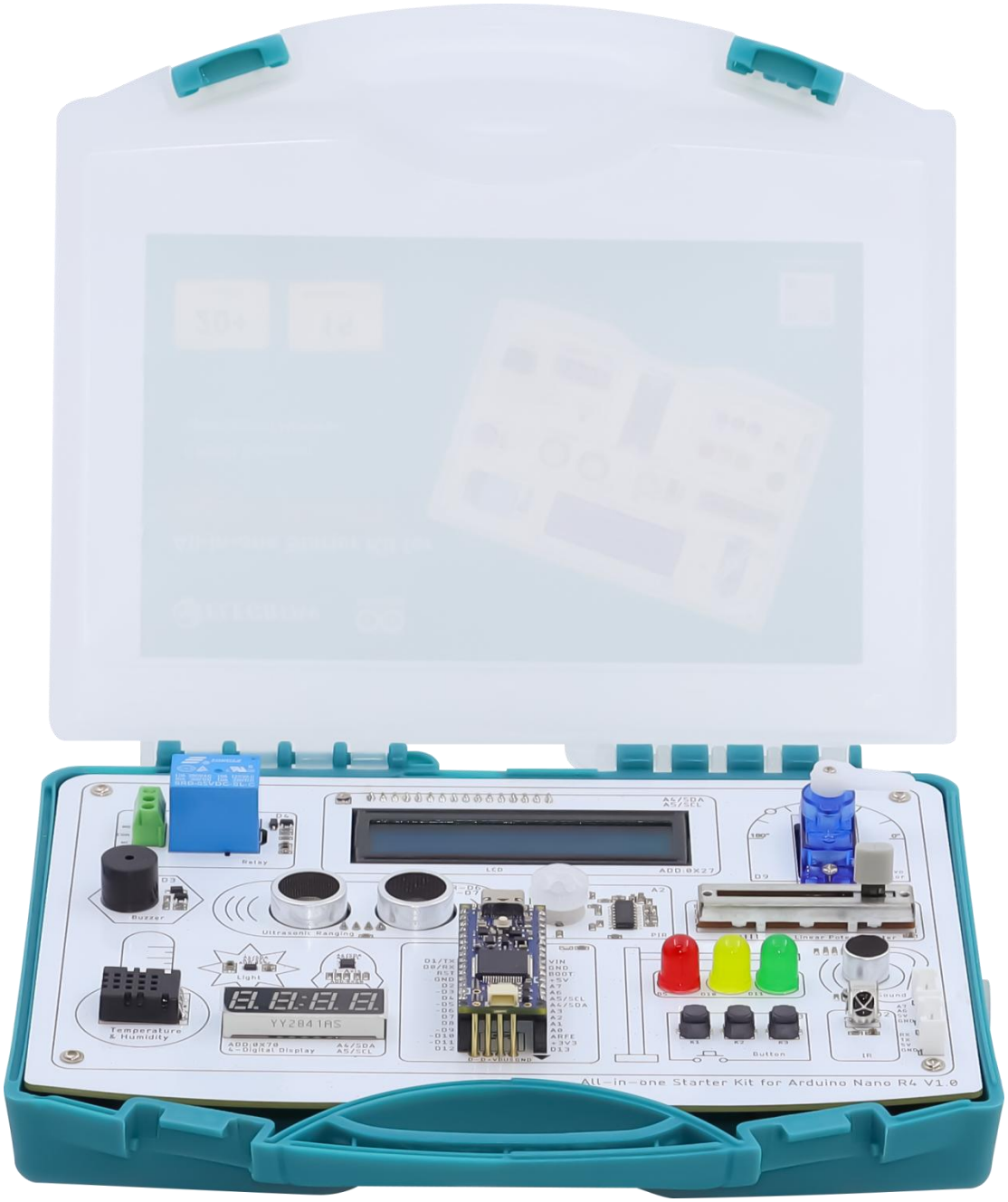
All-in-one Starter Kit for Arduino Nano R4



Introduction

Welcome to the Arduino Nano R4 Development Board User Manual. Let's start exploring the Arduino Nano R4 and learn how to use its wide range of electronic modules!
Don't worry-this board comes with 20 easy-to-understand, engaging, and inspiring lessons that guide you step by step. Here, you'll become familiar with various electronic modules, develop logical thinking, spark creativity, and learn how to program these modules.
The lessons begin with the basics, such as installing the Arduino software, then introduce the Arduino Nano R4 board and its sensors, followed by programming these modules and learning the programming languages used. Finally, you'll learn how to implement practical applications for these sensors. Each step is explained clearly, so even beginners can quickly grasp Arduino programming.
The Arduino Nano R4 board includes 15 electronic modules, each with unique functions and features, making it ideal for beginners. By learning, you'll not only understand the basic principles of sensors, digital and analog signals, analog-to-digital conversion, and programming logic, but also learn how to work with more advanced modules. Most importantly, you'll officially start learning Arduino programming, enhancing your logical thinking skills.
For programming, we'll use the Arduino software. The Arduino platform is one of the world's most popular open-source platforms. It's easy to use, comes with abundant hardware resources (various Arduino boards) and software resources (Arduino IDE), making it one of the best choices for learning programming.

Sensor Accessories

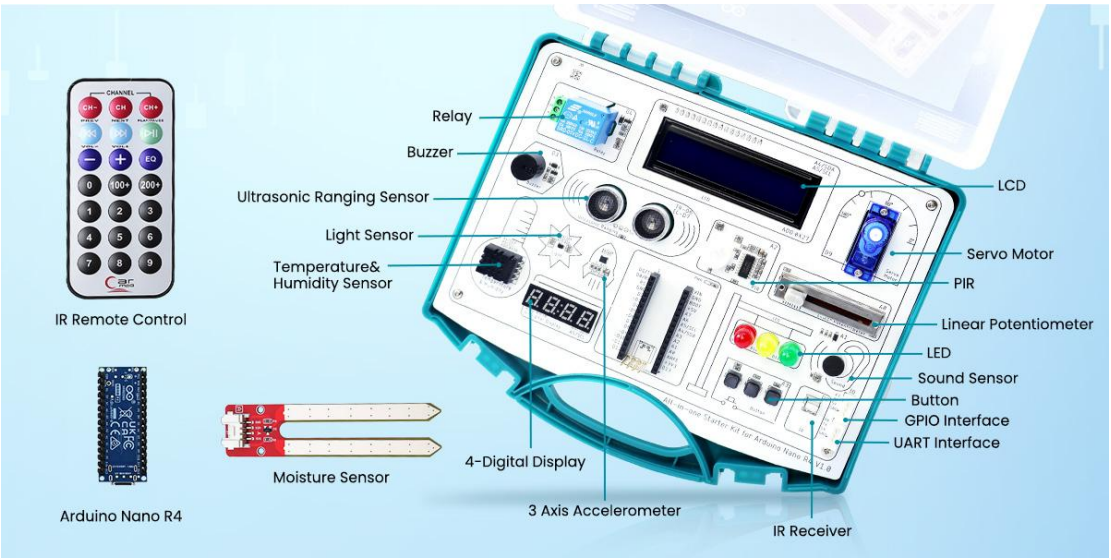
No.	Sensor Accessories	Qty	Notes
1	Temperature & Humidity Sensor	1	
2	Button	3	
3	Ultrasonic Ranging Sensor	1	
4	Light Sensor	1	
5	Linear Potentiometer	1	
6	LED	3	Red + Yellow + Green
7	Buzzer	1	
8	LCD	1	
9	Infrared Remote	1	w/ Remote
10	Relay	1	
11	Servo Motor	1	

12	Sound Sensor	1	
13	3 Axis Accelerometer	1	
14	PIR	1	
15	Moisture Sensor	1	Accessory
16	4-Digital Display	1	

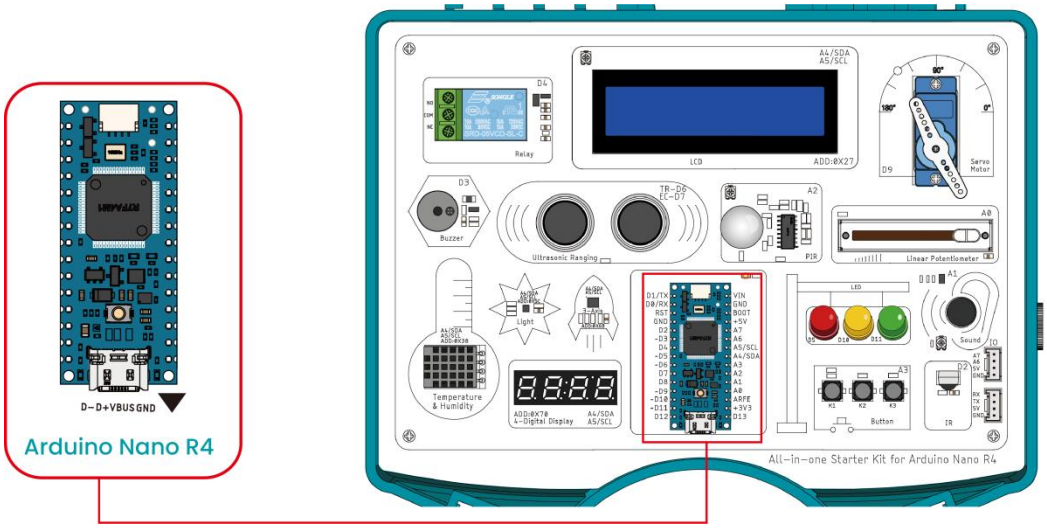
Packing Accessories

Accessory	Qty	Notes
All-in-one Starter Kit for Arduino NANO R4	1	
Type-C Cable	1	
Moisture Sensor	1	with Cable
IR Remote Control	1	
Servo Motor Arm	1 Set	
Arduino NANO R4	1	

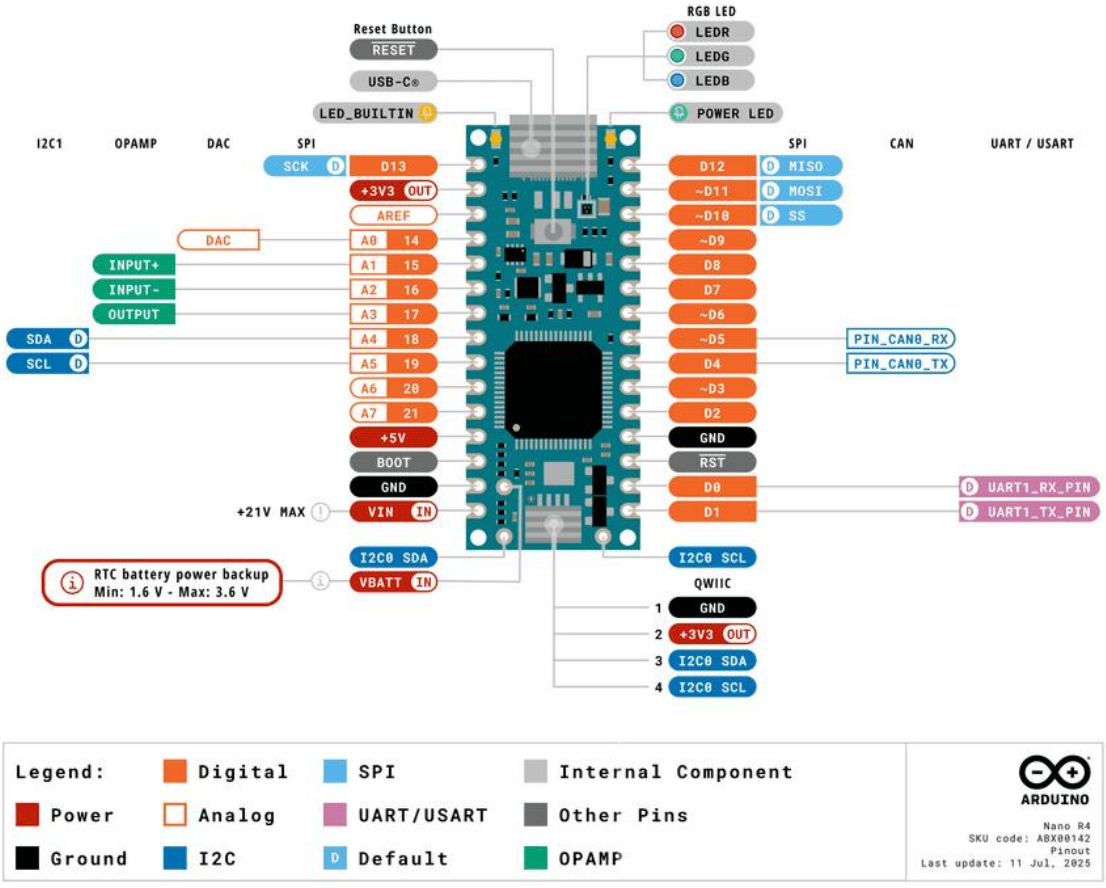
Sensor Module Layout on Base Shield



Introduction to the Development Board



Let's take a closer look at the Arduino Nano R4 Development Board! Many beginners may not know what the numbers and labels on the board represent. Next, we'll go through each label one by one, so you'll understand the purpose of every pin and interface and be able to use the board more effectively.

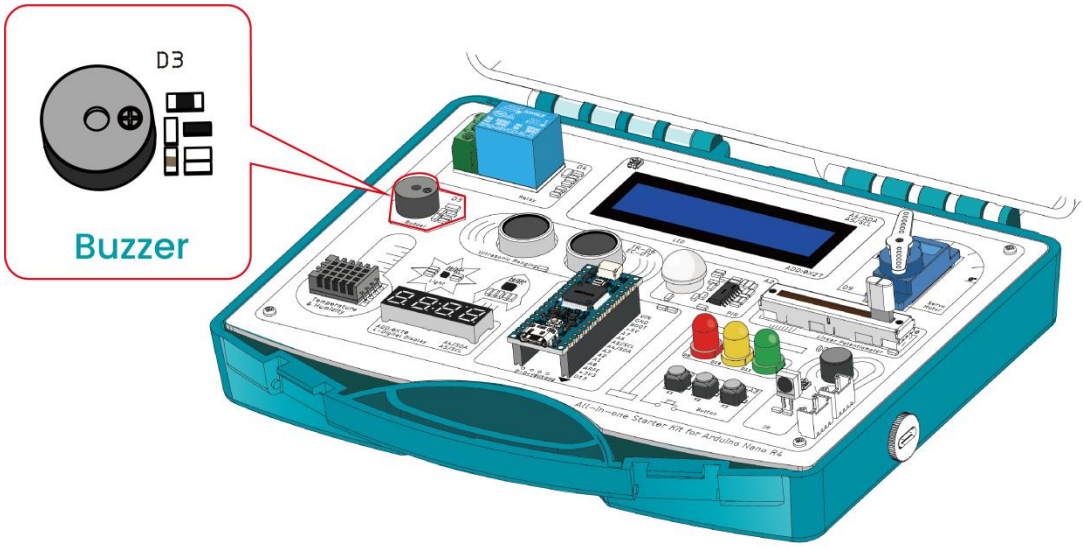


①Install the Main Board

First, insert the Arduino Nano R4 main board into the central slot of the development board. Make sure the Type-C port is facing down. The development board clearly marks the positions of all Arduino Nano R4 pins, making it easy to connect.

②Understanding Pin Labels and Module Connections

On the development board, you'll notice that each sensor module is labeled with its corresponding pin number. For example, the buzzer module is marked"D3"indicating that it connects to the D3 pin on the Arduino Nano R4 main board.



When we look at the main board, we can see a pin labeled"~D3."The"~"symbol is very important-it indicates that this pin supports PWM (Pulse Width Modulation) output.

This means we can do more than just turn the buzzer on or off; we can also program different PWM frequencies to make the buzzer produce varying tones.

"D4"indicates a digital pin. It is typically used to: receive high or low signals from input modules (like button presses or releases); control the on/off state of output modules (such as turning an LED on or off); or read signals from sensors that have only two states (on/off).

"~D3": The tilde "~" indicates PWM support. Pins like this can output both digital high/low signals like regular digital pins and analog-like PWM signals, which can control brightness, speed, or sound pitch. They can also read digital signals from sensors.

"A2" indicates an analog input pin. It is used to read continuous analog signals from sensors, such as varying voltages. This is common for sensors with multiple states, like a sliding potentiometer. Of course, it can also be used for simple two-state signals, though a digital pin is usually preferred in that case.

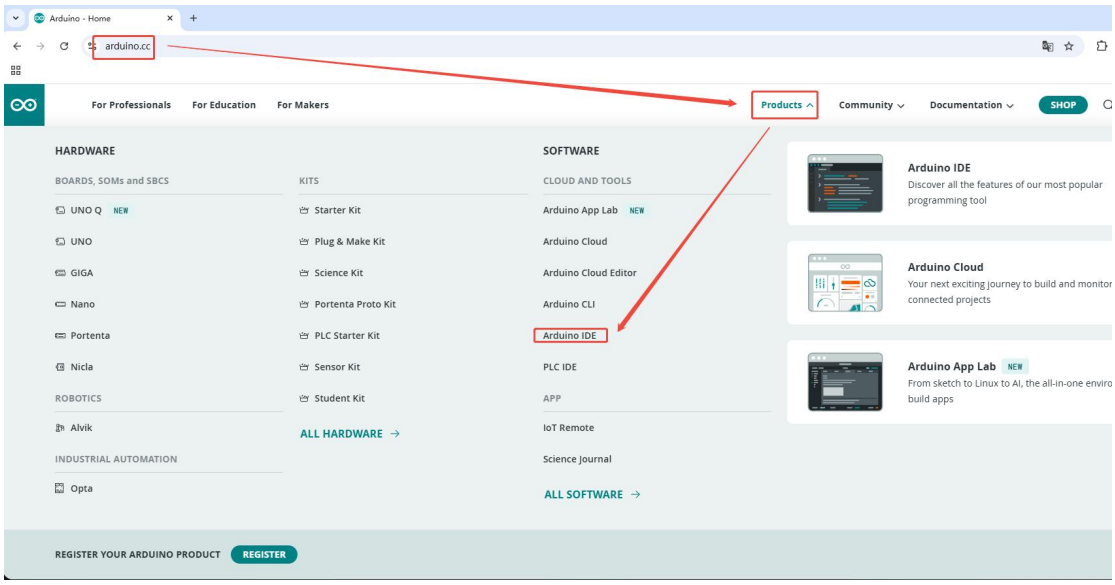
Installing Arduino IDE

Download Arduino in Windows system

Step 1:

Login to Arduino official website, download Arduino

Arduino official website: https://www.arduino.cc/



Step 2:

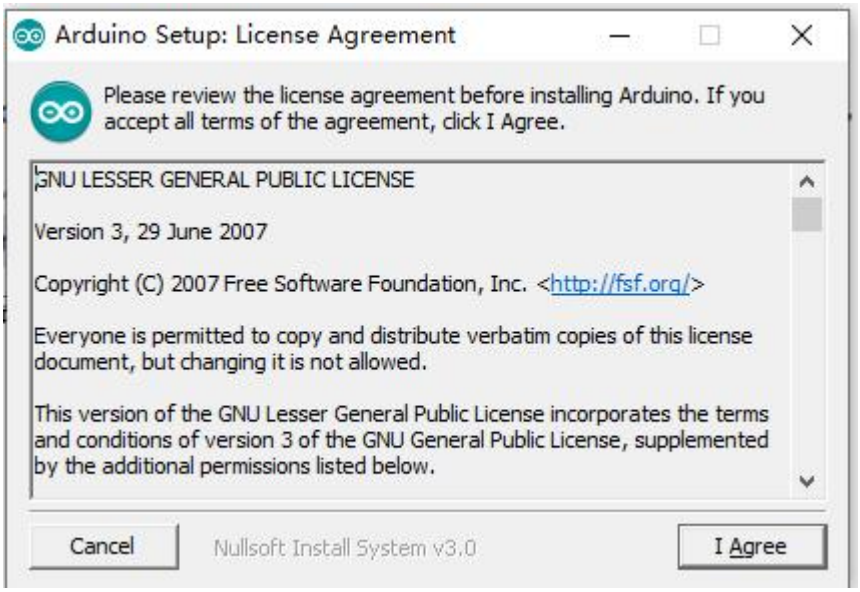
Select your computer's corresponding system to download, such as Window system.

Note: This tutorial uses version 2.3.6. You can try other versions, but if you encounter any issues during flashing, please switch back to version 2.3.6 and try again.

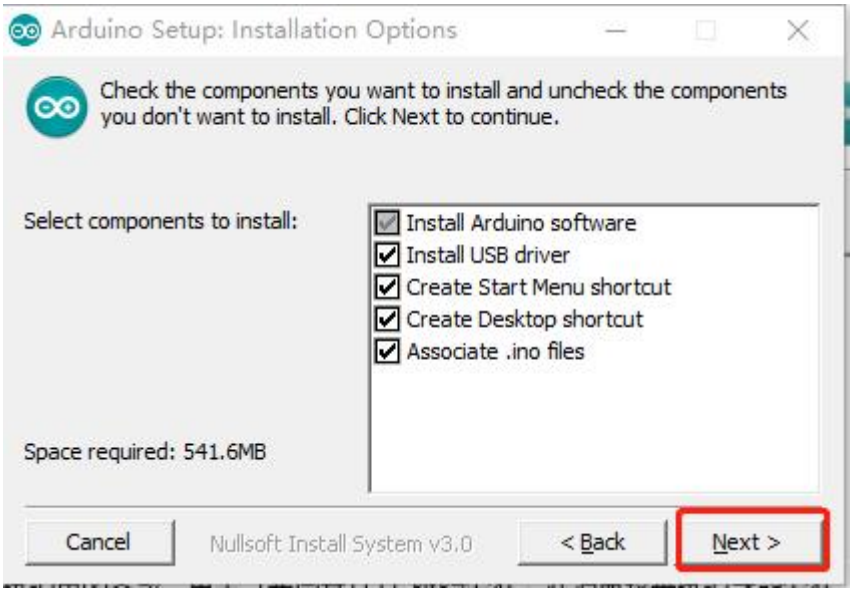
Step 3:

1.When installing Arduino, please locate the executable file with the .exe extension within the folder where you previously saved, which is the Arduino installation package.

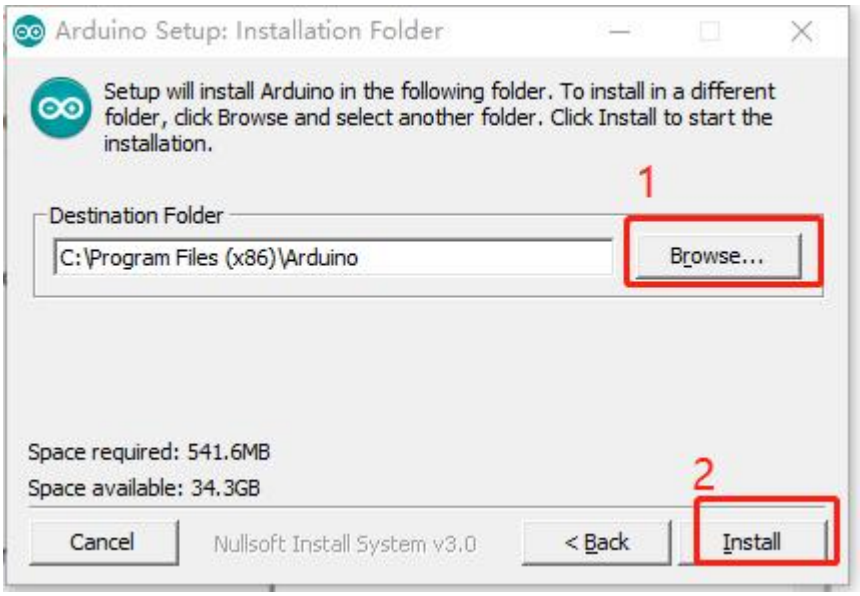
2. After double-clicking the installation package, this page will appear. Click on "I Agree".



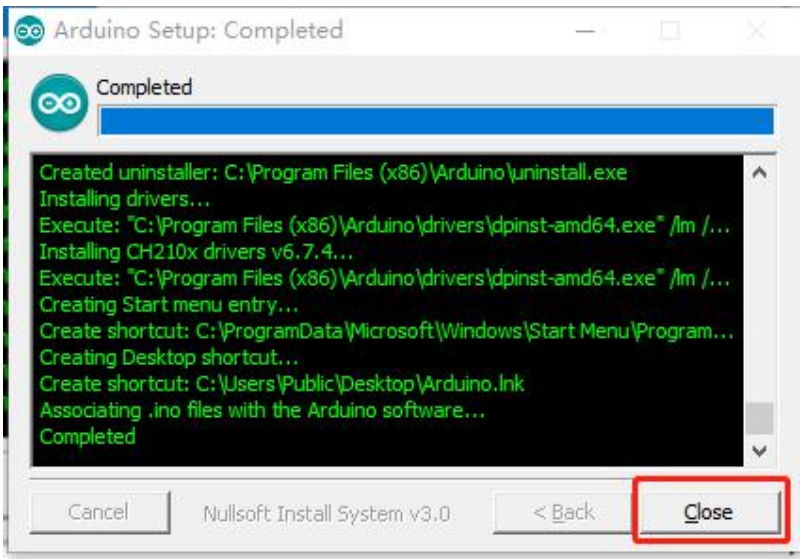
3.Check all options by default and click next.



4.Click on 'Browse' to select the installation location, it is recommended to install it on any drive other than the C: drive. Then click 'Install'.



5.Installation Completed



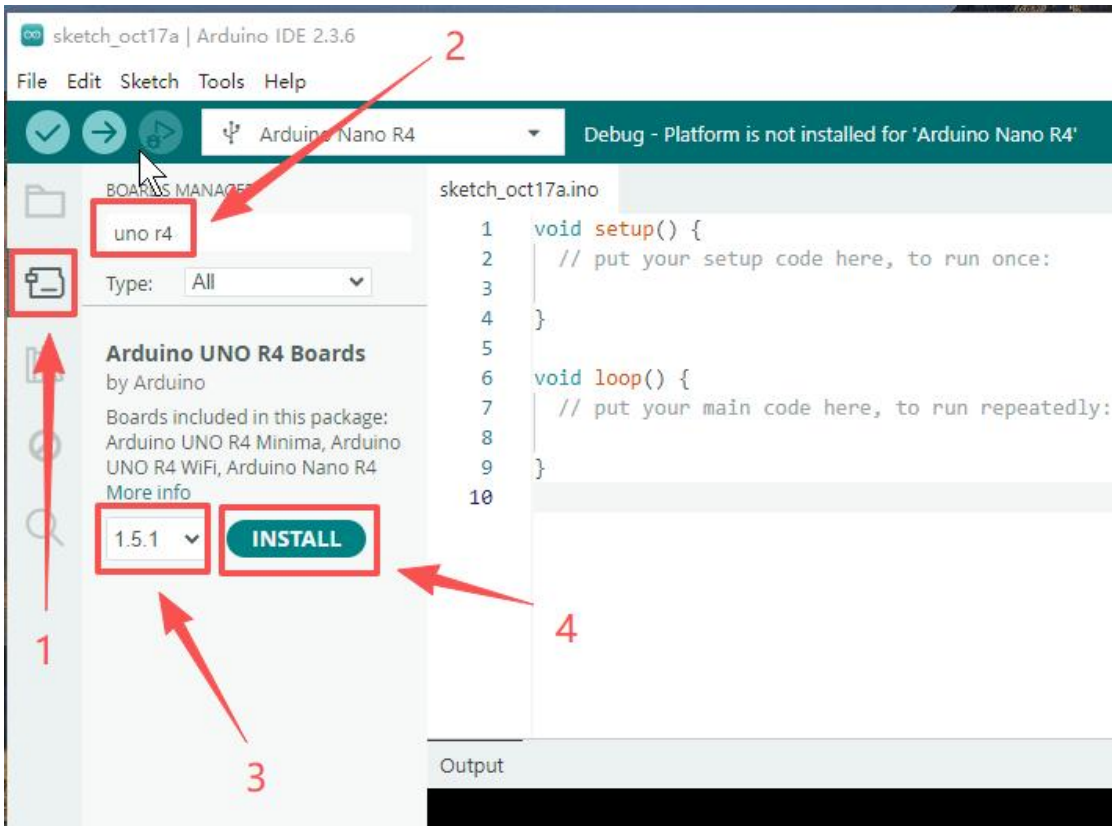
Installing the Nano R4 Development Board

①Click the development board icon on the left

②Type"uno r4"in the search bar

③Select version 1.5.1. You can try other versions, but if uploading the code fails, switch back to 1.5.1. This tutorial's code was developed using version 1.5.1.

④Click"INSTALL"to install



During the installation, a window will pop up-just click"Yes".

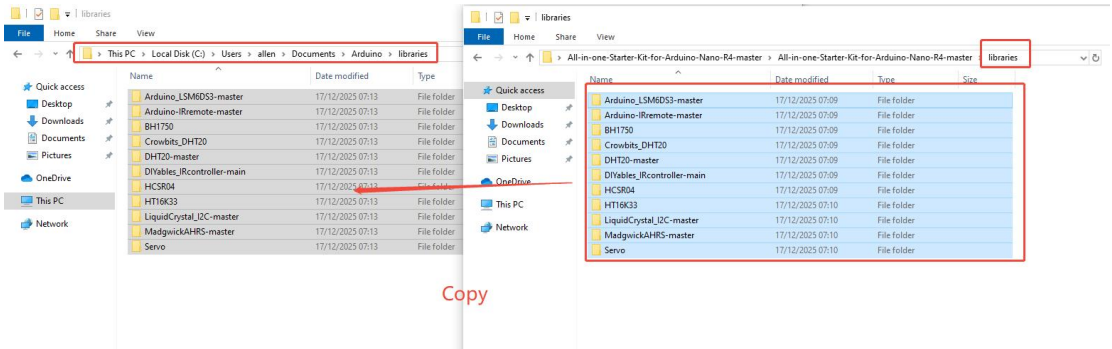
Importing Library Files (Important)

Before getting started, you need to download the required library files included with the kit and place them into the corresponding "/Arduino/libraries" directory on your computer.

Download link:

https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-Arduino-Nano-R4/tree/master/libraries

Copy all the library files from the downloaded libraries folder into the following directory on your computer: "C:\Users\Users name\Documents\Arduino\libraries"

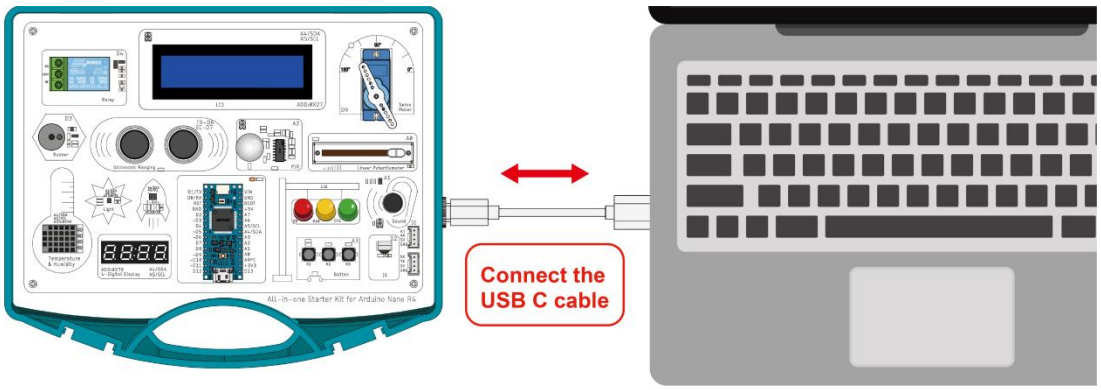


Note: If there is no "libraries" folder in this directory, please create one manually.

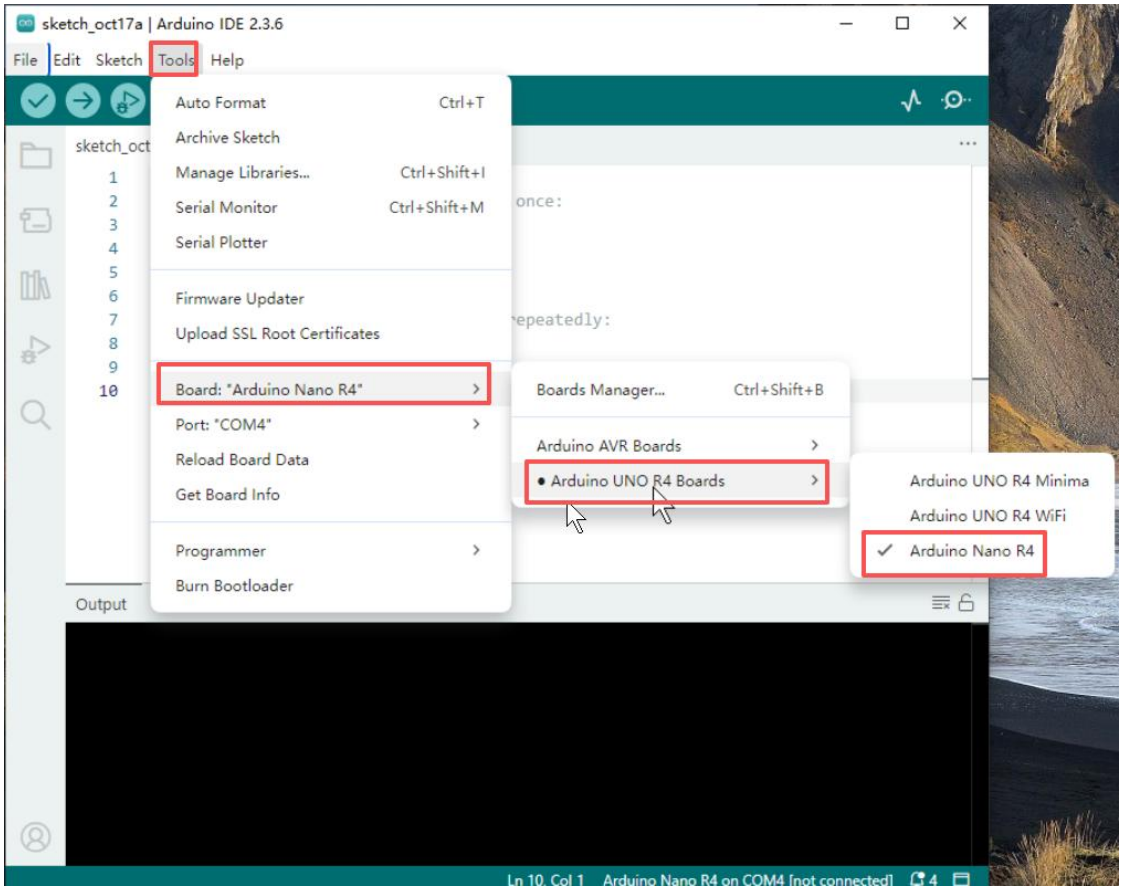
Upload Steps (Important)

In all following lessons, the upload steps are the same. Please read carefully, and if you forget, you can return here and follow the steps:

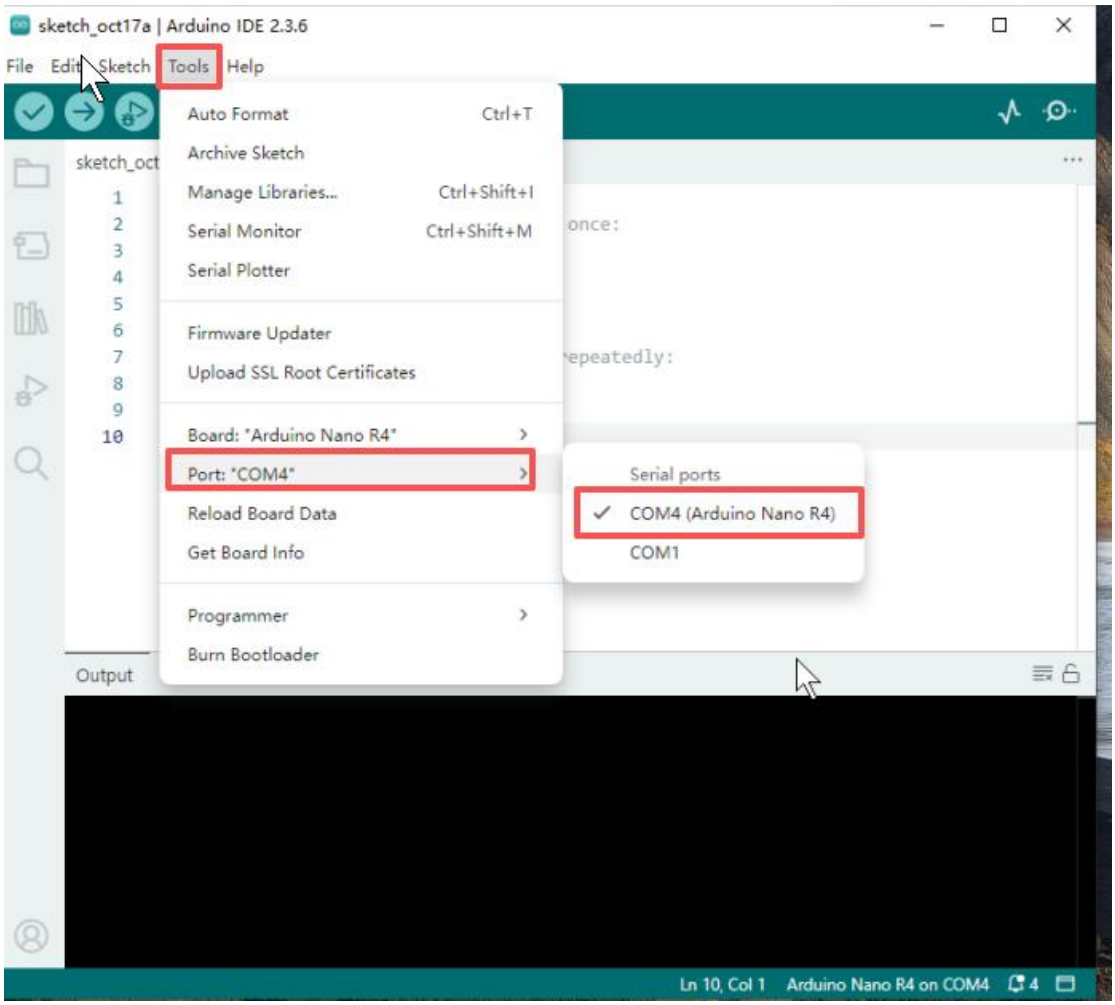
1.Connect the Type-C upload port to your computer.



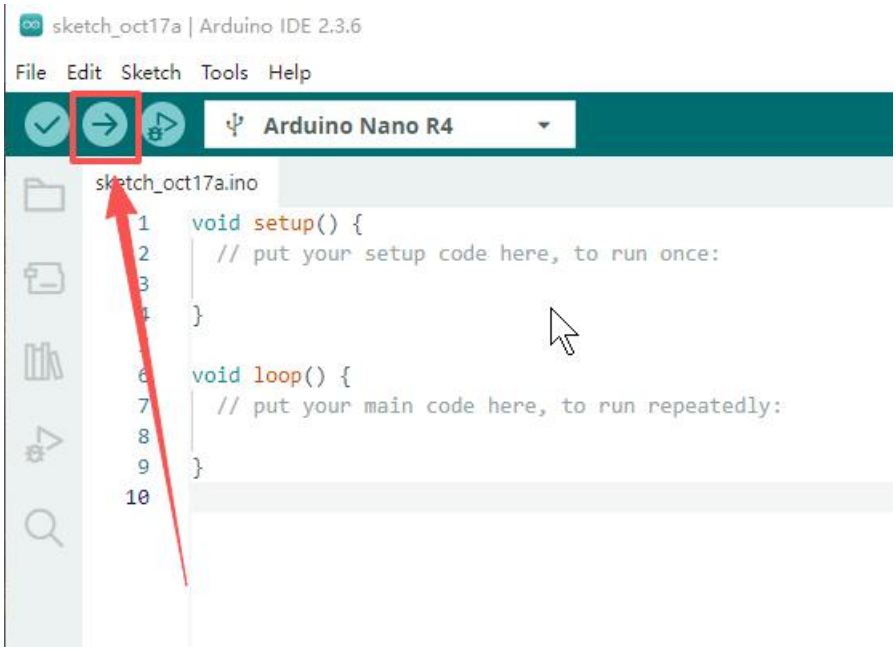
2.After writing the code, select the board you want to upload to:"Tools"->"Board"->"Arduino UNO R4 Boards"->"Arduino Nano R4"



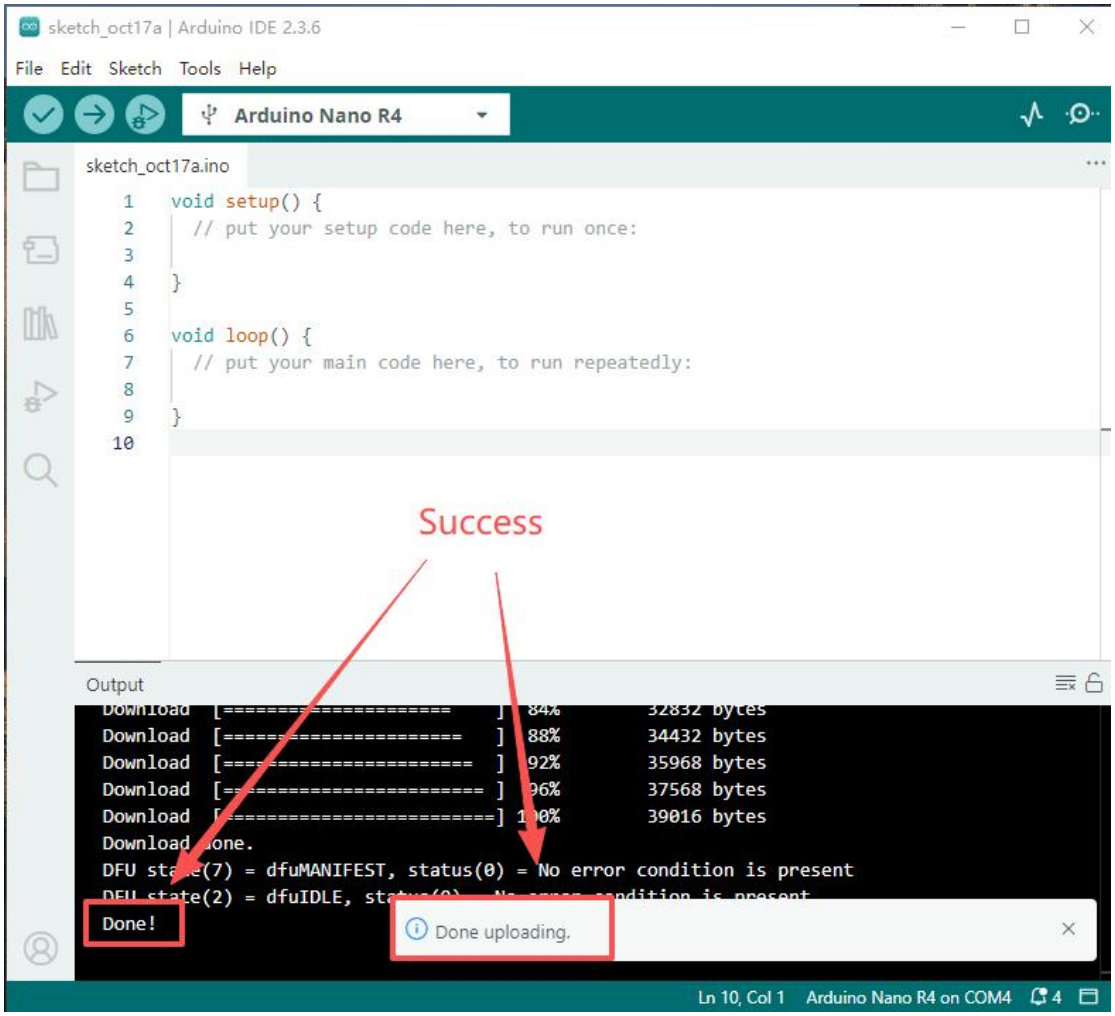
3.Select the serial port for uploading (note: the tutorial shows COM4, but yours may be different-make sure it corresponds to your Arduino Nano R4)."Tools"->"Port"->"COM4(Arduino Nano R4)"



4.Click the"Upload"button to upload the code to the development board.



5.Confirm that the upload was successful.


sketch_oct17a | Arduino IDE 2.3.6
File Edit Sketch Tools Help
Arduino Nano R4
sketch_oct17a.ino
void setup() {
// put your setup code here, to run once:
}
void loop() {
// put your main code here, to run repeatedly:
}
Success
Output
Download [ ] 88% 34432 bytes
Download [ ] 92% 35968 bytes
Download [ ] 96% 37568 bytes
Download done.
DFU state(7) = dfuMANIFEST, status(0) = No error condition is present
Done!
Done uploading.
Ln 10, Col 1 Arduino Nano R4 on COM4

Lesson01---LED

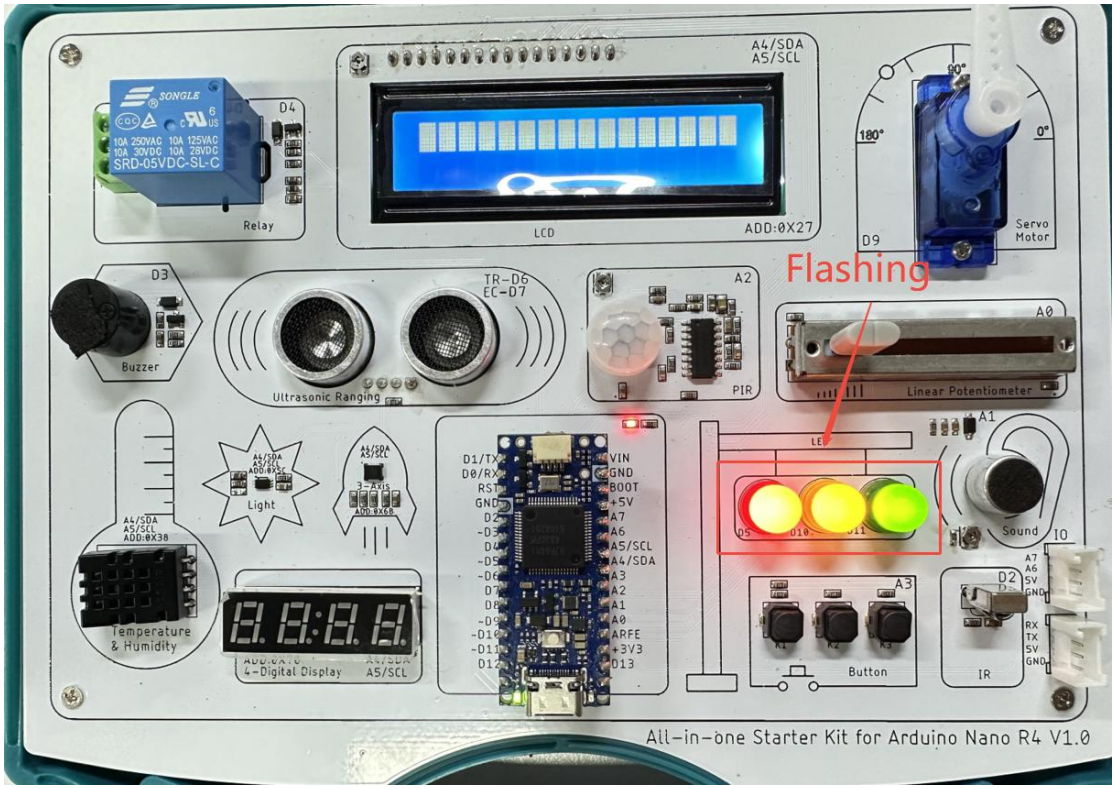
Introduction

In this lesson, we will systematically learn how to use the Arduino programming language to control the three LED indicators on the Arduino Nano R4 development board. By the end of this section, you will not only understand how to write Arduino programs but also master the basic operations of turning the onboard LEDs on and off.

Learning Goals

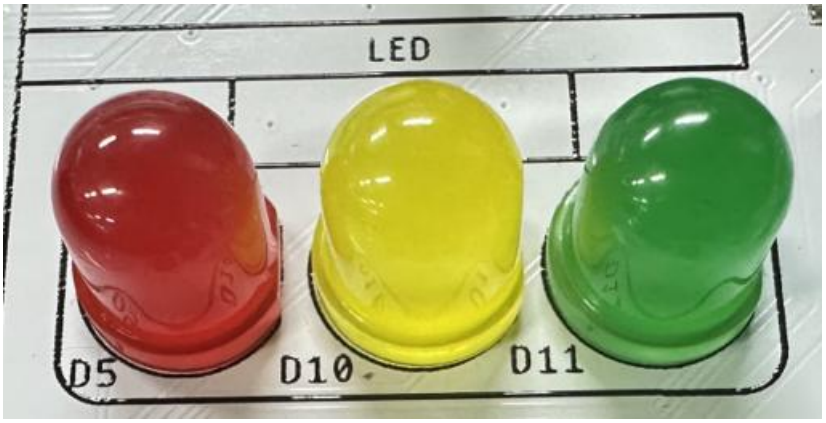
1.Understand the Arduino programming framework — the setup() and loop() functions
2.Learn to use the pinMode() function to configure pin modes
3.Learn to use the digitalWrite() function to turn on an LED
4.Learn to use the delay() function to control LED blinking

Preview of the Result



After uploading the code, as shown in the image above, the three LEDs will start blinking.

Hardware Used in This Lesson



On the Arduino Nano R4 development board, we have set up three LEDs:

Red LED: connected to D5

Yellow LED: connected to D10

Green LED: connected to D11

All three pins are marked with a "~" on the board, indicating that they support PWM (Pulse Width Modulation) output. However, in this lesson, we won't be using PWM yet. We will start with the most basic on/off control, learning how to turn the LEDs on and make them blink.

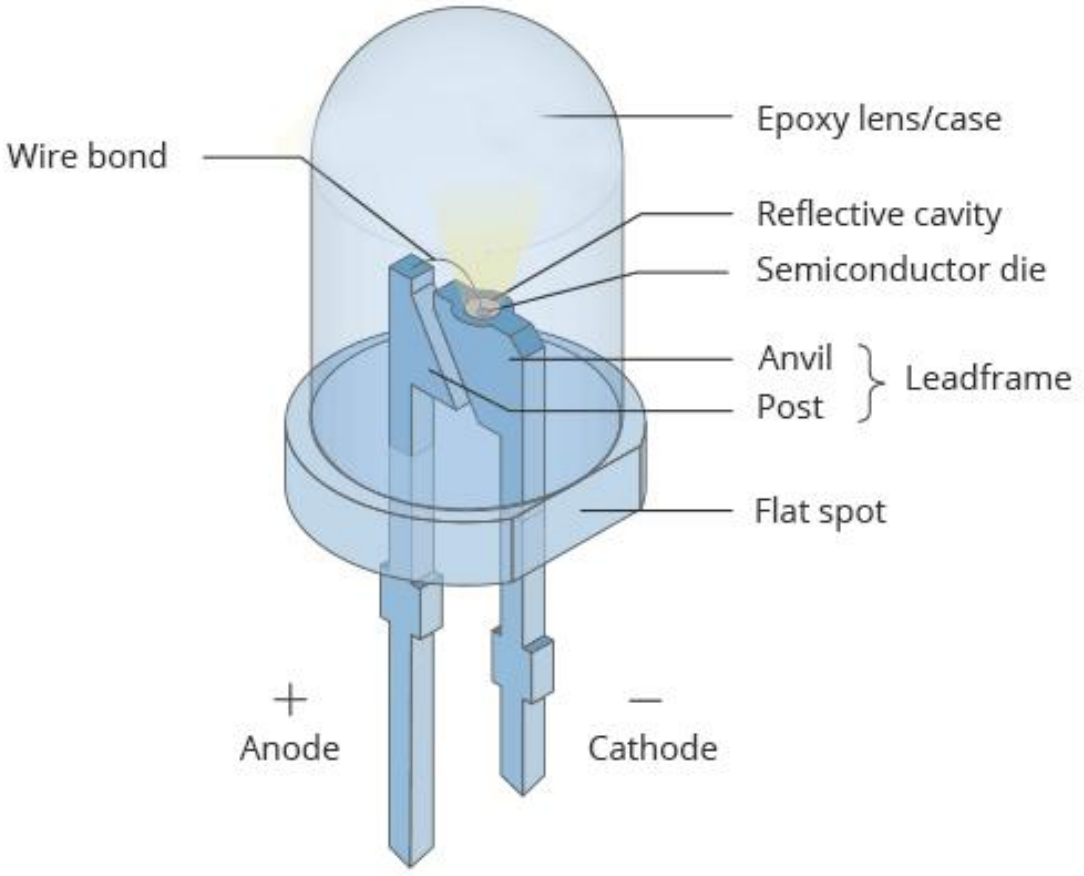
Working Principle of an LED

An LED is a solid-state semiconductor device that converts electrical energy directly into visible light. The LEDs used on Arduino development boards are typically pre-packaged small bulbs, making them easy to insert into a breadboard or connect to pins.

A packaged LED still has two leads:

- **Long lead:** positive (connects to an Arduino Nano R4 output pin, such as D5)
- **Short lead:** negative (connects to ground or GND)

By correctly connecting the positive and negative leads and controlling the digital pins on the Arduino, we can make the LED light up or blink.



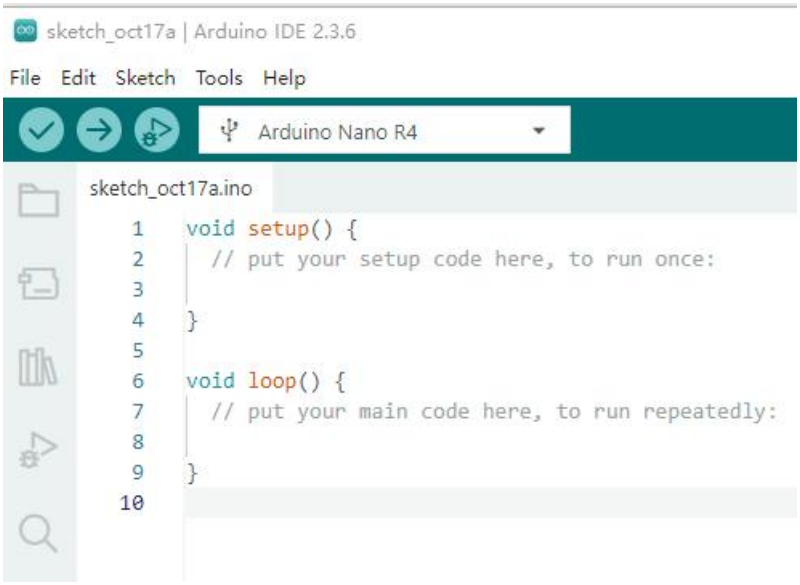
On the Arduino Nano R4 development board, the three LEDs are pre-packaged and connected to the board's pins: D5, D10, and D11.

LED Blinking

Before going through the code, you can download it. Here is the link to download the complete code:
https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-Arduino-Nano-R4/tree/master/lesson_code/1_LED
Open the "1_LED.ino" file located in the "1_LED folder" using the Arduino IDE.

Key Code Explanation

In Arduino programming, almost every program has two core functions:



setup() Function

This part of the program runs only once at the start. It's typically used for initialization, such as setting LED pins as outputs, initializing serial communication, or configuring sensor parameters. For example, if we want to turn on an LED, we first need to tell the Arduino in setup() that this pin will be used for output. Think of setup() as the "preparation stage"—it runs just once.

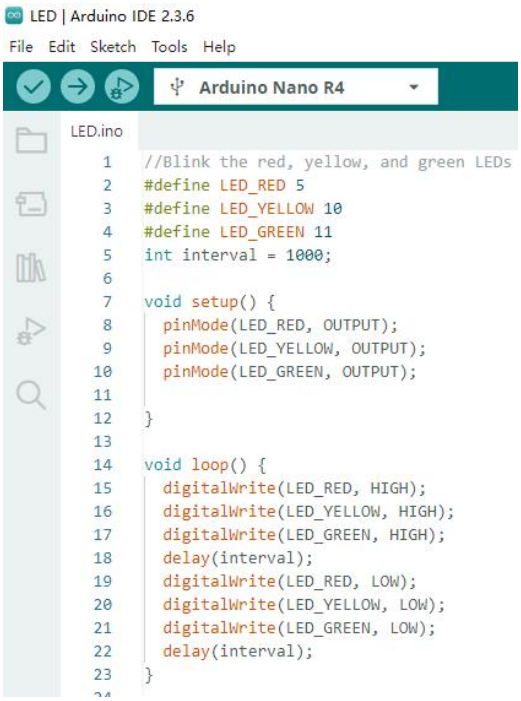
loop() Function

This is the core loop of an Arduino program, which runs continuously. All the main logic—such as turning on LEDs, reading sensors, or controlling buzzers—is placed here. By executing repeatedly, the Arduino can continuously control external devices, enabling functions like blinking, monitoring, or responding to actions. The loop() function is the "working stage," running over and over.

By understanding these two functions, we can control various hardware on the Arduino and make them operate according to our program's logic.

Note: In Arduino code, anything following // is a comment and will not be executed by the program. Comments are used to explain the code or leave notes for yourself and others, making the program easier to understand.

Now, let's move on to the example: the LED Blinking Experiment.



First, we need to define the three LEDs used in this experiment:

```
#define LED_RED 5          // Define the red LED pin as digital pin 5
#define LED_YELLOW 10      // Define the yellow LED pin as digital pin 10
#define LED_GREEN 11       // Define the green LED pin as digital pin 11
```

#define is a C/C++ preprocessor directive. It tells the compiler to replace every occurrence of a macro name with its corresponding value before the code is compiled. In other words, it performs text substitution, not variable creation.

In the code above, "LED_RED" is replaced with 5, "LED_YELLOW" with 10, and "LED_GREEN" with 11. These numbers correspond to the pins connected to the red, yellow, and green LEDs on the development board.By using macro names, we don't need to write pin numbers directly in the program—just use the macro to control each LED.This approach has several advantages:

1.Better readability： When you see LED_RED, you instantly know it refers to the red LED, instead of a random number like 5.

2.Easier maintenance： If the pin changes in the future, you only need to update the macro definition instead of modifying the code everywhere.

3.Fewer errors： Using consistent macro names avoids mistakes such as typing the wrong pin number and causing incorrect LED behavior.

Next, we define a time variable called "interval".

```
int interval = 1000;     // Set the blinking interval to 1000 milliseconds (1 second)
```

Here, int is used to define an integer variable, which stores whole numbers. The variable interval is used to control the blinking interval of the LED, measured in milliseconds. A value of 1000 means 1000 milliseconds.

Why do we store the blinking interval in a variable?For the same reasons mentioned earlier—better readability and easier maintenance. By adjusting the value of interval, you can easily change how fast the LED blinks without modifying multiple lines of code.

Initialization Function

```
void setup() {
  pinMode(LED_RED, OUTPUT);       // Set the red LED pin as an output
  pinMode(LED_YELLOW, OUTPUT);   // Set the yellow LED pin as an output
  pinMode(LED_GREEN, OUTPUT);     // Set the green LED pin as an output
}
```

We've already discussed that the **setup() function** runs only once at the beginning of the program and is mainly used for initialization. Here, we use pinMode() to set the pins for the red, yellow, and green LEDs to output mode (OUTPUT). This tells the board that these pins will send electrical signals to control whether the LEDs are on or off.With the pins properly configured, we can later use digitalWrite() inside the loop() function to turn each LED on or off.

The pinMode() function defines whether a pin is used as an input or output. If you try to control an LED with digitalWrite() without calling pinMode() first, the LED will not work correctly.

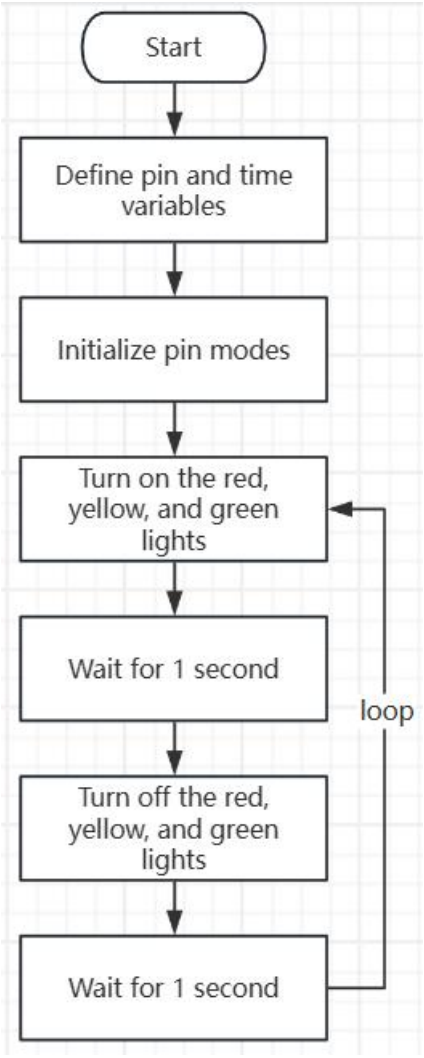
loop() function

```
void loop() {
  digitalWrite(LED_RED, HIGH);       // Turn on the red LED
  digitalWrite(LED_YELLOW, HIGH);   // Turn on the yellow LED
  digitalWrite(LED_GREEN, HIGH);     // Turn on the green LED
  delay(interval);                   // Wait for the defined interval (1 second)

  digitalWrite(LED_RED, LOW);        // Turn off the red LED
  digitalWrite(LED_YELLOW, LOW);     // Turn off the yellow LED
  digitalWrite(LED_GREEN, LOW);      // Turn off the green LED
  delay(interval);                   // Wait for the defined interval (1 second)
}
```

The loop() function is the core loop of an Arduino program. After the setup() function finishes, the program repeatedly executes the code inside loop().We use digitalWrite(LED_RED, HIGH) to set a pin to a high voltage level and digitalWrite(LED_RED, LOW) to set it to a low voltage level, which turns the LED on and off.

The delay() function is used to pause the program. When the program reaches delay(1000), for example, it pauses for 1000 milliseconds (1 second) before continuing.

Overall Code Logic Flowchart



Program Upload Steps

For detailed upload instructions, please refer to the "Upload Steps" on page 8.

Key Takeaways:

void setup()	Initialization function: Runs only once, typically used to configure pin modes and other setup tasks.
void loop()	Loop function: Continuously repeats the program's core functionality.
pinMode()	Configure pin modes: Set pins as input or output.
digitalWrite()	Write high/low values to pins: Control hardware on/off states.
delay()	Delay function: Pauses the program for a specified time in milliseconds.

Lesson02---Sound Sensor

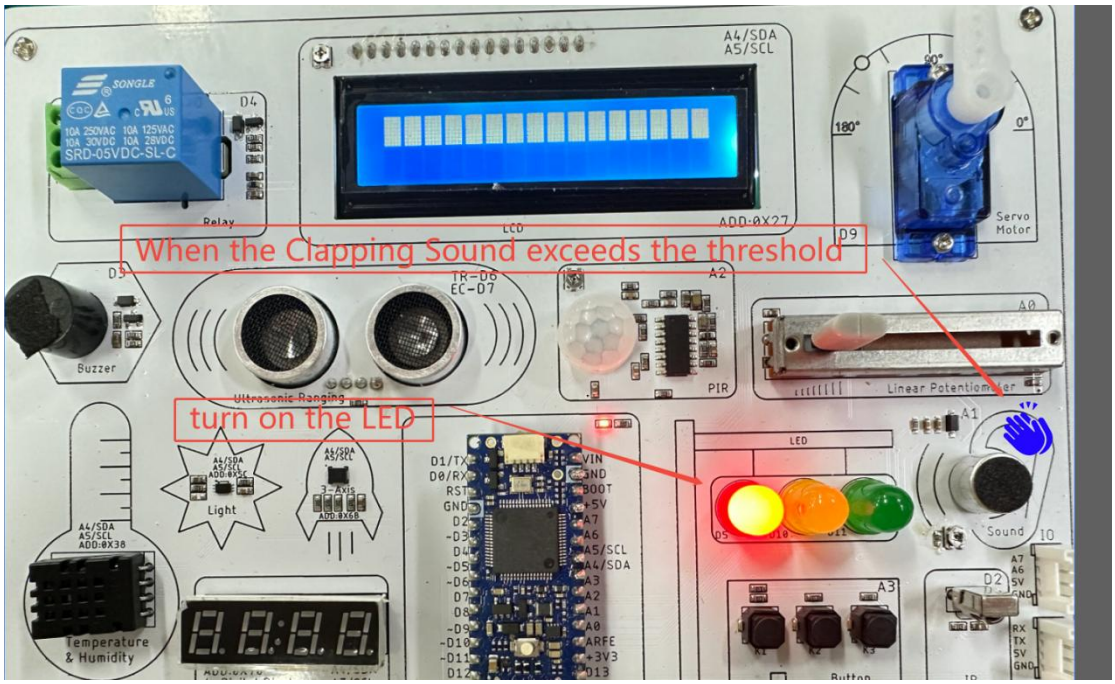
Introduction

In this lesson, we will learn how to read analog values from a sound sensor and use threshold detection to create a "sound-activated hallway light." When a sound is detected in the environment, the LED will turn on; when it's quiet, the LED will automatically turn off. Through this experiment, you will become familiar with reading analog signals and learn how to perform simple logic control based on sensor data.

Learning Goals

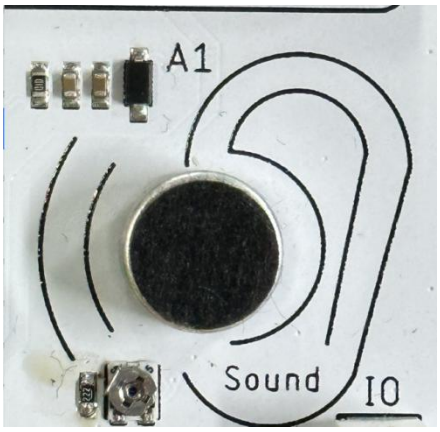
1.Characteristics of analog output from a sound sensor
2.Learn when to use analog pins versus digital pins
3.Master the"analogRead()"function to read analog values from the sensor
4.Learn to use the serial monitor for debugging
5.Learn to use if statements for logical decisions
6.Complete the analog sound-activated hallway light experiment

Preview of the Result



After uploading the code, when the sound sensor detects an analog value greater than the threshold set in the code, the red LED will light up for 3 seconds and then turn off. Conversely, if the value is below the threshold, the LED will not turn on. As shown in the image, when powered on, clapping will trigger the LED to light up for 3 seconds.

Hardware Used in This Lesson



On the Arduino Nano R4 development board, the sound sensor is located on the right side and is connected to the main board via the A1 pin. A1 is an analog input pin, used to read continuously varying analog voltage values, unlike digital pins (D pins) that can only read high or low levels.

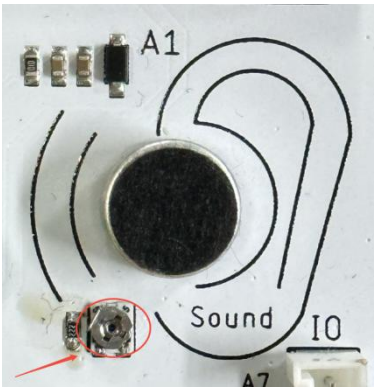
When to choose an analog pin versus a digital pin:

•When there are only two possible states to detect, such as with a button module, use a digital pin. Digital pins only need to detect high or low voltage levels.

•When you need to obtain continuously changing data, such as sound or light intensity, use an analog pin. Analog pins can read continuously varying voltages, providing more precise numerical information.

At the lower-left corner of the sound sensor,there is a sensitivity adjustment knob. Turning it clockwise increases the sensor's sensitivity, so even small sounds will produce larger readings. Turning it counterclockwise decreases the sensitivity, requiring louder sounds to trigger an output.

Note: Do not force the knob beyond its limit when adjusting, as this may damage the internal mechanism.



Working Principle of an Sound Module

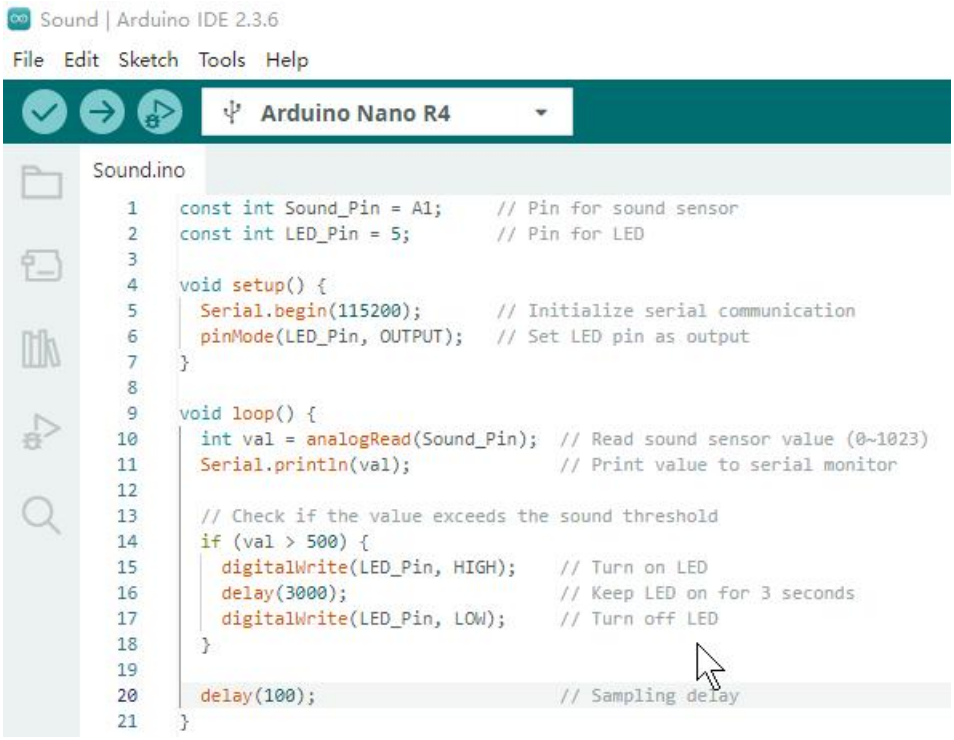
A sound sensor is a small electronic component used to "listen" to sounds. At its core is a tiny microphone that converts surrounding sounds into weak electrical signals. These signals are then amplified and processed by the circuit, finally outputting an analog voltage. The Arduino reads this analog value to determine the sound level in the environment—the louder the sound, the higher the output voltage; the quieter the sound, the lower the output voltage.

Simulated Sound-Activated Hallway Light

You can download it. Here is the link to download the complete code:
https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-Arduino-Nano-R4/tree/master/lesson_code/2_Sound
Open the " 2_Sound.ino" file located in the" 2_Sound" folder using the Arduino IDE.

Key Code Explanation

Now, let's go through the example: the analog sound-activated hallway light.



First, we start by defining the pins that will be used

```
const int Sound_Pin = A1;       // Pin for sound sensor
const int LED_Pin = 5;          // Pin for LED
```

In Arduino (C/C++), **const** is used to make a variable read-only, meaning its value is fixed at compile time and cannot be changed during program execution. This improves code safety by preventing accidental changes to important values, such as pin numbers. Removing const won't cause your code to fail.

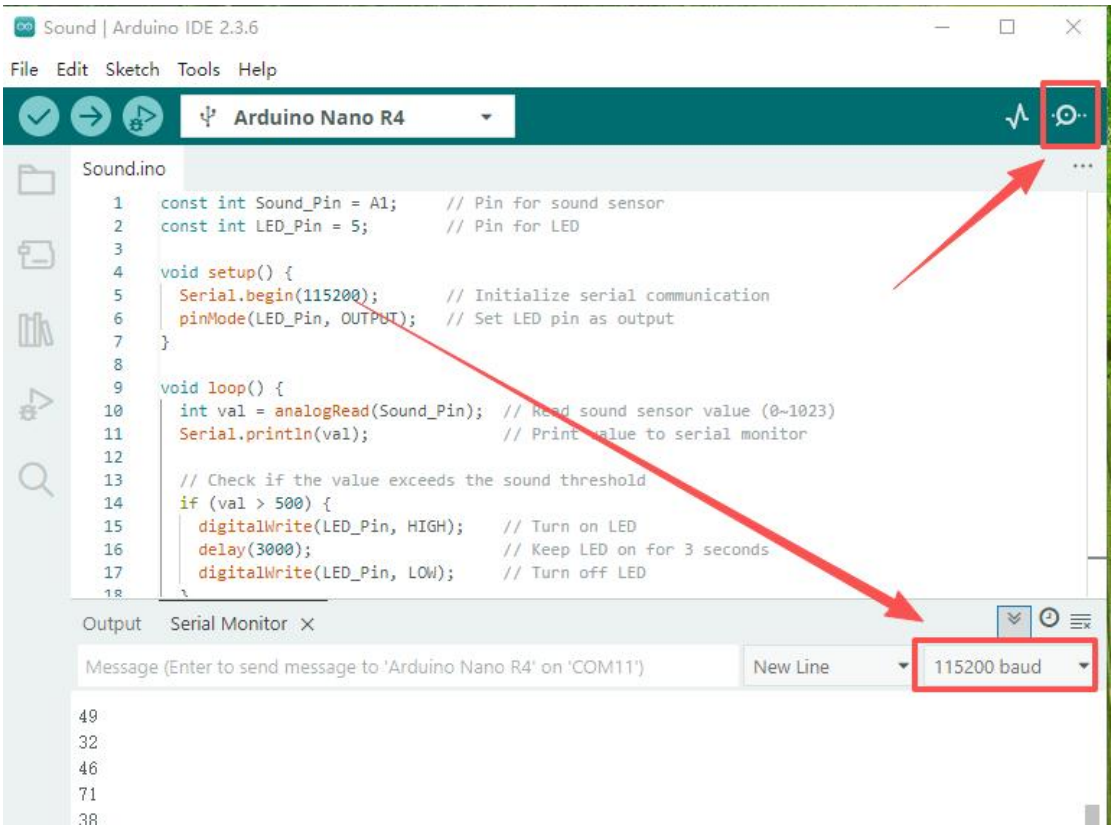
int is used to define an integer variable. You might wonder why we can use int to define "A1", since it doesn't look like a number. The reason is that in the Arduino compiler, "A1" is actually an integer constant. The name A0, A1, etc., is used simply to make analog input pins easier for developers to identify and remember.

Initialization Function

```
void setup() {
  Serial.begin(115200);         // Initialize serial communication
  pinMode(LED_Pin, OUTPUT);     // Set LED pin as output
}
```

In the initialization function, we introduce a new concept: Serial.begin. This initializes the Arduino's serial communication and sets the baud rate to 115200. Note that the baud rate set here must match the rate selected in the serial monitor; otherwise, you may see garbled output.

How to Open the Serial Monitor:



Click the Serial Monitor icon in the upper-right corner. Once it opens, select the same baud rate in the bottom-right corner as set in your code.

Read the sensor's analog value and make a judgment

```
void loop() {
  int val = analogRead(Sound_Pin);   // Read sound sensor value (0~1023)
  Serial.println(val);                    // Print value to serial monitor
```

In the loop() function, our main logic is as follows: first, read the analog value from the sound sensor, then print it to the Serial Monitor. By observing these values, we can determine the sound intensity and set an appropriate threshold to decide at what sound level the LED should turn on, achieving the sound-activated light functionality.

As mentioned earlier, **int** is used to define integer variables

The **analogRead()** function is specifically used to read the voltage value from an analog input pin. When a sensor outputs more than two levels of variation—for example, sound intensity, light brightness, or temperature—you can connect it to an analog pin and use analogRead() to get continuously varying values. This allows the program to make finer judgments and control decisions.

Example: if Conditional Statement

```
if (soundValue > threshold) {
      digitalWrite(ledPin, HIGH);
}
```

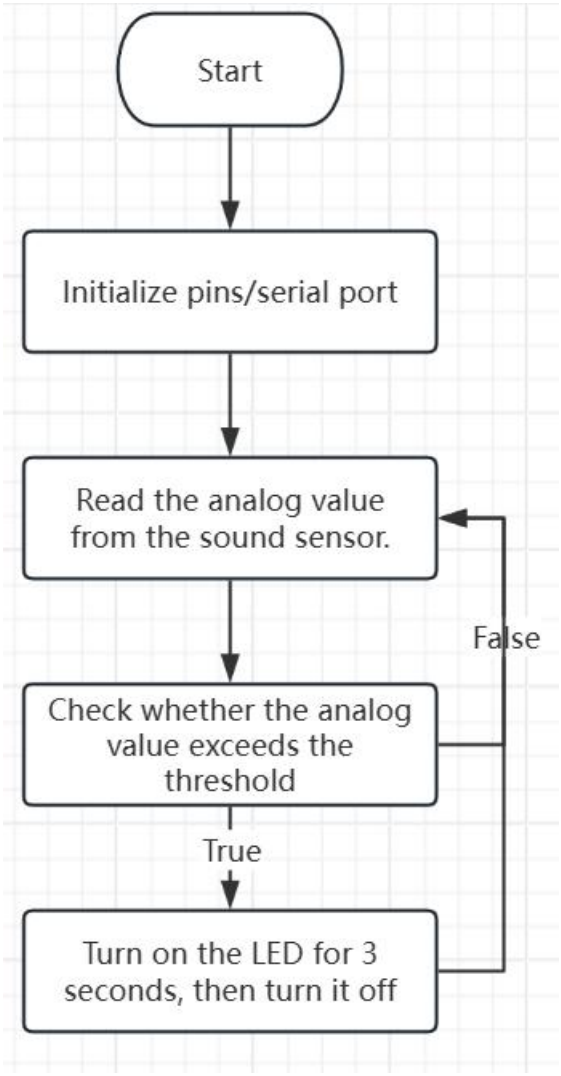
The code inside the curly braces will only execute if the condition in the parentheses (soundValue > threshold) is true, turning on the LED. If the condition is false, the code inside the braces will not run, and the LED will remain off.

The if conditional statement in this example

```
  // Check if the value exceeds the sound threshold
  if (val > 500) {
    digitalWrite(LED_Pin, HIGH);      // Turn on LED
    delay(3000);                      // Keep LED on for 3 seconds
    digitalWrite(LED_Pin, LOW);       // Turn off LED
  }
  delay(100);                         // Sampling delay
}
```

When we read the analog value from the sensor, we compare it to 500. If it exceeds this threshold, we execute digitalWrite(LED_Pin, HIGH); to turn on the LED. To simulate a hallway sound-activated light, we use the delay() function to keep the LED on for 3 seconds, and then execute digitalWrite(LED_Pin, LOW); to turn it off. The final delay(100) pauses for 0.1 seconds before reading the analog value again; this can be kept or removed as needed.

Overall Code Logic Flowchart



Program Upload Steps

For detailed upload instructions, please refer to the "Upload Steps" on page 8.

Key Takeaways:

Serial.begin	Initialize serial baud rate
analogRead	Reading analog values, typically used for sensors with multiple states
if	if conditional statements, used when the program needs to perform actions based on whether a condition is met

Lesson03---PIR Sensor

Introduction

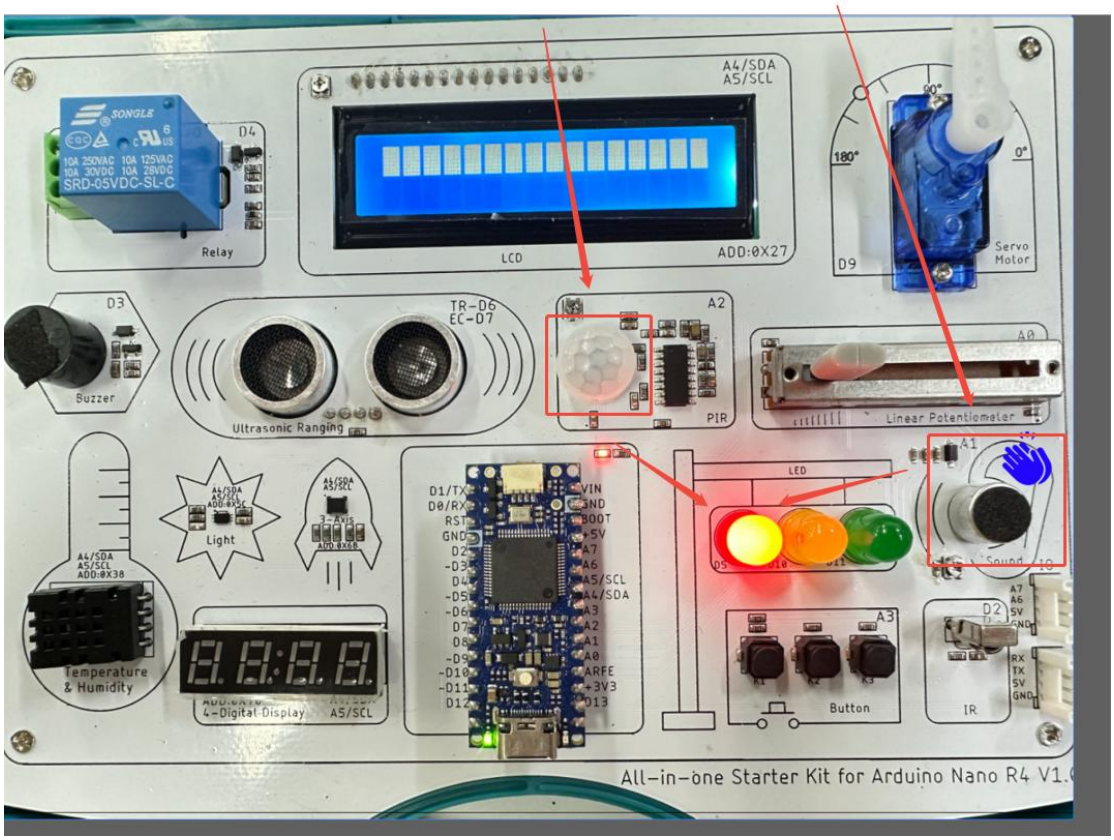
In this lesson, we'll learn one of the most commonly used conditional operators in Arduino programming—the logical OR operator ||. Its purpose is to allow multiple conditions to trigger the next part of the program as long as at least one of them is true.After learning this concept, you'll be able to write control logic that responds to multiple conditions and build an intelligent hallway light that makes decisions using multiple sensors at the same time.

Learning Goals

1.Understand how the PIR sensor works
2.Master the logical OR operator "||"
3.Learn how to read voltage levels using an analog pin
4.Complete the multi-sensor intelligent hallway light simulation example

Preview of the Result

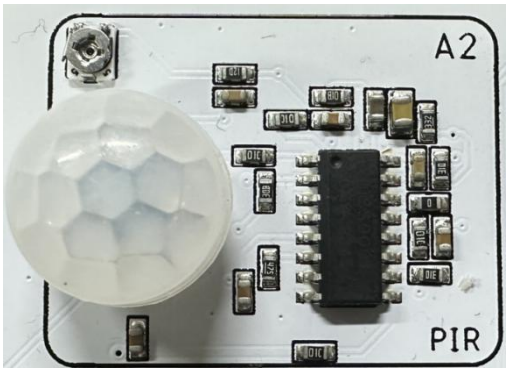
If a person is detected or the sound sensor reads a value above the preset threshold, the LED will turn on.

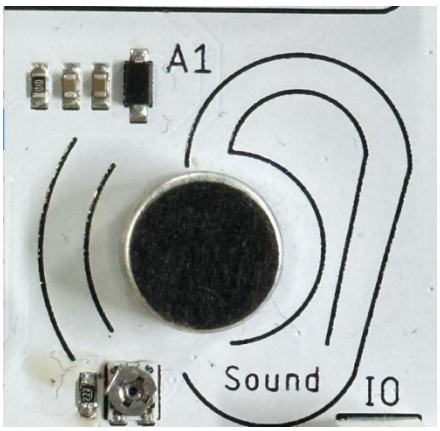


After uploading the code, if the sound sensor detects an analog value greater than the threshold

set in the program or the PIR sensor detects someone, the red LED will turn on for 3 seconds and then turn off.In this simulated smart hallway light project, if a person makes only a very small sound, the light might not turn on in time. Therefore, we add a PIR sensor to detect human presence. As long as either of the two conditions is met, the LED will turn on to provide illumination.

Hardware Used in This Lesson



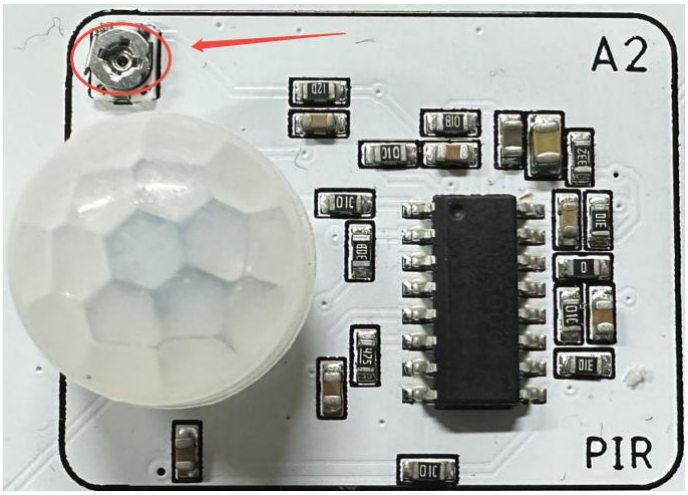


The PIR sensor is located in the middle section of the development board and is connected to the A2 analog input pin. It's important to note that analog pins can read both digital levels (HIGH/LOW) and analog values, while digital pins can only read HIGH or LOW and cannot read analog values.This is because analog pins include an analog-to-digital converter (ADC), which

allows analogRead() to read analog values ranging from 0 to 1023.

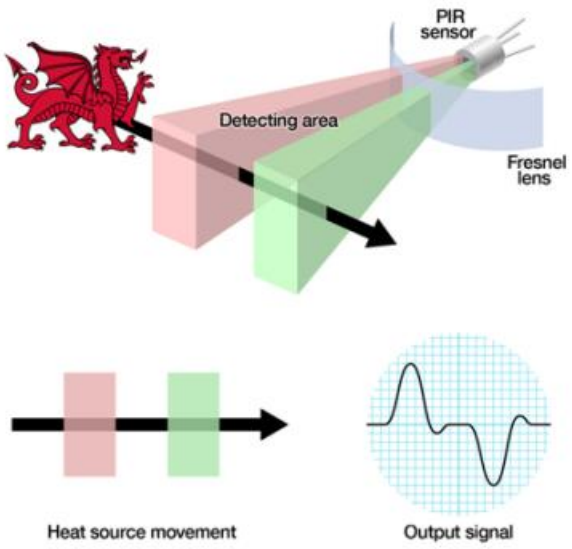
At the upper-left corner of the PIR sensor, there is a sensitivity adjustment knob. Turning it clockwise increases the sensor's sensitivity, while turning it counterclockwise decreases sensitivity. Avoid rotating it too far counterclockwise, or the sensor may fail to detect people even when they are present.

Note: Do not force the knob past its limit. Doing so may damage the internal adjustment mechanism.



Working Principle of an PIR Module

The PIR motion sensor can detect the change in the amount of infrared radiation reflected on it, which depends on the temperature and surface characteristics of the object in front of the sensor. When an object such as a person moves within the detection range of the PIR sensor, the temperature at that point in the sensor's field of view will rise from room temperature to body temperature, and this change will return to the sensor and be detected.

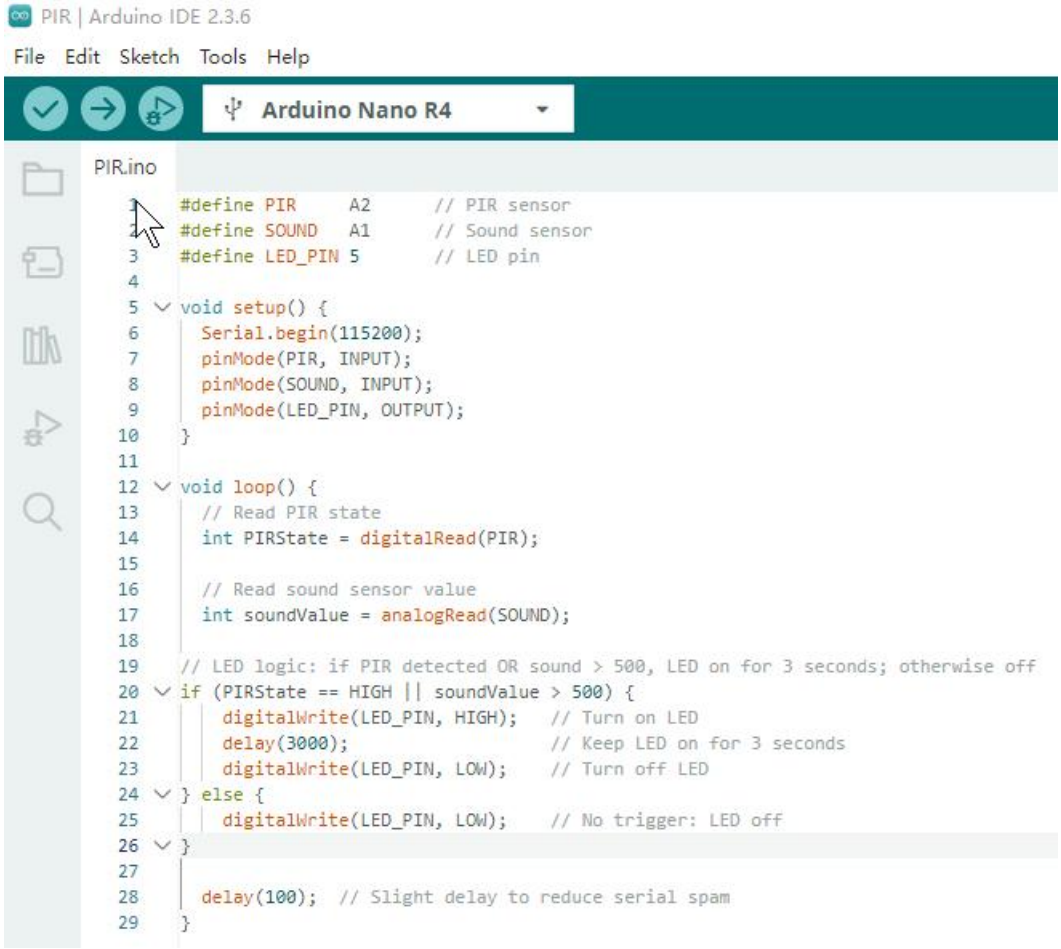


Smart Hallway Light

Before going through the code, you can download it. Here is the link to download the complete code:
https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-Arduino-Nano-R4/tree/master/lesson_code/3_PIR
Open the "3_PIR.ino" file inside the "3_PIR" folder using the Arduino IDE.

Key Code Explanation

Now let's walk through the example: Smart Sound-Activated Corridor Light



First, we again define the pins we need to use:

```
#define PIR       A2       // PIR sensor
#define SOUND     A1       // Sound sensor
#define LED_PIN 5          // LED pin
```

Here we define three pins:

• The PIR sensor is connected to the A2 analog pin, but since the PIR only has two states (someone detected or not), we can simply use digitalRead to read its HIGH or LOW level.

• The sound sensor is connected to the A1 analog pin, and as we learned in the previous lesson,

we use analogRead to obtain its analog value.

• The LED is connected to D5, and we turn it on by writing HIGH and off by writing LOW.

Initialization Function

```
void setup() {
  pinMode(PIR, INPUT);
  pinMode(SOUND, INPUT);
  pinMode(LED_PIN, OUTPUT);
}
```

In the initialization function, we configure the three pins with their appropriate modes. The sensors are set as input mode, while the actuator (LED) is set as output mode.

Inside the loop() function

```
void loop() {
  // Read PIR state
  int PIRState = digitalRead(PIR);

  // Read sound sensor value
  int soundValue = analogRead(SOUND);
```

Inside the loop function, when evaluating conditions, we need to first read the sensor's returned values to determine the current state. These key points were explained earlier, so we won't go over them again.

Key conditional-judgment code

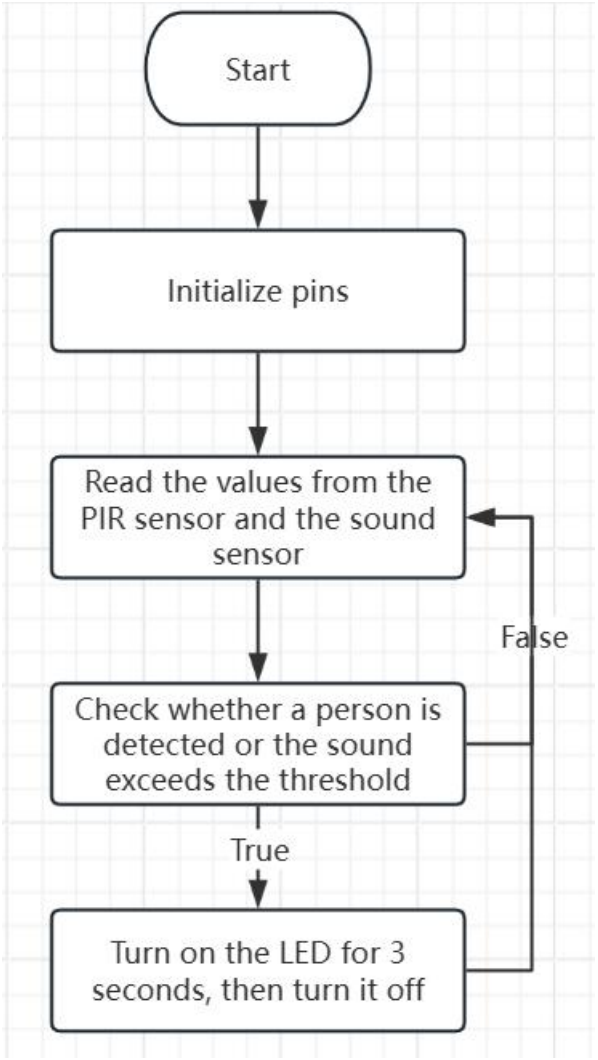
```
// LED logic: if PIR detected OR sound > 500, LED on for 3 seconds; otherwise off
if (PIRState == HIGH || soundValue > 500) {
    digitalWrite(LED_PIN, HIGH);    // Turn on LED
    delay(3000);                    // Keep LED on for 3 seconds
    digitalWrite(LED_PIN, LOW);     // Turn off LED
} else {
    digitalWrite(LED_PIN, LOW);     // No trigger: LED off
}

  delay(100);  // Slight delay to reduce serial spam
}
```

The “||” symbol is a logical OR operator, which you can understand as “or.” As long as either one of the two conditions is satisfied, the whole condition becomes true.

• "PIRState == HIGH || soundValue > 500": If the PIR sensor detects a HIGH level (someone is present), or the sound sensor detects a value greater than 500, the LED will turn on.

• The execution code here will not be explained in detail. Its main function is to set the LED to HIGH for 3 seconds and then set it back to LOW. If the condition is triggered, the LED will turn on for 3 seconds and then turn off.

Overall Code Logic Flowchart



Program Upload Steps

For detailed upload instructions, please refer to the "Upload Steps" on page 8.

Key Takeaways:

||	The logical OR operator considers the entire condition true as long as any one of the multiple conditions is satisfied.

Lesson04---Buzzer

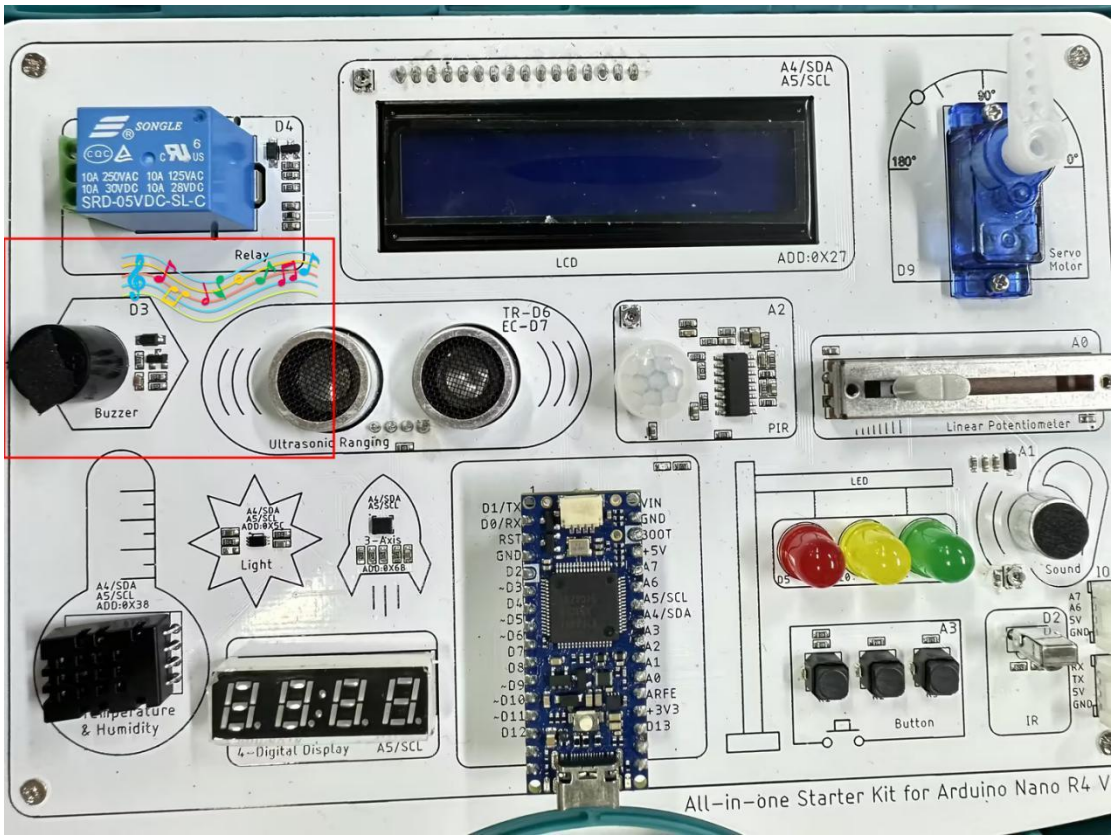
Introduction

In this lesson, we will learn how to control a buzzer to play different melodies. You will understand what a passive buzzer is and use for loops and functions to play the seven basic notes. By the end of this lesson, you will know how to define functions, simplify code with for loops, and change the pitch by adjusting different input signal frequencies.

Learning Goals

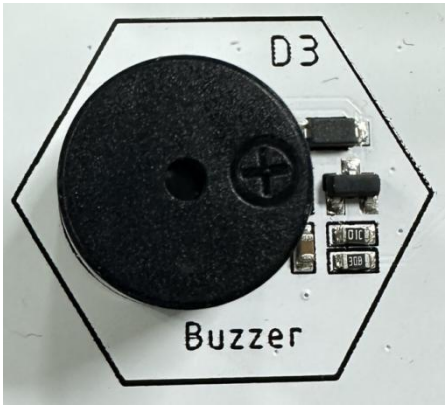
1.Understand what a passive buzzer is
2.Master the use of arrays
3.Master the use of for loops
4.Complete the example of playing the seven basic notes

Preview of the Result



After uploading the code, the buzzer will play the seven standard notes in a loop.

Hardware Used in This Lesson



The buzzer module is an output device located on the left side of the Arduino Nano R4 development board：

Buzzers can be categorized into two types:

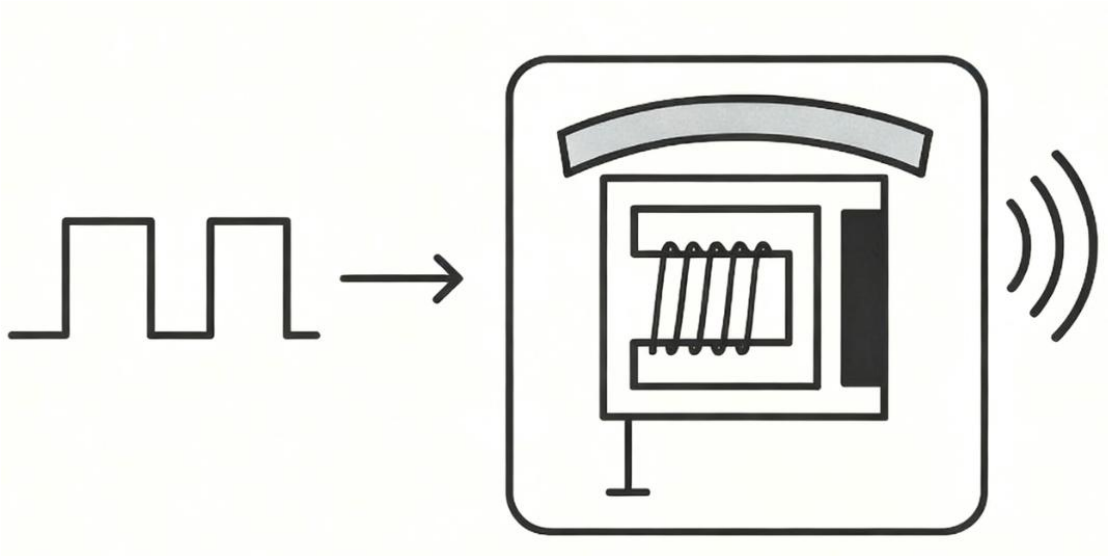
1.Active buzzer: Simple to use, it only requires HIGH or LOW signals to turn on or off. The tone frequency cannot be adjusted

2.Passive buzzer: More complex to use, the sound frequency can be adjusted. It typically uses the built-in tone() function, where you input parameters to set the desired frequency

In this lesson, we use a passive buzzer, which can be programmed to play different melodies

Working Principle of an Buzzer Module

As shown in the figure, the square wave signal on the left represents the driving signal provided by the external circuit. This driving signal is transmitted via wires to the electromagnetic coil inside the passive buzzer. When current flows through the coil, it generates a magnetic field.The coil is usually connected to a vibrating diaphragm. As the current in the coil changes, the resulting magnetic field attracts or repels a permanent magnet, causing the diaphragm to vibrate. Since the driving signal is a square wave, adjusting the signal's frequency and duty cycle allows control over the vibration frequency, thereby producing an adjustable pitch.

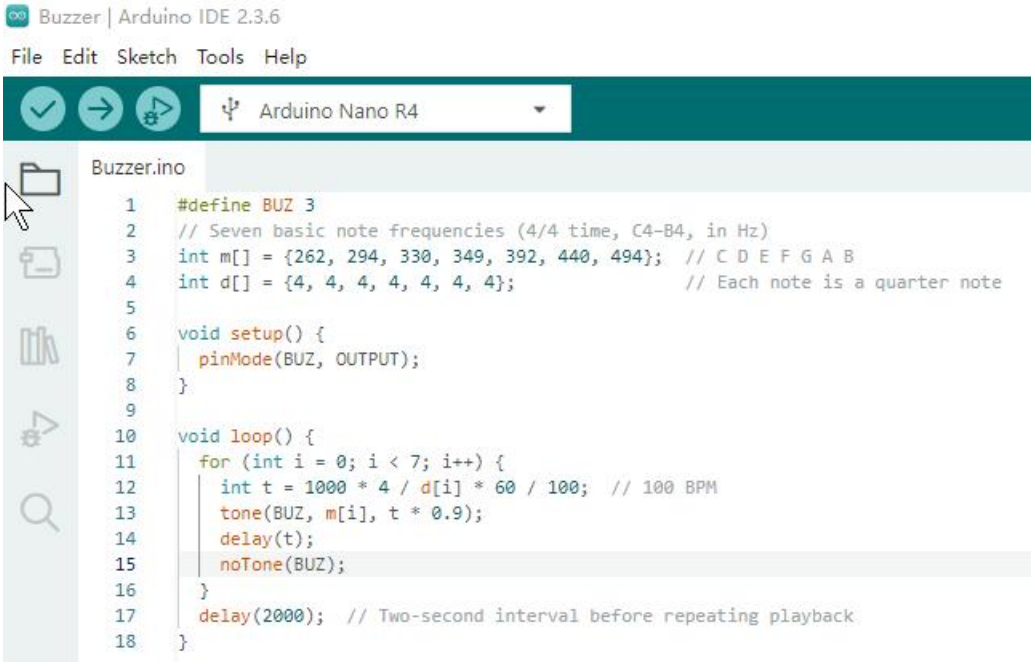


Playing the Seven Basic Notes

Before going through the code, you can download it. Here is the link to download the complete code:
https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-Arduino-Nano-R4/tree/master/lesson_code/4_Buzzer
Open the "4_Buzzer.ino" file inside the "4_Buzzer" folder using the Arduino IDE.

Key Code Explanation

Now let's go through the example: Playing the Seven Basic Notes



First, we define the pins that will be used:

```
#define BUZ 3
```

In this lesson, the buzzer is connected to the D3 pin of the Arduino Nano R4 development board. We use "#define" to name the pin, making it easier to modify and debug the code later.

Next, we use an array to define the frequency values of the seven basic notes

```
// Seven basic note frequencies (4/4 time, C4–B4, in Hz)
int m[] = {262, 294, 330, 349, 392, 440, 494};   // C D E F G A B
int d[] = {4, 4, 4, 4, 4, 4, 4};                 // Each note is a quarter note
```

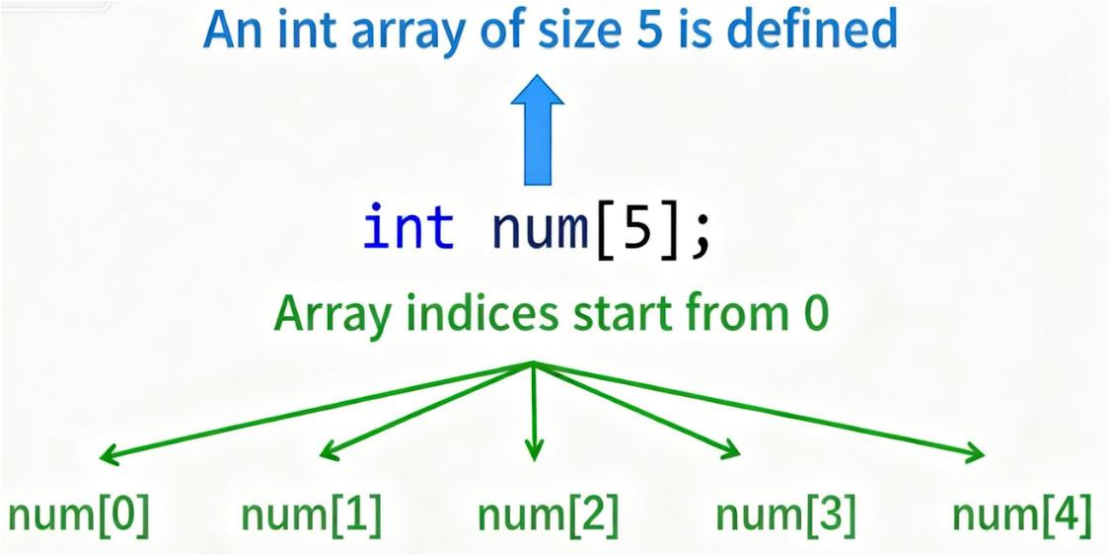
This is the key point of this lesson — arrays

There are many ways to define arrays, but here we will introduce the two most commonly used methods:

1.When specifying the number of elements

Define an array of type int, where int indicates that all elements in the array are integers.

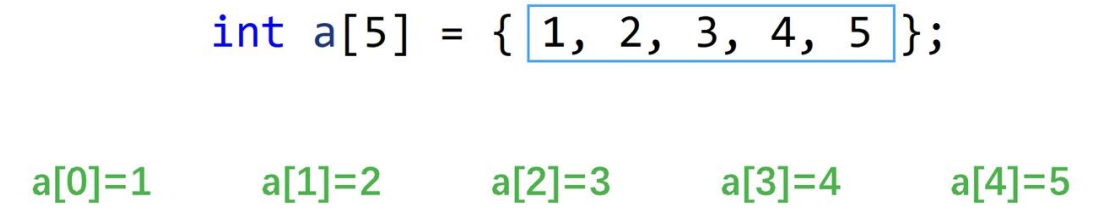
int num[5]:This defines an integer array named num with five elements, but the specific values of the elements have not been assigned yet.



As shown in the figure, the elements of the "num" array are num[0], num[1], num[2], num[3], and num[4]

Note: The indexing starts from 0, so "num[0]" is the first element, and "num[1]" is the second element of the array.

int a[5] = {1,2,3,4,5}:This defines an integer array named "a" with five elements: 1, 2, 3, 4, and 5



As shown in the figure, "a[0]" is the first element of the array "a" and equals 1. Once we define the entire array, we can directly refer to its elements using the "Array[]" notation

2.When the number of elements is uncertain

int a[] = {1,2,3}:When defining an array this way, the [] is left empty, and the array length is automatically determined by the number of elements provided. This method is useful when you might need to add more elements later, as you don't need to modify the array's initial length.

In this lesson's example, we use the second method to define arrays, which means we can directly add elements later to include the additional notes we want to play.

Initialization Function

```
void setup() {
  pinMode(BUZ, OUTPUT);
}
```

In the initialization function, we set the buzzer pin to output mode

Inside the loop() function

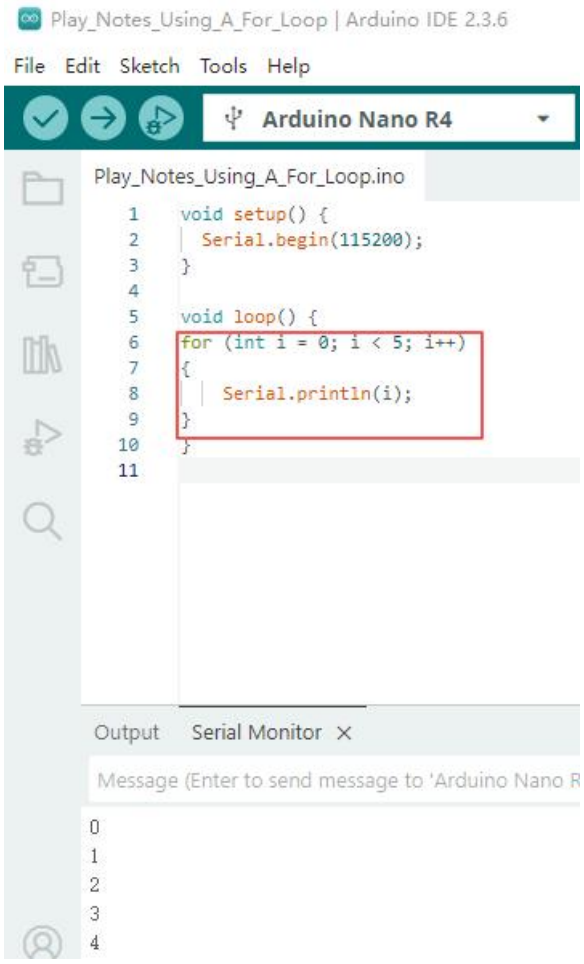
```
void loop() {
  for (int i = 0; i < 7; i++) {
    int t = 1000 * 4 / d[i] * 60 / 100;   // 100 BPM
    tone(BUZ, m[i], t * 0.9);
    delay(t);
    noTone(BUZ);
  }
  delay(2000);   // Two-second interval before repeating playback
}
```

Inside the loop() function, we use a for loop to simplify the code, allowing the program to play all seven standard notes within a single for loop.

for loop

A for loop allows us to automatically and repeatedly execute code a certain number of times. When combined with arrays, it enables us to process multiple elements in bulk, greatly simplifying the code and making the program more efficient and organized.

For example:



Let's break down the main structure of a for loop:

1.Initialization: int i = 0

This defines the loop variable i and initializes it to 0.

2.Loop Condition: i < 5

As long as this condition is true, the code inside the loop body (Serial.println(i)) will execute. When the condition is false, the loop ends

3.Loop Body: Serial.println(i)

Each iteration prints the current value of the variable i

4.Counter Update: i++

After each iteration, i automatically increases by 1. Then it goes back to the condition check to decide whether to continue

Execution Result: 0, 1, 2, 3, 4

When i equals 5, i < 5 is false, and the loop ends

Back to our loop() function code:

We need to loop 7 times, with the index starting at 0. So we initialize i as 0. The condition i < 7 ensures the loop continues until i reaches 7. i++ means that with each iteration, i increases by 1.Thus, during the first iteration, the loop executes as shown in the figure below:

```
int t = 1000 * 4 / d[0] * 60 / 100;  // Calculate note duration at 100 BPM
tone(BUZ, m[0], t * 0.9);           // Play the note
delay(t);                            // Wait for the note to finish
noTone(BUZ);
```

During the second iteration:

```
int t = 1000 * 4 / d[1] * 60 / 100;  // Calculate note duration at 100 BPM
tone(BUZ, m[1], t * 0.9);           // Play the note
delay(t);                            // Wait for the note to finish
noTone(BUZ);
```

You'll notice that in each loop iteration, only the index value changes according to the variable "i", achieving the desired effect.

Explanation of the "tone()" function parameters

The "tone()" function can take three parameters:

pin:Specifies the hardware pin connected to the buzzer.

frequency:Specifies the frequency of the note the buzzer will play. Higher frequency = higher pitch, lower frequency = lower pitch.

duration:Specifies how long the buzzer will sound, in milliseconds. If this parameter is omitted, the buzzer will continue sounding until "noTone()" is called.

tone(3,262,500):The buzzer on pin 3 plays a frequency of 262 Hz for 500 milliseconds. Here, 262 corresponds to the note "C" on a piano.

Explanation of the "noTone()" function

The "noTone()" function takes only one parameter, the pin number, and stops the buzzer on that pin.

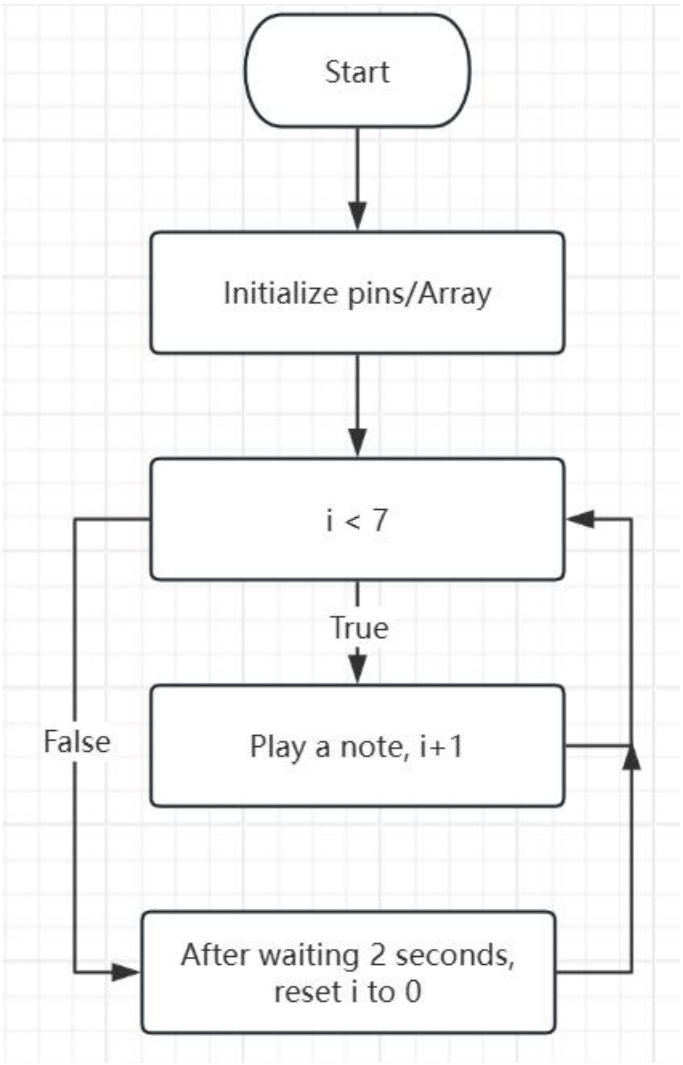
noTone(3):Trun off the buzzer on pin 3.

Let's take a look at this code. We need to understand what the variables "m", "d", and "t0" represent and where their values come from.
In the code for playing melodies with the buzzer, there are several key variables to note:
m:Represents the frequency of the note, which determines the pitch of the buzzer. For example, 262 corresponds to the C note.
d:Represents the note duration value. Here, 4 stands for a quarter note, which is used to calculate the actual play time.
t0:The actual duration of the note, calculated based on the value of "d" (quarter note, eighth note, etc.).
The note frequencies come from predefined musical scores. In later sections, we'll also discuss the specific frequencies for other standard notes.

Key Concept:
t0 = 1000 * 4 / d * 60 / 100:This formula calculates the actual play time of a note. You only need to change the variable "d" to get different note durations, for example:
d = 4 → quarter note
d = 8 → eighth note
d = 2 → half note

tone(BUZ,m,t0*0.9):Plays a note at frequency "262" on the buzzer connected to pin 3 for t0*0.9 milliseconds.
Here's why t0*0.9 is used:
When the program reaches this line, it doesn't wait for the buzzer to finish sounding before moving on. The buzzer receives the command, and the program continues executing the next lines.
There is a delay(t0) after this line to pause the program for to milliseconds.If we used "t0" as the duration in tone(), the buzzer would sound for the full duration with no gap between notes.
For example, if t0 = 1000 ms, the buzzer sounds for 1000*0.9 = 900 ms, and the program delays 1000 ms, leaving a 100 ms gap between notes. This gap adds rhythm, making the melody sound more natural.

Overall Code Logic Flowchart



Program Upload Steps

For detailed upload instructions, please refer to the "Upload Steps" on page 8.

Key Takeaways:

tone()	Used to make the buzzer sound; you can specify the pin, note frequency, and duration.
noTone()	Used to stop the buzzer from sounding.
int a[3] = {1,2,3}	Define an integer array a with three elements: 1, 2, 3.
int a[] = {1,2,3}	Define an integer array a with its length automatically set to 3 based on the number of elements:
for(int i =0; i < 7;i++)	Repeatedly executed 7 times, with "i" increasing from 0 to 6, incrementing by 1 each time.

Lesson05---Relay

Introduction

In this lesson, we will learn about relays, with a focus on understanding how they work. Through an LED lighting experiment, we will visually demonstrate and explain the relay's operating process in detail. By the end of this lesson, students will understand the basic principles of relays and master their basic usage.

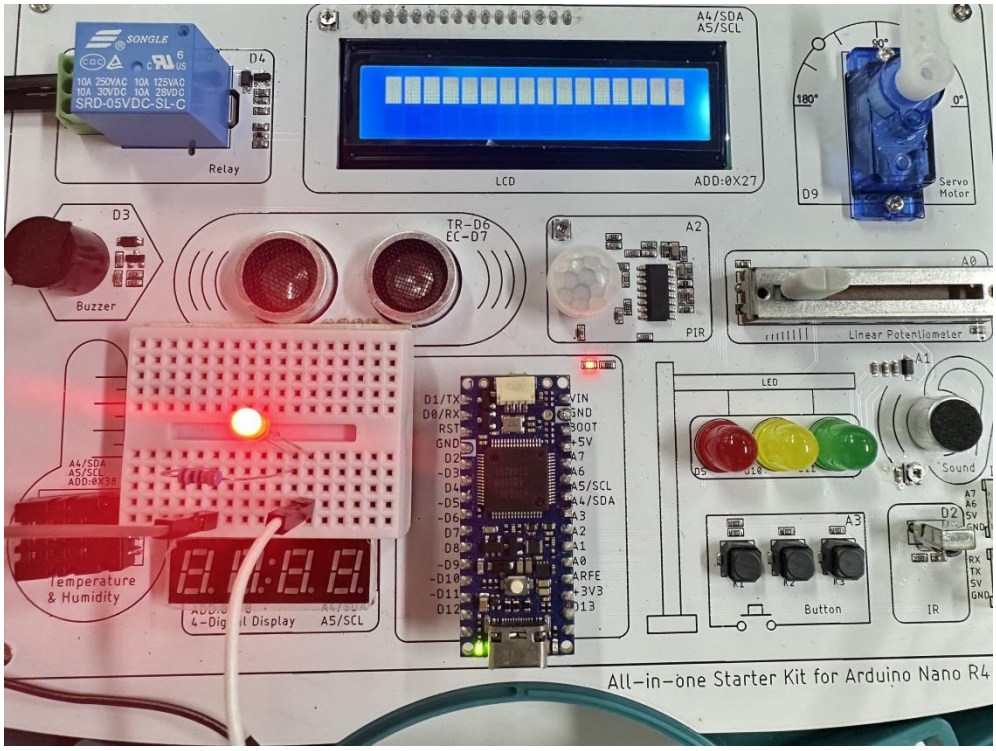
Note: The breadboard, jumper wires, resistors, and LED kit mentioned in this course are not provided and must be prepared separately.

Learning Goals

1.Understand the function of a relay

2.Master the working principle of a relay

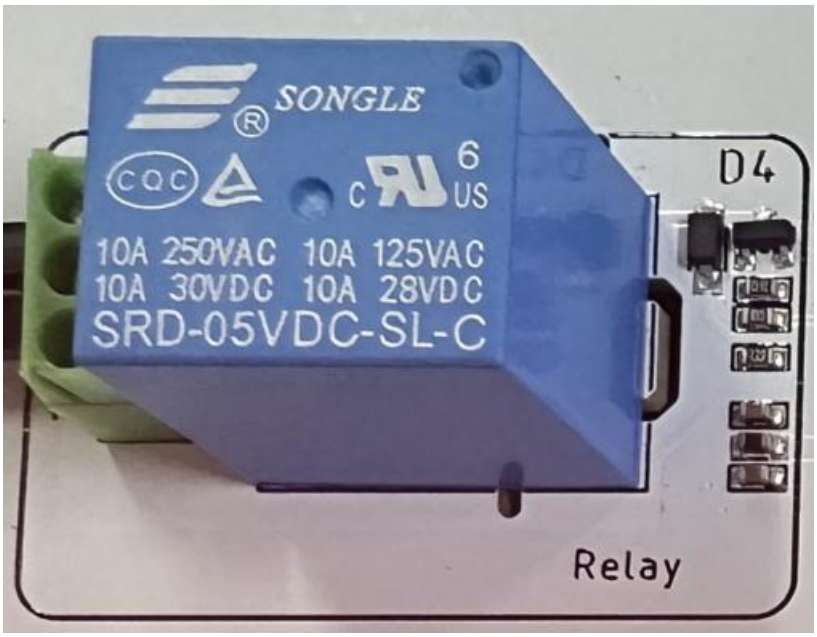
Preview of the Result



After uploading the code, the LED on the breadboard will be controlled by the relay to turn on for 3 seconds and then turn off for 3 seconds.

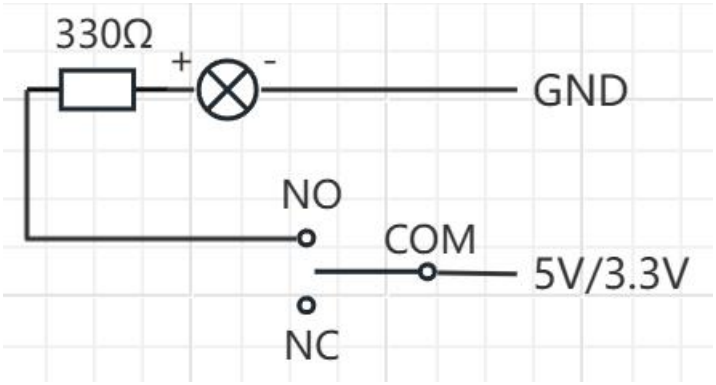
If no external LED is connected, after uploading the code the relay will only make a "clicking" sound. This sound is caused by the relay switching connections between the NO and NC terminals.

Hardware Used in This Lesson



The relay module is an output device located at the upper-left side of the Arduino Nano R4 development board.

In this lesson, we also use a breadboard to build an LED circuit. The circuit setup is shown in the figure below:

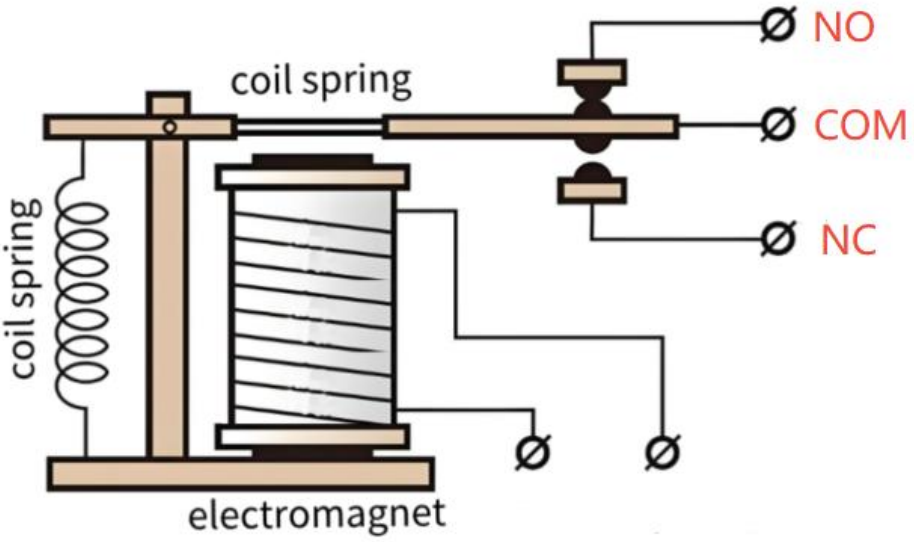


The LED is connected in series with a 330-ohm resistor and wired to the relay's NO terminal. The other end of the LED is connected to GND (here we use the GND pin at the lower-right corner of the Arduino Nano R4). The relay's COM terminal is connected to 5V (using the 5V pin at the lower-right corner of the Arduino Nano R4).

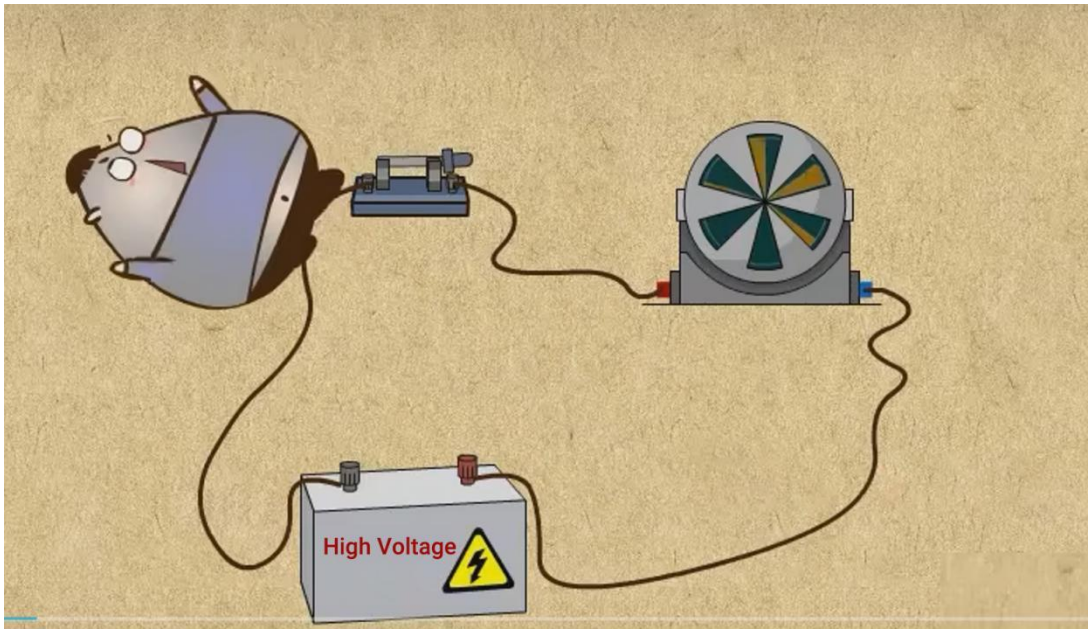
Working Principle of an Relay Module

A relay is an electrically controlled switch with internal coils and contacts. Common contacts include COM (common), NC (normally closed), and NO (normally open). In its normal, unpowered state, COM is connected to NC and disconnected from NO. When the coil is energized, it generates a magnetic force that moves the contacts, switching COM from NC to NO. At this point, COM is connected to NO and disconnected from NC.By using this mechanism, a relay allows a small-current, low-voltage control signal to safely control high-current or high-voltage loads,

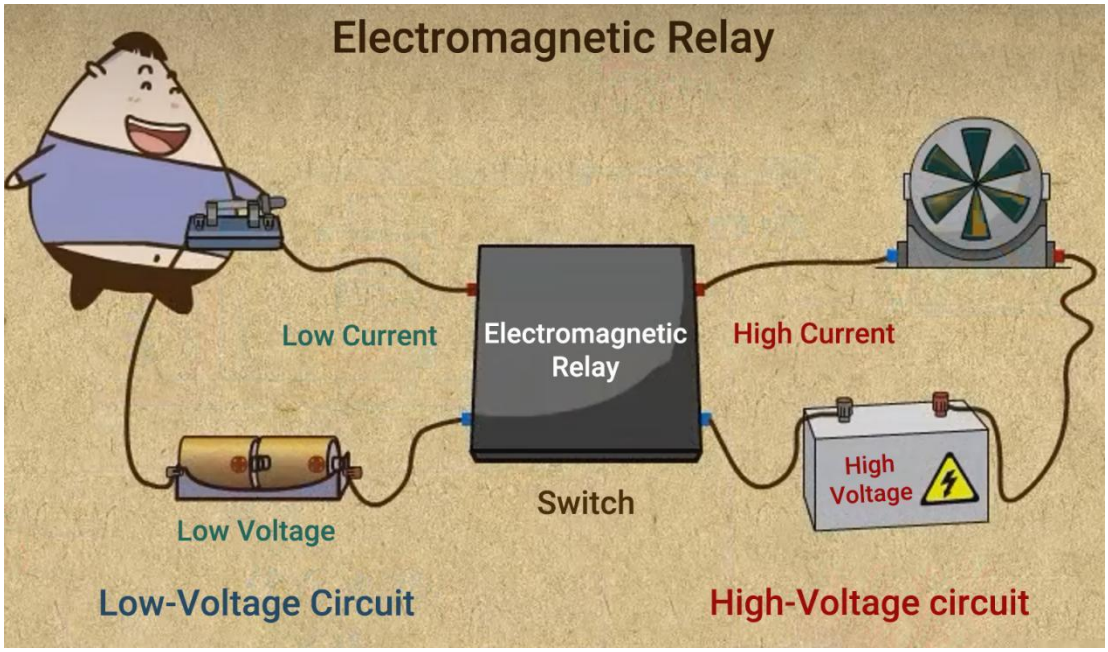
enabling circuits to be switched on, off, or redirected.



Without a relay, directly controlling high-voltage equipment is extremely dangerous:



With the help of a relay to control high-voltage equipment, it is safe for people:



Relay-Controlled LED Example

Before going through the code, you can download it. Here is the link to download the complete code:
https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-Arduino-Nano-R4/tree/master/lesson_code/5_Relay
Open the "5_Relay.ino" file in the "5_Relay" folder using the Arduino IDE.

Key Code Explanation

Now let's walk through the example: Relay-Controlled LED Example

First, we start by defining the pins that will be used.

```
#define Relay 4
```

On the "Arduino Nano R4" development board, the relay is connected to pin D4 of the main controller. We use #define to name pin D4 as "Relay".

Initialization Function

```
void setup() {
  pinMode(Relay, OUTPUT); // Set the relay pin as an output
}
```

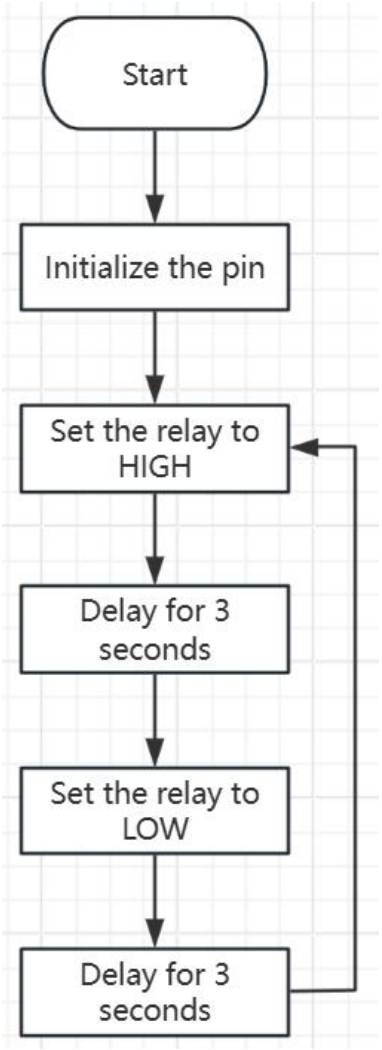
Set the "Relay" pin to output mode.

Loop Function

```
void loop() {
  digitalWrite(Relay, HIGH); // Turn the relay ON
  delay(3000);               // Keep the relay ON for 3 seconds
  digitalWrite(Relay, LOW);  // Turn the relay OFF
  delay(3000);               // Keep the relay OFF for 3 seconds
}
```

Use "digitalWrite" to set the pin HIGH or LOW to control the relay. When a HIGH level is applied, the relay's "COM" terminal connects to the "NO" terminal. Conversely, when a LOW level is applied, the relay's "COM" terminal connects to the "NC" terminal.

Overall Code Logic Flowchart



Program Upload Steps

For detailed upload instructions, please refer to the "Upload Steps" on page 8.

Key Takeaways:

Relay Working Principle	When a HIGH signal is applied to the relay, the COM terminal connects to the NO terminal. When a LOW signal is applied, the COM terminal connects to the NC terminal.

Lesson06---LinearPotentiometer

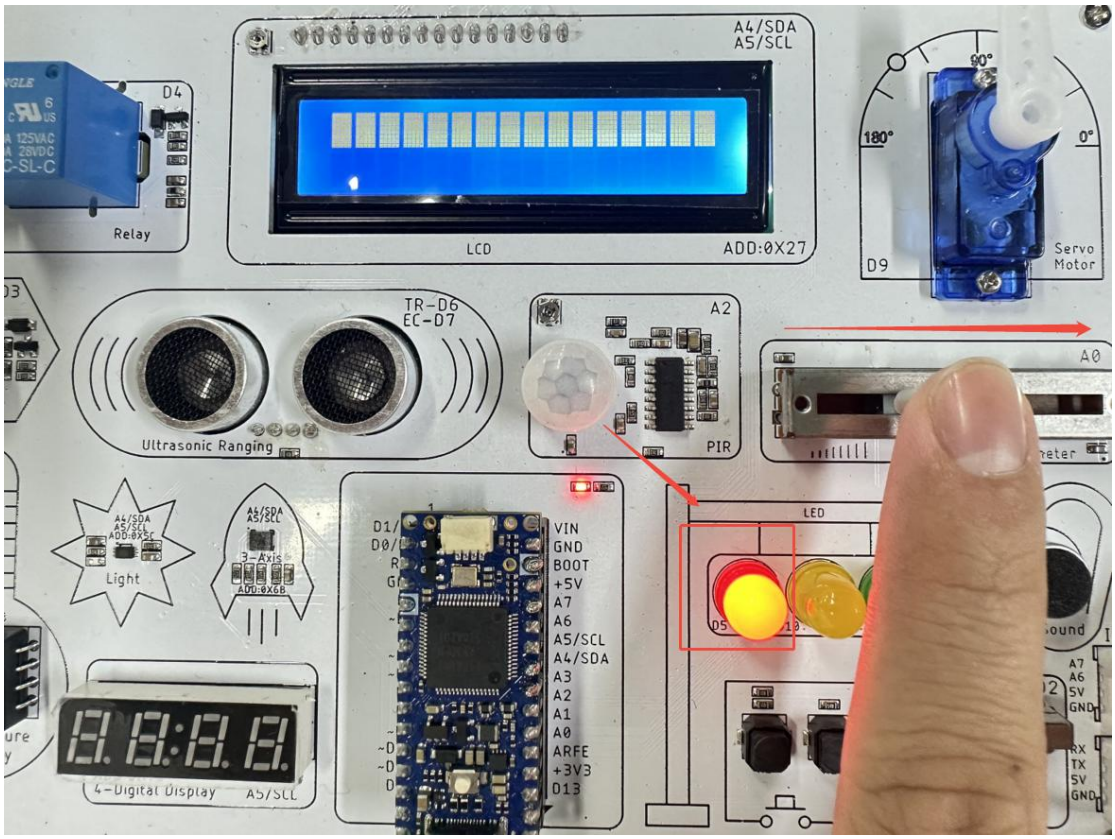
Introduction

In this lesson, you will learn how to use PWM to control the brightness of an LED by working with a "Linear Potentiometer" module. Through a hands-on example, you'll gain an understanding of how PWM works and learn how to adjust LED brightness using "analogWrite".

Learning Goals

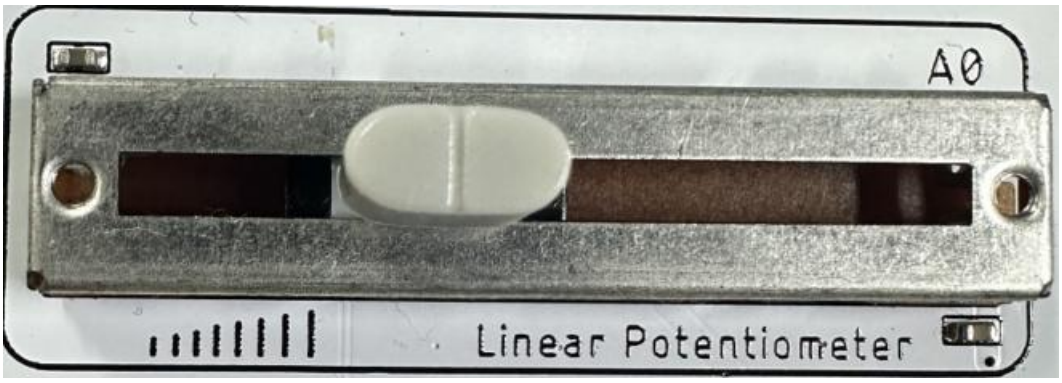
1.Understand how a slide potentiometer works
2.Learn how to use operators in a program and apply them to map values from one range to another
3.Understand the principles of PWM control
4.Master how to control LED brightness using "analogWrite"
5.Complete an experiment that uses a slide potentiometer to smoothly control the brightness of an LED

Preview of the Result



After uploading the code, as you slide the potentiometer from left to right, the red LED will gradually become brighter.

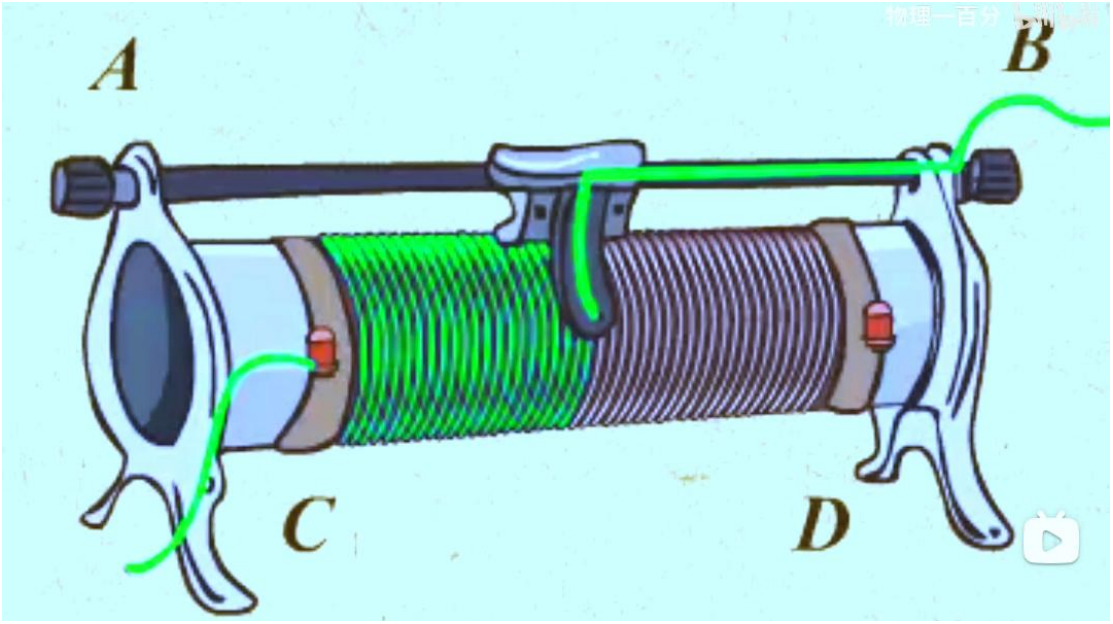
Hardware Used in This Lesson



The Linear Potentiometer module is located on the right side of the Arduino Nano R4 board: Its signal pin is connected to the Arduino Nano R4's A0 pin, which reads analog values ranging from 0 to 1023. When the Linear Potentiometer is slid all the way to the left, the analog reading is 0. When it is slid all the way to the right, the reading reaches 1023.

Working Principle of an Linear Potentiometer Module

In this lesson, the "Linear Potentiometer" module functions like a metal wiper that moves along a resistive track. As shown in the diagram, the sliding range runs from point A to point B. The effective resistance is highlighted in green and represents the portion of the resistive track between point C and the wiper. When the wiper is moved all the way to the right, the length of the resistive track between points C and B is at its maximum, resulting in the highest resistance.



Slide Potentiometer – Controlled LED Brightness Example

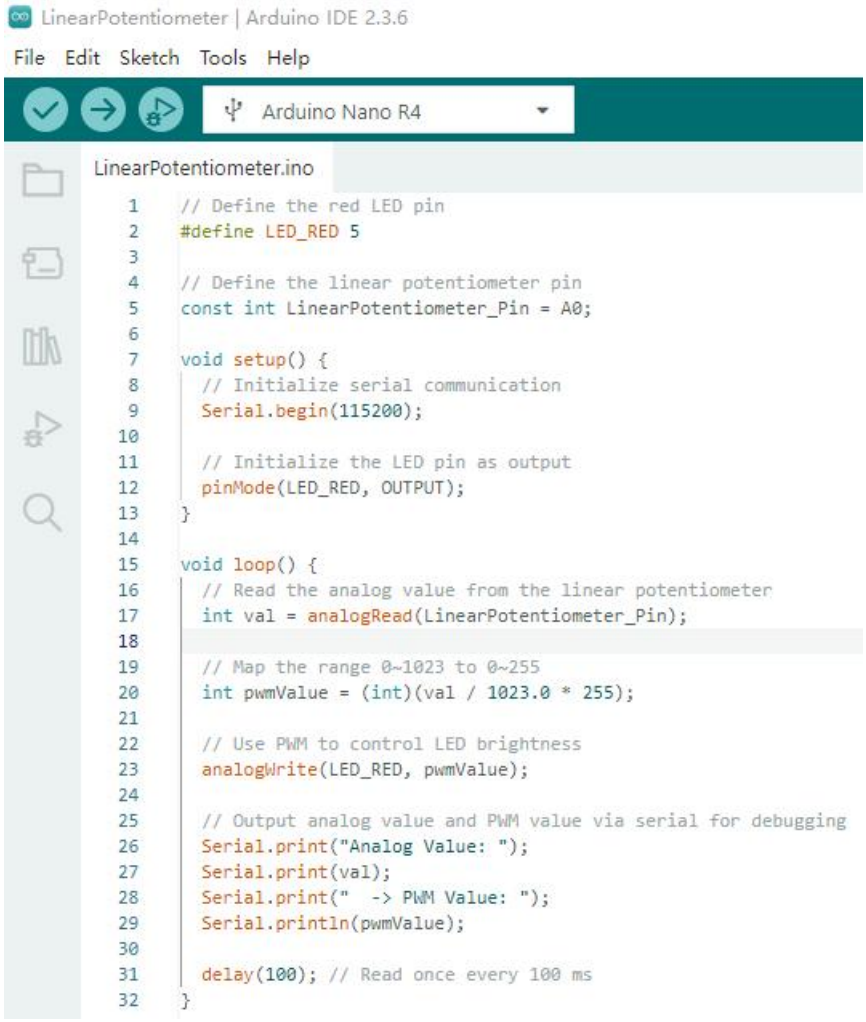
Before walking through the code, you can download it first. Complete code download link:

https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-Arduino-Nano-R4/tree/master/lesson_code/6_Linear_Potentionmeter

Open the "6_Linear_Potentionmeter" folder in the Arduino IDE, then open the "6_Linear_Potentionmeter.ino" file.

Key Code Explanation

Now let's walk through the example: using a slide potentiometer to control the brightness of an LED.



First, let's define the pins that will be used:

```
#define LED_RED 5
const int LinearPotentiometer_Pin = A0;
```

In this example, we'll use one LED and a slide potentiometer. The LED is a red LED connected to pin 5 , and the slide potentiometer is connected to analog pin A0.

Initialization Function

```
void setup() {
  Serial.begin(115200);
  pinMode(LED_RED, OUTPUT);
```

```
}
```

Initialize the serial port and set the LED pin to output mode—both of these have been covered in earlier lessons. Be sure that the baud rate you set here matches the baud rate in the Serial Monitor; otherwise, you'll see garbled output.

In the loop function

```
void loop() {
  int val = analogRead(LinearPotentiometer_Pin);
```

Read the value returned by the slide potentiometer. Its output range is from 0 to 1023, so we define it as an int.

```
  int pwmValue = (int)(val / 1023.0 * 255);
```

This step is critical: we need to convert the range from 0 – 1023 to 0 – 255. We'll explain why this conversion is necessary in a moment. For now, let's focus on how to do it.

1.To convert a range of (0 – 1) to (0 – 255), simply multiply by 255: (0-1)*255 -> (0-255)

2.To convert a range of (0 – 1023) to (0 – 1), divide by 1023: (0-1023)/1023 -> (0-1)

3.Therefore, the formula to convert (0 – 1023) to (0 – 255) is: (0-1023)/1023.0 * 255

Assign the final result to the integer variable "pwmValue".

```
  analogWrite(LED_RED, pwmValue);
```

The "analogWrite" function outputs a PWM signal to a pin, allowing it to simulate different voltage levels and produce varying output intensities.

Let's first take a look at how PWM works:

PWM stands for Pulse Width Modulation. Although a pin can only output either a HIGH or LOW signal, we can simulate different voltage levels by rapidly switching between HIGH and LOW and adjusting how long the signal stays HIGH during each cycle.

Duty Cycle:refers to the percentage of time the signal remains HIGH within one complete cycle.

As shown in the diagram：

HIGH time = 0 → Duty cycle = 0% → LED is off

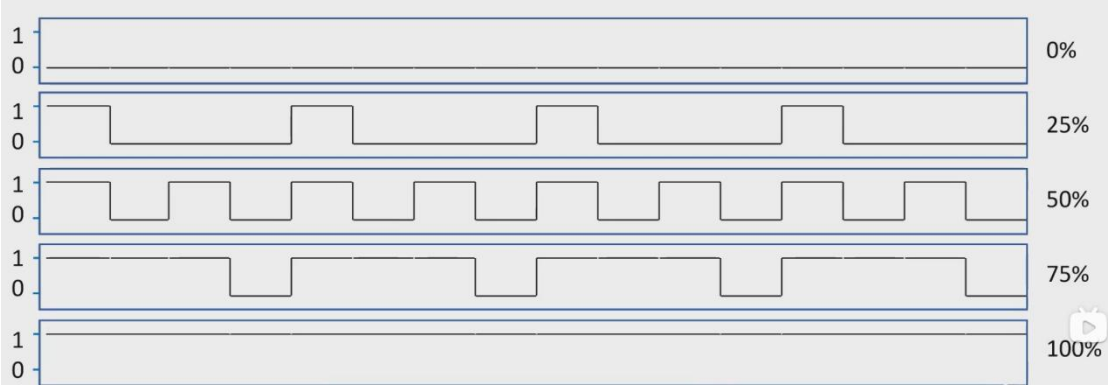
HIGH time = 25% → Duty cycle = 25% → LED brightness is 25%

HIGH time = 50% → Duty cycle = 50% → LED brightness is 50%

HIGH time = 75% → Duty cycle = 75% → LED brightness is 75%

HIGH time = 100% → Duty cycle = 100% → LED is fully on

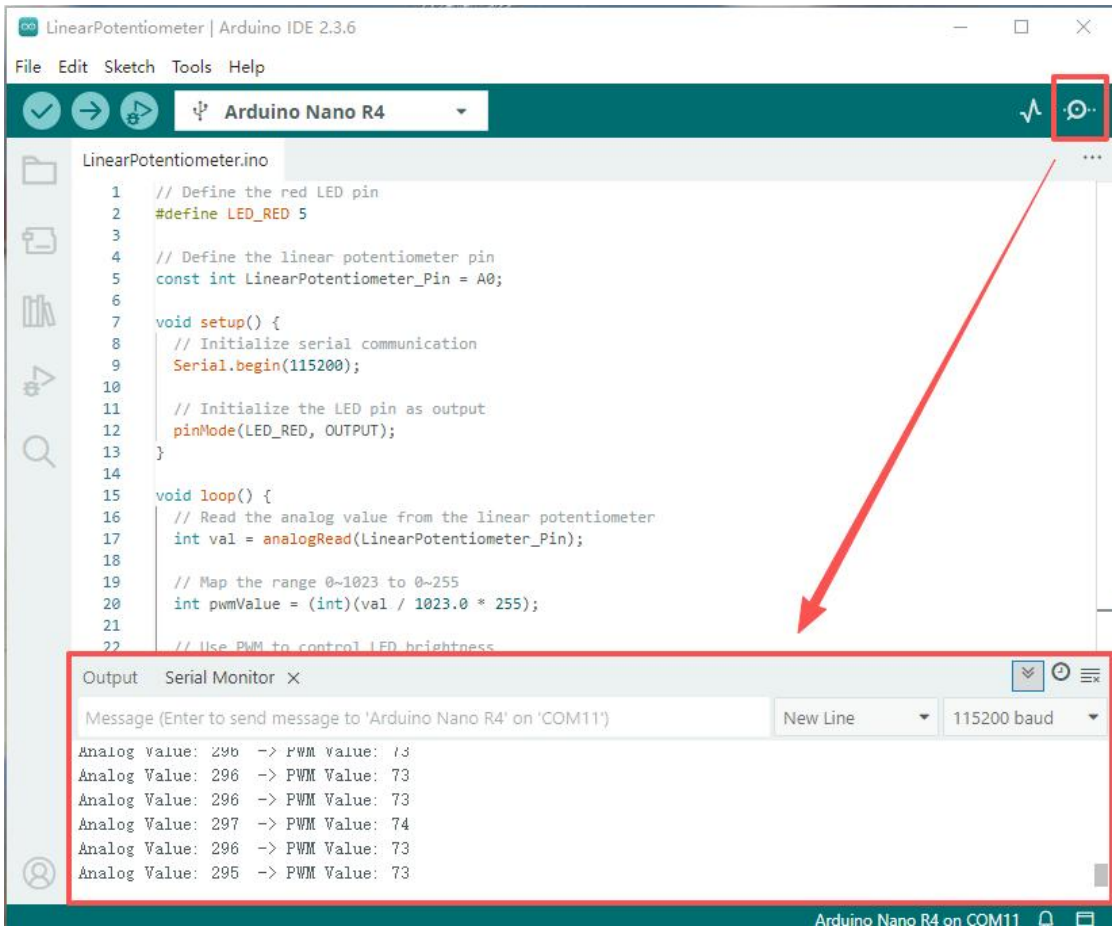
By increasing the duty cycle, the LED becomes brighter. By decreasing the duty cycle, the LED becomes dimmer.

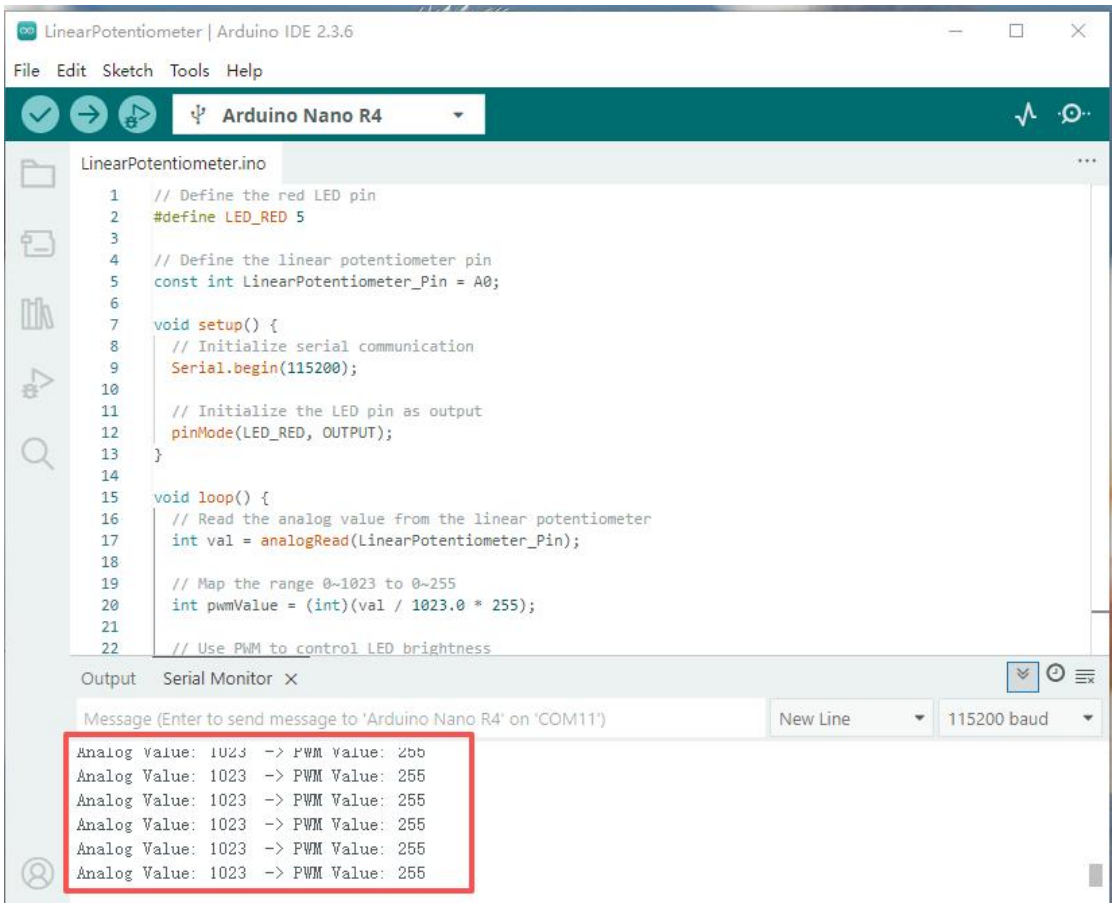


PWM pins accept values in the range of 0 – 255:which is why we need to convert the earlier readings into this range. By mapping the 0 – 1023 input range to 0 – 255, the slide potentiometer can directly control the LED's brightness. When the potentiometer is moved all the way to the left and the value is 0, the PWM value written is 0 and the LED is at its dimmest. When it is moved all the way to the right and the value reaches 1023, the PWM value written is 255 and the LED is at its brightest.

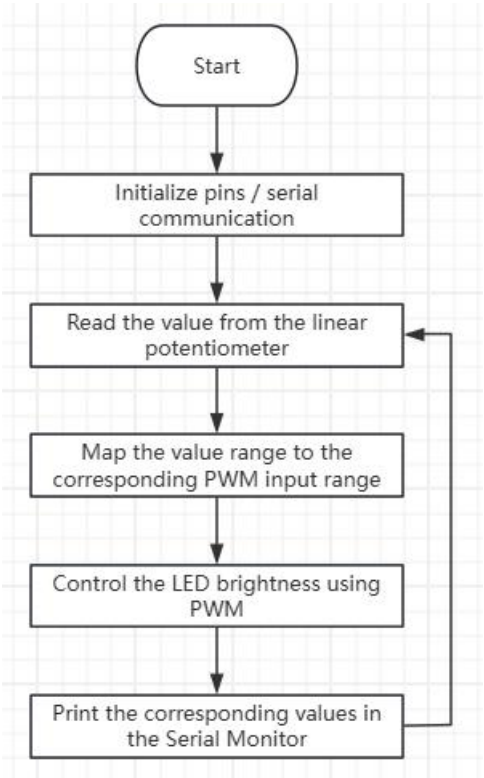
```
  Serial.print("Analog Value: ");
  Serial.print(val);
  Serial.print("   -> PWM Value: ");
  Serial.println(pwmValue);
  delay(100); // Read once every 100 ms
}
```

Finally, you can use the Serial Monitor to observe how the values read from the slide potentiometer are mapped to their corresponding PWM values.


LinearPotentiometer | Arduino IDE 2.3.6
File Edit Sketch Tools Help
Arduino Nano R4
LinearPotentiometer.ino
// Define the red LED pin
#define LED_RED 5
// Define the linear potentiometer pin
const int LinearPotentiometer_Pin = A0;
void setup() {
// Initialize serial communication
Serial.begin(115200);
// Initialize the LED pin as output
pinMode(LED_RED, OUTPUT);
}
void loop() {
// Read the analog value from the linear potentiometer
int val = analogRead(LinearPotentiometer_Pin);
// Map the range 0~1023 to 0~255
int pwmValue = (int)(val / 1023.0 * 255);
Output Serial Monitor
Message (Enter to send message to 'Arduino Nano R4' on 'COM11')
New Line
115200 baud
Analog Value: 296 -> PWM Value: 73
Analog Value: 296 -> PWM Value: 73
Analog Value: 297 -> PWM Value: 74
Analog Value: 296 -> PWM Value: 73
Analog Value: 295 -> PWM Value: 73
Arduino Nano R4 on COM11



Overall Code Logic Flowchart



Program Upload Steps

For detailed upload instructions, please refer to the "Upload Steps" on page 8.

Key Takeaways:

analogWrite()	Analog output: Commonly used to control hardware power levels or brightness through PWM signals
*	Multiplication operator: Used in expressions to multiply two values
/	Division operator: Used in expressions to divide one value by another

Lesson07---Button Control LED

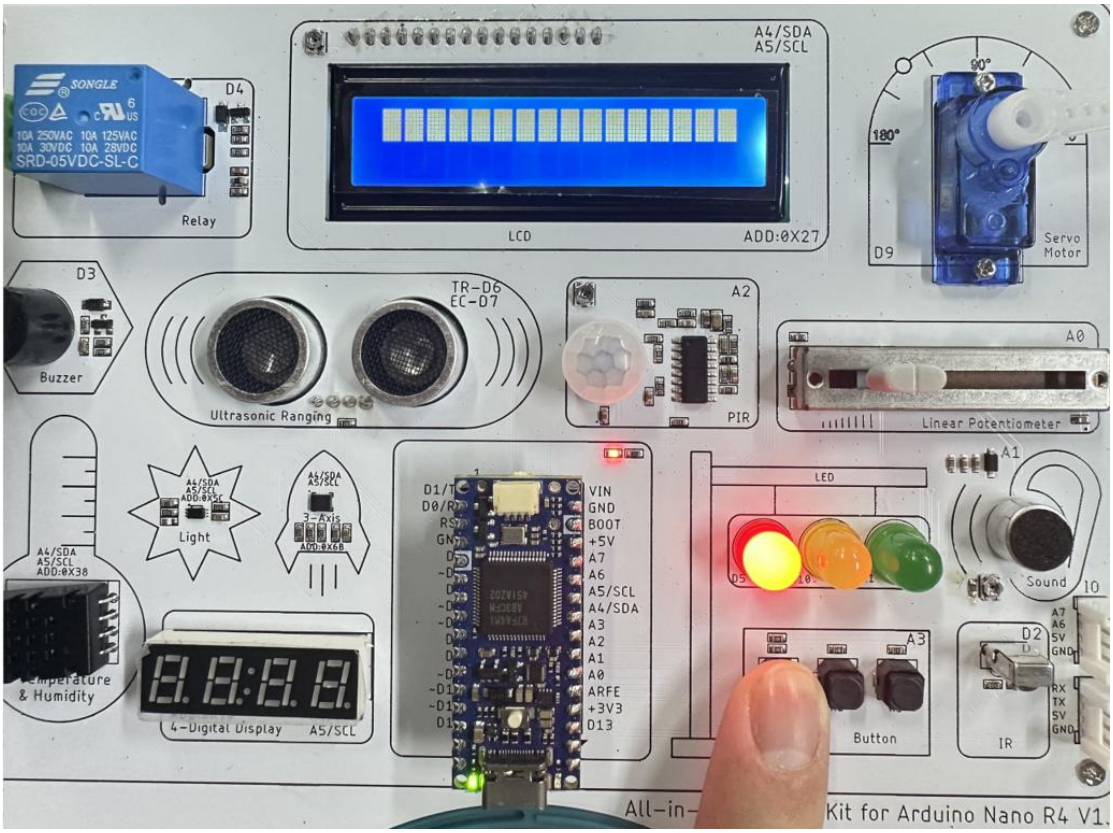
Introduction

In previous lessons, we learned how to use if conditional statements to control program execution based on a single condition. In this lesson, we will build on that foundation to learn multi-condition statements, understanding how Arduino determines and distinguishes which button has been pressed when multiple button inputs are present.By the end of this lesson, you will complete a hands-on example: using three button modules to independently control three LEDs. Through this practice, you will master the basic structure of multi-condition logic and how different conditions trigger different outcomes.

Learning Goals

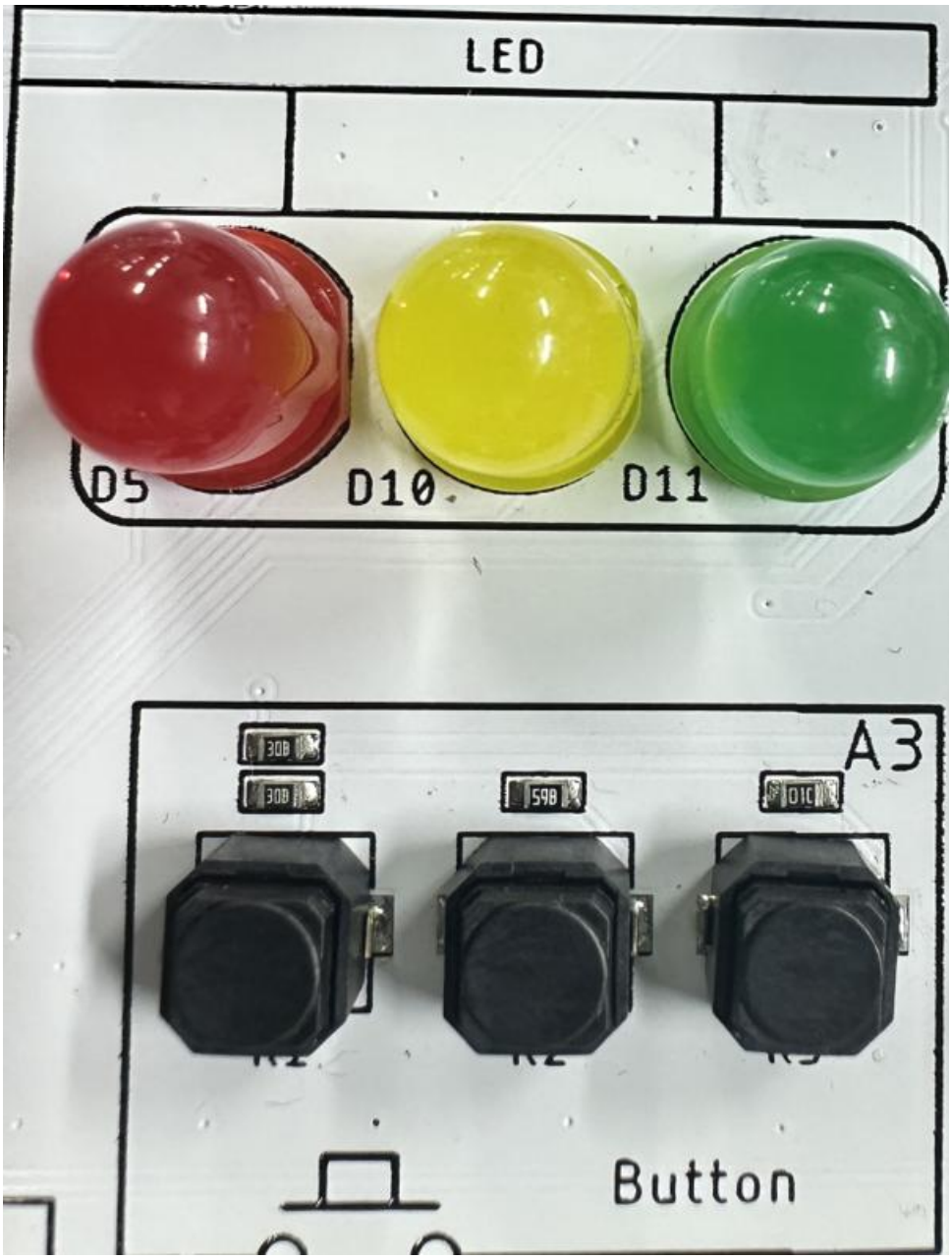
1.Understand the working principle of button modules
2.Master the execution logic of if - else if - else statements
3.Use if - else if - else statements to determine which of the three button modules is pressed and light up the corresponding LED
4.Master logical operators used within conditional statements

Preview of the Result



After uploading the code, pressing a button will turn on the corresponding LED above it.

Hardware Used in This Lesson

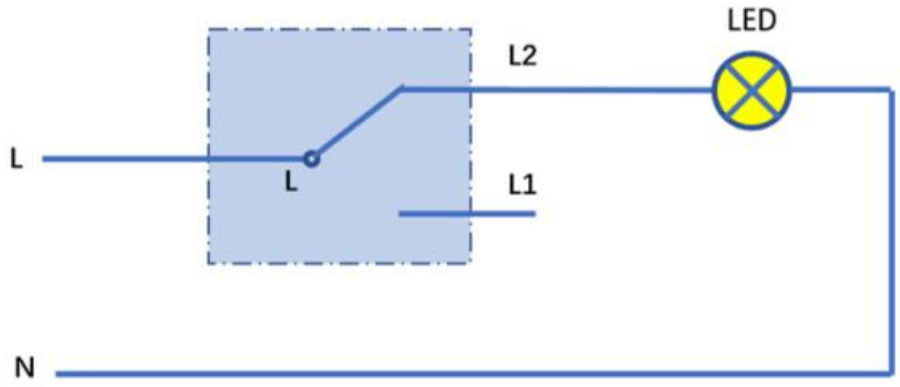


On the Arduino Nano R4 development board, there are three buttons：
All three buttons are connected to the A3 pin of the Arduino Nano R4 main controller. A3 is an analog input pin that can read analog voltage values from the pin. For these three buttons, pressing different buttons causes the A3 pin to return different analog values. By reading these values, the program can determine which button was pressed and then perform the corresponding action.

Working Principle of an Button Module

The button module we used in this lesson is equivalent to a single-pole double-throw switch. As

shown in the figure below: When the button is pressed, it is equivalent to the switch hitting the L2 end, and the circuit is turned on; when the button is released, it is equivalent to the switch hitting the L1 end, and the circuit is disconnected.



Button-Controlled LED

Before going through the code, you can download it. Here is the link to download the complete code:
https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-Arduino-Nano-R4/tree/master/lesson_code/7_Button_Control_LED
Open the "7_Button_Control_LED.ino" file in the "7_Button_Control_LED" folder using the Arduino IDE.

Key Code Explanation

Now let's go through the example: Button-Controlled LED


```
// Combine with LED lights to do button lighting experiment
// Define LED pins
#define LED_RED 5
#define LED_YELLOW 10
#define LED_GREEN 11

// Define button analog input pin
#define Button_Pin A3

void setup() {
  // Initialize LED pins as output
  pinMode(LED_RED, OUTPUT);
  pinMode(LED_YELLOW, OUTPUT);
  pinMode(LED_GREEN, OUTPUT);
  pinMode(Button_Pin, INPUT);

  // Initialize serial communication
  Serial.begin(115200);
}

void loop() {
  // Read analog value
  int val = analogRead(Button_Pin);
  Serial.print("Button Value: ");
  Serial.println(val);

  // Check the analog value range and light up the corresponding LED
  if (val >= 500 && val <= 520) {
    digitalWrite(LED_RED, HIGH);
  }
  else if (val >= 680 && val <= 690) {
    digitalWrite(LED_YELLOW, HIGH);
  }
  else if (val >= 845 && val <= 860) {
    digitalWrite(LED_GREEN, HIGH);
  }
  else{
  // Turn off all LEDs by default
  digitalWrite(LED_RED, LOW);
  digitalWrite(LED_YELLOW, LOW);
  digitalWrite(LED_GREEN, LOW);
  }

  delay(100); // Read every 100ms
}
```

First, we start by defining the pins that will be used：

```
#define LED_RED 5
#define LED_YELLOW 10
#define LED_GREEN 11
#define Button_Pin A3
```

The three LED lights are connected to pins 5, 10, and 11 respectively. There is also the button sensor pin A3 that we learned about in this lesson.

Initialization function

```
void setup() {
  pinMode(LED_RED, OUTPUT);
```

```
  pinMode(LED_YELLOW, OUTPUT);
  pinMode(LED_GREEN, OUTPUT);
  pinMode(Button_Pin, INPUT);
  Serial.begin(115200);
}
```

Initialize the corresponding pin modes: all LED pins are set to output mode, while the button sensor needs to collect returned data, so it is set to input mode.

In the loop function

```
void loop() {
  int val = analogRead(Button_Pin);
  Serial.print("Button Value: ");
  Serial.println(val);
```

Use the "analogRead" method to read the analog value returned by the button sensor. By opening the Serial Monitor, we can see that the analog values of the three buttons are approximately 511, 684, and 853. There may be slight differences between hardware units, and this is normal.

```
  // Check the analog value range and light up the corresponding LED
  if (val >= 500 && val <= 520) {
    digitalWrite(LED_RED, HIGH);
  }
  else if (val >= 680 && val <= 690) {
    digitalWrite(LED_YELLOW, HIGH);
  }
  else if (val >= 845 && val <= 860) {
    digitalWrite(LED_GREEN, HIGH);
  }
  else{
  // Turn off all LEDs by default
  digitalWrite(LED_RED, LOW);
  digitalWrite(LED_YELLOW, LOW);
  digitalWrite(LED_GREEN, LOW);
  }

  delay(100); // Read every 100ms
}
```

This is the key knowledge point of this lesson:if-else if-else. In previous lessons, we learned the if statement, which is a single-condition check. If the condition is true, the code inside the if statement will be executed.

if (val >= 500 && val <= 520) :This condition corresponds to pressing the first button. Earlier, we measured that pressing the first button returns an analog value of 511, so here we set a range that covers this value. Different hardware may return slightly different values, which is normal.

Please use the analog values returned by your own hardware as the reference.

&&:means "and". It is a logical operator. When it connects two conditions, the overall result is True only if both conditions are satisfied.

Think: Why don't we directly use a condition like "**if (val == 511)**" here?
Because, as mentioned earlier, hardware may produce slight errors. Sometimes the returned value may not be exactly 511, but 512 or another nearby value. Therefore, using a range for the condition check is more reliable and safer.

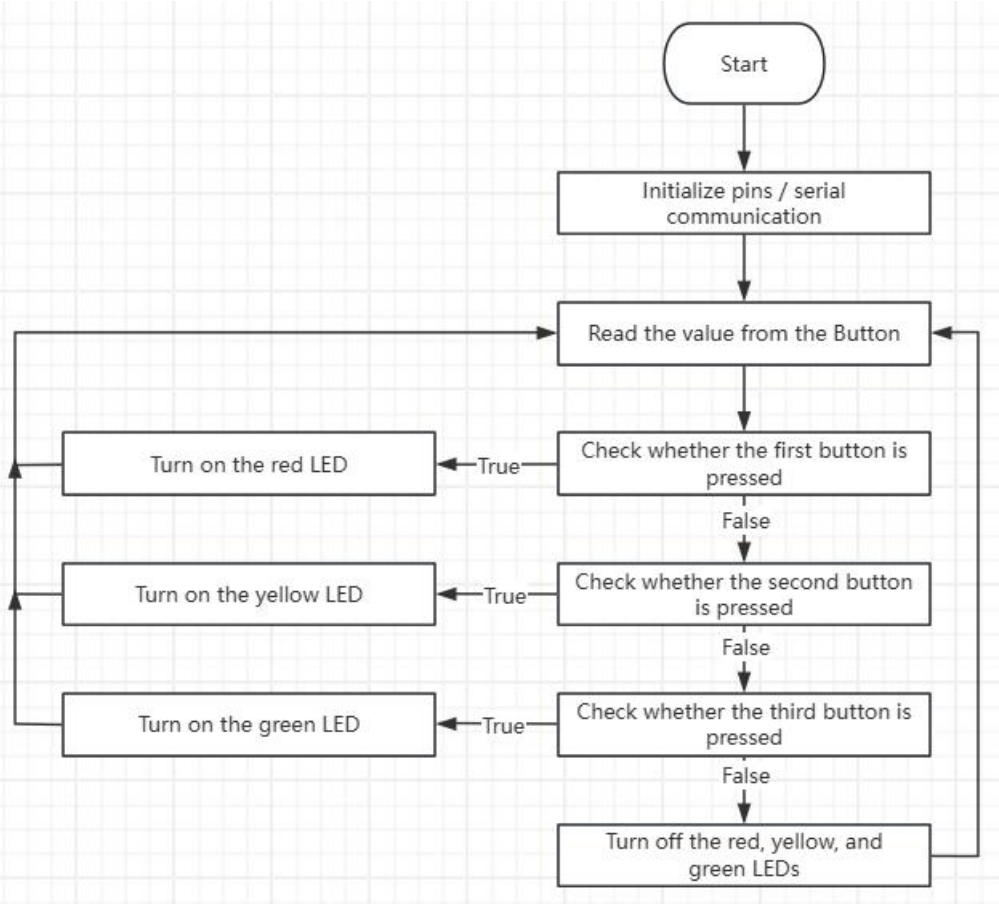
else if (val >= 680 && val <= 690):Only when the previous if condition is not satisfied will the program reach this else if statement to check whether this condition is met. In other words, the program will check whether the second button is pressed only after determining that the first button is not pressed.

else if (val >= 845 && val <= 860):Only when the second button is also not pressed will the program then check whether the third button has been pressed.

else: Finally, if none of the previous conditions are met, the program will enter this part and execute the code inside.

Let's describe the above code in words:
Check if the returned analog value is between 500 – 520 (whether the first button is pressed). If yes, turn on the red LED.
If not, check if it is between 680 – 690 (whether the second button is pressed). If yes, turn on the yellow LED.
If not, check if it is between 845 – 860 (whether the third button is pressed). If yes, turn on the green LED.
If none of these conditions are met, the program enters the else block and turns off all three LEDs.

Overall Code Logic Flowchart



Program Upload Steps

For detailed upload instructions, please refer to the "Upload Steps" on page 8.

Key Takeaways:

&&	A logical operator that represents "and". When it connects two conditions, the entire expression is True only if both conditions are True.
if	If the condition is True, the code inside the "{}" following the "if" statement will be executed.
else if	If the previous "if" condition is False, the program moves on to the next condition in sequence. If this condition is True, the code inside the following "{}" will be executed.
else	If all the previous conditions are False, the code inside the "{}" following "else" will be executed.

Lesson08---Servo

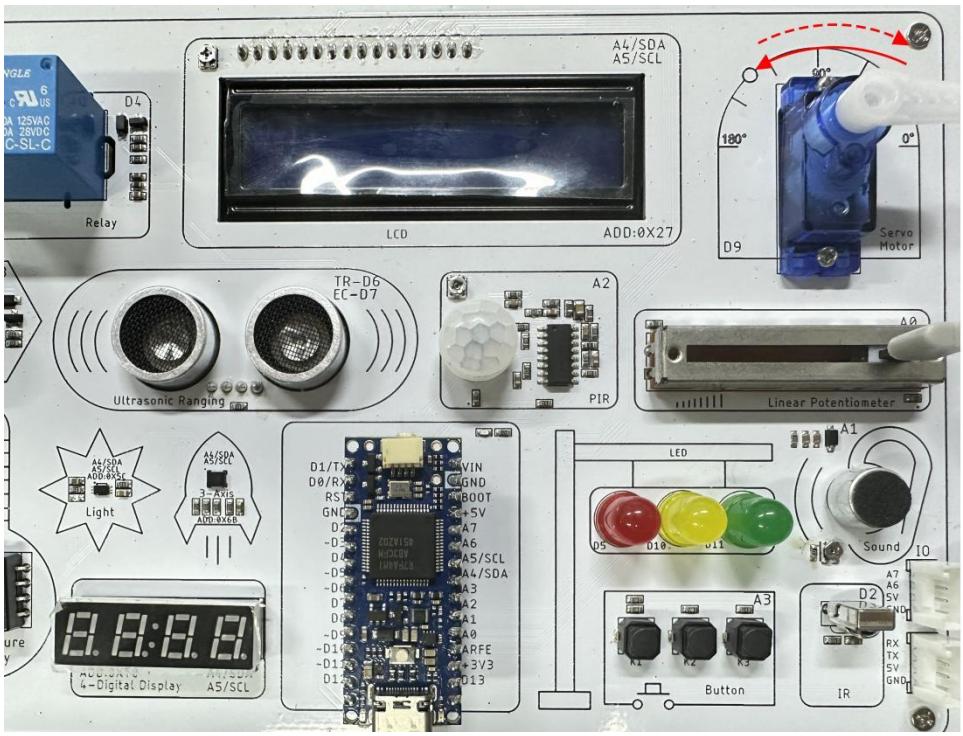
Introduction

In this lesson, you'll learn how to use a servo motor module and get your first introduction to external libraries. You'll see how to import an external library, declare a library object, and call its methods to control a servo. Through this example, you'll gain a complete understanding of the workflow for using external libraries—from importing and object creation to method invocation —laying a solid foundation for working with more modules in future lessons.

Learning Goals

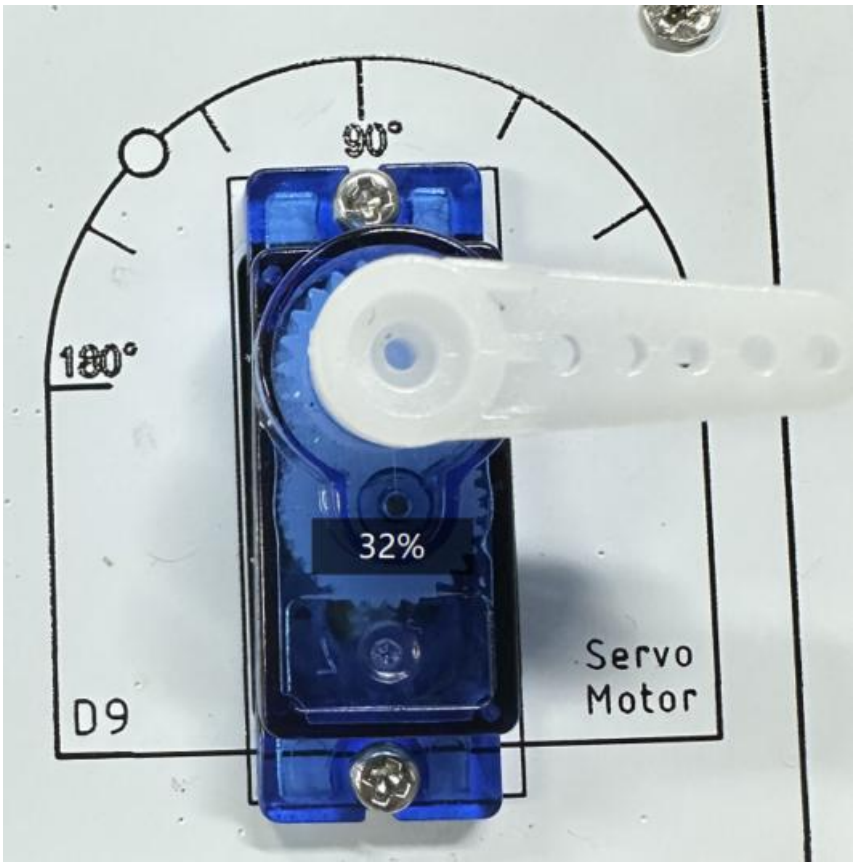
1.Understand how a servo motor module works
2.Learn how to use an external library to control a module
3.Complete an experiment in which the servo motor rotates from 0 to 180 degrees.

Preview of the Result



After uploading the code, the servo module will repeatedly rotate back and forth between 0 – 180 degrees and 180 – 0 degrees.

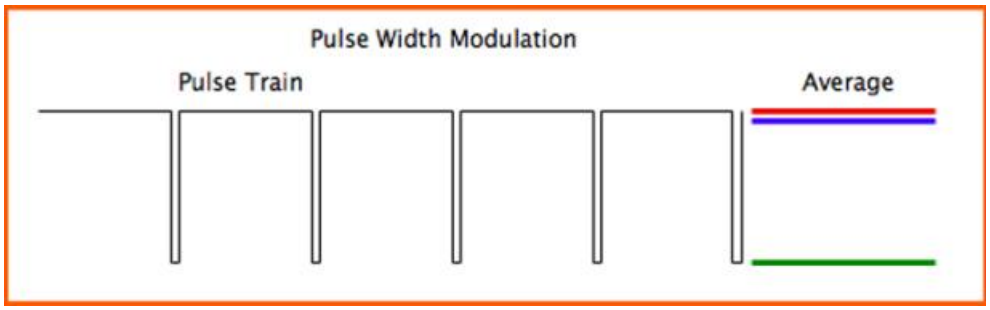
Hardware Used in This Lesson



The servo module is located in the upper-right corner of the Arduino Nano R4 board and is connected to pin D9.
D9 is a digital pin that supports PWM, allowing you to precisely control the servo's rotation angle.

Working Principle of an Servo Module

A servo motor module is also driven using a PWM (Pulse Width Modulation) signal. In earlier lessons, we learned that PWM controls motors or other modules by adjusting the width of the pulses. For a servo, the rotation angle is determined by the pulse width of the PWM signal: a longer pulse width results in a larger rotation angle, while a shorter pulse width produces a smaller angle. This mechanism allows for precise control of the servo's position.



Servo Rotation

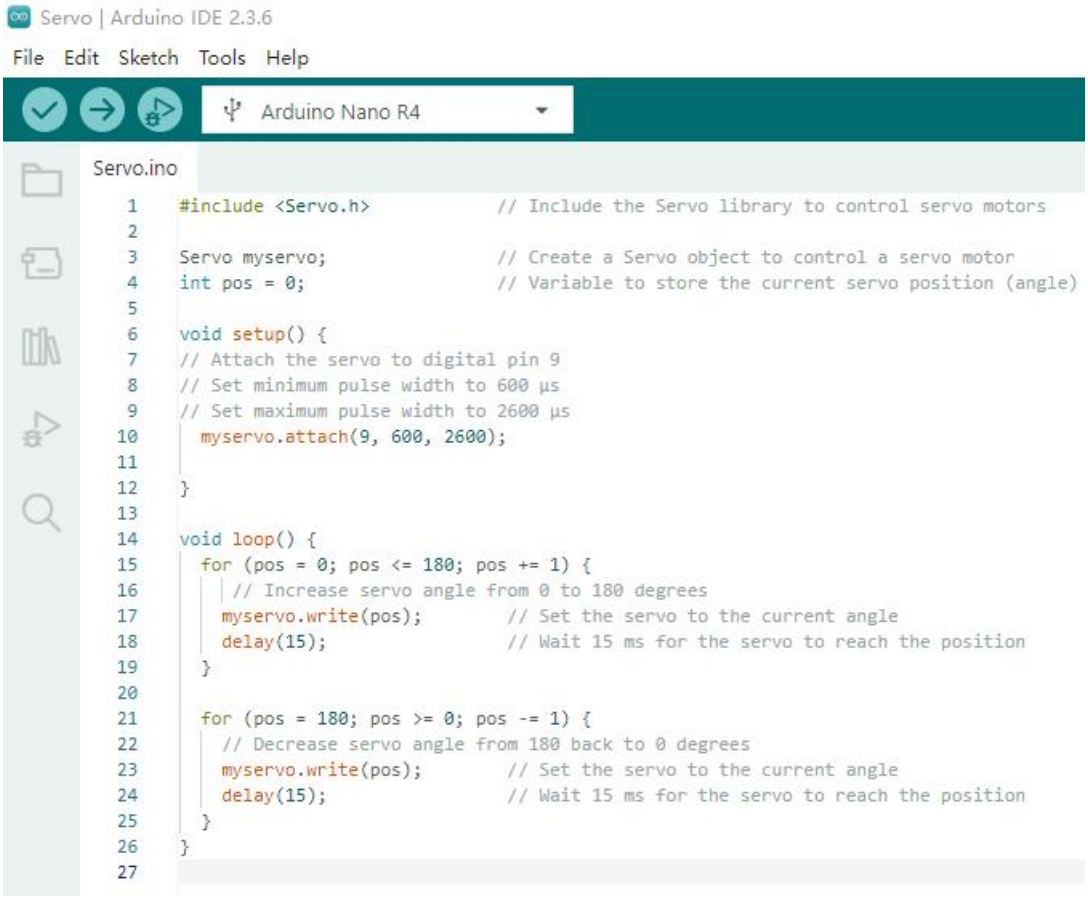
Before walking through the code, you can download it first. Complete code download link:

https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-Arduino-Nano-R4/tree/master/lesson_code/8_Servo

Open the "8_Servo" folder in the Arduino IDE, then open the "8_Servo.ino" file.

Key Code Explanation

Now let's walk through the example: servo motor rotation.



Including the Arduino Servo Control Library

```
#include <Servo.h>
```

"Servo.h" is the official servo control library provided by Arduino. It encapsulates the low-level PWM signal handling, allowing you to control a servo's rotation from 0 to 180 degrees directly without having to generate PWM waveforms yourself.

"#include" is a preprocessor directive. Before compilation, it literally copies the contents of another file into your program.

Because of this, "#include" should be used carefully—bringing in large or unnecessary libraries can increase code size and potentially slow down your project. It's best to avoid including libraries that aren't relevant to your application.

Creating a Servo Object

```
Servo myservo;
```

Create an object of the "Servo" class named "myservo". This object represents a single servo instance, and all subsequent control of the servo will be performed through "myservo".

Defining the Angle Variable

```
int pos = 0;
```

Use an integer variable named "pos" to represent the servo's current angle position, and initialize it to 0. By continuously updating this variable, you can change the servo's angle and achieve smooth, continuous movement.

Initialization Function

```
void setup() {
  myservo.attach(9, 600, 2600);
}
```

In the initialization function, we use the attach method to specify the servo pin as well as the minimum and maximum rotation limits. This step is very important.

Parameter details:

9:The servo control pin. On the Arduino Nano R4 board, this pin assignment is fixed.

600:The pulse width (in microseconds) corresponding to the minimum angle (0°).

2600:The pulse width (in microseconds) corresponding to the maximum angle (180°).

If you feel that the servo's minimum angle isn't small enough, you can decrease the value 600. If the maximum angle isn't large enough, you can increase 2600.

In the Loop Function

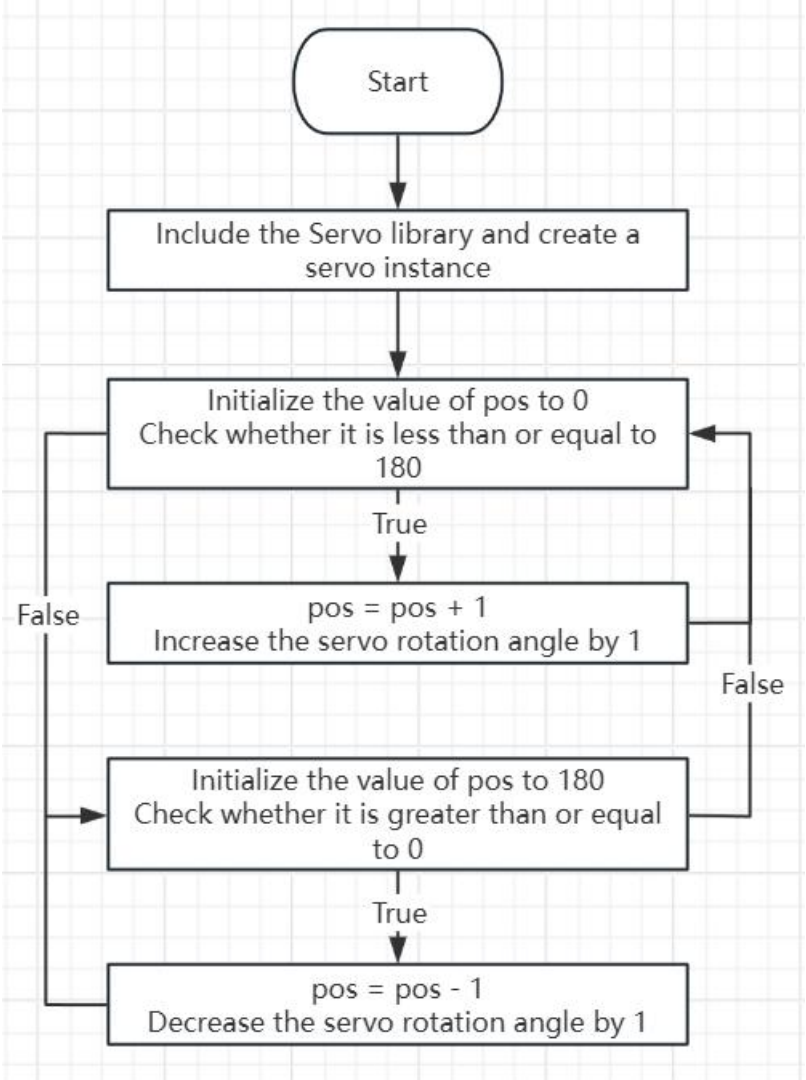
```
void loop() {
  for (pos = 0; pos <= 180; pos += 1) {
    myservo.write(pos);
    delay(15);
  }
  for (pos = 180; pos >= 0; pos -= 1) {
    myservo.write(pos);
    delay(15);
  }
}
```

for (pos = 0; pos <= 180; pos += 1): The variable pos is initialized to 0. On each loop iteration, the condition checks whether pos is still less than or equal to 180. If so, pos is incremented by 1. This causes pos to increase from 0 step by step until it reaches 181, at which point the loop exits.

myservo.write(pos): This line runs 180 times, commanding the servo to move smoothly from 0 to 180 degrees.

The following for loop works in the same way, driving the servo back from 180 degrees down to 0.

Overall Code Logic Flowchart



Program Upload Steps

This project requires additional external libraries. Refer to the "Importing Library Files" section on page 7 and import all the required external libraries at the beginning. If you have already imported them, you can skip this step.

For detailed upload instructions, please refer to the "Upload Steps" on page 8.

Key Takeaways:

#include	When you import an external library, its contents are copied directly into your program as-is before compilation.

Lesson09---Ultrasonic Sensor

Introduction

In this lesson, we'll learn how to use an ultrasonic sensor. As we all know, ultrasonic modules are commonly used for distance measurement—but have you ever wondered how they actually determine distance? By the end of this lesson, you'll understand the working principle behind ultrasonic distance sensing, and you'll use code to control the hardware to emit sound waves, receive the reflected signal, and calculate the distance based on the response.

Learning Goals

1.Understand the working principle of ultrasonic distance measurement
2.Learn how to define and call functions
3.Write and run an ultrasonic distance measurement function to enable real-time distance sensing.

Preview of the Result

Once the program is uploaded to the Arduino Nano R4, the system begins monitoring the distance ahead in real time. The ultrasonic sensor continuously measures the distance between itself and any obstacle, and the results are displayed live in the Serial Monitor.

```
Output   Serial Monitor ×
Message (Enter to send message
343     cm
342     cm
342     cm
342     cm
342     cm
342     cm
343     cm
342     cm
342     cm
342     cm
342     cm
17      cm
11      cm
21      cm
15      cm
```

Hardware Used in This Lesson

The ultrasonic module uses two main control pins: D6 and D7

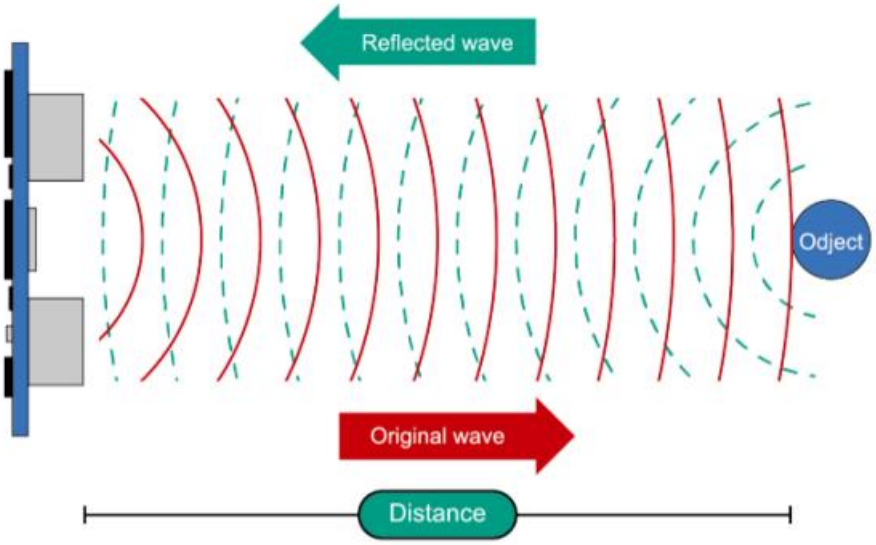
D6 (Trig) is the trigger pin, responsible for transmitting ultrasonic signals: When the program sends it a brief HIGH pulse, the module emits a burst of ultrasonic waves toward the front.

D7 (Echo) is the echo pin, which receives the ultrasonic signals reflected back from an obstacle: The module calculates the distance to the obstacle based on the time interval between when the signal is sent and when it is received.

In this lesson, we will use these two pins to handle signal transmission and reception, and apply the time-of-flight formula to calculate the actual distance between the ultrasonic sensor and the obstacle ahead.

Working Principle of an Ultrasonic Sensor Module

As shown in the diagram below, when the ultrasonic sensor is operating, it first sends out a burst of ultrasonic waves through the transmitting pin (Trig). When these sound waves hit an obstacle in front of the sensor, they are reflected back and detected by the receiving pin (Echo).



How the Ultrasonic Sensor Works:

1.Transmission Stage

The program triggers the sensor's transmit pin to send a short ultrasonic pulse and records the exact moment this pulse is emitted (the transmission time).

2.Propagation and Reflection

The ultrasonic wave travels through the air. When it hits an object in front of the sensor, it is reflected back.

3.Reception Stage

As soon as the receiving pin detects the returning echo, the program immediately records the current time (the reception time).

4.Time Difference Calculation

By calculating the difference between the transmission time and the reception time, we obtain the total round-trip travel time of the ultrasonic wave.

Distance Calculation Formula:Distance = (Speed of sound * Total travel time) / 2

It's important to note that the sound wave makes a round trip—traveling to the obstacle and back—so the result must be divided by 2 to get the actual distance to the object. The speed of sound in air is approximately 343 m/s.

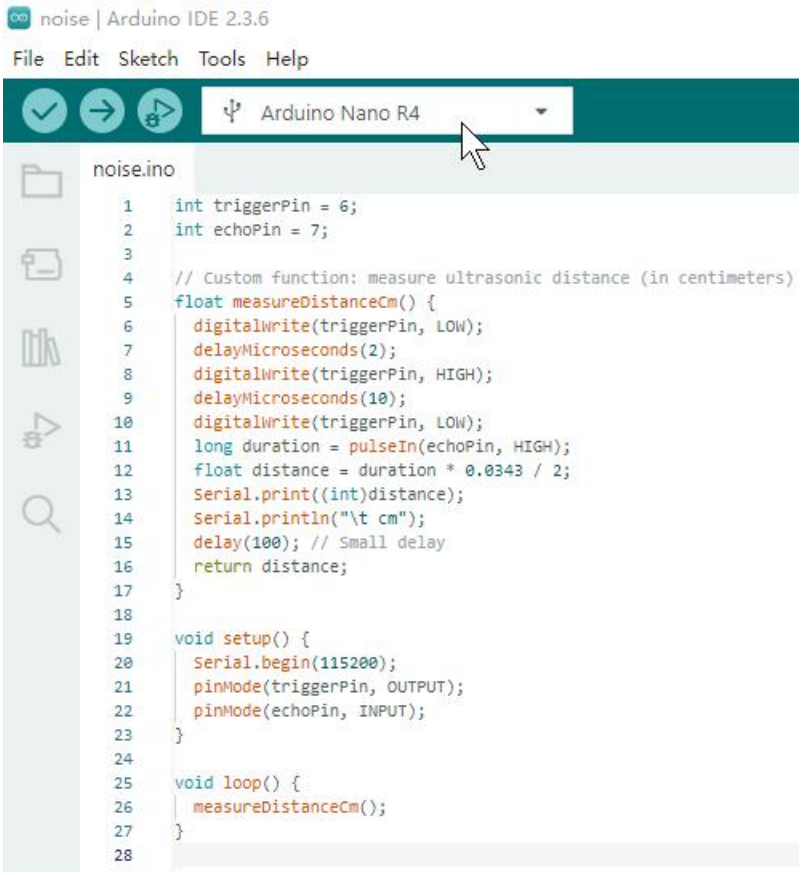
Ultrasonic Distance Measurement

Before diving into the code explanation, you can download the complete program first.Full code download link:
https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-Arduino-Nano-R4/tree/master/lesson_code/9_Ultrasonic_Sensor

Open the "9_Ultrasonic_Sensor" folder in the Arduino IDE, and then open the "9_Ultrasonic_Sensor.ino" file inside it.

Key Code Explanation

Now let's walk through the project example: **Ultrasonic Distance Measurement**



First, we define the pins required for this lesson:

```
int triggerPin = 6;
int echoPin = 7;
```

triggerPin:This is the trigger (transmit) pin, connected to D6 on the Arduino Nano R4.

echoPin:This is the echo (receive) pin, connected to D7 on the Arduino Nano R4.

Define the ultrasonic distance measurement function using float (key focus).

```
float measureDistanceCm() {
  digitalWrite(triggerPin, LOW);
  delayMicroseconds(2);
  digitalWrite(triggerPin, HIGH);
  delayMicroseconds(10);
  digitalWrite(triggerPin, LOW);
  long duration = pulseIn(echoPin, HIGH);
  float distance = duration * 0.0343 / 2;
  Serial.print((int)distance);
  Serial.println("\t cm");
  delay(100); // Small delay
  return distance;
}
```

Key Points to Understand:

From the code, we can see the following steps:

①First, the "triggerPin" is set to a LOW level and held for 2 microseconds. This ensures the pin is in a stable LOW state before sending the trigger pulse.
②Next, the "triggerPin" is set to HIGH and held for 10 microseconds. This step initiates the ultrasonic pulse. For most ultrasonic modules, 10 microseconds is the minimum required trigger pulse width.
③After the 10 microseconds, the pin is pulled LOW again to end the pulse, completing one ultrasonic transmission.
delayMicroseconds(2): delays execution for 2 microseconds
delay(2):delays execution for 2 milliseconds

long duration = pulseIn(echoPin, HIGH);
"long"can store larger integers than "int", which is why we use "long" to define the variable "duration", since we are dealing with microsecond-level numbers.
pulseIn() is an Arduino function that measures how long a pin stays at a certain voltage level. Here, we use it with "echoPin" and "HIGH", meaning it measures the duration that "echoPin" remains HIGH. After the ultrasonic module emits a pulse, "echoPin" goes HIGH, and when the reflected sound wave returns, "echoPin" goes LOW. Measuring the HIGH duration of "echoPin" thus gives the total time it takes for the ultrasonic pulse to travel to the obstacle and back.

float distance = duration * 0.0343 / 2;
float is used to define a floating-point variable (a number with decimals). Since the calculations involve decimal values, we define the variable as float. Using int or long would store only whole numbers, discarding the fractional part and losing precision. Using float preserves the decimals, making the distance measurement more accurate.
duration is the time measured for the ultrasonic pulse to travel to the obstacle and back.
0.0343 represents the speed of sound converted to centimeters per microsecond: 343 meters per second equals 0.0343 cm/μs.

return distance;
The function returns a value—the final measured distance. After calling the "measureDistanceCm" function, we can store its return value in a variable to obtain the measured distance.

Initialization Function

```
void setup() {
  Serial.begin(115200);
  pinMode(triggerPin, OUTPUT);
  pinMode(echoPin, INPUT);
}
```

Note: Here, the "triggerPin" is an output pin, while the "echoPin" is an input pin. Make sure not to mix them up when setting the pin modes.

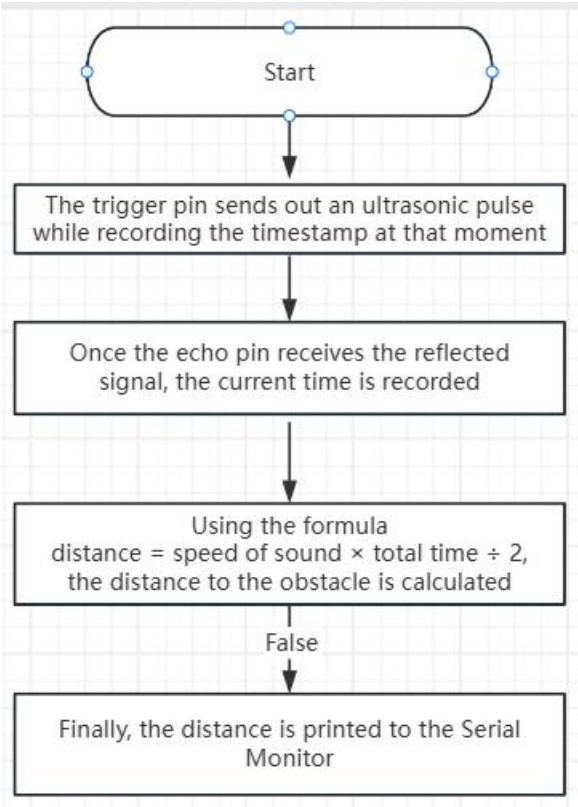
loop Function

```
void loop() {
```

```
  measureDistanceCm();
}
```

In the loop function, we only need to call the previously defined "measureDistanceCm" function to run the ultrasonic distance measurement code.

Overall Code Logic Flowchart



Program Upload Steps

For detailed upload instructions, please refer to the "Upload Steps" on page 8.

Key Takeaways:

delayMicroseconds()	Used to pause execution for a specified number of microseconds; more precise than delay()
pulseIn()	Measures the duration of a pulse on a pin; the return value is in microseconds
long	Integer type used to store large whole numbers
float	Floating-point type used to store numbers with decimals
void	Used to define a function's return type. void means the function does not return any value
float function()	Define a function that returns a floating-point value: The function name is function
return distance	Returns the value of the distance variable from within the function

Lesson10---DigitalDisplay

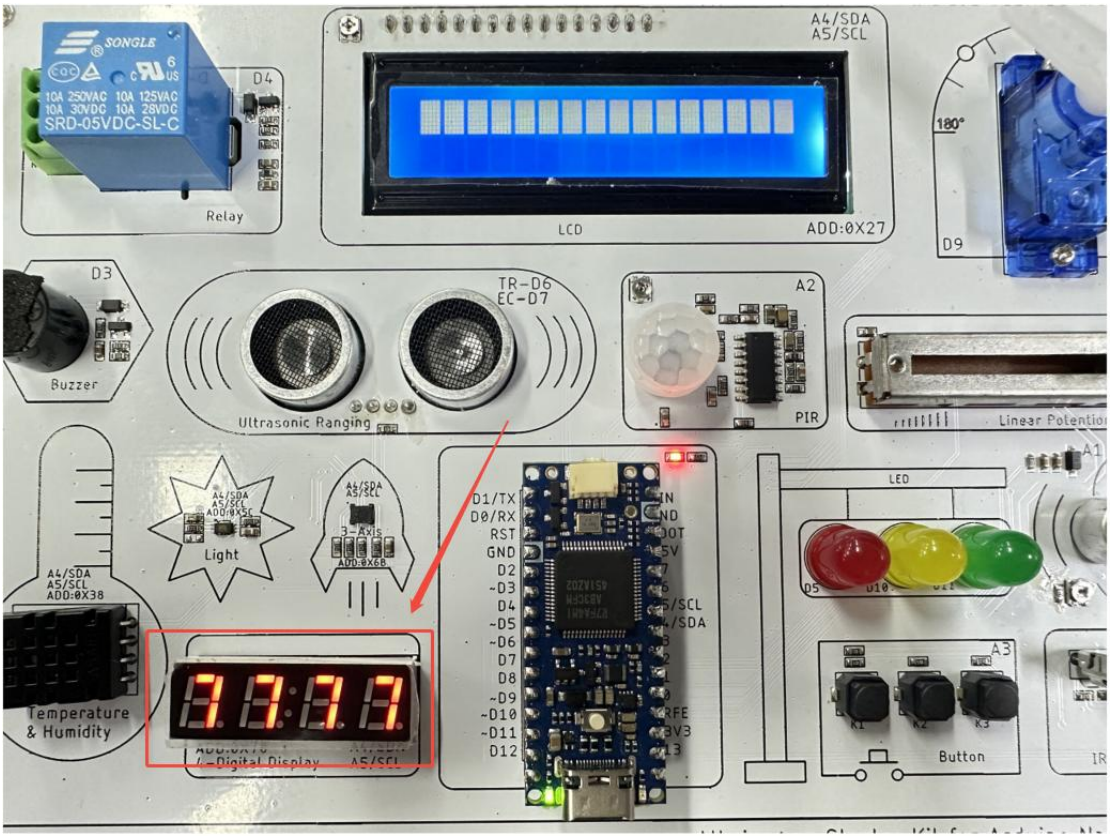
Introduction

In this lesson, we'll learn how to drive the 7-segment display on the "Arduino Nano R4" development board. We'll use the external library "HT16K33.h" to control the display and work through a comprehensive hands-on project: a 7-segment countdown timer. By the end of this lesson, you'll understand the fundamentals of displaying numbers on a 7-segment display and learn how to implement common display effects such as digit switching, blinking, and lighting up decimal points.

Learning Goals

1.Understand the working principles of a 7-segment display
2.Learn the difference between uint8_t and int
3.Understand what I2C is
4.Master multiple functions for controlling a 7-segment display
5.Complete the 7-segment countdown project

Preview of the Result



After the program runs, the 7-segment display initializes first, then starts counting down from

8888 to 1111.

Hardware Used in This Lesson

The 7-segment display is located at the lower-left corner of the development board. It is connected to the board via I2C, with an address of 0x70.

I2C is a synchronous serial bus protocol used for communication between integrated circuits. It uses one clock line (SCL) and one data line (SDA) to transmit data. The master device generates the clock signal and controls the communication process, while slave devices only communicate after being selected by the master through their addresses. Multiple devices can share the same I2C bus and are distinguished by their unique addresses, allowing multi-device communication with very few pins.

Why use I2C? What are its advantages?

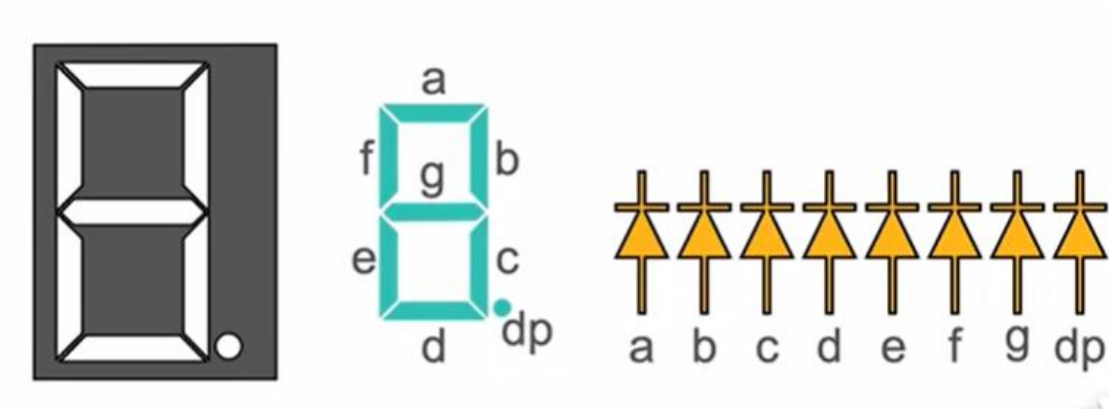
When we have many peripheral devices but a limited number of available pins on the development board, I2C becomes an ideal solution. I2C communication requires only two pins (SDA and SCL) to communicate with multiple devices. By assigning a different address to each device, the development board can communicate with several modules on the same bus. This ability to control multiple devices using minimal pins is the biggest advantage of I2C.

I2C Address

So how does I2C distinguish between different devices? On an I2C bus, each device is assigned a unique I2C address. When multiple devices are connected to the same bus, these addresses are what allow the system to tell them apart. During initialization, we typically define and initialize the device address along with the corresponding object. After that, Arduino can communicate accurately with a specific device by using its I2C address.

Working Principle of an DigitalDisplay Module

A 7-segment display is a display device made up of seven illuminated segments (a, b, c, d, e, f, g) plus one decimal point (dp). The "8"-shaped figure shown in the diagram is formed by combining these seven segments. Each segment is an independent light-emitting unit (such as an LED) that can be controlled individually—turned on or off as needed.



Working Logic: “Light specific segments to form numbers”

A 7-segment display shows different numbers by turning on specific segments to form the shape of each digit.

To display "8"： turn on all seven segments — a, b, c, d, e, f, g.

To display "0"： turn on a, b, c, d, e, f, and leave segment g off.

To display "1"： turn on only segments b and c.

To display a decimal point： simply turn on the "dp" segment.

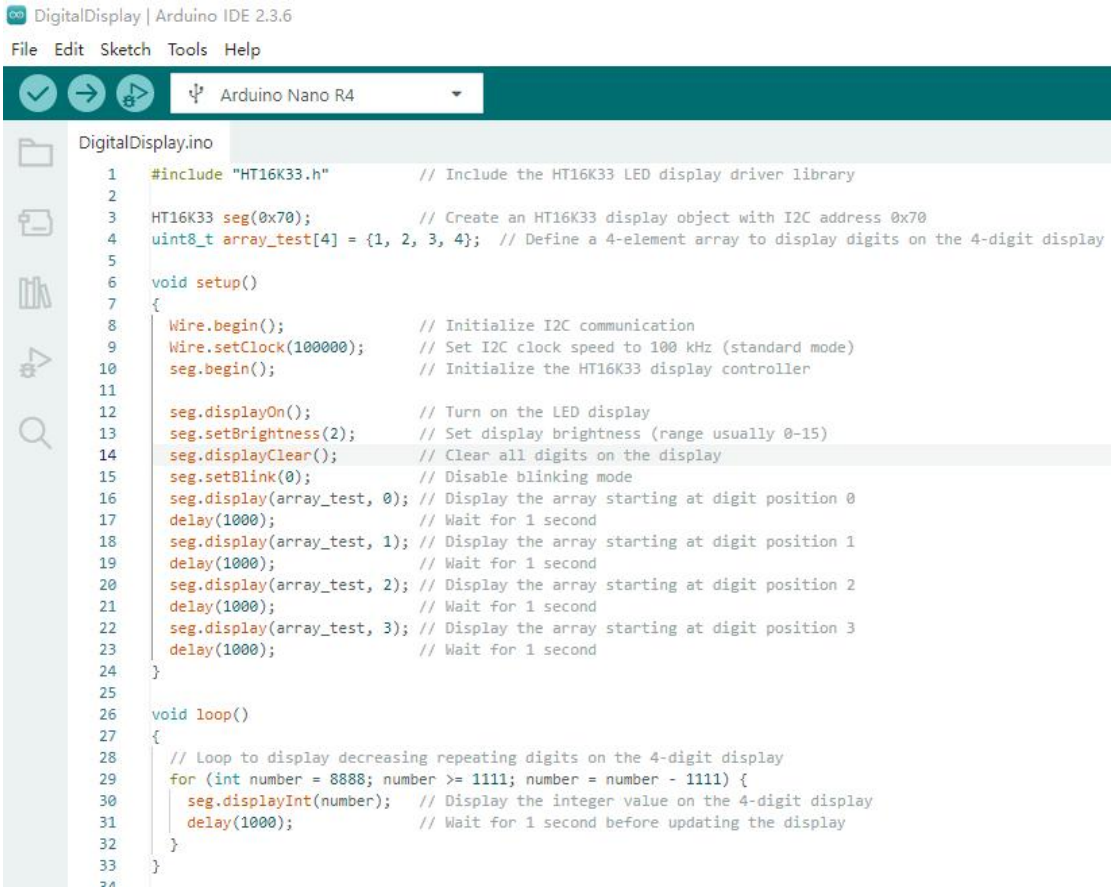
7-Segment Display Countdown

Before we dive into the code explanation, you can download the complete program first.Full code download link:
https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-Arduino-Nano-R4/tree/master/lesson_code/10_DigitalDisplay

Open the "10_DigitalDisplay" folder in the Arduino IDE, then open the "10_DigitalDisplay.ino" file inside it.

Key Code Explanation

Now let's walk through the project example: 7-Segment Display Countdown.




```ino
#include "HT16K33.h"          // Include the HT16K33 LED display driver library

HT16K33 seg(0x70);            // Create an HT16K33 display object with I2C address 0x70
uint8_t array_test[4] = {1, 2, 3, 4};  // Define a 4-element array to display digits on the 4-digit display

void setup()
{
  Wire.begin();               // Initialize I2C communication
  Wire.setClock(100000);      // Set I2C clock speed to 100 kHz (standard mode)
  seg.begin();                // Initialize the HT16K33 display controller

  seg.displayOn();            // Turn on the LED display
  seg.setBrightness(2);       // Set display brightness (range usually 0-15)
  seg.displayClear();         // Clear all digits on the display
  seg.setBlink(0);            // Disable blinking mode
  seg.display(array_test, 0); // Display the array starting at digit position 0
  delay(1000);                // Wait for 1 second
  seg.display(array_test, 1); // Display the array starting at digit position 1
  delay(1000);                // Wait for 1 second
  seg.display(array_test, 2); // Display the array starting at digit position 2
  delay(1000);                // Wait for 1 second
  seg.display(array_test, 3); // Display the array starting at digit position 3
  delay(1000);                // Wait for 1 second
}

void loop()
{
  // Loop to display decreasing repeating digits on the 4-digit display
  for (int number = 8888; number >= 1111; number = number - 1111) {
    seg.displayInt(number);   // Display the integer value on the 4-digit display
    delay(1000);              // Wait for 1 second before updating the display
  }
}
```

First, we import the 7-segment display driver library:

```
#include "HT16K33.h"
```

The "HT16K33.h" display driver library encapsulates the low-level I2C communication with the "HT16K33" chip and is the core file for this lesson. All the methods used in the code to control the 7-segment display are provided by this library.

Create the display object

```
HT16K33 seg(0x70);
```

Create an object named "seg" to control the "HT16K33" 7-segment display with the I2C address "0x70". This tells the Arduino IDE that whenever we use the name "seg", we are operating the "HT16K33" display located at address "0x70".

Define the test array

```
uint8_t array_test[4] = {1, 2, 3, 4};
```

Here, we need to clearly understand the difference between "uint8_t" and "int":

1.Value Range

"**uint8_t**":0-255

"**int**":-32768-32767

2.Memory Usage

"uint8_t" uses less memory than **"int"**.

3.Sign Difference

"uint8_t" is unsigned, meaning it can only represent positive values.

"**int**" is signed and can represent both positive and negative values.
When the values are small and never negative, "uint8_t" is a good choice and is commonly used for hardware-related data.
When the values can be larger and may include both positive and negative numbers, "int" is typically used for numerical calculations.
Here, we define an array specifically for controlling the 7-segment display. Since each element only needs to represent values from **0 to 9**, we use the **"uint8_t"** type. This fully meets the required value range while allowing the underlying library functions to operate efficiently and directly—mainly because the **"display"** function only accepts data in the **"uint8_t"** format.

Initialization Function

```
void setup()
{
  Serial.begin(115200);
  Wire.begin();
  Wire.setClock(100000);
```

Wire.begin():Initializes the Arduino I2C bus. This must be called before using "HT16K33".
Wire.setClock(100000):Sets the I2C clock speed to 100 kHz.
These two lines are used to initialize the I2C bus. After that, we proceed to initialize the 7-segment display driver.

Initialize the 7-segment display module so it's ready for subsequent use

```
  seg.begin();
```

Turn on the 7-segment display

```
  seg.displayOn();
```

Set the display brightness to 2. The brightness level can be set from 0 to 10

```
  seg.setBrightness(2);
```

Clear the previous display buffer before starting a new display cycle

```
  seg.displayClear();
```

Set the blink mode to no blinking. If set to 1, the display will blink.

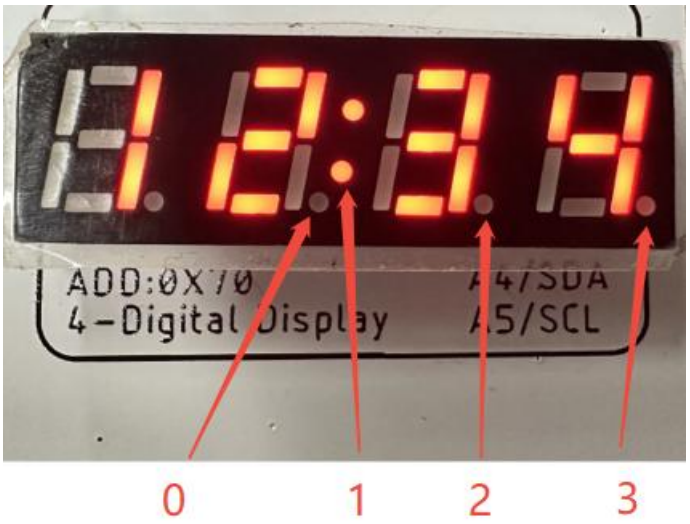
```
  seg.setBlink(0);
```

Test the data in the array, including special symbols such as quotation marks on the 7-segment display. (Important)

```
  seg.display(array_test, 0);
  delay(1000);
  seg.display(array_test, 1);
  delay(1000);
  seg.display(array_test, 2);
  delay(1000);
```

```
  seg.display(array_test, 3);
  delay(1000);
}
```

Here, only the last parameter is different. The earlier parameters all specify displaying the data from the "array_test" array, while the final parameter determines which symbol on the 7-segment display is shown. Values from 0 to 3 correspond to different punctuation marks, as illustrated below.



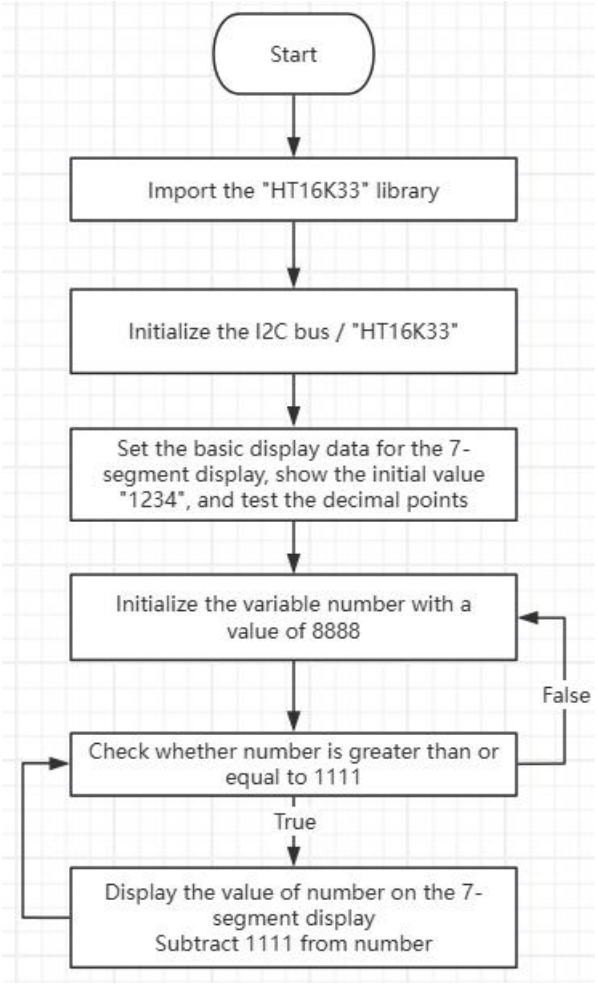
Note: The leftmost punctuation mark is not used.

Main loop function

```
void loop()
{
  // Loop to display decreasing repeating digits on the 4-digit display
  for (int number = 8888; number >= 1111; number = number - 1111) {
    seg.displayInt(number);
    delay(1000);
  }
}
```

In the main loop, we use a for loop and the "displayInt" method to count down from 8888 to 1111. The "displayInt" method allows an "int" value as its parameter, which makes it convenient to decrement the value by 1111 on each iteration of the loop.

Overall Code Logic Flowchart



Program Upload Steps

This project requires additional external libraries. Refer to the "Importing Library Files" section on page 7 and import all the required external libraries at the beginning. If you have already import ed them, you can skip this step.

For detailed upload instructions, please refer to the "Upload Steps" on page 8.

Key Takeaways:

Wire.begin()	Initialize the I2C bus
seg.displayOn()	Turn on the 7-segment display
seg.setBrightness(2)	Set the display brightness to 2
seg.displayClear()	Clear the display buffer
seg.setBlink(0)	Set the display to non-blinking mode
seg.display(array,0)	Display the preset array values on the 7-segment display, showing punctuation mark index 0
seg.displayInt(8888)	Display the integer value 8888 on the 7-segment display

Lesson11---LCD

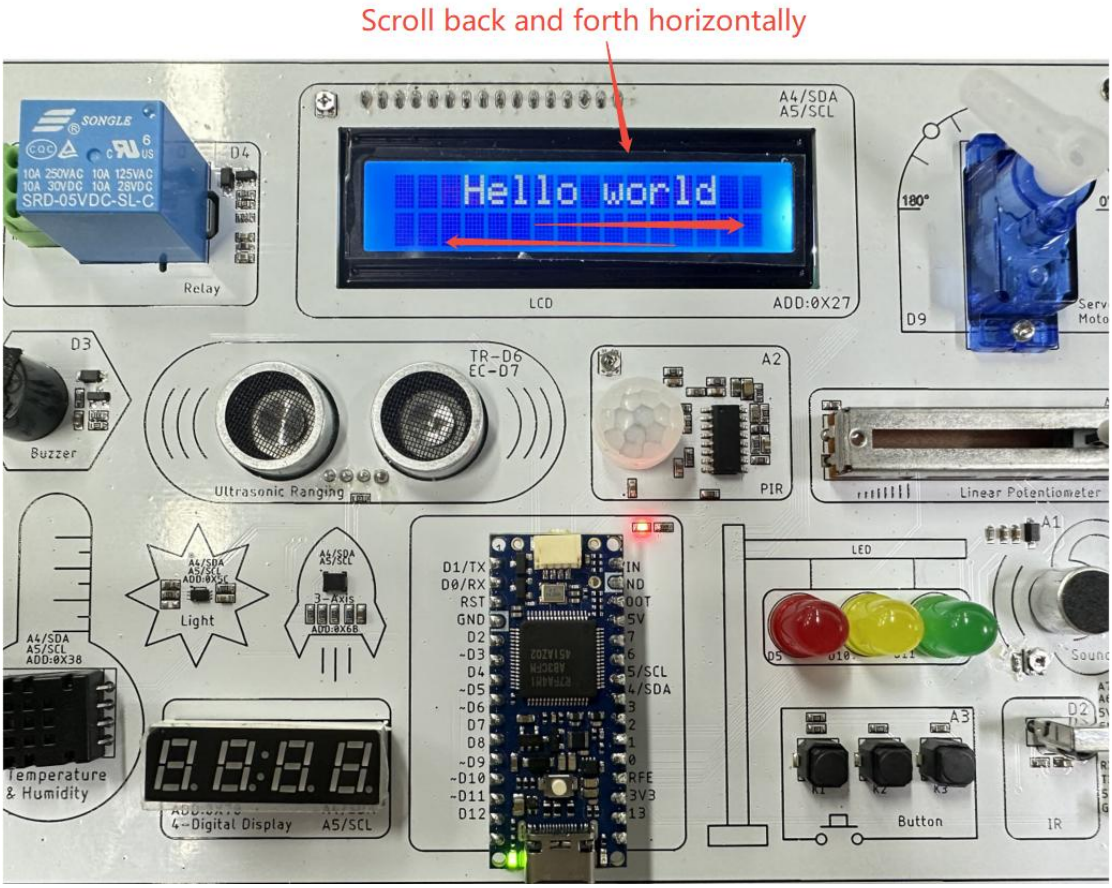
Introduction

In this lesson, we'll learn how to drive the LCD display on the "Arduino Nano R4" development board. We'll use the for loop you learned earlier to make text scroll left and right across the LCD screen.By the end of this lesson, you'll understand how to control an LCD display and become familiar with several common LCD control techniques.

Learning Goals

1.Understand how an LCD display works
2.Master multiple methods for controlling an LCD display
3.Complete an experiment that makes "Hello World" scroll left and right on the LCD screen

Preview of the Result



After the program runs, "Hello World" will scroll back and forth across the LCD screen.

Hardware Used in This Lesson

The LCD display is located at the top of the development board and is connected via the board's I2C interface, with an address of 0x27.

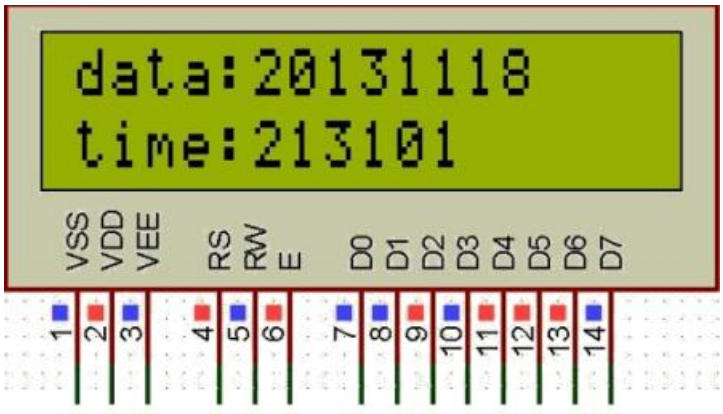
In the upper-left corner of the LCD display, there is an adjustable knob. **This knob is mainly used to adjust the contrast of the screen.**

Turning the knob clockwise (forward rotation) gradually reduces the contrast. The characters will become lighter and may eventually disappear, making the screen look blank.
Turning the knob counterclockwise (reverse rotation) increases the contrast. The characters will appear darker, but if you keep turning it, the screen may show solid blocks instead of readable text.

Working Principle of an LCD Module

The LCD display on the Arduino Nano R4 is an I2C-controlled module. Here's how it works: the microcontroller sends data to an I/O expander chip over the I2C bus using just two lines, SCL and SDA. The expander converts the serial I2C data into parallel signals that drive the LCD's RS, RW, E, and data lines D4 – D7.The LCD controller then uses these signals to distinguish between

commands and character data, writes the character codes into its internal memory, and ultimately drives the liquid crystal pixels to change their light transmission. With the backlight on, the corresponding characters are displayed on the screen. This design allows a 1602 LCD to be controlled using far fewer I/O pins.

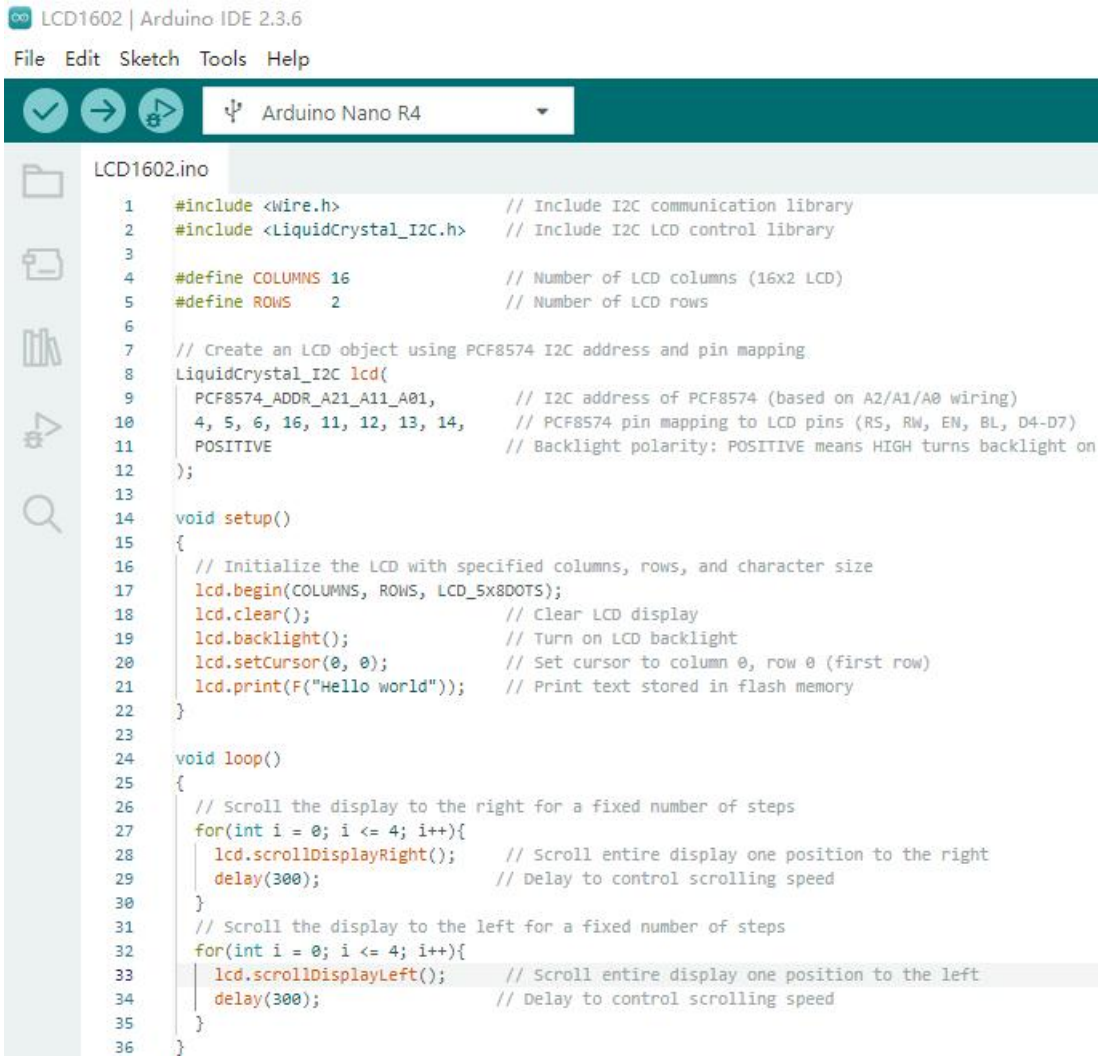


LCD Text Scrolling

Before diving into the code, you can download the program first. The complete code is available at the following link:
https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-Arduino-Nano-R4/tree/master/lesson_code/11_LCD
Open the "11_LCD.ino" file located in the "11_LCD" folder using the Arduino IDE.

Key Code Explanation

Now, let's walk through the example: LCD text scrolling.



First, we include the required header files. In this lesson, we'll continue using I2C, so we need to import both the I2C communication library and the LCD utility library.

```
#include <Wire.h>
#include <LiquidCrystal_I2C.h>
```

The "LiquidCrystal_I2C.h" library is specifically designed for character-based LCDs that use an I2C interface.

Next, we define the number of columns and rows for the LCD

```
#define COLUMNS 16
#define ROWS     2
```

"COLUMNS" and "ROWS" represent the LCD's dimensions—16 columns by 2 rows.

Create the LCD object (this step is critical)

```
LiquidCrystal_I2C lcd(
  PCF8574_ADDR_A21_A11_A01,
  4, 5, 6, 16, 11, 12, 13, 14,
  POSITIVE
);
```

This section of code is extremely important, because all subsequent LCD operations are performed through it. Let's go through the parameters one by one:

PCF8574_ADDR_A21_A11_A01: This represents the I2C address of the PCF8574 chip. You can also write it directly as 0x27. The macro "PCF8574_ADDR_A21_A11_A01" is already defined as 0x27 in the "LiquidCrystal_I2C.h" file.

(4, 5, 6, 16, 11, 12, 13, 14):These values map the expander's pins to the LCD's control and data pins. This mapping must match the actual hardware wiring; otherwise, the LCD will not display correctly. On the Arduino Nano R4, this pin mapping is fixed and must not be changed.

POSITIVE:This turns the backlight on. If it is set to "NEGATIVE", the backlight will be turned off.

Initialization Function

```
void setup()
{
  lcd.begin(COLUMNS, ROWS, LCD_5x8DOTS);
  lcd.clear();                    // Clear LCD display
  lcd.backlight();                // Turn on LCD backlight
  lcd.setCursor(0, 0);            // Set cursor to column 0, row 0 (first row)
  lcd.print(F("Hello world"));    // Print text stored in flash memory
}
```

In the initialization function, we configure the LCD so it's ready to display text properly.

lcd.begin(): Informs the LCD that the screen size is 16 × 2 characters, with a 5 × 8 dot matrix per character.

lcd.clear(): Clears any leftover content from power-up; otherwise, random or garbled characters may appear.

lcd.backlight(): Turns on the LCD backlight. Without it, the text may not be visible even if it's being displayed.

lcd.setCursor(0,0): Sets the cursor position, starting at column 0, row 0.

lcd.print(F("Hello world")): Displays the string "Hello world" on the LCD. The "F" macro stores the string in Flash memory, so it does not consume SRAM.

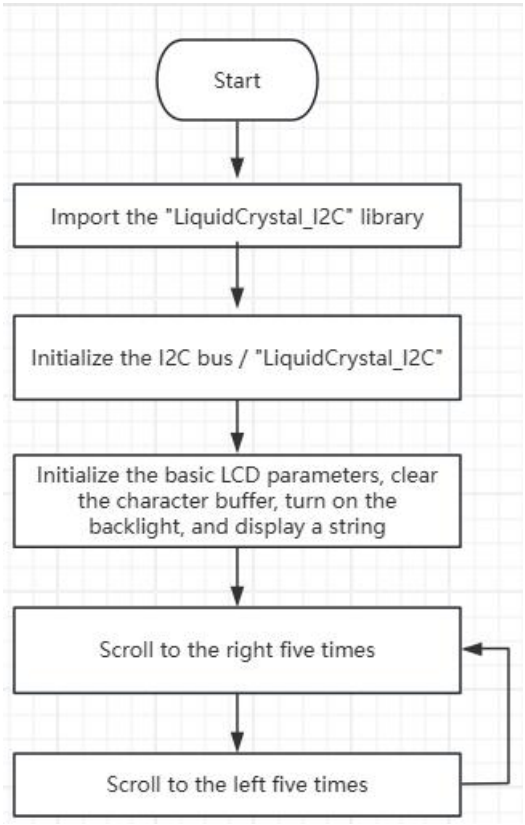
Loop Function

```
void loop()
{
  for(int i = 0; i <= 4; i++){
    lcd.scrollDisplayRight();     // Scroll entire display one position to the right
    delay(300);                   // Delay to control scrolling speed
  }
  for(int i = 0; i <= 4; i++){
    lcd.scrollDisplayLeft();      // Scroll entire display one position to the left
    delay(300);                   // Delay to control scrolling speed
  }
}
```

The main loop is responsible for creating the left-and-right scrolling effect. Each direction runs for 5 iterations, because "Hello world" contains 11 characters and the LCD has 16 columns. This

means the text will hit the screen edges after scrolling 5 positions to the left or right.

Overall Code Logic Flowchart



Program Upload Steps

This project requires additional external libraries. Refer to the "Importing Library Files" section on page 7 and import all the required external libraries at the beginning. If you have already imported them, you can skip this step.

For detailed upload instructions, please refer to the "Upload Steps" on page 8.

Key Takeaways:

lcd.clear()	Clear the LCD buffer
lcd.backlight()	Turn on the LCD backlight
lcd.setCursor(0, 0)	Set the cursor to column 0, row 0
lcd.print()	Display the string on the LCD screen
lcd.scrollDisplayRight()	Scroll the text to the right
lcd.scrollDisplayLeft()	Scroll the text to the left

Lesson12---6-Axis Module

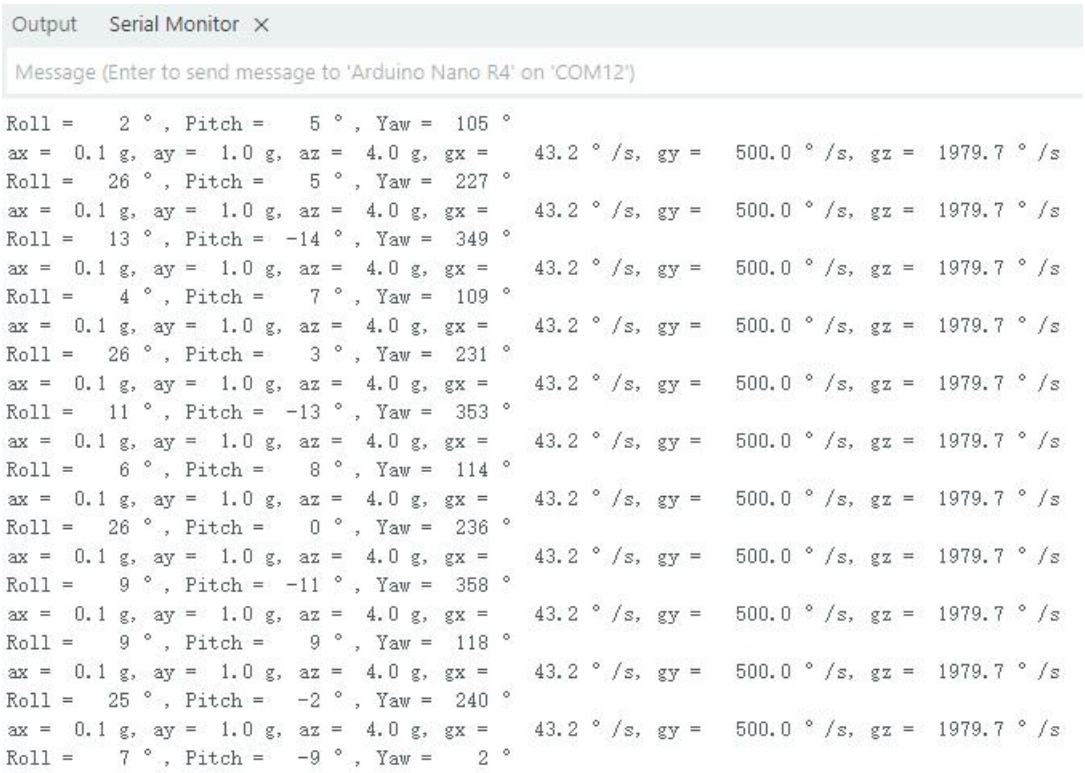
Introduction

In this lesson, we'll learn how to work with the six-axis sensor on the "Arduino Nano R4" development board. Using the external libraries "Arduino_LSM6DS3" and "MadgwickAHRS", we will read data from the accelerometer and gyroscope, fuse the sensor data, and output real-time orientation angles. By the end of this lesson, you'll be able to use a six-axis sensor at a basic level, understand the fundamentals of attitude estimation, and implement periodic sampling and serial output of orientation data through code.

Learning Goals

1.Understand the working principles of a six-axis sensor
2.Become proficient in the various uses of "Serial.print"
3.Acquire raw acceleration and angular velocity data, and use the Madgwick algorithm to compute and output orientation angles

Preview of the Result



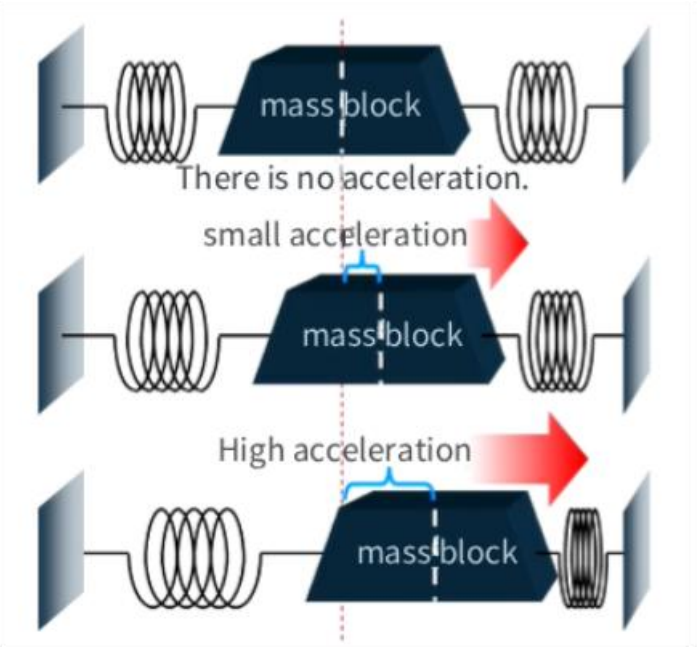
After running the program and opening the Serial Monitor, you'll be able to see the current acceleration, angular velocity, and the calculated orientation angle data.

Hardware Used in This Lesson

The 6-axis sensor is located at the lower-left corner of the development board. It is connected via the board's I2C interface and uses the address 0x6B.

Working Principle of an 6-Axis Module

The figure below illustrates the working principle of a "single-axis accelerometer" inside a six-axis sensor. It relies on an internal proof mass and springs: when the sensor accelerates, the inertia of the mass causes the springs to deform. The difference in deformation corresponds to the magnitude of the acceleration (with no acceleration, the deformation is the same; with small acceleration, the difference is small; with larger acceleration, the difference becomes greater). A six-axis sensor contains three such accelerometer axes—one each along the X, Y, and Z directions (all based on the same principle shown in the figure, just oriented differently)—plus three gyroscope axes that measure angular velocity. Together, this multi-axis structure enables the detection of three-dimensional acceleration and angular velocity.

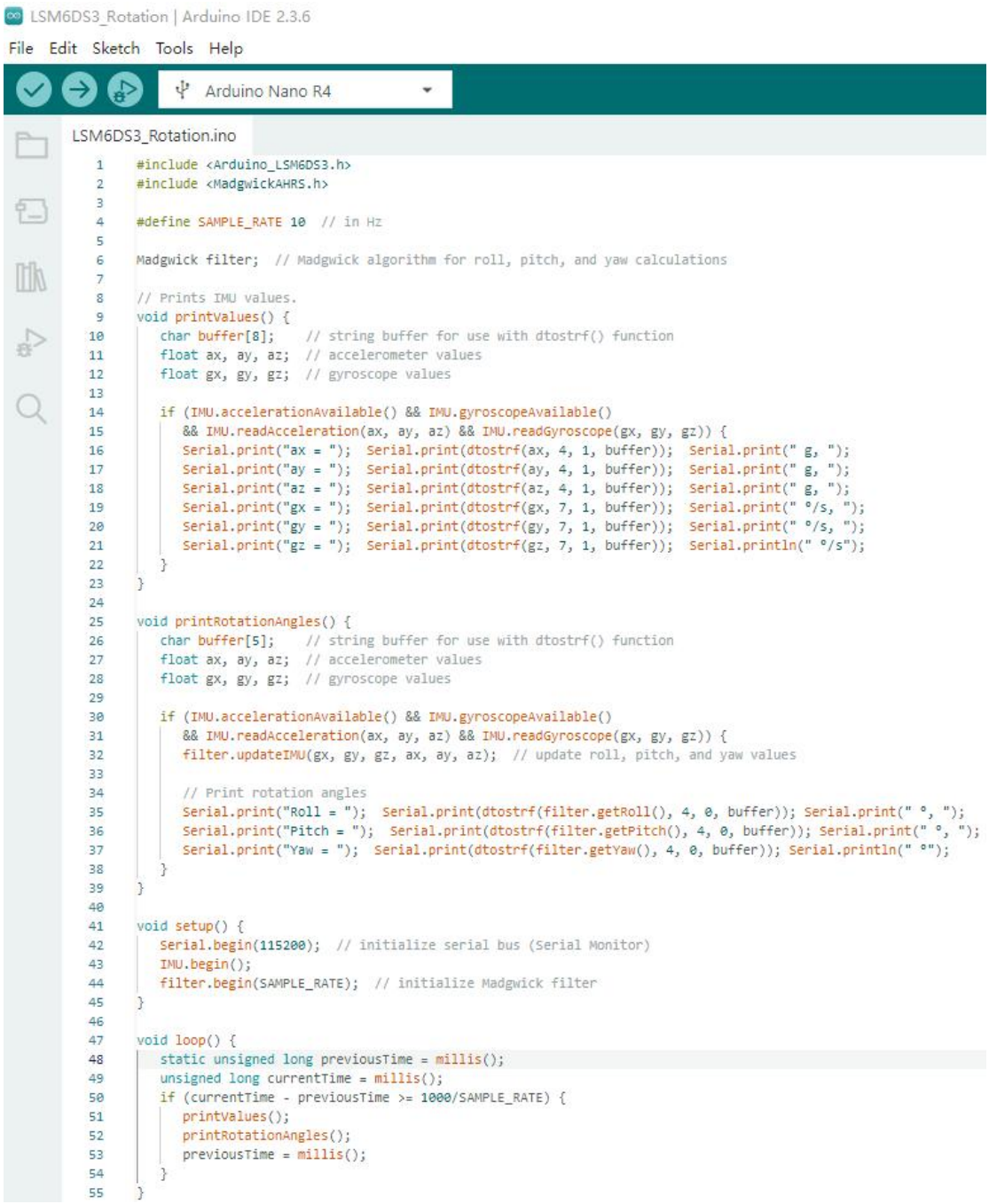


Acquiring and Processing Data from the 6-Axis Sensor

Before diving into the code walkthrough, you can download the complete source code first.Full code download link:
https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-Arduino-Nano-R4/tree/master/lesson_code/12_6-Axis
Open the "12_6-Axis.ino" file located in the "12_6-Axis" folder using the Arduino IDE.

Key Code Explanation

Now let's walk through the example: acquiring and processing data from the 6-axis sensor.




```ino
#include <Arduino_LSM6DS3.h>
#include <MadgwickAHRS.h>

#define SAMPLE_RATE 10  // in Hz

Madgwick filter;  // Madgwick algorithm for roll, pitch, and yaw calculations

// Prints IMU values.
void printValues() {
  char buffer[8];    // string buffer for use with dtostrf() function
  float ax, ay, az;  // accelerometer values
  float gx, gy, gz;  // gyroscope values

  if (IMU.accelerationAvailable() && IMU.gyroscopeAvailable()
    && IMU.readAcceleration(ax, ay, az) && IMU.readGyroscope(gx, gy, gz)) {
    Serial.print("ax = ");  Serial.print(dtostrf(ax, 4, 1, buffer));  Serial.print(" g, ");
    Serial.print("ay = ");  Serial.print(dtostrf(ay, 4, 1, buffer));  Serial.print(" g, ");
    Serial.print("az = ");  Serial.print(dtostrf(az, 4, 1, buffer));  Serial.print(" g, ");
    Serial.print("gx = ");  Serial.print(dtostrf(gx, 7, 1, buffer));  Serial.print(" °/s, ");
    Serial.print("gy = ");  Serial.print(dtostrf(gy, 7, 1, buffer));  Serial.print(" °/s, ");
    Serial.print("gz = ");  Serial.print(dtostrf(gz, 7, 1, buffer));  Serial.println(" °/s");
  }
}

void printRotationAngles() {
  char buffer[5];    // string buffer for use with dtostrf() function
  float ax, ay, az;  // accelerometer values
  float gx, gy, gz;  // gyroscope values

  if (IMU.accelerationAvailable() && IMU.gyroscopeAvailable()
    && IMU.readAcceleration(ax, ay, az) && IMU.readGyroscope(gx, gy, gz)) {
    filter.updateIMU(gx, gy, gz, ax, ay, az);  // update roll, pitch, and yaw values

    // Print rotation angles
    Serial.print("Roll = ");  Serial.print(dtostrf(filter.getRoll(), 4, 0, buffer)); Serial.print(" °, ");
    Serial.print("Pitch = ");  Serial.print(dtostrf(filter.getPitch(), 4, 0, buffer)); Serial.print(" °, ");
    Serial.print("Yaw = ");  Serial.print(dtostrf(filter.getYaw(), 4, 0, buffer)); Serial.println(" °");
  }
}

void setup() {
  Serial.begin(115200);  // initialize serial bus (Serial Monitor)
  IMU.begin();
  filter.begin(SAMPLE_RATE);  // initialize Madgwick filter
}

void loop() {
  static unsigned long previousTime = millis();
  unsigned long currentTime = millis();
  if (currentTime - previousTime >= 1000/SAMPLE_RATE) {
    printValues();
    printRotationAngles();
    previousTime = millis();
  }
}
```

First, we import Arduino's official IMU library, which is used to drive the "LSM6DS3" sensor—also known as the "6-Axis" sensor. We then include the "MadgwickAHRS.h" library to implement the attitude estimation algorithm.

```
#include <Arduino_LSM6DS3.h>
#include <MadgwickAHRS.h>
```

Next, we define a sampling frequency,which is required later for calculating the orientation angles.

```
#define SAMPLE_RATE 10   // in Hz
```

Create a "Madgwick" filter object.

```
Madgwick filter;
```

Create the "filter" object,which we'll use later to feed in "IMU" data and retrieve the orientation angle results.

Define the function "printValues". Its purpose is straightforward: to print the raw "IMU" data so we can easily view it in the Serial Monitor.

```
void printValues() {
  char buffer[8];     // string buffer for use with dtostrf() function
  float ax, ay, az;   // accelerometer values
  float gx, gy, gz;   // gyroscope values
```

First, we declare the variables. Here, "buffer" is defined as a "char" string and is mainly used to store text. The six-axis sensor data is declared as "float", since these values include decimal points.

```
  if (IMU.accelerationAvailable() && IMU.gyroscopeAvailable()
      && IMU.readAcceleration(ax, ay, az) && IMU.readGyroscope(gx, gy, gz)) {
      Serial.print("ax = ");   Serial.print(dtostrf(ax, 4, 1, buffer));   Serial.print(" g, ");
      Serial.print("ay = ");   Serial.print(dtostrf(ay, 4, 1, buffer));   Serial.print(" g, ");
      Serial.print("az = ");   Serial.print(dtostrf(az, 4, 1, buffer));   Serial.print(" g, ");
      Serial.print("gx = ");   Serial.print(dtostrf(gx, 7, 1, buffer));   Serial.print(" °/s, ");
      Serial.print("gy = ");   Serial.print(dtostrf(gy, 7, 1, buffer));   Serial.print(" °/s, ");
      Serial.print("gz = ");   Serial.print(dtostrf(gz, 7, 1, buffer));   Serial.println(" °/s");
  }
}
```

First, we use an if condition to check whether the data is ready. The sensor requires an initialization period, so while it's not ready, we avoid printing any data. Once the data is ready, we use "Serial.print" to output the values to the Serial Monitor.

Serial.print(""):directly prints the string inside the quotation marks.
Serial.print(dtostrf(ax, 4, 1, buffer)):uses "dtostrf()" to control the width and decimal precision of the data. The 4 means the number occupies at least four characters in total, and 1 means one digit is kept after the decimal point. "buffer" is where the formatted result is stored, which was already defined earlier in the code.

Next, we define a second function, "printRotationAngles". The purpose of this function is to use the "Madgwick" algorithm to calculate and output orientation angles based on the "IMU" data.

```
void printRotationAngles() {
  char buffer[5];     // string buffer for use with dtostrf() function
  float ax, ay, az;   // accelerometer values
  float gx, gy, gz;   // gyroscope values
  if (IMU.accelerationAvailable() && IMU.gyroscopeAvailable()
      && IMU.readAcceleration(ax, ay, az) && IMU.readGyroscope(gx, gy, gz)) {
      filter.updateIMU(gx, gy, gz, ax, ay, az);   // update roll, pitch, and yaw values
      // Print rotation angles
```

```
        Serial.print("Roll = ");   Serial.print(dtostrf(filter.getRoll(), 4, 0, buffer)); Serial.print(" °, ");
        Serial.print("Pitch = ");   Serial.print(dtostrf(filter.getPitch(), 4, 0, buffer)); Serial.print(" °, ");
        Serial.print("Yaw = ");   Serial.print(dtostrf(filter.getYaw(), 4, 0, buffer)); Serial.println(" °");
    }
}
```

The output method is similar to that used in the "printValues" function, so we won't go into detail again here.

filter.getRoll():gets the left – right tilt (roll)

filter.getPitch():gets the forward – backward tilt (pitch)

filter.getYaw():gets the horizontal rotation (yaw)

Initialization function

```
void setup() {
    Serial.begin(115200);   // initialize serial bus (Serial Monitor)
    IMU.begin();
    filter.begin(SAMPLE_RATE);   // initialize Madgwick filter
}
```

Set the serial baud rate to 115200,make sure to select the same baud rate when opening the Serial Monitor.

Initialize the "IMU" to start the LSM6DS3 sensor.

Initialize the Madgwick filter,passing in the previously defined sampling frequency; otherwise, the orientation calculation will be inaccurate.

Loop function

```
void loop() {
    static unsigned long previousTime = millis();
    unsigned long currentTime = millis();
    if (currentTime - previousTime >= 1000/SAMPLE_RATE) {
        printValues();
        printRotationAngles();
        previousTime = millis();
    }
}
```

This section of code uses a "software timer" approach to run tasks at fixed intervals without blocking the program.

unsigned long:an unsigned long integer type. The value returned by "millis()" is a large number and can only be positive, so this type is required.

static:ensures the variable is initialized only once and retains its value across multiple function calls. Normally, the "loop()" function runs continuously, and variables would be reassigned every time the loop starts. If that happened, "previousTime" would always be set to the current time, and the timing condition would never be met. By using "static", the variable is initialized only the first time "loop()" runs and keeps its value afterward.

currentTime:this variable is updated every time the loop runs, allowing us to know the current

system time.

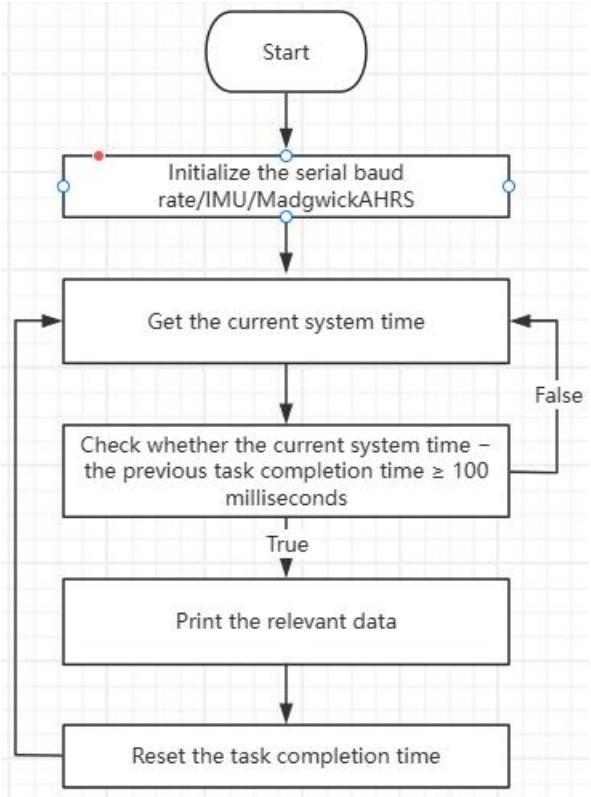
currentTime - previousTime:the difference between these two values represents how much time has elapsed.

1000/SAMPLE_RATE:1000 milliseconds divided by the sampling rate (how many samples are taken per second) gives the time interval for each sample.

When the current time minus the last sampling time is greater than or equal to the defined sampling interval, the two functions below are executed, and the relevant data is printed to the Serial Monitor.

previousTime = millis():after the task finishes, the current system time is assigned to "previousTime" to prepare for the next timing check.

Overall Code Logic Flowchart



Program Upload Steps

This project requires additional external libraries. Refer to the "Importing Library Files" section on page 7 and import all the required external libraries at the beginning. If you have already import ed them, you can skip this step.

For detailed upload instructions, please refer to the "Upload Steps" on page 8.

Key Takeaways:

char	Declare the data type as a string
millis()	Current system time

static	This code runs only once within the loop function and is used to obtain the current system time for timing and condition checks

Lesson13---Light_Sensor

Introduction

In this lesson, we will learn how to use the "Light Sensor" on the "Arduino Nano R4" development board. We will continue using the I2C interface to drive the sensor, read ambient light intensity data from the "Light Sensor", and build a simple simulation of a smart streetlight system—automatically turning the light on at night and off during the day based on the surrounding light conditions.

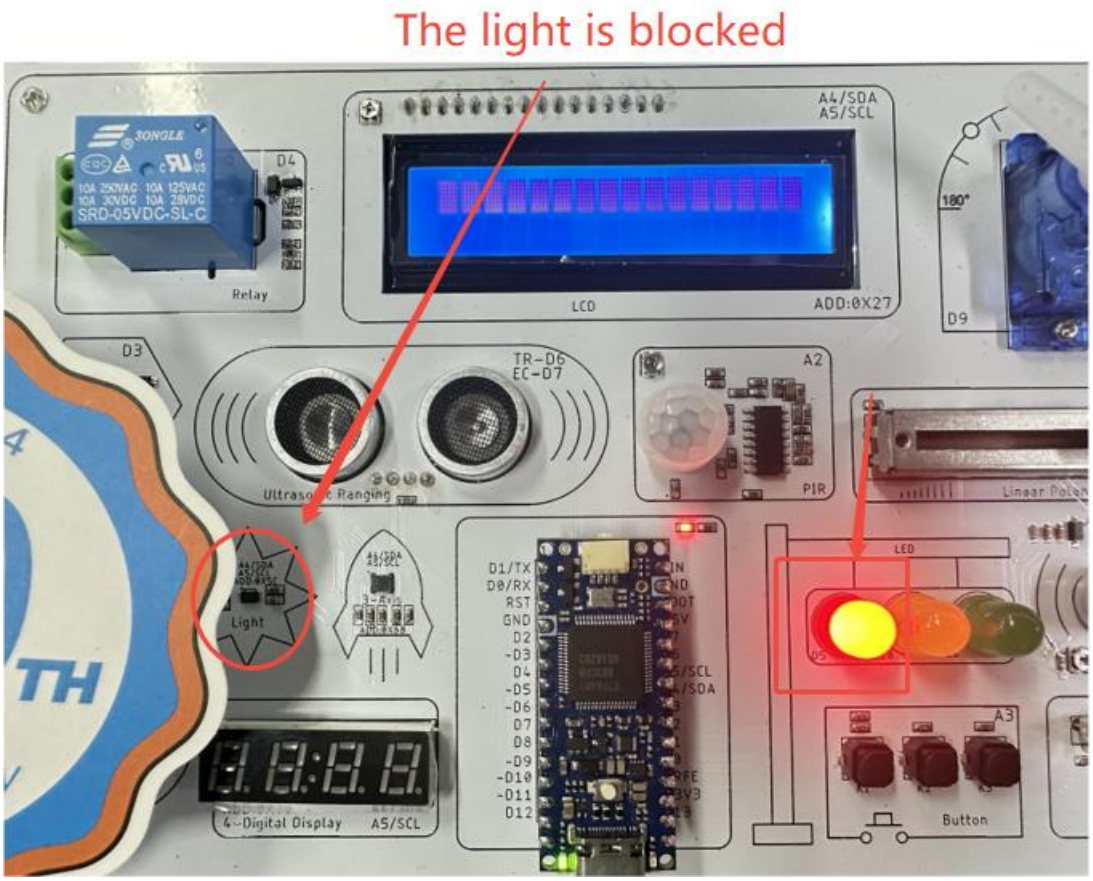
Learning Goals

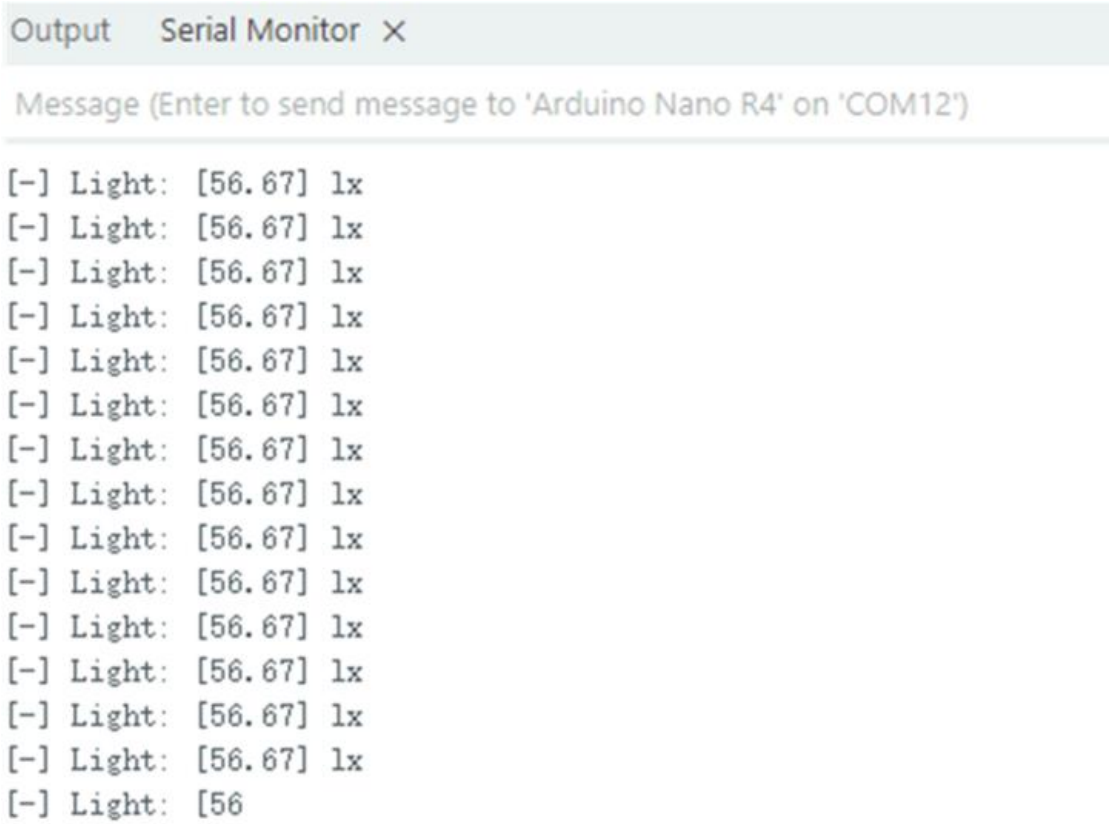
1.Understand how the "Light Sensor" works

2.Review how to use if – else statements

3.Complete a simulated streetlight experiment that reads ambient light intensity and determines whether the light should turn on

Preview of the Result





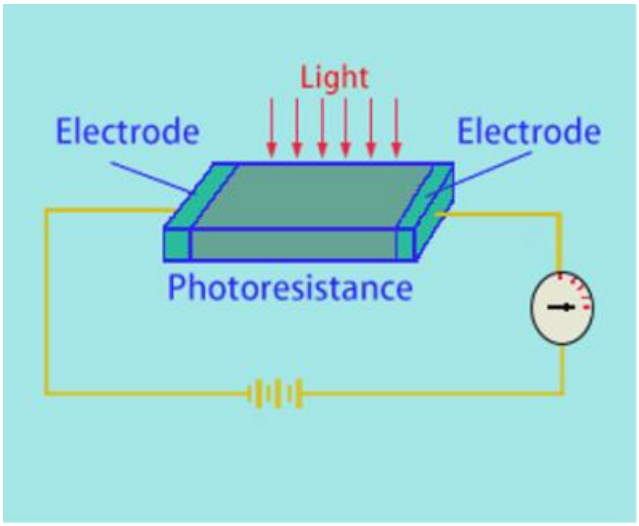
After the program runs, if you block the light reaching the "Light Sensor" and open the Serial Monitor, you will see that the current ambient light intensity value is below 100. The red LED on the Arduino Nano R4 board will turn on. If the ambient light intensity value is greater than 100, the red LED will turn off.

Hardware Used in This Lesson

The "Light Sensor" is located in the lower-left corner of the development board. It is connected to the board via I2C, with an address of 0x5C.

Working Principle of an Light Sensor Module

As shown in the image below, the core component of the light sensor is the photoresistor in the center. When light shines on the surface of the photoresistor, its resistance changes with the light intensity—the stronger the light, the lower the resistance. When the photoresistor is connected in a circuit with a power supply and an ammeter, the current changes accordingly as the resistance changes (lower resistance results in higher current, while higher resistance results in lower current). In this way, the original light signal is converted into an electrical signal in the circuit. The change in the ammeter's reading reflects the change in light intensity. This is the basic working principle behind how a light sensor "uses light to control electricity".

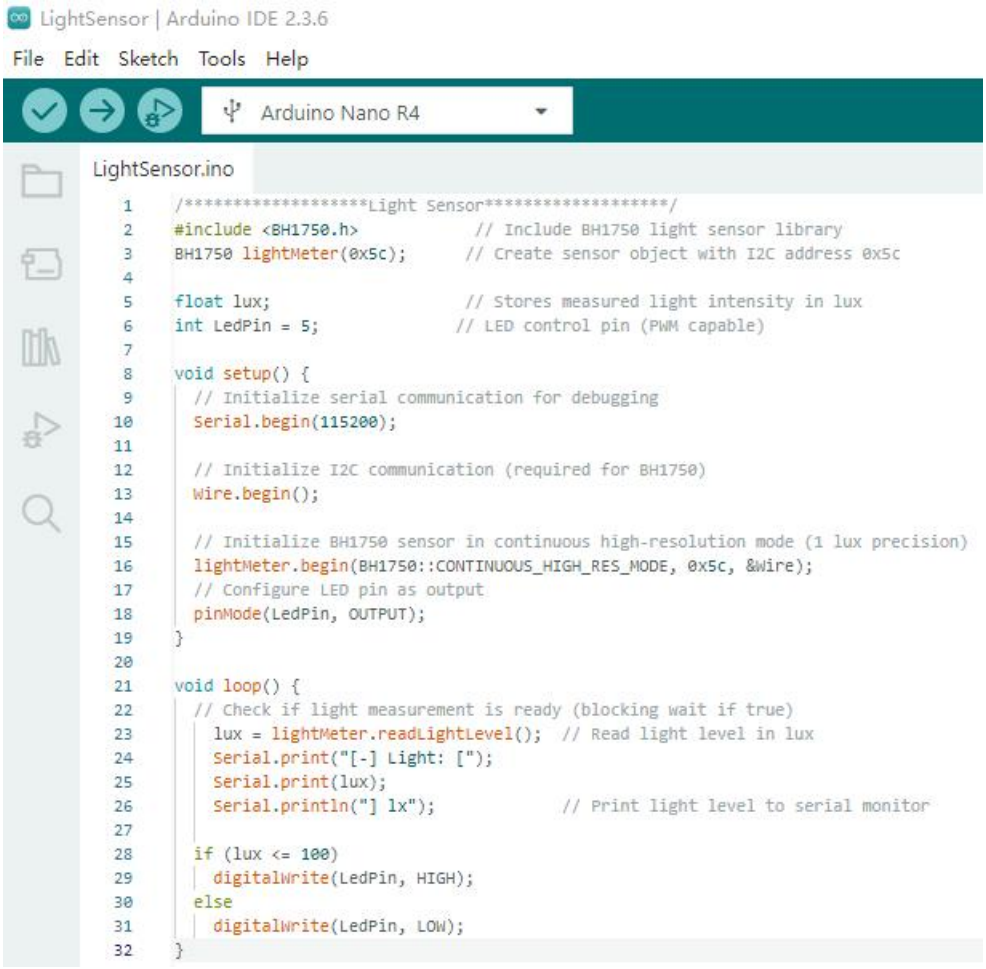


Smart Streetlight

Before we go over the code, you can download it first.Complete code download link:
https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-Arduino-Nano-R4/tree/master/lesson_code/13_Light_Sensor
Open the "13_Light_Sensor.ino" file located in the "13_Light_Sensor" folder using the Arduino IDE.

Key Code Explanation

Now let's walk through the example: Smart Streetlight.



First, import the "BH1750" sensor library used to drive the "Light Sensor".

```
#include <BH1750.h>
```

Create a BH1750 sensor object.

```
BH1750 lightMeter(0x5c);
```

Specify the I2C address of the "Light Sensor" as "0x5C", and create a "BH1750" sensor object named "lightMeter".

Define a light intensity variable named "lux" to store the ambient light level.

```
float lux;
```

We define this variable using float so that the decimal values are preserved, preventing any loss of precision.

Define the control pin for the LED.

```
int LedPin = 5;
```

Initialize the function.

```
void setup() {
  Serial.begin(115200);
  Wire.begin();
```

```
  lightMeter.begin(BH1750::CONTINUOUS_HIGH_RES_MODE, 0x5c, &Wire);
  pinMode(LedPin, OUTPUT);
}
```

Set the serial baud rate to 115200. We also need to open the Serial Monitor to view the returned light intensity values.

Wire.begin():initializes I2C communication, which we covered in detail in earlier lessons.

lightMeter.begin:initializes the "BH1750" sensor.

BH1750::CONTINUOUS_HIGH_RES_MODE:sets the sensor to continuous high-resolution measurement mode. In this mode, the sensor operates continuously and does not need to be repeatedly woken up.

pinMode():configures the pin mode—here, it sets the LED pin as an output.

The loop function

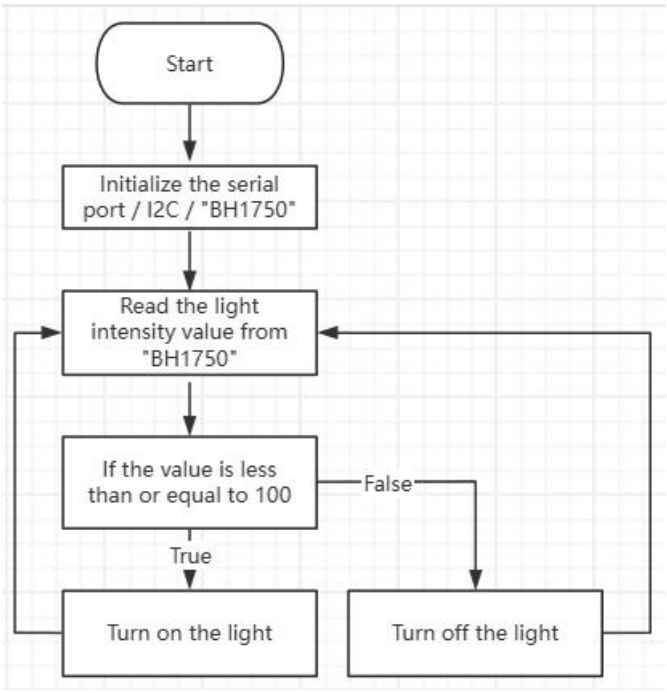
```
void loop() {
  // Check if light measurement is ready (blocking wait if true)
    lux = lightMeter.readLightLevel();   // Read light level in lux
    Serial.print("[-] Light: [");
    Serial.print(lux);
    Serial.println("] lx");                  // Print light level to serial monitor

  if (lux <= 100)
    digitalWrite(LedPin, HIGH);
  else
    digitalWrite(LedPin, LOW);
}
```

lightMeter.readLightLevel():reads the current light level from the BH1750 sensor and assigns the returned value to the "lux" variable, making it easy to print and evaluate later.

Serial.print():Serial.print() is used to output data to the Serial Monitor; we won’t go into detail here, as its main purpose is simply to display the values.

if - else:If the value of "lux" is less than or equal to 100, the LED is driven HIGH and turns on; otherwise, it is set LOW and turns off.

Overall Code Logic Flowchart



Program Upload Steps

This project requires additional external libraries. Refer to the "Importing Library Files" section on page 7 and import all the required external libraries at the beginning. If you have already imported them, you can skip this step.

For detailed upload instructions, please refer to the "Upload Steps" on page 8.

Key Takeaways:

readLightLevel()	This is a method provided by the BH1750 library, and its primary purpose is to read the ambient light intensity value.

Lesson14---Tem&Hum

Introduction

In this lesson, we will learn how to use the temperature and humidity sensor on the "Arduino Nano R4" development board. You will learn how to use the DHT20 library and its built-in methods to read temperature and humidity data and display the values in the Serial Monitor.

Learning Goals

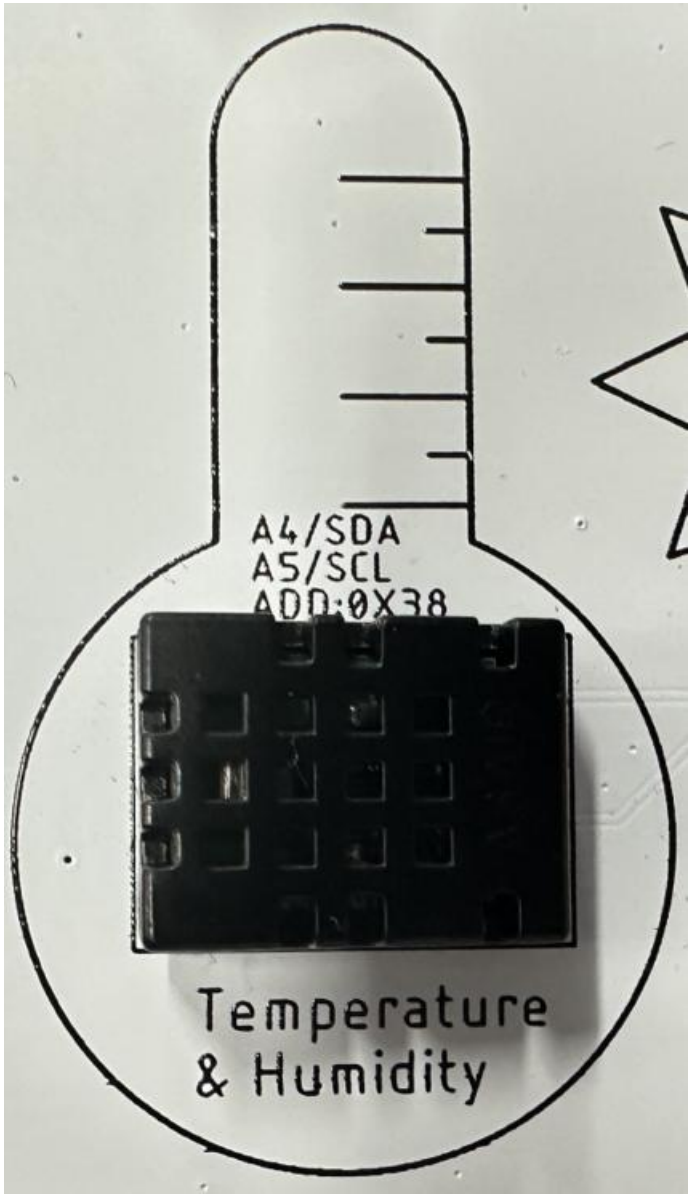
1.Understand how the temperature and humidity sensor works
2.Complete an experiment that reads temperature and humidity data and prints the values to the Serial Monitor

Preview of the Result

```
Temperature: 27.8 ° C   Humidity: 55.3 %
Temperature: 27.8 ° C   Humidity: 55.2 %
Temperature: 27.8 ° C   Humidity: 55.2 %
Temperature: 27.8 ° C   Humidity: 55.2 %
Temperature: 27.8 ° C   Humidity: 55.2 %
Temperature: 27.7 ° C   Humidity: 55.1 %
Temperature: 27.7 ° C   Humidity: 55.1 %
Temperature: 27.8 ° C   Humidity: 55.1 %
Temperature: 27.7 ° C   Humidity: 55.0 %
Temperature: 27.7 ° C   Humidity: 55.0 %
Temperature: 27.7 ° C   Humidity: 55.1 %
Temperature: 27.7 ° C   Humidity: 55.1 %
Temperature: 27.7 ° C   Humidity: 55.0 %
Temperature: 27.7 ° C   Humidity: 55.0 %
Temperature: 27.8 ° C   Humidity: 55.0 %
```

After running the program and opening the Serial Monitor, you will be able to see the current temperature and humidity readings returned by the sensor.

Hardware Used in This Lesson

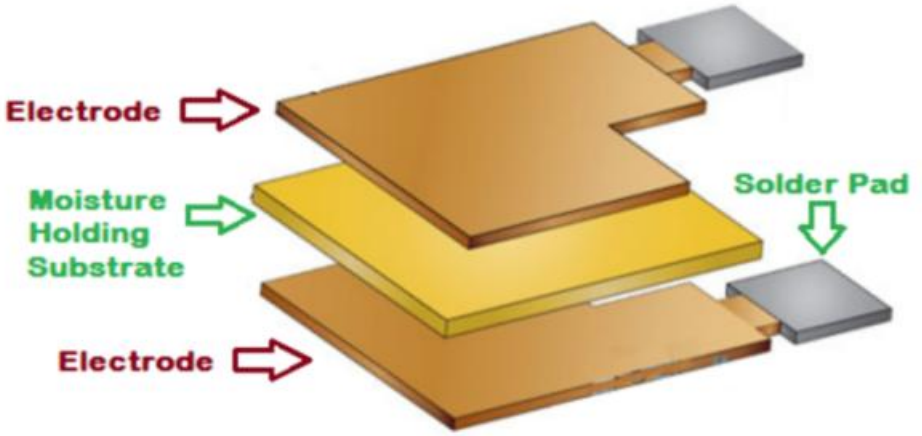


The temperature and humidity sensor is located in the lower-left corner of the development board. It is connected to the board via I2C, with an address of 0x38.

Working Principle of an Temperature and Humidity Sensor Module

The diagram below shows the structure of the temperature and humidity sensor's sensing unit. Its core consists of upper and lower electrodes with a hygroscopic dielectric substrate in between, while the pads provide electrical connections between the electrodes and the external circuit. During humidity measurement, water vapor in the air is absorbed or released by the dielectric substrate. This process changes the substrate's dielectric constant, which in turn alters the capacitance formed by the electrodes and the substrate, enabling the conversion of humidity into an electrical signal. At the same time, an integrated temperature-sensitive element measures the ambient temperature. It outputs temperature data independently and also applies temperature

compensation to correct humidity measurement errors. Finally, accurate temperature and humidity electrical signals are produced through the signal conditioning circuitry.

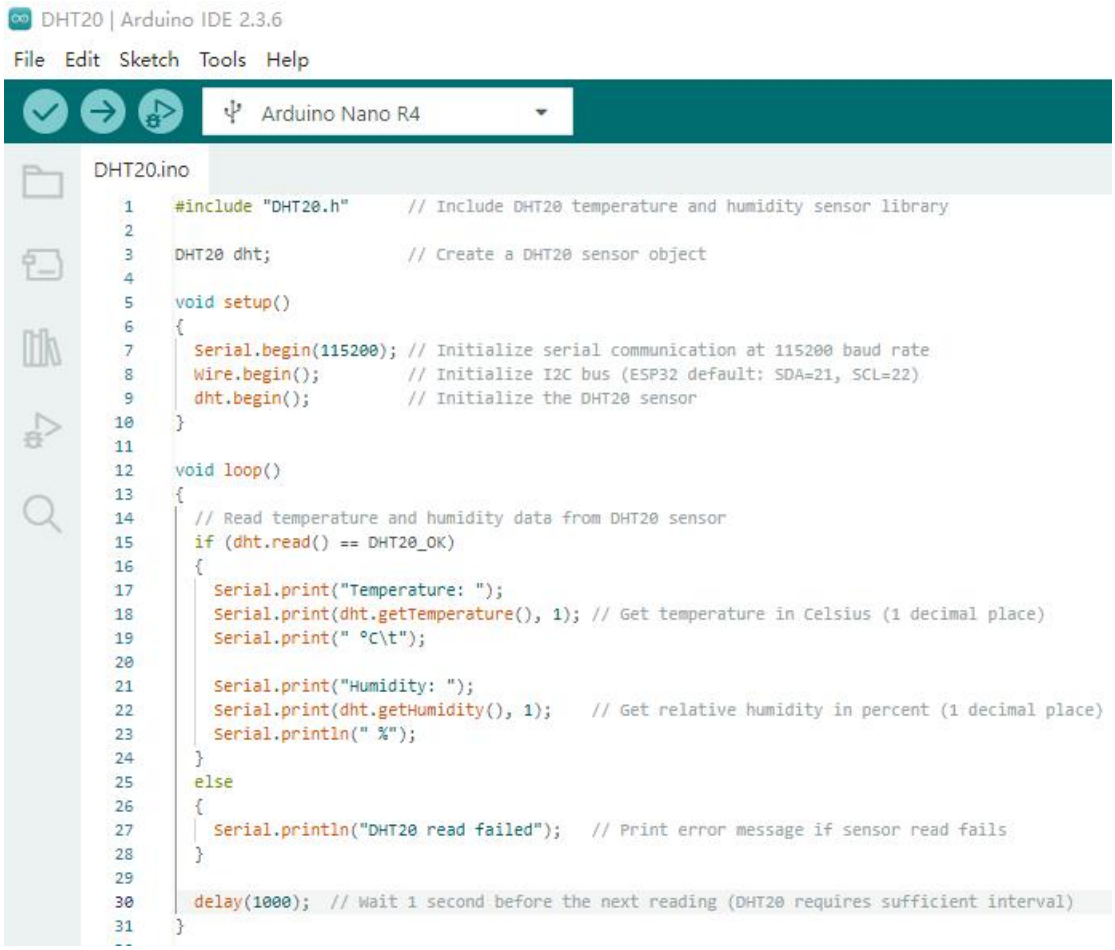


Temperature and Humidity Sensor Data Acquisition

Before going through the code, you can download it first.Complete code download link:
https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-Arduino-Nano-R4/tree/master/lesson_code/14_Tem%26Hum
Use the Arduino IDE to open the "14_Tem&Hum.ino" file located in the "14_Tem&Hum" folder.

Key Code Explanation

Now let's walk through the example: Temperature and Humidity Sensor Data Acquisition.



First, import the "DHT20" library used to drive the temperature and humidity sensor.

```
#include "DHT20.h"
```

Next, create a DHT20 class object named "dht".

```
DHT20 dht;
```

This sensor instance "dht" is used to control and read data from the "DHT20" sensor.

Initialization function

```
void setup()
{
  Serial.begin(115200);
  Wire.begin();
  dht.begin();
}
```

In this function, we do three things: initialize the serial baud rate, initialize the I2C bus, and initialize the temperature and humidity sensor.

Loop Function

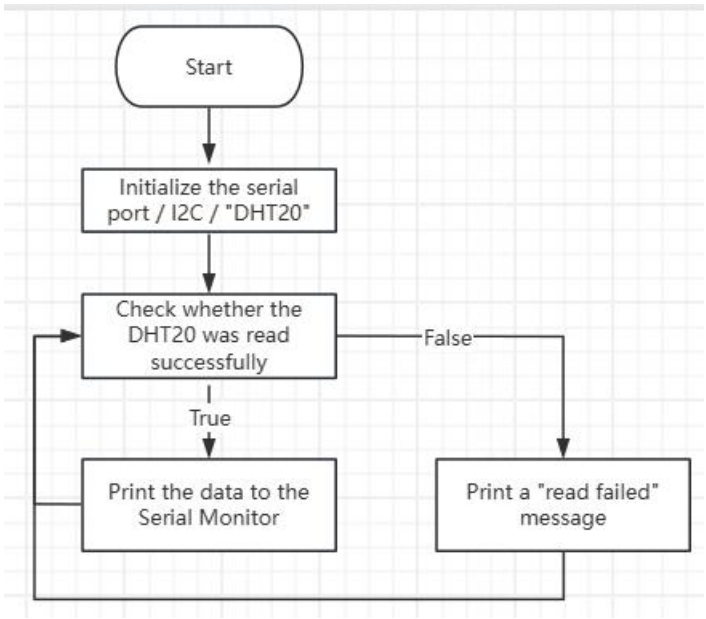
```
void loop()
{
  if (dht.read() == DHT20_OK)
```

Inside the loop function, we first use an if statement to check whether the data was read successfully. This step is very important because it both updates the sensor data and verifies that the read operation was successful.

```
  {
    Serial.print("Temperature: ");
    Serial.print(dht.getTemperature(), 1); // Get temperature in Celsius (1 decimal place)
    Serial.print("  °  C\t");

    Serial.print("Humidity: ");
    Serial.print(dht.getHumidity(), 1);      // Get relative humidity in percent (1 decimal place)
    Serial.println(" %");
  }
  else
  {
    Serial.println("DHT20 read failed");     // Print error message if sensor read fails
  }
  delay(1000);   // Wait 1 second before the next reading (DHT20 requires sufficient interval)
}
```

Print the temperature and humidity data read from the sensor to the Serial Monitor. If the read fails, print "DHT20 read failed".

Overall Code Logic Flowchart



Program Upload Steps

This project requires additional external libraries. Refer to the "Importing Library Files" section on page 7 and import all the required external libraries at the beginning. If you have already imported them, you can skip this step.

For detailed upload instructions, please refer to the "Upload Steps" on page 8.

Key Takeaways:

dht.getTemperature()	Read the temperature data from the DHT20 sensor.
dht.getHumidity()	Read the humidity data from the DHT20 sensor.

Lesson15---IR Module

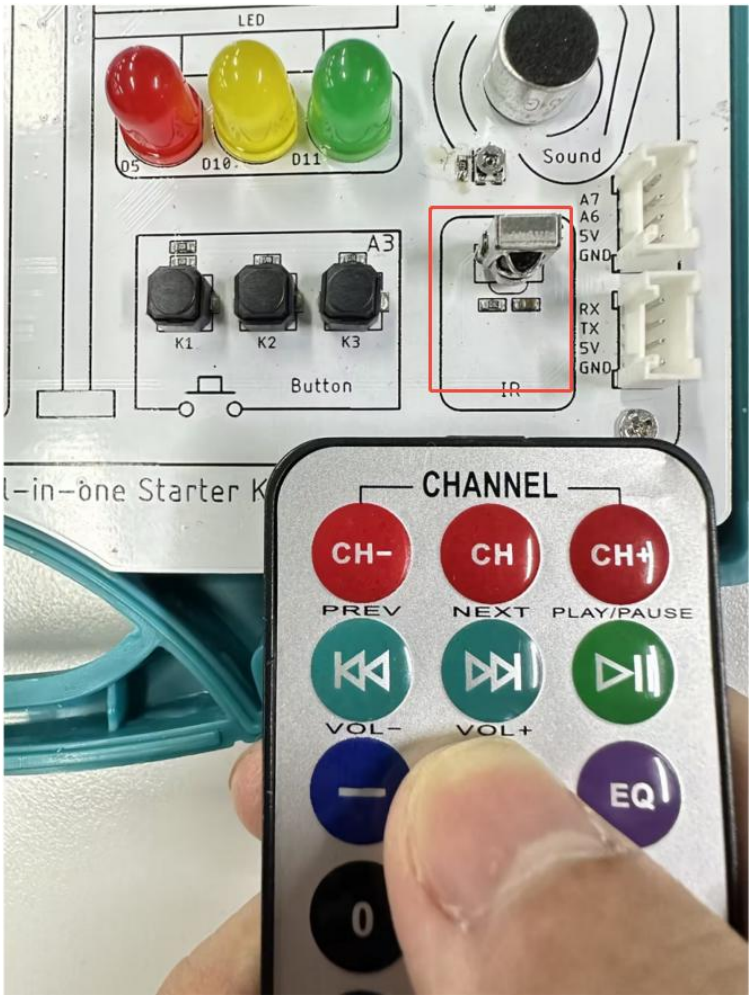
Introduction

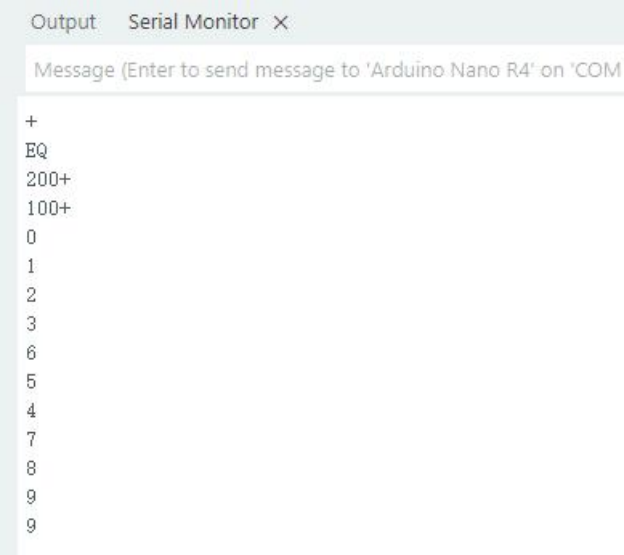
In this lesson, we'll learn how to use the "IR Module" on the Arduino Nano R4 development board. Throughout the course, we'll use a "switch-case" statement to handle different button signals from a remote control and complete a hands-on project that prints the button values to the Serial Monitor. By working through this example, you'll gain a clear understanding of how to use the "switch-case" structure to efficiently and logically handle multiple conditions when dealing with many possible input options.

Learning Goals

1.Understand how the "IR Module" works
2.Learn when and how to use the "switch-case" statement
3.Complete a project that prints button values to the Serial Monitor

Preview of the Result





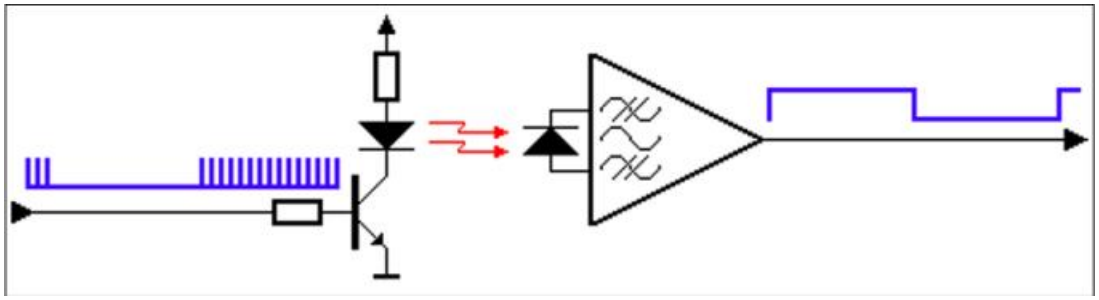
After the program is running, when you press a button on the remote control, the Serial Monitor will display the corresponding button information. Note: Make sure the remote is pointed toward the "IR Module", and check that the batteries in the remote are properly installed.

Hardware Used in This Lesson

The "IR Module" is located at the bottom-right corner of the development board and is connected to the D2 pin.

Working Principle of an IR Module

As shown in the diagram below, the working principle of an infrared remote control is as follows: the encoded electrical signal sent by the remote control drives the transistor and infrared LED in the transmitter circuit, converting the electrical signal into infrared light pulses. These infrared signals are then captured by the infrared receiver, converted back into electrical signals, and processed by an amplifier circuit. Finally, the original control signal is reconstructed, enabling remote control of the device.



Printing Remote Button Information

Before diving into the code explanation, let's first download the program.You can get the complete code from the link below:
https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-Arduino-Nano-R4/tree/master/lesson_code/15_IR_Module
Open the "15_IR_Module.ino" file located in the "15_IR_Module" folder using the Arduino IDE.

Key Code Explanation

Now let's walk through the project: Printing Remote Button Information


```ino
#include <DIYables_IRcontroller.h>              // Include the DIYables IR controller library

#define IR_RECEIVER_PIN 2                       // Define the Arduino pin connected to the IR receiver

DIYables_IRcontroller_21 irController(         // Create an IR controller object for a 21-key remote
  IR_RECEIVER_PIN,                              // Specify the pin used by the IR receiver
  200                                           // Set debounce time to 200 milliseconds
);

void setup() {                                  // Arduino setup function, runs once
  Serial.begin(115200);                         // Initialize serial communication at 115200 baud rate
  irController.begin();                         // Initialize the IR controller
}

void loop() {                                   // Arduino loop function, runs repeatedly
  Key21 command = irController.getKey();        // Read the pressed key from the IR remote

  if (command != Key21::NONE) {                 // Check if a valid key was received
    switch (command) {                          // Determine which key was pressed

      case Key21::KEY_CH_MINUS:                 // If the CH- key is pressed
        Serial.println("CH-");                  // Print "CH-" to the serial monitor
        break;                                  // Exit the switch case

      case Key21::KEY_CH:                       // If the CH key is pressed
        Serial.println("CH");                   // Print "CH" to the serial monitor
        break;                                  // Exit the switch case

      case Key21::KEY_CH_PLUS:                  // If the CH+ key is pressed
        Serial.println("CH+");                  // Print "CH+" to the serial monitor
        break;                                  // Exit the switch case

      case Key21::KEY_PREV:                     // If the previous (<<) key is pressed
        Serial.println("<<");                   // Print "<<" to the serial monitor
        break;                                  // Exit the switch case

      case Key21::KEY_NEXT:                     // If the next (>>) key is pressed
        Serial.println(">>");                   // Print ">>" to the serial monitor
        break;                                  // Exit the switch case

      case Key21::KEY_PLAY_PAUSE:               // If the play/pause key is pressed
        Serial.println(">||");                  // Print ">||" to the serial monitor
        break;                                  // Exit the switch case

      case Key21::KEY_VOL_MINUS:                // If the volume down key is pressed
        Serial.println("-");                    // Print "-" to the serial monitor
        break;                                  // Exit the switch case

      case Key21::KEY_VOL_PLUS:                 // If the volume up key is pressed
        Serial.println("+");                    // Print "+" to the serial monitor
        break;                                  // Exit the switch case

      case Key21::KEY_EQ:                       // If the EQ key is pressed
        Serial.println("EQ");                   // Print "EQ" to the serial monitor
        break;                                  // Exit the switch case
```

First, import the "DIYables_IRcontroller.h" library used to drive the "IR Module".

```
#include <DIYables_IRcontroller.h>
```

Next, define the infrared receiver pin

```
#define IR_RECEIVER_PIN 2
```

Tell the program that the signal pin of the infrared receiver is connected to the Arduino's D2 pin.

Then, create an infrared remote controller object

```
DIYables_IRcontroller_21 irController(
```

```
  IR_RECEIVER_PIN,
  200
);
```

Create the infrared remote controller object with pin 2 and a debounce time of 200 milliseconds, so it can be easily used throughout the rest of the program

Initialization Function

```
void setup() {
  Serial.begin(115200);
  irController.begin();
}
```

Initialize the serial baud rate so we can view the button output in the Serial Monitor. Initialize the infrared controller to enable infrared signal reception.

Main Loop Function

```
void loop() {
  Key21 command = irController.getKey();
```

Read the button input from the infrared receiver. Here, "Key21" is a data type, "command" is the variable, and the return value of "irController.getKey()" is also of type "Key21". Common return values include "Key21::KEY_1", "Key21::KEY_CH_PLUS", "Key21::NONE", and so on.

Next, check whether a valid button has actually been pressed.

```
  if (command != Key21::NONE) {
```

If no button is detected, simply skip the rest of the logic; otherwise, the Serial Monitor would continuously print meaningless data.

Determine which specific button was pressed

```
    switch (command) {
```

Execute different code based on the button value

```
      case Key21::KEY_CH_MINUS:                  // If the CH- key is pressed
        Serial.println("CH-");                  // Print "CH-" to the serial monitor
        break;
```

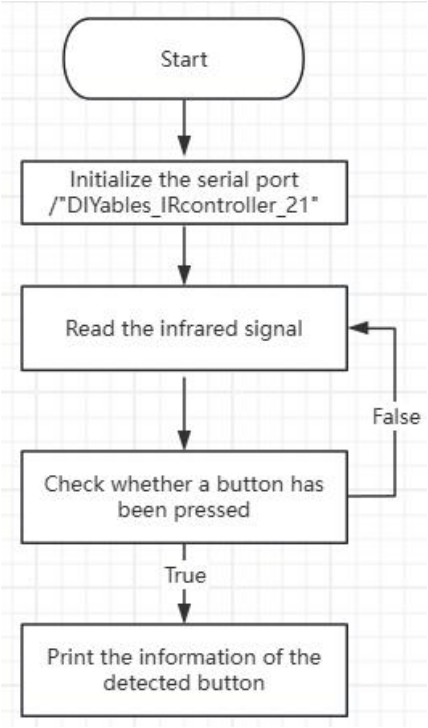
For example, if the return value is "Key21::KEY_CH_MINUS", the Serial Monitor prints "CH-"

If an undefined button is encountered

```
      default:
        Serial.println("WARNING: undefined command:");
        break;
```

output "WARNING: undefined command:"

Overall Code Logic Flowchart



Program Upload Steps

This project requires additional external libraries. Refer to the "Importing Library Files" section on page 7 and import all the required external libraries at the beginning. If you have already imported them, you can skip this step.

For detailed upload instructions, please refer to the "Upload Steps" on page 8.

Key Takeaways:

switch-case	Used to select and execute a specific block of code based on the value of a variable.

Lesson16---Noise_Detector

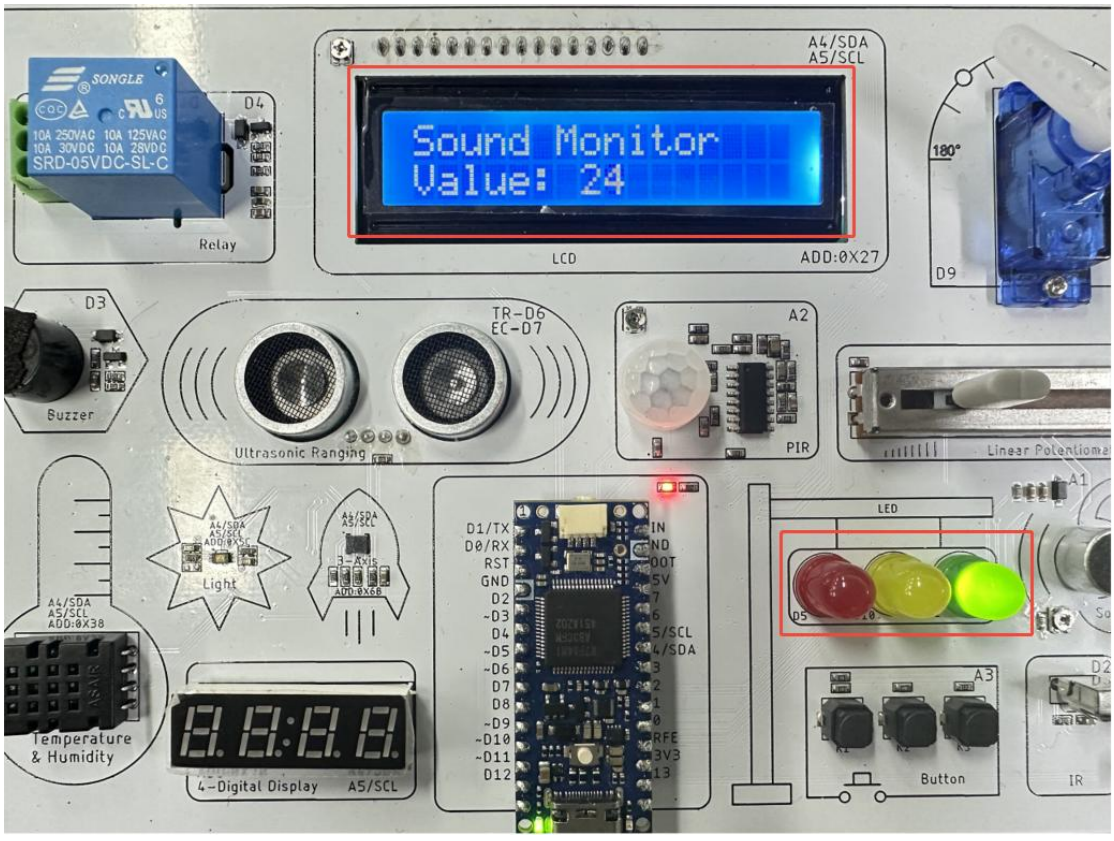
Introduction

In this lesson, we will integrate three modules—the sound sensor, LEDs, and an LCD display—to build a practical environmental noise detector project. Through this lesson, you will review and reinforce previously learned concepts, understand how different modules work together, and learn how to implement coordinated operation across multiple sensors and output devices, further enhancing your overall application skills.

Learning Goals

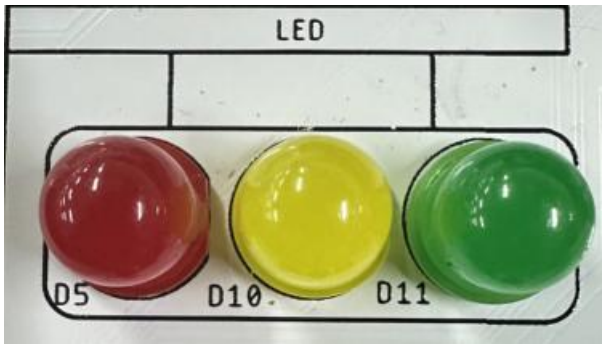
1.Review the basic operations of using Arduino to control hardware
2.Review how Arduino reads values returned by hardware sensors
3.Complete a noise detector experiment that measures ambient sound levels and lights up different LEDs based on the detected sound intensity

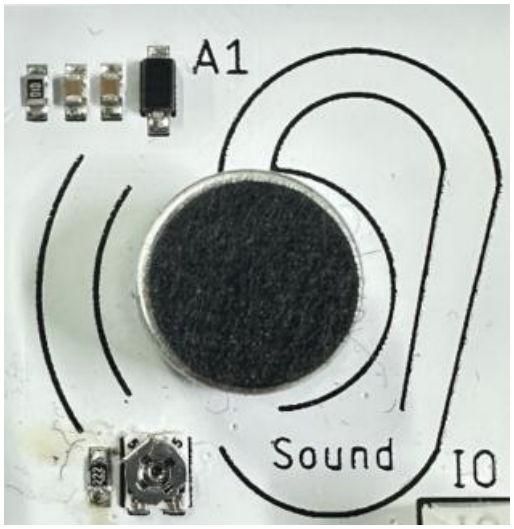
Preview of the Result



After the program runs, the Arduino Nano R4 monitors the ambient sound level. If the environment is quiet, the green LED turns on. If the sound level is moderately noisy, the yellow LED lights up. If the environment is very loud, the red LED turns on

Hardware Used in This Lesson





Noise Detector

Before going through the code, you can download it first.Complete code download link:
https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-Arduino-Nano-R4/tree/master/lesson_code/16_Noise_Detector
Use the Arduino IDE to open the "16_Noise_Detector.ino" file located in the "16_Noise_Detector" folder.

Key Code Explanation

Now let's walk through the example: Noise Detector.

First, import the header files required for this project.

```
#include <Wire.h>
#include <LiquidCrystal_I2C.h>
```

In this example, only the LCD module requires an external library for control, so we include the "LiquidCrystal_I2C.h" header file, along with "Wire.h" to support I2C communication.

Next, define the LCD size.

```
#define COLUMNS 16
#define ROWS    2
```

The LCD has 16 characters per line and a total of 2 lines.

Create the LCD object

```
LiquidCrystal_I2C lcd(
  PCF8574_ADDR_A21_A11_A01,
  4, 5, 6, 16, 11, 12, 13, 14,
  POSITIVE
);
```

Create the LCD object using the device's I2C address.

Define the pins

```
const int Sound_Pin = A1;
const int RED_LED    = 5;
const int YELLOW_LED = 10;
const int GREEN_LED   = 11;
```

Here, we define pins for three LEDs and one sound sensor

Initialization Function

```
void setup() {
  pinMode(RED_LED, OUTPUT);
  pinMode(YELLOW_LED, OUTPUT);
  pinMode(GREEN_LED, OUTPUT);
  lcd.begin(COLUMNS, ROWS, LCD_5x8DOTS);
  lcd.clear();
  lcd.backlight();
  lcd.setCursor(0, 0);
  lcd.print("Sound Monitor");
}
```

Set the three LED pins to output mode, initialize the LCD module, clear the LCD screen, and turn on the backlight. Then, starting at position (0, 0) on the first line, display "Sound Monitor".

In the main loop function

```
void loop() {
  int soundValue = analogRead(Sound_Pin);
```

Continuously read the analog value from the sound sensor and assign it to an integer variable named "soundValue"

Display the sound value on the LCD

```
  lcd.setCursor(0, 1);
  lcd.print("Value: ");
  lcd.print(soundValue);
```

```
  lcd.print("     ");     // Clear remaining characters
```

Print the analog value returned by the sound sensor on the second line of the LCD

Turn off all LEDs

```
  digitalWrite(RED_LED, LOW);
  digitalWrite(YELLOW_LED, LOW);
  digitalWrite(GREEN_LED, LOW);
```

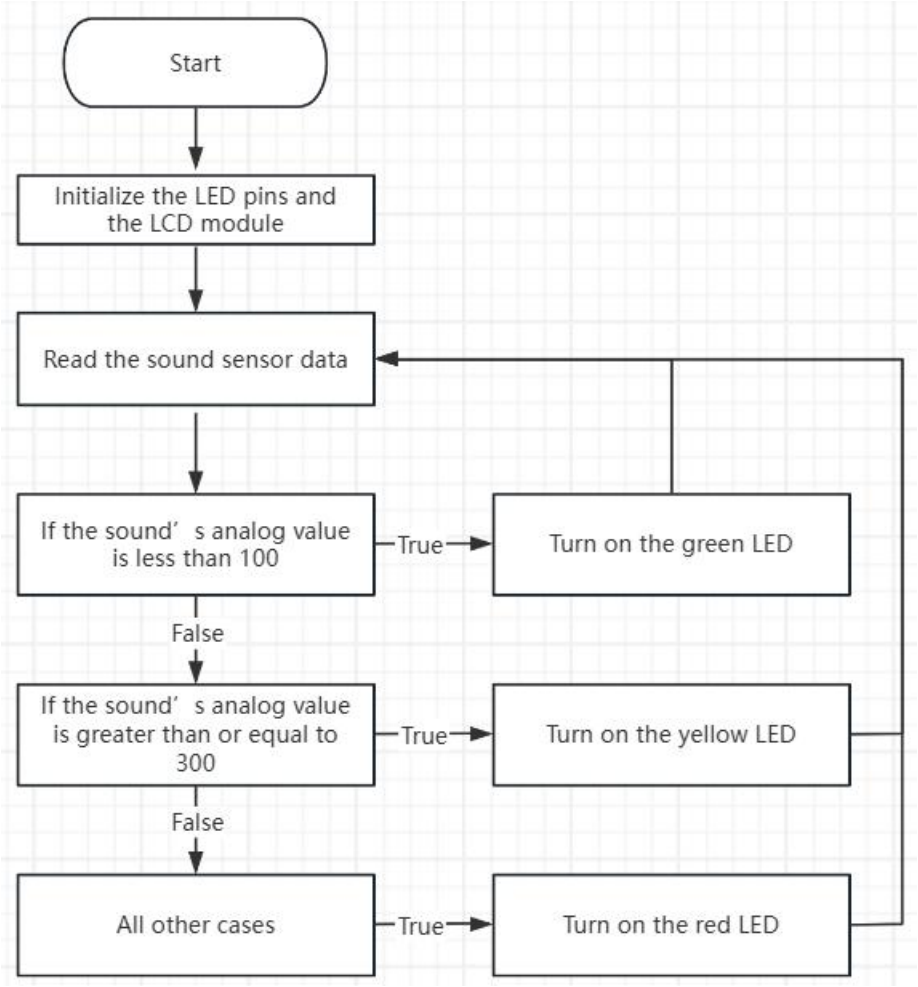
This ensures that only one LED is lit during each cycle.

Determine which LED to light based on the sound level.

```
  if (soundValue < 100) {
    digitalWrite(GREEN_LED, HIGH);
  }
  else if (soundValue <= 300) {
    digitalWrite(YELLOW_LED, HIGH);
  }
  else {
    digitalWrite(RED_LED, HIGH);
  }

  delay(200);   // Refresh rate
}
```

If the sound value is less than 100, turn on the green LED. If the sound value is less than or equal to 300, turn on the yellow LED. All other values will turn on the red LED. A 200 ms delay is used to control the refresh rate.

Overall Code Logic Flowchart



Program Upload Steps

This project requires additional external libraries. Refer to the "Importing Library Files" section on page 7 and import all the required external libraries at the beginning. If you have already imported them, you can skip this step.

For detailed upload instructions, please refer to the "Upload Steps" on page 8.

Lesson17---Ultrasonic Sensor

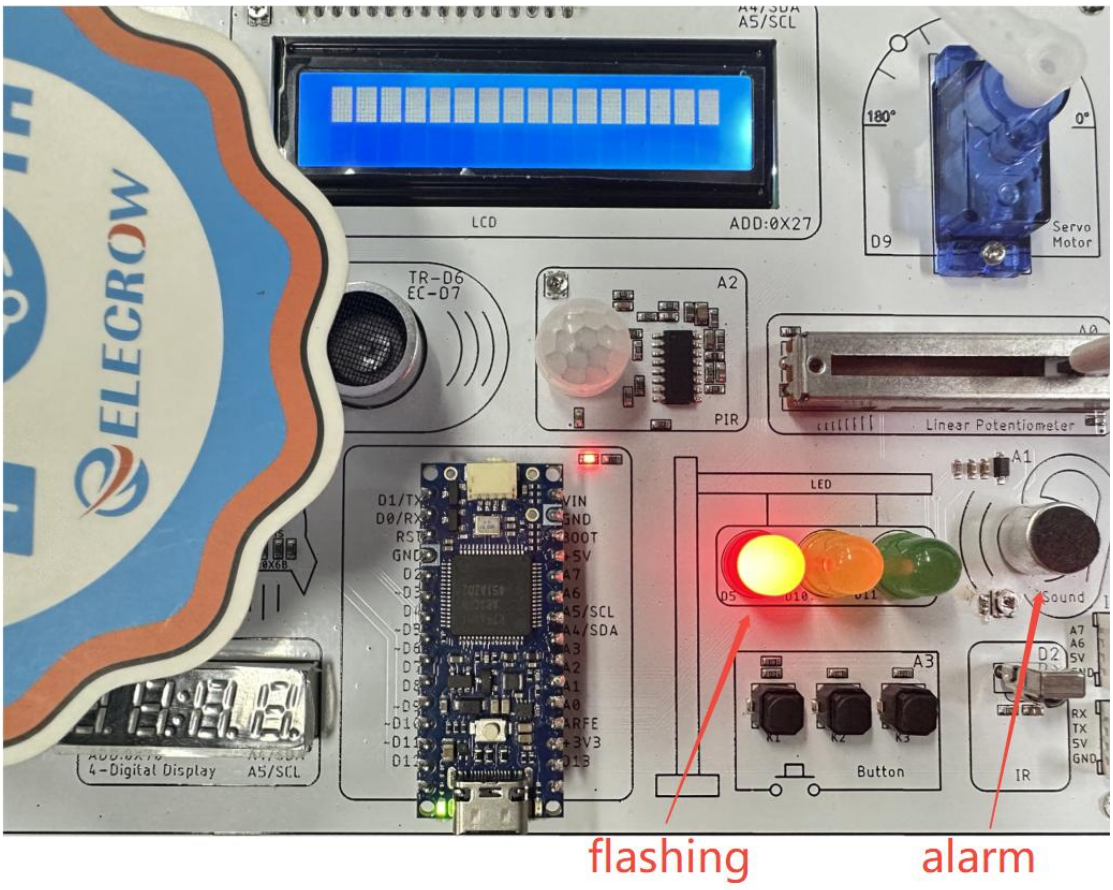
Introduction

In this lesson, we will combine an ultrasonic sensor, a buzzer, and LEDs to build a reverse parking warning system. During the implementation, we will review how functions are called and focus on learning how to use function parameters. By passing different parameters into functions, we can flexibly control the behavior of the buzzer and LEDs. Through this lesson, you will learn how to use sensor data as function parameters and gain a deeper understanding of the importance of functions in program structure and code reuse.

Learning Goals

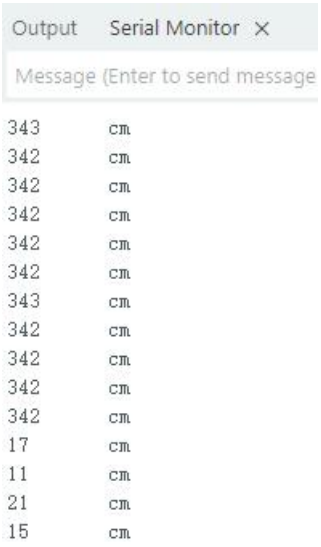
1.Review how to use the ultrasonic sensor
2.Review how to define and call functions
3.Learn and master how to use function parameters
4.Complete the ultrasonic reverse parking alarm experiment.

Preview of the Result



Once the program is uploaded to the Arduino Nano R4, the system begins monitoring the

distance ahead in real time. The ultrasonic sensor continuously measures the distance between itself and any obstacle, and the results are displayed live in the Serial Monitor.



• **When the distance is greater than 20 cm**: The system considers the situation safe. The buzzer remains silent, and the red LED stays off.

• **When the distance is between 11 cm and 20 cm**: The system detects that the obstacle is relatively close. The buzzer sounds at a medium frequency, and the red LED blinks at a moderate rate as a warning.

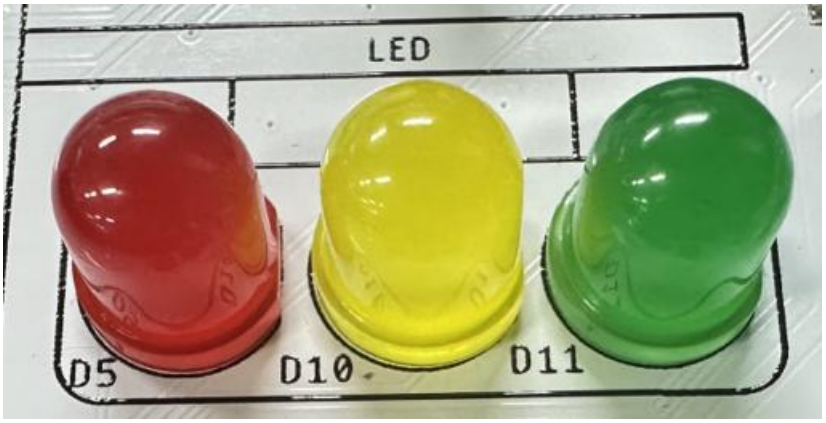
• **When the distance is less than 11 cm**: The system determines that the distance is dangerous. The buzzer sounds at a high frequency, and the red LED flashes rapidly to issue an alert.

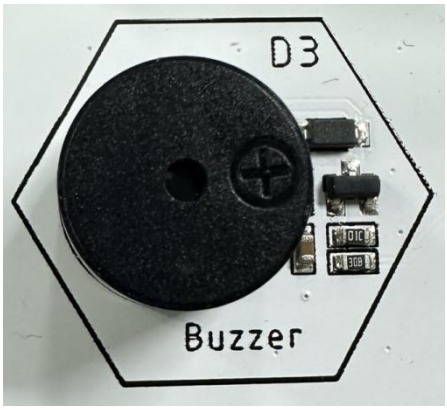
To complete this full reverse parking alarm project, we need to define two functional routines:

1.A function that uses the ultrasonic module to measure distance and return the value in real time.

2.A function that controls the buzzer and red LED to generate different alert behaviors based on the measured distance.

Hardware Used in This Lesson





Ultrasonic Reverse Parking Alarm

Before diving into the code explanation, you can download the complete program first.Full code download link:
https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-Arduino-Nano-R4/tree/master/lesson_code/17_Reverse_Parking_Alarm_Radar
Open the "17_Reverse_Parking_Alarm_Radar" folder in the Arduino IDE, and then open the "17_Reverse_Parking_Alarm_Radar.ino" file inside it.

Key Code Explanation

Now let's walk through the project example: **Ultrasonic Reverse Parking Alarm**.


```
int triggerPin = 6;
int echoPin = 7;
#define BUZ 3         // Buzzer pin
#define LED_RED 5     // Red LED pin

// Function: Blink LED + buzzer sound
void flashLedBuzzer(int interval) {
  digitalWrite(LED_RED, HIGH);
  tone(BUZ, 1000);       // High-frequency buzzer sound
  delay(interval);        // Parameter controls blinking interval
  digitalWrite(LED_RED, LOW);
  noTone(BUZ);
  delay(interval);
}

// Custom function: measure ultrasonic distance (in centimeters)
float measureDistanceCm() {
  digitalWrite(triggerPin, LOW);
  delayMicroseconds(2);
  digitalWrite(triggerPin, HIGH);
  delayMicroseconds(10);
  digitalWrite(triggerPin, LOW);
  long duration = pulseIn(echoPin, HIGH);
  float distance = duration * 0.0343 / 2;
  Serial.print((int)distance);
  Serial.println("\t cm");
  delay(100); // Small delay
  return distance;
}

void setup() {
  Serial.begin(115200);
  pinMode(triggerPin, OUTPUT);
  pinMode(echoPin, INPUT);
  pinMode(BUZ, OUTPUT);
  pinMode(LED_RED, OUTPUT);
}

void loop() {
  float distance = measureDistanceCm();
  if (distance >= 0 && distance <= 10) {
    // 0-10 cm: buzzer beeps rapidly, LED blinks fast
    flashLedBuzzer(50);   // 50ms interval, rapid
  }
  else if (distance >= 11 && distance <= 20) {
    // 11-20 cm: buzzer beeps slowly, LED blinks slowly
    flashLedBuzzer(150);  // 150ms interval, slow
  }
  else {
    // More than 20 cm: buzzer off, LED off
    digitalWrite(LED_RED, LOW);
    noTone(BUZ);
  }
  delay(200);  // Small delay after each measurement
}
```

First, we define the pins required for this lesson:

```
int triggerPin = 6;
int echoPin = 7;
#define BUZ 3          // Buzzer pin
#define LED_RED 5      // Red LED pin
```

triggerPin:This is the trigger (transmit) pin, connected to D6 on the Arduino Nano R4.

echoPin:This is the echo (receive) pin, connected to D7 on the Arduino Nano R4.

Use void to define the alarm function

```
void flashLedBuzzer(int interval) {
  digitalWrite(LED_RED, HIGH);
  tone(BUZ, 1000);          // High-frequency buzzer sound
  delay(interval);          // Parameter controls blinking interval
  digitalWrite(LED_RED, LOW);
  noTone(BUZ);
  delay(interval);
}
```

Our alert behavior works as follows: the red LED turns on and the buzzer sounds → then the red LED turns off and the buzzer stops. The parameter "interval" is the key to controlling different alert frequencies—the shorter the interval, the more urgent the alarm; the longer the interval, the gentler the alert.

void:Specifies the function's return type. Using "void" means the function does not return any value.

flashLedBuzzer:This is the function name. A function name cannot start with a number, but it may contain letters, numbers, and underscores, and it is case-sensitive. Most importantly, a clear function name helps readers immediately understand what the function does.

int interval:This defines a parameter that is passed into the delay function, indicating that it is used to control the delay duration when the function is called.

When calling the function, we pass in an argument inside the parentheses.This argument replaces the parameter throughout the function body.

Parameter: A variable listed inside the parentheses when a function is defined. It is used to receive values passed in from outside the function and is a local variable, meaning it is only valid within the function itself.

Argument: The actual value or variable provided inside the parentheses when calling a function. These values are assigned to the parameters so the function can use them.

Define the ultrasonic distance measurement function using float (key focus).

```
float measureDistanceCm() {
  digitalWrite(triggerPin, LOW);
  delayMicroseconds(2);
  digitalWrite(triggerPin, HIGH);
  delayMicroseconds(10);
  digitalWrite(triggerPin, LOW);
  long duration = pulseIn(echoPin, HIGH);
  float distance = duration * 0.0343 / 2;
  Serial.print((int)distance);
  Serial.println("\t cm");
  delay(100); // Small delay
  return distance;
}
```

Key Points to Understand:

From the code, we can see the following steps:

①First, the "triggerPin" is set to a LOW level and held for 2 microseconds. This ensures the pin is in a stable LOW state before sending the trigger pulse.

②Next, the "triggerPin" is set to HIGH and held for 10 microseconds. This step initiates the ultrasonic pulse. For most ultrasonic modules, 10 microseconds is the minimum required trigger pulse width.

③After the 10 microseconds, the pin is pulled LOW again to end the pulse, completing one ultrasonic transmission.

delayMicroseconds(2): delays execution for 2 microseconds

delay(2):delays execution for 2 milliseconds

long duration = pulseIn(echoPin, HIGH);

"long"can store larger integers than "int", which is why we use "long" to define the variable "duration", since we are dealing with microsecond-level numbers.

pulseIn() is an Arduino function that measures how long a pin stays at a certain voltage level. Here, we use it with "echoPin" and "HIGH", meaning it measures the duration that "echoPin" remains HIGH. After the ultrasonic module emits a pulse, "echoPin" goes HIGH, and when the reflected sound wave returns, "echoPin" goes LOW. Measuring the HIGH duration of "echoPin" thus gives the total time it takes for the ultrasonic pulse to travel to the obstacle and back.

float distance = duration * 0.0343 / 2;

float is used to define a floating-point variable (a number with decimals). Since the calculations involve decimal values, we define the variable as float. Using int or long would store only whole numbers, discarding the fractional part and losing precision. Using float preserves the decimals, making the distance measurement more accurate.

duration is the time measured for the ultrasonic pulse to travel to the obstacle and back.

0.0343 represents the speed of sound converted to centimeters per microsecond: 343 meters per second equals 0.0343 cm/μs.

return distance;

The function returns a value—the final measured distance. After calling the "measureDistanceCm" function, we can store its return value in a variable to obtain the measured distance.

Initialization Function

```
void setup() {
  Serial.begin(115200);
  pinMode(triggerPin, OUTPUT);
  pinMode(echoPin, INPUT);
  pinMode(BUZ, OUTPUT);
  pinMode(LED_RED, OUTPUT);
}
```

Note: Here, the "triggerPin" is an output pin, while the "echoPin" is an input pin. Make sure not

to mix them up when setting the pin modes.

loop Function

```
void loop() {
  float distance = measureDistanceCm();
  if (distance >= 0 && distance <= 10) {
    // 0-10 cm: buzzer beeps rapidly, LED blinks fast
    flashLedBuzzer(50);     // 50ms interval, rapid
  }
  else if (distance >= 11 && distance <= 20) {
    // 11-20 cm: buzzer beeps slowly, LED blinks slowly
    flashLedBuzzer(150);   // 150ms interval, slow
  }
  else {
    // More than 20 cm: buzzer off, LED off
    digitalWrite(LED_RED, LOW);
    noTone(BUZ);
  }
  delay(200);   // Small delay after each measurement
}
```

Since we've already defined the functionality as separate functions—for example, "measureDistanceCm" handles distance measurement and "flashLedBuzzer" can produce different alert sounds simply by passing different parameters—our main loop becomes very concise.

float distance = measureDistanceCm();

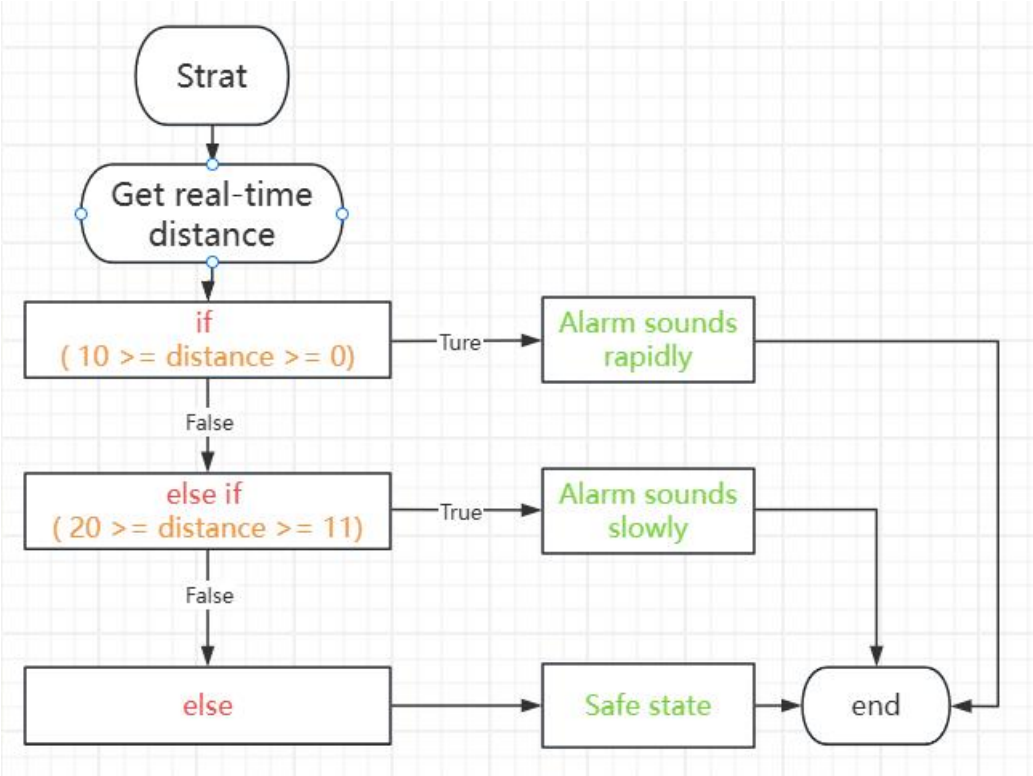
As discussed earlier, "measureDistanceCm" returns the measured distance, which we store in the variable distance. This makes it easy to use in subsequent code for distance-based decisions.

When the distance is 0–10 cm, the buzzer sounds rapidly, and the red LED flashes at a high frequency.

When the distance is 11–20 cm, the buzzer beeps slowly, and the red LED flashes at a medium rate.

When the distance is greater than 20 cm, the system is in a safe state: the buzzer is silent, and the red LED is off.

Overall Code Logic Flowchart



Program Upload Steps

For detailed upload instructions, please refer to the "Upload Steps" on page 8.

Lesson18---LED Light Flashing Memory Game

Introduction

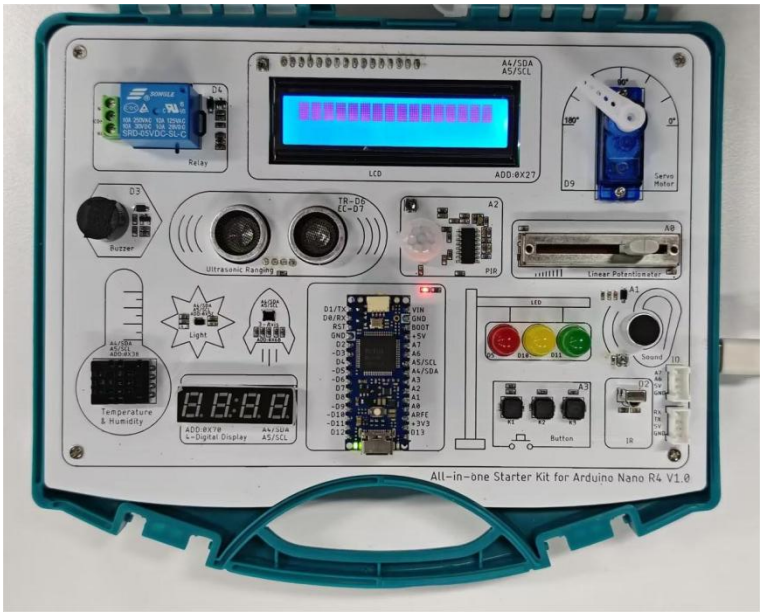
In this class, we will learn how to use LED lights and buttons to create a simple but very classic memory game.
In this project, Arduino will first control three LED lights to flash in a certain sequence rapidly. Students need to remember the flashing sequence of the LEDs, and then by pressing the corresponding buttons, they need to reproduce the flashing process of the LEDs in the correct order. If the sequence is completely correct, the challenge is successful; if there is an error at any point, the challenge fails and they need to start over.
Through the learning of this class, students will no longer just "light up the LED", but will be able to understand the basic ideas of random numbers, arrays, sequential judgment, and human-computer interaction. This is a very important comprehensive improvement course.

Learning Goals

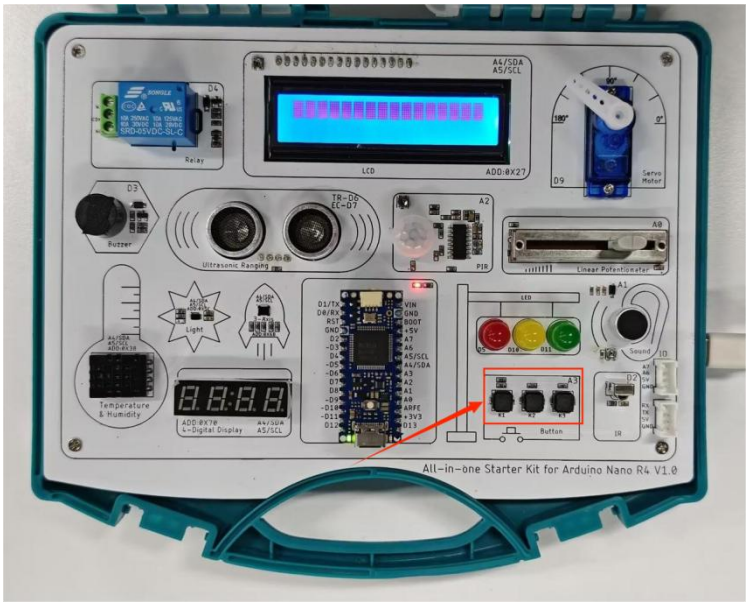
1. Understand the basic implementation concept of sequential memory games.
2. Master the usage of LED blinking control and function encapsulation.
3. Learn to use arrays to store data in sequence.
4. Consolidate the method of simulating key presses (resistor voltage division) to read multiple keys.
5. Be able to adjust the game difficulty (blinking speed and frequency) through variables.

Preview of the Result

After successfully uploading the code, you will see the following effect:
After powering on, the three LEDs will flash in a random sequence one after another (for example: yellow -> green -> green -> red -> green)



After the LED flashing is over, it's your turn to operate.
You need to use the buttons according to your memory and press the sequence exactly the same as the LED flashing.
(For example: follow the sequence given in the above example: yellow -> green -> green -> red -> green)

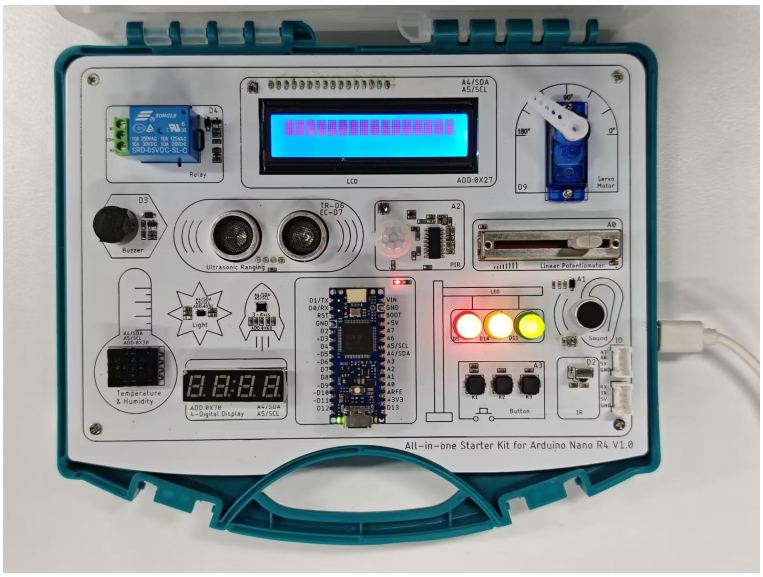


If all are correct:

- The three LEDs will light up simultaneously for 1 second, indicating a successful challenge.

If you make a mistake during the process:

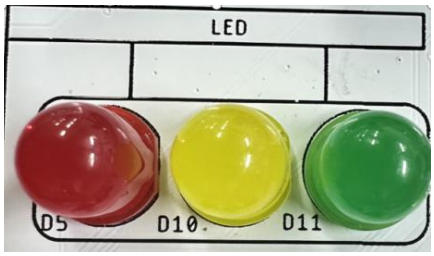
- The three LEDs will flash rapidly, indicating that the challenge has failed.



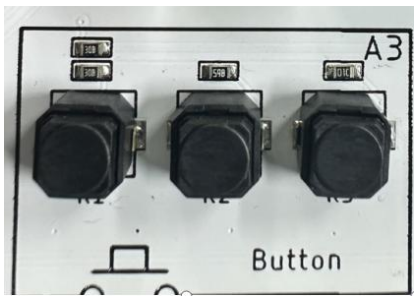
Then the game will automatically restart and enter the next round.

Hardware Used in This Lesson

Three-color LED lights



Three buttons



LED Light Flashing Memory Game Case

Before explaining the code, we can download the code. The overall code download link is: https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-Arduino-Nano-R4/tree/master/lesson_code/18_LED_Light_Flashing_Memory_Game

Open the "18_LED_Light_Flashing_Memory_Game" folder in Arduino IDE and select "18_LED_Light_Flashing_Memory_Game.ino" code file

Key Code Explanation

After opening the code, you will be able to see the code for this lesson:

18_LED_Light_Flashing_Memory_Game.ino

```
// ================== Pin Definitions ==================
#define LED_RED     5
#define LED_YELLOW  10
#define LED_GREEN   11

#define Button_Pin  A3   // Analog button input

// ================== Game Settings ==================
#define MAX_LEVEL     10

int levelLength = 5;      // Sequence length (difficulty)
int flashDelay  = 400;    // LED flash speed (ms)

// ================== Variables ==================
int ledSequence[MAX_LEVEL];
int userIndex = 0;

// ================== Setup ==================
void setup() {
  pinMode(LED_RED, OUTPUT);
  pinMode(LED_YELLOW, OUTPUT);
  pinMode(LED_GREEN, OUTPUT);
  pinMode(Button_Pin, INPUT);

  Serial.begin(115200);
  randomSeed(analogRead(A0));

  startGame();
}

// ================== Main Loop ==================
void loop() {
  int button = readButton();

  if (button != 0) {
    flashSingleLED(button);

    if (button == ledSequence[userIndex]) {
      userIndex++;

      // All correct
```

Definition of pins

```
#define LED_RED         5
#define LED_YELLOW    10
#define LED_GREEN      11
#define Button_Pin    A3      // Analog button input
```

LED_RED, LED_YELLOW, LED_GREEN correspond to digital pins 5, 10, and 11 respectively, and are used to drive the red, yellow, and green LEDs. This way, in the subsequent pinMode() and digitalWrite() functions, the code will be very intuitive to read - seeing LED_RED immediately means it is controlling the red light, without the need to repeatedly check "What is connected to

pin 5?".

Button_Pin is defined as A3 and specifically noted as Analog button input, indicating that this is not a simple digital button, but multiple buttons sharing a single analog input pin through resistor voltage division. Later in the program, different voltage ranges are read using analogRead(A3) to determine which button was pressed.

Define variables

```
#define MAX_LEVEL        10
int levelLength = 5;          // Sequence length (difficulty)
int flashDelay    = 400;       // LED flash speed (ms)
int ledSequence[MAX_LEVEL];
int userIndex = 0;
```

#define MAX_LEVEL 10 Defines a constant for the maximum length of a level, which not only limits the upper limit of game difficulty but also determines the maximum capacity of the ledSequence array to prevent array out-of-bounds errors. This is an important safety design thinking in embedded programming.

levelLength = 5 Indicates the length of the light sequence that needs to be memorized in the current game round. The larger the value, the more steps the player needs to remember. Therefore, it is essentially a game difficulty control parameter. If you want to make the game more difficult in the future, you just need to change this value.

flashDelay = 400 Used to control the duration of each LED's illumination, with the unit being milliseconds. It directly affects the player's rhythm of "seeing the lights clearly". The smaller the value, the faster the flashing, and the higher the reaction difficulty.

The following ledSequence[MAX_LEVEL] is an integer array used to store the randomly generated LED sequence by the system. Each number in the array represents a light (for example, 1 = red light, 2 = yellow light, 3 = green light). This is the core data structure of the entire memory logic.

userIndex = 0 Is a progress pointer variable used to record the step that the player has correctly input so far. Every time the player gets it right, this index increases by one. Once it reaches levelLength, it indicates that the player has successfully memorized the entire level.

Setup Section

```
void setup() {
  pinMode(LED_RED, OUTPUT);
  pinMode(LED_YELLOW, OUTPUT);
  pinMode(LED_GREEN, OUTPUT);
  pinMode(Button_Pin, INPUT);

  Serial.begin(115200);
  randomSeed(analogRead(A0));

  startGame();
}
```

This setup() function is responsible for the initialization work of the entire memory light game when it is powered on or reset. It can be understood as the "preparation stage before the game

officially starts".

Firstly, by using pinMode(LED_RED, OUTPUT), pinMode(LED_YELLOW, OUTPUT), and pinMode(LED_GREEN, OUTPUT), the pins connected to the three LEDs are configured as output mode. This enables the Arduino to actively output high or low levels to these pins, thereby controlling the on/off state of the lights;
while pinMode(Button_Pin, INPUT) sets the analog pin where the button is located to input mode, which is used to read the player's operation. This is the entry point for human-computer interaction.

Then, Serial.begin(115200) initializes the serial communication, which is mainly used for debugging and observing the program's running status, such as checking the analog value of the button or the game process. This is very important during the learning and troubleshooting stages.

"randomSeed(analogRead(A0))" is a crucial but often overlooked detail. It initializes the random number seed by reading the value of an unconnected or noisy analog pin, ensuring that the light sequence generated by "random()" is different each time the power is turned on. Otherwise, the game sequence would repeat, losing the significance of the "random challenge".
"randomSeed()" is a function in Arduino used to initialize the random number sequence, and "analogRead(A0)" reads the voltage value on analog pin A0. In an actual circuit, if pin A0 is not connected to any stable signal, it will be affected by environmental noise, and the value read each time will be slightly different. It is precisely this "unpredictable minor variation" that serves as the source of the random seed. Subsequent calls to "random()" will generate target values that will not repeat the same set of numbers each time the power is turned on, which is extremely important for enhancing the fairness and fun of the game.

The last called function, startGame(), is the actual "start-of-game command". It resets the player's input progress, generates a new random light sequence, and displays it to the player. This is equivalent to connecting all the previous hardware and system preparations together, and officially starting the first round of the game.

loop

```
void loop() {
  int button = readButton();

  if (button != 0) {
    flashSingleLED(button);

    if (button == ledSequence[userIndex]) {
      userIndex++;

      // All correct
      if (userIndex >= levelLength) {
```

```
            successLED();
            delay(500);
            startGame();
        }
    } else {
        failLED();
        delay(500);
        startGame();
    }

    delay(300); // Prevent a single key press from being read multiple times
  }
}
```

This loop() function is the core logic loop of the entire memory light game, which can be regarded as a "continuous input monitoring system that promptly judges right or wrong and serves as a referee".

At the beginning of the program, the readButton() function reads which button the current player has pressed and stores the result in the variable button. Here, the return values 1/2/3 represent the red, yellow, and green lights respectively, while 0 indicates that no button is currently pressed. Only when button != 0, it means that the player has actually made an operation, and the program will continue to execute further, thus avoiding idle running and misjudgment.

```
31  // ================== Main Loop ==================
32  void loop() {
33    int button = readButton();
```

When an effective button is detected, the function flashSingleLED(button) is called first. This causes the corresponding colored LED to flash once immediately. This step serves as a visual feedback, informing the player that "your button has been recognized", and also makes the operation process more intuitive and game-like.

```
35    if (button != 0) {
36      flashSingleLED(button);
```

Next, we come to the most crucial judgment logic: if (button == ledSequence[userIndex]) - here, the current key pressed by the player is compared with the correct answer at the "current position" in the pre-generated lighting sequence by the system; if they are equal, it indicates that this input is correct, so userIndex++ is executed, meaning the player has successfully completed one position in the current sequence, and the system is ready to check the next one.

```
38      if (button == ledSequence[userIndex]) {
39        userIndex++;
```

The subsequent if (userIndex >= levelLength) is used to determine whether all the light sequences of this round have been completed. When the player has correctly arranged the entire sequence, the program will call the successLED() function to play the success light effect, providing the player with clear positive feedback; then, delay(500) will pause for a short while to allow the success effect to be clearly seen. Finally, startGame() will be called, generating a new

light sequence and starting the next round of challenges, which is equivalent to "passing the level and entering a new round".

```
41      // All correct
42      if (userIndex >= levelLength) {
43        successLED();
44        delay(500);
45        startGame();
46      }
```

If the key pressed by the player does not match the correct sequence, it will enter the else branch, indicating an incorrect input. At this point, the program calls the failLED() function, providing a failure prompt through rapid flashing or other means. After a short delay of 500 milliseconds, it also calls the startGame() function, directly restarting the game and allowing the player to try again. This design is both simple and in line with the beginner's intuitive understanding of the game process.

```
47      } else {
48        failLED();
49        delay(500);
50        startGame();
51      }
```

The final "delay(300)" is a very practical little detail. It serves as a software anti-shake and anti-duplication mechanism, preventing a single key press from being repeatedly read by the program due to the finger not releasing in time, thus avoiding input errors.

```
53      delay(300); // Prevent a single key press from being read multiple times
54    }
55  }
```

Overall, the structure of this loop() function is clear: read input → provide feedback → determine correctness → decide success or failure → control the game pace.

startGame function

```
void startGame() {
  userIndex = 0;
  generateSequence();
  showSequence();
}
```

This function, named "startGame()", can be regarded as "the initialization and opening ceremony of each game session". It is called whenever the game starts, the player successfully completes the level, or fails and has to retry. Its function is to restore the system state to a clean and controllable starting point.

The first line of the function, "userIndex = 0;", is used to reset the current position index of the player's input, indicating that the player has not yet started inputting any light sequence. This is equivalent to telling the program: "Start comparing from the first light again."

Then, the function "generateSequence()" is called. This line is the core part of generating the content for this game session, which is to randomly generate a sequence of LEDs and store it in

the "ledSequence" array. Due to the use of random numbers, this step ensures that the lighting combination for each game session is different, enhancing the challenge and playability of the game, and preventing players from "cheating" by relying on remembering the previous sequence. Finally, call the function showSequence(); This is a very crucial "presentation stage" in the game. The system will sequentially light up the LEDs in the order just generated, presenting the correct answer to the player in full.

That is to say, the function startGame() accomplishes all three tasks - "clearing the state → generating questions → presenting the questions to the player" - in one go, preparing the groundwork for the player input and judgment logic in the subsequent loop().

generateSequence function

```
void generateSequence() {
  for (int i = 0; i < levelLength; i++) {
    ledSequence[i] = random(1, 4); // 1=Red, 2=Yellow, 3=Green
  }
}
```

The core function of this generateSequence() is to generate a "random light sequence" for the current round of the memory game. It can be regarded as the process by which the system automatically "sets the question" at the beginning of each level.

The entire function structure is very clear: the outer layer is a for loop, the number of iterations is determined by levelLength, and this variable itself represents "how many lights need to be memorized in this level", which is a direct reflection of the game difficulty - the larger the value, the longer the sequence that the player needs to remember, and the higher the challenge.

Within the loop, the line "ledSequence[i] = random(1, 4);" is the most crucial one. Here, the random() function provided by Arduino is used. The meaning of "random(1, 4)" is to generate a random integer that is greater than or equal to 1 and less than 4. That is to say, the result can only be 1, 2, or 3. In the design of the program, these three numbers have been artificially assigned meanings: 1 represents the red LED, 2 represents the yellow LED, and 3 represents the green LED. Thus, a simple number can directly represent "which light should be on next".

During each cycle, a new random value is generated and stored in the ledSequence array in sequence, with the corresponding position determined by the index i. The result of this is that the array not only stores "which lights are on", but also completely retains the sequence in which the lights appeared. In the subsequent program, the showSequence() function will flash the LEDs in this array's order one by one to help the player remember; while when the player inputs, the same array will be compared step by step with the player's operation to determine if it is correct.

It is very important to encapsulate "generating random sequences" into a separate function: on the one hand, it makes the main process (such as startGame()) more intuitive to read; on the other hand, it leaves room for future expansion of the game, such as adding new LED colors, dynamically increasing levelLength, and even introducing different difficulty modes. All these can

be achieved simply by making a few modifications in this part.

showSequence function

```
void showSequence() {
   delay(800);
   for (int i = 0; i < levelLength; i++) {
      flashSingleLED(ledSequence[i]);
      delay(200);
   }
}
```

The function showSequence() serves to present the randomly generated lighting sequence just generated by the system to the players. It can be regarded as the stage in a memory game where "questions are presented and prompts are played". Throughout the entire game process, it assumes the role of a "teacher demonstration": first, it tells you what the sequence is, and then it's your turn to operate.

The "delay(800);" statement at the beginning of the function was not added randomly. Its purpose is to give the player a buffer time before the lighting display starts. Usually, after the end of the previous round or the beginning of a new round, the player needs some time to switch from the prompt text or mental state. This 800-millisecond pause is equivalent to a countdown for "getting ready to start", making the lighting display less abrupt.

Next comes a for loop, and the number of iterations is also determined by levelLength, which is exactly the same as in the generateSequence() function. This ensures that "the number of lights generated is the same as the number of lights displayed". The core statement within the loop is flashSingleLED(ledSequence[i]);

Here is a crucial point: The ledSequence[i] retrieves not a specific fixed light, but the i-th light number that was randomly generated and stored in the array previously.
That is to say, the system plays back the entire lighting sequence exactly in the order of generation, without missing any or skipping any. The flashSingleLED() function is responsible for the specific hardware actions - turning on the corresponding LED, delaying, and then turning it off. Thus, showSequence() doesn't need to concern itself with the underlying details; it only needs to "tell it which light to flash".

The subsequent "delay(200);" represents the time interval between two adjacent lights. Without this delay line, the flashing of multiple LEDs would be continuous, making it look very chaotic and making it nearly impossible for the player to discern the sequence. With this 200-millisecond gap, each light becomes a distinct and clear "memory point", significantly enhancing readability and the gaming experience.

readButton function

```
int readButton() {
   int val = analogRead(Button_Pin);
```

```
  Serial.println(val);

  if (val >= 500 && val <= 520) return 1;     // Red
  if (val >= 680 && val <= 690) return 2;     // Yellow
  if (val >= 845 && val <= 860) return 3;     // Green

  return 0;
}
```

The function "readButton()" serves to read which button the player has pressed and convert it into a number that can be processed by the game. It is the "core interface for player input detection" in the entire memory game, and can be understood as a "translator for player actions": when the player presses a physical button, this function converts the action into the digital numbers 1, 2, or 3 that the program can understand.

First, the line "int val = analogRead(Button_Pin);" reads the voltage value through the analog pin. Since the three buttons are connected to the same analog input through resistors for voltage division, each button press generates a different voltage, corresponding to a different numerical range. Using analogRead allows us to obtain an integer value ranging from 0 to 1023.

The following Serial.println(val); is a debugging statement, which prints the current analog value on the serial monitor. This is very useful for debugging the sensitivity of the buttons, the error of resistors, or the hardware connections. You can directly see the corresponding voltage values when each button is pressed, which is convenient for calibration.

Then there are three if conditions:

```
87    if (val >= 500 && val <= 520) return 1;    // Red
88    if (val >= 680 && val <= 690) return 2;    // Yellow
89    if (val >= 845 && val <= 860) return 3;    // Green
90
```

These lines are used to map the analog voltage to the button number. Each button has a reasonable voltage range. For example, when the red light button is pressed, the voltage value is approximately between 500 and 520, and the function returns 1; the yellow light returns 2, and the green light returns 3. Through this mapping, the program can internally know which button the player has pressed without directly processing the voltage value.

The final "return 0;" is the default return value, indicating that no button has been pressed. When none of the three buttons are pressed, the function returns 0, letting the main loop know that "there is no input at this moment".

flashSingleLED function

```
void flashSingleLED(int num) {
  allLEDOff();

  if (num == 1) digitalWrite(LED_RED, HIGH);
```

```
  if (num == 2) digitalWrite(LED_YELLOW, HIGH);
  if (num == 3) digitalWrite(LED_GREEN, HIGH);

  delay(flashDelay);
  allLEDOff();
}
```

The function flashSingleLED(int num) serves to make a single LED light flash once, which is used to alert the player or display the game status. It can be regarded as "turn on the light of a specific color for a short period of time and then turn it off", and it is the core function for visual feedback in the game.

The first line: allLEDOff(); Ensure that all LEDs are turned off before the flashing starts to avoid any residual state from the previous flashing affecting the current display, and to ensure that the flashing action is clearly distinguishable.

The following three if statements determine which LED is lit based on the input parameter num:

```
98    if (num == 1) digitalWrite(LED_RED, HIGH);
99    if (num == 2) digitalWrite(LED_YELLOW, HIGH);
100   if (num == 3) digitalWrite(LED_GREEN, HIGH);
```

- When num equals 1 → Turn on the red light
- When num equals 2 → Turn on the yellow light
- When num equals 3 → Turn on the green light

Here, using digitalWrite(..., HIGH) is the standard method in Arduino to power on and light up the LED.

Then, calling delay(flashDelay); will pause the program execution for a period of time, which is determined by flashDelay and is usually set to several hundred milliseconds. This allows players to clearly observe the LED flashing effect. (Here it is 400ms)

Finally, call the function allLEDOff() again to turn off the LEDs, completing a full blinking action.

successLED function

```
void successLED() {
  digitalWrite(LED_RED, HIGH);
  digitalWrite(LED_YELLOW, HIGH);
  digitalWrite(LED_GREEN, HIGH);
  delay(1000);
  allLEDOff();
}
```

By using digitalWrite(..., HIGH) to simultaneously turn on the red, yellow, and green LEDs, a distinct blinking prompt effect is achieved, indicating "success". This is the most intuitive positive feedback in the game, letting the player know that their operation is correct.
The program pauses for 1000 milliseconds (1 second) to ensure that the three LEDs can remain illuminated for a period of time, allowing the player to fully notice this "success prompt" and not

let it pass by in a flash.

The allLEDOff() function defined earlier is called to turn off all three LEDs, preparing for the next game or the next round of blinking.

failLED function

```
void failLED() {
  for (int i = 0; i < 10; i++) {
    allLEDOn();
    delay(100);
    allLEDOff();
    delay(100);
  }
}
```

Inside the function, a for loop is used: it runs 10 times. Each time, all the LED lights will be turned on once and then off once, creating a continuous flashing effect, simulating the feeling of "alarm" or "error prompt".

- allLEDOn(): Turns on the red, yellow, and green LED lights, allowing players to immediately notice the error.
- delay(100): Keeps the lights on for 100 milliseconds.
- allLEDOff(): Turns off all the LED lights.
- delay(100): Keeps them off for 100 milliseconds, creating the flashing interval.

allLEDOn function

```
void allLEDOn() {
  digitalWrite(LED_RED, HIGH);
  digitalWrite(LED_YELLOW, HIGH);
  digitalWrite(LED_GREEN, HIGH);
}
```

By calling the allLEDOn() function, the program can light up all the LEDs at once without having to repeatedly write three digitalWrite statements each time. This simplifies the code and enables logic reuse, achieving the effect of concise code and logical reuse. It is particularly suitable for being called in multiple places where a "full-on" indication is required, such as in successLED() or failLED().

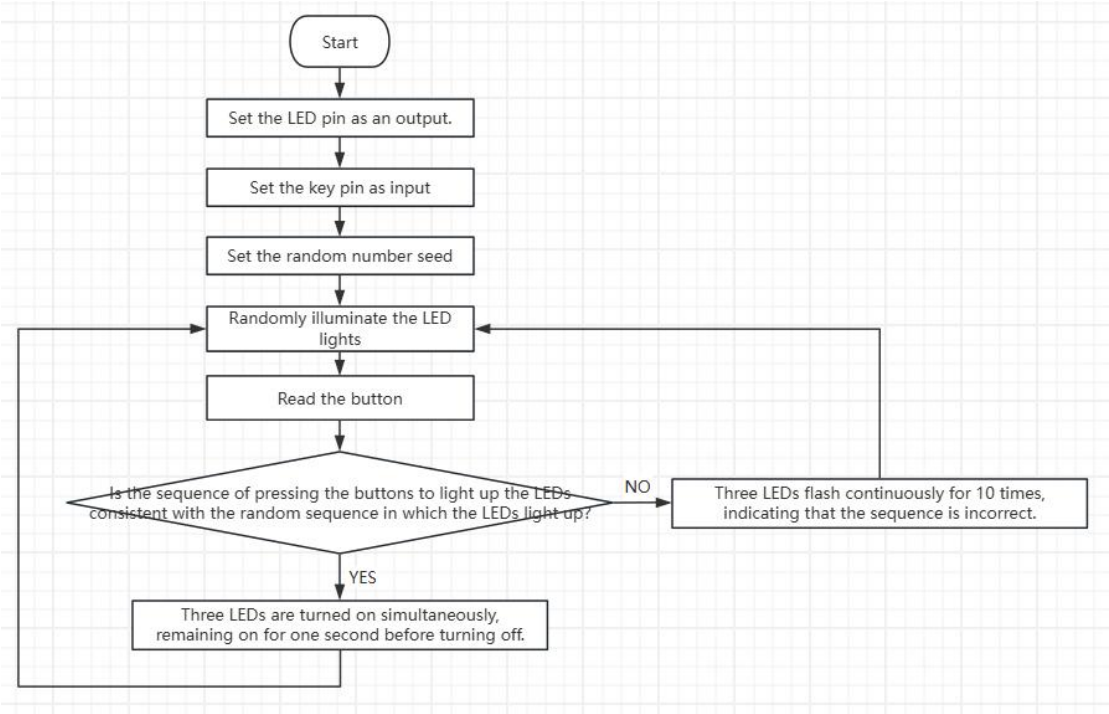
allLEDOff function

```
void allLEDOff() {
  digitalWrite(LED_RED, LOW);
  digitalWrite(LED_YELLOW, LOW);
  digitalWrite(LED_GREEN, LOW);
}
```

By calling the function allLEDOff(), the program can turn off all the LEDs at once, without having to repeatedly write three digitalWrite statements. This makes it convenient to reset or clear the prompt status in the game logic, improving the readability and reusability of the code. For example, in functions like flashSingleLED(), successLED(), or failLED(), this function is used

multiple times to control the lighting on and off.

Overall code logic flowchart



Upload Program Steps

For detailed upload instructions, please refer to the "Upload Steps" on page 8.

Lesson19---Sliding Resistor Guessing Game

Introduction

In this class, we will learn the comprehensive application of potentiometers, LCD1602 displays and LED indicators through an interesting mini-game.

Game rules: The LCD1602 randomly displays a target value ranging from 0 to 1023. The player needs to adjust the potentiometer within 3 seconds to make the resistance value as close as possible to the target value. After the time is up, the system reads the resistance value and makes a comparison. A green light indicates success, while a red light indicates failure.

Through this class, you will master the basic logic of analog signal reading, random number programming, and multi-module collaborative work.

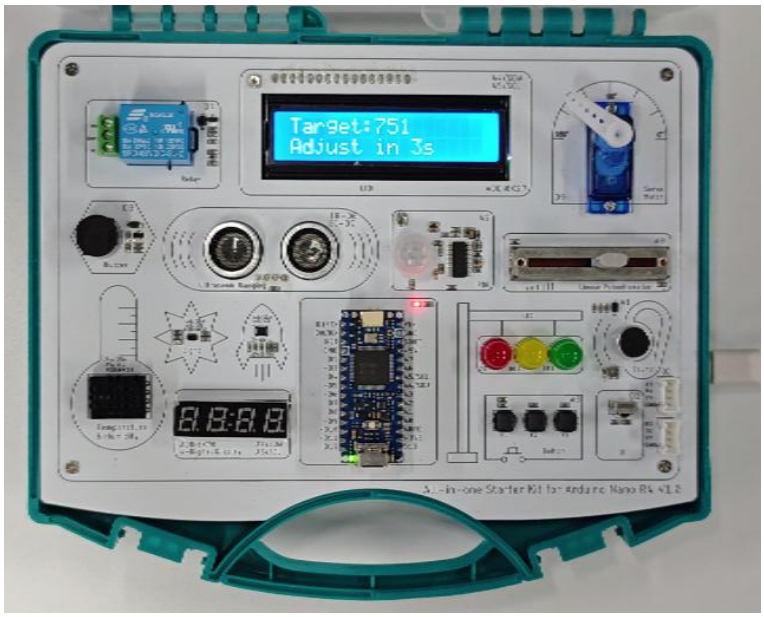
Learning Goals

1. Consolidate the method of reading analog values using analogRead().
2. Learn to use randomSeed to generate random number seeds and learn how to generate random numbers.
3. Display dynamic data on the LCD1602.
4. Complete the making of the sliding potentiometer number guessing game in this section.

Preview of the Result

After uploading the program, the LCD1602 display will show the following process:

1. The LCD will display a random target value (0 to 1023).
2. It will prompt you to adjust the potentiometer within 3 seconds.

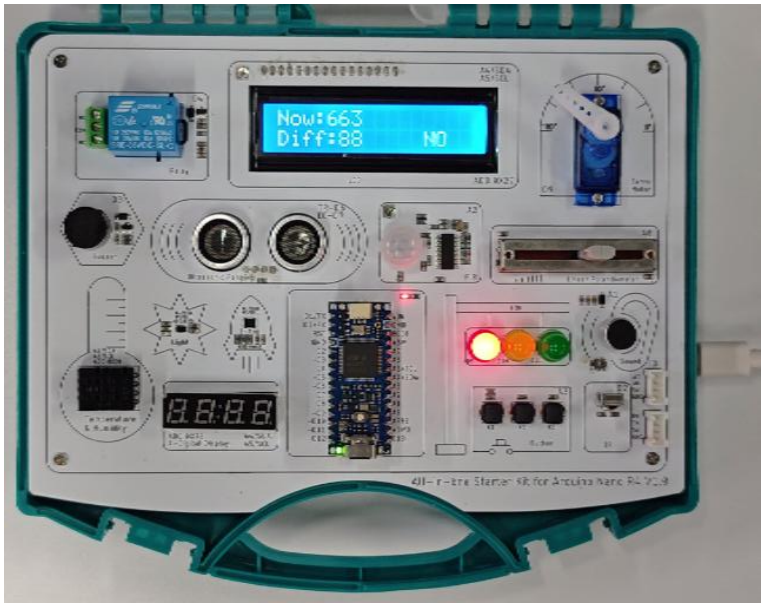


3. At the end of the time period:

- The LCD displays the current value of the sliding rheostat.
- It also shows the difference from the target value (Diff).

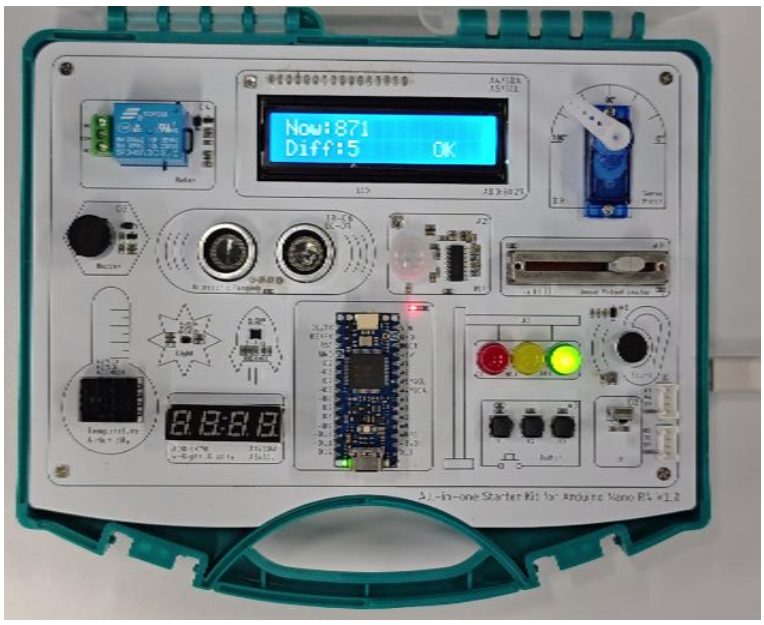
4. If the difference is large:

- The red light is on.
- The LCD displays "NO".



5. If the difference is sufficiently small:

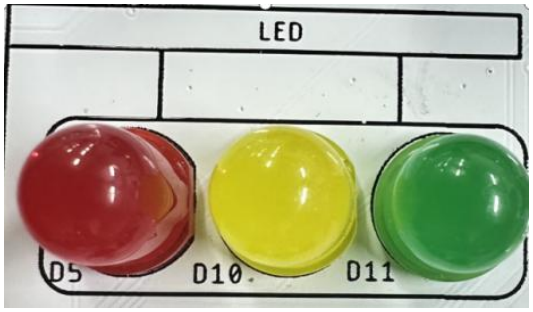
- The green light is on
- The LCD displays "OK"



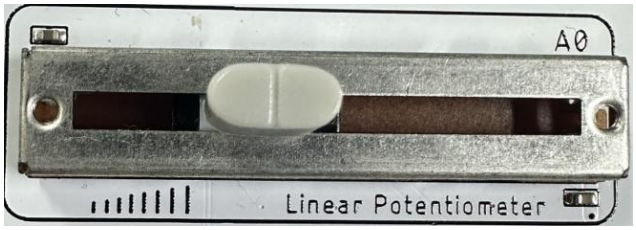
6. The system will display the current score and automatically advance to the next round of the game.

Hardware Used in This Lesson

Three-color LED lights



Linear Potentiometer



LCD1602 display screen

The sliding resistor guessing number game case

Before explaining the code, we can download it. The overall code download link is:
https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-Arduino-Nano-R4/tree/master/lesson_code/19_Sliding_Resistor_Guessing_Game
Open the "19_Sliding_Resistor_Guessing_Game" folder in Arduino IDE and then access the "19_Sliding_Resistor_Guessing_Game.ino" code file.

Key Code Explanation

After opening the code, you will be able to see the code for this lesson:

19_Sliding_Resistor_Guessing_Game.ino

```
#include <Wire.h>
#include <LiquidCrystal_I2C.h>

/* ========== LCD Parameters ========== */
#define COLUMNS 16
#define ROWS    2

LiquidCrystal_I2C lcd(
  PCF8574_ADDR_A21_A11_A01,
  4, 5, 6, 16, 11, 12, 13, 14,
  POSITIVE
);

/* ========== Hardware Pins ========== */
#define POT_PIN A0

#define LED_RED     5
#define LED_GREEN   11

/* ========== Game Variables ========== */
int targetValue = 0;
int currentValue = 0;
int diffValue = 0;
int score = 0;

/* ========== Setup ========== */
void setup()
{
  Serial.begin(115200);

  lcd.begin(COLUMNS, ROWS, LCD_5x8DOTS);
  lcd.clear();
  lcd.backlight();

  randomSeed(analogRead(A1));

  // LED initialization
  pinMode(LED_RED, OUTPUT);
  pinMode(LED_GREEN, OUTPUT);

  digitalWrite(LED_RED, LOW);
  digitalWrite(LED_GREEN, LOW);

```

Introduction to library files

```
#include <Wire.h>
#include <LiquidCrystal_I2C.h>
```

The two lines of code #include <Wire.h> and #include <LiquidCrystal_I2C.h> serve to provide I2C communication capabilities and the control interface for the I2C liquid crystal screen for Arduino: Wire.h is responsible for the underlying I2C bus communication, enabling data transmission between master and slave devices;

LiquidCrystal_I2C.h then provides an advanced encapsulation for the control of the LCD on this basis, allowing us to directly use intuitive functions such as lcd.begin(), lcd.print() to display text, without having to concern about specific communication timings and hardware details. It is the

foundation for the normal operation of the entire LCD.

Parameter definition of LCD1602 display and initialization of I2C interface

```
#define COLUMNS 16      // Number of LCD columns
#define ROWS        2       // Number of LCD rows
LiquidCrystal_I2C lcd(
   PCF8574_ADDR_A21_A11_A01,    // I2C address of PCF8574
   4, 5, 6, 16,                        // RS, RW, EN, Backlight
   11, 12, 13, 14,                    // D4, D5, D6, D7
   POSITIVE                                 // Backlight polarity
);
```

First, by using #define COLUMNS 16 and #define ROWS 2, we clearly inform the program that the current display is a 16-column × 2-row character-type LCD screen. This way, when using lcd.begin(COLUMNS, ROWS) later, the library function can correctly configure the display buffer and cursor system.

Then, a LiquidCrystal_I2C lcd(...) object is created. Here, we do not directly drive the LCD using the parallel port, but indirectly control all the pins of the LCD through the PCF8574 I2C expansion chip.

PCF8574_ADDR_A21_A11_A01 is used to specify the address of this expansion chip on the I2C bus. The following RS, RW, EN, backlight, D4~D7 are the corresponding relationships between the output pins of PCF8574 and the control lines and data lines of the LCD. The last POSITIVE indicates that the backlight is at a high level to light up, and this writing method allows Arduino to stably drive the LCD by only occupying the SDA/SCL two wires, which is a very classic and very resource-saving display solution in embedded projects.

Define pins

```
#define POT_PIN A0
#define LED_RED         5
#define LED_GREEN      11
```

POT_PIN is defined as A0, indicating that the potentiometer (variable resistor) is connected to Arduino's analog input pin A0, used to read analog voltage values ranging from 0 to 1023;

LED_RED and LED_GREEN are respectively defined as digital pins 5 and 11, corresponding to the red and green LEDs.

In this way, in the subsequent code, only by using POT_PIN, LED_RED, and LED_GREEN, the functions of each pin can be intuitively understood.

Define variables

```
int targetValue = 0;
int currentValue = 0;
int diffValue = 0;
int score = 0;
```

These four integer variables are used to store the key data states during the game operation:

targetValue represents the "target value" randomly generated by the system, which is the standard that players need to approach in this round;

currentValue is used to store the current actual analog value read from the potentiometer

(POT_PIN), reflecting the result of the player's operation;
diffValue is the difference between the current value and the target value (usually taking the absolute value), used to determine whether the player is close enough to the target;
score serves as the score variable, used to record the number of times the player successfully hits.

Setup Section

```
void setup()
{
  Serial.begin(115200);

  lcd.begin(COLUMNS, ROWS, LCD_5x8DOTS);
  lcd.clear();
  lcd.backlight();

  randomSeed(analogRead(A1));

  // LED initialization
  pinMode(LED_RED, OUTPUT);
  pinMode(LED_GREEN, OUTPUT);

  digitalWrite(LED_RED, LOW);
  digitalWrite(LED_GREEN, LOW);

  lcd.setCursor(0, 0);
  lcd.print("Pot Guess Game");
  lcd.setCursor(0, 1);
  lcd.print("Get Ready...");
  delay(1500);
  lcd.clear();
}
```

This setup() function is responsible for the initialization of the entire game, which can be regarded as the "preparation stage after the system is powered on".
First, it opens the serial communication using Serial.begin(115200) to facilitate debugging and observing data output;
Then, it initializes the LCD display using lcd.begin(), setting the number of columns, rows and font mode for display, and clears the screen and turns on the backlight using lcd.clear() and lcd.backlight() to ensure a normal display status.

The function of the line of code "randomSeed(analogRead(A1))" is to set a random seed for the random number generator, thereby ensuring that the target values generated in the game have true randomness.

Next, the mode configuration for the red and green LED pins is carried out, and they are initially

set to a low level to ensure that all indicator lights are in the off state when the system starts.

Finally, the startup prompt text "Pot Guess Game" and "Get Ready..." is displayed on the LCD, giving the players a preparation time. After a delay of 1500 milliseconds (1.5 seconds), the screen is cleared to set the stage for the official game loop.

loop

```
void loop()
{
  // Turn off LEDs at the start of each round
  digitalWrite(LED_RED, LOW);
  digitalWrite(LED_GREEN, LOW);

  /* ===== Generate target value ===== */
  targetValue = random(0, 1024);

  lcd.clear();
  lcd.setCursor(0, 0);
  lcd.print("Target:");
  lcd.print(targetValue);

  lcd.setCursor(0, 1);
  lcd.print("Adjust in 3s");

  delay(3000);

  /* ===== Read potentiometer value ===== */
  currentValue = analogRead(POT_PIN);
  diffValue = abs(currentValue - targetValue);

  lcd.clear();
  lcd.setCursor(0, 0);
  lcd.print("Now:");
  lcd.print(currentValue);

  lcd.setCursor(0, 1);
  lcd.print("Diff:");
  lcd.print(diffValue);

  /* ===== Success / Failure ===== */
  if (diffValue <= 10) {
    score++;
    digitalWrite(LED_GREEN, HIGH);      // Success → turn on green LED
    lcd.setCursor(12, 1);
```

```
    lcd.print("OK");
  } else {
    digitalWrite(LED_RED, HIGH);        // Failure → turn on red LED
    lcd.setCursor(12, 1);
    lcd.print("NO");
  }

  delay(2000);

  /* ===== Display score ===== */
  lcd.clear();
  lcd.setCursor(0, 0);
  lcd.print("Score:");
  lcd.print(score);
  lcd.setCursor(0, 1);
  lcd.print("Next round");

  delay(1500);
}
```

This loop() function is not only the "main process" of the game, but also a classic example that integrates random numbers, simulated inputs, mathematical operations, and human-computer interaction.

As soon as the program enters the loop(), it first turns off both the red and green LEDs using digitalWrite(LED_RED, LOW); and digitalWrite(LED_GREEN, LOW);

This is a very important "state reset" step, aiming to prevent the residual success or failure state from the previous round, ensuring that each round starts in a clean initial state.

```
54  void loop()
55  {
56    // Turn off LEDs at the start of each round
57    digitalWrite(LED_RED, LOW);
58    digitalWrite(LED_GREEN, LOW);
59
```

Next comes the generation of the target values. targetValue = random(0, 1024); The random() function built into Arduino is used, which generates a pseudo-random integer ranging from 0 to 1023 (excluding 1024).

This range is exactly the same as the return range of analogRead(), because the analog input of Arduino has a 10-bit precision, corresponding to 0 to 1023.

This design is very ingenious: the target value and the potentiometer reading are in the same "unit", so the comparison later makes sense. Subsequently, the target value is displayed on the LCD, and at the same time, a prompt is given to the player that they have 3 seconds to rotate the potentiometer. The delay(3000) here is equivalent to the "operation window" given to the player.

```
60  /* ===== Generate target value ===== */
61  targetValue = random(0, 1024);
62
63  lcd.clear();
64  lcd.setCursor(0, 0);
65  lcd.print("Target:");
66  lcd.print(targetValue);
67
68  lcd.setCursor(0, 1);
69  lcd.print("Adjust in 3s");
70
71  delay(3000);
72
```

After 3 seconds, the program enters the stage of collecting the player's input. currentValue = analogRead(POT_PIN); It can read the analog voltage on pin A0 and convert it into a digital value ranging from 0 to 1023. This digital value represents the current position that the potentiometer is rotated to.

```
73  /* ===== Read potentiometer value ===== */
74  currentValue = analogRead(POT_PIN);
```

Then this line is very crucial: diffValue = abs(currentValue - targetValue);
Here, the abs() function is used, which serves to calculate the absolute value. Why is it necessary to use the absolute value?
Because the player's value may be greater or smaller than the target value. If we simply subtract them, the result might be negative, and what we really care about is "how much it differs", rather than "the direction of the difference". The abs() function ensures that whether it is higher or lower, the final result will always be a positive error value. This is essential when determining "whether it is close enough to the target".

```
75  diffValue = abs(currentValue - targetValue);
```

Subsequently, the LCD will display the current potentiometer value (Now) and the difference from the target value (Diff), enabling the player to clearly see where they are "falling short".

```
77  lcd.clear();
78  lcd.setCursor(0, 0);
79  lcd.print("Now:");
80  lcd.print(currentValue);
81
82  lcd.setCursor(0, 1);
83  lcd.print("Diff:");
84  lcd.print(diffValue);
85
```

Then comes the logic for determining the outcome:
If (diffValue <= 10), it means that as long as the error is within 10, it is considered that the player has adjusted precisely enough. This 10 actually represents the core parameter of "game difficulty". The smaller the value, the more difficult the game is; the larger the value, the easier it is to succeed.
When successful, the program will provide three types of feedback: the score (score++) increases,

the green LED lights up, and the LCD displays "OK"; when failed, the red LED is lit up and "NO" is displayed. Through visual and numerical dual feedback, the player's perception is strengthened.

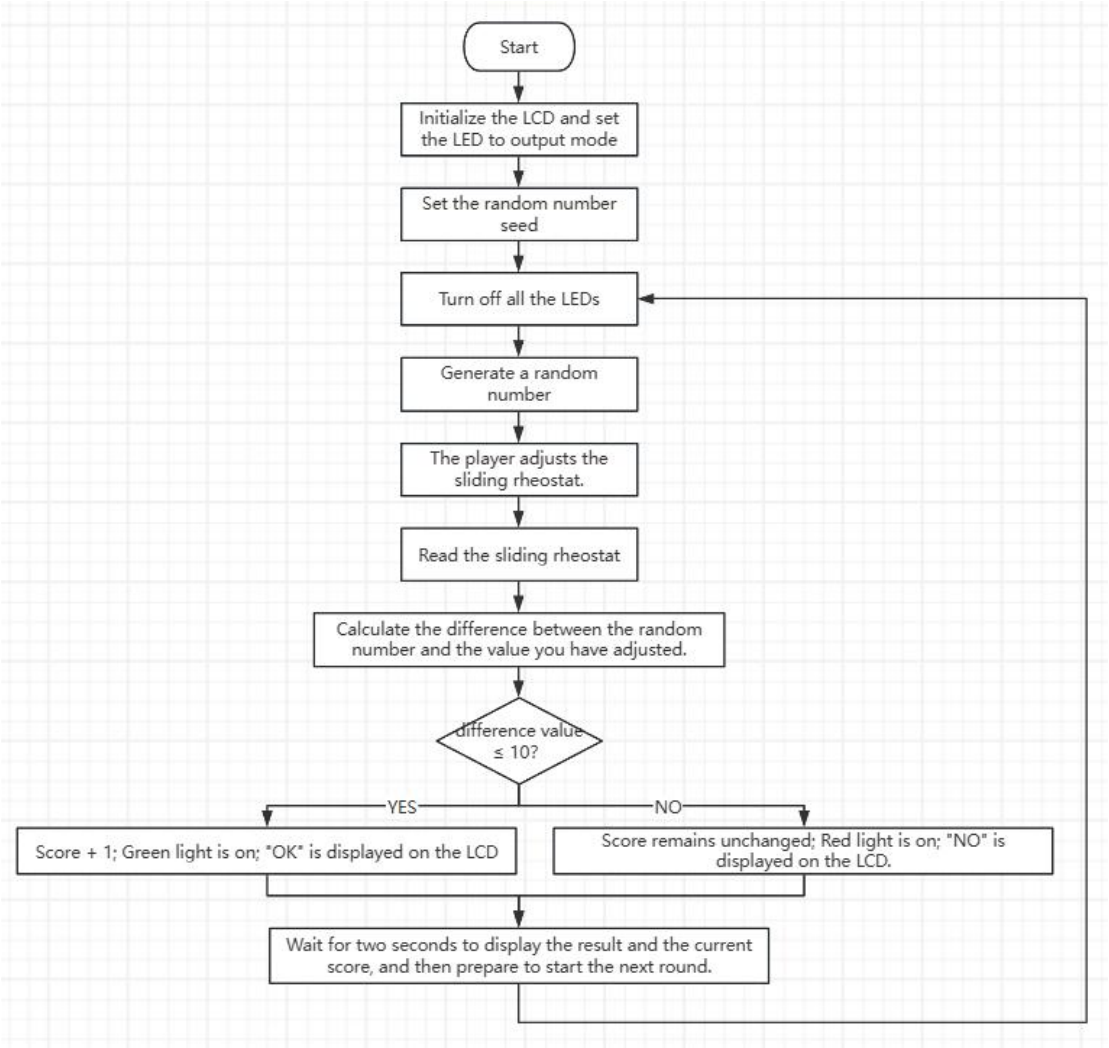
```
/* ===== Success / Failure ===== */
if (diffValue <= 10) {
  score++;
  digitalWrite(LED_GREEN, HIGH);   // Success → turn on green LED
  lcd.setCursor(12, 1);
  lcd.print("OK");
} else {
  digitalWrite(LED_RED, HIGH);     // Failure → turn on red LED
  lcd.setCursor(12, 1);
  lcd.print("NO");
}

```

The final part is the round summary and transition. The program pauses for 2 seconds to allow players to clearly see the result, then clears the screen and displays the current cumulative score as well as the "Next round" prompt. After a delay of 1.5 seconds, it automatically proceeds to the next round. This rhythm design ensures that the game neither goes too fast to be too overwhelming for players nor is too slow to appear sluggish.

```
  delay(2000);

  /* ===== Display score ===== */
  lcd.clear();
  lcd.setCursor(0, 0);
  lcd.print("Score:");
  lcd.print(score);
  lcd.setCursor(0, 1);
  lcd.print("Next round");

  delay(1500);
}
```

Overall, this "loop()" is not merely a simple loop; rather, it fully embodies: the use of random numbers, the collection of analog signals, absolute value mathematical processing, conditional judgment, and human-computer interaction feedback. It is extremely suitable as a classic example for beginners to understand "how a complete Arduino project operates".

Overall code logic flowchart



Upload Program Steps

This project requires additional external libraries. Refer to the "Importing Library Files" section on page 7 and import all the required external libraries at the beginning. If you have already imported them, you can skip this step.

For detailed upload instructions, please refer to the "Upload Steps" on page 8.

Lesson20---Morse Code Decoding Game

Introduction

In this class, we will learn a very interesting and challenging project - the Morse Code decoding game.

Morse Code is a way of transmitting information using long and short signals. It was widely used in telegraph communication in the early days.

In this project, we will use:

- Buttons (Button) to input short and long presses
- LED indicators to show different operation states
- LCD1602 display screen to show the decoded English characters in real time

Through the learning of this class, you will no longer just "press buttons and light up LEDs", but can actually "type" using buttons and see the English words you input on the LCD screen, completing a complete human-computer interaction mini-game.

Learning Goals

1. Understand the basic encoding rules of Morse Code
2. Master the implementation methods of short press / long press in the program
3. Consolidate the use of analogRead to read multiple buttons
4. Consolidate the string display and refresh logic of LCD1602
5. Understand the complete program flow of "input → processing → display"

Preview of the Result

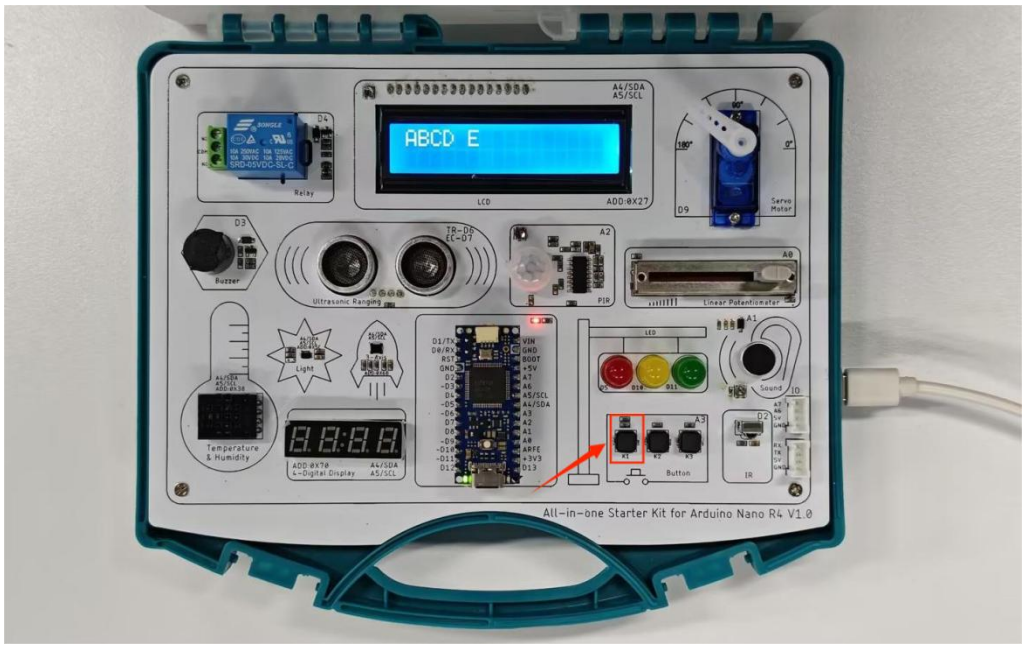
When the program is successfully uploaded, you will see the following effect:

➢ Press the first button

Short press → Enter a "." (Here, the yellow light will turn on, followed by the green light, serving as a prompt)

Long press → Enter a "-" (Here, the yellow light will turn on, followed by the red light, serving as a prompt)

After releasing the button for a while, the program will automatically decode the Morse code into a letter and display it on the LCD.

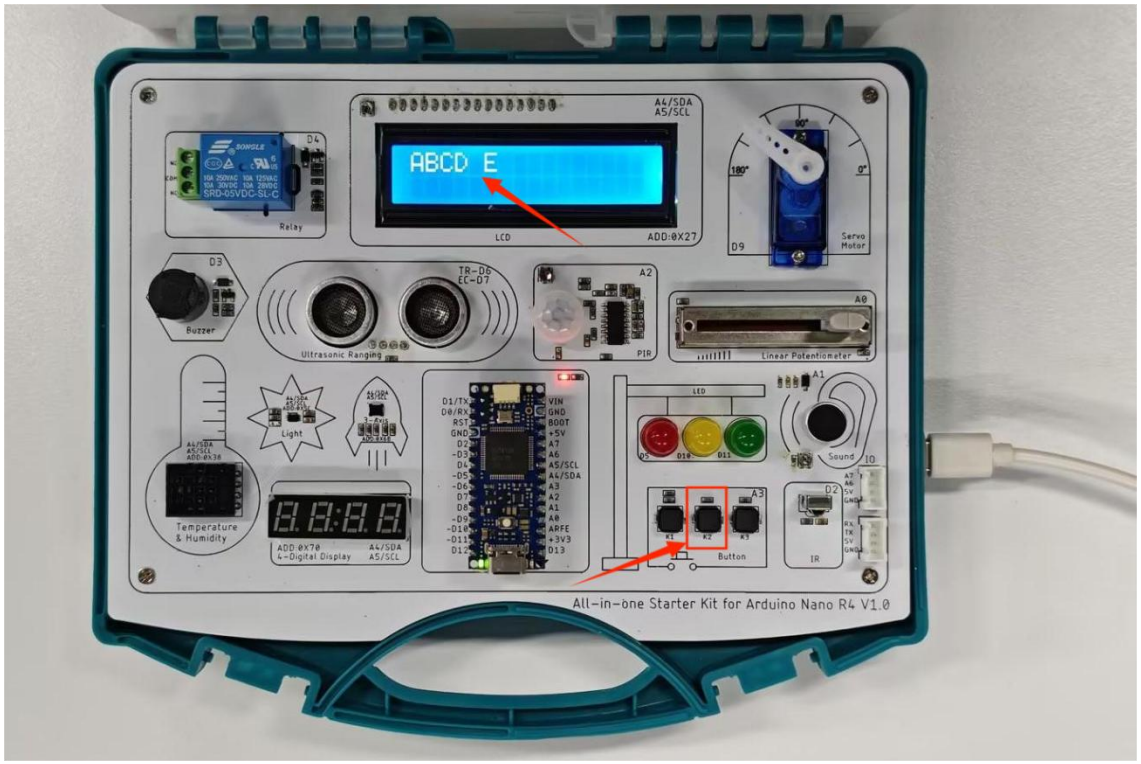


➢ Press the second button

Enter a space

This is used to separate words

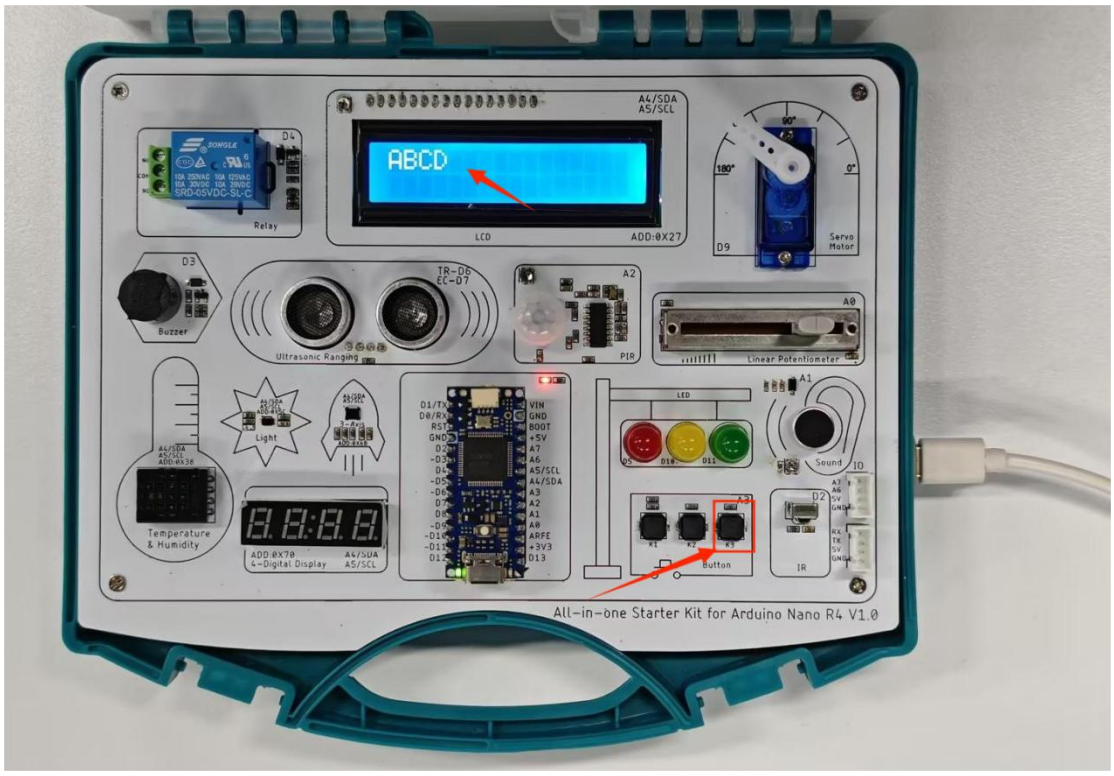
(The green light is on as a prompt)



➢ Use the third button

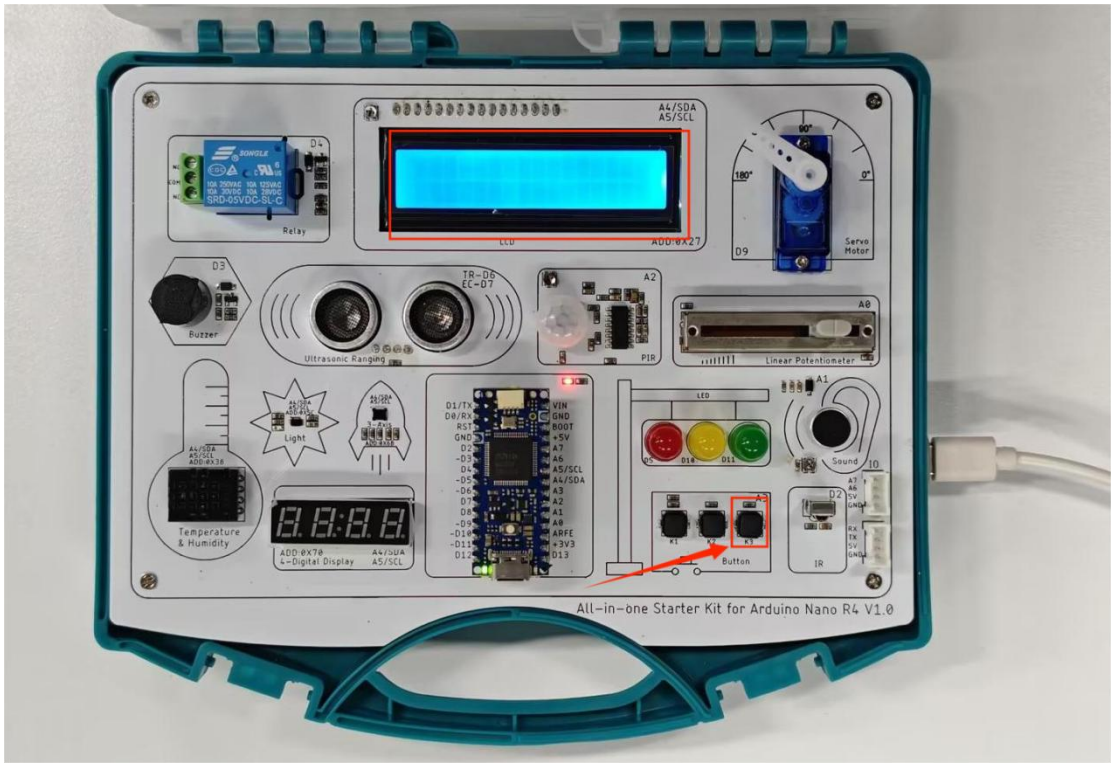
Short press → Delete the last character (the red light is on as a reminder)

(Here, compared with the previous description, it can be seen that the letter E has been deleted)



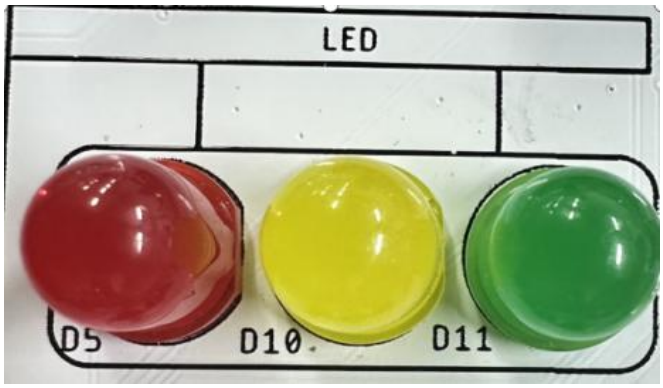
Long press → Clear the entire screen

(The red light flashing serves as a reminder)

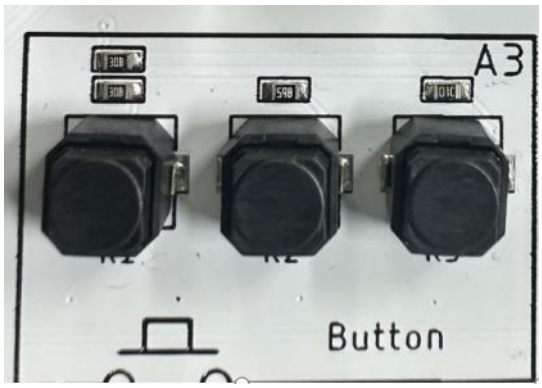


Hardware Used in This Lesson

Three-color LED lights



Three buttons



LCD1602 display screen

Morse Code Game Case

Before explaining the code, we can download it. The overall code download link is:
https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-Arduino-Nano-R4/tree/master/lesson_code/20_Morse_Code_Decoding_Game
Open the "20_Morse_Code_Decoding_Game" folder in Arduino IDE and then access the "20_Morse_Code_Decoding_Game.ino" code file

Key Code Explanation

After opening the code, you will be able to see the code for this lesson:

Introduction to library files

```
#include <Wire.h>
#include <LiquidCrystal_I2C.h>
```

Wire.h is responsible for enabling the I2C bus driver of Arduino, allowing the microcontroller to communicate with external chips through the two lines SDA and SCL;

While LiquidCrystal_I2C.h is a pre-packaged LCD control library built on the Wire library. It hides all the complex I2C timing and underlying signal control, enabling us to directly use the intuitive functions such as lcd.begin(), lcd.setCursor(), and lcd.print() to operate the LCD display content.

In this project, the LCD communicates with Arduino through the PCF8574 I2C expansion chip.

Therefore, both of these libraries are indispensable - the former is responsible for "how the data flows", and the latter is responsible for "what is displayed on the screen", jointly ensuring that the decoding result of Morse code can be stably and clearly displayed on the LCD screen.

Definition of Pins

```
#define LED_RED         5
#define LED_YELLOW    10
#define LED_GREEN      11
#define BUTTON_ANALOG A3
```

"LED_RED", "LED_YELLOW", "LED_GREEN" correspond to the digital pins (5, 10, 11) connected to the red, yellow, and green LEDs respectively. By using this "macro definition" method, in the subsequent code, only the function names of the LEDs need to be written instead of the specific numbers, which greatly improves the readability and maintainability of the program;

While "BUTTON_ANALOG A3" indicates that in this project, the three different functions of the buttons do not each occupy an independent digital pin, but instead share an analog input pin A3 through a resistor divider. Arduino reads different voltage values through "analogRead(A3)", and then determines which button is pressed based on the voltage range.

Parameter definition of LCD1602 display and initialization of I2C interface

```
#define COLUMNS 16      // Number of LCD columns
#define ROWS        2       // Number of LCD rows
LiquidCrystal_I2C lcd(
   PCF8574_ADDR_A21_A11_A01,    // I2C address of PCF8574
   4, 5, 6, 16,                        // RS, RW, EN, Backlight
   11, 12, 13, 14,                   // D4, D5, D6, D7
   POSITIVE                              // Backlight polarity
);
```

First, by using #define COLUMNS 16 and #define ROWS 2, we clearly inform the program that the current display is a 16-column × 2-row character-type LCD screen. This way, when using lcd.begin(COLUMNS, ROWS) later, the library function can correctly configure the display buffer and cursor system.

Then, a LiquidCrystal_I2C lcd(...) object is created. Here, we do not directly drive the LCD using the parallel port, but indirectly control all the pins of the LCD through the PCF8574 I2C expansion chip.

PCF8574_ADDR_A21_A11_A01 is used to specify the address of this expansion chip on the I2C bus. The following RS, RW, EN, backlight, D4~D7 are the corresponding relationships between the output pins of PCF8574 and the control lines and data lines of the LCD. The last POSITIVE indicates that the backlight is at a high level to light up, and this writing method allows Arduino to stably drive the LCD by only occupying the SDA/SCL two wires, which is a very classic and very resource-saving display solution in embedded projects.

Correspondence Table of Morse Code Letters to English Letters

```
const char* morseCodes[] = {
   ".-",       // A
```

```
    "-...",   // B
    "-.-.",   // C
    "-..",    // D
    ".",      // E
    "..-.",   // F
    "--.",    // G
    "....",   // H
    "..",     // I
    ".---",   // J
    "-.-",    // K
    ".-..",   // L
    "--",     // M
    "-.",     // N
    "---",    // O
    ".--.",   // P
    "--.-",   // Q
    ".-.",    // R
    "...",    // S
    "-",      // T
    "..-",    // U
    "...-",   // V
    ".--",    // W
    "-..-",   // X
    "-.--",   // Y
    "--.."    // Z
};

const char letters[] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

The const char* morseCodes[] defines an array of string pointers, where each string element stores the Morse code corresponding to a letter (composed of dots and dashes). The indices of the array strictly correspond to A to Z. For example, index 0 is the Morse code for A (".-"), index 1 is the Morse code for B ("-..."), and so on.

Immediately following is the const char letters[] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ", which defines a character array where the letters themselves are stored in the same order. Thus, in the program, by using the same index value, the input Morse code string and the corresponding English letter can be matched. This "index-aligned lookup table method" is not only logically clear but also has high execution efficiency.

Key parameters of the "Time Determination Rules"

```
const int SHORT_PRESS_THRESHOLD = 500;       // Short press < 500ms → dot
const int CHAR_INTERVAL = 1500;              // Idle time to auto-decode a character
const int CLEAR_SCREEN_THRESHOLD = 1000;     // Long press > 1s → clear screen
const int BLINK_INTERVAL = 200;              // LED blink interval (ms)
```

Essentially, it is informing the program "what each different pressing duration and waiting time

represents":

SHORT_PRESS_THRESHOLD = 500 indicates that when the key is pressed for less than 500 milliseconds, the program will classify it as a "dot (.)", which is the core time threshold for distinguishing dots (.) from dashes (-);

CHAR_INTERVAL = 1500 is used to determine "whether a character has been input", if there is no new input for more than 1.5 seconds after the previous key release, the program will consider that the current Morse sequence has ended and can automatically perform an alphabet decoding;

CLEAR_SCREEN_THRESHOLD = 1000 is used for the function distinction of the clear screen button. When the pressing time exceeds 1 second, it no longer deletes a single character, but directly clears the entire display content;

Finally, BLINK_INTERVAL = 200 controls the rhythm of the LED blinking, making the light feedback clear but not too fast, facilitating human eye recognition.

Identification intervals for each key

```
const int MORSE_BTN_RANGE[2]  = {500, 520};  // Morse input button
const int SPACE_BTN_RANGE[2]  = {680, 690};  // Space button
const int DELETE_BTN_RANGE[2] = {845, 860};  // Delete / clear button
```

Multiple buttons share a single analog input pin. How can the program distinguish which button was actually pressed?

Since the hardware uses a resistor divider (resistor ladder) method, when different buttons are pressed, the voltage value read by analogRead() will fall within different numerical ranges.

MORSE_BTN_RANGE = {500, 520} indicates that when the analog value falls within the range of 500 to 520, the program considers the "Morse input button" to have been pressed, which is the first key;

SPACE_BTN_RANGE = {680, 690} corresponds to the "space bar", which is the second key;

DELETE_BTN_RANGE = {845, 860} represents the "delete / clear screen button", which is the third key.

The program does not determine a fixed value, but rather a range. This is because the actual voltage is affected by factors such as resistance errors and power supply fluctuations. By using this "interval judgment" method, one can use a single analog port to read the states of multiple buttons, which not only saves IO resources but also ensures stable recognition.

The "operational status variables" of the entire Morse input system

```
String currentMorse = "";     // Current Morse sequence being entered
String displayText  = "";     // Full text displayed on LCD (history preserved)

unsigned long buttonPressTime = 0;          // Timestamp when a button is pressed
unsigned long lastButtonReleaseTime = 0;    // Timestamp of last button release

bool morseBtnPressed  = false;     // Morse button state
bool spaceBtnPressed  = false;     // Space button state
```

```
bool deleteBtnPressed = false;      // Delete button state
```

They serve as the "memory" and "judgment basis" of the program, used to store the current input content, time information, and key press status.

The variable "currentMorse" is used to temporarily store the Morse code being input (such as .- and --.), and accumulates continuously before the user finishes a letter;

The variable "displayText" stores the complete text content that has been successfully decoded and displayed on the LCD, even if new characters are continuously input, the previous content will be retained.

The two unsigned long variables "buttonPressTime" and "lastButtonReleaseTime" are specifically used for time measurement. They work together with millis() to determine "how long was pressed" and "how long was released", thereby distinguishing between short presses / long presses and whether the idle time for automatic decoding has been exceeded.

The three bool variables (morseBtnPressed, spaceBtnPressed, deleteBtnPressed) are key status flags, used to record whether a certain key is currently in the "already pressed" state, to prevent a single press from being repeatedly recognized.

Setup Section

```
void setup() {
  Serial.begin(115200);
  // Initialize LED pins
  pinMode(LED_RED, OUTPUT);
  pinMode(LED_YELLOW, OUTPUT);
  pinMode(LED_GREEN, OUTPUT);

  // Initialize LCD
  lcd.begin(COLUMNS, ROWS, LCD_5x8DOTS);
  lcd.clear();
  lcd.backlight();

  // Startup message
  lcd.setCursor(0, 0);
  lcd.print(F("Morse Decoder"));
  lcd.setCursor(0, 1);
  lcd.print(F("Ready"));

  delay(1500);
  lcd.clear();
}
```

The function setup() serves to perform a complete power-on initialization for the entire "Morse Input System", which can be regarded as the "power-on self-check + preparation stage" of the system.

Firstly, Serial.begin(115200) is used to open the serial communication, mainly for debugging and observing the program's running status, making it convenient to check if the system is working properly on the computer side;

Then, pinMode is used to set the red, yellow, and green LED pins to output mode. These LEDs will later be used as input feedback and status indicators.

Next is the initialization of the LCD: lcd.begin(COLUMNS, ROWS, LCD_5x8DOTS) informs the system that this is a 16×2 character LCD and uses the standard 5×8 dot matrix font. lcd.clear() ensures a clean screen, and lcd.backlight() turns on the backlight to make the content visible.

Then, the program displays the startup prompt "Morse Decoder" and "Ready" on the LCD. This is a typical human-computer interaction design, informing the user that the system has successfully started and is in an available state.

Finally, delay(1500) is used to keep the prompt displayed for 1.5 seconds, then clears the screen, preparing for the formal Morse input interface.

loop

```
void loop() {
  // Read the shared analog button value
  int val = analogRead(BUTTON_ANALOG);

  // Handle each button based on analog value
  handleMorseButton(val);
  handleSpaceButton(val);
  handleDeleteButton(val);

  delay(10);    // Simple debounce delay
}
```

This loop() function is the core running loop of the entire Morse decoding system. It can be regarded as the main control brain of the system, which is constantly "scanning and responding to user operations" at all times.

Firstly, analogRead(BUTTON_ANALOG) reads the voltage value from the same analog pin. This is because the three buttons share the same analog pin through a resistor divider (resistor ladder) method. Different buttons correspond to different voltage ranges, and the read val is equivalent to the original basis for determining "which key is pressed currently".

Next, the program does not directly handle the logic in the loop(), but instead passes the same analog value to three functions: handleMorseButton(val), handleSpaceButton(val), and handleDeleteButton(val) to judge and process. This is a very typical and clear modular design concept: the main loop is only responsible for "scheduling", and the specific functions are completed by dedicated functions.

The final delay(10) is a simple software debounce delay, used to reduce misjudgments caused by button jitter and analog reading jitter, while not significantly affecting the system response speed.

handleMorseButton function

```
void handleMorseButton(int val) {
  bool active = (val >= MORSE_BTN_RANGE[0] && val <= MORSE_BTN_RANGE[1]);

  // Button pressed
```

```
    if (active && !morseBtnPressed) {
        morseBtnPressed = true;
        buttonPressTime = millis();
        digitalWrite(LED_YELLOW, HIGH);    // Yellow LED indicates button hold
    }
    // Button released
    else if (!active && morseBtnPressed) {
        morseBtnPressed = false;
        digitalWrite(LED_YELLOW, LOW);

        unsigned long duration = millis() - buttonPressTime;
        lastButtonReleaseTime = millis();

        // Determine dot or dash
        if (duration < SHORT_PRESS_THRESHOLD) {
            currentMorse += ".";
            digitalWrite(LED_GREEN, HIGH);
        } else {
            currentMorse += "-";
            digitalWrite(LED_RED, HIGH);
        }

        delay(100);
        digitalWrite(LED_GREEN, LOW);
        digitalWrite(LED_RED, LOW);
    }

    // Auto-decode if idle timeout reached
    if (!active && currentMorse.length() > 0) {
        if (millis() - lastButtonReleaseTime > CHAR_INTERVAL) {
            char c = decodeMorse(currentMorse);
            displayText += c;
            updateLCDDisplay();
            resetMorseInput();
        }
    }
}
```

This entire "handleMorseButton(int val)" function actually implements a complete "Morse input state machine": it is responsible for identifying "whether the Morse key has been pressed", "how long it has been pressed", "whether it is a dot or a dash", and "when it can be automatically decoded into a letter".

At the beginning, by "val >= MORSE_BTN_RANGE[0] && val <= MORSE_BTN_RANGE[1]",

the analog voltage value is converted into a logical state "active", which is a crucial step from analog button → digital state. Essentially, it replaces "digital pin high/low level" with "voltage

range". When "active && !morseBtnPressed" is detected, it indicates that the button has just been pressed. This is an edge detection, and the program immediately records the pressing time "buttonPressTime = millis()" and turns on the yellow light to provide the user with a real-time feedback of "the key is being held down".

```
149  void handleMorseButton(int val) {
150    bool active = (val >= MORSE_BTN_RANGE[0] && val <= MORSE_BTN_RANGE[1]);
151
152    // Button pressed
153    if (active && !morseBtnPressed) {
154      morseBtnPressed = true;
155      buttonPressTime = millis();
156      digitalWrite(LED_YELLOW, HIGH);  // Yellow LED indicates button hold
157    }
```

When "!active && morseBtnPressed" is encountered, it indicates that the button has just been released. This is a falling edge detection. At this point, the duration "duration" of the press is calculated using "millis() - buttonPressTime". This is the physical basis for distinguishing between "dots" and "dashes": shorter than "SHORT_PRESS_THRESHOLD" is considered a dot (.), longer than the threshold is considered a dash (-), and "." or "-" is appended to the "currentMorse" string respectively. At the same time, a brief flashing of a green or red light is used as visual feedback. Then, "lastButtonReleaseTime" is recorded to prepare for the subsequent "automatic decoding timeout judgment".

```
158    // Button released
159    else if (!active && morseBtnPressed) {
160      morseBtnPressed = false;
161      digitalWrite(LED_YELLOW, LOW);
162
163      unsigned long duration = millis() - buttonPressTime;
164      lastButtonReleaseTime = millis();
165
166      // Determine dot or dash
167      if (duration < SHORT_PRESS_THRESHOLD) {
168        currentMorse += ".";
169        digitalWrite(LED_GREEN, HIGH);
170      } else {
171        currentMorse += "-";
172        digitalWrite(LED_RED, HIGH);
173      }
174
175      delay(100);
176      digitalWrite(LED_GREEN, LOW);
177      digitalWrite(LED_RED, LOW);
178    }
```

The last piece of code embodies the time-driven human-computer interaction design concept: When there are no keys pressed and a Morse code has been input, if the idle time from the last release of the key to the present exceeds "CHAR_INTERVAL", it is considered that the input of a complete character has ended. At this point, the "decodeMorse()" function is called to translate the Morse sequence such as ".-" into letters, and then append it to "displayText", refresh the LCD,

and through "resetMorseInput()" clear the current input status, preparing for the next character.

```
180     // Auto-decode if idle timeout reached
181     if (!active && currentMorse.length() > 0) {
182       if (millis() - lastButtonReleaseTime > CHAR_INTERVAL) {
183         char c = decodeMorse(currentMorse);
184         displayText += c;
185         updateLCDDisplay();
186         resetMorseInput();
187       }
188     }
189   }
```

Overall, this function integrates analog input determination, edge detection, time measurement, state machine control, and user feedback. It is the most core and "engineering-like" part of the entire Morse code decoding system.

handleSpaceButton function

```
void handleSpaceButton(int val) {
  bool active = (val >= SPACE_BTN_RANGE[0] && val <= SPACE_BTN_RANGE[1]);

  if (active && !spaceBtnPressed) {
    spaceBtnPressed = true;
    digitalWrite(LED_GREEN, HIGH);    // Green LED indicates space key
  }
  else if (!active && spaceBtnPressed) {
    spaceBtnPressed = false;
    digitalWrite(LED_GREEN, LOW);

    displayText += " ";
    updateLCDDisplay();
    delay(100);
  }
}
```

This handleSpaceButton(int val) function implements the complete input logic for the "space bar". The logic is very similar to the previous Morse key, but it is simpler in functionality. The core objective is just one: to insert a space into the display text at the appropriate time.

First, by using val >= SPACE_BTN_RANGE[0] && val <= SPACE_BTN_RANGE[1]

the analog input value is converted into a boolean variable active. This is a typical determination of the analog key range, used to determine whether the "space bar" is currently being pressed.

When detecting active && !spaceBtnPressed, it indicates that the space bar has just been pressed. This is a key press detection event. The program immediately sets spaceBtnPressed to true and turns on the green LED, serving as user feedback to clearly inform the user that "the space bar is being held down".

```
195  void handleSpaceButton(int val) {
196    bool active = (val >= SPACE_BTN_RANGE[0] && val <= SPACE_BTN_RANGE[1]);
197
198    if (active && !spaceBtnPressed) {
199      spaceBtnPressed = true;
200      digitalWrite(LED_GREEN, HIGH);  // Green LED indicates space key
201    }
```

When entering the !active && spaceBtnPressed branch, it indicates that the key has been released. This is a detection of key release. At this point, the program turns off the green light and actually executes the "space input" action logically: by calling displayText += " " to append a space to the current displayed string, and then calling updateLCDDisplay() to refresh the LCD, so that the screen content is updated immediately.

```
202    else if (!active && spaceBtnPressed) {
203      spaceBtnPressed = false;
204      digitalWrite(LED_GREEN, LOW);
205
206      displayText += " ";
207      updateLCDDisplay();
208      delay(100);
209    }
210  }
```

Here, the pressing time is not used for judgment. This is because the space key itself does not distinguish between short presses and long presses; it only cares about a complete "press → release" event.

The final delay(100) serves as a simple debounce and anti-flicker function, preventing the insertion of multiple spaces consecutively due to simulated value jitter or an unclean finger lift.

handleDeleteButton function

```
void handleDeleteButton(int val) {
  bool active = (val >= DELETE_BTN_RANGE[0] && val <= DELETE_BTN_RANGE[1]);

  if (active && !deleteBtnPressed) {
    deleteBtnPressed = true;
    buttonPressTime = millis();
    digitalWrite(LED_RED, HIGH);
  }
  else if (deleteBtnPressed) {
    unsigned long pressTime = millis() - buttonPressTime;

    // Long press: clear entire display
    if (pressTime >= CLEAR_SCREEN_THRESHOLD) {
      displayText = "";
      lcd.clear();
      blinkRedLED(3);
      deleteBtnPressed = false;
```

```
        digitalWrite(LED_RED, LOW);
    }
    // Short press: delete last character
    else if (!active) {
        if (displayText.length() > 0) {
            displayText.remove(displayText.length() - 1);
            updateLCDDisplay();
        }
        deleteBtnPressed = false;
        digitalWrite(LED_RED, LOW);
        delay(100);
    }
  }
}
```

This handleDeleteButton(int val) function implements a delete button logic with "short press / long press distinction". It comprehensively utilizes three core knowledge points: simulated key detection, state machine thinking, and time measurement.

The function also initially converts the simulated input value through val >= DELETE_BTN_RANGE[0] && val <= DELETE_BTN_RANGE[1] into active, which is used to determine whether the delete button is in the pressed state; when detecting active && !deleteBtnPressed, it indicates that the delete button has just been pressed, which is a trigger of a press along. The program immediately sets deleteBtnPressed to true, and uses buttonPressTime = millis() to record the time when the button was pressed, while also lighting up the red LED to provide an intuitive feedback to the user that "the deletion operation is being executed".

```
217  void handleDeleteButton(int val) {
218    bool active = (val >= DELETE_BTN_RANGE[0] && val <= DELETE_BTN_RANGE[1]);
219
220    if (active && !deleteBtnPressed) {
221      deleteBtnPressed = true;
222      buttonPressTime = millis();
223      digitalWrite(LED_RED, HIGH);
224    }
```

The following else if (deleteBtnPressed) branch indicates: As long as the delete button is in the "pressed" state, the program will continuously check how long it has been pressed. Here, pressTime is calculated by using millis() - buttonPressTime, which is a typical method for measuring the duration of a key press based on system time. If pressTime >= CLEAR_SCREEN_THRESHOLD, it means this is a long press operation. The program then clears displayText, clears the LCD screen, and calls blinkRedLED(3) to make the red light blink three times as a strong prompt for "full screen clear success", and then resets the deleteBtnPressed state and turns off the red light to prevent repeated triggering.

```
225     else if (deleteBtnPressed) {
226       unsigned long pressTime = millis() - buttonPressTime;
227
228       // Long press: clear entire display
229       if (pressTime >= CLEAR_SCREEN_THRESHOLD) {
230         displayText = "";
231         lcd.clear();
232         blinkRedLED(3);
233         deleteBtnPressed = false;
234         digitalWrite(LED_RED, LOW);
235       }
```

If the long press threshold is not reached and it is detected that !active, this indicates a short press and the finger has been released. At this point, the program only deletes the last character: First, check if the string is empty to avoid out-of-bounds errors. Then, use displayText.remove(displayText.length() - 1) to precisely delete the last character. Subsequently, call updateLCDDisplay() to refresh the display. Finally, reset the state, turn off the red light, and add a short delay(100) as debouncing processing.

```
236       // Short press: delete last character
237       else if (!active) {
238         if (displayText.length() > 0) {
239           displayText.remove(displayText.length() - 1);
240           updateLCDDisplay();
241         }
242         deleteBtnPressed = false;
243         digitalWrite(LED_RED, LOW);
244         delay(100);
245       }
246     }
247   }
```

decodeMorse function

```
char decodeMorse(String morse) {
  for (int i = 0; i < 26; i++) {
    if (morse == morseCodes[i]) return letters[i];
  }
  return '?';    // Unknown Morse sequence
}
```

The function "decodeMorse(String morse)" is the core function for converting "Morse code → letters" in the entire program. Essentially, it implements a lookup table + sequential matching process.

The input parameter "morse" of the function is the entire string of Morse signals that the user has just input (for example, ".-" and "-..."). Using the "String" type here is for the convenience of string comparison;

Inside the function, through a for loop (int i = 0; i < 26; i++) that iterates over the predefined "morseCodes[]" array, each item in this array corresponds one-to-one to the English alphabet (index 0 corresponds to A, 1 to B, and so on). Therefore, i here serves both as the array index and implicitly represents the position of a letter.

When the condition "if (morse == morseCodes[i])" is met, using the overloaded "==" operator of the Arduino "String" class, the content of "the currently input Morse code" is compared with "a certain standard Morse code in the dictionary". Once a match is successful, it immediately returns "letters[i]", returning the corresponding letter to the caller. Here, "letters[]" is exactly the character table with the same indices as "morseCodes[]".

If no matching items are found after the loop ends, it indicates that the Morse code sequence entered by the user is not a valid or defined letter (such as an input error, abnormal rhythm, etc.). At this point, the function returns '?' as a fallback result. This is a design for error handling and exception indication, which can prevent the program from crashing and clearly inform the user "The current input cannot be recognized" at the display level.

resetMorseInput function

```
void resetMorseInput() {
  currentMorse = "";
  lastButtonReleaseTime = millis();
}
```

Firstly, currentMorse = ""; This line clears the current input string of Morse code. Logically, this is equivalent to "After decoding one letter, the next input should start from the beginning". If this step is not performed, the dots and dashes in the subsequent input will mistakenly overlap onto the previous letter, resulting in a chaotic decoding result.

Then, this line "lastButtonReleaseTime = millis(); " is not written randomly. It utilizes the system time function millis() of Arduino to record "the current moment" as the most recent time of button release. The purpose of this is to allow the subsequent "idle time judgment" (CHAR_INTERVAL) to restart the timer, avoiding the time interval just decoded being mistakenly judged as the new timeout condition.

updateLCDDisplay function

```
void updateLCDDisplay() {
  lcd.clear();
  if (displayText.length() <= COLUMNS) {
    lcd.setCursor(0, 0);
    lcd.print(displayText);
  } else {
    lcd.setCursor(0, 0);
    lcd.print(displayText.substring(0, COLUMNS));
    lcd.setCursor(0, 1);
    lcd.print(displayText.substring(COLUMNS));
  }
}
```

This updateLCDDisplay() function embodies the complete design concept of "safely and controllably mapping the string state in memory to the 16×2 LCD screen".

The function begins by using lcd.clear(), which is a very important display operation. Its significance is not "lazy screen clearing", but to prevent residual characters from remaining on the screen when the displayed content is longer than the current content, ensuring that the LCD

is refreshed with a "clean picture" each time.

Then, through displayText.length(), the length of the current string to be displayed is determined. This step reflects the program's active adaptation to the physical limitations of the hardware: since each row of the LCD can display at most COLUMNS (here 16) characters, it is necessary to first determine if the string can fit within one row. If the length is less than or equal to 16, the cursor is moved to the starting position of the first row using lcd.setCursor(0, 0) and the entire string is output, which corresponds to the simplest and most intuitive display situation;
When the string length exceeds 16, the code enters the else branch. By using substring(0, COLUMNS) and substring(COLUMNS), a logically long string is split into two segments. The first 16 characters are displayed on the first row, and the remaining characters are displayed on the second row.

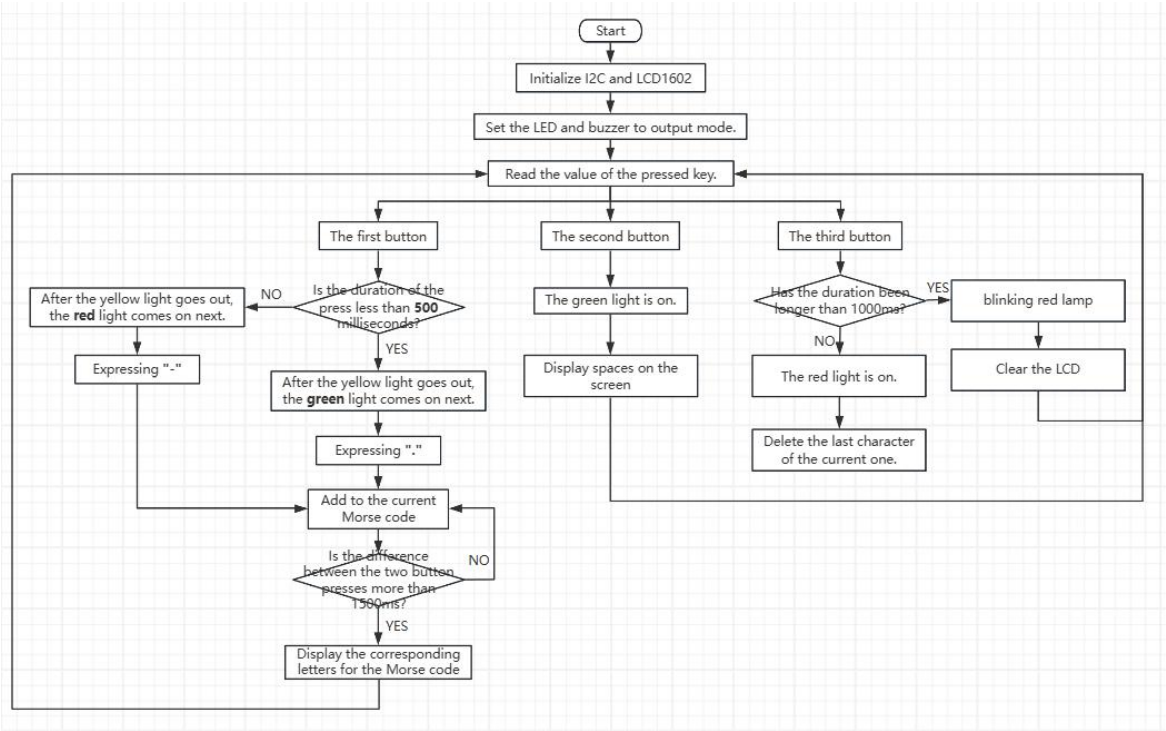
blinkRedLED function

```
void blinkRedLED(int times) {
  for (int i = 0; i < times; i++) {
    digitalWrite(LED_RED, HIGH);
    delay(BLINK_INTERVAL);
    digitalWrite(LED_RED, LOW);
    delay(BLINK_INTERVAL);
  }
}
```

The existence of the function parameter "times" is extremely crucial. It is not simply set to flash a few times, but abstracts the "number of flashes" into a variable. This way, in different scenarios (such as indicating an error, indicating success, or warning the user), the same piece of code can be reused by passing in different values, demonstrating a good modularization concept.

Inside the function, a for loop is used, which is a typical loop control structure of the order type. int i = 0; i < times; 'i++' indicates that the entire flashing process will be executed 'times' times, and each loop corresponds to a complete "on → off" cycle. Within the loop body, 'digitalWrite(LED_RED, HIGH)' sets the red LED pin to a high level, meaning it supplies voltage to the LED, causing it to light up; immediately following is 'delay(BLINK_INTERVAL)', which is used to maintain the illuminated state for a fixed period of time. This time is uniformly controlled by the constant 'BLINK_INTERVAL' to ensure the consistency of the flashing rhythm.
Subsequently, 'digitalWrite(LED_RED, LOW)' pulls the pin low, cutting off the current, causing the LED to turn off. Then, another identical 'delay(BLINK_INTERVAL)' is used to maintain the off state, thus "on + off" together form a complete, visually recognizable flashing cycle.

Overall code logic flowchart



Upload Program Steps

This project requires additional external libraries. Refer to the "Importing Library Files" section on page 7 and import all the required external libraries at the beginning. If you have already import ed them, you can skip this step.

For detailed upload instructions, please refer to the "Upload Steps" on page 8.