# 简介

Arduino 是一个开源电子原型平台，包含硬件和软件，广泛应用于电子制作、物联网、机器人等领域。下面从几个方面简单介绍它：

## 硬件方面

Arduino 有多种不同的开发板，例如 Uno、Mega、Nano 等。以 Arduino Uno 为例，它基于 ATmega328P 微控制器，有 14 个数字输入输出引脚（其中 6 个可用于 PWM 输出）、6 个模拟输入引脚，通过 USB 接口与计算机连接进行编程和供电。

## 软件方面

使用 Arduino IDE 进行编程，其编程语言基于 C/C++。下面是一个简单的 Arduino 代码示例，用于控制一个连接在数字引脚 13 上的 LED 灯闪烁：

cpp

// 定义LED连接的引脚const int ledPin = 13;

void setup() {

// 将LED引脚设置为输出模式

pinMode(ledPin, OUTPUT);}

void loop() {

// 点亮LED

digitalWrite(ledPin, HIGH);

// 延迟1000毫秒（即1秒）

delay(1000);

// 熄灭LED

digitalWrite(ledPin, LOW);

// 延迟1000毫秒（即1秒）

delay(1000);}

## 代码解释

1. **setup()**函数****：在 Arduino 启动时运行一次，用于初始化设置。这里将连接 LED 的引脚设置为输出模式。
2. **loop()**函数****：在setup()函数执行完后，loop()函数会不断循环执行。在这个函数里，先将 LED 引脚设置为高电平（点亮 LED），然后延迟 1 秒，接着将 LED 引脚设置为低电平（熄灭 LED），再延迟 1 秒，如此循环。

## 开发流程

1. ****安装 Arduino IDE****：从 Arduino 官方网站下载并安装适合你操作系统的 Arduino IDE。
2. ****连接硬件****：将 Arduino 开发板通过 USB 线连接到计算机。
3. ****编写代码****：在 Arduino IDE 中编写代码。
4. ****上传代码****：选择正确的开发板型号和端口，点击上传按钮将代码上传到 Arduino 开发板。

# Auduino Ecosystem

## Getting start with Arduino

The Arduino platform has since its start in 2005, grown to become one of the most recognizable brands in the space of electronics and embedded design.

### Arduino Hardware

#### Basic Operation

Most Arduino boards are designed to have a single program running on the microcontroller. This program can be designed to perform one single action, such as blinking an LED. It can also be designed to execute hundreds of actions in a cycle.

The program that is loaded to the microcontroller will start execution as soon as it is powered. Every program has a function called "loop".

The "standard" Arduino typically has two memories: SRAM and Flash memory.

### Arduino API

Visit the **[Arduino Language Reference](https://www.arduino.cc/reference/en/)** to explore the full Arduino API.

The Arduino API, aka(/,eikei'ei /, as known as) the "Arduino Programming Language", consists of several functions, variables and structures based on the C/C++ language.

#### Main Parts

The Arduino API can be divided into three main parts: functions, variables and structure:

Functions: for controlling the Arduino board and performing computations. For example, to read or write a state to a digital pin, map a value or use serial communication.

Variables: the Arduino constants, data types and conversions. E.g. int, boolean, array.

Structure: the elements of the Arduino (C++) code, such as

* sketch (loop(), setup())
* control structure (if, else, while, for)
* arithmetic operators (multiplication, addition, subtraction)
* comparison operators, such as ==(equal to), !=(not equal to), > (greater than).

The Arduino API can be described as a simplification of the C++ programming language, with a lot of additions for controlling the Arduino hardware.

#### Program Structure

The absolute minimum requirement of an Arduino program is the use of two functions: void setup() and void loop(). The "void" indicates that nothing is returned on execution.

void setup() - this function executes only once, when the Arduino is powered on. Here we define things such as the mode of a pin (input or output), the baud rate of serial communication or the initialization of a library.

void loop() - this is where we write the code that we want to execute over and over again, such as turning on/off a lamp based on an input, or to conduct a sensor reading every X second.

The above functions are always required in an Arduino sketch, but you are of course able to add several more functions, which is useful for longer programs.

#### The "Sketch"

In the Arduino project, a program is referred to as a "sketch". A sketch is a file that you write your program inside. It has the .ino extension, and is always stored in a folder of the same name.

The folder can include other files, such as a header file, that can be included in your sketch.

#### Libraries

Arduino libraries are an extension of the standard Arduino API, and consists of thousands of libraries, both official and contributed by the community.

Libraries simplifies the use of otherwise complex code, such as reading a specific sensor, controlling a motor or connecting to the Internet. Instead of having to write all of this code yourself, you can just install a library, include it at the top of your code, and use any of the available functionalities of it. All Arduino libraries are open source and free to use by anyone.

To use a library, you need to include it at the top of your code, as the example below:

#include <Library.h>

Most libraries also have a set of examples that are useful to get started with the library.

You can browse through all official and contributed libraries in the **[Arduino Libraries page](https://www.arduino.cc/reference/en/libraries/)**.

#### Core Specific API

Every Arduino board requires a "core", or "package", that needs to be installed in order to program it. All packages contain the standard Arduino API, but also a specific API that can only be used with specific boards.

For example, the classic **[ArduinoCore-avr](https://github.com/arduino/ArduinoCore-avr)** package, automatically includes the **[EEPROM](https://docs.arduino.cc/learn/built-in-libraries/eeprom)**, and **[SoftwareSerial](https://docs.arduino.cc/learn/built-in-libraries/software-serial)** libraries, and can be used freely without any additional installation. In this package you will find the classic Arduino UNO, Nano, Mega2560 and more.

Another example is the **[ArduinoCore-mbed](https://github.com/arduino/ArduinoCore-mbed)** package, which includes over 40 libraries, designed for specific board features, such as:

* PDM - used for sampling audio from microphones found onboard the Nano 33 BLE Sense and Nano RP2040 Connect.
* Ethernet - for using the Ethernet functionalities of the Portenta Vision Shield.
* GSM - to access GSM functionalities on the Portenta Cat. M1/NB IoT GNSS Shield.

These features are documented in the documentation landing page of each product. A list of all hardware can be found at **[docs.arduino.cc](https://docs.arduino.cc/)**.

### Arduino Software Tools

The Arduino IDEs are available for download for free in the **[Software downloads page](https://www.arduino.cc/en/software)**.

Another integral part of the Arduino ecosystem are its software tools.

The Arduino IDE, as it is commonly referred to, is an **integrated development environment.** But what does that mean exactly?

In order to program your board, you need to write a program, compile that program into machine code, and finally: send over the new program to your board.

The Arduino IDE facilitates all this, from the first line of code written, to have it executed on the Arduino board's microcontroller. It is a program, or application, that you can download (or use an online version), to manage all of your code development. Back in the day, this was a complicated process, that required a good set of knowledge in electronics & computer science. Now, anyone can learn how to do it, with the help of the Arduino IDE.

Today, there are three Arduino IDEs available:

* Arduino IDE 1.8.x (classic)
* Arduino IDE 2 (new)
* Arduino Cloud Editor (online)

#### A Typical Workflow

To upload code to an Arduino board using the IDE, one typically does the following:

1. Install your board - this means installing the right "package" for your board. Without the package, you can simply not use your board. Installing is done directly in the IDE, and is a quick and easy operation.

2. Create a new sketch - a sketch is your main program file. Here we write a set of instructions we want to execute on the microcontroller.

3. Compile your sketch - the code we write is not exactly how it looks like when uploaded to our Arduino: compiling code means that we check it for errors, and convert it into a binary file (1s and 0s). If something fails, you will get this in the error console.

4. Upload your sketch - once the compilation is successful, the code can be uploaded to your board. In this step, we connect the board to the computer physically, and select the right serial port.

5. Serial Monitor (optional) - for most Arduino projects, it is important to know what's going on on your board. The Serial Monitor tool available in all IDEs allow for data to be sent from your board to your computer.

For what is now considered the "legacy" editor, the Arduino IDE 1.8.X, or "Java IDE", is the editor that was first released back when Arduino started.

In 2021, the Arduino IDE 2 was released. The new IDE has the same functionality, but also supports features such as auto-completion and debugging.

The **[Arduino Cloud Editor](https://create.arduino.cc/editor)** is an online IDE, part of the Arduino Cloud suite. Similar in function, this editor is completely web based, with online storage among other features. To use the Cloud Editor, you will need to register an Arduino account.

The **[Arduino Cloud](https://create.arduino.cc/iot/)** allows you to configure, program and control/monitor your devices - all in one web based application. With the use of things, or your "digital twin", you can control and monitor variables directly from dashboards. The service also supports webhooks and integrations with other services, such as Amazon Alexa.

The Cloud is made for anyone to use, and it does not require much previous experience to get started.

Get started by reading the **[Getting Started with the Arduino Cloud](https://docs.arduino.cc/cloud/iot-cloud/tutorials/iot-cloud-getting-started)** guide, or visit the **[full documentation](https://docs.arduino.cc/cloud/iot-cloud)**.

#### Library Manager

Every version of the IDE has a library manager for installing Arduino software libraries. Thousands of libraries, both official and contributed libraries, are available for direct download. Code examples for each library is made available on download.

To explore all available Arduino libraries, visit the **[Arduino Libraries page](https://www.arduino.cc/reference/en/libraries/)**.

#### Arduino CLI

The Arduino CLI is a command line tool that can be used to compile and upload code to your board. It has no visual UI, but is very useful for automation. It is designed for more advanced users.

A proper use of the CLI can speed up your development time by far, as any operation is executed much faster than in the regular IDE.

To learn more, visit the **[Arduino CLI documentation](https://arduino.github.io/arduino-cli/)**.

### Quick Reference

In this section, you will find a list of some of the most common elements in the standard Arduino API. This will help you get familiar with some key building blocks.

To explore the whole Arduino API, please refer to the **[Arduino Language Reference](https://www.arduino.cc/reference/en/)**, an in-depth wiki maintained by Arduino and its community. You will find hundreds of entries, accompanied by code examples and elaborate descriptions.

#### General

The setup() function is where you make program configurations.

The loop() function is where your main program is stored. It will run as long as your board is powered.

The delay() function pauses the program for a set number of milliseconds.

void loop() {

digitalWrite(LED, HIGH); //turn on an LED

delay(1000); //as program is paused, with the LED on

digitalWrite(LED, LOW); //program is unpaused, and the LED is turned off

delay(1000); //program is paused, with the LED off

}

The delay() function is an incredibly useful function, and you will find it in almost all examples. But, for efficiency of the code, it is not the best option, as it prevents the Arduino from doing anything for the duration of the delay.

The millis()function is a bit more advanced, but an incredibly resourceful function. It allows you to have multiple events happening simultaneously, without pausing the program. This is done by measuring time (in milliseconds) passed since the program started.

Then, with the use of intervals and continuously storing the time for last event, a simple algorithm can be made to have events happening at specific times without pausing the program.

See the example below:

unsigned long previousMillis\_1 = 0; //store time for first event

unsigned long previousMillis\_2 = 0; //store time for second event

const long interval\_1 = 1000; //interval for first event

const long interval\_2 = 2000; //interval for second event

void setup(){

}

void loop() {

//check time since program started, and store in "currentMillis"

unsigned long currentMillis = millis();

//conditional that checks whether 1 second has passed since last event

if (currentMillis - previousMillis\_1 >= interval\_1) {

previousMillis\_1 = millis();

//execute a piece of code, every \*1 second\*，如果条件不满足，if内语句不会执行。

}

//conditional that checks whether 2 seconds have passed since last event

if (currentMillis - previousMillis\_2 >= interval\_2) {

previousMillis\_2 = millis();

//execute a piece of code, every \*2 seconds\*，同上

}

}

Functions

Learn more about **[Arduino functions](https://docs.arduino.cc/learn/programming/functions)**.

You can create custom functions that either just executes code and returns to the program, or that returns a result.

Variable Definition

Variables can either be created locally or globally. Variables that are defined in the loop() are considered local, and variables defined at the top of your sketch are considered global.

Data Types

See all data types in the **[Language Reference](https://www.arduino.cc/reference/en" \l "data-types)**.

There are several data types available for use, and below are some of the most common:bool byte char double float int long short String

#### Serial Communication

Read more about the **[Serial class](https://www.arduino.cc/reference/en/language/functions/communication/serial/)**.

Serial.begin()

Initializes serial communication between board & computer. This is defined in the void setup() function, where you also specify baud rate (speed of communication).

void setup() {

Serial.begin(9600);

}

Serial.print()

Prints data to the serial port, which can be viewed in the Arduino IDE Serial Monitor tool.

void loop() {

Serial.print();

}

Serial.read()

Reads the incoming serial data.

void loop() {

int incomingByte = Serial.read();

}

#### GPIO / Pin Management

Configuring, controlling and reading the state of a digital/analog pin on an Arduino.

pinMode()

Configures a digital pin to behave as an input or output. Is configured inside the void setup() function.

pinMode(pin, INPUT); //configures pin as an input

pinMode(pin, OUTPUT); //configures pin as an output

pinMode(pin, INPUT\_PULLUP); //enables the internal pull-up resistor

digitalRead()

Reads the state of a digital pin. Used to for example detect a button click.

int state = digitalRead(pin); //store the state in the "state" variable

digitalWrite()

Writes a high or low state to a digital pin. Used to switch on or off a component.

digitalWrite(pin, HIGH); // writes a high (1) state to a pin (aka turn it on)

digitalWrite(pin, LOW); // writes a low (0) state to a pin (aka turn it off)

analogRead()

Reads the voltage of an analog pin, and returns a value between 0-1023 (10-bit resolution). Used to read analog components.

analogWrite()

Writes a value between 0-255 (8-bit resolution). Used for dimming lights or setting the speed of a motor. Also referred to as PWM, or Pulse Width Modulation.PWM is only available on specific pins (marked with a "~" symbol).

analogWrite(pin, value); //write a range between 0-255 to a specific pin

#### Structure

The structure of the Arduino API is based on C++, and can be considered the building blocks of a program.

They are Conditionals, Loops / Iterations, Arithmetic Operators, Comparison Operators, Boolean Operators, Compound Operators...

## What is Arduino?

Arduino is an open-source electronics platform based on easy-to-use hardware and software. Arduino 是一个开源电子原型平台，包含硬件和软件，广泛应用于电子制作、物联网、机器人等领域。

## Getting started with Arduino tools

#### Arduino Tools

1. If you have a reliable Internet connection, you should use the [online IDE](https://create.arduino.cc/editor) (Arduino Cloud Editor). It will allow you to save your sketches in the Cloud, having them available from any device and backed up. Also, you will always have the most up-to-date version of the IDE without the need to install updates or community generated libraries.

2. If you would rather work offline, you should use the latest version of the [desktop IDE](https://www.arduino.cc/en/Main/Software" \l "download) (Arduino Software IDE).

3. If you wish to create our very own IoT project, you should use the [Arduino Cloud](https://create.arduino.cc/iot/things). It will allow you to track data in real time, trigger remote devices and build wireless systems.

## Using the Arduino Software (IDE)

There are currently two versions of the Arduino IDE, one is the IDE 1.x.x and the other is IDE 2.x.

## Using the Arduino Cloud Editor

Using the The Arduino Cloud Editor allows you to write code and upload sketches to any official Arduino board directly from your web browser (Chrome, Firefox, Safari and Edge). However, we recommend you use Google Chrome.Arduino Cloud Editor

All you need to get started is an Arduino account. The following steps can guide you to start using the Arduino Cloud Editor:

**1.** **[Install the Arduino Create Agent](https://create.arduino.cc/getting-started/plugin/welcome)** plugin.

**2.** **[Create a new Arduino Account at this link](https://login.arduino.cc/login)**. Complete the registration form, then hit the "create account" button. Then you will receive an email with a link to activate your account. Select the link and a new page will open with your confirmed account information.

**3.** **[Log in the Arduino Cloud Editor](http://create.arduino.cc/editor)**.

## Get to know Arduino Libraries

The Arduino environment can be extended through the use of libraries. Just like most programming platforms, libraries provide extra functionality for use in sketches, e.g. working with hardware or manipulating data. To use a library in a sketch, select it from Sketch > Import Library.

A number of libraries come installed with the IDE, but you can also download or create your own. Here are some instructions for setting up a library on the offline IDE:

1. Open the IDE and click "Sketch" on the menu tab and then Include Library > Manage Libraries.

2. Search for the library that you need, click on it, then select the version of the library you want to install.

3. Finally, click on install and wait for the IDE to install the new library.

Now the new library will be available in the Sketch > Include Library menu. If you want to add your own library to Library Manager, follow **[these instructions](https://github.com/arduino/library-registry" \l "adding-a-library-to-library-manager)**.

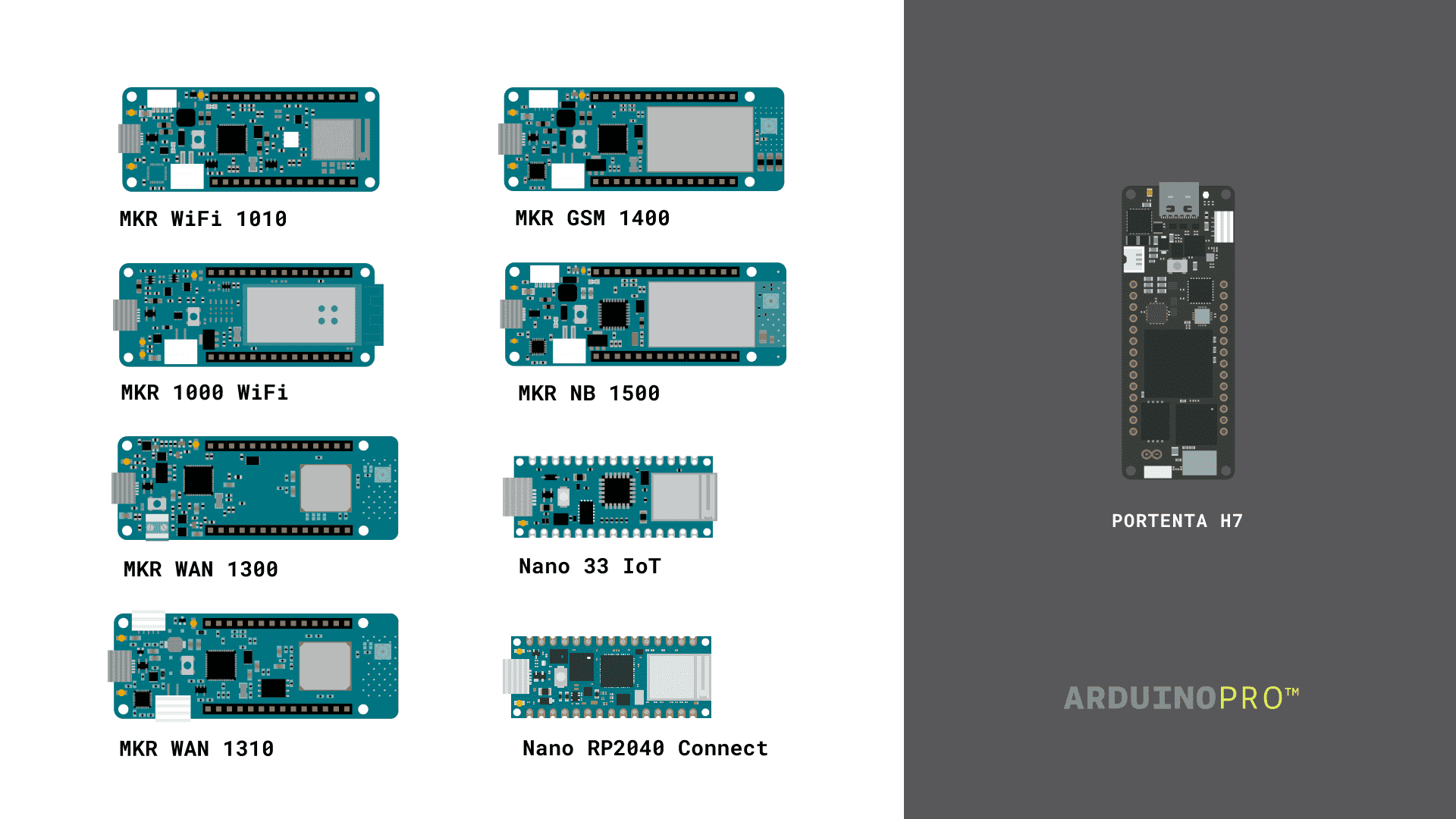
## An intro to the Arduino Cloud

With the Arduino Cloud desktop or mobile platform, you can quickly connect, manage and monitor your devices from anywhere in the world.

The following steps will guide you to start using the Arduino Cloud:

1. Install the **[Arduino Create Agent](https://create.arduino.cc/getting-started/plugin/welcome)** plugin.

2. Check if you have a cloud compatible board. The picture below shows all official Arduino boards that are compatible.



Note: The MKR GSM 1400 and MKR NB 1500 require a SIM card to connect to the Cloud, as they communicate over the mobile networks. The MKR WAN 1300 and 1310 board requires a Arduino PRO Gateway LoRa to connect to the Cloud.

1. Create an Arduino account by [signing up to Arduino](https://login.arduino.cc/login).
2. Access the Arduino Cloud from any page on [arduino.cc](https://www.arduino.cc/) by clicking on the bento menu (9-dots) on the top right corner, or you can go directly to the [Arduino Cloud](https://create.arduino.cc/iot/).

Creating a Thing

Building the Sketch

Creating the dashboard

  You can find more information about the **[Arduino Cloud here](https://docs.arduino.cc/cloud/iot-cloud/tutorials/iot-cloud-getting-started)**.

## Installing additional cores

Starting from the Arduino Software (IDE) version 1.6.2, all Arduino AVR boards are installed by default. Some Arduino boards require an additional core to be installed, therefore we have implemented the **Boards Manager** as the preferred tool to add cores to your Arduino Software (IDE).

## The Arduino Comic Project

Learn Arduino comic（动漫） book style! This community project has been translated into several languages, including Spanish, Chinese & Arabic.

#### The workflow to translate the comic is:

* Download the **[original comic](https://playground.arduino.cc/uploads/Main/arduino_comic_v0004/index.pdf)** just to see how it looks like
* Download the **[original text document](https://content.arduino.cc/assets/Arduino comic - original text document.zip)**, that contains all the strings in English, it comes both as TXT and DOC file
* Translate all the strings to your own language
* **[Download the PDF file with the blank boxes](https://content.arduino.cc/assets/Arduino-comic-blank-boxes.pdf)**
* Edit the PDF file with your graphic tool of choice: InDesign, Illustrator, Inkscape ... and insert the different strings
* Fill in the **[Support Contact Form](https://www.arduino.cc/en/contact-us/)** with information that you have created your comic. You will receive an email with your case number where you can send the translated strings as well as a PDF with the final version of the comic (if you want to share your work files it might also be a nice thing to do)
* Remember to add your name and contact information, we like to give credit to those that contributed, besides people might want to contact you with questions

# Micro-controllers

## Digital Pins

The pins on the Arduino can be configured as either inputs or outputs. While the title of this document refers to digital pins, it is important to note that vast majority of Arduino (Atmega) analog pins, may be configured, and used, in exactly the same manner as digital pins.

### Properties of Pins Configured as INPUT

Arduino (Atmega) pins default to inputs, so they don't need to be explicitly declared as inputs with pinMode() when you're using them as inputs.

### Pullup Resistors with pins configured as INPUT

Often it is useful to steer an input pin to a known state if no input is present. This can be done by adding a pullup resistor (to +5V), or a pulldown resistor (resistor to ground) on the input. A 10K resistor is a good value for a pullup or pulldown resistor.

The pullup resistors are controlled by the same registers (internal chip memory locations) that control whether a pin is HIGH or LOW. Consequently, a pin that is configured to have pullup resistors turned on when the pin is an INPUT, will have the pin configured as HIGH if the pin is then switched to an OUTPUT with pinMode(). This works in the other direction as well, and an output pin that is left in a HIGH state will have the pullup resistors set if switched to an input with pinMode().

### Properties of Pins Configured as INPUT\_PULLUP

There are 20K pullup resistors built into the Atmega chip that can be accessed from software. These built-in pullup resistors are accessed by setting the pinMode() as INPUT\_PULLUP. This effectively inverts the behavior of the INPUT mode, where HIGH means the sensor is off, and LOW means the sensor is on.

NOTE: Digital pin 13 is harder to use as a digital input than the other digital pins because it has an LED and resistor attached to it that's soldered to the board on most boards. If you enable its internal 20k pull-up resistor, it will hang at around 1.7V instead of the expected 5V because the onboard LED and series resistor pull the voltage level down, meaning it always returns LOW. If you must use pin 13 as a digital input, set its pinMode() to INPUT and use an external pull down resistor.

### Properties of Pins Configured as OUTPUT

Short circuits on Arduino pins, or attempting to run high current devices from them, can damage or destroy the output transistors in the pin, or damage the entire Atmega chip. It is a good idea to connect OUTPUT pins to other devices with 470Ω or 1k resistors, unless maximum current draw from the pins is required for a particular application.

## Analog Input Pins

A description of the analog input pins on an Arduino chip (ATmega8, ATmega168, ATmega328P(UNO R3), or ATmega1280).

### A/D converter

The ATmega controllers used for the Arduino contain an onboard 6 channel (8 channels on the Mini and Nano, 16 on the Mega) analog-to-digital (A/D) converter. The converter has 10 bit resolution, returning integers from 0 to 1023. While the main function of the analog pins for most Arduino users is to read analog sensors, the analog pins also have all the functionality of general purpose input/output (GPIO) pins (the same as digital pins 0 - 13).

Consequently, if a user needs more general purpose input output pins, and all the analog pins are not in use, the analog pins may be used for GPIO.

### Pin mapping

The analog pins can be used identically to the digital pins, using the aliases A0 (for analog input 0), A1, etc.

pinMode(A0, OUTPUT);

digitalWrite(A0, HIGH);

### Pull-up resistors

The analog pins also have pull-up resistors, which work identically to pull-up resistors on the digital pins.

pinMode(A0, INPUT\_PULLUP); // set pull-up on analog pin 0,

// same as a digital pin

### Details and Caveats

The analogRead command will not work correctly if a pin has been previously set to an output, so if this is the case, set it back to an input before using analogRead. Similarly if the pin has been set to HIGH as an output, the pull-up resistor will be set, when switched back to an input.

The ATmega datasheet also cautions against switching analog pins in close temporal proximity to making A/D readings (analogRead) on other analog pins. This can cause electrical noise and introduce jitter in the analog system. It may be desirable, after manipulating analog pins (in digital mode), to add a short delay before using analogRead() to read other analog pins.

### Basics of PWM (Pulse Width Modulation)

Pulse Width Modulation, or PWM, is a technique for getting analog results with digital means.

 Arduino's PWM frequency at about 500Hz. A call to [analogWrite](https://arduino.cc/en/Reference/AnalogWrite)() is on a scale of 0 - 255, such that analogWrite(255) requests a 100% duty cycle (always on), and analogWrite(127) is a 50% duty cycle (on half the time) for example.

On some microcontrollers PWM is only available on selected pins. They are denoted with a tilde sign (~).

Once you get this example running, grab your Arduino and shake it back and forth. What you are doing here is essentially mapping time across the space. To our eyes, the movement blurs each LED blink into a line. As the LED fades in and out, those little lines will grow and shrink(收缩，变小) in length. Now you are seeing the pulse width.

## Debugging Fundamentals

Embedded systems integrate all the hardware and software necessary for a particular purpose.Debugging is the process of confirming that, one by one, many things that we believe to be true and functional in our code are true. We find a "bug" in our code when one our more assumptions are not valid.

### Debugging tools and techniques

There are some basic debugging tools and techniques that we can use and implement to validate our code:

* The compiler and syntax errors.
* Traditional techniques: trace code and GPIOs.
* Remote debuggers.
* Simulators.
* In-circuit emulators and in-circuit debuggers.
* Hardware tools: multimeters, logic analyzers, oscilloscopes, and software-defined radios.

### **The Compiler and Syntax Errors**

Using the compiler for debugging syntax errors can be sometimes tricky; let us analyze two commonly encountered situations:

* **The compiler shows 100 errors**: **Only the first error message is genuinely reliable**; try to fix one error at a time and then recompile the program.
* **Getting weird compiler messages**: **Read the error message carefully**; it will always tell where, inside the code, the error occurred.

### **Traditional Techniques: Trace Code and GPIO's**

 For example, determining if a particular function is halting or freezing in our code can be made with trace code as shown in the example code below:

// Print a message if the execution gets here

Serial.println("Code got here");

// Try to execute myFunction1()

myFunction1();

// Print a message if the execution gets here

Serial.println("Code got here, myFunction1 executed");

// Try to execute myFunction2()

myFunction2();

// Print a message if the execution gets here

Serial.println("Code got here, myFunction2 executed");

**Adding trace code to our programs takes a significant amount of processing time and resources.** Therefore, it can easily disrupt critical timing tasks in our programs.

You can pass flash-memory based Strings to

Serial.print() instruction by wrapping them with F(); for example,

Serial.println(F("Code got here")) prints a flash-memory based String.

Another trace code technique consists of dumping strategic information into an array at run time, we can then observe the contents of the array at a later time (for example, when the program terminates); this technique is also known as dump into array.

General Purpose Input/Output (GPIO) pins can help debug purposes when the UART is in use or adding trace code is not particularly helpful.

// Print a message if the execution gets here

Serial.println("Code got here");

// Try to execute myFunction1()

myFunction1();

// Turn on the built-in LED for one second to indicate that myFunction1 was executed

digitalWrite(LED\_BUILTIN, HIGH);

delay(1000);

digitalWrite(LED\_BUILTIN, LOW);

// Try to execute myFunction2()

myFunction2();

// Turn on the built-in LED for one second to indicate that myFunction2 was executed

digitalWrite(LED\_BUILTIN, HIGH);

delay(1000);

digitalWrite(LED\_BUILTIN, LOW);

### **Remote Debuggers**

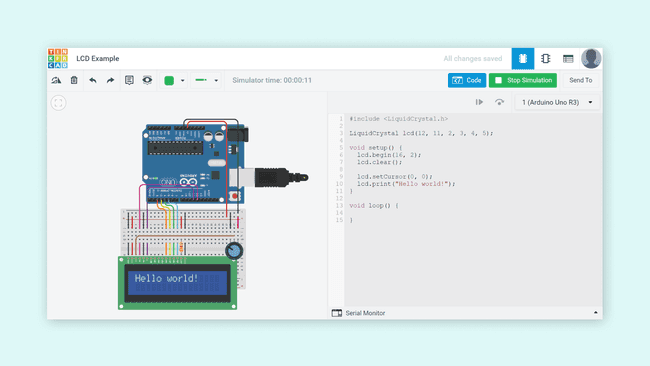
Remote debugging is another common approach used to debug embedded systems. Remote debuggers work by connecting a particular embedded system to a host computer and then using software in the host computer to interact with the embedded system hardware. Remote debuggers are helpful when the development environment is on a different architecture rather than the target system. For example, think about developing code on a Windows-based computer for an ARM-based microcontroller system.

Remote debuggers usually have two essential parts: a front-end debugger and a back-end debugger.

* The front-end debugger contains the user interface (can be graphical or command-line-based) and offers the programmer choices about the execution of the code in the embedded system hardware.
* The back-end debugger, also known as the "debug monitor," is specific for a particular processor architecture or family an usually work with an external hardware tool, like an in-circuit emulator（模拟器） or an in-circuit debugger. It starts when the processor resets and handles the runtime instruction between the front-end debugger and the embedded system hardware.

### **Simulators（仿真器）**

Simulators are tools used to simulate the functionality and the instruction set of the target processor. These tools, usually, can only simulate the target processor functionalities but not its environment and external parts and components. Simulators are handy in the early stages of the development process, where we only have the software but have not implemented any hardware yet.

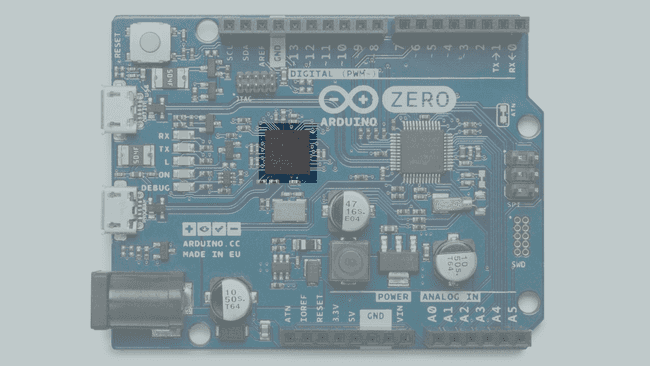


A simulation of a circuit in Tinkercad Circuits.

### **In-Circuit Emulators and In-Circuit Debuggers**

An in-circuit emulator (or ICE) is a specialized tool that allows developers to examine the state of the processor while a particular program is running. ICEs are considered embedded systems by themselves; they are a copy of the target processor and its memory (RAM and ROM); this is why they provide an unintrusive way to debug code at the target processor. Historically, ICEs were the tool of choice of embedded systems developers, but as processor complexity and clock speed increased, ICEs became more expensive, and their availability declined considerably.

An in-circuit debugger (or ICD) is also a specialized tool connected between a host computer and a processor for debugging real-time applications faster and easier; this tool uses some memory and GPIO pins of the target microcontroller during the debugging operations. With an ICD, developers access an on-chip debug module which is integrated into the CPU over an interface (for example, JTAG). This debug module allows developers to load, run, halt and step the processor.



Arduino® Zero EDGB.

Arduino® boards with a SAMD microcontroller feature native on-chip debug capabilities; these debugging capabilities can be used with an external ICD tool over JTAG or SWD interfaces. CMSIS-DAP compliant debug probes can be used with the Arduino IDE 2 out of the boxwithout any configuration file（开箱即用，无需配置）; non-standard debug probes require a special configuration. Check out these tutorials to learn how to use an external ICD tool with SAMD based Arduino boards and the Arduino IDE 2:

* **[Debugging with the SEGGER J-Link](https://docs.arduino.cc/tutorials/mkr-wifi-1010/mkr-jlink-setup)**.
* **[Debugging with the Atmel-ICE](https://docs.arduino.cc/tutorials/mkr-wifi-1010/atmel-ice)**.

The [Arduino® Portenta H7](https://store.arduino.cc/products/portenta-h7), [H7 Lite](https://store.arduino.cc/products/portenta-h7-lite), and [H7 Lite Connected](https://store.arduino.cc/products/portenta-h7-lite-connected) boards from the [Pro family](https://www.arduino.cc/pro) also support ICD debugging; these boards use the TRACE32 debugger from Lauterbach. The TRACE32 debugger allows testing embedded hardware and software by using the in-circuit debug interface of processors. Check out this tutorial to learn how to use the TRACE32 debugger with the Portenta family boards:

* **[Lauterbach TRACE32 GDB Front-End Debugger for Portenta H7](https://docs.arduino.cc/tutorials/portenta-h7/lauterbach-debugger)**.

### **Hardware Tools**

These tools are multimeters, logic analyzers, oscilloscopes, and software-defined radios (SDRs).

#### **Software-Defined Radios**

A software-defined radio (SDR) is a radio communication system that uses software for the modulation and demodulation of radio signals. Traditional radio communications systems processing relies on hardware components; this limits their reprogramming to a very low level. SDRs are much more flexible since they can be reconfigured by software.

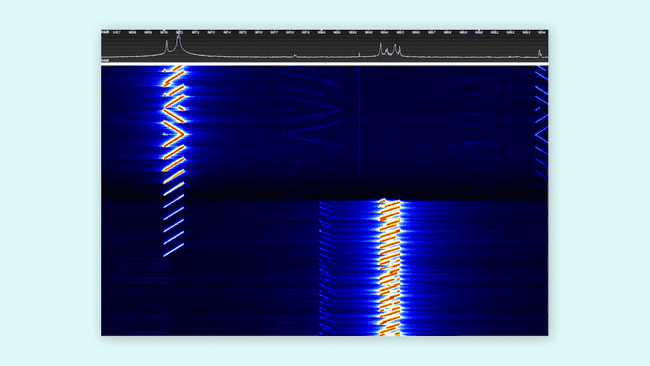
How do we debug wireless communications between devices?

A simple and common technique used to debug wireless communications between two or more devices consists of using acknowledge flags. Using acknowledge flags aids in understanding the device's behavior when communication is established between them by providing their current status. This process is also found on physical communication protocols, such as I2C or SPI. Because of the different protocol types in wireless communications, using acknowledge flags as a debugging method can be applied but differ in their own rules. The easiest way to confirm that the data exchange was successful is to check the data log on each end device. Hardware tools mentioned earlier (DMMs, oscilloscopes, and logic analyzers) can be used to provide more in-depth details and add more value to this type of debugging technique.

However, not everything is connected on a physical layer, but on an abstract layer. This abstract layer in wireless communications is where electromagnetic waves propagate through space. Why would we need to debug electromagnetic waves propagating through space? Sometimes, we need to verify that the wireless transceiver configuration of a particular embedded system is correct, for example, its transmission power. SDRs can be helpful in this situation: SDRs can be used as cheap spectrum analyzers. A spectrum analyzer is a hardware tool that measures the magnitude of a signal versus the frequency; this tool is mainly used to measure a signal's power spectrum. Using SDRs as spectrum analyzers is usually an optional debugging method in the development process, ensuring a more robust system design.

Several programs can be used with SDRs; **[GQRX](https://gqrx.dk/)** is one of the most popular software; GQRX is open-source and with cross-platform support (Linux and macOS). **[AirSpy](https://airspy.com/)** and **[CubicSDR](https://cubicsdr.com/)** are other popular software used with SDRs and cross-platform support (Windows, Linux, and macOS).

Shown visual representation of the signal via SDR software can now be used to verify the transmission power outputted by the device and the amount of data transmitted. This will help visualize the device's wireless communication configuration properties. It will be possible to verify the transmission and reception power, the number of bytes transmitted, and the frequency it is supposed to transmit. These properties can be debugged through the frequency spectrum and refined to provide edge wireless communication performance on embedded systems.



LoRa signals shown in a spectogram. Source: The Things Network.

### Final Thoughts about Debugging

We can end this article by mentioning the four most essential phases of debugging stated by Robin Knoke in this [article](https://www.embedded.com/debugging-embedded-c/) about debugging embedded C that was published in the Embedded Systems Programming magazine:

* **Testing**: this phase exercises the capability of the embedded software by stimulating it with a wide range of input values and in different environments.
* **Stabilization**: this phase attempt to control the conditions that generate a specific bug.
* **Localization**: this phase involves narrowing the range of possibilities until the bug can be isolated to a specific code segment in the embedded software.
* **Correction**: this phase involves eradicating the bug from the software.

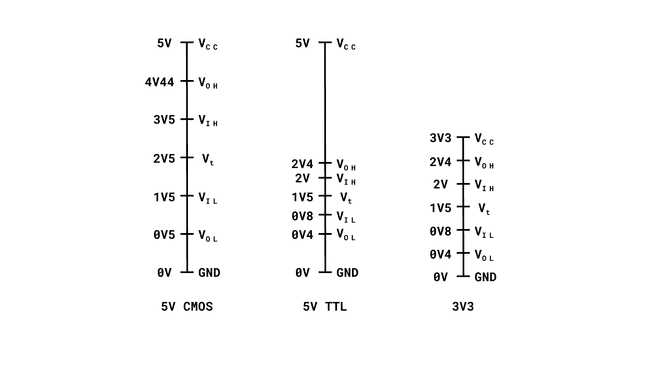
### Further Reading and Resources

* Do you want to improve your debugging and engineering skills? A highly recommended reading is **[Debugging: The 9 Indispensable(不可获取的) Rules for Finding Even the Most Elusive Software and Hardware Problems](http://debuggingrules.com/)** by David J. Agans.
* Do you want to learn more about digital multimeters? Learn more about them in **[this](https://www.fluke.com/en-us/learn/blog/electrical/what-is-a-digital-multimeter)** article from Fluke®.
* Do you want to learn more about oscilloscopes? Learn more about them in **[this](https://www.tek.com/en/blog/what-is-an-oscilloscope)** article from Tektronix®.
* Do you want to learn more about logic analyzers? Learn more about them in **[this](https://articles.saleae.com/logic-analyzers/what-is-a-logic-analyzer)** article from Saleae®.
* Do you want to learn more about spectrum analyzers? Learn more about them in **[this](https://www.tek.com/en/documents/primer/what-spectrum-analyzer-and-why-do-you-need-one)** article from Tektronix®.
* Do you want to learn more about SDRs? Check out the Great Scott Gadgets **[video series](https://greatscottgadgets.com/sdr/)** about SDRs. The video series from Great Scott Gadgets is a complete course about SDRs. You will learn the fundamentals of Digital Signal Processing (DSP) and build flexible SDR applications using GNU Radio.
* P. Koopman, Better Embedded System Software. S.L.: Drumnadrochit Press, 2010.
* M. Barr and A. Massa, Programming Embedded Systems: with C and GNU Development Tools. Johanneshov: MTM, 2013.

## Guide to 3V3 and 5V Power Supplies Differences

### 3V3, the Standard Voltage Level

A standard exists for defining the voltage levels of input and output voltages for every power supply voltage level; this standard was developed by The Joint Electron Device Engineering Council (JEDEC，联合电子器件工程委员会); it is the JEDEC Standard 8-A for LV interface levels. The JEDEC Standard 8-A for LV interface levels is described in the image shown below for 3V3 and 5V logic families:



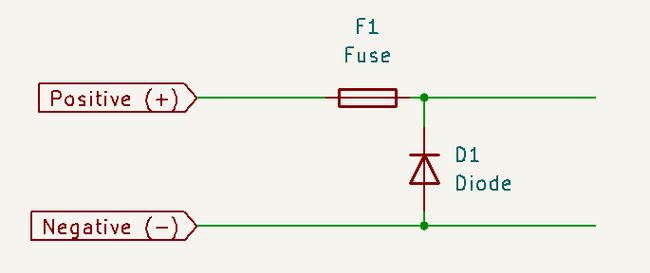
### How to Avoid Burning Circuits

#### Color Coded Power Lines

Color coding the power lines is the easiest yet most effective visual method to avoid an incorrect connection in the power lines of electronic circuits or devices. According to industry regulations and standards, red is typically used to indicate a voltage line, while black is used to indicate a GND line; colors vary depending on the regulation or standard.

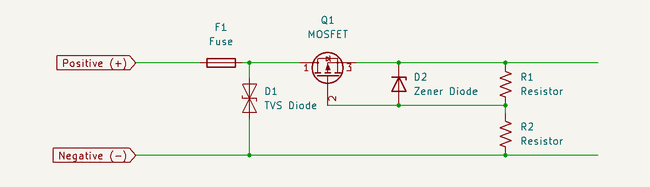
#### Fuse Integration

Fuse integration to electronic devices is not a complicated design process. The following schematic shows a simple reverse polarity protection circuit that can be used in low-power DC circuits:



Simple reverse polarity protection circuit.

#### A Better Reverse Polarity Protection Circuit



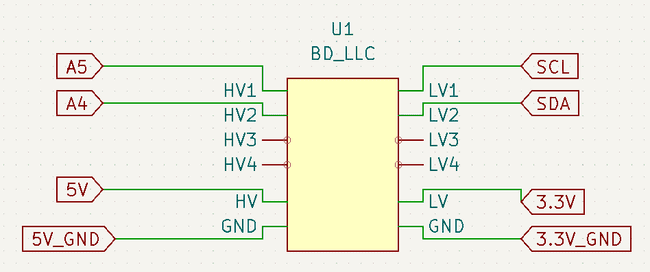
Complete Reverse Polarity Protection

The key points of the circuit presented above are the Transient Voltage Suppressor diode and the MOSFET of P-Channel type. This protection circuit will help you save the Protected Load and to have it as a good reference for protection design. See notebook -> Regular “battery reverse protection circuit”.

### Stepping the Voltage - Level Shifters

#### Bi-Directional Logic Level Converter

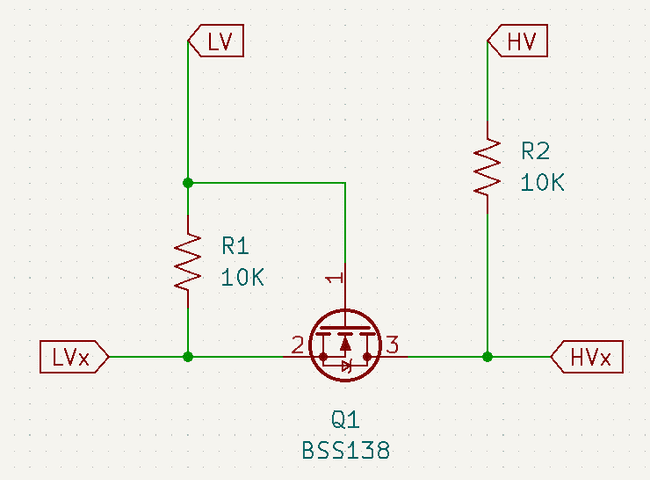
You can use the off-the-shelf(现成的) **[Bi-Directional Logic Level Shifter](https://www.sparkfun.com/products/12009)** from SparkFun to test and also for deployment if the requirements enables its integration. The advantage of this particular shifter is that it provides 4 channels to shift within the voltage references given. High Voltage level and Low Voltage level references are injected with desired voltage level and channels are used to transmit the data in between.



Voltage Stepping- Logic Shifter

The circuit above uses the bi-directional logic shifter to establish I2C interface with any sensor capable of the protocol. The SCL and SDA lines go through a High Voltage channel and establishes communication with the sensor that is connected at its respective Low Voltage Channel.

The configuration of the Logic Shifter usually does not change, as the the purpose is to transmit the signal from a High to a Low Level or vice-versa, depending on the architecture operation. Thus, the previous schematic illustrates usual global connection configuration. As it can be to interface the Arduino board with another computing module working on a different voltage level. Below schematic shows the specific of each channel and focus the scope inside the schematic symbol box of the Logic Shifter.



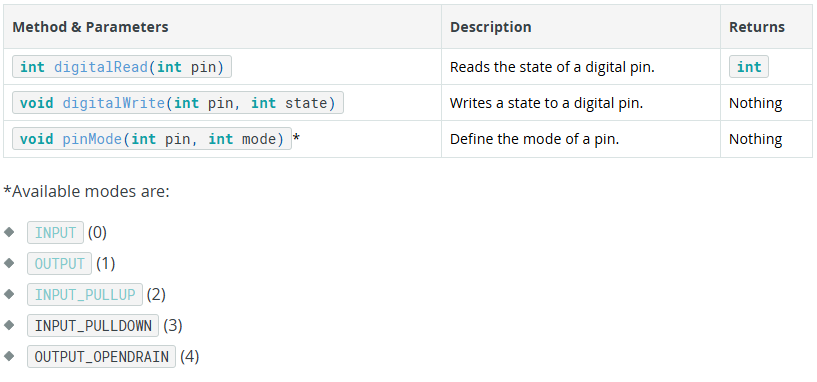
Logic Shifter Insight

# Programming

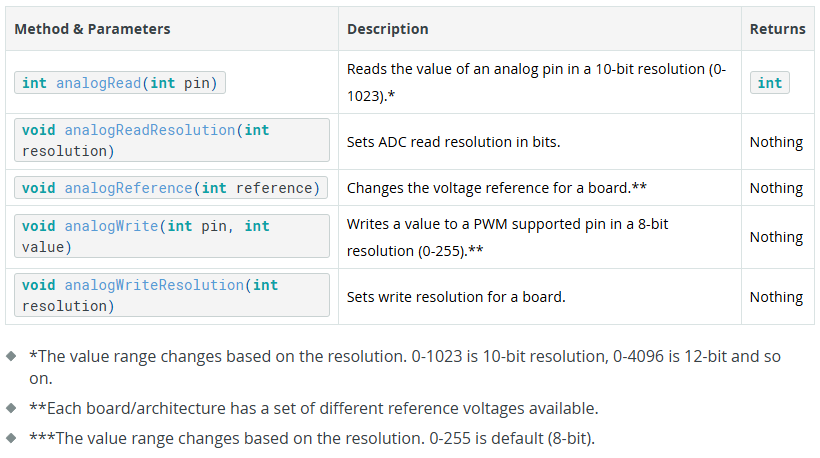
## Arduino API

### Functions

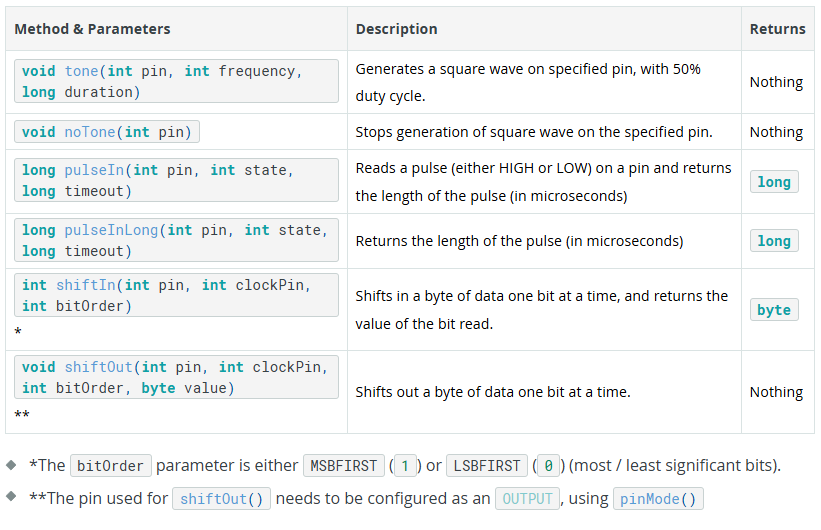
#### Digital I/O



#### Analog I/O



#### Advanced I/O



#### Time

| **Method & Parameters** | **Description** | **Returns** |
| --- | --- | --- |
| **void** delay(**long** milliseconds) | Freezes program execution for specified number of **milliseconds**. | Nothing |
| **void** delayMicroseconds(**int** microseconds) | Freezes program execution for specified number of **microseconds**. | Nothing |
| **long** millis() | Returns **milliseconds** passed since program start. | **long** |
| **long** micros() | Returns **microseconds** passed since program start. | **long** |

#### Math

| **Method & Parameters** | **Description** | **Returns** |
| --- | --- | --- |
| **int** abs(**int** value) | Calculates the absolute value of a number. | **int** |
| **int** constrain(**int** value, **int** min, **int** max) | Constrains a number to be within a range. | **int** |
| **long** map(**long** val, **long** min, **long** max, **long** newMin, **long** newMax) | Re-maps a number from one range to another. | **long** |
| **int** max(**int** val1, **int** val2) | Returns the greater of two values. | **int** |
| **int** min(**int** val1, **int** val2) | Returns the smaller of two values. | **int** |
| **double** pow(**double** base, **double** exponent) | Raises a base to the power of an exponent. | **double** |
| **int** sq(**int** value) | Calculates the square of a number. | **int** |
| **double** sqrt(**double** value) | Calculates the square root of a number. | **double** |

**double** pow(**double** base, **double** exponent)，计算base的exponent次幂。

#### Trigonometry

**三角学，**/ˌtrɪɡəˈnɒmətri/

| **Method & Parameters** | **Description** | **Returns** |
| --- | --- | --- |
| cos(**double** angle) | Calculates the cosine of an angle in radians. | **double** |
| sin(**double** angle) | Calculates the sine of an angle in radians. | **double** |
| tan(**double** angle) | Calculates the tangent of an angle in radians. | **double** |

#### Characters

| **Method & Parameters** | **Description** | **Returns** |
| --- | --- | --- |
| **boolean** isAlpha(char c) | Checks if the character is an alphabetic character. | **boolean** |
| **boolean** isAlphaNumeric(char c) | Checks if the character is an alphanumeric character. | **boolean** |
| **boolean** isAscii(char c) | Checks if the character is a 7-bit ASCII character. | **boolean** |
| **boolean** isControl(char c) | Checks if the character is a control character. | **boolean** |
| **boolean** isDigit(char c) | Checks if the character is a digit (0-9). | **boolean** |
| **boolean** isGraph(char c) | Checks if the character is a printable character, excluding space. | **boolean** |
| **boolean** isHexadecimalDigit(char c) | Checks if the character is a hexadecimal digit (0-9, A-F, a-f). | **boolean** |
| **boolean** isLowerCase(char c) | Checks if the character is a lowercase alphabetic character. | **boolean** |
| **boolean** isPrintable(char c) | Checks if the character is a printable character, including space. | **boolean** |
| **boolean** isPunct(char c) | Checks if the character is a punctuation character. | **boolean** |
| **boolean** isSpace(char c) | Checks if the character is a whitespace character. | **boolean** |
| **boolean** isUpperCase(char c) | Checks if the character is an uppercase alphabetic character. | **boolean** |
| **boolean** isWhitespace(char c) | Checks if the character is a whitespace character according to  isSpaceChar() method. | **boolean** |

#### Random Numbers

| **Method & Parameters** | **Description** | **Returns** |
| --- | --- | --- |
| **int** random() | Generates a pseudo-random number between 0 and RAND\_MAX. | **int** |
| **void** randomSeed(unsigned **long** seed) | Seeds the random number generator. | Nothing |

#### Bits and Bytes

| **Method & Parameters** | **Description** | **Returns** |
| --- | --- | --- |
| **boolean** bit(**int** value, **int** bitNumber) | Gets the value of a specific bit. | **boolean** |
| **void** bitClear(**int** &value, **int** bit) | Clears a specific bit. | Nothing |
| **boolean** bitRead(**int** value, **int** bitNumber) | Reads the value of a specific bit. | **boolean** |
| **void** bitSet(**int** &value, **int** bit) | Sets a specific bit. | Nothing |
| **void** bitWrite(**int** &value, **int** bit, **int** bitValue) | Writes a value to a specific bit. | Nothing |
| **byte** highByte(**int** value) | Returns the high byte of an **int**. | **byte** |
| **byte** lowByte(**int** value) | Returns the low byte of an **int**. | **byte** |

**byte** highByte(**int** value)

Extracts the high-order (leftmost) byte of a word (or the second lowest byte of a larger data type).

**boolean** bit(**int** value, **int** bitNumber)

Computes the value of the specified bit (bit 0 is 1, bit 1 is 2, bit 2 is 4, etc.). n: the bit whose value to compute. Note that n needs to be between 0-31 (32 bit). 这与函数原型矛盾，需验证。

#### Interrupts

| **Method & Parameters** | **Description** | **Returns** |
| --- | --- | --- |
| **void** interrupts() | Enables interrupts globally. | Nothing |
| **void** noInterrupts() | Disables interrupts globally. | Nothing |

#### External Interrupts

| **Method & Parameters** | **Description** | **Returns** |
| --- | --- | --- |
| **void** attachInterrupt(**int** pin, **void** (\***function**)(**void**), **int** mode) | Attaches an interrupt to a specific pin. | Nothing |
| **void** detachInterrupt(**int** pin) | Detaches an interrupt from a specific pin. | Nothing |

**void** attachInterrupt(**int** pin, **void** (\***function**)(**void**), **int** mode)

#### Stream

| **Method & Parameters** | **Description** | **Returns** |
| --- | --- | --- |
| **int** available() | Returns the number of bytes available in the serial buffer. | **int** |
| **int** read() | Reads the next byte from the serial buffer. | **int** |
| **void** flush() | Waits for the transmission of outgoing serial data to complete. | Nothing |
| **int** find(char \*target) | Searches for a target string in the serial buffer. | **int** |
| **int** findUntil(char \*target, char \*terminate) | Searches for a target string until a specified termination string is found. | **int** |
| **int** peek() | Returns the next byte in the serial buffer without removing it. | **int** |
| **int** readBytes(char \*buffer, **int** length) | Reads characters from the serial buffer into a buffer. | **int** |
| **int** readBytesUntil(char terminator, char \*buffer, **int** length) | Reads characters from the serial buffer into a buffer until a terminator is found. | **int** |
| **String** readString() | Reads characters from the serial buffer into a String until a newline character is found. | **String** |
| **String** readStringUntil(char terminator) | Reads characters from the serial buffer into a String until a specified terminator is found. | **String** |
| **int** parseInt() | Reads characters from the serial buffer and converts them to an integer. | **int** |
| float parseFloat() | Reads characters from the serial buffer and converts them to a float. | float |
| **void** setTimeout(unsigned **long** timeout) | Sets the maximum duration for  find(), findUntil(), parseInt(), and parseFloat(). | Nothing |

#### Serial

| **Method & Parameters** | **Description** | **Returns** |
| --- | --- | --- |
| **if**(Serial) | Checks if the Serial object is available. | **boolean** |
| **int** available() | Returns the number of bytes available for reading. | **int** |
| **int** availableForWrite() | Returns the number of bytes available for writing. | **int** |
| **void** begin(unsigned **long** baudrate) | Initializes the Serial communication with the specified baud rate. | **void** |
| **void** end() | Ends the Serial communication. | **void** |
| **int** find(char \*target) | Searches for a target string in the serial buffer. | **int** |
| **int** findUntil(char \*target, char \*terminate) | Searches for a target string until a specified termination string is found. | **int** |
| **void** flush() | Waits for the transmission of outgoing serial data to complete. | **void** |
| float parseFloat() | Reads characters from the serial buffer and converts them to a float. | float |
| **int** parseInt() | Reads characters from the serial buffer and converts them to an integer. | **int** |
| **int** peek() | Returns the next byte in the serial buffer without removing it. | **int** |
| size\_t print() | Prints data to the serial port. | size\_t |
| size\_t println() | Prints data to the serial port followed by a newline character. | size\_t |
| **int** read() | Reads the next byte from the serial buffer. | **int** |
| **int** readBytes(char \*buffer, size\_t length) | Reads characters from the serial buffer into a buffer. | **int** |
| **int** readBytesUntil(char terminator, char \*buffer, size\_t length) | Reads characters from the serial buffer into a buffer until a terminator is found. | **int** |
| **String** readString() | Reads characters from the serial buffer into a String until a newline character is found. | **String** |
| **String** readStringUntil(char terminator) | Reads characters from the serial buffer into a String until a specified terminator is found. | **String** |
| **void** setTimeout(unsigned **long** timeout) | Sets the maximum duration for find(), findUntil(), parseInt(), and parseFloat(). | **void** |
| size\_t write(uint8\_t) | Writes a byte to the serial port. | size\_t |
| **void** serialEvent() | Called when data is available in the serial buffer. | **void** |

#### SPI

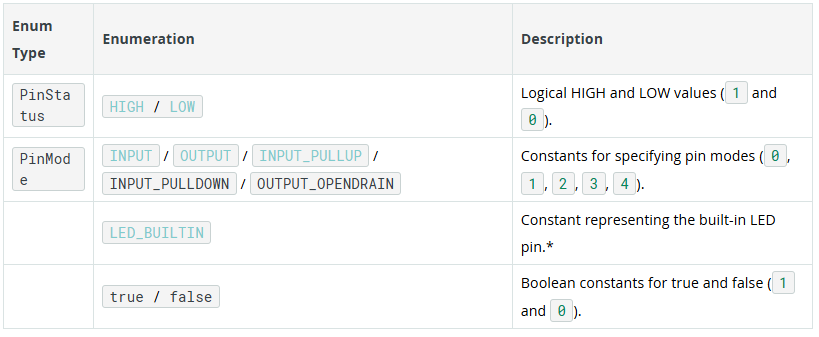
| **Method & Parameters** | **Description** | **Returns** |
| --- | --- | --- |
| SPISettings(uint32\_t clock, uint8\_t bitOrder, uint8\_t dataMode) | Creates an SPISettings object with the specified clock, bit order, and data mode. | SPISettings |
| **void** begin() | Initializes the SPI library. | **void** |
| **void** beginTransaction(SPISettings settings) | Begins an SPI transaction with the specified settings. | **void** |
| **void** endTransaction() | Ends the current SPI transaction. | **void** |
| **void** end() | Ends the SPI library. | **void** |
| **void** setBitOrder(uint8\_t bitOrder) | Sets the bit order (MSBFIRST or LSBFIRST) for SPI communication. | **void** |
| **void** setClockDivider(uint8\_t divider) | Sets the clock divider for SPI communication. | **void** |
| **void** setDataMode(uint8\_t dataMode) | Sets the data mode for SPI communication. | **void** |
| **byte** transfer(**byte** value) | Transfers a byte over SPI. | **byte** |
| **void** usingInterrupt(**int** interruptNumber) | Specifies which interrupt to use for SPI transactions. | **void** |

#### I2C (Wire)

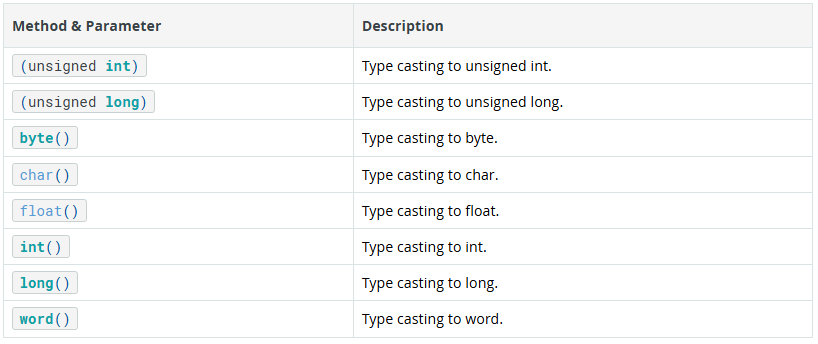
| **Method & Parameters** | **Description** | **Returns** |
| --- | --- | --- |
| **void** begin() | Initializes the Wire library. | **void** |
| **void** end() | Ends the Wire library. | **void** |
| **int** requestFrom(**int** address, **int** quantity) | Requests data from a slave device with the specified address and quantity of bytes. | **int** |
| **void** beginTransmission(**int** address) | Begins a transmission to the slave device with the specified address. | **void** |
| **int** endTransmission() | Ends the transmission and returns the status. | **int** |
| size\_t write(uint8\_t data) | Writes a byte to the I2C bus. | size\_t |
| **int** available() | Returns the number of bytes available for reading. | **int** |
| **int** read() | Reads a byte from the I2C bus. | **int** |
| **void** setClock(uint32\_t frequency) | Sets the I2C clock frequency. | **void** |
| **void** onReceive(**void** (\***function**)(**int**)) | Sets a function to be called when data is received by the slave. | **void** |
| **void** onRequest(**void** (\***function**)(**void**)) | Sets a function to be called when the master requests data from the slave. | **void** |
| **void** setWireTimeout(uint32\_t timeout) | Sets the timeout for I2C operations. | **void** |
| **void** clearWireTimeoutFlag() | Clears the timeout flag. | **void** |
| **bool** getWireTimeoutFlag() | Returns the timeout flag status. | **bool** |

### Variables

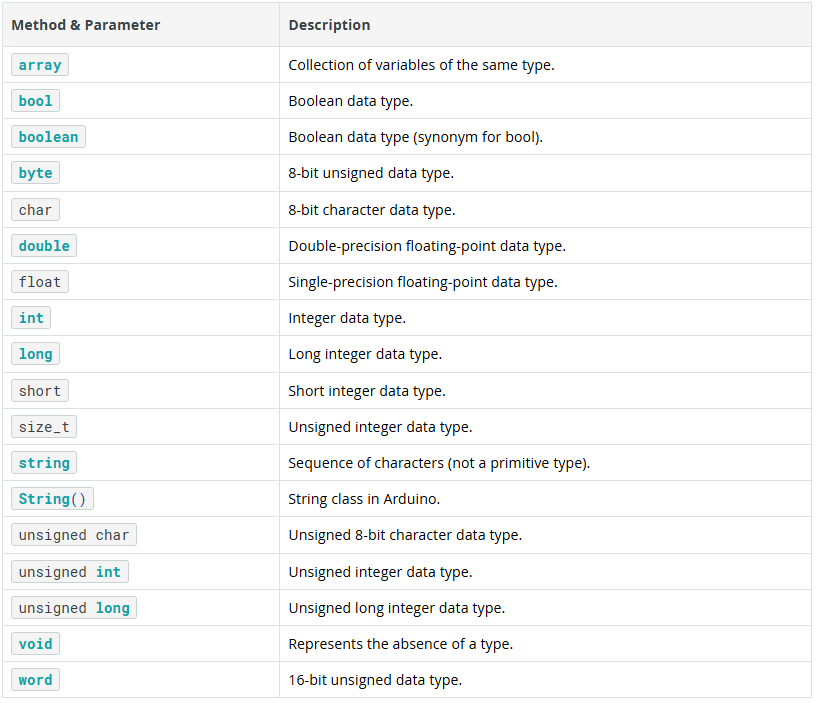
#### Enums



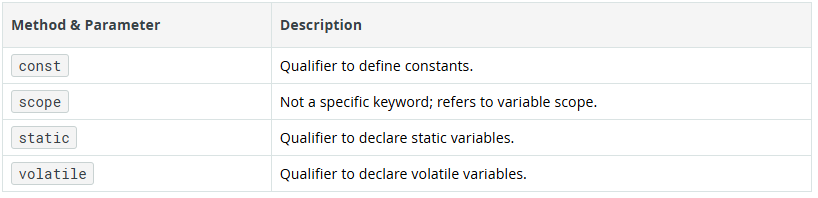
#### Conversion



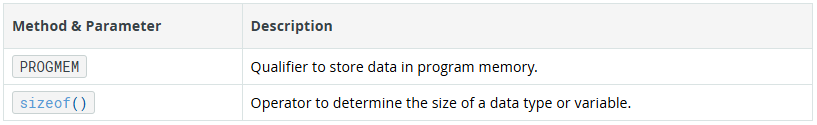
#### Data Types



#### Variable Scope & Qualifiers

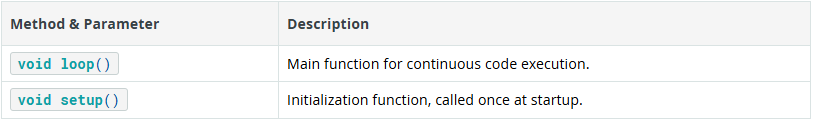


#### Utilities

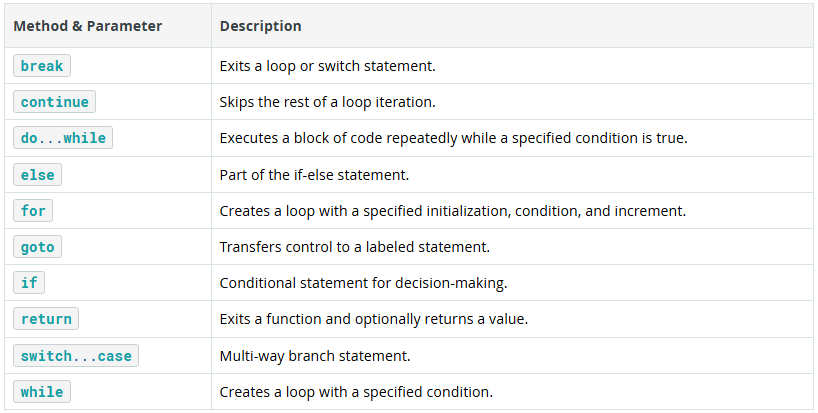


### Structure

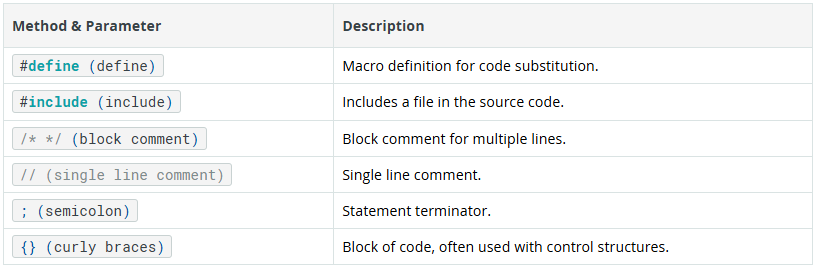
#### Sketch



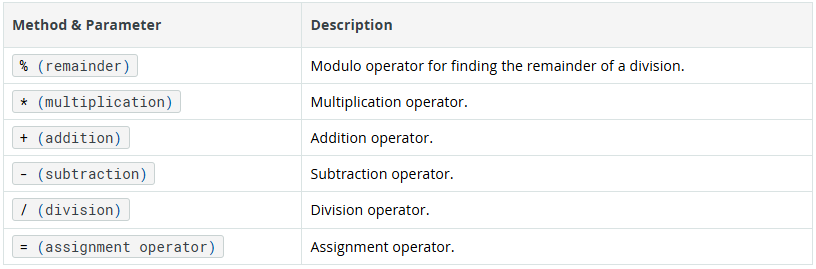
#### Control Structure



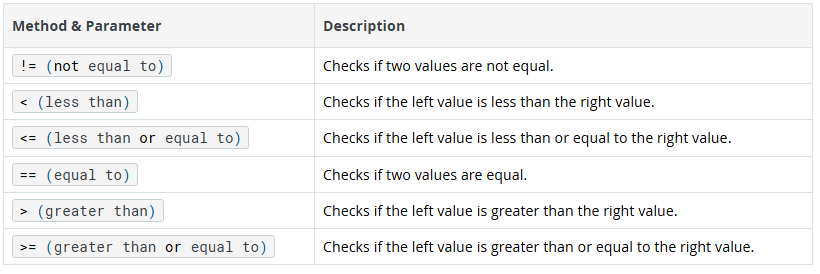
#### Further Syntax



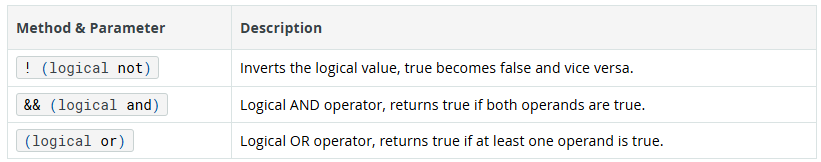
#### Arithmetic Operators



#### Comparison Operators

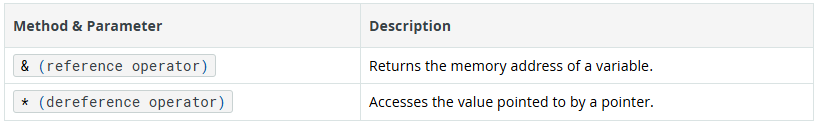


#### Boolean Operators

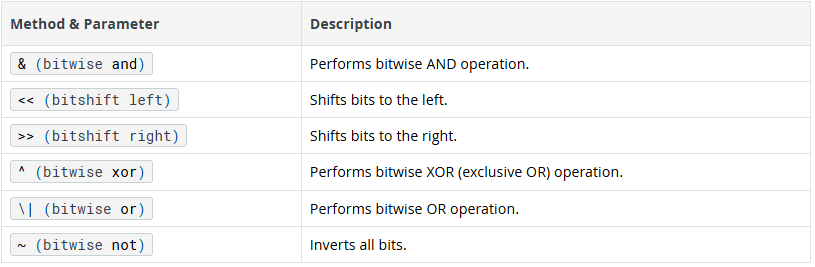


|| logic or operator

#### Pointer Access Operators

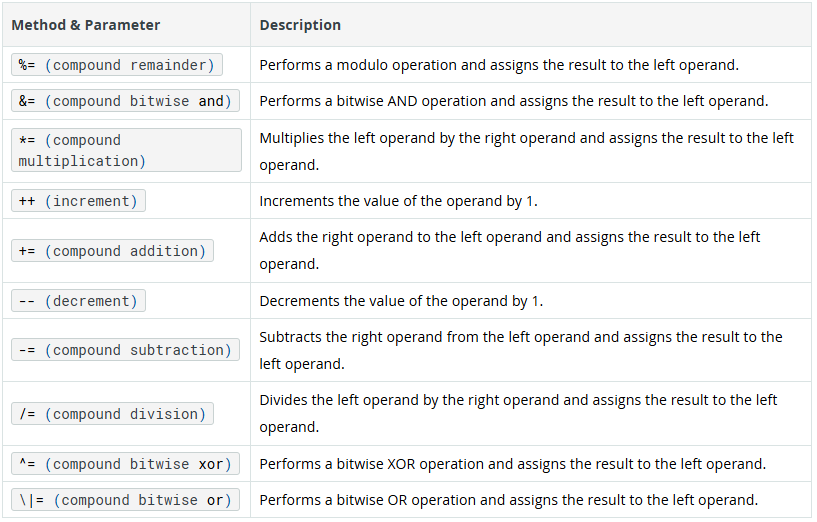


#### Bitwise Operators



| performs bitwise or operation.

#### Compound Operators



Left -= right: left = left - right

Left += right: left = left + right

## Using Variables in Sketches

## Using Functions in a Sketch

## Arduino Sketches

## FPGA HDL Basics

### Field Programmable Gate Arrays

Field Programmable Gate Arrays, in short [FPGAs](https://en.wikipedia.org/wiki/Field-programmable_gate_array) are a relatively old way of creating custom hardware eliminating the costs associated with silicon foundries. Unfortunately most of the complexity of chip design are still there and this is the reason why most people prefers to use off the shelf chips, often accepting their limitations, rather than take the challenge to have an optimized, efficient design with exactly the hardware they need.

 For FPGAs there are "libraries" called IP blocks, however these are usually quite expensive and lack a standardized "plug and play" interface which causes headaches when integrating everything in a system. What Arduino is trying to do introducing FPGAs in its product line is to take advantage of the flexibility of programmable hardware specifically to provide an extendable set of peripherals for microcontrollers taking away most of the complexity. Of course to achieve this it's necessary to impose some limitations and define a standard way to interconnect blocks so that it can be done automatically.

Since we are interfacing with a microcontroller the first port we need to define is a bus to interconnect processor with peripherals. Such bus should at least exist in the controller and peripheral flavors where signals are the same but with inverted directions. For some additional details on buses and controller/peripheral architecture please check this **[document](http://www.cut.ac.zw/escape/mchinyuku/1410878742.pdf)**.

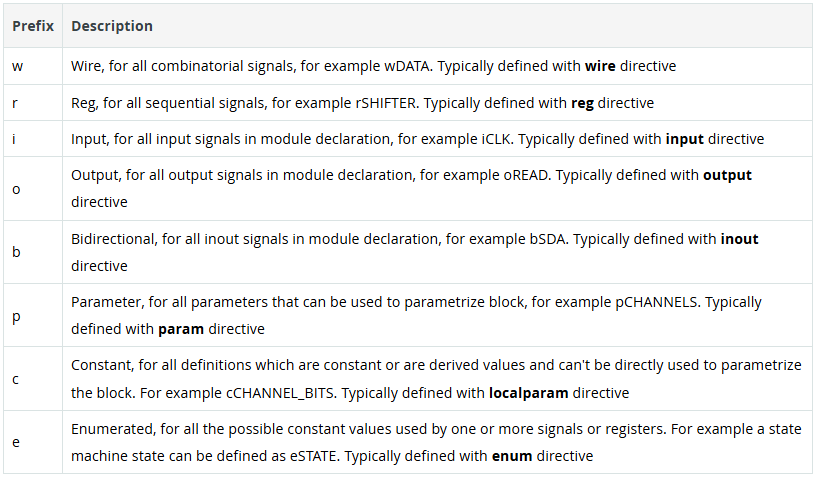
A second interface, which is important but can't be standardized is the input/output signals that connect to the external world. Here we can't define a standard as each block will provide its own set of signals however we can just bundle a set of signals in a group which we'll call a conduit(tunnel，/ˈkɒndjuɪt/).

Finally there is a third class of interfaces which may become useful which carries streaming data. In this case we want to transfer a continuous stream of data but also want to be able to pause the flow if the receiving block is not able to process it, hence along with data we also need some kind of flow control signals pretty much like it happens on a UART.

Since we want to standardize a bit also on readability we also want to set some coding conventions.

### Coding Conventions

We use a prefix in front of every declared entity so that it identifies its type, variable name is completely upper case and multiple words are separated by underscores. In particular:



We prefer spaces over tabs! The reason is that regardless of the tab size code always looks good.

Indentation is set to two spaces.

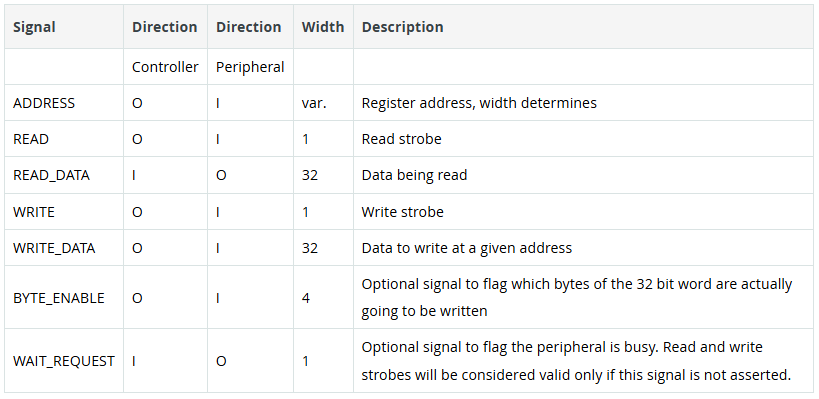
Conditional statement blocks shall always have begin/end constructs even if they have a single statement in them and begin/end should be on the same line of the if/else

Signals belonging to the same group shall share a common prefix

### Interface prototypes

#### Lightweight Bus

A bus to interconnect peripherals. By convention data bus is 32 bits while address bus is variable width, based on the number of registers being exposed. A bus requires the following set of signals:



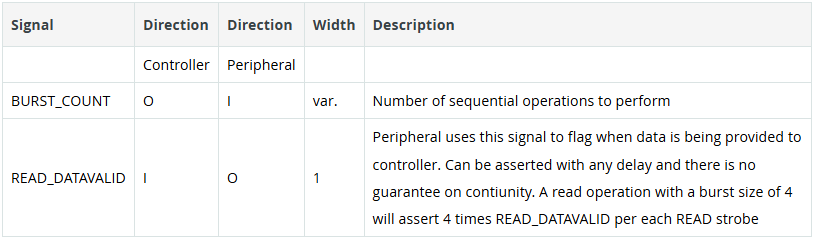
By convention in a write cycle, ADDRESS and WRITE\_DATA are latched in the same clock cycle of the WRITE strobe. By contrast in a read cycle, READ\_DATA is presented by peripheral on the cycle immediately following the READ strobe, which will also indicate the ADDRESS being read.

#### Pipelined Bus

A bus to interconnect complex blocks that can handle more than one command at time and respond in variable time to requests is pilelined bus. This bus extends the Lightweight bus by adding the following signals:

This behavior is also referred as 1 clock read latency and basically means that while peripheral can still have a variable number of clocks to respond to a READ or WRITE operation using the optional WAIT\_REQUEST signal, this would lock the controller preventing it to perform other operations.

In a way this can be considered similar to using busy loops in programming versus delays which yield to OS in order to do multitasking.



The main advantage of this approach is that controller can communicate to the peripheral the intention of reading or writing multiple data for each transaction. Both for read and write strobes in fact the BURST\_COUNT signal tells peripheral how long the transaction will be.

The controller will assert WAIT\_REQUEST until it is ready to accept an operation. In case of writes, BURST\_COUNT and ADDRESS are sampled only on the first strobe, after which peripheral will expect the WRITE strobe to be asserted for the number of words requested and will automatically increment address.

For read operations a single READ strobe, performed when WAIT\_REQUEST is not asserted, will tell the peripheral to read BURST\_COUNT words that will be returned by asserting READ\_DATAVALID for the requested number of words. After a read operation has been initiated it's up to(be up to：忙于……，胜任，达到，取决于) the peripheral to accept or not more operations but in general it should be possible to have at least two concurrent operations to benefit from the Pipelined Bus.

I suspect he is up to no good. 我怀疑他在搞鬼。

Is he up to driving after drinking so much?

The cost of the trip is up to 5000 yuan.

It's up to you to decide when we will start.

### Structure of a (System)Verilog module

A SystemVerilog module declaration can be done in several ways but the one we mostly prefer is the form where you can use parameters so that the block inputs can be customized at compile time. This would look like:

module COUNTER #(

pWIDTH=8

) (

input iCLK,

input iRESET,

output [pWIDTH-1:0] oCOUNTER

);

always @(posedge iCLK)

begin

if (iRESET) begin

oCOUNTER<=0;

end else begin

oCOUNTER<= oCOUNTER+1;

end

end

endmodule

 This way we can use the ’<=’ assignment which is "registered" which means that the assignment will be kept for as long as the next clock cycle in the always block (上下语句间并行执行，非阻塞赋值).The difference here is that  ’<=’ means the signal changes only at clock edges, while ’=’ assigns the value continuously so the signal will eventually change at any time. On the other hand, the verilog statments with ‘=’ in always block is sequential, this means upper assign statement will block lower statement(阻塞赋值)。

## Arduino Memory Guide

Memory in computing systems can be volatile or non-volatile. Volatile memory is a temporary memory, this means that data is stored while the system is running, but it is lost forever when the system is turned off. Non-volatile memory is permanent memory; data is not lost even if the system is turned off.

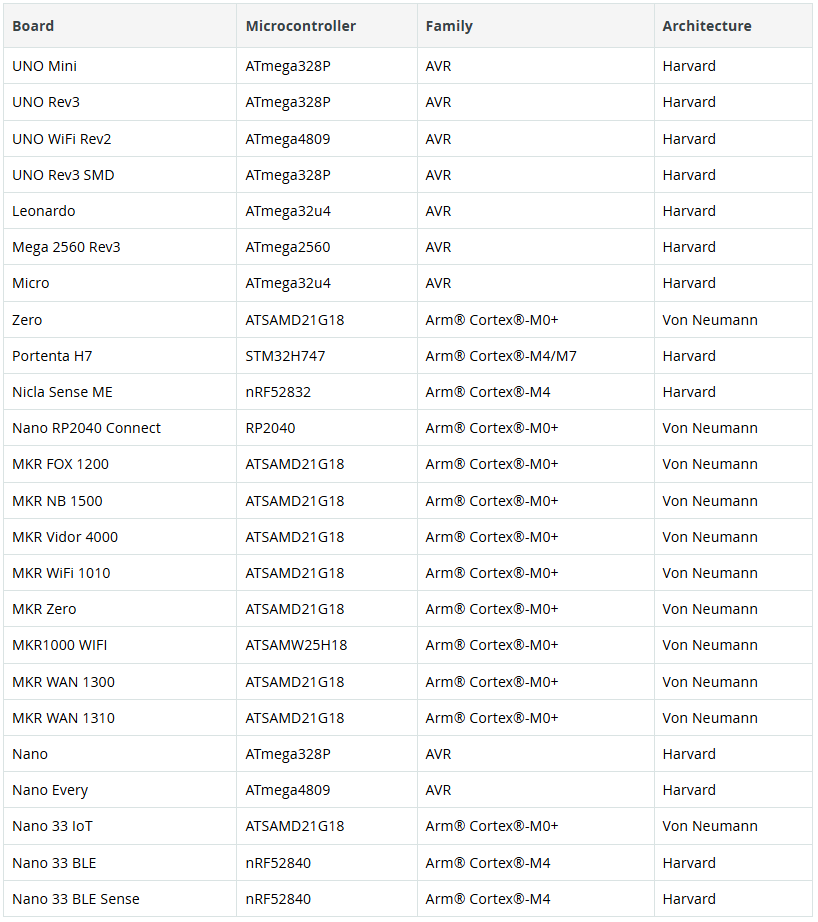
### Memory Architectures

In the early days of computing, two computer architectures, i.e., the organization of the components inside a computing system, emerged: **von Neumann** and **Harvard**.

Modern computing systems use **hybrid** architectures models that maximize performance using the best of both worlds, the von Neumann and the Harvard models.

Microcontrollers are usually used in embedded applications. They must perform defined tasks reliably and efficiently, with low or constrained resources; this is why the Harvard architecture model is mainly used in microcontrollers: microcontrollers have a small program and data memory that needs to be accessed simultaneously. However, Harvard architecture is not always used in microcontrollers; some microcontroller families use hybrid or Von Neumann architecture models.

Arduino® boards are mainly based on two families of microcontrollers: AVR® and ARM®. While AVR® family microcontrollers are based on the Harvard architecture model, ARM® family microcontrollers can be based on either von Neuman or Harvard architectures models. The following table summarizes Arduino boards microcontrollers architectures:

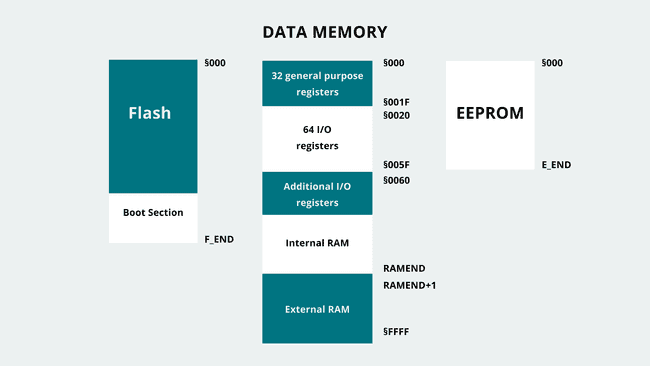


### Memory Types

见computer memory笔记。

### Arduino® Boards Memory Allocation

As stated before, Arduino® boards are mainly based on two families of microcontrollers, AVR® and ARM®; it is important to know that memory allocation differs in both architectures. In Harvard-based AVR architecture, memory is organized as shown in the image below:



AVR memory map.

Important to mention about AVR-based Arduino boards is how their SRAM is organized into different sections:

Text - contains instructions loaded into the flash memory

Data - contains variables initialized in the sketch

BSS - contains uninitialized data

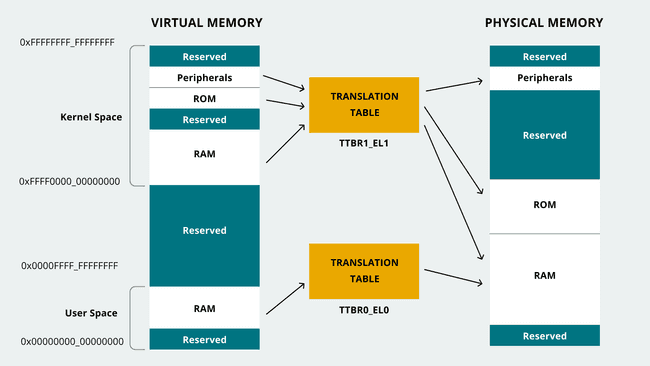
Stack - stores data of functions and interrupts

Heap - stores variables created during run time

In hybrid ARM architectures, a so called memory map is implemented, with a different address map configuration of 32-bit, 36-bit, and 40-bit that depends on the requirement of System On a Chip (SoC) address space with extra DRAM. The Memory Map grants interface with SoC design, while having most system control on a high level coding. Memory access instructions can be used on high level code to manage interrupt modules and built-in peripherals. All of this controlled by Memory Management Unit (MMU).

The memory resource is handled by the MMU. The main role of the MMU is to enable the processor to run multiple tasks independently in its own virtual memory space; the MMU then uses translation tables to establish a bridge between the virtual and the physical memory addresses. Virtual Address is managed via software with memory instructions, and Physical address is the memory system that is controlled depending on the Translation Table input given by the Virtual Address.

An example of how memory is organized in ARM-based microcontrollers, virtually and physically, is shown in the image below:



Memory organization in ARM-based microcontrollers.

The ARM-based microcontroller's memory is organized into the following sections within the address type mentioned previously:

* Virtual address:

Kernel code and data

Application code and data

* Physical address:

ROM

RAM

Flash

Peripherals

The following table summarizes a specific Arduino® board's memory allocation:

| Board | Microcontroller | Family | Architecture | Flash | SRAM | EEPROM |
| --- | --- | --- | --- | --- | --- | --- |
| UNO Mini | ATmega328P | AVR | Harvard | 32kB | 2kB | 1kB |
| UNO Rev3 | ATmega328P | AVR | Harvard | 32kB | 2kB | 1kB |
| UNO WiFi Rev2 | ATmega4809 | AVR | Harvard | 48kB | 6kB | 256B |
| UNO Rev3 SMD | ATmega328P | AVR | Harvard | 32kB | 2kB | 1kB |
| Leonardo | ATmega32u4 | AVR | Harvard | 32kB | 2.5kB | 1kB |
| Mega 2560 Rev3 | ATmega2560 | AVR | Harvard | 256kB | 8kB | 4kB |
| Micro | ATmega32u4 | AVR | Harvard | 32kB | 2.5kB | 1kB |
| Zero | ATSAMD21G18 | Arm® Cortex®-M0+ | Von Neumann | 256kB | 32kB | - |
| Portenta H7 (basic configuration) | STM32H747 | Arm® Cortex®-M4/M7 | Harvard | 16MB | 8MB | - |
| Nicla Sense ME | nRF52832 | Arm® Cortex®-M4 | Harvard | 512kB | 64kB | - |
| Nano RP2040 Connect | RP2040 | Arm® Cortex®-M0+ | Von Neumann | - | 264kB | - |
| MKR FOX 1200 | ATSAMD21G18 | Arm® Cortex®-M0+ | Von Neumann | 256kB | 32kB | - |
| MKR NB 1500 | ATSAMD21G18 | Arm® Cortex®-M0+ | Von Neumann | 256kB | 32kB | - |
| MKR Vidor 4000 | ATSAMD21G18 | Arm® Cortex®-M0+ | Von Neumann | 256kB | 32kB | - |
| MKR WiFi 1010 | ATSAMD21G18 | Arm® Cortex®-M0+ | Von Neumann | 256kB | 32kB | - |
| MKR Zero | ATSAMD21G18 | Arm® Cortex®-M0+ | Von Neumann | 256kB | 32kB | - |
| MKR1000 WIFI | ATSAMW25H18 | Arm® Cortex®-M0+ | Von Neumann | 256kB | 32kB | - |
| MKR WAN 1300 | ATSAMD21G18 | Arm® Cortex®-M0+ | Von Neumann | 256kB | 32kB | - |
| MKR WAN 1310 | ATSAMD21G18 | Arm® Cortex®-M0+ | Von Neumann | 256kB | 32kB | - |
| Nano | ATmega328P | AVR | Harvard | 32kB | 2kB | 1kB |
| Nano Every | ATmega4809 | AVR | Harvard | 48kB | 6kB | 256B |
| Nano 33 IoT | ATSAMD21G18 | Arm® Cortex®-M0+ | Von Neumann | 256kB | 32kB | - |
| Nano 33 BLE | nRF52840 | Arm® Cortex®-M4 | Harvard | 1MB | 256kB | - |
| Nano 33 BLE Sense | nRF52840 | Arm® Cortex®-M4 | Harvard | 1MB | 256kB | - |

### Measuring Memory Usage in Arduino® Boards

 software should always perform without reaching maximum load capacity to avoid problems or issues. Memory load could be observed either as available RAM at disposal for specific tasks or flash storage remaining capacity for required headroom.

#### Flash Memory Measurement

Flash memory on Arduino® boards can be measured with the help of the Arduino IDE. As stated before, Flash memory is where the application code is stored; the Arduino IDE reports Flash memory usage through its compiler output console to let developers know how much Flash memory resources are being used. the compiler's output changes depending on if the board is AVR-based or ARM-based.

#### SRAM Memory Measurement

 It is necessary to understand which code sector the memory demand is going beyond the available resources to solve this.

Reference to **[Arduino-MemoryFree](https://github.com/mpflaga/Arduino-MemoryFree)**.

#### EEPROM Memory Measurement

EEPROM memory management can be done easily using native libraries already installed into the Arduino IDE. The EEPROM library can be used to read, write and erase the EEPROM memory. The following code shows how a byte of information can be stored in the EEPROM memory and then read using the write and read functions:

#include <EEPROM.h>

void setup() {

}

void loop {

// Write data into an specific address of the EEPROM memory

EEPROM.write(address, value);

// Read data of an specific address of the EEPROM memory

EEPROM.read(address);

}

Also, it is possible to clear the entire EEPROM memory by setting it to 0, as shown in the code below:

#include <EEPROM.h>

void setup() {

}

void loop {

for (int i = 0 ; i < EEPROM.length() ; i++) {

// Clear EEPROM memory

EEPROM.write(i, 0);

}

### Optimizing Memory Usage in Arduino-based Systems

Some memory usage optimization techniques are：

#### Flash Memory Optimization

Flash memory optimization is the most likely straightforward optimization possible source. Flash memory is where the capacity used by compiled code can be significantly reduced by considering some details.

* Detach Unused Sources

Detaching new sources includes unused libraries and code residues. Code residues can be composed of no-longer-used functions and floating variables that take up the unnecessary space in memory. This will vastly improve the compiled code size and make a more clear compilation process.

* Modular Tasks

Modular tasks mean functions that wrap code that will be used repetitively or continuously by receiving different parameters. It is a great way to maintain clean code structure and performance while reducing the memory space required for additional tasks that might need to be implemented.

This leads to a compact code structure, which is much easier to understand when debugging is required and demands the developer consider computing complexity when designing the code structure or such a specific algorithm.

#### SRAM Memory Optimization

RAM shortages are usually the most common memory problems found; SRAM optimization can help in reducing this type of issue.

String Wrapper

Serial.print() or Serial.println() instructions uses SRAM space, which can be convenient but not desirable. The ideal way to use a Serial.print() or Serial.println() instruction is with the use of the F() String wrapper around the literals. For example: Serial.println(F("Something"));（"Something"是常量字符串）

Wrapping the String Something with the F() wrapper will move the Strings to Flash memory only rather than to use SRAM space also. Using the F() wrapper can be observed as offloading such data to Flash memory instead of SRAM. Flash memory is much more spacious than SRAM, so it is better to use Flash memory space than SRAM, SRAM will use heap section. This does not mean that memory space will always be available, as Flash memory does have limited space. It is not recommended to clog code with Serial.print() or Serial.println() instructions, but use them where they most matter inside the code.

PROGMEM(对编译器的指令)

Not only Strings occupy SRAM space, but global variables also take up quite a good amount of SRAM space. As global and static variables are streamed into SRAM space.

PROGMEM, which stands for Program Memory, can be used to store variable data into Flash memory space, just as the F() wrapper described before, but the use of PROGMEM presents one disadvantage: data read speed. Using RAM will provide a much faster data read speed, but PROGMEM, as it uses Flash memory, will be slower than RAM, given the same data size. Thus, it is essential to design code knowing which variables are crucial and which do not or have a lower priority.

The use of PROGMEM in an AVR-based Arduino® board is shown in the example code below:

#include <avr/pgmspace.h>

// Basic PROGMEM structure

const PROGMEM DataType Variable\_Name[] = {var0, var1, var2 ...};

// Storing an unsigned, 16-bit, integer

const PROGMEM uint16\_t NumSet[] = {0, 1, 1, 2, 3, 5, 8 ...};

// Storing a char in PROGMEM

const char greetMessage[] PROGMEM = {"Something"};

For ARM-based Arduino® board, to implement similar solution, we will need to use static const over the variables.

static const int Variable = Data;

The usage differs in different levels summarized as following:

* Namespace Level

At Namespace level, we are pointing at the variables and it is differed whether static is declared or not. If declared, it infers that the variable is explicit static; on the other hand, it is implicit static declaration.

* Function Level

If it is declared within static, any type of applicable data that is to be managed will be between function calls.

* Class Level

On a Class level, static declaration will mean any type of applicable data that is handled will be shared in between the instances.

#### **Non-Dynamic Memory Allocation**

Dynamic memory allocations cause heap fragmentation. With heap fragmentation, many areas of RAM affected by it cannot be reused again, leaving dead Bytes that can be taken as an advantage for other tasks. So to avoid heap or RAM fragmentation as much as possible, the following rules can be followed:

* Prioritize using the stack rather than the heap:

Stack memory is fragmentation-free and can be freed up thoroughly when the function returns. Heap, in contrast, may not free up the space even though it was instructed to do so. Using local variables will help to do this and try not to use dynamic memory allocation, composed of different calls: malloc, calloc, realloc

* Reduced global and static data (if possible):

Meantime the code is running, memory area occupied by these data will not be freed up. The data will not be modified as constant data takes up precious space.

* Use short Strings/literals:

It is good to keep Strings/literals as short as possible. A single char takes one Byte of RAM, so the shorter, the better memory space usage. This does not mean keeping it short and using it in several different code areas is possible. Use it when required and keep it as short as possible to spare RAM space for other tasks.

Arrays are also recommended to be at a minimum size. If it requires resizing the array, you can always re-set the array size in code. It may be a tedious, also non-efficient method to hard-code the array sizes. However, if the code utilizes small array sizes and less than three arrays, it may suffice via manual resizing, knowing the requirements. An intelligent way to do this is a resizable array with limited size. The tasks will use the array without going over the size boundary. Thus it is suitable for extensive code. Although, the limit of the array size must be analyzed and kept as small as possible.

#### **Reserve Function**

In tasks in code work with Strings that change in size depending on the operation outcome, reserve() is the way to go. This function will help reserve buffer space and pre-allocate for a String variable, changing its size and avoiding memory fragmentation. A String variable that changes in its size could result from an int type variable wrapped to be used as a String, for example.

// String\_Variable is an String type variable

// Alloc\_Size is the memory to be pre-allocated in number of Bytes with unsigned int type

String\_Variable.reserve(Alloc\_Size);

#### **Buffer Size Control**

Backend processes also require a memory pool for their processing purpose. It is something on which the system will work according to the size of the memory pool defined. This buffer size can be user-defined, which can be reduced to allocate a lower memory size.

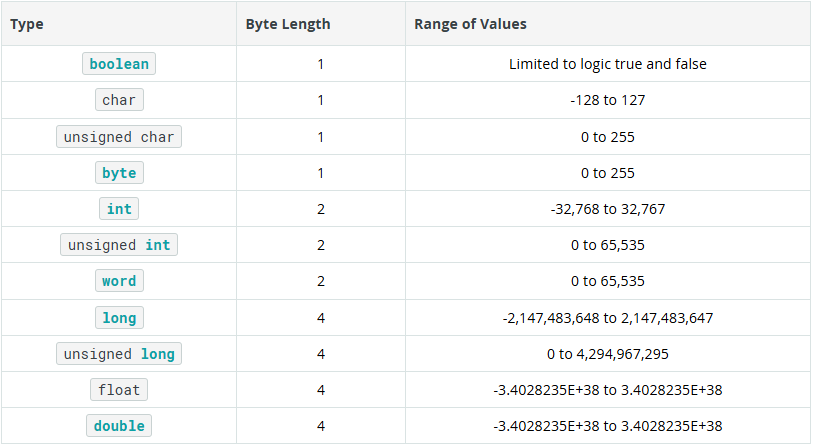
Serial communications is a regularly used service in Arduino-based systems; Serial communications in Arduino work using the preinstalled Serial library (external libraries can also emulate serial communications using software). If high-speed serial communication is not part of the requirements, the serial buffer size can be redefined to save some memory consumption. This can be made easily by modifying the following code line in the HardwareSerial.h file that can be found in the installation folder of the Arduino IDE:

#define SERIAL\_TX\_BUFFER\_SIZE 64

#define SERIAL\_RX\_BUFFER\_SIZE 64

#### **Corrective Data Type Usage**

The following table shows basic value data types in Arduino:



### **EEPROM Memory Optimization**

With EEPROM, it is crucial to know that write operation is limited. The read operation is unlimited for EEPROM; however, the write operation is finite and usually capped at 100,000 cycles. Thus, it is essential to save only essential parameters for sensors or modules to work with primarily unchanging data. Additionally, avoid implementing write operations into loops to avoid constant write operations, these operations should be minimized while the system is working.

#### **EEPROM Emulation with Flash Memory**

As EEPROM is limited with the write operation cycle, it also applies to Flash memory. Both of them are subjected to data retention loss after the manufacturer's defined life cycle. EEPROM is based on NOR-type（异或） memory, while the Flash memory is NAND type, making the EEPROM more costly（expensive） than Flash memory. EEPROM works by accessing the data byte-wise, whereas Flash memory accesses block by block.

Sometimes the developer would have to use the EEPROM as alternative storage for task operations, but we know it will be impractical coding due to its size and behavior properties. It is possible to use Flash memory to emulate the EEPROM to solve this. Thanks to the **[FlashStorage](https://github.com/cmaglie/FlashStorage)** library created by Chrisitan Maglie, it is possible to emulate the EEPROM by using Flash memory.

The FlashStorage library will help you to use the Flash memory to emulate the EEPROM, but of course, please remember the EEPROM's properties when using the library. As for EEPROM, the Flash memory is also limited in the write cycles. With two new additional functions stated in the library, EEPROM.commit() should not be called inside a loop function; otherwise, it will wipe out the Flash memory's write operation cycles, thus losing data retention ability.

### **Further Reading and Resources**

8-bit AVR® Core documentation in the [Microchip® Developer help site](https://microchipdeveloper.com/8avr:avrcore). Here you can find detailed information of the 8-bit AVR® Central Processing Unit (CPU).

# Terminology

Program written with Arduino are called sketches.

Anatomy /əˈnætəmi/ of Arduino board，anatomy，结构，解剖。

we can create a sequence by sending a high or low state rapidly a number of times. This is known as a binary sequence or a bitstream.

A sensor, in simple terms, is used to sense its environment, meaning it records a physical parameter, for example temperature, and converts it into an electronic signal.

Digital sensors are a bit more advanced, depending on the type. They rely on [Serial Communication Protocols](https://docs.arduino.cc/learn/starting-guide/getting-started-arduino/" \l "serial-communication-protocols) to send the data accordingly, and requires a bit more effort to translate the data. As mentioned in the [Electronic Signals](https://docs.arduino.cc/learn/starting-guide/getting-started-arduino/" \l "electronic-signals) section above, data is sent using a binary sequence and this needs to be addressed and configured on a software level. Luckily, a lot of sensors are accompanied by software libraries, which makes it a lot easier to read.

An actuator, in simple terms, is used to actuate or change a physical state. Actuators converts electric signals into e.g. radiant energy (light) or mechanical energy (movement).

Sensors and actuators, are typically referred to as inputs and outputs.

There are several serial communication protocols that uses the aforementioned digital signals to send data. The most common are UART, SPI & I²C.

Universal Asynchronous Receiver/Transmitter, 异步全双工串行通信协议。

Serial Peripheral Interface，即串行外设接口，是一种高速、全双工、同步的通信总线。一主多从。

Inter - Integrated Circuit，集成电路间接口。串行同步通信协议，由飞利浦公司开发。多主多从。一般有标准模式（100 kbit/s）、快速模式（400 kbit/s）和高速模式（3.4 Mbit/s）等，适合对传输速率要求不是特别高的中、低速应用场景。

Internet of Things (IoT). Most modern Arduino boards now come equipped with a radio module, designed to communicate wirelessly. There are several different ones: Wi-Fi, Bluetooth®, LoRa®, GSM, NB-IoT and more. Each are designed to communicate using the various technologies available on the market.