

weather station

Overview

In this tutorial, you will master the usage of the DHT11 temperature and humidity sensor and the water level sensor, and be able to accurately collect temperature, humidity, and water level data in the current environment. These data can be used to simulate core monitoring indicators such as temperature, humidity, and precipitation required by a weather station. Component Required:

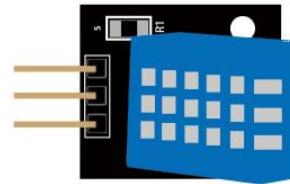
- (1) x Elegoo ESP32
- (2) x 400-hole breadboards
- (1) x DHT11 Temperature and Humidity module
- (1) x water level sensor
- (4) x F-M wires (Female to Male DuPont wires)

Component Introduction

Temp and humidity sensor:

DHT11 digital temperature and humidity sensor is a composite Sensor which contains a calibrated digital signal output of the temperature and humidity. The dedicated digital modules collection technology and the temperature and humidity sensing technology are applied to ensure that the product has high reliability and excellent long-term stability. The sensor includes a resistive moisture sensor and a NTC temperature measurement devices, and connects with a high-performance 8-bit microcontroller.

Applications: HVAC, dehumidifier, testing and inspection equipment, consumer goods, automotive, automatic control, data loggers, weather stations, home appliances, humidity regulator, medical and other humidity measurement and control.



Product parameters

| | |
|---|---|
| Relative humidity: | 1m / s air 6s Hysteresis: |
| Resolution: 8Bit | <± 0.3% RH |
| Repeatability: ±1% RH | Long-term stability: <± 0.5% RH / yr in |
| Accuracy: At 25°C ±5% RH | |
| Interchangeability: fully interchangeable | |
| Response time: 1 / e (63%) of 25°C 6s | |

| | |
|-----------------------|---|
| Temperature: | Response time: 1 / e (63%) 10S |
| Resolution: 8Bit | Electrical Characteristics |
| Repeatability: ±0.2°C | Power supply: DC 3.5~5.5V |
| Range: At 0°C ±50°C | Supply Current: measurement 0.3mA standby 60 μA |

Pin Description:

1. the VDD power supply 3.5~5.5V DC.
2. DATA serial data, a single bus.
3. GND ground, the negative power.

water level sensor

A water sensor brick is designed for water detection, which can be widely used in sensing the rainfall, water level, even the liqueate leakage. The brick is mainly composed of three parts: an electronic brick connector, a $1\text{ M}\Omega$ resistor, and several lines of bare conducting wires.

This sensor works by having a series of exposed traces connected to ground.

Interlaced between the grounded traces are the sensor traces.

The sensor traces have a weak pull-up resistor of $1\text{ M}\Omega$. The resistor will pull the sensor trace value high until a drop of water shorts the sensor trace to the grounded trace. Believe it or not this circuit will work with the digital I/O pins of your UNOR3 board or you can use it with the analog pins to detect the amount of water induced contact between the grounded and sensor traces.

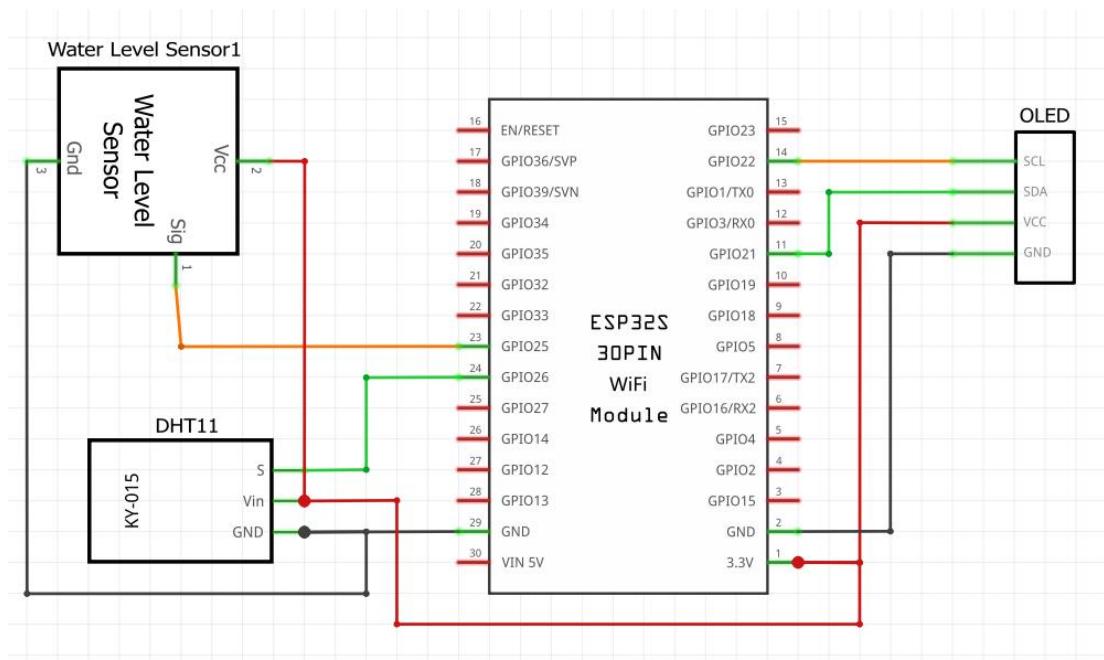
This item can judge the water level through with a series of exposed parallel wires stitched to measure the water droplet/water size. It can easily change the water size to analog signal, and output analog value can directly be used in the program function, then to achieve the function of water level alarm.

It has low power consumption, and high sensitivity.

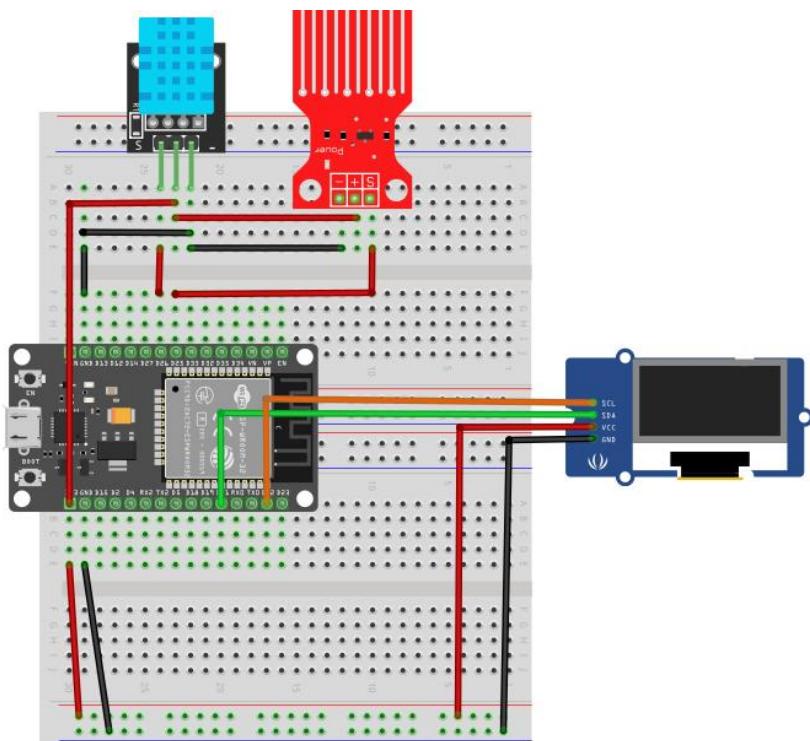
Features:

- 1、Working voltage: 5V
- 2、Working Current: <20ma
- 3、Interface: Analog
- 4、Width of detection: 40mm × 16mm
- 5、Working Temperature: 10°C~30°C
- 6、Output voltage signal: 0~2.4V





Connection Schematic



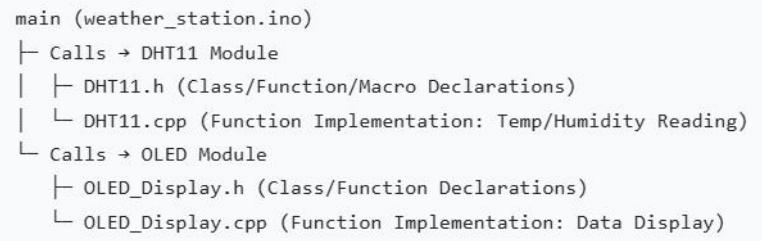
Wiring diagram

Code

After wiring, please open the program in the code folder- **weather station** and click UPLOAD to upload the program. See Lesson 5 in part 1 for details about program uploading if there are any errors.

Before you can run this, make sure that you have installed the <**DHT**> library or re-install it, if necessary. Otherwise, your code won't work.

After opening the program, you will see multiple files, including the .ino main program file, .h header files, and .cpp source files. This program is a weather station data collection and display system, whose core function is to read temperature and humidity data via the DHT11 sensor, collect water level data through the water level sensor, and display all data on the OLED screen in real time. The module call relationship of each file is shown in the figure below.



In the .h header files, there usually include function declarations, definitions of input/output pins, macro definitions, and declarations of relevant data types. By reading the header files, you can quickly grasp the core functions of the corresponding module, the external calling interfaces it provides, and hardware dependencies (such as pin configurations) without checking the specific implementation details.

```
weather_station.ino DHT11.cpp DHT11.h OLED_Display.cpp OLED_Display.h
1 // DHT11.h
2 #ifndef DHT11_H
3 #define DHT11_H
4
5 #include <dht_nonblocking.h>
6
7 #define DHT_SENSOR_TYPE DHT_TYPE_11
8 #define DHT_SENSOR_PIN 26
9
10 class DHT_nonblocking_sensor {
11 public:
12     DHT_nonblocking dht_sensor;
13
14     DHT_nonblocking_sensor() : dht_sensor(DHT_SENSOR_PIN, DHT_SENSOR_TYPE) {}
15
16     bool measure(float *temperature, float *humidity);
17 }
18
19 #endif // DHT11_H
```

Similarly, the .cpp source files serve as the implementation carrier for the header files. Their core function is to implement the function logic declared in the .h files, as well as complete variable definitions and initialization. They contain specific code execution processes (such as sensor data reading timing, OLED screen drawing logic, etc.) and are the actual implementers of module functions. Without modifying the header files, you can optimize module performance or adjust functional details only by modifying the code in .cpp files, without affecting the calling interfaces of other modules.

```
weather_station.ino DHT11.cpp DHT11.h OLED_Display.cpp OLED_Display.h
1
2 #include "DHT11.h"
3 #include <Arduino.h>
4
5 bool DHT_nonblocking_sensor::measure(float *temperature, float *humidity) {
6     static unsigned long measurement_timestamp = millis();
7
8     /* Measure once every four seconds. */
9     if (millis() - measurement_timestamp > 3000ul) {
10         if (dht_sensor.measure(temperature, humidity) == true) {
11             measurement_timestamp = millis();
12             return true;
13         }
14     }
15
16     return false;
17 }
```

The `OLED_Display.cpp` implements the `OLED_Display` class, leveraging the `Adafruit_SSD1306` driver library to provide initialization, data display (temperature/humidity, water level/rainfall), and screen control functions for an OLED display.

```
weather_station.ino  DHT11.cpp  DHT11.h  OLED_Display.cpp  OLED_Display.h
1 // OLED_Display.cpp
2 #include "OLED_Display.h"
3 #include <Arduino.h>
4
5
6
7 OLED_Display::OLED_Display() : oled(SCREEN_WIDTH, SCREEN_HEIGHT, &Wire, OLED_RESET) {}
8
9 void OLED_Display::setupOLED() {
10     Serial.begin(115200);
11     Wire.begin(21, 22); // SDA=21, SCL=22
12     if(!oled.begin(SSD1306_SWITCHCAPVCC, 0x3C)) { // Address 0x3D for 128x64
13         Serial.println(F("SSD1306 allocation failed"));
14         for(;;);
15     }
16     delay(2000);
17     oled.clearDisplay();
18 }
19
20 void OLED_Display::cleanOled(){
21
22     oled.clearDisplay();
23 }
```

Common Pitfalls

Member Variable-Function Name Collision: The OLED instance name must avoid `display` (which conflicts with the `display()` refresh function in the `Adafruit_SSD1306` library). While the current name `oled` circumvents this, renaming it to `display` by mistake will cause compilation errors.

Mismatched I2C Pins or Device Address: The SDA/SCL pins bound in `Wire.begin(21, 22)` must match the hardware wiring, and the address 0x3C in `oled.begin()` must be adjusted based on the OLED module model (some use 0x3D). A mismatch will trigger the "SSD1306 allocation failed" error and force the program into an infinite loop.

Since the DHT11 sensor collects data once every 2 seconds and returns valid results only when the reading is successful, special attention must be paid to the persistent display logic of the data: the key is to reasonably plan the calling positions of the screen-clearing function (`cleanOled()`) and the display refresh function (`updatedisplay()`). It is recommended to execute the screen-clearing operation at the start of each loop, and call the unified refresh function only after all data (temperature, humidity, water level) have been written to the display buffer. This process of "clear screen first → write all data → unified refresh" can avoid screen flickering caused by refreshing before partial data is written, and ensure that historical valid data will not disappear arbitrarily when the sensor reading fails, achieving a stable and persistent display effect.

```
20 void loop() {
21
22     // ***** 1. Read DHT every 2 seconds *****
23     if (millis() - lastDHTUpdate > 2000) {
24         float temp, hum;
25
26         if (sensor.measure(&temp, &hum) == true) {
27             temperature = temp; // Update only when successful
28             humidity = hum;
29         }
30         lastDHTUpdate = millis();
31     }
32
33     // ***** 2. Read water level ADC *****
34     int waterLevel = analogReadadc_id);
35
36     // ***** Refresh OLED (do not clear screen, no flicker) *****
37     display.cleanOled(); // Clear the screen once is enough, but if layout changes are needed, put it here as well
38     display.displayTemperatureHumidity(temperature, humidity);
39     display.displayWaterLevel(waterLevel); // Corrected function name from "displayWaterLevel" to "displayWaterLevel"
40     display.updatedDisplay(); // Corrected function name from "updatedDisplay" to "updateDisplay"
41
42     delay(2000);
43
44 }
```

Upload the program then open the monitor, we can see the data as below:

(It shows the temperature of the environment, we can see it is 27 to 25 degrees with 45.0% humidity)

Click the Serial Monitor button to turn on the serial monitor. The basics about the serial monitor are introduced in details in part 2 Lesson 4.

