APPLICATION NOTE

Ref: **FR-191-AN-RB-004**
Date: **4th December 2010**
To: **General distribution**
From: **Richard Barry – Real Time Engineers Ltd.**
Subject: **A simple FreeRTOS demo for the Cortex-M3 using the Keil simulator**

INTRODUCTION

This application note is intended to assist in building, running and understanding the accompanying simple FreeRTOS demo (click the link for the source code project) that targets the Keil Microcontroller Development Kit (MDK) Cortex-M3 simulator.

The scope of the demo is deliberately kept small and at a basic level with the intention of ensuring it is understandable to those who have no previous RTOS experience. It also uses a minimal FreeRTOS configuration. Further and more detailed reading and reference material can be found in the following places:

1. The FreeRTOS web site (http://www.FreeRTOS.org)

   This provides a lot more information on the FreeRTOS project, including full licensing terms and full API documentation. The Quick Start Guide is a good place to visit after the home page (http://www.FreeRTOS.org/FreeRTOS-quick-start-guide.html). There is also a download link to the latest FreeRTOS release, which may have been updated since this package was put together.

2. The Cortex-M3 and LPC17xx versions of the FreeRTOS book (http://www.FreeRTOS.org/Documentation).

   This provides a tutorial style, step by step course on using real time kernels in microcontroller applications. It comes with a further 16 (soon to be 18) simple example projects that target various low cost Cortex-M3 development boards. It takes a deeper look at and describes the task states, the scheduling algorithm, system behavior, etc.

3. The accompanying PDF memo (document number and file name FR-201-MO-RB-003)

   This contains an introductory overview of FreeRTOS and the FreeRTOS project. It provides a summary description of the license terms and the support options available, with links to relevant pages within the FreeRTOS.org site.

BUILDING AND RUNNING THE PROVIDED DEMO

Obtaining the Build Tools

The free evaluation version of the Keil MDK can be downloaded from https://www.keil.com/demo/eval/arm.htm.

Opening the FreeRTOS Demo Project

The example simple project is provided in a single .zip file archive called “FreeRTOS-simple-demo-for-the-Keil-Cortex-M3-simulator.zip”.

1. Unzip the source files into a convenient location on your computer –

    a. Ensure the directory structure is maintained as the files are extracted.

    b. **Ensure the length of the destination folder name does not exceed the maximum file name length permitted on a Windows host.** If the absolute path to any of the header files exceeds this maximum length then the project will not build.

2. Install the Keil MDK and open the uVision4 IDE.

3. From within the uVision4 IDE, select ‘Open Project’ from the ‘Project’ menu.

4. Navigate to and open ‘Simple-FreeRTOS-Demo.uvproj’, which will be located in the root of the extracted source files.

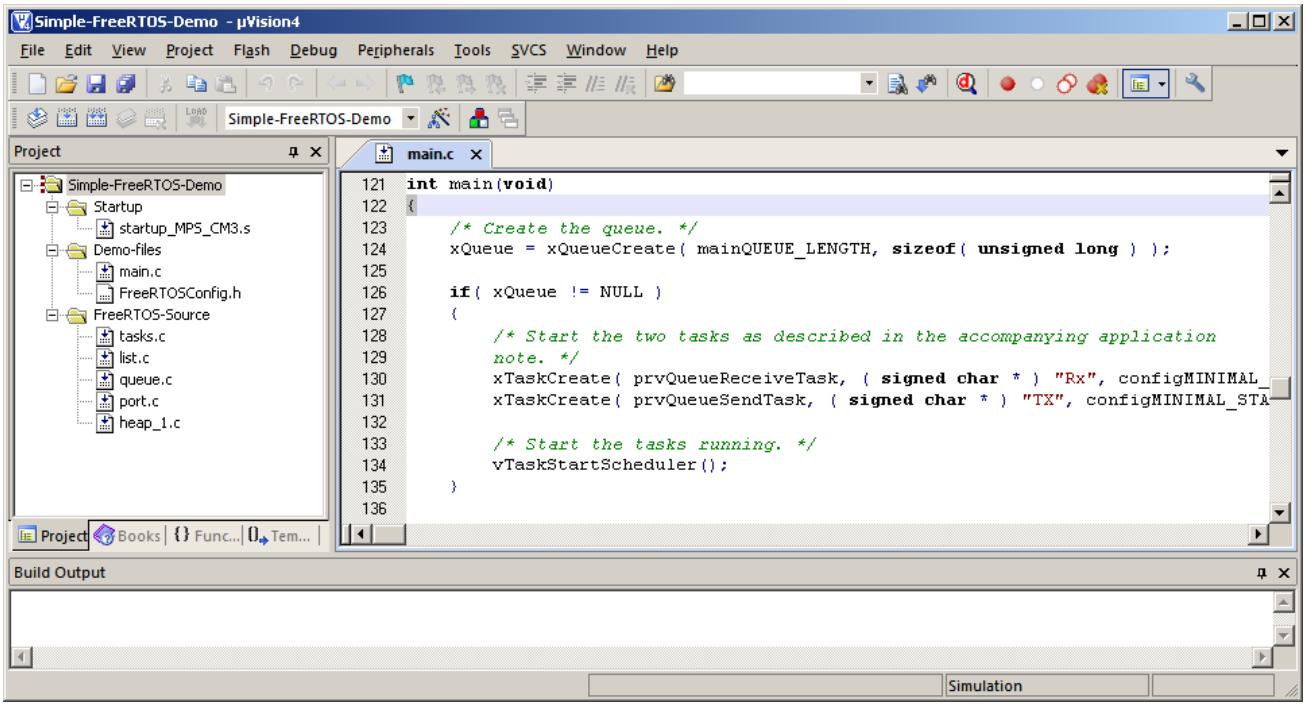

Figure 1. The simple FreeRTOS demo project once opened in the uVision4 IDE

Referring to Figure 1, it can be seen that the project is organized into the following sub-folders:

- Startup

This contains the C start up assembly file. It should be noted that this file is for a generic Cortex-M3 target and not any particular or specific Cortex-M3 device.

- Demo-files

This contains main.c, in which the entire simple demo is defined, and FreeRTOSConfig.h, in which the FreeRTOS build configuration is defined. The constants within the

FreeRTOSConfig.h header file are explained on the FreeRTOS.org web site. A link is provided in the file itself.

- FreeRTOS-source

This contains the FreeRTOS real time kernel source files that are needed for a Cortex-M3 application.

Building the Demo

There are several different ways in which the project can be built – the easiest of which is to simply press F7.

Starting a Simulator Debug Session

The project is already configured to run in the simulator and break on entry to the main() function. Again, there are several different ways to start a debug session – the easiest of which is to press Ctrl + F5. Once the debug session has started and the main() function has been reached, press F5 to start the simple demo executing. The simulator IDE is shown in Figure 2.

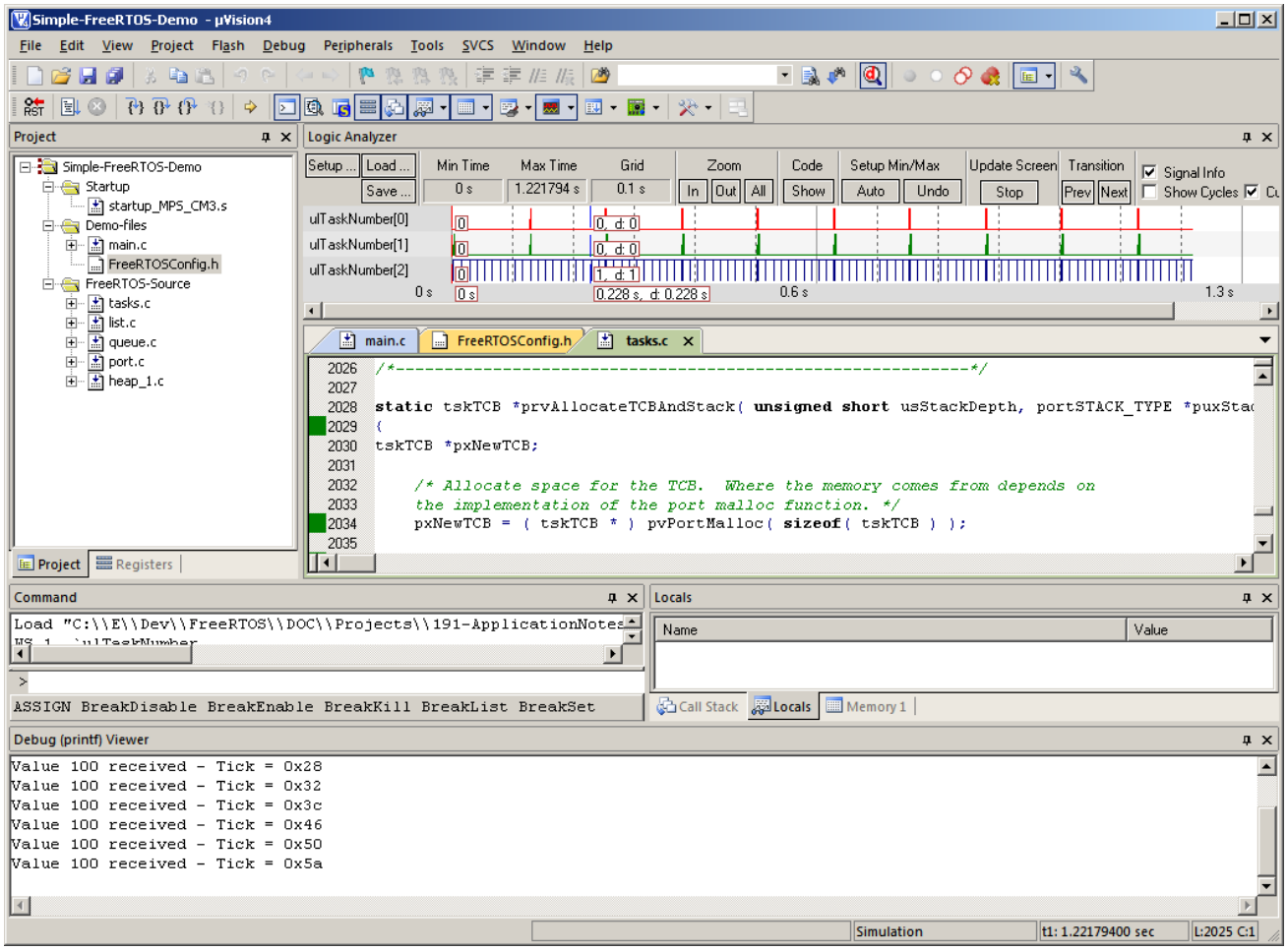

Figure 2 The uVision4 IDE when a debug session is in progress

Points to note in Figure 2:

- The top right window shows three signal traces in a simulated logic analyzer. These are described below. It will be necessary to zoom the view out to see the trace details.

- The bottom window displays the output of calls to printf().

THE SIMPLE DEMO PROJECT

The simple demo project demonstrates task and queue usages only. It is not intended to perform any useful functionality, or demonstrate how best to enhance application design by introducing multi-tasking. Details of other FreeRTOS features (API functions, tracing features, configuration options, diagnostic hook functions, memory management, etc.) can all be found on the FreeRTOS web site and/or in the FreeRTOS tutorial style book – links to both of which are provided in the introductory section of this application note.

All the functions described below are defined in the main.c source file of the project. Additional details can also be found in the comments within the source code itself.

The main() Function

main() creates one queue and two tasks before starting the scheduler. It does not execute past the call to start the scheduler as from that point on the tasks themselves will be executing.

The queue is used to pass a data value from one task (the queue send task) to the other task (the queue receive task). The queue receive task displays a string by calling printf() each time a data value that equals 100 is received on the queue.

The Queue Send Task

The queue send task is implemented by the prvQueueSendTask() function that is defined in main.c.

prvQueueSendTask() sits in a loop that causes it to continuously block for 10 milliseconds before sending the value 100 to the queue that was created within main(). It should be noted that the 10 milliseconds value is relative to the simulated execution time – not the observed time while the simulation is running.

The Queue Receive Task

The queue receive task is implemented by the prvQueueReceiveTask() function that is defined in main.c.

prvQueueReceiveTask() sits in a loop that causes it to continuously attempt to read data from the queue that was created within main(). When data is received it checks the value of the data, and if the value equals 100 the queue receive task outputs a string by calling printf().

The ‘block time’ parameter passed to the queue receive function specifies that the task calling queue receive should be held in the Blocked state indefinitely to wait for data to be available on the queue. The queue receive task will only leave the Blocked state when the queue send task writes to the queue. Because the queue send task writes to the queue every 10 milliseconds, the queue

receive task leaves the Blocked state every 10 milliseconds, which in turn causes printf() to be called every 10 milliseconds.

Observing the Application Behavior in the Logic Analyzer Window

FreeRTOS contains macros that can be defined to trace the execution sequence of an application. For this project a very simple tracing scheme has been implemented to allow the sequence in which tasks execute to be viewed in the debuggers logic analyzer window.

The logic analyzer window has been configured to show three signals – one representing each of the three tasks created in the simple demo application. When the signal is high the task is running, when the signal is low the task is not running. As only one Cortex-M3 core is being simulated only one task can be running at any one time. When no signals are high the kernel itself is running (a high zoom level in the logic analyzer window is necessary to observe this).

Referring to Figure 3 and Figure 4,

- The blue signal represents the Idle task. The idle task is created automatically by the kernel itself. It can be seen that the Idle task is executing for the majority of the time. It is periodically interrupted by the kernel tick interrupt.
- The green signal represents the queue send task. It runs every 10 milliseconds.
- The red signal represents the queue receive task. It runs each time the queue send task sends an item to the queue.

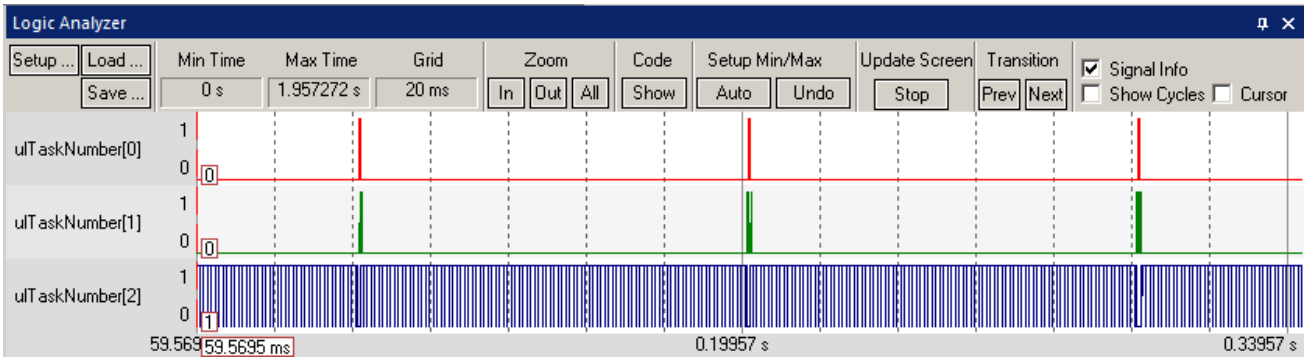

Figure 3 Signals viewed in the logic analyzer window with a medium zoom level

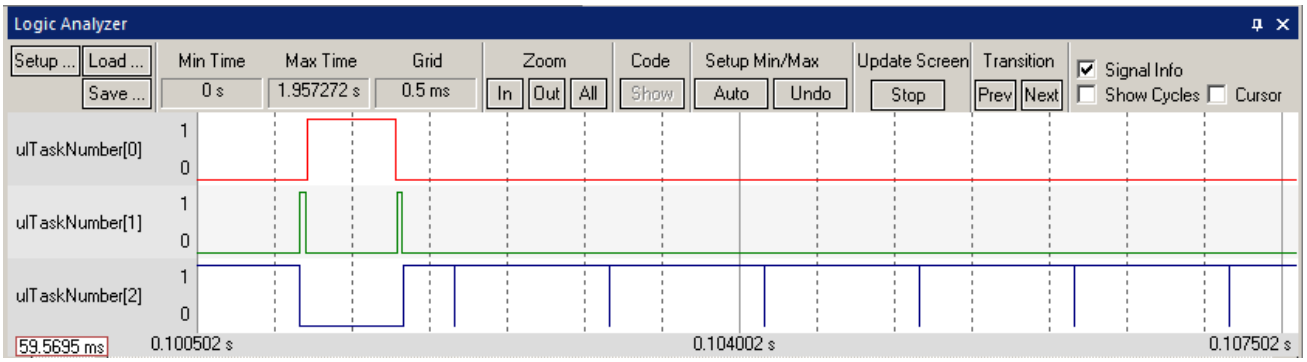

Figure 4 Signals viewed in the logic analyzer windows with a high zoom level to show context switching between tasks

Figure 4 shows a close up of the task execution sequencing during a queue send and queue receive operation. It demonstrates the task prioritization:

1. When the queue send task enters the running state (the green signal goes high) the first thing it does is write to the queue, which unblocks the queue receive task.

2. The priority of the queue receive task is higher than the queue send task, so the queue receive task pre-empts the queue send task (the red signal goes high and the green signal goes low).

3. The queue receive task spends some time outputting characters to the console before returning to block on the now empty queue (the red signal goes low).

4. The queue receive task blocking on the queue makes the queue send task the highest priority Ready state task once more, and the queue send task starts running (the green signal goes high again).

5. The queue send task exits the send API function before returning to the Blocked state itself. The idle task is now the only task that is not in the Blocked state so it starts running once more (the green signal goes low and the blue signal goes high).