



ARDUINO-BASED HEART RATE AND OXYGEN SATURATION MEASUREMENT SYSTEM DESIGN

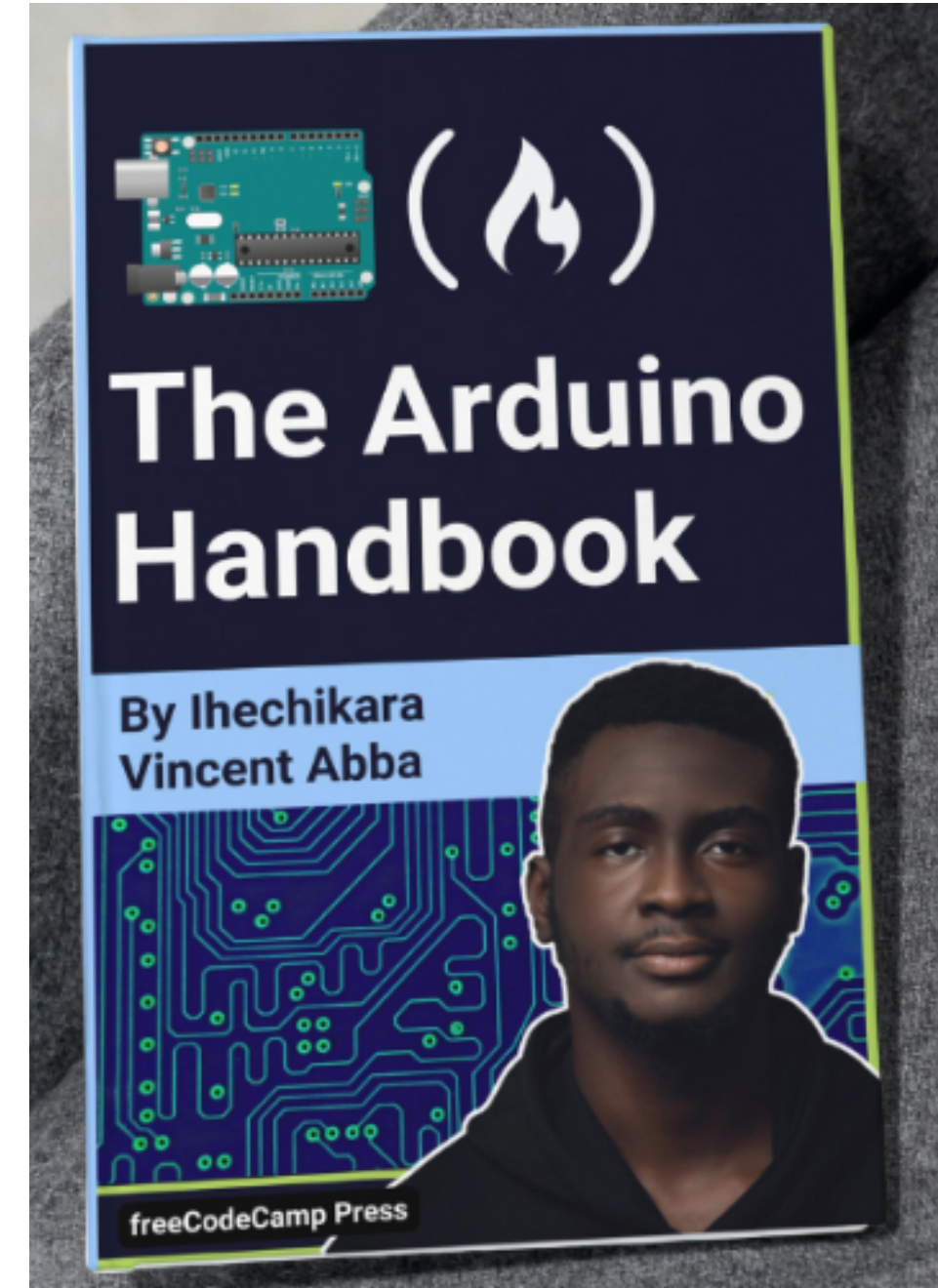
Robi Dany Riupassa

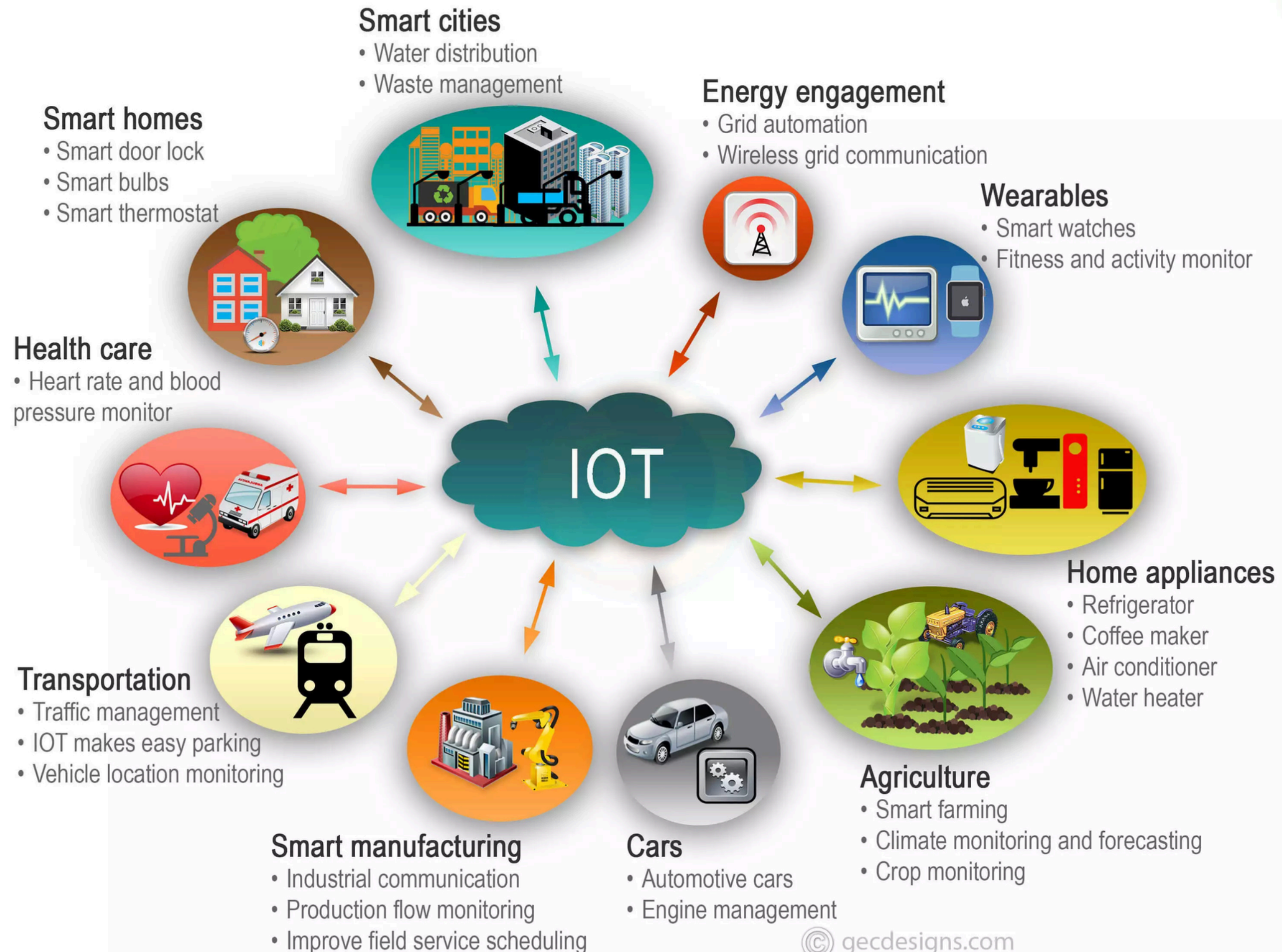
01 – Introduction

02 – Learn from PULOXY

03 – Getting Started with Arduino

04 – MAX30102 Pulse Oximeter and
Heart Rate Sensor



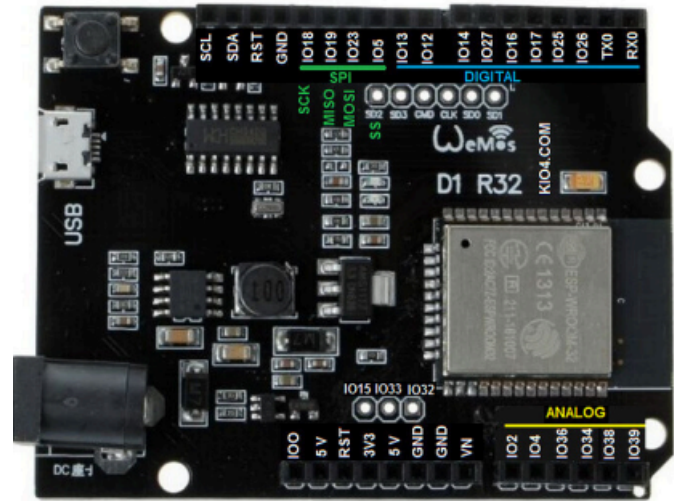


PULOXY

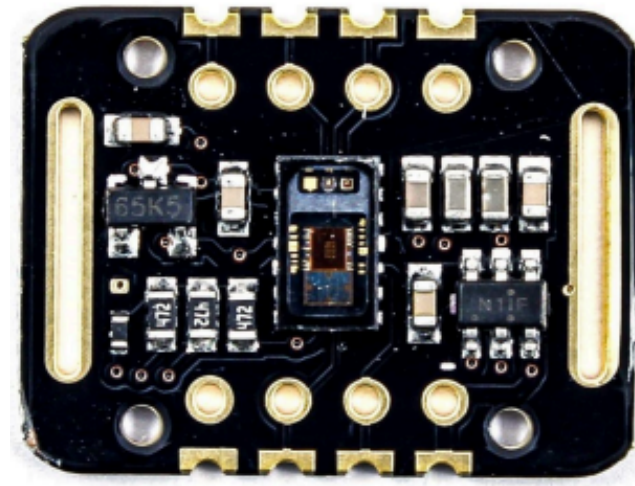
(Nicholas Austrin Theos , Sharon Gracia Edward)

PULOXY is an innovative project that creates an IoT-based device for simultaneous detection of oxygen saturation and heart rate in humans. Utilizing the Wemos D1 R32 microcontroller and MAX30102 sensor, this device provides vital health information with high accuracy. In addition to detecting health parameters, PULOXY also has the ability to store the history of each measurement and provide data access through the Blynk IoT application. The development process involves designing electronic circuits and coding algorithms to ensure accurate measurement results. Test results show that PULOXY has an accuracy rate of up to 99% compared to conventional pulse oximeters. Furthermore, the device is equipped with a feature to display histograms of each measurement, allowing users to track their health trends over time. It is hoped that the presence of PULOXY can enhance overall and more efficient health monitoring, as well as provide a more practical solution for health monitoring for the wider community. The conclusion from the testing indicates that PULOXY can be an effective replacement for conventional pulse oximeters, as it can store measurement data and provide better accessibility through the IoT platform. Thus, PULOXY can make a significant contribution to improving the quality of health monitoring and providing the necessary information for optimal health maintenance.

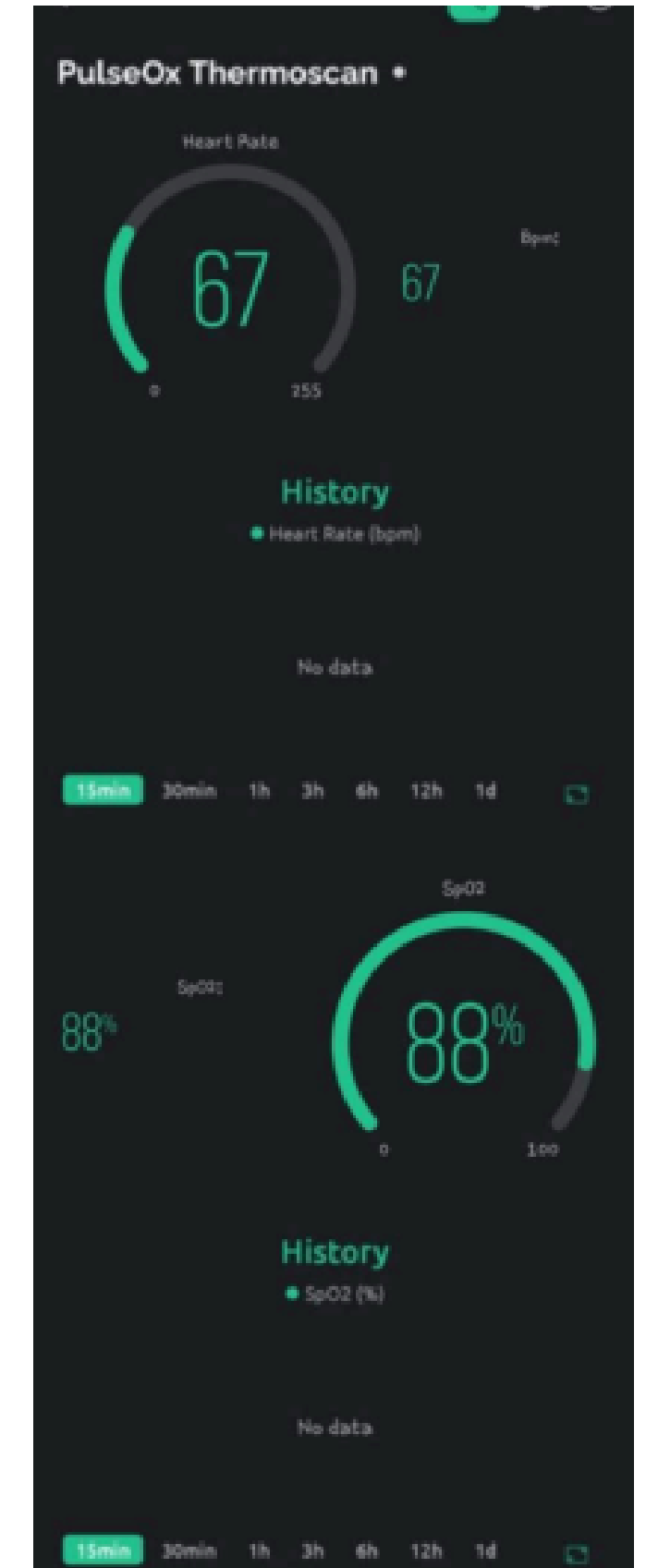
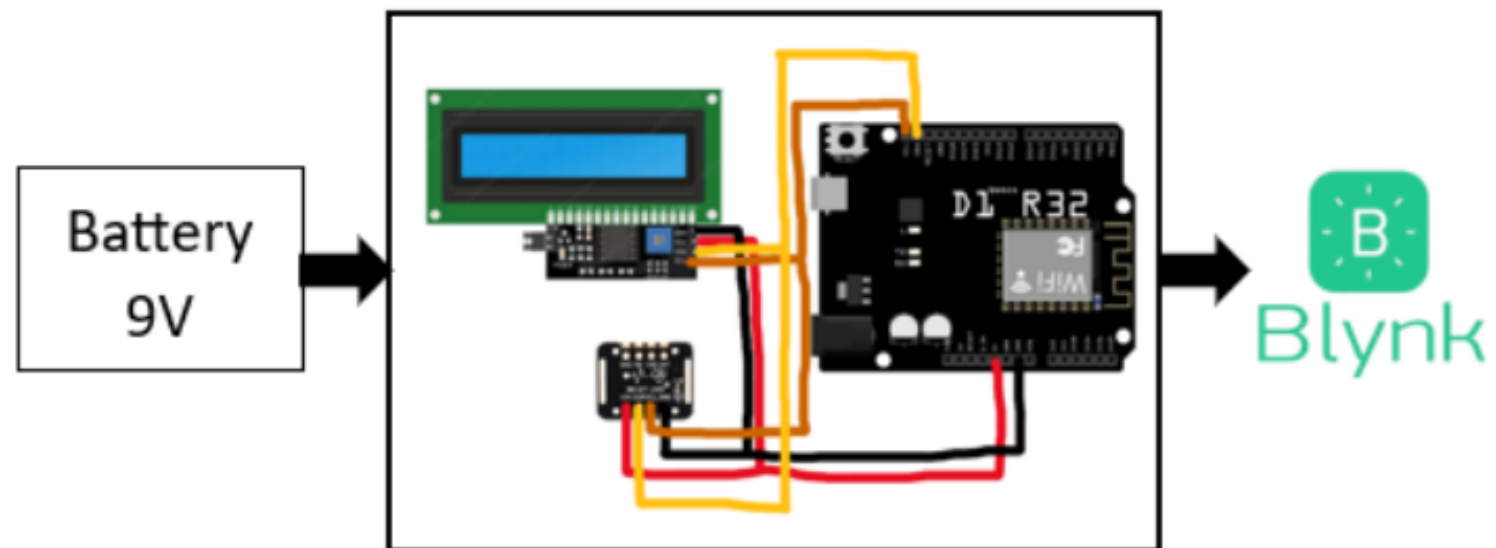
PULOXY



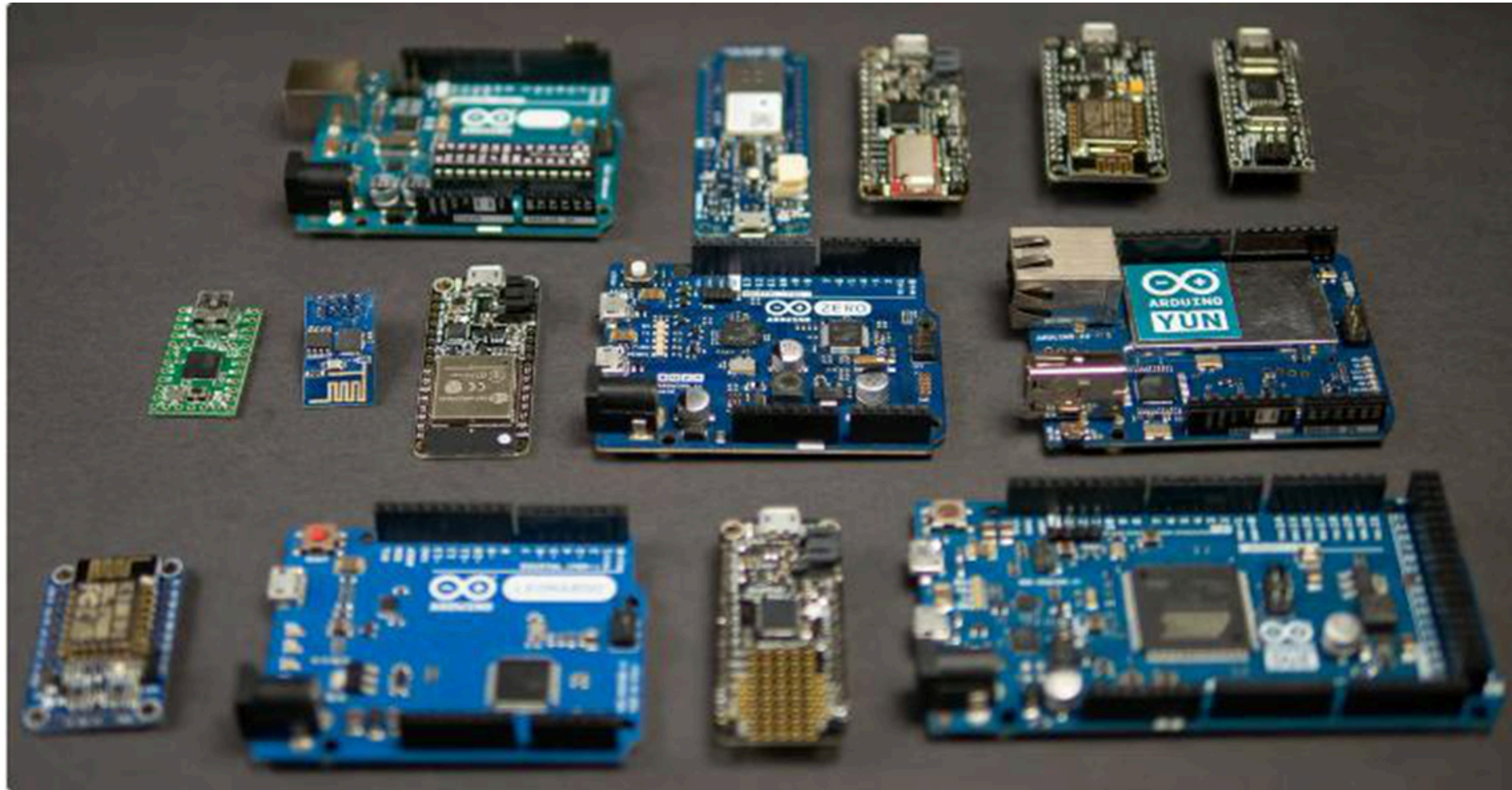
WEMOS D1 R32



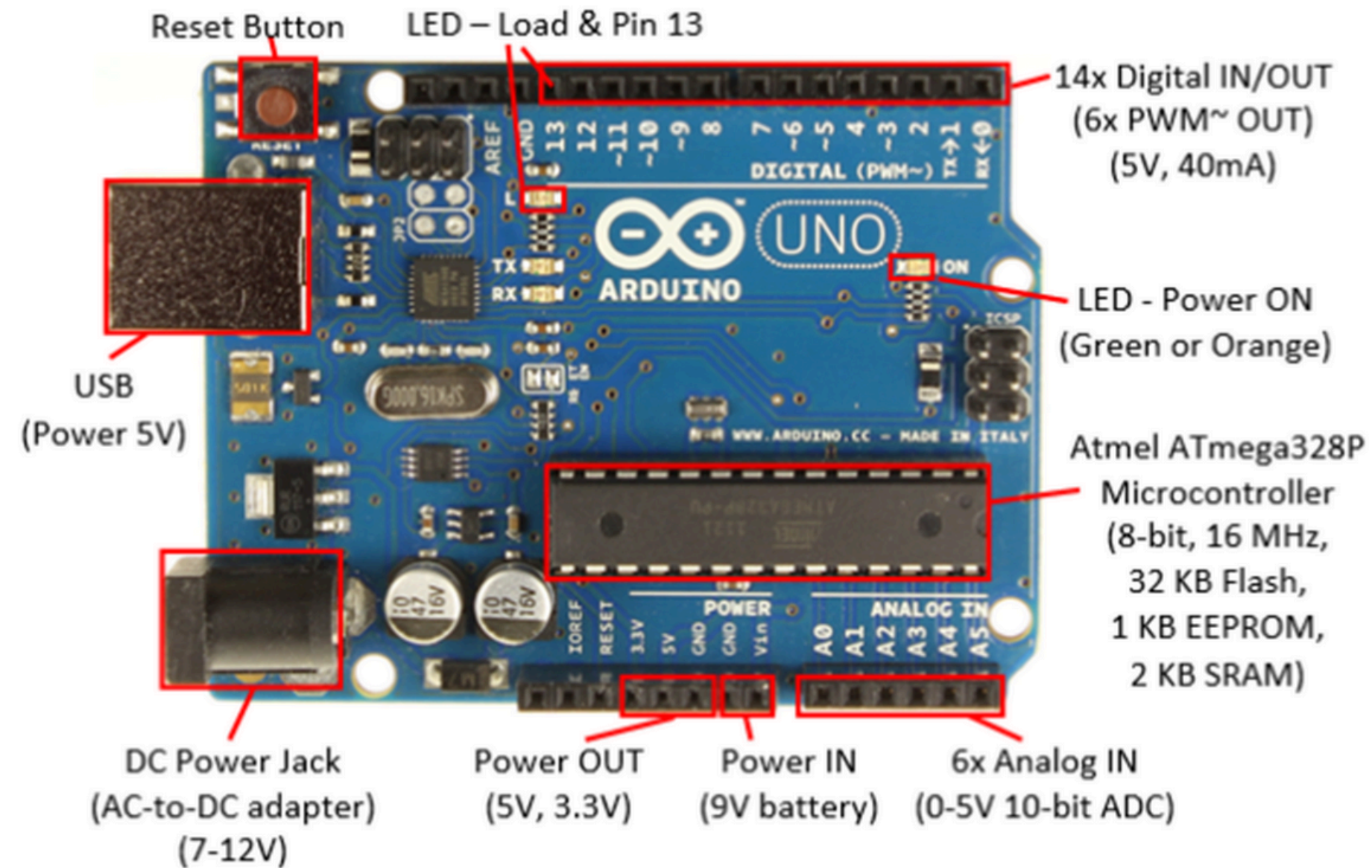
SENSOR MAX30102



Getting Started with Arduino



Arduino UNO R3



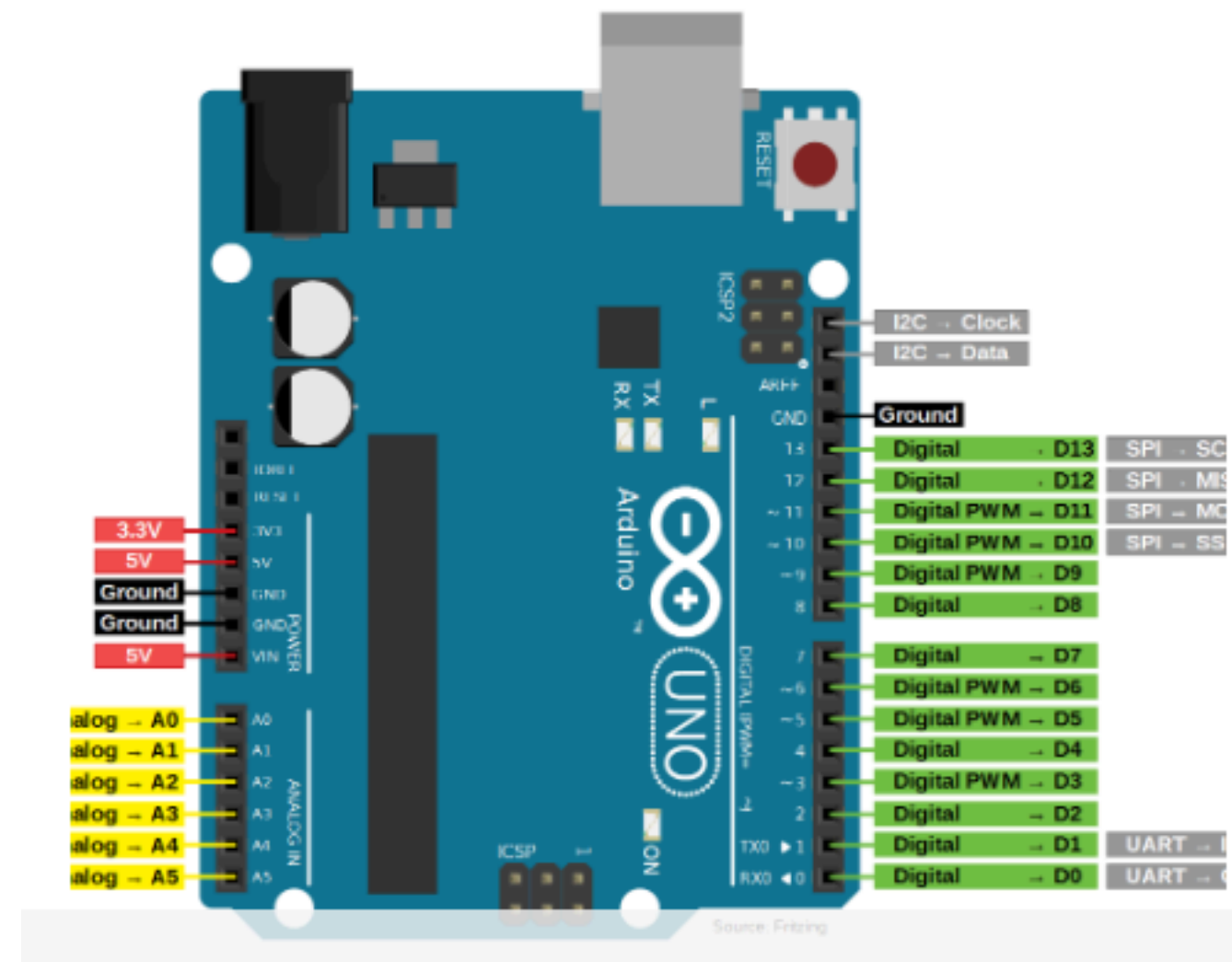
SDA and SCL

Arduino Uno

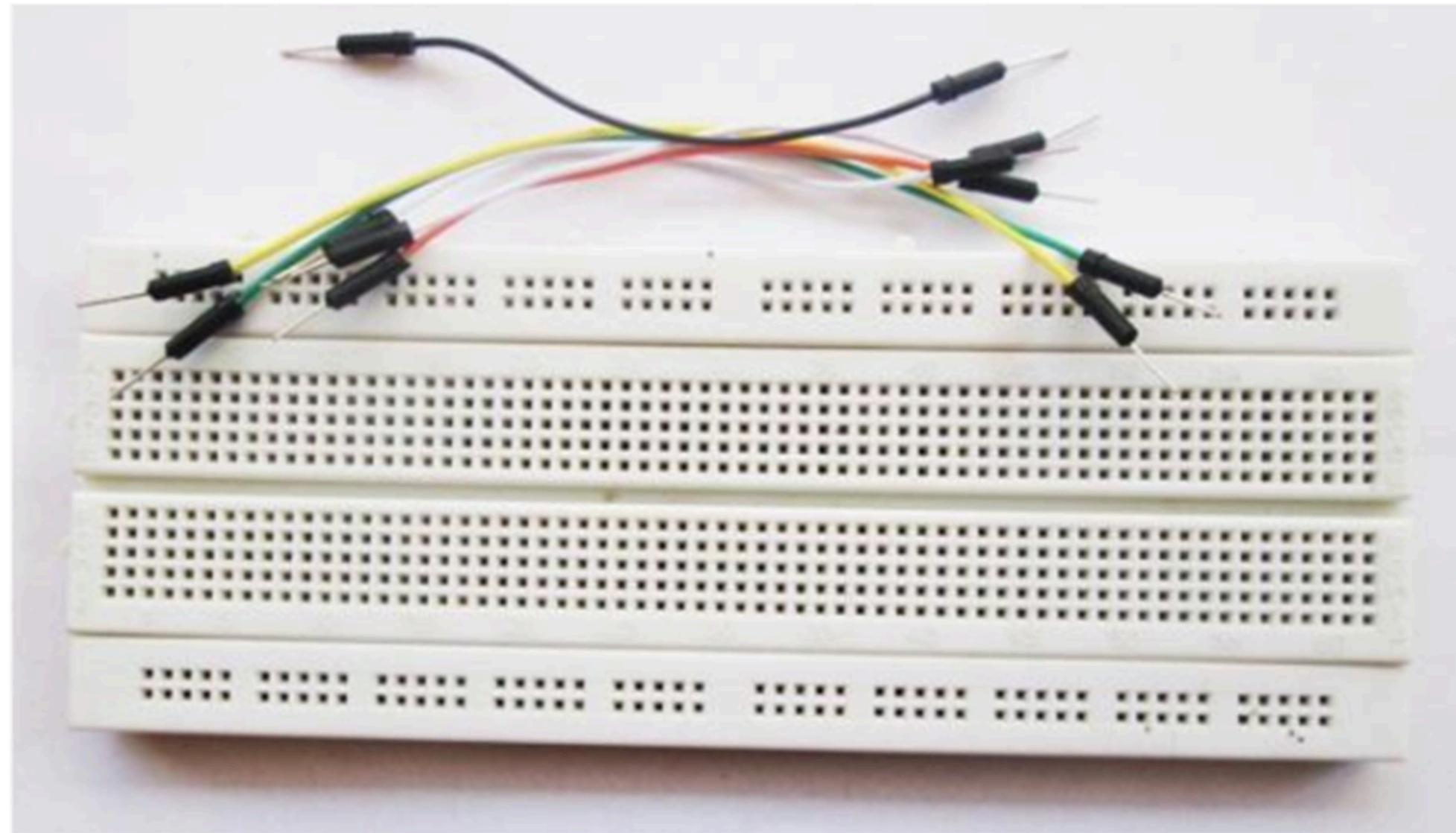
SDA: PIN18

SCL: PIN19

(no label on the PCB front, only visible from the side)



Breadboard and Jumper



Arduino IDE



Arduino IDE 2.1.0

The new major release of the Arduino IDE is faster and even more powerful! In addition to a more modern editor and a more responsive interface it features autocompletion, code navigation, and even a live debugger.

For more details, please refer to the [Arduino IDE 2.0 documentation](#).

Nightly builds with the latest bugfixes are available through the section below.

SOURCE CODE

The Arduino IDE 2.0 is open source and its source code is hosted on [GitHub](#).

DOWNLOAD OPTIONS

Windows Win 10 and newer, 64 bits

Windows MSI installer

Windows ZIP file

Linux Appliance 64 bits (X86-64)

Linux ZIP file 64 bits (X86-64)

macOS Intel, 10.14: "Mojave" or newer, 64 bits

macOS Apple Silicon, 11: "Big Sur" or newer, 64 bits

[Release Notes](#)

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

Arduino IDE




<https://docs.arduino.cc/software/ide-v2/tutorials/getting-started-ide-v2>



Arduino IDE

The Arduino programming language is built upon the C/C++ language so they both share similar syntax and structure. You may come across resources that refer to Arduino code as “embedded C” or “embedded C++”.





Upload Code

How to Upload Code to an Arduino Board

To upload code to an Arduino board, you'll need both hardware and software. The hardware is the board which is the Uno board in our case, and the software is the Arduino sketch in the IDE.

Here are the steps:

Step #1 – Connect the Arduino Board

Connect the Arduino board to your computer using the USB cable. Without this step, you can't go further.

Step #2 – Create a Sketch

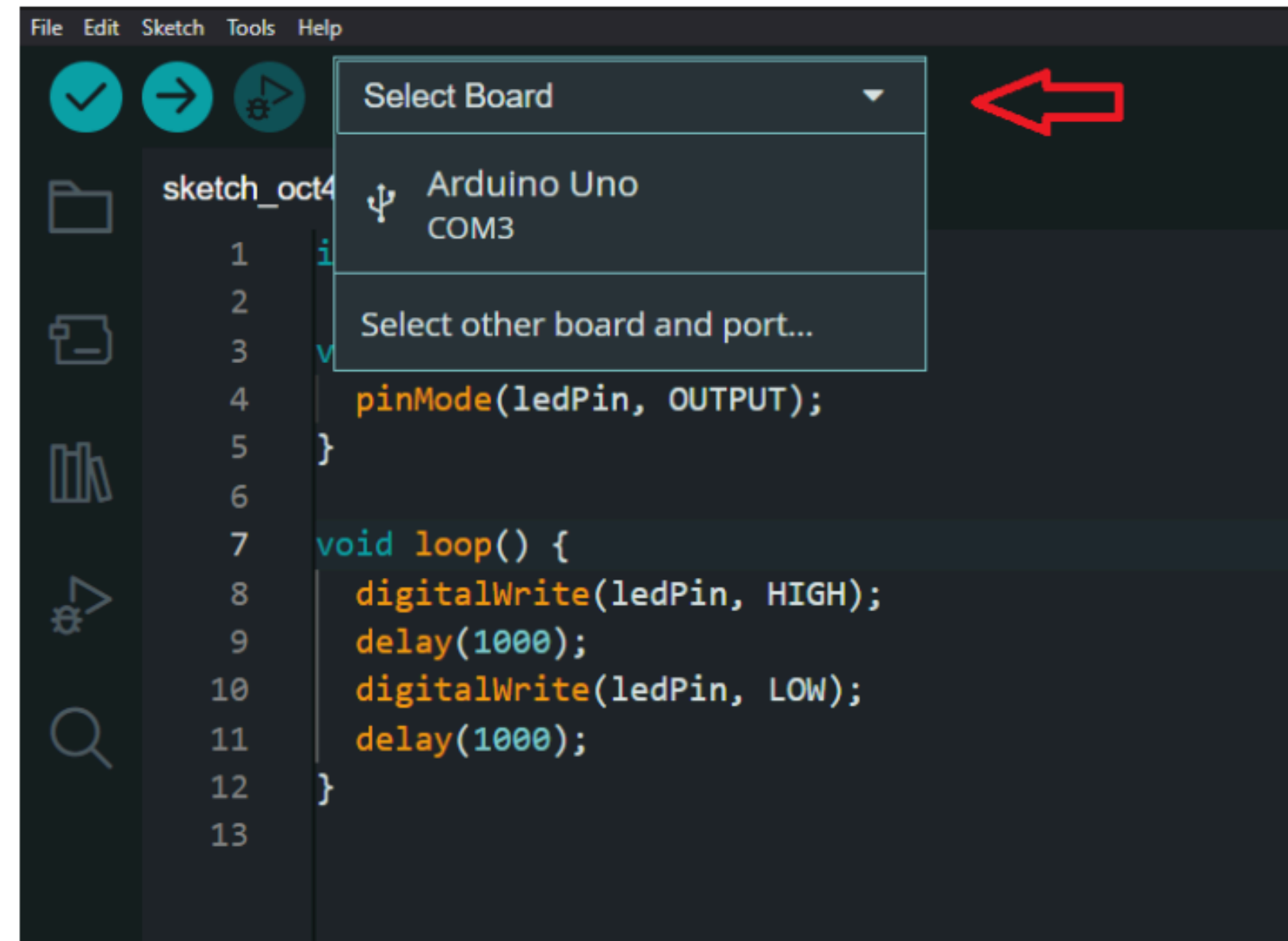
Now it's time to launch the IDE and write some code.



Upload Code

Step #3 – Select the Board and Port

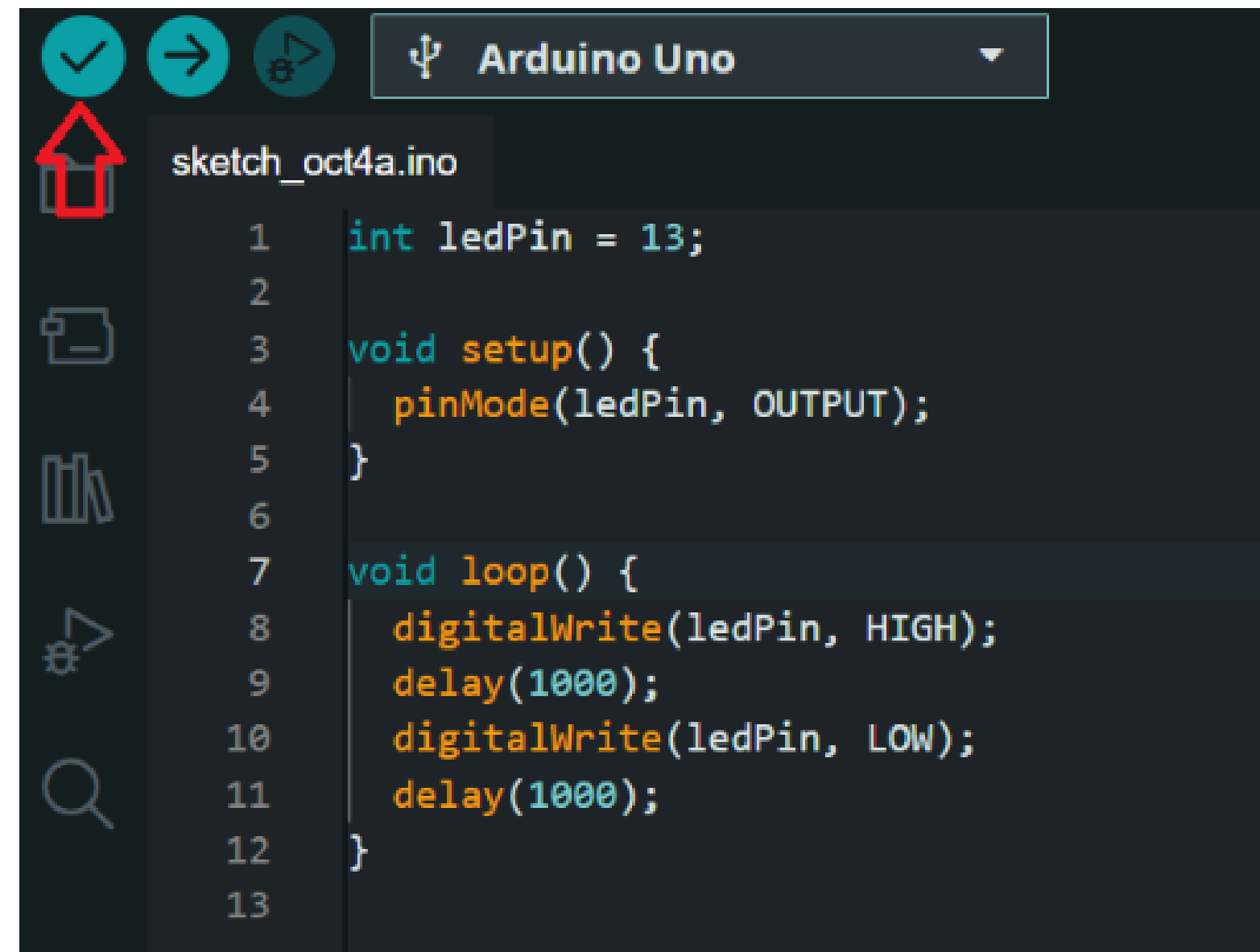
You can select the board to upload your code to from the IDE. Here's an image showing what that looks like:



Upload Code

Step #4 – Verify the Code

You can use the verify button to compile the code and check for errors. If errors exist, you'll get an error message to show you the possible cause.



The screenshot shows the Arduino IDE interface. At the top, there is a toolbar with three icons: a checkmark (Verify), a right arrow (Upload), and a bug icon (Debug). To the right of these icons is a dropdown menu showing 'Arduino Uno'. Below the toolbar, on the left, is a sidebar with icons for Home (a red house icon), Explorer, Sources, Serial Monitor, and Search. The main area displays a code sketch named 'sketch_oct4a.ino'. The code is as follows:

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



Upload Code

Step #5 – Upload the Code

You can upload the code using the upload button (the button after the verify button).





Basics of Arduino Programming

Variables and Data Types in Arduino

Variables and data types are used in most programming languages to store and manipulate data. You can think of variables as containers or storage units. Data types, like the name implies, are the type of data stored in variables.

In Arduino programming, you must specify the data type of a variable before using it. That is:

```
dataType variableName = variableValue
```



Basics of Arduino Programming

`int` Data Type in Arduino

The `int` data type is used to store integer values. The Uno board has a 16-bit integer capacity so it can store values that fall within the range of -32,768 to 32,767.

```
int redLED = 6;
```

In the code above, we created an integer variable called `redLED` with a value of 6.

The `int` data type can also store negative integers:

```
int redLED = -6;
```

Basics of Arduino Programming

`long` Data Type in Arduino

The `long` data type is similar to `int` but has a wider range of integer values. It has a 32-bit integer limit which falls within the range of -2,147,483,648 to 2,147,483,647.

```
long largeNumber = 6000;
```

`float` Data Type in Arduino

The `float` data type can be used to store numbers with decimals. Float variables can store values up to 3.4028235E+38 and values as low as -3.4028235E+38.

```
float num = 10.5;
```



Basics of Arduino Programming


`String` Data Type in Arduino

You can use the `String` data type to store and manipulate text. You'll work with strings occasionally to display information in the form of text when building projects.

Here's a code example:

```
String greeting = "Hello World!";
```

The value of strings are nested within double quotation marks as can be seen in the code above.



Basics of Arduino Programming

`bool` and `boolean` Data Types in Arduino

You can use both `bool` and `boolean` to store/denote boolean values of either `true` or `false`.

```
bool roomIsCold = false;
```

Boolean values are mostly used with logical and comparison operators, and conditional statements (you'll learn about these later in this chapter) to manipulate and control different outcomes in an Arduino program.

`byte` Data Type in Arduino

The `byte` data type has an 8-bit unsigned integer limit that ranges from 0 to 255. Unsigned means that it can't store negative values.

```
byte sensorValue = 200;
```


Basics of Arduino Programming

Operators in Arduino

Operators are symbols or characters that can be used to perform certain operations on operands. An operand is simply any value(s) an operator acts on.

There are different categories of operators in Arduino like:

Arithmetic Operators

Arithmetic operators are used to perform mathematical operations like addition, subtraction, division, multiplication, and so on. Here are some arithmetic operators you should know:

Addition(+) Operator

The addition operator, denoted by the `+` symbol, adds two operands together:

```
int a = 5;
int b = 10;

// we use addition operator to add a and b below
int c = a + b;

Serial.print(c);
// 15
```



Getting Started with Arduino

Subtraction(-) Operator

The subtraction operator subtracts the value of one operand from another operand. It is denoted by the `-` symbol:

```
int a = 5;
int b = 10;

// we use subtraction operator to subtract b from a below
int c = b - a;

Serial.print(c);
// 5
```


Multiplication (*) Operator

You can use the multiplication operator (`*`) to multiply two operands:

```
int a = 5;
int b = 10;

// we use multiplication operator to multiply a by b below
int c = a * b;

Serial.print(c);
// 50
```





Getting Started with Arduino

Conditional Statements in Arduino

You can use conditional statements to make decisions or execute code based on specific conditions. You can combine conditional statements and logic (like operators in the last section) to control how code is executed.

Let's take a look at some conditional statements and how to use them:

`if` Statement

The `if` statement is used to execute code if a condition is `true`. Here's what the syntax looks like:

```
if (condition) {  
    // code to be executed if condition is true  
}
```



Getting Started with Arduino

In the syntax above, `condition` denotes a specified logic. If the condition is `true` then the code in the curly brackets will be executed. Here's an example:

```
int x = 5;
if (x < 10) {
  Serial.print("x is less than 10");
}

// x is less than 10
```


Getting Started with Arduino

`else` Statement

The `else` statement is used to execute code if a condition is `false`.

```
int score = 20;
if (score > 50 ) {
  Serial.print("You passed the exam!");
} else {
  Serial.print("You have to rewrite the exam!");
}

// You have to rewrite the exam
```

In the code above, the condition given is `false`. So the code for the `else` statement will be executed because the `score` variable is not greater than 50.

Remember: the code for the `else` statement only runs when the condition is `false`. If the condition is `true` then the code for the `if` statement will be executed.

Getting Started with Arduino

`else if` Statement

You can use the `else if` statement to define multiple conditions to be checked. Here's the syntax:

```
if (condition1) {  
    // code to be executed if condition1 is true  
} else if (condition2){  
    // code to be executed if condition2 is true  
} else {  
    // code to be executed if condition1 and condition2 are false  
}
```

In the syntax above, there are two conditions (you can create more than two conditions). If `condition1` is `true`, then code in the curly bracket for `condition1` will be executed.

If `condition1` is `false`, then `condition2` will be evaluated. If `condition2` is `true`, its block of code will be executed.

If both `condition1` and `condition2` are `false`, the `else` statement's code will be executed.

Getting Started with Arduino

`switch-case` Statement

In the last section, we saw how to create multiple conditions using `else if` statements. Your code might become hard to read if you have many conditions. We can clean it up and make the code more readable using `switch` statements.

Here's what the syntax looks like:

```
switch (expression) {  
  case 1:  
    // Code to be executed if expression equals case 1  
    break;  
  case 2:  
    // Code to be executed if expression equals case 2  
    break;  
  case 3:  
    // Code to be executed if expression equals case 3  
    break;  
  default:  
    // Code to be executed if expression doesn't match any case  
    break;  
}
```



Getting Started with Arduino

Loops in Arduino

You can use loops to execute code repeatedly until a certain condition is met. You can also use loops to iterate over a collection of data and execute code on all elements of the collection.

There are different type of loops you can use in Arduino like the `for loop`, `while loop`, and `do-while` loop. Let's take a look at their syntax along with some practical examples:

`for` loop

You can use the `for loop` to iterate through a collection or execute code until a certain condition is met. It is commonly used when you know the number of times the loop is supposed to run.

Here's the syntax:

```
for (initialization; condition; increment/decrement) {  
    // code to be executed  
}
```



Getting Started with Arduino

`while` loop

The `while` loop works just like the `for` loop — it executes code as long as the given condition is `true`. But its often used when the number of times the loop is supposed to run is unknown.

Here's the syntax:

```
while (condition) {  
    // Code to be executed  
}
```

In the syntax above, the code will continue to run until the `condition` becomes `false`.



Getting Started with Arduino

Functions in Arduino


In the last chapter, we discussed some built-in functions in Arduino that can be used to carry out a variety of tasks related to Arduino hardware and software components. All we did was write the function name and pass in parameters where necessary and we got the desired outcome.

For instance, the `digitalWrite()` function writes values to digital pins using two parameters (the pin number and the value to be sent to the pin). Under the hood, some code logic handles that operation.

Let's assume that the logic required to send values to digital pins was up to a hundred lines of code. Without functions, you'll have to write those hundred lines every time you wanted to send values to digital pins.

Functions prevent you from having to reinvent the wheel. They also help you break your code down into smaller, more readable and manageable parts.

Just like how built-in functions can be reused to perform a particular task repeatedly, you can also create your own functions for specific functionalities, and that's exactly what you'll learn in this chapter.



Getting Started with Arduino

How to Declare a Function with the `void` Type

In the last chapter, we discussed the `void Setup()` and `void loop()` functions. They are two built-in functions that you'll use in every Arduino sketch. These functions are defined using the `void` keyword because they return nothing

Here's what the syntax looks like for functions that use the `void` type:

```
void functionName(optionalParameters) {  
    // code logic  
}
```

In the syntax above, `functionName` denotes the name of the function. We can use that name to call the function in order to execute the code defined in the function.

`optionalParameters` are used to pass external data to the function while the code logic that runs when the function is called is written between the curly brackets.

Getting Started with Arduino

Here's an example:

```
// function declaration
void printName(String userName) {
    Serial.println("Hello " + userName);
}

void setup() {
    Serial.begin(9600);
}

void loop() {
    printName("Itechikara"); // function call
    delay(1000);
}
```




Getting Started with Arduino

Commonly Used Built-in Functions in Arduino Sketch


In this section, we'll discuss some of the commonly used built-in functions you'll come across when writing or reading Arduino code. We'll make use of them in most of the upcoming chapters of this handbook.

We'll begin with the two main parts of an Arduino sketch — the `setup()` and `loop()` functions.

`setup()` and `loop()` Functions in Arduino

You can use the `setup()` function to configure analog and digital pins, initialize variables, and do other setup functionalities. The `setup()` function is executed once — when the board is powered on or reset.

The `loop()` function runs continuously. This part of the sketch is where you write all the code logic. You can use the `loop()` function to give the Arduino board instructions on different components and sensors.



Getting Started with Arduino

```
void setup() {  
  // put your setup code here, to run once:  
  
}  
  
void loop() {  
  // put your main code here, to run repeatedly:  
  
}
```

Serial Monitor in Arduino

How to Initialize the Serial Monitor With `Serial.begin()`

You can use the `Serial.begin()` function to initialize the serial monitor. It takes in the baud rate as its parameter. Here's what the syntax looks like:

```
Serial.begin(baudRate)
```

The baud rate is the speed of data transfer between the Arduino board and the computer or any other device communicating with the Arduino board through the serial monitor.

The most commonly used baud rate is 9600, but you'll also come across resources that make use of 115200, 57600, and 38400, and so on.

Whichever baud rate you specify in the `Serial.begin()` function should always match the baud rate seen in the serial monitor window.

Serial Monitor in Arduino

How to Send Data with Serial Monitor

You can use different built-in functions reserved for serial communication in Arduino. We won't discuss all the built-in serial functions in Arduino – we'll just look at some you'll use/come across regularly. You can see more functions [here](#).

`print()` and `println()` Functions

The `print()` and `println()` functions both print data to the serial monitor. The difference between the two is that `print()` prints data on the same line while `println()` prints each data on a new line.

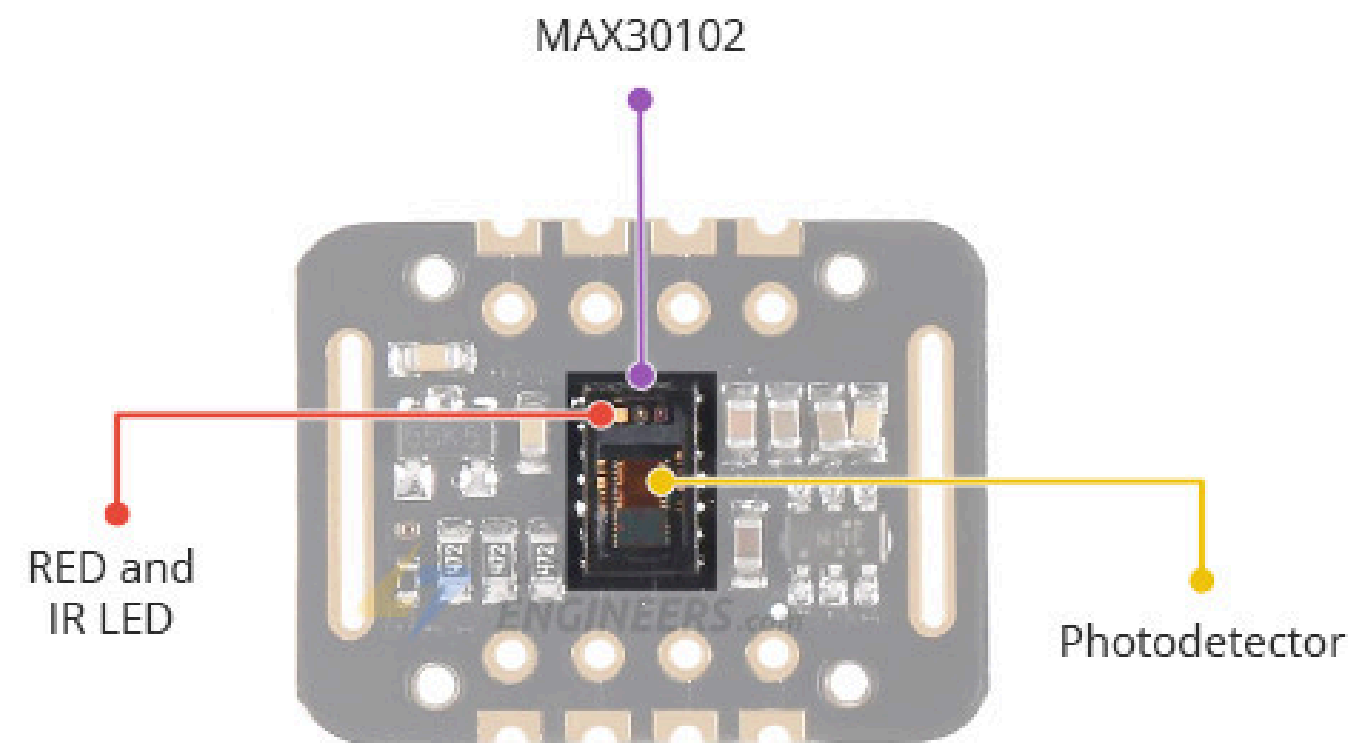
Here are some examples:

```
void setup() {  
  Serial.begin(9600);  
}  
  
void loop() {  
  Serial.print("Hello");  
  delay(1000);  
}
```

MAX30102 Sensor

MAX30102 Module Hardware Overview

The module features the MAX30102 – a modern (the successor to the [MAX30100](#)), integrated pulse oximeter and heart rate sensor IC, from Analog Devices. It combines two LEDs, a photodetector, optimized optics, and low-noise analog signal processing to detect pulse oximetry (SpO2) and heart rate (HR) signals.



MAX30102 Sensor

Power

The MAX30102 uses a 3.3V supply for the RED and IR LEDs.

Power supply	3.3V to 5.5V
Current draw	~600µA (during measurements)
	~0.7µA (during standby mode)
Red LED Wavelength	660nm
IR LED Wavelength	880nm
Temperature Range	-40°C to +85°C
Temperature Accuracy	±1°C


3.3V for the





MAX30102 Sensor

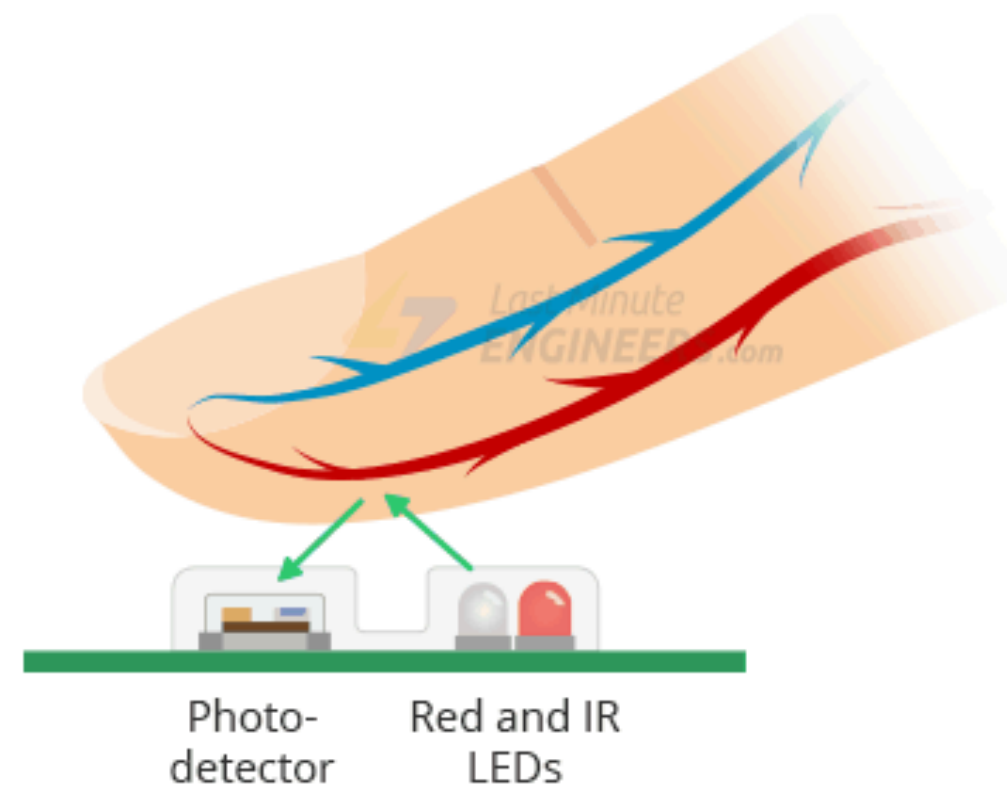
Power supply	3.3V to 5.5V
Current draw	~600μA (during measurements)
	~0.7μA (during standby mode)
Red LED Wavelength	660nm
IR LED Wavelength	880nm
Temperature Range	-40°C to +85°C
Temperature Accuracy	±1°C



MAX30102 Sensor

How MAX30102 Pulse Oximeter and Heart Rate Sensor Works?

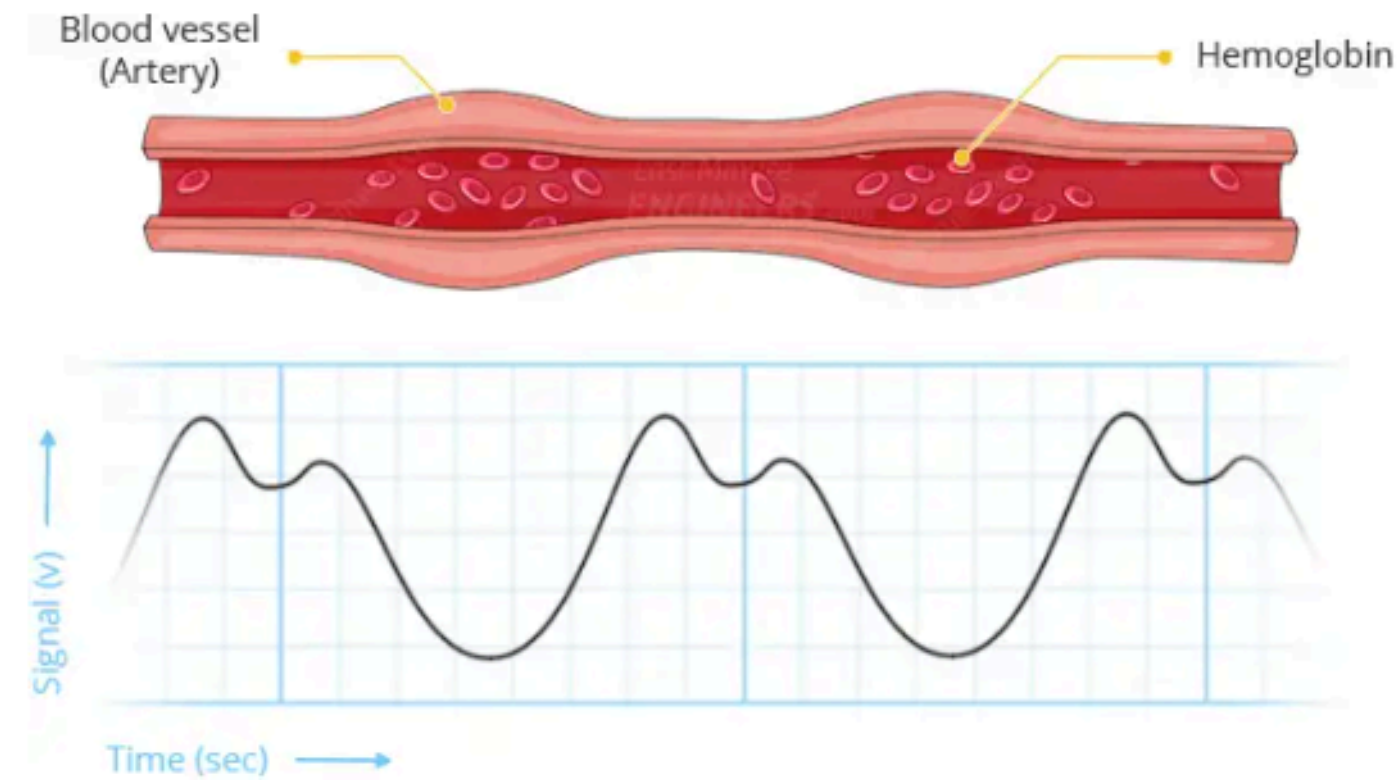
The MAX30102, or any optical pulse oximeter and heart-rate sensor for that matter, consists of a pair of high-intensity LEDs (RED and IR, both of different wavelengths) and a photodetector. The wavelengths of these LEDs are 660nm and 880nm, respectively.



MAX30102 Sensor

Heart Rate Measurement

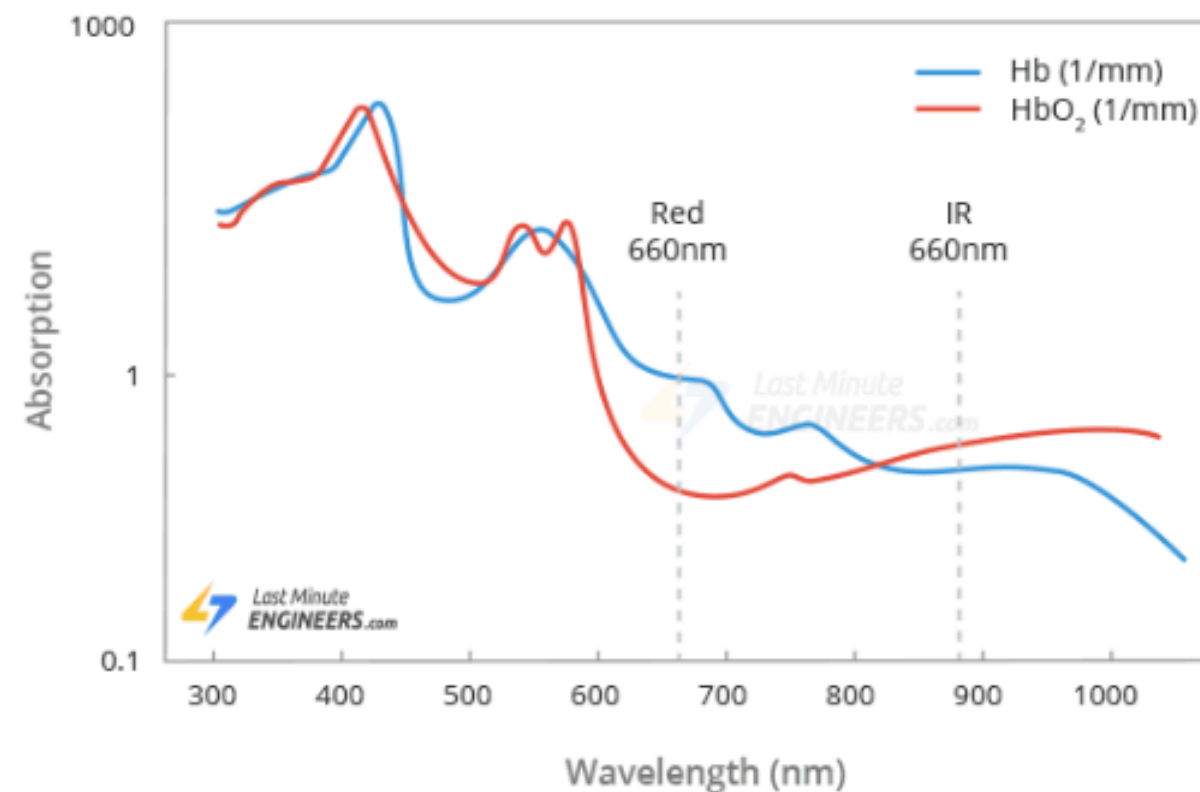
The oxygenated hemoglobin (HbO₂) in the arterial blood has the characteristic of absorbing IR light. The redder the blood (the higher the hemoglobin), the more IR light is absorbed. As the blood is pumped through the finger with each heartbeat, the amount of reflected light changes, creating a changing waveform at the output of the photodetector. As you continue to shine light and take photodetector readings, you quickly start to get a heart-beat (HR) pulse reading.



MAX30102 Sensor

Pulse Oximetry

Pulse oximetry is based on the principle that the amount of RED and IR light absorbed varies depending on the amount of oxygen in your blood. The following graph is the absorption-spectrum of oxygenated hemoglobin (HbO₂) and deoxygenated hemoglobin (Hb).

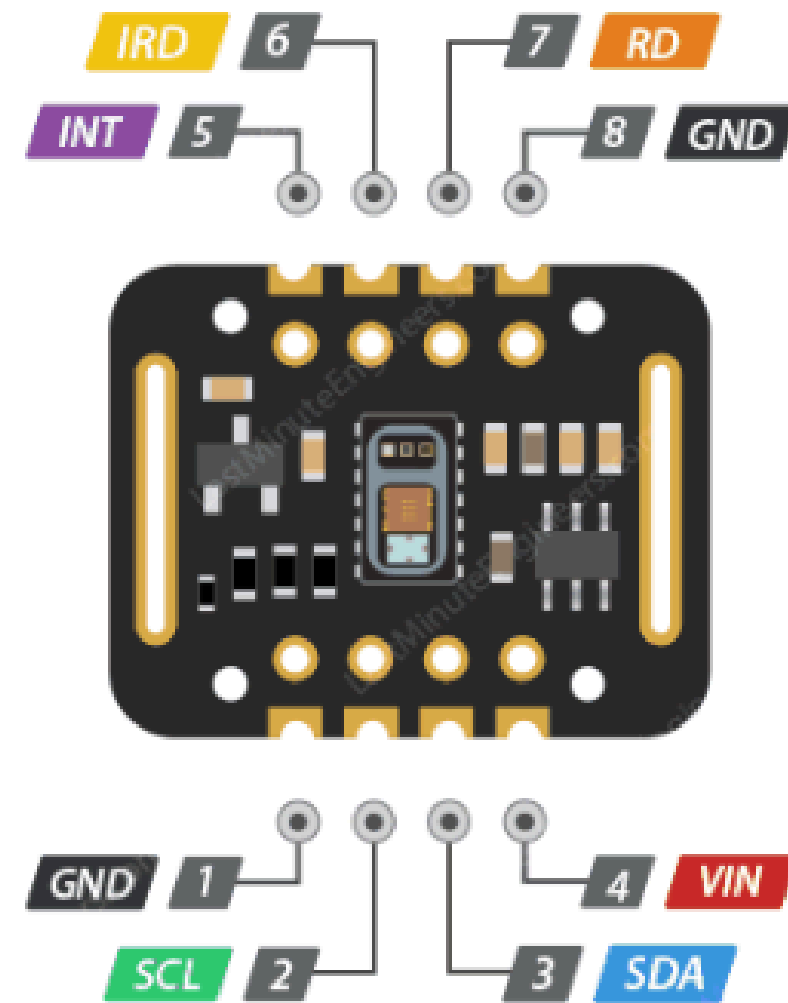


As you can see from the graph, deoxygenated blood absorbs more RED light (660nm), while oxygenated blood absorbs more IR light (880nm). By measuring the ratio of IR and RED light received by the photodetector, the oxygen level (SpO₂) in the blood is calculated.

MAX30102 Sensor

MAX30102 Module Pinout

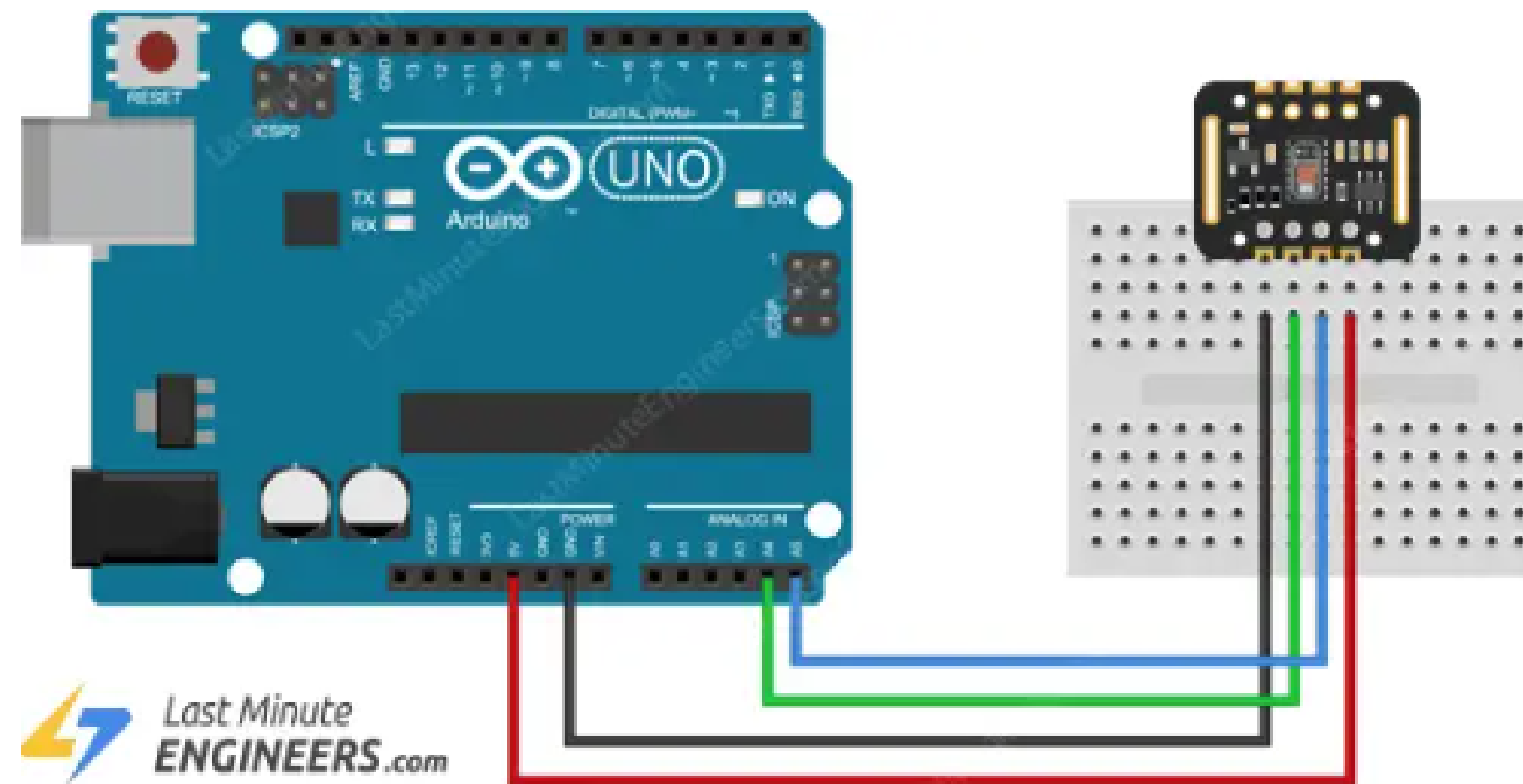
The MAX30102 module brings out the following connections.



MAX30102 Module Pinout

MAX30102 Sensor

Wiring up a MAX30102 Module to an Arduino



Adjust Parameters

```
74 byte ledBrightness = 60; //Options: 0=Off to 255=50mA
75 byte sampleAverage = 4; //Options: 1, 2, 4, 8, 16, 32
76 byte ledMode = 2; //Options: 1 = Red only, 2 = Red + IR, 3 = Red + IR + Green
77 byte sampleRate = 100; //Options: 50, 100, 200, 400, 800, 1000, 1600, 3200
78 int pulseWidth = 411; //Options: 69, 118, 215, 411
79 int adcRange = 4096; //Options: 2048, 4096, 8192, 16384
```

Adjust Parameters

LED Brightness (`ledBrightness`):

```
byte ledBrightness = 60; //Options: 0=Off to 255=50mA
```



- **What it does:** Controls how bright the LEDs on the sensor are.
- **Range:** 0 (off) to 255 (maximum brightness).
- **Why it's important:** The brightness affects how well the sensor can detect the blood flow in your finger. Too dim, and it might not detect anything; too bright, and it might cause too much reflection and noise.

Adjust Parameters

Sample Averaging (`sampleAverage`):

```
byte sampleAverage = 4; //Options: 1, 2, 4, 8, 16, 32
```



- **What it does:** Determines how many samples are averaged together to produce a single reading.
- **Range:** 1 (no averaging) to 32 (high averaging).
- **Why it's important:** Averaging helps to smooth out the readings by reducing random noise. Higher averaging means smoother data but slower response time.

Adjust Parameters

LED Mode (`ledMode`):

```
byte ledMode = 2; //Options: 1 = Red only, 2 = Red + IR, 3 = Red + IR + Green
```



- **What it does:** Sets which LEDs are used for measurements.
- **Options:**
 - 1: Only the Red LED is used.
 - 2: Both Red and Infrared (IR) LEDs are used.
 - 3: Red, IR, and Green LEDs are used.
- **Why it's important:** Different LEDs can provide different types of information. For example, Red and IR LEDs are typically used for measuring heart rate and SpO2, while the Green LED can be used for other types of measurements.

Adjust Parameters

Sample Rate (`sampleRate`):

```
byte sampleRate = 100; //Options: 50, 100, 200, 400, 800, 1000, 1600, 3200
```



- **What it does:** Sets how many times per second the sensor takes a reading.
- **Range:** 50 to 3200 samples per second.
- **Why it's important:** A higher sample rate can capture more detail and provide more accurate readings, but it also uses more power and can generate more data than necessary.

Adjust Parameters

Pulse Width (`pulseWidth`):

```
int pulseWidth = 411; //Options: 69, 118, 215, 411
```

- **What it does:** Determines the duration of each LED pulse.
- **Options:** 69, 118, 215, 411 microseconds.
- **Why it's important:** Longer pulse widths can provide more accurate readings by allowing more light to penetrate the skin, but they also consume more power.

Adjust Parameters

ADC Range (`adcRange`):

```
int adcRange = 4096; //Options: 2048, 4096, 8192, 16384
```



- **What it does:** Sets the range of the Analog-to-Digital Converter (ADC) that converts the analog signal from the sensor to a digital value.
- **Options:** 2048, 4096, 8192, 16384.
- **Why it's important:** A higher range allows the sensor to detect stronger signals without saturating, but it can also reduce the resolution of weaker signals.

The background features four decorative geometric patterns in the corners. The top-left corner has a series of parallel diagonal lines in a light blue-grey color. The top-right corner contains a cluster of overlapping semi-circles in yellow, red, and teal. The bottom-left corner features a similar cluster of overlapping semi-circles in red, teal, and dark blue. The bottom-right corner has a large, light blue-grey arc with several parallel diagonal lines extending from its base.

THANK YOU