Lab 5

1. Using FreeRTOS to recreate the system you created in Lab #4. The system will have three tasks. Each task will turn on and off an LED when it is run. Use the same timing as in Lab #4. Make sure you use Task Delay to delay the tasks. The third LED should be connected to Port4, Pin2. You can use a single function and pass a void pointer to a delay variable to establish the delays for the two timed LED values. The idle process could just always run. You will want to consider the priority parameter. Should they all be the same priority? Should they have different priorities?

They should have the different priority because some tasks have precedence over others.

2. Submit your source code for the lab on I-learn.

/\* Standard includes. \*/

**#include** <stdio.h>

/\* Kernel includes. \*/

**#include** "FreeRTOS.h"

**#include** "task.h"

/\* TI includes. \*/

**#include** "gpio.h"

**static** **void** **vTaskFunction1**( **void** \*pvParameters );

**static** **void** **vTaskFunction2**( **void** \*pvParameters );

**static** **void** **prvSetupHardware**( **void** );

**int** **main**( **void** )

{

/\* See http://www.FreeRTOS.org/TI\_MSP432\_Free\_RTOS\_Demo.html for instructions. \*/

/\* Prepare the hardware to run this demo. \*/

prvSetupHardware();

**printf**("Running task1 function\n");

xTaskCreate( vTaskFunction1, "Task 1", 200, NULL, 1, NULL );

xTaskCreate( vTaskFunction2, "Task 2", 200, NULL, 2, NULL );

/\* Start the scheduler. \*/

vTaskStartScheduler();

/\* If all is well, the scheduler will now be running, and the following

line will never be reached. If the following line does execute, then

there was insufficient FreeRTOS heap memory available for the Idle and/or

timer tasks to be created. See the memory management section on the

FreeRTOS web site for more details on the FreeRTOS heap

http://www.freertos.org/a00111.html. \*/

**for**( ;; );

**return** 0;

}

/\*-----------------------------------------------------------\*/

**void** **vTaskFunction1**( **void** \*pvParameters )

{

/\* As per most tasks, this task is implemented in an infinite loop. \*/

**for**( ;; )

{

**GPIO\_toggleOutputOnPin**(

GPIO\_PORT\_P4,

GPIO\_PIN0

);

/\* Delay for a period. This time we use a call to vTaskDelay() which

puts the task into the Blocked state until the delay period has expired.

The delay period is specified in 'ticks'. \*/

vTaskDelay( 500 / portTICK\_PERIOD\_MS );

}

}

**void** **vTaskFunction2**( **void** \*pvParameters )

{

/\* As per most tasks, this task is implemented in an infinite loop. \*/

**for**( ;; )

{

**GPIO\_toggleOutputOnPin**(

GPIO\_PORT\_P4,

GPIO\_PIN1

);

/\* Delay for a period. This time we use a call to vTaskDelay() which

puts the task into the Blocked state until the delay period has expired.

The delay period is specified in 'ticks'. \*/

vTaskDelay( 250 / portTICK\_PERIOD\_MS );

}

}

**static** **void** **prvSetupHardware**( **void** )

{

**extern** **void** **FPU\_enableModule**( **void** );

/\* Stop the watchdog timer. \*/

MAP\_WDT\_A\_holdTimer();

/\* Ensure the FPU is enabled. \*/

FPU\_enableModule();

/\* Set Flash wait state for high clock frequency. Refer to datasheet for

more details. \*/

**FlashCtl\_setWaitState**( FLASH\_BANK0, 2 );

**FlashCtl\_setWaitState**( FLASH\_BANK1, 2 );

/\* The full demo configures the clocks for maximum frequency, wheras the

blinky demo uses a slower clock as it also uses low power features. Maximum

freqency also needs more voltage.

From the datashee: For AM\_LDO\_VCORE1 and AM\_DCDC\_VCORE1 modes, the maximum

CPU operating frequency is 48 MHz and maximum input clock frequency for

peripherals is 24 MHz. \*/

**PCM\_setCoreVoltageLevel**( PCM\_VCORE1 );

**CS\_setDCOCenteredFrequency**( CS\_DCO\_FREQUENCY\_48 );

**CS\_initClockSignal**( CS\_HSMCLK, CS\_DCOCLK\_SELECT, CS\_CLOCK\_DIVIDER\_1 );

**CS\_initClockSignal**( CS\_SMCLK, CS\_DCOCLK\_SELECT, CS\_CLOCK\_DIVIDER\_1 );

**CS\_initClockSignal**( CS\_MCLK, CS\_DCOCLK\_SELECT, CS\_CLOCK\_DIVIDER\_1 );

**CS\_initClockSignal**( CS\_ACLK, CS\_REFOCLK\_SELECT, CS\_CLOCK\_DIVIDER\_1 );

/\* Selecting P1.2 and P1.3 in UART mode and P1.0 as output (LED) \*/

MAP\_GPIO\_setAsPeripheralModuleFunctionInputPin( GPIO\_PORT\_P1, GPIO\_PIN2 | GPIO\_PIN3, GPIO\_PRIMARY\_MODULE\_FUNCTION );

MAP\_GPIO\_setOutputLowOnPin( GPIO\_PORT\_P1, GPIO\_PIN0 );

MAP\_GPIO\_setAsOutputPin( GPIO\_PORT\_P4, GPIO\_PIN0 );

MAP\_GPIO\_setAsOutputPin( GPIO\_PORT\_P4, GPIO\_PIN1 );

}

/\*-----------------------------------------------------------\*/

**void** **vApplicationMallocFailedHook**( **void** )

{

/\* vApplicationMallocFailedHook() will only be called if

configUSE\_MALLOC\_FAILED\_HOOK is set to 1 in FreeRTOSConfig.h. It is a hook

function that will get called if a call to pvPortMalloc() fails.

pvPortMalloc() is called internally by the kernel whenever a task, queue,

timer or semaphore is created. It is also called by various parts of the

demo application. If heap\_1.c or heap\_2.c are used, then the size of the

heap available to pvPortMalloc() is defined by configTOTAL\_HEAP\_SIZE in

FreeRTOSConfig.h, and the xPortGetFreeHeapSize() API function can be used

to query the size of free heap space that remains (although it does not

provide information on how the remaining heap might be fragmented). \*/

taskDISABLE\_INTERRUPTS();

**for**( ;; );

}

/\*-----------------------------------------------------------\*/

**void** **vApplicationIdleHook**( **void** )

{

/\* vApplicationIdleHook() will only be called if configUSE\_IDLE\_HOOK is set

to 1 in FreeRTOSConfig.h. It will be called on each iteration of the idle

task. It is essential that code added to this hook function never attempts

to block in any way (for example, call xQueueReceive() with a block time

specified, or call vTaskDelay()). If the application makes use of the

vTaskDelete() API function (as this demo application does) then it is also

important that vApplicationIdleHook() is permitted to return to its calling

function, because it is the responsibility of the idle task to clean up

memory allocated by the kernel to any task that has since been deleted. \*/

}

/\*-----------------------------------------------------------\*/

**void** **vApplicationStackOverflowHook**( TaskHandle\_t pxTask, **char** \*pcTaskName )

{

( **void** ) pcTaskName;

( **void** ) pxTask;

/\* Run time stack overflow checking is performed if

configCHECK\_FOR\_STACK\_OVERFLOW is defined to 1 or 2. This hook

function is called if a stack overflow is detected. \*/

taskDISABLE\_INTERRUPTS();

**for**( ;; );

}

/\*-----------------------------------------------------------\*/

**void** \***malloc**( size\_t xSize )

{

/\* There should not be a heap defined, so trap any attempts to call

malloc. \*/

**Interrupt\_disableMaster**();

**for**( ;; );

}

/\*-----------------------------------------------------------\*/

3. Hook up the Logic analyzer to the three pins. Capture a trace. Include it in the Lab report.

