Application Note 210

Running FreeRTOS on the Keil MCBSTM32 Board with the RVMDK Evaluation Tools

Document number: ARM DAI 0210A

Issued: June, 2008

ARM®

Application Note 210

Running FreeRTOS on the Keil MCBSTM32 Board with the RVMDK Evaluation Tools

Release information

Change history

Date	Issue	Change
June 2008	A	First release

Proprietary notice

Confidentiality status

This document is Open Access. This document has no restriction on distribution.

Feedback on this Application Note

If you have any comments on this Application Note, please send email to *errata@arm.com* giving:

- the document title
- the document number
- the page number(s) to which your comments refer
- an explanation of your comments.

General suggestions for additions and improvements are also welcome.

ARM web address

http://www.arm.com

Table of Contents

1 Introduction

This application note describes how to modify an existing port demo of the FreeRTOS operating system that targets the Luminary Micro LM3S102 evaluation board. It uses Keil's (an ARM company) *RealView Microcontroller Development Kit* (RVMDK) evaluation tools. The modifications will be made so that the operating system can run on the Keil MCBSTM32 board using the RVMDK evaluation software. The MCBSTM32 is Keil's first board based on the Cortex-M3 processor core.

FreeRTOS was the first real-time kernel to be available for production Cortex-M3 based microcontrollers.

Note: Although this document does not describe the complete details and functionality of the FreeRTOS kernel, they are not required to complete the modifications. Information on the operating system should be obtained from www.FreeRTOS.org.

In addition to being targeted for the evaluation version of RVMDK, the original port is also targeted at ARM's *RealView Development Suite* (RVDS) because both environments use the same compilation tools, (the compiler, linker, and assembler).. The main difference between the two sets of tools is the debugger and code editor. For the modified demo in this note, RVMDK will be the tool set used.

The modifications made to the OS port demonstration in this application note can also be used as general guidelines on how to modify other OS port demonstrations so that they can run on a different development platform, with the same or similar processor.

This application note contains the following sections:

- *Obtaining the Necessary Materials* describes the required materials and where to get them. This includes the Keil MCBSTM32 evaluation board (which includes an evaluation version of RVMDK, the ULINK-ME JTAG interface, and other necessary cables), serial cable(s) for testing purposes, and the FreeRTOS source and LM3S102 demonstration code.
- *Connecting and Configuring the Hardware and Software* explains how to connect and configure the necessary materials.
- *Modifying the LMS3102 FreeRTOS Port Demo to Run on the MCBSTM32* describes modifications to the existing LM3S102 FreeRTOS port demo that enables it to run on the MCBSTM32.
- *New Task/Co-routine Creation and Exercises* provides some ideas for exercises that can be added on to the modified port demonstration.

2 Obtaining the Necessary Materials

This section describes the necessary materials for modifying and running the FreeRTOS demo on the Keil MCBSTM32 evaluation board.

2.1 PC and Keil MCBSTM32 Evaluation Kit

The Keil MCBSTM32 is based on the STMicroelectronics Cortex-M3 family of ARM devices, and enables you to create and test working programs for this advanced architecture. Specifically, the board includes the STM32F103RB microcontroller, one serial interface, one analog input (via potentiometer), an SD Card interface, a CAN interface, an LCD, a USB interface, and eight LEDs.

RVMDK supports out-of-box ARM7, ARM9, and Cortex-M3 technology-based microcontrollers from many different vendors, and includes a code editor, compilation/link tools, a debugger, and other utilities.

The MCBSTM32 evaluation kit includes the MCBSTM32 evaluation board, a ULINK-ME USB-JTAG interface, and an evaluation version of RVMDK. The kit can be ordered directly from Keil at academic pricing, or through a local Keil distributor.

Note: The evaluation version of RVMDK will enable the building and debugging of the FreeRTOS port demonstrations referenced in this application note, and will be used throughout. See the section that follows for more information.

2.2 FreeRTOS Source Code and LM3S102 Keil/RVDS Demo Application

FreeRTOS is a royalty-free, open-source, real-time operating system kernel. There are many ports and demos available for various processor architectures and development tools from www.FreeRTOS.org.

Depending on the architecture and chip being used, it might be easier to modify an existing port demonstration based on the processor/board itself rather than the software tools. This depends on what ports are already available and if the port layer itself must be changed, as opposed to just changing a demo application. For example, although STM32 ports/demos already exist from www.FreeRTOS.org, they are not targeted at RVMDK, These existing port/demo project files were built specifically for other toolsets, and might have differing assembler/compiler syntax. All of the existing port/demos can be found in the *Port/Demo list* at www.FreeRTOS.org.

There is an STM32 port for the IAR tools, but the FreeRTOS port layer for RVMDK is required (and happens to be available in another port), However, the demonstration files can be compiled and assembled with RVMDK as is, although a new Keil project file must be created with all of these files. Fortunately, the Cortex-M3 port layer of FreeRTOS is the same over different chip implementations such as an ST part vs. a Luminary part. For these reasons, you might think it would be easiest to create a new RVMDK project with the RVMDK FreeRTOS port layer and the demo files from the STM32 IAR port demo. That's probably correct, assuming that you are using a *full* version (not evaluation version) of RVMDK. The problem is the way memory is allocated by the STM32 IAR FreeRTOS port. This can cause the code size to go over the 16K limitation of the evaluation version of RVMDK. For more information, see *User Documentation → Configuration → Memory Management* from the *FreeRTOS.org Menu* at www.FreeRTOS.org.

However, there are other Cortex-M3-based ports and demos targeted at RVMDK that can be used as a work around. These can be found in the *Port/Demo list* under the *FreeRTOS.org Menu*. The *LM3S102 Keil/RVDS* port's kernel allocates heap memory a bit differently. Instead of declaring a fixed-sized array of memory, a pointer is set at the end of RAM and accesses are made from offsets of the pointer as if it was an array. This is in fact more consistent with traditional heap allocation, and the kernel can allocate RAM up to the point where the heap clashes with the compiler allocated memory. With this method, the heap does not show up as a large block of RAM and is not included in the actual code size. Because of this, the base port used in this note will be the *LM3S102 Keil/RVDS* port.

Instructions on how to download and install the FreeRTOS source code and demo applications can be found on the website as well.

Note: All of the source and demos are downloaded together in one package, and it is not necessary to download only specific files.

2.3 RS232 DB9 Loopback Connector

Both the existing and modified demos include an interrupt-driven USART test where a co-routine transmits characters that are then received by a task, so a loopback serial connector will be needed for testing purposes. A loopback connector can be made from a standard male-to-female serial cable by simply shorting pins 2 and 3 on the female side with a paperclip. The male connector is for the MCBSTM32 USART port. More information about this and how to test a homemade loopback connector can be found at *http://zone.ni.com/devzone/cda/tut/p/id/3450*.

If the USART transmission portion must be verified, for example, if you want to actually see the characters being transmitted on the host PC, and the host PC does not have a serial port, a converter will be needed. Parallel-to-serial and USB-to-serial converters can be built or purchased

3 Connecting and Configuring the Hardware and Software

This section describes how to connect the MCBSTM32 and JTAG interface to a host PC running RVMDK. It also describes how to open the FreeRTOS base demo application for editing in RVMDK.

3.1 Setting up the Hardware

Use the following steps to setup your hardware:

1. Connect the ULINK-ME directly to the 20- pin JTAG connector on the MCBSTM32. Power the ULINK-ME by connecting the appropriate end of the included USB cable into it, and the other end into an open USB port on the host PC.

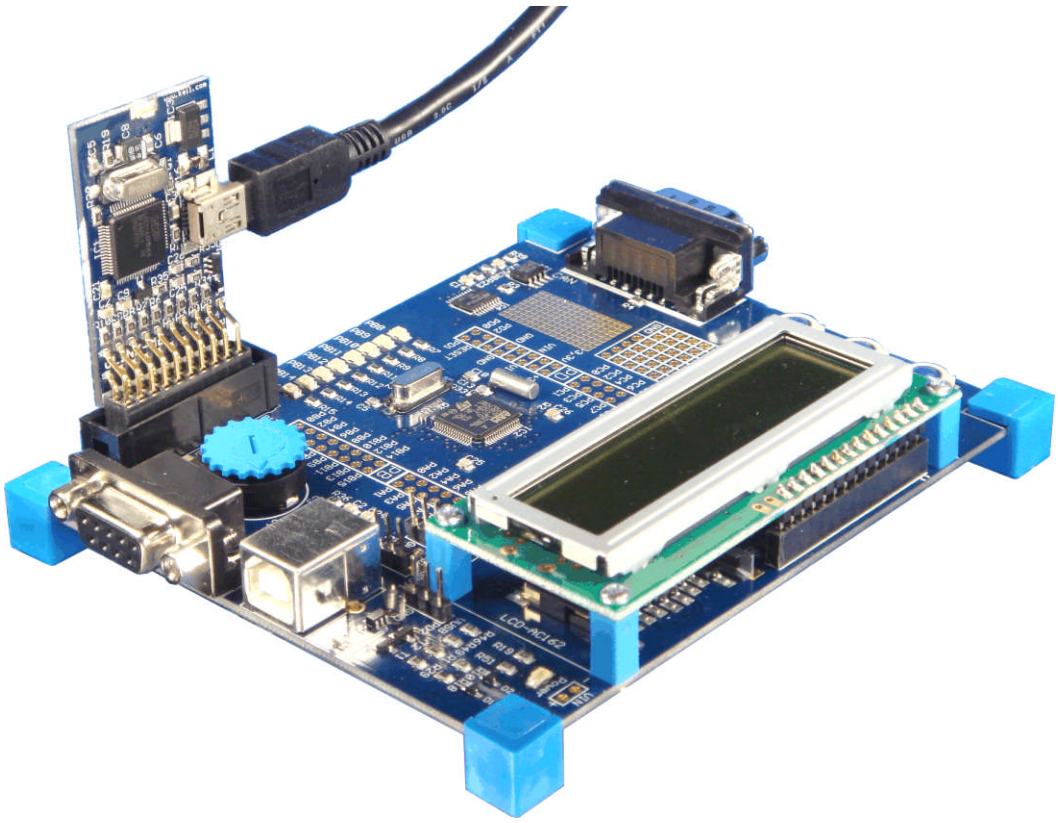

Figure 1. Hardware connections between ULINK-ME and MCBSTM32

If you are using a ULINK2 interface, it powers and connects the same way. If you are using a different JTAG interface, see the appropriate documentation for information on how to connect and power it.

2. Do the same with the included USB cable to power the MCBSTM32. When you power the board, the code that is already in flash memory (perhaps the included demo application) should run.

3. Fit the male side of one of the above described RS232 DB9 cables into the serial port of the MCBSTM32 labelled *COM*.

3.2 Starting RVMDK and Opening the Demo Project

Use the following steps to start RVMDK and open the project:

1. If RVMDK is not already installed on the host PC, insert the RVDMK evaluation CD into the CD-ROM drive of the PC and follow the on-screen instructions to complete the installation. If you do not have the CD, you can download the tools from *www.keil.com*.

2. The editor portion of RVMDK is called the Keil uVision3 (this is also the same name as the debugger). Launch uVision3 from its installed location or shortcut on the PC. RVMDK will startup initially in the uVision3 IDE. This is the editor and project management portion of the tools.

3. To open the LM3S102 demo uVision3 project file, click **Project → Open Project** from the uVision3 IDE menu options, and open the project file located at **\…\FreeRTOS\Demo\CORTEX_LM3S102_KEIL\FreeRTOS.uV2**. At this point, it is possible to compile, debug, and run this port and demo via the simulator. To do this, click on the **Options for Target** icon  , click on the **Debug** tab, and ensure that **Use Simulator** is selected.

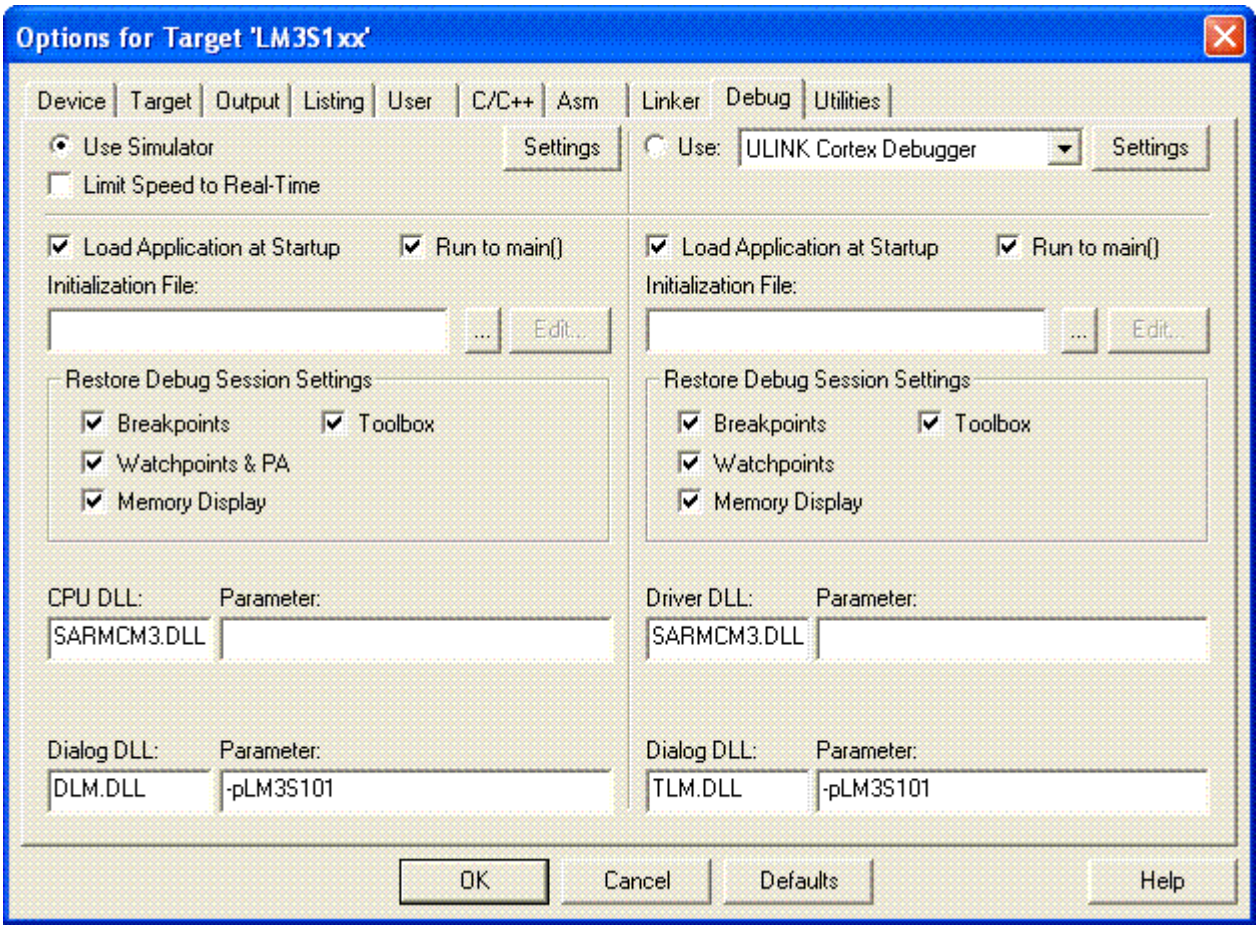

Figure 2. Options for Target Debug dialogue with "Use Simulator" selected

For information on how to compile, debug, and run applications with RVMDK, see the *RealView Compilation Tools for uVision* documentation, and the *uVision User Guide*.

For details about FreeRTOS source code and specific demo applications, see www.FreeRTOS.org.

4 Modifying the LM3S102 FreeRTOS Port Demo to Run on the MCBSTM32

This section describes the modifications required to run the existing LM3S102 port demo on the MCBSTM32. Although these modifications are specific to these two platforms, this section can also be used as general guidelines on how to modify other existing OS ports to run on different hardware platforms based on the same core architecture.

The LM3S102 demo application includes an interrupt-driven UART test where a co-routine transmits characters that are then received by a task via a loopback serial connector. LEDs on-board are used to indicate status, and some are controlled by a co-routine that does nothing but flash some LEDs. The LCD display is also used to display a message. The demo for the MCBSTM32 will have the same functionality, but of course, this can be extended. For more information, see the *LM3S102 Keil/RVDS* port demo application explanation at www.FreeRTOS.org.

4.1 Changing Target Processor Information

The first steps of the modification process include renaming some of the project folders, in addition to changing some of the target and project settings within uVision3.

4.1.1 Renaming the Project Folder Name

To avoid confusion, either change the name of the **\…\FreeRTOS\Demo\CORTEX_LM3S102_KEIL** folder, or create a new folder at that level and copy everything into it. For this application note, the **CORTEX_LMS3102_KEIL** folder was simply renamed to **CORTEX_STM32F103RB_KEIL** because the LMS3102 port demo was no longer needed.

Note: The project must be closed in order to change the directory name.

4.1.2 Changing the Target Processor

The next step is to change the target processor information because it is currently set to **LMS3S1xx**. Re-open the project from the new folder, and click **Components, Environment, and Books** button  . Under **Project Targets**, highlight **LM3S1xx** and delete it by clicking the **delete** icon  . Click the **insert** icon  and input the new target name **STM32F103RB**. Click **OK**. Now the **STM32F103RB** target should be the only one in the list of targets.

Up to this point, nothing has actually changed other than the *names* of the folder and target. To actually change the target processor in the uVision3 IDE, click on the **Options for Target** icon  , click the **Device** tab, and change the selected device from the **Luminary Micro LM3S101** to the **STMicroelectronics STM32F103RB**. After the choice is highlighted, the specs and peripherals for this microcontroller are listed.

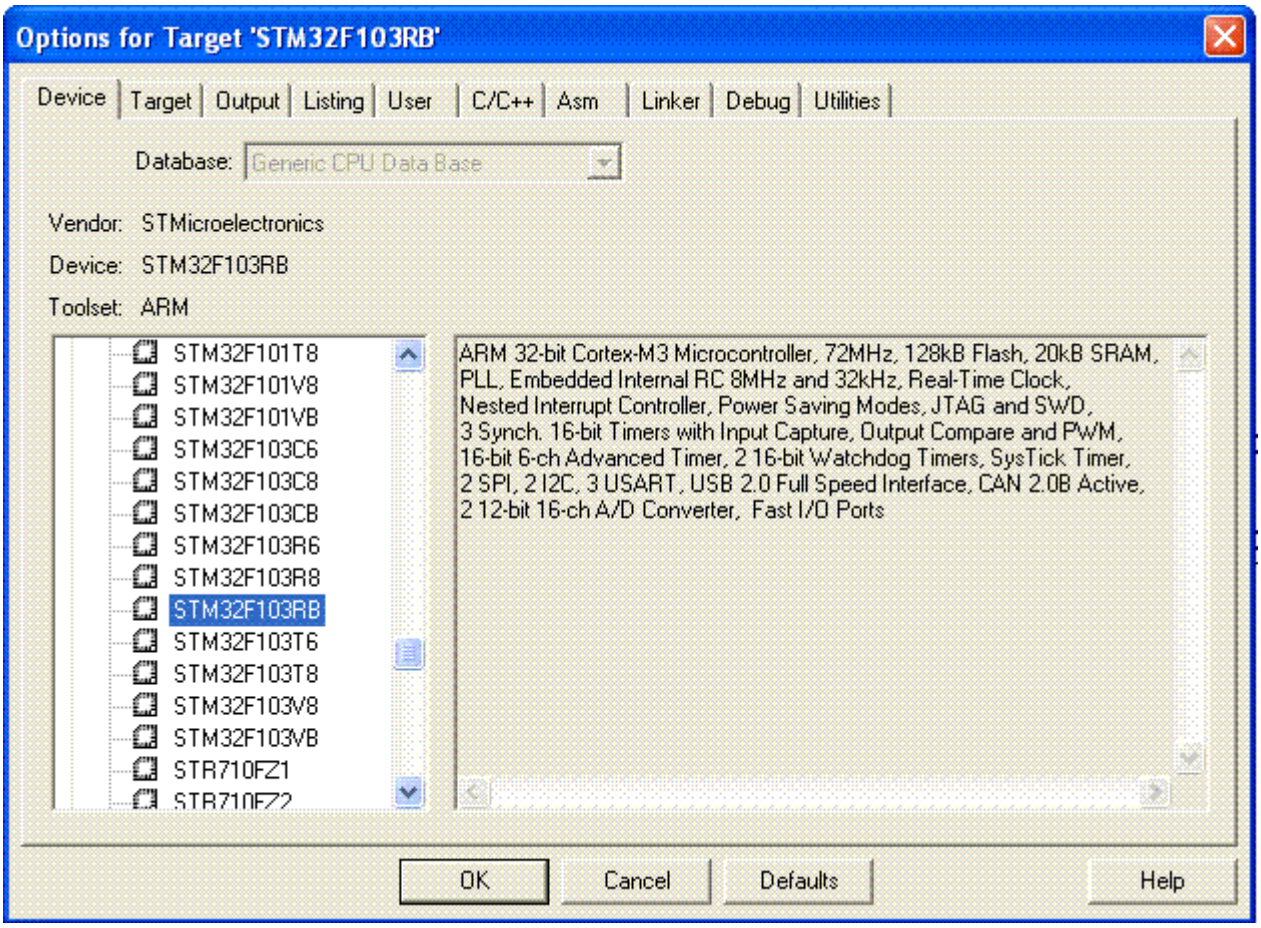

Figure 3. STM32F103RB Options for Target Device selection

Click **OK** to select the device and exit the **Options for Target.**

4.1.3 Modifying the Compiler Include Paths

The next step is to modify a few of the current compiler include paths to have the relevant library functions. To do this, click on the **C/C++** tab under **Options for Target**, then click on the **Include Paths** button  , and change the **\…\Keil\ARM\RV31\LIB\Luminary** path to **\…\Keil\ARM\RV31\LIB\ST**, and change the **\…\CORTEX_LM3S102_KEIL** path to **\…\CORTEX_STM32F103RB_KEIL**.

4.1.4 Changing the Preprocessor Symbol

The current LM3S102 preprocessor symbol must be changed. To do this, click on the **C/C++** tab under **Options for Target**, and change the name of the defined preprocessor symbol **RVDS_ARMCM3_LM3S102** to **RVDS_ARMCM3_STM32F103RB**. The preprocessor symbols are defined in a file that is part of the source directory called **portable.h.** This determines the relevant macro file that will be used for a particular port. The macro file used for all Cortex-M3 ports is the same, so simply change the line that reads:

```
#ifdef RVDS_ARMCM3_LM3S102
```

to:

```
#ifdef RVDS_ARMCM3_STM32F103RB
```

in **portable.h**. This step is mostly to keep naming consistency.

Note: RVDS is a completely different set of tools from RVMDK; however, they use the exact same compilation tools. Therefore, many similarities exist in files used by both RVDS and RVMDK for FreeRTOS ports and demos.

4.2 Adding and Removing Existing Target-specific Files

The next step of the modification process is to add or modify the necessary library, initialization, and startup files for the new target processor, and remove any unnecessary files.

4.2.1 Library Files

At this point, a build of the project using the **build** button  should display two major errors in the code. There are two references to the LM3S102 library file **LM3Sxxx.h** in **main.c** and **pdc.c**. **LM3Sxxx.h** was part of the library compiler include path that we changed in the last section, so change the line `#include "LM3Sxxx.h"` to `#include "stm32f10x_lib.h"`.

To reflect the new library file that defines peripheral register addresses, register bit definitions, and peripheral function prototypes, we also must add a firmware library file. This file defines many useful subroutines for writing software on the STM32F103RB. Keil provides this file that is configured for RVMDK, and can be found in **\...\Keil\ARM\RV31\LIB\ST**. Copy and paste the file **STM32F10xR.LIB** into **…\FreeRTOS\Demo\CORTEX_STM32F103RB_KEIL\include**.

For simplicity, we are also adding an even higher level of abstraction by using MCBSTM32-specific library files for the USART and LCD interfaces. Keil provided these files, and you can find them in most of the MCBSTM32 project folders, such as **…\Keil\ARM\Boards\Keil\MCBSTM32\Blinky**. Copy and paste the files **serial.c**, **LCD_4bit.c**, and **LCD.h** from an MCBSTM32 project folder into **…\FreeRTOS\Demo\CORTEX_STM32F103RB_KEIL\include**. We will use routines from these files later on.

4.2.2 Initialization Files

The file **pdc.c** is a peripheral control file specific to the LM3S102, so remove that file from the project by right-clicking on the file from the **Project Workspace** tree in the uVision3 IDE and selecting **Remove File 'pdc.c'**. Because this file and **pdc.h** are no longer needed, delete these two files from **\…\FreeRTOS\Demo\CORTEX_STM32F103RB_KEIL\include**. New initialization source and header files for the MCBSTM32 are to be included in the project to initialize its own peripherals. Again, Keil provided these files, and you can find them in most of the MCBSTM32 project folders, such as **…\Keil\ARM\Boards\Keil\MCBSTM32\Blinky**. Copy and paste the files **STM32_Init.c**, **STM32_Init.h**, and **STM32_Reg** from an MCBSTM32 project folder into **…\FreeRTOS\Demo\CORTEX_STM32F103RB_KEIL\include\,** and replace the line `#include "pdc.h"` with `#include "STM32_Init.c"` in **main.c**. Also, replace `#include "pdc.h"` with `#include "STM32_Init.h"` in **partest.c**.

4.2.3 Startup Files and Exception Vector Table

The last major file swap involves the startup files that are responsible for some specific chip and peripheral initialization, stack and heap setup, core exception modes and registers setup, in addition to initializing the exception vector table. Using the same steps as above, remove, delete, and replace **\…\FreeRTOS\Demo\CORTEX_STM32F103RB_KEIL\init\Startup.s** with

\…\Keil\ARM\Startup\ST\STM32F10x.s in **\…\FreeRTOS\Demo\CORTEX_STM32F103RB_KEIL\init**. Because the startup code is the first code that the processor executes, it calls the `main()` function, and is never referenced by other files. **STM32F10x.s** must be manually added to the **Demo** folder in the **Project Workspace** by right-clicking the folder and selecting **Add Files to Group 'Demo'**.

The current vector table in **STM32F10x.s** only implements dummy handlers. These files are for the programmer to modify (depending on the system). The FreeRTOS source and demo application make use of four different exception types and provide the exception handlers. These handlers are `vPortSVCHandler`, `xPortPendSVHandler`, `xPortSysTickHandler`, and `vUART_ISR`. The dummy handler calls in **STM32F10x.s** must be replaced with calls to these FreeRTOS handlers.

```
041 ; Vector Table Mapped to Address 0 at Reset
042
043                 AREA    RESET, DATA, READONLY
044                 EXPORT  __Vectors
045
046                 IMPORT  xPortPendSVHandler
047                 IMPORT  xPortSysTickHandler
048                 IMPORT  vUART_ISR
049                 IMPORT  vPortSVCHandler
050
051 __Vectors       DCD     __initial_sp              ; Top of Stack
052                 DCD     Reset_Handler             ; Reset Handler
053                 DCD     NMI_Handler               ; NMI Handler
054                 DCD     HardFault_Handler         ; Hard Fault Handler
055                 DCD     MemManage_Handler         ; MPU Fault Handler
056                 DCD     BusFault_Handler          ; Bus Fault Handler
057                 DCD     UsageFault_Handler        ; Usage Fault Handler
058                 DCD     0                         ; Reserved
059                 DCD     0                         ; Reserved
060                 DCD     0                         ; Reserved
061                 DCD     0                         ; Reserved
062                 DCD     vPortSVCHandler           ; SVCall Handler
063                 DCD     DebugMon_Handler          ; Debug Monitor Handler
064                 DCD     0                         ; Reserved
065                 DCD     xPortPendSVHandler        ; PendSV Handler
066                 DCD     xPortSysTickHandler       ; SysTick Handler
```

Figure 4. STM32F10x.s vector table

The dummy handlers below the vector can now be removed or commented out.

Again, the code for these handlers is written in other parts of the application. Most notably, the FreeRTOS scheduler is started in the `xPortPendSVHandler` handler, and all context switches are performed in this handler as well. Tasks that are blocked (waiting) are awakened by the tick interrupt. On the Cortex-M3, the tick interrupt is generated by the core SysTick timer (which is actually part of the Cortex-M3 core).

You can find more information on these particular handlers at *www.FreeRTOS.org*.

4.3 Modifying the Source Files

Now that most of the "house-cleaning" steps are out of the way, you can try actual code modifications and build tests.

4.3.1 Hardware Setup/Initialization

Rebuilding the project from this point using the **rebuild all** button  will turn up a multitude of errors, most of them (if not all) in **main.c**. These errors occur because of the LM3S102 functions and identifiers that were defined in the library and initialization files that we removed in the last section.

The function `prvSetupHardware()` calls functions that setup specific peripherals of the LM3S102, and those functions can all be removed and replaced with a single call to the `stm32_Init()` function found in **STM32_init.c**. The function `Stm32_Init()` initializes some of the STM32F103RB peripherals, including the clocks, timers, GPIOs, and USART interfaces.

Because we removed the only call to `vParTestInitialise`(), the definition of this function can be completely removed as well (from **ParTest.c**).

The function `vSerialInit()` and its prototype in **main.c** can also be removed completely because it is specific to the LM3S102 and is no longer needed.

More initialization is still needed, including initializing the interrupt controller of the STM32F103RB. Fortunately, this needed code has already been written, and can be found in the existing STM32F103 FreeRTOS port demo for the IAR tools found in the file **serial.c** in **\..\FreeRTOS\Demo\CORTEX_STM32F103_IAR\serial** (keep this file handy because more will be needed from it later). We will need most of the code in the function `xSerialPortInitMinimal()`, but some of the unsupported features of it must be stripped away and changed.

The new function that needs to be added to **main.c** `vSerialPortInitMinimal()` is shown here:

```
248 void vSerialPortInitMinimal( unsigned portLONG ulWantedBaud )
249 {
250 USART_InitTypeDef USART_InitStructure;
251 NVIC_InitTypeDef NVIC_InitStructure;
252 GPIO_InitTypeDef GPIO_InitStructure;
253
254     /* Enable USART1 clock */
255     RCC_APB2PeriphClockCmd( RCC_APB2Periph_USART1 | RCC_APB2Periph_GPIOA, ENABLE );
256
257     /* Configure USART1 Rx (PA10) as input floating */
258     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_10;
259     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
260     GPIO_Init( GPIOA, &GPIO_InitStructure );
261
262     /* Configure USART1 Tx (PA9) as alternate function push-pull */
263     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9;
264     GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
265     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
266     GPIO_Init( GPIOA, &GPIO_InitStructure );
267
268     USART_InitStructure.USART_BaudRate = ulWantedBaud;
269     USART_InitStructure.USART_WordLength = USART_WordLength_8b;
270     USART_InitStructure.USART_StopBits = USART_StopBits_1;
271     USART_InitStructure.USART_Parity = USART_Parity_No ;
272     USART_InitStructure.USART_HardwareFlowControl = USART_HardwareFlowControl_None;
273     USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;
274     USART_InitStructure.USART_Clock = USART_Clock_Disable;
275     USART_InitStructure.USART_CPOL = USART_CPOL_Low;
276     USART_InitStructure.USART_CPHA = USART_CPHA_2Edge;
277     USART_InitStructure.USART_LastBit = USART_LastBit_Disable;
278
279     USART_Init( USART1, &USART_InitStructure );
280
281     USART_ITConfig( USART1, USART_IT_RXNE, ENABLE );
282
283     NVIC_InitStructure.NVIC_IRQChannel = USART1_IRQChannel;
284     NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
285     NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
286     NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
287     NVIC_Init( &NVIC_InitStructure );
288
289     USART_Cmd( USART1, ENABLE );
290 }
```

Figure 5. More needed initialization via `vSerialPortInitMinimal()`

Now simply call this function from `prvSetupHardware()` with the needed baud rate as shown here:

```
292 static void prvSetupHardware( void )
293 {
294     /* Initialise the hardware used to talk to the LCD, LED's and UART. */
295     stm32_Init();
296     vSerialPortInitMinimal( mainBAUD_RATE );
297 }
```

Figure 6. Calling the initialization functions from `prvSetupHardware()`

Make sure that `mainBAUD_RATE` is set to `11520` at the top of **main.c**. This is the new baud rate for the STM32F103RB's USART.

4.3.2 Getting the Status LEDs to Work

Since FreeRTOS makes use of on-board LEDs to provide status and feedback information, it's important to get the LEDs working before running the OS itself. Generic instructions on how to do this for any platform can be found at *www.FreeRTOS.org* under *Demo App Introduction -> Modifying a demo*. A dummy main() is provided to test the LEDs, and should be used temporarily instead of the current main() function. However, instead of calling vParTestInitialise() to initialize the IO, make a call to prvSetupHardware(). Also, the system is currently setup to periodically cause an interrupt (the details are not important now), and FreeRTOS isn't actually going to be running yet to handle this, so delete the crude delay loop, as we will single-step through the code to test the LEDs. Also, comment out the entire original main() function for now.

The functions that blink the LEDs (vParTestToggleLED() and vParTestSetLED()) are defined in **ParTest.c**, and must be modified for the new hardware. These functions configure the ucOutputValue variable depending on what LED needs to be lit or toggled. A call to the LM3S102's peripheral control write function (PDCWrite()) is then made, passing that variable. First we must include the library files to make use of the new hardware by adding the line #include "stm32f10x_lib.h" to **ParTest.c**. Then, delete the calls to PDCWrite(), and replace them each with the line:

```
GPIOB->ODR = (GPIOB->ODR & 0xFFFF00FF) | (ucOutputValue << 8);
```

The function prvPDCWrite() itself (and its declaration) in **main.c** should also be removed.

The file **Stm32f10x_map.h** is provided by RVMDK and contains all of the STM32F103RB's peripheral registers' definitions and memory mappings. ST uses C structures for this particular chip to address these registers. The structure component GPIOB->ODR refers to GPIOB's port output data register. The 8 LEDs on the MCBSTM32 are connected to GPIOB pins [15:8], and can be lit/toggled by writing to these bits of GPIOB->ODR.

For more information about the STM32F103RB's peripherals and peripheral registers, see the *STM32F103xx Reference Manual*.

At this point, a build of the project will still turn up many errors regarding more references to LM3S102-specific hardware. Comment out each of these errors, since we only want to confirm that we have the LEDs working at this time. In the uVision3 IDE, double-click each error to bring the cursor to the error line, and comment out each one. Ignore the warning messages for now.

Note: Make sure to use the latest version of the Cortex-M3 FreeRTOS port layer, as it had been updated during the writing of this application note, and older versions will require different modifications than the ones in this note.

To confirm that the LEDs are working, all of the hardware must be connected as described in *Section 3* above, and uVision3 must be running in hardware mode (instead of simulation mode). To ensure uVision3 is in hardware mode, click on the **Options for Target** icon  , click on the **Debug** tab, and ensure that **Use: ULINK Cortex Debugger** is selected (not **Use Simulator**).

Once the build is error free and an executable image is created, click the **Download to Flash Memory** icon 

. The flash download attempt will fail! To this point, we have not changed the flash download and execution address to reflect the STM32F103RB's memory map. The flash memory of the STM32F103RB starts at address 0x08000000 and should be used as the **R/O Base** under the **Linker** tab under **Options for Target**. The **R/O Base** sets the address for constants and code containing the RO (Read-Only) output section. The **R/O Base** also sets the initial program entry address. Also, be sure to remove any **Misc controls** linker options that might be there (such as **–first Reset_Handler**) that override the **Linker Control String** specifically for the LM3S102. The **Linker** tab under **Options for Target** should look like this:

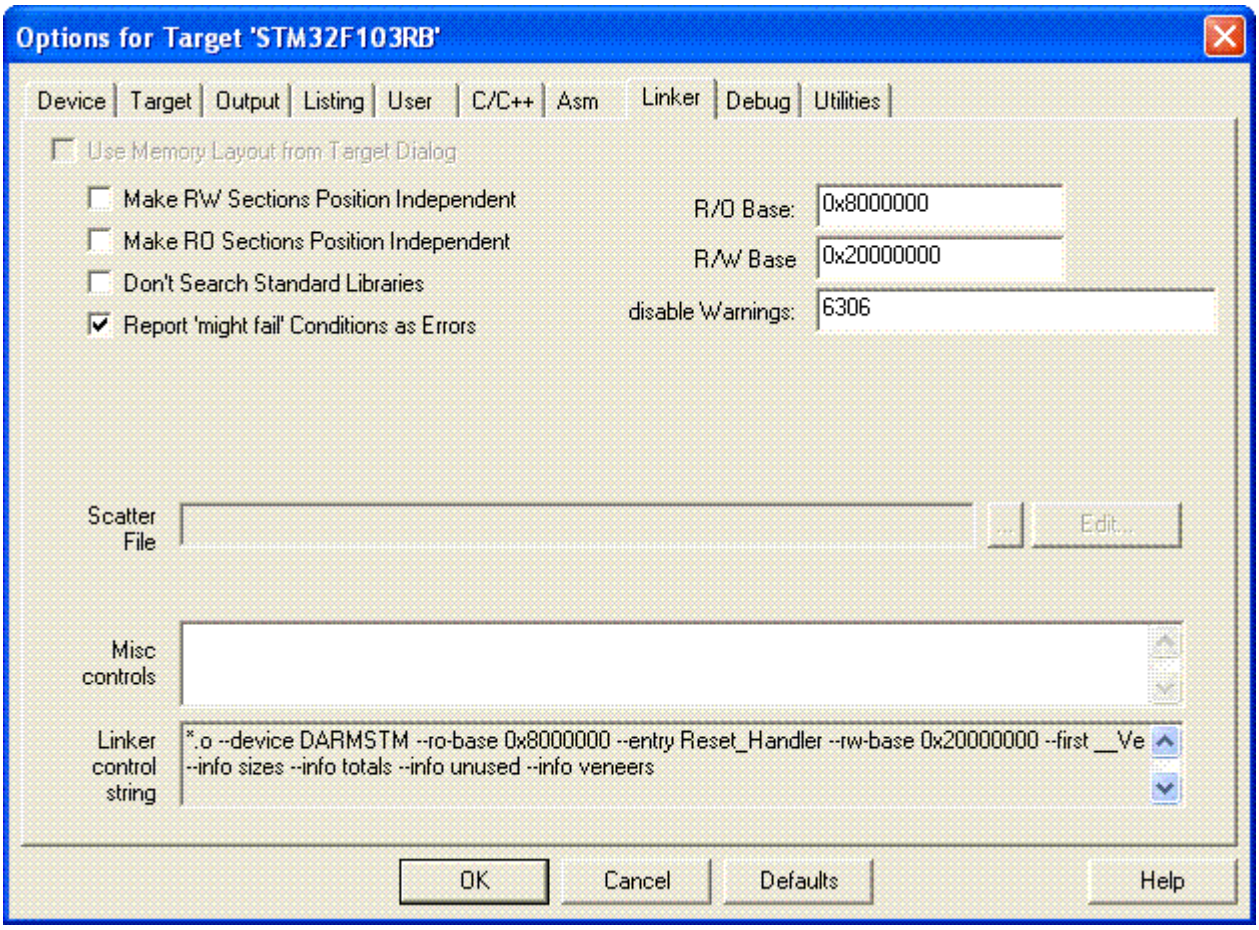

Figure 7. Options for Target Linker dialogue for STM32F103RB

For more information on ARM compilation tools output sections, see the *RealView Compilation Tools for uVision Essentials Guide* and the *RealView Compilation Tools for uVision Developer Guide*.

Another attempt at a flash download will turn up another problem. uVision3 uses specific, built-in flash download algorithms depending on the chip. Under the **Utilities** tab in the **Options for Target**, click on the **ULINK Cortex Debugger Settings**, ensuring that the **Flash Download** tab is selected. Click on the **LM3Sxxx 8kB Flash** programming algorithm, then click the **Remove** button. Click the **Add** button, scroll down and select the **STM32F10x 128kB Flash** algorithm, click **Add**, and finally click **OK** twice to apply the changes.

Now building the code and downloading the executable to flash should work correctly, and the uVision3 output window should verify this. Click the **Start/Stop Debug Session** icon  . The uVision3 debugger will come up with the Program Counter pointing to the first line in `main()`.

Single-step through the code using the **Step Over** button  . Each step over the calls to `vParTestToggleLED()` should toggle the corresponding LED on the MCBSTM32.

4.3.3 LED Co-routines

Now that we know the LEDs work, restore the original `main()` function and delete the dummy `main()` function. At this point we want to create the LED co-routines with:

```
vStartFlashCoRoutines(mainNUM_FLASH_CO_ROUTINES);
```

The co-routines use FreeRTOS to blink 5 LEDs at different rates. Make sure to comment out the UART transmission co-routine, the LCD task, as well as the UART receive task (it will take a bit more work to get these running).

Rebuild the project using the **rebuild all** button  , download the executable to flash, and press the **RESET** push-button on the MCBSTM32 to run the code out of flash. The first five LEDs (**PB8** – **PB12**) should all continuously blink at different rates. These LEDs are under the control of the flash co-routines, each flashing at a specific frequency, with **PB8** being the fastest and **PB12** being the slowest.

Information about FreeRTOS tasks and co-routines can be found at *www.FreeRTOS.org*.

4.3.4 Creating the Queues for the USART

Memory queues are used in FreeRTOS port demos to implement USART functionality.

For this demo modification, create two separate queues in memory for the Tx and Rx sides of the USART communication instead of using the single `xCommsQueue` that was created in the first line in `main()`. To do this, replace the line:

```
xCommsQueue = xQueueCreate(mainRX_QUEUE_LEN, sizeof( portCHAR));
```

with the lines:

```
xRxedChars = xQueueCreate(mainRX_QUEUE_LEN, (unsigned portBASE_TYPE)
sizeof(signed portCHAR));
```

and

```
xCharsForTx = xQueueCreate(mainRX_QUEUE_LEN + 1, (unsigned portBASE_TYPE)
sizeof(signed portCHAR));
```

The use of two queues instead of one implements a slightly different functionality by transmitting characters over one queue and interrupts reading/receiving characters from the other queue. The characters are then sent over the USART. The LMS3102's version that uses only one queue uses the queue to only receive characters, with transmitted characters sent directly from the USART Tx buffer by an interrupt.

These queues are created with `xQueueCreate` (which takes in the size of the queue in bytes and the data size for each element), and must be declared at the top of **main.c** with the lines:

```
static xQueueHandle xRxedChars;
static xQueueHandle xCharsForTx;
```

Also, be sure to remove the old declaration for `xCommsQueue` to avoid warnings about it being declared and not used.

There is one additional step to be done in order to get the two-queue model working. To create the extra queue, we must increase the heap size slightly, otherwise the scheduler won't start (because the idle task won't be created, but that is beyond the scope of this note). To do this, set the constant `configTOTAL_HEAP_SIZE` to `2468` instead of `1468` in **FreeRTOSCOnfig.h**. This is an application-specific hardware configuration file for FreeRTOS.

The parameter `configTOTAL_HEAP_SIZE` might need to be increased even more if new tasks and co-routines are created as per *Section 5* of this note.

4.3.5 USART Tx Co-routine

Now that the queues have been created, we can bring up the USART transmission co-routine that initiates the transmission of sequential characters. This requires editing some code that is specific to the LM3S102. The creation of the co-routine is called in `main()` with:

```
xCoRoutineCreate(vSerialTxCoRoutine, mainTX_CO_ROUTINE_PRIORITY,
mainTX_CO_ROUTINE_INDEX);
```

Some STM32F103RB USART1 settings must be changed in **STM32_Init.c**. The USART configuration definitions are around line 2150 of that file. Change the declaration `#define __USART1_DATABITS` from `0x0` to `0x8` and change the declaration `#define __USART1_STOPBITS` from `0x0` to `0x1`.

In general, the definitions in this file are left for the programmer to define depending on the specific hardware.

More modifications must be made. In **main.c**, all instances of `UART0_BASE` must be replaced with `USART1_BASE` (defined in **STM32F10x_map.h**) to reflect the memory-mapped address of the STM32F103RB's USART interface. Similarly, there are obsolete instances of `UART_INT_TX`, which is the transmit interrupt status bit. Replace all instances of this with the STM32F103RB's equivalent, `USART_IT_TXE`.

The routine `vSerialTxCoRoutine()` in **main.c** is the USART transmission co-routine, and needs a few modifications to work on the MCBSTM32. This part of the existing LMS103 code disables UART transmission interrupts before sending a sequential character, then re-enables the UART transmission interrupts:

```
428         UARTIntDisable( UART0_BASE, UART_INT_TX );
429         {
430             /* Send the first character. */
431             if( !( HWREG( UART0_BASE + UART_O_FR ) & UART_FR_TXFF ) )
432             {
433                 HWREG( UART0_BASE + UART_O_DR ) = cNextChar;
434             }
435
436             /* Move the variable to the char to Tx on so the ISR transmits
437             the next character in the string once this one has completed. */
438             cNextChar++;
439         }
440         UARTIntEnable(UART0_BASE, UART_INT_TX);
```

Figure 8. Part of the LMS3102's vSerialTxCoRoutine()

This part of the code should be replaced with the following code:

```
497         USART_ITConfig( USART1, USART_IT_TXE, DISABLE );
498         {
499             /* Send the first character. */
500             if (cNextChar == '(')
501                 cNextChar = '0';
502             ser_putchar(cNextChar);
503
504             /* Move the variable to the char to Tx on so the ISR transmits
505             the next character in the string once this one has completed. */
506             cNextChar++;
507         }
508         USART_ITConfig( USART1, USART_IT_TXE, ENABLE );
```

Figure 9. Part of the new STM32F103RB's vSerialTxCoRoutine()

The definition of `ser_putchar()` is found in **serial.c** and the definition of `USART_ITConfig()` is contained in the firmware library file **STM32F10xR.LIB**. Both of these files must be manually added to the **Project Workspace** in the uVision3 IDE.

We only want to transmit and receive ASCII characters '0' to 'z', so we use the `if` statement to restart `cNextChar`.

More information about the routines found in **STM32F10xR.LIB** can be found in the *ARM-based 32-bit MCU STM32F101xx and STM32F103xx Firmware Library* document from STMicroelectronics.

The LM3S102 port at the time of writing this application note is coded so that nothing but the '0' character is sent out the serial port. It's more interesting to see all of the sequential characters, so within `vSerialTxCoRoutine()`, move the line

```
cNextChar = mainFIRST_TX_CHAR;
```

from inside of the infinite `for(;;)` loop to outside of it right below the line

```
crSTART( xHandle );
```

To verify that the USART transmission under control of FreeRTOS is actually working, a 'dumb' terminal program must be used, such as Windows HyperTerminal. If it's not already, connect the serial cable from the MCBSTM32 to a working and verified serial port (or another port using a converter) on the host PC.

In Windows XP, start HyperTerminal by clicking **Start → All Programs → Accessories → Communications → HyperTerminal**. Enter a **New Connection Name** (perhaps "USART Tx"), click on the **Connect using** drop-down menu and select the correct COM port, and click **OK**. The **Port Settings** for the COM port are shown, and enter the settings that match those of the STM32F103RB's **STM32_Init.c** file,shown below.

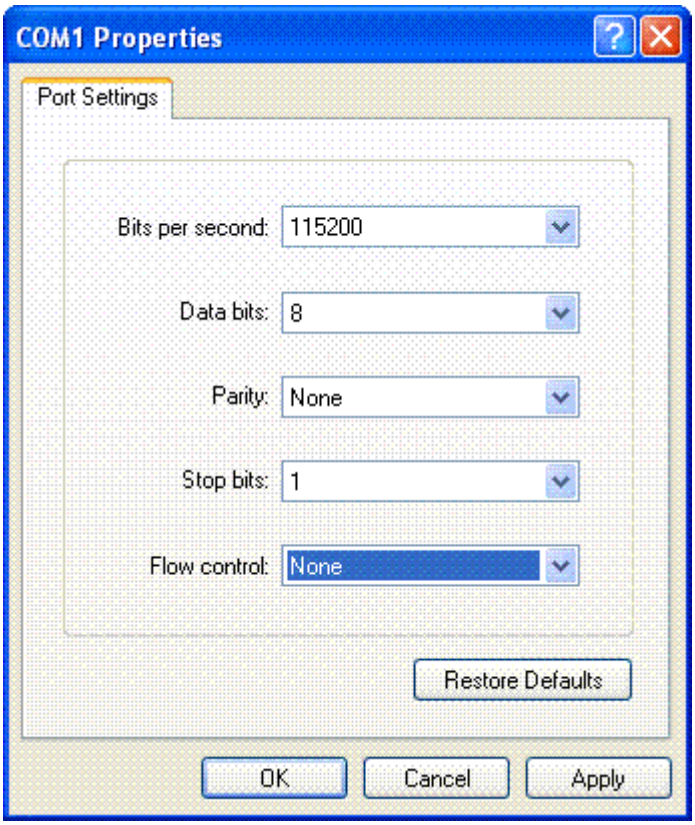

Figure 10. Serial COM port properties for PC in Windows HyperTerminal

Reprogram the flash of the MCBSTM32 with the updated executable and press the **RESET** push-button on the board. Sequential characters should show up on the terminal window starting at '0', as shown in Fig. 11.

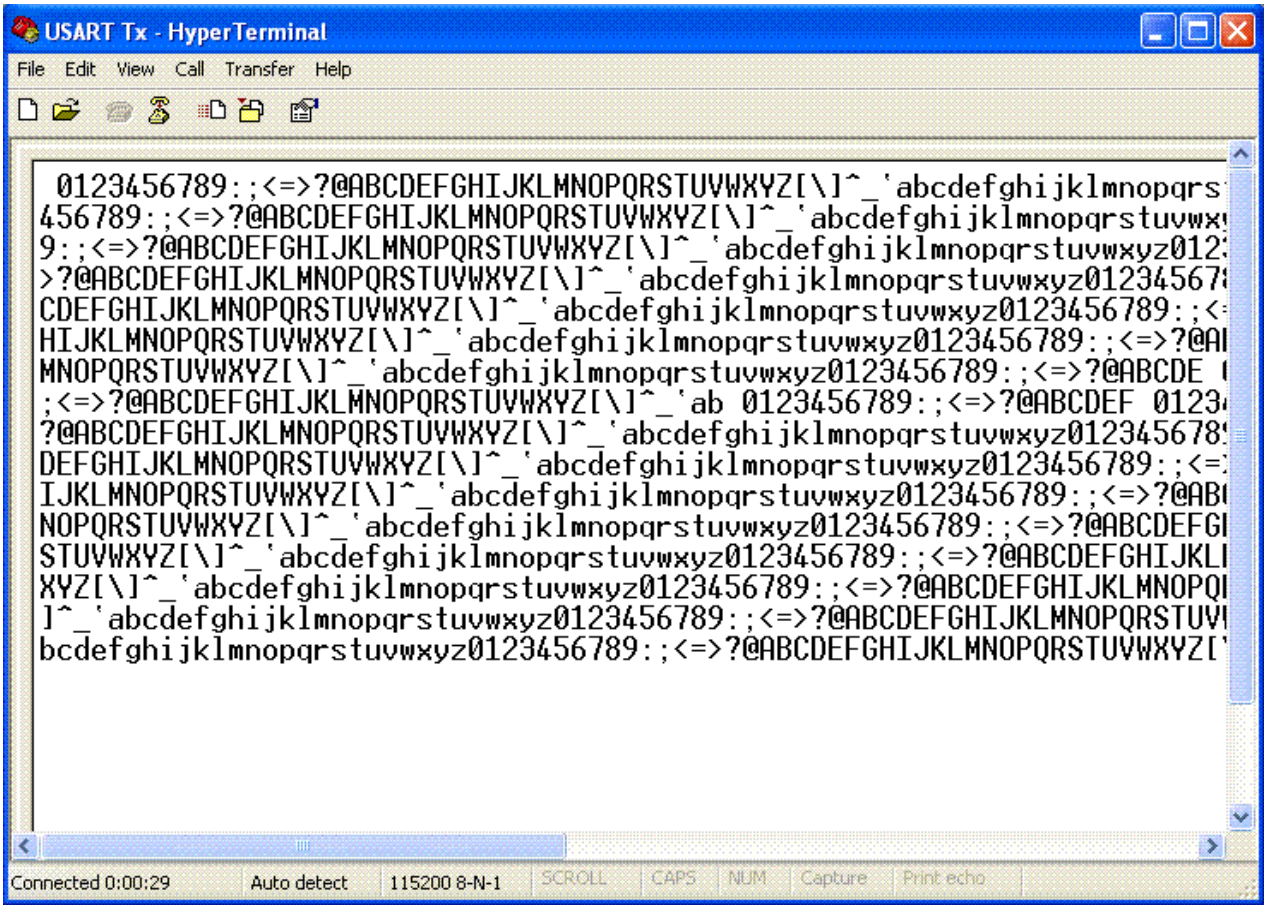

Figure 11. STM32F103RB's USART transmit results in HyperTerminal

The sixth LED on the MCBSTM32 (marked **PB13**) should also be flashing to indicate successful USART character transmissions.

4.3.6 LCD Task

Going in sequential order in `main()`, the next part of the demo to modify is the LCD task `vLCDTask`. This is much simpler than it looks due to the level of abstraction used in the LCD functions in **LCD_4bit.c**.

You can get as creative as you want with the LCD functionality, but for a simple test, just try using the following instead of the LM3S102's version of `vLCDTask`:

```
308 void vLCDTask( void * pvParameters )
309 {
310     /* Configure the LCD. */
311     vTaskDelay( mainCHAR_WRITE_DELAY );
312     lcd_init  ();
313
314     /* Clear display. */
315     vTaskDelay( mainCHAR_WRITE_DELAY );
316     lcd_clear ();
317
318     for( ;; )
319     {
320         /* Display the string on the LCD. */
321         lcd_print (" MCBSTM32 DEMO ");
322         set_cursor(0, 1);
323         lcd_print ("www.FreeRTOS.com");
324         vTaskDelay( mainSTRING_WRITE_DELAY * 3 );
325         lcd_clear ();
326         vTaskDelay( mainSTRING_WRITE_DELAY * 3 );
327     }
328 }
```

Figure 12. New vLCDTask( )

This code simply flashes the text given to `lcd_print()` so that it all fits on the MCBSTM32's LCD screen. Notice that we no longer need `prvWriteString()` for this simple test, so it and its declaration at the top of **main.c** can be removed.

Be sure to include **LCD_4bit.c** in uVision3's **Project Workspace** in order to make use of the new library functions if it's not there already, and declare the functions somewhere at the top of **main.c** (otherwise warnings will be generated at build time about these LCD library functions being declared implicitly. Also, make sure to uncomment the LCD task creation line in `main()`:

```
TaskCreate(vLCDTask, "LCD", configMINIMAL_STACK_SIZE, NULL,
mainLCD_TASK_PRIORITY, NULL);
```

Again, reprogram the flash of the MCBSTM32 with the new executable, and reset the board to confirm that the LCD is working.

4.3.7 USART Rx Task

The next part of the demo to be modified is the USART Rx task `vCommsRxTask()`. Only a very small modification needs to be done here, and that is to change the name of the queue that is used for this task. Just replace the instances of `xCommsQueue` in the calls to `xQueueReceive()` with `xRxedChars`. `xQueueReceive()` is a generic function which receives data from a specified queue.

For more details on the queues and queue functions used by FreeRTOS, see *www.FreeRTOS.org*.

4.3.8 USART Interrupt Service Routine

The final part of the demo to be changed is the USART Interrupt Service Routine `vUART_ISR()`. In *Section 4.2.3* of this note, this ISR call was added to the Cortex-M3's vector table.

Similar to other changes described in this note, the code for this routine can be found in the existing STM32F103 FreeRTOS port demo for the IAR tools, located in the file **serial.c** in **\..\FreeRTOS\Demo\CORTEX_STM32F103_IAR\serial**. This is the same file from which we pulled initialization code in *Section 4.3.1* of this note

The routine `vUARTInterruptHandler()` is the code we need. Use the body of this routine in `vUART_ISR()` in **main.c**. `vUART_ISR()` in **serial.c** should now look like this:

```
451 void vUART_ISR( void )
452 {
453 portBASE_TYPE xTaskWokenByTx = pdFALSE, xTaskWokenByPost = pdFALSE;
454 portCHAR cChar;
455
456     if( USART_GetITStatus( USART1, USART_IT_TXE ) == SET )
457     {
458         /* The interrupt was caused by the THR becoming empty.  Are there any
459         more characters to transmit? */
460         if( xQueueReceiveFromISR( xCharsForTx, &cChar, &xTaskWokenByTx ) == pdTRUE )
461         {
462             /* A character was retrieved from the queue so can be sent to the
463             THR now. */
464             USART_SendData( USART1, cChar );
465         }
466         else
467         {
468             USART_ITConfig( USART1, USART_IT_TXE, DISABLE );
469         }
470     }
471
472     if( USART_GetITStatus( USART1, USART_IT_RXNE ) == SET )
473     {
474         cChar = USART_ReceiveData( USART1 );
475         xTaskWokenByPost = xQueueSendFromISR( xRxedChars, &cChar, xTaskWokenByPost );
476     }
477
478     portEND_SWITCHING_ISR( xTaskWokenByPost || xTaskWokenByTx );
479 }
```

Figure 13. New vUART_ISR()

Again, many of the functions here are defined by the FreeRTOS port layer and the *ARM-based 32-bit MCU STM32F101xx and STM32F103xx Firmware Library* document from STMicroelectronics.

Now if you haven't done so already, uncomment the USART Rx task creation line in `main()`:

```
xTaskCreate( vCommsRxTask, "CMS", configMINIMAL_STACK_SIZE, NULL,
mainCOMMS_RX_TASK_PRIORITY, NULL );
```

4.3.9 Testing the Modified Demo

If all of the steps have been performed correctly to this point, a re-build of the project should turn up no errors or warnings.

Re-build the project using the **re-build all** button  , download the executable to flash, and press the **RESET** push-button on the MCBSTM32 to run the code out of flash. The first seven LEDs (**PB8** – **PB14**) should all continuously blink at different rates, with **PB8** blinking the fastest and **PB14** blinking the slowest.

Again, the first five LEDs are under the control of the flash co-routines, each flashing at a particular frequency, with **PB8** being the fastest and **PB12** being the slowest. **PB13** will flash each time a character is transmitted on the serial port. **PB14** will flash each time a character is received and validated on the serial port though the loopback connector.

PB15 is used to indicate that an error has been detected and should remain off, unless of course the loopback connector is removed. Try it and see!

The LCD will display a message, depending on what was done in *Section 4.3.6*.

5 New Task/Co-routine Creation and Exercises

Now that we have a working port demo for the MCBSTM32, you can add new tasks to be used by FreeRTOS. Be careful not to go over the 16K code limitation of the RVMDK evaluation tools.

General information about tasks and adding new ones can be found in the documentation at *www.FreeRTOS.org*.

5.1 More LED Task Functionality

Try creating a task that blinks a single LED by passing in the particular LED like so:

```
void LEDTask( void *uxLED )
{//code}
```

Use a crude delay mechanism to create a particular blink rate. When this single task is created and the scheduler is started, watch the blink rate. What happens to the blink rate if you create an identical task for a different LED and run both simultaneously? Why?

What happens when the priority of one of these tasks is changed, and why?

Instead of using a crude delay mechanism, call `vTaskDelay()`. What happens now, and why?

5.2 Analogue-To-Digital Conversion

For a slightly more difficult example, try using the tick hook functionality to periodically trigger an analogue-to-digital conversion. Create a new co-routine that receives the result from the STMF103RB ADC's `ADC_IRQHandler` and prints the result to the LCD.

For this, **crhook.c** must be added from **\...\FreeRTOS\Demo\Common\Minimal**.

6 Conclusion

With the steps outlined in this application note, you should be able to get FreeRTOS up and running on the MCBSTM32 with an extendable demo application, using the license-free RVMDK evaluation tools from ARM/Keil. This is useful in seeing FreeRTOS run in real hardware (using one of ARM's latest processor cores: the Cortex-M3) with a completely free set of tools.