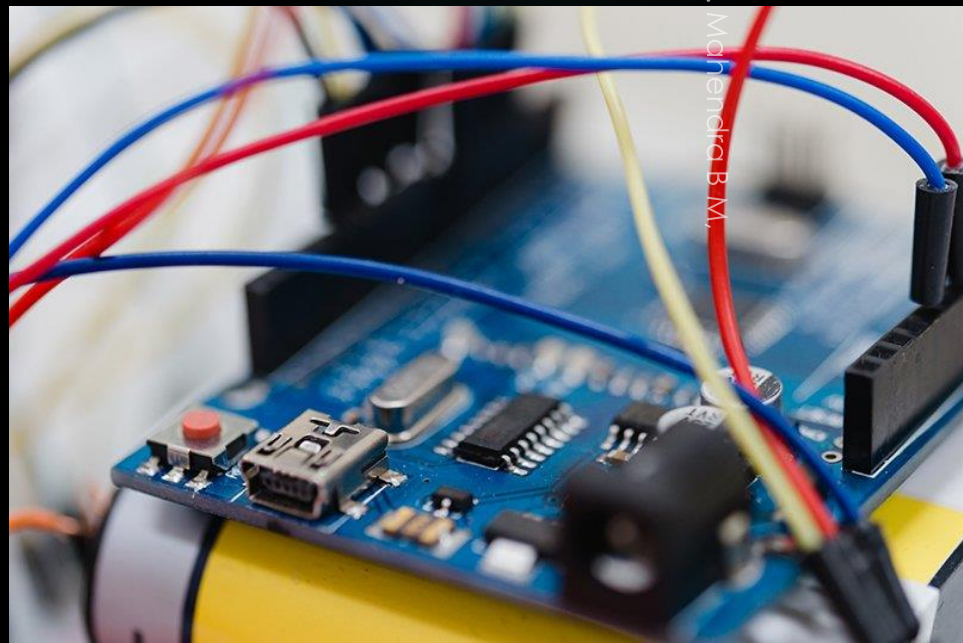
INTRODUCTION TO EMBEDDED SYSTEMS

Category: Emerging Technologies

COURSE CODE : 22EM211



Unit-I: Introduction

- Introduction: Definition of Embedded Systems, Typical examples, and Application domains (Automotive, Consumer, etc), Characteristics, Typical block diagram, Input, Core, Output, Commercial Off the Shelf Components (COTS). Processing Components, Microprocessors & Microcontrollers, Indicative Examples (Microcontrollers on Arduino boards), Development boards (Arduino boards), Concepts and brief introduction to Memory, Interrupts, Power Supply, Clocks, Reset. Case Studies: Washing Machine, Antilock Brake Systems (Block diagram & Working Principle).

Definition of Embedded Systems

- Embedded Systems: An embedded system is a combination of computer hardware and software designed for a specific function.
- Embedded systems may also function within a larger system. The systems can be programmable or have a fixed functionality.
- They are dedicated to a specific function or set of functions.
- Embedded systems have limited resources (memory, processing power) and are often real-time systems.

Embedded System Block Diagram

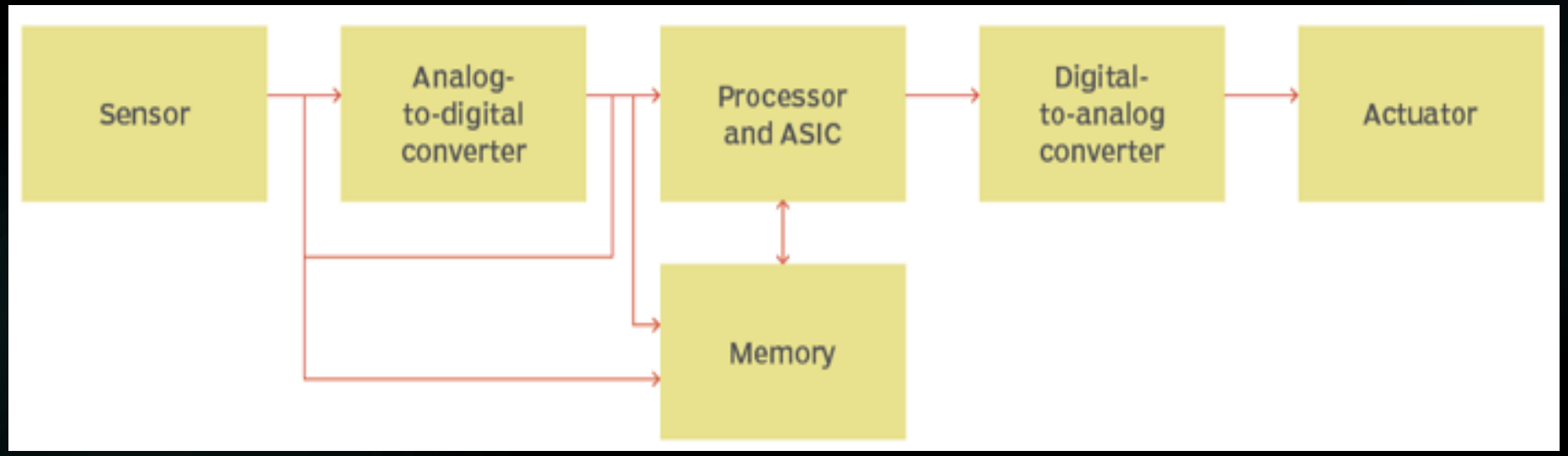


Figure1.1: A diagram of the basic structure and flow of information in embedded systems.

How does an embedded system work?

- Embedded systems always function as part of a complete device -- that's what's meant by the term embedded.
- Generally, they comprise a processor, power supply, and memory and communication ports. Embedded systems use the communication ports to transmit data between the processor and peripheral devices.
- The processor interprets this data with the help of minimal software stored on the memory.
- Often, embedded systems are used in real-time operating environments and use a real-time operating system (RTOS) to communicate with the hardware.

Structure of embedded systems

Embedded systems vary in complexity but, generally, consist of three main elements:

- **Hardware:** The hardware of embedded systems is based around microprocessors and microcontrollers. Microprocessors are very similar to microcontrollers and, typically, refer to a CPU (central processing unit) that is integrated with other basic computing components such as memory chips and digital signal processors (DSPs). Microcontrollers have those components built into one chip.
- **Software and firmware:** Software for embedded systems can vary in complexity. However, industrial-grade microcontrollers and embedded IoT systems usually run very simple software that requires little memory.
- **Real-time operating system:** These are not always included in embedded systems, especially smaller-scale systems. RTOSes define how the system works by supervising the software and setting rules during program execution.

Typical Examples of Embedded Systems

(But not limited to these)

- **Smartphones:**
  - Incorporate various embedded systems like the processor, memory, and sensors.
  - Perform functions such as communication, multimedia, and application execution.
- **Automotive Systems:**
  - Engine control units (ECUs) regulate fuel injection, ignition timing, and emissions.
  - Anti-lock braking systems (ABS) provide improved vehicle control during braking.
- **Consumer Electronics:**
  - Digital cameras, MP3 players, and smart TVs utilize embedded systems for image processing, audio playback, and content streaming.
- **Medical Devices:**
  - Implantable pacemakers and insulin pumps provide life-saving functionality.
  - Medical monitoring devices track vital signs and provide real-time feedback.

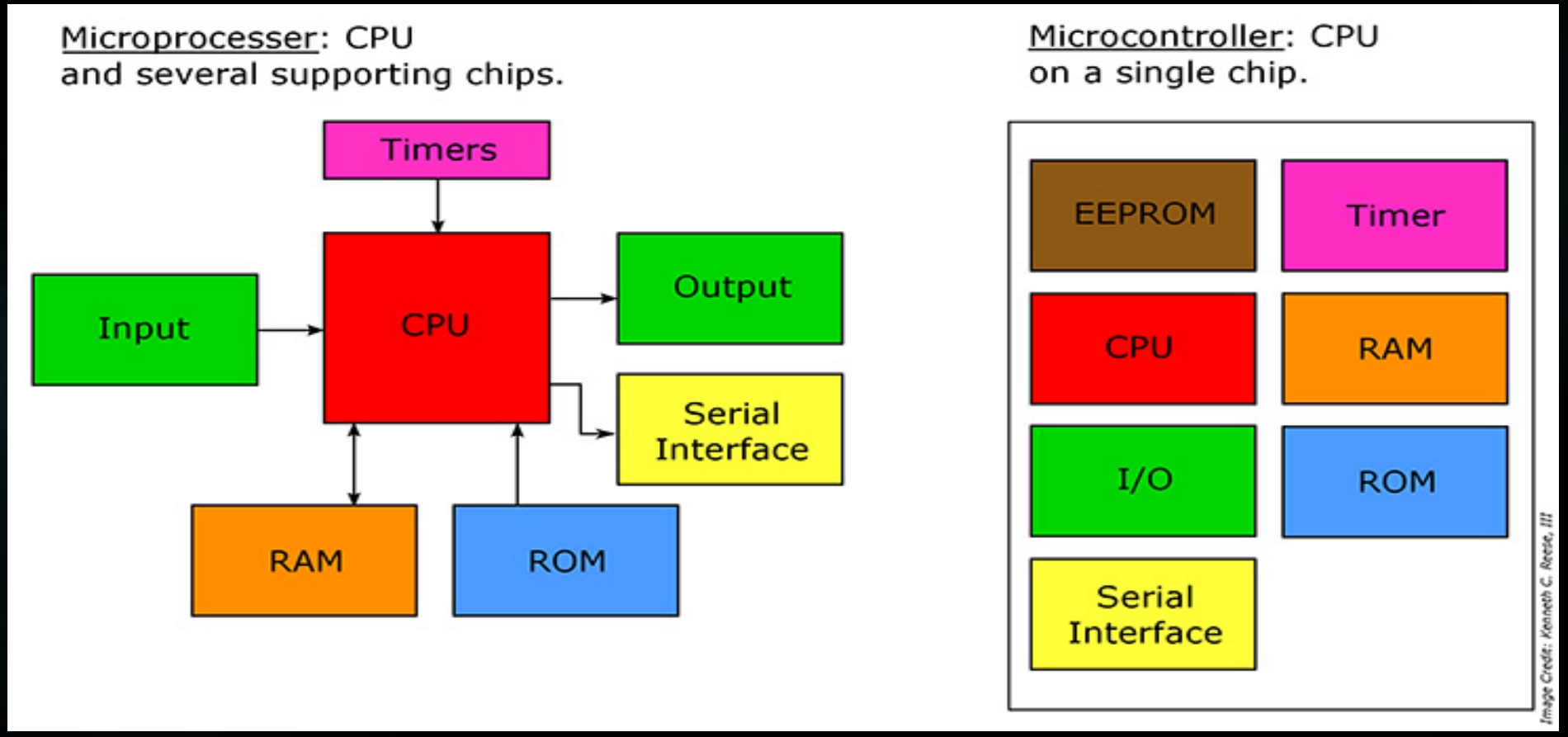
Application Domains of Embedded Systems **(But not limited to these)**

- **Automotive:**
  - Engine management systems
  - Infotainment systems
  - Advanced driver-assistance systems (ADAS)
  - Connected car technologies
- **Consumer Electronics**:
  - Smart home automation
  - Wearable devices
  - Gaming consoles
  - Home entertainment systems
- **Industrial Automation:**
  - Programmable logic controllers (PLCs)
  - Robotics systems
  - Process control systems
  - Machine vision systems
- **Healthcare:**
  - Medical imaging devices
  - Patient monitoring systems
  - Drug delivery systems
  - Prosthetic devices

Conclusion

- Embedded systems are specialized computer systems designed for specific tasks within larger systems or devices.
- They have limited resources and are often real-time systems.
- Typical examples include smartphones, automotive systems, consumer electronics, and medical devices.
- Embedded systems find applications in various domains such as automotive, consumer electronics, industrial automation, and healthcare.

Microprocessor and Microcontroller



Microprocessor and Microcontroller

Microprocessor

- Intel Core i7
- AMD Ryzen
- Qualcomm Snapdragon
- IBM POWER9
- ARM Cortex-A

Microcontroller

- Arduino
- Raspberry Pi
- Texas Instruments MSP430
- Atmel AVR
- STMicroelectronics STM32

Difference between Microcontroller & Microprocessor

Microcontroller

- Application-specific
- Integrated components (CPU, memory, I/O)
- On-chip memory (ROM, RAM)
- Reduced Instruction Set (RISC)
- Lower clock speeds
- Lower power consumption
- Cost-effective
- Embedded systems, control applications
- Robotics, automotive systems, home appliances

Microprocessor

- General-purpose
- CPU-focused
- External memory
- Complex Instruction Set (CISC)
- Higher clock speeds
- High computational power
- Expensive
- Wide range of applications
- Personal computers, servers

Commercial Off the Shelf Components (COTS)

- Commercial Off-the-Shelf (COTS) components refer to ready-made, pre-built products that are readily available in the market and not specifically developed for a particular customer or application.
- These components are mass-produced by manufacturers and sold to a wide range of customers for various purposes.
- examples of Commercial Off-the-Shelf (COTS) components:
  - Computer Processors
  - Memory Modules
  - Display Panels
  - Microcontrollers
  - Sensors and not limited to these.

Microcontrollers on Arduino boards

- Microcontrollers on Arduino boards play a central role in enabling the functionality and versatility of Arduino-based projects.
- Arduino boards are designed to provide an accessible platform for both beginners and experienced users to develop interactive electronic projects.

Key features and examples of microcontrollers used in Arduino boards

ATmega Series Microcontrollers:

The majority of Arduino boards are based on microcontrollers from the ATmega series, developed by Atmel (now Microchip Technology). Some common examples include:

- **ATmega328P:** This is the microcontroller used in Arduino Uno, one of the most widely used Arduino boards. It offers 32KB of flash memory, 2KB of SRAM, and 1KB of EEPROM.
- **ATmega2560:** Arduino Mega 2560 utilizes this microcontroller, which provides more memory and I/O pins compared to Arduino Uno. It has 256KB of flash memory, 8KB of SRAM, and 4KB of EEPROM.
- **ATmega32U4:** The Arduino Leonardo and Arduino Micro boards feature this microcontroller. It includes 32KB of flash memory, 2.5KB of SRAM, and 1KB of EEPROM. The ATmega32U4 also has built-in USB support, allowing the board to appear as a virtual keyboard or mouse when connected to a computer.

Key features and examples of microcontrollers used in Arduino boards

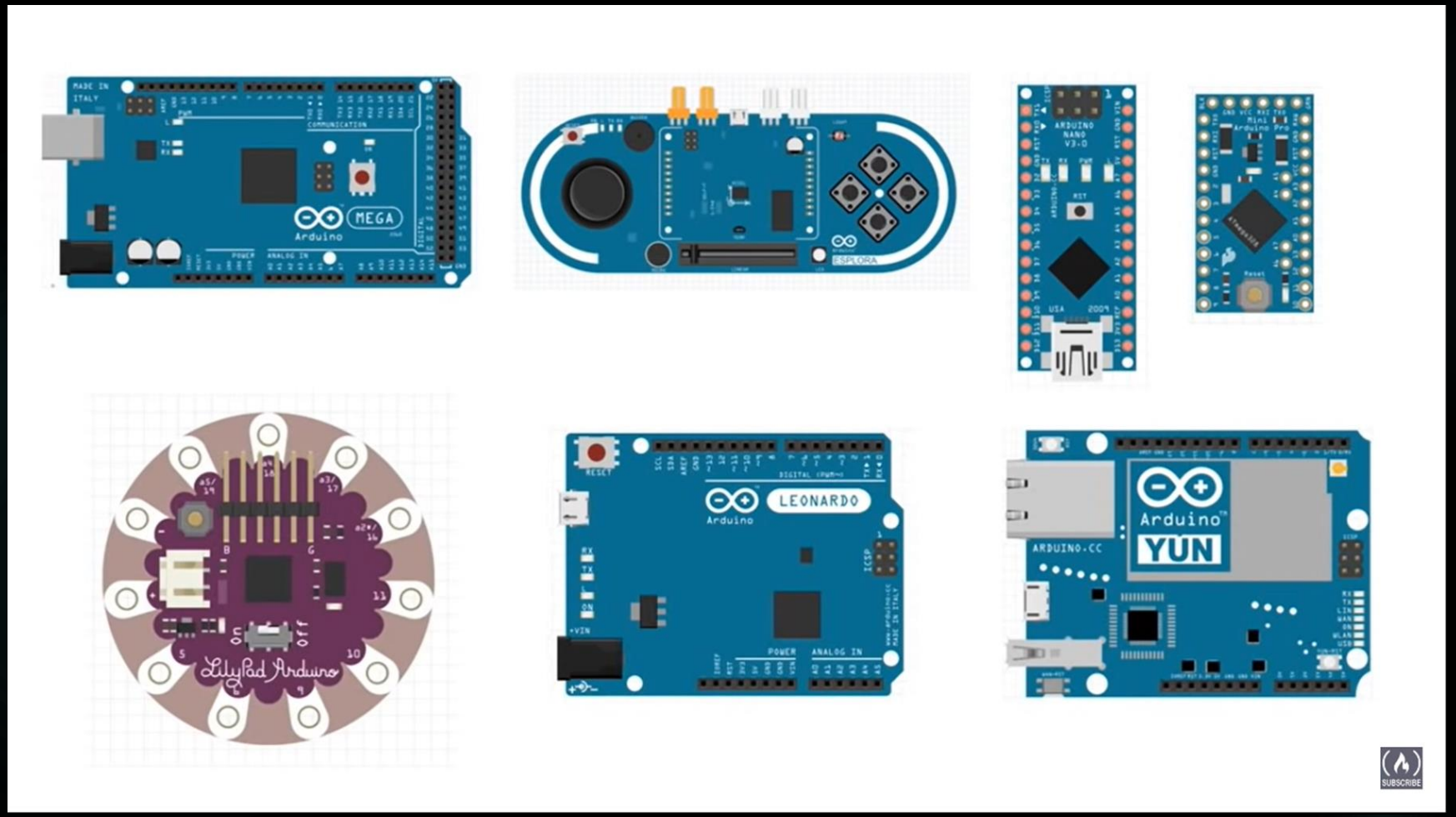
ARM-based Microcontrollers:

Arduino has also introduced boards based on ARM-based microcontrollers, offering more processing power and advanced features. Some examples include:

- **Arduino Due:** The Arduino Due is based on the Atmel SAM3X8E microcontroller, which is powered by an ARM Cortex-M3 processor. It provides 512KB of flash memory, 96KB of SRAM, and various advanced peripherals.

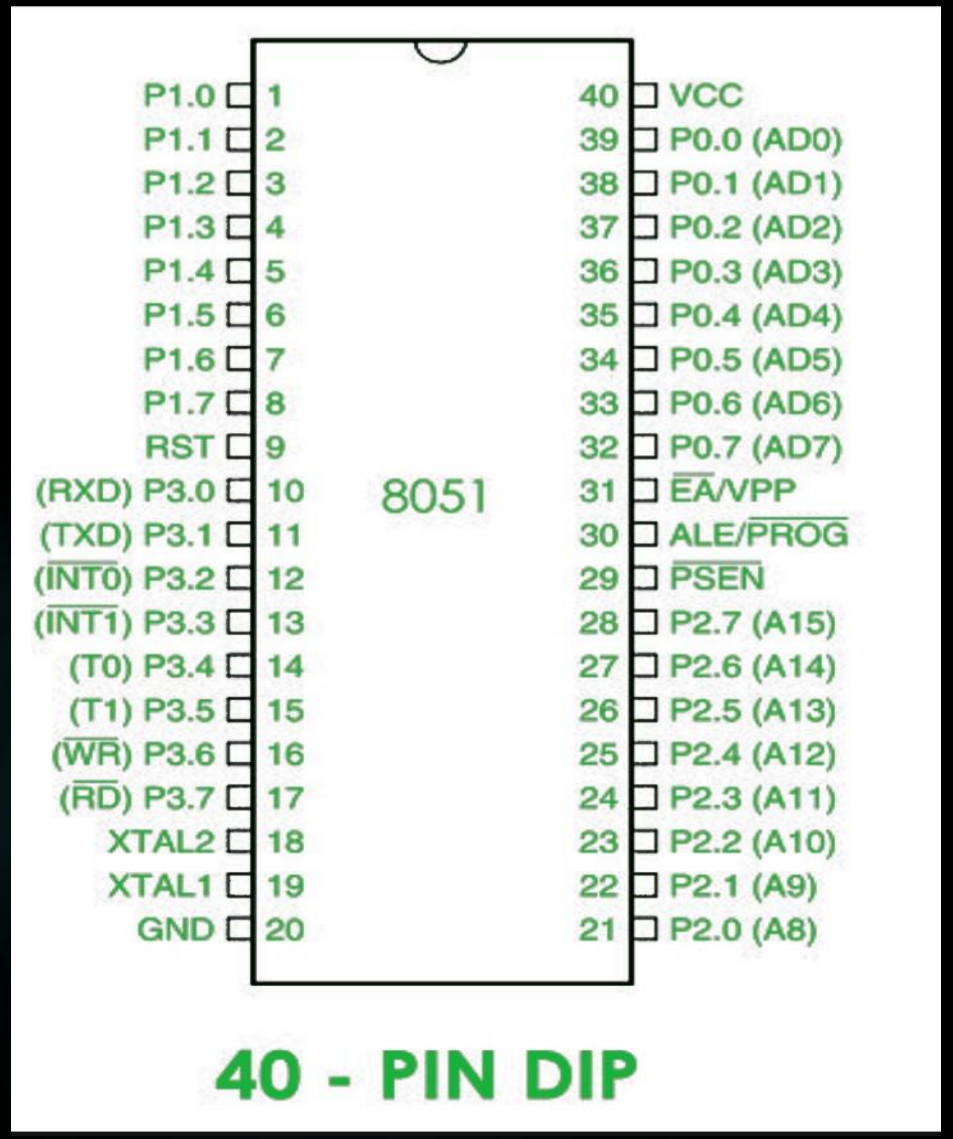
- **Arduino MKR Family:** Arduino MKR boards, such as MKR1000, MKR WiFi 1010, or MKR Zero, are based on various ARM Cortex-M0 or Cortex-M4 microcontrollers. These boards offer different combinations of memory, connectivity options, and power-saving features, catering to diverse IoT and wireless communication applications.

Version of Arduino

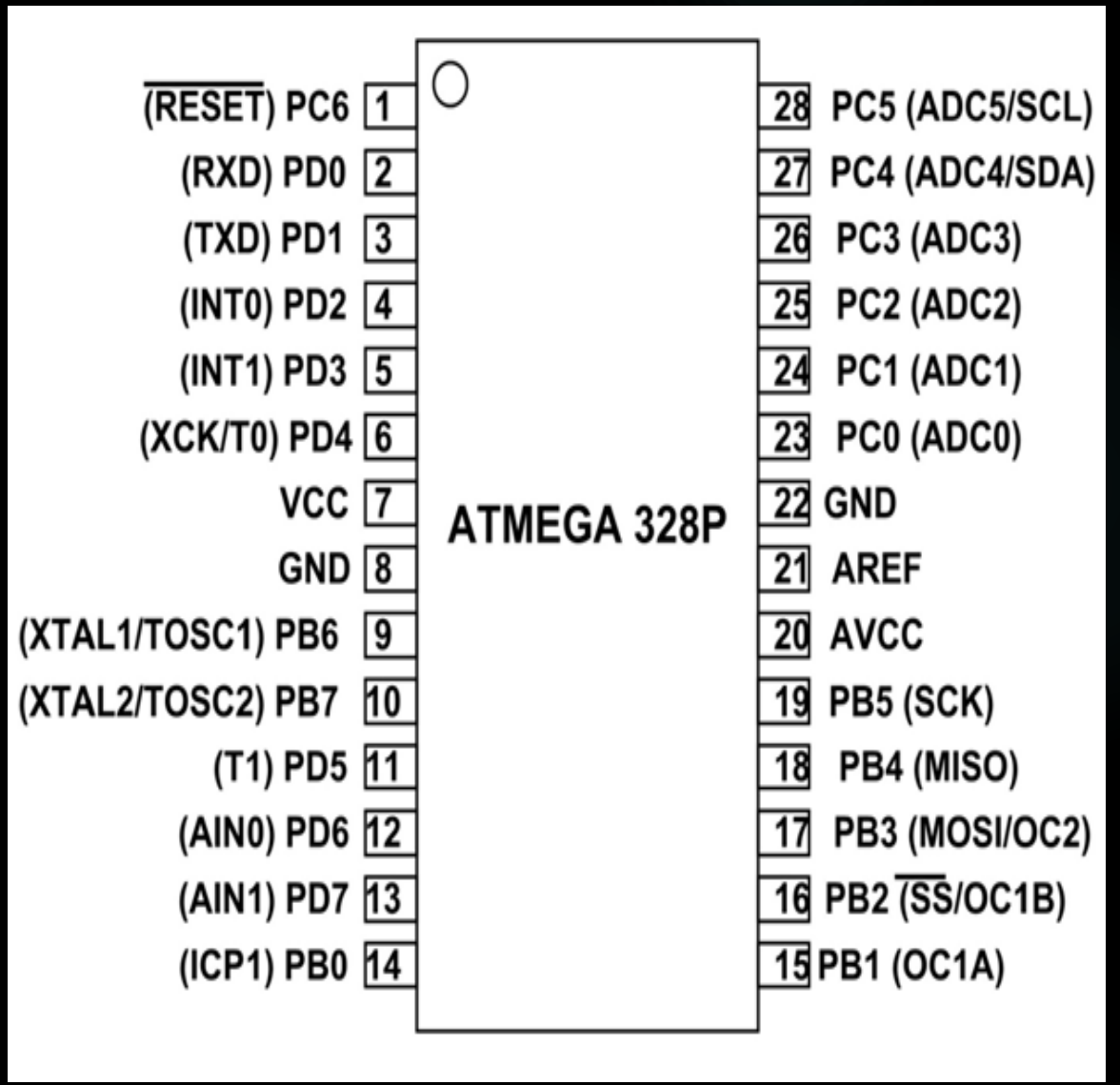


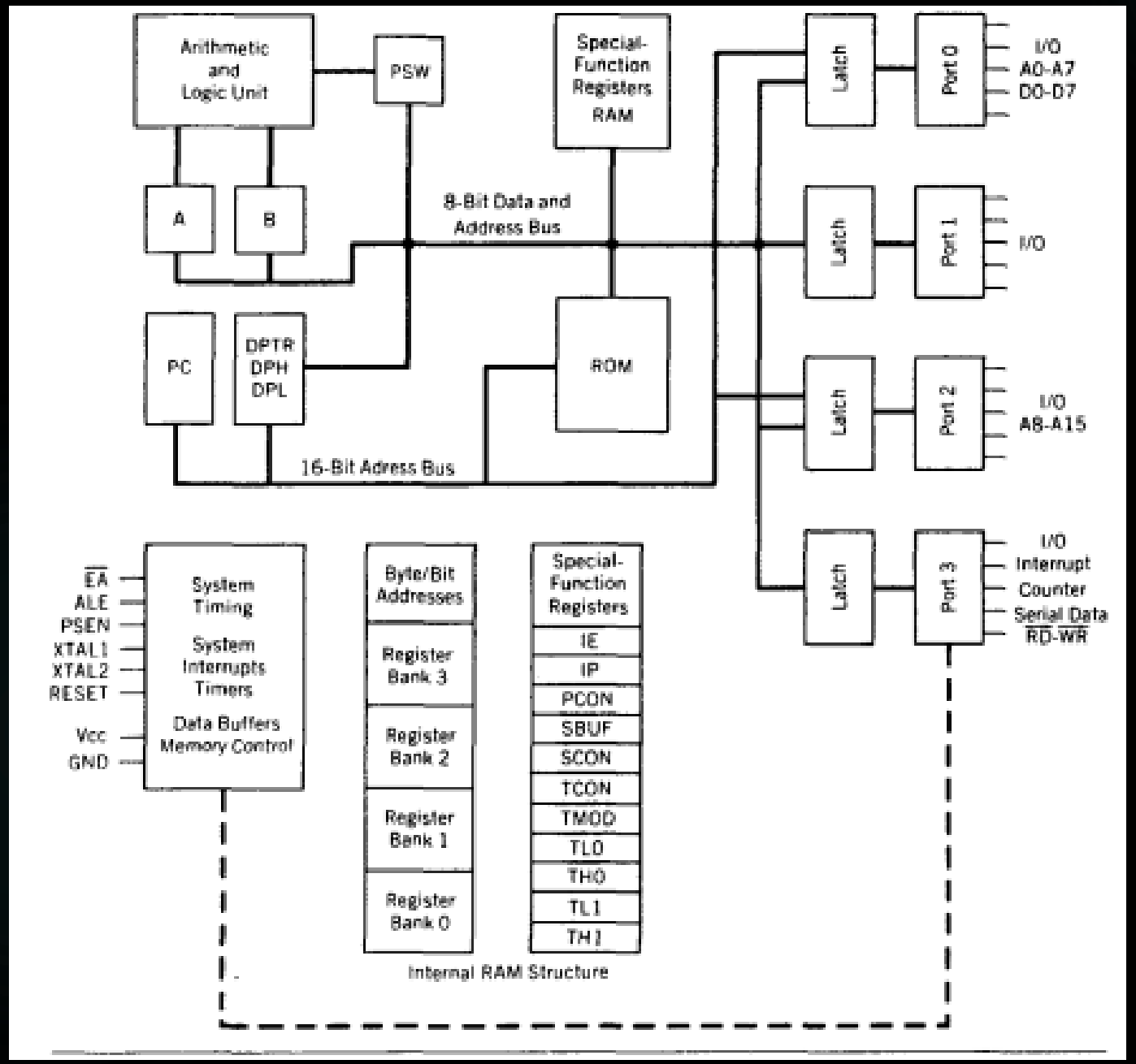
Difference between 8051 and AVR

Parameter	8051 Microcontroller	AVR Microcontroller
Basic	8051 is a simple and low cost 8-bit microcontroller used in several types of embedded system applications.	AVR is a high-performance 8-bit microcontroller used in advanced systems like robotics, industrial control systems, smart home appliances, etc.
Bus width	8051 microcontrollers have a bus width of 8 bits.	AVR microcontrollers have bus width of 8-bit. But, some AVR microcontrollers also have a bus width of 32 bits.
Developer	8051 microcontroller was developed by Intel.	AVR microcontroller was produced by Atmel Corporation.
Memory Architecture	8051 microcontrollers have Von Neumann architecture.	AVR microcontrollers have Modified Harvard architecture.
Instruction set architecture	8051 microcontrollers are based on CISC (Complex Instruction Set Computer) architecture.	AVR microcontrollers are based on RISC (Reduced Instruction Set Computer) architecture.
Registers	8051 microcontrollers have a smaller number of registers.	AVR microcontrollers have a greater number of registers.
Power consumption	The power consumption for 8051 microcontrollers is average.	AVR microcontrollers consume less power than 8051.


P1.0 1
P1.1 2
P1.2 3
P1.3 4
P1.4 5
P1.5 6
P1.6 7
P1.7 8
RST 9
(RXD) P3.0 10
(TXD) P3.1 11
(INT0) P3.2 12
(INT1) P3.3 13
(T0) P3.4 14
(T1) P3.5 15
(WR) P3.6 16
(RD) P3.7 17
XTAL2 18
XTAL1 19
GND 20
8051
40 VCC
39 P0.0 (AD0)
38 P0.1 (AD1)
37 P0.2 (AD2)
36 P0.3 (AD3)
35 P0.4 (AD4)
34 P0.5 (AD5)
33 P0.6 (AD6)
32 P0.7 (AD7)
31 EA/VPP
30 ALE/PROG
29 PSEN
28 P2.7 (A15)
27 P2.6 (A14)
26 P2.5 (A13)
25 P2.4 (A12)
24 P2.3 (A11)
23 P2.2 (A10)
22 P2.1 (A9)
21 P2.0 (A8)

40 - PIN DIP


(RESET) PC6 1
(RXD) PD0 2
(TXD) PD1 3
(INT0) PD2 4
(INT1) PD3 5
(XCK/T0) PD4 6
VCC 7
GND 8
(XTAL1/TOSC1) PB6 9
(XTAL2/TOSC2) PB7 10
(T1) PD5 11
(AIN0) PD6 12
(AIN1) PD7 13
(ICP1) PB0 14
ATMEGA 328P
28 PC5 (ADC5/SCL)
27 PC4 (ADC4/SDA)
26 PC3 (ADC3)
25 PC2 (ADC2)
24 PC1 (ADC1)
23 PC0 (ADC0)
22 GND
21 AREF
20 AVCC
19 PB5 (SCK)
18 PB4 (MISO)
17 PB3 (MOSI/OC2)
16 PB2 (SS/OC1B)
15 PB1 (OC1A)


Arithmetic and Logic Unit
PSW
Special-Function Registers RAM
A
B
8-Bit Data and Address Bus
PC
DPTR
DPH
DPL
ROM
16-Bit Adress Bus
Latch
Port 0
I/O
A0-A7
D0-D7
Port 1
I/O
Port 2
I/O
A8-A15
Port 3
I/O
Interrupt
Counter
Serial Data
RD-WR
EA
ALE
PSEN
XTAL1
XTAL2
RESET
Vcc
GND
System Timing
System Interrupts Timers
Data Buffers Memory Control
Byte/Bit Addresses
Register Bank 3
Register Bank 2
Register Bank 1
Register Bank 0
Special-Function Registers
IE
IP
PCON
SBUF
SCON
TCON
TMOD
TL0
TH0
TL1
TH1
Internal RAM Structure

Concepts of Memory, Interrupts, Power Supply, Clocks, Reset.

These concepts form the foundation of embedded systems and are essential for understanding and developing efficient and reliable embedded applications.

Memory

Memory in embedded systems refers to the storage used to store program instructions and data. There are typically two types of memory used:

- **ROM (Read-Only Memory):** ROM contains non-volatile data, such as firmware or fixed program instructions that are permanently stored and cannot be modified during runtime.

- **RAM (Random Access Memory):** RAM is volatile memory that allows read and write operations during runtime. It is used to store data and variables that can be accessed by the processor.

Memory plays a critical role in executing instructions and storing temporary or permanent data in embedded systems.

Interrupts

- Interrupts are signals generated by external events or internal conditions that pause the normal execution of a program.
- When an interrupt occurs, the processor temporarily suspends the current task to handle the interrupt request.
- Interrupts are used to respond to time-sensitive events, such as sensor inputs, communication requests, or hardware errors.
- Interrupts ensure that critical events are processed promptly and allow the processor to efficiently handle multiple tasks concurrently.

Power Supply

- Power supply is the provision of electrical energy required to operate an embedded system.
- Embedded systems typically require a stable and reliable power source to ensure proper functioning.
- The power supply may come from various sources, including batteries, AC mains, or power adapters.
- Power management techniques are often implemented to optimize power consumption and extend the battery life in battery-powered devices.

Clocks

- Clocks provide a regular and synchronized timing signal to the processor and other components in an embedded system.
- The clock signal determines the pace at which instructions are executed, data is processed, and peripherals operate.
- Clocks ensure proper coordination and synchronization of different components, allowing the system to function accurately and reliably.

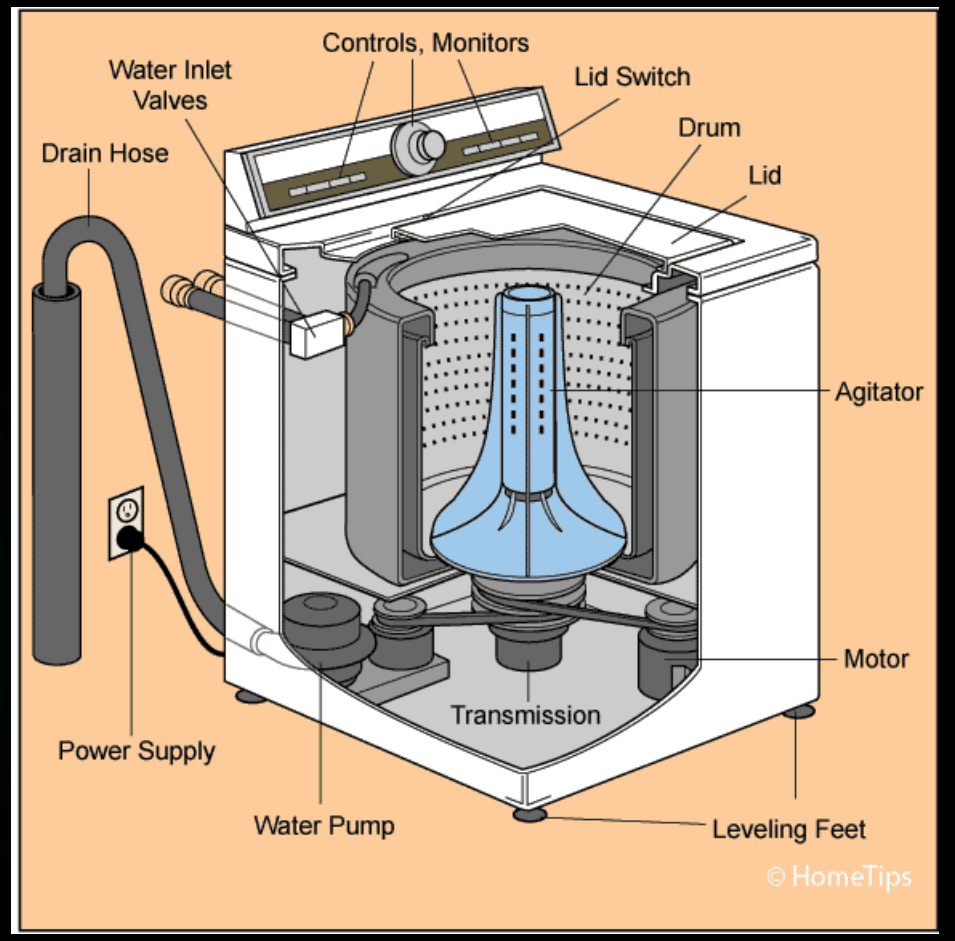
Reset

- Reset is the process of restarting an embedded system or restoring it to its initial state.
- A reset can be triggered by various events, such as power-on, manual reset button press, or a signal from a watchdog timer.
- When a reset occurs, the system restarts, and the processor initializes itself, including clearing memory, resetting peripheral devices, and executing startup routines.
- Reset ensures a clean and predictable state of the system, enabling proper initialization.

Case Studies: Washing Machine, Antilock Brake Systems

These case studies demonstrate the block diagrams and working principles of a washing machine and antilock brake systems, highlighting the key components and operations involved in their functioning.

washing machine



washing machine

The embedded system components and the microcontroller play a crucial role in controlling and coordinating the operations of the washing machine, enabling efficient and automated washing processes.

- **Program Selection:** The user selects a wash program using the control panel. The microcontroller receives the input and processes it.
- **Sensor Monitoring:** The microcontroller continuously monitors the sensor inputs, such as water level sensors, temperature sensors, and rotational speed sensors. It uses this information to ensure proper control and synchronization of the washing process.
- **Control Algorithm Execution:** Based on the selected program and sensor inputs, the microcontroller executes control algorithms to regulate the motor speed, water inlet valve, and other components. It ensures the right amount of water is added, the drum rotates at the desired speed, and the temperature is maintained as per the program.

washing machine (Cont)

- **User Feedback and Display:** The microcontroller communicates with the user interface to provide feedback on the current stage of the wash cycle, display program options, and indicate any errors or notifications.
- **Safety Features:** The microcontroller incorporates safety features, such as detecting abnormal sensor readings, monitoring water overflow, and ensuring proper door lock status, to ensure safe and reliable operation.
- **Power Management:** The microcontroller manages power consumption by controlling the operation of various components, optimizing energy usage, and putting the system into low-power modes when not in use.

Unit-2: Introduction

Integrated Development Environment(Ide) And Programming: Basics of Embedded C Programming, Data Types, Arithmetic & Logical Operators, Loops, Functions, #define Macros, Structures (Declaration and Accessing data members). Integrated Development Environment tools: Editor, Compiler, Linker, Loader, Debugger (Definitions only). Practice: Working with Arduino IDE(Simple programs on Operators, Loops and Functions).

Integrated Development Environment(Ide) and Programming:

- An Integrated Development Environment (IDE) is a software application that provides a comprehensive set of tools and features for writing, testing, and debugging code.
- When it comes to Arduino programming, there are specific IDEs tailored for working with Arduino boards.
- The official Arduino IDE is the most commonly used IDE for Arduino development, but there are also alternative IDEs available, such as PlatformIO and Visual Studio Code with Arduino extensions.

Key components and functionalities of an Arduino IDE:

- **Code Editor**: The IDE provides a code editor where you can write your Arduino programs. It offers features like syntax highlighting, auto-completion, and indentation to assist you in writing clean and error-free code. The code editor is where you write the instructions that control the behavior of your Arduino board.

- **Library Manager**: Arduino libraries are collections of pre-written code that provide additional functionality and simplify complex tasks. The IDE includes a library manager that allows you to easily search, install, and manage libraries required for your project. Libraries provide ready-to-use functions for tasks like controlling LEDs, reading sensors, or communicating with other devices.

- **Compiler and Uploader**: Once you have written your Arduino code, the IDE compiles it into machine-readable instructions that can be understood by the Arduino board's microcontroller. The compiler checks for syntax errors and other issues in your code. After successful compilation, the IDE uploads the compiled code to the Arduino board, making it ready to run.

Key components and functionalities of an Arduino IDE: (Cont.)

- **Serial Monitor:** The IDE provides a serial monitor tool that allows you to communicate with your Arduino board through the serial port. It displays the data sent by the Arduino and allows you to send commands or instructions from the computer to the board. The serial monitor is useful for debugging and monitoring the behavior of your Arduino programs.

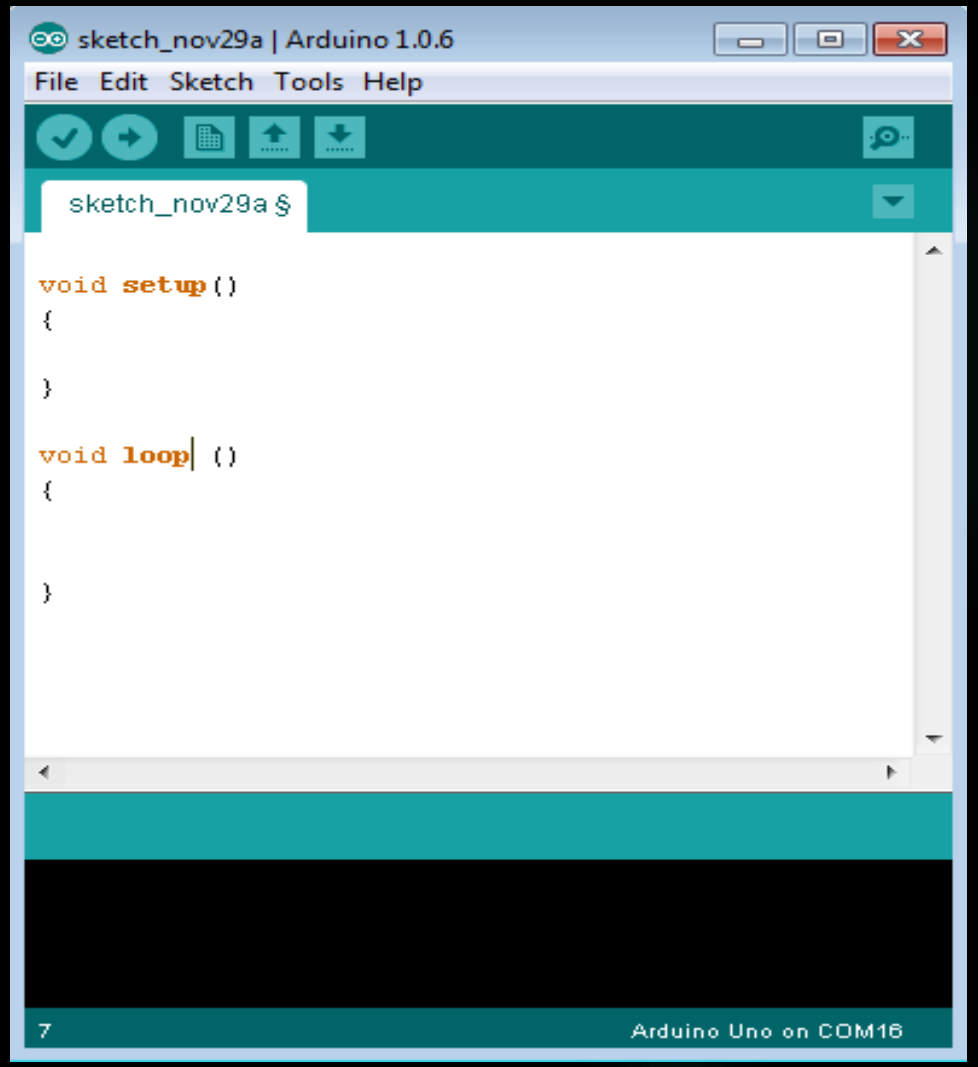
- **Integrated Examples:** Arduino IDE comes with a set of pre-built examples that demonstrate various functionalities of the Arduino board and its peripherals. These examples can be accessed from the IDE's menu and serve as a starting point for learning and building your own projects.

Basics of Embedded C Programming

- Embedded C programming with respect to Arduino involves writing code in the C programming language to control and interact with Arduino boards.
- Arduino uses a simplified version of C/C++ that provides libraries and functions specifically designed for the Arduino platform.
- **Sketch** – The first new terminology is the Arduino program called "sketch".

Structure of an Arduino Program:

- Every Arduino program must have two essential functions:
  - **setup() and loop().**
- The **setup()** function is called once when the Arduino board starts.
- It is used for initializing variables, setting pin modes, and configuring any necessary settings.
- The **loop()** function is called repeatedly after the setup() function.
- It contains the main logic of your program that will execute in a continuous loop until the board is powered off.



Pin Definitions:

- Arduino boards have a set of **digital input/output (I/O) pins and analog input pins.**
- **Digital pins** can be used for both input and output operations. They are typically labeled with numbers (e.g., 2, 3, 4, etc.).
- **Analog input** pins can read analog values from sensors or other analog devices. They are labeled with an 'A' followed by a number (e.g., A0, A1, A2, etc.).
- **Pin mode** should be set using pinMode() function as either INPUT or OUTPUT before using the pin for reading or writing operations.

```
int inputPin = 2;
int outputPin = 3;

void setup() {
  pinMode(inputPin, INPUT);
  pinMode(outputPin, OUTPUT);
}
```

Digital Input/Output:

- To read from a digital pin, you can use the digitalRead() function, which returns either HIGH or LOW.
- To write to a digital pin, use the digitalWrite() function. You can set the pin to HIGH or LOW.

```
void loop() {
  int value = digitalRead(inputPin);
  digitalWrite(outputPin, value);
}
```

Summarize the working of the following code:


```
int inputPin = 2;
int outputPin = 3;

void setup() {
  pinMode(inputPin, INPUT);
  pinMode(outputPin, OUTPUT);
}

void loop() {
  int value = digitalRead(inputPin);
  digitalWrite(outputPin, value);
}
```

Summary:

Analog Input:

- To read analog values from an analog input pin, use the analogRead() function. It returns a value between 0 and 1023.
- For example, to read from analog pin A0 and write the value to a digital pin:

```
int analogPin = A0;
int digitalPin = 3;

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

void loop() {
  int analogValue = analogRead(analogPin);
  digitalWrite(digitalPin, analogValue > 512 ? HIGH : LOW);
}
```

Functions and Libraries:

- Arduino provides various built-in functions and libraries that simplify common tasks.
- Examples include delay() for introducing delays, Serial library for serial communication, and libraries for specific sensors or devices.
- Libraries can be included using the #include directive at the beginning of the program.
- For example, to use the Servo library for controlling a servo motor:

```
#include <Servo.h>

Servo servo;
int angle = 0;

void setup() {
  servo.attach(9);
}

void loop() {
  servo.write(angle);
  delay(1000);
  angle += 45;
}
```

Data Types: ***int***

- Integers are primary data-type for number storage.
- On the Arduino Uno (and other ATmega based boards) an ***int*** stores a 16-bit (2-byte) value.
- This yields a range of -32,768 to 32,767 (minimum value of -2^15 and a maximum value of (2^15) - 1).
- On the Arduino Due and SAMD based boards (like MKR1000 and Zero), an int stores a 32-bit (4-byte) value. This yields a range of -2,147,483,648 to 2,147,483,647 (minimum value of -2^31 and a maximum value of (2^31) - 1).
- ***int's*** store negative numbers with a technique called (2's complement math). The highest bit, sometimes referred to as the "sign" bit, flags the number as a negative number. The rest of the bits are inverted and 1 is added.
- The Arduino takes care of dealing with negative numbers for you, so that arithmetic operations work transparently in the expected manner. There can be an unexpected complication in dealing with the bitshift right operator (>>) however.

Syntax


```
int var = val;
```

Parameters

- var: variable name.
- val: the value you assign to that variable.

Example Code

This code creates an integer called 'countUp', which is initially set as the number 0 (zero).

The variable goes up by 1 (one) each loop, being displayed on the serial monitor.

```
int countUp = 0;            //creates a variable integer called 'countUp'

void setup() {
  Serial.begin(9600);       // use the serial port to print the number
}

void loop() {
  countUp++;                //Adds 1 to the countUp int on every loop
  Serial.println(countUp);  // prints out the current state of countUp
  delay(1000);
}
```

Notes and Warnings

- When signed variables are made to exceed their maximum or minimum capacity they overflow.
- The result of an overflow is unpredictable so this should be avoided.
- A typical symptom of an overflow is the variable ***"rolling over"*** from its maximum capacity to its minimum or vice versa, but this is not always the case.
- If you want this behavior, use ***unsigned int***.

unsigned int

Description

- On the Uno and other ATMEGA based boards, unsigned ints (unsigned integers) are the same as ***ints*** in that they store a 2 byte value. Instead of storing negative numbers however they only store positive values, yielding a useful range of 0 to 65,535 ((2^16) - 1).
- The Due stores a 4 byte (32-bit) value, ranging from 0 to 4,294,967,295 (2^32 - 1).
- The difference between unsigned ints and (signed) ints, lies in the way the highest bit, sometimes referred to as the "sign" bit, is interpreted. In the Arduino int type (which is signed), if the high bit is a "1", the number is interpreted as a negative number, and the other 15 bits are interpreted with (2's complement math).

Syntax

- unsigned int var = val;

Parameters

- var: variable name. val: the value you assign to that variable.

Example Code

- unsigned int ledPin = 13;

Long and Unsigned Long

DIY

https://docs.arduino.cc/learn/starting-guide/getting-started-arduino

For more data types and examples, refer above link

bool

Description

- A bool holds one of two values, ***true or false***. (Each bool variable occupies one byte of memory.)

Syntax

- bool var = val;

Parameters

- var: variable name.
- val: the value to assign to that variable.

Code shows how to use the bool datatype.


```
int LEDpin = 5;     // LED on pin 5
int switchPin = 13; // momentary switch on 13, other side connected to ground
bool running = false;
void setup() {
  pinMode(LEDpin, OUTPUT);
  pinMode(switchPin, INPUT);
  digitalWrite(switchPin, HIGH);  // turn on pullup resistor
}
```

```
void loop() {
  if (digitalRead(switchPin) == LOW) {
    // switch is pressed - pull-up keeps pin high normally
    delay(100);                     // delay to debounce switch
    running = !running;             // toggle running variable
    digitalWrite(LEDpin, running);  // indicate via LED
  }
}
```

float

Description

- Datatype for floating-point numbers, a number that has a decimal point.
- Floating-point numbers are often used to approximate analog and continuous values because they have greater resolution than integers.
- 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 bytes) of information.

Syntax

- float var = val;

Parameters

- var: variable name.
- val: the value you assign to that variable.

z = (float)x / 2.0; // z now contains .5 (you have to use 2.0, not 2)

Example Code


```
float myfloat;
float sensorCalbrate = 1.117;

int x;
int y;
float z;

x = 1;
y = x / 2;          // y now contains 0, ints can't hold fractions
z = (float)x / 2.0; // z now contains .5 (you have to use 2.0, not 2)
```

unsigned char

Description

- An unsigned data type that occupies 1 byte of memory. Same as the byte datatype.
- The unsigned char datatype encodes numbers from 0 to 255.
- For consistency of Arduino programming style, the byte data type is to be preferred.

Syntax

- unsigned char var = val;

Parameters

- var: variable name.
- val: the value to assign to that variable.

char

- A data type used to store a character value. Character literals are written in single quotes, like this: 'A' (for multiple characters - strings - use double quotes: "ABC").
- Characters are stored as numbers however.
- You can see the specific encoding in the ASCII chart.
- This means that it is possible to do arithmetic on characters, in which the ASCII value of the character is used (e.g. 'A' + 1 has the value 66, since the ASCII value of the capital letter A is 65).
- See Serial.println reference for more on how characters are translated to numbers.
- The size of the char datatype is at least 8 bits.
- It's recommended to only use char for storing characters.
- For an unsigned, one-byte (8 bit) data type, use the byte data type.

Syntax

char var = val;

Parameters

- var: variable name.
- val: the value to assign to that variable.

Example Code


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

byte

A byte stores an 8-bit unsigned number, from 0 to 255.

Syntax

- byte var = val;

Parameters

- var: variable name.
- val: the value to assign to that variable.

void

- The void keyword is used to indicate that a function does not return a value.
- This is particularly useful in embedded systems when you have functions that perform actions but don't need to provide a result.

Example Code

The code shows how to use void.

```
// actions are performed in the functions "setup" and "loop"
// but  no information is reported to the larger program

void setup() {
  // ...
}

void loop() {
  // ...
}
```

Example Code


```
int counter;

void setup() {
  Serial.begin(9600);
  counter = 10;

  Serial.print("Counter: ");
  Serial.println(counter);
}
```

```
void loop() {
  counter++;
  Serial.print("Counter: ");
  Serial.println(counter);
  delay(1000);
}
```

Arduino void setup

- As the void setup function is called only once at the very beginning of the program, this will be the place to:
  - Initialize variables' values.
  - Setup communications (ex: Serial).
  - Setup modes for digital pins (input/output).
  - Initialize any hardware component (sensor/actuator) plugged to the Arduino.Etc.
- The void setup, as its name suggest, is made for you to do any setup required at the beginning of the program. Don't write the core functionalities here, just the initialization code.

Arduino void loop

- Now, in the void loop you'll write your main program, knowing that the initialization is already done. In this function, always keep in mind that the last line is followed by the first line!

- Also, any variable you've declared inside the void loop will be lost when the program exits and enters the function again. So, if you want to be able to keep data between 2 void loop, make sure to declare variables in a more global scope.

Arithmetic & Logical Operators

- These operators can be used to manipulate variables, control flow statements, and perform various calculations and logical operations in your Arduino code. Keep in mind the data types and the limitations of the microcontroller you are using to avoid overflow or unexpected behavior.

Arithmetic Operators:

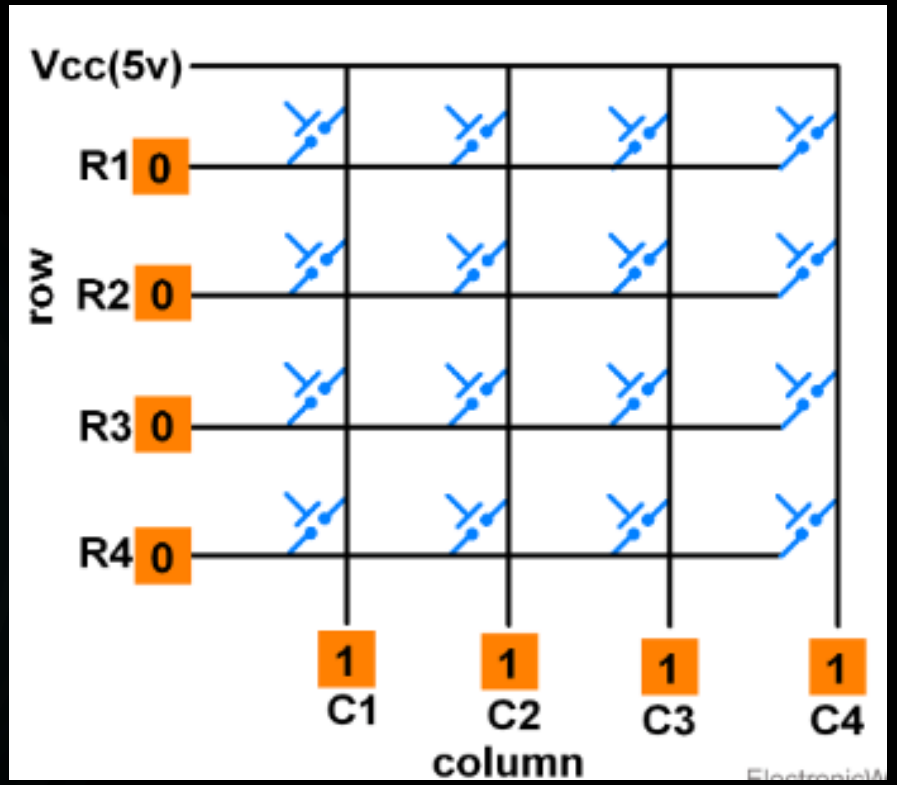
- Addition: +
- Subtraction: -
- Multiplication: *
- Division: /
- Modulus (remainder): %

Simple Calculator using Arithmetic Operators

Introduction to 4x4 keypad

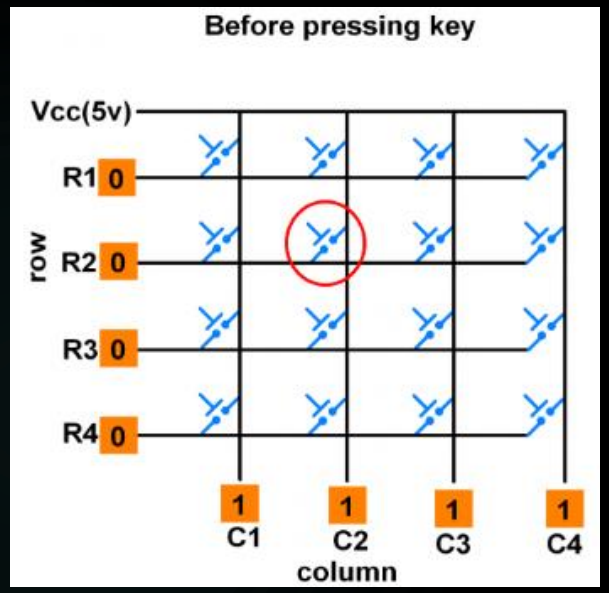
- When we want to interface one key to the microcontroller then it needs one GPIO pin. But when we want to interface many keys like 9, 12 or 16 etc., then it may acquire all GPIO pins of microcontroller.
- To save some GPIO pins of microcontroller, we can use matrix keypad. Matrix keypad is nothing but keys arrange in row and column.
- E.g. if we want to interface 16 keys to the microcontroller then we require 16 GPIO pins but if we use matrix 4x4 keypad then we require only 8 GPIO pins of microcontroller.

4x4 Keypad Matrix Structure



Keypad Matrix Working

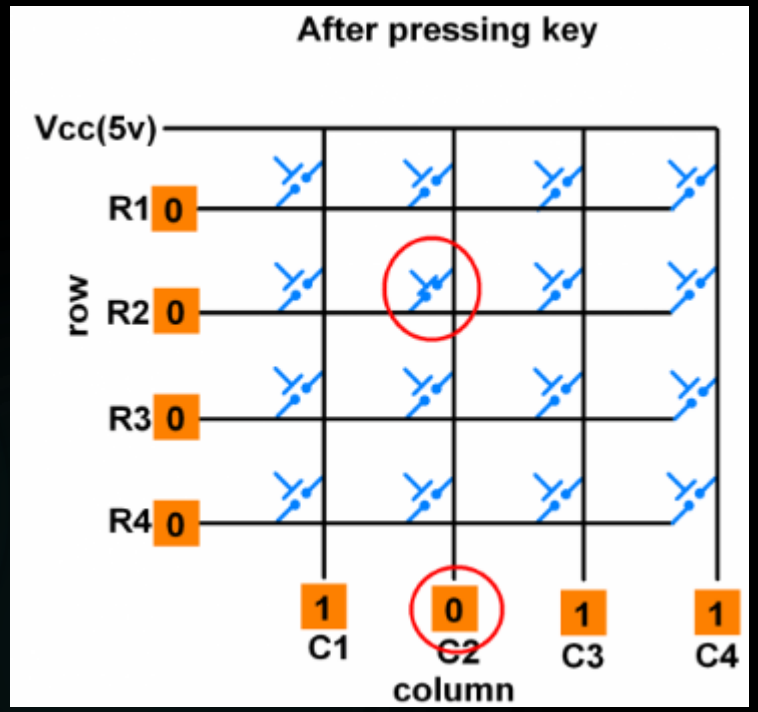
- Scan Keys:
  - To detect a pressed key, the microcontroller grounds all rows by providing 0 to the output latch, and then it reads the columns shown in above fig.



R1 R2 R3 R4 = 0 0 0 0
C1 C2 C3 C4 = 1 1 1 1

If the data read from columns is = 1111, no key has been pressed shown in above fig. and the process continues till key press is detected.

- Now, consider highlighted key in previous fig. is pressed. After pressing key, it makes contact of row with column shown below.



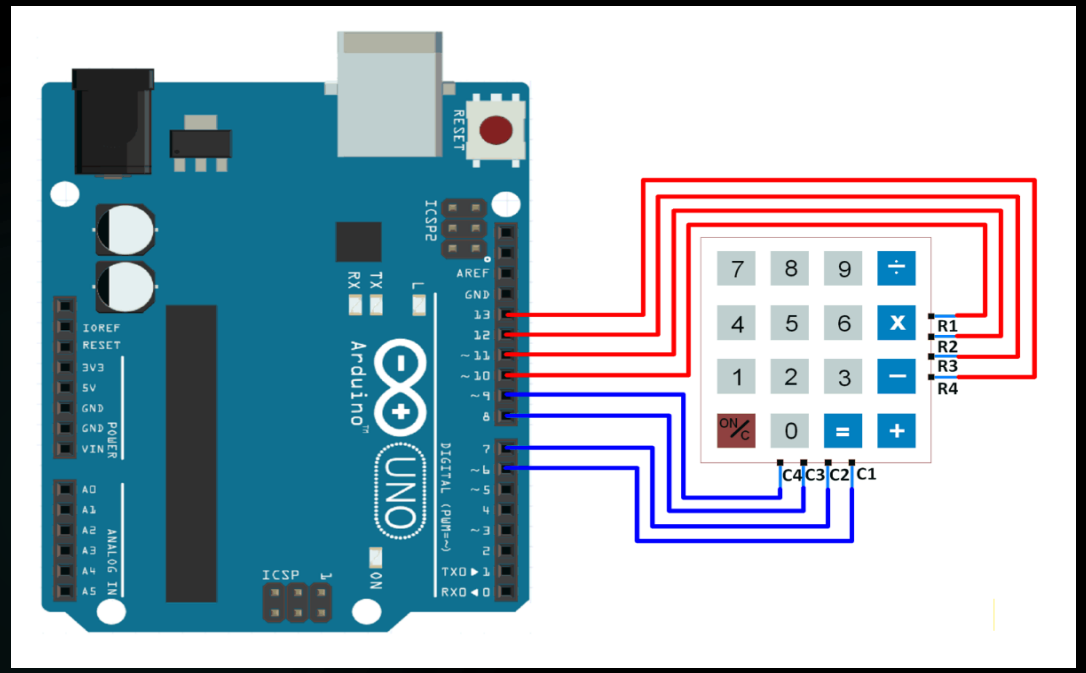
R1 R2 R3 R4 = 0 0 0 0
C1 C2 C3 C4 = 1 0 1 1

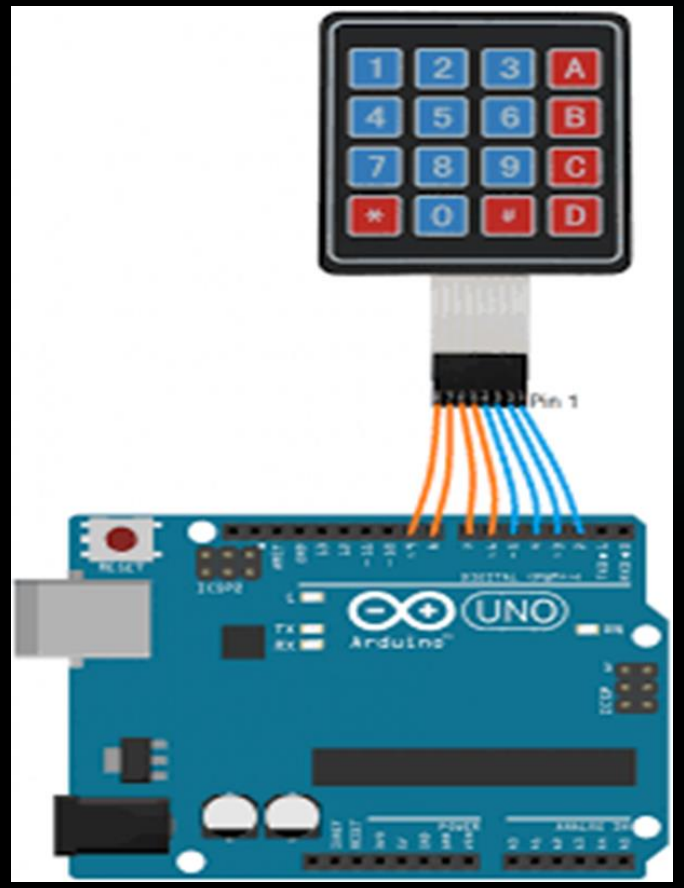
If one of the column bits has a zero, this means that a key press has occurred.
For example, if C1:C4 = 1011, this means that a key in the C2 column has been pressed.
After detecting a key press, microcontroller will go through the process of identifying the key.

Process of Identifying the Key

- Starting from the top row, the microcontroller will ground it by providing a low to row R1 only.
- Now read the columns, if the data read is all 1s, no key in that row is pressed and the process continues for the next row.
- So, now ground the next row, R2. Read the columns, check for any zero and this process continues until the row is identified.
- E.g. In the above case, we will get row 2 in which column is not equal to 1111.
- So, after identification of the row in which the key has been pressed we can easily find out the key by row and column value.

4x4 keypad interfacing with Arduino





```
const byte ROWS = 4; //four rows
const byte COLS = 4; //four columns
char keys[ROWS][COLS] = {
  {'1','2','3','A'},
  {'4','5','6','B'},
  {'7','8','9','C'},
  {'*','0','#','D'}
};
byte rowPins[ROWS] = {13, 12, 11, 10}; //connect to the row pinouts of the keypad
byte colPins[COLS] = {9, 8, 7, 6}; //connect to the column pinouts of the keypad
```

```
void setup() {
  for (byte i = 0; i < ROWS; i++) {
    pinMode(rowPins[i], INPUT_PULLUP);
  }
  for (byte i = 0; i < COLS; i++) {
    pinMode(colPins[i], OUTPUT);
  }
  Serial.begin(9600); //initialize serial communication
}
```

```
void loop() {

 char key = getKey();
  if (key) {
    Serial.println(key);
  }
}
```

```
char getKey() {
  for (byte c = 0; c < COLS; c++) {
    digitalWrite(colPins[c], LOW);
    for (byte r = 0; r < ROWS; r++) {
      if (digitalRead(rowPins[r]) == LOW) {
        digitalWrite(colPins[c], HIGH);
        delay(50);
        return keys[r][c];
      }
    }
    digitalWrite(colPins[c], HIGH);
  }
  return 0;
}
```

Is the pressed key displaying multiple times and why?

Yes…..

- The code lacked a debounce mechanism after detecting a key press. This meant that the code might interpret a single key press as multiple presses due to bouncing.
- Without a debounce delay, a key press might generate multiple state changes in a short period.

Modified Code


```
const byte ROWS = 4; //four rows
const byte COLS = 4; //four columns
char keys[ROWS][COLS] = {
  {'1','2','3','A'},
  {'4','5','6','B'},
  {'7','8','9','C'},
  {'*','0','#','D'}
};
byte rowPins[ROWS] = {13, 12, 11, 10}; keypad
byte colPins[COLS] = {9, 8, 7, 6}; void setup() {
  for (byte i = 0; i < ROWS; i++) {
    pinMode(rowPins[i], INPUT_PULLUP);
  }
  for (byte i = 0; i < COLS; i++) {
    pinMode(colPins[i], OUTPUT);
  }
  Serial.begin(9600); //initialize serial communication
}
```

```
void loop() {

 char key = getKey();
  if (key) {
    Serial.println(key);
  }
}
char getKey() {
  static char lastKey = 0;
  char currentKey = 0;

  for (byte c = 0; c < COLS; c++) {
    digitalWrite(colPins[c], LOW);
    for (byte r = 0; r < ROWS; r++) {
      if (digitalRead(rowPins[r]) == LOW) {
        currentKey = keys[r][c];
      }
    }
    digitalWrite(colPins[c], HIGH);
  }
```

```
if (currentKey != lastKey) {
    delay(50);  // Debounce delay
    lastKey = currentKey;
    return currentKey;
  }

  return 0;
}
```

static char lastKey = 0;: Introduces a static variable lastKey to store the last detected key. This variable will persist its value between function calls, allowing us to compare the current key with the last key.

char currentKey = 0;: Declares a variable currentKey to store the key currently being detected. Initialized to 0.

Debouncing:
Checks if the currentKey is different from the lastKey.
If different, it introduces a delay of 50 milliseconds (delay(50)) to debounce the key. This delay helps prevent registering multiple state changes rapidly.
Updates lastKey with the current key and returns the current key.

Do it Yourself

- Implement a simple calculator using arithmetic operators and concept of 4x4 key pad interfacing with Arduino.

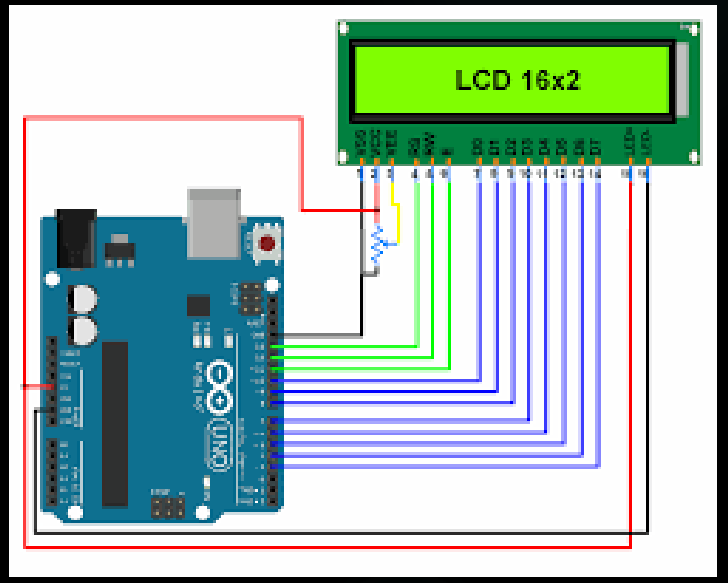
Interfacing 16X2 LCD Display

Interfacing an LCD (Liquid Crystal Display) with an Arduino Uno involves connecting the LCD module to the Arduino and writing code to control the display.

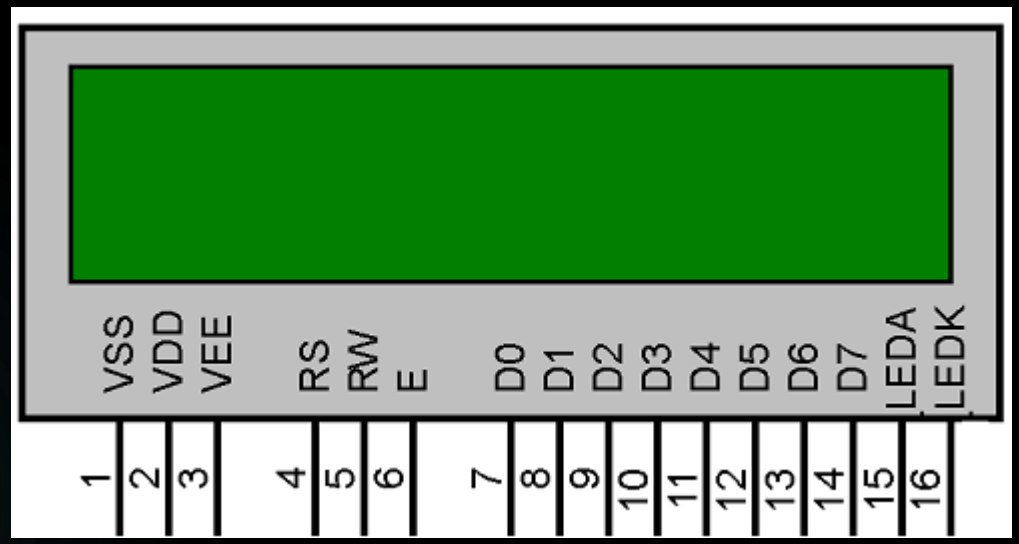
Below is a step-by-step explanation of how to interface an LCD with an Arduino Uno:

Components Required:

- Arduino Uno board
- LCD module (typically 16x2 or 20x4 characters)
- Potentiometer (for contrast adjustment)
- Breadboard and jumper wire



16X2 LCD Display Pin Diagram



Pin Connection:

- Connect the pins of the LCD module to the Arduino Uno as follows:
  - Connect VCC (LCD's power) to +5V on Arduino.
  - Connect GND (LCD's ground) to GND on Arduino.
  - Connect VEE or VO (LCD's contrast control) to the middle pin of the potentiometer.
  - Connect RW (LCD's Read/Write pin) to GND (to set the LCD in write mode).
  - Connect RS (LCD's Register Select pin) to a digital pin (e.g., Pin 12) on Arduino.
  - Connect E (LCD's Enable pin) to a digital pin (e.g., Pin 11) on Arduino.
  - Connect D4-D7 (LCD's data pins) to digital pins (e.g., Pins 5-8) on Arduino.

- **Potentiometer Setup:**
  - The potentiometer is used to adjust the contrast of the LCD. Connect its outer pins to +5V and GND, and the middle pin to VEE on the LCD.

- **Library Installation:**
  - You need the "LiquidCrystal" library to control the LCD. Install it via the Arduino IDE: **Sketch > Include Library > LiquidCrystal**.

Initialization:

In your Arduino sketch, include the LiquidCrystal library and initialize the LCD with the appropriate parameters:

```
#include <LiquidCrystal.h>
LiquidCrystal lcd(12, 11, 5, 6, 7, 8); // RS, E, D4, D5, D6, D7
```

Setup Function:

- In the setup() function, set up the LCD:

```
void setup()
{
   lcd.begin(16, 2); // Set the LCD size (columns x rows)
   lcd.clear();      // Clear the display
   lcd.print("Hello, World!");
}
```

- Remember to connect everything correctly and upload the code to your Arduino Uno. With this setup, the LCD should display the text "Hello, World!" on the first line and "Arduino Uno" on the second line, with a delay and clearing effect in the loop as described.

- **Writing Text:**
  - You can use functions like lcd.print() and lcd.setCursor() to write text and position the cursor.

- **Other Functions:**
  - Explore the library for additional functions like scrolling, custom characters, and more.

Loop Function:

- In the loop() function, you can continuously update the display content:

```
void loop()
{
   lcd.setCursor(0, 1); // Set cursor to the second line
   lcd.print("Arduino Uno");
   delay(1000); // Wait for a second
   lcd.clear(); // Clear the display
   delay(500);  // Wait for half a second
}
```

Temperature Converter


```
void setup() {
  Serial.begin(9600);
}
void loop() {
  float celsius, fahrenheit;
  Serial.println("Enter temperature in Celsius:");
  while(!Serial.available()); // Wait for user input
  celsius = Serial.parseFloat(); // Read temperature in Celsius
  fahrenheit = (celsius * 9.0 / 5.0) + 32.0; // Celsius to Fahrenheit
conversion formula
  Serial.print("Temperature in Fahrenheit: ");
  Serial.println(fahrenheit);
}
```

Display the temperature conversion results on a 16x2 LCD:


```
#include <LiquidCrystal.h>

// Initialize the LCD with the appropriate pins
LiquidCrystal lcd(12, 11, 5, 6, 7, 8);

void setup() {
  lcd.begin(16, 2); // Initialize  LCD:16 col and 2 rows
  Serial.begin(9600); // Initialize serial communication
}

void loop() {
  float celsius, fahrenheit;

  lcd.clear(); // Clear the LCD screen
  lcd.setCursor(0, 0); // Set the cursor to the first line
  lcd.print("Enter temp (C):");
```

```
while (!Serial.available()); // Wait for user input
  celsius = Serial.parseFloat(); // Read temp in Celsius

  fahrenheit = (celsius * 9.0 / 5.0) + 32.0; // Cel to Fahr

  lcd.clear();
  lcd.setCursor(0, 0);
  lcd.print("Temp (C): ");
  lcd.print(celsius);
  lcd.setCursor(0, 1); // Set the cursor to the secondline
  lcd.print("Temp (F): ");
  lcd.print(fahrenheit);
  delay(5000); // Wait for 5 seconds before clearing
the display
}
```

LED Brightness Control with Potentiometer


```
const int potPin = A0; // Analog input pin for the potentiometer
const int ledPin = 9;  // PWM output pin for the LED
void setup() {
  pinMode(ledPin, OUTPUT);
  Serial.begin(9600);
}
void loop() {
  int potValue = analogRead(potPin); // Read potentiometer value (0-1023)
  int brightness = map(potValue, 0, 1023, 0, 255); // Map the value to LED
brightness range (0-255)
  analogWrite(ledPin, brightness); // Set LED brightness using PWM
  Serial.print("Potentiometer Value: ");
  Serial.print(potValue);
  Serial.print(", Brightness: ");
  Serial.println(brightness);
  delay(100); // Small delay for stability
}
```

Display the corresponding brightness value on LCD

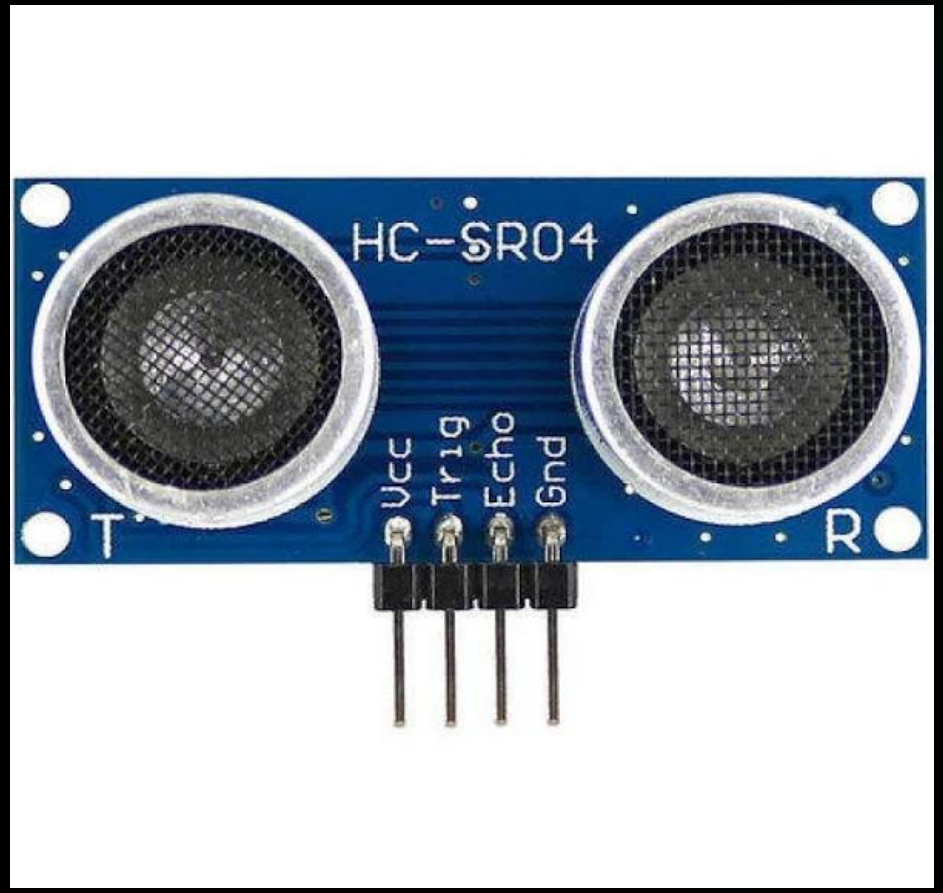
Do it yourself

Ultrasonic sensor to measure distances and display the distance on an LCD or through the serial monitor.

HC SR-04

- The HC-SR04 sensor consists of an ultrasonic transmitter (transducer) and a receiver (transducer).
- The transmitter emits a short burst of ultrasonic waves, which are sound waves with frequencies above the upper limit of human hearing (typically above 20 kHz).
- These ultrasonic waves travel through the air until they encounter an object in their path.
- **Distance Calculation:**

Distance = (Speed of Sound * Time) / 2.



Pins of HC SR04

- **VCC (Voltage Supply):**
  - The VCC pin is used to provide power to the sensor.
  - Connect this pin to the +5V or +3.3V pin on your microcontroller board (such as Arduino) to supply the required operating voltage.
- **Trig (Trigger) Pin:**
  - The Trig pin is used to initiate the ultrasonic pulse transmission.
  - To start the measurement, you need to send a high-to-low pulse (at least 10 microseconds long) to this pin.
  - This pulse prompts the sensor to emit an ultrasonic burst.
- **Echo Pin:**
  - The Echo pin is used to receive the ultrasonic waves that bounce back after hitting an object.
  - The sensor sends out the ultrasonic pulse and then waits for the echo signal.
  - The Echo pin goes high when the sensor receives the reflected signal and stays high for a duration proportional to the time it takes for the signal to travel to the object and back.
- **GND (Ground):**
  - The GND pin is the ground reference for the sensor.
  - Connect this pin to the GND (ground) pin on your microcontroller board.

Step-by-step procedure to interface the HC-SR04 ultrasonic distance sensor with an Arduino:

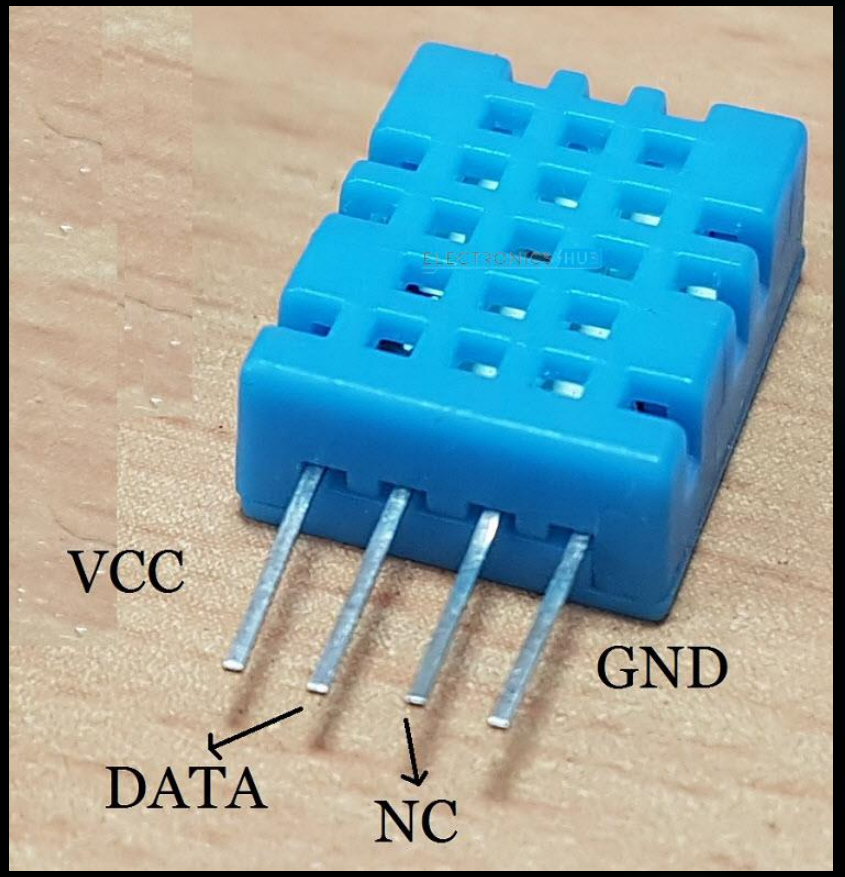
- **Components Needed:**
  - Arduino board (e.g., Arduino Uno)
  - HC-SR04 ultrasonic sensor
  - Breadboard and jumper wires

- **Wiring:**
  - Connect VCC on the HC-SR04 to +5V on the Arduino.
  - Connect GND on the HC-SR04 to GND on the Arduino.
  - Connect TRIG on the HC-SR04 to a digital pin (e.g., Pin 9) on the Arduino.
  - Connect ECHO on the HC-SR04 to another digital pin (e.g., Pin 10) on the Arduino.

```
const int trigPin = 9;
const int echoPin = 10;

void setup() {
  Serial.begin(9600);

  pinMode(trigPin, OUTPUT);
  pinMode(echoPin, INPUT);
}

void loop() {
  digitalWrite(trigPin, LOW);
  delayMicroseconds(2);

  digitalWrite(trigPin, HIGH);
  delayMicroseconds(10);
  digitalWrite(trigPin, LOW);

  long duration = pulseIn(echoPin, HIGH);
  int distance = duration * 0.0343 / 2; // Calculate distance in centimeters

  Serial.print("Distance: ");
  Serial.print(distance);
  Serial.println(" cm");

  delay(1000); // Wait before the next measurement
}
```

DHT11 sensor

- The DHT11 sensor is a widely used digital temperature and humidity sensor that is commonly used in various electronics projects and applications.
- It's a simple sensor that provides accurate temperature and humidity readings.
- **Sensing Elements:** The DHT11 sensor contains two main sensing elements: a humidity sensor and a thermistor (temperature sensor).



Humidity Sensing:

- The humidity sensor in the DHT11 is based on the capacitive sensing principle.
- It consists of two conductive plates with a moisture-absorbing material between them.
- As the surrounding air's humidity changes, the moisture content of the material changes, causing a change in capacitance between the plates.
- The sensor's electronics measure this change in capacitance and convert it into a digital humidity value.

Temperature Sensing:

- The temperature sensing element in the DHT11 is a thermistor—a type of resistor whose resistance changes with temperature.
- The thermistor's resistance decreases as the temperature increases and vice versa.
- The sensor circuitry measures this resistance change and converts it into a digital temperature value.

Signal Processing and Communication:

- The DHT11 sensor has an integrated microcontroller that processes the signals from the humidity and temperature sensing elements.
- It converts the analog sensor outputs into digital values, and these values are then packaged into a simple digital signal format.

Timing and Data Format:

- The DHT11 sensor sends data in a specific format: it starts by sending a low-to-high signal to signal the beginning of data transmission.
- It then sends 40 bits of data (8 bits for integral humidity, 8 bits for decimal humidity, 8 bits for integral temperature, 8 bits for decimal temperature, and 8 bits for the checksum).

Example

Consider the data received from the DHT11 Sensor is

00100101 00000000 00011001 00000000 00111110.

This data can be separated based on the above mentioned structure as follows

00100101	00000000	00011001	00000000	00111110
High Humidity	Low Humidity	High Temperature	Low Temperature	Checksum (Parity)

Correctness of data received

- In order to check whether the received data is correct or not, we need to perform a small calculation.
- Add all the integral and decimals values of RH and Temperature and check whether the sum is equal to the checksum value i.e. the last 8 – bit data.

00100101 + 00000000 + 00011001 + 00000000 = 00111110

- This value is same as checksum and hence the received data is valid.

- Now to get the RH and Temperature values, just convert the binary data to decimal data.

RH = Decimal of 00100101 = 37%

Temperature = Decimal of 00011001 = 25^0C

Digital Output:

- The DHT11 sensor communicates with the outside world using a single-wire digital communication protocol.
- It sends out a series of pulses to transmit data. Each bit of data is represented by the duration of a specific pulse.

Logical Operators:

- AND: &&
- OR: ||
- NOT: !

Temperature Alert System


```
const int temperatureThreshold = 30;
const int fanPin = 9;
const int buzzerPin = 10;
void setup() {
  pinMode(fanPin, OUTPUT);
  pinMode(buzzerPin, OUTPUT);
  Serial.begin(9600);
}
void loop() {
  int temperature = readTemperature();
  if (temperature > temperatureThreshold &&
temperature < 100)
 {
    digitalWrite(fanPin, HIGH);
    digitalWrite(buzzerPin, HIGH);
    Serial.println("Temperature is too high! Cooling
and sounding the alarm.");
  }
else
{
digitalWrite(fanPin, LOW);
    digitalWrite(buzzerPin, LOW);
  }
  delay(1000); // Delay to avoid rapid checks
}
int readTemperature() {
  // Replace this function with your actual
temperature reading code
  // For simplicity, we return a constant value here.
  return 25;
}
```

Password-Protected Access


```
const int buttonPin = 2;
const int ledPin = 13;

const int correctPassword = 1234;
int enteredPassword = 0;

void setup() {
  pinMode(buttonPin, INPUT_PULLUP);
  pinMode(ledPin, OUTPUT);
  Serial.begin(9600);
}

while (Serial.available() < 4); // Wait for user input
  for (int i = 0; i < 4; i++) {
    password = password * 10 + (Serial.read() - '0'); // Convert ASCII to integer
  }
  return password;
}
```

```
void loop() {
  if (digitalRead(buttonPin) == LOW) {
    delay(100); // Debounce delay
    enteredPassword = getPassword();
    if (enteredPassword == correctPassword) {
      digitalWrite(ledPin, HIGH);
      Serial.println("Access granted.");
      delay(2000);
      digitalWrite(ledPin, LOW);
    } else {
      Serial.println("Access denied.");
    }
  }
}
```

```
int getPassword() {
  int password = 0;
  Serial.println("Enter the password (4-digit number):");
  while (Serial.available() < 4); // Wait for user input
  for (int i = 0; i < 4; i++) {
    password = password * 10 + (Serial.read() - '0'); // Convert ASCII to integer
  }
  return password;
}
```

- pinMode(buttonPin, INPUT_PULLUP);: This line configures the buttonPin as an input pin with the internal pull-up resistor enabled. The pull-up resistor ensures that the button reads HIGH when not pressed and LOW when pressed.

Comparison Operators:

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

Assignment Operators:

- Assignment: =
- Addition assignment: +=
- Subtraction assignment: -=
- Multiplication assignment: *=
- Division assignment: /=
- Modulus assignment: %=

Bitwise Operators:

- Bitwise AND: &
- Bitwise OR: |
- Bitwise XOR: ^
- Bitwise NOT: ~
- Left shift: <<
- Right shift: >>

Loops

- In embedded systems programming for Arduino, loops are used to repeat a block of code multiple times.
- Here are two commonly used loops: the ***for loop*** and the ***while loop***.

for Loop


```
for (initialization; condition; increment) {
    // Code to be executed in each iteration
}
```

- Printing numbers from 1 to 5 using a for loop.

```
for (int i = 1; i <= 5; i++) {
    Serial.println(i);
    delay(1000); // Delay for 1 second
}
```

while Loop

- The while loop is useful when the number of iterations is not known in advance, and the loop continues until a certain condition is met.

```
while (condition) {
    // Code to be executed in each iteration
}
```

Reading Analog Input


```
int analogPin = A0;
int threshold = 500;

void setup() {
  Serial.begin(9600);
}

void loop() {
  int sensorValue = analogRead(analogPin);

  while (sensorValue > threshold) {
    Serial.println("Object detected!");
    delay(1000);

    // Read the sensor value again
    sensorValue = analogRead(analogPin);
  }
}
```

UNIT-3

Serial and Parallel Interfaces

Digital and Analog data

- Analog and digital signals are used to transmit information (such as any audio or video), usually through electric signals.
- In digital technology, the translation of information is into binary format (either 0 or 1) and information is translated into electric pulses of varying amplitude in analog technology.

Features of Digital Systems:

- Uses binary code: Digital systems use binary code, which is a combination of zeros and ones, to represent information.
- Accuracy: Digital systems are more accurate than analog systems because the information is represented in a precise and consistent manner.
- Processing speed: Digital systems are capable of processing large amounts of data quickly and accurately.
- Noise immunity: Digital systems are immune to noise and interference, which means that the transmitted information is less likely to be corrupted.

Features of Analog Systems:

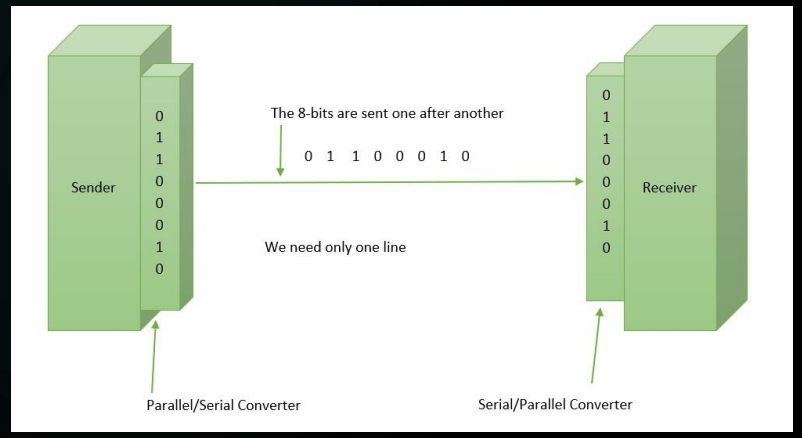
- Uses continuous signals: Analog systems use continuous signals to represent information, such as electrical voltages or sound waves.
- Real-world representation: Analog systems are better suited for representing real-world phenomena such as sound and light, which are continuous in nature.
- Smooth transitions: Analog systems provide smooth and continuous transitions between different values, which can be important in certain applications such as music or video.
- Complexity: Analog systems can be more complex than digital systems due to the need for additional circuitry to process and transmit the signals

Serial Vs Parallel Data Transfer

There are two methods used for transferring data between computers which are: Serial Transmission and Parallel Transmission.

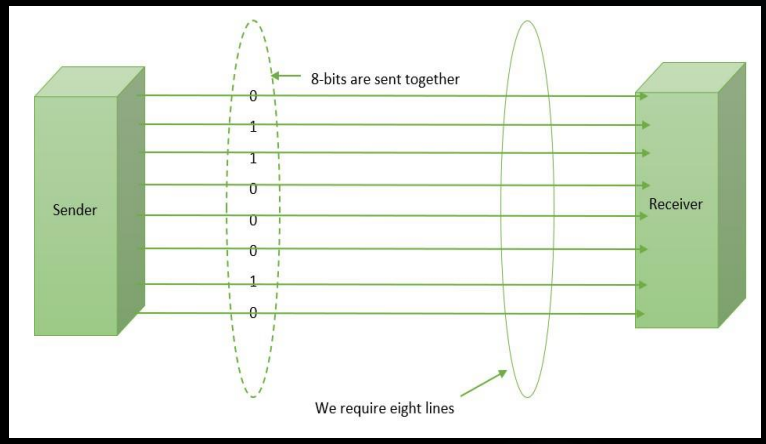
Serial Transmission:

- In Serial Transmission, data-bit flows from one computer to another computer in bi-direction.
- In this transmission, one bit flows at one clock pulse. In Serial Transmission, 8 bits are transferred at a time having a start and stop bit.



Parallel Transmission:

- In Parallel Transmission, many bits are flow together simultaneously from one computer to another computer.
- Parallel Transmission is faster than serial transmission to transmit the bits. Parallel transmission is used for short distance.



Difference between Serial and Parallel Transmission:

Serial Communication

- A single communication link is used to transfer data from one end to another.
- Data(bit) flows in bi-direction.
- Cost-efficient.
- One bit transferred at one clock pulse.
- Slow in comparison of parallel transmission.
- Used for long-distance.
- Full duplex as sender can send as well as receive the data.
- Converters are required

Parallel Communication

- Multiple parallels links used to transmit the data.
- Data flows in multiple lines.
- Not cost-efficient.
- Eight bits transferred at one clock pulse.
- Fast in comparison of serial transmission.
- Used for short distance.
- Half-duplex since the data is either send or receive.
- No converters are required.

Communication Protocols

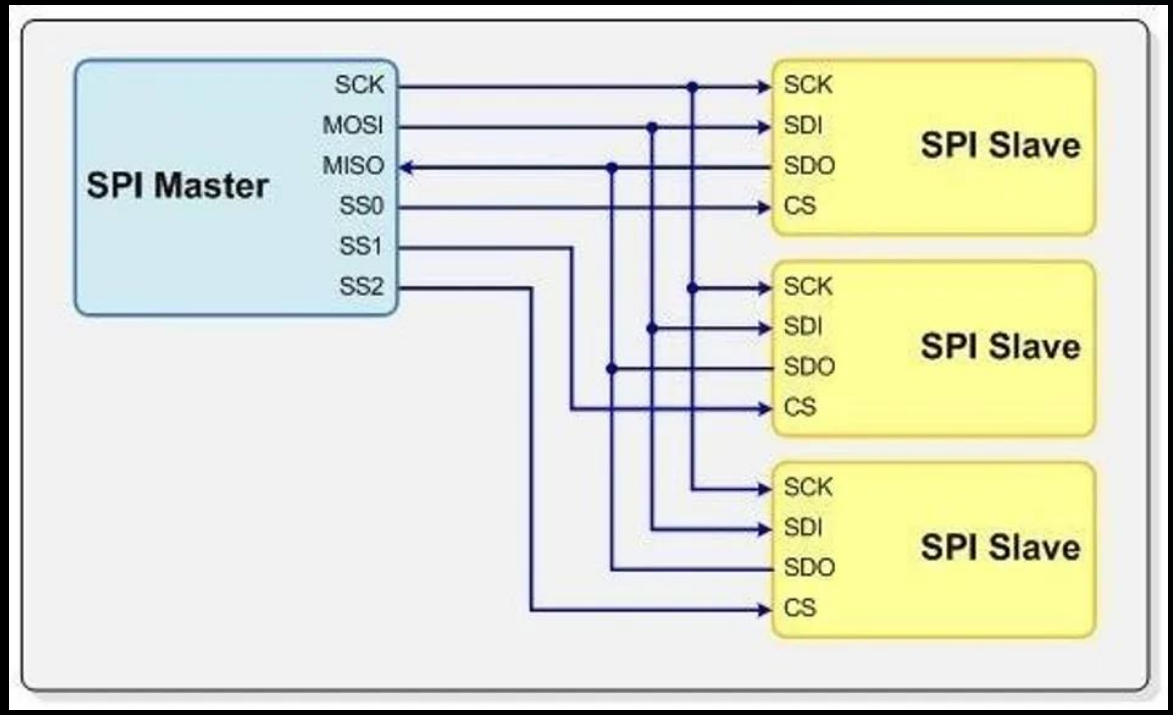
- Embedded systems rely on various protocols for effective data interaction.
- Three commonly used protocols are: Serial Peripheral Interface (SPI), Inter-Integrated Circuit (I2C), and Universal Asynchronous Receiver/Transmitter (UART) is required.
- Each of these protocols serves distinct purposes and is characterized by unique specifications and operational principles.
- An understanding of these technologies is crucial for anyone involved in electronics design or embedded systems development, as these protocols often form the backbone of device communication within such systems.
- They enable the interaction between microcontrollers and various peripheral devices such as sensors, memory devices, and display modules, each with unique data transmission needs.
- By exploring their working principles, advantages, and disadvantages, we can make informed decisions when selecting the most suitable protocol for the specific use-cases.
- For more details: https://www.wevolver.com/article/spi-vs-i2c-vs-uart-in-depth-comparison

Serial Peripheral Interface (SPI)

- SPI is a synchronous serial communication interface specification used for short-distance communication primarily in embedded systems.
- It facilitates full-duplex communication between a master device and one or multiple slave devices.
- SPI is built on four fundamental lines: Master Out Slave In **(MOSI)**, Master In Slave Out **(MISO)**, Serial Clock **(SCLK)**, and Slave Select **(SS)**.
- The MOSI line carries data from the master device to the slave device, while the MISO line carries data in the opposite direction.
- The SCLK line, controlled by the master, paces the communication, while the SS line is used by the master to select which slave device it intends to communicate with.

Basic Principles

- The SPI protocol was developed by Motorola in the mid-1980s, SPI was designed to provide a simple and efficient means of data exchange between microcontrollers and peripherals.
- At its core, SPI communication relies on shifting data between Master and Salve, enabling full-duplex communication.
- Each transmission consists of at least two devices: a master device, which initiates and controls the communication, and a slave device, which responds to the master.



Working

- In a typical SPI transmission, the master generates a clock signal on the ***SCLK*** line and selects a slave device by setting its ***SS*** line to a low voltage level.
- The master and slave then exchange data simultaneously, with the master sending data to the slave on the ***MOSI*** line and the slave sending data to the master on the ***MISO*** line.
- The transmission continues for as many clock cycles as necessary, with the master finally ending the communication by returning the ***SS*** line of the selected slave to a high voltage level.
- By controlling the clock signal and the ***SS*** line, the master can dictate the pace of communication and select which devices to interact with, making SPI an inherently flexible and efficient means of data exchange.

Speed or Data rates

- Compared to other serial communication protocols like UART and I2C, SPI offers significantly higher data transfer rates.
- The SPI communication speed depends on the clock frequency used.
- For example, a clock frequency of 4 MHz allows for a maximum data rate of 4 Mbps.
- This high data transfer rate makes SPI an excellent choice for applications requiring fast and efficient data exchange.

Limitations

- However, it's worth noting that the lack of an in-built error-checking mechanism and the need for a dedicated **SS** line for each slave device are among the limitations of SPI that system designers need to consider.
- Another consideration is that SPI lacks built-in error-checking mechanisms. Therefore, additional error checking and handling procedures may need to be implemented at the software level to ensure data integrity.

Advantages and Disadvantages of SPI

Advantages

- High-speed data transfer.
- Full-duplex Communication.
- Versatility and ease of implementation.
- Arbitrary data size.
- Support for multiple slave devices.

Disadvantages

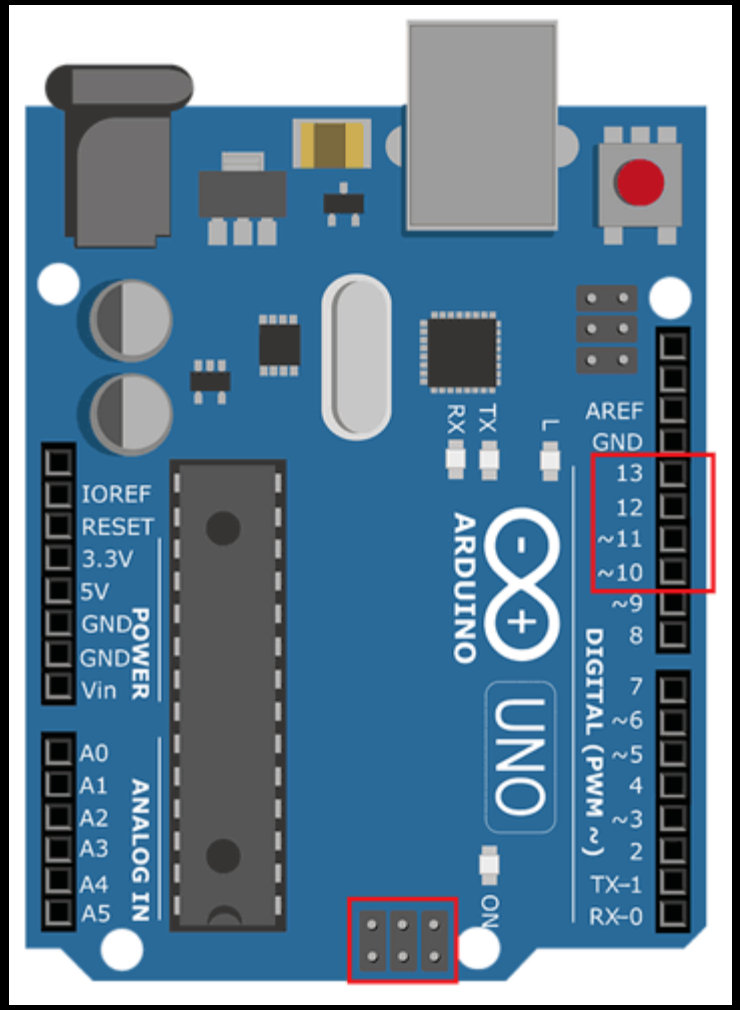
- Lack of in-built error checking.
- Limited scalability.
- Unsuitable for long-distance communication.
- More power consumption.

For more examples:
https://docs.arduino.cc/tutorials/generic/introduction-to-the-serial-peripheral-interface
https://www.circuitbasics.com/how-to-set-up-spi-communication-for-arduino/

For more information:
https://www.wevolver.com/article/spi-vs-i2c-vs-uart-in-depth-comparison

Programming SPI Protocol

- It is expected that the terms MOSI/MISO and SS will be changed to SDI(Serial Data In) /SDO(Serial Data Out)and CS(Chip Select) respectively.
- A SPI has a master/Slave communication by using four lines. A SPI can have only one master and can have multiple slaves. A master is usually a microcontroller and the slaves can be a microcontroller, sensors, ADC, DAC, LCD etc.



SPI Line	Pin
MOSI	11
MISO	12
SCK	13
SS	10

DIY

- Click the following link and implement the same using tinker Cad

https://circuitdigest.com/microcontroller-projects/arduino-spi-communication-tutorial

Arduino SPI library used in Arduino IDE.

- The library **<SPI.h>** is included in the program for using the following functions for SPI communication.
- **SPI.begin()**
  - USE: To Initialize the SPI bus by setting SCK, MOSI, and SS to outputs, pulling SCK and MOSI low, and SS high.
- **SPI.setClockDivider(divider)**
  - USE: To Set the SPI clock divider relative to the system clock. The available dividers are 2, 4, 8, 16, 32, 64 or 128.
  - Dividers: SPI_CLOCK_DIV2, SPI_CLOCK_DIV4, SPI_CLOCK_DIV8, SPI_CLOCK_DIV16, SPI_CLOCK_DIV32, SPI_CLOCK_DIV64, SPI_CLOCK_DIV128
- **SPI.attachInterrupt(handler)**
  - USE: This function is called when a slave device receives data from the master.
- **SPI.transfer(val)**
  - **USE:** This function is used to simultaneously send and receive the data between master and slave.

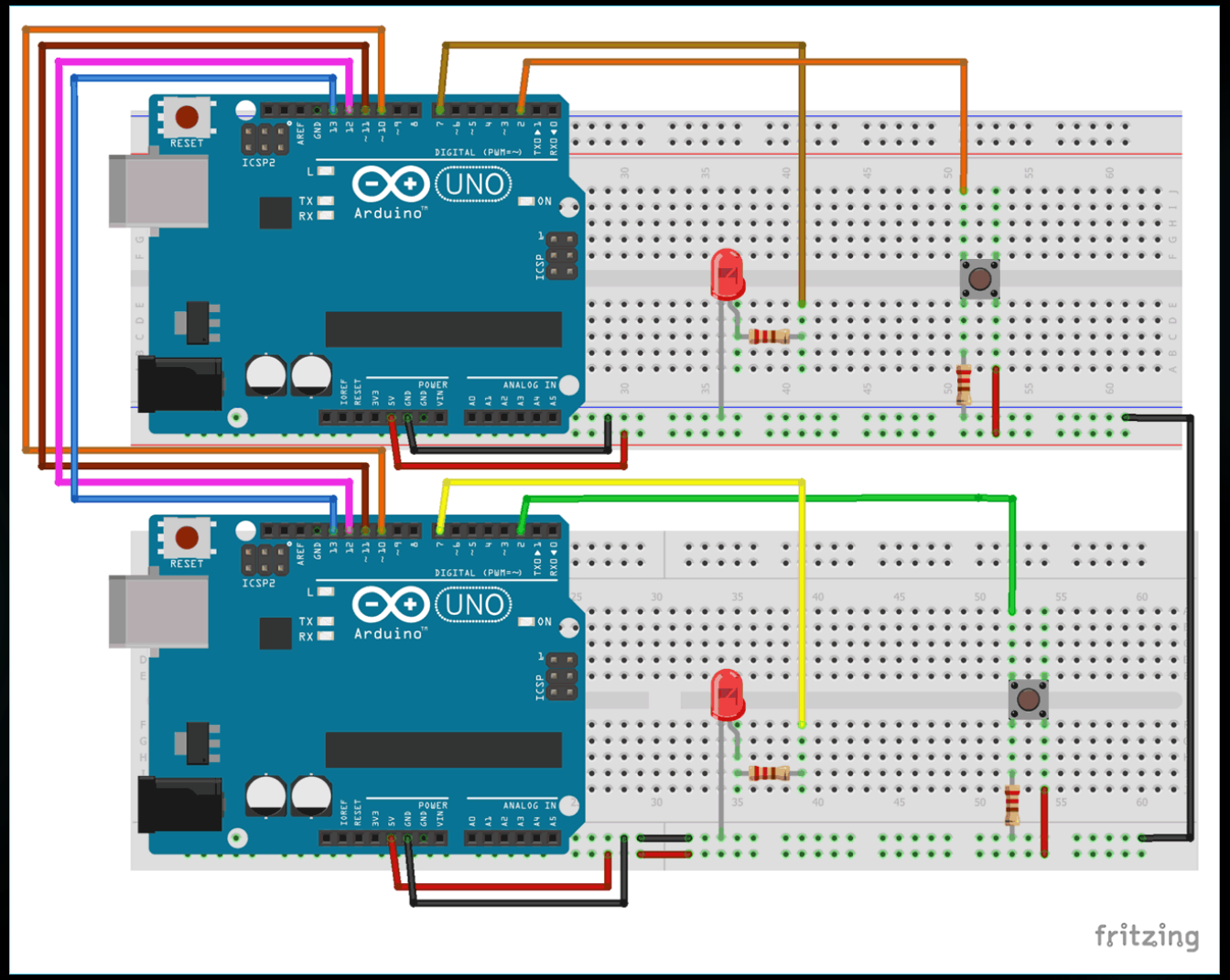
SPI Protocol for communication between two Arduinos.

- In this tutorial we will use two Arduino Uno,s one as master and other as slave.
- Both Arduino are attached with a LED & a push button separately.
- Master LED can be controlled by using slave Arduino's push button and slave Arduino's LED can be controlled by master Arduino's push button using SPI communication protocol present in arduino.

Components Required

- Arduino UNO (2)
- LED (2)
- Push Button (2)
- Resistor 10k (2)
- Resistor 2.2k (2)
- Breadboard
- Connecting Wires

Arduino SPI Communication Circuit Diagram



Arduino SPI Master Programming Explanation


```
#include<SPI.h>
#define ipbutton 2
int buttonvalue, x;
void setup (void)
{
Serial.begin(115200); //Starts Serial Communication at Baud Rate 115200
pinMode(ipbutton,INPUT); //Sets pin 2 as input
pinMode(LED,OUTPUT); //Sets pin 7 as Output
SPI.begin(); //Begins the SPI commnuication
SPI.setClockDivider(SPI_CLOCK_DIV8);//Sets clock for SPI communication at 8 (16/8=2Mhz)
digitalWrite(SS,HIGH);// Setting SlaveSelect as HIGH (So master doesnt connnect with slave)
}
```

```
void loop(void)
{
  byte Mastersend,Mastereceive;
  buttonvalue = digitalRead(ipbutton);
//Reads the status of the pin 2
 if(buttonvalue == HIGH) //Logic for
Setting x value (To be sent to slave)
depending upon input from pin 2
  {
    x = 1;
  }
  else
  {
    x = 0;
  }
digitalWrite(SS, LOW);//Starts communication with
Slave connected to master
Mastersend = x;
Mastereceive=SPI.transfer(Mastersend); //Send the
mastersend value to slave also receives value from
slave
if(Mastereceive == 1) //Logic for setting the LED
output depending upon value received from slave
  {
    digitalWrite(LED,HIGH);              //Sets pin 7 HIGH
    Serial.println("Master LED ON");
  }
  else
  {
   digitalWrite(LED,LOW);               //Sets pin 7 LOW
   Serial.println("Master LED OFF");
  }
  delay(1000);
}
```

Arduino SPI Slave Programming Explanation


```
#include<SPI.h>
#define LEDpin 7
#define buttonpin 2
volatile boolean received;
volatile byte Slavereceived,Slavesend;
int buttonvalue;
int x;
void setup()
{
  Serial.begin(115200);
  pinMode(buttonpin,INPUT);              // Setting pin 2 as INPUT
  pinMode(LEDpin,OUTPUT);                // Setting pin 7 as OUTPUT
  pinMode(MISO,OUTPUT);                  //Sets MISO as OUTPUT (Have to Send
data to Master IN
  SPCR |= _BV(SPE);                      //Turn on SPI in Slave Mode
  received = false;
  SPI.attachInterrupt();                 //Interuupt ON is set for SPI
commnucation
}
ISR (SPI_STC_vect)                       //Inerrrput routine function
{
Slavereceived = SPDR;          // Value received from master if store in
variable slavereceived
received = true;                         //Sets received as True
}
```

Inter-Integrated Circuit (I2C)

- A good way of adding complexity of features to your projects without adding complexity of wiring, is to make use of the Inter-integrated circuit (I2C) protocol.
- The I2C protocol is supported on all Arduino boards. It allows you to connect several peripheral devices, such as sensors, displays, motor drivers, and so on, with only a few wires.
- Giving you lots of flexibility and speeding up your prototyping, without an abundancy of wires.
- Keep reading to learn about how it works, how it is implemented into different standards, as well as how to use the Wire Library to build your own I2C devices.

What Is I2C?

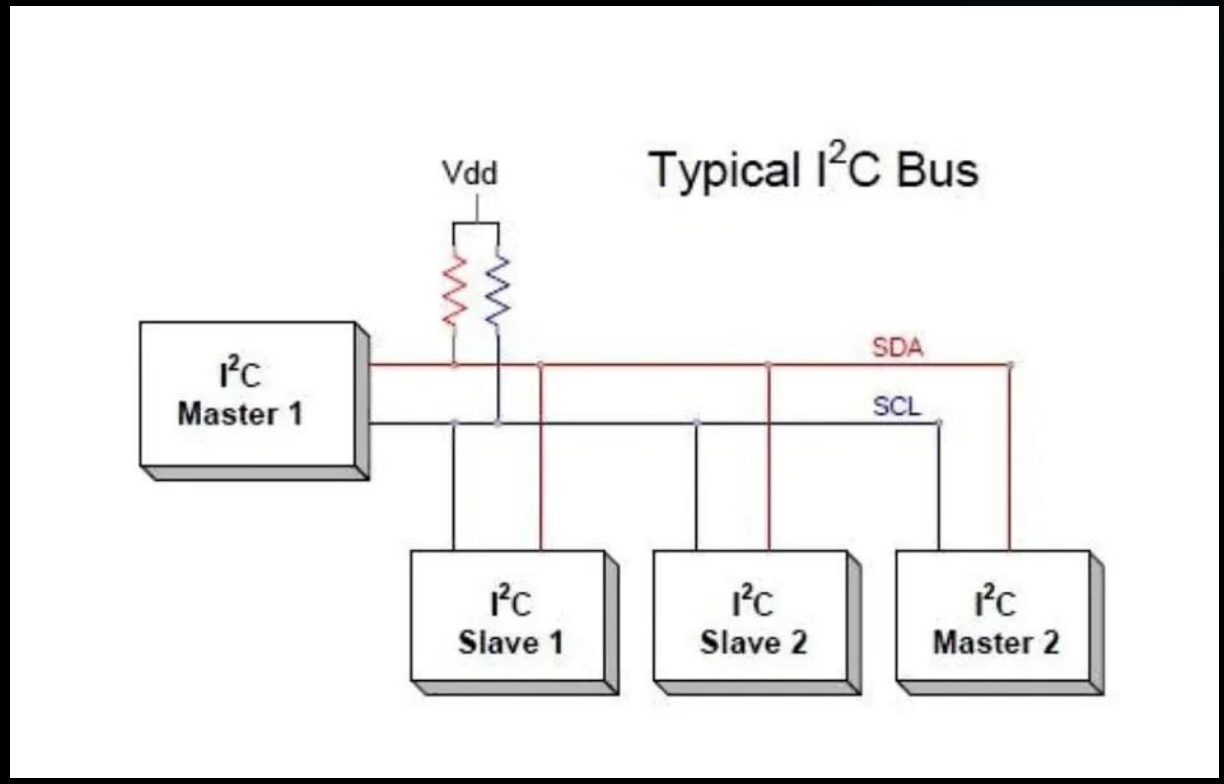
- he I2C protocol involves using two lines to send and receive data: a serial clock pin (SCL) that the Arduino Controller board pulses at a regular interval, and a serial data pin (SDA) over which data is sent between the two devices.
- In I2C, there is one controller device, with one or more peripheral devices connected to the controllers SCL and SDA lines.

Inter-Integrated Circuit (I2C)

- The Inter-Integrated Circuit (I2C) is a **synchronous serial communication protocol** initially developed by Philips Semiconductor (now NXP Semiconductors) in the early 1980s.
- It is specifically designed for **short-distance, intra-board communication** between different components on the same circuit board, enabling the exchange of information.
- I2C utilizes a clock signal for synchronization purposes and offers unique features **that allow multiple masters and slaves** to coexist on a single communication setup, enabling flexible and complex architectures.

Basic Principles

- The protocol utilizes two bidirectional open-drain lines known as the **Serial Data Line *(SDA)*** and the **Serial Clock Line *(SCL)***, which are pulled up with resistors and used by all devices on the I2C bus.
- Devices on the bus can act as senders (masters) or receivers (slaves).
- The ***SDA*** line is responsible for carrying data, while the ***SCL*** line provides the clock signal that synchronizes the data transfer.



Working

- In an I2C transaction, the master initiates the process by creating a START condition, pulling the SDA line low while the SCL line is high.
- This signal alerts all devices on the bus that a transmission is about to begin.
- The master then sends the 7 or 10-bit address of the slave device it wants to communicate with, accompanied by a bit indicating whether it intends to write to the slave (0) or read from it (1).
- Upon receiving its address, the addressed slave acknowledges the receipt by pulling the SDA line low during the next clock pulse.
- Once the master receives this acknowledgment, it can proceed with sending or receiving data.
- Each byte of data is followed by an acknowledgment bit. When the master completes the transmission or reception of data, it generates a STOP condition, where it releases the SDA line to go high while the SCL line remains high.
- I2C protocol incorporates error-checking measures. These include the use of acknowledgment, arbitration and collision detection mechanisms in multi-master systems.

Data Exchange b/w Master & Slave

- As the clock line changes from low to high (known as the rising edge of the clock pulse), a single bit of information is transferred from the board to the I2C device over the SDA line.
- As the clock line keeps pulsing, more and more bits are sent until a sequence of a 7 or 8 bit address, and a command or data is formed.
- When this information is sent - bit after bit -, the called upon device executes the request and transmits it's data back - if required - to the board over the same line using the clock signal still generated by the Controller on SCL as timing.
- Because the I2C protocol allows for each enabled device to have it's own unique address, and as both controller and peripheral devices to take turns communicating over a single line, it is possible for your Arduino board to communicate (in turn) with many devices, or other boards, while using just two pins of your microcontroller.

An I2C message on a lower bit-level looks something like this:

- The controller sends out instructions through the I2C bus on the data pin (SDA), and the instructions are prefaced with the address, so that only the correct device listens.
- Then there is a bit signifying whether the controller wants to read or write.
- Every message needs to be acknowledged, to combat unexpected results, once the receiver has acknowledged the previous information it lets the controller know, so it can move on to the next set of bits.
- 8 bits of data, Another acknowledgement bit, 8 bits of data, Another acknowledgement bit.

Wire Library

- But how does the controller and peripherals know where the address, messages, and so on starts and ends? That's what the SCL wire is for.
- It synchronizes the clock of the controller with the devices, ensuring that they all move to the next instruction at the same time.
- However, you are nearly never going to actually need to consider any of this, in the Arduino ecosystem we have the *Wire library* that handles everything for you.

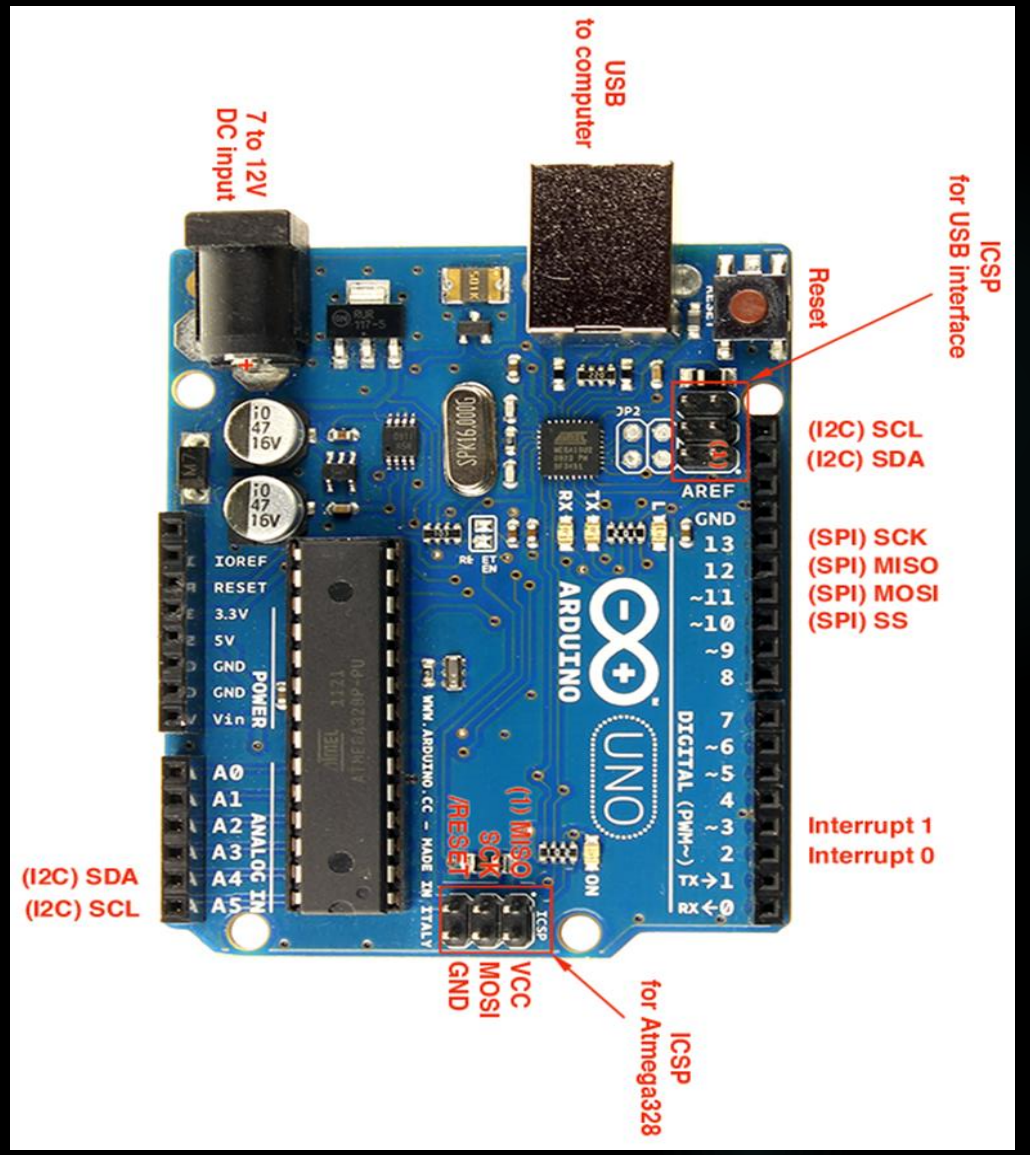
Wire Library

- The Wire library is what Arduino uses to communicate with I2C devices.
- It is included in all board packages, so you don't need to install it manually in order to use it.
- To see the full API for the Wire library, visit its documentation page.

begin() - Initialise the I2C bus
end() - Close the I2C bus
requestFrom()- Request bytes from a peripheral device
beginTransmission() - Begins queueing up a transmission
endTransmission() - Transmit the bytes that have been queued and end the transmission
onReceive() - Register a function to be called when a peripheral receives a transmission
onRequest() - Register a function to be called hwen a controller requests data

Arduino I2C Pins

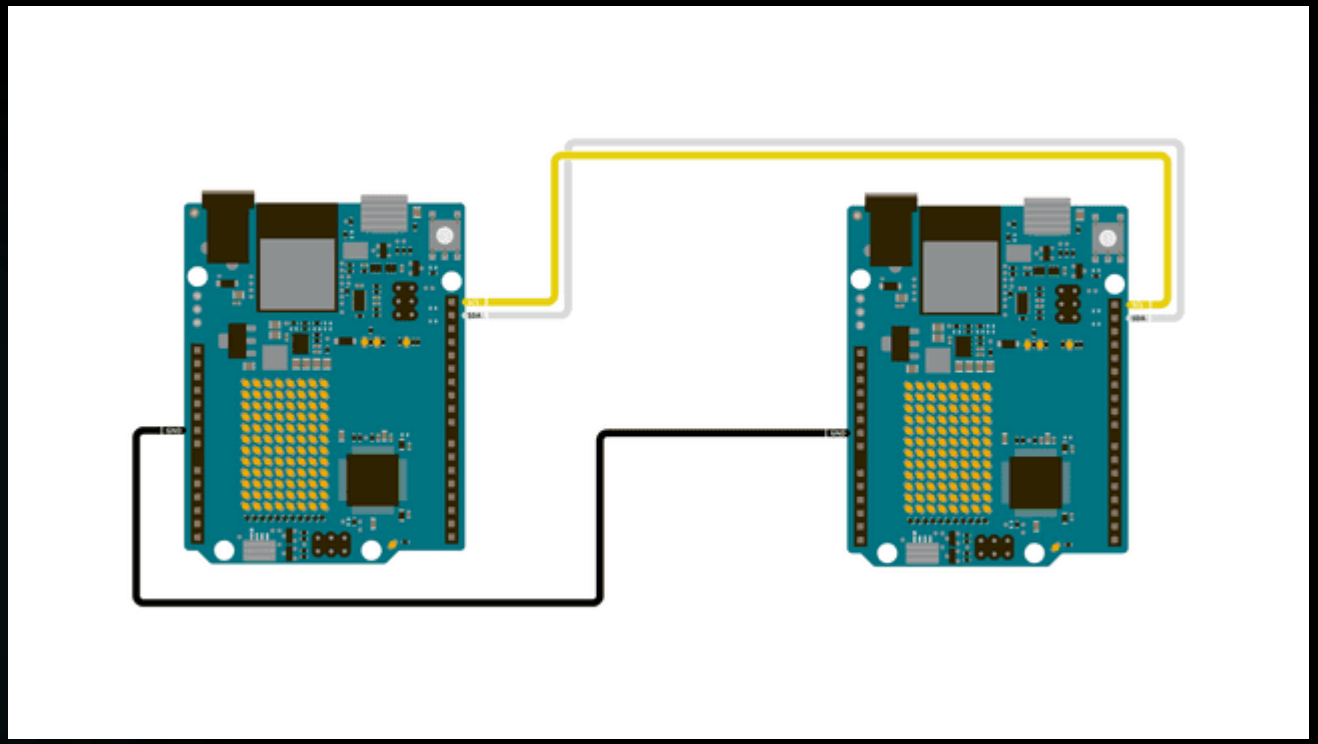
Form factor	SDA	SCL
UNO	SDA/A4	SCL/A5



Example: Controller Reader/Peripheral Sender

- In some situations, it can be helpful to set up two (or more!) Arduino boards to share information with each other.
- In this example, two boards are programmed to communicate with one another in a ***Controller Reader/Peripheral Sender configuration*** via the I2C synchronous serial protocol.
- Several functions of Arduino's Wire Library are used to accomplish this. Arduino 1, the Controller, is programmed to request, and then read, 6 bytes of data sent from the uniquely addressed Peripheral Arduino.
- Once that message is received, it can then be viewed in the Arduino Software (IDE) serial monitor window.

Arduino Boards connected via I2C:
To Perform Controller Reader/Peripheral Sender



Controller Reader Sketch


```
#include <Wire.h>

void setup() {

  Wire.begin();        // join i2c bus (address optional for
master)

  Serial.begin(9600);  // start serial for output

}

void loop() {
  Wire.requestFrom(8, 6);    // request 6 bytes from
peripheral device #8

  while (Wire.available()) { // peripheral may send less
than requested
    char c = Wire.read(); // receive a byte as character
    Serial.print(c);         // print the character
  }
  delay(500);
}
```

Explantion

- **#include <Wire.h>:** This line includes the Wire library, which is necessary for I2C communication.
- **Wire.begin();:** Initializes the Wire library, allowing the Arduino to join the I2C bus. The master doesn't need a specific address in this case.
- **Wire.requestFrom(8, 6);:** Sends a request to the I2C peripheral device with address 8, asking for 6 bytes of data.
- **while (Wire.available()) {:** Checks if there is data available to be read from the peripheral.
- **char c = Wire.read();:** Reads a byte of data from the peripheral and stores it in the variable c.
- **Serial.print(c);:** Prints the received character to the Serial Monitor.
- **delay(500);:** Introduces a delay of 500 milliseconds before the next iteration of the loop. This delay allows time for the peripheral device to respond and for the Serial Monitor to display the received data.

Peripheral Sender Sketch


```
#include <Wire.h>

void setup() {

  Wire.begin(8); // join i2c bus with address #8

  Wire.onRequest(requestEvent); // register event

}

void loop() {
  delay(100);
}

// function that executes whenever data is requested by master
// this function is registered as an event, see setup()
void requestEvent() {
  Wire.write("hello "); // respond with message of 6 bytes
  // as expected by master
}
```

Explanation

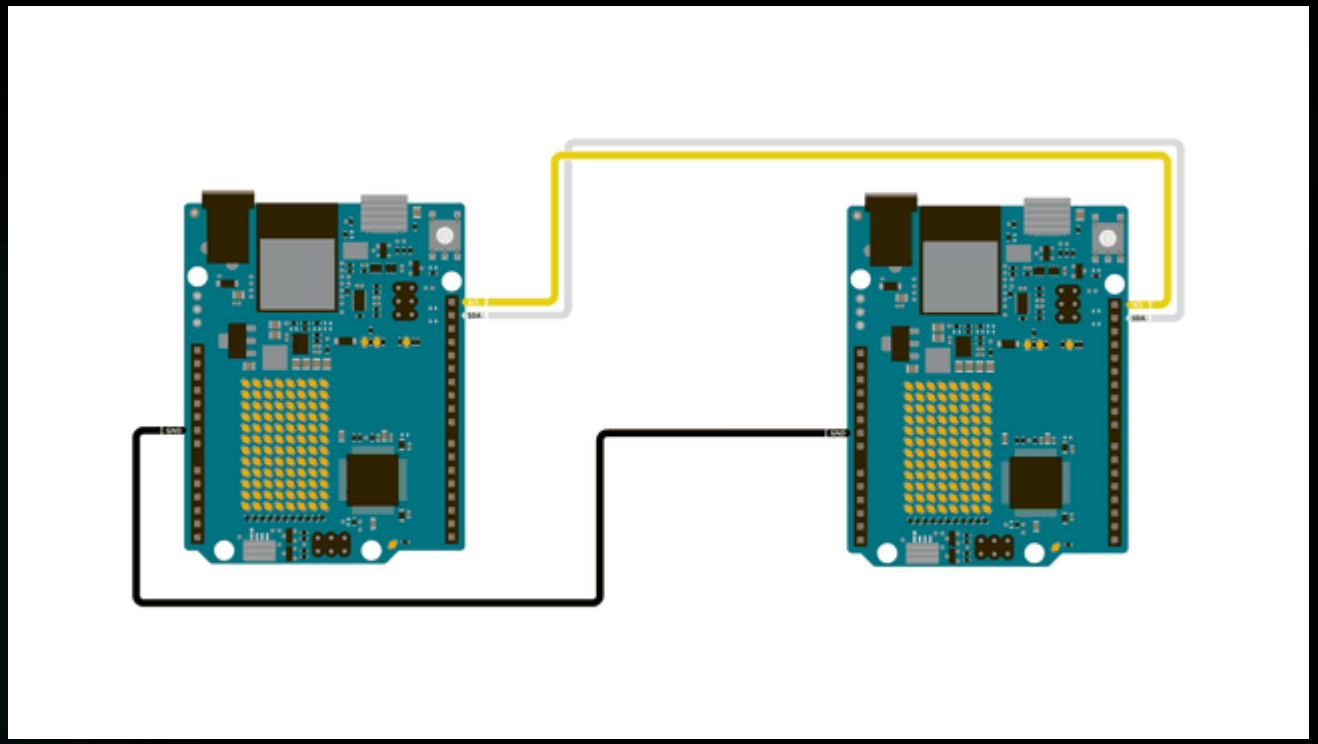
- **Wire.begin(8);:** Initializes the Wire library and joins the I2C bus with the peripheral device's address set to 8. This is the address that the master device will use to communicate with this peripheral.
- **Wire.onRequest(requestEvent);:** Registers the requestEvent function as an event handler. This function will be executed whenever the master device requests data from this peripheral.
- **delay(100);:** Introduces a delay of 100 milliseconds between iterations of the loop. This delay is used to prevent the loop from executing too rapidly.
- **void requestEvent() {:** This is a custom function named requestEvent. It is automatically called by the Wire library when the master device requests data from the peripheral.
- **Wire.write("hello ");:** In response to the master's request, this line sends the string "hello " (containing 6 characters) back to the master using the Wire library's write function.

DIY

- Understand and implement ***Controller Writer/Peripheral Receiver***
- Code and interfacing diagrams given in following slides.

- For more examples: https://docs.arduino.cc/learn/communication/wire

Arduino Boards connected via I2C:
To Perform Controller Writer/Peripheral Receiver



Controller Writer Sketch


```
#include <Wire.h>

void setup()
{
  Wire.begin(); // join i2c bus (address optional for master)
}

byte x = 0;

void loop()
{
  Wire.beginTransmission(4); // transmit to device #4
  Wire.write("x is ");        // sends five bytes
  Wire.write(x);              // sends one byte
  Wire.endTransmission();    // stop transmitting

  x++;
  delay(500);
}
```

Peripheral Receiver Sketch


```
#include <Wire.h>

void setup()
{
  Wire.begin(4);        // join i2c bus with address #4
  Wire.onReceive(receiveEvent); // register event
  Serial.begin(9600); // start serial for output
}

void loop()
{
  delay(100);
}

// function that executes whenever data is received from master
// this function is registered as an event, see setup()
void receiveEvent(int howMany)
{
  while(1 < Wire.available()) // loop through all but the last
  {
    char c = Wire.read(); // receive byte as a character
    Serial.print(c);         // print the character
  }
  int x = Wire.read();    // receive byte as an integer
  Serial.println(x);         // print the integer
}
```

Speed or Data rates

- I2C supports different speed modes to accommodate various application requirements. Standard I2C devices typically support data rates up to 100K bits per second.
- Fast-mode devices extend this to 400K bits per second, while high-speed devices can achieve data rates of up to 3.4Mbits per second.
- These different speed modes enable I2C to cater to a wide range of devices, from slower, low-power components to faster, more complex devices.

Advantages and Disadvantages of I2C

Advantages

- Simplicity and Wire Efficiency.
- Multi-Master and Multi-Slave Configuration.
- Addressing Scheme.
- Speed Modes.
- Synchronous Communication.

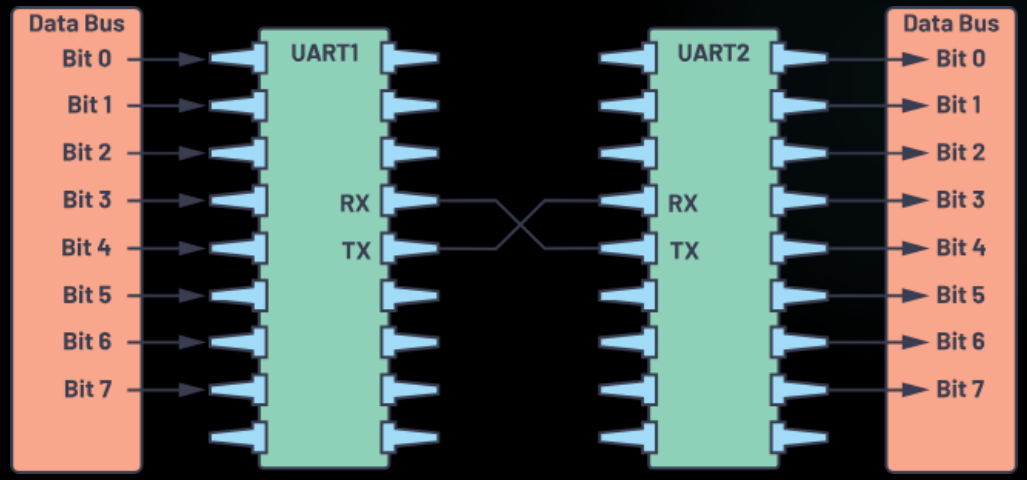
Disadvantages

- Distance Limitation.
- Speed Limitation.
- Pull-up Resistors.
- Addressing Limitation.
- Clock Stretching

For more information:
https://www.wevolver.com/article/spi-vs-i2c-vs-uart-in-depth-comparison

Universal Asynchronous Receiver/Transmitter (UART)

- UART is a fundamental hardware communication protocol extensively used for **asynchronous** serial communication.
- It serves as a vital interface between microcontrollers and peripheral devices, facilitating the exchange of data through serial transmission.
- UART converts data bytes from the processor into a continuous stream of data bits, which are then transmitted serially over a single communication line.
- Similarly, incoming data is converted in reverse order for processing by the receiving device.
- Unlike synchronous communication protocols, UART does not rely on a shared clock signal for data synchronization.



Frame Format

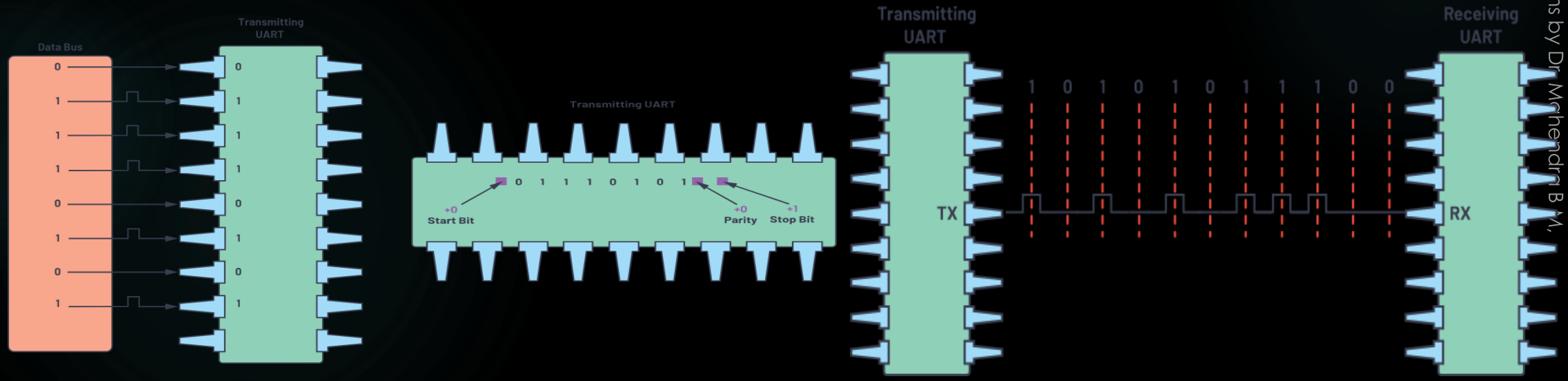
- Each data word transmitted via UART consists of various components. It begins with a start bit, which indicates the start of the data frame.
- The data bits follow the start bit and represent the actual information being transmitted.
- An optional parity bit can be included for error detection, enabling the receiver to identify and correct transmission errors.
- Finally, one or two stop bits mark the end of the data frame, providing the necessary time gap before the transmission of the next frame.

Basic Principles

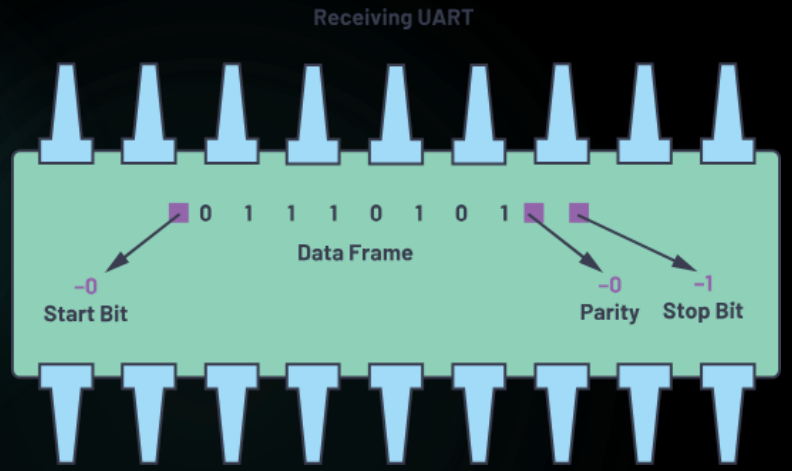
First: The transmitting UART receives data in parallel from the data bus.

Second: The transmitting UART adds the start bit, parity bit, and the stop bit(s) to the data frame.

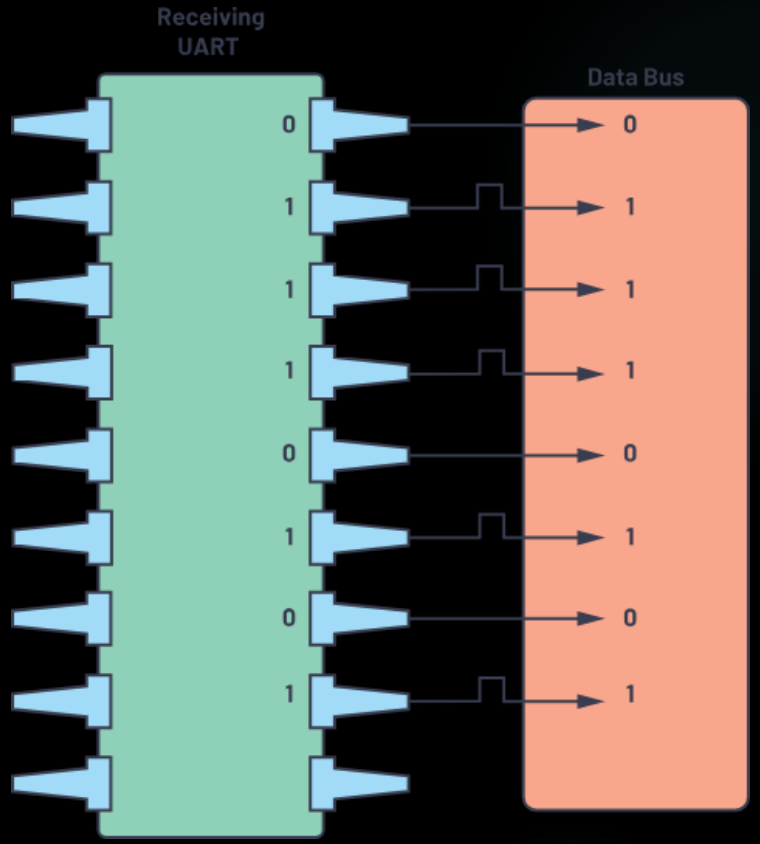
Third: The entire packet is sent serially starting from start bit to stop bit from the transmitting UART to the receiving UART. The receiving UART samples the data line at the preconfigured baud rate



Fourth: The receiving UART discards the start bit, parity bit, and stop bit from the data frame.



Fifth: The receiving UART converts the serial data back into parallel and transfers it to the data bus on the receiving end.



Advantages and Disadvantages of UART

Advantages

- Simple and Easy Implementation.
- Full Duplex Communication.
- Independent Operation.
- Support for Longer Data Frames.
- Point-to-Point Communication.

Disadvantages

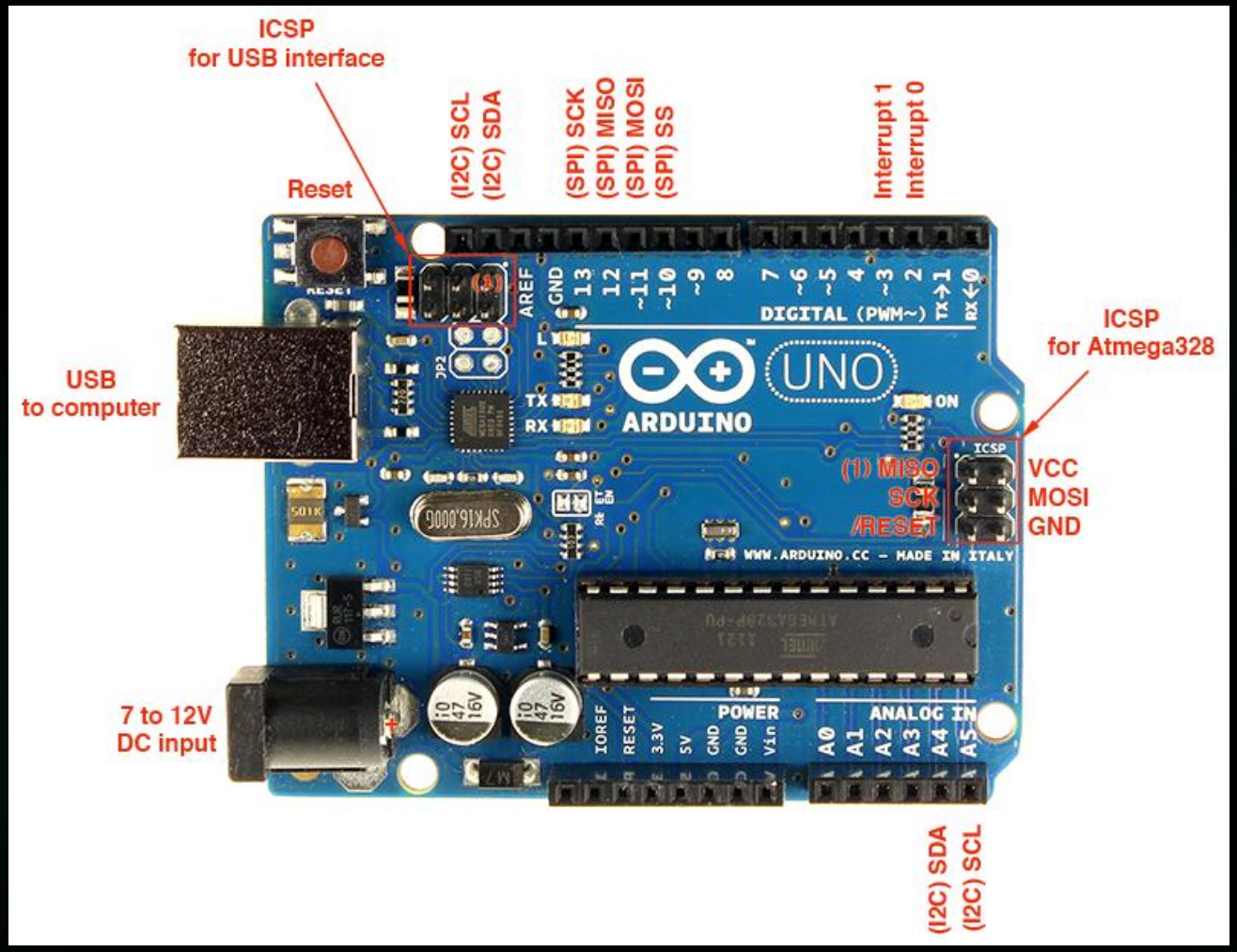
- Synchronization Requirement.
- Potential Synchronization Issues.
- Limited Error Detection.
- Lack of Addressing.
- Limited Speed.

For more information:
https://www.wevolver.com/article/spi-vs-i2c-vs-uart-in-depth-comparison

Programming Communication Protocols

SPI, I2C & UART IN ARDUINO-UNO R3

Protocol Pins in Arduino



I2C Programming with Arduino

- https://docs.arduino.cc/learn/communication/wire

Data Converters

Real World Analog Signals and Analog-to-Digital Conversion

- **Analog Signals:** Analog signals are continuous, real-world representations of physical quantities, such as temperature, voltage, pressure, etc.
- **Importance:** They are crucial in various fields like weather monitoring, healthcare, communication, and industrial processes.
- **Analog-to-Digital Conversion (ADC):** The process of converting continuous analog signals into discrete digital values for processing and analysis.

Real World Analog Signals

Temperature Signals:

- Examples: Weather monitoring (thermometers), industrial processes (thermal sensors).
- Characteristics: Continuous variation, sensitivity to environmental changes.

Biomedical Signals:

- Examples: Electrocardiogram (ECG), electroencephalogram (EEG), blood pressure.
- Clinical Importance: Diagnosis, monitoring, treatment.

Audio Signals:

- Examples: Speech, music.
- Characteristics: Continuous waveforms, complex frequencies.

Analog-to-Digital Conversion (ADC)

- **Definition:** ADC is the process of transforming analog signals into digital form suitable for processing by computers or digital systems.
- **Steps Involved:**
  - **Sampling:** Capturing discrete samples of the analog signal at regular intervals.
  - **Quantization:** Mapping each sample's amplitude to the closest digital value.
  - **Encoding:** Representing the quantized value in binary format.

Sampling

- **Sampling Process:** Selecting and measuring the amplitude of the analog signal at specific time intervals.
- **Nyquist-Shannon Sampling Theorem:** To accurately reconstruct an analog signal from its samples, the sampling frequency must be at least twice the highest frequency component of the signal (Nyquist frequency).
- **Sampling Rate and Frequency Spectrum:** Choosing an appropriate sampling rate to avoid aliasing and preserve signal details.

Quantization

- **Quantization Process:** Assigning digital values to the continuous amplitude levels of the analog signal.
- **Quantization Levels and Resolution:** More quantization levels lead to higher resolution and accuracy, but also larger data size.
- **Quantization Error and SNR:** Quantization introduces an error; Signal-to-Noise Ratio (SNR) measures the quality of quantization.

Encoding

- **Digital Codes:** Binary, Gray, etc.
- **Binary Encoding Process:** Representing each quantized value using a fixed number of binary digits (bits).
- **Digital Word Length and Dynamic Range:** Longer word lengths increase precision but require more data storage and processing.

ADC Architectures

- **Successive Approximation ADC:** Iterative process to approximate the analog value.
- **Flash ADC:** Parallel comparison of input to reference voltages.
- **Sigma-Delta ADC:** Oversampling technique for high-resolution and noise reduction.
- **Pipeline ADC:** Divides the conversion process into stages for higher speed.
- **Comparison of Architectures:** Each architecture has its trade-offs in terms of speed, resolution, and complexity.

Factors Affecting ADC Performance

- **Resolution and Accuracy:** Higher resolution yields more precise measurements.
- **Sampling Rate:** Balancing between capturing signal details and computational load.
- **Input Range and Voltage Reference:** Choosing appropriate voltage levels for accurate conversion.
- **Noise and Distortion:** Minimizing noise and distortion for accurate representation.

Applications of ADC

- **Medical Imaging:** CT scans, MRI, ultrasound.
- **Communication Systems:** Voice and data transmission.
- **Industrial Automation:** Process control, monitoring.
- **Consumer Electronics:** Cameras, touchscreens, audio devices.

Challenges and Future Trends

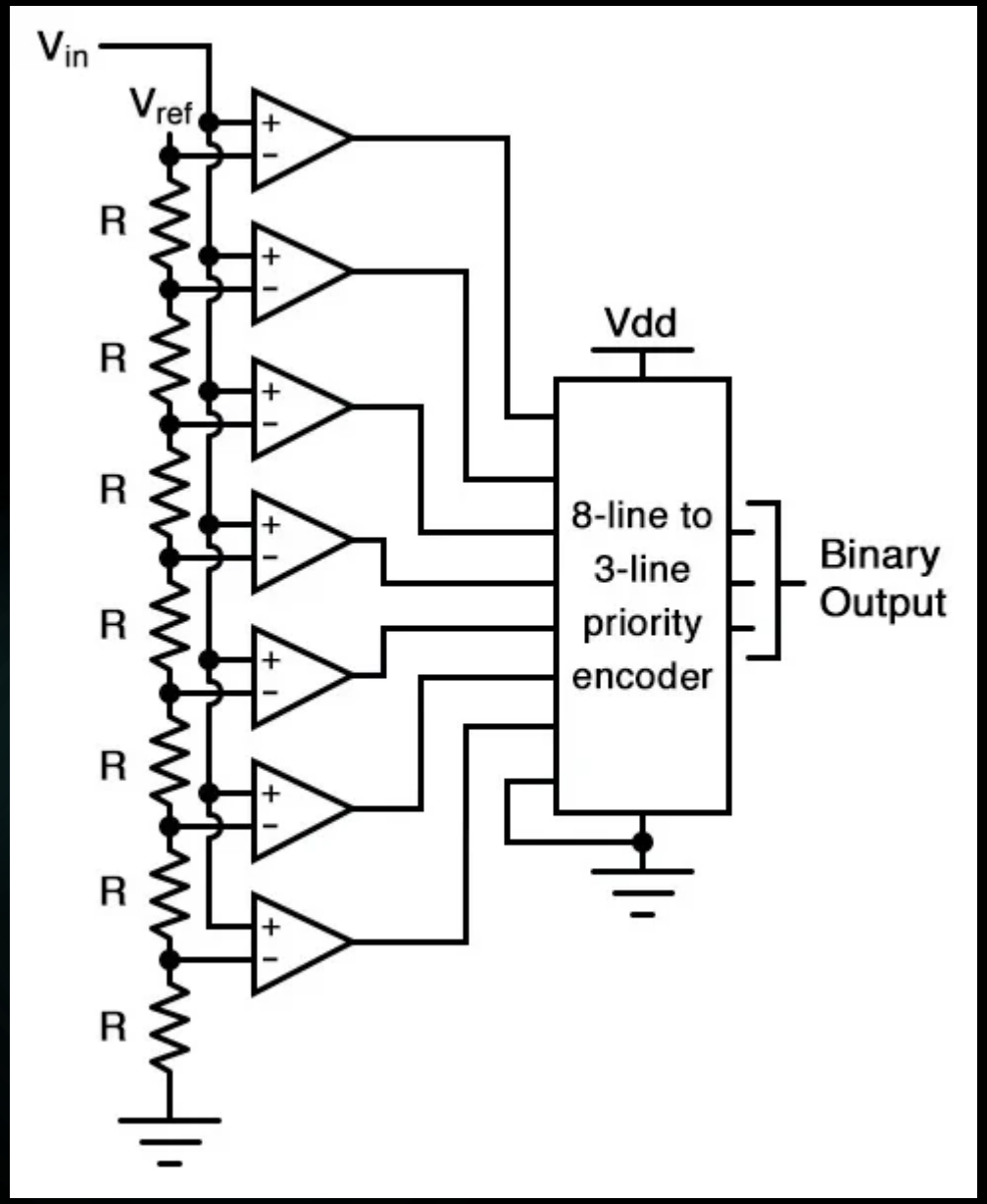
- Increasing Signal Complexity: Handling complex signals with higher accuracy requirements.
- Ultra-High-Resolution ADCs: Pushing the boundaries of resolution for precise measurements.
- Energy Efficiency in ADC Design: Optimizing power consumption for portable devices and IoT.
- Integration with IoT and Edge Computing: Enabling real-time data processing at the edge of networks.

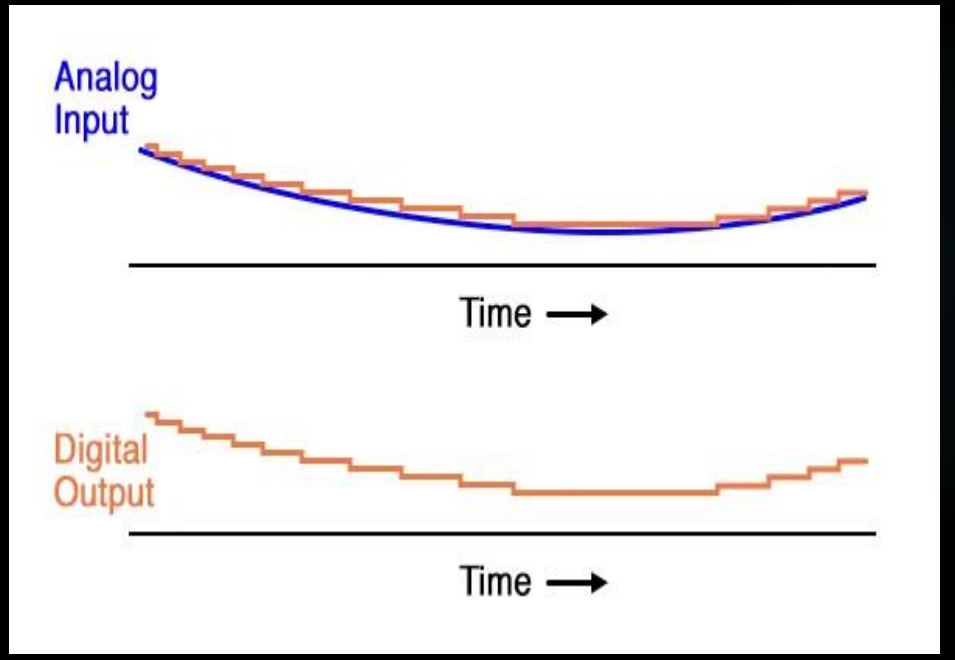
Conclusion

- Importance: Real world analog signals are vital sources of information in various domains.
- ADC Role: Analog-to-digital conversion is a fundamental step for processing and utilizing analog data in digital systems.
- Future Prospects: Continued advancements in ADC technology will drive innovations across industries.

FLASH Type ADC

- **Flash ADC:** High-speed and parallel ADC architecture.
- **Operating Principle:** Uses a set of comparators to quickly determine the digital output.
- It is formed of a series of comparators, each one comparing the input signal to a unique reference voltage.
- The comparator outputs connect to the inputs of a priority encoder circuit, which then produces a binary output.


Vin
Vref
R
R
R
R
R
R
R
Vdd
8-line to
3-line
priority
encoder
Binary
Output

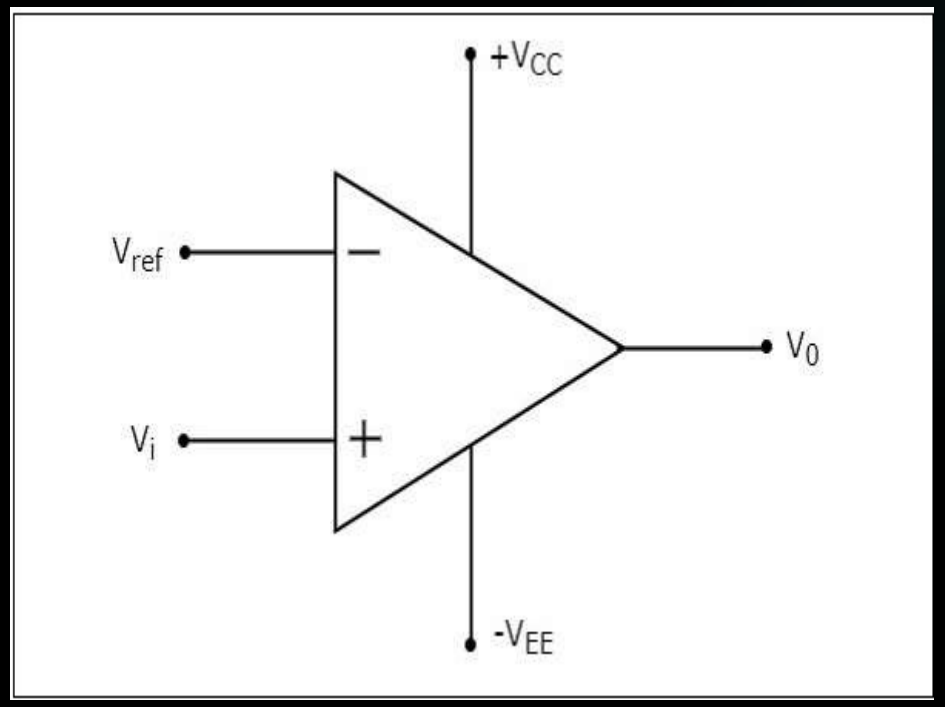

Analog
Input
Time
Digital
Output
Time

Working

- **Vref** is a stable reference voltage provided by a precision voltage regulator as part of the converter circuit, not shown in the schematic.
- As the analog input voltage exceeds the reference voltage at each comparator, the comparator outputs will sequentially saturate to a high state.
- The priority encoder generates a binary number based on the highest-order active input, ignoring all other active inputs.

Non-Inverting Comparator

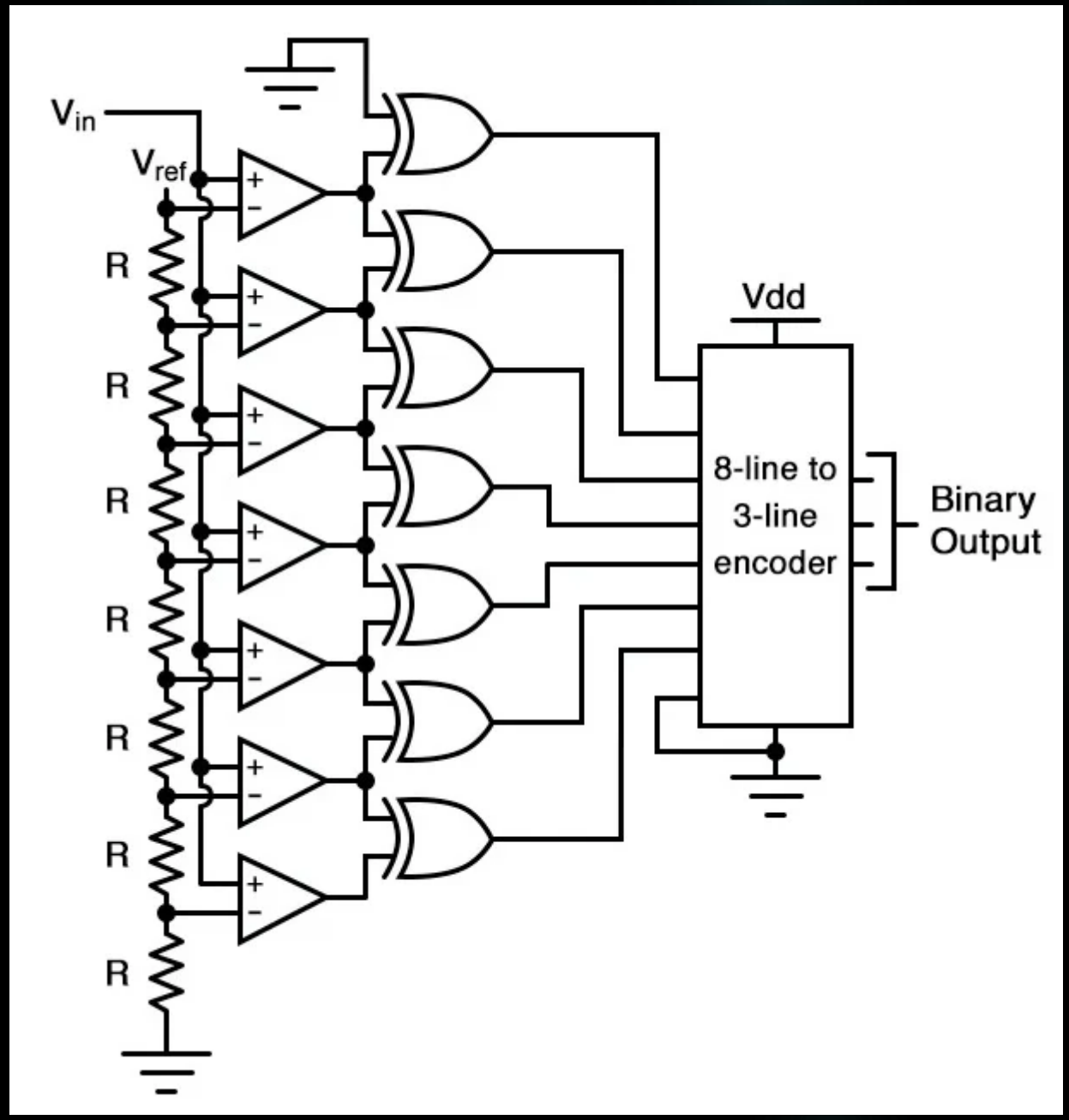
- A non-inverting comparator is an op-amp based comparator for which a reference voltage is applied to its inverting terminal and the input voltage is applied to its non-inverting terminal.
- This op-amp based comparator is called as non-inverting comparator because the input voltage, which has to be compared is applied to the non-inverting terminal of the op-amp.

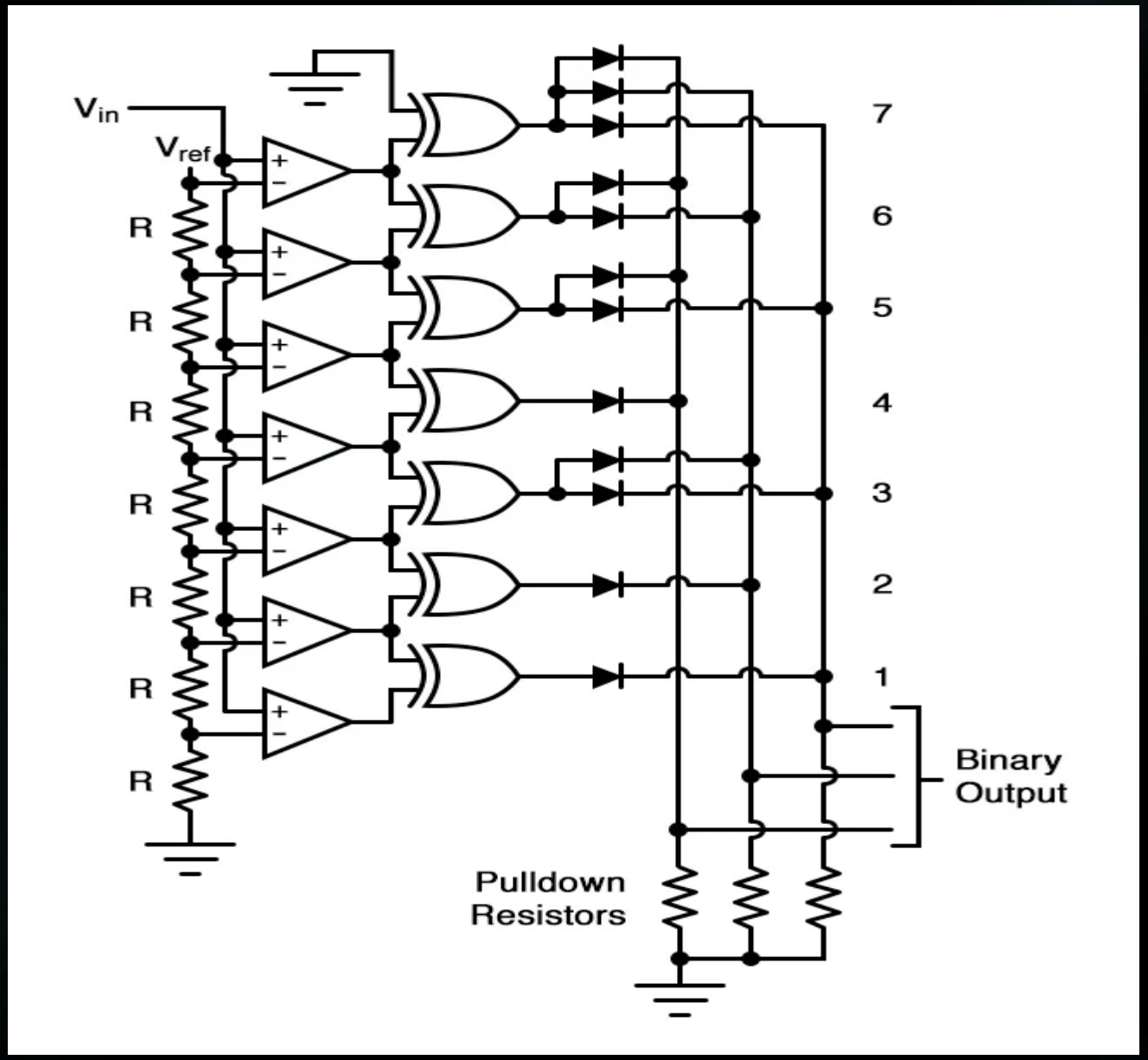


Operation

- The output value of a non-inverting comparator will be +Vsat, for which the input voltage Vi is greater than the reference voltage +Vref.
- The output value of a non-inverting comparator will bee −Vsat, for which the input voltage Vi is less than the reference voltage +Vref.

- 3 Bit Analog to Digital Converter.
  - Part 1: Comparators (Sampling)
  - Part 2: Assign Digital Values to each voltage level (Quantization)
  - Part 3: Assign fixed digital value to each level (Encoding)



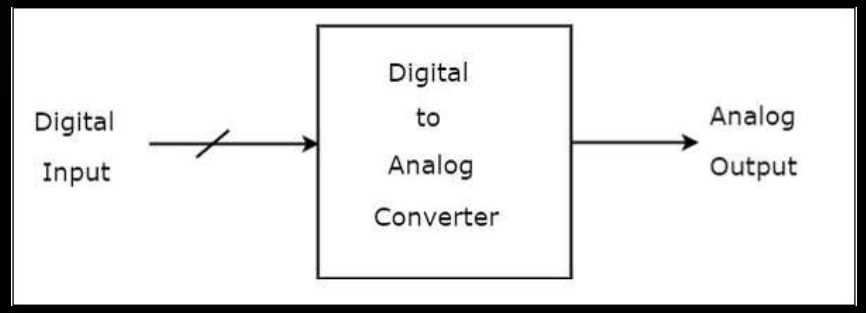

V_{in}
V_{ref}
R
R
R
R
R
R
R
7
6
5
4
3
2
1
Binary Output
Pulldown Resistors

- Not only is the flash converter the simplest in terms of operational theory, but it is the most efficient of the ADC technologies in terms of speed, being limited only in comparator and gate propagation delays.
- This three-bit flash ADC requires seven comparators. A four-bit version would require 15 comparators. With each additional output bit, the number of required comparators doubles.
- Considering that eight bits is generally considered the minimum necessary for any practical ADC (255 comparators needed!), the flash methodology quickly shows its weakness.
- With equal-value resistors in the reference voltage divider network, each successive binary count represents the same amount of analog signal increase, providing a proportional response.

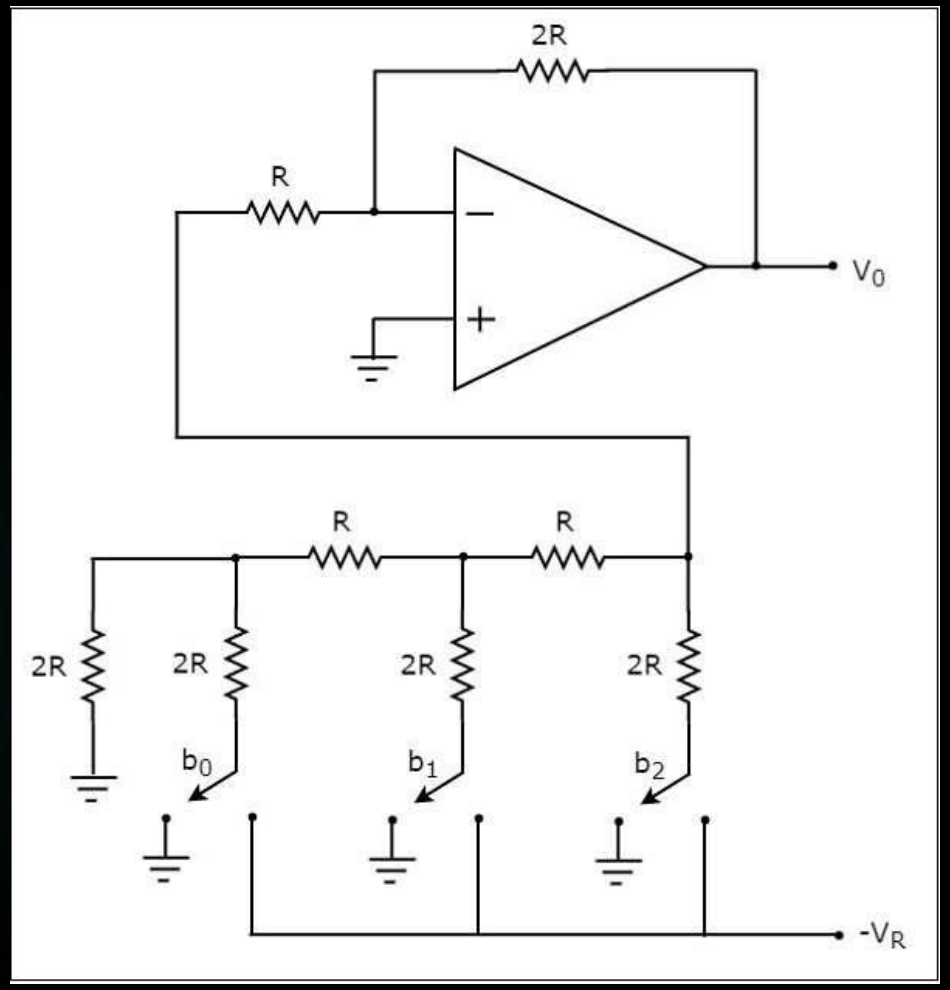
Digital to Analog Converters (DAC)

- A Digital to Analog Converter (DAC) converts a digital input signal into an analog output signal. The digital signal is represented with a binary code, which is a combination of bits 0 and 1.
- In general, the number of binary inputs of a DAC will be a power of two.
- There are two types of DACs

  a. Weighted Resistor DAC, b. R-2R Ladder DAC

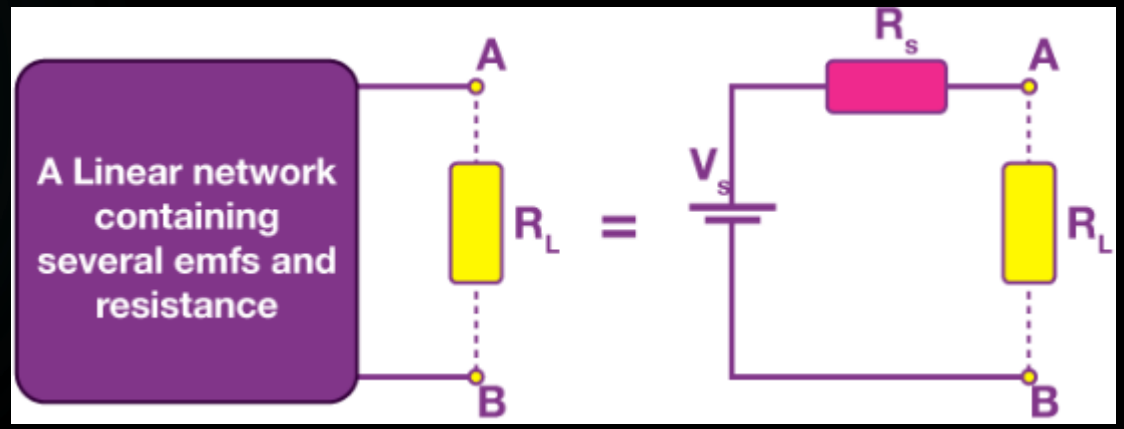


3-bit R-2R Ladder DAC

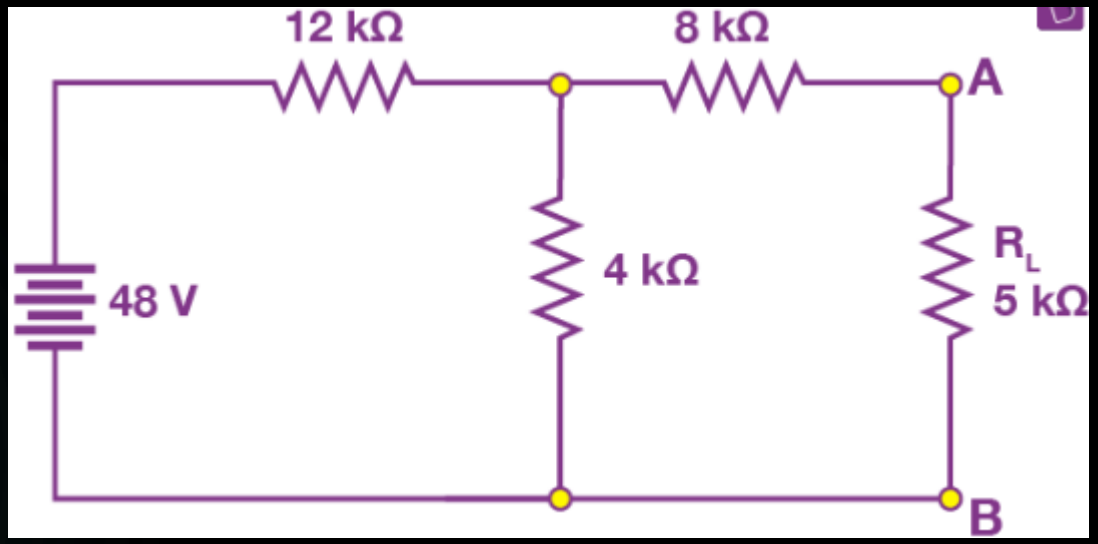


How to Analyze the R-2R Network

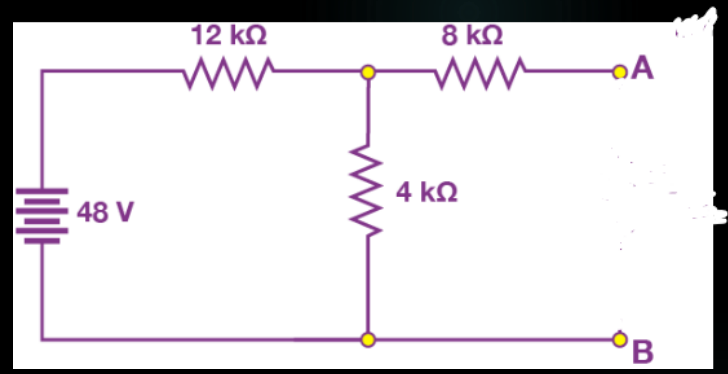
- To understand the working of R-2R Network, understanding Thevenin's theorem is very important.
- Let us understand Thevenin's theorem.
- **Statement:** Any Complex linear bilateral electrical network across any two terminals can be replaced by a single voltage and resistance network connected in series across the terminals.

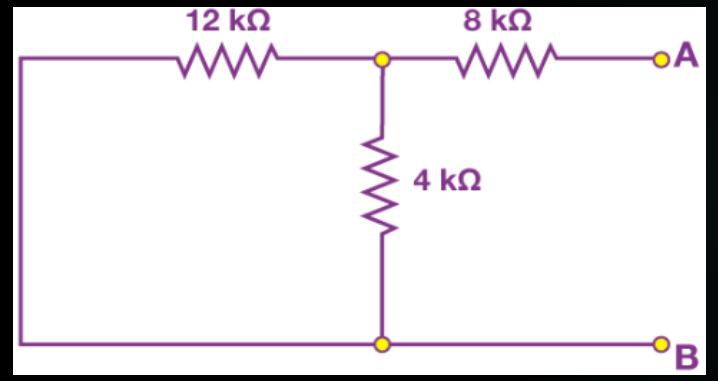


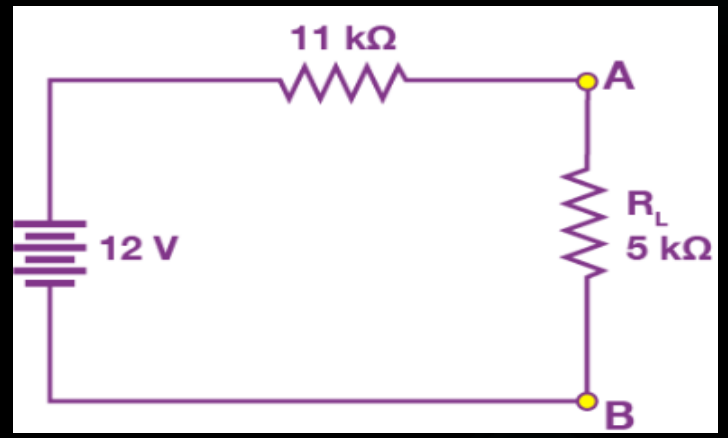
Find V_{TH} and R_{TH} across the load resistor in the circuit below using Thevenin's Theorem.



- Step 1: Remove the 5 kΩ from the circuit.
- Step 2: Measure the open-circuit voltage by applying any basic network laws.
- This will give you the Thevenin's voltage (VTH).
- Calculate the Thevenin's Resistance across the open circuit terminal by short circuiting the voltage source.
- Connect the RTH in series with Voltage Source VTH and the load resistor.

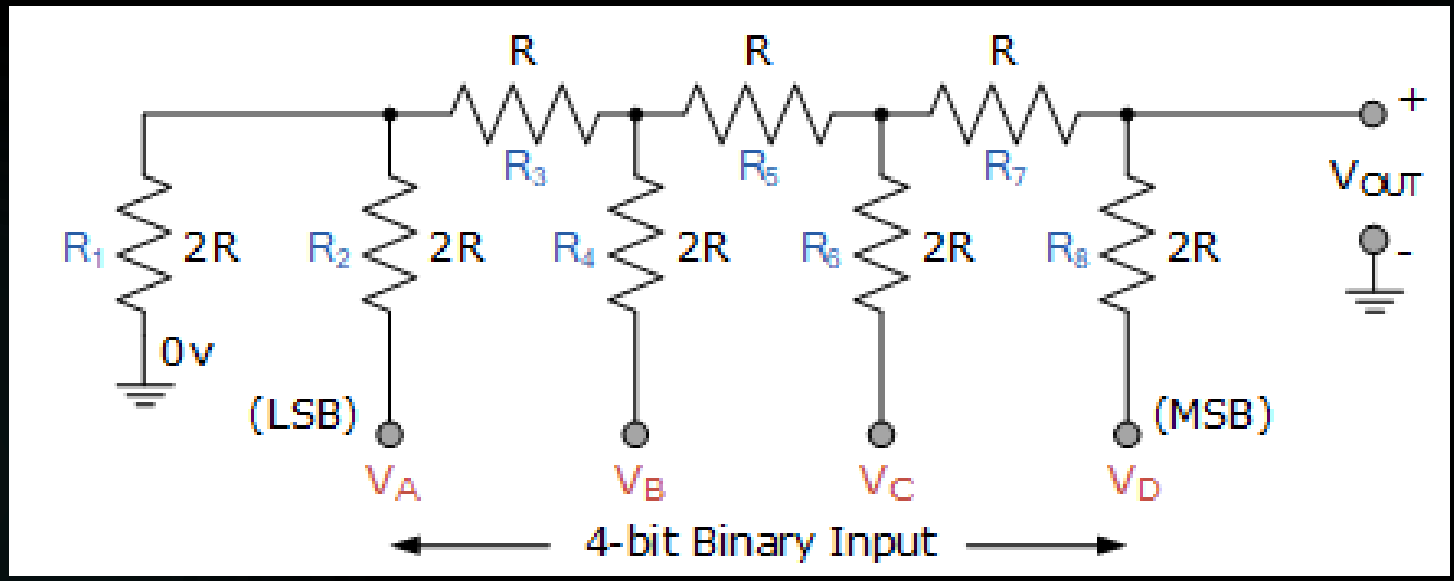


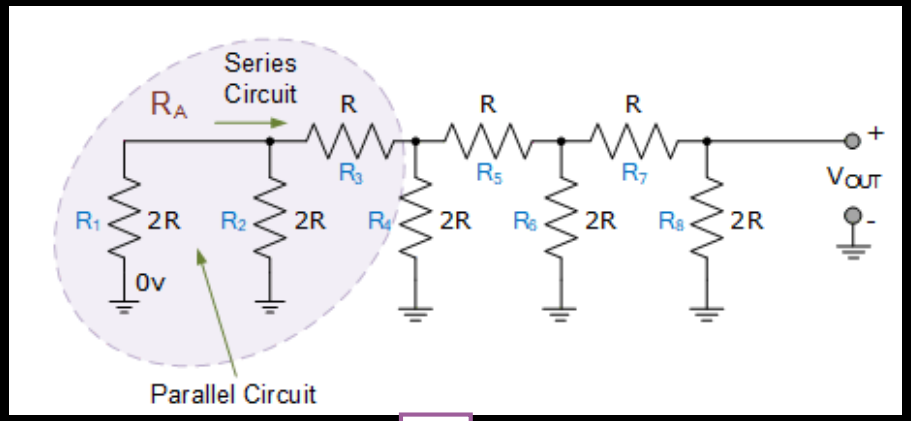




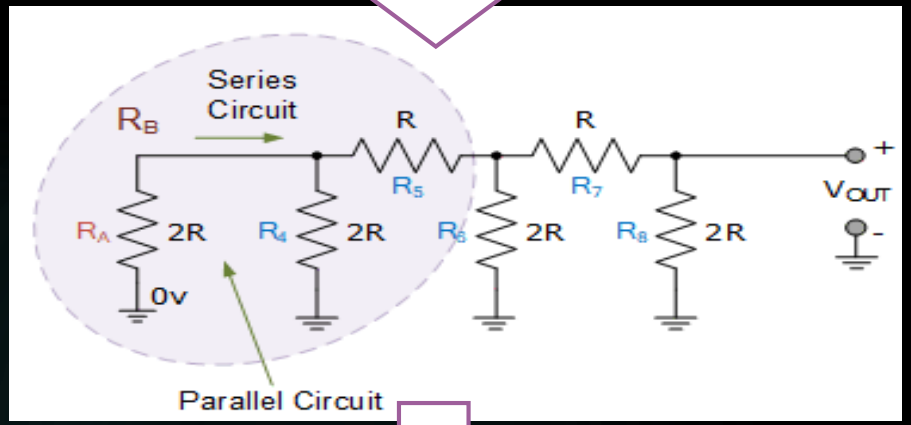
4-bit R-2R Resistive Ladder Network

R-2R DAC Circuit with Four Zero (LOW) Inputs

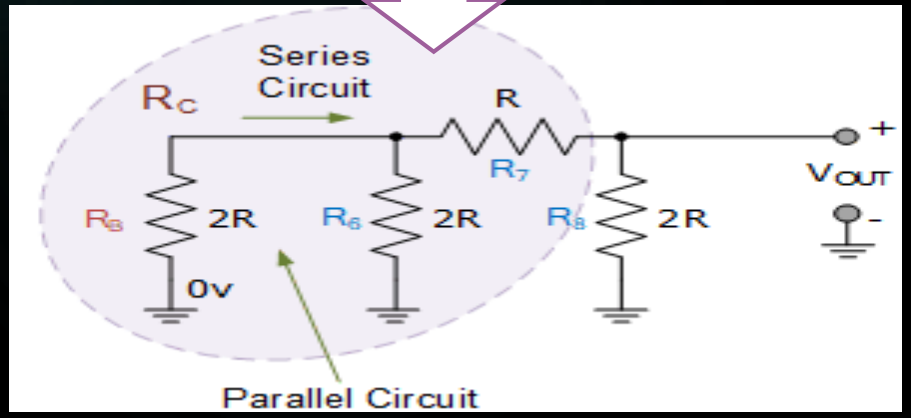




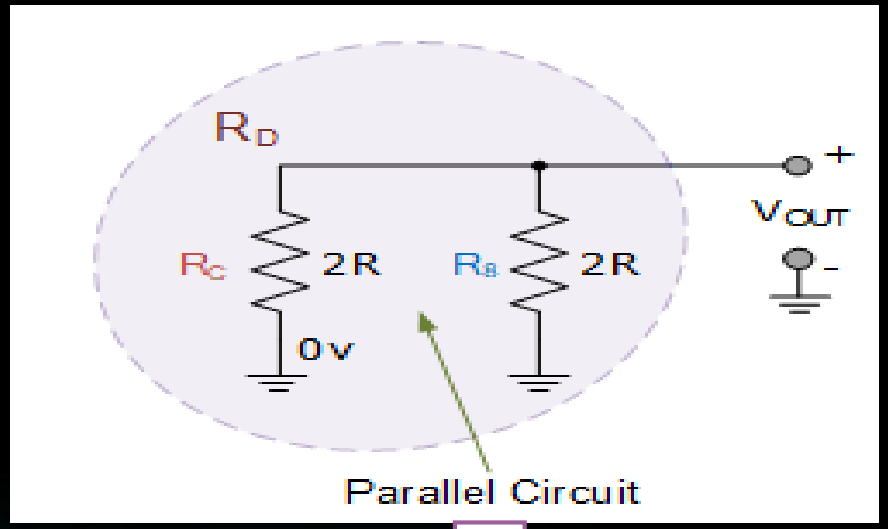
$$R_A = R_3 + \frac{R_1 \times R_2}{R_1 + R_2} = R + \frac{2R \times 2R}{2R + 2R} = R + R = 2R$$



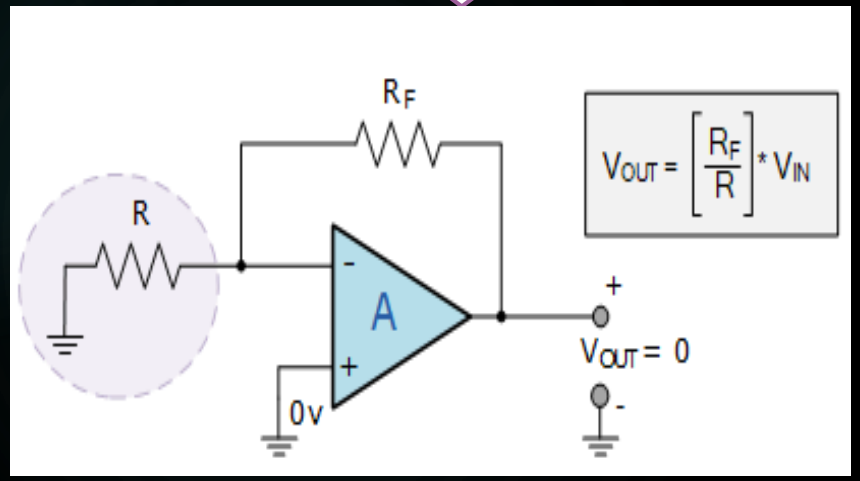
$$R_B = R_5 + \frac{R_A \times R_4}{R_A + R_4} = R + \frac{2R \times 2R}{2R + 2R} = R + R = 2R$$



$$R_C = R_7 + \frac{R_B \times R_6}{R_B + R_6} = R + \frac{2R \times 2R}{2R + 2R} = R + R = 2R$$



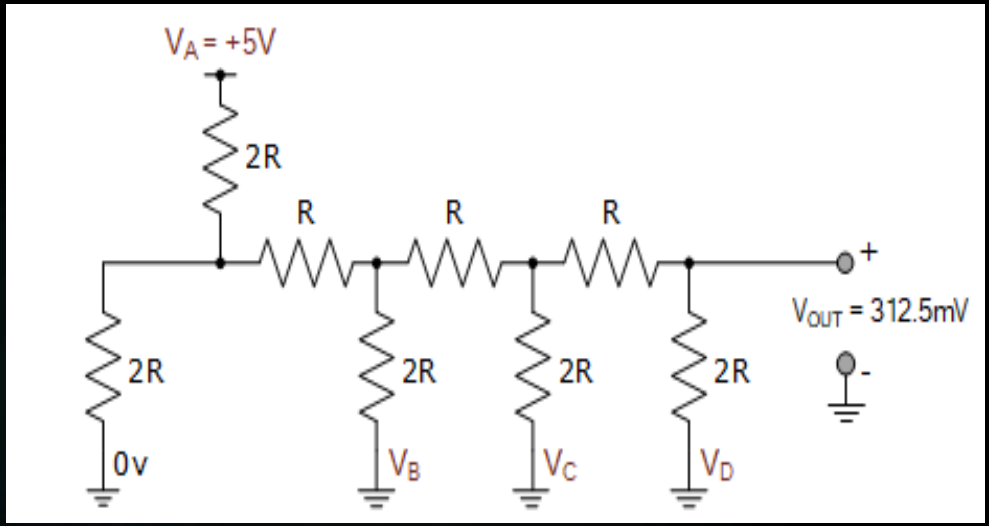
Two equal resistor values are paralled together, the resulting value is one-half, so 2R in parallel with 2R equals an equivalent resistance of R.



- The output voltage for an inverting operational amplifier is given as: (RF/RIN)*VIN.
- If we make RF equal to R, that is RF = R = 1, and as R is terminated to ground (0V), then there is no VIN voltage value, (VIN = 0) so the output voltage would be: (1/1)*0 = 0 volts.
- So for a 4-bit R-2R DAC with four grounded inputs (LOW), the output voltage will be “zero” volts, thus a 4-bit digital input of 0000 produces an analogue output of 0 volts.

So what happens now if we connect input bit V_A HIGH to +5 volts. What would be the equivalent resistive value of the R-2R ladder network and the output voltage from the op-amp.

R-2R DAC with Input V_A



- Input VA is HIGH and logic level "1" and all the other inputs grounded at logic level "0".
- As the R/2R ladder network is a linear circuit we can find Thevenin's equivalent resistance using the same parallel and series resistance calculations as above to calculate the expected output voltage. The output voltage, VOUT is therefore calculated at 312.5 milli-volts (312.5 mV).

As we have a 4-bit R-2R resistive ladder network, this 312.5 mV voltage change is one-sixteenth the value of the +5V input (5/0.3125 = 16) voltage so is classed as the Least Significant Bit, (LSB).

Continue the same for giving weights to each individual bits and find the final equation for 4 Bit DAC. Also write the generalized equation for n-bit DAC

Digital-to-Analogue Output Voltage Equation

$$V_{OUT} = \frac{V_A + 2V_B + 4V_C + 8V_D}{16}$$

Generalised R-2R DAC Equation

$$V_{OUT} = \frac{V_A + 2V_B + 4V_C + 8V_D + 16V_E + 32V_F + \ldots \text{etc}}{2^n}$$

DIY: Example on 4 bit R-2R DAC:

A 4-bit R-2R digital-to-analogue converter is constructed to control the speed of a small DC motor using the output from a digital logic circuit. If the logic circuit uses 10 volt CMOS devices, calculate the analogue output voltage from the DAC when the input code is hexadecimal number "B". Also determine the resolution of the DAC.

For more information:
https://www.electronics-tutorials.ws/combination/r-2r-dac.html