CG3002 Embedded Systems Design Project
Porting FreeRTOS 8 to the Arduino Mega

Note: You should read and complete the steps in "Compiling and Using Arduino Libraries in Atmel Studio 6" before reading this porting guide. This will ensure that you have the Arduino libraries compiled correctly and tested before you proceed.

1. Introduction

FreeRTOS is a professionally developed, strictly quality-controlled, high quality and robust supported Real Time Operating System from Real Time Engineers Ltd. It is also free to use for all commercial and non-commercial products. The main restriction is that modifications made to the FreeRTOS side of the API must remain open source. Any code on the application side of the API can remain closed source.

FreeRTOS is supported on 18 toolchains across 25 platforms. You can obtain more information on FreeRTOS at http://www.freertos.org.

2. Obtaining FreeRTOS

You can obtain the latest version of FreeRTOS from https://sourceforge.net/projects/freertos/files/latest/download?source=files. The latest version is 8.2.2. However the sensors and this manual were tested on 8.1.2, which is recommended. You can obtain 8.1.2 from http://sourceforge.net/projects/freertos/files/FreeRTOS/V8.1.2/.

We will assume that you have unzipped FreeRTOS into C:\FreeRTOSV8.1.2.

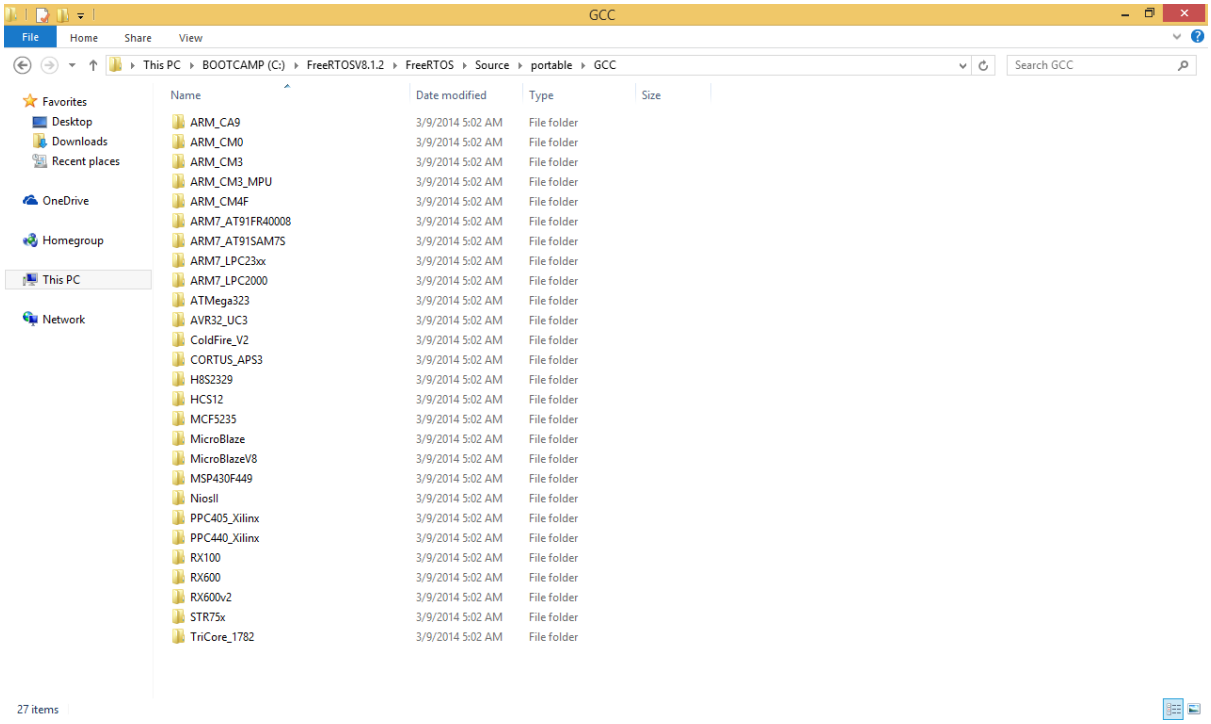
3. How FreeRTOS is Organized

The FreeRTOS directory is organized into FreeRTOS and FreeRTOS-plus directories. FreeRTOS-plus is an enhanced version of FreeRTOS that includes support for FAT, TCP/IP and other cool stuff, but sadly it is not free. We will therefore stay with FreeRTOS. To save disk space you may choose to delete FreeRTOS-plus.

The FreeRTOS directory is organized into Demo and Source directories. As their name suggests, the Demo directory consists of demo code for many platforms. We will not most of this directory, but you may wish to retain it for some reference code. The directory you will be most interested in is **C:\FreeRTOSV8.1.2\FreeRTOS\Demo\AVR_ATMega323_WinAVR**, which contains demo code for the ATMega323. You will also need the **C:\FreeRTOSV8.1.2\FreeRTOS\Demo\Common\include** directory. You can delete all directories except **C:\FreeRTOSV8.1.2\FreeRTOS\Demo\AVR_ATMega323_WinAVR** and **C:\FreeRTOSV8.1.2\FreeRTOS\Demo\Common\.**

The source directory is divided into the **portable** and **include** directories. Navigate to **C:\FreeRTOSV8.1.2\FreeRTOS\Source\portable** and you will see a directory for the 18 toolchains that are supported by FreeRTOS. The Atmel Studio suite uses GCC as its backend compiler, so naviage to **C:\FreeRTOSV8.1.2\FreeRTOS\Source\portable\GCC**.

You will now see a list of supported microcontrollers, similar to what you see below:



You will immediately see one piece of bad news; the Arduino Mega runs on the ATMega 2560 chip, which is, rather inconveniently, not supported by FreeRTOS.

Thankfully though the ATMega 323 is reasonably close to the 2560, and you will modify code written for the 323 to run on the 2560.

4. Porting

1. If you have not already done so, download and install the Arduino IDE from http://www.arduino.cc

2. Complete the steps in “Compiling and Using Arduino Libraries in Atmel Studio 6” also found in the IVLE workbin.

3. To begin, copy the ATMega323 directory to a new directory called ATMega2560. This will allow you to port the ATMega323 code to the 2560 without affecting the original code.

4. Start Atmel Studio 6 and create a new GCC C Executable project called FreeRTOS2560. Select ATMega 2560 from the Device Selection dialog box.

5. Add in the Arduino include directories. Assuming that your Arduino IDE is installed in C:\Program Files\Arduino, the directories you need to include are:

    i. C:\Program Files\Arduino\hardware\arduino\cores\arduino
    ii. C:\Program Files\Arduino\hardware\arduino\variants\mega

6. Add in libarduino.a in the Libraries textbox in AVR/GNU Linker, and add in the path to the directory where you compiled the libarduino.a file.

7. Copy FreeRTOSConfig.h from C:\FreeRTOSV8.1.2\FreeRTOS\Demo\AVR_ATMega323_IAR to C:\FreeRTOSV8.1.2\FreeRTOS\Source\include.

8. Open FreeRTOSConfig.h.

9. The file iom323.h is not included in Atmel Studio since the ATMega323 is not supported. So look in FreeRTOSConfig.h and replace the line:

   ```
   #include <iom323.h>
   ```

   With:

   ```
   #include <avr/io.h>
   ```

10. Add in the include directories for FreeRTOS. Assuming that FreeRTOS is installed in C:\FreeRTOSV8.1.2, the directories you need are:

    i. C:\FreeRTOSV8.1.2\FreeRTOS\Demo\Common\include
    ii. C:\FreeRTOSV8.1.2\FreeRTOS\Source\include
    iii. C:\FreeRTOSV8.1.2\FreeRTOS\Source\portable\GCC\ATMega2560

11. You now need to add in the main source files for the FreeRTOS kernel. Add in the following files:

    C:\FreeRTOSV8.1.2\FreeRTOS\Source\list.c
    C:\FreeRTOSV8.1.2\FreeRTOS\Source\queue.c
    C:\FreeRTOSV8.1.2\FreeRTOS\Source\tasks.c
    C:\FreeRTOSV8.1.2\FreeRTOS\Source\portable\MemMang\heap_1.c
    C:\FreeRTOSV8.1.2\FreeRTOS\Source\portable\GCC\ATMega2560\port.c

12. Compile your project. You will see that it fails with errors in port.c This is expected since we are building for the ATMega2560 and not the ATMega323, and the ATMega323 has completely different register names from the ATMega2560.

5. Customizing FreeRTOS

The fun begins. You will need to do four main things:

i. You need to configure the OS to use the correct clock rate and include only the stuff you need.
ii. You need to replace the code that initializes the ATMega's timers.
iii. You need to replace the interrupt handler for the timer.
iv. You need to fix the context save and restore code, as well as the stack initialization code to cater for the RAMPZ and EIND registers, and the fact that the program counter on the Mega is 3 bytes long, while the program counter on the ATMega323 (and any other ATMega) is only 2 bytes long.

All porting is done in the file port.c. To begin, open this file in the code editor. Open as well the FreeRTOSConfig.h file.

a. Setting Up the Clock Rate and Tick Rate

The first thing we must do is to configure the CPU's clock rate and the tick rate of the OS. The tick rate is the rate at which FreeRTOS triggers the timer interrupt (see later). To configure, locate and open the FreeRTOSConfig.h file. Look for the line:

```
#define configCPU_CLOCK_HZ    ( ( unsigned long ) 8000000 )
```

This tells FreeRTOS that the CPU clock is 8MHz, which is fine for the ATMega323 but not the ATMega2560. Change 8000000 to 16000000:

```
#define configCPU_CLOCK_HZ    ( ( unsigned long ) 16000000 )
```

Immediately below you should see:

```
#define configTICK_RATE_HZ    ( ( TickType_t ) 1000 )
```

This sets the tick rate to 1000, which is good for us. This means that the timer interrupt is triggered every millisecond.

There are several other things that we will not use for now, including the idle hook and tick hook. These are routines that are triggered when the OS is idle, or when the clock interrupt is triggered. In addition we will not be using coroutines for now (similar to threads). Hence change:

```
#define configUSE_IDLE_HOOK          1
```

To:

```
#define configUSE_IDLE_HOOK          0
```

By default FreeRTOS is not configured to use tick hooks, so just ensure that you see:

```
#define configUSE_TICK_HOOK                0
```

We will now disable co-routines. Change:

```
#define configUSE_CO_ROUTINES              1
```

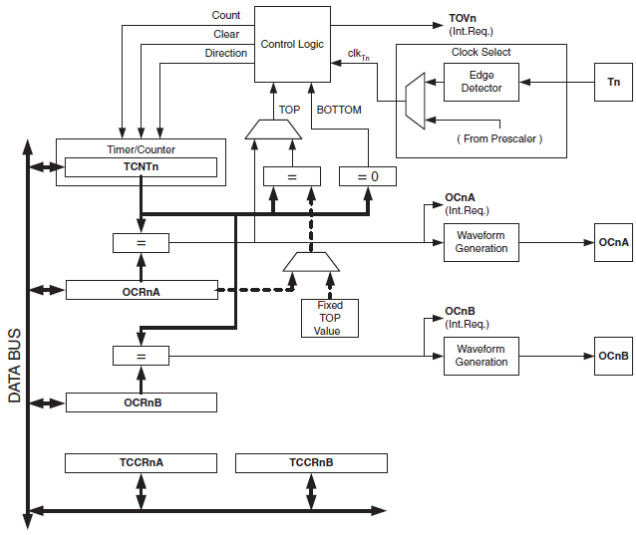
To:

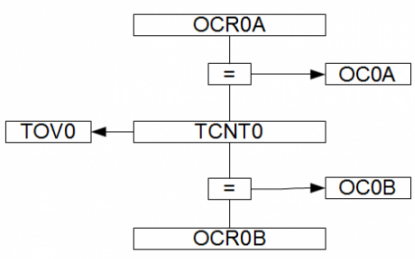
```
#define configUSE_CO_ROUTINES              0
```

We will leave the rest of the configuration entries alone for now.

b. Setting Up the Timer

The Arduino Mega has two 8-bit timers (Timers 0 and 2) and four 16-bit timers (Timers 1, 3 to 5). We will program Timer 2 in something called “Clear Timer on Compare” mode, otherwise known as CTC mode.





Timer 0 uses a register called TCNT0 which holds the current counter value (think of “registers” has being variables that can hold values. Sometimes these values control the behaviour of a piece of hardware, in which case it is called a “configuration register”)

There are also another two registers called the Output Compare Registers, which on timer 0 are called OCR0A and OCR0B.

The idea is that TCNT0 will begin counting from 0, and will increment at a fixed period until it reaches the value stored in either OCR0A or OCR0B. When the value in TCNT0 matches, for example, OCR0A, an interrupt is generated and TCNT0 is reset to 0 and starts counting again.

Timer 0 uses two configuration registers TCCR0A and TCCR0B. We now look at the steps involved in setting up Timer 0.

i. Decide on the Prescaler value P and compare value V.

The Prescalar value P decides the interval between increments to TCNT0, while the compare value V decides when TCNT0 will roll over to 0 and trigger an interrupt. The number of times N to trigger per second is related to P and V by:

$$N = \frac{1000000 \times P \times V}{F}$$

Where F is the CPU clock rate. Note that both N and F are known values, provided in FreeRTOSConfig.h as configCPU_CLOCK_HZ and configTICK_RATE_HZ. Rewriting to solve for V we have:

$$V = \frac{configTick\_Rate\_HZ \times configCPU\_CLOCK\_HZ}{1000000 \times P}$$

Wait, what? Isn't P also unknown? Happily P takes on only a limited set of values as shown in the table below:

$$\text{Resolution} = \frac{1}{(F_{clk} / P)}$$

CS02	CS01	CS00	Prescalar P	Resolution (F_{clk}=16 MHz)
0	0	0	Stops the timer	-
0	0	1	1	0.0625 microseconds
0	1	0	8	0.5 microseconds
0	1	1	64	4 microseconds
1	0	0	256	16 microseconds
1	0	1	1024	64 microseconds
1	1	0	External clock on T0. Clock on falling edge.	-
1	1	1	External clock on T0. Clock on rising edge	-

Bearing in mind that V will be loaded into TCNT0 which is an 8-bit register, V can at most be 255. We can now find P by trial and error:

P	V=(1000 * 16000000)/1000000*P
1	16000
8	2000
64	250
256	62.5
1024	15.625

Since V values of 62.5 and 15.625 cannot be loaded accurately into TCNT0, we will use a prescalar value of 64.

Hence we have P=64, and:

$$V = \frac{configTick\_Rate\_HZ \times configCPU\_CLOCK\_HZ}{64000000}$$

ii. Write the interrupt handler code.

i. Within port.c locate the following piece of code:

```
#if configUSE_PREEMPTION == 1

	/*
	 * Tick ISR for preemptive scheduler.  We can use a naked attribute as
	 * the context is saved at the start of vPortYieldFromTick().  The tick
	 * count is incremented after the context is saved.
	 */
	void SIG_OUTPUT_COMPARE1A( void ) __attribute__ ( ( signal, naked ) );
	void SIG_OUTPUT_COMPARE1A( void )
	{
		vPortYieldFromTick();
		asm volatile ( "reti" );
	}
#else

	/*
	 * Tick ISR for the cooperative scheduler.  All this does is increment the
	 * tick count.  We don't need to switch context, this can only be done by
	 * manual calls to taskYIELD();
	 */
	void SIG_OUTPUT_COMPARE1A( void ) __attribute__ ( ( signal ) );
	void SIG_OUTPUT_COMPARE1A( void )
	{
		xTaskIncrementTick();
	}
#endif
```

Notice the strange declaration void `SIG_OUTPUT_COMPARE1A(void) __attribute__ ((signal, naked));` ? This is the old-fashioned way to create interrupt service routines. Today the easier way is to use the ISR macro.

Replace the code above with:

```
#if configUSE_PREEMPTION == 1

	/*
	 * Tick ISR for preemptive scheduler.  We can use a naked attribute as
	 * the context is saved at the start of vPortYieldFromTick().  The tick
	 * count is incremented after the context is saved.
	 */
	ISR(TIMER0_COMPA_vect, ISR_NAKED)
	{
		vPortYieldFromTick();
		asm volatile ( "reti" );
	}
#else

	/*
	 * Tick ISR for the cooperative scheduler.  All this does is increment the
	 * tick count.  We don't need to switch context, this can only be done by
	 * manual calls to taskYIELD();
	 */
	ISR(TIMER0_COMPA_vect)
	{
		xTaskIncrementTick();
	}
#endif
```

Notice that the first argument of ISR is to declare which interrupt to hook (in this case the interrupt that triggers when TCNT0 == OCR0A), and in the first case the second argument declares that the handler must be “naked”. This means that the compiler generates the code for the handler exactly “as shown”, without any additional assembly code inserted.

iii. Deciding What To Write To TCCR0A.

Now we come to the meat: setting up the timer configuration registers. The first register is TCCR0A which sets up the timer’s mode, as well as what happens to pins connected to the timer (What?!! What does that even mean?!! Never mind for now; just know that these pins are used to generate PWM signals used by analogWrite).

Bit	7	6	5	4	3	2	1	0	
0x24 (0x44)	COM0A1	COM0A0	COM0B1	COM0B0	–	–	WGM01	WGM00	TCCR0A
Read/Write	R/W	R/W	R/W	R/W	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

We will set COM0A1, COM0A0, COM0B1 and COM0B0 to 0 0 0 0 so that we do not affect any external pins. The WGM01 and WGM00 bits control the timer’s mode.

Mode	WGM02	WGM01	WGM00	Timer/Counter Mode of Operation	TOP	Update of OCRx at	TOV Flag Set on[1][2]
0	0	0	0	Normal	0xFF	Immediate	MAX
1	0	0	1	PWM, Phase Correct	0xFF	TOP	BOTTOM
2	0	1	0	CTC	OCRA	Immediate	MAX
3	0	1	1	Fast PWM	0xFF	BOTTOM	MAX
4	1	0	0	Reserved	–	–	–
5	1	0	1	PWM, Phase Correct	OCRA	TOP	BOTTOM
6	1	1	0	Reserved	–	–	–
7	1	1	1	Fast PWM	OCRA	BOTTOM	TOP

We want to set up the timer in CTC mode so we use (WGM02, WGM01, WGM00) = (0,1,0). WGM02 is in TCCR0B.

Hence we set up TCCR0A as:

```
TCCR0A=0b00000010;
```

iv. Deciding What To Write To TCCR0B.

TCCR0B is shown below:

Bit	7	6	5	4	3	2	1	0	
0x25 (0x45)	**FOC0A**	**FOC0B**	–	–	**WGM02**	**CS02**	**CS01**	**CS00**	**TCCR0B**
Read/Write	W	W	R	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

We will set FOC0A, FOC0B and WGM02 all to 0. CS02, CS01 and CS00 are used to set the prescalar value. Write 000 here to turn off the timer.

Earlier we decided on a prescalar of 64. Consulting the table below again:

$$\text{Resolution} = \frac{1}{(F_{clk} / P)}$$

CS02	CS01	CS00	Prescalar P	Resolution (F_{clk}=16 MHz)
0	0	0	Stops the timer	-
0	0	1	1	0.0625 microseconds
0	1	0	8	0.5 microseconds
0	1	1	64	4 microseconds
1	0	0	256	16 microseconds
1	0	1	1024	64 microseconds
1	1	0	External clock on T0. Clock on falling edge.	-
1	1	1	External clock on T0. Clock on rising edge	-

This gives us a value of 011.

Hence we write:

```
TCCR0B=0b00000011;
```

v. Enabling Interrupts

To enable the timer interrupt for OCR0A, we need to set the OCIE0A bit of the Timer 0 Interrupt Mask Register TIMSK0 to 1.

Bit	7	6	5	4	3	2	1	0	
(0x6E)	–	–	–	–	–	OCIE0B	OCIE0A	TOIE0	TIMSK0
Read/Write	R	R	R	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Hence we do `TIMSK0|=0b10;`

vi. Patching the Timer Setup Code

Now that we have finally sorted all of that out, let's actually customize the OS. Timer set up is in `static void prvSetupTimerInterrupt( void )`. Locate the following code:

```
static void prvSetupTimerInterrupt( void )
{
uint32_t ulCompareMatch;
uint8_t ucHighByte, ucLowByte;

	/* Using 16bit timer 1 to generate the tick.  Correct fuses must be
	selected for the configCPU_CLOCK_HZ clock. */

	ulCompareMatch = configCPU_CLOCK_HZ / configTICK_RATE_HZ;

	/* We only have 16 bits so have to scale to get our required tick rate. */
	ulCompareMatch /= portCLOCK_PRESCALER;

	/* Adjust for correct value. */
	ulCompareMatch -= ( uint32_t ) 1;

	/* Setup compare match value for compare match A.  Interrupts are disabled
	before this is called so we need not worry here. */
	ucLowByte = ( uint8_t ) ( ulCompareMatch & ( uint32_t ) 0xff );
	ulCompareMatch >>= 8;
	ucHighByte = ( uint8_t ) ( ulCompareMatch & ( uint32_t ) 0xff );
	OCR1AH = ucHighByte;
	OCR1AL = ucLowByte;

	/* Setup clock source and compare match behaviour. */
	ucLowByte = portCLEAR_COUNTER_ON_MATCH | portPRESCALE_64;
	TCCR1B = ucLowByte;

	/* Enable the interrupt - this is okay as interrupt are currently globally
	disabled. */
	ucLowByte = TIMSK;
	ucLowByte |= portCOMPARE_MATCH_A_INTERRUPT_ENABLE;
	TIMSK = ucLowByte;
}
```

Replace this code with:

```
static void prvSetupTimerInterrupt( void )
{
	TCCR0A=0b00000010;
	TCNT0=0;
	OCR0A=configTICK_RATE_HZ * configCPU_CLOCK_HZ / 64000000;
	TIMSK0|=0b10;
	TCCR0B=0b00000011;
}
```

c. Fixing the Context Save, Context Restore and Stack Setup Routines

FreeRTOS has two macros called `portSAVE_CONTEXT` and `portRESTORE_CONTEXT` for saving and restoring the CPU context. Unfortunately this code does not work because the ATMega2560 has two additional registers called EIND and RAMPZ registers. Both registers allow jumps and data accesses to the much larger Flash memory available on the ATMega2560 as opposed to the smaller ATMega chips like the ATMega328P (used on the UNO) and the ATMega323.

In addition there is a function called `pxPortInitializeStack` that sets up the stack pointer to make it look like a task had been pre-empted even when it is being run for the first time. This routine makes it easier to write single uniform `portSAVE_CONTEXT` and `portRESTORE_CONTEXT` routines instead of having to handle first-time-runs as a special case.

i. **Fixing portSAVE_CONTEXT and PORTRESTORE_CONTEXT**

The portSaveContext macro saves the general purpose registers on the ATMega microcontrollers in preparation for a context switch. We need to modify this macro to save RAMPZ and EIND, found at memory locations 0x3B and 0x3C respectively. To do this, open port.c and look for the `portSAVE_CONTEXT` macro. Locate the following lines:

```
#define portSAVE_CONTEXT()                                          \
    asm volatile (  "push   r0                      \n\t"   \
                    "in     r0, __SREG__            \n\t"   \
                    "cli                            \n\t"   \
                    "push   r0                      \n\t"   \
                    "push   r1                      \n\t"   \
                    "clr    r1                      \n\t"   \
                    "push   r2                      \n\t"   \
                    "push   r3                      \n\t"   \
```

Right in between push r0 and push r1, type in the following code which will save RAMPZ and EIND:

```
"in r0, 0x3c                            \n\t"       \
"push r0                                \n\t"       \
"in r0, 0x3b                            \n\t"       \
"push r0                                \n\t"       \
```

Type it in exactly as you see, complete with quotes and backslashes at the end of the lines. Your `portSAVE_CONTEXT` macro will now look like this:

```
#define portSAVE_CONTEXT()                                          \
    asm volatile (  "push   r0                      \n\t"   \
                    "in     r0, __SREG__            \n\t"   \
                    "cli                            \n\t"   \
                    "push   r0                      \n\t"   \
                    "in r0, 0x3b                    \n\t"   \
                    "push r0                        \n\t"   \
                    "in r0, 0x3c                    \n\t"   \
                    "push r0                        \n\t"   \
                    "push   r1                      \n\t"   \
```

The additional code is shown in the red square above. Similarly locate the `portRESTORE_CONTEXT` macro in port.c, and scroll to the end of the macro

```
                    "pop    r7                      \n\t"   \
                    "pop    r6                      \n\t"   \
                    "pop    r5                      \n\t"   \
                    "pop    r4                      \n\t"   \
                    "pop    r3                      \n\t"   \
                    "pop    r2                      \n\t"   \
                    "pop    r1                      \n\t"   \
                    "pop    r0                      \n\t"   \
                    "out    __SREG__, r0            \n\t"   \
                    "pop    r0                      \n\t"   \
                );
```

Note that this is the `portRESTORE_CONTEXT` macro we are looking at now, not the `portSAVE_CONTEXT` macro! Now insert the following code, exactly as shown, between pop r1 and pop r0:

```
"pop r0                                 \n\t" \
"out 0x3c, r0                           \n\t" \
"pop r0                                 \n\t" \
"out 0x3b, r0                           \n\t" \
```

This will restore the EIND and RAMPZ registers in reverse order in which they were pushed in. When done the bottom of your `portRESTORE_CONTEXT` macro will look like this (red square shows the inserted code):

```
                    "pop    r3                      \n\t"   \
                    "pop    r2                      \n\t"   \
                    "pop    r1                      \n\t"   \
                    "pop r0                         \n\t"   \
                    "out 0x3c, r0                   \n\t"   \
                    "pop r0                         \n\t"   \
                    "out 0x3b, r0                   \n\t"   \
                    "pop    r0                      \n\t"   \
                    "out    __SREG__, r0            \n\t"   \
                    "pop    r0                      \n\t"   \
                );
```

ii. **Fixing pxPortInitializeStack**

When a task is first started, the `pxPortInitializeStack` function is called to place dummy values onto the task stack to make it look like the task had been pre-empted before, even though it is the first time the task is being run. This allows the task swapper to call `portRESTORE_CONTEXT` without having to check whether the task is being run for the first time. It also allows FreeRTOS to jump back to the task code correctly even at the first time the code is run.

`pxPortInitializeStack` currently caters only for 2-byte program counters (PC). You need to add an extra byte to the top of the stack to cater for the 3-byte PC that is used in the ATMega2560.

To begin, look for `pxPortInitializeStack` in port.c, and locate the following segment of code.

```
/* The start of the task code will be popped off the stack last, so place
it on first. */
usAddress = ( uint16_t ) pxCode;
*pxTopOfStack = ( StackType_t ) ( usAddress & ( uint16_t ) 0x00ff );
pxTopOfStack--;

usAddress >>= 8;
*pxTopOfStack = ( StackType_t ) ( usAddress & ( uint16_t ) 0x00ff );
pxTopOfStack--;

/* Next simulate the stack as if after a call to portSAVE_CONTEXT().
portSAVE_CONTEXT places the flags on the stack immediately after r0
to ensure the interrupts get disabled as soon as possible, and so ensuring
the stack use is minimal should a context switch interrupt occur. */
*pxTopOfStack = ( StackType_t ) 0x00;   /* R0 */
pxTopOfStack--;
```

Add in the following code just about the line that says "/* Next simulate the stack…":

```
*pxTopOfStack=0;
pxTopOfStack--;
```

Your code will now look like this (red square indicates added code)

```
/* The start of the task code will be popped off the stack last, so place
it on first. */
usAddress = ( uint16_t ) pxCode;
*pxTopOfStack = ( StackType_t ) ( usAddress & ( uint16_t ) 0x00ff );
pxTopOfStack--;

usAddress >>= 8;
*pxTopOfStack = ( StackType_t ) ( usAddress & ( uint16_t ) 0x00ff );
pxTopOfStack--;

*pxTopOfStack=0;
pxTopOfStack--;

/* Next simulate the stack as if after a call to portSAVE_CONTEXT().
portSAVE_CONTEXT places the flags on the stack immediately after r0
to ensure the interrupts get disabled as soon as possible, and so ensuring
the stack use is minimal should a context switch interrupt occur. */
*pxTopOfStack = ( StackType_t ) 0x00;   /* R0 */
pxTopOfStack--;
```

Now we must make extra space on the stack for the RAMPZ and EIND registers. Scroll a little further down until you see this segment of code:

```
/* Next simulate the stack as if after a call to portSAVE_CONTEXT().
portSAVE_CONTEXT places the flags on the stack immediately after r0
to ensure the interrupts get disabled as soon as possible, and so ensuring
the stack use is minimal should a context switch interrupt occur. */
*pxTopOfStack = ( StackType_t ) 0x00;   /* R0 */
pxTopOfStack--;
*pxTopOfStack = portFLAGS_INT_ENABLED;
pxTopOfStack--;

/* Now the remaining registers.   The compiler expects R1 to be 0. */
*pxTopOfStack = ( StackType_t ) 0x00;   /* R1 */
pxTopOfStack--;
*pxTopOfStack = ( StackType_t ) 0x02;   /* R2 */
pxTopOfStack--;
*pxTopOfStack = ( StackType_t ) 0x03;   /* R3 */
```

Add the following code just above the line that says “/* Now the remaining registers.“

```
*pxTopOfStack=(StackType_t) 0x00; /* EIND */
pxTopOfStack--;
*pxTopOfStack=(StackType_t) 0x00; /* RAMPZ */
pxTopOfStack--;
```

The code will now look like this. Red square indicates where code was added.

```
/* Next simulate the stack as if after a call to portSAVE_CONTEXT().
portSAVE_CONTEXT places the flags on the stack immediately after r0
to ensure the interrupts get disabled as soon as possible, and so ensuring
the stack use is minimal should a context switch interrupt occur. */
*pxTopOfStack = ( StackType_t ) 0x00;   /* R0 */
pxTopOfStack--;
*pxTopOfStack = portFLAGS_INT_ENABLED;
pxTopOfStack--;

*pxTopOfStack=(StackType_t) 0x00; /* EIND */
pxTopOfStack--;
*pxTopOfStack=(StackType_t) 0x00; /* RAMPZ */
pxTopOfStack--;

/* Now the remaining registers.   The compiler expects R1 to be 0. */
*pxTopOfStack = ( StackType_t ) 0x00;   /* R1 */
pxTopOfStack--;
```

Once done, build your project, et voila! FreeRTOS is now usable on the ATMega2560!

6. **Testing FreeRTOS**

   Connect up LEDs to pins 12 and 13 on the Arduino Mega. **PLEASE DO NOT FORGET TO USE A 330 OHM RESISTOR FOR EACH LED OR YOU WILL BURN THE LED!!!**

   Now open up the FreeRTOS2560.c file. Replace the entire contents with this:

```
/*
 * freertos2560.c
 *
*/


#include <avr/io.h>
#include <FreeRTOS.h>
#include <task.h>
#include <Arduino.h>

// Tasks flash LEDs at Pins 12 and 13 at 1Hz and 2Hz respectively.
void task1(void *p)
{
	while(1)
	{
		digitalWrite(12, HIGH);
		vTaskDelay(500); // Delay for 500 ticks. Since each tick is 1ms,
				 //this delays for 500ms.
		digitalWrite(12, LOW);
		vTaskDelay(500);
	}
}
```

```
void task2(void *p)
{
	while(1)
	{
		digitalWrite(13, HIGH);
		vTaskDelay(250);
		digitalWrite(13, LOW);
		vTaskDelay(250);
	}
}
#define STACK_DEPTH 64


void vApplicationIdleHook()
{
	// Do nothing.
}

int main(void)
{
	init();
	pinMode(12, OUTPUT);
	pinMode(13, OUTPUT);
	TaskHandle_t t1, t2;

	// Create tasks
	xTaskCreate(task1, "Task 1", STACK_DEPTH, NULL, 6, &t1);
	xTaskCreate(task2, "Task 2", STACK_DEPTH, NULL, 5, &t2);

	vTaskStartScheduler();
}
```

Build the program and use XLoader or avrdude to download to the Arduino Mega. You will see the LED at pin 12 blink at a rate of 1 Hz while the LED at pin 13 will blink at a rate of 2 Hz. Note if you use avrdude you should use –c wiring instead of –c arduino.

7. A Final Note About Timers

The code given here uses Timer 0 to trigger the FreeRTOS scheduler. This is often not a good idea because key Arduino routines like millis() and micros() also use Timer 0. You may want to rewrite the code to use Timer 2 instead. This is left as an exercise for you.

8. Further Reading

The full FreeRTOS API may be found at http://www.freertos.org/a00106.html. You will find information about how to create tasks, how to pause tasks, how to create coroutines, how to create semaphores, etc. Note that some features require you to include more files into your project. E.g. if you are using coroutines, you need to include coroutine.c in your project.