Lab Exercise 05 Introduction to interrupts

Lab 05.1 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.

Lab 05.2 Introduction

This exercise illustrates the use of interrupts, originating from sources that are external to the microcomputer. The principal activity of your `main` program is to print the value of a counter on the LCD 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 routine* (ISR) thread will simply print out the message, "`interrupt_`". A push-button switch on an external circuit will cause the IRQ to occur.

interrupt request (IRQ)

interrupt service routine (ISR)

Therefore, the overall effect will be that the display will print the count repeatedly, with the word "`interrupt_`" interspersed 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 causes the activity of the CPU to be suspended, and (2) a separate section of code (ISR) executes, 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.

REVIEW DRAFT

Lab 05.3 The Threads

Lab 05.3.1 The `main` thread

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

1. Open the myRIO session.
2. *Register the interrupt* and the digital input (see below).
3. *Create an interrupt thread* to "catch" the interrupt (see below).
4. Begin a loop. Each time through the loop:
    - Wait one second by calling the (5 ms) `wait` function (from Lab Exercise 04) 200 times.
    - Clear the display and print the value of an `int 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 session.

Lab 05.3.2 The ISR thread

The ISR runs in an interrupt thread, separate from the main thread. It should begin a 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.

Lab 05.4 Background

Several library interrupt functions are used in the following. For more documentation on them, see Resource 11.

Lab 05.4.1 Setting up `main` for interrupts, generally

Within `main` 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;
```

REVIEW DRAFT

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

Register the DI0 IRQ The first of these functions reserves the interrupt from the 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 the interrupt condition is met to trigger the interrupt.
5. `type`: the trigger type used to increment the count.

The returned value is `0` for success.

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

```
int pthread_create(
  pthread_t *thread,
```

REVIEW DRAFT

```
  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 the starting function in the new thread. The prototype syntax means the function start_routine, which will be given argument arg in the new thread, should be given to pthread_create with *no argument*.
4. arg: the sole argument to be passed to start_routine. In our case, it will be a *pointer* to the thread resource structure defined above and used in the second argument of Irq_RegisterDiIrq.

This function returns 0 for success.

Lab 05.4.2 Setting up main *for our interrupt, specifically*

We can combine these ideas into *a portion* of the main code needed to initialize the DI IRQ.[2] For interrupts on falling-edge transitions on DIO0 of Connector A, assigned to IRQ 2, we have:

```
int32_t status;
MyRio_IrqDi irqDI0;
ThreadResource irqThread0;
pthread_t thread;
int i, j, count=0;

// Open the myRIO NiFpga Session.
status = MyRio_Open();
if (MyRio_IsNotSuccess(status)) return status;

// Configure the DI IRQ number, incremental times,
// and trigger type.
const uint8_t IrqNumber = 2;
const uint32_t Count = 1;
const Irq_Dio_Type TriggerType = Irq_Dio_FallingEdge;
```

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

REVIEW DRAFT

```
// Specify the settings that correspond to
// the IRQ channel to be accessed.
irqDI0.dioChannel = Irq_Dio_A0;
irqDI0.dioIrqNumber = IRQDIO_A_0NO;
irqDI0.dioCount = IRQDIO_A_0CNT;
irqDI0.dioIrqRisingEdge = IRQDIO_A_70RISE;
irqDI0.dioIrqFallingEdge = IRQDIO_A_70FALL;
irqDI0.dioIrqEnable = IRQDIO_A_70ENA;

// Initiate the IRQ number resource of interrupt thread.
irqThread0.irqNumber = IrqNumber;

// Register  DI0 IRQ. Terminate if not successful.
status=Irq_RegisterDiIrq(
  &irqDI0,
  &(irqThread0.irqContext),
  IrqNumber,
  Count,
  TriggerType
);
if (status != NiMyrio_Status_Success) {
  printf(
    "Status: %d\nConfiguration of DI IRQ failed\n",
    status
  );
  return status;
}

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

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

REVIEW DRAFT

Other `main` tasks go here.

After the other `main` tasks 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;
status = pthread_join(thread,NULL);
```

Finally, the interrupt must be unregistered:

```
status = 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.

Lab 05.4.3 The ISR 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`.

The beginning of the new thread is the starting routine specified in the `pthread_create` function called in `main`: `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, as described in Algorithm 7.

Let's explore how to do this. The `while` loop should continue until the `irqThreadRdy` flag (set in `main`) indicates that the thread should end. For example:[3]

```
while (threadResource->irqThreadRdy == NiFpga_True) {
  // stuff!
}
```

[3] For pointer to a structure `struct * a` with member name `b`, the member value can be accessed with `a->b`, which is equivalent to `(*a).b`.

REVIEW DRAFT

Algorithm 7 ISR thread loop pseudocode

```
while the main thread has not signaled this thread to stop do
    wait for the occurrence (or timeout) of the IRQ
    if the numbered IRQ has been asserted then
        perform operations to service the interrupt (print interrupt_)
        acknowledge the interrupt
    end if
end while
```

The two tasks within the loop are as follows.

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` thread.

2. Because `Irq_Wait` times out after 100 ms, we must check the `irqAssert` *bit flag*[4] 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, using *bitwise operators*,[5]

   bit flag

   bitwise operators

   ```
   if (irqAssert & (1 << threadResource->irqNumber)) {
     // Your interrupt service code here
     Irq_Acknowledge(irqAssert);
   }
   ```

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

[4] A bit flag is bit of independently useful information stored in a (larger) integer variable. This is because a byte is the smallest addressable unit of memory. Of course, multiple bit flags can be assigned to a single integer variable.

[5] The bitwise operator `<<` shifts `1` of `...0001` left `irqNumber` bits. Then the bitwise *and* `&` "bit masks" to see if any bits of both numbers match (there's only potentially one match, the `irqNumber` bit). Note that any nonzero integer is considered *true* (`1`) for a conditional statement.

REVIEW DRAFT

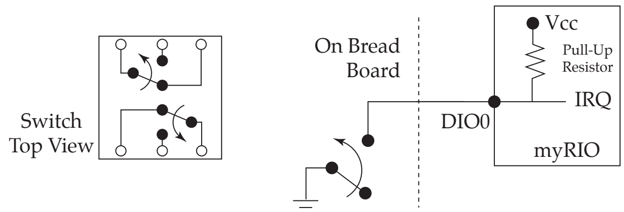

Figure 05.3: Connecting the interrupt signal to myRIO.

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

Lab 05.5 Laboratory procedure

Build, debug, and execute your program.

Provide interrupt signal by connecting the single-pole-double-throw (SPDT)[6] switch on the circuit bread board to `DIO0` of Connector A as shown in Figure 05.3. Try your program. What happens? This undesirable phenomenon is caused by the *bounce* of the mechanical switch.

bounce

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 05.3 with the *debouncing circuit* shown in Figure 05.4. This circuit incorporates a (TTL) quad open-collector NAND gate (7401).

debouncing circuit

Box 05.1 caution

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

Try your program again. 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` thread configures the interrupt thread, how it communicates with the interrupt thread during execution, and how the interrupt thread functions.

[6] The switch is actually double-pole-double-throw (DPDT), but one pole is disconnected.

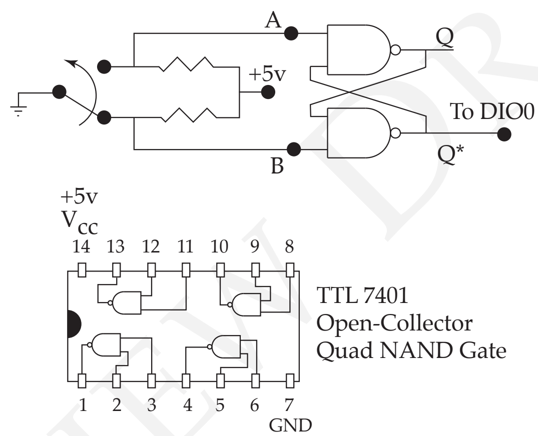

Figure 05.4: Debouncing circuit.

REVIEW DRAFT