06.L Lab Exercise: Transfer function generator

Objectives

The objectives of this exercise are to:

1. Use real-time clock interrupts to provide timing.
2. Implement an arbitrary transfer function generator.
3. Introduce A/D and D/A conversion.

Introduction

In this lab exercise, you will write a general purpose program capable of approximating the performance of any SISO, LTI system! The system input and output will both be analog electrical signals. Your program will implement this with a difference equation.
At the beginning of each BTI, your ISR will read an analog input to obtain the current input value, compute the current value of the output $y(n)$, and apply the current output value to an analog output.
This process continues until ← is entered on the keypad. The input voltage will be provided by a function generator. Both the input and output voltages will be displayed on the oscilloscope.
You will use three new myRIO features in this experiment: an interrupt timer, the ADC, and the DAC. The first is described in detail in Resource 12 and the others in Resource 14.
Although we could implement the difference equation 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, so we use the biquad cascade representation of Lec. 06.4 .

Pre-laboratory preparation

The program consists of a `main` function and an interrupt service routine (ISR) running in a separate thread. The ISR is set to execute with a period of 0.5 ms (determined by the Timer IRQ), and computes the DAC output from the ADC input by means of a difference equation.

Main program

The only tasks of `main` will be the following.

1. Set up and enable the Timer IRQ interrupt,
2. Enter a loop, ending only when a ← is received from the keypad. Use `getkey`.
3. Signal the timer thread to terminate using the `irqThreadRdy` flag, and wait for it to terminate.

Interrupt service routine

The interrupt service routine thread implements a dynamic system. The heart of the ISR is a while loop that checks the `irqThreadRdy` flag (set in `main`) to see if the thread should continue. Before the loop begins, initialize the analog input/output, and set the analog output to 0 V. Each time through the loop:

1. Get ready for the next interrupt by waiting for the IRQ to assert. Then write the time interval to wait between interrupts (BTI) to the `IRQTIMERWRITE` register and write `TRUE` to the `IRQTIMERSETTIME` register.
2. Read the analog input to obtain the current input value $x(n)$.
3. Call a function `cascade` (see below) to calculate the current value of the output $y(n)$ by computing all of the sections in the biquad cascade. Each biquad section is computed according to Eq. 1.
4. Send the output value to the analog output.

5. Acknowledge the interrupt.

After the loop terminates, save the response to `Lab6.mat`.
The ISR must allocate storage for variables and arrays associated with the discrete dynamic system, including:

1. the length of the BTI in microseconds,
2. the number of biquad sections n_s, and
3. the system constants (a_i and b_i) for the biquad sections.

The dynamic system corresponding to the collection of biquad sections can be conveniently referred to and manipulated by first defining a structure to contain the coefficients and previous values of input and output for a single biquad section. We might define a "biquad" structure as follows.

```
struct biquad {
  double  b0; double  b1; double  b2; // numerator
  double  a0; double  a1; double  a2; // denominator
  double  x0; double  x1; double  x2; // input
  double  y1; double  y2;             // output
};
```

This definition should be placed just before the prototypes section of your program.
Then, a specific dynamic system can be defined as an array of these biquad structures, each array element corresponds to an individual biquad section:

```
int myFilter_ns = 2;     // No. of sections
uint32_t timeoutValue = 500;  // T - us; f_s = 2000 Hz
static struct biquad myFilter[] = {
  {1.0000e+00,  9.9999e-01, 0.0000e+00,
   1.0000e+00, -8.8177e-01, 0.0000e+00, 0, 0, 0, 0, 0},
  {2.1878e-04,  4.3755e-04, 2.1878e-04,
   1.0000e+00, -1.8674e+00, 8.8220e-01, 0, 0, 0, 0, 0}
};
```

This system description can be placed within the ISR, near its beginning. The first two lines

establish the number of biquad sections, and the length of the BTI in microseconds. Finally, `myFilter` is the name of an array of biquad structures being initialized.
For testing purposes, the initialized constants in the example above correspond to a system of two biquad sections ($n_s = 2$), encoding a unity-gain low-pass filter, with sampling frequency of 2000 Hz. Derived using Tustin's method, they correspond to a third-order continuous system having a pair of complex poles with natural frequency of 40 Hz, and with damping ratio 0.5. The remaining real pole is at 40 Hz.

Crazy about pointers!

The most challenging part of this task is the calculation of the current output value $y(n)$. The use of pointers makes the calculation both straightforward and efficient.

Box 06.1 hint

Don't be tempted to code this algorithm using array indices (instead of pointers); that would be much too slow for our purposes.

The *cascade* function

The `cascade` function implements the complete dynamic system by passing the measured input through the string of biquad sections. The ISR must pass to `cascade` the value of the current input $x(n)$ measured by the ADC, the number of biquad sections n_s, the array of biquad structures containing the coefficients and history variables (x_i and y_i) for all sections. It might have a prototype that looks like:

```
double cascade(
  double xin,         // input
```

```
  struct biquad *fa,  // biquad array
  int    ns,          // no. segments
  double ymin,        // min output
  double ymax         // max output
);
```

Here, `xin` is the current system input, `fa` is the name of an array of biquad structures, `ns` is the corresponding number of biquad sections, and `ymin` and `ymax` are the saturation limits.
In the above example, `myFilter` would be passed through `fa`. The value returned by `cascade` is the current value of the system output $y(n)$.

Coding *cacade*

An efficient way to code `cascade` is to allocate a pointer `f` in `cascade` that will be used to point to elements of the array of biquad structures.
Begin the function by equating the pointer to the first element in the array (i.e. the first biquad): `f = fa;`. Variables inside the biquad structure are accessed by using the pointer name, e. g. `f->a0`, `f->b0`, `f->x0`, `f->y1`, etc. (The `->` operator is equivalent to dereferencing and then accessing a member (say, `(*f).a0`) and is typed as a minus sign immediately followed by >.)
Then, loop n_s times, to cycle through each of the biquad sections in the array. At the beginning of each loop, the output value `y0` of previous biquad must be passed to the input value `f->x0` of the current biquad.
Within the loop, coding the output value `y0` might look like:

```
y0 = (
  f->b0*f->x0 + f->b1*f->x1 + // ... etc.
)/f->a0;
```

See Equation 1.
Each time through the loop, after the output value has been computed, the previous values x

and y must be updated, so that they will be correct at the next time step. For example,

```
f->x2 = f->x1; f->x1 = f->x0; // ... etc.
```

At the end of the loop, the pointer `f` is incremented to advance to the next biquad in the array.
One more point: if the DAC is given a value beyond its range $[-10, +10]$ V, it will saturate its output value appropriately. However, our difference equation Eq. 1 depends on previous values of the output, but doesn't saturate. To correct this disparity, `cascade` should saturate the output `y0` of the final biquad before it is saved for the next iteration.
For example, define the macro

```
#define SATURATE(x,lo,hi) { \
  ((x) < (lo) ? (lo) : (x) > (hi) ? (hi) : (x)) \
}
```

Pass appropriate values of the `xmin` and `xmax` parameters to `cascade`. Then, for the last biquad, immediately after `y0` is computed, saturate its value:
`y0 = SATURATE(y0, ymin, ymax);`

Laboratory Procedure

A good strategy to follow in writing this program is to first implement and debug everything except the calculation of the biquad cascade. That is, set up the `main` program and the ISR, including all arrays and timing. In the ISR, simply pass the input value from the ADC directly to the DAC. For example,

```
VADin = Aio_Read(&CIO);
Aio_Write(&COO, VADin);
```

This will allow you to observe the input and output on the oscilloscope, and determine if the interrupt timing is functioning properly.

When you have debugged those portions of the program augment the code above with the call to `cascade`.

Does it work?

The low-pass digital filter described above was derived using Tustin's method from the transfer function of the three-pole continuous system:

$$\frac{V_{out}(s)}{V_{in}(s)} = \frac{\omega_n^3}{(s+\omega_n)(s^2+2\zeta\omega_n s+\omega_n^2)} \quad (1)$$

where $\omega_n = 2\pi \times 40$ rad/s, and $\zeta = 0.5$. This system belongs to a class of filters called Butterworth filters. They are signal processing filters designed to have the flattest possible frequency response in the passband.

Step response

Using the oscilloscope (DC coupled), observe the step response of the system by applying a low frequency square wave (e.g. at 8 Hz) with an amplitude of 5 V as the input with the function generator.
Save the input and output of `cascade` in 500-point buffers. After the timer loop ends, save the buffers to `Lab6.mat`, and transfer the data to Matlab. Plot and compare the measured step response to the theoretical response of the corresponding continuous system.[6]. Explain.

6. To simulate the theoretical response, Matlab's `lsim` is a good choice.

Frequency response

Again, using the oscilloscope (AC coupled), observe the frequency response by altering the frequency of a 5 V input sine wave.
Record (write down) the amplitude and phase of the output relative to the input sine wave at the following frequencies:
$[5, 10, 20, 40, 60, 100, 140, 200]$ Hz. Given the input amplitude, compute the transfer function magnitude (dB) at each frequency.

In Matlab, plot the theoretical magnitude (dB) and phase (deg) versus the frequency (Hz on a logarithmic scale) for the continuous system transfer function.[7] Plot the corresponding measured data as discrete symbols on top of the theoretical frequency response. Explain.

7. To generate the data for this plot, Matlab's `bode` is a good choice. Note that the specifications for the plot format require you to generate the plot separately from your call to `bode`.

Resource R13 Discrete−time controllers

For reference, Table 06.1 contains Tustin equivalents for some common continuous-time controllers.

Table 06.1: Tustin equivalents for common continuous-time controllers. Usage of z is contextual, meaning a zero in continuous transfer functions and meaning the z-transform z in discrete transfer functions.

	phase lag/lead	PI	PID
continuous	$k\dfrac{s+z}{s+p}$	$K_p + \dfrac{K_i}{s}$	$K_p + \dfrac{K_i}{s} + K_d s$
discrete	$k\dfrac{b_0 + b_1 z^{-1}}{a_0 + a_1 z^{-1}}$	$\dfrac{b_0 + b_1 z^{-1}}{a_0 + a_1 z^{-1}}$	$\dfrac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{a_0 + a_1 z^{-1} + a_2 z^{-2}}$
differential equation	$\dfrac{dy}{dt} + py = k\left(\dfrac{dx}{dt} + zx\right)$	$y = K_p x + K_i \int_0^t x\,dt$	$y = K_p x + K_i \int_0^t x\,dt + K_d \dfrac{dx}{dt}$
difference equation	$y(n) = -\dfrac{a_1}{a_0} y(n-1) + \dfrac{b_0}{a_0} x(n) + \dfrac{b_1}{a_0} x(n-1)$	$y(n) = -\dfrac{a_1}{a_0} y(n-1) + \dfrac{b_0}{a_0} x(n) + \dfrac{b_1}{a_0} x(n-1)$	$y(n) = -\dfrac{a_1}{a_0} y(n-1) - \dfrac{a_2}{a_0} y(n-2) + \dfrac{b_0}{a_0} x(n) + \dfrac{b_1}{a_0} x(n-1) + \dfrac{b_2}{a_0} x(n-2)$
a_0	1	1	1
a_1	$(pT-2)/(pT+2)$	-1	0
a_2			-1
b_0	$k(zT+2)/(pT+2)$	$K_p + K_i T/2$	$K_p + K_i T/2 + 2K_d/T$
b_1	$k(zT-2)/(pT+2)$	$-K_p + K_i T/2$	$K_i T - 4K_d/T$
b_2			$-K_p + K_i T/2 + 2K_d/T$

Resource R14 Analog input and output

Analog initialization

For our project, we will use the analog input channel `CI0` 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(&CI0, &CO0);
```

Call it once, where `CI0` and `CO0` are structures that must be of type `MyRio_Aio`. This initialization function is included in the `me477` library.

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 in Figure 06.1. The ADC has a resolution of 4.883 mV, with absolute accuracy of ± 200 mV. Each channel has input impedance of > 500 kΩ and overload protection of ± 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);
```

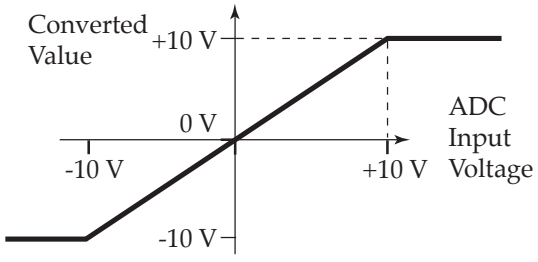

Figure 06.1: ADC saturation.

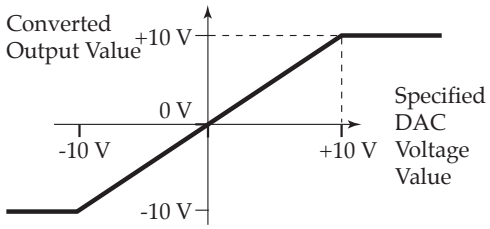

Figure 06.2: DAC saturation.

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 in Figure 06.2. The DAC has a resolution of 4.883 mV, with absolute accuracy ± 200 mV. Each channel has a maximum drive current of 3 mA, a maximum slew rate of 2 V/µs, and an overload protection of ± 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.

07

Closed–loop control