Lab 2

1. This lab assumes you are at least familiar with the MSP432P401R and Code Composer. It will also assume that you have created a simple FreeRTOS project inside of the Code Composer studio, and have two tasks, one that simply blinks an LED at a certain rate, the other that changes the LED based on the pressing of a button.In this lab you’ll learn how to add a queue to communicate between the two processes.

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

/\* Standard includes. \*/

**#include** <stdio.h>

/\* Kernel includes. \*/

**#include** "FreeRTOS.h"

**#include** "task.h"

**#include** "semphr.h"

/\* TI includes. \*/

**#include** "gpio.h"

/\* Priorities at which the tasks are created. \*/

**#define** mainQUEUE\_RECEIVE\_TASK\_PRIORITY ( tskIDLE\_PRIORITY + 2 )

**#define** mainQUEUE\_SEND\_TASK\_PRIORITY ( tskIDLE\_PRIORITY + 1 )

/\* The rate at which data is sent to the queue. The 200ms value is converted

to ticks using the portTICK\_PERIOD\_MS constant. \*/

**#define** mainQUEUE\_SEND\_FREQUENCY\_MS ( pdMS\_TO\_TICKS( 1000UL ) )

/\* The number of items the queue can hold. This is 1 as the receive task

will remove items as they are added, meaning the send task should always find

the queue empty. \*/

**#define** mainQUEUE\_LENGTH ( 1 )

/\* Values passed to the two tasks just to check the task parameter

functionality. \*/

**#define** mainQUEUE\_SEND\_PARAMETER ( 0x1111UL )

**#define** mainQUEUE\_RECEIVE\_PARAMETER ( 0x22UL )

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

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

**void** **main\_blinky**( **void** );

**static** **void** **prvConfigureClocks**( **void** );

**static** **void** **prvConfigureButton**( **void** );

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

/\* The queue used by both tasks. \*/

**static** QueueHandle\_t xQueue = NULL;

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

**void** **main\_blinky**( **void** )

{

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

instructions and notes regarding the difference in power saving that can be

achieved between using the generic tickless RTOS implementation (as used by

the blinky demo) and a tickless RTOS implementation that is tailored

specifically to the MSP432. \*/

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

blinky demo uses a slower clock as it also uses low power features. \*/

prvConfigureClocks();

/\* Configure a button to generate interrupts (for test purposes). \*/

prvConfigureButton();

/\* Create the queue. \*/

xQueue = xQueueCreate( mainQUEUE\_LENGTH, **sizeof**( uint32\_t ) );

**if**( xQueue != NULL )

{

/\* Start the two tasks as described in the comments at the top of this

file. \*/

xTaskCreate( prvQueueReceiveTask, /\* The function that implements the task. \*/

"Rx", /\* The text name assigned to the task - for debug only as it is not used by the kernel. \*/

configMINIMAL\_STACK\_SIZE, /\* The size of the stack to allocate to the task. \*/

( **void** \* ) mainQUEUE\_RECEIVE\_PARAMETER, /\* The parameter passed to the task - just to check the functionality. \*/

mainQUEUE\_RECEIVE\_TASK\_PRIORITY, /\* The priority assigned to the task. \*/

NULL ); /\* The task handle is not required, so NULL is passed. \*/

xTaskCreate( prvQueueSendTask, "TX", configMINIMAL\_STACK\_SIZE, ( **void** \* ) mainQUEUE\_SEND\_PARAMETER, mainQUEUE\_SEND\_TASK\_PRIORITY, NULL );

/\* Start the tasks and timer running. \*/

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. \*/

**for**( ;; );}

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

**static** **void** **prvQueueSendTask**( **void** \*pvParameters )

{

TickType\_t xNextWakeTime;

**const** **unsigned** **long** ulValueToSend = 100UL;

/\* Check the task parameter is as expected. \*/

configASSERT( ( ( **unsigned** **long** ) pvParameters ) == mainQUEUE\_SEND\_PARAMETER );

/\* Initialise xNextWakeTime - this only needs to be done once. \*/

xNextWakeTime = xTaskGetTickCount();

**for**( ;; )

{

/\* Place this task in the blocked state until it is time to run again.

The block time is specified in ticks, the constant used converts ticks

to ms. While in the Blocked state this task will not consume any CPU

time. \*/

vTaskDelayUntil( &xNextWakeTime, mainQUEUE\_SEND\_FREQUENCY\_MS );

/\* Send to the queue - causing the queue receive task to unblock and

toggle the LED. 0 is used as the block time so the sending operation

will not block - it shouldn't need to block as the queue should always

be empty at this point in the code. \*/

xQueueSend( xQueue, &ulValueToSend, 0U );}}

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

**static** **void** **prvQueueReceiveTask**( **void** \*pvParameters )

{

**unsigned** **long** ulReceivedValue;

**static** **const** TickType\_t xShortBlock = pdMS\_TO\_TICKS( 50 );

/\* Check the task parameter is as expected. \*/

configASSERT( ( ( **unsigned** **long** ) pvParameters ) == mainQUEUE\_RECEIVE\_PARAMETER );

**for**( ;; )

{

/\* Wait until something arrives in the queue - this task will block

indefinitely provided INCLUDE\_vTaskSuspend is set to 1 in

FreeRTOSConfig.h. \*/

xQueueReceive( xQueue, &ulReceivedValue, portMAX\_DELAY );

/\* To get here something must have been received from the queue, but

is it the expected value? If it is, toggle the LED. \*/

**if**( ulReceivedValue == 100UL )

{

/\* Blip the LED for a short while so as not to use too much

power. \*/

configTOGGLE\_LED();

vTaskDelay( xShortBlock );

configTOGGLE\_LED();

ulReceivedValue = 0U;}}}

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

**static** **void** **prvConfigureClocks**( **void** )

{

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

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

From the datasheet: For AM\_LDO\_VCORE0 and AM\_DCDC\_VCORE0 modes, the maximum

CPU operating frequency is 24 MHz and maximum input clock frequency for

peripherals is 12 MHz. \*/

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

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

**CS\_setDCOCenteredFrequency**( CS\_DCO\_FREQUENCY\_3 );

**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 );

/\* The lower frequency allows the use of CVORE level 0. \*/

**PCM\_setCoreVoltageLevel**( PCM\_VCORE0 );

}

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

**static** **void** **prvConfigureButton**( **void** )

{

**volatile** uint8\_t ucPin;

/\* Configure button S1 to generate interrupts. This is done to test the

code path were low power mode is exited for a reason other than a tick

interrupt. \*/

**GPIO\_setAsInputPinWithPullUpResistor**( GPIO\_PORT\_P1, GPIO\_PIN1 );

**GPIO\_enableInterrupt**( GPIO\_PORT\_P1, GPIO\_PIN1 );

**Interrupt\_enableInterrupt**( INT\_PORT1 );

}

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

**void** **PORT1\_IRQHandler**( **void** )

{

**static** **volatile** uint32\_t ux = 0;

/\* This is the handler for interrupt generated by the button. The

interrupt is only used to bring the MCU out of low power mode. It

doesn't perform any other function. The ux increment is just to

have something to set breakpoints on and check the interrupt is

executing. \*/

ux++;

/\* Clear the interrupt. \*/

( **void** ) P1->IV;

}

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

**void** **vPreSleepProcessing**( uint32\_t ulExpectedIdleTime ) {}

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

**#if**( configCREATE\_SIMPLE\_TICKLESS\_DEMO == 1 )

**void** **vApplicationTickHook**( **void** )

{

/\* This function will be called by each tick interrupt if

configUSE\_TICK\_HOOK is set to 1 in FreeRTOSConfig.h. User code can be

added here, but the tick hook is called from an interrupt context, so

code must not attempt to block, and only the interrupt safe FreeRTOS API

functions can be used (those that end in FromISR()). \*/

/\* Only the full demo uses the tick hook so there is no code is

executed here. \*/}

**#endif**

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

