Co-funded by the
Erasmus+ Programme
of the European Union

UNIT 2: DIGITAL INPUTS/OUTPUTS AND INTERRUPTS

AIMS

We're not going to talk about what digital inputs and outputs are. We're going to assume that you've used them by now and you're familiar with them. What we are going to do is clarify a few ideas on controlling digital devices and explain what pull-up and pull-down resistors are.

We're going to have a close look at interrupts in this unit. Arduino is capable of suspending a program that's in progress and going on to run another one or perform a different task. Once this has been done, Arduino goes back to the original program. All this can happen when input pins detect particular signals; let's have a look any way.

THEORY SECTION

- DIGITAL OUTPUTS; ACTIVE HIGH LOGIC LEVELS
- DIGITALES INPUTS; PULL-UP AND PULL-DOWN RESISTORS
- INTERRUPTS

PRACTICE SECTION

- EXAMPLE 1: Illuminating LEDs
- EXAMPLE 2: Monitoring inputs
- EXAMPLE 3: Monitoring inputs with pull-up resistors
- EXAMPLE 4: Monitoring inputs without interrupts
- EXAMPLE 5: Monitoring inputs with interrupts
- EXAMPLE 6: Controlling two interrupts

PRACTICE MATERIALS

-Lap top or desk top computer

-Arduino IDE work environment; this should include the supplementary material already installed and configured.

-Arduino UNO controller board

-A USB cable

TABLE OF CONTENTS

THEORY SECTION

1. DIGITAL OUTPUTS AND LOGIC LEVELS

It may sound a little far-fetched but it's really not very difficult to understand what we're going to talk about in this section. We assume you're already familiar with the functions related to digital output operation. Let's just go through them again:

- **pinMode(n,OUTPUT)**: this enables us to configure any of the D0:D13 pins as outlets.
- **digitalWrite(n,LOW)**: this sends a logic level of "0" from 0 V through the n pin indicated.
- **digitalWrite(n,HIGH)**: this sends a logic level of "1" from +5 V through the pin indicated.

Now let's just imagine you want to control the illumination of a LED that's connected to the D4 digital output pin. You'd almost certainly write a sequence of functions like this:

```
pinMode(4,OUTPUT);        //Output pin D4

digitalWrite(4,HIGH);     //send level "1" through the D4 pin
```

Why do we always talk about setting a level "1" (+5 V) every time we want to turn something on? Do we only use level "0" (0 V) for turning something off? Can't we do it the other way round? Well, we can! That's right, we can do it the other way round. Both level "1" and level "0" are equally representative. Have a look at the diagrams in Figure 1. What kind of logic level would we have to send through the D4 signal (pin 7) to turn on the LED in the circuit on the left? And what about the LED on the right?

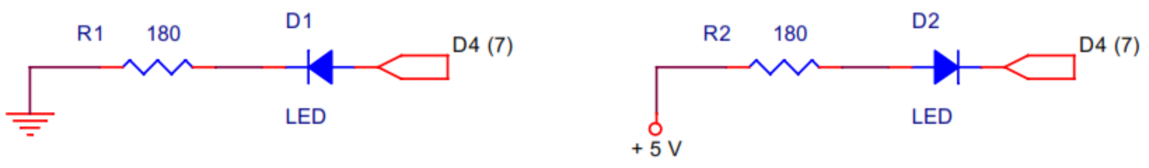


Figure 1

Bearing in mind that the anode has to be positive compared to the cathode to turn on a LED we have to send a "1" (+5 V) through D4 (pin 7) to the anode in the circuit on the left. This is because the cathode is connected to GND (0 V) through the absorption resistor. Nevertheless, we have to send a "0" (0 V) through D4 to the cathode in the circuit on the right as the anode is connected to +5 V through the resistor. Get all that?

We're talking about a LED here, but the same applies to other devices like relay coils, motors, buzzers etc. Have a close look at the diagrams in the following figure. There's just one thing I'd like to mention here: you may notice that the rotational direction of the motor changes when you turn it on with level "1" instead of level "0".

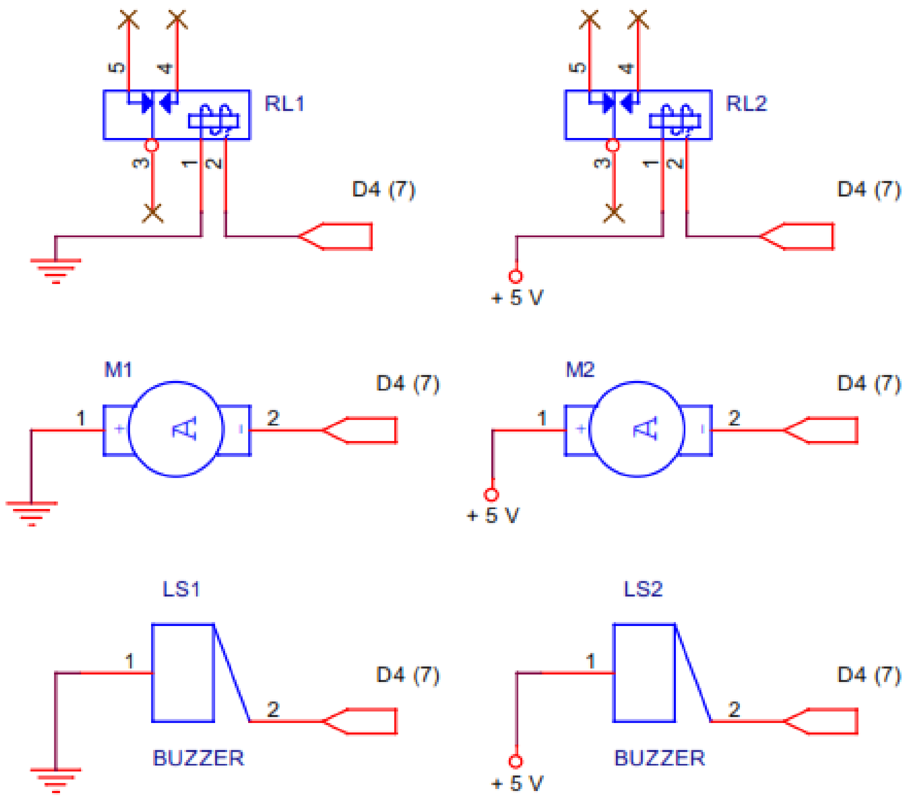


Figure 2

To sum up: there are output peripherals that turn on when you apply level “1” to them and they’re called “active high signals”. There are also output peripherals that turn on when you apply level “0” to them. You’ll come across both types. It basically depends on how the particular peripheral is put together and connected.

2. DIGITAL INPUTS; PULL-UP AND PULL-DOWN RESISTORS

You may well think you know everything about digital inputs but we’re going to look at a problem that you may have encountered but not know why. To start off with, let me remind you that the most basic and economical digital input peripherals are just simple pushbuttons and switches. You’ll come across all kinds of shapes and sizes. Some are designed for industrial use in machine tools, control panels etc...

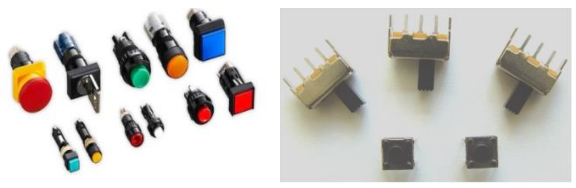


Figure 3

There are others that are much simpler (and cheaper); they're widely used in the home, in education, to solder things on to printed cards, in small appliances etc…We're going to use some small pushbuttons like the two you can see in the bottom part of the photo.

In any case, it doesn't really matter what they're used for because, the way they work is fairly simple. When they're triggered, a metal sheet opens and shuts an electrical contact between two or more pins in the device. We say that an interrupt or commutator is "embedded". When you turn it on, the switch stays where it is until you move it again. Light switches are like that but pushbuttons aren't: they open and close one or more circuits only when turned on. After that they return to their rest position when you release them - like the door bell at home for example. You can see a diagram of a sliding commutator and a pushbutton in the following pictures. You can also see their respective electrical symbols.

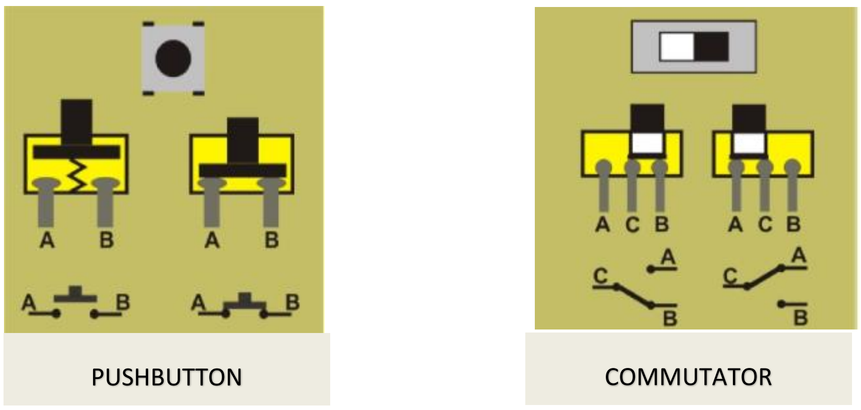


Figure 4

The commutator in the Figure 4 has three pins; one of them is common to the other two pins. When we slide it to the right the sheet closes the circuit between the C and B pins. The C-A circuit remains open but not connected. If we slide it to the left the C-A circuit closes and the C-B circuit remains open.

The pushbutton has two pins, A and B. We say that in rest position it's usually open. The pins aren't connected. When you press it you close the circuit and both pins are connected. When you release it a spring pushes the metal sheet back to its rest position. For your information there are also pushbuttons that are usually closed and only open when pressed. (Figure 4)

I'm sure you're already familiar with the two Arduino language functions for controlling digital inputs:

- **pinMode(n,INPUT)**: this enables us to configure any of the D0:D13 digital pins as inputs. In actual fact you don't really have to do anything: whether you connect the power or press RESET every time you boot the system up all the pins default to inputs.
- **digitalRead(n)**: this reads the logic level of whatever n pin you like.

Nevertheless, have a think about the circuits in the following diagrams (Figure 5). A pushbutton has been connected to D2 (pin 5) which is apparently configured as an input.

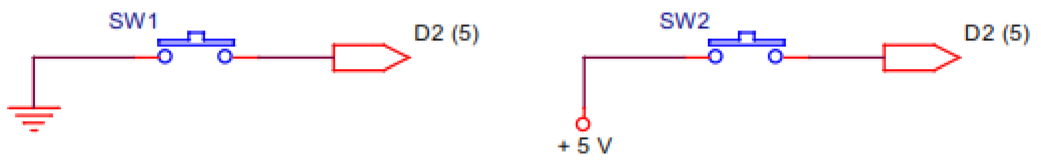


Figure 5

It's quite clear that each time you press the pushbutton in the circuit on the left the D2 pin connects up with GND (0 V) and is therefore at "0" level. The one on the right is the other way round. When you press the pushbutton it connects up with +5 V and therefore is at level "1".

And now for the million dollar question: what happens to the pins if we don't press either of the pushbuttons in the two circuits? I know some of you will say "just the opposite to when you press them". In other words, if we don't press the pushbutton in the circuit on the left the D2 pin remains at level "1" and the one on the right at level "0".

Wrong! Now let's have a look: if the pushbutton isn't pressed in either of the two circuits, what's the difference as far as the D2 pin is concerned? Absolutely none. The D2 remains disconnected in both cases. We say it's "floating". Which one wins: level "1" or level "0"? There's really no definite answer; sometimes the "1" comes out on top and other times the "0". Understandably this is not an ideal situation.

As far as the electrics are concerned, the best thing to do is add a resistor to the circuit. If it's connected to the positive +5 V it's called a resistor or "***PULL-UP"***; if it's connected to GND or 0 V, it's called "***PULL-DOWN***". Have a look at these two circuits (Figure 6).

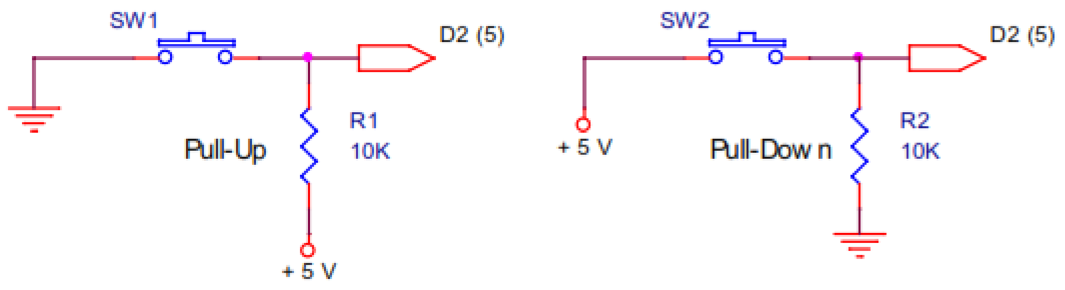


Figure 6

In the one on the left, the R1 pull-up resistor keeps the D2 (pin 5) signal at level "1" (+5 V) when the SW1 pushbutton isn't pressed, or in other words, it's open. When the D2 pin is pressed it goes to level "0". In the circuit on the right, the R2 pull-down resistor holds down the pin to level "0" (GND) when the SW2 pushbutton isn't pressed. When the D2 pin is pressed it goes to level "1".

Okay, but what value should we assign to the resistors? It's not all that important. Remember that each time you press a pushbutton, there's a flow of electrons between GND and +5 V, or, in other words, current (I) goes round the circuit. We should try and minimize this current or consumption. For instance, if you put 100 Ω through R1 or R2, the consumption will be this:

$$I = \frac{V}{R} = \frac{5}{100} = 0.05\ A = 50\ mA$$

If you choose a value of 10 KΩ, as in the example, the consumption will be this:

$$I = \frac{V}{R} = \frac{5}{10000} = 0.0005\ A = 0,5\ mA$$

But, be careful, you mustn't overdo it. You might be tempted to use a powerful resistor to reduce the consumption to a minimum. What happens with such a high value is that the D2 pin is isolated from the +5 V (in the case of the pull-up) or GND (in the case of the pull-down) and doesn't stay put. If this happens you're back to where you started from so it's better not to use any resistor at all. Common values lie between 4700 and 10 KΩ.

Once again we should stress that the active level – when we press the pushbutton – doesn't necessarily have to be "1"; it could also be "0". It all depends on how you connect that pushbutton.

How does Arduino help us? It provides us with the pull-up resistor and this saves you having to install one externally in your circuits. Have a look at the Figure 7: the resistor has been integrated in the Arduino controller.

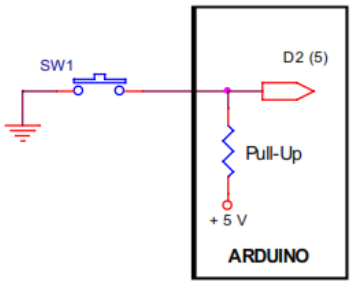


Figure 7

You can configure this option using this function:

- **pinMode (n, INPUT_PULLUP)**: where n represents the input pin you want to link to the corresponding pull-up resistor.

You can do this with any one of the D0:D13 pins that's going to be configured as an input. It doesn't make any sense to apply this to the outputs and you don't have the chance of configuring an input as a pull-down resistor either.

Maybe you don't think it's important to save an input resistor. They only cost a few eurocents after all. But think of it from a business point of view: if your project needs a number of pull-up resistors you're not only going to save on them but also on the following:

- You'll save on space because you won't have to keep them in stock.
- You'll save on the size of the printed circuit boards. They're sold by the square decimetre and the more components there are, the bigger the board and the more you pay.
- You'll save time on the design of the tracks for the printed circuit boards: the more components you use, the more time you'll need to design them. Time is money.
- You'll save time assembling the circuit board. The more components you use, the longer it'll take you to assemble.

If you multiply all this by the number of circuit boards or kits you're going to sell the saving could be tens or hundreds of euro.

Now you know how, use the pull-up function whenever you think it's necessary.

3. INTERRUPTS

Now let's have a look at another topic : The interrupts. With a name like that maybe you think it means stopping the controller: pausing it so that it ceases to execute the program; actually it's got nothing to do with it.

Imagine a machine tool that's making parts. Suddenly the alarm goes off for one of the following reasons:

- A part poorly positioned and as a result the finished product will be defective.
- The motor that drives the conveyer belt has broken down and the parts are piling up.
- A sensor detects that a component of the machine is overheating.
- A worker sticks his hand in the machine where he's not supposed to.

What should the controller do to stop this malfunction?

a) Nothing special; continue to execute the usual program as if nothing had happened.

b) Come to a halt and stop executing the program immediately.

c) Stop executing the usual program and execute another one that deals with the causes that set the alarm off by issuing warnings and carrying the necessary tasks: removing the defective part, stopping the manufacture of parts, turning on a cooling system, give the worker a warning signal, etc…

The answer's pretty obvious, isn't it? An interrupt doesn't necessarily mean that the controller stops altogether; it drops the task it's doing at that moment and executes another one.

How do we cause an interrupt? There are lots of different events that can trigger one off. It all depends on what model of controller we're talking about. The most common way is for external peripherals to send signals through certain controller pins. In the case we're talking about, Arduino UNO has two different interrupt sources or pins: D2/INT0 and D3/INT1.

I'm sure you'll remember that you've used these pins on a number of occasions and they've never caused any interrupt. They're actually digital input/output pins just like all the other D0:D13. It's true. The thing is that for the D2 and D3 pins to work as INT0 e INT1 interrupt inputs, they have to be configured the right way and be equipped with a special kind of authorization. Causing an interrupt isn't just a detail. And this is exactly what you're going to learn in this chapter.

What happens when the controller receives an interrupt? Have a close look at the Figure 8; it shows how the process works:

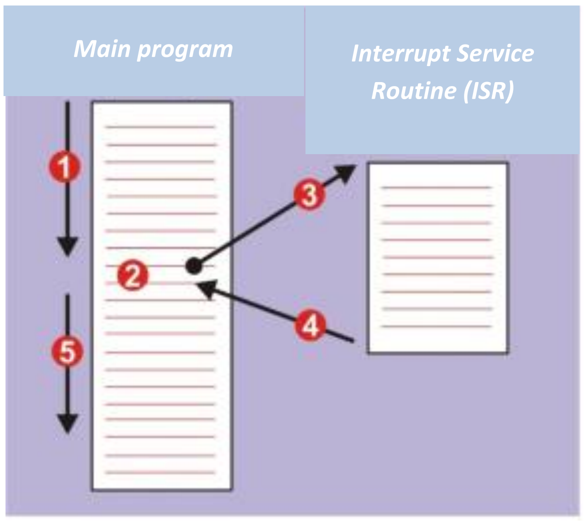


Figure 8

1. The controller is executing its usual main program.

2. At some point a peripheral requests and causes an interrupt.

3. The controller suspends the main program and goes on to execute a program to deal with the event, also known as an ISR (Interrupt Service Routine).

4. Once the execution of the interrupt handler is finished, the controller returns to the main program.

5. The execution is renewed from where it left off.

What's so new about all this? To manage and use the Arduino UNO interrupts, you've got four functions at your disposal:

- **Function attachInterrupt()**

It configures how an interrupt should work.

Syntax:

attachInterrupt(pin, ISR, mode);

pin: This represents the pin that's going to be configured as an interruption input. In the case of Arduino UNO it could be INT0 (D2) or INT1 (D3) as well.

ISR: this is the name of the routine or function that must be executed each time the interrupt happens.

mode: This is when the interrupt should be triggered:

LOW: when the pin sends through a "0" level.

CHANGE: when a change of status in the pin is detected.

RISING: when the pin detects a rising edge ("0" -> "1").

FALLING: when the pin detects a falling edge ("1"-> "0").

- **Function detachInterrupt()**

Turns off an interrupt.

Syntax:

detachInterrupt(pin);

pin: this represents the pin interrupt to be turned off. In the case of Arduino UNO this might be INT0 (D2) or INT1 (D3) as well.

- **Function interrupts()**

This turns on the interrupts. Think of it as a kind of general authorization for the interrupts that have been previously configured through attachInterrupt().

Syntax:

interrupts();

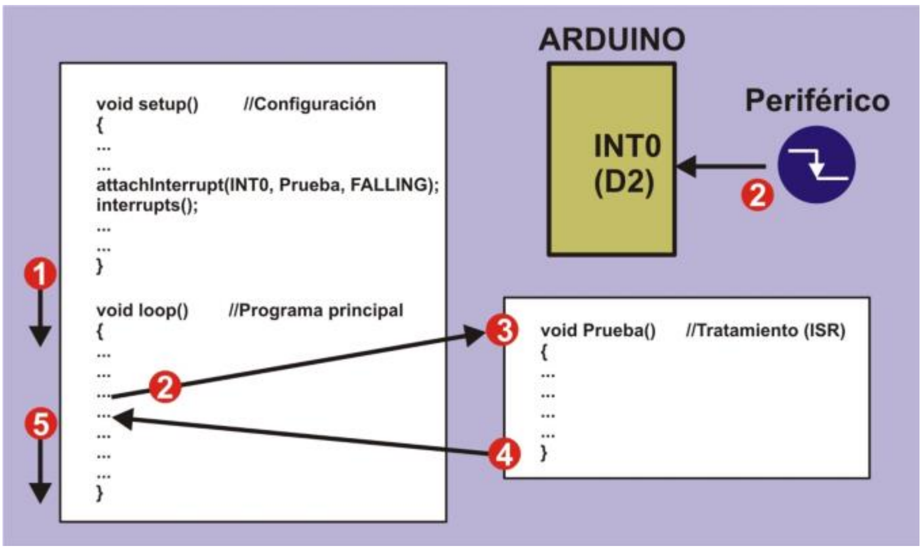
- **Function noInterrupts()**

This turns off all the interrupts. Think of it as a kind of general prohibition that prevents all the interrupts from functioning.

Syntax:

noInterrupts();

The following Figure 9 may give you an idea of what a program that uses interrupts might look like.



```
void setup()      //Configuración
{
...
...
attachInterrupt(INT0, Prueba, FALLING);
interrupts();
...
...
}

void loop()      //Programa principal
{
...
...
...
...
...
...
...
}

void Prueba()      //Tratamiento (ISR)
{
...
...
...
...
}
```

Figure 9

Each time a falling edge is detected the D2 (INT0) pin configures the interrupt in the configuration function **setup()** using **attachInterrupt()**. If this happens the **Test()** function that corresponds to the ISR will be executed. Almost at once **interrupts()** issues general authorization. This is what happens:

1. It starts to be execute the main **loop()** program.

2. At some point a peripheral issues an interrupt request with a falling edge through the INT0 (D2) pin. The controller stops executing the program.

3. The controller executes the **Test()** function (Interrupt Service Routine).

4. Once the **Test()** function has been executed the controller goes back to the main program.

5. The controller renews the execution of the program from where it left off.

Limitations

Because of the very nature of Arduino, its interrupt system is a bit limited. One example is that the well-known delay() and millis() functions don't work if they're already in the interrupt service routine. This is because these functions use their own internal interrupt system and Arduino can't respond to an interrupt from another system.

You can't transfer parameters to an interrupt service routine and the ISR can't return anything either. If you did need it, you'd have to use global variables to transfer the data to be used by the main program and the interrupt service routine function.

Remember that an interrupt is an event that may or may not happen from time to time. These alarm situations may or may not occur. A peripheral may or may not send the interrupt signal or not. It's a good idea to execute the interrupt service routine function as quickly as possible. Remember that during the execution of the interrupt service routine function, the controller suspends the execution of the main program.

Advantages

It's got advantages too. Generally speaking a program that uses interrupts is a lot more efficient. You do have to be careful about how you use them. Imagine that you're doing a project and you have to carry out a certain task every time the peripheral sends a signal.

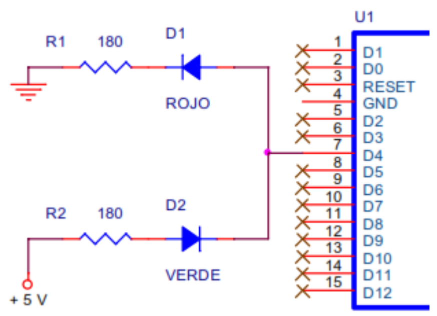
You've got two options: one via the digitalRead() function, which reads and waits for the signal to be triggered. Remember that while you're waiting for the signal you can't do anything else.

The other solution - the best one really – is to use an interrupt. Your main program can be doing whatever you like but when the signal reaches it, then, and only then, the controller "responds" to the interrupt and performs the corresponding task. What this means is that there's no down time because the controller is always doing something useful- as if it were doing a couple of tasks at the same time.

PRACTICE SECTION

4. EXAMPLE 1: ILLUMINATING LEDS

You’re not going to learn anything new about programming from this example. By this time you should know how to turn LEDs on and off. What you are going to learn here is how to do the electrical installation to check that a LED or other peripherals can be turned on after receiving a logic level “0” or a logic level “1”. Have a look at the diagram in the figure.

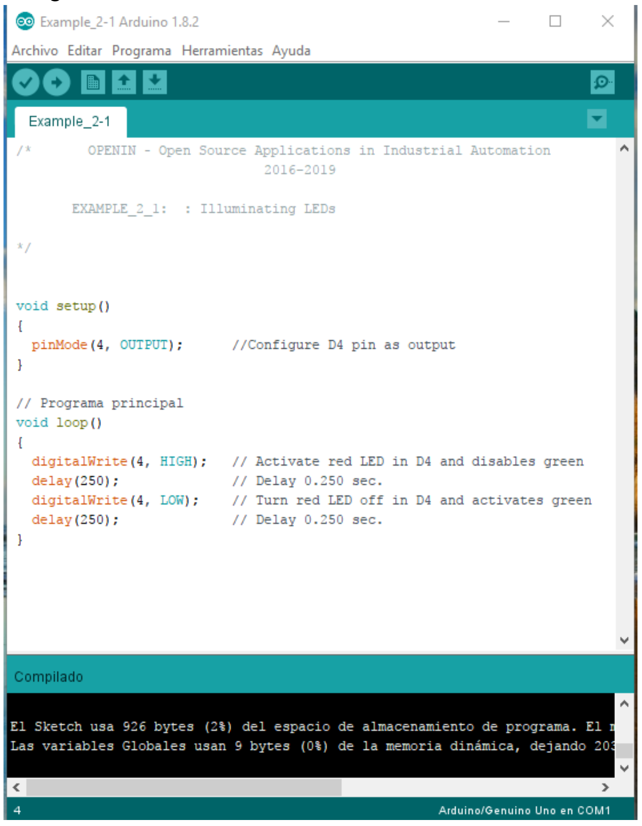


The same D4 output pin will act on both LEDs. But notice that it’s the red LED anode with the cathode that goes to GND that’s connected. Therefore this LED will go on when D4=”1”. On the other hand, D4 is connected to the cathode of the green LED with the anode that goes to +5 V; this LED therefore goes on when D4=”0”.

The program is very simple. The D4 pin is configured as an output in the setup() function.

The main program is a continuous loop. The D4 pin is at level “1”. The red LED goes on while the green one goes off.

Timed at 0.25 seconds, the D4 pin is set at level “0”. The red LED goes off and now the green LED goes on. After another 0.25 seconds, the cycle is repeated.


Example_2-1 Arduino 1.8.2
Archivo Editar Programa Herramientas Ayuda
Example_2-1

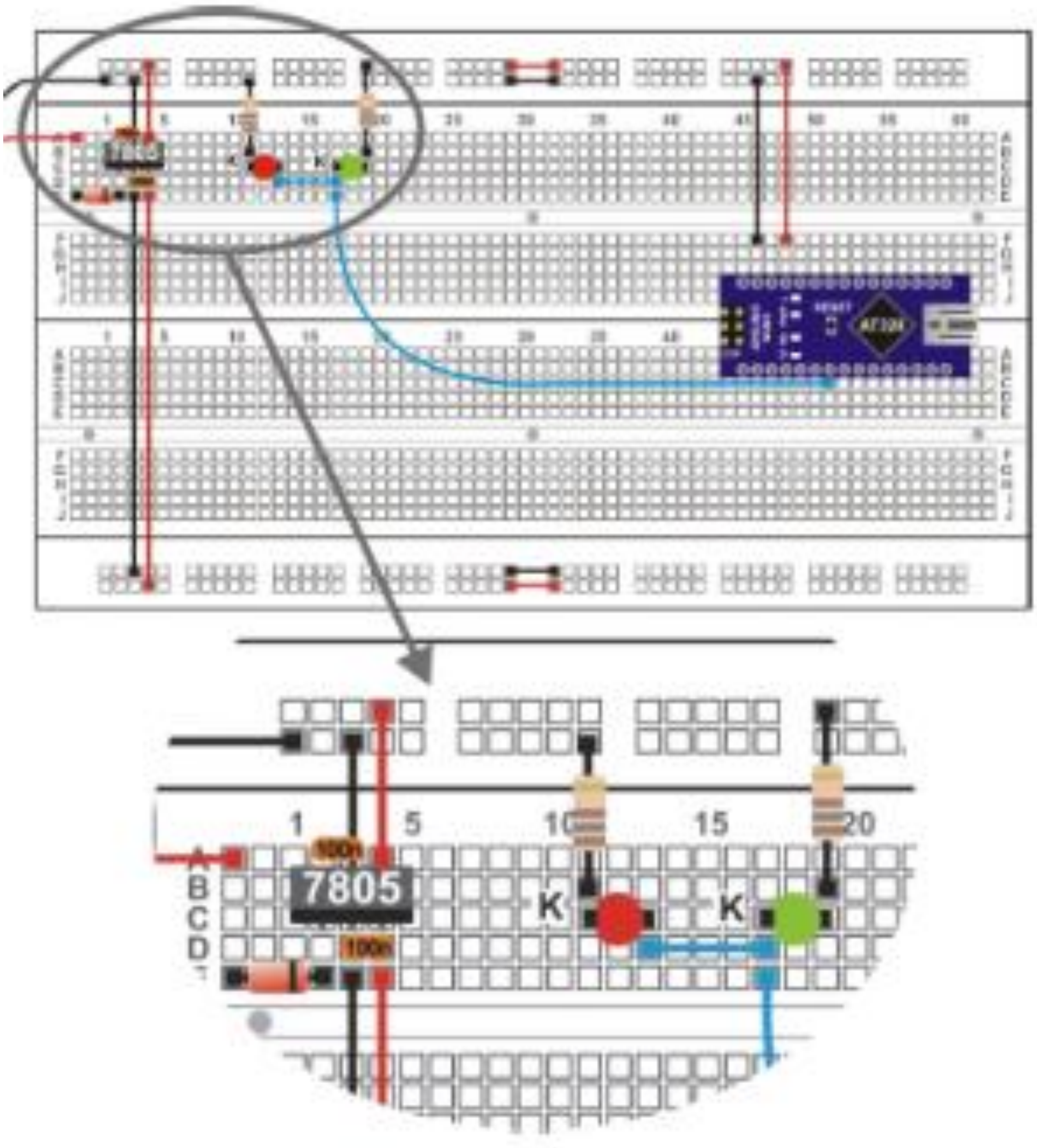
```
/*       OPENIN - Open Source Applications in Industrial Automation
                                2016-2019

      EXAMPLE_2_1:  : Illuminating LEDs

*/


void setup()
{
  pinMode(4, OUTPUT);       //Configure D4 pin as output
}

// Programa principal
void loop()
{
  digitalWrite(4, HIGH);    // Activate red LED in D4 and disables green
  delay(250);               // Delay 0.250 sec.
  digitalWrite(4, LOW);     // Turn red LED off in D4 and activates green
  delay(250);               // Delay 0.250 sec.
}
```

Compilado

```
El Sketch usa 926 bytes (2%) del espacio de almacenamiento de programa. El r
Las variables Globales usan 9 bytes (0%) de la memoria dinámica, dejando 203
```

4 Arduino/Genuino Uno en COM1

Here's an idea of what the practice assembly looks like on the module board. Make sure you pay attention to the layout and positioning of the components.



Now it’s your turn.

The program is very simple so you’re not going to change it. One thing you could do though is change the value of the green LED R2 absorption resistor. Its value is 180 Ω, so change it to 10 KΩ.

What do you notice?

-

How come?

-

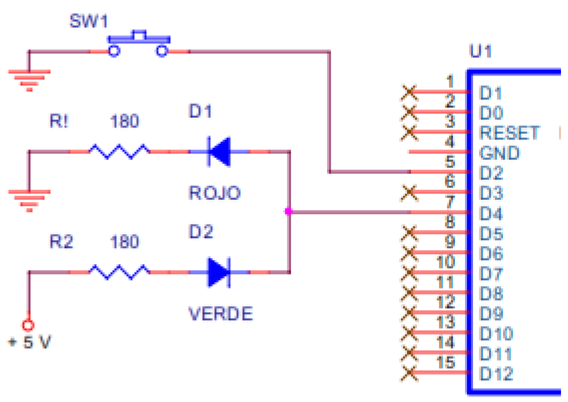
Try and work out how much current is going through the green LED at the moment:

$$I = \frac{V}{R} = \frac{V - V_{AK}}{R} = \frac{5 - 1.5}{10000} = 0.00035\ A = 0,35\ mA$$

You can see now that the current (I) that’s going through the LED is 0.35 mA when the manufacturer suggests it should be approximately 20 mA. Now you know another way to regulate the brightness of an illuminated LED. Put in the R2 resistor again with its original value of 180 Ω.

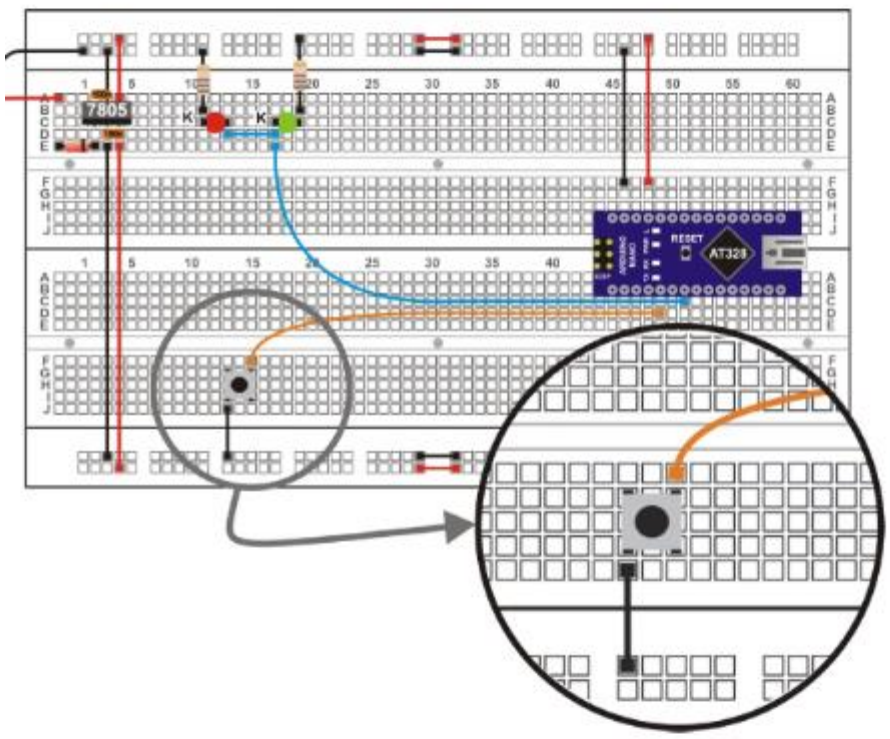
5. EXAMPLE 2: MONITORING INPUTS

Here's the diagram. It's exactly the same as the one in the previous example except for an SW1 pushbutton that's been connected to D2.



This example doesn't tell us anything new about programming either. What you have to do is read the logic level of the D2 input pin and apply it to the red and green LEDs. If the input is on "1" the red one goes on and if it's on "0" the green one goes on. Both the LEDs are controlled by the D4 output pin.

I suggest you use some flat-pointed pliers to straighten the pins on the pushbutton as shown in

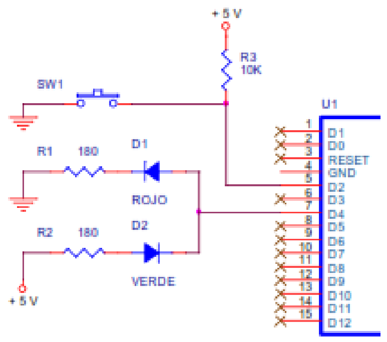


the figure and insert the pushbutton properly on the module board. The pins should be completely perpendicular to the body of the pushbutton. The following figure shows what the practice assembly looks like. You should gradually familiarize yourself with the circuit assembly on the module board.

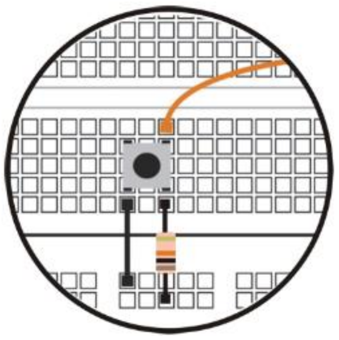
Now it's your turn

As I said before, there's nothing in the program in the example _2-2 that you don't already know. All you have to do is record it on Arduino UNO and make sure it works properly. Maybe you'll get a surprise.

It's obvious that when you press the pushbutton you send a level "0" through the D2 pin and the green LED should go on. But if you don't press it, what do you notice? And another thing: if you touch the pushbutton wire or the LED wire or even the Arduino UNO controller pins with your fingers, what happens?



It looks like the LEDS go on at random. This is because when you don't press SW1, the D2 input is "floating" and doesn't have a definite logic level. Have another look at 2.2 in the theory section.

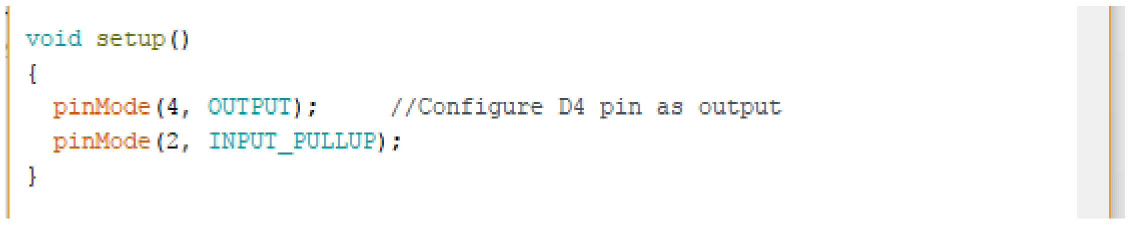


Change the electrical circuit slightly by adding a 10 KΩ resistor as shown in the theoretical diagram and in the one in the electrical installation. We're talking about the pull-up resistor here. This resistor makes sure the D2 pin remains anchored on level "1" whilst the pushbutton hasn't been pressed.

Make sure that the same example_2-2 works properly without modifying it in any way and that the LEDs light up completely normally. You'll see that the detail is important.

6. EXAMPLE 3: MONITORING THE INPUTS WITH PULL-UP RESISTORS

This example provides us with a definitive solution to the pull-up resistors. The D2 pin is configured as an input with a pull-up resistor so you don't have to add an external one. And now the electrical circuit looks exactly like the one in the first diagram from the previous example. The only thing that's different is the program.



```
void setup()
{
  pinMode(4, OUTPUT);      //Configure D4 pin as output
  pinMode(2, INPUT_PULLUP);
}
```

The D2 pin is configured so that it works like an input with a pull-up resistor by using the pinMode(2, INPUT_PULLUP) function that's in setup().

Before you record the program pull out the 10 K pull-up resistor you installed in the previous example. There can't be more than one resistor and in this case Arduino already provides it.

Now record the program and make sure everything works properly. The LEDs connected to the D4 output monitor the logic status of the pushbutton connected the D2 pin. The red one comes on when the pushbutton hasn't been pressed (level "1" due to the internal pull-up resistor) and the green one when it's pressed (level "0").

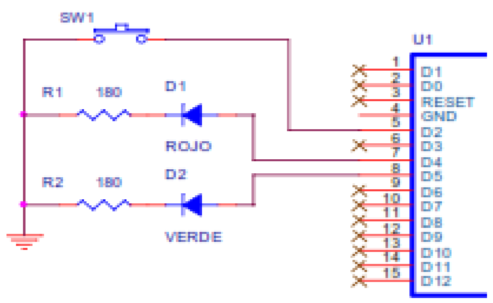
To sum up the first three examples we can draw the following conclusions:

- Any digital peripheral output can be turned on using level "1" or level "0"; it all depends on how it's connected.
- In just the same way, the digital peripherals may generate a level "1" or even a level "0" when they're pressed. It also depends on how they're connected.
- Level "0" is just as valid as level "1". They're both just as representative.
- Some input peripherals may or may not include a pull-up resistor. If they don't, you'll have to act accordingly and install an external one yourself or use the Arduino UNO.

7. EXAMPLE 4: MONITORING INPUTS WITHOUT INTERRUPTS

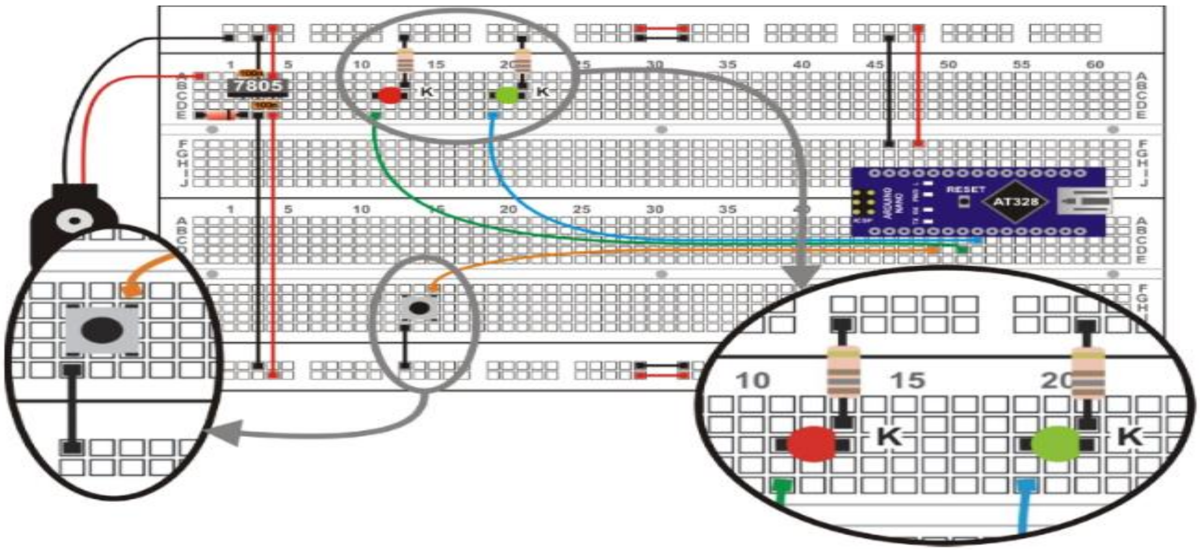
Once again here’s an example that isn’t going to show you anything that you didn’t already know about programming but all the same it’ll enable you to assess whether or not it’s a good idea to use interrupts or not.

A red LED has been connected to the D4 output pin and it’ll constantly monitor the status of the input pin connected to D2. The idea is that the other LED, the green one, connected to the D5 output, flashes on and off every five seconds at the same time. Sounds easy, doesn’t it?



The figure shows the circuit diagram. The red LED is controlled from D4; it monitors the status of the pushbutton connected at D2. The green LED is controlled from D5 and flashes on and off every five seconds. Note that both LEDs go on at level “1”. This is the practice set up you have to do on the module board.

Once you’ve done the setup and recorded the program, make sure it works properly.



The red LED that monitors the status of the pushbutton looks like it’s on “delay”. And sometimes, if you press the pushbutton really quickly, the red LED “doesn’t even notice”. Why is that?

Whilst the controller is executing one of the delay(500) functions it can’t execute the digitalWrite(4, digitalRead(2)) function at the same time. In other words, it can’t keep up with what’s happening to the D2 input pushbutton and so it can’t monitor its status at D4 either.

In fact, the controller only monitors each time an on and off cycle from the D5 output finishes; this happens every one second. Do you get it? Let’s try and solve the problem in the following example by using an interrupt.

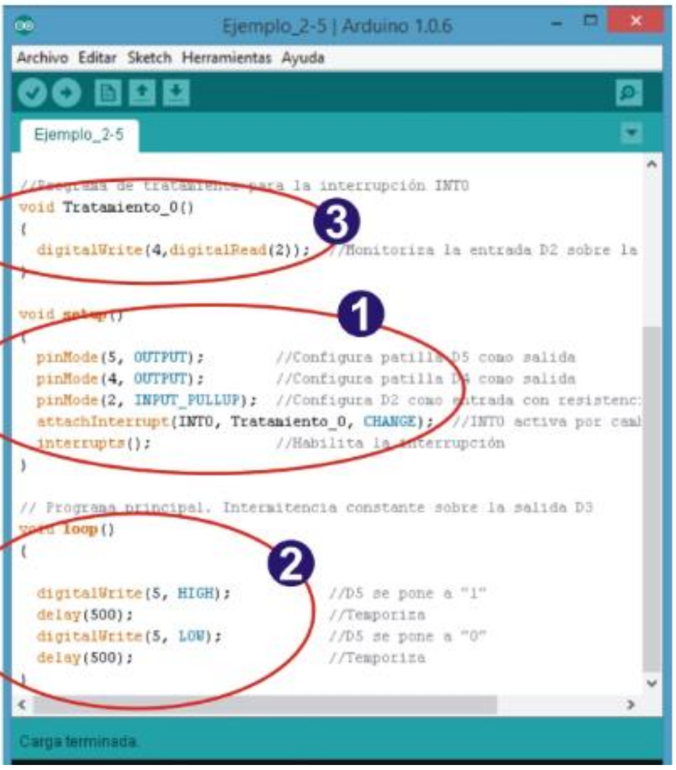
8. EXAMPLE 5: MONITORING USING INPUTS WITH INTERRUPTS.

The example sets out to do exactly the same as the last one but properly this time. The LED connected to the D5 output goes on for half a second and goes off for half a second. At the same time the D4 output monitors the logic status of the D2 input.

Use the same circuit diagram and the same practice circuit. Change the program that has three really different sections. Have a look at the following figure:

Setup():

The D4 and D5 pins are configured as outputs. D2 is configured as an input with a pull-up resistor. The interrupt is configured by using the attachInterrupt() function. This is associated with the INT0 (D2) pin; it goes on when a change of status is detected in the pin and each time this happens, the Treatment_0() function is executed. The interrupts() function enables and authorizes the interrupts, which in this case is INT0.


Ejemplo_2-5 | Arduino 1.0.6
Archivo Editar Sketch Herramientas Ayuda
Ejemplo_2-5
3
1
2
Carga terminada.

```
//Programa de tratamiento para la interrupción INT0
void Tratamiento_0()
{
  digitalWrite(4,digitalRead(2));   //Monitoriza la entrada D2 sobre la
}

void setup()
{
  pinMode(5, OUTPUT);           //Configura patilla D5 como salida
  pinMode(4, OUTPUT);           //Configura patilla D4 como salida
  pinMode(2, INPUT_PULLUP);     //Configura D2 como entrada con resistenc
  attachInterrupt(INT0, Tratamiento_0, CHANGE);   //INT0 activa por cam
  interrupts();                 //Habilita la interrupción
}

// Programa principal. Intermitencia constante sobre la salida D3
void loop()
{

  digitalWrite(5, HIGH);             //D5 se pone a "1"
  delay(500);                        //Temporiza
  digitalWrite(5, LOW);              //D5 se pone a "0"
  delay(500);                        //Temporiza
}
```

loop(): This is the main program. It causes intermittence in the D5 output. The LED goes on for half a second and goes off for half a second; nothing new there.

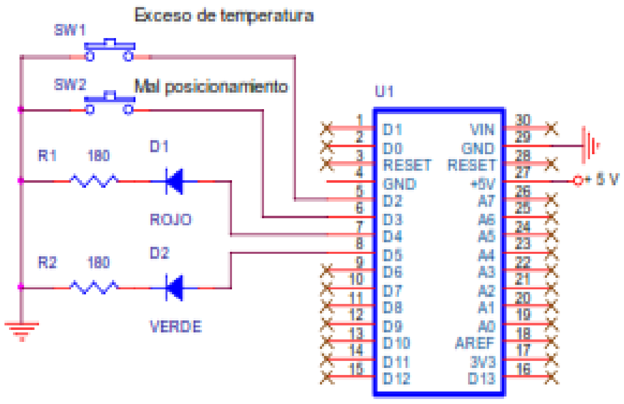
Treatment_0():

This is the program that treats the interrupt service routine or ISR. Every time a change of status in the INT0 (D2) pin is detected, the controller stops what it's doing and executes the Treatment_0() function. This function reads the status of the D2 input and monitors the D4 output. When it's finished this it goes back to the main program and starts where it left off.

Now when you record the program you'll be able to check that the controller instantaneously monitors the red LED that registers the input status. In spite of this, the green LED keeps flashing on and off as if nothing had happened. Remember that the treatment program – the one that monitors the input status – is executed in a few microseconds and doesn't have any appreciable effect on the main program.

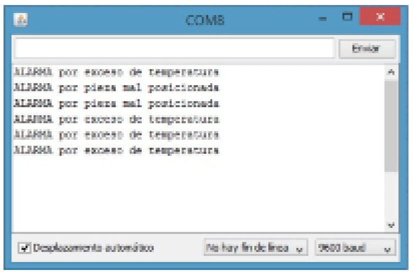
9. EXAMPLE 6: CONTROLLING THE TWO INTERRUPTS

We're going to finish off Unit 2 with another example that uses the two interrupts available in Arduino UNO: INT0 (D2) and INT1 (D3). Imagine a machine tool that's responsible for controlling the two outputs, D4 y D5, that follow a definite sequence.



Two sensors detect two possible alarm situations: overheating and a poorly positioned part. The sensors, which are co-ordinated by the SW1 and SW2 pushbuttons, are connected to the D2 and D3 pins; they trigger the INT0 and INT1 interrupts respectively. Have a look at the circuit diagram in the figure.

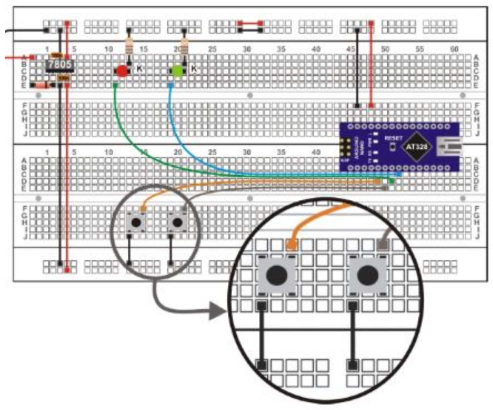
The program turns the red and green LEDs on and off in sequence. When one of the interrupts is detected, the appropriate treatment program transmits a serial warning message like the ones you can see in the window attached to the serial communication monitor.



```
ALARMA por exceso de temperatura
ALARMA por pieza mal posicionada
ALARMA por pieza mal posicionada
ALARMA por exceso de temperatura
ALARMA por exceso de temperatura
ALARMA por exceso de temperatura
```

Both interrupts are configured to be falling edge active. This happens each time you press the appropriate pushbutton. Remember that the D2 (INT0) and D3 (INT1) pins are configured as inputs with pull-up resistors. When the pushbuttons aren't down, the pins are on "1".

Here you can see the practice assembly. It's very similar to the previous example. The SW2



pushbutton connected to the D3 (INT1) has been added.

Once you've recorded the program check that it works properly. Open the serial monitor. The main program limits itself to turning the red and green LEDs on and off without any of the pressbuttons being pushed. If you want to, you can change it.

If any one of the pressbuttons is pressed, you'll see the appropriate message in the window of the serial monitor.

We've seen that from time to time when either of the two interrupts is triggered, the system closes down. You have to reboot it by pressing RESET. This happens because of the "knock on" effect of the two pressbuttons. And that's what happens: even though you only press it once, it can be read as several consecutive interrupts and Arduino is unable to treat them the right way. We've also seen in the laboratory that if the interrupt or the alarm signals come from a generator that doesn't have any knock on effect the system functions properly and doesn't shut down. We've triggered up to 10 interrupts a second for five minutes and the system has always stood up to it.

You can experiment with other peripherals besides press buttons. Time to get down to work!

REFERENCES

BOOKS

[1]. **EXPLORING ARDUINO**, Jeremy Blum, Ed.Willey
[2]. Practical ARDUINO, Jonathan Oxer & Hugh Blemings, Ed.Technology in action
[3]. Programing Arduino, Next Steps, Simon Monk
[4]. Sensores y actuadores, Aplicaciones con ARDUINO, Leonel G.Corona Ramirez, Griselda S. Abarca Jiménez, Jesús Mares Carreño, Ed.Patria
[5]. ARDUINO: Curso práctico de formación, Oscar Torrente Artero, Ed.RC libros
[6]. 30 ARDUINO PROJECTS FOR THE EVIL GENIUS, Simon Monk, Ed. TAB
[7]. Beginning C for ARDUINO, Jack Purdum, Ph.D., Ed.Technology in action
[8]. ARDUINO programing notebook, Brian W.Evans

WEB SITES

[1]. https://www.arduino.cc/
[2]. https://www.prometec.net/
[3]. http://blog.bricogeek.com/
[4]. https://aprendiendoarduino.wordpress.com/
[5]. https://www.sparkfun.com