Computer Systems Engineering Technology
CST 347 – Real-Time Operating Systems

Lab 03 – Intertask Communication using Queue's

Name_____

Possible Points: 10

Instructions

Complete the following procedures. Zip up your final project and upload to Blackboard.

Procedure

In the last lab we explored more of the TASK API of FreeRTOS by creating, Destroying, Delaying, Suspending and Resuming tasks. We used the task handles set by the *xTaskCreate()* function to manipulate the tasks. We also used the *vTaskDelay()* function to sleep our tasks for a specified time.

In this lab, you will create three main control tasks. Only one of these tasks will be active at a time (the other two will be in the suspend state). This task will poll the buttons of the PIC32 Starter Kit and take the following actions. If button1 is pressed decrease the amount of delay for the currently selected led, button2 will increase the amount of delay for the current led and button3 will go to the next led. There will also be three independent LED tasks that will be blinking LED's of the PIC32 Starter kit. You will also create one additional task to send status messages through the UART. You will also create 4 Queues to talk between these tasks. Each Control Task will communicate to one LED task through one Queue. The last Queue will be for the UART communication.

Even if there are many ways to model this function, you are required to follow the program architecture specified here. You are only to use the FreeRTOS functions provided as part of this Lab 3 Description.

Application Function:

Queue Creation:

You will create 4 queues for this application. The queue's should have a depth of 5. For the LED queue's you should create a custom **`enum`** type **`{INCR, DECR}`** to serve as the passed message values. The UART queue should be set to a size of char [50]. You will figure out a method to get the Queue handles to the appropriate tasks.

MainControl Tasks:

The taskMainControl task will poll the status of all three switches (just as we did in lab 2). The switches should be debounced and perform a lockout function (to prevent a user from just holding

down the button). There will be three of these tasks (only 1 active at a time). Each of these task will send messages via a queue to its corresponding LED task. The switches will perform the following function.

SW1:

Switch 1 will decrease the delay time of the corresponding LED task. It will accomplish this by sending a message to this task via the queue setup for this pair. It will accomplish this by calling *vQueueSendToBack()* function. The message will tell the LED task to decrease its delay time.

SW2:

Switch 2 will increase the delay time of the corresponding LED task. It will accomplish this by sending a message to this task via the queue setup for this pair. It will accomplish this by calling *vQueueSendToBack()* function.

SW3:

Switch 3 will cause the next LED to be selected. It will accomplish this by resuming the next MainControl (*vTaskResume*) Task and then suspending (*vTaskSuspend*) itself. So if LED1 is the current MainControl Task then after the SUSPEND/RESUME LED2 will be the active MainControl Task. During this operation a message will also be sent to the UART task to identify the new active LED. This message will be in the form of a *char and contain a message stating "LED N IS NOW ACTIVE". Make sure that it includes the appropriate Line feeds to move cursor to the next line.

The polling of the switches will happen in sequence and actions will be taken accordingly. After all three switches have been polled, the task will perform a *vTaskDelay()* for 100ms. Error checking must be implemented to assure that the Message was sent to the appropriate queue with the *vQueueSendtoBack()*. Suggestion: create an enum for the message to say either increment or decrement the delay time.

LED Tasks:

The LED tasks will blink the corresponding LED on and off. It will do this at an initial rate of 500ms. It will also check its Queue to see if it has a message waiting (*uxQueueMessagesWaiting*). If there is a message waiting it should read the message (*xQueueReceive*) and perform the necessary action. The wait time to the *xQueueReceive()* should be set to 0. The action will be either to increase or decrease the delay time in increments of 50ms. This should be bounded to a MAX delay of 1000ms and a MIN delay of 200ms. There will be three of these tasks as well. The task should use your LED Driver from Lab 1.

UART Task:

The UART task will block until a message is ready for it. To accomplish this blocking set the last parameter of the *xQueueReceive()* to *portMAX_DELAY*. It will then get the message and write it out on the UART using your *UARTPutStr()*. You will need to write a very rudimentary UART driver for your program. The filenames for the driver will be uartdrv.c and uartdrv.h. This should include three

functions. The functions should be *initUART()*, *UARTPutC()* and *UARTPutStr()*. The *initUART()* function will initialize the UART. This is done by calling a couple of functions that get included with *#include <plib.h>*. The *UARTPutC()* will print a single byte (char) to the UART. The *UARTPutStr()* function will use the *UARTPutC()* to write a complete string to the UART.

initUART()

Prototype for function.

```
void initUART(UART_MODULE umPortNum, uint32_t ui32WantedBaud);
```

umPortNum in our case will be UART2. This is the UART for the Explorer 16 board. We will use a **ui32WantedBaud** baud rate of 9600 for this lab.

Calls to initialize serial port.

```
/* Set the Baud Rate of the UART */
UARTSetDataRate(umPortNum, (uint32_t)configPERIPHERAL_CLOCK_HZ,
ui32WantedBaud);


/* Enable the UART for Transmit Only*/
UARTEnable(umPortNum, UART_ENABLE_FLAGS(UART_PERIPHERAL |
UART_TX));
```

UARTPutC()

Prototype for function.

```
void vUartPutC(UART_MODULE umPortNum, char cByte);
```

Calls Needed

```
UARTTransmitterIsReady(umPortNum)
UARTSendDataByte(umPortNum, cByte);
```

`UARTTransmitterIsReady()` will return a 0 until the transmitter is ready to send a byte. So if you get a 0 back from this call delay for 2MS and try again. Once the function no longer returns a 0 then you can transmit the data to the UART by calling `UARTSendDataByte()`.

UARTPuStr()

Prototype for function

```
void vUartPutStr(UART_MODULE umPortNum, char *pString,
            int iStrLen);
```

Implement a put string function by calling your `vUartPutC()`.

FreeRTOS API:

Calls you are allowed to use on this lab:

- `xTaskCreate()`
- `vTaskSuspend()`
- `vTaskResume()`
- `vTaskDelay()`
- `xQueueCreate()`
- `xQueueSendToBack()`
- `uxQueueMessagesWaiting()`
- `xQueueReceive()`

It is suggested that you create a helperSystemStart helper file in myTasks.c that will create all of the tasks and queues. helperSystemStart will then be called from main(). This will save on trying to relate newly added code.

You will need to use RealTerm, puTTY, or HyperTerminal, or some terminal emulator to see the UART messages.