Ch6: Microprocessors and Microcontrollers – Part 2

Contents:

Arduino Boards

C/C++ Language Overview

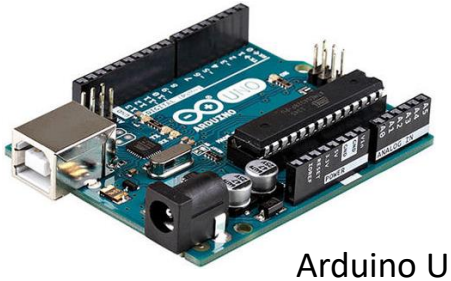
Arduino Programming

Digital I/O

Arduino Boards

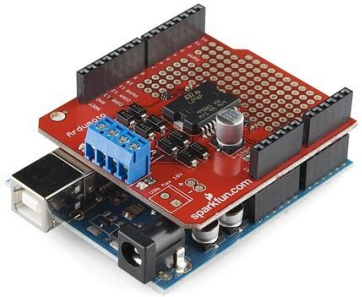
What Is an Arduino?

The **Arduino** is a **microcontroller development platform** paired with an intuitive programming language that you develop using the Arduino integrated development environment (IDE).



Arduino UNO

By equipping the Arduino with sensors, actuators, lights, speakers, add-on modules (called **shields**), and other integrated circuits, you can turn the Arduino into a programmable "**brain**" for just about any control system.



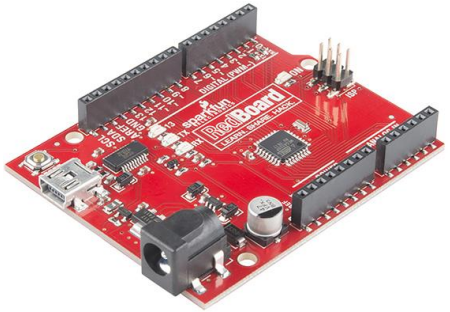


Arduino Is an Open-Source Hardware

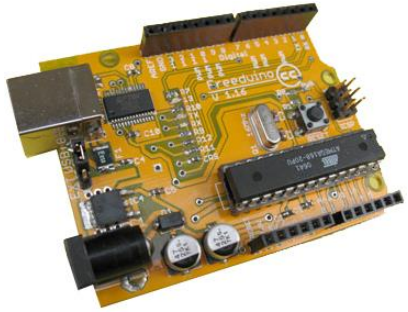
Because the Arduino is **open-source hardware**, all the design files, schematics, and source code are freely available to everybody.

You can integrate the Arduino platform into your designs, make and sell Arduino **clones**, and use the Arduino software libraries in other projects.

Three Examples of **Arduino Clones**:



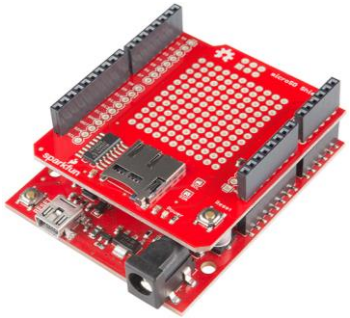
SparkFun RedBoard



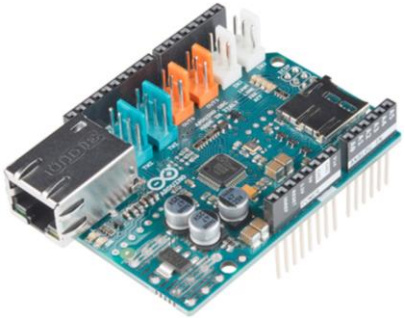
Freeduino

Seeduino

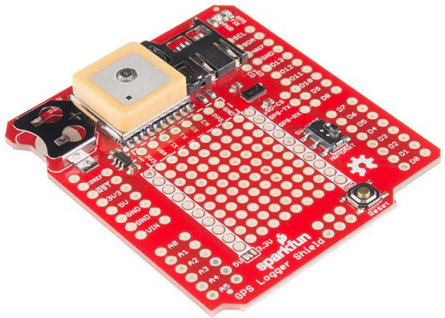
Some Arduino Shields



microSD Shield



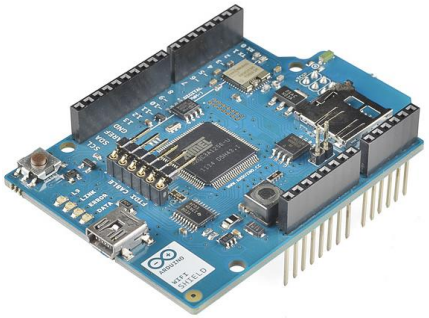
Ethernet Shield



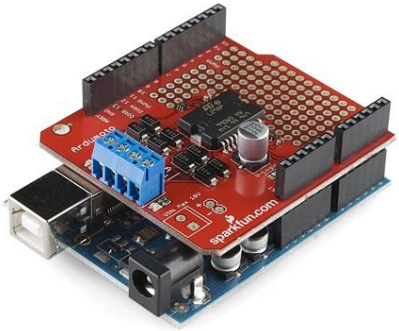
GPS Logger Shield

Color LCD Shield

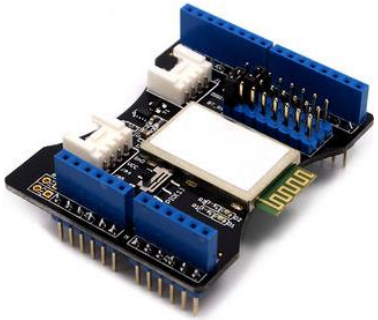
Cellular Shield



Wi-Fi Shield



Motor Driver Shield

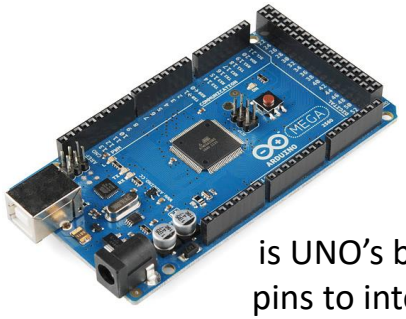


Bluetooth Shield

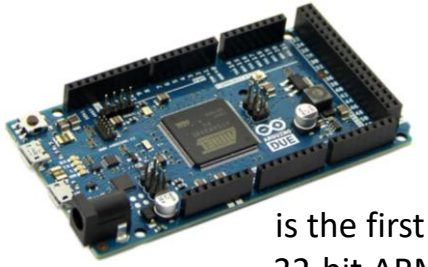
Arduino Boards

There are many different official Arduino boards, each with different capabilities (Applications, number of inputs and outputs, speed, operating voltage, form factor, etc.).

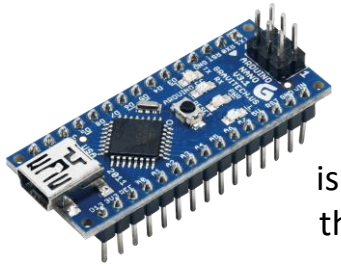
Arduino Uno
is the best board to get started with electronics and Arduino Programming.



Arduino Mega 2560
is UNO's big brother with many general I/O pins to interface with many more devices. It is designed for more complex projects (e.g., 3D printers and robotics projects).



Arduino Due
is the first Arduino board based on a 32-bit ARM core microcontroller and it is the perfect board for powerful larger scale Arduino projects.

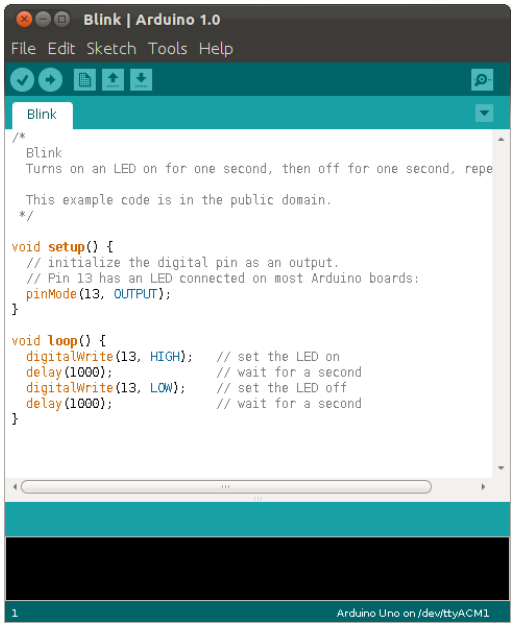


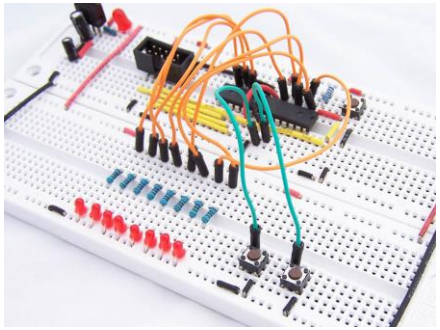
Arduino Nano
is a compact board similar to the UNO and designed to be mounted right into a breadboard socket.

A full range of official Arduino products: *https://www.arduino.cc/en/hardware*

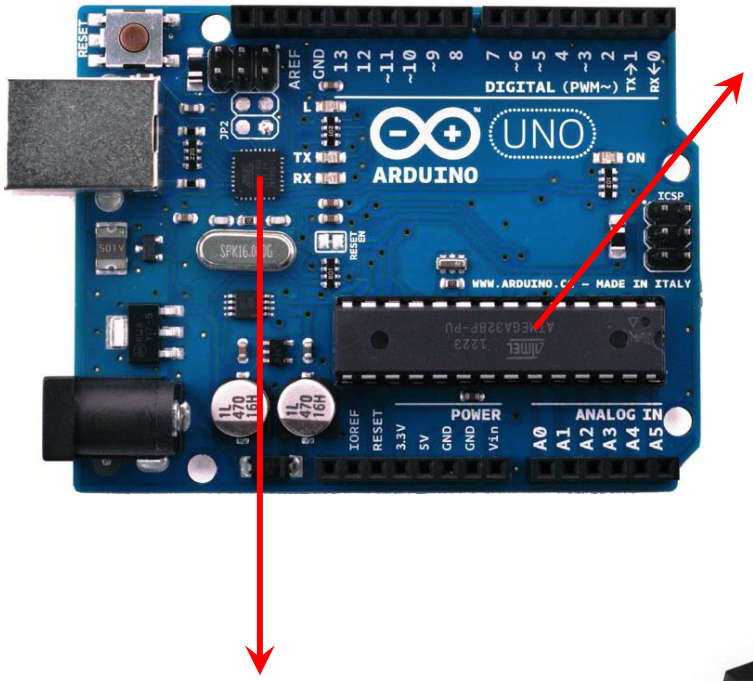
Arduino-Based Projects

Three main components of Arduino based projects:
- **Arduino IDE** (Integrated Development Environment)
- One of the Arduino boards
- External hardware (including shields and hand-made circuits)



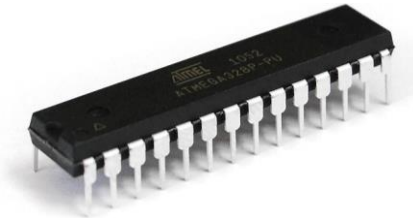


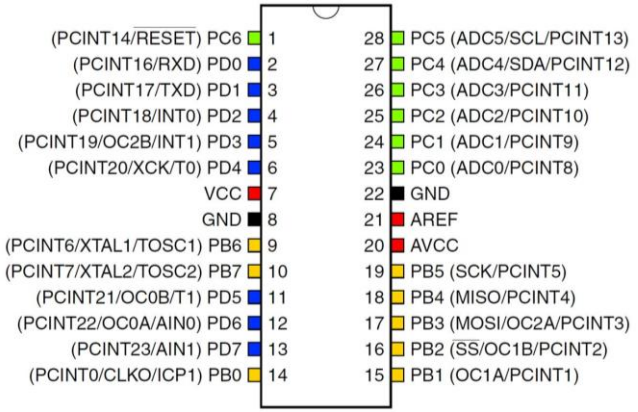
Arduino Uno Components



ATmega328P Microcontroller

Arduino Uno uses an AVR ATMega328P microcontroller. It has 32 KB (with 0.5 KB occupied by the bootloader for programming the MCU). It also has 2 KB of SRAM and 1 KB of EEPROM

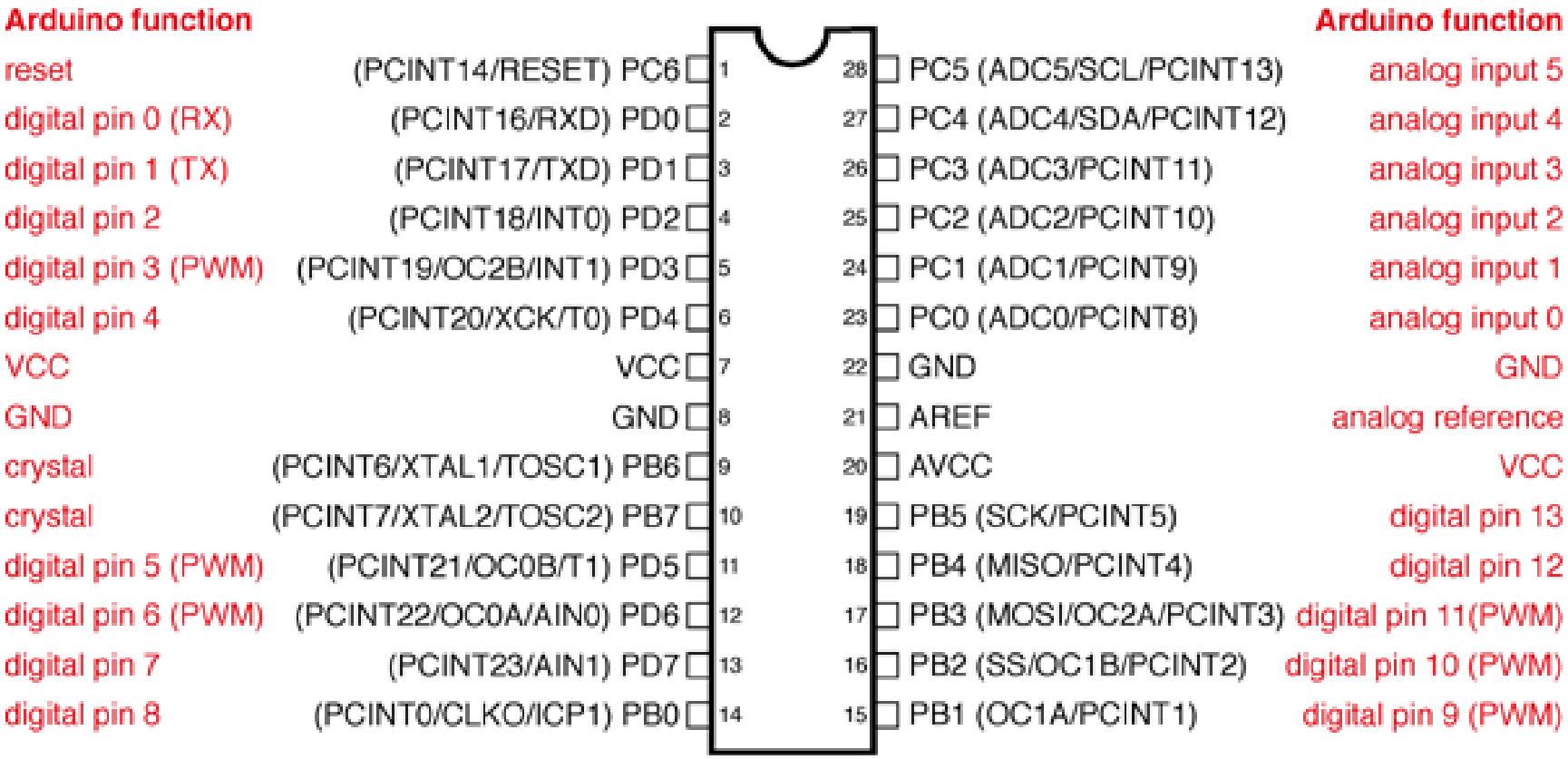




ATmega16U2 Microcontroller

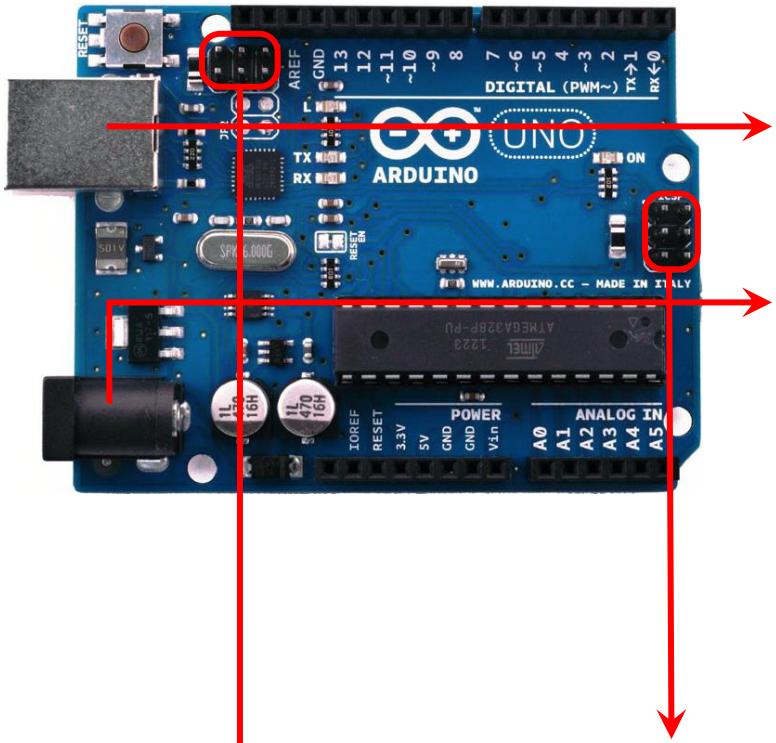
It takes care of the USB connection and serves as an interface between a USB cable and the serial USART pins on the main microcontroller.

ATmega328P-Arduino Pin Mapping



A few pins are already irreversibly wired up and unavailable for use. For instance, PB6 and PB7 are already hard-wired up to the crystal oscillator.

Arduino Uno Components



USB Connector

It is used to power the Arduino and also load code onto the Arduino.

Power Jack

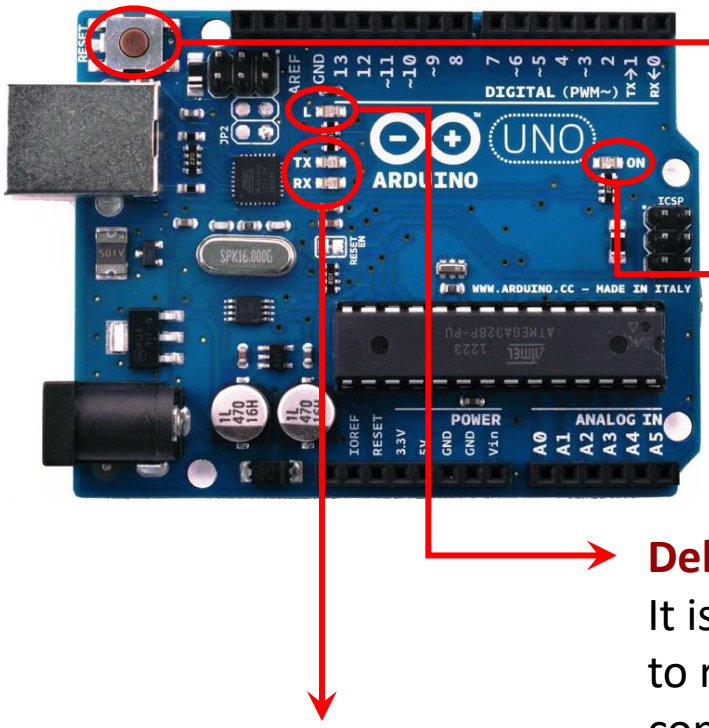
It is used to power the Arduino when **no** USB cable is connected to the USB Connector. The recommended voltage for Arduino Uno is between 7 and 12 volts.

ICSP for Atmega328P MCU

ICSP for ATmega16U2 MCU

ICSPs (In-Circuit Serial Programming) are used to update or load the firmware into the microcontroller.

Arduino Uno Components



Reset button

It can be used to restart the execution of the program uploaded in the MCU.

Power LED Indicator

This LED should light up whenever you plug your Arduino into a power source.

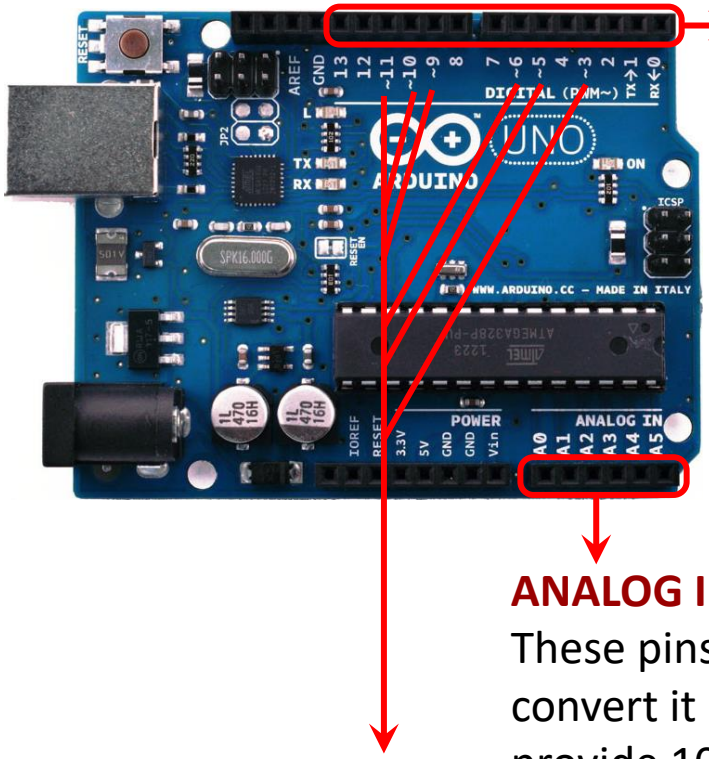
Debug LED

It is already connected to pin 13, which enables you to run your first program (blinking an LED) without connecting any additional circuitry.

TX & RX LEDs

These LEDs will give us visual indications whenever the Arduino is receiving or transmitting data like when we are loading a new program onto the board (TX: Transmit, RX: Receive).

Arduino Uno Components



DIGITAL Pins (1-13)

These pins can be used for both digital **input** (like telling if a button is pushed) and digital **output** (like powering an LED). Each pin can provide (source) or receive (sink) 20 mA as recommended operating condition and 40mA as maximum value to avoid permanent damage to the microcontroller. Each digital pin has an internal pull-up resistor (disconnected by default) of 20-50k ohm.

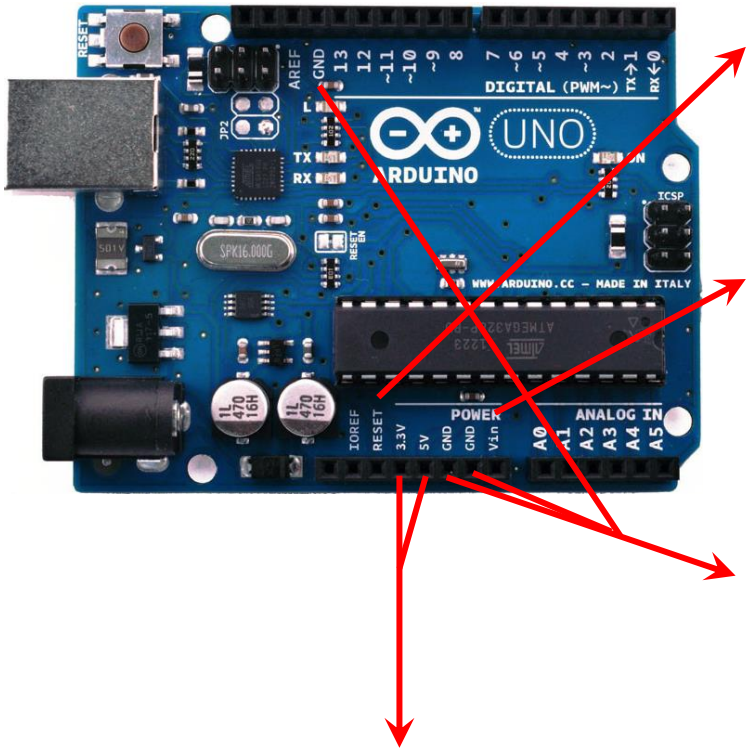
ANALOG IN Pins (A0-A5)

These pins can read the signal from an analog sensor and convert it into a digital value that we can read. Each can provide 10 bits of resolution (i.e., 10^{10}=1024 different values).

PWM Pins (~)

These pins act as normal digital pins, but can also be used for something called Pulse-Width Modulation (PWM). Think of these pins as being able to simulate analog output (like fading an LED in and out).

Arduino Uno Components



RESET

It is used to reset the microcontroller when the reset button on the board is blocked by shields.

Vin

It is used to supply voltage for Arduino through this pin (7-12V), **or**, if supplying voltage via the power jack, access it through this pin.

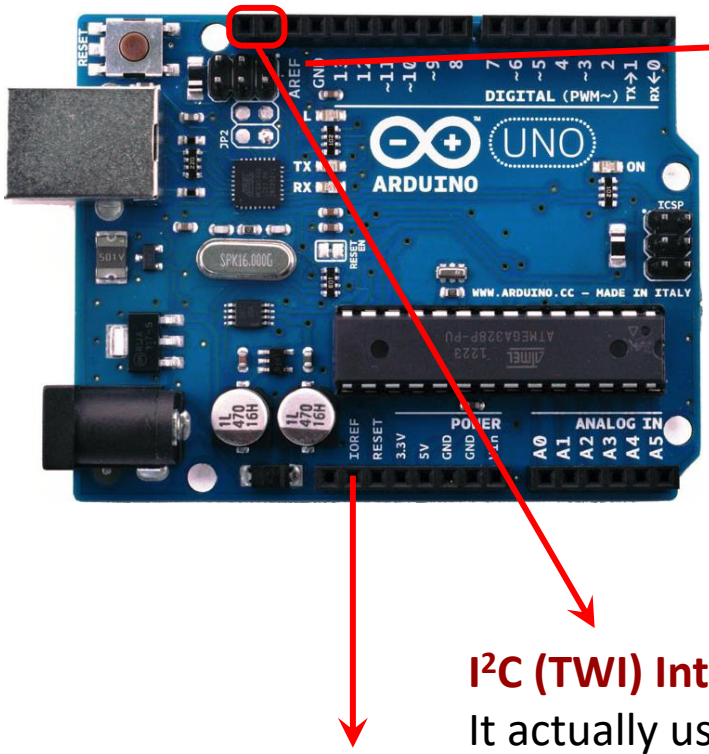
GND

There are 3 GND (Ground) pins on the Arduino, any of which can be used to ground your circuit.

5V & 3.3V

These pins supply 5 & 3.3 volts generated by the on-board regulators. Maximum current draw is 50 mA.

Arduino Uno Components



AREF

It is used as reference voltage for the analog inputs (upper end of the input range). By default, it is the same as the chip supply voltage (5V on most Arduino boards), so the analog inputs can measure between 0 and 5V. If you connect the AREF pin to a lower voltage and set the analog reference to EXTERNAL by analogReference(), you can then measure between 0 and AREF voltage to increase the measuring resolution.

I²C (TWI) Interface

It actually uses two of the analog pins (A4 or SDA and A5 or SCL).

IOREF

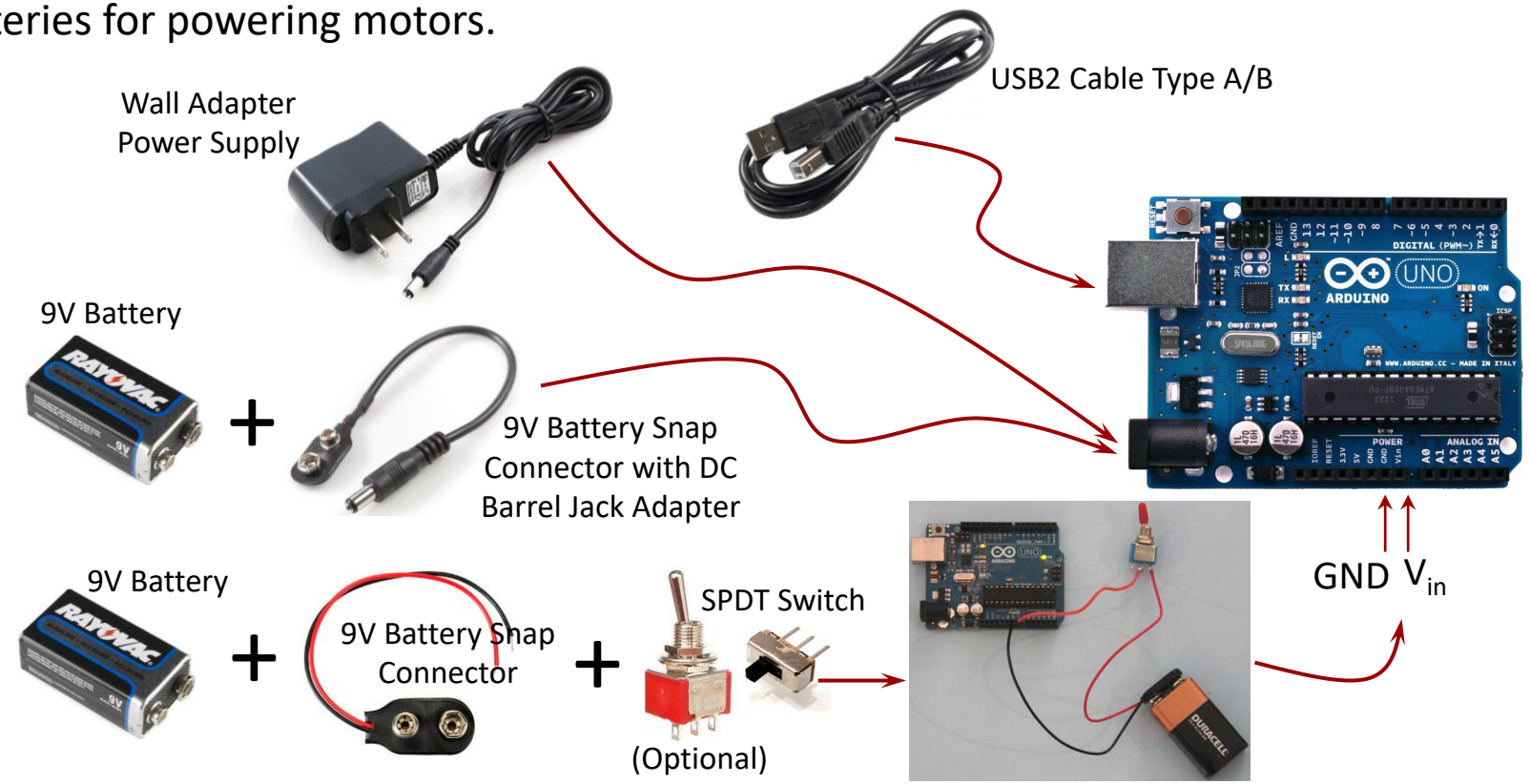
It provides the voltage reference with which the microcontroller operates. A properly configured shield can read the IOREF pin voltage and select the appropriate power source or enable voltage translators on the outputs to work with the 5V or 3.3V.

Arduino vs. AVR Microcontroller

- Arduino's microcontroller comes **pre-flashed with a bootloader** that can flash the chip for you, without need for a hardware programmer. However, the bootloader take up a portion of the flash memory so uploaded program must be smaller.
- A few pins of Arduino's microcontroller are already **irreversibly wired up** and unavailable for use.
- Arduino board comes with a built-in **USB-to-serial converter**, so you don't have to buy a separate one.
- Arduino can easily be powered by your computer's **USB power supply**. However, for some small projects, Arduino is still **bulky** (since cannot be installed directly on breadboard) and expensive.

Powering Arduino

- For **testing** your project, use **USB** power or **wall adapter power supply** for powering Arduino.
- For **actual demonstration**, use an **external 9V battery** for powering Arduino, and AA batteries for powering motors.



C/C++ Language Overview

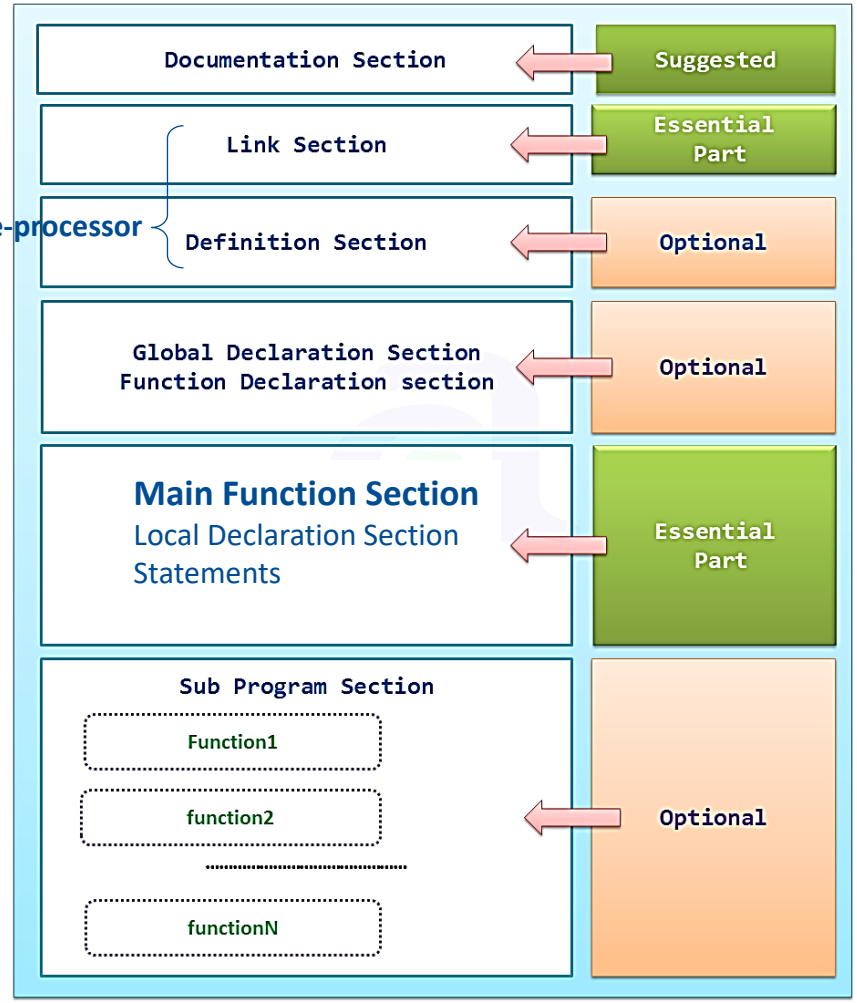
The C Programming Language

C is a **middle-level** language that is often used in place of **assembly language** for the programming of microprocessors.

Advantages of C Language:

- It is **easier** to use.
- It is **standardized**, i.e., the **same** program can be used with different microprocessors (an appropriate compiler is used to translate the C program into the machine language for the specific microprocessor). Assembly language is different for the different microprocessors.

Basic Structure of C Programming




```c
/*Documentation Section:
  Program Name: program to find the area of circle
  Author: Rumman Ansari
  Date : 12/01/2013
*/

#include"stdio.h" //Link section
#include"conio.h" //Link section

#define PI 3.14 //Definition section

float area; //Global declaration section
void message(); //function prototype declaration section

void main()
{
  float r;                          //Declaration part
  printf("Enter the radius \n");  //Executable part
  scanf("%f",&r);
  area=PI*r*r;                      // Calculation Part
  printf("Area of the circle=%f \n",area);
  message();                        // Function Calling
}

// Sub function
void message()
{
  printf("This Sub Function \n");
  printf("we can take more Sub Function \n");
}
```

C Programming

1. Keywords:

In C, certain words (in lower case letters) are reserved as **keywords** with specific meanings and should not be used for any other purpose in a C program. The ANSI C standard keywords:

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

C Programming

2. Statements:

- Statements are the entries which make up a program.
- Every statement is terminated by a **Semicolon** (**;**). Forgetting to end a line in a semicolon will result in a compiler error.
- Statements can be grouped together in **blocks** by putting them between braces, i.e., { }.

```
{
statement 1;
statement 2;
}
```

Note: Normally, statements in a program execute one after the other in the order in which they are written, which is called **sequential execution**.

3. Comments:

Comments are **important** for annotating specific lines of code and documentation.

- When **//** is put on any line, the compiler ignores all text after that symbol on the same line (**single-line comment**).
- Everything written between **/*** and ***/** will not be compiled (**multiline comment**).

```
// comment…
```

```
/*
Comments…
*/
```

C Programming

4. Main Function:

Every C program must have a function called **main()**. By convention, a **return** value of **0** from **main()** is used to indicate **normal program termination**.

```
int main (void) {
  ...
  ...
  ...
  return (0);
}
```

Example: Addition operation in C

```
#include <stdio.h>
int main() {
  int op1, op2, sum; // variable declaration
  op1 = 5; // variable definition
  op2 = 3;
  sum = op1 + op2; // addition operation
  printf("sum of %d and %d is %d", op1, op2, sum);
  return (0);
}
```

❖ Note that in C programs all spaces are ignored by the compiler.

Arduino Programming

Installing and Running the Arduino IDE

- Access the Arduino website at *www.arduino.cc* and download the newest version of the IDE from the Download page.
- Install Arduino IDE and the required drivers.
- Connect the Arduino to your computer via USB 2.0 cable type A to B.
- Launch the Arduino IDE and go to **Tools** → **Board** and ensure that the right board is selected i.e., **Arduino Uno**.
- Navigate to **Tools** → **Port** and select the appropriate port.

If you still have a problem, refer to
https://www.arduino.cc/en/Guide/HomePage or
https://learn.sparkfun.com/tutorials/installing-arduino-ide

- To compile and upload your program onto your Arduino, Click the Upload button

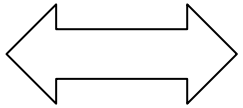
Programming Arduino

Arduino IDE provides a set of Arduino core functions and libraries to make programming much easier. Arduino Boards can also be programmed in **pure C** (which is faster and more optimum) using Arduino IDE.

Arduino Programming

```
void setup() {
pinMode(8,OUTPUT);
}
void loop() {
digitalWrite(8, HIGH);
delay(1000);
digitalWrite(8, LOW);
delay(1000);
}
```

Sketch uses **928 bytes** (2%) of program storage space.



C

```
#include <avr/io.h>
#include <util/delay.h>
int main (void) {
  DDRB = 0b00000001;
  while (1) {
    PORTB = 0b00000001;
    _delay_ms(1000);
    PORTB = 0b00000000;
    _delay_ms(1000);
  }
  return (0);
}
```

Sketch uses **178 bytes** (~0%) of program storage space.

Types of Interfaces to MCUs

You can use three types of interfaces to your microcontroller:

- **Digital I/O**: Switches as inputs, LEDs or buzzers as outputs
- **Analog I/O**: Sensors of various types, PWM
- **Serial**: A serial communications protocol of which there are four main types: TTL Serial, I^2C, 1-Wire, and Serial Peripheral Interface (SPI).

Digital I/O

Arduino Programming: Digital I/O

```
void setup() {
    // put your setup code here, to run once:
    pinMode(LED_BUILTIN, OUTPUT); // Pin 13
}
void loop() {
    // put your main code here, to run repeatedly:
    digitalWrite(LED_BUILTIN, HIGH);
    delay(1000);
    digitalWrite(LED_BUILTIN, LOW);
    delay(1000);
}
```

setup() and **loop()** **functions must** be included in **all** Arduino programs and by convention, do not return anything

Code within the **curly braces** of the **setup()** function is executed **once** at the start of the program. This is useful for one-time settings. Code within the **curly braces** of the **loop()** function **repeats forever** as long as the Arduino is on.

Arduino Programming

```
void setup() {
    // put your setup code here, to run once:
    pinMode(LED_BUILTIN, OUTPUT); // Pin 13
}
void loop() {
    // put your main code here, to run repeatedly:
    digitalWrite(LED_BUILTIN, HIGH);
    delay(1000);
    digitalWrite(LED_BUILTIN, LOW);
    delay(1000);
}
```

delay(time)

Pauses the program for the amount of time (in **milliseconds**)

Configuring pin direction:

pinMode(pin , direction)

Pin Number

Pin Direction: INPUT (to sink current) or OUTPUT (to sourse current)

(Pins are INPUTs by default)

Setting the pin state:

digitalWrite(pin , state)

Pin Number

Pin State: HIGH (5V) or LOW (0V)

(It remains in this state until it is changed again)

Note: Arduino language is C/C++. Arduino Language Reference: https://www.arduino.cc/reference/en/

Basic Variable Types

A **Variable** is a named memory location that can **hold** various **values**. To declare a variable, the type is inserted before the variable name.

```
VariableType VariableName = Value;
```

Type	Size	Value range	
char	1 byte	0 to 255 (2^8-1)	`char myChar = 'A';`
int	2 bytes	-32,768 to 32,767 (-2^{15} to 2^{15}-1)	
float	4 bytes	-3.4028235E+38 t0 3.4028235E+38	Precision: 6 digits
double	8 bytes		Precision: 14 digits

Qualifiers alters the meaning of base data types to yield a new data type:

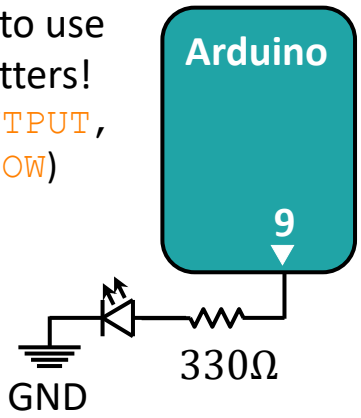
- **Size qualifiers**: `long` and `short`
- **Sign qualifiers**: `unsigned` and `signed` (a variable is signed by default)
- **Constant qualifier**: `const` (value of cost cannot be changed in the program)
- **Volatile qualifier**: `volatile`

Boolean:
```
boolean myBool = true;
boolean myBool = false;
```

Turn On an LED Using Arduino

```
const int LED = 9;
void setup() {
  pinMode (LED, OUTPUT);
  digitalWrite(LED, HIGH);
}
void loop() {
  //we are not doing anything in the loop!
}
```

Remember to use CAPITAL Letters! (INPUT, OUTPUT, HIGH, LOW)



This code is a **variable declaration**. All instances of LED in the program will be **replaced** with 9 when they are called.

const is a **variable qualifier** and makes a variable "**read-only**" and its value cannot be changed during program execution.

(When you are defining values that will not change, using the **const** qualifier is **recommended**.)

Note: **loop**() repeats forever as long as the Arduino is on. If you want your Arduino to do something **once at boot only**, you still need to include the **loop**() function, but you can leave it empty.

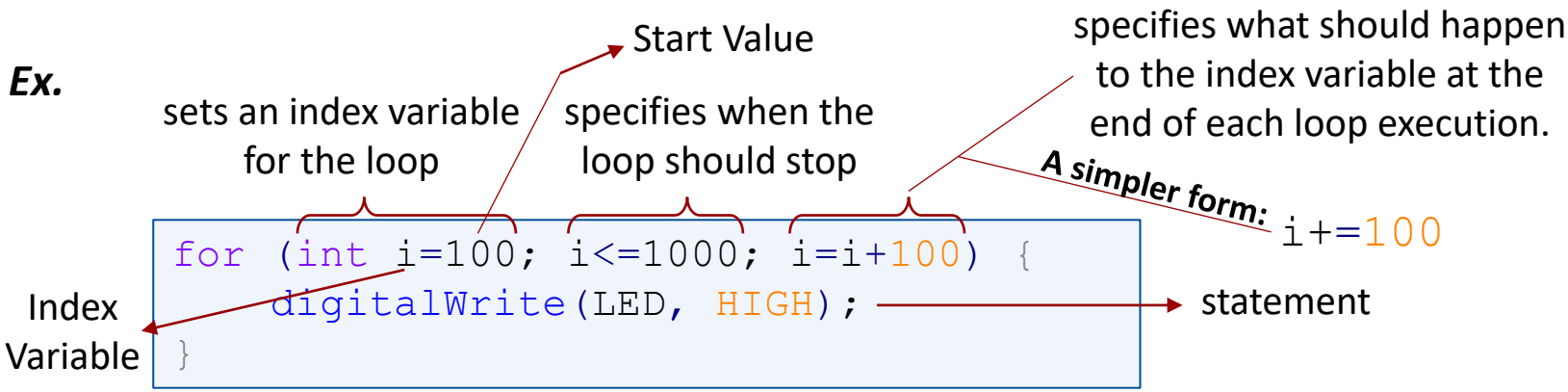
"for" Loop

The **for loop** is used to **repeat** a **block of statements** enclosed in **curly braces**.

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

The **initialization** happens first and exactly once. Each time through the loop, the **condition** is tested; if it's ***true***, the **statement block**, and the **increment or decrement** is executed; then, the **condition** is tested <u>again</u>. When the condition becomes ***false***, the loop <u>ends</u>.

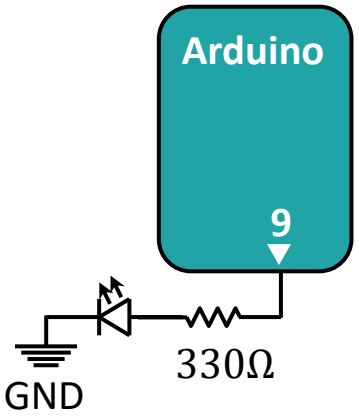
Ex.



LED with Changing Blink Rate

```
const int LED=9;
void setup() {
  pinMode (LED, OUTPUT);
}
void loop() {
  for (int i=100; i<=1000; i=i+100) {
    digitalWrite(LED, HIGH);
    delay(i);
    digitalWrite(LED, LOW);
    delay(i);
  }
}
```

"for" Loop



"for" Loop Description

```
…
void loop() {
  for (int i=100; i<=1000; i=i+100) {
    digitalWrite(LED, HIGH);
    delay(i);
    digitalWrite(LED, LOW);
    delay(i);
  }
}
```

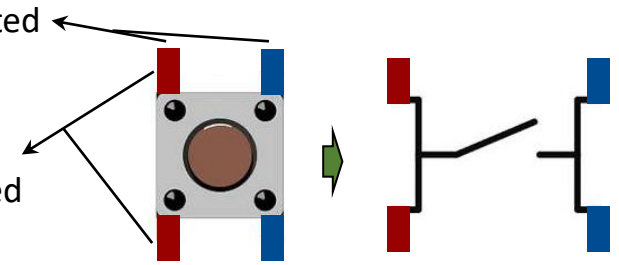
1. `i` equals 100. The condition is true, hence,
2. The `LED` is set high, and stays high for 100ms (i.e., the current value of `i`).
3. The `LED` is set low, and stays low for 100ms, (i.e., the current value of `i`).
4. At the end of the loop, `i` is incremented by 100, so it is now 200.
5. 200 is less than or equal to 1000, so the loop repeats again.
6. The `LED` is set high, and stays high for 200ms, (i.e., the current value of `i`).
7. The `LED` is set low, and stays low for 200ms, (i.e., the current value of `i`).
8. At the end of the loop, `i` is incremented by 100, so it is now 300.
9. This process repeats until `i` surpasses 1000 and loop ends.

The outer **loop() function** repeats again, sets the `i` value back to 100 and starts the process again.

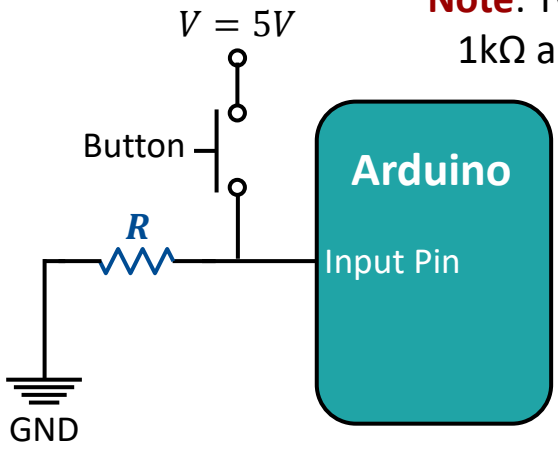
Reading Digital Inputs

Till now, you are able to **generate** (**write**) **digital outputs** (like Blinking LEDs), the next step is to **read digital inputs**, such as switches and buttons, so that you can interact with your project in real time.

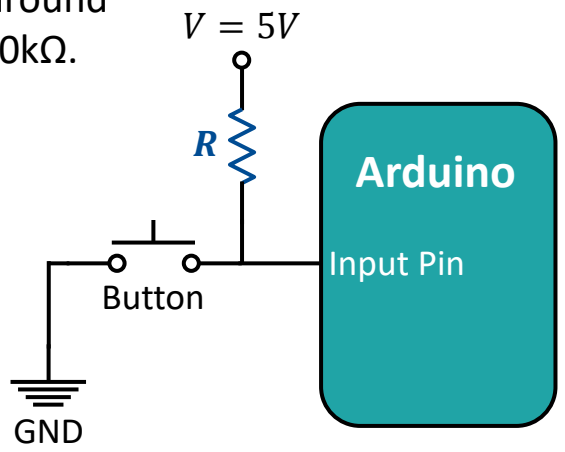
Tactile Push Button



Note: Typically, a pulldown resistor is around 1kΩ and a pullup resistor is around 10kΩ.



Pulldown Resistor is used to make the initial state of input pin **LOW**



Pullup Resistor is used to make the initial state of input pin **HIGH**

"if/else" Statement

"**if**" statement tests whether a certain **condition** is **true**.

If its **condition** is true, the **statement(s) A** is executed, if not, the **statement(s) B** (i.e., "**else**" statement) is executed.

```
if ( condition ) {
  statement(s) A;
}
else {
  statement (s) B;
}
```

Two Other Formats:

```
if ( condition 1 ) {
  statement(s) A;
}
else if ( condition 2) {
  statement(s) B;
}
else {
  statement(s) C;
}
```

An unlimited number of such **else if** branches is allowed.

```
if ( condition ) {
 statement(s) A;
}
```

"**if**" statement **without** "**else**" statement

Comparison and Boolean Operators

Comparison Operators can be used inside the **condition** of an "if" statement.

Beware of accidentally using the **single equal sign** (=), which is the assignment operator (**puts** a value into a variable), in "**if**" statements. Instead, use the **double equal sign** (==), which is the comparison operator (**tests** whether the values on the both sides of the sign are equal).

Comparison Operators:
== (equal to)
!= (not equal to)
< (less than)
> (greater than)
<= (less than or equal to)
>= (greater than or equal to)

Boolean Operators can be used with a **condition** or between the different **conditions** of an "if" statement.

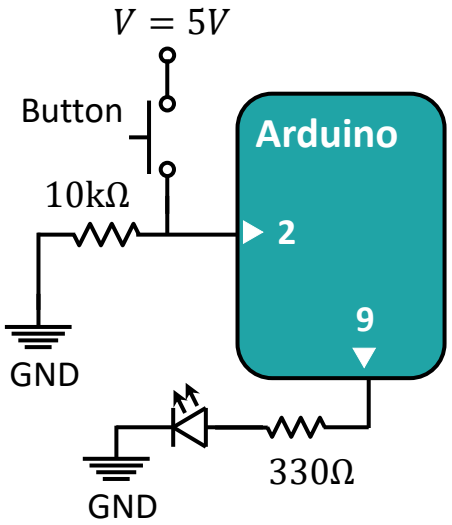
- True only if **both** conditions are true: "if (x > 0 && y > 0)" is true only if both x and y are greater than 0.
- True if **either** condition is true: "if (x > 0 || y > 0)" is true if either x or y is greater than 0.
- True if the condition is **false**: "if (!(x>0))" is true if x is not greater than 0.

Boolean Operators:
&& (logical **and**)
|| (logical **or**)
! (**not**)

LED Control with a Button (1)

The LED stay ON while the button is held down, it stay off while the button is unpressed.

```
const int LED=9;
const int BUTTON=2;
void setup() {
  pinMode (LED, OUTPUT);
  pinMode (BUTTON, INPUT);
}
void loop() {
  if (digitalRead(BUTTON) == HIGH){
    digitalWrite(LED, HIGH);
  }
  else {
    digitalWrite(LED, LOW);
  }
}
```



Since pins are **inputs by default**, this line is not necessary.

Value of a digital input, either HIGH or LOW, is read by digitalRead(pin)

Pin Number

"while" Loop

A **while loop** is used to **repeat** a **block of statements** enclosed in **curly braces** continuously, and infinitely, until the expression inside the parenthesis, becomes false.

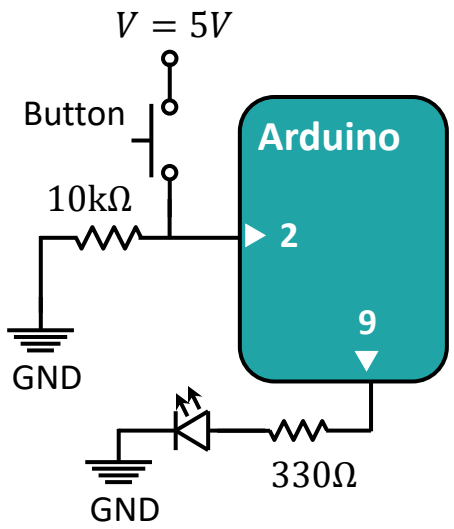
```
while (condition) {
  statement(s);
}
```

Note: Something must change the **condition** state, or the while loop will never exit.

LED Control with a Button (2)

The LED blinks while the button is held down, it stay ON while the button is unpressed.

```
const int LED=9;
const int BUTTON=2;
void setup() {
  pinMode (LED, OUTPUT);
  pinMode (BUTTON, INPUT);
}
void loop() {
  while (digitalRead(BUTTON)){
     digitalWrite(LED, HIGH);
     delay(500);
     digitalWrite(LED, LOW);
     delay(500);
  }
  digitalWrite(LED, HIGH);
}
```



Note: Both expressions have the same meaning:

```
while (digitalRead(BUTTON)){
…
}
```

=

```
while (digitalRead(BUTTON) == HIGH){
…
}
```

Functions

Functions are blocks of code that can accept input arguments, execute code based on those arguments, and optionally return a result.

Functions are created to **simplify** a program and also to **encapsulate** actions which are needed to be performed **multiple times** in a program.

```
outputType functionName (input1Type input1, input2Type input2, ...) {
  ...
  statement(s);
  ...
  return output;
}
```

Function Body

Datatype of **output** (returned value)

Note: **outputType** is "**void**" if nothing is returned.

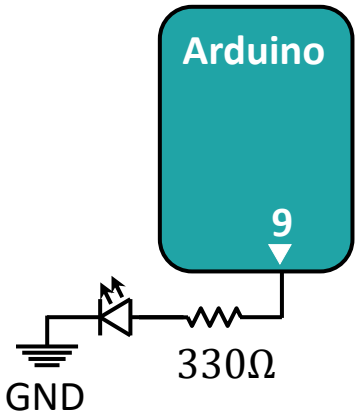
There are two required functions in an Arduino sketch, **setup()** and **loop()**. Both does not have any input and returned value. Other functions must be created everywhere **outside** the brackets of those two functions.

Functions can be "**called**" from everywhere in the program. →

```
output = functionName (Arg1, Arg2, ...);
```

Blinking an LED Using a Function

In this example, the LED blinks every "i" second.




```
const int LED=9;

void setup() {
pinMode (LED, OUTPUT);
}

void loop() {
int i = 2;
int k;
k = secondFunction(i);
digitalWrite(LED, HIGH);
delay(k);
digitalWrite(LED, LOW);
delay(k);
}

int secondFunction(int x) {
int result;
result = x * 1000;
return result;
}
```

Code Description

Global and Local Variables:

The variables which are declared outside of all functions in the program are **global variables**. These variables can be used and changed by any function within the program.

For example, *Global variable*: `LED`

The variables which are declared within a specific function are **local variables**. These variables can be used and changed only within that specific function.

For example, *Local variables*:
in **loop** function: `i`, `k`
in **secondFunction** function: `x`, `result`

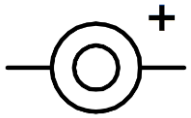
```
const int LED=9;

void setup() {
pinMode (LED, OUTPUT);
}

void loop() {
int i = 2;
int k;
k = secondFunction(i);
digitalWrite(LED, HIGH);
delay(k);
digitalWrite(LED, LOW);
delay(k);
}

int secondFunction(int x) {
int result;
result = x * 1000;
return result;
}
```

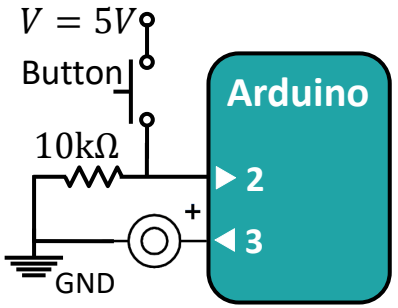
Function is "called" here.

A function to multiply a number by 1000.

Buzzer

A **buzzer** or **beeper** is an audio signaling device, which is usually **piezoelectric**. Typical uses of buzzers and beepers include alarm devices, timers, and confirmation of user input.





An **Active Buzzer** has an internal oscillator and generates the sound itself. Hence, you can simply turn it ON/OFF with an Arduino digital pin, just like turning ON/OFF an LED.

A **Passive Buzzer** has no internal oscillator and needs a signal source that provides the sound signal. You need to use a function in your code to create frequencies.

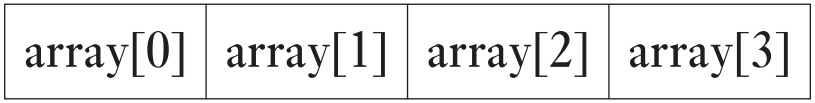
Array

An **Array** is a collection of variables which have the same data type and are accessed with an index number.

❖ Different ways to create (declare) an array:

```
int myInts[6]; // without initialization
int myArray[5] = {9,3,2,4,3}; // with initialization
int myPins[] = {2, 4, 8, 3, 6}; // without array size
```

- Size of the array is indicated between square brackets [].
- The compiler counts the elements and creates an array of the appropriate size.



❖ Accessing an Array: Arrays are zero indexed (the first element is at index 0).

myArray[0] contains 9
myArray[4] contains 3
myArray[5] is invalid and contains random information (other memory address)

Array

❖ Assigning a value to an array:
myInts[3] = 10;

```
int myInts[5]={9,3,2,4,3};
```

❖ Retrieving a value from an array:
x = myInts[2];

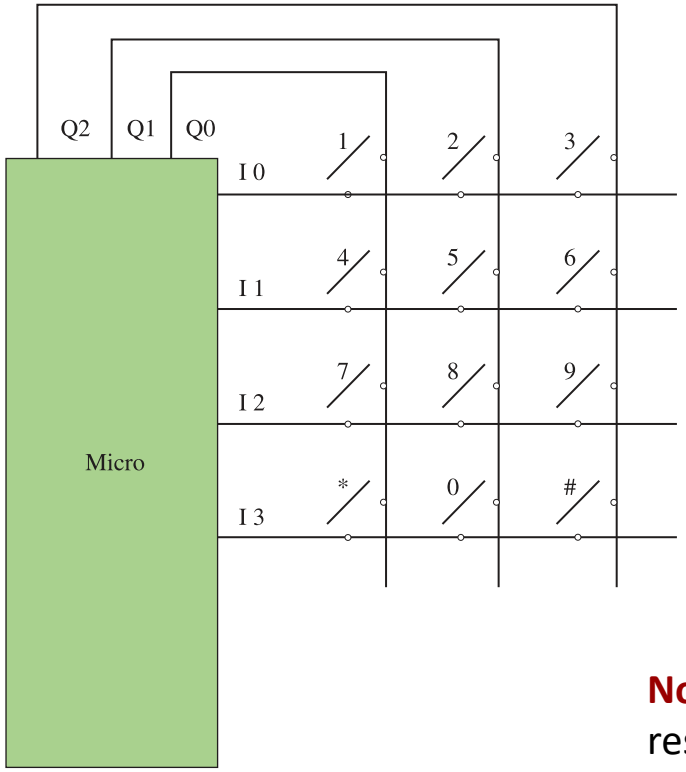
❖ Similarly, **multidimensional arrays** can be defined as: array[x][y];

```
char keys[4][3] = {
{'1','2','3'},
{'4','5','6'},
{'7','8','9'},
{'*','0','#'} };
```

array[0][0]	array[0][1]
array[1][0]	array[1][1]
array[2][0]	array[2][1]

Using a Matrix Keypad

Keypads use switches arranged in a **matrix**. To determine which keys are pressed, the microcontroller will take each of the output pins Q0 to Q2 high in turn, and see what value is presented at each of the inputs I0 to I3.



Note that if the microprocessor does not support internal pullup resistors, then pullup resistors would be required on each input.

Using a Matrix Keypad

```
#include <Keypad.h>
char keys[4][3] = {
{'1','2','3'},
{'4','5','6'},
{'7','8','9'},
{'*','0','#'} };
byte rowPins[4] = {2, 7, 6, 4};
byte colPins[3] = {3, 8, 5};
Keypad myKeypad = Keypad(makeKeymap(keys), rowPins, colPins,4,3);
void setup() {
}
void loop() {
char key = myKeypad.getKey();
if (key != null) {
// Do something…
}
}
```

#include is used to include outside **libraries** in the code. Note that **#include** has **no semicolon** terminator.

To prepare the Arduino to control a keypad, first, a keypad "**object**" must be created. Using this **object**, you can control the keypad during the code.

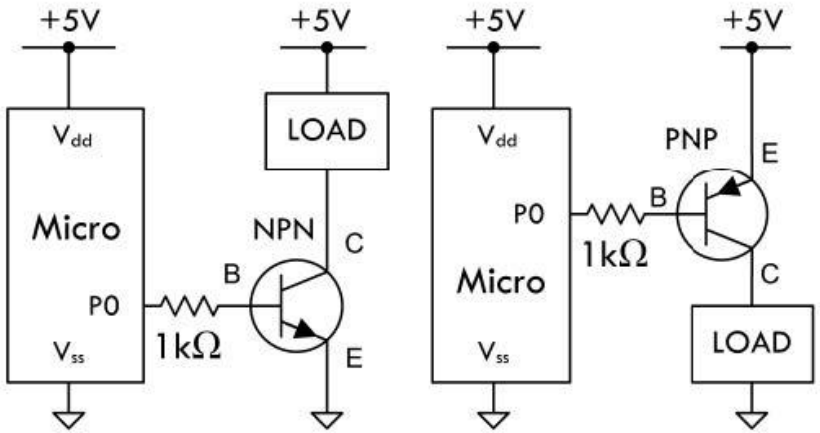
Switch Debouncing

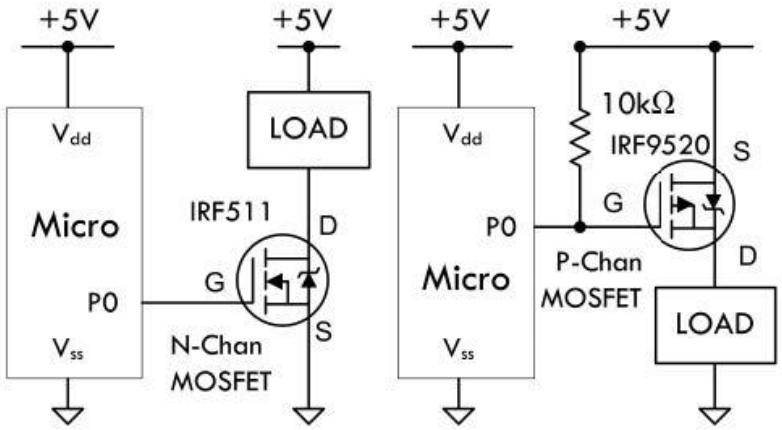
Here is one of the software methods for debouncing the pushbutton:

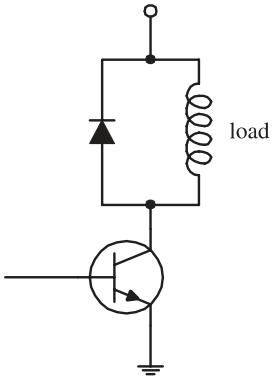
```
const int debouncePeriod = 100; //ms
long lastKeyPressTime = 0;
void loop() {
long timeNow = millis();
  if (digitalRead(5) == LOW && lastKeyPressTime > timeNow + debouncePeriod) {
  // button pressed, and enough time elapsed since last press
  // do what you need to do
  // …
  lastKeyPressTime = timeNow;
  }
}
```

High-Power Digital Outputs

Most microcontrollers will reliably provide us with only around 40 mA of source or sink current as a direct digital output. If you want to drive a higher power load, such as a relay or a high-power LED, then you need to use a **transistor** (BJT or MOSFET).







Remember to use a reverse-biased fly-back diode across the inductive loads (DC Motors, Solenoids, Relays, ...), to prevent voltage spikes damaging the transistor during switching.