Edge Devices

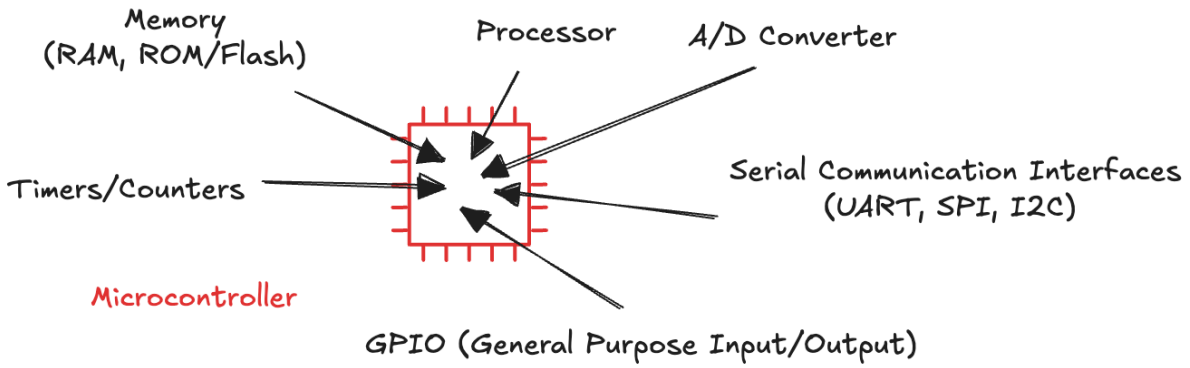
Prof. Riccardo Berta

2025-03-20

Contents

1 Edge Devices

In the context of fog computing, an **edge device** refers to a computing unit located at or near the point where data is generated. This close proximity enables **faster data processing** and **minimizes latency**, which is crucial for applications requiring real-time responses. The core component within edge devices is the **microcontroller** a compact, specialized integrated circuit designed to handle specific tasks in embedded systems. Microcontrollers integrate a processor core, memory, and input/output peripherals on a single chip, making them ideal for lightweight, task-focused operations:



Microcontrollers are often combined with **development boards** to simplify prototyping, testing, and

developing embedded systems. While a standalone microcontroller chip provides the essential computing capabilities, it typically requires **additional components** and configurations to function effectively in real-world applications. Development boards address this need by providing a **ready-to-use platform** that integrates the microcontroller with other necessary elements, such as leds, buttons, voltage regulators, and sometimes even displays, sensors and communication modules (e.g., Wi-Fi or Bluetooth). These features allow developers to **focus on the application logic** rather than building basic hardware. Moreover, development boards are often supported by **software development environments** and **libraries** that simplify coding and reduce development time. For example:

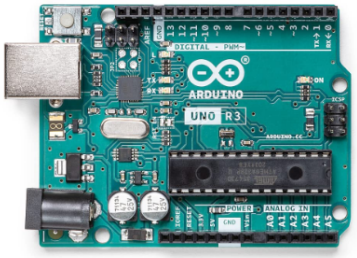
- Arduino boards come with a large collection of libraries for various hardware components,
- STM32 development boards are supported by software tools, easing the configuration of peripherals.

It' important to highlight the difference between a microcontroller and a microprocessor. A microcontroller is a self-contained system designed to perform specific tasks efficiently and repeatedly. The high level of integration reduces the need for external components, making microcontrollers compact, cost-effective, and energy-efficient. Microcontrollers excel at performing **a single dedicated function**, such as controlling a microwave, thermostat, or radio. They typically run one program stored in ROM, repeating it consistently without change. A microprocessor, on the other hand, is primarily a central processing unit (CPU) and lacks integrated components like memory and I/O peripherals. It relies on external modules such as RAM, ROM, storage, and I/O controllers to function, which makes it **more versatile** but also more complex. Microprocessors are typically used in systems requiring flexibility and the ability to handle large, multifaceted programs, such as personal computers, servers, or gaming consoles. The distinction between microcontrollers and microprocessors is becoming less clear, especially with the advent of advanced processors like ARM-based systems. Many ARM processors (e.g., 32-bit Cortex-M series) combine features of both, offering the simplicity of microcontrollers with the power of microprocessors. These hybrid systems are used in applications that demand both real-time processing and computational power, such as smartphones.

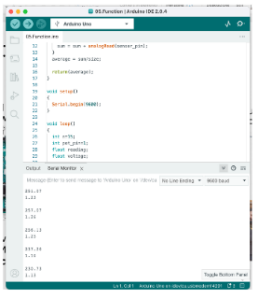
1.1 Arduino

Arduino is an **open-source electronics platform** designed to provide accessible and easy-to-use hardware and software for creating electronic projects. It was introduced in 2005 at the Interaction Design Institute in Italy, where it was conceived as an affordable and user-friendly tool for students exploring electronics. Since its inception, Arduino has become a widely recognized platform, empowering individuals to prototype and build innovative devices.

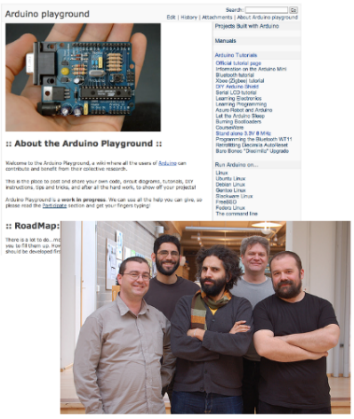
A physical piece of hardware



A programming environment

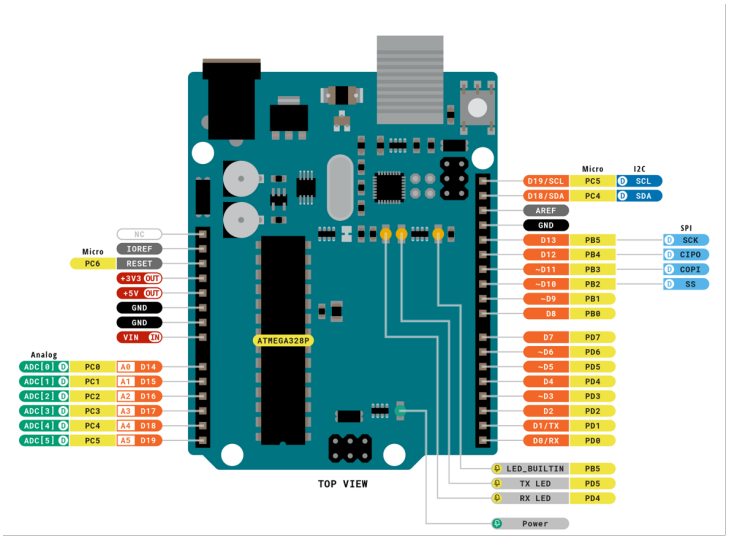


A community



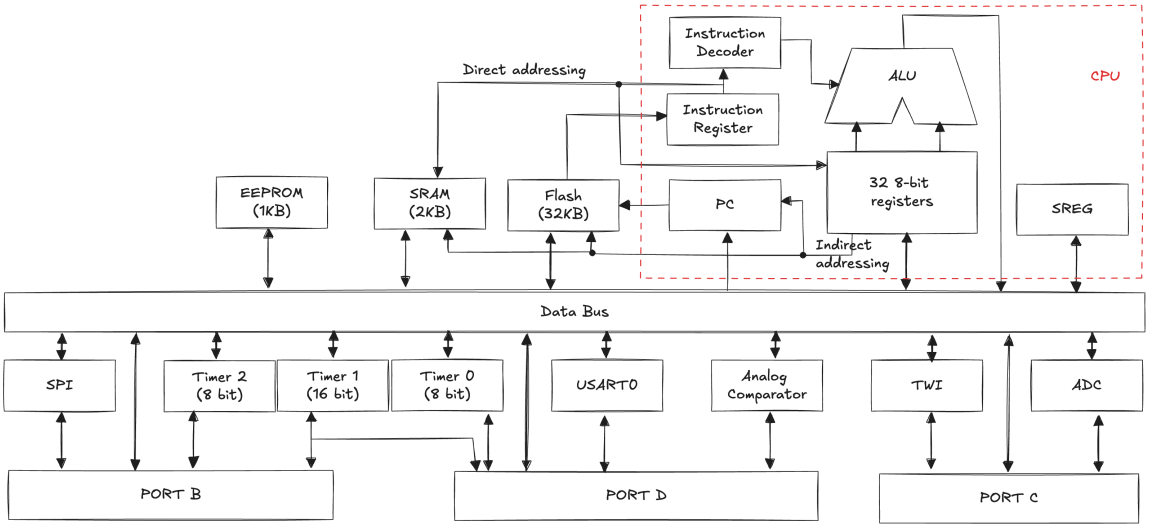
At the core of Arduino is its hardware, a compact printed circuit board that houses a **microcontroller**, alongside essential components to support its operation. The platform also includes **intuitive software**. The Arduino IDE allows users to write, compile, and upload code to the board effortlessly. The programming language, based on C/C++, is straightforward and approachable, even for beginners. A cornerstone of Arduino's success is its **community**, consisting of makers, hobbyists, students, and professionals worldwide. The platform open-source nature encourages collaboration and knowledge sharing, leading to a rich repository of projects, tutorials, and libraries. This openness reflects Arduino philosophy: to make electronics accessible to everyone.

The following image depicts the layout of an **Arduino Uno board** (the simple and basic board) with pins and components:



In general, before we can before we can write code for our edge device, we need to understand how

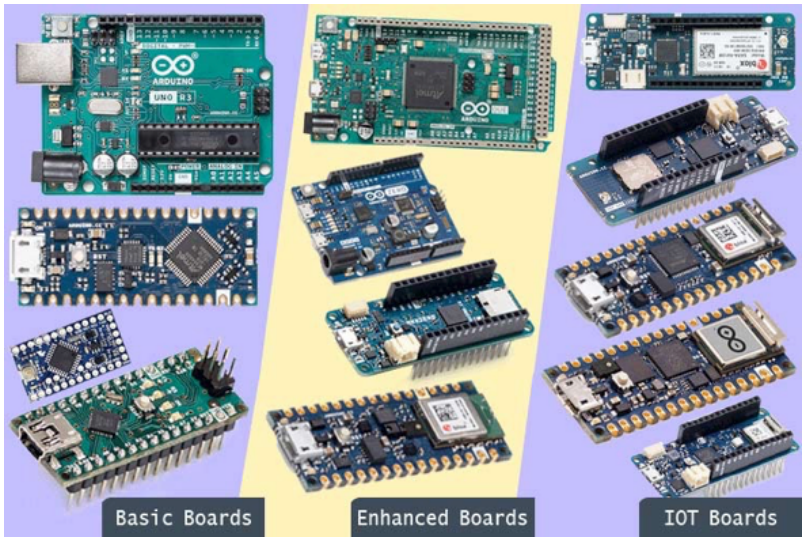
our specific microcontroller is structured and functions. This is typically referred to as its architecture. The schematic of the Arduino shows the **ATMEL ATmega328P microcontroller** at the heart of the development board, which handles the execution of programs (or ”sketches”) uploaded to it. It is a low-power CMOS 8-bit microcontroller based on the Harvard architecture, with physically separate storage and buses for program and data. Note that this is in contrast to the von Neumann architecture which operates with a single storage structure to hold both program and data. While one instruction is being executed, the next instruction is pre-fetched from the program memory. This concept enables instructions to be executed in every clock cycle. A simplified block diagram containing the most pertinent components is the following:



The ATmega328P provides the following significant features. Some memory modules to store programs (32 KBytes flash), to hold run-time variables (2 KBytes SRAM) and to store any data that programs wish to retain after power is cycled (1 KBytes of EEPROM). The microcontroller also includes several general purpose input/output (GPIO) lines: 4 **digital pins** (D0 - D13), some of which can also provide PWM (Pulse Width Modulation) output and 6 **analog input pins** (A0 - A5) for reading analog signals from sensors or other devices. Moreover it has 32 general purpose working registers, three timer/counters with compare modes, internal and external interrupts, an USART, a 2-wire Serial Interface (TWI) port, an SPI serial port, and a 6-channel 10-bit ADC. The 32 8-bit general purpose registers are directly connected to the ALU. Within a single clock cycle, arithmetic operations between general purpose registers or between a register and an immediate are executed. The ALU operations are divided into the three main categories arithmetic, logical, and bit-functions. After an arithmetic operation, the status register (SREG) is updated to reflect information about the result of the operation. Six of the 32 registers can be used as three 16-bit indirect address register pointers for data space addressing, enabling efficient address calculations. One of the these address pointers can also be used as an address pointer for look up tables in flash program memory. Program flow is provided by conditional and unconditional jump and call instructions, able to directly address the whole address space.

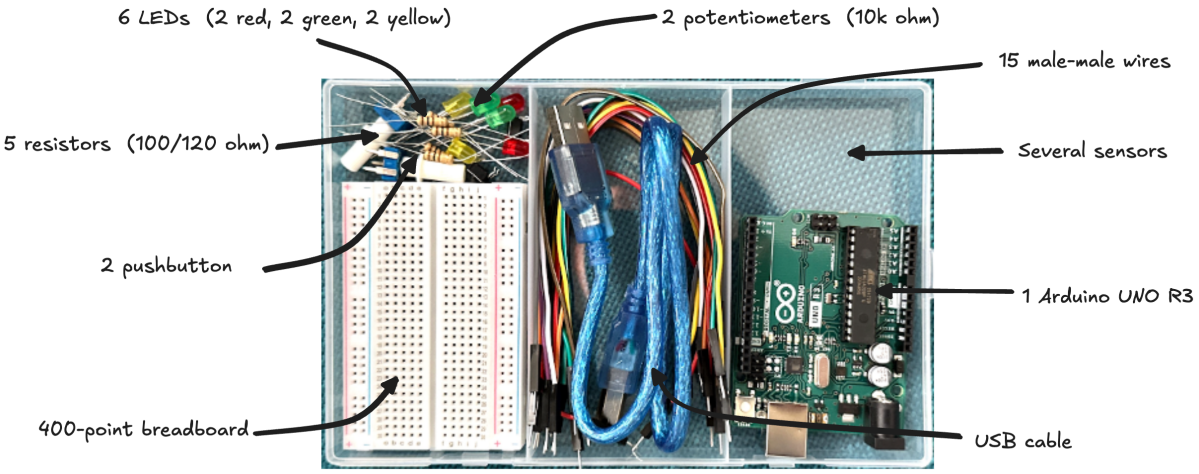
Most instructions have a single 16-bit word format. Every program memory address contains a 16- or 32-bit instruction. During interrupts and subroutine calls, the return address Program Counter (PC) is stored on the stack. The stack is effectively allocated in the general data SRAM, and, consequently, the stack size is only limited by the total SRAM size and the usage of the SRAM. All user programs must initialize the SP in the reset routine before subroutines or interrupts are executed. The Stack Pointer (SP) is read/write accessible in the I/O space. The data SRAM can easily be accessed through the five different addressing modes supported in the architecture. All of the peripheral devices are controlled via sets of specific registers, each of which is connected to the 8-bit data bus. Thus, a program interacts with each peripheral by accessing various memory-mapped registers (via C pointers). Additionally, each peripheral device accesses the off-chip pins via one of the three available ports B, C or D. Each port is configurable via memory-mapped registers to allow different functions access to external pins. The flash memory allows the program memory to be changed via either a SPI serial interface, a conventional non-volatile memory programmer, or an on-chip boot loader running on the processor core. The Arduino boot loader uses the USART configured for RS-232 via the USB-to-serial converter chip on the development board. Software in the boot flash section will continue to run while the application flash section is updated. Finally, the board provides also some **pwer pins** (+3.3V, +5V and GND to provide regulated power for external components), some **status leds** to show the power and activity (including TX and RX for data transmission or reception) and a **reset button** to restart the microcontroller program manually.

Arduino boards come in various models, each tailored to different needs, ranging from basic prototypes to advanced projects requiring wireless communication or extensive input/output capabilities:



1.1.1 Makers Kit

As student of ”Makers”, you can use the ”Makers Kit”, specifically designed to support you in coursework, particularly in introductory of embedded system design. It includes a carefully curated selection of components to enable hands-on learning and experimentation, making it ideal for both classroom activities and individual projects. Here's a detailed overview:

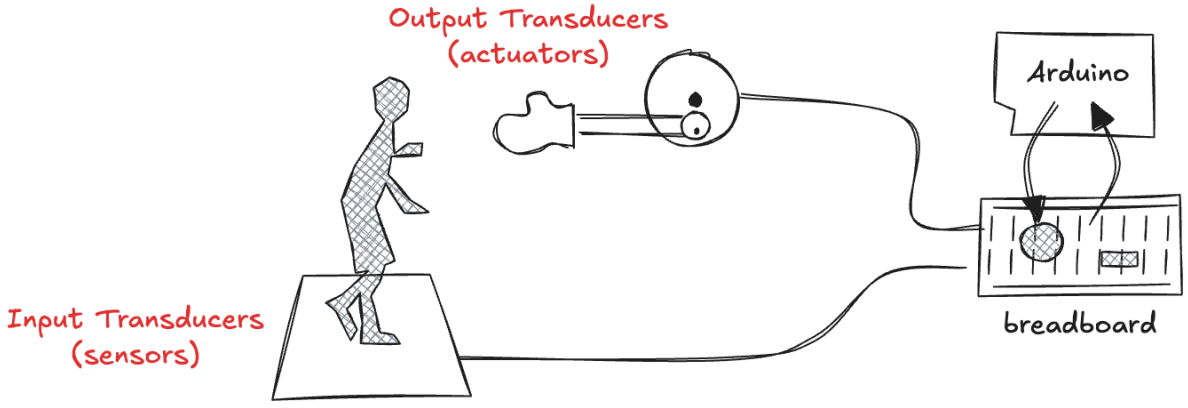


- Arduino UNO R3: a beginner-friendly microcontroller board widely used in educational settings for developing interactive projects
- Several Sensors: these may include basic sensors like temperature, light, or motion detectors, allowing students to explore data collection and environmental interaction
- 6 LEDs (2 red, 2 green, 2 yellow): used for visual feedback in projects, such as signaling or debugging
- 2 Potentiometers (10k ohm): for adjusting values like brightness or resistance in a circuit
- 400-Point Breadboard: a versatile platform for assembling circuits without the need for soldering
- 15 Male-Male Wires: essential for making connections on the breadboard or to the Arduino UNO R3
- 2 Pushbuttons: commonly used for user input in circuits, such as turning on/off devices or triggering actions
- 5 Resistors (100/120 ohm): protect LEDs or other components by limiting the current in the circuit
- USB Cable: enables programming of the Arduino UNO R3 and provides power to it during operation

1.2 Programming

Standard programming typically involves developing software for general-purpose computers that have **abundant resources** of processing power, memory, and storage. These systems can handle complex computations and multitasking, where performance, usability, and maintainability are the primary concerns. In standard programming, developers use high-level programming languages and powerful development environments and thw software is usually run on an operating system which provides features like memory management, multitasking, and device drivers. On the other hand, embedded programming is focused on writing software for hardware with **limited resources** and should run on a bare-metal setup (without an operating system) or with lightweight operating systems. This requires careful management of hardware resources. Debugging is also more challenging because tools for monitoring may be limited, and developers often rely on specialized equipment. While standard programming may involve developing applications with complex user interfaces (UIs) and handling large datasets, embedded systems typically have minimal or no UIs. Instead, interaction is usually through simple components like leds, buttons, or small displays. In some cases, communication with other systems may occur via network protocols or serial communication. The key difference lies in how embedded programming prioritizes resource constraints, real-time processing, and direct hardware interfacing, while standard programming focuses on performance and user experience for general-purpose computing.

In combination with sensors and actuators, the programmiong style of a microcontroller can be harnessed to implement the concept of **physical computing**: an interactive system that integrates hardware and software to enable devices to sense and respond to the physical environment:



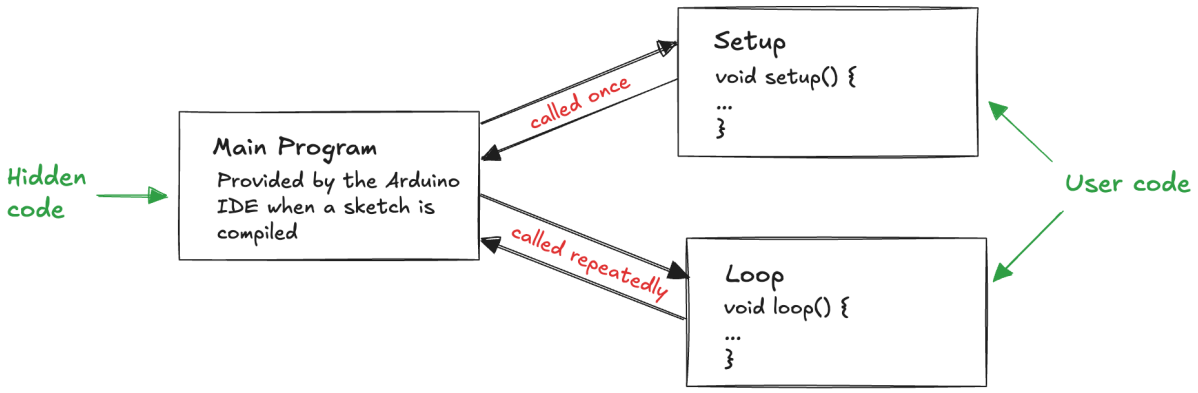
From the perspective of the microcontroller, **inputs** refer to signals or information that are received by the board. These could come from buttons, switches, light sensors, flex sensors, humidity sensors, temperature sensors, and more. **Outputs**, on the other hand, are signals that leave the board, driving actions such as lighting up LEDs, powering DC motors, controlling servo motors, triggering piezo buzzers, or changing the color of RGB LEDs. Almost every physical computing system incorporates

some form of output to respond to the inputs it receives, making interaction with the physical world possible.

To develop a physical computing system, the first step is to **design the circuit**. Begin by determining the electrical requirements of the sensors or actuators we need to use. This includes understanding the voltage, current, and power requirements of each component. Next, identify the analog inputs, such as sensors that provide variable signals (e.g., temperature, light, or humidity sensors). These inputs will typically be connected to the analog pins of the microcontroller. Similarly, identify the digital inputs and outputs, like buttons or switches that act as inputs, and devices like LEDs or motors that serve as outputs. It's important to carefully plan how each component will connect to the microcontroller, ensuring that each part of the circuit is properly powered and grounded.

Once the circuit is designed, we can move on to **writing the code**. It's best to build the code incrementally. Start by getting the simplest piece of functionality working first, such as reading an input or turning on an LED. Once the basic operation works, we can begin to add complexity, step by step, testing our system at each stage to ensure it functions as expected. Throughout the development process, be sure to save and back up our work frequently to avoid losing progress.

In the case of Arduino, the core of a program consists of two C functions: **setup()** and **loop()**. The setup() function runs once when the program starts and is used to initialize settings like pin modes and serial communication. The loop() function runs continuously after setup() and contains the main logic of the program, repeating over and over:



1.2.1 Controlling input and output

Controlling pins is fundamental and the **pinMode()** function is used to configure a pin behavior, specifying whether it acts as an input (to read data) or an output (to send data). Once a pin is set as an output, the **digitalWrite()** function allows us to set the pin to **HIGH** (providing +5V, which can turn on a LED or activate a device) or **LOW** (0V, turning a device off). Conversely, if a pin is configured as an input, the **digitalRead()** function can detect its state, determining whether it is receiving a HIGH or LOW

signal. For analog operations, **analogRead()** reads the voltage from an analog pin as a value ranging from 0 (0V) to 1023 (5V), enabling the measurement of varying signals like those from a potentiometer or a sensor. To simulate an analog output, the **analogWrite()** function generates a **pulse-width modulation (PWM) signal** (which alternates between HIGH and LOW rapidly, creating the illusion of varying voltage levels. This is especially useful for tasks like dimming leds or controlling motor speed.

1.2.2 Timing

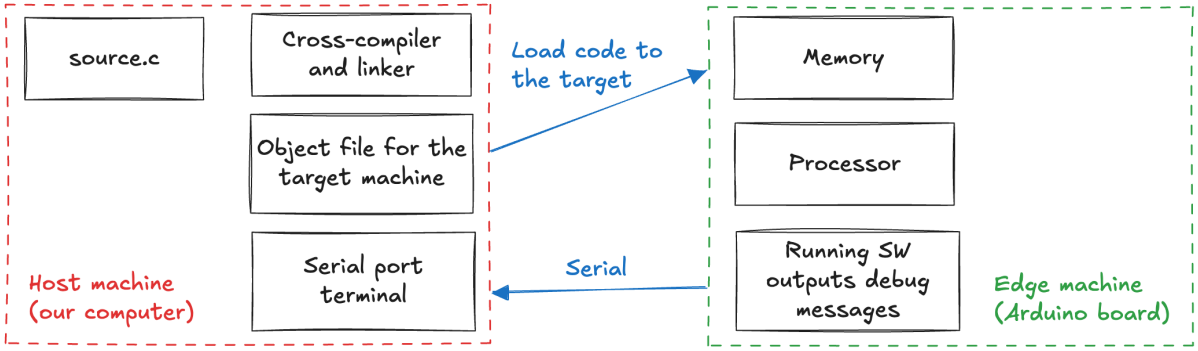
Timing functions are essential for controlling the flow of a program. The **delay()** function pauses the execution of the program for a specified number of milliseconds, making it useful for simple timing tasks, such as blinking a led or waiting between actions. However, because delay() blocks all other code execution during the pause, it can limit the responsiveness of your program For more precise and non-blocking timing, the **millis()** function is a better alternative. It returns the number of milliseconds that have elapsed since the program started running. By comparing the current value of millis() with a previously recorded value, we can track elapsed time while allowing the rest of code to run concurrently. This makes millis() ideal for implementing non-blocking delays.

1.2.3 Hello World!

The first Arduino sketch is typically a simple program called ”Blink,” often referred to as the **hello world** of physical computing. The program makes an onboard led blink, demonstrating the basics of controlling output (see *01.Blink.ino* sketch):

```
// On most Arduino boards, a led is built into pin 13
const int ledPin = 13;

// The setup function runs once when the board starts or resets
void setup() {
  // Configure the LED pin as an output
  pinMode(ledPin, OUTPUT);
}

// The loop function runs continuously after setup
void loop() {
  // Turn the LED on (set the pin to HIGH voltage)
  digitalWrite(ledPin, HIGH);
  // Wait for 1 second.
  delay(1000);

  // Turn the LED off (set the pin to LOW voltage)
  digitalWrite(ledPin, LOW);
  // Wait for 1 second.
  delay(1000);
}
```

Embedded systems are special, offering special challenges to developers. The system's resources are very limited, making it impossible to perform standard code compilation directly on the device. This is because there is no compiler or linker available to run on these devices. Therefore, **cross-compilation** is required. This means using a different machine (the **host**) to compile the source code into a machine language that is executable on the embedded system (the **target**).



In the case of Arduino, we use our computer as the host machine, the Arduino IDE serves as the cross-compiler, and the compiled code is then transferred to the board via a USB connection. In essence, cross-compilation allows us to leverage the computational power of our computer to generate efficient code for our resource-constrained embedded system. When compiled, the program uses like a thousand bytes of the **available memory** to generate the sequence of instructions needed for blinking. With approximately several tens of thousands bytes of memory on most boards, some space remains unused for more complex programs. This simple exercise introduces key concepts like compiling, uploading, and basic hardware control.

1.2.4 Data type, Selection, Repetition and Function

In Arduino programming, we use **standard C data types** to define variables and store values. The most common data types include:

- **Integers**: used to store whole numbers
- **Floats**: used to store decimal numbers with floating-point precision

The following examples demonstrates how integer arithmetic works, particularly how truncation (the removal of the decimal portion of a number) occurs when using integer division (see *02.Integers.ino* sketch) and the well-known issue of round-off error that can occur with floating-point arithmetic (see *03.Floatingpoint.ino* sketch). Arduino supports **serial communication** for debugging. We can initialize serial communication using **Serial.begin()**, and functions like **Serial.print()** to send data to the serial monitor:

```
1  void setup() {
```

```
  int i,j;

  // initialize serial communication
  Serial.begin(9600);

  //  wait for user to open the serial monitor
  delay(3500);

  // 2/3 is performed using integer division, which results in 0
  // because 2 and 3 are both integers and the result of their division is truncated
  // (i.e., the fractional part is discarded).
  // This result is then multiplied by 4, which gives i = 0.
  i = (2/3)*4;

  // The variable j is then assigned the value of i + 2, which results in j = 2.
  j = i + 2;

  // Print the values of i and j to the serial monitor
  Serial.println("First test");
  Serial.print(i);
  Serial.print("  ");
  Serial.println(j);

  // In this case, 2.0 and 3.0 are floating-point numbers, so the division result
  // is a floating-point number, yielding approximately 0.6667
  // This value is then multiplied by 4.0, resulting in i = 2.6667
  // However, because i is declared as an int, it will be truncated when stored, so i = 2
  i = (2.0/3.0)*4.0;

  // The variable j is then assigned the value of i + 2, which gives j = 4
  j = i + 2;

  Serial.println("Second test");
  Serial.print(i);
  Serial.print("  ");
  Serial.println(j);
}
```

```
void setup() {
    // define the variables
    float w,x,y,z;

    Serial.begin(9600);
    delay(2500);

    // 4.0 / 3.0 gives an approximation of 1.333..., which is not exactly 4/3,
    // as floating-point numbers cannot always represent exact decimal values.
    w = 4.0/3.0;

    // w - 1 results in 0.333...
    x = w - 1;

    //3 * x results in 0.999..., which should theoretically equal 1,
    // but due to the floating-point approximation, it is just shy of 1.
    y = 3*x;

    // 1 - y results in a small value close to zero, which demonstrates
    // the rounding error introduced in the earlier steps
    z = 1 - y;

    // Print the values of w, x, y, and z to the serial monitor
    // using two-parameter verson of Serial.print(), the second parameter
    // specifies the number of digits in value sent to the Serial Monitor

```

```
    Serial.println("\nFloating point arithmetic test");
    Serial.println(w,16);
    Serial.println(x,8);
    Serial.println(y,8);
    Serial.println(z,8);

    // Prints z multiplied by a big number to better visualize the effect of
    // the round-off error when scaling up the small floating-point value
    Serial.println(z*1.0e7,8);
}
```

Arduino also supports **common control structures** found in C language:

- **Selection (if statement)**: used to make decisions based on conditions.
- **Repetition (loops)**:
  - **For loop**: a loop that repeats a block of code a specified number of times
  - **While loop**: a loop that continues to execute as long as a condition remains true.

The following example (see *04.Repetition.ino* sketch) demonstrates the basics of using a loop for a repetitive task:

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

void loop() {
    // define the variables
    int i;

    // Print the numbers from 0 to 9 using a for loop
    for (i=0; i<10; i++) {
        Serial.println(i);
        delay(100);
    }

     Serial.println("for loop over\n");
}
```

Functions are a powerful way to organize and reuse code. By encapsulating details of a task into a function, we can break our program into smaller, more manageable blocks. Well-written functions improve code readability and maintainability. As usual, functions can accept input parameters (e.g., values passed to the function to perform tasks) and optionally return an output (e.g., a value resulting from the function's operations).

The following example (see *05.Function.ino* sketch) demonstrates how to use a function to read an analog sensor (like a potentiometer) multiple times and calculate the average value of those readings, in order to smooth out noise and getting a stable sensor reading:

```
// A function to calculates the average of a number of readings
// from a sensor connected to a specific pin
float average_reading(int sensor_pin, int size) {
    int i;
    float average;
```

```
    float sum;

    // Initialize the sum to zero
    sum = 0.0;

    // Read the sensor value 'size' times and add them to the sum
    for (i=1; i<=size; i++) {
        sum = sum + analogRead(sensor_pin);
    }

    // Calculate the average by dividing the sum by the
    // number of readings
    average = sum/size;

    // Return the average
    return(average);
}

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

void loop() {
    // Numer of readings
    int n=15;

    // Pin to which the potentiometer is connected
    int pot_pin=1;

    // The reading from the potentiometer
    float reading;

    // Call the average_reading function to calculate the average reading
    reading = average_reading(pot_pin,n);

    // Print the reading and voltage to the serial monitor
    Serial.println(reading);
    Serial.println();
}
```

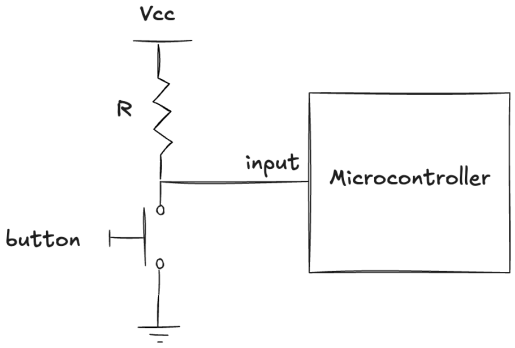
1.2.5 Interrupt

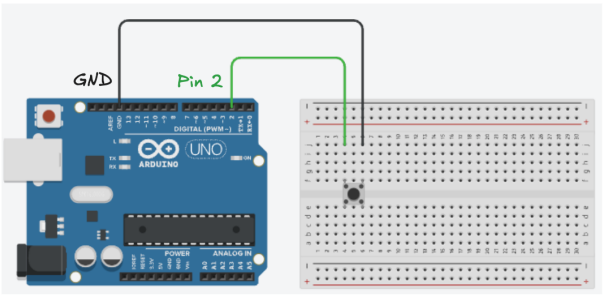
An interrupt is a mechanism that allows a microcontroller to temporarily stop its current task to handle a more immediate task. It is an essential feature in embedded systems, enabling a program to **respond immediately** to certain events without constantly checking for them in the main loop. They are managed using the **attachInterrupt()** function, which links a specific pin to an **interrupt service routine (ISR)**, a special function that runs when the event occurs. When an interrupt occurs, the currently executing program is paused, the ISR is executed to handle the interrupt and after it is finished, the main program resumes from where it left off. For example, we can use interrupts to detect a button press, even if our main program is busy with other tasks. With attachInterrupt(), we specify the interrupt pin, the ISR to execute, and the **trigger mode**, that determines when the interrupt will fire:

- **RISING**: triggers when the pin transitions from LOW to HIGH
- **FALLING**: triggers when the pin transitions from HIGH to LOW

- **CHANGE**: triggers on any change in the pin's state (from HIGH to LOW or vice versa).

Interrupts are particularly useful for **handling time-sensitive events** or for tasks where **polling** (repeatedly checking the status of an event at regular intervals to see if it has occurred) would be inefficient. As an example, consider the problem of **handling a push-button input** to toggle an LED on and off like in the following schematic:





When the button is not pressed, the pin is **pulled HIGH (connected to +5V)** through a pull-up resistor (that limits the current flowing), when the button is pressed, it creates a connection between the pin and ground, pulling the pin LOW. Many Arduino pins have **built-in internal pull-up resistors**. We need to configure the digital pin as an **INPUT_PULLUP** in our code in order to enable this internal resistor. In a **polling-based approach**, we would need to continuously check the button state in the main loop, which could lead to missed button presses if the loop is busy with other tasks (see *06.NoInterrupt.ino* sketch):

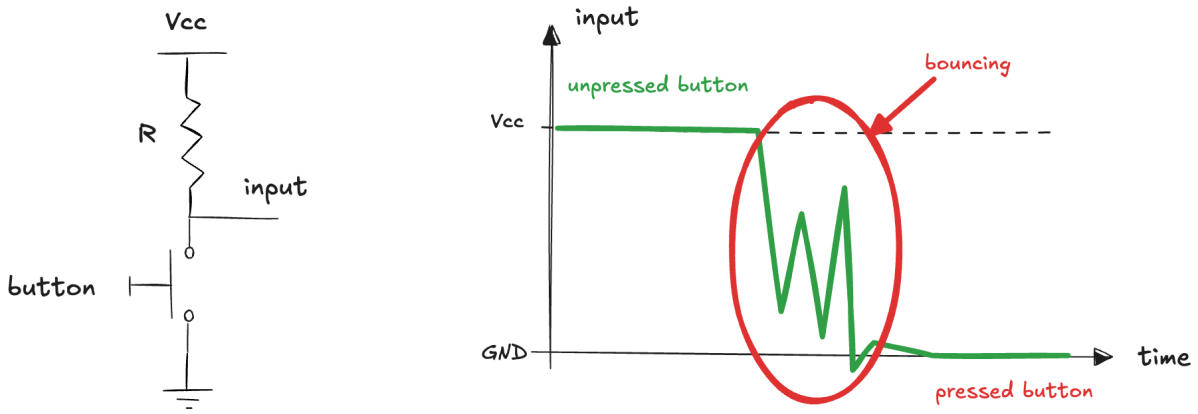
```
// Define the pin with the LED (the one on the board)
int led_pin = 13;

// define the pin with the button
int button_pin = 2;

// Function to handle the button input and control the LED
void handle_button() {
    // Read the button and set the LED accordingly
    digitalWrite(led_pin, digitalRead(button_pin));
}

// Function to handle other tasks that take time
void handle_other_stuff() {
    // Simulate a delay for a long-running task
    delay(250);
}

void setup() {
    // Set LED pin as output
    pinMode(led_pin, OUTPUT);
    // Set button pin as input with internal pull-up resistor
    pinMode(button_pin, INPUT_PULLUP);
}

void loop() {
    // Check the button and update LED
```

```
28      handle_button();
29
30      // Perform other tasks (with delay)
31      handle_other_stuff();
32  }
```

The button state is only checked periodically, and if the button is pressed while the program is inside the handle_other_stuff() function, the press might be missed. Additionally, the button input might not be handled quickly enough. We can break up the long tasks into smaller chunks and handle the button press more frequently. Still, this approach can become inefficient as the time between button presses decreases. By using an interrupt, we can immediately respond to the button press, regardless of what the main loop is doing. The following example demonstrates how to use an interrupt to toggle an LED when a button is pressed (see *07.Interrupt.ino* sketch):

```
1   void setup() {
2       pinMode(LED, OUTPUT);
3       pinMode(SW, INPUT_PULLUP);
4
5       // Set up an interrupt on button_pin (which corresponds
6       // to INT0 on the Arduino Uno
7       attachInterrupt(INT0, handleSW, CHANGE);
8   }
9
10  // The loop function call only the handleOtherStuff function
11  void loop() {
12      handleOtherStuff();
13  }
```

The attachInterrupt() function is used to set up an interrupt on button_pin (pin 2, which corresponds to INT0 on the Arduino Uno). The interrupt triggers when the state of the button changes (from LOW to HIGH or HIGH to LOW) by using the CHANGE mode. When the button state changes (i.e., when the button is pressed or released), the handle_button() function is called immediately, regardless of what the main loop is doing. In the loop(), the handle_other_stuff() function is called, which simulates performing other tasks (e.g., running a long process). This ensures that button presses are detected and processed instantly, even if the program is executing other tasks. However, there is a potential problem, the interrupt might be triggered multiple times for a single button press. This is known as **bouncing**, a common issue with mechanical switches. When a button is pressed, the contacts inside the switch can bounce back and forth before settling into a stable state:



To show the button bouncing problem, we can modify the code to show how the led can rapidly flicker when a button is pressed or released due to the mechanical bouncing (see *08.NoDebouncing.ino* sketch):

```
// Add a variable to count the number of times the button is pressed
// to observe the bouncing problem
volatile int count = 0;

void handle_button() {

    digitalWrite(led_pin, digitalRead(button_pin));

    // Increment the count each time the button is pressed
    count++;

    // Print the count to the serial monitor
    Serial.println(count);
}
```

Notice that the variable count is shared between an ISR and the loop() function, the program modify this variable in the ISR while the main program can read and act upon it. It is important to declare it as **volatile**, since this specification tells the compiler that the variable can be modified by an external source (the ISR in that case) and should not be optimized, the main program always has to fetch the latest value directly from memory, reflecting any changes made during the interupt.

When dealing with the mechanical bouncing problem we can use either hardware or software techniques. The **hardware solution** involves the use of external electronic components, such as a resistor-capacitor RC filter. These components smooth out the signal from the button and eliminate oscillations. The primary advantage of this approach is that it offloads the task from the microcontroller, freeing it for other computations. Additionally, it ensures consistent debouncing without using the microcontroller's processing resources. However, this method requires additional components, which can increase cost and complexity. Moreover, it is less flexible since modifying the timing requires physical changes to the circuit. The **software solution**, on the other hand, relies on programming techniques such as introducing a delay or using timing mechanisms to ignore button state changes

until the bouncing has subsided. This approach does not require any extra hardware, making it cost-effective and easy to implement. Furthermore, it is highly flexible, as the debounce timing can be adjusted directly in the code. Despite its simplicity, the software solution has some drawbacks. It adds slight delays to the system's response time and consumes some of the microcontroller's processing cycles, which could affect performance in resource-constrained systems (see *09.Debouncing.ino* sketch):

```
// Timestamp of last valid button press
volatile unsigned long lastDebounceTime = 0;

// Debounce delay in milliseconds
const unsigned long debounceDelay = 50;

// Function to handle the button with debouncing
void handle_button() {
    // Check the time since the last valid button press
    unsigned long currentTime = millis();
    if (currentTime - lastDebounceTime > debounceDelay) {
        digitalWrite(led_pin, digitalRead(button_pin));

        // Print the count to the serial monitor
        count++;
        Serial.println(count);

        // Update the last debounce time
        lastDebounceTime = currentTime;
    }
}
```

In the handle_button() function, the time elapsed since the last button press is checked using millis(). If the elapsed time exceeds the debounce delay, the button press is considered valid and the led state is toggled only when a valid button press is detected.

1.3 Serial communication

To send and receive text and data, we can use the serial communication interface. This allows us to transmit information from the microcontroller to be any serial device. To initiate serial communication, the speed (baud rate) must be specified using the **Serial.begin()** function:

```
Serial.begin(9600);
```

Here, 9600 refers to the **baud rate**, which is the number of symbols transmitted per second. It's essential that both the sending (Arduino) and receiving (PC) sides use the same baud rate; otherwise, the output will be unreadable or completely absent.

1.3.1 Send and receive text

Using the serial interface, we can send text or data to be displayed. For instance:

```
Serial.print("The number is ");
Serial.println(number);
```

Serial.print() sends text or numbers without moving to a new line, Serial.println() sends text or numbers and then moves to the next line. We can also define the format for numerical values, such as decimal, hexadecimal, or binary. For example:

```
Serial.println(number, HEX); // Display the number in hexadecimal
Serial.println(number, BIN); // Display the number in binary
```

To receive data on an Arduino from a computer, for example for reacting to commands or data sent from an external source, we can use the same library. To check if data is available, we use the **Serial.available()** function. This returns the number of characters in the serial buffer. For example:

```
if (Serial.available() > 0) {
    // There is data to read
}
```

Once data is available, we can use **Serial.read()** to retrieve the next byte from the buffer. For example:

```
char receivedChar = Serial.read();
Serial.print("You sent: ");
Serial.println(receivedChar);
```

As an example, the following program (see *10.SerialReceive.ino*) controls the blink rate of a led based on numeric input received over the serial connection:

```
// The pin to which the LED is connected
const int led_pin = 13;

// Initial blink rate
int blink_rate = 100;

// Function to controls the LED by turning it on, waiting for the blink_rate duration,
// then turning it off for the same duration.
void blink() {
    digitalWrite(led_pin, HIGH);
    delay(blink_rate);
    digitalWrite(led_poin, LOW);
    delay(blink_rate);
}

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

void loop() {
    // Check to see if at least one character is available
    if (Serial.available()) {

        // If data is available, reads a character
        char ch = Serial.read();

        // If the character is a numeric ASCII digit ('0' to '9'),
```

```
29          // convert it to its numeric equivalent using (ch - '0')
30          if(ch >= '0' && ch <= '9') {
31              blink_rate = (ch - '0') * 100;
32          }
33      }
34
35      // Call the blink function to blink the LED
36      blink();
37  }
```

When receiving numbers with more than one digit, we need to accumulate characters until we encounter a non-numeric character (e.g., a space or newline). This allows us to construct the full number before processing it. For example (see):

```
1   // Variable to accumulate the digits of the number
2   int value;
3
4   void loop() {
5       if (Serial.available()) {
6           char ch = Serial.read();
7
8           // The accumulated number is updated as a decimal value by multiplying the
9           // existing value by 10 and adding the new digit
10          if(ch >= '0' && ch <= '9') {
11              value = (value * 10) + (ch - '0');
12          }
13
14          // When a newline character is received, the accumulated value is used to
15          // calculate the new blinkRate. The value variable is reset to 0
16          // to prepare for the next input.
17          else if (ch == 10) {
18              blink_rate = value * 100;
19              Serial.println(blink_rate);
20              value = 0;
21          }
22      }
23      blink();
24  }
```

This process enables the Arduino to interpret and respond to commands or numerical data, making it possible to control its behavior from a computer or other serial device.

1.3.2 Send and receive multiple text fields

To transmit multiple pieces of data (*fields*) from an Arduino, such as sensor readings for temperature, humidity, and pressure, it is crucial to define the data clearly and format it in a structured way. A common method is to use a **comma-separated format** with a unique "**header**" to indicate the start of the message and a consistent "**delimiter**" (e.g., commas) to separate the fields. The header must be distinct, ensuring it does not appear within any data fields or as the delimiter. This ensures proper parsing on the receiving side. Once the message is formatted, it can be transmitted using Serial.print() to send the data over the serial connection. For example, temperature, humidity, and pressure readings can be sent in a comma-separated format with 'H' as the header, see *11.CommaDelimitedOutput.ino*

sketch:

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

void loop() {
  // Simulated sensor readings
  float temperature = 25.3;
  float humidity = 60.1;
  float pressure = 1013.2;

  // Send a comma-delimited message with a header
  Serial.print('H');
  Serial.print(",");
  Serial.print(temperature);
  Serial.print(",");
  Serial.print(humidity);
  Serial.print(",");
  Serial.print(pressure);
  Serial.print(",");
  Serial.println();

  // Wait before sending the next message
  delay(100);
}
```

From the receiving side, the data have to be parsed by reading the serial buffer and extracting the fields based on the delimiter. In order to implement an example of this, we need to write code also on the receiving side. As an example, we can exploit the **Processing**, a powerful tool to ”talk” with Arduino, particularly for visualizing or processing incoming data. It is based on a simplified version of the Java programming language and provides a simple and intuitive interface for creating interactive graphics and animations. Like Arduino programming, it uses a setup() function to initialize the program and a draw() fuction to continuously update the display. Moreover, it is based on a event-driven model, where the program responds to user input or other events (like data coming from the serial port). The following example demonstrates how to receive and parse the comma-separated data received by Arduino:

```
// imports the Serial class from the processing.serial library
import processing.serial.*;

// Serial port object
Serial myPort;

// Buffer to store received data
String received_data;

// Parsed data
float temperature = 0.0;
float humidity = 0.0;
float pressure = 0.0;

void setup() {
    // Set up the application window
    size(400, 200);

    // Connect to the right port (the one that the Arduino is connected to):
```

```
    println(" Connecting to -> " + Serial.list()[2]);
    myPort = new Serial(this,Serial.list()[2], 9600);
}

void draw() {
    // Write the received data to the screen
    background(0);
    text("Temperature: " + temperature, 10, 50);
    text("Humidity: " + humidity, 10, 60);
    text("Pressure: " + pressure, 10, 70);
}

// This method is called whenever data is received from the serial port
void serialEvent(Serial p) {

    // Read data until newline character
    received_data = myPort.readStringUntil('\n');

    // If data is received, parse the data by splitting the
    // received string
    if (received_data != null) {
        String[] fields = split(received_data, ',');

        // check the header
        if (fields[0].equals("H")) {
            temperature = float(fields[1]);
            humidity = float(fields[2]);
            pressure = float(fields[3]);
        }
    }
}
```

To receive a message containing multiple fields on Arduino, such as an identifier for a specific device (e.g., a motor or actuator) and a corresponding value (e.g., speed to set it to), it is necessary to parse the incoming serial data. The message should follow a structured format where each field is separated by a delimiter, such as a comma. The following example, first checks for incoming data using Serial.available() to determine if there is data to read. It then reads and parses the message using Serial.read(), collecting characters until the message is complete. Once the message is fully received, it is split into fields using the designated delimiter. After parsing the message, the identifier and value are extracted from the fields and processed to perform the corresponding task, such as setting the speed of the specified motor. The following ecample demonstrates how to receive a message containing multiple numeric fields separated by commas and store the values into an array. The expected format of the message is 12,345,678, where each number is separated by a comma (see *12.SerialReceiveMultipleFields.ino* sketch):

```
// number of expected fields
const int NUMBER_OF_FIELDS = 3;

// current field being received
int field_index = 0;

// Array to hold velues
int values[NUMBER_OF_FIELDS];

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

```
}

void loop() {
    if( Serial.available()) {
        char ch = Serial.read();

        // Accumlate digits to build the value of a field
        if(ch >= '0' && ch <= '9') {
            values[field_index] = (values[field_index] * 10) + (ch - '0');
        }

        // Comma is our separator, so move on to the next field
        else if (ch == ',') {
            if(field_index < NUMBER_OF_FIELDS-1)
            field_index++;
        }

        // Any other character ends the acquisition of fields and
        // we provide some feedback, the we set the field to zero
        // to start collecting the new value
        else {
            Serial.print(field_index +1);
            Serial.println(" fields received:");
            for(int i=0; i <= field_index; i++) {
                Serial.println(values[i]);
                values[i] = 0;
            }
        }

        // Ready to start over
        field_index = 0;
    }
}
```

1.3.3 Binary data excange

In addition to exchanging textual information (formatted as characters), it is also possible to exchange data directly in **binary format**. The following code demonstrates how to send binary data over a serial connection. The **lowByte()** and **highByte()** functions can split a 16-bit integer into two 8-bit segments (low and high bytes). These bytes are sent over the serial connection using Serial.write() function (see *13.SendBinary.ino* sketch):

```
// Declare a 16-bit integer variable
int value;

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

void loop() {
    // send a header character
    Serial.print('H');

    // Generate a random integer
    value = random(599);

    // Send the low and high bytes
    Serial.write(lowByte(value));
```

```
    Serial.write(highByte(value));

    delay(1000);
}
```

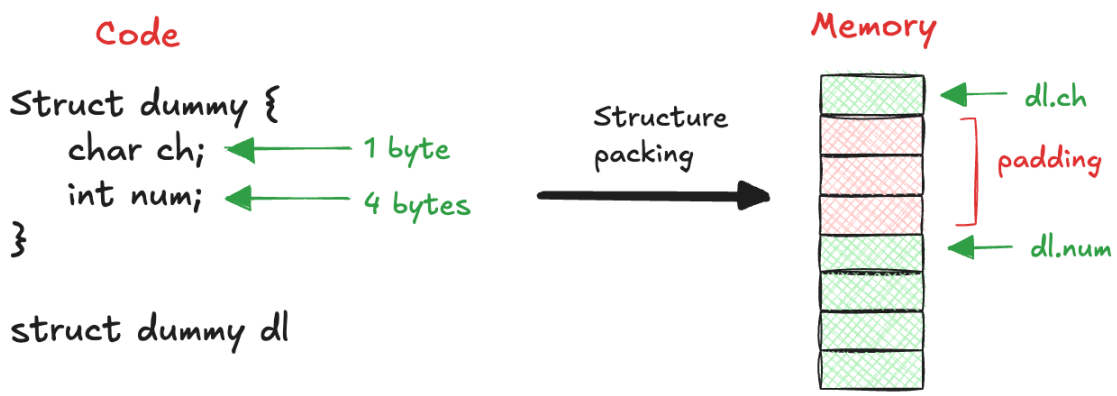
Binary formats **require fewer bytes** to represent the same information compared to text-based formats. For instance, an integer occupies 2 bytes in binary, while its textual representation (e.g., ”12345”) requires significantly more bytes. This compactness reduces both transmission time and bandwidth usage.Another advantage is that it **eliminates the need for parsing** numeric values from text, making data processing faster and more efficient. However, binary data has notable drawbacks. It is **not human-readable**, making debugging and manual inspection challenging without specialized tools or decoding scripts. Additionally, platform-specific issues such as endianness (e.g., little-endian vs. big-endian) and differences in data structure interpretation (e.g., identifying integers, floats, or strings) can cause inconsistencies unless both sender and receiver adhere to a predefined protocol.In the provided example, the receiving program must first read the header 'H' to identify the start of a message. It then reads the low and high bytes for each integer and reconstructs the original value. Let's see this in action using Processing:

```
void draw() {
    // Wait until at least 3 bytes are available
    //(header + 2 bytes for the integer)
    if (myPort.available() >= 3) {

        // Check for the header character
        if (myPort.read() == HEADER) {

            // Read the low byte
            int low = myPort.read();

            // Read the high byte
            int high = myPort.read();

            // Reconstruct the 16-bit integer
            value = (high << 8) | low;

            println("Message received: " + value);
        }
    }
}
```

Sending binary data requires **careful planning and coordination** between the sender and receiver to ensure that the data is correctly interpreted. The sender must format the data consistently, while the receiver must parse the data according to the agreed-upon structure. In particular, key considerations inlclude:

- **Variable Size**: ensure that the size of the data being sent matches the size expected by the receiver. Check the programming language’s documentation for the exact size of data types (e.g., an int is 2 bytes on Arduino but might be 4 bytes in Processing). One approach can be to use **explicitly sized types** (e.g., uint16_t in Arduino) and verify the range of values sent does not exceed the maximum value the receiving type can hold to avoid **overflow**.

- **Byte Order (Endianness)**: ensure that the bytes within multi-byte values are sent in the order expected by the receiver (little-endian vs. big-endian). A solution can be standardize the byte order between systems, typically by sending data in little-endian order. Also using helper functions (like lowByte() and highByte()) on the sending side and reconstruct values correctly on the receiving side.
- **Synchronization**: the receiver must recognize the start and end of a message. If the receiver begins listening mid-stream, it could interpret bytes incorrectly. A possibile solution is to include a header or a delimiter in the message that uniquely identifies the start of valid data (e.g., 'H' as a header). Of course, we need also to avoid the use of values in the message body that could match the header.
- **Structure Packing**: data alignment (see figure) can vary across compilers, leading to mismatches if a structure is packed differently on each side. Compilers may pad data to align it for better performance. A solution is to explicitly define structure packing. For example, in C/C++ the #pragma pack(1) preprocessor statement ensure no padding. In general, it is better to avoid sending complex structures directly, instead serialize data manually to ensure consistent packing.



- **Flow Control**: if the sender transmits data faster than the receiver can process it, data could be lost or corrupted. A solution is to use a **handshake mechanism** where the receiver signals readiness before the sender transmits more data or choose an appropriate baud rate that allows the receiver to process data at the same rate it is sent.

Sending binary data from an external device to a microcontroller follows similar principles. It's important to ensure that the data format is correctly understood by the microcontroller's firmware. Depending on the system, the binary data might represent control commands, sensor readings, or other structured data that the microcontroller needs to process efficiently. The following code defines a simple communication system between a Processing application and an Arduino using serial communication. The message consists of an header ('|') and a tag ('M') to define the mouse X-coordinate and the

mouse Y-coordinate clicked on a Processing window. The tag is used becouse the message can in prnciple contain multiple fields, each one identified by a different tag.The windows size is set to 200x400 pixels, so the X-coordinate ranges from 0 to 200 and needs only one byte to be represented, the Y-coordinate ranges from 0 to 400 and needs two bytes to be represented: (see *14.ReceiveBinary.ino* sketch):

```
import processing.serial.*;

Serial myPort;

// Define the header and the tag for the message
public static final char HEADER = '|';
public static final char MOUSE  = 'M';

void setup(){
  size(200, 400);
  println(" Connecting to -> " + Serial.list()[2]);
  myPort = new Serial(this,Serial.list()[2], 9600);
}

void draw(){ }

// This callback function is called whenever data is received
// and it is used to echoes the received data to the console
void serialEvent(Serial p) {
  String inString = myPort.readStringUntil('\n');
  if(inString != null) {
      println( inString );
  }
}

// This callback function is called whenever the mouse is pressed
// It gets the x and y coordinates of the mouse and sends them
void mousePressed() {
  int x = mouseX;
  int y = mouseY;
  sendMessage(MOUSE, x, y);
}

// The function sends a header, a tag, x-coordinate as single bytes and
// the y-coordinate as two bytes.
void sendMessage(char tag, int x, int y) {
  myPort.write(HEADER);
  myPort.write(tag);
  myPort.write(x);
  myPort.write((byte)(y >> 8));  // MSB
  myPort.write((byte)(y & 0xFF)); // LSB
}
```

The receiving part on Arduino side is the following:

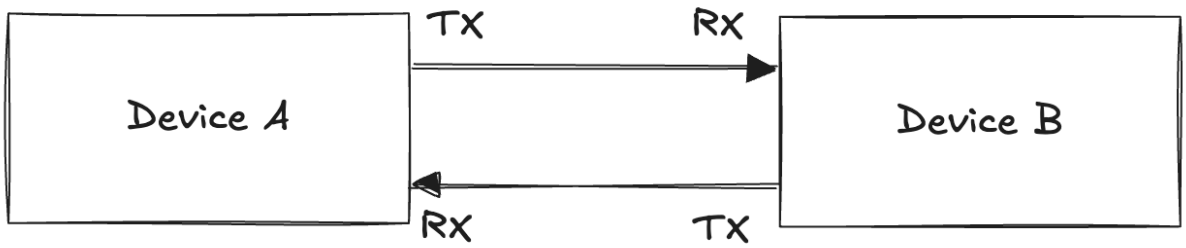
```
// Define header, tag, and message length
#define HEADER '|'
#define MOUSE 'M'
#define MESSAGE_BYTES 5

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

void loop() {
```

```
    // Check if there are enough bytes available to read
    if ( Serial.available() >= MESSAGE_BYTES) {

      // Check the header byte
      if( Serial.read() == HEADER) {

        // Read the tag byte
        char tag = Serial.read();

        // Check if the tag is for a mouse message
        if(tag == MOUSE) {

            // Read the X-coordinate (1 byte)
            int x = Serial.read();

            // Read the Y-coordinate (2 byte) and reconstruct the value
            int y = Serial.read() * 256;
            y = y + Serial.read();

            // Send back the received values
            Serial.print("Received mouse msg, x = ");
            Serial.print(x);
            Serial.print(", y ");
            Serial.println(y);
        }

        // If the code gets here, the tag was not recognized.
        // this helps to ignore data that may be incomplete or corrupted.
        else {
            Serial.print("got message with unknown tag ");
            Serial.println(tag);
        }
      }
    }
}
```

In general, the binary approach is particularly useful for transmitting **large amounts of data** or when **speed and efficiency are critical**. However, it is essential to consider the **trade-offs between efficiency and readability** when choosing between binary and text-based formats. Binary format is recommended for systems where efficiency, fast data processing, and low latency are crucial. For all other cases, the ease of debugging and the human-readability offered by text-based formats typically make them the better choice.

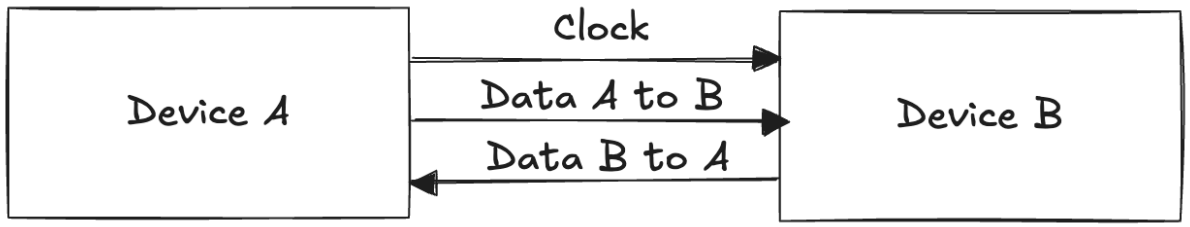
1.3.4 Synchronous communication

In the example we have seen so far, based on USB protocol and on the Serial library the communication between the Arduino and the computer is **asynchronous**:



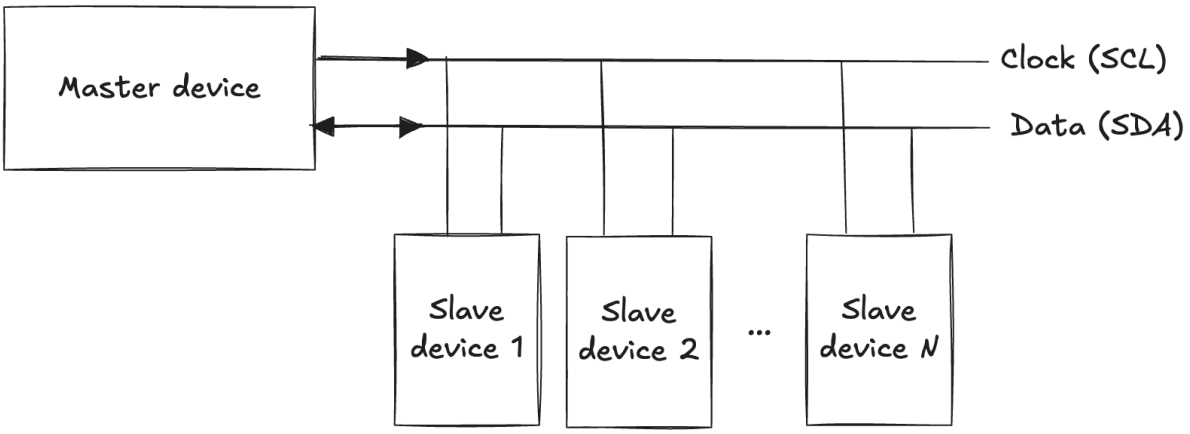
In asynchronous communication, data is sent without a shared clock signal, relying on the sender and receiver to agree on a specific data rate (baud rate) and interpret the data based on their internal timing mechanisms, making it crucial for both devices to have precise internal oscillators. Separate wires are used for transmitting (TX) and receiving (RX) data, allowing bidirectional communication. This approach is simple and effective for many applications, but it can be prone to timing errors, especially over long distances or in noisy environments.

An alternative approach is **synchronous communication**, where devices use a **shared clock** signal to synchronize data transmission. This method eliminates the need for independent timing mechanisms within the devices, as the clock acts as a ”conductor” guiding the flow of data.



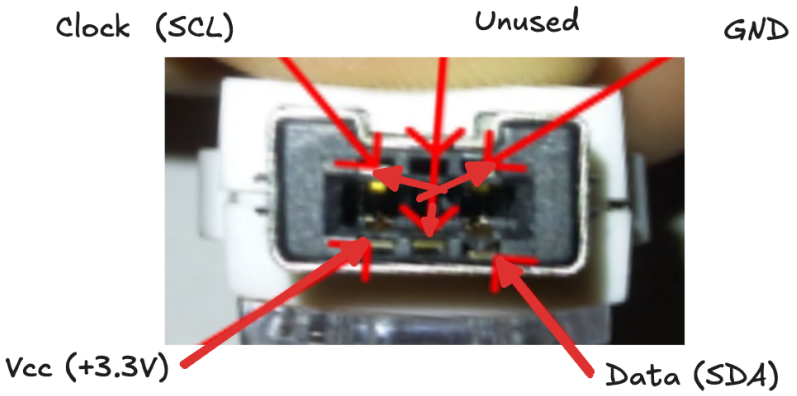
This method ensures that both devices operate in lockstep, sampling data at the correct moments. Synchronous communication offers more reliable data exchange by eliminating timing mismatches and ensuring precise synchronization between devices. However, it requires **additional wiring for the clock signal**, which can increase hardware complexity. The choice between asynchronous and synchronous communication depends on the application requirements for timing precision, hardware constraints, and error tolerance.

The **I2C (Inter-Integrated Circuit)** protocol is an example of synchronous communication, In I2C, devices (master and slaves) communicate over a shared bus consisting of two wires: one for data (**SDA: Serial Data Line**) and one for the clock (**SCL: Serial Clock Line**). It supports multiple devices on the same bus, where each device has a unique address. The master device controls the communication by generating the clock signal and initiating data transfers, while the slave devices respond to commands from the master:

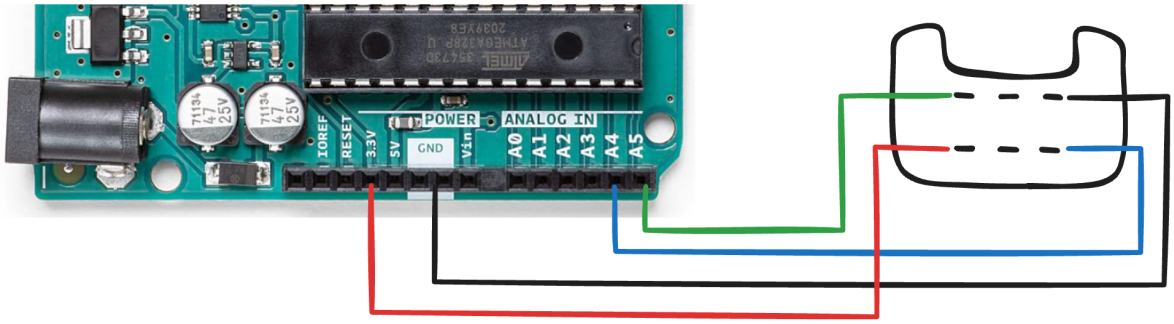


Data is transmitted bit by bit, and each bit is sampled on the clock rising or falling edge (depending on the implementation), this synchronization is what makes I2C a synchronous protocol. When a master device wants to communicate with a slave, it first sends the address of the slave along with a read/write bit. The slave device that matches the address acknowledges the request, and data can then be transferred between the devices. Data is transferred in 8-bit chunks (1 byte at a time), with each byte being transmitted in sync with the clock. After each byte, the receiver sends an acknowledgment (ACK) bit, signaling that the byte was successfully received. The clock-driven nature of I2C ensures that both the master and slave devices stay synchronized, which eliminates the need for start and stop bits, a feature that is common in asynchronous protocols like UART. The data integrity in I2C is guaranteed by the clock signal, which ensures that each bit is transmitted at the correct time. Because of this synchronization, the protocol is reliable and efficient in systems where multiple devices need to communicate with a single master.

As an exmaple, we can interact with the **Wii Nunchuck**, a popular I2C device, to read its accelerometer and joystick data:



Most Arduino boards support I2C communication mapped to specific analog pins: SCL on pin A5 and SDA on pin A4:



Although there are no official datasheets, the functionality of Wii Nunchuck has been **reverse-engineered**, making it accessible to developers. The I2C slave address is 0x52 and the communication is based on an handshake signal to initialize the Nunchuck (send a sequence of two bytes 0x40 and 0x00) and a data request command (send one byte 0x00) to request data. The Nuunchuck responds with a 6-byte data packet containing the joystick positions (X byte 0 and Y byte 1), accelerometer data (X byte 2, Y byte 3, Z byte 4) and the button states and some least significant bits of acceleration (byte 5). The data are encoded using a specific scheme to ensure data integrity:

```
data = (reading XOR 0x17) + 0x17
```

Arduino provides the **Wire library**, which abstracts the I2C protocol. The library simplifies the process of sending and receiving data over the I2C bus. The function *Wire.begin()* intialize the bus in master mode. For sending data che *Wire.beginTransmission(address)* starts communication with a specific slave device (identified by its address), *Wire.write(data)* sends data (a single byte, a string, or an array of bytes) to the slave device and *Wire.endTransmission()* ends the transmission and releases the bus. To request data, *Wire.requestFrom(address, quantity)* requests a specified number of bytes from a slave device, *Wire.read()* reads the received data byte by byte and *Wire.available()* returns the number of bytes available to read. The following example demonstrates how to read data from a Nunchuck and send data to an external device connected on the serial port (see *15.Nunchuck.ino*):

```
// Include the Wire library for I2C communication
#include <Wire.h>

// Define the Nunchuk  slave address
#define NUNCHUK_ADDRESS 0x52

// Buffer for data received
static uint8_t nunchuck_buf[6];

// Establishes I2C communication with the nunchuck
static void nunchuck_init() {
    // Join the I2C bus a s master
    Wire.begin();

    // Initialize the slave device (nunchuck) on address 0x52
    Wire.beginTransmission(NUNCHUK_ADDRESS);

    // Send handshake signal to initialize the Nunchuk
    Wire.write((uint8_t)0x40);
```

```
    Wire.write((uint8_t)0x00);
    delay(100);
    Wire.endTransmission();

    Serial.println("Wii Nunchuk initialized!");
}

static void nunchuck_send_request() {
    Wire.beginTransmission(NUNCHUK_ADDRESS);
    Wire.write((uint8_t)0x00);
    Wire.endTransmission();
}

// A function to decode Nunchuk data
char nunchuk_decode(char reading) {
  return (reading ^ 0x17) + 0x17;
}

// Receive data from the nunchuck
static int nunchuck_get_data() {
    int cnt=0;

    // Get six bytes of data from device 0x52 (nunchuck).
    Wire.requestFrom (NUNCHUK_ADDRESS, 6);

    // Indicate how many bytes have been received
    while (Wire.available ()) {
      nunchuck_buf[cnt] = nunchuk_decode( Wire.read() );
      cnt++;
    }

    nunchuck_send_request();

    if (cnt >= 5) {
        return 1;
    }

    return 0;
}

int nunchuck_zbutton() { return ((nunchuck_buf[5] >> 0) & 1) ? 0 : 1; }
int nunchuck_cbutton() { return ((nunchuck_buf[5] >> 1) & 1) ? 0 : 1;  }
int nunchuck_joyx() { return nunchuck_buf[0]; }
int nunchuck_joyy() { return nunchuck_buf[1]; }
int nunchuck_accelx() { return nunchuck_buf[2]; }
int nunchuck_accely() { return nunchuck_buf[3]; }
int nunchuck_accelz() { return nunchuck_buf[4]; }

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

void loop() {

  nunchuck_get_data();

  Serial.write(nunchuck_joyy());

  delay(100);
}
```

We can write a Processing program to visualize data received from the Nunchuk connected to the

Arduino. It draws a line on the canvas that dynamically updates its position based on the data sent via the serial port:

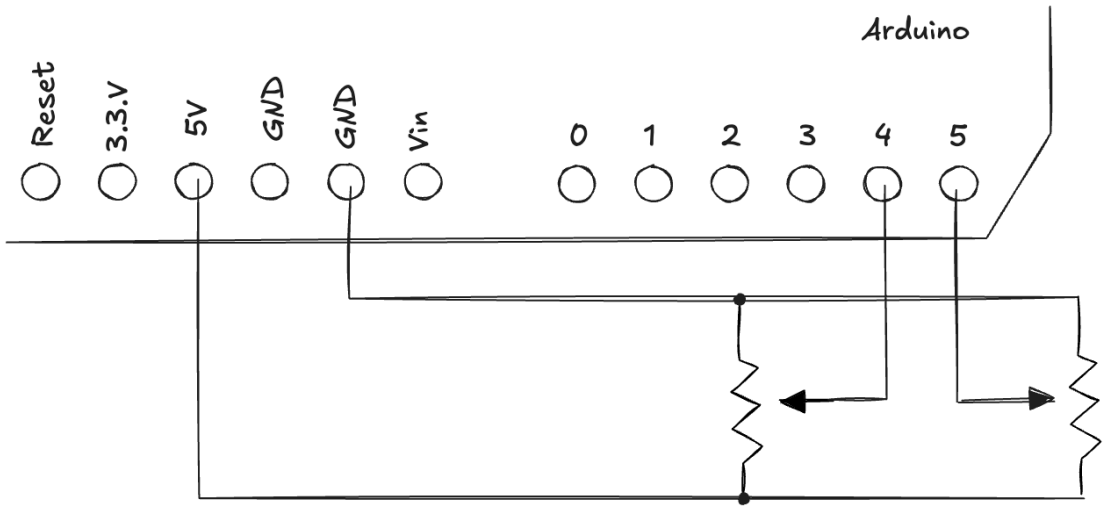
```
import processing.serial.*;

Serial myPort;

void setup() {
  size(200, 200);
  myPort = new Serial(this, Serial.list()[2], 9600);
}

void draw() {
  if (myPort.available() > 0) {
    int value = myPort.read();
    background(255);
    println(value);
    line(0, 226-value, 200, 226-value);
  }
}
```

1.4 Hands-on Activity

1 - Create a simple traffic light system. The system should alternate between green, yellow, and red lights, simulating the behavior of a real traffic light. Solution: *16.TrafficLights*

2 - Enhance the traffic light system by adding pedestrian lights and a push button. When the button is pressed, the pedestrian light will turn green, allowing pedestrians to cross safely. Solution: *17.PedestrianTraffiLights*

3 - Control the computer mouse cursor using two potentiometers connected to an Arduino, see the next figure. You should use Processing to receive data from the Arduino and move the mouse cursor using the Robot class, which generates native system input events.



This technique for controlling mouse is easy to implement and should work with any operating system that can run the Processing application If we require Arduino to actually appear as a mouse to the computer, we have to emulate the actual USB protocol (**Human Interface Devices (HID)**)Solution: *18.Mouse*