Jacob Sickafoose

Lab # 3

3/3/2020

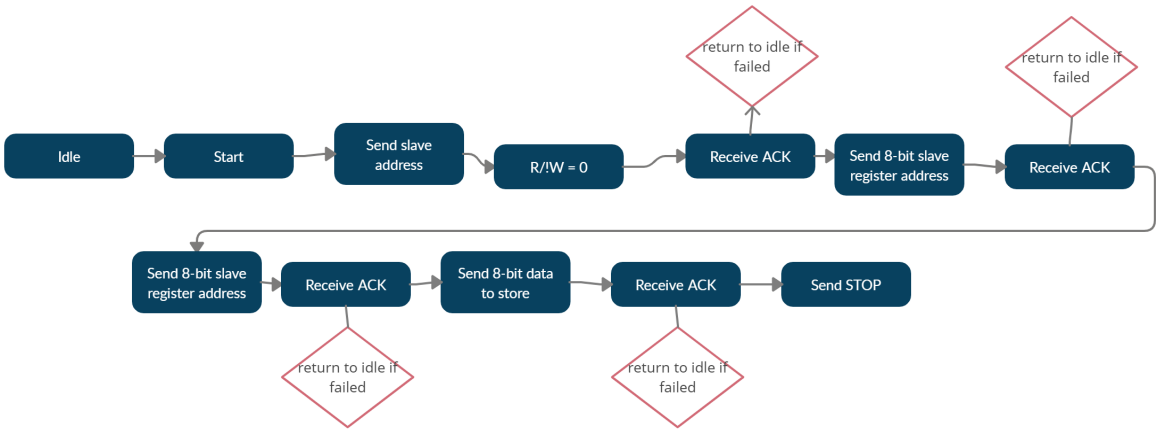
Description -

This lab required using the EEPROM on board the I/O shield on the board to read and write bytes and pages. Communication between the processor and the NVM required use of the I2C bus module. This was implemented in the NonVolatileMemory.c library. The lab also required setting up the ADC module to take in an analog reading and apply a digital filter to it. These analog readings were taken in through the potentiometer, as well as the frequency generator. This was all done in the ADCFilter.c library. Finally the application put the two together by loading in the filter values from the NVM on start of the program. It also allowed the user to send new values, and store those to overwrite the old ones in the NVM. There were also multiple channels of analog input that were all being kept track of, and displayed one active one at a time.

Method -

Non Volatile Memory -

The NVM chip on the I/O shield was communicated with using the onboard I2C module. An I2C transaction consists of different steps with signals being sent, and waiting for a response from the slave. This was the state machine utilized for the transactions for the 1st pre lab part:



Because the flow of instructions was different between the different operations, I found it was much less confusing to just have the steps separate, yet clearly labeled with comments rather than to use a switch statement. I also found it easier to start by writing the ReadByte function rather than anything else, because supposedly reading was harder than writing and it was easy to run the precompiled hex to write a byte, then test my read function knowing what I should be receiving. Once I figured out to use an if-statement to check for ACK rather than using a while loop to wait for it, I was able to get read byte working fine.

The WriteByte function was easy too, I just needed to implement slightly different steps from the lab manual. To answer the pre lab part 2, I used the sequence we were given of

1. Send START
2. Send device address
3. Send write flag R/!W = 0
4. Receive ACK
5. Send memory address high byte
6. Receive ACK
7. Send memory address low byte
8. Receive ACK

9. Send Data to be stored in address location
10. Receive ACK from the device
11. Send STOP

If at any point I failed to receive an ACK in enough time, I would send STOP and abort the function. To answer pre lab part 3, for my single byte read I did the same thing until part 8 where instead of sending data, I sent the following:

1. Send repeated START
2. Send device address with R/!W = 1
3. Receive ACK
4. Receive NACK to the device
5. Send STOP

The read and write page functions were mostly the same, other than needing to loop through sending/receiving data, and sending ACK. I did have an issue with my ReadPage, where I accidentally still sent an ACK when I was done receiving data, then sent the NACK which resulted in locking up the EEPROM and requiring a power reset. I fixed this by adding an if-statement to when I send ACK, making sure I don't send one on my final loop through.

ADCFilter -

The ADC filtering module required applying a filter algorithm to the input values from the ADC module onboard the PIC32. To start out, the ADC module was initialized to use the 2, 4, 8, and 10 pins to read from as analog input. It was also set according to the steps in the manual.

1. Connect the desired pins to the ADC
2. Add those pins to the scanner

3. Set the ADC to auto-sample, auto-convert, and 16 bit unsigned mode
4. Set the ADC to use internal voltage references, scan mode and to interrupt after each set of measurements
5. Set the ADC to use 16-bit buffers, set T_{AD} to use the peripheral bus clock
6. Set the ADC T_{AD} to 348 = T_{PB}
7. Set the sample time to 16 = T_{AD}
8. Turn the ADC on, and enable the interrupt

As an answer to pre lab part 4, I ended up setting the ADC1CON3bits.ADCS register to (348 / 2) - 1 = 173 according to the formula given in the FRM. I had to subtract 1 from the result of 348/2 to account for the value index starting at 0. I set the AD1CON3bits.SAMC register equal to 16 to set T_{AD}. I also used ADC pins, 2, 4, 8, and 10.

Two, 2-Dimensional arrays also needed to be initialized. One storing the input raw data per input channel, for a given filter length. The next array was storing the filter values to apply our filter to the same respective data values per channel, for each of the values down the filter length. Inside the ADC interrupt, the new values received from the ADC module were stored into the data array at each of the pins, overwriting the previous data when it hit the end of the filter length. Running the RawReading function simply returns the value at the given index of the last data reading. Running the SetWeights function would store the given array of filter values into the correct filter array, based on the given channel. Returning the filtered reading involves running ApplyFilter which multiplies the filter values with their respective raw data inputs, then summing all the products, and dividing by 2^{15}. This value is then returned as the filtered reading. I was running into a problem where feeding any channel the frequency generator's output in testing, would give me unexpected results. When I changed the frequency to a high frequency,

and the filter to high pass, rather than the filtered value being close to the raw value, I would only get 0. This is documented in my check off videos and I was sure to include a recording of running the precompiled ADC hex file, to show that it reacts the same way. Other than that, I encountered no issues with ADCfilter.

Lab 3 Application -

The final application needed to use the SetWeights function by reading pages from NVM for each of the channels on startup. It also needed to rewrite these pages with new weights when given the command from the interface to overwrite the stored filter type. Each channel needed to store two different filters. These filters were switched between using SW3 on the board. This also meant that 8 pages total needed to be used in the EEPROM to store 2 filter weights, per the 4 channels. The channels were changed using SW0 and SW1 on the board or using the interface. The LEDs on the board needed to display either the absolute value of the filtered reading, or calculated peak to peak values as a bar graph. The value displayed was also controlled by a switch on the board. This one with SW3. For calculating the LEDs to light up with the absolute value reading, I just took the filtered value and divided it by 127 to turn the 0 - 1023 range of the values to 0 - 8 of the LEDs. For the peak to peak I did the same thing but I first had to calculate the peak to peak value. I did this by storing 69 filtered values and every 10ms, a loop would iterate through them to find the min and max values. After the loop, the peak to peak value would be calculated and turn on the appropriate LEDs. I also had an if statement detect when a change in the switches occurred so it could send an update message to the interface.

Conclusion -

The NonVolatileMemory.c library now initializes the on board I2C module and uses I2C transactions to interact with the EEPROM on the I/O shield and implement the functions

necessary. It can now Read and write bytes as well full pages of 64 bytes. The test harness simply uses the interface buttons to read and write bytes and pages to whichever address is given. The only snag encountered was with accidentally sending an extra ACK, then the final NACK at the end of ReadPage, causing the EEPROM to get into a stuck state.

The ADCfilter.c library now initializes the on board ADC module and reads the analog input from AN0-AN3 to implement the functions. The RawReading function returns the last read raw input value, as stored in the internal 2D array. The FilteredReading function applies the filter to the current raw reading by running ApplyFilter, and returns the filtered reading. The SetWeights function sets the weights from the given input array, into it's respective filter array for the given pin/channel.

The Lab3Application.c is able to boot up with the previously stored filter values, as well as overwrite them with new ones based on input from the interface. It can switch between the two stored filters per channel with SW2. It can switch between channels using a combination of SW0 and SW1. The application also displays either the peak to peak of the filtered values, or the current reading based on SW3 on the LEDs.