SECTION 1

Overview of IoT concept

What is IoT?

The Internet of Things(IoT) refers to a network or interconnection of devices, appliances, vehicles or physical objects embedded with sensors, software or any other technology that allows them to communicate, collect and share data over the internet.

IoT is also termed as “smart devices” and can range from simple smart home devices to complex industrial and transportation systems.

IoT has revolutionized industries and daily life by enabling automation, improving efficiency, and providing real-time insights.

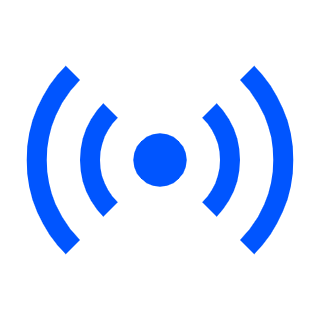
Key Concepts of IoT



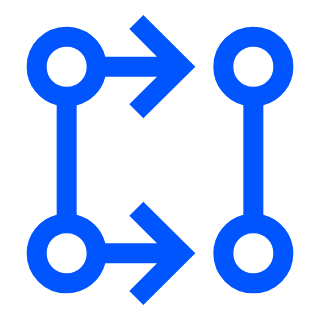
Devices and Sensors

IoT systems comprises of physical devices like smart appliances, sensors or actuators. These devices read and gather data and parameters from their environments, example of such data and parameters can be temperature, humidity, pressure etc.

Connectivity



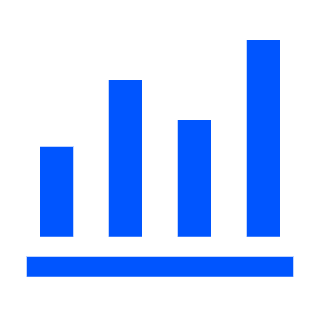
The IoT devices will need a medium for communication, such mediums can be physical (cables) or wireless (Bluetooth, Wi-Fi, Lora WAN, Cellular etc.)



Data Processing

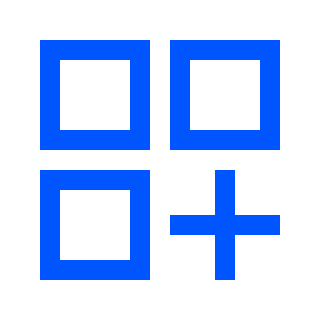
Once data is being collected from the IoT system. It is then processed locally on the device or sent to the cloud for more complex analysis and processing.

Cloud Storage and Analytics



Data received and collected from the IoT devices is stored on cloud servers which makes to safe for our data collected.

User Interface



IoT systems have user interface such as web apps, mobile apps or web dashboards where user can interact with the system hence monitor data, control devices and make decisions based on real-time information.

Applications of IoT

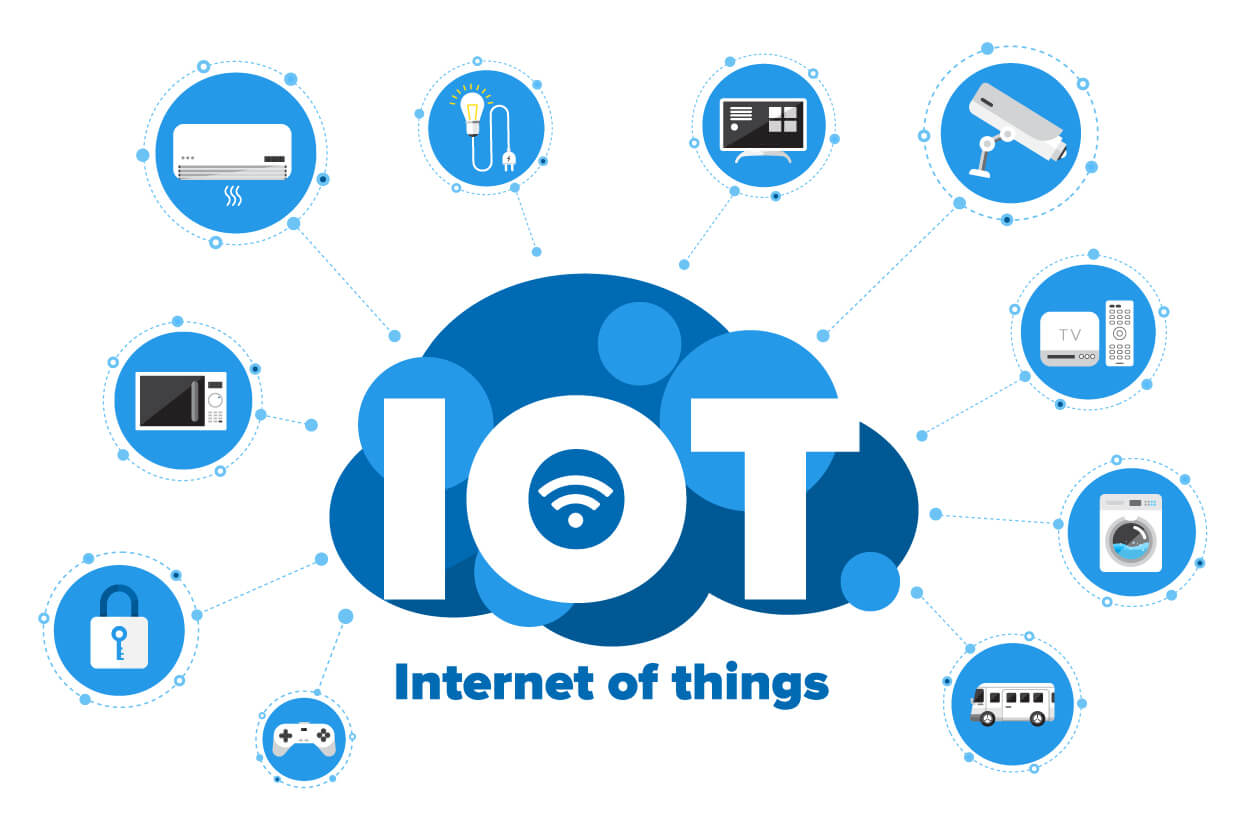
* Smart Homes: Devices like thermostats and lights adjust automatically.
* Healthcare: Wearables track health data like heart rate.
* Agriculture: Sensors monitor soil and weather to help farmers.
* Industry: Machines detect problems and report before they break.

Benefits of IoT

* Real-Time Insights and Monitoring : Provides instant data that can be acted and worked upon immediately, improving decision-making.
* Automation: Reduces human intervention by allowing devices to communicate and make decisions on their own hence reducing human error.
* Efficiency: Optimizes resource usage and operations, reducing costs and improving performance.
* Improved Quality of Life: Enhances convenience and safety in homes, healthcare, and transportation.

Future of IoT

* AI Integration: IoT systems with artificial intelligence (AI) will enhance predictive capabilities and automate decision-making with high precision.
* 5G Networks: With faster and more reliable connections, 5G will enable massive IoT deployments with low latency and high bandwidth.
* Edge Computing: Processing data closer to the source will reduce latency and improve real-time responses in critical applications like autonomous vehicles and smart cities.



Introduction GAIA board hardware

The GAIA board is a versatile IoT development board and platform designed to simplify the creation of IoT systems and also improve the rate at which data is being transferred and processed. The GAIA board comes in various variants with each variants performing specific tasks.

The GAIA Board hardware includes.

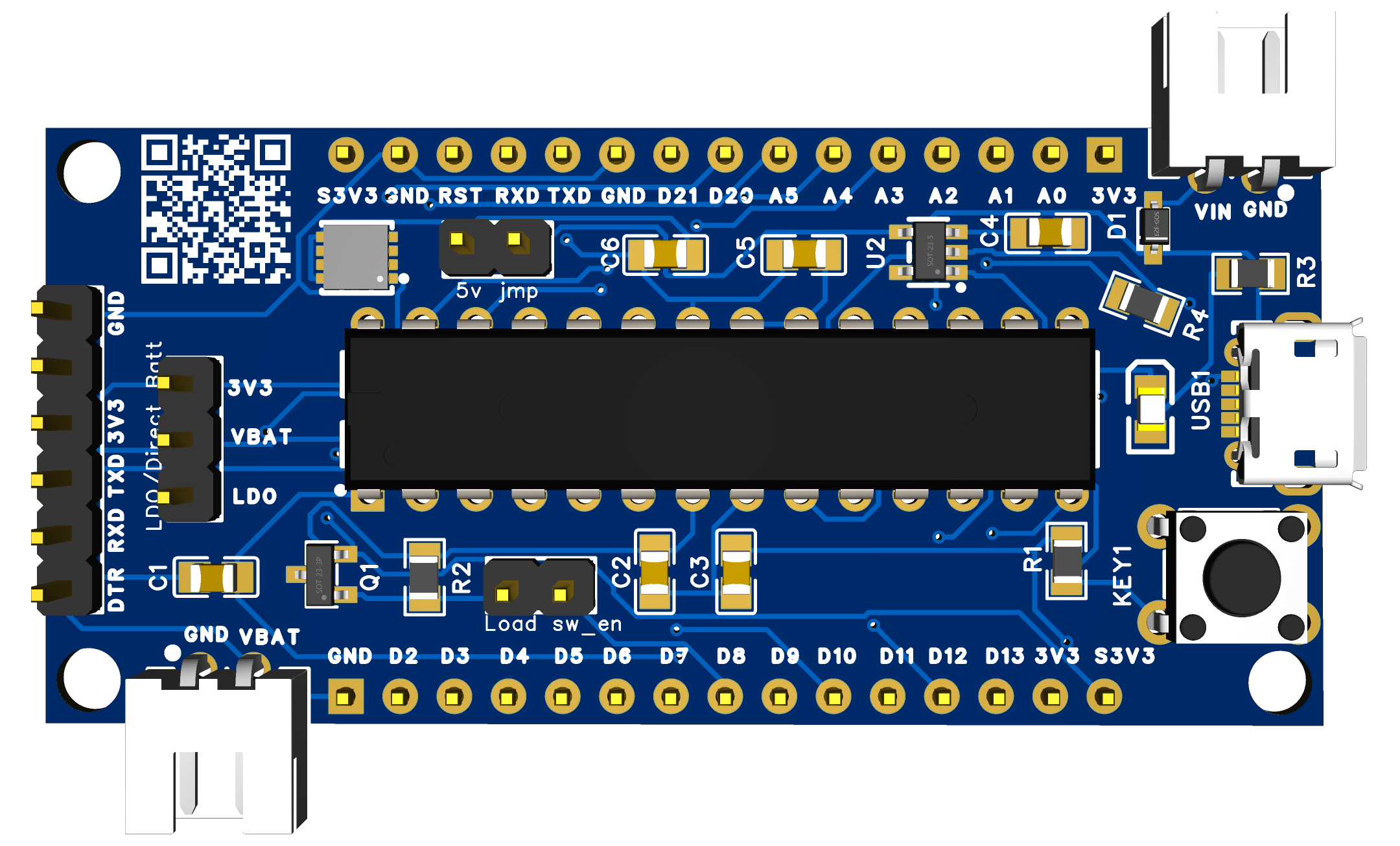
* ATmega328P: A low-power microcontroller commonly used in Arduino boards, ideal for managing sensors and simple control tasks.
* LoRa RFM96: A low-power, long-range wireless communication module used for IoT applications, enabling data transmission over several kilometers.
* GPS Module: Provides location data for tracking and navigation purposes.
* ESP Variant: ESP modules offer Wi-Fi capabilities, making it easy to connect devices to the internet for IoT communication and remote control.

The GAIA Board series consist of three(3) variants.

1. GAIA MCU 2. GAIA Basic Carrier Board 3. GAIA LoRa Reciever

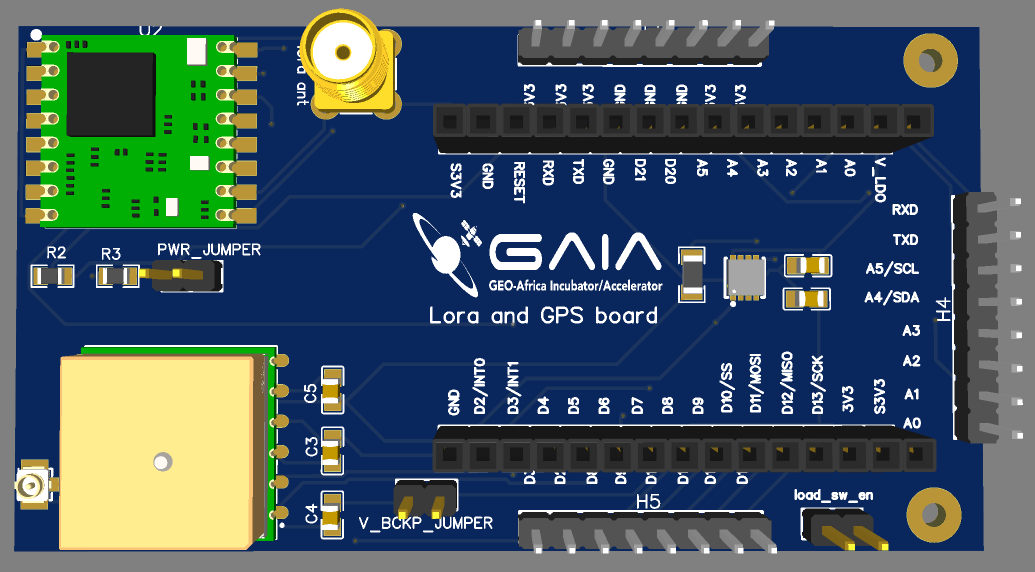
GAIA MCU (Micro Controller Unit)

A Micro Controller Unit is an integrated circuit(IC) that controls device operations and specific systems. The GAIA MCU incorporate a versatile microcontroller unit based on the ATmega328P, offering robust support for basic IoT applications and sensor integration.



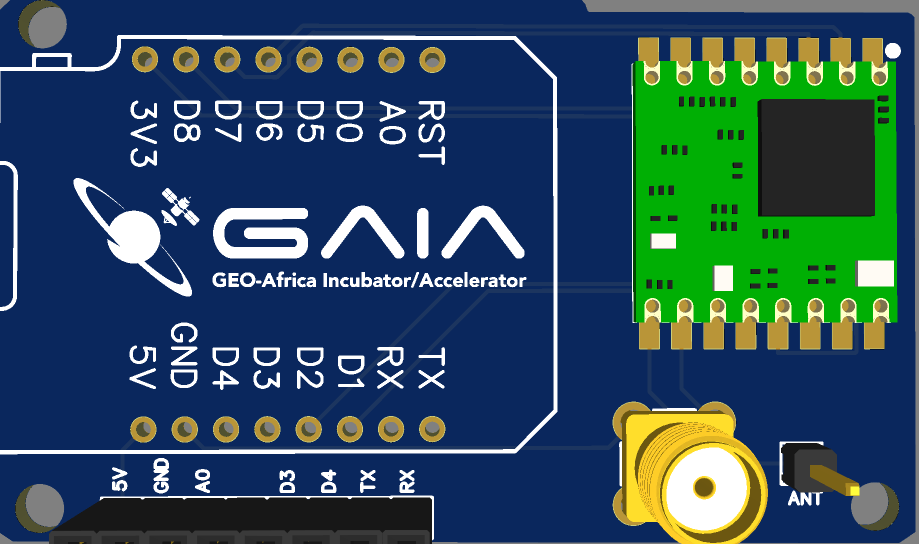
**GAIA Basic Carrier Board**

The GAIA basic carrier board is used to complement the [***GAIA Basic mcu***](https://www.gaiaclubs.org/boards/view/?document_group_id=66433e1397c63fefe8057acf) . It handles the peripheral connections and come with built in Lora transceiver and GPS module.



GAIA LoRa Receiver

This board is used to receive LoRa packets from the carrier board. It allows users to send the data received via LoRa to be transmitted to the internet using wifi (ESP8266 D1 mini).



For more information on the GAIA Boards kindly visit the GAIA CLUBS website for more insight : <https://gaiaclubs.org/boards>

Setting Up Development Environment:

Installing Arduino IDE and necessary Libraries.

Downloading and Installing the Arduino IDE.

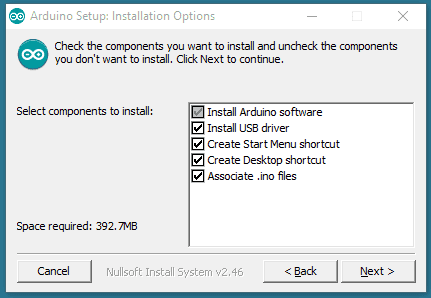
The Arduino IDE is an open-source software, which is used to write and upload code to the Arduino boards. The IDE application is suitable for different operating systems such as **Windows, Mac OS X, and Linux**. It supports the programming languages C and C++. Here, IDE stands for **Integrated Development Environment**.

You can download the latest Arduino IDE from their website with the link below:

<https://www.arduino.cc/en/software>

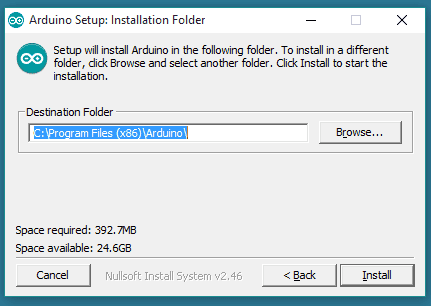
Installing the Arduino IDE

When the download is done , you can proceed to the installation of the software.

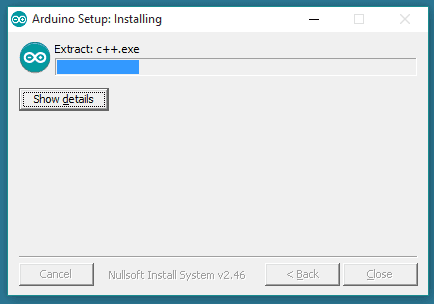


**1.**

**2.**



*Select the Installation Directory that suits you for storage.*



**3**.

*Installation in progress*

The installation process will extract and install the necessary files to properly execute the Arduino IDE .

Installing Libraries in Arduino IDE.

What are Libraries?

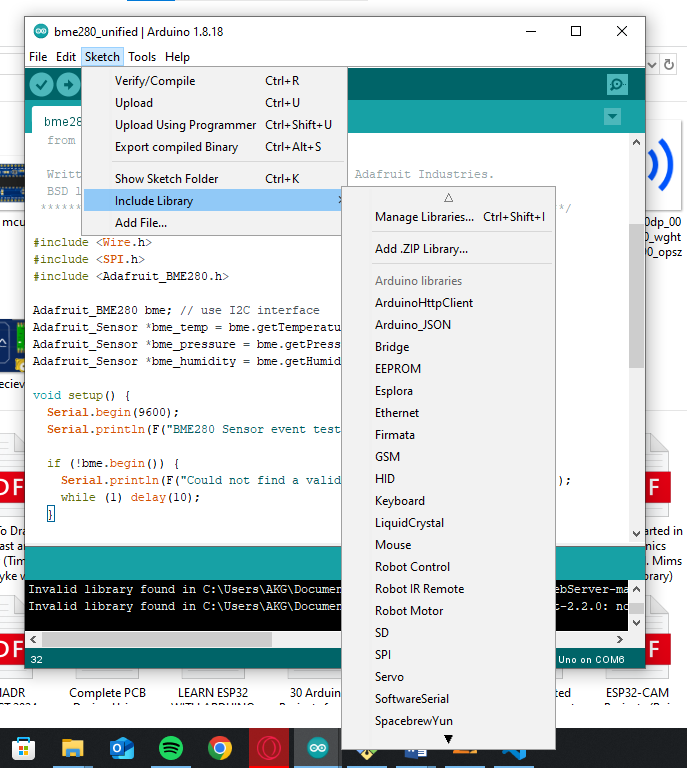
Libraries are a collection of code that makes it easy for you to connect to a sensor, display, module, etc. For example, the [Liquid Crystal library](https://www.arduino.cc/reference/en/libraries/liquidcrystal/) makes it easy to talk to character LCD displays.

There a numerous library which can be downloaded via the Arduino IDE.

Using the Arduino Library Manager.

You can Install any library of your choice using the Arduino Library Manager in the IDE.

 Open the IDE and click to the "Sketch" menu and then Include Library > Manage Libraries.



Here the Library Manager will open indicating list of libraries installed already or ready for installation. The Library Manager also makes it possible for library update and upgrade to keep your libraries up till date.

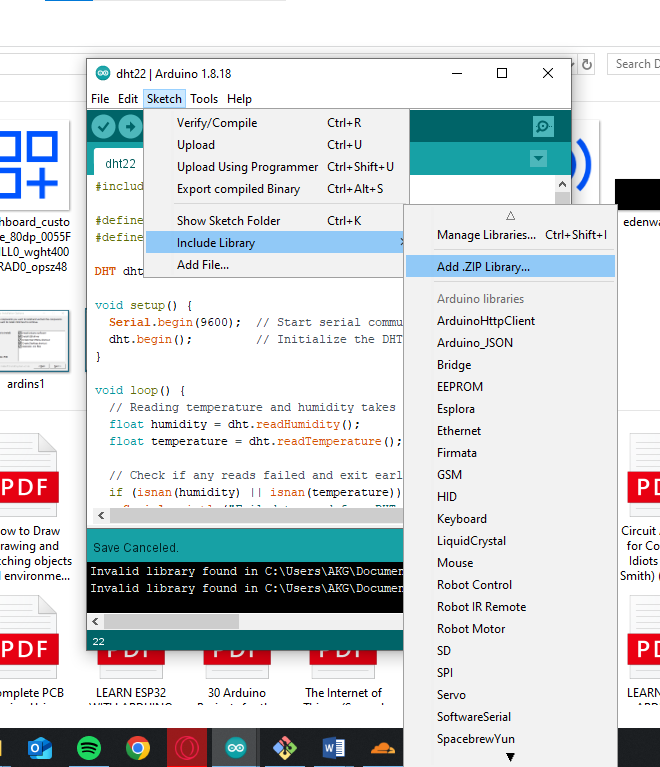
By using the search option in the manager, you can search for the desired library you require and install.

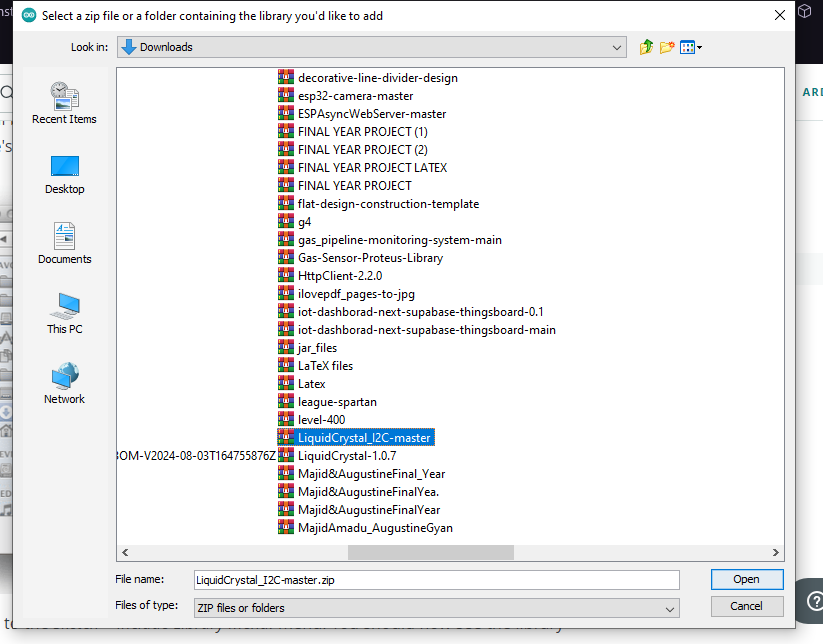
Scroll through the list to find what library you need and click on it, then select the version of library you want to install.

Finally click on install to install the library.

Importing a .zip library

Libraries are mostly distributed as a zip folder. Inside this folder is the name of the library. You can install third party libraries in the IDE. But kindly note that you are not supposed to unzip the zip file when doing the import.

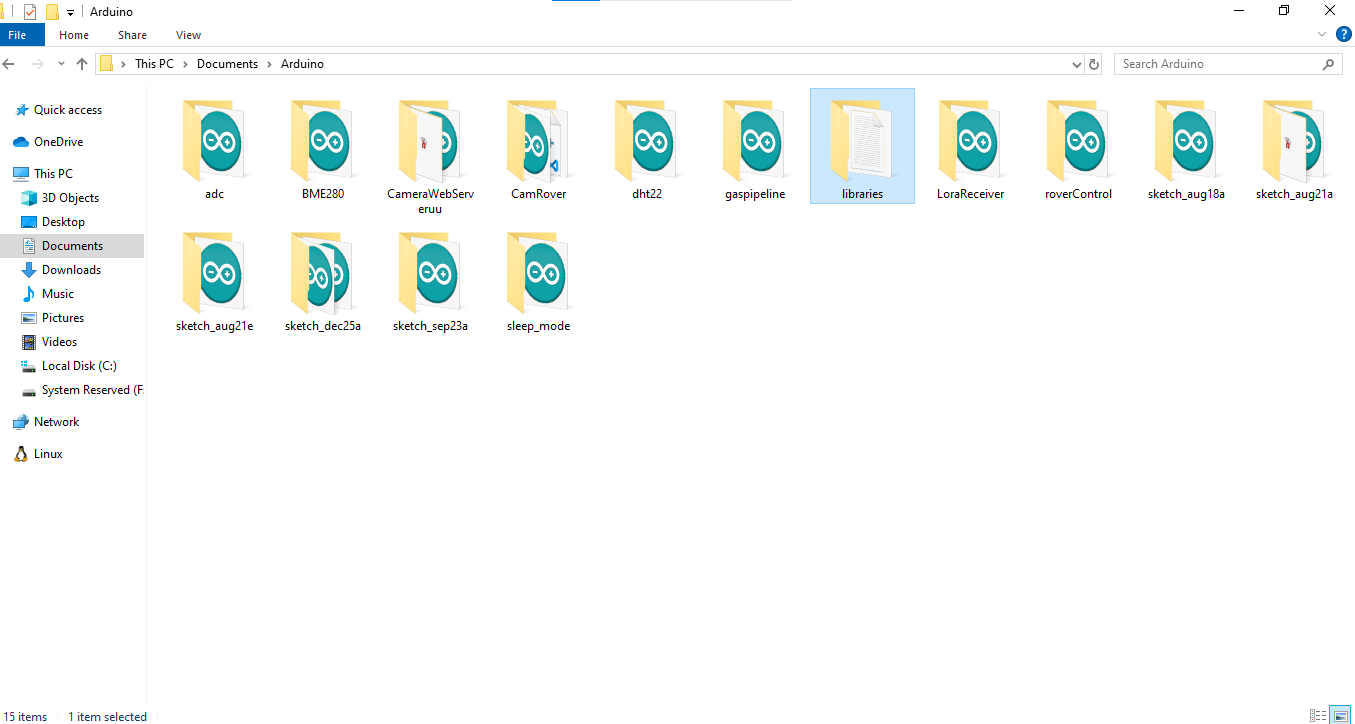


Here we will be importing the “LiquidCrystal\_I2C\_master” library into the IDE 

Adding Library Manually.

Libraries can be added manually by placing the required extracted file in the “Libraries” Folder in the Arduino system files located at the installation path we installed the IDE.

This is done by first extracting the file and copying the folder. After copying the folder, you paste it in the “Libraries” Folder.



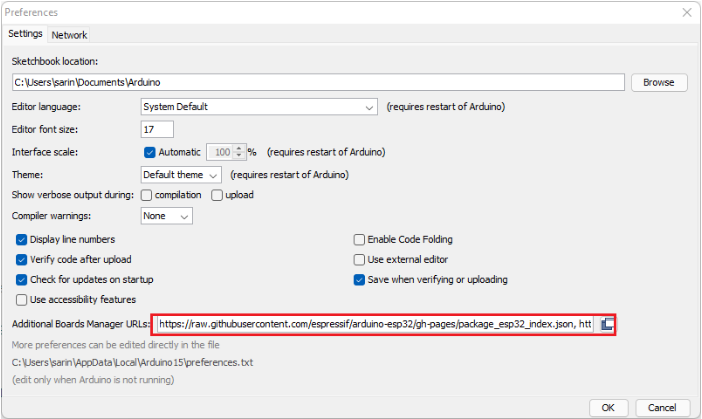
Setting up board support for ESP32

1 . In your Arduino IDE, go to **File**> **Preferences**

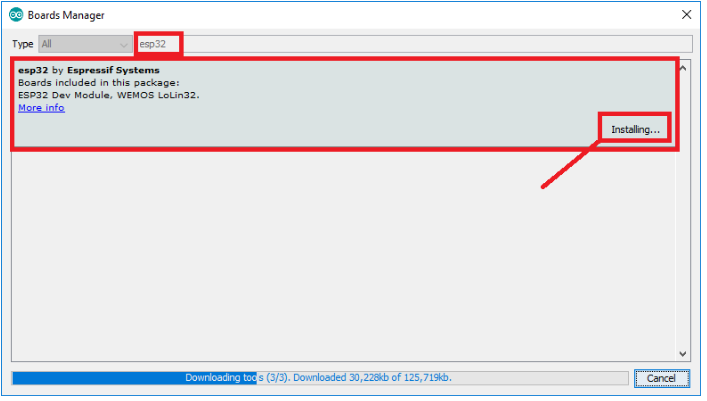
2. Enter the following into the “Additional Board Manager URLs” field:

https://raw.githubusercontent.com/espressif/arduino-esp32/gh-pages/package\_esp32\_index.json

Then click “OK” button



Open the Boards Manager. Go to **Tools** > **Board** > **Boards Manager…**



Search for **ESP32** and press install button for the “**ESP32 by Espressif Systems**“:

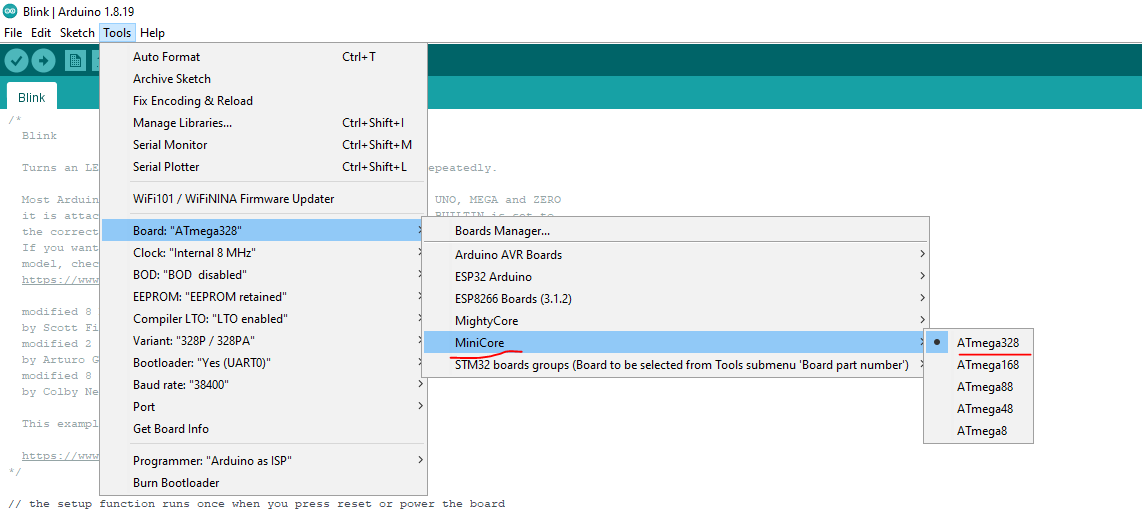
Setting up board support for ESP32

The following selections have to be made when uploading code to board when using the arduino IDE.

**1. Click on tools**

**2. Select MiniCore**

**3. ATmega328**



Basic Arduino Programming

Arduino Setup

The Arduino programming language is based on C/C++ and consists of two main functions:

setup(): Runs once at the start of the program to initialize variables, pin modes, etc.

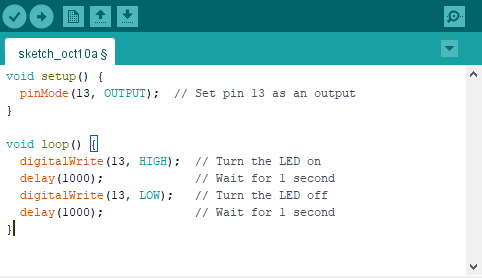
loop(): Repeats continuously after the setup, executing the main logic of your program.

What is Digital I/O?

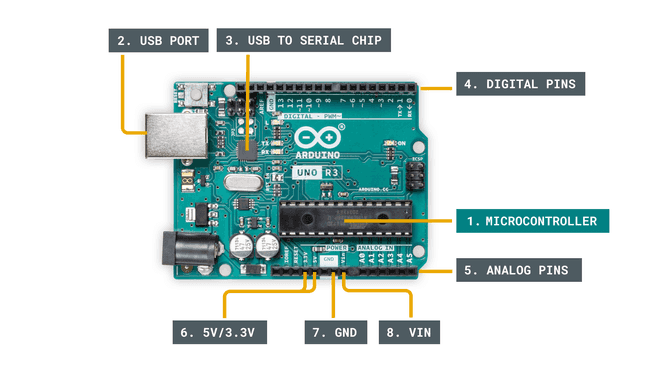
Digital Input allows the Arduino to read the state of a device, such as a button. The state can be either HIGH (5V) or LOW (0V).

Digital Output lets the Arduino control devices like LEDs or relays, turning them on or off by sending a HIGH (5V) or LOW (0V) signal to a pin.

Example Project: Blinking an LED



* pinMode(13, OUTPUT): Sets pin 13 to control an LED.
* digitalWrite(13, HIGH): Turns the LED on (5V).
* delay(1000): Waits for 1000 milliseconds (1 second).



Anatomy of an Arduino Board

What is Analog I/O?

Analog Input reads varying voltage levels, while analog Output (PWM) controls devices like motors or dimming LEDs.

Analog I/O is commonly used with sensors like potentiometers, temperature sensors, etc.

Analog Output pulse-width modulation (PWM) to simulate varying voltages on output pins.

Example Project: Reading a Potentiometer



analogRead(A0): Reads the analog value from pin A0 (ranging from 0-1023).

Serial.println(sensorValue): Prints the sensor value to the serial monitor for observation.

The potentiometer can be turned to change the voltage, which will change the reading in the serial monitor.

Summary:

pinMode(): Configures a pin as input or output.

digitalWrite(): Writes a HIGH or LOW signal to a pin.

digitalRead(): Reads the state (HIGH/LOW) of a digital input.

analogRead(): Reads an analog input (value from 0-1023).

analogWrite(): Sends a PWM signal to simulate analog output.

Serial.begin(): Starts serial communication.

Serial.println(): Prints data to the Serial Monitor.

Introduction to Sensors.

Sensors are essential components in IoT systems because they collect data from the environment. Sensors are sophisticated devices which will detect and measure any non-electrical physical quantity. A sensor converts physical parameters(e.g. temperature, blood pressure, humidity , speed , etc.) into a signal which can be measured electrically.

Sensors typically fall into one of three categories:

**Analog Sensors**: These sensors provide continuous signals or values, usually in the form of voltages. The values change gradually based on environmental factors. Examples include light-dependent resistors (LDRs), potentiometers, and temperature sensors like thermistors.

Example: A light sensor (LDR) changes its resistance based on the light intensity. The Arduino can read this changing voltage and convert it to a value that represents the light level.

Digital Sensors: These sensors provide a discrete or on/off signal (either HIGH or LOW). They often use binary signals to indicate if something is detected or not.

Example: A push-button or PIR motion sensor is digital. The sensor outputs a HIGH signal if motion is detected or the button is pressed, and LOW otherwise.

I2C Sensors: These sensors use a specific communication protocol called I2C (Inter-Integrated Circuit). This protocol allows multiple sensors to share the same data lines to communicate with a microcontroller using a unique address for each sensor. It is highly useful in systems where you need to connect multiple sensors without using too many pins.

Example: A BME280 environmental sensor uses I2C communication to measure temperature, humidity, and pressure. It communicates over just two wires (SDA and SCL) to send data to the microcontroller.

Sensor Interfacing

Analog Sensor Interfacing:

With Analog Interfacing we connect the sensor output to an analog pin on the Arduino (e.g., A0).

For Reading we use the analogRead() function to read the voltage level, which returns a value between 0 and 1023. This value corresponds to the input voltage (0V to 5V).e.g (Basic LDR reading using the analogRead()

Digital Sensor Interfacing

With Digital Interfacing we connect the sensor output to a digital pin (e.g., pin 2).

Reading: Use the digitalRead() function to check the state of the pin. It will return HIGH (1) if the signal is present and LOW (0) if it is not.

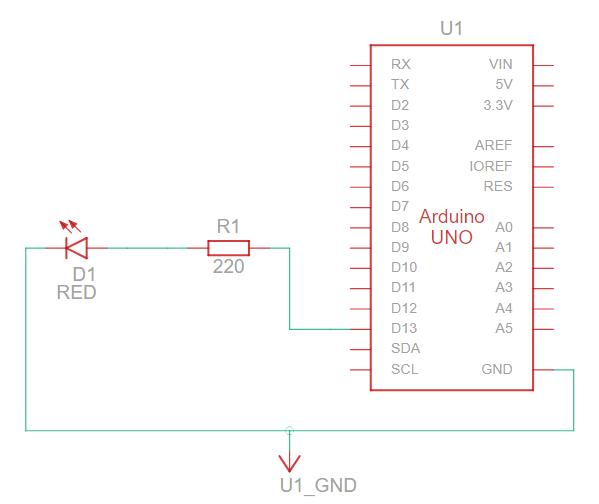
Example: A push button can be connected to a digital pin. When the button is pressed, it sends a HIGH signal; when released, it sends a LOW signal.

I2C Sensor Interfacing:

The sensor is connected to the SDA (data line) and SCL (clock line) pins on the Arduino. This allows for communication with multiple I2C devices.

Example: The BME280 sensor, which measures temperature, humidity, and pressure, communicates via I2C. You can read all three measurements using a single set of wires.

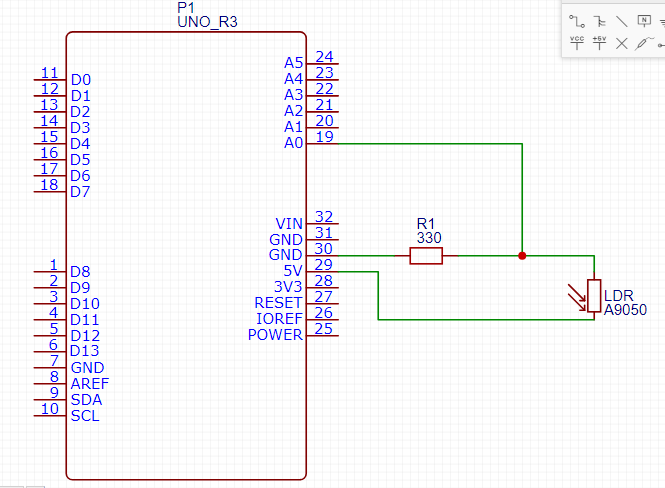
Hands on Activities

a. Simple LED Blinking and serial output project

Get code for all IoT projects via the github link below:

<https://github.com/augustinegyan/Gaia_Iot_Codes/>

b. Reading an Analog sensor

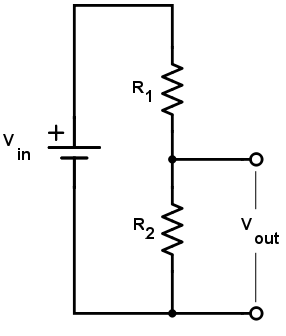
Here we will be using the LDR (Light Dependent Resistor) to demonstrate the readings

SECTION 2

Advanced Sensor Integration and Power Management

Voltage Dividers

Voltage Dividers are a series of resistors or capacitors which can be tapped at any intermediate point to produce a specific fraction of the voltage applied between its ends.

 Purpose ?

The primary purpose of this circuit is to scale down or divide the input voltage to a lower value based on the ratio of the two resistors.

**Equation**

Vout = Vin x

**Where**

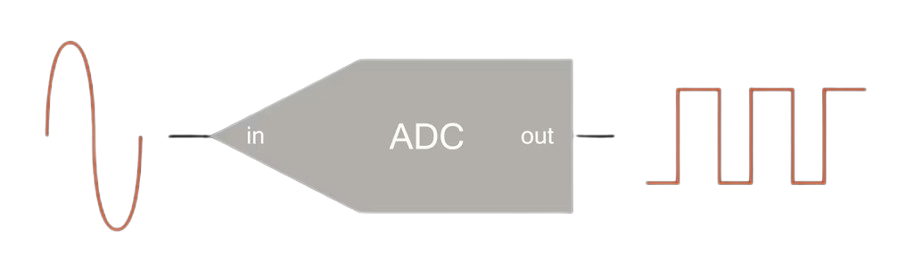
V**out**=Output Voltage(Scaled down Voltage. V**in**=Input Voltage.

R1 and R2 are resistor values in ohms.

**Get more insight on Voltage Dividers from this link below**

[https://www.allaboutcircuits.com/tools/voltage-divider-calculator/](https://www.allaboutcircuits.com/tools/voltage-divider-calculator/%20%20)

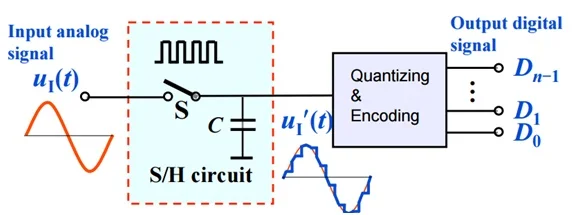
ADC (Analog-to-Digital Converter)

An Analog-to-Digital Converter (ADC) is a device that converts an analog signal (continuous values) into a digital signal (discrete values). In the context of Arduino, the ADC is used to read analog sensor values, such as temperature, light, or pressure, which are often represented as varying voltages. The ADC translates these voltage levels into numerical values that the microcontroller can process. Example is the ADS1115 16-bit ADC

How ADC Works

Analog Signal: This is the input voltage you want to measure. Analog signals can vary continuously over time, like the changing brightness of light on an LDR (light-dependent resistor).

Digital Signal: The output of the ADC is a discrete number that represents the analog signal's level. For example, in an Arduino, the ADC will convert an input voltage (between 0V and 5V) into a digital value between 0 and 1023 (in the case of a 10-bit ADC).

The Process:

Sampling: The ADC takes snapshots of the analog signal at regular intervals.

Quantization: The ADC divides the signal into discrete levels (steps). For a 10-bit ADC (used in most Arduino boards), there are 1024 levels (from 0 to 1023).

Conversion: The analog value is then converted into a corresponding digital value. For instance, if you have a 5V input range, an analog voltage of 2.5V would be approximately converted to the midpoint of the range, which is 512 (half of 1023).

Why converts Analog Signals to Digital Signals ?

* Digital devices can't directly read analog signals like temperature or sound. So, we convert analog signals to digital so that devices like Arduino or computers can understand and process them.
* Less Noise and Interference: Analog signals can pick up noise or interference, making them less reliable. Digital signals (made up of 0s and 1s) are more resistant to these issues.
* Easier to Store and Process: Digital data can be stored in memory (like MP3 music or JPEG images) without losing quality.
* Precise and Reproducible: Digital systems can reproduce the same data exactly, without small variations that happen in analog systems. This ensures consistent performance and are easy to manipulate.

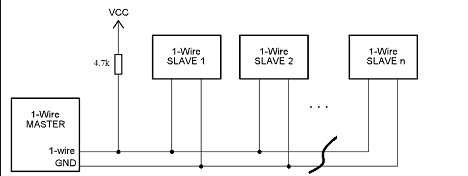
Digital Protocols

Digital protocols are standardized rules and methods that electronic devices use to communicate with each other. These protocols define how data is transmitted, ensuring that devices can exchange information correctly and efficiently. Below are examples common digital communication protocols:

* I2C(Inter-Integrated Circuit)
* **SPI (Serial Peripheral Interface)**
* **UART (Universal Asynchronous Receiver-Transmitter)**
* One-Wire
* **CAN (Controller Area Network)**
* **Ethernet**
* Wi-Fi

Here we will focus mostly on One-Wire and I2C .

One-Wire is a communication protocol developed by Dallas Semiconductor (now Maxim Integrated) that allows devices to communicate using just a single data wire (plus a ground connection). It's designed for simple, low-speed communication between a master device (e.g., a microcontroller) and one or more slave devices (e.g., sensors).

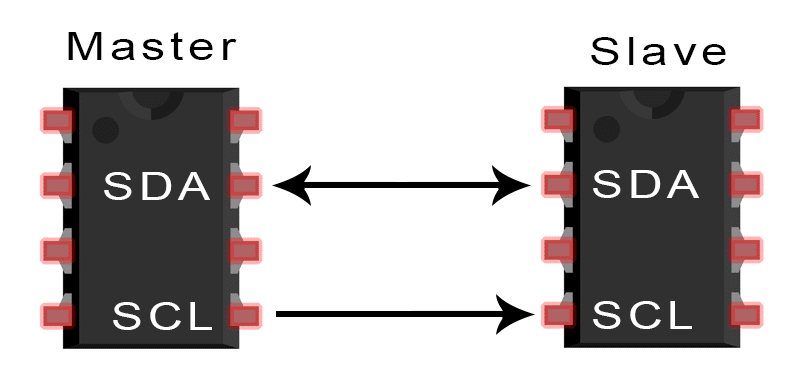


How One-Wire Works:

* Data Transmission: Communication happens by the master sending a signal (typically a voltage change) over the data line. The slaves listen and respond accordingly.
* Addressing: Each One-Wire device has a unique 64-bit identifier, allowing the master to select which device to communicate with.
* Bit Timing: One-Wire communicates by pulling the data line low for different amounts of time to represent bits (1 or 0). This is controlled through a process of timing, where both the master and the slaves are synchronized.

Learn more about One-Wire here:  [https://en.wikipedia.org/wiki/1-Wire](%20https://en.wikipedia.org/wiki/1-Wire)

I2C (Inter-Integrated Circuit)

I2C is a communication protocol used to connect multiple devices (sensors, displays, etc.) to a microcontroller using just two wires: SCL (clock) and SDA (data). It’s widely used in embedded systems for short-distance communication between devices.

Features of I2C Protocol

* Two-Wire Communication

SDA (Serial Data Line) is used to send and receive data.

SCL (Serial Clock Line) is used to synchronize data transfer between devices.

* Master-Slave Architecture

A single master controls communication. Multiple slave devices (e.g., sensors) can be connected to the same bus.

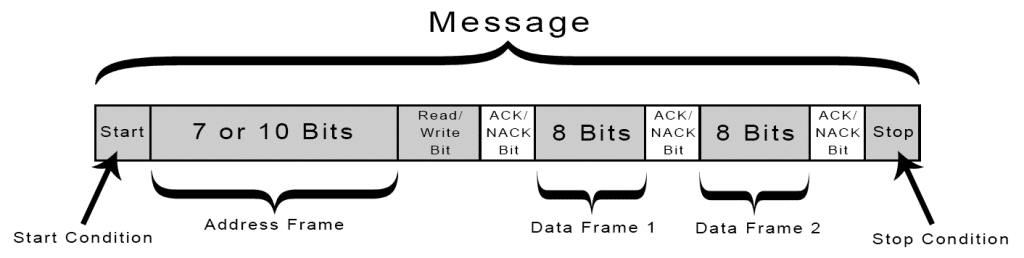
* Addressing

Each slave device on the I2C bus has a unique 7-bit or 10-bit address.

The master uses these addresses to communicate with the specific slave.

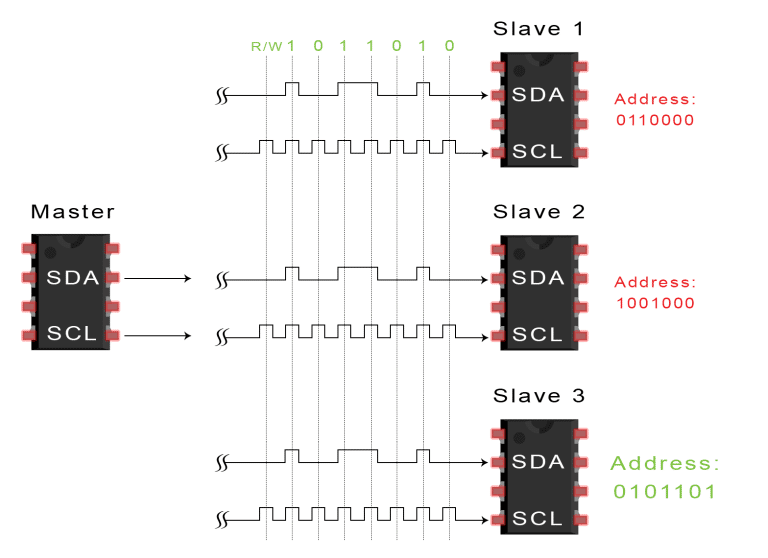
* Speed

Standard I2C operates at speeds up to 100 kHz (Standard Mode).

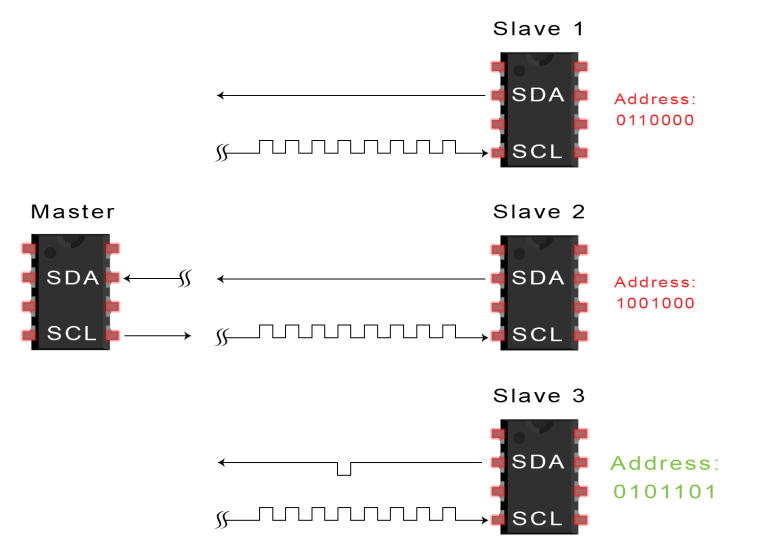
Fast Mode runs at 400 kHz, and some implementations allow even higher speeds.

How I2C Works

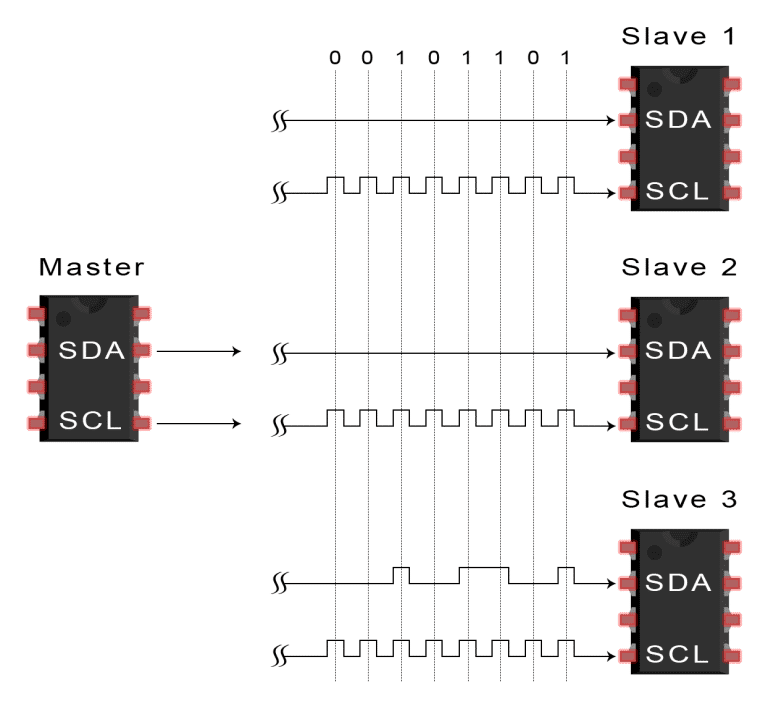
1. Start Condition: Communication starts when the master pulls the SDA line low while SCL is high (indicating the start of a communication frame).
2. Addressing: The master sends the 7-bit address of the slave it wants to communicate with, followed by a read/write bit.
3. Data Transfer: After the address, the data is transferred in 8-bit chunks, with each byte being acknowledged (ACK) or not acknowledged (NACK) by the receiver.
4. Stop Condition: The communication ends when the master releases the SDA line (allows it to go high) while the SCL line is high.

Example to illustrate I2C protocol

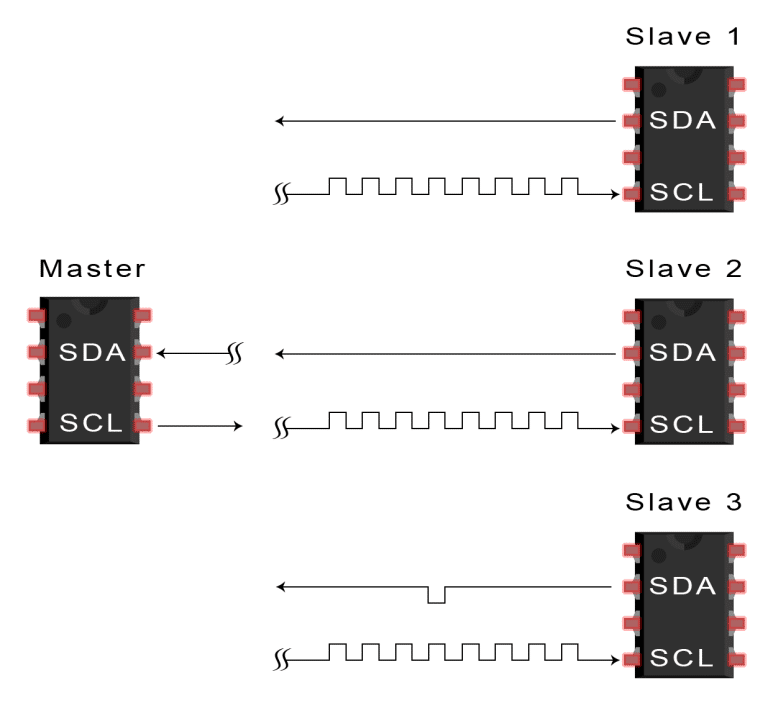
The master sends each slave the 7 or 10 bit address of the slave it wants to communicate with, along with the read/write bit:



 Each slave compares the address sent from the master to its own address. If the address matches, the slave returns an ACK bit by pulling the SDA line low for one bit. If the address from the master does not match the slave’s own address, the slave leaves the SDA line high.



The master then sends or receives the data frame to the desired destiny that is Slave 3 per the diagram :



After each data frame has been transferred, the receiving device returns another ACK bit to the sender to acknowledge successful receipt of the frame:

Read more about I2C here : <https://www.circuitbasics.com/basics-of-the-i2c-communication-protocol/>

Wire Library Basics

The Wire Library makes it possible to work with I2C protocol in our Arduino IDE.

Wire Library Functions:

1. Wire.begin():

Initializes the I2C bus.

For a master: Call Wire.begin() without arguments.

For a slave: Call Wire.begin(address) where address is the 7-bit I2C address of the slave device.

2. Wire.requestFrom(address, quantity):

Used by the master to request data from a slave.

address: The I2C address of the slave device.

quantity: The number of bytes to request from the slave.

3. Wire.beginTransmission(address):

Begins communication with a slave.

address: The 7-bit I2C address of the slave device.

4. Wire.write(data):

Sends data to the slave. The data can be a single byte or an array of bytes.

Must be used after Wire.beginTransmission() and before Wire.endTransmission().

5. Wire.endTransmission():

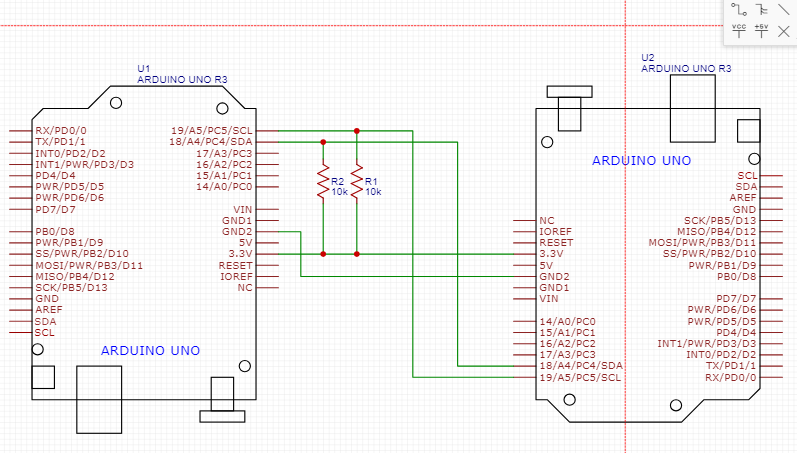
Ends the transmission and sends the data to the slave.

6. Wire.read():

Reads a byte of data from a slave (used after Wire.requestFrom()).

7. Wire.available():

Returns the number of bytes available for reading after a Wire.requestFrom() call.

Basic illustration of I2C with two Arduino boards, one being and slave and the other the master

SLAVE

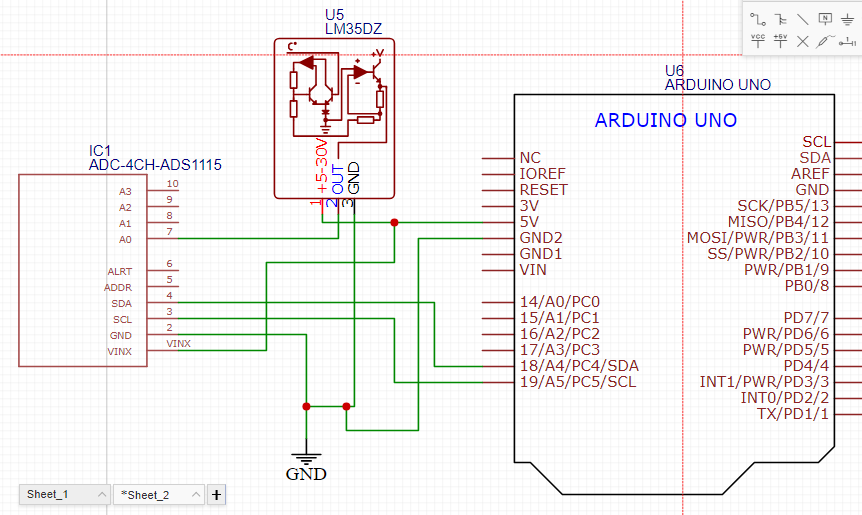
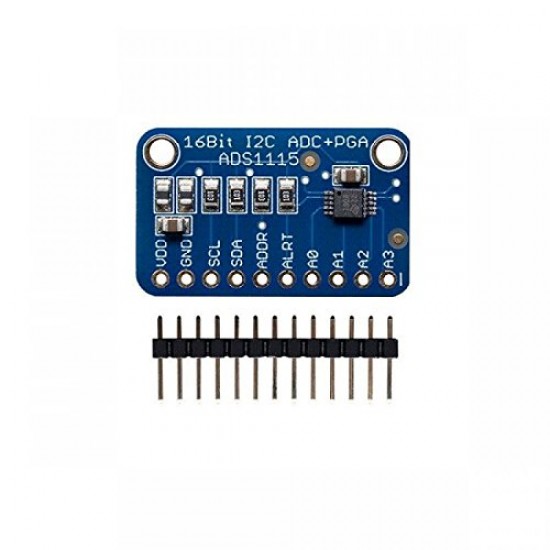
MASTER

With this illustration the “Master” endpoint will send the “Slave” a message with indication “Hey Slave” and the Slave respond by giving “Hello Boss” to indicate a successful command from the master to slave.

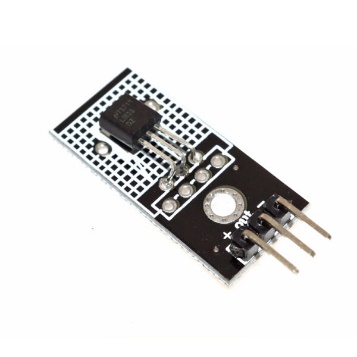
Link to the code for both master and slave. <https://github.com/augustinegyan/Gaia_Iot_Codes/tree/master/I2C_Comm>

Upload each code and run on each Arduino separately.

Working with ADC’s(Analog-Digital Converter)

We will be converting analog signals from a LM35DZ temperature sensor to digital sensor to our microcontroller for accurate readings since Analog signals are prone to noise.

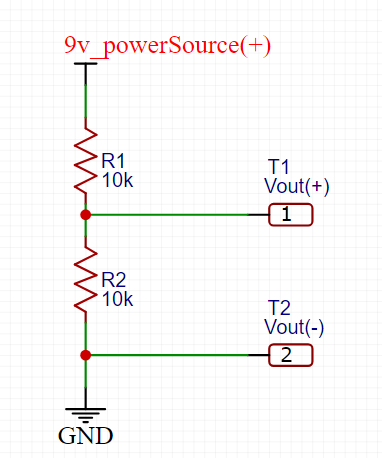
ADS1115 ADC



LM35D7 Analog Temperature Sensor

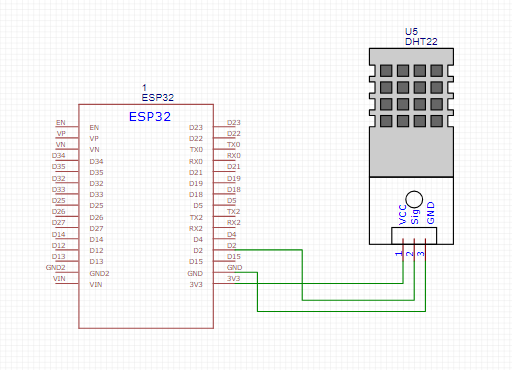
Link to code : <https://github.com/augustinegyan/Gaia_Iot_Codes/tree/master/adc>

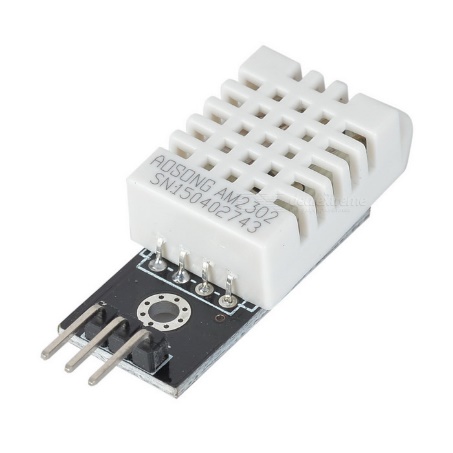
Hands-on Sensor Projects:

a. Working with Voltage dividers.

With this use case , we input a source power of 9 volt into the voltage divider. We are to receive an appropriate division of the power source to half of its value. T1 and T2 indicate the V out , hence we get to know the value across R2 to justify our outcome.

b. Reading data from a DHT22 sensor.

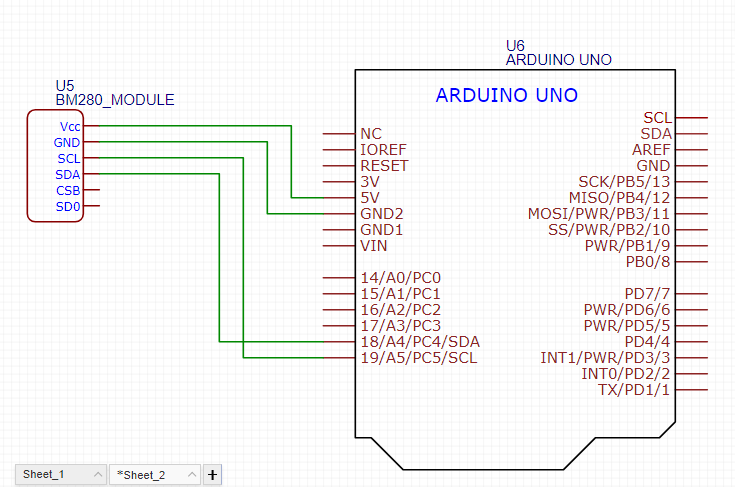
The DHT22 is a low-cost digital sensor used to measure temperature and humidity. It is widely used in various IoT projects, weather monitoring systems, and home automation because of its simplicity and accuracy. Here we will be reading humidity and temperature with the help of an ESP32 to get the values on our serial monitor



DHT22 Sensor

Link to Code : <https://github.com/augustinegyan/Gaia_Iot_Codes/tree/master/dht22>

c. Using BME280 for environmental monitoring.

The BME280 is an advanced sensor that measures temperature, humidity, and barometric pressure. It is widely used in weather monitoring, environmental sensing, and Internet of Things (IoT) applications because of its high accuracy and low power consumption.



BME280 Sensor

Power Management and Sleep modes

Sleep modes are designed to conserve power by reading its processes or activities or even shutting down completely certain components when they are not needed. This is very useful and essential for battery powered or energy efficient projects.

There are multiple sleep modes when working with modern microcontrollers like Arduino , Esp32 etc. Below are some of the sleep modes.

Active Mode

The microcontroller operates at full power with all systems such as CPU , WiFi, Bluetooth , Senor etc. active and functioning.

Use Cases : Live video streaming or real-time data process where the microcontroller needs to be active (ESP32-CAM). For project where performance is prioritized over power consumption.

Modem-Sleep Mode

CPU stays active but WiFi, Bluetooth go into a lower-power state. The microcontroller can turn of WiFi when its not needed in other to save energy .

Use Cases: Periodic data transmission where the device process information locally and ocassionally needs to communicate to a server or the internet. Example Smart Thermostat : The microcontroller monitors temperature continuously but only checks or sends data over WiFi occassionaly . In Modem-Sleep , the WiFi is turned off but can quickly wakeup to transmit data , making it good for projects needing intermittent connectivity.

Light-Sleep Mode

CPU, the parts and the main memory is placed in a low-power mode state. However the WiFi remains connected in a lower power state. This allows the microcontroller to maintain connectons to routers or cloud while saving energy .

Use Cases: Wearable, environmental sensor- Battery powered devices that require consistent communication. Example BME280 reads and sends data from the microcontroller and sends to the cloud every few minutes.

Deep-Sleep Mode.

The microcontroller turns off almost everything including CPU and most peripherals, but retains the RTC(Real-Time Clock) to track time and wakeup based on timers or external triggers.

Use Cases: Example Data loggers or IoT devices that only wake up periodically to take a measurement or send data. Example Weather Station whereby the microcontroller wakes up every 15 minutes to read sensor data or goes back to sleep

Power-Saving Techniques for Microcontrollers

Sleep Modes

Put the microcontroller in a low-power state when it’s not actively doing tasks. The microcontroller can “wake up” when needed, such as on a timer or external event (e.g., button press).

For Arduino: Sleep modes like "Power-down" drastically reduce power consumption by turning off most parts of the chip.

For ESP32: "Deep Sleep" mode reduces power significantly by turning off almost everything except memory and specific wake-up sources (like timers or GPIOs).

Disable Unnecessary Peripherals

If you’re not using certain features (like the Wi-Fi, Bluetooth, or ADC modules), turn them off to save energy.

For Arduino: You can turn off modules like analog-to-digital converters (ADCs).

For ESP32: You can turn off Wi-Fi, Bluetooth, and other modules to save power.

Wake-up on Interrupts

Use hardware interrupts to wake the microcontroller from sleep when needed. For instance, the microcontroller can be asleep and wake up only when a button is pressed, a sensor value changes, or an external signal is detected.

This ensures the microcontroller stays in sleep mode until necessary.

Reduce Clock Speed

Lowering the microcontroller's clock speed reduces power consumption, though it may slow down the processing. This is useful when full performance isn't required.

Power Control for Sensors/Components

Power off sensors, modules, or LEDs when they are not needed. You can control the power supply to these components via transistors or other switches, reducing power usage significantly when components are idle.

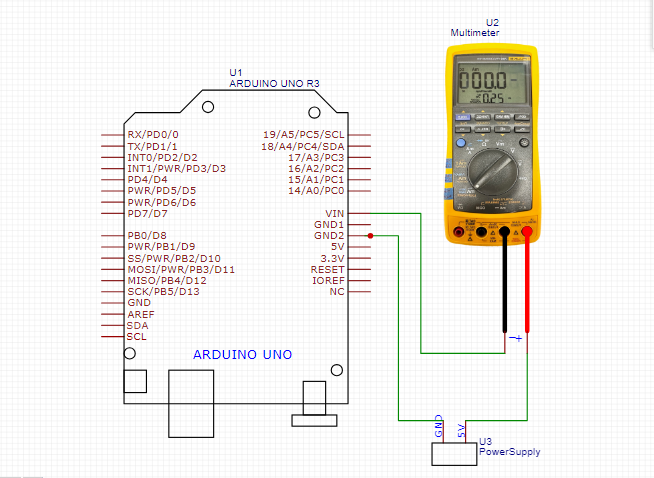
Limit Active Time

Minimize how often the microcontroller stays awake. For example, set it to only wake up periodically (every few minutes) to check sensor readings, and then go back to sleep. This dramatically reduces power consumption compared to staying active constantly.

Use Efficient Components

Choose components that are designed for low-power consumption, like energy-efficient sensors and displays (e.g., E-ink or OLED displays that use minimal power when static).

Measuring Power Consumption to Optimize battery life.

To measure power management in sleep mode on an Arduino or ESP32, a **practical approach** involves using a **multimeter** to measure current consumption before and during sleep mode. This allows you to see the difference in power usage and justify the effectiveness of the power-saving modes. Below is the schematic of the illustration

We connect the multimeter in series to measure the current drawn by the board in series with the power supply. One probe connects to the positive side of the power source, and the other probe to the input pin (VIN or 5V pin) of the microcontroller.

Alternatively, you can power the board through the VIN or 3.3V pin, depending on whether you're using an Arduino or ESP32.

We then upload the sketch to the microcontroller to go into sleep mode over a certain interval of time and back to normal time . Tune the multimeter to the milliamps reading and monitor the outcome as the microcontroller goes through normal and sleep mode.

Get code/sketch here : [SleepMode\_Codes](https://github.com/augustinegyan/Gaia_Iot_Codes/tree/master/sleepmode)

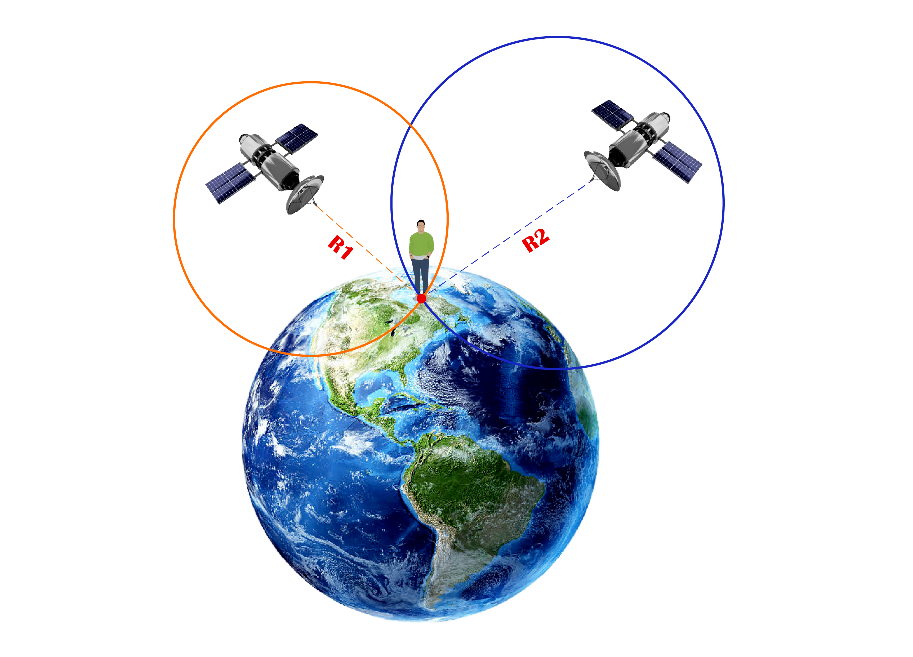
1. Measure Current in Normal Mode note down the reading in milliamps
2. Measure Current in Sleep Mode
3. Compare Results. You can check the battery life using this website below

<https://www.allaboutcircuits.com/tools/battery-lifetime-calculator/>

HANDS ON ACTIVITIES (Will be done soon)

SECTION 3

GPS Module Integration and Introduction to LoRa Technology

Introduction to GPS Technology

**GPS (Global Positioning System)** is a satellite-based navigation technology that provides real-time location, velocity, and time data to a GPS receiver anywhere on Earth. Developed by the U.S. Department of Defense, GPS has become a widely used technology for civilian applications.

GPS operates using a network of satellites that transmit signals to Earth. A GPS receiver uses signals from at least four satellites to calculate its precise location by triangulation. The data provided by GPS systems include latitude, longitude, altitude, and time.

 How GPS Works

* Satellites: There are about 31 GPS satellites orbiting Earth, transmitting radio signals.



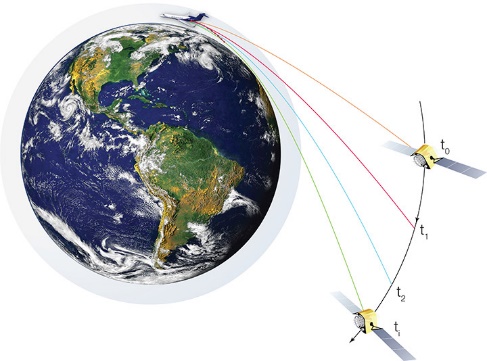
* GPS Receiver: The receiver detects signals from multiple satellites and uses the time delay to calculate its position.



* Triangulation: By calculating the distance from at least four satellites, the receiver determines its exact location (latitude, longitude, altitude).
* Time Synchronization: GPS also provides precise time information based on atomic clocks in satellites, which is critical for synchronizing networks.

GPS Applications in IoT

In the context of IoT, GPS technology plays a significant role in **location-based services, tracking systems,** and **remote monitoring.**

* **Asset Tracking & Fleet Management**: GPS helps track vehicles, shipments, and valuable assets in real-time for better route planning and security.
* **Smart Agriculture:** Farmers use GPS to monitor crops, track equipment, and manage livestock, improving efficiency and yield.
* **Wearables:** Fitness trackers and safety wearables use GPS to monitor movement and provide real-time location data.
* **Autonomous Vehicles & Drones:** GPS guides self-driving cars and drones, allowing precise navigation and route mapping.
* **Smart Cities:** GPS helps manage traffic, locate emergency services, and optimize public transportation.
* **Environmental Monitoring:** Track wildlife, monitor air quality, and tag environmental data with GPS for analysis

Connecting the GPS module to ATmega328P and using TinyGPS+ library to parse GPS data

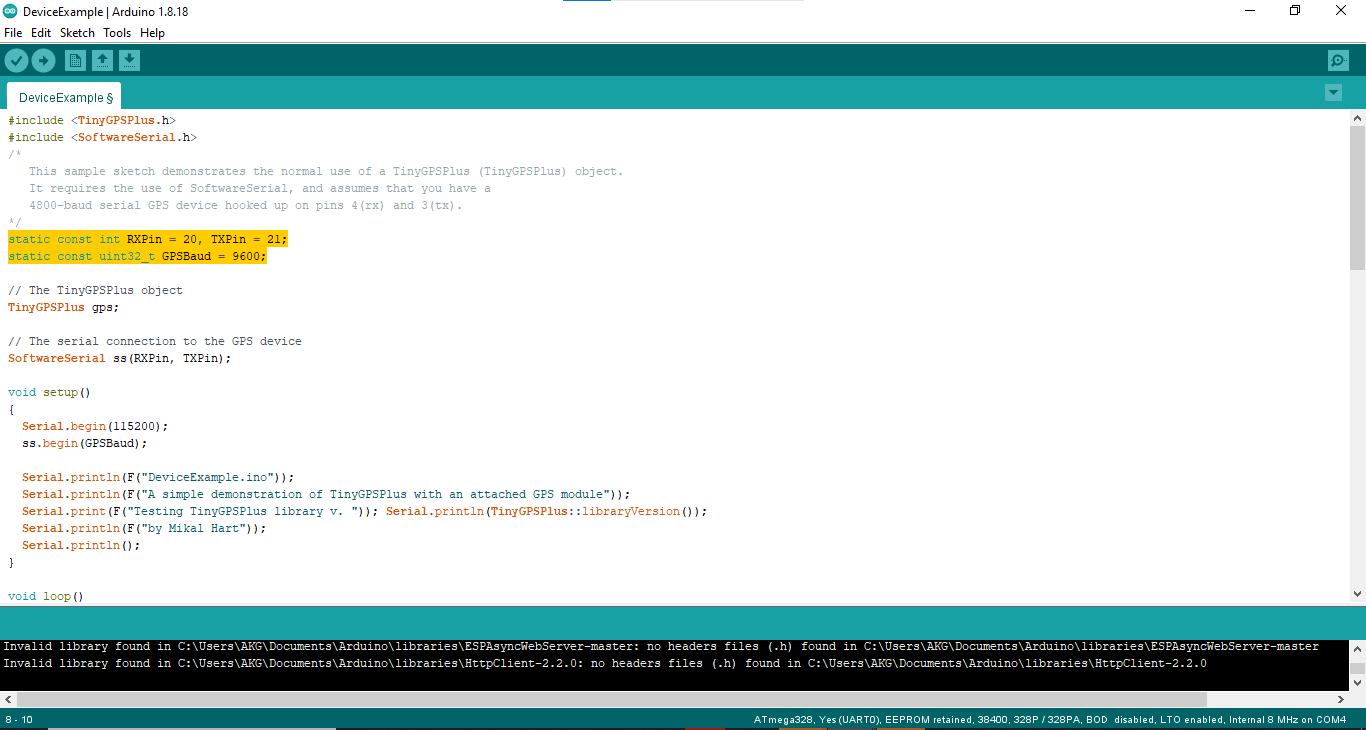
With this demonstration we will be using the Gaia Carrier board with the Gaia MCU board. The Gaia Carrier Board already has a GPS module for GPS functionalities embedded on the board.

To connect a GPS module to an **ATmega328P** microcontroller and use the **TinyGPS+** library to parse GPS data, follow these steps:

Using the TinyGPS+ Library

* **TinyGPS+** is a library that helps decode the GPS data (like latitude, longitude, altitude) from the NMEA sentences transmitted by the GPS module.

#### Steps to Set Up TinyGPS+:

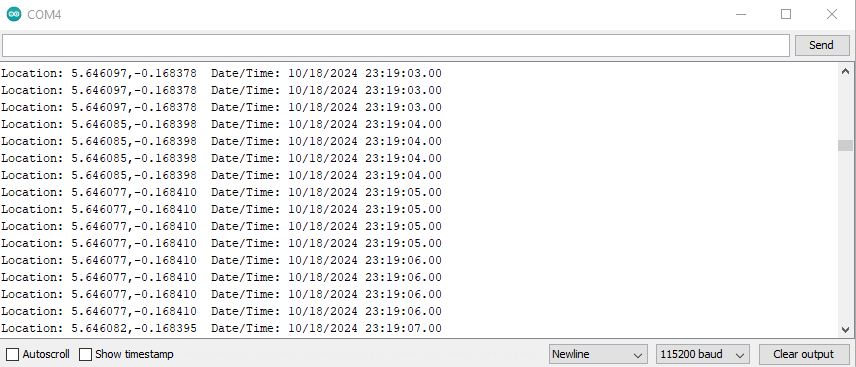
1. Open **Arduino IDE.**
2. Go to **Sketch > Include Library > Manage Libraries** and search for **TinyGPS+.** Install it.
3. Use the **SoftwareSerial** library to set up a software-based serial port for communication with the GPS module if you're not using the hardware serial pins.
4. Go to File > Examples > TinyGPSPlus > DeviceExample.

Kindly Note: Change the parameters in code for the Device Example

* Set the *RXPin = 20 and TXPin = 21.*
* Set the *GPSBaud = 9600.*

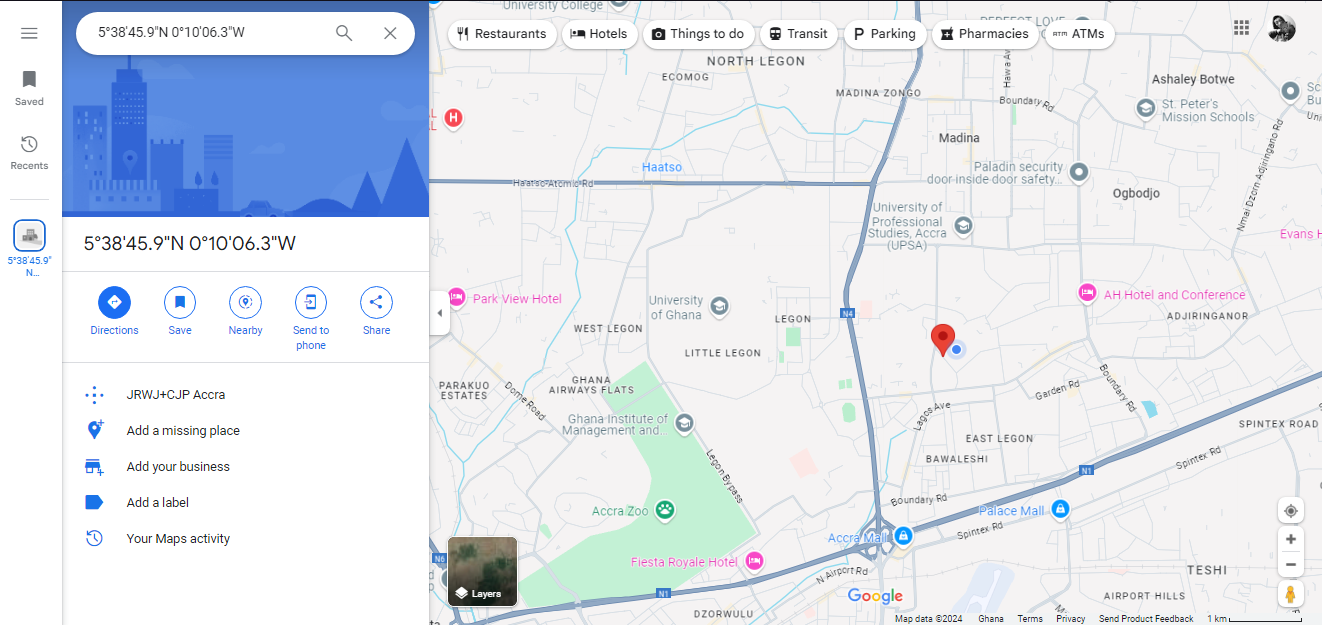
Upload the sketch to the Gaia board and view the GPS data output

Outcome



After successfully parsing GPS data we can check on Google Map to see the real location obtained from the coordinates.

Visit [GoogleMaps](https://www.google.com/maps) and input your coordinates to get the current location of the device



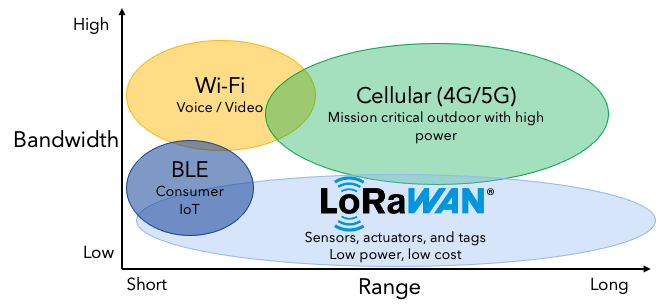
Here we can see the red marker indicating our current location.

Introduction to LoRa Technology

LoRa (Long Range) is a wireless communication technology designed for long-range, low-power, and low-data-rate applications. It operates in the unlicensed Industrial, Scientific, and Medical (ISM) frequency bands (e.g., 868 MHz in Europe, 915 MHz in the US) and is widely used in the **Internet of Things (IoT)** to connect sensors, devices, and systems over long distances with minimal power consumption.

Features of LoRa

* Long Range**:** Communicates over distances of 5-15+ km, ideal for remote areas.
* Low Power:Uses very little power, allowing devices to run for years on small batteries.
* Low Data Rate**:** Transmits small amounts of data at slow speeds, perfect for sending sensor info.
* License-Free:Works on free frequency bands like 868 MHz (Europe) and 915 MHz (US).
* Handles Many Devices:One gateway can connect to thousands of devices at the same time.
* **Two-Way Communication:** Devices can both send and receive data, useful for updates or control commands.
* Interference Resistant:Strong against signal interference, making it reliable in crowded areas.
* Secure:Protects data with AES-128 encryption.
* Location Tracking**:** Can track device locations without needing GPS.
* Adaptive Data Rate (ADR**):** Adjusts settings automatically to improve battery life and communication range.
* Wide Use**:** Can be used for smart cities, agriculture, industry, and more.





LoRa vs. Other Wireless Communication Technologies

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Technology | Range | Power Consumption | Data Rate | Cost | Applications |
| LoRa | 10-15 km | Very low | Low (0.3 - 27 kbps) | Low | Long-range IoT (smart cities, agriculture, metering) |
| Wi-Fi | |  | | --- | | 50-100 m |  |  | | --- | |  | | |  | | --- | |  |  |  | | --- | | High | | High (54 - 600 Mbps) | Moderate | High-bandwidth apps (home automation, cameras) |
| Bluetooth | 10-100 m | |  | | --- | |  |  |  | | --- | | Low to moderate | | Moderate (1-3 Mbps) | Low | Short-range devices (wearables, audio) |
| Zigbee | 10-100 m | |  | | --- | |  |  |  | | --- | | Low | | Low (up to 250 kbps) | Low | Home automation, smart meters |
| NB-IoT | 1-10 km | |  | | --- | |  |  |  | | --- | | Low to moderate | | Low (up to 100 kbps) | Moderate to High | Cellular IoT (smart grids, healthcare) |
| Cellular  (3g,4g,5g) | Several km | High | High (Mbps to Gbps) | High | High-data rate applications (mobile broadband) |

LoRa parameters and their effects on communication

#### 1. ****Spreading Factor (SF)****

Spreading Factor refers to the ratio between the symbol rate and the chip rate in LoRa modulation. It defines how much a data packet is "spread" over the spectrum.

* **Range:** SF6 to SF12 (higher values increase the range but reduce the data rate).
* **Effect:**
  + **Higher SF (e.g., SF12):**
    - **Increases range** by improving signal sensitivity.
    - **Decreases data rate** since the packet takes longer to transmit.
    - **Increases power consumption** because the device spends more time transmitting.
  + **Lower SF (e.g., SF7):**
    - **Decreases range** but allows for faster data transmission.
    - **Reduces power consumption** since the transmission duration is shorter.

#### 2. ****Bandwidth (BW)****

Bandwidth is the width of the frequency range over which data is transmitted.

* **Range:** 125 kHz, 250 kHz, or 500 kHz.
* **Effect:**
  + **Narrower Bandwidth (e.g., 125 kHz):**
    - **Increases range** but **lowers data rate**.
    - **Improves receiver sensitivity**, allowing the signal to travel farther.
  + **Wider Bandwidth (e.g., 500 kHz):**
    - **Increases data rate** but **reduces range**.
    - **Consumes more power** but is suitable for applications requiring faster data transmission over shorter distances.

#### 3. ****Coding Rate (CR)****

The coding rate introduces forward error correction to improve reliability in noisy environments.

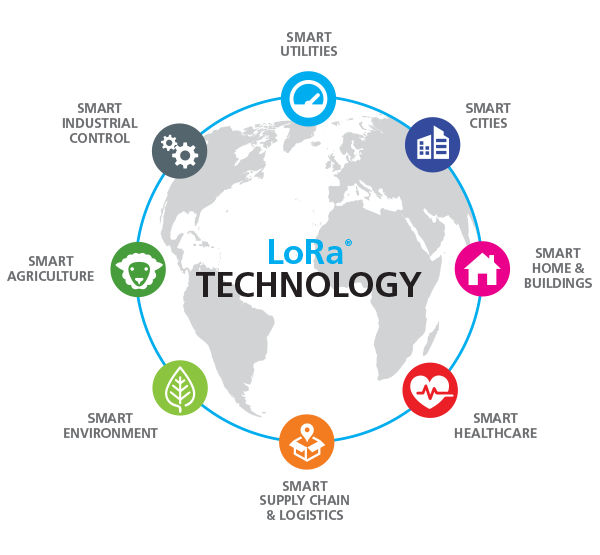
* **Range:** 4/5 to 4/8 (the amount of redundant data sent along with the message).
* **Effect:**
  + **Higher CR (e.g., 4/8):**
    - Increases resilience to interference but reduces data rate.
    - More redundant bits are transmitted, so the overall transmission takes longer.
  + **Lower CR (e.g., 4/5):**
    - Higher data rate but less error correction, suitable for cleaner environments with less interference.

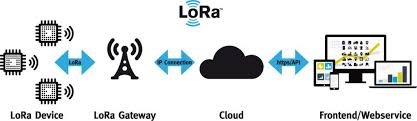
#### 4. ****Transmission Power****

The output power of the LoRa device when transmitting a signal.

* **Effect:**
  + Higher transmission power improves range but increases power consumption.
  + Reducing transmission power can save battery life but limits the range.

### **Applications of LoRa in IoT**

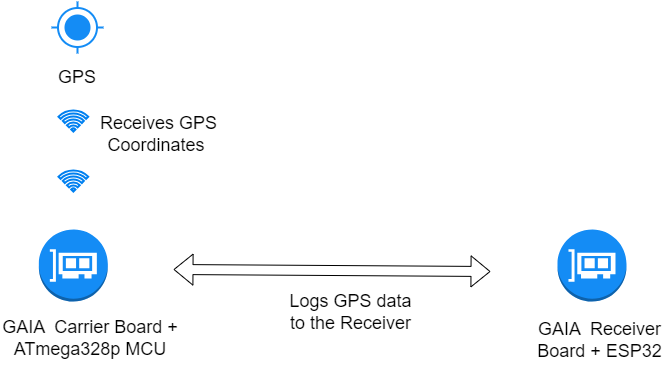
* **Smart Cities:** Parking management, environmental monitoring, street lighting.
* **Agriculture:** Soil moisture monitoring, weather station data collection.
* **Industrial IoT:** Asset tracking, predictive maintenance, sensor networks in factories.
* **Smart Homes:** Connected devices for automation, security, and energy management.
* **Metering:** Gas, water, and electricity metering in remote locations.



Hands-on Activities

Creating a GPS logger with LoRa transmission capabilities

This Hands-on activity will involve the use of all the variants of the GAIA boards.

Below is an illustration of the project

To setup the GPS Logging with LoRa transmission , we first need to install the necessary libraries for LoRa

### Installing the LoRa Library by Sandeep Mistry

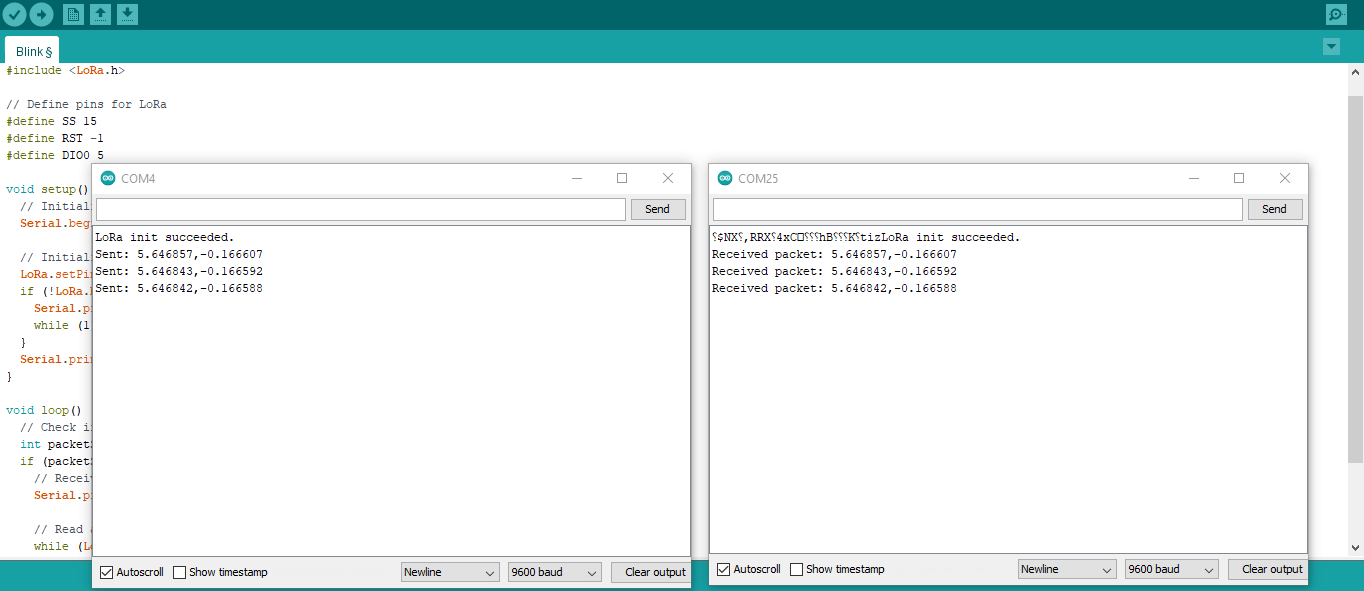
1. **Open Arduino IDE.**
2. Go to **Sketch > Include Library > Manage Libraries.**
3. In the Library Manager, type **"LoRa"** in the search bar.
4. Look for **"LoRa by Sandeep Mistry".**
5. Click on it and then click the **Install** button.

### Installing the RadioHead Library

1. **Open Arduino IDE.**
2. Go to **Sketch > Include Library > Manage Libraries.**
3. In the Library Manager, type **"RadioHead"** in the search bar.
4. Look for **"RadioHead"** (by Mike McCauley).
5. Click on it and then click the **Install** button.

After the necessary installations we open two (2) different sketch with separate serial monitors for both the GAIA Carrier Board and the GAIA Receiver Board

Upload the code (Sketch) for the GAIA Carrier Board and the GAIA Receiver Board using this Link : [GPS\_CarrierCode](https://github.com/augustinegyan/Gaia_Iot_Codes/tree/master/gpsLogger_LoRa)

The Outcome should look like this below:

SECTION 4

Basic and Advanced LoRa Communication

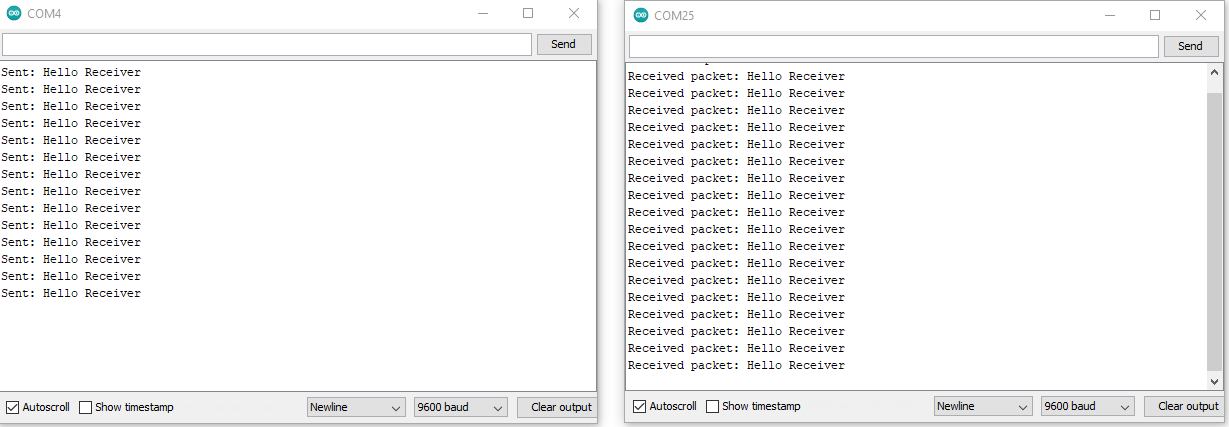
Sending and Receiving messages using LoRa.

**Objective**: Understand the basic functionality of LoRa by setting up a sender and receiver.

The sender sends a simple message (“*Hello Receiver*”) to the Receiver . Code for the project is below

Link : [Sender-and-Receiver-Code](https://github.com/augustinegyan/Gaia_Iot_Codes/tree/master/Send_Receive_LoRa)

Kindly upload the various codes to each variant namely the receiver and sender .

Below is how the Outcome is suppose to look.

Advanced LoRa Techniques

Optimizing LoRa communication with spreading factor and bandwidth adjustments.

Optimizing LoRa communication involves adjusting key parameters like the **spreading factor (SF)** and **bandwidth (BW)**, which directly impact the range, data rate, and power consumption.

Let’s explore how to optimize these parameters to meet different communication needs.

### Optimizing These Parameters:

#### Use Cases:

* **Long-range, low data rate**: Use **higher spreading factor** (e.g., SF12) and **lower bandwidth** (e.g., 125 kHz). This is ideal for sensors transmitting small packets over long distances.
* **Short-range, high data rate**: Use **lower spreading factor** (e.g., SF7) and **higher bandwidth** (e.g., 250-500 kHz). Ideal for short-range, high-speed applications like video streaming.

### Example Code: Optimizing Spreading Factor and Bandwidth

Here’s how you can adjust the spreading factor and bandwidth in your LoRa communication using the **LoRa.h** library.

Sender Code is being optimized for long range



Receiver Code is also being optimized for long range.



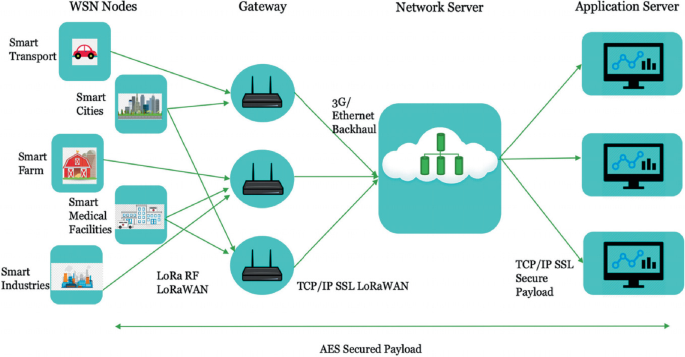
### Key LoRa Configuration Methods:

* **LoRa.setSpreadingFactor(sf)**: Sets the spreading factor. Values range from 6 to 12, where 12 is the longest range and 6 is the shortest.
* **LoRa.setSignalBandwidth(bw)**: Sets the signal bandwidth. Values are in Hz: common options include 125E3 (125 kHz), 250E3 (250 kHz), and 500E3 (500 kHz).
* **LoRa.setCodingRate4(cr)**: Sets the coding rate, which also affects communication robustness. Typically set to 5 for standard robustness.

### Practical Tips:

* **Longer Range**: Increase spreading factor (SF) and decrease bandwidth.
* **Faster Data Rate**: Decrease spreading factor and increase bandwidth.
* **Power Saving**: Use higher spreading factors and lower bandwidth to extend range while consuming less power.

By adjusting the **spreading factor** and **bandwidth**, you can tailor LoRa communication for different use cases, adjusting conflicts in inefficiency between range, data rate, and power consumption. This optimization allows you to build efficient communication systems for various real-world applications.



Implementing acknowledgments, retries and advanced power saving techniques.

Implementing **acknowledgments (ACKs)**, **retries**, and **advanced power-saving techniques** in LoRa communication enhances reliability and efficiency, especially for sensor networks or low-power IoT devices.

### **Acknowledgments (ACKs)**

Acknowledgments are used to confirm that a message was successfully received. The sender waits for an acknowledgment from the receiver after sending a message, and if it doesn't receive one within a specific time, it can resend the message.

### **Retries**

Retries come into play when the sender does not receive an acknowledgment. If no ACK is received within the timeout, the message is resent a specified number of times.

#### Adding Retries to Sender:

You can modify the previous sender code to include retries. After each failed attempt (i.e., when ACK is not received), the sender will retry a certain number of times before giving up.

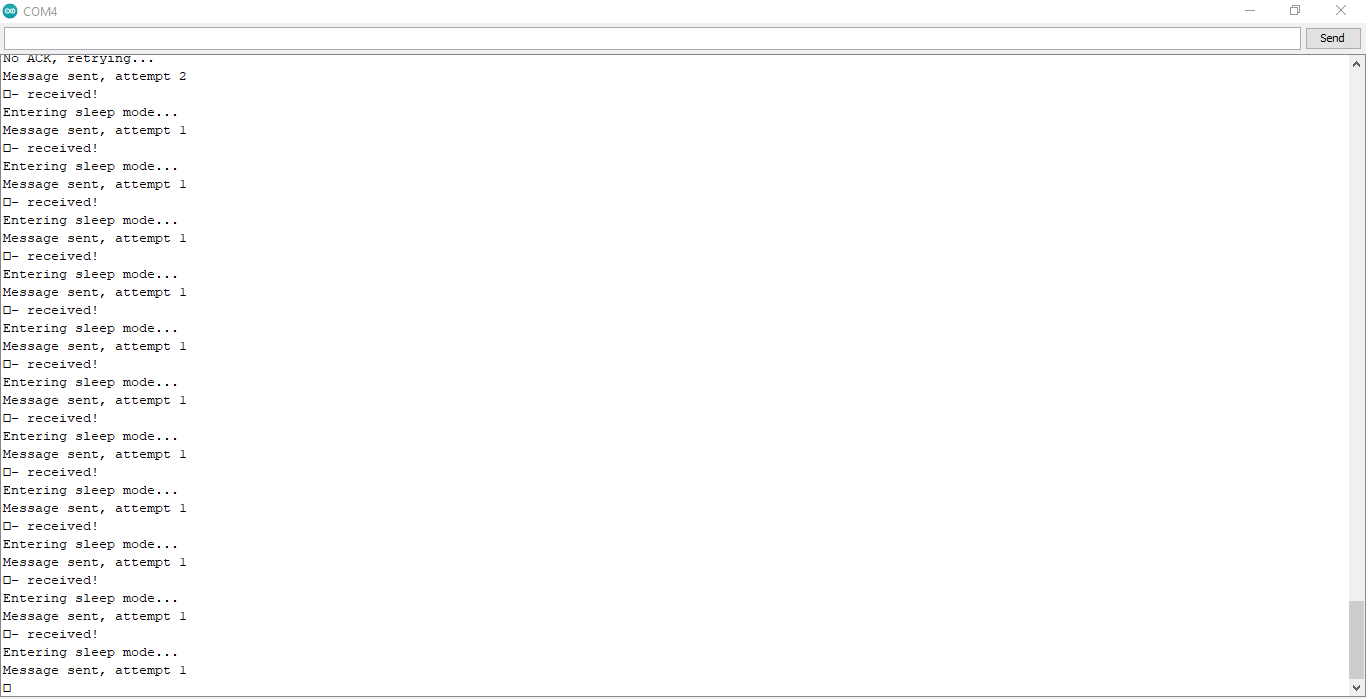
### **Advanced Power-Saving Techniques**

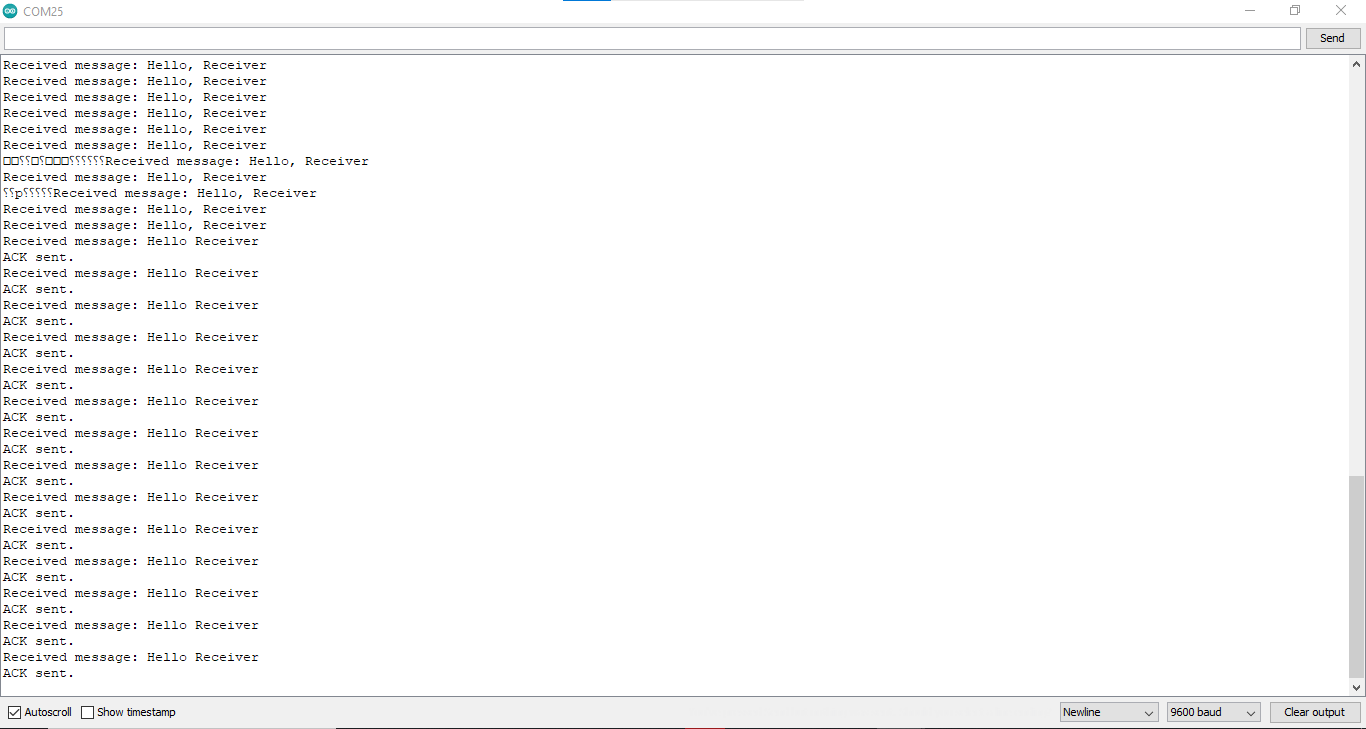
LoRa devices can be configured to save power by entering **sleep mode** when not actively transmitting or receiving. This is critical for battery-powered sensors.

* **LowPower.h** library can help put the device to sleep.
* You can also adjust the LoRa settings (e.g., decreasing the bandwidth and using a higher spreading factor) to minimize power consumption during communication.

Kindly get the code from this link : [LoRa\_Implementation\_Codes](https://github.com/augustinegyan/Gaia_Iot_Codes/tree/master/LoRa_implementation)

Expected Outcome

Sender

Receiver

#### ****Sender:****

1. **Sending a message:** The sender sends "Hello Receiver" and waits for an acknowledgment (ACK).
2. **Retries:** If no ACK is received within 2 seconds, it retries up to 3 times before giving up.
3. **Low-Power Mode:** After completing the transmission (successful or failed), the ATmega328 goes into low-power sleep mode for 8 seconds using the LowPower.powerDown() function.
4. **Optimized LoRa Settings:** The signal bandwidth is reduced to 125 kHz for power-saving, and the spreading factor is set to 12 to maximize range.

#### ****Receiver:****

1. **Receiving the message:** The ESP12F listens for incoming messages and prints the received message to the serial monitor.
2. **Sending an ACK:** Once the message is received, the receiver sends an "ACK" back to the sender to confirm successful receipt.
3. **Optimized LoRa Settings:** The signal bandwidth and spreading factor are also set to optimized values for power saving and better range.

### **Power-Saving Techniques:**

* **Low bandwidth** and **high spreading factor** reduce power consumption while improving range.
* The ATmega328 enters **sleep mode** using the LowPower.h library, allowing the microcontroller to save energy between transmissions.

This setup is suitable for low-power IoT applications, where **reliable communication** and **energy efficiency** are critical. By adjusting the **spreading factor**, **bandwidth**, and using **sleep modes**, the system maximizes battery life while ensuring robust communication.

Hands-on Activities

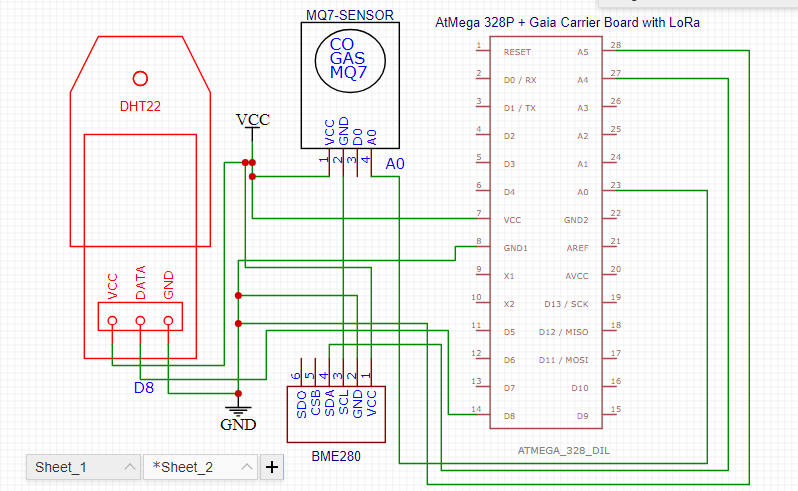
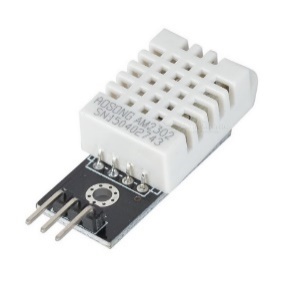
Building an optimized LoRa-based multi-sensor weather station .

In this hands-on project, we will build a weather station using multiple sensors to measure environmental data such as temperature, humidity, pressure, and possibly air quality. The data will be transmitted over long distances using LoRa (Long Range) communication. This setup will include **sensor data collection**, **LoRa communication**, and **power-saving techniques** to optimize the performance of the weather station.

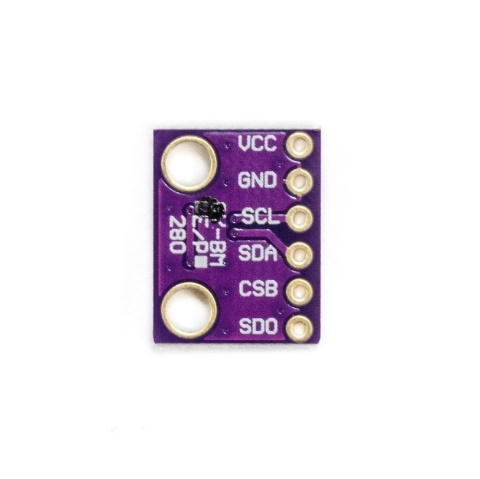
Components used

­DHT22 Sensor, BME280, MQ-7 sensor and both variants of the GAIA LoRa Receiver and Sender Boards.

Below is the Schematics and Components



DHT22 Sensor



BME280 Sensor



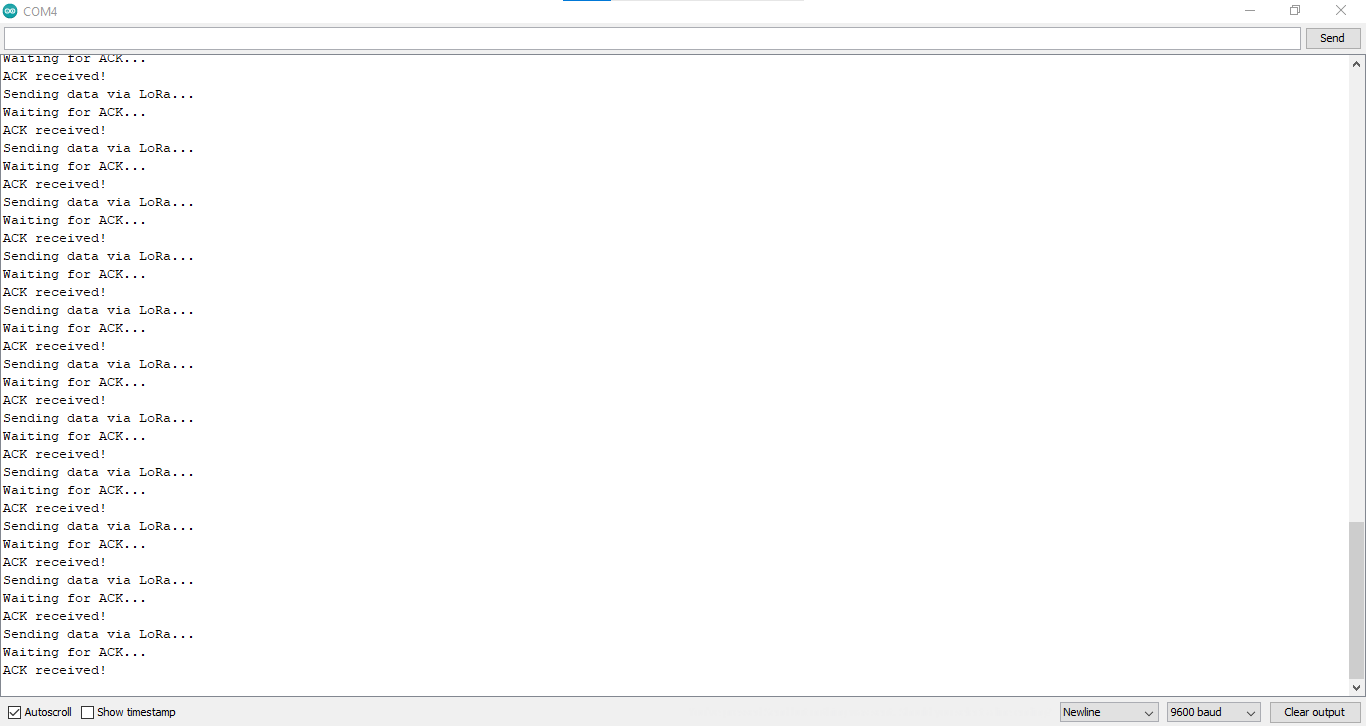
MQ-7 Sensor

### **Outcome:**

By following these steps, you have built a **LoRa-based weather station** that collects sensor data, transmits it over long distances using optimized LoRa communication, and implements **retries**, **ACKs**, and **power-saving techniques** to ensure reliability and efficiency.

Below is the Link to the Code for Upload

[LoRa\_Weather\_Station\_Codes](https://github.com/augustinegyan/Gaia_Iot_Codes/tree/master/LoRa_Multi_Sensor-WeatherStation)

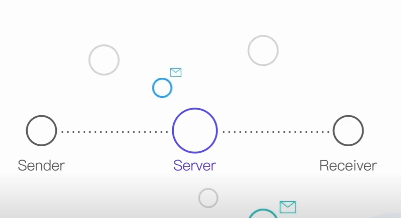
Sender Outcome

Receiver Outcome

SECTION 5

Introduction to MQTT Protocol and ESP Integration

MQTT (originally an initialism of MQ Telemetry Transport[a]) is a lightweight, publish-subscribe, machine to machine network protocol for message queue/message queuing service. It is designed for connections with remote locations that have devices with resource constraints or limited network bandwidth, such as in the Internet of Things (IoT). It must run over a transport protocol that provides ordered, lossless, bi-directional connections—typically, TCP/IP.[1] It is an open OASIS standard and an ISO recommendation (ISO/IEC 20922).

Features of MQTT

**MQTT Sender (Client)**:

The **sender** in MQTT is typically referred to as a **client**. Clients can be **publishers** (sending data to the broker) or **subscribers** (receiving data from the broker). A client in MQTT can act as both publisher and subscriber, but it does so at different times.

Example: An ESP32 device can **publish** temperature sensor data and **subscribe** to commands for adjusting its operation.

**MQTT Receiver (Client)**:

The **receiver** is also a **client** that subscribes to topics and listens for messages. Whenever data is published on a topic that the client is subscribed to, it will receive that data.

Example: A smart home system that subscribes to home/livingroom/light and reacts when a command to turn the lights on or off is published.

#### ****MQTT Topic****

Topics are the categories that MQTT messages are published to and subscribed from. They serve as the routing information for the broker to send messages to the right clients. Topics have a hierarchical structure separated by /.

* **Example Topics**:
  + home/livingroom/temperature – could carry temperature data from a sensor.
  + home/garage/door/status – could carry the status of a garage door (open/closed).

### **Publisher and Subscriber in MQTT**

The MQTT protocol operates on a **publish-subscribe model** where devices (clients) can take on the role of either **Publisher** or **Subscriber** (or both) in an MQTT communication system.

#### 1. ****Publisher****

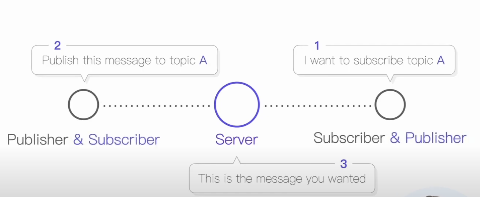
A **Publisher** is a client that **sends data** to the MQTT broker. It does so by publishing messages to specific **topics**.

* **How It Works**: The Publisher connects to the MQTT broker and then sends messages to a specific topic (like a "channel"). The MQTT broker then distributes this message to any subscribers that have subscribed to that topic.
* **Example**:
  + An ESP32 equipped with a temperature sensor reads temperature data every minute and publishes the reading to the topic *home/livingroom/temperature.*

#### 2. ****Subscriber****

A **Subscriber** is a client that **receives data** from the MQTT broker. It subscribes to a specific topic or set of topics to receive messages published to those topics.

* **How It Works**: The Subscriber connects to the MQTT broker and subscribes to specific topics. Once a message is published to a topic that it subscribes to, the broker forwards the message to the Subscriber.



#### ****MQTT Session****

* An **MQTT session** is the persistent relationship between a client and the MQTT broker. A session allows a client to maintain continuity, even through intermittent connections, which is especially useful for IoT devices that sleep or reconnect frequently.

#### ****MQTT Client and Server (Broker)****

* **MQTT Client**:
  + A client can be any device or application that connects to the broker. It could be an ESP32, Raspberry Pi, or even a desktop application. Clients can **publish** messages to a topic or **subscribe** to topics to receive messages.
* **MQTT Server (Broker)**:
  + The **broker** is the central component of the MQTT architecture. It acts as an intermediary between the clients (publishers and subscribers). Its job is to manage connections, distribute messages, and ensure that messages reach all subscribers to a particular topic.
* **Popular MQTT Brokers**:
  + **Mosquitto**: Open-source MQTT broker widely used for local development.
  + **HiveMQ**: Offers both cloud-based and on-premise solutions.
  + **Eclipse Paho**: MQTT client and broker libraries used in IoT projects.

#### ****MQTT Quality of Service (QoS)****

**QoS** (Quality of Service) determines the reliability of message delivery between clients and brokers in MQTT. It defines how a message is handled in terms of acknowledgment and retrying in case of communication failure.

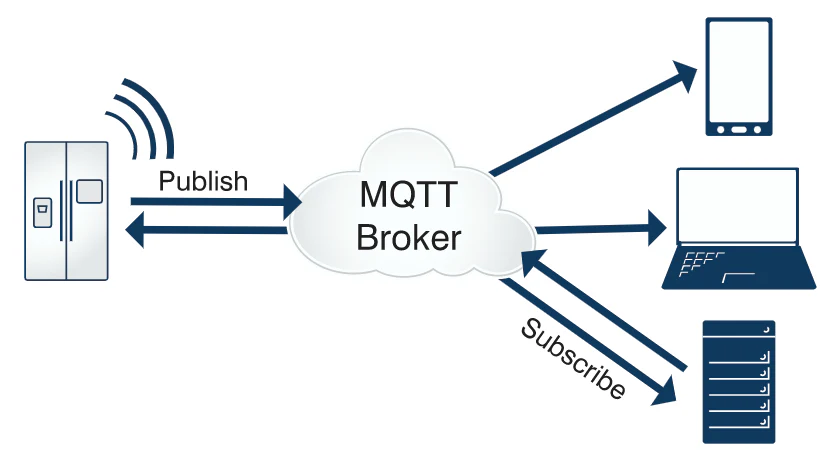
* **QoS 0 – At Most Once**:
  + **Description**: The message is delivered once, with no acknowledgment required. There is no guarantee that the message will arrive.
  + **Use Case**: For non-critical data where occasional message loss is acceptable (e.g., temperature readings every minute).
* **QoS 1 – At Least Once**:
  + **Description**: The message is delivered at least once, meaning it could be delivered more than once if acknowledgments are not received.
  + **Use Case**: For cases where it's acceptable to receive duplicate messages, but message loss is not acceptable (e.g., alarms or commands).
* **QoS 2 – Exactly Once**:
  + **Description**: The message is guaranteed to be delivered exactly once, with a full handshake to ensure no duplicates are sent.
  + **Use Case**: For critical systems where it is essential that a message arrives only once (e.g., payment transactions or critical alerts).

**Publish-Subscribe Model**:

* **Publisher**: Sends messages to a topic.
* **Subscriber**: Receives messages from a topic.
* **Broker**: Acts as a mediator between publishers and subscribers, managing the distribution of messages.

**Significance of MQTT in IoT Communication**:

* Efficient for low-bandwidth environments.
* Supports a large number of devices (scalable).
* Allows for remote monitoring and control of devices.
* Ensures message delivery .
* Helps transmit real-time data over the internet.



### Introduction to ESP and Its Wi-Fi Capabilities

The ESP series, particularly the **ESP8266** and **ESP32**, are popular microcontroller modules widely used in Internet of Things (IoT) projects due to their **Wi-Fi** capabilities and ease of use. These modules are small, cost-effective, and powerful, making them ideal for connecting devices to the internet.

#### 1. ****What is ESP?****

ESP stands for **Espressif Systems**, the company that produces the ESP8266 and ESP32 modules. These chips have built-in **Wi-Fi**, allowing them to connect to local networks or the internet without needing additional hardware.

* **ESP8266**: This is one of the earlier models. It offers basic Wi-Fi connectivity and is commonly used in IoT projects. It has a single-core processor and limited GPIO (General Purpose Input/Output) pins.
* **ESP32**: An upgraded version of the ESP8266, the ESP32 is more powerful, with dual-core processors, Bluetooth capabilities, and more GPIO pins. It is also more efficient in power usage, making it better suited for projects requiring low power consumption.



#### 2. ****Wi-Fi Capabilities****

The key feature of ESP modules is their ability to connect to **Wi-Fi networks**. This capability allows ESP-based devices to interact with the cloud, other devices, or web services. Here’s a breakdown of its main Wi-Fi functions:

* **Station Mode (STA)**: In this mode, the ESP acts as a client and connects to an existing Wi-Fi network (like your home router). It can then send or receive data from the internet.
* **Access Point Mode (AP)**: In this mode, the ESP becomes a Wi-Fi hotspot, allowing other devices to connect directly to it. This is useful in cases where you want to control or monitor the ESP device locally without an internet connection.
* **Station + Access Point Mode (STA+AP)**: This hybrid mode lets the ESP connect to a Wi-Fi network while simultaneously acting as a hotspot for other devices. This can be useful for creating local networks while still maintaining internet access.

#### 3. ****Wi-Fi Communication****

ESP modules use standard Wi-Fi protocols to communicate. They support:

* **2.4 GHz Wi-Fi**: This is the standard Wi-Fi frequency, offering reasonable speed and range.
* **TCP/IP Protocol**: They can connect to servers or act as a server themselves, using the TCP/IP stack to communicate with other devices over the internet.

#### 4. ****Wi-Fi Security****

ESP modules support modern Wi-Fi security standards:

* **WPA/WPA2 Encryption**: This ensures secure connections between the ESP and Wi-Fi networks, protecting data sent over the network.

### Implementing Deep Sleep Modes for Power Conservation in ESP32

One of the most powerful features of the **ESP32** is its ability to enter **deep sleep mode**, which allows the device to drastically reduce power consumption. This is especially useful for **battery-powered** Internet of Things (IoT) devices that need to operate for long periods without recharging. In deep sleep mode, most of the ESP32's functions are turned off, consuming just a few microamps of power, which can extend battery life significantly.

#### ****How to Implement Deep Sleep Mode in ESP32****

To implement deep sleep mode, you need to decide on how the ESP32 should wake up. It can wake up after a certain time interval (using a timer) or from an external signal (such as a sensor or button press).

Deep Sleep can be implemented either the Timer based or GPIO-Based.

Timer=Based meaning ESP32 enters deep sleep and wakes up after a predefined time interval.

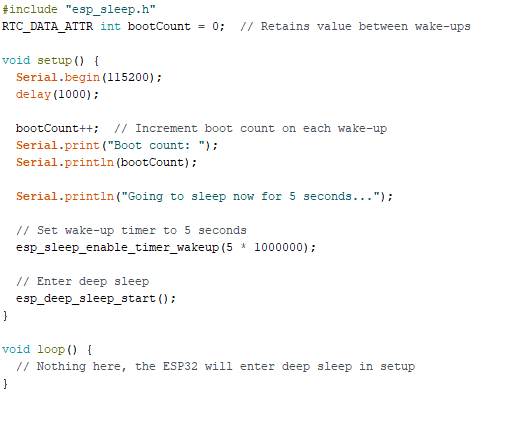
GPIO-Based meaning ESP32 wakes up from deep sleep when a button is pressed or a sensor triggers a signal on a specific GPIO pin.

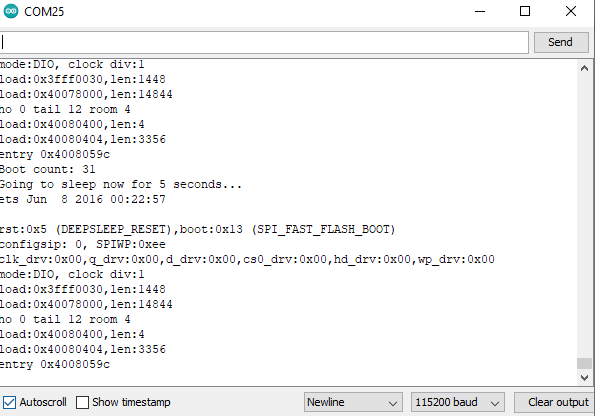
We will be implementing simple codes to show Deep Sleep Mode in ESP32 using the Timer-Based and GPIO-Based examples.

Timer-Based Implementation of Deep Sleep with ESP32.

As said earlier the Timer-Based implementation ensures that the ESP32 goes to deep sleep at certain time intervals.

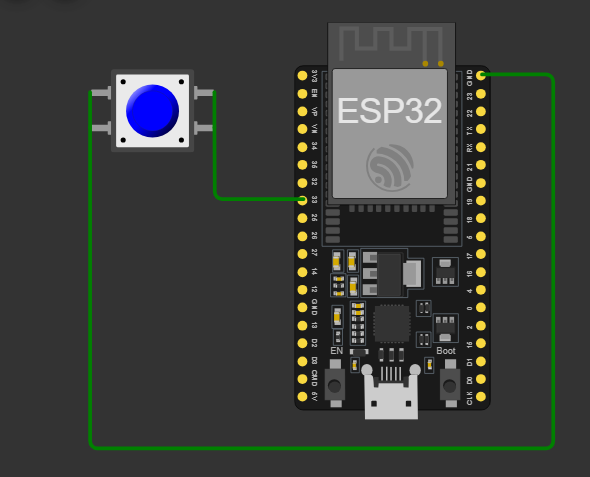
Timer-Based Deep Sleep

Link to Code : Timer\_Based\_Deep\_Sleep

Below is the outcome

The ESP32 will sleep for 5 seconds and then automatically restart from the setup function. You can see the output in the serial monitor after each wake-up.

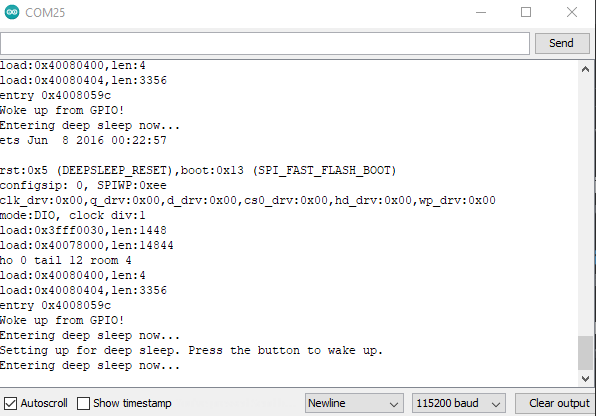
GPIO-Based Timer-Based Implementation of Deep Sleep with ESP32.



Link to Code : GPIO\_Based\_Deep\_Sleep

### How It Works:

* When you press the 4-pin switch, it will connect GPIO 33 to GND, sending a LOW signal.
* This triggers the wake-up from deep sleep, and the ESP32 will print a message to the Serial Monitor.

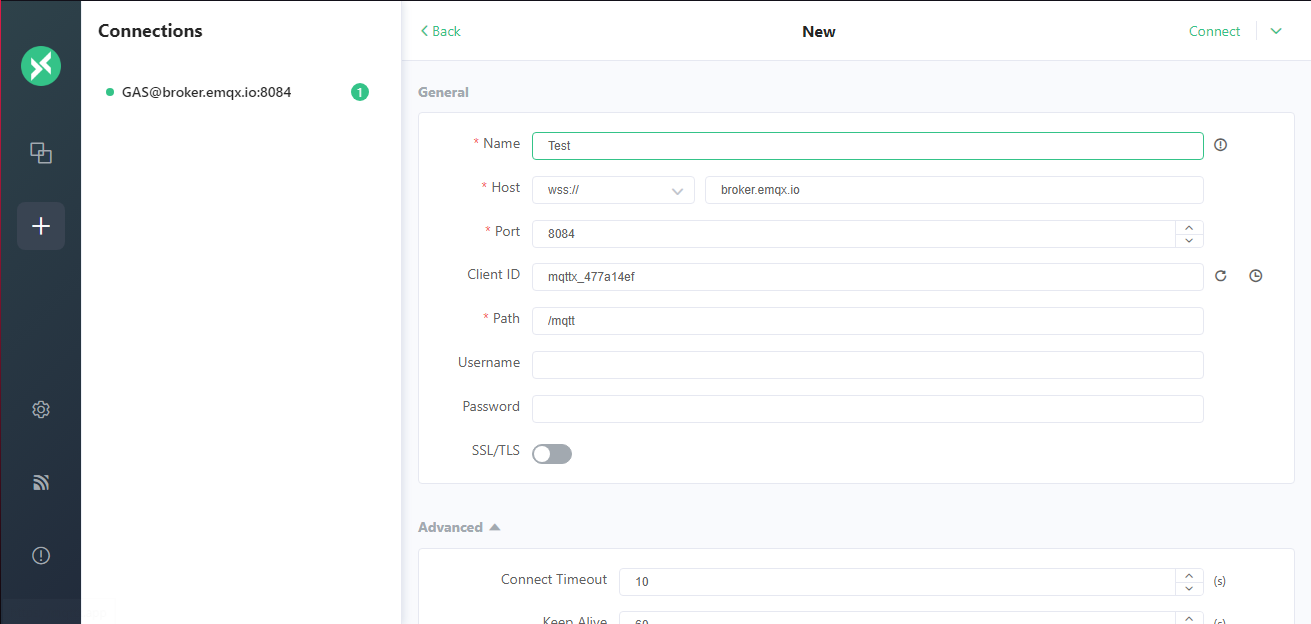
Outcome

Hands-on Activities

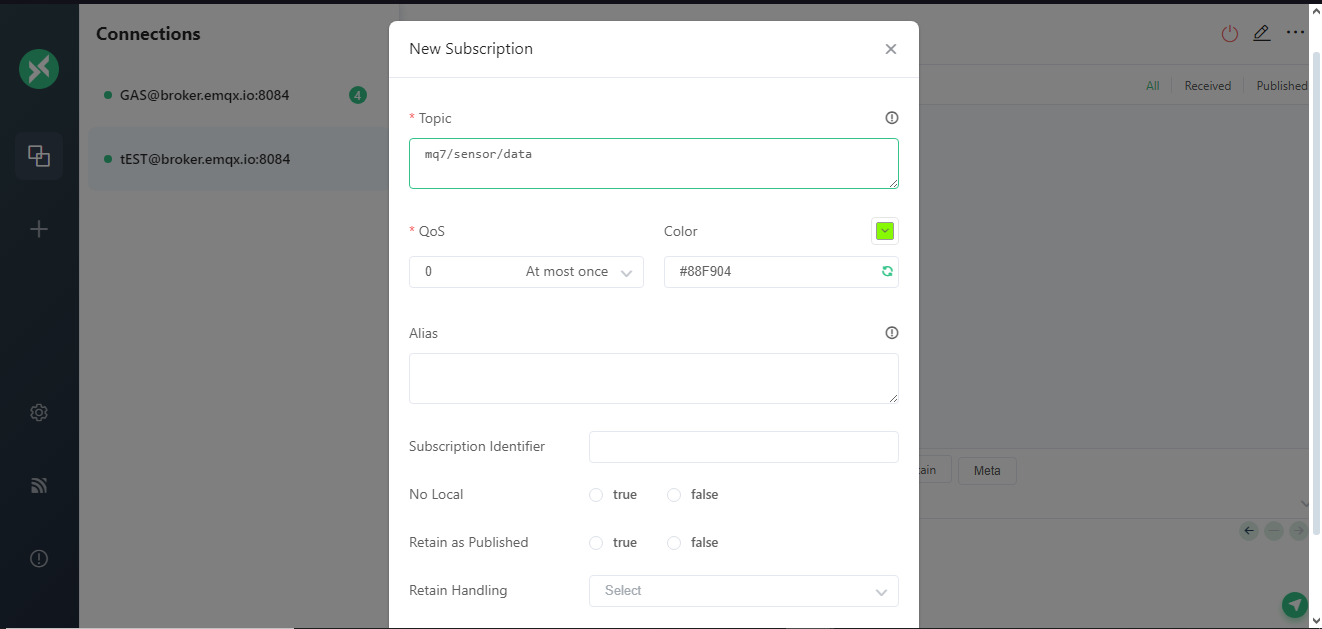
Creating a LoRa to MQTT bridge using ESP variants and publishing on MQTT broker

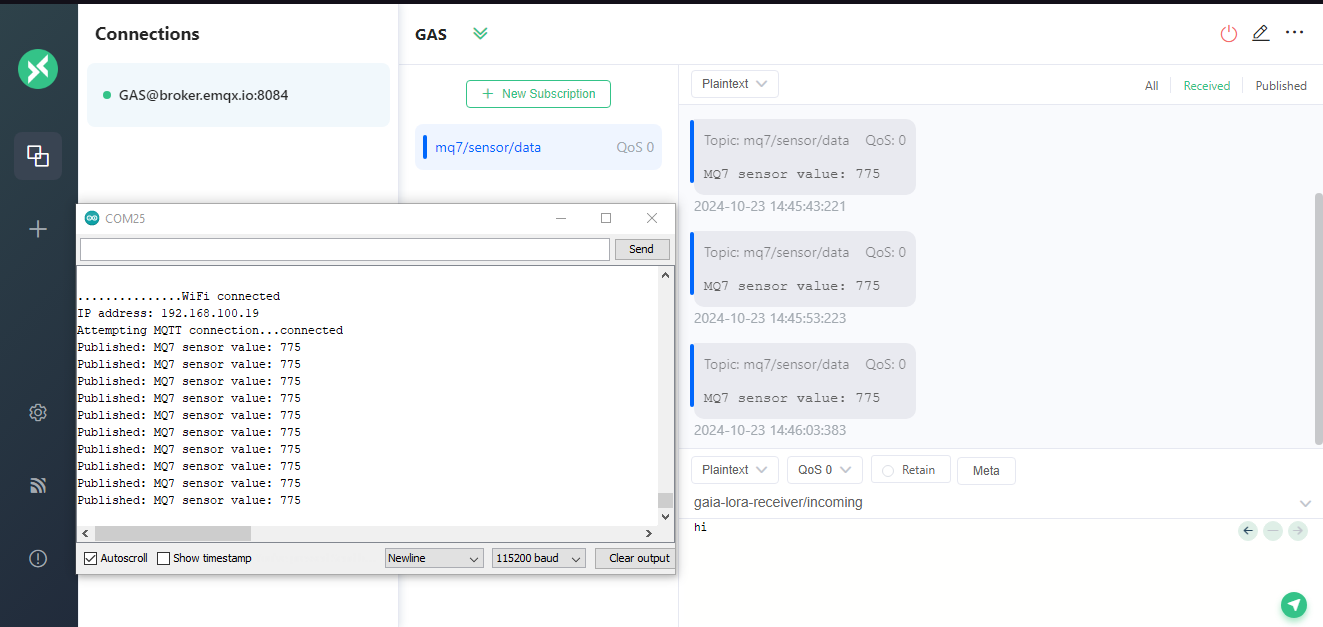
The Objective of this project is to send our gas sensor data from the ESP To the MQTT broker using the MQTTX Broker.

Upload the following sketch in the Arduino IDE – [Link to Code](https://github.com/augustinegyan/Gaia_Iot_Codes/tree/master/Mqttx_sensorData_internet)

Open MQTTX Online Broker

Input your desired name in the “Name” section and click on connect

After that we subscribe our connection by using the topic “***mq7/sensor/data***”

Below is the outcome to see

This process will continuously send the **MQ7 sensor data** to the **MQTT broker**, which you can monitor using **MQTTX**.

SECTION 6

Advanced MQTT Techniques and Secure Data Communication

Advanced MQTT Client and Server Optimization

MQTT Client and Server Optimization

MQTT (Message Queuing Telemetry Transport) is a lightweight messaging protocol designed for low-bandwidth, high-latency networks. It is widely used in IoT (Internet of Things) applications for its efficiency and simplicity. As IoT systems become more complex, optimizing MQTT communication and ensuring data security becomes crucial. This article explores advanced techniques for optimizing MQTT clients and servers, as well as implementing secure data communication practices.

Advanced MQTT Client and Server Optimizations

1. Connection Management

Efficient connection management is essential for reducing overhead and improving response times. Here are a few techniques:

Persistent Sessions: Use persistent sessions to retain subscriptions and message states across sessions. This minimizes the need for resubscription and allows clients to receive messages sent while they were disconnected.

Last Will and Testament (LWT): Configure a last will message for each client. If a client unexpectedly disconnects, the broker sends this message to a specified topic, allowing other clients to be informed about the disconnection.

Keep-Alive Intervals: Adjust the keep-alive interval to suit the application’s needs. A shorter interval can help detect disconnections faster, while a longer one may reduce network traffic.

2. Quality of Service (QoS) Levels

MQTT supports three QoS levels:

QoS 0 (At most once): The message is sent without confirmation. Use this for non-critical data.

QoS 1 (At least once): Ensures that the message is delivered at least once. This is useful for important notifications.

QoS 2 (Exactly once): Guarantees message delivery exactly once, suitable for critical data transactions. However, it introduces more overhead.

Choosing the appropriate QoS level can optimize network performance based on the application's reliability requirements.

3. Message Batching

Batching multiple messages into a single transmission reduces the number of packets sent over the network, which can significantly lower latency and improve throughput. This technique is particularly useful in scenarios with frequent data updates.

4.Efficient Topic Structures

Design an efficient topic structure to minimize the number of topics. Group related data under a common hierarchy. This allows for more efficient subscriptions and reduces the load on the broker.

Implementing Data Encryption and Security consideration for safe IoT communication.

Data encryption is essential for protecting sensitive information in IoT communications. It ensures that data sent between devices (like an ESP12F) and servers (like an MQTT broker) cannot be easily intercepted or tampered with.

Implementing Data Encryption

1. Transport Layer Security (TLS)

TLS is a widely used protocol for securing communications over networks. It provides encryption, data integrity, and authentication.

TLS encrypts data transmitted between clients and servers, ensuring that even if data is intercepted, it cannot be easily read.

2. End-to-End Encryption (E2EE)

E2EE ensures that data is encrypted on the sender's device and only decrypted on the recipient's device.

Data is encrypted before transmission, and only authorized devices can decrypt it using private keys.

3. Message-Level Encryption

Encrypting individual messages can enhance security, especially in scenarios where data is stored or processed by intermediate devices.

Utilize encryption libraries available for your IoT platform to encrypt payloads before sending them. For example, encrypt sensor data before publishing it to an MQTT topic.

4. Secure Protocols

Utilize secure protocols designed for IoT environments:

MQTT over TLS: Use MQTT with TLS to secure the communication channel.

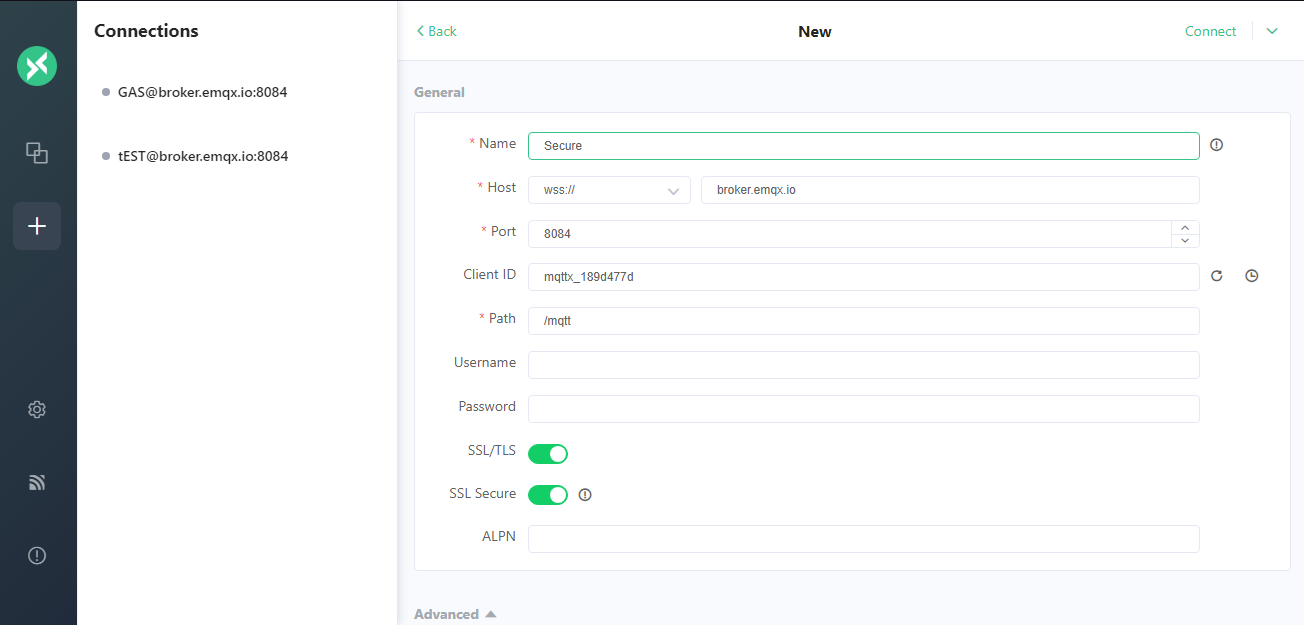
CoAP with DTLS: For lightweight applications, use Constrained Application Protocol (CoAP) with Datagram Transport Layer Security (DTLS) for securing communications.

Hands-On Activities: Enhancing the LoRa to MQTT Bridge

Our Goal is to set up a bridge that receives data from LoRa devices and forwards it securely to an MQTT broker while implementing advanced data handling techniques.

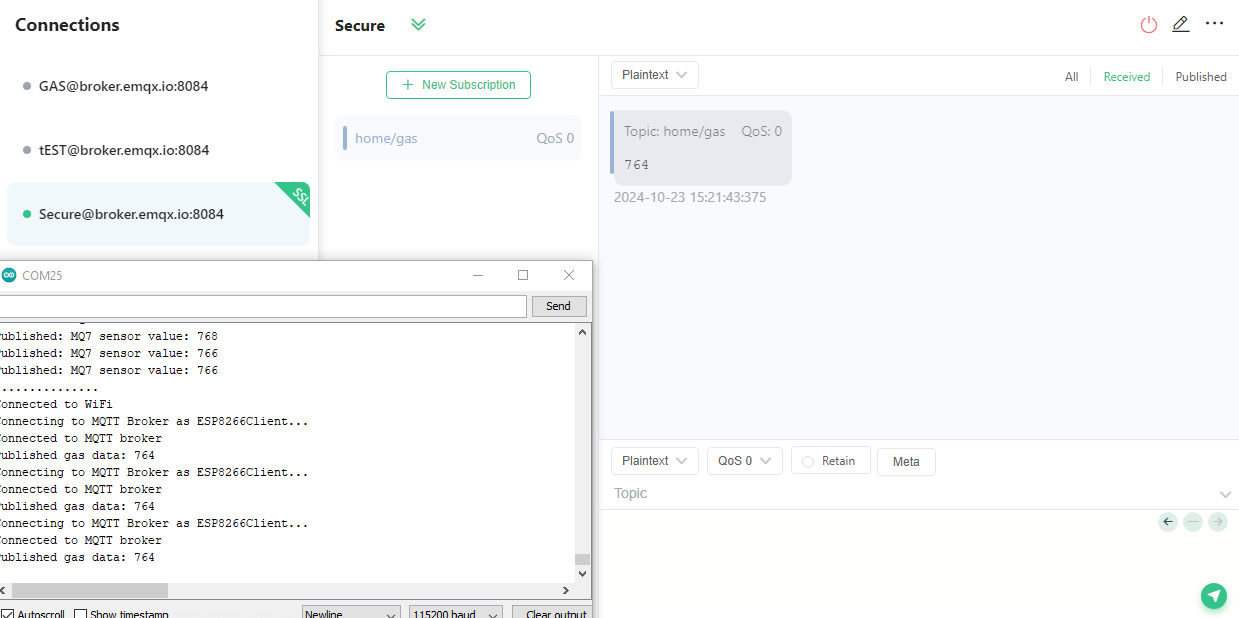
Link to the Code to implement

1.Open MQTTX broker and toggle to on TLS/SSL for the secure communication



**TSL** typically refers to **Transport Layer Security**, a cryptographic protocol designed to provide secure communication over a computer network. TSL ensures privacy, data integrity, and authentication between two communicating applications. It is widely used in various applications, including web browsing (HTTPS), email, instant messaging, and other forms of communication.

2.Connect the topic with the required topic name to read data on the mqttx broker as done in earlier projects.

Below is the Outcome to expect

This implementation connects an ESP12F to WiFi, reads Gas data from a sensor, and securely sends this data to an MQTT broker using TLS encryption. The use of secure communication ensures that the data transmitted is protected from interception, making it safer for IoT applications and good data handling.

Section 7

Data Integration and Power Management for End-to-End IoT systems

The rapid growth of the Internet of Things (IoT) has enabled a wide range of applications across industries, from agriculture to smart cities. Integrating various IoT components and managing power consumption are critical for developing sustainable and efficient systems. This article outlines a framework for sending multi-sensor and GPS data over LoRa, receiving data with an ESP microcontroller, and publishing it to an MQTT broker, while also implementing efficient power management schemes for IoT nodes.

Integration of IoT Components

Sending Multi-Sensor and GPS Data over LoRa

LoRa (Long Range) technology is an effective way to transmit data over long distances with low power consumption. The ability to send data from multiple sensors and GPS modules is essential for real-time monitoring and analytics.

For this implementation we have already done an illustration from the previous chapters were we read data from each sensor and the GPS module and the send them to the LoRa boards for transmission.

Receiving Data with ESP and Publishing to an MQTT Broker

Once the data is transmitted via LoRa, it needs to be received by another ESP device and published to an MQTT broker for further processing or monitoring. Here we used the MQTTX broker to receive and monitor the data collected from our endpoints.

Implementing an Efficient Power Management Scheme for IoT Nodes

Efficient power management is crucial for the longevity of IoT devices, especially those deployed in remote or hard-to-reach locations. Implementing a power management scheme can significantly reduce energy consumption.

Power Management Techniques

Sleep Modes: Utilize deep sleep modes available on ESP8266/ESP32 to minimize power consumption when not in active use.

Low-Power Sensors: Use energy-efficient sensors that require minimal power during operation.

Adaptive Data Transmission: Implement data aggregation or batching to reduce the frequency of data transmissions. Send data only when significant changes are detected.

Integrating IoT components like multi-sensors and GPS with LoRa technology and managing power consumption effectively are essential for creating robust IoT systems.

Hands-On Activities

Buildign an end-to-end IoT system focusing on sensor integration, LoRa communication , and power management

Overview of the System

Sensors:

BME280: For temperature, humidity, and pressure readings.

MQ7: For carbon monoxide (CO) gas detection.

LoRa Communication:

GAIA Sender Node(ATmega Node): Collects data from the sensors and sends it over LoRa.

GAIA Receiver Node(ESP12F Node): Receives data via LoRa and pushes it to an MQTT broker.

Power Management: Use sleep modes to conserve energy, especially on the ATmega node.

Sender Schematics

