Microelectronic Workshop Projects

Yi Liu

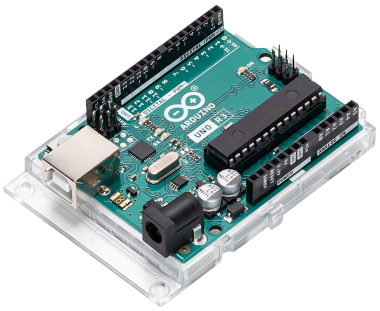
Contents

1 Arduino Introduction

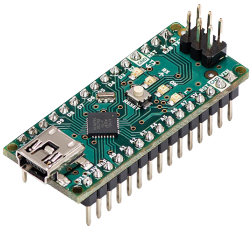
1.1 What is Arduino

Figure 1: Arduino Logo

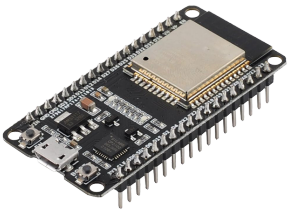
Arduino is an open-source electronics platform that consists of hardware, such as the Arduino UNO and Arduino Nano, which can be programmed using the Arduino IDE with a simplified version of C++. These boards are capable of reading inputs from sensors, processing data, and controlling outputs like motors, LEDs, and displays; and more advanced options such as ESP32 series, which offer enhanced performance and wireless connectivity. Due to its ease of use, affordability, and versatility, Arduino is widely used in DIY projects, automation, and prototyping.



(a) Arduino UNO R3



(b) Arduino Nano



(c) ESP32

Figure 2: Three widely used Arduino Boards

Arduino was created in 2005 in Ivrea, Italy by a group of academics. The name Arduino

comes from a bar in Ivrea, called *Antica Caffetteria Arduino*. This bar was a favorite gathering place for the project founders. The bar itself was named after **King Arduin of Ivrea**, who ruled Italy in the early 11th century. When the creators of Arduino were developing their open-source hardware platform, they decided name it after this local landmark. The name stuck, and today Arduino is one of the most well-known platforms for hobbyist and professional electronics projects.

Figure 3: Arduino Team

Figure 4: Antica Caffetteria Arduino

1.2 Hardware — Arduino Board Components

In simple word, Arduino is an embedded development platform. An embedded system is designed for a specific task, such as controlling a washing machine, managing a car's braking system, or operating a medical device like a pacemaker. In contrast, a regular system like a personal computer is built for general-purpose use, allowing users to run multiple applications such as web browsing, gaming, and video editing.

Hardware differences are significant between the two. Embedded systems use microcontrollers (MCUs) or microprocessors (MPUs) with minimal memory and power consumption, often including dedicated interfaces for sensors and motors. Regular systems, on the other hand, have powerful CPUs, large amounts of RAM, and extensive storage, supporting ex-

ternal peripherals like keyboards, monitors, and printers.

The operating system also varies. Many embedded systems run on a Real-Time Operating System (RTOS) like FreeRTOS, while some operate without an OS, using only firmware. Regular system rely on full-featured operating system like Windows, Linux, macOS, which support multitasking and extensive user interaction.

User interaction is another major difference. Embedded systems often work in the background with little or no direct user control. For example, a car's anti-lock braking system (ABS) activates automatically without the driver needing to interact with it. In contrast, regular system are designed for direct user engagement, with graphical interfaces and input devices such as keyboards and mice.

Power consumption and performance also set them apart. Embedded systems are optimized for low power use, making them ideal for battery-operated devices or situations where energy efficiency is critical. Regular systems consume more power but deliver higher performance, often requiring active cooling systems to prevent overheating.

Software updates behave differently in both types of systems. Embedded systems often have limited or no updates, as they are designed to function reliably without frequent modifications. Regular systems, on the other hand, frequently receive updates to improve security, performance, and features.

As figure (5) and figure (6) showing, the basic parts of Arduino boards are highlighted. In Arduino boards, the main component is the microcontroller, a pre-programmed with a bootloader that allows users to directly upload the program to Arduino through USB without the need for an external programmer. Another important component is crystal oscillator which provides clock source for a microcontroller to work.

For the microcontroller, the basic logic is controlling the input and output of the pins. Arduino boards have digital I/O (Input/Output) and analog inputs. The digital I/O can either receive or send digital signals, which are typically binary — either **HIGH** (1) or **LOW** (0). These pins can be used to control simple devices like LEDs or read signals from switches or sensors. Analog input refers to pins that can read continuous signals, such as voltage levels that vary smoothly over a range. These signals are typically used for sensors like temperature sensors and light sensors where value can change gradually and isn't limited to just HIGH or LOW states. The microcontroller converts this analog signal into a digital value using an **ADC** (Analog-to-Digital Converter) for processing.

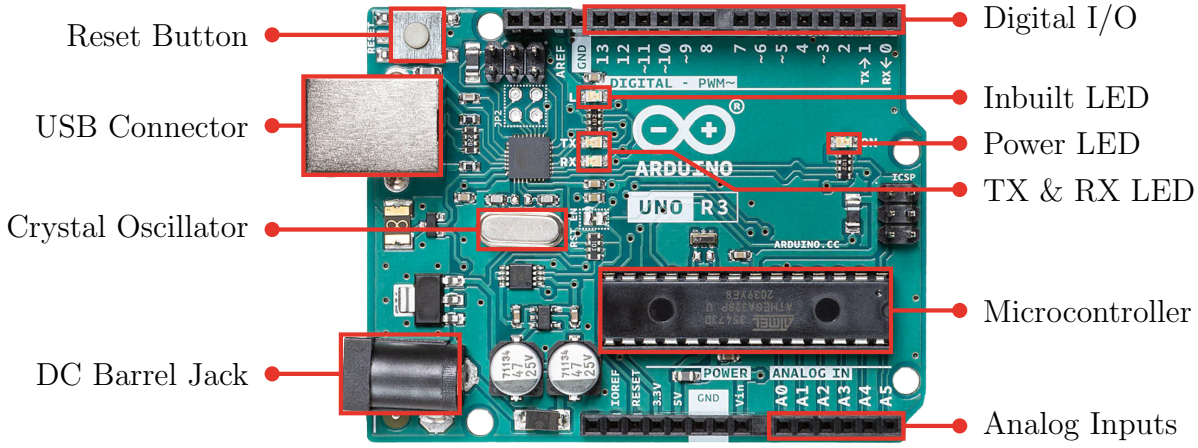


Figure 5: Annotated Diagram of Arduino UNO R3 Key Components

In figure (5) and figure (6), there are also pins named **5V**, **3.3V** or **3V3**, **Vin**, and **GND**. Each of these serves a specific purpose in providing or regulating power for the board and connected components.

The 5V pin provides a regulated 5V output that can be used to power external components such as sensors, modules, and LEDs that require 5V.

Similarly, 3.3V or 3V3 pin provides a regulated 3.3V output, which is useful for powering components that operate at 3.3V, such as certain sensors and communication modules.

Vin pin is the input power pin. To power the Arduino through an external power source instead of USB, this is where the voltage enters the board. For Arduino UNO boards, there is a DC Barrel Jack which allows users to power the boards with external 12V power supply.

GND pins are the reference point for all voltage levels in the circuit. Any component connected to the board must have its negative connection attached to GND to complete the circuit.

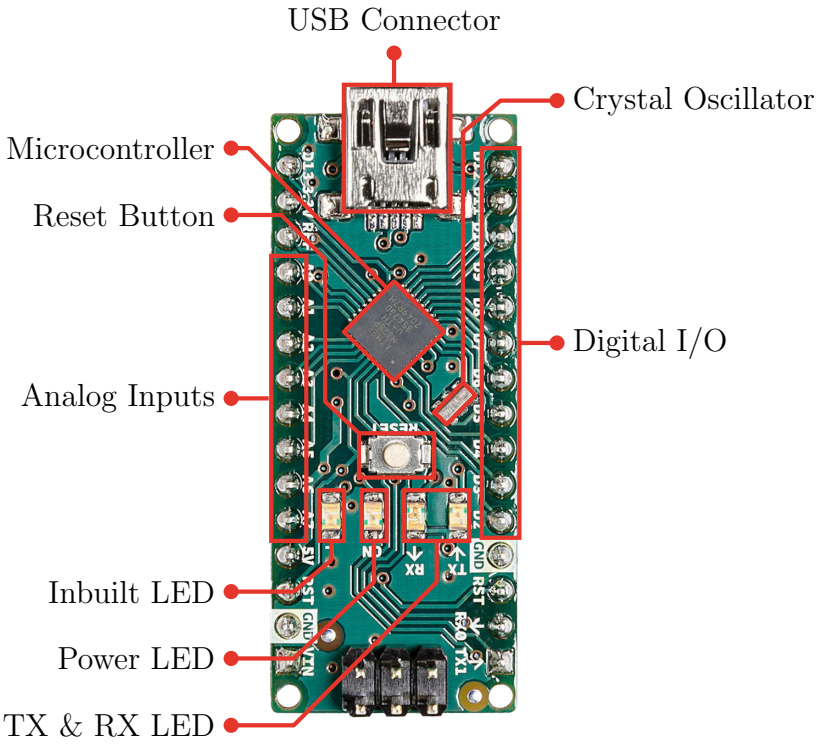


Figure 6: Annotated Diagram of Arduino Nano Key Components

Most Arduino development boards have 4 LEDs onboard. One is used as power indicator and two are used to show the activity of the RX (receive) and TX (transmit) pin. The other one is tied to one of pins (pin 13 for Arduino UNO and Arduino Nano) which can be used to test the Arduino board or simply as an indicator.

Last but not least is the reset button. The reset button is used to restart the microcontroller

and reset the program running on it. When pressed, it momentarily disconnects the reset pin from power, forcing the microcontroller to stop execution and restart from the beginning of the uploaded code.

1.3 Software — Arduino IDE

The Arduino IDE (Integrated Development Environment) is a software application used for writing, compiling, and uploading code to Arduino microcontrollers. It provides a simple interface for programming Arduino boards using the C++ — based Arduino programming language. The Arduino IDE can be found and downloaded from their website: `https://www.arduino.cc/en/software/#ide`.

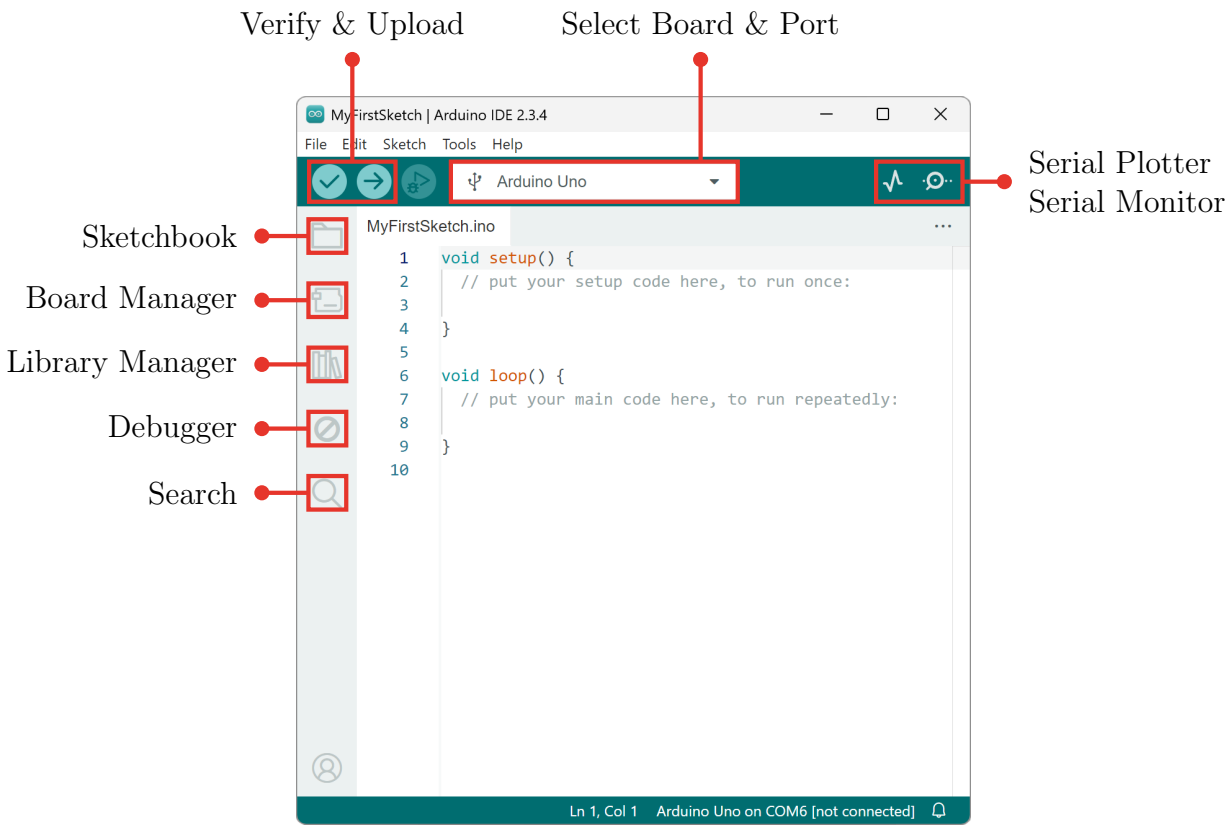


Figure 7: Annotated Interface of Arduino IDE 2.3.4

The interface of Arduino IDE is as shown in figure (7).

- **Verify/Upload**: Compiling and uploading code to Arduino board
- **Select Board & Port**: This drop-down menu allows users select the connected Arduino board model and the communication port used for uploading code.
- **Serial Plotter / Serial Monitor**: The graph-like icon opens the Serial Plotter, which visualizes real-time data from the board; the magnifying glass with dots opens the Serial Monitor, which displays real-time textual data from the board (e.g. `Serial.print()`).
- **Sketchbook**: This icon provides access to saved Arduino sketches.
- **Board Manager**: This section allows users to install and manage Arduino boards.
- **Library Manager**: Enables users to install, update, and manage external libraries required for Arduino projects.
- **Debugger**: A feature that helps debug the code by setting breakpoints and stepping through execution (only available for some supported boards).
- **Search**: Allow users to search for specific lines, variables, or functions within the code for easier navigation.

1.4 First Arduino Program

When opening the Arduino IDE for the first time, the Arduino IDE will use the default script as shown below. An Arduino C++ program follows a simple structure with two main functions:

0_BasicStructure.ino

```
1    void setup() {
2        // put your setup code here, to run once:
3    }
4
5    void loop() {
6        // put your main code here, to run repeatedly:
7    }
```

In every Arduino program, there are two essential functions: `setup()` and `loop()`.

```
1    void setup() {}
```

The `void` means this function does not have return any information or values after it finishes. This is where you typically initialize configurations such as setting pin modes, starting serial communication, or setting up sensors. It only executes one time at the beginning of the program.

```
5    void loop() {}
```

This function is called automatically, immediately, and repeatedly after `setup()`. This is the core of the program where you define tasks that you want the Arduino to perform over and over, such as reading sensor values, controlling LEDs, motors, or handling communication.

Now, after understanding the basic structure of Arduino program and how Arduino program works, let's start our first Arduino program. For most programming languages, the first thing you learn is printing "Hello World!" on the screen. But in the world of Arduino, our "Hello World!" does not just sit there — it **glows**. Below is our first Arduino program — lighting

up an LED.

1_LightUpLED.ino

```
1    const int ledPin = 13;
2
3    void setup() {
4        pinMode(ledPin, OUTPUT);
5    }
6
7    void loop() {
8        digitalWrite(ledPin, HIGH);
9    }
```

Let's explain our first Arduino program in details.

```
1    const int ledPin = 13;
```

As chapter 1.2 introduces both Arduino UNO and Arduino Nano has inbuilt LED, which attached to pin 13. This line declares a **constant value** `ledPin` and assigns it the value `13`.

```
4    pinMode(ledPin, OUTPUT);
```

The function `pinMode()` takes two parameters: the pin number to be controlled and its mode (state). Since pin 13 has already been defined as `ledPin` and the voltage signal need to be output, `ledPin` and `OUTPUT` are the two parameters to be passed.

Syntax:

```
    pinMode(pin, mode)
```

- **pin**: `int`, the pin number you want to configure.
- **mode**: The mode of operation of the pin. It can be one of following:
  - INPUT: Configures the pin as an input.
  - OUTPUT: Configures the pin as an output.
- **returns**: `void`

```
8     digitalWrite(ledPin, HIGH);
```

Similar to `pinMode()`, `digitalWrite()` also takes two parameters: pin number to be controlled and voltage signal. To light up this LED, the voltage signal on `ledPin` need to be set as `HIGH`.

Syntax:

```
      pinMode(pin, value)
```

- **pin**: `int`, The pin number where you want to write the value.
- **value**: The value to write to the pin. It can be:
  - HIGH: Set the pin to a high voltage level (typically 5V or 3.3V depending on the board).
  - LOW: Set the pin to a low voltage value (0V).
- **returns**: `void`

1.5 Uploading

After finishing our first Arduino program, the code need to be verified and uploaded to Arduino board. By clicking the check mark button, shown in figure (7), the Arduino IDE will check the code for errors or warnings.

If the compilation has no syntax errors, a "Done compiling" message will pop up in the southeast corner of the Arduino IDE window as shown in red box in figure (8). Also, the Output window will display in white text how much memory the sketch occupies on the Arduino board. If the compilation fails, the Output window will show in red text which line or lines of the code contain syntax errors.

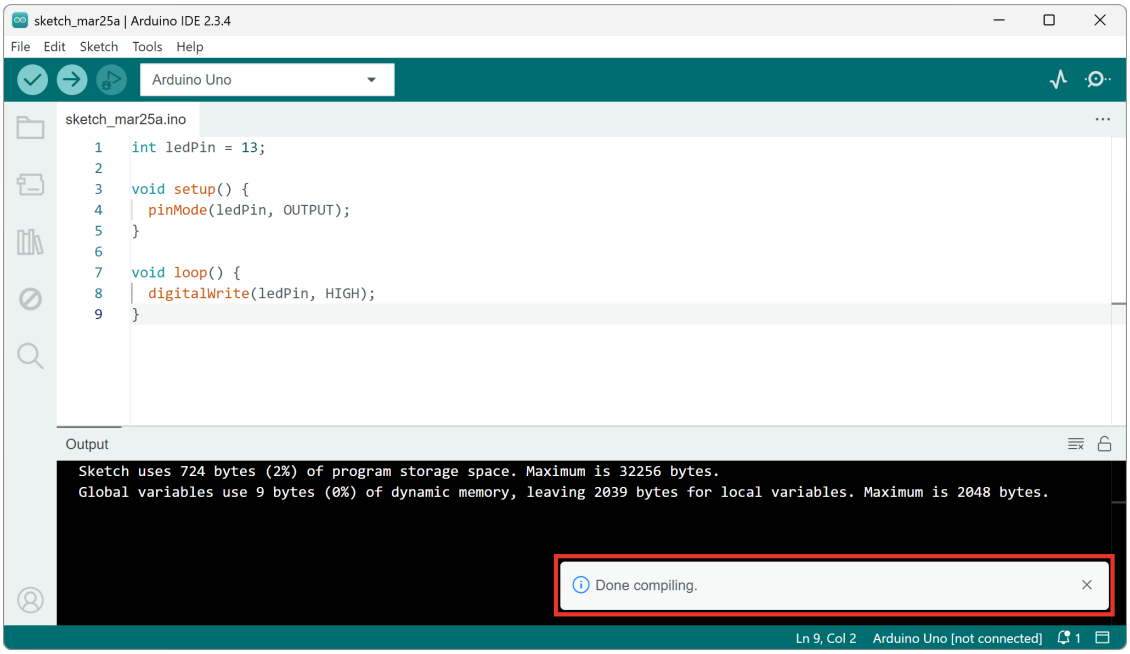


Figure 8: Compiling verified message box

1.6 Using delay — a blinking LED

Since we finished our first Arduino project, which is a good start but not fancy, we can do some simple modifications and make our LED blinking. The code is shown below:

2_BlinkLED.ino

```
1   const int ledPin = 13;
2
3   void setup() {
4       pinMode(ledPin, OUTPUT);
5   }
6
7   void loop() {
8       digitalWrite(ledPin, HIGH);
9       delay(1000);
10
11      digitalWrite(ledPin, LOW);
12      delay(1000);
13  }
```

Comparing our first sketch, the new sketch has something new — `delay(1000)`.

```
9       delay(1000);
```

This function is used to pause the program for a specific amount of time, given in **milliseconds**. During this delay, the microcontroller does nothing else — it simply waits.

Syntax:

```
delay(milliseconds)
```

- **milliseconds**: The number of milliseconds to pause.

$$1s = 1000ms$$

- **returns**: `void`

1.7 Debugging with Serial

Sometimes things don't work as expected, so debugging sketch is an important step. Here is the main debugging way which is high recommended: using `Serial`.

As introduced in section 1.3, the serial monitor displays real-time textual data from the board and this is a good way to output variables, parameters, and states to computer for debugging purpose.

3_SerialPrint_BlinkLED.ino

```
1   const int ledPin = 13;
2
3   void setup() {
4       pinMode(ledPin, OUTPUT);
5       Serial.begin(9600);
6       Serial.print("Starting LED Blink on pin ");
7       Serial.println(ledPin);
8   }
9
10  void loop() {
11      digitalWrite(ledPin, HIGH);
12      Serial.println("LED is ON");
13      delay(1000);
14
15      digitalWrite(ledPin, LOW);
16      Serial.println("LED is OFF");
17      delay(1000);
18  }
```

More `Serial` functions are added in our `2_BlinkLED.ino` script.

```
5     Serial.begin(9600);
```

This function is used to start serial communication between the Arduino board and the computer.

Syntax:

```
      Serial.begin(speed)
```

- **speed**: It refers to baud rate and defines how quickly data is sent and received

between devices. The common value is **9600** for simple communication. Another widely used baud rate for large amounts of sensor data or using real-time communication is **115200**. The serial monitor provides more baud rate options after the Arduino board is connected as shown below.

Figure 9: Baud rate options in drop-down menu of serial monitor

- **returns**: `void`

```
6     Serial.print("Starting LED Blink on pin ");
```

In `Serial` class, `print()` is used to send data to the serial port, which you can view in the Serial Monitor.

Syntax:

```
Serial.print(value)
```

- **value**: The value can be any data type (e.g. `int`, `float`, `char`, etc.)
- **returns**: `void`

```
7     Serial.println(ledPin);
```

Works the same as `Serial.print()` but adds a new line `\n` at the end. Every time you use `Serial.println()` , the next printed data will appear on a new line.

As below figure, by uploading the code to Arduino board and selecting baud rate correctly, the serial output, or debugging information, can be observed through the Serial Monitor.

Figure 10: LED states in Serial Monitor

2 Arduino Basic Programming and Interfacing

2.1 Using if — Control an LED

In the previous section, we explored how to use `Serial.print()` and `Serial.println()` to send data from the Arduino board to the Serial Monitor for debugging or output purposes. When observing output data, you may notice that there is a input field, the white input field with gray text above red box in figure (10). This input field is used to send commands from the Arduino IDE to the Arduino board. This allows for bidirectional communication: while the Arduino can send data to the Serial Monitor, we can also interact with the Arduino by typing commands directly into the Serial Monitor's input field.

Next, we are going to use `if` to control turning on and off of our LED. The entire code is shown below and we will discuss the logic first. Since the code is too long, displaying it across multiple pages would break its integrity. Therefore, overly long code will be presented function by function instead.

4_IF_SerialRead_BlinkLED.ino — `setup()`

```
1    const int ledPin = 13;
2    String command;
3
4    void setup() {
5      pinMode(ledPin, OUTPUT);
6      Serial.begin(9600);
7      Serial.println("Enter a command: ON or OFF");
8    }
```

Similar to the previous script, the baud rate is set to 9600 and a message is displayed during

the initialization phase to indicate whether the LED should be turned on or off. A `String` type variable named `command` which used to store messages from Serial Monitor input field is defined above `setup()`.

4_IF_SerialRead_BlinkLED.ino — `loop()`

```
10    void loop() {
11      if (Serial.available() > 0) {
12        command = Serial.readStringUntil('\n');
13
14        if (command == "ON") {
15          digitalWrite(ledPin, HIGH);
16          Serial.println("LED is ON");
17        }
18        if (command == "OFF") {
19          digitalWrite(ledPin, LOW);
20          Serial.println("LED is OFF");
21        }
22      }
23    }
```

Before talking about the logic within `loop`, let's introduce our first control statement — `if`, a selection statement.

Syntax:

```
if (condition) {
    // code to execute if the condition is true
}
```

- **if**: `if` is a selection statement keyword.

- **condition**: `condition` is a `bool` expression, can be `true` or `false`.
  - If the `condition` is `true`, the code in the the `{}` block will be executed.
  - If the `condition` is `false`, the code in the the `{}` block will be skipped.

```
11      if (Serial.available() > 0)
```

At the very beginning of `loop()`, Arduino first checks if serial data is available.

Syntax:

```
        Serial.available()
```

- **returns**: `int`, the numbers of bytes available to read.

```
12      command = Serial.readStringUntil('\n');
```

If there are bytes in buffer, then, the Arduino reads characters into the previously defined `command` until it encounters a newline character `\n`.

Syntax:

```
        Serial.readStringUntil(terminator)
```

- **terminator**: `char`, the character to search for.
- **returns**: `String`, contents in serial buffer before terminator.

If the received command is `ON`, the code sets the `ledPin` output to `HIGH`, turning the LED on, and then sends a message `LED is ON` to Serial Monitor.

Conversely, if the command is `OFF`, it sets `ledPin` to `LOW`, turning the LED off, and sends `LED is OFF` to Serial Monitor.

2.2 Using for — A Breathing LED

Many devices, such as fancy keyboards, smart phones, speakers, have a breathing light effect and that is very cool. In this part, we will explore how to create this effect using analog signals.

Since we know that digital output is either 0 or 1, it can only turn the LED fully on or off. However, to create a breathing LED effect, we also need levels like dim, medium, or bright. Fortunately, Arduino UNO already provides a way to do this — using analog output.

In figure (5), you will notice that some pin numbers have tilde "∼" in front them. This means those pins support analog output. Thus, in below script, we switch to pin 11 from pin 13 for analog output.

5_For_Analog_BreathingLED.ino — setup()

```
1    const int ledPin = 11;
2
3    void setup() {
4      pinMode(ledPin, OUTPUT);
5    }
```

5_For_Analog_BreathingLED.ino — loop()

```
7     void loop() {
8       for (int bright = 0; bright < 256; bright++) {
9         analogWrite(ledPin, bright);
10        delay(20);
11      }
12
13      for (int bright = 255; bright > -1; bright--) {
14        analogWrite(ledPin, bright);
15        delay(20);
16      }
17    }
```

In section `loop()`, there is a new control statement `for`.

Syntax:

```
for(initialization; condition; increment) {
    // statement(s)
}
```

- **initialization**: `int`, `float`, `long`, `char`, etc., run once before the loop starts.
- **condition**: checked before each loop iteration; if `true`, loop continuous; if `false`, loop ends.
- **increment**: runs after each iteration.

```
8     for (int bright = 0; bright < 256; bright++)
```

The loop starts where the variable `bright` begins at 0 and increases by 1 each time and stops when it reaches 256.

```
9     analogWrite(ledPin, bright);
```

`analogWrite()` is used to write an analog value to a pin. For this project, it lights the LED with varying brightness. It can also drive a motor with various speed.

Syntax:

```
      analogWrite(pin, value)
```

- **pin**: `int`, the analog-capable pin number where you want to write the value.
- **value**: `int`, a value between **0** (always off) to **255** (always on).
- **returns**: `void`

Next, a `delay(20)` is used to make the transition visible and smooth. You can try and observe what will happen if using `delay(10)` or `delay(30)`.

If the section from line 8 to 11 is called fade in, then the section from line 13 to 16 is called fade out. After reaching the maximum brightness (255), oppositely, the brightness value begins to decrease from 255 down to 0 with step of 1.

Below equation describes the duration of one complete fade-in and fade-out cycle. This duration can be calculated by adding time of fade-in (T_{in}) and fade-out (T_{out}), then, the duration is

$$T_{period} = T_{in} + T_{out}$$

and

$$T_{in} = N_{in} \times T_{delay,in},$$
$$T_{out} = N_{out} \times T_{delay,out},$$

where N_{in} and N_{out} are number of steps of fade-in and fade-out respectively; $T_{delay,in}$ and $T_{delay,out}$ are delay time at each step. Notice that the unit of `delay()` is millisecond.

In this case, the delay time is $20ms$, numbers of steps of fade-in and fade-out are the same which is 256. Thus, by applying above equation, the duration of one cycle is

$$\begin{aligned} T_{period} &= 256 \times 20ms + 256 \times 20ms \\ &= 10240ms \\ &= 10.24s \end{aligned}$$

2.3 *A Breathing LED with Digital I/O

2.3.1 *PWM Signals

While a breathing LED effect is implemented using analog output through `analogWrite()` in preceding chapter, it can also be approximated using digital output with carefully timed `digitalWrite()` pulses — a technique known as **Pulse Width Modulation** (PWM). PWM is technique used to simulate analog voltage levels using digital signals. Instead of continuously voltage, PWM rapidly switches the output between `HIGH` and `LOW` at a fixed frequency.

Figure (11) illustrates a PWM signal with a 40% duty cycle over three consecutive periods.

Each period is marked with dash vertical lines and labeled as “Period 100%”, indicating the full duration of one PWM cycle. Within each cycle, the signal alternates between an “On” and “Off” state. The blue region represents the **on-time** of the signal, which occupies 40% of each period. During this time, the output voltage remains high at a 5 V. Conversely, the gray region marks the **off-time**, covering 60% of each period, during this time, the signal voltage drops to 0 V.

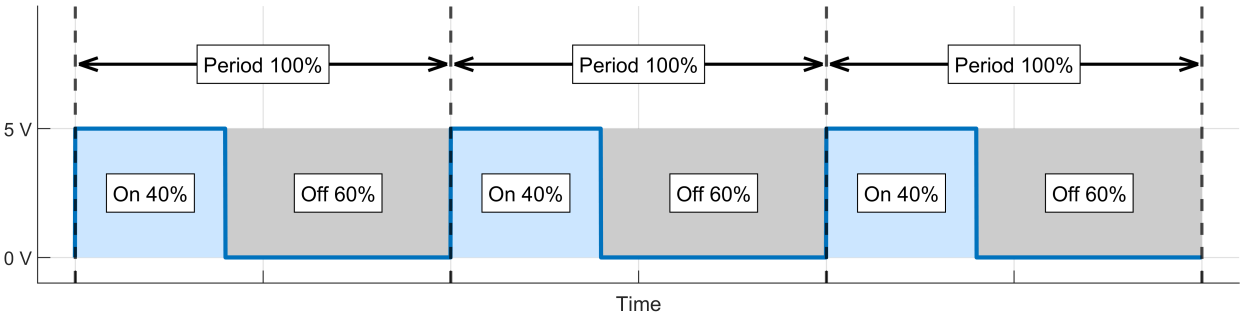


Figure 11: PWM Waveform with 40% Duty Cycle

6_1_DigitalPin_PWM.ino

```
1   const int ledPin = 13;
2   const int period = 1000;
3   const int dutyCycle = 50;
4
5   void setup() {
6     pinMode(ledPin, OUTPUT);
7   }
8
9   void loop() {
10    digitalWrite(ledPin, HIGH);
11    delayMicroseconds(dutyCycle);
12    digitalWrite(ledPin, LOW);
13    delayMicroseconds(period - dutyCycle);
14  }
```

```
1     const int period = 1000;
2     const int dutyCycle = 500;
```

As introduced, a complete PWM signal consists a period and a duty cycle. The duty cycle here is 50% which is $1000 \times 50\% = 500$.

```
10    digitalWrite(ledPin, HIGH);
11    delayMicroseconds(dutyCycle);
```

Unlike previous scripts in which we use `delay()` to create gaps in milliseconds, here, for creating PWM signals, we use `delayMicroseconds()` for finer tuning. First the LED pin is set to `HIGH`. Then, the delay 500 microseconds, hold the `HIGH` state for a specific duration defined by `dutyCycle`. These two lines of code represents how long the output remains in the on-time.

```
12    digitalWrite(ledPin, HIGH);
13    delayMicroseconds(dutyCycle);
```

Similarly, then, the LED pin is set to `LOW`, and `delayMicroseconds()` holds the off-time for the rest of this period.

2.3.2 *PWM Breathing LED and User-Defined Function

Since the brightness of an LED can be controlled via PWM signals through a digital I/O pin, to create a breathing LED, the only thing left is creating a series of PWM signals with a smoothly varying duty cycle. Figure (12) demonstrates during fade-in effects, how a PWM signal with a fixed period can vary its duty cycle.

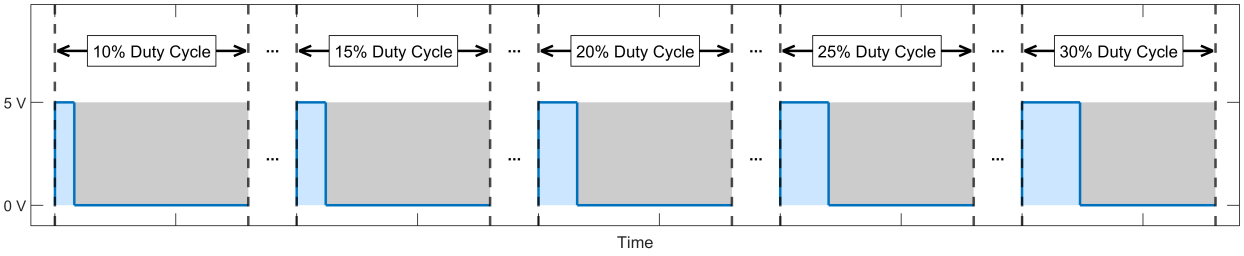


Figure 12: Increasing PWM Duty Cycle for Breathing LED Fade-in Effect

6_2_Digital_BreathingLED.ino — setup()

```
1     const int ledPin = 13;
2     const int period = 1000;
3
4     void setup() {
5     pinMode(ledPin, OUTPUT);
6 }
```

In setup() part, only one thing that different from 6_1_DigitalPin_PWM.ino is the duty cycle changing during the entire progress. Thus, the dutyCycle is no longer a constant parameter.

6_2_Digital_BreathingLED.ino — PWM_LEDBrightness(int dutyCycle)

```
18    void PWM_LEDBrightness(int dutyCycle) {
19      digitalWrite(ledPin, HIGH);
20      delayMicroseconds(dutyCycle);
21      digitalWrite(ledPin, LOW);
22      delayMicroseconds(period - dutyCycle);
23    }
```

```
1     void PWM_LEDBrightness(int dutyCycle) {}
```

Before talking about the loop() section, let's focus on a function that never shown in

previous scripts, `PWM_LEDBrightness(int dutyCycle)` which is a user-defined function. Writing how to create a PWM signals as a function helps make the code more reusable, readable, and modular. Instead of repeating the same `digitalWrite()` and `delayMicroseconds()` logic throughout the sketch, you can simply call this function just with different `dutyCycle` values to control LED brightness.

Syntax:

```
returnType functionName(type1 parameter1, ...) {
    // function body
    return returnValue;
}
```

- **returnType**: the data type the function returns (e.g., `int`, `float`, `bool`, `void` if nothing is returned)
- **functionName**: the name give to the function.
- **type1 parameter1**: optional input values the function uses, with specific types.
- **return returnValue**: the value returned to the place where the function was called (skip if `void`).

Back to our function `PWM_LEDBrightness(int dutyCycle)`, since the function does not send anything back to the code, the return value is `void`. Inside the function is our logic of creating a PWM signal, and by passing the `dutyCycle` which is the most important factor that affect the PWM signals, the code generates a PWM signal with a specific duty cycle to a predefined digital I/O pin.

6_2_Digital_BreathingLED.ino — loop()

```
8     void loop() {
9       for (int i = 0; i < period; i++) {
10        PWM_LEDBrightness(i);
11      }
12
13      for (int i = period; i > 0; i--) {
14        PWM_LEDBrightness(i);
15      }
16    }
```

By introducing PWM_LEDBrightness(int dutyCycle), the loop() section is now highly readable, containing only a few lines of code. The first loop is fade-in effect. It gradually increases the duty cycle from 0% to 100%. Then, the second loop, the fade-in effect, gradually decreases the duty cycle from 100% back to 0%.

3 Sensors

3.1 HC-SR04 Ultrasonic Sensor

The HC-SR04 ultrasonic sensor as shown in figure (13) is a commonly used, affordable electronic component designed to measure distances by utilizing ultrasonic sound waves. It operates based on the echo principle that emitting high-frequency sound pulses, typically around 40 kHz, and detecting the reflected waves bounced back from an object.

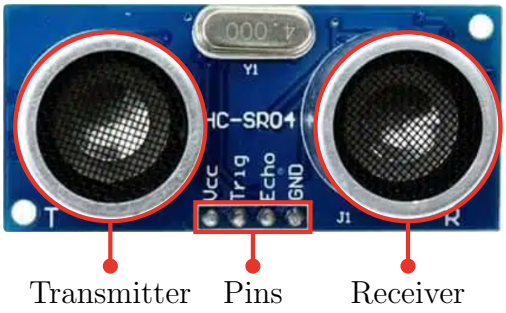


Figure 13: HC-SR04 Ultrasonic Sensor

Structurally, the HC-SR04 sensor consists of two primary transducers: one transmitter and one receiver, marked as **T** and **R** in figure (13). Figure (14) represents the principle of HC-SR04 ultrasonic sensor. When triggered, the transmitter emits a short burst of ultrasonic pulses that travel through the air at the speed of sound, and the sensor immediately sets the echo pin to `HIGH`. Once these pulses encounter an object, they are reflected and captured by the receiver. At that moment, the echo pin is set to `LOW`, marking the end of the measurement. The distance can then be calculated by measuring the duration that the echo pin remains `HIGH`, multiplying it by the speed of sound, and dividing the result by two.

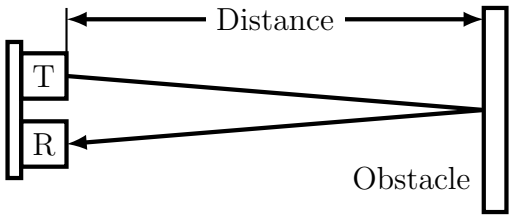


Figure 14: Ultrasonic ranging principle: transmit and receive path

7_UltrasonicSensor.ino — setup()

```
1   const int trigPin = 12;
2   const int echoPin = 13;
3
4   long duration;
5   long distance;
6
7   void setup() {
8     pinMode(trigPin, OUTPUT);
9     pinMode(echoPin, INPUT);
10    Serial.begin(9600);
11  }
```

At the very beginning, the trigger and echo pin are defined as 12 and 13 separately and the `duration` and `distance` are also created to store data from sensor. Then, with background knowledge just introduced, in `setup()` section, the trigger pin is configured as `OUTPUT` while the echo pin is configured as `INPUT`.

7_UltrasonicSensor.ino — loop()

```
13    void loop() {
14      digitalWrite(trigPin, HIGH);
15      delayMicroseconds(10);
16      digitalWrite(trigPin, LOW);
17
18      duration = pulseIn(echoPin, HIGH);
19      distance = duration * 0.034 / 2;
20
21      Serial.print("Distance: ");
22      Serial.print(distance);
23      Serial.println(" cm");
24
25      delay(100);
26    }
```

In `loop()` section, the trigger pin is set `HIGH` for 10 microseconds to send out ultrasonic pulses. It's also recommend that pull the trigger pin `LOW` for 2 microseconds to ensure a clean signal. After that, it is pulled `LOW` again.

```
20      duration = pulseIn(echoPin, HIGH);
```

The `pulseIn()` function measures the duration of the echo signal by recalling the duration is the time the echo pin remains `HIGH`.

Syntax:

```
        pulseIn(pin, value);
```

- **pin**: `int`, the number of pin you want to read the pulse.

- **value**: `HIGH` or `LOW`.
- **returns**: `long`, the length of the pulse (in microseconds) or 0 if no pulse started before the timeout.

The duration is then converted to distance in centimeters using the speed of sound (approximately 340 m/s or 0.034 cm/μs).

$$\text{distance} = \frac{\text{duration} \times 340 \text{ m/s}}{2}$$

Then the result is printed to the Serial Monitor. The loop repeats every 100 milliseconds to give the sensor enough time to settle before next trigger.

3.2 DHT22 Temperature and Humidity Sensor

The DHT22, shown in figure (15), is a digital sensor that measures both temperature and humidity. It's commonly used in weather stations, smart homes, and DIY electronics projects for it's cheap and easy to use with microcontroller like Arduino.

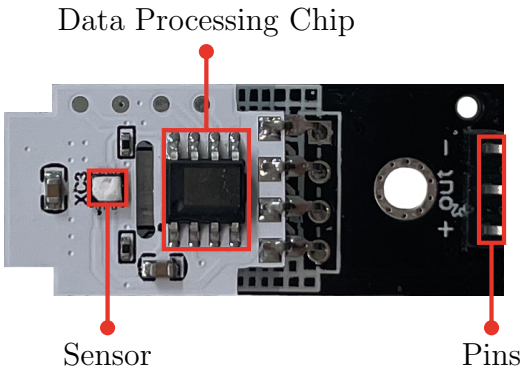


Figure 15: Teardown View of DHT22 Temperature and Humidity Sensor

The DHT22 works by using a humidity-sensitive resistor and a thermistor. As humidity

changes, the resistance of the humidity resistor varies. For temperature sensing, it uses a small built-in thermistor whose resistance changes with temperature. An onboard chip converts these analog changes into digital signals and sends them to the microcontroller through a single data pin.

8_TemperatureAndHumiditySensor.ino — setup()

```
1   #include "DHT.h"
2   const int dhtPin = 13;
3
4   DHT myDHT(dhtPin, DHT22);
5
6   float humidity;
7   float temperature;
8
9   void setup() {
10    Serial.begin(9600);
11    myDHT.begin();
12  }
```

```
1   #include "DHT.h"
```

First of all, this script includes the `DHT.h` library which provides the necessary functions and definitions for DHT22, similar to how `stdio.h` is included in most C++ programs. Without DHT.h, developers would have to implement the sensor's digital communication protocol from scratch like what we did for the HC-SR04 ultrasonic sensor. The library can be found and downloaded by clicking library manager button, see figure (7), then typing "DHT Sensor Library".

Then, the DHT22 sensor pin is defined.

```
4     DHT myDHT(dhtPin, DHT22);
```

This line creates an object named `myDHT` using the `DHT` class from the `DHT.h` library. By specifying `dhtPin` and `DHT22` as parameters, the library knows which pin to use for communication and what type of sensor is connected.

`humidity` and `temperature` need to declared to store data from DHT22 sensor.

```
11    myDHT.begin();
```

This function initializes communication between the Arduino and the DHT22 sensor.

8_TemperatureAndHumiditySensor.ino — loop()

```
14    void loop(){
15      humidity = myDHT.readHumidity();
16      temperature = myDHT.readTemperature();
17
18      Serial.print("Humidity: ");
19      Serial.println(humidity);
20      Serial.print("Temperature: ");
21      Serial.println(temperature);
22
23      delay(1000);
24    }
```

```
15    humidity = myDHT.readHumidity();
16    temperature = myDHT.readTemperature();
```

After initializing the DHT22 sensor, the humidity and temperature data can be re-

trieved by calling these two functions. Temperature in Fahrenheit can be also obtained by using `myDHT.readTemperature(true)`.

Then the result is printed to the Serial Monitor.