06.4 The biquad cascade

Although we could implement Eq. 4 as shown, the sensitivity of the output to the coefficients leads to numerical inaccuracies as the order of the system N becomes large. We will solve this problem by breaking the Nth order system it into a series of n_s second-order systems.
The technique is called a biquad cascade and is illustrated in Figure 06.1.
Notice that the output of each second-order section (biquad)[4] is the input to the subsequent section. Each biquad implements the same second-order difference equation, but with different coefficients, inputs, and outputs.
For example, the current output $y_i(n)$ from the ith section would be:

$$y_i(n) = \frac{1}{a_{0_i}}\big(b_{0_i}x_i(n) + b_{1_i}x_i(n-1) + b_{2_i}x_i(n-2) + \\ - a_{1_i}y_i(n-1) - a_{2_i}y_i(n-2)\big). \tag{1}$$

4. "Biquad" is short for "biquadratic." The biquad transfer function has second-order polynomials in both numerator and denominator.

Of course, a first or second order transfer function would require only one biquad. Depending on the value of N, some of the coefficients of at least one biquad may be zero. We will implement a function to handle any value of N.
There are a variety of algorithms for breaking a transfer function into biquadric sections. Matlab's Signal Processing Toolbox contains a

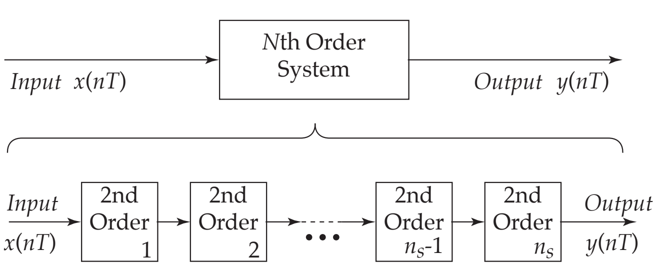

Figure 06.1: a biquad cascade.

function `tf2sos` (transfer function to second order sections) for this purpose.

Resource R12 Timer interrupts

This resource describes how to program the myRIO in C to perform timer interrupts.

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`, 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_Bool irqThreadRdy;     // IRQ thread ready flag
} ThreadResource;
```

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

Register the Timer IRQ

The first of these functions reserves the interrupt from the 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 three 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.

Create the interrupt thread

A new thread must be configured to service the timer interrupt. In `main` 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`: the name of the 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_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 Timer IRQ.[5]

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

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` are completed, it should signal the new thread to terminate by setting the `irqThreadRdy` flag in the `ThreadResource` structure. Then it should 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 arguments from above.

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`.
The new thread is the starting routine specified in the `pthread_create` function called in `main`. 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, as described in Algorithm 06.1.

Algorithm 06.1 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
    schedule the next interrupt
    if the Timer IRQ has been asserted then
        perform operations to service the interrupt
        acknowledge the interrupt
    end if
end while
```

The `while` loop should continue until the `irqThreadRdy` flag (set in `main`) 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, with `TIMERIRQNO` a constant defining the Timer

IRQ's IRQ number, defined in `TimerIRQ.h`:

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

Notice that it receives the `ThreadResource` context and Timer IRQ number information, and returns the `irqThreadRdy` flag set in the `main` 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 μs (`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 & (1 << TIMERIRQNO)) { // Bit mask
  // 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;
```