Tampere University

Joakim Unha

REMOTE CONTROL OF A TWO-WHEELED SELF-BALANCING VEHICLE VIA ACCELEROMETER DATA MANIPULATION

Hardware and Software Implementation

Bachelor's Thesis
Faculty of Engineering and Natural Sciences
Examiner: Nikolay Serbenyuk
May 2025

ABSTRACT

Joakim Unha: Remote Control of a Two-Wheeled Self-Balancing Vehicle via Accelerometer Data Manipulation
Bachelor’s Thesis
Tampere University
Bachelor’s Programme in Automation Engineering
May 2025

Two-wheeled self-balancing vehicles have been used for human transportation for quite some time and the interest in turning them into autonomous robots has existed for just as long. This thesis investigates the feasibility of converting a two-wheeled self-balancing vehicle into remotely controllable system without any permanent alterations to the vehicle, including the original firmware. The approach involves inserting a microcontroller between the inertial measurement unit and the main processor to intercept and manipulate sensor data as a form of control.

Keywords: Segway, Two-Wheeled Self-Balancing Vehicle, IMU, I2C, ESP32, Citybot Offroad Self-Balance

The originality of this thesis has been checked using the Turnitin Originality Check service.

TIIVISTELMÄ

Joakim Unha: Etäohjauksen integrointi kaksipyöräiseen itsestabiloituvaan ajoneuvoon kiihtyvyys anturin manipuloinnin avulla
Kandidaatintyö
Tampereen yliopisto
Automaatiotekniikan tutkinto-ohjelma
Toukokuu 2025

Kaksipyöräisiä itseään tasapainottavia laitteita on käytetty liikkumiseen jo pitkään, ja niistä on haluttu tehdä itsestään ohjautuvia lähes yhtä kauan. Tämä kandidaatintyö tutkii mahdollisuutta muuntaa kaksipyöräinen itseään tasapainottava ajoneuvo etäohjattavaksi järjestelmäksi ilman pysyviä muutoksia itse ajoneuvoon tai alkuperäiseen laiteohjelmistoon. Menetelmään kuuluu mikrokontrollerin sijoittaminen inertiamittausyksikön ja suorittimen väliin sieppaamaan ja muokkaamaan sensoridataa systeemin ohjauksena.

Avainsanat: Segway, kaksipyöräinen itsestabiloituva ajoneuvo, IMU, I2C, ESP32, Citybot Offroad Self-Balance

Tämän julkaisun alkuperäisyys on tarkastettu Turnitin Originality Check -ohjelmalla.

USE OF AI IN THESIS

I have utilised AI tools in my thesis:

- ☐ No
- ☒ Yes

The AI tools utilised in my thesis and their purposes are described below:

Names and versions of AI tools: ChatGPT 4o, Scopus AI

Purpose of using AI tools: ChatGPT 4o was used for proofreading and sentence structure improvements. Scopus AI was used to search for references in the planning phase.

Sections where AI tools were used: Every chapter was proofread and almost every chapter has had sentence structure improvements.

I acknowledge that I am fully responsible for the entire content of my thesis, including the parts generated by AI, and accept accountability for any violations of ethical standards in publications.

PREFACE

This topic was chosen because of my interest in learning about practical work with real hardware and it provided valuable experience in both software and hardware development.
I would like to thank Nikolay Serbenyuk for the technical help he provided during every step of the project.

Tampere, 19.5.2025

CONTENTS

LIST OF SYMBOLS AND ABBREVIATIONS

ACK	Acknowledgement Bit
Ay	Acceleration along y-axis
Ax	Acceleration along x-axis
Az	Acceleration along z-axis
CPU	Central Processing Unit
EMI	Electromagnetic Interference
HIGH	A logical 1, where the data line is at the supply voltage
I2C, IIC	Inter-Integrated Circuit
IMU	Inertial Measurement Unit
LOW	A logical 0, where the data line is at ground (0 V)
MC	Microcontroller
MCU	Microcontroller Unit
MPU	Microprocessor Unit
NACK	Negative or Not Acknowledged Bit
RAM	Random Access Memory
ROM	Read Only Memory
SCL	Serial Clock
SDA	Serial Data
VDD	Voltage Device Drain

1. INTRODUCTION

Advancements in the development of two-wheeled self-balancing transportation vehicles, popularized by the Segway, have sparked interest in creating autonomous versions of these machines. When equipped with LiDARs or visual-based sensors, two-wheeled self-balancing robots could potentially be used for a wide range of applications, such as deliveries, warehouse logistics, and security or surveillance tasks.

Mobile robots with three or more wheels have better static stability and simpler dynamics, but they require a suspension system unless the ground is flat. On the other hand, two-wheeled systems do not require suspension and are highly maneuverable, as they can rotate 360 degrees in place, making them ideal for navigating tight spaces. [1, p. 90]

The goal of this technical study is to determine whether it’s possible to alter a two-wheeled self-balancing vehicle into a remote controllable device while keeping it as original as possible. This means that there cannot be any permanent modifications to the device or its firmware. Due to these limitations, the method used becomes very challenging and impractical. For simplicity, this study focuses solely on achieving forward and backward movement, without addressing rotational control. The two-wheeled self-balancing vehicle used in this study is “Citybot Offroad Self-Balance” made by Shenzhen Easy Step Robot Co., see figure 1.

Figure 1: *Citybot Offroad Self-Balance*

2. BACKGROUND AND RELATED WORK

The core concept of this method is to integrate a microcontroller within the device to modify accelerometer data measured and sent by the Inertial Measurement Unit (IMU). Given the self-balancing nature of the vehicle, motor control can be achieved by providing false state data, influencing its movement. However, since this project is highly unique and lacks prior examples, it is uncertain whether this method will remain stable and whether the vehicle can reliably drive forward or backward. Therefore, the validity of this approach must be verified through experimentation.

2.1. Segways and Self-Balancing Technology

The Segway, a self-balancing mode of transportation, was invented by Dean Kamen in 2001 [2]. Since then, many companies have introduced similar vehicles with handlebars, commonly referred to as “Segways”. For the sake of clarity, the paper uses the term “two-wheeled self-balancing vehicle” to address them.

The two-wheeled self-balancing vehicle is in its core an inverted pendulum on two wheels. When the person on board pushes or pulls the handlebar, their center of gravity shifts beyond the point of contact between the wheels and the ground, causing the vehicle to tilt. The system within the self-balancing vehicle counteracts this by continuously adjusting the wheels to remain beneath the rider’s center of mass, resulting in the vehicle moving forwards or backwards. Turning is achieved by rotating the handlebar either to the right or left, causing one motor to move forward and the other to move backward. [3]

2.2. Inter-Integrated Circuit

Inter-Integrated Circuit (I2C or IIC) is a serial communication protocol developed by Philips Semiconductors in the 1980s for short-distance data exchange between microcontrollers and peripherals, and is now maintained by NXP Semiconductors, which acquired Philips' semiconductor division. Serial communication is a method of transmitting one bit at a time over a communication channel or bus. The protocol operates using only two bidirectional lines: a serial data line (SDA) and a serial clock line (SCL). Each device connected to both lines can be accessed by a unique address. [4]

I2C follows a master-slave architecture meaning a device can either be a master initiating communication, generating clock signals that determine the transaction speed and requesting data, or a slave responding to the requests accordingly [4,5]. NXP Semiconductors, as part of its Inclusive Language Project, updated the terms “master” and “slave” to “controller” and “target” in the 7th revision of the I2C Bus specification and user manual in October 2021 [4]. This study adopts the updated terminology accordingly.

The SCL and SDA lines are connected to a positive supply voltage through pull-up resistors as shown in figure 3. When there is no communication on the bus, both lines are at HIGH, meaning the voltage on both lines is at the supply voltage level (e.g., 3.3 V or 5 V). Devices can pull the lines LOW by driving them to ground (0V), overriding the pull-up resistors. Since I2C is an open-drain system, a device can only pull the lines LOW and release them but never drive them HIGH [4]. The SCL line determines the speed of the communication, while the SDA line determines the content. When the SDA line is LOW for the duration of SCL HIGH, it represents a logical 0. Conversely, when SDA is HIGH for the duration of SCL HIGH, it represents a logical 1, see figure 2. These 1’s and 0’s are called bits, and eight bits form a byte.

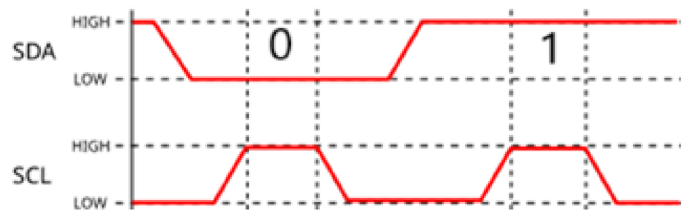

Figure 2: *Bit transfer on the I2C bus. During the first SCL HIGH, SDA is LOW, representing a logical 0 and HIGH on the next one, representing a logical 1. Original picture modified by adding numbers for clarity. [6]*

As stated in the official NXP I2C Specification [4], the generation of clock signals is always the responsibility of the controller device. Each cycle of the clock signal consists of a clock pulse, which includes a HIGH and LOW period. The duration of a single clock pulse is determined by the clock frequency. In Standard mode, the clock frequency can reach up to 100 kHz, resulting in a maximum data transfer rate of 100 kbit/s. In Fast mode, the clock frequency can go up to 400 kHz, allowing for a maximum data rate of 400 kbit/s. Finally, in High-speed mode, the clock can reach 3.4 MHz, enabling data transfer rates of up to 3.4 Mbit/s. The SDA line must stay stable during the HIGH SCL period, meaning SDA can only change states during a LOW SCL period, see figure 2. [4]

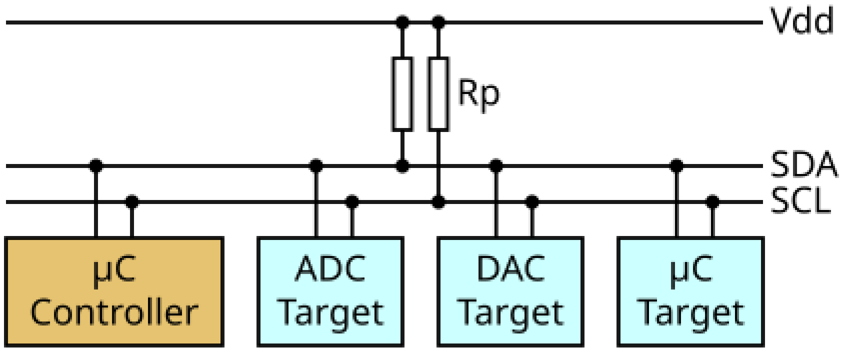

Figure 3: *Single-Controller Multi-Target setup with external pull-up resistors [7].*

I2C is capable of multi-controller and multi-target operation, but it is mostly used for a combination of single-controller and multi-target [5]. Since it's possible for multiple controllers to try and initiate data transfer simultaneously, bus arbitration has been developed to prevent data corruption. As noted in the NXP-Semiconductors document [4], a controller can only start a transfer while the bus is free. If it attempts to release SDA line, but notices that the line stays LOW, it realizes another device is holding it down and in response, stops transmitting and waits for its turn. [4]

The data transmission over the I2C bus can be divided into several steps, see figure 4 and 5:

- Start condition: A unique case where SDA is released from LOW to HIGH during a HIGH SCL period. This signals to the devices that a data transaction is about to start.

- Target device address and R/W bit: a 7-bit long address of target slave device is transmitted with an additional bit, which can be either 0, representing Write or 1, representing Read. This bit determines whether the controller device wants to send or receive data. They are then combined into a single byte.

- ACK/NACK: target device will pull the SDA LOW as a sign of acknowledgement (ACK) for the duration of acknowledgement clock pulse. When a target device doesn’t pull the SCL LOW during the specific clock pulse, it’s interpreted as a negative acknowledgement (NACK) bit.

- Data transfer: A byte (8 bits) is sent over the SDA line, followed by an ACK or NACK. This process repeats until a stop condition is detected.

- Stop condition: SDA line is released from LOW to HIGH and SCL stays at HIGH, signaling that the bus is free for communication. [4]

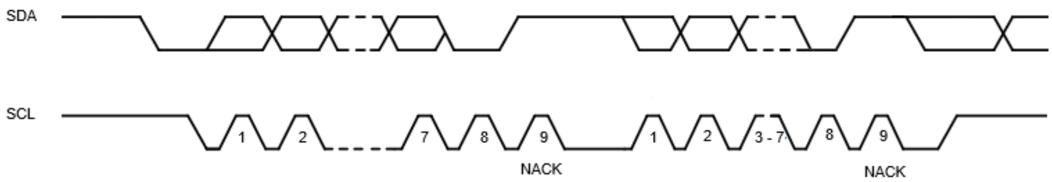

Figure 4: *Data communication on the I2C-bus.*

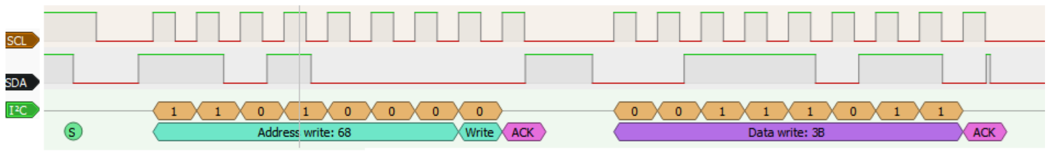

Figure 5*: Data transaction within the Citybot's I2C bus, captured using PulseView software and an AZ-Delivery logic analyzer.*

As seen in figure 5, the communication begins with Citybot's Atmega32 microprocessor pulling SDA low while SCA is HIGH, claiming the bus. After that seven address bits are sent (1101000, converted to 0x68 in hexadecimal) followed up by a Write bit, signaling that the controller device wants to send data to the address 0x68. That is followed by the target device responding with an ACK bit, after which the controller sends a byte of data and again the target responds with an ACK bit.

2.3. Microcontrollers

Microcontrollers (MCs) are computer chips that are used in a variety of consumer products and commercial applications (e.g. automobiles, microwaves, cameras, robots etc.) [8,9]. They include a Central Processing Unit (CPU), digital input/output capabilities, Random-Access-Memory (RAM), Read-Only-Memory (ROM), serial communication interface and various integrated resources. The CPU runs the software stored in ROM and manages all microcontroller components. RAM temporarily holds settings and values required by the executing program, while ROM stores the program and any permanent data, since it's non-volatile, meaning it doesn't lose any data when turned off. Microcontrollers units (MCUs) refer to the same chips, but the term 'unit' emphasizes that the microcontroller is integrated into a larger system or device (e.g. ESP32 Development Board). [8, p. 258-265]

The MCU used in this project is the ESP32 DEV KIT V1. As the official Espressif ESP32 technical reference manual [10] states, the ESP32 is a dual-core system featuring two Xtensa LX6 CPUs based on Harvard Architecture. It contains 448 KB of ROM, 520 KB of RAM and 4 MB of flash memory.

2.4. Inertial Measurement Units

Inertial Measurement Units (IMUs) are electronic devices used practically in any autonomous technology or in machines requiring position tracking. They measure and report an object's acceleration, angular rate, and in some cases, the surrounding magnetic field. IMUs typically consist of a combination of accelerometers and gyroscopes. The gyroscopes' output is used to determine the orientation in space and accelerometers' output is used for acceleration and position tracking. These primary sensors are arranged within a three-dimensional coordinate system, where each axis of the device corresponds to the sensitivity axes of the accelerometer and gyroscope. An IMU with these components offers six degrees of freedom, enabling it to track motion across all directions. [11]

Tilt refers to an object's orientation relative to the Earth's surface and can typically be described in terms of three angles: pitch (ρ), roll (φ) and yaw (θ), as illustrated in figure 6. According to Tuck [12], pitch is defined as the angle between the x-axis and the ground, roll as the angle between the y-axis and the ground, and yaw as the angle between the z-axis and the ground. These angles can be calculated in radians from accelerometer readings along the x, y and z axes (A_X, A_Y and A_Z) using formulas 1, 2 and 3: [12]

$$\rho = \arctan\left(\frac{A_X}{\sqrt{A_Y^2 + A_Z^2}}\right) \tag{1}$$

$$\varphi = \arctan\left(\frac{A_Y}{\sqrt{A_X^2 + A_Z^2}}\right) \tag{2}$$

$$\theta = \arctan\left(\frac{\sqrt{A_X^2 + A_Y^2}}{A_Z}\right) \tag{3}$$

For two-wheeled self-balancing vehicles, the orientation and alignment of the IMU determines the required angle for balancing. In the case of the Citybot, the IMU is positioned such that the pitch (ρ) is the only critical angle for balancing, see figure 6.

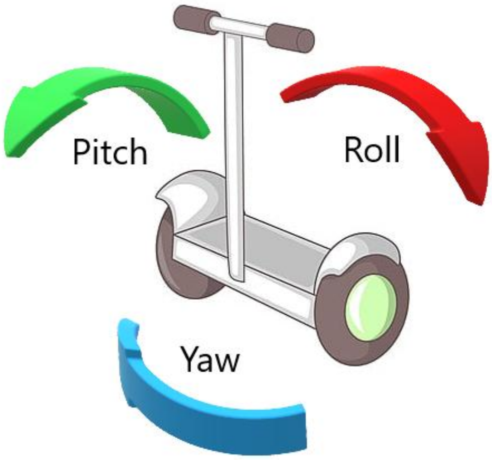

Figure 6: *Pitch, roll and yaw illustrated on a two-wheeled self-balancing vehicle with the IMU installed in the same orientation as in the Citybot.*

For the IMU, the Citybot uses a microprocessor unit 6050 (MPU-6050) mounted on a GY-521 breakout board, see figure 7. According to its inventor InvenSense [13], the MPU-6050 is the world's first six-axis motion tracking device. It combines the three-axis gyroscope and three-axis accelerometer with a digital motion processor used for onboard sensor fusion and motion processing. The MPU-6050 operates with a supply voltage (VDD) ranging from 2.375 V to 3.46 V [13]. However, it is often used with 5 V systems through the GY-521 breakout board's voltage regulator, see figure 8.

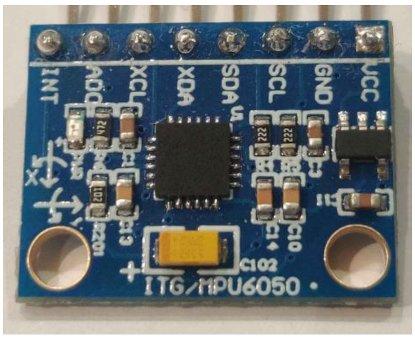

Figure 7: *GY-521 breakout board with MPU-6050 chip in the middle.*

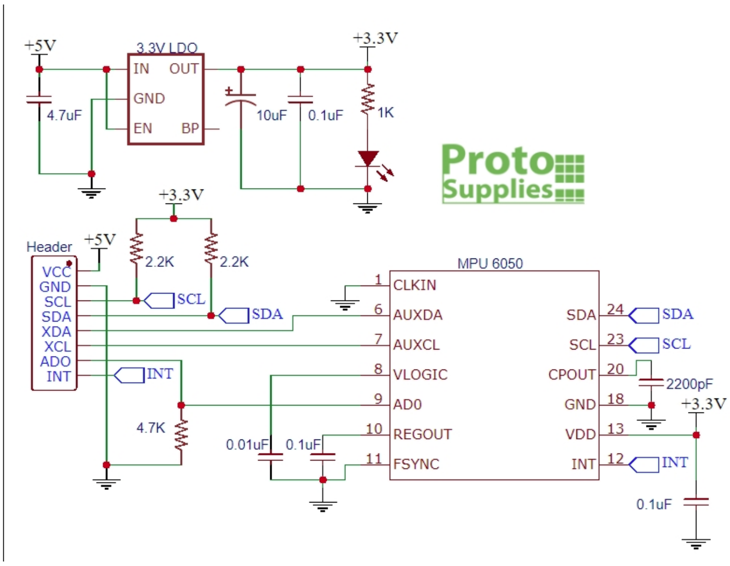

Figure 8: *Schematic diagram of the GY-521 breakout board, with the voltage regulator at the top [14].*

The MPU-6050 communicates through I2C and has a hexadecimal address of 0x68. By connecting the breakout board’s AD0 pin to a supply of 3.3 V or 5 V, the address can be changed to 0x69, as noted by InvenSense [13]. This feature is particularly useful when connecting two different MPU-6050’s to the same I2C bus, allowing for unique addressing to avoid conflicts.

3. TECHNICAL INFORMATION AND PLANNING

This section introduces the Citybot and its key components. It offers a detailed overview of the design choices and planning process. The difficulties related to maintaining the Citybot's original functionality while incorporating new features are also addressed.

3.1. Components and Technical Information

The Citybot Offroad Self-Balance was made and sold by Shenzhen Easy Step Robot Co., however the site where it was purchased doesn't exist anymore and there are no technical documents available.

The device is powered by a 38 V DC battery and features two 36 V DC parallel shaft gear motors, see figure 9. It also includes a motherboard containing an Atmel Atmega32A-AU microprocessor, along with various supporting components. A GY-521 breakout board with an MPU-6050 is used for accelerometer data readings, while the system also includes a potentiometer to monitor handlebar rotation. Four safety switches, that prevent the motors from activating while no one is standing on the device, are located beneath the driver's feet.

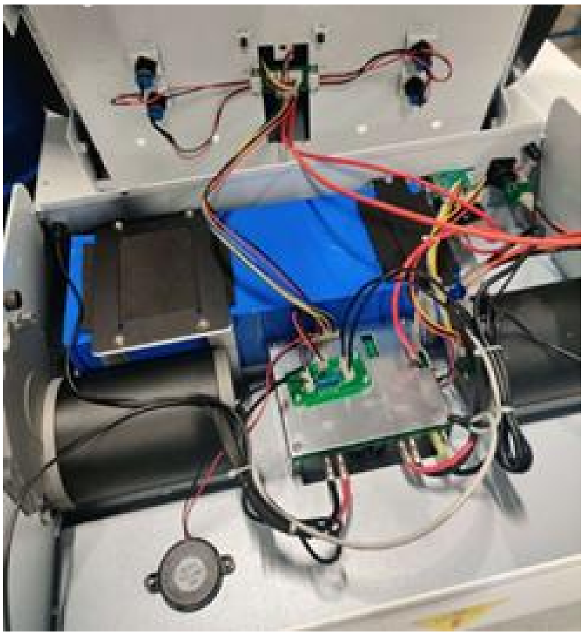

Figure 9: *Citybot's internal components.*

3.2. Planning Process

The main goals for this project were to achieve remote controllability and to ensure that the Citybot remained undamaged. In addition to IMU manipulation, methods such as installing a pendulum or a weight-shifting device on top of the segway, and reprogramming the firmware were considered. However, these methods had a high risk of damaging the device and were consequently discarded. Next best solution was to manipulate IMU data with a microcontroller.

3.2.1. Basic Concept and Control System Design

The core idea was to integrate a microcontroller between the Citybot's Atmega32 and MPU-6050 to be able to forward and modify accelerometer data sent by the MPU-6050. Since in Atmega32's control system, the setpoint for the Citybot's tilt angle is fixed at 0 degrees, as it is hardcoded into the firmware, the system continuously attempts to maintain this angle by comparing MPU-6050's data to the setpoint and adjusting the motor output accordingly. For forward movement, the Citybot would typically need to tilt within the range of approximately -5 to 0 degrees.

After installing the microcontroller, during normal operation, it would receive the accelerometer data from the MPU-6050 and simply relay it to the Atmega32 upon request, functioning as a transparent intermediary. Forward movement would be accomplished by coding the microcontroller to send modified accelerometer data, simulating a tilt that never actually happens. Atmega32 would respond by activating the motors, causing forward movement.

This approach presented a challenge: when the Citybot was initially stationary and balanced, activating the motors without an actual tilt would cause the device to lose stability and begin falling in the other direction. Another control system would need to be developed that, for forward movement, would keep the real tilt angle between –5 to 0 degrees by feeding controlled accelerometer data to the Atmega32's control system. During the planning phase, it was uncertain whether this control system could be successfully implemented, so it was determined that testing would be necessary.

3.2.2. Hardware Planning

After opening and studying the Citybot, the first step was to choose the microcontroller. The three main options were Arduino nano, Raspberry Pi and the ESP32. Both the Nano and ESP32 were more familiar than the Raspberry Pi, and the ESP32's built-in Wi-Fi and ROS support made it the most suitable choice. Additionally, a feature of the ESP32 that turned out to be highly useful later, was its dual I2C interface, allowing it to join the I2C bus as a controller and a target. [10].

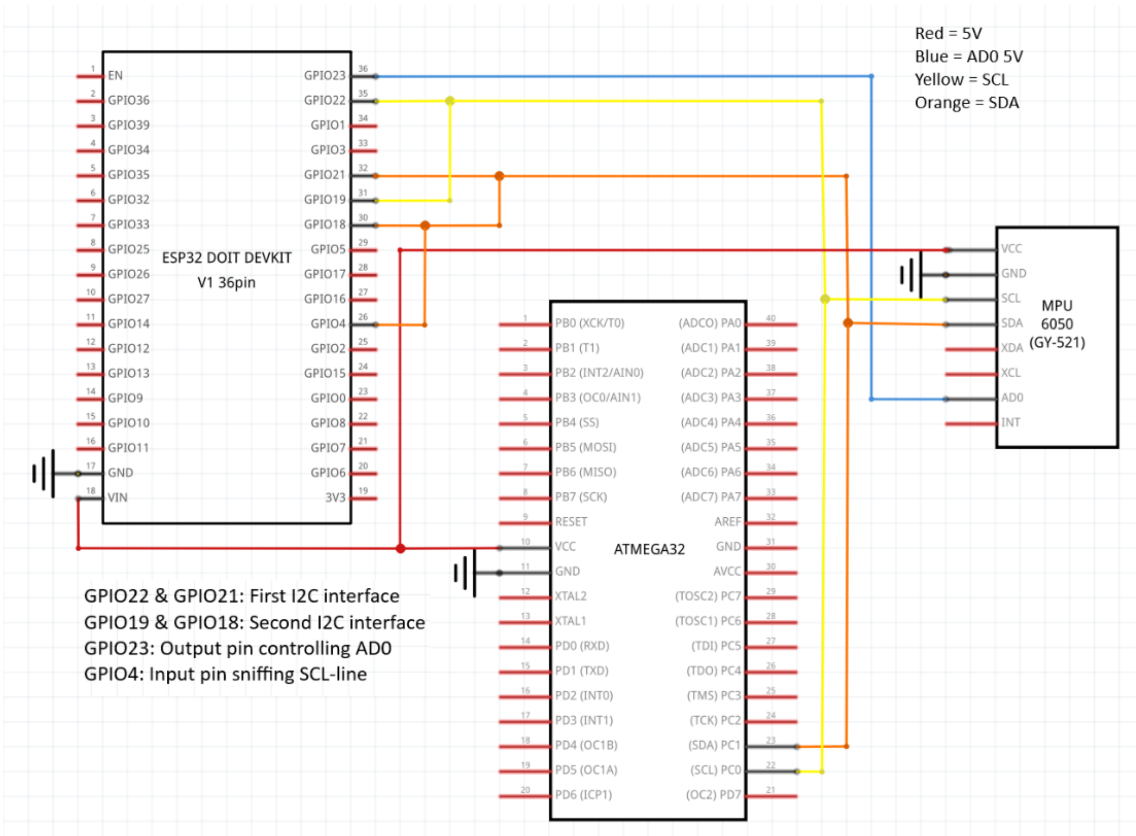

Figure 10: *Connection schematics. Drawn using Fritzing 1.0.4.*

The following step was to plan out the required connections. During testing, the ESP32 was powered by an USB cable from a laptop. The ESP32 requires 3.3 V to operate and the GY-521 was powered by 5V, so for a final version, a voltage regulator or an external power source would be required. Originally, the plan was to use two different I2C buses: one for the communication between ESP32 and MPU-6050, and the other for communication between ESP32 and Atmega32, but that would have required breaking the connection between Atmega32 and MPU-6050, which was against the idea of keeping the device as original as possible. Using a singular bus would create complications on the software side, since I2C communication overlaps cause system instability and crashes, and therefore, it would need to be prevented by code. The single

I2C bus and use of two different ESP32 interfaces can be seen in figure 10. As a result, the only hardware modification that was necessary was soldering a four-pin header into the MPU-6050, which alone didn't affect its normal operation.

3.2.3. Software Planning

From the software perspective, the primary goals, in order, were to first implement the logic for data forwarding, followed by developing the functionality for data manipulation, and lastly, integrating remote controllability using for example Wi-Fi and a joystick.

Since only one I2C bus was utilized, the forwarding logic had to be entirely implemented in software. Since Atmega32's firmware couldn't be modified, it would only ever request data from the address it was set to, which was the MPU-6050 address of 0x68. To implement communication between the two microcontrollers, the ESP32 had to be set up with that same address as well. However, if there are two devices with the same address on the I2C bus, and a controller device requests data from it, due to bus arbitration, only one of them would be able to respond. Fortunately, the GY-521 board included an AD0 pin, which changed the MPU-6050's address from 0x68 to 0x69 when 3-5 V was applied to it through ESP32's digital output pin, see figure 10. The ESP32 could now change MPU-6050's address and join the I2C bus using MPU-6050's old address. As a result, the Atmega32 continued to communicate as if it were interacting with the MPU-6050 at 0x68, when it was receiving data from the ESP32. Meanwhile, the ESP32 could be configured to request data from the MPU-6050 at 0x69, making seamless interception and modification of the data possible.

4. PROGRESS, CHALLENGES, AND SOLUTIONS

This chapter offers an in-depth overview of the entire project, including the challenges encountered, results achieved and ideas for further development. It begins by introducing the testing setup used to simulate the Citybot and is followed by an overview of the challenges encountered on both the software and the hardware sides. The chapter ends with an analysis of the project results and explores ideas for future development.

4.1. Test Setup

Since the Citybot was stored in the university premises, a setup simulating its behavior was implemented, enabling code testing from home. Initially, the setup consisted of three different ESP32s on a breadboard, one emulating the MPU-6050, another one acting as the Atmega32 and the third one serving as the actual ESP32 used in the project. Shortly after the ESP32 acting as an MPU-6050 was replaced with an actual MPU-6050 simplifying the setup, see figure 11 and 12. As there was no access to Atmega32's source code, the simulation code was an educated assumption, but it proved to be highly effective and valuable for testing.

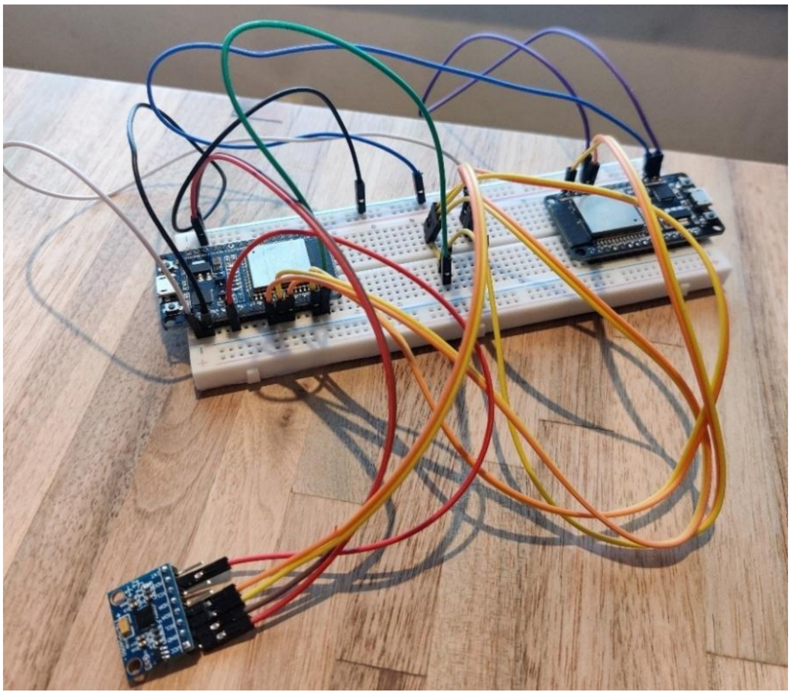

Figure 11: *Test setup used during software development.*

The setup was used with external pull-up resistor values of 2 kΩ, 4.7 kΩ and 10 kΩ, but none had any significant effect on the process as the ESP32's internal pull-up resistors

(45 kΩ) were strong enough. As a result, external pull-up resistors were left out of the final test setup. Longer connection wires were used to test the limits of I2C bus capacitance, but they did not introduce any issues in the test setup.

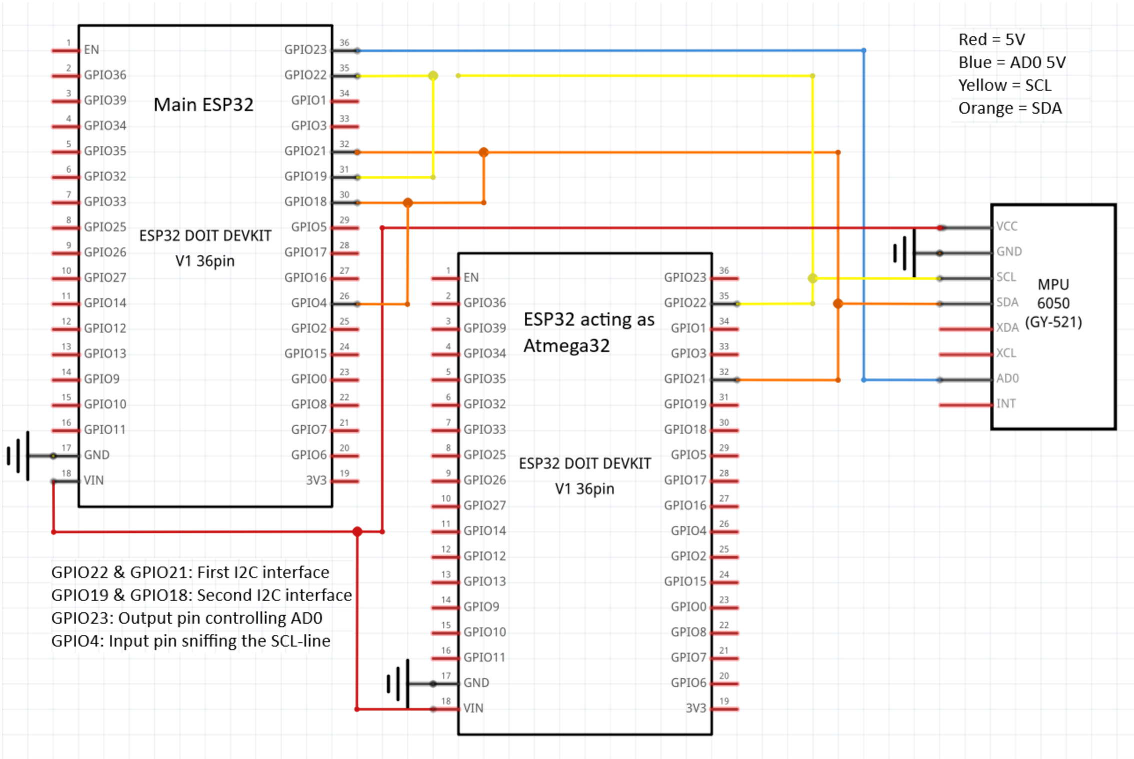

Figure 12: *Test setup schematics. Drawn using Fritzing 1.0.4.*

4.2. Software Implementation and Challenges

The editor used to write, compile and upload the code for the ESP32 was Arduino IDE version 2.3.4. A logic analyzer made by AZ-Delivery based on the Salea architecture capable of recording serial communication, was used to aid in debugging the system. PulseView by Sigrok was used to visualize and analyze the signals recorded by the logic analyzer. The Wire library was utilized to facilitate communication between the I2C devices, simplifying the complex I2C protocol into straightforward lines of code for sending and requesting data.

The first major challenge was implementing logic to correctly time the I2C communication. Many different methods were tried without success, until a working solution was finally found. A logic analyzer was used to check the I2C bus frequency, which was set to 222 kHz by Atmega32. From this information, the duration of the HIGH phase of the clock pulse could be calculated using formula 4.

$$T = \frac{1}{2 \cdot f} = \frac{1}{2 \cdot 222000\ Hz} = 2.25\ \mu s. \quad (4)$$

The duration was also validated using the logic analyzer, see figure 13. This confirmed that during data transfer, the SCL line remained HIGH for a maximum of 2.25 microseconds at a time. Additionally, it was possible to check the time between every data request, which was measured to be around 10 milliseconds, as seen in figure 14. It is important to note that when an I2C line is free, the SCL line remains HIGH until a controller device takes control of it again. This implies that in Citybot's case, if the SCL has been high for more than 2.25 microseconds, it is free and will be for the next 10 milliseconds. This idle period is vital for timing the communication, as the ESP32 should request new data from the MPU-6050 only during this period.

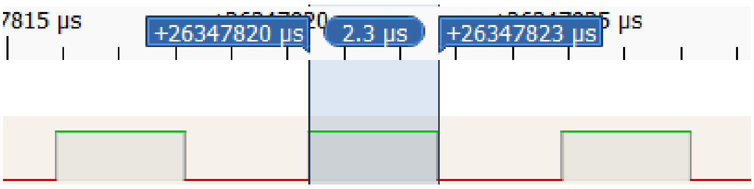

Figure 13: *Duration of SCL HIGH during a clock pulse. Captured using PulseView and an AZ-Delivery logic analyzer.*

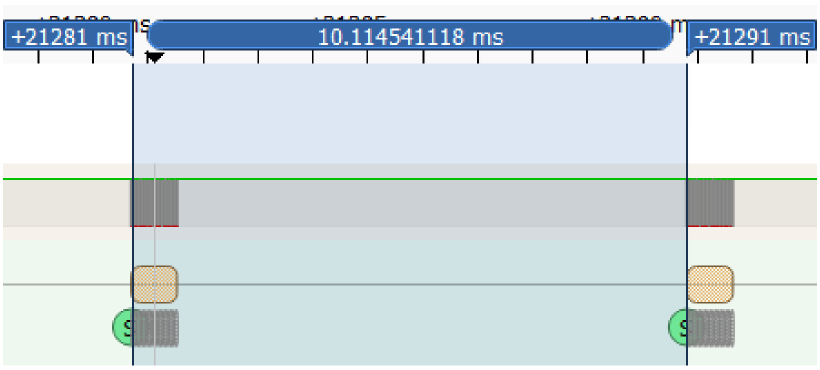

Figure 14: *Time between Atmega32's data requests. Captured using PulseView and an AZ-Delivery logic analyzer.*

By connecting one of ESP32's digital input pins into the SCL line (pin 04), it became possible for the ESP32 to track when the SCL line was released HIGH, calculating the idle time between every data request, see figures 10 and 15.

```
void fetchTimer() {
  static uint32_t scl_start_time = 0;   // Time when SCL is pulled HIGH
  static uint32_t scl_end_time = 0;     // Time when SCL is pulled LOW
  static uint32_t idle_start_time = 0; // Start of idle period
  static uint32_t idle_end_time = 0;  // End of idle period
  static uint32_t idle_duration = 0; // Duration of idle period
  static uint32_t max_idle_duration = 100; // Time between Atmega32's and ESP32's requests

  static bool scl_high_detected = false;     // Tracks SCL HIGH state
  static bool idle_detected_once = false;  // Ensures idle_begun flag is triggered once per idle period

 if (digitalRead(SCL_PIN) == 1) {  // 1 = HIGH
    if (!scl_high_detected) {
      scl_start_time = micros();
      scl_high_detected = true;
      idle_detected_once = false;
    }

    if (!idle_detected_once && (micros() - scl_start_time) > max_idle_duration) {
      Serial.println("Idle detected");
      idle_start_time = scl_start_time;  // Sets the correct idle start time

      idle_bus = true; // Flag for address change and data requests
      idle_detected_once = true; // Prevents multiple triggers
    }

} else if (digitalRead(SCL_PIN) == 0 && scl_high_detected) {  // 0 = LOW
    scl_end_time = micros();
    scl_high_detected = false;

    if (idle_detected_once) {
      idle_end_time = micros();
      idle_duration = (idle_end_time - idle_start_time) * 1.0002 // Applies small correction factor measured from logic analyzer

      Serial.print("Idle Duration: ");
      Serial.println(idle_duration);
    }
  }
}
```

Figure 15: *Timing logic coded in Arduino IDE.*

According to InvenSense, before data can be requested from the MPU-6050, the sensor must first be initialized. This process involves waking up the sensor and configuring various parameters and ranges by sending specific values to different registers [13]. The timing logic also proved to be a great solution for this. When the Citybot was powered on, Atmega32 first initialized the MPU-6050 and then started requesting data from it. Because changing the address doesn't reset the MPU-6050's registers, the ESP32 didn't need to initialize it again.

As shown in figure 16, when Atmega32 had finished initializing and requesting the first batch of data, ESP32 detected the idle time, and before 10 milliseconds had passed it changed the MPU-6050's address from 0x68 to 0x69. After that the ESP32 joined the I2C bus with the address 0x68 and requested new data from MPU-6050. Lastly, ESP32 prepared to send the newly acquired data to the Atmega32 on the next data request. These data requests cycle until the Citybot is powered off.

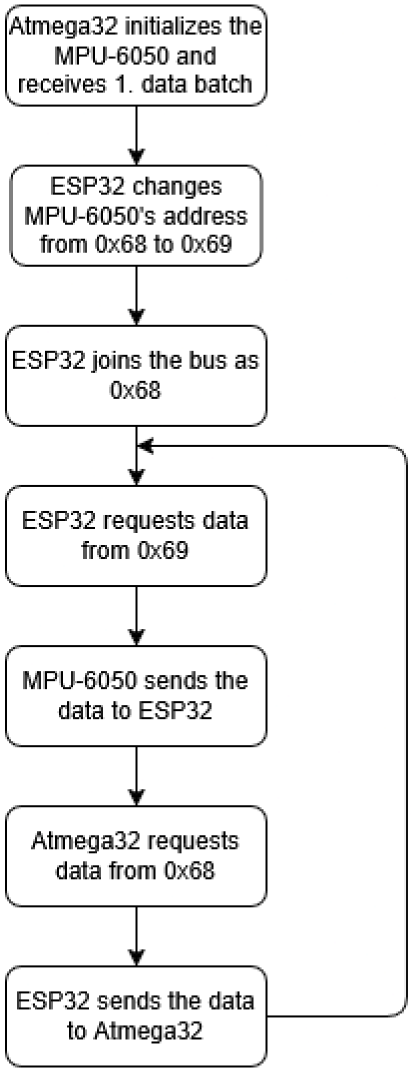

Figure 16: *Data transfer sequence as a block diagram.*

As shown in figures 18 and 19, The first batch of bytes is Atmega32 requesting data from the ESP32, followed by ESP32 requesting new data from the MPU-6050. This iteration introduced a delay of 10 ms between the MPU-6050 reading the accelerometer data and Atmega32 receiving it, which could have been reduced to around 2 ms by timing the ESP32's request to be closer to the next Atmega32 request, if deemed necessary.

Since the data transfer took around 650 microseconds, this method relied on the fact that there was at least that much time between each Atmega32 data request. Fortunately, the Citybot was configured to request new data every 10 milliseconds, providing more than enough time for the ESP32. Normal Citybot's I2C communication can be seen in figure 17. Ultimately, the method succeeded in forwarding and modifying the data acquired and sent by the MPU-6050.

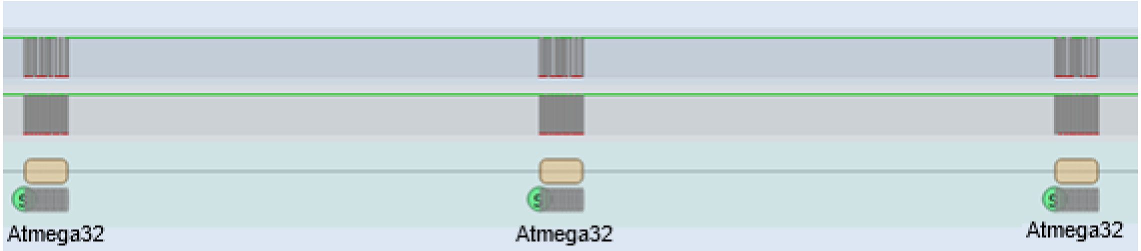

Figure 17: *Normal I2C communication within the Citybot with no modifications done (motors turned off). Captured using PulseView and an AZ-Delivery logic analyzer.*

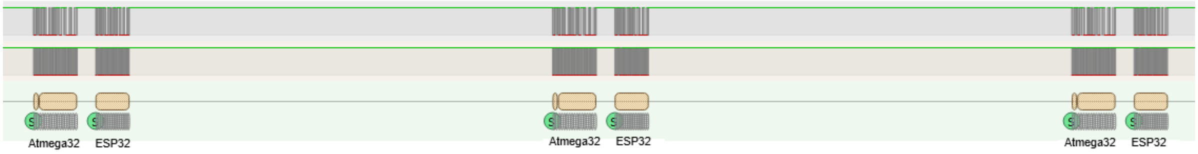

Figure 18: *I2C communication within Citybot while ESP32 is plugged in (motors turned off). Captured using PulseView and an AZ-Delivery logic analyzer.*

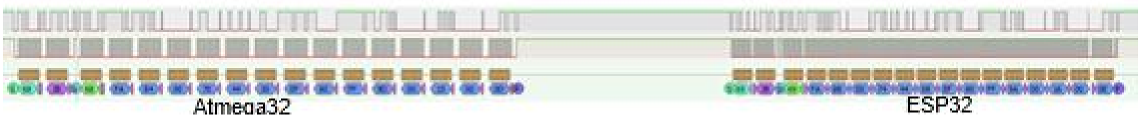

Figure 19: *Single Atmega32 and ESP32 data requests (motors turned off). Captured using PulseView and an AZ-Delivery logic analyzer.*

4.3. Hardware Challenges

The most significant issue in the entire project was caused by hardware. When a logic analyzer or the ESP32 was connected to the Citybot, everything functioned correctly until the safety switch was pressed, at which point the motors began drawing current and started spinning. After just a split second, the motors stopped spinning and the whole I2C data bus crashed. Sometimes waiting for a few seconds was enough to retry and sometimes the device had to be restarted.

First, tests were conducted to determine whether the crashes were caused by incorrectly sized pull-up resistors. The original setup only utilized the internal pull-up resistors of the ESP32 (45 kΩ) and the GY-521 (2.2 kΩ). Tests with external pull-up resistors of 10 kΩ, 2.67 kΩ and 1.7 kΩ connecting the I2C lines to 3.3 V were performed. However, these modifications did not have any significant impact on the crashing issue.

After more troubleshooting, it was discovered that even connecting a 12 cm long single-core copper cable into either SDA- or SCL line caused the I2C bus to crash when motors drew current. To investigate further, a test was conducted where the SDA line was briefly connected with a 12 cm long piece of tin-based solder and the safety switch was pressed. Every time the system crashed, one centimeter was cut from the piece, and it was connected into the SDA. This repeated until the system didn't crash anymore, which was at 3 cm.

The total I2C bus capacitance is determined by the combined capacitance of wire, connections and pins. The I2C fast mode standard is rated for a maximum capacitance of 400 pF. [4] The solder wire acts as a parasitic capacitor, increasing the bus capacitance. However, Addabbo et. al. estimate that a basic transmission line typically introduces 80 pF/m of parasitic capacitance [15]. Based on this, a 4 cm long solder wire should increase the bus capacitance by a mere 8.2 pF. Additionally, according to NXP's specifications a single I/O pin increases the bus capacitance by a maximum of 10 pF [4]. Given these values, Citybot's system remains well below the 400 pF limit, strongly implying that excessive capacitance is not the cause for the sudden crashes.

A theory was formed suggesting that when the I2C line is extended by 3-4 centimeters, either the lines, the soldered four-pin header, or the MPU-6050 itself began picking up electromagnetic interference (EMI) from the two 36 V motors drawing current. The motors were positioned on both sides of the MPU-6050 about 8 cm away from it, as shown in figure 9. This was furthermore validated by analyzing the logic analyzer recording when the switch was pressed, shown in figure 20.

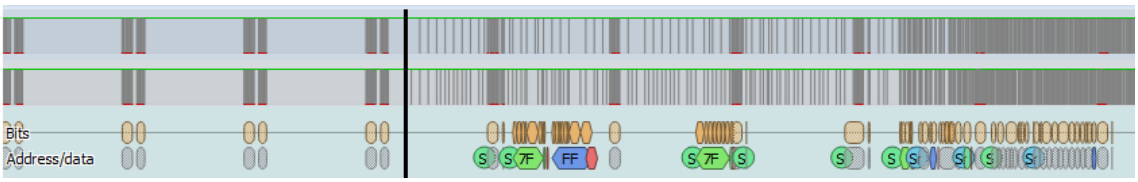

Figure 20: *Recording of the system crash. Captured using PulseView and an AZ-Delivery logic analyzer.*

Communication functioned properly until the safety switch was pressed, vaguely represented by the black line. After this point, the I2C lines were pulled LOW at random for a few tens of microseconds, with frequency of occurrences increasing over time. The actual data transfers were visible among the noise until a certain point, where the bus was overwhelmed by the noise, rendering communication impossible. A method to test if the noise was picked up by the MPU-6050 would be to disconnect it from the I2C bus or to completely remove it. However, time was running out and this would have been against the idea of keeping the vehicle as original as possible.

Despite the significant findings made from troubleshooting the I2C communication issues, time constraints and the complexity of mitigating the possible electromagnetic interference and ensuring operation under the given limitations, unfortunately led to the decision to end the project at this stage.

4.4. Results

The project successfully implemented an intermediary microcontroller that could intercept and modify data between the MPU-6050 and the Atmega32A-AU. The ESP32 was able to join the I2C bus, take control of the MPU-6050 and time the data communication flawlessly.

However, hardware-related challenges ultimately prevented full system functionality. When Citybot’s motors began drawing power, EMI or other environmental factors affected the I2C communication, causing bus instability. Even small extensions of the I2C wiring significantly increased vulnerability to interference, leading to sudden system failures. Despite efforts to solve this issue by testing different pull-up resistor values and alternative wiring setups, no solution was found within the given timeframe

Although the data forwarding and modification process was successfully implemented, the actual control system required for remote operation was never developed. While the ability to manipulate tilt data was demonstrated, it is unknown whether this approach could reliably control Citybot’s movement without causing instability or unintended behavior. Further testing and refinement would be necessary to determine the actual feasibility of the method.

4.5. Possible Solutions and Further Development

To better understand the root cause of the I2C instability, further testing is needed. This could involve isolating variables in a controlled setup to determine whether EMI is the primary culprit or if alternative factors, such as power fluctuations are responsible.

To combat potential EMI, several mitigation strategies may be considered. Implementing shielding around sensitive components such as the MPU-6050, along with the use of shielded cabling, could significantly reduce vulnerability to external interference. In addition, integrating high-frequency noise filters, such as ferrite beads or transient-voltage-suppression diodes could help suppress unwanted signal disturbances.

An alternative method involves replacing the MPU-6050 with a better inertial measurement unit. However, this would require modifying the output of the new IMU to match the MPU-6050’s data format.

The most important features for future software development are the implementation of a feedback control system for manipulating the accelerometer data and a method to remotely control the control system, such as through Wi-Fi and a joystick.

5. CONCLUSIONS

The purpose of this thesis was to investigate the feasibility of achieving remote control of a two-wheeled self-balancing vehicle through the manipulation of accelerometer data. This approach was selected because, in theory, it could be implemented without making any permanent modifications to the firmware or the device itself.

An external microcontroller was integrated successfully into the vehicle, and it was able to take control of the accelerometer and time the communication without causing any issues on the I2C bus. Due to hardware implications and time constraints, the main goal was not achieved, but the project served as a great learning experience in both I2C and manipulating a functioning system through the integration of an external device. It also provided key insight into the critical factors that need to be taken into consideration when attempting a method of this nature.

If given more time, the future developments would focus on resolving the I2C's instability issues, implementing a feedback control system for accelerometer data manipulation and adding remote controllability, such as through Wi-Fi and a joystick. A fully functioning, remote-controllable two-wheeled self-balancing vehicle could be equipped with cameras and a longer-range data transfer method. It could then be used for a variety of tasks, such as surveillance and remote inspections. Integrating LiDAR or visual-based sensors would make autonomous navigation possible, unlocking more applications such as autonomous mapping, search and rescue operations, and infrastructure inspections.

REFERENCES

[1] R.P.M. Chan, K.A. Stol, C.R. Halkyard, Review of modelling and control of two-wheeled robots, Annual reviews in control, 2013, pp. 89–103.

[2] Mr. Segway's difficult path, The Economist (London), London: Economist Intelligence Unit N.A. Incorporated, vol. 395, 2010.

[3] G. Ciaburro, Dynamic Modeling of a Segway as an Inverted Pendulum System, Keras Reinforcement Learning Projects, United Kingdom: Packt Publishing, Limited, 2018.

[4] NXP Semiconductors, "I2C-bus specification and user manual," rev 7.0 - 1. October 2021, Available from (cited 2025 Feb 27): https://www.nxp.com/docs/en/user-guide/UM10204.pdf

[5] V.K. Pandey, S. Kumar, V. Kumar, P. Goel, A review paper on I2C communication protocol. Int J Adv Res Ideas Innov Technol, April 2018.

[6] Mayayu, I2C ACK [Internet], San Francisco: Wikimedia Foundation, 2006, Available from (cited 2025 Mar 30):

https://commons.wikimedia.org/wiki/File:I2C_ACK.svg

[7] Tim Mathias, I2C controller-target [Internet], San Francisco: Wikimedia Foundation, 2006, Available from (cited 2025 Mar 30):

https://commons.wikimedia.org/wiki/File:I2C_controller-target.svg

[8] D.A. Alciatore, M.B. Histand, Introduction to Mechatronics and Measurement Systems, New York, McGraw-Hill, 4th ed, 2012, pp. 258-265.

[9] F. J. Mercede, Hands-on projects to introduce Electrical and Computer Engineering, IEEE Frontiers in Education Conference, 2008.

[10] Espressif Systems, ESP32 Technical Reference Manual v5.3 [Internet], January 2025, Available from (cited 2025 Feb 27):

https://www.espressif.com/en/support/documents/technical-documents

[11] S. Hodas, J. Izvoltova, D. Rekus, Trends in Inertial Navigation Technologies, IOP conference series Earth and environmental science, 2021

[12] K. Tuck, Tilt sensing using linear accelerometers, Accelerometer Systems and Applications Engineering, Tempe, AZ.

[13] InvenSense Inc., MPU-6000 and MPU-6050 Product Specification Revision 3.4, August 2013.

[14] Proto Supplies, MPU-6050 Module with GY-521 Breakout Board – Schematic Diagram [Internet], Available from (cited 2025 Feb 27):

https://protosupplies.com/product/mpu-6050-gy-521-3-axis-accel-gryo-sensor-module/

[15] T. Addabbo, A. Fort, M. Mugnaini, S. Parrino, A. Pozzebon, V. Vignoli, Using the I2C bus to set up Long Range Wired Sensor and Actuator Networks in Smart Buildings, In: 2019 4th International Conference on Computing, Communications and Security (ICCCS), IEEE; 2019