ECOLE NATIONALE
SUPÉRIEURE
D'INFORMATIQUE

الجـــمهوريـــة الجزائريـــة الديمـــقراطيـــة الشـعبيـة
وزارة التعليـــم العـــالي والبـــحث العلمـــي
People's Democratic Republic of Algeria
Ministry of Higher Education and Scientific Research

Intelligent and Communicating Systems, ICS

2^{nd} Year Specialty SIQ G02, 2CS SIQ2

LAB report n°3 Part 1

Title:

Arduino Communications

Interrupts-PWM-Sensors-Actuators

Studied by:

First Name: Nour el Imane
Last Name: HEDDADJI
E-mail: jn_heddadji@esi.dz

A. Theory

1. Force Sensor

1.1. Definition

A force sensor is a device that measures the amount of force applied to it. It is also known as a load cell or a force transducer. Force sensors are used in a wide variety of applications, including robotics, industrial automation, medical devices, and consumer electronics

1.2. Principle

The operation of a force sensor depends on its type and design. However, the basic principle behind most force sensors is that they all work by converting mechanical force into an electrical signal. The electrical signal can then be measured and processed by a computer or other device.

1.3. Category

Some common types of force sensors include:

Strain gauge force sensors

These sensors use a thin wire or foil that is attached to a diaphragm. When force is applied to the diaphragm, the wire or foil is stretched or compressed. This changes the electrical resistance of the wire or foil, which can be measured to determine the amount of force.

Figure 1: Strain gauge force sensor

Piezoelectric force sensors

These sensors use a piezoelectric crystal that generates an electrical voltage when it is compressed or stretched. The amount of voltage generated is proportional to the amount of force applied to the crystal.

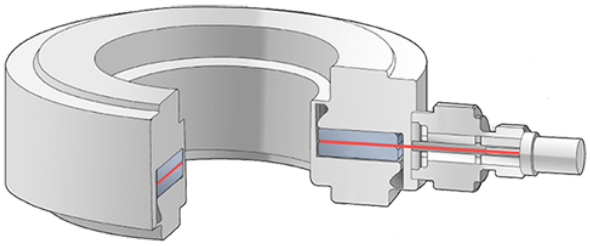


Figure 2: Piezoelectric force sensor

Capacitance force sensors

These sensors use two plates that are separated by a thin dielectric material. When force is applied to the plates, the distance between them decreases. This changes the capacitance between the plates, which can be measured to determine the amount of force.

Figure 3: Capacitance force sensors

1.4. Connection with Microcontrollers

Force sensors can be connected to microcontrollers using a variety of methods. The most common method is to use an analog to digital converter (ADC). The ADC converts the analog output signal from the force sensor into a digital signal that can be read by the microcontroller. To connect a force sensor to an Arduino, connect the output signal to an analog input pin and use the ADC library to read the analog input signal.

2. Interruptions

2.1. Definition

In Arduino, an interruption is a mechanism that allows the microcontroller to temporarily halt its current program execution and switch to executing a specific function or routine

in response to an external event or signal. Interrupts are used for time-critical events, such as reading a sensor or receiving a message from another device.

When an interrupt occurs, the microcontroller saves its current state and jumps to a special function called an interrupt service routine (ISR).

The ISR performs the necessary actions to handle the interrupt, and then the microcontroller returns to its previous state and continues running the main program.

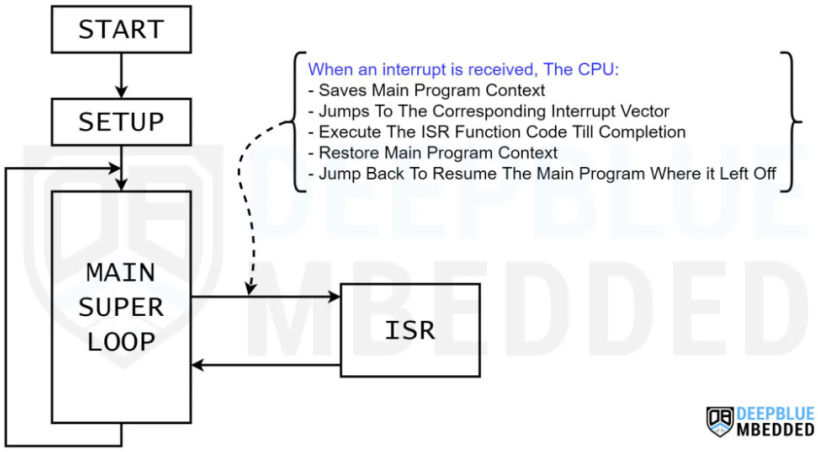


Figure 4: The interrupt handling mechanism in Arduino

2.2. Arduino pins that allow interruptions

Not all Arduino pins support interruptions, generally, Arduino pins that support interruptions are labeled with an "*INT*" or "*tilde*" symbol, and are typically associated with specific interruption types or modes, such as falling edge, rising edge, low level, or high level.

Here is a summarized table for the external interrupt pins available in each Arduino board.

Arduino Board	**External Interrupts Pins**
Arduino Uno, Nano, Mini	2, 3
Arduino Mega	2, 3, 18, 19, 20, 21
Arduino Micro, Leonardo	0, 1, 2, 3, 7
Arduino Zero	All IO pins (except pin 4)
Arduino Due	All IO pins

Table 1: Arduino Board External Interrupts Pins

2.3. Arduino Internal Pull-up

An internal pull-up resistor is a feature available on many Arduino boards that allows you to set a default input state for digital pins.

When you enable the internal pull-up resistor on an input pin, the Arduino will connect the pin to a fixed voltage (typically 5V or 3.3V, depending on the board), which helps to stabilize the pin's state.

This can be useful for reading digital inputs from switches or buttons, for example. To enable the internal pull-up resistor on an Arduino pin, you can use the pinMode() function with the INPUT_PULLUP argument, like this

```
pinMode(pinNumber , INPUT_PULLUP) ;
```

Listing 1: Arduino Internal Pull-up Programming

2.4. Structure of program-based interrupts

The structure of an Arduino program based on interrupt, functions, and libraries can vary depending on the specific application, but generally follows this basic format:

1. Include any necessary libraries.

```

#include <Wire.h>
#include <NinaWIFI.h>

```

Listing 2: Arduino Internal Pull-up Programming

2. Define any global variables.

3. Set up any required hardware configurations or pin modes in the setup() function.

```
void setup () {
pinMode(7 , INPUT) ;
attachInterrupt ( digitalPinToInterrupt (7),interruptHandler ,
CHANGE) ;
}

```

Listing 3: Arduino Internal Pull-up Programming

4. Define any necessary helper functions.

```

void readSensorData () {
// code to read sensor datahere }

```

Listing 4: Arduino Internal Pull-up Programming

5. Define the interrupt service routine (ISR) that will be executed when an interrupt occurs on the designated interrupt pin:

```
1
2
3   void BUTTON_PRESSED_interruptHandler () {
4   // code to handle interrupt here}
5
```

Listing 5: Arduino Internal Pull-up Programming

2.5. Volatile variable type compared with other types

Feature	**Volatile**	**Non-volatile**
Memory location	RAM	Flash memory
Visibility	Can be accessed by interrupts	Cannot be accessed by interrupts
Atomicity	Guaranteed to be updated atomically	Not guaranteed to be updated atomically
Use in Arduino code	Recommended for variables that are accessed by interrupts	Not recommended for variables that are accessed by interrupts

Table 2: Comparison of volatile variable type with other types

Why use volatile type for Arduino code?

Volatile variables are important for Arduino code because they allow interrupts to safely access and update variables. When an interrupt occurs, the microcontroller may save its current state and jump to a special function called an interrupt service routine (ISR). The ISR may then need to access and update variables that are also being accessed by the main program. If these variables are not volatile, then the ISR may overwrite the values that the main program is trying to write.

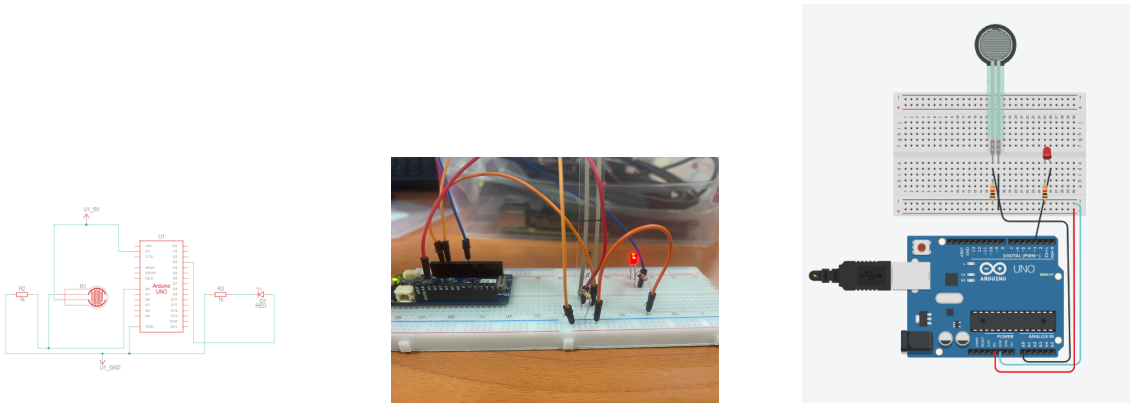
To avoid this problem, volatile variables are guaranteed to be updated atomically. This means that the compiler will not re-order instructions or optimize code in a way that could cause the ISR to overwrite the values that the main program is trying to write.

B. Activity

1. Force Sensor

Hardware

A force sensor and a LED are connected using an Arduino MKR 1010 board. The LED turn on if the force applied exceeds a certain threshold.



(a) Schematic view (b) electronics mounting (c) Circuit view

Figure 5: Force Sensor connection electronics views

Software

```
int force = 0;
int threshold = 20; // Definition of the threshold
void setup()
{
  pinMode(A1, INPUT);
  pinMode(12, OUTPUT);
  Serial.begin(9600);
}
void loop()
{
  force = analogRead(A1);
  Serial.println("Force value: ",force);
  if (force >= threshold) / Comparing the value to the threshold
  {
    digitalWrite(12, HIGH); // Turning on the LED
  }
  else {
    digitalWrite(12, LOW); // Turning off the LED
  }
  delay(1000);
}
```

Listing 6: basic Force sensor program

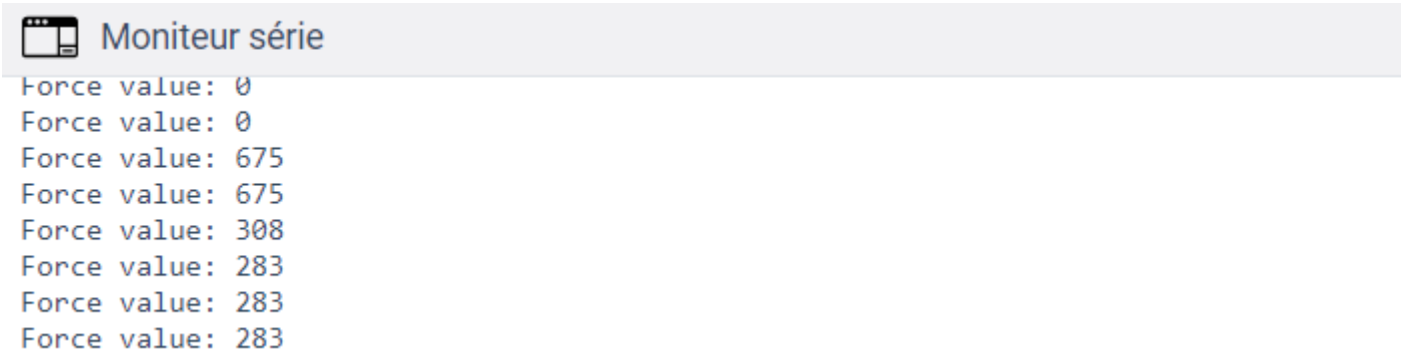


Figure 6: Force sensor connections analysis

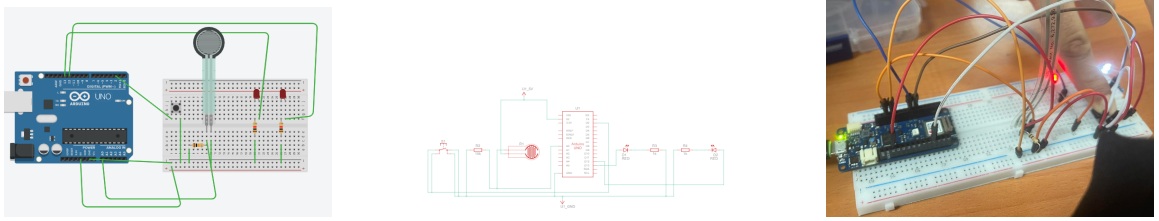
Analysis

The value read from the sensor is printed in the console and the led is lighted on according to this value.

2. Pushbutton with Internal Pull-Up

2.1. Hardware

A second led is added to the circuit , this led is lighted on/off according to the state of the pushButton , the internal pull_up resistance is used.



(a) Schematic view (b) electronics mounting (c) Circuit view

Figure 7: Force Sensor connection electronics views

Software

In the setup fonction we declare our input (sensor and button) and the output (the two leds).

```
int force = 0;
int threshold = 50;
int button = 0;
void setup()
{
  pinMode(A1, INPUT);
  pinMode(2, INPUT_PULLUP);
  pinMode(11, OUTPUT);
```

```
  pinMode(12, OUTPUT);
  Serial.begin(9600);
}
void loop()
{
  force = analogRead(A1);
  Serial.print("Force value: "); // Printing the force value
  Serial.println(force);
  button = digitalRead(2);
  Serial.println(button); // Printing the button state
  if (force >= threshold)
  {
    digitalWrite(12, HIGH); // Turning on the LED}
  else
  {
    digitalWrite(12, LOW); // Turning off the LED}
  if (!(button == HIGH))
  {
    digitalWrite(11, HIGH); // Turning on the second LED}
  else
  {
    digitalWrite(11, LOW); // Turning off the second LED}
  delay(200);
}
```

Listing 7: System saturation without interrupts

Analysis

In the provided code, consistently checking the button state within the loop, especially during delay periods, may cause delays in responsiveness. This could result in situations where the button is pressed, but the LED state doesn't change promptly, leading to potential inconsistencies in event recognition.

3. Pushbutton with Internal Pull-Up

3.1. Hardware and Software

We keep the same circuit from before and also the same code.

Analysis

In the previously provided code, when the push button is pressed multiple times, and the sensor is also pressed, the system occasionally fails to capture and print the output in the console. This is due to the saturation phenomenon

4. Interruptions

Hardware

We keep the same circuit.

Software

The *attachInterrupt* function is used to set up an interrupt on the specified pin (buttonPin). When the state of the button changes, the *buttonInterrupt* function is called.

```
int force = 0;
int threshold = 50;
int buttonPin = 2; // Change this to the correct pin number
int button = 0;
void setup()
{
  pinMode(A1, INPUT);
  pinMode(buttonPin, INPUT_PULLUP);
  // The interrupt declaration
  attachInterrupt(digitalPinToInterrupt(buttonPin), buttonInterrupt,
CHANGE);
  pinMode(11, OUTPUT);
  pinMode(12, OUTPUT);
  Serial.begin(9600);
}
void loop()
{
  force = analogRead(A1);
  Serial.print("Force value: "); // Printing the force value
  Serial.println(force);
  if (force >= threshold)
  { digitalWrite(12, HIGH); // Turning on the first LED
  }
  else
  { digitalWrite(12, LOW); // Turning off the first LED
  }
  delay(200);
}
void buttonInterrupt()
{
  button = digitalRead(buttonPin); // Reading the pushButton state
  Serial.println(button);
  if (!(button == HIGH))
  {
    digitalWrite(11, HIGH); // Turning on the second LED
  }
  else
  { digitalWrite(11, LOW); // Turning off the second LED
  }
}
```

Listing 8: System saturation using interrupts

Analysis

Even when attempting to saturate the system by pressing the button and sensor simultaneously, all values are printed normally. This is because a change in the button state triggers the interrupt routine, pausing the main program and ensuring the prompt processing of the button state. The use of interrupts allows the system to handle simultaneous events efficiently.

C. Conclusion

In this comprehensive lab exploration, we delved into the integration of a force sensor and LED, dynamically adjusting the LED state based on force thresholds.

We then added a pushbutton and another LED, using an internal pull-up resistor this time. We tested the system thoroughly, trying out various button presses, sensor readings, and delays to see how it handles different scenarios.

The highlight came when we faced **system saturation** , where things got tricky. To tackle this, we introduced interrupts – a smart way to deal with interruptions in the system. We learned how interrupts promptly respond to changes, like button presses, even during other activities.

This hands-on experience not only showcased the versatility of force sensors and pushbuttons but also provided a practical understanding of interrupt usage in microcontroller programming.