ECE 121
Lab 3: Non-volatile Memory and ADC Filters

I2C, EEPROM, and Analog-to-Digital Converters

Mel Ho
Student ID #: 1285020

Winter 2021

February 27, 2021

1 Introduction and Overview

In lab 3, we learned how to use the SPI interface with an EEPROM module and analog pins inside of our microcontroller. Using ADC, we then took samples from four different input channels, applied filters onto our signals, and then stored in them in memory with our non-volatile memory (NVM) library.

2 Non-Volatile Memory

Using the Inter-Integrated Circuit (I2C) protocol, an NVM library was developed to interface with our Serial EEPROM module on the basic I/O shield; the library consists of four functions: **read byte**, **write byte**, **read page**, and **write page**. These functions provide the standard basis for storing and retrieving data on the EEPROM module.

2.1 Inter-Integrated Circuit (I2C)

The I2C protocol is the underlying standard used for our NVM library. To initialize I2C on the UNO32, the following code block was used:

```
  #define BAUDRATE_VALUE 0x00C5 //see FRM table 24-2
  int NonVolatileMemory_Init(void)
{
    I2C1CON = 0;
    I2C1CONbits.DISSLW = HIGH;
    I2C1BRG = BAUDRATE_VALUE; //as shown in FRM table 24-2 for FPB = 40MHz
    I2C1CONbits.ON = ENABLED;

    I2C_Stop();

    return SUCCESS;
}

```

Our desired clock frequency for the EEPROM module is 100KHz and we use the preceding equation to calculate our baud rate value

$$\text{PR} = \frac{F_{PB}}{2 \cdot F_{\text{desired}}} - 1 - \frac{F_{PB} \cdot T_{\text{PGOB}}}{2} \tag{1}$$

Where T_{PGOB} is nominally 130ns. A table is provided in the FRM (see table 24-2) which provides us a calculated hex value given out F_{PB} and F_{desired}, simplifying the calculation process.

While the EEPROM requires a specific sequence of calls to interact with our microcontroller, the I2C protocol has core functions that work with any slave unit. Dedicated I2C functions were implemented to isolate these steps and also maintain an organized coding structure.

```c
void I2C_Idle(void)
{
    while(I2C1CON & 0x1F | I2C1STATbits.TRSTAT);
}

void I2C_Start(void)
{
    I2C1CONbits.SEN = ENABLED; // START condition enabled
    while(I2C1CONbits.SEN); // wait for START to end
}

void I2C_Stop(void)
{
    I2C1CONbits.PEN = ENABLED; // STOP condition enabled
    while(I2C1CONbits.PEN); // wait for STOP to end
}

void I2C_Restart(void)
{
    I2C1CONbits.RSEN = ENABLED; // RESTART condition enabled
    while(I2C1CONbits.RSEN == ENABLED); // wait for RESTART to end
}

void I2C_Send(unsigned char byte)
{
    I2C1TRN = byte;
    while(I2C1STATbits.TRSTAT); // wait for transmission to finish
    while(I2C1STATbits.TBF); // wait for transmission buffer to be empty

    while(I2C1STATbits.ACKSTAT == NACK); // wait until we receive an ACK from slave
}

unsigned char I2C_Receive(void)
{
    I2C1CONbits.RCEN = ENABLED;
    while(!I2C1STATbits.RBF); // wait for receive buffer to be full

    return I2C1RCV; // return buffer value
}

void I2C_SendACK(void)
{
    I2C1CONbits.ACKDT = ACK; // set to ACK
    I2C1CONbits.ACKEN = ENABLED; // send ACK

    while(I2C1CONbits.ACKEN); // wait until cleared
}

void I2C_SendNACK(void)
{
    I2C1CONbits.ACKDT = NACK; // set to NACK
    I2C1CONbits.ACKEN = ENABLED; // send NACK

    while(I2C1CONbits.ACKEN); // wait until cleared
}
```

With the scaffolding in place, we can now develop the logic for communicating with the EEPROM module.

2.2 EEPROM

Based on the steps provided in section 3 of our lab 3 manual, a state machine was designed for our NVM read and write byte functions. From Fig. 1, we can see the expected sequence for reading a byte and writing a byte. For our **write byte function**, we first send a **START** to our transmit buffer, then we send our I2C slave's device address with a **WRITE** byte included to indicate to our slave that we will be sending write information to the 24LC256 EEPROM unit on our basic I/O shield. We check for an **ACK** from our slave that this request has been received, then we send the **high**

byte of the address we wish to write to and then the **low byte** of our address, waiting for an ACK between each send. Once the low byte has been ACKed by our EEPROM unit, we finally send the data we want to store in our address location. Once our slave sends back an ACK, we send over a **STOP** to end our transmission sequence.

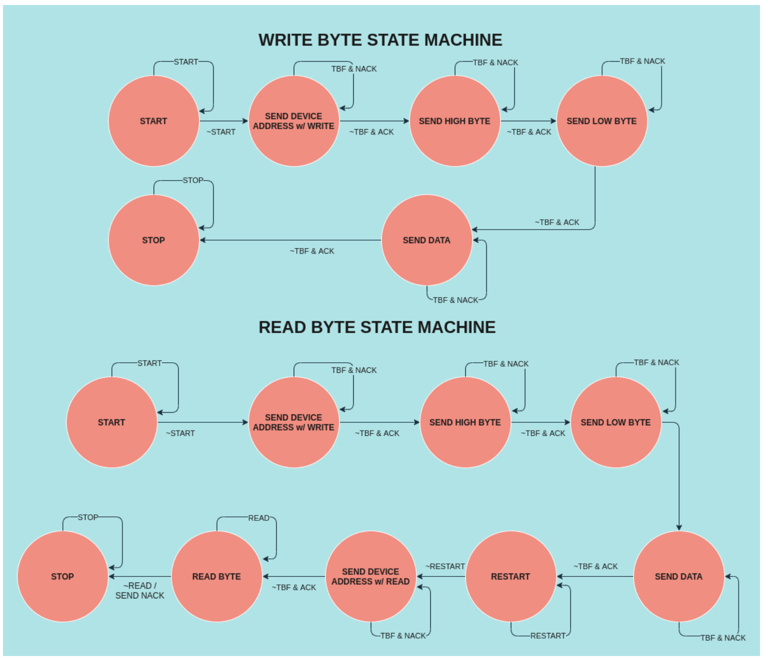


Figure 1: I2C State Machine for EEPROM

The **read byte function** uses the same logic as write byte for the first half of its protocol sequence, but it then sends a **REPEATED START** to the EEPROM and sends the device address again but with a **READ** byte; this tells the EEPROM that we wish to read from our memory address instead of writing to it. We then wait for the slave to send back an ACK and then state then **RECEIVE** sequence and wait for our slave to finish transmitting our desired data. Once the data has be transferred over, a **NACK** is sent to the device to indicate that we are done with reading. We finish the read byte sequence by sending a STOP, freeing up our I2C module. Read page and Write page follow a similar sequence with the exception that a for loop is used to continuously read or write bytes from the EEPROM module. See Appendix A. for the specific code implementation. Once everything in our NVM library has been implemented, a test harness was used before moving onto implementing our ADC filters.

2.3 Filtering Analog Signals

The second component for our Lab 3 program is sampling a signal using analog pins on the UNO32 and then applying a digital filter to it. We can achieve this using the built-in Analog-to-Digital Converter (ADC) module and functions that handle the storing and applying of our filter weights to our sampled data.

2.3.1 Analog-to-Digital Conversion

The UNO32 contains a 10-bit ADC that we will be using to map our signal's voltage differences to a value between 0-1023. This allows us to quantify our analog signal and perform processing on it such as applying a low pass, high pass, or band pass filter. For our application, we will be applying a low pass and high pass filter to the data measured by the analog pins, A0-A3. Before we can start implementing the filter functions in our ADC library, we must first configure our microcontroller to interface with its built-in 10-bit ADC; this is done with the code block below:

```c
#define TPB_K 348
#define UNSIGNED_16BIT_INT 0b000
#define AUTO_CONVERT 0b111
#define AUTO_SAMPLE 1

#define LOW 0
#define HIGH 1

#define INTERNAL_VOLT_REF 0
#define SCAN_MODE 1
#define INTERRUPT_AT_FOURTH 3
#define BUFFER_16BIT 0

#define DISABLED 0
#define ENABLED 1

#define CLEAR 0

#define USE_PERIPHERAL_CLK 0
#define ADC_CLOCK_PRESCALER 173 //((TPB_K / 2) - 1)
#define SAMPLE_TIME 16

int ADCFilter_Init(void)
{
    AD1CON1 = DISABLED;
    IEC1bits.AD1IE = DISABLED;

    AD1PCFGbits.PCFG2 = LOW;
    AD1PCFGbits.PCFG4 = LOW;
    AD1PCFGbits.PCFG8 = LOW;
    AD1PCFGbits.PCFG10 = LOW;

    AD1CON1bits.FORM = UNSIGNED_16BIT_INT;
    AD1CON1bits.SSRC = AUTO_CONVERT;

    AD1CON2bits.VCFG = INTERNAL_VOLT_REF;
    AD1CON2bits.CSCNA = SCAN_MODE;
    AD1CON2bits.SMPI = INTERRUPT_AT_FOURTH; // based on A0-A3
    AD1CON2bits.BUFM = BUFFER_16BIT;

    AD1CON3bits.ADRC = USE_PERIPHERAL_CLK;
    AD1CON3bits.SAMC = SAMPLE_TIME;
    AD1CON3bits.ADCS = ADC_CLOCK_PRESCALER;

    AD1CSSLbits.CSSL2 = HIGH;
    AD1CSSLbits.CSSL4 = HIGH;
    AD1CSSLbits.CSSL8 = HIGH;
    AD1CSSLbits.CSSL10 = HIGH;


    IFS1bits.AD1IF = CLEAR;
    IEC1bits.AD1IE = ENABLED;
    IPC6bits.AD1IP = 5;
    AD1CON1bits.ON = ENABLED;
    AD1CON1bits.ASAM = AUTO_SAMPLE;
    return SUCCESS;

}
```

With this configuration, we have set the pins A0-A3 to be analog and for them to be automatically sampled and converted. The ADC interrupt will also trigger every four samples and has a sample time of $16T_{ad}$ using the peripheral clock. The ADC prescaler chosen is calculated using the following equation:

$$\text{PR} = \frac{\text{TPB}_K}{2} - 1 \tag{2}$$

Lastly, we will be using our internal voltages as reference for our conversion. With this configuration, every time the interrupt is triggered, our ADC buffers **ADC1BUF0-ADC1BUF3** will contain measured data from each analog pin; these values are then stored inside a 2D array where the first column holds the channel number, and the second column contains the measured values.

```
void __ISR(_ADC_VECTOR) ADCIntHandler(void)
{
    IFS1bits.AD1IF = 0;

    ADCData[0][lastIndex] = ADC1BUF0;
    ADCData[1][lastIndex] = ADC1BUF1;
    ADCData[2][lastIndex] = ADC1BUF2;
    ADCData[3][lastIndex] = ADC1BUF3;

    lastIndex = (lastIndex + 1) % FILTERLENGTH;
}
```

Because we will be constantly sampling data in this program, we treat our data array as a circular buffer where old data is over written every **32 samples**. A index variable (lastIndex) is used to keep track of our next data location.

2.3.2 Filtering using Finite Impulse Response (FIR)

For this lab, a finite impulse response filter is used to filter our data. The FIR filter consists of a number of "taps" and a corresponding weight for each one; these weights are sent from our Lab interface and are stored into a dedicated 2D filter array where the first column are the column numbers and the second column holds each channel's **32 filter weights**. Depending on whether the filter we are using is a high pass or low pass variant of FIR, certain frequency values will be weighted higher than others. We then sum all of our weighted data and divide that value by the total amount of weights. We can see how this done mathematically:

$$\text{FIR} = \frac{1}{\sum w_i} \sum w_i x_i \tag{3}$$

where w_i is our FIR weights, and x_i is our samples. For our implementation a slight variant is used where we divide our total filtered data by 2^{15} instead to normalize our filtered value within our ADC range. We can see this equation used in the apply filter function below:

```c
short ADCFilter_ApplyFilter(short filter[], short values[], short startIndex)
{
    int filteredValue = 0;
    short i =  startIndex;
    unsigned int j;

    for (j = 0; j < FILTERLENGTH; j++)

    {
        i--;
        if (i < 0 || i == -1)
            i = FILTERLENGTH - 1;

        filteredValue += (filter[j] * values[i]);
    }
    filteredValue = ((filteredValue >> 15) & 0xffff); // divide by 2^15 and mask bits.

    return ((short)filteredValue);
}
```

After our FIR filter ready and thoroughly tested with a test harness, we can finally combined both our NVM and ADC libraries in our main Lab 3 program.

2.3.3 Lab 3 Interface

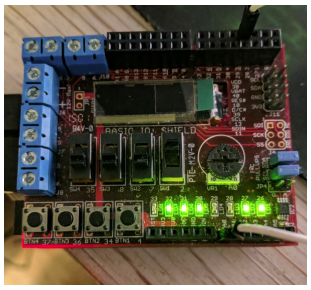


Figure 2: UNO32 Displaying Absolute Value using basic I/O shield LEDs

For our lab 3 program, we first needed to print a debug message with the date and time of when our program was compiled. We then would read from our NVM the existing filter coefficients for each corresponding analog pins, A0-A3. The lab 3 interface would display the raw, filtered, absolute, peak-to-peak, min, and max values of one channel and filter setting depending on **SW1-SW3** on our basic I/O shield. **SW1** and **SW2** determined which channel is currently selected for viewing using the following mapping:

SW1	SW2	Channel
0	0	0
1	0	1
0	1	2
1	1	3

SW3 determined which of the two different saved filters will be used on our current channel. These values are updated in the interface every time their state changed. The LEDs on the basic I/O shield are also used to display either the **peak-to-peak** or **absolute** value of our filtered signal depending on the position of **SW4**. In peak-to-peak mode, every 100 samples, we take the difference between the max and min values of that set and display our peak-to-peak value on our LEDs. This is done using the preceding code block:

```
case (PEAK_TO_PEAK):
    // take min and max of filter readings over some window of measurements.
    if (measurementCount % MEASUREMENT_WINDOW == 0)
    {
        LEDS_SET(0x0);
        max = -9999;
        min = 9999;
        for (i=0; i < MEASUREMENT_WINDOW; i++)
        {
            if (filteredReadings[i] > max)
            {
                max = filteredReadings[i];
            }


            if (filteredReadings[i] < min)
                min = filteredReadings[i];
        }

        peakToPeak = max - min;

        // Use linear interpolation to scale values from 0-1261 (highest peak to peak value) -> 0-255
        lerpRatio = peakToPeak * LERP_FRAC_1261;
        scaledValue = lerp(LEDS_MIN,LEDS_MAX,lerpRatio);
        LEDS_SET(scaledValue);

    }
    break;
```

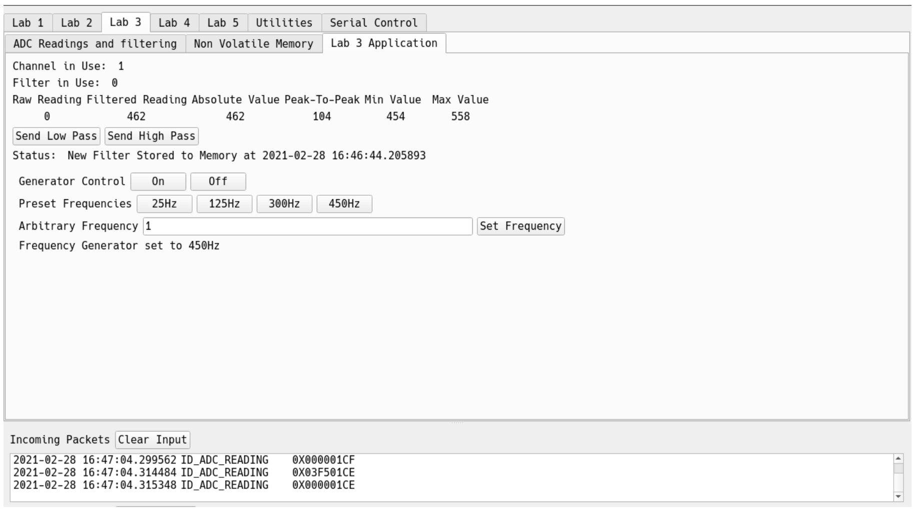
Because our peak-to-peak values can go outside of our LED range, linear interpolation was used to map our peak-to-peak range (0-1261) to 0-255. A similar technique is used when our microcontroller is in absolute value mode:

```
case (ABSOLUTE_VALUE):
    LEDS_SET(0x0);

    absoluteRead = ADCReading.filtered;

    if (ADCReading.filtered < 0)
        absoluteRead = (ADCReading.filtered * -1);

    // Use linear interpolation to scale values from 0-1023 -> 0-255
    lerpRatio = absoluteRead * LERP_FRAC_1023;
    scaledValue = lerp(LEDS_MIN, LEDS_MAX, lerpRatio);
    LEDS_SET(scaledValue);

    break;
```

When we change our filter values using the lab interface, these new filter weights are stored into their designated NVM address and are applied to our existing signal. This can be seen in our code:

```c
case (ID_ADC_FILTER_VALUES):
    Protocol_GetPayload(&filterWeights);

    for (i=0; i < FILTERLENGTH; i++)
    {
        filterWeights[i] = Protocol_ShortEndednessConversion(filterWeights[i]);
    }

    operationStatus = ADCFilter_SetWeights(channel, filterWeights);
    if (operationStatus == SUCCESS)
    {
        Protocol_SendMessage(sizeof(operationStatus), ID_ADC_FILTER_VALUES_RESP, &operationStatus);

        // Store new filter coefficients
        if (filter == FILTER_1){
            NonVolatileMemory_WritePage((channel * 2) + 1, FILTERLENGTH, (unsigned char *) filterWeights);
        }
        if (filter == FILTER_0)
        {
            NonVolatileMemory_WritePage((channel * 2), FILTERLENGTH, (unsigned char *) filterWeights);
        }
    }

    break;
```

These filter weights are stored in pages 0-8 where the filter weights on filter 1 take up the even pages and the weights on filter 0 take up the odd pages. Lastly, the lab 3 program sends an update of the raw and filtered ADC readings of the current channel every 10ms using the **ID_ADC_READING** ID. Please consult Appendix A for the full implementation of this program.



```
Channel in Use:  1
Filter in Use:  0
Raw Reading Filtered Reading Absolute Value Peak-To-Peak Min Value  Max Value
     0            462            462           104        454       558
Status:  New Filter Stored to Memory at 2021-02-28 16:46:44.205893
 Generator Control
 Preset Frequencies
 Arbitrary Frequency 1
 Frequency Generator set to 450Hz
Incoming Packets
2021-02-28 16:47:04.299562 ID_ADC_READING    0X000001CF
2021-02-28 16:47:04.314484 ID_ADC_READING    0X03F501CE
2021-02-28 16:47:04.315348 ID_ADC_READING    0X000001CE
```

Figure 3: Snapshot of the Lab 3 Interface with a 450Hz square wave signal.

3 Conclusion

Lab 3 familiarized us with how to use the I2C protocol and analog pins on our micro-controller. We also learned the different types of digital filters we can implement to a measured signal, such as low pass, high pass, band pass, and FIR filters. A lot of the challenges of this lab centered around the I2C module; because an I2C transmission consisted of many moving parts, there were more points of failure compared to using the SPI interface. Once the I2C module would fail, it would lock the EEPROM, making it impossible to communicate with the slave unit without disconnecting the USB and reconnecting it to the computer. Therefore, a lot of precautionary delays and blocking was necessary to ensure a stable signal. Overall, the lab provided a peak into how memory is stored on a computer, how analog signals are converted to digital values, and how digital filters can be used on measured signals from an input.

Appendix

A. Source Code

3.0.1 Lab 3 Application

```
/*
 * File:   lab3Application.c
 * Author: Mel Ho
 *
 * Created on February 15, 2021, 4:48 PM
 */
#include "xc.h"
#include "BOARD.h"
#include "Protocol.h"
#include <stdio.h>
#include "ADCFilter.h"
#include "NonVolatileMemory.h"
#include "FreeRunningTimer.h"
#include "FrequencyGenerator.h"
#include "MessageIDs.h"
#include <strings.h>
static char message[MAXPAYLOADLENGTH];
#define INPUT 1
#define OUTPUT 0
#define MEASUREMENT_WINDOW 100
#define LOW 0
#define HIGH 1
#define MAX_DATA_SIZE 64
#define PAGE_PACKET_SIZE (MAX_DATA_SIZE + ADDRESS_SIZE)
#define BYTE_PACKET_SIZE (ADDRESS_SIZE + BYTE_SIZE)
#define READ_LIMIT 10
#define ADDRESS_SIZE 4
#define BYTE_SIZE 1
#define CHANNEL_0 0
#define CHANNEL_1 1
#define CHANNEL_2 2
#define CHANNEL_3 3
#define FILTER_0 0
#define FILTER_1 1
#define ABSOLUTE_VALUE 0
#define PEAK_TO_PEAK 1
#define SWITCH_1_INIT (TRISDbits.TRISD8 = INPUT) // PIN 2
#define SWITCH_1 (PORTDbits.RD8)
#define SWITCH_2_INIT (TRISDbits.TRISD9 = INPUT) // PIN 7
#define SWITCH_2 (PORTDbits.RD9)
#define SWITCH_3_INIT (TRISDbits.TRISD10 = INPUT) // PIN 8
#define SWITCH_3 (PORTDbits.RD10)
#define SWITCH_4_INIT (TRISDbits.TRISD11 = INPUT) // PIN 35
#define SWITCH_4 (PORTDbits.RD11)
#define LEDS_MIN 0
#define LEDS_MAX 255
#define LERP_FRAC_1261 .000793  // 1/1261: used to create lerp ratio
#define LERP_FRAC_1023 .0009775 // 1/1023: used to create lerp ratio
```

```
static short displayMode;
static short prevChannel = 0;
static short prevFilter = 0;
static short rawReading;
static unsigned char messageID;
static int operationStatus;
static unsigned int milliseconds;
static unsigned int hz = 10;
static unsigned int measurementCount = 0;
static unsigned char channelFilter;
static signed short filterWeights[FILTERLENGTH];
static signed short filteredReadings[MEASUREMENT_WINDOW];
unsigned char pageData[MAX_DATA_SIZE];
static signed short min = 0;
static signed short max = 0;
static unsigned short absoluteRead;
static unsigned int i;
static unsigned int page = 0;
static unsigned int pin = 0;
union ADCReadings
{
    struct {
        short unfiltered;
        short filtered;
    };
    char adc[4];
};
union NVMData {
    struct {
        unsigned int address;
        unsigned char data;
    };
    char asChar[BYTE_PACKET_SIZE]; // size of 5 bytes
};
union NVMPageData {
    struct {
        int page;
        unsigned char data[MAX_DATA_SIZE];
    };
    char asChar[PAGE_PACKET_SIZE]; // size of 68 bytes
};
static char message[MAXPAYLOADLENGTH];
/**
 * @Function lerp(uint min, uint max, float ratio)
 * @param min, max, ratio
 * @return uint
 * @brief Linear interpolation function used to map our encoder angles into a range between 600-2400 motor ticks.*/
unsigned int lerp(unsigned int min, unsigned int max, float ratio)
{
    return min + ratio * (max - min);
}
int main(void)
{
    BOARD_Init();
    Protocol_Init();
    ADCFilter_Init();
    NonVolatileMemory_Init();
    FreeRunningTimer_Init();
    FrequencyGenerator_Init();
    union ADCReadings ADCReading;
    ADCReading.filtered = 0;
    ADCReading.unfiltered = 0;
    unsigned char channel = CHANNEL_0;
    unsigned char filter = FILTER_0;
    union NVMData byteData;
    union NVMPageData pageInfo;
    unsigned int payload = 0;
    unsigned char byte = 0;
    unsigned char channelFilter = 0;
    float lerpRatio = 0;
    unsigned int scaledValue;
    signed int peakToPeak;
    signed int max = 0;
    signed int min = 0;
    unsigned int currentFrequency = 0;
    unsigned int freqState = 0;
    SWITCH_1_INIT;
    SWITCH_2_INIT;
    SWITCH_3_INIT;
    SWITCH_4_INIT;
    LEDS_INIT();
    sprintf(message, "Protocol Test Compiled at %s %s", __DATE__, __TIME__);
    Protocol_SendDebugMessage(message);
```

```
// initializes channel and filter values to 0.
channelFilter = channel << 4 | filter;
Protocol_SendMessage(sizeof(channelFilter), ID_LAB3_CHANNEL_FILTER, &channelFilter);
while (1)
{
    milliseconds = FreeRunningTimer_GetMilliSeconds();
    if (milliseconds > hz)
    {
        hz += 10;
        messageID = Protocol_ReadNextID();
        /********************
        * CHANNEL SELECTION *
         ********************/
        // Determine what channel is selected based on SW1-SW2
        if (SWITCH_1 == LOW && SWITCH_2 == LOW)
            channel = CHANNEL_0;
        else if (SWITCH_1 == HIGH && SWITCH_2 == LOW)
            channel = CHANNEL_1;
        else if (SWITCH_1 == LOW && SWITCH_2 == HIGH)
            channel = CHANNEL_2;
        else if (SWITCH_1 == HIGH && SWITCH_2 == HIGH)
            channel = CHANNEL_3;
        /*********************
        *  FILTER SELECTION  *
         *********************/
        // Determine which filter is selected using SW3
        if (SWITCH_3 == LOW)
            filter = FILTER_0;
        else if (SWITCH_3 = HIGH)
            filter = FILTER_1;
        // updates channel and filter configurations
        if (channel != prevChannel || filter != prevFilter)
        {
            channelFilter = channel << 4 | filter; //combines channel filter values and sends them over
            sprintf(message, "Sending new channel filter configuration data.");
            Protocol_SendDebugMessage(message);
            Protocol_SendMessage(sizeof(channelFilter), ID_LAB3_CHANNEL_FILTER, &channelFilter);
            prevChannel = channel;
            prevFilter = filter;
        }
        /********************
        *   DISPLAY MODE    *
         ********************/
        // Determine which display mode is selected using SW4
        if (SWITCH_4 == LOW)
            displayMode = ABSOLUTE_VALUE;
        else if (SWITCH_4 == HIGH)
            displayMode = PEAK_TO_PEAK;
        /********************
        *     MESSAGE ID    *
         ********************/
        switch(messageID)
        {
        case (ID_ADC_FILTER_VALUES):
            Protocol_GetPayload(&filterWeights);
            for (i=0; i < FILTERLENGTH; i++)
            {
                filterWeights[i] = Protocol_ShortEndednessConversion(filterWeights[i]);
            }
            operationStatus = ADCFilter_SetWeights(channel, filterWeights);
            if (operationStatus == SUCCESS)
            {
                Protocol_SendMessage(sizeof(operationStatus), ID_ADC_FILTER_VALUES_RESP, &operationStatus);
                // Store new filter coefficients
                if (filter == FILTER_1){
                    NonVolatileMemory_WritePage((channel * 2) + 1, FILTERLENGTH, (unsigned char *) filterWeights);
                }
                if (filter == FILTER_0)
                {
                    NonVolatileMemory_WritePage((channel * 2), FILTERLENGTH, (unsigned char *) filterWeights);
                }
            }
            break;
        case (ID_LAB3_SET_FREQUENCY):
            Protocol_GetPayload(&currentFrequency);
            currentFrequency = Protocol_ShortEndednessConversion(currentFrequency);
            FrequencyGenerator_SetFrequency(currentFrequency);
            break;
        case (ID_LAB3_FREQUENCY_ONOFF):
            Protocol_GetPayload(&freqState);
            if (freqState)
                FrequencyGenerator_On();
            else
```

```
                FrequencyGenerator_Off();
                break;
                case (ID_NVM_READ_BYTE):
                  sprintf(message, "Read byte");
                  Protocol_SendDebugMessage(message);
                  Protocol_GetPayload(&payload);
                  payload = Protocol_IntEndednessConversion(payload);
                  byte = NonVolatileMemory_ReadByte(payload);
                  Protocol_SendMessage(sizeof(byte), ID_NVM_READ_BYTE_RESP, &byte);
                break;
                case (ID_NVM_WRITE_BYTE):
                    sprintf(message, "Write byte");
                    Protocol_SendDebugMessage(message);
                    Protocol_GetPayload(&byteData);
                    byteData.address = Protocol_IntEndednessConversion(byteData.address);
                    operationStatus = NonVolatileMemory_WriteByte(byteData.address, byteData.data);
                    if (operationStatus == SUCCESS)
                        Protocol_SendMessage(sizeof(operationStatus), ID_NVM_WRITE_BYTE_ACK, &operationStatus);
                    break;
//
                case (ID_NVM_READ_PAGE):
                      sprintf(message, "Read Page");
                      Protocol_SendDebugMessage(message);
                      Protocol_GetPayload(&payload);
                      payload = Protocol_IntEndednessConversion(payload);
                      NonVolatileMemory_ReadPage(payload, sizeof(pageData), pageData);
                      Protocol_SendMessage(sizeof(pageData), ID_DEBUG, &pageData);
                      Protocol_SendMessage(sizeof(pageData), ID_NVM_READ_PAGE_RESP, &pageData);
                break;
                case (ID_NVM_WRITE_PAGE):
                    sprintf(message, "Write Page");
                    Protocol_SendDebugMessage(message);
                    Protocol_GetPayload(&pageInfo);
                    sprintf(message, "data size: %i", sizeof(pageInfo.data));
                    Protocol_SendDebugMessage(message);
                    pageInfo.page = Protocol_IntEndednessConversion(pageInfo.page);
                    operationStatus = NonVolatileMemory_WritePage(pageInfo.page, sizeof(pageInfo.data), pageInfo.data);
                    if (operationStatus = SUCCESS)
                        Protocol_SendMessage(sizeof(operationStatus), ID_NVM_WRITE_PAGE_ACK, &operationStatus);
                    break;
            }
//
            ADCReading.unfiltered = Protocol_ShortEndednessConversion(ADCFilter_RawReading(channel));
            ADCReading.filtered = ADCFilter_FilteredReading(channel);
            filteredReadings[measurementCount] = ADCReading.filtered;
            switch (displayMode)
            {
            case (PEAK_TO_PEAK):
//                // take min and max of filter readings over some window of measurements.
                if (measurementCount % MEASUREMENT_WINDOW == 0)
                {
                    LEDS_SET(0x0);
                    max = -9999;
                    min = 9999;
                    for (i=0; i < MEASUREMENT_WINDOW; i++)
                    {
                        if (filteredReadings[i] > max)
                        {
                            max = filteredReadings[i];
                        }
                        if (filteredReadings[i] < min)
                            min = filteredReadings[i];
                    }
                    peakToPeak = max - min;
                    // Use linear interpolation to scale values from 0-1261 (highest peak to peak value) -> 0-255
                    lerpRatio = peakToPeak * LERP_FRAC_1261;
                    scaledValue = lerp(LEDS_MIN,LEDS_MAX,lerpRatio);
                    LEDS_SET(scaledValue);
                }
                break;
            case (ABSOLUTE_VALUE):
                LEDS_SET(0x0);
                absoluteRead = ADCReading.filtered;
                if (ADCReading.filtered < 0)
                    absoluteRead = (ADCReading.filtered * -1);
                // Use linear interpolation to scale values from 0-1023 -> 0-255
                lerpRatio = absoluteRead * LERP_FRAC_1023;
                scaledValue = lerp(LEDS_MIN, LEDS_MAX, lerpRatio);
                LEDS_SET(scaledValue);
                break;
            default:break;
            }
            // NVM load filters in. Currently commented  due to finnicky nature of page read. To check and see if the filters successfully write
            // please switch to Non Volatile Memory and click read page for pages 0-8. It appears that page reading in every 10ms seems to overwhelm
            // I2C EEPROM and then it pretty much hangs. Haven't figured out a great work that consistently fixes this issue :(
//            switch(filter)
//            {
//            case (FILTER_0):
//                NonVolatileMemory_ReadPage((channel * 2), FILTERLENGTH, (unsigned char *) filterWeights);
////                ADCFilter_SetWeights(channel, filterWeights);
//                break;
//            case (FILTER_1):
//                NonVolatileMemory_ReadPage((channel * 2) + 1, FILTERLENGTH, (unsigned char *) filterWeights);
////                ADCFilter_SetWeights(channel, filterWeights);
//                break;
//            }
            measurementCount = (measurementCount + 1) % MEASUREMENT_WINDOW;
            ADCReading.filtered = Protocol_ShortEndednessConversion(ADCReading.filtered);
            Protocol_SendMessage(sizeof(ADCReading.adc), ID_ADC_READING, ADCReading.adc);
        }
    }
    return 0;
}
```

3.0.2 Non-volatile Memmory

```
/*
 * File:   NonVolatileMemory.c
 * Author: Mel Ho
 * Brief: I2C NonVolatile Memory module
 * Created on <month> <day>, <year>, <hour> <pm/am>
 * Modified on <month> <day>, <year>, <hour> <pm/am>
 */
/*******************************************************************************
 * #INCLUDES                                                                   *
 ******************************************************************************/
#include <proc/p32mx340f512h.h>
#include <xc.h>
#include "NonVolatileMemory.h" // The header file for this source file.
#include "BOARD.h"
#include "Protocol.h"
#include "delays.h"
/*******************************************************************************
 * PRIVATE #DEFINES                                                            *
 ******************************************************************************/
#define EEPROM_24LC256_ADDRESS 0b1010000
#define FPB (BOARD_GetPBClock())
#define I2C_CLOCK_RATE 100000
#define BAUDRATE_VALUE 0x00C5 //see FRM table 24-2
#define DISABLED 0
#define ENABLED 1
#define ACK 0
#define NACK 1
#define NOT_COMPLETED 0
#define COMPLETED 1
#define FINISHED 0
#define IN_PROGRESS 1
#define IS_BUSY (I2C1CON & 0x1F)
#define WRITE_BIT 0
#define READ_BIT 1
#define PAGE_SIZE 64
#define FALSE 0
#define TRUE 1
#define LOW 0
#define HIGH 1
#define READ_LIMIT 10
#define MAX_DATA_SIZE 64
#define ADDRESS_SIZE 4
#define BYTE_SIZE 1
#define PAGE_PACKET_SIZE (MAX_DATA_SIZE + ADDRESS_SIZE)
#define BYTE_PACKET_SIZE (ADDRESS_SIZE + BYTE_SIZE)
//#define NVM_TEST
/*******************************************************************************
 * PRIVATE TYPEDEFS                                                            *
 ******************************************************************************/
union NVMData {
    struct {
        unsigned int address;
        unsigned char data;
    };
    char asChar[BYTE_PACKET_SIZE]; // size of 5 bytes
};
```

```
union NVMPageData {
    struct {
        int page;
        unsigned char data[MAX_DATA_SIZE];
    };
    char asChar[PAGE_PACKET_SIZE]; // size of 68 bytes
};
static char message[MAXPAYLOADLENGTH];
/*******************************************************************************
 * PRIVATE FUNCTIONS PROTOTYPES                                                 *
 ******************************************************************************/
void I2C_Idle(void);
void I2C_Start(void);
void I2C_Stop(void);
void I2C_Restart(void);
void I2C_Write(unsigned char byte);
unsigned char I2C_Read(void);
void I2C_SendACK(void);
void I2C_SendNACK(void);
/*******************************************************************************
 * PUBLIC FUNCTION IMPLEMENTATIONS                                             *
 ******************************************************************************/
/*******************************************************************************
 * I2C FUNCTION IMPLEMENTATIONS                                                *
 ******************************************************************************/
void I2C_Idle(void)
{
    while(I2C1CON & 0x1F | I2C1STATbits.TRSTAT);
}
void I2C_Start(void)
{
    I2C1CONbits.SEN = ENABLED; // START condition enabled
    while(I2C1CONbits.SEN); // wait for START to end
}
void I2C_Stop(void)
{
    I2C1CONbits.PEN = ENABLED; // STOP condition enabled
    while(I2C1CONbits.PEN); // wait for STOP to end
}
void I2C_Restart(void)
{
    I2C1CONbits.RSEN = ENABLED; // RESTART condition enabled
    while(I2C1CONbits.RSEN == ENABLED); // wait for RESTART to end
}
void I2C_Send(unsigned char byte)
{
    I2C1TRN = byte;
    while(I2C1STATbits.TRSTAT); // wait for transmission to finish
    while(I2C1STATbits.TBF); // wait for transmission buffer to be empty
    while(I2C1STATbits.ACKSTAT == NACK); // wait until we receive an ACK from slave
}
unsigned char I2C_Receive(void)
{
    I2C1CONbits.RCEN = ENABLED;
    while(!I2C1STATbits.RBF); // wait for receive buffer to be empty
    return I2C1RCV; // return buffer value
}
void I2C_SendACK(void)
{
    I2C1CONbits.ACKDT = ACK; // set to ACK
    I2C1CONbits.ACKEN = ENABLED; // send ACK
    while(I2C1CONbits.ACKEN); // wait until cleared
}
void I2C_SendNACK(void)
{
    I2C1CONbits.ACKDT = NACK; // set to NACK
    I2C1CONbits.ACKEN = ENABLED; // send NACK
    while(I2C1CONbits.ACKEN); // wait until cleared
}
/*******************************************************************************
 * NVM FUNCTION IMPLEMENTATIONS                                                *
 ******************************************************************************/
/**
 * @Function NonVolatileMemory_Init(void)
 * @param None
 * @return SUCCESS or ERROR
 * @brief initializes I2C for usage */
int NonVolatileMemory_Init(void)
{
    I2C1CON = 0;
    I2C1CONbits.DISSLW = HIGH;
    I2C1BRG = BAUDRATE_VALUE; //as shown in FRM table 24-2 for FPB = 40MHz
    I2C1CONbits.ON = ENABLED;
```

```c
    I2C_Stop();
    return SUCCESS;
}
/**
 * @Function NonVolatileMemory_ReadByte(int address)
 * @param address, device address to read from
 * @return value at said address
 * @brief reads one byte from device
 * @warning Default value for this EEPROM is 0xFF */
unsigned char NonVolatileMemory_ReadByte(int address)
{
    unsigned char byte = 0;
    I2C_Idle();
    I2C_Start(); // START
    I2C_Send((EEPROM_24LC256_ADDRESS << 1) | WRITE_BIT); // loads EEPROM address with write
    I2C_Send((address >> 8) & 0xFF); // Memory Address High Byte
    I2C_Send(address & 0xFF); // Memory Address Low Byte
    I2C_Restart();
    I2C_Send((EEPROM_24LC256_ADDRESS << 1) | READ_BIT); // loads EEPROM address with read
    byte = I2C_Receive();
    I2C_SendNACK();
    I2C_Stop();
    delay_ms(50); // delay to prevent overwhelming
    return byte;
}
/**
 * @Function char NonVolatileMemory_WriteByte(int address, unsigned char data)
 * @param address, device address to write to
 * @param data, value to write at said address
 * @return SUCCESS or ERROR
 * @brief writes one byte to device */
char NonVolatileMemory_WriteByte(int address, unsigned char data)
{
    I2C_Idle();
    I2C_Start();
    I2C_Send((EEPROM_24LC256_ADDRESS << 1) | WRITE_BIT); // loads address with write
    I2C_Send((address >> 8) & 0xFF); // Memory Address High Byte
    I2C_Send(address & 0xFF); // Memory Address Low Byte
    I2C_Send(data);
    I2C_Stop();
    delay_ms(20); // delay to prevent overwhelming
    return SUCCESS;
}
/**
 * @Function int NonVolatileMemory_ReadPage(int page, char length, unsigned char data[])
 * @param page, page value to read from
 * @param length, value between 1 and 64 bytes to read
 * @param data, array to store values into
 * @return SUCCESS or ERROR
 * @brief reads bytes in page mode, up to 64 at once
 * @warning Default value for this EEPROM is 0xFF */
int NonVolatileMemory_ReadPage(int page, char length, unsigned char data[])
{
    int address = page << 6;
    I2C_Start();
    I2C_Send((EEPROM_24LC256_ADDRESS << 1) | WRITE_BIT); // loads address with write
    I2C_Send((address >> 8) & 0xFF); // Memory Address High Byte
    I2C_Send(address & 0xFF); // Memory Address Low Byte
    I2C_Restart();
    I2C_Send((EEPROM_24LC256_ADDRESS << 1) | READ_BIT); // loads address with read
    char i;
    for (i=0; i < length;i++)
    {
        data[i] = I2C_Receive();
        I2C_SendACK();
    }
    I2C_SendNACK();
    I2C_Stop();
    delay_ms(50); // delay to prevent overwhelming
    return SUCCESS;
}
/**
 * @Function char int NonVolatileMemory_WritePage(int page, char length, unsigned char data[])
 * @param address, device address to write to
 * @param data, value to write at said address
 * @return SUCCESS or ERROR
 * @brief writes one byte to device */
int NonVolatileMemory_WritePage(int page, char length, unsigned char data[])
{
    int address = page << 6;
    I2C_Idle();
    I2C_Start();
    I2C_Send(EEPROM_24LC256_ADDRESS << 1); // loads address with write
    I2C_Send((address >> 8) & 0xFF); // Memory Address High Byte
    I2C_Send(address & 0xFF); // Memory Address Low Byte
    char i;
    for (i=0; i < length;i++)
    {
        I2C_Send(data[i]);
    }
    I2C_Stop();
    delay_ms(20); //delay to prevent overwhelming
    return SUCCESS;
}
#ifdef NVM_TEST
#include "xc.h"
#include "BOARD.h"
#include <stdio.h>
#include <strings.h>
#include "MessageIDs.h"
#include "FreeRunningTimer.h"
int main(void)
{
    BOARD_Init();
    Protocol_Init();
    FreeRunningTimer_Init();
    NonVolatileMemory_Init();
    unsigned char messageID;
    unsigned char byte = 0;
    unsigned char operationStatus;
    unsigned char pageData[MAX_DATA_SIZE];
    unsigned int payload = 0;
    unsigned int milliseconds = 0;
    unsigned int hz = 10;
    union NVMData byteData;
    union NVMPageData pageInfo;
    unsigned int i = 0;
    unsigned int check = 0;
    unsigned int byteSize = 0;
    sprintf(message, "Protocol Test Compiled at %s %s", __DATE__, __TIME__);
    Protocol_SendDebugMessage(message);
//
    while(TRUE)
    {
            messageID = Protocol_ReadNextID();
            milliseconds = FreeRunningTimer_GetMilliSeconds();
            if (milliseconds > hz)
            {
                hz += 10;
                if (Protocol_IsMessageAvailable())
                {
                    switch(messageID)
                    {
                        case (ID_NVM_READ_BYTE):
                          sprintf(message, "Read byte");
                          Protocol_SendDebugMessage(message);
                          Protocol_GetPayload(&payload);
                          payload = Protocol_IntEndednessConversion(payload);
                          byte = NonVolatileMemory_ReadByte(payload);
                          Protocol_SendMessage(sizeof(byte), ID_NVM_READ_BYTE_RESP, &byte);
                        break;
                        case (ID_NVM_WRITE_BYTE):
                            sprintf(message, "Write byte");
                            Protocol_SendDebugMessage(message);
                            Protocol_GetPayload(&byteData);
                            byteData.address = Protocol_IntEndednessConversion(byteData.address);
                            operationStatus = NonVolatileMemory_WriteByte(byteData.address, byteData.data);
                            if (operationStatus == SUCCESS)
                                Protocol_SendMessage(sizeof(operationStatus), ID_NVM_WRITE_BYTE_ACK, &operationStatus);
                            break;
//
                        case (ID_NVM_READ_PAGE):
                              sprintf(message, "Read Page");
                              Protocol_SendDebugMessage(message);
                              Protocol_GetPayload(&payload);
                              payload = Protocol_IntEndednessConversion(payload);
                              NonVolatileMemory_ReadPage(payload, sizeof(pageData), pageData);
                              Protocol_SendMessage(sizeof(pageData), ID_DEBUG, &pageData);
                              Protocol_SendMessage(sizeof(pageData), ID_NVM_READ_PAGE_RESP, &pageData);
                        break;
                        case (ID_NVM_WRITE_PAGE):
                            sprintf(message, "Write Page");
                            Protocol_SendDebugMessage(message);
                            Protocol_GetPayload(&pageInfo);
                            sprintf(message, "data size: %i", sizeof(pageInfo.data));
                            Protocol_SendDebugMessage(message);
                            pageInfo.page = Protocol_IntEndednessConversion(pageInfo.page);
                            operationStatus = NonVolatileMemory_WritePage(pageInfo.page, sizeof(pageInfo.data), pageInfo.data);
                            if (operationStatus = SUCCESS)
                                Protocol_SendMessage(sizeof(operationStatus), ID_NVM_WRITE_PAGE_ACK, &operationStatus);
                            break;
                    default:break;
                    }
                }
            }
    }
   return 0;
}
#endif
```

3.0.3 ADC Filters

```
/*
 * File:   ADCFilter.c
 * Author: Mel Ho
 * Brief: The ADC Filter module
 * Created on <month> <day>, <year>, <hour> <pm/am>
 * Modified on <month> <day>, <year>, <hour> <pm/am>
 */
/*******************************************************************************
 * #INCLUDES                                                                   *
 ******************************************************************************/
#include <proc/p32mx340f512h.h>
#include <xc.h>
#include "ADCFilter.h" // The header file for this source file.
#include "BOARD.h"
#include "stdio.h"
#include "string.h"
#include "MessageIDs.h"
#include <sys/attribs.h>
#include "FrequencyGenerator.h"
#include "FreeRunningTimer.h"
#include "Protocol.h"
/*******************************************************************************
 * PRIVATE #DEFINES                                                            *
 ******************************************************************************/
#define DESIRED_PINS 0b1111101011101011 // 0xFAEB
#define CSSL_PINS ~(DESIRED_PINS)
#define TPB_K 348
#define UNSIGNED_16BIT_INT 0b000
#define AUTO_CONVERT 0b111
#define AUTO_SAMPLE 1
#define LOW 0
#define HIGH 1
#define INTERNAL_VOLT_REF 0
#define SCAN_MODE 1
#define INTERRUPT_AT_FOURTH 3
#define BUFFER_16BIT 0
#define DISABLED 0
#define ENABLED 1
#define CLEAR 0
#define USE_PERIPHERAL_CLK 0
#define ADC_CLOCK_PRESCALER 173 //((TPB_K / 2) - 1)
#define SAMPLE_TIME 16
//#define ADC_TEST
/*******************************************************************************
 * PRIVATE VARIABLES                                                           *
 ******************************************************************************/
static signed short ADCFilter[NUM_OF_CHANNELS][FILTERLENGTH];
static signed short ADCData[NUM_OF_CHANNELS][FILTERLENGTH];
static short lastIndex = 0;
static unsigned char message[MAXPAYLOADLENGTH];
union adcReading {
    struct {
        short unfiltered;
        short filtered;
    }; unsigned char aschar[4];
};
/*******************************************************************************
 * PUBLIC FUNCTION IMPLEMENTATIONS                                             *
 ******************************************************************************/
/**
 * @Function ADCFilter_Init(void)
 * @param None
 * @return SUCCESS or ERROR
 * @brief initializes ADC system along with naive filters */
int ADCFilter_Init(void)
{
    AD1CON1 = DISABLED;
    IEC1bits.AD1IE = DISABLED;
    AD1PCFGbits.PCFG2 = LOW;
    AD1PCFGbits.PCFG4 = LOW;
    AD1PCFGbits.PCFG8 = LOW;
    AD1PCFGbits.PCFG10 = LOW;
    AD1CON1bits.FORM = UNSIGNED_16BIT_INT;
    AD1CON1bits.SSRC = AUTO_CONVERT;
    AD1CON2bits.VCFG = INTERNAL_VOLT_REF;
    AD1CON2bits.CSCNA = SCAN_MODE;
    AD1CON2bits.SMPI = INTERRUPT_AT_FOURTH; // based on A0-A3
    AD1CON2bits.BUFM = BUFFER_16BIT;
    AD1CON3bits.ADRC = USE_PERIPHERAL_CLK;
    AD1CON3bits.SAMC = SAMPLE_TIME;
    AD1CON3bits.ADCS = ADC_CLOCK_PRESCALER;
    AD1CSSLbits.CSSL2 = HIGH;
    AD1CSSLbits.CSSL4 = HIGH;
    AD1CSSLbits.CSSL8 = HIGH;
    AD1CSSLbits.CSSL10 = HIGH;
    IFS1bits.AD1IF = CLEAR;
    IEC1bits.AD1IE = ENABLED;
    IPC6bits.AD1IP = 5;
    AD1CON1bits.ON = ENABLED;
    AD1CON1bits.ASAM = AUTO_SAMPLE;
    return SUCCESS;
}
/**
 * @Function ADCFilter_RawReading(short pin)
 * @param pin, which channel to return
 * @return un-filtered AD Value
 * @brief returns current reading for desired channel */
short ADCFilter_RawReading(short pin)
{
    return ADCData[pin][lastIndex];
}
/**
 * @Function ADCFilter_FilteredReading(short pin)
 * @param pin, which channel to return
 * @return Filtered AD Value
 * @brief returns filtered signal using weights loaded for that channel */
short ADCFilter_FilteredReading(short pin)
{
    short filteredData = 0;
    filteredData = ADCFilter_ApplyFilter(ADCFilter[pin],ADCData[pin], lastIndex);
    return filteredData;
}
/**
 * @Function short ADCFilter_ApplyFilter(short filter[], short values[], short startIndex)
 * @param filter, pointer to filter weights
 * @param values, pointer to circular buffer of values
 * @param startIndex, location of first sample so filter can be applied correctly
 * @return Filtered and Scaled Value
 * @brief returns final signal given the input arguments
 * @warning returns a short but internally calculated value should be an int */
short ADCFilter_ApplyFilter(short filter[], short values[], short startIndex)
{
    int filteredValue = 0;
    short i =  startIndex;
    unsigned int j;
    for (j = 0; j < FILTERLENGTH; j++)
    {
        i--;
        if (i < 0 || i == -1)
            i = FILTERLENGTH - 1;
        filteredValue += (filter[j] * values[i]);
    }
    filteredValue = ((filteredValue >> 15) & 0xffff); // divide by 2^15 and mask bits.
    return ((short)filteredValue);
}
/**
 * @Function ADCFilter_SetWeights(short pin, short weights[])
 * @param pin, which channel to return
 * @param pin, array of shorts to load into the filter for the channel
 * @return SUCCESS or ERROR
 * @brief loads new filter weights for selected channel */
int ADCFilter_SetWeights(short pin, short weights[])
{
    unsigned int i;
    for (i=0; i < FILTERLENGTH; i++)
    {
        ADCFilter[pin][i] = weights[i];
    }
    return SUCCESS;
}
void __ISR(_ADC_VECTOR) ADCIntHandler(void)
{
    IFS1bits.AD1IF = 0;
    ADCData[0][lastIndex] = ADC1BUF0;
    ADCData[1][lastIndex] = ADC1BUF1;
    ADCData[2][lastIndex] = ADC1BUF2;
    ADCData[3][lastIndex] = ADC1BUF3;
    lastIndex = (lastIndex + 1) % FILTERLENGTH;
}
#ifdef ADC_TEST
int main()
{
    BOARD_Init();
    FreeRunningTimer_Init();
    FrequencyGenerator_Init();
    Protocol_Init();
    ADCFilter_Init();
    unsigned int milliseconds = 0;
    unsigned int hz = 10;
    unsigned char channel = 0;
    unsigned char messageID;
    unsigned int i;
    unsigned int freqState = 0;
    short CurFrequency;
    unsigned int operationStatus;
//    union filters filterValues;
    short filters[FILTERLENGTH];
    union adcReading adcValues;
    sprintf(message, "Protocol Test Compiled at %s %s", __DATE__, __TIME__);
    Protocol_SendDebugMessage(message);
    while (1)
    {
        milliseconds = FreeRunningTimer_GetMilliSeconds();
        if (milliseconds > hz)
        {
            hz += 10;
            if (Protocol_IsMessageAvailable())
            {
                messageID = Protocol_ReadNextID();
                sprintf(message, "message ID: %d",messageID);
                Protocol_SendDebugMessage(message);
                switch(messageID)
                {
                case (ID_ADC_SELECT_CHANNEL):
                    Protocol_GetPayload(&channel);
                    sprintf(message, "channel: %c", channel);
                    Protocol_SendDebugMessage(message);
                    Protocol_SendMessage(sizeof(channel), ID_ADC_SELECT_CHANNEL_RESP, &channel);
                    break;
                case (ID_ADC_FILTER_VALUES):
                    Protocol_GetPayload(&filters);
                    Protocol_SendMessage(sizeof(filters), ID_DEBUG, &filters);
                    Protocol_SendMessage(sizeof(filters[10]), ID_DEBUG, &filters[10]);
                    for (i=0; i < FILTERLENGTH; i++)
                    {
                        filters[i] = Protocol_ShortEndednessConversion(filters[i]);
                    }
                    Protocol_SendMessage(sizeof(filters), ID_DEBUG, &filters);
                    Protocol_SendMessage(sizeof(filters[10]), ID_DEBUG, &filters[10]);
                    operationStatus = ADCFilter_SetWeights(channel, filters);
                    if (operationStatus == SUCCESS)
                    {
                        operationStatus = Protocol_IntEndednessConversion(operationStatus);
                        Protocol_SendMessage(sizeof(ADCFilter[channel]), ID_DEBUG, &ADCFilter[channel]);
                        Protocol_SendMessage(sizeof(ADCFilter[channel][10]), ID_DEBUG, &ADCFilter[channel][10]);
                        Protocol_SendMessage(sizeof(operationStatus), ID_ADC_FILTER_VALUES_RESP, &operationStatus);
                    }
                    break;
                case (ID_LAB3_SET_FREQUENCY):
                    Protocol_GetPayload(&CurFrequency);
                    CurFrequency = Protocol_ShortEndednessConversion(CurFrequency);
                    sprintf(message,"Current Freq: %i", CurFrequency);
                    Protocol_SendDebugMessage(message);
                    FrequencyGenerator_SetFrequency(CurFrequency);
                    break;
                case (ID_LAB3_FREQUENCY_ONOFF):
                    Protocol_GetPayload(&freqState);
                    if (freqState)
                    {
                        FrequencyGenerator_On();
                        sprintf(message,"ON");
                        Protocol_SendDebugMessage(message);
                    }
                    else
                    {
                        FrequencyGenerator_Off();
                        sprintf(message,"OFF");
                        Protocol_SendDebugMessage(message);
                    }
                    break;
                }
            }
//            Protocol_SendMessage(&ADCData[channel], ID_DEBUG, &ADCData[channel]);
//            adcValues.filtered = ADCFilter_FilteredReading(channel);
            adcValues.unfiltered = Protocol_ShortEndednessConversion(ADCFilter_RawReading(channel));
            adcValues.filtered = Protocol_ShortEndednessConversion(ADCFilter_FilteredReading(channel));
            Protocol_SendMessage(sizeof(adcValues.aschar), ID_ADC_READING, &adcValues.aschar);
//                LEDS_SET(0x0);
        }
    }
    return 0;
}
#endif
```