

INTERFACE NOTES

v4.0

Analog Initialization

For our project we will use the analog input channel CIO and the analog output channel CO0 on connector-C. They communicate with the processor through the FPGA.

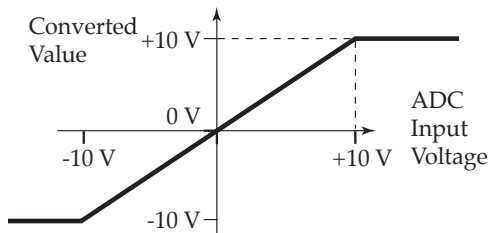
Before they can be used they must be initialized using

```
AIO_initialize(&CIO, &CO0);
```

Call it once, where CIO and CO0 are structures that must be of type MyRio_Aio. This initialization function is included in the me477Library.

Analog-to-Digital Converter

The single-channel 12-bit analog-to-digital converter (ADC) measures the current value of the applied voltage in the range [-10.000, +9.995] V. Voltages outside that range “saturate” the conversion as shown.



The ADC has a resolution of 4.883 mV, with absolute accuracy of ± 200 mV. Each channel has input impedance > 500 k Ω . Overload protection: ± 16 V.

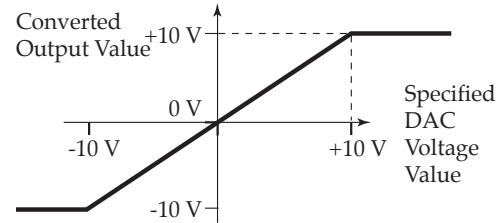
Our library contains a function that reads a specified channel of the ADC, and returns the converted value. Its prototype is:

```
double Aio_Read(MyRio_Aio* channel);
```

where `channel` is the pointer to the channel structure defined above: `&CIO`.

Digital-to-Analog Converter

The single-channel 12-bit digital-to-analog converter (DAC) produces a voltage at the output terminal in the range [-10.000, +9.995] V. Again, specified voltages outside that range “saturate” the conversion as shown. The



DAC has a resolution of 4.883 mV, with absolute accuracy of ± 200 mV. Each channel has a maximum drive current of 3 mA, and a maximum slew rate of $2V/\mu s$. Overload protection: ± 16 V.

Our library contains a function that accepts a specified channel for the DAC, and returns the converted value. Its prototype is:

```
void Aio_Write(MyRio_Aio* channel, double value);
```

where `channel` is the pointer to the channel structure defined above: `&CO0`, and `value` is the specified value of the analog output voltage.

Timer IRQ

Main Thread: Background

Initializing the Timer interrupt is similar to initializing the Digital Input interrupt.

We will use a separate thread to produce interrupts at periodic intervals. Within `main.c` we will configure the Timer 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_Boolean irqThreadRdy; // IRQ thread ready flag
} ThreadResource;
```

National Instruments provides C functions to set up the Timer interrupt request (IRQ).

1) Register the Timer IRQ – The first of these functions reserves the interrupt from FPGA and configures the Timer and IRQ. Its prototype is:

```
int32_t Irq_RegisterTimerIrq( MyRio_IrqTimer* irqChannel,
                             NiFpga_IrqContext* irqContext,
                             uint32_t timeout);
```

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 `TimerIRQ.h` as:

```
typedef struct {
    uint32_t timerWrite; // Timer IRQ interval register
    uint32_t timerSet; // Timer IRQ setting register
    Irq_Channel timerChannel; // Timer IRQ supported I/O
} MyRio_IrqTimer;
```

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. `timeout` - The timeout interval in μs .

The returned value is 0 for success.

2) Create the interrupt thread – A new thread must be configured to service the Timer interrupt. In `main.c` we will use `pthread_create()` to set up that thread. 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 and used in the second argument of `Irq_RegisterTimerIrq()`.

This function also returns 0 for success.

Main Thread: Our Case

We can combine these ideas into a *portion* of the `main.c` code needed to initialize the timer IRQ.¹ For interrupts triggered by the timer in the FPGA, we have:

```
int32_t status;
MyRio_IrqTimer irqTimer0;
ThreadResource irqThread0;
pthread_t thread;

// Registers corresponding to the IRQ channel
irqTimer0.timerWrite = IRQTIMERWRITE;
irqTimer0.timerSet = IRQTIMERSETTIME;
timeoutValue = 5;

status = Irq_RegisterTimerIrq( &irqTimer0,
                              &irqThread0.irqContext,
                              timeoutValue);

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

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

Finally, the timer interrupt must be unregistered:

```
status = Irq_UnregisterTimerIrq( &irqTimer0,
                                irqThread0.irqContext);
```

using the same above arguments.

¹The IRQ settings symbols associated with the timer interrupt, are defined in the header file: `TimerIRQ.h`.

3) The 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 run until signaled to stop by `main.c`.

The new thread is the starting routine specified in the `pthread_create()` function called in `main.c`. In our case: `void *Timer_Irq_Thread(void* resource)`.

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

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

The **second step** is to enter a while loop. Two functions 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.
     - if it has, "schedule" the next interrupt.
  2. if the Timer IRQ has been asserted {
     - Perform operations to service the interrupt.
     - Acknowledge the interrupt.
  }
}
```

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

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,
          TIMERIRQNO,
          &irqAssert,
          (NiFpga_Boot*) &(threadResource->irqThreadRdy));
```

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

Schedule the next interrupt by writing the time interval into the `IRQTIMERWRITE` register, and setting the `IRQTIMERSETTIME` flag. That is,

```
NiFpga_WriteU32( myrio_session,
                 IRQTIMERWRITE,
                 timeoutValue );

NiFpga_WriteBool( myrio_session,
                  IRQTIMERSETTIME,
                  NiFpga_True);
```

The `timeoutValue` is the number of microseconds (`uint32_t`) until the next interrupt. The `myrio_session` used in these functions should be declared within this timer thread. That is,

```
extern NiFpga_Session myrio_session;
```

This variable was defined when you called `MyRio_Open()` in the main thread.

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

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

```
if(irqAssert) {
    % Your interrupt service code here

    Irq_Acknowledge(irqAssert);
}
```

In the **third step** (after the end of the loop) we terminate the new thread, and return from the function:

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