VORAGO VA108xx FreeRTOS port application note

Oct 21, 2016, Version 1.1

VA10800 / VA10820

Abstract

Real-Time Operating System (RTOS) is a popular software principle used for real-time applications to arbitrate CPU time across multiple tasks while minimizing buffering delays. FreeRTOS is a popular and simple real-time operating system kernel which has been ported to over 35 different microcontroller families from over 20 MCU suppliers, including some based on the ARM® Cortex®-M0 CPU.

This application note provides a brief guide to getting started with FreeRTOS ports on these cores with the Keil MDK IDE. Vorago's VA10800 and VA10820 are Cortex®-M0 based and the existing FreeRTOS M0 port was readily applied to them. It also provides step by step instructions for porting an existing project to FreeRTOS and includes sample code for the Vorago REB1-VA108x0 development board which implements a time-sharing "round-robin" solution.

A comprehensive guide to FreeRTOS, its operating principals, the source code, and its porting is available at http://www.freertos.org/

Table of contents

1 RTOS basics 2

2 RTOS kernel and support files 3

3 Modifying a Keil project for FreeRTOS 6

4 Creating FreeRTOS threads and starting the scheduler 6

5 Conclusion 10

6 Other Resources 12

1 RTOS basics

This section covers some of the terminology and structures used in RTOS based systems.

Threads are ongoing tasks implemented as an infinite loop and added to a queue. Thread and task are often used synonymously. A handle is necessary to reference a thread once it is created. Threads may be intended for static use, or allowed to have dynamic creation/removal.

While traditional non-RTOS solutions typically have a system of prioritized interrupts which can take control from the main program or lower priority interrupts, an RTOS differs in that active threads of the same priority will automatically share CPU control via arbitration.

There is a wide range of arbitration options. The simplest is time-sharing round-robin, where threads are switched back and forth on a regular time period. It is preemptive in that a given thread does not yield control when it has completed, rather control is taken by the scheduler. This is implemented in the example code provided with this Application Note.

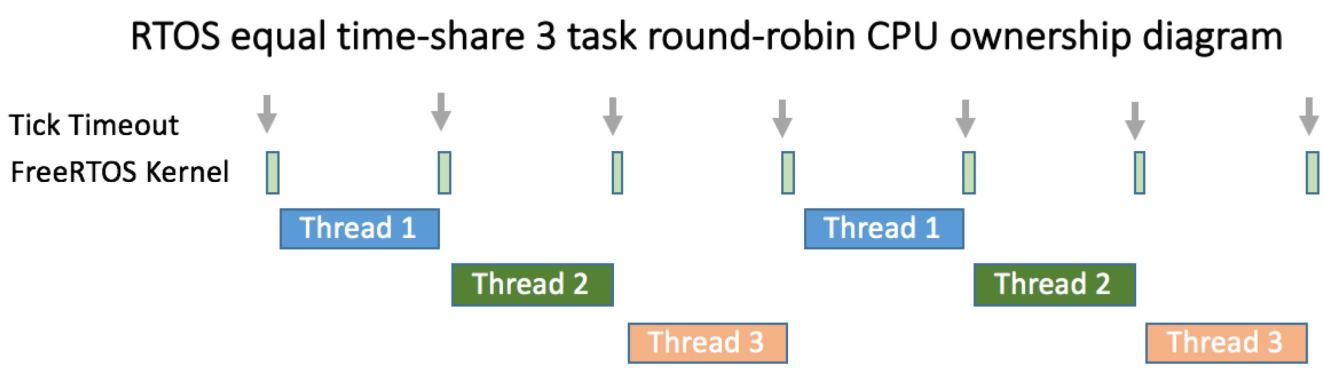

Figure 1 - Round-robin CPU time sharing diagram

Resources can be accessed by multiple tasks, yet it is often unsafe for a second thread to access the resource while it is already in use. Examples of resources could be the entire UART module or as simple as a common variable being changed while math operations are being performed on it. A similar conflict can exist in a non-RTOS system when an interrupt needs to access a resource being used by the main code or a lower priority interrupt.

One simple solution to this is masking/disabling interrupts during a critical section to stop arbitration. Control cannot be handed to another thread in that time, but this blocks all other threads entirely, not just attempts to access that resource. A more elegant solution is the use of semaphores to lock off access to specific resources to prevent other threads from accessing them. Semaphores are software structures that signal the availability of a resource. A mutex is a binary semaphore that can only be unlocked by the thread that locked it.

See http://www.freertos.org/Inter-Task-Communication.html for more details on these commonly used structures and inter-task communications.

2 RTOS kernel and support files

A standard file structure is used to allow FreeRTOS to be portable between different MCUs. This section reviews the structure which must be followed.

The FreeRTOS kernel itself is contained within 3 essential files:

1. tasks.c,
2. queue.c, and
3. list.c.

Four other support files are normally required:

1. Port.c is essential and contains architecture-specific code. Since M0 is a standardized architecture, a port.c made for any M0 based MCU should work.
2. Heap_1.c is the simplest of 5 memory management options for using a heap. A heap is a preserved area of memory that can be temporarily allocated to a task. For instance, a block of data from a serial bus may be temporarily stored in the heap area. When the data is processed, the allocated heap space is released. Memory management options are described in detail at http://www.freertos.org/a00111.html
3. FreeRTOSConfig.h is essential. This contains options specific to your application and should be located with your project.
4. Timers.c is only necessary if the application uses timers. This application note utilizes the tick counter which is part of the M0 CPU complex but not peripheral timers.

In order for the compiler to locate the above files, it must be instructed where to find them. To do this, start the Keil MDK IDE and open the C/C++ Options tab:

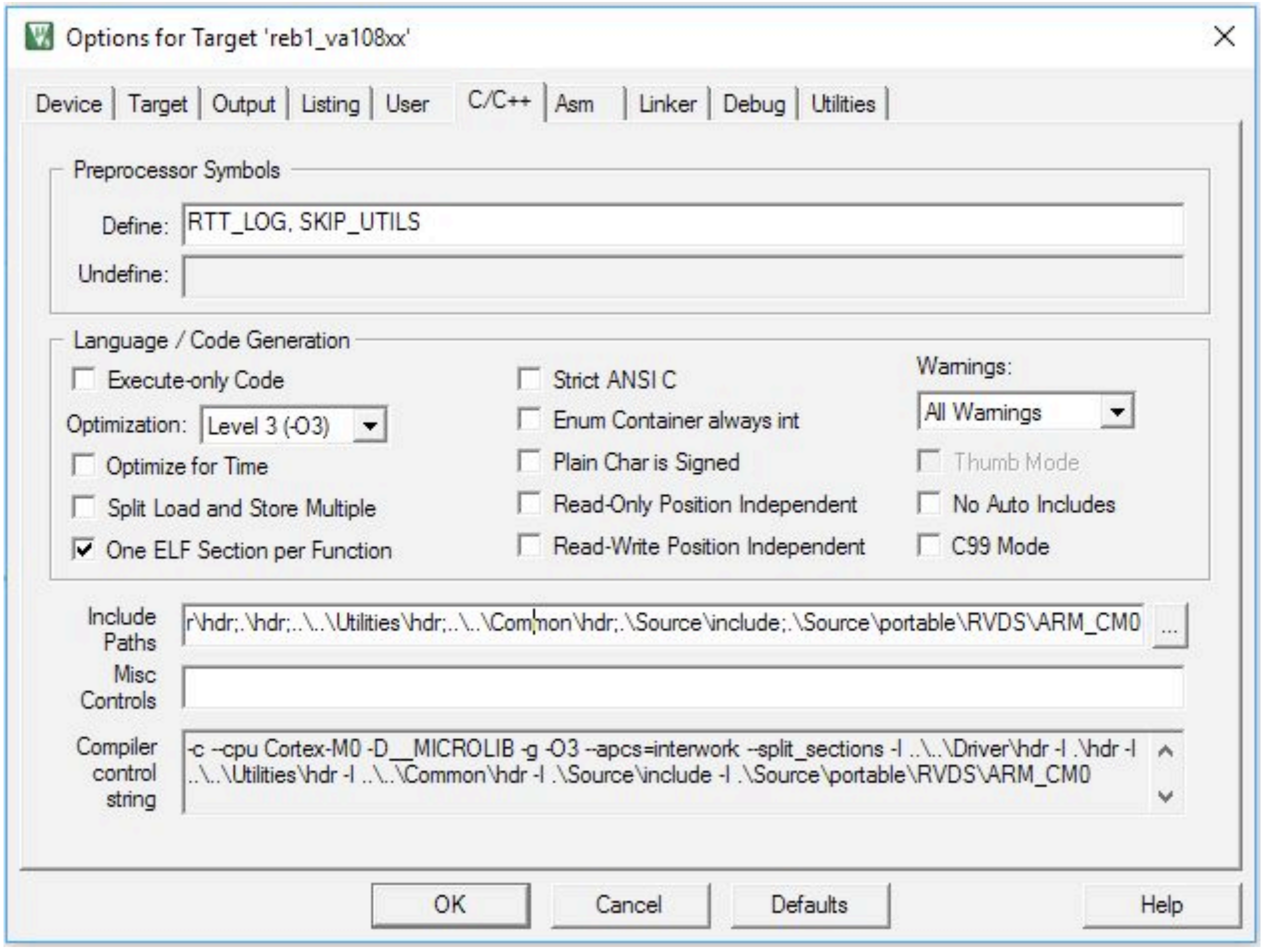

Figure 2 - Options menu from Keil IDE with include path definitions

Under the Include path field, add the FreeRTOS directories to your project's existing Includes:

- .\Source\include;
- .\Source\portable\RVDS\ARM_CM0
- .\Source\portable\MemMang

The example project has the required *.c and *.h files grouped together as shown here:

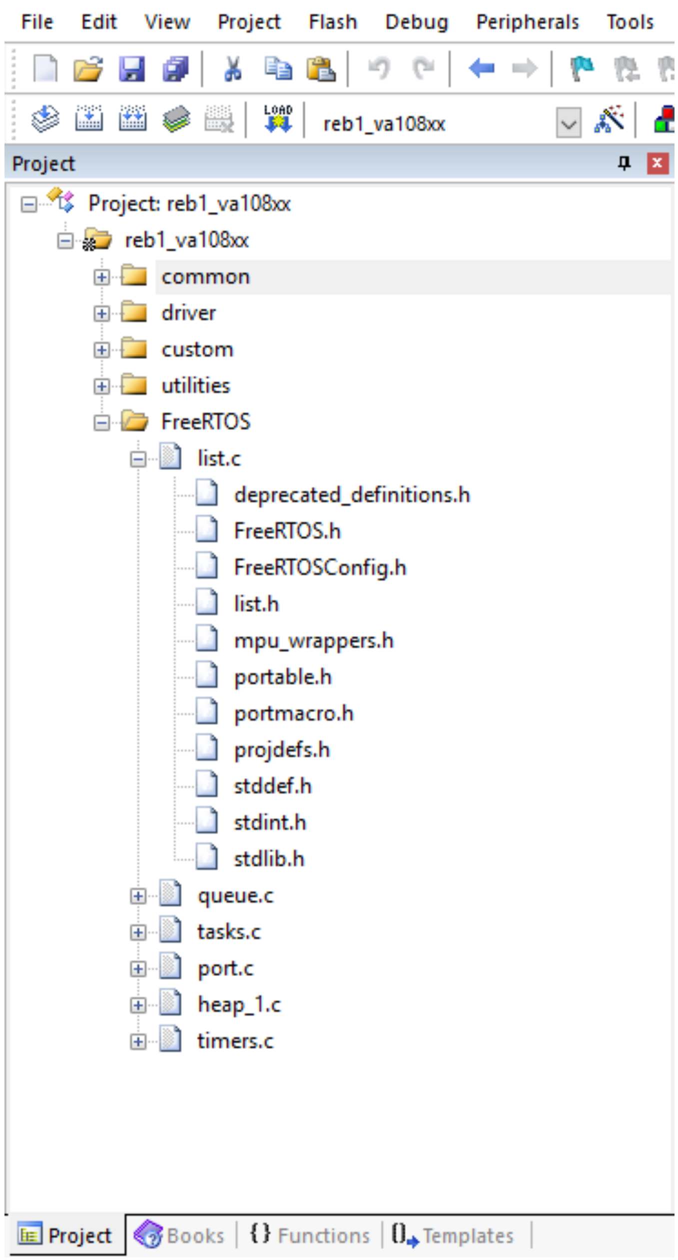

Figure 3 - File organization for the example project that accompanies this Application note

3 Modifying a Keil project for FreeRTOS

3.1 Step 1 – Main.c include and definition additions

In the main .c source file, add the following 3 lines:

- #include "FreeRTOS.h"
- #include "task.h"
- #define mainCREATE_SIMPLE_BLINKY_DEMO_ONLY 1

In FreeRTOS a "hook" is a callback function that can be invoked by any thread to perform a common task such as handling catastrophic events such as a stack overflow. The following hook tasks should be included with the code bodies given in the demonstration code included with this application note:

- void vApplicationMallocFailedHook(void)
- void vApplicationStackOverflowHook(TaskHandle_t pxTask, char *pcTaskName)
- void vApplicationTickHook(void)

Details on the function of these hooks is documented here: http://www.freertos.org/a00016.html

These hooks are not essential in all cases, and other optional hooks do exist.

3.2 Step 2 – FreeRTOSConfig.h changes

FreeRTOSConfig.h should be modified to configure the tick counter to schedule the task arbitration with a frequency = 500Hz (period = 2ms):

- #define configUSE_TICK_HOOK 1
- #define configTICK_RATE_HZ ((TickType_t) 500)

4 Creating FreeRTOS threads and starting the scheduler

4.1 Example program threads

The example program creates two new functions to be used as threads. Typically these will be while(1) loops which never end. The example uses "vLED1Code" and "vLED2Code".

If you want to control threads after creation, you will need to have a handle defined. These two lines placed early in main() routine creates handles for the two threads in the example project :

- TaskHandle_t xLED1Handle = NULL;
- TaskHandle_t xLED2Handle = NULL;

The application note example thread code is shown here.

```
void vLED1Code( void * pvParameters )
{
    while(1)
    {
        uint32_t i;
        for(i=0;i<100000;i++)
            __NOP();

            portENTER_CRITICAL();
        VOR_GPIO->BANK[0].DIR |= (1 << PORTA_10_D2);
          VOR_GPIO->BANK[0].DATAMASK |= (1 << PORTA_10_D2);
          VOR_GPIO->BANK[0].DATAOUT ^= (1 << PORTA_10_D2);
            portEXIT_CRITICAL();
    }
}

```

Figure 4 – Thread routine from the example program

4.2 Non-reentrant considerations

Any operation which cannot be interrupted should protected. One option to do this is to frame the vulnerable code with "portENTER_CRITICAL();" and "portEXIT_CRITICAL();"

For example:

```
 portENTER_CRITICAL();
VOR_GPIO->BANK[0].DIR |= (1 << PORTA_7_D3);
VOR_GPIO->BANK[0].DATAMASK |= (1 << PORTA_7_D3);
VOR_GPIO->BANK[0].DATAOUT ^= (1 << PORTA_7_D3);
portEXIT_CRITICAL();
```

If the MCU were to switch contexts once this section started, but before DATAOUT was written back, the other thread may alter the port registers, leading to undesirable results.

The context saving of the conventional interrupt structure protects the working registers when threads are switched. FreeRTOS uses any of the 5 inherent heap memory management structures to protect and preserve variables declared locally in a thread. However, access to global variables and port registers are not inherently protected.

More information on kernel control can be found at http://www.freertos.org/a00021.html.

4.3 Adding a thread to scheduler list

After initializing the MCU's peripheral clocks and any other essentials, add the threads to scheduler list as documented at http://www.freertos.org/a00125.html . These two lines added the two tasks to the scheduler list in the example program.

- xTaskCreate(vLED1Code, "LED1Flasher", configMINIMAL_STACK_SIZE, NULL, 2, xLED1Handle);
- xTaskCreate(vLED2Code, "LED2Flasher", configMINIMAL_STACK_SIZE, NULL, 2, xLED2Handle);

See Figure 6 - Main routine from example application for placement of these lines in the example program.

Each field in the brackets is explained in the below diagram.

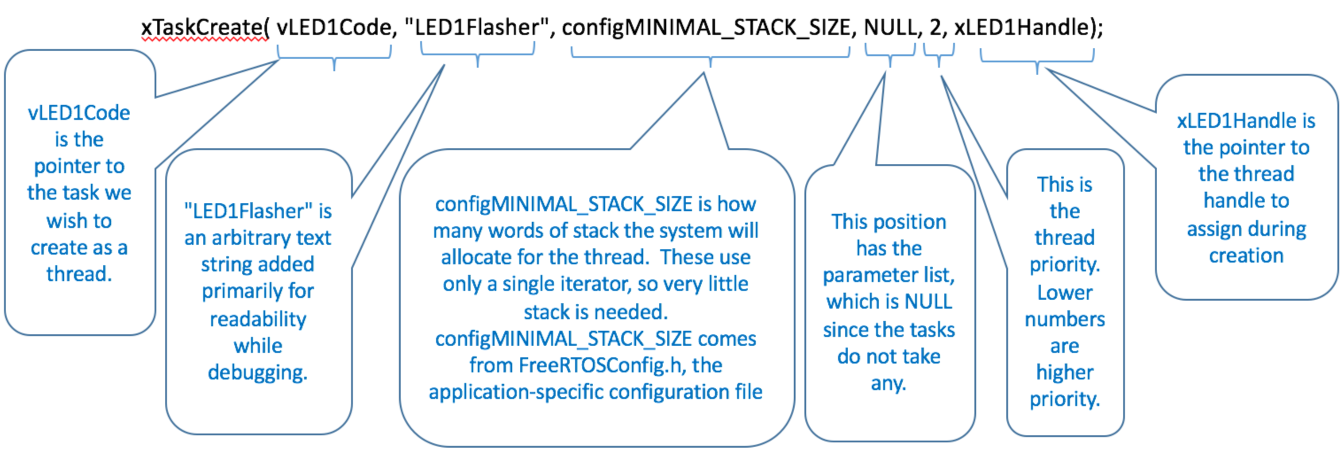

Figure 5 - Explanation of parameters passed to the xTaskCreate routine.

4.4 Start the FreeRTOS scheduler

At this point the threads are on a list, but not running.

After all thread creation, add:

vTaskStartScheduler();

return 0; //Will never be reached

This permanently turns control over to FreeRTOS scheduler. vTaskStartScheduler() should never return so no code below it will ever execute.

The MCU should switch between vLED1Code and vLED2Code every 2 ms on a time-sharing "round-robin", based on the 500Hz variable defined in FreeRTOSConfig.h. Each task receives half of the CPU bandwidth.

The application note example main routine is shown here

```
int main()
{
    uint8_t cur_temp = 0;
    int32_t ret_code = 0;
    uint16_t adc_data = 0;
    TaskHandle_t xLED1Handle = NULL;
    TaskHandle_t xLED2Handle = NULL;

#ifdef GPIO_DRIVER
    VOR_DRIVER_GPIO *PA10 = &Driver_GPIOA_10;
#endif

    /* Enable clock for all peripherals */
    VOR_SYSCONFIG->PERIPHERAL_CLK_ENABLE = ( CLK_ENABLE_PORTA | CLK_ENABLE_PORTB | CLK_ENABLE_SPIA | CLK_ENABLE_SPIB | CLK_ENABLE_SPIC |
                                              CLK_ENABLE_UARTA | CLK_ENABLE_UARTB | CLK_ENABLE_I2CA | CLK_ENABLE_I2CB | CLK_ENABLE_IRQSEL |
                                              CLK_ENABLE_IOMGR | CLK_ENABLE_UTILITY | CLK_ENABLE_PORTIO | CLK_ENABLE_SYSTEM );
    xTaskCreate( vLED1Code, "LED1Flasher", configMINIMAL_STACK_SIZE, NULL, 2, xLED1Handle);
    xTaskCreate( vLED2Code, "LED2Flasher", configMINIMAL_STACK_SIZE, NULL, 2, xLED2Handle);
    vTaskStartScheduler();

    return 0;

```

Figure 6 - Main routine from example application

5 Running the example project

The example software that accompanies this document (AN1203_SW.zip) was developed in the Keil MDK development environment. To run the project, the Keil MDK must be downloaded and installed. Since the file is below 32kbytes, the evaluation version of the IDE can be used.

Installing Keil MDK and loading the va108xx.pack file is detailed in the REB1 User manual in section 2. See http://www.voragotech.com/products/reb1.

After the IDE is installed, unzip the AN1203_SW.ZIP file. Navigate to .mcu/project/FreeRTOS folder and click on the project file, "reb1_va108xx.uvprojx". This will start the IDE and load the FreeRTOS example project. Build the project by selecting "Rebuild all target files" in the Project pulldown menu. Download the project to the REB1 board by selecting "Start/Stop Debug Session" under the Debug pulldown menu. Run the program by selecting "Run" under the Debug pulldown menu. Two LEDs, D2 and D3, should be flickering with D3 noticeably longer on/off time.

5.1 Memory use and task switch time

As detailed in the below table, the FreeRTOS portion of this program uses approximately 4.1kbytes of ROM space. The heap size was inadvertently set to a large value of 11k bytes but could be reduced to 1k byte without any change to the performance of this simple two task example.

The task switch time was measured on an oscilloscope and determined to be 5.8 microseconds with a 50 MHz clock. (~290 cycles to switch tasks)

Example Project map file summary table

Code	(inc.data)	RO data	RW data	ZI data	Object Name
56	16	0	0	0	driver.common.o
14	0	0	0	0	hardfault_handler.o
106	14	0	8	11000	heap_1.o
352	124	0	80	0	i2c_drv_api.o
3290	106	4	464	0	i2c_va108xx.o
552	50	0	0	0	irq_va108xx.o
136	0	0	0	0	list.o
360	58	0	4	0	port.o
1446	4	0	0	64	queue.o
246	52	0	0	0	reb_main.o
64	12	0	2	0	reb_timer.o
0	0	0	0	168	segger_rtt.o
480	180	0	120	0	spi_drv_api.o
1848	34	4	660	0	spi_va108xx.o
36	8	192	0	1024	startup_va108xx.o
16	8	0	4	0	system_va108xx.o
1820	156	0	60	200	tasks.o
776	74	0	20	40	timers.o
Total for FreeRTOS portion of project					
3868	**232**	**0**	**72**	**11264**	

Table 1 - Memory Use summary table

6 Conclusion

This application note provides step-by-step instructions to implement a very simple project using FreeRTOS to schedule tasks. The associated software project has the necessary files to allow a user to quickly port their own VA108xx MCU program to a FreeRTOS based application.

7 Other Resources

General RTOS information: https://en.wikipedia.org/wiki/FreeRTOS

Specifics on FreeRTOS: http://www.freertos.org/

Vorago MCU products: http://www.voragotech.com/vorago-products

Vorago Application notes: http://www.voragotech.com/resources