THE USAGE OF AN ACCELEROMETER AND I2C

Individual Report

ELEC 3907, WINTER 2022

Group 2A

Aditya Wiwekananda

101147416

March 16, 2022

Executive Summary

The goal of group 2A's project is to develop a motion-controlled vehicle. The main features that I worked on are the usage of the accelerometer module and how it is interfaced by an Arduino using I^2C. The accelerometer is used to measure the controller's static acceleration due to gravity. These measurements are used to determine the direction and speed at which the vehicle should drive. To obtain the measurements, I^2C communication must be utilized. The Arduino environment features a library that is optimized to handle these features. The accelerometer readings can be processed after it is retrieved.

Introduction

The project that is being developed by group 2A is a vehicle that is controlled using a wireless-motion sensing controller. The vehicle is 4-wheeled and must be able to move in the direction that the user tilts the controller. A motion controller is intended to provide users with an alternative input method. This is especially important for people who are unable to control a vehicle using the more traditional thumb sticks. The main features of this project will be motion-sensing, wireless connectivity, and maneuverability.

Since this report serves to detail the contributions that I have made towards the project, this report focuses on the implementation of the accelerometer used to measure the controller's position in space. Furthermore, a discussion is made on the usage of the I^2C communication protocol and its implementation. This method of communication is used to allow the Arduino to retrieve the data measured from the accelerometer.

Design Process

For the project, motion-sensing was conducted using the MPU-6050 inertial measurement unit (IMU). This device was utilized as it was readily available and inexpensive. Some features that this integrated circuit includes are an accelerometer, a gyroscope, and a temperature sensor [1].

A user will be able to control the vehicle by tilting the controller. For example, the vehicle moves forward if the controller is tilted forward (i.e., the top of the controller is at a lower z position when compared to its leveled position). Therefore, the measurement of static acceleration due to gravity is required. Figure 1 shows a visualization of the position at which the controller must be placed if the user wants the vehicle to move in the positive x-direction.

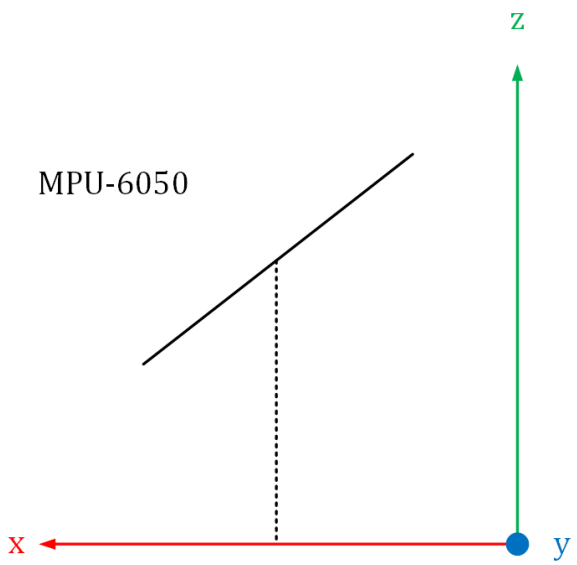

Figure 1: A diagram showing the position that the motion-controller should be placed if the vehicle is desired to move in the positive x-direction

Since the static acceleration is the only measurement required from the IMU, only the accelerometer is required from this device. For our project, this device is connected to an Arduino UNO using the I^2C bus. The diagram of the connections made between the two devices is shown in Figure 2.

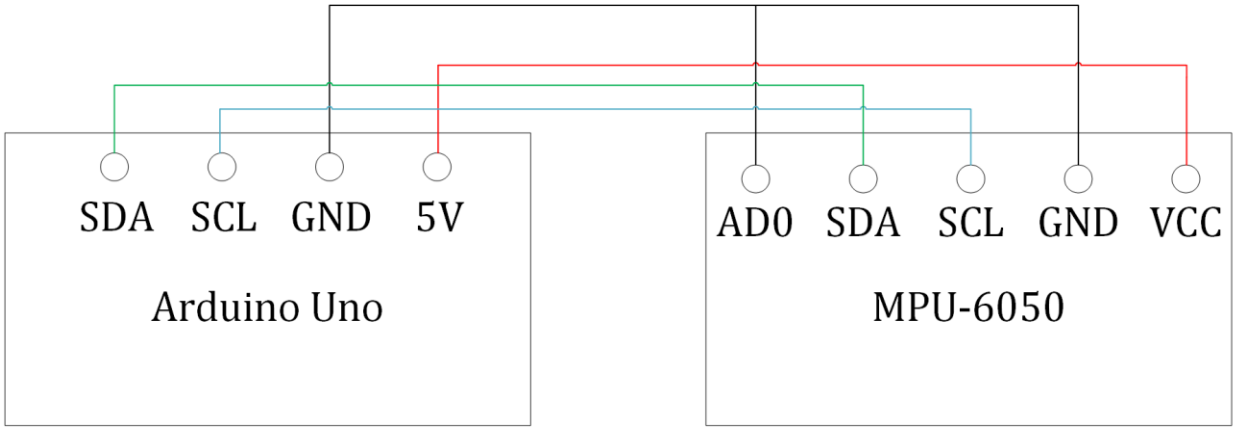

Figure 2: The circuit diagram of the MPU-6050 and Arduino Uno to measure acceleration data.

To understand the scope of the usage of I^2C in the project, an understanding of the basic principles must be established. In a simple case, I^2C can be used to connect at least two devices together. The terminologies that will be used for the two devices are the "master" and the "slave" device. The SDA (serial data line) is a shared line for the master and slave to read and write data to. Meanwhile, the SCL (serial clock line), as the name implies, will synchronize the data transfer over the I^2C bus. It should be noted that the SCL signal is generated by the master device. To add, multiple slave devices can be added to the same bus. This is useful as designers can utilize many devices and conserve the number of pins used on the microcontroller unit [2].

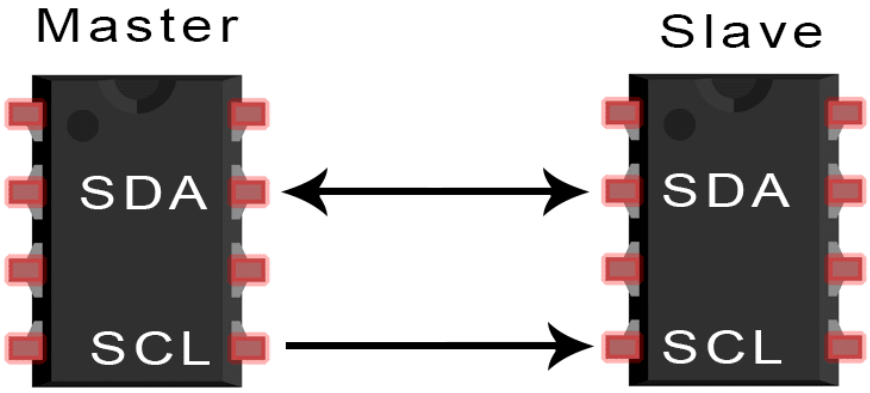

Figure 3: A diagram showing the connections between the "Master" and "Slave" devices [2]

To send/receive data to/from the slave device(s), the address for the device is required. This address is commonly a 7-bit binary value however 10-bits may be used by some devices. Note that the MPU-6050 is a 7-bit addressed IC (0b1101000) [3]. For all I^2C compatible devices, its address will be available on its datasheet.

The transfer of data on the I^{2}C bus happens on the SDA line. The transfer of data will start whenever the SDA line is pulled low when the SCL is high as shown in Figure 4. Afterwards, the 7-bit address of the slave will need to be sent by the master device. The address bits are followed by the Read/Write bit. 1 indicates read, and 0 indicates write. If this connection is successful, the connected slave device will send an acknowledge bit by pulling the SDA line low when SCL is high [4]. Often, a slave device will have multiple internal registers that can be written to or read from. Persons unfamiliar with registers can think of one as a look-up table; each address in the table contains a value. Therefore, to read/write data to an internal register; the address of the register is required. Register addresses can be found in the device's register map; a document that is like the datasheet. Again, an acknowledge/not acknowledge bit will be sent by the slave device after the transmission of the register address [4].

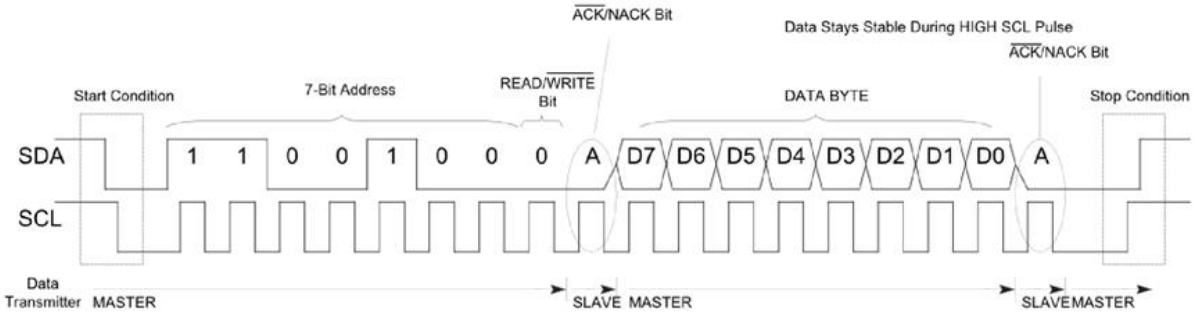

Figure 4: The timing diagram of a write transmission [4]

If a write was requested, the master device can send bytes of data until the stop condition is requested. This stop condition happens when the master pulls the SDA line high while the SCL is high. For a read request, the slave device will control the SDA line and send the data that was requested at the specified register to the master [4]. Once the master no longer needs to receive data, it will send a not acknowledge bit which will transfer the control of the SDA line back to the master device [4].

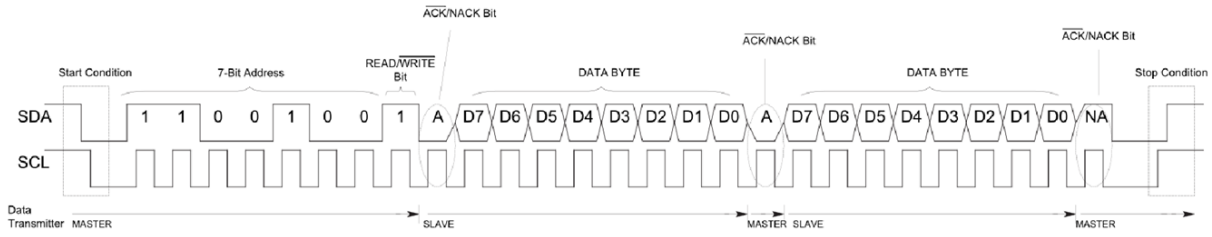

Figure 5: The timing diagram of a read transmission [4]

Now that the basis of I^{2}C communication is established, a discussion on the methods to use I^{2}C with Arduino microcontroller units will be discussed. Examples of the usage of an MPU-6050 IC will be demonstrated.

I^{2}C communication can be used with an Arduino environment by using the Wire library. To set up I^{2}C communications, the following code snippet shown in Figure 6 must be present in your Arduino *.ino* file. Leaving the parameters of the Wire.begin() function empty will set the device to join the I^{2}C bus as a master [5].

```
#include <Wire.h>

void setup() {
  Serial.begin(9600);
  Wire.begin(); //Access I2C
  setupMPU();
}
```

Figure 6: The code that must be inserted to establish I²C communications

After enabling I²C features, the setup of the slave devices is recommended. As shown in Figure 7, a function labeled setupMPU was created to properly set up the MPU-6050 to function as intended. The beginTransmission() function will commence I²C communication with the slave device specified by the input argument [6]. In this case, the slave address is 0b1101000. The write() function will queue bytes to be transmitted to the slave during the I²C transmission [5]. Finally, the endTransmission() will send the stop condition to the bus [5]. The lines from line 2 to line 5 will set up the power management registers with an internal register address of 0x6B [6]. The next 4 non-whitespace lines will setup the gyroscope configuration register; and the 4 lines after will setup the accelerometer register. More details on the settings used can be found in the MPU-6050's data sheet.

```
void setupMPU(){
  Wire.beginTransmission(0b1101000); //This is the I2C address of the MPU (b1101000/b1101001 for AC0 low/high datasheet sec. 9.2)
  Wire.write(0x6B); //Accessing the register 6B - deals with power section (register map sec 4.28)
  Wire.write(0x00); //Setting SLEEP register to 0.(Required register map, sec 4.8)
  Wire.endTransmission();

  Wire.beginTransmission(0b1101000); //I2C address of the MPU
  Wire.write(0x1B); //Accessing the Gyroscope configuration (register map, sec 4.4)
  Wire.write(0x00); //Setting the gyro to full scale +/- 250deg (lowest setting)
  Wire.endTransmission();

  Wire.beginTransmission(0b1101000); //I2C address of the MPU
  Wire.write(0x1C); //Accessing the Accelerometer configuration (register 1C)
  Wire.write(0x00); //Setting the accelerometer to +/- 2g (lowest setting)
  Wire.endTransmission();
}
```

Figure 7: Setup of three internal registers by way of the I²C write transmission process

Next, a function called recordAccelRegisters() is created to read data from the accelerometer readings register as shown in Figure 8. The read process is more involved compared to the write function.

First, the register at which the accelerometer readings start will need to be accessed using the write() function in between the transmission functions. This segment of the code effectively places the 'pointer' at the register with an address of 0x3B (the starting accelerometer reading register) [6]. Afterwards, the requestFrom() function will read the data stored at the register that is pointed to. The arguments to the requestFrom() function are the slave address, and the number of bytes to read [5]. Since there are 6 accelerometer registers, 6 is passed as an argument.

```
void recordAccelRegisters(){
  Wire.beginTransmission(0b1101000); //I2C address of the MPU
  Wire.write(0x3B); //Starting register for Accel Readings
  Wire.endTransmission();

  Wire.requestFrom(0b1101000, 6); //Request Accel Registers (3B - 40)
  while(Wire.available() < 6);  //Store bytes on a buffer on each iteration?
  accelX = Wire.read() <<8 | Wire.read(); // Store the first 2 bytes into accelX
  accelY = Wire.read() <<8 | Wire.read(); // Store the middle 2 bytes into accelY
  accelZ = Wire.read() <<8 | Wire.read(); // Store last first 2 bytes into accelZ
  processAccelData();
}
```

Figure 8: The recordAccelRegisters() function

The requestFrom() function will increment the pointer to the next register address. The function will sequentially store the bits in each register which is shown in Figure 9. Each direction; x, y, and z; has two registers each. Therefore, the data that is stored for each direction will need to be concatenated into one variable. The while loop will iterate and store the requested bytes on a buffer that is made by the available() function [5].

Addr (Hex)	Addr (Dec.)	Register Name	Serial I/F	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
3B	59	ACCEL_XOUT_H	R	ACCEL_XOUT[15:8]							
3C	60	ACCEL_XOUT_L	R	ACCEL_XOUT[7:0]							
3D	61	ACCEL_YOUT_H	R	ACCEL_YOUT[15:8]							
3E	62	ACCEL_YOUT_L	R	ACCEL_YOUT[7:0]							
3F	63	ACCEL_ZOUT_H	R	ACCEL_ZOUT[15:8]							
40	64	ACCEL_ZOUT_L	R	ACCEL_ZOUT[7:0]							

Figure 9: The register map of the MPU-6050 showing the addresses and bits of the accelerometer reading registers [6]

As previously mentioned, each direction will have a value that is 16 bits long. Therefore, to store the values, the first byte needs to be bit-shifted to the left by 8 bits. Then a bitwise or is performed between the first register and the second register to concatenate the recorded values.

After receiving the data measured by the MPU-6050; developers must process the data to implement a feature that is desired. For our project, the measured accelerometer readings were used to determine the direction and the speed at which the vehicle should move.

Conclusion

During the semester, I oversaw the development of the motion-sensing capabilities of the controller. An accelerometer along with I^2C communications were the main features that I worked on for our project. This aspect is very important for the vehicle as it must be able to move in the direction that the user tilts the controller. A motion controller is intended to provide users with an alternative input method. This is especially important for people who are unable to control a vehicle using the more traditional thumb sticks.

References

[1] “MPU-6050 | TDK.” [Online]. Available: https://invensense.tdk.com/products/motion-tracking/6-axis/mpu-6050/. [Accessed: Mar. 15, 2022]

[2] S. C. | D. Electronics | 55, “Basics of the I2C Communication Protocol,” *Circuit Basics*, Feb. 13, 2016. [Online]. Available: https://www.circuitbasics.com/basics-of-the-i2c-communication-protocol/. [Accessed: Mar. 15, 2022]

[3] “MPU-6000-Datasheet1.pdf.” [Online]. Available: https://invensense.tdk.com/wp-content/uploads/2015/02/MPU-6000-Datasheet1.pdf. [Accessed: Mar. 16, 2022]

[4] “I2C Primer: What is I2C? (Part 1) | Analog Devices.” [Online]. Available: https://www.analog.com/en/technical-articles/i2c-primer-what-is-i2c-part-1.html. [Accessed: Mar. 15, 2022]

[5] “Arduino - Wire.” [Online]. Available: https://www.arduino.cc/en/Reference/Wire. [Accessed: Mar. 16, 2022]

[6] “MPU-6000-Register-Map1.pdf.” [Online]. Available: https://invensense.tdk.com/wp-content/uploads/2015/02/MPU-6000-Register-Map1.pdf. [Accessed: Mar. 16, 2022]

Appendix

The code used for I^2C communications with the MPU-6050:

```
//Author: Aditya Wiwekananda
#include <Wire.h>

//Datasheet: https://invensense.tdk.com/wp-content/uploads/2015/02/MPU-6000-Datasheet1.pdf
//Register Map: https://invensense.tdk.com/wp-content/uploads/2015/02/MPU-6000-Register-Map1.pdf

//Variables to store data from MPU 6050
long accelX, accelY, accelZ;
float gForceX, gForceY, gForceZ;

long gyroX, gyroY, gyroZ;
float rotX, rotY, rotZ;

void setup() {
  Serial.begin(9600);
  Wire.begin(); //Access I2C
  setupMPU();
}

void setupMPU(){
  Wire.beginTransmission(0b1101000); //This is the I2C address of the MPU (b1101000/b1101001 for AC0 low/high datasheet sec. 9.2)
  Wire.write(0x6B); //Accessing the register 6B - deals with power section (register map sec 4.28)
  Wire.write(0x00); //Setting SLEEP register to 0.(Required register map, sec 4.8)
  Wire.endTransmission();

  Wire.beginTransmission(0b1101000); //I2C address of the MPU
  Wire.write(0x1B); //Accessing the Gyroscope configuration (register map, sec 4.4)
  Wire.write(0x00); //Setting the gyro to full scale +/- 250deg (lowest setting)
  Wire.endTransmission();

  Wire.beginTransmission(0b1101000); //I2C address of the MPU
  Wire.write(0x1C); //Accessing the Accelerometer configuration (register 1C)
  Wire.write(0x00); //Setting the accelerometer to +/- 2g (lowest setting)
  Wire.endTransmission();
}
```

```
void loop() {
  recordAccelRegisters();
  recordGyroRegisters();
  printData();
  delay(100);
}

void recordAccelRegisters(){
  Wire.beginTransmission(0b1101000); //I2C address of the MPU
  Wire.write(0x3B); //Starting register for Accel Readings
  Wire.endTransmission();

  Wire.requestFrom(0b1101000, 6); //Request Accel Registers (3B - 40)
  while(Wire.available() < 6);  //Store bytes on a buffer on each iteration?
  accelX = Wire.read() <<8 | Wire.read(); // Store the first 2 bytes into accelX
  accelY = Wire.read() <<8 | Wire.read(); // Store the middle 2 bytes into accelY
  accelZ = Wire.read() <<8 | Wire.read(); // Store last first 2 bytes into accelZ
  processAccelData();
}

void processAccelData() {
  gForceX = accelX / 16384.0;
  gForceY = accelY / 16384.0;
  gForceZ = accelZ / 16384.0;
}

void recordGyroRegisters(){
  Wire.beginTransmission(0b1101000); //I2C address of the MPU
  Wire.write(0x43); //Starting register for Gyro Readings
  Wire.endTransmission();

  Wire.requestFrom(0b1101000, 6); //Request Accel Registers (43 - 48)
  while(Wire.available() < 6);  //Store bytes on a buffer on each iteration?
  gyroX = Wire.read() <<8 | Wire.read(); // Store the first 2 bytes into gyroX
  gyroY = Wire.read() <<8 | Wire.read(); // Store the middle 2 bytes into gyroY
  gyroZ = Wire.read() <<8 | Wire.read(); // Store last first 2 bytes into gyroZ
  processGyroData();
}

void processGyroData() {
  rotX = gyroX / 131.0;
  rotY = gyroY / 131.0;
  rotZ = gyroZ / 131.0;
}
```

```
void printData() {
  Serial.print("Gyro (deg)");
  Serial.print(" X=");
  Serial.print(rotX);
  Serial.print(" Y=");
  Serial.print(rotY);
  Serial.print(" Z=");
  Serial.print(rotZ);
  Serial.print(" Accel (g)");
  Serial.print(" X=");
  Serial.print(gForceX);
  Serial.print(" Y=");
  Serial.print(gForceY);
  Serial.print(" Z=");
  Serial.println(gForceZ);
}
```