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 TMS320F28377S 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 ADCB1 assigned to?
- **Sections 10.1, 10.2, 10.5 and 10.6:** We will be using 12bit resolution. 10.5 is very important and 10.6 has examples. Try to understand the idea of sequencing presented in 10.6.
- **Section 10.16:** Find and study the registers ADCSOCPRICTL, ADCSOCFRC1, ADCSOC0CTL, ADCSOC1CTL, ADCSOC2CTL.
- **Section 14.10:** 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 TMS320F28377S processor's ADCB generates an interrupt when its sequence of ADC samples have finished and the converted values are stored in the results registers of the ADCB 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 sample ADC channels ADCINB0, ADCINB1 and ADCINB2 in this lab assignment, but actually, we are only going to use ADCINB0 in this lab. I am having you also sample ADCINB1 and ADCINB2 to show you how multiple ADC channels (ADC input pins) can be sampled with one event and so you have starter code sampling the cantilever beam's feedback for Lab 8. Pins ADCINB0, ADCINB1 and ADCINB2 are brought through a multiplexer inside the F28377S processor into the ADCB peripheral. Since all three can be routed into the same ADC, they cannot be sampled at the same time. However, using the sequencer of ADCB, they can be sampled sequentially. We will initialize ADCB to perform 12 bit ADC conversions. The range of input voltage for this ADC is 0 volts to 3.0 volts. But for both ADCINB0 and ADCINB1, I have placed a gain and shift front end circuit that allows 10V to -10V to be input to those two channels. For ADCINB2, which is connected to the cantilever beam experiment's Hall Effect sensor, it only needs the base voltage range of 0V to 3V. So for channels ADCINB0 and ADCINB1, a 12 bit result equaling 0(decimal) indicates that 10.0 volts is coming into the ADC. The maximum value of a 12 bit ADC result, 4095, indicates that -10.0 volts is on the ADC input pin. Hence, there is a linear progression between 0 and 4095 covering all the voltages from 10.0 to -10.0 volts with steps of -20.0V/4096 = -4.883mV.
We will command ADCB to sample channels ADCINB0, ADCINB1 and ADCINB2 in sequence. After these three conversions are completed, the results will be stored in registers AdcbResultRegs.ADCRESULT0, AdcbResultRegs.ADCRESULT1 and AdcbResultRegs.ADCRESULT2. In addition, after these three conversions, the ADCB1 interrupt will be flagged and executed. In the function "SetupADCSoftware()" the ADCB1 interrupt was told to be initiated when SOC2 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 the three ADC channels ADCINB0, ADCINB1 and ADCINB2 and echoes ADCINB0's voltage result to the DAC1 channel.

1. Before adding any code, first look at the function SetupADCSoftware() in f28377sADC.c. There is additional code, some commented out, in this function that sets up ADCA. I left it there for reference if you ever need to use the ADCA peripheral. In the ADCB section of the code notice that SOC0 is assigned to ADC channel ADCINB0, SOC1 is assigned to ADCINB1 and SOC2 is assigned to ADCINB2. 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 ADCB is setup to flag an interrupt when SOC2 is finished converting. This makes sense since we first want SOC0 to convert channel ADCINB0, then the SOC1 to convert channel ADCINB1 and last SOC2 to convert channel ADCINB2. 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 ADCB conversions every 1 millisecond. The remainder of your code for this exercise

will be put in the ADCB1's ISR (interrupt service routine). Recall what you read about the ADCSOCFRC1 register for the prelab. In your CPUTimer0 interrupt function, add the following two lines of code and complete the assignment of ADCSOCFRC1 to force the conversion of SOC0, SOC1 and SOC2.

```
// Start ADC conversion.
AdcbRegs.ADCSOCPRICTL.bit.RRPOINTER = 0x10;  //rr piointer reset, soc0 next
AdcbRegs.ADCSOCFRC1.all |= ??????; //start conversion of ADCINB0, ADCINB1 & ADCINB2
```

3. Now add your ADCB1 hardware interrupt function. *Note: The naming can get confusing here. There are ADCB inputs channels labeled ADCINB0, ADCINB1, etc and there are also four ADCB interrupts labeled ADCB1, ADCB2, ADCB3 and ADCB4 (We will only be using ADCB1 interrupt). The given LABstarter code sets up the ADCB1 interrupt to be called when all three channels ADCINB0, ADCINB1 and ADCINB2 are finished converting.* You will be adding most of the code for this exercise inside this ADCB1 hardware interrupt function (code template given below the next steps). Perform the following steps:
    - Create your interrupt function as a global function with "void" parameters and of type "__interrupt void" with the following line of code `__interrupt void ADCB_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 F28377s processor to call your defined functions when certain interrupt events occur. Looking at Table 3-2 in the F28377s Technical Reference find ADCB1. You should see that it is PIE interrupt 1.2. Since TI labels this interrupt ADCB1, there is a PieVectTable field named ADCB1_INT. So in between the EALLOW and EDIS statements, assign PieVectTable.ADCB1_INT to the memory address location of your ISR function, &ADCB_ISR. Look at the other PieVectTable assignments for an example.
    - Next step is to enable the PIE interrupt 1.2 that the F28377S associated with ADCB1. A little further down in the main() code, find the section of code with "IER |=" statements. This code is enabling the base interrupt for the multiple PIE interrupts. Since ADCB1 is a part of interrupt INT1, INT1 needs to be enabled. 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 2nd interrupt source of interrupt 1. Below the "IER |=" statements you should see the enabling of TIMER0 which is PIEIER1.bit.INTx7. Do the same line of code but enable PIE interrupt 1.2.
    - Now with everything setup to generate the ADCB1 interrupt, put code in your ADCB1 interrupt function to read the values of ADCINB0, ADCINB1 and ADCINB2, convert these values to units of volts and echo ADCINB0's voltage value to DAC1. ADCINB0 and ADCINB1 have a front end

circuit that allow them to sample a voltage in the range of 10 Volts to -10 Volts. ADCB is a 12 bit ADC so its results have an integer range 0 to 4095. For channels ADCINB0 and ADCINB1, 0 equates to 10 volts, 2048 equates to 0 volts and 4095 equates to -10 volts. Saying this in another way, the 12 bit conversion value, which has a value between 0 and 4095, linearly represents the input voltage range 10.0V to -10.0V.

For ADCINB2, which is connected to the cantilever beam's hall effect sensor, it has an input voltage range of 0 volts to 3.0 volts. So for ADCINB2 only use the scaling where 0 equates to 0.0 volts and 4095 equates to 3.0 volts.

Notice in the below given C code that the converted ADC integer results are stored in the "Results" registers. To store these results create three global int16_t results variables. In addition, create three global float variables to store the scaled voltage value of ADCINB0, ADCINB1 and ADCINB2. Once you have converted these three readings to volts, just write ADCINB0's voltage value to DAC1 using its function setDAC1().

- Set UARTPrint = 1 every 100ms and change the code in the main()'s while loop so that the ADCINB0 voltage value is printed to the text LCD. Make sure to create your own int32_t count variable for this ADCB 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.

```
//Adcb1 pie interrupt
__interrupt void ADCB_ISR (void)
{
   adcb0result = AdcbResultRegs.ADCRESULT0;
   adcb1result = AdcbResultRegs.ADCRESULT1;
   adcb2result = AdcbResultRegs.ADCRESULT2;

   // Here covert ADCINB0, ADCINB1 and ADCINB2 to volts

   // Here write ADCINB0's voltage value to DAC1 channel

   // Print ADCINB0's voltage to the text LCD every 100ms by setting UARTPrint to 1
   // and the function UART_printfLine is called in main()'s while(1) loop

   AdcbRegs.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 F28377S system. The function generator will be connected to ADCINB0 and the oscilloscope will be connected to output DAC1. Input a sine wave with amplitude 16 volts peak to peak and a zero offset. 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, 250Hz, 500Hz, 900Hz, 990Hz, 1000Hz? **Demonstrate to your TA.**

Exercise 2: Have EPWM5 start ADC conversion and use Oversampling

EPWM5 will be used in this exercise just as a timer to signal to ADCB when to convert its ADC channels. EPWM5 was chosen since it 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 1ms. Oversampling can reduce noise in the input signal. We will oversample all three channels ADCINB0, ADCINB1 and ADCINB2 but only echo the value of ADCINB0 to the DAC1 channel. *(Again, sampling ADCINB2, is getting your code ready for sampling the feedback of the cantilever beam experiment in Lab 8 which works the best with this oversampling technique.*)

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 the 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 f28377sADC.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 three lines of code. These lines of code tell the ADCB peripheral to schedule the three 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 F28377S Technical Reference Guide for the explanation of ADCSOC0CTL and find the required value of TRIGSEL. This will cause ADCINB0, ADCINB1 and ADCINB2's conversion whenever the EPWM5 times out. It will trigger them all at the same time so SOC0 will convert first and then SOC1 and then SOC2.

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

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 ADCB 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 ADCB conversion in CPUTimer0's ISR function. Remove the two lines of code you added in Exercise 1, AdcbRegs.ADCSOCPRICTL.bit.RRPOINTER and AdcbRegs.ADCSOCFRC1.all.

3. With these added changes, the ADCB1 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 ADCB1 interrupt function is just going to be used to read the ADCB results registers, convert their value to a voltage between 10.0V and -10.0V, keep a running sum of those voltages and keep track of the number conversions taken. Once 16 samples have been collected, the ADCB1 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 ADCB1 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 sumAdcb0volts = 0;
float sumAdcb1volts = 0;
float sumAdcb2volts = 0;
```

```
int16_t sampleCount = 0; // counts number of times ADC channels have been sampled
float avgAdcb0 = 0;
float avgAdcb1 = 0;
float avgAdcb2 = 0;

//adcb1 pie interrupt
__interrupt void ADCB_ISR (void)
{
  adcb0result = AdcbResultRegs.ADCRESULT0;
  adcb1result = AdcbResultRegs.ADCRESULT1;
  adcb2result = AdcbResultRegs.ADCRESULT2;

  sampleCount++;
  sumAdcb0volts += ????; // equation converting 12bit value to 10 to -10V voltages
  sumAdcb1volts += ????; // equation converting 12bit value to 10 to -10V voltages
  sumAdcb2volts += ????; // equation converting 12bit value to 0 to 3V voltages


  if (sampleCount == 16){
    avgAdcb0 = sumAdcb0volts/16.0;  // oversampled value
    avgAdcb1 = sumAdcb1volts/16.0;  // oversampled value
    avgAdcb2 = sumAdcb2volts/16.0;  // oversampled value

    // Reset variables for next cycle of oversampling
    sampleCount = 0;
    sumAdcb0volts = 0;
    sumAdcb1volts = 0;
    sumAdcb2volts = 0;

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

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

4. The SWI is posted after the 16th ADC conversion and hence the SWI runs every 1 ms (16 x (1/16) ms). Find the SWI_isr function after main() and then inside the SWI function __interrupt void SWI_isr(void) move the code from the ADCB1 interrupt function that writes to the DAC channel and write the ADCINB0 oversampled value to the DAC1. In addition, every 100ms, set UARTPrint to 1 and change your UART_printfLine statement to print your new averaged voltage reading.
5. Build and Run your code. Observe any differences between this response and exercise 1's response on the oscilloscope.

6. Before going on to Exercise 3 you need to make a backup of this C file. This over sampling code you just finished is the starting point for the control code you will develop for controlling the cantilever beam experiment in Lab 8. There is bit of noise in the hall effect feedback signal the cantilever beam experiment uses for position feedback. This 16 point oversample does a pretty good job of filtering this noise. So in Code Composer right click on your main C file and select Copy and call it something like Lab8starterCode.c. Then also make sure to right click on this file and select "Exclude from Build."

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 ADCB. 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, and 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 control will be discretized using "ZOH". The continuous controller will be discretized using an emulation method like Tustin or the backwards rule.*

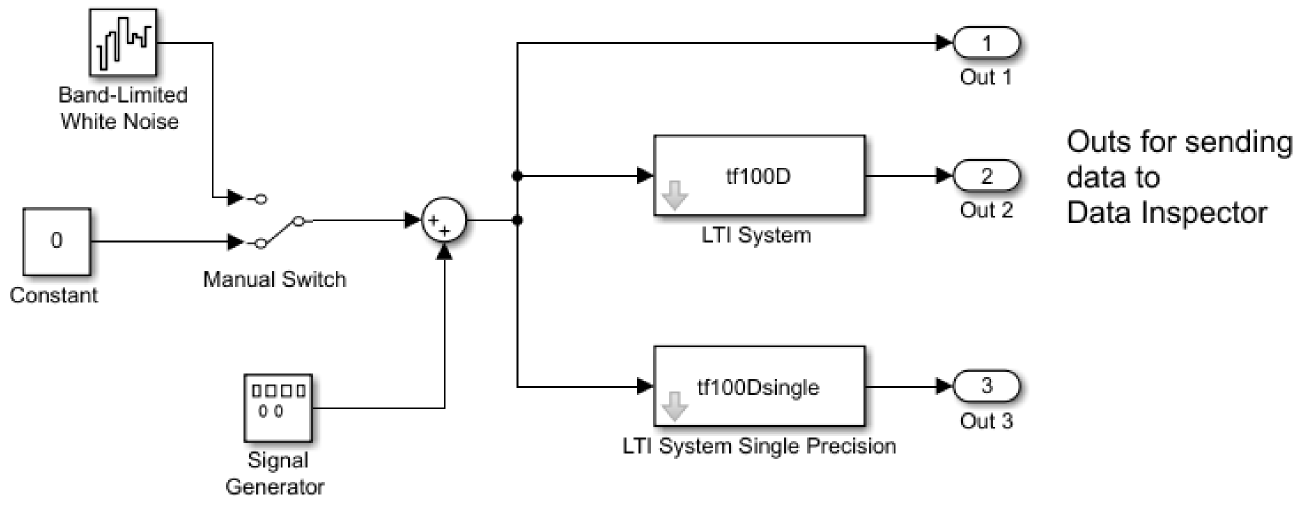
Then compare the Bode plots and step responses of these three discrete transfer functions i.e. step(tf500D,tf100D,tf10D) and bode(tf500D,tf100D,tf10D) . **Show your TA and comment on the difference between the filters.**

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 F28377S 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. 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 ADCINB0 input. Just as in Exercise 1 and 2, bring the function generator’s output into ADCINB0 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. If you recall this code oversampled the ADCB 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 ADCB 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 ADCB channels every 1ms instead of every 1/16 ms. In addition, get rid of taking the average of the 16 samples inside the ADCB1 ISR. Get this code working first, just echoing the ADCINB0 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 SWI_isr apply the filter difference equations on the sample of ADCINB0. Here is some partial code to get your started.

```
xk = adcinb0volt;
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 F28377S 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 ADCINB0 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 ADCINB0
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.