GE420 Laboratory Assignment 5

DAC and ADC Signal I/O

Goals for this Lab Assignment:

1. Introduce the hardware interrupt (HWI) SYS/BIOS object.
2. Demonstrate signal aliasing.
3. Implement a simple proportional control to compare simulation to an actual implementation.

DSP/BIOS Objects Used:

- Hwi
- Clock

Library Functions Used:

setDAC1, setDAC2

Matlab Functions Used:

rltool

Prelab Exercise

Read through sections of the TMS320F28377S Technical Reference Guide that explain interrupts and the ADC module. The ADC module is very complex so I am not asking you to understand completely what you are reading but these sections should give you a good introduction to this peripheral. There are also assigned sections to read about the EPWM peripheral. In regards to the ADC we are just going to use the EPWM module as a timer and not as an output driving say our DC motor. Read the following assigned sections:

Chapter 2.4, 2.4.1, 2.4.2, 2.4.3, 2.4.4,

Chapter 2.4.5: In Table 2-2, what PIE interrupt is ADCB1 assigned to? In Table 2-3, what Vector ID # is ADCB1 assigned to?

Chapter 9.1, 9.1.2, 9.1.3, 9.1.4 important, 9.1.5 look at the different examples, 9.1.6 try to understand this idea of sequencing, In the register section 9.4 find and study a bit the registers ADCSOCPRICTL, ADCSOCFRC1, ADCSOC0CTL and ADCSOC1CTL.

Chapter 13.10, see that the EPWM module can start the conversion of one or multiple ADC channels.

Laboratory Exercise

Exercise 1: Using the ADC

To demonstrate the hardware interrupt (HWI) section in SYS/BIOS, we will take advantage of the fact that the TMS320F28377S processor's ADC generates an interrupt when its sequence of ADC samples have finished and the converted values are stored in the results registers of the ADC peripheral. When the ADC conversion is complete, the DSP automatically stops the code it is currently processing and jumps to the interrupt service routine (ISR) specified for the ADC. On completion of the ISR code, the program counter (PC) will automatically jump back to the code that was interrupted and resume processing.

The two ADC channels we will be using in lab are ADCB0 and ADCB1. These are wired to the two ADC banana jacks on the top of our experiment. We are not using ADCA0 and ADCA1 because those two pins can also be configured

as DACA and DACB channels, the two DAC channels we worked with in Lab 4. Pins ADCB0 and ADCB1 are brought through a multiplexer inside the F28377S processor into the ADCB ADC peripheral. Since they both can be routed into the same ADC they cannot be sampled at exactly the same time but using the sequencer of ADCB they can be sampled one right after the other. We will initialize ADCB to perform 12 bit ADC conversions. The range of voltage for this ADC is 0 volts to 3.0 volts. So a 12 bit result equaling 0(decimal) indicates that 0.0 volts is coming into the ADC. The maximum value of a 12 bit ADC result, 4095, indicates that 3.0 volts is on the ADC input pin. Then there is a linear interpolation between 0 and 4095 covering all the volts from 0.0 to 3.0 volts with steps of 3.0V/4095 = .73mv.

For our lab assignments using the ADC, we are going to need are larger voltage swing of 0.0 to 3.0 volts. Instead we are going to need a swing from -10.0 volts to 10.0 volts. This is achieved for you in our experiment by adding a gain and shift op amp circuit. This circuit wires the op amps as inverting op amps so keep in mind the sign change. So the final scaling of our input voltage into the ADC's banana jacks is that -10.0 volts equates to an ADC result of 4095(decimal) and 10.0 volts equates to 0(decimal). For example, a voltage input of 5.0 volts would give an ADC result of 1024(decimal).

We will be commanding ADCB to sample channels ADCB0 and ADCB1 in sequence and after completion of these two conversions the results will be located in AdcbResultRegs.ADCRESULT0 and AdcbResultRegs.ADCRESULT1. In addition, after the two conversions, the ADCB1 (this means ADCB interrupt source 1) interrupt will be flagged. Your job will be to write the interrupt service routine (ISR) that is called after the interrupt source is flagged.

Your first task will be to build an application that samples both of the ADC channels (B0 and B1) and echoes their values to the two DAC channels. Your application should:

1. Before adding any code first look at the function SetupADCSoftware() in f28377sADC.c. In the ADCB section of the code notice that SOC0 is assigned to ADCB0 channel and SOC1 is assigned to ADCB1. Also notice that no TRIGGER select was set for these SOCs, so the only way currently to tell the ADC to convert is to force it by writing to the ADCSOCFRC1 register. Also notice that ADCB is setup to flag an interrupt when SOC1 is finished converting which makes sense since we first want SOC0 to convert channel ADCB0 and the SOC1 to convert channel ADCB1.
2. Now start coding by including a Clock object that is called every 1 ms. The only job of the Clock's associated function is to start the ADC conversion. All the rest of the code you write will be put in the ADCB's ISR discussed below. Your prelab reading had you look at the ADCSOCFRC1. In your Clock function add only the following code and complete the assignment of ADCSOCFRC1 to force the conversion of both SOC0 and SOC1. This way ADCB will be told to convert every 1ms.

```
 // Start ADC conversion.
AdcbRegs.ADCSOCPRICTL.bit.RRPOINTER = 0x10;  //rr pionter reset, soc0 next
AdcbRegs.ADCSOCFRC1.all |= ??????;        //start conversion B0 and B1
```

3. In your *.cfg file, add a hardware interrupt instance and associate it with a function that you will be creating. Also set it so that the interrupt source that will cause this interrupt is ADCB1's vector ID number. You can find this number in Table 2-4 of the F28377S technical reference guide. Leave all the other items in the Hwi instance as the default: Enable on Startup, Automatically Acknowledge PIE interrupts and MaskingOption_SELF. The associated function should first read the two ADC integer values stored in the result registers. Then using the conversion from 0 to 4095 to 10.0 Volts to -10.0 Volts, convert the 12 bit sampled integer values to volts stored in global float variables. Then simply write these two floating point voltage values to the two DAC channels. As

a final step clear the interrupt source flag so that processor will wait for the next interrupt flag before the ADC ISR is called again. The below code has all these steps except for the conversion to -10 Volts to 10 Volts and writing those values to the DACs.

```
//adcb1 pie interrupt
void processAdcb(void)
{
   adcb0result = AdcbResultRegs.ADCRESULT0;
   adcb1result = AdcbResultRegs.ADCRESULT1;
   // Here convert to Volts

   // Here write values to DAC channels

   AdcbRegs.ADCINTFLGCLR.bit.ADCINT1 = 1;  //clear interrupt flag
}
```

4. With this echo program you can show the effects of signal aliasing. Drive the ADC input with the function generator at your bench. Input a sine wave with amplitude 8 volts, (volts peak to peak of 16) to the DSP and watch the DAC output on the oscilloscope. Vary the frequency of the input sine wave and demonstrate at what frequency the output begins aliasing.

Exercise 2: Proportional controller

1. First disconnect the Function Generator cable. We will not need the Function Generator for this section because the step input to drive the system will be generated in your code.
2. Wire the Comdyna GP-6 analog computer as the double integrator (second order) system shown in Figure 5 below. The transfer function of this system is $\frac{10}{s^2+s+1}$. It is a standard second order system, $gain*\frac{w_n^2}{s^2+2s\zeta w_n+w_n^2}$, with w_n = 1, ζ= 0.5 and a gain of 10. Set the analog computer's potentiometer 1 equal to 0.1 in order to set ζ equal to 0.5. Make sure to be looking at a color print out of the wiring diagram so you do not miss any wires. Ask your TA to help you with this.

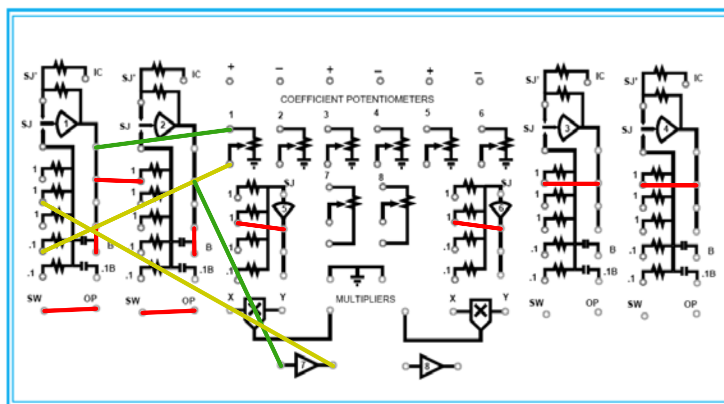

Figure 5: Analog computer wiring diagram

3. In Code Composer, change your code so that it now performs the following:

a. Read ADC1 every 50 milliseconds. ADC1 should be connected to the output of op-amp #2. This is the output or feedback signal of our system. Also connect channel 1 of the oscilloscope to ADC1 so you can view the system response.

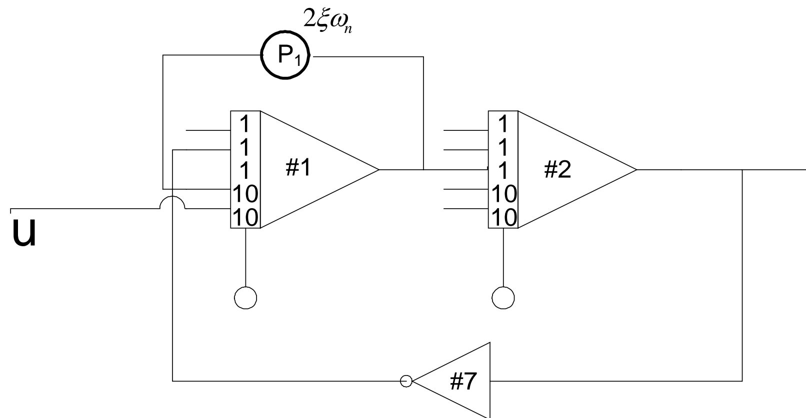

Figure 6: Effective analog computer diagram

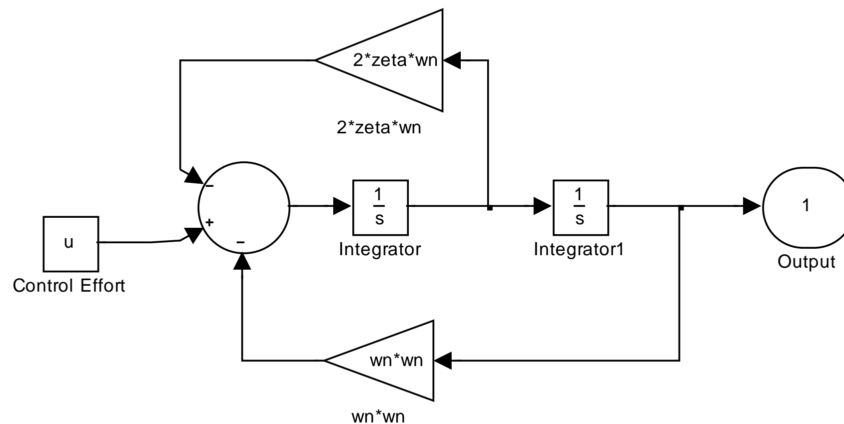

Figure 7: All integrator block diagram

b. Write code in your interrupt function that changes a floating point variable between 0.5 and –0.5 every 15s. This will act as our step (reference) input to the proportional controller.

c. Implement a proportional controller with the equation:

$$u = (\text{reference} - \text{ADC}) * K_{Proportional}$$

d. Output this u variable's value to DAC1. Connect DAC1 to one of Op-amp #1's 10x inputs. This is our input or control effort to the plant. Also in order to view the reference step input, output to DAC2 the reference value. Connect channel 2 of the oscilloscope to DAC2.

e. Print ADC1, reference and u values to the LCD every 100ms.

f. Starting with a $K_{Proportional}$ equal to 0.1, build and run your controller. Before running your code, set the analog computer to "IC" (initial condition) mode. Then run your DSP code. Finally, press the "OP" (operate) button on the analog computer to activate the plant. Observe the response on the oscilloscope.

g. Increase the proportional gain $K_{Proportional}$ at increments of 1.0. Use MATLAB's "rltool" function and Simulink to compare the root locus of the system to its response at each increment of $K_{Proportional}$. A Simulink simulation file of the double integrator system can be found at < n:\labs\ge420\lab5\double_integrator.mdl >. Notice how the simulation is converting the continuous transfer function to a discrete transfer function using the "C2D" Matlab function. Your TA will familiarize you with this simulation and rltool. Also note at what value the $K_{Proportional}$ gain makes the systems go unstable.

Exercise 3: Have EPWM5 start ADC conversion.

1. To show an additional feature of this processor, have EPWM5 acting simply as a timer start the ADC conversion sequence. When you were writing the above code you may have thought it was a waste to create a Clock function that only told the ADC to convert. And indeed it is not necessary as both timers and PWM timers can tell the ADC to run its conversions. So as a first step modify the SetupADCSoftware() in f28377sADC.c by adding two lines in the ADCB section of code and finding for the register explanation of ADCSOC0CTL what to set TRIGSEL to.

```
AdcbRegs.ADCSOC0CTL.bit.TRIGSEL = ??????;  // EPWM5 ADCSOCA will cause SOC0
AdcbRegs.ADCSOC1CTL.bit.TRIGSEL = ??????;  // EPWM5 ADCSOCA will cause SOC1
```

2. Then in main() after the two UART_printfLine function calls add the following lines of code. This code sets up EPWM5 to time out almost every 50 milliseconds. Explain to your TA why this does not time out exactly every 50 ms.

```
EALLOW;
EPwm5Regs.ETSEL.bit.SOCAEN = 0; // Disable SOC on A group
EPwm5Regs.TBCTL.bit.CTRMODE = TB_FREEZE; // freeze counter
EPwm5Regs.ETSEL.bit.SOCASEL = 2; // Select Event when equal to PRD
EPwm5Regs.ETPS.bit.SOCAPRD = 1; // Generate pulse on 1st event
EPwm5Regs.TBCTR = 0x0000; // Clear counter
EPwm5Regs.TBPHS.bit.TBPHS = 0x0000; // Phase is 0
EPwm5Regs.TBCTL.bit.PHSEN = TB_DISABLE; // Disable phase loading
EPwm5Regs.TBCTL.bit.CLKDIV = 6; // divide by 64  50Mhz divide by 64 is 781250 Hz
EPwm5Regs.TBPRD = 39062;  // 50ms sample actually 39062.5 is 50ms.
EPwm5Regs.ETSEL.bit.SOCAEN = 1; //enable SOCA
EPwm5Regs.TBCTL.bit.CTRMODE = TB_COUNT_UP; //unfreeze, and enter up count mode
EDIS;
```

3. Because EPWM5 will now tell the ADC to convert every 50ms we no longer need the Clock instance and its function. In your *.cfg file delete the Clock instance and comment out your function that was associated with the Clock. Build and Run your code and you should see your code run exactly the way it performed when using the Clock to start the ADC.

Lab Check Off:

1. Demonstrate signal aliasing.
2. Demonstrate the implemented proportional control and how it compares to simulation.
3. Demonstrate your code working with EPWM5 triggering ADCB conversions.