REPUBLIC OF THE PHILIPPINES
POLYTECHNIC UNIVERSITY OF THE PHILIPPINES
STA. MESA, MANILA
COLLEGE OF ENGINEERING
COMPUTER ENGINEERING DEPARTMENT

CMPE 30184

MICROPROCESSORS

INSTRUCTIONAL MATERIAL

ENGR. JULIUS CANSINO **ENGR. ROLITO MAHAGUAY**
ENGR. ORLANDO PAJABERA

POLYTECHNIC UNIVERSITY OF THE PHILIPPINES
COLLEGE OF ENGINEERING
DEPARTMENT OF COMPUTER ENGINEERING

Microprocessor Systems Syllabus

I. COURSE IDENTIFICATION	
Course Title	MICROPROCESSOR SYSTEMS
Course Code	
Course Credit	4 units
Course Pre-requisite	Logic Circuits and Switching Theory
Course Description	The course is the study of the design and applications of microprocessor systems. The focus is on the basic understanding of its structure and function in order to appreciate the architectural design of microprocessor.

II. COURSE OBJECTIVES	
General Objective	The students can create a microprocessor-based or microcontroller-based system.
Specific Objectives	At the end of the course, the students should be able to: 1. Design a circuit with microprocessor or microcontroller; 2. Implement microprocessor based system using different levels of implementation; 3. Develop the control software for the given system implementation; 4. Interface a device that will control by microprocessor or microcontroller; 5. Create a program that will control the device to work.

III. REFERENCES	
References	Blum, J. (2013). Exploring Arduino: Tools and Techniques for Engineering Wizardry. https://www.arduino.cc/

IV. COURSE OUTLINE		
Week	Modules	Hours
1	Orientation: Review of Course Syllabus Discussion of course goals, expected outcomes, classroom policies, requirements and grading system in lecture and laboratory. **Module 1: Introduction to Microcontrollers and Embedded Systems** - Microprocessor - Microcontroller - Features of Microcontroller - Embedded System	6
2	**Module 2: Exploring the Arduino Board** - Arduino - Advantages of Using Arduino	6

	History of Arduino Arduino Boards Arduino Shields **Laboratory Exercise: Familiarization of Arduino Commands**	
3	The Arduino UNO Boards Arduino IDE Code Structure **Laboratory Exercise: Familiarization of Arduino IDE**	6
4	**Module 3: Arduino Programming** Arduino Sketch Variable Type Arrays Strings Constants **Laboratory Exercise: Arduino Program1**	6
5	Operators Control Structures Functions **Laboratory Exercise: Arduino Program2**	6
6	**Module 4: Arduino Interfacing** Arduino Connection Arduino Communication Arduino to Computer Sending Information Baud Rate **Laboratory Exercise: Interfacing Arduino**	6
7	**Long Examination** **Practical Examination**	6
8	**Module 5: Digital Output** Interfacing LEDs DC Characteristics and Operation Interfacing to Arduino Board Programming Digital Output **Laboratory Exercise: Interfacing LED to Arduino**	6
9	**Midterm Examination** **Practical Examination**	6
10-11	**Module 6: Digital Input** The Push Button Interfacing Push Button to Arduino Programming Digital Input Switch Bounce **Laboratory Exercise: Interfacing Push Button to Arduino**	9
11-12	**Module 7: Enhanced Digital Output** 7-Segment Display Common Cathode Common Anode Displaying Digital Digits Driving a 7-Segment Display	9

	Interfacing 7-Segment Display to Arduino BCD Digit Display **Laboratory Exercise: Interfacing 7-Segment Display to Arduino**	
13	**Long Examination** **Practical Examination**	6
14	**Module 8: Enhanced Digital Input** - The Phone Keypad - Scanning the Keypad - Interfacing the Keypad **Laboratory Exercise: Interfacing Keypad to Arduino**	6
15	**Module 9: Interfacing the LCD Display** - LCD - LCD Types - Pin Configuration - Operation Mode - Controlling the LCD **Laboratory Exercise: Interfacing LCD to Arduino**	6
16-17	Project	12
18	FINAL EXAMINATION	6
	Total Number of Hours	108

V. COURSE REQUIREMENTS

Lectures/Instructional Materials
Long Examinations
Laboratory Exercises
Practical Exams
Midterm Exam
Final exam
Frequency of meetings: 6 hrs/week

VI. GRADING SYSTEM

✓ Long Examinations	20%
✓ Laboratory Exercises	15%
✓ Practical Examination	15%
✓ Midterm Examination	10%
✓ Project	15%
✓ Final Exam	20%
✓ Teacher's Evaluation	5%
TOTAL	100%

(Passing Mark: 75%)

Table of Contents

Module 1: Introduction to Microcontrollers and Embedded Systems

Objectives:

At the end of this module, the student should be able to:

Define microprocessor, microcontroller and embedded system;

Explain the function of microprocessor and microcontroller;

Discuss the components and features of microcontroller;

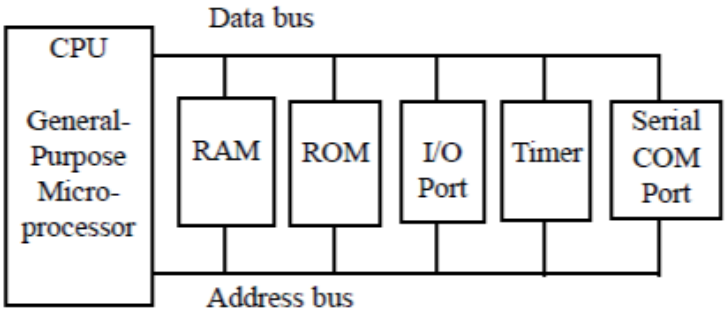
Explain the advantages and disadvantages of microcontroller;

Discuss the embedded system.

Microprocessor

A microprocessor (sometimes abbreviated as µP) is a digital electronic component with miniaturized transistors on a single semiconductor integrated circuit (IC). One or more microprocessors typically serve as a central processing unit (CPU) in a computer system or handheld device. It is commonly referred as general-purpose microprocessor.

A system designer using a microprocessor must add interfaces such as RAM, ROM, I/O ports and timers to make it functional.

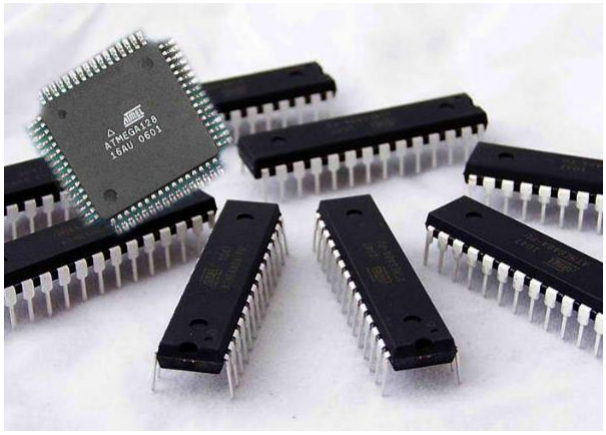


(a) General-Purpose Microprocessor System

The addition of external hardware makes these bulkier and much more expensive. However, this makes it versatile, the designer can decide on the amount of RAM, ROM and I/O ports needed to fit the required system.

Microcontroller

A microcontroller or MCU has a microprocessor in addition to a fixed amount of RAM, ROM, I/O ports and a timer all on a single chip. In other words, the processor, RAM, ROM, I/O ports and timer are all embedded together on one chip.



The designer can use the fixed amount of on-chip ROM, RAM and number of I/O ports in microcontroller makes it ideal for applications in which cost and space are critical.

CPU	RAM	ROM
I/O	Timer	Serial COM Port

(b) Microcontroller

Features of Microcontroller

Central Processing Unit (CPU) – usually small and simple

Input/Output interfaces such as serial ports

Peripherals such as timers and watchdog circuits

RAM for data storage

ROM for program storage

Clock generator – often as oscillator for a quartz timing crystal, resonator or RC circuit

Advantages of Microcontroller

Microcontroller is inexpensive

Microcontroller development tools are typically free or significantly discounted for promotional purposes

Microcontroller development tools have minimal computing resource requirements

Microcontroller I/O ports are easily accessible in comparison with computer I/O

Microcontroller is easy to interface to other electronic circuits and devices.

Disadvantages of Microcontroller

Limited programming language and programming tool choices

Microcontroller is well suited to simple control and interface tasks.

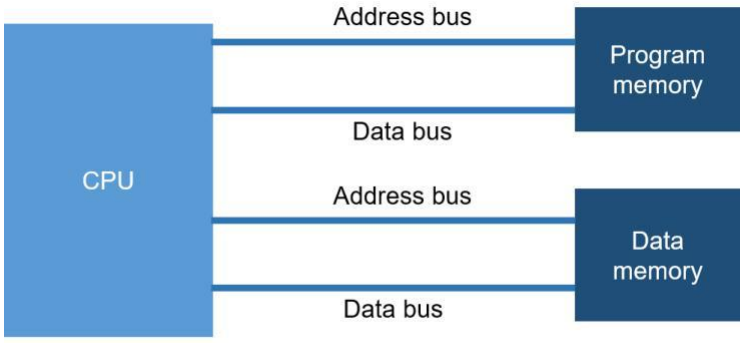
Brief History of Microprocessor

1970-71 when Intel was working on inventing the world's first microprocessor, Gary Boone of Texas Instruments invented the microcontroller, TMS100 microcontroller.
1976 when 8048 was introduced as the first Intel's microcontroller. It was used as the processor in the PC keyboard of IBM.
1980 when 8051 microcontroller was introduced and became one of the most popular microcontroller.
1990's when the advanced microcontroller with electrically erasable and programmable ROM memory started in the market.
Present day, microcontrollers like AVR and PIC have become smaller and sleeker yet more powerful.

Types of Microcontroller based on:

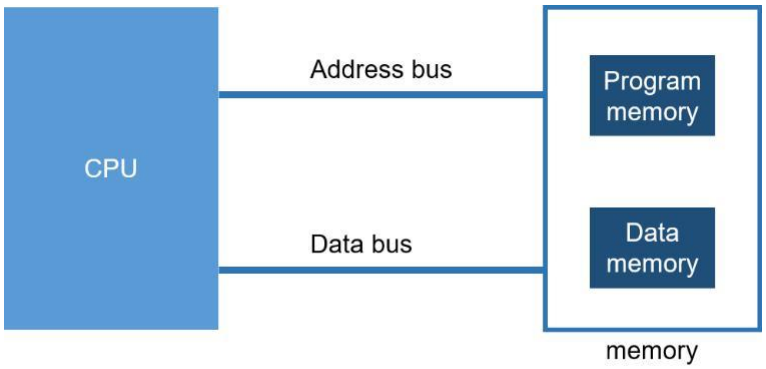
Architecture

Harvard – program and data are stored in separate memory banks.



Harvard architecture

Von Neumann – program and data are stored in one memory bank.



Von neumann architecture

Provider

AVR (Alf and Vegard's RISC Processor) – developed by Atmel service provider based on Harvard architecture
PIC (Peripheral Interface Controller) – based on Harvard architecture supports programming in C, Assembly and BASIC C, made by Microchip technology.
Hitachi – belongs to H8 family of the controller, (8-bit, 16-bit and 32-bit microcontroller), developed by Renesas Technology in Hitachi semiconductors.
Motorola – highly integrated microcontroller that is used for high-performance data manipulation operation, uses a System Integration Module (SIM), Time Processing Unit (TPU) and Queued Serial Module 9QSM).

Embedded System

An embedded system is an electronic/electromechanical system designed to perform a specific function using a combination of both hardware and firmware (software). Embedded system uses a microprocessor (or microcontroller) to do one task only.

Embedded systems are becoming an essential part of any product or equipment in all fields including:

House appliances – microwave, washing machine, etc.

Office equipment – fax machine, scanner, printer, etc.

Telecommunications – mobile phones, wifi, etc.

Transportation – speedometer, fuel sensor, gas sensor, etc.

Traffic control – traffic light, etc.

Medical equipment – thermal scanner, blood pressure device, etc.

Industrial control – sensor, manufacturing machine, etc.

Embedded system versus General Purpose Computer

A PC can be used for any number of applications such as word processor, print server, bank teller terminal, video game player, network server or internet terminal. Software for a variety of applications can be loaded and run. PC can perform variety of tasks because it has operating system that loads application software to memory to run it.

Embedded system uses only one application software that is typically burned into ROM.

A PC is connected to various embedded devices such as keyboard, mouse, monitor or printer.

Best Microcontroller for Engineers and Geeks (*engineeringpassion.com, February 2020*)

Arduino Uno R3 Microcontroller Board – the cheapest, ready to connect board with various available online libraries and resources.

Teensy 4.0 – latest and fastest board available today.

Arduino Pro Mini 328 – this mini board is only for small-scale applications up to 5 volts.

ESP32 Microcontroller Board – Bluetooth and Wi-Fi duo combo on a single-chip, bit costly used for smart home and IoT based projects.

Raspberry Pi 4 – fastest microcontroller with 4GB RAM, wireless LAN, Bluetooth, USB port, HDMI ports and Ethernet port.

MBED LPC1768 – mostly designed for prototyping applications, with I/O peripherals, USB port and Ethernet port.

BeagleBone Black – cheapest development board available, with 46 x 2 number of header pins, Ethernet and USB port.

ESP8266 Microcontroller Board – small in size as compared to other microcontroller with IoT capability.

Quark D2000 – one of the most robust microcontroller and has more I/O controls.

Launchpad MSP430 – most useful for application of Energy Trace on-board emulation and debugging.

Module 2: Exploring the Arduino Board

Objectives:

At the end of this module, the student should be able to:

Define the Arduino and its component;

Discuss the advantages and disadvantages of Arduino;

Discuss the history of Arduino;

Discuss the different Arduino boards;

Explain the different Arduino shields;

Discuss the Arduino UNO boards and its pin connections;

Explain the Arduino IDE.

Arduino

Arduino is an open-source electronics platform based on easy-to-use hardware and software. Arduino consists of the following components:

Physical programmable circuit board – designed around an Atmel microcontroller

Integrated Development Environment (IDE) – which is used to write the code on a computer and then upload it to the physical board.

Arduino does not need a separate piece of hardware (called a programmer) to load new code onto the board, simply a USB cable.

Arduino IDE uses a simplified version of C++ making it easier to learn to program.

Advantages of Using Arduino

Inexpensive – relatively inexpensive compared to other microcontroller.

Cross-platform – IDE runs on Windows, Macintosh OSX and Linux. Most microcontroller are limited to Windows.

Simple, clear programming environment – easy-to-use for beginners, yet flexible enough for advanced users to take advantage of as well.

Open source and extensible software – open source tools, available for extension by experienced programmers.

Open source and extensible hardware – experienced circuit designers can make their own version of the module.

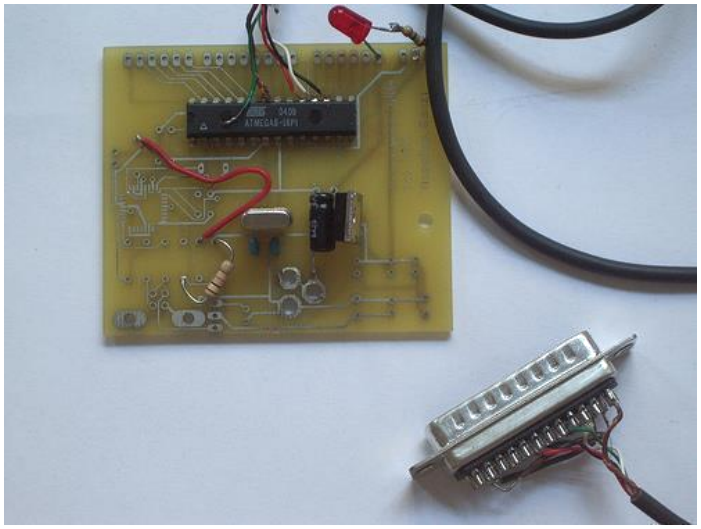
Disadvantages of Using Arduino

Not easy to modify – shields are difficult to modify.

Syntax – need to memorize the syntax and sketches in programming the Arduino.

History of Arduino

2005 – Hernando Barragan created the development platform Wiring as a Master's thesis project at IDII (Interaction Design Institute Ivrea) in Ivrea, Italy. ***Massimo Banzi*** and ***David Cuartielles*** created Arduino. ***David Mellis*** developed the Arduino software. ***Gianluca Martino*** and ***Tom Igoe*** joined the project.



The First Arduino Board

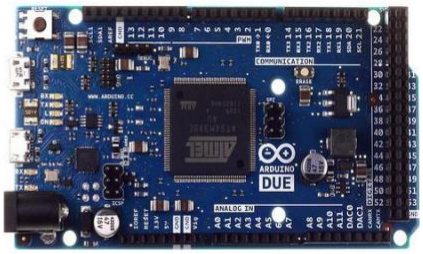
Arduino Boards

Arduino boards vary in the following aspects:

Microcontroller type;
Microcontroller speed (frequency);
Physical size;
Number of input and output pins;
Memory space for programs
Price

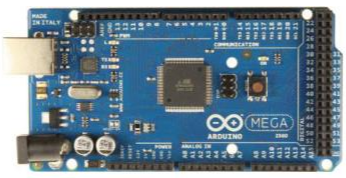
Arduino Uno

Arduino Leonardo

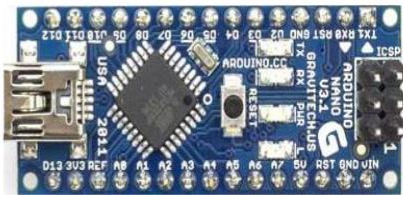


Arduino Due

Arduino Micro



Arduino Mega 2560



Arduino Nano

Comparison Between Arduino Boards

Name	Processor	Operating Voltage/Input Voltage	CPU Speed	Analog In/Out	Digital IO/PWM	EEPROM [KB]	SRAM [KB]	Flash [KB]	USB	UART
Uno	ATmega328	5V/7-12V	16Mhz	6/0	14/6	1	2	32	Regular	1
Due	AT91SAM3X8E	3.3V/7-12V	84Mhz	12/2	54/12	-	96	512	2 Micro	4
Leonardo	ATmega32u4	5V/7-12V	16Mhz	12/0	20/7	1	2.5	32	Micro	1
Mega 2560	ATmega2560	5V/7-12V	16Mhz	16/0	54/15	4	8	256	Regular	4
Micro	ATmega32u4	5V/7-12V	16Mhz	12/0	20/7	1	2.5	32	Micro	1
Nano	ATmega168 ATmega328	5V/7-9V	16Mhz	8/0	14/6	.512 1	1 2	16 32	Mini-B	1

Arduino Shields

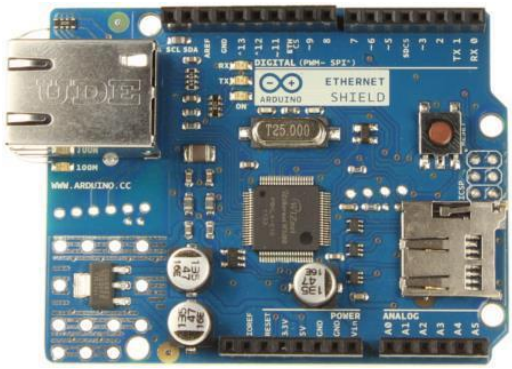
A shield is a circuit board that connects via pins to the sockets on the sides of an Arduino.

GSM Shield

The GSM Shield with a SIM card allows an Arduino board to connect to the internet, make/receive voice calls and send/receive SMS messages.

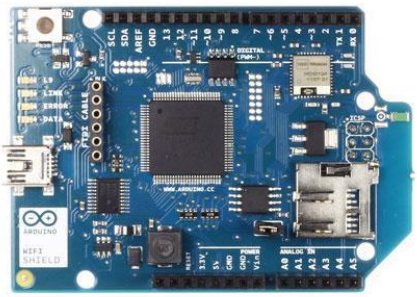
Ethernet Shield

The Ethernet Shield allows an Arduino board to connect to the internet. It provides a network (IP) stack capable of both TCP and UDP.

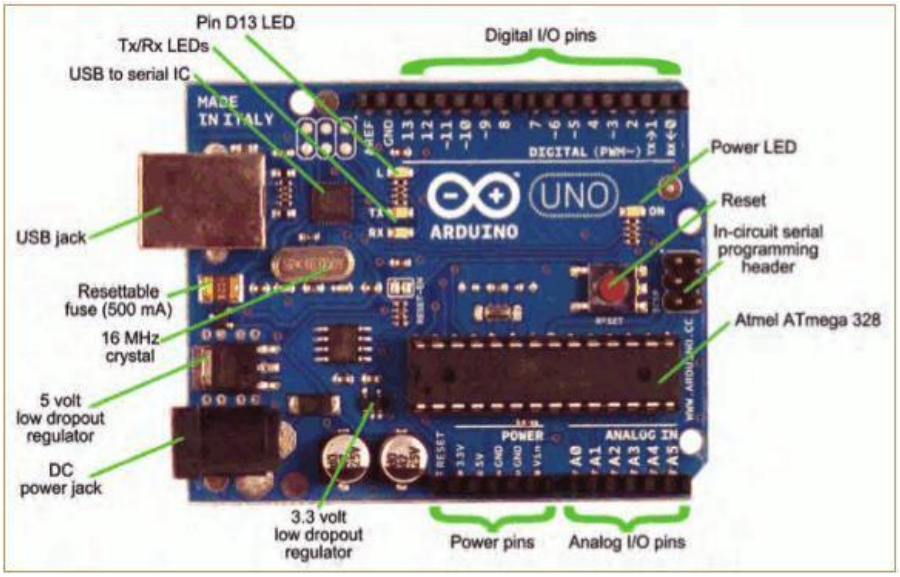


Wi-Fi Shield

The Wi-Fi Shield allows an Arduino board to connect to the internet using the 802.11 wireless specification (Wi-Fi). It provides a network (IP) stack capable of both TCP and UDP.



The Arduino UNO Board



Arduino UNO Parts

ATmega382 microcontroller

16Mhz clock

14 digital input/output pins (6 can be used as PWM outputs)

6 analog inputs

ICSP header

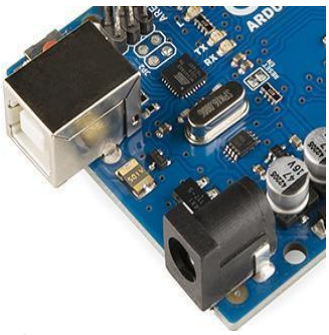
USB connection

Power jack

Reset button

LEDs

USB (Universal Serial Bus)



It Connects the board to a computer for three reasons:

Supply power to the board

Upload code to the Arduino

Send data to and receive it from a computer

Power Connector

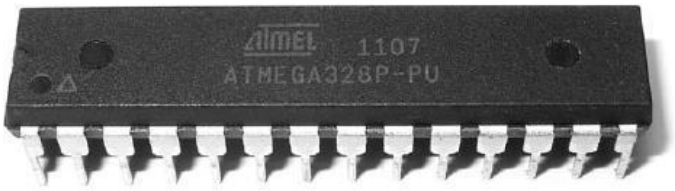
It is used to power the Arduino with a standard mains power adapter.

Microcontroller

It includes various types of memory to hold data and programs.

It executes instructions.

It provides various means of sending and receiving data.



Power Pins

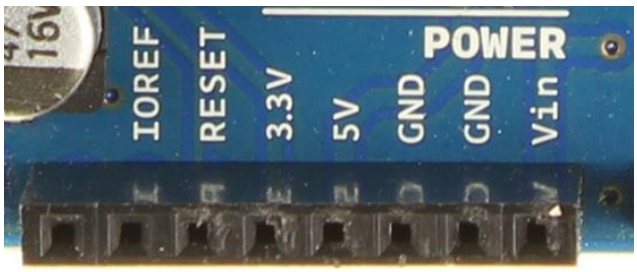
VIN – this pin is used to receive input voltage to the Arduino board when its using an external power source.

5V – this pin outputs a regulated 5V from the regulator on the board.

3.3V – this pin outputs a regulated 3.3V from the regulator on the board.

GND – this pin provides the Ground signal.

IOREF – this pin on the Arduino board provides the voltage reference with which the microcontroller operates.

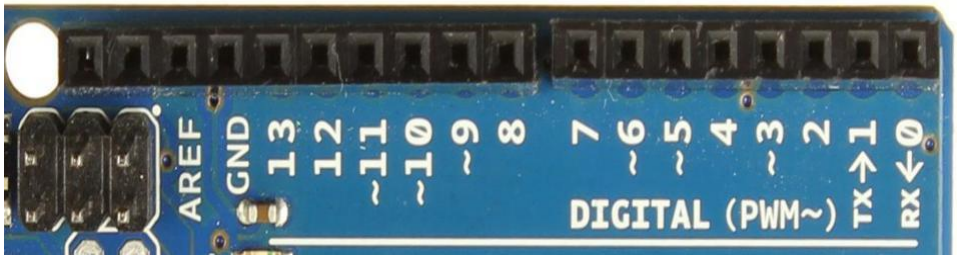


Analog Pins

Offers 6 analog inputs that are used to measure electrical signals that vary in voltage. Furthermore, pins **A4** and **A5** can also be used for sending data to and receiving it from other devices.

Digital Input/Output Pins

Pins 0 to 13 are digital input/output (I/O) pins. Pins **0** (**RX**) and **1** (**TX**) are also known as the serial port, which is used to send and receive serial data to other devices. Pins with tilde (**~**) can also generate a varying electrical signal (PWM).



Arduino IDE

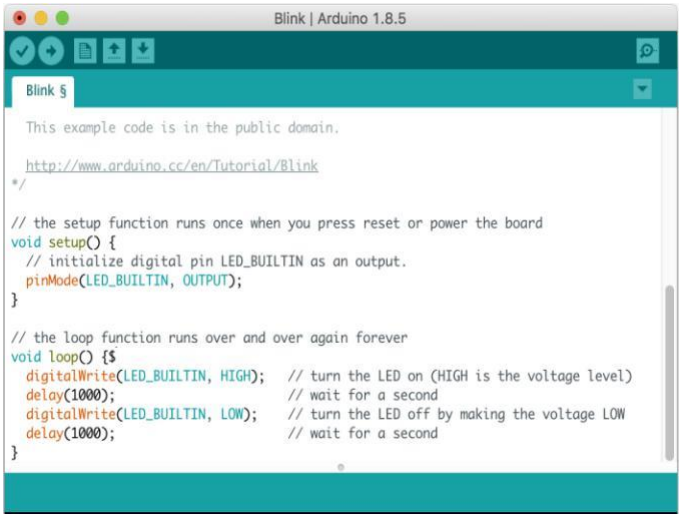
The open-source software where to write and compile the code for Arduino.

Using the IDE is a 3-step process:

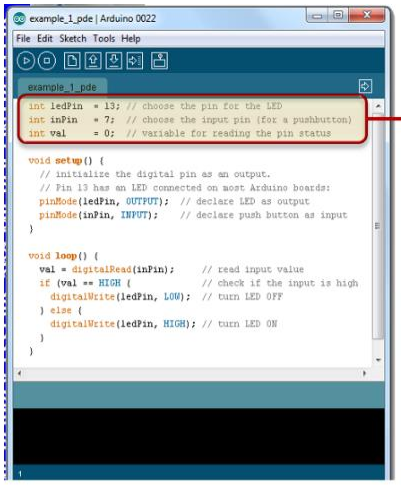
Write the code

Compile the code

Upload to Arduino



Code Structure



Variable declaration

Variables are **containers for values**.

You can give a variable any name you like.

The **value** of a variable can be changed and used dynamically as many times as you like.

Variables are good to use **when you use values more than once in your program**.

Arduino Uno has 32 kilobytes of flash memory and can store up to **1024 bytes** as variables.

The setup method

The Setup method is used to initiate the board before the code gets executed.

This **code runs only once** when the reset button has been pressed.

It defines if pins should be used **as inputs or outputs**, **whether serial communication is going** to be used, etc.

The program loop

The loop is where the action is!

This is where you write your program

It executes **from top to bottom**, line by line, and then starts over (looping back).

Module 3: Arduino Programming

Objectives:

At the end of this module, the student should be able to:

Explain the Arduino sketch;

Discuss the parts of Arduino sketch;

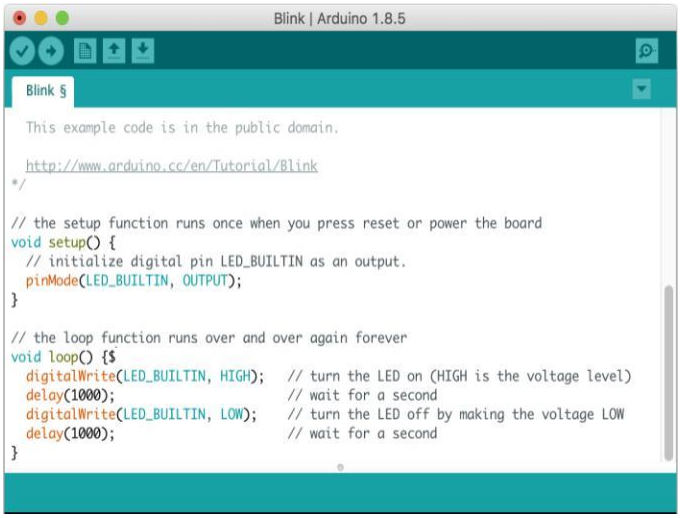
Create Arduino sketch;

Discuss the variables and constants in Arduino sketch;

Explain the operators, flow controls and functions in Arduino sketch.

Arduino Sketch

An Arduino sketch is a set of instructions (program) that you create to accomplish a certain task.



Parts of Arduino Sketch

Setup Function

Loop Function

```
void setup() {
  // put your setup code here, to run once
}

void loop() {
  // put your main code here, to run repeatedly
}
```

Setup Function

The **setup()** function is called when a sketch starts.

It is used to initialize variables, pin modes, start using libraries, etc.

The setup function will only run once, after each power up or reset of the Arduino board.

Loop Function

The **loop()** function does precisely what its name suggests and loops consecutively, allowing the program to change and respond as it runs.

Code in the **loop()** section of the sketch is used to actively control the Arduino board.

Sample Program

The code will display the message "***Hello World!***".

```
void setup() {
     initialize and start the serial
  port Serial.begin(9600);

     display the message on the monitor
  Serial.println("Hello World!");
}

void loop() {
}
```

Any line that starts with two slashes (***//***) is a comment and will not be read by the compiler.

Comments in the code give label and explain the task of that line of code.

Serial class is used for communication between the Arduino board and a computer or other devices.

Serial.begin() function is used to initialized and start the serial port.

Serial.println() function is used to display a text message on the Arduino environment's built-in serial monitor which is used to communicate with the Arduino board.

Variable Type

It tells the compiler what type or how data will be used in the program.

Void type

The void keyboard is used only in function declarations.

It indicates that the function is expected to return no information/value to the calling instruction.

```
void setup() {
    // put your setup code here, to run once
}
```

Boolean type

A Boolean variable can hold one of two values: true or false.

Each Boolean variable occupies one byte (8-bits) of memory.

```
boolean test = false;
```

Char type

The **char** variable stores a character value. Each char variable occupies one byte (8-bits) of memory.

Character literals are written in single quotes (‘**A**’). Characters are stored as number using ASCII encoding. This means that is it possible to do arithmetic on characters, in which the ASCII value of the character is used (‘**A**’ + 1 = 66).

The **char** data type is a signed type, that is encodes numbers from -128 to 127.

```
char myChar = 'A';
char myChar = 65; // both are equivalent
```

Byte type

The **byte** variable stores 8-bit unsigned number, from 0 to 255.

```
byte b = B10010; // "B" is the binary formatter
                 // (B10010 = 18 decimal)
```

Int and Unsigned Int type

An **int** variable stores a 16-bit (2-byte) value. This yields a range of -32,768 to 32,767.

The **unsigned int** is the same as the int type however, it only stored positive values, from 0 to 65,535.

```
int x = -13;
unsigned int y = 13;
```

Long and Unsigned Long type

A **long** variable can stores 32-bits (4-byte), from -2,147,483,648 to 2,147,483,647. If doing math with integers, at least one of the numbers must be followed by an **L**, forcing it to be a long.

An **unsigned long** variable can stores 32-bits (4-byte). It does not store negative numbers, making its range from 0 to 4,294,967.295.

```
long s = -18600L;
unsigned long v = 18600L;
```

Float and double type

A **float** variable is used to express floating-point numbers (with decimal point). Floating-point numbers can be as large as 3.4028235E+38 and as low as -3.4028235E+38. They are stored as 32-bits (4-byte).

On most Arduino boards, **double** is the same size as **float**. On Arduino Due, **double** has 64-bits (8-byte).

If doing math with **float**, it needs to add a decimal point, otherwise it will be treated as an **int**.

```
int x;
int y;
float z;
x = 1;
y = x / 2; // y now contains 0, int can't hold fractions
z = (float)x / 2.0; // z now contains 0.5 (use 2.0, not 2)
```

Arrays

An array is a collection of variables that are accessed with an index number.

Declaring an Array

It can declare an array without initializing it, as in **myInts**.

It can declare an array without explicitly choosing a size, as in **MyPins** (compiler counts the elements and creates an array of the appropriate size).

It can both initialize and size the array, as in **mySensVals**.

```
int myInts[6];
int myPins[] = {2, 4, 8, 3, 6};
int mySensVals[6] = {1, 4, -8, 3, 2, 7};
```

Accessing the Array

Arrays start in element 0 as the first element of an array.

```
int myInts[6];
int myPins[] = {2, 4, 8, 3, 6};
int mySensVals[6] = {1, 4, -8, 3, 2, 7};
```

myPins[0] = 2

myPins[2] = 8

myPins[4] = 6

mySensVals[1] = 4

mySensVals[2] = -8

mySensVals[5] = 7

Assigning and Retrieving Values from Arrays

It can assign or retrieve a value from an array using as index.

```
int myInts[6];
int myPins[] = {2, 4, 8, 3, 6};
int mySensVals[6] = {1, 4, -8, 3, 2, 7};
```

x = myPins[3];

y = mySensVals[4];

therefore:

x = 3

y = 2

Strings

Text string can be represented in two ways:

Make a string out of an array of type char and null-terminate it
- Use the string object data type

Char-Array

An **array** of type ***char*** can be declared in one of the following ways:

- Declare an array of chars without initializing it, as in Str1
- Declare an array of chars (with one extra char) and the compiler will add the required null character, as in Str2
- Explicitly add the null character, as in Str3
- Initialize with a string constant in quotation marks; the compiler will size the array to fit the string constant and a terminating null character, as in Str4
- Initialize the array with an explicit size and string constant, as in Str5
- Initialize the array, leaving extra space for a larger string, as in Str6

```
char Str1[15];

char Str2[8] = {'a', 'r', 'd', 'u', 'i', 'n', 'o'};

char Str3[8] = {'a', 'r', 'd', 'u', 'i', 'n', 'o', '\0'};

char Str4[ ] = "arduino";

char Str5[8] = "arduino";

char Str6[15] = "arduino";
```

An array can be terminated using:

Generally, strings are terminated with a **null character** (**ASCII code 0**).

This allows functions (like ***Serial.print()***) to tell where the end of a string is.

Otherwise, they would continue reading subsequent bytes of memory that aren't actually part of the string.

This means that the string needs to have space for one more character than the text wants to contain.

That is why **Str2** and **Str5** need to be eight characters, even though **"arduino"** is only seven - the last position is automatically filled with a null character.

Str4 will be automatically sized to eight characters, one for the extra null.

In **Str3**, we've explicitly included the null character (written '\0') ourselves.

Note that it's possible to have a string without a final null character (e.g. it had specified the length of **Str2** as seven instead of eight).

This will break most functions that use strings, so it shouldn't do it intentionally.

If there is something behaving strangely (operating on characters not in the string), however, this could be the problem.

Single Quote or Double Quotes

Strings are always defined inside double quotes (**"Abc"**) and characters are always defined inside single quotes(**'A'**).

Wrapping long strings

It can wrap long strings.

```
 char myString[] = "This is the first line"
this is the second line"
etcetera";
```

Arrays of strings

It is often convenient, when working with large amounts of text to setup an array of strings. Because strings themselves are arrays, this is in actually an example of a two-dimensional array.

All array names are actually pointers, so this is required to make an ***array*** of ***arrays***.

Pointers are one of the more esoteric parts of C for beginners to understand.

```
char* myStrings[]={"This is string 1",
                   "This is string 2",
                   "This is string 3",
                   "This is string 4",
                   "This is string 5",
                   "This is string 6"};
```

String-object

The ***String*** class allows to use and manipulate strings of text in more complex ways than **characters array** do.

It can concatenate strings, append to them, search for and replace substrings, and more.

It takes more memory than a simple character array, but it is also more useful.

```
String stringOne = "Hello String";

 String stringTwo = String("This is a string");
```

The ***String*** class has the following methods:
- *StringConstructors*
- *StringAdditionOperator*
- *StringIndexOf*
- *StringAppendOperator*
- *StringLengthTrim*
- *StringCaseChanges*
- *StringReplace*
- *StringCharacters*
- *StringStartsWithEndsWith*
- *StringComparisonOperators*
- *StringSubstring*

Constants

Constants are predefined variables in the Arduino language.

They are used to make the programs easier to read.

Constants are classified into the following groups:

Logical Levels
- Pin Levels
- Digital Pins

Logical Levels

There are two constants used to represent truth and falsity in the Arduino language:

- **FALSE** – false is defined as 0 (zero)
- **TRUE** – true is often said to be defined as 1, which is correct, but true has a wider definition, Any integer which is non-zero is true, in Boolean sense.

Pin Levels

When reading or writing to a digital pin there are only two possible values:

HIGH - The meaning of **HIGH** (in reference to a pin) is somewhat different depending on whether a pin is set to an **INPUT** or **OUTPUT**. When a pin is configured as an **INPUT**, the microcontroller will report **HIGH** if a voltage of **3 volts** or more is present at the pin. When a pin is configured to **OUTPUT** and set to **HIGH**, the pin is at **5 volts**.

LOW - The meaning of **LOW** also has a different meaning depending on whether a pin is set to **INPUT** or **OUTPUT**. When a pin is configured as an **INPUT**, the microcontroller will report **LOW** if a voltage of **2 volts** or less is present at the pin. When a pin is configured to **OUTPUT** and set to **LOW**, the pin is at **0 volts**.

Digital Pins

Digital pins can be used as:

INPUT - Pins configured as **INPUT** are said to be in a **high-impedance** state.

These pins make extremely small demands on the circuit that they are sampling. This makes them useful for reading a sensor. If the pin configured as an **INPUT**, it needs to be connected to a reference ground, often accomplished with a **pull-down resistor** (a resistor going to ground).

OUTPUT - Pins configured as **OUTPUT** are said to be in a **low-impedance** state. This means that they can provide a substantial amount of current to other circuits. These pins can **source** (provide positive current) or sink (provide negative current) up to **40 mA** of current to other devices/circuits. This makes them useful for powering LED's. Pins configured as outputs can also be damaged or destroyed if short circuited to either **ground** or **5V** power rails. The amount of current provided by an **OUTPUT** pin is also not enough to power most **relays** or **motors**, and some interface circuitry will be required.

Integers

Integer constants are numbers used directly in a sketch, like 123.

By default, an integer constant is treated as an int with the attendant limitations in values. To specify an integer constant with another data type, follow it with:

- a '**u**' or '**U**' to force the constant into an ***unsigned int*** data format. Example: **33u**
- a '**l**' or '**L**' to force the constant into a ***long*** data format. Example: **100000L**
- a '**ul**' or '**UL**' to force the constant into an ***unsigned long*** constant. Example: **32767UL**

Normally, integer constants are treated as base 10 (decimal) integers, but special notation (formatters) may be used to enter numbers in other bases:

- Decimal is base 10
  Constants without other prefixes are assumed to be in decimal format.

  Binary is base 2
  Only characters 0 and 1 are valid. The binary formatter only works on bytes (8 bits) between 0 (B0) and 255 (B11111111). Example: **B101** // same as 5 decimal

  Octal is base 8
  Only characters 0 through 7 are valid. Octal values are indicated by the prefix "0“. Example: **0101** // same as 65 decimal

  Hexadecimal is base 16
  Valid characters are 0 through 9 and letters A through F. Note that A-F may be written in upper or lower case (a-f). Hex values are indicated by the prefix "0x". Example: **0x101** // same as 257 decimal

Floating Point

Similar to integer constants, floating point constants are used to make code more readable. Example: **0.005**

Floating point constants can be expressed in a variety of scientific notation:

10.0 = 10

2.34E5 = 2.34 x 10^5 = 234000

67e-12 = 67.0 x 10^{-12} = 0.000000000067

Operators

Arithmetic Operators

Five arithmetic operations are supported:

- Addition +
- Subtraction -
- Multiplication *
- Division /
- Modulo %

```
int x = 9;
int y = 2;
int z;
z = x + y; // z = 11 (9+2=11)
z = x - y; // z = 7  (9-2=7)
z = x * y; // z = 18 (9*2=18)
z = x / y; // z = 4  (9/2=4 not 4.5)
z = x % y; // z = 1  (remainder of 9/2 = 1)
```

Comparison Operators

Six comparison operators are supported:

- Equal to ==
- Not equal to !=
- Less than <
- Greater than >
- Less than or equal to <=
- Greater than or equal to >=

```
int x = 9; int y = 2; int z = 9;
```

```
(x == y) // (9 == 2) false
(x == z) // (9 == 9) true
(x != y) // (9 != 2) true
(x != z) // (9 != 9) false
(x  < y) // (9  < 2) false
(x  < z) // (9  < 9) false
(x  > y) // (9  > 2) true
(x  > y) // (9  > 9) false
(x <= z) // (9 <= 9) true
(x >= z) // (9 >= 9) true
```

Boolean Operators

Three Boolean operators are supported:

- AND &&
- OR ||
- NOT !

```
int x = 8; int y = 2;
```

```
  ((x < 9) && (y > 1)) // T: (8<9)=T and (2>1)=T
  ((x < 9) && (y > 3)) // F: (8<9)=T and (2>3)=F
```

```
((x < 7)  || (y > 3)) // F: (8<7)=F or (2>3)=F
((x < 7)  || (y > 1)) // T: (8<7)=F or (2>1)=T
```

```
!(x < 7)              // T: (8<7)=F
!(x > 7)              // F: (8>7)=T
```

Bitwise Operators

Six bitwise operations are supported:

- Bitwise AND &
- Bitwise OR |
- Bitwise XOR ^
- Bitwise NOT ~
- Bit Shift Left <<
- Bit Shift Right >>

```
int x = 92;    // in binary: 0000000001011100
int y = 101;   // in binary: 0000000001100101
```

```
int z = x & y; // result:0000000001000100,or 68  in decimal
```

```
int z = x | y; // result:0000000001111101,or 125 in decimal
```

```
int z = x ^ y; // result:0000000000111001,or 57  in decimal
```

```
int z = ~x;    // result:1111111110100011,or -93 in decimal
```

```
int z = x << 3;// result:0000001011100000,or 736 in decimal
```

```
int z = x >> 3;// result:0000000000001011,or 11  in decimal
```

Compound Operators

The following compound operators are supported:

- Increment ++
- Decrement --
- Compound Addition +=
- Compound Subtraction -=
- Compound Multiplication *=
- Compound Division /=
- Compound Bitwise AND &=
- Compound Bitwise OR |=

```
int x = 5;  // in binary: 0000000000000101
int y = x++; // y = 5; x = 6
int y = ++x; // y = 6; x = 6
x += 2;      // x = 7  (x = 5 + 2)
x *= 2;      // x = 10 (x = 5 * 2)
x /= 2;      // x = 2 (x = 5 / 2)
x &= 2;      // x = 0 (x = 0…0101 & 0…0010 = 0…0000)
```

Control Structures

Flow Control

The if statement

```
if (boolean_experssion) {
    // statement(s)
}
```

Rules:

If the ***Boolean expression*** evaluates to **true**, then the block of code inside the if statement will be executed. If the ***Boolean expression*** evaluates to **false**, then the first set of code after the end of the if statement

```
int x = 5; int y = 4;
if (x < 9) {
    y = 25;
}
Serial.println("Result = " + y);
```

```
Result = 25
```

The if-else statement

```
if (boolean_experssion) {
    // statement(s)
}
else {
    // statement(s)
}
```

Rules:

If the ***Boolean expression*** evaluates to **true**, then the **if block** of code will be executed, otherwise **else block** of code will be executed.

```
int x = 5; int y = 4;
if (x < 9) {
    y = 25;
}
else {
    y = 10;
}
Serial.println("Result = " + y);
```

```
Result = 25
```

The switch-case statement

```
switch(expression){
   case constant_expression: // statement(s)
                        break;
   case constant_expression: // statement(s)
                        break;
   ...
   default :                    // statement(s)
}
```

```
char grade = 'P';
switch(grade){
   case 'F': Serial.println("Fail");
             break;
   case 'P': Serial.println("Pass");
              break;
   default :
Serial.println("Invalid");
}
```

```
Pass
```

Rules:

The expression must have an integral or enumerated type.

It can have any number of case statements.

The constant-expression must be the same data type as the variable in the switch.

When the variable being switched on is equal to a case, the statements following that case will execute until a **break** statement is reached.

When a **break** statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.

If no **break** appears, the flow of control will *fall through* to subsequent cases until a break is reached.

A **switch** statement can have an optional **default** case, which must appear at the end of the switch.

The default case can be used for performing a task when none of the cases is true.

No **break** is needed in the default case.

Looping Construct

The while loop

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

Rules:

The ***condition*** may be any expression. The loop iterates while the ***condition*** is **true**. When the ***condition*** becomes **false**, program control passes to the line immediately following the loop.

```
int i = 3;
while (i < 5) {
   Serial.println('*');
   i++;
}
```

```
*
*
```

The do-while loop

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

Rules:

The ***statement(s)*** in the loop execute **once** before the ***condition*** is tested. If the ***condition*** is **true**, the flow of control jumps back up to ***do***, and executes the ***statement(s)*** again. This process repeats until the given ***condition*** becomes **false**.

```
int i = 3;
do {
   Serial.println('*');
   i++;
} while (i  < 5)
```

```
*
*
*
```

The for loop

```
for (init; condition; increment) {
   // statement(s)
}
```

Rules:

The init step is to initialize the loop control variable(s).

Next, the condition is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and flow of control jumps to the next statement just after the for loop.

After the body of the for loop executes, the flow of control jumps back up to the increment statement. This statement allows you to update the loop control variable(s).

The condition is now evaluated again.

If it is true, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition).

After the condition becomes false, the for loop terminates.

```
int a[5] = {3, 1, 7, 4, 5};
int sum = 0;
for(j = 0; j < 5; j++) {
        sum += a[j];
}
 Serial.println("Sum = " + sum);
```

```
Sum = 20
```

Functions

Functions are used to organize the actions performed by your sketch into functional blocks.

Functions package functionality into:

Inputs – information given to a function

Outputs – information provided by a function

This makes the code easier to

Structure

Maintain

- Reuse

User Defined Function

Creating a function by declaring its: ○

Return type

- Name
- Optional parameter

```
   finds the average of three real numbers
double avg3(double a, double b, double c){
   double result = (a + b +
   c)/3.0; return result;
}
```

```
   using the avg3 function to find the
   average of three real numbers
double x = 5.0;
double y = 7.5;
double z = 4.5;
Serial.println( avg3(x, y, z) );
```

Predefined Function

Arduino libraries provide a large set of predefined functions:

Digital I/O Functions

These functions deal with digital inputs and outputs:

pinMode(): Configures the specified Arduino pin to behave either as an input or an output.

digitalWrite(): Writes a **HIGH** or a **LOW** value to a digital pin.

digitalRead(): Reads the value from a specified digital pin, either **HIGH** or **LOW**.

Analog I/O Functions

These functions deal with analog inputs and outputs:

analogReference() : Configures the reference voltage used for analog input.

- ***analogRead()*** : Reads the voltage value of the specified analog pin.
- ***analogWrite()*** : Writes an analog value (PWM wave) to a pin.

Time Functions

These functions deal with time:

millis() : Returns the number of **milliseconds** since the Arduino board began running the current program.

micros() : Returns the number of **microseconds** since the Arduino board began running the current program.

delay() : **Pauses** the program for the amount of time (in **milliseconds**) specified as parameter.

delayMicroseconds() : **Pauses** the program for the amount of time (in **microseconds**) specified as parameter.

Math Functions

This group contains method for performing basic numeric operations:

min() : Calculates the minimum of two numbers.

- ***max()*** : Calculates the maximum of two numbers.
- ***abs()*** : Computes the absolute value of a number.
- ***pow()*** : Calculates the value of a number raised to a power.
- ***sqrt()*** : Calculates the square root of a number.

Trigonometry Functions

This group contains method for computing basic trigonometry functions: ○

sin() : Calculates the **sine** of an angle (in **radians**).

- ***cos()*** : Calculates the **cos** of an angle (in **radians**).
- ***tan()*** : Calculates the **tan** of an angle (in **radians**).

Random Numbers

These functions are used to generate pseudo random numbers:

randomSeed(): Initializes the pseudo-random number generator, causing it to start at an arbitrary point in its random sequence.

random(): Generates pseudo-random numbers.

Bits and Bytes

These functions deal with byte and bit data types:

lowByte() : Extracts the **low-order** (**rightmost**) **byte** of a variable.

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

bitRead() : Reads a **bit** of a number.

bitWrite() : Writes a **bit** of a numeric variable.

bitSet() : Sets (writes a **1** to) a **bit** of a numeric variable.

bitClear() : Clears (writes a **0** to) a **bit** of a numeric variable.

bit() : Computes the value of the specified bit (bit 0 is 1, bit 1 is 2, bit 2 is 4, etc.)

External Interrupts

These functions handle external interrupts:

- ***interrupts()***: **Re-enables** interrupts (after they've been disabled by ***noInterrupts()***).

noInterrupts(): **Disables** interrupts (you can re-enable them with ***interrupts()***).

Module 4: Arduino Interfacing

Objectives:

At the end of this module, the student should be able to:

Know the Arduino connection;

Know the sample output of Arduino program;

Know the sending of information from Arduino to computer;

Know the baud rate in Arduino program.

Arduino Connection

Arduino can use same USB cable for programming and to talk with computers.

Talking to other devices uses the “Serial” commands

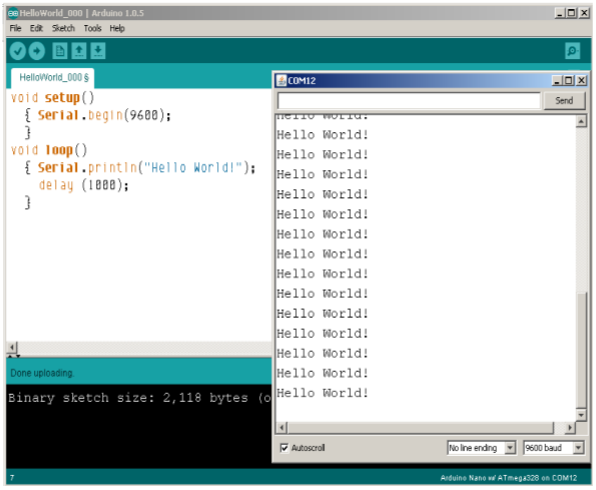
TX – sending to PC

RX – receiving from PC

Serial Communications

Sends “Hello World!” to the computer.

Click on “Serial Monitor” button to see output.



Arduino Communications

Is just serial communications

Arduino doesn't really do USB

It really is "serial", like old RS-232 serial

All microcontrollers can do serial

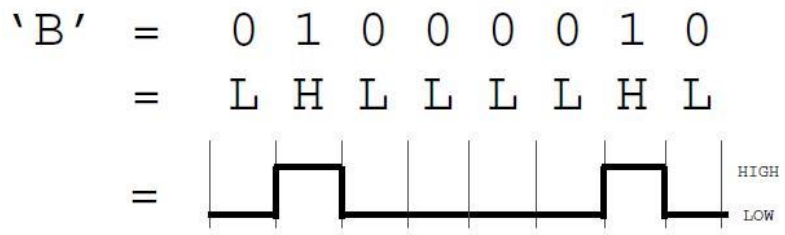
Not many can do USB

Serial is easy, USB is hard

Serial Communications

"Serial" because data is broken down into bits, each sent one after the other down a single wire.

The single ASCII character 'B' is sent as:



Toggle a pin to send data, just like blinking an LED

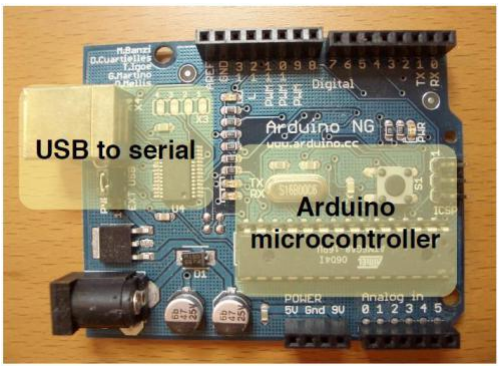
You could implement sending serial data with digitalWrite() and delay()

A single data wire needed to send data. One other to receive.

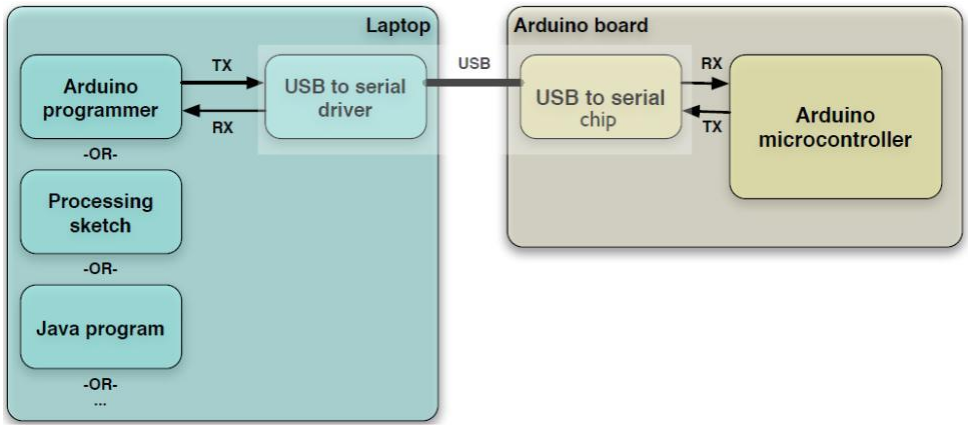
Arduino and USB-to-Serial

A standard Arduino has a single hardware serial port

But serial communication is also possible using software libraries to emulate additional ports.



Arduino to Computer



USB is totally optional for Arduino, but it makes things easier

Original Arduino boards were RS-232 serial, not USB

All programs that talk to Arduino (even the Arduino IDE) think that they're talking via a serial port

Serial Message Protocol

Where each message begins and ends?

Sides must agree how information is organized in the message (*communications protocol*)

Header - one or more special characters that identify the start of message

Footer - one or more special characters that identify the end of message

Sending Debug Information

This sketch prints sequential numbers on the Serial Monitor:

```
void setup(){
 Serial.begin(9600); // send and receive at 9600 baud
}
int number = 0;
void loop(){
  Serial.print("The number is ");
  Serial.println(number); // print the number
  delay(500); // delay half second between
  numbers number++; // to the next number
}
```

```
The number is 0
The number is 1
The number is 2
```

Sending Information

Display text using the Serial.print() or Serial.println() function

```
int val = 123;
Serial.print(val);      // sends 3 ASCII chars "123"
Serial.print(val,DEC);  // same as above
Serial.print(val,HEX);  // sends 2 ASCII chars "7B"
Serial.print(val,BIN);  // sends 8 ASCII chars "01111011"
Serial.print(val,BYTE); // sends 1 byte, the verbatim value
```

println() – prints the data followed by a carriage return character and a newline character

These commands can take many forms

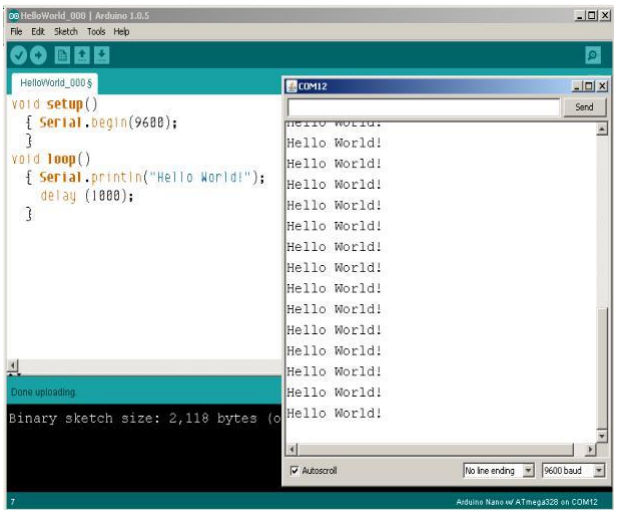
Numbers are printed using an ASCII character for each digit

Floats are similarly printed as ASCII digits, defaulting to two decimal places

- Bytes are sent as a single character
- Characters and strings are sent as is

Baud Rate

Baud is a measure of the number of bits transmitted per second.



First call the Serial.begin()

The function takes a single parameter: the desired communication speed (baud)

Use the same speed for the sending side and the receiving side.

Module 5: Digital Output

Objectives:

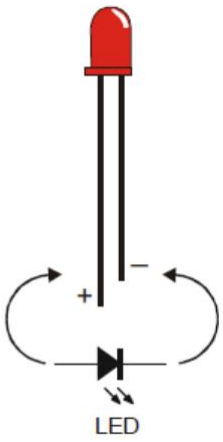
At the end of this module, the student should be able to:

Explain the interfacing of output devices to Arduino;

Create Arduino program.

Interfacing LEDs

A Light Emitting Diode or LED is basically a specialized type of PN junction diode capable of emitting light when conducting.



LED is an important output device for visual indication in embedded systems: ○

Status indicator lamps (e.g. Power ON)

- Direction signs
- Advertising
- Traffic signals

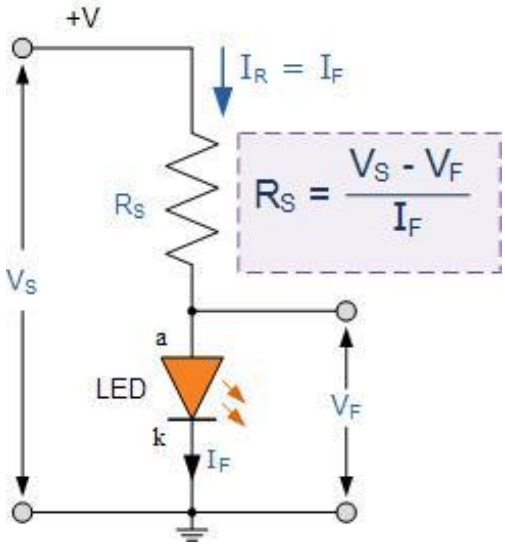
DC Characteristics and Operation

The **LED** has two DC characteristics:

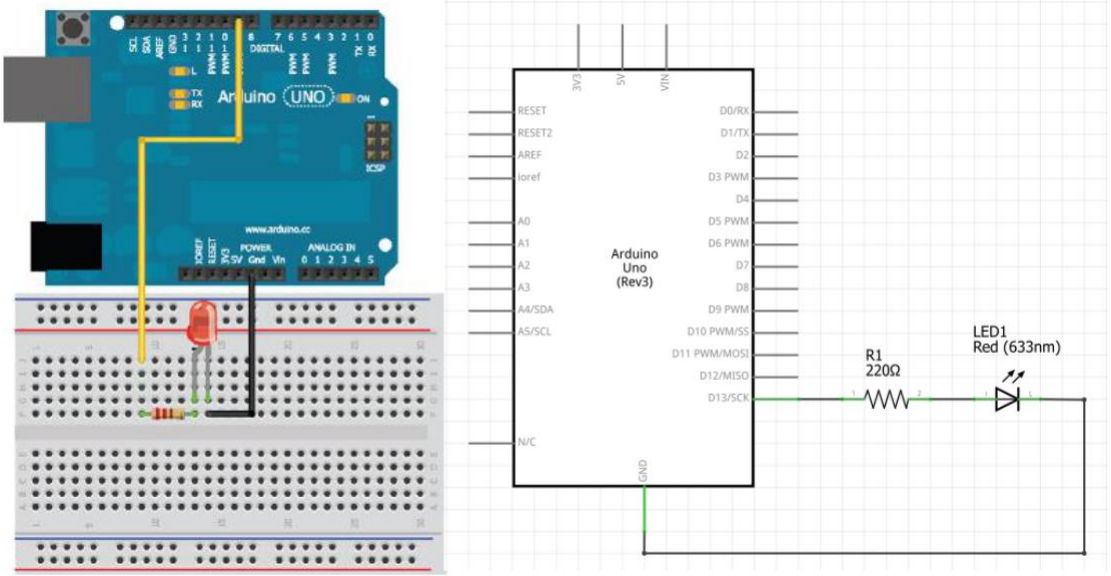
- **Forward voltage** (V_F) which is the minimum required voltage to turn on the LED
- **Full Drive/Forward Current** (I_F) which is the maximum allowed current across the LED (i.e. a current larger than I_F will burn the LED).

In order to protect the LED, a current limiting resistor R is usually used.

The value of this resistor is calculated using the following formula:



Interfacing to Arduino Board



Programming Digital Output

Programming digital inputs/outputs involves two operations:

- Configuring the desired pin as either input or output.
- Controlling the pin to read/write digital data.

Configuring Digital Output

By default, all Arduino pins are set to inputs.

To make a pin an output, it needs to first tell the Arduino how the pin should be configured (INPUT/OUTPUT).

In the Arduino programming language, the program requires two parts:

(1) the ***setup()*** function and

- (2) the ***loop()*** function.

The ***setup()*** function runs one time at the start of the program, and the ***loop()*** function runs over and over again.

Since it dedicates each pin to serve as either an input or an output, it is common practice to define all your pins as inputs or outputs in the setup.

```
   set pin 9 as a digital
output void setup() {
  pinMode(9, OUTPUT);
}
```

The pinMode Function

The ***pinMode()*** function configures the specified pin to behave either as an input or an output.

- **Syntax:**

  pinMode(pin, mode)

- **Parameters:**

  pin: the number of the pin whose mode you wish to set.

  mode: *INPUT or OUTPUT*

  Returns:

  None

Writing Digital Outputs

Because the ***loop()*** function runs over and over again, outputs are usually written in this function.

```
   set pin 9 high

void loop() {

  digitalWrite(9, HIGH);

}
```

The digitalWrite Function

The ***digitalWrite()*** function writes a HIGH or a LOW value to a digital pin.

If the pin has been configured as an OUTPUT with ***pinMode()***, its voltage will be set to the corresponding value: **5V** for HIGH, **0V** for LOW.

Syntax:

digitalWrite(pin, value)

- **Parameters:**
  pin: the pin number.
  value: *HIGH or LOW*
- **Returns:**
  None

Sample: Arduino program that turns **ON** an LED connected to **pin 13** for 1 second, then **OFF** for 1 second, repeatedly.

```
   Turns on an LED on for one second, then off for one second,
   repeatedly.
int led = 13;

   the setup routine runs once when you press
reset: void setup() {
     initialize the digital pin as an output.
  pinMode(led, OUTPUT);
}

   the loop routine runs over and over again
forever: void loop() {
  digitalWrite(led, HIGH);  // turn the LED on (HIGH)
  delay(1000);              // wait for a second
  digitalWrite(led, LOW);   // turn the LED off (LOW)
  delay(1000);              // wait for a second
}
```

The delay Function

The ***delay()*** function Pauses the program for the amount of time (in milliseconds) specified as parameter.

- **Note:** there are 1000 milliseconds in a second.

  Syntax:
  delay(ms)

- **Parameters:**

  ms: the number of milliseconds to pause (*unsigned long*)

- **Returns:**
  None

Module 6: Digital Input

Objectives:

At the end of this module, the student should be able to:

Explain how the Push button works;

Explain how to interface the Push button to Arduino;

Create program as inputs to Arduino;

Explain the program of Bouncy button.

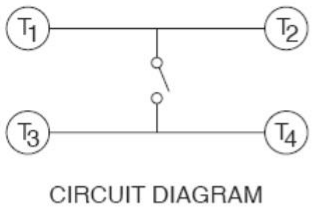
The Push Button

The simplest form of digital input is a push button.

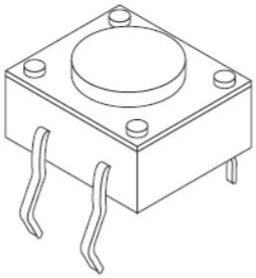
It can insert these directly into breadboard.

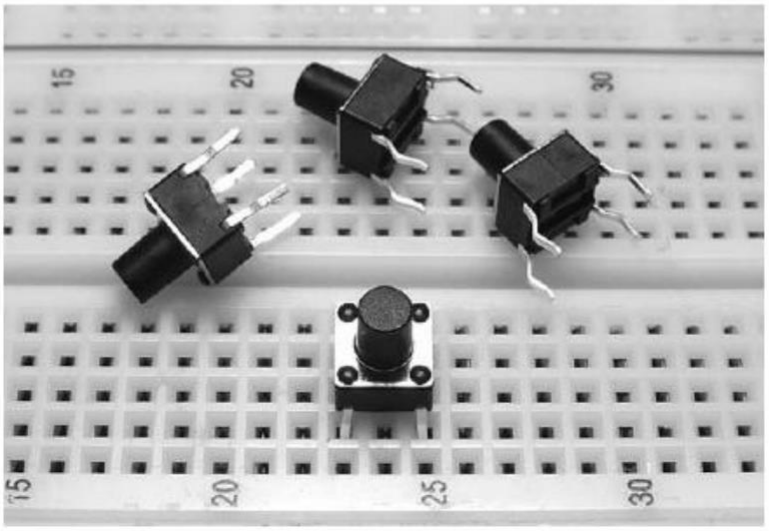
A push button allows a voltage or current to pass when the button is pressed.

Push buttons are generally used as **reset** switches and **pulse** generators.



CIRCUIT DIAGRAM





Interfacing Push Button to Arduino

A push button can be interface to a circuit using:

(1) **Pull-Up Resistors**: In this case, the output of the button is **normally HIGH** and when the button is pressed it generates a **LOW pulse** (Figure 1).

(2) **Pull-Down Resistors**: In this case, the output of the button is **normally LOW** and when the button is pressed it generates a **HIGH pulse** (Figure 2).

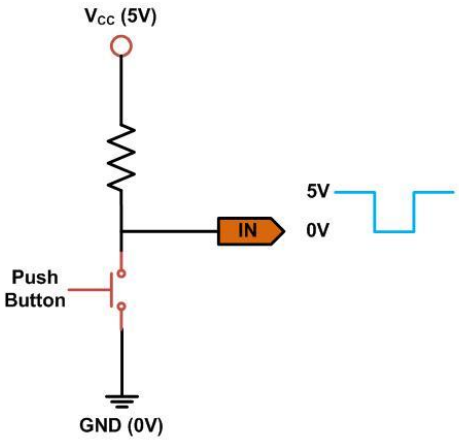


Figure 1: Push Button with Pull-Up Resistor

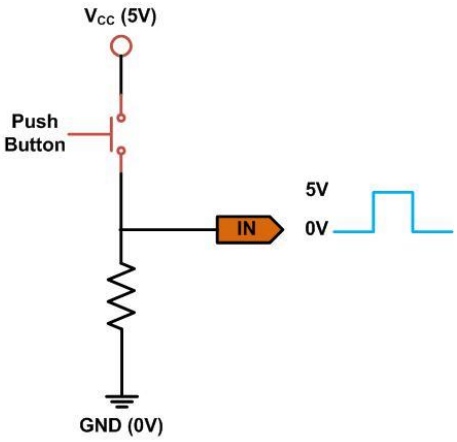
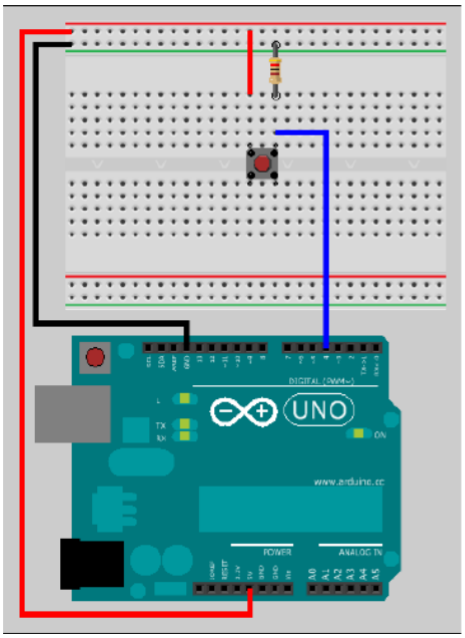
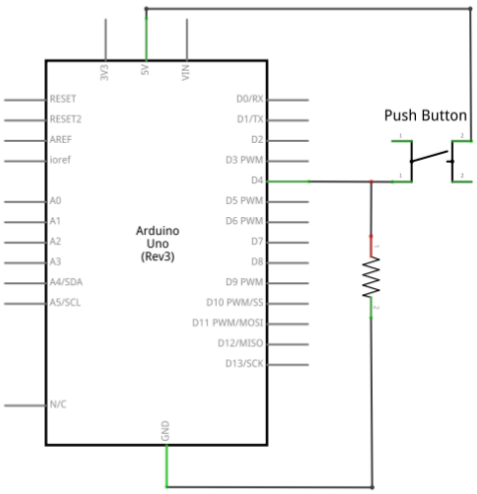


Figure 2: Push Button with Pull-Down Resistor



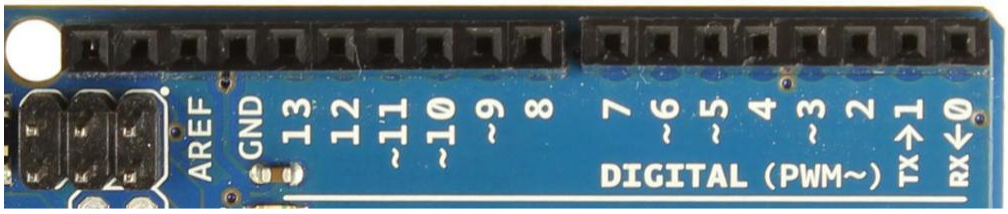


Programming Digital Inputs

Programming digital inputs/outputs involves two operations:

(1) Configuring the desired pin as either input or output.

- (2) Controlling the pin to read/write digital data.



Configuring Digital Inputs

Since it dedicates each pin to serve as either an input or an output, it is common practice to define all your pins as inputs or outputs in the setup.

```
   set pin 4 as a digital input

void setup() {

  pinMode(9, INPUT);

}
```

The pinMode Function

The ***pinMode()*** function configures the specified pin to behave either as an input or an output.

- **Syntax**:
  pinMode(*pin*, *mode*)

  Parameters:
  pin: the number of the pin whose mode you wish to set.
  mode: *INPUT or OUTPUT*

  Returns:
  None

Reading Digital Inputs

Because the ***loop()*** function runs over and over again, inputs are usually read in this function.

```
   read the state of pin 4

int pinState;

void loop() {

  pinState = digitalRead(4);

}
```

The digitalRead Function

The ***digitalRead()*** function reads the value from a specified digital pin, either HIGH or LOW.

Syntax:
digitalRead(pin)

- **Parameters:**
  pin: the number of the digital pin you want to read (*int*) .
- **Return:**
  HIGH or **LOW**

Sample: Arduino program that repeatedly reads the state of a **push button** connected to **pin 4** and display the message "Button Pressed" on the Arduino environment's built-in serial monitor whenever the button is pushed.

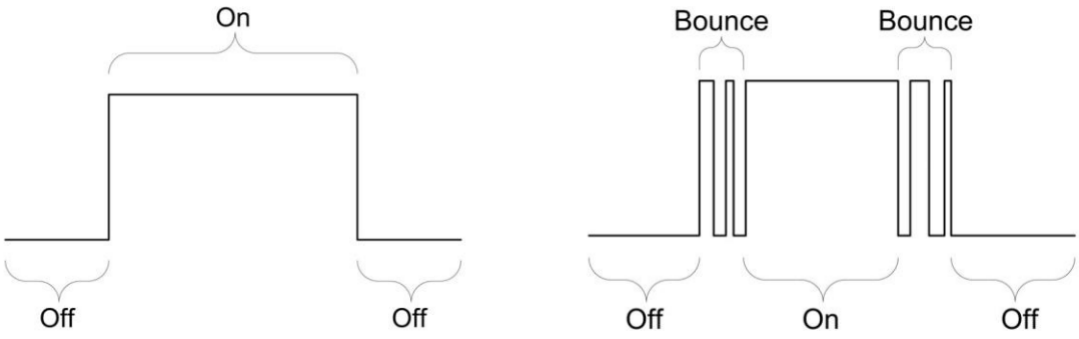
```
   Repeatedly Reads the state of a bush button connected to pin 4,
   and displays the message "Button Pressed" whenever the button
   is pushed.

int buttonState;

   the setup routine runs once when you press reset:
void setup() {
     initialize the digital pin as an input.
  pinMode(4, OUTPUT);
     initialize and start the serial port
  Serial.begin(9600);
}

   the loop routine runs over and over again forever:
void loop() {
  buttonState = digitalRead(4); // read the state of pin 4 if
  (buttonState == HIGH) { // if pin state is HIGH
  Serial.println("Button Pressed"); // display the message
  }
}
```

Switch Bounce

Push buttons exhibit a phenomenon called ***switch bounce***, or bouncing, which refers to a button's tendency to turn on and off several times after being pressed only once by the user.

This phenomenon occurs because the metal contacts inside a push button are so small that they can vibrate after a button has been released, thereby switching on and off again very quickly.

Switch Bounce can cause wrong behavior of digital circuits due to reading multiple values for a single press of the button.



Ideal Switch (No Bounce) Switch Bounce

Switch Debounce

There are two approaches for cleaning up switch bounce: ○

(1) **Hardware Debounce**

○ (2) **Software Debounce**

It is relatively straightforward to deal with switch bounce problem in software.

Hence, we will cover only software switch debouncing.

Software Switch Debounce

Basic Idea:

look for a button state change, wait for the bouncing to finish, and then reads the switch state again.

This software logic can be expressed as follows:

1. Store a previous button state and a current button state (initialized to LOW). ○ **2.** Read the current button state.

○ **3.** If the current button state differs from the previous button state, wait **5ms** because the button must have changed state.

○ **4.** After **5ms**, reread the button state and use that as the current button state. ○ **5.** Set the previous button state to the current button state.

○ **6**. Return to step 2.

```
// Modify previous program to debounce the switch
int buttonState; // the current reading from the input pin int
previousButtonState = LOW; // the previous reading from the input pin
long lastDebounceTime = 0;     // the last time the output pin was toggled
long debounceDelay = 50; // the debounce time; void
setup() {
  pinMode(4, OUTPUT);
  Serial.begin(9600);
}
void loop() {
     read the state of the switch into a local
  variable: int reading = digitalRead(4);
     If the switch changed, due to noise or
  pressing: if (reading != lastButtonState) {
       reset the debouncing timer
    lastDebounceTime = millis();
  }
  if ((millis() - lastDebounceTime) > debounceDelay) {
       the reading has been there for longer than the debounce delay, so take
    it if (reading != buttonState) {
      buttonState = reading;
      if (buttonState == HIGH) {            // if pin state is HIGH
         Serial.println("Button Pressed");    // display the message
      }
    }
  }
  // save the reading
  lastButtonState = reading;
}
```

The millis() Function

The ***millis()*** function returns the number of **milliseconds** since the Arduino board began running the current program.

○ This number will overflow (go back to zero), after approximately **50 days**.

○ **Syntax:**

millis()

○ **Parameters:**

None.

○ **Return:**

Number of milliseconds since the program started (*unsigned long*)

Module 7: Enhanced Digital Output

Objectives:

At the end of this module, the student should be able to:

Explain the 7-segment display;

Discuss types of 7-segment display and how it works;

Explain how to display digital digits in 7-segment display;

Discuss the circuit connection of the 7-segment display;

Explain how to display digit number to 7-segment display;

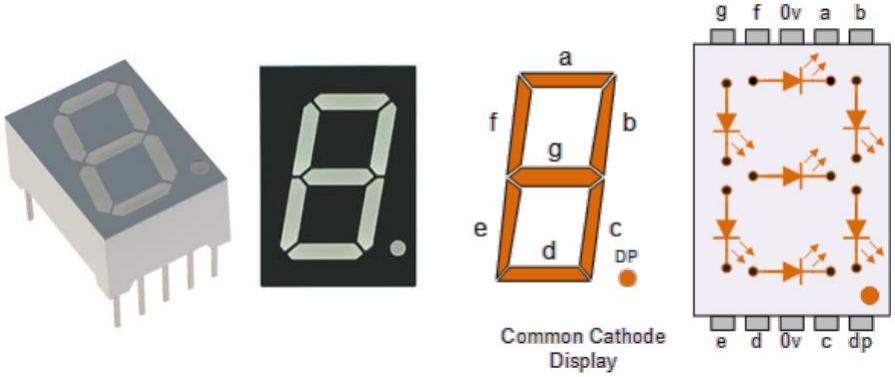
Discuss how to interface the 7-segment display to Arduino;

Discuss how to program the 7-segment display using Arduino.

7-Segment Display

The **7-segment display**, consists of **seven LEDs** arranged in a rectangular fashion.

Each of the seven LEDs is called a **segment**.



Each one of the seven LEDs in the display is given a positional segment which is controlled by one **pin**.

These LED **pins** are labeled **a**, **b**, **c**, **d**, **e**, **f**, and **g** representing each individual LED.

The other LED pins are connected together and wired to form a **common pin**.

By forward biasing the appropriate pins of the LED segments, some segments will be light and others will be dark allowing the desired character pattern of the number to be generated on the display.

This allows to display each of the ten decimal digits **0** through to **9** on the same 7-segment display.

The displays **common pin** is generally used to identify which type of 7-segment display it is.

As each LED has two connecting pins, one called the "**Anode**" and the other called the "**Cathode**".

Therefore , there are two types of LED 7-segment display:

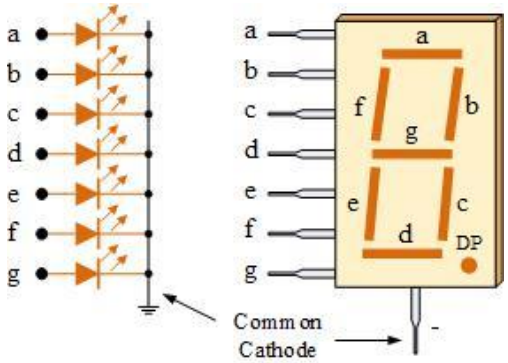
(1) **Common Cathode (CC)**

- (2) **Common Anode** (**CA**)

Common Cathode

In the common cathode display, all the cathode connections of the LED segments are joined together to **logic "0"** or **ground**.

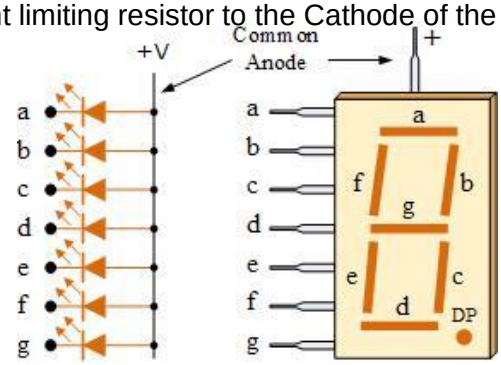
The individual segments are **illuminated** by application of a **"HIGH"**, or **logic "1"** signal via a current limiting resistor to forward bias the individual Anode terminals (a-g).



Common Anode

In the common anode display, all the anode connections of the LED segments are joined together to **logic "1"**.

The individual segments are **illuminated** by applying a ground, **logic "0"** or **"LOW"** signal via a suitable current limiting resistor to the Cathode of the particular segment (a-g).

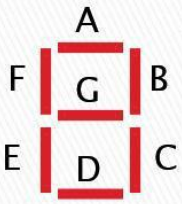


Displaying Digital Digits

Depending upon the decimal digit to be displayed, the particular set of LEDs is forward biased.

The various digits from 0 through 9 can be displayed using a 7-segment display.

Decimal Digit	G	F	E	D	C	B	A
0	0	1	1	1	1	1	1
1	0	0	0	0	1	1	0
2	1	0	1	1	0	1	1
3	1	0	0	1	1	1	1
4	1	1	0	0	1	1	0
5	1	1	0	1	1	0	1
6	1	1	1	1	1	0	1
7	0	0	0	0	1	1	1
8	1	1	1	1	1	1	1
9	1	1	0	1	1	1	1



Driving a 7-Segment Display

A 7-segment display can be thought of as a single display, it is still **seven individual LEDs** within a single package and as such these LEDs need protection from over-current.

LEDs produce light only when it is forward biased with the amount of light emitted being **proportiona**l to the **forward current**. This means that an LEDs light intensity increases in an approximately linear manner with an increasing current. So this **forward current** must be **controlled** and **limited** to a safe value by an **external resistor** to prevent damaging the LED segments.

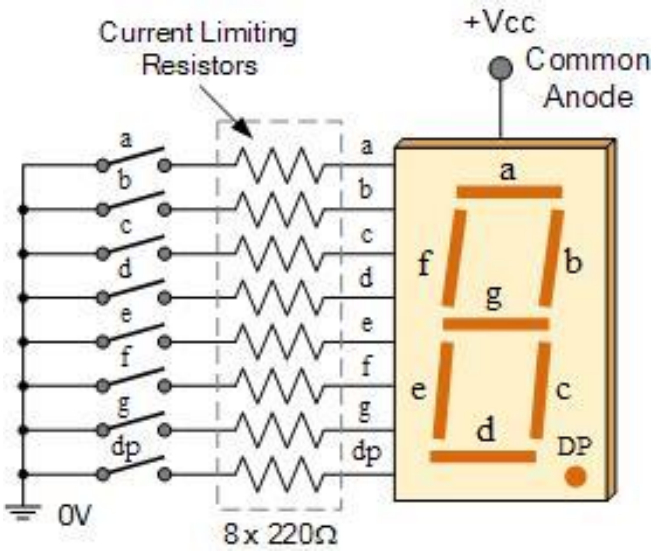
The forward **voltage drop** across a red LED segment is very low at about **2-to-2.2 volts**. To illuminate correctly, the LED segments should be connected to a voltage source in excess of this forward voltage value with a **series resistance** used to **limit** the **forward current** to a desirable value.

Typically for a standard red colored 7-segment display, each LED segment can draw about **15mA** to illuminated correctly, so on a **5 volt** digital logic circuit,

- the value of the current limiting resistor would be about **200Ω (5v – 2v)/15mA**,

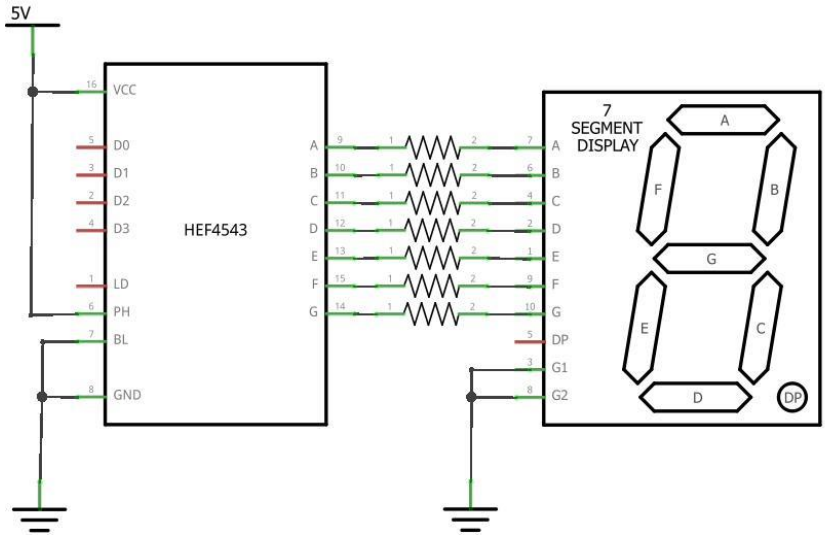
or - **220Ω** to the **nearest** higher preferred value.

So to understand how the segments of the display are connected to a 220Ω current limiting resistor consider the circuit below.



BCD to 7-Segment Decoder

The 7-segment Display is usually driven by a special type of integrated circuit (IC) known as a BCD to 7-segment decoder.



The 4543 BCD to 7-Segment Decoder

The 4543 is a BCD to 7-segment latch/decoder/driver.

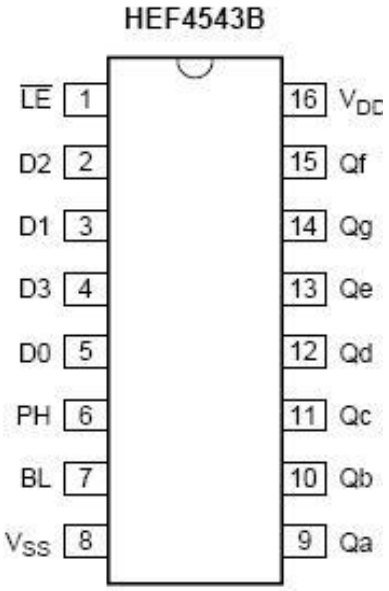
Its function is to convert a BCD digit into signals which will drive a 7-segment display.

Inputs:

four address inputs (**D0** to **D3**),
an active LOW latch enable input (**LE**),
an active HIGH blanking input (**BL**),
an active HIGH phase input (**PH**)

Outputs:

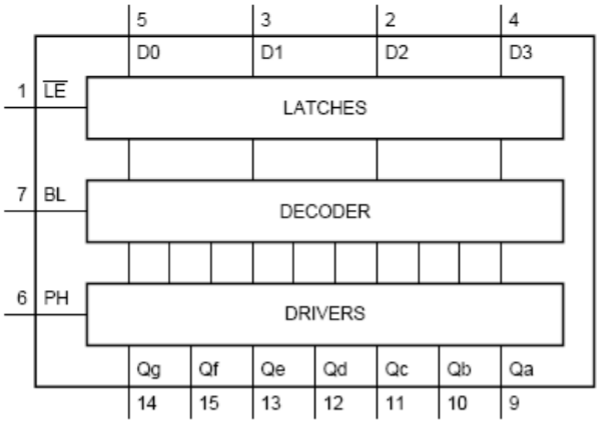
seven buffered segment outputs (**Qa- Qg**).



The functions of the three control inputs PH, BL and LE are as follows: ○

The **phase (PH)** input is used to reverse the function table phase.

- ○ The **blanking (BL)** input is used to blank (turn off) the display.
- ○ The **latch enable (LE)** input is used to store a BCD code.

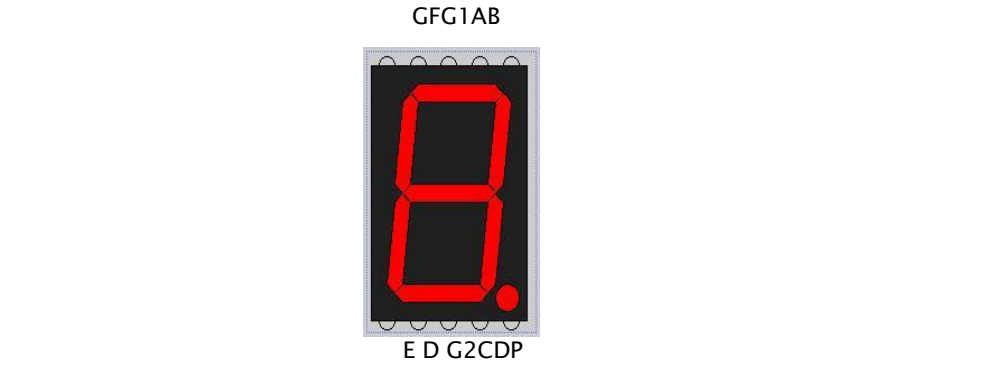


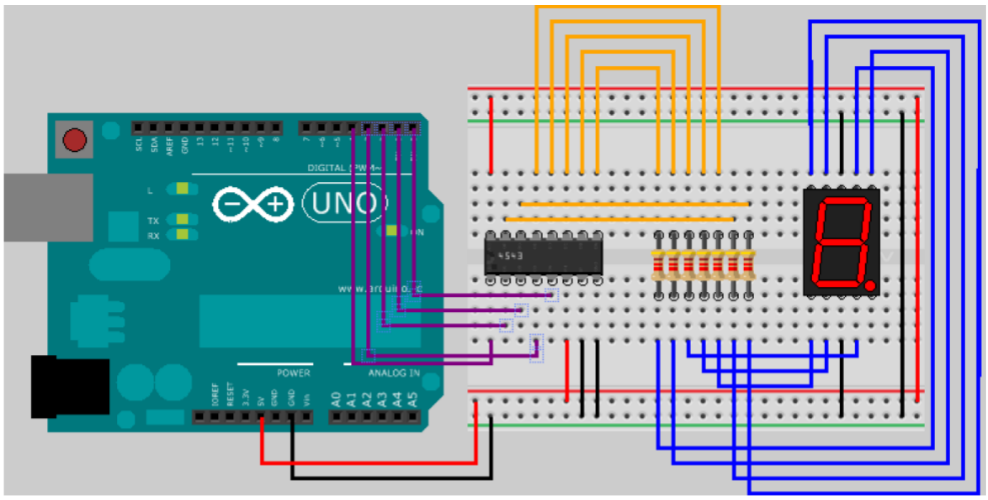
Inputs							Outputs							Display
LE	BL	PH	Dd	Dc	Db	Da	Qa	Qb	Qc	Qd	Qe	Qf	Qg	
X	1	1	X	X	X	X	0	0	0	0	0	0	0	
1	0	1	0	0	0	0	1	1	1	1	1	1	0	0
1	0	1	0	0	0	1	0	1	1	0	0	0	0	1
1	0	1	0	0	1	0	1	1	0	1	1	0	1	2
1	0	1	0	0	1	1	1	1	1	1	0	0	1	3
1	0	1	0	1	0	0	0	1	1	0	0	1	1	4
1	0	1	0	1	0	1	1	0	1	1	0	1	1	5
1	0	1	0	1	1	0	1	0	1	1	1	1	1	6
1	0	1	0	1	1	1	1	1	1	0	0	0	0	7
1	0	1	1	0	0	0	1	1	1	1	1	1	1	8
1	0	1	1	0	0	1	1	1	1	1	0	1	1	9
1	0	1	1	0	1	X	0	0	0	0	0	0	0	
1	0	1	1	1	X	X	0	0	0	0	0	0	0	
0	0	1	X	X	X	X	N.C	N.C	N.C	N.C	N.C	N.C	N.C	N.C
As Above		0	As Above				Inverse of Above							As Above

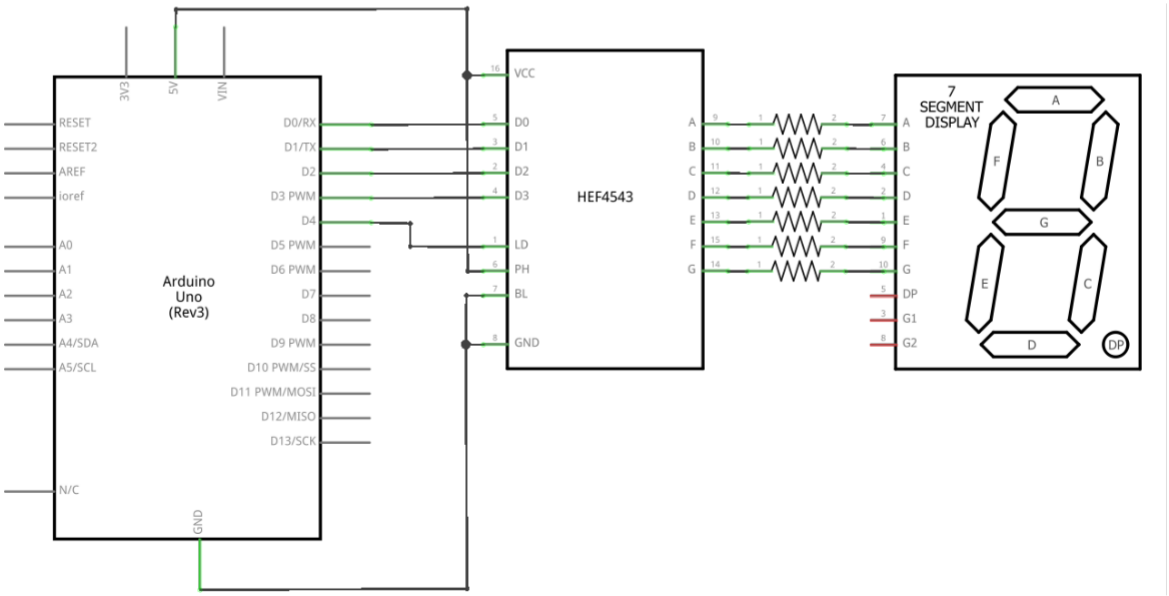
N.C = No Change

Interfacing 7-Segment Display to Arduino

The 7-Segment Display can be interfaced to the Arduino Uno Board.







BCD Counter Sample Program

A program to display a BCD counter (counting from 0 – to – 9) with a period of 1 second.

The required program consists of three main parts:

(1) Output Pins Configuration

- (2) Counter Logic
- (3) BCD Digit Display

Output Pins Configuration

In this part, pins 0 to 4 are defined as output pins to control illumination of the 7 segments:

- **Pins 0-3:** control the **address inputs** (**Dd**, **Dc**, **Db**, **Da**) of the 4543 decoder
- **Pin 4**: control the **Latch Enable** (**LE**) of the 4543 decoder.

```
public void setup(){
        configure pins 0-4 as output pins
    for (int i = 0; i < 5; i++) {
        pinMode(i, OUTPUT);
    }
}
```

Counter Logic

In this part, a FOR loop is used to implement the one-digit BCD counter logic with 1 second delay.

```
public void loop(){
        for every bcd digit (0-9) do the following
    for (int digit = 0; digit < 10; digit++) {
            (1) call displayDigit function
        displayDigit(digit);
            (2) wait for 1 second
         delay(1000);
    }
}
```

BCD Digit Display

In this part, a function is used to display the given **decimal digit** (**BCD code**) as follows: ○ (1) Set the **LE** line of the 4543 decoder **HIGH** to write the received BCD code into the 4543 decoder.

○ (2) for each bit in the received BCD code do the following:
(a) Determine the value of the bit (**0** or **1**).
(b) Set the corresponding output pin (HIGH or LOW) accordingly.

(3) Set the **LE** line of the 4543 decoder **LOW** to **latch** (**store**) the received BCD code into the 4543 decoder

```
   displays a given BCD digit on a 7-segment display
void displayDigit(int digit) {
       (1) set LE line high to write the received BCD code into the decoder.
    digitalWrite(4, HIGH);
       (2) for each bit in the received BCD code do the following
    for (int i = 0; i < 4; i++) {
           (a) determine the value of the bit (0/1)
        int bitValue = bitRead(digit, i);
           (b) set the corresponding output pin (0-3) (HIGH/LOW) accordingly
        digitalWrite(i, bitValue);
    }
       (3) Set the LE line low to store the received BCD code into the decoder
    digitalWrite(4, LOW);
}
```

The bitRead() Function

The ***bitRead()*** function reads a bit of a number. ○

Syntax:

bitRead(x, n)

Parameters:

x: the number from which to read.

n: which bit to read, starting at 0 for the least-significant (rightmost) bit

○ **Returns:**

the value of the bit (0 or 1).

Module 8: Enhanced Digital Input

Objectives:

At the end of this module, the student should be able to:

Explain the phone keypad and how it works;

Discuss how the phone keypad performs as input;

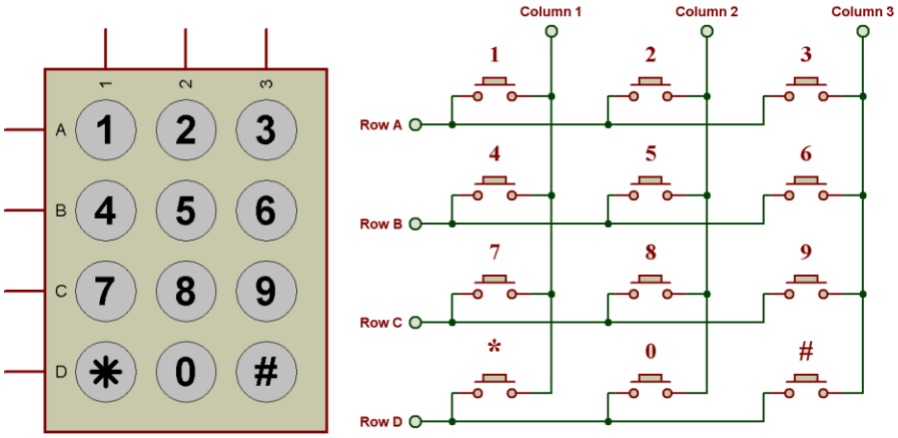
Explain how to interface the phone keypad into Arduino;

Create a program that read phone keypad and display to output.

The Phone Keypad

The phone keypad is a 4x3 key matrix consisting of 4 rows by 3 columns. ○

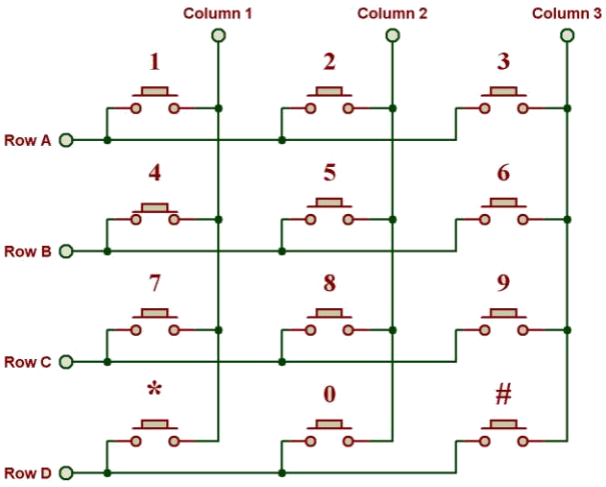
When a key is pressed it connects its column pin to its row pin.



Scanning the Keypad

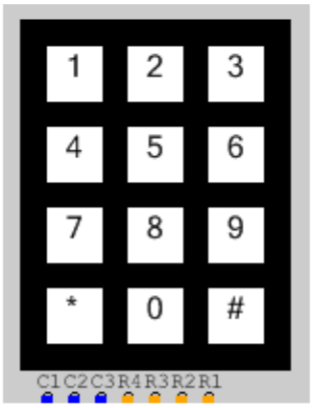
When interfacing the keypad to a microcontroller, this can be done using a very simple procedure called "key scanning".

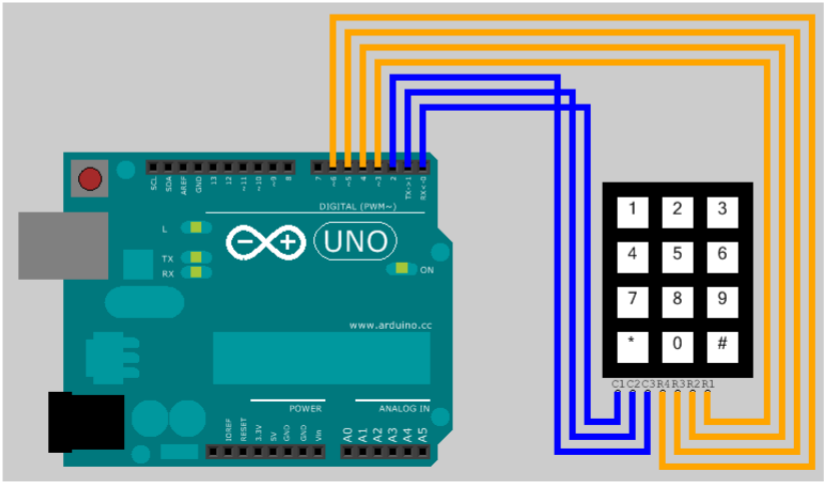
- **(1)** Apply Logic 1 on column 1 and Logic 0 on remaining columns.
- **(2)** Scan the logic signals on all rows A to D.
- **(3)** If a Logic 1 is read on one row then this indicates that the corresponding key (i.e. 1, 4, 7 or *) is pressed.
- **(4)** Repeat steps 1-3 for column 2 to determine if any of the keys (2, 5, 8 and 0) is pressed.
- **(5)** Repeat steps 1-3 for column 3 to determine if any of the keys (3, 6, 9 and #) is pressed.



Interfacing the Keypad

The phone keypad can be interfaced to the Arduino Uno Board.





Design Example

Interface a phone keypad to the Arduino Uno Board and write a simple program to display the value of pressed key on the Arduino environment's built-in serial monitor.

The code that will scan the keys and display the value of the pressed key on the Arduino environment's built-in serial monitor consists of the following main pars:

- **(1)** Global variables declarations.
- **(2)** Input/output pins configurations.
- **(3)** Key scan logic.
- **(4)** Key value display logic.

Global Variables Declaration

In this part, an array is declared to hold character values corresponding to the buttons of the keypad.

```
   character values corresponding to the keypad
buttons char [][] keypad = {
                        {'1', '2', '3'},
                        {'4', '5', '6'},
                        {'7', '8', '9'},
                        {'*', '0', '#'}
                    };
```

Input/Output Pin Configuration

In this part, we configure the digital pins of the Arduino board to scan the keypad:

- Configure pins 0-2 as output pins to control COL1-COL3 of the keypad.
- Configure pins 3-6 as input pins to read ROW1-ROW4 of the keypad.

```
   set pins 0-2 as outputs to control COL1-3 of the keypad
for (int col = 0; col < 3; col++) {
        pinMode(col, OUTPUT);
}

   set pins 3-6 as inputs to read ROW1-4 of the keypad
    for (int row = 3; row < 7; row++) {
        pinMode(row, INPUT);
}
```

Key Scan Logic

In this part, scan the keypad and return the character value of the pressed key or NULL if no key is pressed.

- (1) Set key value to null
- (2) For each pin 0-2 connected to COL1-3 do the following
  (a) Set the column HIGH
  (b) For each pin 3-6 connected to ROW1-4 do the following
  Read the state of the row
  If the state of the row is HIGH the do then set key value to the character corresponding to the pressed button
  (c) Set the column back to LOW
  (3) Return the key value

```
char getKey() {
       set key value to null
    char keyValue = '\0';
       a variable to read the row
    state int rowState;
       for each pin 0-2 connected to COL1-3 do the
    following for (int col = 0; col < 3; col++) {
           set the column HIGH
        digitalWrite(col, HIGH);
           for each pin 3-6 connected to ROW1-4 do the
        following for (int row = 3; row < 7; row++) {
               read the state of the row
            rowState = digitalRead(row);
               if the state of the row is HIGH the do the
            following if (rowState == HIGH) {
                   set key value to the character
                   corresponding to the pressed button
                keyValue = (keypad[row - 3][col]);
            }
        }
           set the column back to
        LOW digitalWrite(col, LOW);
    }
       return the key value
    return (keyValue);
}
```

Display Key Value Logic

In this part, use the getKey() function to get the value of the pressed key: ○

Get key value using the function getKey().

- ○ If the returned key value is not NULL then display the value on the serial monitor

```
   get key value
char key = getKey();

   if the key value is not NULL then
if (key != '\0') {
          display the key value on the serial monitor
        Serial.println(key);
}
```

Module 9: Interfacing the LCD Display

Objectives:

At the end of this module, the student should be able to:

Explain the LCD and how it works;

Discuss the pin configuration of LCD;

Discuss how the LCD performs as output;

Explain how to interface the LCD into Arduino;

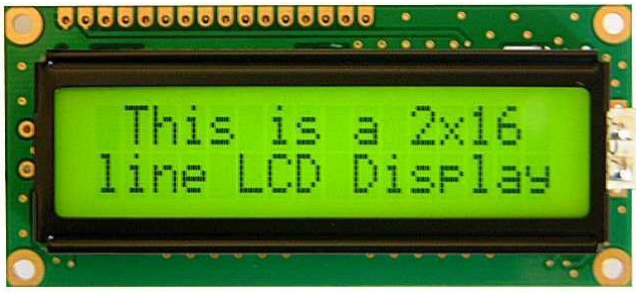
Create a program that read input and display to LCD.

LCD

LCD stands for “Liquid Crystal Displays”.

LCDs are used to display numbers, characters, symbols and graphics.

They are used in many embedded systems such as, digital clocks, watches, microwave ovens, CD players etc.



LCD Features

As compared to LEDs, LCDs have the following features:

The ability to display numbers, characters, symbols and graphics (LEDs are limited to numbers and a few characters)

Incorporation of a refreshing controller, thereby relieving the CPU of the task of refreshing the LCD (LEDs must be refreshed by the CPU to keep displaying the data)

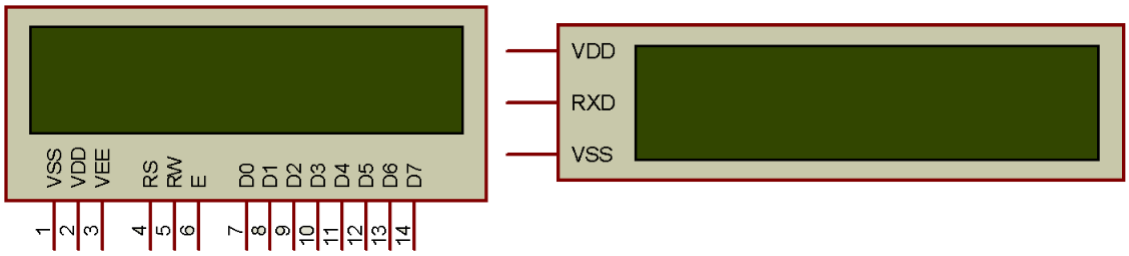
Ease of programming for characters and graphics.

LCD Types

There are basically two types of LCDs as far as the interfacing method is concerned:

(1) **Parallel LCDs** are the most commonly used ones and they are connected to microcontrollers using four to eight data lines and some control lines.

(2) **Serial LCDs** are connected to microcontrollers using only one data line and data is sent to the LCD using the RS232 serial communications protocol. Serial LCDs are easier to use, but they cost a lot more than the parallel ones.



LCD Sizes

Depending upon the requirements, LCDs come in different sizes:

Displays with 8, 16, 20, 24, 32 and 40 characters are available. ○ The row size can be selected as 1, 2 or 4.

○ Display types are identified by specifying the number of rows and number of characters per row for example:

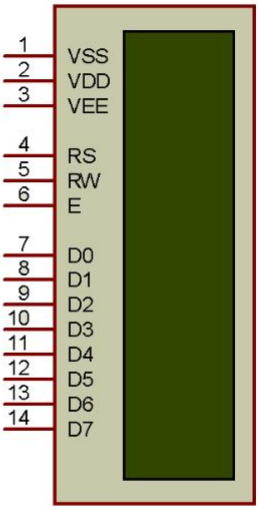
a 1 × 16 display has one row with 16 characters, and

a 4 × 16 display has 4 rows and 16 characters on each row

LCD Pin Configurations

Table 12-1: Pin Descriptions for LCD

Pin	Symbol	I/O	Description
1	V_{SS}	--	Ground
2	V_{CC}	--	+5V power supply
3	V_{EE}	--	Power supply to control contrast
4	RS	I	RS = 0 to select command register, RS = 1 to select data register
5	R/W	I	R/W = 0 for write, R/W = 1 for read
6	E	I	Enable
7	DB0	I/O	The 8-bit data bus
8	DB1	I/O	The 8-bit data bus
9	DB2	I/O	The 8-bit data bus
10	DB3	I/O	The 8-bit data bus
11	DB4	I/O	The 8-bit data bus
12	DB5	I/O	The 8-bit data bus
13	DB6	I/O	The 8-bit data bus
14	DB7	I/O	The 8-bit data bus



V_{DD}: provides +5V

V_{SS}: provides ground

V_{EE}: controls LCD contrast

RS (Register Select):

There are two very important registers inside the LCD.

The RS pin is used for their selection as follows.

If **RS = 0**, the **instruction command code register** is selected, allowing the user to send a command (e.g. clear display).

If **RS = 1**, the **data register** is selected, allowing the user to send data to be displayed on the LCD.

R/W (Read/Write):

R/W input allows the user to write information to the LCD or read information from it.

R/W = 1 when **reading**;

- **R/W = 0** when **writing**.

E (Enable):

The enable pin is used by the LCD to latch information presented to its data pins. When data is supplied to data pins, a high-to-low pulse must be applied to this pin in order for the LCD to latch in the data present at the data pins.
This pulse must be a minimum of 450 ns wide.

D0–D7:

The 8-bit data pins, D0–D7, are used to send information to the LCD or read the contents of the LCD's internal registers.

To display letters and numbers, we send ASCII codes for the letters A–Z, a–z, and numbers 0–9 to these pins while making RS = 1.
There are also instruction command codes that can be sent to the LCD

Table 12-2: LCD Command Codes

Code (Hex)	Command to LCD Instruction Register
1	Clear display screen
2	Return home
4	Decrement cursor (shift cursor to left)
6	Increment cursor (shift cursor to right)
5	Shift display right
7	Shift display left
8	Display off, cursor off
A	Display off, cursor on
C	Display on, cursor off
E	Display on, cursor blinking
F	Display on, cursor blinking
10	Shift cursor position to left
14	Shift cursor position to right
18	Shift the entire display to the left
1C	Shift the entire display to the right
80	Force cursor to beginning of 1st line
C0	Force cursor to beginning of 2nd line
38	2 lines and 5x7 matrix

Note: This table is extracted from Table 12-4.

Operation Modes

The LCD display can be operated using one of two operation modes: ○

(1) **8-bit Mode:**

data/code is transferred 8-bits at a time over data lines D0-D7 ○ (2) **4-bit Mode:**

data is transferred over data lines D4-D7 (D0-D3 may float).

8-bit data/code is sent 4-bits at a time, with the most significant 4-bits sent first.

Interfacing the 8-bit LCD

One possible interface of the LCD display to the Arduino Uno Board in **8-bit mode** is given below:

- **VDD** = 5.0V (VCC)
- **VSS** = 0.0V (GND)
- **VEE** = Float (Not Connected)
- **R/W**= 0.0V (GND)
- It is free to connect the remaining pins of the LCD to the digital I/O pins (0-13) of the Arduino Board any order. One possible configuration is given below:

E = pin 11

- **RS** = pin 10

D0 = pin 7

D1 = pin 6

D2 = pin 5

D3 = pin 4

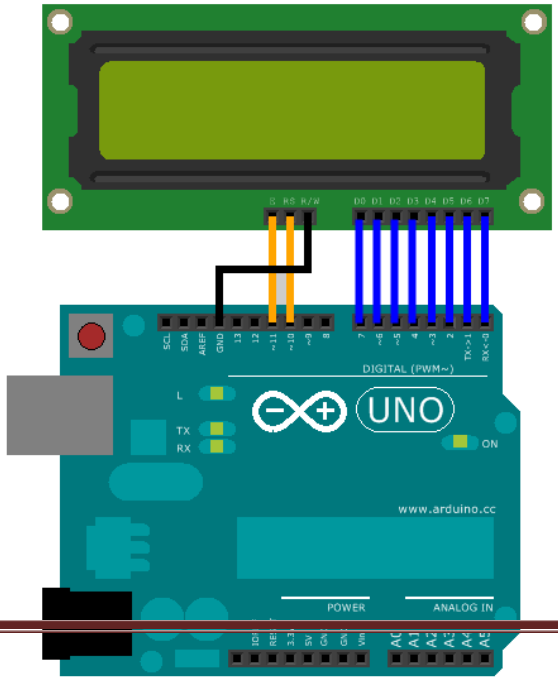
D4 = pin 3

D5 = pin 2

D6 = pin 1

D7 = pin 0

LCD Display Interface (8-bit Mode)



Interfacing the 4-bit LCD

One possible interface of the LCD display to the Arduino Uno Board in **4-bit mode** is given below:

- **VDD** = 5.0V (VCC)
- **VSS** = 0.0V (GND)
- **VEE** = Float (Not Connected)
- **R/W**= 0.0V (GND)
- It is free to connect the remaining pins of the LCD to the digital I/O pins (0-13) of the Arduino Board any order you like. One possible configuration is given below:

  E = pin 11

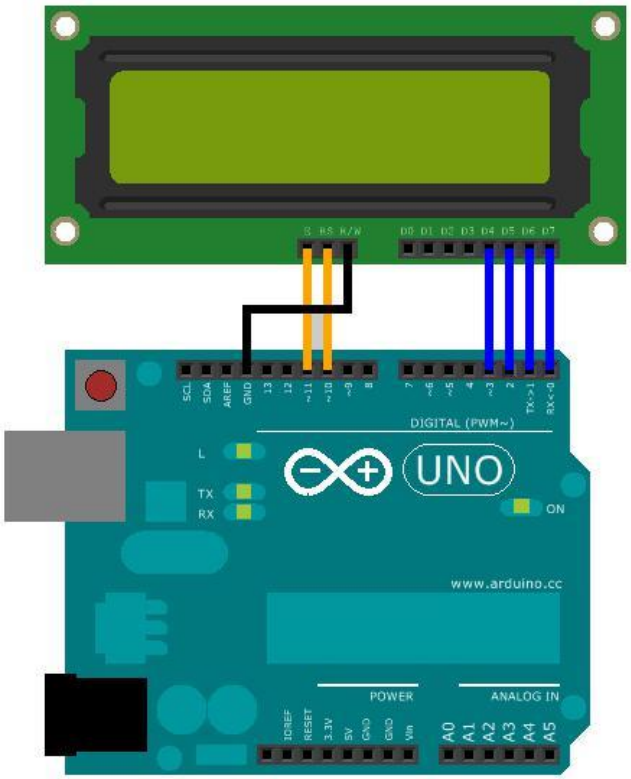
  RS = pin 10

  D4 = pin 3

  D5 = pin 2

  D6 = pin 1

  D7 = pin 0



Controlling the LCD

The ***LiquidCrystal*** library allows an Arduino board to control LCD displays based on the Hitachi HD44780 (or a compatible) chipset.

The library works in either

4-bit mode or

8-bit mode

It uses

4 or **8** data lines

the **RS** control line,

the **enable** control line, and, optionally,

- the **RW** control line.

Functions

The ***LiguidCrystal*** library supports the following functions: ○

begin()

- clear()
- home()
- setCursor()
- write()
- print() ○

cursor()

- noCursor()
- blink()
- noBlink() ○

display() ○

noDisplay()

- scrollDisplayLeft()
- scrollDisplayRight()
- autoscroll()
- noAutoscroll()
- leftToRight()
- rightToLeft()
- createChar()

The LiquidCrystal() Function

Description:

Creates a variable of type LiquidCrystal.

The display can be controlled using **4 or 8 data lines**.

If the former, omit the pin numbers for **d0** to **d3** and leave those lines unconnected.

The **RW** pin can be tied to ground instead of connected to a pin on the Arduino; if so, omit it from this function's parameters.

Syntax:

LiquidCrystal(rs, enable, d4, d5, d6, d7)

LiquidCrystal(rs, rw, enable, d4, d5, d6, d7)

LiquidCrystal(rs, enable, d0, d1, d2, d3, d4, d5, d6, d7)

LiquidCrystal(rs, rw, enable, d0, d1, d2, d3, d4, d5, d6, d7)

Parameters:

rs: the number of the Arduino pin that is connected to the RS pin on the LCD

rw: the number of the Arduino pin that is connected to the RW pin on the LCD (*optional*)

enable: the number of the Arduino pin that is connected to the enable pin on the LCD

d0, d1, d2, d3, d4, d5, d6, d7: the numbers of the Arduino pins that are connected to the corresponding data pins on the LCD.

d0, d1, d2, and d3: are optional; if omitted, the LCD will be controlled using only the four data lines (d4, d5, d6, d7).

```
LiguidCrystal lcd = new LiquidCrystal(11,10,3,2,1,0);
```

The begin() Function

Description:

Initializes the interface to the LCD screen, and specifies the dimensions (width and height) of the display.

begin() needs to be called before any other LCD library commands.

Syntax:

lcd.begin(*cols*, *rows*)

Parameters:

lcd: a variable of type *LiquidCrystal*

cols: the number of columns that the display has

- **rows:** the number of rows that the display has

```
LiguidCrystal lcd = new LiquidCrystal(11,10,3,2,1,0);

void setup() {
    lcd.begin(16,2);
}
```

The clear() Function

Description:

Clears the LCD screen and positions the cursor in the upper-left corner.

Syntax:

lcd.clear()

Parameters:

lcd: a variable of type *LiquidCrystal*

```
LiguidCrystal lcd = new LiquidCrystal(11,10,3,2,1,0);

void setup() {
    lcd.begin(16,2);
    lcd.clear();
}
```

The home() Function

Description:

Positions the cursor in the upper-left of the LCD.

That is, use that location in outputting subsequent text to the display.

To clear the display, use the clear() function instead.

Syntax:

lcd.home()

Parameters:

lcd: a variable of type *LiquidCrystal*

```
LiguidCrystal lcd = new LiquidCrystal(11,10,3,2,1,0);

void setup() {
    lcd.begin(16,2);
    lcd.home();
}
```

The setCursor() Function

Description:

Position the LCD cursor; that is, set the location at which subsequent text written to the LCD will be displayed.

Syntax:

lcd.setCursor(*col*, *row*)

Parameters:

lcd: a variable of type *LiquidCrystal*

col: the column at which to position the cursor (with 0 being the first column)

- **row:** the row at which to position the cursor (with 0 being the first row)

```
LiguidCrystal lcd = new LiquidCrystal(11,10,3,2,1,0);

void setup() {
    lcd.begin(16,2);
    lcd.setCursor(4,1);
}
```

The write() Function

Description:

Write a character to the LCD.

Syntax:

lcd.*write*(*data*)

Parameters:

lcd: a variable of type LiquidCrystal

data: the character to write to the display

```
LiguidCrystal lcd = new LiquidCrystal(11,10,3,2,1,0);

void setup() {
    lcd.begin(16,2);
    lcd.write('a');
}
```

The print() Function

Description:

Prints text to the LCD.

Syntax:

lcd.*print*(*text*)

Parameters:

lcd: a variable of type LiquidCrystal

text: the string of text to print to the display

```
LiguidCrystal lcd = new LiquidCrystal(11,10,3,2,1,0);

void setup() { lcd.begin(16,2);
    lcd.print("hello, world!");

}
```

The cursor() Function

Description:

Display the LCD cursor: an underscore (line) at the position to which the next character will be written.

Syntax:

lcd.cursor()

Parameters:

lcd: a variable of type LiquidCrystal

```
LiguidCrystal lcd = new LiquidCrystal(11,10,3,2,1,0);

void setup() {
    lcd.begin(16,2);
    lcd.cursor();
}
```

The noCursor() Function

Description:

Hides the LCD cursor.

Syntax:

lcd.noCursor()

Parameters:

lcd: a variable of type LiquidCrystal

```
LiguidCrystal lcd = new LiquidCrystal(11,10,3,2,1,0);

void setup() {
    lcd.begin(16,2);
    lcd.noCursor();
}
```

The blink() Function

Description:

Display the blinking LCD cursor.

If used in combination with the cursor() function, the result will depend on the particular display.

Syntax:

lcd.blink()

Parameters:

lcd: a variable of type LiquidCrystal

```
LiguidCrystal lcd = new LiquidCrystal(11,10,3,2,1,0);

void setup() { lcd.begin(16,2);
    lcd.print("hello, world!");

}
void loop() {
  lcd.noblink();
  delay(500);
  lcd.blink();
}
```

The noBlink() Function

Description:

Turns off the blinking LCD cursor.

Syntax:

lcd.no*Blink*()

Parameters:

lcd: a variable of type LiquidCrystal

```
LiguidCrystal lcd = new LiquidCrystal(11,10,3,2,1,0);

void setup() { lcd.begin(16,2);
    lcd.print("hello, world!");

}
void loop() {
  lcd.noblink();
  delay(500);
  lcd.blink();
}
```

The display() Function

Description:

Turns on the LCD display, after it's been turned off with noDisplay(). This will restore the text (and cursor) that was on the display.

Syntax:

lcd.display()

Parameters:

lcd: a variable of type LiquidCrystal

```
LiguidCrystal lcd = new LiquidCrystal(11,10,3,2,1,0);

void setup() {
    lcd.begin(16,2);
    lcd.print("hello, world!");
}
void loop() {
  lcd.noDisplay();
  delay(500);
  lcd.display();
}
```

The noDisplay() Function

Description:

Turns off the LCD display.

Syntax:

lcd.noDisplay()

Parameters:

lcd: a variable of type LiquidCrystal

```
LiguidCrystal lcd = new LiquidCrystal(11,10,3,2,1,0);

void setup() { lcd.begin(16,2);
    lcd.print("hello, world!");

}
void loop() {
  lcd.noDisplay();
  delay(500);
  lcd.display();
}
```

The scrollDisplayLeft() Function

Description:

Scrolls the contents of the display (text and cursor) one space to the left.

Syntax:

lcd.scrollDisplayLeft()

Parameters:

lcd: a variable of type LiquidCrystal

```
LiguidCrystal lcd = new LiquidCrystal(11,10,3,2,1,0);

void setup() {
    lcd.begin(16,2);
    lcd.print("hello, world!");
}
void loop() {
  lcd.scrollDisplayLeft();
  delay(150);
}
```

The scrollDisplayRight() Function

Description:

Scrolls the contents of the display (text and cursor) one space to the right.

Syntax:

lcd.scrollDisplayRight()

Parameters:

lcd: a variable of type LiquidCrystal

```
LiguidCrystal lcd = new LiquidCrystal(11,10,3,2,1,0);

void setup() {
    lcd.begin(16,2);
    lcd.print("hello, world!");
}
void loop() {
  lcd.scrollDisplayRight();
  delay(150);
}
```

The autoscroll() Function

Description:

Turns on automatic scrolling of the LCD.

This causes each character output to the display to push previous characters over by one space.

If the current text direction is left-to-right (the default), the display scrolls to the left; if the current direction is right-to-left, the display scrolls to the right.

This has the effect of outputting each new character to the same location on the LCD.

Syntax:

lcd.autoscroll()

Parameters:

lcd: a variable of type LiquidCrystal

```
LiguidCrystal lcd = new LiquidCrystal(11,10,3,2,1,0);

void setup() {
    lcd.begin(16,2);
    lcd.autoscroll();
    lcd.print("hello, world!");
}
```

The noAutoscroll() Function

Description:

Turns off automatic scrolling of the LCD.

Syntax:

lcd.noAutoscroll()

Parameters:

lcd: a variable of type LiquidCrystal

```
LiguidCrystal lcd = new LiquidCrystal(11,10,3,2,1,0);

void setup() {
    lcd.begin(16,2);
    lcd.noAutoscroll();
    lcd.print("hello, world!");
}
```

The leftToRight() Function

Description:

Set the direction for text written to the LCD to left-to-right, the default.

This means that subsequent characters written to the display will go from left to right, but does not affect previously-output text.

Syntax:

lcd.leftToRight()

Parameters:

lcd: a variable of type LiquidCrystal

```
LiguidCrystal lcd = new LiquidCrystal(11,10,3,2,1,0);

void setup() {
    lcd.begin(16,2);
    lcd.leftToRight();
}
```

The rightToLeft() Function

Description:

Set the direction for text written to the LCD to right-to-left.

This means that subsequent characters written to the display will go from right to left, but does not affect previously-output text.

Syntax:

lcd.rightToLeft()

Parameters:

lcd: a variable of type LiquidCrystal

```
LiguidCrystal lcd = new LiquidCrystal(11,10,3,2,1,0);

void setup() {
    lcd.begin(16,2);
    lcd.rightToLeft();
}
```

The createChar() Function

Description:

Create a custom character for use on the LCD. Up to 8 characters of 5x8 pixels are supported.

The appearance of each custom character is specified by an array of eight bytes, one for each row.

Syntax:

lcd.createChar(num, data)

Parameters:

lcd: a variable of type LiquidCrystal

num: which character to create (0 to 7)

- **data:** the character's pixel data

```
LiquidCrystal lcd(12, 11, 5, 4, 3, 2);

byte smiley[9] = { B00000, B10001, B00000,
                   B00000, B10001, B01110, B00000};
void setup() {   lcd.createChar(0, smiley);
                 lcd.begin(16,2);
                 lcd.write(byte(0));}
void loop (){}
```

Sample Interfacing the LCD

Interface the LCD display to the Arduino Uno Board using 4-bit mode and write a simple program to display the number of seconds since the reset of the Arduino board.

```
   create an LCD object with 4-bit mode and set interface pins
LiquidCrystal lcd = new LiquidCrystal(this, 10, 11, 3, 2, 1, 0);

public void setup(){
            set up the LCD's number of columns and rows:
         lcd.begin(16, 2);
            Print a message to the LCD.
         lcd.print("# of seconds:");
}

public void loop(){
            set the cursor to column 0, line 1
         lcd.setCursor(0, 1);
            print the number of seconds since reset:
         lcd.print(millis() / 1000);
}
```