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

Lab 05 – Interrupts in FreeRTOS v2 Name_____

Possible Points: 15

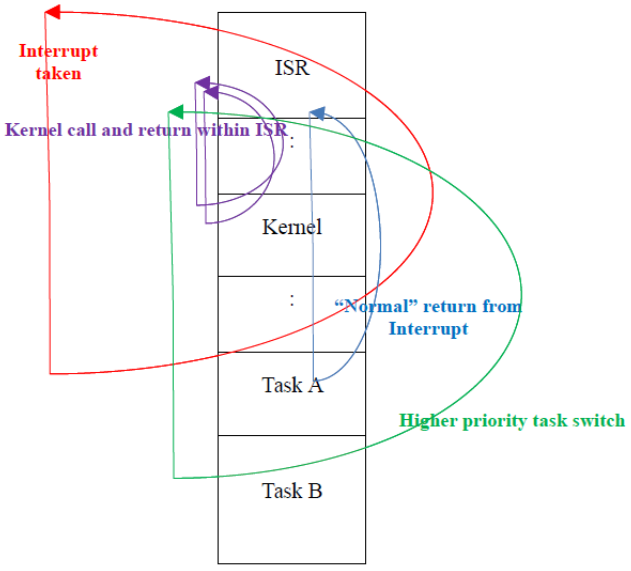
Instructions

Complete the following procedures. Answer any questions given on this lab. Zip up your final project and answers to the questions, then upload to blackboard.

Procedure

Part 1

- Lab 4 focused on the scheduling issues of preemptive versus cooperative scheduling and prioritization. Tasks were run according to several scenarios and behavior varied based on the specific conditions of the scenario.



- This lab will bring in functionality that is essential to an embedded real time system and that is the interrupt service routine (ISR). ISRs sit outside of the multitasking structure. Since a timer interrupt is necessary for the kernel's tick timing, the interrupt system is required even without application specific interrupts. However, when application specific interrupts are brought in to have event driven processing, the kernel must be allowed to continue its chore of scheduling once the ISR is complete.

- Complexity arises when an ISR intends to cause a task (through special kernel calls) to change state. Remember, if an ISR is executing, neither the kernel nor any tasks are executing. But, the kernel must have some way to change execution flow upon return from an ISR. For example, the task that was interrupted could be of lower priority than a task that changes state as a result of a kernel call within the ISR. In this case, the kernel should have the opportunity to cause a context switch and return back to the higher priority task in the Running state as well as place the interrupted lower priority task in the Ready state. This scenario changes the way "normal" execution flow proceeds. That is, the kernel must have the chance to change the return address that was saved when the ISR commenced, as well as change the task states.

Application Function

The function for this lab is to receive characters from a terminal emulator keyboard through the UART and perform some sort of control action. In previous labs, the UART was used to display a message to the terminal emulator. Only the transmit (TX) function of the UART was required. In this lab, both the TX and the receive (RX) functions are required. In addition, the UART transmitter and receiver will be interrupt-driven rather than polled, as was done previously. The received characters will be interpreted as very simple command codes. All non-command code characters will be ignored. Lastly, characters received through the RX need to be echoed back through the TX for display on the terminal.

Tasks

1. LED Task – The LED task will simply block on a message arrival in the LEDQueue. The message will be to toggle one of the LEDs LED1-LED3. Therefore, the message must include an LED number which will be "1", "2", or "3". The LED task will be created with task priority 1.
2. TX Task – The UART TX task will block awaiting a TXQueue message. The message will be a string of maximum 50 characters for display to the terminal. The TX task will call the PutStr UART driver function which will start and interrupt-driven string transfer to the TX. The TX task will be created with task priority 2.
3. RX Task – The UART RX task will be suspended most of the time awaiting resumption by the UART ISR. When resumed, it will call the GetChar UART driver function. It will use the character data acquired through the RX ISR. In response, it will send the character data to the TX queue for echoing purposes. In addition, it will decode the character value to determine if it is a simple command code. If so, it will message the LED task with the LED number. Assume that "1" = LED1, "2" = LED2, and "3" = LED3. After this character processing, it will then suspend itself, awaiting the next character. The RX task will be created with task priority 3.

Queues

1. TX Queue – This queue will be created with a message size of 50 bytes and a depth of 20 messages.
2. LED Queue – This queue will be created with a message size of 1 byte and a depth of 5 messages.

UART Interrupt Driven Driver

Both the RX and the TX use the same ISR. Operations will be within the single ISR functions.

1. UARTPutStr – This function will take the TX task data and format it for the TX function of the ISR. The data will be a null-terminated string, including "\r\n", that the ISR can read. After formatting the data, it will enable the UART TX interrupt.
2. UARTGetChar – This function will return the character buffer that the RX ISR populated.

UART Interrupt Service Routine

1. RX function – This operation will populated the character buffer to be made available to the UART driver. It will use UARTGetData() from the plib to extract the RX data value. After getting the value, clear the RX interrupt flag and then resume the RX task using xTaskResumeFromISR().

2. TX function – This operation will use UARTSendDataByte() to send characters to the terminal. It will clear the TX interrupt flag AND disable the TX interrupt when character is NULL. (End of String)

Part 2

In previous labs, the buttons SW1-SW3 were strictly polled. In this lab, the buttons will be interrupt-detected. An ISR will be integrated to "catch" the button PRESS or RELEASE actions. The same ISR will be vectored to for either action. No data processing is performed in the ISR. However, the ISR must manage the pin Change Notification Process that a single button press with switch bounce does not cause extra interrupts. This will be handled through a disable interrupt action. The application piece of this lab will toggle an LED when at button is pressed AND released.

Semaphores

buttonPress – This will be created as a mutex and initialized to 0 [a pre-xSemaphoreTake() needs to be executed after it is created]. It will be "given" by the change notification ISR. It will be "taken" by the Button task.

ledNAction – There will be three semaphores required; one for each LED Task. Each will be created as a mutex and initialized to 0 [a pre-xSemaphoreTake() needs to be executed after it is created]. It will be "taken" by the LEDn Task and "given" by the Button Task.

Tasks

1. Button Task – Although somewhat brief, the purpose of this lab is to integrate in the interrupt-driven button function [versus polling as in past labs]. As a result, application operation of this task is trivial. This task must keep track of a Global Button State for the three buttons. Within the task's while(1) loop, the following needs to occur:

    a. "Take" the buttonPress mutex.

    b. vTaskDelay() for 10 ms to serve as a debounce delay.

    c. Read Port D pins. In addition to capturing the pin values, it clears out any Change Differences that currently exist in the module.

    d. Compare the Port D pin values with the global State to determine which buttons were pressed OR released.

    e. When change is detected, a button state change from 0-to-1 (button RELEASE) will "give" the appropriate ledNAction mutex. Just to be clear, this action is performed ONLY on the release, i.e. not on PRESS. If a 1-to-0 change is detected, no "give" action is performed.

    f. After the "give" action/s is/are taken, if at all, the Global Button State is updated.

    g. Finally, the Change Notification Interrupt is enabled.

2. ledN Task – There will be three Tasks associated with the three LEDs. Within the task's while(1) loop, the following two actions occur:

    a. "Take" the appropriate ledNAction mutex.

b. Toggle the appropriate LED.

Button Driver

The application actions related to a button press will be handled through Tasks. [For this lab, the actions are trivial.] As such, only two functions are required in the driver. A mostly complete driver file "buttondrv.c/h" are provide for your convenience.

1. initCN() – This function will initialize the Change Notification Module. It will modify and replace the original pullup enable actions from previous labs.

2. vCN_ISR() – This is the ISR. You will need to implement the mutex operations.

Change Notification Interrupt Service Routine Interrupt Service Routine

This ISR is very brief. All of the data-driven actions will be handled by Tasks. The three actions required here are:

3. Disable the Change Notification Interrupt

4. Clear the Change Notification Interrupt Flag

5. "Give" the buttonPress semaphore.

Additional Notes

You will need to include the semphr.h file.
You will need to make sure of the following in the FreeRTOSConfig.h file:

```
#define configUSE_PREEMPTION       1
#define configUSE_MUTEXES          1
#define INCLUDE_vTaskSuspend       1
```