

# Tecnología Electrónica de Computadores

Juan Carlos Álvarez Antón – anton@uniovi.es – Version V1.0, 13.04.2019

---

## Tabla de Contenidos

### MICROCONTROLADORES

- Microcontroladores
  - Hardware Arduino
  - IDE Arduino
  - Programación: funciones.
  - Resumen de funciones
  - Interrupciones
  - Temporizadores
  - Modos de bajo consumo
  - Watchdog
  - Brown-Out Detector(BOD)
  - Buses de comunicación del uC
  - Conversión Analógica Digital y Digital Analógica
- 

Prohibida la reproducción total o parcial de esta obra, por cualquier medio, sin la autorización escrita del autor. Copyright © Juan Carlos Álvarez Antón

## MICROCONTROLADORES

### *Objetivos*

- Introducir el concepto de microcontrolador (uC)
- Describir los bloques funcionales básicos de los uC
- Describir el entorno de programación del uC (ATmega 328P)

Este documento describe las unidades funcionales del uC ATmega328P de Atmel y su programación. Le servirá de ayuda para realizar las prácticas de laboratorio correspondientes a la 2º parte de la asignatura. Las sesiones de laboratorio incluyen aplicaciones con sensores y actuadores utilizando el hardware y entorno de programación del ecosistema Arduino.

### Microcontroladores

Un [microcontrolador](https://es.wikipedia.org/wiki/Microcontrolador#Caracter%C3%ADsticas) (<https://es.wikipedia.org/wiki/Microcontrolador#Caracter%C3%ADsticas>) (uC) es un procesador acondicionado especialmente para el procesamiento de eventos, es decir, está en contacto con el mundo exterior capturando, procesando y reaccionando ante sucesos externos. También se denomina “computador en un chip” (computer on a chip) porque integra parte de la funcionalidad habitual de un computador convencional en una única pastilla de silicio, aunque con prestaciones más reducidas. Los uC están presentes en multitud de dispositivos desarrollando una tarea específica ([sistema empotrado](https://es.wikipedia.org/wiki/Sistema_embebido) ([https://es.wikipedia.org/wiki/Sistema\\_embebido](https://es.wikipedia.org/wiki/Sistema_embebido))): electrodomésticos, automóviles, equipamiento informático, robótica, dispositivos del IoT, electrónica de consumo, máquinas industriales, ... Un uC incorpora diversas unidades funcionales: convertidores A/D, temporizadores (timers o contadores), comparadores, puertos digitales de E/S, salidas analógicas (DAC, PWM), bloques de comunicación (serie asíncrono - UART-, serie síncrono, SPI, I<sup>2</sup>C), memoria (SRAM para datos y Flash para el programa), etc.

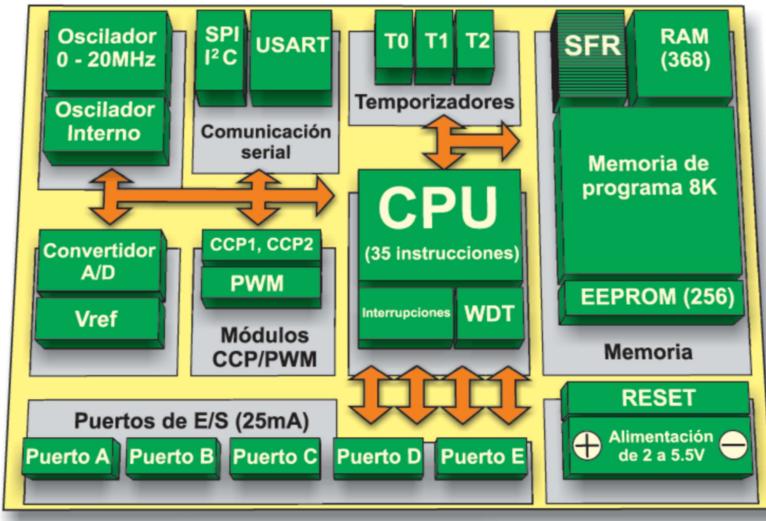


Figure 1. Microcontrolador: unidades funcionales

### ¿Qué buscamos en un uControlador?

- Disponibilidad de periféricos y unidades funcionales avanzadas (hardware)
- Bajo consumo
- Disponibilidad de librerías de software

What does a developer want in an  
MCU? ■

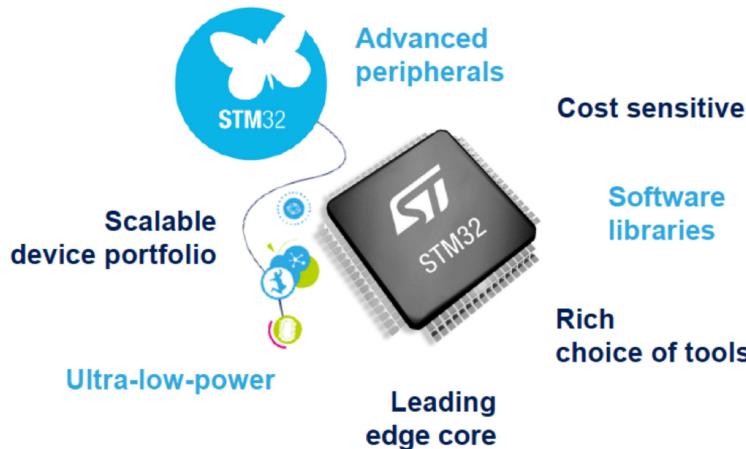


Figure 2. Microcontrolador: características deseables

### ¿Quiénes son los principales fabricantes?

Las siguientes figuras muestran los principales fabricantes de microcontroladores y de semiconductores a nivel mundial.

#### Leading MCU Suppliers (\$M)

2021 Rank	Company	Headquarters	2020	2021	21/20 % Chg	2021 Marketshare
1	NXP	Europe	2,980	3,795	27%	18.8%
2	Microchip	U.S.	2,872	3,584	25%	17.8%
3	Renesas	Japan	2,748	3,420	24%	17.0%
4	ST	Europe	2,506	3,374	35%	16.7%
5	Infineon	Europe	1,953	2,378	22%	11.8%

Source: Company reports, IC Insights

Figure 3. Principales fabricantes de microcontroladores

### Top 10 Worldwide Semiconductor Sales Leaders

Rank	Company	2016		2017		2018		2019F			
		Sales (\$B)	Company	Sales (\$B)	Change	Company	Sales (\$B)	Change	Company	Sales (\$B)	Change
1	Intel	\$57.0	Samsung	\$65.9	48.8%	Samsung	\$78.5	19.2%	Intel	\$70.6	1.0%
2	Samsung	\$44.3	Intel	\$61.7	8.2%	Intel	\$69.9	13.2%	Samsung	\$63.1	-19.7%
3	TSMC (2)	\$29.5	TSMC (2)	\$32.2	9.1%	SK Hynix	\$36.8	37.6%			
4	Qualcomm (1)	\$15.4	SK Hynix	\$26.7	79.5%	TSMC (2)	\$34.2	6.4%			
5	Broadcom (1)	\$15.2	Micron	\$23.9	76.7%	Micron	\$31.0	29.6%			
6	SK Hynix	\$14.9	Broadcom (1)	\$17.8	16.9%	Broadcom (1)	\$18.5	3.7%			
7	Micron	\$13.5	Qualcomm (1)	\$17.0	10.5%	Qualcomm (1)	\$16.4	-3.8%			
8	TI	\$12.5	TI	\$13.9	11.3%	Toshiba*	\$14.9	12.1%			
9	Toshiba	\$10.9	Toshiba	\$13.3	21.9%	TI	\$14.9	6.8%			
10	NXP	\$9.5	Nvidia (1)	\$9.4	36.1%	Nvidia (1)	\$12.0	27.1%			
<b>Top 10 Total (\$B)</b>		<b>\$222.8</b>	<b>—</b>		<b>\$281.9</b>	<b>26.5%</b>	<b>—</b>		<b>\$327.0</b>	<b>16.0%</b>	
<b>Semi Market (\$B)</b>		<b>\$364.0</b>	<b>—</b>		<b>\$445.2</b>	<b>22.3%</b>	<b>—</b>		<b>\$504.1</b>	<b>13.2%</b>	<b>—</b>
Source: IC Insights (1) Fabless (2) Pure-Play foundry						*Includes Toshiba Memory					

Figure 4. Principales fabricantes de semiconductores

¿Qué arquitectura tienen los microcontroladores?

Casi todos los microcontroladores avanzados son máquinas [RISC](#)

([https://es.wikipedia.org/wiki/Reduced\\_instruction\\_set\\_computing](https://es.wikipedia.org/wiki/Reduced_instruction_set_computing)) con arquitectura [Harvard](#)

([https://es.wikipedia.org/wiki/Arquitectura\\_Harvard](https://es.wikipedia.org/wiki/Arquitectura_Harvard)). Dos de las familias más importantes y conocidas de microcontroladores son: ARM (Advanced RISC Machine) y AVR.

## ARQUITECTURAS



Figure 5. Arquitecturas de microcontroladores

Arquitectura AVR

Los uC con arquitectura AVR (Alf-Egil and Vegard Wollan, RISC) son una familia de microcontroladores RISC del fabricante estadounidense Atmel (ahora Microchip) con arquitectura Harvard de 8 bits. Fue uno de los primeros microcontroladores en incorporar memoria flash para almacenar el código del programa.



La arquitectura de los AVR fue concebida por dos estudiantes del Norwegian Institute of Technology (Alf-Egil and Vegard Wollan). Posteriormente fue refinada en Atmel Norway, una empresa subsidiaria de Atmel fundada por los dos arquitectos del chip.

Esta arquitectura se hizo muy popular por su diseño simple y fácil programación. Atmel fue adquirida por Microchip en 2016. La figura muestra el catálogo de uC de Atmel-Microchip.

## Atmel Microcontroller Portfolio

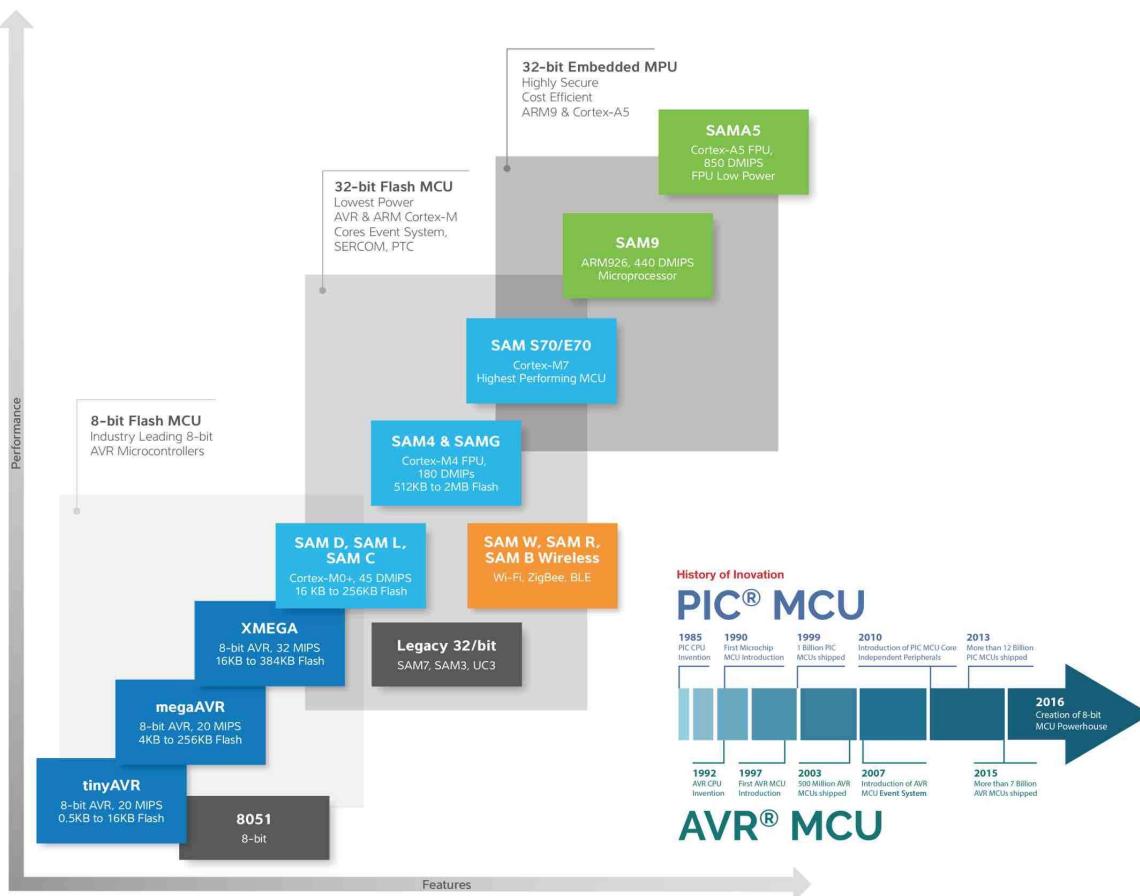


Figure 6. Ejemplo de Arquitecturas

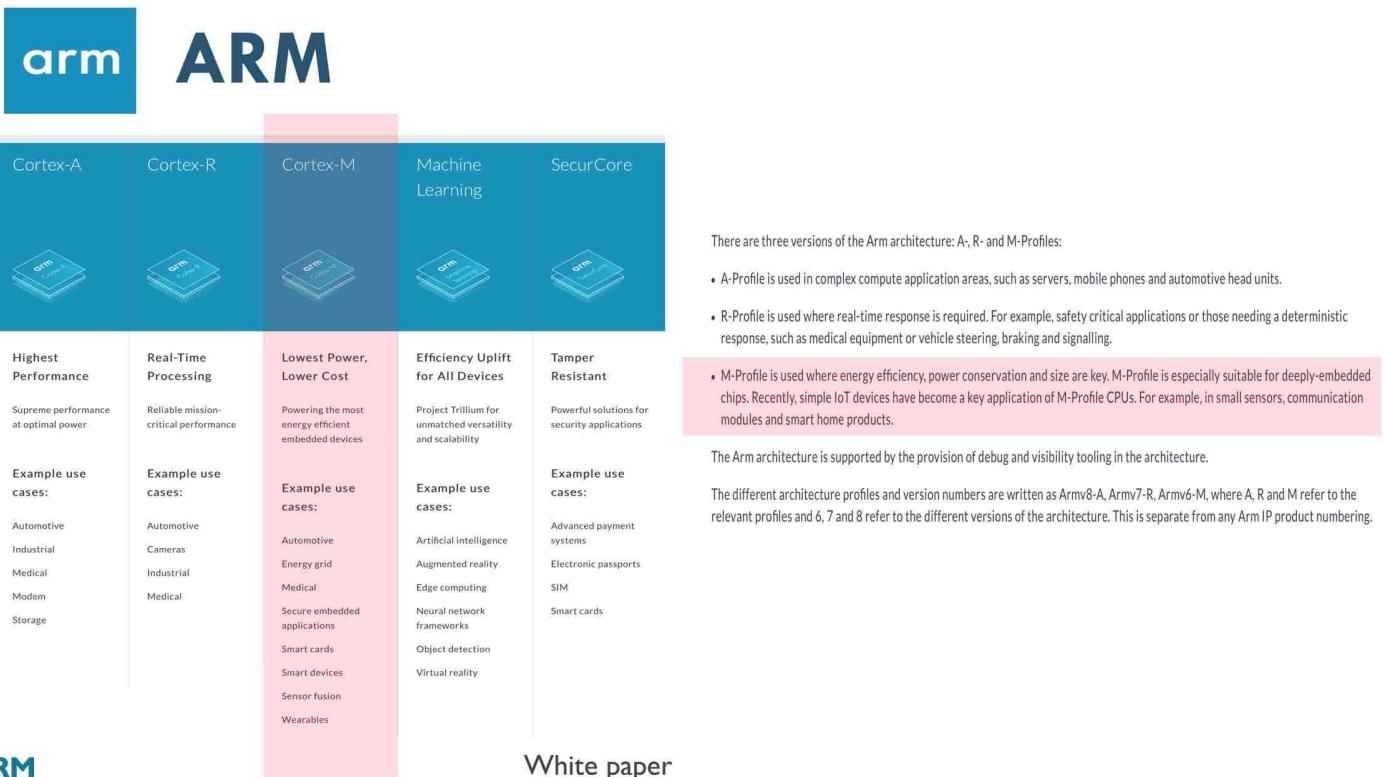
### Familias principales uC Atmel 8 bits:

- **ATxmega:** procesadores potentes de 16 kB a 384 kB de memoria flash programable, encapsulados de 44, 64 y 100 pines, capacidad de DMA, eventos, criptografía y amplio conjunto de periféricos, incluyendo DACs.
- **ATmega:** microcontroladores AVR de 4 kB a 256 kB de memoria flash programable, encapsulados de 28 a 100 pines, conjunto de instrucciones extendido (multiplicación y direccionamiento) y amplio conjunto de periféricos.
- **ATtiny:** pequeños microcontroladores AVR con 0,5 kB a 8 kB de memoria flash programable, encapsulados de 6 a 20 pines y un conjunto limitado de periféricos.

En este laboratorio utilizaremos un microcontrolador de la familia ATmega (ATmega328P).

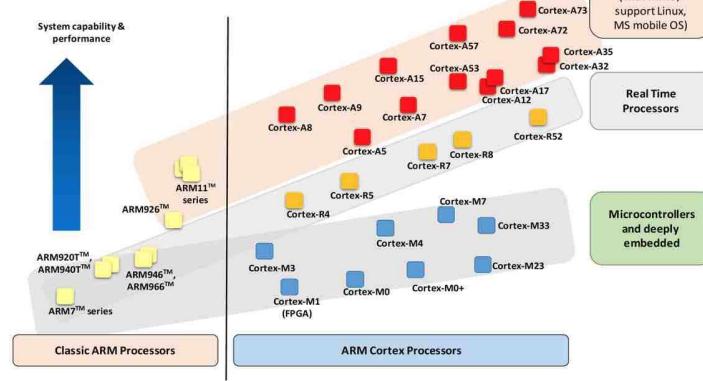
### Arquitectura ARM

ARM es una arquitectura RISC de 32 bits y 64 bits, desarrollada por **ARM Holdings**. Actualmente, cerca del 75% de los procesadores de 32 bits poseen este chip en su núcleo y se ha convertido en uno de los más usados del mundo. Teléfonos, periféricos, juguetes, electrodomésticos, etc. incorporan esta arquitectura. La arquitectura ARM es licenciable. Esto significa que el negocio principal de ARM Holdings es la venta de núcleos IP. Estas licencias se utilizan para crear microcontroladores y CPUs basados en este núcleo. Las empresas que son titulares de licencias ARM actuales incluyen, entre otras, a: Alcatel-Lucent, Apple Inc., AppliedMicro, Atmel, Broadcom, Cirrus Logic, Digital Equipment Corporation, Ember, Energy Micro, Freescale, Intel (a través de DEC), LG, Marvell Technology Group, Microsemi, Microsoft, NEC, Nintendo, Nokia, Nuvoton, Nvidia, Sony, MediaTek, NXP (antes Philips Semiconductors), Oki, ON Semiconductor, Psion, Qualcomm, Samsung, Sharp, STMicroelectronics, Symbios Logic, Texas Instruments, VLSI Technology, Yamaha, y ZilLABS...



ARM

White paper



*Figure 1: ARM processor family*

*Figure 7. Principales familias ARM*

# Hardware Arduino

Arduino es una plataforma open-source donde el usuario puede construir o adaptar la plataforma a sus necesidades. Está basada en una placa con un microcontrolador y un entorno de desarrollo software (IDE) orientado al diseño de sistemas programables en proyectos multidisciplinares.

El hardware consiste en una placa de circuito impreso con un microcontrolador de arquitectura Atmel AVR, o más recientemente SAMD ARM. La placa puede conectarse a otros módulos de expansión, "shields", para ampliar su funcionalidad. Las características básicas del entorno Arduino pueden resumirse en:

- Bajo coste
  - Entorno de programación sencillo y claro
  - Open-source software
  - Open-source hardware

Durante este curso utilizaremos la placa Arduino NANO con el microcontrolador de Atmel ATMega328P.

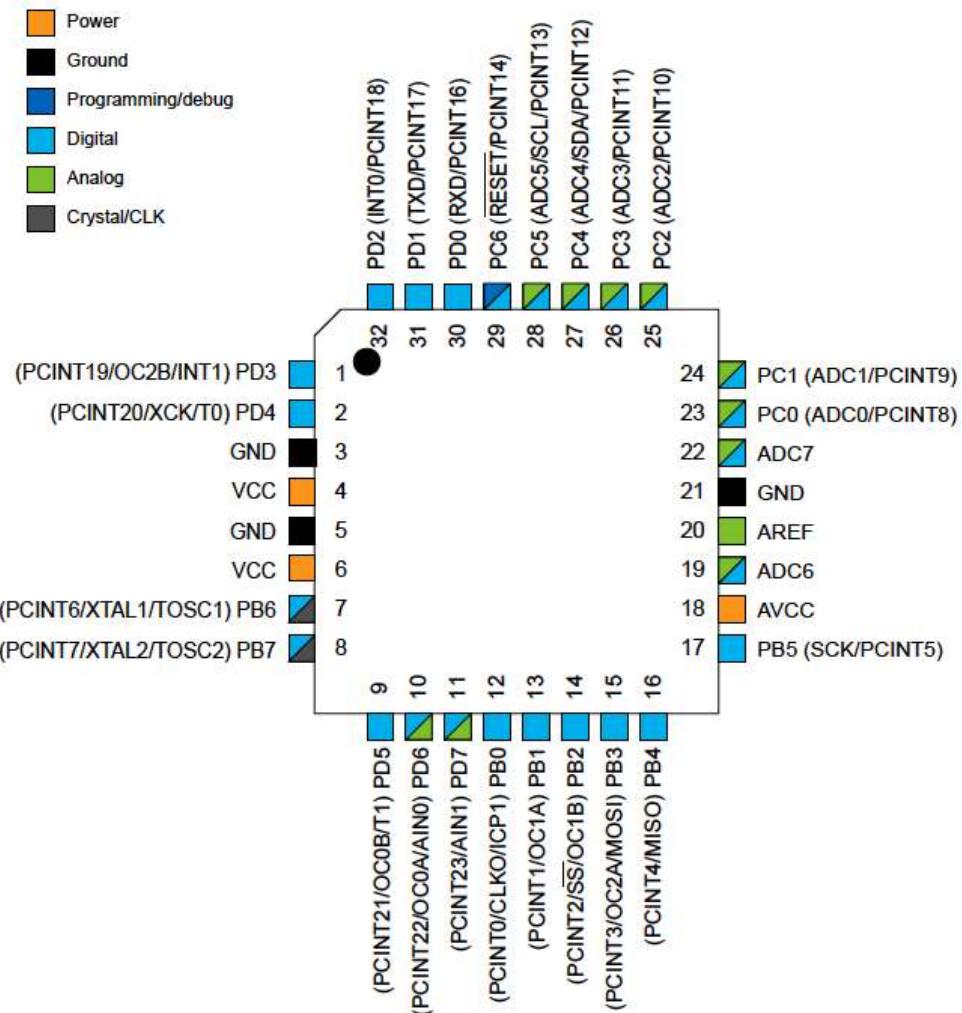


Figure 8. Microcontrolador ATMega328P (32 pin - TQFP ([https://es.wikipedia.org/wiki/Quad\\_Flat\\_Package](https://es.wikipedia.org/wiki/Quad_Flat_Package)) -Thin Quad Flat Package-) Top View

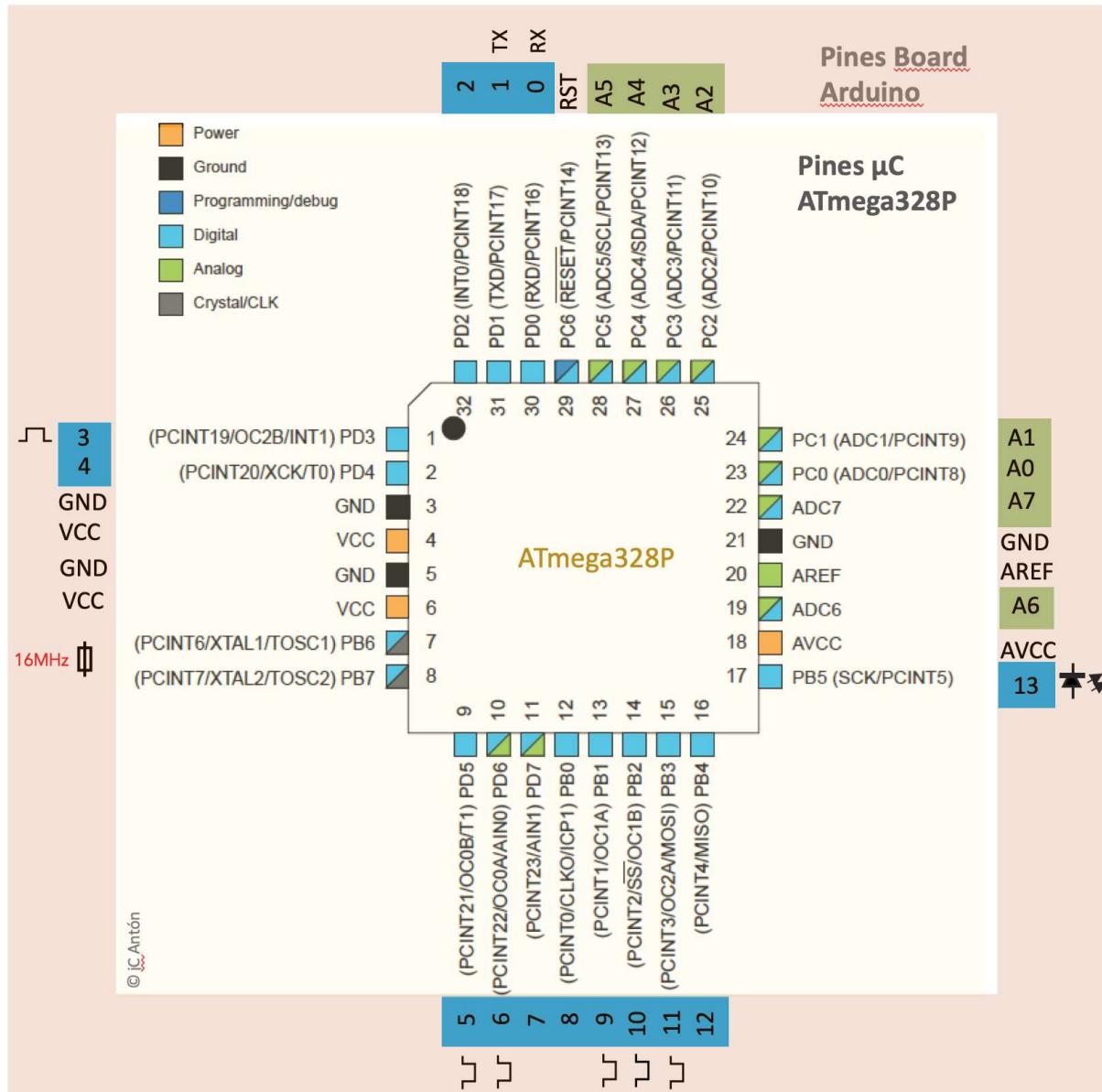


Figure 9. Arduino NANO Board con ATmega328P (TQFP) pin mapping



El ícono del pulso indica una salida PWM con hardware dedicado (OCnA/OCnB). Los pines XTAL1 y XTAL2 están conectados a un cristal de 16 MHz que proporciona la señal de reloj del sistema. El pin 13 (PB5) está conectada a un diodo LED incorporado a la placa.

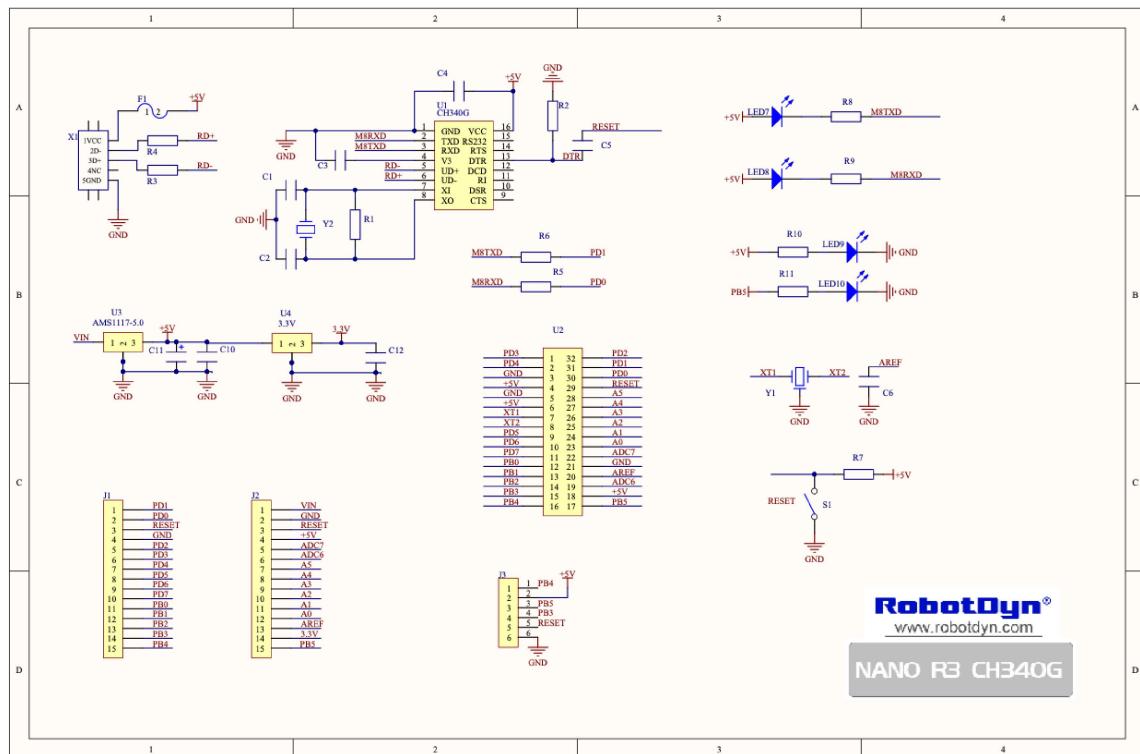


Figure 10. Arduino NANO esquema eléctrico

**NANO V3 CH340G**

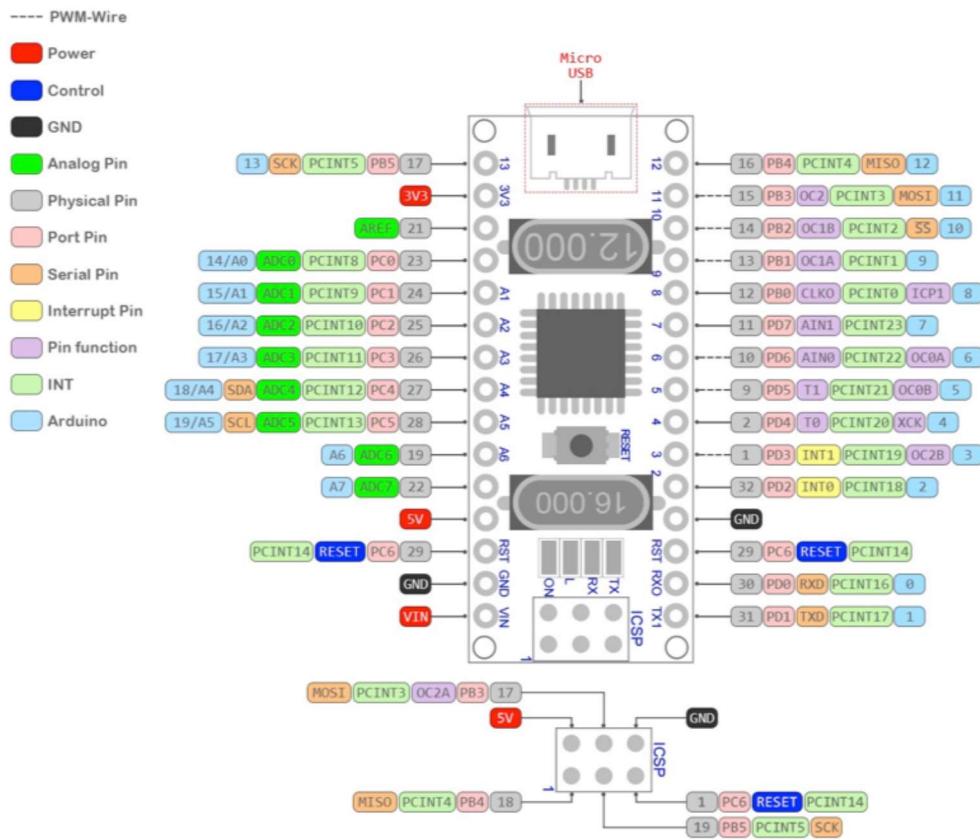


Figure 11. Arduino NANO pinout

## IDE Arduino

El IDE Arduino se describe [aquí](https://docs.arduino.cc/software/ide-v2) (<https://docs.arduino.cc/software/ide-v2>). Para comenzar puedes visitar [esta guía de introducción](https://docs.arduino.cc/software/ide-v2/tutorials/getting-started-ide-v2) (<https://docs.arduino.cc/software/ide-v2/tutorials/getting-started-ide-v2>)

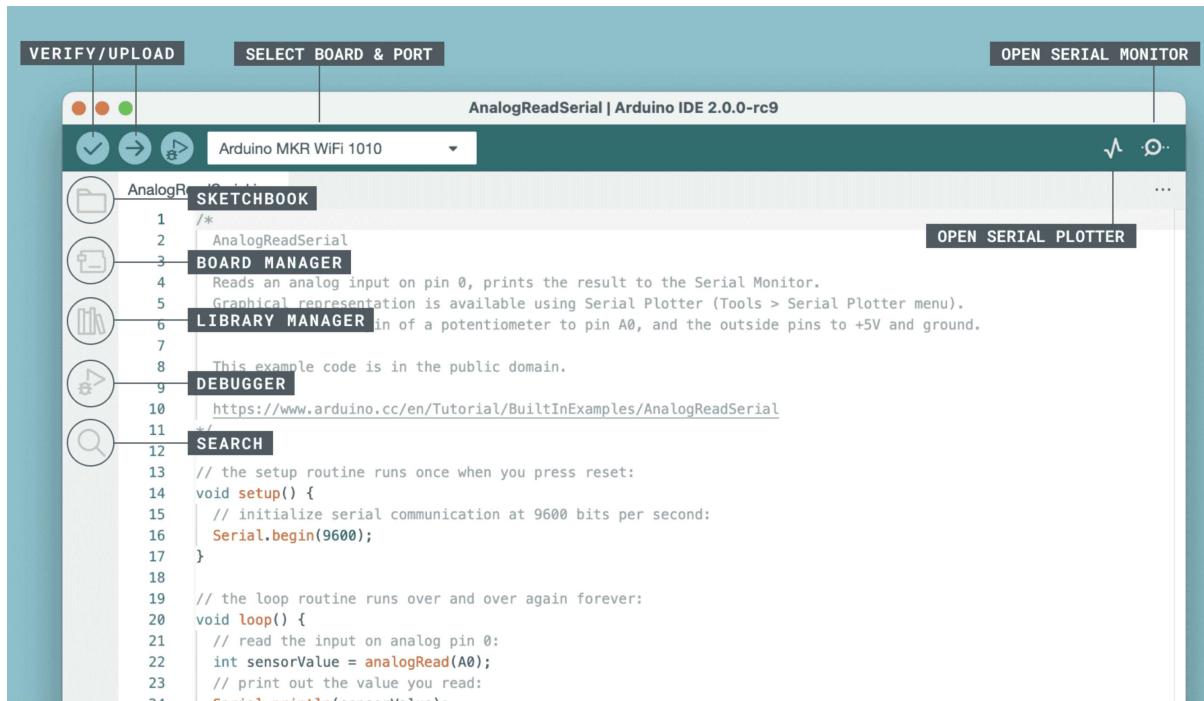


Figure 12. IDE Arduino

El proceso de compilación se realiza de forma automática pulsando sobre el botón de flecha del menú de herramientas.

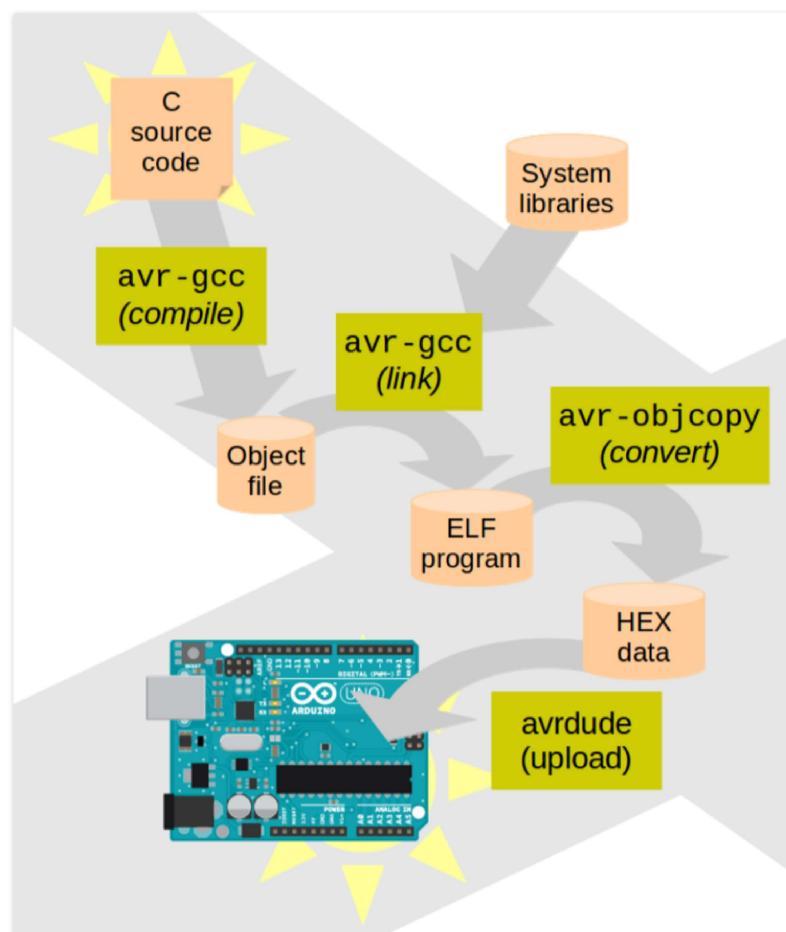


Figure 13. Proceso de compilación, linkado, conversión y carga.

#### Configuración de Placa y Procesador

Compruebe en el IDE Arduino que la placa y el procesador seleccionado son:

Tools (Herramientas) -> Board (Placa)-> Arduino NANO

Tools (Herramientas) -> Processor (Procesador) -> ATmega328P

### *Comunicación de placa Arduino con el bus USB del ordenador.*

Para asegurar la comunicación del PC con la placa Arduino verifique la siguiente configuración del IDE:

Tools (Herramientas) -> Port (Puerto) -> COMn

donde n debe ser 3, 4 ó 5, generalmente.

Compruebe que el puerto de comunicación es uno de los anteriormente mencionados. NO seleccione el puerto **COM1**. Si solo aparece este puerto, es posible que el PC no esté detectando la placa Arduino NANO. En tal caso, pruebe a desconectar el cable USB del PC y vuelva a conectarlo.



La placa NANO del laboratorio es una versión compatible con Arduino. Esta tarjeta se comunica con el PC a través del bus USB utilizando el chip TTL-USB "CH340G", en lugar del "FTDI" utilizado por las placas Arduino originales. El driver para el chip de FTDI se instala automáticamente pero el CH340G requiere instalar su driver ([http://www.wch.cn/download/CH341SER\\_EXE.html](http://www.wch.cn/download/CH341SER_EXE.html)) desde la página web del fabricante (WCH). Este driver se encuentra ya instalado en el PC del laboratorio.

## Programación: funciones.

El lenguaje de programación de Arduino puede dividirse en tres partes principales: funciones (<https://www.arduino.cc/reference/en/>), variables (<https://www.arduino.cc/reference/en/#variables>) y estructuras (<https://www.arduino.cc/reference/en/#structure>). A continuación se describen las funciones fundamentales.

### setup()

setup (<https://www.arduino.cc/reference/en/language/structure/sketch/setup/>) es la función que se ejecuta cuando arranca el programa o "sketch" (nombre de un programa en el contexto Arduino). Se utiliza para inicializar variables, configuración de los pines, funciones de inicialización de librerías, etc. La función setup() solo se ejecuta una vez después de cada encendido (alimentación) o reset de la placa Arduino.

### loop()

loop (<https://www.arduino.cc/reference/en/language/structure/sketch/loop/>) se ejecuta después de la función setup. El código dentro de esta función se ejecuta de forma consecutiva y cíclica. Controla la tarjeta Arduino siguiendo las instrucciones del programa.

### delay()

delay() (<https://www.arduino.cc/en/Reference/Delay>) para el programa durante los milisegundos especificados por el parámetro.

Sintaxis:

`delay(ms)`

Parámetros:

ms : número de milisegundos a parar (unsigned long)

Retorno: nada

## pinMode()

[PinMode\(\)](https://www.arduino.cc/en/Reference/PinMode) (<https://www.arduino.cc/en/Reference/PinMode>) configura un [pin](https://www.arduino.cc/en/Tutorial/DigitalPins) (<https://www.arduino.cc/en/Tutorial/DigitalPins>) como entrada (INPUT, o INPUT\_PULLUP) o como salida (OUTPUT).

La resistencia de pull-up interna de un pin se activa con el modo INPUT\_PULLUP. El modo INPUT deshabilita la resistencia de pull-up interna.

Sintaxis:

```
pinMode(pin, mode)
```

Parámetros:

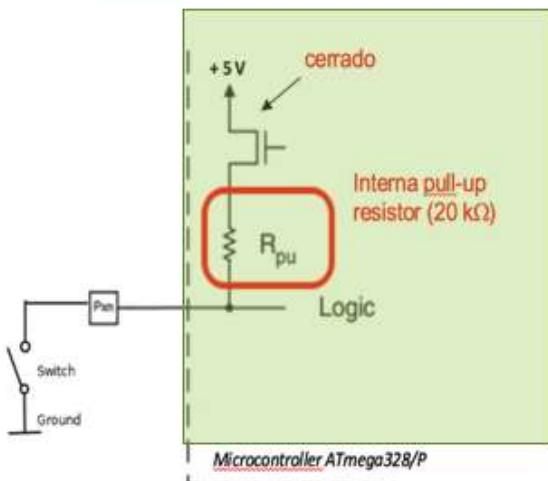
pin : número del pin cuyo modo deseamos fijar

mode : INPUT, OUTPUT, or INPUT\_PULLUP

Retorno: Nada

Puede activar la resistencia interna de pull-up mediante pinMode(INPUT\_PULLUP)

### pinMode(INPUT\_PULLUP)



### pinMode(INPUT)

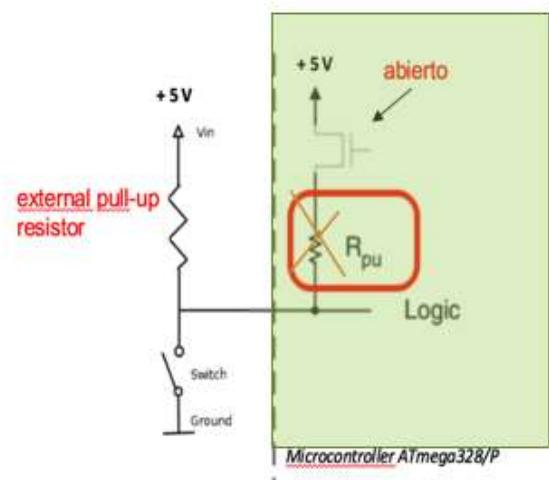


Figure 14. Resistencia de pull-up interna y externa

## digitalRead()

[digitalRead\(pin\)](https://www.arduino.cc/reference/en/language/functions/digital-io/digitalread/) (<https://www.arduino.cc/reference/en/language/functions/digital-io/digitalread/>) lee el valor digital presente sobre el pin especificado.

Sintaxis:

```
digitalRead(pin)
```

Parámetros:

pin : número del pin digital que deseamos leer (int)

Retorno: HIGH o LOW

## digitalWrite()

[digitalWrite\(\)](https://www.arduino.cc/reference/en/language/functions/digital-io/digitalwrite/) (<https://www.arduino.cc/reference/en/language/functions/digital-io/digitalwrite/>) escribe un valor digital (HIGH o LOW) sobre el pin indicado. Si el pin se ha configurado como OUTPUT, con pinMode(OUTPUT), la tensión del pin será:

- 5V para HIGH
- 0V (ground) para LOW.

Sintaxis:

```
digitalWrite(pin, value)
```

Parámetros:

pin : número del pin

value : HIGH o LOW

Retorno: nada



La configuración por defecto de un pin es INPUT. Para configurar un pin como OUTPUT debe configurarlo explícitamente utilizando pinMode(OUTPUT). Si omite esta configuración y ejecuta la instrucción digitalWrite(HIGH) con el pin configurado como INPUT, no obtendrá un nivel lógico alto (5 V) por el pin, sino que habilitará su resistencia interna de pull-up. Si ejecuta digitalWrite(LOW) con el pin configurado como INPUT, no obtendrá un nivel lógico bajo sino que deshabilitará la resistencia interna de pull-up.

## analogWrite()

[analogWrite\(\)](https://www.arduino.cc/reference/en/language/functions/analog-io/analogwrite/) (<https://www.arduino.cc/reference/en/language/functions/analog-io/analogwrite/>) genera una señal PWM (<https://www.arduino.cc/en/Tutorial/PWM>) sobre un pin con un duty-cycle especificado

Sintaxis:

```
analogWrite(pin, value)
```

Parámetros:

**pin**: el pin sobre el que se genera la señal PWM

**value**: entre 0 (salida fija a 0 V ó 0% duty-cycle) y 255 (salida fija a 5 V ó 100% duty-cycle)

Retorno: nada



No es necesario configurar `pinMode(OUTPUT)` antes de llamar a la función `analogWrite()`. Esta función puede utilizarse sobre los pines 3, 5, 6, 9, 10 y 11 del Arduino NANO.

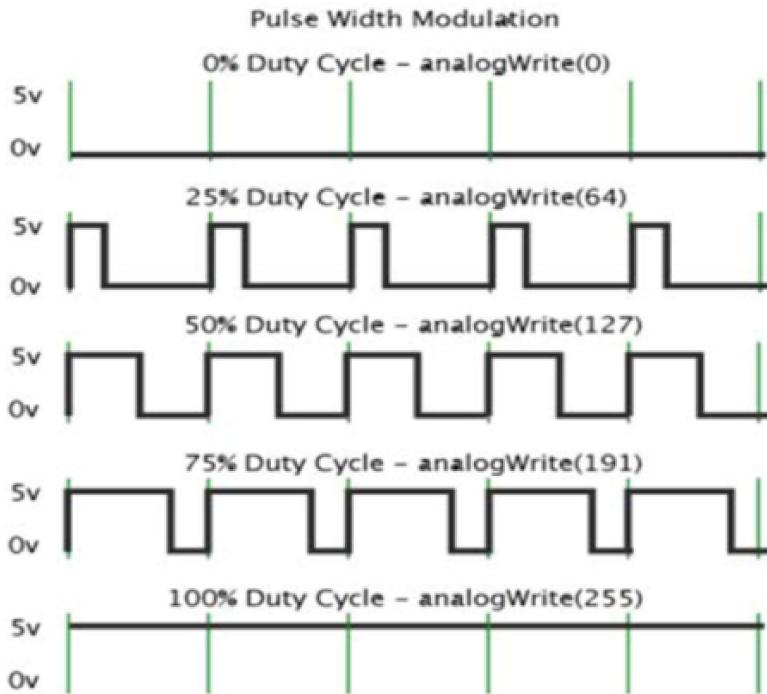


Figure 15. Señal PWM

### `analogRead()`

[analogRead\(\)](https://www.arduino.cc/reference/en/language/functions/analog-io/analogread/) (<https://www.arduino.cc/reference/en/language/functions/analog-io/analogread/>) lee el valor de tensión presente sobre el pin de entrada (canal) especificado y lo convierte en un número digital comprendido entre 0 y 1023.

Sintaxis:

```
analogRead(pin)
```

Parámetros:

**pin**: canal o pin de entrada del convertidor (entre 0 y 7) desde el que se realiza la lectura.

Retorno: número entre 0 a 1023 (int)

El Arduino NANO dispone de un [convertidor A/D](https://microchipdeveloper.com/8bit:adc) (<https://microchipdeveloper.com/8bit:adc>) de 10-bits y 8 canales analógicos de entrada, identificados como: A0 ... A7. El A/D permite que una tensión comprendida entre GND (0 V) y VREF (5 V), su margen de entrada por defecto, se convierta linealmente en  $2^n$  pasos o códigos distintos. El menor

código leido es "0" y el más alto  $2^n - 1$  "1023" para  $n=10$ . La función [analogReference\(\)](#) (<https://www.arduino.cc/reference/en/language/functions/analog-io/analogreference/>) permite modificar el valor de VREF y, por tanto, el margen de entrada del convertidor.

En un convertidor A/D, el valor del bit menos significativo recibe el nombre de "LSB" y su valor asociado de tensión depende del margen de entrada del convertidor. Por ejemplo, en un convertidor de 10 bits con un margen de entrada entre 0 V y 5 V, a 1 LSB le corresponde un valor de tensión de:

$$LSB = \frac{\text{Margen}}{2^{10}} = \frac{5V}{1024} = 4.88mV$$

Esta cantidad es la resolución de tensión del convertidor e indica la cantidad mínima que debe incrementarse la señal a convertir para que se modifique el bit menos significativo del convertidor.

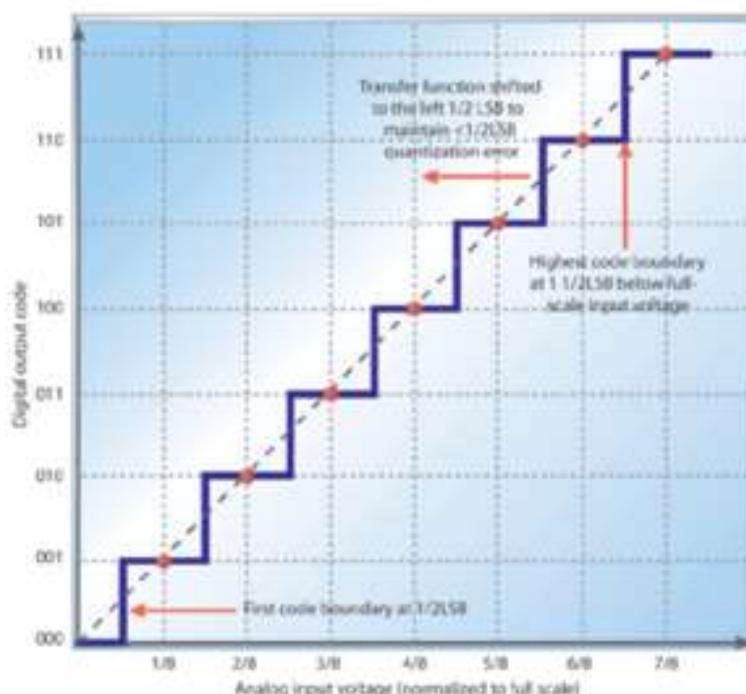


Figure 16. A/D ideal de 3 bits (000 a 111)

El margen de entrada del convertidor puede modificarse usando la función [analogReference\(\)](#) (<https://www.arduino.cc/reference/en/language/functions/analog-io/analogreference/>)

El tiempo de conversión del A/D depende de la frecuencia (o periodo) de reloj del convertidor A/D. La frecuencia de reloj del sistema (16MHz) se divide (prescaler) entre 2, 4, 8, ... hasta 128, para obtener la señal de reloj que utiliza el A/D. El circuito de aproximaciones sucesivas requiere una frecuencia de entrada de 50 kHz a 200 kHz para mantener la máxima resolución. Si se requiere una resolución menor de 10 bits, la frecuencia del conversor puede incrementarse para alcanzar una mayor tasa de muestreo. La librería de analogRead0 utiliza un prescaler de 128 ( $16\text{Mhz}/128 = 125\text{ KHz} < 200\text{ KHz}$ ) pues es el único divisor que garantiza la máxima resolución.



La primera conversión utiliza 25 ciclos del reloj del A/D (tiempo necesario para inicializar el A/D) pero las conversiones sucesivas solo utilizan 13 ciclos de reloj. La función analogRead0 utiliza por defecto el divisor por 128 y su periodo de reloj es de 8 us (16 Mhz/128). Una conversión de 13 ciclos supone 104 us (8us\*13), luego podría realizar aproximadamente hasta 9.615 conversiones por segundo (1/104us) con esta configuración. Puede ver más detalles aquí o consulte el [data-sheet](#)

([http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P\\_Datasheet.pdf](http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf))  
del ATmega328P.

El código digital que se corresponde con la señal de entrada se calcula como:

$$\text{código\_digital} = \text{VIN} \cdot 1023/\text{Vref}$$

donde:

$\text{Vref}$  : tensión de referencia utilizado por el convertidor; por defecto 5 V.

$\text{VIN}$  : tensión analógica sobre el canal de entrada.

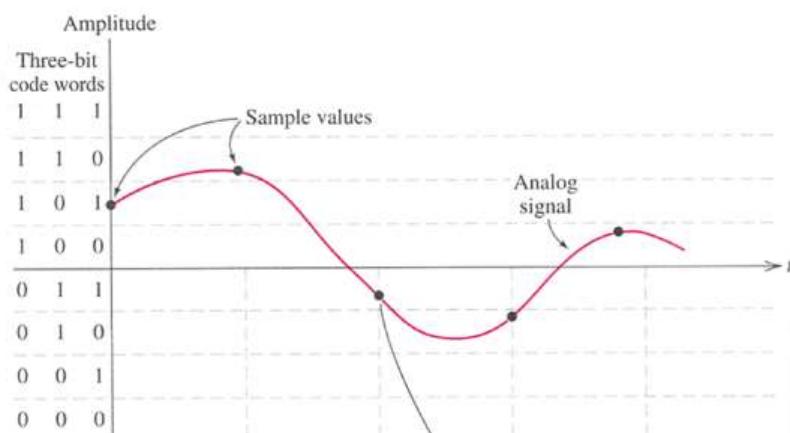


Figure 17. Conversión A/D

### analogReference()

[analogReference\(\)](https://www.arduino.cc/reference/en/language/functions/analog-io/analogreference/) configura la referencia de tensión utilizada por el convertidor. Determina el margen de entrada del convertidor.

Sintaxis:

`analogReference(type)`

Parámetros:

type:

DEFAULT

referencia de 5 V (valor por defecto de alimentación, Vcc)

INTERNAL

referencia interna de 1,1 V (ATmega328)

EXTERNAL

la referencia es la tensión sobre el pin AREF (entre 0 V y 5 V)

Retorno: nada



La relación entre la entrada del convertidor y la salida depende del valor de la referencia de tensión. La exactitud de esta referencia es casi siempre el factor determinante en la exactitud del ADC.

## Resumen de funciones

### ARDUINO CHEAT SHEET

Content for this Cheat Sheet provided by Gavin from Robots and Dinosaurs.  
For more information visit: <http://arduino.cc/en/Reference/Extended>



#### Structure

`void setup() void loop()`

#### Control Structures

```
if (x<5) { } else { }
switch (myvar) {
    case 1:
        break;
    case 2:
        break;
    default:
}
for (int i=0; i <= 255; i++)()
while (x<5) {
do {} while (x<5);
continue; // Go to next in
do/while loop
return x; // Or 'return;' for voids.
goto // considered harmful :-)
```

#### Further Syntax

```
// (single line comment)
/* (multi-line comment) */
#define DOZEN 12 // Not baker's!
#include <avr/pgmspace.h>
```

#### General Operators

```
= (assignment operator)
+ (addition) - (subtraction)
* (multiplication) / (division)
% (modulo)
== (equal to) != (not equal to)
< (less than) > (greater than)
<= (less than or equal to)
>= (greater than or equal to)
&& (and) || (or) ! (not)
```

#### Pointer Access

```
& reference operator
* dereference operator
```

#### Bitwise Operators

```
& (bitwise and) | (bitwise or)
^ (bitwise xor) ~ (bitwise not)
<< (bitshift left) >> (bitshift right)
```

#### Compound Operators

```
++ (increment) -- (decrement)
+= (compound addition)
-= (compound subtraction)
*= (compound multiplication)
/= (compound division)
&= (compound bitwise and)
|= (compound bitwise or)
```

#### Constants

```
HIGH | LOW
INPUT | OUTPUT
true | false
143 // Decimal number
0173 // Octal number
0b11011111 // Binary
0x7B // Hex number
7U // Force unsigned
10L // Force long
15UL // Force long unsigned
10.0 // Forces floating point
2.4e5 // 240000
```

#### Data Types

```
void
boolean (0, 1, false, true)
char (e.g. 'a' -128 to 127)
unsigned char (0 to 255)
byte (0 to 255)
int (-32,768 to 32,767)
unsigned int (0 to 65535)
word (0 to 65500 (0 to 65535)
long (-2,147,483,648 to
2,147,483,647)
unsigned long (0 to 4,294,967,295)
float (-3.4028235E+38 to
3.4028235E+38)
double (currently same as float)
sizeof(myint) // returns 2 bytes
```

#### Strings

```
char S1[15];
char S2[8]={'a','r','d','u','i','n','o'};
char S3[8]={'a','r','d','u','i','n','o','\0'};
//Included \0 null termination
char S4[] = "arduino";
char S5[8] = "arduino";
char S6[15] = "arduino";
```

#### Arrays

```
int myInts[6];
int myPins[] = {2, 4, 8, 3, 6};
int mySensVals[6] = {2, 4, -8, 3, 2};
```

#### Conversion

```
char() byte()
int() word()
long() float()
```

#### Qualifiers

```
static // persists between calls
volatile // use RAM (nice for ISR)
const // make read-only
PROGMEM // use flash
```

#### Digital I/O

```
pinMode(pin, [INPUT,OUTPUT])
digitalWrite(pin, value)
int digitalRead(pin)
/Write High to inputs to use pull-up res
```

#### Analog I/O

```
analogReference(DEFAULT,
INTERNAL,EXTERNAL)
int analogRead(pin) //Call twice if
switching pins from high Z source.
analogWrite(pin, value) // PWM
```

#### Advanced I/O

```
tone(pin, freqHz)
tone(pin, freqHz ,duration_ms)
noTone(pin)
shiftOut(dataPin, clockPin,
[MSBFIRST,LSBFIRST], value)
unsigned long pulseIn(pin,[HIGH,LOW])
```

#### Time

```
unsigned long millis() // 50 days overflow.
unsigned long micros() // 70 min overflow
delay(ms)
delayMicroseconds(us)
```

#### Math

```
min(x, y) max(x, y) abs(x)
constrain(x, minval, maxval)
map(val, fromL, fromH, toL, toH)
pow(base, exponent) sqrt(x)
sin(rad) cos(rad) tan(rad)
```

#### Random Numbers

```
randomSeed(seed) // Long or int
long random(max)
long random(min, max)
```

#### Bits and Bytes

```
lowByte()
highByte()
bitRead(x,bitnr)
bitWrite(x,bitn,bit)
bitSet(x,bitn)
bitClear(x,bitn)
bit(bitn) //bitn: 0-LSB 7-MSB
```

#### External Interrupts

```
attachInterrupt(interrupt, function,
[LOW,CHANGE,RISING,FALLING])
detachInterrupt(interrupt)
interrupts()
noInterrupts()
```

#### Libraries:

	ATMega168	ATMega328	ATMega1280
Flash (2k for bootloader)	16kB	32kB	128kB
SRAM	1kB	2kB	8kB
EEPROM	512B	1kB	4kB

	Duemilanove/ Nano/Pro/ Pro Mini	Mega
# of IO	14 + 6 analog (Nano has 14 + 8)	54 + 16 analog
Serial Pins	0 - RX 1 - TX	0 - RX1 1 - TX1 18 - RX2 19 - TX2 17 - RX3 16 - TX3 15 - RX4 14 - TX4
Ext Interrupts	2 - (Int 0) 1 - (Int 1)	2,3,21,20,19,18 (IRQ0 - IRQ5)
PWM Pins	5,6 - Timer 0 9,10 - Timer 1 3,11 - Timer 2	0 - 13
SPI	10 - SS 11 - MOSI 12 - MISO 13 - SCK	53 - SS 51 - MOSI 50 - MISO 52 - SCK
I2C	Analog4 - SDA Analog5 - SCK	20 - SDA 21 - SCL

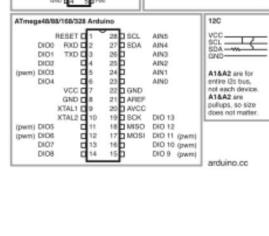
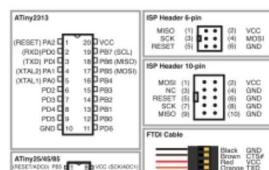


Figure 18. Arduino resumen de funciones

Puede ver tutoriales con diversas aplicaciones [aqui](https://www.arduino.cc/en/Tutorial/HomePage) (<https://www.arduino.cc/en/Tutorial/HomePage>)

## Interrupciones

Las interrupciones son un mecanismo hardware del uC que permite suspender la ejecución del programa principal y ejecutar una subrutina cuando se produzca un suceso o evento particular.

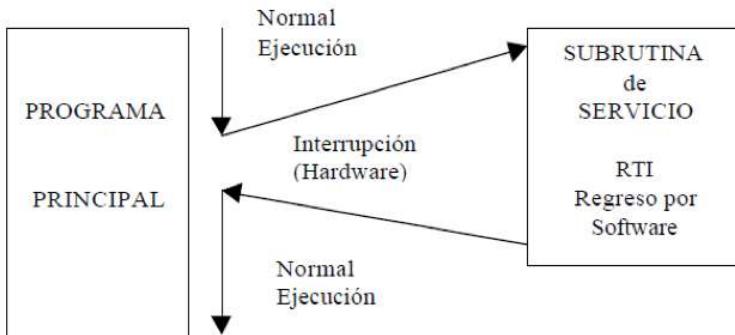


Figure 19. Funcionamiento de una interrupción.

Por ejemplo, puede desencadenarse una interrupción si por un pin se produce un flanco de bajada, o de subida, o cambia el estado, o un módulo del uC concluye una tarea, etc. Los distintos sucesos que pueden generar una interrupción se denominan **fuentes de interrupción**. Cuando el uC detecta un suceso, suspende la ejecución del programa principal y ejecuta una rutina especial denominada **rutina de servicio de interrupción** (o ISR, Interrupt Service Routine). Cuando finaliza la ISR, el programa reanuda su ejecución desde el punto en que tuvo lugar la interrupción.

En general, el código de respuesta a una interrupción suele ser de poca extensión para que el programa principal recupere el control lo antes posible. La ventaja de las interrupciones radica en que no es preciso que nuestro programa utilice tiempo de CPU comprobando constantemente si se produce un evento determinado (técnica de encuesta o polling).

¿Qué sucesos pueden interrumpir al uC?

Los sucesos que pueden desencadenar una interrupción son variados: un flaco, un cambio de estado o un nivel determinado sobre un pin del uC, el fin de conversión del A/D, el overflow de temporizador, el fin de transmisión o recepción de un carácter desde la UART, ...

A cada uno de estos sucesos se le asocia una función (o ISR) de tratamiento cuya dirección de comienzo reside en una tabla con tantas filas como fuentes de interrupción. Esta tabla se denomina **tabla de vectores de interrupción**.

Vector No	Program Address <sup>(2)</sup>	Source	Interrupts definition
1	0x0000 <sup>(1)</sup>	RESET	External Pin, Power-on Reset, Brown-out Reset and Watchdog System Reset
2	0x0002	INT0	External Interrupt Request 0
3	0x0004	INT1	External Interrupt Request 0
4	0x0006	PCINT0	Pin Change Interrupt Request 0
5	0x0008	PCINT1	Pin Change Interrupt Request 1
6	0x000A	PCINT2	Pin Change Interrupt Request 2
7	0x000C	WDT	Watchdog Time-out Interrupt
8	0x000E	TIMER2_COMPA	Timer/Counter2 Compare Match A
9	0x0010	TIMER2_COMPB	Timer/Couther2 Compare Match B
10	0x0012	TIMER2_OVF	Timer/Counter2 Overflow
11	0x0014	TIMER1_CAPT	Timer/Counter1 Capture Event
12	0x0016	TIMER1_COMPA	Timer/Counter1 Compare Match A
13	0x0018	TIMER1_COMPB	Timer/Coutner1 Compare Match B
14	0x001A	TIMER1_OVF	Timer/Counter1 Overflow
15	0x001C	TIMER0_COMPA	Timer/Counter0 Compare Match A
16	0x001E	TIMER0_COMPB	Timer/Coutner0 Compare Match B
17	0x0020	TIMER0_OVF	Timer/Counter0 Overflow
18	0x0022	SPI STC	SPI Serial Transfer Complete
19	0x0024	USART_RX	USART Rx Complete
20	0x0026	USART_UDRE	USART Data Register Empty
21	0x0028	USART_TX	USART Tx Complete
22	0x002A	ADC	ADC Conversion Complete
23	0x002C	EE READY	EEPROM Ready
24	0x002E	ANALOG COMP	Analog Comparator

Figure 20. Vectores de interrupción del ATmega328/P.

#### Detección de evento externos

Existe dos formas de detectar eventos externos sobre los pines del uC. La primera es mediante las **interrupciones externas**, a través de un cambio, un flanco o un nivel bajo sobre los pines INT0 (pin 2) y INT1 (pin 3) del uC (NANO). Los pines INT0 e INT1 disponen de su propio vector de interrupción. La otra posibilidad para detectar eventos externos es mediante las denominadas **Pin Change INTerrupts**, pines PCINTxx. Casi cualquier pin del NANO puede generar una interrupción de este tipo si está habilitado y se produce un cambio del nivel lógico del pin. No se dispone de un vector de interrupción por cada pin PCINTxx, sino que varios pines se agrupan y comparten el mismo vector. Existen tres vectores de interrupción (PCINT0, PCINT1 y PCINT2) cuya función ISR dentro de un programa se especifica como: ISR (PCINT0\_vect), ISR (PCINT1\_vect) y ISR (PCINT2\_vect). La rutina ISR (PCINT2\_vect), por ejemplo, se llama si alguno de los pines comprendidos entre PCINT16 a PCIN23 se encuentra habilitado y cambia de estado. Lo mismo sucede con ISR (PCINT1\_vect) pero para los pines PCINT14 al PCINT8 y con ISR (PCINT0\_vect) para PCINT7 al PCINT0. Los registros del uC PCMSK2, PCMSK1 y PCMSK0 (registros de máscara) controlan qué pines se habilitan para permitir la interrupción por cambio de pin. En resumen, las interrupciones externas son activadas por los pines INT o cualquiera de los pines PCINTxx. Además, si están habilitadas, las interrupciones se dispararán incluso si los pines INT o PCINT están configurados como salidas. Esta característica proporciona una forma de generar una interrupción por software.

Alguna de las prácticas utilizan las **interrupciones externas** a través de los pines **INT0** e **INT1**. Los pines asociados con estas interrupciones externas dependen de la placa utilizada.

*Table 1. Pines de interrupciones externas*

BOARD	Digital pin usado para interrupción
Uno, Nano, Mini, other 328-based	2, 3
Uno WiFi Rev.2	all digital pins
Mega, Mega2560, MegaADK	2, 3, 18, 19, 20, 21
Micro, Leonardo, other 32u4-based	0, 1, 2, 3, 7
Zero	all digital pins, except 4
MKR Family boards	0, 1, 4, 5, 6, 7, 8, 9, A1, A2
Due	all digital pins

La programación de interrupciones externas es muy sencilla, solo debemos indicar qué tipo de suceso desencadena la interrupción, por ejemplo, un flaco de bajada o de subida, o un nivel bajo, sobre el pin INT0/INT1, y proporcionar el nombre de la rutina que atenderá dicho suceso. Todo esto se realiza mediante la función [attachInterrupt\(\)](#) (<https://www.arduino.cc/reference/en/language/functions/external-interrupts/attachinterrupt/>).

### Configuración de Interrupciones externas

```
attachInterrupt(digitalPinToInterrupt(pin), ISR, mode)
```

#### Parámetros

**pin**      numero del pin.

**ISR**      rutina de servicio de interrupción (ISR) que se llamará cuando suceda el evento que dispara la interrupción. Esta función no tiene parámetros y no retorna nada.

**mode**      define el tipo de flanco o nivel que se dispara la interrupción. Se utilizan cuatro constantes predefinidas:

- **LOW** dispara la interrupción cuando el nivel sobre el pin es LOW,
- **CHANGE** dispara la interrupción cuando el nivel sobre el pin cambia,
- **RISING** dispara la interrupción cuando detecta un flanco de subida sobre el pin (low to high)
- **FALLING** dispara la interrupción cuando ocurre un flanco de bajada sobre el pin (high to low),

```
detachInterrupt(digitalPinToInterrupt(pin))
```

#### Parámetros

**pin**      deshabilita la interrupción externa sobre el pin indicado.

### Código de ejemplo

```

const byte ledPin = 13;
const byte interruptPin = 2;
volatile byte state = LOW;

void setup() {
  pinMode(ledPin, OUTPUT);
  pinMode(interruptPin, INPUT_PULLUP);
  attachInterrupt(digitalPinToInterruption(interruptPin), blink, CHANGE); 1
}

void loop() {
  digitalWrite(ledPin, state);
}

void blink() { ..... 2
  state = !state;
}

```

1 configuración de la interrupción

2 ISR, rutina de servicio de interrupción



Dentro de una rutina de interrupción no funciona `delay()` y el valor returnedo por `millis()` no se incrementa. Además, los datos recibidos por el puerto serie pueden perderse.



Cualquier variable global que se modifique dentro de la rutina de interrupción debe declararse como `volatile`.

## Temporizadores

Los temporizadores son módulos del uC dedicados a la medida de tiempo (función de temporizador) o al contejo de eventos (función de contador). El ATMega328P dispone de tres temporizadores denominados: **TIMER0** (<https://microchipdeveloper.com/8bit:timer0>) (8 bits), **TIMER1** (<https://microchipdeveloper.com/8bit:timer1>) (16 bits) y **TIMER2** (<https://microchipdeveloper.com/8bit:timer2>) (8 bits). Un temporizador se implementa mediante un contador. La entrada de reloj del contador se conectada a un generador de reloj de frecuencia fija y conocida (interno o externo al uC) y la salida es un registro que contiene el número de pulsos recibidos por su entrada de reloj (tiempo\_temporizado = #ciclos x periodo). Los temporizadores suelen funcionar incrementando su valor de cuenta hasta llegar a su máxima capacidad de contejo (que depende del número de bits del contador). Al llegar al valor máximo, el siguiente pulso de reloj reinicia el contador. Cuando ocurre este reinicio, se produce un **overflow** del temporizador y puede generar una interrupción. También es posible modificar el periodo de la señal de reloj interna mediante divisores de frecuencia.

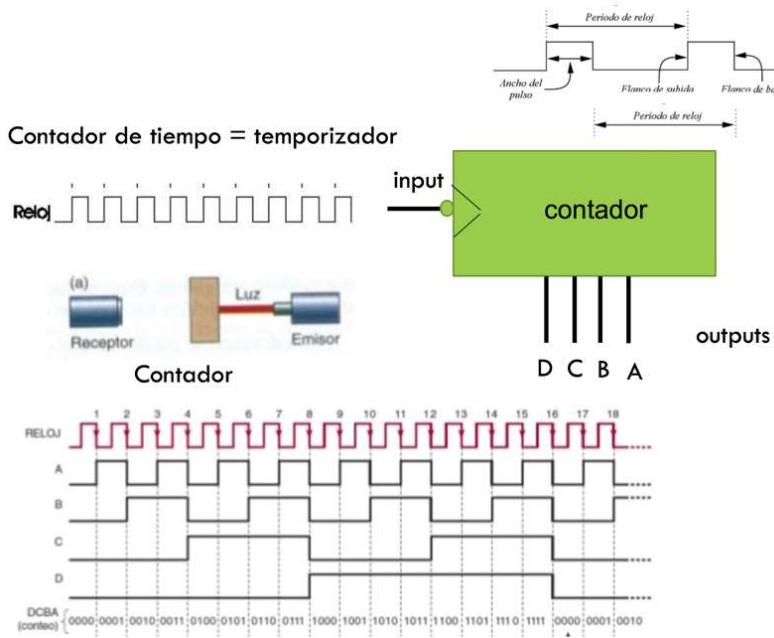


Figure 21. Temporizadores vs contadores

El mismo recurso puede funcionar también como contador cuando por su entrada no se introduce una señal de reloj, sino otra relacionada con la ocurrencia de algún evento (p.e. cuando un objeto sobre una cinta transportadora pasa por una determinada zona se genera un pulso que incrementa el contador).

La configuración de los temporizadores se simplifica mucho recurriendo a librerías. La librería [TimerOne](#) (<https://www.arduinolibraries.info/libraries/timer-one>) puede utilizarse para realizar temporizaciones o para generar una señal PWM con el TIMER1.

Puede ver una descripción de las funciones de esta librería [aquí](#) (<https://playground.arduino.cc/Code/Timer1/>).

Las funciones principales de esta librería se describen a continuación:

`initialize(period)`

You must call this method first to use any of the other methods. You can optionally specify the timer's period here (in microseconds), by default it is set at 1 second. Note that this breaks analogWrite() for digital pins 9 and 10 on Arduino.

`setPeriod(period)`

Sets the period in microseconds. The minimum period or highest frequency this library supports is 1 microsecond or 1 MHz. The maximum period is 8388480 microseconds or about 8.3 seconds. Note that setting the period will change the attached interrupt and both pwm outputs' frequencies and duty cycles simultaneously.

`pwm(pin, duty, period)`

Generates a PWM waveform on the specified pin. Output pins for Timer1 are PORTB pins 1 and 2, so you have to choose between these two, anything else is ignored. On Arduino, these are digital pins 9 and 10, so those aliases also work. Output pins for Timer3 are from PORTE and correspond to 2,3 & 5 on the Arduino Mega. The duty cycle is specified as a 10 bit value, so anything between 0 and 1023. Note that you can optionally set the period with this function if you include a value in microseconds as the last parameter when you call it.

`attachInterrupt(function, period)`

Calls a function at the specified interval in microseconds. Be careful about trying to execute too complicated of an interrupt at too high of a frequency, or the CPU may never enter the main loop and your program will 'lock up'. Note that you can optionally set the period with this function if you include a value in microseconds as the last parameter when you call it.

`setPwmDuty(pin, duty)`

A fast shortcut for setting the pwm duty for a given pin if you have already set it up by calling `pwm0` earlier. This avoids the overhead of enabling pwm mode for the pin, setting the data direction register, checking for optional period adjustments etc. that are mandatory when you call `pwm0`.

`detachInterrupt()`

Disables the attached interrupt.

`disablePwm(pin)`

Turns PWM off for the specified pin so you can use that pin for something else.

`read()`

Reads the time since last rollover in microseconds.

*Código de ejemplo de la librería TimerOne*

```
#include <TimerOne.h>

// This example uses the timer interrupt to blink an LED
// and also demonstrates how to share a variable between
// the interrupt and the main program.

const int led = LED_BUILTIN; // the pin with a LED

void setup(void)
{
  pinMode(led, OUTPUT);
  Timer1.initialize(150000);
  Timer1.attachInterrupt(blinkLED); // blinkLED to run every 0.15 seconds
  Serial.begin(9600);
}

// The interrupt will blink the LED, and keep
// track of how many times it has blinked.
int ledState = LOW;
volatile unsigned long blinkCount = 0; // use volatile for shared variables

void blinkLED(void)
{
  if (ledState == LOW) {
    ledState = HIGH;
    blinkCount = blinkCount + 1; // increase when LED turns on
  } else {
    ledState = LOW;
  }
  digitalWrite(led, ledState);
}

// The main program will print the blink count
// to the Arduino Serial Monitor
void loop(void)
{
  unsigned long blinkCopy; // holds a copy of the blinkCount

  // to read a variable which the interrupt code writes, we
  // must temporarily disable interrupts, to be sure it will
  // not change while we are reading. To minimize the time
  // with interrupts off, just quickly make a copy, and then
  // use the copy while allowing the interrupt to keep working.
  noInterrupts();
  blinkCopy = blinkCount;
  interrupts();

  Serial.print("blinkCount = ");
  Serial.println(blinkCopy);
  delay(100);
}
```



Los temporizadores disponen de dos configuraciones denominadas modo **capture** y modo **compare**. La funcionalidad del modo compare permite lanzar una interrupción o cambiar el estado de un pin cuando el valor del temporizador coincide con el contenido de un registro especial. El modo capture permite registrar el instante (timestamp) en el que se produce un determinado evento, por ejemplo, cuando cambia de estado un pin, puede registrarse ese instante, almacenando en un registro especial el valor actual del temporizador. El modo compare permite generar una señal PWM o una señal de frecuencia variable sobre los pines del uC (Output-Compare Pins): OCnA y OCnB. El modo capture permite medir el periodo o en ancho de un pulso externo.

## Modos de bajo consumo

El **modo de bajo consumo**, o **modo sleep**, es un recurso del uC que permite desconectar alguno de sus módulos internos para reducir el consumo de energía. Este recurso tiene mucha utilidad en los sistemas alimentados a baterías, donde es necesario reducir el consumo para mantener la carga de la batería el mayor tiempo posible. Los modos de ahorro de energía disponibles se indican en la tabla adjunta.

**Table 14-1. Active Clock Domains and Wake-up Sources in the Different Sleep Modes.**

Sleep Mode	Active Clock Domains					Oscillators		Wake-up Sources							Software BOD Disable
	clkCPU	clkFLASH	clkIO	clkADC	clkASY	Main Clock Source Enabled	Timer Oscillator Enabled	INT and PCINT	TWI Address Match	Timer2	SPM/EEPROM Ready	ADC	WDT	Other I/O	
Idle			Yes	Yes	Yes	Yes	Yes <sup>(2)</sup>	Yes	Yes	Yes	Yes	Yes	Yes	Yes	
ADC Noise Reduction				Yes	Yes	Yes	Yes <sup>(2)</sup>	Yes <sup>(3)</sup>	Yes	Yes <sup>(2)</sup>	Yes	Yes	Yes	Yes	
Power-down								Yes <sup>(3)</sup>	Yes					Yes	Yes
Power-save					Yes		Yes <sup>(2)</sup>	Yes <sup>(3)</sup>	Yes	Yes				Yes	Yes
Standby <sup>(1)</sup>						Yes		Yes <sup>(3)</sup>	Yes					Yes	Yes
Extended Standby					Yes <sup>(2)</sup>	Yes	Yes <sup>(2)</sup>	Yes <sup>(3)</sup>	Yes	Yes				Yes	Yes

### Note:

1. Only recommended with external crystal or resonator selected as clock source.
2. If Timer/Counter2 is running in asynchronous mode.
3. For INT1 and INTO, only level interrupt.

*Figure 22. Sleep Modes (reloj activos y fuentes para despertar)*

La siguiente figura ilustra las fuentes principales de reloj del uC y cómo se distribuyen por los diferentes módulos funcionales. Todos los relojes no necesitan estar activos de forma simultánea. Para reducir el consumo se deshabilita la señal de reloj de ciertos módulos, según el modo sleep seleccionado.,

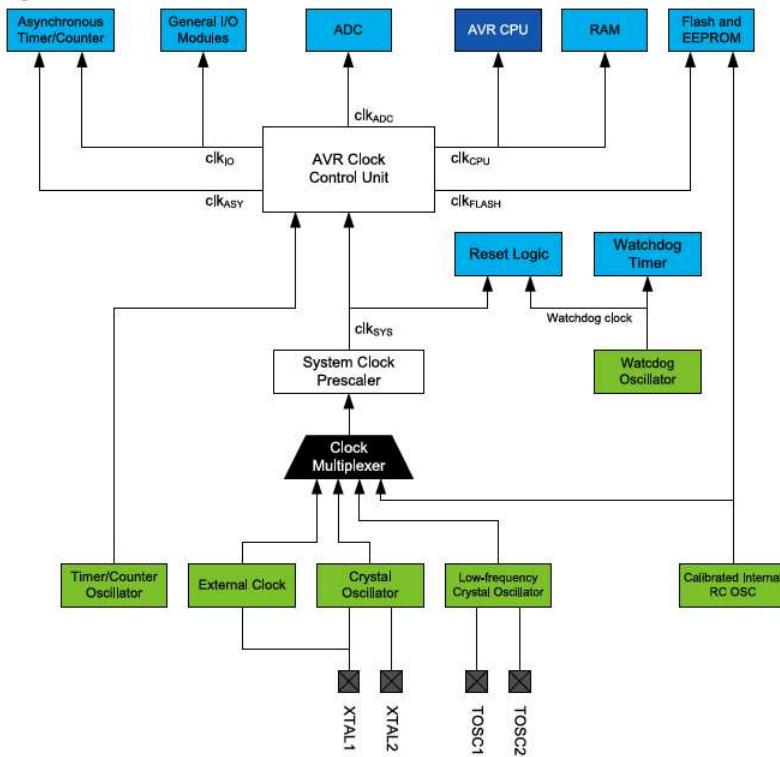


Figure 23. Sistema de clock del uC



Junto a estos modos sleep existe también un registro (**PPR**, Power Reduction Register) que permite deshabilitar el reloj de un módulo del uC en particular. Los módulos cuyo reloj puede deshabilitarse son: I2C, TIMER2, TIMER1, TIMER0, SPI, UART, ADC. Esta funcionalidad puede utilizarse en los modos sleep **Idle** y **Active**. En los otros modos sleep el reloj de los módulos ya está deshabilitado.

La frecuencia de reloj del sistema ( $\text{clk}_{\text{sys}}$ ) se refiere a la frecuencia después del prescaler (system clock prescaler).

La programación de los modos de bajo consumo se simplifica recurriendo a funciones de librería como "LowPower".

#### Funciones de LowPower

```
void idle (period_t period, adc_t adc, timer2_t timer2, timer1_t timer1, timer0_t timer0, spi_t spi,
uart0_t uart, twi_t twi);
void adcNoiseReduction(period_t period, adc_t adc, timer2_t timer2);
void powerDown(period_t period, adc_t adc, bod_t bod);
void powerSave(period_t period, adc_t adc, bod_t bod, timer2_t timer2);
void powerStandby(period_t period, adc_t adc, bod_t bod);
void powerExtStandby(period_t period, adc_t adc, bod_t bod, timer2_t timer2);
```

#### Parámetros de LowPower

period\_t:

- . SLEEP\_15MS,
- . SLEEP\_30MS,
- . SLEEP\_60MS,
- . SLEEP\_120MS,
- . SLEEP\_250MS,
- . SLEEP\_500MS,
- . SLEEP\_1S,
- . SLEEP\_2S,
- . SLEEP\_4S,
- . SLEEP\_8S,
- . SLEEP\_FOREVER

bod\_t:

- . BOD\_OFF,
- . BOD\_0

adc\_t:

- . ADC\_OFF,
- . ADC\_ON

timer2\_t:

- . TIMER2\_OFF,
- . TIMER2\_ON

timer1\_t:

- . TIMER1\_OFF,
- . TIMER1\_ON

timer0\_t:

- . TIMERO\_OFF,
- . TIMERO\_ON

spi\_t:

- . SPI\_OFF,
- . SPI\_ON

uart0\_t:

- . USART0\_OFF,
- . USART0\_ON

twi\_t:

- . TWI\_OFF,
- . TWI\_ON

*Descripción de librería LowPower*

```
// **** INCLUDES *****
#include "LowPower.h"

/*
period_t:

    SLEEP_15MS,
    SLEEP_30MS,
    SLEEP_60MS,
    SLEEP_120MS,
    SLEEP_250MS,
    SLEEP_500MS,
    SLEEP_1S,
    SLEEP_2S,
    SLEEP_4S,
    SLEEP_8S,
    SLEEP_FOREVER

bod_t:

    BOD_OFF,
    BOD_0

adc_t:

    ADC_OFF,
    ADC_ON

timer2_t:

    TIMER2_OFF,
    TIMER2_ON

timer1_t:

    TIMER1_OFF,
    TIMER1_ON

timer0_t:

    TIMERO_OFF,
    TIMERO_ON

spi_t:

    SPI_OFF,
    SPI_ON

uart0_t:

    USART0_OFF,
    USART0_ON

twi_t:
    TWI_OFF,
    TWI_ON

// funciones de la libreria

void idle(period_t period, adc_t adc, timer2_t timer2, timer1_t timer1, timer0_t timer0, spi_t spi, usart0_t
uart0, twi_t twi);
void adcNoiseReduction(period_t period, adc_t adc, timer2_t timer2);
void powerDown(period_t period, adc_t adc, bod_t bod);
void powerSave(period_t period, adc_t adc, bod_t bod, timer2_t timer2);
void powerStandby(period_t period, adc_t adc, bod_t bod);
void powerExtStandby(period_t period, adc_t adc, bod_t bod, timer2_t timer2);
```

```

*/
void setup()
{
    // No setup is required for this library
    Serial.begin(115200);
    pinMode(13, OUTPUT);
    digitalWrite(13,HIGH);
}

void loop()
{
    // Enter idle state for 8 s with the rest of peripherals turned off
    LowPower.idle(SLEEP_8S, ADC_OFF, TIMER2_OFF, TIMER1_OFF, TIMER0_OFF,
                   SPI_OFF, USART0_OFF, TWI_OFF);

    // Hacer algo aqui ...
    // Example: Read sensor, data logging, data transmission.
    //delay(100);
    Serial.println("Hola, en 1 segundo me pongo a dormir...");
    delay(1000);
}

```

### *Ejemplo de modo de bajo consumo despertando por interrupción externa*

```

// **** INCLUDES *****
#include "LowPower.h"

// Use pin 2 as wake up pin
const int wakeUpPin = 2;

void wakeUp()
{
    // Just a handler for the pin interrupt.
}

void setup()
{
    // Configure wake up pin as input.
    // This will consumes few uA of current.
    pinMode(wakeUpPin, INPUT);
}

void loop()
{
    // Allow wake up pin to trigger interrupt on low.
    attachInterrupt(digitalPinToInterrupt(wakeUpPin), wakeUp, LOW);

    // Enter power down state with ADC and BOD module disabled.
    // Wake up when wake up pin is low.
    LowPower.powerDown(SLEEP_FOREVER, ADC_OFF, BOD_OFF);

    // Disable external pin interrupt on wake up pin.
    detachInterrupt(digitalPinToInterrupt(wakeUpPin));

    // Do something here
    // Example: Read sensor, data logging, data transmission.
}

```

### *Ejemplo de modo de bajo consumo despertando periódicamente (temporización)*

```

// **** INCLUDES *****
#include "LowPower.h"

void setup()
{
    // No setup is required for this library
}

void loop()
{
    // Enter power down state for 8 s with ADC and BOD module disabled
    LowPower.powerDown(SLEEP_8S, ADC_OFF, BOD_OFF);

    // Do something here
    // Example: Read sensor, data logging, data transmission.
}

```



Consulte el [datasheet](#) para más detalles

## Watchdog

El **watchdog**, o **perro guardián**, es un temporizado autónomo (dispone de su propio reloj, independiente del reloj del sistema) que resetea el uC si transcurre un determinado tiempo sin que se "refresque". El refresco se realiza invocando a una instrucción especial dentro del programa de usuario. Su función es evitar que el uC se bloquee de forma permanente o entre en un ciclo por un fallo durante su funcionamiento.

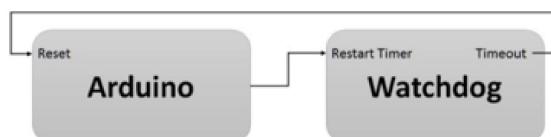


Figure 24. Diagrama bloques de funcionamiento del Watchdog

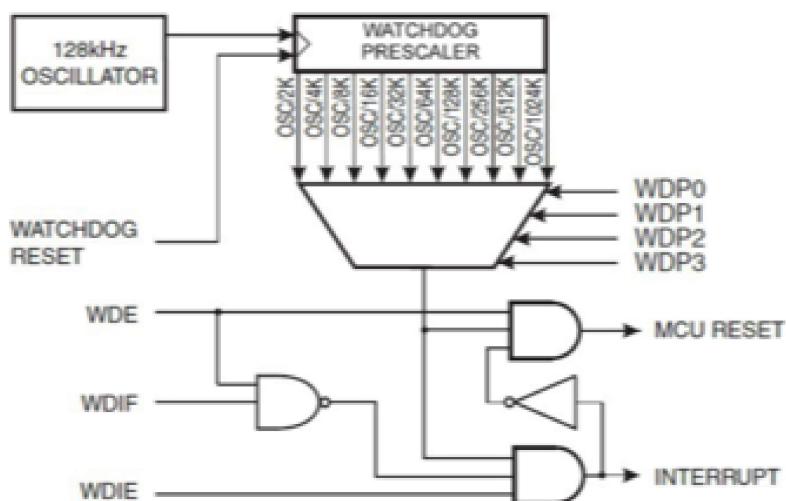


Figure 25. Watchdog o perro guardián

El programa del usuario debe invocar la instrucción de refresco `wdt_reset()` periódicamente en determinados puntos del programa. Si por cualquier circunstancia el programa se cuelga o entra en un bucle no previsto, la instrucción de refresco no se ejecutará, el temporizador expirará, y el uC se reseteará volviendo a un estado conocido. Para gestionar el watchdog, se utiliza la librería `avr/wdt.h`. Se dispone de tres funciones:

`wdt_disable()`

desactiva el watchdog, necesario llamar antes de configurar un nuevo watchdog

```
wdt_enable(tiempo)
```

configura y activa el watchdog con el tiempo de expiración indicado en el parámetro.

```
wdt_reset()
```

renueva el intervalo de cuenta del watchdog.

### Ejemplo de watchdog

```
#include <avr/wdt.h>

int loops = 0;

// watchdog interrupt
ISR (WDT_vect)
{
    Serial.println("Inside Interrupcion del WDT");
} // end of WDT_vect //al salir se desactiva WDIE=0

void setup()
{
    wdt_disable(); //hay que respetar esta secuencia
    wdt_enable(WDTO_8S);

    // ...
    WDTCSR |= 0b11000000; //activa las interrupciones del WDT
    Serial.begin(115200);
    Serial.println("iniciando");
    delay(1000);
}

void loop()
{
    Serial.println("hola " + String(loops));
    delay(1000);
    if (loops > 5) {
        Serial.println("entro en bucle largo");
        for (int i = 0; i < 100; i++) {
            Serial.println(i);
            delay(1000);
        }
    }
    wdt_reset();
    loops++;
}
```



Consulte el [datasheet](#) para más detalles

## Brown-Out Detector(BOD)

**Brown-out detection** es un recurso del uC que le permite detectar si su tensión de alimentación desciende por debajo de un umbral establecido (Brown-out Reset threshold ( $V_{BOT}$ )). Si la tensión desciende de este valor prefijado, el uC se resetea y se mantiene en reset para evitar un comportamiento inestable hasta que la alimentación se recupere.

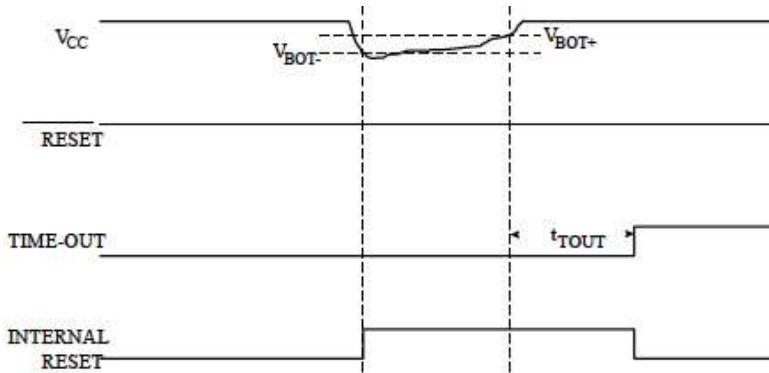


Figure 26. Brown-out Reset

Observe que se considera una histéresis para evitar perturbaciones en la detección. Los niveles de histéresis son  $V_{BOT^+} = V_{BOT} + VHYST/2$  y  $V_{BOT^-} = V_{BOT} - VHYST/2$ . Una vez que se recupera la alimentación se inicia una temporización ( $t_{TOUT}$ ) durante la que se mantiene el uC en Reset.

El umbral  $V_{BOT}$  puede seleccionarse mediante un parámetro denominado **BODLEVEL** que se configura a través de los FUSES del uC. Los FUSES son 3 bytes de memoria no volátil que configuran el microcontrolador, por ejemplo, si se usa un bootloader, qué fuente de reloj se va a utilizar, etc. Los FUSES se configuran con el Arduino IDE durante el proceso de grabación del bootloader (Herramientas → Quemar bootloader) utilizando un programador ISP (In Serial Programmer). La configuración de FUSES se toma del fichero `boards.txt` de la placa correspondiente. También puede configurar los fuses manualmente, sin cargar un bootloader. Para ello se requiere la herramienta `avr-dude`. Por ejemplo, el siguiente comando escribe los FUSES con los valores 0xFF, 0xDE y 0x05:

```
avrdude -c stk500v2 -p m328p -P /dev/ttyACM0 -b 19200 -U lfuse:w:0xFF:m -U hfuse:w:0xDE:m -U efuse:w:0x05:m
```

Puede ver los fuses con la utilidad [AVR fuse calculator](http://www.engbedded.com/fusecalc/) (<http://www.engbedded.com/fusecalc/>). Junto a los FUSES, el uC dispone de 6 bits denominados **lock bits** que restringen el acceso a la memoria de programa.



Consulte más detalles en la página 348 del [datasheet](#).

## Buses de comunicación del uC

Los principales buses de comunicación local del uC se indican en la tabla adjunta:

Protocol	Transfer Type	# of Wires	# of Peripherals	Transfer Speed
I <sup>2</sup> C	Synchronous	2	Up to 127	Low
SPI	Synchronous	4+	Unlimited	High
UART	Asynchronous	2 or 4	1	Medium

Figure 27. Comunicación serie

### Comunicación serie

La comunicación **serie** o secuencial es el proceso de envío de datos bit a bit, de forma secuencial, sobre un canal de comunicación o bus. La transmisión serie puede realizarse de forma **síncrona** o **asíncrona** según se utilice o no una línea de reloj independiente para sincronizar los bits transmitidos. La comunicación serie síncrona utiliza, además de la línea de datos, una de reloj para indicar al receptor los instantes de tiempo en los que debe leer la señal.

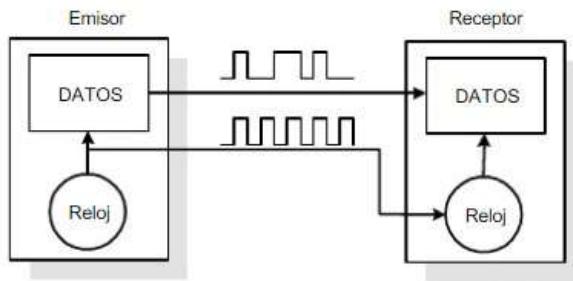


Figure 28. Comunicación serie síncrona

La comunicación **serie asíncrona** está basada en la transmisión secuencial de datos bit a bit utilizando una línea de datos pero sin linea de reloj independiente.

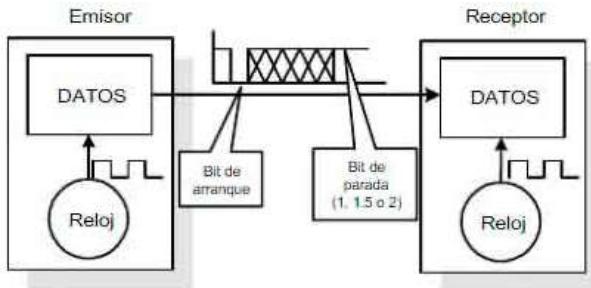


Figure 29. Comunicación serie asíncrona

Para realizar la transmisión de forma adecuada, transmisor y receptor deben acordar: una velocidad de transmisión común, la longitud de trama, la duración de los bits de "start" y "stop" y la presencia o no de un bit de paridad -utilizado para el control de errores-. El sincronismo se realiza mediante el bit de "Start". El receptor sincroniza su reloj con el transmisor usando el bit (Star bit) que llega al principio de cada carácter. La figura muestra un ejemplo de este tipo de transmisión.

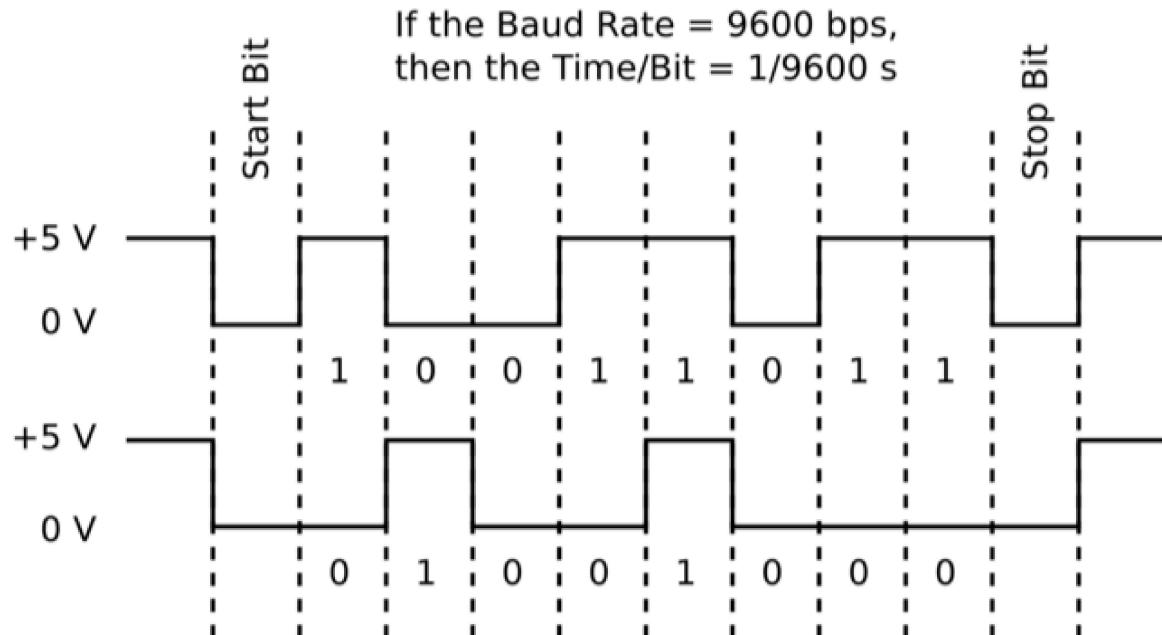


Figure 30. Comunicación serie asíncrona

El receptor detecta el flanco de bajada del bit de start y sincroniza la fase de su reloj para que los flancos de lectura se produzcan, aproximadamente, en el medio de cada bit recibido.

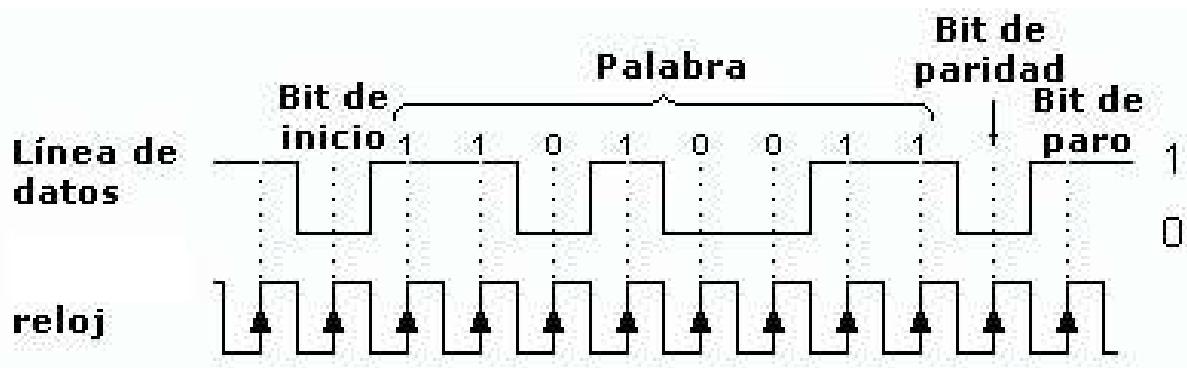


Figure 31. Comunicación serie asincrónica

En general, este tipo de comunicación se realiza utilizando un circuito integrado denominado UART (Universal Asynchronous Receiver Transceiver). La [UART](https://es.wikipedia.org/wiki/Universal_Asyncronous_Receiver-Transmitter) ([https://es.wikipedia.org/wiki/Universal\\_Asyncronous\\_Receiver-Transmitter](https://es.wikipedia.org/wiki/Universal_Asyncronous_Receiver-Transmitter)) es el componente clave del subsistema de comunicaciones serie asincrónica de los microcontroladores. Una UART contiene como bloque básico un [registro de desplazamiento](https://es.wikipedia.org/wiki/Registro_de_desplazamiento) ([https://es.wikipedia.org/wiki/Registro\\_de\\_desplazamiento](https://es.wikipedia.org/wiki/Registro_de_desplazamiento)) encargado de la conversión serie-paralelo para la recepción y paralelo-serie para la transmisión. El transmisor consiste en un buffer de escritura, un registro de desplazamiento, un generador de paridad y otra lógica de control para el manejo de diferentes tipos de formato de la trama serie. El receptor es la parte más compleja de la UART debido a su reloj y la unidad de recuperación de datos. En la figura se muestran dos líneas de datos para la comunicación serie: una para transmitir (TxDn) y otra para recibir (RxDn).



Puede consultar más detalles en la página 224 del [datasheet](#).

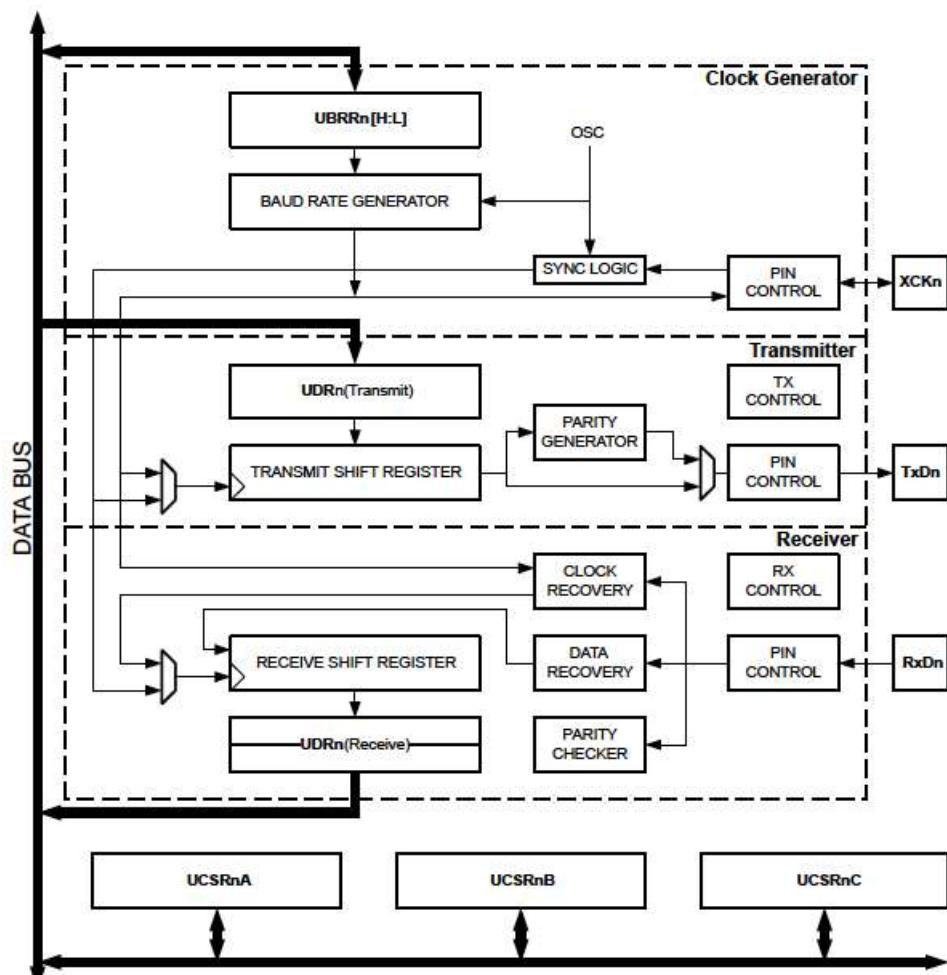


Figure 32. Comunicación serie

El módulo UART trabaja, generalmente, con niveles lógicos 0 V - 5 V y se sirve de drivers para convertir las señales lógicas a niveles eléctricos más apropiados para transmitir la señal como: [RS-232](https://es.wikipedia.org/wiki/RS-232) (<https://es.wikipedia.org/wiki/RS-232>), [RS-422](https://es.wikipedia.org/wiki/RS-422) (<https://es.wikipedia.org/wiki/RS-422>) ó [RS-485](https://es.wikipedia.org/wiki/RS-485) (<https://es.wikipedia.org/wiki/RS-485>).

La librería para manejar el bus Serie en Arduino es [Serial](#) (<https://www.arduino.cc/reference/en/language/functions/communication/serial/>)

Habitualmente, utilizaremos este tipo de comunicación serie para enviar mensajes de texto desde nuestro programa a un emulador de terminal. Arduino proporciona soporte hardware (UART del ATMega) para la comunicación serie sobre los pines 0 (TX) y 1 (RX) de la placa NANO. Estos pines están conectados, a su vez, a un driver TTL-USB (CH340G de WCH o FT232 de FTDI) que convierte los niveles TTL provenientes de la UART a los niveles eléctricos del bus USB para comunicarse con un PC. La UART del uC funciona como hardware independiente y es posible, por ejemplo, recibir datos serie mientras el uC ejecuta otra tarea.



Es posible replicar la funcionalidad de la UART por software y realizar la transmisión sobre otros pines del uC, o incluso disponer de múltiples puertos serie. En este caso, se requiere la librería [SoftwareSerial.h](#) (<https://www.arduino.cc/en/Reference/SoftwareSerial>).

### Ejemplo de comunicación serie

```
/*
 * DigitalReadSerial
 *
 * Reads a digital input on pin 2, prints the result to the Serial Monitor
 *
 * This example code is in the public domain.
 *
 * http://www.arduino.cc/en/Tutorial/DigitalReadSerial
 */

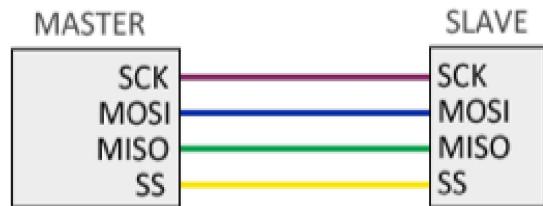
// digital pin 2 has a pushbutton attached to it. Give it a name:
int pushButton = 2;

// the setup routine runs once when you press reset:
void setup() {
    // initialize serial communication at 9600 bits per second:
    Serial.begin(9600);
    // make the pushbutton's pin an input:
    pinMode(pushButton, INPUT);
}

// the loop routine runs over and over again forever:
void loop() {
    // read the input pin:
    int buttonState = digitalRead(pushButton);
    // print out the state of the button:
    Serial.println(buttonState);
    delay(1); ..... // delay in between reads for stability
}
```

### SPI

[SPI](#) ([https://es.wikipedia.org/wiki/Serial\\_Peripheral\\_Interface](https://es.wikipedia.org/wiki/Serial_Peripheral_Interface)) es un estándar de comunicaciones, usado principalmente para la transferencia de información entre circuitos integrados de equipos electrónicos. Incluye una línea de reloj, dato entrante, dato saliente y un pin de "chip select", que selecciona o desSelecciona el dispositivo con el que se desea comunicar. El SPI se implementa mediante un registro de desplazamiento donde se captura un bit con cada pulso de reloj.



- **MOSI** (Master-out, slave-in) para la comunicación del maestro al esclavo.
- **MISO** (Master-in, slave-out) para comunicación del esclavo al maestro.
- **SCK** (Clock) señal de reloj enviada por el maestro.

Figure 33. Comunicación serie

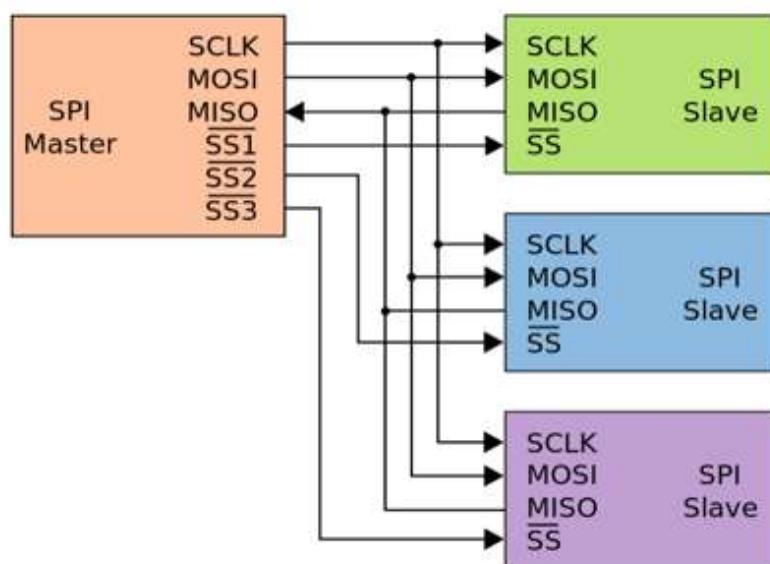


Figure 34. Comunicación serie

El bus SPI se define mediante 4 pines:

**SCLK o SCK**

Señal de reloj del bus. Esta señal controla la velocidad a la que se transmite cada bit.

**MISO(Master Input Slave Output)**

Por esta línea el maestro recibe los datos transmitidos desde el dispositivo esclavo (p.e. un sensor).

**MOSI(Master Output Slave Input)**

Por esta línea el maestro transmite los datos al dispositivo esclavo (p.e. un sensor).

**SS o CS**

Chip Select o Slave Select, habilita el dispositivo hacia el que se envían los datos.



A diferencia de otros buses, el SPI no implementa el nivel del enlace entre dispositivos, es decir, no hay un campo para la dirección, ni un campo para ACK, etc. No se precisa un direccionamiento puesto que habilitamos el integrado al que queremos enviar los datos mediante la señal Chip select.

La librería para manejar el bus SPI en Arduino es [SPI.h](https://www.arduino.cc/en/Reference/SPI) (<https://www.arduino.cc/en/Reference/SPI>)

I<sup>2</sup>C (<https://es.wikipedia.org/wiki/I%20C>) es un bus de comunicación serie de baja velocidad y con solo dos hilos (TWI, Two Wire Interface) para la conexión de dispositivos de baja velocidad como microcontroladores, EEPROMs, A/D and D/A converters, I/O interfaces y otros periféricos similares habituales en sistemas empotrados. Sigue un protocolo maestro-esclavo. I<sup>2</sup>C precisa de dos líneas de señal: reloj (CLK, Serial Clock) y la línea de datos (SDA, Serial Data). Ambas líneas necesitan resistencias de pull-up. Los dispositivos conectados al bus I<sup>2</sup>C tienen una dirección única y pueden ser maestros o esclavos. El dispositivo maestro inicia la transferencia de datos y genera la señal de reloj. El primer byte enviado por el maestro se corresponde con la dirección del esclavo. Los primeros 7 bits representan la dirección y el octavo bit (R/W-Bit) indica al esclavo si debe recibir datos del maestro (0) o enviar datos al maestro (1). Por tanto, I<sup>2</sup>C utiliza un espacio de direccionamiento de 7 bits, permitiendo hasta 112 nodos sobre el bus (16 de las 128 direcciones posibles están reservadas para fines especiales).

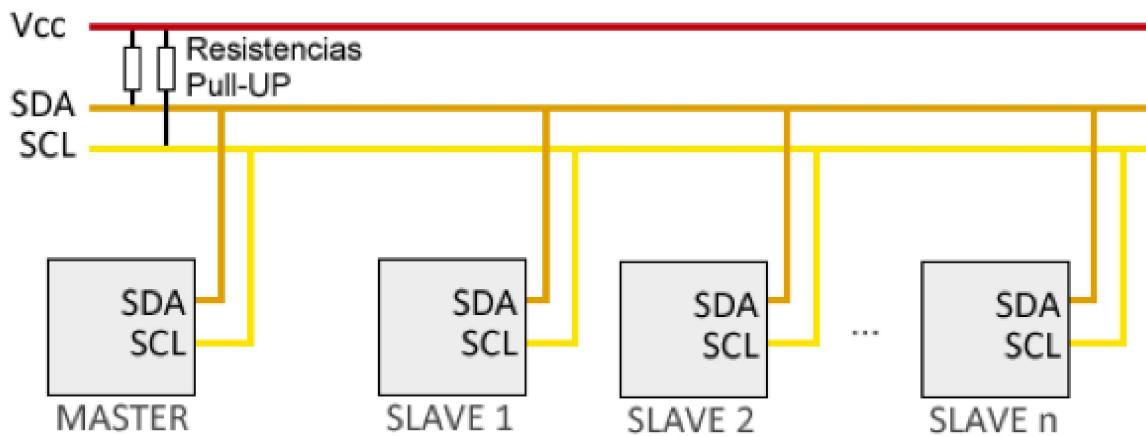


Figure 35. Comunicación serie

Las transacciones en el bus I<sup>2</sup>C tienen este formato:

| start | A7 A6 A5 A4 A3 A2 A1 R/W | ACK | ... DATA ... | ACK | stop | idle |

La librería del bus I<sup>2</sup>C en Arduino se denomina [wire.h](https://www.arduino.cc/en/reference/wire) (<https://www.arduino.cc/en/reference/wire>)

## Conversión Analógica Digital y Digital Analógica

### Conceptos básicos

Una **señal eléctrica** es una alteración en una tensión, corriente o potencia, u otra magnitud eléctrica, empleada para transmitir información. Las señales eléctricas pueden ser analógicas o digitales. Las señales analógicas pueden tomar cualquier valor dentro de un margen determinado, pudiendo ser continuas, si están definidas en cualquier instante de tiempo, o discontinuas, en el caso de que existan en instantes concretos de tiempo, no necesariamente equiespaciados. La información puede residir en la amplitud o en otro parámetro de la señal, como la frecuencia o la fase. Las señales digitales solo pueden tomar uno entre los valores de un conjunto discreto (discontinuo) y finito. Generalmente, el conjunto discreto se limita a dos valores, denominados "1" digital y "0" digital. La información puede estar en la secuencia de unos y ceros, caso de una señal en formato serie, o en la combinación de unos y ceros de un conjunto de líneas eléctricas, caso de las señales en paralelo.

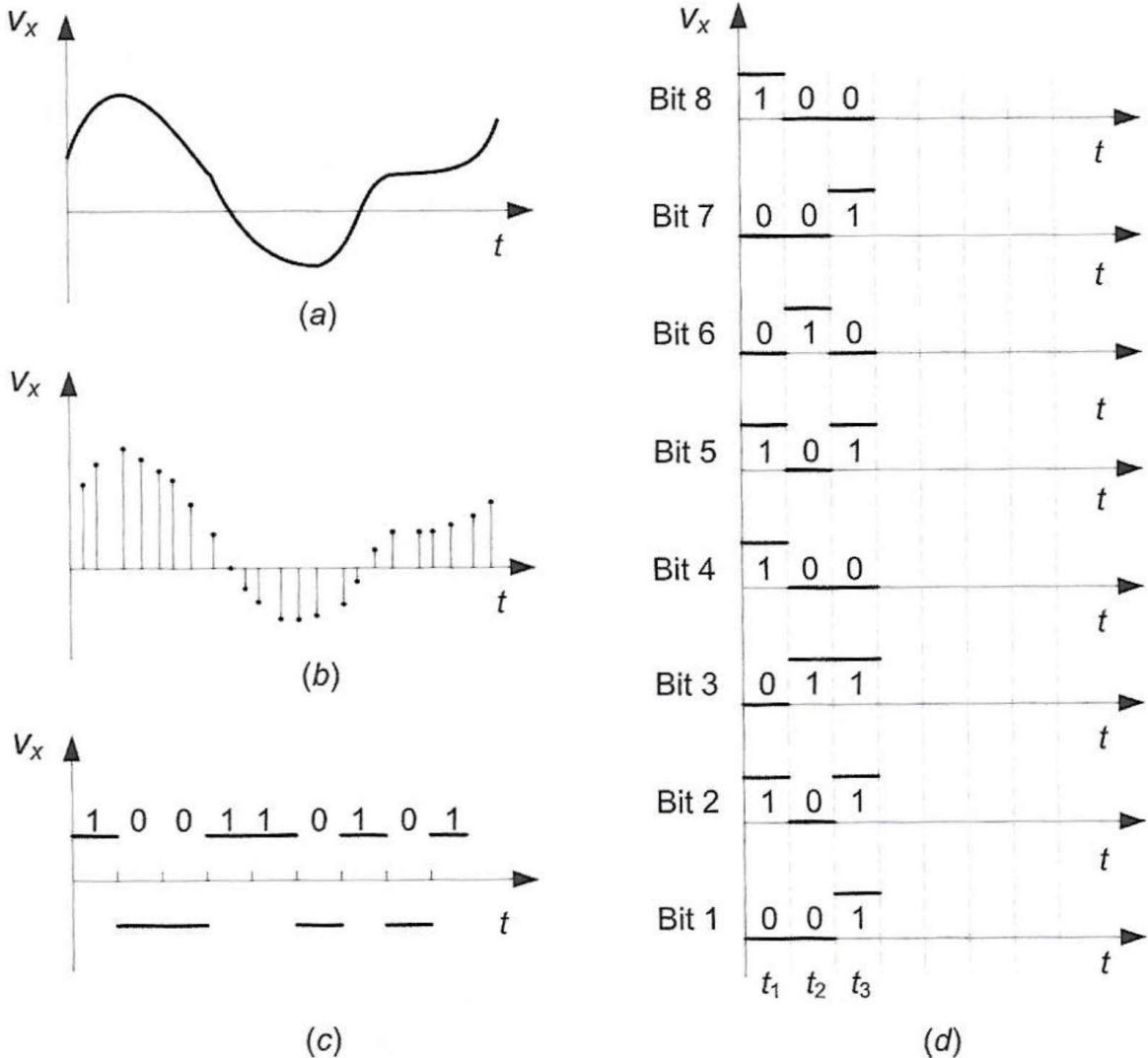


Figure 36. Señales (a) analógica continua (b)analógica discontinua, (c) digital en formato serie, (d) digital en formato paralelo.

#### Conversión A/D (analógica/digital)

Generalmente, el procesamiento se realiza en formato digital. Un convertidor analógico-digital (A/D, ADC o CAD) obtiene una señal digital a partir de la amplitud de una señal analógica, generalmente una tensión. La señal de entrada al ADC se puede aplicar entre el terminal de entrada y masa (o punto de referencia de potencial respecto al que se miden las tensiones), en cuyo caso se dice que el CAD dispone de entrada asimétrica o "single-ended" input. Otra posibilidad es que la señal de tensión se conecte a dos entradas del ADC ninguno de los cuales es el de masa, en cuyo caso se dice que el ADC dispone de entrada diferencial. En ambos tipos de circuitos la entrada se dice flotante cuando el terminal de masa no está conectado a tierra. La entrada asimétrica flotante también se denomina seudo-diferencial.

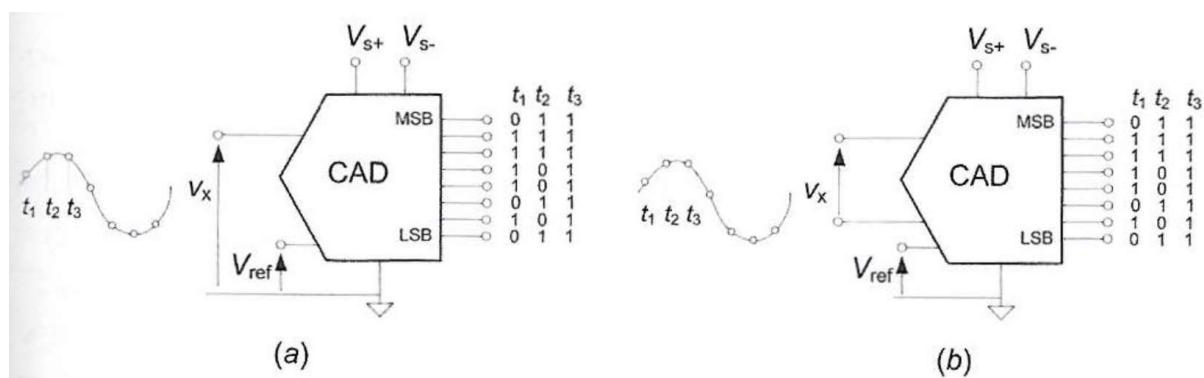


Figure 37. Entrada del convertidor (a) asimétrica y (b) diferencial



En los circuitos electrónicos hay tres puntos que por razones de seguridad o para reducir interferencias pueden estar separados o conectados entre sí, según interese: (1) el punto equipotencial, que es el punto respecto al que se miden las tensiones y sumidero para la corriente, (2) masa, que alude al chasis o envolvente metálica y (3) la tierra. La masa está conectada generalmente al punto equipotencial o a tierra o a ambos.

Los valores máximo y mínimo aceptables a la entrada del convertidor dependen de la tensión de alimentación  $V_{S+}$  y  $V_{S-}$  y de la tecnología del convertidor. Ninguna de las tensiones de entrada debe ser mayor que  $V_{S+}$  o menor que  $V_{S-}$  medidas respecto de masa. Digitalizar equivale a comparar la señal de entrada  $V_x$  que se va a convertir con una serie finita de niveles de tensión fijos cada uno de los cuales tiene asignado un conjunto de bits o código digital específico. Los niveles finitos de comparación se obtienen a partir de una tensión continua de referencia,  $V_{ref}$ , que debe ser muy estable en el tiempo, frente a cambios de temperatura y de la tensión de alimentación. Si se dispone de  $N$  símbolos binarios se puede formar  $2^N$  códigos diferentes (definen  $2^N-1$  niveles de comparación y  $2^N$  intervalos de tensión) y se dice que el convertidor tiene una resolución de  $N$  bits. Si tenemos en cuenta  $V_{ref}$ , podemos determinar la amplitud de cada intervalo de tensión del convertidor (resolución de tensión). El dígito binario (bit) de menor peso en el código digital se denomina bit menos significativo (LSB) y el de mayor peso, bit más significativo (MSB). El código digital asociado a una tensión en particular es el correspondiente al intervalo de tensión definido por los niveles de tensión inmediatamente inferior y superior a dicha tensión, es decir, el código asociado al intervalo que contiene a dicha tensión. En instrumentación, es habitual que los niveles de tensión sean equidistantes, lo que determina que los intervalos de tensión sean de ancho uniforme. Este ancho se denomina **intervalo de cuantificación**  $Q$  o resolución de tensión del convertidor.

El intervalo de cuantificación se expresa como:

$$Q = V_{ref}/2^N$$

donde  $Q = 1$  LSB.



El ancho de un intervalo de tensión se define como 1 LSB y se usa habitualmente como unidad de referencia para especificar otras características del conversor.

Para asignar los códigos digitales a los intervalos de cuantificación un método consiste en asignar el primer código al primer intervalo de cuantificación, es decir, al situado entre los niveles de tensión 0 V y  $Q$ ; el segundo código al segundo intervalo, situado entre niveles  $Q$  y  $2Q$  y así sucesivamente. Con este método si  $V_{min}=0$  V a una tensión  $V_x$  se le asigna el código numérico que se muestra en la figura.

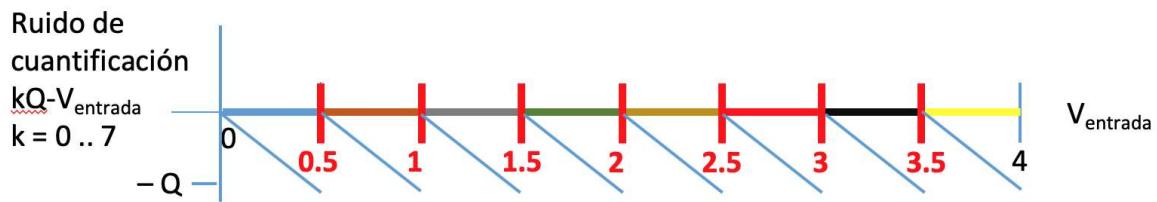
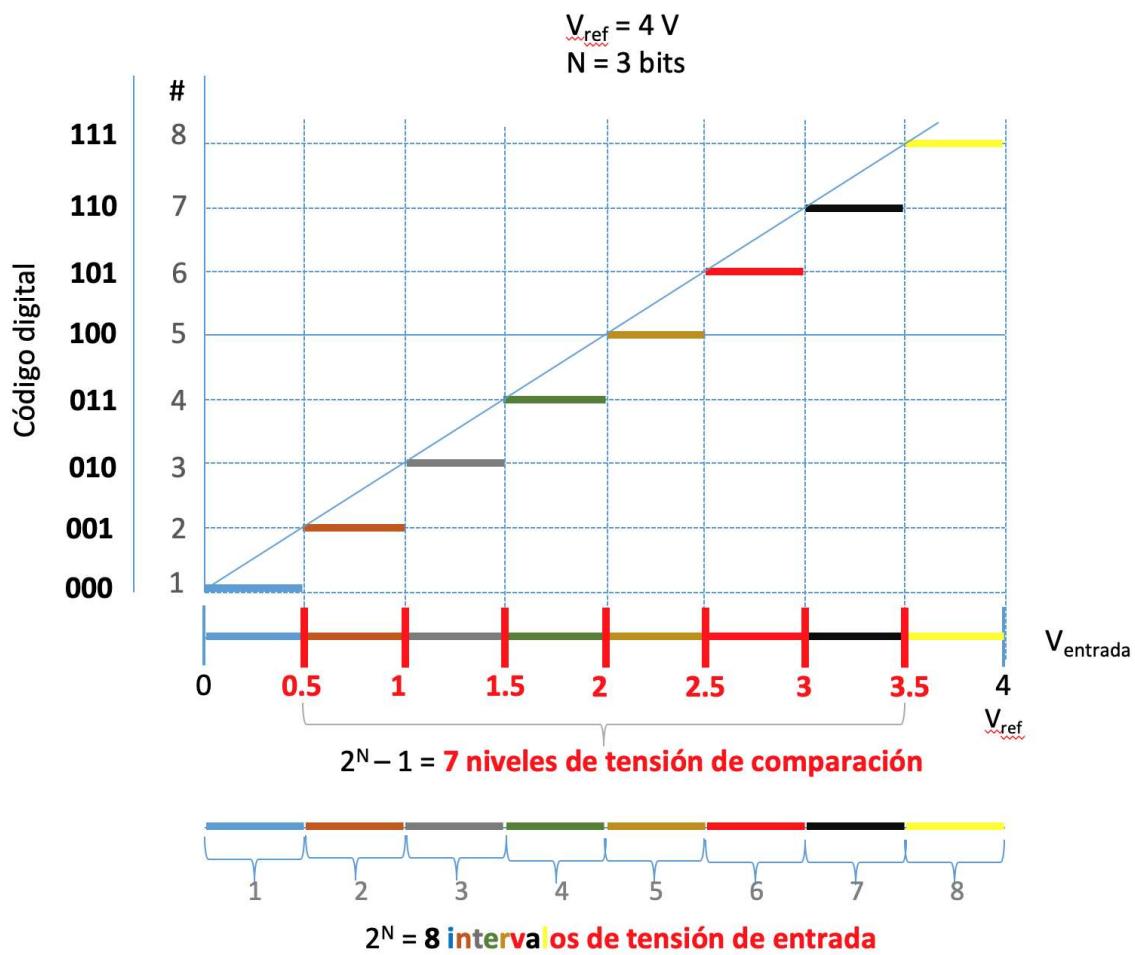


Figure 38. Cuantificación y ruido de cuantificación

Por ejemplo, supongamos  $N=3$  y  $V_{ref}= 4 \text{ V}$ . Dispondremos de  $8 (2^N)$  códigos diferentes: 000, 001, ..., 111; 7 niveles de comparación ( $2^N - 1$ ) ó niveles de tensión distribuidos uniformemente entre 0V y 4V ( $V_{ref}$ ) y 8 ( $2^N$ ) intervalos de tensiones tal como se muestra en la tabla adjunta.

Table 2. Intervalos de entrada, niveles y códigos binarios

#	1	2	3	4	5	6	7	8
Códigos	000	001	010	011	100	101	110	111
Intervalos de tensión	[0 V-0.5V]	[0.5 V-1 V)	[1 V- 1.5 V)	[1.5 V-2 V)	[2 V-2.5 V)	[2.5 V- 3 V)	[3 V-3.5 V)	[3.5 V- 4 V]
Niveles de comparación	0.5 V	1 V	1.5 V	2 V	2.5 V	3 V	3.5 V	

En condiciones ideales, con infinitos códigos, cada entrada debería estar representada por un código distinto y los escalones de la figura se sustituirían por una recta continua o curva ideal del convertidor, ver figura.



La 1<sup>a</sup> fila de la tabla enumera las distintas respuestas del A/D y puede sustituirse por la combinación binaria o código binario deseado, habitualmente se utiliza el binario natural. Por ejemplo, al código número 1 se le asocia la combinación 000; al código número 2 el 001, ... y al 8 el 111.

Dado que a todas las tensiones de entrada cuyo valor esté situado entre dos niveles contiguos de tensión -dentro del mismo intervalo- se le asigna el mismo código digital, hay una discrepancia entre la significación de un código de salida y la tensión real de entrada, salvo en los niveles de transición. Esta discrepancia, consecuencia del proceso de cuantificación, se puede asociar a un ruido de cuantificación superpuesto a la entrada y cuya amplitud máxima es Q.

Otra forma de asignar códigos digitales a los intervalos de cuantificación es asociar el primer código al intervalo situado entre los niveles de tensión 0 V y Q/2, el segundo a las tensiones entre Q/2 y 3Q/2 ( $Q/2+Q=3Q/2$ ), el tercero a las tensiones entre 3Q/2 y 5Q/2; el cuarto entre 5Q/2 y 7Q/2, el quinto entre 7Q/2 y 9Q/2, sexto 9Q/2 y 11Q/2, el séptimo entre 11Q/2 y 13Q/2 y el octavo entre 13Q/2 y Vref. El primer intervalo tiene una anchura de Q/2, el último 3Q/2 y el resto Q. La tabla siguiente refleja esta asignación.

*Table 3. Intervalos de entrada, niveles y códigos binarios*

#	1	2	3	4	5	6	7	8
Códigos	000	001	010	011	100	101	110	111
Intervalos de tensión	[0 V- 0.25V)	[0.25 V- 0.75 V)	[0.75 V- 1.25)	[1.25 V- 1.75 V)	[1.55 V- 2.25 V)	[2.25 V- 2.75 V)	[2.75 V- 3.25 V)	[3.25 V- 4 V]
Niveles de comparación	0.25 V	0.75 V	1.25 V	1.75 V	2.25 V	2.75 V	3.25 V	

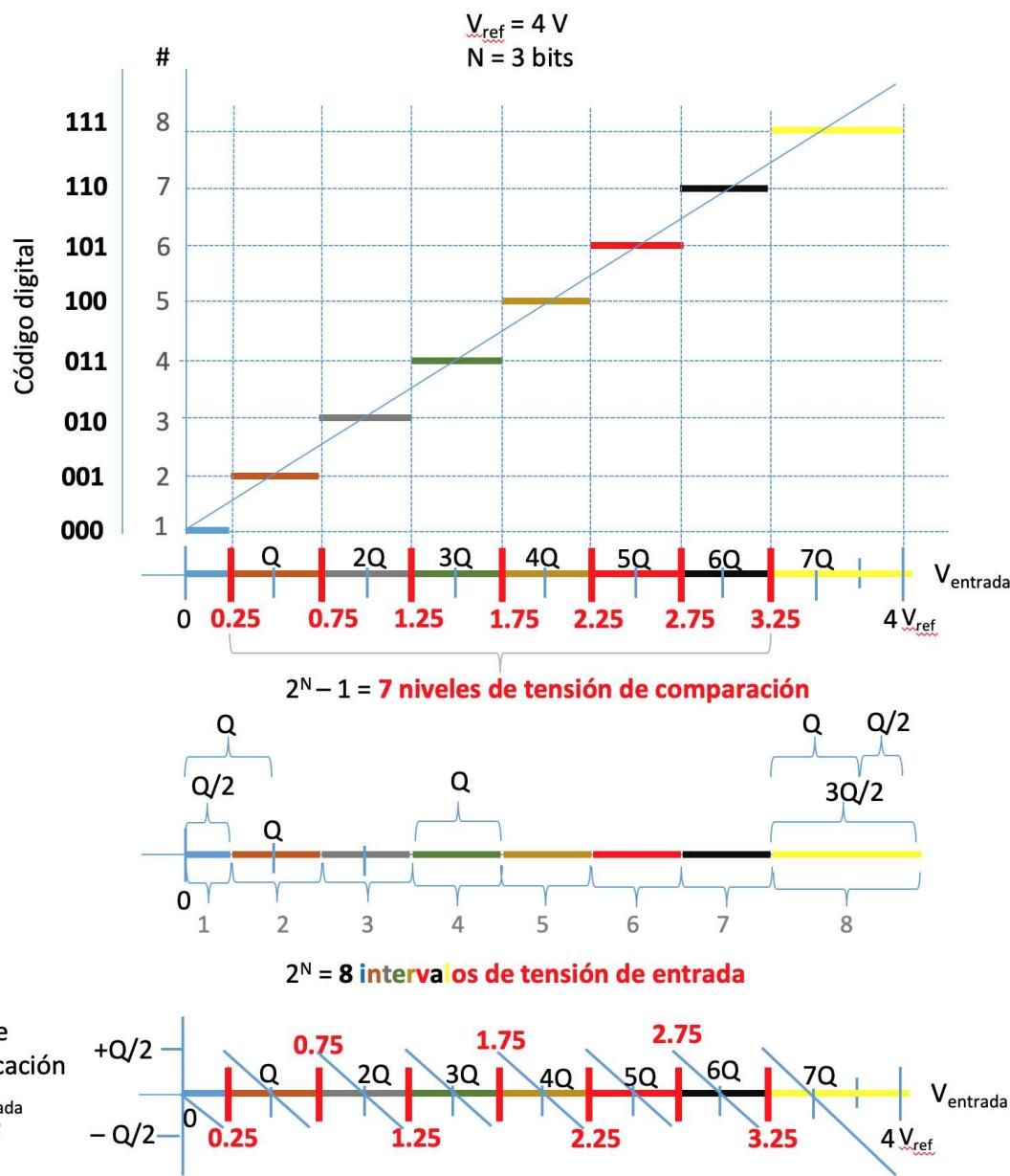


Figure 39. Cuantificación y ruido de cuantificación

Con esta segunda forma de asignación el error de cuantificación es de  $\pm Q/2$ . A partir de un código de salida, por tanto, solo se puede conocer la entrada con una incertidumbre  $\pm Q/2$ , de nuevo hay una discrepancia entre la señal de entrada y la significación del código asociado salvo, en este caso, en la mitad del intervalo. Por ejemplo, el código 100 lo puede haber producido cualquier tensión entre 1.75 V y 2.25 V. Esta indeterminación disminuye si aumenta el número de bits.

Al conjunto de valores de tensión admisibles a la entrada del convertidor se denomina **campo o rango** de entrada del conversor. La diferencia entre el valor máximo y mínimo admitido a la entrada del convertidor ( $V_{\text{ref}}$  y 0 V habitualmente), se denomina **alcance**. El rango de entrada del conversor debe ser compatible con la señal analógica a convertir si se pretende aprovechar toda su resolución, es decir, si se pretende utilizar todos los códigos disponibles del conversor. Por ejemplo, si la señal analógica a convertir es compatible con la entrada del convertidor pero tiene un alcance menor algunos códigos del convertidor no se utilizarán jamás. En este caso, la resolución real sería inferior a la que podríamos conseguir si hubiésemos adaptado la señal al rango de entrada del convertidor.

Problema 1. Se desea digitalizar una señal que varía entre 2 V y 3 V con un convertidor de  $N=10$  bits de resolución con un margen de entrada entre 0 V y 5 V. ¿Cuál es la resolución real del convertidor?

La resolución de tensión del convertidor la expresamos como:

$$r = 5 \text{ V} / 2^N = 4,8 \text{ mV}$$

El número de escalones o códigos para el intervalo de entrada de la señal a digitalizar será:

$$\#-\text{Códigos} = 3 \text{ V} - 2 \text{ V} / 4,88 \text{ mV} = 209 \text{ códigos}$$

$$(\log_2 209=7,7)$$

la resolución real sería inferior a 8 bits (256 códigos), habríamos perdido más de 2 bits de resolución por no adaptar la señal de entrada al margen 0 V-5 V del convertidor. Una posible solución puede ser restar 2 V a la entrada para obtener una señal entre 0 V y 1 V y después amplificar por 5.

### *Conversión Digital a Analógica*

Un convertidor digital-analógico (D/A, DAC en inglés o CDA) entrega una salida analógica (tensión o corriente) equivalente a un código digital introducido por su entrada. La regla de equivalencia depende de la ponderación que tenga cada bit del código digital. Para el código binario natural, el peso del MSB es 1/2 y el peso del resto de bits es siempre la mitad del peso del bit anterior. Si el código de entrada tiene N bits y la salida es una tensión, su valor será:

$$V_o = (\frac{B_N}{2} + \frac{B_{N-1}}{2^2} + \frac{B_{N-2}}{2^3} + \dots + \frac{B_2}{2^{N-1}} + \frac{B_1}{2^N})xV_{ref}$$

donde  $B_i$  representa cada uno de los dígitos de la combinación binaria de entrada. Como el DAC representa un conjunto discreto de códigos digitales de entrada por un número discreto de tensiones analógicas a su salida, su función de transferencia es una serie discreta de puntos, tal como muestra la figura.

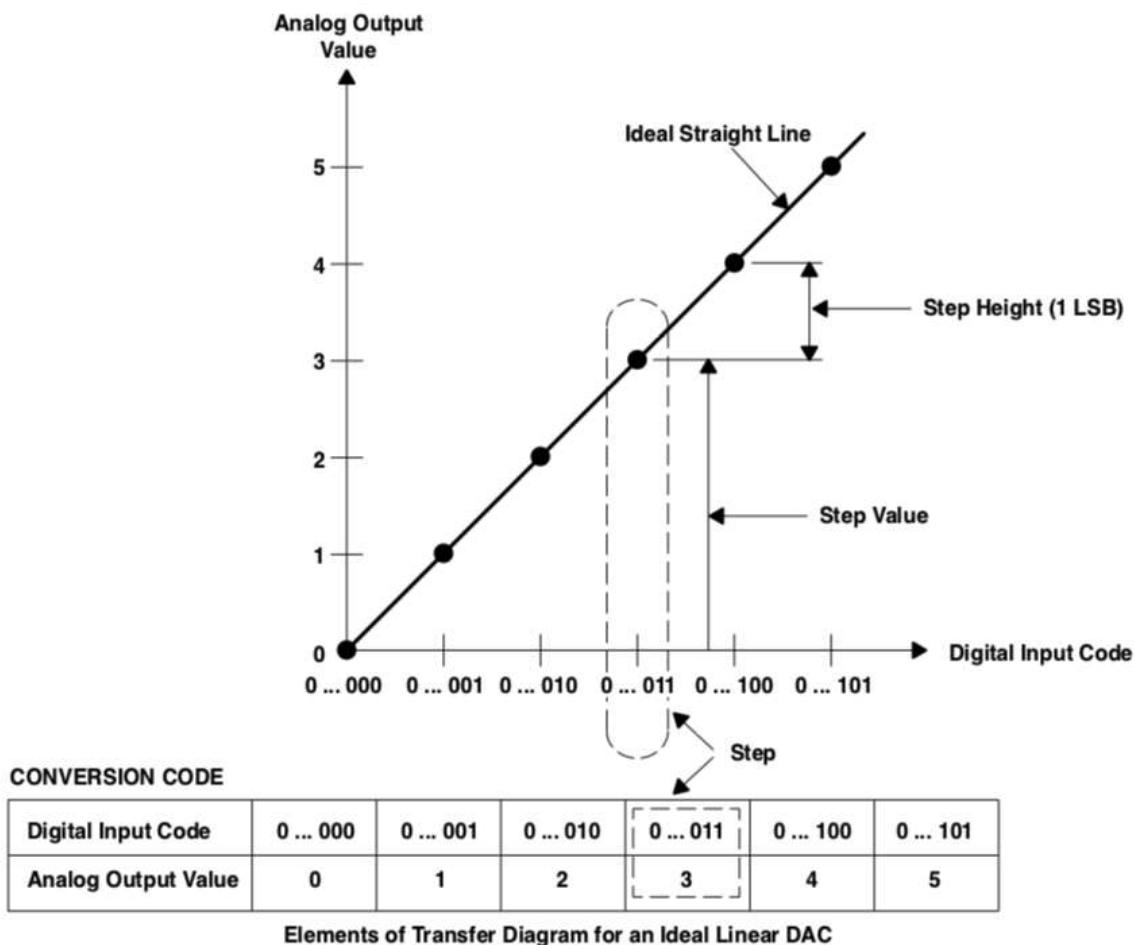


Figure 40. Función de transferencia ideal de un DAC

En un DAC, 1 LSB se corresponde con el intervalo entre dos niveles consecutivos de tensión analógica de salida. Existe también una cuantificación (discretización) pues a la salida solo se pueden obtener  $2^N$  niveles de tensión distintos.

### Tipos de convertidores A/D

Un simple comparador se comporta como un ADC de 1 bit. Cuando la entrada es superior al umbral de comparación, la salida es uno lógico, si es inferior, es un cero lógico. En general, todos los ADC realizan alguna comparación para realizar su función. La forma más sencilla de implementar un comparador es con un amplificador operacional. Puede producirse una pequeña oscilación cuando la diferencia entre las señales de entrada (entrada diferencial) se aproxima a cero, por ello, es habitual dotar al comparador de una pequeña histéresis (siempre inferior a la resolución del comparador). La histéresis se consigue dotando al circuito de una pequeña cantidad de realimentando positiva tal como estudió en el tema dedicado a los amplificadores operacionales.

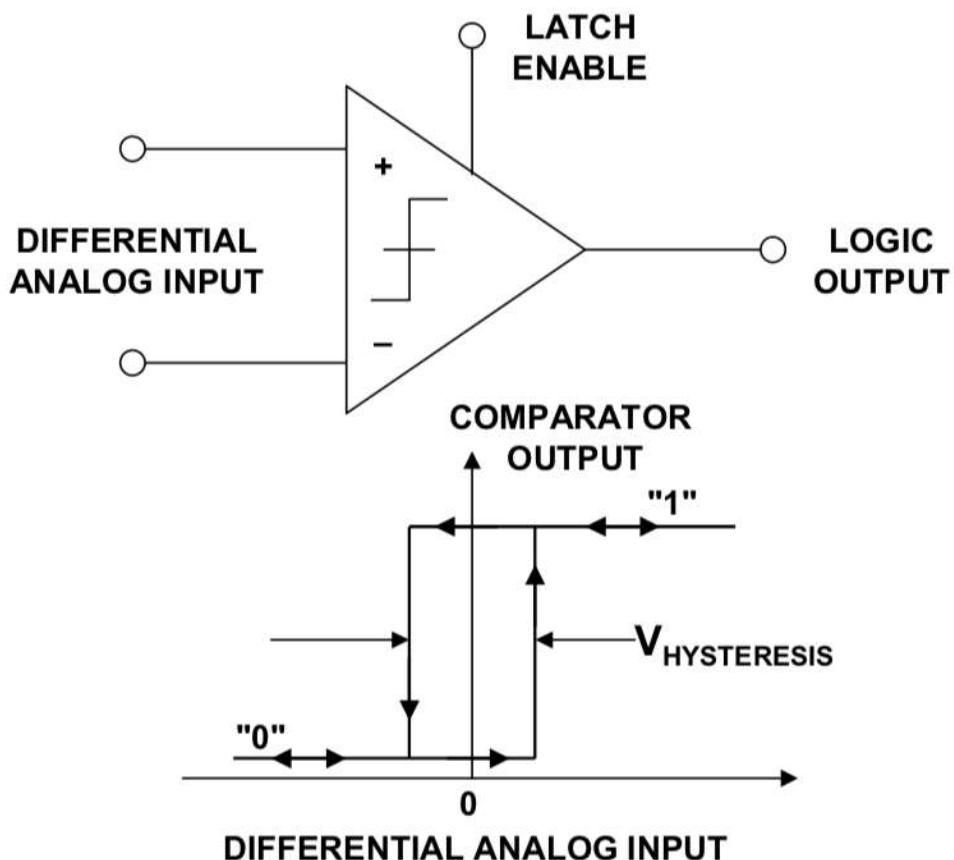
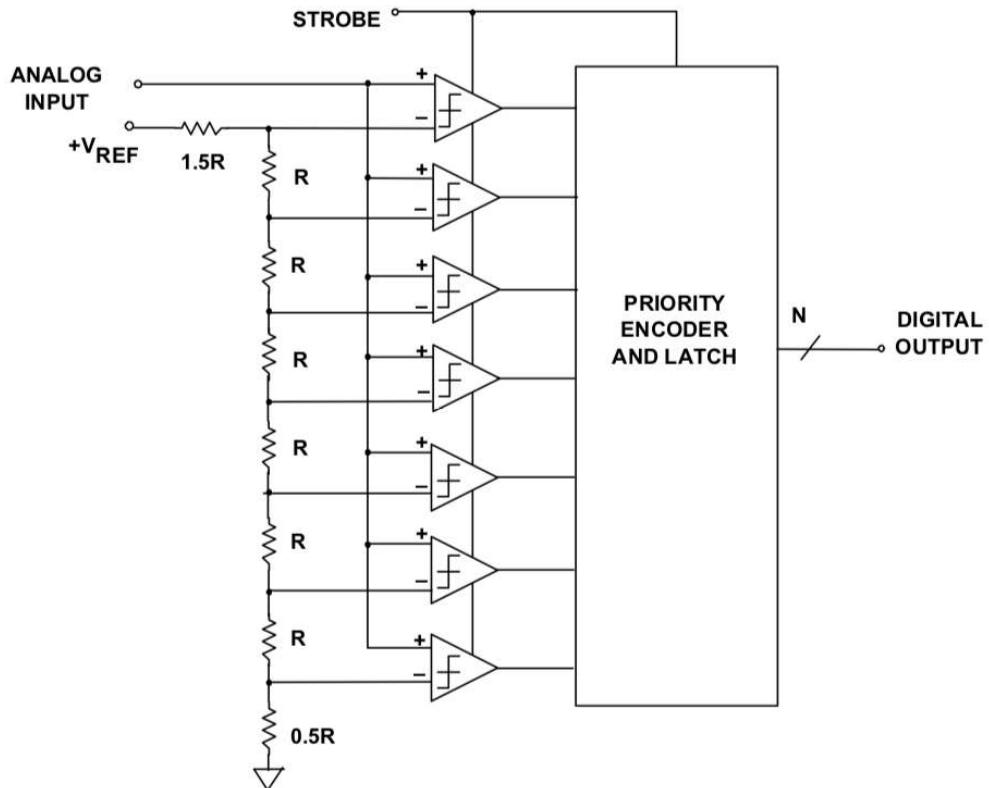


Figure 41. Convertidor Flash o paralelo.

Ampliar este ADC de 1 bit requiere más comparadores y es el fundamento de los convertidores **flash** o convertidores paralelo. Un convertidor flash de  $N$  bits requiere de  $2^N$  resistencias y de  $2^N - 1$  comparadores dispuestos según muestra la figura.



*Figure 42. Convertidor Flash o paralelo.*

Cada comparador tiene una referencia de tensión, proporcionada a través de la cadena de resistencias, que es un LSB superior del nivel inmediatamente inferior. La combinación de salida de los comparadores dependerá de si la entrada es superior o no a cada nivel de referencia. El resultado de la comparación se conecta a un codificador con prioridad cuya salida es el código digital resultado de la conversión. Este tipo de convertidores son los más rápidos en realizar la conversión, aunque tienen limitaciones: requieren un gran número de comparadores y resistencias, presentan un alto consumo si trabajan a alta velocidad de conversión y el tamaño del chip es considerable.

#### *Convertidor de aproximaciones sucesivas*

Uno de los convertidores más habituales, dadas sus prestaciones, es el **convertidor de aproximaciones sucesivas**. El algoritmo de aproximaciones sucesivas es similar al proceso que se sigue cuando se pesa un objeto con una báscula de brazos. Para calcular el peso (con la resolución de la pesa más pequeña) se van poniendo o quitando pesas de uno de los brazos según se incline la balanza. Cada pesa tiene un peso que es la mitad de la anterior; es decir, a partir de una referencia, las pesas son de  $1/2, 1/4, 1/8 \dots$  del peso de la referencia. Para estimar el peso de un objeto se empieza situando la pesa de mayor peso sobre uno de los platos ( $1/2$ ). Si el peso del objeto es menor que dicha pesa, se retira del plato y se pone la siguiente pesa de mayor peso. Si el peso del objeto fuese mayor que la pesa, entonces se mantiene en el plato y se sitúa la siguiente de mayor peso. Así se procede hasta situar o retirar la pesa más pequeña. Sumando las pesas que permanecen en el plato se obtiene la respuesta.

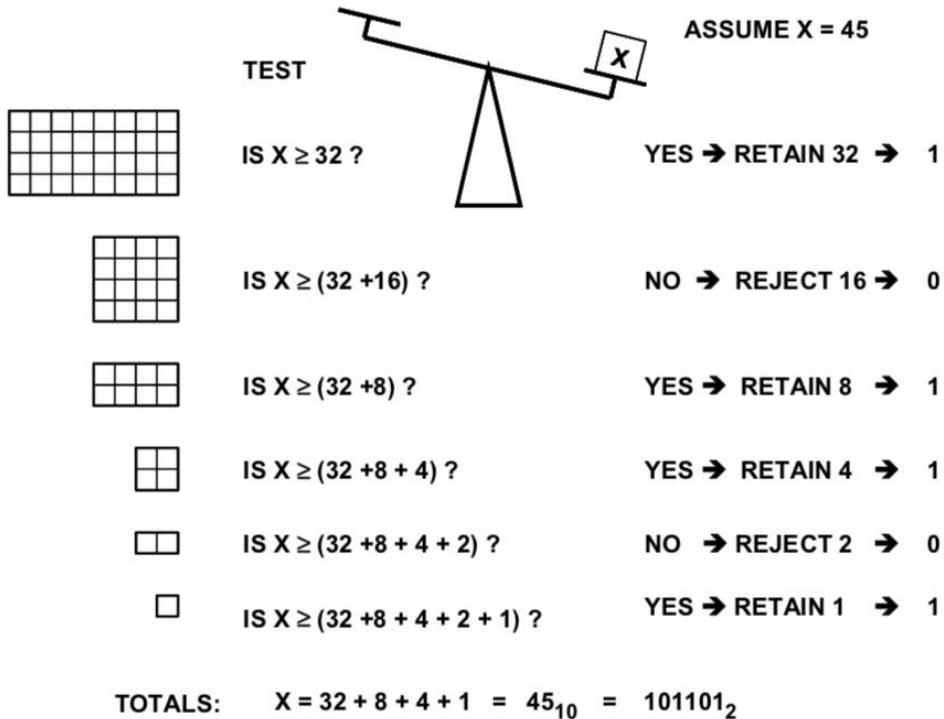


Figure 43. Algoritmo de aproximaciones sucesivas: peso de un objeto.

Desde una perspectiva electrónica, este tipo de convertidor se basa en comparar sucesivamente una señal analógica de entrada con fracciones cada vez menores de la referencia de tensión del convertidor. El proceso sigue en una secuencia de pasos sincronizados por una señal de reloj. Según sea el resultado de la comparación, el bit de salida se mantiene en uno si la tensión de entrada es mayor que el nivel de comparación y cero en caso contrario. Se realizan tantos pasos de comparación como bits de salida disponga el convertidor. Un elemento muy importante de este convertidor es que la tensión de referencia, a partir de la cual el DAC genera las fracciones de tensión para realizar la comparación, sea muy estable. Para ello, suelen utilizarse referencias de tensión específicas.

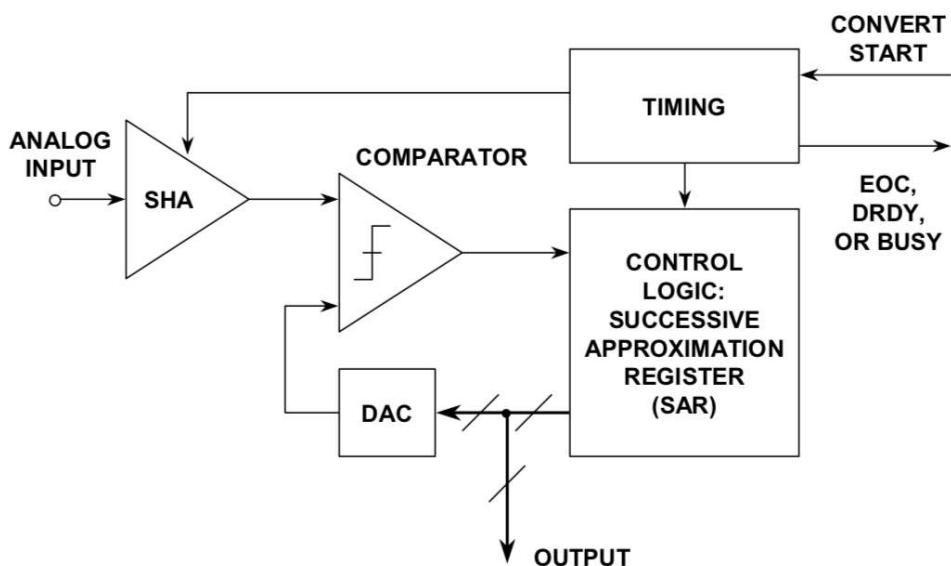


Figure 44. Diagrama de bloques de un ADC de aproximaciones sucesivas.

En la figura vemos que en una primera iteración la salida digital de la conversión se corresponde con la mitad de los códigos de conversión (1000...0). La tensión analógica correspondiente a este código generada por el DAC sería la mitad de la tensión de referencia (Vref). Si esta tensión es menor que la señal de entrada el bit correspondiente se mantiene a 1 lógico y se pasa al siguiente paso estableciendo ahora una nueva combinación binaria que pone a 1 lógico el siguiente bit de mayor peso (11000...0). Si la entrada fuese menor que la señal generada por el DAC para la combinación (1000...0)

entonces el bit de mayor peso se pone a cero y se pasa a la combinación (01000...0). Este proceso se repite con todos los bits hasta llegar al menos significativo. El tiempo de conversión es igual al numero de bits multiplicado por el periodo de reloj que marca la secuencia de comparaciones a realizar.

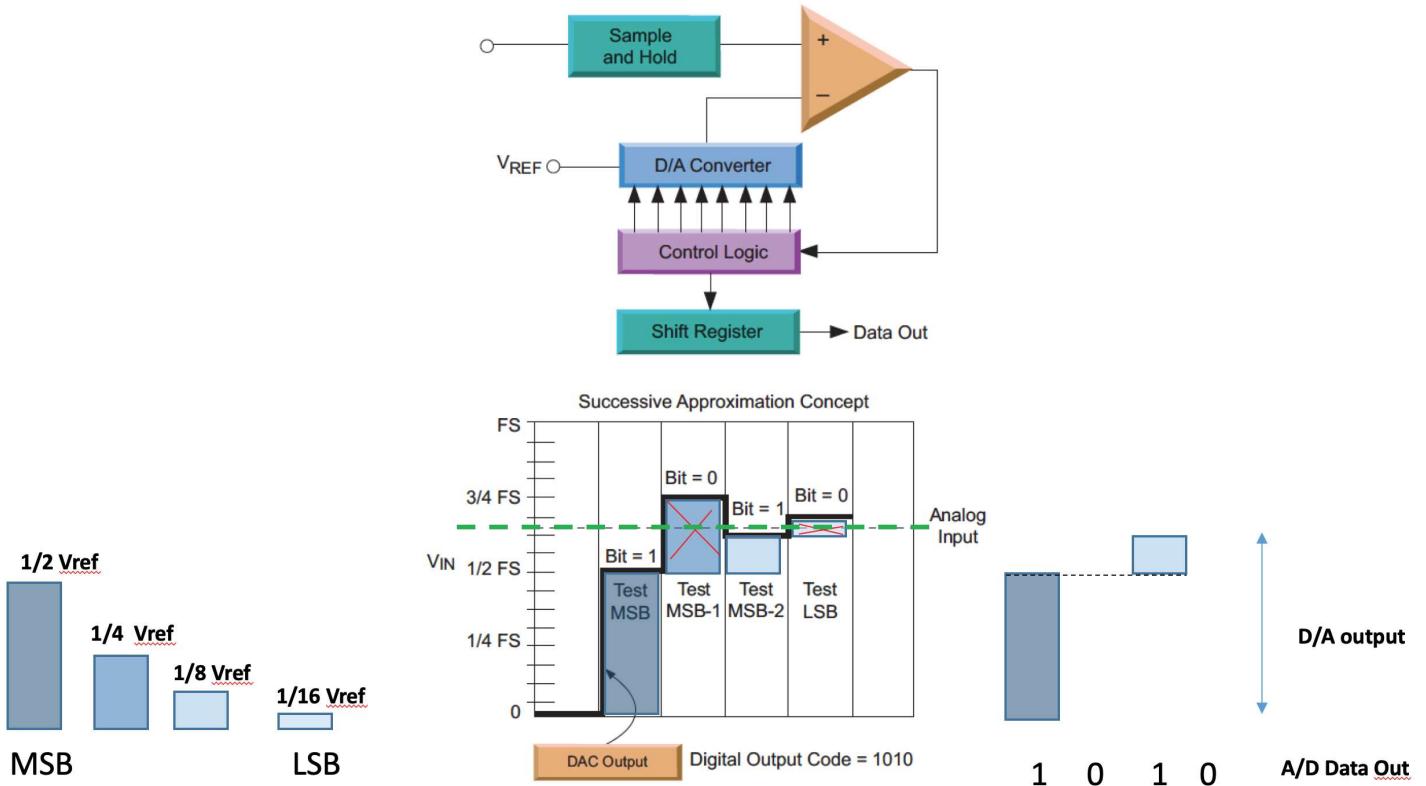


Figure 45. Funcionamiento del A/D de aproximaciones sucesivas.

También es muy importante que la señal de entrada se mantenga estable durante el proceso de conversión, es decir, la entrada no debe cambiar hasta que finalice la conversión. Para ello, se precede la entrada del convertidor A/D con un circuito de **muestreo y retención** o **Sample & Hold** (S&H). Básicamente carga un condensador con el valor instantáneo de la tensión de entrada a convertir (Sample) y después desconecta el condensador de la entrada manteniendo (Hold) cargado el condensador durante el **tiempo de conversión**. El tiempo que el interruptor permanece cerrado para que se cargue el condensador desde que se da la orden de comienzo de conversión se denomina **tiempo de adquisición**.

Version v1.0

Last updated 2023-10-04 11:30:25 +0200