SOAR CHALLENGE

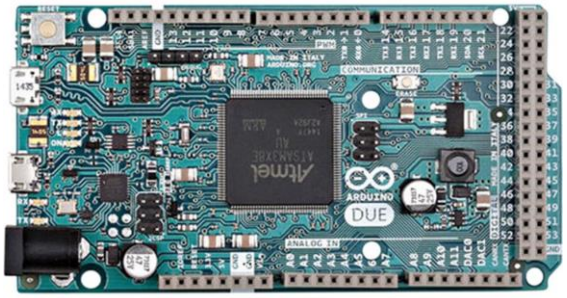
Arduino Guidebook

Table of Contents

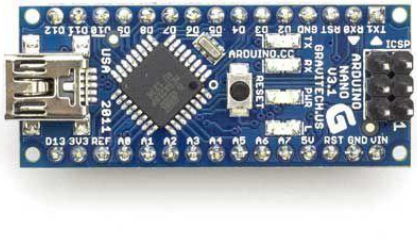
Microcontroller Introduction

Microcontroller is a task-oriented integrated controller. Any board comprises of an integrated circuit system (A board with CPU) with numerous input and output pins, or I/O for short is a microcontroller. Microcontrollers are typically designed for broad use, although they could also be customized for specialized applications. Teensy, PIC, Raspberry PI, Arduino are examples of a broad use application microcontrollers.

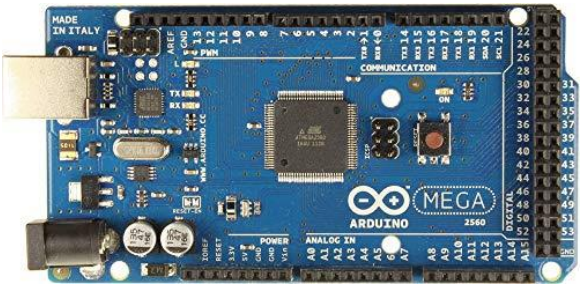
Arduino Uno will be used to introduce microcontrollers in this guide. Arduino is a commonly used microcontroller that is designed to be simple0. Arduino makes a variety of microcontroller models to meet a variety of customer needs:



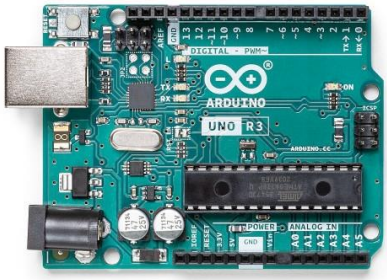
Arduino Due



Arduino Nano



Arduino Mega



Arduino Uno

Arduino Uno (8 Bit Microcontroller)

Specifications:

Input Voltage:	6V to 12V	$(V_{in}, Power\ Jack)$	
Output Voltage:	3.3V and 5V		Current: 200mA
Analog Pins:	6	*(A0 to A5)*	
Digital Pins:	14	*(0 to 13)*	Current: 40mA
Serial Communication:	Pin 0 and Pin 1	*0 (Rx), 1 (Tx)*	
External Interrupts:	Pin 2 and Pin 3	*(Trigger Interrupts)*	
PWM:	Pin 3, 5, 6, 9, 11	*(8 Bit PWM)*	
Frequency:	16 MHz		

Warning!

Do not power Arduino with both V_{in} and Power Jack, or both V_{in} and USB at the same time.

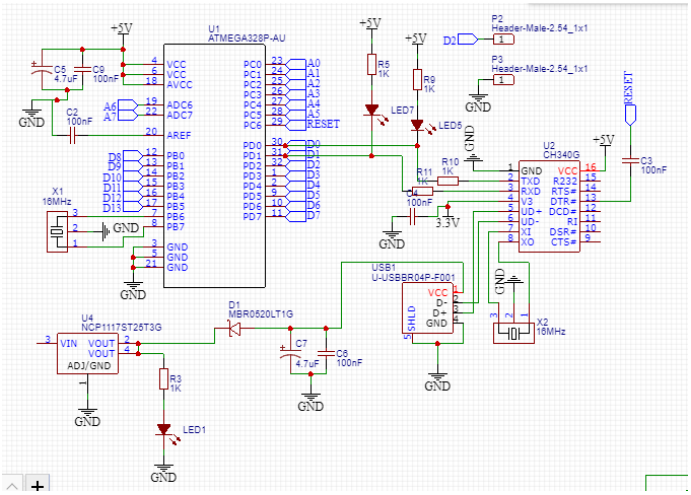
Arduino Driver

With the raising popularity of Arduino, purchasing 3rd party Arduino board has become a norm. 3rd party Arduino board significantly less expensive, and they function almost identically. When the program becomes more complex, 3rd party Arduino board may not function as well as the original.

Some 3rd party Arduino board requires an additional driver to function. We will be using a 3rd party Arduino Uno due to cost, and this board will require the installation of a "ch340" driver on the computer to operate. You can download the driver in this link: https://sparks.gogo.co.nz/ch340.html.

Schematic

A schematic is a blueprint of the component. It displays what the component consists of, thus enabling the developer to build their system around the component.



Datasheet

Datasheet is a document (usually in PDF) that contains all of the relevant information of a component. It may comprise schematics, specification, and many other useful details.

Digital pins

There will be 3 parts in this topic:

1. Digital presentation;
2. Signal Transmission;
3. Communication.

Digital Presentation

In the digital world, digital signals are stored as 1s and 0s. So, what are these 1s and 0s? When we say that a signal is showing 1 or “high”, we meant that the signal is reading 5V. On the other hand, 0V is shown as 0 or “Low”. Note that 1 may not represent 5V. Some microcontrollers use 1 to represent 3.3V, 9V, 12V or 24V (Wi-Fi transmission).

1	5V (Most common)	High
0	0V	Low

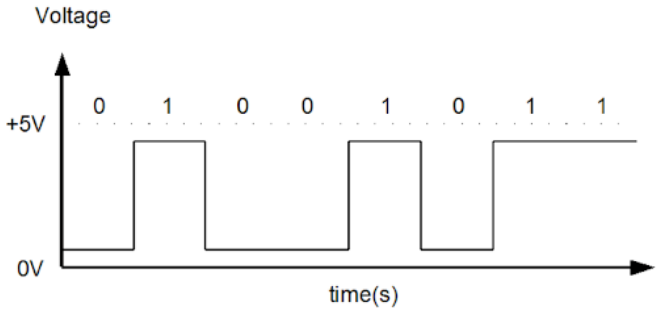
So, why does the voltage vary? The voltage used depends on the usage. For instance, mobile phone reads 3.3V as “high” in order to reduce power consumption. Wi-Fi transmission uses 24V to allow signals to travel further with fewer noise. (Noise is stray electricity from the surroundings that interferes with the signal's reading)

Now, what about voltage signals in between 0V and 5V?
In Arduino, > 3V is recorded as “high” on a 5V board
> 2V is recorded as “high” on a 3.3V board

Higher voltage allows you to widen the range of a high signal thus lowering the inaccuracy produced by processing the signal in a noisy environment. (*You should be able to see why a larger voltage is required for a high signal)*

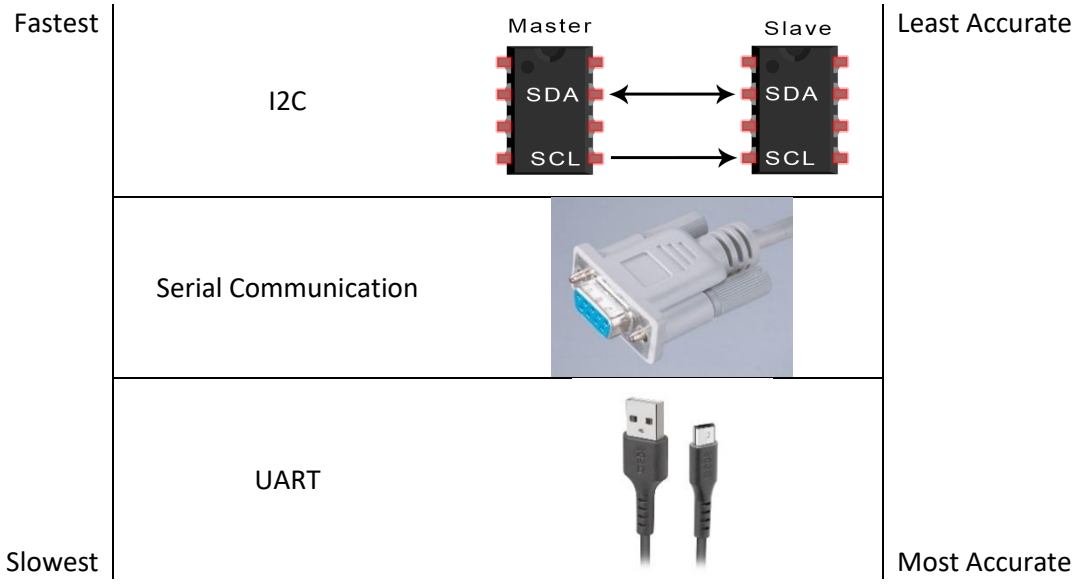
Signal Transmission

You may recognise digital signals being presented just like the image displayed below. The image below represents a timed digital signal transmission where each bit is sent at a specific interval. The frequency of the microcontroller determines the timed interval. As signals are transmitted at a timed interval, the microcontroller's resources (CPU computations) are taxed. As a result, not all pins will be capable of high-speed data transmission. (*You may still use the Digital Input/Output (DIO) pins to design your own signal transmission, but it will be slower and possibly less precise than conventional protocol. We are talking about transmission rates in microseconds*)



Communication

There are three main types of communications in microcontrollers: I2C, serial communication, and UART.



I2C

I2C is a communication protocol that allows you to transmit data at high speed. It synchronises data transfer between two or more devices by using a shared clock between the microcontrollers. There is a master controller (primary controller) and several slave controllers in this protocol *(other controllers communicating with the main controller).*

Due to the lack of parity bits to restore damaged data, data can be transmitted at a faster pace. However, data transmitted using this protocol are prone to be corrupted. It is recommended that shielded wires or extremely short wires are used while using this protocol. (It is definitely not wise to use a chain of short wires)

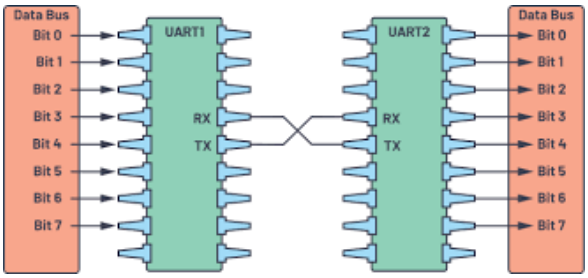
Serial Communication

Serial communication transmit data in parallel and at a slower pace than I2C, making it more precise than I2C. Moreover, serial communication is considerably quicker than UART because it uses parallel transmission. This communication protocol is used in monitors and certain devices in the modern world that demand both speed and precision.

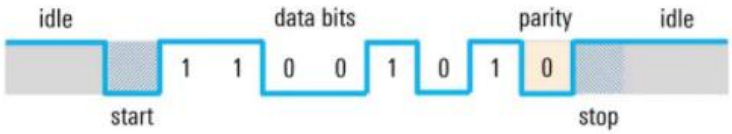
HDMI, VGA and DGI sends signals in parallel.

UART

UART is widely known as USB. It makes use of a transmitter (Tx) and a receiver (Rx) pin *(Rx – Pin 0, Tx- Pin 1 on Arduino Uno)*. It is used to communicate between 2 devices and is connected in the in the manner shown below. A microcontroller's Tx pin is connected to another microcontroller's Rx pin.



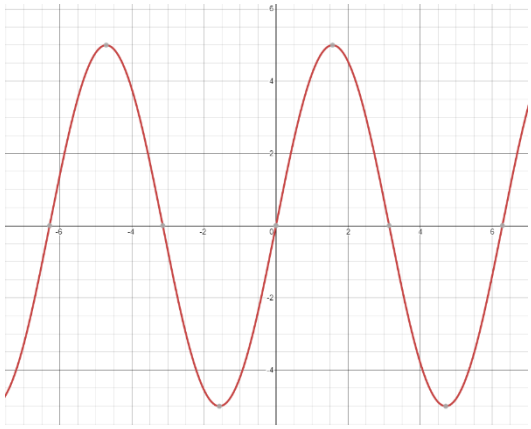
This protocol consists of start bit, stop bit, data bit and parity bit. Data are split into smaller packages and is contained between start and end bit before it is sent. As data is transmitted in smaller packets, the protocol can request a data resend if the recipient receives corrupted data beyond repair with parity bits. The data checking function in this protocol results in slower data transmission than other protocols.



In Arduino, the USB or UART communication uses pin 0 and 1. As a result, these pins are reserved for communication and cannot be used for any other purpose. So, what if we need the Arduino to communicate with both the computer and with other devices simultaneously? Arduino has a "SoftwareSerial" library which uses other pins for the second UART communication *(pin 2 and 3 for Arduino Uno)*. More information on SoftwareSerial can be found in the following link: https://www.arduino.cc/en/Reference/softwareSerial

Analog Pins

Analog pins are pins that read and produce voltage signals in sinusoidal forms. These pins are required to read and produce signals that are not supported by the digital pins. For instance, reading light intensity, sound control, resistivity sensor, and so on.

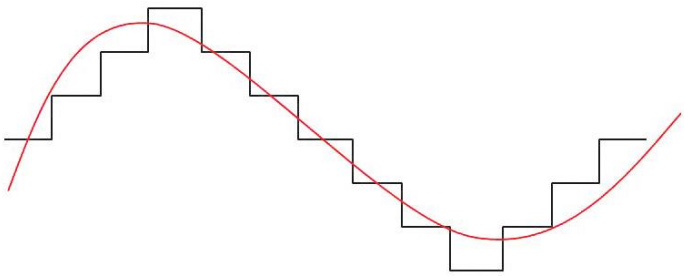


Maximum Voltage (5V for most component) at its peak and 0V at the minimum point

Signals must be converted before they can be read or sent by a microcontroller. Analog signals are converted into digital signals with *Analog to Digital converter (ADC)* before microcontroller reads it and the microcontroller converts digital signal into analog signal with *Digital to Analog converter (DAC)* before sending it to the analog pin.

So, how is this analog signal read? The signal is read in bits, so 0 represents 0V and 255 represents 5V in Arduino Uno. The value 255 does not always represent 5V as this depends on the number of bits the microcontroller has. For Arduino Uno, 255 is due to the 8-bit system Arduino Uno has (2^8 = 256).

As signals are being converted back and forth, these converter uses a step-like system to convert the signal shown below.

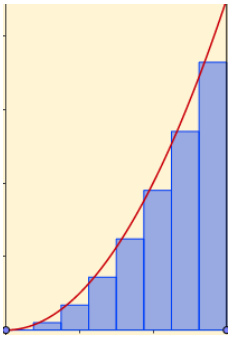


Red – Analog Signal

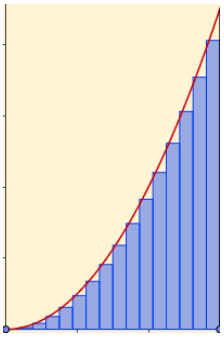
Black – Converted signal

A higher resolution converter will be required to smoothen the reading and transmission of analog signals. Since Arduino Uno is an 8-bit microcontroller, it can only transform analog signals with 8-bit precision. As a result, the sound quality generated by the Arduino Uno will not be superior than that produced by a 16, 32, or 64-bit microcontroller.

A better representation for ADC and DAC resolution will be the idea behind Riemann Sum.



8-bit converter

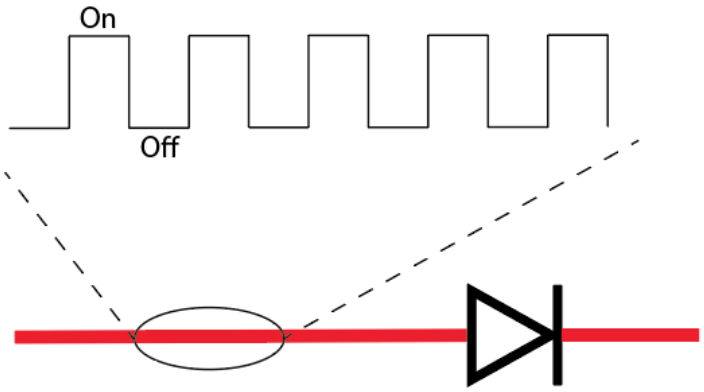


16-bit converter

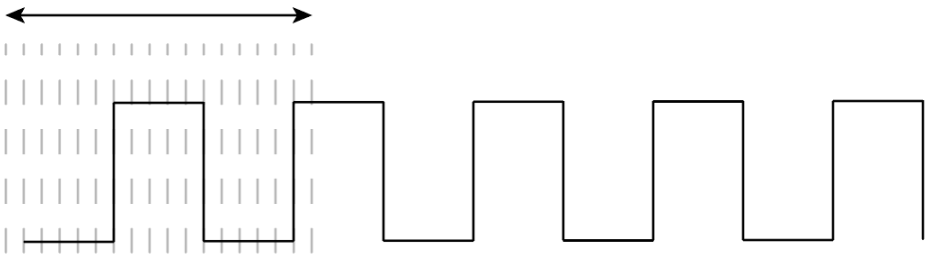
The more rectangles you can fit under a curve in Riemann Sum, the better the representation of the area under the curve. The image above is taken from GeoGebra. You can play with the site's function here: https://www.geogebra.org/m/RCVce5W4. Therefore, in the context of ADC and DAC, the more bits you can supply to represent the signal, the better the signal resolution.

Pulse Width Modulation (PWM)

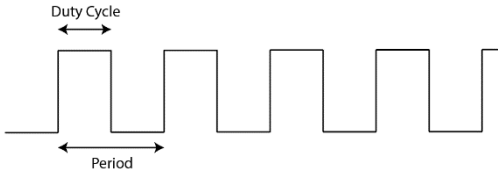
PWM is a technical innovation that allows us to alter power delivered through our equipment. You may or may not have heard of the term. PWM averages out the amount of power or current sent to a component. So, what is the mechanism behind this? PWM turns on and off electrical power at a high frequency that is undetectable to the naked eye. The image below showcase an LED powered with PWM.

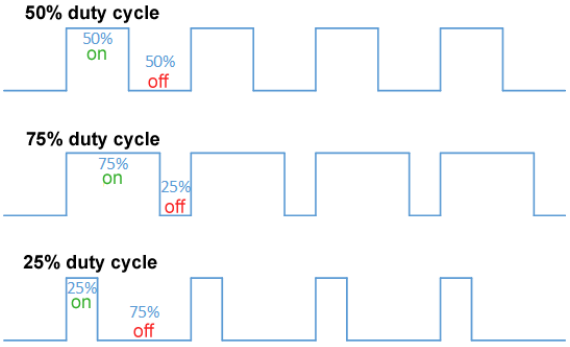


The LED in the above image is switched on 50% of the time and turned off 50% of the time, resulting in a 50% reduction of the LED's brightness. So, what if we want to tweak the brightness of this LED even more? We may modify the amount of time the LED is turned on by splitting the PWM period into smaller portions.



The period at which the PWM stays on high is called the ***duty cycle*** and the accuracy of the duty cycle is dependent on the frequency of the microcontroller.



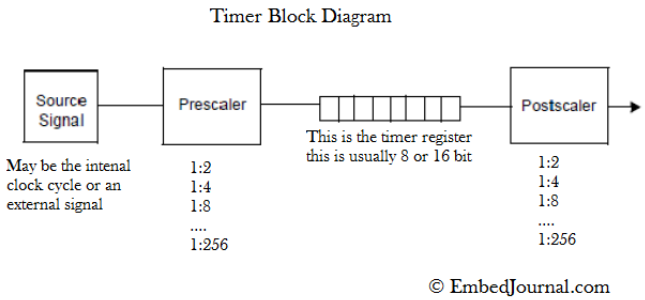


You may ask, why do we not simply adjust the voltage to the brightness we want to achieve? This is because all components have an operating voltage. For instance, a 3.3V LED will only operate at a voltage range of 1.8V to 3.3V. Therefore, the LED will stop working if the voltage is reduced to the point below the operating voltage.

Another example of PWM usage can be observed with a DC motor. In order to vary the speed of the motor while maintaining the torque of the motor, we should not deprive the motor of power by lowering the voltage. Instead, we should use PWM to operate the motor.

Timer

In a microcontroller, a timer is a counter function. Within a microcontroller, there may be many clock sources, each of which allows the user to count at a distinct clock range. The speed at which the timer counts at depend on the internal clock of the microcontroller. So, how can we make the timer count in seconds, minutes, and hours? To modify the count rate, we'll have to utilize prescaler and postscaler function. Prescaler and postscaler divides the counting rate by 2^x. In the image below, the numbers under prescaler and postscaler module indicate "original rate" : "decreased rate". The fundamental reason for postscalar's existence is to fine-tune the count rate. There is no method to divide the rate by 7 times since the rate decrease is fixed at 2^x.



You may learn more about flip flop timers in the following link to have a better understanding of how to count time at a certain pace. https://www.electronics-tutorials.ws/counter/count_3.html.

However, functions have been built in Arduino to do the time counting for you. You can get timer counters in milliseconds and microseconds using functions like *millis()* and micros(). The functions *millis()* and *micros()* begin counting when the microcontroller powers up. There are also *delay()* and *delayMicroseconds()* functions that pause your program for a set amount of time.

You may think that the need for a timer is explainable, but what is the rationale for having a function that pauses the application. Software operates according to your instructions, but it is unaware of the hardware characteristics. For example, you want to read the light intensity after pushing the light dependant resistor ahead by 10cm with a motor. Without the pause function, the software will tell the motor to move 10cm and then read the light intensity without waiting for the motor to travel 10cm. This will cause inaccuracy in your data thus it is always important to wait for your hardware to perform its task.

Even though millis() and micros() begin counting as soon as the microcontroller is powered up, time lapse counting can still be done. The example below records the current timer count and performs task for 3 seconds.

```
int StartTime = millis();

while (StartTime – millis() < 3000)

{

        **Performs task**;

}
```

Interrupt and Priority Interrupt

Interrupting the present task is exactly what it sounds like. Interrupt is comparable to a software application's event handler. It pauses all processing in a microcontroller and performs a specific operation before returning to its previous task. So, what's the point about this? Assume that the microcontroller is doing some activity and that new data is being delivered to it. Because it is still executing its job, the microcontroller will be unable to read the incoming data. Therefore, in order for the microcontroller to read the incoming data, the first byte of the data will trigger the interrupt, forcing the microcontroller to read the incoming data first.

The **attachInterrupt(digitalPinToInterrupt(pinNumber), ISR, mode)** method will be used to create an interrupt. With the function, you connect a pin to an Interrupt Service Routine (ISR), a function that will be triggered when the pin is interrupted. The mode determines how the interrupt is triggered. Note that not all pins on the Arduino can be used for interrupt. For additional information on the interrupt function, you may visit the Arduino website. https://www.arduino.cc/reference/en/language/functions/external-interrupts/attachinterrupt/

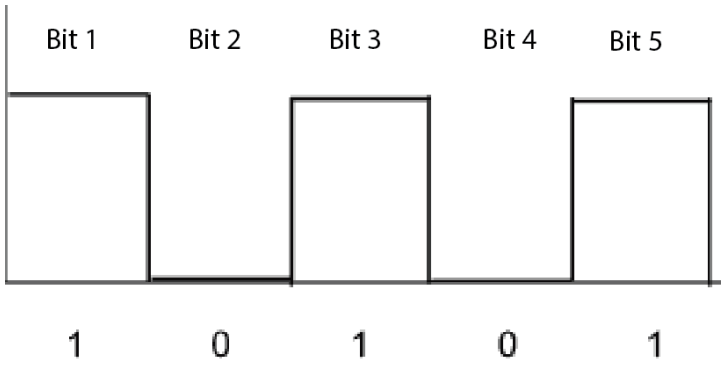
What happens if we have more than one interrupt? Is it possible for an interrupt to interrupt another interrupt? Unless you've expressly written for it, this won't happen in Arduino. This comes down to the next point, priority interrupt. The ISR with the highest priority will always execute first in a priority interrupt. For example, if a higher priority interrupt is activated after a lower priority interrupt, the lower priority ISR will be halted and the higher priority ISR will be performed first.

How can we organize our interrupt such that it behaves like a priority interrupt even though Arduino doesn't allow us to specify the interrupt's priority? We can make use of a flag. A flag is an indication that indicates whether or not an interrupt has been triggered. The variables btn1 and btn2 acts as flags in the example below, and according to the if else statement, the task of pin 2 to be executed first.

```
bool btn1 = false;
bool btn2 = false;

void setup()
{
    attachInterrupt(digitalPinToInterrupt(2), Button1, HIGH);
    attachInterrupt(digitalPinToInterrupt(3), Button2, HIGH);
}

void loop()
{
    if (btn1)
    {
        btn1 = false;
    }
    else if (btn2)
    {
        btn2 = false;
    }

}
void Button1()
{
    btn1 = true;
}
void Button2()
{
    btn2 = true;
}
```

Bits

In earlier subjects, we discussed bits, but what precisely is a bit? As explained in digital pins, a bit is a binary digit ($Binary_2$) that indicates a high or low state. A bit can be either high or low, but not both.



Different terminologies are used when more than one bit is employed.

Bit	1 Bit
Nibble	4 Bit
Byte	8 Bit
Kilobyte (KB)	8000 Bit
Megabyte (MB)	8000000 Bit
Gigabyte (GB)	8000000000 Bit

We know that numbers are made up of integers, decimals, real numbers, conjugates, and so on in the field of mathematics. However, in the digital world, mathematics comprises of more than just the numbers we are familiar with. Decimals are the numbers we learned in mathematics. We do not only have decimals as a numerical representation.

	Types	Base
1 Bits	Binary	2
integers, decimals, real numbers, conjugates, etc	Decimals	10
3 Bits	Octodecimal (0 to 7, total 8 count)	8
4 Bits	Hexadecimal (0 to 15, total 16 count)	16

Decimals, octodecimals and hexadecimal are represented by binary in the following way:

Binary	Decimal	Octodecimal	Hexadecimal
0000_2	0_{10}	00_8	00_{16}
0001_2	1_{10}	01_8	01_{16}
0010_2	2_{10}	02_8	02_{16}
0011_2	3_{10}	03_8	03_{16}
0100_2	4_{10}	04_8	04_{16}
0101_2	5_{10}	05_8	05_{16}
0110_2	6_{10}	06_8	06_{16}
0111_2	7_{10}	07_8	07_{16}
1000_2	8_{10}	10_8	08_{16}
1001_2	9_{10}	11_8	09_{16}
1010_2	10_{10}	12_8	$0A_{16}$
1011_2	11_{10}	13_8	$0B_{16}$
1100_2	12_{10}	14_8	$0C_{16}$
1101_2	13_{10}	15_8	$0D_{16}$
1110_2	14_{10}	16_8	$0E_{16}$
1111_2	15_{10}	17_8	$0F_{16}$

Now that you're familiar with these four numeral systems, always remember to include the number's base in the subscript. This is to guarantee that others read your numbers in the same way you do.

Moving on, why do we have octodecimal and hexadecimal since a computer understands binary and decimals are easier to read? These numbering systems are useful for communicating huge numbers since they convert the numbers into a new base, which reduces the chances of human error as the number grows larger. Take a look at the tables below to help you visualize.

Hexadecimal	16^0 = 0000	16^1 = 000F	16^2 = 00FF	16^3 = 0FFF	16^4 = FFFF
Decimal	1	16	256	4096	65536

Octodecimal	8^0 = 0000	8^1 = 0007	8^2 = 0077	8^3 = 0777	8^4 = 7777
Decimal	1	8	64	512	4096

With this, we can see that it is easier to send “0xFFFF” rather than 65536 or “07777” rather than 4096 as it is easier to type and less prone to human error.

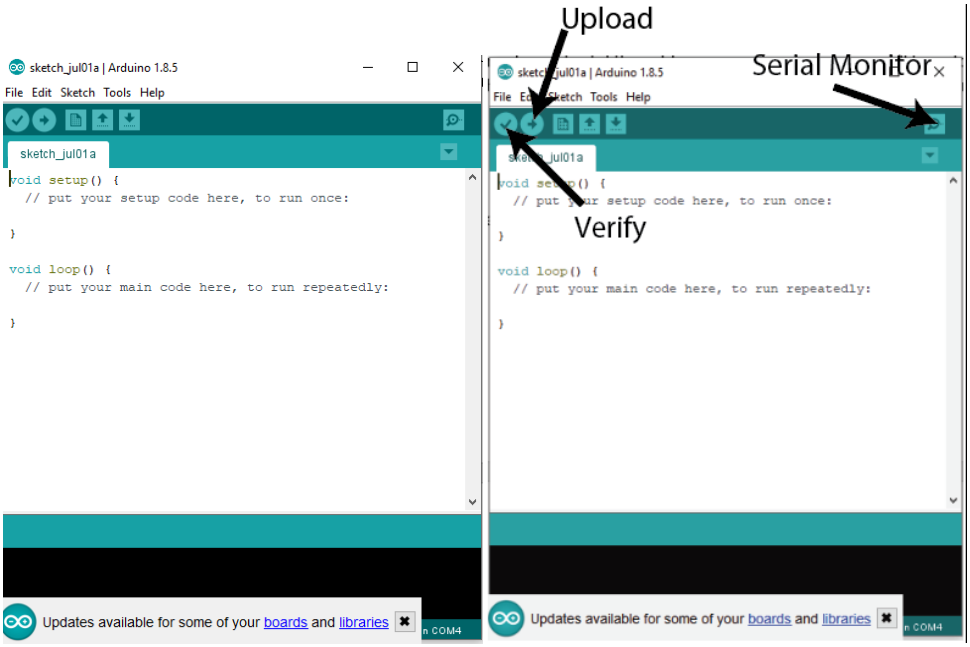
Now, what is 0x and 0 in front of the numbers? A computer does not understand subscript or superscript, thus when sending decimals of different bases in your program, you will have to specify your base to the compiler in another way. The table below shows a common way of writing the codes where “*a*” is the number.

Hex	0x*aaa*
Octo	0*aaa*
Deci	*aaa*
Binary	B*aaa*

Different compiler might have a different way of reading bases.

Arduino Compiler

An Arduino Compiler compiles codes into bits that can be recognized by the Arduino board. Arduino uses C++ to compile into its board. Here are some functions you should take note of to ease the use of using the Arduino compiler.

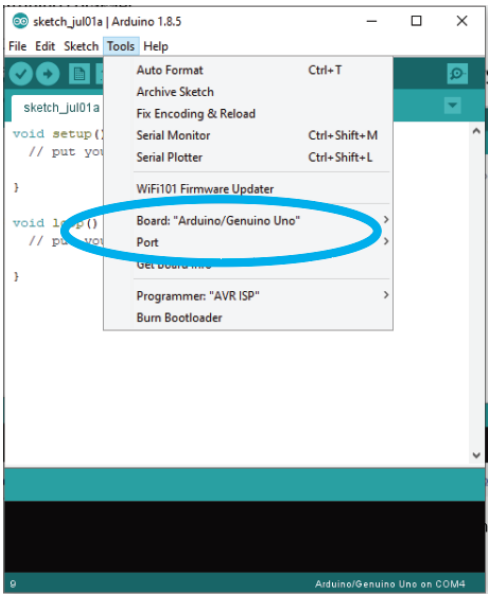


The above Graphical User Interface (GUI) appears on opening.

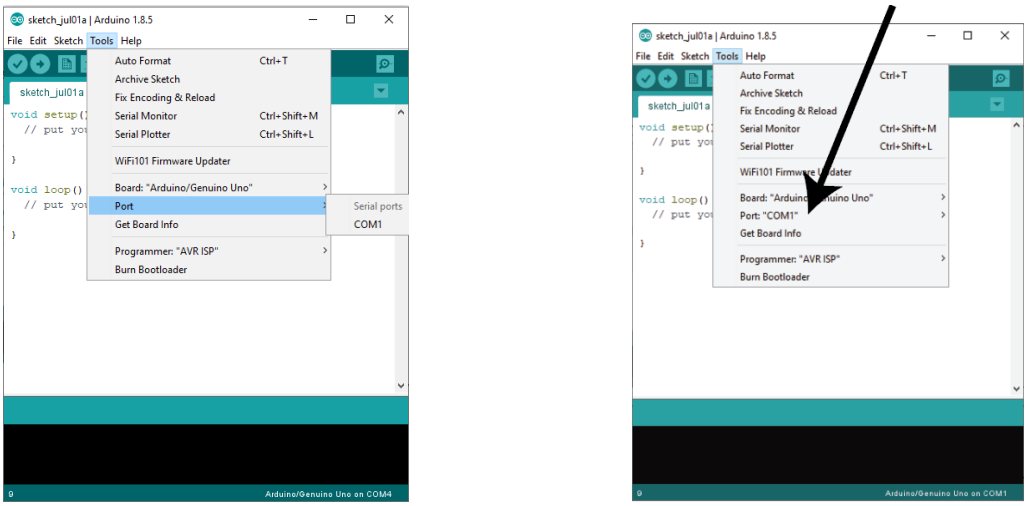
In Arduino, the file that stores your code is called a sketch.

Port and board

Always check the port and the board type before you upload into the board.



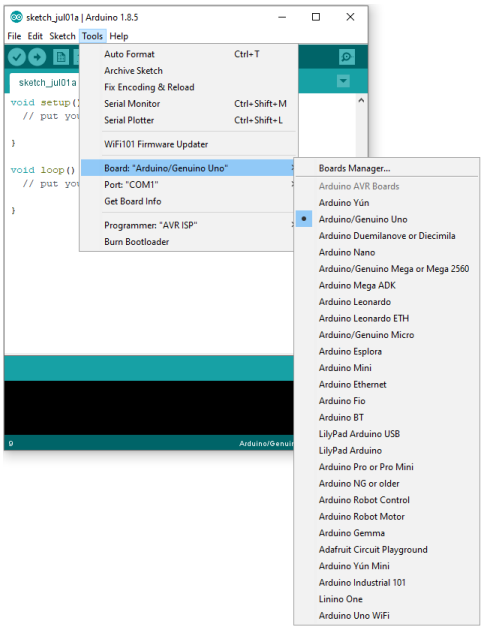
Port



Selected port will appear on the drop-down menu

If the wrong port is selected, an error will appear during uploading that indicates the failure of code upload.

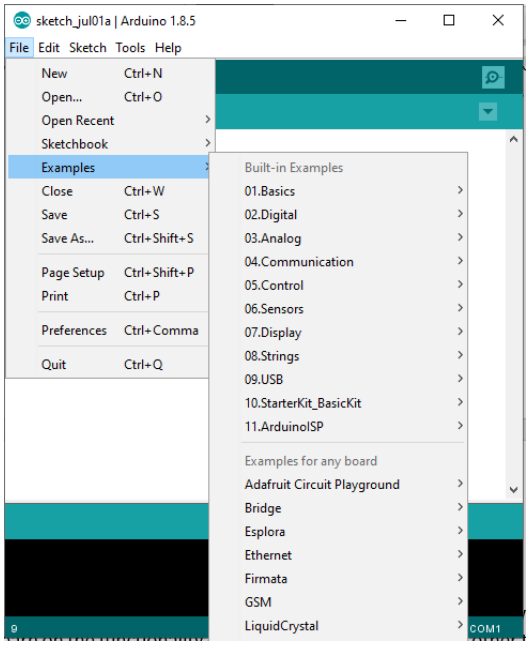
Board



We must choose the proper board type since Arduino offers several models of microcontrollers, and each microcontroller works differently. There may be an error indicating a failure to interact with the board if the selected board is wrong, or the board may not work as intended. The teensy board will appear in this section if teensy is installed as an addon to your Arduino.

Examples

Examples are code written by others to test the component you are working with. It can be used as a guide to learn on the functionality of the component you are using other than reading the documentation.

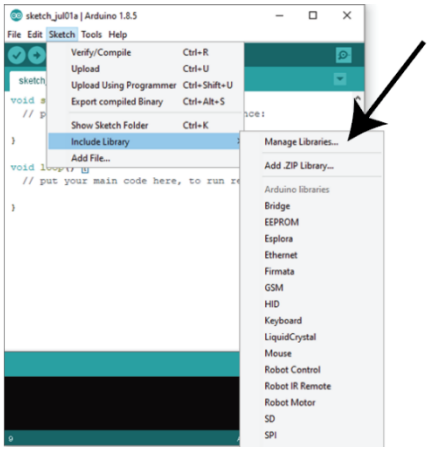


You can find built in examples or examples from installed libraries in the “Example” tab. You can also choose to change the example code directly to fit your implementation.

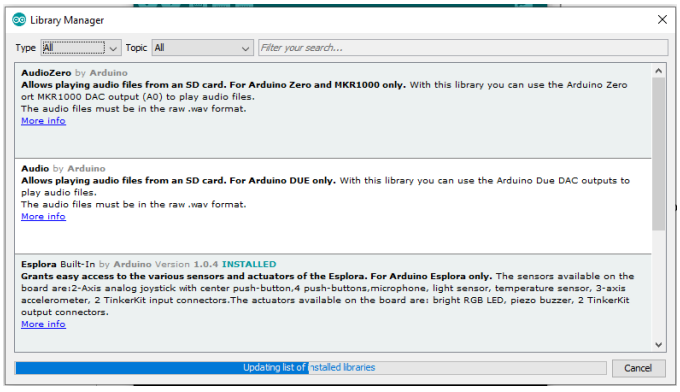
Libraries

Libraries are external files with pre-programmed code to aid you in programming your components. There are three ways to install external libraries.

1) Manage Libraries

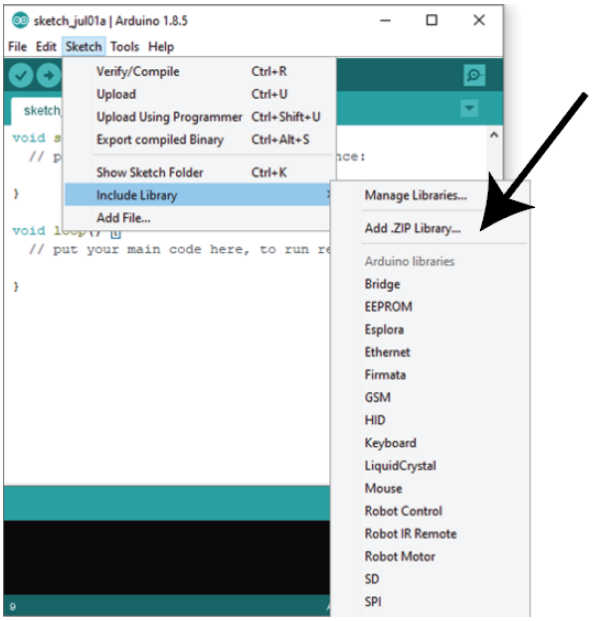


Click on “Manage Libraries”

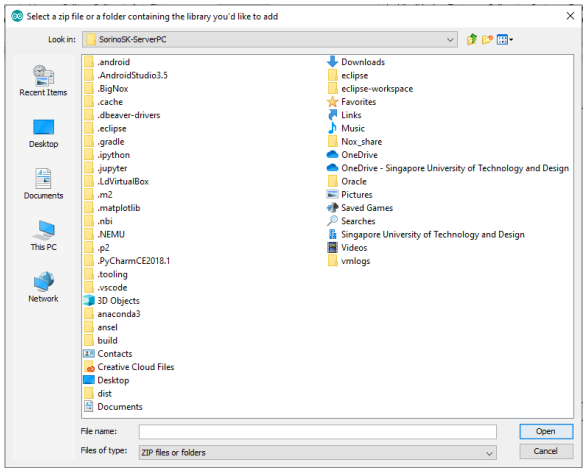


A “Library Manager” will appear. Search for the library you want to use and install.

2) Add Zip Libraries (For downloaded external libraries)

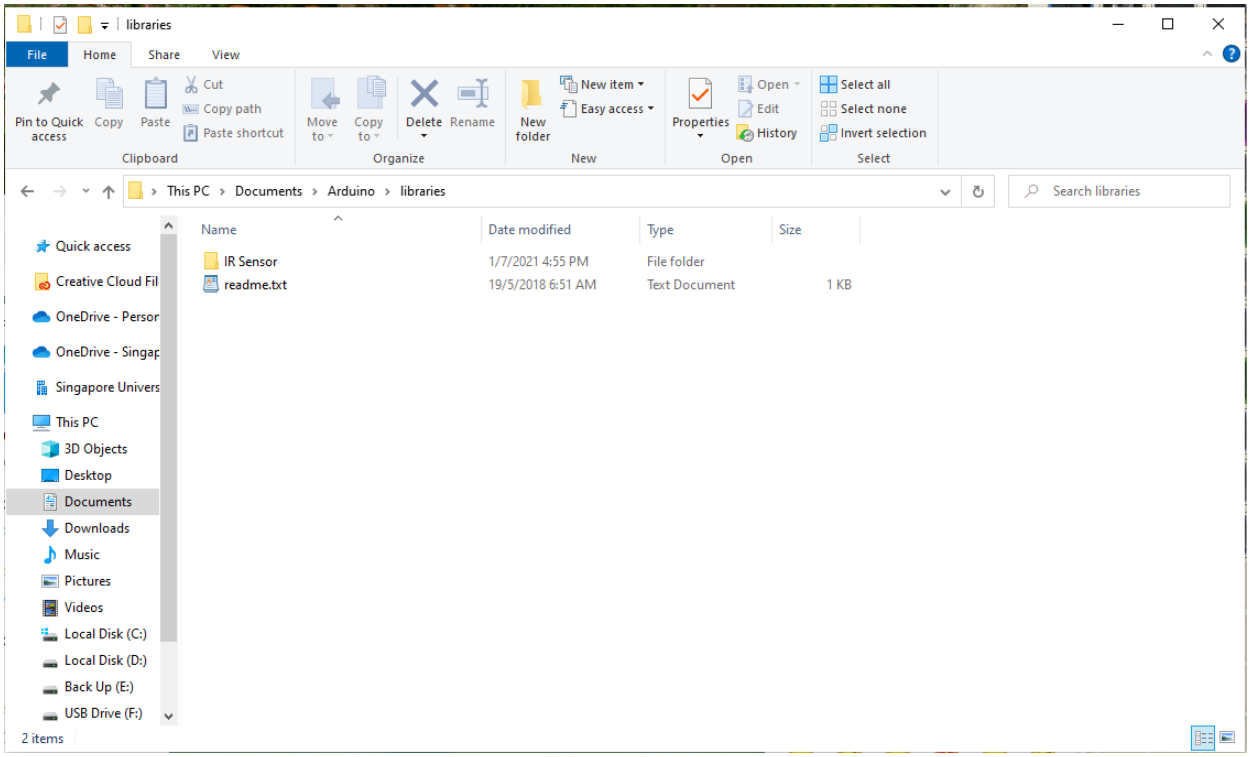


Select “Add .ZIP Library”



A file browser will appear. Search for your downloaded library and press open.

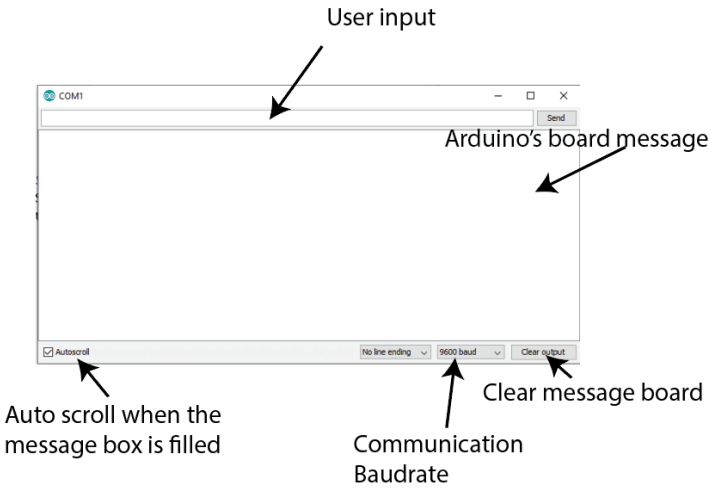
3) Drop the unzipped libraries file into the library folder of Arduino.



Windows	Primary Drive → User → Documents → Arduino → Libraries
Mac	iCloud Drive → Documents → Arduino → Libraries
Linux	~/Downloads/Arduino-*version*-linux*architecture*/Arduino-*version*/libraries Or wherever your download folder is.

Serial Monitor

Serial Monitor is a function made by Arduino to allow the user to communicate with the Arduino board through the computer.



Serial Monitor communicates with the computer through UART. The baud rate is the rate at which data is transferred, and it is used by UART to communicate with devices. Setting various baud rates for each device interacting with each other will result in device misunderstanding. For example, Ali can only listen at a rate of 20 words per second, but Brenda is conversing with Ali at a rate of 40 words per second. Brenda's words will go overlooked by Ali. This misunderstanding will result in erroneous message being read.

Here are some syntaxes you will use to program on your Arduino to communicate with the serial monitor.

Serial.begin()	This function is used to set the baud rate of your Arduino. Serial.begin(your baud rate).
Serial.available()	This function checks for any user input from the serial monitor. If there is data input, Serial.available() will return the number of bytes received.
Serial.read()	This function reads all string input from serial monitor. If there is no user input, it will continuously read "-1".
Serial.print()	Arduino board return values using this function. The returned values will be displayed on the message box.
Serial.println()	Similar to Serial.print() but prints next line, "\n" after the message is displayed.
Serial.write()	Similar to Serial.print(), Serial.write() sends and print in byte while Serial.print() converts the byte into ASCII character.

C++

There are many form of programming languages but two main languages remain popular in the robotics space: C++(C also but since C is a subset of C++ I will subsume it under here) and Python.

Our main focus in this book will be on C++ and we will go through the key concepts needed to start programming on microcontrollers such as Arduino.

Programming is a cool thing as it gives you the power to control your robots at the tip of your fingertips (literally).

This guide is non exhaustive and I would recommend the juniors to come up with something that will be even better than what the three of us, Adeline, Chun Hui and I (Clarence) have done for this guidebook

Good luck on your journey in SOAR!

History of C++

Who doesn't love a history lesson? (just kidding you can skip if you want to jump straight to the content)

Originally called C with classes, C++ was created with the goal of adding object oriented programming into the C language. It included classes, inheritance, inlining, default function arguments and strong type checking in addition to all the features of the C language.

In 1983, the name of the language was changed from C with Classes to C++. For those familiar with programming, the ++ operator is typically an operator used to increment a variable. As such, C++ is typically viewed to be a superset of the C language, and is an object oriented language as compared to a procedural language such as C.

C++ and C are compiled languages while python is considered an interpreted language:

Compiled language: compilers generate machine code from source code before executing the code

Interpreted language: interpreters directly execute the code step by step without pre runtime translation

The key point of this section is understanding the motivations of why C++ was created to improve upon the C language, and also understand the differences between compiled and interpreted language.

Concepts of Programming with C++

Programming gives us the ability to talk to our robots. I'm not sure if you've noticed, but shouting at the robot oftentimes doesn't get it to work. This is not because our robot doesn't love us, but rather it speaks a different language.

In any language, we need a way to define *what things are*, for example homo sapiens have arms legs noses

It is now our turn to define the key elements in our code. We start off this process by giving names to different parts of our code, and this is through variables.

Variables:

A variable is when a programmer gives instructions to the compiler to create storage in a memory location. A good example of variables and memory storage is that of an octopus. Every time you create a variable, it is as if the octopus grabs an object with one of its tentacles. The octopus has limited tentacles and it can only grab one item with each tentacle. This is like our computer memory, assigning variables is like assigning a tentacle to an object, we have limited memory and once the memory is filled, nothing else can be stored unto it until it is cleared.

A variable is created using the following syntax:

```
data_type variable_name;
```

```
data_type variable_name, variable_name, variable_name;
```

Image taken from: C++ Variables (w3schools.in)

Where data_type depends on the type of data your variable is going to store, while the variable name is what you want to call it.

For example, if you want to save a number 100 and want to call it a name: yourGrades, then you would define it in the following way:

```
// int is used to define variables that store integers
int motorSpeed = 100;
```

Other types of data types include: int, char, boolean, float, double, void.

Mainly:

int -> for integers (negative or positive whole numbers)

float -> number with decimals

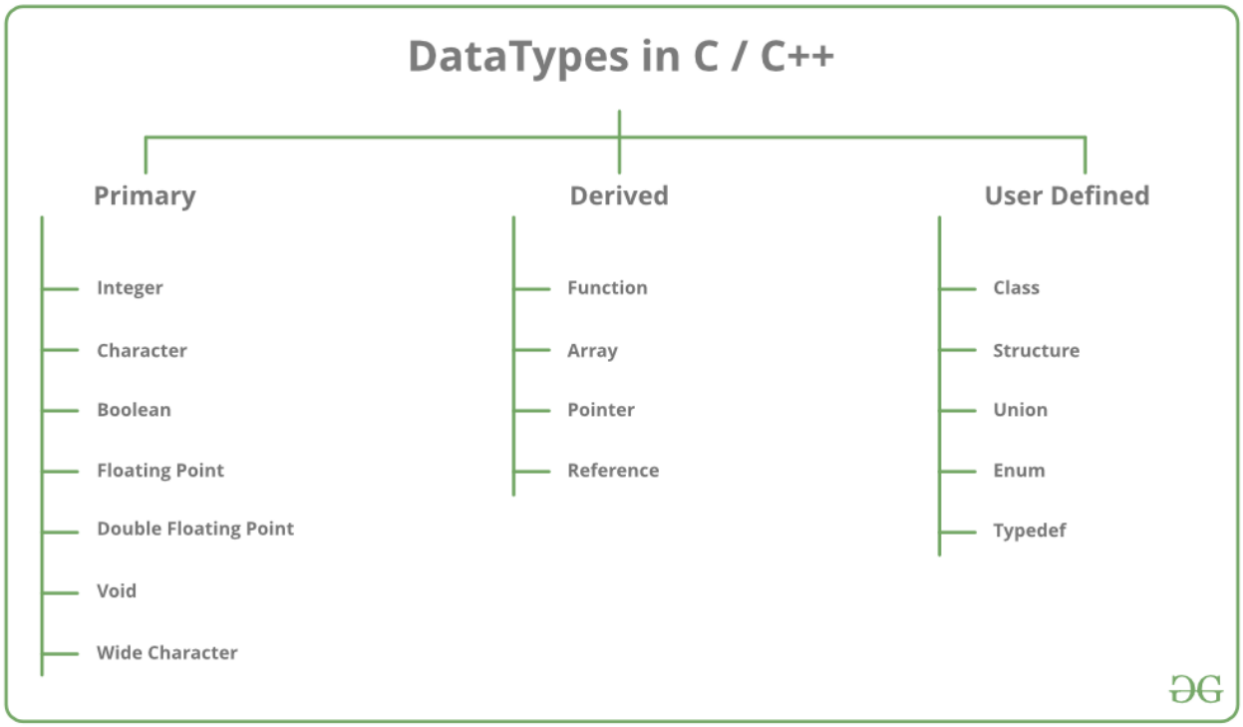
double -> uses double the memory float takes so is able to store more precise numbers

boolean -> true or false value

void -> represents a null or empty state

```
// int is used to define variables that store integers
int motorSpeed = 100;

// bool (short for boolean) is used to define true or false values.
bool isSOARGreat = true;

// char is used to define variables that store letters
char letter = 'a' ;

// float is used to store numbers that store decimals or precision floating point values
// uses 32 bits of memory
float angleOfRotation = 5.0;

// double is used to store numbers that store decimals or double precision floating
// point values, it uses 64 bits of memory
double morePreciseFloat = 5.0;

// void is used to declare that the function does not return anything.
// This is like the void of chasing our ego, we put in alot of effort but return nothing
// More information of this is taught in the function section
void functionName(){

}
```

There are many datatypes in C++ (some covered in this guide, and some not), and the world is your oyster. As you programme more and more, do look into the various options you have for the types of variables you can define. These options are listed in a broad overview in the diagram below.



Taken from: [C++ Data Types - GeeksforGeeks]

(The above link is a great link on the different data types, and the memory storage for each of them, so I'd recommend checking it out for more knowledge)

Comments:

Sometimes you want to leave notes for other programmers or note to yourself as you are writing your code, as such you can write single line comments using double slash (//). The lines preceded by double slashes will not be executed when you compile and run your code. For multi line, comments use /* comment here */

```
//single line comment
/* this is multi line comment
   first line
   second line
   many lines */
```

Declaring arrays:

We have learnt how to declare single variables above, but there may be cases that we need to work with a group of data stored together, and we can do that using arrays.

For the more official definition of an array:

An array in C or C++ is a collection of items stored at contiguous(adjacent, next in sequence) memory locations and elements can be accessed randomly using indices of an array.

Arrays in C++ must store values of the same type, this is unlike lists in python where lists can store a mix of different types, types referring to integers(int), characters (char), float, doubles etc,and is written by enclosing the array of values is with {}:

```
// declare array as:
// type arrayName [arraySize]
// In C++ every element in an array must have the same type
// so no mixing of types is allowed
int arrayOfIntegers[3] = {1, 2, 3};
char arrayOfChars[4] = {'a', 'b', 'c', 'd'};
```

Once you store the information, you can access the values using indexes. In most computer programming languages, we count starting from 0 instead of 1, hence, the first element of a list is 0, second element is 1 and so on ...

```
//store values from the light sensors on our robot as an array
float lightSensorValues = {0.0, 10.0, 20.0, 15.0, 5.0};
//read values from our array
float firstSensor = lightSensorValues[0]; // 0.0
float thirdSensor = lightSensorValues[2]; // 20.0
//change values in our array
lightSensorValues[4] = 40.0; // array becomes {0.0, 10.0, 20.0, 15.0, 40.0};
```

Logic of code:

In the previous sections, we have learnt how to store data in memory in two ways, one is through definition of a single variable and one is through defining multiple of them stored together in an array, We can now to use the data stored to perform actions, and execute logic.

A very important logic statement is the if else statement. The if else statement allows you to run code only when certain conditions have been met.

The basic structure of an if else statement looks like the following:

```
if(boolean_expression) {
   /* statement(s) will execute if the boolean expression is true */
} else {
   /* statement(s) will execute if the boolean expression is false */
}
```

C - if...else statement - Tutorialspoint

To analyse the example code above, we see two sets of curly brackets. Each curly bracket { } encloses a block of code that will execute when the condition defined in the Boolean expression is met (Recap Boolean values stores True False values)

To visualize this more clearly, we use the example where we add an if else statement to turn on a led every time the ultrasound distance is less than 20 and off the led if it is further than 20

```
float distance;
//...
//code above that reads ultrasound sensor values and saves into variable, distance
//...
bool ledOn;

if (distance < 20){
    ledOn = true;
    //code to on led
}else{
    ledOn = false;
    //code to off led
}
```

However, there may be cases where our logic is more complex and thus, we want to be able to add more conditions. For example, a logic statement involving distances may require more complex conditions instead of just two conditions (“smaller than 20” and “larger than or equal to 20” For example, we may not just want a condition between two ultrasound distances (smaller than 20 and larger or equal to 20), we may want have multiple conditions (“smaller than 20”, “between 20 and 40”, “between 40 and 60” etc). We can do this by adding as many elif statements as we like.

Here, we show how to vary the brightness of an LED at different ultrasound distances. Note the usage of conditions here in the if/elif/else statements.

```
float distance;
//...
//code above that reads ultrasound sensor values and saves into variable, distance
//...
int ledBrightness;

if (distance < 20){
    ledBrightness = 0;
}elif(distance < 40){
    ledBrightness = 50;
}elif(distance < 60){
    ledBrightness = 100;
}else{
    ledBrightness = 150;
}
```

Operators:

In the above examples, we arbitrarily mentioned terms like conditions and Booleans for "if else" statements without much explanation on what forms these up. This leads us to talk more about operators, which allows us to come up with True/False statements and perform manipulation of data. There are three main types of operators: Arithmetic (data manipulation), Relational and Logical operators (generate True/False statements)

Arithmetic Operators:

This is typically your mathematical operations, and the list of operators is described below:

1. + **(Addition)** – This operator is used to add two operands.
2. – **(Subtraction)** – Subtract two operands.
3. * **(Multiplication)** – Multiply two operands.
4. / **(Division)** – Divide two operands and gives the quotient as the answer.
5. **% (Modulus operation)** – Find the remains of two integers and gives the remainder after the division.
6. ++ **(Increment)** – Used to increment an operand.
7. — **(Decrement)** – Used to decrement an operand.

Image taken from: [6 Types of Operators in C and C++ | Enhance Your Fundamental Skills [Quiz included] - DataFlair (data-flair.training)]

```
//addition
1 + 1 //2

//subtraction
2 - 1 //1

//Multiplication
2 * 2 // 4

//Division for ints (Get Qoutient)
//no decimal
//return type is an int
10 / 7 //1

//Division for floats (direct division)
//return type is a float
10 / 7 //1.42857

//Modulus (Get Remainder)
//return type is an int
10 % 7 //3

//increment
int a = 5;
++a // 6

//decrement
int b = 5
--b // 4
```

Relational Operators:

Relational Operators lets you compare values between two elements

In the "if else" example, we used the condition distance < 20 to form up our if else condition. The less than operator is one of the many operators that you can use to determine the truthy values. Below is a list of all the operators:

1. == **(Equal to)**– This operator is used to check if both operands are equal.
2. **!= (Not equal to)**– Can check if both operands are not equal.
3. > **(Greater than)**– Can check if the first operand is greater than the second.
4. < **(Less than)-** Can check if the first operand is lesser than the second.
5. >= **(Greater than equal to)**– Check if the first operand is greater than or equal to the second.
6. <= **(Less than equal to)**– Check if the first operand is lesser than or equal to the second

Image taken from: [6 Types of Operators in C and C++ | Enhance Your Fundamental Skills [Quiz included] - DataFlair (data-flair.training)]

```
//Equality operator
1 == 1 //True
1 == 2 //False

"hello" == "hello"//True
"Yes" == "No" //False
"three" == 3 //False

//Not equal operator
1 != 3 //True
1 != 1 //False

//greater than operator
2 > 1  //True
2 > 2  //False
2 > 1  //False
//greater than or equal to operator
2 >= 1 //True
2 >= 2 //True
2 >= 3 //False

//lesser than operator
1 < 2  //True
1 < 1  //False
2 < 1  //False
//lesser than or equal to operator
1 <= 2 //True
1 <= 1 //True
2 <= 1 //False
```

Logical Operators:

Logical operators are used if you want to combine multiple true false validation statements together, for example you may want to check if two conditions are true before returning a value

Logical Operators in C/C++ Includes –

1. **&& (AND)** – It is used to check if both the operands are true.
2. **|| (OR)** – These operators are used to check if at least one of the operand is true.
3. **! (NOT)** – Used to check if the operand is false

Image taken from:[6 Types of Operators in C and C++ | Enhance Your Fundamental Skills [Quiz included] - DataFlair (data-flair.training)]

```
//AND
//&& operator

1==1 && 2==2 //True
1==1 && 2==3 //False

//OR
//|| operator
//The programme executor will check the truth values of statements from left to right
1==1 || 2==2 //True
1==1 || 2==3 //False
// For the below example
// the statement 2==2 will not be checked as the first statement had already returned false
1==3 || 2==2 //False

//NOT
//!
bool isBlue = true;
!isBlue //False
isBlue //True
```

Conditionals:

Logic Operators can be used to determine if a statement is true or false (a condition). We can use conditions to determine which blocks of code we want to execute.

An extension of the “if else” conditional is that of Switch case, which allows us to account for many different cases without having to rely on many if else statements. The advantage of this is cleaner code and we can avoid repeating the == operator

For example, imagine a case of having many different categorical features, rather than using an if else statement, we can use a switch case instead.

```
#include <string>
std::string topic_to_talk_about = "SOAR";

switch ( topic_to_talk_about ) {
case 'Elephants in the room': // False
  // Code is not executed here as topic doesnt match case
  break;
case 'Does the colour Pink really exist?': // False
  // Code is not executed here as topic doesnt match case
  break;
case 'SOAR':
  // Code is executed :D, you have thus entered into the conversation of your soul
  // Hello SOARers
  // You have entered the cult of SOAR
  // We are only looking for one and only one thing in SOAR, passion
  // What is your passion? Are you ready to put in the hours for SOAR?
  // Are you ready to fulfil the promise of commitment when you filled up
  // the sign up sheet to join SOAR
  // If you do decide to show that you are a great person who can keep promises
  // you will find the hope that pandora found in her box
  // You will see the joy when finding nemo was found
  // last but not least you will change what robotics is in SUTD
default:
  // Code is executed here if all the above statement returns false
  break;
}
```

The next concept is the concept of loops.

The two key ideas to loops:

1. There is a block of code I want to execute repeatedly
2. There must be a way to escape the loop

The three main types of loops:

1. For loops
2. While loops
3. Do while loops

We use for loops **when we know the number of iterations we want to execute the loop for** and we use while loops **when we don't know how many runs the loop will take.**

Below is a demonstration of defining for loops:

```cpp
// import the relevant packages to print out our values
#include <iostream>
// setting the namespace to std so we dont have to write std::cout for everything
using namespace std;
// Focus on the loops SOARers O.o, the main point of this part is loops not the cout :D

// basic loop to loop 3 times
for( int a = 1; a < 4; a = a + 1) {
    // code here is executed, we can access the value of a
    cout << "value of a: " << a << endl; // this statement is used to print out values
    // when code finishes execution, it will restart and go into new loop
    // until the condition is met, which in this case is a no longer being smaller than 4
}
// this is printed
// value of a: 1
// value of a: 2
// value of a: 3

// looping using increment operator (be more like a pro)
for (int a = 1; a < 4; a++) {
    cout << "value of a: " << a << endl; // this statement is used to print out values
}
// this is printed
// value of a: 1
// value of a: 2
// value of a: 3

//looping with other increments (or even decrements)
for (int a = 1; a < 4; a += 2) {
    cout << "value of a: " << a << endl; // this statement is used to print out values
}

// this is printed
// value of a: 1
// value of a: 3
```

While loops are useful when we don't know how many iterations our code is going to run for, and hence we depend on conditions to determine when to stop our loop

Below is the basic structure of a while loop

The syntax of a while loop in C++ is −

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

Image taken from [C++ while loop - Tutorialspoint]

Below is an example to illustrate the concept of a while loop, note that in this example we know how many iterations the loop can be run for so we can easily replace it with a for loop, but this is used to illustrate the concept:

```
#include <iostream>
using namespace std;

int main () {
   // Local variable declaration:
   int a = 10;

   // while loop execution
   while( a < 20 ) {
      cout << "value of a: " << a << endl;
      a++;
   }

   return 0;
}
```

To give a more intuitive feel of while loops, we may want to continuously execute a code until a condition is met (sensor hits certain value). We illustrate this concept below where the code will keep printing object is far away until the ultrasound detects a distance that is less than 20.

```
  //while loop

  #include <iostream>
  using namespace std
  int distance;
  // code to read ultrasound distance value and save it to distance

  while (distance > 20) {
      cout << "Object is far away" << endl; //this is a print statement
      // INSERT code to read ultrasound value and update distance variable
      // You will learn this in future sessions
  }

  cout << "loop is exited, distance is near :D and below 20" << endl;
```

Imagine a scenario where the ultrasound distance detected started out below 20, then the code in the while loop would never be executed. If we want to make sure that the code in the while loop is executed at least once, we use a do while loop, where the difference is that the condition of the while loop is checked at the bottom of the loop rather than at the top.

Example of do while loop:

```cpp
#include <iostream>
using namespace std

bool isSUTDent = true;

// we want to say soar is great regardless whether the dood is a SUTDent or not
do {
   cout << "SOAR is greatttttttt wooohoooo" << endl;
}while(isSUTDent)
```

Lastly, under the chapter of loops, you can prematurely break out of loops or skip iterations using break, continue and return

Continue:

Continue allows you to skip the rest of the code in the loop and move on to the next loop

```cpp
#include <iostream>
using namespace std

for (int i = 0; i < 4; i++)
{
    if (i == 1)
    {
        continue
    };
    cout << i << endl; //this is skipped when i=1 as a continue will be called
}
//prints
// 0
// 2
// 3
```

Break:

break allows you to break out of while loop before the condition has been met

```cpp
#include <iostream>
using namespace std

int i = 0;

while (i < 20)
{
    i++;
    cout << i << endl;
    if (i==5) //break at 5 instead of when it reaches 20
    {
        break;
    }
}

//prints
// 0
// 1
// 2
// 3
// 4
// 5
```

Return:

Return works in the same way that break works in that it will exit the loop immediately, however return is used in functions (explained in next section) when you want to give back an output from the function call.

```cpp
#include <iostream>
using namespace std

int functionName()
{
    int i = 0;

    while (i < 20)
    {
        i++;
        cout << i << endl;
        if (i==5) //break at 5 instead of when it reaches 20
        {
            return i;
        }
    }
}

functionName()
//prints
// 0
// 1
// 2
// 3
// 4
// 5
```

Modularising code:

We have learnt how to provide logic to our code, now we want to scale up our code to form larger programmes. It is very common to have code that you want to be repeated many times over the rest of the code. Rather than typing the same commands over and over, we can use functions to wrap a block of code, and when we want to call that set of code, we can call the function using one line.

The basic structure of a function is as such:

The general form of a C++ function definition is as follows −

```
return_type function_name( parameter list ) {
   body of the function
}
```

image taken from: [C++ Functions - Tutorialspoint]

A basic example of a function, void means that the function is meant to only execute the code inside without returning any value back

```cpp
// Create a function
void myFunction() {
  cout << "I just got executed!";
}

int main() {
  myFunction(); // call the function
  return 0;
}

// Outputs "I just got executed!"
```

Image taken from: [C++ Functions (w3schools.com)]

A final example to illustrate a practical use case for this, where we want to create a function that calculates the distance based on the pulse duration received from an ultrasound

```cpp
//Create your function
//float is the expected type of value that the function should return
//calculateDistance is the functionName
//float time is the parameters/ inputs that the function expects

float calculateDistance(float time)
{

    // speed of sound is 340m/s
    // time is in microseconds
    // calculation is ((time in seconds)/2) * speed of sound (in cm)
    // need convert time to seconds so divide by 1000000
    // need convert meters to cm so need multiply by 100
    // calculation is thus time* (340 * 100)/(2*1000000)
    float distance=time*340/20000; // distance in cm
    return distance
}

float distance; //type should match return type of the function
//call as many times as you like
distance = calculateDistance(58.0);
distance = calculateDistance(1000.0);

float duration;
//INSERT code to read sensor value and save in variable duration
distance calculateDistance(duration);
```

Note that the in the function above, we defined the variable distance twice, once in the function and once outside. Why does the variable not conflict? This brings the idea of scope, where the variables in the function are block scoped and the variables defined outside are global variables.

Block scope:

When we work with functions, we are consolidating a set of operations and logic into what is called a code block. This code block is enclosed within the curly brackets of the function we defined. As such, the scope in which the variables exist, (or the space in which the variable is defined) is confined within the code block (or within the curly brackets). This means that the variables we define in the code block is only accessible by the operations and functions that reside in the code block. Other code that lives outside the function will not be able to access the variables defined inside

Global scope:

Any variables we define outside of the functions are global variables. This means that all functions and operations will be able to use these variables.

Below is an example of global and block scoped variables:

```cpp
#include<iostream>
using namespace std;

int global = 1;

void blockScopedFunctionOne()
{
    int global = 2; // this variable is block scoped, inaccessible by outside
    cout << global << endl;
}

void blockScopedFunctionTwo()
{
    int global = 3; // this variable is block scoped, inaccessible by outside
    cout << global << endl;
}

void globalVariableFunction()
{
    cout << global << endl; //variable is able to be accessed from global scope
}

blockScopedFunctionOne(); // prints 2
blockScopedFunctionTwo(); // prints 3
globalVariableFunction(); // prints 1
cout << global << endl; // prints 1
global = 50
blockScopedFunctionOne(); // prints 2
blockScopedFunctionTwo(); // prints 3
globalVariableFunction(); // prints 50
cout << global << endl; //prints 50
```

A more comprehensive guide on what scope is and the different types of scope:

Scope (C++) | Microsoft Docs

Pointers and references (Advanced topic)

Pointers and references are one of the trickier concepts in programming in C++ as they risk falling into issues such as memory leak.

However it is important to understand these concepts as you may want to work with passing parameters in functions or make use of the efficiency and speed that C++ offers.

What is meant by passing parameters in functions is that we want one function to be able to access the variables in another function (which we know is normally not possible due to scoping)

An understanding of how memory works is required before understanding pointers and references. There is two types of memory, the stack and the heap:

Stack: the place in computer memory where the variables are declared and initialized before runtime (runtime is the period of time when the code is executing), most values are stored here

Heap: The heap is the section of computer memory where all variables created or initialized at runtime (values that are computed and not explicitly defined by us), typically values are stored here when pointers, new or malloc is used

There is a few steps of memory allocation from when we write code to execution:

1. Writing the code
2. Compiling the code: our written code is traced line by line is converted to machine code
3. All variable and function definitions are stored in the stack memory
4. Code starts to execute (runtime)
5. Any pointers defined save values into the heap during runtime

How accessing things in memory works:

Like books in a library, we have limited memory slots in our computer, when variables and function values are saved, memory is allocated to store these values (e.g. 2/4 bytes for integers) and an address is assigned to them. Addresses are numbers which are returned to us in hexadecimal format indicating where in our computer memory the information is saved

We can use this calculator to do conversion from hexadecimal to decimals [Hex Calculator.]

Below is an example of a hexadecimal number representing a memory address:

```
Address of example_class object 1 : 0x61ff03
```

References

Before understanding what pointers is, an understanding of references would be helpful. What a reference is that it is. A reference is denoted by an ampersand (&) before the variable. A reference allows you to retrieve the address of the variable.

```
1  // Online C++ compiler to run C++ program online
2  #include <iostream>
3  using namespace std;
4
5  int main() {
6      // Write C++ code here
7      int x = 5;
8      cout << &x;
9  }
```

```
/tmp/nTcgYnwqdo.o
0x7fffe15b8ba4
```

References are also useful as it allows you to give different aliases(or different variable names) to the same information in memory. In the example below, we are now able to use x and y to refer to both the value 5 in memory (incrementing y by one increments the value of both y and x to 6)

```cpp
// Online C++ compiler to run C++ program online
#include <iostream>
using namespace std;

int main() {
    // Write C++ code here
    int x = 5;
    int &y = x;
    y ++;
    cout << y << endl;
    cout << x;
}
```

```
/tmp/nTcgYnwqdo.o
6
6
```

Pointers

The difference between a pointer and a variable is that a variable directly stores the information that it holds and hence a variable's memory address is where the information is stored. However, a pointer, does not directly store the value in its memory address, rather, it is a **variable that stores the memory address of another variable.** A pointer can be identified with an asterisk (*) behind the variable

For example,

```cpp
// Online C++ compiler to run C++ program online
#include <iostream>
using namespace std;

int main() {
    // Write C++ code here
    int x = 5;
    int *p = &x;

    cout << p << endl;
    cout << *p;
}
```

```
/tmp/nTcgYnwqdo.o
0x7ffd0ef190bc
5
```

In the above example, we see three things:

1. We initialised a pointer using an asterisk (*)
2. Printing p gives us the memory address of x (remember a pointer is a variable that stores the memory address of another variable)
3. Printing *p gives back the value at the memory address it is storing

There are 3 main types of problems when using pointers (so use with caution):

1. Uninitialised pointer: you declare the pointer variable but you dont assign any address or value to it
2. Memory leak: when you reassign the pointer without deleting the previous stored value in memory
3. Dangling pointer: two pointers assigned to the same address in memory and one of the pointers is deleted, causing the first pointer to now point to nothing in memory.

Parameter passing in functions

With knowledge of references and pointers, we can now understand the three ways in which we can pass variables into functions:

1. Passing parameters by variables (normal way)
2. Passing parameters by addresses
3. Passing parameters by reference

Example 1 (Passing parameters by variables(normal way)):

```cpp
// Online C++ compiler to run C++ program online
#include <iostream>
using namespace std;

// normal way of defining and calling functions
int addition(int a, int b)
{
    cout << "a address:" << &a <<  ", b address:" << &b << endl;
    return a+b;
}

int main() {
    // Write C++ code here
    int x=5, y=10;
    cout << "x address:" << &x <<  ", y address:" << &y << endl;
    int sum = addition(x, y);
    cout << sum;
}
```

```
/tmp/nTcgYnwqdo.o
x address:0x7ffd0e61122c, y address:0x7ffd0e611230
a address:0x7ffd0e61120c, b address:0x7ffd0e611208
15
```

Looking at the example above, we see that the addresses are different, which means that when the function is called, the values of x and y are saved into new variables called a and b, equivalent of saying int a = x and int b = y.

Example 2 (Passing parameters by addresses):

```cpp
// Online C++ compiler to run C++ program online
#include <iostream>
using namespace std;

// normal way of defining and calling functions
int addition(int *a, int *b)
{
    cout << endl;
    cout << "pointer a address:" << &a <<  ", pointer b address:" << &b << endl;
    cout << "a address:" << a <<  ", b address:" << b << endl << endl;
    return *a+*b;
}

int main() {
    // Write C++ code here
    int x=5, y=10;
    cout << "x address:" << &x <<  ", y address:" << &y << endl;
    int sum = addition(&x, &y);
    cout << sum;
}
```

```
/tmp/nTcgYnwqdo.o
x address:0x7ffd586d6ccc, y address:0x7ffd586d6cd0

pointer a address:0x7ffd586d6ca8, pointer b address:0x7ffd586d6ca0
a address:0x7ffd586d6ccc, b address:0x7ffd586d6cd0

15
```

In this example, we pass in the addresses of x and y into the function instead of their actual values. As such in our function definition we have to define pointers, so all the variables a and b are changed to pointers instead with an asterisk behind to extract the value from the pointer. In the print statements, you can see that the pointers a and b have their own addresses, however, the value that they store is the addresses of x and y respectively.

Example 3 (Passing parameters by reference):

```cpp
// Online C++ compiler to run C++ program online
#include <iostream>
using namespace std;

// normal way of defining and calling functions
int addition(int &a, int &b)
{
    cout << "a address:" << &a <<  ", b address:" << &b << endl;
    return a+b;
}

int main() {
    // Write C++ code here
    int x=5, y=10;
    cout << "x address:" << &x <<  ", y address:" << &y << endl;
    int sum = addition(x, y);
    cout << sum;
}
```

```
/tmp/nTcgYnwqdo.o
x address:0x7ffeb735377c, y address:0x7ffeb7353780
a address:0x7ffeb735377c, b address:0x7ffeb7353780
15
```

The implementation of this example is similar to that of example one, however, the difference here is that we are using references instead, which means that if we were to change the values of a and b inside the function, the changes in values will also be reflected outside the function in x and y as a and b are no longer separate variables, but rather aliases to x and y. We can see this by seeing that the addresses of x and a are now the same and similarly for the addresses of y and b.

Conclusion of C++ section

This section introduces most of the fundamentals required to write C++ programmes, which should be sufficient for you to work with microcontrollers for programming after learning how to work with pinouts. Cheers, and as always … don't forget your semicolons ;). UwU

Components

Now that we know how the Arduino roughly functions, we can make use of physics concepts to help us detect certain environmental variables precisely (for research/for certain projects), or to control motion, or anything as long as it emits/receives digital signals in the form of voltage. Here, I introduce some of the most useful components in the hopes that they can help specific projects. What I really want you to take away, however, is the way I dissect each of these components and where to obtain the information.

Here is a list of things you can try when you face an unknown component. This is more common than you think, especially when working in a shared lab (where people just throw things around).

- Usually, components will have a serial number/model number on it. Look out for the model number and google it for the datasheet. The datasheet will consist of a bunch of helpful information such as its dimensions (to make your housings), the operating voltage, as well as pinout information which helps you identify what to connect to the component.
- Any microcontroller with GPIO (general purpose input/output) pins typically support 3.3V or 5V input/output, so the component must have a operating voltage equal to that of the microcontroller. Eg. Arduino generates or receive up to 5V in all of its input/output pin, thus we have to ensure that the voltage is in the correct range to prevent burning some stuff.
    - If the voltage is insufficient or overflow for your component to operate, you can use a transistor or a logic level converter to step up/down the voltage.
    - This may not be an obvious point but sometimes people forget to put resistors in series with your LED, so once you turn on the LED, you have a high chance of being redirected to gg.com.
- If the breakout board doesn't have any pins, try soldering header pins onto it or look for connector heads where you can plug and play.
- If all else fails, check documentation on that project (check the purchase lists) or ask anyone else.

And here is a list of things to note when connecting pins.

- VCC usually means 5V (but still good to double check its datasheet).
- 3V3 usually means 3.3V
- GND is ground
    - Make sure everything is on common ground.
        - make sure all the grounds pins in a whole system are connected.
        - If two system connected to each other does not common ground, signals or current will not be able to flow between both system.
- For certain protocols, they will require certain special pins.
    - Eg. SCL, SDA are mostly used for I2C
    - Warning: Don't touch pins 0 and 1 on the Arduino Uno as they're used for Tx/Rx
    - Don't use communication pins for other stuff. Examples of communication pins are Tx/Rx pins.
        - Unless you are have to use Tx/Rx pins for multi-communication purposes, you may look into software serial.
- Make sure your circuit has a resistance. Don't short anything.
    - Always check your circuitry in such that there is no lone wire connected from VCC to GND.

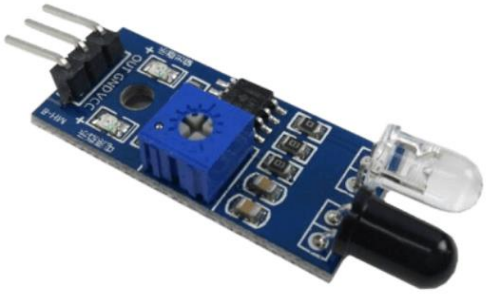
    - If your power supply does not have a short circuit protection function, you may break your components.
- If anything is not working, check for voltage or resistance using a digital multi-meter. If everything is connected well, then check for the fault of the component by swapping it out if you can, or use another board.
- If you don't have enough pins on the Arduino to control all of your components, consider three things: use another microcontroller with more pins (Arduino Mega), use a shield, or multiplexing (use the IC).

With that, let's begin analysing some of the more common components.

Sensors

a. Infrared

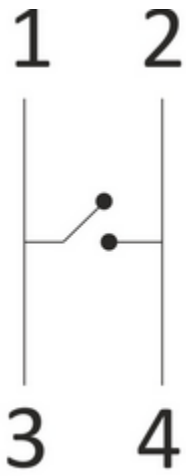
Infrared sensors detect infrared waves, which is a type of EM wave. They travel very fast (speed of light), so this is pretty good for normal usage that doesn't require extreme precision (e.g. metal reflects more light than non-reflective material). However, do check the operating range of the IR sensor.

Active	Passive
	
One of the bulbs is an emitter, the other a receiver. The distance is calculated by reading the remaining energy reflected into the receiver.	This whole thing is just a receiver that detect heat by reading infrared emitted from objects nearby. Note, not everything might emit an infrared signal strong enough to be detected by this thing. This component is a motion sensor PIR module, so it should only pick up changes in infrared signals.
General obstacle detection	Motion sensor

b. *Ultrasound*

Similar to the active IR sensor, one of these parts emit sound pulses while the other receives the signal. The time taken for the pulse to travel back and forth is recorded to calculate distance the sensor is away from the obstacle. (Use “ping” in Arduino Examples - Sensors to test this if unsure.)

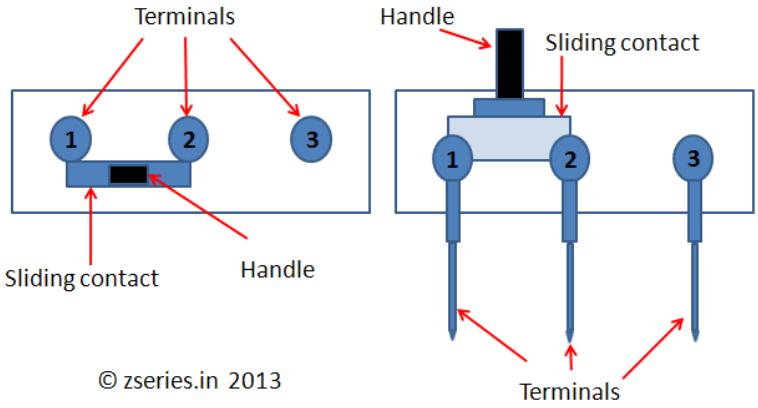
c. *Buttons*



Shown above is a standard push button commonly used on breadboards. Adjacent legs should be soldered for the button to work as a button when pushed – else, the button will not work as expected as connecting to pin 1 and 3 short circuit the circuitry.

Note: Always check the schematics of the specific button online first, some are wired differently which will affect the soldered pins. There are two different types of buttons: momentary switches and latching switches. Both works the same but are used for different purposes.

d. Switches



This is a type of switch – a slide switch. The switch connects two adjacent pins depending on its current position. This is useful as a power or a signal switch as it can continually keep a circuit closed for its compact size. There are a few types of switches:

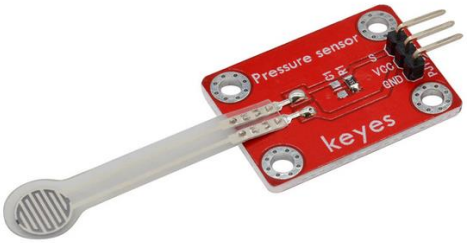
1) Toggle switches
2) Switches
3) Microswitches
4) Limit switches
5) Etc …

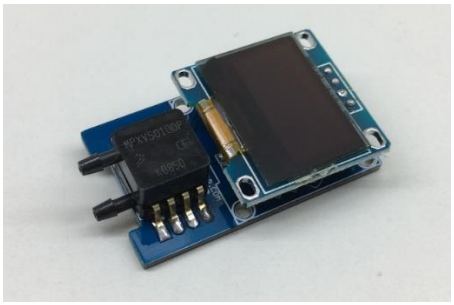
Toggle switches and limit switches, for example, perform the same job as the ones illustrated in the figure above. The phrases Normally Closed (NC) and Normally Open (NO) are used and this may provide more functionality to your projects by using these options.

e. Relays

When we talk about switches in microcontroller, we should not miss out on relays. A relay is an electrical switch where you turn it on or off programmatically. It allows high voltage and current to flow through it unlike a transistor. For example, I would like to turn my 230VAC fan on when I am home. A motion sensor can be used to detect my presence in my home and send a signal to the microcontroller. The microcontroller will then send a high (Binary 1) output to the relay, which will turn it on. The relay will then allow current to flow through it.

f. Pressure/Force sensor





Pressure sensor pads like the above breakout boards are useful if a touchscreen is not necessary. This type of sensor uses the change of resistance in the wires to detect the presence of any force exerted onto the pad.

Another type of pressure sensor would be the barometric sensor which senses differences in air pressure. This would be useful in things like leakage detectors where any slight difference in air pressure could mean a potentially hazardous leak.

However, if you are looking for a sensor which could measure the weight of an object, the strain gauge sensor is what you are looking for.

g. Indicators

Essentially things that give visual/audio feedback, this is very helpful for making things more interactive. If you have something that runs on its own, then you need indicators to ease your troubleshooting (eg. Red LED to show that it's turned on)

1) LED (Light-emitting diode, current only flows in one direction and it will not light up if the direction of current flow is wrong)

Individual LEDs: Long leg is anode, connect that to the + terminal of the battery.

LED strip: Make sure not to buy the RGB ends (the kind with 5V, then ground the R/G/B lead that you want to light). Buy the one with 5V, DIN, GND instead and use Neopixel Arduino library, or some other led library that can control LED strips.

2) Buzzer/Speaker

Buzzer	Speaker
	
A buzzer is usually a self-oscillating device that moves a diaphragm repetitively to produce a sound. There are 2 types of buzzers, passive, and active buzzer. Both produce a single sound when supplied with a constant voltage. However, if you would like to change the tone, active buzzer will requires an oscillating DC voltage while a passive requires an oscillating AC voltage.	A speaker has a moving coil connected to a diaphragm and a magnet inside the coil. It requires an external oscillator to produce different tone of sound. Any oscillating signal applied to the coil moves the diaphragm thus changing the frequency of the sound.

3) Servo, Direct Current motors, Stepper motors

In general, motors are very useful for controlling moving parts in a robot. They are commonly found in wheels, joints, arm mechanics, gripper mechanics, and generally anything that involves moving a certain piece of hardware. The specifications of each will therefore determine which to use to accomplish a certain objective – for instance, DC motors are more likely to be used for cooling fans as the angle of rotation is unlimited and it has a high RPM.

	Servo	DC Motor	Stepper Motor
Angle of rotation	1. Positional rotation: Limited (usually $\frac{90}{180}$ / $360°$) 2. Continuous rotation: Unlimited angle and rotate according to the angle controlled by the driver.	Unlimited	Unlimited (each step is $1.8°$
Wiring	Power, ground, control	Power, ground	Controlled using microcontroller Typically, 4 pins: Enable1, Enable2, Direction1, Direction2
Controller board	Servo motor controller	Electronic speed controller	Microcontroller (eg. Arduino/Raspberry Pi)
Rotations per minute (RPM)	Mid	High	Not as high (~1500)
Motor position	Determined using potentiometer or driver board	Determined using encoder	Determined through steps recording by the driver board.
Accuracy Ranking	2	3	1

i. Servos

Servos have a certain limit to their range of motion but there are some continuous servos as well. They're often used in foam planes, smaller robotic arms, and anything that requires small and precise movements.

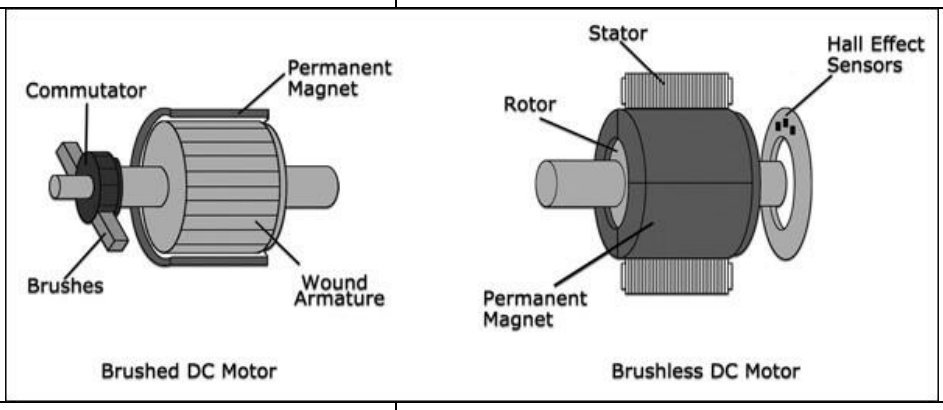
	Reason for factor
Weight	For weight distribution on the whole system/ account for torque on other motors
Torque	Check if motor/servo can handle load
Speed	Especially for speed sensitive things
Dimensions	To ensure housings are snug
Operating Voltage	Power requirements to prevent overloading/undervoltage

Here are the two commonly used servos in SOAR as they are relatively cheap and usually serves the purpose. Feel free to look for other servos that might better fit your use case.

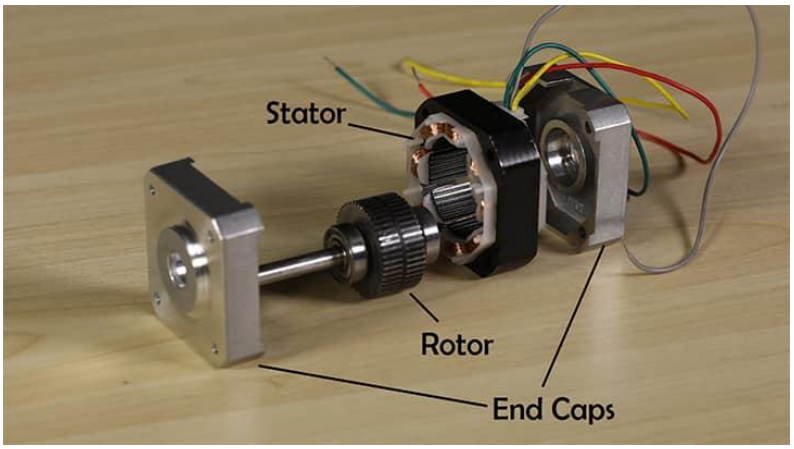
SG90 datasheet: http://www.ee.ic.ac.uk/pcheung/teaching/DE1_EE/stores/sg90_datasheet.pdf
MG996 datasheet: https://www.electronicoscaldas.com/datasheet/MG996R_Tower-Pro.pdf

ii. DC Motors

Motors are commonly used for RC cars (high RPM), or anything that requires high torque (with gearbox).

<table>
<tr><th>Brushed</th><th>Brushless</th></tr>
<tr><td colspan="2"> 
</td></tr>
<tr><td>Typically has carbon brushes</td><td>Use magnets to generate power</td></tr>
<tr><td>Low overall construction costs;
Can often be rebuilt to extend life;
Simple and inexpensive controller;
Controller not needed for fixed speed;
Ideal for extreme operating environments.</td><td>Less overall maintenance due to lack of brushes;
Operates effectively at all speeds with rated load;
High efficiency and high output power to size ratio;
Reduced size with far superior thermal characteristics;
Higher speed range and lower electric noise generation.</td></tr>
</table>

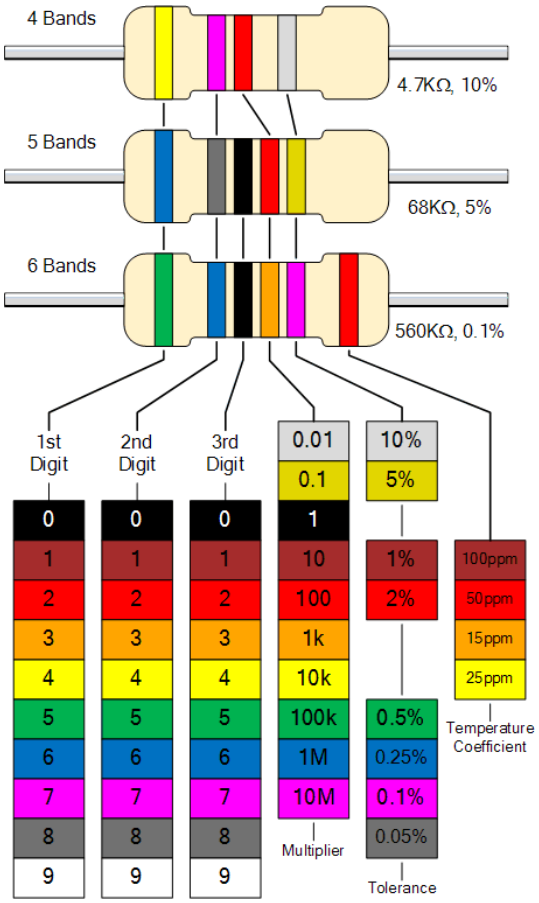
iii. Stepper Motors



Stepper motors are a type of brushless DC motors. Here, you'll find more information on stepper motors. They are commonly used in 3D printing and in precision-based projects, so if you hear a grinding sound, there's a high chance that something might be wrong with your stepper motor.

2. Resistors

There are two types of colour banding as shown in the chart below.

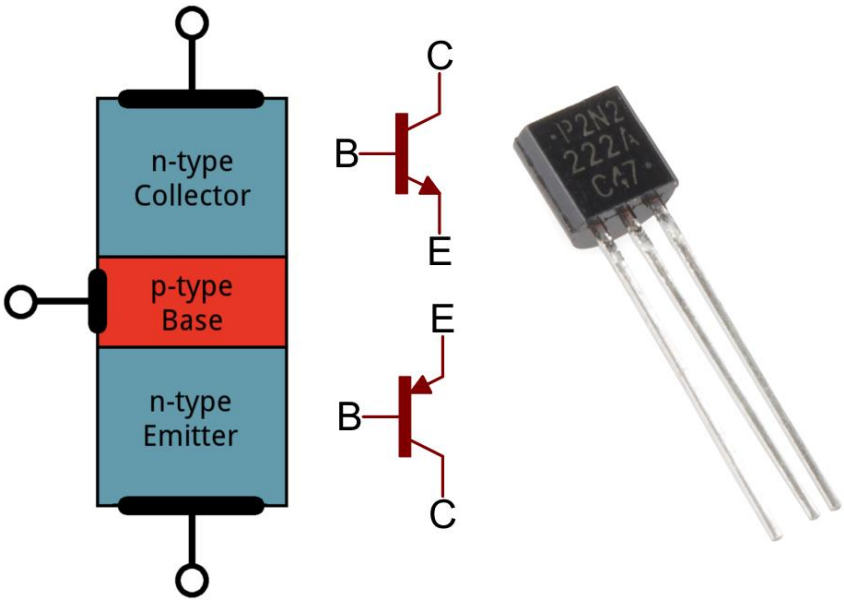


Resistance is the description of the disruption in current flow. Materials, shape and sizes or conductors contribute to the behaviour of current flow. I'll give you a quick example to help you understand. A concrete route is easier to walk on than a sandy path, and you can walk quicker on a wider path than on a very narrow one. You will be learning this in your upcoming modules in term 2.

These days, you can purchase any resistor you need and combine them in series or parallel to get the resistance you need. However, if you are making repairs in a remote location, you may not have access to all of the resources necessary to get the exact resistor you want. As an example, if you need a 100kΩ resistor but only have a 10kΩ resistor, you will have to scrape the 10kΩ resistor to thin it till you get a 100kΩ resistance since the thinner the current flow is, the higher the resistance.

So why disrupt the flow of the current? Isn't it wasting electricity to heat? As previously stated, various components have different operating voltages. As we all know, Voltage = Current x Resistance, thus a resistor can be used to control voltage flow. For instance, certain LEDs require 3.7V to function, while the nearest common power source can only deliver 3.3V or 5V. To fix the problem, we'll need to connect a resistor to the LED in series. Another usage of a resistor is to use it as a voltage divider.

3. Transistors/FETS (field effect transistor)



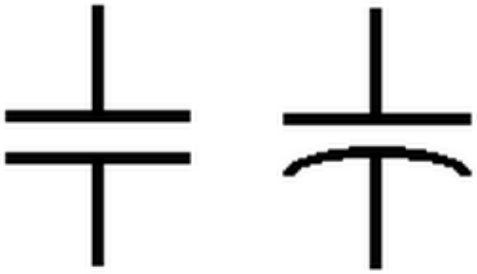
There are 2 types of transistors, NPN and PNP transistor. (N = Negative, P = Positive). In the above diagram, the 1st schematic at the top in red shows a NPN transistor whereas the 2nd schematic at the bottom in red shows a PNP transistor.

Transistors mainly consist of 3 pins, Collector, Base, and Emitter. Power flows from the collector to emitter or emitter to the collector. A transistor allows us to control voltages flowing to a component. To control the amount of voltage flowing between the emitter and the collector, we will have to control the current flowing into the base.

1) This can be a programmable switch or an amplifier. You can check out how it works here.
2) You can use this to increase the current of a circuit by providing an external power source.
    i. This will be very helpful for lasers, powering a lot of LEDs, and for many other things that require current higher than what the Arduino can provide.

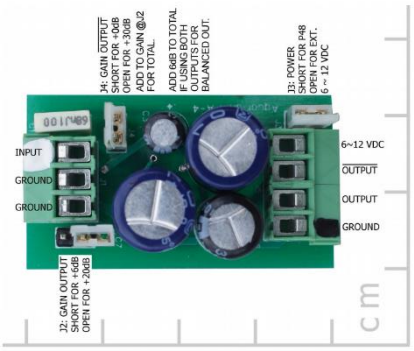
4. Capacitors

There are two types of capacitors, ceramic and electrolyte capacitor. Both types of capacitors hold electrical charges; however, electrolyte capacitors are polarized. In a ceramic capacitor, current can flow in both directions, whereas current can only flow in one direction in an electrolyte capacitor (anode to cathode).



Non-Polarised vs Polarised

You'll be learning more about how they work in ~term 2. A capacitor is something that stores electrical energy, and they're commonly used in amplifiers. Here's a picture of an audio amplifier that SAUVC uses to amplify hydrophone signals using a system of capacitors, I found this picture by searching for the pinout diagram for this specific model.

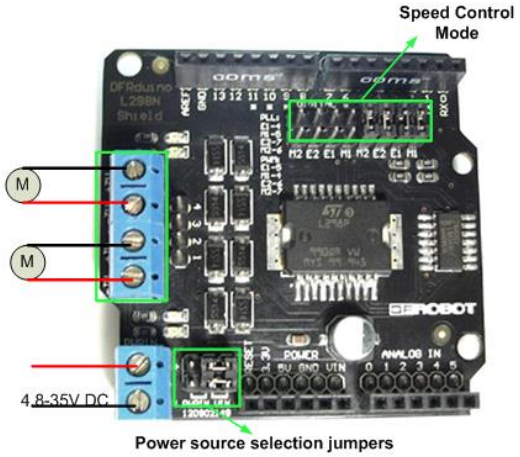


As you can see, the blue and black circles are actually capacitors. These components will result in a magnetic field generated around them, which may result in noise (in amplifers, this is very bad because you already have a small signal to begin with, but most commercially available amps should be able to function without the need for excessive shielding else there's no point selling them). Proper shielding will have to be done. We used aluminium foil when we were testing this right next to a power source, it seems to work so you can try that.

5. Modular boards
    1) Arduino Shield – PCB board added to Arduino to simplify your work
        i. Motor shield

This is the official motor shield for Arduino Uno. Note the difference between a motor driver and a motor shield; a motor driver is a chip that drives motors while a motor shield is a circuit board with connections on it that contains a motor driver chip. A motor shield simplifies the connections as the shield generally only requires the power input, ground, and signal cables (differs per shield) whereas a motor driver offers more flexibility but if you aren't sure of what you're doing, there's a higher chance of frying it.

Sample motor shield using L298N chip:

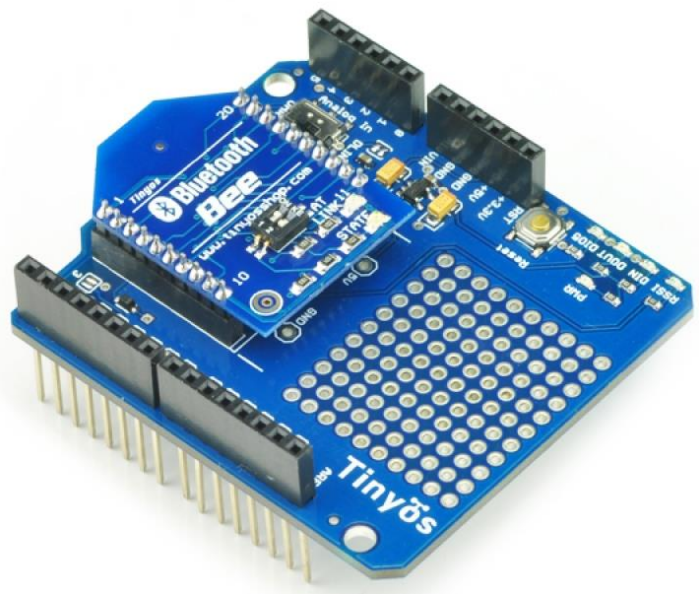


Sample L298N chip:

ii. Wifi/Bluetooth shield

There's a board called ESP32 with 30 or 36 pins that has onboard wifi and Bluetooth chips. If wifi and Bluetooth capabilities are your main considerations, consider using this instead.

Else, there are shields available that give you Bluetooth capabilities like these:



As for wifi, we've been using these (similar to this, please check project documentation for exact model):

Note: Please check the compatibility of the chip with your operating system (including version number), else you'll be spending 2 days in front of various monitors just debugging to get wifi.

iii. Custom PCBs

If you want to make a lot of connections in a tight space, or for better wire management, you can make your own Printed Circuit Boards. Easyeda is what someone recommended to me and it looks pretty good, or you can also use Eagle from Autodesk.

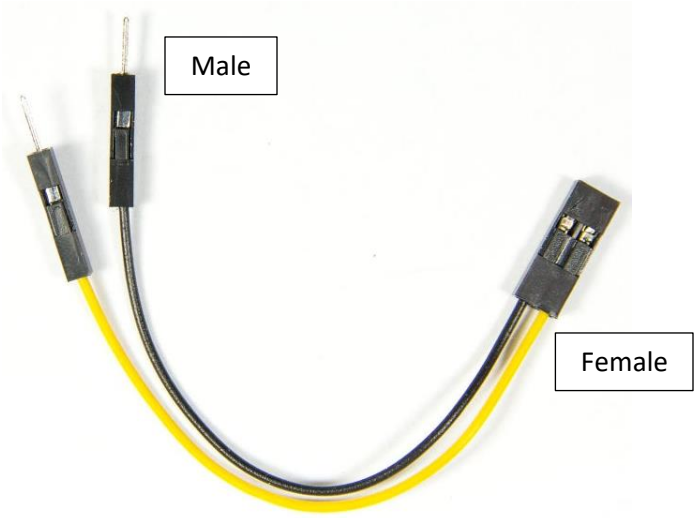
2) Board connectors

i. Terminals

There are many types of terminals that all serve one purpose – they are insulated blocks that secure two or more wires together. If you see metal pins below the terminal block, they can be soldered on (to PCBs) to allow thicker wires to be connected to that pin.

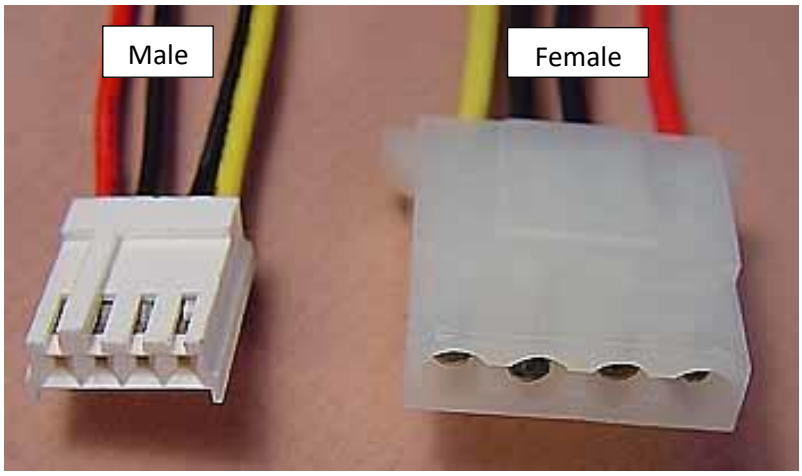
ii. Dupont connectors

Usually used for thinner wires, you can customize the length of the wire with the crimps and the housing as long as you have the crimping tool.

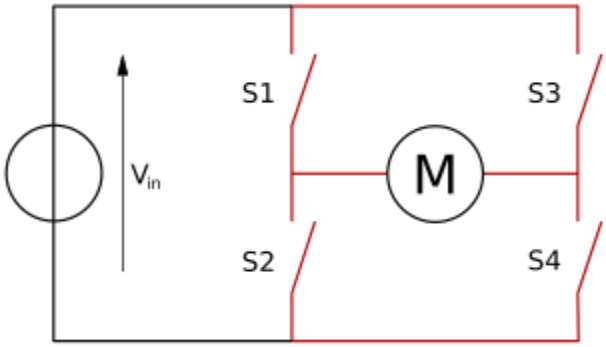


iii. Molex

Note: These look really similar to Grove connectors, so check the component. If it's not a component stating Grove, or Seeed, then high chance it's Molex. Just bring the component with you when you buy other parts/connectors.



3) H-bridge



https://learn.digilentinc.com/Documents/325

i. Motors have NO polarity
ii. To make them spin in a specific direction, need to manage the direction of current flow
iii. If additional power is required, H-bridge can allow you to connect additional power source
iv. It *is* a transistor so you don't need additional transistors