GETTING STARTED WITH RASPBERRY PI BOARD AND IMU SENSOR INTERFACING

CSCI502/702 – HARDWARE/SOFTWARE CO-DESIGN

Project No. 01

SUBMITTED BY

AZAT BALAPAN

BINA BATOOL

MINH HIEU LE

SUBMITTED TO

DR. ALMAS SHINTEMIROV

NAZARBAYEV UNIVERSITY

SCHOOL OF ENGINEERING AND DIGITAL SCIENCE
NAZARBAYEV UNIVERSITY

DEPARTMENT OF ROBOTICS ENGINEERING

Part 1: Interfacing Raspberry Pi

Task 1: Getting Started with Raspberry Pi

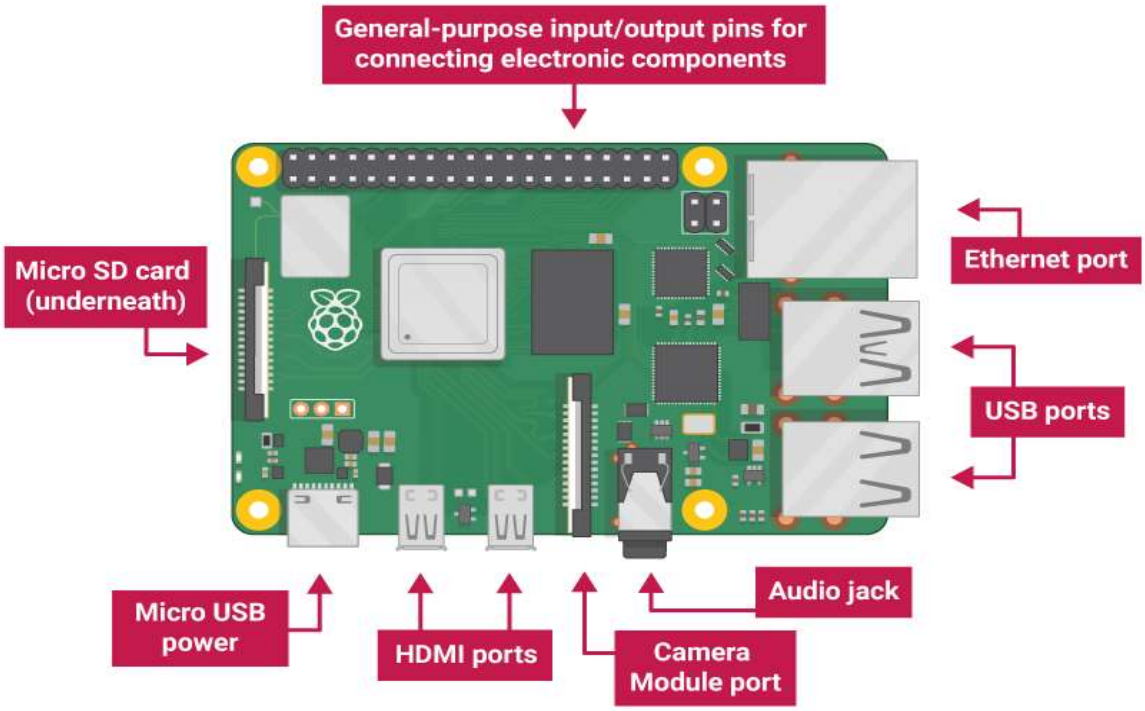

Fig.01. Raspberry Pi Hardware Overview

Quick Overview

The Raspberry Pi is a compact single-board computer that contains a processor, memory, networking interfaces, and general-purpose input/output (GPIO) pins on a single circuit board. It can be used to run a full operating system and interact with external sensors and devices.

Before starting the configuration, we identified the main hardware components of the Raspberry Pi board:

- **USB ports** were used to connect a keyboard and mouse.
- **SD card slot** was used to insert the microSD card containing the operating system.
- **Ethernet port** allows wired network connection.
- **Audio jack** can be used for headphones or speakers.
- **HDMI port** was used to connect the monitor.
- **Micro-USB power connector** supplies power to the board.
- **GPIO header** allows connection of sensors and electronic components.

Task 1.1: Raspberry Pi setup

In the first part of the project, we prepared the Raspberry Pi board and configured the operating system so that it could communicate with external sensors.

We used the **Raspberry Pi Imager** software to write the **Raspberry Pi OS (32-bit)** image onto a microSD card. The image was downloaded and written to the card using a computer with a card reader.

Fig. 2. Raspberry Pi Imager interface for OS and SD card selection.

After writing the OS image, we inserted the microSD card into the Raspberry Pi board.

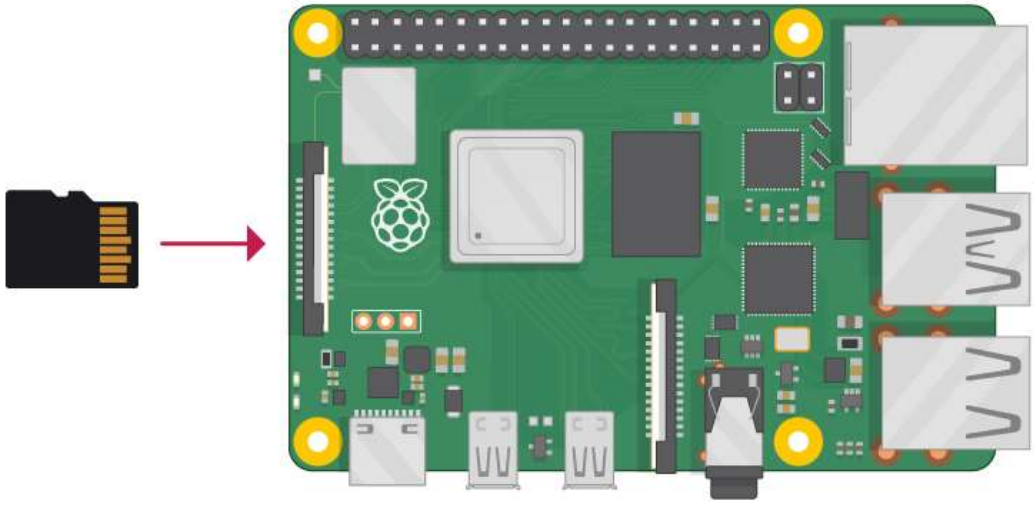

Fig. 3. Inserting the microSD card into the Raspberry Pi.

We then connected a monitor, keyboard, and mouse to the board as shown in fig.4., before powering it on, following the safety instructions provided in the course material.

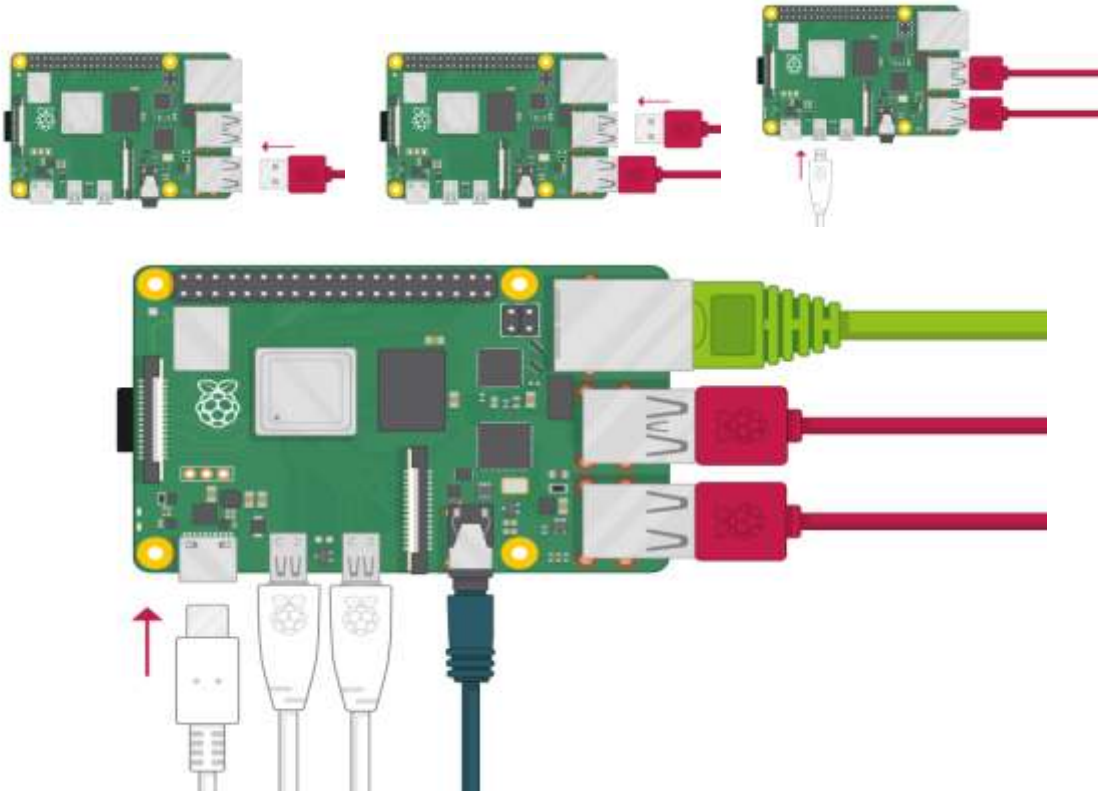

Fig. 4. Raspberry Pi with Connected Peripherals

After all peripherals were connected, the power supply was attached to the micro-USB power connector. The board booted successfully, and the Raspberry Pi desktop interface appeared on the monitor.

Task 1.2: Connecting to Wi-Fi Network

Once the desktop environment was loaded, we connected the Raspberry Pi to the laboratory Wi-Fi network.

This step was necessary so that:

- The operating system could be updated.
- Remote SSH connection could be established later.

After entering the Wi-Fi credentials, the board successfully connected to the network.

Fig. 5. Raspberry Pi desktop welcome screen after successful boot.

Task 1.3: System update

After confirming that the system was working, we opened the terminal and updated the operating system using the following commands:

```
sudo apt update
```

```
sudo apt upgrade
```

The first command updated the package lists, and the second command upgraded the installed software to the latest versions.

```
linuxuser@linux:~/Desktop$ sudo apt-get update
[sudo] password for linuxuser:
Get:1 http://security.ubuntu.com/ubuntu jammy-security InRelease [110 kB]
Hit:2 http://us.archive.ubuntu.com/ubuntu jammy InRelease
Get:3 http://us.archive.ubuntu.com/ubuntu jammy-updates InRelease [119 kB]
Get:4 http://security.ubuntu.com/ubuntu jammy-security/main amd64 Packages [728
kB]
0% [3 InRelease 102 kB/119 kB 86%] [4 Packages 432 kB/728 kB 59%]
```

Fig. 6. Terminal output showing system update using apt-get update

This step ensured that:

- All system packages were up to date.

- The latest drivers and libraries were installed.

Task 1.4: Enabling required interfaces

To communicate with external devices, certain interfaces needed to be enabled.

We opened the Raspberry Pi configuration tool using the command:

sudo raspi-config

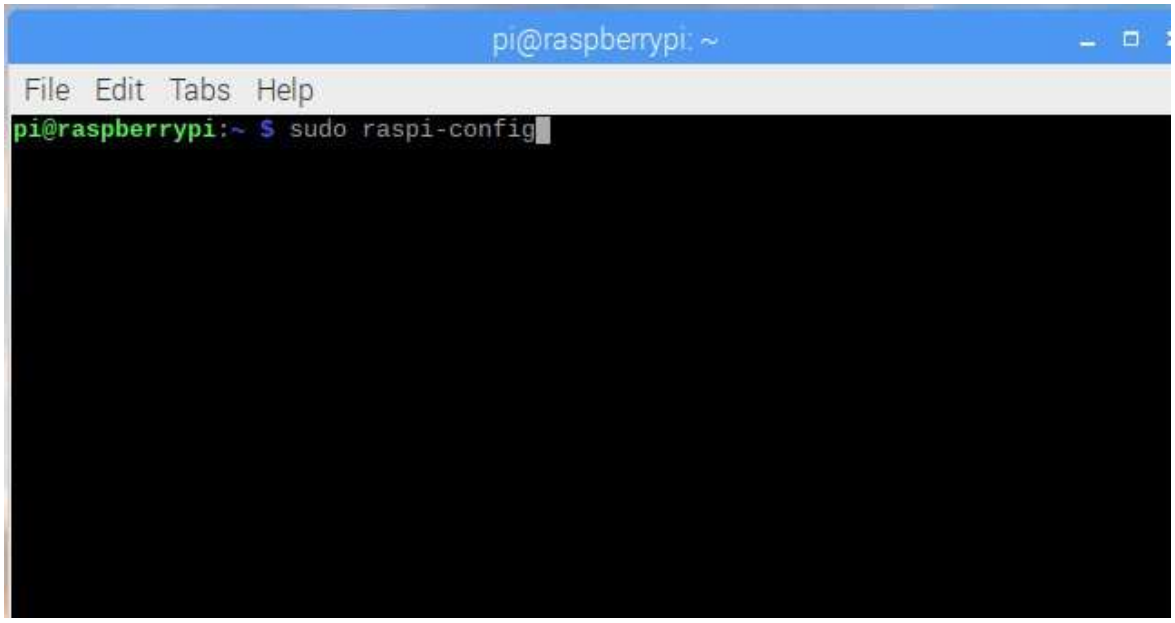

Fig. 7. Opening the raspi-config utility from the terminal.

The following interfaces were enabled:

- SSH
- I²C

Inside the configuration menu, we enabled the **SSH interface** so that we could connect to the board remotely, and we also enabled the **I²C interface**, which is required for communication with the IMU sensor. We also expanded the filesystem to use the full capacity of the microSD card.

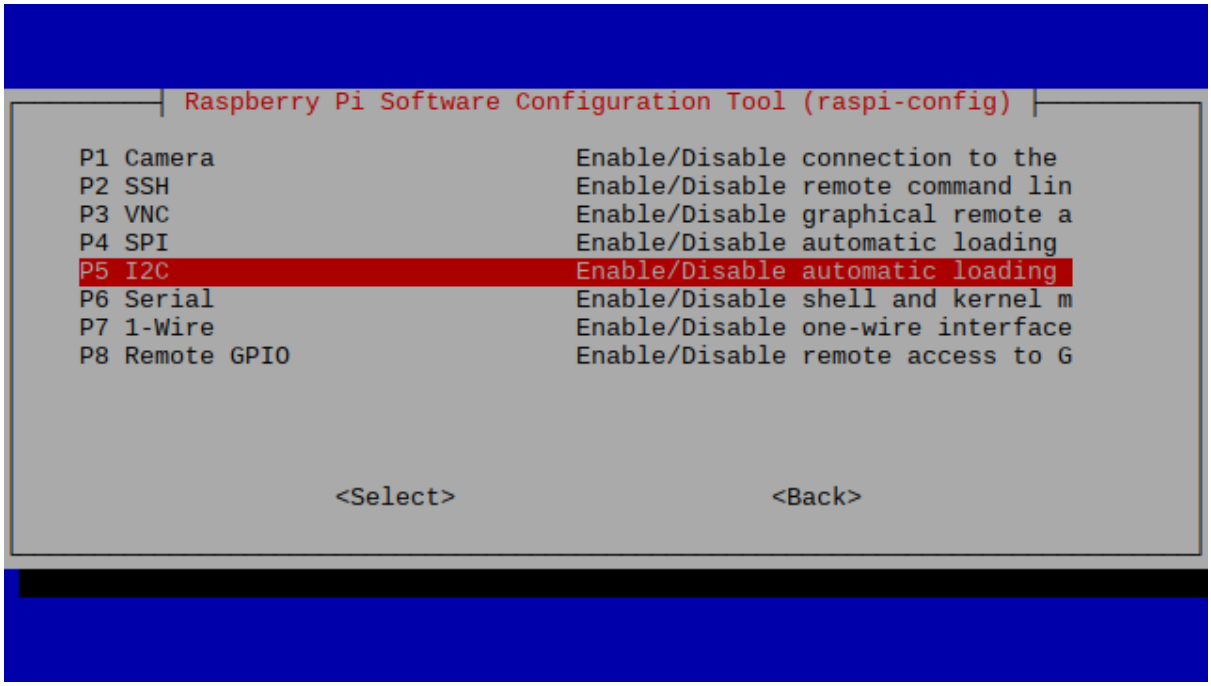

Fig. 8. Enabling the I²C interface in the raspi-config tool.

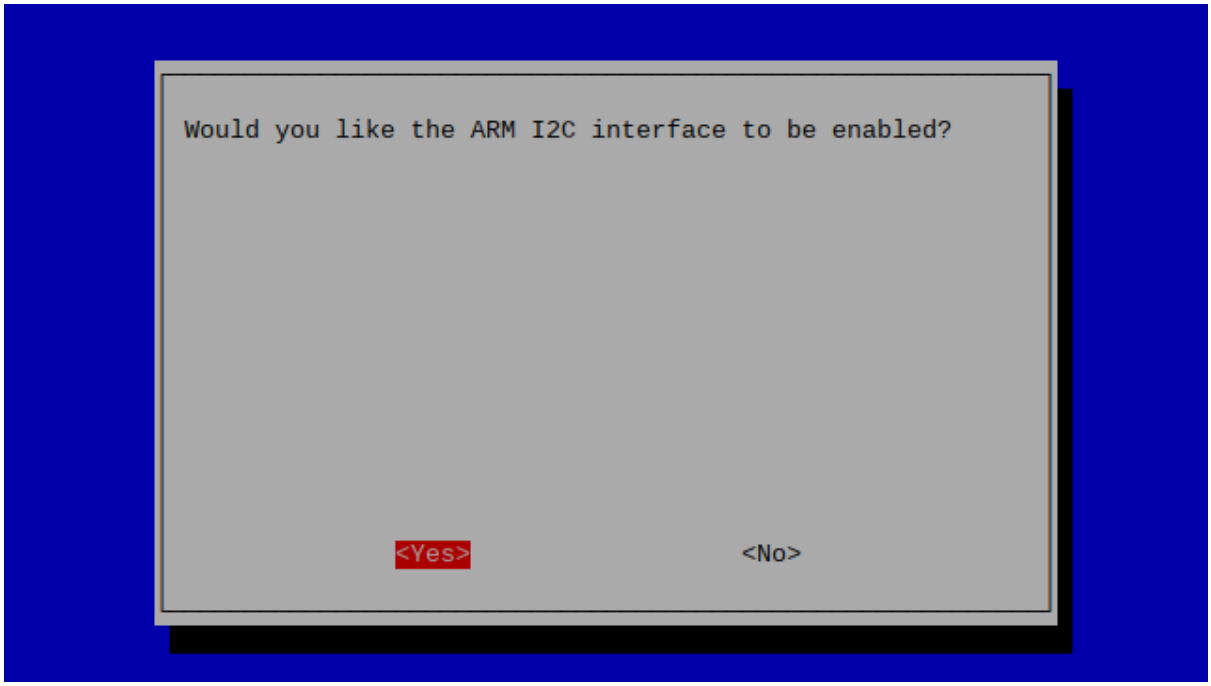

Fig. 9. Confirmation dialog for enabling the I²C interface.

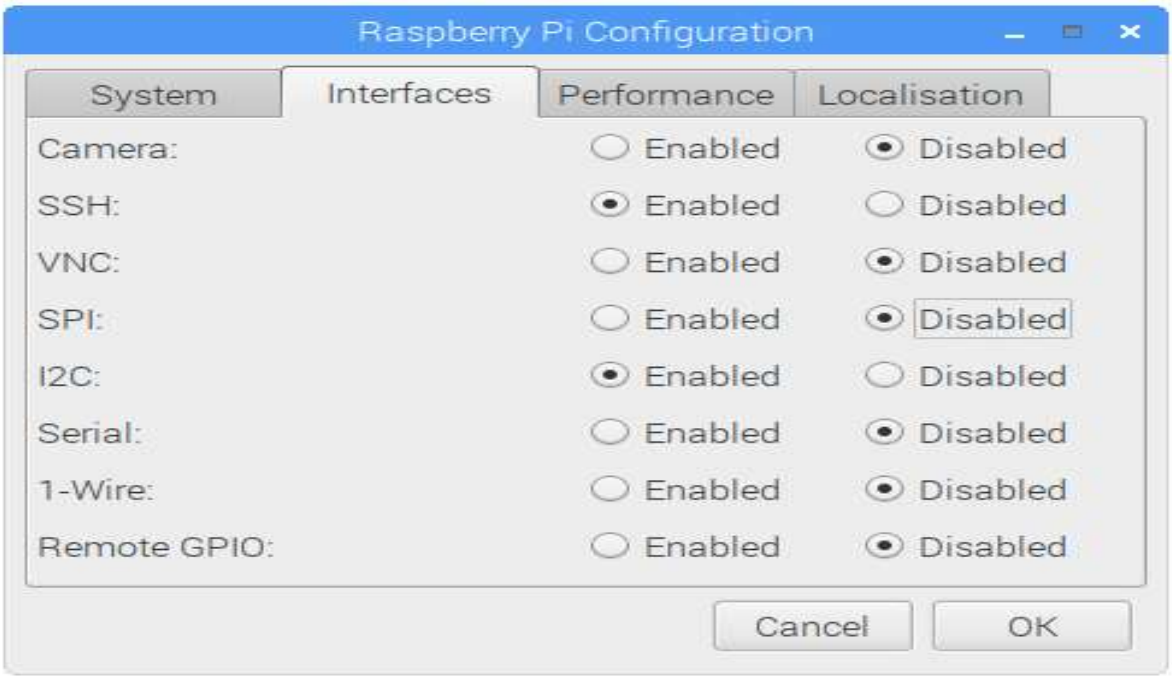

Fig. 10. Raspberry Pi configuration window showing enabled I²C and SSH interfaces.

After completing the configuration, we rebooted the system to apply the changes.

To verify the hardware setup, we observed the Raspberry Pi running with the connected peripherals and the monitor displaying the desktop interface.

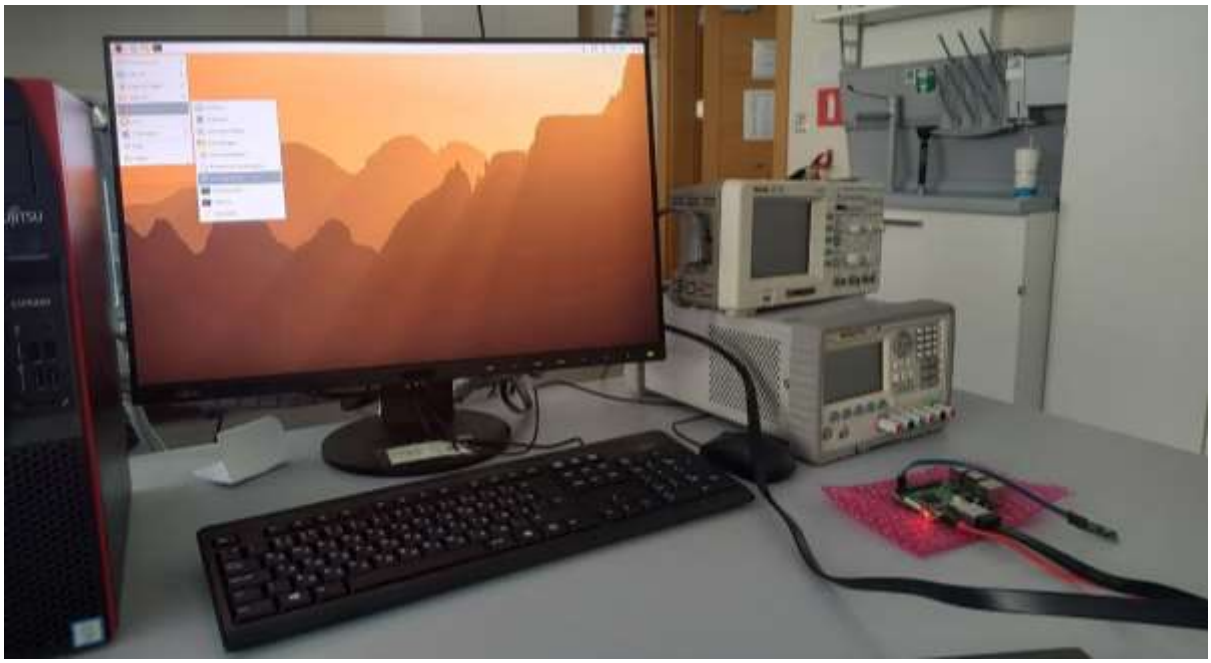

Fig. 11. Raspberry Pi connected to monitor, keyboard, and peripherals during system setup.

Task 2: Git Practice

Task 2.1: Creating a GitHub Repository

In the next part of the assignment, we practiced basic Git operations using GitHub.

We created a GitHub repository for our group and uploaded a simple C++ program.

The steps included:

1. Initializing the repository.
2. Adding files to the repository.
3. Committing the changes.
4. Pushing the repository to GitHub.

This exercise was done to ensure that we understood version control and could use GitHub Classroom for future assignments.

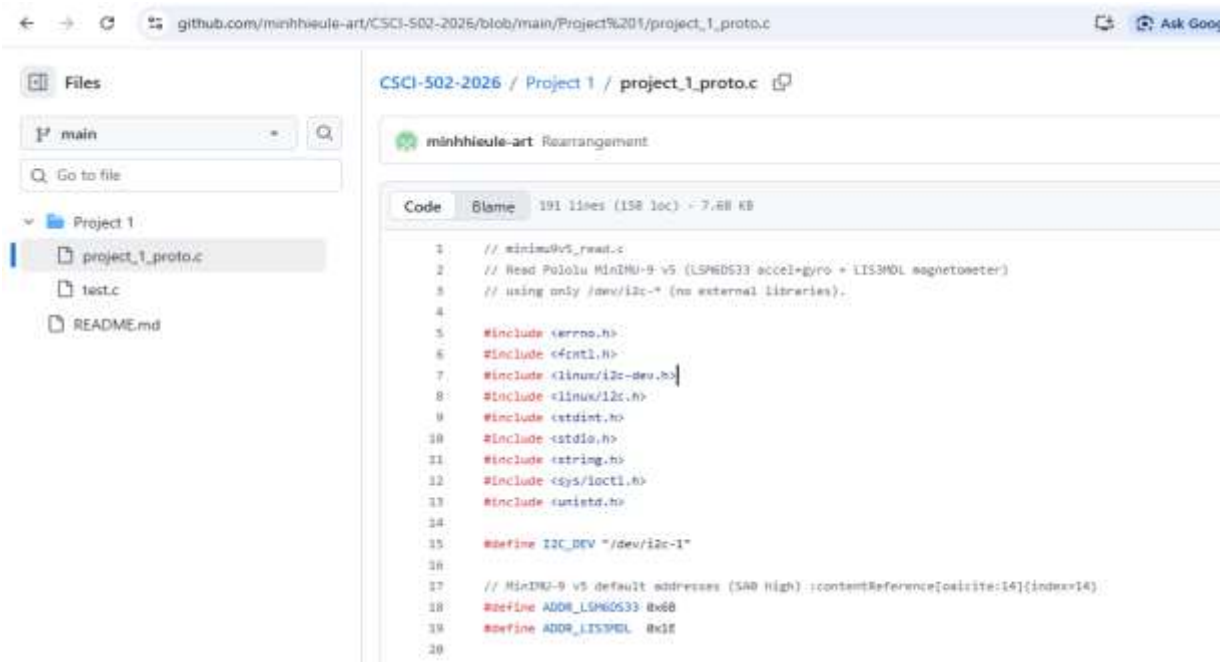

Fig. 12. GitHub repository showing the uploaded files

Part 2: IMU Sensor Interfacing

Task 3: IMU Sensor Setup

Task 3.1: IMU connection to Raspberry Pi

In the second part of the project, we connected the Pololu MinIMU-9 v5 sensor to the Raspberry Pi using the I²C interface.

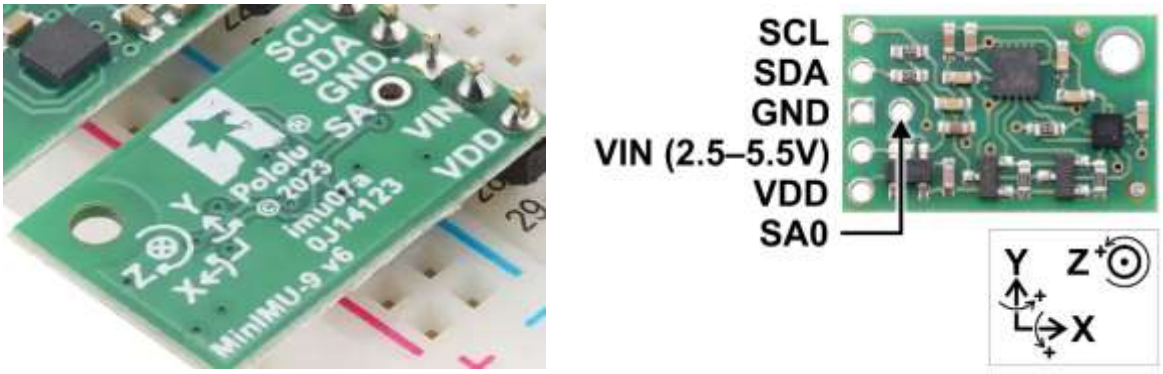

Fig. 13. Pololu MinIMU-9 v5 sensor and its I²C pin connections.

We connected the sensor pins to the Raspberry Pi GPIO header according to the instructions:

IMU Pin	Raspberry Pi Physical Pin
SCL	Pin 5
SDA	Pin 3
GND	Pin 6
VDD	Pin 1
VIN	Not connected

Explanation:
VIN is not connected because the Raspberry Pi already provides regulated 3.3 V power through the VDD pin. Connecting VIN could bypass voltage regulation and potentially damage the sensor.

Task 3.2: Detecting the IMU on the I²C bus

After connecting the sensor, we verified the I²C communication. We opened the terminal and used the following command:

i2cdetect -y 1

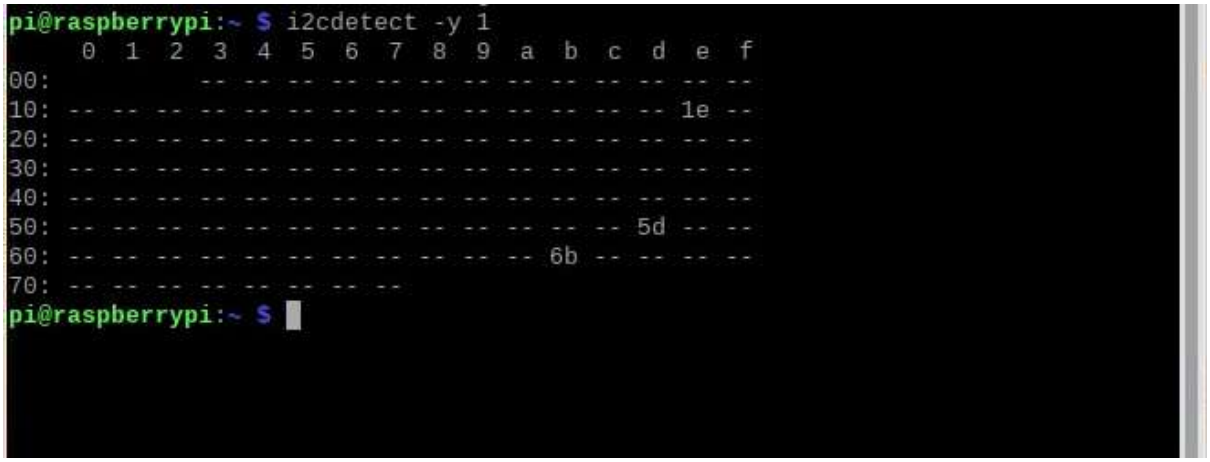

Fig. 15. Output of the `i2cdetect -y 1` command showing detected IMU devices on the I²C bus.

This command scans the I²C bus and displays connected device.

The following device addresses were detected:

- `0x6B` – Gyroscope and accelerometer (LSM6DS33)

These addresses match the expected values from the sensor datasheet, which confirmed that the IMU was correctly connected and detected by the Raspberry Pi.

Task 4: Sensor Activation and Code Implementation

Task 4.1: Observing the Default Sensor State

After detecting the IMU on the I²C bus, the next step was to verify whether the sensors were producing measurement data by default.

According to the assignment instructions, the gyroscope, accelerometer, and magnetometer are initially in a **power-down state**. This means they do not provide orientation or motion data until they are activated through their control registers.

To verify this behavior, we used the following commands in the terminal:

i2cget -y 1 0x6B 0x22

i2cget -y 1 0x6B 0x23

These registers correspond to the low and high bytes of the gyroscope X-axis data.

Why is value “1” used in the i2cget command?

The value **1** in the command represents the I²C bus number.
On the Raspberry Pi, the main I²C interface is available as:

/dev/i2c-1

Therefore, the number 1 is used in all I²C commands.

Initially, the values were not changing, which indicated that the sensors were still in their default power-down state.

Task 4.2: Activating the Sensors Using Control Registers

Since the sensors were not producing data, the next step was to activate them by writing control words to their configuration registers.

The assignment required the sensors to be configured with the following settings:

Sensor	Output Frequency	Measurement Range
Gyroscope	26 Hz	±500 dps
Accelerometer	26 Hz	±2 g
Magnetometer	5 Hz	±4 gauss

Instead of manually configuring the sensors using multiple terminal commands, we implemented a C program that performs all configuration steps automatically.

Task 4.3: Development of the Sensor Reading Program

A C program was developed to:

1. Open the I²C communication channel.
2. Verify the sensor identities.
3. Configure the sensors using control words.
4. Continuously read sensor measurements.
5. Convert raw data into physical units.
6. Display real-time output.

This approach ensured consistent configuration and easier data acquisition.

Code: minimu9v5_read.c

```
// minimu9v5_read.c

// Read Pololu MinIMU-9 v5 (LSM6DS33 accel+gyro + LIS3MDL magnetometer)

// using only /dev/i2c-* (no external libraries).



#include <errno.h>

#include <fcntl.h>
```

```
#include <linux/i2c-dev.h>
#include <linux/i2c.h>
#include <stdint.h>
#include <stdio.h>
#include <string.h>
#include <sys/ioctl.h>
#include <unistd.h>

#define I2C_DEV "/dev/i2c-1"
// MinIMU-9 v5 default addresses
#define ADDR_LSM6DS33 0x6B
#define ADDR_LIS3MDL  0x1E

// LSM6DS33 registers
#define LSM6_WHO_AM_I   0x0F
#define LSM6_CTRL1_XL   0x10
#define LSM6_CTRL2_G    0x11
#define LSM6_CTRL3_C    0x12
#define LSM6_OUTX_L_G   0x22
#define LSM6_OUTX_L_XL  0x28

// LIS3MDL registers
#define LIS_WHO_AM_I    0x0F
#define LIS_CTRL_REG1   0x20
#define LIS_CTRL_REG2   0x21
```

```
#define LIS_CTRL_REG3   0x22
#define LIS_CTRL_REG4   0x23
#define LIS_CTRL_REG5   0x24
#define LIS_OUT_X_L     0x28

// Write one register
static int i2c_write_reg(int fd, uint8_t addr7, uint8_t reg, uint8_t val)
{
    uint8_t buf[2] = { reg, val };
    struct i2c_msg msg = {
        .addr  = addr7,
        .flags = 0,
        .len   = sizeof(buf),
        .buf   = buf
    };
    struct i2c_rdwr_ioctl_data xfer = { .msgs = &msg, .nmsgs = 1 };

    if (ioctl(fd, I2C_RDWR, &xfer) < 0) return -1;
    return 0;
}

// Read multiple registers
static int i2c_read_regs(int fd, uint8_t addr7, uint8_t start_reg, uint8_t *out, uint16_t n)
{
    struct i2c_msg msgs[2];
```

```
    msgs[0].addr  = addr7;
    msgs[0].flags = 0;
    msgs[0].len   = 1;
    msgs[0].buf   = &start_reg;

    msgs[1].addr  = addr7;
    msgs[1].flags = I2C_M_RD;
    msgs[1].len   = n;
    msgs[1].buf   = out;
    struct i2c_rdwr_ioctl_data xfer = { .msgs = msgs, .nmsgs = 2 };
    if (ioctl(fd, I2C_RDWR, &xfer) < 0) return -1;
    return 0;
}
static int16_t le16_to_i16(uint8_t lo, uint8_t hi)
{
    return (int16_t)((uint16_t)lo | ((uint16_t)hi << 8)); }

static void die(const char *msg)
{
    fprintf(stderr, "%s: %s\n", msg, strerror(errno));}

int main(void)
{
    int fd = open(I2C_DEV, O_RDWR);
```

```
if (fd < 0) { die("open(/dev/i2c-1) failed"); return 1; }

// Identify LSM6DS33

uint8_t who = 0;

if (i2c_read_regs(fd, ADDR_LSM6DS33, LSM6_WHO_AM_I, &who, 1) < 0) {

    die("LSM6DS33 WHO_AM_I read failed"); close(fd); return 1;

}

printf("LSM6DS33 WHO_AM_I = 0x%02X (expected 0x69)\n", who);


// Configure LSM6DS33

// CTRL3_C: BDU=1, IF_INC=1

if (i2c_write_reg(fd, ADDR_LSM6DS33, LSM6_CTRL3_C, 0x44) < 0) {

    die("LSM6DS33 write CTRL3_C failed"); close(fd); return 1;

}


// CTRL1_XL: 26 Hz, ±2g

if (i2c_write_reg(fd, ADDR_LSM6DS33, LSM6_CTRL1_XL, 0x20) < 0) {

    die("LSM6DS33 write CTRL1_XL failed"); close(fd); return 1;

}


// CTRL2_G: 26 Hz, ±500 dps

if (i2c_write_reg(fd, ADDR_LSM6DS33, LSM6_CTRL2_G, 0x24) < 0) {

    die("LSM6DS33 write CTRL2_G failed"); close(fd); return 1;

}

// Identify LIS3MDL

uint8_t who_m = 0;
```

```
if (i2c_read_regs(fd, ADDR_LIS3MDL, LIS_WHO_AM_I, &who_m, 1) < 0) {
    die("LIS3MDL WHO_AM_I read failed"); close(fd); return 1;
}
printf("LIS3MDL WHO_AM_I = 0x%02X (expected 0x3D)\n", who_m);


//  Configure LIS3MDL
// CTRL_REG1: 5 Hz
if (i2c_write_reg(fd, ADDR_LIS3MDL, LIS_CTRL_REG1, 0x60) < 0) {
    die("LIS3MDL write CTRL_REG1 failed"); close(fd); return 1;
}
// CTRL_REG2: ±4 gauss
if (i2c_write_reg(fd, ADDR_LIS3MDL, LIS_CTRL_REG2, 0x00) < 0) {
    die("LIS3MDL write CTRL_REG2 failed"); close(fd); return 1;
}
// CTRL_REG3: continuous mode
if (i2c_write_reg(fd, ADDR_LIS3MDL, LIS_CTRL_REG3, 0x00) < 0) {
    die("LIS3MDL write CTRL_REG3 failed"); close(fd); return 1;
}
// CTRL_REG4: high performance Z
if (i2c_write_reg(fd, ADDR_LIS3MDL, LIS_CTRL_REG4, 0x0C) < 0) {
    die("LIS3MDL write CTRL_REG4 failed"); close(fd); return 1;
}
// CTRL_REG5: BDU enabled
if (i2c_write_reg(fd, ADDR_LIS3MDL, LIS_CTRL_REG5, 0x40) < 0) {
    die("LIS3MDL write CTRL_REG5 failed"); close(fd); return 1;
```

```
    }

    // Scale factors

    const double acc_mps2_per_lsb = 0.000061 * 9.80665; // ±2g

    const double gyr_dps_per_lsb  = 0.01750;            // ±500 dps

    const double mag_gauss_per_lsb = 1.0 / 6842.0;      // ±4 gauss



    // Read loop

    while (1) {

        uint8_t gbuf[6], abuf[6];

        if (i2c_read_regs(fd, ADDR_LSM6DS33, LSM6_OUTX_L_G, gbuf, 6) < 0) {
die("Read gyro failed"); break; }

        if (i2c_read_regs(fd, ADDR_LSM6DS33, LSM6_OUTX_L_XL, abuf, 6) < 0) {
die("Read accel failed"); break; }



        int16_t gx = le16_to_i16(gbuf[0], gbuf[1]);

        int16_t gy = le16_to_i16(gbuf[2], gbuf[3]);

        int16_t gz = le16_to_i16(gbuf[4], gbuf[5]);



        int16_t ax = le16_to_i16(abuf[0], abuf[1]);

        int16_t ay = le16_to_i16(abuf[2], abuf[3]);

        int16_t az = le16_to_i16(abuf[4], abuf[5]);



        uint8_t mbuf[6];

        uint8_t start = (uint8_t)(LIS_OUT_X_L | 0x80);

        if (i2c_read_regs(fd, ADDR_LIS3MDL, start, mbuf, 6) < 0) { die("Read mag failed");
break; }

        int16_t mx = le16_to_i16(mbuf[0], mbuf[1]);
```

```
    int16_t my = le16_to_i16(mbuf[2], mbuf[3]);
    int16_t mz = le16_to_i16(mbuf[4], mbuf[5]);

    double ax_mps2 = ax * acc_mps2_per_lsb;
    double ay_mps2 = ay * acc_mps2_per_lsb;
    double az_mps2 = az * acc_mps2_per_lsb;

    double gx_dps = gx * gyr_dps_per_lsb;
    double gy_dps = gy * gyr_dps_per_lsb;
    double gz_dps = gz * gyr_dps_per_lsb;

    double mx_g = mx * mag_gauss_per_lsb;
    double my_g = my * mag_gauss_per_lsb;
    double mz_g = mz * mag_gauss_per_lsb;

    printf("ACC raw[%6d %6d %6d] m/s^2[%7.3f %7.3f %7.3f] | ",
           ax, ay, az, ax_mps2, ay_mps2, az_mps2);
    printf("GYR raw[%6d %6d %6d] dps[%7.3f %7.3f %7.3f] | ",
           gx, gy, gz, gx_dps, gy_dps, gz_dps);
    printf("MAG raw[%6d %6d %6d] gauss[%7.4f %7.4f %7.4f]\n",
           mx, my, mz, mx_g, my_g, mz_g);
    usleep(100000);
  }
  close(fd);
  return 0; }
```

Task 4.4: Opening the I²C Communication Channel

At the start of the program, the I²C bus is opened using:

open("/dev/i2c-1", O_RDWR);

This allows the Raspberry Pi to communicate with all devices connected to I²C bus number 1.

Task 4.5: Verifying Sensor Identity

Before configuring the sensors, the program reads the **WHO_AM_I** register from each device to confirm proper communication.

Expected identification values:

Sensor	Expected WHO_AM_I value
LSM6DS33	0x69
LIS3MDL	0x3D

If these values are returned, it confirms that the correct sensors are connected and responding.

Task 4.6: Configuring Sensor Control Registers

After confirming communication, the program writes control words to the sensor configuration registers.

These control words define:

- Output data rate
- Measurement range
- Operating mode

Task 4.6.1 LSM6DS33 (Gyroscope and Accelerometer)

Device address: **0x6B**

Register	Address	Control Word	Function
CTRL3_C	0x12	0x44	Enables block data update and auto-increment
CTRL1_XL	0x10	**0x20**	Accelerometer: 26 Hz, ±2g
CTRL2_G	0x11	**0x24**	Gyroscope: 26 Hz, ±500 dps

Explanation:

- **CTRL1_XL = 0x20**
  Activates the accelerometer with:
  - Output data rate: 26 Hz
  - Measurement range: ±2g

- **CTRL2_G = 0x24**
  Activates the gyroscope with:
  - Output data rate: 26 Hz
  - Measurement range: ±500 degrees per second

Task 4.6.2 LIS3MDL (Magnetometer)

Device address: **0x1E**

Register	Address	Control Word	Function
CTRL_REG1	0x20	**0x60**	Output data rate: 5 Hz
CTRL_REG2	0x21	0x00	±4 gauss range
CTRL_REG3	0x22	0x00	Continuous-conversion mode
CTRL_REG4	0x23	0x0C	Ultra-high-performance Z-axis
CTRL_REG5	0x24	0x40	Block data update enabled

Explanation:

- **CTRL_REG1 = 0x60**
  Sets magnetometer output rate to approximately 5 Hz.
- **CTRL_REG2 = 0x00**
  Sets measurement range to ±4 gauss.
- **CTRL_REG3 = 0x00**
  Enables continuous measurement mode.

Task 4.7: Reading Raw Sensor Data

After configuring the sensors, the program enters a continuous loop and reads raw data from the output registers.

Each measurement is stored as:

- A low byte
- A high byte

These two bytes are combined to form a signed 16-bit value.

The program reads:

- Gyroscope X, Y, Z
- Accelerometer X, Y, Z
- Magnetometer X, Y, Z

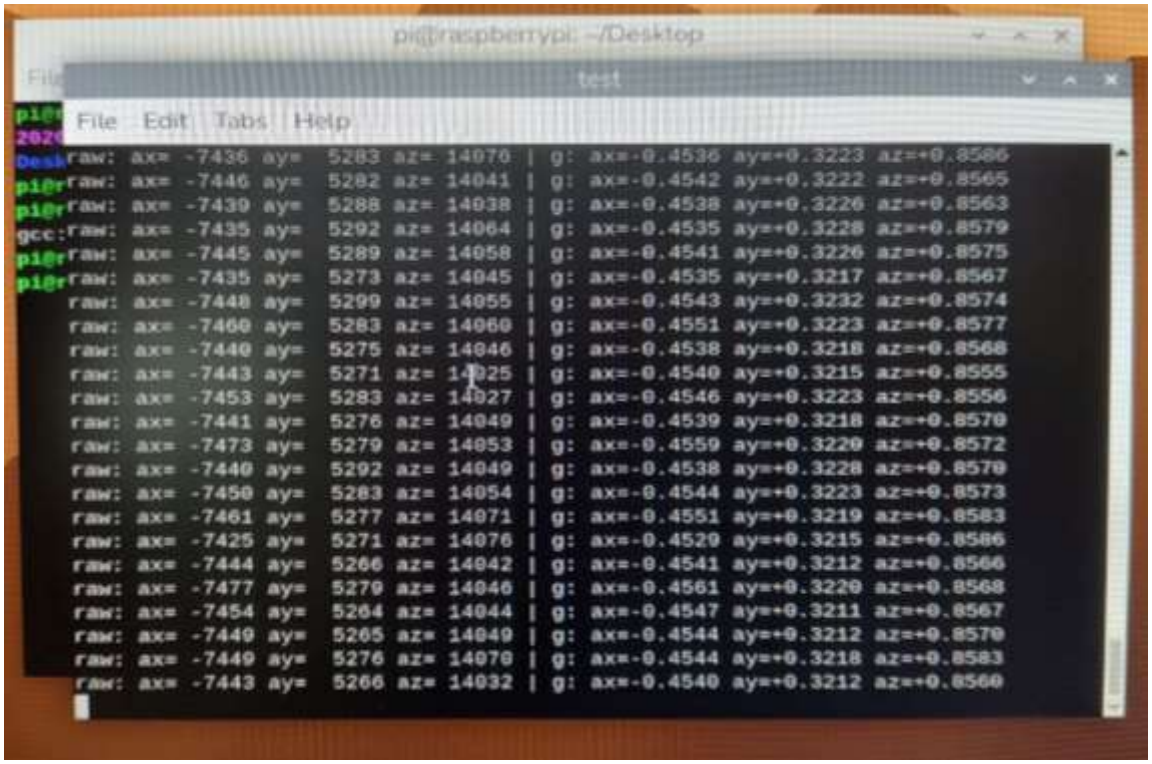

Fig. 14. Real-time accelerometer output displayed by the IMU reading program.

Task 4.8: Converting Raw Data to Physical Units

The raw values are converted into real-world units using scale factors based on the selected ranges.

Sensor	Unit
Accelerometer	meters per second squared (m/s^2)
Gyroscope	degrees per second (dps)
Magnetometer	gauss

This makes the data easier to interpret.

Task 4.9: Verifying Accelerometer Output

After running the program, we first observed the accelerometer output. The values changed continuously as the sensor was tilted and moved.

This confirmed that:

- The accelerometer was successfully activated.
- Control registers were correctly configured.
- I^2C communication was functioning properly.

Then we ran the full IMU data program, which displayed readings from the accelerometer, gyroscope, and magnetometer simultaneously. The values changed as the sensor was moved, confirming that all three sensors were working correctly.

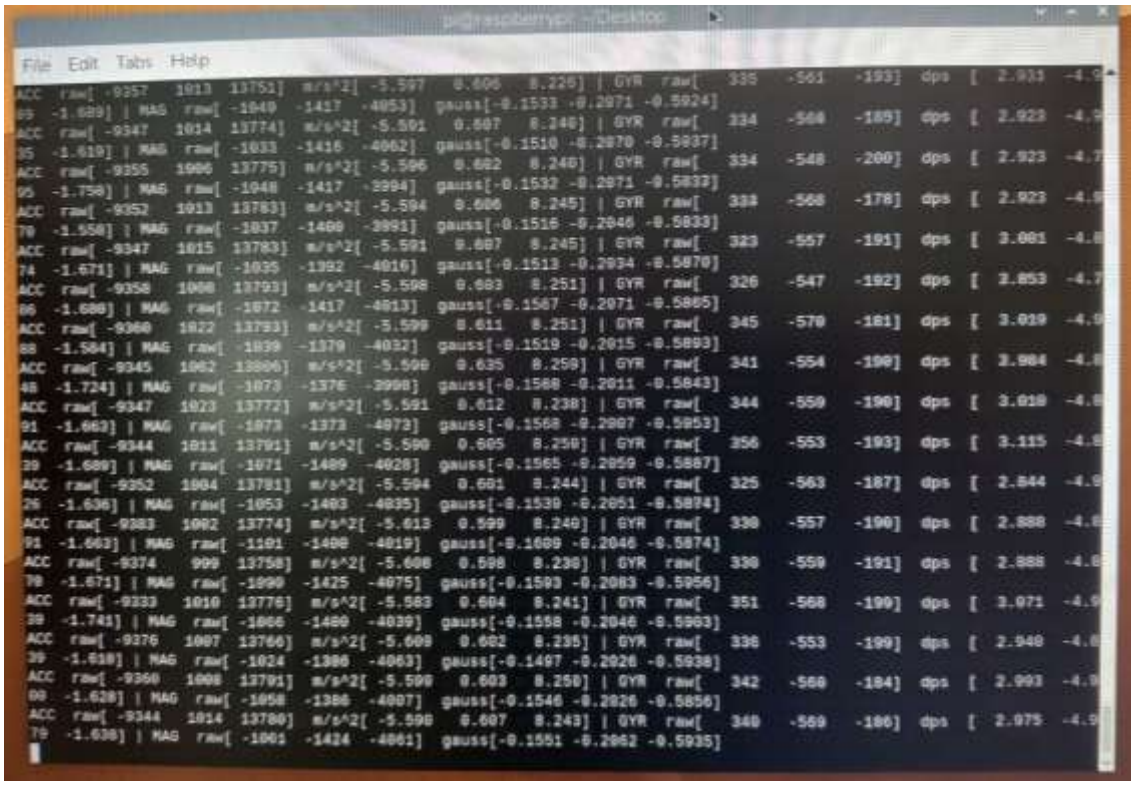

Fig. 16. Real-time accelerometer, gyroscope, and magnetometer output from the IMU program.

Explanation of Control Words

In digital sensors, a control word is a binary or hexadecimal value written into a configuration register to define how the sensor operates. Each bit in the control word corresponds to a specific function, such as output data rate, measurement range, operating mode, or axis enable settings.

When a control word is written to a register through the I²C interface, the sensor changes its internal configuration according to the bit values. This process is necessary because the sensors on the IMU are in a power-down state by default and must be configured before they can produce measurements.

In this project, control words were written to the gyroscope, accelerometer, and magnetometer configuration registers to set the required data rates and measurement ranges.

Gyroscope Control Word

The gyroscope is part of the **LSM6DS33** sensor. Its operation is controlled through the **CTRL2_G** register.

In the program, the following control word was written:

CTRL2_G = 0x24

The hexadecimal value **0x24** corresponds to the following binary value: 0010 0100

Each group of bits represents a specific function:

- Bits [7:4] define the **output data rate (ODR)**.
- Bits [3:2] define the **measurement range**.
- Bits [1:0] are reserved or used for additional settings.

For this configuration:

- The output data rate is set to **26 Hz**.
- The measurement range is set to **±500 degrees per second**.

This control word activates the gyroscope and configures it to measure angular velocity at the required frequency and range.

Gyroscope: Configured for:

- Output frequency: 26 Hz
- Range: ±500 deg/s

Control register bits set to enable axes and select the required scale.

Accelerometer Control Word

The accelerometer is also part of the **LSM6DS33** sensor and is controlled using the **CTRL1_XL** register.

In the program, the following control word was written:

CTRL1_XL = 0x20

The hexadecimal value **0x20** corresponds to the binary value: 0010 0000

The bit functions are:

- Bits [7:4] define the **output data rate**.
- Bits [3:2] define the **full-scale range**.
- Bits [1:0] define bandwidth or filtering settings.

For this configuration:

- The output data rate is set to **26 Hz**.
- The measurement range is set to **±2 g**.
- The accelerometer operates in normal mode.

Accelerometer: Configured for:

- Output frequency: 26 Hz
- Range: ±2g

This control word activates the accelerometer and allows it to measure linear acceleration along all three axes.

Magnetometer Control Words

The magnetometer is a separate sensor, the **LIS3MDL**, and requires multiple configuration registers.

CTRL_REG1 – Output Data Rate

The following control word was written:

CTRL_REG1 = 0x60

Binary representation: 0110 0000

This sets:

- Output data rate to approximately **5 Hz**.
- High-performance mode for the X and Y axes.

CTRL_REG2 – Measurement Range

The following control word was written:

CTRL_REG2 = 0x00

This sets the measurement range to:

- **±4 gauss**

TRL_REG3 – Operating Mode

The following control word was written:

CTRL_REG3 = 0x00

This sets the magnetometer to:

- **Continuous-conversion mode**

In this mode, the magnetometer continuously measures the magnetic field and updates the output registers automatically.

Magnetometer: Configured for:

- Output frequency: 5 Hz
- Range: ±4 gauss
- Continuous-conversion mode enabled.

Summary of Sensor Configurations

The control words written to the registers resulted in the following sensor settings:

Sensor	Output Frequency	Measurement Range	Operating Mode
Gyroscope	26 Hz	±500 deg/s	Normal mode
Accelerometer	26 Hz	±2 g	Normal mode
Magnetometer	5 Hz	±4 gauss	Continuous mode

Teamwork and Individual Contribution

We worked together to understand the hardware setup, I²C communication, and sensor configuration. All team members participated in the testing, and verification of the IMU sensor.

One team member prepared the report and presentation, while another returned to the laboratory to repeat the experiment and capture the required screenshots that were missed during the initial run.

Example

- Minh Hieu Le: Raspberry Pi setup, SSH configuration and Github.
- Azat Balapan: IMUwiring and I²C testing. Code.
- Bina Batool: Final Report and screenshots
- All Members: Register configuration and sensor data verification.

Conclusion

In this part of the project, we successfully activated and configured the IMU sensors according to the assignment requirements.

The control registers were programmed with the specified output frequencies and measurement ranges. After activation, the accelerometer, gyroscope, and magnetometer produced real-time data that changed with sensor motion.

These results confirmed that the hardware connections and software configuration were correct, and that the Raspberry Pi was successfully interfaced with the IMU sensor using the I²C protocol.
