CSUN | COLLEGE OF ENGINEERING AND COMPUTER SCIENCE

Department of Electrical & Computer Engineering

ECE 425L – Microprocessor Systems

Summer 2024

Lab 8 - MARG Sensor

Final Project

4 June 2024

Instructor Shahnam Mirzaei

By: Jacob Bonilla & Gian Fajardo (Group 9)

Table of Contents

Introduction

The term MARG stands for Magnetic, Angular Rate, and Gravity Sensor which can be any device used to measure the orientation of a device.

The **accelerometer** is used to measure acceleration in three dimensions which include forces such as static forces such as gravity or dynamic forces. They are used to detect changes in motion, vibration, and orientation (using the gravity frame of reference). The **gyroscope** measures the change in angular rate around the x, y, and z directions. The **magnetometer** measures the strength and direction of magnetic fields around it and, like a compass, can be used to determine the 3D orientation relative to Earth's magnetic field. Some uses of these sensors in the real world include navigation systems like a GPS, electronics such as smartphones and tablets, and drones and robots.

The use of MARG sensors in determining the so-called **body-frame orientation** and, through its backward calculation, the **world-frame orientation**, is paramount for determining the orientation relative to the Earth. The way this is done is to take all of this data which have their own noise characteristics and fuse them with a discrete state observer algorithm, which can output a state estimate of an orientation variable, the most stable of which is the **quaternion**. For more information on quaternions, Grant Sanderson and Ben Eater created a visually oriented web application of quaternions as it relates to orientation.

The accel/gyro in an MPU6050 and the magnetometer of the QMC5883L are the two I^2C-based sensors that we will use for our project. Within the narrow scope of this project, we are simply looking to create code that allows us to read data from the MARG sensors and be able to display the readings. We will also discuss any issues or challenges we faced along the way and the step-by-step process we followed to complete the project and solve our issues.

Materials

Materials	Notes	External Links
MPU-6050	Main 6-axis accelerometer/gyroscope board, made by Invensense many years ago. This has many features: a digital low- and high-pass filter, and the Digital Motion Processor (DMP) to output a quaternion. To access the DMP, I^2Cdevlib's example Arduino sketch can be followed.	Most economical Amazon purchase I^2Cdevlib MPU-6050 landing page Reference Arduino sketch using the Teapot Demo
QMC5883L	Main 3-axis magnetometer that was used. Note that Honeywell, the original manufacturer, had authorized QST to make this IC, resulting in this variant. Usage of any C/++ library for either chip, as far as we can tell, are very similar in all but the I^2C peripheral addresses.	Reference Library by MPrograms

2.54 mm M/F header pins	This is used to make a quick-disconnect system, which will make for an easy swap if one chip does not work.	Amazon search results for 2.54 mm M/F header pins
4.7 KΩ resistors	Pull-up resistors for both the Serial Clock and Serial Data lines.	N/A
---------- **(OPTIONAL)** ----------		
MPU-9250	The next best from the MPU-6050 to the accel/gyro and magnetometer. This MARG sensor has everything that we need in one package. Gian had also used this in previous tests, however briefly. Data can be accessed via I^2C or SPI.	Amazon purchase link Reference Library by Bolder Flight Systems
ICM-20948	This appears to be the latest and greatest from Invensense. Due to Adafruit's qwiic-connect system and provided Arduino (C++) libraries, this reduces the development time. All that is required is time spent adapting the files from C++ to C.	Adafruit product page

Coding Style

I^2C Wiring

It would be a feat of deception not to point out the guide we used to make this project possible. We used the work of the authors from the Microcontrollers Lab blog which outlines their methods of reading from the MPU6050. Their work revolves around reading I^2C data from an unstated TM4C123 series microcontroller of theirs. We continue from this work to incorporate how to read I^2C data from the given TM4C123GH6PM microcontroller.

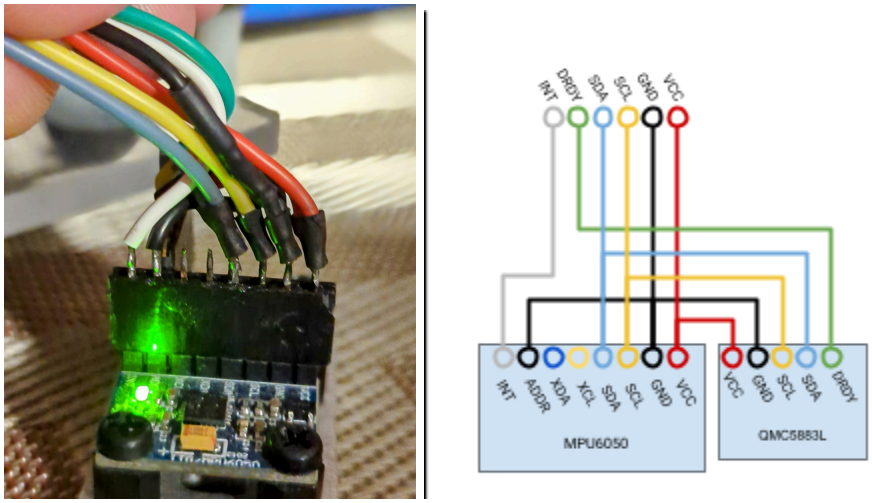

Figure 8.1 - Wiring diagram of MPU6050 and QMC5883L

We start with the wiring diagram. In addition to the 3v3 and GND wires, the I^2C bus uses the serial data wire SDA and the serial clock wire SCL. Both peripheral devices that we set out to use can share the same bus, so their SCL, SDA, 3v3, and GND are shared. Additionally, both devices are interrupt-capable. Finally, the ADDR pin of the MPU6050 is pulled low, which guarantees that this device will use the device address 0x68. If pulled high, then the device can be accessed with 0x69. The wiring diagram is shown in the image above.

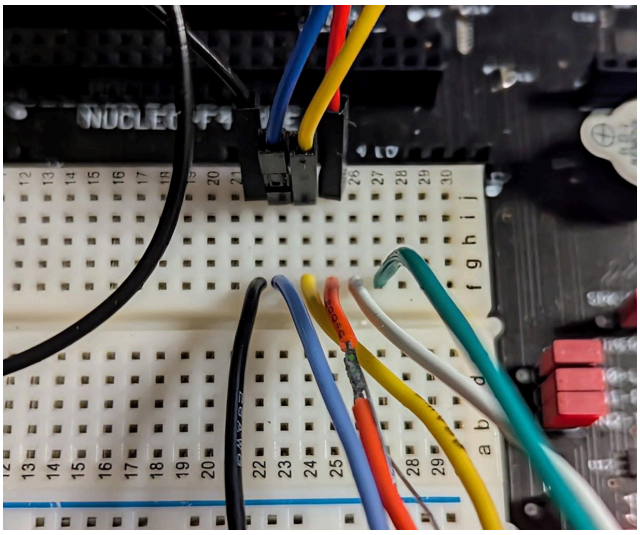

Figure 8.2 - Inter-connections between homemade wires and Dupont Connectors

With the connections made on the breadboard between the homemade wires and the breadboard, the Dupont wires connected SCL and SDA to pins PD0 and PD1, in addition to the 3v3 and ground pins (picture not taken). To be sure, they were connected to the upwards-facing set of pins on the Tiva board itself. In addition, the SCL and SDA pins are pulled high with 4.7 KΩ pull-up resistors. It was unknown whether or not these served their purposes but its inclusion was not harmful.

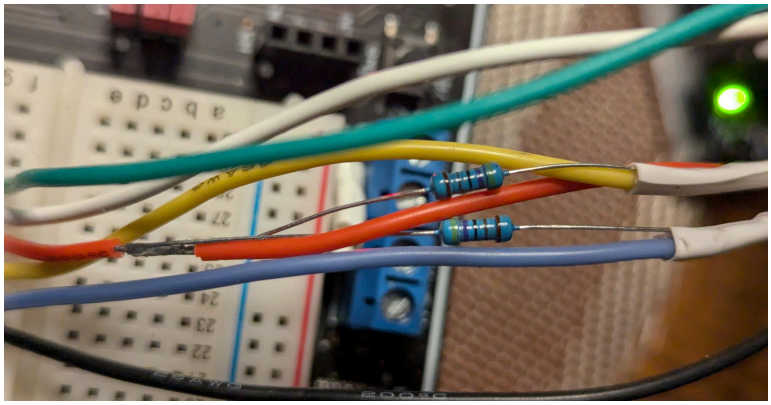

Figure 8.3 - Inter-connections between Homemade Wires & Dupont Connectors

I²C and Device Configuration

All rely on the given I2C3_Rd() and I2C3_Wr() to send or receive data from the host or controller to the peripheral device. The following function, firstly, indicates how the I²C bus is initialized.

```
5  void I2C3_Init() {
6
7    SYSCTL->RCGCGPIO |= 0x00000008; // Enable the clock for port D
8    SYSCTL->RCGCI2C  |= 0x00000008; // Enable the clock for I2C 3
9    GPIOD->DEN       |= 0x03;       // Assert DEN for port D
10
11   // Configure Port D pins 0 and 1 as I2C 3
12   GPIOD->AFSEL    |= 0x00000003 ;
13   GPIOD->PCTL     |= 0x00000033 ;
14   GPIOD->ODR      |= 0x00000002 ;  // SDA (PD1) pin as open darin
15   I2C3->MCR        = 0x0010 ;      // Enable I2C 3 master function
16
17   /* Configure I2C 3 clock frequency
18   (1 + TIME_PERIOD ) = SYS_CLK /(2* ( SCL_LP + SCL_HP ) * I2C_CLK_Freq )
19   TIME_PERIOD = 16,000,000/(2(6+4) *100000) - 1 = 7
20   */
21   I2C3->MTPR       = 0x07;
22
23 }
```

Figure 8.4 - I2C3 Port initialization

In the case of the port usage, their work called for using the alternate function of GPIOD pins 0 and 1 for I²C functionality, the first half of the configuration reflects this. RCGCGPIO enables port D, digital enable is enabled for ports 1 and 2 which uses the mask 0x11. AFSEL sets the alternative functions for GPIOD and PCTL registers set at 0x3 in nibbles 0 and 1 are choosing the I2C3SCL and I2C3SDA alternate functions.

PD0	61	AIN7	SSI3Clk	SSI1Clk	I2C3SCL	M0PWM6	M1PWM0	
PD1	62	AIN6	SSI3Fss	SSI1Fss	I2C3SDA	M0PWM7	M1PWM1	
PD2	63	AIN5	SSI3Rx	SSI1Rx	-	M0FAULT0	-	

Figure 8.5 - Snippet of Table 23.5 on Alternate Functions

The master control register MCR is set at the second nibble as 0x1. The first three bits of this nibble enables the host as the controller, the peripheral, or enables the I²C glitch mode. The 0x1 designates the Tiva device as the controller. Finally, the I²C Master Timer Period register MTPR enables the clock period for which the I2C3 buses and the devices within it will operate at.

When writing or reading to each of the devices, functions are used to reduce the error. We can use a function that writes to the device in order to configure them. The type of data that will be sent is illustrated in the figure on the next page.

It starts with a start condition, for which the data is pulled low. The peripheral address is sent next. The read and write bit is sent, then an acknowledge bit is transmitted from the peripheral to acknowledge that the information had been received. Then, a series of 1-byte data frames are sent with an acknowledge bit from the peripheral until there are no more requests.

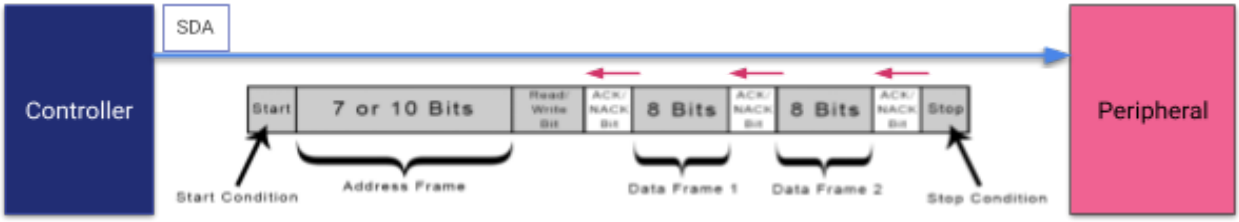

Figure 8.6 - I^2C Data Packet

When writing to one of our devices, the requirements are quite similar: we needed the identifier address of the **peripheral** device, the so-called **register address**, and the **data**. The MPU6050 has the **peripheral address(es) 0x68** or **0x69**, while the QMC5883L has **0x0D**. The register addresses are read or written to in the same way as registers within the microcontroller itself i.e. the manufacturer declares these register addresses in their respective datasheets and information is provided on how to configure them.

```
/* Write one byte only */
/* byte write: S-(saddr+w)-ACK-maddr-ACK-data-ACK-P */
char I2C3_Wr(int peripheral_addr, char register_addr, char data) {

  char error;

  /* send slave address and starting address */
  I2C3->MSA = peripheral_addr << 1;
  I2C3->MDR = register_addr;
  I2C3->MCS = 3;                        /* S-(saddr+w)-ACK-maddr-ACK */

  error = I2C_wait_till_done();         /* wait until write is complete */
  if (error) return error;

  /* send data */
  I2C3->MDR = data;
  I2C3->MCS = 5;                        /* -data-ACK-P */
  error = I2C_wait_till_done();         /* wait until write is complete */
  while(I2C3->MCS & 0x40);              /* wait until bus is not busy */
  error = I2C3->MCS & 0xE;
  if (error) return error;

  return 0;       /* no error */

```

Figure 8.7 - I2C3_Wr() function

In the figure above, both the peripheral and register addresses are sent sequentially. Then the I2C_wait_till_done() is waiting for the acknowledge bit. Then, the data is sent and waits for another acknowledge bit.

The read function is quite similar with a few addons. A new parameter *byteCount* and a character array *data* were added to the list. But the same register accessing stages appear along with the I2C_wait_till_done() call. The function takes one direction if only one byte of data

needs to be retrieved, but otherwise reads the data sequentially and updates the entries of *data* through its memory address. The full steps are shown in the next page.

```
64  char I2C3_Rd(int peripheral_addr, char register_addr, int byteCount, char* data) {
65
66    char error;
67
68    if (byteCount <= 0)
69      return -1;          /* no read was performed */
70
71    /* send slave address and starting address */
72    I2C3->MSA = peripheral_addr << 1;
73    I2C3->MDR = register_addr;
74    I2C3->MCS = 3;        /* S-(saddr+w)-ACK-maddr-ACK */
75    error = I2C_wait_till_done();
76    if (error)
77      return error;
78
79    /* to change bus from write to read, send restart with slave addr */
80    I2C3->MSA = (peripheral_addr << 1) + 1;   /* restart: -R-(saddr+r)-ACK */
81
82    if (byteCount == 1)             /* if last byte, don't ack */
83      I2C3->MCS = 7;              /* -data-NACK-P */
84    else                            /* else ack */
85      I2C3->MCS = 0xB;            /* -data-ACK- */
86    error = I2C_wait_till_done();
87    if (error) return error;
88
89    *data++ = I2C3->MDR;            /* store the data received */
90
91    if (--byteCount == 0) {         /* if single byte read, done */
92      while(I2C3->MCS & 0x40);    /* wait until bus is not busy */
93      return 0;       /* no error */
94    }
```

Figure 8.8 - I2C3_Rd() function

Through the read function, the I2C.c file along with its header file are declared for the MPU_6050.c and QMC5883L.c files to use. Once called, the following set of figures show how the two devices were configured. Only the bare minimum of configuration changes were made.

```
2  #include "MPU_6050.h"
3
4  #include "I2C.h"
5
6  #define MPU_ADDR 0x68
7
8  void MPU6050_Init() {
9
10   I2C3_Wr(MPU_ADDR, SMPLRT_DIV,    0x07);
11   I2C3_Wr(MPU_ADDR, PWR_MGMT_1,    1);
12   I2C3_Wr(MPU_ADDR, CONFIG,        0x00);
13   I2C3_Wr(MPU_ADDR, ACCEL_CONFIG,  0x00);
14   I2C3_Wr(MPU_ADDR, GYRO_CONFIG,   0x18);
15   I2C3_Wr(MPU_ADDR, INT_ENABLE,    0x00);
16
17 }
```

```
2  #include "QMC5883L.h"
3  #include "I2C.h"
4
5  void QMC5883L_init() {
6
7    // recommended by the datasheet
8    I2C3_Wr(QMC_ADDR, SET_RESET_PERIOD_REG, 0x01);
9
10   //I2C3_Wr(ADDR, CTRL_REG_1, 0x0D); // 0b11000101
11   I2C3_Wr(QMC_ADDR, CTRL_REG_1, 0x01); // 0b11000101
12
13   //I2C3_Wr(ADDR, CTRL_REG_2, 0xC1);
14   I2C3_Wr(QMC_ADDR, CTRL_REG_2, 0x01);
15
16 }
```

Figure 8.9 - MPU6050 and QMC5883L Configuration

Main Program Loop

```
19  char  row_1_string[11] = {' '},
20        row_2_string[11] = {' '};
21  int num_elems = 11, i;
22
23
24  char  mpu_data[14],
25        qmc_data[ 6];
26
27  int16_t accX,   accY,   accZ,
28          GyroX,  GyroY,  GyroZ,
29          Temp,
30          magX,   magY,   magZ;
31
32  float AX, AY, AZ, t, GX, GY, GZ, MX, MY, MZ;
33
34  I2C3_Init();      delay_ms(100);
35  MPU6050_Init();   delay_ms(100);
36  QMC5883L_init();  delay_ms(100);
37  LCD_init();       delay_ms(100);
```

Figure 8.10 - State Machine Declarations

The state variables, as they exist, consist of three: character arrays *mpu_data[14]* and *qmc_data[6]* which are used to store the raw data. Then, int16_t variables from each of the sensors are declared. Finally, the floating point representations are declared. In addition, the LCD was used to store the value, and so character arrays *row_1_string[]* and *row_2_string[]* are used to save the incoming floating point data and print them afterwards.

```
42
43      I2C3_Rd(MPU_ADDR, ACCEL_XOUT_H, 14, mpu_data);
44      delay_ms(2);
45      I2C3_Rd(QMC_ADDR, XOUT_LSB, 6, qmc_data);
46
```

After each of the initialization functions are called, including the LCD initialization, it starts with declaring I2C3_Rd() for each of the sensors. At this point, it is important to stress why this can be done in the first place. In the case of the magnetometer sensor, we can enable what is known on page 15 of the QMC datasheet as the I^2C pointer roll-over to read only the topmost register and carry on to reading the rest of the registers. This is neat as this enables the input of the device address and only one register address, which is preceded by the 6 other bytes of data with an acknowledge bit sent. This applies to the MPU6050 as well.

8.2.5 I^2C Pointer Roll-over

QMC5883L has an embedded I^2C pointer roll-over function which can improve the data transmission efficiency. The I^2C data pointer will automatically roll between 00H ~ 06H if I^2C read begins at any position among 00H~06H. This function is enabled by set 0AH[6] = 01H.

- Read measured data, if any of the six data register is accessed, DRDY and DOR turn to "0".
- Data protection, if any of the six data register is accessed, data protection starts. During Data protection period, data register cannot be updated until the last bits 05H (ZOUT [15:8]) have been read.

Figure 8.11 - I^2C Read Configuration

In the normal read sequence section of page 12, though, it says that the output data registers cannot be updated if and until the last bit is read. This is part of a data protection protocol of the chip. This was the source of our problems early on as we were only trying to read two bytes at a time for our initial tests. When making I^2C read calls, calling for the whole set of addresses was a requirement.

```
48      accX =  (int16_t) ( (mpu_data[ 0] << 8 ) | mpu_data[ 1] );
49      accY =  (int16_t) ( (mpu_data[ 2] << 8 ) | mpu_data[ 3] );
50      accZ =  (int16_t) ( (mpu_data[ 4] << 8 ) | mpu_data[ 5] );
51      //Temp =  (int16_t) ( (mpu_data[ 6] << 8 ) | mpu_data[ 7] );
52      GyroX = (int16_t) ( (mpu_data[ 8] << 8 ) | mpu_data[ 9] );
53      GyroY = (int16_t) ( (mpu_data[10] << 8 ) | mpu_data[11] );
54      GyroZ = (int16_t) ( (mpu_data[12] << 8 ) | mpu_data[13] );
55
56      magX =  (int16_t) ( (qmc_data[1] << 8) | qmc_data[0] );
57      magY =  (int16_t) ( (qmc_data[3] << 8) | qmc_data[2] );
58      magZ =  (int16_t) ( (qmc_data[5] << 8) | qmc_data[4] );
59
60      // Convert The Readings
61      AX = (float) accX/16384.f;
62      AY = (float) accY/16384.f;
63      AZ = (float) accZ/16384.f;
64      GX = (float) GyroX/131.f;
65      GY = (float) GyroX/131.f;
66      GZ = (float) GyroX/131.f;
67      //t = ((float) Temp/340.00) + 36.53;
68
69      MX = (float) magZ/16384.f;
70      MY = (float) magZ/16384.f;
71      MZ = (float) magZ/16384.f;
72
```

Figure 8.12 - Floating Point Conversion

In any case, the chars were concatenated together in the order specified by the register maps of each and type-casted to int16_t. It is set up this way so that this can properly represent negative values. Finally, with the correct conversion factors, the 16-bit integers were converted to floats. Finally, using the LCD functions that were shown in previous labs, two of the 9 available data were printed onto the LCD panel. Acceleration and magnetic readings in the Z direction were printed and demonstrated.

```
75      sprintf(row_1_string, "AZ=%.5f", AZ);
76      sprintf(row_2_string, "MZ=%.5f", MZ);
77
78      for (i = 0; i < num_elems; i++) {
79        LCD_data(row_1_string[i]);
80      }
81      LCD_command(0xC0);
82
83      for (i = 0; i < num_elems; i++){
84        LCD_data(row_2_string[i]);
85      }
86
```

Figure 8.13 - LCD Operation

Going Above & Beyond

This project as a whole was something that required us to go above and beyond because we had not been taught the UART protocol or the I^2C protocol that was used in the project. At the time we had decided to go forward with this project, we were told we didn't have all the information needed from the class to do the project so we had to put it upon ourselves to learn the code needed and the overall use of UART and I^2C. We used various outside resources which included GitHub and other code files for us to review and be able to understand what each line did and how we would have to implement it. We also were given the slides from another course to be able to read over and learn more about I^2C. This already was a big task to go through with and on top of it adding the complexity of the code and being able to write it. Gian took it upon himself to personally teach anything that was a struggle to understand met many times outside of class and over the weekends to review the information and write the code together.

Results & Discussion

Thankfully in the end, we were able to complete all the tasks we set out for ourselves even though we thought at times that we wouldn't be able to fully complete them. We experienced a few issues at first with the accelerometer because the data wasn't being read properly and we were getting some weird and random symbols to display on the LCD.

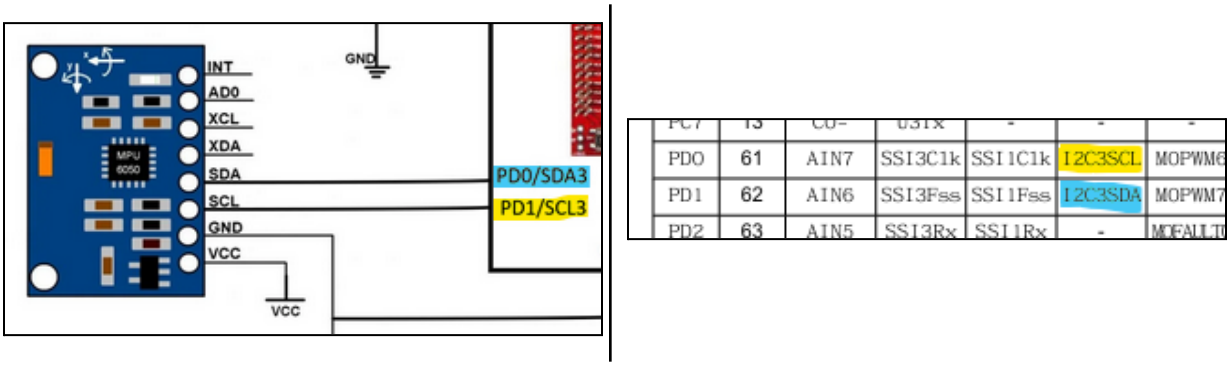

Figure 8.14 - Microcontrollers Lab Pinout (left) vs. TM4C123GH6PM Pinout (right)

After a few times of going through the code and all the connections on the board we were able to realize that the only issue we had was that the SCL and SDA pins were mixed up. According to Table 23-5 in GPIO Pins and Alternate Functions in page 1351 on the datasheet, and contrary to the instructions given to us in Microcontrollers Lab, the procedure called for installing the serial data pin SDA and serial clock SCL onto pins PD0 and PD1. Once we switched them we were able to correctly see the data displayed on the LCD.

The next part of the project was for us to be able to read the data from the magnetometer and we could not figure out at first what the issue was. After we presented the project in class we thought we wouldn't be able to figure out the issue but Gian took it upon

himself to work on the code and double-check all connections and over the past weekend he was able to figure out the issue and get the magnetometer to work properly.

Aside from the complexity of the code and having to learn a lot of the information outside of class the biggest challenge was finding time to work on the project together face to face. A lot of the time some of the code and how it worked was confusing but Gian helped to understand the code and went by the whole thing step by step while clarifying any issues. As mentioned in the results section we experienced various issues whether they were small or big at times and were able to get through them by double-checking all the code and connections we had.

Conclusion

In the end, we were able to complete all the tasks that we initially wanted to complete and learned about new concepts that are used in the real world. We learned how accelerometers and magnetometers work and how they are used to collect data in three dimensions. We then learned how to use UART and I^2C protocol to be able to write code to read the data from the sensors and display it on an LCD display. Overall this project was a good learning experience because we pushed ourselves to learn something outside of the classroom and understand these devices that are used in our everyday lives like with smartphones.