GROUP 04

DOLON CHAPA, SHAMILA THENNAKOON, MAZIDUL ISLAM

PILL DISPENSER

PROJECT REPORT

School of IT

Metropolia University of Applied Sciences

16 December 2024

Abstract

Automated Pill Dispenser project aims to enhance medication adherence and improve healthcare efficiency through a smart, reliable dispensing system. The device, built around a Raspberry Pi Pico microcontroller, includes a rotating dispenser wheel, stepper motor for movement, and sensors for pill detection and calibration. The system operates autonomously, dispensing medication at scheduled times while ensuring accurate pill counts and real-time monitoring. It records essential data, such as pill dispensed counts and operational status, in an EEPROM and communicates device status via LoRaWAN. Designed for both personal and clinical use, the dispenser aims to reduce human error and improve patient compliance. Challenges in hardware integration, calibration, and network connectivity were addressed through effective teamwork and iterative problem-solving. The system demonstrates significant potential for future expansion, including mobile integration, multi-dosing capabilities, and enhanced connectivity options, ultimately contributing to a more efficient healthcare experience.

Contents

Introduction

Automated medication management systems play a crucial role in improving patient compliance, reducing human errors, and enhancing overall healthcare efficiency. This project focuses on the design and implementation of an Automated Pill Dispenser, a device that dispenses medication daily at a scheduled time, tailored to support patients with their medication routines.

The system's key components include a Raspberry Pi Pico microcontroller, a dispenser base, and a rotating dispenser wheel with eight compartments, seven for storing pills and one for calibration. A stepper motor controls the wheel's movement. The system also features a piezoelectric sensor to detect if a pill is dispensed and an optical sensor (opto-fork) for accurate wheel positioning and calibration.

The program records critical information, such as the total number of pills dispensed, the system's operational state, and detailed log messages, in an EEPROM using I2C communication. This ensures data is retained even after a system reboot. Additionally, device status updates are sent to a server via the LoRaWAN network and logged locally through *stdout* using a Debug Probe. The software is developed using Embedded C within the Raspberry Pi Pico SDK framework.

This documentation describes the purpose and the structure of the project, details the necessary technical features for successful implementation, and provides an insight to the challenges and the outcome of the project.

Methods and Material

In this section, the methodology and components utilized in the project will be outlined.

Pill dispenser

Pill dispenser based on controller PCB, dispenser base and dispenser wheel.

Controller PCB

For the project, the PCB controller was chosen as the microcontroller board, depicted in Figure 1. It is an electronic system designed to manage and regulate the functions of a printed circuit board (PCB). It includes a microcontroller, power circuitry, and input/output (I/O) interfaces. By programming the controller, it can oversee tasks like data acquisition, signal processing, and power management. Its primary role is to handle the timing, sequencing, and overall operation of the PCB. [1]

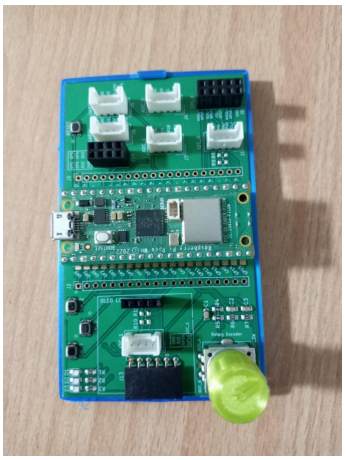


Figure1: Controller PCB

Dispenser base and dispenser wheel

Dispenser base contains a stepper motor for turning the dispenser wheel. The dispenser wheel has eight compartments, seven for pills and one for calibration. The wheel has no sensors but has an opening that can detected by the optical sensor of the base to calibrate the wheel position. [2]

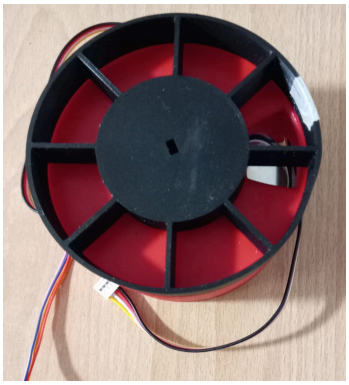


Figure2: dispenser base and dispenser wheel [2]

Stepper motor

A stepper motor is a type of electric motor designed to rotate its shaft in precise, fixed increments, or steps. This characteristic is achieved through the motor's internal design, enabling the exact angular position of the shaft to be determined by counting the number of steps it takes, without requiring a sensor. This capability makes stepper motors well suited for a variety of applications.[4]

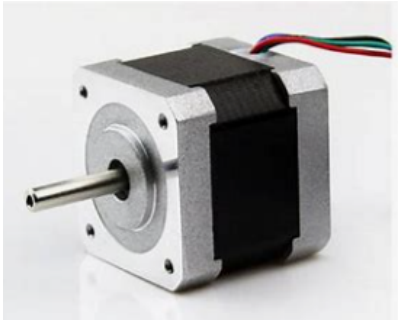


Figure3: Stepper motor [3]

Optical sensor and Piezo sensor

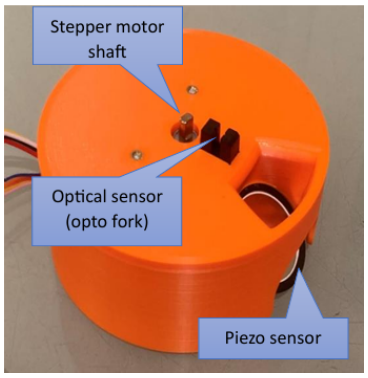


Figure 4: Optical sensor and Piezo sensor[2]

A piezo electric sensor for detecting if a pill is dispensed. When a pill hits the piezo sensor, it will create one or more falling edges in the input and an optical sensor for calibrating the wheel position.[2]

Stepper motor driver

Stepper motor drivers are devices engineered to operate stepper motors, enabling precise position control and continuous rotation without the need for a feedback system. So, it can connect the JST connector to the 6-pin connector on the stepper driver board.[2]

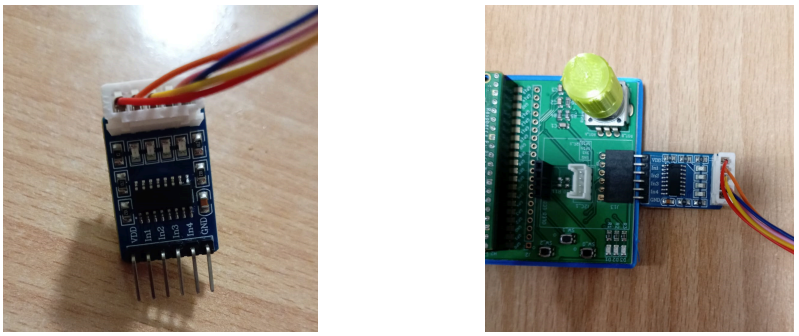


Figure 5: Stepper motor driver

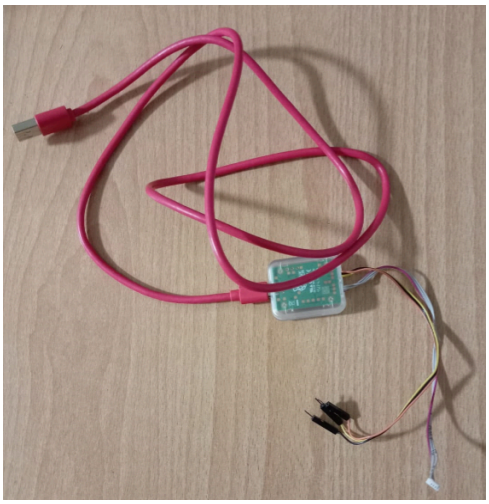
I^2C EEPROM

EEPROM stands for Electrically Erasable Programmable Read-Only Memory. It is a type of memory that retains data even when power is off, making it non-volatile. It can be erased and rewritten electrically, allowing for multiple updates to the stored data. This makes it ideal for applications requiring flexible and persistent storage of small data sets, like configuration settings or system parameters. Unlike traditional ROM, which is static and unchangeable after manufacture, EEPROM provides the advantage of reprogramming as needed, enhancing its utility in modern electronic devices.[5]

Figure 6: I^2C EEPROM

Pico Probe Debugger

The Raspberry Pi Debug Probe is a USB device offering a UART serial port and a standard Arm Serial Wire Debug (SWD) interface. It is designed for effortless, solder-free, plug-and-play debugging.[6]



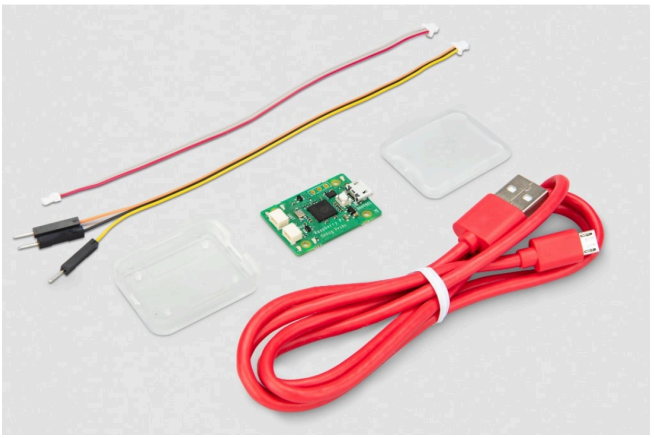


Figure 7: Pico Probe Debugger [6]

LoRaWAN

LoRaWAN is a low-bandwidth, long-range radio network standard designed for transmitting tiny amounts of data at a time. A LoRaWAN module enables devices to communicate their status to a server. The network can be configured as a private setup or purchased as a service from operators.[2]

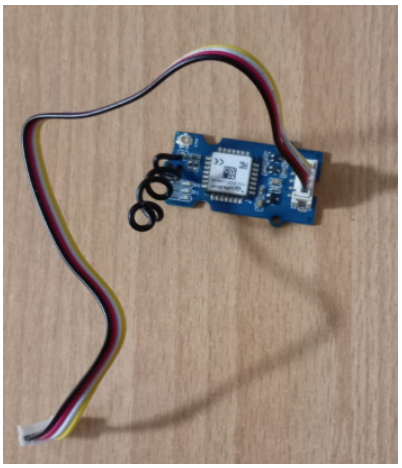


Figure 8: LoRaWAN

The pin dispenser base is equipped with a 6-pin JST connector and a 4-pin Grove connector. The JST connector has been connected to the 6-pin connector on the stepper driver board. There are multiple Grove connectors on the PCB. The Grove connector from the dispenser base has been connected to the ADC_1 connector. The LoRaWAN module has been connected to the UART1 connector under the stepper motor driver. The EEPROM has been connected to I2C_0 by a Grove connector.[2].

Pin Configuration

Component Configuration	GPIO Pin	Configuration
Optical Sensor (GP28)	Input	Pull-up, reads 0 at opening
Piezo Sensor (GP27)	Input	Pull-up, falling edges
Stepper Motor Driver	GP2, GP3	Outputs, connected to IN1-IN4
LEDs	GP20-GP22	Outputs
LoRaWAN Module (UART1)	GP4, GP5	RX/TX

Here is the experimental setup we have designed for the pill dispenser.

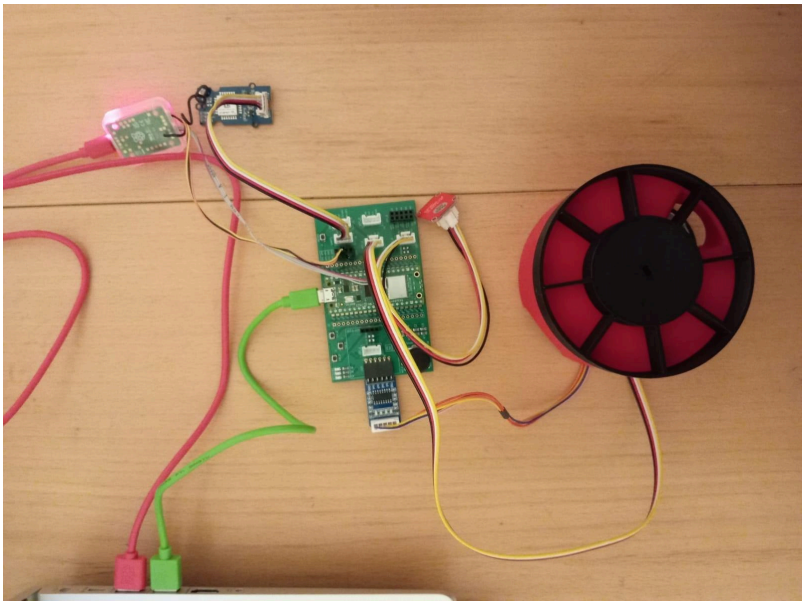


Figure 9: pill dispenser setup

Operating Principles of the Software

The purpose of this project was to implement an automated pill dispenser that dispenses daily medication to an individual. The Raspberry Pi Pico was utilized to control the motor, dispense pills, and facilitate communication with a LoRa module. For this purpose, the software was developed in C using the CLion IDE. The software was designed to manage a pill-dispensing mechanism, perform motor calibration, detect sensor triggers, save operational states to EEPROM for persistence, enable user interaction through buttons, and transmit messages via a LoRa communication module. Below is a detailed explanation of the operating principles of the software, categorized into features implemented under Minimum Requirements, Advanced Requirements, and LoRa Communication.

1. Minimum Requirements

- **Idle mode**

Idle State and Button Detection upon startup, the dispenser enters an idle state, indicated by a blinking LED, the device is ready for calibration. When the button (SW_1) is pressed, the dispenser proceeds to calibrate the motor.

For this purpose, initially we use 4 libraries
#include <stdio.h> - Provide functions for input and output libraries such as printf(), Scanf().
#include <stdbool.h>- This handle logical operations. bool type,in C, boolian operations (true, false)
#include "pico/stdlib.h"- Provide basic functions for the Raspberry Pi Pico's SDK. Such as GPIO control, UART Communication and time management function.
#include "hardware/gpio.h" - This provide advanced GPIO functions like gpio_set_dir(), gpio_put(), and gpio_get()

Then define the component with relevant connected pins on the board.
#define SW_0 9, SW_2 7, LED_1 20, IN1 2,IN2 3, IN3 6, IN4 13, OPTO_FORK 28, PIEZO_SENSOR 27, STEP_DELAY_MS 2

Code 1:Idle Mode

```
void IdleMode() {
    gpio_put(LED_1, value: 1);
    sleep_ms(200);
    gpio_put(LED_1, value: 0);
    sleep_ms(200);
}
```

- **Motor Calibration**

The motor performs at least one full turn to locate and align the wheel's compartment with the drop tube

Code 2: Motor Calibration

Code 3: Alignment

- **Pill Dispensing**

After calibration, it keeps the LED on until user press SW_2 button. Then it begins dispensing pills every 30 seconds starting from the button press. When the wheel is turned 7 times, the piezo sensor detects if a pill is dispensed. If no pill is detected, the LED blinks five times.

After dispensing seven pills, the system has been programmed to reset and return to the idle state, awaiting calibration.

Code 4: Pill Dispensing

```
void DispensePills() {
    int step_and_stop = 0;
    int pills_dispensed = 0;

    while (step_and_stop < 7) {
        sensor_triggered = false;
        run_motor(N:512);
        absolute_time_t time = make_timeout_time_ms(500);
        while (!sensor_triggered && !time_reached(time)) {
            tight_loop_contents();
        }

        if (sensor_triggered) {
            pills_dispensed++;
            printf("Pill dispensed %d.\n", step_and_stop + 1);
            sensor_triggered = false;
            sendLoRaMessage("Pill dispensed!");
        } else {
            printf("No pill dispensed %d.\n", step_and_stop + 1);
            sendLoRaMessage("No pill dispensed!");

            // Blink LED 5 times to indicate error
            for (int blink = 0; blink < 5; blink++) {
                gpio_put(LED_1, value:1);
                sleep_ms(200);
                gpio_put(LED_1, value:0);
                sleep_ms(200);
            }
        }
        sleep_ms(2000); // Wait for next cycle
        step_and_stop++;
    }
```

After completing the minimum requirements, the output has been displayed on the console as follows.

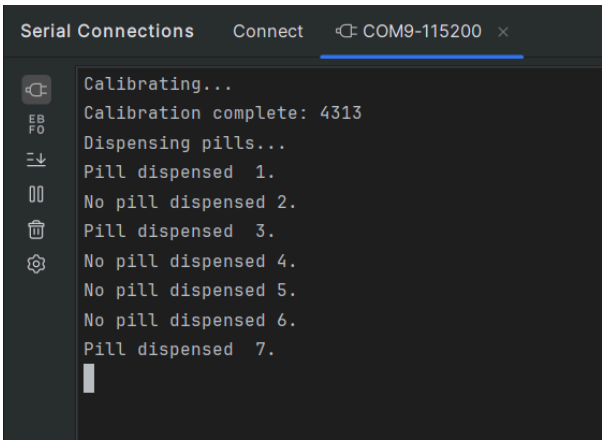


Figure 10: output of the minimum requirement

2. Advanced Requirements

Additionally in advance requirements it wanted to the device has been designed to retain its operational state across boot or power loss. Also it has been configured to connect to a LoRaWAN network and report status updates whenever a change has been detected, such as during boot, successful or failed pill dispensing, an empty dispenser, or power interruption during motor operation. Next the system has been programmed to detect if it was reset or powered off while the motor was turning and Following a mid-turn power loss, the device has been calibrated to automatically realign without dispensing the remaining pills from the interrupted cycle.

As a team, we have made our best effort to fulfill the advance requirements. While some aspects could not be fully achieved due to challenges in developing the appropriate code, we have successfully implemented the following parts.

- **State Retention Across Power Loss**

The motor state (position and whether it was turning) was saved to EEPROM before power loss and restored upon startup, allowing the device to resume the turn after power restoration if it powered down while the motor was turning.

Code 5: EEPROM State Management

```
void store_motor_state() {
    eeprom_write(address:EEPROM_CALIBRATION_STATE_ADDR, motor_position); // Store motor position
    eeprom_write(address:EEPROM_CALIBRATION_STATE_ADDR + 1, motor_state); // Store motor state
    printf("Motor state stored: Position = %d, State = %d\n", motor_position, motor_state);
}

void restore_motor_state() {
    motor_position = eeprom_read(address:EEPROM_CALIBRATION_STATE_ADDR);  // Load motor position
    motor_state = eeprom_read(address:EEPROM_CALIBRATION_STATE_ADDR + 1); // Load motor state
    printf("Motor state restored: Position = %d, State = %d\n", motor_position, motor_state);

    // If the motor was turning before the power loss, resume the turn
    if (motor_state == 1) {
        printf("Power loss detected, resuming turn from position %d...\n", motor_position);

        run_motor_to_position(motor_position);
    } else {
        printf("Motor is idle. Start with calibration...\n");
    }
}
```

- **I2C Communication with EEPROM**

Data read and written from an external EEPROM through the I2C protocol

Code 6: Data read and written from EEPROM through the I2C protocol

```
void initI2C() {
    i2c_init(I2C_PORT, baudrate: I2C_SPEED);
    gpio_set_function(I2C_SDA_PIN, fn: GPIO_FUNC_I2C);
    gpio_set_function(I2C_SCL_PIN, fn: GPIO_FUNC_I2C);
    gpio_pull_up(I2C_SDA_PIN);
    gpio_pull_up(I2C_SCL_PIN);
}
```

```
void eeprom_write(uint16_t address, uint8_t data) {
    uint8_t buffer[3];
    buffer[0] = (address >> 8) & 0xFF;  // High byte of the address
    buffer[1] = address & 0xFF;         // Low byte of the address
    buffer[2] = data;                   // Data to be written

    i2c_write_blocking(I2C_PORT, addr: EEPROM_I2C_ADDRESS, buffer, len: 3, nostop: false);
    sleep_ms(5);  // EEPROM write cycle time
}

uint8_t eeprom_read(uint16_t address) {
    uint8_t buffer[2];
    buffer[0] = (address >> 8) & 0xFF;  // High byte of the address
    buffer[1] = address & 0xFF;         // Low byte of the address

    i2c_write_blocking(I2C_PORT, addr: EEPROM_I2C_ADDRESS, buffer, len: 2, nostop: true);
    uint8_t data;
    i2c_read_blocking(I2C_PORT, addr: EEPROM_I2C_ADDRESS, &data, len: 1, nostop: false);
    return data;
}
```

- **Automatic Recalibration**

Having restored power during a turn, the device has been recalibrated without dispensing residual pills.

Code 7: Resuming Motor State

```
void run_motor_to_position(int target_position) {
    // Logic to move the motor to the saved position (if the motor was turning)
    while (motor_position != target_position) {
        activate_motor_step();  // Move the motor by one step
        if (motor_position < target_position) {
            motor_position++;
        } else {
            motor_position--;
        }
        sleep_ms(STEP_DELAY_MS);
    }
    printf("Motor successfully moved to position %d\n", target_position);
}
```

3. LoRa Communication

The LoRa communication module is used to facilitate remote communication for the pill dispensing system. The device is designed to send and receive messages via the LoRaWAN protocol, ensuring that status updates such as successful or failed pill dispensing, system calibration, and power loss recovery can be transmitted remotely.

The communication setup follows a series of initialization steps that ensure reliable connectivity, state reporting, and error handling. This process involves joining a LoRa network and sending status messages about the device's operations. The device leverages the LoRa-E5 module, which connects to an MQTT server to transmit messages to our terminal.

- **Initialization**

The device has stored the motor position and calibration status in the EEPROM. It has checked the system's last state. Before sending any data, the system has tested the connection to the LoRa module by sending the AT command to verify the module's readiness.

After initialization, the following commands have been executed to establish a connection with the LoRa network.

AT+ MODE =LWOTAA - Set the device to Over-The-Air Activation (OTAA) mode

AT+KEY=APPKEY, \"b184cdbc7ed2528badfd4efd9a28820f\ - Configure the application key for secure communication.

AT+CLASS=A - Set the LoRa class to A for low-power operation.

AT+PORT=8 - Specify port number 8 for communication

Code 8: LoRa Initialization

```
void initLoRa() {
    uart_init(UART_ID, baudrate: BAUD_RATE);
    gpio_set_function(4, fn: GPIO_FUNC_UART);
    gpio_set_function(5, fn: GPIO_FUNC_UART);

    uart_puts(UART_ID, s: "AT+MODE=LWOTAA\r\n");
    sleep_ms(500);
    readLoRaResponse();


    uart_puts(UART_ID, s: "AT+KEY=APPKEY,\"b184cdbc7ed2528badfd4efd9a28820f\"\r\n");
    sleep_ms(500);
    readLoRaResponse();

    uart_puts(UART_ID, s: "AT+CLASS=A\n");
    sleep_ms(500);
    readLoRaResponse();

    uart_puts(UART_ID, s: "AT+PORT=8\n");
    sleep_ms(500);
    readLoRaResponse();

}
```

- **Join Process**

The device has been connected to the network.

We have installed the paho-mqtt Python package, which is used for communication between devices. It was installed using **pip install paho-mqtt**. Then, we connected to a special Wi-Fi network (KME662) using the password SmartIot. This network connection was necessary for the device to communicate with the LoRaWAN gateway or receive data. After that, I ran python **lorareceive.py <device name>**. Subsequently, I was able to connect to the internet to send messages via the LoRaWAN module.

```
PS C:\Users\shami> python lorareceive.py Mazidul
Connecting to 192.168.1.10 with id Mazidul-577
C:\Users\shami\lorareceive.py:54: DeprecationWarning: Callback API version 1 is deprecated, update to latest version
  self.client = mqtt_client.Client(mqtt_client.CallbackAPIVersion.VERSION1)
Connected to MQTT Broker!
Text: TEST
```

Figure 11: Connect to the network

After ensuring that the device is successfully connected to the network, then join the network +JOIN command.

code 9: Join LoRa to the network

```
// Join LoRa Network
bool joinLoRa() {
    uart_puts(UART_ID, s:"AT+JOIN\n");
    printf("Joining LoRa network...\n");

    // Define a timeout (e.g., 30 seconds)
    uint32_t start_time = to_ms_since_boot(t:get_absolute_time());
    uint32_t timeout_ms = 30000; // 30 seconds timeout
    bool result = false;

    while (strstr(lora_response, "+JOIN: Done") == NULL) {
        readLoRaResponse();

        if (strstr(lora_response, "+JOIN: NetID") == NULL) {
            result = true;
        }

        // Check if timeout has occurred
        if (to_ms_since_boot(t:get_absolute_time()) - start_time > timeout_ms) {
            printf("Timeout while waiting for LoRa join response.\n");
            return false;
        }
    }
```

```
    // Check if the join was successful
    if (result) {
        printf("Successfully joined LoRa network.\n");
        return true;
    } else {
        printf("Failed to join LoRa network. Response: %s\n", lora_response);
        return false;
    }
```

- **Send Messages**

Once the device has joined the network, it can check by sending messages using the AT+MSG= "<message>" Command.

code 10: Send Massage via LoRa

```
// Function to send a LoRa message
void sendLoRaMessage(const char* message) {
    uint32_t start_time = to_ms_since_boot(get_absolute_time());
    uint32_t timeout_ms = 30000; // 30 seconds timeout
    bool result = false;
    bool timeout = false;
    char command[BUFFER_SIZE];
    snprintf(command, BUFFER_SIZE, "AT+MSG=\"%s\"\r\n", message);

    for (int i = 0; i < 3 && !result; ++i) {
        timeout = false;
        uart_puts(UART_ID, command);
        //sleep_ms(500);
        while (strstr(lora_response, "+MSG: Done") == NULL && !timeout) {
            readLoRaResponse();

            if (strstr(lora_response, "+JOIN: FPENDING") == NULL) {
                result = true;
            }

            // Check if timeout has occurred
            if (to_ms_since_boot(get_absolute_time()) - start_time > timeout_ms) {
                printf("Timeout while waiting for LoRa join response.\n");
                //return false;
                timeout = true;
            }
        }
    }
```

To check the read LoRa response we can use this code,

Code 11: Read LoRa response

```
void readLoRaResponse() {
    if (uart_is_readable_within_us(UART_ID, us: UART_TIMEOUT * 1000)) {
        int len = uart_read_line_with_timeout(UART_ID, lora_response, len: BUFFER_SIZE, timeout_ms: UART_TIMEOUT);
        if (len > 0) {
            printf("LoRa Response: %s\n", lora_response);
        } else {
            printf("No response from LoRa module.\n");
        }
    }
}
```

Finally we could send a message to the internet via LoRa module.

Here is the Response from cLion console:

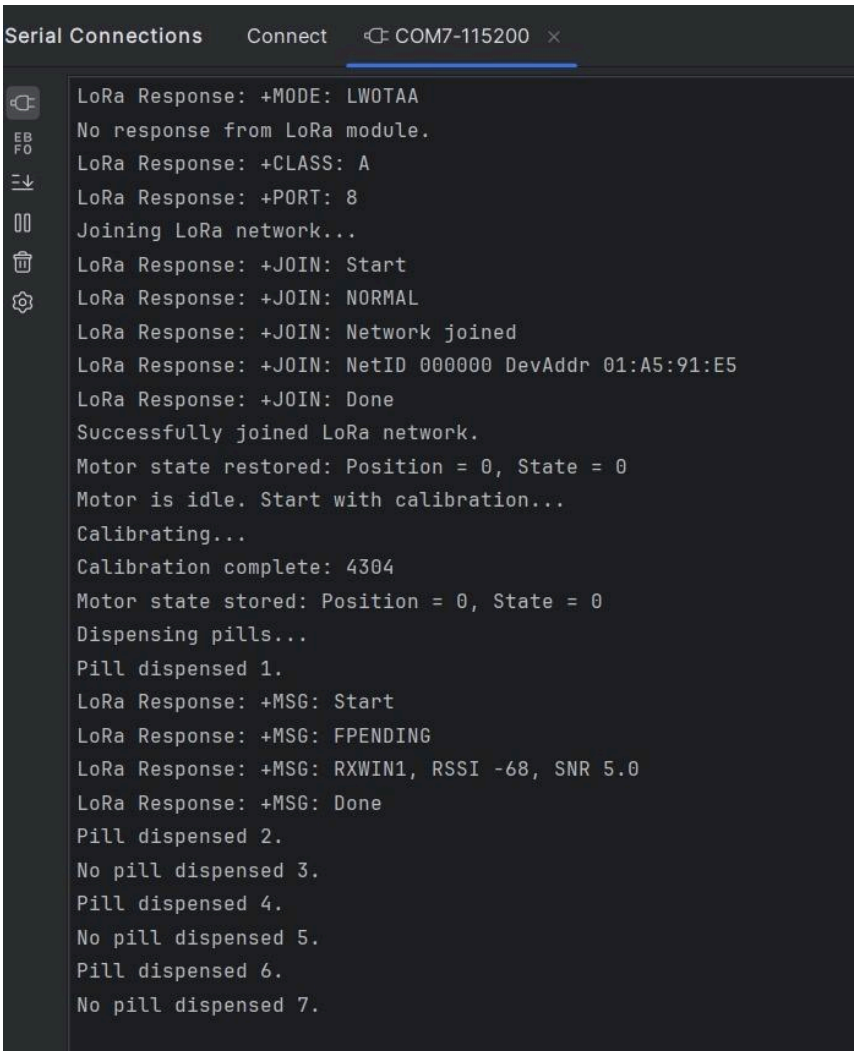


Figure12: Response of Clion console

Here is the LoRa response from windows power shell command prompt,

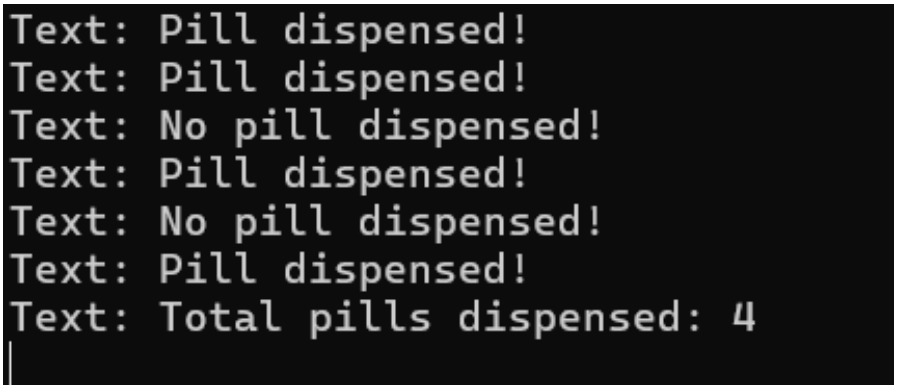


Figure 13: LoRa response of windows power shell command prompt

Flow Charts

The system works in general.

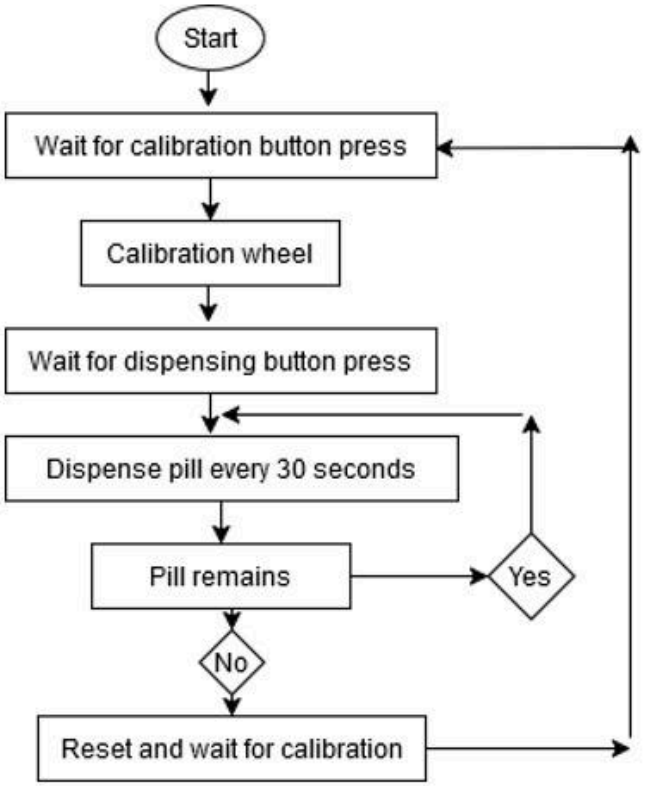


Figure 14: Flow chart of general system work

The software working flow chart:

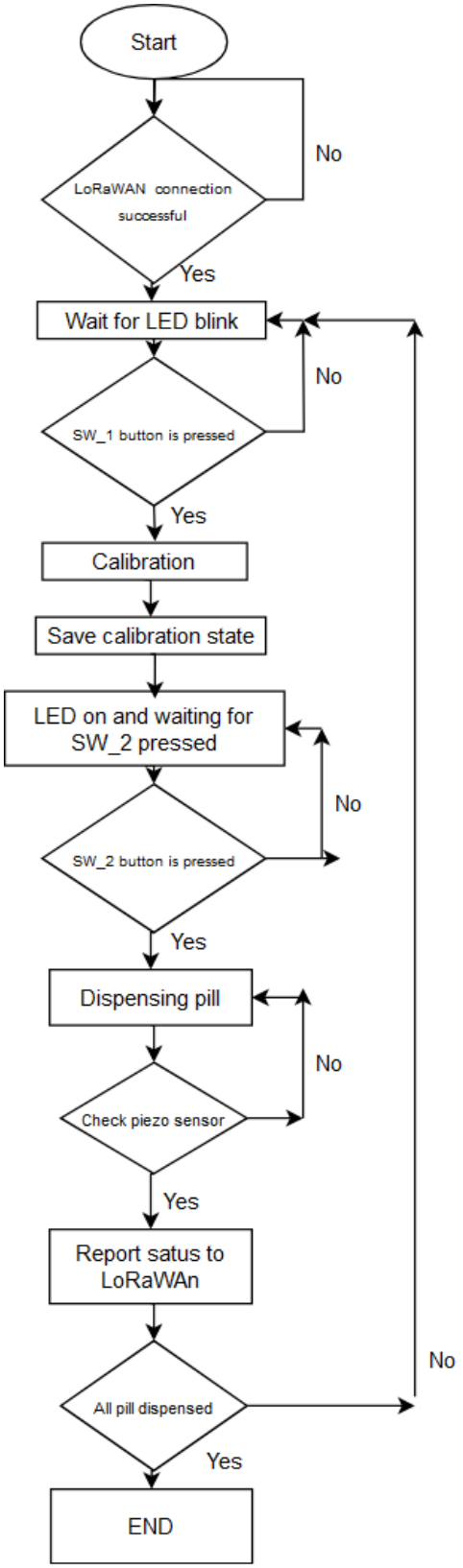


Figure 15: Flow chart of software work

Divide work between team members

DOLON CHAPA : During the project, I played an active role in all implementation sections, including system development and functionality testing. I was responsible for integrating and coding essential functions such as initLoRa, joinLoRa, sendLoRaMessage, calibrate_motor, restore_motor_state, and store_motor_state. I ensured the motor control system operated reliably by implementing functions like activate_motor_step, run_motor_to_position, and align_motor. Additionally, I worked on EEPROM data management, I2C communication, and UART handling. I contributed to debugging critical issues and ensuring the codebase remained consistent. Beyond coding, I collaborated closely with my team to refine workflows and ensure the project objectives were met effectively.

SHAMILA THENNAKOON : In this project, I was responsible for completing the minimum requirement section and successfully accomplished all the specified tasks. Additionally, I worked on advanced features, such as storing the motor state, and put significant effort into developing and implementing the advanced requirement sections. I also contributed to the LoRa communication component by sending messages using the lorareceive.py script over the LoRaWAN network. Throughout the project, I effectively applied theoretical concepts that I had researched to the coding tasks, ensuring the development of functional and efficient solutions. I contributed to the project report by writing the abstract, introduction, and detailing the methods and components used, with relevant references. I also elaborated on the operating principles of the software, implemented the flowcharts, reflected on the project work, and wrote the conclusion.Additionally, I collaborated with my team to modify, validate, and finalize the project report while streamlining processes to ensure the project goals were achieved efficiently.

MAZIDUL ISLAM : My role in this project involved testing and checking both the minimum requirements and advanced requirements of the project, which included, for example, checking whether the LoRaWAN module worked. I also provided documentation on the project by drawing flow charts on the general workings as well as software working. Mycontribution enabled testing to be done adequately so that the system functioned well and the processes were clear to the person involved in the team.

Reflection of the project work

In our project, we developed an automated pill dispenser designed to dispense daily medications with accuracy and efficiency. The system combines a stepper motor-driven dispenser wheel, optical and piezoelectric sensors for calibration and pill detection, and a LoRaWAN module for remote communication. The software ensures precise motor control, real-time monitoring, and state preservation using EEPROM, while the dispenser operates autonomously with user-friendly button controls and LED indicators. Designed to deliver reliable performance, the system is robust even during power interruptions. This project involved a collaborative effort to integrate hardware and software into a functional and dependable prototype.

In our pill dispenser project, we faced challenges with hardware compatibility, coding complexity, and calibration accuracy. Aligning the wheel with the sensor during calibration and implementing automatic recalibration after power loss were particularly difficult. LoRaWAN connectivity issues also posed challenges. Despite these setbacks, we are working to deliver a reliable and functional device.

Teamwork was essential to our success. Each member contributed unique skills, including hardware integration, software development, coding, and documentation, allowing us to approach problem-solving and project execution from multiple perspectives. Regular discussions kept the team aligned with project goals. By addressing challenges collectively, we developed efficient and effective solutions while also building technical expertise and strengthening teamwork dynamics.

The automated pill dispenser can greatly improve how people take their medication and help make healthcare more efficient. Adding features like a mobile app for reminders and scheduling, support for dispensing pills multiple times a day, and integration with health monitoring devices can make it more user-friendly. Adapting the design for use in hospitals or elderly care homes could help manage medications for many patients. Additionally, adding Wi-Fi or Bluetooth connectivity, along with LoRaWAN, can make it more versatile for personal and medical use, turning it into an essential healthcare device.

Conclusion

In conclusion, the automated pill dispenser project achieved significant milestones, demonstrating its potential to improve medication adherence and healthcare efficiency. The system effectively integrated hardware and software components, offering reliable pill dispensing, state management, and remote monitoring through LoRaWAN connectivity. While the project successfully delivered a functional prototype, challenges in advanced features, coding complexity, and network connectivity highlighted areas for improvement. Issues such as precise calibration after power interruptions and detecting resets during motor operation remain unresolved, underscoring the need for further development.

Despite these challenges, the project exemplified the power of teamwork, with each member contributing unique expertise to overcome obstacles and achieve shared goals. Regular collaboration allowed for efficient problem-solving and strengthened both individual and group capabilities. The lessons learned during this project, particularly in debugging, hardware-software integration, and network communication, will be invaluable for future endeavors.

Moving forward, addressing current limitations is essential to enhance the dispenser's accuracy, scalability, and user experience. Future enhancements could include features like a mobile app for scheduling, support for multiple dispensing times, and integration with health monitoring devices. By building upon this foundation, the dispenser could evolve into a versatile, impactful tool for both personal and clinical healthcare applications.

References

[1] Blog / PCB Controller: Everything You Need to Know – NextPCB. Accessed: 14 december 2024.

https://www.nextpcb.com/blog/pcb-controller

[2] Workspaces - OMA - Embedded Systems Programming TX00EX76-3005-document-project-pill-despenser (PDF).

https://oma.metropolia.fi/tyotilat?p_p_id=WorkspacePortlet_WAR_workspaceportlet&p_p_lifecycle=0&p_p_state=normal&p_p_mode=view&_WorkspacePortlet_WAR_workspaceportlet_struts.portlet.action=%2Fworkspace%2Fdocuments%2Findex&_WorkspacePortlet_WAR_workspaceportlet_struts.portlet.mode=view&workspace.id=340042128&workspace.name=Embedded%20Systems%20Programming%20TX00EX76-3005&currentFolder=10036746

[3] Nanotec, ST4118, Stepper Motor - NEMA 17(Data Sheet) . Accessed: 14 December 2024.

https://www.farnell.com/datasheets/2337767.pdf?_gl=1*jrwsxm*_gcl_au*MjA5ODk1MjY4Ni4xNzM0MTk4Mjg2

[4] Carmine Fiore ,**2020-stepper-motors-basics-types-uses-and-working-principles_r1.0_1,** PDF (media.monolithicpower.com). Accessed: 14 December 2024.

https://media.monolithicpower.com/mps_cms_document/2/0/2020-stepper-motors-basics-types-uses-and-working-principles_r1.0_1.pdf?_gl=1*1cg7t2q*_ga*MjAzMDkwNjU0NS4xNzM0MTk3MzQ3*_ga_XNRPF6L9DD*MTczNDE5NzM0NC4xLjEuMTczNDE5NzQwNi40LjAuMA..*_gcl_au*NDQwNDI5NTM2LjE3MzQxOTczNTE.&_ga=2.151768703.730536013.1734197347-2030906545.1734197347

[5] Blog admin, EEPROM (Electrically Erasable Programmable Read-Only Memory) – Definition & Detailed Explanation – Hardware Glossary Terms, June 6, 2024. Assessed: 15.12.2024.

https://pcpartsgeek.com/eeprom-electrically-erasable-programmable-read-only-memory/

[6] Raspberry Pi Debug Prob. Assessed: 14.12.2024.

https://www.raspberrypi.com/documentation/microcontrollers/debug-probe.html