ECE341

**Lab8 - I2C and EEPROM Communication**

Report

**Seth Cram**

**11/09/2021**

**Introduction:**

Goal:

To delve into synchronous communication through using a master-slave multi-drop network communication protocol called I2C.

Background Information:

We’ll use the I2C protocol to communicate with the serial EEPROM. I2C stands for Inter-Integrated Circuit protocol. I2C is also a serial protocol. We’ll need a synchronizing clock as a separate signal for I2C. I2C is supported by hardware in a large number of microprocessors. I2C is half-duplex since its data line is bi-directional, but can’t send data in both directions at the same time. For this lab, the PIC32 will act as the master as the EEPROM acts as the slave, so we’ll only have one slave but we could theoretically add more onto the I2C bus. The master doesn’t have to stay the same the entire time and can change, but it’ll only be the PIC32 for our purposes.

Each slave has a unique device ID number, addressable through the 7-bit address sent from the master. An I2C Frame consists of a Start condition, four bit control code, three bit chip select address, a read/write, and an acknowledge bit. The transfer ends with a Stop condition from the Master. The acknowledged bit must be set low by the receiving device after a data transfer. Otherwise, an error occurred with the transfer. The read/write line indicates the direction of the next data transfer. After an I2C frame, we can read or write successive bytes.

The I2C frame occurs on the SDA (Serial Data Line), with each bit separated by the SCL (Serial Clock Line) going low. SDA goes high to low for a start and low to high for a stop, all while SCL is high. Otherwise, SDA can't change while SCL is high. Which makes our communication synchronous. The most significant bit is sent first.

The I2C controller will help us satisfy timing constraints. The slowest device in the I2C network will dictate the maximum data transfer rate. When the memory chip encounters an error, it won’t be readily apparent. We might have to cycle the power. We’ll only be able to tell from the lack of an acknowledged bit when writing to it.

The EEPROM (electrically erasable programmable read only memory) is nonvolatile, so it still saves data even when powered off. It’s capable of storing 256 Kbits of data, or 32,768 bytes. Writes are slow because the data we write is first stored in the Page Latches before being placed into the EEPROM array. Reads are fast because they don’t require the middle man. The EEPROM array is where the addresses of the memory chip are located.

We will poll the acknowledge bit to know when the receiver is ready for a data transfer. This will be done by repeatedly writing to the EEPROM and seeing whether the acknowledge is set low. To end a data transfer, the receiver needs to not set the acknowledge bit low. This is referred to as “NACK” or no acknowledge. Otherwise, data bytes can be read/written back-to-back. The source of a NACK could be the device is busy, isn’t functional, or an incorrect device address was sent.

We’ll create an EEPROM device driver to write and read a reasonable number of bytes to/from the EEPROM in succession. For this, we’ll have to take pages into account. The EEPROM’s page latch is 64 bytes. When writing information into the EEPROM, we’ll have to take into account that it stores this information in its page latch, and we can’t write to it when it commits this information from the page latches into its actual memory array. This commitment process happens at the end of each page.

*Background information is from lecture notes and the lab 8 handout.*

Plan:

I’ll base my design off of the code supplied to us during the first week of this lab. With this method, the polling of the EEPROM to check if it's ready for another data transfer was pretty much already done. We’d only have to add a counting variable to exit the loop when too many writes have been attempted.

Moving onto the EEPROM read function, I’ll do some error checking before reading any data. First, I’ll verify the memory address is within range by comparing its specified page number to the maximum page numbers addressable. Next, I’ll make sure length is positive, and the passed in array isn’t NULL. Finally, I’ll use the code supplied for a read operation. Instead of writing a set number of bytes, I’ll individually write out the control bytes and memory address, then start looping for however many bytes I’ll need to read-in. Some error checking done in the loop include the current array position being inaccessible, the page number becoming too large, and the length not reaching zero so we can trigger an acknowledgement. We’ll issue a no-acknowledge and stop if an error occurs, or if we’ve read in all the requested bytes.

For the EEPROM write function, we’ll have to consider page numbers more integrally. The same error checking as for the read function should be implemented. Following the same structure as the reading function, we’ll differentiate by not changing the bus direction, and allowing the EEPROM time to commit the data passed into the page latches when a different page is crossed into. When considering pages we’ll need to use a counting variable initialized to the memory address and increment it everytime we write data. Unfortunately, there’s no way for us to check the offset within a page directly, so we keep track of it manually.

During the test file, we’ll have to initialize a variety of test variables to both pass into the read and write functions, and keep track of the error codes they return. First, we’ll fill our test array with the same character using a for loop. Then, we’ll use the above write function to commit the entire array to the EEPROM’s memory array. We’ll have to pass the length of our array, our array, the desired memory address to write to, and the EEPROM’s slave address. In return, we’ll get the write error code. We also pass in the same parameters to the read function, but a different array that’s empty and receives the read error code. Finally, we’ll loop through each individual array entry and verify that the data received is the same as the data written out to the EEPROM. If so, we’ll output a success to the LCD, otherwise we’ll output a failure. We could also print the error codes after some delay, if desired.

*What if the starting address is at the last byte in a page and the data is two bytes long? What happens if the data size exceeds two pages? What happens when the data length is less than a page length, but crosses a page boundary?*

I implemented a check in my write function to wait and poll the acknowledge flag whenever a new page is crossed into before continuing the write.

*What happens if the data has a length of zero? Or the pointer to the data is NULL?*

At the top of both my read and write functions, I had error checking for both. Whichever error occurred, a macro was returned that indicated the offending error.

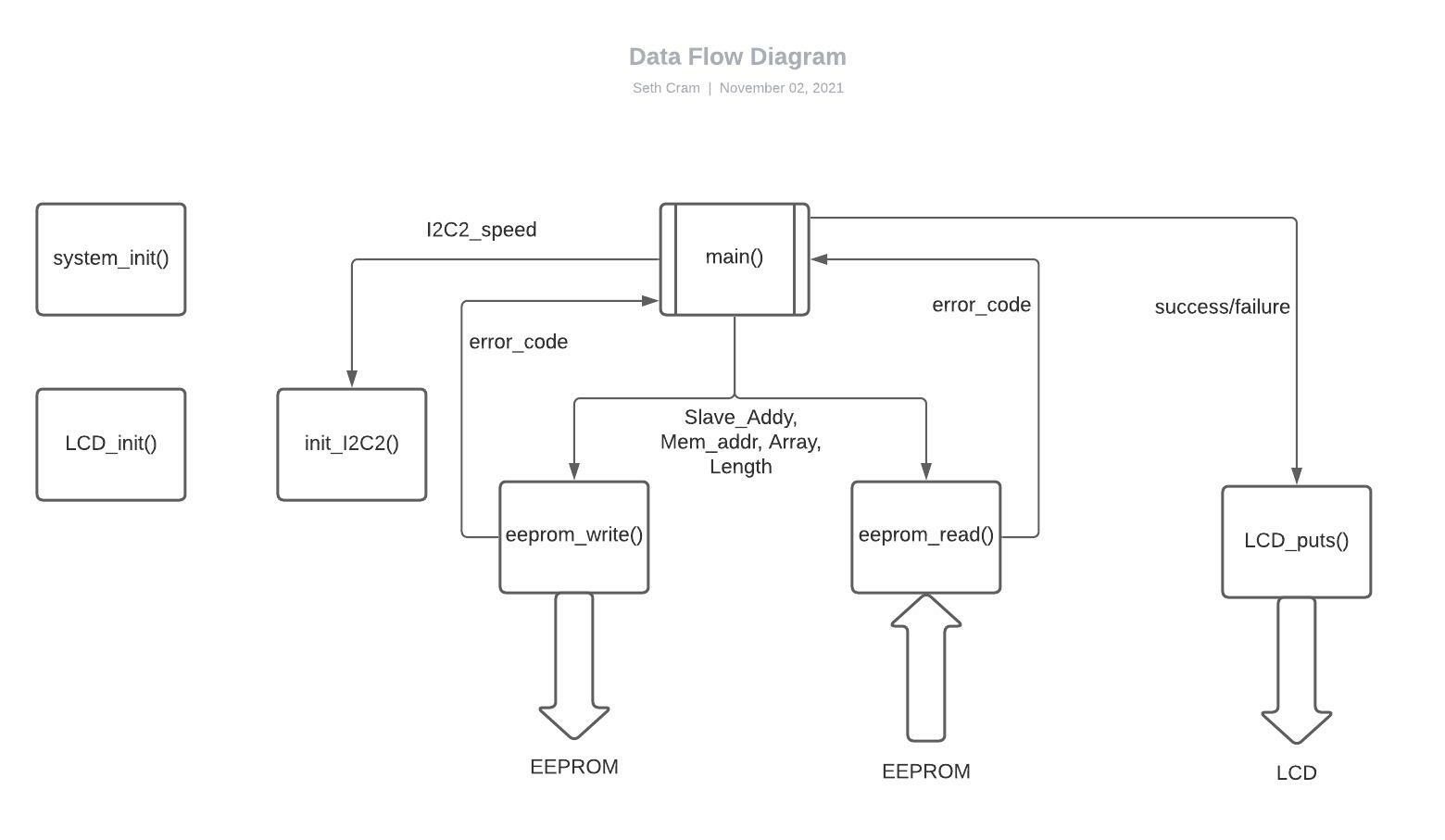
*What happens if the memory address is invalid?*

I implemented a check for the maximum memory address allowed at the top of both the read and write functions, and whenever a new page gap is crossed in my write or a new byte is read in my read, it’d verify the memory location is still valid. Otherwise, it’d return the appropriate error code.

**Implementation Discussion:**

Other possible errors were taken into account, such as the array passed into the read or write function being too small to account for the length of the data needed to be read/written. This was done by checking whether the array location being read/written is NULL and erroring if so.

Before implementation, I designed a data flow diagram to get a visual of what files I’d need to incorporate and their overarching functioning in the grand scheme.



Moving onto the implementation of code from start to finish, I began by moving functions created during week one of lab 8 into an EEPROM device driver library. For this library, I included its corresponding header file.

**Listing 1. I2C\_EEPROM\_LIB.h:**

A wide variety of macros and function prototypes are used in the library header file. For example,

*//prototypes:*

*void I2C2\_Start\_Condition();*

*void I2C2\_Stop\_Condition();*

*void I2C2\_Nack();*

*void I2C2\_Ack();*

*void I2C2\_Restart();*

*//macros:*

*#define EEPROM\_I2C\_ADDY 0x50 //cntrl byte shifted right once*

*#define SLAVE\_SHIFT 1 //however much we shift slave addy left by*

*#define PAGE\_LENGTH 64 //bc each page is 64 bytes, 512 bits, which is 2^6*

*#define PAGE\_OFFSET\_BITS 0x003F //6 lsb*

*#define PAGE\_NUMBER\_BITS 0xFFC0 //9 msb*

*#define SHIFT\_BYTE 8 //amt to shift a byte to switch it to the MSB or LSB*

*#define SHIFT\_PAGE\_NUMS 6 //amt to shift page numbers over to get rid of offset*

*#define MAX\_BYTES 32768 //this is 2^15 bc the 16th bit of addy not used*

*#define Fsck 400000 //desired baud rate*

*#define WRITE 0*

*#define READ 1*

*#define LOWER\_BYTE 0x00FF*

*//error codes:*

*#define NO\_ERR 0*

*#define TIMEOUT\_ERR 2*

*#define WRITE\_ERR 4*

*#define NULL\_PNTR\_ERR 8*

*#define ARRAY\_SMALL\_ERR 16*

*#define FAULTY\_LEN\_ERR 32*

*#define MEM\_UPPER\_BOUNDS\_ERR 64*

In addition to the obvious function prototypes needed for the init, read, write, and polling functions, I included helper functions which just did what their name proposes, in addition to idling after the requested action. I have a wide variety of macros included for increased readability, mainly for EEPROM device specifics and error codes. Using macros for each error received greatly increased my debugging capabilities, especially since each error is a single bit it made them easy to differentiate when multiple were thrown.

**Listing 2. test\_proj8.c:**

Now that we know some of the functions and macros we’re dealing with, we’ll delve into the test code. I declared a wide variety of variables at the top of my test file to decrease the number of ‘hard-coded’ values used. Some essential ones that took some care were:

*unsigned char i2c\_data[ LEN ];*

*unsigned char i2c\_byte[ LEN ]; //store enough data to be returned*

*unsigned int inCorrectEntries = 0; //0 for same data, non-zero for dif data*

*//for returned errors:*

*unsigned int write\_err = 0;*

*unsigned int read\_err = 0;*

For the arrays used in writing and reading the data to/from the EEPROM, I had to be sure that they were the same length. I also had to initialize the number of incorrect entries to zero. Later on, we’ll see how I use this variable to count the number of incorrect entries when comparing them after a read and write is done. Finally, the error variables are essential to finding out what went wrong.

First, I initialized the system, LCD, and the I2C bus, by opening it. Then I filled my array to write out to the EEPROM with a for-loop and incrementing the stored value every loop to get different values. Then, I wrote my filled array to EEPROM by passing in the slave address, memory location to write to, the filled array, and the filled array’s length.

*//Write LEN bytes from i2c\_data to EEPROM at mem\_addr:*

*write\_err = eeprom\_write( EEPROM\_I2C\_ADDY, mem\_addr, i2c\_data, LEN );*

*//read n byte from i2c\_data at mem\_addr:*

*read\_err = eeprom\_read( EEPROM\_I2C\_ADDY, mem\_addr, i2c\_byte, LEN );*

As seen above, I performed a similar operation with the reading of the EEPROM. Both my reading and writing stored an error code that I look at down below.

Now, we need to compare our read-in array to our written-out values, element by element.

*//check array if any bytes not the same tween read+write:*

*for( i = 0; i < LEN; i++ )*

*{*

*//if read in byte not same as og value:*

*if( i2c\_byte[ i ] != i )*

*{*

*//Data is different*

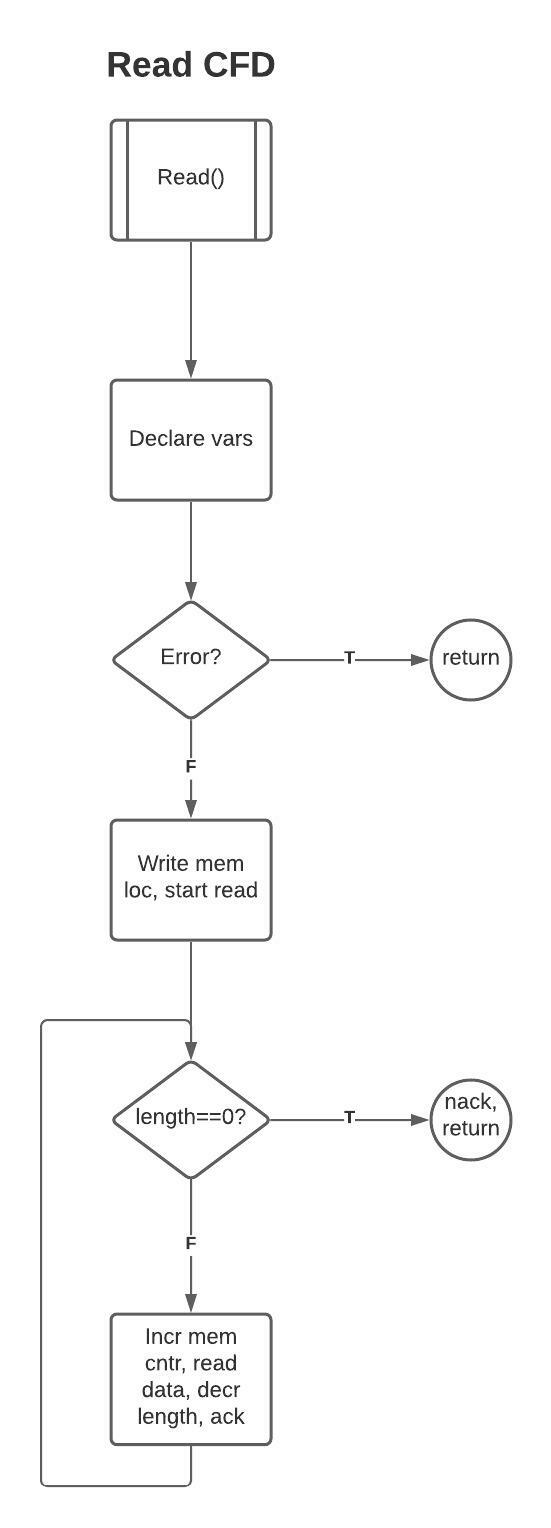
*inCorrectEntries++;*

*}*

*}*

As seen above, I used a similar for-loop to my last one, except every time the data wasn’t the same I’d increment the number of incorrect entries. Finally, if write\_err or read\_err is non-zero, I’d output a corresponding error message to the LCD. Then, if there’s more than zero incorrect entries I’d output an overall failure to the LCD, otherwise I’d output a success. In between these LCD writes, I had to make sure to delay, so they’d be readable to the human eye.

**Listing 3. eeprom\_read():**



Now it's time to get into the details of how I read from the EEPROM. First, I had to split the passed in memory address into an upper and lower byte. Then, I stored the current memory address I wanted to write to.

*unsigned int upper\_addr = mem\_addr >> SHIFT\_BYTE; //shift MSB to LSB*

*unsigned int lower\_addr = mem\_addr & 0x00FF; //mask off upper byte*

*unsigned int memAddyCntr = mem\_addr;*

After that, I performed a variety of error checking capabilities outlined in other portions of the report.

*//check if mem addr valid:*

*if( memAddyCntr > MAX\_BYTES )*

*{*

*//cant address it, so error*

*return MEM\_UPPER\_BOUNDS\_ERR; //mem addy too high*

*}*

*if( len < 1 )*

*{*

*return FAULTY\_LEN\_ERR; //faulty length error*

*}*

*if( i2c\_byte == NULL )*

*{*

*return NULL\_PNTR\_ERR; //Null pntr error*

*}*

Now that I was sure I was ready to initiate a read sequence, I sent the control byte to write, both memory bytes, and then started a read with another control byte. All of these stored their respective error codes in sticky bits. Now, in my while loop I looped until the passed in length variable was zero. First, I checked the array we’re dealing with isn’t already out of bounds.

*//if curr loc in array is NULL:*

*if( i2c\_byte[ len ] == NULL )*

*{*

*I2C2\_Nack(); //end read with a NACK*

*I2C2\_Stop\_Condition();*

*return ARRAY\_SMALL\_ERR | write\_err; // array too small error*

*}*

If so, I end the read by issuing a nack and return the appropriate error. After incrementing both the inex and stored memory address counter, I verified we weren’t yet at the maximum location in memory.

*//if next byte written is out of bounds:*

*if( memAddyCntr > MAX\_BYTES )*

*{*

*//stop reading and ret*

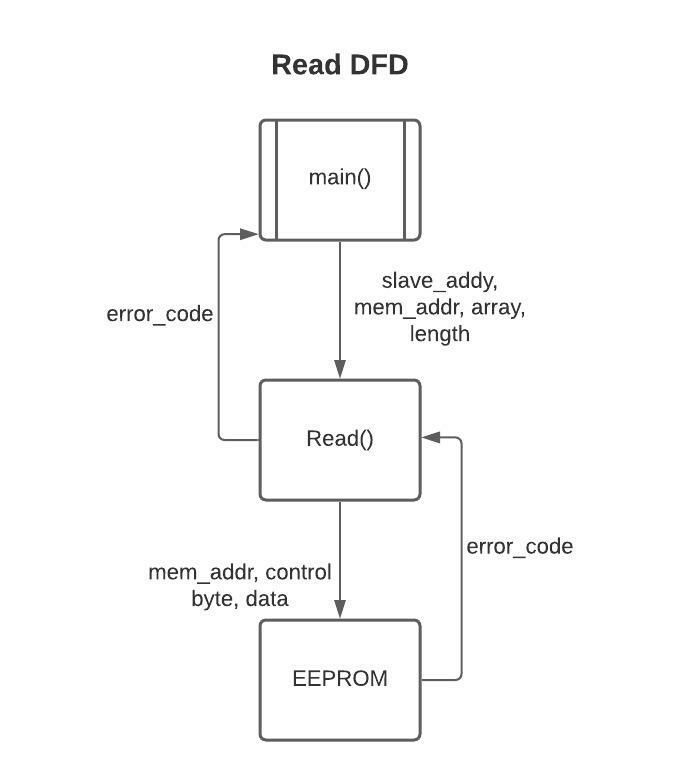
*I2C2\_Nack();*

*I2C2\_Stop\_Condition();*

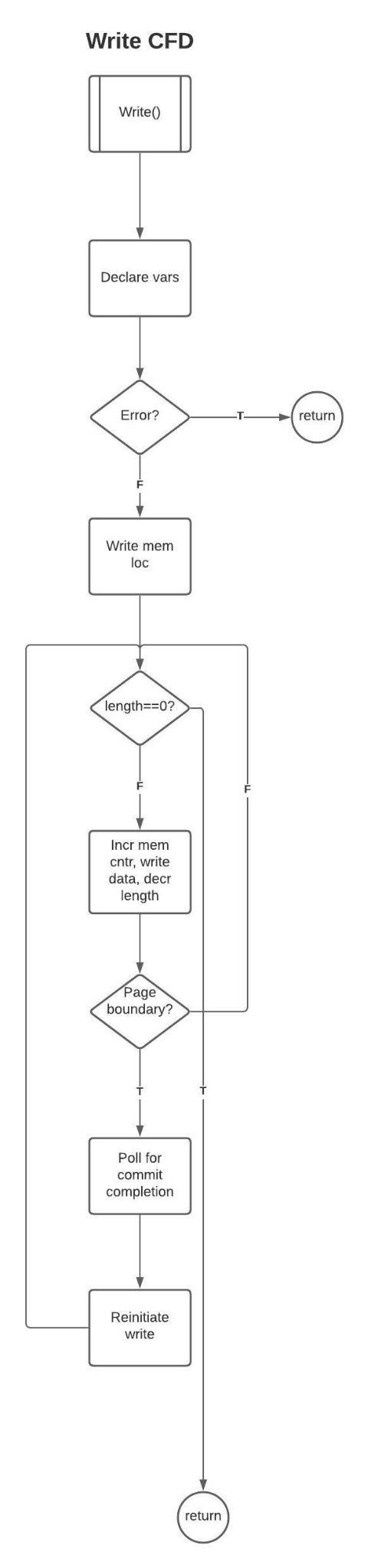
*return MEM\_UPPER\_BOUNDS\_ERR | write\_err; //mem addy too high*

*}*

Finally, if all went well we’d keep reading bytes until we’d had enough and then return with either a write error or no error at all.



**Listing 4. eeprom\_write():**



I had most of the same variables and did the same error checking as in my reading function. I even initiated my write the same with a control byte and the two memory address bytes, but then I entered my while-loop since I wanted to keep writing. The same error checking was done here, except now I only tested if we were at the maximum memory location after a page boundary was crossed.

*//if mem addy crossed over into another page:*

*if( memAddyCntr % PAGE\_LENGTH == 0 )*

*{*

*//stop transfer*

*I2C2\_Stop\_Condition();*

*//poll until data committed to mem:*

*poll\_err = wait\_i2c\_xfer( slaveAddy );*

*//if next byte written is out of bounds:*

*if( memAddyCntr > MAX\_BYTES )*

*{*

*return (MEM\_UPPER\_BOUNDS\_ERR | poll\_err | write\_err);*

*}*

*I2C2\_Start\_Condition();*

*//write cntrl byte:*

*write\_err |= MasterWriteI2C2( ( slaveAddy << SLAVE\_SHIFT ) | WRITE ); //write mem addy:*

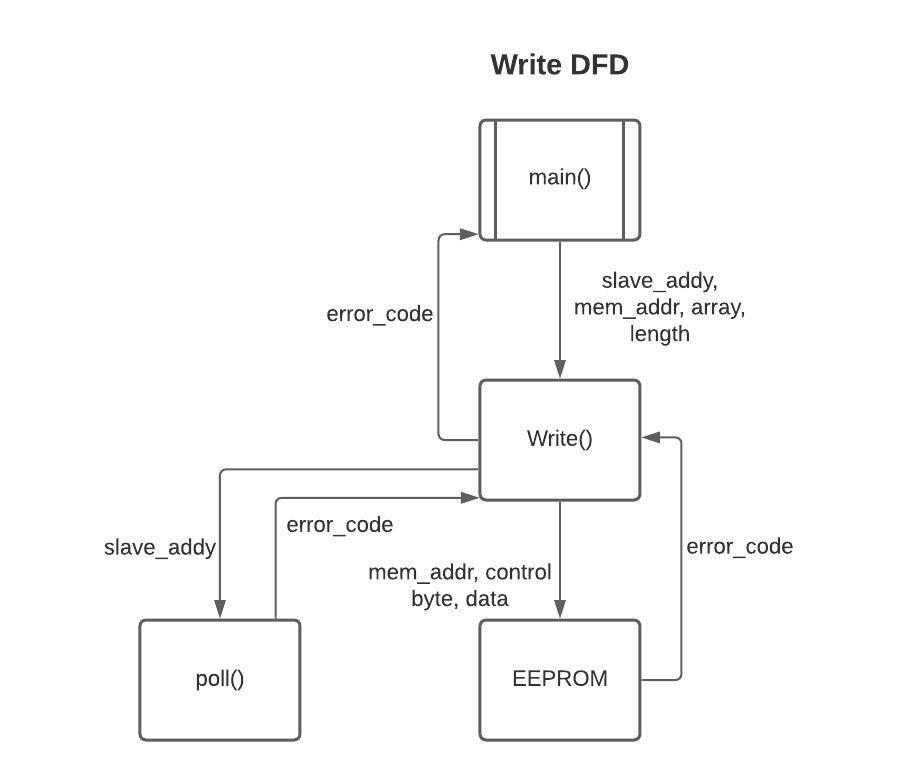
*write\_err |= MasterWriteI2C2( memAddyCntr >> SHIFT\_BYTE ); //upper byte*

*write\_err |= MasterWriteI2C2( memAddyCntr & LOWER\_BYTE ); //lower byte*

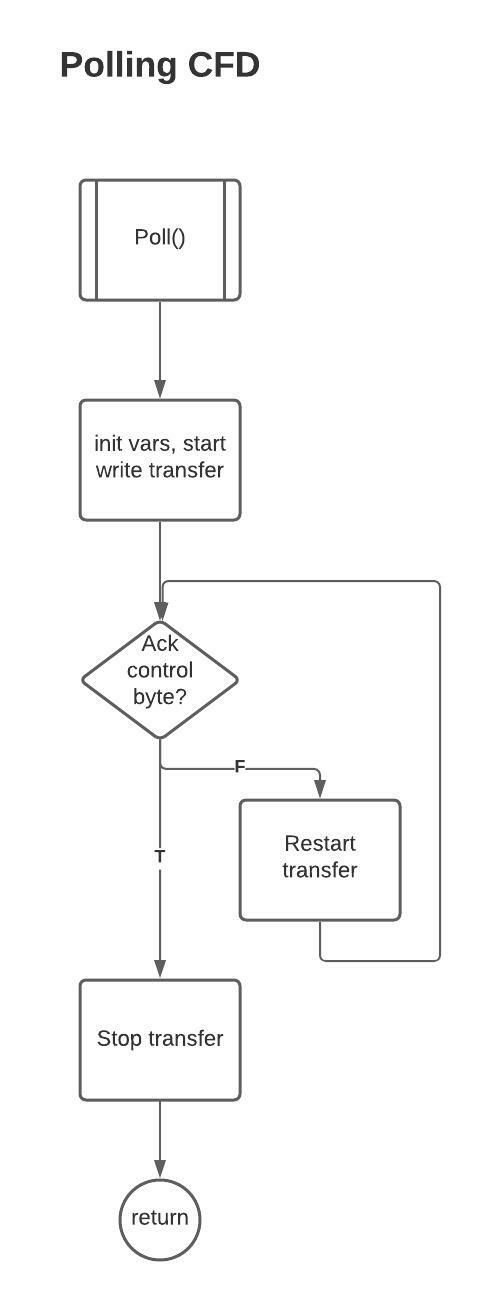
*}*

*}*

Checking if a page boundary has been crossed is an integral difference between the reading and writing functions. If so, we’d have to stop the transfer and poll until the page latches were committed to the memory array. Then, we’d have to begin a new write transmission at the same memory address as left off. Finally, if all went well I’d return with a polling, writing, or no error at all.



**Listing 5. wait\_i2c\_xfer():**



As seen in the write function, this function is used to poll until the EEPROM is ready for more writing. I implemented a maximum write number, so that if the polling function was waiting to receive an acknowledge for enough writes, it’d just exit the polling function and throw an error code.

*const unsigned int maxWrites = 1000;*

*while( MasterWriteI2C2( ( slaveAddy << SLAVE\_SHIFT ) | WRITE ) )*

*{*

*//if no ack:*

*//if too many writes attempted:*

*if( currWrite++ >= maxWrites )*

*{*

*//timed out, so write complete*

*I2C2\_Stop\_Condition();*

*return TIMEOUT\_ERR; //timeout error*

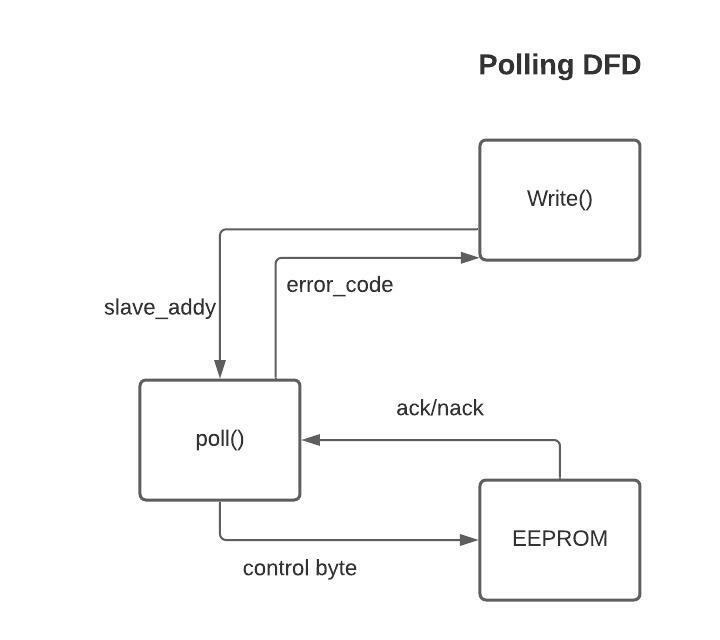
*}*

*//try restart if no ack*

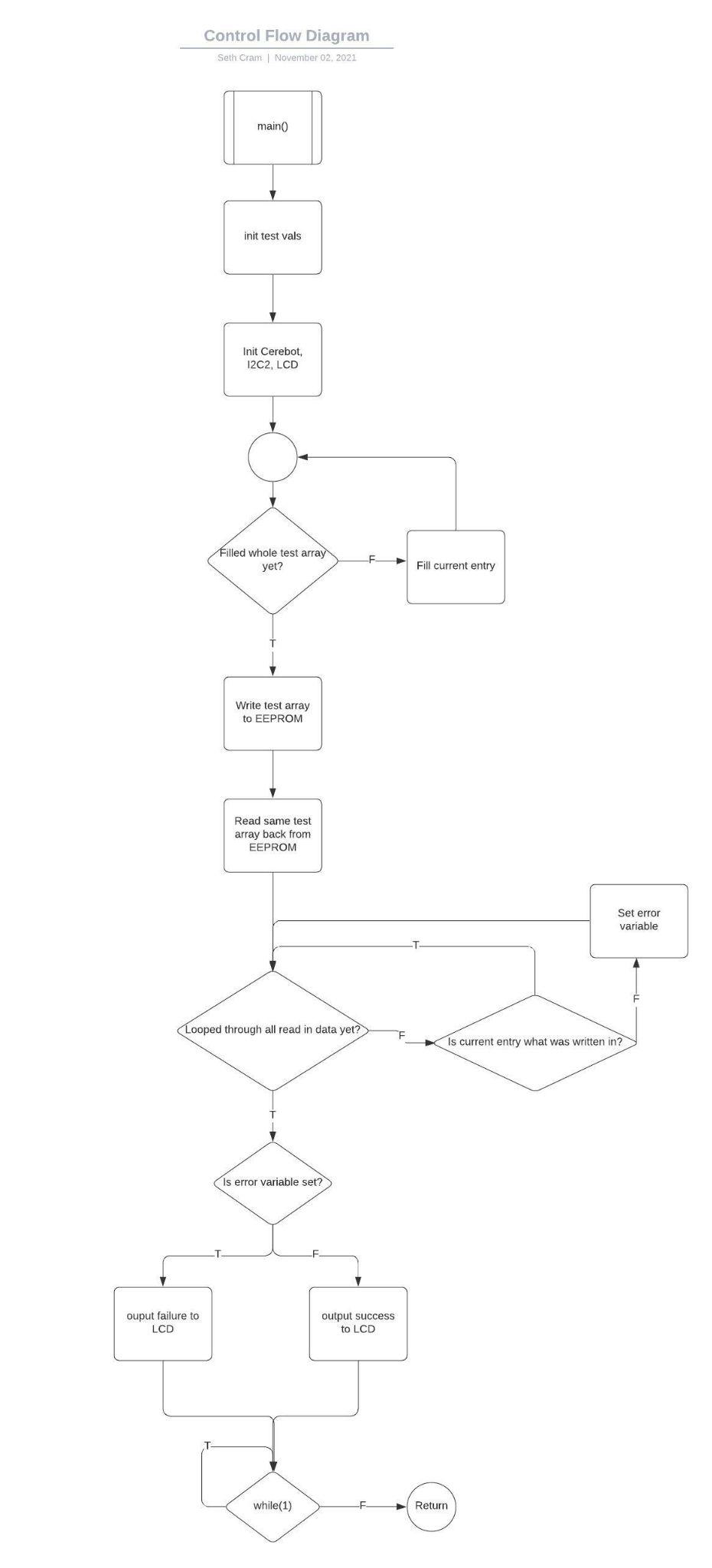
*I2C2\_Restart();*

*}*

I polled by repeatedly writing the control byte until either I timed-out or the control byte is acknowledged as received by the EEPROM. After that, I stopped the transfer and returned that no error was encountered. The rest of my helper functions aren’t worth delving into since they’re all just two lines of code with a condition then an idle.



Finally, my control flow diagram models the behavior of the above specified listings:



**Testing and Validation:**

I demonstrated to the TA through showing the contents of my receiving and writing array after writing and reading all the data, were the same. I also displayed a success or failure message to the LCD. After that message, I delayed for some time then would display whether there was a write or read error code present.

*Identify cases that need to be tested. For each case identified, describe the test you used to show that your library works correctly.*

All cases that were tested are described and covered above in the end of the introduction, the implementation discussion, and were performed when testing for the write and read time of a select number of bits below. These test cases are implicitly identified and performed through my mention of my error checking and debugging process. Some explicit test cases included: writing more than the maximum number of bytes, crossing a page boundary or multiple when reading and writing, passing invalid parameters to by library functions, and reading more bytes than written to the EEPROM.

Timing table:

| **Length (Bytes)** | **Write Time (ms)** | **Read Time (ms)** | **Write Data Rate (bytes/s)** | **Read Data Rate (bytes/s)** |
| --- | --- | --- | --- | --- |
| 1 | 3.176 | 0.136 | 314.861 | 7352.941 |
| 32 | 3.939 | 0.936 | 8123.889 | 34,188 |
| 63 | 4.702 | 1.736 | 13,398.6 | 36,290.32 |
| 64 | 4.840 | 1.762 | 13,223 | 36,322 |
| 65 | 7.901 | 1.788 | 8226.8 | 36,353.5 |
| 127 | 9.427 | 3.387 | 13,472 | 37,496 |
| 128 | 9.565 | 3.413 | 13,382 | 37,504 |
| 129 | 12.626 | 3.439 | 10,217 | 37,511 |
| 1024 | 75.441 | 26.325 | 13,573.5 | 38,898.4 |
| 8096 | 597.121 | 207.368 | 13,558.4 | 39,041.7 |
| 12,384 | 912.550 | 317.141 | 13,570.76 | 39,048.88 |
| 32,768 | 2428 | 848.802 | 13,495.9 | 38,605 |

*Describe how you made the measurements.*

Start each operation at the beginning of a page. Get accurate timing by typecasting operands to doubles. I used the core timer to read the start tick before an operation (write or read), then the end tick after the operation, and divide the difference between the two by the core timer ms tick rate to just get the ms that the operation took. In doing the operation at the end, I typecasted the operands to doubles to get a higher level of precision in my resulting measurement.

*startTick = ReadCoreTimer();*

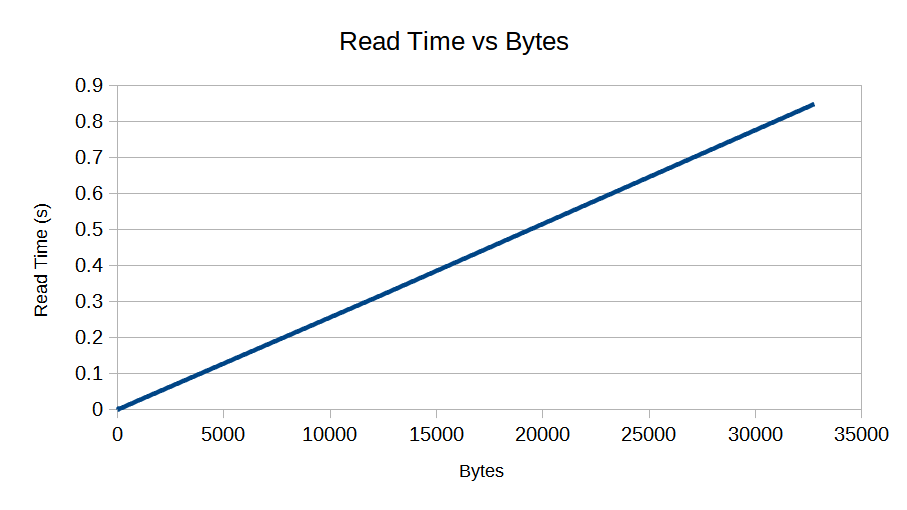
*/\*read or write operation\*/*

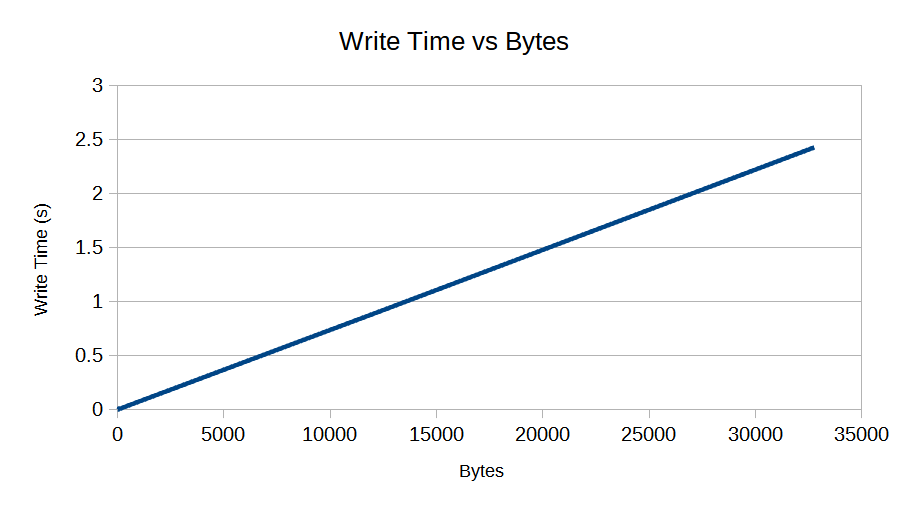
*elapsedTicks = ReadCoreTimer() - startTick; //elapsed time is in ticks*

*elapsedTime = ( (double) elapsedTicks ) / ( (double) CORE\_MS\_TICK\_RATE );*

*– Include a plot of the data.*

Other plots for the data weren’t included because they were viewed as not necessary due to their flat line-linear relations.





The y and x axis of these graphs could be easily flipped, which would show the data rate instead of the inverse of the data rate that they model now. The data was modeled in this way to find the expression for the approximate time required to write x bytes.

These graphs show that the more bytes written/read, the longer the corresponding operation will take. Looking at the y-axises, the write operation takes around three times as long as the read operation.

*– What is the average data rate when writing to the EEPROM? What is the average data rate when reading from the EEPROM?*

Average write data rate: 10,180 bytes per second

This was found through averaging the write data rate values in the table above.

Average read data rate: 34,884 bytes per second

This was found through averaging the read data rate values in the table above.

*– Develop an expression for the approximate time required to write x bytes, Twrite(x).*

Time to write a page was found to be 3.061 ms, found through subtracting the write time before and after a page gap is crossed. So, every additional page we write, we’ll need to add this additional delay. So, the ceiling of 64 (bytes/page) / x(bytes) should be multiplied by 3.041(ms), and added to the average time to write a byte multiplied by the number of bytes being written. Taking the ceiling of a calculation means to round up its result to the next highest whole number. To find the average time to write a byte without taking into account page gap commit time, we’ll have to find how many pages are used and subtract that many page commit times, then divide that by the number of bytes written. The resulting average write data rate without taking pages into account is about 26 ms.

*Twrite(x) (ms) = ciel( x/64 ) \* 3.041 + 0.026 \* x*

*– Develop an expression for the approximate time required to read x bytes, Tread(x).*

The average read data rate is 34,884 bytes per second, so the expression to approximate the time required to read x bytes would be the inverse of the average read data rate multiplied by the number of bytes to read:

*Tread(x) (us) = 28.7(us/byte) \* x(bytes)*

Aka

*Tread(x) (us) = x \* 28.7*

Don’t use a simple linear regression. Rather, determine the number of pages that need to be written, and determine how long it takes to write a page to the memory. Use that information as a starting point for your expression.

**Conclusion:**

In conclusion, I2C is a synchronous, serial communication protocol that used a master-slave multi-drop network to communicate. Although we only used a single slave device, more could be connected. Using the network, reading was fast but writing was slow due to the internal mechanisms of our slave device, the EEPROM. The EEPROM’s use of page latches limited the write speed. Since our communication is synchronous, that means we didn’t have to do any hand-shaking or synchronizing of clock domains, since the data transfer occurred in-line with clock cycles.

*On any bidirectional bus, there is a risk that two (or more) devices may try to place different voltage levels on the bus at the same time. When interfacing with the LCD, this problem was overcome with tristate outputs. How does I2C overcome this problem? In other words, how does I2C handle bus contention (where two devices attempt to drive the same wire (the SDA line, for example) to different voltage levels at the same time)? What are the consequences of this design? Why wouldn’t a tristate solution (like the LCD interface) work for I2C?*

When multiple devices are trying to write to the bus, the one writing more zeros wins. The consequences of this design is that if there’s another master, they’ll have to immediately discontinue any operations on the bus. As well as each master device has to be aware of when the network is in the control of another master device. So, each device must be able to recognise ongoing network communication and not interfere. This is implemented through waiting for a stop condition before attempting to write to a bus.

A tristate solution wouldn’t work because the line can’t be floating due to the resistor pull-up network. Without the resistor pull-up network, SDA could be floating. A floating SDA would cause a variety of errors and mess with the functionality of the protocol, since after the start and stop condition, as well as a variety of other points in communication, the SDA and SCL need to idle before writing or reading. Idle means both of these signals need to be settled in the high, or 1, state.