**Bingo : Smart Waste management solutions**

1. **Introduction**

The BinGo system is an innovative waste management solution designed to optimise the collection process by monitoring the fill levels of dustbins in real-time. Traditional waste collection methods are often inefficient, leading to overfilled or half-empty bins, which can cause sanitation issues or wasted resources. BinGo addresses these challenges by using smart sensors to track bin fill levels and transmit this data wirelessly to a centralised system.

With this system, waste collection services can plan routes more efficiently, ensuring bins are emptied only when necessary. The mobile app provides a user-friendly interface where workers and drivers can view bin statuses, receive notifications when a bin is full, and navigate to bins that require immediate attention. This technology helps reduce fuel costs, labour, and environmental impact, while improving overall cleanliness in urban areas.

The BinGo system is inspired by the need to optimise waste collection processes, reduce environmental impact, and lower operational costs through real-time monitoring of bin fill levels, enabling smarter and more efficient waste management in urban areas.

1. **System Architecture**

The BinGo system consists of both hardware and software components that work together to provide a real-time waste management solution. This section outlines the key elements of the system's architecture.

**2.1.** **Hardware Components**

* · **Ultrasonic Sensors**: Two ultrasonic sensors are mounted on the lid of each dustbin to measure the fill levels. These sensors calculate the distance between the top of the waste and the lid, determining how full the bin is.
* · **Raspberry Pi Pico W**: This microcontroller is the central unit in the sensor box. It processes the data collected by the ultrasonic sensors and sends it to the receiver using a LoRa transmitter.
* · **LoRa Transmitter and Receiver**: LoRa (Long Range) technology is used to transmit data wirelessly over long distances with low power consumption. The transmitter, connected to the Raspberry Pi Pico W, sends the fill level data to the receiver, which is set up in a central location.
* · **Power Supply**: A portable battery powers the sensor box, ensuring that it can operate autonomously without external power sources.

**2.2.**  **Software Components**

* **Arduino (C++)**: The sensors and transmitters are programmed using Arduino C++, which controls the data collection and transmission processes. The code running on the Raspberry Pi Pico W gathers data from the ultrasonic sensors and sends it via LoRa.
* **Firebase**: Firebase acts as the backend for storing and managing bin data. It is responsible for storing the bin fill levels and updating the mobile app in real-time. It also supports user authentication and notifications.
* **Flutter Mobile App**: The BinGo mobile app is developed using Flutter, a cross-platform framework. The app displays real-time data on bin statuses, provides route navigation for waste collection staff, and sends notifications when a bin is full or approaching full capacity.

**2.3.** **Data Flow**

* **Data Collection**: The ultrasonic sensors measure the fill levels of the dustbins.
* **Data Transmission**: The Raspberry Pi Pico W processes this data and sends it to the receiver using LoRa technology.
* **Data Upload**: The receiver forwards the data to Firebase, where it is stored and made available to the mobile app.
* **Data Display**: The mobile app fetches data from Firebase, displaying the fill levels of all bins on a map. Notifications are triggered if any bin's fill level exceeds a certain threshold.

1. **How It Works**

The BinGo system operates by gathering real-time fill level data from dustbins, transmitting it wirelessly, and displaying it in a mobile app. This process involves four key stages:

**3.1.** **Data Collection**

Each dustbin is fitted with a sensor box containing two ultrasonic sensors. These sensors measure the distance between the bin lid and the waste, calculating the fill percentage. The Raspberry Pi Pico W processes the sensor data, ensuring it is accurate and free of noise or irregularities.

**3.2.**  **Wireless Data Transmission**

The processed data is transmitted from the sensor box to a central system via **LoRa** technology. The **LoRa transmitter** sends the fill level data wirelessly over long distances to the **LoRa receiver**, which is positioned centrally. This low-power, long-range communication ensures reliable data transfer without frequent battery replacements.

**3.3.** **Data Storage and Backend**

The LoRa receiver collects data from all sensor boxes and sends it to **Firebase**, where it is stored and managed. Firebase enables real-time synchronization of the bin data, making it accessible to users through the mobile app. Additionally, Firebase handles user authentication and manages push notifications based on bin statuses.

**3.4.** **Mobile App Interaction**

The **BinGo mobile app** allows waste collection workers to monitor bin statuses in real time:

* **Map Display**: The app shows bins on a map, color-coded based on their fill levels (e.g., red for full bins, yellow for near-full).
* **Notifications**: Workers receive alerts when a bin’s fill level exceeds a defined threshold.
* **Route Optimization**: The app offers route suggestions for efficient collection, prioritizing bins that need immediate attention.

1. **Hardware-**

**4.1. Components used:**

1. Microcontroller: Raspberry Pi Pico W

2. Sensor: HC-SR04 ultrasonic sensor module

3. Communication – LoRa SX1278 433MHz long range module

4. Lithium Polymer battery

A. **THE MICROCONTROLLER**-

The Raspberry Pi Pico W is an open-source microcontroller board based on the RP2040 chip developed by the Raspberry Pi Foundation. It brings Wi-Fi connectivity to the already popular Raspberry Pi Pico platform. This board is aimed at makers, educators, and developers for IoT applications, where wireless communication is crucial.

Processor: Dual-core ARM Cortex-M0+ @ 133MHz.

Memory: 264KB SRAM, and up to 16MB external flash memory.

Connectivity: 2.4GHz 802.11n Wi-Fi.

GPIOs: 26 multifunctional GPIO pins.

Interfaces: I2C, SPI, UART, ADC, PWM, and PIO (Programmable I/O).

· **Why Rpi Pico W?**

a) Better Ecosystem for Learning and Prototyping: Compared to other Wi-Fi-enabled microcontroller platforms like the ESP32, the Pico W has better educational resources and a more stable ecosystem, which ensures more consistent performance and smoother development.

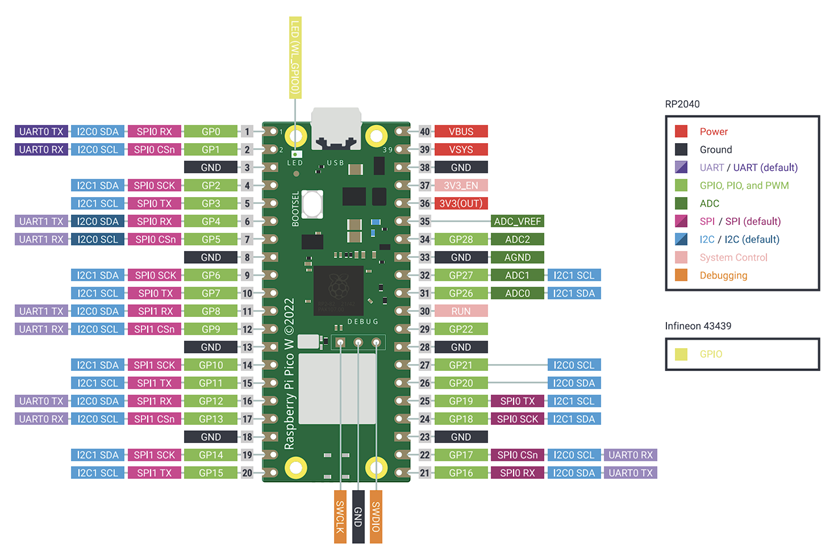
b) Wi-Fi Integration: Unlike the regular Pico, which requires external Wi-Fi modules, the Pico W has onboard Wi-Fi, simplifying hardware connections.

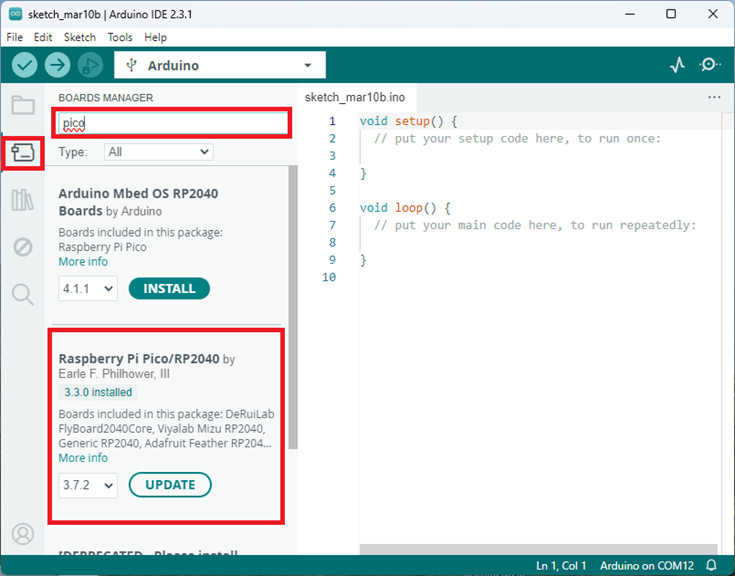
c) Power Efficiency: Although ESP32 is also low power, the RP2040 chip is specifically designed for efficient power management, making the Pico W more suitable for projects where low power consumption is critical.

d) Open-source benefits: Due to the Rpi Pico W board being open-source, it’s implementation in IoT becomes much easier. The hardware is free to use and anyone can put it to their desired use as they want. Arduino IDE is used to program the microcontroller , which again is an open-source software.

**Documentation**:

[https://datasheets.raspberrypi.com/picow/pico-w-datasheet.pdf?\_gl=1\*16youy\*\_ga\*MTkyMTc4Mzc0OS4xNzIxODMxMTEy\*\_ga\_22FD70LWDS\*MTcyNzY3NjU2My4yLjEuMTcyNzY3Nzk5OC4wLjAuMA](https://datasheets.raspberrypi.com/picow/pico-w-datasheet.pdf?_gl=1*16youy*_ga*MTkyMTc4Mzc0OS4xNzIxODMxMTEy*_ga_22FD70LWDS*MTcyNzY3NjU2My4yLjEuMTcyNzY3Nzk5OC4wLjAuMA)..



For adding the MCU to Arduino IDE:<https://github.com/earlephilhower/arduino-pico/releases/download/global/package_rp2040_index.json>

B. **Sensor module:** The HC-SR04 is an affordable ultrasonic distance sensor used in various applications like obstacle detection, distance measurement, and object avoidance systems. It works by emitting ultrasonic sound waves and measuring the time it takes for the waves to return after reflecting off an object.

Operating Voltage: 5V

Ultrasonic Frequency: 40kHz

Range: 2cm to 400cm (with an accuracy of ±3mm)

Angle of Detection: 15 degrees

Trigger Input Signal: 10µs pulse

Echo Output Signal: Pulse width proportional to the distance of the object

· How Ultrasonic Sensors Work for Dustbin Fill-Level Measurement:

a) Sound Wave Emission: The ultrasonic sensors, typically mounted on the lid or top of the dustbin, emit high-frequency sound waves toward the waste material inside.

b) Reflection: When the sound waves hit the top surface of the waste in the bin, they bounce back to the sensors.

c) Output: The sensor outputs this distance, and based on the bin’s total height, the system can calculate how full the dustbin is as a percentage.

· Reasons for Using HC-SR04 Over Other Modules:

a) Cost-Effectiveness: Compared to other sensors like infrared or laser distance sensors, the HC-SR04 is far more affordable, making it an excellent choice for budget-sensitive projects.

b) Wide Range of Use: While infrared sensors are limited by light conditions and laser sensors are more expensive, the HC-SR04 works well in a wide range of conditions and distances.

c) Simplicity: The sensor requires minimal external components and software to function. Many libraries are readily available to handle the sensor in popular microcontroller environments like Arduino, making it easier to integrate.

d) Proven Reliability: The HC-SR04 has been widely adopted and proven reliable in many projects, including obstacle-avoiding robots, level sensing, and proximity detection.

· Documentation and datasheet :

<https://web.eece.maine.edu/~zhu/book/lab/HC-SR04%20User%20Manual.pdf>



**C.** **COMMUNICATION MODULE (SX1278):**

The SX1278 is a long-range, low-power transceiver module that operates at 433MHz and is based on LoRa (Long Range) wireless technology. LoRa allows communication over vast distances while maintaining low power consumption, making it ideal for IoT applications such as remote sensing, smart cities, and agriculture.

**a)**  **Modulation**: LoRa uses a modulation technique known as Chirp Spread Spectrum (CSS), which spreads the signal across a wide frequency band. This makes LoRa highly resistant to interference and noise, allowing it to operate over long distances, up to 15-20 km in rural areas and a few kilometers in urban environments.

**b)**  **Low Data Rate**: LoRa is designed for low data rate communication, which is ideal for devices that need to send small amounts of data occasionally (such as sensor data in IoT networks).

**c)** **Power Efficiency**: LoRa devices are extremely power-efficient and can run for years on a single battery, making them suitable for remote sensors.

**LoRa vs. Other RF Technologies:**

· Better Range: Compared to traditional RF modules (e.g., NRF24L01 or ESP8266), the SX1278 with LoRa modulation offers far superior range (up to 10 km in rural areas) while maintaining low power consumption.

· Low Power Advantage: Other RF communication methods like Wi-Fi (used by ESP8266) consume significantly more power, making them unsuitable for remote or battery-powered applications where energy conservation is critical.

· Open-Source Support: The SX1278 has strong community support in the open-source ecosystem. Many open-source LoRa libraries for platforms like Arduino, ESP32, and Raspberry Pi enable rapid development.

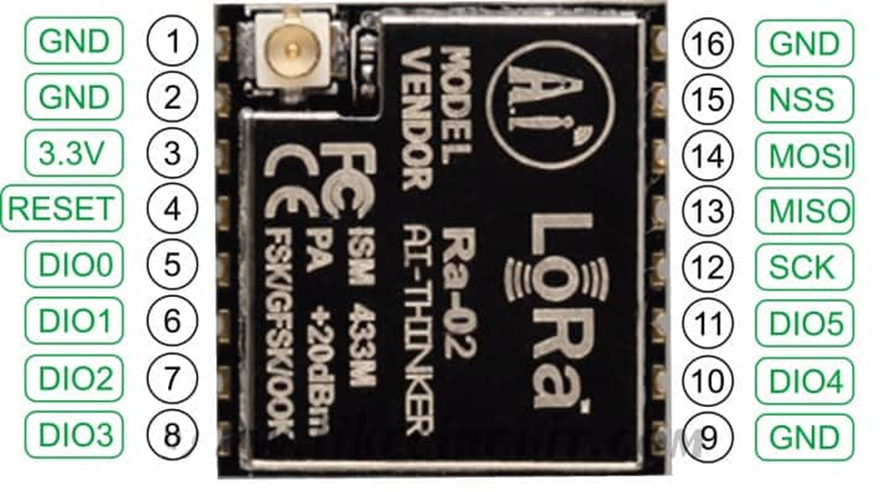
· LoRa operates in the unlicensed ISM (Industrial, Scientific, and Medical) bands, which vary by country. The SX1278 module operates in the 433 MHz band, commonly used in Asia and parts of Europe.

**LoRaWAN:**

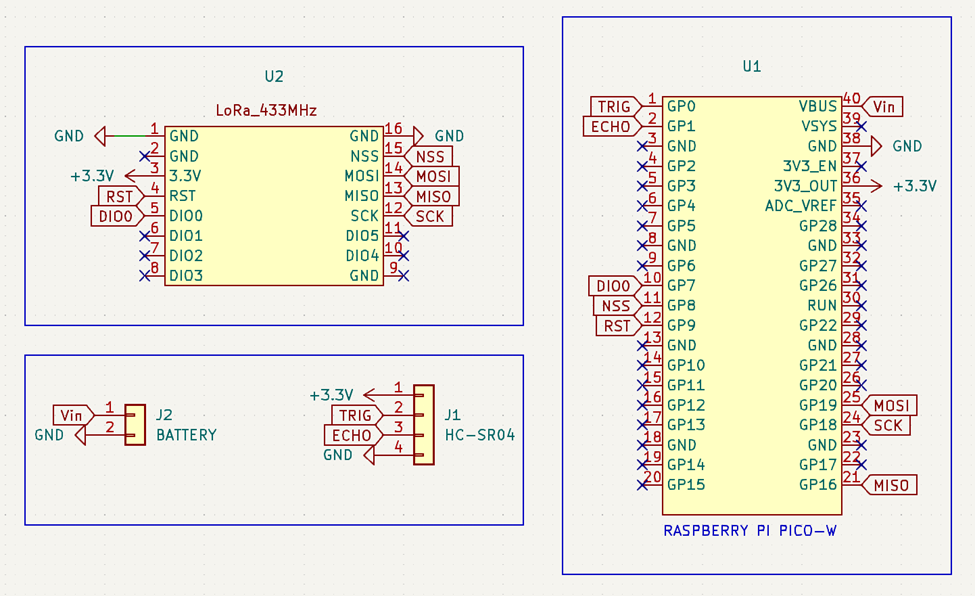
The system architecture and communication protocol for controlling LoRa devices is known as LoRaWAN (Long Range Wide Area Networks). It outlines the connections that LoRa devices make to gateways and the means by which they communicate with a central network server. End-user devices communicate with a gateway, which subsequently forwards the information to a cloud server, according to the star topology used by LoRaWAN.

· Documentation and datasheet:

<https://www.e-gizmo.net/oc/kits%20documents/LORA%20Module%20RA-02%20V.1/LORA%20rev2.pdf>



- **SCHEMATIC OF THE CIRCUIT:**

****

Pin connections on the circuit:

1. LoRa module :

a) NSS- GPIO pin 8 on MCU

b) RST- GPIO pin 9 on MCU

c) DIO0- GPIO pin 7 on MCU

d) MISO- GPIO pin 16 on MCU

e) MOSI- GPIO pin 19 on MCU

f) SCK- GPIO pin 18 on MCU

g) Power connections from Rpi

2. Ultrasonic sensor:

a) TRIG pin- GPIO 0 on MCU

b) ECHO pin- GPIO 1 on MCU

c) Power connections from battery.

3. Rpi Pico W:

a) Power connections: 3.7 V battery at the VBUS and GND terminals

b) 3.3V output power connections for regulated voltage to LoRa module

1. **Software-**

**Transmitter Code-**

This Arduino code is designed to measure the fill level of a bin using an ultrasonic sensor and transmit that data wirelessly via a LoRa (Long Range) communication module. Below is a detailed explanation of each part of the code.

## **Libraries Included**

#include <SPI.h> // Library for SPI communication, used by the LoRa module

#include <LoRa.h> // Library for LoRa communication

* **SPI.h**: Provides functions for SPI communication, which is essential for interfacing with the LoRa module.
* **LoRa.h**: Contains functions specifically for LoRa radio communication.

## **Pin Definitions**

#define NSS 8 // LoRa module's Slave Select pin connected to Arduino pin 8

#define RST 9 // LoRa module's Reset pin connected to Arduino pin 9

#define DIO0 7 // LoRa module's DIO0 pin connected to Arduino pin 7

#define TRIG\_PIN1 0 // Trigger pin for the ultrasonic sensor connected to Arduino pin 0

#define ECHO\_PIN1 1 // Echo pin for the ultrasonic sensor connected to Arduino pin 1

//#define TRIG\_PIN2 3 // (Commented out) Trigger pin for a second ultrasonic sensor

//#define ECHO\_PIN2 2 // (Commented out) Echo pin for a second ultrasonic sensor

#define BIN\_HEIGHT 41.3 // Height of the bin in centimeters

* **NSS, RST, DIO0**: Pins used to control the LoRa module.
* **TRIG\_PIN1, ECHO\_PIN1**: Pins connected to the ultrasonic sensor for sending and receiving ultrasonic pulses.
  + **Note**: Pins 0 and 1 are typically used for serial communication on Arduino Uno; using them for other purposes might cause conflicts.
* **BIN\_HEIGHT**: The total height of the bin, used to calculate the fill percentage.
* **TRIG\_PIN2, ECHO\_PIN2**: (Commented out) Pins for a second ultrasonic sensor, if needed.

## 

## **Node Addresses**

byte MasterNode = 0xFF; // Sender address

byte Node2 = 0xCC; // Recipient address

* **MasterNode**: The address of the sender (this device).
* **Node2**: The address of the recipient device.

## **Setup Function**

void setup() {

Serial.begin(9600); // Initialize serial communication at 9600 baud

pinMode(TRIG\_PIN1, OUTPUT); // Set trigger pin as output

pinMode(ECHO\_PIN1, INPUT); // Set echo pin as input

LoRa.setPins(NSS, RST, DIO0); // Set the LoRa module's control pins

if (!LoRa.begin(433E6)) { // Initialize LoRa at 433 MHz

Serial.println("LoRa init failed. Check your connections.");

while (true); // Stay here if LoRa initialization fails

}

Serial.println("LoRa init succeeded.");

}

* **Serial.begin(9600)**: Starts serial communication for debugging.
* **pinMode**: Sets the ultrasonic sensor's trigger and echo pins as output and input, respectively.
* **LoRa.setPins(NSS, RST, DIO0)**: Configures the control pins for the LoRa module.
* **LoRa.begin(433E6)**: Initializes the LoRa module at a frequency of 433 MHz.
  + **Note**: The frequency should match the hardware's capabilities and legal regulations in your region.
* **Error Handling**: If LoRa initialization fails, the code enters an infinite loop after printing an error message.

## **Distance Measurement Function**

long getDistance(int trigPin, int echoPin) {

long duration, distance;

// Send a 10µs pulse to trigger the sensor

digitalWrite(trigPin, LOW);

delayMicroseconds(2);

digitalWrite(trigPin, HIGH);

delayMicroseconds(10);

digitalWrite(trigPin, LOW);

// Read the echo pin and measure the duration of the pulse

duration = pulseIn(echoPin, HIGH);

// Calculate distance in centimeters

distance = (duration \* 0.034) / 2;

return distance;

}

* **Purpose**: Measures the distance from the ultrasonic sensor to the surface of the material in the bin.
* **Process**:
  + Sends a trigger pulse to start the measurement.
  + Uses pulseIn to measure the duration of the echo pulse.
  + Calculates the distance using the speed of sound (approx. 0.034 cm/µs).
  + Divides by 2 because the pulse travels to the object and back.

## **Main Loop**

void loop() {

long distance1 = getDistance(TRIG\_PIN1, ECHO\_PIN1);

//long distance2 = getDistance(TRIG\_PIN2, ECHO\_PIN2); // For a second sensor

// Calculate the average distance (only one sensor is used here)

long averageDistance = distance1;

// Calculate the fill percentage

int fillPercentage = ((BIN\_HEIGHT - averageDistance) \* 100) / BIN\_HEIGHT;

// Ensure the fill percentage is between 0% and 100%

if (fillPercentage < 0) {

fillPercentage = 0;

} else if (fillPercentage > 100) {

fillPercentage = 100;

}

// Prepare the message to send

String binID = "1"; // The ID of the bin

String message = binID + "," + String(fillPercentage);

// Send the message via LoRa

LoRa.beginPacket();

LoRa.write(Node2); // Recipient address

LoRa.write(MasterNode); // Sender address

LoRa.write(0); // Message ID (can be incremented if needed)

LoRa.write(message.length()); // Payload length

LoRa.print(message); // Payload data

LoRa.endPacket(); // Send the packet

Serial.println("Message sent: " + message); // Debug output

delay(5000); // Wait 5 seconds before the next measurement

}

### 

### 

### 

### **Steps Explained**

1. **Distance Measurement**:
   * **distance1**: Measures the distance using the first ultrasonic sensor.
   * **distance2**: (Commented out) If using a second sensor, you can measure and include it.
2. **Average Distance**:
   * **averageDistance**: Since only one sensor is used, this is just distance1. If multiple sensors were used, you could calculate the average.
3. **Fill Percentage Calculation**:
   * **Formula**: ((BIN\_HEIGHT - averageDistance) \* 100) / BIN\_HEIGHT
     + Subtracts the measured distance from the bin height to get the filled height.
     + Converts it to a percentage of the total bin height.
   * **Bounds Checking**: Ensures the fill percentage is between 0% and 100%.
4. **Prepare the Message**:
   * **binID**: A string representing the bin's identifier, set to "1" here.
   * **message**: Concatenates the binID and fillPercentage into a comma-separated string, e.g., "1,75".
5. **LoRa Communication**:
   * **LoRa.beginPacket()**: Starts assembling a new LoRa packet.
   * **LoRa.write(Node2)**: Adds the recipient's address.
   * **LoRa.write(MasterNode)**: Adds the sender's address.
   * **LoRa.write(0)**: Adds a message ID (static 0 here, can be incremented for each message if needed).
   * **LoRa.write(message.length())**: Adds the length of the payload.
   * **LoRa.print(message)**: Adds the actual message content.
   * **LoRa.endPacket()**: Sends the assembled packet over the LoRa network.
6. **Debug Output**:
   * **Serial.println**: Prints the sent message to the serial monitor for debugging purposes.
7. **Delay**:
   * **delay(5000)**: Waits for 5 seconds before repeating the loop, reducing the frequency of measurements and transmissions to save power and bandwidth.

## **Additional Notes**

* **Using Pins 0 and 1**:
  + Be cautious when using pins 0 and 1 on Arduino Uno, as they are also used for serial communication (RX and TX). This can interfere with uploading code and serial debugging. It's recommended to use different pins if possible.
* **Second Ultrasonic Sensor**:
  + The code includes commented-out sections for a second ultrasonic sensor. If you wish to use it, uncomment the relevant lines and adjust the calculations to include distance2.
* **LoRa Frequency**:
  + The code sets the LoRa module to operate at **433 MHz**. Ensure that this frequency is legal and appropriate for LoRa use in your region. Other common frequencies for LoRa are 868 MHz and 915 MHz.
* **Error Handling**:
  + If the LoRa module fails to initialize, the code enters an infinite loop after printing an error message, effectively halting the program.
* **Message ID**:
  + The message ID is currently set to 0 in the code. If you need to track messages or handle duplicates, consider incrementing this ID with each transmission.

## **Summary**

* **Purpose**: Measure the fill level of a bin and send that information wirelessly to another device.
* **How it Works**:
  + **Measurement**: Uses an ultrasonic sensor to measure the distance to the material's surface in the bin.
  + **Calculation**: Determines the fill percentage based on the bin's height.
  + **Communication**: Sends the fill percentage and bin ID via LoRa to a designated recipient.
* **Use Case**: Ideal for remote monitoring of bin levels in applications like waste management, agriculture (e.g., grain bins), or any scenario where knowing the fill level is essential.

**Receiver Code-**

### **1. Library Includes**

#include <SPI.h>

#include <LoRa.h>

#include <WiFi.h>

#include <HTTPClient.h>

* **SPI.h**: Required for communication with the LoRa module.
* **LoRa.h**: Handles communication via LoRa (long-range radio).
* **WiFi.h**: Manages Wi-Fi connectivity.
* **HTTPClient.h**: Used to make HTTP requests (for sending data to Firebase).

### **2. Wi-Fi and Firebase Configuration**

#define STASSID "yourwifiname" // Wi-Fi SSID

#define STAPSK "yourwifipassword" // Wi-Fi Password

#define FIREBASE\_HOST "testapp-8ef1c" // Firebase Project ID

#define FIREBASE\_API\_KEY "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" // Firebase API Key

#define FIREBASE\_COLLECTION "bin\_data" // Firestore Collection Name

* **STASSID**: Your Wi-Fi network name (SSID).
* **STAPSK**: Your Wi-Fi network password.
* **FIREBASE\_HOST**: The unique identifier for your Firebase project.
* **FIREBASE\_API\_KEY**: The API key used for authentication when making requests to Firebase.
* **FIREBASE\_COLLECTION**: The name of the Firestore collection where bin data is stored.

### **3. LoRa Module Pin Configuration**

#define NSS 8 // LoRa module chip select

#define RST 9 // LoRa module reset pin

#define DIO0 7 // LoRa module interrupt pin

* **NSS**, **RST**, and **DIO0**: Pins used to interface with the LoRa module.

### **4. Node Addresses**

byte MasterNode = 0xFF; // Address for the Master Node

byte Node2 = 0xCC; // Address for the Receiving Node (this device)

* **MasterNode**: The sender’s address.
* **Node2**: The address of this device (receiver node).

### **5. Setup Function**

void setup() {

Serial.begin(115200);

// Connect to Wi-Fi

WiFi.begin(STASSID, STAPSK);

while (WiFi.status() != WL\_CONNECTED) {

delay(500);

Serial.print(".");

}

Serial.println("Connected! IP address: ");

Serial.println(WiFi.localIP());

// Initialize LoRa

LoRa.setPins(NSS, RST, DIO0);

if (!LoRa.begin(433E6)) {

Serial.println("LoRa init failed.");

while (true);

}

Serial.println("LoRa init succeeded.");

}

* **Wi-Fi Initialization**: Connects to the Wi-Fi network using the provided credentials. It loops until a successful connection is established.
* **LoRa Initialization**: Sets up the LoRa module to communicate on the 433 MHz frequency. If the LoRa setup fails, the code halts.

### **6. Main Loop**

void loop() {

int packetSize = LoRa.parsePacket();

if (packetSize) {

Serial.print("Received packet: ");

// Read LoRa packet header

int recipient = LoRa.read();

byte sender = LoRa.read();

byte incomingMsgId = LoRa.read();

byte incomingLength = LoRa.read();

// Read message content

String incoming = "";

while (LoRa.available()) {

incoming += (char)LoRa.read();

}

// Validate message length

if (incomingLength != incoming.length()) {

Serial.println("Error: length mismatch");

return;

}

// Check recipient address

if (recipient != Node2 && recipient != MasterNode) {

Serial.println("Message not for this node.");

return;

}

// Process the received data

Serial.println(incoming);

// Split the message into binID and fillPercent

int separatorIndex = incoming.indexOf(',');

if (separatorIndex == -1) {

Serial.println("Error: Invalid message format");

return;

}

String binID = incoming.substring(0, separatorIndex);

String fillPercent = incoming.substring(separatorIndex + 1);

fillPercent.trim();

// Print extracted data

Serial.print("Bin ID: ");

Serial.println(binID);

Serial.print("Fill Percent: ");

Serial.println(fillPercent);

// Send the data to Firebase

sendToFirebase(binID, fillPercent);

}

}

* **LoRa Packet Parsing**: The device continuously checks for incoming LoRa packets.
  + **Recipient Check**: The message is processed only if the recipient address matches this device or the master node.
  + **Message Validation**: Ensures the message length is correct.
  + **Message Content**: Splits the received message (e.g., "1,75") into binID and fillPercent values.

### **7. Sending Data to Firebase**

void sendToFirebase(String binID, String fillPercent) {

if (WiFi.status() == WL\_CONNECTED) {

HTTPClient http;

http.setInsecure();

String queryUrl = "https://firestore.googleapis.com/v1/projects/" + String(FIREBASE\_HOST) + "/databases/(default)/documents:runQuery?key=" + FIREBASE\_API\_KEY;

http.begin(queryUrl);

http.addHeader("Content-Type", "application/json");

String queryPayload = "{\"structuredQuery\": {\"where\": {\"fieldFilter\": {\"field\": {\"fieldPath\": \"bin\_id\"}, \"op\": \"EQUAL\", \"value\": {\"integerValue\": \"" + binID + "\"}}}, \"from\": [{\"collectionId\": \"" + String(FIREBASE\_COLLECTION) + "\"}]}}";

int queryHttpCode = http.POST(queryPayload);

if (queryHttpCode > 0) {

String queryResponse = http.getString();

Serial.println("Query response: " + queryResponse);

if (queryResponse.indexOf("document") != -1) {

// Document found, update fillPercent

int idStartIndex = queryResponse.indexOf("\"name\": \"") + 9;

int idEndIndex = queryResponse.indexOf("\",", idStartIndex);

String documentPath = queryResponse.substring(idStartIndex, idEndIndex);

String patchUrl = "https://firestore.googleapis.com/v1/" + documentPath + "?key=" + FIREBASE\_API\_KEY + "&updateMask.fieldPaths=fill\_percent";

http.begin(patchUrl);

http.addHeader("Content-Type", "application/json");

String jsonPayload = "{\"fields\": {\"fill\_percent\": {\"integerValue\": \"" + fillPercent + "\"}}}";

int patchHttpCode = http.PATCH(jsonPayload);

} else {

// No document found, create a new one

String postUrl = "https://firestore.googleapis.com/v1/projects/" + String(FIREBASE\_HOST) + "/databases/(default)/documents/" + String(FIREBASE\_COLLECTION) + "?key=" + FIREBASE\_API\_KEY;

http.begin(postUrl);

http.addHeader("Content-Type", "application/json");

String jsonPayload = "{\"fields\": {\"bin\_id\": {\"integerValue\": \"" + binID + "\"}, \"fill\_percent\": {\"integerValue\": \"" + fillPercent + "\"}}}";

int postHttpCode = http.POST(jsonPayload);

}

}

http.end();

} else {

Serial.println("Wi-Fi not connected");

}

}

* **Firebase Query**: Sends an HTTP POST request to query Firestore for a document matching the bin\_id.
* **Document Update or Creation**:
  + **If Found**: A PATCH request is sent to update the fill\_percent field.
  + **If Not Found**: A POST request is sent to create a new document with the bin\_id and fill\_percent.