ME 477 Embedded Computing
Laboratory Experiment #5
Introduction To Interrupts

Objectives

The objectives of this exercise are to:

1. Introduce the use of interrupts in I/O programming,
2. Introduce the use of multiple threads,
3. Become familiar with digital signal conditioning for interrupts, and
4. Use TTL gates to "Debounce" a switched input.

Introduction

This exercise illustrates the use of interrupts, originating from sources that are external to microcomputer. The principal activity of your main program is to print the value of a counter on the display. If uninterrupted, the counter display, which is updated once per second, would continue for 60 counts.

Generally, the "service" of an interrupt, may be arbitrarily complex in both form and function. However, in this exercise, each time an interrupt request (IRQ) occurs the interrupt service thread will simply print out the message, "`interrupt_`". A push-button switch on an external circuit will cause the IRQ to occur.

Therefore, the overall effect will be that the display will print the count repeatedly, with the word "`interrupt_`" interspersed randomly (for each push of the switch).

Although this program is not long, it is essential that you understand the events that take place at the time of the interrupt: (1) an unscheduled (asynchronous) external event has caused the activity of the CPU to be suspended, and (2) a separate section of code (the ISR) has been executed, before returning control to the original program at precisely the point where the execution was interrupted. That the counter display continues to run accurately both before and after the interrupt illustrates that the main program is not altered, regardless of where the interrupt occurs in the execution.

The Threads

The Main Program—The main program runs in the main thread. It will perform the following tasks:

1. Open the myRIO.
2. **Register the interrupt** and the digital input.
3. **Create an interrupt thread** to "catch" the interrupt.
4. Begin a loop. Each time through the loop:
   - Wait one second, by calling the "5 millisecond wait" function (from Lab #4) 200 times.
   - Clear the display, and print the value of count.
   - Increment the value of count .
5. After a count of 60. Signal the interrupt thread to stop, and wait until it terminates.
6. Unregister the interrupt.
7. Close the myRIO.

The Interrupt Service Routine—The interrupt service routine runs in a interrupt thread, separate from the main thread. It should begin a while loop that terminates only when signaled by the main thread.
Within the loop it will:

1. Wait for an external interrupt to occur on `DIO0`.
2. Service the interrupt by printing the message:, "`interrupt_`" on the LCD display.
3. Acknowledge the interrupt.

Main Thread: Background

Within `main.c` we will configure the DI interrupt, and create a new thread to respond when the interrupt occurs. The two threads communicate through a *globally defined* thread resource structure:

```
typedef struct  {
 NiFpga_IrqContext irqContext; // IRQ context reserved
 NiFpga_Bool irqThreadRdy;     // IRQ thread ready flag
 uint8_t irqNumber;            // IRQ number value
} ThreadResource;
```

National Instruments provides two C functions to set up the Digital Input (DI) interrupt request (IRQ).

I. Register the DI0 IRQ – The first of these functions reserves the interrupt from FPGA and configures the DI and IRQ. Its prototype is:

```
int32_t
Irq_RegisterDiIrq( MyRio_IrqDi* irqChannel,
                   NiFpga_IrqContext* irqContext,
                   uint8_t irqNumber,
                   uint32_t count,
                   Irq_Dio_Type type)
```

where the five input arguments are:

1. `irqChannel`- A pointer to a structure containing the registers and settings for the IRQ I/O to modify; defined in `DIIRQ.h` as:

   ```
   typedef struct{
    uint32_t dioCount;          // count register
    uint32_t dioIrqNumber;      // number register
    uint32_t dioIrqEnable;      // enable register
    uint32_t dioIrqRisingEdge;  // rising edge-trig reg.
    uint32_t dioIrqFallingEdge; // falling edge-trig reg.
    Irq_Channel dioChannel;     // supported I/O
   } MyRio_IrqDi;
   ```

2. `irqContext` - a pointer to a context variable identifying the interrupt to be reserved. It is the first component of the thread resources structure.
3. `irqNumber` - The IRQ number (1-8).
4. `count` - The number times that interrupt condition is met to trigger the interrupt.
5. `type` - The trigger type that you use to increment the count.

the returned value is 0 for success.

II. Create the interrupt thread – The second function, `pthread_create()` called from `main.c`, creates a new thread and configures it to "service" the DI interrupt. Its prototype is:

```
int pthread_create( pthread_t *thread,
                    const pthread_attr_t *attr,
                    void *(*start_routine) (void *),
                    void *arg);
```

where the four input arguments are:

1. `thread` - A pointer to a thread identifier.
2. `attr` - A pointer to thread attributes. In our case, use NULL to apply the default attributes.
3. `start_routine` - Name of starting function in the new thread.
4. `arg` - The sole argument to be passed to the new thread. In our case, it will be a *pointer* to the thread resource structure defined above in the second argument of `Irq_RegisterDiIrq()`.

This function also returns 0 for success.

Main Thread: Our Case

We can combine these ideas into *a portion* of the `main()` code needed to initialize the DI IRQ.[1] For interrupts on falling-edge transitions on `DIO0` of connector-A, assigned to IRQ 3 we have:

```
int32_t status;
ThreadResource irqThread0;
pthread_t thread;
MyRio_IrqDi irqDI0;

// Configure the DI IRQ
const uint8_t IrqNumber = 2;
const uint32_t Count = 1;
const Irq_Dio_Type TriggerType = Irq_Dio_FallingEdge;

// Specify  IRQ channel settings
irqDI0.dioCount = IRQDIO_A_0CNT;
irqDI0.dioIrqNumber = IRQDIO_A_0NO;
irqDI0.dioIrqEnable = IRQDIO_A_70ENA;
irqDI0.dioIrqRisingEdge = IRQDIO_A_70RISE;
irqDI0.dioIrqFallingEdge = IRQDIO_A_70FALL;
irqDI0.dioChannel = Irq_Dio_A0;

// Initiate the IRQ number resource for new thread.
irqThread0.irqNumber = IrqNumber;

// Register DI0 IRQ.  Terminate if not successful
status=Irq_RegisterDiIrq(
     &irqDI0,
     &(irqThread0.irqContext),
     IrqNumber,
     Count,
     TriggerType);

// Set the indicator to allow the new thread.
irqThread0.irqThreadRdy = NiFpga_True;

// Create new thread to catch the IRQ.
status = pthread_create(
    &thread,
    NULL,
    DI_Irq_Thread,
    &irqThread0);
```

Other `main()` tasks go here.

After the tasks of `main.c` are completed, it should signal the new thread to terminate by setting the `irqThreadRdy` flag in the `ThreadResource` structure. Then, wait for the thread to terminate. For example,

```
irqThread0.irqThreadRdy = NiFpga_False;
pthread_join(thread,NULL);
```

Finally, the interrupt must be unregistered:

```
int32_t
    Irq_UnregisterDiIrq( MyRio_IrqDi* irqChannel,
                         NiFpga_IrqContext irqContext,
                         uint8_t irqNumber);
```

using the same above arguments. To use the pthread functions, `#include <pthread.h>` in your code.

[1] Note: The IRQ channel settings symbols (and others) associated with the DI interrupt, are defined in header files: DIIRQ.h and IRQConfigure.h.

Interrupt Thread

This is the separate thread that was named and started by the `pthread_create()` function. Its overall task is to perform any necessary function in response to the interrupt. This thread will execute until signaled to stop by `main.c`.

The beginning of the new thread is the starting routine specified in the `pthread_create()` function called in `main.c`: `void *DI_Irq_Thread(void* resource)`.

The first step in `DI_Irq_Thread()` is to cast its input argument into appropriate form. In our case, we cast the `resource` argument back to the `ThreadResource` structure. For example, declare

```
ThreadResource* threadResource = (ThreadResource*) resource;
```

The second step is to enter a loop. Two tasks are performed each time through the loop:

```
- while the main thread does not signal this thread to stop {
    1. Wait for the occurrence (or timeout) of the IRQ.
    2. if the numbered IRQ has been asserted {
            - Perform operations to service the interrupt.
                 For our case, print "interrupt_" on the LCD.
            - "Acknowledge" the interrupt.
     }
  }
```

Let's see how this is done:

The while loop should continue until the `irqThreadRdy` flag (set in `main.c`) indicates that the thread should end. For example,

```
while (threadResource->irqThreadRdy == NiFpga_True) { ...
```

The two tasks within the loop are:

1. Use the `Irq_Wait()` function to pause the loop while waiting for the interrupt. For our case the call might be:

   ```
   uint32_t irqAssert = 0;
   Irq_Wait( threadResource->irqContext,
             threadResource->irqNumber,
             &irqAssert,
             (NiFpga_Bool*) &(threadResource->irqThreadRdy));
   ```

   Notice that it receives the `ThreadResource` context and IRQ number information, and returns the `irqThreadRdy` flag set in the `main.c` thread.

2. Because the `Irq_Wait()` times out after 100 ms, we must check the `irqAssert` flag to see if our numbered IRQ has been asserted.

   In addition, after the interrupt is serviced, it must be acknowledged to the scheduler. For example,

   ```
   if (irqAssert & (1 << threadResource->irqNumber)) {

       % Your interruptservice code here

       Irq_Acknowledge(irqAssert);
   }
   ```

The third step terminates the new thread and returns from the function:

```
pthread_exit(NULL);
return NULL;
```

Laboratory Procedure

Build, debug, and execute your program.

Provide interrupt signal by connecting the single-pole-double-throw (SPDT) switch on the circuit bread board to `DIO0` of connector-A as shown in Figure 1. Try your program. What happens? This undesirable phenomenon is caused by the "bounce" of the mechanical switch.

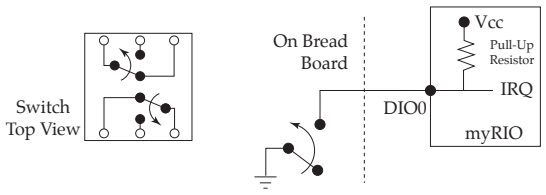

Figure 1: Connecting the interrupt signal to myRIO.

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?

Correct the problem by replacing the switch in Figure 1 with the "debouncing" circuit shown in Figure 2. This circuit incorporates a Transistor-Transistor-Logic (TTL) Quad open-collector NAND gate (7401).

Caution: Be certain that V_{cc} and GND are connnected to the chip before wiring the rest of the circuit.

Try your program again.

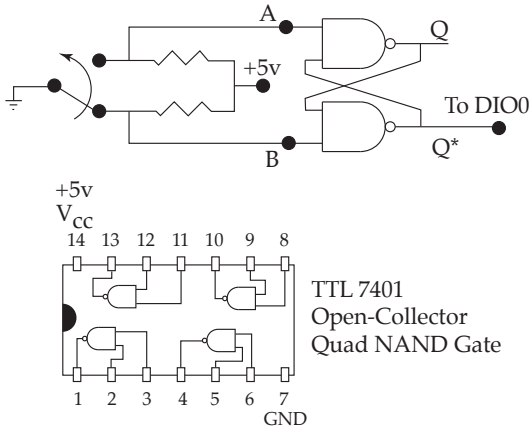

Figure 2: Debouncing circuit.

Explain, in detail, why this circuit should solve the switch bounce problem. That is, graph the time-history of signals at points A and B that would occur during the operation of a bouncing switch. Then, graph the corresponding signals at Q and $Q*$.

Finally, in your own words, explain how the main program thread configures the interrupt thread, how it communicates with the interrupt thread during execution, and how the interrupt thread functions.