

# ME 461 Laboratory #4 (Two Week Lab)

## Hardware Interrupts, Analog to Digital Converters (ADCs), and Sampling

### Goals

---

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 an audio signal sensed by a microphone.
5. Design and implement a Finite Impulse Response (FIR) low-pass filter and band-pass filter.

### Prelab Readings

---

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 EPWM5 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](#).

### 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 equalling 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 (or very close to) is on the ADC input pin. Hence, there is a linear interpolation between 0 and 4095 covering all the voltages from 0.0 to almost 3.0 volts with steps of  $3.0V/4096 = .73mV$ . (*4096 instead of 4095 due to how an ADC is designed.*)

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 convert 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. You will be using the 3D printed robot car for this exercise. There are quite a number of things to set up, so I am giving you most of the code. You will need to 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 set up the other ADC peripherals.

1. 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 EPWM5 as just a timer (no duty cycle output) to trigger the ADCD conversion sequence. In `main()`, after

the `init_serialSCIA` function, add the following code and fill in the `???` blanks by reading each line's comments and studying the EPWM reference. Don't forget to copy `EALLOW` and `EDIS`.

```
EALLOW;
EPwm5Regs.ETSEL.bit.SOCAEN = 0; // Disable SOC on A group
EPwm5Regs.TBCTL.bit.CTRMODE = 3; // freeze counter
EPwm5Regs.ETSEL.bit.SOCASEL = ???; // Select Event when counter equal to PRD
EPwm5Regs.ETPS.bit.SOCAPRD = ???; // Generate pulse on 1st event ("pulse" is the same as
"trigger")
EPwm5Regs.TBCTR = 0x0; // Clear counter
EPwm5Regs.TBPHS.bit.TBPHS = 0x0000; // Phase is 0
EPwm5Regs.TBCTL.bit.PHSEN = 0; // Disable phase loading
EPwm5Regs.TBCTL.bit.CLKDIV = 0; // divide by 1 50Mhz Clock
EPwm5Regs.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
EPwm5Regs.ETSEL.bit.SOCAEN = 1; //enable SOCA
EPwm5Regs.TBCTL.bit.CTRMODE = ???; //unfreeze, and enter up count mode
EDIS;
```

- Next, we would like to set up ADCD so that it uses 2 of its 16 SOC's (Start of Conversions). These SOC's are used by the ADC sequencer to sample the ADC channels in any desired order and if necessary, assign priority to certain SOC's. It can be very confusing for a 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. 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 is in an ADC example I found. It retrieves the default calibration values for the ADCD in the F28379D's read only memory (ROM). Add the following to `main()`, filling in the appropriate `???` blanks. The comments in the following code 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 = ???; // EPWM5 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 = ???; // EPWM5 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
//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 = ???; // EPWM5 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 = ???; // EPWM5 ADCSOCA or another trigger you choose will
trigger SOC1
//AdcbRegs.ADCSOC2CTL.bit.CHSEL = ???; //SOC2 will convert Channel you choose Does not have to
be B2
//AdcbRegs.ADCSOC2CTL.bit.ACQPS = 99; //sample window is acqps + 1 SYSCLK cycles = 500ns
//AdcbRegs.ADCSOC2CTL.bit.TRIGSEL = ???; // EPWM5 ADCSOCA or another trigger you choose will
trigger SOC2
//AdcbRegs.ADCSOC3CTL.bit.CHSEL = ???; //SOC3 will convert Channel you choose Does not have to
be B3
//AdcbRegs.ADCSOC3CTL.bit.ACQPS = 99; //sample window is acqps + 1 SYSCLK cycles = 500ns
//AdcbRegs.ADCSOC3CTL.bit.TRIGSEL = ???; // EPWM5 ADCSOCA or another trigger you choose will
trigger SOC3
//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

//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 = ???; // EPWM5 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 = ???; // EPWM5 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 = ???; // EPWM5 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 = ???; // EPWM5 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;

```

- The initializations for DACA and DACB output are much simpler than 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 for your final project.* 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 almost 3.0V. *Remember the resolution is 3.0/4096 due to how the DAC is designed.*

Copy these functions somewhere above `main()` and fill in the scaling.

```

//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) {
    int16_t DACOutInt = 0;
    DACOutInt = ???; // perform scaling of 0 - almost 3V to 0 - 4095
    if (DACOutInt > 4095) DACOutInt = 4095;
    if (DACOutInt < 0) DACOutInt = 0;

    DacaRegs.DACVALS.bit.DACVALS = DACOutInt;
}

void setDACB(float dacouta1) {
    int16_t DACOutInt = 0;
    DACOutInt = ???; // perform scaling of 0 - almost 3V to 0 - 4095
    if (DACOutInt > 4095) DACOutInt = 4095;
    if (DACOutInt < 0) DACOutInt = 0;

    DacbRegs.DACVALS.bit.DACVALS = DACOutInt;
}

```

4. 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:*

- a. Starting with this shell code, create your interrupt function as a global function with `void` parameters and of type `__interrupt void`. For example: `__interrupt void ADCD_ISR(void)`

```

//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

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

```

- b. 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.
- c. 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` (if you kept the same name as the shell function).
- d. 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 need to enable the 6<sup>th</sup> 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.
- e. 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 almost 3.0V. The 12-bit conversion values are stored in the results registers. 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 a voltage, pass that variable to the `setDACA()` function to echo the sampled voltage back out to DACA.

- f. Set `UARTPrint = 1` every 100ms and change the code in the `main()` while loop so that your ADC voltage value is printed to TeraTerm. 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 or other interrupt function.
  - g. 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 shell code has already code for this step.
5. 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. 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 labelled "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 → → → ↓ ↓ → 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

---

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. 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" 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 ADCIND0 and ADCIND1's voltage value to TeraTerm every 100ms

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

```

Implement this filter. With this filter running, what happens when you increase the frequency of the of the input sine wave from the function generator? 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 another low-pass filter using the `FIR1` function. In MATLAB, type `b = fir1(4, .1)`. This designs a 4<sup>th</sup> 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. If we apply to this filter input frequencies from the function generator greater than 50Hz, the output amplitude will be attenuated (reduced) when compared to the input amplitude. To see how much attenuation occurs, you can plot the bode plot using the MATLAB function “`freqz(b)`”. In this bode plot, the x-axis is frequency (scaled 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 five 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 21<sup>st</sup> order FIR filter with 75Hz cutoff frequency? Or what if I ask for a 100<sup>th</sup> 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 implement a for loop to loop through each array element. 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 along with your “b” coefficients. **First, do this with your 4<sup>th</sup> order filter.** When that is working, implement a 21<sup>st</sup> order low-pass FIR filter with a **75Hz cutoff frequency**. How does the 21<sup>st</sup> order filter compare to the 4<sup>th</sup> order filter? **Show your TA.**

## Exercise 3: Sample Joystick and Filter X and Y Voltages with 21<sup>st</sup> Order FIR Filter

---

For this exercise use your own boards with the joystick given to you. You will set up ADCA to sample the two rotation potentiometers of the small joystick. Set up ADCA very similar to ADCD in exercise 1. You will have to find out the interrupt number for ADCA1 and enable it similar to how you enabled the ADCD1 interrupt. In order to not call the ADCD1 interrupt for this exercise, comment out the line of code from exercise 1 and 2 where you enabled interrupt 1.6. ADCINA2 is connected to one potentiometer and ADCINA3 is connected to the other potentiometer. As you move the joystick, the voltage into the two ADC channels changes between 0 at 3.3V (that’s a bit of a problem because the ADC channels sense up to 3.0V, but this only effects a small amount of movement range on the greater than 1.65V direction of the potentiometers). Sample the joystick voltages every 1ms and filter the sample ADC readings with the 21<sup>st</sup> order FIR filter designed and implemented in exercise 2. Every 100ms, print the filtered values of both rotations to TeraTerm and demonstrate the voltages changing as you move the joystick.

## Exercise 4: Sample Microphone and Demonstrate Filtering Effects on Signal

---

For Exercise 4, go back to using the 3D printed robot car which has a microphone installed. Setup ADCB to sample the microphone’s audio signal. ADCINB4 is connected to the audio signal from the microphone board that ranges from 0V to 3.0V. Setup ADCB very similar to ADCD in exercise 1 but here you will only need SOC0 since there is only one ADC channel. You will have to find out the interrupt number for ADCB1 and enable it similar to how you enabled the ADCD1 interrupt. In order for the ADCD1 and ADCA1 interrupts not to be called, comment out the lines of code from exercise 1, 2, and 3 where you enabled interrupt 1.6 and ADCA1’s interrupt. Sample the microphone at first every 0.25 milliseconds and echo the ADC voltage reading to DACA every sample. **Demo to your TA.**

Using what you learned in Exercise 2 and that you are sampling at 4000Hz, design and implement a 31<sup>st</sup> order low-pass filter with cutoff frequency of 500Hz that filters the microphone signal. Send the filtered value to DACA to observe the output of the filter. Using your Cell phone, find an app that can generate tones. While playing a 1000Hz tone, place your phone close to the microphone. Observe what the filtered output looks like on the Oscilloscope. Try some other tones. **Demo to your TA.**

Let's see how much time it takes for the F28379D processor to run the FIR filter equations. We are going to use the logic analyzer to measure the time your code takes to process inside the ADCB1 interrupt function. To do this use pin GPIO52 which currently has no other use. Just like all the LED GPIO pins are setup as output pins, setup GPIO52 as an output pin. We are going to connect GPIO52 to the logic analyzers D0 input. To time the processing of your ADCB, when you first come into your interrupt function SET GPIO52. The last thing you do in your interrupt function before you exit is CLEAR GPIO52. Then, measure the pulse width on the scope to see how long your code keeps you inside the interrupt function. **Demo to your TA.**

Change your sample rate to 10000Hz. Switch your filter to a band-pass filter that only allows a certain range of frequencies through. Can you design a filter that can identify when a 2000 Hz tone is played in the microphone? You may need to change the order of the FIR filter to create a better band-pass filter. **Demo to your TA.**

## Lab Checklist

---

1. Demonstrate your sampled signal and signal aliasing.
2. Demonstrate your implemented 4<sup>th</sup> order averaging filter.
3. Demonstrate your implemented 4<sup>th</sup> order 50Hz cutoff filter.
4. Demonstrate your implemented 21<sup>st</sup> order 75Hz cutoff filter.
5. Demonstrate your joystick voltages changing in TeraTerm.
6. Demonstrate your microphone signal echoed to DACA every .25 ms.
7. Demonstrate your microphone signal filtered with a band-pass filter passing 2000Hz.
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.