Microcontroller

Lecture-3

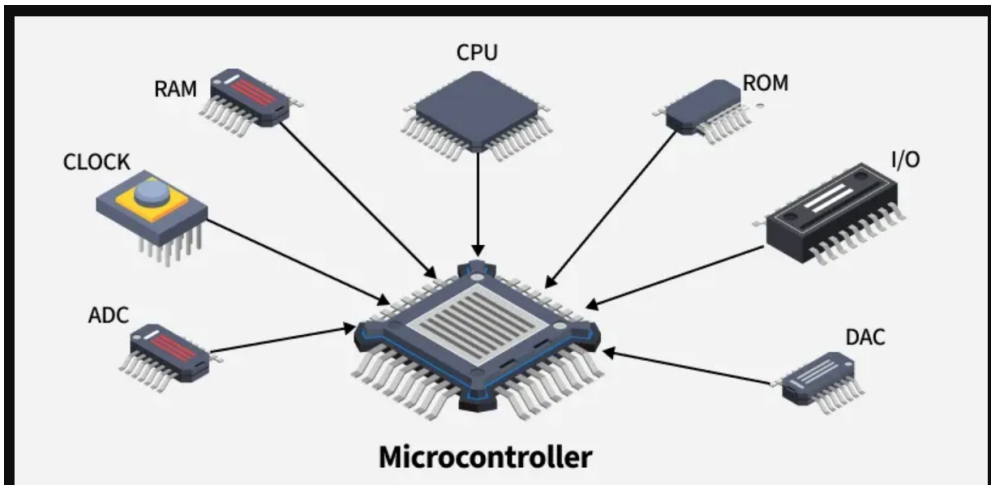
Dr. Naushin Nower

Contents

- Microcontroller
- Types of microcontroller
- RISC and CISC based microcontroller
- Ardunio

Microcontroller (MCU)

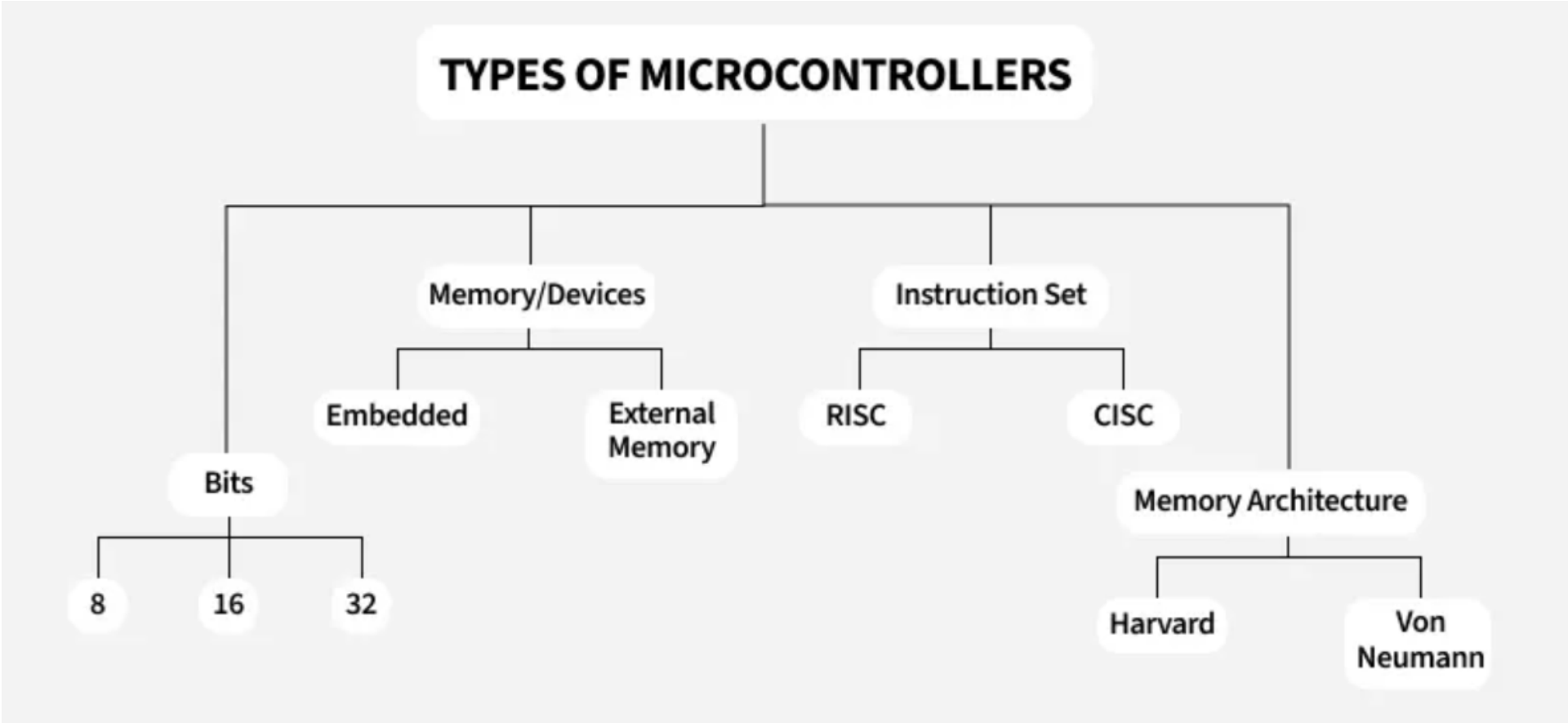
- is a small computer on a single integrated circuit that is designed to control specific tasks within electronic systems.
- It combines the functions of a central processing unit (CPU), memory, and input/output interfaces, all on a single chip.
- A typical microcontroller consists of a processor core, volatile and non-volatile memory, input/output peripherals, and various communication interfaces.



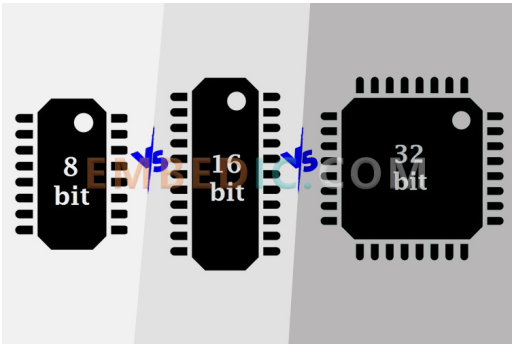
Microcontroller vs. microprocessor

- A microprocessor is a single integrated chip that contains a device's CPU. However, it doesn't contain any RAM or ROM memory, or any other peripherals a device may have. The chip instead relies on inputs/outputs (I/Os) to connect to memory and peripherals.
- On the other hand, a microcontroller has the CPU, RAM and ROM, as well as peripherals all embedded onto a single chip, effectively making it a computer itself.

Types of Microcontroller



- **8-bit Microcontrollers**
  - The internal bus is 8-bits wide.
  - The ALU (Arithmetic Logic Unit) performs operations on 8 bits at a time.
  - Intel 8031/8051, PIC1x, Motorola MC68HC11.



- **16-bit Microcontrollers**
  - The internal bus is 16-bits wide, providing better precision and performance than 8-bit.
  - A 16-bit microcontroller can handle a wider range of numbers (0x0000 to 0xFFFF, or 0 to 65535) compared to the 8-bit range (0x00 to 0xFF, or 0 to 255).
  - Example microcontrollers: Extended 8051XA, PIC2x, Intel 8096, Motorola MC68HC12.
- **32-bit Microcontrollers**
  - These microcontrollers use 32-bit instructions for operations.
  - Used in advanced applications like medical devices, engine control systems, office machines, and other embedded systems.
  - Example microcontrollers: Intel/Atmel 251 family

Classification of Microcontrollers by Memory Type

- **Embedded Memory Microcontroller**
  - All functional blocks are built into the chip.
  - Includes program memory, data memory, I/O ports, serial communication, counters, timers, and interrupts.
  - Example: 8051 microcontroller (has everything on the chip).
- **External Memory Microcontroller**
  - The microcontroller does not have all functional blocks on the chip.
  - Some components, like program memory, need to be connected externally.
  - Example: 8031 microcontroller (does not have program memory on the chip).

Classification of Microcontrollers by Memory Architecture

- **Harvard Memory Architecture Microcontroller**
  - Has separate memory for program and data.
  - Allows simultaneous access to both program and data memory, improving efficiency.
  - Common in microcontrollers that require faster data processing.
- **Von Neumann Memory Architecture Microcontroller**
  - Uses shared memory for both program and data.
  - Simplifies design but can be slower since the program and data share the same bus.

Architecture

- The majority of MCUs use one of the following architectures:
- ARM, MIPS, or X86, RISC-V etc
- Different instruction set architectures (ISAs). The ISA of microcontroller architecture dictates the format of instructions managed by the CPU.
  - ❖ ARM, MIPS, and RISC-V all have reduced instruction set computer (RISC) ISAs,
  - ❖ while X86 uses complex instruction set computers (CISC), which supports more complex and flexible instructions.
- Different processing power. Overall, X86 has more processing power and high performance. In contrast, ARM, RISC-V, and MIPS don't require as much processing power due to simpler, set-length instructions dictated by RISC architecture, lending themselves to applications that don't require high performance.
- Different energy consumption. X86 uses more energy to function and support higher processing power. MIPS uses less power and ARM has even greater efficiency.

Instruction set architectures (ISAs)

- Informally ISA as a “language” that is used to communicate with the CPU
- RISC and CISC are popular processor architectures that utilize different data processing instruction sets to perform basic logical and input/output operations
- As per this equation, CPU performance depends on the number of instructions in a computer program. This implies that the more instructions, the more time is needed to execute them. It can further be simplified into the number of clock cycles per instruction and time per cycle.
- Thus, the CPU performance can be optimized in either of two ways:
  - Reduce the number of instructions per program, or
  - Minimize cycles per instruction.

$$\text{CPU time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$$

What Is RISC?

- RISC stands for 'Reduced Instruction Set Computer.'
- These were introduced in the 1980s to overcome the complexities of CISC processors.
- RISC processors work with more instructions; however, the number of cycles an instruction may take to execute is minimized.
- In general terms, a RISC machine takes one CPU cycle to complete one instruction.
- Let's consider the same ADD instruction and look at how RISC devices accomplish it.

Load X, 1600
Load Y, 1601
ADD X, Y
Store 1600, X

RISC processors use simpler instruction sets, complex and high-level instructions need to be divided into multiple, simple instructions.

ARM (Advanced RISC Machine) is a well-known example of the RISC framework. Its processors are observed in desktops, laptops, smartphones, gaming consoles, and several other smart IoT devices that are battery-operated where energy efficiency is essential.

CISC

- Complex Instruction Set Computer.
- This architecture was introduced in the 1970s by Intel Corporation when the earliest computers focused on enhancing CPU speed by minimizing the number of instructions per program (as per equation 1).
- This objective was achieved by combining multiple simple commands into one complex instruction.
- Let's understand this with a simple CISC ADD instruction, which requires two inputs:
  - Memory locations of two numbers essential for an addition
  - Perform addition and store the result in the first memory location

ADD 1800, 1801

The ADD instruction picks up numbers from memory locations 1800 and 1801 or registers. Later, the picked-up numbers are added and eventually stored in location 1800.

AMD, Intel x86, VAX, and System/360.

https://www.spiceworks.com/tech/tech-general/articles/risc-vs-cisc/

CISC vs RISC

CISC	RISC
Emphasis on hardware	Emphasis on software
Includes multi-clock complex instructions	Single-clock, reduced instruction only
Memory-to-memory: "LOAD" and "STORE" incorporated in instructions	Register to register: "LOAD" and "STORE" are independent instructions
Small code sizes, high cycles per second	Low cycles per second, large code sizes
Transistors used for storing complex instructions	Spends more transistors on memory registers

What is Arduino?

Note:
Most Arduino boards use the Atmel 8-bit AVR microcontroller.

- At the heart of the Arduino is the microcontroller. A microcontroller is a standalone, single chip integrated circuit that contains a CPU, read-only memory, random access memory and various I/O busses. The Arduino UNO R3 which we will be using uses the ATmega328 chip.
- Specs are:

Flash Memory	32 KB (ATmega328P) of which 0.5 KB used by bootloader
SRAM	2 KB (ATmega328P)
EEPROM	1 KB (ATmega328P)
Clock Speed	16 MHz

EEPROM:
Electrically Erasable Programmable Read-Only Memory
Non-volatile memory.
Can be erased and reprogrammed using a pulsed voltage

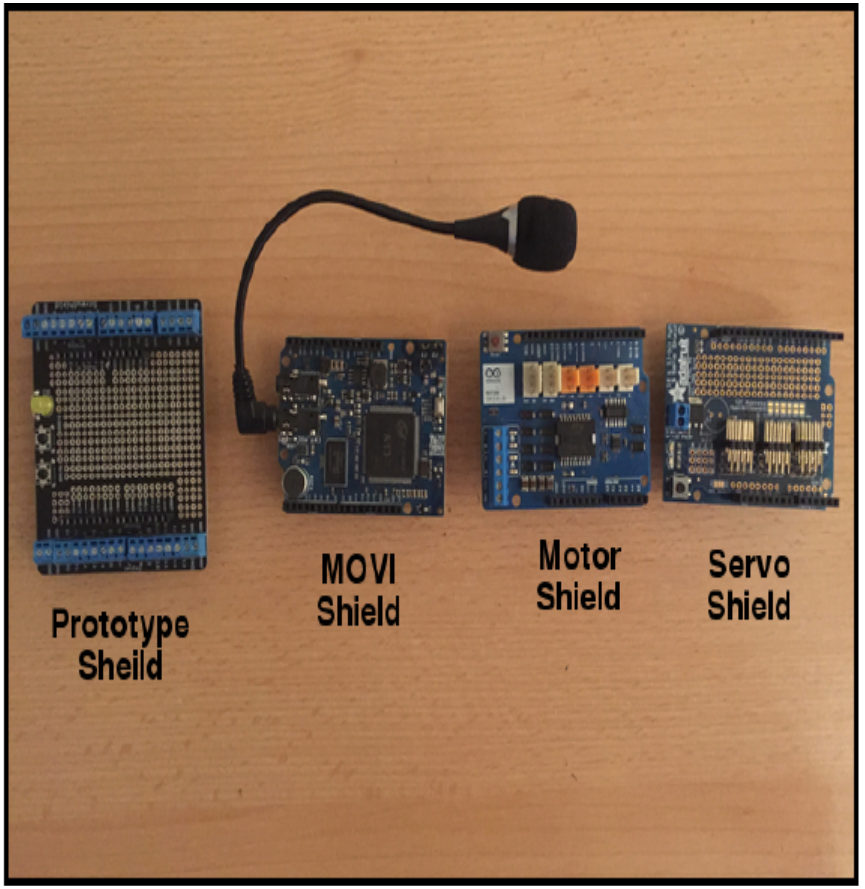
- However do not be too concerned if you do not understa all those specifications because we will be interacting with the microcontroller using the interface that the Arduino board provides us.

What is Arduino?

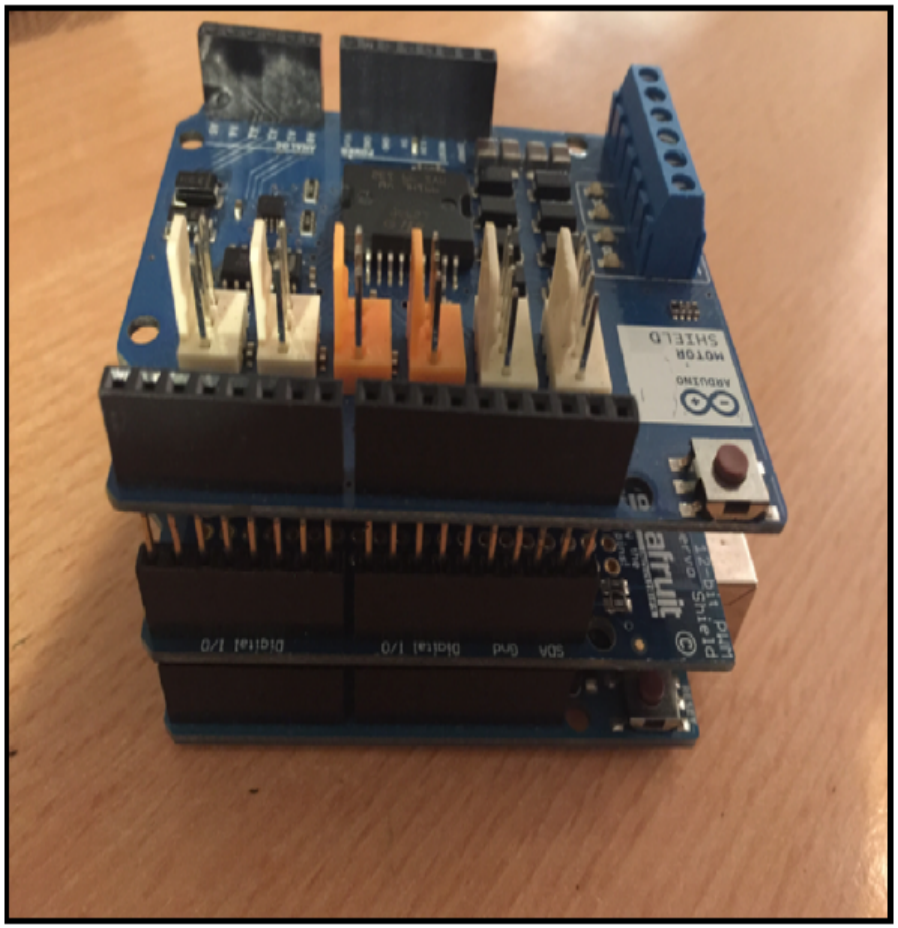
- The Arduino is an open source hardware and software platform that is incredibly powerful yet easy to use.
- You can look at and download the code from any of the Arduino repositories on GitHub here:
  - https://github.com/arduino
- Projects can range from simply making an LED blink or recording the temperature to controlling 3D printers or making robots.
- While there are numerous models of the Arduino, in this course we will primarily be using the very popular Arduino UNO board.

Arduino shields

- An Arduino shield is a modular circuit board that plugs directly into the pin headers of the Arduino board.
- These shields will add extra functionality to the Arduino board.
- If we are looking to connect to the internet, do speech recognition, control DC motors or add other functionality to the Arduino, there is probably a shield that can help us.
- While you don't have to use shields, they do make adding extra functionality to our Arduino boards very easy.



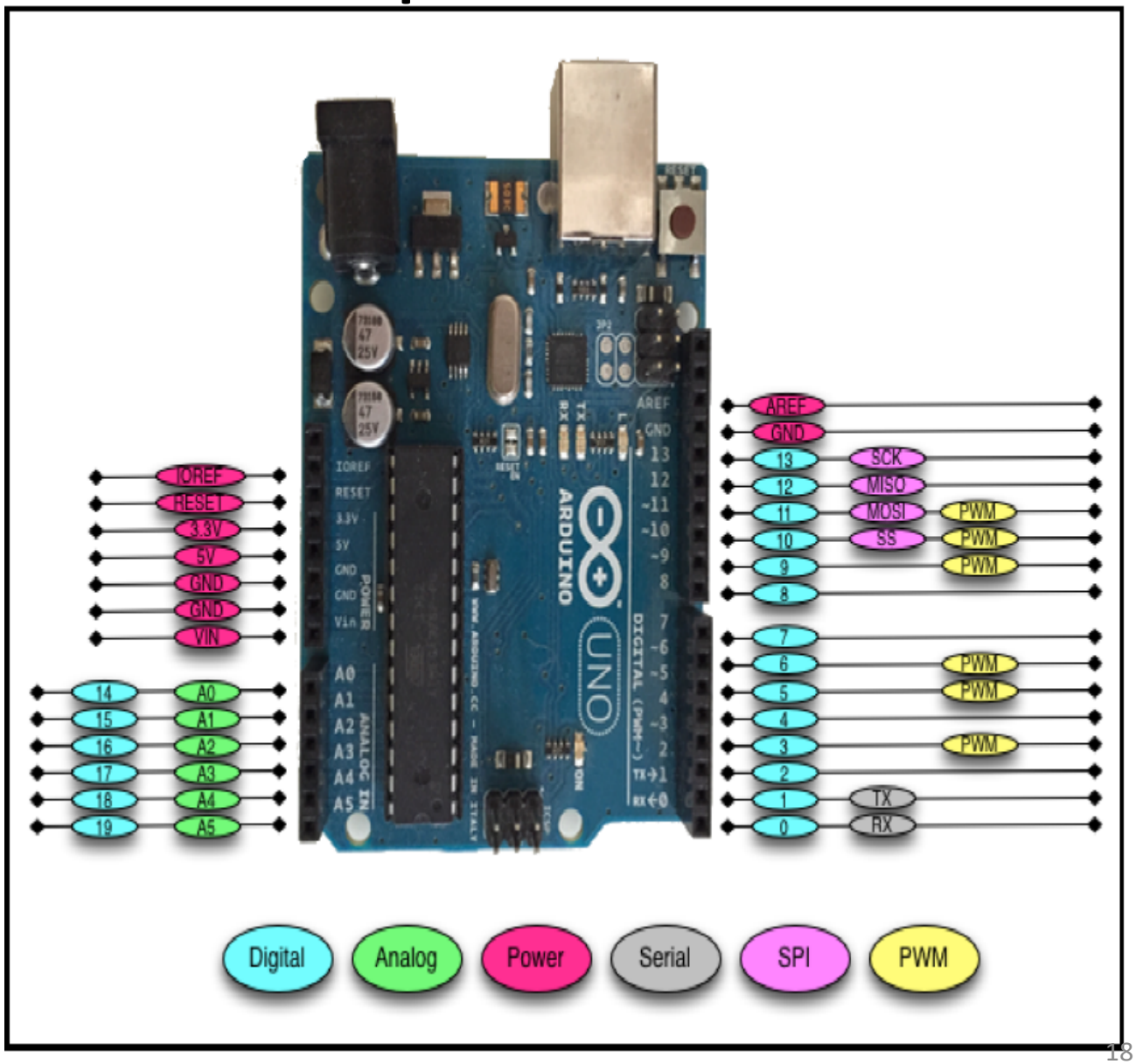
Arduino looks with two shields attached:



- A shield fits on top of the Arduino by plugging directly into the pin headers.
- We can also stack one shield on top of another if they do not use the same resources. Here is how an

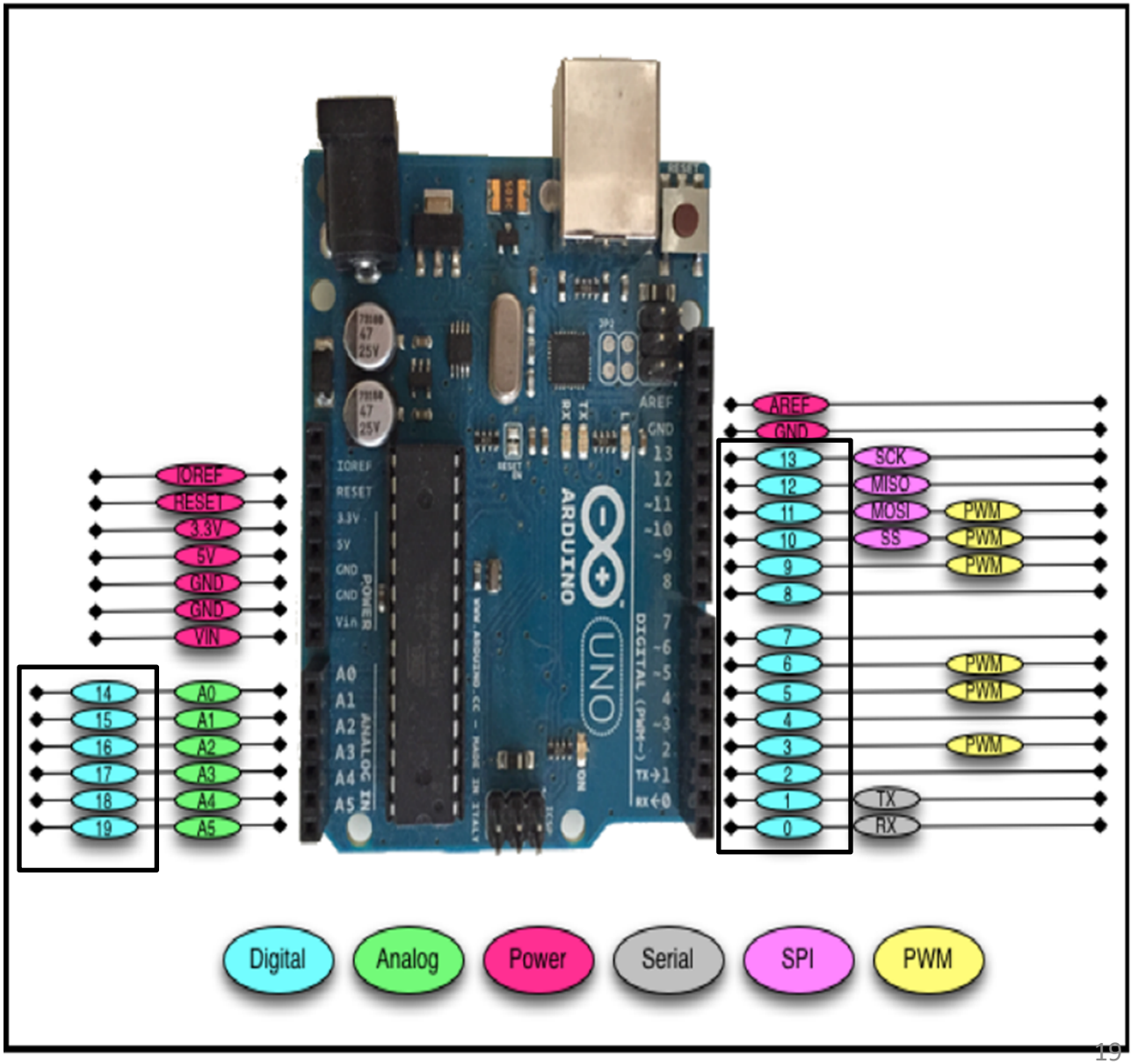
Arduino pin

- There is a total of 31 pins in the Arduino Uno pin headers.
- Most of these pins can be configured to perform different functions.



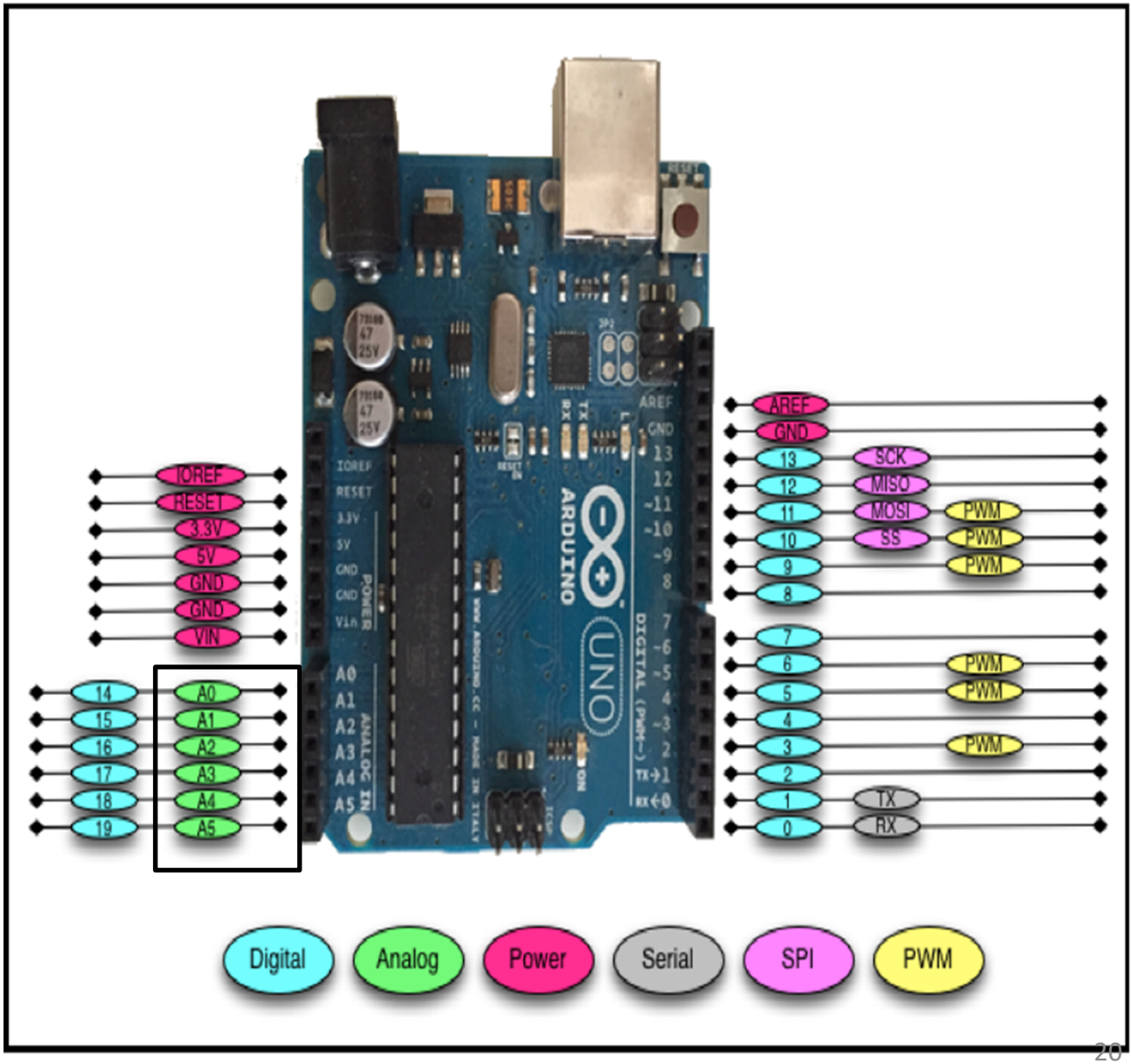
Digital pins

- Used to connect external sensors.
- These pins can be configured for either input or output.
- These pins default to an input state
- The digital pins will have one of two values: HIGH (1), which is 5V, or LOW (0), which is 0V.



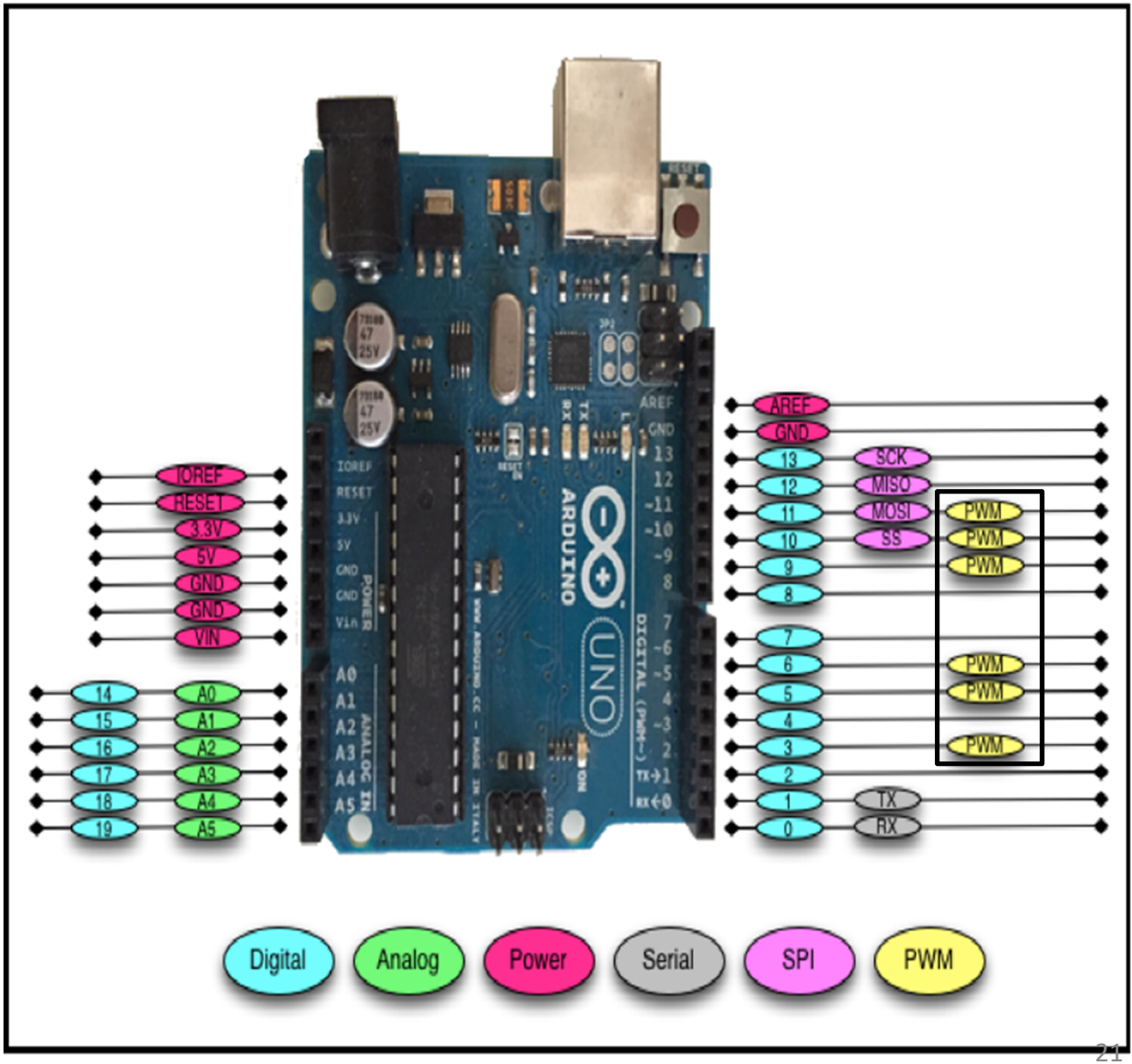
Analog input pins

- The Arduino Uno contains a built-in **Analog-To-Digital** (**ADC**) converter with six channels, which gives us six analog input pins. The ADC converts an analog signal into a digital value.
- While the digital pins have two values, either high or low, the analog input pins have values from **0 to 1023** relative to the reference value of the Arduino.
- The Arduino Uno has a reference value of 5V.
- Used to read analog sensors such as rangefinders and temperature sensors.
- The six analog pins can also be configured as digital pins if we run out of digital pins in our project.



PWM pins

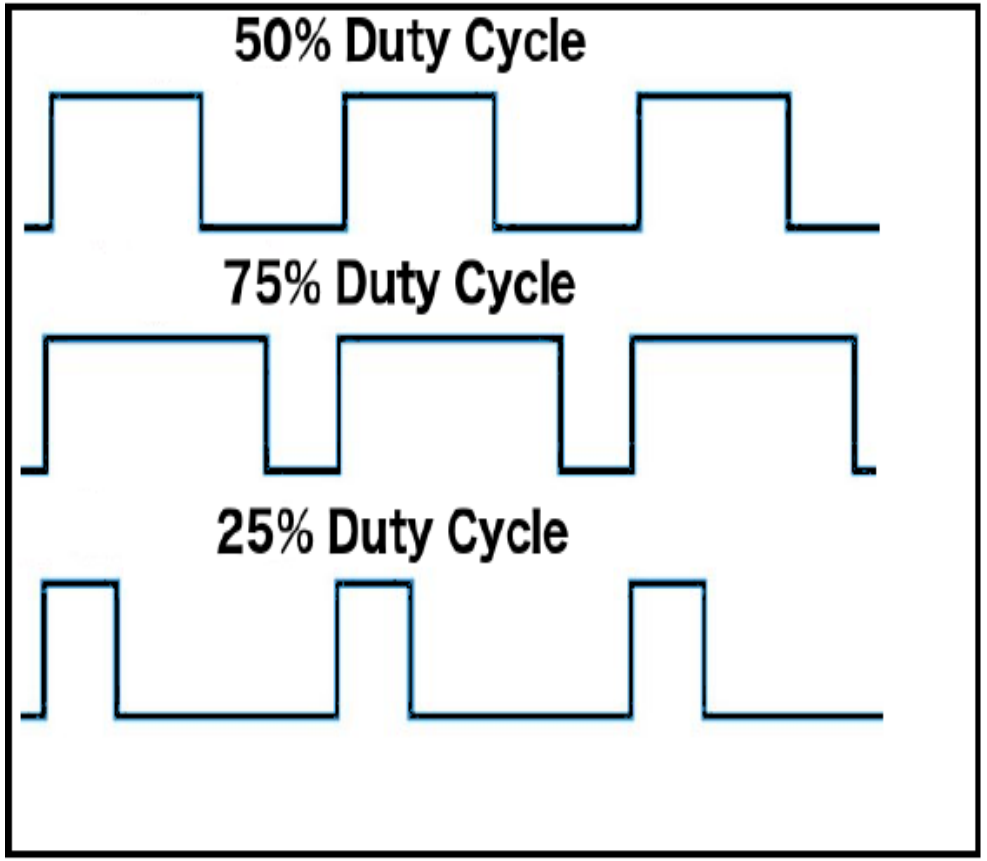
- Where the analog input pins are designed to read analog sensors (input), the PWM pins are **designed for output**. PWM is a technique for obtaining **analog results with digital output**.
- Since a digital output can be either on or off, to obtain the analog output the digital output is switch between HIGH and LOW rapidly.
- The percentage of the time that the signal is high is called the **duty cycle**.



Duty cycle

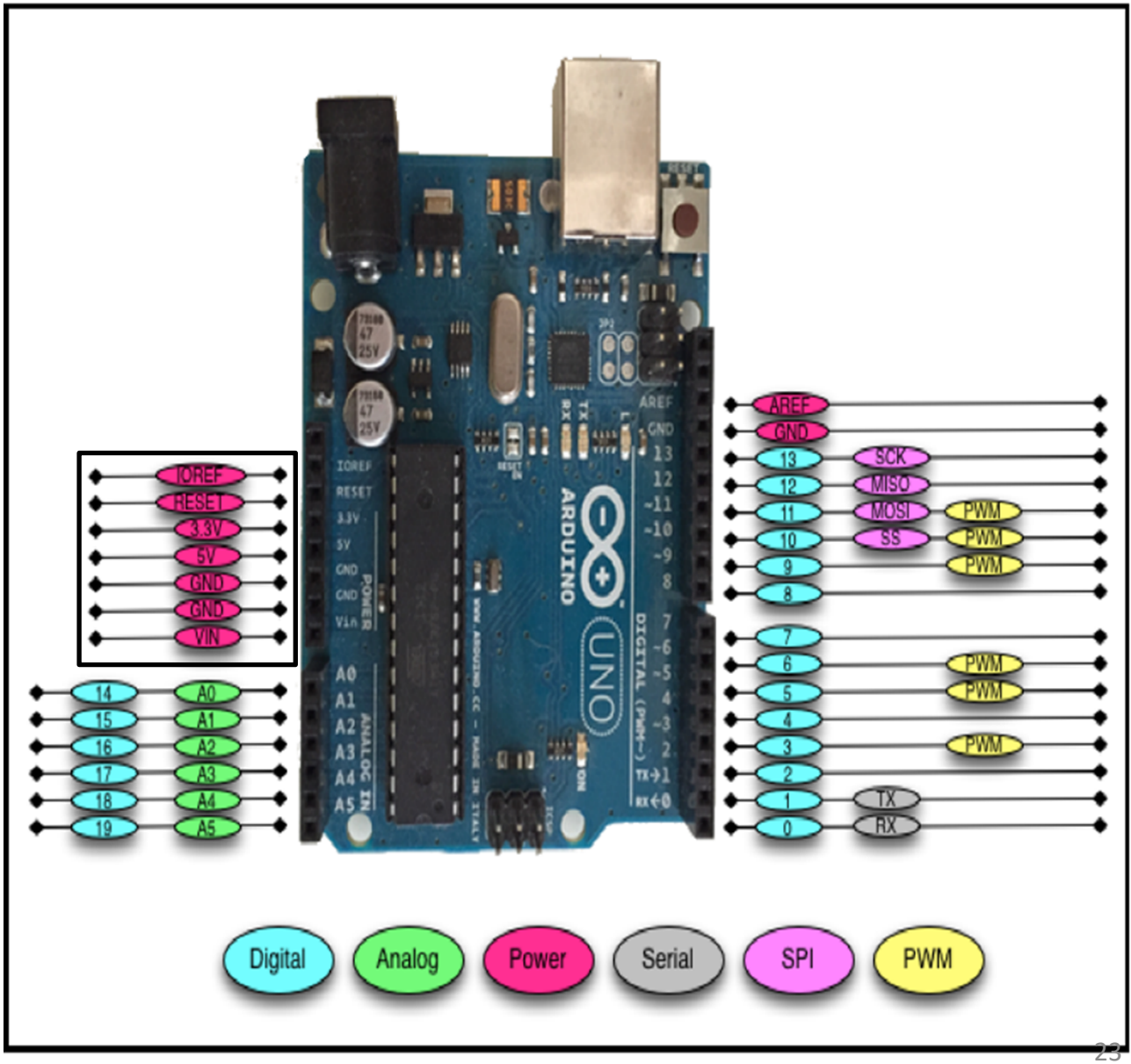
- We have the ability to set the frequency of how fast the signal can switch between HIGH and LOW.
- This frequency is measured in **Hertz** and sets how many times the signal can switch per second.
- For example, if we set the frequency to **500 Hz**, that would mean that the signal could switch **500 times a second**.

- This will be come clearer as we use the pins.



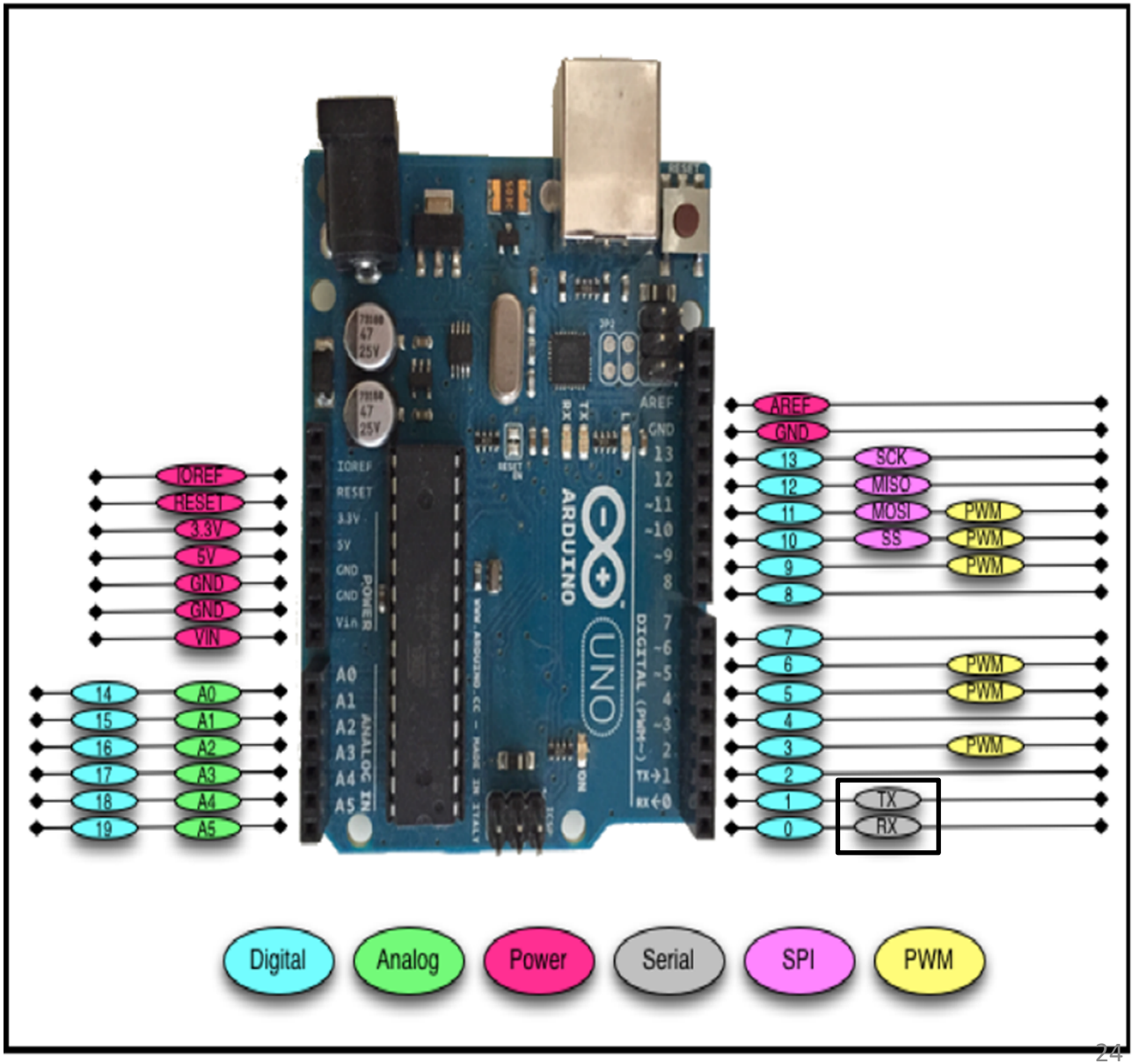
Power pins

- **VIN**: This pin is used when we power the Arduino board using an external power supply.
- **GND**: These are the ground pins.
- **5V**: This is 5V out and is used to power most sensors.
- **3.3V**: This is 3.3V out and can be used to power sensors that are compatible with 3.3V.
- **Reset**: This pin can be used to reset the Arduino board by an external source.
- **ioref**: This is the reference voltage for the board. For the Arduino, this will be 5V.



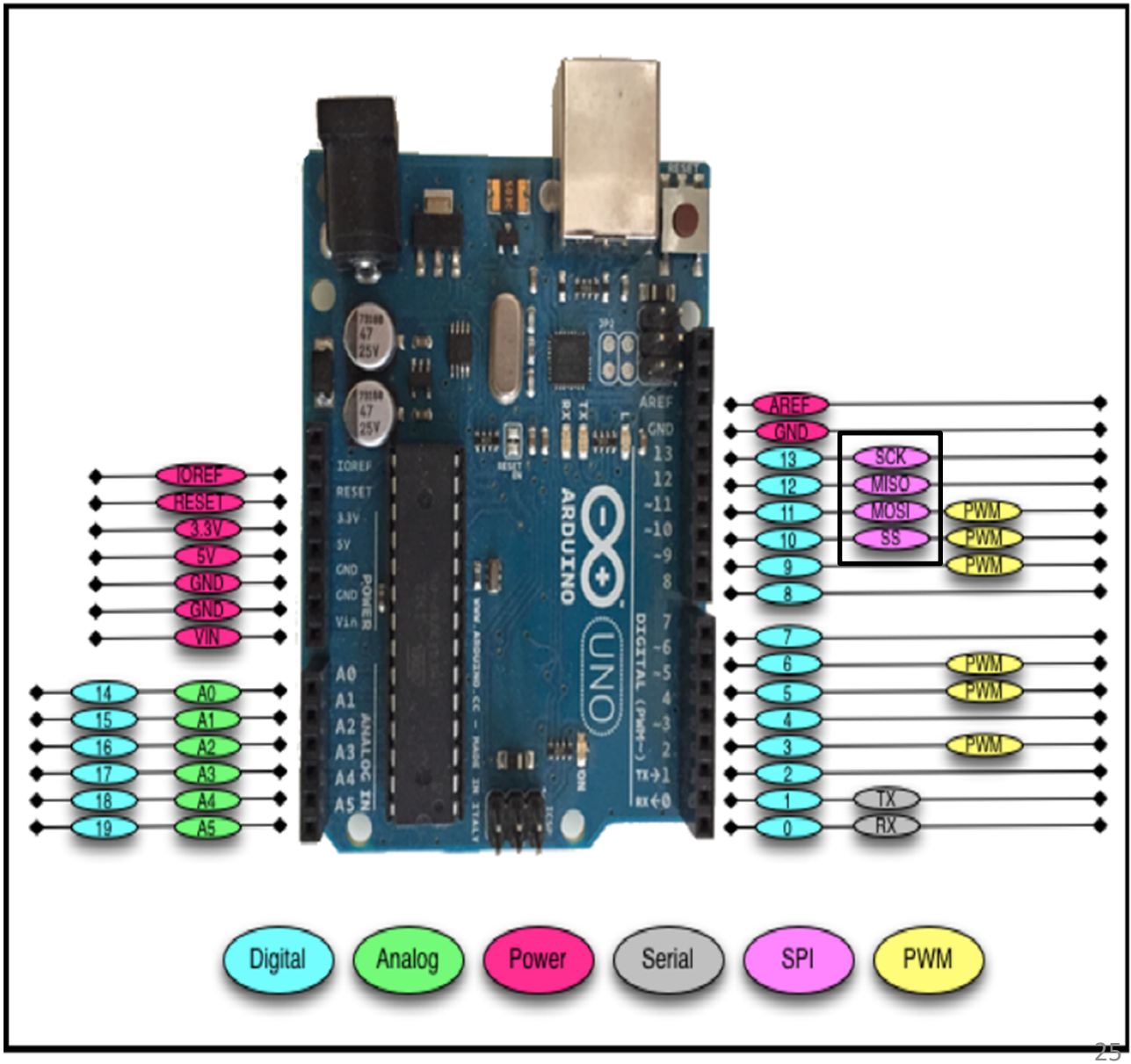
Serial pins

- Used for serial communication.
- The **RX (digital pin 0)** is used to receive.
- **TX (digital pin 1)** is used to transmit.
- Serial communications work on binary (1's and 0's).
- Provided for legacy reasons primarily.



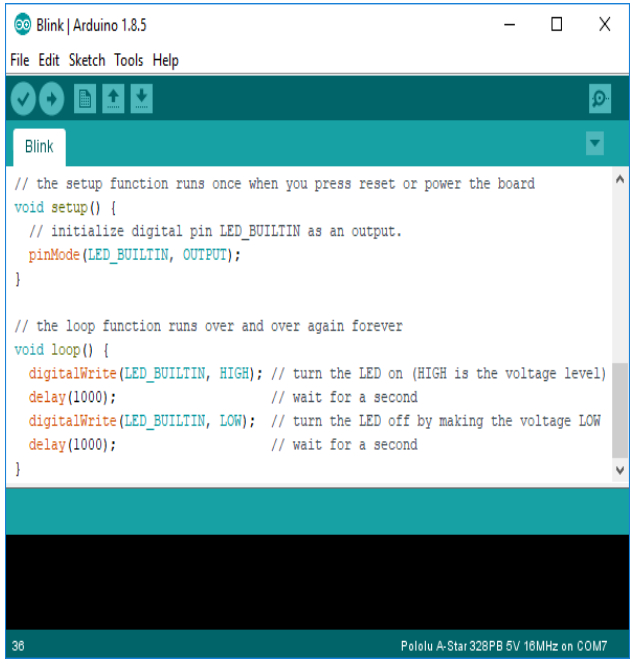
SPI pins

- The **Serial Peripheral Interface** (**SPI**) pins are used for a synchronous serial data protocol that is used by microcontrollers for communicating with peripheral devices.
- This protocol always has one master with one or more slave devices.
- **MISO**: The **Master in Slave out** pin is used to send data from the slave to the master device.
- **MOSI**: The **Master out Slave in** the pin is used to send data from the master to the slave device.
- **SCK**: The **serial clock** synchronizes the data transmission and is generated by the master.
- **SS**: The **slave select** pin tells the slave to go active or to go to sleep. This is used to select which slave device should receive the transmission from the master.



Installing the IDE

- The first step in programming an Arduino board install the Arduino IDE (integrated development environment).
- **Linux/Raspberry PI**
- This program checks code and loads it onto the Arduino. Install the latest version of Arduino IDE using apt:
  - sudo apt-get update && sudo apt-get upgrade
  - sudo apt-get install Arduino
- **Windows**
- Download from: https://www.arduino.cc/en/Main/Software

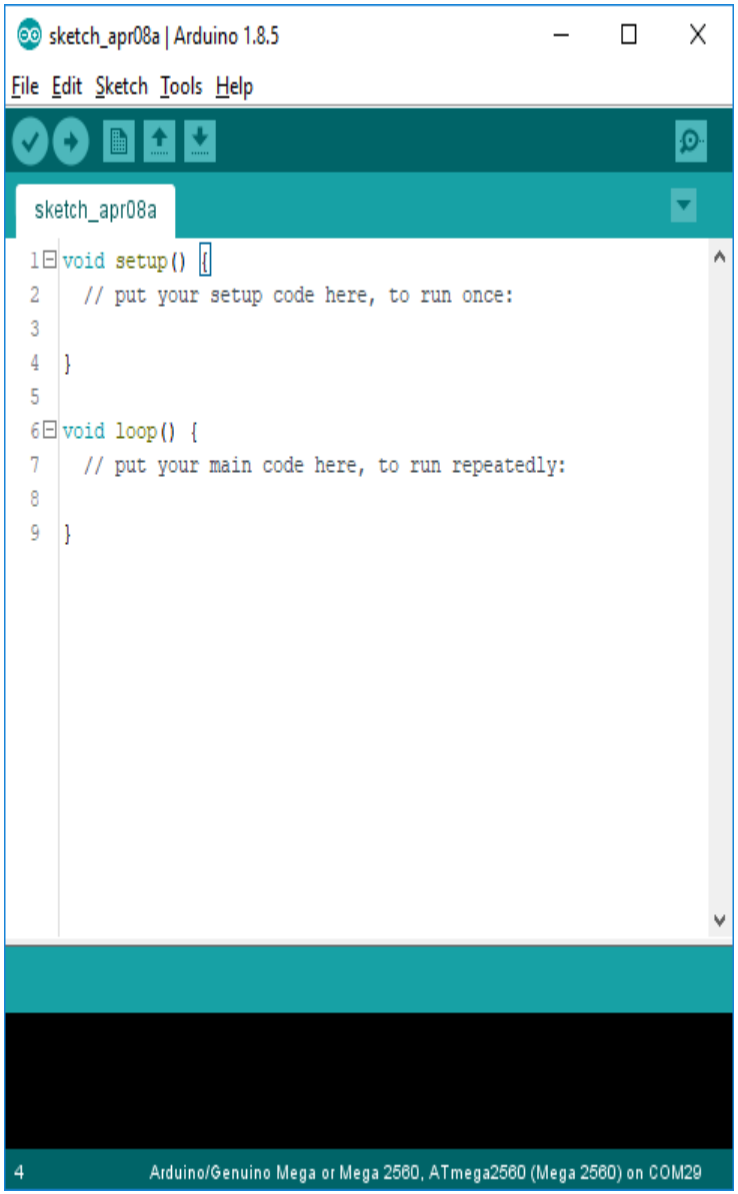


setup() and *loop()* functions

- *setup()*: This function is called once when a sketch starts after power-up or reset. It is used to initialize variables, input and output pin modes, and other libraries needed in the sketch

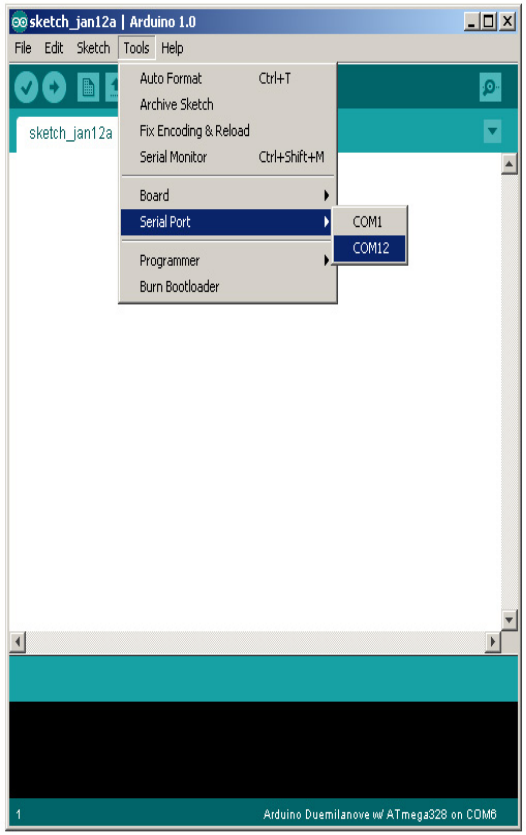
- *loop()*: After *setup()* has been called, function *loop()* is executed repeatedly in the main program. It controls the board until the board is powered off or is reset.

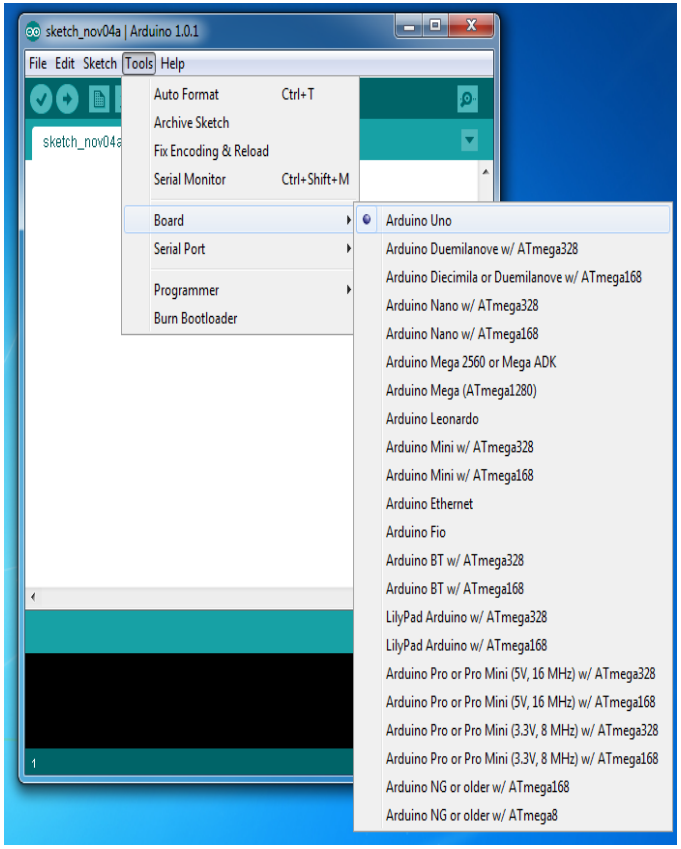
Arduino IDE



See: http://arduino.cc/en/Guide/Environment for more information

ALWAYS: Select Serial Port and Board





Programming in for Arduino

- The Arduino programming language is based on a very simple hardware programming language called processing, which is **similar to the C language**.
- You create sketches which contain your code
- After the sketch is written in the Arduino IDE, it should be uploaded on the Arduino board for execution.

Basic code structure

```
void setup(
)
{
statements
;
}

void loop( )
{
statement;
}
```

The setup function is the first to execute when the program is executed, and this function is called only once.

Used to initialize the pin modes and start serial communication. **This function has to be included even if there are no statements to execute.**

The execution block runs after setup and hosts statements like reading inputs, triggering outputs, checking conditions etc..

Example program

LED fade-in and fade-out

```
int led = 9; // The digital pin to which the LED is connected
int brightness = 0; // Brightness of LED is initially set to 0
int fade = 5; // By how many points the LED should fade

void setup()
{
pinMode(led, OUTPUT); //pin 10 is set as output pin
}

void loop() // The loop function runs again and again
{
analogWrite(led, brightness); // set the brightness of LED


brightness = brightness + fade; //Increase the brightness of LED by 5 points

if (brightness <= 0 || brightness >= 255) // check the level of brightness

{

fade = -fade;

}
delay(30); // Wait for 30 milliseconds
}
```