None** is selected in the **Operating System** combo box.
- Go to **C/C++** tab.
- Press the **...** button to the right of the **Include Paths** box.
- Press the **New/Insert** button (the button on the left, in the right upper corner of the new window).
- Press the **...** button of the blank entry.
- Navigate to the `source\include` folder of the `STM32-BareMetal-CM3_ARMCC-1.0.0` project (to find the location of this, in MicroEJ, right click on the folder and select **Properties**).
- Press **OK** to close the configuration window.

3.7 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 output will appear in the terminal emulator connected to the serial port :

```
START
VM START
This is a simple application
Thread A starts...
A : 1/15
Thread B starts...
B : 1/8
Thread C starts...
C : 1/30
C : 2/30
C : 3/30
C : 4/30

(…)

B : 4/8
C : 25/30
C : 26/30
C : 27/30
A : 7/15
C : 28/30
C : 29/30
C : 30/30
Thread C ends
A : 8/15
B : 5/8
A : 9/15
B : 6/8
A : 10/15
A : 11/15
B : 7/8
A : 12/15
A : 13/15
B : 8/8
A : 14/15
A : 15/15
Thread B ends
Thread A ends
Now, the end
VM END (exit code = 0)
END
```

4 Discussion

4.1 The LLVJVM implementation

The document *ARM Cortex-Mx ARMCC - User's Manual* devotes a section to the LLJVM API. Anyone who wants to understand how the MicroJvm virtual machine works (with or) without an OS should read that document first.

The `LLMJVM_impl.h` file defines a set of functions that developers must implement with native code. These functions are used by the MicroJvm virtual machine and must be implemented according to the hardware platform and the native software capability. For instance, if an RTOS is available and the MicroJvm virtual machine runs as a task of this RTOS, RTOS functions can be used – this is how the LLMJVM implementation provided with the *TLT-0625-AN-MICROEJ_FromScratch* application note works, using Keil's RTX RTOS.

Here, no RTOS function can be used. Instead, we use our own functions and global variables to fulfill the needs of the `LLMJVM_impl.h` API. The main requirement is to be able to schedule the requests of the virtual machine. These are requests to schedule an alarm that will be triggered at a specified time. The requests are saved with the `LLMJVM_IMPL_scheduleRequest()` function. When the timeout is reached, the callback function `LLMJVM_schedule()`, which is provided by the platform, must be called.

When an RTOS is used, `LLMJVM_IMPL_scheduleRequest()` asks for the RTOS to trigger an event and call the callback function. Without an RTOS, we need to find a way to regularly check if next alarm time has been reached. This can be done in several ways: here we choose to use the `SYSTICK` timer. The `SYSTICK` timer is a feature of the Cortex-M family and regularly generates an interrupt. The associated routine is `SysTick_Handler()`. The request scheduling mechanism is as follows :

- the `SYSTICK` timer is configured to generate an interrupt every 10ms.
- `LLMJVM_IMPL_scheduleRequest()` memorizes the next request time in the `microjvm_nextWakeupTick` variable.
- `SysTick_Handler()` increments its own counter (`microjvm_tick`) and compares it to `microjvm_nextWakeupTick` to determine if it is time (or not) to call `LLMJVM_schedule()`.

This method is easy to use since functions are provided by the CMSIS library to configure the `SYSTICK` timer.

4.2 Explanation of the C main function

This section briefly describes the C main function. More details about how the virtual machine is started can be found in the SNI Specification. Two sections of that document are dedicated to this issue: part 2.4 *Starting the "Java world"* and part 4 *JAVA VIRTUAL MACHINE STARTUP*.

Since no RTOS is used for this Application Note, the virtual machine is initialized and started in the C main function itself :

```
int main(void)
{
    // Variables
    int32_t err;
    int32_t exitcode;

    // Code
    printf("START\n");
    SRAM_initialize();
    vm = SNI_createVM();

    if(vm == NULL)
    {
        printf("VM initialization error.\n");

    }
    else
    {
        printf("VM START\n");
        err = SNI_startVM(vm, 0, NULL);

        if(err &lt; 0)
        {
            //Error occurred
            if(err == LLMJVM_E_EVAL_LIMIT)

            {
                printf("Evaluation
                limits reached.\n");
            }
            else
            {
                printf("VM execution
                error (err = %d).\n", err);
            }
        }
        else
        {
            //VM execution ends normally

            exitcode = SNI_getExitCode(vm);
            printf("VM END (exit code = %d)\n", exitcode);
        }
    }

    SNI_destroyVM(vm);
    printf("END\n");
}
```

`SNI_startVM()` function starts and runs the virtual machine and the Java application linked to the project. It returns when this Java application ends. Then, we can check the exit code and destroy the virtual machine.

4.3 Standard output

When the Java application needs to write on the standard output (using a method like `System.out.println()`), the virtual machine implicitly calls low level functions from the C library, namely the `putchar()` function [3]. In fact, `putchar()` is not a such low level function and calls `fputc()` to do the job.

As stated in Keil's documentation [4], one must redefine the `fputc()` function to behave properly with available I/O devices. In our case, this function has been redefined to write on the UART output (connector CN16 on the board). The implementation can be found in the `putchar.c` file provided with this

---

[3] Note that native functions like `printf()` also use `putchar()`.

[4] See : http://www.keil.com/support/man/docs/armlib/armlib_bajfidff.htm [http://www.keil.com/support/man/docs/armlib/armlib_bajfidff.htm]

application note. This is why using `printf()` in C or `System.out.println()` in Java results in readable UART text.

5 Document History

Date	Revision	Description
December 17th, 2012	A	First release
May 13th, 2013	B	Minor changes in main document. Improve LLMJVM functions, mainly `LLMJVM_getCurrentTime()`.
November 12th, 2013	C	MicroEJ 2.0.0 compatibility
July 18th, 2014	D	MicroEJ 3.0.0 compatibility
October 1st, 2014	E	MicroEJ 3.1.0 compatibility

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