Using FreeRTOS on the ARM/NXP mbed

Frank Mölendörp, Doulos, Feb 2011

Abstract

This article explains how to use the ARM/NXP mbed prototyping kit with FreeRTOS. We will describe how to combine the best features of both, device abstraction in mbed, and OS functionality in FreeRTOS and forge them into a powerful prototyping device.

What is mbed?

mbed is a highly abstracted microcontroller prototyping platform, originally developed by two ARM engineers and supported by lead partner NXP. The software abstraction allows the user to access all MCU resources through C++-classes. In most cases no additional configuration will be necessary. Peripherals are represented by C++ classes and their functionality is accessed via method calls.

mbed is based on an LPC1768 MCU by NXP and provides two SPI-, two I^2C-, and three serial ports, as well as a USB and an Ethernet-interface. There are also six analog-inputs, six PWM-outputs and a n analogue-output. 30 of the pins can be used as digital in- or output. It is also possible to use them as bidirectional pins.

In which way does mbed differ from evaluation kits? First, there is he online compiler. Using mbed does not require installation of any compiler toll chain or flash programmer. After creating an account for the mbed project page, developers will use an online DIE which exists only in a web browser. Pressing the compiler button uploads the source code into the cloud and returns an executable image file. When plugged into USB port, mbed will itself to the computer as mass storage device to which the image must be stored. After pressing the only button on the mbed board the application will start.

This work does not explain real time operating system. General information about RTOS and how they work can be found in the internet. The main reason of this work is to explain how to get Real Time Engineers Ltd.'s FreeRTOS running on mbed and take advantage of mbed-'s device abstraction.

What is mbed software library?

The library is an object oriented hardware abstraction layer implemented in C++ which provides mechanisms to access MCU peripherals in a very natural way. Currently mbed does not provide multi-tasking capabilities. The entire program is a single task. Multiple accesses to the different interfaces are hard to handle. One possible workaround is to use an operating system. This allows the user to create tasks, for example one task for the Ethernet-interface and another one for the

serial-interface. An operating system gives also some possibilities to let the task communicate to each other. Here we show how to use FreeRTOS on the LPC1768 mbed.

Can we run FreeRTOS on the mbed?

Yes, we can! There is no way to choose the compiler flags used in the cloud, never mind setting up a linker-script. But there is a very simple way to use FreeRTOS on the mbed. First some background information about the structure of FreeRTOS. It consists of four main files, which represents the actual kernel. There is *tasks.c*, which implements the scheduler for switching between tasks. Functions for queue handling are defined in the file *queue.c*. Queues are used inside the kernel itself and messages passing between tasks. There is also *croutines.c*, which implements co-runtines, and *list.c*, which defines a linked list type.

There are some more files that are needed. For example some memory management routines like malloc and free is provided by the file called *heap_x.c*, some header files as well as a file which realize the porting, *port.c*.

To get a working version of FreeRTOS there are two possibilities. The first one is to import the required files into workspace and compile it. The other one is to make a precompiled (partial linked) object. Both possibilities have some advantages and disadvantages.

Let us begin with the advantages of both opportunities. Importing everything into the project gives the advantage, that the configuration is easy changeable. On the other side to have a precompiled object filed avoid any problem caused by the online compiler. The advantage of the second variant shows the disadvantage of the first one. The online compiler will not compile the source files of FreeRTOS without a couple of changes. An offline compiler will translate the source files without any change. So it is possible to generate object files and add them to the project. The header files have still to be imported as well, but they will be compiled without modifications by the online compiler.

Different ways to use FreeRTOS

There are some possibilities to use external material in own projects with mbed. The online compiler is able to use object files and every one can create libraries with them and publish it. Libraries will be used in existing projects and brings additional functionality. It is also possible to publish a whole project. This is normally done for examples.

In the description above it is explained that there are two different ways to compile FreeRTOS and use it. Both ways are available as library as well as example project. The libraries can be used to have a quick start up. A fast integration into existing projects is also possible. On the other side the example projects allow a deeper look into the code. It is also possible to modify the settings. Although never used FreeRTOS, an initial step can be found here.

However, when using an existing project or library the files, which are called, must not be downloaded. To get a running version of FreeRTOS on the mbed, importing the called projects or libraries will suffice.

Using object files in the project

First download FreeRTOS from http://www.freertos.org and extract the archive. All references to the FreeRTOS source code in this document are based on version 6.1.0. For compiling FreeRTOS with GCC, we recommend using "Sourcery G++ Lite", from CodeSourcery. Now all files that are needed to compile FreeRTOS should be identified and copied into two folders, one named *source* for the C source files and a second one, *include*, for header files.

The following files belong into *source*:

- *FreeRTOS\Source\croutines.c*
- *FreeRTOS\Source\list.c*
- *FreeRTOS\Source\tasks.c*
- *FreeRTOS\Source\portable\GCC\ARM_CM3\port.c*
- *FreeRTOS\Source\portable\MemMang\heap_2.c*

These header files go into *include*:

- *FreeRTOS\Source\include*.h*
- *FreeRTOS\Source\portable\GCC\ARM_CM3\portmacro.h*
- *FreeRTOS\Source\Demo\CORTEX_LPC1768_GCC_RedSuite\src\core_cm3.h*
- *FreeRTOS\Source\Demo\CORTEX_LPC1768_GCC_RedSuite\src\LPC17xx.h*
- *FreeRTOS\Source\Demo\CORTEX_LPC1768_GCC_RedSuite\src\system_LPC17xx.h*
- *FreeRTOS\Source\Demo\CORTEX_LPC1768_GCC_RedSuite\src\FreeRTOSConfig.h*

Because the settings are used now cannot be changed later without recompile, there should be take a look at the *FreeRTOSConfig.h*. Here are some configurations that should be changed but they need not necessary to be changed. The first value is configCPU_CLOCK_HZ, which is set to 9900000. The correct value for the mbed is 9600000. On the other side there is reserved heap in configTOTAL_HEAP_SIZE. It is recommended to change it to 4 kB instead of the given 19 kB. The heap is used for malloc and free, nut only used by the FreeRTOS kernel and maybe your user application.

Now the compilation is on the term. The following command line shows the used parameters to compile the source files:

```
arm-none-eabi-gcc -c -fno-builtin -ffunction-sections -marm
-mno-thumb-interwork -funwind-tables –mthumb
-Wa,-mimplicit-it=always  -march=armv7-m -Wa,-march=armv7-m
-msoft-float -Iinclude --short-wchar -fshort-enums -Os *.c
```

Option	Description
-c	Compile, generate object file
-fno-builtin	Use no builtin functions of the compiler
-ffunction-sections	Generate for a function a section. So only functions that are used will be added to the binary.
-marm	Target is an ARM device
-mno-thumb-interwork	The assembler must not generate interwork instructions
-funwind-tables	Use ARM unwind tables
-mthumb	Target is Thumb device
-Wa,-mimplicit-it=always	The assembler generate automatically IT-instructions
-march=armv7-m **-Wa,-march=armv7-m**	Tells the compiler and the assembler it is device with a ARMv7-M architecture
-msoft-float	Use soft-float
-Iinclude	Tells the compiler where to find the header files
--short-wchar	Use 16 bit for wide char. Is needed for compatibility with the online compiler.
-fshort-enums	Use 16 bit for enumerations. Is needed for compatibility with the online compiler
-Os	Generate size optimized output

The six resulting object files will have to be pre-linked into a single object file using the command below:

```
arm-none-eabi-ld -r -o freertos.o croutine.o list.o queue.o tasks.o
port.o heap_2.o
```

Option	Description
-r	Generate relocatable output (partial linking)
-o <filename>	Output file is *<filename>*

Now generate a new project in the online compiler. Generate a new Folder into this project called "*FreeRTOS*" with a subfolder called "*include*". To import files to a folder right click it and choose "Import Files...".

Import the following file to the "FreeRTOS" folder:

- FreeRTOS.o

Import the following files to the "include" folder:

- *FreeRTOS\Source\include*.h*
- *FreeRTOS\Source\portable\ RVDS\ARM_CM3\portmacro.h*
- *FreeRTOS\Demo\CORTEX_LPC1768_GCC_RedSuite\src\core_cm3.h*
- *FreeRTOS\Demo\CORTEX_LPC1768_GCC_RedSuite\src\LPC17xx.h*
- *FreeRTOS\Demo\CORTEX_LPC1768_GCC_RedSuite\src\FreeRTOSConfig.h*

Now copy the following code to the *main.c*:

```
#include "main.hpp"

Serial pc(USBTX, USBRX);
xSemaphoreHandle xSerialPcSemaphore;

void ethTask(void *param);
void ledTask(void *param);

int main() {
    //vSemaphoreCreateBinary(xSerialPcSemaphore);
    prvSetupHardware2();
    pc.printf("This sample shows you how to run FreeRTOS on the
               mbed!\r\n");
    pc.printf("Setting up tasks...");

    xTaskCreate( ethTask, ( signed char * ) "EthTask", 500,
                ( void * ) NULL, tskIDLE_PRIORITY, NULL );
    xTaskCreate( ledTask, ( signed char * ) "LedTask",
                configMINIMAL_STACK_SIZE, ( void * ) NULL,
                tskIDLE_PRIORITY, NULL );
    pc.printf("done.\r\nStarting scheduler!\r\n");
    vTaskStartScheduler();
    while(1); // should never reach here
}
```

In this main function to tasks will be created. One handles the network port of mbed and the other one sets the LEDs. However, the main starts with a hardware setup called `prvSetupHardware2`. Here the serial port will be set up and the interrupt vector table will be copied and adapted. In the next step the two tasks will e created. Now everything is ready to run, so the Scheduler can be started.

The hardware setup is done as the following: (copy this to *init.c*)

```
#include "main.hpp"

unsigned long intbuf[512];
unsigned long *intvec;

void prvSetupHardware2(void){
    unsigned long *oldintvec = 0x0;
    pc.baud(115200);
    pc.printf("Starting...\r\n");
    pc.printf("Copy interrupt-vectors...");
    intvec = (unsigned long *)(((unsigned long) intbuf + 255) & (~ 255));

    for (int x = 0; x < 256; x++)
        intvec[x] = oldintvec[x];
    intvec[11] = (unsigned long) vPortSVCHandler;
    intvec[14] = (unsigned long) xPortPendSVHandler;
    intvec[15] = (unsigned long) xPortSysTickHandler;

    *VTOR = (unsigned long) intvec;
    pc.printf("done.\r\n");

    pc.printf("Hardware setup...");
    prvSetupHardware();
    pc.printf("done.\r\n");
}
```

As a short description, the interrupt vector table has to be 256 byte aligned and has a size of 256 bytes. The simplest way to make this possible is to use a 512 byte buffer. In this buffer there must be place that matches to these conditions. After this is done, the old vector table is copied into the buffer and needed interrupts will be set. The last step, before calling the FreeRTOS hardware setup, is to set the new address of the interrupt vector table to the VTOR-register.

This Project has been published as “FreeRTOS with object file” and should be imported as project. Alternatively a variant is available as “FreeRTOS with object file as library” and should be imported as library to an existing project.

Using source files in the project

The following table shows the modifications on the kernel itself to get it complied with the online compiler:

File	Line	Description
StackMacros.h	101	taskFIRST_CHECK_FOR_STACK_OVERFLOW: Use xTaskHandle instead of xTaskHandle * as declared
	118	taskFIRST_CHECK_FOR_STACK_OVERFLOW: Use xTaskHandle instead of xTaskHandle * as declared
	140	taskFIRST_CHECK_FOR_STACK_OVERFLOW: Use xTaskHandle instead of xTaskHandle * as declared
	165	taskFIRST_CHECK_FOR_STACK_OVERFLOW: Use xTaskHandle instead of xTaskHandle * as declared
list.h	187	wrapper to the new cast variant of this macro
	188	cast problem, solving by intruducing a cast parameter
	198	use of the cast parameter
task.h	114	removing a const declaration, which causes a lot of compiler warnings
tasks.c	1620	replace with the new macro from list.h and (tskTCB *) cast parameter
	2068	replace with the new macro from list.h and (volatile tskTCB *) cast parameter
	2071	replace with the new macro from list.h and (volatile tskTCB *) cast parameter
heap_2.c	139	replace (void *) cast with (A_BLOCK_LINK *)
	148	replace (void *) cast with (xBlockLink *)
	213	replace (void *) cast with (xBlockLink *)
	256	replace (void *) cast with (xBlockLink *)
croutine.c	320	replace with the new macro from list.h and (corCRCB *) cast parameter
queue.c	856	add a cast to (signed char *)

935	replace (void *) cast with (xTaskHandle *)
1058	replace (void *) cast with (xTaskHandle *)

Because this ends up in a lot of source code, it is recommended to import the files from the project. It is published as “FreeRTOS with source code”. It is also available as library as “FreeRTOS with source code as library” and should be imported as library to an existing project.

However, the procedure is the same as above, but the source files must also be imported into the project when it is created manually.

Is an RTOS on mbed useful for further applications and can it be used with other libraries?

A clear answer: yes. The example project with is published as “FreeRTOS example” demonstrate a web-server application with flashing LEDs and the use of message queues for the serial port. This is done in three tasks.

The advantage of an RTOS is obvious: Partitioning a complex application into individual tasks with different priorities and getting the whole benefits of real time scheduling.

The example application shows how to divide a program in different tasks allowing them to communicate. The first task (ethTask) initializes the Ethernet port and handles it. The second task (ledTask) flashes the led. All that is left to do for the main() function is setting up the tasks and starting the scheduler. After this there will be no more code execution in main().

What are the advantages and disadvantages of the use of an RTOS?

A great advantage of the use is that several tasks can be handled in multitasking mode. Without an operating system this must be done with counters or other mechanisms to make the handling possible.

The major disadvantage is that more flash and memory is needed. This is about 15 kb of flash and 6 kb RAM. But both are scalable with the integrated configuration options and can be adapt to the use of the application. Another “disadvantage” is that there is a need to think about the inter-task communication and possible deadlocks as well as strategies to avoid these pitfalls.

Conclusion

- It is possible to use object files generated with GCC in the online compiler and in general with the RVDS.

- Code might have to be modified to be able to compile it in the online compiler. The RealView compiler seems to implement stricter typing than GCC.
- Thanks to the ARM Cortex-M3 built-in OS-support, common to all devices, it is rather easy to use an existing port of FreeRTOS and port it to your own platform.