0. General Information

Student Name:	Ellis Hobby
Student ID Number:	01928869
Team Name:	Tube Disaster
Team Member Names:	Ellis Hobby
Date of Completion:	03/26/23
Demonstration Method:	Zoom

1. Design

1.1 Hardware Design

The current project requires an embedded system that facilitates user control over the classic "snake" game by way of joystick or digital motion processing (DMP™). Both devices must measure user input and use the corresponding data to make changes to the snake's direction within a two-dimensional plane. The movements will be referred to as *left*, *down*, *right*, and *up*. Additionally, the DMP™ shall allow a player to gesture for a "double points" round by shaking the device. The movement requests shall be communicated to the application running the game client via serial data transmission. Finally, a buzzer shall be used to indicate when a player a completed a round and earned points by "eating the apple" [1].

The core of the system is built upon the ATmega2560 from Atmel. This 8-bit microcontroller (MCU) has a processing speed of approximately 16 MHz and a multitude of input-output (IO) capabilities that are well suited for the desired system functionality, namely the analog to digital converter (ADC) and inter-integrated circuit bus (I2C) [2]. The ADC permits analog voltage measurements generated by the joystick to be utilized for processing by way of digital conversion. The I2C bus manages communication with the DMP™ using serial data streams. Considering that game updates occur once every 100 ms (i.e. 10 Hz) with only minor speed increase over time, the 16 MHz clock leaves a plentiful amount of headroom for reliable processing and data acquisition. Figure 1 shows a circuit diagram depicting the MCU with connections to external peripherals, net labels were used to improve readability.

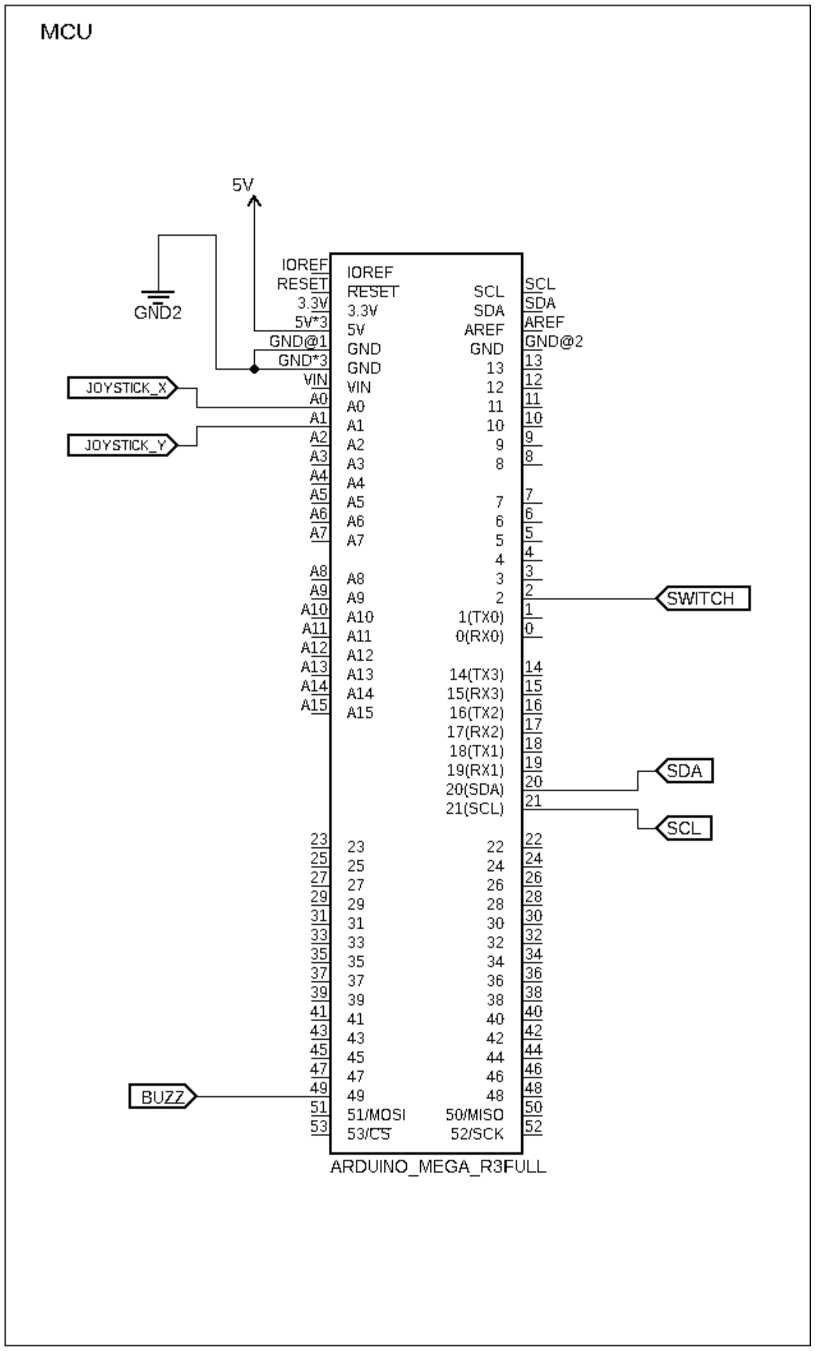


Figure 1: Atmega 2560 Circuit Block

The joystick module, Figure 2, utilizes two 10kΩ potentiometers with opposing rotational orientations. The module has two output pins for each wiper and two input pins for the high and low voltage references of both pots. This creates a system where two analog voltages can be used to describe positional displacement within a cartesian plane. For example, consider that a traditional coordinate system is used where X is the value of horizontal displacement and Y the vertical. Additionally, consider that both the X and Y potentiometers are wired up to 2.5V and 0V allowing for a continuous sweep between the thresholds. At the center position both X and Y would read exactly halfway between the set range, approximately 1.25V. Increasing the joystick to the maximum position along horizontal axis would result in an X value of 2.5V. The value of Y in this situation would remain at 1.25V as the vertical position has not increased. Here we can clearly see how the variations in voltages can be used to map positional data. However, the system requires that the data be processed in the digital domain. Therefore, it is necessary to convert these continuous analog signals to stepped digital values. This is achieved using two ADCs available on the ATmega2560, *ADC0* and *ADC1*. The MCU offers 16 individual inputs for obtaining measurements through multiplexing (MUX). The analog signals are scaled to 10-bit value, where 0x0000 is an integer that indicates the lowest voltage (0V) and 0x03FF is an integer representing the highest voltage (2.54V). A stable analog reference of 2.45V is generated by the MCU and used along with ground (0V) for the input voltage range to the joystick. Figure 3 shows a schematic drawn for the joystick circuit block. This includes voltage reference rails and net labels for connections to the MCU. Additionally, there is switching circuitry for the built in button, this will be explained below.

A button is integrated into the joystick module, when pressed connects the SW line to ground. While the design requirements did not call for this to be used, it seemed like a waste, so it was added for extra functionality. Essentially it acts to mediate between the control methods. If the current control method is the joystick and the button is pressed control will be given to the DMP™ and vice versa. This allows both control methods to be used within the same game session without the need to restart any programs. The button is connected to *PORTE 4* on the MCU and triggers an external interrupt (*INT4*) when pressed. A physical debouncing network, made up by the combination of *R1* and *C1,* was used to prevent switch chatter and erroneous data. The resistor capacitor combination creates a lowpass filter with a small time constant. This eliminates transient spikes resulting from mechanical vibration. Implementing a 0.1 μF capacitor and 100 kΩ creates a time constant of approximately 10 ms, as described by Equation (1) [3]. The resistor is connected to 5V and the capacitor to 0V, their common node then connects to the SW terminal of the joystick, Figure 3. This allows the signal to be held high and then brought low when the button is pressed.

$$\tau = R * C \tag{1}$$

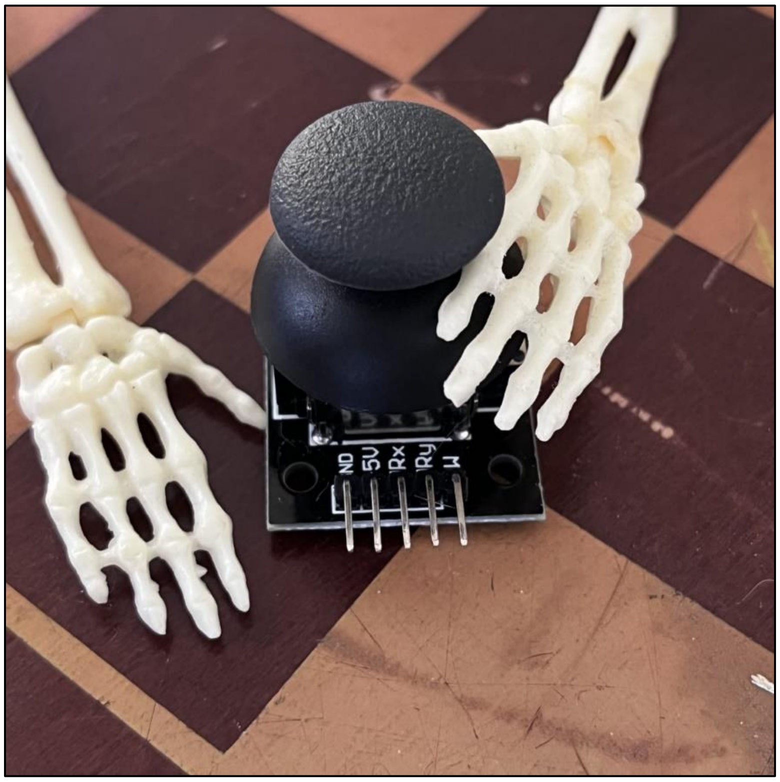


Figure 2: Joystick Module

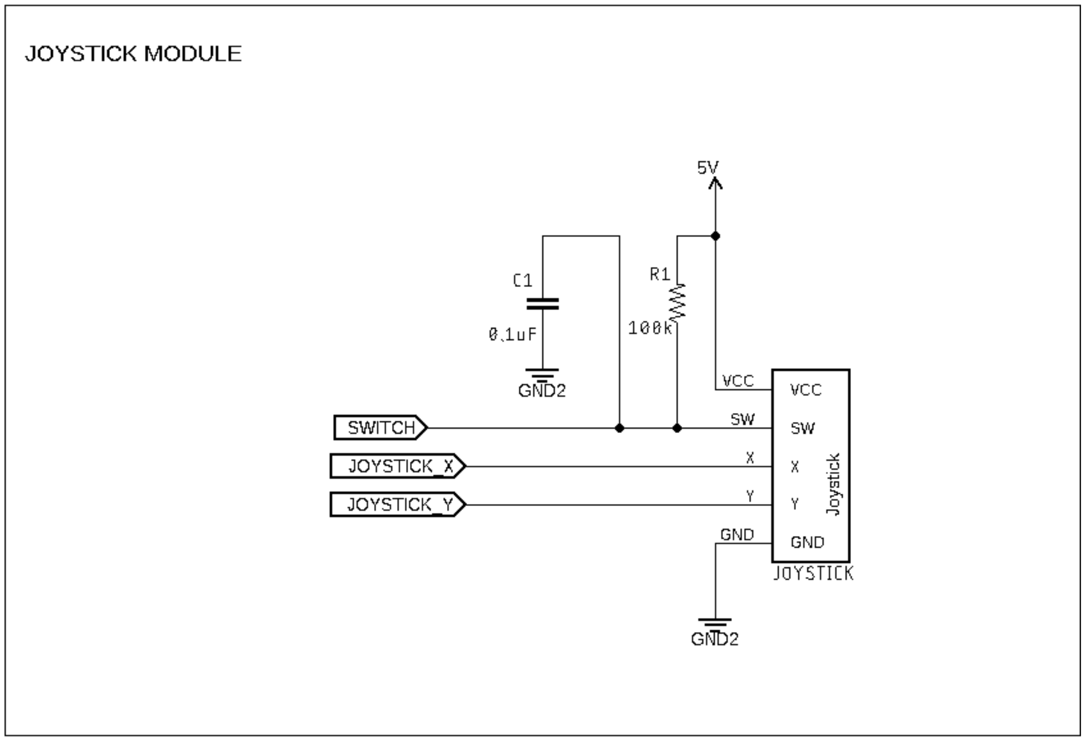


Figure 3: Joystick Circuit Block

The alternate control method utilizes the MPU-6050 DMP™ module from InvenSense. This module is a offers 6-axis motion tracking capability through combining a 3-axis gyroscope with a 3-axis accelerometer [4], Figure 4. As shown in the diagram, the gyroscope measures the rotational displacement around each axis, while the accelerometer measures the acceleration along each axis. The current project uses the x and y gyroscopes to control direction in the game. Where rotating along the x-axis controls left and right direction, while rotating along the y-axis controls up and down direction. The x-axis accelerometer is used to identify shaking gestures, which indicate the player wishes to increase the points for the round. The x-axis accelerometer was experimentally chosen as it seemed to give the best results without interfering with other measurements, this will be discussed below in the software design section. DMP™ data is obtained as serial data using the I2C protocol. The GY-521 breakout board, Figure 4, includes pull up resistors for the clock (SCL) and data (SDA) lines of the I2C bus. Additionally, this board provides a voltage regulator enabling the supply a positive supply of either 3.3V or 5V, the latter was chosen for the current system. Figure 5 shows the DMP™ circuit block with relevant connections for power and data.

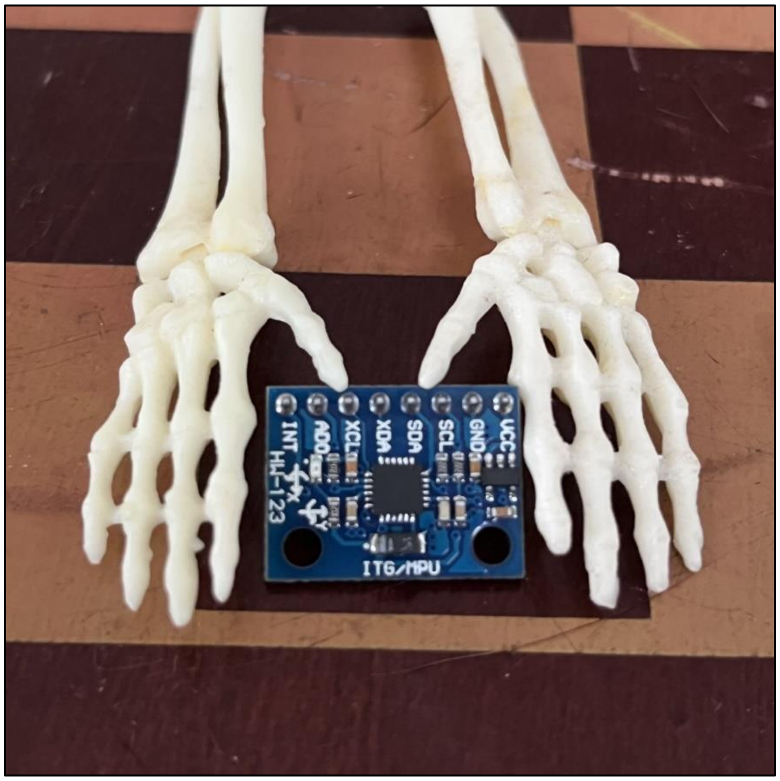


Figure 4: DMP™ Module

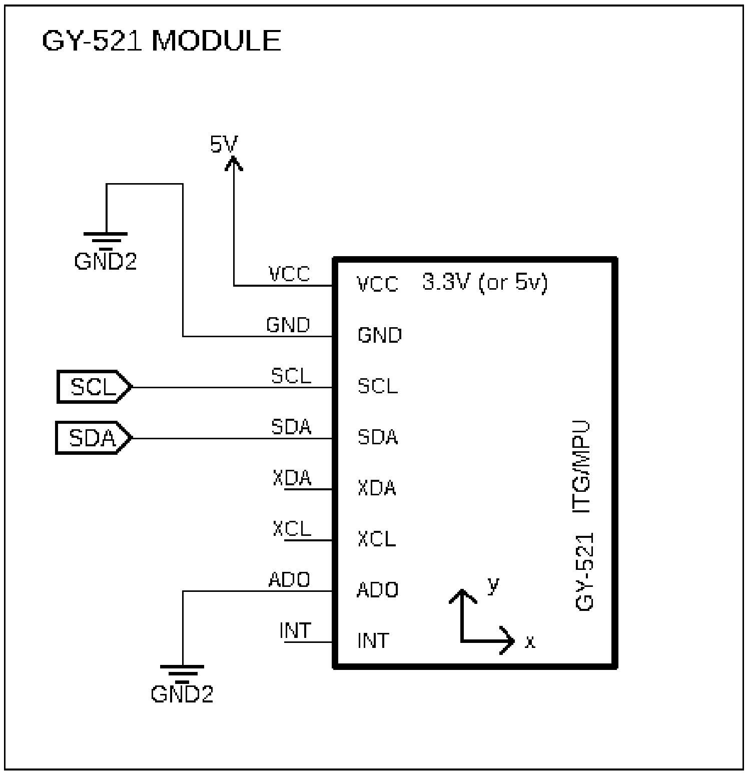


Figure 5: MPU-6050 DMP Circuit Block

An active buzzer was included as a means to audibly indicate that points had been scored. The buzzer is simply wired to a digital output on the MCU, *PORTL 0*. This pin is held low until points have been scored at which point it is brought high causing the buzzer to sound. The sound rings for approximately one second and is then stopped by bringing the pin low again. Figure 6 shows the circuit block for the active buzzer with relevant connections to the MCU.

The system is configured to run on 5V with minimal power consumption. Full System schematics may be found in the appendix.

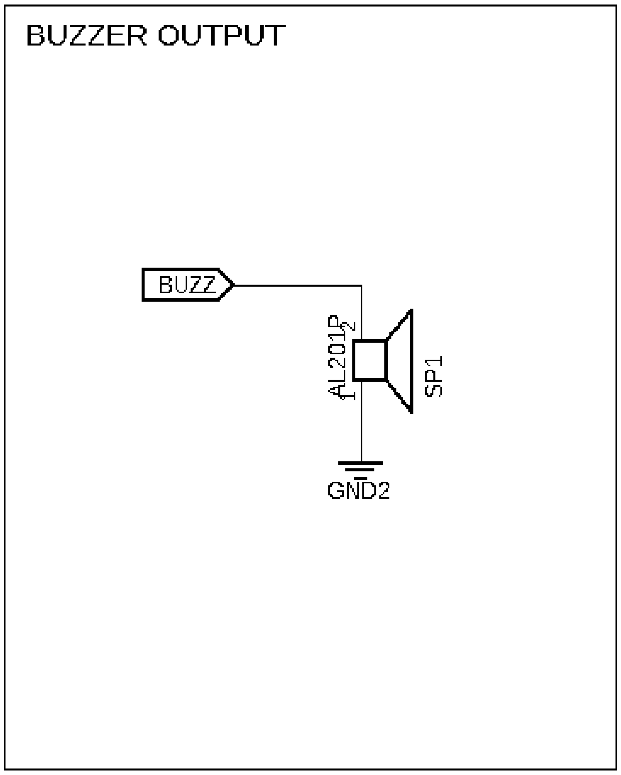


Figure 6: Active Buzzer Circuit Block

1.2 Software Design

The full gaming system exists as two separate processes that communicate with one another via serial data transmission. The frontend application handles the display of the graphical user interface (GUI) and dispatching services for the controller. The backend application, which runs on the MCU, monitors user input data and listens for dispatch requests from the frontend. Figure 7 Figure 8 are flowcharts drafted for each end of the full system. These abstractions helped assist the design process by clarifying the essential functions required by each end.

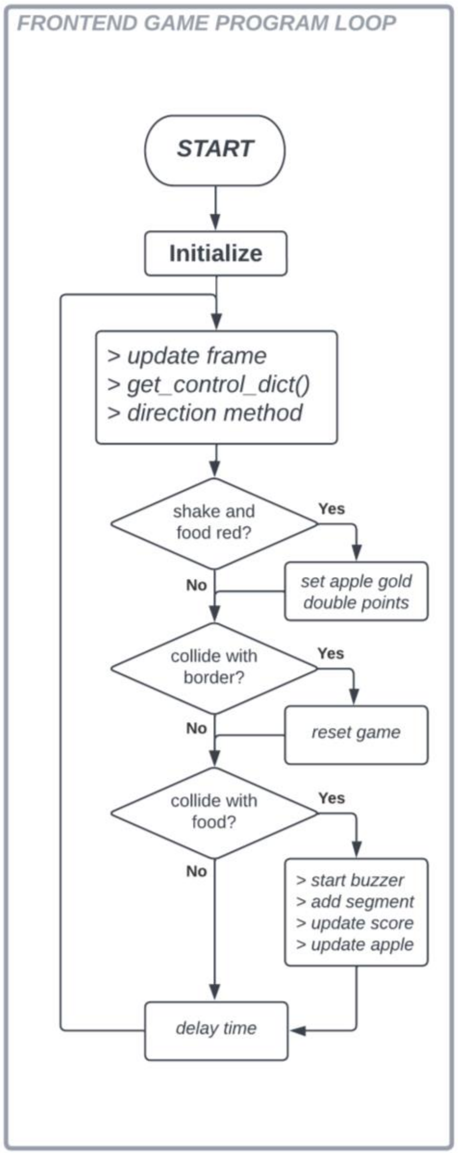


Figure 7: Front End Flowchart

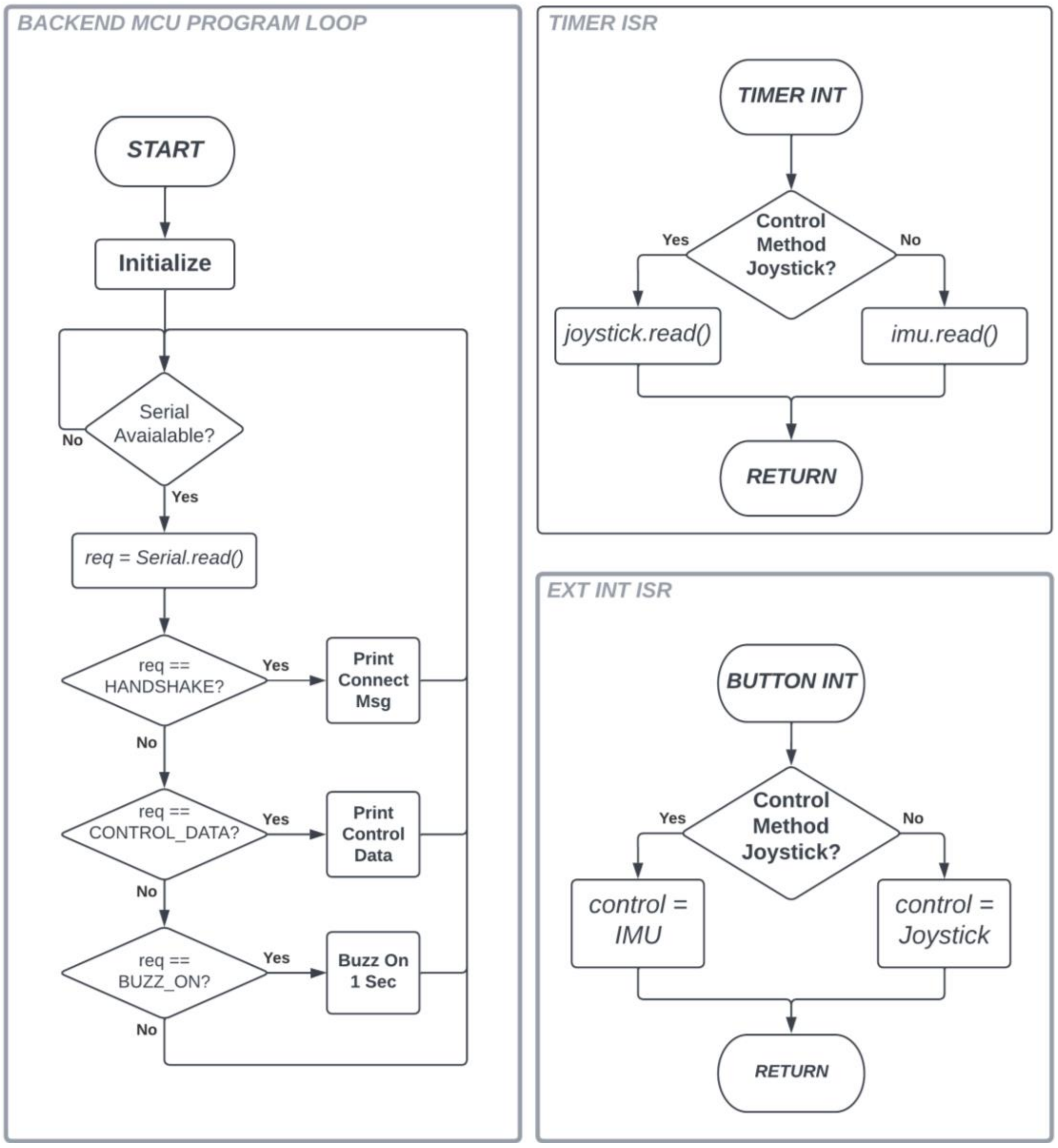


Figure 8: Backend Flowchart

1.2.1 Frontend – Game Application

The frontend program is implemented using two python scripts. The backbone of the system exists in "*main.py*". A template of this script was provided that handled running and displaying the snake game, originally authored by TokyoEdTech and modified by Yan Luo. Initially the script only allowed the game to be played using a computer keyboard, where *w*, *a*, *s*, and *d* sent control requests for *up, left, down,* and *right* respectively. Therefore, further modifications were necessary in order to allow control mechanisms that met the required specifications.

To provide a pipeline for serial data transmission between the "*pyserial*" library was used. This enables the front end to connect to the MCU via a communication (COM) port. When successfully connected the front end can read and write serial data to and from the backend. The core of this functionality was organized into a separate python script titled "*serial_control.py*" which contains a single class called *SerialController*. Initialization arguments for this calls included *device*, *baudrate*, and *timeout* which are defaulted to *'Arduino'*, *9600*, and *2* respectively. Once an object has been initialized using this class it may call "*find_device( )*". This class method searches all the connected COM ports and looks for the relevant device manufacturer, in the case of this application *'Arduino'*. If found it connects to it via *pyserial* and saves it as a class object called *dev*. This method of using the device manufacturer to connect enables the MCU to be plugged into any port and still be recognized. After calling the "*find_device( )*" the *"check_connection( )"* class method can be called. This will flush any existing serial data in the buffer, check to ensure the port is properly connected by sending a dispatch message (*REQ_HANDSHAKE = 0*) , and print the reply received from the MCU. After a successful connection has been verified the class has three more methods that can be called to control dispatching. Additionally, if the "*serial_control.py*" is run as the main program it will create its own *SerialControl* object, attempt to connect, and print out data received from the MCU. This was provided as a way to test for expected functionality without running the game.

Requests for user input data from the MCU are generated via the "*get_control_dict( )*" method can be called. This method sends a dispatch message (*REQ_DIRECTION = 1*), reads the reply, and returns the data. The reply message is sent/received as a string formatted in the style of a python dictionary (i.e. {dir: 'up', shake: 0}) which is parsed using json to transform it into a usable python dictionary object. To tell the MCU that the buzzer should be sounded, the "*start_buzzer*" method can be called. This method transmits a dispatch request (*REQ_BUZZER = 3*) that results in the buzzer being sounded. To synchronize "shake" control the "*reset_shake( )*" method can be called. This transmits a dispatch request (*REQ_SHAKE = 2*) which causes the MCU to set a shake variable to *false*. This is used to prevent duplicate shake messages from being transmitted while the game is already in the double points state.

The main program ("*main.py*") starts by importing relevant libraries and initializing objects used to display and control the game state. The GUI display is generated using the core python library "*turtle*". Four objects are initially created using this library: *wn*, *head*, *food*, and *pen*. The *wn* object is of the *Screen* class from *turtle*, this is the main window that hold all display objects and updates each frame. The other three objects are of the *Turtle* class from *turtle*. The *head* object is the beginning of the "snake" in the game and is what the player actually controls. Every time a player eats an apple a new *Turtle* object is added and stored in a list called *segments.* These *segments* follow the head and make up the snake's body. *The food* object is used to display the "apple" that the player is trying to "eat". The *pen* object is used to write text to the screen pertaining to the game state such as the current score. Frame updates take place every 100 ms and decrease each time an apple is eaten by 1 ms. Within a frame cycle the data from the MCU is read and dispatched, collisions of the snake head with are checked, and the snake's head and segments are moved.

To interface the MCU backend the main application creates a *SerialControl* object as described above. To use data from the MCU, a list called *dir_list* is created that stores four methods for controlling the snake's direction: *go_up*, *go_down*, *go_right*, and *go_left*. At the start of the main loop the "*get_control_dict( )*" is called and the direction value used as an index to call the correct direction method form *dir_list*. Following the shake value is checked, if it is 1 (*true*) the food color is changed to gold and the points are doubled. If the snakes head has collided with food the "*start_buzzer( )*" method is called to dispatch the MCU to sound the buzzer. Then the foods color is checked and if it is gold the "*reset_shake( )*" method is called to reset the shake Boolean on the MCU, the color is reset to red, and the point value is reset.

1.2.2 Backend – MCU

The core of the backend code lies in an Arduino program called "*GameController.ino*". This program initializes the objects and peripherals used to capture user input data and polls for available dispatch data from the frontend. Additional files and libraries were created to facilitate some of the more complex functionality required by the system: *axis*, *analog_joystick, mpu6050,* and *game_control*. All peripheral functionality was implemented using low level AVR programing such as direct *PORT* manipulation in an attempt to increase processing speed and better understand the internal workings of the ATmega2560.

The "*axis.h*" is a small file that containing a simple a data structure used to store positional data. This data structure is type defined as *Axis* and contains three float values *x*, *y*, and *z*, which are each initialized to zero. *Axis* is used in the main backend program as well as the custom *analog_joystick* and *mpu6050* libraries as a way to simplify the storage and sharing of data.

The joystick functionality was partitioned into a custom library named *analog_joystick(.h and .cpp)*. The header of this library contains a single class called *JoyStick* which is used read ADCs connected to the joystick module. Additionally, it contains definitions necessary for accessing the ADC peripherals at the low level (i.e., *#define ADC1 0b00000001*). Upon creation of a *Joystick* object the channels for x and y reading may be set to any of the 16 available ADCs using *ADCx*. However, by default they are set to *ADC0* for x and *ADC1* for y. Then channel references are stored as private class members, the debugging level is set as default off, and the ADC is setup. Initialization of the ADC peripheral set the voltage reference to the supply voltage, left adjusts for 8-bit resolution, sets the ADC clock prescaler to 128 (for a rate of 125 kHz), disables the digital functions of the chosen pins, and enables the ADC. Once initialized the *Joystick* object can call the "*read( )*" method which returns a value of the *Axis* type described above. Within this method both the x and y channels are read and stored into a buffer also of the *Axis* type and returned to the caller. Additionally if the debugging level is set to 1 it will print the values to the serial monitor. This was included during the development stage to help with troubleshooting. The debugging level can be set by calling "*setDebugLevel( )*" and providing it with one of the debug definitions defined in the "*axis.h*" file.

To assist in capturing data from the DMP™ device the custom library named "*mpu5050(.h and .cpp)*" was created. This library facilitates the communication of gyroscope and accelerometer data with the GY-521 breakout board via I2C protocol. Although libraries already exist to simplify the use of I2C (*Wire.h*) and enable communication with the DMP™ (*MPU6050.h*) it was thought that the creation of a custom library would be a good learning experience while at the same time stripping out some of the unneeded functionality. However, it should be noted that these libraries were studied to better understand how to achieve similar operation. The "*mpu6050.h*" contains a single class called *IMU* (internal measurement unit) as well as definitions for accessing register locations within the MPU-6050 device (i.e., *#define GYRO_XOUT_H 0x43*) as defined in its register map data sheet [5].

Upon creation of an *IMU* object the device address may be set, however, it is defaulted to *MPU6050_ADDR_LOW* (0x68) as defined in the *"mpu6050.h*" file. The data sheet for the MPU-6050 states that the device can have to different I2C address values by setting the *AD0* pin either high or low. This pin controls the least significant bit (LSB) of the device, therefore, it may also be *MPU6050_ADDR_HIGH* (0x69). After creating a *IMU* object the "*init( )*" method should be called. This method initializes the I2C peripheral on the MCU by setting the bit rate to 100 kHz. After that wakes the MPU-6050 device up by setting the *PWR_MGMT_1* register to zero, resetting the device by writing 0x80 to it, and then setting the clock to reference the gyroscope x-axis by writing 0x01 to it. Next the range values for the accelerometer and gyroscope are set by writing their range values to the appropriate registers on the MPU-6050. The range values for each control how sensitive the device is and have four different settings that can be found in the data sheet. Definitions for these setting are included in the "*mpu6050.h*" file. The ranges are defaulted to +/-250 deg/s for the gyroscope and +/-2g for the accelerometer. These can be changed at any time by calling the "*setRange( )*" method which takes arguments for the gyroscope and accelerometer ranges respectively. After the ranges are set the final responsibility of the "*init( )*" method is to test the connection. This simply writes to the *WHO_AM_I* (0x75) register and reads the reply which is expected to be 0x68.

After the *IMU* object has been successfully initialized it should be calibrated by calling the "*calibrate( )*". This method reads a number of values (default 1000 can be changed by sending an argument) from each axis on the gyroscope and accelerometer and then calculates the mean to use as an offset value on future readings. This allows for a higher level of accuracy and better control. It was noticed during initial testing that the values being received for each axis were not as expected. For example, it was thought that when lying flat on the take the x-axis gyroscope readings would be 0, however, they were found to have a sizable offset. Using this calibration technique resulted in the idle values being much closer to what was expected. It should be noted that for proper calibration the device should be sitting flat on a table until the process is complete.

Within the main backend loop the only *IMU* method that is called is "*read( )*". This method reads all the output data from the MPU-6050 via I2C. However, although it saves all data for troubleshooting purposes, it only returns the x and y values from the gyroscope and the x value from the accelerometer. The values it receives are raw values that do not properly describe the real-world units for the gyroscope (deg/s) and accelerometer (g). Therefore, these the raw values are divided by a conversion factor that relates to their range as described in the MPU-6050 datasheet. The relevant values are then returned as an *Axis* data structure to the caller. The *IMU* class includes a number of other reading functionality used for development and troubleshooting, however, these will not be described as they do not pertain to the current requirements of the game system.

For testing the debug level may be set by calling "*setDebugLevel( )*" with one of the 10 debug definitions defined in "*mpu6050.h"*as an argument. By default, the class is set to *DEBUG_OFF* and therefore will not transmit any debugging data. The debugging levels enable the class to print data to the terminal for an area of interest. For example, setting the debug level to *DEBUG_RAW_GYRO* will cause the raw gyroscope data to be printed to the terminal every time that "*read( )*" is called.

To cover all bases the I2C operation within the *IMU* class will be briefly described. To begin an I2C transaction the private method "*i2cStart_( )*" is called where the MCU pulls the SDA line low while the SCL line is high. Next the MCU loads the DMP™ modules address into TWDR7:1 data register along with a read/write bit into TWDR0. Read and write operation is done by setting TWDR0 to 1 or 0 respectively. The process is achieve using the private *IMU* methods "*i2cSlaveWrite_( )*" or "*i2cSlaveRead ( )"*. When DMP™ module has successfully acknowledged its addressing data may be shared with it. To write data to the module the "*i2cWrite_( )*" method is called with the byte of data to be written as an argument. The read operation offers two different private class methods, "*i2cReadACK( )*" and "*i2cReadNACK( )*". The first of the two read operations transmits an acknowledgment (ACK) to the module once data has been successfully received. The second transmits a not acknowledged (NACK) to the module once data has been successfully received, it used after the last byte has been read before ending the transaction. Data received is returned to the caller from the TWDR register. To end the transaction "*i2cStop( )*" is called which brings the SDA line high while the SCL line is low. A general example of how read and write processes should take place is shown by Figure 9 Figure 10 as described within the datasheet for the MPU-6050 [4].

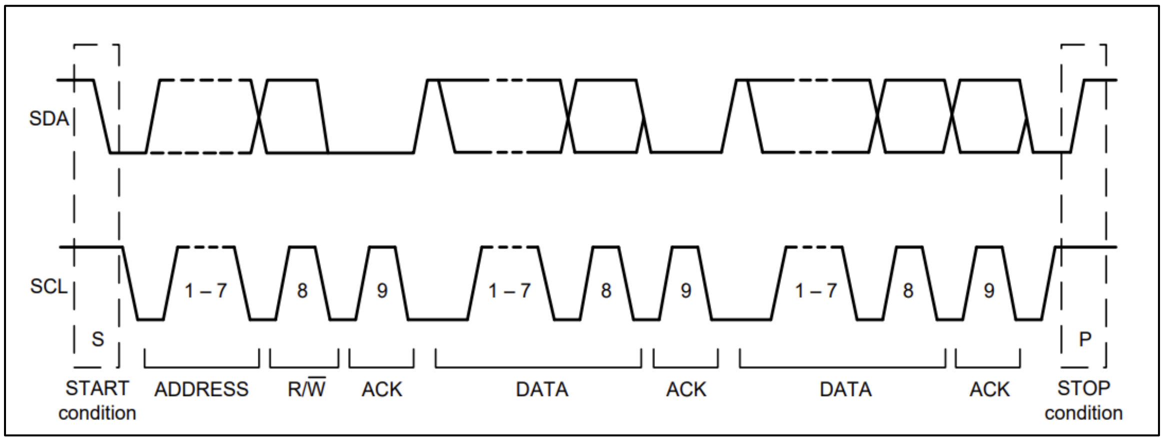
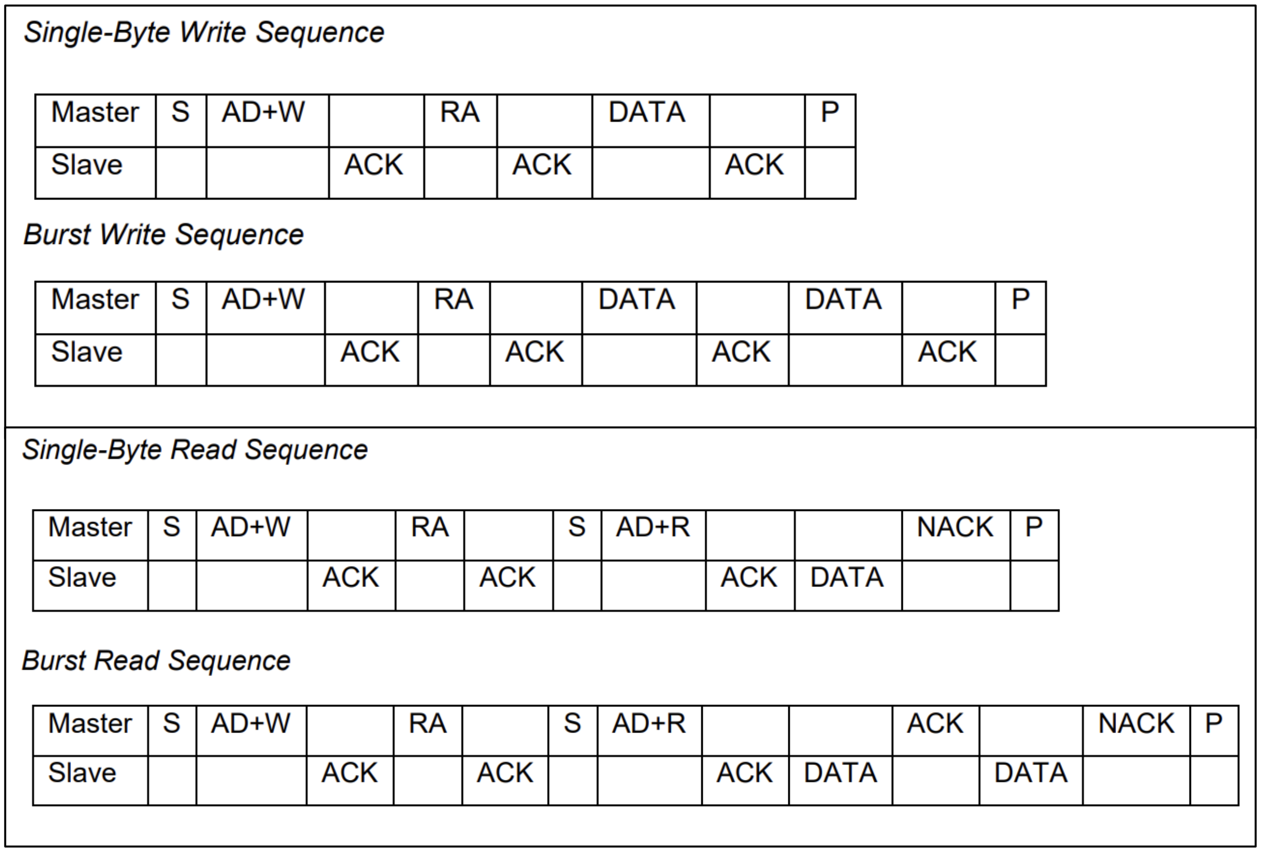


Figure 9: 2C Data Transfer Sequence [4]



Single-Byte Write Sequence

Master	S	AD+W		RA		DATA		P
Slave			ACK		ACK		ACK	

Burst Write Sequence

Master	S	AD+W		RA		DATA		DATA		P
Slave			ACK		ACK		ACK		ACK	

Single-Byte Read Sequence

Master	S	AD+W		RA		S	AD+R			NACK	P
Slave			ACK		ACK			ACK	DATA		

Burst Read Sequence

Master	S	AD+W		RA		S	AD+R			ACK		NACK	P
Slave			ACK		ACK			ACK	DATA		DATA		

Figure 10: MPU-6050 I2C Read Write Sequence [4]

1.3 Results

A completed prototype built on breadboard is depicted in Figure 11. Colored jumper wires were used to clarify signal nets and the +5V power supply was taken from the ATmega2560 breakout board module. Individual circuit blocks were partitioned to help organize the overall build, Figure 12. Code segments were developed to test each part of the circuit individually, before implementing the system as a whole. These can be found in the git repository withing the test directory, which contains files for testing the joystick, DMP™ module, and serial data transfer using python. After bugs were fixed for each circuit block the code was combined and validated through further testing of the system whole.

Improvements could be made if this system were to be actually implemented under the same specifications. The biggest and most obvious would be using a smaller MCU device. The ATMega2560 is a bit overkill for this project given the minimal use of digital I/O and peripherals. Suitable replacements would be the ATmega32u4 [6] or ATmega328 [7] which cost $5.05 and $2.26 respectively for a TQFP package. These come in at over half the cost when compared to the ATmega2560 TQFP package at $18.86 [8]. Furthermore, if given a larger window of time for development the *IMU* library could be ironed out for more reliable functionality. Currently if the device is unplugged and repowered while the MCU code is running it will not reconnect and therefore not share data with the system. This is due to it needing to be woken up again by writing 0x00 to the PWR_MGMT_1 register. A possible fix for this may take shape by monitoring I2C transactions and flagging an error within the *IMU* class. Once an error is flagged it could attempt to reinitialize the device and then clear said flag upon success.

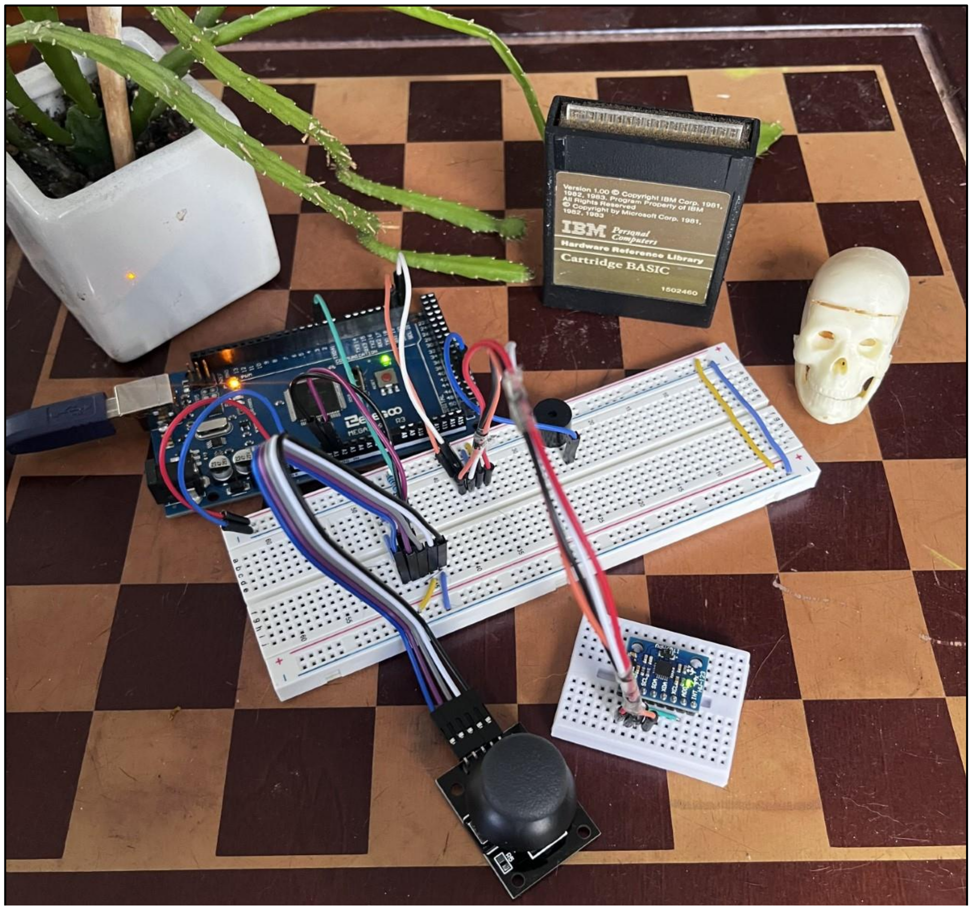


Figure 11: Game Controller Breadboard Prototype

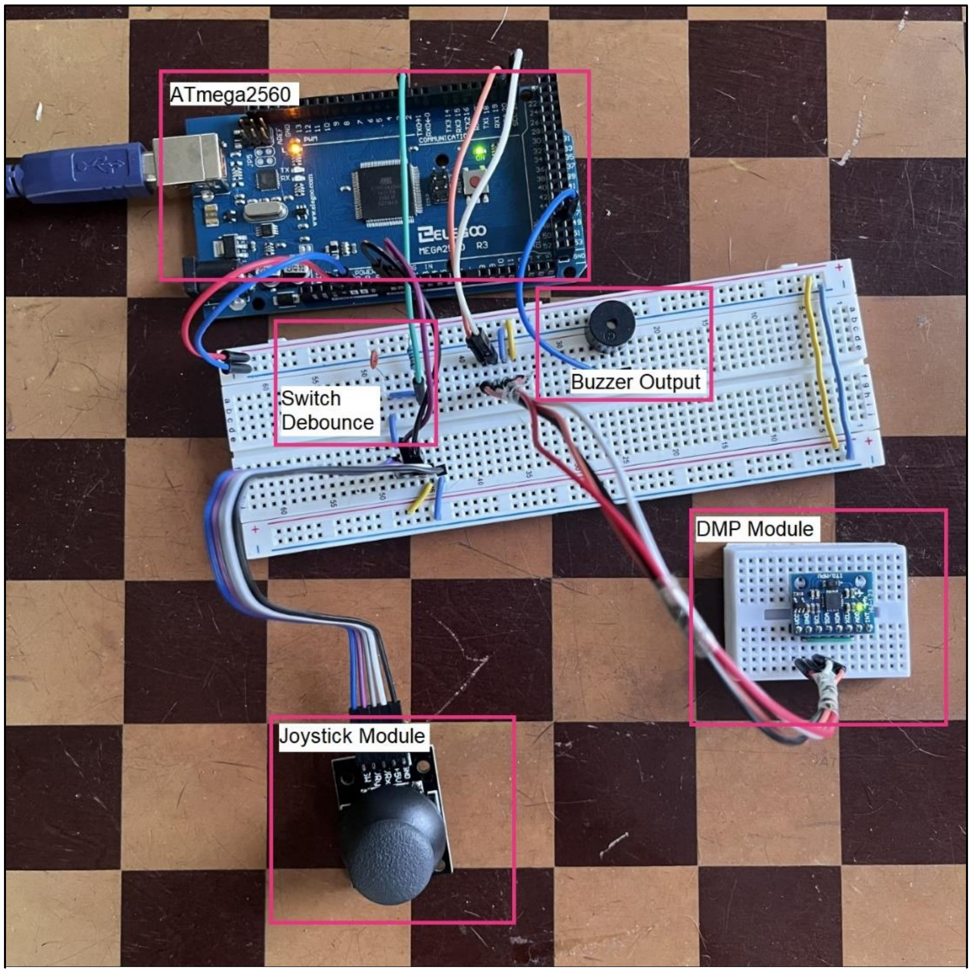


Figure 12: Game Controller Breadboard Circuit Blocks

2. Problems Encountered and Solved:

During initial development of the *Joystick* class, it was observed that the values being obtained were not changing when moving the joystick module to different positions. Reading the MCU documentation [2] revealed that the ADC data registers were being accessed incorrectly. The ATmega2560 splits the 10-bit value into two 8-bit registers, ADCH and ADCL, and states that ADCL must be read before ADCH otherwise the data registers will not be updated. Initially the implemented code reversed the necessary read order. However, when researching this issue it was discovered that an AVR macro exists for combining the separate data registers into a 16-bit word during read operations [9]. This was the chosen solution for the final code.

Development of the *IMU* class presented many roadblocks that needed small attention to detail and proper understanding of both I2C and MPU-6050 operation. The first issue took shape in the form as a result of incorrectly using status codes to detect errors during I2C operations. This was simply a misunderstanding of the differences in status codes for transmit and receive modes of the MCU and remedied through rereading the relevant material in the datasheet. The next big issue took shape when the DMP™ module was sending ACKs when being addressed but not sending data. It was discovered that the device must be woken up to properly operate by writing 0x00 to the PWR_MGMT_1 register. Performing this step upon initialization of the class allowed for the expected sharing of data. The last big problem encountered during development of this class related to the data being received for accelerometer and gyroscope measurements. It was initially though that the values would read 0 while the device was level on a table. However, there was a large degree of error which made it hard to understand how the values could be used to implement the required control mechanisms. Researching the topic revealed that determination of the average error could be used as an offset [10]. After implementing the calibration error described above the values were much closer to 0 when idle. This made the task of actually using the data much easier to understand.

When testing and developing the *SerialControl* python class it was thought that the most effective way to share control data would be using a python dictionary. If the MCU was to send data in this format it could simply be accessed using the relevant dictionary keys. Initially it was thought that formatting the data on the MCU side in such a way that it resembled a python dictionary would suffice. However, it was quickly realized that this would not work as python still interpreted the data a string. Researching the issue revealed that the JSON library can be used within python to realize strings as dictionaries assuming they are formatted as such [11]. Using this method proved to work as expected and was implemented in the final design.

3. Personal Contribution to the Lab (Technical Details):

The graduate section of this class (EECE.5520) allows students to work solo on lab projects. This was the case for this project, therefore all contributions discussed above were contributed by the author Ellis Hobby. Full schematic files and project code can be found under the *Lab2* directory in the github repository at: https://github.com/EllisHobby/Microprocessors-Labs-2023. This repository is currently private; therefore, access will need to be requested to review the files. To request access to the repository please email the author at: ellis_hobby@student.uml.edu.

4. Lessons Learnt:

Choosing to implement this system using low level AVR access proved to be a big challenge. However, a much greater understanding of the internal operation of the ATmega2560 was gain, specifically those pertaining to I2C and ADC functionality. This required a firm grasp on the concepts presented within the MCU datasheet.

Implementation of the ADC functionality appeared to be a daunting task when first approached. However, once understood it is quite simple and explained very well within the ATmega2560 datasheet [2]. First the user must understand what ADC pins are being used and ensure the ADMUX and ADCSRB registers are properly set before attempting to perform a conversion. The reason for this lies in the fact that there is really only one ADC within the MCU, therefore, it must be switched to the desired input using the multiplexer. Additionally, care should be taken to mask the registers when switching inputs in order to maintain data bits within them unrelated to the multiplexing. For example, bits 7 and 6 within ADMUX set the voltage reference, if they were to be changed before a conversion the difference in reference would result in an incorrect result. Next it is imperative to understand the ADCSRA register, which allows the peripheral to be enabled, starts a conversion, monitors the conversion complete flag, and sets the prescaler clock. Furthermore, it must be realized that the conversion is not instantaneous, and the program must wait util the conversion flag bit, ADIF, has been set. Lastly the data registers should be accessed in the proper order otherwise they will not be updated. ADCL must be read before ADCH in order to retrieve the correct value.

Development of the *IMU* class required a strong understanding of both I2C transactions and the internal operation of the MPU-6050. For I2C transactions it is necessary to follow the correct sequence of events as depicted in Figure 9. Furthermore, it is important to monitor the status of these transactions to ensure that the data being sent and received is correct using the TWSR register on the MCU. Similar to the ADC the I2C module does not perform instantaneous transactions. Therefore, the program must wait for completion by monitoring the interrupt bit, TWINT, within the TWCR register on the MCU. For operation of the MPU-6050 it is imperative that the register map data sheet be studied [5]. In order to retrieve the measured values, the correct data registers must be accessed through a write operation before reading occurs. For example, to read the x-axis of the gyroscope one must start an I2C transaction, write the corresponding register address, stop the transaction, and then perform another I2C sequence for reading. Additionally, as explained above the device must be woken up before any operations occur, otherwise it will stay in sleep mode and not recognize commands via I2C.

Programming the system using the presented methods proved much more difficult when compared to implementation using the equivalent Arduino libraries. However, eliminating the high level of abstraction that these libraries provide leads to a much greater understanding of the hardware's low-level operation. Furthermore, although the implementation of these mechanisms may not be exactly the same from one device to another, the core concepts that support how they work remain. Therefore, understanding how implementing the same functionality on a different will be better understood in the future.

References

[1] Y. Lou, *Lab 2: Game Control with Joy Stick, Gyro and Accelerometer,* Lowell: Massachusetts, 2023.

[2] Microchip, *ATmega640/1280/1281/2560/2561 - Complete Datasheet,* Microchip, 2000.

[3] T. Instruments, *Debounce a Switch,* Dallas: Texas Instruments Incorporated, 2020.

[4] InvenSense, *MPU-6000/MPU-6050 Product Specification,* InvenSense, 2013.

[5] InvenSense, *MPU-6000/MPU-6050 Register Map and,* InvenSense, 2013.

[6] Microchip, "ATmega32u4," Microchip, [Online]. Available: https://www.microchip.com/en-us/product/ATmega32U4.

[7] Microchip, "ATmega328," Microchip, [Online]. Available: https://www.microchip.com/en-us/product/ATmega328.

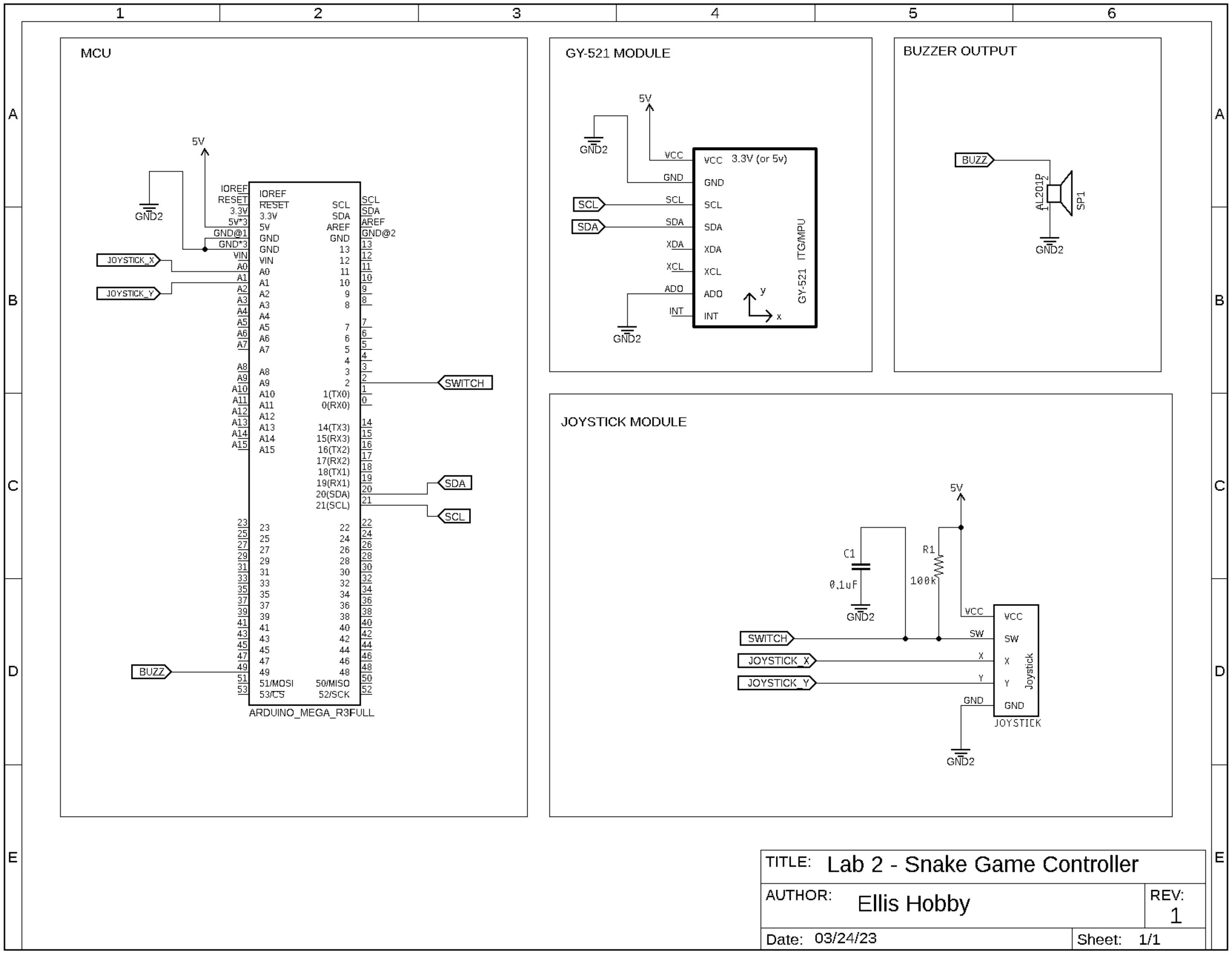
[8] Microchip, "ATmega2560," Microchip, [Online]. Available: https://www.microchip.com/en-us/product/ATMEGA2560.

[9] B. Thomsen, "Analogue Input," [Online]. Available: https://bennthomsen.wordpress.com/arduino/peripherals/analogue-input/.

[10] S. Bluett, "Calibrating & Optimising the MPU6050," 2015. [Online]. Available: https://wired.chillibasket.com/about/.

[11] Python, "json — JSON encoder and decoder," 2023. [Online]. Available: https://docs.python.org/3/library/json.html.

Appendix


MCU
GY-521 MODULE
BUZZER OUTPUT
JOYSTICK MODULE
5V
GND2
JOYSTICK_X
JOYSTICK_Y
BUZZ
SWITCH
SDA
SCL
ARDUINO_MEGA_R3FULL
VCC
GND
XDA
XCL
ADO
INT
3.3V (or 5v)
GY-521 ITG/MPU
AL201P
SP1
C1
0.1uF
R1
100k
SW
X
Y
JOYSTICK
TITLE: Lab 2 - Snake Game Controller
AUTHOR: Ellis Hobby
REV: 1
Date: 03/24/23
Sheet: 1/1