10/10

ECE 341 Lab 8

Zachary DeLuca

February 21st 2023

Introduction

This lab is meant to be an introduction to the I2C communication protocol, which is a synchronous serial communication protocol. The end goal of this lab was to create a driver like program that includes functions that ease the communication with the EEPROM (electrically erasable programmable read only memory). The functions created in the lab include being able to read and write from the EEPROM without having to worry about page boundaries, or keeping track of the address mid read or write process or deal with the communication protocol specifics. The other functions are just to initialize the EEPROM and create a wait phase. The wait phase is important for within the driver to keep data from being written to the page registers while the registers are trying to be loaded into the EEPROM's memory. The wait phase function most likely will not be used by anyone using the driver in their code.

Implementation

✓

The first of the functions that was required was the function to initialize the I2C. We simply set the baud rate and open the connection using the macros.

Listing 8.1

```
void init_I2C2(int SICK_FREQ){
   #define Fsck 40000
   #define BRG_VAL ((FPB/2/Fsck)-2)
   OpenI2C2(I2C_EN,BRG_VAL);
}
```

The next function is the read function, which was much more interesting, and as such we will start with the CFD and DFD for it, shown in figures 8.1 and 8.2. In the DFD the EEPROM is shown as a triangle to illustrate the timing of the data sent, with the longer lines being sent after the shorter lines.

Figure 8.1: Read CFD

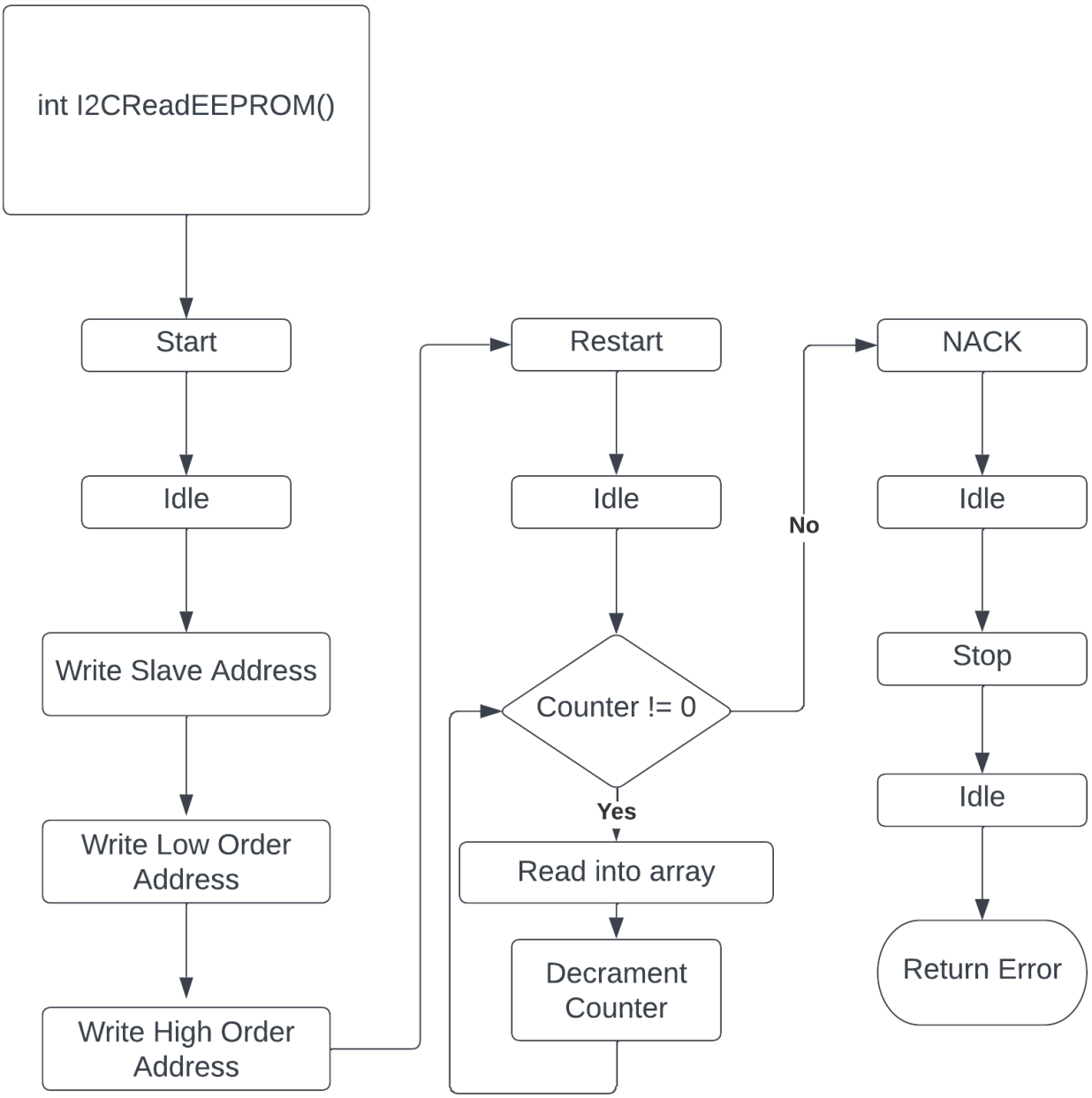
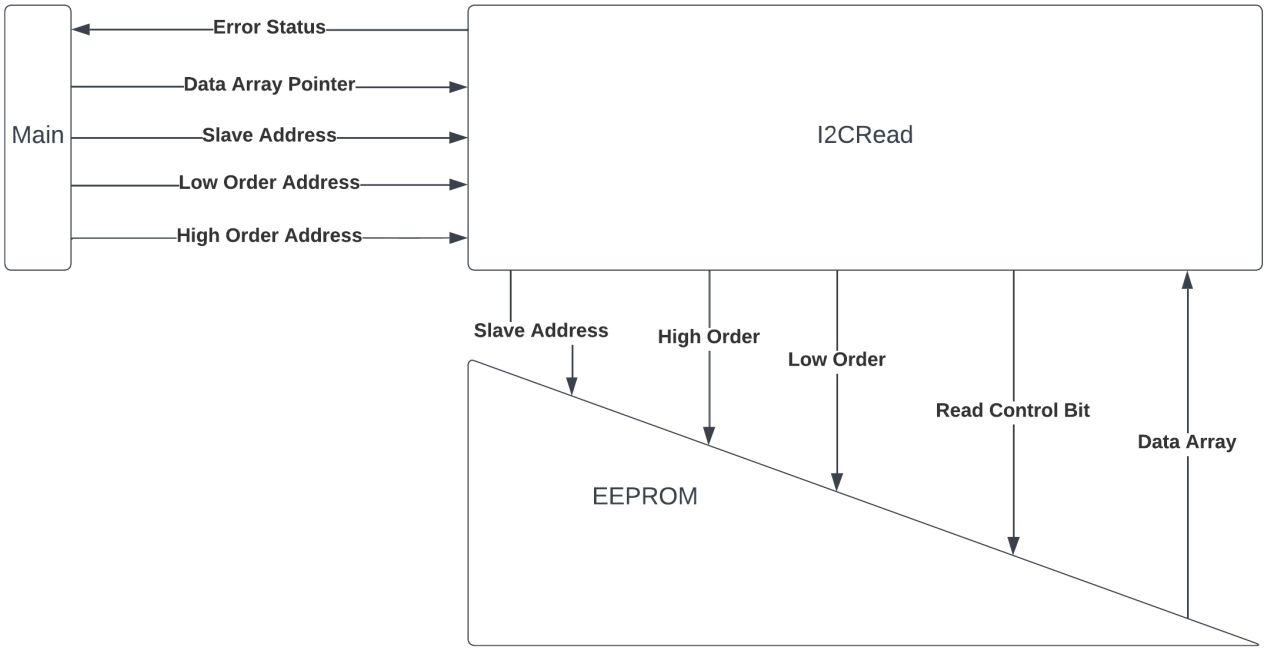


Figure 8.2: Read DFD



Listing 8.2

```
int I2CReadEEPROM(int slv_addr, int mem_addr, char *i2cData, int size){
   int write_err = 0;
   int count = size;
   int index = 0;

   StartI2C2();
   IdleI2C2();
   write_err = error_check(mem_addr,slv_addr);
   RestartI2C2();
   IdleI2C2();
   MasterWriteI2C2((slv_addr << 1 | 1));
   while(count--){
      i2cData[index++] = MasterReadI2C2();
      if(count){
         AckI2C2();
         IdleI2C2();
      }
   }
   NotAckI2C2();
   IdleI2C2();
   StopI2C2();
   IdleI2C2();
   return write_err;
}
```

In the code, there is a function called named error_check() which was meant to check for errors when using the MasterWrite macro to write the slave address, high order address and the low order address. As it does the write as it checks for them, the writing of the address bits has been allocated to this function, which is shown in listing 8.3

Listing 8.3

```
int error_check(int mem_addr, int slv_addr){
   char lsb = mem_addr & 0xFF;
   char msb = mem_addr >> 8;
   int write_err = 0;
   write_err |= MasterWriteI2C2((slv_addr<<1)|0);
   write_err |= MasterWriteI2C2(msb);
   write_err |= MasterWriteI2C2(lsb);
   return write_err;
}
```

This error check function is also used in the writing function that is shown below.

As the write function is a bit complex in the order in which the steps must proceed, a CFD has been constructed to illustrate the flow of the function: vspace12pt

Figure 8.3

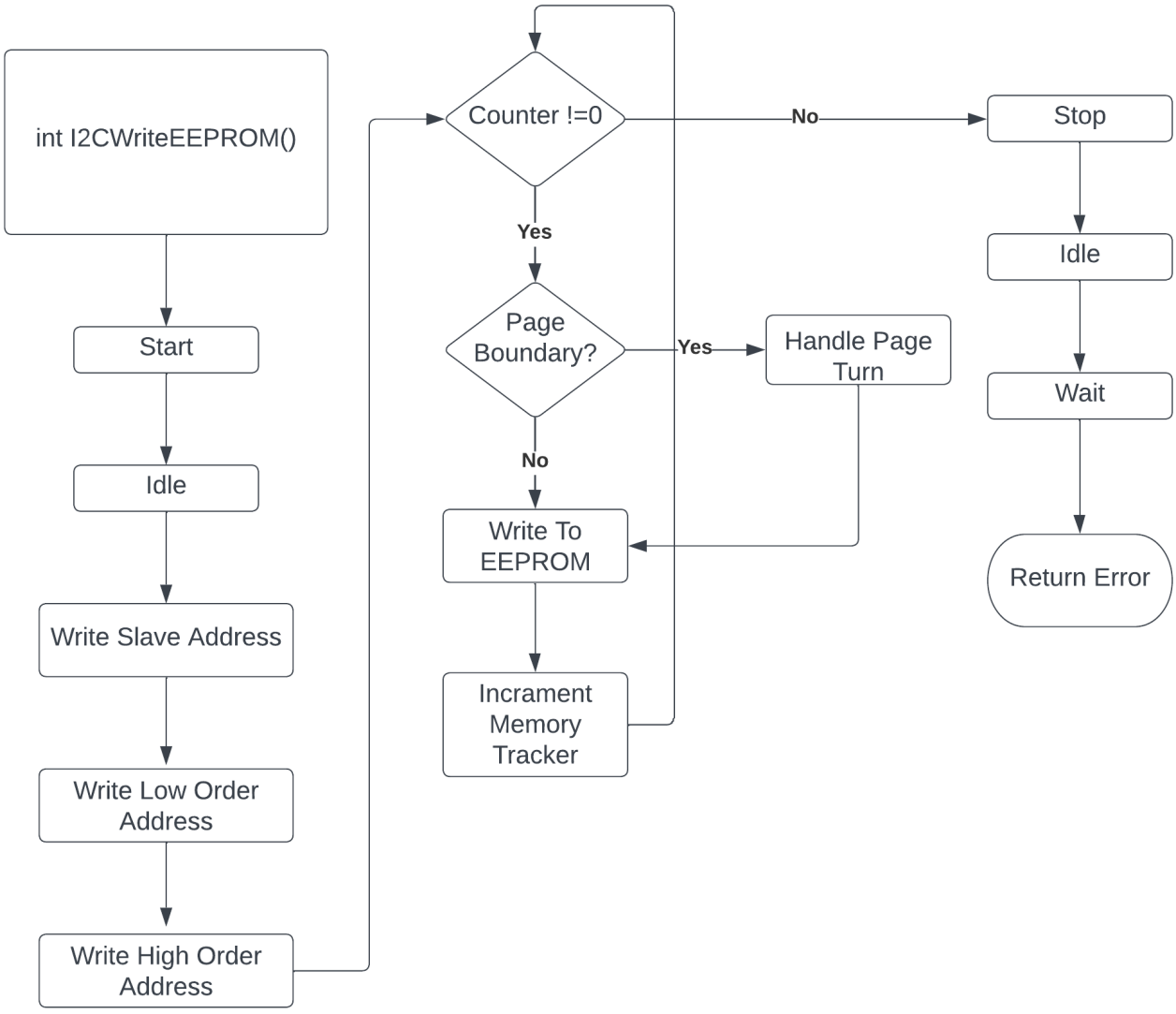
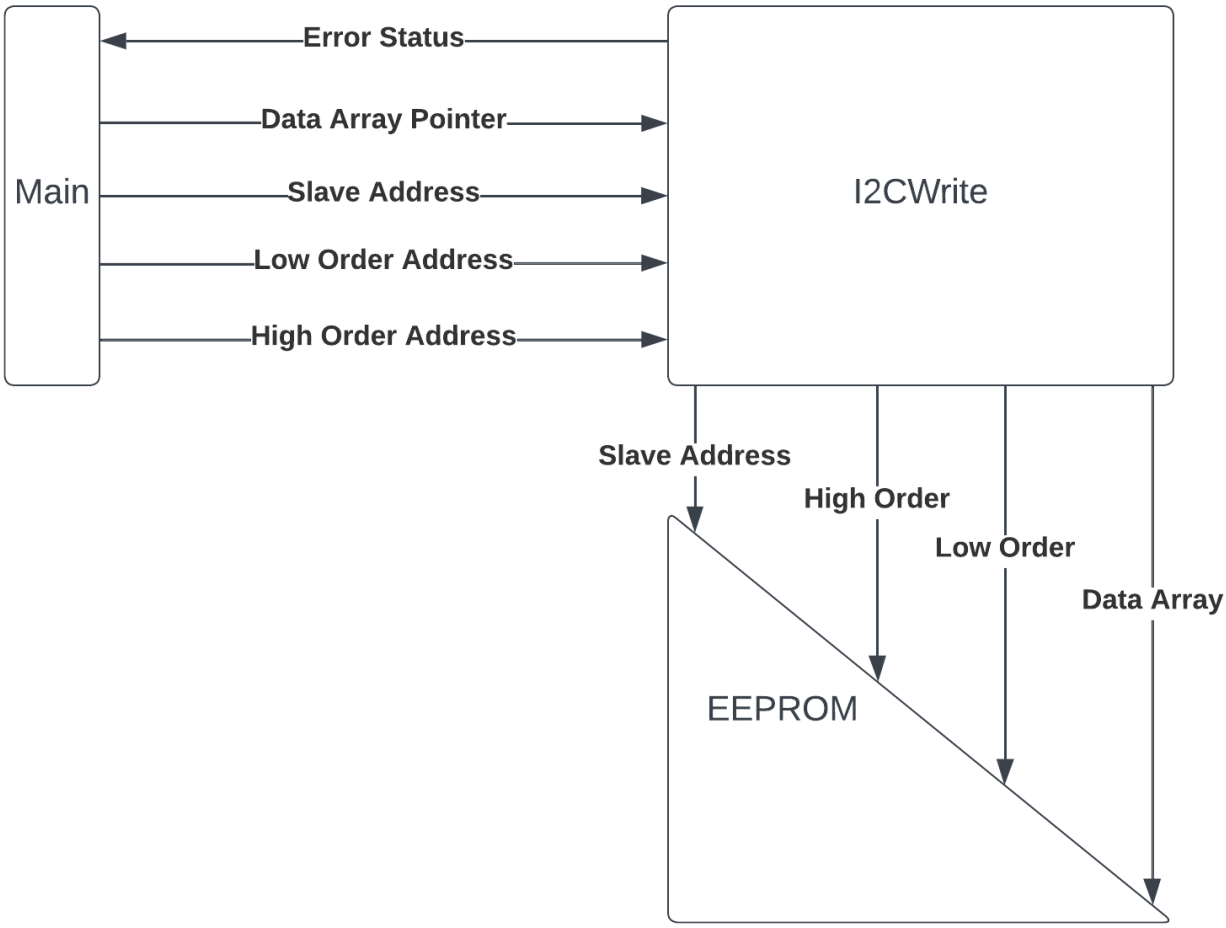

int I2CWriteEEPROM()
Start
Idle
Write Slave Address
Write Low Order Address
Write High Order Address
Counter !=0
No
Yes
Page Boundary?
Yes
Handle Page Turn
No
Write To EEPROM
Incrament Memory Tracker
Stop
Idle
Wait
Return Error

Figure 8.4: Write DFD



Before we can go into the write function we must first define the wait function. The wait function simply poles the EEPROM for write completion, so the program is blocked from proceeding until the ACK signal has been received.

Listing 8.4

```
int wait_i2c_xfer(int slv_addr){
   while(MasterWriteI2C2((slv_addr<< 1) | 0)){
      RestartI2C2();
      IdleI2C2();
   }
}
```

Listing 8.5

```
int I2CWriteEEPROM(int slv_addr, int mem_addr, char *i2cData, int size){

   int write_err = 0;
   int count = size;
   int index = 0;

   StartI2C2();
   IdleI2C2();
   write_err = error_check(mem_addr,slv_addr);
   if(write_err){
      return 1;
   }
   int check = 1;
   while(count){
      check = mem_addr%0x40;
      if(!check){
         pageTurn(mem_addr);
      }
      mem_addr++;

      write_err |= MasterWriteI2C2(i2cData[index++]);
      count--;
      if(!count){
         StopI2C2();
         IdleI2C2();

         wait_i2c_xfer(slv_addr);
      }
      if(write_err){
         return 1;
      }
   }
   return 0;
}
```

The magic step in the CFD that shows the page turn being handled was outsourced to its own function, shown in the next listing:

Listing 8.6

```
void pageTurn(int mem_addr){
   StopI2C2();
   IdleI2C2();
   StartI2C2();
   IdleI2C2();
   wait_i2c_xfer(0x50);
   char lsb = mem_addr & 0xFF;
```

```
    char msb = mem_addr >> 8;
    MasterWriteI2C2(msb);
    MasterWriteI2C2(lsb);
}
```

Testing and Verification

For the testing stage, we needed waveform verification of the read and write operation for a single bit.

Figure 8.5: Read Single Byte

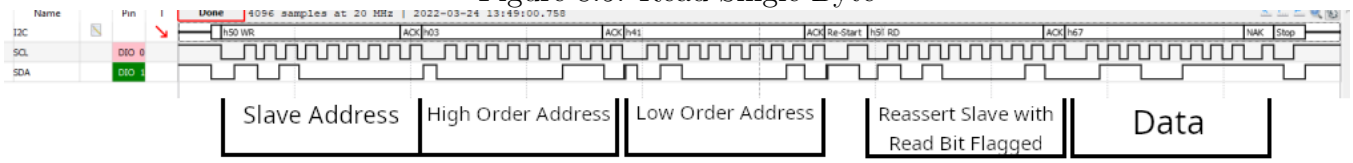
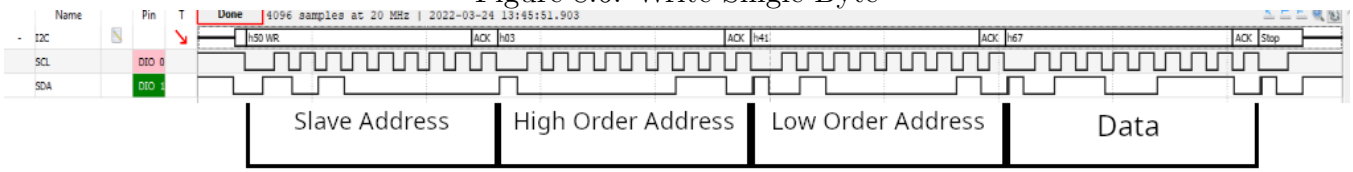


Figure 8.6: Write Single Byte



Reading and writing a single bit was easy, but the cases that needed to be watched for was writing to an invalid address, in either the low order, high order or the slave address. These errors were detected by the error_check() function and the return values of the read and write functions. This was less something to fix and more to be aware of when writing to each of the functions. The other issue was the page boundaries. These happened every 64 bytes in the memory of the EEPROM and would disrupt the write cycle if not handled. This was handled in this example in the pageTurn() function. The function was invoked every time the address tracker showed that it was the 64th bit, and it would stop the write cycle and let the page registers load into the memory. It would then restart the I2C write and return to the writing loop. To test this, a sample array of chars was written on the page boundary and read from, and each time was as success. The test success case was shown by the LCD and confirmed by the TA, and is unable to be shown in this document. The next thing to be collected was a table showing the time it took to read a certain amount of bytes and how long it took to write the same amount of bytes. The table below shows the time it took for each operation and the average rate of reading and writing per byte. The timing was found by checking the timer before and after each operation then multiplying that by the tick rate per ms as defined in the header file.

Figure 8.7

Bytes	Write Time (ms)	Read Time (ms)	Write Rate (B/ms)	Read Time (B/ms)
1	1	1	1	1
32	13	8	2.46153846153846	4
63	15	15	4.2	4.2
64	16	15	4	4.26666666666667
65	21	15	3.0952380952381	4.33333333333333
127	35	30	3.62857142857143	4.23333333333333
128	35	30	3.65714285714286	4.26666666666667
129	40	30	3.225	4.3
1024	309	235	3.31391585760518	4.35744680851064
8096	2474	1852	3.27243330638642	4.37149028077754
16384	5008	3748	3.27156549520767	4.37139807897545
32768	10015	7495	3.27189216175736	4.37198132088059

Figure 8.8: Read Single Byte

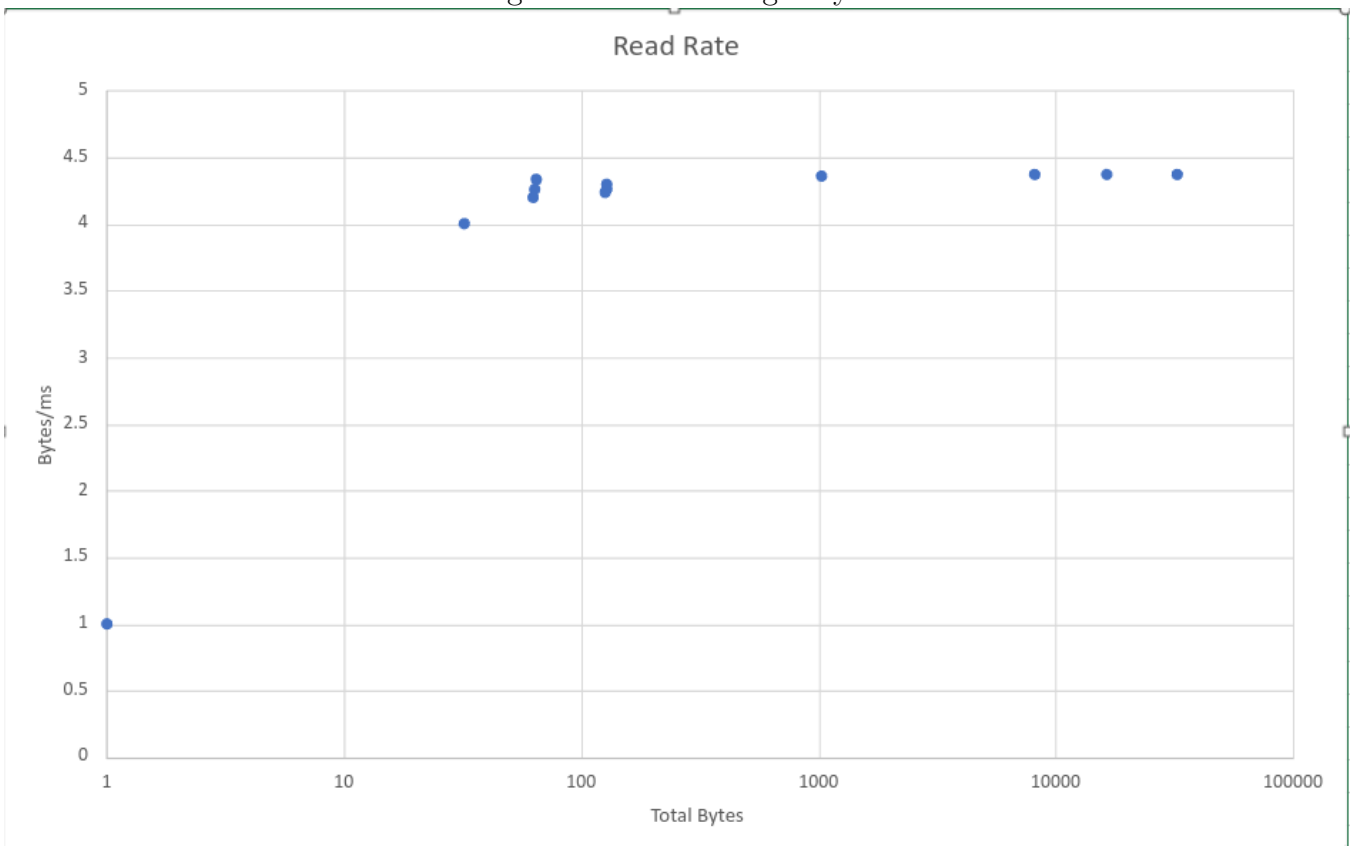
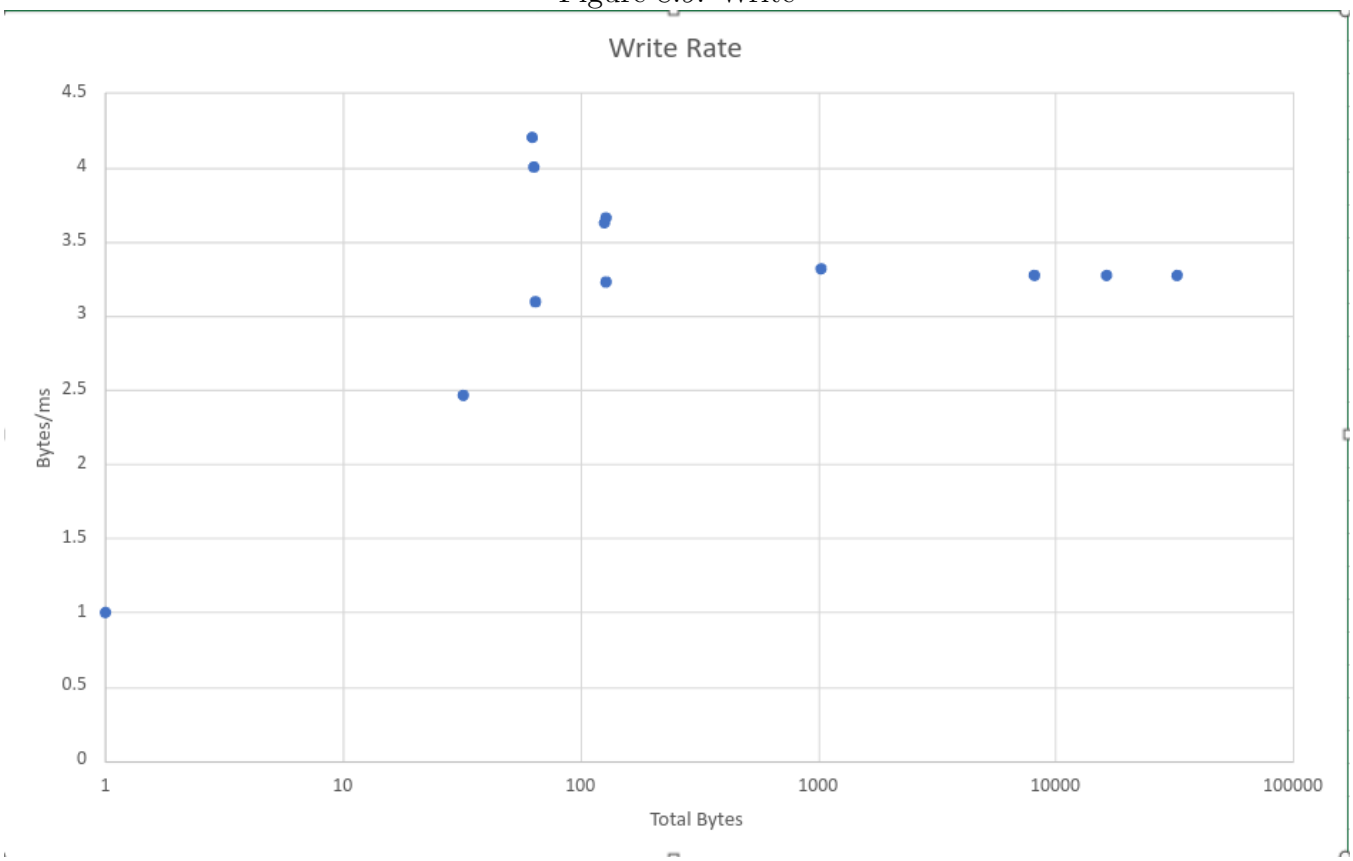


Figure 8.9: Write



The average read rate was 4.28 B/ms and the average write rate was 3.40 B/ms. The formulae that could be (roughly) used to estimate the time of transaction are as follows:

$$Read\ Time(ms) = \frac{Bytes}{4.28}$$

$$Write\ Time(ms) = \frac{Bytes}{3.40}$$

Conclusion

In this lab we were able to handle the driver for the I2C communications. In this instance, there was a single pair of master and slave, whereas in other setups there are multiple slaves that report to a master. In a situation in which a bidirectional bus can be driven by a multitude of slave devices, conflicting drivers need to be resolved. The way that this is resolved in the I2C case is by clock synchronization, in which the master sends out a well defined clock signal to the slaves if there is an issue with one of the slaves pulling the data line low. If the master tried to use the data line to solve the issue, the line being pulled low by one or more devices would guarantee that the signal was never received, but as the slaves have less control over the clock signal, the master can pulse the clock signal to arbitrate the conflict.