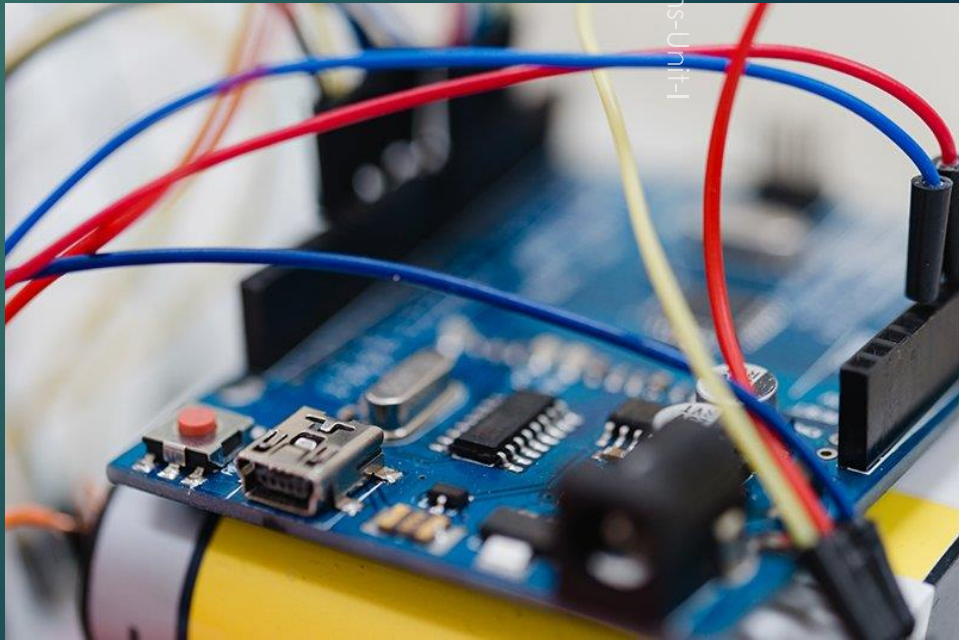
INTRODUCTION TO EMBEDDED SYSTEMS

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

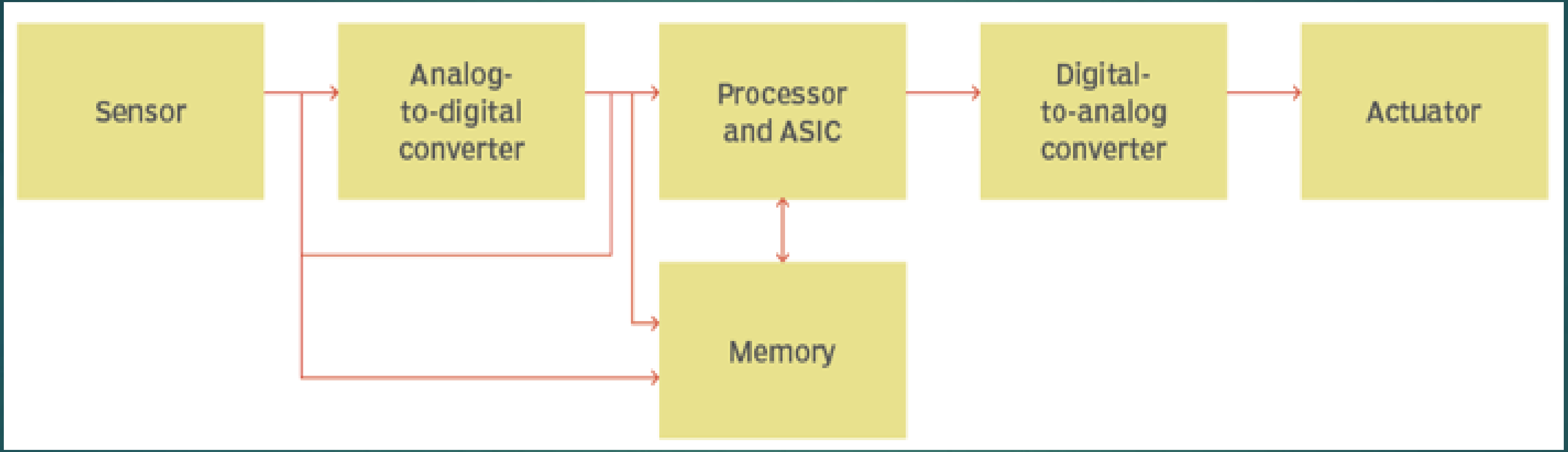


Figure1.1: A diagram of the basic structure and flow of information in embedded systems.

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.

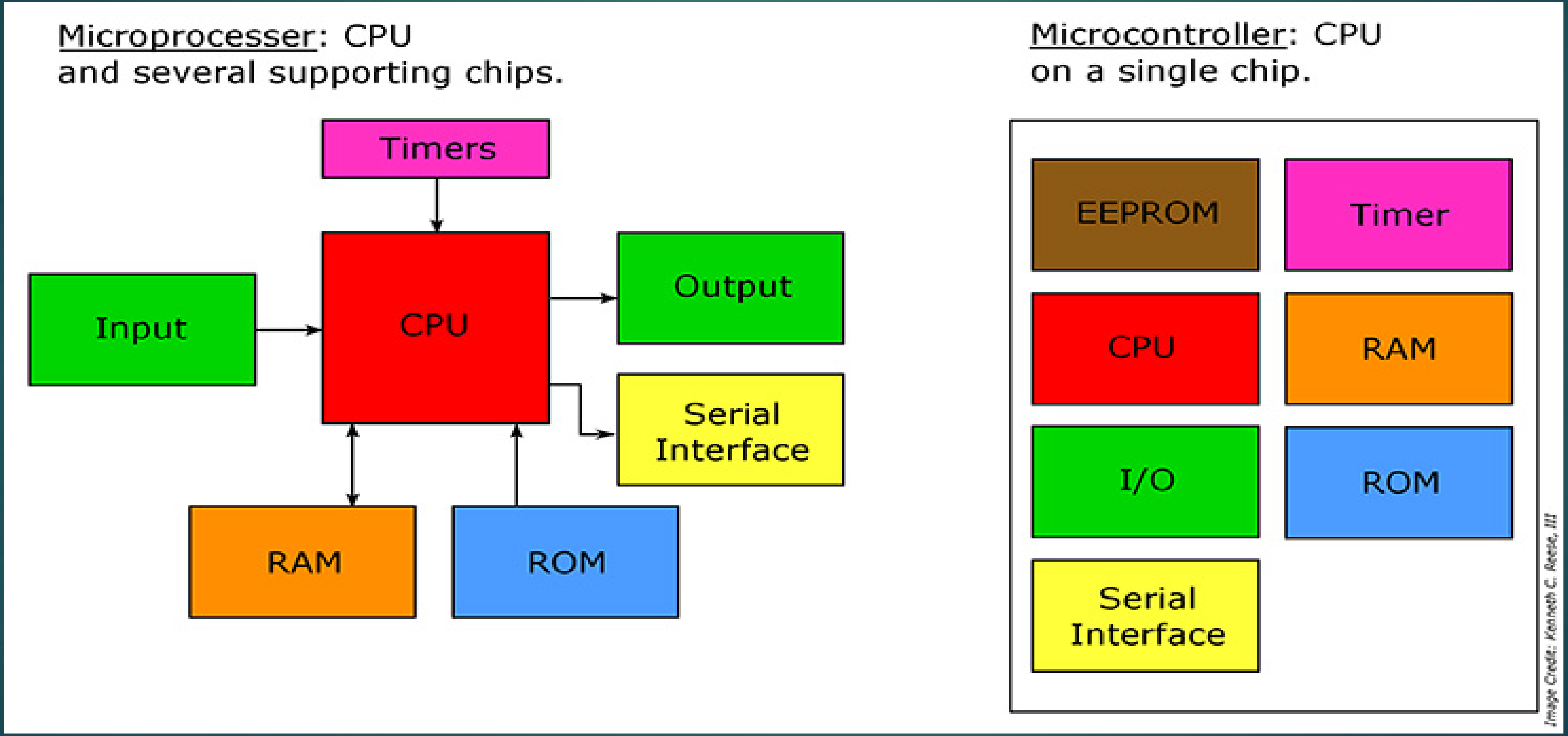
Application Domains of Embedded Systems **(But not limited to these)**

- **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.

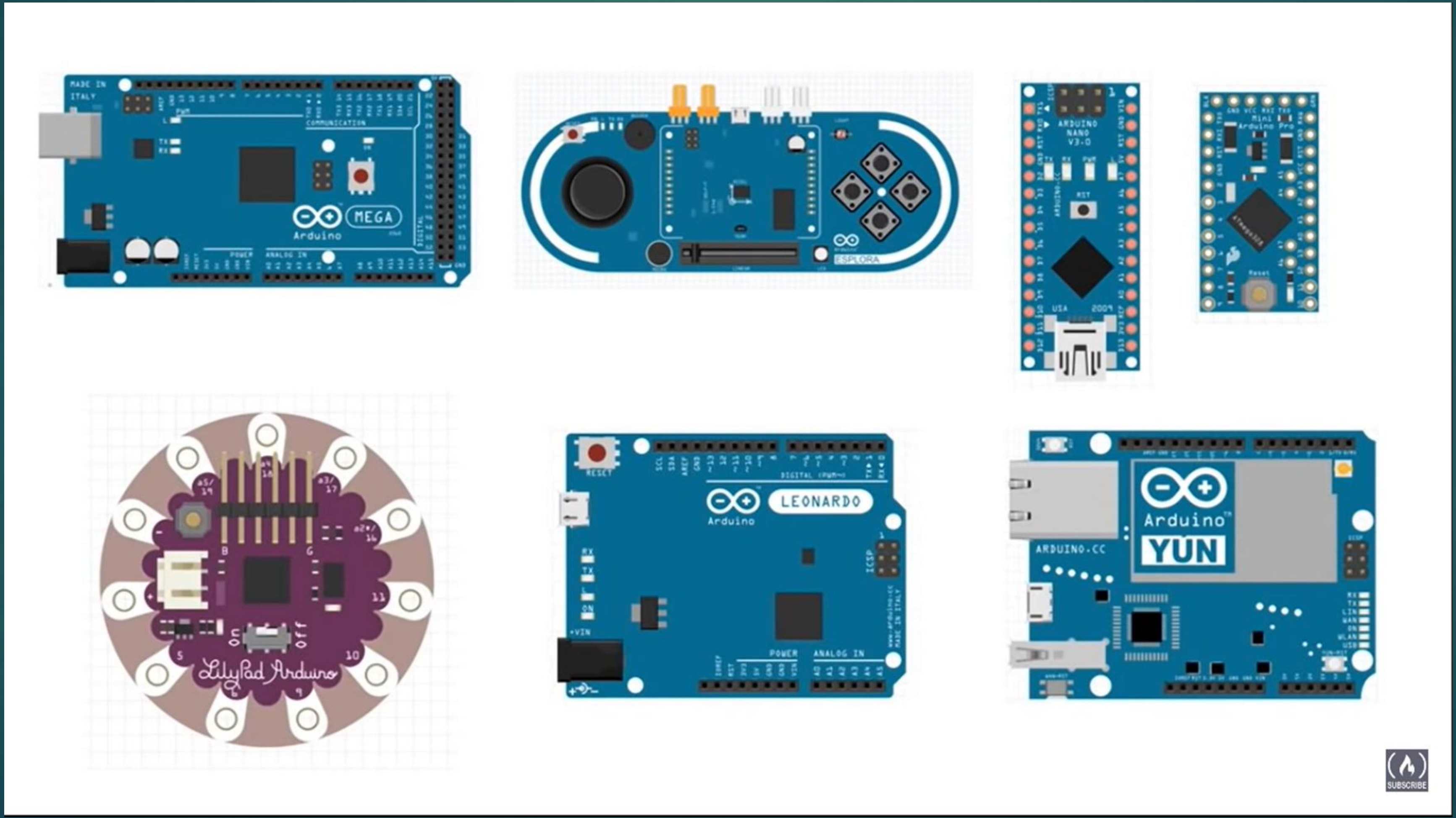
Key features and examples of microcontrollers used in Arduino boards

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



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.

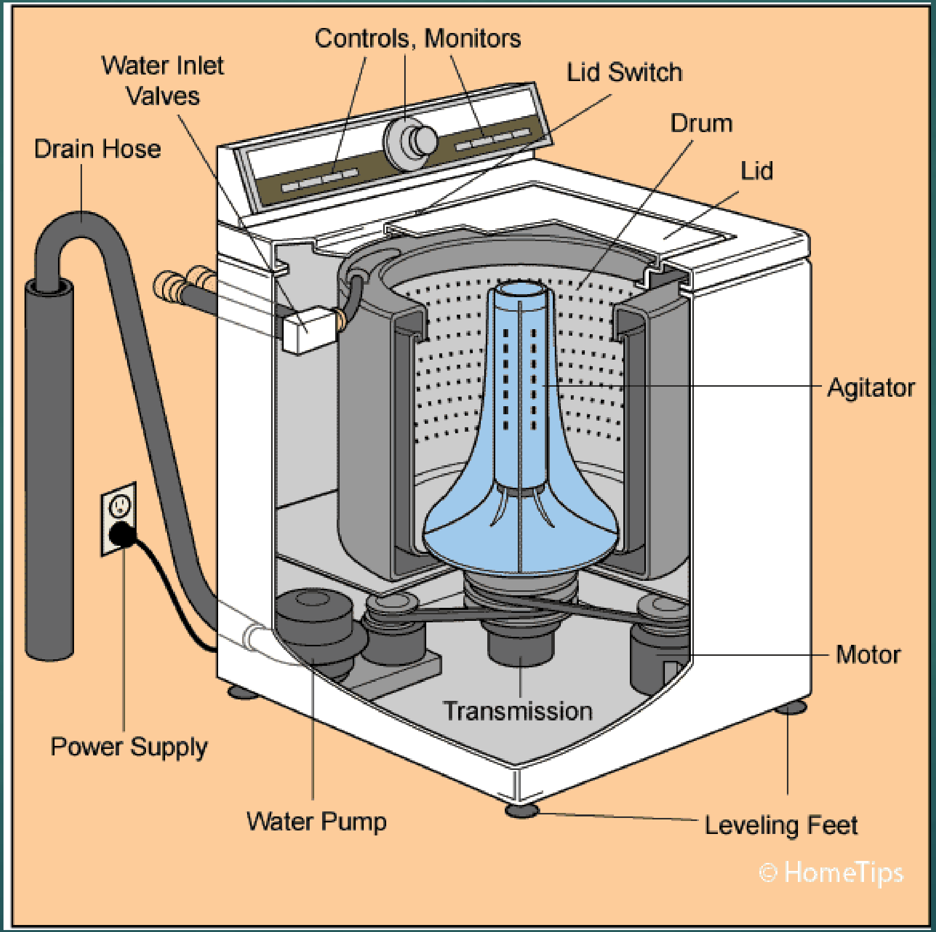
Reset

- 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 and preventing any lingering effects from previous operations.

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.

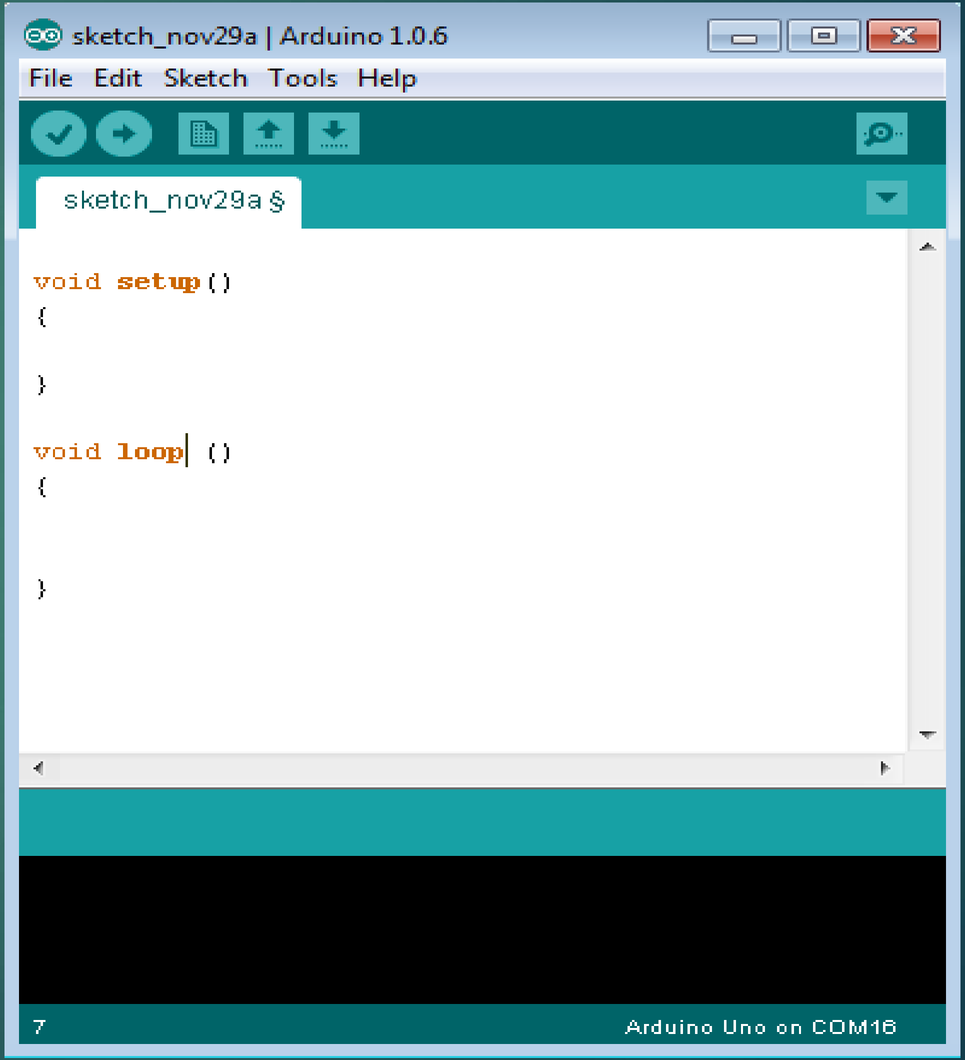
- **Integrated Examples:** Arduino IDE comes with a set of pre-built examples that demonstrate various functionalities of the Arduino board and its peripherals. These examples can be accessed from the IDE's menu and serve as a starting point for learning and building your own projects.

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.

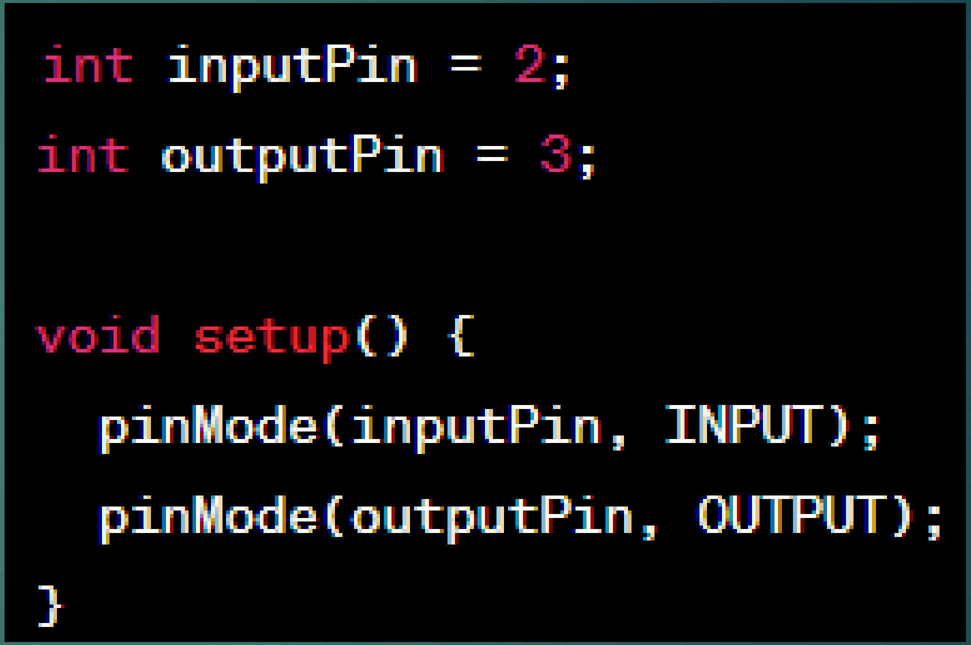


```
void setup()
{

}

void loop ()
{


}
```

```
7                                   Arduino Uno on COM16
```

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.

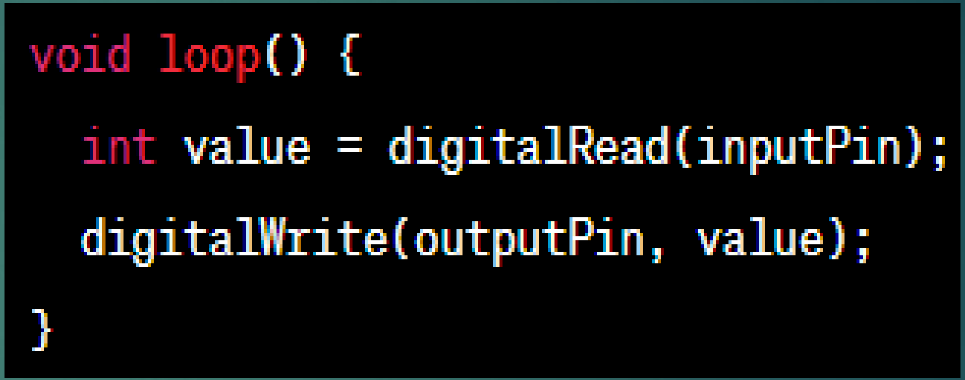


```
int inputPin = 2;
int outputPin = 3;

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

Digital Input/Output:

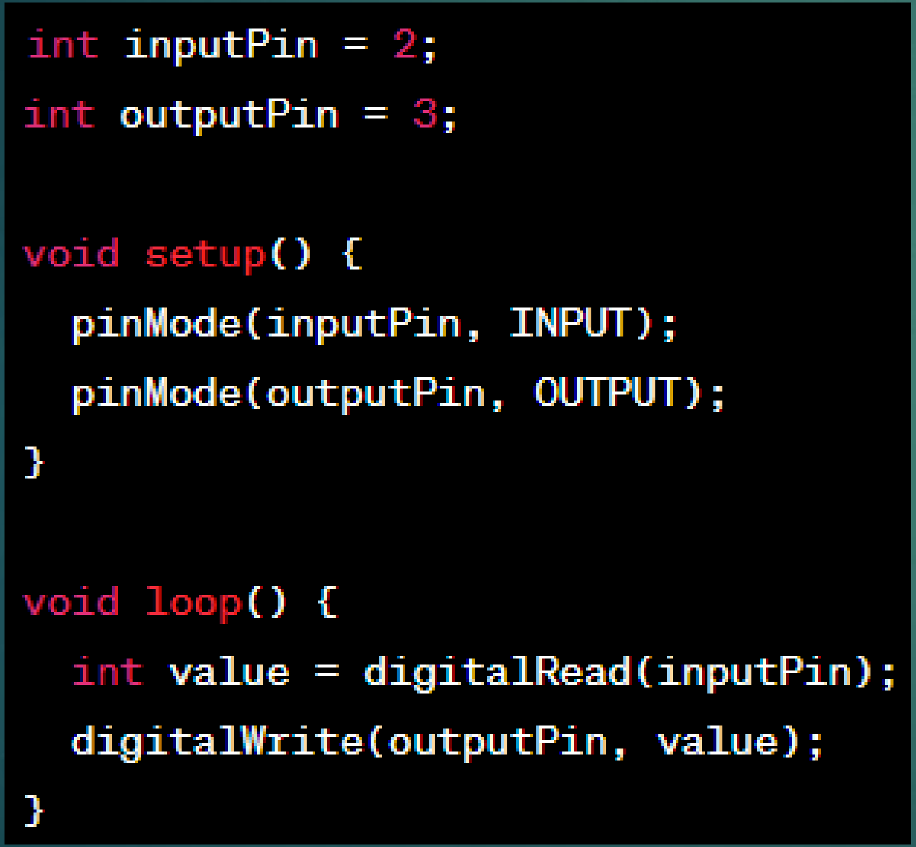
- To read from a digital pin, you can use the digitalRead() function, which returns either HIGH or LOW.

- 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:

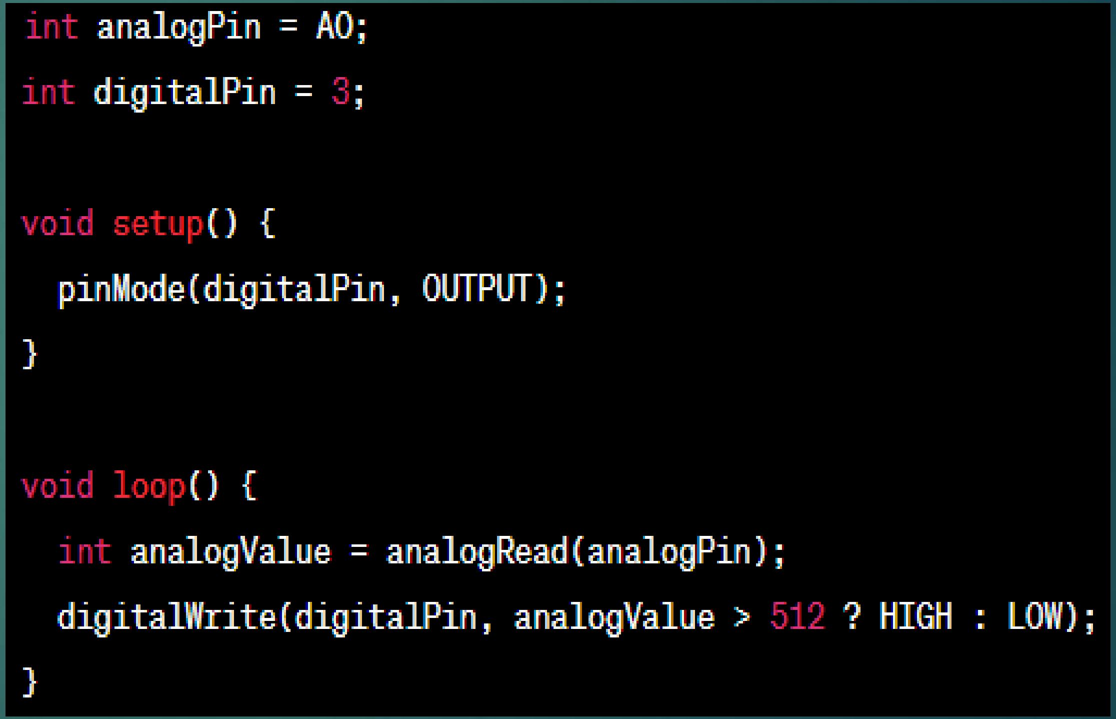


```
int inputPin = 2;
int outputPin = 3;

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

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

Summary:

Analog Input:

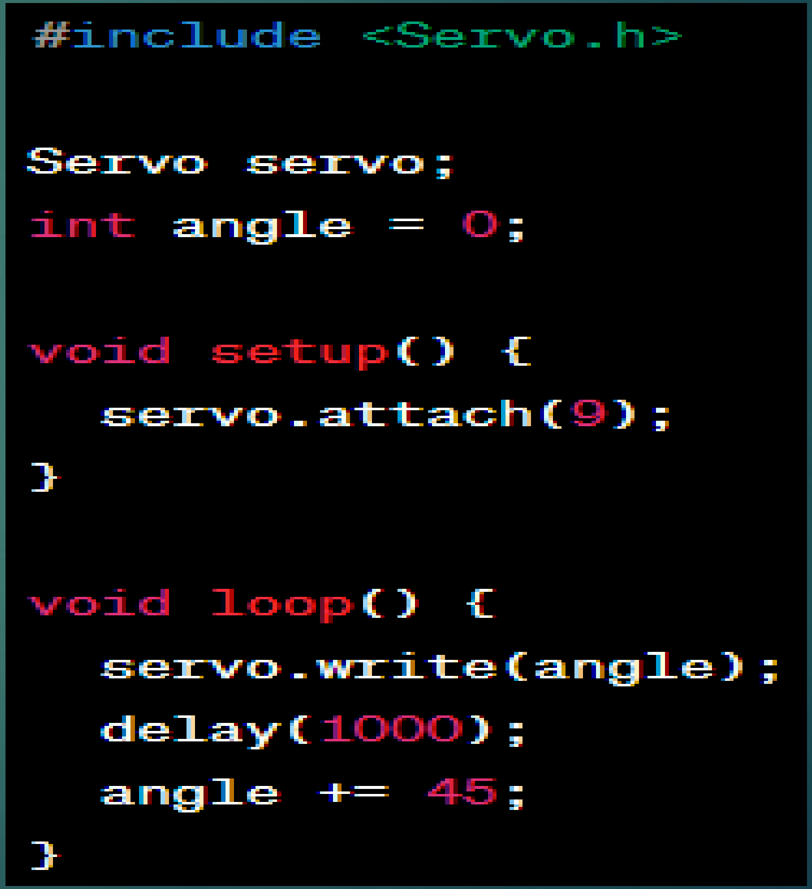
- To read analog values from an analog input pin, use the analogRead() function. It returns a value between 0 and 1023.
- For example, to read from analog pin A0 and write the value to a digital pin:



```
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:

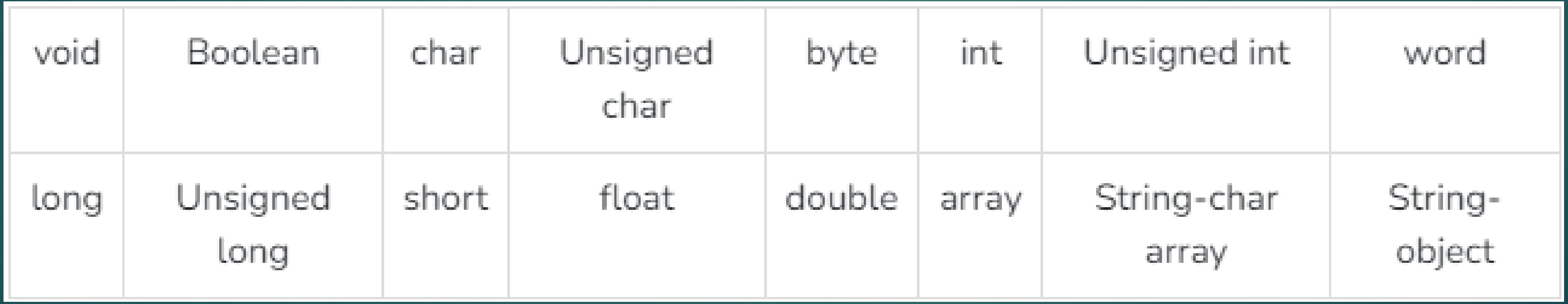
- Arduino provides various built-in functions and libraries that simplify common tasks.
- Examples include delay() for introducing delays, Serial library for serial communication, and libraries for specific sensors or devices.
- Libraries can be included using the #include directive at the beginning 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

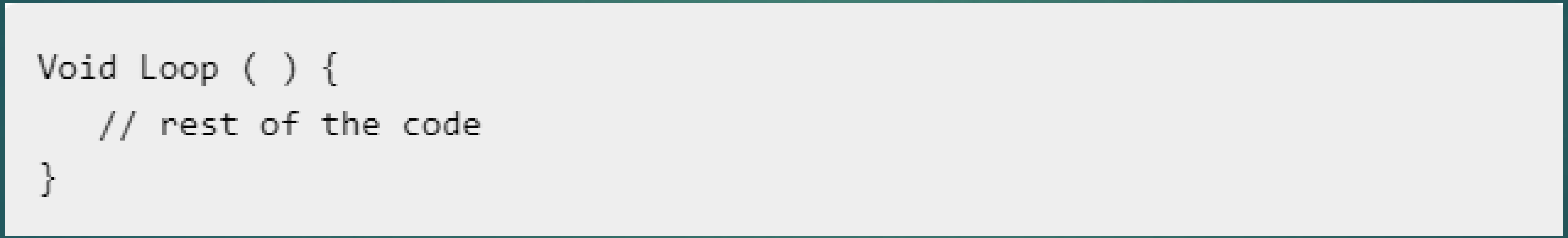
- Data types in C refers to an extensive system used for declaring variables or functions of different types. The type of a variable determines how much space it occupies in the storage and how the bit pattern stored is interpreted.
- The following table provides all the data types that you will use during Arduino programming.



void	Boolean	char	Unsigned char	byte	int	Unsigned int	word
long	Unsigned long	short	float	double	array	String-char array	String-object

void

- The void keyword is used only in function declarations.
- It indicates that the function is expected to return no information to the function from which it was called.



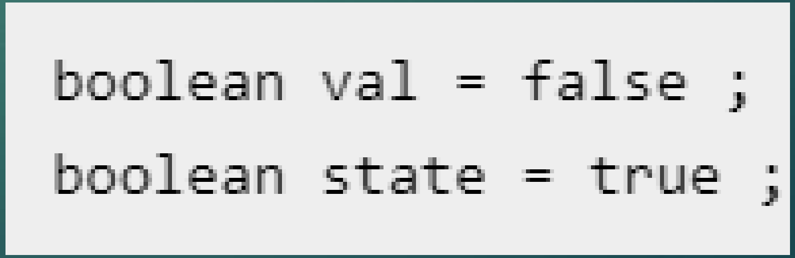
```
Void Loop ( ) {
   // rest of the code
}
```

Boolean

- A Boolean holds one of two values, true or false.
- Each Boolean variable occupies one byte of memory.

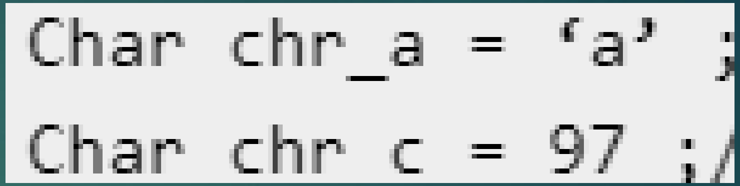
boolean val = false ; // declaration of variable ***"val"*** with type boolean and initialize it with false

boolean state = true ; // declaration of variable ***"state"*** with type boolean and initialize it with true



```
boolean val = false ;
boolean state = true ;
```

Char/Signed Char



```
Char chr_a = ‘a’ ;
Char chr c = 97 ;/
```

- A data type that takes up one byte of memory that stores a character value.
- Character literals are written in single quotes like this: ***'A'*** and for multiple characters, strings use double quotes: ***"ABC“.***
- The range of char is typically from -128 to 127.
- When used for arithmetic operations, char behaves as a signed data type.

Char chr_a = ‘a’ ;//declaration of variable “***char_a”*** with type char and initialize it with character a.

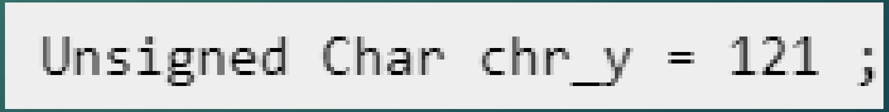
Char chr_c = 97 ;//declaration of variable ”***chr_c”*** with type char and initialize it with character 97.

Unsigned chr_h= -97 ;//declaration of variable ”***chr_h”*** with type signed char and initialize it with numeric number -97.

unsigned char

- It is also a 1-byte (8-bit) data type.
- It is explicitly specified as an unsigned data type, meaning it can only represent positive values.
- The range of unsigned char is typically from 0 to 255.
- When used for arithmetic operations, unsigned char behaves as an unsigned data type, allowing only non-negative results.

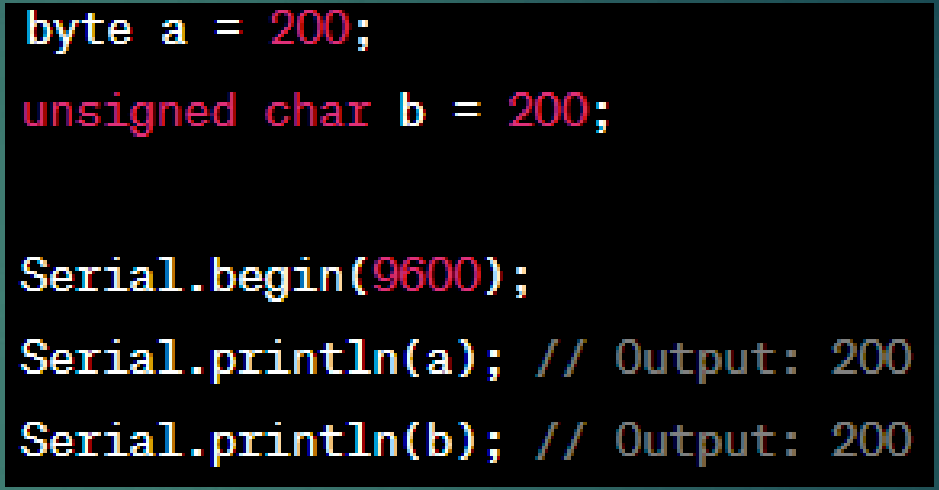
Unsigned Char chr_y = 121 ; // declaration of variable ***chr_y*** with type Unsigned char and initialize it with non negative number 121.



```
Unsigned Char chr_y = 121 ;
```

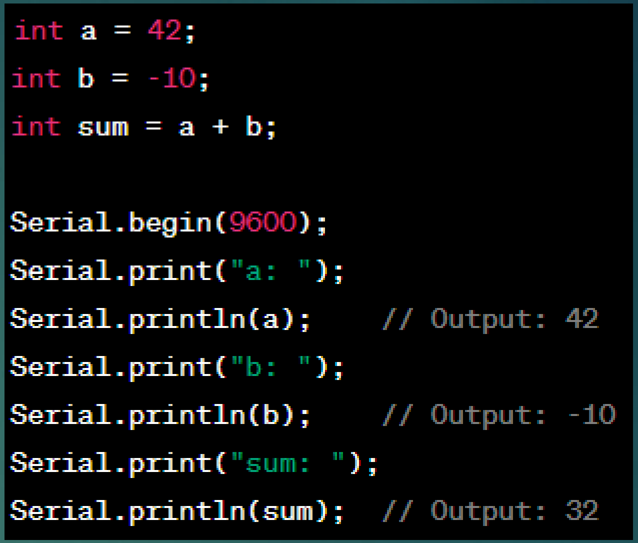
byte

- byte is an unsigned 8-bit data type.
- It can store values from 0 to 255.
- byte is specifically defined in the Arduino language and is often used to represent raw binary data or when you need to work with individual bits.



```
byte a = 200;
unsigned char b = 200;


Serial.begin(9600);
Serial.println(a); // Output: 200
Serial.println(b); // Output: 200
```

int



```
int a = 42;
int b = -10;
int sum = a + b;

Serial.begin(9600);
Serial.print("a: ");
Serial.println(a);    // Output: 42
Serial.print("b: ");
Serial.println(b);    // Output: -10
Serial.print("sum: ");
Serial.println(sum);  // Output: 32
```

Size and Range:

- The int data type is a signed 16-bit integer.
- It can store values from -32,768 to 32,767.
- The exact size of an int may vary depending on the Arduino board you are using. In most cases, it is a 16-bit signed integer.

Usage:

- The int data type is commonly used when you need to work with integer values within the range mentioned above.
- It is suitable for most general-purpose integer calculations and variables.
- int is the default data type for integer literals unless explicitly specified otherwise.

Signedness:

- The int data type is signed, which means it can represent both positive and negative values.
- If you need to work with only positive values, you can use the unsigned int data type, which extends the range to 0 to 65,535.Memory Usage:
- The int data type requires 2 bytes (16 bits) of memory to store the value.

Unsigned int

Unsigned int data type is used to store unsigned integer values. Here are some key points about the unsigned int data type:

Size and Range:

- The unsigned int data type is an unsigned 16-bit integer. It can store values from 0 to 65,535.
- Like the int data type, the size of an unsigned int may vary depending on the Arduino board you are using. In most cases, it is a 16-bit unsigned integer.

Usage:

- The unsigned int data type is commonly used when you only need to work with positive integer values within the specified range.
- It is suitable for scenarios where negative values are not required, such as representing sensor readings, loop counters, or array indices.

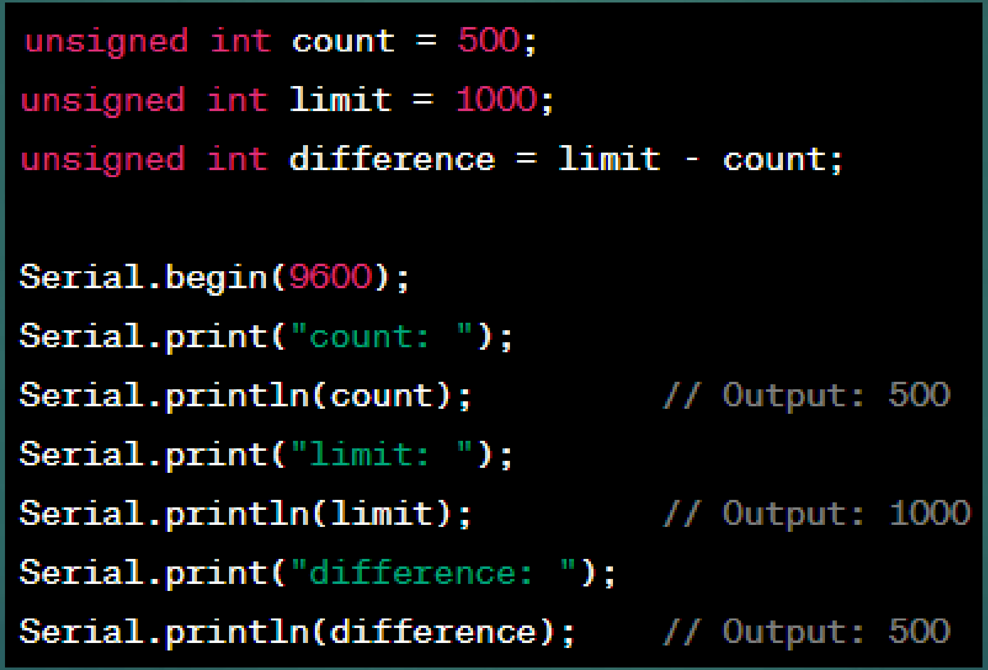
Signedness:

- Unlike the int data type, unsigned int is an unsigned data type and can only represent non-negative values.It cannot store negative values.
- If you need to work with both positive and negative values, you should use the int data type instead.

Memory Usage:

- The unsigned int data type requires 2 bytes (16 bits) of memory to store the value, similar to int.

Unsigned int (Cnt.)



```
unsigned int count = 500;
unsigned int limit = 1000;
unsigned int difference = limit - count;

Serial.begin(9600);
Serial.print("count: ");
Serial.println(count);          // Output: 500
Serial.print("limit: ");
Serial.println(limit);          // Output: 1000
Serial.print("difference: ");
Serial.println(difference);     // Output: 500
```

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

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

void setup(): This is the setup function in Arduino, which is called once when the board starts or is reset.

Serial.begin(9600);: This line initializes the serial communication at a baud rate of 9600. It allows communication between the Arduino board and a computer through the Serial Monitor.

```
void loop() {
  int num1, num2;
  char operatorSymbol;
```

void loop(): This is the main loop function in Arduino, which runs repeatedly after the setup() function.

int num1, num2;: These are integer variables to store the two numbers entered by the user.

char operatorSymbol;: This is a character variable to store the arithmetic operator entered by the user.

```
Serial.println("Enter the first number:");
while(!Serial.available());
num1 = Serial.parseInt();
```

Serial.println("Enter the first number:");: This line prints a message to the Serial Monitor asking the user to enter the first number.

while(!Serial.available());: This is a while loop that waits until data is available on the Serial port (until the user enters something and sends it to the Arduino).

num1 = Serial.parseInt();: This line reads the number entered by the user and stores it in the variable num1.

```
Serial.println("Enter an arithmetic operator (+, -, *, /):");
while(!Serial.available());
operatorSymbol = Serial.read();
Serial.println("Enter the second number:");
while(!Serial.available());
num2 = Serial.parseInt();
```

int result;: This is an integer variable to store the result of the arithmetic operation.

switch (operatorSymbol) { ... }: This is a switch statement that checks the value of operatorSymbol, which was entered by the user, and performs the corresponding arithmetic operation.

case '+': result = num1 + num2; break;: If the user entered '+', the code inside this case block adds num1 and num2 and stores the result in result.

case '-': result = num1 - num2; break;: If the user entered '-', the code inside this case block subtracts num2 from num1 and stores the result in result.

case '*': result = num1 * num2; break;: If the user entered '*', the code inside this case block multiplies num1 and num2 and stores the result in result.

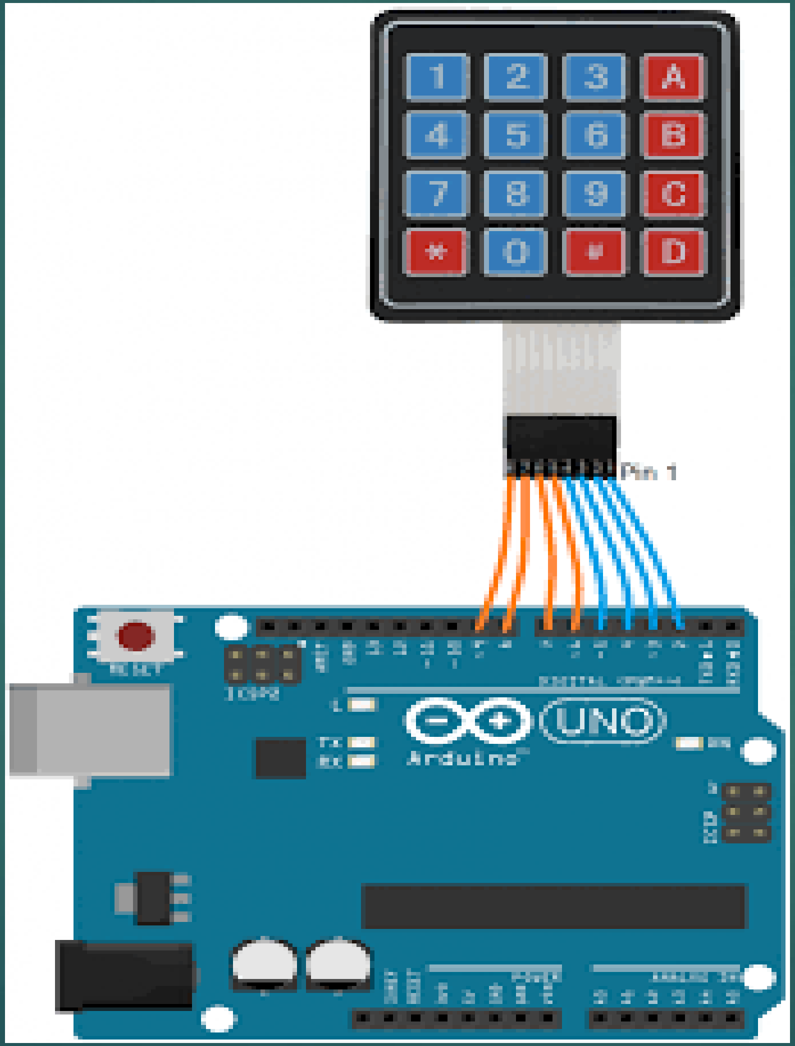
case '/': result = num1 / num2; break;: If the user entered '/', the code inside this case block divides num1 by num2 and stores the result in result.

default: Serial.println("Invalid operator!"); return;: If the user entered an invalid operator, the code inside this default block prints an error message to the Serial Monitor and exits the loop.

Serial.print("Result: ");: This line prints the text "Result: " to the Serial Monitor.

Serial.println(result);: This line prints the value of the result variable (the calculated result) to the Serial Monitor.

4X4 Keypad with Arduino



To create a simple calculator using an Arduino and a 4x4 hex keypad, you'll need to connect the keypad to the Arduino and write a program to read input from the keypad and perform basic arithmetic operations.

Components needed:

- Arduino board (e.g., Arduino UNO)
- 4x4 hex keypad
- Jumper wires

Wiring connections:

- **Connect the 4x4 hex keypad to the Arduino as follows:**
  - **Connect ROW0 to digital pin 2**
  - **Connect ROW1 to digital pin 3**
  - **Connect ROW2 to digital pin 4**
  - **Connect ROW3 to digital pin 5**
  - **Connect COL0 to digital pin 6**
  - **Connect COL1 to digital pin 7**
  - **Connect COL2 to digital pin 8**
  - **Connect COL3 to digital pin 9**

```
#include <Keypad.h>
```

The program uses the Keypad library to read input from the 4x4 hex keypad. When you press an operator (+, -, *, /) or the equals (=) key, it will perform the corresponding operation and display the result on the Serial Monitor.

Make sure to install the Keypad library in the Arduino IDE by going to "Sketch" -> "Include Library" -> "Manage Libraries" and then search for "Keypad" and install it.

//define the number of rows and columns of the keypad

```
const byte ROWS = 4;
const byte COLS = 4;
```

// Define the keypad layout, It maps each button's value to its corresponding position on the keypad. It represents a 4x4 matrix where each cell contains a character representing the button's value.

```
char keys[ROWS][COLS] =
{
  {'1', '2', '3', 'A'},
  {'4', '5', '6', 'B'},
  {'7', '8', '9', 'C'},
  {'*', '0', '#', 'D'}
};
```

```
byte rowPins[ROWS] = {2, 3, 4, 5};

byte colPins[COLS] = {6, 7, 8, 9};
```

//These arrays define the connections of the keypad to the Arduino. The rowPins array contains the Arduino's digital pins connected to the keypad's rows, and the colPins array contains the Arduino's digital pins connected to the keypad's columns.

```
Keypad keypad = Keypad(makeKeymap(keys), rowPins, colPins, ROWS, COLS);
```

//This line initializes the Keypad object keypad with the defined keys array and the pin connections rowPins and colPins. It also specifies the number of rows and columns in the keypad.

```
void setup() { Serial.begin(9600); }
void loop() {
  char key = keypad.getKey();
  if (key) {
    handleKeyPress(key);
  }
}
```

It reads a character from the keypad using the getKey() function. If a key is pressed (the key variable is not NULL), it calls the handleKeyPress function to handle the keypress.

```
void handleKeyPress(char key) {
  if (key == 'A' || key == 'B' || key == 'C' ||
key == 'D') {
    // Handle operator keys (+, -, *, /)
    operatorSymbol = key;
    operatorPressed = true;
    operand1 = inputStr.toDouble();
    inputStr = "";
  } else if (key == '#') {
    // Handle the equals key (=)
    operand2 = inputStr.toDouble();
    performOperation();
    inputStr = String(operand1);
  } else {
    // Append the digit to the input string
    inputStr += key;
  }
  Serial.println(inputStr);
}
```

The handleKeyPress function processes the key pressed by the user. If the key is an operator key (A, B, C, D representing +, -, *, /), it sets the operatorSymbol, indicates that an operator is pressed (operatorPressed), converts the current input string to the first operand (operand1), and clears the inputStr.

If the key is the equals key (#), it converts the current input string to the second operand (operand2), calls the performOperation function to execute the arithmetic operation, and stores the result back into the inputStr.

If the key is a digit, it appends the digit to the inputStr.

```
void performOperation() {
 switch (operatorSymbol) {
  case 'A':
   operand1 += operand2;
   break;
  case 'B':
   operand1 -= operand2;
   break;
  case 'C':
   operand1 *= operand2;
   break;
  case 'D':
   operand1 /= operand2;
   break;
 }
 operatorPressed = false;
}
```

Finally, it prints the inputStr to the Serial Monitor.

The performOperation function is called when the equals key (#) is pressed. It performs the arithmetic operation based on the operatorSymbol value and updates the operand1 variable with the result. It then resets the operatorPressed flag to false.

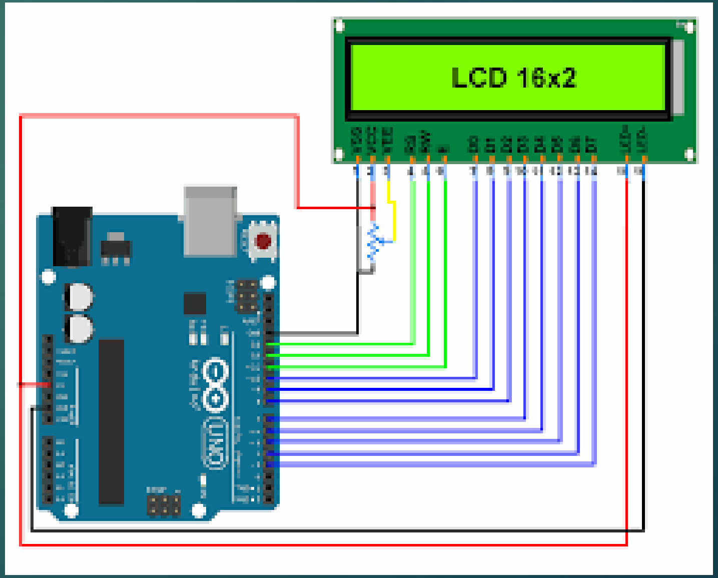
Interfacing 16X2 LCD Display

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.

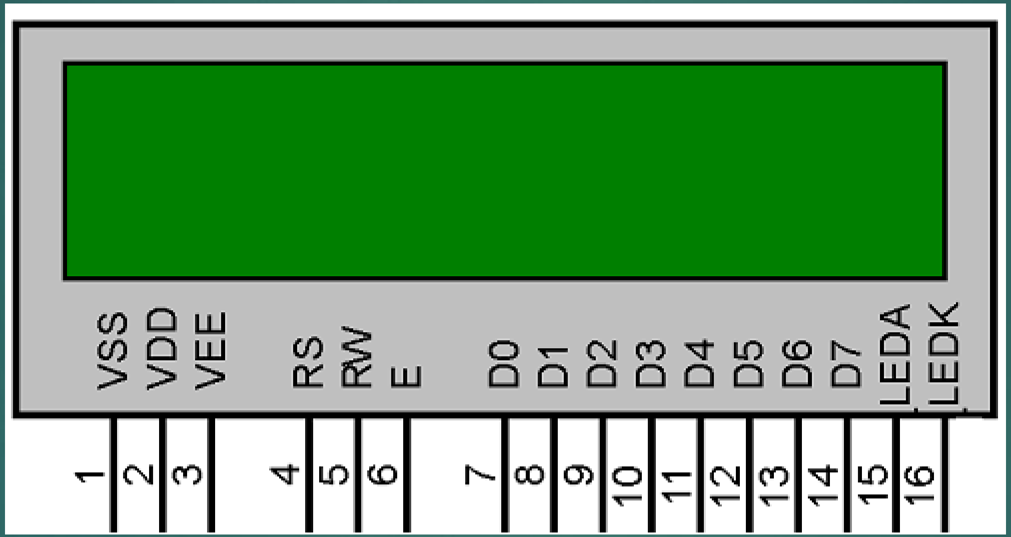
Below is a step-by-step explanation of how to interface an LCD with an Arduino Uno:

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
}
```

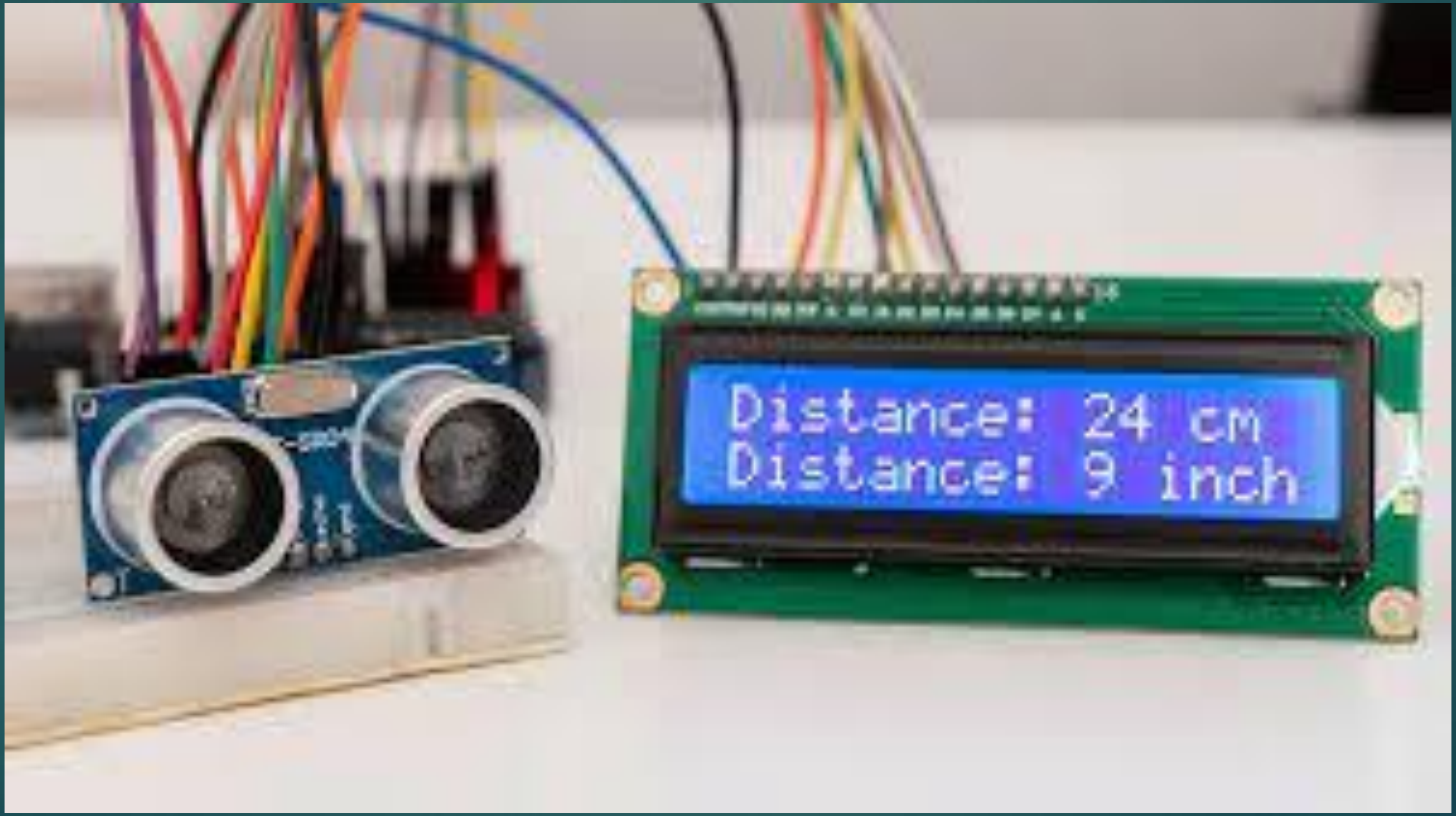
LED Brightness Control with Potentiometer

```
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.

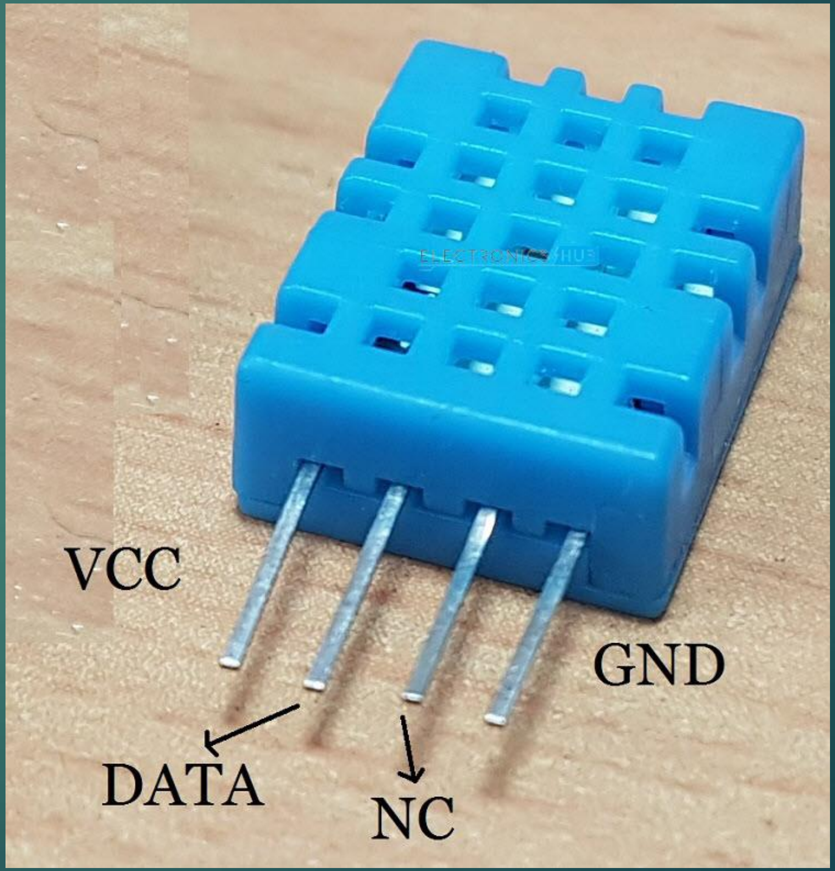
Step-by-step procedure to interface the HC-SR04 ultrasonic distance sensor with an Arduino:

- **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:

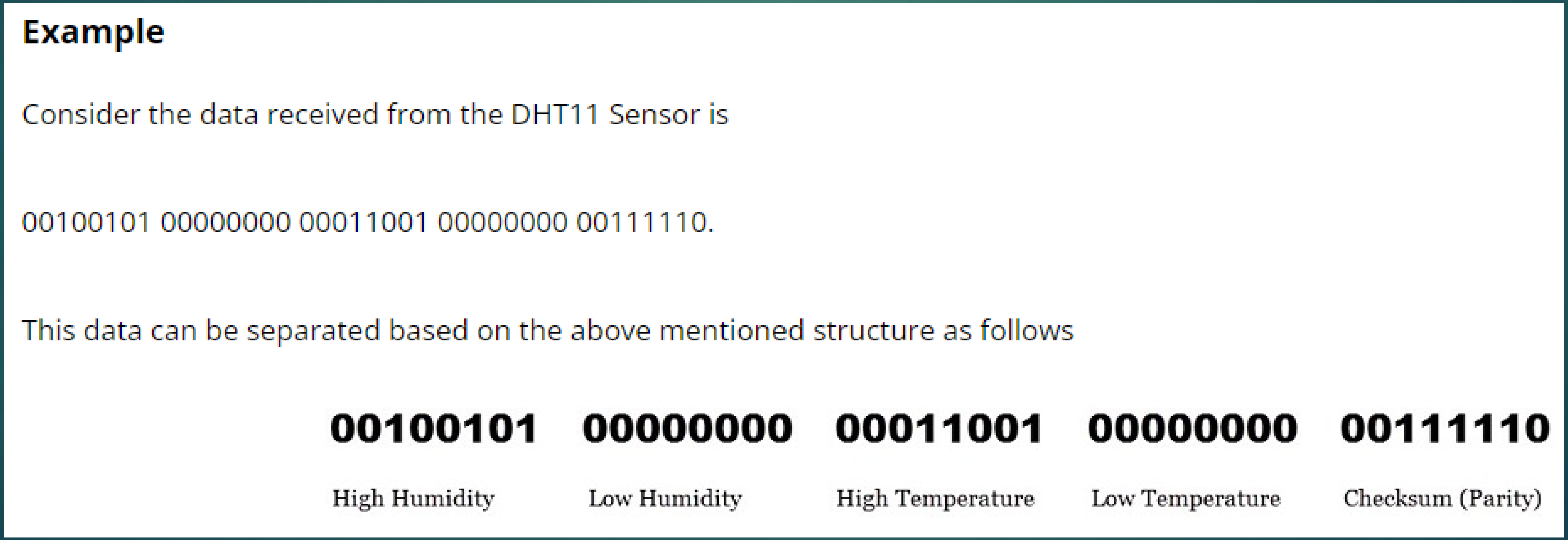
- The temperature sensing element in the DHT11 is a thermistor—a type of resistor whose resistance changes with temperature.
- The thermistor's resistance decreases as the temperature increases and vice versa.
- The sensor circuitry measures this resistance change and converts it into a digital temperature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

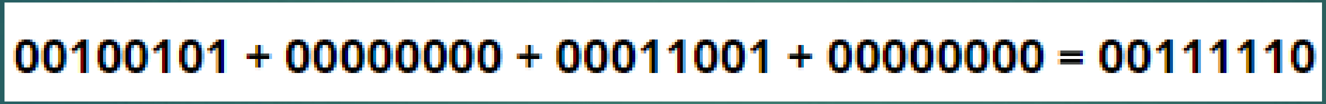
00100101 00000000 00011001 00000000 00111110.

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

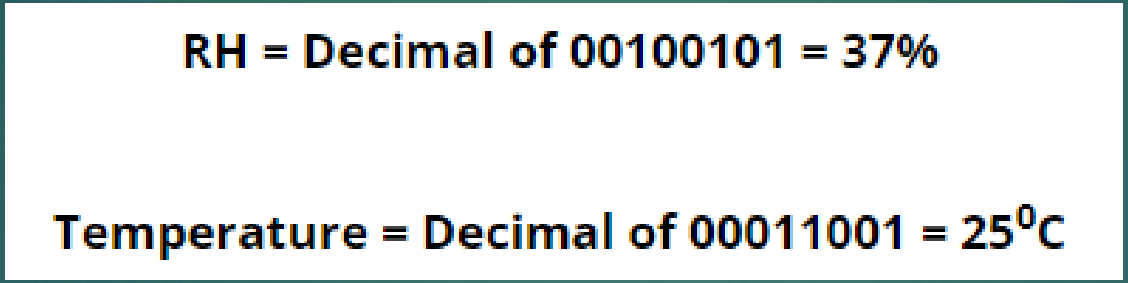
- In order to check whether the received data is correct or not, we need to perform a small calculation.
- Add all the integral and decimals values of RH and Temperature and check whether the sum is equal to the checksum value i.e. the last 8 – bit data.



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^{0}C

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

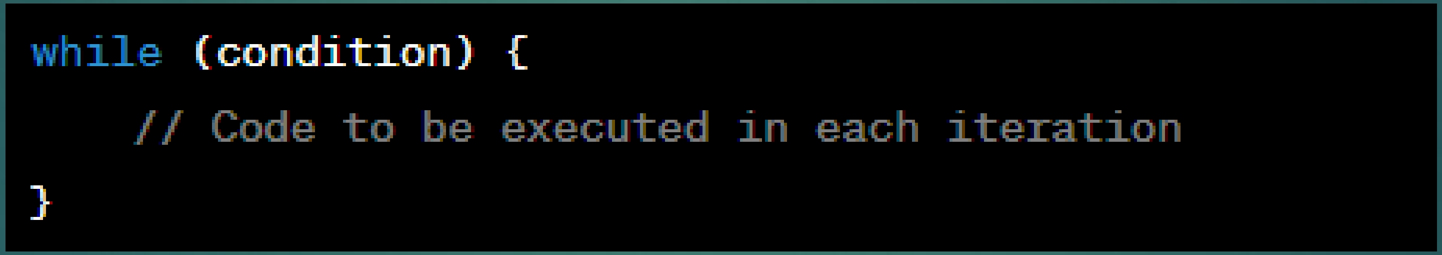
```
for (initialization; condition; increment) {
    // Code to be executed in each iteration
}
```

- 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
}
```

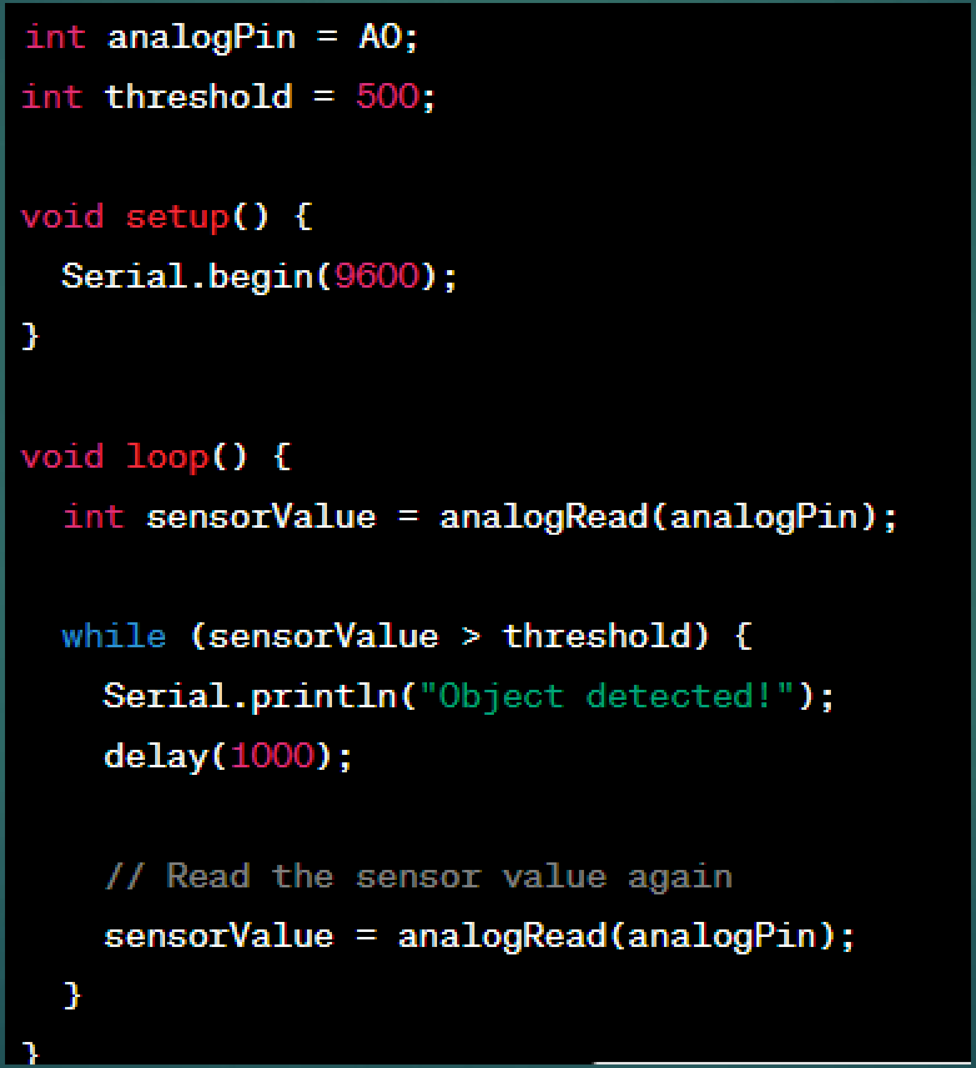
while Loop

- 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);
  }
}
```