HOCHSCHULE
RHEIN-WAAL
Rhine-Waal University
of Applied Sciences

The Talking Mailbox

2907 Sensors and Actuator Networks

Winter Semester 2025/26

Authors:

Justin Julius Chin Cheong
34140
MSE

Abhinav Kothari
33349
MSE

Contents

1 Introduction

1.1 Problem Statement

All Professors and Lecturers have a lot to do and may not always have time to check their mailbox. Imagine how long some letters are left in the mailbox for days just because a professor is busy. On the other hand, checking your mailbox only to find nothing is quite frustrating. What if there was a way that your mailbox could tell you when there is mail? What if you had a talking mailbox?

To solve this problem, we introduce **The Talking Mailbox**. The aim of The Talking Mailbox project is to design and assemble a system that can detect the presence of mail within a mailbox in Building 06 and notify the owner of the mailbox.

1.2 Literature Review

Before developing The Talking Mailbox, various existing smart mailbox solutions were reviewed, considering their sensor technology and communication methods as well as their advantages and shortcomings.

Perhaps the simplest solution is presented in frankenfoamy (2021). A wire connected to the door protrudes out of the box and holds down a flag. Once opened, the flag is released indicating the mailbox has been opened. On the opposite end of the technological spectrum, several commercial smart mailboxes are available. The Fouvin (2025) model uses a passive infrared (PIR) motion sensor to detect mail presence, while Notific (2025) employs a motion sensor[1] attached to the mailbox door. The Fouvin (2025) connects directly to Wifi and the Notific (2025) to LoRaWAN, providing remote notifications through dedicated apps. Similar to the Notific (2025), the InstaView (2024) detects door openings utilising a tilt sensor, while the X-Sense (2025) combines a tilt sensor and an IR motion sensor for enhanced detection. Both of these models communicate with a base station within the home via radio frequency (RF) technology, with the X-Sense (2025) capable of managing multiple mailboxes.

While these solutions certainly have merit, The Talking Mailbox offers some advantages over them and has a degree of novelty. While very cheap and reliable, the frankenfoamy (2021) has no means of remote notification. The Fouvin (2025) and X-Sense (2025) both use IR sensors which may be prone to false positives from environmental heat sources. The InstaView (2024) and X-Sense (2025) require an additional base station device which increases complexity of setup. The Talking Mailbox eliminates these issues as it connects to LoRaWAN directly and uses multiple sensors (that are not IR-based) to mitigate false positives. Most critically, all of these solutions rely on inferring mail presence from door movement or motion within the box, which may not always be accurate. The Talking Mailbox directly detects mail presence using a weight sensor, providing a more reliable solution.

1.3 Project Plan

1.3.1 Schedule

The Talking Mailbox project is planned to be executed over a period of 3 months, starting from October 2025 to January 2026. The project is divided into several milestones as shown in Figure 1. These deliverables and milestones adhere to the requirements set out in Section 1.4.3.

[1] While no literature could be found on the exact technology used, the sensor is likely simple contact switch like a reed switch

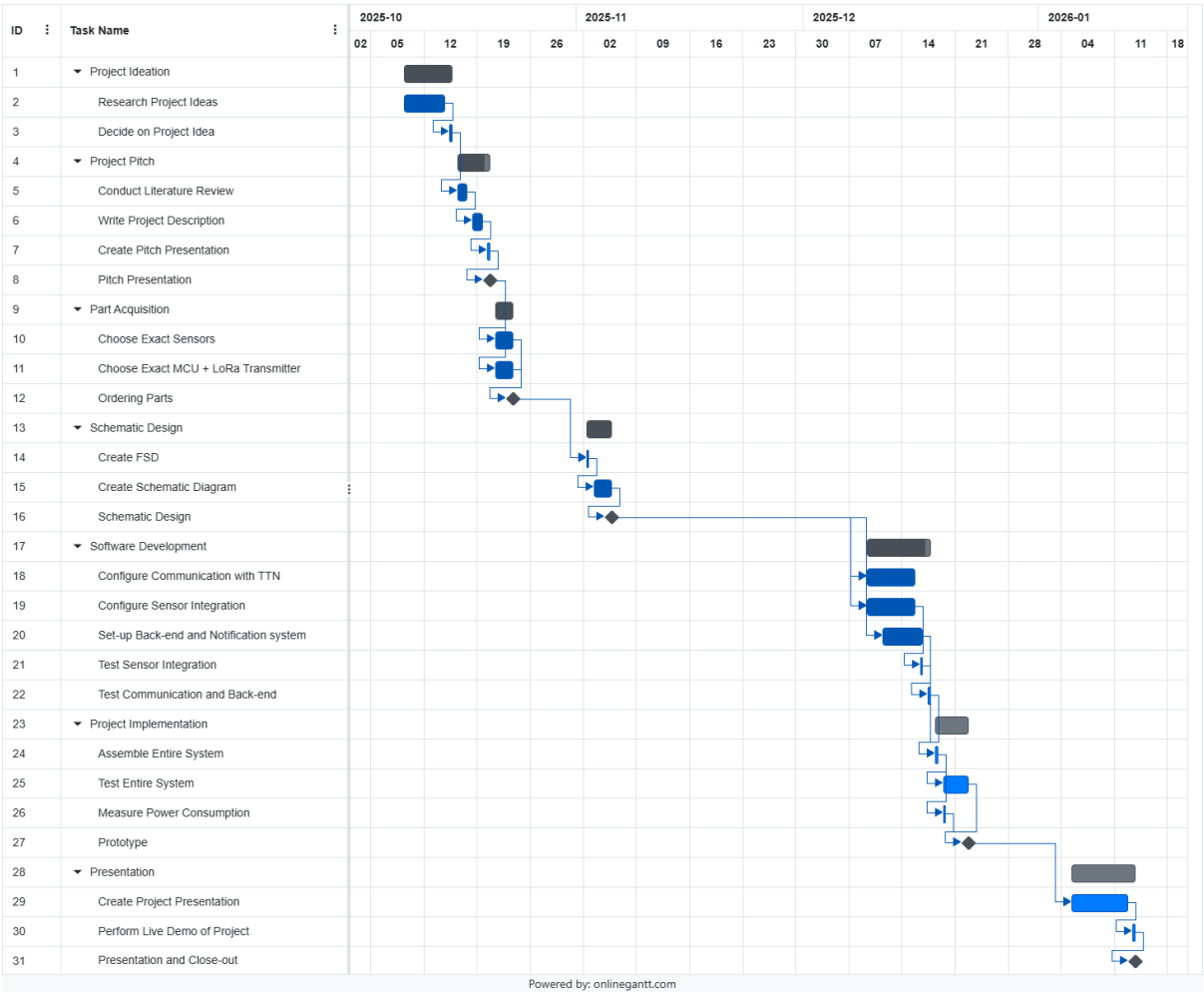


Figure 1: Gantt Chart for The Talking Mailbox Project

1.3.2 Resources

The resources for the project include the budget shown in Section 1.4.3 and all materials purchased as part of the Bill of Materials in Section 3.1.2.

The project team consists of two Mechatronics Engineering Students whose responsibilities are divided as follows:

- **Justin Julius Chin Cheong (34140):** Schematic Design, Sensor integration, Microcontroller programming
- **Abhinav Kothari (33349):** Backend Development, Server Setup and Email Notification System.
- **Both Students:** Component Choice, System Design, System Assembly & Implementation, System Testing, Report Writing and Presentation Preparation.

1.4 System Requirements

1.4.1 Functional Requirements

For The Talking Mailbox to be a satisfiable product, the following functional requirements must be implemented:

- It can detect if the mailbox is opened.
- It can detect light as a redundancy for confirming the opening status of the mailbox.

- It can detect whether or not mail is present within the mailbox.
- It can communicate if mail is in the box to a website / dashboard (based on LoRaWAN).
- It alerts the responsible person via email or dashboard upon mail detection.
- It can check the battery status.
- It sends battery status updates to a website at regular intervals.
- It sends a low battery warning to a website when the battery falls below a defined threshold.
- It should run for at least 1 week on a single charge.

1.4.2 Technical Requirements

For The Talking Mailbox to operate and perform its functions, the following technical requirements must be implemented:

- The weight sensor can detect a change in weight of approximately 20 g. This indicates when a piece of mail has been placed within the box.
- The tilt sensor can detect the rotation of the post box lid. This indicates when the lid is opened.
- The LDR can detect the change in light intensity by a defined threshold. This indicates when the lid is opened.
- The transmitter can reliably connect and communicate via the LoRaWAN Gateway.
- The server with which the LoRaWAN communicates can send emails to relevant personnel about the mail.
- The power supply is a battery with a working voltage of 3.1 V to 4.2 V.
- The enclosure can protect the system within a typical indoor environment (IP 31).
- The system should function at temperatures ranging 0–40°C and humidity 10–90%.

1.4.3 Project Requirements

For The Talking Mailbox project to produce a functional product upon close out, the following project requirements must be met:

- The budget is 100€.
- The project workload is estimated at 100 h.
- The project schedule adheres to the following deadlines:

Pitch:	2025-10-21
Bill of Materials:	2025-10-23
Schematic Design:	2025-11-23
Project Implementation:	2025-12-19
Project Report:	2026-01-05
Project Presentation and Demo:	2026-01-17

2 Theoretical Background

Before starting with actual project, some theoretical framework is required.

2.1 Communication

As for the communication, this project used LoRa as the communication encoding and LoRaWAN as the MAC Protocol.

2.1.1 LoRa

LoRa (**Lo**ng **Ra**nge) is the physical layer and is a modulation technique which allows for wireless communication. It is able to send information long ranges, with relatively less energy. It is derived from Chirp Spread Spectrum (CSS). It encodes information similar to how bats/dolphins communicate. LoRa is used extensively with sensors and actuator projects for the following reasons:

- Low power consumption
  - Transmitting: 10 mA
  - Sleep: 100 nA
- Long range → upto 15 km
- Robust against interferences

There are many more reasons as well, but these are the primary which were kept in mind for selecting it for this project.
LoRa works on a license free frequency range, in Europe this is EU868 (863–870/873 MHz). This will be used in this project (Precisely: 868.1 MHz).

2.1.2 LoRaWAN

LoRaWAN (**LoRa** **W**ide **A**rea **N**etwork) on the other hand is the data link layer on top of LoRa. It defines the communication protocols and architecture. After the initial release in January 2015, many versions have been released, with latest being 1.0.4 (Series 1.0) and 1.1 (Series 1.1) being released in October 2020 and October 2017 respectively. (Yes, 1.0.4 is newer than 1.1). The version used in this project is 1.0.4. Figure 2, shows how LoRa and LoRaWAN differ and work together.

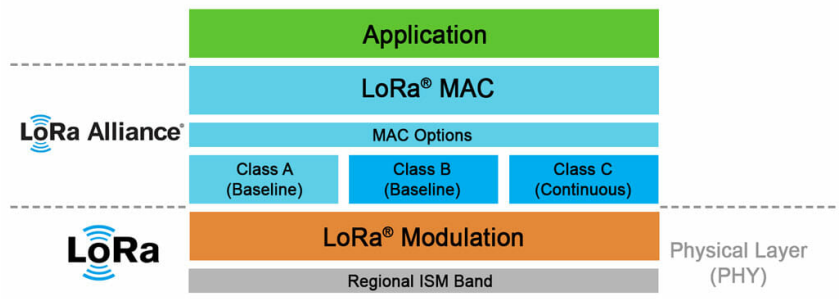


Figure 2: LoRa and LoRaWAN

2.2 Sensors

This project used 3 sensors, their functioning has been briefly described below. This framework is essential to understand the design choices made in this project in Section 3.1.

2.2.1 Load Cells

Load Cells are the primary sensor for this project and should be able to detect the presence of mail. Load cells are used to measure force, and hence can achieve this task. The specific load cell used in this project, is a strain gauge load cell. Strain gauges in the cell are arranged in a way that applying force changes resistance of the gauges in arranged in a wheatstone bridge and hence send out a voltage. This voltage is very small, and hence must be amplified. This amplification is done with the HX711 board, which makes it readable for the microcontroller (ESP32-S3). More specific details regarding these equipment can be seen in Section 3.1.2.

2.2.2 Tilt Switch

Tilt switch is being used to detect the opening of the lid of the postbox. There are multiple types of tilt switches mechanical (rolling ball/ liquid mercury) or electronic (MEMs). The one used in this project (IDUINO, n.d.) is a mechanical rolling ball switch, due to its lower voltage requirement as well as it being a safer option. The functioning can be demonstrated by Figure 3. Whenever the switch

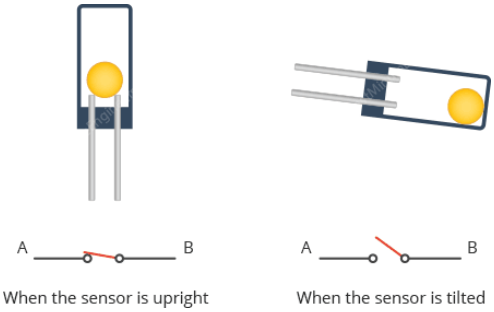


Figure 3: Tilt switch working

is in a specific orientation the ball allows for contact and hence making an electrical connection, else there is no connection.

2.2.3 LDR

A light dependent resistor is just a resistor which varies its resistance based on the light intensity. This can be detected and hence compared to a threshold to check if the box is open or not.

3 Methodology and Design

3.1 System Layer

3.1.1 System Architecture

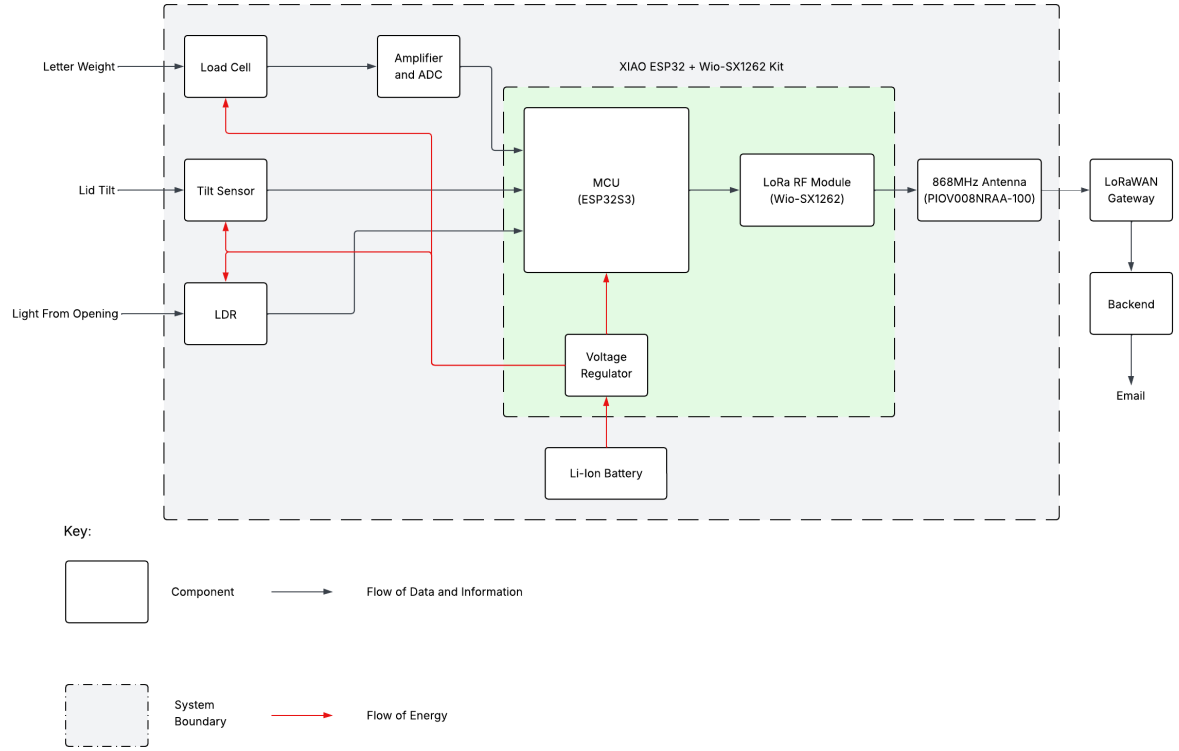


Figure 4: Functional Structure Diagram of the System Architecture

3.1.2 Bill of Materials

This is an estimate of the materials required to make this project, these are all over estimates, as all components/materials except 2.1 - 2.6 were used from the university stock.

Item	Part	Description	Qty	Notes	Total Price (€)
1.0 Mechanical Components					
1.1	Top Plate	Detection Plate	1	Wood (40x30x0.5cm)	5.00
1.2	Bottom Plate	Base for load cell	1	Wood (15x15x0.5cm)	3.00
1.3	M4 Screw	Top plate screw	2	-	1.00
1.4	M5 Hex Nut	Height spacer nut	4	-	0.50
1.5	M5 Screw	Bottom plate screw	2	-	1.00
1.6	M6 Hex Nut	Height spacer nut	4	-	0.50
1.7	Misc.	Tape, Glue	-	-	0.50
2.0 Electrical Components					
2.1	Load Cell + HX711	Load cell + Amplifier module	1	JOY-IT	6.40
2.2	Tilt Switch	Ball tilt switch	1	IDUINO	0.94
2.3	LDR	Light resistor	1	SERTRONICS	1.35
2.4	Battery	1800 mAh Li-Ion	1	SOLDERED	10.24
2.5	MCU + LoRa	Xiao ESP32 + SX1262	1	Seeedstudio	11.68
2.6	Antenna	Long range antenna	1	Amphenol-SAA	2.69
2.7	1 kOhm resistor	Through Hole	1	YAGEO	0.10
2.8	2 kOhm resistor	Through Hole	1	YAGEO	0.10
2.9	Wires	Jumper wires of different length	-	-	0.30
2.10	Misc.	Breadboard, Wire Sleeves, Solder	-	-	0.50
				Tax (VAT 20%)	9.16
				Grand Total (€)	**54.96**

Table 1: Combined Mechanical and Electrical Bill of Materials with Total Cost

3.2 Hardware Layer

3.2.1 Load Cell Assembly

According to Joy-IT (n.d.), to reliably detect any weight, the load cell must be allowed to bent as easily as possible and thus be secured within a platform. To prevent any mail from "missing" the detection platform, the platform must cover the entire box area. The design which allows this is shown in Figure 5.

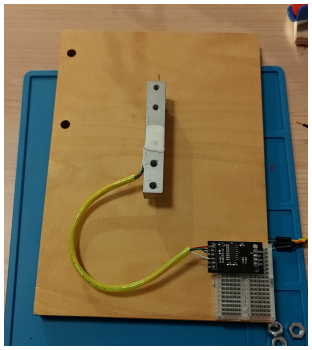
(a) 3D Model of Load Cell Arrangement

(b) Platform for mail detection

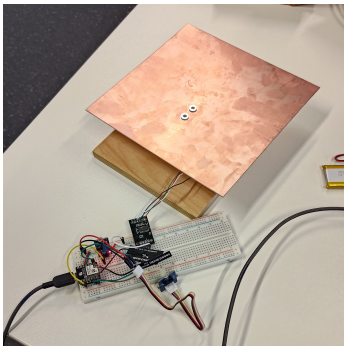
Figure 5: 3D Model of detection platform

The height with the hex nuts, allow the load cell to have clearance to bend during load from the upper platform. The off center screw platform on both platform allow for best readings from the load cell. The size of the upper platform should match the post box size, however the lower platform allows for flexibility, it can be made just large enough to not cause toppling over and allow us to place necessary modules (such as the HX711).

Some other design considerations include the requirement of the platform being rigid and lightweight, to allow load cell to be as sensitive as possible. A good material to use for the platform is wood. The screws should also be flat with the platform, to prevent mail from tearing / getting stuck. The process of assembling the load cell platform is shown in Figure 6.



(a) Construction of Lower Platform



(b) Load Cell Platform Prototype

Figure 6: Weight Sensor Construction

The prototype shown in Figure 6b was then improved upon by increasing the size of the top platform to cover a greater area within the mailbox as well as adding a backboard to ensure mail does not slide off of the platform when the mailbox is opened. The final design is shown in Figure 11.

3.2.2 Schematic Design

The complete schematic design of the system is shown in Figure 7. The design is quite simple as the light and tilt sensors are digital sensors and connect directly to any GPIO pins on the microcontroller. The load cell on the other hand connects to the HX711 amplifier board, which then has a data line and clock line connecting to the ESP32-S3. The SX1262 LoRa module is connected via SPI, but the depiction in Figure 7 is a bit inaccurate. The kit comes with B2B connection which allows the MCU and LoRa module to be connected directly without any wiring. The pin mapping between the sensors and the microcontroller is shown in Table 2.

Sensor Pin	**ESP32-S3 Pin**
Tilt Sensor DO	GPIO1
Battery Monitor	GPIO2
Light Sensor DO	GPIO3
HX711 CLK	GPIO6
HX711 DAT	GPIO43
SX1262 SCK	GPIO7
SX1262 MISO	GPIO8
SX1262 MOSI	GPIO9

Table 2: Sensor-to-ESP32-S3 Pin Mapping

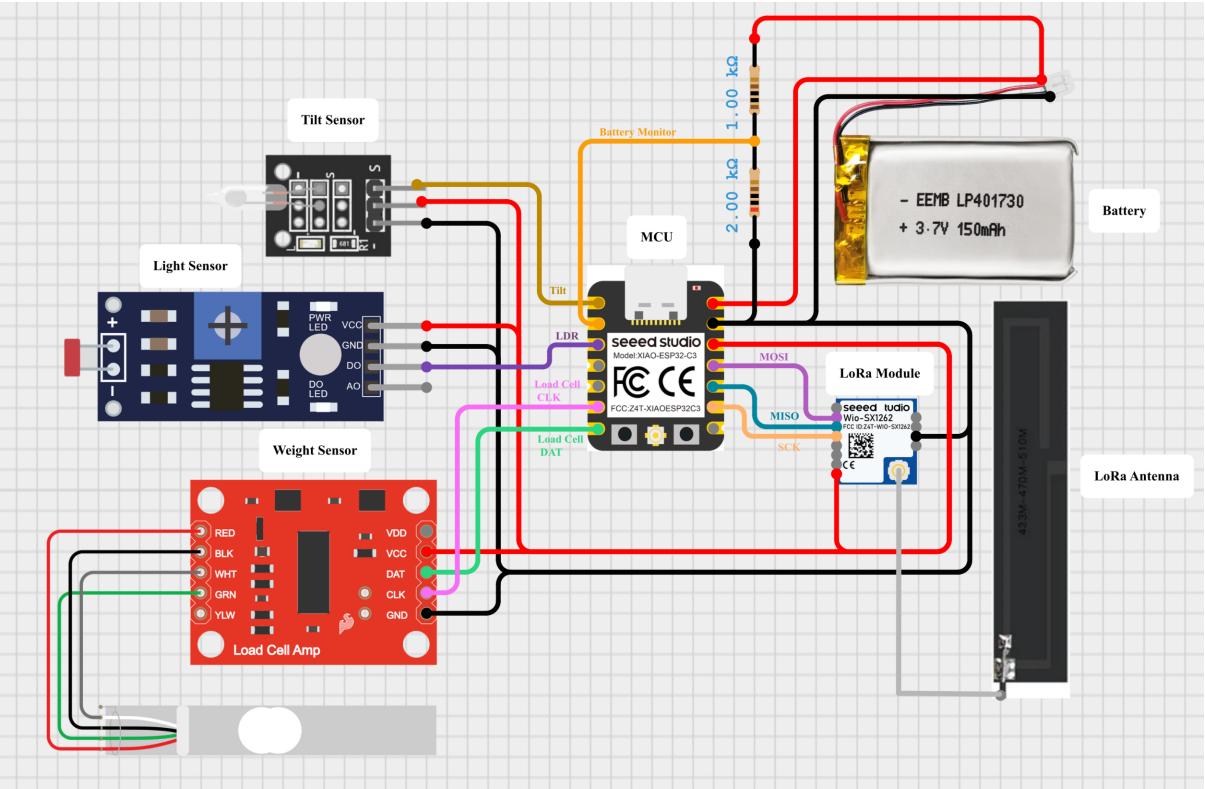


Figure 7: Schematic Diagram of the System

3.3 Embedded Software Layer

3.3.1 System Flow

The complete logical flow of the system is illustrated in 8.

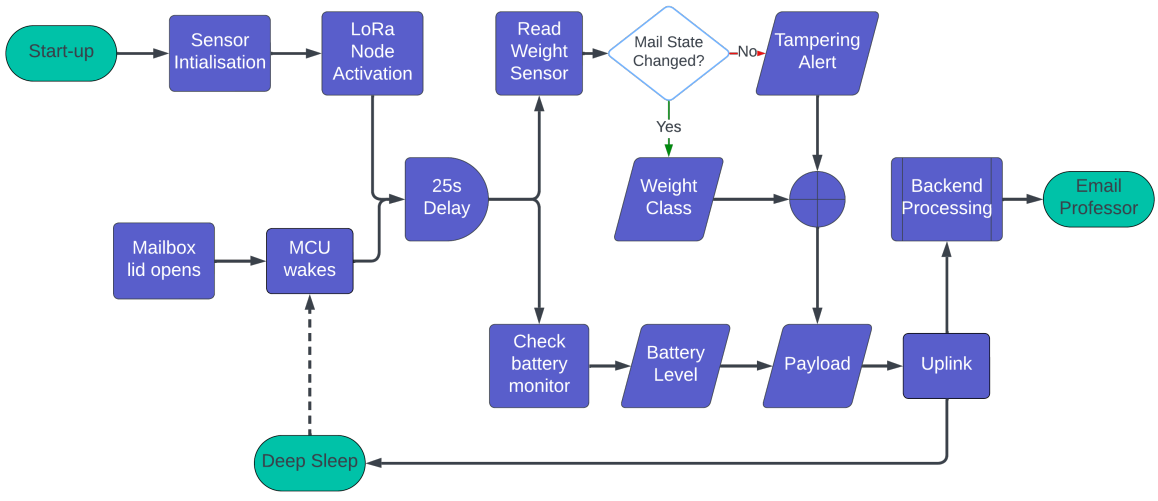


Figure 8: Programming Flow Chart of the Embedded Software

The general idea is that upon start-up, the sensors are initialised, the LoRaWAN node is activated, the mail detection logic is executed to build the payload, the payload is transmitted and the controller is put to sleep. Once the light or tilt sensor is triggered, the ESP32-S3 wakes up and the process repeats.

The actual execution of this flow is implemented in the **LR-V1.0.ino** as shown in Listing 1. All of the functions used are defined and explained in the subsequent sections.

```
#include "LoRaConfig.h"
#include "SensorManager.h"
// Start Here upon Wake-up
void setup() {
  Serial.begin(115200);

  // Initialising all the Sensors
  sensorManager_init();

  // Radio Setup
  Serial.println(F("\nSetup... "));

  ...

  // Activating the node for LoRaWAN
  state = lwActivate();
  Serial.println(F("Ready!\n")); // Ready to begin uploading

  // Reading Sensors and Building in Payload
  uplinkPayloadLen = detectMail(uplinkPayload);

  // Uploading the Payload
  state = lwUplink(uplinkPayload);

  delay(5000);

  // Sleep
  radio.sleep();
  goToSleep();
}
void loop() {
  // Empty since sleep interrupt operates as the loop
}
```

Listing 1: Main Code in LR-V1.0.ino

3.3.2 Microcontroller Code Structure

The microcontroller code was structured with modularity in mind and divided into four files:

- **SensorManager.cpp:** This file contains all the functions related to sensor initialization, reading sensor values, and processing the sensor data.
- **SensorManager.h:** This is the header file for SensorManager.cpp, containing function prototypes and necessary libraries and macros.
- **LoRaConfig.h:** This file handles the LoRaWAN communication, initializing required parameters, including libraries, defining functions for LoRa activation and upload.
- **LR-V1.0.ino:** This is the main file which essentially just runs the functions defined in the preceding files in the specific order required.

To program the ESP32-S3, the Arduino IDE was used. The complete code can be found in the project repository (or in the attached zip file). Additionally, all libraries required are also included in the repository for ease of access. Simply click on File > Preferences and change the sketchbook location to the path of the project repository. The ESP32 board manager must also be installed following the instructions from Express-If Documentation. Once installed, select "XIAO_ESP32S3" from Tools > Board menu.

3.3.3 Sensor Management

To initiliase the pins and parameters of the sensors, the `sensorManager_init()` function was defined as shown in Listing 2 from the **SensorManager.h** file.

```
void sensorManager_init(){
  // Weight Sensor
  scale.begin(LOADCELL_DOUT_PIN, LOADCELL_SCK_PIN); // initiliase weight sensor
  scale.set_scale(883.f);                           // Set the default (
      ↪ calibrated against scale)
  scale.power_down();                               // Put HX711 to sleep to
      ↪ save power
  delay(2000);
  SensorReading weightReading = readWeightSensor();
  weightState_prev = weightReading.value; // initial weight state

  // Light Sensor
  pinMode(LIGHT_SENSOR_PIN, INPUT); // Input mode for light sensor

  // Tilt Sensor
  pinMode(TILT_SENSOR_PIN, INPUT); // Input mode for tilt sensor

  // Setting up ADC for Battery Monitor
  analogReadResolution(12);       // 0 to 4095
  analogSetAttenuation(ADC_11db); // Allows full 0 to 3.3V range

  // End Intialisation
  Serial.println("Sensor Manager Initialized.");
}
```

Listing 2: sensorManager_init() Function

Importantly, this function sets the scale factor for the load cell after calibration and configures the ADC for battery monitoring. The process of calibration is discussed in Section 3.3.4. The ADC is set to 12-bit resolution to allow readings from 0 to 4095, and the attenuation is set to 11dB to allow the full voltage range of 0 to 3.3V to be read instead of the default 1.1V. The function also puts the HX711 load cell amplifier to sleep after initialization to save power.

Additionally, to keep the sensor readings organised, a struct called `SensorReading` was created. This struct contains the value read from the sensor as well as an identifying number. This is useful for logging and debugging purposes. The struct is defined in Listing 3 from the **SensorManager.h** file.

```
struct SensorReading
{
    uint8_t id;    // Unique for each sensor ID
    uint8_t value; // The actual value
};
```

Listing 3: sensorReading struct

3.3.4 Weight Sensor Implementation

To use the HX711 load cell amplifier, the HX711 library (V0.7.5) by Bogdan Necula was used. This library provides an easy to use interface for reading values from the load cell.

Before using the load cell, it must be calibrated. This was done by weighing an object with a calibrated scale and then placing it on the load cell. The ESP32-S3 was programmed to tare the scale and then call the function `scale.get_units(10)` to get the average of 10 readings from the load cell. This value is then divided by the known weight to get the scale factor. After multiple iterations, a scale factor of 883.f was found to be accurate for this setup. This value was then set during initialization as shown in Listing 2.

The actual reading of the weight sensor is done in the `readWeightSensor()` function defined in the **SensorManager.cpp** file. The function first wakes up the HX711 from sleep mode, reads the weight value and compares against predefined thresholds to determine if a light, medium, or heavy package is present. The most important line is where the load cell value is read as shown in Listing 4. The average of 10 values are taken and normalised by subtracting the tare value of 270 (the reading when no weight is present). This was done instead of the `scale.tare()` function from the HX711 library because running the tare function would require the user to ensure no weight is present either upon startup/wake-up or before every reading which is not feasible for this application. While this is certainly possible during initial installation and start up, every time the ESP32 wakes from deep sleep it would tare the load cell and thus ignore any mail which was placed while the device was asleep.

```
SensorReading readWeightSensor() {
  scale.power_up(); // Wake HX711 from sleep
  ...

  if (scale.is_ready()) {
    weight = scale.get_units(10) - 270; // Normalised Average of 10 readings (
        ↪ based on calibration)
    ...
  } else {
    Serial.println("HX711 not found.");
  }

  reading.value = (uint8_t)weightState; // storing state value

  scale.power_down(); // Put HX711 to sleep to save power
  delay(500);
  return reading;
}
```

Listing 4: Excerpt ofreadWeightSensor() Function

3.3.5 Battery Monitor Implementation

A battery monitor was implemented using the `batteryMonitor()` function defined in the **SensorManager.cpp** file and shown in Listing 5. From Figure 7, we can see a voltage divider is used because the maximum battery voltage of 4.2V, which exceeds the maximum input voltage of the ESP32-S3 (3.3V). This is compensated for in the code by the multiplication of the `adcVoltage` by the voltage divider ratio, which in this case is 1.5. The function then maps the voltage linearly to a percentage based on the operating voltage range of 3.1V (0%) to 4.2V (100%) to give the user an easily readable indication of the battery level.

```
SensorReading batteryMonitor() {
  SensorReading reading;
  uint8_t batPercentage;
  reading.id = ID_BAT_MONITOR;
  int adcValue = analogRead(BAT_MONITOR_PIN);
  float adcVoltage = (adcValue * 3.3) / 4095; // Calculating U[V] at the pin
  float batVoltage = adcVoltage * 1.5; // Using ratio to find actual U[V]
  if (batVoltage >= 4.20) { //4.20V ^= 100%
    reading.value = 100;
  } else if (batVoltage <= 3.10) { //3.10V ^= 0%
    reading.value = 0;
  } else { // Mapping to percentage
    batPercentage = (batVoltage - 3.10) * 100 / (1.1);
    reading.value = batPercentage;
  }
  return reading;
}
```

Listing 5: batteryMonitor() Function

3.3.6 Mail Detection logic

With all the sensors setup, detecting the state of the mailbox is handled by the `detectMail()` function found in **SensorManager.cpp**. The function first reads the weight sensor to determine if there is mail present, then checks for tampering by comparing the current weight state with the previous one. If no change is detected, it indicates that the lid was opened, but no mail was added or removed which by our logic means someone is looking into the mailbox for potentially nefarious reasons. If a change is detected, the function encodes the type of mail detected based on the weight state from the `readWeightSensor()` function. The function's argument is a pointer to a payload buffer where the encoded data is stored for transmission via LoRaWAN. Thus the actual encoding process is as simple as adding the relevant information into the buffer with a line like `payloadBuffer[len++] = ID;`.

One significant feature of this function is a 25 second delay implemented before the weight sensor is read to ensure the mail person has sufficient time to place all the mail onto the platform. This was added after testing showed that the load cell could miss mail events if not placed quickly enough as discussed in Section 4.3.2.

3.3.7 Wake-up and Sleep Logic

In an effort to save as much power as possible, the ESP32-S3 is put into sleep mode after each mail detection session and transmission. This is done using the function `goToSleep()` defined in **SensorManager.cpp** and shown in Listing **??**. This function configures the wake-up sources to be the light sensor and tilt sensor pins, such that when either of these pins goes low (indicating the mailbox lid is opened or light is detected inside the mailbox), the ESP32 will wake up from deep sleep.

```
void goToSleep() {
  Serial.println("Going to deep sleep...");

  // Configure Wake Interrupt
  esp_sleep_enable_ext1_wakeup((1ULL << LIGHT_SENSOR_PIN) | (1ULL <<
      ↪ TILT_SENSOR_PIN), ESP_EXT1_WAKEUP_ANY_LOW);

  // Sleep
  Serial.flush();
  esp_deep_sleep_start();
}
```

Listing 6: goToSleep() Function

3.3.8 LoRaWAN Configuration

All of the configuration and functions required for LoRaWAN communication are defined in the **LoRaConfig.h** file. The libraries required are RadioLib (V7.4.0) by Jan Gromes and Preferences (installed in Arduino IDE by default). The configuration process was also completed with guidance from Seeed Studio Wiki (2026).

The first and most important step is defining the access keys for The Things Network (TTN). Setting up a device on TTN is discussed in Section 3.4.1, but once the device is created, the keys can be found in the console. These keys were first defined as macros for easy access and then copied into variables for use as shown in Listing 7.

```
    // TTN Keys for Accessing the Network
    #ifndef RADIOLIB_LORAWAN_JOIN_EUI // All zeros for TTN compliance
    #define RADIOLIB_LORAWAN_JOIN_EUI 0x0000000000000000
    #endif

    #ifndef RADIOLIB_LORAWAN_DEV_EUI
    #define RADIOLIB_LORAWAN_DEV_EUI 0x070B3D57ED0074D0
    #endif

    #ifndef RADIOLIB_LORAWAN_APP_KEY
    #define RADIOLIB_LORAWAN_APP_KEY 0x02, 0xBC, 0xE7, 0xC1, 0x02, 0xB3, 0x18,
        ↪ 0xD7, 0x02, 0xF2, 0x18, 0xDF, 0x9E, 0x45, 0xD1, 0x8E
    #endif

    ...

    // Copy over Keys
    uint64_t joinEUI = RADIOLIB_LORAWAN_JOIN_EUI;
    uint64_t devEUI = RADIOLIB_LORAWAN_DEV_EUI;
    uint8_t appKey[] = {RADIOLIB_LORAWAN_APP_KEY};
```

Listing 7: TTN Key Definition

After the keys are set, the LoRa node is activated using the function `lwActivate()` found in the **LoRaConfig.h** file. The Over the Air Activation (OTAA) method is used instead of the Activation By Personalization (ABP) method for better security and flexibility. This is done using the RadioLib command `node.activateOTAA()` as shown in Listing 8. Because this process sometimes fails due to network issues or recognition issues on the TTN side, a loop is implemented to retry joining until successful.

```
while (state != RADIOLIB_LORAWAN_NEW_SESSION) {
    Serial.println(F("Joining LoRaWAN fresh..."));
    state = node.activateOTAA();

    // Storing Nonces
    uint8_t *persist = node.getBufferNonces();
    memcpy(lwNonces, persist, RADIOLIB_LORAWAN_NONCES_BUF_SIZE);
    store.putBytes("nonces", lwNonces, RADIOLIB_LORAWAN_NONCES_BUF_SIZE);

    if (state == RADIOLIB_LORAWAN_NEW_SESSION)
        break;

    // Fail and Retry
    Serial.print(F("Join failed: "));
    Serial.println(state);
    delay(5000); // Retry every 5s
}
Serial.println(F("Join successful"));
```

Listing 8: TTN Joining Loop in lwActivate()

To make re-joining faster on subsequent wake-ups and power cycles, the nonces are stored in the ESP32's non-volatile memory (NVM) using the Preferences library as shown in Listing 8. Once the nonces are restored, the node attempts to re-join in a loop until successful, after which the new nonces are stored. This solved the major rejoining issue discussed in Section 4.2.1. Additionally, the session is also stored in the RTC memory allowing for even faster re-joining after deep sleep wake-ups. This is done because storing sessions indefinitely within the NVM may lead to unnecessary wear on the memory. This re-joining process is shown in Listing 9. Notice that the program checks if restoring the session is possible first before looping through the nonces. This solution was adapted from example code from radiolib-org (2025).

```
    // Restore nonces from NVM
    store.getBytes("nonces", lwNonces, RADIOLIB_LORAWAN_NONCES_BUF_SIZE);
    state = node.setBufferNonces(lwNonces);
    debug(state != RADIOLIB_ERR_NONE, F("Restoring nonces failed"), state,
        ↪ false);

    // Restore session from RTC
    state = node.setBufferSession(lwSession);

    // Joining TTN (if restore not immediate)
    while (state != RADIOLIB_LORAWAN_NEW_SESSION) {
        Serial.println(F("Joining LoRaWAN..."));
        state = node.activateOTAA();
        if (state == RADIOLIB_LORAWAN_SESSION_RESTORED) {
            Serial.println(F("LoRaWAN session restored"));
            break;
        } else if (state == RADIOLIB_LORAWAN_NEW_SESSION) {
            Serial.println(F("Join successful, saving nonces"));
            uint8_t *persist = node.getBufferNonces();
            Serial.print("Dev Nonces: ");
            Serial.println((char *)persist);
            memcpy(lwNonces, persist, RADIOLIB_LORAWAN_NONCES_BUF_SIZE);
            store.putBytes("nonces", lwNonces, RADIOLIB_LORAWAN_NONCES_BUF_SIZE
                ↪ );
            break;
        }
        Serial.print(F("Join failed: "));
        Serial.println(state);
        delay(5000); // Retry every 5s
    }
```

Listing 9: TTN Re-Joining Loop using Stored Nonces or Session in TTN Joining Loop in lwActivate()

3.3.9 LoRaWAN Data Transmission

The actual transmission of data via LoRaWAN is handled by the `lwUplink()` function defined in the **LoRaConfig.h** file. This function transmits the payload encoded in the `detectMail()` function using the RadioLib command `node.sendReceive(uplinkPayload, uplinkPayloadLen)`. Similar to in the joining process in in Listing 8, the function loops to until the data is successfully sent.

3.4 Transport Layer

3.4.1 TTN Configuration

To set up the device on The Things Network (TTN), first an account was created on The Things Network. After logging in, a new application and end-device were created. The end-device was configured with a Join EUI of all zeros to comply with TTN's non-commercial regulations and the Europe 868.1 MHz frequency plan since the device is being developed in Germany. Most importantly, the LoRaWAN specification was set to 1.0.4 as it is the most up-to-date and RadioLib V7.4.0 is designed to comply with the specifications.

3.5 Application Layer

This section covers the implementation of the backend server and application layer, which is responsible for receiving data from The Things Network, decoding the payload, storing the data, and providing a user interface for monitoring the mailbox status. The link with the The Things Network is done using webhooks, which send HTTP POST requests to the python server whenever new data is received from the LoRaWAN device.

3.5.1 Backend Server Setup

To set up the backend server, we require three files, which include: 'server.py', '.env', and 'requirements.txt'. Each file servers a specific purpose:

- **server.py:** This is the main server file that contains the Flask application code. It handles incoming requests, decodes the payload, updates the mailbox status, and serves the user interface.
- **.env:** This file contains environment variables such as server port, authentication token, email sender credentials, and recipient email addresses. This allows for easy configuration without hardcoding sensitive information in the code.
- **requirements.txt:** This file lists all the Python libraries required to run the server.

These all need to be placed in the same directory for the server to function correctly. The server can then be started by running the command 'python server.py' in the terminal. The dependencies, to be added to requirements.txt as shown in Listing 10, can be installed using the command 'pip install -r requirements.txt'.

```
Flask==2.3.3
python-dotenv==1.0.0
```

Listing 10: requirements.txt file contents

These can be installed directly using pip as well, however using requirements.txt makes it easier and more future proof. These steps can be also be avoided, and the bash script from Section 4.2.2 be used directly to install the required libraries if they are missing.

Next a random authentication token must be generated, for Windows 11, the command shown in Listing 11 can be used.

```
$bytes = New-Object byte[] 16; (New-Object System.Security.Cryptography.
    ↪ RNGCryptoServiceProvider).GetBytes($bytes);
[System.BitConverter]::ToString($bytes) -replace '-'
```

Listing 11: Generate random auth token command

Now you can open the .env file using a text editor of your choice (here edit by MS was used). The following variables must be added as shown in Listing 12:

```
export SERVER_PORT=3000
export AUTH_TOKEN=<token from above>

EMAIL_SENDER=your_email@gmail.com
EMAIL_PASSWORD=your_app_password_here
```

Listing 12: .env file contents

The main contents of server.py are explained in the following sections and full code can be found in the project repository.

3.5.2 Payload Decoder

The LoRaWAN is able to send the bits to The Things Network. However for these to be actually useful to the user they must be decoded and used to represent relevant information for a user, this includes the mail status, the battery and which post box it is. For this first a payload decoder must be made. This is made keeping in mind how bits were encoded in the first place. The decoder can be seen in Listing 13

```
def decode_mailbox_data(base64_string):
    try:
        raw_bytes = base64.b64decode(base64_string)
        if len(raw_bytes) < 4:
            return None, "Error: Short Data", "red", None

        # 1. Decode ID (Hex to Int logic)
        try:
```

```
            device_id = int(f"{raw_bytes[0]:x}")
        except:
            device_id = raw_bytes[0]

        state_byte = raw_bytes[1]
        value_id = raw_bytes[2]
        val = raw_bytes[3]

        # 2. Decode Status (Aligned with new JS Decoder)
        if state_byte == 0x04:
            status, color = "Tampering Alert", "red"
        elif state_byte == 0x05:
            status, color = "Heavy Mail", "#004d40"
        elif state_byte == 0x06:
            status, color = "Medium Mail", "#00897b"
        elif state_byte == 0x07:
            status, color = "Light Mail", "#4db6ac"
        elif state_byte == 0x08:
            status, color = "No Mail", "blue"
        else:
            status, color = f"Unknown: {hex(state_byte)}", "orange"

        # 3. Decode Battery (valueID 0x09)
        battery = int((val/64)*100) if value_id == 0x09 else None
        # If battery, map it correct linear level (64 is 100%)
        return device_id, status, color, battery

    except Exception as e:
        logger.error(f"Decoding error: {e}")
        return None, "Decoding Failed", "black", None
```

Listing 13: Payload Decoder Function

This allows us to correctly identify if a heavy, medium or light package was detected. This information can then be used to update the website to represent the appropriate information and also be included in the mail sent to the user.
The key parts of the decoder are explained below:

- ”state_byte” is used to determine the mail status, this is done by checking the value of the byte and mapping it to the appropriate status message.
- There is return at the start for robustness, in case the payload is too short or empty.
- ”value_id” has been used as a form of future proofing, in case more values are added to the payload in the future.

For actual mapping of device ID to device name and email addresses, a dictionary is used as shown in Listing 14.

```
# --- Device Mapping ---
DEVICE_NAMES = {
    33: "PostBox SAN"
    # Add more boxes here
}
# --- Email Mapping ---
DEVICE_RECIPIENTS = {
    33: "email1@gmail.com, email2@gmail.com"
    # Add more emails here
}
```

Listing 14: Device mapping method

More can be added as just new rows with the similar syntax just a comma at the end of all rows except the last one. Multiple email addresses can also be added by separating them with a comma within the double quotes.

3.5.3 Frontend Implementation

For the actual server, which the user interacts, with a python server was created. This server uses the Flask framework to create a simple web application that displays the status of the mailbox. The server listens for incoming data from The Things Network and updates the mailbox status accordingly. The relevant code snippet is shown in Listing 15

```
@app.route('/', methods=['GET'])
def show_dashboard():
    d = dashboard_data
    html = """
    <!DOCTYPE html>
    <html>
    <head>
        <title>Mailbox Monitor</title>
        <meta http-equiv="refresh" content="5">
        <style>
            body {
                font-family: 'Segoe UI', sans-serif;
                text-align: center;
                padding: 40px;
                background-color: #f0f2f5; }
            .card {
                background: white;
                padding: 40px;
                border-radius: 15px;
                display: inline-block;
                box-shadow: 0 4px 12px rgba(0,0,0,0.1);
                width: 450px; }
            .status-box {
                font-size: 32px;
                font-weight: bold;
                margin: 20px 0;
                padding: 20px;
                color: white;
                border-radius: 10px;
                background-color: {{ d.status_color }}; }
            .battery-indicator {
                font-weight: bold;
                font-size: 18px;
                padding: 5px 15px;
                border-radius: 20px;
                display: inline-block;
                color: white;
                background-color: {{ d.battery_color }}; }
            .meta {
                color: #888;
                font-size: 13px;
                margin-top: 15px; }
            table {
                width: 100%;
                border-collapse:
                collapse;
                margin-top: 20px;
                text-align: left; }
            th, td {
                padding: 10px;
                border-bottom: 1px
                solid #eee;
                font-size: 14px; }
            .dot {
                height: 10px;
                width: 10px;
                border-radius: 50%;
```

```
                display: inline-block;
                margin-right: 5px; }
        </style>
    </head>
    <body>
        <div class="card">
            <h1>Smart Mailbox</h1>
            <h2 style="color:#666">{{ d.device_name }}</h2>
            <div class="status-box">{{ d.status_text }}</div>
            <div class="battery-indicator">{{ d.battery_level }}</div>
            <p class="meta">Last Update: {{ d.timestamp }}</p>
            <h3>Recent Activity</h3>
            <table>
                <tr><th>Time</th><th>Event</th><th>Bat</th></tr>
                {% for event in d.history %}
                <tr>
                    <td>{{ event.time }}</td>
                    <td>
                        <span class="dot"
                            style="background-color: {{ event.color }};">
                        </span>{{ event.status }}</td>
                    <td>{{ event.battery }}</td>
                </tr>
                {% endfor %}
            </table>
        </div>
    </body>
    </html>
    """
    return render_template_string(html, d=d)
```

Listing 15: Flask Server Code Snippet

The final website can be run by just running the bash file and afterwards going to this link: **Smart Mailbox Dashboard**

3.6 Validation Methods

During the development of the system, it became essential to validate system at different stages. Components were tested against the requirements outlined in Sections 1.4.1 and 1.4.2.
The following briefly describes the validation methods used for the main requirements:

- **Mail Detection:** Various objects ranging from around 20 g to above 100 g were weighed on a scale and then placed on the load cell platform to check if the load cell could detect the presence of mail and classify them accurately. 10 different objects were used for testing including letters, pieces of cardboard, and power bank. The setup for this test is shown in Figure 9.
- **Email Notification:** The email notification system was tested by simulating mail detection events and verifying that the mail was detected and emails were sent to the designated recipient. This was tested with the system installed in the mailbox as shown in Figure 11. The test was conducted with 3 different objects. For each object, placing inside, removing and opening the lid without placement were tested.
- **Battery Life:** The system was powered by the battery and current consumption was monitored while idle and while sending messages. The readings were used to estimate the battery life. The experimental setup for this test is shown in Figure 10.

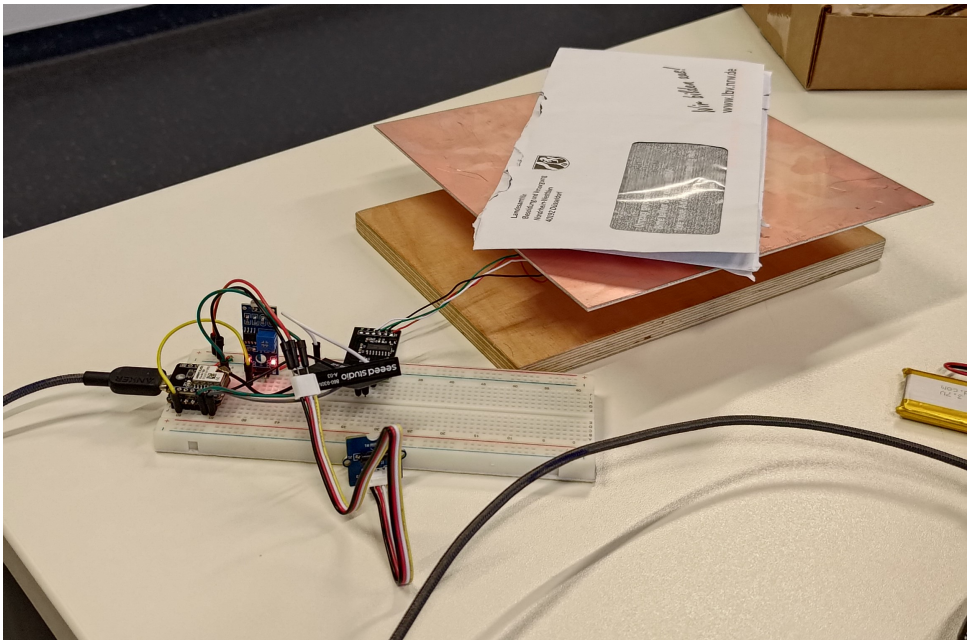


Figure 9: Experimental Setup for Mail Detection Testing Using Various Objects of Different Weights.

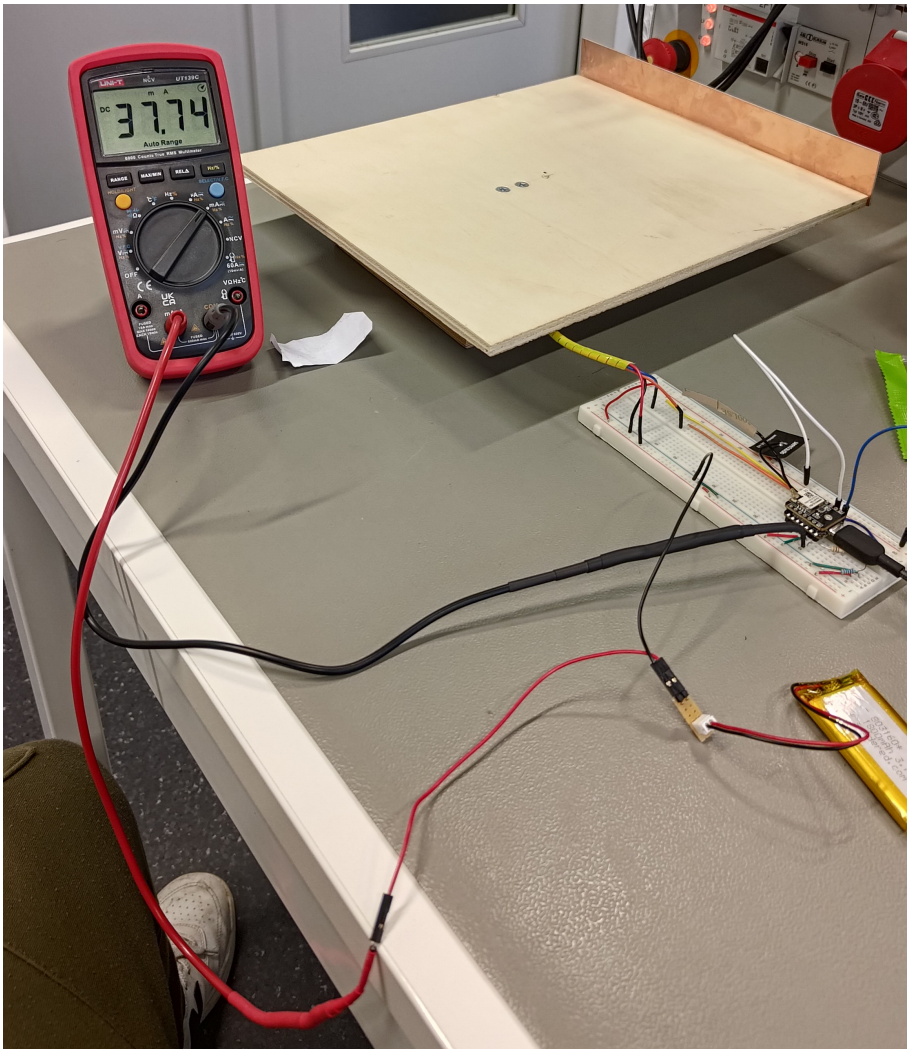
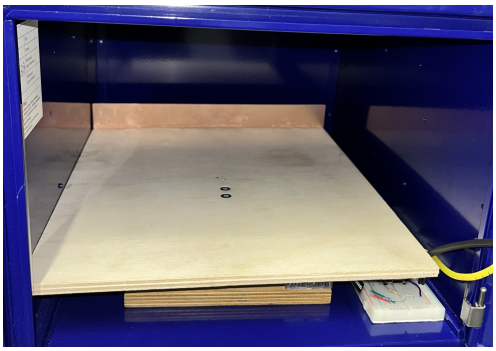


Figure 10: Experimental Setup for Battery Life Estimation by Measuring Current Consumption.

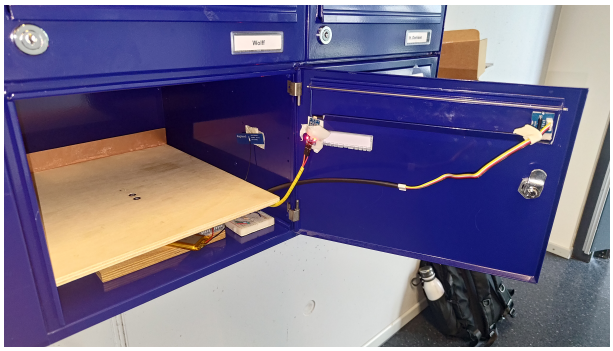
4 Results

4.1 Complete System

The completely assembled system can be seen in Figure 11. A short demonstration video of the system in action can be found at the following link: `https://youtube.com/shorts/1nNW60RcUi4`.



(a) Front View



(b) Side View

Figure 11: Final Product Inside the Post Box

4.2 Issues

4.2.1 TTN Re-join Issue

The most significant issue faced during the project was with The Things Network (TTN) and the LoRaWAN re-join process. After successfully joining the network for the first time, whenever the device was put to sleep or restarted, it took increasingly longer and longer to re-join the network, sometimes taking upwards of 30 minutes. The error message in the TTN console simply said `Devnonces too small`. This was a major hindrance to testing and development, as every time the code was modified and re-uploaded, the device had to re-join the network.

Through extensive research, a number of potential solutions were attempted before the issue was resolved. Firstly, the LoRaWAN specification in the TTN configuration was changed from 1.1.3 to 1.0.4 as this form has more relaxed security requirements. This only resolve the issue for a few power cycles. Next, the RadioLib library was upgraded to the latest version, which required a reworking of most of the LoRaWAN configuration code. This also did not resolve the issue. Finally, it was discovered that the issue was due to the DevNonce value being reset to 0 on every power cycle. This was fixed by storing the DevNonce value in the non-volatile memory of the microcontroller and incrementing it on every join attempt. This solution finally resolved the issue, allowing for quick re-joining on subsequent power cycles as discussed in Section 3.3.8.

4.2.2 Issues with static IP and hosting

The server was initally hosted locally, however to provide access to the user over the web, a public IP address was required. This was also meant to be static, so user can always access the server without having to worry about changing IP addresses. This is what led to the use of ngrok, which allows for a static IP address to be used. However, this meant the user had to run multiple scripts to start the server. Which can be annoying for a non technical user.

To combat this issue, a .bat file was created, which runs all the necessary commands to start the server and ngrok tunnel. This allows the user to just double click the .bat file and have everything start automatically. This also helps us solve another issue, which is incase the required python libraries are not installed, the script checks for them and installs them if they are missing. It also hides all ngrok instances, to prevent confusion. The bat script contents can be seen in Listing 16

```
1 @echo off
2 title Smart Mailbox Dashboard
```

```
color 0A

echo ======================================================
echo           SMART MAILBOX PROJECT LAUNCHER
echo ======================================================
echo.

:: --- STEP 1: INSTALL/UPDATE REQUIREMENTS ---
echo [1/4] Checking Python libraries...
pip install flask python-dotenv
if %errorlevel% neq 0 (
    color 0C
    echo.
    echo [ERROR] Python or PIP is not installed or not in your PATH.
    echo Please install Python from python.org and try again.
    pause
    exit /b
)
echo Libraries are ready.
echo.

:: --- STEP 2: CHECK CONFIGURATION ---
echo [2/4] Checking configuration file...
if not exist .env (
    color 0E
    echo.
    echo [WARNING] .env file was not found!
    echo I have created a template .env file for you.
    echo Please open ".env", add your passwords, and run this script again.

    :: Create a default .env file
    echo SERVER_PORT=3000> .env
    echo AUTH_TOKEN=BC5FB17A739C64639751B59209E07F88>> .env
    echo EMAIL_SENDER=my-iot-project@gmail.com>> .env
    echo EMAIL_PASSWORD=REPLACE_WITH_APP_PASSWORD>> .env
    echo EMAIL_RECIPIENT=your_personal_email@gmail.com>> .env

    pause
    exit /b
)
echo Configuration found.
echo.

:: --- STEP 3: START NGROK (NEW WINDOW) ---
echo [3/4] Launching Ngrok Tunnel...
:: This opens a separate popup window for Ngrok so it doesn't block the script
start "Ngrok Tunnel" ngrok http --domain=unarithmetically-peppiest-libbie.ngrok
    ↪ -free.dev 3000

:: --- STEP 4: START PYTHON SERVER ---
echo [4/4] Starting Python Server...
echo.
echo ======================================================
echo    Dashboard: https://unarithmetically-peppiest-libbie.ngrok-free.dev
echo    Local:     http://localhost:3000
echo    Status:    RUNNING (Keep this window open)
echo ======================================================
echo.

python server.py

:: If python crashes, keep window open to see error
pause
```

Listing 16: .bat Script to start server and ngrok

However, this is still not an ideal solution, as the user still has to run the .bat file manually, must have python installed on their machine and most importantly run the server continuously. A better solution would be to host the server on a local raspberry pi or similar device, which can run the server 24/7 without any user intervention. This would also eliminate the need for ngrok, as the raspberry pi can be given a static IP address.

4.3 Validation Results

The results from the validation tests described in Section 3.6 are summarized below.

4.3.1 Mail Detection

The system was able to successfully detect 10 out of 10 objects, resulting in a detection accuracy of 100%. Classification accuracy on the other hand was 80%, with 8 out of 10 objects being correctly classified into light, medium, and heavy categories. The misclassifications occurred for a stack of papers and a pack of cards. Both objects had weights close to the classification thresholds, leading to incorrect categorization.

4.3.2 Email Notification

The system was able to successfully send email notifications in all 9 test cases. However, events were only classified accurately 7 out of 9 times. The 2 misclassifications were no mail notifications being sent when mail was placed inside. This was due to the weight being read before the mail landed on the platform. To compensate this issue, a delay of 25 seconds was added before reading the weight after mail detection as mentioned in Section 3.3.6. It was also found that the entire mail detection event from lid opening to email receipt took approximately 65 seconds on average.

4.3.3 Battery Life

It was found that during an uplink event, the system drew a peak current of 39.5mA on average. While in deep sleep mode, the system drew around 7.25mA on average. This is likely due to the power consumption of the sensors. Using these values, the average current consumption over a 24 hour period can be calculated using the equations below. Assuming 4 mail events per day, each taking 65 seconds, and a battery capacity of 1800mAh, the estimated battery life is approximately 10 days.

$$I_{awake}^{avg} = \frac{4 \cdot 39.5mA \cdot 65s}{24h \cdot 3600\frac{s}{h}} = 118.86\mu A \tag{1}$$

$$I_{sleep}^{avg} = \frac{(24h \cdot 3600\frac{s}{h} - 65s \cdot 4) \cdot 7.25mA}{24h \cdot 3600\frac{s}{h}} = 7.23mA \tag{2}$$

$$T = \frac{1800mAh}{7.23mA + 0.11886mA} = 244.94h \approx 10days \tag{3}$$

Interestingly, the battery life of the system can be very easily extended by using any power bank and a USB-C cable. This is because the XIAO ESP32-S3 has an on board USB-C port which allows for convenient powering. Also, most commercially power banks have a much larger capacity than the 1800mAh battery used in this project. This can be seen in Figure 12, where the system is powered by a power bank. Using a power bank with a capacity of 10000mAh, the estimated battery life extends to approximately 55 days.

Figure 12: Final Product Powered by a Power Bank

5 Discussion

As demonstrated in Section 4.1, The Talking Mailbox successfully fulfils all functional and technical requirements defined in Section 1.4. The completed system reliably detects mailbox opening events, determines the presence of mail, categorize the current amount of mail into weight class, and communicates this information remotely via LoRaWAN. The results confirm that the proposed design is both technically feasible and effective within the intended indoor deployment environment.

A key outcome of this project is the system's ability to detect the current state of the mailbox rather than merely inferring activity. Unlike many of the smart mailbox solutions discussed in 1.2 that rely solely on door movement or motion detection, The Talking Mailbox directly measures mail presence using a load cell. This enables the system to distinguish between meaningful events (mail added or removed) and non-meaningful events (lid opened without delivery), thereby reducing false notifications. The use of the tilt switch and light-dependent resistor in addition to load cell also proved effective in increasing system robustness. It essentially uses the same door motion logic as the InstaView (2024), X-Sense (2025) and Notific (2025) along with a LDR as a redundancy. This makes the system even more reliable in detection.

Another significant distinction is the system's communication architecture. While the InstaView (2024) and X-Sense (2025), require base stations that connect to the internet for them, The Talking Mailbox connects directly to LoRaWAN infrastructure. This reduces installation complexity and allows for simple communication deployment without additional hardware in the user's home or office. However, this LoRaWAN communication protocol requires existing gateways to exist in the user's area; a short-coming the Notific (2025) also faces. Only the Fouvin (2025) which connects directly to Wifi over come this issue of additional infrastructure as it can be assumed that most users have Wifi already.

Despite its successful operation, the system does present some limitations. The load cell and HX711 sensor requires calibration for each installation as each platform constructed maybe have slight differences in weight and orientation. This of course increases the installation complexity. The load cell may also experience long term drift and may require maintenance. The aforementioned LoRaWAN communication has the issue of relying on existing infrastructure which is dependent on the area. Users can certainly install a LoRaWAN gateway themselves, however this renders the communication solution no better than existing products. Finally, the current backend implementation requires a continuously running server, which introduces deployment complexity for non-technical users. This however is easily fixed as the employment of a Datacake dashboard handles all of the hosting and just presents the user with an aesthetic user interface.

6 Conclusion

6.1 Project Summary

Overall, the results validate the original design objectives of The Talking Mailbox and demonstrate a clear improvement over existing smart mailbox solutions. By combining direct mail detection, low-power operation, and long-range wireless communication, the project delivers a practical and novel solution to the problem of unattended mailboxes. The Talking Mailbox not only confirms the feasibility of weight-based mail detection but also establishes a foundation for future enhancements such as multi-mailbox management and tampering alerts.

6.2 Future Work / Improvements

While working on The Talking Mailbox project, several areas for potential improvements were identified that could enhance the functionality, reliability, and user experience of the system. These improvements include:

- **Better lid opening detection**: While the LDR and tilt sensor provide good and reliable detection of the lid opening, they can be replaced with a single reed switch. This would reduce complexity and power consumption.
- **Improved housing**: Housing of the components can be combined with load cell platform for a better fit and protection of the components.
- **Additional information**: A camera can be added to capture images of the mail inside the box. This would provide visual confirmation of mail presence and enhance user experience. An LED can be used in conjunction with the camera to provide illumination inside the mailbox even when closed.
- **Easier server access**: The server can be deployed on a local raspberry pi or similar device to allow for local access and control. This would eliminate the need for an external hosting service and provide more flexibility.
- **Tampering alert**: A buzzer can be added to sound an alarm when tampering is detected. This would enhance security.
- **Improved Battery Life**: Implementing low power design could improve battery life. This may include using a custom PCB with minimal components instead of development boards or integrating more low power components.
- **Energy harvesting**: A piezo-element can also be added to the lid to harvest energy from the opening and closing of the mailbox. This would extend battery life and reduce maintenance.
- **Expansion to multiple mailboxes**: The system can be expanded to support multiple mailboxes, serving multiple users from a single controller. With the existing device ID set-up in the code, this can be achieved simply by connecting multiple sensor configs to the MCU using multiplexers and expanding the payload by prefixing the unique device ID before the sensor data.

Acknowledgement

The development team would like to express sincere gratitude to Mr Friedrich Muhs for his guidance and aid throughout the project.

References

Fouvin. (2025). *Mailbox sensor with remote monitoring via tuya app.* Amazon. Retrieved from https://www.amazon.de/Bewegungssensor-Intelligenter-Bewegungsmelder-Heimsicherheit-Fern%C3%BCberwachungs/dp/B0DDGGDMPX/ (Accessed: 2025-12-29)

frankenfoamy. (2021). *Automatic mailbox alert.* YouTube. Retrieved from https://youtu.be/3E_R6cfH0M8?si=_cTojZNmOHxOoKMr (Accessed: 2025-12-29)

IDUINO. (n.d.). *Ball switch sensor module (se059).* (Datasheet)

InstaView. (2024). *Long-range mail arrival indicator device.* Amazon. Retrieved from https://www.amazon.com/InstaView-Delivered-Notification-Long-range-Indicator/dp/B0DPDRX6MG (Accessed: 2025-12-29)

Joy-IT. (n.d.). *Load cell with hx711 amplifier.* Retrieved from https://www.reichelt.com/de/en/shop/product/arduino_-_pressure_sensor_20_g_to_10_kg-284398 (User Manual and Datasheet)

Notific. (2025). *Smart mailbox sensor.* Notific.at. Retrieved from https://notific.at/en (Accessed: 2025-12-29)

radiolib-org. (2025). *Lorawan_esp32.ino — example for lorawan persistence on esp32.* GitHub repository. Retrieved from https://github.com/radiolib-org/radiolib-persistence/blob/main/examples/LoRaWAN_ESP32/LoRaWAN_ESP32.ino (Accessed: 2025-12-19)

Seeed Studio Wiki. (2026). *Lorawan sensor node.* Retrieved from https://wiki.seeedstudio.com/wio_sx1262_xiao_esp32s3_for_lora_sensor_node/ (Accessed: 2025-11-28)

X-Sense. (2025). *Smart briefkastensensor sma51.* Bigshopper. Retrieved from https://bigshopper.de/product/x-sense-smart-briefkastensensor-erfordert-sbs50-basisstation-funk-melder-mit-grosser-reichweite-fuer-briefkaesten-briefkasten-alarm-zugestellte-post-sma51-2941520710.htm? (Accessed: 2025-12-29)