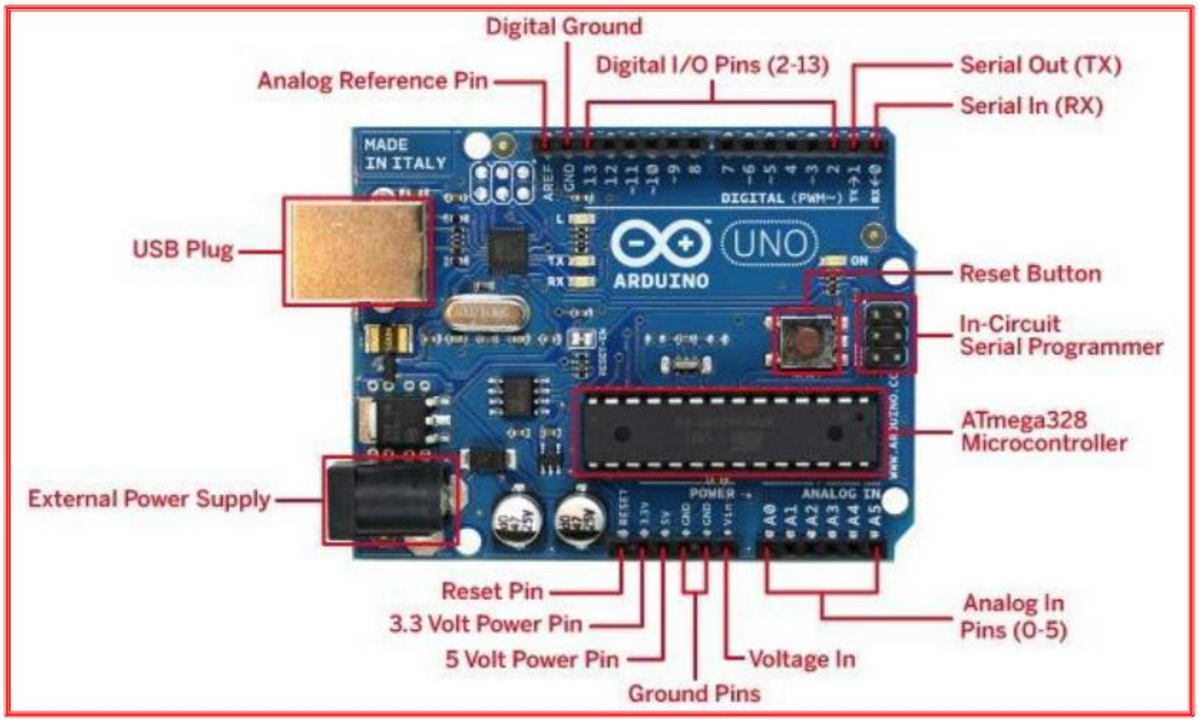
Al-Mustaqbal University
Department of Medical Instrumentation Techniques Engineering
Class: four
Subject: Advanced logic design
Lecturer: Dr. Zahraa hashim kareem
Lecture- 1: The Arduino environment

The Arduino environment.

Arduino is used in many educational programs around the world, particularly by designers and artists who want to easily create prototypes but do not need a deep understanding of the technical details behind their creations. Because it is designed to be used by nontechnical people, the software includes plenty of example code to demonstrate how to use the Arduino board s various facilities. Though it is easy to use, Arduino s underlying hardware works at the same level of sophistication that engineers employ to build embedded devices.

People already working with microcontrollers are also attracted to Arduino because of its agile development capabilities and its facility for quick implementation of ideas. Arduino is best known for its hardware, but you also need software to program that hardware. Both the hardware and the software are called Arduino.



The combination enables you to create projects that sense and control the physical world. The software is free, open source, and cross-platform. The boards are inexpensive to buy, or you can build your own (the hardware designs are also open source).

In addition, there is an active and supportive Arduino community that is accessible worldwide through the Arduino forums and the wiki (known as the Arduino Playground). The forums and the wiki offer project development examples and solutions to problems that can provide inspiration and assistance as you pursue your own projects.

Introduction to Arduino een designed to be easy to use for beginners who have no software or electronics experience. With Arduino, you can build objects that can respond to and/or control light, sound, touch, and movement. Arduino has been used to create an amazing variety of things, including musical instruments, robots, light sculptures, games, interactive furniture, and even interactive clothing

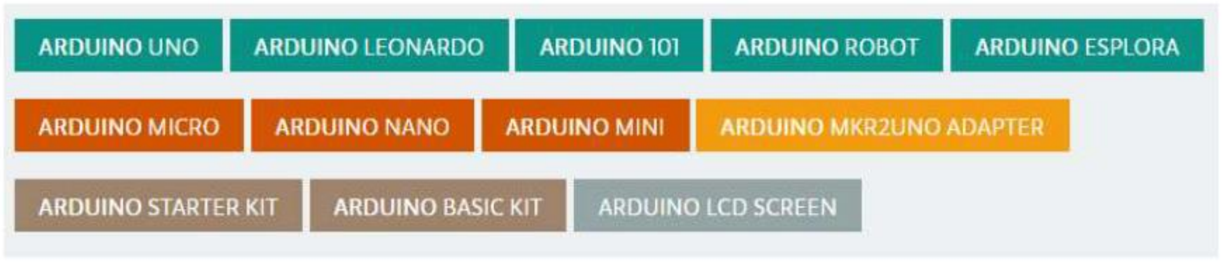
Arduino Software

Software programs, called sketches, are created on a computer using the Arduino integrated development environment (IDE). The IDE enables you to write and edit code and convert this code into instructions that Arduino hardware understands. The IDE also transfers those instructions to the Arduino board (a process called uploading).

Arduino Hardware

The Arduino board is where the code you write is executed. The board can only control and respond to electricity, so specific components are attached to it to enable it to interact with the real world. These components can be sensors, which convert some aspect of the physical world to electricity so that the board can sense it, or actuators, which get electricity from the board and convert it into something that changes the world. Examples of sensors include switches, accelerometers, and ultrasound distance sensors. Actuators are things like lights and LEDs, speakers, motors, and displays.

Arduino Types

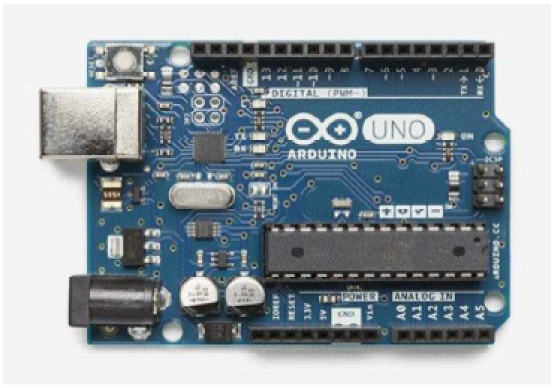


ARDUINO UNO | ARDUINO LEONARDO | ARDUINO 101 | ARDUINO ROBOT | ARDUINO ESPLORA

ARDUINO MICRO | ARDUINO NANO | ARDUINO MINI | ARDUINO MKR2UNO ADAPTER

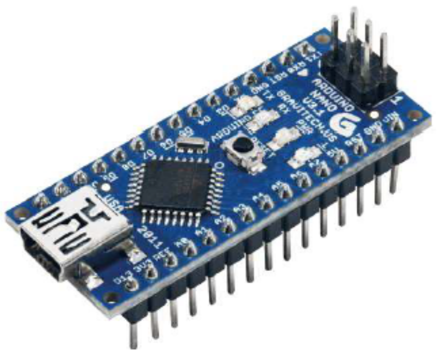
ARDUINO STARTER KIT | ARDUINO BASIC KIT | ARDUINO LCD SCREEN

1- Arduino UNO



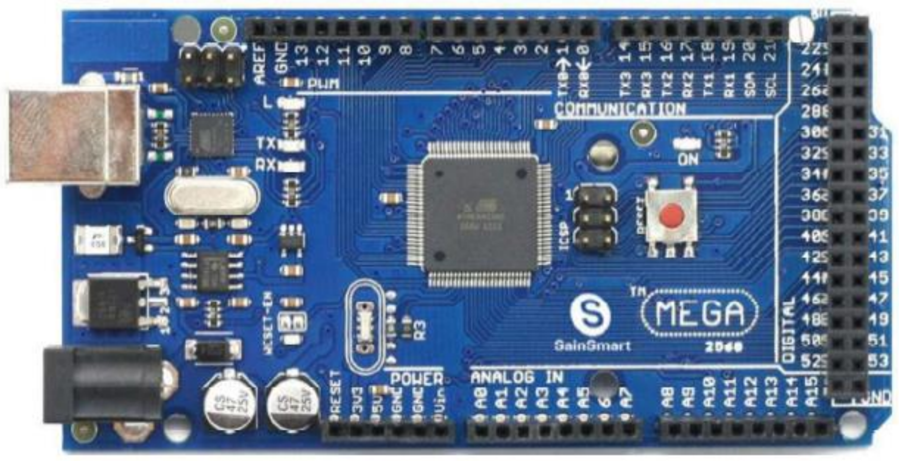
Microcontroller	*ATmega328P*
Operating Voltage	5V
Input Voltage (recommended)	7-12V
Input Voltage (limit)	6-20V
Digital I/O Pins	14 (of which 6 provide PWM output)
PWM Digital I/O Pins	6
Analog Input Pins	6
DC Current per I/O Pin	20 mA
DC Current for 3.3V Pin	50 mA
Flash Memory	32 KB (ATmega328P) of which 0.5 KB used by bootloader
SRAM	2 KB (ATmega328P)
EEPROM	1 KB (ATmega328P)
Clock Speed	16 MHz
LED_BUILTIN	13
Length	68.6 mm
Width	53.4 mm
Weight	25 g

2- Arduino Nano



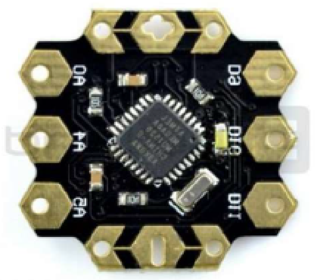
Microcontroller	*Atmel ATmega168 or ATmega328*
Operating Voltage (logic level)	5 V
Input Voltage (recommended)	7-12 V
Input Voltage (limits)	6-20 V
Digital I/O Pins	14 (of which 6 provide PWM output)
Analog Input Pins	8
DC Current per I/O Pin	40 mA
Flash Memory	16 KB (ATmega168) or 32 KB (ATmega328) of which 2 KB used by bootloader
SRAM	1 KB (ATmega168) or 2 KB (ATmega328)
EEPROM	512 bytes (ATmega168) or 1 KB (ATmega328)
Clock Speed	16 MHz
Dimensions	0.73" x 1.70"
Length	45 mm
Width	18 mm
Weigth	5 g

3- Arduino Mega 2560



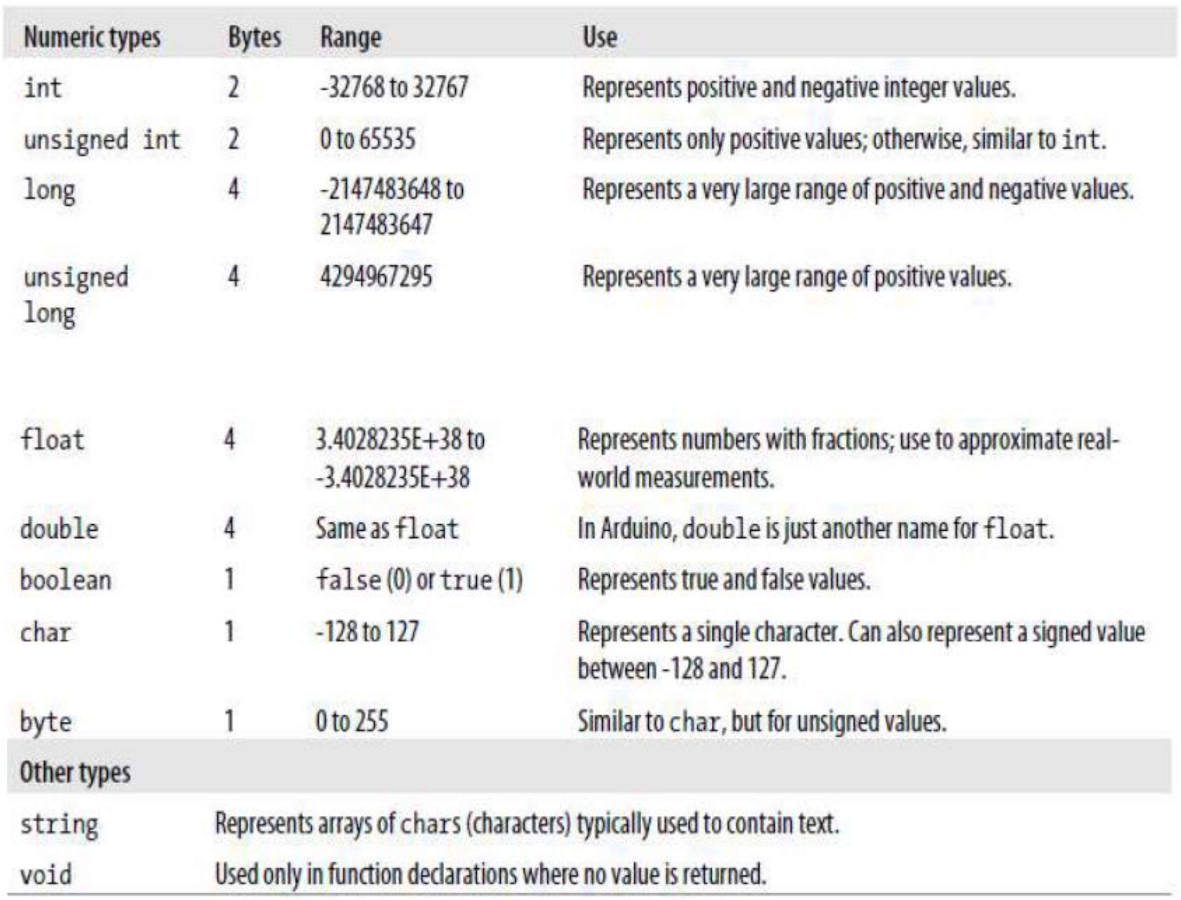
Microcontroller	ATmega2560
Operating Voltage	5V
Input Voltage (recommended)	7-12V
Input Voltage (limit)	6-20V
Digital I/O Pins	54 (of which 15 provide PWM output)
Analog Input Pins	16
DC Current per I/O Pin	20 mA
DC Current for 3.3V Pin	50 mA
Flash Memory	256 KB of which 8 KB used by bootloader
SRAM	8 KB
EEPROM	4 KB
Clock Speed	16 MHz
LED_BUILTIN	13
Length	101.52 mm
Width	53.3 mm
Weight	37 g

4- Cheapduino



- Working voltage: 3~5 volts
- Recommended power supply: 5v
- Microctonroller: Atmel AVR ATmega8
- bootloader(Board option in Arduino IDE): Arduino NG / w ATmega8
- 3 digital pins, 3 analog pins with easy-to-solder hexagonal pads
- Integrate 3 pwm pins,I2C interface and UART interface
- Suitable for workshop,education usage,DIY or compact size projects and E-Textiles
- Low cost Arduino compatible controller
- Dimensions: 2cm x 2cm x 0.2cm (0.78x0.78x0.08")

Arduino data type



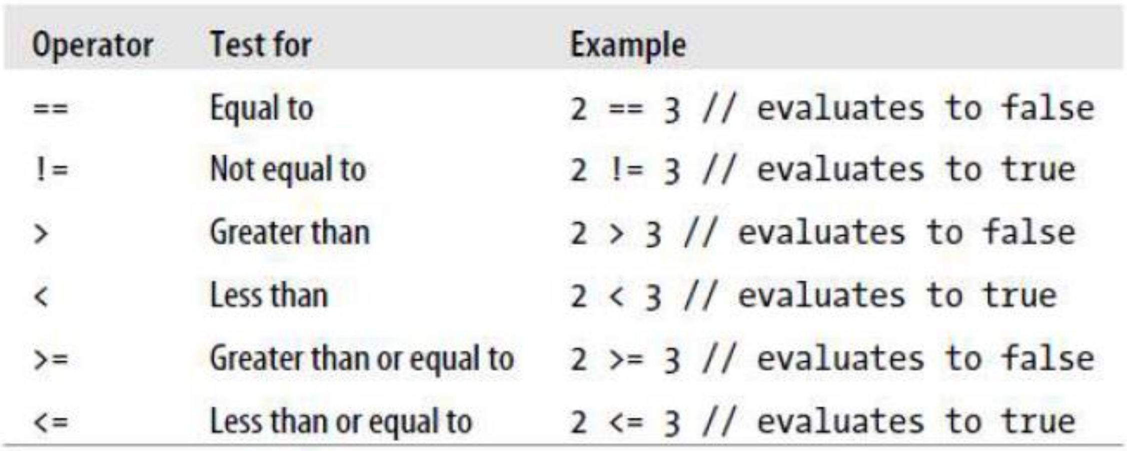
Numeric types	Bytes	Range	Use
`int`	2	-32768 to 32767	Represents positive and negative integer values.
`unsigned int`	2	0 to 65535	Represents only positive values; otherwise, similar to `int`.
`long`	4	-2147483648 to 2147483647	Represents a very large range of positive and negative values.
`unsigned long`	4	4294967295	Represents a very large range of positive values.
`float`	4	3.4028235E+38 to -3.4028235E+38	Represents numbers with fractions; use to approximate real-world measurements.
`double`	4	Same as `float`	In Arduino, `double` is just another name for `float`.
`boolean`	1	`false` (0) or `true` (1)	Represents true and false values.
`char`	1	-128 to 127	Represents a single character. Can also represent a signed value between -128 and 127.
`byte`	1	0 to 255	Similar to `char`, but for unsigned values.
Other types			
`string`	Represents arrays of `char`s (characters) typically used to contain text.		
`void`	Used only in function declarations where no value is returned.		

Although the int (short for integer, a 16-bit value in Arduino) data type is the most common choice for the numeric values encountered in Arduino applications, you can use Arduino data type to determine the data type that fits the range of values your application expects.

Discussion

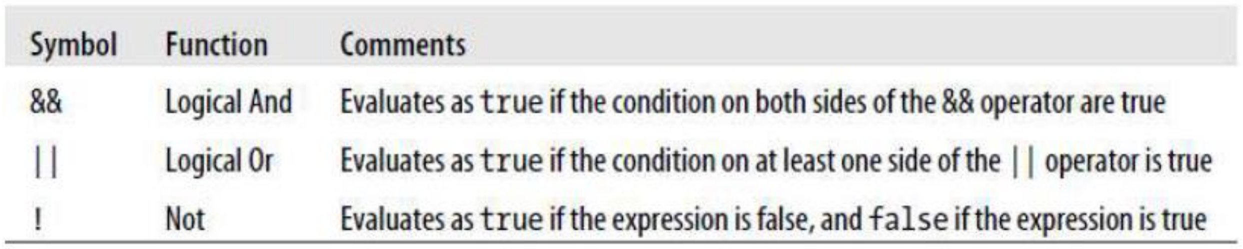
Except in situations where maximum performance or memory efficiency is required, variables declared using **int** will be suitable for numeric values if the values do not exceed the range (**shown in the first row in Arduino data type**) and if you don t need to work with fractional values. Most of the official Arduino example code declares **numeric variables** as int.
But sometimes you do need to choose a type that specifically suits your application. Sometimes you need negative numbers and sometimes you don t, so numeric types come in two varieties: **signed and unsigned.** unsigned values are always positive. Variables without the keyword unsigned in front are signed so that they can represent negative and positive values. One reason to use unsigned values is **when the range of signed values will not fit the range of the variable (an unsigned variable has twice the capacity of a signed variable)**.
Another reason programmers choose to use unsigned types is to clearly indicate to people reading the code that the value expected will never be a negative number.
boolean types have two possible values: **true or false.** They are commonly used for things like checking the state of a switch (if it s pressed or not). You can also use HIGH and LOW as equivalents to true and false where this makes more sense; **digitalWrite(**pin, HIGH) is a more expressive way to turn on an LED than digitalWrite(pin, true) or **digitalWrite**(pin,1), although all of these are treated identically when the sketch actually runs, and you are likely to come across all of these forms in code posted on the Web.

Rational and equality operators



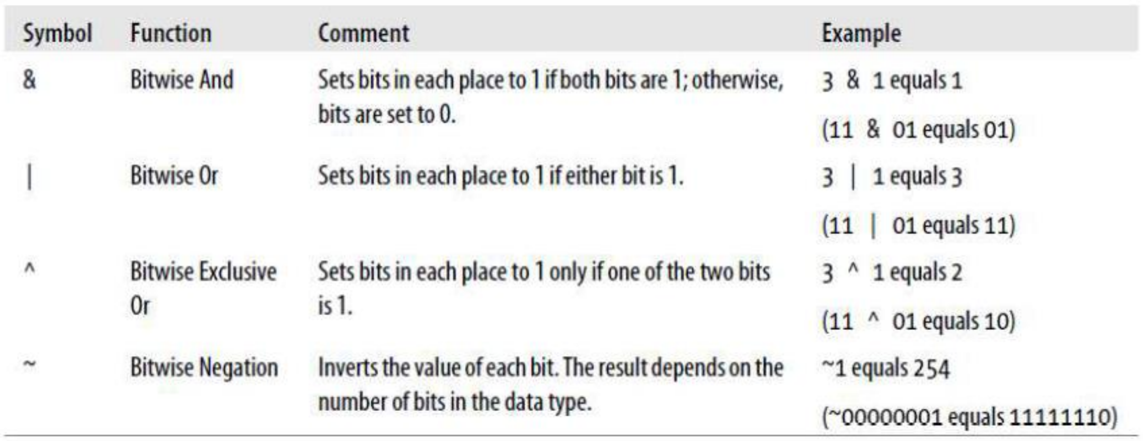
Operator	Test for	Example
==	Equal to	`2 == 3 // evaluates to false`
!=	Not equal to	`2 != 3 // evaluates to true`
>	Greater than	`2 > 3 // evaluates to false`
<	Less than	`2 < 3 // evaluates to true`
>=	Greater than or equal to	`2 >= 3 // evaluates to false`
<=	Less than or equal to	`2 <= 3 // evaluates to true`

- **Logical operators**e



Symbol	Function	Comments
&&	Logical And	Evaluates as `true` if the condition on both sides of the && operator are true
||	Logical Or	Evaluates as `true` if the condition on at least one side of the || operator is true
!	Not	Evaluates as `true` if the expression is false, and `false` if the expression is true

Bit operators



Symbol	Function	Comment	Example
&	Bitwise And	Sets bits in each place to 1 if both bits are 1; otherwise, bits are set to 0.	3 & 1 equals 1 (11 & 01 equals 01)
|	Bitwise Or	Sets bits in each place to 1 if either bit is 1.	3 | 1 equals 3 (11 | 01 equals 11)
^	Bitwise Exclusive Or	Sets bits in each place to 1 only if one of the two bits is 1.	3 ^ 1 equals 2 (11 ^ 01 equals 10)
~	Bitwise Negation	Inverts the value of each bit. The result depends on the number of bits in the data type.	~1 equals 254 (~00000001 equals 11111110)

Here is a sketch that demonstrates the example values shown in Table of Bit operators

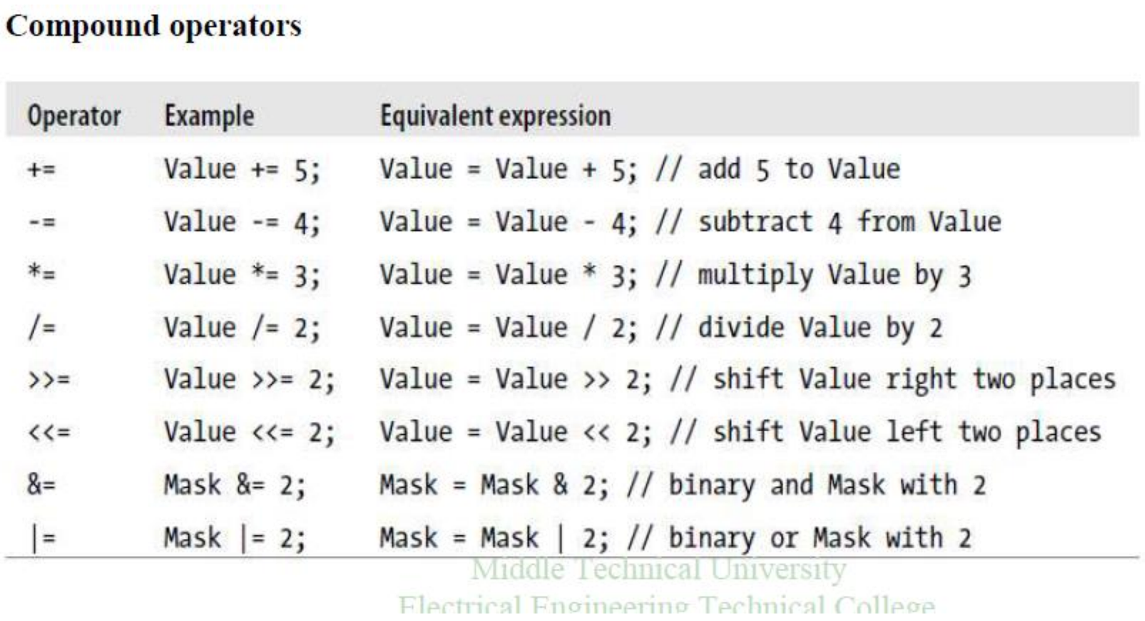
```
//*
* bits sketch
* demonstrates bitwise operators
*//
void setup() {
Serial.begin(9600);
}
void loop(){
Serial.print("3 & 1 equals "); // bitwise And 3 and 1
Serial.print(3 & 1); // print the result
Serial.print(" decimal, or in binary: ");
Serial.println(3 & 1 , BIN); // print the binary representation of the result
Serial.print("3 | 1 equals "); // bitwise Or 3 and 1
Serial.print(3 | 1 );
Serial.print(" decimal, or in binary: ");
Serial.println(3 | 1 , BIN); // print the binary representation of the result
Serial.print("3 ^ 1 equals "); // bitwise exclusive or 3 and 1
Serial.print(3 ^ 1);
Serial.print(" decimal, or in binary: ");
Serial.println(3 ^ 1 , BIN); // print the binary representation of the result
byte byteVal = 1;
int intVal = 1;
byteVal = ~byteVal; // do the bitwise negate
intVal = ~intVal;
Serial.print("~byteVal (1) equals "); // bitwise negate an 8 bit value
Serial.println(byteVal, BIN); // print the binary representation of the result
Serial.print("~intVal (1) equals "); // bitwise negate a 16 bit value
Serial.println(intVal, BIN); // print the binary representation of the result
delay(10000);
}
```

This is what is displayed on the Serial Monitor:

```
3 & 1 equals 1 decimal, or in binary: 1
3 | 1 equals 3 decimal, or in binary: 11
3 ^ 1 equals 2 decimal, or in binary: 10
~byteVal (1) equals 11111110
~intVal (1) equals 11111111111111111111111111111110
```

Discussion

Bitwise operators are used to set or test bits. When you And or Or two values, the operator works on each individual bit. It is easier to see how this works by looking at the binary representation of the values. Decimal 3 is binary 00000011, and decimal 1 is 00000001. Bitwise And operates on each bit. The rightmost bits are both 1, so the result of And-ing these is 1. Moving to the left, the next bits are 1 and 0; And-ing these results in 0. All the remaining bits are 0, so the bitwise result of these will be 0. In other words, for each bit position where there is a 1 in both places, the result will have a 1; otherwise, it will have a 0. So, 11 & 01 equals 1.



Compound operators

Operator	Example	Equivalent expression
`+=`	`Value += 5;`	`Value = Value + 5; // add 5 to Value`
`-=`	`Value -= 4;`	`Value = Value - 4; // subtract 4 from Value`
`*=`	`Value *= 3;`	`Value = Value * 3; // multiply Value by 3`
`/=`	`Value /= 2;`	`Value = Value / 2; // divide Value by 2`
`>>=`	`Value >>= 2;`	`Value = Value >> 2; // shift Value right two places`
`<<=`	`Value <<= 2;`	`Value = Value << 2; // shift Value left two places`
`&=`	`Mask &= 2;`	`Mask = Mask & 2; // binary and Mask with 2`
`|=`	`Mask |= 2;`	`Mask = Mask | 2; // binary or Mask with 2`