Module 2:

I/O Interfacing Techniques

BCSE305L

Dr. Nitish Katal

Outline

- *Memory interfacing.*
- *A/D, D/A.*
- *Timers.*
- *Watch-dog timer.*
- *Counters.*
- *Encoder & Decoder.*
- *UART.*
- *Sensors and actuators interfacing.*

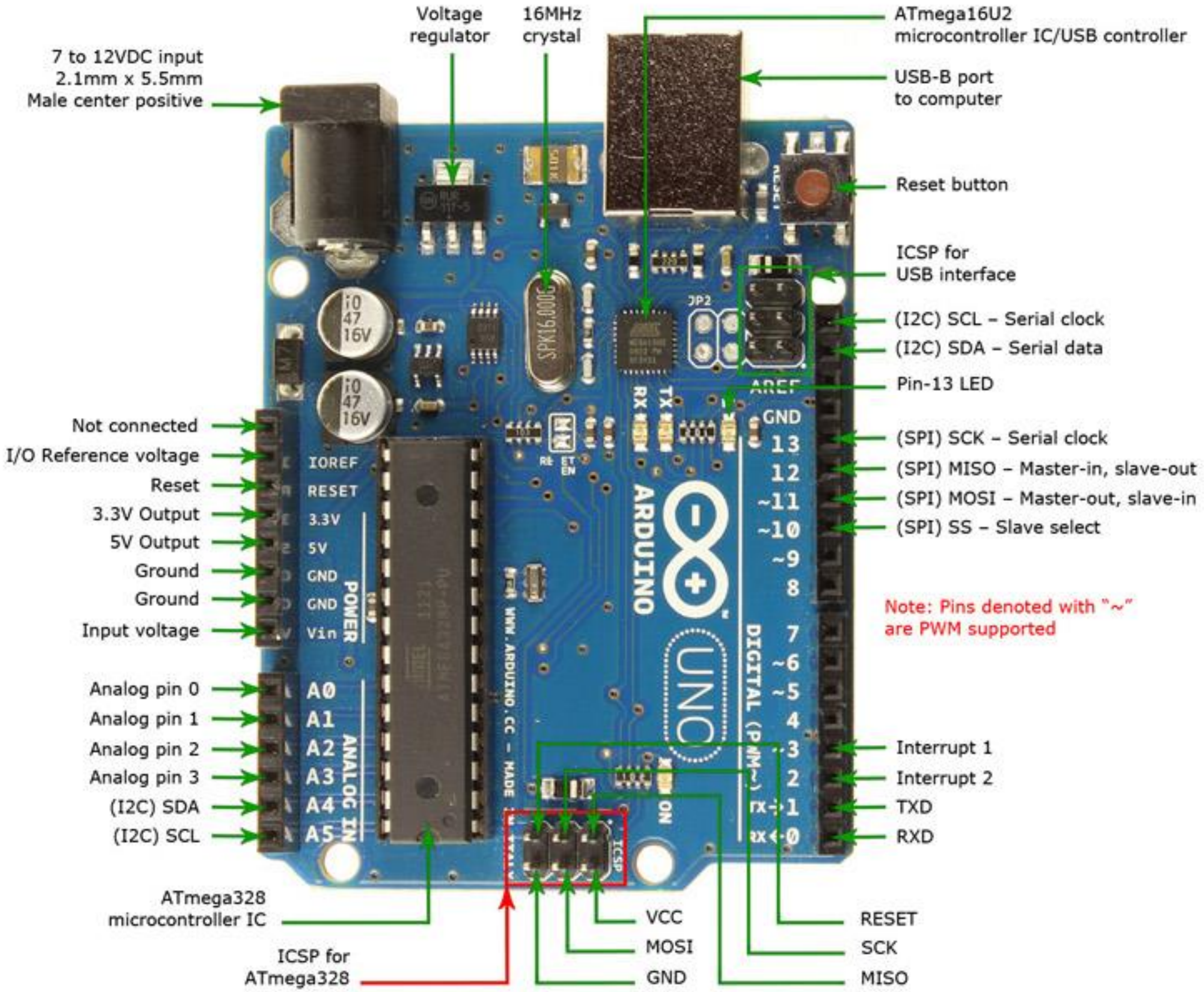
Introduction on **Arduino Uno**

Arduino Uno: Features

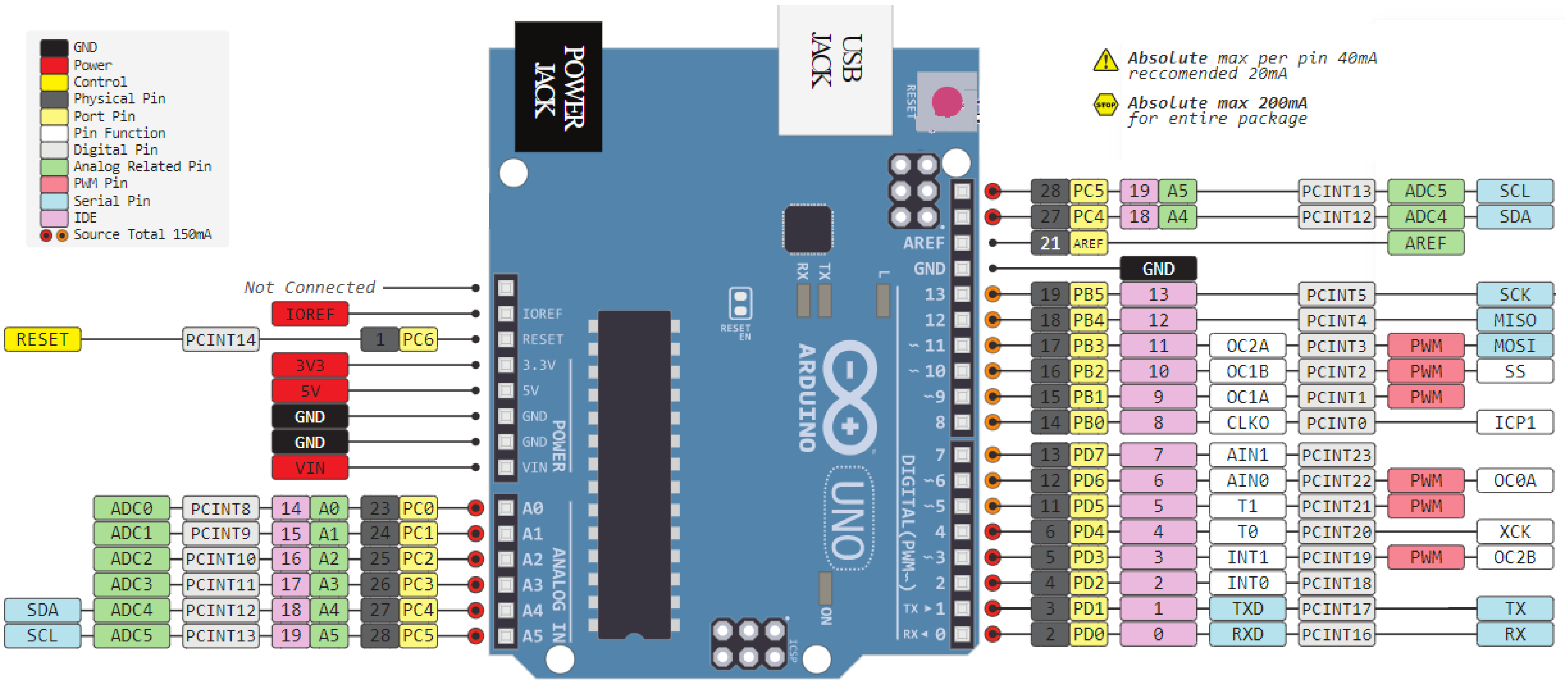
An open-source microcontroller board

- Microcontroller : ATmega328
- Operating Voltage : 5V
- Input Voltage (external) : 6-20V (7-12 recommended)
- Digital I/O Pins : 14 (6 PWM output)
- Analog Input Pins : 6
- DC Current per I/O Pin : 40 mA
- DC Current for 3.3V Pin : 50 mA
- Flash Memory : 32 KB (0.5 KB for bootloader)
- Clock Frequency : 16 MHz
- Programming Interface : USB (ICSP)

Arduino Uno: **Board**



Arduino Uno: Pins



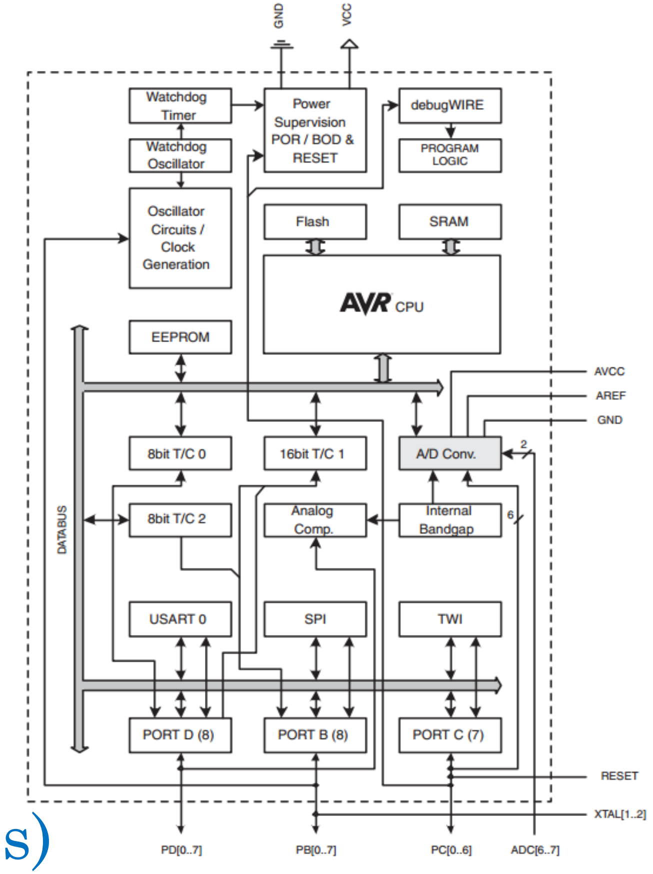
6

Arduino Uno: µC

ATMega328 : Features

- Instruction set Architecture : RISC
- Data bus size : 8-bit
- Address bus size : 16-bit
- Program memory (ROM) : 32 KB
- Data memory (RAM) : 2 KB
- Data memory (EEPROM) : 1 KB
- Input/output ports : 3 (B, C, D)
- General purpose registers : 32
- Timers : 3 (Timer0, 1 & 2)
- Interrupts : 26 (2 Ext. interrupts)
- ADC modules (10-bit) : 6 channels
- PWM modules : 6
- Analog comparators : 1
- Serial communication : SPI, USART, TWI(I2C)

Architecture



Programming
Inputs & Outputs

Programming: Inputs & Outputs

- ***Information exchange*** is done by ***input*** and ***output*** devices
- It may be ***digital*** or ***analog*** signal
- **Digital signals** will be directly interfaced to CPU
- **Analog devices** requires some converter for interfacing with CPU
- All peripherals can not be always on-chip

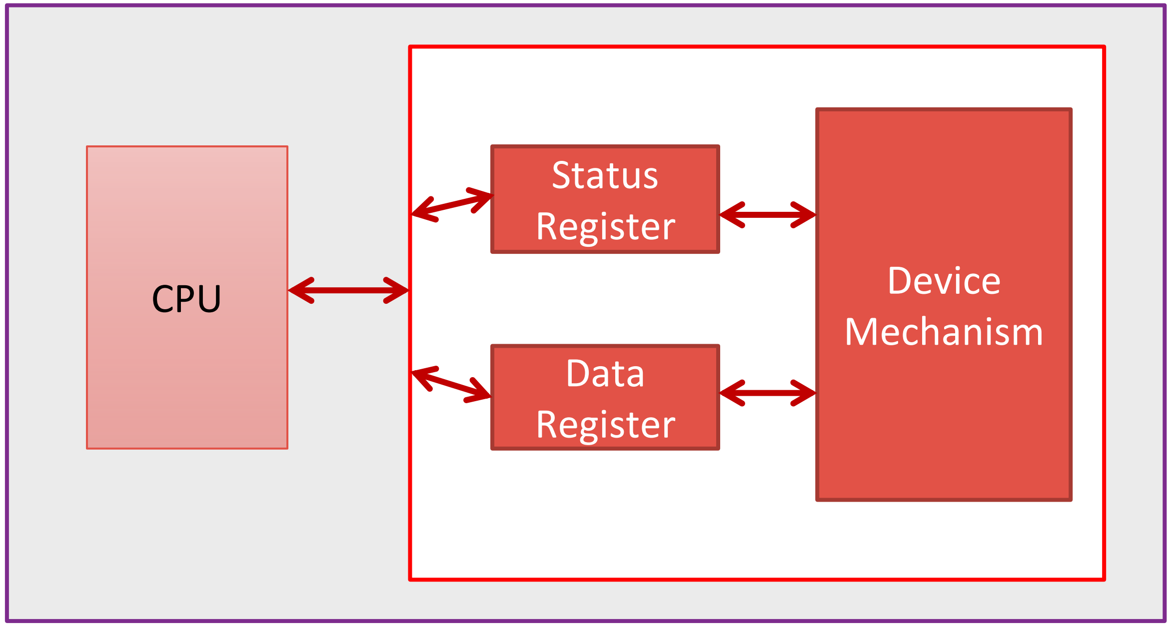
Programming: Inputs & Outputs

Types

- *Internal I/O devices*
  - Physically silicon devices
  - Small size
  - Ex: RTC, ADCs, memory
- *External I/O devices*
  - Have large size
  - Has bus interface
  - Ex: LCD, keypad, MIC

Programming: Inputs & Outputs

- **CPU communicates** to the device by **reading** and **writing** to the **registers.**
  - **Data register – read and write,**
  - **Status register – read only**



Programming: Inputs & Outputs

- **Microcontroller programming input and output**
  - I/O mapped I/O
  - Memory mapped I/O (widely used)

- **Memory mapped I/O –**
  - **Common address** shared by **both memory** and **I/O**
  - **Read** and **write instructions** to **communicate** with the **devices**
  - Example: In 8051 Timer Register - MOV TMOD, #01H

Programming: Inputs & Outputs

- Busy-Wait I/O
  - Checking an I/O device if it has completed the task by reading its status register often called polling.
  - Example: AGAIN: JNB TF0, AGAIN

- Interrupts
  - It enables I/O devices to signal the completion or status and force execution of a particular piece of code.
  - Example: MOV IE, #82H

Programming
ADC & DACs

ADC Convertor

- **An ADC is an electronic circuit whose digital output is proportional to its analog input.**
- **Effectively it "measures" the input voltage and gives a binary output number proportional to its size.**
- **The list of possible analog input signals is endless, (e.g. audio and video, medical or climatic variables)**
- ***ADC when operates within a larger environment*, often called a data acquisition system.**

ADC Convertor

- **ADC Conversion**

$$D = \frac{V_i}{V_r} \times 2^n$$

- ***Where,***
  - *Vi* is the input voltage
  - *Vr* the reference voltage
  - *n* the number of bits in the converter output
  - *D* the digital output value.
- **The output binary number D is an** integer
- **For an *n*-bit number can take any value from 0 to (2^n - 1).**

ADC Convertor

- **Resolution**

$$\textit{Resolution} = \frac{V_r}{2^n}$$

- **ADC** has ***maximum*** and ***minimum permissible input values***.
- ***Resolution is a measure of how precisely an ADC can convert and represent a given input voltage.***
- ***Example:***
  - Arduino ADC is 10-bit.
  - This leads to a resolution of 5/1024, or 4.88 mV.

ADC Convertor

- ***Not all*** the ***pins*** on a ***microcontroller*** have the ***ability*** to do ***analog*** to ***digital conversions***.

- **In Arduino board**, analog pins have an **'A' in front of their label (A0 through A5)**.

- A **10-bit device** a total of **1024** different values.

- An **input of 0 volts** would result in a **decimal 0**.

- An **input of 5 volts** would give the **maximum of 1023**.

ADC Convertor

- ***Example:***
  - *ADC Value: 434*

$$434 = \frac{V_i}{5} \times 1024$$

$$V_i = 2.12 \text{ V}$$

ADC Convertor

- Analog I/O
  - AREF pin and analogReference(type) ADC reference other than 5 V.
  - Increase the resolution available to analog inputs
    - That operate at some other range of lower voltages below +5
    - Must be declared before analogRead()

```
analogReference(type)

Parameters type:
which type of reference to use
(DEFAULT, INTERNAL, INTERNAL1V1,
INTERNAL2V56, or EXTERNAL)
Returns
None
```

ADC Convertor

Analog I/O

analogRead(pin)

- ***Reads*** the ***value*** from a ***specified analog pin*** with a ***10-bit resolution***.
- Works for analog pins (0–5).
- It will return a value between 0 to 1023.
- 100 microseconds for one reading.
- Reading rate 10000 per sec.
- INPUT nor OUTPUT need not be declared.

```
analogRead(pin)

Parameters pin:
the number of the analog input
pin to read from (0-5)

Returns
 int(0 to 1023)
```

ADC Convertor

Analog I/O

analogWrite(pin,value)

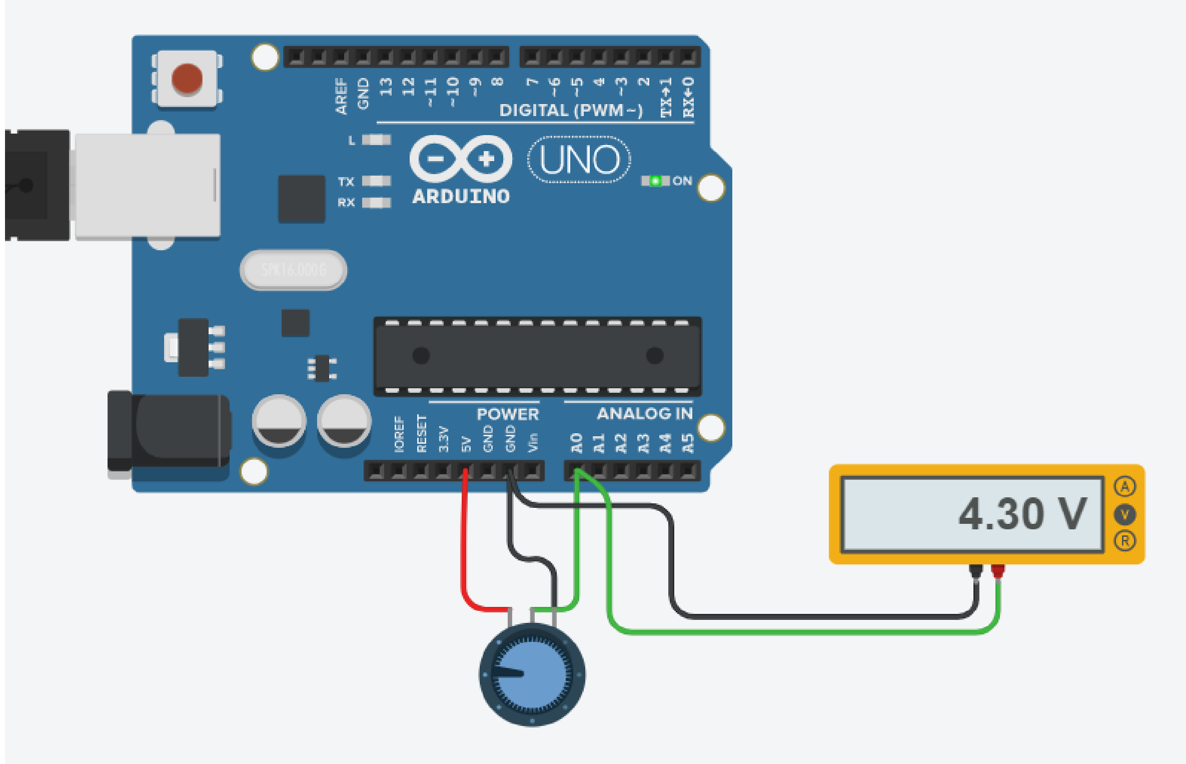
- Write an analog output on a digital pin.
- This is called *Pulse Width Modulation*.
- PWM uses digital signals to control analog devices.
- *Digital Pins # 3, # 5, #6, # 9, # 10, and # 11 can be used as PWM pins.*

```
analogWrite(pin,value)

Parameters pin: the number of
the pin you want to write
value: the duty cycle between 0
(always off, 0%) and 255
(always on, 100%)

Returns
None
```

ADC Convertor

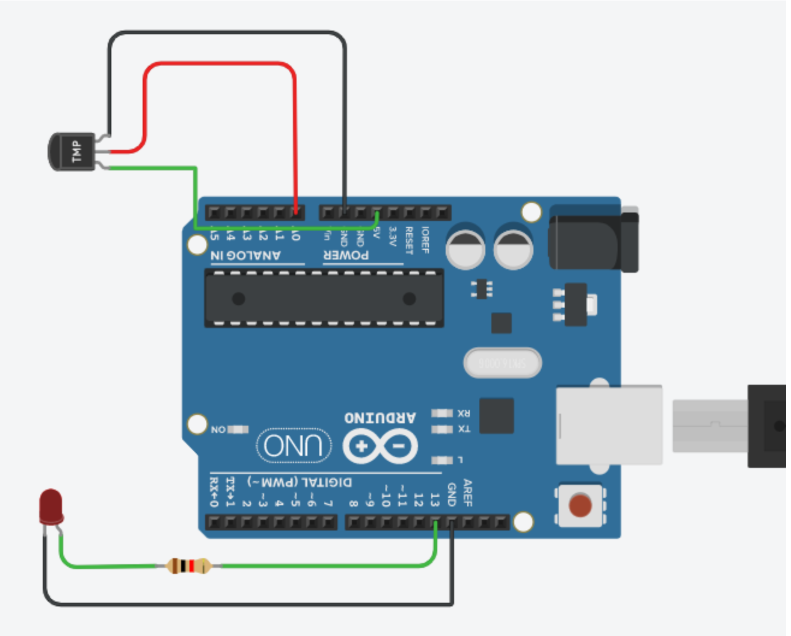
Program an Arduino board to read analog value from an analog input pin, convert the value into digital value and display it in a serial window.

```
int sensorValue = 0;

void setup()
{
        //monitor output in serial port
Serial.begin(9600);
        //intitating serial communication with
            9600 baud rate
}
void loop()
{
  sensorValue = analogRead(A0);
        //analog input is read from A0 pin
  Serial.println(sensorValue);
        //printing the output in serial port
  delay(100);  // simple delay of 100ms
}
```



ADC Convertor

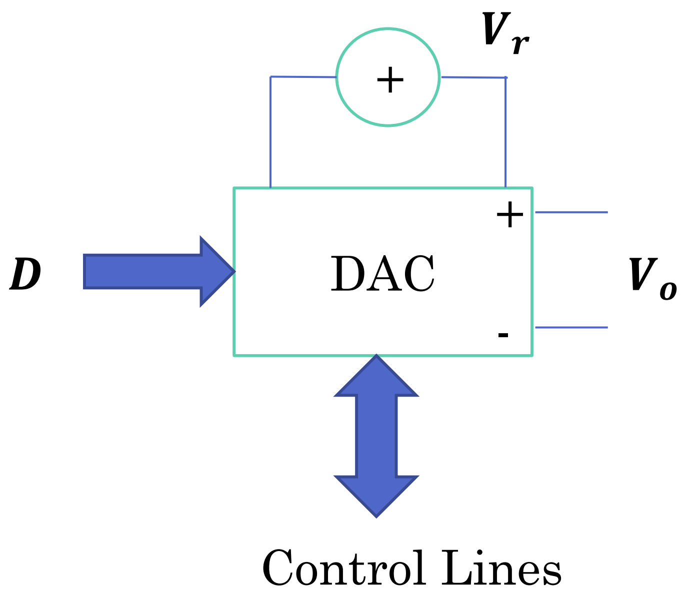
Code an Arduino board to read a temperature sensor connected to an analog input A0 and display the temperature value in serial window. Each time the value goes beyond 40 degree Celsius an LED connected to a digital output glows as warning.



```
const int lm35_pin = A0; /* LM35 O/P pin */
const int ledPin = 13;
void setup() {
  Serial.begin(9600);
  pinMode(ledPin,OUTPUT);
}
void loop() {
  int temp_adc_val;
  float temp_val;
  temp_adc_val = analogRead(lm35_pin);   /* Read Temperature */
  temp_val = (temp_adc_val * 4.88);
  /* Convert adc value to equivalent voltage */
  temp_val = (temp_val/10);  /* LM35 gives output of 10mv/°C */
  if (temp_val > 40) {
        digitalWrite(ledPin, HIGH); // turn LED on:
  } else {
        digitalWrite(ledPin, LOW); // turn LED off:
  }
  Serial.print("Temperature = ");
  Serial.print(temp_val);
  Serial.print(" Degree Celsius\n");
  delay(1000);
}
```

DAC Convertor

- DAC converter is a circuit which converts a binary input number into an analog output.
- DAC has a digital input, represented by *D*, and an analog output, represented by *Vo*.



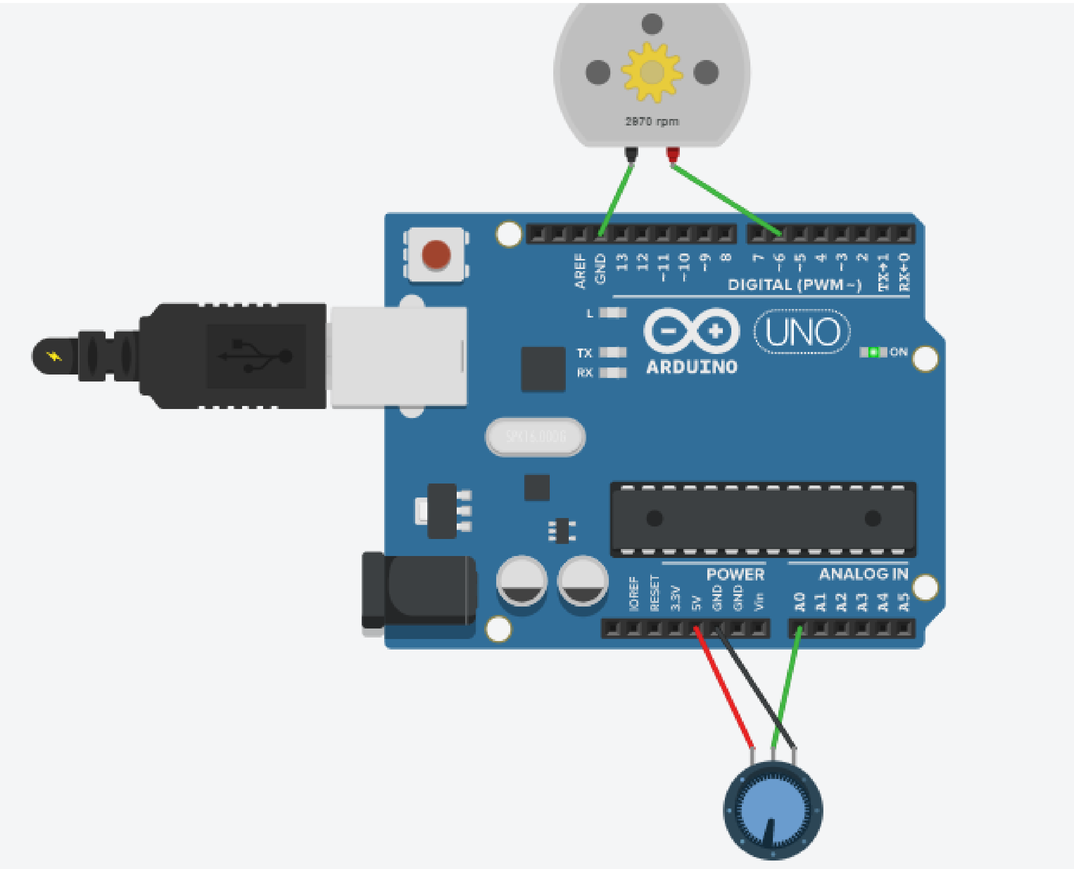
$$V_o = \frac{D}{2^n} \times V_r$$

DAC Convertor

- For each input digital value, there is a corresponding analog output.
- The number of possible output values is given by 2^n.
- The step size by $V_r/2^n$ this is called the resolution.
- The maximum possible output value occurs when $D = (2^n-1)$, so the value of V_r as an output is never quite reached.

DAC Convertor

- ***Code an Arduino board to control the speed of a DC motor using potentiometer.***



DAC Convertor

- ***Code an Arduino board to control the speed of a DC motor using potentiometer.***

```
int potvalue;
void setup()
{
    Serial.begin(9600);
    pinMode(A0, INPUT);
    pinMode(6, OUTPUT);
}

void loop()
{
    potvalue=analogRead(A0)/4;
    # scale the 10-bit value down to an 8-bit value
    analogWrite (6, potvalue);
}
```

Programming
Timers & Counters

Timer / Counter

- Digital timer/counters are used throughout embedded designs
  - To provide a ***series of time*** or ***count related events*** within the system with the ***minimum*** of ***processor*** and ***software overhead***

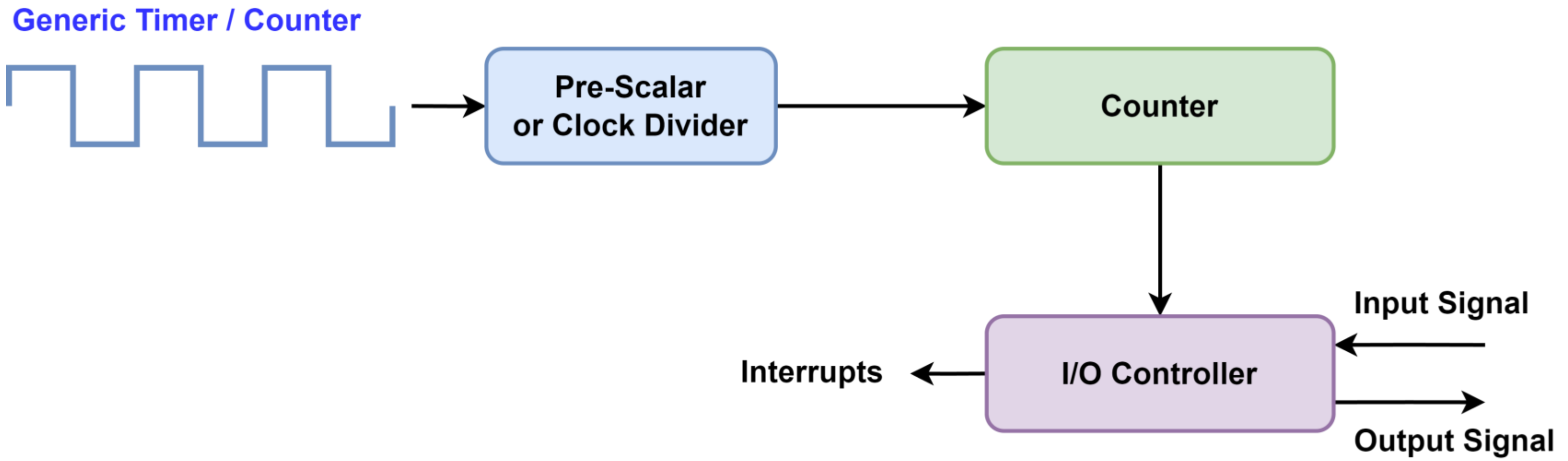
- Most embedded systems have a ***time component within them*** such as
  - ***Timing references*** for control sequences
  - To provide ***system ticks for operating systems*** and
  - Even the ***generation of waveforms***
  - ***Serial port baud rate generation*** and audible tones.

Timer / Counter

- **Can be differentiated by their use, not their logic.**
  - *Timer* – Periodic Signal
  - *Counter* – Aperiodic signal

Timer / Counter

- **Central timing is derived from a clock input.**
  - ***Internal*** **to the timer/counter (or)**
  - ***External*** **and thus connected via a separate pin.**
- ***Clock may be divided using a simple divider***
  - ***Pre-scalar*** - effectively ***divides the clock*** by the ***value*** that is ***written into*** the ***pre-scalar register***
  - The divided clock is then **passed** to a **counter**
  - **Count-down** operation or **count-up** operation
- **When a zero count is reached, an event occurs**
  - **such as an interrupt of an external line changing state.**
- ***The* final block *is an* I/O control *block***
  - **It generates interrupts**
  - **Can control the counter based on external signals**



Control Registers

- Timer/Counter *Control* Registers *A*: TCCR*n*A
- Timer/Counter *Control* Registers *B*: TCCR*n*B
- Timer Coun*NT*: TCNT*n*
- Output *Compare* Register *A*: OCR*n*A
- Output *Compare* Register *B*: OCR*n*B
- *n = C or T* number 0,1,2

Control Modes

- **Normal** : Counter count up to maximum value and reset to 0
- **Phase correct PWM**: symmetric with respect to system clock using PWM
- **Clear Time on Compare** (CTC): Used in interrupt generation
- **Fast PWM**: PWM goes as fast as clock but not synchronized with timing clock

Timers on the Arduino

- **Timer/Counter Control Registers (TCCRnA/B)**
  - Controls the *mode of timer*
  - Contains the *pre-scalar* values of timers i.e. 1, 8, 64, 256, 1024
- **Timer/Counter Register (TCNTn)**
  - Control the *counter value*
  - Set the *pre-loader* value

To calculate preloaded value for timer1 for time of 2 Sec:

TCNT1 = 65535 -(16x10^6 x time(s)/ Prescalar value)
TCNT1 = 65535 – (16x 10^6x 2 / 1024)
= 34285

Timers on the Arduino

- **delay(ms)**
  - Pauses the program for the amount of time.
  - Time is specified in milliseconds
  - A delay of 1000 ms equals 1 s

```
delay(ms)

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

Returns
None
```

Timers on the Arduino

- **delayMicroseconds()**
  - Used to delay for a much shorter time.
  - A time period of 1000 μs would equal 1 ms.

```
delayMicroseconds()

Parameters
us: the number of microseconds to pause (unsigned int)

Returns
None
```

Timers on the Arduino

- **micros()**
  - Returns the value in microseconds.

```
micros()

Parameters
None

Returns
Number of microseconds since the program started (unsigned long)
```

Timers on the Arduino

- **Example**
  - Code an Arduino board to read a pushbutton connected to a digital input to display millis() when ON and display micros() when OFF. Blink an LED connected to a digital output with 500 ms delay.

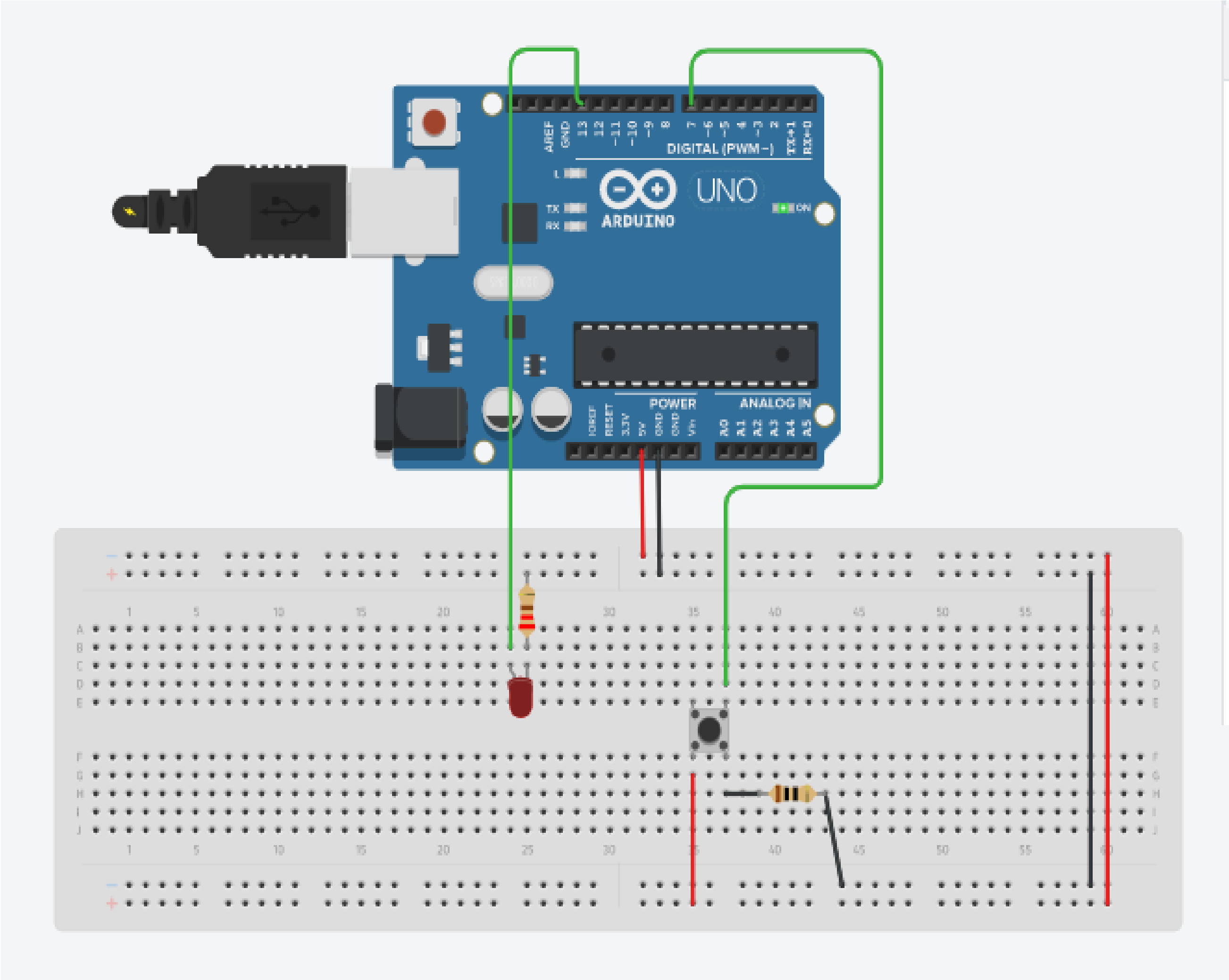
Timers on the Arduino

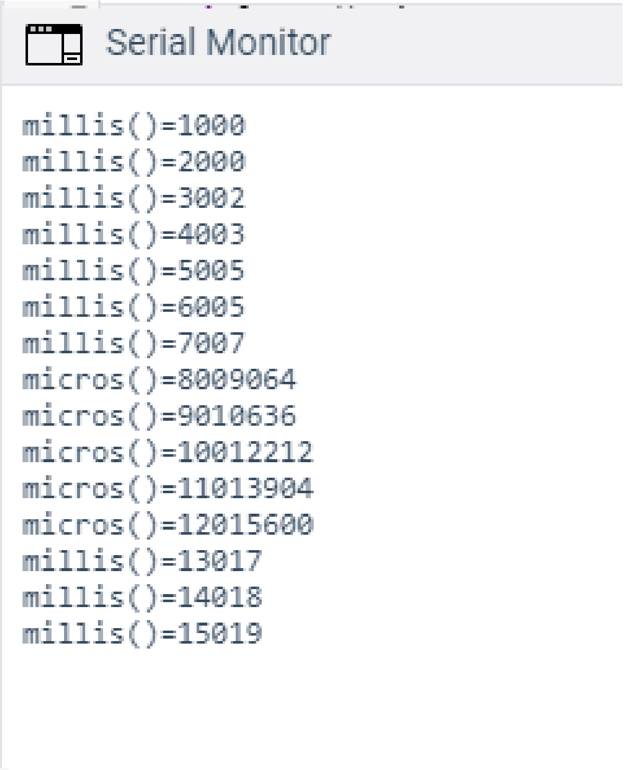
- **Example**

```
#define LED 13
int buttonState = 0;
void setup() {
    Serial.begin(9600);
    pinMode(LED, OUTPUT);
    pinMode(7, INPUT);}
void loop() {
    digitalWrite(LED, HIGH);
    delay(500);
    digitalWrite(LED, LOW);
    delay(500);
    buttonState = digitalRead(7);
    if(buttonState==LOW) {
            Serial.write("micros()=");
            Serial.println(micros());
            }
    else {
            Serial.write("millis()=");
            Serial.println(millis());
    }}
```

Timers on the Arduino

- Example





Generation of **PWM**

Pulse Width Modulation (PWM)

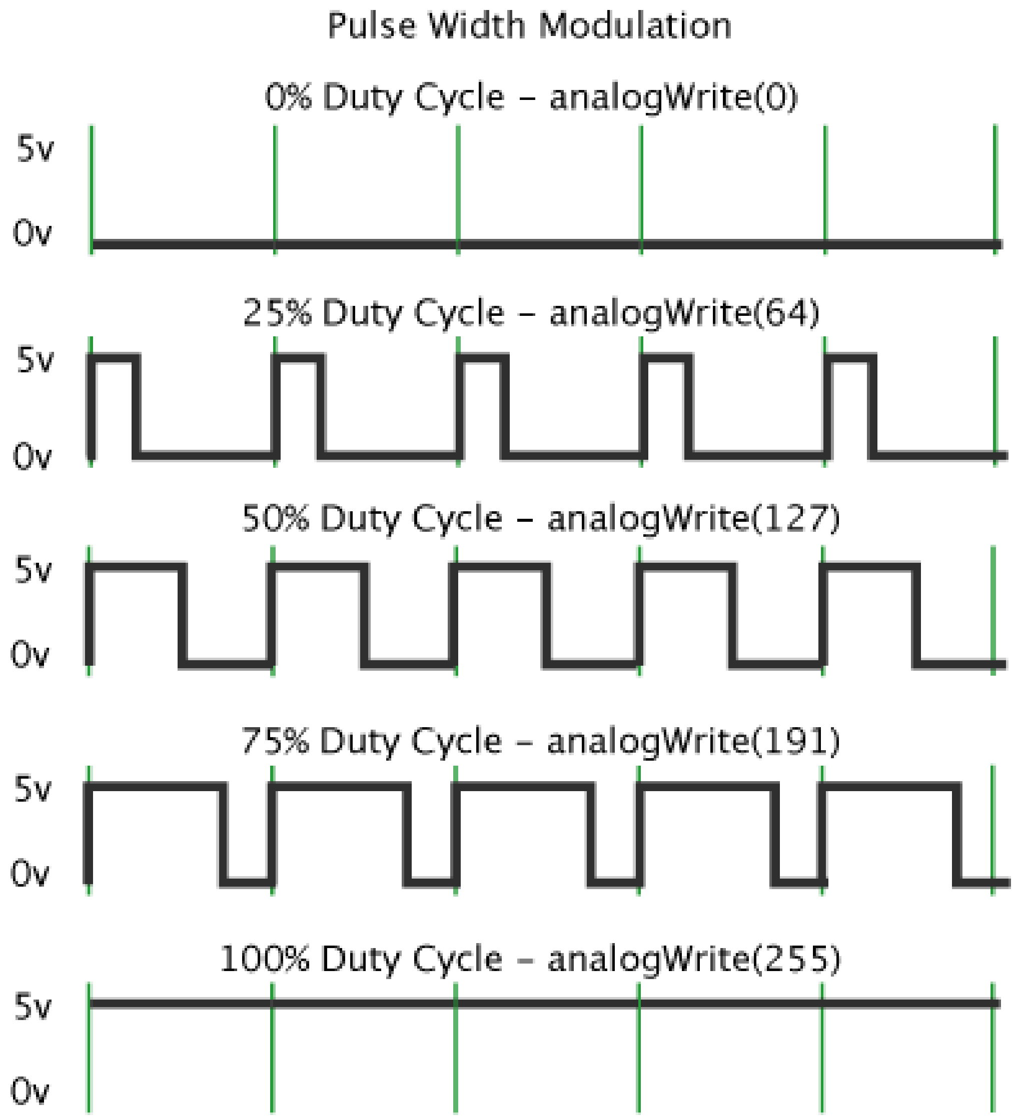
- PWM create ***simple square wave*** by ***varying*** the ***duty cycle***.
- It used to ***control*** an ***analog variable***, usually ***voltage*** or ***current***.
- Used in a variety of applications, ranging from telecommunications to robotic control.
- **Arduino** has **six PWM outputs**
  - Pin11, Pin10, Pin9, Pin6, Pin5, Pin3.

Pulse Width Modulation (PWM)

- **PWM depends on pulse width and duty cycle.**
- **The pulse width (also called a period) is a short duration of time in which the duty cycle will operate.**
- **The duty cycle is the proportion of time that the *pulse* is "*on*" or "*high*" and is expressed as a percentage.**
  - **100% duty cycle - continuously on**
  - **0% duty cycle - continuously off**

$$Duty\ cycle = \frac{pulse\ on\ time}{pulse\ period} \times 100\% = \frac{t_{on}}{T} \times 100\%$$

Pulse Width Modulation (PWM)



Pulse Width Modulation (PWM)

- **DC motor control is a very common task in robotics.**
  - ***Speed*** **of** ***DC motor*** **is proportional to the** ***applied DC voltage*****.**
- **If Conventional DAC output is used,**
  - drive it through an ***expensive*** and ***bulky power amplifier***
  - use the ***amplifier output*** to ***drive*** the ***motor***
- **Alternatively, a PWM signal can be used**
  - **to drive a power transistor directly**

Pulse Width Modulation (PWM)

Analog I/O

- **analogWrite(pin, value)**
  - Write an analog output on a digital pin. This is called PWM
  - ***PWM*** uses ***digital signals*** to ***control analog devices***.
  - Digital Pins # 3, # 5, #6, # 9, # 10, and # 11 can be used as PWM pins.

```
analogWrite(pin,value)

Parameters pin: the number of the pin you want to write
value: the duty cycle between 0 (always off, 0%) and 255 (always on, 100%)

Returns
None
```

Pulse Width Modulation (PWM)

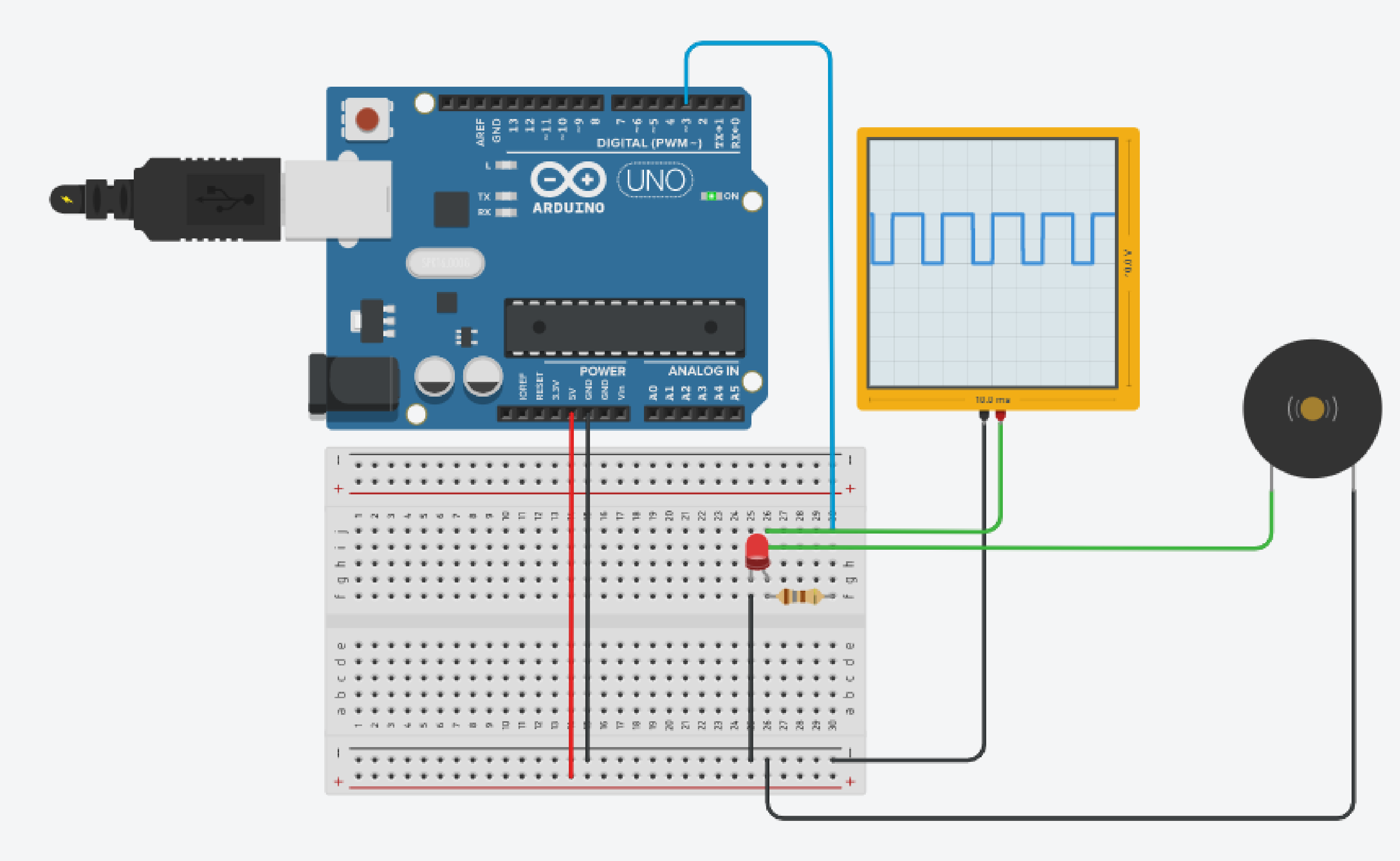
Example

Code an Arduino board to control the brightness of an LED. Monitor the PWM waveform in an oscilloscope and interface a buzzer to listen to different tones.

```
int brilho = 0;

void setup()
{
  pinMode(3, OUTPUT);
}

void loop()
{
  for (brilho = 0; brilho <= 255; brilho += 5) {
    analogWrite(3, brilho);
    delay(30); // Wait for 30 millisecond(s)
  }
  for (brilho = 255; brilho >= 0; brilho -= 5) {
    analogWrite(3, brilho);
    delay(30); // Wait for 30 millisecond(s)
  }
}
```

Pulse Width Modulation (PWM)

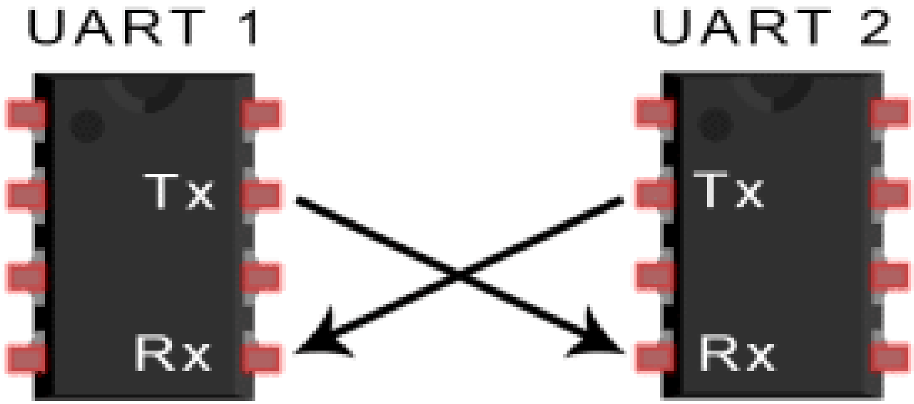
Example



UART

UART

- UART stands for *Universal Asynchronous Receiver/ Transmitter*
- It is used to **transmit** & **receive serially**
- *Two wires* are *required* to transmit and receive data *Tx* and *Rx Pin*



UART

- UARTs has ***no clock signal*** and follows ***asynchronous data communication***
- **Start and stop bits** are added to ***data packet*** for ***identifying beginning*** and ***end*** of data packet
  - Start bit and stop bit are used instead of clock signal

- ***After detecting start bit*** **UART starts** to **read** the **incoming data packet**
  - A specific frequency is used for data communication known as baud rate
  - **Baud rate** is expressed as ***bits per second (bps)***. It is a measure of speed of data transfer.
  - Both receiving and transmitting UART should work in same baud rate

UART

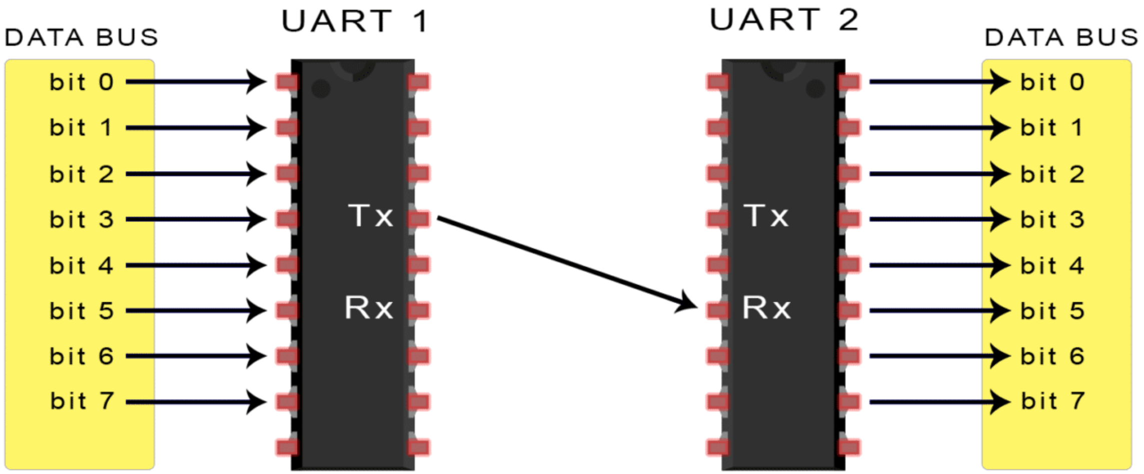
Parameters	Specification
Wires used	2
Maximum speed	115200 baud rate
Synchronous ?	No
Serial?	Yes
Max number of master	1
Max number of slave	1

UART: Working

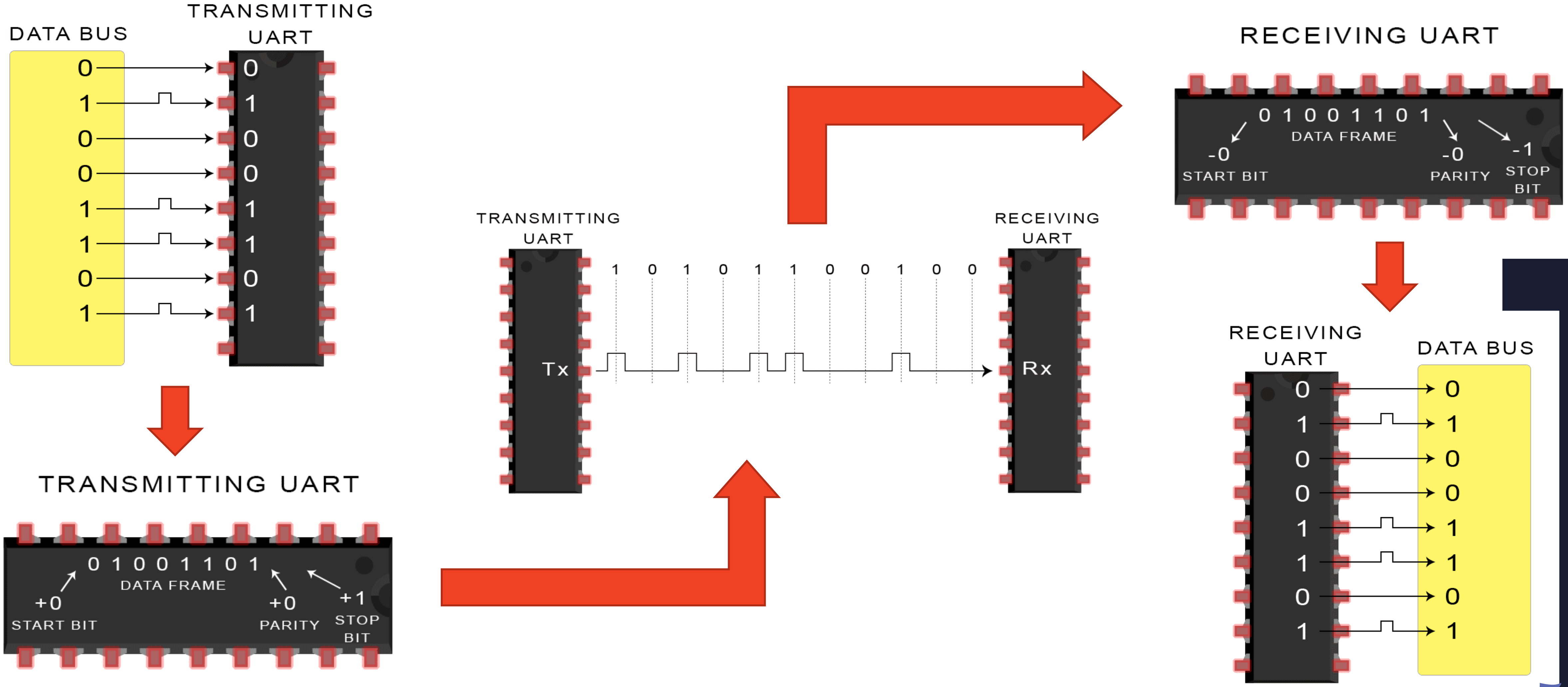
- In UART, ***data*** is ***transmitted*** in ***serial*** from ***data bus*** to ***transmitting UART***
- ***Transmitting UART adds*** a ***start*** bit, ***stop*** bit and a ***parity bit*** to ***parallel data*** from ***data bus***
- **Data** packet is **transmitted serially** from **Tx** to **Rx** pin
- **Rx** pin **reads** the **data** and converts parallelly by removing other bits.

Frame Format

1 start bit	5 to 9 data bit	0 or 1 parity bit	1 to 2 stop bits



UART: Working



https://www.circuitbasics.com/basics-uart-communication/

UART: In Arduino

- **UART** allows the **Atmega chip** to do **serial communication *while working*** on ***other tasks***, through **64 byte serial buffer**.
- **All Arduino boards** have at **least one serial port** (aka. UART or USART): **Serial**.
- **Serial** is used for **communication** between the **Arduino board** and a **computer** or **other devices**.
  - It ***communicates*** on ***digital pins 0*** (RX) and ***1*** (TX) as well as with the **computer via USB**.
  - *Thus, if you use these functions, you cannot also use pins 0 and 1 for digital input or output.*

UART: Arduino Functions

if (Serial)	• indicates whether or not the USB serial connection is open
Serial.available()	• Get the number of bytes available for reading from the serial port.
Serial.begin()	• Sets the data rate in bits per second (baud) for serial data transmission.
Serial.end()	• Disables serial communication, allowing the RX,TX to be used for input & output.
Serial.find()	• reads data from the serial buffer until the target is found.
Serial.println()	• Prints data as ASCII text followed by a carriage return and newline character.
Serial.read()	• Reads incoming serial data.
Serial.readString()	• reads characters from the serial buffer into a String.
Serial.write()	• Writes binary data to serial port. This data is sent as a byte or series of bytes

UART: In Arduino

- **Example-1:** **Print data received through serial communication on to the serial monitor of Arduino**

```
void setup() {
   Serial.begin(9600); //set up serial library baud rate to 9600
}

void loop() {
   if(Serial.available()) //if number of bytes (characters) available
                          // for reading from serial port
{
Serial.print("I received:"); //print I received
Serial.write(Serial.read()); //send what you read
   }
}
```

UART: In Arduino

- **Example-2: Arduino code for serial interface to blink switch ON LED when “a” is received on serial port**

```
int inByte;                                  // Stores incoming command
void setup() {
        Serial.begin(9600);
        pinMode(13, OUTPUT);                 // Led pin
        Serial.println("Ready");             // Ready to receive commands
}
void loop() {
        If(Serial.available() > 0) {         // A byte is ready to receive
                inByte = Serial.read();
        if(inByte == 'a') {                  // byte is 'a'
                digitalWrite(13, HIGH);
                Serial.println("LED - On");
}
        else {                               // byte isn't 'a'
                digitalWrite(13, LOW);
                Serial.println("LED - off");
} } }
```

UART: In Arduino

Arduino Software Serial Library

- *SoftwareSerial* library has been developed to allow **serial communication** on other **digital pins** of the **Arduino**
  - Uses software to ***replicate*** the ***functionality*** of the ***hardwired RX*** and ***TX*** lines hence the name "*SoftwareSerial*".
- It is ***possible*** to have ***multiple software serial ports*** with ***speeds up*** to *115200* bps.
  - This can be ***extremely helpful*** when the ***need arises*** to ***communicate*** with ***two or more serial enabled devices***
- **Limitations:**
  - **Maximum RX** speed is **57600bps**
  - RX doesn't work on Pin 13
  - If using multiple software serial ports, **only one receive data** at a time.

UART: In Arduino

Arduino Software Serial Library

SoftwareSerial()	• Need to enable serial communication
avaialble()	• gets the no of bits available for reading from serial port
begin()	• sets the speed of serial communication
overflow()	• checks the serial buffer overflow has occurred
peek()	• returns the character received in the serial port
read()	• returns the character that was received on the Rx pin of serial port
Print()	• works same as serial.print
Listen()	• enables the selected serial port to listen
Write()	• print data to transmit pin of software serial

UART: In Arduino

- **Example-3:** **Arduino code to Receives from the hardware serial, sends to software serial and Receives from software serial, sends to hardware serial.**

```
#include <SoftwareSerial.h>
SoftwareSerial mySerial(10, 11);                // RX, TX
void setup() {
   Serial.begin(57600);     // Open serial communications and wait for port to open:
  while (!Serial) {
                    ;       // wait for serial port to connect. Needed for native USB port only
  }
  mySerial.begin(4800);                         // set the data rate for the SoftwareSerial port
  mySerial.println("Hello, world?");
}
void loop() {                                   // run over and over
  if (mySerial.available()) {
    Serial.write(mySerial.read());
  }
  if (Serial.available()) {
    mySerial.write(Serial.read());
  } }
```

Sensors & **Actuators**

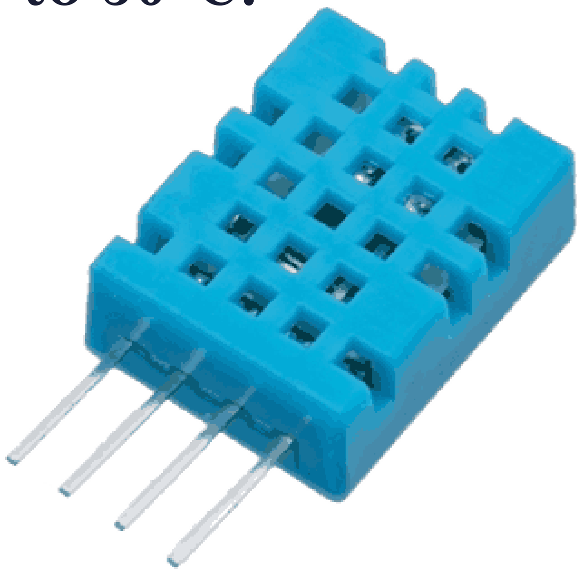
Sensor

- Electronic Elements
- Convert physical quantity/ measurements into electrical signals.
- Can be analog or digital

- Types:
  - Temperature
  - Humidity
  - Light
  - Sound, etc.

Sensor interfacing with Arduino

- **Digital Humidity and Temperature Sensor (DHT)**
- *DHT11 sensor measures and provides humidity and temperature values serially over a single wire.*
- **Measure relative humidity in percentage : 20 to 90% RH**
- **Temperature in degree Celsius in the range of 0 to 50°C.**
  - 4 Pins {1 … 4, from Left to Right}
    - Pin 1 : 3.3 – 5 V Power Supply
    - Pin 2 : Data
    - Pin 3 : Null
    - Pin 4 : Ground



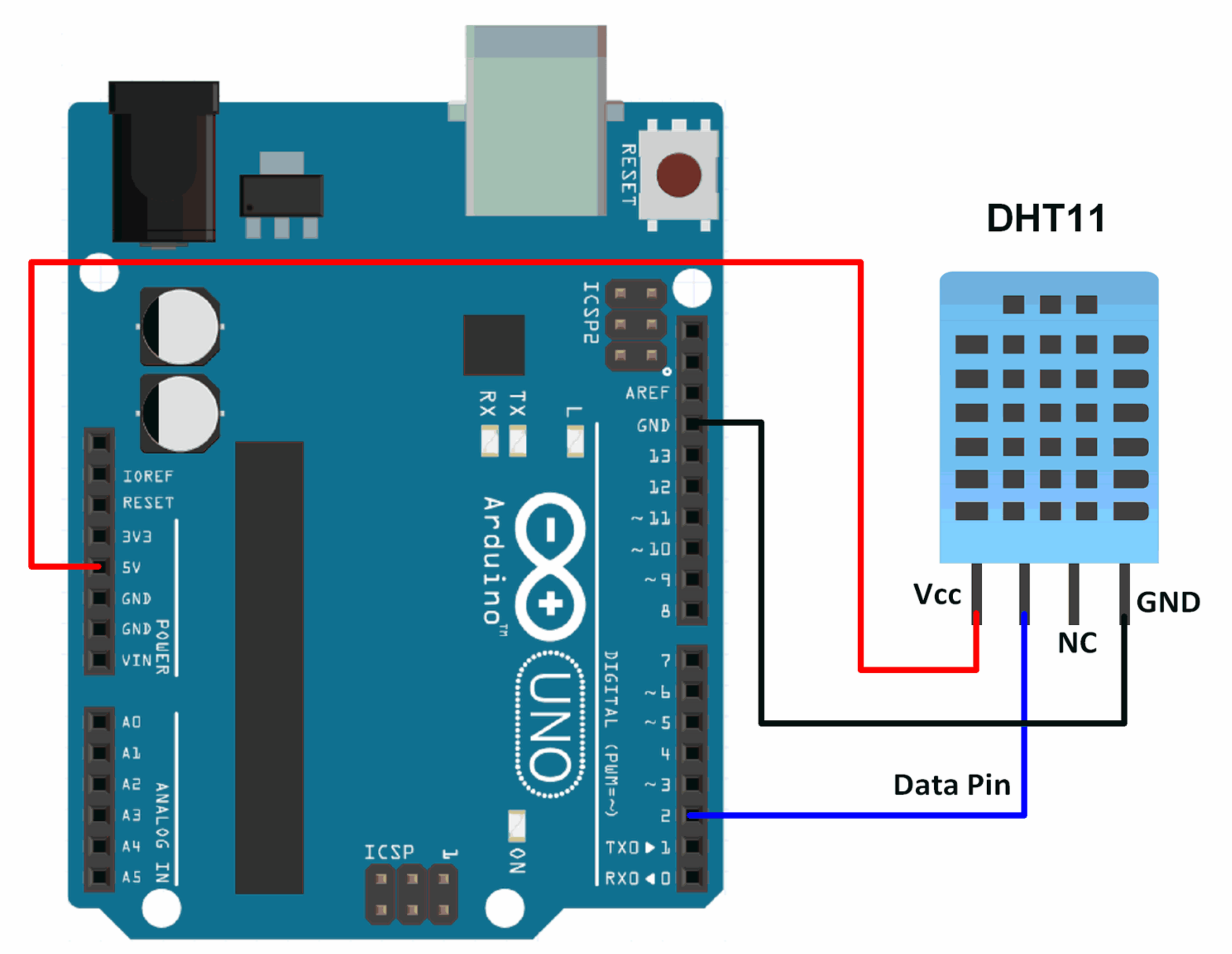
DHT11 Sensor

Sensor interfacing with Arduino

- **DHT Sensor Library**
  - Arduino supports special library for DHT11 and DHT22
  - Provides function to read Temperature & Humitdity values from data pin:
    - `dht.readHumidity()`
    - `dht.readTemperature()`

Sensor interfacing with Arduino

- **Connections**
  - Connect pin 1 to the supply.
  - Connect Pin 2, Data Pin, to any digital input; here pin 2 of board.
  - Connect Pin 4 to Ground pin of the board.
- *Import* **DHT_SENSOR** *Library*



Sensor interfacing with Arduino

❑ Code

```
#include<dht.h>;
DHT dht(2,DHT11);
float humidity;
float temperature;
value

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

void loop()
{
// read data from sensor and store in
variables
humidity = dht.readHumidity();
temperature dht.readTemperature();

// print values
Serial.print(“Humidity = ”)
Serial.print(humidity)
Serial.print(“%, Temperature = ”)
Serial.print(temperature)
Serial.print(“Celsius”)
Delay(2000);
}
```

Actuators

- Mechanical / Electro-mechanical Devices
- Converts Energy to Motion
- Mainly used to provide *controlled motion* to other components.
- *Working Principle*: Uses combination of various mechanical components like, motor assemblies with bearing, gears, etc.
- *Types of Motor Actuators*: Servo motor, Stepper motor, Hydraulic, AC motor,
- *Other Actuators*: Solenoid, Relay, etc.

Actuators : Servo Motor

- High Precision Motor
- Provides rotary motion 0 to 180 deg.
- 3 wire configuration:
  - Black/ Dark Wire : Ground
  - Red Wire : Supply voltage
  - Yellow Wire : Control signal

Actuators : Servo Motor

- Arduino provides a separate library – **SERVO**, to operate the servo motor.
- Instance can be created as: **`Servo myservo`**.

```
#include<Servo.h>;  // include servo lib.
int servoPin = 12;
Servo myServo; create a servo object

void setup()
{
myServo.attach(servoPin)
}
void loop()
{
myServo.write(0);      // Move servo motor by 0 deg.
delay(1000);
myServo.write(90);     // Move servo motor by 90 deg.
delay(1000)
myServo.write(180);    // Move servo motor by 180 deg.
delay(1000)
}
```

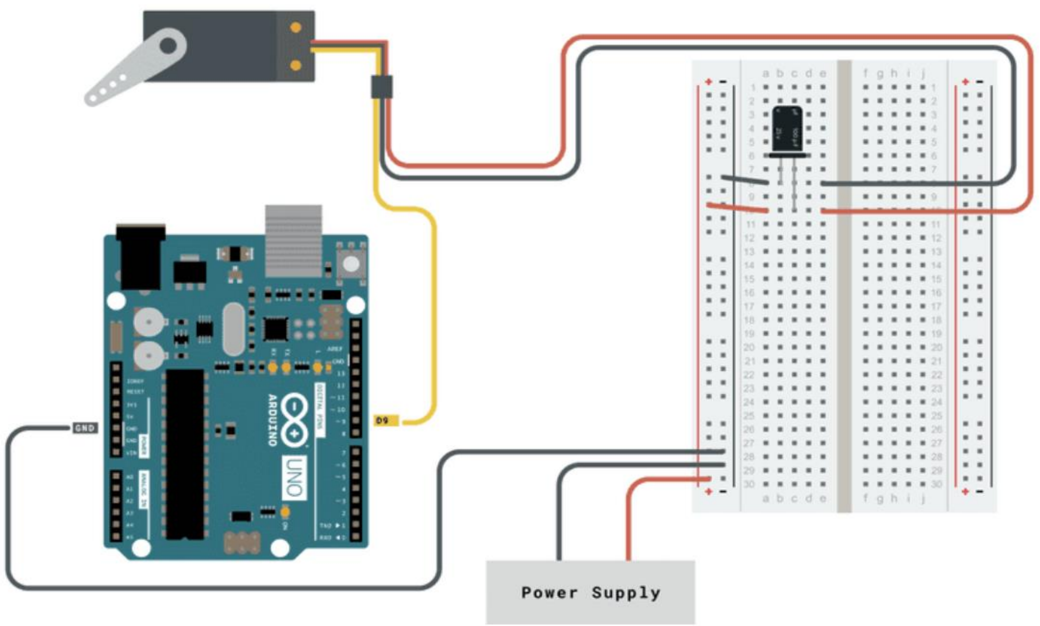
Actuators : Servo Motor

- *Connections*
  - Black Wire : GND of the board
  - Red Wire : 5V supply of board
  - Yellow Wire : Signal wire to the Pin (here 12 in prev. example)

- *Other SERVO functions*
  - Knob()
  - Sweep()
  - write()
  - writeMicroseconds()
  - read()
  - attached()
  - detatch()

Actuators : Servo Motor

❑ Knob Circuit

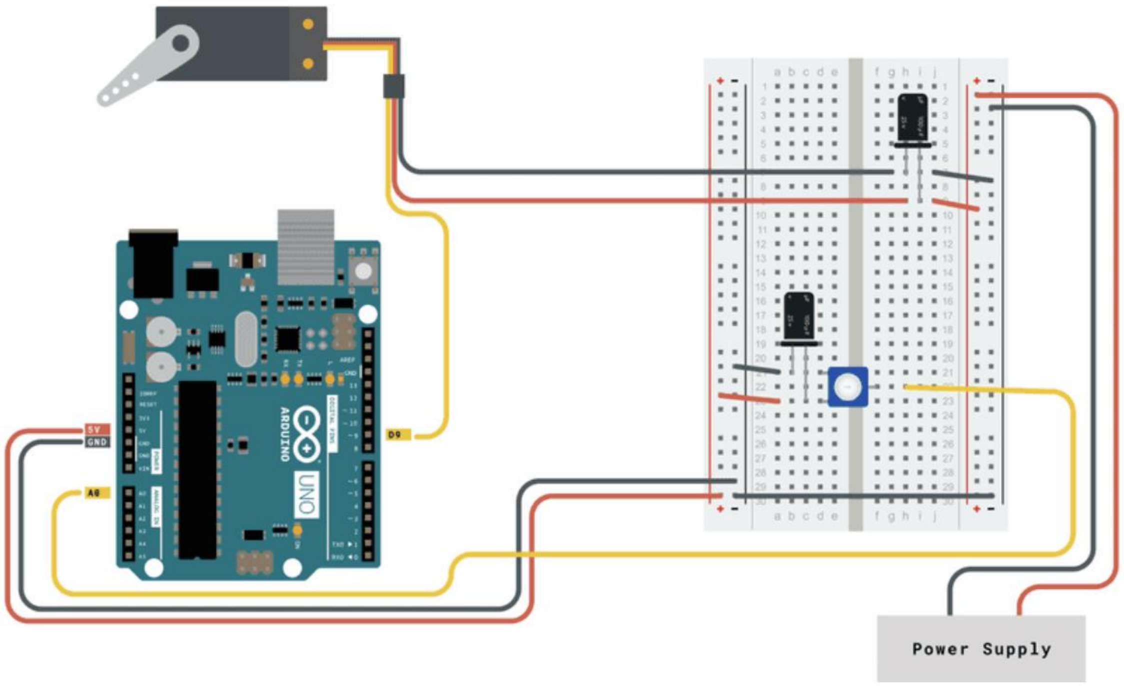



```
#include <Servo.h>

Servo myservo;  // create servo object to control a servo

int potpin = 0;  // analog pin used to connect the potentiometer
int val;    // variable to read the value from the analog pin

void setup() {
  myservo.attach(9);  // attaches the servo on pin 9 to the servo object
}

void loop() {
  val = analogRead(potpin);            // reads the value of the potentiometer
  val = map(val, 0, 1023, 0, 180);     // scale it to use it with the servo (va
  myservo.write(val);                  // sets the servo position according to
  delay(15);                           // waits for the servo to get there
}
```

Actuators : Servo Motor

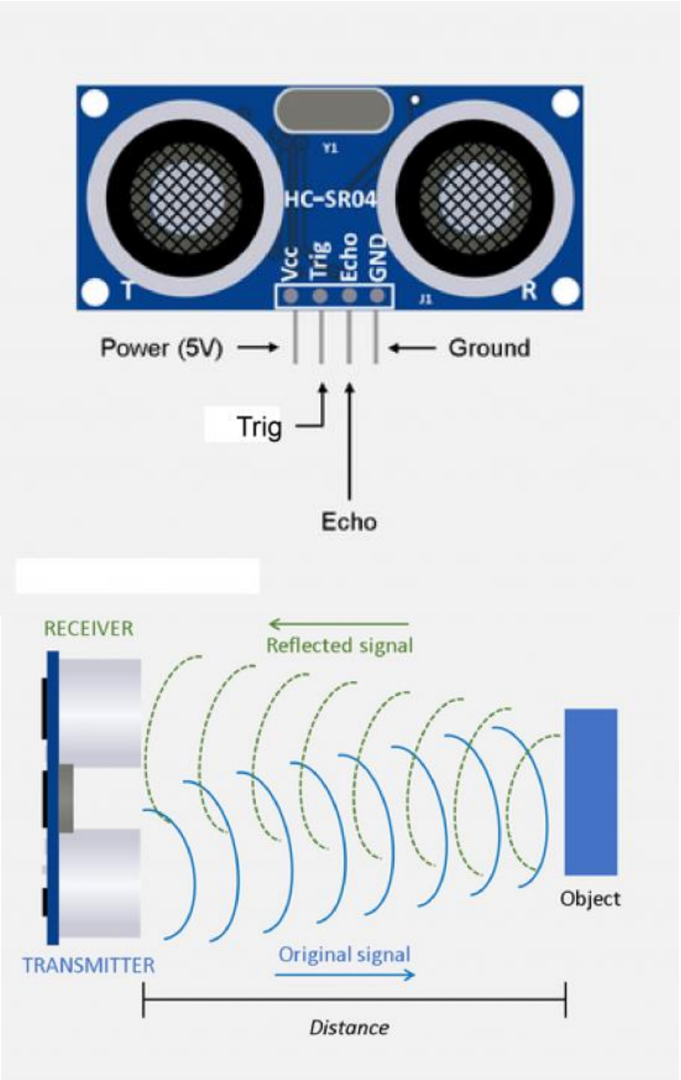
❑ Sweep Circuit

```
#include <Servo.h>

Servo myservo;  // create servo object to control a servo
// twelve servo objects can be created on most boards

int pos = 0;    // variable to store the servo position

void setup() {
  myservo.attach(9);  // attaches the servo on pin 9 to the servo object
}

void loop() {
  for (pos = 0; pos <= 180; pos += 1) { // goes from 0 degrees to 180 degrees
    // in steps of 1 degree
    myservo.write(pos);              // tell servo to go to position in variabl
    delay(15);                       // waits 15ms for the servo to reach the p
  }
  for (pos = 180; pos >= 0; pos -= 1) { // goes from 180 degrees to 0 degrees
    myservo.write(pos);              // tell servo to go to position in variabl
    delay(15);                       // waits 15ms for the servo to reach the p
  }
}
```



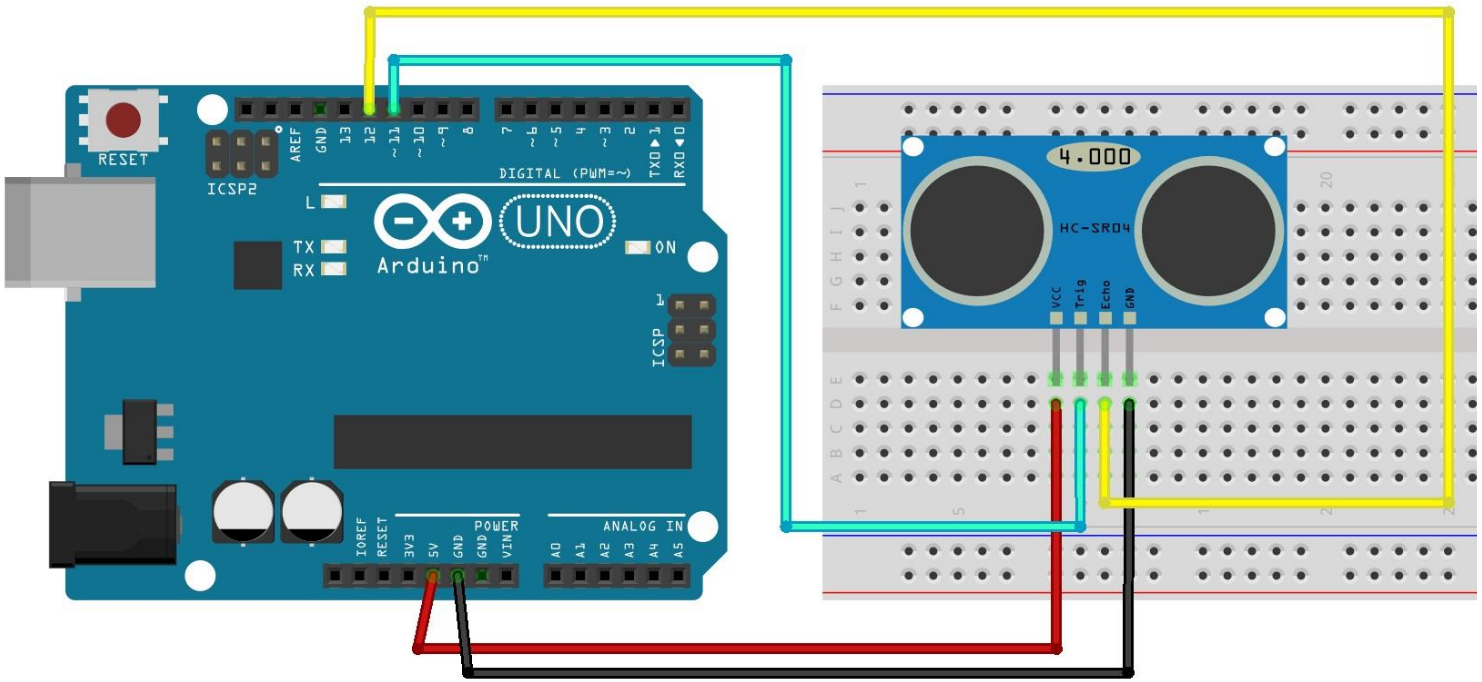
Sensor : Ultrasonic Sensor

- An electronic sensor that **measures the distance of a target object by emitting ultrasonic sound waves**, and **converts the reflected sound into an electrical signal**.
- Ultrasonic waves **travel faster** than **the speed of audible sound** (i.e. the sound that humans can hear).
- Ultrasonic sensors have two main components:
  - **The transmitter** (which emits the sound using piezoelectric crystals) and
  - **The receiver** (which encounters the sound after it has travelled to and from the target).
- **Fromula**: $\boldsymbol{D} = \frac{1}{2}\boldsymbol{T} * \boldsymbol{C}$
  - where D is the distance, T is the time, and C is the speed of sound ~ **343 meters/second**).



Sensor : Ultrasonic Sensor

❑ Connections



Technical Specifications

- Power Supply – +5V DC
- Quiescent Current – <2mA
- Working Current – 15mA
- Effectual Angle – <15°
- Ranging Distance – 2cm – 400 cm/1″ – 13ft
- Resolution – 0.3 cm
- Measuring Angle – 30 degree

Ultrasonic Sensor	Arduino
VCC	5V
Trig	Pin 11
Echo	Pin 12
GND	GND

Sensor : Ultrasonic Sensor

```
int trigPin = 11; // Trigger
int echoPin = 12; // Echo
long duration, cm, inches;
void setup()
{
//Serial Port begin
Serial.begin (9600);

//Define inputs and outputs
pinMode(trigPin, OUTPUT);
pinMode(echoPin, INPUT);
}

void loop()
{
// The sensor is triggered by a HIGH pulse of 10
// or more microseconds.
// Give a short LOW pulse beforehand to ensure a
clean HIGH pulse:
digitalWrite(trigPin, LOW);
delayMicroseconds(5);
digitalWrite(trigPin, HIGH);
delayMicroseconds(10);
digitalWrite(trigPin, LOW);

// Read the signal from the sensor:
// a HIGH pulse whose duration is the time (in
microsec.)
// from the sending of the ping to the reception of
// its echo off of an object.

pinMode(echoPin, INPUT);
duration = pulseIn(echoPin, HIGH);

// Convert the time into a distance
cm = (duration/2) / 29.1;
// Divide by 29.1 or multiply by 0.0343
inches = (duration/2) / 74;
// Divide by 74 or multiply by 0.0135
Serial.print(inches);
Serial.print("in, ");
Serial.print(cm);
Serial.print("cm");
Serial.println();
delay(250);
}
```

Memory **Interfacing**

Memory Interfacing

- Arduino interface with an **AT25HP512 Atmel serial EEPROM** using the **Serial Peripheral Interface (SPI) protocol**.
- Such **EEPROM chips** are **very useful** for **data storage**.
  - Note that the chip on the Arduino board contains an internal EEPROM.

EEPROM interface using SPI

- **Serial Peripheral Interface (SPI):**
  - ***Synchronous serial data*** protocol used by ***μC*** for ***communicating*** with ***one*** or ***more peripheral*** devices quickly ***over short distances***.

- **An SPI connection: One master device** (usually a μC) which ***controls*** the ***peripheral*** devices.

- ***Three lines common to all the devices,***
  - **Master In Slave Out (MISO)** - ***Slave line*** for ***sending data*** to the ***master***,
  - **Master Out Slave In (MOSI)** - ***Master line*** for ***sending data*** to ***peripherals***,
  - **Serial Clock (SCK)** - ***Clock pulses*** which ***synchronize data transmission generated*** by the ***master***, and
  - **Slave Select pin** - ***Allocated*** on ***each device*** which the ***master*** can ***use*** to ***enable*** and ***disable specific devices*** and ***avoid false transmissions*** due to line ***noise***.

EEPROM interface using SPI

- **Arduino SPI Control Register (SPCR):**
  - Determine SPI settings
  - **Control Registers Code:**
    - ***Control settings*** for various ***microcontroller functionalities***.
    - Usually ***each bit*** in a ***control register effects*** a ***particular setting***, such as ***speed*** or ***polarity***.
- **Data Registers (SPDR):**
  - Simply hold bytes.
  - For example, the **SPI data register (SPDR)** holds the **byte** which is **about** to be **shifted out** the **MOSI line**, and the data which has just been shifted in the MISO line.
- **Status Registers (SPSR):**
  - ***Change*** their ***state*** based on ***various microcontroller conditions***.
  - For example, the ***7th bit*** of the ***SPI status register (SPSR)*** gets set to *1* when a ***value*** is ***shifted in or out*** of the ***SPI***.

EEPROM interface using SPI

```
SPCR
| 7    | 6    | 5    | 4    | 3    | 2    | 1    | 0    |
| SPIE | SPE  | DORD | MSTR | CPOL | CPHA | SPR1 | SPR0 |

SPIE - Enables the SPI interrupt when 1

SPE - Enables the SPI when 1

DORD - Sends data least Significant Bit First when 1, most Significant Bit first wh

MSTR - Sets the Arduino in master mode when 1, slave mode when 0

CPOL - Sets the data clock to be idle when high if set to 1, idle when low if set to

CPHA - Samples data on the falling edge of the data clock when 1, rising edge when

SPR1 and SPR0 - Sets the SPI speed, 00 is fastest (4MHz) 11 is slowest (250KHz)
```

EEPROM interface using SPI

- **AT25HP512** is a **65,536 byte serial EEPROM**.
- **Supports** SPI **modes 0** and **3**,
- Runs at up to **10MHz** at **5v** and can run at **slower speeds** down to **1.8v**.
- Memory is organized as **512 pages** of **128 bytes** each.
- It can **only be written 128 bytes** at a time, but it **can be read 1-128 bytes at a time**.
- Device also **offers** various degrees of **write protection** and a **hold pin**

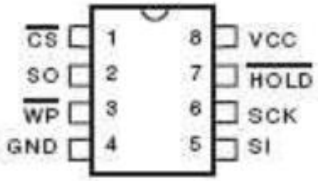
EEPROM interface using SPI

- **Device** is **enabled** by **pulling** the **Chip Select** (CS) **pin low**.
- **Instructions** are sent as **8 bit operational codes** (opcodes) and
- Are **shifted in** on the **rising edge of the data clock**.
- It takes the EEPROM about **10 milliseconds** to **write a page** (128 bytes) of data,
- So a **10ms pause** should **follow each** EEPROM **write** routine.

Pin Configurations

Pin Name	Function
$\overline{\text{CS}}$	Chip Select
SCK	Serial Data Clock
SI	Serial Data Input
SO	Serial Data Output
GND	Ground
VCC	Power Supply
$\overline{\text{WP}}$	Write Protect
$\overline{\text{HOLD}}$	Suspends Serial Input

8-pin PDIP

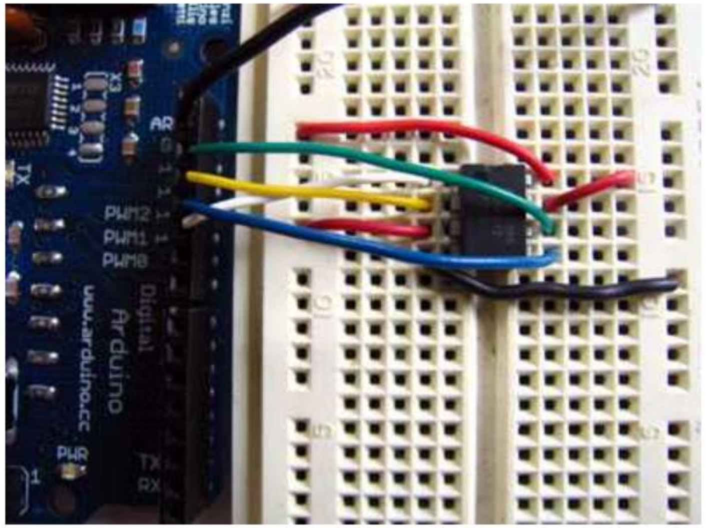


EEPROM interface using SPI

- ***Pin Configuration:***
  - Connect **EEPROM**
    - **Pins 3, 7 and 8** to **5v**
    - **Pin 4** to ground.
    - **Pin 1** to Arduino **pin 10** (Slave Select - SS),
    - **Pin 2** to Arduino **pin 12** (Master In Slave Out - MISO),
    - **Pin 5** to Arduino **pin 11** (Master Out Slave In - MOSI),
    - **Pin 6** to Arduino **pin 13** (Serial Clock - SCK).

Pin Configurations

Pin Name	Function
$\overline{CS}$	Chip Select
SCK	Serial Data Clock
SI	Serial Data Input
SO	Serial Data Output
GND	Ground
VCC	Power Supply
$\overline{WP}$	Write Protect
$\overline{HOLD}$	Suspends Serial Input



EEPROM interface using SPI

- 1st step is **setting up** our **pre-processor directives**.
- We define the pins we will be using for our SPI connection, *DATAOUT*, *DATAIN*, *SPICLOCK* and *SLAVESELECT*. Then we define our opcodes for the EEPROM.

```
#define DATAOUT 11//MOSI
#define DATAIN  12//MISO
#define SPICLOCK  13//sck
#define SLAVESELECT 10//ss

//opcodes
#define WREN  6
#define WRDI  4
#define RDSR  5
#define WRSR  1
#define READ  3
#define WRITE 2
```

- `char buffer [128]` this is a **128 byte array** we will be using to store the data for the EEPROM write:

```
byte eeprom_output_data;
byte eeprom_input_data=0;
byte clr;
int address=0;
//data buffer
char buffer [128];
```

EEPROM interface using SPI

- First we **initialize** our **serial connection**, set our input and output pin modes and set the ***SLAVESELECT*** line high to start.
- This **deselects the device** and ***avoids any false transmission messages*** due to line noise:

```
void  setup()
{

   Serial.begin(9600);

   pinMode(DATAOUT, OUTPUT);

   pinMode(DATAIN, INPUT);

   pinMode(SPICLOCK,OUTPUT);

   pinMode(SLAVESELECT,OUTPUT);

   digitalWrite(SLAVESELECT,HIGH); //disable device
```

EEPROM interface using SPI

- Set the **SPI Control register (SPCR)** to the binary value **01010000**.

- In the control register each bit sets a different functionality.
  - ***8th bit*** **disables** the **SPI interrupt**,
  - ***7th bit*** **enables** the **SPI**,
  - ***6th bit*** **chooses transmission** with the **most significant bit** going first,
  - ***5th bit*** puts the **Arduino** in **Master** mode,
  - ***4th bit*** sets the **data clock idle** when it is **low**,
  - ***3rd bit*** sets the **SPI** to **sample data** on the **rising edge** of the data clock, and
  - ***2nd & 1st bits*** set the **speed** of the **SPI** to **system speed** / 4 (the fastest).

- After ***setting*** our ***SPCR*** up we ***read*** the ***SPI status register*** (***SPSR***) and ***data register*** (***SPDR***) in to the **junk `clr` variable** to **clear out any spurious data** from **past runs**:

```
// SPCR = 01010000
//interrupt disabled, spi enabled, msb 1st,master, clk low when idle,
//sample on leading edge of clk, system clock/4 rate (fastest)
SPCR = (1<<SPE)|(1<<MSTR);
clr=SPSR;
clr=SPDR;
delay(10);
```

EEPROM interface using SPI

- **EEPROM MUST** be **write enabled** before **every write instruction**.
- To **send the instruction** we **pull** the **SLAVESELECT** line **low**, **enabling** the **device**, and then **send** the **instruction** using the `spi_transfer` **function**.
- Note that we use the **WREN opcode** we defined at the **beginning of the program**.
- Finally we **pull** the **SLAVESELECT** line **high again** to **release** it:

```
//fill buffer with data
  fill_buffer();
  //fill eeprom w/ buffer
  digitalWrite(SLAVESELECT,LOW);
  spi_transfer(WREN); //write enable
  digitalWrite(SLAVESELECT,HIGH);
```

EEPROM interface using SPI

- Now we pull the **SLAVESELECT** line **low** to **select** the **device again** after a **brief delay**.
- We send a **WRITE instruction** to tell the EEPROM we will be **sending data** to record into **memory**.
- We send the **16 bit address** to **begin writing** at in **two bytes**, **Most Significant Bit first**.
- Next we send our **128 bytes** of **data** from our **buffer array**, *one byte after another without pause*.
- Finally we set the **SLAVESELECT** pin **high** to **release** the **device** and pause to allow the EEPROM to write the data:

```
delay(10);

  digitalWrite(SLAVESELECT,LOW);

  spi_transfer(WRITE); //write instruction

  address=0;

  spi_transfer((char)(address>>8));   //send MSByte address first

  spi_transfer((char)(address));      //send LSByte address

  //write 128 bytes

  for (int I=0;I<128;I++)

  {

    spi_transfer(buffer[I]); //write data byte

  }

  digitalWrite(SLAVESELECT,HIGH); //release chip

  //wait for eeprom to finish writing

  delay(3000);
```

EEPROM interface using SPI

- In the **main loop** we just **read one byte** at a **time from** the **EEPROM** and ***print*** it ***out*** the ***serial port***.
- We add a line feed and a pause for readability.
- Each time ***through the loop*** we **increment** the **EEPROM address** to read.
- When the **address increments** to **128** we **turn** it **back to 0** because we have only filled 128 addresses in the EEPROM with data.

```
void loop()
{
  eeprom_output_data = read_eeprom(address);
  Serial.print(eeprom_output_data,DEC);
  Serial.print('\n',BYTE);
  address++;
  delay(500); //pause for readability
}
```

EEPROM interface using SPI

- The ***spi_transfer*** function **loads** the **output data** into the **data transmission register**, thus starting the SPI transmission.
- It **polls a bit** to the **SPI Status register** (**SPSR**) to ***detect*** when the ***transmission*** is ***complete*** using a bit mask, ***SPIF***.
- It then ***returns any data*** that ***has been shifted*** in to the ***data register*** by the EEPROM.

```
char spi_transfer(volatile char data)
{
   SPDR = data;                     // Start the transmission
   while (!(SPSR & (1<<SPIF)))      // Wait for the end of the transmission
   {
   };
   return SPDR;                     // return the received byte
}
```

EEPROM interface using SPI

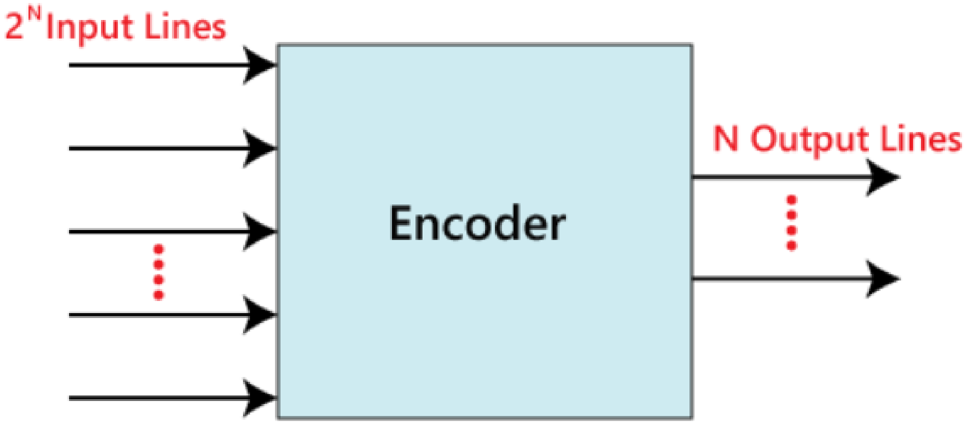
- The ***read_eeprom*** function **allows** us to **read data back out** of the EEPROM.
- First we set the **SLAVESELECT** line **low** to **enable** the device.
- Then we transmit a **READ instruction**, followed by the **16-bit address** we wish to read from, Most Significant Bit first.
- Next we send a ***dummy byte*** to the ***EEPROM*** for the purpose of ***shifting*** the ***data out***.
- Finally we **pull** the **SLAVESELECT** line **high** again to release the device after reading one byte, and return the data.

```
byte read_eeprom(int EEPROM_address)
{
  //READ EEPROM
  int data;
  digitalWrite(SLAVESELECT,LOW);
  spi_transfer(READ); //transmit read opcode
  spi_transfer((char)(EEPROM_address>>8));    //send MSByte address first
  spi_transfer((char)(EEPROM_address));       //send LSByte address
  data = spi_transfer(0xFF); //get data byte
  digitalWrite(SLAVESELECT,HIGH); //release chip, signal end transfer
  return data;
}
```

Encoders & **Decoders**

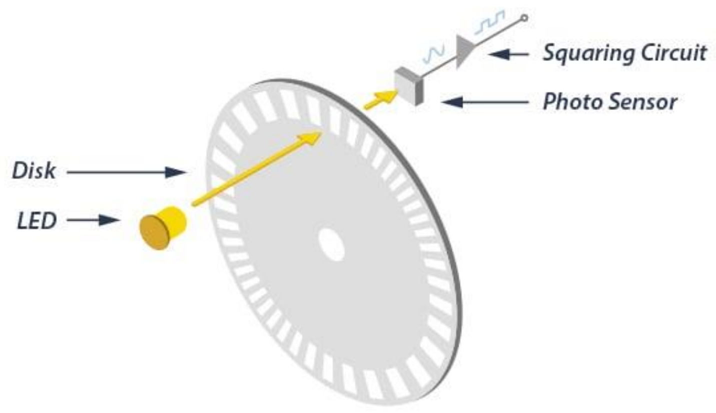
Encoder

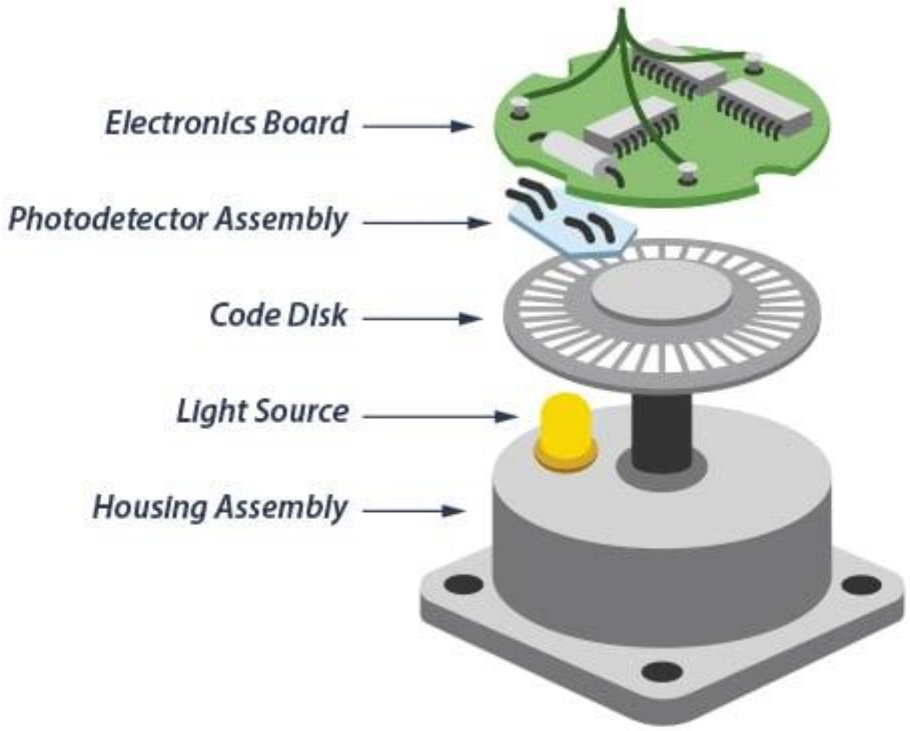
- Encoder is a **combinational circuit** that ***converts*** a ***set of input signals*** into a ***coded output***.
- Essentially, an encoder has **multiple input lines** and **fewer output lines**, and it generates a ***binary code*** corresponding to the ***input signal*** that is ***active*** (high)
- It has maximum of ***2^n input lines*** and ***'n' output lines***.
- It will produce a **binary code equivalent** to the **input**, which is active High.
- So, the encoder encodes ***2^n*** **input lines** with ***'n' bits***.



Working of Encoder

- A beam of light emitted from an LED passes through the Code Disk (see Figure 1), which is patterned with opaque lines , much like the spokes on a bike wheel.
- As the encoder shaft rotates, the light beam from the LED is interrupted by the opaque lines on the Code Disk before being picked up by the Photodetector Assembly.
- This produces a pulse signal: light = on; no light = off.
- The signal is sent to the counter or controller, which will then send the signal to produce the desired function.

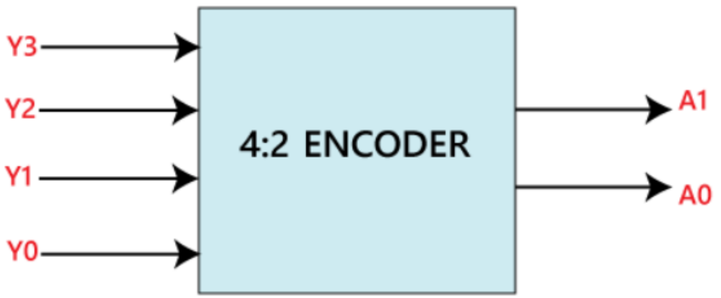




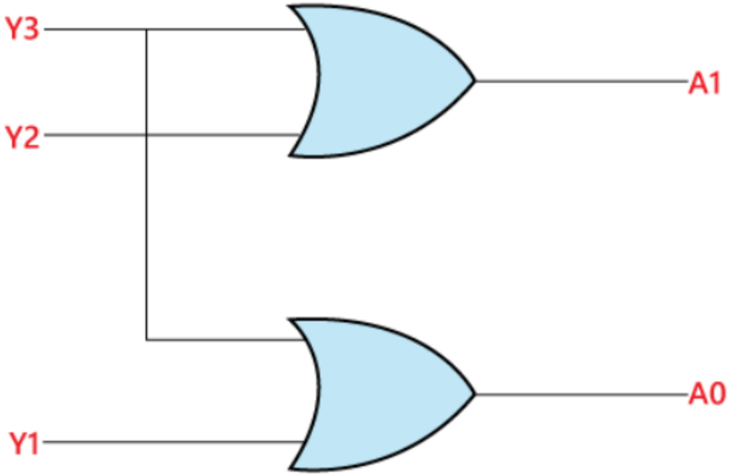
Truth Table of Encoder

❑ 4 to 2 Encoder

Block Diagram:



Circuit Diagram:



Expression

$$A_1 = Y_3 + Y_2$$
$$A_0 = Y_2 + Y_1$$

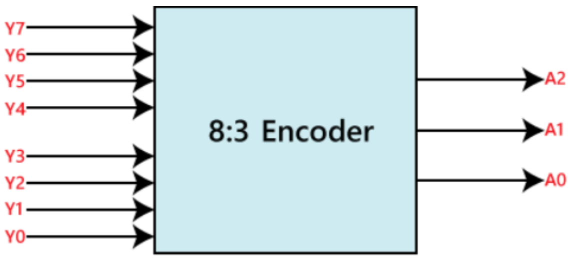
Truth Table

Inputs				Outputs	
Y_3	Y_2	Y_1	Y_0	A_1	A_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

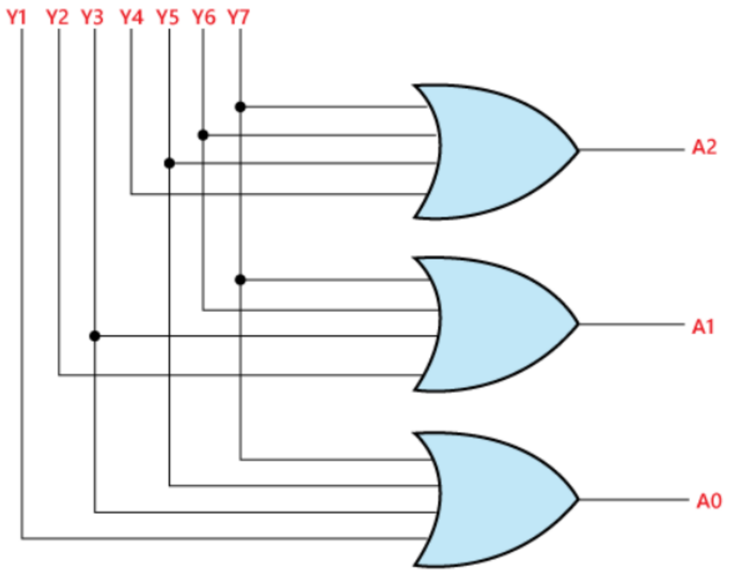
Truth Table of Encoder

- 8 to 3 (Octal to Binary) Encoder

Block Diagram:



Circuit Diagram:



Expression

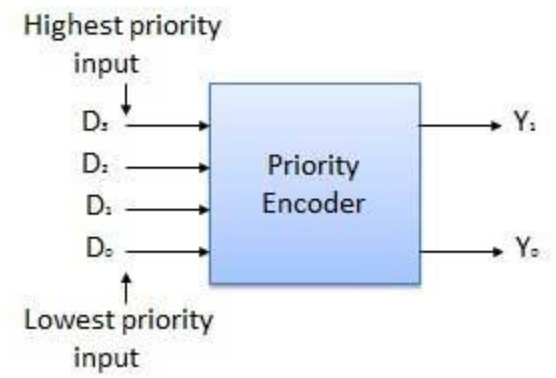
$$A_0 = Y_7 + Y_5 + Y_3 + Y_1 A_1$$
$$= Y_7 + Y_6 + Y_3 + Y_2 A_2$$
$$= Y_7 + Y_6 + Y_5 + Y_4$$

Truth Table

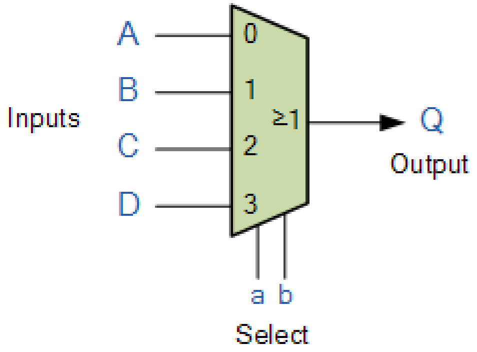
Inputs								Outputs		
Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0	A_2	A_1	A_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

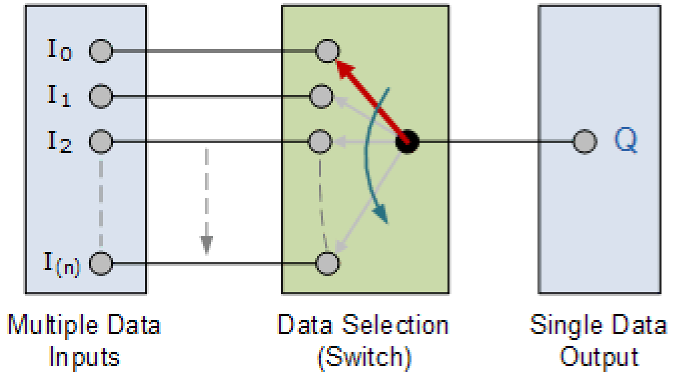
Types of Encoder

- ## Priority Encoder
  - A 4 to 2 priority encoder has 4 inputs: Y_3, Y_2, Y_1 & Y_0, and 2 outputs: A_1 & A_0.
  - Here, the input, Y_3 has the highest priority,
  - Whereas the input, Y_0 has the lowest priority.
  - *In this case, even if more than one input is '1' at the same time, the output will be the (binary) code corresponding to the input, which is having higher priority.*



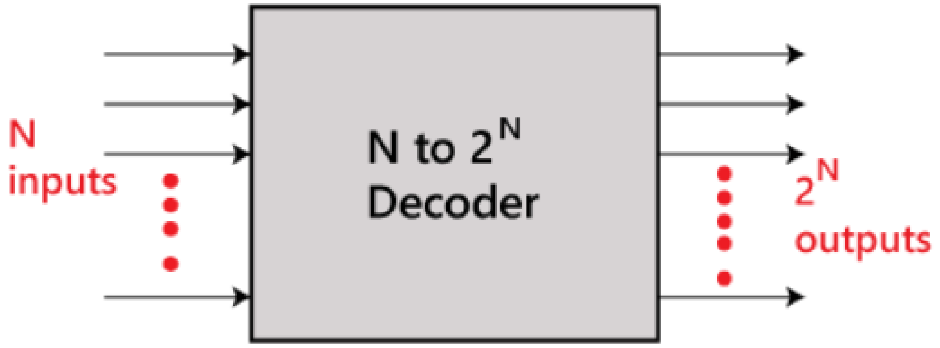
- ## Multiplexer
  - An encoder with **enable pins** is called multiplexer.
  - MUX is a combinational circuit that has **several inputs** and **only one output**.
  - MUX ***directs one of the inputs*** to its ***output line*** by using a ***control bit word*** (selection line) to its ***select lines***.
  - MUX acts like an **electronic switch**.





Decoder

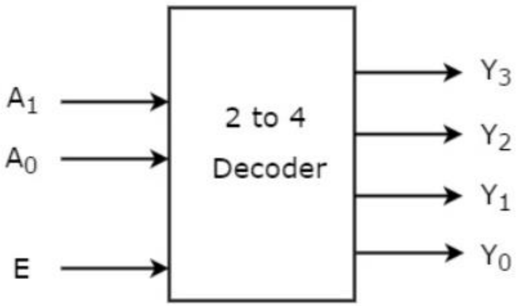
- The combinational circuit that ***change*** the ***binary information*** into ***2^N output lines*** is known as Decoders.
- The binary information is passed in the form of N input lines.
- The output lines define the 2^N-bit code for the binary information.
- At a time, only one input line is activated for simplicity.
- The produced 2^N-bit output code is equivalent to the binary information.



Truth Table of Decoder

❑ 2 : 4 Decoder

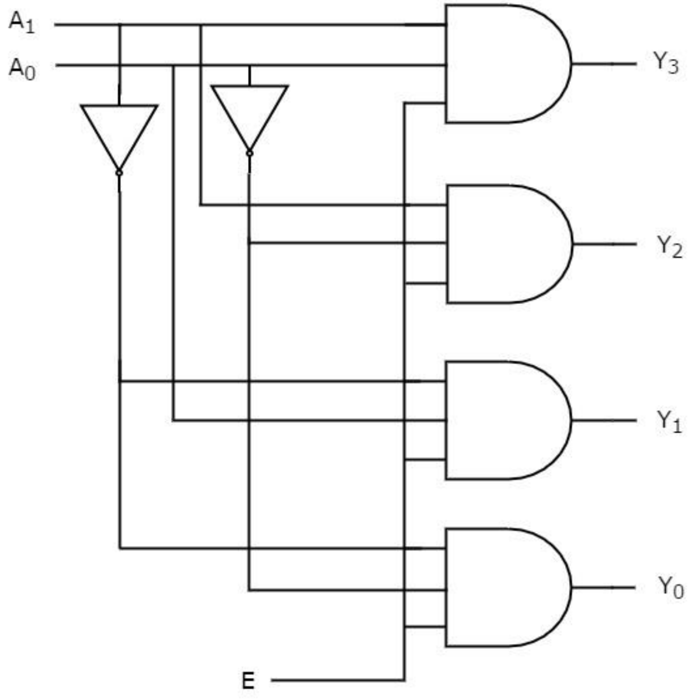
Block Diagram:



Expression

$$Y_3 = E.A_1.A_0 Y_3 = E.A_1.A_0$$
$$Y_2 = E.A_1.A_0' Y_2 = E.A_1.A_0'$$
$$Y_1 = E.A_1'.A_0 Y_1 = E.A_1'.A_0$$
$$Y_0 = E.A_1'.A_0' Y_0 = E.A_1'.A_0'$$

Circuit Diagram:



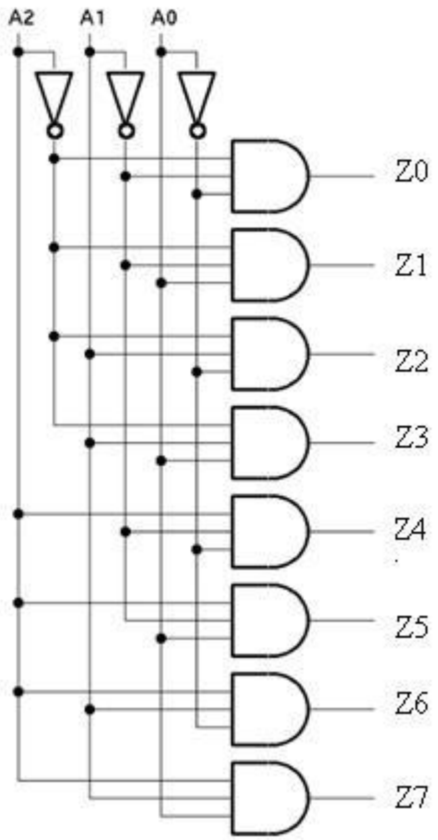
Truth Table

Enable	Inputs		Outputs			
E	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	x	x	0	0	0	0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0

3 to 8 Decoder

- In 3 to 8 line decoder, it includes three inputs and eight outputs. Here the inputs are represented through A, B & C whereas the outputs are represented through D0, D1, D2…D7.

$D0 = A'B'C'$
$D1 = A'B'C$
$D2 = A'BC'$
$D3 = A'BC$
$D4 = AB'C'$
$D5 = AB'C$
$D6 = ABC'$
$D7 = ABC$



A	B	C	D0	D1	D2	D3	D4	D5	D6	D7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	1	0	0	0	0	0	0	1

Watch Dog Timers

Watch Dog Timer in Arduino

- Arduino UNO board has ATmega328P chip as its controlling unit.
- The ATmega328P has a **Watchdog Timer** which is a useful feature to help the **system recover** from scenarios where the **system hangs** or **freezes due to errors** in the **code written** or **due to conditions** that may **arise** due to **hardware issues**.

Watch Dog Timer in Arduino

- **Working:**
  - Watchdog timer ***needs*** to be ***configured according*** to the ***need*** of the ***application***.
  - The watchdog timer uses an **internal 128kHz clock** source.
  - When **enabled**, it ***starts counting from 0*** to ***a value*** selected by the user.
  - *If the **watchdog timer** is **not reset** by the **time it reaches the user selected value**, the **watchdog resets** the **microcontroller**.*
  - **ATmega328P watchdog timer** can be ***configured for 10 different time settings*** (the time after which the watchdog timer overflows, thus causing a reset).
    - The various times are : 16ms, 32ms, 64ms, 0.125s, 0.25s, 0.5s, 1s, 2s, 4s and 8s.

Watch Dog Timer in Arduino

- **Example:**
  - **A simple example of LED blinking.**
  - The LEDs are blinked for a certain time before entering a while(1) loop. The while(1) loop is used as a substitute for a system in the hanged state.
  - Since the watchdog timer is not reset when in the while(1) loop, the watchdog causes a system reset and the LEDs start blinking again before the system hangs and restarts again. This continues in a loop.
  - Here, we will be using the on-board LED connected to the pin 13 of the Arduino UNO board. For this example sketch, the only thing required is the Arduino UNO board.

Watch Dog Timer in Arduino

```
void setup()   {
    Serial.begin(9600); /* Define baud rate for serial communication */
    Serial.println("Watchdog Demo Starting");
    pinMode(13, OUTPUT);
    wdt_disable();     /* Disable the watchdog and wait for more than 2 seconds */
    delay(3000);       /* Done so that the Arduino doesn't keep resetting infinitely
                       in case of wrong configuration */
    wdt_enable(WDTO_2S); /* Enable the watchdog with a timeout of 2 seconds */
}
void loop()
{
    for(int i = 0; i<20; i++)             /* Blink LED for some time */
        {
            digitalWrite(13, HIGH);
            delay(100);
            digitalWrite(13, LOW);
            delay(100);
            wdt_reset();          /* Reset the watchdog */
        }
while(1);                         /* Infinite loop. Will cause watchdog timeout and
system reset. */
}
```