Lab #5 Introduction To Interrupts

1. Description

In this laboratory, we learn about implementing an interrupt that creates a new thread that runs while pausing the main function upon detecting a specific signal. In addition, we learn about implementing combination of TTL NAND gates to debounce the signal that is triggered by a switch. The hierarchical structure is described in below.

```
main()
|_ myRio_Open()
|_ Irq_RegisterDiIrq()
|_ pthread_create()
    |_DI_Irq_Thread()
        |_ Irq_Wait()
        |_ printf_lcd()
        |_ Irq_Acknowledge()
        |_ pthread_exit()
|_ wait()
|_ printf_lcd()
|_ pthread_join()
|_ Irq_UnregisterDiIrq()
```

The main program which runs in the background registers the DI0 IRQ (Interrupt Request) which sets the special interrupt signal to be at DI0. Then, it creates new interrupt thread, and configures it to service the DI interrupt. Next, we implement a code that can detect a falling-edge transition on DIO0. After these tasks are completed, it should signal the new thread to terminate, then waits for the thread to actually terminate. Finally, the interrupt thread is unregistered.

The interrupt thread is started by calling *DI_Irq_Thread(void* resource). It's task is to perform activities in response to the interrupt. The first step is to cast the input into a ThreadResource* type. Then, it enters a loop which waits for the timeout of the IRQ, and prints "interrupt_" and acknowledges the interrupt if the numbered IRQ has been asserted. Lastly, it terminates the new thread and exits the function.

We control the DI0 by pressing a switch. The issue with switches is that there is a bouncing profile that causes DI0 to bounce between high an low in one press. We can implement a combination of TTL NAND gates to debounce the circuit. This allows only one interrupt to be detected for every button press.

2. Testing

There are two components of this lab to be tested: the main.c code and the debouncing circuit. To test the main.c code:

1. Connect the debouncing circuit to the switch as shown by the diagram below:

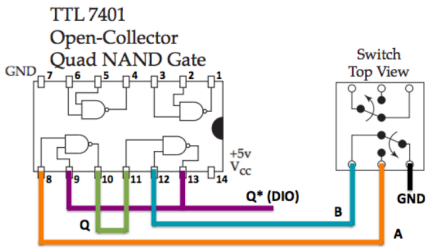

Figure 1: Debouncer Circuit

2. Run main.c code in Eclipse
3. Once the program is executed, there LCD screen should print "Counter: " + values from 0 to 60, which are incremented every 1 s.
4. Press the switch, the LCD screen should print "interrupt_" for a brief moment, then the LCD screen switches back to "Counter: " + the previous value incremented by one. Upon terminating the interrupt thread, the background process should pick up where it started. One line of code I fixed from the code provided in the *DI_Irq_Thread(void* resource) routine is to add " in the print statement to only print "interrupt_" once to the LCD in one interrupt.
5. You should be able to keep on pressing the button to call out the interrupt thread.
6. The program should stop running once it reaches "Counter: 60".

The hardware part of the debouncer circuit can be tested by observing the oscilloscope.

1. Connect the SPDT switch as described in the diagram below.
2. Configure the oscilloscope to show Ch0, and press the "Pattern" button.
3. Press the SPDT switch

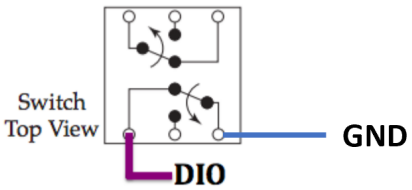

Figure 2: SPDT Circuit

4. Adjust the time delay and the time increment as needed to show a bouncing pattern.
5. Connect the SPDT switch to the debouncing circuit as shown in Figure 1.
6. Press the switch once more and observe that there is no bouncing. If the oscilloscope still shows a bouncing pattern, ensure that the wiring is correct.

The test is complete!

3. Results

The testing sequence performed as expected. When the main.c is run, there is a counter that increments the value every one second which is printed in the screen. Each time a button is pressed, the LCD prints "interrupt_" and it comes back to the main() function printing the counter which is incremented from its previous value before the SPDT switch is pressed. In addition, only pressing the button activates the interrupt thread, unlike releasing the button. Therefore, our main() function is validated and the debouncing circuit works to eliminate bouncing DI0 reading. Secondly, analyzing the oscilloscope tells us the before and after condition of the signal upon a button press. When the debouncing circuit is not implemented, we can see a bouncing profile in the oscilloscope shown in Figure 3.

When the debouncing circuit consisting of the TTL Quad NAND Gate is implemented, there is no more bouncing, instead, there is only one falling edge in every button press. This is shown in Figure 4.

THE TTL Quad Nand Gate operates by confirming to the logic that governs them. Firstly, if A is 0, its output: Q must be 1. Conversely, if B is 0, its output: Q* must be 1. In addition, when both A and B are 1, there are two possibilities for both Q and Q*, that is, they must be either 1 and 0 or 0 and 1 respectively. However, we can test individual cases. If when A and B are 1 and Q and Q* are 0 and 1, this does not hold according the truth table, sinche Q has to be 1 when A is one. Therefore, Q and Q* must be 1 and 0 respectively for the circuit to work. The resulting time history of signals A and B are shown in Figure 5.

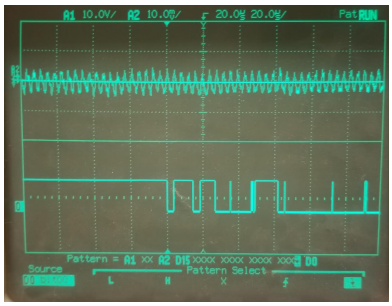

Figure 3: Bouncing DI0

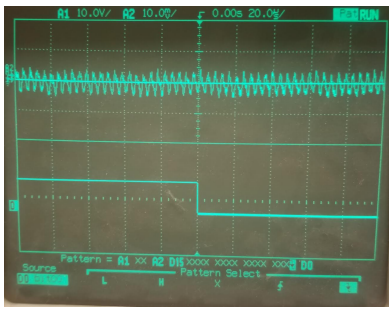

Figure 4: Bouncing DI0

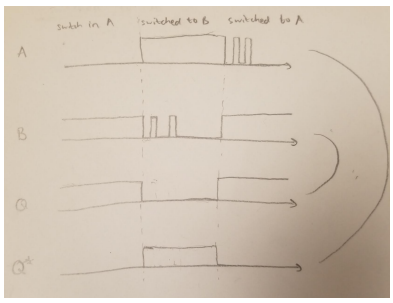

Figure 5: Time-history of Signals A and B