UNIT 3: EXPRESSIONS, PAUSES AND SOUNDS

AIMS

The first thing we're going to do is to take a closer look at variables, constants and expressions in general. We talked a little bit about them in the last unit but now it's time to really nail them down.

I'm also going to introduce you to some new functions from the Arduino programming language; they'll enable you to create pauses and sounds. You'll be able to use to them to create considerably more versatile and interesting programs.

THEORY SECTION

- EXPRESSIONS
  - Constants
  - Variables
  - Variable Scope
  - Operations
- TIMINGS
  - The delayMicroseconds() function
  - The delay() function
  - The micros() function
  - The millis() function
- SOUND
- SOUND FUNCTIONS

PRACTICE SECTION

- EXAMPLE 1: Flashing lights
- EXAMPLE 2: Sets of lights
- EXAMPLE 3: The V1 Traffic signal
- EXAMPLE 4: The electric bell
- EXAMPLE 5: Melodies
- EXAMPLE 6: The V2 Traffic signal

"The European Commission support for the production of this publication does not constitute an endorsement of the contents which reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein."

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

"The European Commission support for the production of this publication does not constitute an endorsement of the contents which reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein."

TABLE OF CONTENTS

"The European Commission support for the production of this publication does not constitute an endorsement of the contents which reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein."

THEORY SECTION

1. EXPRESSIONS

By this time, you've probably realized that all the Arduino language functions you've studied until now need one or more parameters to work. They go inside the function brackets and there are commas like this "," between them. Let's just go over them again:

- **pinMode(pin, mode)**: you have to specify the number of the pin you're referring to and whether it's an input or an output.
- **digitalRead(pin)**: you have to specify which number pin you want to read.
- **digitalWrite(pin,value)**: you have to specify which number pin you want to use to establish the value.

You can express these parameters in different ways which is exactly what you did in some of the examples in the previous unit. The way you indicate these parameters is called an "expression". There are three basic types:

- **Constant**: we establish the value of the parameter in the function itself. For example, **digitalRead(4)** reads the value of input pin number 4.
- **Variable**: You've already defined the value of the parameter with a variable. For instance, **digitalRead(pushbutton**) reads the input pin defined in the "pushbutton" variable. The controller has to locate this variable in its RAM memory and extract the pin number to be read. If it doesn't, an error message is generated.
- **Operation**: The value of the parameter is established as a result of any kind of arithmetic or logic operation between the variables and the constants. For instance, **digitalRead((1+1)*2)** reads pin 4 as this is the result of (1+1)*2.

A. CONSTANTS

A program value that never changes may be used as a "constant". If you have to read a button that's permanently connected to pin 4 you can use the digitalRead(4) function.

Each time the controller executes this function, it always reads the same pin: number 4. There's **NO** way of changing this **during** the execution of the program; if you do decide to change it you have to modify the program and record it again onto the controller's memory. Here's another example: imagine you want to create a program to calculate circumference based on a certain diameter. The equation would be **l = d * π.**

l = the length of the circumference. **d** = the diameter of the circumference **π** = 3.141592

"The European Commission support for the production of this publication does not constitute an endorsement of the contents which reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein."

In light of this example you might say the following:

Which one's the constant?_____

And which one's the variable? _____

The constants are an integral part of the program as they appear in the instructions or functions. They're stored in the controller's FLASH memory program and so can't be modified unless the program's rerecorded. They're not volatile and even when the power is disconnected they maintain their value.

More examples:

digitalRead(12); *//read input pin 12*

digitalWrite(6,HIGH); *//set output pin 6 on level "1"*

pinMode(9,OUTPUT); *//Configure pin 9 as an output*

B. VARIABLES

The variables are spaces in the controller's RAM memory specially designed for storing different kinds of data. Think of them as a series of boxes or containers where you can store any kind of information and use it later on whenever you need it.

Don't forget you can read and write the RAM memory as much as you want as long as you do it the right way. Yes, you can modify its contents during the execution of a program. It's volatile and the contents disappears when the power is disconnected.

You used variables in some examples in the previous unit. You have to declare them before you use them. To declare them, first you have to give them a name and some contents just in case using the following syntax:

type name = value

type: Establishes the type of data that the variable contains. This may be numbers, letters or texts. We'll be looking at them directly.

name: This is the name you assign to the variable or container; you have to use it in the future when you want to access the contents. Try to use concise names that remind you of the contents in some way. Always begin with a letter, not a number, and don't leave any blank spaces.

value: This is the content or value we put in the variable. It's optional: you can create a variable that's completely empty. If you do this, make provision in the controller's RAM memory to assign it a value when appropriate for your program.

"The European Commission support for the production of this publication does not constitute an endorsement of the contents which reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein."

The following table contains a summary of the most common types of variables:

TYPE	BYTES	DESCRIPTION	EXAMPLES
char	1	A char takes up 1 byte of memory and stores a character value (8 bits). Character literals are written in single quotation marks (').The char datatype is a signed type, meaning that it encodes numbers from -128 to 127.	char N = '1' char letter ='A'
byte	1	A byte stores an 8-bit unsigned number and yields a range of 0 to 255 (28-1).	byte C = 23 byte result = C +18
int	2	Integers are your primary data-type for number storage. They store a signed 16-bit (2-byte) value and yield a range of -32768 to +32767	int C = 2453 int value = C * 10 number = 2453
unsigned int or word	2	Stores an 16-bit unsigned number and yields a range of 0 to 65535 (216-1)	unsigned int valor = 60000 word valor = 4328 * 10
long	4	These are signed variables that store 32 bits (4 bytes) and yield a range of -2,147,483,648 to 2,147,483,647.	long counter = -123456789
unsigned long	4	Unsigned variable that stores 32 bits (4 bytes) and yields a range of 0 to 4,294,967,295 (2^32 - 1).	unsigned long speedOfLight = 299792468
float	4	Floating-point numbers can be as high as 3.4028235E+38 and as low as -3.4028235E+38. They are stored as 32 bits (4 bytes) of information.	float pi = 3.1416 float L = 4 * pi

It's pretty clear that you consume a greater or lesser quantity of RAM memory depending on the type of variable you define. I advise you to define variables according to the sort of information they store. For instance, if you define a "long" type variable you'll consume 4 bytes of RAM memory but it's not particularly efficient to use one of these if the information it's going to store is only falls between 0 and 255. It's probably better to define it as a "byte" type that consumes only 1. The RAM memory is limited; if you use it the wrong way you might get an unpleasant surprise when it runs out.

On the other hand, if you declare a "byte" type variable and you give it a value of greater than 255 the variable overloads and loses the information. It's a bit like what happens to the milometer in a car when exceeds its capacity: it starts again from 0.

The so-called **"Arrays"** are another kind of variable. An array is a group of variables and it may comprise any of the ones in the above table. To access the contents of the variables you have to indicate their index number. Have a look at these examples.

"The European Commission support for the production of this publication does not constitute an endorsement of the contents which reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein."

- **int Table[4]**

Creates an empty array called “Table” and saves space for storing four int. type numbers. It therefore takes up 8 bytes in the controller’s RAM memory.

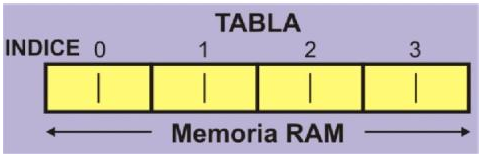

Figure 1

- **int Table[4]={5678,750,1234}**

Creates an array “Table” with enough space for storing 4 int. type numbers; this is where the three values indicated are stored.

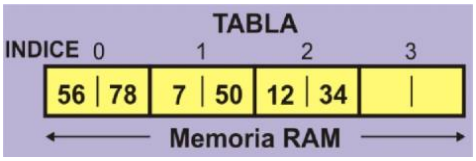

Figure 2

- **int Suma = Table[0] + Table[2]**

Retrieves the values for the 0 variable (5678) and the 2 variable (1234) from the “Table” array. It adds them and then stores the result (6912) in the “Sum” variable.

- **unsigned int Values[12]**

Creates a “Values” array that saves enough space for 12 entire unsigned type variables. It takes up a total of 24 bytes in the RAM memory.

- **Values[5] = 12345 * 3**

The result of the calculation (37035) is inserted in the number 4 variable of the “Values” array.

- **char Text[8] = “ARDUINO”**

Creates a character “Text” type array which takes up 8 bytes in the RAM memory and stores the “ARDUINO\o” chain (the \o is an override code that indicates the end of the chain).

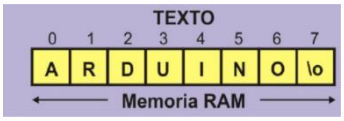

Figure 3

- **char Letter = Text[2]**

Retrieves the number 2 variable from the “Text” array (Letter D) and stores it in the char datatype “Letter” variable.

"The European Commission support for the production of this publication does not constitute an endorsement of the contents which reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein."

C. VARIABLE SCOPE

Variables may be “global” or “local” depending on where you declare them in the program. This won’t affect you much at the moment but may be important later on when you begin to create bigger and more complex programs.

Globals: These are the variables that are declared or defined outside all the functions in the program including setup() and loop(). Some of the examples in the previous unit used these types of variables. Have a look at the figure and you’ll see that the “Data” and “Result” variables are global. They may be used by all the functions in the program.

Locals: These are the variables that are declared and created within a certain function and may only be used by that function. This way we avoid a variable declared in one function being used or modified, or both, in another.

When the execution of a function finishes all the local variables disappear until we use the function again. According to the figure, “My_Function_1” has the following local variables: “Time”, “Hour” and “Result”.

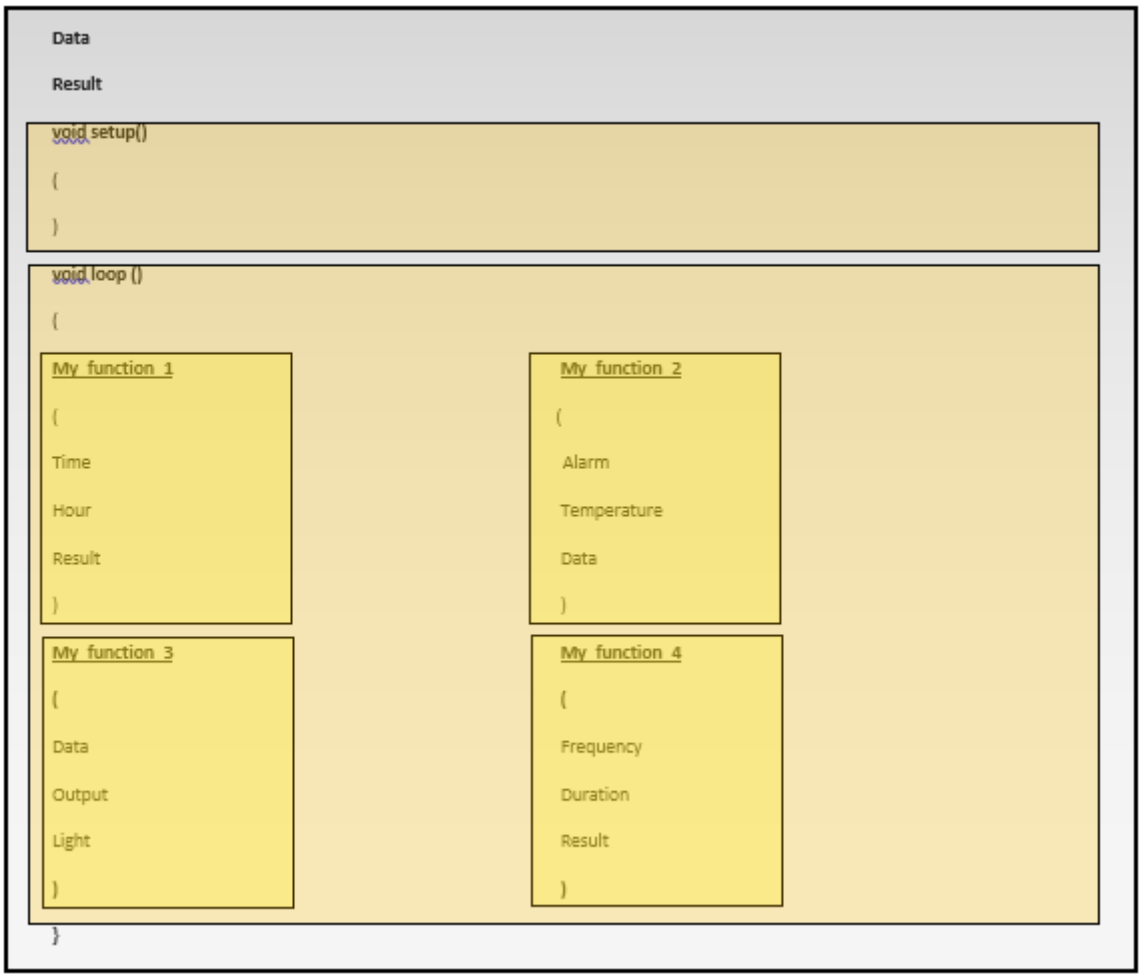

Figure 4

"The European Commission support for the production of this publication does not constitute an endorsement of the contents which reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein."

Answer the following questions:

1. Can “My_Function_2” use the “Light” variable? _____

2. Can “My_Function_1” use the “Hour” variable? _____

3. Can “My_Function_4” use the “Data” variable? _____

D. OPERATIONS

I’m sure you’ve already guessed that the parameters that go with the sentences in Arduino language as well as the values you load into the variables can be obtained as a result of doing all sorts of arithmetic operations between variables and/or onstants.

These are the most common mathematical operations:

= **Equivalence**

+ **Addition**

- **Subtraction**

***** **Multiplication**

/ **Division**

% **Remainder (the remainder of a division between two whole numbers)**

All calculations are performed from left to right, just like we do. Multiplication and division get priority over addition and subtraction. You can use parentheses to establish the priority you want in an operation. Have a close look at the following examples:

-> 3 + 5 – 2 = 6

-> 20 – 2 × 3 = 14 and not 54

-> (20 - 2) × 3 = 54 and not 14

-> 20 / 2 × 3 = 30

-> 24 / (2 × 3) = 4

-> (12 - 2) + 5 × 3 = 25 and not 45

-> ((12 - 2) + 5) × 3 = 45 and not 25

->13 % 4=1

-> 7 % 5 =2

-> 5 % 5 = 0

"The European Commission support for the production of this publication does not constitute an endorsement of the contents which reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein."

This section will give you much more flexibility and capability when you express the parameters that some Arduino functions use. Have a look at the examples and try and answer the questions.

```
byte Light=3;

pinMode(Light×2,OUTPUT);

digitalRead(Light+1);

digitalWrite((Light-1)×3),HIGH);
```

Which pin is configured as an output? _____

Which pin is read? _____

Which pin is set at level “1”? _____

```
byte A = 250;

byte B; byte C;

unsigned int D;

A = A + 3;

B = (A - 200) × 2 / 4;

C = A × 2 + 20;

D = (A × 100) % 2;
```

A = _____; B = _____; C = _____; D = _____

"The European Commission support for the production of this publication does not constitute an endorsement of the contents which reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein."

2. TIMINGS

This might seem like a contradiction difficult but it's often a good idea for the controller to do absolutely nothing useful or, in other words, for it to waste a bit of time.

The Arduino programming language has a number of functions that pause the execution of a program for a set amount of time. You mightn't understand exactly why at the moment but the examples will make things clearer.

A. THE DELAYMICROSECONDS FUNCTION

Pauses the program for the amount of time (in microseconds) specified. There are a thousand microseconds in a millisecond and a million microseconds in a second. Remember the following: 1 second = 1000 milliseconds (mS) and 1 millisecond = 1000 millionths of a second given that 1 second = 1,000,000 millionths of a second (µS) and 1 µS = 0.000001 seconds.

Syntax:

delayMicroseconds(n);

n: indicates the number of milliseconds you want to pause the program. It's a 16 bit unsigned integer (unsigned int). Currently, the largest value that will produce an accurate delay is 16383 (approximately 16 mS or 0.016 seconds). For delays longer than a few thousand microseconds, you should use **delay()** instead; we'll be taking a look at it right away.

Examples:

A = 100;

delayMicroseconds(A); *//pauses the program for 100 µS = 0.1 mS = 0.0001 seconds*

delayMicroseconds(A×10-100); *//pauses the program for 900 µS = 0.9 mS = 0.0009 seconds*

B. THE DELAY() FUNCTION:

Pauses the program for as many milliseconds as you indicate. Bear the following in mind: 1 second = 1000 milliseconds, or in other words, 1 millisecond (mS) = 0.001 seconds.

Syntax:

delay(n);

n: indicates the number of milliseconds (mS) you want to pause the program. These are referred to as long numbers without a 32 bit sign or unsigned long numbers and their range is from 0 to 4,294,967,295 (2^{32}- 1).

Examples:

A=100; delay(A); *//Pauses the program for 100 mS = 0.1 second delay(1000)*
*//Pauses the program for 1000 mS = 1 segundo delay(A*600);*
//Pauses the program for 60000 mS = 60 seconds = 1 minute

"The European Commission support for the production of this publication does not constitute an endorsement of the contents which reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein."

Remember that when you pause the program using the **delayMicroseconds()** function or the **delay()** function, the controller no longer executes your program instructions until the pause is over.

C. THE MICROS() FUNCTION

Returns the number of microseconds elapsed since Arduino began running your program. It returns an unsigned long value and the number will overflow (go back to zero) after approximately 70 minutes. Remember there are 1,000 µS (microseconds) in a millisecond (mS) and 1,000,000 µS in a second.

Syntax:

var = micros();

var: This is the variable that stores the microseconds (µS) elapsed since the system was restarted.

D. THE MILLIS() FUNCTION

Returns the number of milliseconds (mS) elapsed since Arduino began running your program. This number will overflow (go back to zero), after approximately 50 days.

It returns an unsigned long value and the number will overflow (go back to zero) after approximately 50 days. Remember there are 1,000 milliseconds (mS) in a second.

Syntax:

var = millis();

var: This is the variable that stores the milliseconds (mS) elapsed since the system was restarted.

3. SOUND

What happens when you throw a stone into a pond? The stone hits the water and changes its pressure; this sets off a series of waves that spread outwards and then gradually disappear.

Sound is much the same. All you need is a device able to vibrate and produce pressure changes or vibrations in the air. These vibrations reach our ear drums and our brain perceives them as sound. When we talk or sing, our vocal chords are responsible for producing the pressure changes or waves that spread out through the air until they reach other people's ear drums.

There are lots of well known devices that cause changes in the air pressure and thus create waves that carry sound: loud speakers, headphones, buzzers, sirens etc.

All these devices rely on the same three basic components: a diaphragm, a coil and a permanent magnet. The diaphragm can be made of paper, cardboard, plastic or any other material that has been properly treated. The diaphragm is joined to the coil which in turn is mounted on a permanent magnet.

"The European Commission support for the production of this publication does not constitute an endorsement of the contents which reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein."

An electric signal is applied to the coil which creates a magnetic field; this in turn interacts with the magnetic field of the permanent magnet. These two magnetic fields may attract or repel each other. This attraction or repulsion between the coil and the magnet "drags" or "pushes" the diaphragm joined to the magnet. The diaphragm's movements cause changes of pressure, vibrations or waves. And that's how sound travels through the air. In addition to this, these vibrations or waves are a true reproduction of the electric signal that generated them in the first place.

The electric signal that causes this process has to be variable and fluctuate constantly between level "1" and "0" at a certain speed or frequency. Fixed signals don't produce the same effect. Imagine a signal that's constantly on level "1" or "0"; the coil goes into or comes out of the permanent magnet and that's where it stays; the diaphragm doesn't vibrate and no sound waves are produced.

Humans are able to hear sound frequencies between approximately 20 Hz and 20,000 Hz (20 KHz) or cycles per second. This indicates the frequency or number of times per second a signal passes from "1" to "0".

We perceive frequencies as "pitches". A low frequency produces a low pitched sound; the higher the frequency the higher the pitch.

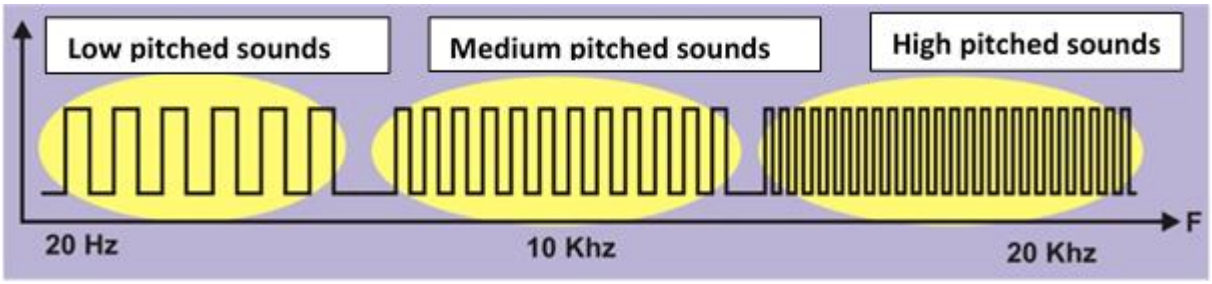

Figure 5

The vibrations with a frequency lower than 20 Hz are called "*infrasound*" and the ones over 20 KHz "*ultrasound*". Humans are unable to hear these sounds although some animals are.

Let's make some calculations. Suppose you want to generate a sound signal with a frequency of 100 Hz:

1. Based on F, the desired frequency (100 Hz), we calculate the T period, or the time that a cycle lasts.

$$T = \frac{1}{F} = \frac{1}{100} = 0.01\ s = 10\ mS = 10000\ \mu S$$

2. Now we calculate the S semi-period or, in other words, the time the signal will be at level "1" and at level "0".

$$S = \frac{T}{2} = \frac{0.01}{2} = 0.005\ s = 5\ mS = 5000\ \mu S$$

3. With what you've learnt already you could now create a program that would generate the desired frequency through the speaker connected to pin 2. Have a close look at the following solution:

"The European Commission support for the production of this publication does not constitute an endorsement of the contents which reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein."

```
void setup()
{
pinMode(2,OUTPUT);                    // Output pin 2 (speaker)
}
// T Period for total of 10000 µS at an F frequency of 100 HZ
void loop()
{
digitalWrite(2,HIGH);                 //Pin 2 at level "1"
delayMicroseconds(5000);              //Pauses for 5000 µS
digitalWrite(2,LOW);                  //Pin 2 at level "0"
delayMicroseconds(5000);              //Pauses for 5000 µS
}
```

Note that pin 2 stays at level "1" (HIGH) for 5000 µS and at level "0" (LOW) for the same period. In other words, the "**T**" cycle at level "1" and level "0" is produced on pin 2 and lasts for 10000 µS. As it's constantly repeated, a second contains 100 cycles (1000000/10000). This is the **F** frequency.

Record the program and ensure it works properly. This will mean a constant and annoying sound at a frequency of 100 Hz.

Modify the program to get sounds with pitches from a number of different **F** frequencies; calculate their **T** periods and **S** semi-periods using the table below:

F	T	S	F	T	S
50 Hz			2.5 KHz		
200 Hz			5 KHz		
500 Hz			10 KHz		
1 KHz			15 KHz		
2 KHz			20 KHz		

4. SOUND FUNCTIONS

The Arduino programming language makes it very easy for you to produce all sorts of sounds. You won't have to work out either the **P** period or the **S** semi-period. All you have to do is indicate the **F** frequency and Arduino will take care of the rest.

"The European Commission support for the production of this publication does not constitute an endorsement of the contents which reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein."

- **The Tone() Function**

Generates a sound with the required **F** frequency on the output pin and for the duration indicated.

Syntax:

tone(pin, frequency, duration);

pin: indicates which pin you're going to generate the sound on.

frequency: this is a whole unsigned number (unsigned int) that represents the frequency of the tone in Hz (hertz). Remember that humans are only able to hear sound frequencies between approximately 20 Hz and 20,000 Hz (20 KHz) or cycles per second.

duration: This parameter is optional. It's an unsigned long number that represents the duration of the tone in milliseconds (if there is no indication the tone continues indefinitely or until the noTone() function is executed).

Examples:

```
int Pin=11;

int F = 1200;

int D = 3000;

tone(2,F,D);          //Generates a 1200 Hz tone for 3 seconds on pin 2

tone(6,F*10,500);     //Generates a 12 KHZ tone for 0.5 seconds on pin 2.

tone(Pin,200);        //Generates an indefinite 200 Hz tone on pin 11
```

- **The noTone() Funtion:**

Stops a tone on a designated pin.

Syntax:

noTone(pin);

pin: *represents on which pin the sound is going to be stopped.*

Example:

```
tone(2,13000);        //Generates an indefinite tone 13 KHz tone on pin 2.

….                    //The tone keeps sounding.


noTone(2);            //The tone stops.
```

"The European Commission support for the production of this publication does not constitute an endorsement of the contents which reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein."

PRACTICE SECTION

5. EXAMPLE 1: Flashing lights

This is the simplest example program you can imagine: a flashing white LED connected to pin 6. Take a look at the solution in the Figure 6.

The "Tiempo" variable contains the duration of the pause, 500 mS = 0.5 seconds. Pin 6, the one the white LED is connected to, is configured as an output.

The main body of the program contained in the **loop()** function confines itself to switching pin 6 on and off at the intervals defined by the "*Tiempo*" variable, 500 mS.

```
int Tiempo = 500;              //Valor de la temporización

//Sentencias iniciales de configuración
void setup()
{
  pinMode(6, OUTPUT);          //La patilla 6 del led blanco se confi
}

void loop()
{
  digitalWrite(6,HIGH);        //El led blanco se ilumina
  delay(Tiempo);               //Temporiza
  digitalWrite(6,LOW);         //El led blanco se apaga
  delay(Tiempo);               //Temporiza
}
```

Figure 6

- **Now it's your turn**

You can do a number of different activities with this example. Here are a couple of suggestions:

1. You can try varying the pause interval using the "*Time*" variable. I suggest you shorten it. Bring it down bit by bit until you get to 10 mS. What do you notice?

   It looks like the white LED is always on but this isn't the case. The LED is actually going on and off very quickly. In fact our retina is unable to detect such rapid changes and creates an optical illusion: we perceive the LED as being permanently alight.

   Time is truly an elastic concept! What seems constant to you isn't that way for the controller. It has to keep the LED on for 10 mS with and keep it off for another 10 mS but you don't perceive this.

2. Instead of using the delay() function try using another one: delaymicroseconds(). You can create extremely short and precise pauses with it: as little as a millionth of a second (µS).

3. You can also modify the program to simulate a lighthouse. For instance, you can have the LED give off a 0.1 second flash of light every 2 seconds.

"The European Commission support for the production of this publication does not constitute an endorsement of the contents which reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein."

6. EXAMPLE 2: Sets of lights

This is a nice example. Four LED lights are going to go on and off one after the other for the same amount of time starting with the white one. It'll give you a feeling of movement from right to left.

The program might be a bit long but it's really quite simple. It turns the white light on, pauses it, turns it off and then turns the green one on. Then it pauses the green one, turns it off and turns the amber on one. Next it pauses the amber light, turns it off and then turns the red one on. Finally it pauses the red one, turns it off and the cycle starts again.

- **Now it's your turn**

1. Change the time: the idea is to go faster or slower as you move from right to left.
2. First make the lights go on and off from right to left and next from left to right.

7. EXAMPLE 3: Traffic signal V1

What about doing a short project with all the things you've learnt? I suggest you simulate a simple traffic signal; this will be the first version of a series of traffic signals that you'll improve as you go along.

The program as such doesn't really offer anything new. It just involves turning the green, amber and red LEDs on and off in the right order for the time you indicate. There's one difference you might notice though.

- **Conclusion**

Notice that all the program functions that establish the working sequence of the traffic signal are included in the setup() function; the main loop() function is empty.

This is perfectly normal. We looked at it in past lessons; remember? You have to include the loop() function even if it doesn't have anything in it. On the other hand all the functions included in the setup() function are only executed once. So every time you want your traffic signal to perform a new sequence you'll have to push the RESET button again; give it a go.

8. EXAMPLE 4: Electric bell

Here's another simple project. This time you're going to simulate an electric bell. Every time you press the pushbutton the bell makes a sound consisting of three different pitches.

This will be the first time you're going to use the **tone()** function and you'll see that it's really very simple. All you have to do is stipulate the output pin, the frequency and the duration (this last one is optional).

The solution to the program is contained in the figure. An initial tone of 1000 Hz is generated for 0.2 seconds. Next, after a pause of 0.4 seconds, another tone of 15000 Hz is generated for 0.3 seconds. Last of all, 0.4 seconds later, a third tone of 2000 Hz is generated for 0.4 seconds.

"The European Commission support for the production of this publication does not constitute an endorsement of the contents which reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein."

We get the tones on pin 2 of the controller.

Take note that all the functions are included in the **setup()** function so the sequence is executed every time you press **RESET**.

- **Now it's your turn**

You're going to improve the bell by adding a signal light. As well as the previously described sequence I suggest you turn on the white LED every time you press **RESET**. Once the third tone is generated and the sequence finishes, the LED light should go out.

9. EXAMPLE 5: Melodies

Here's something for the musical people (unfortunately I'm not one of them). This example combines a series of musical notes of different frequencies and durations that generate a well-known melody. To put it more simply, each musical note corresponds to a certain frequency. We get the sharp and flat notes and combinations of them from these frequencies. Give it a go and try different combinations!

10. EXAMPLE 6: Traffic signal V2

Here's another little project for you. It's an improved version of the V1 traffic signal: we've added some acoustic warning signals for the visually impaired.

The program may seem a bit long to you but if you have close look you'll see it's not really very complicated. It's very sequential. It starts off by illuminating the green light at the same time as it generates six 1 KHz notes 0.5 seconds long. Next it illuminates the amber light and generates six 1 KHZ notes 0.4 seconds long. Last of all, the red light goes on and a 1KHz note sounds for 6 seconds.

As all the program functions are included in the **setup()** function, they are only executed once: each time you push **RESET** the sequence starts.

- **Now it's your turn**

So the program execution doesn't last too long when we're testing it you'll notice that the time each light stays on for is actually much shorter than its actual length.

Look for a traffic signal in your street or area and time how ling each light stays on for. You should also listen to the warning sounds each signal makes and whether or not the lights and the sounds are intermittent. The idea is to modify your program so that it simulates the way a real traffic signals works as closely as possible.

"The European Commission support for the production of this publication does not constitute an endorsement of the contents which reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein."

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

"The European Commission support for the production of this publication does not constitute an endorsement of the contents which reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein."