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

Lab 02 – Multiple Tasks & State Changes Name_____

Possible Points: 10

Instructions

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

Procedure

This lab will expose a few more RTOS kernel functions. Last time, we used the "task create" function to spawn a single task that sequenced the three PIC32 STARTER KIT LEDs in one of three patterns. A void * parameter reference was provided to the task at creation which held the pattern indication and the delay time in milliseconds.

In this lab, you will create multiple tasks, delete tasks, and suspend/resume tasks. 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 RTOS functions provided as part of this Lab 2 Description.

Application Function:

taskSystemControl:

This task will poll the status of the three switches and act accordingly. No parameters are required. Three switches will cause the following behavior:

SW1:

The first time **SW1** is pressed, **LED1** will start flashing. The second press will start **LED2**, and finally, the third press will start **LED3** flashing. Any subsequent presses will not do anything. This will be accomplished by doing a ***xTaskCreate*** for the next LED. Unlike last time, the task creation call must take back the task handle, which was the last parameter in the call. It was set **NULL** last time as the handle was not needed. However, in this function, the handle (one for each LED task created) will be necessary to delete the task. The task will toggle the appropriate LED and then do a ***taskDelay*** of **500 ms**. This will be done using an RTOS call versus the busy wait that was implemented in the last lab.

SW2:

Each time **SW2** is pressed, the highest LED flashing will stop. LEDs will stop until all of them have stopped. After all have LEDs have stopped flashing, subsequent presses will do nothing. This will be accomplished by deleting the highest flashing LED task. This will be accomplished by doing a ***vTaskDelete*** for the next LED. To delete a task, the task handle created during the task creation is used. After deleting a task **SW1** can restart it, so if all three LEDs were blinking then **SW1** did nothing. If **SW2** is pressed **LED3** would stop blinking, **SW1** would once again start this task blinking.

SW3:

A press of **SW3** will "freeze" all current flashing LEDs. If fewer than three LEDs are currently flashing, **SW1** will have the opportunity to cause any higher LEDs to start flashing as described above. A second press of **SW3** will "unfreeze" and frozen LEDs. This is accomplished by suspending all currently running LED tasks on the "first" button press. The second button press will resume all created LED tasks. ***vTaskSuspend()*** and ***vTaskResume()*** are used to suspend a resume tasks.

Within a single while(1) loop, 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**. Use the "raw" switches in your task. Do this with the **mPORTDReadBits** macro (ports.h).

Hardware Considerations:

The "logical names" of the LEDs and switches referenced above have different names on the silkscreen print on the Explorer 16 Board. The following mapping is assumed:

Logical Name	Explorer 16 Silkscreen Names	Explorer 16 PIC32 Pins
LED1	D3	RA0
LED2	D4	RA1
LED3	D5	RA2
SW1	S3	RD6
SW2	S6	RD7
SW3	S4	RD13

The input will come via the three switches **S3 (RD6)**, **S6 (RD7)** and **S4 (RD13)** as shown in the Figure 1 below. The switches on have pull-up resistors connected to them.

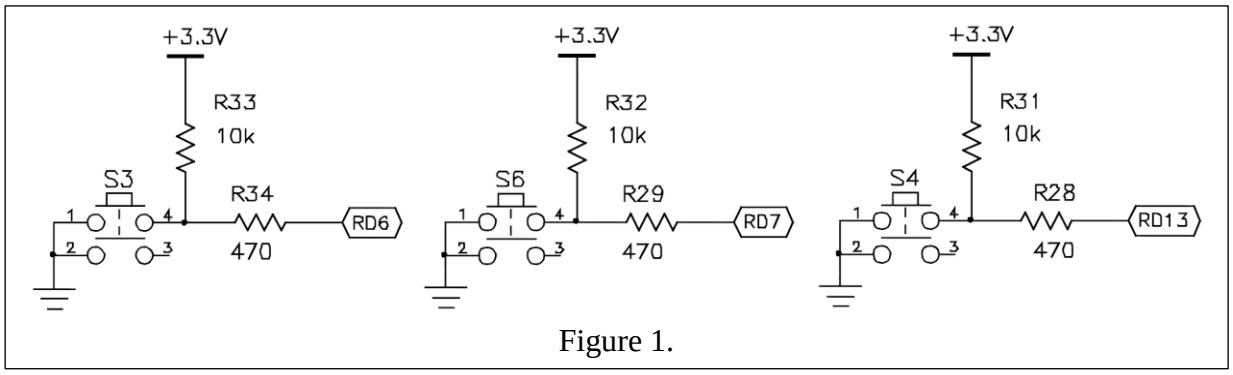

Figure 1.

Even if pull-ups are connected, however, the switches still need to be debounced. The debounce is implemented with the three stage strategy:

- Read the switch
- Delay 10 ms
- Read again to verify change

Final Considerations:

The FreeRTOS API calls you will use are:

xTaskCreate()
vTaskDelete() {Must have `#define INCLUDE_vTaskDelete 1` in FreeRTOSConfig.h}
vTaskSuspend()) {Must have `#define INCLUDE_vTaskSuspend 1` in FreeRTOSConfig.h}
vTaskResume()
vTaskDelay()

The API documentation can be found on the web at the freeRTOS.org site at the following location:

http://www.freertos.org/a00106.html

You are writing two tasks, ***taskSystemControl***, and ***taskToggleAnLED*** from the demo lab (and Lab 1).

During operation, a maximum of four tasks could be running: Your main control task and three LED tasks. You are required to use your LED driver from last week to control the LEDs.

If there are any questions please ask....

Appendix:

PIC32 Starter Kit Considerations:

The input will come via the three STARTER KIT switches **SW1 (RD6)**, **SW2 (RD7)** and **SW3 (RD13)** as shown in the Figure to the right. The switches are raw and simply ground the respective **Port D** pin when pressed. They do not have external pullups and they not debounced. Both of these issues must be addressed by the software.

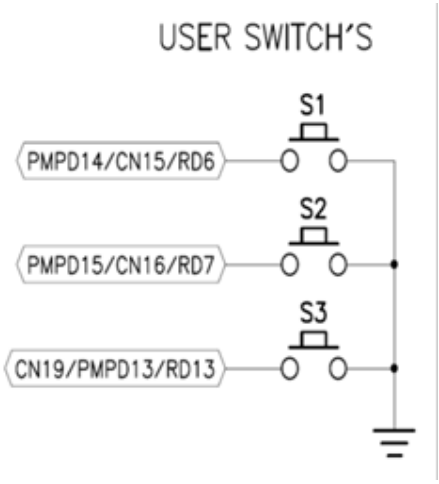

Figure A1.

The first issue of pullups is handled by enabling the PIC32-provided internal pullups that are part of the **Change Notification (CN)** peripheral. Only the pin references that specify a **CNx** notion have this capability, such as those pins used for the STARTER KIT switches as illustrated in the Figure. You will notice that **RD6** is associated with **CN15**, **RD7** with **CN16**, and **RD13** with **CN19**. These **CN** pins will need to have their pullups enabled. This PIC32 C environment provides a macro that performs the enable function in a similar way to interacting with the Port bits themselves. The following macro statement should be written into the ***prvSetupHardware()*** function.

```
ConfigCNPullups(CN15_PULLUP_ENABLE | CN16_PULLUP_ENABLE |
CN19_PULLUP_ENABLE);
```

The second issue of debounce is handled in the same manner as the Explorer 16 Board.