Chapter 2 Building a Sensor Node Using ESP32

As we move forward into this chapter, we're transitioning from foundational principles of ESP32 development setup to the practical construction of a sensor node device. In this chapter, you will explore how to integrate real-world sensors such as temperature, humidity, soil moisture, or light illuminance with the ESP32 development board, collect environmental data, and make it sensible. Based on real-time sensor data, you will construct a data-driven decision-making process to trigger an event or automate the system. You'll learn not only how to read and process analog and digital signals using Arduino C++ code, but also how to design reliable, low-power sensor routines, manage timing and sampling rates.

By the end of this chapter, you will have successfully built a functioning sensor node capable of collecting meaningful data from its surroundings and automating a system based on the sensor's real-time value.

In this chapter, you will be able to

- Interface environmental sensors such as temperature, humidity, soil moisture, and light, and connect them properly to the ESP32 using digital and analog GPIO pins.
- Write Arduino code that reads sensor data, handles potential errors, manages timing intervals, and event triggers.
- Incorporate simple algorithms to smooth raw sensor data for greater precision.
- Use low-power strategies such as deep sleep, controlled wake cycles, and CPU frequency scaling to extend battery life in deployed sensor nodes.
- Build a versatile sensor node that aligns with future steps in this book, like mobile app integration, web dashboard, cloud integration, and more.

2.1 A Sensor Node

Sensor devices are crucial in the rapidly changing IoT environment, bringing a new era of connected and intelligent systems. A sensor node is an autonomous system that measures and detects environmental attributes, such as temperature, humidity, and pressure, and converts them into digital signals for monitoring and automation purposes. It is a core component in Wireless Sensor Network (WSN) or IoT that relays data across the network to a main site where the data can be observed and analyzed [1], [2]. The versatility of wireless sensors in addressing various application domains has seen a significant increase in popularity. Numerous application areas, including healthcare, industrial monitoring, environmental sensing, agriculture, biomedical, and more, have been successfully implemented.

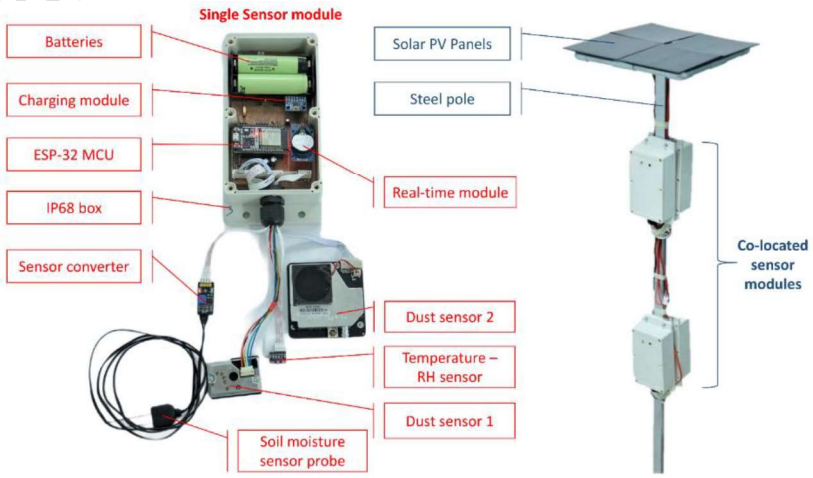


Figure 1: *A prototype of a wireless sensor node using ESP32 powered by mini solar panels. Adopted from Santanu Metia [3].*

There are a lot of challenges posed by the deployment of wireless sensors. It communicates wirelessly from a distance, as shown in **Figure 1**. For instance, smart farming, as an example, uses networked sensor nodes for real-time monitoring of arable land. As seen in the figure below, a sensor node comprises a wireless communication module, a processor, a power source, and a sensing unit [2].

2.2 Sensor Interfacing

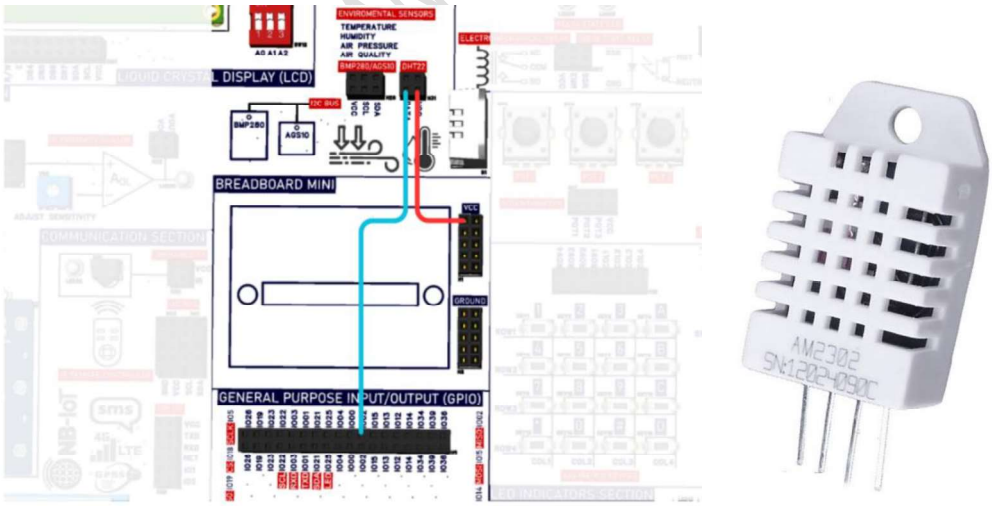
2.2.1 Digital Temperature and Humidity Sensor

The trainer board has AM2302 (DHT22), a digital temperature and humidity sensor module. It is an ultra-low-power, low-cost, high-precision, and fully automated calibration sensor based on capacitive technology that uses a standard digital single-bus output [4]. Based on its datasheet, the sensor specification is shown below.

TABLE 1
AM2302 (DHT22) Digital Temperature and Humidity Sensor Specification [4]

Parameter	Min	Typical	Max
Voltage	3.3V	5.0V	5.5V
Relative Humidity Resolution		0.1% RH	
Relative Humidity Range	0		99.9% RH
Relative Humidity Accuracy		± 2% RH	
Temperature Resolution		0.1˚C	
Temperature Range	-40˚C		80˚C
Temperature Accuracy		± 0.5˚C	± 1˚C

The trainer board and AM2302 (DHT22) connection typical application circuit is shown in **Figure 2 (a)**. The single bus data pin is wired to the ESP32 I/O port (i.e., IO02) is connected. The VCC pin is wired to the breadboard 5V power rail.



(a) (b)

Figure 2: *Trainer board digital temperature and humidity sensor (a) wiring interface, (b) AM2302 (DHT22) sensor component used in the platform.*

To read the sensor values, it is required to have a minimum interval of 2 seconds. A reading interval is less than 2 seconds might cause the sensor reading to communication unsuccessful. It is recommended to repeatedly read sensors, and for each reading, the sensor interval should be greater than 2 seconds, to obtain accurate data. The Arduino code example to read temperature and relative humidity values is shown in **Listing 1**.

Listing 1 *Reading Temperature and Relative Humidity from AM2302 (DHT22) sensor*

```
#include <DHT.h>               // DHT sensor library
#include <Wire.h>              // I2C communication
#include <Adafruit_GFX.h>      // Core graphics library
#include <Adafruit_SSD1306.h>  // OLED display driver

// Hardware configuration
#define OLED_SDA 21    // OLED SDA pin (ESP32 default)
#define OLED_SCL 22    // OLED SCL pin (ESP32 default)
#define OLED_RST -1    // Reset pin (-1 = unused)
#define DHTPIN 2       // DHT sensor data pin
#define DHTTYPE DHT22  // Sensor type (DHT22/AM2302)

// OLED display parameters
#define SCREEN_WIDTH 128  // OLED width in pixels
#define SCREEN_HEIGHT 64  // OLED height in pixels

// Initialize hardware objects
Adafruit_SSD1306 oled(SCREEN_WIDTH, SCREEN_HEIGHT, &Wire, OLED_RST);
DHT dht(DHTPIN, DHTTYPE);
```

This foundational segment establishes the hardware-software interface. Library inclusions leverage modular design principles. You should install the following library before running this example.

- DHT sensor library by Adafruit
- Adafruit SSD1306 by Adafruit
- Adafruit GFX library by Adafruit

With the Adafruit DHT library abstracting the complex code requirements of the AM2302 (DHT22) sensor, and the Adafruit GFX and SSD1306 libraries implementing a display driver pattern, the overall process is simplified. The object initialization includes oled, and dht, which define hardware-specific parameters such as pin assignments, display dimensions, and sensor type within the constructor.

```
void setup() {
  Serial.begin(115200);  // Start serial communication at 115200 baud
  dht.begin();           // Initialize DHT sensor

  // Configure I2C communication for OLED
  Wire.begin(OLED_SDA, OLED_SCL);

  // Initialize OLED display
  if (!oled.begin(SSD1306_SWITCHCAPVCC, 0x3C)) {  // 0x3C = I2C address
    Serial.println(F("SSD1306 allocation failed"));
    while (true)
      ;  // Halt execution if display fails
  }

  // Display boot-up configuration
  oled.setTextSize(1);
  oled.setTextColor(WHITE);
  oled.display();
}
```

The setup routine demonstrates management by phased hardware activation. The serial communication initiates first at 115200 baud, establishing a debugging function before

the initialization of the sensor. The dht.begin() initializes the sensor's internal state, preparing it for periodic sampling. The I2C bus configuration defined by Wire.begin() function explicitly specifies SDA and SCL pins, overriding default assignments for hardware flexibility. The display initialization includes error detection. Error detection is included in display initialization. If the OLED fails to be detected by !oled.begin() syntax, the system enters an infinite loop while printing the debug message via the serial monitor. The text configuration commands are defined by setTextSize(), and setTextColor() functions establish default rendering parameters, while the final display() confirms hardware responsiveness. This sequence shows the initialization of peripherals before the operational loops.

```
void loop() {
  // Read sensor data (250ms sampling time required)
  float humidity = dht.readHumidity();  // Relative humidity (%)
  float tempC = dht.readTemperature();  // Temperature in °C

  // Validate sensor readings
  if (isnan(humidity) || isnan(tempC)) {
    Serial.println(F("DHT read error!"));
    return;  // Skip failed cycle
  }
```

This core sensing block implements data acquisition through the readHumidity() and readTemperature() functions, which encapsulate the DHT22 1-wire communication protocol. Sensor outputs are immediately captured in floating-point variables to preserve measurement precision. The subsequent isnan() function checks to form a critical validation, detecting Not-a-Number (i.e., nan) errors that indicate signal corruption or sensor disconnection. Upon failure, the system logs the error via the serial monitor and returns to the first line rather than halting the entire cycle. The 2-second delay time respects the sensor's sampling rate specification, preventing communication errors in data acquisition.

```
  // Calculate heat index (perceived temperature)
  float heatIndexC = dht.computeHeatIndex(tempC, humidity, false);
```

This single line of code computes the value of the Heat Index. The computeHeatIndex() function implements Rothfusz and Steadman's equations [5], [6]. The regression equation of Rothfusz is shown below.

$$\begin{aligned} \mathrm{HI} = {} & -42.379 + 2.04901523 * \mathrm{T} + 10.14333127 * \mathrm{RH} - .22475541 * \mathrm{T} * \mathrm{RH} - .00683783 * \mathrm{T} * \mathrm{T} \\ & - .05481717 * \mathrm{RH} * \mathrm{RH} + .00122874 * \mathrm{T} * \mathrm{T} * \mathrm{RH} + .00085282 * \mathrm{T} * \mathrm{RH} * \mathrm{RH} \\ & - .00000199 * \mathrm{T} * \mathrm{T} * \mathrm{RH} * \mathrm{RH} \end{aligned}$$

Where T is the temperature in degrees Fahrenheit (F) and RH is the relative humidity in percent. HI is the heat index expressed as an apparent temperature in degrees F. The Rothfusz regression is not appropriate when conditions of temperature and humidity warrant a heat index value below about 80 degrees F. In that case, a simpler formula is applied to calculate values consistent with Steadman's equation.

$$\mathrm{HI} = 0.5 * \{\mathrm{T} + 61.0 + [(\mathrm{T} - 68.0) * 1.2] + (\mathrm{RH} * 0.094)\}$$

For extreme temperature and relative humidity conditions outside of Steadman's data range, the Rothfusz regression is invalid. The computeHeatIndex() function uses both equations where it is best suited to actual temperature and humidity conditions. The false parameter inside the function explicitly selects degrees Celsius (C) units, aligning with the international system of units. The heat index variable is purposefully named heatIndexC to enforce unit awareness, a critical practice in scientific computing.

```
  // Serial Monitor Output
  Serial.print(F("Temp: "));
  Serial.print(tempC);
  Serial.print(F("°C "));
  Serial.print(F("HeatIdx: "));
  Serial.print(heatIndexC);
  Serial.print(F("°C "));
  Serial.print(F("Humidity: "));
  Serial.print(humidity);
  Serial.println(F("%"));

  // OLED Display Routine
  oled.clearDisplay();
  oled.setCursor(0, 0);

  // Display formatted data
  oled.print(F("Temp: "));
  oled.print(tempC);
  oled.write(247);
  oled.println(F("C"));
  oled.print(F("HeatIdx: "));
  oled.print(heatIndexC);
  oled.write(247);
  oled.println(F("C"));
  oled.print(F("Humidity: "));
  oled.print(humidity);
  oled.println(F("%"));
  oled.display();
```

This dual-output strategy that uses a serial channel with print() statements with F() wrapped strings to minimize RAM fragmentation, while the OLED employs clearDisplay() eliminates ghosting text, followed by batched rendering commands that optimize I2C communication. The write(247) command efficiently outputs the degree symbol using its ASCII code. The use of the println() function prints a newline character that creates automatic line advance, maintaining display alignment without manual coordination. All OLED display operations are buffered in memory until the display() command. This pattern demonstrates maximizing bandwidth efficiency through aggregating all characters before displaying.

```
  delay(2000);  // Sampling interval (match sensor limitations)
}
```

The loop termination ends with a blocking delay. The 2000-millisecond interval respects DHT22's minimum sampling rate to maintain proper data acquisition.

2.2.2 Soil Moisture Sensor

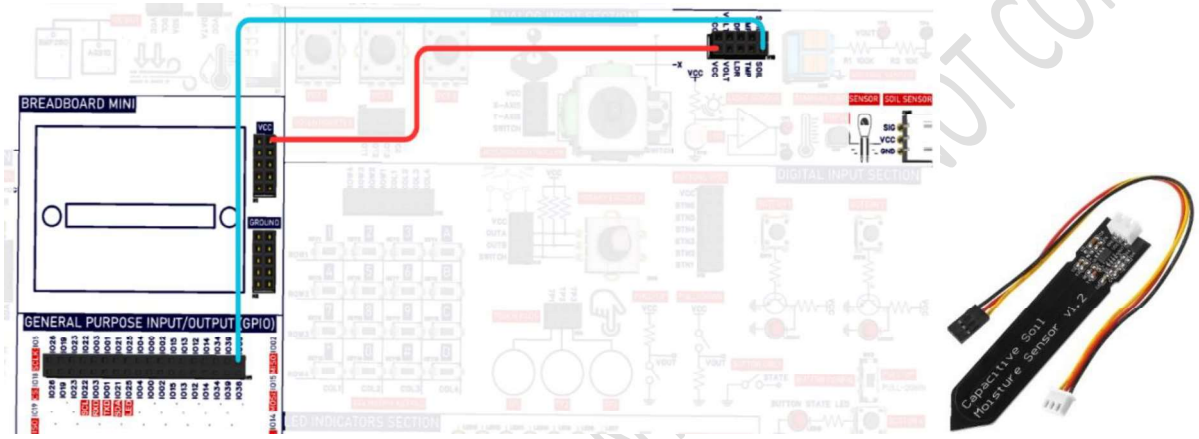
The trainer board includes a capacitive soil moisture sensor interface, as shown in **Figure 3 (a)**. This capacitive soil moisture sensor is different from resistive sensors on the market. It uses the principle of capacitive sensing to detect soil moisture. The problem is that the resistance sensor is easily corroded and has a very short working life. This sensor is the same as the DFRobot-Gravity interface, ensuring compatibility of the interface, and can be directly connected to the Gravity IO expansion board [7].

The sensor has a built-in voltage regulator chip, which supports a 3.3~5.5V working environment. For more details, the capacitive soil moisture sensor specification is shown in **Table 2**.

TABLE 2
Capacitive Soil Moisture Sensor Specification [7]

Parameter	Min	Typical	Max
Voltage	3.3V	5.0V	5.5V
Output Voltage Signal	0		3.0V
Operating Current		5mA	
Interface	PH2.0-3P		
Sensor Probe Dimensions	3.86 x 0.905 inches (L x W)		

Connect the soil sensor in the trainer board interface, and put a wire as shown in **Figure 3 (b)**. Before officially deploying the soil moisture sensor, a calibration process is required. Flash the example code shown in **Listing 2**.



(a) (b)

Figure 3: *Trainer board soil moisture sensor (a) wiring interface, (b) capacitive soil moisture sensor v1.2 component can be interfaced with the platform connector.*

Listing 2 *Reading a capacitive soil moisture sensor*

```
#include <Wire.h>               // I2C communication
#include <Adafruit_GFX.h>       // Core graphics library
#include <Adafruit_SSD1306.h>   // OLED display driver

// Hardware configuration
#define SENSOR_PIN 36    // Use GPIO36 pin
#define ADC_Res_bit 12   // Set the resolution to 9-12 bits (default is 12 bits)
#define OLED_SDA 21      // OLED SDA pin (ESP32 default)
#define OLED_SCL 22      // OLED SCL pin (ESP32 default)
#define OLED_RST -1      // Reset pin (-1 = unused)

// OLED display parameters
#define SCREEN_WIDTH 128  // OLED width in pixels
#define SCREEN_HEIGHT 64  // OLED height in pixels

// Initialize hardware objects
Adafruit_SSD1306 oled(SCREEN_WIDTH, SCREEN_HEIGHT, &Wire, OLED_RST);

// Timing parameters
const uint32_t READ_INTERVAL = 1000;  // Measurement interval (ms)
uint32_t lastReadTime = 0;            // Last measurement timestamp
```

Some parts of the code are the same as Listing 1. This implementation demonstrates a low-cost soil moisture monitoring system that combines analog reading with digital display

using ESP32. The example employs a capacitive soil moisture sensor connected to IO36 (ADC1_CH0), leveraging the ESP32's 12-bit analog-to-digital converter (ADC) resolution has a 0 – 4095 range for precision measurements [8], [9]. What's new in this example is the implementation of non-blocking timing is established using a fixed interval of 1000 milliseconds and a timestamp variable.

```
void setup() {
  Serial.begin(115200);  // Initialize serial at 115200 baud

  analogReadResolution(ADC_Res_bit);  // Configure ADC to 12-bit resolution (0-
4095)

  // Initialize OLED display
  if (!oled.begin(SSD1306_SWITCHCAPVCC, 0x3C)) {  // 0x3C = I2C address
    Serial.println(F("SSD1306 allocation failed"));
    while (true)
      ;  // Halt execution if display fails
  }

  // Display boot-up configuration
  oled.setTextSize(1);
  oled.setTextColor(WHITE);
  oled.display();
}
```

In the setup routine, it calls the analogReadResolution() function to configure the ADC resolution to 12 bits, providing 4096 different values (0 – 4095) [8], [9].

```
void loop() {
  uint32_t currentTime = millis();  // Get current time

  // Execute at fixed intervals without blocking
  if (currentTime - lastReadTime >= READ_INTERVAL) {
    lastReadTime = currentTime;  // Update timing reference

    // Read raw ADC value (0-4095)
    uint16_t rawValue = analogRead(SENSOR_PIN);

    // Convert ADC value to milli-voltage (0-3.3V)
    uint16_t voltage = analogReadMilliVolts(SENSOR_PIN);
```

In the first cycle of the main loop, it calls the millis() function to return time in milliseconds [10]. By comparing the difference between current and previous timing with the defined interval time, it ensures that the microcontroller can handle other tasks during the 1000-millisecond interval between readings. This is critical for applications requiring multitasking while maintaining precise timing for sensor sampling. The analogRead() function is called to return ADC raw value ranges from 0 to 4095. The input voltage is directly measured in millivolts using analogReadMilliVolts() function [8], [11]. This example reads the soil moisture sensor using the ADC and displays the raw ADC value and the voltage in millivolts on the serial monitor and the OLED display.

```
    // Print formatted data to serial monitor
    Serial.print("ADC value:");
    Serial.print(rawValue);  // Raw ADC value (0-4095)
    Serial.print("\t ADC voltage:");
    Serial.print(voltage);  // Input voltage in milli-volts
    Serial.println("mV");
```

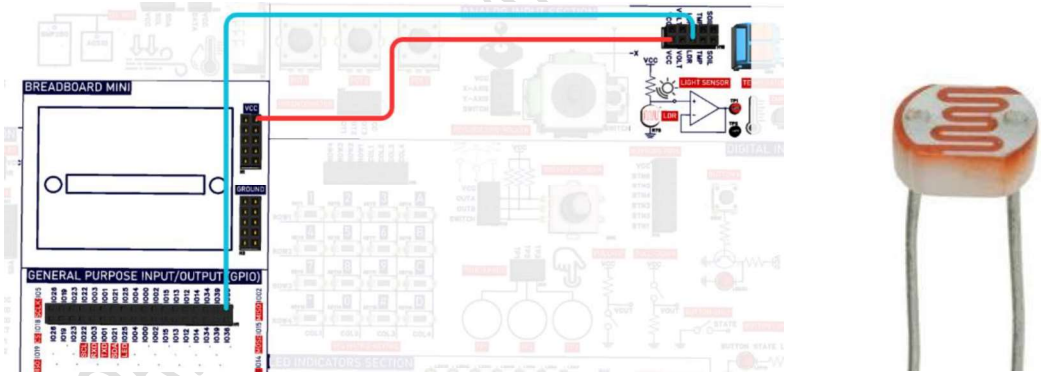
```
    // OLED Display Routine
    oled.clearDisplay();
    oled.setCursor(0, 0);

    // Display formatted data
    oled.print(F("ADC value: "));
    oled.println(rawValue);
    oled.print(F("ADC voltage: "));
    oled.print(voltage);
    oled.println(F("mV"));
    oled.display();
  }
}
```

You can define threshold values for defined soil moisture conditions from the sensor data. First, place the sensor in the air to read the analog value, which represents the reading when it is dry. Then take a cup of water and place the sensor probe in it, and the depth shall not exceed the warning line printed on the probe, as indicated in the datasheet [7]. At this time, record the read ADC value, which represents 100% soil moisture. Take note that the ADC value is inversely proportional to the moisture level. Thus, the value should be the smallest in the direct water. Since the sensor is based on a capacitive sensing method, sensor readings might vary by the depth and tightness of the soil.

2.2.3 Light Intensity Sensor

The trainer board has a light sensor based on a photoresistor based resistor which is also known as light light-dependent resistor (LDR), shown in **Figure 4 (b)** in series with a 10K resistor. Between them is the LDR pin, where the analog signal is connected to the IO interface.



(a) (b)

Figure 4: *Trainer board soil moisture sensor (a) wiring interface, (b) capacitive soil moisture sensor v1.2 component can be interfaced with the platform connector.*

Table 3 summarizes the specifications of the LDR used on the board.

TABLE 3
GL5528 Photoresistor Light Sensor Specification [12]

Parameter	Min	Typical	Max
Voltage			150V
Wavelength	0		540nm
Cell Resistance at 10lx	8kΩ	10kΩ	20kΩ
Dark Resistance			1MΩ
Response Time			30ms
Gamma Value γ		0.65	

The voltage on the LDR pin depends on the light intensity that strikes the sensor. You can read this voltage by connecting the LDR pin to an analog input GPIO interface, as shown in **Figure 13 (a)**. The LDR can be used to measure lux, a unit of illuminance. This change in resistance can be used to measure and correlate with lux levels. Wokwi [13] provided a table showing the relationship between the lux level, resistance (R), and the voltage across the photoresistor at different light conditions.

TABLE 4
Light Illuminance Relationship with Photoresistor Parameters [13]

Light Condition	Illumination (lux)	LDR Resistance	LDR Voltage
Full moon	~0.1lx	1.25MΩ	4.96V
Deep twilight	~1lx	250kΩ	4.81V
Twilight	~10lx	50kΩ	4.17V
Computer monitor	~50lx	16.2kΩ	3.09V
Stairway lighting	~100lx	9.98kΩ	2.50V
Office lighting	~400lx	3.78kΩ	1.37V
Overcast day	~1,000lx	1.99kΩ	0.83V
Full daylight	~10,000lx	397Ω	0.19V
Direct sunlight	~100,000lx	79Ω	0.04V

The test condition is when the supply voltage is 5V, the gamma value γ is 0.7, and the cell resistance at 10lx RL10 is 50kΩ. The equation to calculate the lux level based on the photoresistor's resistance-illuminance (R) relationship is expressed below.

$$lux = (RL10 * 1000 * 10^{\gamma}/R)^{\frac{1}{\gamma}} * 10$$

This equation transforms raw resistance measurements into photometrically valid illuminance values. To demonstrate this, follow the example shown in **Listing 3**.

Listing 3 *Reading the lux level from the photoresistor (LDR) sensor*

```
#include <Wire.h>               // I2C communication
#include <Adafruit_GFX.h>       // Core graphics library
#include <Adafruit_SSD1306.h>   // OLED display driver

// Hardware configuration
#define SENSOR_PIN 36   // Use GPIO36 pin
#define ADC_Res_bit 12  // Set the resolution to 9-12 bits (default is 12 bits)
#define OLED_SDA 21     // OLED SDA pin (ESP32 default)
#define OLED_SCL 22     // OLED SCL pin (ESP32 default)
#define OLED_RST -1     // Reset pin (-1 = unused)

// OLED display parameters
#define SCREEN_WIDTH 128  // OLED width in pixels
#define SCREEN_HEIGHT 64  // OLED height in pixels

// These constants should match the photoresistor's "gamma" and "RL10" attributes
#define GAMMA 0.65
#define RL10 10.00

// Initialize hardware objects
Adafruit_SSD1306 oled(SCREEN_WIDTH, SCREEN_HEIGHT, &Wire, OLED_RST);

// Timing parameters
const uint32_t READ_INTERVAL = 1000;  // Measurement interval (ms)
```

```
uint32_t lastReadTime = 0;            // Last measurement timestamp
```

This implementation demonstrates a low-cost solution of an illuminance measurement system that combines analog sensing and physical modeling. The circuit is composed of a photoresistor in a voltage-divider configuration with a 10kΩ reference resistor, interfaced to GPIO36 (ADC1_CH0). As you can see, the gamma value GAMMA and cell resistance at 10lx illuminance RL10 are defined according to the sensor datasheet. The rest of the code is the same as the previous examples.

```
void setup() {
  Serial.begin(115200);  // Initialize serial at 115200 baud

  analogReadResolution(ADC_Res_bit);  // Configure ADC to 12-bit resolution (0-
4095)

  // Initialize OLED display
  if (!oled.begin(SSD1306_SWITCHCAPVCC, 0x3C)) {  // 0x3C = I2C address
    Serial.println(F("SSD1306 allocation failed"));
    while (true)
      ;  // Halt execution if display fails
  }

  // Display boot-up configuration
  oled.setTextSize(1);
  oled.setTextColor(WHITE);
  oled.display();
}

void loop() {
  uint32_t currentTime = millis();  // Get current time

  // Execute at fixed intervals without blocking
  if (currentTime - lastReadTime >= READ_INTERVAL) {
    lastReadTime = currentTime;  // Update timing reference

    // Convert ADC value to voltage (0-3.3V)
    double voltage = analogReadMilliVolts(SENSOR_PIN) / 1000.00;

    // Calculate equivalent photoresistor resistance using voltage divider
    double resistance = (voltage * 10000.00) / (5.00 - voltage);

    // Compute for lux level
    double lux = pow(RL10 * 1e3 * pow(10, GAMMA) / resistance, (1 / GAMMA)) *
10.00;
```

The main loop performs the following computations, which include voltage measurement, resistance calculation, and gamma-corrected illuminance calculation. The analogReadMilliVolts() function returns a measured voltage in milli-volts in a specified ADC pin. Thus, its value needs to be divided by 1000 to convert it into voltage. This value is used to calculate the resistance across the LDR sensor through the voltage-divider equation, as expressed below.

$$R_{LDR} = V_{ADC}R_{10k}/(5 - V_{ADC})$$

Using the lux level equation above, the lux level can be calculated with the gamma value and the power-law relationship. The gamma power exponents model the photoresistor's non-linear response. The lux level value might reach infinity when the photo-resistive sensor is in a very bright environment.

```
    if (isfinite(lux)) {
      // Print formatted data to serial monitor
      Serial.print("LDR voltage:");
      Serial.print(voltage);
      Serial.print("V \t Resistance:");
      Serial.print(resistance);
      Serial.print("Ω \t Lux level:");
      Serial.print(lux);
      Serial.println("lx");

      // OLED Display Routine
      oled.clearDisplay();
      oled.setCursor(0, 0);

      // Display formatted data
      oled.print(F("LDR voltage:"));
      oled.print(voltage);
      oled.println(F("V"));
      oled.print(F("Res:"));
      oled.print(resistance);
      oled.println(F("ohm"));
      oled.print(F("Lux level:"));
      oled.print(lux);
      oled.println(F("lx"));
      oled.display();
    } else {
      Serial.println("Illuminance cannot be measured!");
    }
  }
}
```

The isfinite() function is called to check if the value is not infinite, negative, or a not-a-number before printing sensor data. If the value is finite, the function returns true; else false. Corrupted calculations occur when the voltage across the LDR approaches 0V (i.e., direct light) or 5V (i.e., total darkness).

2.2.4 Smoothing Analog Readings

Many analog sensors are typically based on a variable resistor that changes its resistance corresponding to changes in the physical quantity to be measured. It can be a physical quantity like light, temperature, or pressure [14]. For example, as mentioned above, the LDR is a type of variable resistor where resistance changes based on light intensity. This variable resistance produces a voltage signal, representing a physical value. This signal is subject to fluctuations because no circuit is ideal that is unsusceptible to electrical noise. In some cases, to calculate a quantity you are interested in precisely, you need a stable voltage signal.

The moving average filter is a simple digital signal processing (DSP) technique to reduce noise in the signals [15]. It is defined as a method of averaging a number of points from the input signal to its array size. The moving average filter algorithm is expressed as

$$y[n] = \frac{1}{N} \sum_{j=0}^{N-1} x[n + i]$$

In this equation, x[n] is the input signal and an array of values, y[n] is the average value, and N is the number of points or array size used in the moving average. As new data points

are added, a new dynamic average value is obtained. The array of values moves with the new data point and bypasses its past value according to its array size to compute another average value. To better understand this equation, a simple moving average can be expressed as

$$Y_1 = \frac{X_1 + X_2 + X_3 + \ldots + X_n}{N}$$

To implement this, the Arduino code in **Listing 4** sequentially stores 20 values from randomly generated numbers into an array, one by one. With each new value, the sum of all the numbers is generated and divided by its size, producing an average value which is then used to smooth outlying data. Because this averaging takes place each time a new value is added to the array, rather than waiting for 20 new values, for instance, there is no lag time in calculating this running average.

Listing 4 *Moving average filter algorithm Arduino implementation*

```
#define numReadings 20  // Value to determine the size of the readings array
#define ADC_Res_bit 12  // Set the resolution to 9-12 bits (default is 12 bits)

int readings[numReadings];  // The readings from the analog input
int readIndex = 0;          // The index of the current reading
int total = 0;              // The running total
int average = 0;            // The average value
```

This example is based on the Arduino built-in example, smoothing readings from an analog input [16]. The implementation demonstrates a circular buffer-based moving average filter. The selection of 20 samples represented by numReadings variable reflects the size of the array. The larger it is, the better it reduces noise, but it introduces more delays and requires more memory. A shorter array size responds faster to signal changes but has less noise reduction. A circular buffer readings[] stores recent data samples. A running accumulator total maintains the sum of current values. An index pointer readIndex implements first-in-first-out (FIFO) data replacement in the buffer array. The filter works by continuously updating the buffer array with new data and replacing only the oldest sample during each iteration.

```
void setup() {
  // Initialize serial communication with computer
  Serial.begin(115200);

  // Configure ADC to 12-bit resolution (0-4095)
  analogReadResolution(ADC_Res_bit);

  // Initialize the pseudo-random number generator
  randomSeed(10);

  // Initialize all the readings to 0
  for (int thisReading = 0; thisReading < numReadings; thisReading++) {
    readings[thisReading] = 0;
  }
}
```

The setup routine explicitly initializes the circular buffer array to zero. It serves filter output will gradually converge to the true signal value as real samples fill the buffer. Without initialization, the array would contain random memory values, causing unpredictable filter output during the first N samples.

```
void loop() {
  // Subtract the last reading
```

```
total = total - readings[readIndex];
// Read from the sensor
readings[readIndex] = 10 + random(-5, 5);

// Add the reading to the total
total = total + readings[readIndex];

// Calculate the average value
average = total / numReadings;
```

The loop routine requires only a few arithmetic operations, regardless of array size. Subtracting the outgoing value from the running sum, as preparation to compute a new running sum with the new data sample. Signal generation is used to create a testing environment. The constant value 10 represents the underlying signal and generates uniformly distributed noise using the random() function [17] with amplitude ±5, representing a 50% peak-to-peak noise level relative to the signal. The uniform distribution of random numbers creates what signal processing engineers call white noise [18]. This type of noise is particularly well-suited for demonstrating low-pass filter effectiveness. After the signal generation, it accumulates to a running sum, then performs a division operation to complete the moving average equation.

```
// Send it to the Arduino Serial plotter
Serial.print("Raw:");
Serial.print(readings[readIndex]);
Serial.print(",");
Serial.print("Average:");
Serial.println(average);
```

The data samples and computed average value are formatted for the Arduino Serial Plotter to visualize real-time data streams. The comma-separated value (CSV) format with descriptive labels allows the plotter to automatically parse and graph multiple data series simultaneously [19]. **Figure 5** shows how the moving average algorithm responds, represented by the orange line, to the noisy signal represented by the blue line.

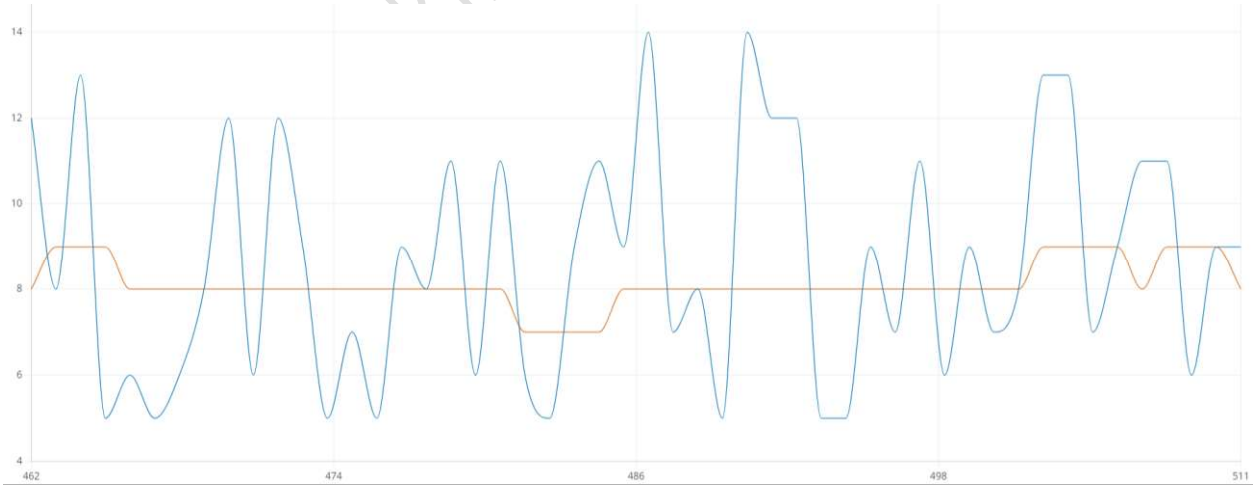


Figure 5: *Visualizing the moving average filter output with the generated noisy test signal.*

```
// Advance to the next position in the array
readIndex = readIndex + 1;

// if we're at the end of the array...
if (readIndex >= numReadings) {
  // ...wrap around to the beginning
  readIndex = 0;
```

```
  }

  // Delay in between reads for stability
  delay(30);
}
```

After each calculation, the pointer index is increased by 1 for the new readings and filter calculations. It checks if the pointer index exceeds the buffer size, if it does, the pointer index is reset to zero, which automatically overwrites the oldest sample and ensures continuous data flow without buffer overflows. A delay of 30 milliseconds at the end of the loop routine creates an effective 33.3Hz sampling rate for the demonstration.

2.2.5 Reading Button Switch

Push buttons or switches are the basic user interface components in embedded systems, enabling digital input through mechanical contact points when you press them. When connecting a button to a microcontroller, it is necessary to connect a pull-up or pull-down resistor to the power supply rail [20]. The trainer board has configurable pull-up/pull-down resistors via a sliding switch, and two buttons only, without any internal resistors, as shown in **Figure 6**.

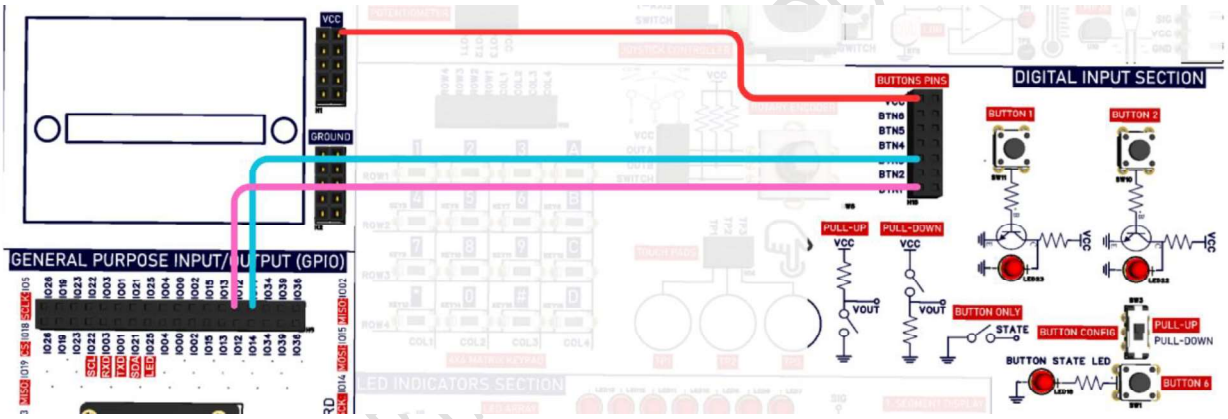


Figure 6: *Trainer board configurable buttons and their wiring interface with GPIO pins.*

If the button config switch position is up, the button is in pull-up configuration, the resistor connects between the microcontroller's GPIO pin and VCC, yielding a high logic state when the button is open and a low state when pressed. A config switch is positioned down, the button switches to a pull-down configuration that connects the resistor between GPIO and GND, producing a low state when open and a high state when pressed [21]. It has a built-in circuitry whenever the button is pressed, triggers an LED button state indicator.

The example below is a demonstration of reading a digital signal from a GPIO. The code is based on Arduino built-in examples for reading push button state and how to utilize MCU's internal resistor for button interfacing [20], [22]. Furthermore, this example demonstrates hardware and software resistor configuration to understand the push buttons operational modes with external and internal resistor termination.

Listing 5 *Reading digital input from a push button*

```
// Define the pin number for the button
#define BUTTON1_PIN 12  // Button only that uses MCU pull-up resistor
#define BUTTON3_PIN 14  // Configurable resistor push button
```

The implementation shows two distinct button interfacing, one utilizing an external pull-up or pull-up resistors button configuration (i.e., Button 3 as BUTTON3_PIN on GPIO 12) and

another uses the microcontroller's internal pull-up or pull-down resistor (i.e., Button 1 as BUTTON1_PIN on GPIO 14) capability through software initialization.

```
void setup() {
  // Initialize serial communication at 115200 baud for debugging
  Serial.begin(115200);

  // Set BUTTON3_PIN as an input to read button state
  pinMode(BUTTON3_PIN, INPUT);

  // Set BUTTON1_PIN as an input to read button state
  // MCU initialize pull-up/pull-down resistor
  // uncomment the line of code  below if using Button 1
  // pinMode(BUTTON1_PIN, mode);   // mode: INPUT_PULLDOWN / INPUT_PULLUP

  // Set the built-in LED pin as an output (indicator)
  pinMode(LED_BUILTIN, OUTPUT);
}
```

The software routine initializes serial communication at 115200 baud for debugging purposes, the pinMode() function configures GPIO pins as inputs with appropriate pull-up or pull-down resistors, and a built-in LED as an output indicator for visual feedback. If you interface a push button without a resistor, when it is in the open state, the input pin will be floating, resulting in unpredictable results. Pull-up or pull-down resistors are used to ensure proper reading when the switch is open. The purpose of this resistor is to pull the pin to a known state when the switch is open. A 10KΩ resistor is usually chosen, as it is a low enough value to reliably prevent a floating input, and at the same time, a high enough value to not draw too much current when the switch is closed [21].

When using Button 1, you must uncomment the line of code configuring Button 1 to utilize the microcontroller's internal 45kΩ resistor [23], through INPUT_PULLDOWN or INPUT_PULLUP mode, thereby establishing a known state at the GPIO pin, without the need for an external resistor.

```
void loop() {
  // Read the state of the button (HIGH if pressed, LOW if not)
  int button3State = digitalRead(BUTTON3_PIN);

  // Uncomment the line of code below if using button 1
  // then comment out the line of code  above
  // int button1State = digitalRead(BUTTON1_PIN);

  // If the button state is HIGH
  // If using button 1, then use button1State variable
  if (button3State == HIGH) {
    Serial.println("Turn on LED");    // Print status to Serial Monitor
    digitalWrite(LED_BUILTIN, HIGH);  // Turn ON the built-in LED
  } else {
    Serial.println("Off LED");       // Print status to Serial Monitor
    digitalWrite(LED_BUILTIN, LOW);  // Turn OFF the built-in LED
  }
}
```

The main loop of the program implements a polling-based button state detection using a digitalRead() function to read the button's digital state, followed by conditional logic that controls the built-in LED and provides serial monitor and visual feedback based on the button's current state [21], [22]. For Button 3, the circuit has an external pull-down or pull-up resistor. If it

is configured pull-down resistor, it initially LOW (0V) state when open and a HIGH (3.3V) state when the button is pressed. The conditional if (button3State == HIGH) checks if the button state is HIGH voltage level as a logical trigger. Use button1State if you are using Button 1. The serial output statements are within Serial.println() function and built-in LED provide real-time button state monitoring, serving both as a debugging tool for the hardware-software interface synchronization.

In conclusion, you must remember that resistor termination defines not just electrical behavior, but logical truth in digital systems. An internal MCU resistor is preferred for space-constrained devices such as wearables and IoT sensors. However, for industrial interfaces, an external resistor is required for safety and standard compliance. An internal resistor saves board space but increases susceptibility to electromagnetic interference. Furthermore, redundant termination or double-buffered reading is recommended to address the effects of mechanical vibration.

2.3 Power Management

A critical element of any wireless sensor node is reducing the system's power consumption. The most important factor affecting the lifespan of a wireless sensor is its power consumption, as sensor nodes are often powered by batteries. Energy awareness can facilitate optimization of all aspects of design and operation. To optimize sensor lifetime, protocols must be developed from the beginning with the aim of optimizing energy resource management.

2.3.1 ESP32 Sleep Modes

Energy management-related operating modes are typically designed for microprocessors. ESP32 has a power management algorithm that regulates the CPU frequency and enters sleep mode on the chip to run the application with the lowest possible power consumption, considering the power requirements of the application components [24], [25]. Depending on the specific needs of the application, ESP32 offers developers the ability to choose from five different performance modes, as listed below.

- **Active mode**: The chip radio is powered on. The chip has three functions: listening, transmitting, and receiving packets.
- **Modem sleep mode**: The Central Processing Unit (CPU) is ready for operation, and the clock is configurable. The Wi-Fi/Bluetooth baseband and radio are disabled.
- **Light sleep mode**: The CPU is paused. The Real-Time Clock (RTC) memory and RTC peripherals, as well as the Ultra Low Power (ULP) coprocessor, are running. All wake events (i.e., RTC timer or external interrupts) wake the chip.
- **Deep sleep mode**: Only RTC memory and RTC peripherals are turned on. Wi-Fi and Bluetooth connection data are stored in RTC memory. The ULP coprocessor is functional.
- **Hibernation mode**: The internal 8 MHz oscillator and ULP coprocessor are disabled. RTC recovery memory is turned off. Only an RTC timer on the slow clock and certain RTC GPIOs are active. The RTC timer or RTC GPIOs can wake the chip from sleep.

Each mode has its own set of features and power-saving functions. Among the ESP32 series of SoCs, when Wi-Fi is enabled, the chip switches between Active and Modem-sleep modes. Therefore, power consumption changes accordingly. In Modem-sleep mode, the CPU frequency changes automatically. The frequency depends on the CPU load and the peripherals used. During Deep sleep, when the ULP coprocessor is powered on, peripherals such as GPIO and RTC I2C can operate. The summary of ESP32 power modes and their respective power consumption is shown in **Table 5**.

TABLE 5
Power Consumption by Power Modes of ESP32 Series

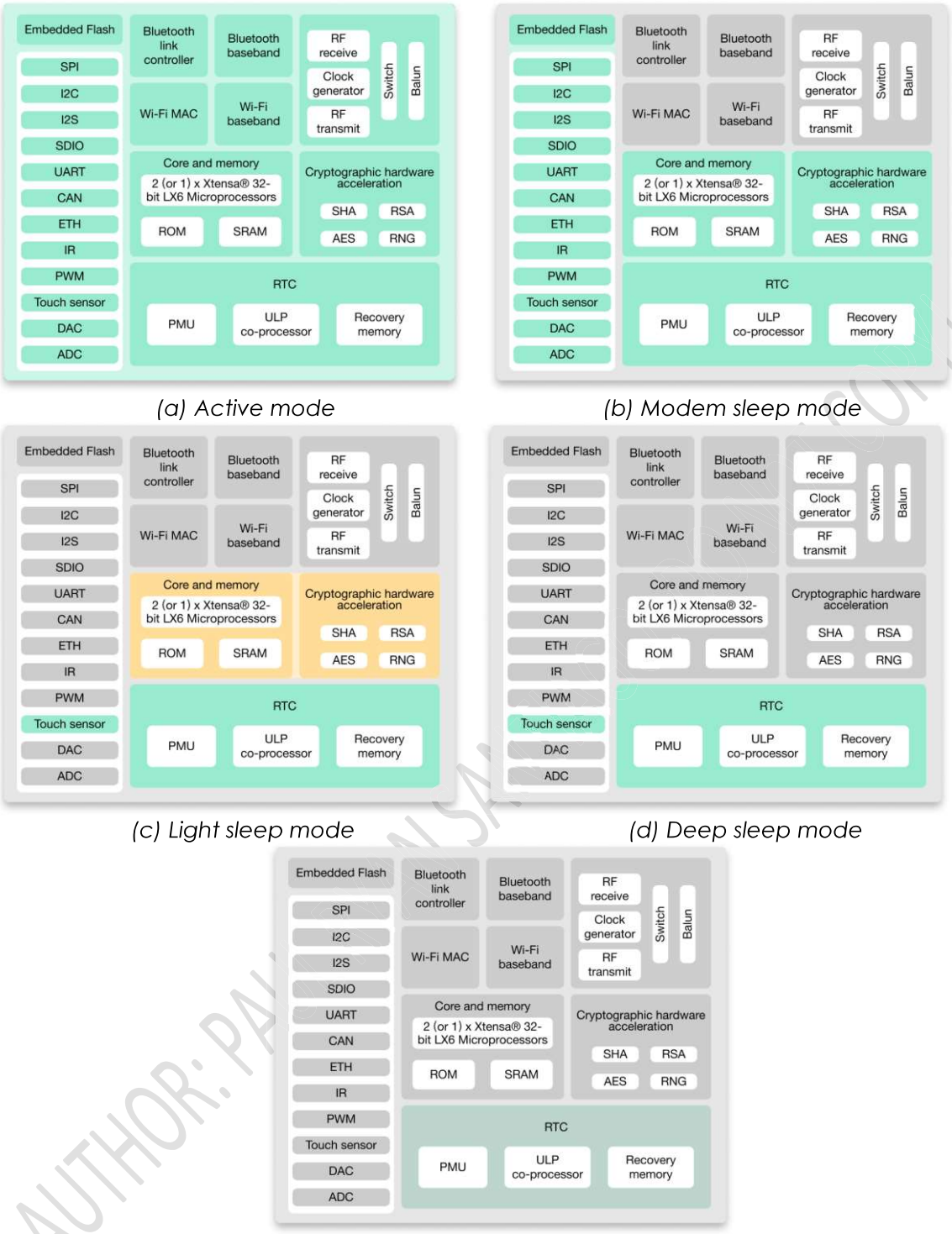
Power Mode	Description			Power Consumption
Active (RF working)	Wi-Fi Tx packet			180 ~ 240 mA
	BT/BLE Tx packet			130 mA
	Wi-Fi/BT Rx and listening			95 ~ 100 mA
Modem sleep	The CPU is powered on	240 MHz	Dual-core chip	30 mA ~ 68 mA
			Single-core chip	N/A
		160 MHz	Dual-core chip	27 mA ~ 44 mA
			Single-core chip	27 mA ~ 34 mA
		Normal speed 80 MHz	Dual-core chip	20 mA ~ 31 mA
			Single-core chip	20 mA ~ 25 mA
Light sleep	-			0.8 mA
Deep sleep	The ULP coprocessor is powered on			150 µA
	ULP sensor-monitored pattern			100 µA @1% duty
	RTC timer + RTC memory			10 µA
Hibernation	RTC timer only			5 µA

The documentation provided by Espressif [24] lists each module that makes up the ESP32. As shown in **Figure 7(a)**, all these modules operate when the microcontroller is in active mode. Nonetheless, the ESP32 can be used in sleep mode, which reduces energy consumption by turning off some of its modules. For instance, **Figure 7 (b)** shows that when a microcontroller is in modem sleep mode, all the circuits that power the radio, Bluetooth, and Wi-Fi modules are turned off, but the CPU and all peripheral modules that allow the microcontroller to connect to the outside world via its GPIOs continue to function.

The ESP32 module must be woken up at specific intervals to switch between sleep to active mode if Bluetooth or Wi-Fi connections still need to be maintained. This sleep pattern is called an association sleep pattern. Digital peripherals, RAM, and CPUs are clock-gated in light sleep mode, as shown in **Figure 7 (c)**. CPUs and peripherals resume operation once light sleep mode is terminated. To reduce the power consumption of digital circuitry, clock gating is implemented by deactivating the switching states of flip-flops via the clock pulses. Disabling the state-switching feature saves a significant amount of power.

When running on battery power, the ESP32's deep sleep mode is incredibly effective and uses little power. As shown in **Figure 7 (d)**, this mode shuts down all digital peripherals, the CPU, and most of the RAM. The RTC peripherals, which include the ULP coprocessor and memories, are the only components of the chip that continue to function. The primary CPU is turned off while the system is in deep sleep mode. On the other hand, the ULP coprocessor can still receive timer signals and measure data from the sensors. It can also take sensor readings and wake the main system when a timer is set or when it detects an event. This association sleep pattern is called the ULP sensor monitored pattern [25].

The chip's primary memory is disabled along with the CPU. Everything stored in this memory is deleted and can no longer be restored. When the chip wakes up, data can be retrieved even in deep sleep mode using RTC memory. The chip then resets itself to start over, losing any data not stored in RTC recovery memory. This means that the program will run again from the beginning.



(a) Active mode *(b) Modem sleep mode*

(c) Light sleep mode *(d) Deep sleep mode*

(e) Hibernation mode

Figure 7: *Espressif ESP32 association sleep pattern. Adopted from Steph's µLab* [26].

Unlike deep sleep mode, the chip turns off the ULP coprocessor and internal 8MHz clock when it enters hibernation sleep mode. There is no way to save data in hibernation mode as the RTC memory is turned off. Except for an RTC timer and a few active RTC GPIOs, everything else is turned off, as shown in **Figure 7 (e)**. This means the chip can be woken up from sleep using the RTC GPIOs or the RTC timer. This significantly reduces power consumption.

2.3.2 ESP32 Sleep Arduino Implementation

The Arduino code in **Listing 6** demonstrates the ESP32's power management capabilities to implement a power-saving mode with internal and external wake sources. It uses GPIO pins as an external wake-up source, a GPIO, and a touch pin as shown in **Figure 8**.

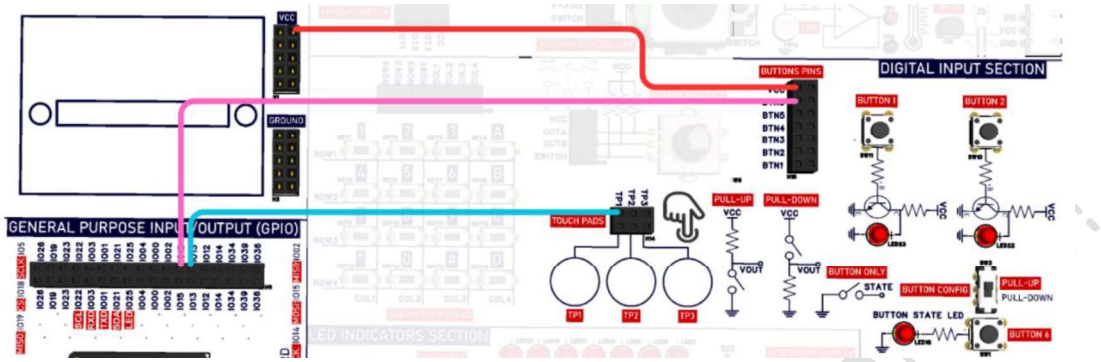


Figure 8: *Trainer board touch pin and push button interface as sleep wake-sources.*

The provided Arduino code shows a comprehensive implementation for managing low-power sleep modes on an ESP32 microcontroller, demonstrating the utilization of light sleep, deep sleep, and hibernate sleep modes, alongside the configuration of wake-up sources such as timer, touch pin, and GPIO pin.

Listing 6 ESP32 Sleep Modes with Timer, Touch, and GPIO Wake-Up Source

```
#include "esp_sleep.h"
#include "driver/touch_sens.h"
#include "driver/gpio.h"
#include "driver/rtc_io.h"
```

The code is written in the Arduino framework and leverages necessary Espressif IoT Development Framework (ESP-IDF) libraries for sleep management (i.e., esp_sleep.h), internal touch sensor pin (i.e., driver/touch_sens.h), GPIO operations (i.e., driver/gpio.h), and RTC GPIO support (i.e., driver/rtc_io.h).

```
// Define type of wake-up source (TIMER_WAKEUP | TOUCH_WAKEUP | GPIO_WAKEUP)
#define TIMER_WAKEUP

// Define RTC memory variable to store data
RTC_DATA_ATTR int bootCount = 0;

// Define touch sensor pin object
touch_pad_t touchPin;
```

A macro definition TIMER_WAKEUP specifies the wake-up source using the internal real-time clock (RTC), with options for TOUCH_WAKEUP or GPIO_WAKEUP, allowing you to select the desired wake-up source by defining one of these definitions. A variable bootCount, stored in the RTC memory using the RTC_DATA_ATTR attribute, a non-volatile memory that persists in its value across sleep cycles, serving as a counter for the number of system wake-ups.

```
void setup() {
  // Initialize serial at 115200 baud
  Serial.begin(115200);

#if defined(TIMER_WAKEUP)
// Time in seconds the ESP32 will go to sleep
#define TIME_TO_SLEEP 10
```

```
  // Enable internal timer as wake-up source
  esp_sleep_enable_timer_wakeup(TIME_TO_SLEEP * 1000000ULL);
#elif defined(TOUCH_WAKEUP)
// Touchpad sensitivity threshold
#define TOUCH_THRESHOLD 40
  // Enable touch sensor 5 (GPIO 13) as an external wake-up source
  touchSleepWakeUpEnable(T5, TOUCH_THRESHOLD);
#elif defined(GPIO_WAKEUP)
  // Enable GPIO 33 as the external wake-up source when triggering HIGH
  esp_sleep_enable_ext0_wakeup(GPIO_NUM_15, 1);
#endif
```

In the setup routine, depending on the defined wake-up source, the preprocessor directive configures the corresponding wake-up mechanism. For TIMER_WAKEUP, the system is configured to wake up the processor from sleep mode after a specified duration TIME_TO_SLEEP by 10 seconds. Using the esp_sleep_enable_timer_wakeup() function, enable the timer sleep wakeup source that accepts a time interval in microseconds. Thus, TIME_TO_SLEEP, a value of 10, needs to be multiplied by 1,000,000 to convert it from seconds to microseconds as required by the ESP-IDF application program interface (API), and the ULL (unsigned long long integer) suffix ensures proper handling to prevent overflow in time calculations.

For TOUCH_WAKEUP, the touch sensor on GPIO 13 (touch pad T5) is enabled as a wake-up source with a sensitivity threshold TOUCH_THRESHOLD of 40, configured via the touchSleepWakeUpEnable() function. In the case of GPIO_WAKEUP, GPIO 15 is set as an external wake-up source triggered by a high logic level using the esp_sleep_enable_ext0_wakeup() function. It is important to note that there are only two GPIOs that can be used as an external wakeup source (i.e., ext0 and ext1), and only the RTC GPIO pins can be used [25], [27].

```
  // Increment the boot number and print it every reboot
  ++bootCount;
  Serial.println("Boot count: " + String(bootCount));

  // Print the wakeup reason for ESP32 and touchpad
  Serial.println("**********************************");
  print_wakeup_reason();
  print_wakeup_touchpad();
  Serial.println("**********************************");
```

Following wake-up source configuration, the bootCount variable is incremented and printed to the serial monitor to indicate the number of wake-up events. Followed by print_wakeup_reason() and print_wakeup_touchpad() functions to log the cause of the wake-up source that are defined in the later part of the code.

```
  Serial.println("Entering sleep...");
  // Uncomment one of the following lines based on the sleep mode you want to use
  //esp_light_sleep_start();
  //esp_deep_sleep_start();
  esp_hibernate_start();
}
```

Finally, the code initiates a sleep cycle by uncommenting one of three functions, which include esp_light_sleep_start(), esp_deep_sleep_start(), and esp_hibernate_start() [25]. The code defaults to hibernate mode, as indicated by the uncommented esp_hibernate_start() call, and is configured for the lowest power state.

```
void loop() {
  // This will never be reached
}
```

The loop routine remains empty as the ESP32 enters sleep mode and only executes the setup routine again upon waking, a common practice in low-power applications.

```
void esp_hibernate_start() {
  // RTC peripherals power domain set to auto
  esp_sleep_pd_config(ESP_PD_DOMAIN_RTC_PERIPH, ESP_PD_OPTION_AUTO);
  esp_sleep_pd_config(ESP_PD_DOMAIN_RTC_SLOW_MEM, ESP_PD_OPTION_AUTO);
  esp_sleep_pd_config(ESP_PD_DOMAIN_RTC_FAST_MEM, ESP_PD_OPTION_AUTO);
```

The esp_hibernate_start() function is a custom implementation designed to configure the ESP32 for hibernate sleep. Within this function, it configures the power domains for RTC peripherals, RTC slow memory, and RTC fast memory into automatic power management using the esp_sleep_pd_config() function with the ESP_PD_OPTION_AUTO option [25]. It keeps the RTC power domain enabled in sleep mode if one of the wake-up options is required. Otherwise, it turns off.

```
  // Turn off ESP32 main power domain
  esp_sleep_pd_config(ESP_PD_DOMAIN_MAX, ESP_PD_OPTION_OFF);
```

The main power domain ESP_PD_DOMAIN_MAX is explicitly turned off using ESP_PD_OPTION_OFF to achieve the lowest power consumption [25]. The ESP_PD_DOMAIN_MAX is a constant defined in the ESP32's SDK that represents the maximum or all-encompassing power domain, typically the digital and analog components within the ESP32 architecture, such as the CPU, memory, and non-RTC peripherals, including Wi-Fi, Bluetooth, SPI, I2C, UART, ADC, DAC, and GPIO controllers. This configuration ensures that the main ESP32 power domain is turned off, thereby minimizing power consumption to the lowest possible level.

```
  // Isolate all GPIO before entering sleep mode
  esp_sleep_config_gpio_isolate();
```

The esp_sleep_config_gpio_isolate() function is used to configure all ESP32 GPIO pins to an isolated state before the ESP32 enters a sleep mode. The term isolation usually means setting the GPIO pin to a high impedance state or tristate logic, which deactivates the internal pull-up or pull-down resistors of the GPIO pins. When the ESP32 goes into deep sleep mode, the main power domains are turned off, but the GPIO pins can still maintain electrical connections to external circuits. This may result in unintentional power consumption caused by floating pins, pull-up or pull-down resistors, or external connections that drive current through the pins, also known as current leakage [25], [27]. By placing all the GPIO pins in a high-impedance state, it effectively disconnects the internal GPIO circuitry from all external connections and prevents current leakage, which is especially important in ultra-low power applications where the goal is to reduce power consumption to a micro-ampere range for battery-powered operation. However, for pins that have dual functionality as a GPIO and an RTC GPIO where the pins remain operational in a sleep mode. This function ensures that only RTC functionality is maintained when isolating internal GPIO circuits. However, for pins that have dual functionality as both regular GPIO and RTC GPIO, the pins remain functional during sleep mode. The esp_sleep_config_gpio_isolate() function ensures that only the RTC functionality is preserved while isolating the GPIO internal circuitry.

```
  // Isolate all RTC GPIO to reduce current leakage
  rtc_gpio_isolate(GPIO_NUM_0);
  rtc_gpio_isolate(GPIO_NUM_2);
  rtc_gpio_isolate(GPIO_NUM_4);
```

```
  rtc_gpio_isolate(GPIO_NUM_12);
  rtc_gpio_isolate(GPIO_NUM_13);
  rtc_gpio_isolate(GPIO_NUM_14);
  rtc_gpio_isolate(GPIO_NUM_15);
  rtc_gpio_isolate(GPIO_NUM_25);
  rtc_gpio_isolate(GPIO_NUM_26);
  rtc_gpio_isolate(GPIO_NUM_27);
  rtc_gpio_isolate(GPIO_NUM_32);
  rtc_gpio_isolate(GPIO_NUM_33);
  rtc_gpio_isolate(GPIO_NUM_34);
  rtc_gpio_isolate(GPIO_NUM_35);
  rtc_gpio_isolate(GPIO_NUM_36);
  rtc_gpio_isolate(GPIO_NUM_37);
  rtc_gpio_isolate(GPIO_NUM_38);
  rtc_gpio_isolate(GPIO_NUM_39);
```

The rtc_gpio_isolate() function is called on each RTC GPIO pin to disconnect the internal RTC GPIO input, output, pull-up, and pull-down resistors, and hold state feature from an RTC GPIO pin. If the RTC GPIO is not required during sleep mode, this function helps minimize leakage current.

```
  // Enter sleep mode
  esp_deep_sleep_start();
}
```

The hibernation sleep mode also uses the esp_deep_sleep_start() function to enter a sleep state. It is simply a deep sleep mode configured with power management, in which the main power area of the ESP32 and the IO peripherals are off, the RTC and memory peripherals are partially on, and only the internal timing circuit as a wake-up source is running, in order to achieve the lowest possible power consumption.

```
// Method to print the reason by which ESP32 has been awaken from sleep
void print_wakeup_reason() {
  esp_sleep_wakeup_cause_t wakeup_reason;

  wakeup_reason = esp_sleep_get_wakeup_cause();

  switch (wakeup_reason) {
    case ESP_SLEEP_WAKEUP_EXT0: Serial.println("Wakeup caused by external signal
using RTC_IO"); break;
    case ESP_SLEEP_WAKEUP_EXT1: Serial.println("Wakeup caused by external signal
using RTC_CNTL"); break;
    case ESP_SLEEP_WAKEUP_TIMER: Serial.println("Wakeup caused by timer"); break;
    case ESP_SLEEP_WAKEUP_TOUCHPAD: Serial.println("Wakeup caused by touchpad");
break;
    case ESP_SLEEP_WAKEUP_ULP: Serial.println("Wakeup caused by ULP program");
break;
    default: Serial.printf("Wakeup was not caused by deep sleep: %d\n",
wakeup_reason); break;
  }
}

// Method to print the touchpad by which ESP32 has been awaken from sleep
void print_wakeup_touchpad() {
  touchPin = (touch_pad_t)esp_sleep_get_touchpad_wakeup_status();

#if CONFIG_IDF_TARGET_ESP32
  switch (touchPin) {
```

```
    case 0: Serial.println("Touch detected on GPIO 4"); break;
    case 1: Serial.println("Touch detected on GPIO 0"); break;
    case 2: Serial.println("Touch detected on GPIO 2"); break;
    case 3: Serial.println("Touch detected on GPIO 15"); break;
    case 4: Serial.println("Touch detected on GPIO 13"); break;
    case 5: Serial.println("Touch detected on GPIO 12"); break;
    case 6: Serial.println("Touch detected on GPIO 14"); break;
    case 7: Serial.println("Touch detected on GPIO 27"); break;
    case 8: Serial.println("Touch detected on GPIO 33"); break;
    case 9: Serial.println("Touch detected on GPIO 32"); break;
    default: Serial.println("Wakeup not by touchpad"); break;
  }
#else
  if (touchPin < TOUCH_PAD_MAX) {
    Serial.printf("Touch detected on GPIO %d\n", touchPin);
  } else {
    Serial.println("Wakeup not by touchpad");
  }
#endif
}
```

Two callback functions, print_wakeup_reason() and print_wakeup_touchpad(), provide feedback information about the wake-up source. The print_wakeup_reason() function retrieves the wake-up cause using esp_sleep_get_wakeup_cause() function and employs a switch statement to print a descriptive message corresponding to the reason, such as external GPIO and touch pad wake-up sources, timer, or ultra-low-power (ULP) coprocessor. Similarly, print_wakeup_touchpad() identifies the specific touch pad that triggered a wake-up event, determining the touch pad number to its corresponding GPIO pin.

2.4 Summary

This chapter focuses on the practical construction of a sensor node device for the IoT and WSN applications using the ESP32 microcontroller. It provides you with a comprehensive guide in building a sensor node using the ESP32, integrating environmental sensors such as temperature, humidity, soil moisture, and light, and implementing sleep-mode low-power strategies. You have learned to read and process analog and digital signals with C++ code in the Arduino, to control the rate of timing and sampling, and to design routines for low-power sensors. By combining hardware interfacing and software development, and energy-efficient designs, you should be able to build a working low-power sensor node capable of collecting meaningful data from sensors in real time. The practical knowledge you will gain is the basis for IoT applications and sets the stage for advanced features, including integration with mobile applications, web dashboards, and cloud connectivity, as discussed in the following chapters.

2.5 Mini Project 1: A Low-Power Sensor Node for Plant Monitoring

2.5.1 Introduction

This mini project enables you to apply the practical knowledge gained to design a smart indoor plant monitoring system using the ESP32 microcontroller. Smart sensors are defined as devices that integrate a sensing unit and data processing to directly convert analog signals to digital signals while performing preprocessing tasks such as digital filtering, calibration, and decision making [28]. You will design a system that monitors critical environmental parameters such as air temperature, humidity, soil moisture, and ambient light, designed for the needs of a specific indoor plant. By integrating various environmental sensors, processing their data, implementing autonomous decision-making via displaying results on an OLED screen, driving actuators when conditions deviate from ideal ranges, and managing

power consumption, this project prepares you to develop advanced WSN and IoT applications. Upon completion, a smart plant monitor will be developed, providing real-time insights into a plant's well-being and timely alerts when specific environmental conditions require attention.

Upon completing this project, you will be able to:

- Interface multiple environmental sensors with the ESP32 for real-time data acquisition.
- Process sensor data to evaluate environmental conditions against plant-specific thresholds.
- Implement decision-making logic to trigger actuation based on meaningful data.
- Display sensor readings on an OLED screen for user interaction.
- Utilize deep sleep mode to optimize power consumption in a battery-powered smart sensor.

2.5.2 Components Needed

Hardware Components:

1. LILYGO TTGO LoRa32 OLED ESP32 Development Board
2. Digital Temp & Humidity Sensor (AM2302/DHT22)
3. Capacitive Soil Moisture Sensor
4. Light Intensity Sensor (Photoresistor (LDR) GL5528)
5. OLED Display (SSD1306 128x64 pixels)
6. LED Indicator
7. Jumper Wires
8. Power Source (Battery or USB power)

Software Tools:

1. Arduino IDE
2. Digikey Scheme-it Free Flow Chart Creation or similar applications

2.5.3 System Architecture

The system architecture that integrates the ESP32 with sensors, a display, and an actuator, as illustrated in a conceptual block diagram, is shown in **Figure 9**.

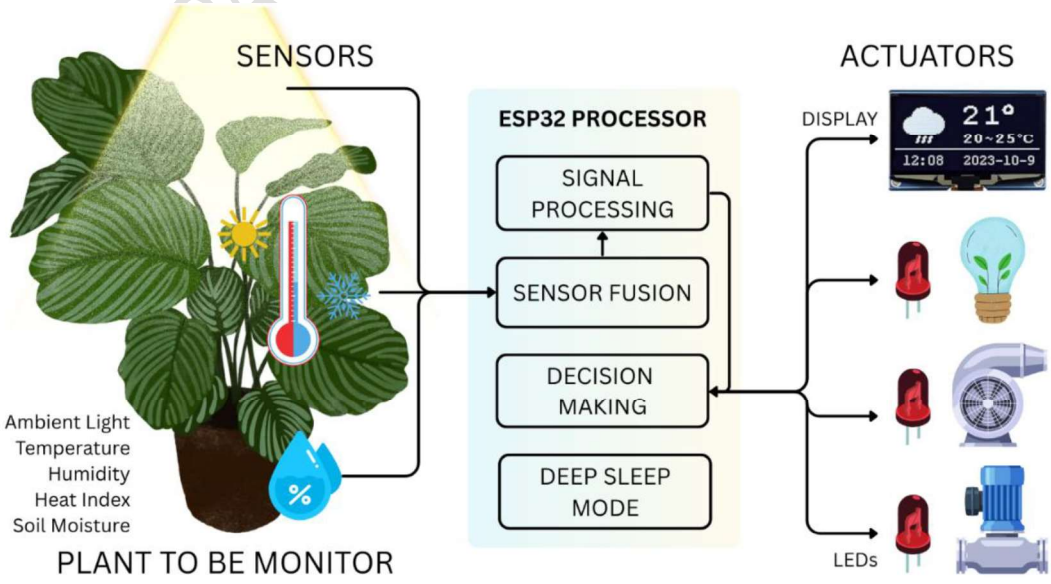


Figure 9 *Mini-Project 1: A Low-Power Sensor Node for Plant Monitoring System Architecture*

The sensors collect raw environmental data and are read by the ESP32 microcontroller by means of a fusion of sensors. This is the process of combining signals obtained from different sensor sources to produce a more valuable and precise output than individual sensors can provide [29]. The ESP32 processes this data, applying algorithms for conversion, smoothing, and making sense of the data based on the ideal plant conditions. The ESP32 makes decisions based on the information extracted from the environmental sensors, which are displayed on the OLED screen, and activates events represented by LEDs as actuators, such as the water pump, the air-conditioning, and the lights for cultivation. After completing its cycle of monitoring, processing, decision-making, display, and actuation, the ESP32 enters a deep sleep mode to conserve energy until an internal timer wakes it for the next cycle.

2.5.4 Project Plan

You must complete the following tasks to meet the project objectives.

1. Select a specific indoor plant and research its ideal environmental conditions. For example, the optimum moisture content of the soil for crops depends on the specific species of plant, but for most plants, the optimum soil moisture content is between 21 and 80 percent [30].
2. You are free to assemble the circuit by connecting sensors, the OLED display, and the LED to the ESP32 from the trainer board.
3. Write Arduino code to collect raw environmental data. Makes the raw sensor data meaningful and compares it with the predefined parameters that represent the ideal environmental conditions for a specific plant. Once ideal ranges are determined, they must be translated into quantifiable thresholds within the Arduino code. Then implement decision logic to determine whether the plant needs attention. This includes conditional statements that compare the current values with the defined ideal range. Show the current reading on the OLED screen and switch the LED if any of the parameters are outside the ideal value of the plant.
4. Configure the ESP32 to enter deep sleep for a specified duration. However, it must maintain the LED activation state during the sleep period if it is triggered based on the last sensor readings.

2.5.5 Guidelines and Tips

- Sensor fusion involves combining data from multiple, distinct sensors to achieve a more comprehensive, accurate, and better understanding of an environment than what a single sensor could provide alone. For example, if both temperature and humidity are high, this means that watering is needed.
- Implement a timing interval to capture and display sensor data at regular intervals of a ten-second sleep period.
- Note that the ADC2 channel is inactive in deep sleep mode. To overcome this limitation, switch the ADC sensor pins to the ADC1 channel. This ensures continuous sensor reading even when the ESP32 is in power-saving mode.
- You may create a graphic design for the SSD1306 OLED display using an online tool to display sensor data. Online Arduino AdafruitGFX SSD1306 OLED Code Generator (https://arduinogfxtool.netlify.app).
- Drive the actuators, represented by LEDs, during deep sleep mode using the RTC-enabled GPIO of the ESP32 to maintain output control even in a low-power state. Espressif's documentation [25], [27] on GPIO and RTC-GPIO will serve as a valuable resource for configuring this aspect of the sensor node.

References

[1] M. A. Matin, M. M. Islam, M. A. Matin, and M. M. Islam, "Overview of Wireless Sensor Network," *Wireless Sensor Networks - Technology and Protocols*, Sep. 2012, doi: 10.5772/49376.

[2] L. Hamami and B. Nassereddine, "Application of wireless sensor networks in the field of irrigation: A review," *Comput Electron Agric*, vol. 179, Dec. 2020, doi: 10.1016/j.compag.2020.105782.

[3] S. Metia, H. A. D. Nguyen, and Q. P. Ha, "IoT-Enabled Wireless Sensor Networks for Air Pollution Monitoring with Extended Fractional-Order Kalman Filtering," *Sensors 2021, Vol. 21, Page 5313*, vol. 21, no. 16, p. 5313, Aug. 2021, doi: 10.3390/S21165313.

[4] "Temperature and humidity module AM2302 Product Manual", Accessed: Jul. 23, 2025. [Online]. Available: www.aosong.com

[5] Adafruit, "DHT Sensor Library: DHT Class Reference." Accessed: Jul. 23, 2025. [Online]. Available: https://adafruit.github.io/DHT-sensor-library/html/class_d_h_t.html#a0d23921017e3d827e49bfd136b40a6aa

[6] Weather Prediction Center, "Heat Index Equation," NOAA/ National Weather Service. Accessed: Jul. 23, 2025. [Online]. Available: https://www.wpc.ncep.noaa.gov/html/heatindex_equation.shtml

[7] "HW-101 HW-moisture sensor V1.2 Specification".

[8] Espressif, "ADC: Arduino ESP32 Documentation," Espressif Systems (Shanghai) Co., Ltd. Accessed: Jul. 26, 2025. [Online]. Available: https://docs.espressif.com/projects/arduino-esp32/en/latest/api/adc.html

[9] Arduino Docs, "analogReadResolution() | Arduino Documentation," Arduino. Accessed: Jul. 26, 2025. [Online]. Available: https://docs.arduino.cc/language-reference/en/functions/analog-io/analogReadResolution/

[10] Arduino Docs, "millis() | Arduino Documentation," Arduino. Accessed: Jul. 26, 2025. [Online]. Available: https://docs.arduino.cc/language-reference/en/functions/time/millis/

[11] Arduino Docs, "analogRead() | Arduino Documentation," Arduino. Accessed: Jul. 26, 2025. [Online]. Available: https://docs.arduino.cc/language-reference/en/functions/analog-io/analogRead/

[12] "GL5528(10-20) | Datasheet | JCHL(Shenzhen Jing Chuang He Li Tech) | LCSC Electronics." Accessed: Jul. 24, 2025. [Online]. Available: https://lcsc.com/datasheet/lcsc_datasheet_2410121312_JCHL-Shenzhen-Jing-Chuang-He-Li-Tech-GL5528-10-20_C10081.pdf

[13] Wokwi, "Wokwi Photoresistor Sensor Reference | Wokwi Docs," CodeMagic LTD. Accessed: Jul. 24, 2025. [Online]. Available: https://docs.wokwi.com/parts/wokwi-photoresistor-sensor

[14] Analog Devices, "Chapter 3:Sensors Section." Accessed: Jul. 25, 2025. [Online]. Available: https://www.analog.com/media/en/training-seminars/design-handbooks/Basic-Linear-Design/Chapter3.pdf

[15] S. W. Smith, "The Scientist and Engineer's Guide to Digital Signal Processing," 2nd ed., California Technical Publishing, 1999, ch. 15. Accessed: Jul. 25, 2025. [Online]. Available: https://www.analog.com/media/en/technical-documentation/dsp-book/dsp_book_ch15.pdf

[16] Arduino Docs, "Smoothing Readings From an Analog Input | Arduino Documentation," Arduino. Accessed: Jul. 26, 2025. [Online]. Available: https://docs.arduino.cc/built-in-examples/analog/Smoothing/

[17] Arduino Docs, "random() | Arduino Documentation," Arduino. Accessed: Jul. 26, 2025. [Online]. Available: https://docs.arduino.cc/language-reference/en/functions/random-numbers/random/

[18] ScienceDirect Topics, "White Noise - An Overview," Elsevier. Accessed: Jul. 26, 2025. [Online]. Available: https://www.sciencedirect.com/topics/engineering/white-noise

[19] "Using the Serial Plotter Tool | Arduino Documentation." Accessed: Jul. 25, 2025. [Online]. Available: https://docs.arduino.cc/software/ide-v2/tutorials/ide-v2-serial-plotter/
[20] Arduino Docs, "How to Wire and Program a Button | Arduino Documentation," Arduino. Accessed: Jul. 26, 2025. [Online]. Available: https://docs.arduino.cc/built-in-examples/digital/Button/
[21] Arduino Docs, "INPUT | INPUT_PULLUP | OUTPUT | Arduino Documentation," Arduino. Accessed: Jul. 26, 2025. [Online]. Available: https://docs.arduino.cc/language-reference/en/variables/constants/inputOutputPullup/
[22] Arduino Docs, "InputPullupSerial | Arduino Documentation," Arduino. Accessed: Jul. 26, 2025. [Online]. Available: https://docs.arduino.cc/built-in-examples/digital/InputPullupSerial/
[23] Espressif, "GPIO: Arduino ESP32 Documentation," Espressif Systems (Shanghai) Co., Ltd. Accessed: Jul. 26, 2025. [Online]. Available: https://docs.espressif.com/projects/arduino-esp32/en/latest/api/gpio.html
[24] Espressif Systems, "ESP32 Series Datasheet," 2022. [Online]. Available: https://www.espressif.com/en/support/download/documents.
[25] Espressif Systems, "Sleep Modes - ESP32," ESP-IDF Programming Guide. Accessed: Nov. 26, 2023. [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/sleep_modes.html
[26] S. Calderoni, "Sleep Modes," Steph's µLab. Accessed: Nov. 17, 2023. [Online]. Available: https://m1cr0lab-esp32.github.io/sleep-modes/sleep-modes/
[27] Espressif Systems, "GPIO & RTC GPIO - ESP32," ESP-IDF Programming Guide. Accessed: Nov. 26, 2023. [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/peripherals/gpio.html#gpio-rtc-gpio
[28] Science Direct, "Intelligent Sensor - An Overview | ScienceDirect Topics," Elsevier. Accessed: Jul. 29, 2025. [Online]. Available: https://www.sciencedirect.com/topics/engineering/intelligent-sensor
[29] Science Direct, "Sensor Fusion - An Overview | ScienceDirect Topics," Elsevier. Accessed: Jul. 29, 2025. [Online]. Available: https://www.sciencedirect.com/topics/engineering/sensor-fusion
[30] M. Redmon, "How Moist Should Soil Be? A Soil Moisture Gardening Guide," Tempest. Accessed: Jul. 29, 2025. [Online]. Available: https://tempest.earth/resources/how-moist-should-soil-be/