SE423 Laboratory Assignment 4 (Two Week Lab)

Hardware Interrupts, ADC (Analog to Digital Converter), Sampling, Filtering and the FFT (Fast Fourier Transform)

Goals for this Lab Assignment:

1. Create a hardware interrupt (HWI) function associated with the conversion of an ADC sequence
2. Understand sampling of a signal at a discrete interval
3. Demonstrate signal aliasing
4. Sample the analog signals from the robot's LPR510 gyro sensor
5. Design and implement FIR (Finite Impulse Response) filters and demonstrate the filter working
6. Sample an audio signal sensed by the Robot's microphone
7. Use DMA (Direct Memory Access) and TI's FFT (Fast Fourier Transform) library function to find the power spectrum of the robot's microphone signal to detect when a certain tone is heard.

Reading before Lab

1. Read through this entire lab.
2. Find the full chapter 11 discussing the F28379D's ADC peripheral in the TMS320F28379D Technical Reference Guide just in case you need some extra detail. Also in this document, read through sections 3.4 through 3.4.5 which discuss Hardware Interrupt events on the F28379D. Table 3-2 is important when setting up Hardware Interrupts (HWI) so I have created a separate file with that table only.
3. I have created a condensed version of the ADC peripheral that should explain the majority of the topics we need for this lab. ADC Condensed Technical Reference Guide These included sections and register descriptions, should give you an introduction to the ADC peripheral.
4. The EPWM4 peripheral, in this lab, will be used to signal the ADC when to convert instead of driving a duty cycle varying square wave. EPWM Condensed Technical Reference Guide.

Laboratory Exercises

Exercise 1: Using the ADC

For demonstrating more about the hardware interrupt (HWI), we will take advantage of the fact that each of the TMS320F28379D processor's ADC peripherals can generate an interrupt when its sequence of samples have been converted and stored in the results registers of the ADC peripheral. When the ADC conversion is complete, the F28379D will automatically stop the code it is currently processing and jump 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.

There are four ADC peripherals in the F28379D, ADCA, ADCB, ADCC and ADCD. For this first exercise we will use ADCD and its channels ADCIND0, ADCIND1. ADCD is chosen here due to the location of its pins on the F28379D Launchpad (Red Board). We will use additional ADC peripherals in the next exercises and future labs.

Input pins, or channels, ADCIND0 and ADCIND1 are brought through a multiplexer inside the F28379D processor into the ADCD peripheral. The ADCD peripheral can only sample one of these voltage input pins at a time. So the multiplexer allows the ADCD sequencer to sample one channel and then when finished, the sequencer can change the multiplexer so that the next channel can be converted. The sequencer can schedule up to 16 conversions each trigger. We are only going to setup two conversions, ADCIND0 and ADCIND1. We actually are only going to use ADCIND0 in this lab, but I am having you also sample ADCIND1 to show you how multiple ADC channels (ADC input pins) can be sampled with one event.

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 on the ADC input pin. 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 interpolation 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 and ADCIND1 in sequence. We will setup Start of Conversion 0 (SOC0) to convert ADCIND0 and SOC1 to covert ADCIND1. When SOC1 is finished converting the input voltage to its corresponding 12 bit value, hardware interrupt ADCD1 will be flagged for execution. You will write the hardware interrupt function that is called when ADCD1 is flagged. The conversion results will be stored in registers AdcdResultRegs.ADCRESULT0 and AdcdResultRegs.ADCRESULT1.

Your first task will be to build an application that samples ADC channels ADCIND0 and ADCIND1 echoes ADCIND0's voltage value to the F28379D's DACA voltage output. There are quite a number of things to setup so I am giving you most of the code. You will need refer to the register descriptions to fill in the blanks. I also give you some commented out code that will help you in future exercises and labs to setup the other ADC peripherals.

1. Create a new project with LABstarter and rename in Lab4<yourinitials>. Then rename the project's main file accordingly.
2. We would like to command the ADCD peripheral to sample ADCIND0 and ADCIND1 every 1ms. There are a few ways to accomplish this, but for this lab we are going to use EPWM4 as just a timer (no duty cycle output) to trigger the ADCD conversion sequence. In main() after the init_serial function, add the following code and fill in the ??? blanks by studying the EPWM reference. Don't forget to copy EALLOW and EDIS.

```
EALLOW;
EPwm4Regs.ETSEL.bit.SOCAEN = 0; // Disable SOC on A group
EPwm4Regs.TBCTL.bit.CTRMODE = 3; // freeze counter
EPwm4Regs.ETSEL.bit.SOCASEL = ???; // Select Event when counter equal to PRD
EPwm4Regs.ETPS.bit.SOCAPRD = ???; // Generate pulse on 1st event (“pulse” is the same as “trigger”)
EPwm4Regs.TBCTR = 0x0; // Clear counter
EPwm4Regs.TBPHS.bit.TBPHS = 0x0000; // Phase is 0
EPwm4Regs.TBCTL.bit.PHSEN = 0; // Disable phase loading
EPwm4Regs.TBCTL.bit.CLKDIV = 0; // divide by 1  50Mhz Clock
EPwm4Regs.TBPRD = ???;  // Set Period to 1ms sample.  Input clock is 50MHz.
// Notice here that we are not setting CMPA or CMPB because we are not using the PWM signal
EPwm4Regs.ETSEL.bit.SOCAEN = 1; //enable SOCA
EPwm4Regs.TBCTL.bit.CTRMODE = ???; //unfreeze, and enter up count mode
EDIS;
```

3. Next, we would like to setup ADCD so that it uses 2 of its 16 SOCs (Start of Conversions). These SOCs are used by the ADC sequencer to sample the ADC channels in any desired order and if necessary, assign priority to certain SOCs. It can be very confusing for the first time user of this peripheral, but if we stick to the basics it is pretty straightforward. We are going to trigger SOC0 and SOC1 with the same EPWM PRD time event. When triggered, SOC0 has the higher priority so it will sample/convert its channel first and when complete SOC1 will sample/convert its channel. We will assign channel ADCIND0 to SOC0 and channel ADCIND1 to SOC1. These below initializations also setup ADCD to flag an interrupt when SOC1 is finished converting. This makes sense since we first want SOC0 to convert channel ADCIND0 and then the SOC1 to convert channel ADCIND1. This is the default order for the round robin sequencer. Some of the given code I found in an ADC example from TI. It retrieves the default calibration values for the ADCD in the F28379D’s read only memory (ROM). Add the following to main() after the “init_serial() function calls and filling in the appropriate blanks (???). Comment below should be very helpful.

```
EALLOW;
//write configurations for all ADCs  ADCA, ADCB, ADCC, ADCD
AdcaRegs.ADCCTL2.bit.PRESCALE = 6; //set ADCCLK divider to /4
AdcbRegs.ADCCTL2.bit.PRESCALE = 6; //set ADCCLK divider to /4
AdccRegs.ADCCTL2.bit.PRESCALE = 6; //set ADCCLK divider to /4
AdcdRegs.ADCCTL2.bit.PRESCALE = 6; //set ADCCLK divider to /4
AdcSetMode(ADC_ADCA, ADC_RESOLUTION_12BIT, ADC_SIGNALMODE_SINGLE);  //read calibration settings
AdcSetMode(ADC_ADCB, ADC_RESOLUTION_12BIT, ADC_SIGNALMODE_SINGLE);  //read calibration settings
AdcSetMode(ADC_ADCC, ADC_RESOLUTION_12BIT, ADC_SIGNALMODE_SINGLE);  //read calibration settings
AdcSetMode(ADC_ADCD, ADC_RESOLUTION_12BIT, ADC_SIGNALMODE_SINGLE);  //read calibration settings
//Set pulse positions to late
AdcaRegs.ADCCTL1.bit.INTPULSEPOS = 1;
AdcbRegs.ADCCTL1.bit.INTPULSEPOS = 1;
AdccRegs.ADCCTL1.bit.INTPULSEPOS = 1;
AdcdRegs.ADCCTL1.bit.INTPULSEPOS = 1;
//power up the ADCs
```

```
AdcaRegs.ADCCTL1.bit.ADCPWDNZ = 1;
AdcbRegs.ADCCTL1.bit.ADCPWDNZ = 1;
AdccRegs.ADCCTL1.bit.ADCPWDNZ = 1;
AdcdRegs.ADCCTL1.bit.ADCPWDNZ = 1;
//delay for 1ms to allow ADC time to power up
DELAY_US(1000);

//Select the channels to convert and end of conversion flag
//Many statements commented out,  To be used when using ADCA or ADCB
//ADCA
//AdcaRegs.ADCSOC0CTL.bit.CHSEL = ???;  //SOC0 will convert Channel you choose Does not have to be A0
//AdcaRegs.ADCSOC0CTL.bit.ACQPS = 99; //sample window is acqps + 1 SYSCLK cycles = 500ns
//AdcaRegs.ADCSOC0CTL.bit.TRIGSEL = ???;// EPWM4 ADCSOCA or another trigger you choose will trigger SOC0
//AdcaRegs.ADCSOC1CTL.bit.CHSEL = ???;  //SOC1 will convert Channel you choose Does not have to be A1
//AdcaRegs.ADCSOC1CTL.bit.ACQPS = 99; //sample window is acqps + 1 SYSCLK cycles = 500ns
//AdcaRegs.ADCSOC1CTL.bit.TRIGSEL = ???;// EPWM4 ADCSOCA or another trigger you choose will trigger SOC1
//AdcaRegs.ADCINTSEL1N2.bit.INT1SEL = ???; //set to last SOC that is converted and it will set INT1 flag ADCA1
//AdcaRegs.ADCINTSEL1N2.bit.INT1E = 1;   //enable INT1 flag
//AdcaRegs.ADCINTFLGCLR.bit.ADCINT1 = 1; //make sure INT1 flag is cleared

//ADCB  Microphone is connected to ADCINB4
//AdcbRegs.ADCSOC0CTL.bit.CHSEL = ???;  //SOC0 will convert Channel you choose Does not have to be B0
//AdcbRegs.ADCSOC0CTL.bit.ACQPS = 99; //sample window is acqps + 1 SYSCLK cycles = 500ns
//AdcbRegs.ADCSOC0CTL.bit.TRIGSEL = ???; // EPWM7 ADCSOCA or another trigger you choose will trigger SOC0
//AdcbRegs.ADCSOC1CTL.bit.CHSEL = ???;  //SOC1 will convert Channel you choose Does not have to be B1
//AdcbRegs.ADCSOC1CTL.bit.ACQPS = 99; //sample window is acqps + 1 SYSCLK cycles = 500ns
//AdcbRegs.ADCSOC1CTL.bit.TRIGSEL = ???; // EPWM7 ADCSOCA or another trigger you choose will trigger SOC1
//AdcbRegs.ADCINTSEL1N2.bit.INT1SEL = ???; //set to last SOC that is converted and it will set INT1 flag ADCB1
//AdcbRegs.ADCINTSEL1N2.bit.INT1E = 1;   //enable INT1 flag
//AdcbRegs.ADCINTFLGCLR.bit.ADCINT1 = 1; //make sure INT1 flag is cleared

//ADCC
//AdccRegs.ADCSOC0CTL.bit.CHSEL = ???;  //SOC0 will convert Channel you choose Does not have to be B0
//AdccRegs.ADCSOC0CTL.bit.ACQPS = 99; //sample window is acqps + 1 SYSCLK cycles = 500ns
//AdccRegs.ADCSOC0CTL.bit.TRIGSEL = ???; // EPWM4 ADCSOCA or another trigger you choose will trigger SOC0
//AdccRegs.ADCSOC1CTL.bit.CHSEL = ???;  //SOC1 will convert Channel you choose Does not have to be B1
//AdccRegs.ADCSOC1CTL.bit.ACQPS = 99; //sample window is acqps + 1 SYSCLK cycles = 500ns
//AdccRegs.ADCSOC1CTL.bit.TRIGSEL = ???; // EPWM4 ADCSOCA or another trigger you choose will trigger SOC1
//AdccRegs.ADCSOC2CTL.bit.CHSEL = ???;  //SOC2 will convert Channel you choose Does not have to be B2
//AdccRegs.ADCSOC2CTL.bit.ACQPS = 99; //sample window is acqps + 1 SYSCLK cycles = 500ns
//AdccRegs.ADCSOC2CTL.bit.TRIGSEL = ???; // EPWM4 ADCSOCA or another trigger you choose will trigger SOC2
//AdccRegs.ADCSOC3CTL.bit.CHSEL = ???;  //SOC3 will convert Channel you choose Does not have to be B3
//AdccRegs.ADCSOC3CTL.bit.ACQPS = 99; //sample window is acqps + 1 SYSCLK cycles = 500ns
//AdccRegs.ADCSOC3CTL.bit.TRIGSEL = ???; // EPWM4 ADCSOCA or another trigger you choose will trigger SOC3
//AdccRegs.ADCINTSEL1N2.bit.INT1SEL = ???; //set to last SOC that is converted and it will set INT1 flag ADCB1
//AdccRegs.ADCINTSEL1N2.bit.INT1E = 1;   //enable INT1 flag
//AdccRegs.ADCINTFLGCLR.bit.ADCINT1 = 1; //make sure INT1 flag is cleared

//ADCD
AdcdRegs.ADCSOC0CTL.bit.CHSEL = ???;  // set SOC0 to convert pin D0
```

```
AdcdRegs.ADCSOC0CTL.bit.ACQPS = 99; //sample window is acqps + 1 SYSCLK cycles = 500ns
AdcdRegs.ADCSOC0CTL.bit.TRIGSEL = ???; // EPWM4 ADCSOCA will trigger SOC0
AdcdRegs.ADCSOC1CTL.bit.CHSEL = ???;  //set SOC1 to convert pin D1
AdcdRegs.ADCSOC1CTL.bit.ACQPS = 99; //sample window is acqps + 1 SYSCLK cycles = 500ns
AdcdRegs.ADCSOC1CTL.bit.TRIGSEL = ???; // EPWM4 ADCSOCA will trigger SOC1
//AdcdRegs.ADCSOC2CTL.bit.CHSEL = ???;  //set SOC2 to convert pin D2
//AdcdRegs.ADCSOC2CTL.bit.ACQPS = 99; //sample window is acqps + 1 SYSCLK cycles = 500ns
//AdcdRegs.ADCSOC2CTL.bit.TRIGSEL = ???; // EPWM4 ADCSOCA will trigger SOC2
//AdcdRegs.ADCSOC3CTL.bit.CHSEL = ???;  //set SOC3 to convert pin D3
//AdcdRegs.ADCSOC3CTL.bit.ACQPS = 99; //sample window is acqps + 1 SYSCLK cycles = 500ns
//AdcdRegs.ADCSOC3CTL.bit.TRIGSEL = ???; // EPWM4 ADCSOCA will trigger SOC3
AdcdRegs.ADCINTSEL1N2.bit.INT1SEL = ???; //set to SOC1, the last converted, and it will set INT1 flag ADCD1
AdcdRegs.ADCINTSEL1N2.bit.INT1E = 1;   //enable INT1 flag
AdcdRegs.ADCINTFLGCLR.bit.ADCINT1 = 1; //make sure INT1 flag is cleared
EDIS;
```

4. The initializations for DACA and DACB output are much simpler then the ADC. When setup as DAC outputs pins ADCINA0 and ADCINA1 are DACA and DACB respectively. Note: In this lab, we will only be using DACA but we will setup DACB just in case you need it later in the class. See Pinmux Table for pin location of DACA and DACB. In main() initialize the DACs with these lines of code:

```
// Enable DACA and DACB outputs
EALLOW;
DacaRegs.DACOUTEN.bit.DACOUTEN = 1;//enable dacA output-->uses ADCINA0
DacaRegs.DACCTL.bit.LOADMODE = 0;//load on next sysclk
DacaRegs.DACCTL.bit.DACREFSEL = 1;//use ADC VREF as reference voltage

DacbRegs.DACOUTEN.bit.DACOUTEN = 1;//enable dacB output-->uses ADCINA1
DacbRegs.DACCTL.bit.LOADMODE = 0;//load on next sysclk
DacbRegs.DACCTL.bit.DACREFSEL = 1;//use ADC VREF as reference voltage
EDIS;
```

While we are thinking of the DAC output, let's quickly write to the DAC output function setDACA(float volt). It takes a float parameter that is a voltage value between 0.0V to 3.0V. The DAC register is 16 bits but only the bottom 12 bits are used since it is a 12 bit DAC. Therefore, the DAC register is looking for a value between 0 and 4095, where 0 is 0.0V and 4095 is 3.0V. Copy these functions somewhere above main() and fill in the scaling ??? that converts the voltage output value to a 12bit value.

```
//This function sets DACA to the voltage between 0V and 3V passed to this function.
//If outside 0V to 3V the output is saturated at 0V to 3V
//Example code
//float myu = 2.25;
//setDACA(myu);   // DACA will now output 2.25 Volts
void setDACA(float dacouta0) {
    if (dacouta0 >  3.0) dacouta0 =  3.0;
    if (dacouta0 < 0.0) dacouta0 = 0.0;

    DacaRegs.DACVALS.bit.DACVALS = ???;  // perform scaling of 0-3 to 0-4095
```

```
}

void setDACB(float dacouta1) {
    if (dacouta1 >  3.0) dacouta1 =  3.0;
    if (dacouta1 < 0.0) dacouta1 = 0.0;

    DacbRegs.DACVALS.bit.DACVALS = ???;  // perform scaling of 0-3 to 0-4095
}
```

5. Add your ADCD1 hardware interrupt function. **A shell function definition is given below**. *Note: The naming can get confusing here. There are ADCD inputs channels labeled ADCIND0, ADCIND1, ADCIND2, ADCIND3, etc and there are also four ADCD interrupts labeled ADCD1, ADCD2, ADCD3 and ADCD4. Above in item 2 we setup ADCD1 interrupt to be called when two channels ADCIND0 and ADCIND1 are finished converting.* You will be adding most of the remainder of your 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) {` (See shell below)
    - 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. 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. ADCIND0 and ADCIND1 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 from 0.0V to 3.0V. The 12 bit

conversion values are stored in the results registers, see given code below. Create two global int16_t result variables to store ADCIND0 and ADCIND1's raw reading. Create one global float variable to store the scaled (0V-3V) voltage value of ADCIND0. Once you have converted the ADCIND0 12 bit integer result value to voltage, pass that variable to the setDACA() function to echo the sampled voltage back out to DACA.

- Set UARTPrint = 1 every 100ms and change the code in the main() while loop so that your ADC voltage value is printed to TeraTerm and the robot's text LCD screen. Make sure to create your own int32_t count variable for this ADCD1 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;

  // Here covert ADCIND0 to volts

  // Here write voltages value to DACA

  // Print ADCIND0's voltage value to TeraTerm every 100ms by setting UARTPrint =1

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

}
```

6. 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 robot's breakout board. Connect the function generator's output first to the oscilloscope's CH1 and then to ADCIND0's input which is labeled "ADC Scope" on the break out board. Also connect DACA's output, which is labeled "DAC Scope" on the break out board, to the oscilloscope's CH2. 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 (SHIFT MENU > > > V V > ENTER). 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 and 990Hz? **Demo to your TA.**

Exercise 2: Discrete Filters

One big problem with analog signals is that electrical noise can be present in the signal. If the noise is large it can cause the sensed signal to be very poor for control and other applications. One way to handle this noise is to implement filtering algorithms to filter the noisy signal, but always remember that this causes phase lag and if too much phase lag is added your control system becomes unstable. For this lab we are going to design our filters in Matlab using the FIR1 function. There are many other functions in Matlab to design filters but we will just focus on FIR1. Before we get into the Matlab FIR designs, I want to implement the simplest FIR filter. A simple averaging filter. See the below code implementing a "5 tap" or 5 sample averaging filter.

```
//xk is the current ADC reading, xk_1 is the ADC reading one millisecond ago, xk_2 two milliseconds ago, etc
float xk = 0;
float xk_1 = 0;
float xk_2 = 0;
float xk_3 = 0;
float xk_4 = 0;
//yk is the filtered value
float yk = 0;
//b is the filter coefficients
float b[5] = {0.2,0.2,0.2,0.2,0.2};  // 0.2 is 1/5th therefore a 5 point average
//adcd1 pie interrupt
__interrupt void ADCD_ISR (void)
{
    adcd0result = AdcdResultRegs.ADCRESULT0;
    adcd1result = AdcdResultRegs.ADCRESULT1;
    // Here covert ADCIND0, ADCIND1 to volts
    xk = ADCIND0 voltage reading;
    yk = b[0]*xk + b[1]*xk_1 + b[2]*xk_2 + b[3]*xk_3 + b[4]*xk_4;

    //Save past states before exiting from the function so that next sample they are the older state
    xk_4 = xk_3;
    xk_3 = xk_2;
    xk_2 = xk_1;
    xk_1 = xk;

    // Here write yk to DACA channel

    // Print xk and yk's voltage value to TeraTerm every 100ms

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

Implement this code/filter. With this filter running what happens to the output signal when you increase the frequency of the input sine wave from the function generator? Does it match the Bode plot of the filter, type in Matlab "freqz([.2,.2,.2,.2,.2])"? In this bode plot the x axis is frequency scale such that 1 is 500Hz. Also answer the question why the filter equations above would not have worked properly if the "Save Past States" statements would have been written in this order:

```
xk_1 = xk;
xk_2 = xk_1;
xk_3 = xk_2;
xk_4 = xk_3;
```

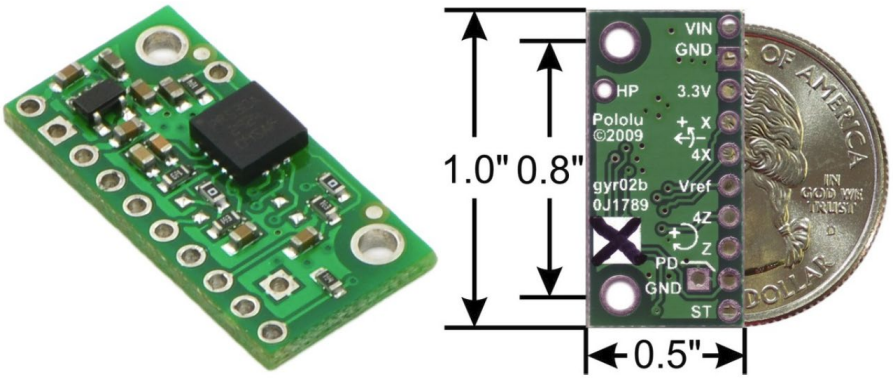
Tell Answer to Your TA

Now let's use Matlab to design a bit better low-pass filter using the FIR1 function. In Matlab type b = fir1(4,.1) This designs a 4th order FIR (Finite Impulse Response) low-pass filter with cutoff frequency .1 of the Nyquist frequency. Remember that the Nyquist frequency is half the sample frequency. So here the sample frequency is 1000Hz so Nyquist is 500Hz. Therefore, the cutoff frequency of this filter is 50Hz. So if we apply to this filter input frequencies from the function generator greater than 50 Hz the output amplitude will be attenuated (reduced) when compared to the input amplitude. To see how much attenuation, you can plot the bode plot using the Matlab function "freqz(b)". In this bode plot the x axis is frequency scale such that 1 is Nyquist frequency. Implement this filter. Since this filter is the same order as your average filter already implemented, all you have to do is change the b coefficients. To copy the b coefficients to your C code use the function "arraytoCformat" in Matlab. This function is already on the PCs in the lab. If you would like to use it on your own PC download it here. Type arraytoCformat(b') and the coefficients will be printed out in a C array statement. Copy this into your C file to use instead of the 5 0.2 coefficients. Run this code and see if you think this filter works better than the average filter. **Show your TA.**

Now what if I ask you design a 21st order FIR filter with 75 Hz cutoff frequency? Or what if I ask for a 100th order filter? You would not want to type in all those past states xk_100, xk_99, xk_98, xk_97, etc. Instead, if your past states were an array, you could just change the size of the array and maybe a for loop length and your filter code would work after you copy the new b array from Matlab. So change your filter code so that the xk states are part of an array. First do this with your 4th order filter. When that is working implement a 21st order low-pass FIR filter with 75 Hz cutoff frequency. How does the 21st order filter compare to the 4th order filter? Does the 21st order filter match its Bode plot? **Show your TA.**

Exercise 3: Sample the two axes of the LPR510 Gyro.

In this exercise I would like you to sample the two axes of the Robot's LPR510 gyro. This gyro produces analog output signals in proportion to the degrees/second that the robot is rotating about two axes. Each axis has two outputs, one 4 times the resolution of the other. Which resolution to use depends on the rate of turn you need to measure. We will measure both resolutions to have both possibilities.



4Z(ADCINC4) and 4X(ADCINC3) (4 here means 4 times the accuracy) Voltage Range ≈0.23V -> -100°/s, ≈1.23V -> 0°/s, ≈2.23V -> 100°/s

Z(ADCINC5) and X(ADCINC2) Voltage Range ≈0.23V -> -400°/s, ≈1.23V -> 0°/s, ≈2.23V -> 400°/s

Looking at the Pinmux table for the robot's break out board, you will find that channels ADCINC2, ADCINC3, ADCINC4 and ADCINC5 are connected to the four signals from the LPR510. Setup ADCC to sample these four channels every 1ms. Since EPWM4 is already set to trigger every 1ms you can use EPWM4 to also trigger ADCC.

1. Find in the code you already copied into main that setup the ADC peripherals, the commented out lines that setup ADCC. Uncomment these lines and set this code so SOC0 samples ADCINC2 and triggered by EPWM4, SOC1 samples ADCINC3 and triggered by EPWM4, SOC2 samples ADCINC4 and triggered by EPWM4 and SOC3 samples ADCINC5 and triggered by EPWM4. Finally make sure that the ADCC1 interrupt occurs when SOC3 is done converting.
2. Find in the interrupt sources table 3-2 the ADCC1 PIE interrupt and enable that interrupt i.e. PieCtrlRegs.PIEIER1.????? = 1
3. Write the Interrupt Service Routine (ISR) for ADCC. Inside this function read the four result registers and convert the values that are between 0 and 4095 (12bit) to 0V to 3V. Finally using the scaling given in the above picture of the LPR510, convert voltage to degrees/second. Print these rate values every 100ms by setting UARTPrint to 1 and changing what prints in main()'s while loop serial_printf function call. Also at the end of this ISR function clear the ADC interrupt flag using AdccRegs and

acknowledge PIEACK_GROUP1. Don't forget to create a predefinition of your ISR function toward the top of your C file.

4. Set the PieVectTable ADCC1_INT element to the address of your ADCC ISR function i.e. &ADCC_ISR.

Compile and run this code and see if the printed angular rates make sense. **Demo this to your TA.** You will probably notice that with the robot at rest, the turn rate is not exactly zero. This is due to 1.23V not being the exact zero voltage that is subtracted from your voltage readings. This is normal for gyro sensors and this zero value changes slightly due to temperature or other reasons. So I want you to add a little more code to your ISR function. Create a global int32_t numADCCcalls variable to count (add 1) the number of times your ADCC ISR function has been called. Put this numADCCcalls++ at the bottom of your function. Then using this count variable, create an if, else if, else if, else statement that for the first 1 second (1000 calls) does nothing with the ADCcresult readings allowing the zero of the gyro to settle after a power on. Then for the next 2 seconds sum each voltage reading into a float variable i.e. sum4Z, sumZ, sum4X, sumX. After 2 seconds of summing up 2000 samples divide the sum variables by 2000 to find a good value for each output's zero value. Finally after all the steps, in the else part of this code, use the zero values just found to subtract that value from each measurement from that point forward. Once in the else, keep your code that takes the new readings with the calculated zero values and scales the turn rates to degrees/second. (See the text under the gyro picture above for scaling to degrees/seconds.) Print these new values every 100ms. Are the readings now closer to 0 degrees/second when the robot is still?

Demo this to your TA.

Exercise 4: Sample the Microphone's signal

Setup ADCB to sample the microphone's audio signal at a rate of 0.1ms. and echo its value on DACA. ADCINB4 is connected to the audio signal from the microphone board. The microphone's output voltage ranges from approximately 0.5V to 2.5V so it is inside the ADC range of 0V to 3.0V. Setup ADCB similar to ADCD in exercise 1 but here you will only need SOC0 since there is only one microphone signal and it is connected to ADCINB4. Also we would like to sample the microphone 10 times faster at a sample period of 0.1ms. Setup EPWM7 to be the trigger for ADCB, triggering ADCB every 0.1ms. You will have to find out the interrupt number for ADCB1 and enable it similar to how you enabled the ADCD1 interrupt. Inside the interrupt function also write the sampled voltage to DACA. You will need to comment out your writing to DACA in your ADCD1 interrupt function. Scope the DACA's output to see what your voice looks like in analog voltage. Whistle or have your phone play some tones to see sinewave outputs from the microphone. **Demo to your TA.**

Exercise 5: Load the FFT starter code that uses DMA to store a block of ADC conversions and then the FFT library is used to produce the Power Spectrum of the microphone signal.

First open Matlab and run the below script in an M-file to familiarize yourself a bit with the FFT operator and power spectrum plots.

```
t = 0:.001:1.023;
X = sin(2*pi*60*t');
Y = fft(X,1024);
Pyy = Y.*conj(Y)/1024;
f = 1000/1024*(0:511);
figure(1)
plot(f,Pyy(1:512))
title('Power spectral density')
xlabel('Frequency (Hz)')
figure(2)
plot(t,X)
```

Then change the X = line to X = sin(2*pi*60*t')+sin(2*pi*29*t'); and run the script

Then change to X = sin(2*pi*250*t')+sin(2*pi*300*t'); and run the script

Finally change to X = sin(2*pi*250*t')+sin(2*pi*300*t') + 1.5; and run the script

Talk to your TA about the differences.

For this exercise, you will start a new project but this time you will import the CCS project "fftDMAstarter". Go ahead and import this new project starter. Then rename the project to "fftDMAlab4<yourinitials>". Debug and Run this code. This code is using DMA (Direct Memory Access) to transfer each ADC sample to an array at a 10000Hz rate. Once 1024 samples have been stored by the DMA, an interrupt occurs. That interrupt issues a command for the FFT to be calculated with these ADC samples and produce the power spectrum of the signal. Given the power spectrum, a for loop loops over the spectrum data and finds the highest peak. If a loud whistle or some other generated tone is being played, that highest peak relates to the tone's frequency. While running this code open Tera Term and watch the print outs of maximum power and frequency. Whistle or play a number of tones and see the frequency changing. **Show the code working to your TA.** Take a look through this code and find the code where the ADC samples are first being stored in a Ping array and then in a Pong array. The reason for this Ping and Pong array is that the FFT takes about 0.6ms to run and we are sampling every 0.1ms. If the Ping Pong arrays were not used some ADC samples would be ignored. Also as you are looking through the code notice that I add comment instructions how to cut and paste the need code from this file into another project's code. As we get closer to the final project for this class you will be merging your

code from multiple projects to create your final code. **Show your TA the Ping Pong code and sections that you would need to copy into another project.**

Lab Check Off:

1. Demonstrate your sampled signal and signal aliasing.
2. Demonstrate your implemented 4th order averaging filter.
3. Demonstrate your implemented 4th order FIR 50Hz cutoff filter.
4. Demonstrate your implemented 21st order FIR 75Hz cutoff filter.
5. Demonstrate the four LPR510 gyro readings displaying to the text LCD.
6. Demonstrate your Microphone signal echoed to DACA every .1ms.
7. Demonstrate the FFT code running and your understanding of the given FFT code.
8. Submit all your written code after adding comments explaining your code and what you learned. Also be clear what code is for what exercise.
9. Submit your HowTo document with at minimum the following items:
    a. Using EPWM8 (Note: NOT EPWM4 like in the lab) and ADCA, write initialization code that has EPWM8 trigger ADCA's SOCs (Start of Conversions) every 0.125 milliseconds (0.000125 seconds). Also have this initialization code setup ADCA so that every 0.125 milliseconds ADCA channels ADCINA2, ADCINA3, ADCINA4 and ADCINA5 are sampled. Finally make sure that after the last SOC has converted that ADCA's ADCA1 interrupt is called.

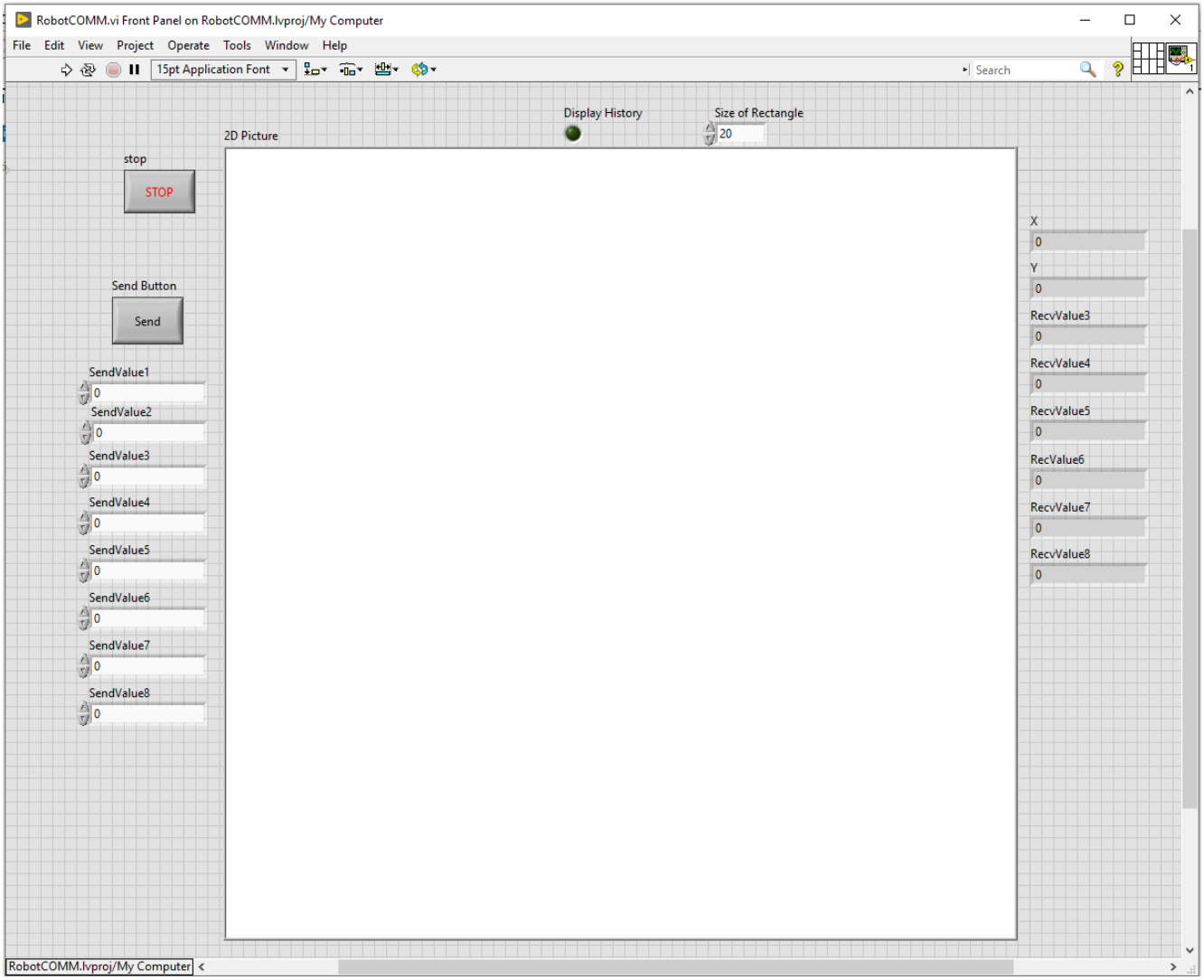
Exercise to Finish before Lab 6:

The goal of this assignment is to get you up to speed faster using LABVIEW to program a host interface to your robot. This quick example shows you how to both send data from the LabVIEW to the Raspberry Pi4 and then to the F28379D and receive data from the PI4/F28379D at the PC/LabVIEEW. It also shows you how to move an object inside a 2D picture item.

You will just be only creating the LabVIEW block diagram given below. You will not be able to test this LabVIEW code until the end of Lab 6. Getting this LabVIEW code put together before Lab 6 will help you finish Lab 6. Using this LabVIEW code in Lab 6, you will be able to type values into the numeric controls and click Send and the values will be sent first to the PI and then to your F28379D processor. Also every 250ms the F28379D will send values to the PI and then the PI will send the values to this LABVIEW application. The values will be displayed in two indicators and a square inside the picture box will move to the x, y coordinates. As the X, Y values received change, you will see the square move on the screen which is going to indicate the location

of your robot. Some of the code is similar to your LabView #2 assignment and the LabView code in Lab 2.

NOTE!!!: The pink string constants that have "\FD" and "\FF", need to be changed, by right clicking on them, to "\ Code Display". This allows the Hex numbers 0xFD and 0xFF to be sent to the PI, start and stop characters.



1. Open up LABVIEW, create a blank project and add a single VI to the project. You may want to save this project and VI in your personal repository.
2. Add a 2D Picture to your front panel and make its dimensions 800 by 800. Later we will be equating 50 pixels to 1 tile. This will then make your course 16 tiles by 16 tiles.
3. Reproduce the following block diagram along with the front panel above. It prints a square at the coordinates sent from Linux.