SE420 Laboratory Assignment 4 (Two Week Lab)

ADC (Analog to Digital Converter) and Sampling

Goals for this Lab Assignment:

1. Introduce the hardware interrupt (HWI)
2. Demonstrate signal aliasing
3. Observe effect of oversampling on system response
4. Design and Implement a few Discrete Filters.

Library Functions Used:

setDAC1(float value), setDAC2(float value)

Prelab Exercise

Read through sections listed below of the TMS320F28379D Technical Reference Guide that explain interrupts and the ADC module. These sections should give you an introduction to the ADC peripheral. I have also assigned sections to read about the EPWM peripheral. We will use the EPWM module just as a timer to trigger the ADC and not as an output driving something (such as driving a DC motor). Read the following assigned sections:

- **Sections 3.4, 3.4.1 to 3.4.5:** In 3.4.5, Table 3-2, what PIE interrupt is ADCD1 assigned to?
- **Sections 11.1, 11.2, 11.5 and 11.6:** We will be using 12bit resolution. 11.5 is very important and 11.6 has examples. Try to understand the idea of sequencing presented in 11.6.
- **Section 11.16:** Find and study the registers ADCSOCPRICTL, ADCSOCFRC1, ADCSOC0CTL, ADCSOC1CTL, ADCSOC2CTL and ADCSOC3CTL.
- **Section 15.10:** Read how the EPWM module can start the conversion of one or multiple ADC channels.

Laboratory Exercise

Exercise 1: Using the ADC

For demonstrating the hardware interrupt (HWI), we will take advantage of the fact that the TMS320F28379D processor's ADCD generates an interrupt when its sequence of ADC samples have finished and the converted values are stored in the results registers of the ADCD 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.

We will use ADC channels ADCIND0, ADCIND1, ADCIND2 and ADCIND3 in this lab assignment. We will sample all four channels but use only ADCIND0 and ADCIND1 for this lab. Pins ADCIND0, ADCIND1, ADCIND2 and ADCINB3

are brought through a multiplexer inside the F28379D processor into the ADCD ADC peripheral. Since all four can be routed into the same ADC, they cannot be sampled at the same time. However, using the sequencer of ADCD, they can be sampled sequentially. We will initialize ADCD to perform 12 bit ADC conversions. The range of input 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. Hence, there is a linear prgression between 0 and 4095 covering all the voltages from 0.0 to 3.0 volts with steps of 3.0V/4095 = .73mv.

We will command ADCD to sample channels ADCIND0, ADCIND1, ADCIND2 and ADCIND3 in sequence. After these four conversions are completed, the results will be stored in registers AdcdResultRegs.ADCRESULT0, AdcdResultRegs.ADCRESULT1, AdcdResultRegs.ADCRESULT2 and AdcdResultRegs.ADCRESULT3. In addition after these four conversions, the ADCD1 interrupt will be flagged and executed. In the function "SetupADCSoftware()" the ADCD1 interrupt was told to be initiated when SOC3 has finished converting. Your job will be to write the interrupt service routine (ISR) that is called after this interrupt source is flagged.

Your first task will be to build an application that samples all 4 ADC channels ADCIND0, ADCIND1, ADCIND2 and ADCIND3 and echoes ADCIND0 and ADCIND1's voltage result to the DAC1 and DAC2 channels.

1. Before adding any code, first look at the function SetupADCSoftware() in f28379dADC.c. There is a bunch of commented code in this function that setup ADCA and ADCB. I left it there for reference if you ever need to use the ADCA or ADCB peripherals. In the ADCD section of the code notice that SOC0 is assigned to ADC channel ADCIND0, SOC1 is assigned to ADCIND1, SOC2 is assigned to ADCIND2 and SOC3 is assigned to ADCIND3. Also notice that no TRIGGER select was set for these SOCs, so with this code, the only way to tell the ADC to convert is to force it by writing to the ADCSOCFRC1 register. Also notice that ADCD is setup to flag an interrupt when SOC3 is finished converting. This makes sense since we first want SOC0 to convert channel ADCIND0, then the SOC1 to convert channel ADCIND1, then the SOC2 to convert channel ADCIND2 and last SOC3 to convert ADCIND3. This is the default order for the round robin sequencer.
2. Change CPUTimer0 to have a period of 1ms. You will add code to CPUTimer0's interrupt routine to make it start the ADCD conversion every 1 millisecond. The remainder of your code for this exercise will be put in the ADCD1's ISR (interrupt service routine). Recall what you read about the ADCSOCFRC1 register for the prelab. In your CPUTimer0 interrupt function, add only the following two lines of code and complete the assignment of ADCSOCFRC1 to force the conversion of SOC0, SOC1, SOC2 and SOC3.

```
// Start ADC conversion.
AdcdRegs.ADCSOCPRICTL.bit.RRPOINTER = 0x10;  //rr pionter reset, soc0 next
AdcdRegs.ADCSOCFRC1.all |= ??????; //start conversion of ADCIND0, ADCIND1, ADCIND2 & ADCIND3
```

3. Now add your ADCD1 hardware interrupt function. *Note: The naming can get confusing here. There are ADCD inputs channels labeled ADCIND0, ADCIND1, etc and there are also four ADCD interrupts labeled ADCD1, ADCD2, ADCD3 and ADCD4. In the earlier code called in main() we setup ADCD1 interrupt to be call when all four channels ADCIND0, ADCIND1, ADCIND2 and ADCIND3 are finished converting.* You will be adding most of the code for this exercise inside this ADCD1 hardware interrupt function. Perform the following steps:
    - Create your interrupt function as a global function with "void" parameters and of type "__interrupt void" for example __interrupt void ADCD_ISR(void) {
    - Put a predefinition of your ISR function at the top of your C file in the same area as the predefinitions of the CPU Timer ISRs.
    - Now inside main() find the PieVectTable assignments. This is how you tell the F28379d processor to call your defined functions when certain interrupt events occur. Looking at Table 3-2 in the F28379d Technical Reference find ADCD1. You should see that it is PIE interrupt 1.6. Since TI labels this interrupt ADCD1, there is a PieVectTable field named ADCD1_INT. So inside the EALLOW and EDIS statement assign PieVectTable.ADCD1_INT to the memory address location of your ISR function. For example &ADCD_ISR.
    - Next step is to enable the PIE interrupt 1.6 that the F28379D associated with ADCD1. A little further down in the main() code, find the section of code of "IER |=" statements. This code is enabling the base interrupt for the multiple PIE interrupts. Since ADCD1 is a part of interrupt INT1, INT1 needs to be enabled. Well, Timer 0's interrupt is also a part of interrupt 1. So the code we need is already there "IER |= M_INT1;". You do though need to enable the 6th interrupt source of interrupt 1. Below the "IER |=" statements you should see the enabling of TINT0 which is PIEIER1.bit.INTx7. Do the same line of code but enable PIE interrupt 1.6.
    - Now with everything setup to generate the ADCD1 interrupt, put code in your ADCD1 interrupt function to read the value of the ADCIND0 and ADCIND1 channels and echo their value to the DAC1 and DAC2 output channels. ADCD0, ADCD1, ADCD2 and ADCD3, are setup to convert the input voltage on their corresponding pin to a 12 bit integer. The range of the ADC inputs can be from 0.0V to 3.0V. So the 12 bit conversion value which has a value between 0 to 4095, linearly represents the input voltage 0.0V to 3.0V. The 12 bit conversion values are stored in the results registers, see below. Create four global int16_t results variables and two global float variables to store the scaled voltage values of ADCIND0 and ADCIND1. Once you have converted the ADCIND0 12 bit integer result value to voltage, pass that variable to the setDAC1() function. Do the same thing with the ADCIND1 value to setDAC2().

- Set UARTPrint = 1 every 100ms and change the code in the main() while loop so that your two ADC voltage values are printed to TeraTerm. Make sure to create your own int32_t count variable for this ADCD interrupt function. It is normally not a good idea to use a count variable from a different timer function.
- As a final step, clear the interrupt source flag and the PIE peripheral so that processor will wait for the next interrupt flag before the ADC ISR is called again. The below code has many of these steps.

```
//adcd1 pie interrupt
__interrupt void ADCD_ISR (void)
{
  adcd0result = AdcdResultRegs.ADCRESULT0;
  adcd1result = AdcdResultRegs.ADCRESULT1;
  adcd2result = AdcdResultRegs.ADCRESULT2;
  adcd3result = AdcdResultRegs.ADCRESULT3;

  // Here covert ADCIND0, ADCIND1 to volts

  // Here write values to DAC channels

  // Print ADCIND0 and ADCIND1's voltage value to TeraTerm every 100ms

  AdcdRegs.ADCINTFLGCLR.bit.ADCINT1 = 1;  //clear interrupt flag
  PieCtrlRegs.PIEACK.all = PIEACK_GROUP1;

}
```

4. Using this echo program, you can show the effects of sampling and signal aliasing. Have your TA show you how to connect the appropriate signals to your F28379D Launchpad board. Using both the PinMux table and the F28379D Launchpad schematic, connect the ADCIND0 input to the function generator at your bench. Input a sine wave with amplitude 2.5 volts peak to peak and an offset of 1.5volts. MAKE SURE to put the function generator in "High Z" output. Watch the DAC output on the oscilloscope. Vary the frequency of the input sine wave and demonstrate at what frequency the output begins aliasing. What do you notice about sampling of a sinewave at 10Hz, 100Hz, and 250Hz?

Exercise 2: Have EPWM5 start ADC conversion and use Oversampling

EPWM5 will be used in this exercise just as a timer to signal to ADCD when to convert its ADC channels. EPWM5 was chosen since it is more than likely that EPWM5 will not be needed to drive PWM outputs. Oversampling means that we will convert the same ADC channels multiple times in the time of one sample and use the average of the multiple conversions as our reading for the sample. For example, we will tell the ADC to convert 16 times

in 1 millisecond (So a conversion every 1/16 of a millisecond). Once we have the 16 conversions, they will be averaged and the average will be the value used for that sample period of 1 ms. Oversampling can reduce noise in the input signal. We will oversample all four channels ADCIND0, ADCIND1, ADCIND2 and ADCIND3 but only echo the values of ADCIND0 and ADCIND1 to the DAC channels.

1. To show an additional feature of this processor, we will have EPWM5 act simply as a timer to start the ADC conversion sequence. When you were writing and understanding Exercise 1's above code, you may have thought that it was a waste to have CPUTimer0's interrupt tell the ADC to convert and that was pretty much all it did. It is indeed not necessary as EPWM peripheral can be used just as a timer and trigger the ADC to run its conversions. To do this we will need to change a few setups from Exercise 1. First open the f28379dADC.c file and copy the function SetupADCSoftware(). Paste the function in your C file somewhere above main() but not above the #include statements. Change the name of this function to SetupADCOverSampling(). In main() find where you are calling SetupADCSoftware() and change the line to call SetupADCOverSampling(). In this new function keep all the code but add the below four lines of code. These lines of code tell the ADCD peripheral to schedule the four SOC's for conversion whenever the EPWM5->TBCTR register reaches the value in the EPWM5->TBPRD register. You will need to look in the ADC register section of the F28379D Technical Reference Guide for the explanation of ADCSOC0CTL and find the required value of TRIGSEL. This will cause ADCIND0, ADCIND1, ADCIND2 and ADCIND3's conversion whenever the EPWM5 times out. It will trigger them all at the same time so SOC0 will convert first and then SOC1, then SOC2 and then SOC3.

```
AdcdRegs.ADCSOC0CTL.bit.TRIGSEL = ??????;  // EPWM5 ADCSOCA will cause SOC0
AdcdRegs.ADCSOC1CTL.bit.TRIGSEL = ??????;  // EPWM5 ADCSOCA will cause SOC1
AdcdRegs.ADCSOC2CTL.bit.TRIGSEL = ??????;  // EPWM5 ADCSOCA will cause SOC2
AdcdRegs.ADCSOC3CTL.bit.TRIGSEL = ??????;  // EPWM5 ADCSOCA will cause SOC3
```

2. Then in main(), after the SetupADCOverSampling() function call, add the following lines of code. This code sets up EPWM5 to time out every 1/16th millisecond and enables its SOCA trigger that happens when TBCTR equals the value in TBPRD to command ADCD to start a conversion sequence.

```
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 = 0; // divide by 1  50Mhz Clock
EPwm5Regs.TBPRD = 3125;  // (1/16)ms sample.
EPwm5Regs.ETSEL.bit.SOCAEN = 1; //enable SOCA
```

```
EPwm5Regs.TBCTL.bit.CTRMODE = TB_COUNT_UP; //unfreeze, and enter up count mode
EDIS;
```

Since EPWM5 will now tell the ADC to convert every (1/16) ms, we no longer need to force the ADCD conversion in CPUTimer0's ISR function. Remove AdcdRegs.ADCSOCPRICTL.bit.RRPOINTER and AdcdRegs.ADCSOCFRC1.all lines of code you added in Exercise 1.

3. With these added changes, the ADCD1 interrupt function will be called every 1/16 of a millisecond but we still want our overall sample period to be 1 millisecond. So the ADCD1 interrupt function is just going to be used to read the ADCD results registers, convert their value to a voltage between 0.0V and 3.0V, keep a running sum of those voltages and keep track of the number conversions taken. Once 16 samples have been collected the ADCD1 ISR should divide the running sum by 16, store this as the 1 millisecond sample, reset the running sum to zero along with the conversion count and post the function I am calling a Software Interrupt (SWI). This will allow the ADCD1 interrupt function to continue be called every 1/16 of a millisecond and the SWI function can run the every 1ms code that in Exercise 3 will have you implement filter equations. The SWI will be discussed in lecture but mainly it is the lowest priority hardware interrupt that has been setup to allow other hardware interrupts to interrupt it's processing. So code in the SWI is more time critical than main()'s while(1) loop but not as time critical as the other hardware interrupts. Here in this exercise we will only write this averaged ADC value to DAC1 inside the SWI and set UARTPrint every 100ms to print the averaged voltage readings. Look at the ADC interrupt code below for clarification.

```
// global variables
float sumAdcd0volts = 0;
float sumAdcd1volts = 0;
float sumAdcd2volts = 0;
float sumAdcd3volts = 0;
int sampleCount = 0; // counts number of times ADC channels have been sampled
float avgADCd0 = 0;
float avgADCd1 = 0;
float avgADCd2 = 0;
float avgADCd3 = 0;

//adcd1 pie interrupt
__interrupt void ADCD_ISR (void)
{
   adcd0result = AdcdResultRegs.ADCRESULT0;
   adcd1result = AdcdResultRegs.ADCRESULT1;
   adcd2result = AdcdResultRegs.ADCRESULT2;
   adcd3result = AdcdResultRegs.ADCRESULT3;

   sampleCount++;
   sumAdcd0volts += ????; // equation converting 12bit value to 0 to 3V voltages
   sumAdcd1volts += ????; // equation converting 12bit value to 0 to 3V voltages
   sumAdcd2volts += ????; // equation converting 12bit value to 0 to 3V voltages
   sumAdcd3volts += ????; // equation converting 12bit value to 0 to 3V voltages
```

```
    if (sampleCount == 16){
      avgADCd0 = sumAdcd0volts/16.0;  // oversampled value
      avgADCd1 = sumAdcd1volts/16.0;  // oversampled value
      avgADCd2 = sumAdcd2volts/16.0;  // oversampled value
      avgADCd3 = sumAdcd3volts/16.0;  // oversampled value
      // Reset variables for next cycle of oversampling
      sampleCount = 0;
      sumAdcd0volts = 0;
      sumAdcd1volts = 0;
      sumAdcd2volts = 0;
      sumAdcd3volts = 0;

      // This below line causes the SWI to run when priority permits
      PieCtrlRegs.PIEIFR12.bit.INTx9 = 1;  // Manually cause the interrupt for the SWI
    }

    AdcdRegs.ADCINTFLGCLR.bit.ADCINT1 = 1;  //clear interrupt flag
    PieCtrlRegs.PIEACK.all = PIEACK_GROUP1;
  }

  __interrupt void SWI_isr(void) {
    // These three lines of code allow SWI_isr, to be interrupted by other interrupt functions
    // making it lower priority than all other Hardware interrupts.
    PieCtrlRegs.PIEACK.all = PIEACK_GROUP12;
    asm("      NOP");                // Wait one cycle
    EINT;                            // Clear INTM to enable interrupts

    // Insert SWI ISR Code here.....

    numSWIcalls++;

    DINT;

  }
```

4. The SWI is posted after the 16th ADC conversion and hence the SWI runs every 1 ms (16 x (1/16) ms). Inside the SWI function __interrupt void SWI_isr(void) move the code from the ADCD1 interrupt function that writes to the DAC channels and write the ADCIND0 and ADCIND1 oversampled values to the DAC channels. In addition, every 100ms, set UARTPrint so that your new averaged voltage readings are printed to TeraTerm.
5. Build and Run your code. Observe any differences between this response and exercise 1's response on the oscilloscope.

Exercise 3: Implementing discrete Filter's and a look at Numerical Issues

In digital control design, it is useful to think of your system and digital controller in both continuous time and discrete time. Since you will be implementing the controller in source code on a microcontroller, the final controller must be represented in discrete equations but it still is useful to look at the controller in continuous time. So throughout the remaining labs you will be asked to represent your system and controller in continuous time (Laplace transform) and possibly perform some design steps in continuous time. In addition you will convert the continuous equations to discrete time representations (Z transform) and perform design steps on the discrete system. Always though, the final design and tests must be performed in discrete time because you will be implementing the controller in discrete time. For example, during your design process in these upcoming labs you will be performing simulations in Matlab/Simulink. Part of this design will be done in continuous time and part in discrete time. The final design and simulation, though, must be done in discrete time.

In this exercise, we are going to look at filtering the input voltage sampled by ADCD. The transfer functions of the filter will be designed in the continuous Laplace domain and then converted to discrete Z domain transfer functions for simulation and implementation.

An important filter we will be using throughout this semester is a simple single pole continuous transfer function $\frac{500}{s+500}$. Filtering out noise is an important part of digital control but you have to be careful with filtering at too high of an order of filter. The higher the order, the more phase lag is added to your overall system. This is why we will start out with a low order filter and see if it works well enough. The 500 in $\frac{500}{s+500}$ is a value that is "tunable." Depending on the time response (bandwidth) desired the 500 could be changed higher or lower to 100 or 10, etc. Start this exercise by looking at the difference between $\frac{500}{s+500}$, $\frac{100}{s+100}$, $\frac{10}{s+10}$. First at the Matlab command prompt, using a script M-file, enter in these three continuous transfer functions, i.e. tf100 = tf(100,[1 100]). Using a sample period of 0.001 seconds convert these transfer functions to the Z domain i.e. tf100D = c2d(tf100,.001,'tustin'). Type "help c2d" at Matlab's command prompt for more information on c2d.

> *Notice here the Tustin (Trapezoidal Integration method) method instead of the Zero Order Hold (zoh) is used to discretize the transfer function. This is because the ZOH method should only be used for the system being sampled, i.e. the transfer function of a motor being controlled. All semester it will be emphasized that the continuous system we are trying to controller will be discretized using "ZOH". The continuous controller will be discretized using an emulation method like Tustin or the backwards rule.*

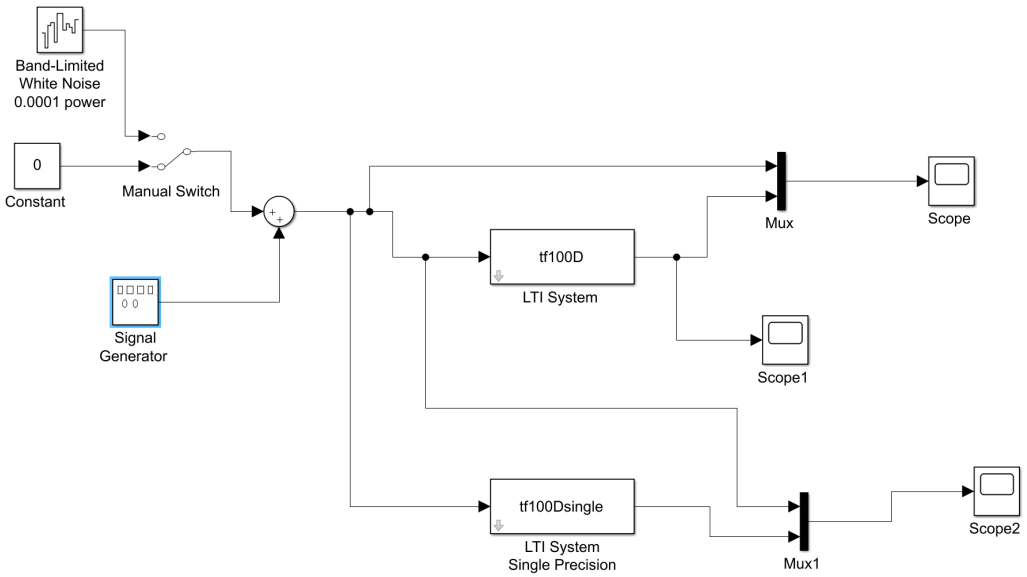
Then compare the Bode plots and step responses of these three discrete transfer functions i.e. step(tf500D,tf100D,tf10D) and bode(tf500D,tf100D,tf10D) . **Show your TA and comment on the difference between the filters.**

Perform some more comparisons of these three filters using Simulink. Create/copy this script M-file to setup the transfer functions needed for the comparisons.

> *Note that we are creating a "Single" (32bit float) precision version of the transfer function. Matlab by default uses 64 bit double precision numbers. With these first order transfer functions, we will not see any numerical issues but in steps below when you design higher order filters you will have to check for numerical issues with the 32 bit float. The reason we like 32 bit floats is that code using 32 bit floats runs faster than code written using 64 bit doubles on the F28379D processor.*

```
clear all;
tf500 = tf(500,[1 500]);
tf500D = c2d(tf500,.001,'tustin');
bsingle = single(tf500D.num{1});
asingle = single(tf500D.den{1});
tf500Dsingle = tf(bsingle,asingle,.001);
tf100 = tf(100,[1 100]);
tf100D = c2d(tf100,.001,'tustin');
bsingle = single(tf100D.num{1});
asingle = single(tf100D.den{1});
tf100Dsingle = tf(bsingle,asingle,.001);
tf10 = tf(10,[1 10]);
tf10D = c2d(tf10,.001,'tustin');
bsingle = single(tf10D.num{1});
asingle = single(tf10D.den{1});
tf10Dsingle = tf(bsingle,asingle,.001);
```

Then replicate the Simulink simulation here in this figure. I will go over Simulink in lecture. Most, if not all, of these Simulink blocks can be found by left clicking once in white space of the Simulink file and typing in the name of the block. Band-Limited White Noise set to 0.0001 power and 0.001 sample time. Signal Generator set to Wave form sine, Time Use simulation time, Amplitude 3, Frequency 2, Units Hertz. The LTI System block just needs the name of the filter's transfer function created by your M-file.



When you have all the Simulink blocks wired as shown, right click in white space and select "Model Configuration Parameters." In the "Solver" item, set Type to Fixed-step. In "Solver details" set Fixed Step Size to 0.001.

Start with tf500D and the manual switch set to no noise. Set the Signal Generator to a number of frequencies between 0 and 250 Hertz and compare the input sine wave to the output sine wave. I am having you stop at 250 Hertz because we are sampling at 1000 Hertz and anything larger than 250 Hz will have less than 4 samples per period. You may have to change the "Stop Time" of the simulation to create a useful plot that shows only a few periods of the sine wave. Remember that the unit of the 500 in $\frac{500}{s+500}$ is radians/second so the cutoff frequency of this filter is $\frac{500}{2\pi}$Hertz. You should see that the single precision transfer function has the same output as the normal transfer function calculated with double precision numbers.

While comparing to the bode plot of the filter, note the amplitude change and phase lag between the input and output at different frequencies. The simulation should match the Bode plot. With a frequency of 3 Hz, switch on the noise and note how well the filter gets rid of the noise in the output signal. Perform this same analysis for $\frac{100}{s+100}$ and $\frac{10}{s+10}$. Report what you found to your TA.

Now implement these filters one at a time by developing code to filter the ADCIND0 input. Just as in Exercise 1 and 2, bring the function generator's output into ADCIND0 and scope both that input signal along with the output signal from DAC1 (pin 30). Develop this code by starting with the code you created for exercise 2. (Make a backup of your Exercise 2 code so you have a copy for your lab submission.) If you recall this code oversampled the ADCD channels every 1/16 ms and then found the average of those 16 samples to produce an averaged sample every 1ms. This in itself is a type of filter. For this exercise, I would like you to analyze the filters you implement and compare them to their Bode plots. If you have the oversampling also happening on the ADCD

channel it will be harder to compare the output to its Bode plot. For this reason change the code for this exercise to just have EPWM5 trigger the ADCD channels every 1ms instead of every 1/16 ms. In addition, get rid of taking the average of the 16 samples inside the ADCD1 ISR. Get this code working first and just echoing the ADCIND0 conversion to DAC1 very similar to what you did in exercise 1 but now using EPWM5.

Once the echoing code is working, implement the discrete filter equations (difference equations). I will go over this implementation of the discrete transfer function to difference equations in lecture, but we have already done a simple transfer function in Lab 3 to calculate the velocity of the motor. When transferring the coefficients of your discrete transfer function it is very important that you copy all the precision of the numbers from Matlab to your C code. I have created two functions to do this for you in both single precision numbers and double precision numbers. arraytoCformat.m and arraytoCformat_double.m. Both of these functions are already on the lab computers. If you are using Matlab on your own PC, please copy these two files into your Matlab work directory. We will always attempt to use single precision floats in our code since they process faster. If needed we will use double precision but then take note of how fast our code is processing. To copy your filter coefficients to C run the commands

arraytoCformat(tf500D.num{1}); and arraytoCformat(tf500D.den{1});

The use of this arraytoCformat function to copy your coefficients to C is not as important with these first order filters, but when you switch to a fourth order filter below, copying all the precision of the numbers to your C program will mean the difference between a filter that works and a filter that does not.

Using the arraytoCformat function you can create an array num[2] and an array den[2]. Inside the ADCD1 ISR apply the filter difference equations on the sample of ADCIND0. Here is some partial code to get your started.

```
xk = adcind0volt;
yk = num[0]*xk + num[1]*xk_1 – den[1]*yk_1;  // Make sure you know where this equation came from
setDAC1(yk);  // write filtered value to the DAC to see filtered value on the Oscilloscope
// save past xk states for next interrupt call to this function
// save past yk states for next interrupt call to this function
```

With your first order filter equations, try a number of input frequencies and note the phase shift and amplitude difference between the input and output sinewave. Try all three of your filters tf500D, tf100D, tf10D by copying each of their coefficients to your C code using arraytoCformat in Matlab. **Show your TA your code with all three filter coefficients and explain what observations you found at different frequencies and with different first order filters.**

In some cases, these first order filter transfer functions do not accomplish the amount of filtering needed. So higher order filters may be necessary. For this exercise I do not want to get into whether or not this order of filter is needed or necessarily the best, instead I want to use these higher order filters to give you more practice implementing difference equations from discrete transfer functions and also look at numerical error. So one

method of choosing and implementing a filter, is to string a number of these low order filters together. In other words find your continuous filter transfer function $\frac{500}{s+500} * \frac{500}{s+500} * \frac{500}{s+500} * \frac{500}{s+500} = \frac{62500000000}{(s+500)(s+500)(s+500)(s+500)}$. Using this technique, create two more fourth order transfer functions from the other two first order transfer functions you used above. Again play around with these filters in Simulink noticing the phase lag and amplitude changes given different frequencies. Also check if there are any differences between the single precision and double precision transfer function responses. You should find that the $\frac{10}{s+10} * \frac{10}{s+10} * \frac{10}{s+10} * \frac{10}{s+10}$ filter has some numerical issues when using single precision. **Show the Simulink responses of these three fourth order filters to you TA.**

Since we saw some numerical issues with one of the fourth order transfer functions, implement the fourth order difference equations using the "long double" type. *The F28379D processor's C compiler has three types for floating point variables, float (32 bit), double (32 bit) and long double (64bit). As you can see with TI defining a double as a 32 bit floating point number, TI is discouraging the use of 64bit floating point variables on this processor because they run at least twice as slow. But for these small difference equations we can use the 64 bit "long double" type in our code.* I have created another function for Matlab that helps with double precision coefficients, arraytoCformat_double(). You will need to define all the variables that are involved with the filter difference equations as "long double" instead of float. **Show to your TA your fourth order filter implementation that filters the input from ADCIND0 and sends the filtered output to DAC1. Make sure to show that your $\frac{10}{s+10} * \frac{10}{s+10} * \frac{10}{s+10} * \frac{10}{s+10}$ filter works with the long double filter implementation.**

Lab Check Off:

1. Demonstrate your sampled signal and signal aliasing.
2. Demonstrate your sampled signal when over oversampled.
3. Demonstrate your First order Filter's Bode plot and Step response
4. Demonstrate your Simulink file helping you test out your filter before implementing.
5. Show your three first order filter transfer functions implemented C and working to filter ADCIND0
6. Show the numerical problem with one or more of your fourth order filters.
7. Demonstrate your code that implements the fourth order filter equations using "long double" coefficients and variables.