RILEY POOLE
ME 477 – LAB REPORT 05
WINTER 2019

I. DESCRIPTION

This experiment is to illustrate the digital (external) interrupt capabilities of the myRIO, and to experiment hardware debounce circuits. It accomplishes this by using one thread for a count displayed on the screen, and another to show a message only during an interrupt. The interrupt will be detected with the digital I/O pin 0 falling edge voltage. The fact that the count continues unchanged during the message from the interrupt thread shows that the interrupt is working.

THE MAJOR TASKS PERFORMED BY THIS PROGRAM WERE

- illustrate the use of external interrupts to the myRIO
- illustrate the use of multithreading on the myRIO

THE LIMITATIONS OF THIS PROGRAMS CAPABILITY ARE

- It still uses the inferior wait() function
- Program does not provide a way to loop after reaching 60
- No debounce prevention is implemented in the original hardware or code
- In the original code the interrupt only runs until the main function calls for printf(). This leads to “interrupt_” being displayed and almost immediately being erased.

EXPLAIN ANY ALGORITHMS

The interrupt thread and the LCD counting thread are different to prevent the count from halting while the “interrupt_” message is displayed.

The interrupt is registered to the DI0 pin and uses a rising edge as the detection. As an extension to the program I have created a #ifdef structure that will set the program mode to a rising or falling edge and allow the interrupt prompt to be displayed for as long as the button is pressed.

Data between the threads is passed through a thread resource structure.

HIERARCHICAL STRUCTURE

- main
  - MyRio_Open
  - MyRio_IsNotSuccess
  - Irq_RegisterDiIrq
  - pthread_create
  - printf_lcd
  - pthread_join
  - Irq_UnregisterDIIrq
  - MyRio_Close

- DI_Irq_Thread
  - pthread_exit

- wait
  - none.

II. TESTING

I have designed this program to hold the interrupt message until the button is released. This means the tester should set the #define with the commenting at the top of main to the correct starting position of the input.

#define input_starts_high or #define input_starts_low

1. **Test that the interrupt routine does not affect the count on the display**
    a. set the #define setting; compile and run the program
    b. make a mental note of the count number before pressing the interrupt button
    c. press and hold the button
        i. The LCD should display “interrupt_” until the button is released.
    d. Release the button and check that the timer has advanced past your mental note

    A stopwatch could also be used instead of a mental note to prove that the timer rate is also unaffected.

III. RESULTS

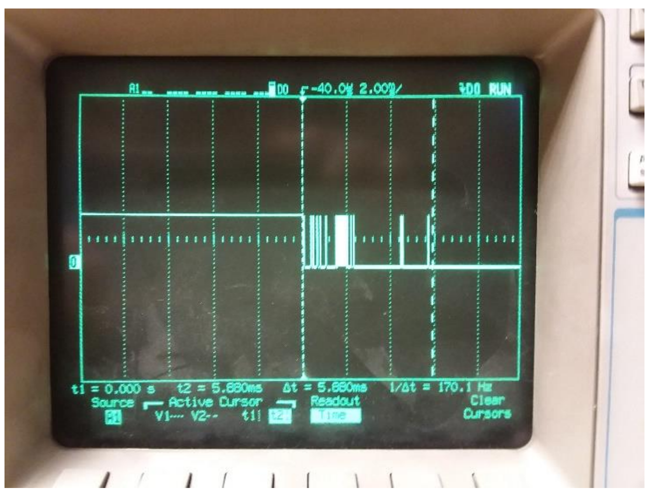
HOW SUCCESSFULLY THE PROGRAM RUNS AND UNSOLVED PROBLEMS

There are no unsolved problems with this program it functions as it is intended. It even includes an extension to keep the interrupt message displayed for the duration of the button press. Perhaps an unsolved problem could be in the intended design. This program is not designed to loop back to zero after the 60 seconds have been counted which is unnatural. However, it shows that the program will close the myRio successfully.

SPECIFIC QUESTIONS IN THE ASSIGNMENT

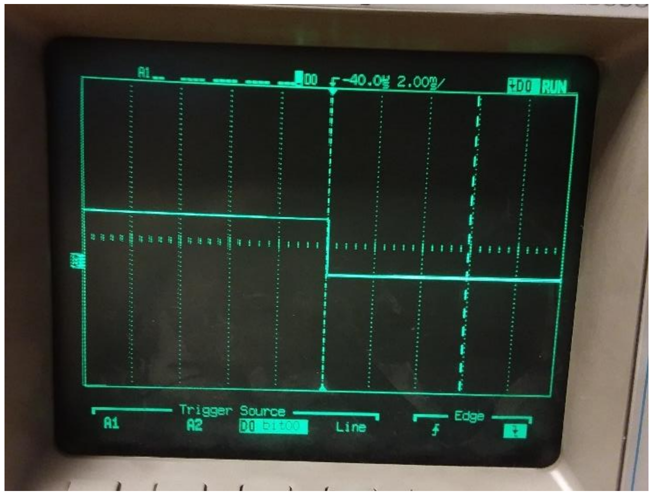
1. Try your program. What happens? Adjust the oscilloscope to examine the high-to-low transition of the IRQ signal. Typically, what length of time is required for the transition to settle at the low level? How many TTL triggers occur during the settling?.

Trying the program initially “interrupt_” is displayed quickly and flashes on the screen rapidly. This happens because of the many triggers that occur before the switch settles. Typically, it takes roughly 8.9 ms to transition to the low level and there are at least 9 triggers that occur in that time see the figure below.



Correct the problem by replacing the switch in Figure 1 with the “debouncing" circuit

With the debounce circuit there is very little bounce and the transition occurs cleanly.

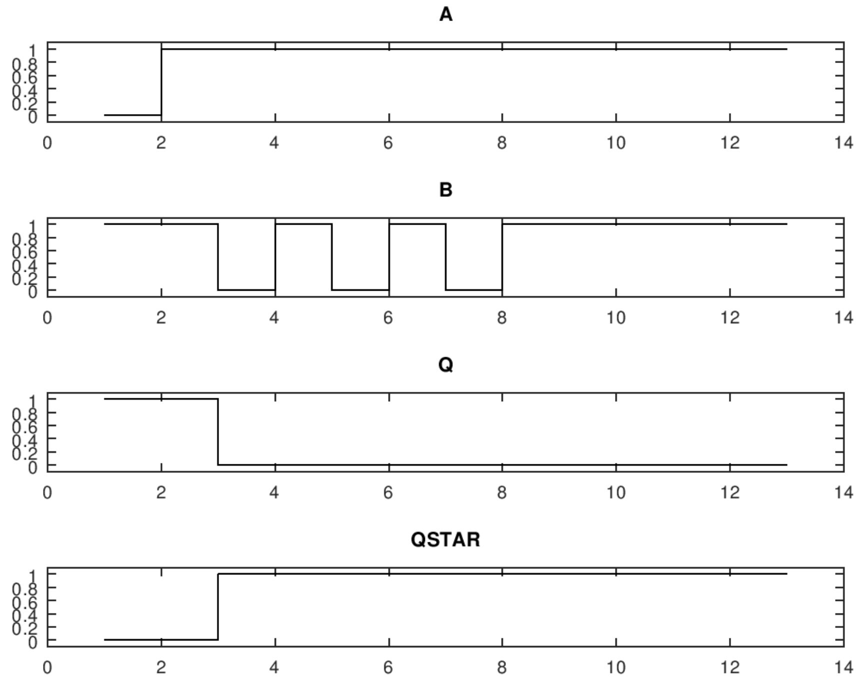


2. Explain in detail why this circuit should solve the switch bounce problem.

This circuit works because of the NAND gate truth table (1110). It is always on unless both inputs are true. For this circuit the output of each gate is connected to the input of the next. The other input of each gate is connected to a pull-up resistor. When the switch touches a terminal it forces a 0 to one of the inputs of a NAND gate. The makes its output 1. This output is fed back to the other gate to make one of the inputs a 1 and since it is connected to a pull-up resistor, and the switch is in the other position, both inputs must now be 1 which makes the output a 0. When the switch is between positions (i.e. bouncing) the previous output state is retained because of the feedback loop just described. Two NAND gates connected this way is essentially a 1-bit register. Because the values are retained while the switch is in between states this eliminates the bouncing problem.

Graph the time-history of signals at points A and B that would occur during the operation of a bouncing switch

Graph of switching from A to B with bouncing



3. In your own words, explain how the main program thread configures the interrupt thread

Configuring the interrupt is accomplished in two parts. (1) Creating a new thread and (2) Registering the interrupt

pthread_create() makes the new thread. Essentially the compiler tells the processor to run a different section of code (DI_Irq_Thread function) concurrently with the rest of main(). This other section of code is identified by its thread handle (thread) and is passed arguments through the ThreadResource structure (irqThread0). The thread handle allows telling the CPU which section of code to run. The IrqNumber allows the thread to only run after the correct interrupt has been asserted.

Irq_RegisterDiIrq() registers the digital interrupt. The compiler tells the CPU to watch falling edge, count, and channel registers and to assert an IrqNumber when an edge is detected. This assertion of the IrqNumber will signal the correct thread to run. I believe this has to do with the vector interrupt table at the lowest level addresses of the MyRio. To be honest I am not entirely sure about the table and operating system’s role in this process. I would assume that scheduler and the vector table are working together somehow.

How it communicates with the interrupt thread

Main() communicates with the interrupt thread through the global variables and the ThreadResource structure. They also share the memory space and registers on the MyRio. Main() is able to stop the other thread with the ThreadRdy flag and is waiting for the acknowledgement that the interrupt has been fulfilled. It also can tell when the thread has finished operation with pthread_join() to wait for it to finish before removing the thread handle. It is also responsible for unregistering the interrupt with the channel and the context structure which includes the irqNumber.

how it communicates with the interrupt thread during execution

Communication between threads is accomplished with a globally defined thread resource structure. The threads run concurrently and this structure helps define what memory space is available to both. Both threads can also access the global variables in the source file so caution is needed especially with multiple sources. Main can communicate by changing the values of the global variables and the ThreadResource structure. I’m not sure but I think we are using system calls to set up these threads which would mean communication between them is ultimately determined by the operating system.

how the interrupt thread functions

The interrupt thread function must first import the ThreadResource structure and make it the correct type. Then it checks to make sure main still allows the function to run with the irqThreadRdy flag. It then waits for the interrupt to occur and checks that it is the right one with IrqWait() and irqNumber. Lastly, it acknowledges the interrupt has occurred. When main tells the interrupt thread to end it exits the thread and allows the entire program (main and interrupt) to end. It also returns NULL in accordance with its prototype.

POSSIBLE IMPROVEMENTS AND EXTENSIONS

- It would be nice for the counter to loop continuously or to chime when it is finished.
- User should be able to specify the duration or the number of counts
- Software debouncing could be implemented along with the hardware debounce for even better quality
- The type of interrupt display could be set by the user (pause and display vs. wait for key retraction)
- Newer and faster NAND gate chip which a higher bandwidth in case we want to measure extremely rapid key presses (MHz)

FUNCTIONS OF THE PROGRAM

- **main()**
  - Prototype:
    - int main(void)
  - Purpose:
    - test the interrupt thread by allowing for inputs and outputs while printing to LCD
  - Rationale for creation:
    - Need a procedural section to layout testing environment
  - Inputs and parameters:
    - No inputs or parameters used
  - Return:
    - LCD displays the count every second for a minute
    - LCD displays “interrupt_” when it senses a falling edge on DIO0
- **DI_Irq_Thread()**
  - Prototype:
    - void* DI_Irq_Thread(void* resource)
  - Purpose:
    - Provide another thread and routine to catch the interrupt
  - Input and Parameters:
    - void* thread resource structure (to be recast as type ThreadResource after being passed)
  - Return:
    - void* -no return
- **wait()**
  - prototype:
    - void wait(void)
  - purpose:
    - waits for a period
  - parameters
    - none
  - returns
    - none

ALGORITHMS AND PSEUDOCODE

```
main()
      open the myRIO
      Register the interrupt
      Create an interrupt thread
      while(count<60)
            for 200 times //wait 1 sec
                  call wait() for 5 ms
                  clear the display
                  print the value of count
                  increment the value count
      signal the interrupt thread to stop
      wait until the thread stops
      unregister the interrupt
      close the myRIO

DI_Irq_Thread() //interrupt service routine
      wait for external interrupt to occur on DIO0
      service the interrupt by printing "interrupt_"
      acknowledge the interrupt

wait()
      loop 417000 times
```