All Embedded C ++ Codes, Node Red Flow JSON, Simulation Link And Wiring Diagram(Latrine Fill Level And Hygiene Monitor)

NAMAKHANYU WILLIAMS 2301600082

KEINEBAGAZA LINCOLN 2301600088

BABIRYE AISHA 2301600101

AMONGI EDNA 2301600105

Bluetooth communication

Blue tooth sender.

```
#include <BLEDevice.h>

#include <BLEServer.h>
#include <BLEUtils.h>
#include <BLE2902.h>

#define PIR 2
#define ULTRASONIC_PIN 34

BLECharacteristic *sensorCharacteristic;
int pirState = 0; int distance = 0;

void setup() {
Serial.begin(9600);   pinMode(PIR,
INPUT);
  pinMode(ULTRASONIC_PIN, INPUT);

  BLEDevice::init("NodeA-BLE");
  BLEServer *pServer = BLEDevice::createServer();
  BLEService *pService = pServer->createService("180C");  // Custom service UUID
   sensorCharacteristic = pService-
>createCharacteristic(
    "2A56",  // Custom characteristic UUID
    BLECharacteristic::PROPERTY_READ | BLECharacteristic::PROPERTY_NOTIFY
  );    sensorCharacteristic->addDescriptor(new
BLE2902());   pService->start();
  BLEAdvertising *pAdvertising = BLEDevice::getAdvertising();   pAdvertising-
>start();
  Serial.println("BLE Node A ready...");
}  void
loop() {
  int sensorValue = analogRead(ULTRASONIC_PIN);
distance = map(sensorValue, 0, 4095, 0, 400);   pirState
= digitalRead(PIR);

  String payload = String("{\"distance_cm\":") + distance +
                   ",\"handwash\":" + (pirState == HIGH ? "1" : "0") + "}";
sensorCharacteristic->setValue(payload.c_str());   sensorCharacteristic-
>notify();
```

```
  Serial.println("Updated BLE characteristic:");
Serial.println(payload);   delay(2000);
}
```

BLUE TOOTH RECEIVER

```
#include <BLEDevice.h>
#include <BLEUtils.h>
#include <BLEScan.h>
#include <BLEAdvertisedDevice.h>

#define SERVICE_UUID        "180C"
#define CHARACTERISTIC_UUID "2A56"

BLEAdvertisedDevice* myDevice;
BLERemoteCharacteristic* pRemoteCharacteristic;
bool doConnect = false; bool connected = false;
BLEClient* pClient;

// ---------------- Notification callback ---------------- static void
notifyCallback(BLERemoteCharacteristic* pBLERemoteCharacteristic,
uint8_t* pData, size_t length, bool isNotify) {   String received = "";
  for (size_t i = 0; i < length; i++) received += (char)pData[i];
  Serial.print("Received payload: ");
  Serial.println(received);
}

// ---------------- Callback class for scanning ---------------- class
MyAdvertisedDeviceCallbacks : public BLEAdvertisedDeviceCallbacks {
void onResult(BLEAdvertisedDevice advertisedDevice) {
    // Look for our Node A name
    if (advertisedDevice.getName() == "NodeA-BLE") {
Serial.println("Found NodeA-BLE! Stopping scan...");
advertisedDevice.getScan()->stop();       myDevice = new
BLEAdvertisedDevice(advertisedDevice);       doConnect =
true;
    }
  }
};

// ---------------- Connect to server ---------------- bool
connectToServer() {
  Serial.print("Connecting to ");
  Serial.println(myDevice->getAddress().toString().c_str());
pClient = BLEDevice::createClient();   if (!pClient-
```

```
>connect(myDevice)) {     Serial.println("Connection
failed.");     return false;
  }
  Serial.println("Connected to NodeA-BLE");

  BLERemoteService* pRemoteService = pClient->getService(SERVICE_UUID);
if (pRemoteService == nullptr) {
    Serial.println("Failed to find service UUID.");
pClient->disconnect();     return false;
  }    pRemoteCharacteristic = pRemoteService-
>getCharacteristic(CHARACTERISTIC_UUID);   if (pRemoteCharacteristic == nullptr)
{
    Serial.println("Failed to find characteristic UUID.");
pClient->disconnect();     return false;
  }

  // Register notification callback   if
(pRemoteCharacteristic->canNotify()) {
pRemoteCharacteristic->registerForNotify(notifyCallback);
  }    return
true;
}

// ---------------- Setup ---------------- void
setup() {
  Serial.begin(9600);
  Serial.println("Starting BLE Node B (Receiver)...");

  BLEDevice::init("");
  BLEScan* pBLEScan = BLEDevice::getScan();
  pBLEScan->setAdvertisedDeviceCallbacks(new MyAdvertisedDeviceCallbacks());
pBLEScan->setInterval(1349);   pBLEScan->setWindow(449);   pBLEScan-
>setActiveScan(true);   pBLEScan->start(5, false);
}

// ---------------- Loop ----------------
void loop() {   if (doConnect) {     if
(connectToServer()) {
      Serial.println("We are now connected and listening for updates...");
connected = true;
    } else {
      Serial.println("Connection failed. Restarting scan...");
      BLEDevice::getScan()->start(5, false);
    }     doConnect =
false;
```

```
  }    if
(!connected) {
delay(2000);
  }
}
```

LoRa commnuication

LoRa receiver

```
#include <Arduino.h>
#include "LoRa_E32.h"

// LoRa connected: TXD → GPIO17, RXD → GPIO16 (same as sender)
#define LORA_RX_PIN 16
#define LORA_TX_PIN 17

LoRa_E32 e32ttl(&Serial2);

String getValue(const String &msg, const String &key) {
int start = msg.indexOf(key);   if (start < 0) return
"N/A";   int colon = msg.indexOf(':', start);   if
(colon < 0) return "N/A";   int i = colon + 1;   //
skip spaces   while (i < (int)msg.length() &&
isspace(msg[i])) i++;   int j = i;
  // read until whitespace or end    while (j <
(int)msg.length() && !isspace(msg[j])) j++;   return
msg.substring(i, j);
}

void printHexDump(const String &s) {
Serial.print("Raw hex: ");   for (size_t i
= 0; i < s.length(); ++i) {     uint8_t b =
(uint8_t)s[i];     if (b < 16)
Serial.print('0');
    Serial.print(b, HEX);
    Serial.print(' ');
  }
  Serial.println();
}
void setup() {
  Serial.begin(115200);
  // Explicitly set Serial2 pins and format for ESP32
Serial2.begin(9600, SERIAL_8N1, LORA_RX_PIN, LORA_TX_PIN);
e32ttl.begin();
```

```
  Serial.println("🌿 Node B — LoRa Receiver Ready");
  Serial.println("Waiting for incoming data...\n");
}  void loop() {   if
(Serial2.available() > 0) {
    // Read all currently available bytes into a String
String msg = "";
    unsigned long start = millis();
    // read for up to 50 ms to gather a full packet (adjust if needed)
while (millis() - start < 50) {       while (Serial2.available() > 0)
{         char c = (char)Serial2.read();         msg += c;
      }     }
msg.trim();
    if (msg.length() == 0) return;

    // Debug: raw print and hex dump
    Serial.println("📥 Received via LoRa → " + msg);
printHexDump(msg);

    // Robust parsing
    String distanceStr = getValue(msg, "DISTANCE");
    String pirStr = getValue(msg, "PIR");
    String statusStr = getValue(msg, "STATUS");

    Serial.println("📌 Parsed Data:");
    Serial.println("  Distance → " + distanceStr);
    Serial.println("  PIR → " + pirStr);
    Serial.println("  Status → " + statusStr);

    // ---- Send ACK back to sender ----
delay(10); // small safety gap
e32ttl.sendMessage("ACK");     delay(10);
    Serial.println("📤 ACK sent to Node A");
    Serial.println("--------------------------------\n");
  }
}
```

LoRa Sender

```
#include <Arduino.h>
#include "LoRa_E32.h"

// Sender (Node A) pins for ESP32
#define LORA_RX_PIN 16 // this is ESP32 RX pin (connect to LoRa TX)
#define LORA_TX_PIN 17 // this is ESP32 TX pin (connect to LoRa RX)
```

```
LoRa_E32 e32ttl(&Serial2);

float readDistance() {

  return random(20, 200) / 1.0; // simulated cm
}
int readPIR() {
  return random(0, 2); // simulated 0/1
}
String readStatus() {
  // TODO: your logic for status
  return "OK";
}
String readFromLoRaTimeout(unsigned long timeoutMs) {
  String s = "";
  unsigned long start = millis();
  while (millis() - start < timeoutMs) {
    while (Serial2.available() > 0) {
      char c = (char)Serial2.read();
      s += c;
    }
  }
  s.trim();
  return s;
}

void setup() {
  Serial.begin(115200);
    Serial2.begin(9600, SERIAL_8N1, LORA_RX_PIN, LORA_TX_PIN);
  e32ttl.begin();

  Serial.println("🌿 Node A — LoRa Sender Ready");
  Serial.println("Sending periodic telemetry to Node B...\n");

  randomSeed(analogRead(0));
}

void loop() {
  // --- Gather sensor data (replace with real sensors) ---
  float dist = readDistance();
  int pir = readPIR();
  String status = readStatus();
```

```
  String msg = "DISTANCE:" + String(dist, 1) + " PIR:" + String(pir) + " STATUS:"
+ status + "\n";

  // Log locally
  Serial.print("📤 Sending → ");
  Serial.println(msg);

  // Send via LoRa
  e32ttl.sendMessage(msg);

  // Wait a short time for ACK from Node B (and read any response bytes)
  String ack = readFromLoRaTimeout(200); // 200 ms timeout, adjust if needed
  if (ack.length() > 0) {
    Serial.print("📥 From LoRa (reply): ");
    Serial.println(ack);
    if (ack.indexOf("ACK") >= 0) {
      Serial.println("✅ ACK received from Node B");
    } else {
      Serial.println("ℹ️ Received non-ACK reply");
    }
  } else {
    Serial.println("⚠️ No reply received (timeout)");
  }

  Serial.println("--------------------------------\n");

  // Wait before next transmission (adjust as needed)
  delay(2000);
}
```

Wi-Fi communication

Wi-Fi sender

```
/* Node A - UDP sender (ESP32)
   Sends JSON payloads via UDP broadcast every 2000 ms.
```

```
   Payload example:
   {"seq":12,"send_ms":12345678,"distance_cm":125,"handwash":1}
*/

#include <WiFi.h>
#include <WiFiUdp.h>
#include <Wire.h>
 const char* WIFI_SSID = "TECNO SPARK 10 Pro";      // <--
replace const char* WIFI_PASS = "#keine21";      // <-- replace

const IPAddress destIP = IPAddress(10,57,85,66); // broadcast (change to Gateway
IP if desired) const uint16_t UDP_PORT = 4210;

#define PIR_PIN 2
#define ULTRASONIC_PIN 34

WiFiUDP Udp; unsigned long seq = 0;
unsigned long sendIntervalMs = 2000;
unsigned long lastSend = 0;
 void setup() {
pinMode(PIR_PIN, INPUT);
pinMode(ULTRASONIC_PIN, INPUT);
Serial.begin(115200);
delay(100);

  Serial.printf("Connecting to Wi-Fi SSID: %s\n", WIFI_SSID);
  WiFi.mode(WIFI_STA);
  WiFi.begin(WIFI_SSID, WIFI_PASS);
   unsigned long start = millis();
while (WiFi.status() != WL_CONNECTED) {
delay(200);
    if (millis() - start > 10000) {
      Serial.println("Still trying to connect to Wi-Fi...");
start = millis();
    }
  }
  Serial.print("Wi-Fi connected, IP: ");
  Serial.println(WiFi.localIP());

  Udp.begin(UDP_PORT); // local port for sending doesn't strictly matter for
broadcast
  Serial.printf("UDP ready, broadcasting to %s:%u\n", destIP.toString().c_str(),
UDP_PORT);
}  void loop() {   if (millis() - lastSend <
sendIntervalMs) return;   lastSend = millis();
```

```
  // Read sensors   int sensorValue =
analogRead(ULTRASONIC_PIN);
  int distance = map(sensorValue, 0, 4095, 0, 400); // calibrate experimentally
int pirState = digitalRead(PIR_PIN);

  // Build JSON payload
seq++;
  unsigned long send_ms = millis();
  String payload = String("{\"seq\":") + seq +
                   ",\"send_ms\":" + send_ms +
                   ",\"distance_cm\":" + distance +
                   ",\"handwash\":" + (pirState == HIGH ? 1 : 0) +
                   "}";

  // Send UDP packet (broadcast or specific gateway IP)
  Udp.beginPacket(destIP, UDP_PORT);
  Udp.write((const uint8_t*)payload.c_str(), payload.length()); Udp.endPacket();

  Serial.print("Sent: ");
  Serial.println(payload);
}
```

Wi-Fi receiver

```
/* Node B - UDP receiver (ESP32)
   Listens on UDP_PORT and expects JSON payloads from Node A.
   Computes one-way latency = (now_ms - send_ms), tracks seq numbers for PRR,
prints per-packet info and periodic summaries (every 60s) and final summary
after 5 minutes.
*/

#include <WiFi.h>
#include <WiFiUdp.h>
 const char* WIFI_SSID = "TECNO SPARK Pro ";      // <--
replace const char* WIFI_PASS = "#Keine";      // <--
replace
 const uint16_t UDP_PORT =
4210;

WiFiUDP Udp; unsigned long
startTime = 0;
```

```
// Metrics unsigned long
packetsReceived = 0; unsigned long
highestSeqSeen = 0; unsigned long
sumLatencyMs = 0;
unsigned long reportsIntervalMs = 60000; // show metrics every 60s unsigned
long lastReport = 0;
const unsigned long testDurationMs = 5UL * 60UL * 1000UL; // 5 minutes

void setup() {
Serial.begin(115200);   delay(100);

  Serial.printf("Connecting to Wi-Fi SSID: %s\n", WIFI_SSID);
  WiFi.mode(WIFI_STA);
  WiFi.begin(WIFI_SSID, WIFI_PASS);
   unsigned long s = millis();   while
(WiFi.status() != WL_CONNECTED) {
delay(200);
    if (millis() - s > 10000) {
      Serial.println("Still trying to connect to Wi-Fi...");
s = millis();
    }
  }
  Serial.print("Wi-Fi connected, IP: ");
  Serial.println(WiFi.localIP());

  if (Udp.begin(UDP_PORT) == 1) {
    Serial.printf("UDP listening on port %u\n", UDP_PORT);
  } else {
    Serial.println("Failed to start UDP listener.");
  }    startTime =
millis();   lastReport =
millis();
}

String extractValue(String &s, const char* key) {
  // crude JSON extractor: finds "key":NUMBER and returns NUMBER string (no
quotes)
  String pattern = String("\"") + key + "\":";   int
idx = s.indexOf(pattern);   if (idx < 0) return
String("");   idx += pattern.length();   int endIdx =
idx;   while (endIdx < (int)s.length()) {     char c =
s[endIdx];     if ((c >= '0' && c <= '9') || c == '-'
) endIdx++;     else break;
  }
  return s.substring(idx, endIdx);
```

```
}  void
loop() {
  int packetSize = Udp.parsePacket();
if (packetSize) {     // receive into
a buffer     char incoming[512];
    int len = Udp.read(incoming, sizeof(incoming) - 1);
if (len > 0) incoming[len] = 0;
    String payload = String(incoming);

    // crude parsing of fields
    String seq_s = extractValue(payload, "seq");
    String sendms_s = extractValue(payload, "send_ms");
    String dist_s = extractValue(payload, "distance_cm");
    String hw_s = extractValue(payload, "handwash");
     unsigned long seq = seq_s.length() ? (unsigned long) seq_s.toInt() : 0;
unsigned long send_ms = sendms_s.length() ? (unsigned long) sendms_s.toInt()
: 0;     int distance = dist_s.length() ? dist_s.toInt()
: -1;     int handwash = hw_s.length() ? hw_s.toInt() : -
1;
     unsigned long now_ms = millis();     unsigned long latency =
(now_ms >= send_ms) ? (now_ms - send_ms) : 0xFFFFFFFF;

    // update metrics     packetsReceived++;     if
(seq > highestSeqSeen) highestSeqSeen = seq;     if
(latency != 0xFFFFFFFF) sumLatencyMs += latency;

    // print packet info
    Serial.printf("Pkt #%lu : seq=%lu distance=%d handwash=%d latency_ms=%lu\n",
packetsReceived, seq, distance, handwash, latency);
  }

  // Periodic reports and final summary after testDurationMs   unsigned long
now = millis();   if (now - lastReport >= reportsIntervalMs) {
lastReport = now;     unsigned long seen = highestSeqSeen;     float prr =
seen > 0 ? (100.0f * (float)packetsReceived / (float)seen) :
0.0f;
    float avgLatency = packetsReceived ? ((float)sumLatencyMs /
(float)packetsReceived) : 0.0f;
    Serial.println("========== METRICS (interval) ==========");
    Serial.printf("Elapsed: %lu s\n", (now - startTime) / 1000);
    Serial.printf("Packets received: %lu\n", packetsReceived);
    Serial.printf("Highest seq seen: %lu\n", highestSeqSeen);
    Serial.printf("Packet success rate (PRR) approx: %.2f %%\n", prr);
    Serial.printf("Average latency: %.2f ms\n", avgLatency);
    Serial.println("========================================");
  }
```

```
  // Final result after 5 minutes   if
(now - startTime >= testDurationMs) {
unsigned long seen = highestSeqSeen;
float prr = seen > 0 ? (100.0f *
(float)packetsReceived / (float)seen) :
0.0f;     float avgLatency = packetsReceived ?
((float)sumLatencyMs /
(float)packetsReceived) : 0.0f;

    Serial.println("========== FINAL 5-MIN SUMMARY ==========");
    Serial.printf("Total packets received: %lu\n", packetsReceived);
    Serial.printf("Highest seq seen: %lu\n", highestSeqSeen);
    Serial.printf("Estimated Packet Success Rate: %.2f %%\n", prr);
    Serial.printf("Average one-way latency: %.2f ms\n", avgLatency);
Serial.println("========================================");
    )       // packetsReceived = 0; highestSeqSeen = 0; sumLatencyMs = 0;
startTime = millis();
    // Or halt further reporting by sleeping the MCU (not done here). We'll just
keep reporting periodically.
    while (true) { delay(1000); } // stop here so you can read final results;
press reset to run again
  }
}
```

MQTT CODE.

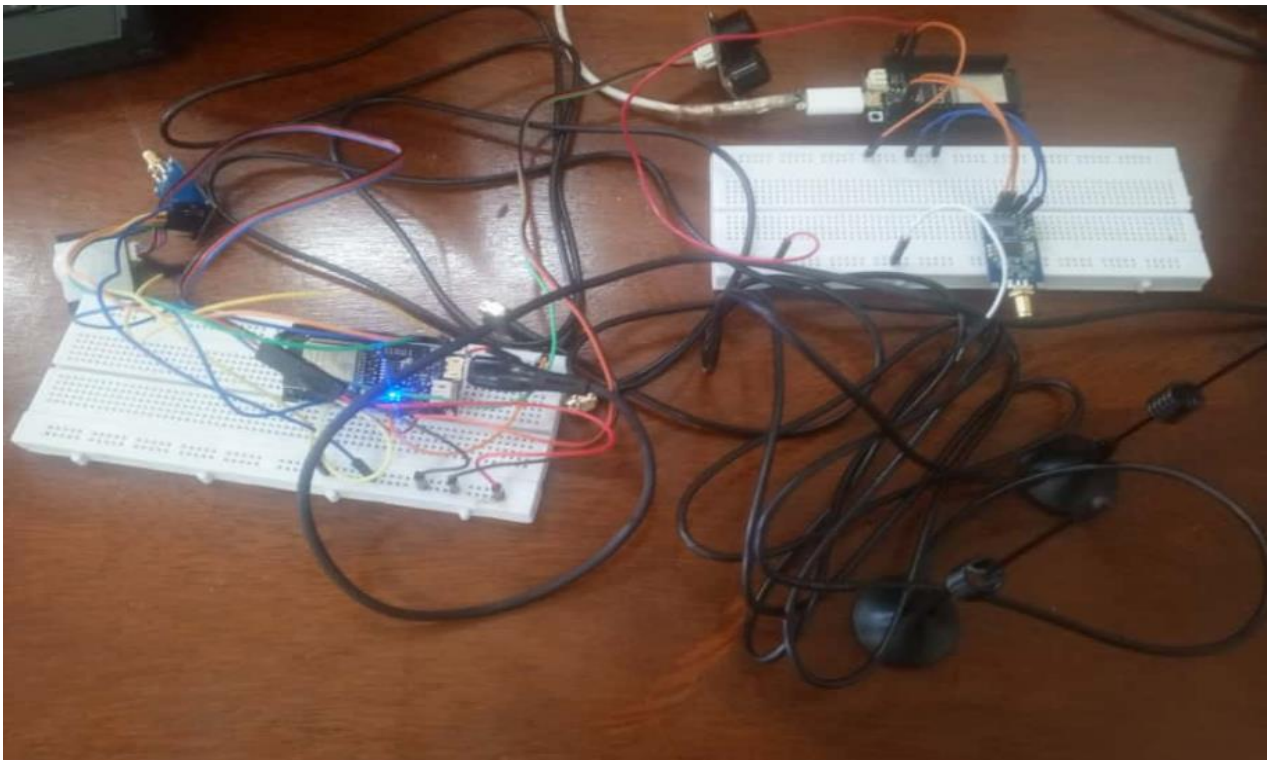
```
#include <WiFi.h>
#include <PubSubClient.h>
#include <Wire.h>

// ------------------- Wi-Fi Credentials -------------------
const char* ssid = "linco";       // ◆ Change to your Wi-Fi SSID
const char* password = "12345678";   // ◆ Change to your Wi-Fi password

// ------------------- MQTT Broker Settings -------------------
const char* mqtt_server = "broker.hivemq.com";  // ◆ Public broker (can replace
with local IP)
const int mqtt_port = 1883;

const char* topic_distance = "latrine_monitor/distance";
const char* topic_motion   = "latrine_monitor/motion";
```

```
const char* topic_alert    = "latrine_monitor/alert";

// ------------------- Pins ------------------
#define PIR 2
#define ULTRASONIC_PIN 34   // Analog output from 3-pin ultrasonic

// ------------------- Variables ------------------
int pirState = 0;
int distance = 0;

// ------------------- MQTT Client ------------------
WiFiClient wifiClient;
PubSubClient client(wifiClient);

// ====================================================
//                   Helper Functions
// ====================================================

// --- Connect to Wi-Fi ---
void setup_wifi() {
  Serial.print("Connecting to Wi-Fi: ");
  Serial.println(ssid);
  WiFi.begin(ssid, password);

  int retry = 0;
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
    if (++retry > 20) {
      Serial.println("\n⚠ Wi-Fi connect timeout, restarting...");
      ESP.restart();
    }
  }

  Serial.println("\n✅ Wi-Fi connected!");
  Serial.print("IP address: ");
  Serial.println(WiFi.localIP());
}

// --- Reconnect MQTT ---
void reconnect_mqtt() {
  while (!client.connected()) {
    Serial.print("Connecting to MQTT broker...");
    if (client.connect("ESP32_Latrine_Node")) {
      Serial.println("✅ connected!");
```

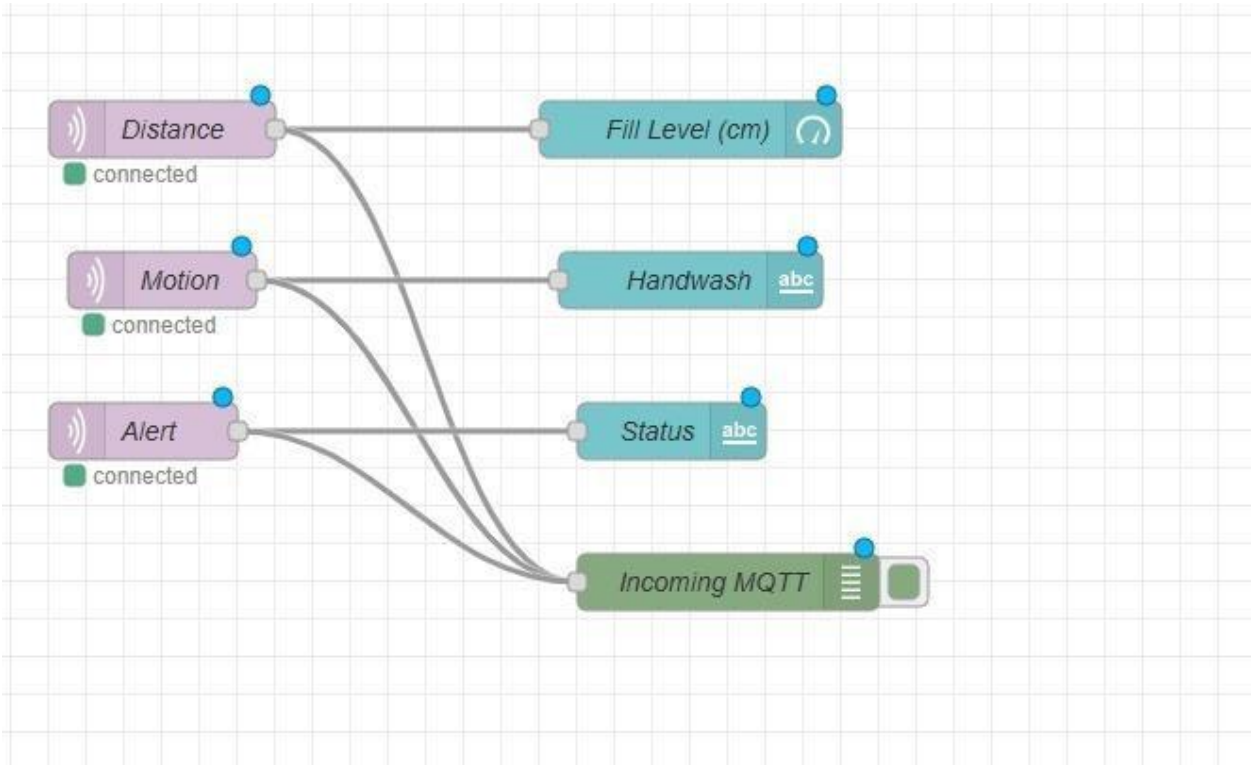
```
    } else {
      Serial.print("❌ failed, rc=");
      Serial.print(client.state());
      Serial.println(" — retrying in 5 seconds...");
      delay(5000);
    }
  }
}

// ====================================================
//                        SETUP
// ====================================================
void setup() {
  pinMode(PIR, INPUT);
  pinMode(ULTRASONIC_PIN, INPUT);

  Wire.begin();
  Serial.begin(9600);
  Serial.println("System initializing...");
  delay(1000);

  setup_wifi();
  client.setServer(mqtt_server, mqtt_port);
}

// ====================================================
//                        LOOP
// ====================================================
void loop() {
  // Keep MQTT alive
  if (WiFi.status() != WL_CONNECTED) setup_wifi();
  if (!client.connected()) reconnect_mqtt();
  client.loop();

  // ---------------- Read Sensors ----------------
  int sensorValue = analogRead(ULTRASONIC_PIN);
  distance = map(sensorValue, 0, 4095, 0, 400);
  pirState = digitalRead(PIR);

  // ---------------- Publish Data ----------------
  // 1️⃣ Distance
  String distancePayload = "{\"distance_cm\":" + String(distance) + "}";
  client.publish(topic_distance, distancePayload.c_str());
  Serial.println("📡 Sent distance → " + distancePayload);
```

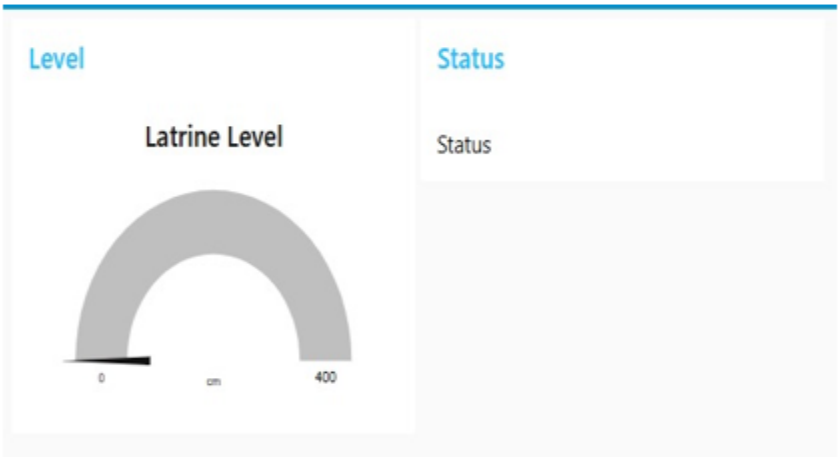
```
  // 2️⃣ PIR
  String motionState = (pirState == HIGH) ? "Detected" : "None";
  String motionPayload = "{\"motion\":\"" + motionState + "\"}";
  client.publish(topic_motion, motionPayload.c_str());
  Serial.println("📡 Sent motion → " + motionPayload);

  // 3️⃣ Alert
  if (distance < 10) {
    String alertPayload = "{\"alert\":\"Latrine Full!\"}";
    client.publish(topic_alert, alertPayload.c_str());
    Serial.println("⚠️ Sent alert → " + alertPayload);
  }

  Serial.println("----------------------------");
  delay(3000);  // Send every 3 seconds
}
```

WIRIING DIAGRAM



NODE RED FLOW DIAGRAM





Simulation link in tinker cad

https://www.tinkercad.com/things/0otRpwz9k5G-start-simulating?sharecode=cguxJbMZIGlG2i0_O65iasud1rh6-K6ayjE9Us3zEpk