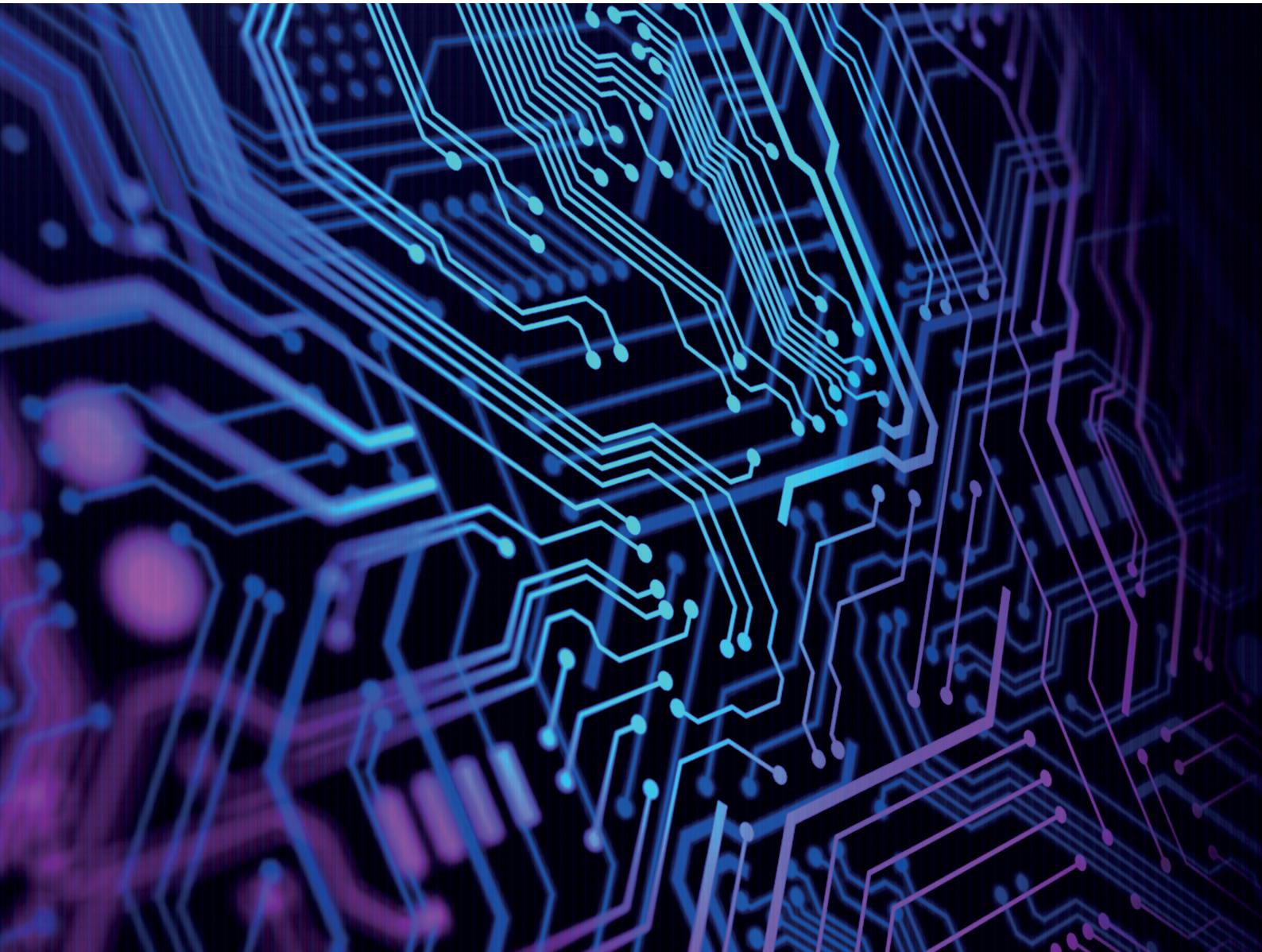


# ATmega2560



Teoría y aplicaciones  
usando Atmel Studio

David Fernando Pozo Espín  
Jorge Luis Rosero Beltrán  
Santiago Leonardo Solórzano Lescano



Estimado/a lector/a:

DESCARGAR



En este enlace (<https://www.udlaediciones.com.ec/wp-content/uploads/2022/05/CODIGO-MODIFICABLE-DEL-LIBRO.docx>) se encuentra un archivo con el que podrán descargar todos los códigos utilizados en esta publicación, para su uso estudiantil pertinente.

Algunos gráficos en este libro podrían parecer muy pequeños para leer sus detalles, por lo que hemos añadido un botón que los llevará a una versión expandida y completamente legible del mismo.

u/o.



# **ATmega2560:**

## Teoría y aplicaciones usando Atmel Studio



# **ATmega2560:**

## **Teoría y aplicaciones usando Atmel Studio**

### **Autores**

David Fernando Pozo Espín  
Jorge Luis Rosero Beltrán  
Santiago Leonardo Solórzano Lescano

**ATmega2560:**

**Teoría y aplicaciones usando Atmel Studio**

© David Fernando Pozo Espín

Jorge Luis Rosero Beltrán

Santiago Leonardo Solórzano Lescano

© Universidad de Las Américas

Facultad de Ingeniería y Ciencias Aplicadas FICA

Campus UDLA Park

Redondel del Ciclista

Vía a Nayón, s/n

[www.udla.edu.ec](http://www.udla.edu.ec)

Facebook: [@udlaQuito](#)

Quito, Ecuador

Tercera edición:

#### **EDICIÓN**

Susana Salvador Crespo

Coordinadora UDLA Ediciones

#### **CUIDADO DE LA EDICIÓN**

Fabricio Cerón Rivas

Analista editorial UDLA Ediciones

#### **CORRECCIÓN DE ESTILO**

Editorial El Conejo

#### **DISEÑO DE CUBIERTA**

Editorial El Conejo

#### **DISEÑO Y DIAGRAMACIÓN**

Editorial El Conejo

#### **EDITORIAL**

UDLA Ediciones

Gracias por respetar las leyes del copyright al no reproducir, escanear ni distribuir ninguna parte de esta obra, sin la debida autorización. Al hacerlo está respetando a los autores y permitiendo que la UDLA continúe con la difusión del conocimiento. Reservados todos los derechos. El contenido de este libro se encuentra protegido por la ley y es publicado bajo licencia exclusiva mundial.

Antes de su publicación, esta obra fue evaluada bajo la modalidad de revisión por pares anónimos.

**ISBN:** 978-9942-779-51-9

Impreso en Quito, Ecuador, 2022

# Contenidos

<b>Introducción</b>	<b>1</b>		
<b>Capítulo 1</b>			
<b>Lógica digital</b>	<b>5</b>		
1.1 Anotaciones preliminares sobre la lógica digital	5	2.3 Clasificación	25
1.2 Sistemas de numeración	5	2.4 Familias de microcontroladores	25
1.3 Transformación entre sistemas de numeración	6	2.5 Aplicaciones de microcontroladores	26
1.3.1 Transformación decimal a binario	6	2.6 Placas de desarrollo comercial	26
1.3.2 Transformación binario a decimal	7	2.6.1 Samsung ARTIK 710	26
1.4 Aritmética binaria	7	2.6.2 SensorTile	27
1.5 Suma binaria	7	2.6.3 Adalm-Pluto	27
1.6 Resta binaria	9	2.6.4 AudioSmart 2-Mic	28
1.6.1 Complemento a1	9	2.6.5 Thunderboard sense	28
1.6.2 Complemento a2	10	2.6.6 Flora AdaFruit	28
1.7 Números con signo	11	2.6.7 LaunchPad (MSP-EXP430G2ET)	29
1.8 Signo magnitud	11	2.6.8 PICAXE 18X	29
1.9 Signo usando complemento a1	11	2.6.9 Wiring	29
1.10 Signo usando complemento a2	11	2.6.10 Netduino	30
1.11 Compuertas lógicas	14	2.6.11 Arduino	30
1.11.1 Compuerta lógica NOT	14	2.6.12 Arduino mega	31
1.11.2 Compuerta lógica AND	14	2.6.12.1 Alimentación	32
1.11.3 Compuerta lógica OR	14	2.6.12.2 Memoria	32
1.11.4 Compuerta lógica NAND	14	2.6.12.3 Pines E/S	32
1.11.5 Compuerta lógica NOR	14	2.6.12.4 Comunicaciones	34
1.11.6 Compuerta lógica XOR	14	2.7 Software AVR ATMEL Studio	34
1.12 Códigos binarios	17	2.7.1 Mi primer programa	35
1.12.1 BCD (Binary Coded Decimal)	17		
1.12.2 Gray	18		
1.12.3 ASCII (American Standard Code for Information Interchange)	18	<b>Capítulo 3</b>	
1.12.4 Representación de punto fijo	20	<b>Puertos de entrada-salida</b>	<b>39</b>
<b>Capítulo 2</b>			
<b>Introducción a los microcontroladores</b>	<b>23</b>		
2.1 Anotaciones preliminares sobre microcontroladores	23	3.1 Anotaciones preliminares sobre manejo de puertos	39
2.2 Estructura del microcontrolador	24	3.2 Registros de entrada-salida	40
		3.2.1 Ejercicios prácticos de configuración de puertos	42
		3.3 Enmascaramiento y operaciones bit a bit	44
		3.3.1 Ejercicios prácticos de enmascaramiento	44
		3.4 Operaciones abreviadas	47
		3.4.1 Ejercicios prácticos de configuración E/S	47
		<b>Capítulo 4</b>	
		<b>Interrupciones</b>	<b>55</b>

4.1 Anotaciones preliminares sobre interrupciones	55	<b>Glosario de términos</b>	125
4.2 Interrupciones externas	57	<b>Autores</b>	127
4.2.1 Interrupciones tipo INT	58	<b>Índice de figuras</b>	129
4.2.2 Interrupciones tipo PCINT	59	<b>Índice de tablas</b>	134
4.3 Ejercicios prácticos de configuración de interrupciones externas	61	<b>Índice de imágenes</b>	135
 		<b>Bibliografía</b>	136
<b>Capítulo 5</b>			
<b>Comunicación serial</b>	<b>73</b>		
5.1 Anotaciones preliminares sobre comunicación serial	73		
5.2 Modos de transmisión serial	73		
5.2.1 Transferencia asíncrona de datos	74		
5.2.2 Transferencia síncrona de datos	74		
5.2.3 Parámetros de la comunicación serial	74		
5.2.4 Protocolos de comunicación en serie	75		
5.2.5 Tipos de conexiones o arreglos del puerto serial	76		
5.3 Comunicación serial ATmega2560	77		
5.3.1 Configuración de USART	77		
5.3.2 Registro de velocidad en baudios UBRRn	78		
5.3.3 Registro de control y estado UCSRnA	78		
5.3.4 Registro de control y estado UCSRnB	79		
5.3.5 Registro de control y estado UCSRnC	80		
5.3.6 Registro de datos UDRn	81		
5.4 Ejercicios prácticos de configuración	81		
<b>Capítulo 6</b>			
<b>Temporizadores y contadores</b>	<b>91</b>		
6.1 Anotaciones preliminares temporizadores y contadores	91		
6.2 Contadores de 8 y 16 bits	91		
6.3 Modos de operación	93		
6.3.1 Modo normal	94		
6.3.2 Modo de comparación CTC	97		
6.3.3 Modo Fast PWM	100		
6.3.4 Modo Phase Correct PWM	103		
6.3.5 Modo Phase and Frequency Correct PWM	104		
6.3.6 Unidad de captura	104		
<b>Capítulo 7</b>			
<b>Conversión analógica digital</b>	<b>107</b>		
7.1 Anotaciones preliminares sobre conversión analógica digital	107		
7.2 Canales, voltaje de referencia y resultado	107		
7.3 Tiempos de conversión y señal de reloj	111		
7.4 Inicio y modos de conversión	112		
7.5 Guía para configurar el módulo ADC	113		
7.6 Ejercicios prácticos de configuración del conversor analógico digital	114		

# Introducción

Este trabajo pretende que los lectores adquieran nuevos conocimientos a través de una herramienta de consulta, con contenidos científico-técnicos actualizados, didácticos y prácticos, que les permitan desarrollar, crear e innovar aplicaciones tecnológicas basadas en microcontroladores, y comprender las potencialidades y limitaciones que estos poseen. La obra está dirigida a profesionales y a la comunidad académica de las distintas áreas de la ingeniería que deseen conocer, de mejor manera, el funcionamiento y la configuración de los distintos módulos que pueden encontrarse en un microcontrolador. Además, puede servir como material de apoyo para los lectores entusiastas que busquen desarrollar prototipos aplicados a solucionar problemas cotidianos, por medio de microcontroladores. El libro no está dirigido para el aprendizaje de lógica de programación y requiere conocimientos en sintaxis y programación básica basada en lenguaje C por parte de los lectores.<sup>1</sup>

A partir del desarrollo de dispositivos semiconductores, de la posterior aparición de circuitos integrados, y de la búsqueda constante por integrar y potencializar en un solo dispositivo tareas de cálculo matemático, procesamiento de información e interacción física con su entorno, apareció lo que hoy se conoce como microcontrolador. En la actualidad, la gran mayoría de aparatos de uso cotidiano contienen, aunque no se note, un microcontrolador, ya sea en dispositivos de alta tecnología o en algo tan común como un control remoto de un tele-

visor. No obstante, en el campo de la ingeniería estos dispositivos cobran una relevancia significativa, ya que con ellos se pueden desarrollar aplicaciones en áreas como la robótica, domótica, internet de las cosas, ciudades y hogares inteligentes, dispositivos médicos, agricultura de precisión, entretenimiento, comunicaciones, entre otros.

En este libro, en una primera parte, presentamos los conceptos fundamentales de la lógica digital, que son las bases para entender el principio de funcionamiento, la estructura interna y la operación de un microcontrolador. Con el objetivo de profundizar y estudiar con mayor detalle el funcionamiento de un microcontrolador, abordamos los conceptos de configuración y operación relacionados a los periféricos de uso común en el desarrollo de aplicaciones con microcontroladores de 8 bits AVR ATMEL. Tomamos, como ejemplo, el manejo de microcontrolador ATmega2560. Las temáticas inician estratégicamente con el manejo de puertos de entrada y salida, a fin de que el lector logre discernir cómo el *hardware* del microcontrolador interactúa con el mundo digital. Una de las características relevantes de un microcontrolador es su capacidad para procesar eventos inesperados que pueden ocurrir en sus periféricos, tanto interna como externamente. Por ello, estudiamos esta capacidad, de manera detallada, en el capítulo de interrupciones externas.

Más adelante, y considerando que el envío y la re-

<sup>1</sup> Si se desea iniciar con el aprendizaje de la sintaxis, las funciones, los operadores de control, las variables y demás elementos del lenguaje C, se puede encontrar esta información en otras fuentes (Mazidi,Naimi, & Sepehr Naimi, 2011)(Richard, Cox, & O'Cull, 2007).

cepción de información es una de las características básicas en sistemas microcontrolados, analizamos uno de los protocolos de comunicación más utilizados en la actualidad por fabricantes de dispositivos electrónicos: el Universal Asynchronous Receiver Transmitter (UART). También exploramos uno de los elementos más importantes en el microcontrolador, el oscilador (reloj o *clock*), que puede ser interno o externo, y del cual hace uso el procesador o unidad central de proceso (CPU) para realizar las distintas operaciones de cálculo y procesamiento de información. En el capítulo referente a temporizadores y contadores, examinamos cómo este reloj principal puede ser usado por este periférico, para generar señales de tipo PWM y bases de tiempo. Finalmente, ya que las variables físicas que se encuentran en la naturaleza son de origen analógico y que un microcontrolador necesita procesarlas a nivel digital, en el capítulo final, estudiamos dicha funcionalidad en el periférico de conversión analógico digital (ADC).

Con el fin de explicar de una manera didáctica las temáticas, y sin dejar de lado un análisis basado en sólidos conceptos teóricos, proponemos una gran variedad de ejemplos de aplicación práctica. El software Atmel Studio, utilizado en este libro, es intuitivo y sirve como base para desarrollar los ejercicios de aplicación, que están programados en Lenguaje C. Además, dichos ejemplos van acompañados de diagramas esquemáticos de conexión electrónica que facilitan al lector su implementación física y permiten verificar su funcionamiento. Asimismo, los conceptos y el análisis de configuraciones se fundamentan en el manual del microcontrolador.

En el capítulo uno, cuyo título es «Lógica digital», presentamos una panorámica general de los conceptos básicos referentes a los sistemas digitales. En ese sentido, revisamos los sistemas de numeración y su transformación entre sí; examinamos las compuertas lógicas y su principal funcionamiento; y, finalmente, mostramos las formas de representación binaria, tales como la codificación BCD, ASCII y Gray.

En el segundo capítulo, denominado «Introducción a los microcontroladores», planteamos una aproximación a la historia, la arquitectura, los fabricantes y las aplicaciones de estos chips que han revolucionado el mercado actual de la electrónica. Enfatizamos en el microcontrolador ATmega2560 e incluimos una explicación del software Atmel Studio para desarrollar soluciones de aplicación práctica.

En el tercer capítulo, correspondiente a «Puertos de entrada-salida», uno de los puntos esenciales es la interacción del microcontrolador con el medio externo. Aquí detallamos el comportamiento que pueden tener los pines<sup>2</sup> de un microcontrolador (entradas y salidas) y el tipo de información que puede ser manejada (analógica y digital), con un énfasis especial en una interacción de tipo digital.

En el capítulo 4, llamado «Interrupciones», señalamos que existen acciones que el microcontrolador debe ser capaz de procesar tan pronto como sea posible. Las interrupciones son un recurso que permite que el microcontrolador (MCU, por sus siglas en inglés de Microcontroller Unit) pueda ejecutar rutinas de servicio de interrupción (ISR) para eventos súbitos y realizar las operaciones que sean requeridas. En este capítulo, detallamos, además, los distintos eventos que pueden producir interrupciones de tipo externas y las configuraciones necesarias del microcontrolador para poder atenderlas.

Por otra parte, la comunicación es vital en la mayoría de los sistemas de microcontroladores. Esta comunicación existe desde el momento en que se conectan entre sí o a la PC para ser programados; o en comunicaciones más complejas e inalámbricas. En el capítulo 5, denominado «Comunicación serial», presentamos una introducción al modo de transmisión serial asíncrona (UART, por sus siglas en inglés Universal Asynchronous Receiver Transmitter), junto con sus parámetros, conexiones y los registros que se utilizan en el microcontrolador para su configuración.

En el capítulo 6, «Temporizadores y contadores»,

<sup>2</sup> Elemento de tipo físico que forma parte del microcontrolador y cuya función fundamental es permitir el intercambio de información (analógica o digital) entre el medio físico (externo al microcontrolador) y el microcontrolador.

revisamos de manera integral los diferentes modos de operación de los temporizadores y contadores de 8 y 16 bits. Explicamos los registros asociados, su funcionamiento y las aplicaciones más comunes. Finalmente, analizamos todos los modos de operación y los resultados medidos por medio de un osciloscopio de precisión.

Si bien en el capítulo 2 destacamos el comportamiento digital de los pines del microcontrolador, no es menos cierto que también podemos interactuar con información de tipo analógica. En el capítulo 7, denominado «Conversión analógica digital (ADC)», tratamos el comportamiento de los pines para adquirir señales analógicas y su conversión a valores digitales. Exploramos los distintos registros asociados y su configuración, mediante ejemplos que permitan aclarar las funcionalidades básicas del periférico de conversión analógica digital (ADC).



# Capítulo 1

## Lógica digital

### 1.1 Anotaciones preliminares sobre la lógica digital

Sin lugar a duda, la lógica digital es el punto de partida para estudiar la electrónica digital y los microcontroladores. Conceptos fundamentales, como la transformación entre sistemas de numeración, operaciones aritméticas a nivel binario y compuertas lógicas, son claves para el estudio de dispositivos más complejos y fascinantes, como los microcontroladores. En este capítulo estudiamos los sistemas de numeración binario, decimal y hexadecimal; examinamos la representación de números positivos y negativos, así como los conceptos de la aritmética binaria, suma y resta; además, revisamos las compuertas lógicas, su simbología y tablas de verdad; por último, exploramos los códigos binarios BCD, Gray y ASCII.

### 1.2 Sistemas de numeración

Un sistema de numeración es un conjunto de símbolos que, combinados entre sí bajo determinadas condiciones, permiten representar una cantidad numérica. Estos símbolos también se denominan dígitos de un sistema de numeración. La cantidad de dígitos que un sistema de numeración tiene define el valor de su base. Además, la posición en la que cada dígito se encuentra escrito determina, también, su ponderación. Es decir, un mismo dígito ubicado en la posición cero tendrá diferente ponderación si se encuentra en la posición 3. A esto se conoce como sistema posicional, puesto que un dígito colocado a la izquierda tendrá siempre mayor ponderación que uno colocado a la derecha. Esto

puede representarse matemáticamente a través de la ecuación 1.1 o, de una forma más compacta, en la ecuación 1.2.

$$X_b = \delta_k b^k + \delta_{k-1} b^{k-1} + \cdots + \delta_0 b^0 + \delta_{-1} b^{-1} + \cdots + \delta_{-n+1} b^{-n+1} + \delta_{-n} b^{-n} \quad (1.1)$$

$$X_b = \sum_{k=-n}^{-n} \delta_i b^i \quad (1.2)$$

Donde:

$b$  es la base del sistema de numeración,

$X_b$  es un número expresado en base  $b$ ,

$k$  es la posición más a la izquierda o más significativa,

$\delta_k$  es el dígito más a la izquierda o más significativo,

$n$  es la posición más a la derecha o menos significativa,

$\delta_{-n}$  es el dígito más a la derecha o menos significativo,

$b^k$  es la ponderación más significativa,

$b^{-n}$  es la ponderación menos significativa.

Estas expresiones pueden ser usadas para representar cualquier número de cualquier sistema de numeración. Por ejemplo, el número 3 847,95 en decimal luciría de esta manera:

$$3 \cdot 10^3 + 8 \cdot 10^2 + 4 \cdot 10^1 + 7 \cdot 10^0 + 9 \cdot 10^{-1} + 5 \cdot 10^{-2}$$

El sistema de numeración que los seres humanos utilizamos de manera natural es el decimal. Sin embargo, la teoría formulada por George Boole, en 1854, se basa en que todo lo que nos rodea puede ser presentado por dos estados. Esto dio origen a uno de los sistemas de numeración que revolucionaría el entendimiento humano y el desarrollo de sistemas computacionales que, hasta la actualidad, usan ese mismo principio. Este sistema de numeración se conoce como binario, y se caracteriza por tener únicamente dos dígitos (0 y 1). A su vez, la base de este sistema de numeración es dos. Usando la ecuación 1.1 o 1.2, el número binario 1101,101 podría ser representado de la siguiente forma:

$$1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3}$$

No obstante, el sistema de numeración binario puede ser representado por un sistema más compacto y práctico para los seres humanos: el sistema hexadecimal. Este permite representar cuatro dígitos binarios con un solo dígito hexadecimal. Los dígitos del sistema hexadecimal son los mismos del sistema decimal más las letras A, B, C, D, E y F, hasta completar un total de 16, en el cual este es, también, el valor de su base. Del mismo modo, un número expresado en este sistema puede ser representado usando la ecuación 1.1 o 1.2. Por ejemplo, el número A3E4,6CD se representa así:

$$A \cdot 16^3 + 3 \cdot 16^2 + E \cdot 16^1 + 4 \cdot 16^0 + 6 \cdot 16^{-1} + C \cdot 16^{-2} + D \cdot 16^{-3}$$

Además de conocer la forma en la que un sistema de numeración puede ser representado, es importante conocer cómo poder transformar números de un sistema a otro. En el siguiente apartado, compartimos algunos ejemplos.

### 1.3 Transformación entre sistemas de numeración

Para transformar números desde cualquier base a otra, recomendamos, previamente, pasar al sistema binario. Esto, debido a la practicidad y sencillez de este sistema. Para ello, primero mostramos la transformación de números desde el sistema decimal a binario, y viceversa.

#### 1.3.1 Transformación decimal a binario

Probablemente, esta transformación es una de las más usadas y estudiadas en un curso de ingeniería. El método utilizado se denomina “transformación por ponderaciones”.

**Ejercicio 1.1:** Transformar el número decimal 287 a binario, usando 10 bits.

#### Solución:

Primero, se define la cantidad de dígitos binarios o bits en los que se desea representar el número decimal y se coloca la ponderación que corresponde a cada dígito.

512	256	128	64	32	16	8	4	2	1
-----	-----	-----	----	----	----	---	---	---	---

Luego, se busca la ponderación binaria más cercana o igual al número decimal, visto desde la parte inferior. Esta ponderación debe ser restada del número decimal. En la posición en la que se encuentra la ponderación encontrada, se escribirá 1. En este caso, dicha ponderación sería 256. La operación y resultado parcial quedaría así:

**287 Número decimal**

-256 Ponderación binaria

**31 Resultado de la resta**

<b>1</b>	256	128	64	32	16	8	4	2	1
----------	-----	-----	----	----	----	---	---	---	---

Este último procedimiento debe realizarse de forma iterativa, hasta que el resultado de la resta sea cero. En las ponderaciones donde no se haya colocado el dígito 1, se deberá llenar con ceros, como se muestra a continuación:

0	1	0	0	0	1	1	1	1	1
---	---	---	---	---	---	---	---	---	---

El número binario equivalente sería: 0100011111.

### 1.3.2 Transformación binario a decimal

La transformación de números expresados en sistema binario a sus equivalentes en sistema decimal es de suma importancia, ya que permite entender con mayor claridad las operaciones básicas que abordamos más adelante. Para ello, presentamos el siguiente ejemplo:

**Ejercicio 1.2:** Transformar el número binario 101011001 a su equivalente decimal.

**Solución:**

De manera similar, la transformación de un número binario a su equivalente decimal empieza definiendo las ponderaciones que tiene cada dígito del número binario.

$$\begin{array}{r} 1 \quad 0 \quad 1 \quad 0 \quad 1 \quad 1 \quad 0 \quad 0 \quad 1 \\ \hline 256 \quad 128 \quad 64 \quad 32 \quad 16 \quad 8 \quad 4 \quad 2 \quad 1 \end{array}$$

Luego, se suman las ponderaciones binarias sobre las cuales haya un 1. El resultado de esta suma será el valor decimal buscado. En este caso, el proceso sería:

$$\begin{array}{r} + \quad 256 \\ 64 \\ 16 \\ 8 \\ 1 \\ \hline 345 \end{array}$$

Para las otras transformaciones, por ejemplo, hexadecimal binario o viceversa, es conveniente usar las equivalencias de la tabla 1.1. Como señalamos al inicio, es aconsejable transformar primero al sistema binario y, posteriormente, al sistema que se desee. Este paso ahorrará tiempo y esfuerzo.

La tabla 1.1 presenta los sistemas de numeración más utilizados hoy en día.

**Tabla 1.1.**

*Sistemas de numeración*

Decimal	Hexadecimal	Binario
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

### 1.4 Aritmética binaria

La aritmética es la rama de las matemáticas que se dedica al estudio, las relaciones y la resolución de problemas usando números. Algunas operaciones aritméticas son bastante conocidas en sistemas de numeración decimal: suma, resta, multiplicación, división y potenciación. A continuación, estudiamos algunas de ellas desde el punto de vista de los sistemas binarios. Cabe señalar que muchos de los criterios usados en sistemas decimales son aplicables y de mucha utilidad para entender el funcionamiento de sistemas digitales y microcontrolados más complejos.

### 1.5 Suma binaria

La suma es la operación aritmética más básica. No obstante, es una de las más importantes en los sistemas de numeración binaria. En la actualidad, todas las operaciones matemáticas que un procesador realiza tienen asociados, en su interior, mi-

les de millones de operaciones básicas de suma. Estas operaciones se ejecutan en millonésimas de segundo, y son totalmente imperceptibles para el ser humano. Antes de entrar a revisar la suma binaria, explóremos, de manera general, la suma decimal y su proceso.

**Ejercicio 1.3:** Realizar una suma de dos números decimales 19 y 26.

**Solución:**

En este ejemplo, detallamos paso a paso el proceso para obtener el resultado.

Ponderación	$10^1$	$10^0$
Carry	1	
Sumandos	+ 1	9
	2	6
Resultado	4	5

Como señalamos al inicio de este capítulo, la base del sistema decimal es 10. Cada dígito de este sistema posee su propia ponderación. En el ejemplo anterior, al sumar la ponderación  $10^0$ , más conocido como unidades, el resultado ( $9 + 6 = 15$ ) supera al valor de la base ( $15 \geq 10$ ), por lo que es imposible representar dicho valor con los dígitos del sistema de numeración decimal (0 al 9). En este caso, es necesario acarrear este exceso (10) a la siguiente ponderación y dejar su diferencia ( $15 - 10 = 5$ ) como dígito de las unidades. Continuando con el proceso, al sumar la ponderación  $10^1$ , más conocido como decenas, el resultado ( $1 + 1 + 2 = 4$ ) no supera el valor de la base (10); por lo tanto, no es necesario acarrear ningún valor a la siguiente ponderación.

**Ejercicio 1.4:** Sumar dos números binarios aplicando el mismo procedimiento usado para el sistema de numeración decimal.

**Solución:**

Ponderación	$2^2$	$2^1$	$2^0$
Carry	1	1	
Sumandos	+ 0	1	1
	0	0	1
Resultado	1	0	0

Al sumar los dígitos de la ponderación  $2^0$ , el resultado ( $1 + 1 = 2$ ) es igual al valor de la base ( $2 \geq 2$ ), lo cual es imposible representar con los dígitos del sistema de numeración (0 o 1). En este caso, es necesario acarrear este exceso (2) a la siguiente ponderación y dejar su diferencia ( $2 - 2 = 0$ ) como dígito menos significativo. Continuando con el proceso, al sumar la ponderación  $2^1$ , el resultado ( $1 + 1 = 2$ ), nuevamente, es igual al valor de la base ( $2 \geq 2$ ). Por lo tanto, se acarrea el exceso y se representa únicamente su diferencia ( $2 - 2 = 0$ ) como el siguiente dígito. Por último, la suma de la ponderación  $2^2$  da como resultado 1, que sí puede ser representado en el sistema binario.

En una operación aritmética de suma binaria, este análisis puede ser demasiado exhaustivo. Sin embargo, puede ser reemplazado por un proceso de memorización, más sencillo y rápido, como el que se expone en la tabla 1.2.

**Tabla 1.2.**

*Operación de suma*

Operación	Resultado	Carry
$0 + 0$	0	0
$0 + 1$	1	0
$1 + 0$	1	0
$1 + 1$	0	1

Del mismo modo, se puede sumar usando sistemas hexadecimales. Para este propósito, es necesario trabajar con la equivalencia decimal, así:

Ponderación	$16^2$	$16^1$	$16^0$
Carry	1	1	
Sumandos	+ 0	A	7
	0	D	F
Resultado	1	8	6

Al sumar los dígitos de la ponderación  $16^0$ , el dígito F se considera con su ponderación decimal, es decir, la suma sería ( $7 + 15 = 22$ ). Este valor supera al de la base ( $22 \geq 16$ ), por lo que es imposible representarlo con los dígitos del sistema de numeración hexadecimal. En este caso, se debe acarrear este exceso (16) a la siguiente ponderación y de-

jar su diferencia ( $22 - 16 = 6$ ) como dígito menos significativo. Continuando con el proceso, y considerando las ponderaciones decimales, al sumar la ponderación  $16^1$ , el resultado ( $1 + A + D = 24$ ) supera el valor de la base ( $24 \geq 16$ ). En consecuencia, se acarrea el exceso y solo se representa su diferencia ( $24 - 16 = 8$ ) como el siguiente dígito. Para concluir, la suma de la ponderación  $16^2$  da como resultado 1, que sí puede ser representado en el sistema hexadecimal.

En la actualidad, la mayoría de las operaciones aritméticas binarias que realizamos utilizan el sistema de numeración hexadecimal, gracias a su simplicidad y semejanza con el sistema decimal. Los sistemas computacionales continúan trabajando en binario, por lo que es común encontrar microprocesadores de 32, 64 y 128 bits. Para un ser humano, manejar esta gran cantidad de dígitos binarios sería prácticamente imposible.

**Ejercicio 1.5:** Sumar los números hexadecimales 4F9C y DAC1.

**Solución:**

Ponderación	$16^4$	$16^3$	$16^2$	$16^1$	$16^0$
Carry	1	1	1		
Sumandos	+		4	F	9 C
		D	A	C	1
Resultado	1	2	A	5	D

## 1.6 Resta binaria

Para los niños, el hecho de representar una cantidad con signo negativo constituye un verdadero reto. Muchos se preguntan ¿cómo puede ser posible que existan, menos dos ( $-2$ ) manzanas? Este concepto puede ser difícil de asimilar en un principio. Es así como la resta no existe en los sistemas binarios; por el contrario, aparecen otros conceptos, como el complemento, que permiten manejar una operación de resta como una suma.

Esto no implica que en sistemas binarios no se tengan representaciones de números negativos. La principal diferencia que existe es que, en estos sistemas, la resta termina siendo una suma. Esto se

logra gracias a dos conceptos: el primero tiene que ver con la representación de números negativos y el segundo, con el equivalente positivo de números negativos, llamado complemento.

Un número binario de  $n$  bits se considera positivo hasta que no se indique lo contrario. No obstante, para convertir este número a su equivalente negativo, se debe asignar un bit para el signo. Esto reduce a la mitad la cantidad de números positivos, pero brinda la oportunidad de representar números negativos. Por ejemplo, si se dispone de un número de 4 bits, y se lo usa para representar números positivos, este estará en la capacidad de almacenar números desde 0 hasta 15. Sin embargo, si se asigna un bit para el signo, apenas podrá almacenar números positivos desde 0 hasta 7. Pero ¿qué sucede con los números negativos? En estos apartados mostramos cómo pueden ser representados y cuáles son sus ventajas y desventajas.

### 1.6.1 Complemento a1

Para convertir un número binario usando el complemento a1, primero se debe asignar un bit para el signo y, luego, cambiar unos por ceros, y viceversa. Si el bit del signo es 0, el número se considera positivo, y si el bit del signo es 1, el número se considera negativo.

**Ejercicio 1.6:** Convertir el número binario 0b1110010 a su complemento a1.

**Solución:**

Signo

A 0 1 1 1 0 0 1 0

$\bar{A}$  1 0 0 0 1 1 0 1 Complemento a1

El complemento a1 también puede ser aplicado en números expresados en el sistema hexadecimal. Una primera alternativa es transformar el dígito hexadecimal a binario y cambiar ceros por unos y unos por ceros.

**Ejercicio 1.7:** Convertir el número hexadecimal 0XA8E5 a su complemento a1.

## Solución:

### Signo

A	0	A	8	E	5	Hexadecimal
	0	1010	1000	1110	0101	Binario
$\widehat{A}$	1	0101	0111	0001	1010	Complemento a1 binario
$\widehat{A}$	1	5	7	1	A	Complemento a1 hexadecimal

Nótese que si se suma cada dígito hexadecimal con su respectivo complemento  $a1$ , el valor que se obtiene siempre es F, o 15 en decimal. Esto brinda una pista muy importante respecto al complemento en números hexadecimales. El complemento  $a1$  corresponde al valor necesario para alcanzar el dígito más alto, que en este caso es F. Siendo así, este paso permite ahorrar tiempo valioso de cálculo para futuros ejercicios.

## 1.6.2 Complemento a2

Ciertamente, complemento  $a2$  es una de las representaciones binarias más usadas a nivel de sistemas computacionales. Esto, debido a su sencillez y practicidad, así como a la posibilidad de representar el cero de una única forma, lo cual no es posible con el complemento  $a1$ . El proceso es similar al usado anteriormente, es decir, se debe definir un bit para el signo. Hay dos alternativas para hallar el complemento  $a2$ ; la primera es convertir el número a complemento  $a1$  y, luego, sumar 1.

**Ejercicio 1.8:** Convertir el número binario 0b1110010 a su complemento a2.

## Solución:

### Signo

A	0	1	1	1	0	0	1	0	Número para transformar
$\widehat{A}$	1	0	0	0	1	1	0	1	Complemento a1

+ 1 Se suma 1

A*	1	0	0	0	1	1	1	0	Complemento a2
----	---	---	---	---	---	---	---	---	----------------

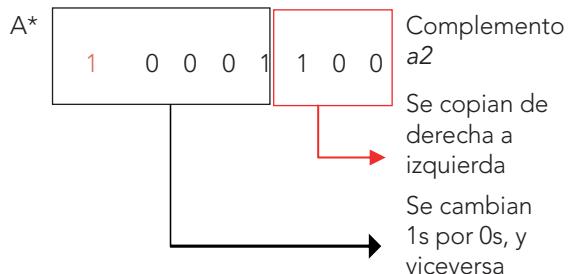
La segunda alternativa consiste en aplicar un pequeño algoritmo que dice: empezar de derecha a izquierda, copiar todos los ceros hasta encontrar un uno, que también se lo copiará; de ahí en adelante, cambiar ceros por unos y unos por ceros. Es importante mencionar que el bit del signo debe estar previamente definido.

**Ejercicio 1.9:** Convertir el número binario 0b1110010 a su complemento a2 usando el algoritmo.

## Solución:

### Signo

A	0	1	1	1	0	1	0	0	Número para transformar
---	---	---	---	---	---	---	---	---	-------------------------



El complemento  $a2$  también puede ser utilizado con números hexadecimales. En este caso, se recomienda hallar el complemento  $a1$  y, luego, sumar 1 a ese resultado. No aconsejamos usar el algoritmo anterior, puesto que su aplicabilidad puede generar algunas confusiones.

**Ejercicio 1.10:** Convertir el número hexadecimal 0XDE8C a su complemento a2.

## Solución:

### Signo

A	0	D	E	8	C	Número por transformar
$\widehat{A}$	1	2	1	7	3	Complemento a1 hexadecimal

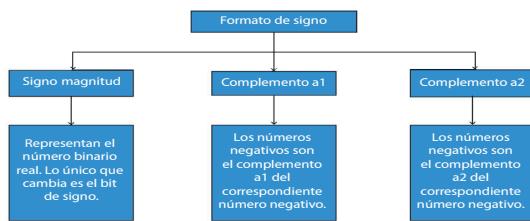
+ 1 Sumar 1

A*	1	2	1	7	4	Complemento a2 hexadecimal
----	---	---	---	---	---	----------------------------

## 1.7 Números con signo

Existen tres formas para representar números con signo: 1) signo magnitud, 2) complemento  $a_1$ , y 3) complemento  $a_2$ . La última forma es la que se utiliza actualmente en sistemas computacionales. La principal razón radica en que el complemento  $a_2$  tiene solo una forma de representar el cero, mientras que las otras dos tienen dos formas diferentes para hacerlo. Esto generaría problemas a los sistemas microprocesados para interpretar la información. La figura 1.1 exhibe las diferentes formas de representar números con signo.

Figura 1.1 Formas de representación de números con signo



## 1.8 Signo magnitud

Para representar un número usando signo magnitud, primero se debe transformar el número a binario y añadir un bit para el signo. El siguiente paso es cambiar únicamente el signo del número a negativo y mantener la magnitud. Esta forma de representación de números negativos es extremadamente sencilla. No obstante, el principal problema surge al momento de representar el cero, ya que existen dos formas diferentes de representación, como se indica a continuación:

### Signo

0	0	0	0	0	0	0	0	Número 0 positivo
1	0	0	0	0	0	0	0	Número 0 negativo

Ejercicio 1.11: Representar el número 25 en su equivalente negativo usando signo magnitud.

## Solución:

En este caso, el número será representado en 7 bits y se agregarán un octavo bit para el signo, así:

### Signo

0	0	0	1	1	0	0	1	Número 25 positivo
1	0	0	1	1	0	0	1	Número 25 negativo

## 1.9 Signo usando complemento $a_1$

Para representar un número usando complemento  $a_1$ , primero se debe transformar el número a binario y añadir un bit para el signo. El siguiente paso es transformar el número usando el complemento  $a_1$ . Esta forma de representación de números negativos es bastante conocida. Sin embargo, al igual que en signo magnitud, el problema surge cuando se quiere representar el cero, ya que existen dos formas diferentes de representación:

### Signo

0	0	0	0	0	0	0	0	Número 0 positivo
1	1	1	1	1	1	1	1	Número 0 negativo

Ejercicio 1.12: Representar el número 25 en su equivalente negativo usando complemento  $a_1$ .

## Solución:

### Signo

0	0	0	1	1	0	0	1	Número 25 positivo
1	1	1	0	0	1	1	0	Número 25 negativo

## 1.10 Signo usando complemento $a_2$

Para representar un número usando complemento  $a_2$ , primero se debe transformar el número a binario y añadir un bit para el signo. A continuación, se transforma el número usando el complemento  $a_2$ . En la actualidad, todos los sistemas computacionales usan esta forma de representación de números negativos. La principal razón es que, a diferencia de las dos representaciones anteriores, signo magnitud y complemento  $a_1$ , en esta existe una sola forma de representar el cero.

Ejercicio 1.13: Convertir el número binario 0b00000000 a su complemento  $a_2$ .

### Solución:

#### Signo

0	0	0	0	0	0	0	0	Número 0 positivo
1	1	1	1	1	1	1	1	Carry
1	1	1	1	1	1	1	1	Número 0 negativo complemento a1
							+ 1	
1	0	0	0	0	0	0	0	Número 0 en complemento a2

**Ejercicio 1.14:** Representar el número 25 en su equivalente negativo usando complemento a2.

### Solución:

0	0	0	1	1	0	0	1	Número 25 positivo
								Carry
1	1	1	0	0	1	1	0	Número 25 negativo complemento a1
							+ 1	
1	1	1	0	0	1	1	1	Número 25 negativo complemento a2

La tabla 1.3 permite apreciar, de manera resumida, las tres formas de representación de números con signo, para números de 3 bits, incluido el signo. En ella se identifican los casos en los cuales la representación del cero es diferente. Como se aprecia en la columna del complemento a2, tenemos más números negativos que positivos. De forma general, para números con  $n$  bits, se usa esta fórmula para saber el rango de valores positivos y negativos ( $-2^{n-1}$ ) hasta ( $+2^{n-1}-1$ ).

Tabla 1.3.

Representación de números con signo

Número	Signo magnitud	Complemento a1	Complemento a2
011	+3	+3	+3
010	+2	+2	+2
001	+1	+1	+1
000	+0	+0	0
100	-0	-3	-4
101	-1	-2	-3
110	-2	-1	-2
111	-3	-0	-1

Ahora, aplicamos el concepto del complemento a1 a una resta binaria, a través de un problema.

**Ejercicio 1.15:** Restar dos números: A = 17 y B = 37 usando complemento a1.

### Solución:

El resultado de restar A y B es R = A - B = A + (-B) = A + B\*

Se usa el complemento a1 para hallar el equivalente negativo (B\*) del número positivo B.

En primer lugar, se transforma los números a binarios y se añade un bit para el signo.

#### Signo

B	0	0	1	0	0	1	0	1
B*	1	1	0	1	1	0	1	0

Una vez hallado el complemento a1, se suma A + B\*.

	Signo	1	Carry
A	0	0 0 1 0 0 0 1	
B*	1	1 0 1 1 0 1 0	Complemento a1
R	1	1 1 0 1 0 1 1	

Como era de esperarse el resultado es negativo. Si observamos detenidamente, la magnitud arroja un valor de 107, que a simple vista es erróneo. Sin embargo,

se debe recordar que este valor corresponde al complemento o negativo del número positivo. Para hallar la magnitud del número, en signo positivo, será necesario volver a complementar este resultado.

R 0 0 0 1 0 1 0 0 Positivo

Esta vez, el resultado es 20, ya que se está hallando la magnitud del número positivo.

Ahora se analiza qué sucede si se resta  $B - A$ . Se sigue el procedimiento anterior. Primero, se halla el complemento  $a1$  de A, es decir  $A^*$  y, luego, se suma  $A^* + B$ . Se observa algo interesante a continuación:

#### Ejercicio 1.16: Restar dos números:

$B = 37$  y  $A = 17$  usando complemento  $a1$ .

#### Solución:

$$\begin{array}{r} \text{Signo} \\ \hline A & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ A^* & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \end{array} \quad \text{Complemento } a1$$

$$\begin{array}{r} \text{Signo} \\ \hline A^* + & 1 & 1 & 1 & 1 & 1 & & \\ B & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ \hline R & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \end{array} \quad \text{Complemento } a1$$

El *carry* que se genera después del bit del signo se conoce como *carry del bit del signo*. En el caso de usar complemento  $a1$  para la resta, este exceso debe ser sumado siempre al resultado final para ajustar la respuesta.

$$\begin{array}{r} \text{Signo} \\ \hline R & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ & & & & & & + & 1 \\ \hline & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \end{array} \quad \text{Carry}$$

#### Ejercicio 1.17: Restar dos números:

$A = 0X9A7$  y  $B = 0X3FC$  usando complemento  $a1$ .

#### Solución:

En primer lugar, se transforma el número B a su complemento  $a1$ .

$$\begin{array}{r} \text{Signo} \\ \hline B & 0 & 3 & F & C \\ B^* & 1 & C & 0 & 3 \end{array} \quad \text{Complemento } a1$$

Una vez hallado el complemento  $a1$ , se suma  $A + B^*$ .

$$\begin{array}{r} \text{Signo} \\ \hline A & + & 0 & 9 & A & 7 \\ B^* & & 1 & C & 0 & 3 \\ \hline R & 1 & 0 & 5 & A & A \end{array} \quad \text{Complemento } a1$$

Como hemos descrito anteriormente, es necesario sumar el *carry* del bit del signo para ajustar la respuesta.

$$\begin{array}{r} \text{Signo} \\ \hline R & 0 & 5 & A & A \\ & & + & 1 \\ \hline & 0 & 5 & A & B \end{array} \quad \text{Carry}$$

#### Ejercicio 1.18: Restar dos números:

$A = 0XC59$  y  $B = 0XA89$  usando complemento  $a2$ .

#### Solución:

Esta vez se transforma el número a su complemento  $a2$ . Para ello, primero se transforma a complemento  $a1$  y, luego, se suma una unidad.

$$\begin{array}{r} \text{Signo} \\ \hline B & 0 & A & 8 & 9 \\ B^* & 1 & 5 & 7 & 6 \end{array} \quad \text{Complemento } a1$$

$$\begin{array}{r} + 1 \\ \hline B^* & 1 & 5 & 7 & 7 \end{array} \quad \text{Complemento } a2$$

$$\begin{array}{r} \text{Signo} \\ \hline A & + & 0 & C & 5 & 9 \\ B^* & & 1 & 5 & 7 & 7 \\ \hline R & 1 & 0 & 1 & D & 0 \end{array} \quad \text{Complemento } a2$$

A pesar de que en esta ocasión también aparece un *carry del bit del signo*, en complemento  $a_2$  no es necesario sumar este dígito al resultado final, ya que previamente ya fue incluido al momento de la transformación a complemento  $a_2$ . Este sistema, junto con otros, se utiliza, en la actualidad, en los computadores y microcontroladores comerciales.

### 1.11 Compuertas lógicas

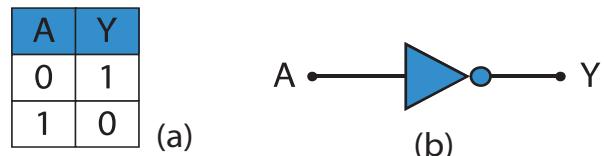
Las compuertas lógicas, como circuitos integrados o chips, están conformadas por dispositivos electrónicos semiconductores. Son usadas, sobre todo, para implementar circuitos electrónicos que se originan de expresiones lógicas booleanas. Su funcionamiento está determinado por tablas de verdad, en las que sus estados lógicos son representados por rangos de voltaje denominados “alto” o “bajo” o, generalmente, uno lógico (1L) o cero lógico (0L), respectivamente. En otras aplicaciones, es más conveniente representar estos estados como “verdadero” o “falso”, o también como “abierto” o “cerrado”. Hoy en día, el uso de estos chips es poco común. No obstante, los sistemas computacionales modernos están formados por miles de millones de compuertas lógicas, tanto así, que sería prácticamente imposible su funcionamiento sin ellas. Como suele suceder en la electrónica, estas compuertas están sujetas al estándar ANSI/IEEE 91-1984, que define su simbología y funcionamiento. Dicho estándar establece, entre otros aspectos, que las señales de entrada se ubicarán a la izquierda y las señales de salida, a la derecha. A continuación, mostramos las principales características de estos dispositivos de origen digital.

#### 1.11.1 Compuerta lógica NOT

La compuerta lógica NOT realiza una operación de inversión del estado lógico de entrada. Es decir, si la entrada se encuentra en alto, en la salida se obtendrá bajo. A nivel de lógica binaria, realiza el mismo trabajo que el complemento  $a_1$ . La figura 1.2 expone la tabla de verdad y su símbolo.

**Figura 1.2**

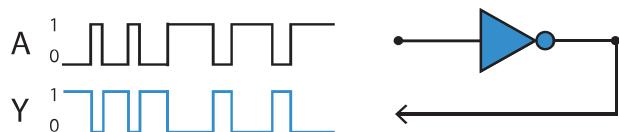
Símbolo lógico y tabla de verdad de la compuerta NOT



La presencia de un círculo a la entrada o salida de una compuerta lógica señala una inversión de estado lógico. Este distintivo se encuentra, también, en otras compuertas y circuitos integrados, e indica una inversión lógica de la señal de entrada o salida. La figura 1.3 expone el comportamiento de la compuerta NOT cuando es sometida a una señal digital que oscila en el tiempo.

**Figura 1.3**

Operación de la compuerta lógica NOT

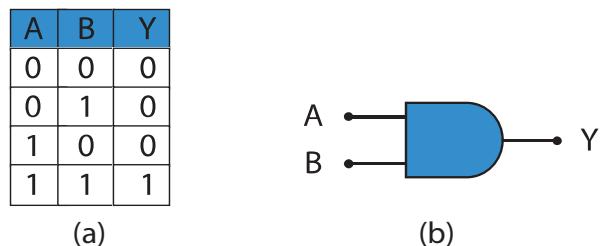


#### 1.11.2 Compuerta lógica AND

La compuerta lógica AND posee dos o más entradas y una sola salida. Al tener mínimo dos entradas, la tabla de verdad más pequeña puede ser de cuatro combinaciones. La tabla de verdad y su símbolo se muestran en la figura 1.4.

**Figura 1.4**

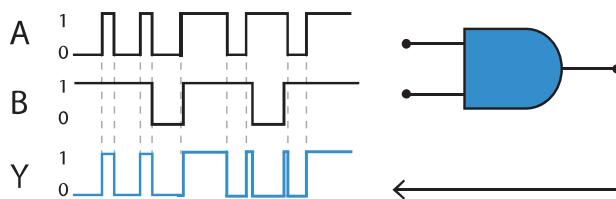
Símbolo lógico y tabla de verdad de la compuerta AND



El funcionamiento de la compuerta lógica AND indica que solo cuando las dos entradas son verdaderas, la salida es verdadera. En todas las otras combinaciones, la salida es falsa. A nivel de circuitos eléctricos, esta operación se relaciona de forma directa con una conexión en serie de dos o más interruptores. Es decir, para que la corriente pueda fluir, todos los interruptores deben estar cerrados; mientras que, si al menos uno se abre, el flujo de corriente se interrumpe. A nivel de circuitos electrónicos, esta compuerta es ampliamente utilizada para habilitar o deshabilitar el paso de información de la entrada a la salida. La figura 1.5 exhibe el comportamiento de la compuerta AND cuando es sometida a dos señales digitales que oscilan en el tiempo.

**Figura 1.5**

Operación de la compuerta lógica AND



### 1.11.3 Compuerta lógica OR

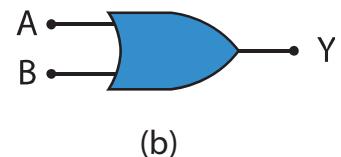
La compuerta lógica OR también puede poseer dos o más entradas y una sola salida. La figura 1.6 expone la tabla de verdad y su símbolo. El funcionamiento de la compuerta lógica OR indica que únicamente cuando las dos entradas son falsas, la salida es falsa. En todas las otras posibilidades, la salida es verdadera. A nivel de circuitos eléctricos, esta operación se relaciona de forma directa con una conexión en paralelo de dos o más interruptores. En este caso, la corriente tendría dos o más caminos para circular y bastaría con que uno de los interruptores esté cerrado para que lo haga. El único caso en el que la corriente no circularía sería cuando todos los interruptores estén abiertos.

**Figura 1.6**

Símbolo lógico y tabla de verdad de la compuerta OR

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

(a)

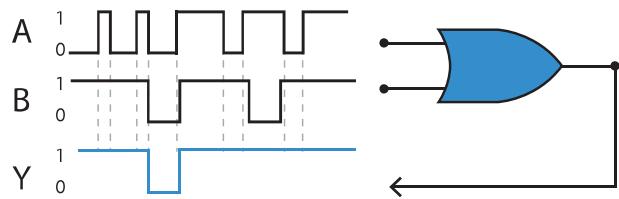


(b)

La figura 1.7 ilustra el comportamiento de una compuerta OR cuando es sometida a dos señales digitales que oscilan en el tiempo.

**Figura 1.7**

Operación de la compuerta lógica OR

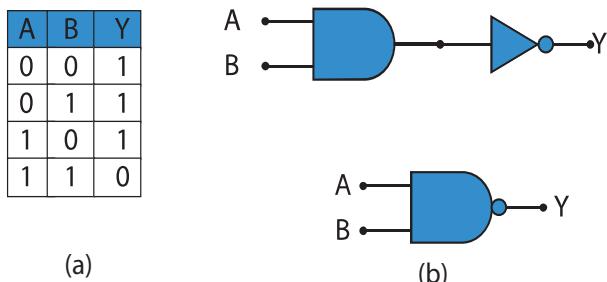


### 1.11.4 Compuerta lógica NAND

Esta compuerta lógica es el producto de la combinación de dos compuertas: una AND y una NOT. El uso de este tipo de compuertas simplifica los circuitos lógicos, así como su representación algebraica. Además, universaliza el uso de un solo tipo de compuertas lógicas. La figura 1.8 indica la tabla de verdad, el símbolo estándar y la equivalencia con compuertas AND y NOT.

**Figura 1.8**

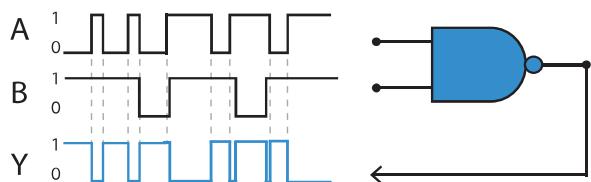
Símbolo lógico y tabla de verdad de la compuerta NAND



El funcionamiento de la compuerta lógica NAND es exactamente inverso al de la compuerta AND. A nivel de circuitos eléctricos, esta operación se relaciona de forma directa con la conexión de interruptores conectados en serie, los cuales que están normalmente cerrados. La figura 1.9 ilustra el comportamiento de la compuerta NAND cuando es sometida a dos señales digitales que oscilan en el tiempo.

**Figura 1.9**

Operación de la compuerta lógica NAND

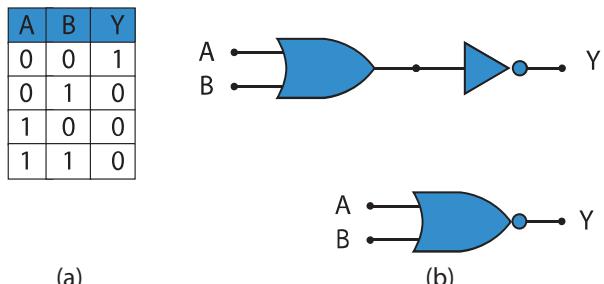


### 1.11.5 Compuerta lógica NOR

Esta compuerta lógica es el producto de la combinación de dos compuertas: una OR y una NOT. La figura 1.10 muestra la tabla de verdad, el símbolo estándar y la equivalencia con compuertas OR y NOT.

**Figura 1.10**

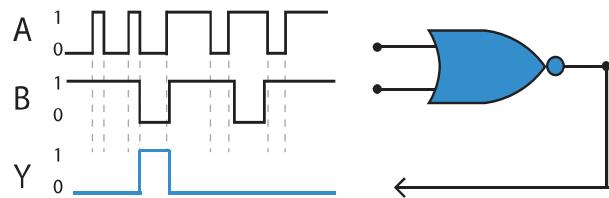
Símbolo lógico y tabla de verdad de la compuerta NOR



El funcionamiento de la compuerta lógica NOR es exactamente inverso al de la compuerta OR. A nivel de circuitos eléctricos, esta operación se relaciona de forma directa con la conexión de interruptores en paralelo, los cuales normalmente están cerrados. La figura 1.11 expone el comportamiento de la compuerta NOR cuando es sometida a dos señales digitales que oscilan en el tiempo.

**Figura 1.11**

Operación de la compuerta lógica NOR

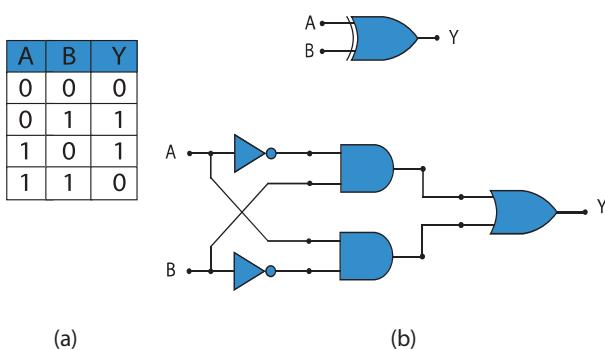


### 1.11.6 Compuerta lógica XOR

Esta compuerta lógica es el producto de la combinación de tres compuertas: AND, OR y NOT. La figura 1.12 exhibe la tabla de verdad, la equivalencia con compuertas AND, OR y NOT, y su símbolo.

**Figura 1.12**

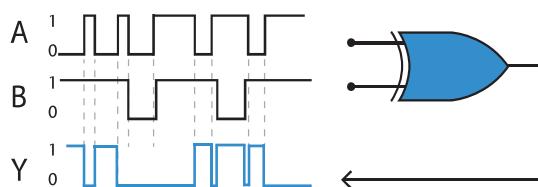
Símbolo lógico y tabla de verdad de la compuerta XOR



El funcionamiento de la compuerta lógica XOR es muy particular. La primera parte de la tabla de verdad actúa como una compuerta lógica OR, mientras que la segunda mitad lo hace como si fuera una compuerta NOT. Además, esta compuerta representa exactamente la operación de suma binaria entre dos operadores lógicos. La figura 1.13 ilustra el comportamiento de la compuerta NOR cuando es sometida a dos señales digitales que oscilan en el tiempo.

**Figura 1.13**

Operación de la compuerta lógica XOR



## 1.12 Códigos binarios

Los números binarios pueden ser usados para representar dígitos del sistema de numeración decimal, ya que este es más fácil de ser interpretado y manejado por los seres humanos. Para ello, se deben tener en cuenta algunas observaciones y consideraciones que se explicamos a continuación.

### 1.12.1 BCD (Binary Coded Decimal)

Esta representación binaria permite codificar un número decimal en 4 dígitos binarios. Esto quiere decir que, de las 16 posibles combinaciones, apenas las 10 primeras serán utilizadas. Esta representación es muy usada cuando se trabaja con *display* de 7 segmentos. La tabla 1.4 expone una equivalencia entre el sistema binario y decimal, así como el equivalente en la representación BCD.

**Tabla 1.4**

Representación de números binarios en BCD

Decimal	Binario	BCD
0	0000	0000
1	0001	0001
2	0010	0010
3	0011	0011
4	0100	0100
5	0101	0101
6	0110	0110
7	0111	0111
8	1000	1000
9	1001	1001
10	1010	No definido
11	1011	No definido
12	1100	No definido
13	1101	No definido
14	1110	No definido
15	1111	No definido

Para representar números decimales en BCD, basta con reemplazar cada dígito decimal con su equivalente BCD.

**Ejercicio 1.19:** Representar el número 1983 en su equivalente BCD.

**Solución:**

Decimal	1	9	8	3
BCD	0001	1001	1000	0011

### 1.12.2 Gray

Este sistema de codificación, a diferencia del hexadecimal o BCD, no posee ninguna relación matemática ni ponderaciones para su transformación. Una de las principales características es que apenas uno de sus bits cambia a la vez. Esta condición permite que sus aplicaciones se orienten a sistemas extremadamente robustos, como codificadores de posición para brazos robóticos, robots móviles o medidores de velocidad en motores eléctricos industriales, bandas transportadoras de alta precisión, prensas hidráulicas automatizadas y muchas otras. Además, esta codificación puede tener cualquier número de bits. La tabla 1.5 describe la representación de números en código Gray de 4 bits.

**Tabla 1.5**

Representación de números en código Gray

Decimal	Binario	Gray
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100
9	1001	1101
10	1010	1111
11	1011	1110
12	1100	1010
13	1101	1011
14	1110	1001
15	1111	1000

El uso de tablas puede ser complicado cuando el número de bits usados es mayor a cuatro. En esos casos, se puede transformar un binario a código Gray, usando los siguientes pasos: empezar desde la izquierda a derecha; copiar el primer dígito binario como primer dígito Gray; luego, sumar los dígitos binarios adyacentes uno a uno, y el resulta-

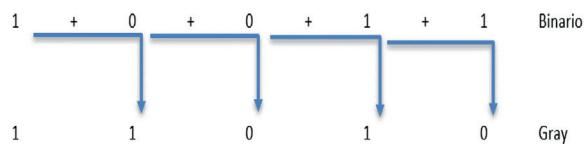
do será el siguiente dígito Gray; repetir el procedimiento hasta terminar con todos los dígitos.

**Ejercicio 1.20:** Representar el número binario 10011 en su equivalente Gray.

**Solución:**

**Figura 1.14**

Transformación binaria a Gray



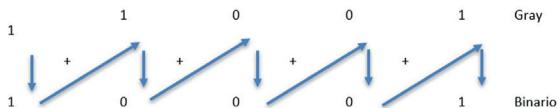
En cambio, si se desea transformar un número en código Gray a binario, el procedimiento es el siguiente: empezar de izquierda a derecha; copiar el primer dígito Gray como primer dígito binario; luego, sumar el primer dígito binario con el siguiente dígito Gray, y el resultado será el segundo dígito binario; repetir el procedimiento hasta terminar con todos los dígitos.

**Ejercicio 1.21:** Representar el número Gray 11001 en su equivalente binario.

**Solución:**

**Figura 1.15**

Transformación Gray a binario



### 1.12.3 ASCII (American Standard Code for Information Interchange)

Esta codificación es, probablemente, la más conocida a nivel mundial, aunque en la actualidad es algo obsoleta para la gran mayoría de computadoras. Sin embargo, para sistemas microcontrolados y

embebidos, es la codificación preferida para intercambiar información. Este estándar utiliza 7 bits para codificar letras, números, símbolos especiales y señales de control. Con la aparición de las computadoras, estas reemplazaron las antiguas máquinas de escribir. Esto, a su vez, permitió desarrollar este estándar, que posee todos los símbolos que se pueden encontrar en un teclado común. La tabla 1.6 expone todos los caracteres ASCII de 7 bits.

**Tabla 1.6**

*Caracteres ASCII*

DEC	HEX	Carácter	DEC	HEX	Carácter
0	00	NUL–Null	64	40	@
1	01	SOH–Start of Heading	65	41	A
2	02	STX–Start of Text	66	42	B
3	03	ETX–End of Text	67	43	C
4	04	EOT–End of Transmission	68	44	D
5	05	ENQ–Enquiry	69	45	E
6	06	ACK–Acknowledge	70	46	F
7	07	BEL–Bell	71	47	G
8	08	BS–Backspace	72	48	H
9	09	TAB–Horizontal Tab	73	49	I
10	0A	LF–Line Feed (New Line)	74	4A	J
11	0B	VT–Vertical Tab	75	4B	K
12	0C	FF–Form Feed (New Page)	76	4C	L
13	0D	CR–Carriage Return	77	4D	M
14	0E	SO–Shift Out	78	4E	N
15	0F	SI–Shift In	79	4F	O

16	10	DLE–Data Link Escape	80	50	P
17	11	DC1–Device Control 1	81	51	Q
18	12	DC2–Device Control 2	82	52	R
19	13	DC3–Device Control 3	83	53	S
20	14	DC4–Device Control 4	84	54	T
21	15	NAK–Negative Acknowledge	85	55	U
22	16	SYN–Synchronous Idle	86	56	V
23	17	ETB–End of Trans. Block	87	57	W
24	18	CAN–Cancel	88	58	X
25	19	EM–End of Medium	89	59	Y
26	1A	SUB–Substitute	90	5A	Z
27	1B	ESC–Escape	91	5B	[
28	1C	FS–File Separator	92	5C	
29	1D	GS–Group Separator	93	5D	]
30	1E	RS–Record Separator	94	5E	^
31	1F	US–Unit Separator	95	5F	_
32	20	Space	96	60	`
33	21	!	97	61	a
34	22	"	98	62	b
35	23	#	99	63	c
36	24	\$	100	64	d
37	25	%	101	65	e
38	26	&	102	66	f
39	27	'	103	67	g
40	28	(	104	68	h
41	29	)	105	69	i

42	2A	*	106	6A	j
43	2B	+	107	6B	k
44	2C	,	108	6C	l
45	2D	-	109	6D	m
46	2E	.	110	6E	n
47	2F	/	111	6F	o
48	30	0	112	70	p
49	31	1	113	71	q
50	32	2	114	72	r
51	33	3	115	73	s
52	34	4	116	74	t
53	35	5	117	75	u
54	36	6	118	76	v
55	37	7	119	77	w
56	38	8	120	78	x
57	39	9	121	79	y
58	3A	:	122	7A	z
59	3B	;	123	7B	{
60	3C	<	124	7C	
61	3D	=	125	7D	}
62	3E	>	126	7E	~
63	3F	?	127	7F	DEL

Tomado de Britannica, 2019

A pesar de que 7 bits permiten codificar una gran cantidad de caracteres, muchos otros no tienen cabida en este formato. En ese sentido, la primera actualización al estándar se conoce como ASCII extendido; esta usa 8 bits para representar hasta 256 caracteres diferentes. También se conoce como codificación ANSI, en referencia al American National Standards Institute. Algunas de las extensiones de ASCII son Latin5 (ISO 8859-5 Turco), Latin7 (ISO 8859-7 Celtic) o Latin10 (ISO 8859-10 del sudeste europeo).

Con la masificación de las computadoras a nivel mundial y la traducción de los teclados y archivos de texto a casi todos los idiomas internacionales, las codificaciones ASCII y ANSI quedaron muy li-

mitadas. En la actualidad, existen representaciones con un número mayor de bits, que permiten intercambiar archivos de texto en diferentes formatos, sin perder sus caracteres originales. La codificación estándar, hoy en día, es UNICODE. Esta, a lo largo de los años, también ha sido modificada y actualizada. El primer formato UNICODE en aparecer fue UTF-7, que es compatible con ASCII por tener 7 bits para representar caracteres. Luego, surge UTF-8, que tiene la característica de poder representarse en un rango de 1 a 4 Bytes. También se encuentra UTF-16, que utiliza dos bits para su representación e incluye caracteres europeos, asiáticos y africanos. En esta codificación aparecen UTF-16 Little Endian, que se usa en sistemas operativos Windows, Linux e iOS. Finalmente, el formato UTF-32 utiliza 4 Bytes para representar caracteres. Requiere mayor espacio de memoria, pero es capaz de garantizar compatibilidad absoluta con cualquier otro formato de texto.

#### 1.12.4 Representación de punto fijo

Muchas veces, representar números decimales o binarios implica trabajar con partes fraccionarias. En general, la gran mayoría de aplicaciones requieren este tipo de representación. Al inicio de este capítulo, revisamos la representación de números enteros positivos y negativos. Sin embargo, es importante mencionar que existe la posibilidad de representar números con parte fraccionaria, de una forma relativamente sencilla. Esto, a su vez, acarrea ciertos inconvenientes, como la exactitud de la representación. Por esta razón, se debe seleccionar adecuadamente el tipo de aplicación en la que se pueda utilizar este tipo de representación binaria.

Para representar un número binario en punto fijo, es necesario sacrificar bits para la parte entera y estos asignarlos para la parte fraccionaria. Esto hará que el número entero tenga un rango menor de representación, pero con la característica que podrá contar con algunos bits para la parte fraccionaria. Por ejemplo, si se desea representar un número con 8 bits enteros y 8 bits decimales, el número tendrá que estar almacenado en una variable de 16 bits. De manera general, a un número binario X se le puede asignar  $n$  dígitos binarios para la parte

entera y  $m$  dígitos binarios para la parte decimal, quedando de este modo:  $X(n,m)$ .

Hay que señalar que no todos los números se pueden representar de manera exacta. La resolución dependerá del número de bits en la parte fraccionaria. Por ejemplo, tenemos el número binario 11,01 cuya resolución es de 0,25. Esto se debe a que los exponentes del sistema binario a la derecha de la coma llevan signo negativo, similar a lo que tenemos en el sistema decimal.

$$1 * 2^1 + 1 * 2^0 + 0 * 2^{-1} + 1 + 2^{-2}$$

Generalizando, podríamos decir que la resolución de una representación en punto fijo está dada por  $2^{-m}$ .

**Ejercicio 1.22:** Calcular la resolución de un número binario  $X(6,10)$ .

#### Solución:

Como hemos descrito, la resolución viene dada por el número de bits de la parte decimal o fraccionaria. En este caso, tenemos 10 bits asignados para la parte fraccionaria, por lo que su resolución será  $2^{-10}$ , o lo que es lo mismo 0,0009765625.

Por el contrario, a pesar de que el número es de 16 bits apenas 6 de ellos están destinados para representar la parte entera. En este caso, el mínimo valor entero será 0 y el máximo valor entero será 63.

**Ejercicio 1.23:** Representar el número decimal 17,03478 en formato  $X(5,11)$ .

#### Solución:

En primer lugar, la parte entera representada en binario es: 10001. Para calcular la parte fraccionaria, multiplicamos el factor 0,03478 por dos de forma sucesiva, como se presenta a continuación. Cuando el resultado supere 1, restaremos de 1 y nuevamente empezaremos con la multiplicación sucesiva.

0,03478	x	2	=	0,06956	Primer dígito fraccionario
0,06956	x	2	=	0,13912	Segundo dígito fraccionario
0,13912	x	2	=	0,27824	Tercer dígito fraccionario
0,27824	x	2	=	0,55648	Cuarto dígito fraccionario
0,55648	x	2	=	1,11296	Quinto dígito fraccionario
0,11296	x	2	=	0,22592	Sexto dígito fraccionario
0,22592	x	2	=	0,45184	Séptimo dígito fraccionario
0,45184	x	2	=	0,90368	Octavo dígito fraccionario
0,90368	x	2	=	1,80736	Noveno dígito fraccionario
0,80736	x	2	=	1,61472	Décimo dígito fraccionario
0,61472	x	2	=	1,22944	Décimo primer dígito fraccionario

El número 17,03478 representado en formato  $X(5,11)$  sería: 10001,00001000111 con una resolución de 0,00048828125.



# Capítulo 2

## Introducción a los microcontroladores

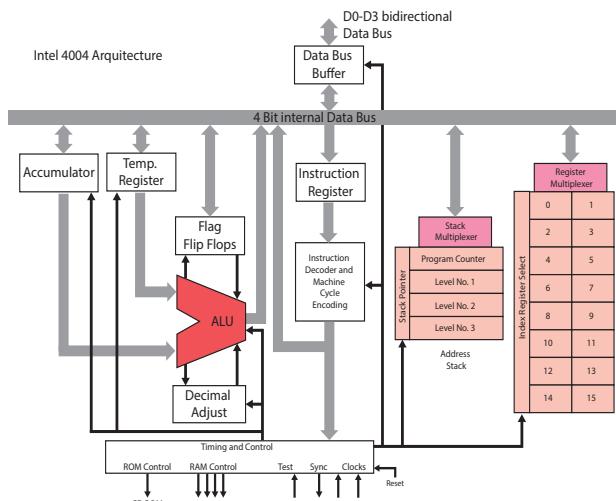
### 2.1 Anotaciones preliminares sobre microcontroladores

A lo largo del tiempo, los diseños de circuitos electrónicos han evolucionado. Con la existencia de la lógica cableada, mediante el uso de transistores, diodos, condensadores, entre otros, los cálculos se volvieron más complejos y generaron demasiados ajustes y fallas. A partir de la aparición del microprocesador y la lógica programable, se impuso una nueva tecnología que permitió modificar su comportamiento lógico digital, sin tener que cambiar su configuración física.

A inicios de la década de los setenta, apareció el primer microprocesador, creado por la compañía Intel. Este fue el resultado de la evolución de distintas tecnologías predecesoras. El chip Intel 4004, con una CPU de 4 bits, 2300 transistores, 16 pines y una velocidad de reloj 740 KHz (ver figura 2.1), podía generar hasta 60 000 operaciones por segundo, lo que produjo un cambio radical en el diseño de la mayoría de los equipos. En 1973, Texas Instruments (TI) obtuvo la primera patente de un microprocesador en un solo chip, pero tanto Intel como TI comparten el mérito por la invención casi simultánea del microprocesador (Huang, 1996). La aparición de este circuito integrado facilitó el empleo y la creación de diseños más pequeños y simplificados, que realizan una mayor cantidad de tareas en menos tiempo; sin embargo, estos sistemas requieren circuitos adicionales para implementar un sistema de trabajo completo (Cass, 2018).

Figura 2.1

Arquitectura chip Intel 4004



Tomado de Velasco, 2011

De la evolución de los microprocesadores y las necesidades de control de distintos dispositivos industriales y comerciales, aparece la idea tecnológica de incorporar estas funcionalidades en un solo *chip* a la estructura básica de un sistema de cómputo, lo que dio lugar al microcontrolador. Por tanto, un microcontrolador (UC, MCU o  $\mu$ C) se define como un circuito integrado programable, capaz de ejecutar las órdenes grabadas en su memoria. Está compuesto por tres unidades principales (CPU: unidad central de procesamiento, memoria y periféricos de E/S: entrada y salida, I/O).

Uriel y González (2009, p. 295) especifican que el microcontrolador es un circuito integrado, compacto y de arquitectura cerrada, que realiza procesos lógicos programables por medio de un lenguaje ensamblador del usuario. Señalan, además, que este incluye todos los componentes de un computador: CPU, memoria y unidades I/O, que son aplicadas a situaciones específicas de control, capaces de incorporar unidades adicionales e interactuar con el medio, comportándose como sistemas abiertos.<sup>3</sup> En la actualidad, el microcontrolador está presente en la mayoría de los artefactos que se usan a diario en la industria textil, alimenticia, automotriz, de electrodomésticos, informática, etcétera. Gracias a esta tecnología, se puede integrar inteligencia a casi cualquier artefacto. En áreas como la robótica, se utiliza para el control de motores y captura de señales de los diferentes sensores, la fabricación de controladores para sistemas automáticos, entre otros. En el campo de la electrónica, se pueden encontrar en los tacómetros digitales, controladores de display LCD, decodificadores de TV, analizadores de espectros, fotocopiadoras, teléfonos celulares, cerraduras electrónicas, sistemas de seguridad.

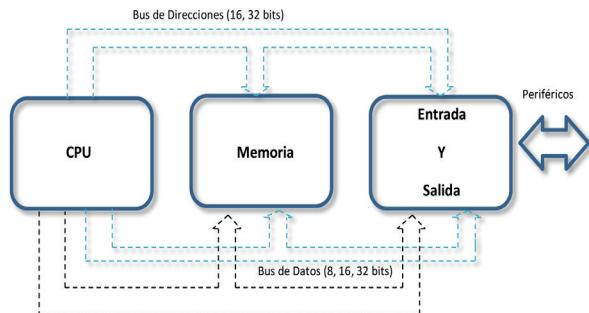
Los microprocesadores y microcontroladores disponen de un *hardware* bien diferenciado. Los microprocesadores han evolucionado para manejar grandes volúmenes de datos; suelen encontrarse en los computadores, como parte primordial para su funcionamiento. En cambio, los microcontroladores incorporan unidades funcionales que interactúan con sensores para la integración con el medio físico en tiempo real; además, proporcionan un mejor desempeño y robustez en aplicaciones industriales. Es muy común encontrarlos para controlar periféricos de las computadoras.

## 2.2 Estructura del microcontrolador

Los microcontroladores son como pequeñas computadoras, que se utilizan para aplicaciones puntuales, como la domótica, IoT, medicina, electrodomésticos. En su arquitectura se distinguen tres partes principales, que se aprecian en la figura 2.2.

**Figura 2.2**

*Esquema básico de un microcontrolador*



Tomado de Madruga, 2016

- ✓ CPU es la unidad central de procesamiento, para controlar y ejecutar el programa almacenado en la memoria. Los microcontroladores, generalmente, se basan en el núcleo del microprocesador, como el Intel 8080, Motorola 6800, entre otros.
- ✓ Memoria *ROM* y *RAM*: los microcontroladores disponen de bloques de memoria en donde se alojan los programas, datos y registros necesarios para implementar los procesos. Dispone de dos tipos:
  - ROM es aquella que almacena el código de máquina. Puede contar con uno cuantos kB (memoria de programa).
  - RAM es aquella memoria de almacenamiento temporal, denominada memoria de datos.
  - EEPROM es un tipo de memoria ROM que puede ser programada, borrada y reprogramada eléctricamente, a diferencia de la EPROM, que ha de borrarse mediante un aparato que emite rayos ultravioleta.
- ✓ Los puertos E/S son pines del microcontrolador, destinados para comunicar al dispositivo con el entorno real. De ellos depende la conexión con variables como temperatura, humedad ambiental etcétera. Pueden tener varias funciones, dependiendo de los fabricantes.

<sup>3</sup> Sistema que tiene interacciones externas, que pueden tomar forma de información, energía o materia.

## 2.3 Clasificación

Los microcontroladores, al igual que los microprocesadores, se clasifican por una cadena finita de bits, desde los 4 bits, 8 bits, 16 bits y, con mayores prestaciones, hasta de 32 bits. Los bits identifican la capacidad de operaciones que pueden existir en un microcontrolador. Pero no solo existe la clasificación por bits; en realidad, depende de cada fabricante, ya que incorporan unidades con diferentes funcionalidades, como temporizadores (*timer*), conversores análogos a digitales (ADC), puertos de comunicación (*I2C*, *USART*), puerto serial síncrono (SSP), entre otros. Estas unidades se incorporan con diferentes combinaciones, según las gamas; llegan a tener varios pines DIP (*Dual In-line Package*), que pueden ir desde 8 a 40.

En el mercado actual, aún existen dispositivos que constan de 4 y 8 bits, ya que no tiene mucho sentido disponer de un microcontrolador de altas prestaciones 16 o 32 bits para operaciones sencillas y que resulten más económicas al momento de su implementación. En realidad, los microcontroladores de 8 bits cubren gran parte del mercado actual, gracias a su bajo costo.

En la imagen 2.1 se pueden observar algunos ejemplos de la clasificación de los microcontroladores, con sus respectivos fabricantes.

Imagen 2.1

Clasificación de microcontroladores



a) Microchip 32 bits. Tomado de Microchip Technology, 2020d); b) 32 bits STM32 series. Tomado de directindustry, 2020; c) AVR-Atmel 32 bits. Tomado de Microchip Technology, 2020a)

## 2.4 Familias de microcontroladores

Existe una variedad de empresas que han enfocado su estudio en el desarrollo de microcontroladores de 4, 8 16 bits; entre ellas, Texas Instruments, Dallas, ARM, Microchip, Analog Devices, Holtek, Intel, Motorola, Atmel AVR, Zilog, Nacional Semiconductor. La imagen 2.2 muestra algunas de las marcas más reconocidas que, actualmente, están en el mercado. En este texto, enfatizaremos en dos empresas: Microchip PIC y Atmel AVR, familias reconocidas en el mundo de la programación de microcontroladores.

Imagen 2.2

Familias de microcontroladores



La empresa Microchip dispone de la familia PIC, en diferentes gamas. Las características de cada tipo de gama se relacionan con el número de instrucciones, bits, puertos, complejidad del circuito integrado, entre otros. La figura 2.3 presenta algunas de esas características.

**Figura 2.3**

Gamas de microcontroladores tipo PIC

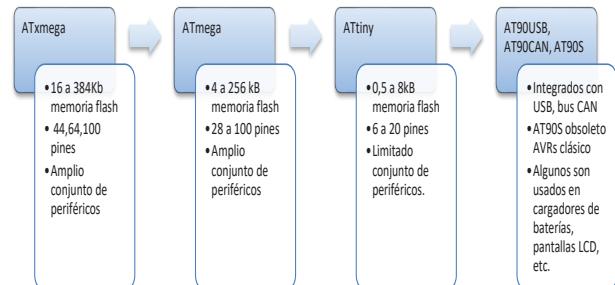


Además de las gamas mencionadas, la empresa Microchip ofrece una línea de microcontroladores de 32 bits. Estas poseen varias características innovadoras, como múltiples dominios de potencia, niveles de rendimiento, periféricos. Entre sus aplicaciones, están industriales, médicas, IoT, nodos LPWAN como LoRa y monitores portátiles de EKG.

La empresa ATMEL maneja microcontroladores basados en arquitectura RISC<sup>4</sup>. Las CPU pueden llegar hasta 32 bits. Existen varios grupos; por ejemplo, el 8051 Intel, que incorpora una memoria flash. También posee la familia de microcontroladores AVR. La figura 2.4 permite apreciar las diferentes gamas de AVR.

**Figura 2.4**

Gama microcontroladores AVR



## 2.5 Aplicaciones de microcontroladores

Debido a la gran cantidad de familias y gamas de microcontroladores, estos se han utilizado en dife-

rentes aplicaciones y es muy común encontrarlos en dispositivos que ocupamos a diario. Se venden más microcontroladores que microprocesadores, pese a que estos últimos siguen evolucionando a 32 y 64 bits, con múltiples núcleos. No obstante, los microcontroladores de 4 y 8 bits no dejan de existir, sobre todo, por sus aplicaciones (en especial, en educación) y su bajo costo.

Uno de los sectores en el que tienen mayor presencia es el automovilístico, debido a que enfrentan varios retos en condiciones extremas, como vibración, ruido, choques. Pero también se pueden encontrar en sectores de telecomunicaciones, informático, de instrumentación industrial. La figura 2.5 presenta algunas aplicaciones de los microcontroladores.

**Figura 2.5**

Aplicación microcontroladores



## 2.6 Placas de desarrollo comercial

En la actualidad, varias empresas dedicadas al desarrollo de placas comerciales trabajan con las diferentes familias de microcontroladores. A continuación, presentamos una descripción de placas de desarrollo en las que se señalan algunas de sus características.

### 2.6.1 Samsung ARTIK 710

La familia ARTIK de módulos IoT de Samsung fue desarrollada para una gama completa de aplica-

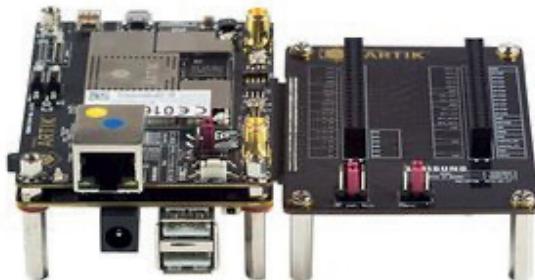
<sup>4</sup> RISC, Reduced Instruction Set Computer. es un procesador RISC cuando la misma instrucción que carga datos de memoria no realiza operaciones sobre ellos.

ciones de *IoT*, desde accesorios personales hasta dispositivos de integración. Con la combinación de dispositivos de comunicación, como *bluetooth* y *wifi*, esta placa (ver imagen 2.3) permite la integración de sensores y apoya con una capacidad de diseño de usuario *UI/UX*, con opciones de cámara y pantalla (Digi-Key Electronics, 2019). La placa posee un microcontrolador de la familia ARM Cortex A53. Además, incorpora:

- GPU: Mali T400 y acelerador de gráficos 3D
- Memoria flash de 4 GB
- Cámara I/F: MIPI CSI de 4 carriles
- WLAN
- Bluetooth

### Imagen 2.3

*Samsung Artika 710*



Tomado de Digi-Key Electronics, 2019

El objetivo final de la placa de desarrollo ARTIK es combinar módulos de *hardware* y servicios en la nube. Samsung ha creado un ecosistema de herramientas y socios para ayudar a los desarrolladores a diseñar con mayor rapidez aplicaciones para Internet de las cosas.

### 2.6.2 SensorTile

Es un pequeño módulo de *IoT* que incluye capacidades de procesamiento basado en el microcontrolador STM32L476JGY de 80 MHz (ver imagen 2.4). Este dispositivo puede incorporar módulos de sensores, módulo de comunicación, entre otros (Zephyr Project, 2019). Entre las características principales, la placa de desarrollo SensorTile cuenta con:

- Placa de expansión equipada con DAC de audio, USB, STM32, Arduino UNO, Conector SWD
- Cargador de batería, sensor de humedad y temperatura
- Ranura de memoria SD
- Batería de litio 100 mAh
- Cable de programación

### Imagen 2.4

*Placa SensorTile*



Tomado de Zephyr Project, 2019

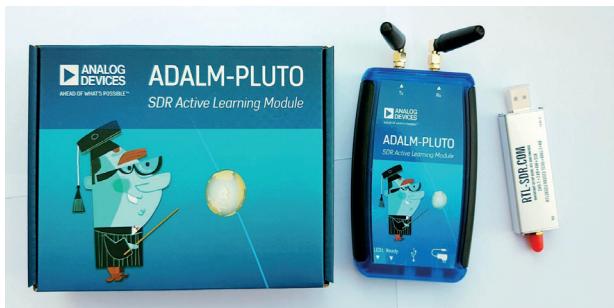
### 2.6.3 Adalm-Pluto

Es un módulo orientado al aprendizaje nivel de ingeniería. Enseña los fundamentos de radio, definido por *software* (SDR), la radiofrecuencia (RF) y las comunicaciones inalámbricas (ver imagen 2.5). Esta placa puede usarse con programas de *software* como MATLAB y Simulink, para trabajar de una forma más práctica (Analog Devices, 2019). Entre las características más importantes de esta placa, se encuentran:

- Módulo de aprendizaje de RF autónomo portátil
- Cobertura de RF de 325 MHz a 3.8 GHz
- ADC de 12 bits
- Soporte MATLAB
- Interfaz USB

### Imagen 2.5

Módulo Adalm-Pluto



Tomado de Analog Devices, 2019

### 2.6.4 AudioSmart 2-Mic

La tarjeta de desarrollo AudioSmart (ver imagen 2.6) fue creada para propósitos domésticos inteligentes, orientados al *IoT*. Incorpora un *software*, el asistente de voz ALEXA, que permite controlar los sistemas del microcontrolador por medio de mensajes auditivos. La tarjeta incluye unos micrófonos para la integración con sistemas y dispositivos en el hogar (Synaptics Incorporated, 2019). Entre sus principales características, se encuentran:

- Micrófonos estéreo con cable integrado
- Tablero de soporte de micrófono
- Cable micro USB
- Fuente de alimentación

### Imagen 2.6

Placa AudioSmart



Tomado de Synaptics Incorporated, 2019

### 2.6.5 Thunderboard sense

Orientada al *IoT*, es una placa de desarrollo que permite conectar varios sensores (ver imagen 2.7). Dispone de una aplicación móvil que posibilita ver los sensores conectados; controlar salidas; transmitir datos de los sensores conectados a una base de datos, denominada Firebase, lo que facilita la visualización de *dashboard* (Synaptics Incorporated, 2019). Entre las principales características de esta placa, se encuentran:

- SoC inalámbrico 2.4 GHz
- Microcontrolador ARM Cortex M4 con 32 kB de RAM
- Interfaces periféricas MCU flexibles
- Puerto serie virtual
- Depurador integrado

### Imagen 2.7

Placa Thunderboard Sense



Tomado de Synaptics Incorporated, 2019

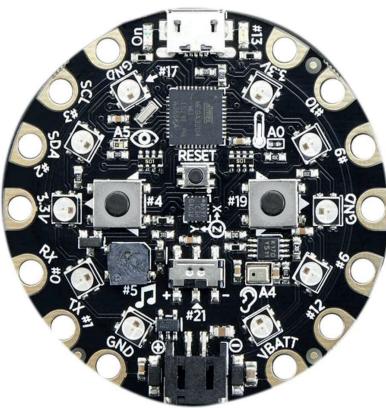
### 2.6.6 Flora AdaFruit

Flora es una placa de desarrollo portátil compatible con los sensores de la empresa Adafruit (ver imagen 2.8). Es redondo y cosible, para colocarlo en prendas de vestir; es compatible con la plataforma de programación Arduino IDE (Adafruit, 2019); dispone de un USB incorporado. Entre sus principales características, se encuentran:

- Compatible con Arduino IDE
- Comunicación serial de datos
- Microcontrolador ATmega32u4 a 8 MHz
- Puerto para batería LiPo

## Imagen 2.8

Placa Flora AdaFruit



Tomado de Adafruit, 2019

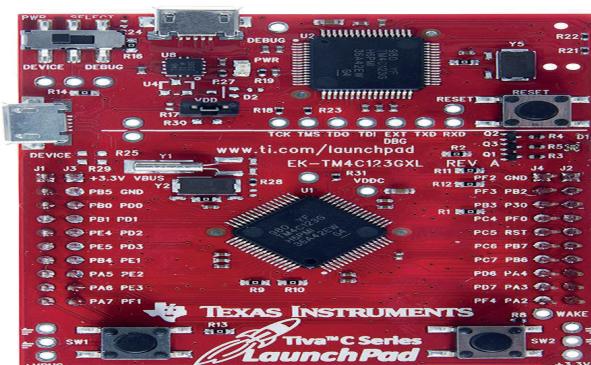
## 2.6.7 LaunchPad (MSP-EXP430G2ET)

Desarrollada por la empresa Texas Instruments (ver Imagen 2.9), cuenta con un zócalo DIP de 14/20 pines en los cuales se puede incorporar sensores o una pantalla; dispone de conexión USB para cargar programas (Texas Instruments Incorporated, 2019). Entre sus principales características están:

- Microcontrolador MSP430G2425
- Cristal de 32 KHz
- Conector PCB de 10 pines
- Conector mini USB
- Bajo consumo de energía

## Imagen 2.9

Placa LaunchPad



Tomado de Texas Instruments Incorporated, 2019

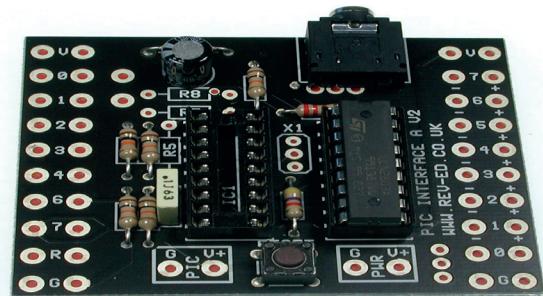
## 2.6.8 PICAXE 18X

Picaxe es un sistema de microcontroladores de nivel básico, diseñado para aplicaciones educativas y soporte para sistemas embebidos (ver imagen 2.10). Este sistema emula las placas de Arduino y es compatible con todas las *shields* de esta empresa (Martin, 2012). La placa contiene las siguientes características:

- Programación USB
- 2 conectores para servos
- Compatible con el pinout de Arduino
- Microcontrolador PICAXE

## Imagen 2.10

Placa PICAXE



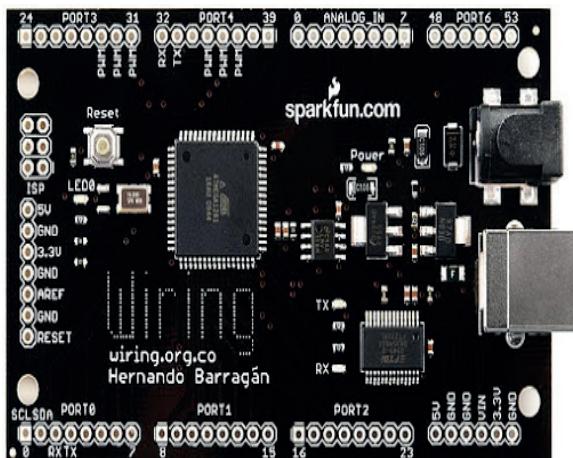
Tomado de Martín, 2012

## 2.6.9 Wiring

El entorno Wiring es una plataforma de prototipado electrónico de fuente abierta, que se complementa con un lenguaje de programación. Es compatible con el microcontrolador AVR de Atmel (Barragán, 2019) luz, distancia a un objeto, etc.. Una de sus últimas placas es la Wiring S (ver imagen 2.11), que dispone de puertos libres para colocar *shields* de Arduino y conectar casi todos los sensores de esta empresa.

### Imagen 2.11

Placa Wiring S.



Tomado de Barragán, 2019

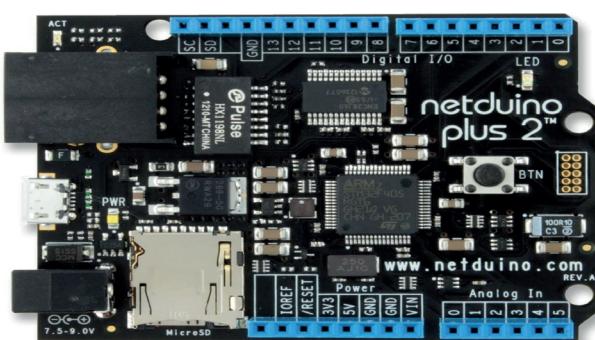
Wiring dispone de una gama de placas de desarrollo que se pueden acoplar a cualquier aplicación tanto *IoT*, como proyectos experimentales; conecta con ellos diferentes sensores, actuadores, etcétera.

### 2.6.10 Netduino

Netduino es una plataforma de desarrollo similar a Arduino (ver imagen 2.12). Es compatible con las *shields* existentes. Dispone de una gama extensa de placas para diferentes aplicaciones, basadas en microcontroladores de 8 a 16 bits. Es compatible con el lenguaje de programación C# c Sharp (Netduino.foundation, 2019).

### Imagen 2.12

Placa Netduino



Tomado de Netduino.foundation, 2019

### 2.6.11 Arduino

Arduino es una plataforma electrónica de código abierto, basada en *hardware* y *software* orientados para varias áreas técnicas, profesionales, experimentales e *IoT* (ver imagen 2.13). Las placas Arduino, mediante sensores, pueden leer en sus entradas (luz, calidad de aire, temperatura, etcétera) y pueden interpretar en acciones con sus salidas, como actuadores, encendido de led, transmisión de datos, entre otros.

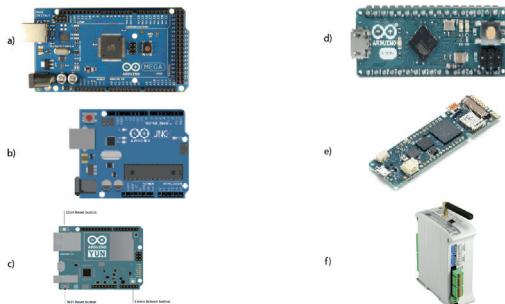
Arduino es utilizado en miles de proyectos, desde objetos cotidianos hasta instrumentos científicos complejos y tiene el apoyo de una comunidad mundial (estudiantes, aficionados, artistas, programadores y profesionales). Estas contribuciones han sido de gran utilidad para mejorar las aplicaciones tanto en *hardware* como en *software* (Arduino, 2020b).

Arduino ofrece al usuario una gama amplia de tarjetas basadas en microcontroladores ATmega, que se caracterizan por:

- ser económicas en comparación con otras plataformas de microcontroladores.
- ser multiplataforma; no solo soportan el Arduino IDE si no que son compatibles con otras plataformas.
- su *hardware* es de código abierto y extensible; trabajan bajo la licencia de Creative Commons, por lo que todas las personas y diseñadores de circuitos experimentales pueden realizar su propia versión y seguir mejorándolo.
- Arduino permite una gama amplia de *shields*, que son placas electrónicas que complementan las aplicaciones del microcontrolador. En estas tarjetas se pueden encontrar matriz de ledes, sensores de calidad de aire, expansores para manejo de motores, servomotores, entre otras.

**Imagen 2.13**

Placas comerciales Arduino



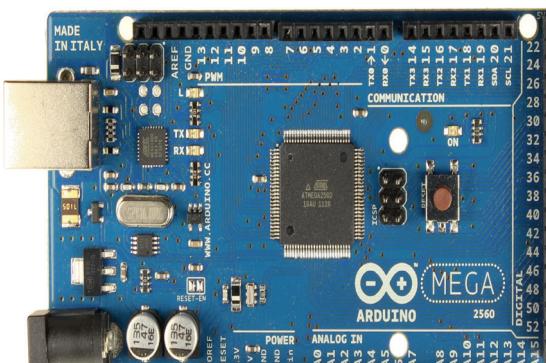
a) Arduino Mega 2560 b) Arduino UNO c) Arduino Yun Rev2  
d) Arduino Micro e) Arduino MKR Vidor 4000 f) PLC Arduino ArdBox PLC 20. Tomado de Arduino, 2020a

### 2.6.12 Arduino mega

La placa de desarrollo Mega 2560 está basada en el microcontrolador ATmega2560 de 8 bits (ver imagen 2.14). Está diseñada para proyectos más complejos. Esta tarjeta cuenta con 54 pines de E/S digitales, 16 entradas analógicas, 4 puertos UART (puertos serie de *hardware*), un oscilador de cristal de 16 MHz, una conexión USB, un conector de alimentación, un conector o ICSP y un botón de reinicio.

**Imagen 2.14**

Arduino Mega 2560



Tomado de Arduino, 2018

Las especificaciones técnicas se pueden apreciar en la tabla 2.1.

**Tabla 2.1**

Especificaciones técnicas Arduino Mega 2560

Microcontrolador	ATmega2560
Tensión de funcionamiento	5 V
Voltaje de entrada recomendado	7-12 V
Voltaje de entrada límite	6-20 V
Pines de E/S digitales	54 (15 con salida PWM)
Pines de entrada analógica	16 ADC
Corriente CC por E/S pin	20 mA
Corriente CC para pin de 3.3 V	50 mA
Memoria flash	256 kB (8 kB usados por el gestor de arranque)
SRAM	8 kB
EEPROM	4 kB
Velocidad de reloj	16 MHz
LED_BUILTIN	Pin 13
Longitud	101.52 mm
Anchura	53.3 mm
Peso	37 g

Tomado de Arduino, 2018

El Mega 2560 dispone de un fusible múltiple que protege los puertos USB de la computadora contra cortocircuitos y sobre corrientes, aunque las PC disponen de su propia protección interna. El fusible proporciona una protección adicional: si se aplica más de 500 mA al puerto USB, el fusible interrumpe automáticamente la conexión hasta que se elimine la sobrecarga. La alimentación externa (no USB) puede provenir de un adaptador de CA a CC; si es CA, se debe colocar en el conector 2.1 mm; si es CC, puede conectarse en el pin GND y VIN. La alimentación externa debe estar en un rango de 7 a 12 voltios.

#### 2.6.12.1 Alimentación

Los pines de alimentación son:

- VIN: el voltaje de entrada de la placa cuando se usa una fuente de alimentación externa.

- 5 V: este pin emite 5 V regulados desde la placa.
  - 3 V3: suministro 3,3 voltios generado por la placa la corriente máxima es de 50 mA.
  - GND: dos pines de tierra.
  - IOREF: proporciona la referencia de voltaje con la que funciona el microcontrolador. Un circuito integrado está configurado para leer la tensión del pin IOREF y selecciona la fuente de alimentación adecuada o habilita las salidas de 5 V o 3.3 V.

### 2.6.12.2 Memoria

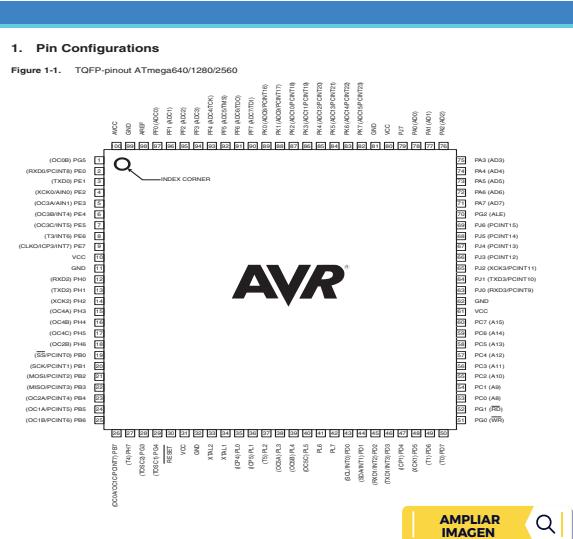
El ATmega2560 tiene 256 kB de memoria *flash* para almacenar códigos (de los cuales 8 kB se utilizan para el cargador de arranque), 8 kB de SRAM y 4 kB de EEPROM.

### 2.6.12.3 Pines E/S

Cada uno de los 54 pines digitales en el Mega se pueden usar como entrada o salida, y operan a 5 voltios. Cada pin puede proporcionar o recibir 20 mA como corriente recomendada y tiene una resistencia de *pull-up* interna (desconectada por defecto) de 20-50 k $\Omega$ . En la imagen 2.15 se pueden observar los pines del microcontrolador ATmega2560.

Imagen 2.15

## *Mega Pinout*

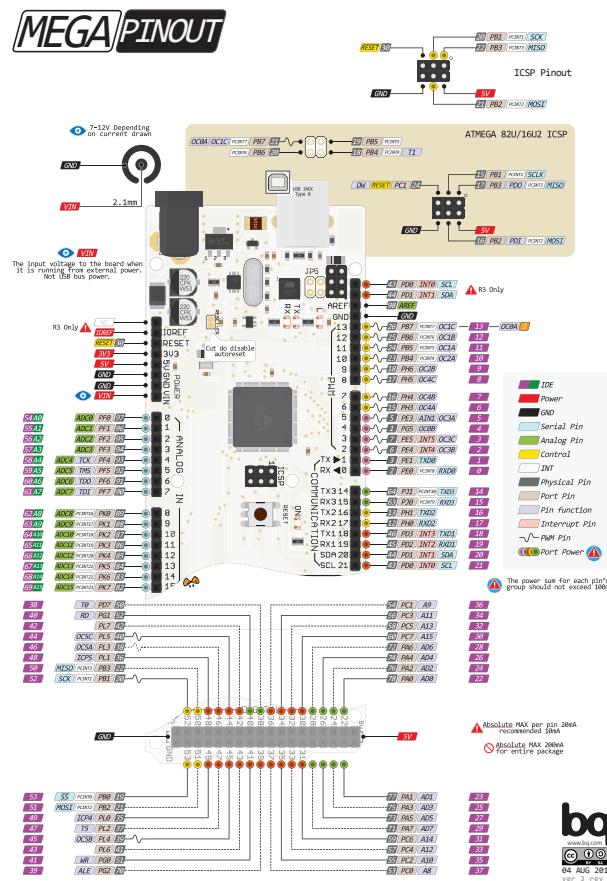


Para que se pueda apreciar de mejor manera, en la imagen 2.16, se dispone del *pinout* del Arduino Mega2560. Este diagrama proporciona la información de todos los pines y conectores que dispone esta placa.

## Imagen 2.16

**AMPLIA  
IMAGEN**

## *Arduino Mega pinout*



Tomado de Arduino, 2018

La tabla 2.2 contiene una descripción general de los pines, en la que se evidencia la relación de los pines del microcontrolador con los pines de la placa comercial Arduino Mega2560.

**Tabla 2.2***Descripción general de los pines*

Número de pin	Nombre pin	Nombre del pin asignado
1	PG5 (OC0B)	Pin digital 4 (PWM)
2	PE0 (RXD0/PCINT8)	Pin digital 0 (RX0)
3	PE1 (TXD0)	Pin digital 1 (TX0)
4	PE2 (XCK0/AIN0)	
5	PE3 (OC3A/AIN1)	Pin digital 5 (PWM)
6	PE4 (OC3B/INT4)	Pin digital 2 (PWM)
7	PE5 (OC3C/INT5)	Pin digital 3 (PWM)
8	PE6 (T3/INT6)	
9	PE7 (CLKO/ICP3/INT7)	
10	VCC	VCC
11	GND	GND
12	PH0 (RXD2)	Pin digital 17 (RX2)
13	PH1 (TXD2)	Pin digital 16 (TX2)
14	PH2 (XCK2)	
15	PH3 (OC4A)	Pin digital 6 (PWM)
16	PH4 (OC4B)	Pin digital 7 (PWM)
17	PH5 (OC4C)	Pin digital 8 (PWM)
18	PH6 (OC2B)	Pin digital 9 (PWM)
19	PB0 (SS/PCINT0)	Pin digital 53 (SS)
20	PB1 (SCK/PCINT1)	Pin digital 52 (SCK)
21	PB2 (MOSI/PCINT2)	Pin digital 51 (MOSI)
22	PB3 (MISO/PCINT3)	Pin digital 50 (MISO)
23	PB4 (OC2A/PCINT4)	Pin digital 10 (PWM)
24	PB5 (OC1A/PCINT5)	Pin digital 11 (PWM)
25	PB6 (OC1B/PCINT6)	Pin digital 12 (PWM)
26	PB7 (OC0A/OC1C/PCINT7)	Pin digital 13 (PWM)
27	PH7 (T4)	
28	PG3 (TOSC2)	
29	PG4 (TOSC1)	

30	Reiniciar	Reiniciar
31	VCC	VCC
32	GND	GND
33	XTAL2	XTAL2
34	XTAL1	XTAL1
35	PL0 (ICP4)	Pin digital 49
36	PL1 (ICP5)	Pin digital 48
37	PL2 (T5)	Pin digital 47
38	PL3 (OC5A)	Pin digital 46 (PWM)
39	PL4 (OC5B)	Pin digital 45 (PWM)
40	PL5 (OC5C)	Pin digital 44 (PWM)
41	PL6	Pin digital 43
42	PL7	Pin digital 42
43	PD0 (SCL/INT0)	Pin digital 21 (SCL)
44	PD1 (SDA/INT1)	Pin digital 20 (SDA)
45	PD2 (RXDI/INT2)	Pin digital 19 (RX1)
46	PD3 (TXD1/INT3)	Pin digital 18 (TX1)
47	PD4 (ICP1)	
48	PD5 (XCK1)	
49	PD6 (T1)	
50	PD7 (T0)	Pin digital 38
51	PG0 (WR)	Pin digital 41
52	PG1 (RD)	Pin digital 40
53	PC0 (A8)	Pin digital 37
54	PC1 (A9)	Pin digital 36
55	PC2 (A10)	Pin digital 35
56	PC3 (A11)	Pin digital 34
57	PC4 (A12)	Pin digital 33
58	PC5 (A13)	Pin digital 32
59	PC6 (A14)	Pin digital 31
60	PC7 (A15)	Pin digital 30
61	VCC	VCC
62	GND	GND
63	PJ0 (RXD3/PCINT9)	Pin digital 15 (RX3)
64	PJ1 (TXD3/PCINT10)	Pin digital 14 (TX3)
65	PJ2 (XCK3/PCINT11)	
66	PJ3 (PCINT12)	

67	PJ4 (PCINT13)	
68	PJ5 (PCINT14)	
69	PJ6 (PCINT15)	
70	PG2 (ALE)	Pin digital 39
71	PA7 (AD7)	Pin digital 29
72	PA6 (AD6)	Pin digital 28
73	PA5 (AD5)	Pin digital 27
74	PA4 (AD4)	Pin digital 26
75	PA3 (AD3)	Pin digital 25
76	PA2 (AD2)	Pin digital 24
77	PA1 (AD1)	Pin digital 23
78	PA0 (AD0)	Pin digital 22
79	PJ7	
80	VCC	VCC
81	GND	GND
82	PK7 (ADC15/ PCINT23)	Pin analógico 15
83	PK6 (ADC14/ PCINT22)	Pin analógico 14
84	PK5 (ADC13/ PCINT21)	Pin analógico 13
85	PK4 (ADC12/ PCINT20)	Pin analógico 12
86	PK3 (ADC11/ PCINT19)	Pin analógico 11
87	PK2 (ADC10/ PCINT18)	Pin analógico 10
88	PK1 (ADC9/ PCINT17)	Pin analógico 9
89	PK0 (ADC8/ PCINT16)	Pin analógico 8
90	PF7 (ADC7/TDI)	Pin analógico 7
91	PF6 (ADC6/TDO)	Pin analógico 6
92	PF5 (ADC5/TMS)	Pin analógico 5
93	PF4 (ADC4/TCK)	Pin analógico 4
94	PF3 (ADC3)	Pin analógico 3
95	PF2 (ADC2)	Pin analógico 2
96	PF1 (ADC1)	Pin analógico 1
97	PF0 (ADC0)	Pin analógico 0
98	AREF	Referencia analógica

99	GND	GND
100	VCC	VCC

Tomado de Arduino, 2018

#### 2.6.12.4 Comunicaciones

La placa Mega 2560 tiene varias opciones para comunicarse con una computadora u otros microcontroladores. El ATmega2560 proporciona cuatro *UART* de *hardware* para comunicación serie TTL (5 V). Un ATmega16U2 canaliza esta comunicación a través del puerto USB. Las led Rx y Tx de la placa parpadean cuando los datos se transmiten a través del *chip*. La placa también es compatible con la comunicación TWI y SPI.

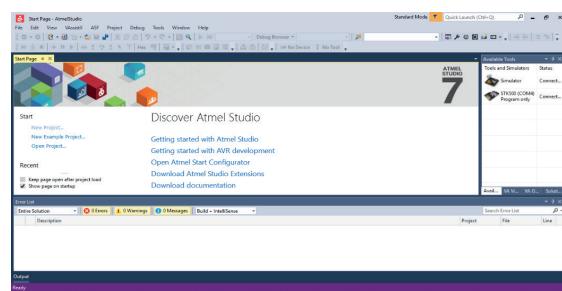
#### 2.7 Software AVR ATMEL Studio

Atmel Studio es la plataforma de desarrollo integrado (IDP) para desarrollar y depurar todas las aplicaciones de microcontroladores AVR y SAM (ver imagen 2.17). El IDP de Atmel Studio 7 brinda un entorno transparente y fácil de usar para escribir, compilar y depurar las aplicaciones escritas en C / C ++ o código ensamblador. También se conecta sin problemas a los depuradores, programadores y kits de desarrollo que admiten dispositivos AVR y SAM.

Además, Atmel Studio incluye Atmel Gallery, una tienda de aplicaciones en línea que le permite ampliar su entorno de desarrollo con complementos desarrollados por Microchip, así como proveedores de herramientas integradas y de terceros. Atmel Studio 7 también puede importar sin problemas sus bocetos de Arduino como proyectos de C ++ , lo que proporciona una ruta de transición simple desde Makerspace a Marketplace (Microchip Technology, 2020b).

Imagen 2.17 AMPLIAR IMAGEN Q

Software Atmel Studio



Tomado de Microchip Technology, 2020b

Entre los aspectos más importantes de usar Atmel Studio, se pueden mencionar los siguientes:

- soporte para más de 300 dispositivos basados en AVR y SMART ARM.
  - contiene más de 1 600 ejemplos de proyectos con código fuente, servicios gráficos, funcionalidades a través de Atmel Software Framework (ASF).
  - escribir y depurar código escrito en lenguaje C/C++.
  - editor integrado con asistencia visual.

### 2.7.1 Mi primer programa

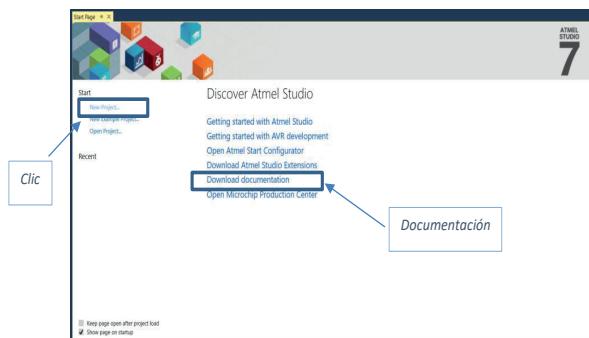
A continuación, presentamos los pasos más representativos para iniciar con el desarrollo de programas y su posterior implementación en el microcontrolador. Mostramos un ejemplo base de código; sin embargo, la explicación puntual del funcionamiento del microcontrolador y sus diversas funcionalidades detallamos más adelante.

Una vez realizada la instalación de Atmel Studio y ejecutado el programa, observaremos una ventana (imagen 2.18). En esta ventana principal también se puede encontrar documentación importante sobre Atmel.

Imagen 2.18



## Ventana principal Atmel Studio 7



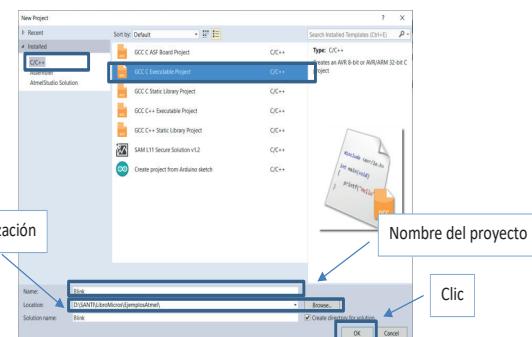
Tomado de Microchip Technology, 2020b

En la siguiente ventana (ver imagen 2.19), se puede escoger el nombre del proyecto, la localización, y se debe seleccionar la opción de generar un código ejecutable en C/C++. Una vez configurados estos parámetros, se puede dar clic en OK.

### Imagen 2.19



## *Configuración nuevo proyecto*



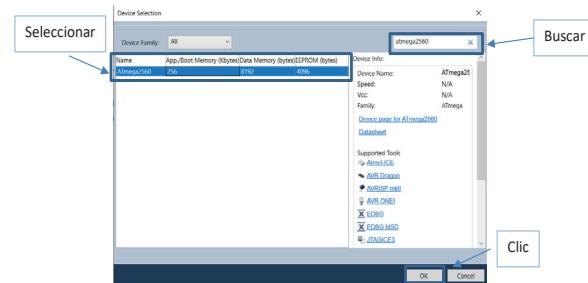
Tomado de Microchip Technology, 2020b

En la siguiente ventana (ver imagen 2.20), se debe buscar el dispositivo (microcontrolador), seleccionarlo y dar clic en OK.

Imagen 2.20



## *Selección de microcontrolador*



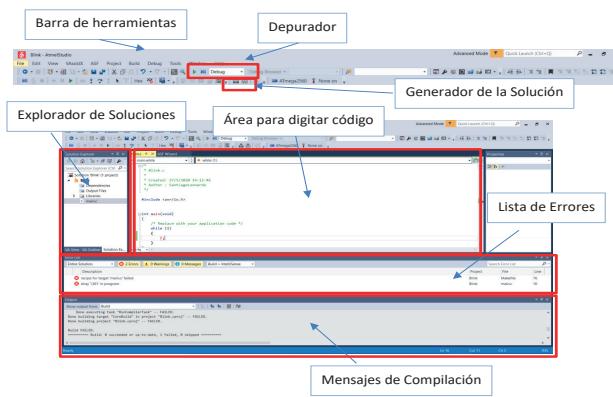
Tomado de Microchip Technology, 2020b

Si ha realizado de manera correcta los pasos mencionados anteriormente, aparecerá una pantalla similar a la de la imagen 2.21. En esta, se presentan los nombres de las áreas de trabajo para poder generar su código de programación.

Imagen 2.21

AMPLIAR  
IMAGEN

## Área de trabajo Atmel Studio 7



Tomado de Microchip Technology, 2020b

En el área para digitar el código, se introduce el siguiente texto. En este caso, corresponde a un programa para generar el parpadeo de un led en el puerto B.

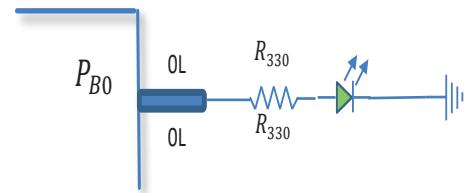
```
#include <avr/io.h>
#include <util/delay.h>

int main(void)
{
    /* Replace with your application code */
    DDRB=0B00000000;
    while (1)
    {
        PORTB=0B00000001;
        _delay_ms(1000);
        PORTB=0B00000000;
        _delay_ms(1000);
    }
}
```

La figura 2.6 ilustra el esquema de conexión que se debe realizar.

Figura 2.6

Esquema de conexión led parpadeante



En Atmel Studio se podrá apreciar algo semejante a la imagen 2.22, en la que se encuentra el código digitado. En este caso, el programa usa el puerto B con el pin PB0, en el cual está conectado un led. Este código permite que el led se encienda durante 1 segundo y se apague durante 1 segundo, de manera infinita, debido al bucle creado.

Imagen 2.22

AMPLIAR  
IMAGEN

## Código en Atmel Studio 7

```
main.c # X Click ASF Wizard

#include <avr/io.h>
#include <util/delay.h>

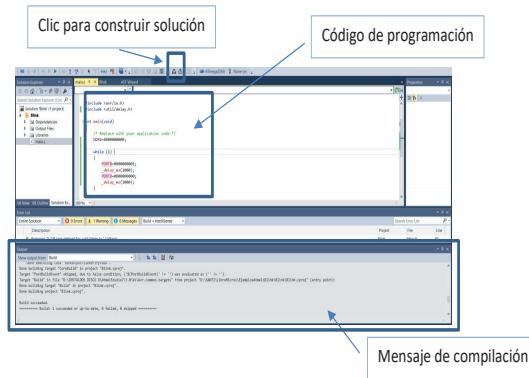
int main(void)
{
    /* Replace with your application code */
    DDRB=0B00000000;
    while (1)
    {
        PORTB=0B00000001;
        _delay_ms(1000);
        PORTB=0B00000000;
        _delay_ms(1000);
    }
}
```

Tomado de Microchip Technology, 2020b

Una vez generado el código, se construye la solución. Para ello, se da clic en (Build), en la barra de herramientas (ver imagen 2.23). Esta imagen también presenta una captura de pantalla del área de mensajes de compilación; si no ha tenido errores, se debe mostrar un mensaje como el aparece a continuación.

**Imagen 2.23**

AMPLIAR IMAGEN

*Construcción de la solución*

Tomado de Microchip Technology, 2020b

Si la construcción de la solución fue exitosa, en la carpeta del proyecto se han creado varios archivos. en una carpeta de nombre *Debug*; entre ellos, el más importante es de extensión (.hex) (ver imagen 2.24).

**Imagen 2.24***Archivos generados por la compilación*

Nombre	Fecha de modificación	Tipo	Tamaño
BlinkEEP	27/5/2020 17:59	Archivo EEP	1 KB
Blinkelf	27/5/2020 17:59	Archivo ELF	13 KB
<b>Blinkhex</b>	27/5/2020 17:59	Archivo HEX	<b>1 KB</b>
Blinklss	27/5/2020 17:59	Archivo LSS	9 KB
Blinkmap	27/5/2020 17:59	Linker Address Map	16 KB
Blinksrec	27/5/2020 17:59	Archivo SREC	1 KB
main.d	27/5/2020 17:59	Archivo D	3 KB
main.o	27/5/2020 17:59	Archivo O	4 KB
makedep.mk	27/5/2020 16:49	Makefile	1 KB
Makefile	27/5/2020 17:59	Archivo	4 KB

El archivo *Blink.hex* contiene un código a ser cargado en el microcontrolador. Este archivo podrá ser subido a una placa de desarrollo, por ejemplo el Arduino Mega2560, o directamente a los microcontroladores ATmega, mediante el uso de un TTL USB conectado a la PC.

**Nota.** Cuando se compila un código fuente, se crean archivos objeto (.o), archivos binarios (.bin o .hex) makefiles o Cmakefiles



# Capítulo 3

## Puertos de entrada-salida

### 3.1 Introducción a manejo de puertos

El microcontrolador es un dispositivo con una estructura de periféricos definidos, cuyas funcionalidades dependen enteramente del código de programa desarrollado por el usuario. En este capítulo, nos enfocamos, de manera especial, en mostrar la interacción entre el microcontrolador y el entorno que lo rodea; es decir, la manera en que se puede recibir y enviar información desde y hacia el mundo físico de manera digital.

La principal característica a tomar en cuenta es el tipo de información (analógica o digital) que puede salir o ingresar, desde o hacia el dispositivo. En este contexto, es relevante también conocer los niveles de voltaje analógicos y digitales asociados a las entradas y salidas del microcontrolador. El máximo valor de entrada permitido en los pines, según la hoja de especificaciones (Atmel, 2014,p. 355), es de  $V_{cc}^5 + 0,5\text{ V}$  y el mínimo es  $-0,5\text{ V}$ . Esto descarta la posibilidad de ingresar voltajes fuera de estos rangos. Una representación de los niveles de voltaje permitidos se encuentra en la figura 3.1.

Si se toman en cuenta los niveles lógicos de entrada, 1L corresponde a los voltajes comprendidos entre  $0,6\text{ V}_{cc}$  y  $V_{cc} + 0,5\text{ V}$ ; además, se considera un estado de 0L a voltajes comprendidos entre  $-0,5\text{ V}$  y  $0,3\text{ V}$ , donde  $V_{cc}$  es la alimentación de energía del microcontrolador. Al contrario, la salida digital en cada uno de los pines del microcontrolador para el nivel 0L, es un valor de voltaje igual o menor a  $0,9\text{ V}$  y la salida en 1L debe ser equivalente a un valor mínimo de  $4,2\text{ V}$  (Atmel, 2014,p. 355). Otra característica importante es que los pines (PXn) del

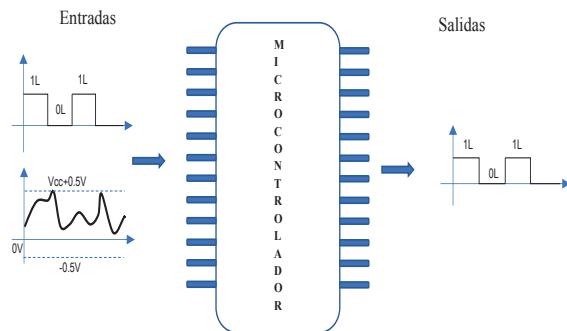
microcontrolador son de tipo Entrada-Salida (E/S) digital. Sin embargo, hay un grupo de pines que, dependiendo de su configuración, pueden ser utilizados como entradas analógicas. Analizaremos este aspecto con mayor profundidad en el capítulo 7.

**Nota.** Para establecer los voltajes de los estados lógicos de entrada y salida, es necesario tomar en cuenta el voltaje de alimentación y recurrir al manual del microcontrolador.

**Nota.** Se puede decir que los niveles de voltaje manejados en los pines del microcontrolador, tanto para entradas como para salidas digitales, cuando la alimentación es de  $5\text{ V}$ , corresponden a una lógica tipo TTL (Transistor-Transistor Logic).

Figura 3.1

Niveles de tensión para E/S



<sup>5</sup> En circuitos de tipo digital, la notación  $V_{cc}$  se refiere al voltaje de alimentación aplicado al circuito.

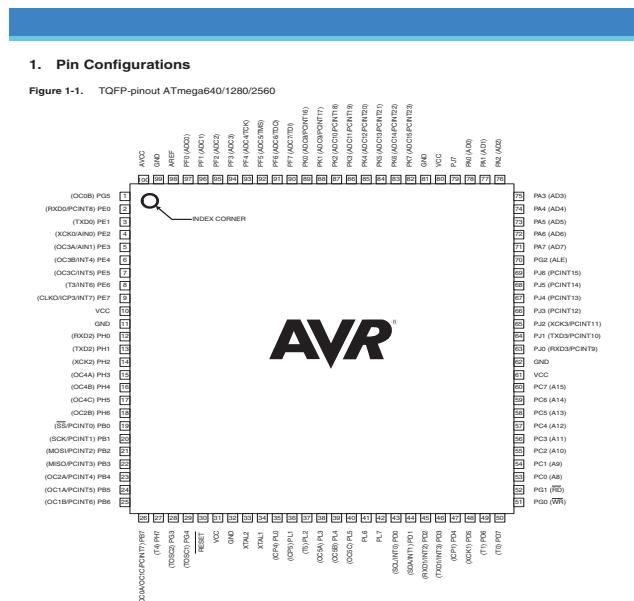
### 3.2 Registros de entrada-salida

Cada uno de los pines de E/S del microcontrolador tienen la opción de escritura y de lectura. Además, en el caso de su comportamiento como entradas, estos pueden configurarse con características de alta impedancia<sup>6</sup> o *pull-up*<sup>7</sup> interna. Todo depende de la aplicación y configuración establecidas por el usuario en los registros<sup>8</sup> de configuración respectivos.

Los registros que gobiernan el comportamiento de los pines del microcontrolador ATmega2560 son: DDRx, PORTx, MCUCR y PINx, donde el subíndice  $x$  hace referencia al nombre del pórptico A,B,C,D,E,F. La imagen 3.1 representa el esquema de pines del microcontrolador ATmega2560.

Imagen 3.1 | AMPLIAR IMAGEN 

## *Esquema de pines para el microcontrolador ATmega2560*



Tomado de Atmel 2014 p. 2

<sup>6</sup> La alta impedancia se refiere a una oposición elevada al flujo de corriente que presenta un valor de resistencia elevado. Dicho comportamiento puede asemejarse a un circuito abierto.

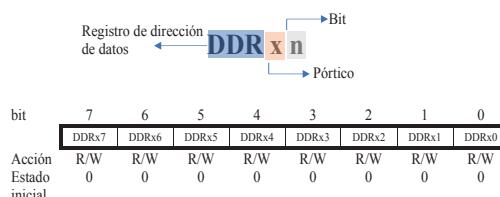
<sup>7</sup> Las resistencias de *Pull-up* permiten establecer un nivel de estado lógico HIGH en un pin de entrada cuando este no tiene un estado definido de entrada. Este comportamiento se logra haciendo una conexión por medio de un elemento resistivo hacia la fuente de alimentación.

<sup>8</sup> Son espacios de memoria disponibles, cuya manipulación afecta las configuraciones y el comportamiento del microcontrolador.

DDR<sub>x</sub>. Este registro permite establecer el comportamiento como entrada o salida de cada pin de un puerto determinado. Cuando un bit específico del registro DDR<sub>x</sub> se encuentra en estado 1L, el pin asociado del puerto se encontrará funcionando como salida, mientras que si es 0L se comportará como entrada.

Figura 3.2

### *Asignación de bits para un registro DDRx*



Tomado de Atmel, 2014, p. 96-100

Como se puede observar en la figura 3.2, el registro DDRx está conformado por 8 bits. Cada uno de ellos tiene opción de lectura y escritura (R/W) y puede ser utilizado para conocer cómo se ha configurado un pin (PXn) o, también, para cambiar su configuración de E/S. Además, hay que tener en cuenta que cuando el microcontrolador se enciende todos estos bits tienen un estado de 0L, por lo que todos los pines están definidos como entrada.

PORPx. Este registro es utilizado para establecer el valor de salida en cada pin de un puerto determinado (PXn). Este valor puede ser 0L o 1L. Al igual que el registro DDRx, este dispone de 8 bits que pueden ser leídos y modificados, y cuyo valor, por defecto, es 0L.

**Figura 3.3**

Asignación de bits para un registro PORTx

bit	7	6	5	4	3	2	1	0
Acción	R/W							
Estado inicial	0	0	0	0	0	0	0	0

Tomado de Atmel, 2014, p. 96-100

El registro PORTx presenta una funcionalidad especial cuando el registro DDRx se encuentra configurado como entrada (ver figura 3.3). En ese caso, es posible que cada pin de un pótico específico pueda ser configurado como una entrada con *pull-up* interno.

MCUCR. El registro de control permite deshabilitar todas las entradas de *pull-up* en el microcontrolador, aun cuando los registros DDRx y PORTx se encuentren configurados para esta tarea. Su valor, por defecto, es 0L y puede ser modificado en cualquier momento, mediante el bit 4 (PUD) (ver figura 3.4).

**Figura 3.4**

Registro de control

bit	7	6	5	4	3	2	1	0
JTD	-	-	PUD	-	-	-	IVSEL	IVCE
Acción	R/W	R	R	R/W	R	R	R/W	R/W
Estado inicial	0	0	0	0	0	0	0	0

Tomado de Atmel, 2014, p. 96

El comportamiento de los PxN en el microcontrolador puede resumirse en la tabla 3.1.

**Tabla 3.1**

Comportamiento de los pines E/S en función de los registros asociados PORTx, DDRx y MCUCR

DDRxn	PORTxn	PUD (en MCUCR)	E/S	Pull- up	Estado
0	0	X	Entrada	No	Alta impedancia (Hi-Z)
0	1	0	Entrada	Sí	20 -50 KΩ interna
0	1	1	Entrada	No	Alta impedancia (Hi-Z)
1	0	X	Salida	No	Salida en bajo 0L
1	1	X	Salida	No	Salida en alto 1L

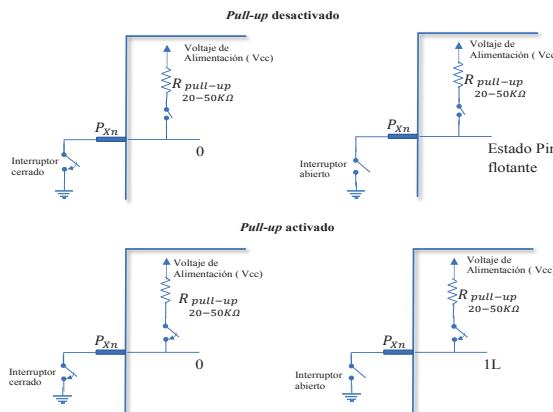
Tomado de Atmel, 2014, p. 69

El estado de *pull-up* se puede representar como una resistencia interna al microcontrolador que, dependiendo de las configuraciones, puede o no conectarse al pin internamente. El uso más frecuente se da en el caso de manejo de pulsantes o interruptores para especificar estados lógicos de entrada en un pin.

Así, el estado de pin flotante se refiere a una característica en la cual no es posible definir un estado lógico de la entrada de un pin en el microcontrolador; es decir, no se puede asegurar que haya un estado 0L o 1L conectado en el pin PxN. Este estado se debe evitar, con el fin de reducir el consumo de energía. La figura 3.5 presenta los diferentes casos para la conexión de una señal externa a un pin del microcontrolador, tomando en cuenta *pull-up* activado y desactivado.

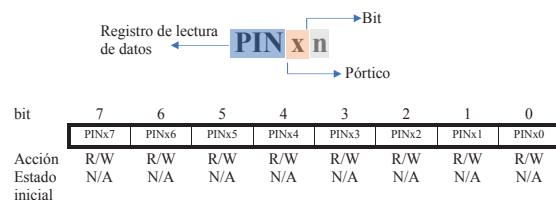
**Figura 3.5**

Configuraciones de pull-up y estados lógicos asociados



PIN<sub>x</sub>. Es un registro para entrada de datos de 8 bits. En sí, cada uno puede ser utilizado tanto para la lectura como para la escritura. Su estado, por defecto, no es conocido. Cuando el DDR<sub>xn</sub> se encuentra configurado como entrada (0L), en el PIN<sub>xn</sub> es posible leer el estado lógico del pin asociado P<sub>Xn</sub>. Al contrario, si el DDR<sub>xn</sub> se encuentra configurado como salida (1L), el PIN<sub>xn</sub> contendrá el valor cargado en el PORT<sub>xn</sub> (ver figura 3.6).

**Figura 3.6**



Asignación de bits para un registro PIN<sub>x</sub>

Tomado de Atmel, 2014, p. 96-100

### 3.2.1 Ejercicios prácticos de configuración de puertos

Con el fin de entender de mejor manera el funcionamiento y la configuración de los distintos registros asociados a los pines de E/S, a continuación, presentamos una serie de casos en los que, de

manera progresiva, profundizamos los conceptos tratados a lo largo de este capítulo, mediante ejercicios prácticos.

**Ejercicio 3.1:** Se dispone de 8 ledes conectados al pórtico B (salidas), de los cuales solo los ledes conectados a los pines PB1, PB3 y PB5 deben encenderse, mientras que los demás ledes deben estar apagados.

**Solución:**

Las configuraciones en los registros serían las siguientes:

**Figura 3.7**

Configuración de registros. Ejercicio 3.1

bit	7	6	5	4	3	2	1	0
DDR <sub>B</sub>	1	1	1	1	1	1	1	1
PORT <sub>B</sub>	0	0	1	0	1	0	1	0

En código sería:

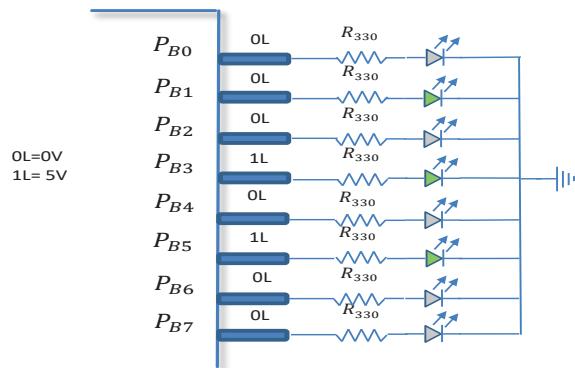
```
#include <avr/io.h>

int main(void)
{
    DDRB=0B11111111; // también DDRB=255; //DDR=0xFF
    PORTB=0B00101010; // también PORTB=42; // PORTB=0x2A
    while (1)
    {
    }
}
```

Tomando en consideración este primer ejemplo, es posible apreciar la inclusión de la librería *avr/io.h*. Esta permite incorporar las definiciones de todos los registros ..... PORT y DDR (Williams, 2014). La figura 3.8 presenta el esquema para la conexión de 8 salidas conectadas a ledes.

**Figura 3.8**

Esquema de conexión y estado de salidas. Ejercicio 3.1



**Nota.** El uso de resistencias de  $330\Omega$  toma en cuenta que 1L representa 5 V y asume que el voltaje de barrera del led es 2,2 V; entonces, la corriente a obtener en el diodo será de alrededor de:  $I_{diodo} = \frac{5 - 2.2}{300} = 8.48 \text{ [mA]}$ .

**Nota.** Es posible realizar una conexión de los diodos de manera invertida, en donde el ánodo de todos los diodos se conecta a la fuente de energía. En esta configuración, la corriente proviene de la fuente, y no de los pines del microcontrolador. En este caso, la lógica de encendido/apagado de los ledes es invertida; es decir, cada led se enciende colocando 0L, en lugar de 1L, y viceversa.

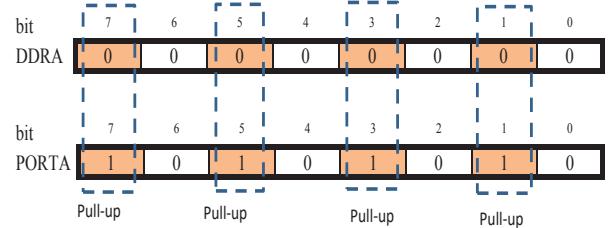
**Ejercicio 3.2:** Se requiere ingresar información al microcontrolador. La información proviene de 8 interruptores conectados en el pótico A. Dicha información debe ser almacenada en el microcontrolador, en una variable de 8 bits, para su posterior procesamiento. Además, físicamente solo 4 interruptores disponen de resistencia de *pull-up*. Los 4 interruptores restantes deben usar resistencias de *pull-up* internas en el microcontrolador. Los interruptores con resistencia de *pull-up* interna tienen que estar conectados a los pines del microcontrolador PA7, PA5, PA3, PA1.

### Solución:

La configuración de los registros debe ser:

**Figura 3.9**

Configuración de registros. Ejercicio 3.2



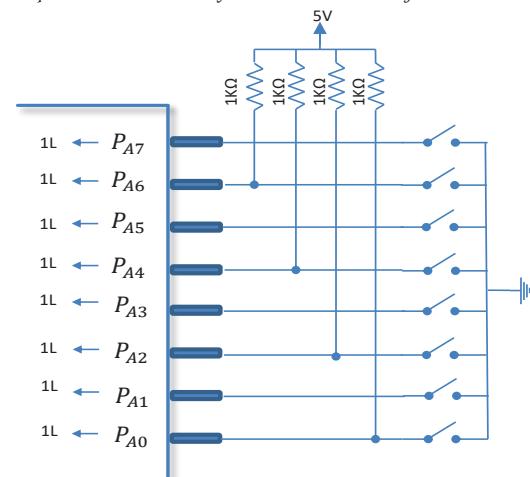
El código para la lectura de la información de entrada sería el siguiente:

```
#include <avr/io.h>
unsigned char a=0; //variable global de 8 bits sin signo
int main(void)
{
    DDRA=0b00000000; //en decimal DDRA=0; // en hexadecimal DDRA=0x00;
    PORTA=0b10101010; //en decimal PORTA=170; // en hexadecimal PORTA=0xAA;
    while (1)
    {
        a=PINA; // la variable a copia el contenido del registro PINA
    }
}
```

La figura 3.10 ilustra los interruptores con *pull-up* externo y los configurados con *pull-up* interno. Al estar abiertos, generan un estado de 1L en cada pin del pótico A. El esquema de conexiones es el siguiente:

**Figura 3.10**

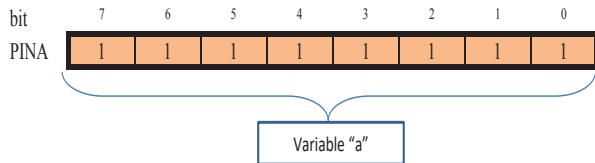
Esquema de conexión y estados entrada. Ejercicio 3.2



La información recibida por los pines del pótico A se almacena en el registro de entrada de datos PINA, como indica la figura 3.11:

**Figura 3.11**

Representación de la información en registro PINx. Ejercicio 3.2



**Ejercicio 3.3:** Se dispone del pótico C para la adquisición y salida de señales digitales. Los pines PC0, PC1 y PC2 serán tratados como entradas y conectados a interruptores sin resistencia de *pull-up* externa, mientras que los pines PC3 a PC7 serán configurados como salidas.

Solución:

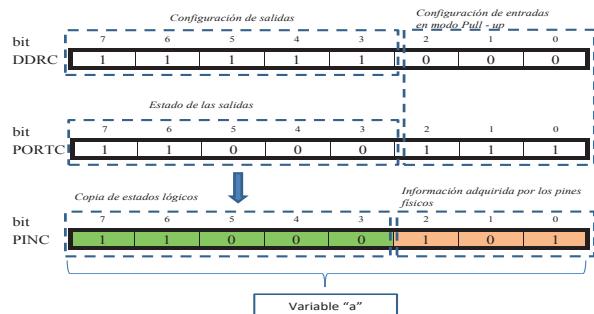
La figura 3.13 muestra el esquema de conexión.

```
#include <avr/io.h>
unsigned char a=0;
int main(void)
{
    DDRC=0b11111000; //también 0xf8 o 248 decimal
    PORTC=0b11000111; //también 0xC7 o 199 decimal
    while (1)
    {
        a=PINC;
    }
}
```

Si en el caso presentado se lee el contenido del registro PINC, se advierte un inconveniente al momento de leer los estados que interesan (PC0, PC1 y PC2). Entonces, ¿qué sucede con los bits del puerto que fueron configurados como salidas? Pues bien, es necesario revisar el contenido del registro PINC que fue almacenado en la variable “a”.

**Figura 3.12**

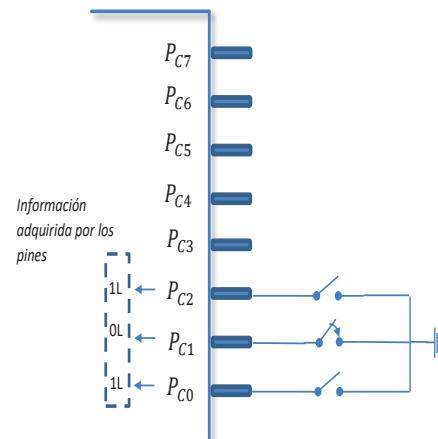
Configuración de registros. Ejercicio 3.3



Si se toma en cuenta el contenido del registro PINC, este, además de contener los 3 bits de entrada, también contiene información proveniente del registro PORTC en sus bits del 3 al 7. Con esta información en mente, se debe realizar un procedimiento por el cual solo se puedan leer los bits de interés como entradas y descartar los bits no deseados, provenientes del registro PORTC. Esto se denomina enmascaramiento (ver figura 3.12).

**Figura 3.13**

Esquema de conexión y estados entrada. Ejercicio 3.3



### 3.3 Enmascaramiento y operaciones bit a bit

El enmascaramiento es un proceso que puede afectar el estado lógico de bits específicos de un determinado registro. Los principales enmascaramientos se basan en las operaciones de AND, OR y XOR.

**AND:** la operación AND puede ser utilizada para colocar 0L en bits determinados de un registro, sin alterar a los restantes. Además, puede usarse como filtro para obtener el valor de un grupo de bits dejando en 0 los demás.

**OR:** la operación OR puede ser utilizada para colocar 1L en bits determinados de un registro, sin cambiar los estados de los restantes.

**XOR:** la operación XOR puede ser utilizada para hacer un cambio de estado anterior (TOOGLE) en bits determinados de un registro, dejando sin cambio a los restantes.

### 3.3.1 Ejercicios prácticos de enmascaramiento

**Ejercicio 3.4:** Tomando en cuenta que inicialmente en un código se configuró el registro DDRA = 0b11001100 y que, posteriormente, se desea cambiar los estados de los bits 2 y 6 a 0L y dejar inalterados los demás bits. En este caso, como el objetivo es colocar en bajo dichos bits, se debe utilizar un enmascaramiento AND, para que la operación bit a bit resulte de la siguiente manera:

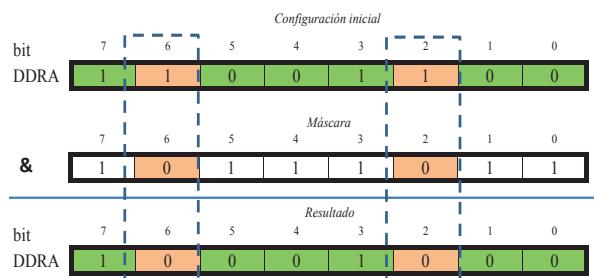
**Solución:**

$$\text{DDRA} = \text{DDRA} \& 0b10111011 \rightarrow \text{Máscara}$$

En la figura 3.14 se puede observar el efecto del enmascaramiento para el ejemplo propuesto.

Figura 3.14

Enmascaramiento tipo AND para modificaciones de bit a 0L en el registro DDRA. Ejercicio 3.4



Se puede apreciar que solo los bits, cuya máscara tenía cero, cambiaron su estado a 0L, mientras que

los demás permanecieron inalterados. Esto resulta de suma utilidad cuando se desconoce el estado anterior de los bits que deberían modificarse. De esta manera, se pueden cambiar únicamente los estados de los bits que sí deberían ser alterados.

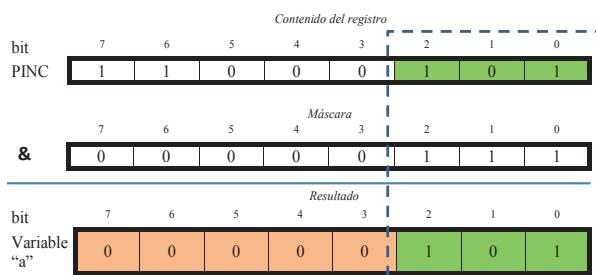
**Ejercicio 3.5:** Si se toma en cuenta el caso del ejercicio 3.3, tras leer el registro PINC (0b11000101), se obtuvieron los valores de bits de entrada y otros no deseados pertenecientes a los estados del registro PORTC. En sí, un enmascaramiento AND puede solucionar este problema.

**Solución:**

Las operaciones de enmascaramiento se pueden observar en la figura 3.15.

Figura 3.15.

Enmascaramiento tipo AND para rescate de información en bits específicos. Ejercicio 3.5



Se puede apreciar que la variable "a" contiene solo la información proveniente de los pines de entrada, descartando los valores no deseados de los bits 3 – 7 y colocando un valor de 0L en ellos. El código para realizar esta operación se presenta a continuación:

```
#include <avr/io.h>
unsigned char a=0;
int main(void)
{
    DDRC=0b11111000; //también 0xf8 o 248 decimal
    PORTC=0b11000111; //también 0xc7 o 199 decimal
    while (1)
    {
        a=PINC&0b00000011;
    }
}
```

**Ejercicio 3.6:** Si se configuró el registro DDRA = 0b10001100 al inicio del programa y en algún momento posterior se desea cambiar los estados de los bits 0 y 5 a 1L y, a su vez, dejar inalterados los demás bits. En este caso particular, como se busca colocar en alto dichos bits, se debe utilizar un enmascaramiento OR para la operación bit a bit de la siguiente manera:

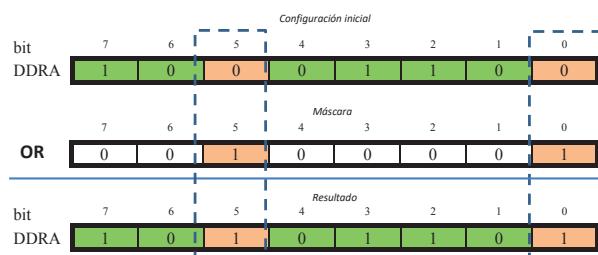
**Solución:**

$$\text{DDRA} = \text{DDRA} | 0b00100001 \rightarrow \text{Máscara}$$

Las operaciones de enmascaramiento se pueden observar en la figura 3.16.

Figura 3.16

Enmascaramiento tipo OR para modificaciones de bit a 1L en el registro DDRA. Ejercicio 3.6

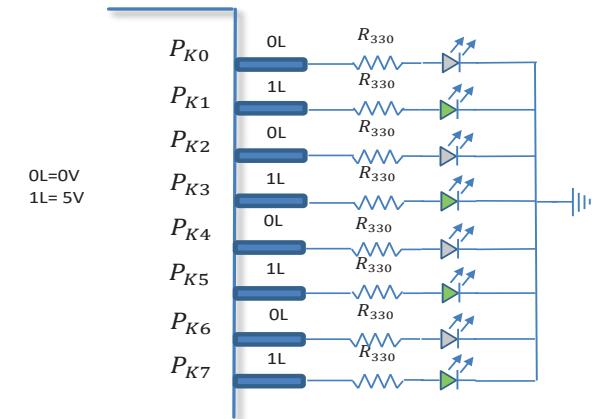


Se puede apreciar que en el registro DDRA, solo los bits cuya máscara tenía 1L cambiaron su estado de 0L a 1L, mientras que los demás permanecieron inalterados. Este caso resulta de suma utilidad en situaciones en las que se desconoce inicialmente el estado anterior de bits que no se desean modificar. Así se pueden cambiar los estados únicamente de los bits seleccionados.

**Ejercicio 3.7:** En el siguiente código se aprecia el uso de la operación XOR para realizar el encendido y apagado (*blinking*) de ledes conectados en los pines PK7, PK5, PK3, PK1, mientras los demás pines del pótico K permanecen en 0L (ledes apagados). En la figura 3.17 se observa el esquema de conexiones.

Figura 3.17

Esquema de conexiones para encendido y apagado de ledes (blinking). Ejercicio 3.7

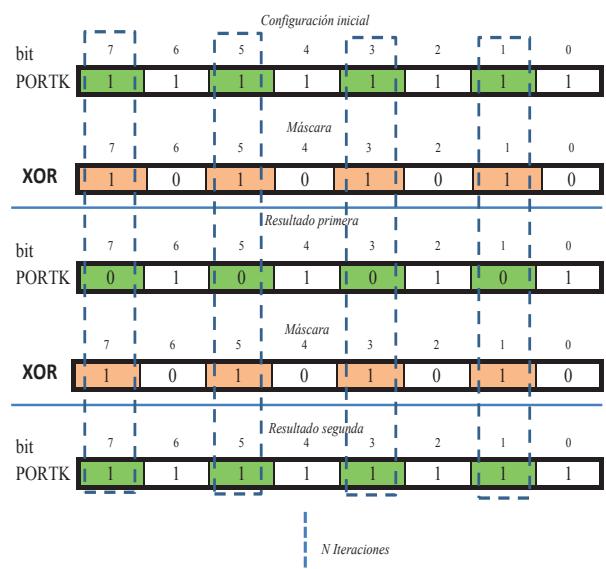


**Solución:**

La figura 3.18 ilustra con claridad que el enmascaramiento XOR provoca un cambio en el estado lógico de cada iteración en aquellos bits en los cuales la máscara contiene el valor de 1L. Los demás bits del Pótico K quedan inalterados en las N iteraciones.

Figura 3.18

Enmascaramiento tipo XOR para operaciones bit a bit. Ejercicio 3.7



A continuación, se presenta el código que demuestra el funcionamiento de la máscara XOR. Primero, se define al puerto K como salida en cada uno de sus bits y se deja como estado inicial todos los bits en 1L a la salida del puerto. Una vez que ingresa al lazo recursivo *While (1)*, se aplica un enmascaramiento al puerto y se lo sobrescribe haciendo una operación XOR (^).

```
#include <avr/io.h>
#include <util/delay.h>
int main(void)
{
    DDRK=255;
    PORTK=255;
    while (1)
    {
        PORTK=PORTK^0b10101010; Máscara
        _delay_ms(500);
    }
}
```

### 3.4 Operaciones abreviadas

Existe una manera de escribir operaciones de manera abreviada, para facilitar la comprensión del código y disminuir el tiempo de desarrollo. Algunos ejemplos se pueden observar en la tabla 3.2.

**Tabla 3.2**

*Operaciones abreviadas para Atmel Studio*

Aplicación	Operación	Operación abreviada
Enmascaramiento OR	DDRA= DDRA 0b10101010	DDRA =0b10101010
Enmascaramiento AND	DDRA=DDRA&0b10101010	DDRA&=0b10101010
Enmascaramiento XOR	DDRA=DDRA^0b10101010	DDRA^=0b10101010
1L en bit específico*	DDRA=DDRA (1<<bit_pos)	DDRA = (1<<bit_pos)
0L en bit específico*	DDRA=DDRA&~(1<<bit_pos)	DDRA&=~(1<<bit_pos)

\*Donde *bit\_pos* representa el número de desplazamiento a realizarse y puede tener valores entre 0-7

### 3.4.1 Ejercicios prácticos de configuración E/S

A continuación, presentamos ejercicios en los que se aplican los distintos conceptos del presente capítulo.

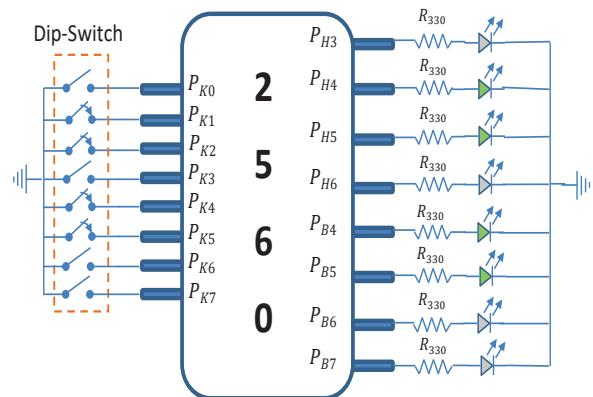
**Ejercicio 3.8 (Enmascaramiento básico):** Los métodos de enmascaramiento tienen una participación relevante cuando se requiere el uso compartido o parcial de los pines de un puerto. En el siguiente ejemplo, el *dip-switch*<sup>9</sup> está conectado a un puerto entero, 4 ledes, están conectados al pótico H y 4 ledes en el pótico B. El objetivo es que los ledes se enciendan o apaguen en función del estado de los respectivos interruptores del *dip-switch*.

#### Solución:

La figura 3.19 representa el esquema de conexiones, en el cual el *dip-switch* está conectado sin resistencia al *pull-up* externa.

**Figura 3.19**

*Esquema de conexiones con dip-switch y ledes. Ejercicio 3.8*



Primero, se define el encabezado conformado por librerías y declaraciones de variables globales.

```
#include <avr/io.h>
unsigned char var=0;
```

A continuación, se establece una función de configuración de puertos, según el esquema planteado en la figura 3.19.

<sup>9</sup> Elemento que contiene varios interruptores eléctricos de dos estados (abierto-cerrado) en un solo encapsulado tipo DIP (*Dual in line package*).

```

void config_ports()
{
    DDRK=0;//configuro como entradas todos los pines
    PORTK=255; // coloco pull-ups en todos los pines
    DDRB|=0b11110000; // configuro como salidas el nibble alto
    DDRH|=0b01111000; // configuro como salidas los pines PH6, PH5, PH4 y PH3
}

```

Finalmente, se ingresa la función de configuración dentro de programa principal (*main*) y se establece un lazo infinito de lectura y escritura de puertos, mediante el uso de técnicas de enmascaramiento y operaciones bit a bit.

```

int main(void){

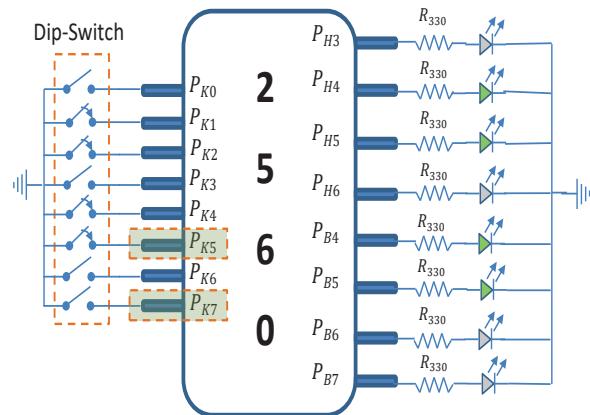
    config_ports();
    while(1)
    {
        var=PINK;
        PORTB=(PORTB&0b00001111)|(var&0b11110000);
        PORTH=(PORTH&0b10000111)|(var&0b00001111)<>3;
    }
}

```

**Ejercicio 3.9 (Luces inteligentes):** En este ejemplo se desarrolla un programa que permita, mediante la lectura del estado de dos interruptores (abierto, cerrado) pertenecientes a un *dip-switch*, realizar 4 acciones sobre los ledes: contador ascendente, descendente, parpadeo y luces tipo auto fantástico. Para tal efecto, los interruptores a ser leídos serán los correspondientes a los pines PK5 y PK7. En la figura 3.20 se puede observar el esquema del circuito a implementar.

**Figura 3.20**

Esquema de conexiones para luces inteligentes. Ejercicio 3.9



### Solución:

Se inicia con la declaración de librerías, incluyendo la librería para retardos y la definición de una etiqueta *F\_CPU*, que es utilizada por dicha librería. El valor de *F\_CPU* depende del cristal que se esté utilizando, en este caso 16 Mhz. Además, se declaran las variables globales a ser usadas a lo largo del código.

```

#define F_CPU 16000000UL
#include <avr/io.h>
#include <util/delay.h>
unsigned char var=0, contador=0, luces=0;

```

Se definen las configuraciones de entradas, salidas y *Pull-up*, correspondientes a los pines a ser utilizados.

```

void config_ports()
{
    DDRK&=~(1<<5); // pin PK5 como entrada
    DDRK&=~(1<<7); // pin PK7 como entrada
    PORTK|=(1<<5); // pull-up en el pin PK5
    PORTK|=(1<<7); // pull-up en el pin K7
    DDRB|=0b11110000; // el nibble alto como salidas
    DDRH|=0b01111000; // los pines PH6, PH5, PH4 y PH3 como salidas
}

```

Debido a que los ledes se encuentran en pórticos diferentes, es necesario la creación de una función que enmascare el dato de 8 bits del comportamiento de ledes en los dos pórticos asignados.

```
void encender_leds(unsigned char aux)
{
    PORTB=(PORTB&0b00001111)(aux&0b11110000);
    PORTH=(PORTH&0b10000111)((aux&0b00001111)<<3);
}
```

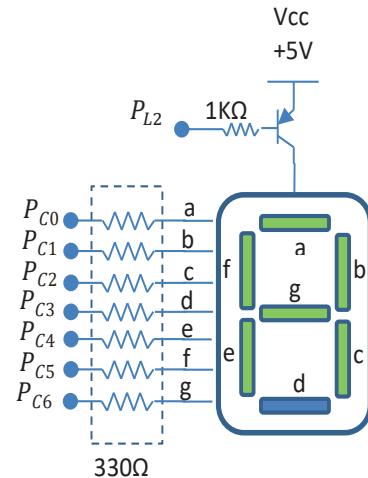
Finalmente, se ingresa la función de configuración de puertos dentro de programa principal (*main*), y se establece un lazo infinito de lectura y escritura de puertos mediante el uso de técnicas de enmascaramiento y estructuras condicionales “*if*” y “*for*”. En el código se observan las cuatro posibles opciones, según la posición de los dos interruptores a ser leídos a través de los pines PK5 y PK7.

```
int main(void)
{
    config_ports();
    while (1)
    {
        var=PINK&0b10100000;
        if (var==0) // contador ascendente
        {
            contador++;
            encender_leds(contador);
            // enmascara el dato en los 8 ledes pertenecientes a dos pórticos
        }
        if (var==160) // contador descendente
        {
            contador--;
            encender_leds(contador);
            // enmascara el dato en los 8 ledes pertenecientes a dos pórticos
        }
        if (var==128)// luces parpadeantes
        {
            encender_leds(0);
            _delay_ms(500);
            encender_leds(255);
            _delay_ms(500);
        }
        if (var==32)//luces tipo auto fantástico
        {
            for (unsigned char i=0;i<=7;i++)
            {
                luces=(1<<i);
                encender_leds(luces);
                _delay_ms(100);
            }
            for (unsigned char i=0;i<=7;i++)
            {
                luces=(128>>i);
                encender_leds(luces);
                _delay_ms(100);
            }
        }
    }
}
```

**Ejercicio 3.10 (Display 7 segmentos):** El *display* de 7 segmentos es un elemento visual de suma utilidad en casos en los que se requiere mostrar información de manera sencilla y de fácil visualización. Para el desarrollo de este ejemplo, se tiene en cuenta un *display* de 7 segmentos de ánodo común, en el cual se programa un contador ascendente en BCD (0, 1.... F).

Figura 3.21

Esquema de conexiones para display ánodo común Ejercicio 3.10



### Solución:

Primero, se establecen las conexiones del *display* con el microcontrolador, como indica la figura 3.21. Para el desarrollo del código, es necesario colocar las librerías necesarias y las variables globales a utilizar. Un punto que se debe tener en cuenta es la variable tipo arreglo, llamada “*anodo\_comun*”, la cual contiene 16 elementos de tipo “char” sin signo. Estos elementos representan la codificación necesaria para que en el *display* se muestren los dígitos del 0 al 9 y las letras a, b, c, d, e, f.

```
#define F_CPU 16000000UL
#include <avr/io.h>
#include <util/delay.h>

unsigned int numero=0;
unsigned char anodo_comun[16]={0xC0,0xF9,0xA4,0XB0,0x99,
0x92,0x82,0xF8,0x80,0x90,0X88,0X83,0XC6,0XA1,0X86,0XE};
```

El siguiente paso consiste en configurar los pines de salida conectados a los segmentos del *display* de ánodo común.

```
void config_puertos()
{
    DDRD|= (1<<2); // se coloca el pin PL2 como salida (activación display)
    DDRC|= 0b01111111; // se establecen desde PC0 hasta PC6 como salidas (segmentos)
}
```

En el lazo principal se establece, en primera instancia, la función de configuración de puertos, seguida de una orden (OL) para el transistor que permite encender el *display*. Enseguida, en el lazo recursivo se coloca la codificación del símbolo que se desea presentar en el *display* con base en un contador acumulativo “número”. Si este contador llega a un valor de 16, la cuenta se reinicia a cero, ya que no es posible desplegar codificaciones de símbolo superiores a 15, es decir, la letra F.

```
int main(void)
{
    config_puertos();
    PORTL&=~(1<<2);

    while(1)
    {
        if (numero==16)
        {
            numero=0;
        }
        PORTC=(PORTC&0x80)|anodo_comun[numero];
        // se envía el código de display sin alterar el PC7
        _delay_ms(1000);
        numero++;
    }
}
```

Con el fin de que los valores sean visibles por un breve instante de tiempo, se ha anexado un retardo de 1000 ms entre cada iteración.

**Ejercicio 3.11 (Barrido de displays):** Para poder visualizar varios dígitos sin utilizar excesivos pines del microcontrolador, se debe utilizar una técnica conocida como “barrido de *displays*”. Para este ejemplo, se plantea un contador de 0 a 9999 ascendente. La figura 3.22 expone el esquema de activación y desactivación de cada uno de los *displays*. La figura 3.23 presenta el esquema de conexiones para configurar el barrido de *display* de ánodo común.

Figura 3.22

Esquema de tiempos para barrido de displays. Ejercicio 3.11

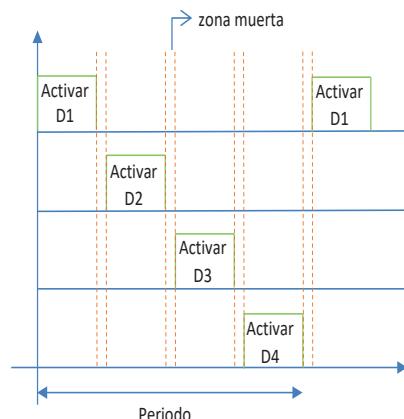
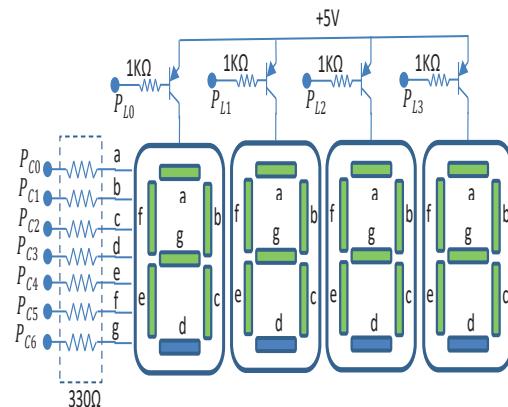


Figura 3.23

Esquema para conexión barrido 4 displays tipo ánodo común. Ejercicio 3.11



## Solución:

Hay que tener en cuenta que cada *display* se opera de manera individual y responde a una secuencia como la que exhibe la figura 3.22. La zona muerta hace referencia a un intervalo de tiempo corto, en el que ninguno de los *displays* se encuentra activado. Durante el arranque, cada *display* se enciende de manera consecutiva, mientras que los demás se encuentran apagados. Esto provoca que, a cierta frecuencia, exista la percepción visual de que todos los *displays* se encuentren encendidos.

A continuación, se definen las librerías y variables a usar. Se debe tomar en cuenta que como el contador será mayor a 255, es necesario usar una variable tipo “int”. Además, se deben utilizar las variables vistas en el ejercicio de *display de 7 segmentos*.

```
#define F_CPU 16000000UL
#include <avr/io.h>
#include <util/delay.h>
unsigned int numero=0;
unsigned char centenas=0,decenas=0,unidades=0, miles=0;
unsigned char anodo_comun[16]={0xC0,0xF9,0xA4,0XB0,0x99,
0x92,0x82,0xF8,0x80,0x90,0X88,0X83,0XC6,0XA1,0X86,0X8E};
```

Ahora se crea una función que permita descomponer un número en sus dígitos de unidades, decenas y centenas. En esta función es posible tener, como parámetro de entrada, un número de valor máximo 9999.

```
void conver_digi(unsigned int val)
{
    miles=val/1000;
    centenas=(val%1000)/100;
    decenas=((val%1000)%100)/10;
    unidades=((val%1000)%100)%10;
}
```

**Nota.** El operador módulo “%” es aquel que entrega el resto o residuo de una división entre dos números.

En la siguiente función se presenta la sucesión de arranque secuencial de los *displays* y la colocación de la codificación por cada dígito del barrido. Al acabar la función, se deja, por defecto, todos los *displays* apagados.

```
void mostrar_disp()
{
    PORTL&=~(1<<3);
    PORTC=anodo_comun[unidades];
    _delay_ms(1);
    PORTL=255;
    PORTL&=~(1<<2);
    PORTC=anodo_comun[decenas];
    _delay_ms(1);
    PORTL=255;
    PORTL&=~(1<<1);
    PORTC=anodo_comun[centenas];
    _delay_ms(1);
    PORTL=255;
    PORTL&=~(1<<0);
    PORTC=anodo_comun[miles];
    _delay_ms(1);
    PORTL=255;
}
```

A continuación, se establecen las configuraciones de los puertos de salida tanto para los pines conectados a los segmentos del *display* como para los pines asociados al control de encendido conectados en la base de los transistores.

```
void config_puertos()
{
    DDRC=255; // defino salida de datos el pótico C para los displays 7 segmentos
    DDRL=0xff; // defino salidas para el control de los cuatro dígitos del display, los 4 bits más bajos son los utilizados
    PORTC=0; // inicializo con salidas en cero
    PORTL=255; // inicializo con salidas de control en cero.
}
```

En el lazo recursivo se muestra la estructura condicional “*for*”, usada para mantener durante un cierto número de iteraciones un mismo valor mostrado en el barrido del *display*, antes que este sea incrementado. El objetivo de esta acción es que la cuenta ascendente no sea tan rápida y pueda ser apreciada por el usuario. Por último, la estructura condicional “*if*” permite encender el contador si este supera el límite de 9999.

```
int main(void)
{
    config_puertos();
    while(1)
    {
        conver_digi(numero);
        for(unsigned char i=0;i<=2;i++)
        {
            mostrar_disp();
        }
        numero++;
        if(numero==10000)
        {
            numero=0;
        }
    }
}
```

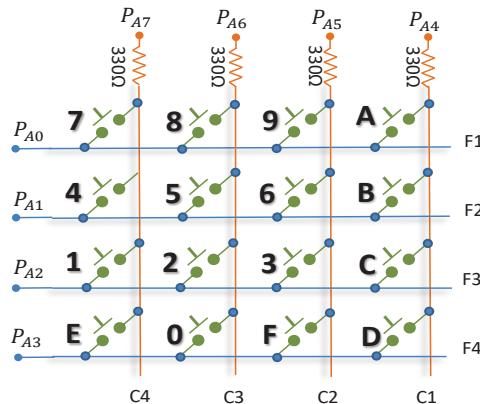
Hay que tener en cuenta que en un barrido de *display* de ánodo común la lógica de activación es invertida y se usan transistores tipo PNP; es decir que, para activar un determinado *display*, su pin de control asociado debe estar en estado 0L. Además, la activación de los segmentos también tiene lógica invertida, por lo cual cada segmento se encenderá al colocar un 0L en el pin respectivo.

**Ejercicio 3.12 (Barrido de teclado):** El barrido de teclado es una técnica muy utilizada para obte-

ner información de varios pulsantes con el uso de pocos pines. La figura 3.24 permite apreciar un esquema del circuito para un arreglo de teclado de 4x4. Con el fin de probar el código de barrido de teclado, se usa un *display* de ánodo común para mostrar el valor del pulsante que haya sido presionado.

**Figura 3.24**

Esquema para conexión teclado 4x4. Ejercicio 3.12



### Solución:

Primero, se debe realizar la declaración de librerías y variables que serán utilizadas.

```
#include <avr/io.h>          // Librería de Entrada y Salida
unsigned char anodo_comun[16]={0xC0,0xF9,0xA4,0XB0,0x99,
0x92,0x82,0xF8,0x80,0x90,0X88,0X83,0XC6,0XA1,0X86,0X8E};
unsigned char clave, tecla=0;
```

Después, se establecen las entradas con *pull-up* (filas del teclado matricial) y las salidas (columnas del teclado matricial). Además, se tienen que definir las salidas para los segmentos del *display* y el transistor de activación.

```

void config_puertos()
{
    DDRA=0XF0;           // Bajos Entrada Altos Salida
    PORTA=0XF0;          // Habilito Pull-Up
    DDRC=0xFF;
    DDRD|=(1<<2); // se coloca el pin PL2 como salida (activación display)
}

```

Dentro del lazo recursivo, se efectúa un barrido sucesivo de ceros, a través de los pines conectados a las columnas del teclado. Esto provoca que las entradas, que normalmente leen el valor de 1L debido al *pull-up* (PA0 – PA3), al ser presionado un pulsante, cambian al estado 0L. Al leer el contenido del pórtico, si en alguno de los pines de entrada se obtiene 0L, significa que el pulsante asociado a la columna que envía cero y la fila que lo recibe ha sido presionado. Una vez que el caso ha sido encontrado a través de la estructura *switch-case*, y tomando en cuenta la disposición de teclas de la figura 3.24, se fija una codificación para la tecla presionada, la cual es mostrada por medio del *display* de 7 segmentos.

```

int main(void)
{
    config_puertos();
    while(1)
    {
        for(unsigned char i=0;i<=3;i++)
        {
            PORTA =~(0X10<<i);
            clave=PINB;
            switch (clave)
            {
                case (0X7E): tecla=anodo_comun[7]; break;
                case (0XBE): tecla=anodo_comun[8]; break;
                case (0XDE): tecla=anodo_comun[9]; break;
                case (0X7D): tecla=anodo_comun[4]; break;
                case (0XBD): tecla=anodo_comun[5]; break;
                case (0XDD): tecla=anodo_comun[6]; break;
                case (0X7B): tecla=anodo_comun[1]; break;
                case (0XBB): tecla=anodo_comun[2]; break;
                case (0XDB): tecla=anodo_comun[3]; break;
                case (0XB7): tecla=anodo_comun[0]; break;
                case (0XEE): tecla=anodo_comun[10]; break;
                case (0XED): tecla=anodo_comun[11]; break;
                case (0XEB): tecla=anodo_comun[12]; break;
                case (0XE7): tecla=anodo_comun[13]; break;
                case (0XD7): tecla=anodo_comun[15]; break;
                case (0X77): tecla=anodo_comun[14]; break;
            }
        }
        PORTC =tecla;
    }
}

```



# Capítulo 4

## Interrupciones

### 4.1 Anotaciones preliminares sobre interrupciones

El microcontrolador es un dispositivo que ejecuta una serie de instrucciones de manera secuencial, a una velocidad definida por la frecuencia de reloj. Esto implica que, para que una acción u operación sea llevada a cabo, debe esperarse que las anteriores hayan terminado su ciclo de ejecución.

Cuando se trabaja con una estructura de programación estructurada y secuencial en la que los cambios lógicos asociados a los pines del microcontrolador deben ser leídos y analizados mediante sentencias *if*, *case* o *while*, se ocasiona un retardo en la toma de decisiones. El microcontrolador debe ejecutar secuencialmente cada una de las instrucciones de su programa hasta llegar a la instrucción esperada.

Muchas veces, y dependiendo de las aplicaciones en las que se use el microcontrolador, la necesidad de ejecutar una acción con base en un evento externo o interno resulta de vital importancia, especialmente cuando una operación inmediata por parte del microcontrolador sea requerida. Los eventos externos al microcontrolador pueden referirse, por ejemplo, al cambio de estado lógico en uno de sus pines, por acción de un pulsante, sensores de barrera o *encoders*, etc., en donde la rápida respuesta por parte del microcontrolador a dichos eventos deba ser prioritaria. Los eventos internos pueden desencadenarse por avisos de temporizadores, conversiones analógicas o, incluso, por la finalización o recepción de información, mediante protocolos de comunicación.

En el microcontrolador ATmega2560, se dispone de vectores de interrupción que permiten identificar cada uno de los eventos que pueden causar una interrupción junto con su registro de memoria asociado. La tabla 4.1 expone cada uno de los vectores de interrupción.

**Tabla 4.1**

*Vectores de reset e interrupción*

Vector	Dirección memoria de programa	Fuente del evento	Definición de la interrupción
1	\$0000	RESET	External Pin, Power-on Reset, Brown-out Reset, Watchdog Reset, and JTAG AVR Reset
2	\$0002	INT0	External Interrupt Request 0
3	\$0004	INT1	External Interrupt Request 1
4	\$0006	INT2	External Interrupt Request 2
5	\$0008	INT3	External Interrupt Request 3
6	\$000A	INT4	External Interrupt Request 4
7	\$000C	INT5	External Interrupt Request 5

8	\$000E	INT6	External Interrupt Request 6
9	\$0010	INT7	External Interrupt Request 7
10	\$0012	PCINT0	Pin Change Interrupt Request 0
11	\$0014	PCI NT1	Pin Change Interrupt Request 1
12	\$0016	PCINT2	Pin Change Interrupt Request 2
13	\$0018	WDT	Watchdog Time-out Interrupt
14	\$001A	TIMER2 COMPA	Timer/Counter2 Compare Match A
15	\$001C	TIMER2 COMPB	Timer/Counter2 Compare Match B
16	\$001E	TIMER2 OVF	Timer/Counter2 Overflow
17	\$0020	TIMER1 CAPT	Timer/Counter1 Capture Event
18	\$0022	TIMER1 COMPA	Timer/Counter1 Compare Match A
19	\$0024	TIMER1 COMPB	Timer/Counter1 Compare Match B
20	\$0026	TIMER1 COMPC	Timer/Counter1 Compare Match C
21	\$0028	TIMER1 OVF	Timer/Counter1 Overflow
22	\$002A	TIMER0 COMPA	Timer/Counter0 Compare Match A
23	\$002C	TIMER0 COMPB	Timer/Counter0 Compare match B
24	\$002E	TIMER0 OVF	Timer/Counter0 Overflow

25	\$0030	SPI, STC	SPI Serial Transfer Complete
26	\$0032	USART0 RX	USART0 Rx Complete
27	\$0034	USART0 UDRE	USART0 Data Register Empty
28	\$0036	USART0 TX	USART0 Tx Complete
29	\$0038	ANALOG COMP	Analog Comparator
30	\$003A	ADC	ADC Conversion Complete
31	\$003C	EE READY	EEPROM Ready
32	\$003E	TIMER3 CAPT	Timer/Counter3 Capture Event
33	\$0040	TIMER3 COMPA	Timer/Counter3 Compare Match A
34	\$0042	TIMER3 COMPB	Timer/Counter3 Compare Match B
35	\$0044	TIMER3 COMPC	Timer/Counter3 Compare Match C
36	\$0046	TIMER3 OVF	Timer/Counter3 Overflow
37	\$0048	USART1 RX	USART1 Rx Complete
38	\$004A	USART1 UDRE	USART1 Data Register Empty
39	\$004C	USART1 TX	USART1 Tx Complete
40	\$004E	TWI	2-wire Serial Interface
41	\$0050	SPM READY	Store Program Memory Ready
42	\$0052	TIMER4 CAPT	Timer/Counter4 Capture Event
43	\$0054	TIMER4 COMPA	Timer/Counter4 Compare Match A
44	\$0056	TIMER4 COMPB	Timer/Counter4 Compare Match B

45	\$0058	TIMER4 COMPC	Timer/Counter4 Compare Match C
46	\$005A	TIMER4 OVF	Timer/Counter4 Overflow
47	\$005C	TIMER5 CAPT	Timer/Counter5 Capture Event
48	\$005E	TIMER5 COMPA	Timer/Counter5 Compare Match A
49	\$0060	TIMER5 COMPB	Timer/Counter5 Compare Match B
50	\$0062	TIMER5 COMPC	Timer/Counter5 Compare Match C
51	\$0064	TIMER5 OVF	Timer/Counter5 Overflow
52	\$0066	USART2 RX	USART2 Rx Complete
53	\$0068	USART2 UDRE	USART2 Data Register Empty
54	\$006A	USART2 TX	USART2 Tx Complete
55	\$006C	USART3 RX	USART3 Rx Complete
56	\$006E	USART3 UDRE	USART3 Data Register Empty
57	\$0070	USART3 TX	USART3 Tx Complete

Tomado de Atmel, 2014, p.101-102

**Nota.** La descripción del uso de los distintos vectores de interrupción puede verificarse con detalle en el manual del microcontrolador ATmega2560 (Atmel, 2014).

**Tabla 4.2**

*Interrupciones y pines asociados*

Vectores de interrupción	Pines asociados a cada interrupción	Pin físico	Registros de configuración asociados
INT0	INT0	PD0	DDRD
INT1	INT1	PD1	PORTD
INT2	INT2	PD2	EICRA
INT3	INT3	PD3	EIMSK EIFR
INT4	INT4	PE4	DDRE
INT5	INT5	PE5	PORTE
INT6	INT6	PE6	EICRB
INT7	INT7	PE7	EIMSK EIFR
PCINT0	PCINT0	PB0	
	PCINT1	PB1	DDRB
	PCINT2	PB2	PORTB
	PCINT3	PB3	PCICR
	PCINT4	PB4	PCIFR
	PCINT5	PB5	PCMSK0
	PCINT6	PB6	
	PCINT7	PB7	
PCINT1	PCINT8	PE0	DDRE
	PCINT9	PJ0	PORTE
	PCINT10	PJ1	DDRJ
	PCINT11	PJ2	PORTJ
	PCINT12	PJ3	PCICR
	PCINT13	PJ4	PCIFR
	PCINT14	PJ5	PCMSK1
	PCINT15	PJ6	
PCINT2	PCINT16	PK0	DDRK
	PCINT17	PK1	PORTK
	PCINT18	PK2	PCICR
	PCINT19	PK3	PCIFR
	PCINT20	PK4	PCMSK2
	PCINT21	PK5	

## 4.2 Interrupciones externas

Las interrupciones externas están asociadas a los pines del microcontrolador y se dividen en dos clases: INT y PCINT. Más adelante, presentamos sus detalles. Como consta en la tabla 4.1, las interrupciones externas son aquellas asociadas a las direcciones de memoria desde \$0002 hasta la \$0016. La tabla 4.2 detalla los pines que pueden generar eventos para estas interrupciones y los registros asociados con su configuración.

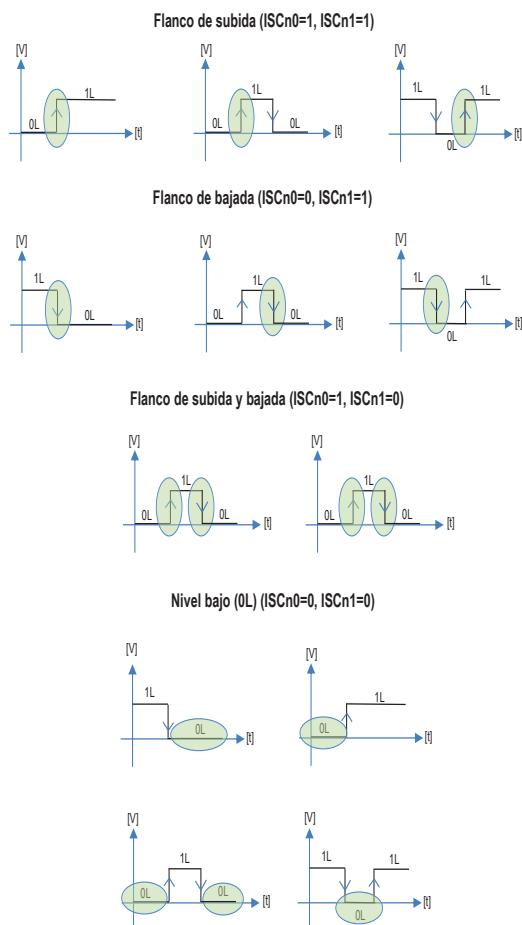
PCINT22	PK6
PCINT23	PK7

#### 4.2.1 Interrupciones tipo INT

Estas interrupciones se caracterizan por permitir al programador configurarlas para que el microcontrolador perciba en los pines INT las siguientes acciones o eventos externos: flancos de subida, flancos de bajada, estado cero y flanco de subida/bajada. En la figura 4.1 se presentan en verde los eventos que son válidos para generar una interrupción, dependiendo de la configuración realizada en los registros EICRA e EICRB, cuyos contenidos bit a bit se pueden observar en las figuras 4.2 y 4.3, respectivamente.

**Figura 4.1**

Eventos configurables para una interrupción externa tipo INT



**Figura 4.2**

Bits de configuración del registro EICRA

bit	7	6	5	4	3	2	1	0
Acción	R/W							
Estado inicial	0	0	0	0	0	0	0	0

Tomado de Atmel, 2014, p. 110

**Figura 4.3**

Bits de configuración del registro EICRB

bit	7	6	5	4	3	2	1	0
Acción	R/W							
Estado inicial	0	0	0	0	0	0	0	0

Tomado de Atmel, 2014, p. 111

Las configuraciones en los registros EICRA e EICRB permiten establecer el evento que el microcontrolador utiliza para generar una interrupción. Como indica la tabla 4.2, existen 8 interrupciones tipo INT que tienen 4 diferentes configuraciones de eventos por cada una. Estas configuraciones se logran a partir de la combinación de estados en dos bits, como puede verse en la tabla 4.3. La figura 4.4 muestra el significado de los bits en los registros EICRA e EICRB.

**Tabla 4.3**

Configuración de eventos para generación de interrupciones

ISCn1	ISCn0	Comportamiento
0	0	Solicitud de interrupción en nivel bajo
0	1	Cualquier flanko genera una solicitud de interrupción (subida y bajada)
1	0	Solicitud de interrupción en flanko de bajada
1	1	Solicitud de interrupción en flanko de subida

Tomado de Atmel, 2014, p. 110

Figura 4.4

*Bits de configuración del registro EICRB y EICRA*



Si bien con los registros EICRA e EICRB es posible establecer el evento que puede generar una interrupción en un PIN asociado a un INT (0,1,2 ... 7), para que esta interrupción se ejecute, es necesario habilitarla en el microcontrolador, por medio del registro EIMSK (ver figura 4.5). Además, hay que recordar que, para que cualquier interrupción pueda ser ejecutada en el microcontrolador, el bit global de habilitación de interrupciones (I) en el registro SREG (registro de estado AVR) debe ser colocado en 1L (ver figura 4.6).

Figura 4.5

*Bits de configuración en el registro EIMSK*

bit	7	6	5	4	3	2	1	0
Acción	R/W							
Estado inicial	0	0	0	0	0	0	0	0

Tomado de Atmel, 2014, p. 111

Figura 4.6

*Bit 7 encargado de la habilitación global de interrupciones en el registro SREG*

bit	7	6	5	4	3	2	1	0
Acción	I	T	H	S	V	N	Z	C
Estado inicial	0	0	0	0	0	0	0	0

Tomado de Atmel, 2014, p. 13

Un concepto que se debe tener en cuenta cuando se trabaja con interrupciones es el manejo y la interpretación de las banderas de aviso de evento

(*flags*). Estos bits –que pueden tomar valores de 0L o 1L, dependiendo de su estado–, permiten que el microcontrolador pueda dirigirse hacia el vector de interrupción asociado; es decir, informan al MCU que un evento externo ha sucedido. En el caso de las INT, esta información se encuentra en el registro EIFR (figura 4.7). Para este registro, cada bit puede ser de escritura o lectura. Cada vez que un evento para un INT específico se detecta, un 1L es colocado en el bit asociado. Este estado de 1L se encerra una vez que la rutina de interrupción es ejecutada o de manera manual, reescribiendo un 1L.

Figura 4.7

*Banderas de INT en el registro EIFR*

bit	7	6	5	4	3	2	1	0
Acción	R/W							
Estado inicial	0	0	0	0	0	0	0	0

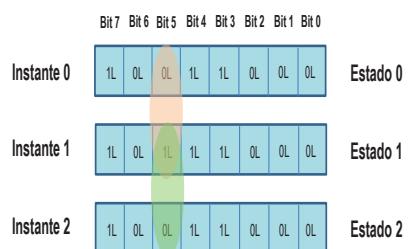
Tomado de Atmel, 2014, p. 112

#### 4.2.2 Interrupciones tipo PCINT

A diferencia de las anteriores, en las interrupciones PCINT no es posible configurar los eventos que producen una interrupción. Otra diferencia es la cantidad: existen 24 pines del microcontrolador que pueden ser usados como PCINT; sin embargo, solo existen 3 vectores de interrupción asociados (ver tabla 4.2). Las PCINT son llamadas, normalmente, como interrupciones por cambio de estado, pues para entender su funcionamiento, es necesario pensar en su comportamiento en el tiempo.

Figura 4.8

*Interpretación de cambios de estado en el tiempo de un grupo de bits*



Supóngase que se quiere establecer los cambios de estado en un grupo de bits en tres instantes de tiempo (figura 4.8). El cambio de estado no se refiere solo al cambio en el valor de un bit; también alude al cambio de estado de uno o un grupo de bits entre un instante  $n$  y un instante  $n-1$ . En el ejemplo planteado, entre el instante 1 y el instante 0 hubo una variación en el bit 5, lo que provoca que el grupo de bits cambie su estado respecto del instante anterior. En el instante 2, el bit 5 vuelve a su valor inicial lo que provoca que el grupo de bits vuelva a su valor original; no obstante, también se considera un cambio de estado, pero esta vez con respecto al instante 1.

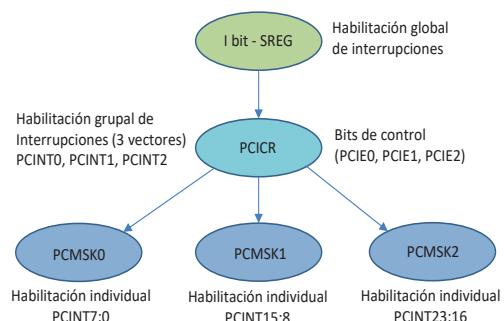
Esta aclaración sobre el significado de un cambio de estado es necesaria, pues en muchas ocasiones se piensa que si un bit cambia de 0L a 1L, o vice-versa, se tiene un cambio de flanco, confundiéndose con los conceptos manejados en interrupciones INT, donde sí se trabaja con cambios de flanco.

Entrando ya en detalle de los registros de configuración de interrupciones tipo PCINT, se identifican tres grupos: los registros de habilitación individual para cada uno de los 24 pines asociados, los registros de habilitación grupal asociados a los tres vectores de interrupción y los registros asociados a las banderas de aviso de evento para cada uno de los tres vectores de interrupción.

La figura 4.9 presenta un diagrama esquemático de los registros de habilitación asociados con las interrupciones tipo PCINT. En cambio, la figura 4.10 muestra la agrupación de los pines con funcionalidad PCINT y su relación con los vectores de interrupción.

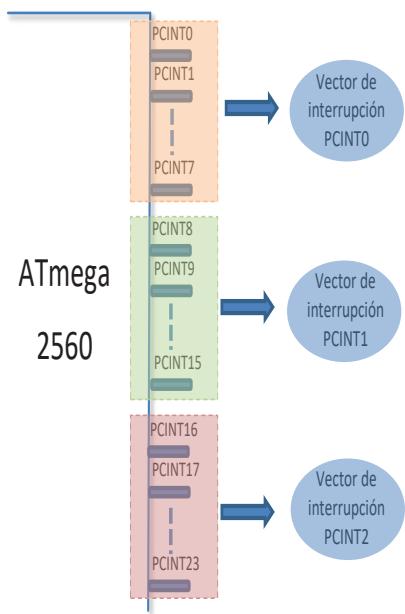
**Figura 4.9**

Registros de habilitación para interrupciones tipo PCINT



**Figura 4.10**

Representación de grupos de pines con función PCINT asociados a los vectores de interrupción



El registro PCICR es el encargado de habilitar la interrupción de cada uno de los grupos. En la figura 4.11 se detallan los bits que permiten habilitar cada uno de los grupos representados en la figura 4.10. De esta manera, los bits PCIE0, PCIE1, PCIE2 habilitan los grupos cuyos eventos son atendidos por los vectores de interrupción PCINT0, PCINT1 y PCINT2.

**Figura 4.11**

Registro PCICR y bits asociados con la habilitación de los grupos para los vectores de interrupción PCINT0, PCINT1 y PCINT2

bit	7	6	5	4	3	2	1	0
	-	-	-	-	-	PCIE2	PCIE1	PCIE0
Acción	R	R	R	R	R	R/W	R/W	R/W
Estado inicial	0	0	0	0	0	0	0	0

Tomado de Atmel, 2014, p.112

Una vez habilitado un grupo, es necesario establecer los pines específicos que podrán generar un evento para ser atendido por un vector de interrupción. Los registros de habilitación individual para cada grupo PCIE0, PCIE1, PCIE2 se muestran en las figuras 4.12, 4.13 y 4.14, respectivamente.

**Figura 4.12**

Registro PCMSK0 y bits para la habilitación individual de pines por cambio de estado

bit	7	6	5	4	3	2	1	0
	PCINT7	PCINT6	PCINT5	PCINT4	PCINT3	PCINT2	PCINT1	PCINT0
Acción	R/W							
Estado inicial	0	0	0	0	0	0	0	0

Tomado de Atmel, 2014, p. 114

**Figura 4.13**

Registro PCMSK1 y bits para la habilitación individual de pines por cambio de estado

bit	7	6	5	4	3	2	1	0
	PCINT1	PCINT1	PCINT1	PCINT1	PCINT1	PCINT1	PCINT	PCINT
Acción	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Estado inicial	0	0	0	0	0	0	0	0

Tomado de Atmel, 2014, p. 113

**Figura 4.14**

Registro PCMSK2 y bits para la habilitación individual de pines por cambio de estado

bit	7	6	5	4	3	2	1	0
	PCINT23	PCINT22	PCINT21	PCINT20	PCINT19	PCINT18	PCINT17	PCINT16
Acción	R/W							
Estado inicial	0	0	0	0	0	0	0	0

Tomado de Atmel, 2014, p. 113

**Nota.** La nomenclatura de PCINT0, PCINT1 y PCINT2 se utiliza para definir los 3 vectores de interrupción, así como los bits asociados al comportamiento de un pin físico. Sin embargo, su funcionalidad es totalmente diferente, según el caso. Esta característica puede observarse de manera clara en la tabla 4.2.

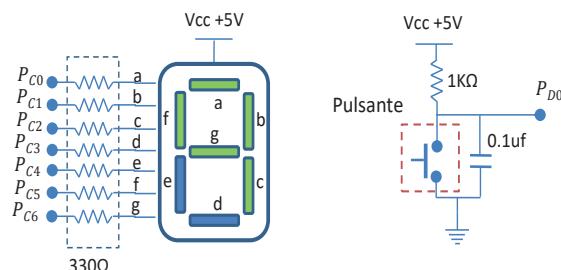
### 4.3 Ejercicios prácticos de configuración de interrupciones externas

Con el fin de entender, de mejor manera, el funcionamiento y la configuración de los distintos registros que permiten la activación y el funcionamiento de interrupciones externas, presentamos una serie de casos para profundizar, progresivamente, los conceptos tratados en este capítulo.

**Ejercicio 4.1:** Se desea implementar un contador BCD 0-9 ascendente en un *display* de 7 segmentos tipo ánodo común. El conteo debe incrementarse mediante un pulsante conectado al pin PDO (INT0). Se considera que la acción de conteo debe incrementarse cuando el pulsante sea soltado (flanco de subida). El esquema de conexión del *display* y la conexión del pulsante se muestran en la figura 4.15.

**Figura 4.15**

Esquema de conexiones para display ánodo común. Ejercicio 4.1



**Nota.** Para resolver este ejercicio, se asume que no se habilita la resistencia interna de pull-up del pin PDO.

#### Solución:

Los registros para configurar se muestran en la figura 4.16.

**Figura 4.16**

Configuración de registros E/S. Ejercicio 4.1

bit	7	6	5	4	3	2	1	0
DDRC	x	1	1	1	1	1	1	1
bit	7	6	5	4	3	2	1	0
DDRD	x	x	x	x	x	x	x	0
bit	7	6	5	4	3	2	1	0
PORTD	x	x	x	x	x	x	x	0

Como se puede apreciar en los registros, primero se establece cada pin, según su funcionamiento como E/S. En el caso de los registros DDRC, DDRD y PORTD, los bits que no se van a utilizar como convención se encuentran marcados con **X**. Esto se debe a que se asume como desconocido su estado anterior, y que para cambiar el valor del bit deseado se aplican los conceptos sobre enmascaramiento, desarrollados en el capítulo 3.

En código, se escribiría de la siguiente manera:

```
DDRC |= 0b01111111;
DDRD &= 0b11111110;
PORTD &= 0b11111110;
```

Máscaras

**Nota.** El registro PORTD se configura para asegurar que no sea habilitada la resistencia de *pull-up* interna del microcontrolador para el pin PD0.

Una vez establecidas las configuraciones de E/S, se deben configurar las interrupciones externas por cambio de flanco. De acuerdo con lo solicitado en el ejercicio, la interrupción que debe ser activada es la INT0 (figura 4.17) y debe ser configurada como flanco de subida, con base en la tabla 4.3.

**Figura 4.17**

Configuración de registros interrupciones. Ejercicio 4.1

bit	7	6	5	4	3	2	1	0
EICRA	0	0	0	0	0	0	1	1
EIMSK	0	0	0	0	0	0	0	1

Esta configuración de registros puede ser escrita en código de la siguiente manera:

```
EICRA=0b00000011;
EIMSK=0b00000001;
```

Una vez que los registros se hayan configurado para habilitar y configurar las interrupciones deseadas, es necesario crear las rutinas de servicio de interrupción (ISR). Estas son funciones que se ejecutan cuando un evento de interrupción se ha presentado. En el caso del vector de interrupción asociado al INT0, se tendría lo siguiente:

```
ISR(INT0_vect)
{
    //código a ser ejecutado en caso de evento externo
}
```

Finalmente, el código para el ejercicio propuesto sería el siguiente:

```
#include <avr/io.h>
#include <avr/interrupt.h>
unsigned char cont=0;
unsigned char anodo_comun[16]={0xC0,0xF9,0xA4,0XB0,0x99,
0x92,0x82,0xF8,0x80,0x90,0X88,0X83,0XC6,0XA1,0X86,0X8E};

ISR(INT0_vect)
{
    cont++;
}

int main(void)
{
    DDRC|=0b01111111;
    DDRD &= 0b11111110;
    PORTD &= 0b11111110;
    EICRA=0b00000011;
    EIMSK=0b00000001;
    sei();
    while(1)
    {
        PORTC=(PORTC&0x80)|anodo_comun[cont];           // se envía el código de display
        // sin alterar el PC7
    }
}
```

**Nota.** Las ISR están asociadas directamente a los vectores de interrupción detallados en la tabla 4.1.

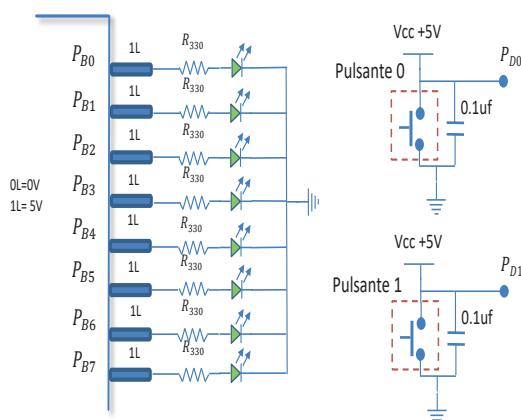
**Nota.** En el código siempre deberá ser anexada una librería de interrupciones (avr/interrupt.h), para que los vectores de interrupción puedan ser reconocidos.

**Nota.** En caso de utilizar interrupciones, siempre es necesario habilitar el bit 7 (I) en el registro SREG. Para mayor facilidad, esto se puede realizar mediante la instrucción sei ()�.

**Ejercicio 4.2:** Se dispone de 8 ledes conectados al pótico B (salidas), los cuales deben encenderse y apagarse. Cada vez que se presione un pulsante conectado al pin PD0 (INT0), los ledes deben encenderse. Para apagar los ledes se emplea otro pulsante conectado al pin PD1 (INT1). La acción de encendido debe ser ejecutada en el momento que el pulsante sea soltado (flanco de subida), mientras que el apagado de los ledes debe ser ejecutado al presionar el pulsante (flanco de bajada). El esquema de conexiones se exhibe en la figura 4.18.

**Figura 4.18**

Esquema de conexiones. Ejercicio 4.2



**Nota.** Para resolver este ejercicio, se asume que la resistencia interna de *pull-up* del pin PD0 y PD1 se encuentra habilitada. Por tanto, no es necesario colocar una resistencia externa.

### Solución:

Los registros para configurar se muestran en la figura 4.19.

**Figura 4.19**

Configuración de registros E/S. Ejercicio 4.2

bit	7	6	5	4	3	2	1	0
DDRB	1	1	1	1	1	1	1	1
PORTB	0	0	0	0	0	0	0	0
bit	7	6	5	4	3	2	1	0
DDRD	x	x	x	x	x	x	0	0
bit	7	6	5	4	3	2	1	0
PORTD	x	x	x	x	x	x	1	1

En este caso, nótese que en el pótico B todos sus bits se colocan en cero. Esto indica que en el estado inicial del sistema todos los ledes están apagadas. Además, los registros DDRD y PORTD están configurados para que los pines PD0 y PD1 se encuentren como entradas, con sus respectivas resistencias de *pull-up* internas habilitadas.

Descrito en código, sería de la siguiente manera:

```
DDRB=255;
PORTB=0;
DDRD &=0b11111100;
PORTD |=0b00000011;
```

En el caso de los registros para la configuración y habilitación de las interrupciones, se debe tener en cuenta la tabla 4.3, con el fin de establecer los bits para configurar flancos de subida y bajada para los eventos en INT0 e INT1, respectivamente. La configuración de los registros se presenta en la figura 4.20.

**Figura 4.20**

Configuración de registros interrupciones. Ejercicio 4.2

bit	7	6	5	4	INT1 (flanco de bajada)		INT0 (flanco de subida)		
	EICRA	0	0	0	0	1	0	1	1
bit	7	6	5	4	3	2	1	0	
EIMSK	0	0	0	0	0	0	1	1	

Observar que los bits de configuración asociados a cada interrupción (EICRA) estén dados en pares y su combinación depende del tipo de evento que se requiere registrar, con base en la tabla 4.3. En el registro EIMSK se establece qué interrupciones van a ser habilitadas; en este caso, la INT0 (bit0) y la INT1 (bit1).

Finalmente, el código sería el siguiente:

```
#include <avr/io.h>
#include <avr/interrupt.h>
ISR(INT0_vect)
{
    PORTB=255;
}
ISR()
{
    PORTB=0;
}
int main(void)
{
    DDRB=255;
    PORTB=0;
    DDRD &=0b11111100;
    PORTD |=0b00000011;
    EICRA=0b00001011;
    EIMSK=0b00000011;
    sei();
    while (1)
    {
    }
}
```

Nótese que, en el caso de estar habilitadas dos interrupciones (INT0, INT1), se deben incluir dos ISR, uno por cada interrupción habilitada.

**Nota.** Por cada interrupción habilitada, debe existir obligatoriamente su ISR asociado.

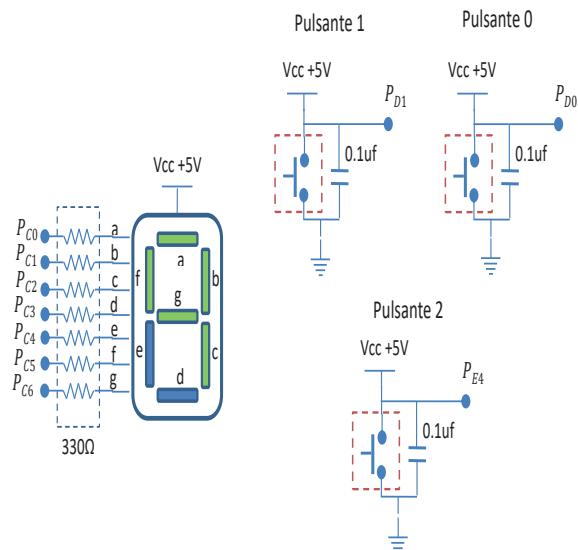
**Ejercicio 4.3:** Ahora, analicemos el caso del uso de tres interrupciones externas por cambio de flanco. Para tal efecto, se considera un contador BCD 0-9 ascendente y descendente, con opción de encerado. Para cumplir con este objetivo, se fijan 3 pulsantes: uno para aumentar la cuenta (INT0), un segundo para disminuir la cuenta (INT1) y el tercer pulsante (INT4) para encerar la cuenta. Para mostrar la cuenta se utiliza un *display* de 7 segmentos de ánodo común. Los eventos en cada pulsante serán detectados en flancos de bajada y se considera que se utilizarán *pull-up* internos del microcontrolador para conectar los pulsantes.

### Solución:

La figura 4.21 muestra el esquema de conexiones.

**Figura 4.21**

Esquema de conexiones. Ejercicio 4.3



Los registros para configurar se presentan en la figura 4.22.

**Figura 4.22**

Configuración de registros E/S. Ejercicio 4.3

bit	7	6	5	4	3	2	1	0
DDRC	X	1	1	1	1	1	1	1

bit	7	6	5	4	3	2	1	0
DDRD	X	X	X	X	X	X	0	0

bit	7	6	5	4	3	2	1	0
DDRE	X	X	X	0	X	X	X	X

bit	7	6	5	4	3	2	1	0
PORTD	X	X	X	X	X	X	1	1

bit	7	6	5	4	3	2	1	0
PORTE	X	X	X	1	X	X	X	X

En código, se tiene:

```
DDRC |=0b1111111;
DDRD &=0b1111110;
DDRE &=0b1110111;
PORTD |=0b00000011;
PORTE |=0b00010000;
```

Nótese las máscaras tipo OR y tipo AND, y que además se utilizan los registros PORTD y PORTE para establecer resistencias de *pull-up* internas.

En el caso de la habilitación y configuración de interrupciones, se tienen los siguientes registros:

**Figura 4.23**

Configuración de registros interrupción. Ejercicio 4.3

bit	7	6	5	4	3	2	1	0
EICRA	0	0	0	0	1	0	1	0
EICRB	0	0	0	0	0	0	1	0
EIMSK	0	0	0	1	0	0	1	1

INT1 (flanco de bajada)  
INT4 (flanco de bajada)  
INT2 (flanco de bajada)  
INT3 (flanco de bajada)  
INT0 (flanco de bajada)

El código asociado sería:

```
EICRA=0b00001010;
EICRB=0b00000010;
EIMSK=0b00010011;
```

Finalmente, el código funcional para este ejercicio sería el siguiente:

```
#include <avr/io.h>
#include <avr/interrupt.h>
unsigned char cont=0;
unsigned char anodo_comun[16]={0xC0,0xF9,0xA4,0XB0,0x99,
0x92,0x82,0xF8,0x80,0x90,0X88,0X83,0XC6,0XA1,0X86,0X8E};
ISR(INT0_vect)
{
    cont++;
}
ISR(INT1_vect)
{
    cont--;
}
ISR(INT4_vect)
{
    cont=0;
}

int main(void)
{
    DDRC |=0b1111111;
    DDRD &=0b1111110;
    DDRE &=0b1110111;
    PORTD |=0b00000011;
    PORTE |=0b00010000;
    EICRA=0b00001010;
    EICRB=0b00000010;
    EIMSK=0b00010011;
    sei();
    while (1)
    {
        PORTC=(PORTC&0x80)|anodo_comun[cont]; // se envía el código de
                                                // Display sin alterar el PC7
    }
}
```

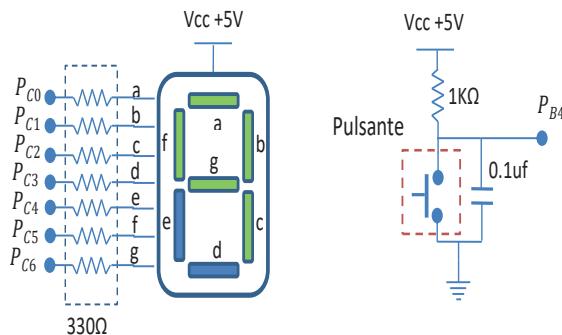
Al incluir la interrupción INT4, es necesario insertar el ISR correspondiente. Al respecto, es importante advertir que no existe una restricción cuando la variable de conteo “cont” supera el número 9 o desciende por debajo de 0. Esto puede resolverse mediante comparación de límites a través del uso de estructuras tipo “if”.

Los siguientes ejercicios retoman los ejemplos 4.1, 4.2 y 4.3 y aplican interrupciones por cambio de estado. De esta manera, podemos apreciar las diferencias entre los dos tipos de interrupciones.

**Ejercicio 4.4:** Se desea implementar un contador BCD 0-9 ascendente en un *display* de 7 segmentos tipo ánodo común. El conteo debe incrementarse por cada pulsación del usuario, por medio de un pulsante conectado al pin PB4 (PCINT4). El esquema de conexión del *display* y la conexión del pulsante se muestran en la figura 4.24.

**Figura 4.24**

Esquema de conexiones. Ejercicio 4.4



**Nota.** Para resolver este ejercicio, se asume que no se habilita la resistencia interna de *pull-up* del pin PB4.

### Solución:

Los registros para configurar se representan en la figura 4.25.

**Figura 4.25**

Configuración de registros E/S. Ejercicio 4.4

bit	7	6	5	4	3	2	1	0
DDRC	x	1	1	1	1	1	1	1
DDRB	x	x	x	0	x	x	x	x
PORTB	x	x	x	0	x	x	x	x

En código, se escribiría de la siguiente manera:

```
DDRC |= 0b01111111;
DDRB &= 0b11101111;
PORTB &= 0b11101111;
```

Máscaras

**Nota.** El registro PORTB se configura para asegurar que no sea habilitada la resistencia de *pull-up* interna del microcontrolador para el pin PB4.

Una vez establecidas las configuraciones de E/S, es necesario configurar las interrupciones externas por cambio de estado. De acuerdo con lo solicitado en el ejercicio, se debe habilitar la interrupción asociada al pin PB4 (PCINT4) (ver tabla 4.2). La figura 4.26 presenta los registros que deben ser modificados.

**Figura 4.26**

Configuración de registro PCICR. Ejercicio 4.4

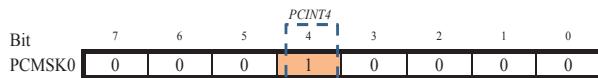
bit	7	6	5	4	3	2	1	0
PCICR	0	0	0	0	0	0	0	1

El registro PCICR es el encargado de habilitar cada uno de los 3 posibles vectores de interrupción asociados a los eventos por cambio de estado. En este caso, el pin asociado al PCINT4 corresponde al grupo que se encuentra comandado por el vector de interrupción PCINT0.

Luego de habilitar un vector de interrupción, se deben establecer los pines correspondientes a ese grupo, que podrán generar un evento de cambio de estado. Al tratarse del grupo referente al vector de interrupción PCINT0, el registro a modificar es el PCMSK0.

**Figura 4.27**

Configuración de registro PCMSK0. Ejercicio 4.4



Esta configuración de registros puede ser escrita en código, de la siguiente manera:

```
PCICR=0b00000001;
PCMSK0=0b00010000;
```

Una vez que los registros se hayan configurado para habilitar y configurar las interrupciones deseadas, es necesario crear las ISR, que atenderán los eventos por cambio de estado. En el caso del vector de interrupción PCINT0 asociado al pin PCINT4, se tendría lo siguiente:

```
ISR(PCINT0_vect)
{
    //código a ser ejecutado en caso de evento externo
}
```

Finalmente, el código para el ejercicio propuesto sería el siguiente:

```
#include <avr/io.h>
#include <avr/interrupt.h>
unsigned char cont=0;
unsigned char anodo_comun[16]={0xC0,0xF9,0xA4,0XB0,0x99,
0x92,0x82,0xF8,0x80,0x90,0X88,0X83,0XC6,0XA1,0X86,0X8E};
ISR(PCINT0_vect)
{
    cont++;
}
int main(void)
{
    DDRD|=0b01111111;
    DDRB &=0b11101111;
    PORTB &=0b11101111;
    PCICR=0b00000001;
    PCMSK0=0b00010000;
    while(1)
    {
        PORTC=(PORTC&0x80)|anodo_comun[cont];      // se envía el código de
                                                       // Display sin alterar el PC7
    }
}
```

En el código propuesto, se aprecia que cada vez que ocurre un evento el contador aumenta su valor en 1. Hay que considerar que cuando un pulsante es presionado, el estado cambia de 1L a 0L y cuando este es liberado, el estado pasa de 0L a 1L; así, cuando una persona presiona el pulsante y lo suelta se registran dos eventos de cambio de estado. En el ejemplo planteado, esto implicaría que el contador aumente en 2 cada vez que el usuario presione y suelte el pulsante.

Si se desea que el contador aumente en 1 cada vez que el usuario presione y suelte el pulsante, se debe efectuar un cambio en el código propuesto para la ISR.

```

ISR(PCINT0_vect)
{
    if ((PINB&0b00010000)==0)
    {
        cont++;
    }
}

```

En el código propuesto se incluye una sentencia “if”, que se encarga de testear el estado del pin PB4 (PCINT4) y que cada vez que se ingrese al ISR solo se incrementa el contador si el pulsante está presionado, es decir, en estado de 0L. Si se ingresa al ISR, pero el PB4 está en 1L, el contador no se incrementará. De igual manera, se puede aplicar la lógica de manera inversa, cambiando la condición en la sentencia “if”.

```

ISR(PCINT0_vect)
{
    if ((PINB&0b00010000)==16)
    {
        cont++;
    }
}

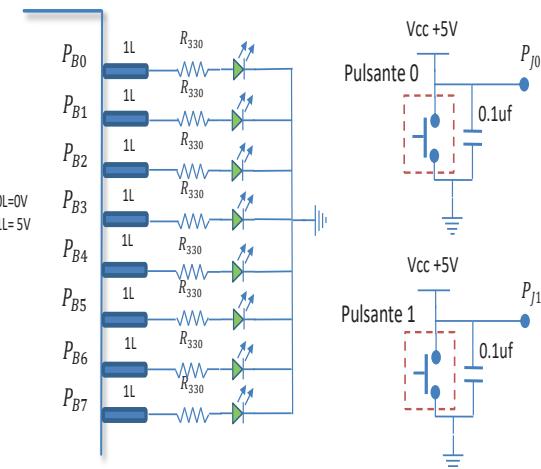
```

**Nota.** El uso de PCINT implica cambios de estado. Hay que poner atención al evento deseado para ejecutar una acción determinada.

**Ejercicio 4.5:** Se dispone de 8 ledes conectadas al pótico B (salidas), que deben encenderse y apagarse. Cada vez que se presione un pulsante conectado al pin PJ0 (PCINT9), las ledes deben encenderse. Para apagar los ILEDs se utiliza otro pulsante conectado al pin PJ1 (PCINT10). La acción de encendido y apagado debe ser ejecutada en el momento que el pulsante es presionado. La figura 4.28 ilustra el esquema de conexiones.

Figura 4.28

Esquema de conexiones. Ejercicio 4.5



**Solución:**

Los registros para configurar se presentan en la figura 4.29.

Figura 4.29

Configuración de registros E/S. Ejercicio 4.5

bit	7	6	5	4	3	2	1	0
DDRB	1	1	1	1	1	1	1	1
bit	7	6	5	4	3	2	1	0
DDRJ	x	x	x	x	x	x	0	0
bit	7	6	5	4	3	2	1	0
PORTJ	x	x	x	x	x	x	1	1

En código, se escribiría de la siguiente manera:

```

DDRB=255;
PORTB=0;
DDRJ &=0b11111100;
PORTJ|=0b00000011;
} } Máscaras

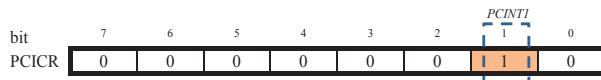
```

**Nota.** Las configuraciones de los pines PJ0 y PJ1 tienen activadas *pull-up* internas.

Una vez establecidas las configuraciones de E/S, es necesario configurar las interrupciones externas por cambio de estado. De acuerdo con lo solicitado en el ejercicio, es necesario habilitar la interrupción asociada a los pines PJ0 (PCINT9) y PJ1 (PCINT10), (ver tabla 4.2). Los registros que deben ser modificados se encuentran a continuación (ver figuras 4.30 y 4.31).

**Figura 4.30**

Configuración de registro PCICR. Ejercicio 4.5

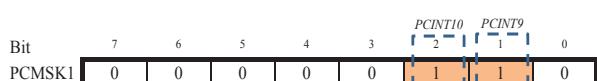


El registro PCICR habilita cada uno de los 3 posibles vectores de interrupción asociados a los eventos por cambio de estado. En este caso, los pines asociados a PCINT9 y PCINT10 corresponden al grupo que se encuentra comandado por el vector de interrupción PCINT1.

Luego de habilitar un vector de interrupción, es necesario establecer qué pines correspondientes a ese grupo podrán generar un evento de cambio de estado. Al tratarse del grupo correspondiente al vector de interrupción PCINT1, el registro a modificar es el PCMSK1 (ver figura 4.31).

**Figura 4.31**

Configuración de registro PCMSK1. Ejercicio 4.5



Esta configuración de registros puede ser escrita en código, de la siguiente manera:

```
PCICR=0b00000010;
PCMSK1=0b00000110;
```

Una vez que los registros se hayan configurado para habilitar y configurar las interrupciones deseadas, es necesario crear las ISR, que servirán para atender los eventos por cambio de estado. En el caso del vector de interrupción PCINT1 asociado al pin PCINT9 y PCINT10, se tendría lo siguiente:

```
ISR(PCINT1_vect)
{
    //código a ser ejecutado en caso de evento externo
}
```

Finalmente, el código para el ejercicio es el siguiente:

```
#include <avr/io.h>
#include <avr/interrupt.h>
ISR(PCINT1_vect)
{
    if ((PINJ&0b00000001)==0)
    {
        PORTB=255;
    }
    if ((PINJ&0b00000010)==0)
    {
        PORTB=0;
    }
}
int main(void)
{
    DDRB=255;
    PORTB=0;
    DDRJ &=0b11111100;
    PORTJ |=0b00000011;
    PCICR=0b00000010;
    PCMSK1=0b00000110;
    sei();
    while (1)
    {
    }
}
```

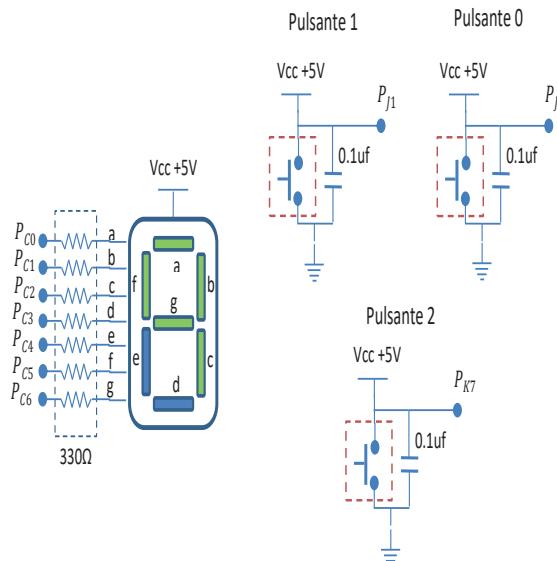
En el código desarrollado se debe prestar especial atención al ISR. Si bien se dispone de dos pines que pueden generar eventos por cambio de estado, estos están asociados a un solo vector de interrupción, en este caso, el PCINT1. Además, debido a que existe solo una ISR para analizar dos pines que generan eventos, se deben incluir condicionantes para discriminar cuál de los pines provocó la interrupción.

**Ejercicio 4.6:** Ahora, se examina el caso del uso de tres interrupciones externas por cambio de estado. Para tal efecto, se considera un contador BCD 0-9 ascendente y descendente, con opción de encerado. Para cumplir con este objetivo, se fijan 3 pulsantes: uno para aumentar la cuenta (PCINT9), otro para disminuir la cuenta (PCINT10) y un tercero (PCINT23) para encerar la cuenta. Para mostrar la cuenta, se utiliza un *display* de 7 segmentos de ánodo común. Los eventos en cada pulsante serán detectados cuando estos sean presionados; se considera que se utilizarán *pull-up* internos del microcontrolador para conectar los pulsantes.

La figura 4.32 muestra el esquema de conexiones.

**Figura 4.32**

*Esquema de conexiones. Ejercicio 4.6*



### Solución:

La figura 4.33 señala los registros para configurar.

**Figura 4.33**

*Configuración de registros E/S. Ejercicio 4.6*

bit	7	6	5	4	3	2	1	0
DDRC	x	1	1	1	1	1	1	1
DDRJ	x	x	x	x	x	x	0	0
DDRK	0	x	x	x	x	x	x	x
PORTJ	x	x	x	x	x	x	1	1
PORTK	1	x	x	x	x	x	x	x

En código, se tiene:

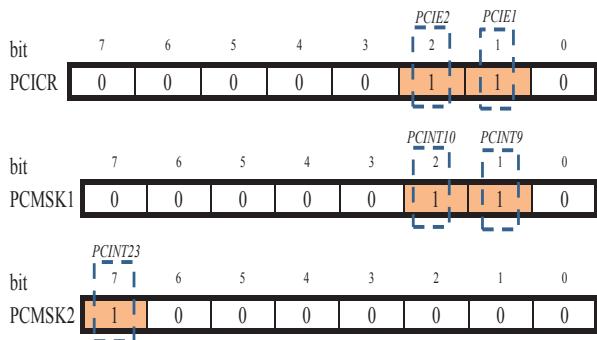
```
DDRC |= 0b01111111;
DDRJ &= 0b11111100;
DDRK &= 0b01111111;
PORTJ |= 0b00000011;
PORTK |= 0b10000000;
```

Es importante observar las máscaras tipo OR y tipo AND, así como los registros PORTJ y PORTK utilizados para establecer resistencias de *pull-up* internas.

En el caso de la habilitación y configuración de interrupciones, los registros se muestran en la figura 4.34.

**Figura 4.34**

Configuración de registros interrupciones. Ejercicio 4.6



El código asociado sería:

```
PCICR=0b00000110;
PCMSK1=0b00000110;
PCMSK2=0b10000000;
```

Finalmente, el código funcional para este ejercicio es:

Como se puede apreciar en el código propuesto, si bien existen 3 pines que generan eventos por cambio de estado, se implementan solo 2 rutinas ISR, en las cuales se valida cada uno de los eventos.

```
#include <avr/io.h>
#include <avr/interrupt.h>
unsigned char cont=0;
unsigned char anodo_comun[16]={0xC0,0xF9,0xA4,0XB0,0x99,
0x92,0x82,0xF8,0x80,0x90,0X88,0X83,0XC6,0XA1,0X86,0X8E};

ISR(PCINT1_vect)
{
    if ((PINJ&0b00000001)==0)
    {
        cont++;
    }
    if ((PINJ&0b00000010)==0)
    {
        cont--;
    }
}

ISR(PCINT2_vect)
{
    if ((PINK&0b10000000)==0)
    {
        cont=0;
    }
}

int main(void)
{
    DDRC |=0b01111111;
    DDRJ &=0b11111100;
    DDRK &=0b01111111;
    PORTJ |=0b00000011;
    PORTK |=0b10000000;
    PCICR=0b00000110;
    PCMSK1=0b00000110;
    PCMSK2=0b10000000;
    sei();
    while (1)
    {
        PORTC=(PORTC&0x80)|anodo_comun[cont];      // se envía el código de
                                                       //Display sin alterar el PC7
    }
}
```



# Capítulo 5

## Comunicación serial

### 5.1 Anotaciones preliminares sobre comunicación serial

La comunicación serial en telecomunicaciones y ciencias de la computación es un procedimiento muy sencillo y se puede entender su funcionamiento como el envío y la recepción de datos un bit a la vez (ver figura 5.1). En cambio, la comunicación paralela es el proceso de enviar y recibir múltiples bits de datos a la vez, a través de canales paralelos (ver figura 5.2).

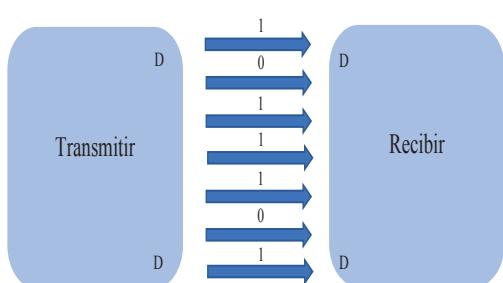
Figura 5.1

Transmisión serial



Figura 5.2

Transmisión paralela



Tomado de Maxembedded, 2014

Existen algunas diferencias entre este tipo de comunicaciones, que se resumen en la tabla 5.1.

Tabla 5.1

Diferencias entre serial y paralelo

Comunicación paralela	Comunicación serial
Múltiples bits de datos se transmiten a la vez	Un bit de datos es transmitido a la vez
Más rápido	Más lento
Mayor cantidad de cables	Menor cantidad de cables

Al observar los datos de la tabla 5.1, se podría concluir que la comunicación paralela es mejor que la comunicación serial, pero existen ciertos factores que limitan a esta comunicación. Entre los factores que más afectan la comunicación paralela están la longitud del cable y la diafonía, que crea interferencias entre las líneas paralelas y cuyo efecto es proporcional a la longitud del enlace. Esto reduce la distancia a la que se puede usar la comunicación paralela.

En cambio, la comunicación serial, que también es afectada por la longitud, presenta algunas ventajas, ya que usa menos cables; por lo tanto, la diafonía no es un problema muy significativo; también es más económica al implementar. Además, muchos sistemas microcontrolados usan comunicación serial por la menor cantidad de pines, por lo tanto, son menos costosos en comparación con la comunicación paralela.

Tomado de Maxembedded, 2014

## 5.2 Modos de transmisión serial

Los datos en serie pueden transferirse en dos modos: asíncrono y síncrono.

### 5.2.1 Transferencia asíncrona de datos

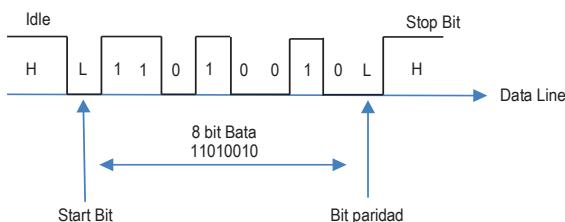
Cuando la transferencia de datos no está sincronizada con una línea de reloj, es decir, no hay línea de reloj en absoluto, se denomina asíncrona.

El primer bit es de inicio. Le siguen los bits de datos (generalmente, 8 bits) y uno o dos bits de detención o parada. Puede existir un bit de paridad antes del de detención. El bit de inicio siempre es bajo (0 Lógico), mientras que el bit de detención siempre será alto (1 Lógico) (ver figura 5.3).

Figura 5.3

Transferencia asíncrona de datos

Nivel bajo (0), Nivel alto (1), High (H), Low (L)



Tomado de Maxembedded, 2014

### 5.2.2 Transferencia síncrona de datos

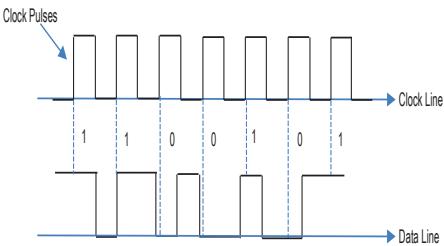
La transferencia de datos síncrona ocurre cuando los bits de datos están sincronizados con un pulso de reloj. El principio básico es que el muestreo de bits de datos se cuenta dependiendo de los pulsos de reloj; dado que las fuentes de reloj son muy confiables, hay menos error en síncrono que en asíncrono (ver figura 5.4).

Figura 5.4

Transferencia síncrona de datos

Los datos se muestran en el borde ascendente o descendente de los pulsos de reloj

Los datos se muestran en el borde ascendente o descendente de los pulsos de reloj



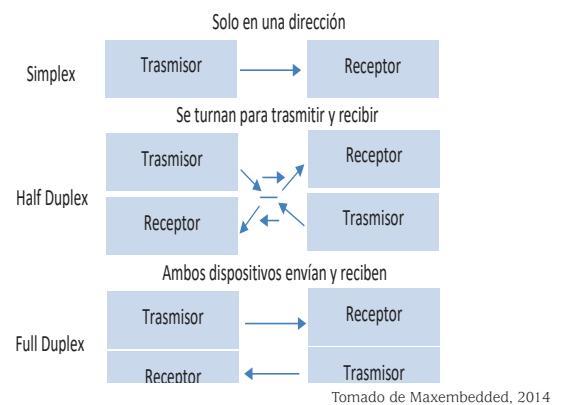
Tomado de Maxembedded, 2014

Como se observa en la figura 5.5, la comunicación puede ser *simplex* (solo en una dirección, sin que el dispositivo receptor pueda enviar información al dispositivo transmisor), *full duplex* (ambos dispositivos envían y reciben al mismo tiempo) o *half duplex* (los dispositivos se turnan para transmitir y recibir) (HETPRO, 2017).

Cuando se diseña una aplicación que requiere comunicación serial se pueden cometer errores básicos, como configurar al dispositivo en modo *simplex* solo enviando información. Lo recomendable es usar el modo *half duplex* para enviar información solo si lo pide uno de los elementos de la comunicación.

Figura 5.5

Modos de comunicación



Tomado de Maxembedded, 2014

**Nota.** La conexión es cruzada Rx con Tx, y viceversa, entre dos dispositivos.

### 5.2.3 Parámetros de la comunicación serial

La comunicación entre dos microcontroladores es el intercambio de datos (bits) en formato ASCII. En general, para realizar esta comunicación se utilizan dos líneas principales:

- Transmisión (Tx)
- Recepción (Rx)

Las características más importantes de la comunicación serial son:

- Velocidad de transmisión
- Bits de datos
- Bits de parada
- Paridad

**Baud rate (velocidad de transmisión).** Indica el número de bits o pulsos por segundo que se transfieren a través de un canal de datos. Se miden en baudios (*bauds*). Las velocidades más comunes son: 2 400, 4 800, 9 600, 19 200, 38 400. No es posible elegir ninguna velocidad de transmisión arbitraria; hay algunos valores fijos en ciertos dispositivos.

Por ejemplo, 9 600 baudios representan 9 600 bits por segundo (bit/s) para señales binarias.

Cuando se hace referencia a los ciclos de reloj, se está hablando de velocidad de transmisión. Esta depende del cristal del microcontrolador. Es posible tener velocidades altas cuando un dispositivo se encuentra a corta distancia.

Por ejemplo, si el protocolo hace una llamada a 4 800 ciclos de reloj, entonces el reloj está corriendo a 4 800 Hz, lo que significa que el puerto serial está muestreando las líneas de transmisión a 4 800 Hz.

**Bits de datos (bits de transmisión).** Se refiere a la cantidad de bits en la transmisión. Esta dependerá de la información que se transfiere. Cuando una computadora envía un paquete de información, el tamaño de ese paquete no necesariamente es de 8

bits. La cantidad más común de bits por paquete es de 5, 7, 8 bits.

Por ejemplo, el ASCII estándar tiene un rango de 0 a 127; es decir, utiliza 7 bits. Para el ASCII extendido es de 0 a 255, por lo que se usa 8 bits.

**Bits de parada.** Indican el fin de la comunicación de un solo paquete. Los valores más frecuentes son 1, 2, 1,5 bits. Debido a la manera como se transfiere la información a través de las líneas de comunicación y dado que cada dispositivo tiene su propio reloj, es posible que los dos no estén sincronizados; por tanto, los bits de parada no solo indican el fin de la transmisión, sino que dan un margen de tolerancia para esa diferencia de los relojes.

Por ejemplo, mientras más bits de parada se usen, mayor será la tolerancia a la sincronía de los relojes; sin embargo, la transmisión será más lenta.

**Paridad.** Es la forma sencilla de verificar si hay errores en la transmisión serial. En esta opción se dispone de cuatro tipos de paridad (par, impar, marcada, espaciada). Para la paridad par e impar, el puerto serial fijará el bit de paridad (el último después de los bits de datos) a un valor, para asegurarse que la transmisión tenga un número par o impar de bits en estado lógico.

Por ejemplo, si la información a transmitir es 011 y la paridad es par, el bit de paridad será 0 para mantener el número de bits en estado alto lógico como par.

Por ejemplo, si la paridad fuera impar, entonces el bit de paridad será 1, para tener 3 bits en estado alto lógico.

La paridad marcada y espaciada no verifican el estado de los bits de datos; simplemente fija el bit de paridad en estado lógico alto para la marcada, y en estado lógico bajo para la espaciada.

Esto permite al dispositivo receptor conocer de antemano el estado de un bit, lo que sirve para deter-

minar si hay ruido que está afectando de manera negativa la transmisión de los datos, o si los relojes de los dispositivos no están sincronizados.

**Nota.** Para que dos dispositivos puedan comunicarse entre sí, es necesario que los parámetros en el receptor y en el transmisor sean iguales.

#### 5.2.4 Protocolos de comunicación en serie

Se han desarrollado una variedad de protocolos de comunicación, basados en la comunicación en serie en los últimos años. A continuación, presentamos algunos de ellos.

**Interfaz periférica en serie (SPI):** es un sistema de comunicación basado en tres cables, un cable para la señal de reloj y los otros dos pueden ser configurados como maestro a esclavo, y viceversa. Existe una línea adicional SS (*Slave select*), que se usa, principalmente, cuando se quiere enviar o recibir datos entre múltiples circuitos integrados.

**Circuito inter-integrado (I2C):** es una forma avanzada de USART. Las velocidades pueden ser más altas, alrededor de los 400 KHz. Dispone de dos cables: uno para la señal de reloj y otro para los datos, que es bidireccional.

**FireWire:** desarrollado por Apple, son buses de alta velocidad capaces de transmitir Audio/Video. El bus contiene varios cables dependiendo del puerto, que puede ser de 4, 6 u 8 pines.

**Ethernet:** se usa principalmente en conexiones LAN. El bus consta de 8 líneas o 4 pares de TX /RX.

**RS-232 estándar recomendado 232:** El RS-232, generalmente, se conecta mediante un conector DB9, que tiene 9 pines, de los cuales 5 corresponden a entradas, 4 a salidas y 1 es tierra. Todavía se puede encontrar este puerto en algunas PC antiguas.

**USART y UART:** USART (*Universal Asynchronous Receiver Transmitter*) y USART (*Universal Synchronous Asynchronous Receiver Transmitter*), son básicamente un módulo que convierte los datos paralelos en datos seriales. La diferencia entre ellos es

que USART admite los dos modos de comunicación síncrono y asíncrono, mientras que UART solo admite el modo asíncrono.

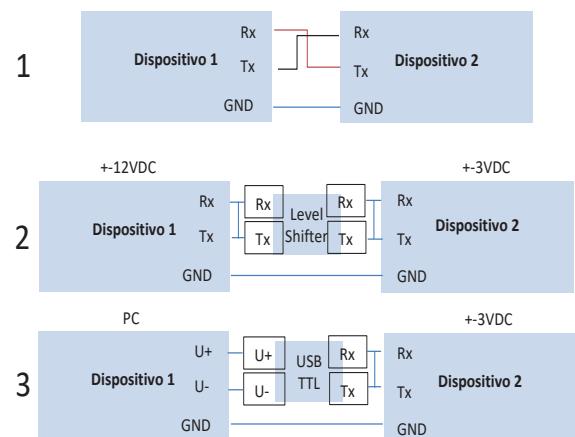
**Bus serie universal (USB):** este es el más popular de todos. Se usa para prácticamente todo tipo de conexiones. El bus tiene 4 líneas (VCC, Tierra, Data + , Data-).

#### 5.2.5 Tipos de conexiones o arreglos del puerto serial

Cuando un puerto serial opera o trabaja a un voltaje lógico distinto del microcontrolador o sistema embebido, se requiere hacer ciertas modificaciones al circuito. En la figura 5.6 se pueden observar tres tipos de comunicación digital: en la primera se está comunicando a dos sistemas con el mismo voltaje lógico; en la segunda se considera que uno de los dos dispositivos trabaja a un voltaje distinto, por lo que requiere de una etapa intermedia que se ajuste a los voltajes lógicos. La figura 5.6 numeral 3 muestra la comunicación con una computadora; se aprecia que es necesario un conversor de estados, conocido como USB-TTL.

Figura 5.6

Tipos de conexiones puerto serial



(1) Dos dispositivos seriales con el mismo voltaje lógico. (2) Distintos voltajes lógicos. (3) Comunicación serial con una computadora mediante una tarjeta de conversión de protocolo serial a USB

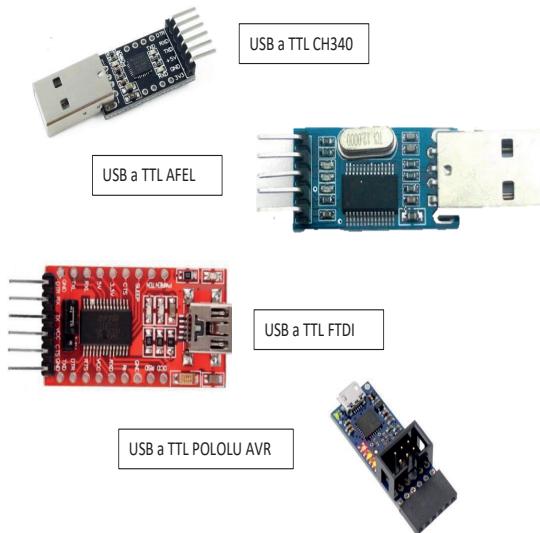
Tomado de HETPRO, 2017

Las computadoras de escritorio y portátiles, en su mayoría, ya no cuentan con puertos DB9. Para lo-

grar comunicar un microcontrolador o un sistema embebido con una PC, se requiere de un circuito adicional. Este elemento electrónico se conoce como convertidor USB-TTL. El convertidor está programado para convertir el protocolo de comunicación serial al protocolo USB; en este caso, la PC reconoce a dicho CI como un puerto serial virtual. Por tanto, en los sistemas operativos Windows se le asigna el nombre de COM, seguido de un número. La imagen 5.1 presenta los conversores más comunes disponibles en el mercado (Hall, s.f.).

### Imagen 5.1

Convertidores USB-TTL



### 5.3 Comunicación serial ATmega2560

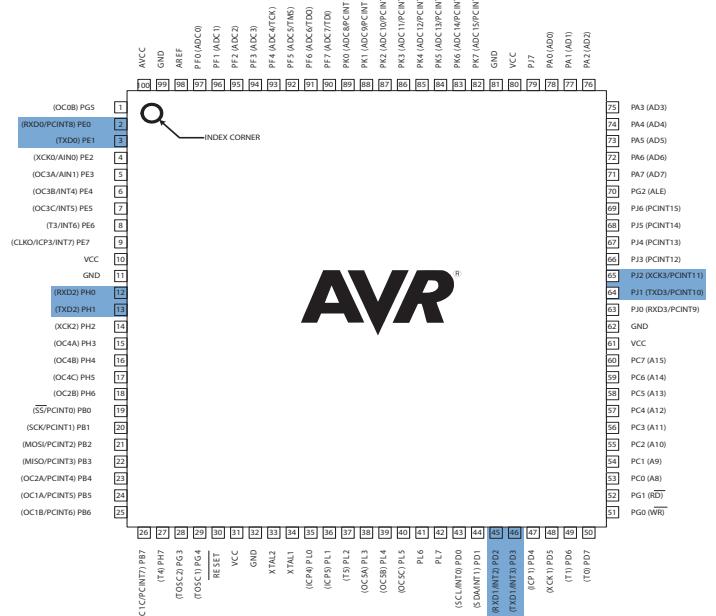
El Mega 2560 dispone de algunas facilidades para comunicarse con un computador, otra placa u otros microcontroladores. El ATmega2560 cuenta con el USART receptor y transmisión serial síncrono asíncrono universal, dispositivo de comunicación altamente flexible con las siguientes características:

- Operación *full duplex* (registros de transmisión y recepción en serie independientes)
  - Operación síncrona con reloj de maestro o esclavo
  - Generador de alta velocidad en baudios de alta resolución

- Admite series de 5, 6, 7, 8 o 9 bits de datos y 1 o 2 bits de parada
  - Generación de paridad par o impar y verificación de paridad compatibles con *hardware*
  - Detección de errores de trama
  - El filtrado de ruido incluye detección de bits de inicio falso y filtro digital de paso bajo
  - Tres interrupciones separadas en Tx Complete, Tx Data Register vacío, Rx Complete
  - Modo de comunicación multiprocesador
  - Modo de comunicación asíncrona de doble velocidad
  - Cuatro USART (USART0, USART1, USART2, USART3) que tienen diferentes registros de E/S

La imagen 5.2 exhibe la identificación de los pines de comunicación serial en el Mega 2560.

Imagen 5.2 | AMPLIAR IMAGEN



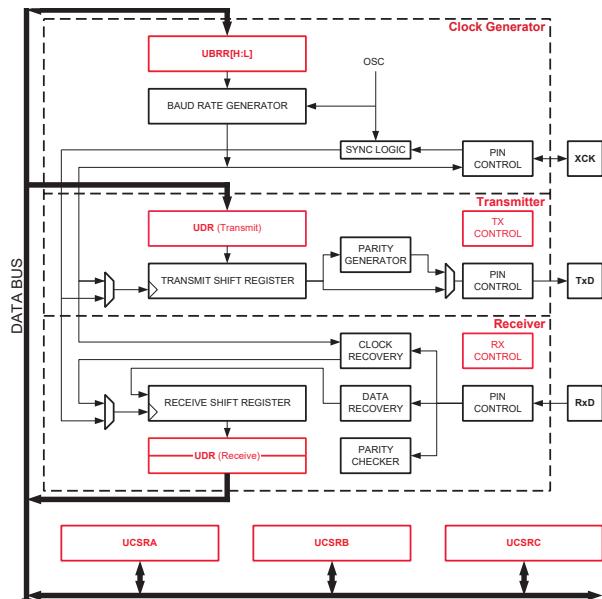
Tomado de Atmel 2014, p.2

### 5.3.1 Configuración de USART

El módulo USART consta de tres secciones principales: generador de reloj, transmisor y receptor (figura 5.7). Los registros clave incluyen tres registros de control y estado (UCSRnA, UCSRnB, UCSRnC), compartidos para las tres secciones. El registro (UDRn) es compartido por las secciones de transmisor y receptor. El registro (UBRRn) es utilizado por el generador de reloj (Microchip Technology, 2020c).

**Figura 5.7**

Diagrama de bloques USART



Tomado de Atmel, 2014, p.201

En este capítulo, vamos a desarrollar las configuraciones de los ejemplos solo para modo asíncrono.

Como recomendaciones el UART debe ser inicializado antes de que pueda tener lugar cualquier comunicación. El proceso de inicialización normalmente consiste en: establecer la velocidad de transmisión, establecer el formato de datos y habilitar el transmisor o el receptor, dependiendo del uso.

**Nota.** "n", en el nombre del registro/bit, identifica la instancia del hardware USART específica (0,1,2) a la que está asociado el registro/bit; por ejemplo, UCSR0A se refiere al registro de control y estado USART0 A.

### 5.3.2 Registro de velocidad en baudios UBRRn

El registro UBRR establece la velocidad en baudios de USART/UART (ver figura 5.8). Este registro se utiliza para generar la transmisión de datos a la velocidad que se requiera. El UBRR es un registro de 16 bits que consta de dos registros, UBRR(H) y UBRR(L), cada uno de 8 bits. Dado que el microcontrolador ATmega2560 es de 8 bits, cualquier tamaño de registro es de 8 bits; por tanto, la única manera de obtener 16 bits es tener dos registros asociados.

**Figura 5.8**

Bits de configuración del registro UBRR

bit	15	14	13	12	11	10	9	8
UBRRHn	-	-	-	-	UBRR [11:8]			
UBRRLn	UBRR [7:0]							
Acción	7	6	5	4	R/W	R/W	R/W	R/W
Estado	R	R/W	R/W	R	R/W	R/W	R/W	R/W
inicial	0	0	0	0	0	0	0	0

Tomado de Atmel, 2014, p. 222

Para calcular el valor del registro o de los baudios se aplica la siguiente fórmula:

$$UBRRn = \frac{fosc}{16Baud} - 1 \quad (5.1)$$

$$Baud = \frac{fosc}{16(UBRRn + 1)}$$

Donde:

*fosc* = frecuencia del cristal del microcontrolador

*Baud* = Velocidad de transmisión que se requiera en bps

**Nota.** Para los cálculos de frecuencia en baudios, generalmente se acepta porcentajes de error menores a  $\pm 2\%$ .

### 5.3.3 Registro de control y estado UCSRnA

El primer registro de control y estado que se analiza es el UCSRnA. La figura 5.9 muestra el nombre de los bits que componen este registro.

**Figura 5.9**

Bits de configuración registro UCSRnA

bit	7	6	5	4	3	2	1	0
UCSRnA	RXCn	TXCn	UDREn	FEn	DORn	UPEn	U2Xn	MPCMn
Acción	R	R/W	R	R	R	R	R/W	R/W
Estado inicial	0	0	1	0	0	0	0	0

Tomado de Atmel, 2014, p. 219

**Bit 7 RXCn.** Este bit se coloca en 1 lógico automáticamente, cuando se ha completado la recepción de algún dato en el registro UDRn, y se coloca en 0 lógico automáticamente cuando se haya leído el dato. Si se habilita el uso de la interrupción por recepción del módulo USART, este bit se utiliza para detectar la interrupción.

**Bit 6 TXCn.** Este bit se coloca en 1 automáticamente cuando se ha complementado la transmisión de algún dato que se encontraba en el registro UDRn; se coloca en 0 automáticamente al cargar otro dato en el registro UDRn a ser transmitido. Si se ha habilitado el uso de la interrupción por transmisión del módulo USART, este bit se utiliza para detectar la interrupción.

**Bit 5 UDREn.** Este bit, al ponerse en 1 lógico en forma automática, indica que el registro UDRn está vacío, por lo que se le podrá cargar con algún dato. Cuando se cargue con algún valor, el registro UDR0 se pondrá automáticamente en 0 lógico. Se puede de habilitar la interrupción por detección cuando el registro UDCRn está vacío, y este bit será el que indique esta interrupción.

**Bit 4 FEn.** Este bit se pondrá en 1 automáticamente cuando exista un error en la recepción de algún dato. El error se detecta cuando el bit de parada del dato es un 0; normalmente debe ser un 1 lógico. Se recomienda poner ese bit en 0 antes de recibir algún dato.

**Bit 3 DORn.** Este bit se coloca en 1 lógico automáticamente cuando se sobrescribe algún dato del registro UDRn que no haya sido leído. Se coloca en 0 lógico automáticamente cuando se lea el dato. Se recomienda colocar este bit en 0 antes de recibir algún dato.

**Bit 2 UPEn.** Este bit se pondrá en 1 automáticamente cuando se produzca un error de paridad de algún dato; se pondrá en 0 automáticamente cuando se lea el dato. Se recomienda poner este bit en 0 antes de recibir algún dato.

**Bit 1 U2Xn.** Este bit interviene en la velocidad de los datos, esto es, en los *baudios*. Si se coloca en 0, se dice que la velocidad será normal; si se coloca en 1 lógico será a doble de velocidad.

**Bit 0 MPCn.** Este bit es utilizado en el modo síncrono y detecta cuál de los microcontroladores esclavo ha sido elegido. En modo asíncrono se coloca en 0 lógico.

### 5.3.4 Registro de control y estado UCSRnB

El segundo registro de control y estado que se analiza es el UCSRnB. La figura 5.10 presenta el nombre de los bits que componen este registro.

**Figura 5.10**

Bits de configuración registro USCRnB

bit	7	6	5	4	3	2	1	0
UCSRnB	RXCIEn	TXCIEn	UDRIEn	RXCEn	TXEEn	UCSZn2	RXB8n	TXB8n
Acción	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W
Estado inicial	0	0	1	0	0	0	0	0

Tomado de Atmel, 2014, p. 220

**Bit 7 RXCIEn.** Al colocar este bit a 1 lógico se habilita el uso de la interrupción USART por recepción.

**Bit 6 TXCIEn.** Al colocar este bit a 1 lógico se habilita el uso de la interrupción USART por transmisión.

**Bit 5 UDRIEn.** Al colocar este bit a 1 lógico se habilita el uso de la interrupción USART. Escribiendo este bit a uno se habilita la interrupción del indicador UDREn.

**Bit 4 RXENn.** Al colocar este bit a 1 lógico se habilita el uso del pin RXD para la recepción del módulo USART. Se habilita uso de recepción.

**Bit 3 TXENn.** Al colocar este bit en 1 lógico se habilita el uso del pin TXD para la transmisión del módulo USART. Se habilita uso de la recepción.

**Bit 2 UCSZn2.** Este bit junto con los bits 2 y 1 del registro UCSRnC permiten elegir cuántos bits serán los datos para recibir o transmitir en la comunicación serial.

**Bit 1 RXB8n.** Si se elige la comunicación serial a 9 bits, este será el noveno bit en la recepción del dato.

**Bit 0 TXB8n.** Si se elige la comunicación serial a 9 bits, este será el noveno bit en la transmisión del dato.

### 5.3.5 Registro de control y estado UCSRnC

El modo de transmisión de datos (síncrono o asíncrono), paridad, número de bits de parada y número de bits de datos lo realiza el registro UCSRnC, en el microcontrolador ATmega2560. La figura 5.11 indica el nombre de los bits que componen este registro.

**Figura 5.11**

Bits de configuración del registro UCSRnC

bit	7	6	5	4	3	2	1	0
UCSRnC	UMSELn1	UMSELn0	UPMn1	UPMn0	USBSn	UCSZn1	UCSZn0	UCPOLn
Acción	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Estado inicial	0	0	0	0	0	1	1	0

Tomado de Atmel, 2014, p. 221

**Bit 7 UMSELn1 y bit 6 UMSELn0.** Para configurar el modo de transmisión de los datos, que puede ser síncrono o asíncrono, se utilizan los bits UMSELn1/UMSELn0. Estos pueden tener las configuraciones que se detallan en la tabla 5.2.

**Tabla 5.2**

Bits de configuración de tipo de transmisión

UMSELn1	UMSELn0	Modo
0	0	USART asíncrono
0	1	USART síncrono
1	0	Reservado
1	1	Máster SPI

Tomado de Atmel, 2014, p.221

**Bit 5 UPMn1 y bit 4 UPMn0.** Para establecer el tipo de generación y verificación de paridad, se utilizan los bits UPMn1/UPMn0. Estos pueden tomar los valores que se especifican en la tabla 5.3.

**Tabla 5.3**

Bits de configuración de paridad

UPMn1	UPMn0	Modo
0	0	Deshabilitado
0	1	Reservado
1	0	Paridad par
1	1	Paridad impar

Tomado de Atmel, 2014, p 221

**Bit 3 USBSn.** Para establecer el número de bits de parada, se utiliza el bit USBSn y puede tomar los valores que se señalan en la tabla 5.4.

**Tabla 5.4**

Configuración bits de parada

USBSn	Stop Bits
0	1-bit
1	2-bit

Tomado de Atmel, 2014, p 221

**Bit 2 UCSZn1 y bit 1 UCSZn0 junto con el bit 2 UCSZn2 del registro (UCSRnB).** Para establecer la cantidad de bits que se van a enviar o recibir en el ATmega2560, se utiliza la configuración en tres bits: UCSZn2/UCSZn1/UCSZn0, que pueden tomar los valores expuestos en la tabla 5.5.

**Tabla 5.5**

Cantidad de bits de envío y recepción

UCSZn2	UCSZn1	UCSZn0	Cantidad de bits
0	0	0	5-bit
0	0	1	6-bit
0	1	0	7-bit
0	1	1	8-bit
1	0	0	Reservado

1	0	1	Reservado
1	1	0	Reservado
1	1	1	9-bit

Tomado de Atmel, 2014, p 222

**Bit 0 UCPOLn.** El bit UCPOLn establece la relación entre el cambio de salida de datos y muestra la entrada de datos. Este bit solo se utiliza para el modo síncrono. Cuando sea asíncrono, este bit tiene que ser 0 (ver tabla 5.6).

**Tabla 5.6**

*Configuración del bit UCPOLn*

UCPOLn	Datos transmitidos modificados (salida del pin TxDn)	Datos recibidos muestreados (entrada en el pin RxDn)
0	Rising XCKn Edge	Falling XCKn Edge
1	Falling XCKn Edge	Rising XCKn Edge

Tomado de Atmel, 2014, p 222

### 5.3.6 Registro de datos UDRn

El registro UDRn tiene doble función: se colocará el carácter que se quiera transmitir y también se encontrará el carácter recibido. Los bits de registro se pueden observar en la figura 5.12.

**Figura 5.12**

*Bits de configuración registro UDRn*

bit	7	6	5	4	3	2	1	0
UDRn(Read)					RX [7:0]			
UDRn(Write)					TX [7:0]			
Acción	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Estado inicial	0	0	0	0	0	0	0	0

Tomado de Atmel, 2014, p. 218

**Nota.** La velocidad en baudios, el modo de operación y el formato de la trama deben configurarse antes de realizar cualquier transmisión.

### Usando el USART (Resumen)

Para la operación de funcionamiento básico, se deben seguir estos pasos:

1. Elija una velocidad en baudios y programe los registros UBRRn.
2. Habilite las secciones de transmisión y recepción en serie de USART.
3. Si está transmitiendo, espere hasta que el registro de desplazamiento de transmisión esté vacío (UCSRnA.UDREN); luego cargue su bit de datos en UDRn.
4. Si recibe, espere hasta que se establezca el bit de recepción de datos del receptor (UCSRnA.RXCn); luego, lea los datos de UDRn. La lectura de UDRn borra automáticamente el bit y prepara el hardware para recibir el siguiente bit.

### 5.4 Ejercicios prácticos de configuración

A continuación, presentamos una serie de casos, en los que, de manera progresiva, profundizamos los conceptos tratados en este capítulo, mediante ejercicios prácticos.

**Ejercicio 5.1:** Se desea realizar una función que inicialice la comunicación serial de transmisión y recepción con los siguientes parámetros: modo asíncrono, sin paridad, un bit de parada, 8 bits de datos, suponiendo un cristal de 16 MHZ a una velocidad en baudios de 9600.

#### Solución:

*Configuración del registro UCSR0A:*

Todos los bits de este registro se colocan en 0 lógico, ya que, a excepción del bit 1 U2X0 y el bit 0 MPCMn, todos los demás trabajan de forma automática. El bit 1 se coloca en 0 lógico porque se trabaja en la velocidad de baudios en forma normal. El bit 0 se coloca en 0 lógico porque se trabaja en modo asíncrono (ver figura 5.13).

**Figura 5.13**

*Configuración velocidad normal registro UCSR0A. Ejercicio 5.1*

bit	7	6	5	4	3	2	1	0
UCSRnA	0	0	0	0	0	0	0	0

Esta configuración del registro puede ser escrita en código, de la siguiente manera:

```
UCSRIA=0b00000000; //velocidad normal
```

Configuración del registro UCSR0B:

Como en este ejemplo no se utilizan interrupciones, se habilitarán los pines de RXD y TXD para la comunicación serial, y será a 8 bits.

Los bits 7 y 6 se colocan en 0 lógico porque no habrá interrupción por recepción y transmisión, respectivamente (ver figura 5.14).

**Figura 5.14**

Configuración registro UCSR0B sin interrupción por TX y RX.  
Ejercicio 5.1

bit	7	6	5	4	3	2	1	0
UCSRnB	0	0	x	x	x	x	x	x

Como no se solicita la detección de que el registro UDRO quede vacío, el bit 5 del registro UCSR0B se coloca en 0 lógico (ver figura 5.15).

**Figura 5.15**

Configuración bit UDRIEn. Ejercicio 5.1

bit	7	6	5	4	3	2	1	0
UCSRnB	0	0	0	x	x	x	x	x

El bit 4 y bit 3 se colocan en 1 lógico para habilitar el uso del pin RXD y TXD, respectivamente (ver figura 5.16).

**Figura 5.16**

Habilitar TX y RX en el registro UCSR0B. Ejercicio 5.1

bit	7	6	5	4	3	2	1	0
UCSRnB	0	0	0	1	1	x	x	x

El bit 2, junto con los bits 2 y 1 del registro UCSR0C, son utilizados para la cantidad de datos que van hacer enviados. Como en este ejemplo solo se hace uso de 8 bits, este bit se coloca en 0 lógico (ver figura 5.17).

**Figura 5.17**

Configuración bit UCSZn2. Ejercicio 5.1

bit	7	6	5	4	3	2	1	0
UCSRnB	0	0	0	1	1	0	x	x

Los bits 1 y 0 se colocan en 0 lógico, debido a que en este ejemplo los datos son de 8 bits (ver figura 5.18).

**Figura 5.18**

Configuración bit RXB8n y TXB8n. Ejercicio 5.1

bit	7	6	5	4	3	2	1	0
UCSRnB	0	0	0	1	1	0	0	0

Esta configuración del registro puede ser escrita en código, de la siguiente manera:

```
UCSRIB=0b00011000; //transmisión y recepción habilitados a 8 bits
```

Configuración del registro UCSR0C

Como parte de los requisitos del ejercicio, se debe elegir modo asíncrono (ver figura 5.19); por lo tanto, los bits 7 y 6 del registro UCSR0C deberán estar en 0 lógico, como indica la tabla 5.2 Bits de configuración de tipo de transmisión .

**Figura 5.19**

Configuración modo asíncrono registro UCSRnC. Ejercicio 5.1

bit	7	6	5	4	3	2	1	0
UCSRnC	0	0	x	x	x	x	x	x

Para no establecer paridad, los bit 5 y 4 se colocan en 0 lógico en el registro UCSR0C (ver figura 5.20).

**Figura 5.20**

Configuración sin paridad registro UCSRnC. Ejercicio 5.1

bit	7	6	5	4	3	2	1	0
UCSRnC	0	0	0	0	x	x	x	x

Para utilizar 1 bit de parada, el bit 3 del registro UCSR0C se coloca en 0 lógico (ver figura 5.21).

**Figura 5.21**

Configuración 1 bit de parada registro UCSRnC. Ejercicio 5.1

bit	7	6	5	4	3	2	1	0
UCSRnC	0	0	0	0	0	x	x	x

Como se desea 8 bits de datos, los bits 2 y 1 del registro UCSR0C se colocan en 1 lógico, como se puede apreciar en la tabla 5.5 Cantidad de bits de envío y recepción (ver figura 5.22).

**Figura 5.22**

Configuración 8 bits de datos registro UCSRnC. Ejercicio 5.1

bit	7	6	5	4	3	2	1	0
UCSRnC	0	0	0	0	0	1	1	x

El bit 0 del registro UCSR0C es utilizado únicamente en el modo síncrono; por tanto, en este ejercicio el bit se coloca en 0 lógico (ver figura 5.23).

**Figura 5.23**

Configuración bit UCPOLn registro UCSRnC. Ejercicio 5.1

bit	7	6	5	4	3	2	1	0
UCSRnC	0	0	0	0	0	1	1	0

Esta configuración del registro puede ser escrita en código, de la siguiente manera:

```
UCSR1C=0b00000110; //asíncrono, sin bit de paridad, 1 bit de parada, 8 //bits de datos
```

### Configuración del registro UBRR0

Para una transmisión a 9600 bps con un cristal a 16 MHz, se realiza el siguiente cálculo, mediante la aplicación de la ecuación 5.1, descrita a continuación:

$$UBRR0 = \frac{16000000}{16(9600)} - 1 \quad (5.1)$$

$$UBRR0 = \frac{16000000}{153600} - 1$$

$$UBRR0 = 103.166666$$

Por tanto, el valor del registro será:

$$UBRR0 = 103$$

Para obtener el valor de los baudios, se aplica la ecuación 5.2:

$$Baud = \frac{16000000}{16(103+1)} \quad (5.2)$$

$$\begin{aligned} Baud &= 9615.38 \text{ representa } 9600 \text{ por el error de} \\ &\pm 2\% \end{aligned}$$

Esta configuración del registro puede ser escrita en código, de la siguiente manera:

```
UBRR1=103; //9600bps a 16MHz
```

Finalmente, el código para el ejercicio propuesto sería el siguiente:

```
#define F_CPU 16000000UL
#include <avr/io.h>
void config_UART1()
{
    UCSR1B=0b00011000; // sin interrupciones, habilitación de transmisión y
    // recepción, transmisión a 8 bits de datos
    UCSR1C=0b00000110; // configuración asíncrona, sin paridad, 1bit de parada
    // transmisión a 8 bits de datos
    UBRR1=103; //9600 bps.
}
```

**Nota.** La inicialización se podría hacer dentro de la función config\_UART1(); además, el registro UCSR0A siempre se inicia en 0; por lo tanto, no es necesario colocarlo en el código.

**Ejercicio 5.2:** Se desea realizar una función que inicialice la recepción de datos, a través del pin RXD que reciba un carácter y una función que reciba una cadena de caracteres.

### Solución:

Cuando se va a recibir un dato en el microcontrolador, los datos que se reciben son de tipo carácter (Char 8 bits); por lo tanto, se tendrá que esperar que el bit 7 del registro UCSR1A se ponga a 1. Este bit indica que se ha completado la recepción del dato. Este dato está en el registro UDR1, cuando recibe el dato guardándolo en una variable de tipo carácter (char), este bit se pone en 0 automáticamente (ver figura 5.24).

Figura 5.24

Bit 7 registro UCSRnA. Ejercicio 5.2



El código para el ejercicio propuesto sería el siguiente:

```
unsigned char Rx_UART1()
{
    if(UCSR1A&(1<<7)) // Si el bit de registro se ha colocado en 1
    {
        return UDR1;
    }
    else
    {
        return 0; //caso contrario retornar 0
    }
}
```

Para implementar la recepción de varios caracteres en forma de cadena, se usa la función anterior más un arreglo que almacene la longitud de la cadena enviada. También se recurre a un lazo *while* para que analice bit a bit los datos recibidos hasta cuando se identifique un salto de línea, se suspenda el almacenamiento de datos y retorne la cadena recibida.

```
char *Rx_cadena_UART1(char *st)
{
    char c;
    int count=0;
    while ((count<255) && ((c = Rx_char_UART1()) != '\r'))
    {
        st[count++] = c; // añadir char al string
    }
    st[count] = '\0';
    return st;
}
```

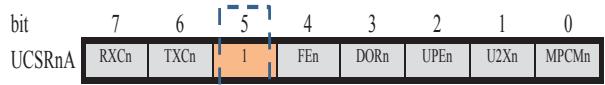
**Ejercicio 5.3:** Se desea realizar dos funciones que inicialicen la transmisión de datos (carácter y cadena de caracteres), a través del pin TXD.

### Solución:

En primer lugar, para el envío de datos de tipo carácter se tiene que esperar que el registro UDR1 donde se colocan los datos que se van a enviar esté vacío. Esto se realiza esperando que el bit 5 UDRE1 del registro UCSR1A se coloque automáticamente en 1 lógico. Este bit indica que se ha completado la transmisión de un dato. Cuando se vuelve a cargar un dato en el registro UDR1, este bit se coloca automáticamente en 0 lógico (ver figura 5.25).

Figura 5.25

Bit 5 del registro UCSRnA. Ejercicio 5.3



El código para el ejercicio propuesto, envío de un carácter, sería el siguiente:

```

void Tx_char_UART1(unsigned char caracter)
{
    while (!(UCSR1A&(1<<5))); // mientras el registro UDR1 esté lleno esperar
    UDR1=caracter; // cuando el registro UDR1 este vacío enviar
}

```

Para el envío de una cadena de caracteres, se puede utilizar la función anterior más un acumulador, en el cual se almacenen los caracteres en forma de cadena.

```

void Tx_cadena_UART1(char* cadena) //cadena de caracteres tipo chart
{
    while (*cadena != 0x00) // mientras el último carácter sea diferente de nulo

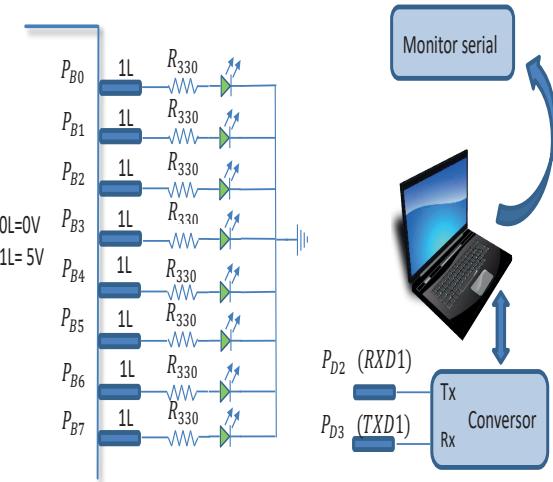
    {
        Tx_char_UART1(*cadena); // transmite los caracteres de la cadena
        cadena++; //incrementa los caracteres
    }
}

```

**Ejercicio 5.4:** Con el uso de las funciones creadas en los ejercicios 5.1, 5.2 y 5.3, realizar la comunicación serial entre el ATmega2560 y una PC, la cual presenta el siguiente comportamiento: cuando se inicie la comunicación, el microcontrolador envía la cadena “Serial Iniciado”; la PC envía el carácter “a”, el cual realiza el encendido de un led en el microcontrolador (PB0). Cuando este finalice, el microcontrolador responderá con la cadena “led on”. Despues, la PC enviará una cadena “encender”, lo que ocasionará que el microcontrolador encienda todos los ledes conectados al puerto B y responderá con la cadena “encendido”; caso contrario enviará la cadena “apagado”. La figura 5.26 exhibe el esquema de conexión.

**Figura 5.26**

Esquema de conexión. Ejercicio 5.4.



### Solución:

El código para el ejercicio propuesto sería el siguiente:

```

#include <avr/io.h>
#define F_CPU 16000000UL
unsigned char dato;
unsigned char Rx_char_UART10
{
    while(!(UCSR1A & (1 << RXC1)));
    return UDR1;
}
char *Rx_cadena_UART1(char *st)
{
    while(!(UCSR1A & (1 << RXC1)));
    char c;
    int count=0;
    while ((count<254) && ((c = Rx_char_UART10) != '\r'))
    {
        st[count++]= c; // añadir char al string
    }
    st[count]='\0';
    return st;
}
void config_UART10
{
    UCSR1B=0B00011000;
    UCSR1C=0B00000110;
    UBRR1=103;
}
void Tx_char_UART1(unsigned char caracter)
{
    while (!(UCSR1A & (1 << 5)));
    UDR1=caracter;
}
void Tx_cadena_UART1(char* cadena)
{
    while (*cadena != 0x00)
    {
        Tx_char_UART1(*cadena);
        cadena++;
    }
}
int main(void)
{
    DDRB=255;
}

```

```

PORTB=0;
DDRD&=~(1<<2); //configuración de entrada para PD2 (RXD1)
DDRD|= (1<<3); //configuración de salida para PD3 (TXD1)

config_UART1();
Tx_cadena_UART1("Serial Iniciado");
UCS1A|=0b01000000; //baja la bandera de transmisión

while (1)
{
    dato=Rx_char_UART1(); //para leer dato recibido
    if (dato=='a')
    {
        PORTB=0B00000001; // encender led
        Tx_cadena_UART1("led on");
        UCS1A|=0b01000000; //baja la bandera de trasmisión
        UCS1A|=0b10000000; //baja la bandera de RX
    }
    else
    {
        PORTB=0B00000000; // todo apagado
        UCS1A|=0b10000000; //baja la bandera de RX
    }
    char st[8];
    if (strcmp(Rx_cadena_UART1(st),"encender")==0)
    {
        PORTB=0B11111111; // encender led
        Tx_cadena_UART1("encendido");
        UCS1A|=0b01000000; //baja la bandera de transmisión
        UCS1A|=0b10000000; //baja la bandera de RX
    }
    else
    {
        Tx_cadena_UART1("apagado");
        UCS1A|=0b01000000; //baja la bandera de transmisión
        PORTB=0B00000000; // todo apagado*
        UCS1A|=0b10000000; //baja la bandera de RX
    }
}
}

```

**Nota.** La función `strcmp` compara dos cadenas de caracteres. Si la salida es menor a 0, indica que la primera cadena es menor que la segunda. Mayor que 0 indica que la segunda cadena es menor que la primera, mientras que si es igual que 0, indica que las cadenas son iguales.

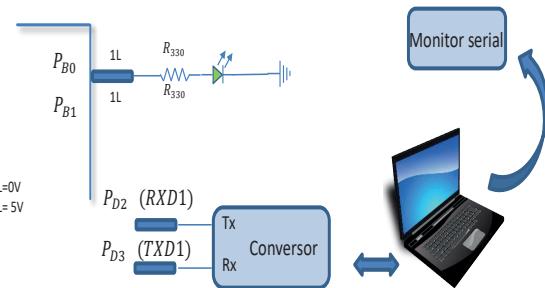
Si se observa detenidamente la solución del ejercicio anterior, se puede apreciar que el algoritmo tiene una ejecución secuencial; eso significa que efectúa las acciones una a continuación de la otra, en el orden que está establecido. Estas ejecuciones secuenciales involucran un retardo en el código si es que estas acciones están ligadas con ciclos repetitivos tipo `while`, `for`, etc., por lo que se recomienda utilizar el uso de interrupciones para, de esa manera, ejecutar en paralelo: mientras se ejecuta alguna acción en el bucle principal, se puede transmitir o recibir datos. Para aprender el uso de dichas interrupciones, se plantea la solución a los siguientes ejercicios.

**Ejercicio 5.5:** Realizar la transmisión del carácter “a” con el uso de interrupciones, mientras que se esté ejecutando en el bucle principal el parpadeo de un led.

El esquema de conexión se presenta en la figura 5.27.

**Figura 5.27**

*Esquema de conexión. Ejercicio 5.5*

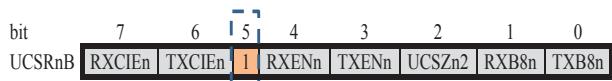


### Solución:

Para resolver este problema, se activa la interrupción del registro UDRn para verificar que esté vacío, antes de enviar un dato. La configuración inicial del registro B se puede observar en la figura 5.28.

**Figura 5.28**

Configuración inicial registro UCSRnB. Ejercicio 5.5



Las interrupciones son recursos o mecanismos del microcontrolador para responder a eventos. Estas permiten suspender temporalmente el programa principal, para ejecutar una subrutina de interrupción ISR. Una vez terminada dicha subrutina, se reanuda el programa principal; por tanto, en C, el código es el siguiente:

```
ISR(USART1_UDRE_vect)
{
    UDR1=dato;
    UCSR1B&=~(1<<5); //deshabilita la interrupción USART AVR
}
```

Además se implementa el uso de la función sei(), que habilita el uso de interrupciones globales:

```
sei();//habilita las interrupciones globales
```

Finalmente, el código para el ejercicio propuesto sería el siguiente:

```
#define F_CPU 16000000UL
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>

unsigned char dato='a';// iniciar con una transmisión
void config_UART1();//prototipo de función para iniciar el USART

int main(void)
{
    DDRB=255;
    DDRD&=~(1<<2); //configuración de entrada para PD2 (RXD1)
    DDRD|=1<<3; //configuración de salida para PD3 (TXD1)

    config_UART1();
    sei();//habilita las interrupciones globales

    while (1)
    {
        PORTB=0B00000001;
        _delay_ms(1000);
        PORTB=0B00000000;
        _delay_ms(1000);
    }
}

ISR(USART1_UDRE_vect)
{
    UDR1=dato;
    UCSR1B&=~(1<<5); //deshabilita la interrupción USART AVR
}

void config_UART1()
{
    UCSR1B=0B00101000;
    UCSR1C=0B00000110;
    UBRR1=103;//9600
}
```

**Ejercicio 5.6:** Realizar la recepción de dos caracteres con el uso de interrupciones; cuando reciba el carácter “a” se encienda un led, y con el carácter “b” se apague (usar el esquema de conexión del ejercicio 5).

### Solución:

Para habilitar la interrupción USART AVR, al completar la recepción de un carácter, en el registro UCSRnB se coloca en 1 lógico el bit 7, lo que en C sería lo siguiente:

UCSR1B |= (1<<7); //habilita interrupción por recepción USART AVR.

El bit 7 del registro UCSRnA se pondrá en 1 lógico automáticamente para indicar que se ha producido una interrupción USART AVR, al completar la recepción de un carácter en el registro UDRn. Cuando se lea el carácter recibido, este bit se pondrá en 0 automáticamente para seguir detectando la interrupción por recepción.

La función para la interrupción por recepción es la siguiente:

```
ISR(USART1_RX_vect)
{
    dato=UDR1; //al dato de tipo char se le asigna el valor del registro UDR0
    //otras tareas a realizar
}
```

Finalmente, el código fuente propuesto para el ejercicio 5.6 es:

```
#define F_CPU 16000000UL
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>

unsigned char dato;//iniciar con una trasmisión

void config_UART1(); //prototipo de función para iniciar el USART
int main(void)
{
    DDRB=255;
    DDRD&=~(1<<2); //configuración de entrada para PD2 (RXD1)
    DDRD|= (1<<3); //configuración de salida para PD3 (TXD1)

    config_UART1();
    sei(); //habilita las interrupciones globales

    while (1)
    {
        if(dato=='a')
        {
            PORTB=0B00000001;
        }
        if(dato=='b')
        {
            PORTB=0B00000000;
        }
    }
}

ISR(USART1_RX_vect){
    dato=UDR1; //al dato de tipo char se le asigna el valor del registro UDR0
    //otras tareas a realizar
}

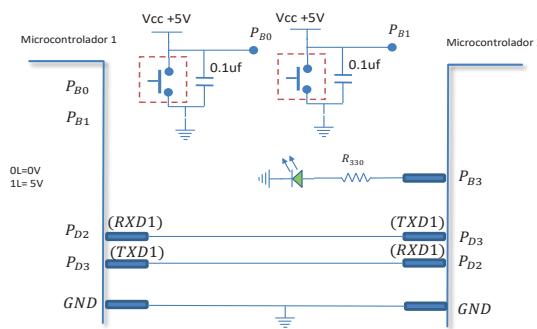
void config_UART1()
{
    UCSRB=0B00011000;
    UCSCR=0B00000110;
    UBRR1=103; //9600
    UCSRB=(1<<7); //habilita interrupción por recepción USART AVR.
}
```

**Ejercicio 5.7:** Realizar la comunicación serial entre dos microcontroladores ATmega2560 que cumplan con el siguiente funcionamiento: en el primer microcontrolador se deben conectar dos pulsadores, los cuales son los encargados de encender y apagar un led que está conectado en el segundo microcontrolador.

El esquema de conexión se presenta en la figura 5.29.

**Figura 5.29**

Esquema de conexión. Ejercicio 5.7



### Solución:

El código fuente para el primer microcontrolador es el siguiente:

```
#define F_CPU 16000000UL
#include <avr/io.h>
#include <avr/interrupt.h>
unsigned char dato;// iniciar con una transmisión
void config_UART();//prototipo de función para iniciar el USART
int main(void)
{
    DDRB&=~((1<<0)|(1<<1));//PB0 y PB1 como entradas digitales para los pulsadores
    PORTB=((1<<0)|(1<<1));// habilita las resistencias internas pull-up de PB0 y PB1
    DDRD&=(1<<2); //configuración de entrada para PD2 (RXD1)
    DDRD|=(1<<3); //configuración de salida para PD3 (TXD1)
    config_UART();
    (1)
    {
        if(!(PINB&(1<<0)))
            {//si se presiona el pulsador para encender el led
                while (!(UCSR1A&(1<<5)));// mientras el registro UDR1 esté lleno esperar
                UDRI='a';// cuando el registro UDR1 esté vacío enviar
                UCSRA|=0b01000000; //baja la bandera de transmisión
            }
        if(!(PINB&(1<<1)))
            {//si se presiona el pulsador para apagar el led
                while (!(UCSR1A&(1<<5)));// mientras el registro UDR1 esté lleno esperar
                UDRI='b';// cuando el registro UDR1 esté vacío enviar
                UCSRA|=0b01000000; //baja la bandera de transmisión
            }
    }
    void config_UART()
    {
        UCSRB=0B00001000;
        UCSRIC=0B00000110;
        UBRR1=103;//9600
    }
}
```

El código para el segundo microcontrolador es el siguiente:

```
#define F_CPU 16000000UL
#include <avr/io.h>
#include <avr/interrupt.h>

unsigned char dato;// iniciar con una transmisión
void config_UART1();//prototipo de función para iniciar el USART

int main(void)
{
    DDRB=255;
    DDRD&=~(1<<2); //configuración de entrada para PD2 (RXD1)
    DDRD|= (1<<3); //configuración de salida para PD3 (TXD1)

    config_UART1();
    sei();//habilita las interrupciones globales

    while (1)
    {
        if(dato=='a')
        {
            PORTB=0B00000001;
        }
        if(dato=='b')
        {
            PORTB=0B00000000;
        }
    }
}

ISR(USART1_RX_vect)
{
    dato=UDR1; //al dato de tipo char se le asigna el valor del registro UDR0
}

d config_UART1()
{
    UCSRB=0B00011000;
    UCSRC=0B00000110;
    UBRL=103;//9600
    UCSRB|= (1<<7);//habilita interrupción por recepción USART AVR.
}
```

# Capítulo 6

## Temporizadores y contadores

### 6.1 Anotaciones preliminares sobre temporizadores y contadores

Un contador es un dispositivo que permite contar eventos de cualquier índole, sean aleatorios o cílicos. Un ejemplo de evento aleatorio es el ingreso de vehículos a un parqueadero. En este caso, no se tiene la certeza de que van a ingresar de forma continua o periódica; mucho menos se conoce la cantidad de vehículos que pueden ingresar en un determinado tiempo. En cambio, en los eventos cílicos, se tiene la certeza del tiempo en el que ocurren los eventos. Por ejemplo, el sol sale cada mañana y es posible contar los días, las horas, los minutos, los segundos, etcétera. Es decir, el tiempo es el único evento que con certeza ocurrirá de forma periódica. Es por esta razón que a los contadores de tiempo se les denomina temporizadores. En cambio, a los contadores que cuentan eventos no periódicos se les denomina contadores.

En este capítulo, abordamos el estudio de contadores y temporizadores usando el microcontrolador. Revisamos sus diferentes modos de operación, la ubicación de sus terminales y las aplicaciones que podemos darles. Además, explicamos, de forma detallada, los cálculos y consideraciones a tomar en cuenta para que un programa pueda trabajar con una base de tiempo fijo, ampliamente utilizada en aplicaciones de sistemas de control.

### 6.2 Contadores de 8 y 16 bits

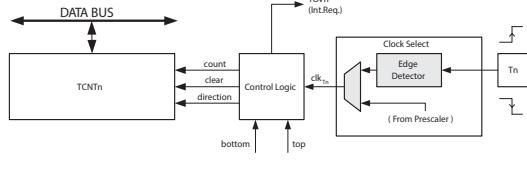
En el microcontrolador ATmega2560 existen dos tipos de temporizadores/contadores (TC): los que

almacenarán su cuenta en registros de 8 bits y los que lo hacen en registros de 16 bits. En el primer tipo se encuentran los TCs 0 y 2, mientras que en el segundo están los TCs 1, 3, 4 y 5. Pero ¿qué implica tener un contador de 8 o 16 bits? ¿Afecta en algo, usar el uno o el otro?, ¿es más difícil programar uno u otro? Estas preguntas responderemos, poco a poco, en el transcurso de este capítulo. Además, presentamos un análisis de ambos tipos de forma paralela, con el fin de que la comparación y las conclusiones que se puedan obtener sean inmediatas.

La figura 6.1 permite reconocer la estructura interna del TCO.  $T_n$  simboliza el terminal físico del microcontrolador, por donde es posible ingresar pulsos eléctricos para incrementar la cuenta del registro  $TCNT_n$ . El subíndice “n” indica el número del temporizador a usar. El terminal  $T_0$  se encuentra ubicado en el pin 50 del chip del microcontrolador o en el pin 38 de la placa Arduino Mega, como se ilustra en la imagen 6.1. El bloque identificado como Edge Detector es un detector de flanco del pulso entrante, ya sea flanco de subida o flanco de bajada. Tanto los pulsos que ingresan por los terminales  $T_n$ , así como el proveniente del preescalador son multiplexados para convertirse en la señal de reloj  $clkT_n$  que posibilita que el registro  $TCNT_n$  incremente o decremente su valor, en función del modo de operación seleccionado. Cuando el TCO alcanza su valor máximo y lo supera, una bandera llamada  $TOV_n$  (*Timer Overflow*) permite generar una interrupción.

**Figura 6.1**

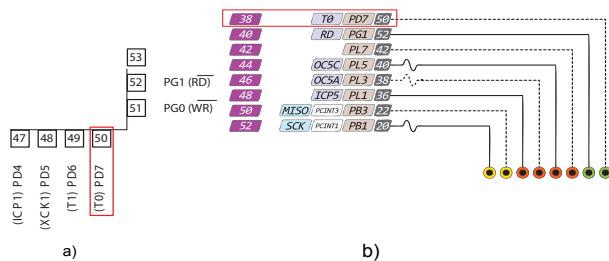
Diagrama de bloques del contador del timer 0



Tomado de Atmel, 2014, p. 117

**Imagen 6.1**

Distribución de terminales



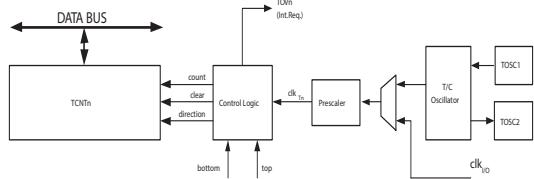
(a) en el chip. 2; (b) en la placa Arduino MEGA

Tomado de Atmel, 2014, p.

De forma similar, está el TC2, pero con un par de diferencias que se advierten a primera vista en la figura 6.2. Una de ellas es que este TC no posee terminal Tn. No obstante, posee dos terminales, denominados TOSC1 y TOSC2, que permiten conectar un cristal de cuarzo con una frecuencia de 32768 Hz y generar una señal de reloj exacta de un segundo, gracias al módulo T/C Oscillator y al bloque Prescaler. Esto, a su vez, ayuda a obtener un reloj en tiempo real (RTC) bastante confiable. Por lo demás, el TC2 tiene el mismo funcionamiento que el TC0.

**Figura 6.2**

Diagrama de bloques del contador del timer 2



**Nota.** En la tarjeta Arduino MEGA, no están disponibles los terminales TOSC1 y TOSC2, que se ubican en los puertos PG4 y PG3, respectivamente.

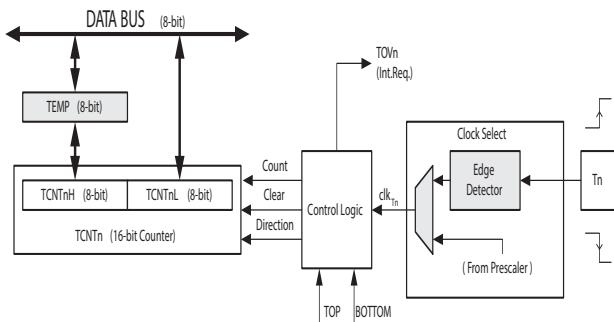
Tomado de Atmel, 2014, p. 171

Por otro lado, en la figura 6.3 se puede apreciar la estructura interna de un TC de 16 bits. En este caso, esta estructura es la misma para los TC 1, 3, 4 y 5. El subíndice n indica el número del TC a usar. Vale señalar que la estructura interna en ambos casos (8 y 16 bits) es similar. La principal diferencia radica en que el TC de 16 bits tiene dos registros de 8 bits, llamados TCNTnH y TCNTnL, con lo cual puede alcanzar un número mayor de combinaciones, que se traduce en un número más alto de cuentas o, a su vez, en una base de tiempo más grande. Los terminales T1, T3, T4 y T5 están disponibles en los pines 49, 8, 27 y 37 del microcontrolador ATmega2560, respectivamente. El terminal T5 se encuentra en el puerto PL2 y en el terminal 47 del Arduino MEGA.

**Nota.** En la tarjeta Arduino MEGA, no están disponibles los terminales T1, T3 y T4, que se encuentran en los puertos PD6, PE6 y PH7, respectivamente.

**Figura 6.3**

Diagrama de bloques de los contadores de 16 bits



Tomado de Atmel, 2014, p. 139

### 6.3 Modos de operación

En el caso de los TC, se dispone de registros de control y estado que permiten seleccionar el modo de operación deseada, así como conocer el valor en tiempo real de la cuenta que lleva el TC, o saber si alguna bandera de estado se activó o no. El truco para comprender todo el potencial que tiene un microcontrolador está en saber manipular las diferentes combinaciones que se puede realizar con estos registros. No todos se utilizan al mismo tiempo; en algunos casos, basta con utilizar uno o dos de ellos para que el periférico pueda trabajar. En otros casos, es probable que se deban usar todos los registros para un funcionamiento completo. Entre ellos, se tiene generación de ondas de ancho de pulso variable, o PWM (*Pulse Width Modulation*, por sus términos en inglés), que se usa, principalmente, para dosificar o controlar la cantidad de energía que alimenta algún equipo o dispositivo. Estas ondas, por lo general, se las puede encontrar en control de velocidad de motores eléctricos, control de iluminación, entre otras. Además, se pueden generar señales de frecuencia variable con niveles de precisión bastante altos.

El TC0 y el TC2 tienen cuatro modos de operación, que se sintetizan en la tabla 6.1; dos de ellos poseen ciertas particularidades que detallaremos más adelante. El TC1, TC3, TC4 y TC5 tienen cinco modos de operación; sin embargo, existen variantes y consideraciones especiales que se deben tomar

en cuenta en cada uno de ellos. En la tabla 6.2 se detalla cada uno de los modos y sus diferencias.

**Tabla 6.1**

Modos de operación de los TC0 y TC2

WGM n2	WGM n1	WGM n0	Modo de operación	Valor TOP
0	0	0	Normal	0xFF
0	0	1	PWM, Phase Correct	0xFF
0	1	0	CTC	OCR0A
0	1	1	Fast PWM	0xFF
1	0	0	Reservado	-
1	0	1	PWM, Phase Correct	OCR0A
1	1	0	Reservado	-
1	1	1	Fast PWM	OCR0A

Tomado de Atmel, 2014, p. 128

**Tabla 6.2**

Modos de operación de los TC1, TC3, TC4 y TC5

WGM n3	WGM n2	WGM n1	WGM n0	Modo de operación	Valor TOP
0	0	0	0	Normal	0xFF
0	0	0	1	PWM, Phase Correct, 8-bit	0xFF
0	0	1	0	PWM, Phase Correct, 9-bit	OCR0A
0	0	1	1	PWM, Phase Correct, 10-bit	0xFF
0	1	0	0	CTC	-
0	1	0	1	Fast PWM, 8-bit	OCR0A
0	1	1	0	Fast PWM, 9-bit	-
0	1	1	1	Fast PWM, 10-bit	OCR0A

1	0	0	0	PWM, Phase and Frequency Correct	0xFF
1	0	0	1	PWM, Phase and Frequency Correct	0xFF
1	0	1	0	PWM, Phase Correct	OCR0A
1	0	1	1	PWM, Phase Correct	0xFF
1	1	0	0	CTC	-
1	1	0	1	Reservado	OCR0A
1	1	1	0	Fast PWM	-
1	1	1	1	Fast PWM	OCR0A

Tomado de Atmel, 2014, p. 145

### 6.3.1 Modo normal

Es el modo más simple de operación del TC tanto de 8 bits como el de 16 bits. En este modo, una señal proveniente, ya sea del *prescaler* del TC o externa, a través del pin Tn (excepto el TC2), permite incrementar el valor del registro TCNTn desde 0 hasta su valor tope (TOP).

En el modo normal de 8 y 16 bits, para la configuración se debe seleccionar el modo de operación y la señal de reloj que ingresa al contador para incrementar su cuenta. Esto se logra modificando los bits WGM (*Waveform Generation Mode*) del registro TCCRnA y los tres bits menos significativos del registro TCCRnB, llamados *Clock Select* (CS) (ver figuras 6.4 a 6.9). Estos tres bits generan ocho posibles combinaciones (ver tablas 6.3 y 6.4). Nótese que las figuras 6.6 y 6.7, y la tabla 6.4 aplican para el TC2, mientras que las figuras 6.8 y 6.9 lo son para el TCn.

**Figura 6.4**

*Registro de control A del TC0*

bit	7	6	5	4	3	2	1	0
<b>TCCR0A</b>	COM0A1	COM0A0	COM0B1	COM0B0	-	-	WGM01	WGM00
Acción	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Estado inicial	0	0	0	0	0	0	0	0

Tomado de Atmel, 2014, p. 126

**Figura 6.5**

*Registro de control B del TC0*

bit	7	6	5	4	3	2	1	0
<b>TCCR0B</b>	FOC0A	FOC0B	-	-	WGM02	CS02	CS01	CS00
Acción	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Estado inicial	0	0	0	0	0	0	0	0

Tomado de Atmel, 2014, p.129

**Tabla 6.3**

*Descripción de los bits de selección de reloj del TCn (excepto TC2)*

CSn2	CSn1	CSn0	Descripción
0	0	0	Sin fuente de reloj. TC detenido
0	0	1	fosc / 1. Sin Prescaler
0	1	0	fosc / 8. Prescaler de 8
0	1	1	fosc / 64. Prescaler de 64
1	0	0	fosc / 256. Prescaler de 256
1	0	1	fosc / 1024. Prescaler de 1024
1	1	0	Fuente de reloj externa en el Pin T0. Flanco de bajada
1	1	1	Fuente de reloj externa en el Pin T0. Flanco de subida

Tomado de Atmel, 2014, p. 150

**Figura 6.6**

*Registro de control A del TC2*

bit	7	6	5	4	3	2	1	0
<b>TCCR2A</b>	COM2A1	COM2A0	COM2B1	COM2B0	-	-	WGM21	WGM20
Acción	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Estado inicial	0	0	0	0	0	0	0	0

Tomado de Atmel, 2014, p. 182

**Figura 6.7**

Registro de control B del TC2

bit	7	6	5	4	3	2	1	0
TCCR2B	FOC2A	FOC2B	-	-	WGM22	CS22	CS21	CS20
Acción Estado initial	R/W 0							

Tomado de Atmel, 2014, p. 185

**Tabla 6.4**

Descripción de los bits de selección de reloj del TC2

CS22	CS21	CS20	Descripción
0	0	0	Sin fuente de reloj. TC detenido
0	0	1	fosc/ 1. Sin Prescaler
0	1	0	fosc/ 8. Prescaler de 8
0	1	1	fosc/ 32. Prescaler de 32
1	0	0	fosc/ 64. Prescaler de 64
1	0	1	fosc/ 128. Prescaler de 128
1	1	0	fosc/ 256. Prescaler de 256
1	1	1	fosc/ 1024. Prescaler de 1024

Tomado de Atmel, 2014, p. 186

**Figura 6.8**

Registro de control A del TCn

bit	7	6	5	4	3	2	1	0
TCCRnA	COMnA1	COMnA0	COMnB1	COMnB0	COMnC1	COMnC0	WGMn1	WGMn0
Acción Estado initial	R/W 0							

Tomado de Atmel, 2014, p. 154

**Figura 6.9**

Registro de control B del TCn

bit	7	6	5	4	3	2	1	0
TCCRnB	ICNCn	ICESn	-	WGMn3	WGMn2	CSn2	CSn1	CSn0
Acción Estado initial	R/W 0							

Tomado de Atmel, 2014, p. 156

En las figuras 6.10 y 6.11 se ilustra cómo los temporizadores de 8 y 16 bits incrementan su valor desde cero

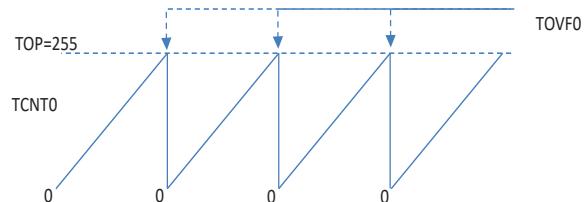
hasta su valor tope (TOP), de forma cíclica. En este modo, el TC se desborda automáticamente al llegar a su valor máximo, y reinicia su cuenta desde 0. Cuando esto sucede, una bandera, llamada *Timer Overflow* (TOVn) se activa, lo que permite desencadenar una interrupción. Esta puede ser utilizada para programar una base de tiempo, que se puede calcular con la ecuación 6.1. En ella, TOP corresponde al valor máximo de cada TC; N, al valor de *prescaler* seleccionado; y *fosc*, a la frecuencia de oscilación del microcontrolador.

$$t = \frac{TOP * N}{fosc} \quad (6.1)$$

Este modo de operación posibilita generar bases de tiempo de una forma sencilla, aunque su exactitud no sea la más adecuada, como se muestra en la tabla 6.5. No obstante, también se lo usa para sistemas de conteo o, incluso, para generar un reloj en tiempo real, como en el caso de TC2.

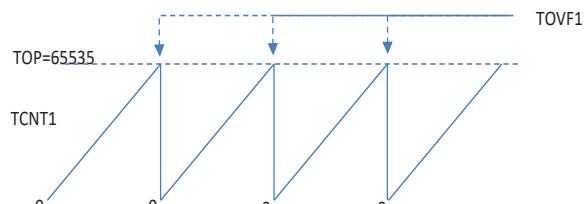
**Figura 6.10**

Representación gráfica del funcionamiento del TC de 8 bits



**Figura 6.11**

Representación gráfica del funcionamiento del TC de 16 bits



Para ilustrar, de mejor manera, el modo normal, planteamos un problema modelo y lo desarrollaremos paso a paso. Es importante indicar que se requiere haber revisado los temas de los capítulos anteriores, como manejo de puertos de entrada y salida, e interrupciones.

**Ejercicio 6.1:** Generar 3 bases de tiempo de aproximadamente 1 ms, 2 ms y 4 ms, que sirvan para complementar el estado lógico de los puertos PL7, PJ1 y PA0. Usar los TC 0, 2 y 4.

### Solución:

Para resolver el problema planteado, se debe utilizar el modo normal de los TC 0, 2 y 4. En función al requerimiento de tiempo, se calculan los posibles valores que se pueden obtener y se seleccionan los más adecuados. Además, se incluye el concepto de interrupciones, visto en el capítulo 4.

Como primer paso, se selecciona el modo de operación como modo normal en los registros TCCR0A, TCCR2A, TCCR4A, TCCR0B, TCCR2B, TCCR4B. Además, en estos últimos tres registros también se selecciona el valor de *prescaler*, calculado con la ecuación 6.1. De acuerdo con los resultados obtenidos en la tabla 6.5, se escogen los *prescaler* de 64, 128 y 1 para los TC 0, 2 y 4, respectivamente. En este caso, el problema no pide un valor exacto, por lo que se pueden aproximar sus valores. En el código se han escrito los valores de los registros en formato hexadecimal; no obstante, se lo podría hacer en formato binario, decimal o, incluso, usando algunos macros.

**Tabla 6.5**

Possibles bases de tiempo usando el modo normal. Ejercicio 6.1

**Prescaler para TC0, TC4**

TOP	1	8	-	64	-	256	1024
255	15,9375 [us]	127,5 [us]	-	1020 [us]	-	4080 [us]	16320 [us]
65535	4,0959375 [ms]	32,7675 [ms]	-	262,14 [ms]	-	1048,56 [ms]	4194,24 [ms]

**Prescaler para TC2**

255	1	8	32	64	128	256	1024
	15,9375 [us]	127,5 [us]	510	1020 [us]	2040 [us]	4080 [us]	16320 [us]

**Nota.** En el modo normal, no se pueden seleccionar valores exactos de tiempo, ya que dependen únicamente del valor de frecuencia de reloj (16 MHz) y de los valores de prescaler disponibles.

Luego, se debe activar la interrupción por *Timer Overflow* (TOIE) en cada uno de los TC. Esto se logra activando el bit correspondiente en los registros TIMSKn, que se muestran en las figuras 6.12 y 6.13. El código resultante se ha guardado en una función llamada config\_tc().

**Figura 6.12**

Registro de enmascaramiento de interrupciones de los TC de 8 bits. Ejercicio 6.1

bit	7	6	5	4	3	2	1	0
TIMSK <sub>n</sub>	-	-	-	-	-	OCIE <sub>nB</sub>	OCIE <sub>nA</sub>	TOIE
Acción	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Estado inicial	0	0	0	0	0	0	0	0

Tomado de Atmel, 2014, p. 131

**Figura 6.13**

Registro de enmascaramiento de interrupciones de los TC de 16 bits. Ejercicio 6.1

bit	7	6	5	4	3	2	1	0
TIMSK <sub>n</sub>	-	-	ICIE <sub>n</sub>	-	OCIE <sub>nB</sub>	OCIE <sub>nB</sub>	OCIE <sub>nA</sub>	TOIE
Acción	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Estado inicial	0	0	0	0	0	0	0	0

Tomado de Atmel, 2014, p. 161

Además, el problema nos solicita invertir el estado lógico de los puertos PL7, PJ1 y PA0. Para ello, se define una función llamada config\_io() en la que se los determina como salida.

Finalmente, se escriben las subrutinas de interrupción que atenderán cada uno de los eventos generados por los TC, y se colocan los códigos que permitirán invertir el estado lógico de los puertos, cada vez que suceda una interrupción.

A continuación, se presenta el código completo que cumple con las especificaciones del problema. Además, la imagen 6.2 presenta las ondas capturadas con un osciloscopio. En ella, se aprecian los valores medidos por el equipo, lo que corrobora la exactitud de la programación.

```

#include <avr/io.h>
#include <avr/interrupt.h>

ISR(TIMER0_OVF_vect){
    PORTL ^= 0X80; // Complemento el estado lógico del puerto PL7
}

ISR(TIMER2_OVF_vect){
    PORTJ ^= 0X02; // Complemento el estado lógico del puerto PJ1
}

ISR(TIMER4_OVF_vect){
    PORTJ ^= 0X01; // Complemento el estado lógico del puerto PA0
}

void config_tc(){
    TCCR0A = 0X00; // WGM00=0, WGM01=0
    TCCR2A = 0X00; // WGM20=0, WGM21=0
    TCCR4A = 0X00; // WGM40=0, WGM41=0
    TCCR0B = 0X03; // Prescaler de 64 en TC 0
    TCCR2B = 0X05; // Prescaler de 128 en TC 2
    TCCR4B = 0X01; // Prescaler de 1 en TC 4
    TIMSK0= 0X01; // Habilito interrupción de Overflow en TC0
    TIMSK2= 0X01; // Habilito interrupción de Overflow en TC1
    TIMSK4= 0X01; // Habilito interrupción de Overflow en TC1
}
void config_io(){
    DDRL |= 0X80; // PL7 como salida
    DDRJ |= 0X02; // PJ1 como salida
    DDRA |= 0x01; // PA0 como salida
}
int main(void)
{
    config_tc();
    config_io();
    sei(); // habilitación global de interrupciones
    while(1)
    {
        // No se ejecuta ningún programa de forma cíclica
    }
}

```

### Imagen 6.2

Ondas generadas usando el modo normal de los TCO, TC2 y TC4.  
Ejercicio 6.1



### 6.3.2 Modo de comparación CTC

En este modo, el valor creciente del TCNTn se compara con los registros OCRnA, OCRnB, OCRnC e ICRn. Cuando el TCNTn es igual al valor de estos registros, se pueden desencadenar tres posibles comportamientos (invertir su estado, poner su estado en bajo o poner su estado en alto), en los puertos asociados OCnA, OCnB y OCnC (solo para 16 bits), como se ilustra en las figuras 6.14 y 6.15. Estos comportamientos pueden ser seleccionados en el registro TCCRnA, a través de los bits COM, como se señala en la tabla 6.6.

Solo los registros OCRnA e ICRn pueden ser utilizados para definir el valor máximo de comparación o TOP. Además, es necesario seleccionar el modo de operación en los registros TCCRnA y TCCRnB, a través de los bits WGM (*waveform generation mode*), como se muestra en la figura 6.1 e imagen 6.1.

Tabla 6.6

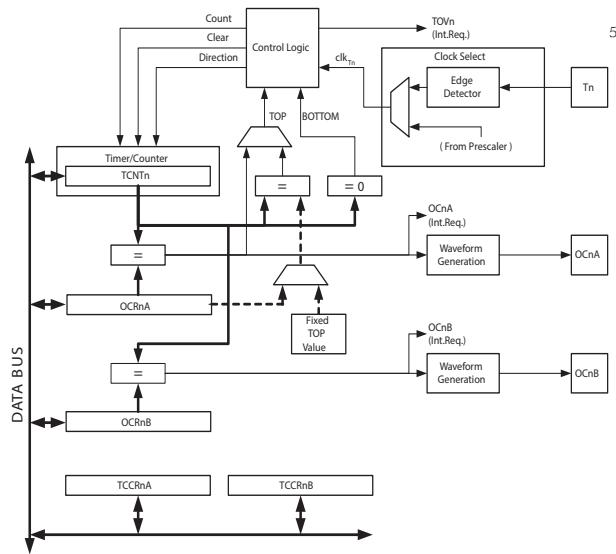
Descripción de los bits COM de los TCn en el Modo CTC

COMnX1	COMnX0	Descripción
0	0	La salida OC0X está desconectada
0	1	La salida OC0X invierte su estado ( <i>toggle</i> )
1	0	El estado de la salida OC0X se pone en bajo
1	1	El estado de la salida OC0X se pone en alto

Tomado de Atmel, 2014, p. 126

Figura 6.14

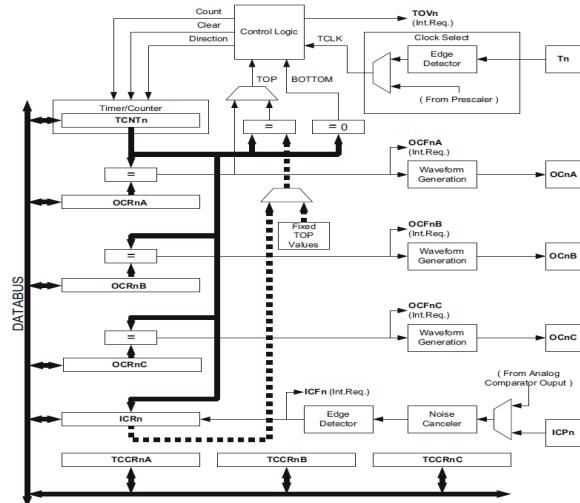
Diagrama de bloques del TCO



5

Figura 6.15

Diagrama de bloques de los TC de 16 bits



Tomado de Atmel, 2014, p. 134

En el modo normal, generar bases de tiempo exactas era imposible. Sin embargo, en este modo, ese problema desaparece, ya que se puede definir el valor TOP al cual el TC puede llegar por medio del registro OCRnA. Además, el verdadero potencial de este modo de operación está en la generación de ondas cuadradas a diversas frecuencias e, incluso, con diferentes desfases entre ellas. La ecuación 6.2 permite obtener el valor de frecuencia para este modo de operación, donde  $f_{OCnx}$  es la frecuencia de salida;  $f_{osc}$ , la frecuencia de oscilación del microcontrolador;  $N$ , el valor de prescaler; y  $TOP$ , el valor máximo al cual puede llegar el TC.

$$f_{OCnx} = \frac{f_{osc}}{2 * N * (1 + TOP)} \quad (6.2)$$

**Ejercicio 6.2:** Generar dos ondas cuadradas de 1,25 KHz por los terminales PB7 y PG5.

**Solución:**

El siguiente código permite generar dos ondas por los terminales PB7 y PG5, asociados a las salidas OC0A y OC0B. El prescaler seleccionado es 64 y el valor del registro OCR0A es 99. Con esto se puede

obtener una frecuencia de trabajo de 1,25 KHz o un periodo de 800 us, como se ilustra en la imagen 6.3. En ella, se aprecia el valor capturado por el osciloscopio, con lo cual podemos verificar los cálculos matemáticos. En la figura 6.16, se observa la forma en la que las ondas se generan, producto de su comparación entre los registros TCNT0 y OCR0A.

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

int main(void)
{
    TCCR0A = 0B01010010; // La salida OC0A y OC0B hacen un TOGGLE
    TCCR0B = 0B00000011; // Prescaler de 64
    OCR0A = 99;           // El TC se encera al llegar al valor TOP = 100
    OCR0B = 49;           // El OC0B hace un toggle al llegar al valor de 49
    DDRB |= 0X80;          // Habilito la salida PB7 (OC0A)
    DDRG |= 0X20;          // Habilito la salida PG5 (OC0B)

    while (1)
    {
    }
}
```

Imagen 6.3

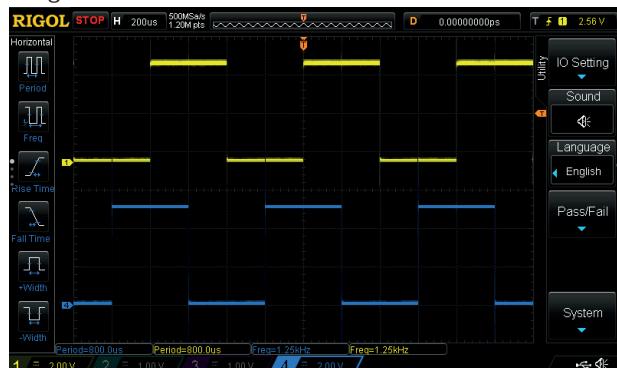
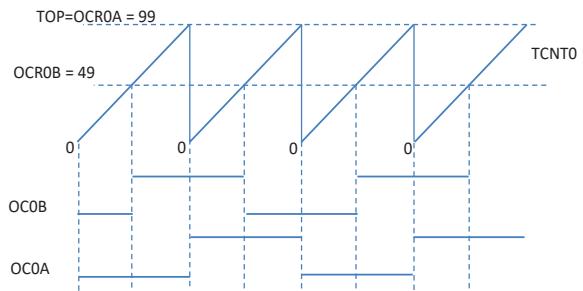


Figura 6.16

Generación de ondas en OC0A y OC0B del TCO. Ejercicio 6.2



Del mismo modo, se puede utilizar cualquiera de los TC de 16 bits para obtener tres ondas simultáneas. En total, este microcontrolador es capaz de generar hasta 16 señales, con rangos de frecuencia prácticamente ilimitados. Las principales aplicaciones en las que se puede aprovechar al máximo este periférico, tienen que ver con la generación de señales de control para dispositivos electrónicos de potencia. Transistores, MOSFET, IGBT, SCR y TRIACs son apenas algunos ejemplos de semiconductores que pueden ser controlados.

**Ejercicio 6.3:** Generar 3 ondas cuadradas usando el TC1, con una frecuencia de 1KHz.

### Solución:

En el siguiente código se muestra el algoritmo para generar 3 ondas cuadradas, usando el TC1. Para ello, se ha seleccionado el modo de trabajo en el que el registro ICR1 es el valor TOP. Ahora el *prescaler* seleccionado es de 1 y las tres señales están desfasadas simétricamente. La imagen 6.4 muestra la captura hecha con un osciloscopio. La frecuencia resultante es de 1 KHz, que puede ser fácilmente comprobada usando la ecuación 6.2.

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

int main(void)
{
    TCCR1A = 0B01010010; // La salida OC1A, OC1B y OC1C hacen un TOGGLE
    TCCR1B = 0B00011001; // Prescaler de 1
    OCR1A = 7999;         // El TC se encera al llegar al valor TOP = 7999
    OCR1B = 5333;         // El OC1B hace un toggle al llegar al valor de 5333
    OCR1C = 2666;         // El OC1C hace un toggle al llegar al valor de 2666
    DDRB |= 0XE0;          // Habilito la salida OC1A, OC1B y OC1C

    ICR1 = 7999;           // Se define el valor TOP
    while (1)
    {
    }
}
```

**Imagen 6.4**

Ondas de salida por OC1A, OC1B y OC1B del TC1. Ejercicio 6.3



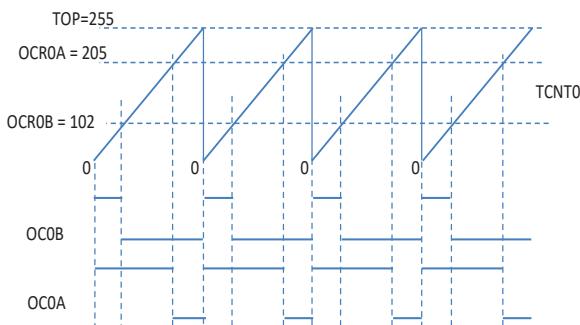
### 6.3.3 Modo Fast PWM

Este modo de operación posibilita generar ondas PWM (*Pulse Width Modulation*) de alta frecuencia. Se caracteriza por que la onda se genera durante la pendiente de subida del TC, que va desde 0 hasta su valor TOP (ver figura 6.17). Además, la onda PWM generada puede ser invertida o no invertida. Esto se logra seleccionando adecuadamente los bits COM en el registro TCCRnA, como se detalla en la tabla 6.7.

Al igual que el modo CTC, el modo Fast PWM permite seleccionar el valor TOP a través del registro OCRnA, para los TC de 8 bits. En cambio, para los TC de 16 bits se pueden usar los registros OCRnA e ICRn, así como valores predefinidos para 8 bits, 9 bits y 10 bits (ver tablas 6.1 y 6.2). Para regular el ancho de pulso de la señal, se pueden utilizar los registros OCR0A y OCR0B en los TC de 8 bits. Además, en los TC de 16 bits se puede usar el registro OCRnC.

**Figura 6.17**

Generación de ondas en OC0A y OC0B del TCO



**Tabla 6.7**

Descripción de los bits COM de los TCn en el Modo Fast PWM

COMnX1	COMnX0	Descripción
0	0	La salida OC0X está desconectada
0	1	Caso especial*
1	0	PWM no invertida
1	1	PWM invertida

\*Consulte el manual del fabricante

Tomado de Atmel, 2014, p. 126

**Ejercicio 6.4:** Generar dos ondas PWM de, aproximadamente, 1 KHz, usando el TCO.

### Solución:

A continuación, se muestra el código para generar dos ondas PWM usando el TCO. En este caso, el valor TOP es 255, que es valor máximo al cual el TCO puede llegar. El ancho de pulso es de 80 % para la salida OC0A y 60 % para la salida OC0B, aproximadamente. El tipo de onda que se desea obtener es una PWM no invertida. Estos valores se calculan estableciendo una relación directa entre el valor tope y el valor deseado. La ecuación 6.3 permite determinar el valor de frecuencia de la onda PWM donde  $f_{OCnXPWM}$  es la frecuencia de salida PWM;  $f_{osc}$  es la frecuencia de oscilación del microcontrolador;  $N$ , el valor de prescaler; y  $TOP$ , el valor máximo al cual puede llegar el TC. La imagen 6.5 presenta el resultado obtenido, junto con las lecturas que arroja el osciloscopio. Nuevamente, las lecturas del osciloscopio indican la gran exactitud con la que se puede trabajar en estos modos de operación.

$$f_{OCnXPWM} = \frac{f_{osc}}{N * (1 + TOP)} \quad (6.3)$$

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

int main(void)
{
    TCCR0A = 0B10100011;      // Por OC0A y OC0B PWM no invertida
    TCCR0B = 0B00000011;      // Prescaler de 64
    OCR0A = 205;              // PWM por OC0A con ancho de pulso del ~80 %
    OCR0B = 102;              // PWM por OC0B con ancho de pulso del ~40 %
    DDRB |= 0X80;             // Habilito la salida PB7 (OC0A)
    DDRG |= 0X20;             // Habilito la salida PG5 (OC0B)
    while(1)
    {
    }
}
```

Imagen 6.5

Ondas PWM por OC0A y OC0B. Ejercicio 6.4



No obstante, en ciertas aplicaciones es necesario que la frecuencia generada sea exacta. En este caso, se debe recurrir al modo en el que el valor TOP es el registro OCR0A. Esto provoca que la salida OC0A no pueda ser utilizada como salida PWM, ya que el registro que regulaba su ancho de pulso ahora está siendo usado para definir el valor tope.

**Ejercicio 6.5:** Generar una onda PWM de 8,125 KHz usando el TCO.

## Solución:

Para ello, se utiliza la ecuación 6.3 y se despeja la variable TOP. Al tener como datos la frecuencia del microcontrolador (16 MHz) y la frecuencia deseada, las únicas incógnitas son el *prescaler* y el valor TOP. La tabla 6.8 contiene todos los posibles valores que puede tomar el registro OCR0A como valor tope para cada valor de *prescaler*. Sin embargo, los registros únicamente pueden almacenar valores enteros, por lo que es necesario identificar cuál de estos valores obtenidos genera la señal PWM con menor error.

La tabla 6.9 muestra los valores de frecuencia que se generaría si se hubiesen seleccionado los valores obtenidos en la tabla 6.8 a su valor más próximo. En el primer caso, cuando el *prescaler* es igual a 1, el valor de TOP necesario supera la capacidad del registro, lo cual es imposible conseguir. En los otros casos, se puede asignar un valor al registro OCR0A, que satisfaga el problema.

El valor de TOP que genera menor error es con el *prescaler* de 8. Depende del criterio del diseñador evaluar si este error es aceptable o no. Si este no es el caso, la recomendación sería usar un *timer* de 16 bits para disminuir el error.

Tabla 6.8

Valores de TOP en función del prescaler. Ejercicio 6.5

Prescaler	1	8	64	256	1024
TOP	1968,23077	245,153846	29,7692	6,692308	0,923

Tabla 6.9

Valores de frecuencia y errores según valores de la tabla 6.4

Prescaler	1	8	64	256	1024
Frecuencia obtenida	Valor máximo de TOP=255	8130,1	8333,3	8928,6	7812,5
Error en frecuencia [Hz]	-	5,1	208,3	-312,5	-312,5

A continuación, se muestra el código necesario para resolver el problema planteado. La imagen 6.6 expone la onda PWM medida con un osciloscopio. Las mediciones muestran una frecuencia de 8,13 KHz y un ancho de pulso de 21 %.

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

int main(void)
{
    TCCR0A = 0B00100011; // Por OC0B PWM no invertida
    TCCR0B = 0B00001010; // Prescaler de 8
    OCR0A = 245;          // OCR0A se usa como TOP
    OCR0B = 50;           // PWM por OC0B con ancho de pulso del ~21 %
    DDRG |= 0X20;         // Habilito la salida PG5 (OC0B)

    while (1)
    {
    }
}
```

**Imagen 6.6**

Onda PWM por OC0B con valor “exacto”. Ejercicio 6.5



**Ejercicio 6.6:** Generar tres ondas PWM de 8,125 KHz usando el TC1.

#### Solución:

Como hemos señalado anteriormente, estas ondas PWM son de mucha utilidad para la regulación de potencia, rectificadores, convertidores análogo-digitales, entre otros. Por ello, muchas veces es importante generar más de dos señales simultáneas y con un valor de frecuencia exacto. Para esto, los TC de 16 bits son ideales, ya que permiten definir el valor TOP en un registro adicional llamado ICRn; de esta forma, se libera el registro OCRnA y, por ende, su salida OCnA asociada. A continuación, se muestra el código necesario para generar una onda

de 8,125 KHz y el resultado medido con un osciloscopio en los tres canales, OC1A, OC1B y OC1C, en la imagen 6.7. De forma análoga al problema anterior, se escoge el *prescaler* que permite obtener un menor error.

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

int main(void)
{
    TCCR1A      = 0B10101010; // Por OC1A, OC1B y OC1C PWM no invertida
    TCCR1B      = 0B00011001; // Prescaler de 1
    ICR1       = 1968; // Valor TOP
    OCR1A      = 1476; // PWM por OC1A con ancho de pulso del ~75 %
    OCR1B      = 984;    // PWM por OC1B con ancho de pulso del ~50 %
    OCR1C      = 492;    // PWM por OC1C con ancho de pulso del ~25 %
    DDRB |= 0XE0; // Habilito la salida OC1A, OC1B y OC1C

    while (1)
    {
    }
}
```

**Imagen 6.7**

Onda PWM por OC1A, OC1B y OC1C con valor “exacto”. Ejercicio 6.6



**Nota.** En el modo fast PWM, es imposible generar un ancho de pulso de 0 %. Esto puede generar pérdidas eléctricas o ruidos tanto eléctricos como mecánicos.

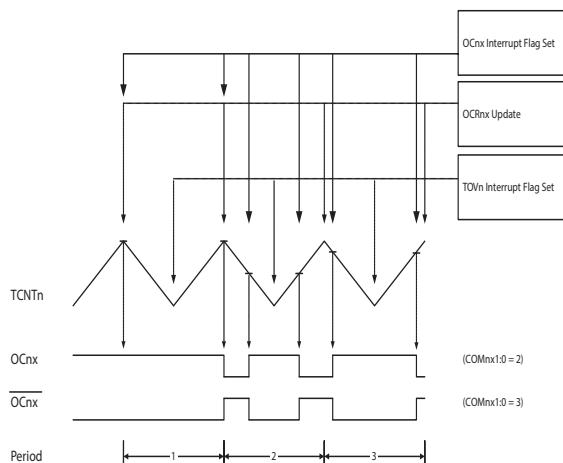
Cabe indicar, además, que para los TC de 16 bits existen algunas otras variantes respecto al modo Fast PWM, como se muestra en la tabla 6.2. Estas variantes se enfocan en la resolución que pueden tener las ondas generadas de 8, 9 y 10 bits. Se sugiere al lector probar de forma análoga a los ejercicios planteados con estas variantes, puesto que los códigos no se modificarán de forma significativa.

### 6.3.4 Modo Phase Correct PWM

Este modo de operación permite generar ondas PWM, con una mayor resolución y con una frecuencia más baja que en el caso del Modo Fast PWM. Esto se logra gracias a que la onda PWM se genera durante la pendiente de subida y bajada del timer. Este modo es recomendado cuando se trata de controlar motores eléctricos, ya que en valores bajos de relación de trabajo la salida es cero. En la figura 6.18 se aprecia el diagrama de tiempos para tres periodos. En ella se observa que la bandera de interrupción TOVn se genera cuando el TC llega a cero. Además, el registro usado para realizar la comparación OCRnX se actualiza en el extremo TOP del ciclo.

**Figura 6.18**

Diagrama de tiempos del modo PWM Phase Correct



Tomado de Atmel, 2014, p. 122

Para calcular la frecuencia de la onda de salida, se debe utilizar la ecuación 6.4.

$$f_{OCnxPWM} = \frac{f_{osc}}{2 * N * (TOP - 1)} \quad (6.4)$$

De manera similar al caso anterior, ilustramos el funcionamiento de este modo de operación a través de un ejemplo práctico.

**Ejercicio 6.7:** Generar tres ondas PWM de 1,125 KHz usando el TC3.

### Solución:

En este caso, se desea un valor de frecuencia exacto de 1,125 KHz. Para ello, se usará como TOP el registro ICR3. Este valor se calculará usando la ecuación 6.4. Los resultados se muestran en la tabla 6.10. El prescaler seleccionado es 1 y el valor tope 7112. Con estos valores, se puede obtener una frecuencia de 1,124 KHz.

**Tabla 6.10**

Valores de TOP en función del prescaler. Ejercicio 6.7

Prescaler	1	8	64	256	1024
TOP	7112,11	889,89	112,11	28,78	7,94

Para poder visualizar las señales PWM, se usan los puertos asociados al TC3. En este caso, son OC3A, OC3B y OC3C, que se encuentran en los puertos PE3, PE4 y PE5, respectivamente. Todas las ondas que se desean obtener deben ser del tipo No Invertidas. Este tipo de señal es la más usada para señales de control de motores, y se la puede observar en la imagen 6.8.

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

int main(void)
{
    TCCR3A = 0B10101010; // Por OC3A, OC3B y OC3C PWM no invertida
    TCCR3B = 0B00010001; // Prescaler de 1
    ICR3 = 7112; // Valor TOP
    OCR3A = 5334; // PWM por OC3A con ancho de pulso del ~75 %
    OCR3B = 3556; // PWM por OC3B con ancho de pulso del ~50 %
    OCR3C = 1778; // PWM por OC3C con ancho de pulso del ~25 %
    DDRE= 0X38; // Habilite la salida OC3A, OC3B y OC3C
    while (1)
    {
    }
}
```

### Imagen 6.8

Ondas PWM Phase Correct por OC3A, OC3B y OC3C. Ejercicio 6.7

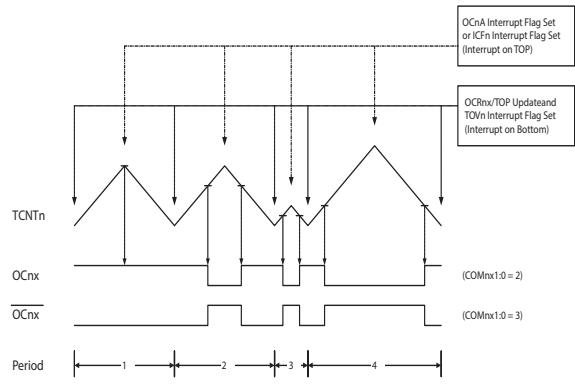


### 6.3.5 Modo Phase and Frequency Correct PWM

Este modo de operación solo está disponible en los TC de 16 bits. La principal diferencia con el modo PWM Phase Correct radica en que este modo permite únicamente definir los valores TOP, a través de los registros OCRnA e ICRn; además, el momento en el cual se realiza la actualización de los registros de comparación OCRnX se lleva a cabo cuando el TC llega a su valor mínimo. Por lo demás, este modo tiene las mismas aplicaciones y bondades que sus modos anteriores. Las sutiles diferencias se pueden apreciar en la figura 6.19.

**Figura 6.19**

Diagrama de tiempos del modo PWM Phase and Frequency Correct



Tomado de Atmel, 2014, p. 151

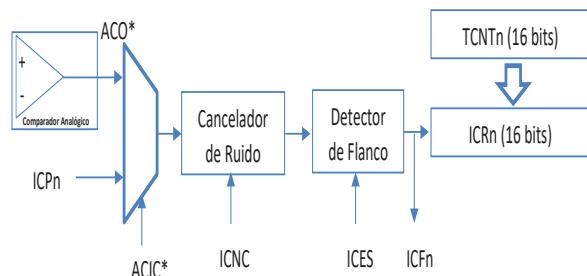
### 6.3.6 Unidad de captura

Los TC de 16 bits vienen equipados con una unidad que permite capturar eventos externos y generar una marca del tiempo en el cual ocurrieron. Estos eventos pueden ser registrados a través de los pines marcados como ICPn o de forma exclusiva para el TC1, a través de la unidad de comparación analógica. La marca de tiempo registrada puede ser usada para medir frecuencias, ciclos de trabajo u otras características que involucren una ocurrencia de tiempo. Esta unidad es de mucha utilidad cuando se tiene que trabajar con sensores; por ejemplo, los ultrásónicos, cuya señal de salida es un ancho de pulso digital que varía en función de la distancia.

Cuando una señal digital ingresa por el terminal ICPn o por el comparador analógico ACO (solo para el TC1), los flancos de subida o de bajada pueden ser filtrados por el bloque Cancelador de Ruido o detectados por medio del bloque Detector de Flanco (ver figura 6.20). Estos flancos generan una orden al registro ICRn para que, automáticamente, tome el valor que se encuentra en el registro TCNTn en ese instante de tiempo. Además, se puede generar, al mismo tiempo, una interrupción por captura si el bit ICIE n se encuentra habilitado en el registro TIMSKn, como se muestra en la figura 6.20.

**Figura 6.20**

Diagrama de bloques de la Unidad de Captura



Para ilustrar la funcionalidad de esta unidad de captura, planteamos un ejercicio muy común, que puede ser posteriormente utilizado para otros propósitos.

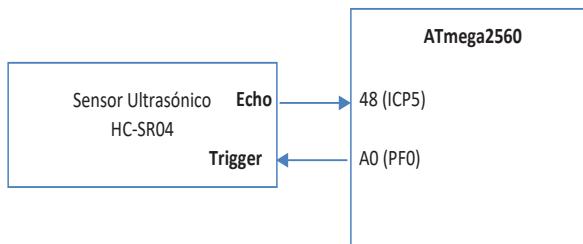
**Ejercicio 6.8:** Se desea medir distancia usando un sensor de ultrasonido HC-SR04, y su valor en centímetros enviarlo vía comunicación serial hacia el computador.

## Solución:

Como primer paso, se analiza el sistema en su conjunto y se bosqueja un diagrama de bloques, que permite visualizar la interacción entre los diferentes componentes. En este caso, el diagrama de bloques está compuesto por el sensor ultrasónico y el microcontrolador, como puede observarse en la figura 6.21.

**Figura 6.21**

Diagrama de bloques del sensor conectado al microcontrolador.  
Ejercicio 6.8

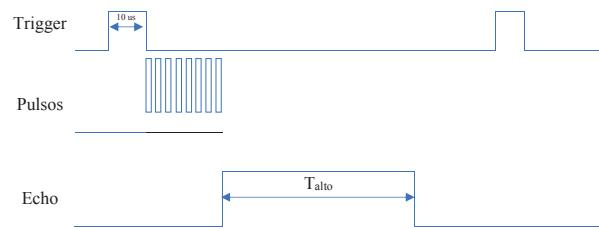


## Sensor ultrasónico

El dispositivo HC-SR04 es un sensor ultrasónico de muy bajo costo, que permite obtener medidas de distancia en un rango de hasta 4 m. Es utilizado en aplicaciones de robótica de entretenimiento. Se lo puede configurar con relativa facilidad. Posee cuatro terminales de conexión: VCC, GND, Echo y Trigger. Usando un puerto de salida del microcontrolador, se debe generar un pulso positivo no menor a 10us y aplicarlo en el terminal Trigger. El sensor recibe del microcontrolador un pulso de control, llamado Trigger, que le permite generar un tren de 8 pulsos a una frecuencia de 40 KHz, para producir una onda de ultrasonido. El terminal Echo, en cambio, genera un pulso cuyo ancho varía de acuerdo con el tiempo que tarda la señal en ir y volver del objeto (ver figura 6.22). Durante este tiempo, el terminal Echo cambia su estado a alto hasta que la onda regrese al receptor del módulo. Este pulso está directamente relacionado con la distancia, a través de la ecuación 6.5.

**Figura 6.22**

Diagrama de tiempos del sensor HC-SR04. Ejercicio 6.8



$$d = \frac{T_{alto} * V_{sonido}}{2} \quad (6.5)$$

Para medir este tipo de señales, es necesario utilizar la unidad de captura. Para ello, se debe configurar el TC5 en modo normal. No obstante, hay que tomar en cuenta la resolución en tiempo que tiene el TC5 y, por ende, la resolución en centímetros que el sensor es capaz de entregar. Según datos del fabricante, la distancia máxima a medir es 4 m y la precisión puede llegar hasta 3 mm. Esto quiere decir que se tendrán, aproximadamente, 1333 lecturas diferentes de distancia. Además, un incremento de 3 mm de distancia representa 17,7 [μs] de tiempo. Este dato es muy importante, ya que el TC5 debe ser capaz de satisfacer esta demanda por parte del sensor.

Con estos resultados preliminares obtenidos a partir de la hoja de especificaciones, se puede efectuar un cálculo sencillo para conocer cuál es el mejor valor de *prescaler* para el TC5. Si se aplica la ecuación 6.6, donde N son los diferentes valores de *prescaler* del TC5, y  $f_{osc}$  la frecuencia del microcontrolador, se puede deducir que el valor de *prescaler* que cumple con los requerimientos es N = 256. Esto significa que cada incremento del TC5 será en múltiplos de 16 [μs].

$$t_{pulso} = \frac{N}{f_{osc}} \quad (6.6)$$

Una vez configurado el TC5 en modo normal, el siguiente paso es configurar la unidad de captura en el registro TCCR5B, tal como se detalla en la

figura 6.9. El bit ICES5 permite escoger el flanco de la señal entrante, que en este caso será la señal Echo. La primera vez se detectará flanco positivo y después flanco negativo. También se habilitará la generación de interrupción, a través del bit ICIE5 en el registro TIMSK5 (ver figura 6.13). Cada vez que un flanco es detectado, se genera una interrupción donde se copia al valor de TCNT5 y se cambia el flanco a detectar para una próxima ocasión.

```
#define F_CPU 16000000UL
#include <avr/io.h>
#include <util/delay.h>

uint8_t Xcm=0, cont=0, Tsensor=0, distancia=0;
uint8_t dist[] = {0, 0, 0, 0, 'I', 'c', 'm', 'T', ' '}; // MCDU

void Bin2BCD(uint16_t x){
    dist[0]=(x/100)+48;
    dist[1]=((x%100)/10)+48;
    dist[2]=(((x%100)%10)/10)+48;
    dist[3]=(((x%100)%10)%10)+48;
}

void send_char(char data){
    while(!(UCSR0A & (1<<UDRE0)));
    UDR0 = data;
}

void send_string(char *s){
    while (*s){
        send_char(*s);
        s++;
    }
}

void medir_distancia(){
    while(!(TIFR5 & 0X20)); // Si es flanco de subida
    TCNT5 = 0; // Inicializo el TC5
    TIFR5 |= 0X20; // Bajo la bandera de interrupción
    TCCR5B ^= 0X40; // Invierto el flanco
    while(!(TIFR5 & 0X20)); // Si es flanco de bajada
    TIFR5 |= 0X20; // Bajo la bandera de interrupción
    TCCR5B ^= 0X40; // Invierto el flanco
    Xcm = ICR5*2/58;
}

int main(void)
{
    DDRB |= 0X80; // PF0-Trigger (A0)
    DDRF |= 0X01; // PF0-Trigger (A0)
    DDRL &= ~0X02; // L1 ICP5 (48)
    UBRR0 = 103; // 9600 bps
    UCSRB = (1<<TXEN0); // Transmisión de datos
    TCCR5A = 0X00; // Modo Normal TC5
    TCCR5B = 0X42; // TC5 N=8 y flanco de subida
    while (1)
    {
        PORTF |= 0X01;
        _delay_us(20);
        PORTF &= ~0X01;
        PORTB|= 0X80;
        medir_distancia();
        PORTB &= ~0X80;
        Bin2BCD(Xcm);
        send_string(dist);
        _delay_ms(500);
    }
}
```

# Capítulo 7

## Conversión analógica digital

### 7.1 Anotaciones preliminares sobre conversión analógica digital

La importancia de recibir y procesar señales de tipo analógico está íntimamente ligada con el entorno real de aplicación de los sistemas microcontrolados. En su mayoría, la interacción que nosotros, como humanos, percibimos en el ambiente es de tipo analógico: el sonido que escuchamos, el aroma que percibimos, la luz que ingresa por nuestros ojos, etcétera. Aunque para nuestros sentidos es común interpretar este tipo de información, para un sistema microcontrolado resulta un poco más complicado, ya que es necesario incluir sensores y transductores que permitan obtener esta información de fenómenos físicos, y transformarla a señales de tipo eléctricas.

Si bien en capítulos anteriores se ha interpretado a la información que ingresa y sale del microcontrolador de manera digital, con base en niveles lógicos TTL (0V-0L y 5V-1L), es necesario aclarar que no es la única manera en la que se puede interactuar. Existe variedad de microcontroladores en los cuales es posible interpretar señales de entrada de tipo analógicas, así como brindar salidas de este tipo. En el caso específico del microcontrolador estudiado en este libro, el ATmega2560, este posee solamente entradas de tipo analógico, mas no salidas. Una entrada de tipo analógico implica que el microcontrolador podrá recibir estas señales y, mediante un conversor, transformarlas a valores de tipo digital, que es el modo de trabajo del microcontrolador.

En este contexto, en el presente capítulo abordamos las características básicas de funcionamiento que ofrece el conversor analógico digital (ADC) del microcontrolador y su configuración, por medio de registros. Además, incluimos una serie de ejemplos que permitan entender las configuraciones del conversor través de registros.

### 7.2 Canales, voltaje de referencia y resultado

El conversor analógico digital (ADC) convierte un voltaje de entrada analógico en un valor que posee una resolución de 10 bits (0-1023). Se encuentra conectado a un multiplexor de 16 canales de entrada (ADC0, ADC2, ... ADC15), de voltaje simple referenciado a 0 V (GND), con los cuales es posible obtener 32 combinaciones de entradas diferenciales, con ganancias entre 1x, 10x y 200x, según la combinación seleccionada (Atmel, 2014). En la tabla 7.1 se pueden observar los bits MUX5:0 que permiten seleccionar los canales de entrada simples, mientras que la tabla 7.2 contiene un resumen de los canales diferenciales, según su ganancia. La columna MUX5:0 hace referencia a la configuración en bits que debe realizarse para obtener la funcionalidad descrita. Analizaremos, más adelante, el detalle del registro que maneja estos bits. La tabla muestra que en configuración de ganancia 1X los únicos canales que pueden utilizarse como negativos son: ADC1, ADC2, ADC9, ADC10; mientras que en ganancias de 10x y 200x se utilizan las mismas combinaciones para ambos casos, diferenciándose únicamente por las configuraciones de los bits relacionados con MUX5:0.

**Tabla 7.1**

Canales de entrada simple

MUX5:0	Canal simple	Pin asociado (empaquetado TQFP 100 pines)
000000	ADC0	PF0
000001	ADC1	PF1
000010	ADC2	PF2
000011	ADC3	PF3
000100	ADC4	PF4
000101	ADC5	PF5
000110	ADC6	PF6
000111	ADC7	PF7
100000	ADC8	PK0
100001	ADC9	PK1
100010	ADC10	PK2
100011	ADC11	PK3
100100	ADC12	PK4
100101	ADC13	PK5
100110	ADC14	PK6
100111	ADC15	PK7

**Tabla 7.2**

Canales diferenciales y ganancias

MUX5:0	Entrada diferencial positiva	Entrada diferencial negativa	Ganancia
010000	ADC0	ADC1	
010001	ADC1	ADC1	
010010	ADC2	ADC1	
010011	ADC3	ADC1	
010100	ADC4	ADC1	
010101	ADC5	ADC1	
010110	ADC6	ADC1	
010111	ADC7	ADC1	
011000	ADC0	ADC2	
011001	ADC1	ADC2	
011010	ADC2	ADC2	

011011	ADC3	ADC2	<b>1X</b>
011100	ADC4	ADC2	
011101	ADC5	ADC2	
110000	ADC8	ADC9	
110001	ADC9	ADC9	
110010	ADC10	ADC9	
110011	ADC11	ADC9	
110100	ADC12	ADC9	
110101	ADC13	ADC9	
110110	ADC14	ADC9	
110111	ADC15	ADC9	
111000	ADC8	ADC10	
111001	ADC9	ADC10	
111010	ADC10	ADC10	
111011	ADC11	ADC10	
111100	ADC12	ADC10	
111101	ADC13	ADC10	
001000*	ADC0	ADC0	<b>10x</b>
001001*	ADC1	ADC0	
001100*	ADC2	ADC2	
001101*	ADC3	ADC2	
101000*	ADC8	ADC8	
101001*	ADC9	ADC8	
101100*	ADC10	ADC10	
101101*	ADC11	ADC10	
001010*	ADC0	ADC0	<b>200x</b>
001011*	ADC1	ADC0	
001110*	ADC2	ADC2 1X	
001111*	ADC3	ADC2	
101010*	ADC8	ADC8	
101011*	ADC9	ADC8	
101110*	ADC10	ADC10	
101111*	ADC11	ADC10	

\*Ganancias de 10x y 200x no deben ser usadas con voltajes de operación por debajo de los 2,7 V; caso contrario, la precisión de la conversión puede verse afectada (Atmel, 2014).

Tomado de Atmel, 2014, p. 282-284

Con el fin de realizar una conversión, es necesario conocer el voltaje usado como referencia; es decir,

el límite máximo que el ADC interpreta como el valor decimal 1023.

La fórmula para calcular el valor de conversión ADC para una entrada simple es (Atmel, 2014, p. 280):

$$ADC = \frac{V_{in} * 1024}{V_{ref}} \quad (7.1)$$

$V_{in}$  es el voltaje analógico conectado a uno de los canales simples; ADC y  $V_{ref}$  son los distintos valores que pueden asignarse como rangos de voltaje para la conversión. Este voltaje referencial puede ser fijado interna como externamente. Los voltajes internos pueden ser fijados en 2,56 V, 1,1 V y AVCC, o externamente mediante el pin AREF. El ADC tiene una entrada para colocar una fuente separada de voltaje en el pin AVCC, cuyo valor no debe diferir más de  $\pm 0,3$  V del voltaje de alimentación del microcontrolador; también puede ser seleccionado como un voltaje de referencia. El voltaje AREF puede ser fijado por una fuente externa, siempre que no supere el voltaje de entrada permitido por el microcontrolador; si se ha colocado un voltaje en este pin, no se deben usar las opciones internas de AVCC, 1,1 V y 2,56 V (Atmel, 2014, p. 268).

Una de las principales inquietudes al configurar el ADC tiene que ver con el voltaje de referencia adecuado para una determinada aplicación. Básicamente, el voltaje de referencia está asociado con la resolución de la conversión en relación con el voltaje de entrada aplicado. Se puede plantear el siguiente caso:

Supóngase que el voltaje máximo de entrada es 1,5 V en un canal simple cualquiera del ADC. ¿Cuál sería un voltaje de referencia adecuado para este caso? Tomando en cuenta que el voltaje de referencia establece el rango máximo de la entrada analógica, un de 1,1 V no sería viable. Sin embargo, se puede considerar la opción de 2,56 V y AVCC, como referencias internas.

Si se toma en cuenta que el microcontrolador está alimentado con 5 V, entonces se interpreta que el AVCC posee el mismo valor; por tanto, si se selecciona  $V_{ref}$  tiene asignado un valor de 5 V. En este caso, se puede analizar el equivalente de 1LSB (bit

menos significativo) en el valor de entrada analógica, mediante el uso de la expresión (7.1). Se aprecia que el cambio de 1LSB representa una variación en la entrada de alrededor de 4,88 mV.

$$1LSB = \frac{V_{in} * 1024}{5}$$

$$V_{in} = 4,88mV$$

Ahora se puede analizar el caso en el que  $V_{ref} = 2,56 V$

$$1LSB = \frac{V_{in} * 1024}{2,56}$$

$$V_{in} = 2,5 mV$$

En este último caso, se advierte que la resolución ha mejorado. Esto implica que es posible medir variaciones más pequeñas del voltaje de entrada, a medida que el  $V_{ref}$  disminuye; no obstante, es necesario asegurarse que el voltaje de entrada no supere el valor del  $V_{ref}$  establecido. El uso de AREF implicaría que externamente se puede colocar una fuente que se ajuste, de mejor manera, a los rangos de conversión que se desea utilizar, lo que permite contar con una mejor resolución de conversión en el voltaje de entrada.

En el caso de realizar una conversión mediante canales diferenciales, es necesario aplicar la ecuación 7.2 (Atmel, 2014, p. 280):

Tanto el voltaje de referencia como la elección de

$$ADC = \frac{(V_{POS} - V_{NEG}) * 512}{V_{ref}} \quad (7.2)$$

los canales (simple o diferencial), se pueden seleccionar mediante dos registros: ADMUX y ADCSRB.

La figura 7.1 permite observar el registro ADMUX, cuya configuración de bits es utilizada para establecer el voltaje de referencia  $V_{ref}$  y el canal, ya sea en modo simple o en modo diferencial. Hay que tomar en cuenta que para la selección de canales se deben configurar los bits MUX5:0 (ver tablas 7.1 y 7.2); sin embargo, en el registro ADMUX solo se encuentran los bits MUX4:0. El último bit (MUX5) debe configurarse en el registro ADCSRB (ver figura 7.2). Los demás bits de este registro no tienen injerencia en el voltaje de referencia y la selección de canal.

**Figura 7.1**

Bits de configuración del registro ADMUX

Tomado de Atmel, 2014, p. 281

**Figura 7.2**

bit	7	6	5	4	3	2	1	0
Acción	REFS1	REFS0	ADLAR	MUX4	MUX3	MUX2	MUX1	MUX0
Estado	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
inicial	0	0	0	0	0	0	0	0

Bits de configuración del registro ADCSRB

Tomado de Atmel, 2014, p. 282

bit	7	6	5	4	3	2	1	0
Acción	-	ACME	-	-	MUX5	ADTS2	ADTS1	ADTS0
Estado	R	R/W	R	R	R/W	R/W	R/W	R/W
inicial	0	0	0	0	0	0	0	0

Los bits REFS0:1, en el registro ADMUX, se utilizan para establecer el voltaje de referencia. Esta configuración puede realizarse con base en la tabla 7.3.

**Tabla 7.3**

Canales de entrada simple

REFS1	REFS0	Voltaje de referencia
0	0	AREF, Voltaje interno de referencia desactivado (1,1 V, 2,56 V, AVCC)
0	1	AVCC*
1	0	Voltaje interno 1,1V*
1	1	Voltaje interno 2,56V*

\* Se sugiere colocar un capacitor entre el pin AREF y tierra para aumentar la inmunidad al ruido.

Tomado de Atmel, 2014, p. 281

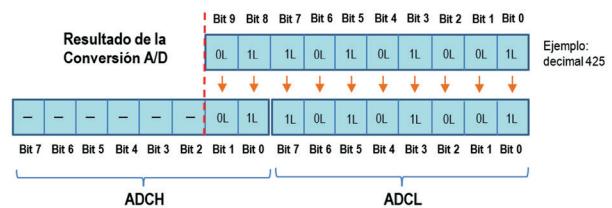
Cuando una conversión A/D ha finalizado, el resultado se almacena en los registros ADCH y ADCL (8 bits cada uno). Este resultado puede almacenarse de dos maneras distintas: con justificación a derecha y a izquierda.

La justificación a derecha se establece cuando el bit 5 del registro ADMUX es igual a cero (ADLAR = 0); es la más comúnmente utilizada, ya que permite obtener la mejor resolución posible del conversor. Esta

opción implica que el bit menos significativo de la conversión A/D se posicionará en el bit 0 del registro ADCL y se continuará llenando los bits hasta terminar en el bit 1 del registro ADCH (figura 7.3).

**Figura 7.3**

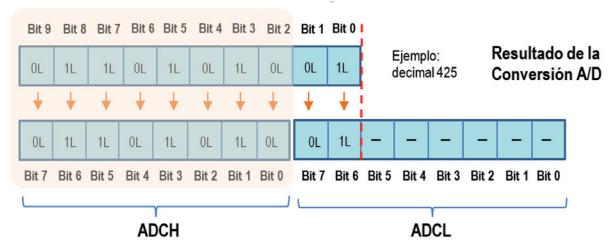
Resultado de una conversión con justificación a derecha



La justificación a izquierda (ADLAR = 1), por el contrario, implica que los 10 bits de la conversión comenzarán a almacenarse desde el bit más significativo del registro ADCH hasta el bit 6 del registro ADCL (figura 7.4).

**Figura 7.4**

Resultado de una conversión con justificación a izquierda



Esta configuración se utiliza cuando se requiere almacenar el valor de la conversión, pero solo a 8 bits, lo que implica eliminar los dos bits menos significativos; es decir, no se toma en cuenta el contenido del registro ADCL y únicamente se usa el valor almacenado en el ADCH. En el ejemplo planteado en las figuras 7.3 y 7.4, pese a que el valor de voltaje analógico convertido es el mismo en ambos casos, al rescatar su contenido, se obtiene 425 con 10 bits, mientras que con 8 bits el valor sería de 424. Al perder dos bits de resolución, se estaría adquiriendo un error de entre 0 y 3 en decimal, en cada conversión. Esta característica se utiliza,

sobre todo, cuando se trabaja en lenguajes de bajo nivel, como ensamblador, donde es más fácil manipular variables de 8 bits.

En el caso del IDE ATMEL STUDIO, este nos brinda la facilidad de obtener todo el resultado de la conversión sin tener que leer individualmente los registros ADCH y ADCL, como presentamos en los ejercicios propuestos.

### 7.3 Tiempos de conversión y señal de reloj

Cuando se realiza una conversión de una señal analógica, es necesario considerar el tiempo que esta puede tomar. Toda conversión pasa por dos etapas: de muestra/retención y de tiempo de conversión. La etapa de muestra y retención se utiliza para garantizar que el voltaje a ser convertido se encuentre en un nivel constante, mientras que el tiempo de conversión se emplea para asegurar que el circuito interno de aproximaciones sucesivas transforme el voltaje analógico retenido en un valor digital con una resolución de 10 bits (Atmel, 2014, p. 271-273).

Por otra parte, no todas las conversiones A/D poseen el mismo tiempo total de conversión (número de ciclos del reloj ADC). Esto se puede observar en la tabla 7.4. La figura 7.5 muestra un ejemplo de los ciclos de operación para una conversión de canal simple.

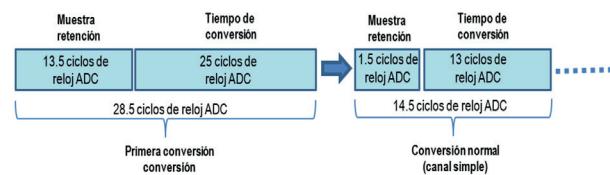
**Tabla 7.4**

Ciclos de muestra/retención y tiempo de conversión

Condición	Muestra/retención (ciclos desde el inicio de la conversión)	Tiempo de conversión (ciclos)
Primera conversión	13,5	25
Conversión normal (canal simple)	1,5	13
Conversiones auto disparadas	2	13.5
Conversión normal (canal diferencial)	1,5/2,5	13/14

**Figura 7.5**

Ejemplo de ciclos para una conversión AD



El módulo ADC posee una entrada de reloj propia, generada por medio de un *prescaler* aplicado a la frecuencia de reloj principal ( $f_{osc}$ ). Los valores de prescalamiento disponibles para el ATmega2560 son: 2,4,8,16,32,64,128.

El manual indica que es posible realizar hasta 15kSPS (kilo muestras por segundo) a máxima resolución (10 bits) (Atmel, 2014, p. 268). Este valor es de suma importancia, ya que implica que no necesariamente el uso de un cristal de frecuencia más alta junto con un *prescaler* bajo para la señal de reloj del ADC lograrán una conversión más rápida exitosa y con resolución de 10 bits. Por defecto, Atmel (2014) p.271 menciona que la frecuencia de reloj del módulo ADC para realizar una conversión a 10 bits debe estar entre los 50 KHz y los 200 KHz. Sin embargo, si son necesarias resoluciones más bajas, es posible superar ese valor hasta valores cercanos a los 1 000 KHz.

Es posible establecer una ecuación para escoger el *prescaler* más adecuado, en función de la frecuencia del  $f_{osc}$  y la frecuencia máxima de la señal de

$$\text{Preescalador} = \frac{f_{osc}}{\# \text{ciclos} * \text{Max conversiones por segundo}} \quad (7.3)$$

$$f_{reloj\ ADC} = \frac{f_{osc}}{\text{Preescalador}} \quad (7.4)$$

reloj ADC, tal como se muestra a continuación.

Ahora, se puede suponer que se ha colocado un cristal de 16 Mhz al microcontrolador si se conoce que es posible realizar un máximo de 15 000 conversiones por segundo y que el número total de ciclos de reloj ADC de una conversión normal es

de 14,5 ciclos (ver figura 7.5). Entonces, es posible calcular la frecuencia máxima del reloj del ADC y el *prescaler* mínimo correspondiente. Aplicando las ecuaciones 7.3 y 7.4 a los datos proporcionados, se obtiene:

$$\text{Preescalador} = \frac{16000000}{14.5 * 15000} = 73,56$$

$$\text{frecuencia máxima reloj ADC} = \frac{16000000}{73,56} = 217\ 509\ [\text{KHz}]$$

Sin embargo, debe notarse que no existe un valor de *prescaler* disponible para este requerimiento específico; por tanto, se escoge el valor de *prescaler* más cercano, por encima del calculado. En este caso, se optaría por 128. Con el *prescaler* corregido, se obtiene que la frecuencia de reloj del ADC sería de 125 [KHz]. Si bien se podría seleccionar el *prescaler* de 64, no es recomendable hacerlo, pues podría ocasionar errores en el proceso de conversión, debido a que los tiempos tanto en el muestreo como en la conversión se verían reducidos.

El *prescaler* puede configurarse en el registro ADCSRA, como el que consta en la figura 7.6, con base en las diferentes opciones detalladas en la tabla 7.5.

**Tabla 7.5**

Configuración de bits para seleccionar prescaler

ADPS2	ADPS1	ADPS0	Factor de división
0	0	0	2
0	0	1	2
0	1	0	4
0	1	1	8
1	0	0	16
1	0	1	32
1	1	0	64
1	1	1	128

Tomado de Atmel, 2014, p.285

**Figura 7.6**

Bits de configuración del registro ADCSRA para seleccionar prescaler

bit	7	6	5	4	3	2	1	0
	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0
Acción	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

Estado inicial 0 0 0 0 0 0 0 0 0

Tomado de Atmel, 2014, p. 285

## 7.4 Inicio y modos de conversión

Existen dos formas de realizar conversiones: única (manual) o por disparos (automática). La conversión única es aquella en la cual es necesario que cada vez que se necesite empezar una conversión se coloque el bit ADSC = 1. Una vez que la conversión haya iniciado, el bit ADSC se mantiene con el valor de 1, hasta que la conversión haya finalizado; es decir, se puede utilizar este bit para monitorear si una conversión ha llegado a su fin.

El método por disparos se activa colocando ADATE = 1 y escogiendo las fuentes que permiten realizar conversiones de manera automática (ver tabla 7.6). En el modo libre o *freerunning*, solo la primera conversión debe ser inicializada mediante ADSC = 1; mientras que todas las conversiones siguientes responden a la fuente de disparo ADIF. En los demás modos de la tabla 7.6, no es necesario inicializar la conversión con ADSC = 1; en este caso, el origen del inicio de cada conversión son las fuentes respectivas de disparo. Los bits de la tabla 7.6 deben ser configurados en el registro ADCSRB (ver figura 7.7). Si ADATE = 0, no importa qué configuración se coloque en los bits de disparo en el registro ADCSRB; el módulo ADC se encontrará en modo único o manual.

**Tabla 7.6***Configuración de bits para seleccionar las fuentes de disparo*

ADTS2	ADTS1	ADTS0	Fuente de disparo	Banderas que disparan la conversión
0	0	0	Modo libre ( <i>freerunning</i> )	ADIF
0	0	1	Comparador análogo	ACI
0	1	0	Requerimiento por interrupción externa 0	INTF0
0	1	1	Evento de comparación A en temporizador/contador 0	OCF0A
1	0	0	Evento de desborde en temporizador/contador 0	TOV0
1	0	1	Evento de comparación B en temporizador/contador 1	OCF1B
1	1	0	Evento de desborde en temporizador/contador 1	TOV1
1	1	1	Evento de captura en temporizador/contador 1	ICF1

Tomado de Atmel, 2014, p.287

**Figura 7.7***Bits de configuración del registro ADCSRB para seleccionar fuentes de disparo*

bit	7	6	5	4	3	2	1	0
Acción	-	ACME	-	-	MUX5	ADTS2	ADTS1	ADTS0
Estado inicial	R	R/W	R	R	R/W	R/W	R/W	R/W

Tomado de Atmel, 2014, p. 282

Sea cual fuese el método de conversión (manual o automático), el resultado de la conversión debe ser leído de los registros ADCH y ADCL; caso contrario, será borrado y reemplazado por el resultado de la siguiente conversión. Una manera eficiente de realizar la lectura de los registros es a través del uso de interrupciones, mediante la activación del bit ADIE = 1. Esto permite que cuando una conversión haya finalizado, se genere un evento y la bandera ADIF se coloque en 1. De esa manera, es posible acceder al ISR correspondiente al vector de interrupción 30 “ADC” (ver tabla 4.1) y almacenar en una variable el valor resultante de la conversión.

**Figura 7.8***Bits de configuración del registro ADCSRA para modos de conversión*

bit	7	6	5	4	3	2	1	0
Acción	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0
Estado inicial	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

Tomado de Atmel, 2014, p.285

Hay que tomar en cuenta, además, que debido a que existe un solo conversor y varios canales multiplexados, cuando se deseé cambiar de canal, es necesario haber terminado la conversión en curso, antes de reconfigurar el cambio de canal en el registro ADMUX y ADCSRB.

**Nota.** El funcionamiento de todo lo expuesto anteriormente depende de que el bit ADEN se configure con un valor de 1, sin importar si se encuentra en modo manual o automático.

## 7.5 Guía para configurar el módulo ADC

Debido a la cantidad de parámetros que se deben configurar en el conversor A/D, presentamos una pequeña guía de los lineamientos básicos que pueden considerarse antes y durante el proceso de configuración de registros.

- Es necesario establecer el rango de voltaje analógico que se va a tener como entrada, así como la resolución deseada. Esto permite fijar el voltaje de referencia más adecuado y la selección de una

- justificación a izquierda (8 bits) o derecha (10 bits) en el resultado de la conversión.
- Es necesario establecer si la conversión se efectúa en modo de canal diferencial o canal simple, seleccionar los canales y configurar los pines respectivos como entradas.
  - Se debe conocer la frecuencia del cristal principal del microcontrolador, con el fin de establecer el **prescaler** y no sobrepasar el número de muestras por segundo máximas permitidas por el módulo ADC.
  - Según la aplicación a usarse, tiene que establecerse si la conversión se realiza de manera única (manual) o por disparos (automática).
  - No olvidar habilitar el módulo de conversión A/D.
  - Si se utiliza interrupción de fin de conversión, no olvidar generar la rutina de servicio de interrupción ISR.

Por otra parte, cuando un PXn se encuentra trabajando como entrada analógica, se sugiere configurar el registro DIDR0 y DIDR2 (ver figuras 7.9 y 7.10), de tal manera que todos los bits asociados a las entradas analógicas utilizadas puedan ser colocados en un estado de 1. Esto posibilita reducir el consumo de energía en el *buffer* de entrada digital que no se estará utilizando, al mismo tiempo que permite desconectar el pin de la parte digital, para que no afecte las lecturas analógicas (Atmel, 2014, p. 288).

**Figura 7.9**

Bits de configuración del registro DIDR0 para deshabilitar buffer entrada digital (0-7)

bit	7	6	5	4	3	2	1	0
Acción	R/W							
Estado inicial	0	0	0	0	0	0	0	0

Tomado de Atmel, 2014, p. 287

**Figura 7.10**

Bits de configuración del registro DIDR2 para deshabilitar buffer entrada digital (8-15)

bit	7	6	5	4	3	2	1	0
Acción	R/W							
Estado inicial	0	0	0	0	0	0	0	0

Tomado de Atmel, 2014, p. 288

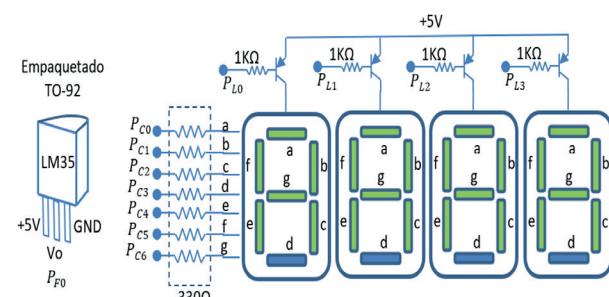
## 7.6 Ejercicios prácticos de configuración del conversor analógico digital

Con el fin de entender de mejor manera el funcionamiento y la configuración de los distintos registros que permiten la habilitación y el funcionamiento del conversor analógico digital (ADC), a continuación, presentamos una serie de casos en los que, manera progresiva, profundizamos los conceptos tratados en este capítulo.

**Ejercicio 7.1 (Conversión simple):** Se desea implementar una conversión simple sin interrupciones y presentar el resultado mediante un barrido de *display*. Se asume el uso de un sensor LM35 que posee una salida lineal de  $10\text{mV}^{\circ}\text{C}$ , y cuyo rango de medición es de  $-50^{\circ}\text{C}$  hasta  $150^{\circ}\text{C}$ . Además, se toma en cuenta que el cristal del microcontrolador es 16 MHz. La figura 7.11 muestra el esquema de conexiones.

**Figura 7.11**

Esquema de conexiones para visualizar una conversión A/D mediante barrido de display. Ejercicio 7.1



### Solución:

Para resolver este ejercicio, se considera que la salida del elemento LM35 está conectada al pin  $P_{F0}$ , el cual está asignado al canal ADC0 para conversio-

nes analógico-digitales. Además, se considera que no se van a medir temperaturas menores a 0 °C, ya que esto involucra voltajes menores a 0 V, lo cual no puede ser ingresado al microcontrolador. A partir de estas condiciones, se puede establecer que el posible rango de voltajes de salida del sensor es de 0 V a 1,5 V.

Con esta información, se puede determinar el voltaje de referencia más adecuado para el rango de voltaje. En este caso, se opta por una referencia interna  $V_{ref} = 2,56$  V. Además, se toma en cuenta que se desea una conversión a 10 bits de resolución (justificación a derecha). Con base en la tabla 7.2, se escoge la configuración necesaria para seleccionar el canal ADC0 (MUX 5:0), mientras que en la tabla 7.3 se selecciona el voltaje de referencia (REFS 1:0). A continuación, se presentan los bits necesarios para las configuraciones que cumplan con los requerimientos establecidos.

**Figura 7.12**

Configuración de registros ADMUX, ADCSRB. Ejercicio 7.1

bit	REFS 1:0			ADLAR		MUX 4:0					MUX 5		
ADMUX	1	1	0	0	0	0	0	0	0	0	0	0	0
Bit	7	6	5	4	MUX 5	3	2	1	0	x	x	x	x
ADCSR	x	x	x	x	0	x	x	x	x	x	x	x	x

Por otra parte, es necesario fijar el *prescaler* que puede usarse en la aplicación. Mediante la ecuación (7.3) se puede calcular el mínimo *prescaler*, en función del cristal principal. Con los datos actuales del ejercicio, se puede fijar el *prescaler* en 128. Los bits de configuración necesarios para escoger este valor pueden ser revisados en la tabla 7.5.

**Figura 7.13**

Configuración de registros ADCSRA. Ejercicio 7.1

bit	7	6	5	4	3	2	1	0	ADPS 2:0
ADCSRA	1	0	0	x	0	1	1	1	

**Nota.** Para la configuración, se tomó en cuenta que se requería una conversión simple (bit 5) ADATE=0, sin interrupciones (bit3) ADIE=0, y que es necesario habilitar el conversor (bit7) ADEN=1.

**Nota.** El (bit4) ADIF, debido a que no se está trabajando con interrupciones ni con eventos de disparo automático, puede tener cualquier valor; sin embargo, en caso de usar interrupciones, puede asignarse ADIE=1 para inicializar la conversión con la bandera de interrupción bajada.

El código de las configuraciones se escribiría de la siguiente manera:

```
ADMUX=0b11000000;
ADCSR&=~(1<<3);
ADCSRA=0b10000111;
```

**Nota.** Se está realizando una configuración inicial, por tanto (bit6) ADSC=0. Esto implica que el conversor A/D estará listo para comenzar una conversión, pero no se realizará hasta que en el lazo principal "main" se cambie al estado ADSC=1.

Ahora, para mostrar el valor de temperatura mediante el barrido de *displays*, se debe tener en cuenta la relación lineal del sensor 10mV/°C y la ecuación (7.1) para un  $V_{ref} = 2,56$  V, de la siguiente manera:

$$temp = \frac{ADC * V_{ref}}{1024 * (sensor_{resolución})}$$

Donde:

$$sensor_{resolución} = \frac{10mV}{^{\circ}C} = \frac{0.01V}{^{\circ}C}$$

(Texas Instruments Incorporated, 2017)

Finalmente, se debe considerar que para la conversión inicial, es necesario colocar el bit ADCS = 1, y para conocer si la conversión ha terminado se debe comprobar el estado de este bit. Cuando su estado cambie a 0L, estará indicando que la conversión ha finalizado y es posible recoger el valor de la conversión. El código que permite hacer el inicio de la conversión y testeo de bit es el siguiente:

```
ADCSRA|=1<<6;
while (ADCSRA&0b01000000);
```

No se deben olvidar las configuraciones de los registros para los pines de entrada y salida. El programa final muestra a continuación.<sup>10</sup>

```
#define F_CPU 16000000UL
#include <avr/io.h>
#include <util/delay.h>
unsigned int temp=0;
unsigned char centenas=0,decenas=0,unidades=0,miles=0;
unsigned char anodo_comun[16]={0xC0,0xF9,0xA4,0XB0,0x99,
0x92,0x82,0xF8,0x80,0x90,0X88,0X83,0XC6,0XA1,0XB6,0X8E};
void conver_digi(unsigned int val)
{
    miles=val/1000;
    centenas=(val%1000)/100;
    decenas=(val%1000)%100/10;
    unidades=(val%1000)%100%10;
}
void mostrar_disp()
{
    PORTI&=-(1<<3);
    PORTC=anodo_comun[unidades];
    _delay_ms(1);
    PORTL=255;
    PORTL&=-(1<<2);
    PORTC=anodo_comun[decenas];
    _delay_ms(1);
    PORTL=255;
    PORTL&=-(1<<1);
    PORTC=anodo_comun[centenas];
    _delay_ms(1);
    PORTL=255;
    PORTL&=-(1<<0);
    PORTC=anodo_comun[miles];
    _delay_ms(1);
    PORTL=255;
}
void config_puertos()
{
    DDRC=255; // definio salida de datos el pótico C para los displays 7 segmentos
    DDRL|=0x0f;// definio salidas para el control de los cuatro digitos del display, los 4 bits más bajos son los utilizados
    PORTC=0; // inicializo con salidas en cero
    PORTL&=0x0f; // inicializo con salidas de control en cero
    DDRF&=-(1<<0); // declaro como entrada el pin PF0
}
void config_AD()
{
```

```
{           ADMUX=0b11000000;
            ADCSRB&=-(1<<3);
            ADCSRA=0b10000111;
}

int main(void)
{
    config_puertos();
    config_AD();
    while (1)
    {
        ADCSRA|=1<<6;
        while (ADCSRA&0b01000000);
        temp=(ADC*2.56)/(1024*0.01);
        conver_digi(temp);
        mostrar_disp();
    }
}
```

Fijarse que en cada iteración se coloca ADSC=1. Esto, debido a que las configuraciones están para una conversión simple y esta debe ser inicializada cada vez que se desee una nueva conversión.

**Nota.** Fijarse que el código generado permite obtener valores de temperatura, pero en formato de números enteros, no decimales. En caso de requerir decimales, se puede usar el método de "Fixed Point Number" y utilizar un punto en el barrido de *displays*.

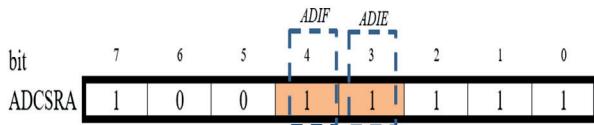
**Nota.** Recordar que la ADC es una variable reservada en el IDE, que contiene ADCH y ADCL.

**Ejercicio 7.2 (uso de interrupción):** Con base en el ejercicio anterior, ahora se analiza el caso de leer el dato convertido, pero mediante el uso de interrupción. Las funciones y variables declaradas son las mismas que en el ejercicio anterior. Se debe analizar la nueva configuración en el ADCSRA con ADIE = 1 y la rutina de servicio de interrupción ISR asociada al ADC. También se sugiere confirmar que la bandera de interrupción esté abajo. Esto se realiza forzando la escritura de un 1L en el bit correspondiente. La nueva configuración del registro ADCSRA puede observarse en la figura 7.1.4.

<sup>10</sup> La explicación del código de barrido de *display* puede ser revisada en el capítulo 3, sobre puertos de entrada y salida.

**Figura 7.14**

Configuración de registros ADCSRA. Ejercicio 7.2



### Solución:

Las configuraciones quedarían de la siguiente manera:

```
ADMUX=0b11000000;
ADCSRB&=~(1<<3);
ADCSRA=0b10011111;
```

Además, se debe incluir la función de servicio de interrupción para la lectura del dato convertido:

```
ISR(ADC_vect)
{
    temp=(ADC*2.56)/(1024*0.01)
    ADCSRA|=1<<6;
}
```

Hay que considerar que las configuraciones indican conversión simple; es decir, manualmente se deben arrancar, de manera consecutiva, las conversiones. Además, la librería de interrupciones debe estar incluida y se deben habilitar las interrupciones del microcontrolador con la instrucción sei().

El código final con los cambios propuestos sería el siguiente:

```
#define F_CPU 16000000UL
#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>
unsigned int temp=0;
unsigned char centenas=0,decenas=0,unidades=0,miles=0;
unsigned char anodo_comun[16]={0xC0,0xF9,0xA4,0XB0,0x99,
0x92,0x82,0xF8,0x80,0x90,0X88,0X83,0XC6,0XA1,0X86,0X8E};
void conver_digi(unsigned int val)
{
    miles=val/1000;
    centenas=(val%1000)/100;
    decenas=((val%1000)%100)/10;
    unidades=((val%1000)%100)%10;
}
void mostrar_disp()
{
    PORTL&=~(1<<3);
    PORTC=anodo_comun[unidades];
    _delay_ms(1);
    PORTL=255;
    PORTL&=~(1<<2);
    PORTC=anodo_comun[decenas];
    _delay_ms(1);
    PORTL=255;
    PORTL&=~(1<<1);
    PORTC=anodo_comun[centenas];
    _delay_ms(1);
    PORTL=255;
    PORTL&=~(1<<0);
    PORTC=anodo_comun[miles];
    _delay_ms(1);
    PORTL=255;
}
void config_puertos()
{
    DDRC=255; // defino salida de datos el pótico C para los displays 7 segmentos
    DDRL|=0x0f;// defino salidas para el control de los cuatro dígitos del display, los 4 bits más bajos son los utilizados
    PORTC=0;// inicializo con salidas en cero
    PORTL&=0xf0;// inicializo con salidas de control en cero
    DDRF&=~(1<<0);// declaro como entrada el pin PF0
}
```

```

void config_AD()
{
    ADMUX=0b11000000;
    ADCSRB&=~(1<<3);
    ADCSRA=0b10011111;
}

ISR(ADC_vect)
{
    temp=(ADC*2.56)/(1024*0.01);
    ADCSRA|=1<<6;// después de cada conversión se debe iniciar nuevamente
}

int main(void)
{
    config_puertos();
    config_AD();
    sei();
    ADCSRA|=1<<6;//se debe iniciar la conversión
    while (1)
    {
        conver_digi(temp);
        mostrar_disp();
    }
}

```

Es importante advertir que el programa principal “main” y, específicamente, el lazo infinito “while” ya no requieren de una espera para obtener el dato resultante de la conversión. En cambio, el dato se actualiza automáticamente cada vez que la conversión termine.

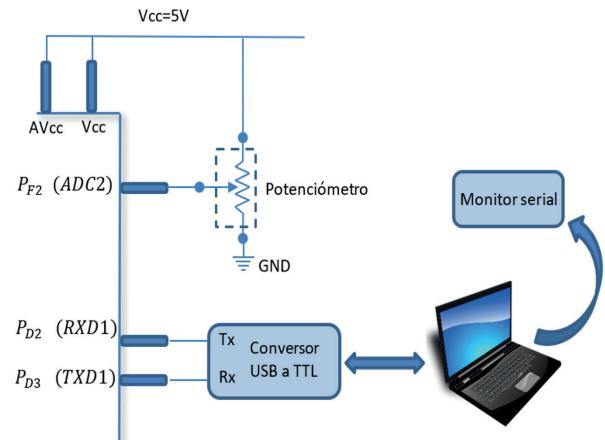
Este es un concepto muy parecido al manejado por la configuración por eventos de disparo; en específico, el configurado para *freerunning*, en el que cada conversión inicia una vez que la anterior haya finalizado.

**Ejercicio 7.3 (*freerunning*):** Se propone medir la variación de voltaje mediante el uso de un potenciómetro (figura 7.15) y enviar el valor medido a una terminal serial a través de comunicación UART. Se debe considerar un cristal de 16 MHz y

una resolución de 10 bits; además, el envío de los datos debe realizarse de manera automática, por medio de eventos de disparo en modo *freerunning* (ver figura 7.15).

**Figura 7.15**

Esquema de conexiones para visualizar una conversión A/D mediante monitor serial. Ejercicio 7.3

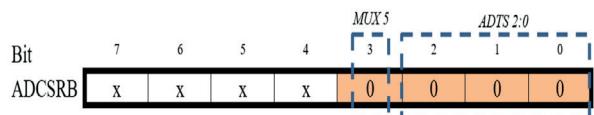
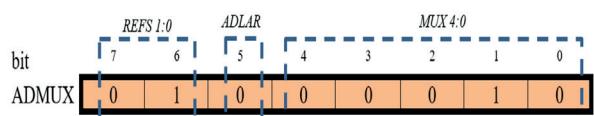


### Solución:

Como se puede observar en la figura 7.15, el potenciómetro se encuentra conectado a 5 V. Esto implica que la configuración del conversor AD debe ser capaz de realizar conversiones en el rango de los 0 a los 5 V. Para esto, se debe configurar el  $V_{ref} = AVcc$  y se debe tomar en cuenta que el ingreso de la señal analógica se realiza mediante el canal ADC2. Las configuraciones de los registros se muestran en la figura 7.16.

**Figura 7.16**

Configuración de registros ADMUX, ADCSRB. Ejercicio 7.3



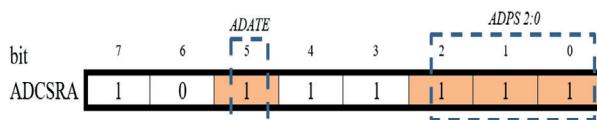
El modo *freerunning* se configura en el registro ADCSRB, con base en la tabla 7.6.

Para que esta configuración tenga efecto, se debe colocar ADATE = 1 (bit5) en el registro ADCSRA. Con los datos actuales del ejercicio, se puede fijar el *prescaler* en 128. Los bits de configuración necesarios para escoger este valor pueden ser revisados en la tabla 5. Por otra parte, con el fin de poder leer el valor de cada conversión y que la bandera de interrupción ADIF (bit4) baje de manera automática a 0L, se usa la interrupción por finalización de conversión ADIE = 1L (bit3).

Hay que recordar, asimismo, que es recomendable inicializar las configuraciones bajando la bandera de interrupción, ADIF = 1.

**Figura 7.17**

Configuración de registros ADCSRA. Ejercicio 7.3



La configuración propuesta de los registros es la siguiente:

```
ADMUX=0b01000010;//referencia AVcc, 10 bits, canal ADC2
ADCSRB=0b00000000;//canal ADC2, freerunning
ADCSRA=0b10111111;//Habilitación ADC, habilitar interrupción,
//bajar bandera, activar eventos de disparo, prescaler 128
```

Para este ejercicio, se ha escogido una comunicación UART, transmisión habilitada, deshabilitada la recepción, con 8 bits de datos, 9600 baudios, 1 bit de parada y ningún bit de paridad, sin interrupciones.<sup>11</sup>

Configuración de registros para comunicación UART:

```
UCSRB=0b00001000; // sin interrupciones, habilitación de transmisión,
//transmisión a 8 bits de datos
UCSRC=0b00000110; // configuración asíncrona, sin paridad, 1bit de parada
// transmisión a 8 bits de datos
UBRRL=103; //9600 bps
```

El código final puede implementarse de la siguiente manera:

```
#define F_CPU 16000000UL
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>

float Vin=0;
char str[7];
void config_puertos()
{
    DDRF&=~(1<<2); //configuración de entrada para PF2 (ADC2)
    DDRD&=~(1<<2); //configuración de entrada para PD2 (RXD1)
    DDRD|=1<<3; //configuración de salida para PD3 (TXD1)
}
void config_ADC()
{
    ADMUX=0b01000010;//referencia AVcc, 10 bits, canal ADC2
    ADCSRB=0b00000000;//canal ADC2, freerunning
    ADCSRA=0b10111111;//Habilitación ADC, habilitar interrupción,
    //bajar bandera, activar eventos de disparo,
    prescaler 128
}
void config_UART1()
{
    UCSRB=0b00001000; // sin interrupciones, habilitación de transmisión,
    //transmisión a 8 bits de datos
    UCSRC=0b00000110; // configuración asíncrona, sin paridad, 1bit de parada
    // transmisión a 8 bits de datos
    UBRRL=103; //9600 bps
}
void enviar_dato(float val)
{
    sprintf(str,"%f\r\n",val);
    for (unsigned char i=0;i<=6;i++)
    {
        UDR1=str[i];
        while ((UCSR1A&0b01000000)==0);//espera hasta que el dato se haya
        enviado
        UCSR1A|=0b01000000; //baja la bandera de transmisión
    }
}
```

<sup>11</sup> Para mayor referencia sobre la configuración de los registros, se puede revisar el capítulo 5.

```

ISR(ADC_vect)
{
    Vin=(ADC*5.0)/1024.0;
}

int main(void)
{
    config_puertos();
    config_UART1();
    config_ADC();
    ADCSRA|=1<<6;//se debe iniciar la primera conversión
    sei();
    while (1)
    {
        enviar_dato(Vin);
    }
}

```

**Nota.** Recordar que el cálculo de Vin, en el código propuesto, proviene de la ecuación (7.1) con un  $V_{ref}=5\text{ V}$ .

#### Ejercicio 7.4 (Timer/Counter1 Compare Match B):

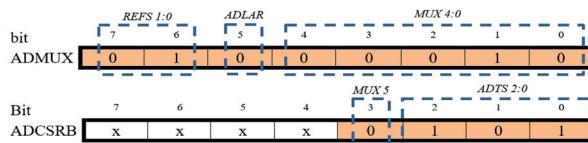
Se desea implementar un programa que permita realizar conversiones analógicas de manera temporizada y automática, y enviar el dato de voltaje convertido a un terminal serial cada segundo. Se plantea el mismo escenario de la figura 7.15, en donde el voltaje variable proviene de un potenciómetro alimentado por 5 V.

#### Solución:

Para resolver este ejercicio, se considera que la temporización proviene del temporizador/contador1 y que se utilizará el evento de comparación B, para iniciar una conversión analógica. Para esto, es necesario habilitar en el conversor A/D los eventos por disparos y configurar el evento mencionado ADTS 2:0, como se detalla en la tabla 7.6. El canal para configurar es el ADC2 y el  $V_{ref}=AVcc=5\text{ V}$ . El resultado de la conversión se obtendrá con justificación derecha (ver figura 7.18).

Figura 7.18

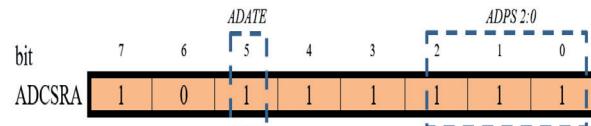
Configuración de registros ADMUX, ADCSRB. Ejercicio 7.4



En el registro ADCSRA de la figura 7.19 se tiene en cuenta la habilitación del conversor, la activación de interrupción por finalización de conversión, el *prescaler* de 128 y la activación de eventos por disparo (*trigger source*); se forzará la bandera a cero.

Figura 7.19

Configuración de registros ADCSRA. Ejercicio 7.4



La configuración de los registros para inicializar el conversor analógico son los siguientes:

ADMUX=0b01000010;//referencia AVcc, 10 bits, canal ADC2

ADCSRB=0b00000101;//canal ADC2, Timer/Counter1 Compare Match B

ADCSRA=0b10111111;//Habilitación ADC, con interrupción,

//bajar bandera, activar eventos de disparo, prescaler 128

**Nota.** La configuración de registros y las características de la comunicación UART son las mismas que las planteadas en el ejercicio 7.3.

Ahora, una parte importante es configurar el temporizador/contador1, en la que es necesario establecer una interrupción por evento de comparación B. Para ello, se configura el temporizador en modo normal, con el máximo *prescaler* (1024), y se plantea, como ejemplo práctico, que la comparación se realice cada 1 segundo. Al realizarse esta comparación, su bandera asociada OCF1B en el registro TIFR1 cambiará de estado de 0L a 1L, lo que provocará que la conversión analógica inicie. Con el fin de que el algoritmo sea cíclico, es necesario bajar la bandera de comparación y encerar el TCNT1. Esto se hace con la ayuda de la interrupción asociada a la bandera OCF1B<sup>12</sup>. El programa implementado es el siguiente:

<sup>12</sup> El detalle de comportamiento del temporizador/ contador1, sus modos de operación y la configuración de registros pueden ser consultados en el capítulo 6.

```

#define F_CPU 16000000UL
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>
float Vin=0;
char str[7];
unsigned char enviar=0;
void config_puertos()
{
    DDRF&=~(1<<2); //configuración de entrada para PF2 (ADC2)
    DDRD&=~(1<<2); //configuración de entrada para PD2 (RXD1)
    DDRD|=1<<3; //configuración de salida para PD3 (TXD1)
}
void config_ADC()
{
    ADMUX=0b01000010; //referencia AVcc, 10 bits, canal ADC2
    ADCSRB=0b00000101; //canal ADC2, Timer/Counter1 Compare Match B
    ADCSRA=0b10111111; //Habilitación ADC, con interrupción,
    //bajar bandera, activar eventos de disparo, prescaler 128
}
void config_UART1()
{
    UCSR1B=0b00000100; // sin interrupciones, habilitación de transmisión,
    //transmisión a 8 bits de datos
    UCSR1C=0b00000110; // configuración asíncrona, sin paridad, 1bit de parada
    // transmisión a 8 bits de dato
    UBRR1=103; //9600 bps
}
void config_timer1()
{
    TCCR1A=0b00000000; // sin generación de ondas, modo normal (no pwm)
    TCNT1=0;
    TCCR1B=0b00000101; //modo normal, prescaler 1024
    OCR1B=15624; //límite de comparación B
    TIMSK1|=1<<2;//interrupción por comparación B
    TIFR1|=1<<2;// bajar bandera de interrupción de comparación B
}
void enviar_dato(float val)
{
    sprintf(str,"%6.3f\r\n",val);
    for (unsigned char i=0;i<=6;i++)
    {
        UDR1=str[i];
        while ((UCSR1A&0b01000000)==0); //espera hasta que el dato se haya
        //enviado
        UCSR1A|=0b01000000; //baja la bandera de transmisión
    }
}
ISR(ADC_vect)
{
    enviar=1;
}
ISR(TIMER1_COMPB_vect)
{
    TCNT1=0;
}
int main(void)
{
    config_puertos();
    config_UART1();
    config_timer1();
    config_ADC();
    sei();
    while (1)
    {
        PORTK=PORTK;
        if (enviar==1)
        {
            Vin=(ADC*5.0)/1024.0;
            enviar_dato(Vin);
            enviar=0;
        }
    }
}

```

Se debe tener en consideración que la variable “enviar” es usada para realizar el envío si y solo si la conversión ha finalizado.

Por otra parte, la interrupción de comparación en el temporizador/contador1 es necesaria, ya que permite bajar la bandera de interrupción y, al mismo tiempo, encera el contador para iniciar un nuevo ciclo de tiempo.

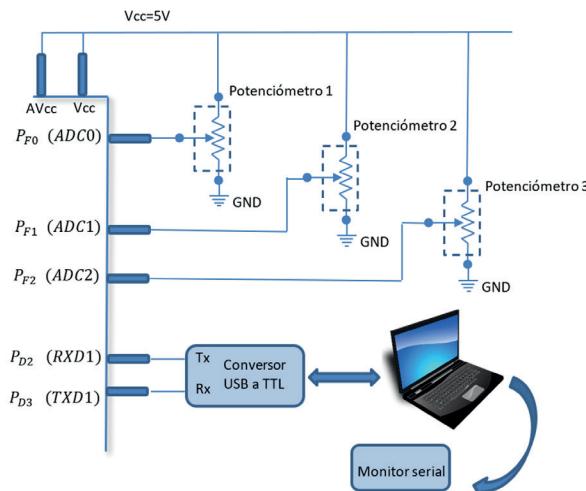
**Nota.** El prescaler en el temporizador/contador1 se establece en 1024, para que los tiempos de comparación puedan ser modificados fácilmente, cambiando el valor del registro OCR1B y alcanzando tiempos de hasta 4,19 [s], con un valor de 65535.

**Nota.** Si bien sería posible colocar toda la rutina de envío en la interrupción de finalización de conversión, no es recomendable, ya que se debe tener en cuenta que, como buena práctica de programación, las ISR deben ser de ejecución rápida.

**Ejercicio 7.5 (Conversión de varios canales simples):** Se propone realizar la conversión de 3 entradas analógicas conectadas a los canales ADC0, ADC1 y ADC2, y enviar el valor de voltaje de la conversión de cada uno de los potenciómetros a un monitor serial. Las conversiones se van a realizar sin interrupción. El esquema de conexiones se muestra en la figura 7.20.

Figura 7.20.

Esquema de conexiones para visualizar varias conversiones A/D mediante monitor serial. Ejercicio 7.5



### Solución:

Al igual que en los ejercicios anteriores, los primeros pasos son establecer las configuraciones necesarias. En este caso, se puede inferir que el  $V_{ref} = AVcc = 5V$ . Para obtener 10 bits de resolución, es necesaria una configuración de resultado a derecha y se debe establecer un *prescaler* de 128. Como no se utilizarán eventos de disparo ADATE = 0L, al igual que los bits ADTS 2:0, en el registro ADCSRB se ha decidido inicializar la configuración para el canal cero. En la figura 7.21, se puede observar la configuración de bits en los registros correspondientes al módulo ADC.

Figura 7.21

Configuración de registros ADMUX, ADCSRB. Ejercicio 7.5

bit	REFS 1:0		ADLAR		MUX 4:0			
ADMUX	0	1	0	0	0	0	0	0
Bit	MUX 5		ADTS 2:0					
ADCSR B	X	X	X	X	0	0	0	0

En el registro ADCSRA (ver figura 7.22), en cambio, hay que tomar en cuenta la habilitación del convertor y el *prescaler* de 128.

Figura 7.22

Configuración de registros ADCSRA. Ejercicio 7.5

bit	ADEN		ADPS 2:0				
ADCSRA	1	0	0	0	0	1	1

La configuración de los registros sería la siguiente:

ADMX=0b01000000; //referencia AVcc, 10 bits, inicializa canal ADC0

ADCSR B=0b00000000; //canal ADC0

ADCSRA=0b10000111; //Habilitación ADC, sin interrupción,

//sin eventos de disparo, prescaler 128

**Nota.** La configuración de la comunicación UART es la misma que en los ejercicios anteriores.

La configuración de puertos ahora tiene una ligera variación, ya que se deben configurar 3 entradas, además de los pines de comunicación UART.

DDRF&=0b11111000; //configuración de entrada para PF2,PF1,PF0

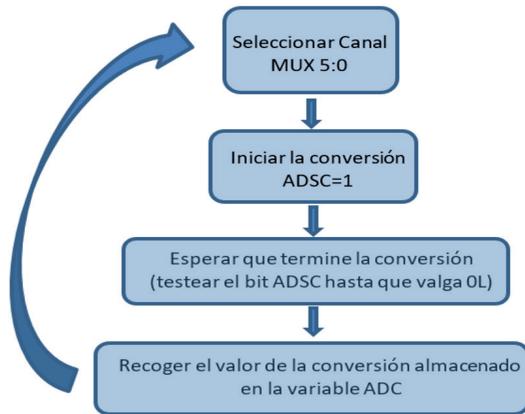
DDRD&=(1<<2); //configuración de entrada para PD2 (RXD1)

DDRD|=1<<3); //configuración de salida para PD3 (TXD1)

Hay que recalcar que el microcontrolador cuenta solo con un conversor analógico digital y posee un multiplexor que permite que los 16 canales puedan ser seleccionados individualmente, para el proceso de conversión. En este contexto, si se desea realizar conversiones en varios canales, se debe hacer de manera individual y secuencial. El procedimiento se describe en la figura 7.23.

**Figura 7.23**

Esquema para la conversión de varios canales. Ejercicio 7.5



A continuación, se presenta la implementación del código:

```

#define F_CPU 16000000UL
#include <avr/io.h>
#include <util/delay.h>

float Vin=0;
char str[7];
char sensor1[]{"sensor 1:"};
char sensor2[]{"sensor 2:"};
char sensor3[]{"sensor 3:"};

void config_puertos()
{
    DDRF&=0b11111000; //configuración de entrada para PF2,PF1,PF0.
    DDRD&=~(1<<2); //configuración de entrada para PD2 (RXD1)
    DDRD|=1<<3; //configuración de salida para PD3 (TXD1)
}

void config_ADC()
{
    ADMUX=0b01000000;//referencia AVcc, 10 bits, inicializa canal ADC0
    ADCSRB=0b00000000;//canal ADC0
    ADCSRA=0b10000111;//Habilitación ADC, sin interrupción,
    //sin eventos de disparo, prescaler 128
}

void config_UART1()
{
    UCSR1B=0b00001000; // sin interrupciones, habilitación de transmisión,
    //transmisión a 8 bits de datos
    UCSR1C=0b00000110; // configuración asíncrona, sin paridad, 1bit de parada
    // transmisión a 8 bits de datos
    UBRR1=103; //9600 bps
}

void enviar_dato(float val)
{
    sprintf(str,"%0.3f\r\n",val);
    for (unsigned char i=0;i<=6;j++)
    {
        UDR1=str[i];
        while ((UCSR1A&0b01000000)==0);//espera hasta que el dato se haya
        enviado
        UCSR1A|=0b01000000; //baja la bandera de transmisión
    }
}

void enviar_texto(unsigned char val[])
{
    for (unsigned char i=0;i<=9;j++)
    {
        UDR1=val[i];
        while ((UCSR1A&0b01000000)==0);//espera hasta que el dato se haya
        enviado
        UCSR1A|=0b01000000; //baja la bandera de transmisión
    }
}
  
```

```

int main(void)
{
    config_puertos();
    config_UART1();
    config_ADC();
    while (1)
    {
        ADMUX=0b01000000;// inicializa canal ADC1
        ADCSRA|=1<<6; //empieza conversión ADC0
        while (ADCSRA&0b01000000);
        Vin=(ADC*5.0)/1024.0;
        enviar_texto(sensor1);
        enviar_dato(Vin);
        //*****
        ADMUX=0b01000001;// inicializa canal ADC1
        ADCSRA|=1<<6; //empieza conversión ADC1
        while (ADCSRA&0b01000000);
        Vin=(ADC*5.0)/1024.0;
        enviar_texto(sensor2);
        enviar_dato(Vin);
        /////
        ADMUX=0b01000010;// inicializa canal ADC2
        ADCSRA|=1<<6; //empieza conversión ADC2
        while (ADCSRA&0b01000000);
        Vin=(ADC*5.0)/1024.0;
        enviar_texto(sensor3);
        enviar_dato(Vin);
    }
}

```

**Nota.** No olvidar configurar ADMUX en cada selección de canal.

# Glosario de términos

## Bauds.

Hace referencia a la velocidad con la que se transmiten los datos en un canal de comunicación.

## Bits.

Forma parte del sistema binario, considerada como una señal electrónica que cuenta con dos estados (1 o 0, encendido o apagado).

## Bucle.

Hace referencia a un estado de flujo de control que se ejecuta repetidamente con base en una condición booleana dada.

## E/S.

Siglas asociadas a los pines en comportamiento como entrada o como salida.

## Encabezados ICSP.

Hace referencia a la programación de chips que permiten que el microcontrolador reciba el *firmware* o programa que realiza todas las funcionalidades avanzadas que se desean.

## F\_CPU.

Nombre utilizado en el entorno de Atmel Studio referido a la frecuencia del cristal que se está utilizando en el microcontrolador.

## For.

Hace referencia a la sentencia “Para” en programación.

## GND.

Está asociado con “Tierra”, punto de referencia para las señales desde donde se pueden medir todos los voltajes.

## IOREF.

Está asociado al pin de referencia de E/S.

## IoT.

Siglas asociadas a Internet de las cosas (Internet of things), cuyos dispositivos informáticos se encuentran interrelacionados.

## L/E.

Siglas asociadas a los bits de los registros indicando si pueden ser de lectura o escritura.

## LoRa.

Término asociado a la tecnología inalámbrica de largo alcance. Es una técnica de modulación de espectro ensanchado.

## Microchip PIC.

Nombre asociado a un circuito integrado programable.

## Nibble.

Término asociado a un grupo de 4 bits. Puede referirse a *nibble* bajo o *nibble* alto en función de si se agrupan los 4 primeros o bits o los 4 últimos bits en un registro de 8 bits.

**Nivel lógico.**

Estado de voltaje asociado, por lo general, a un PXn.  
Puede ser HIGH o LOW.

**Pines DIP.**

Nombre asociado a la encapsulación de pines en hileras paralelas para poder trabajar en circuitos integrados. Es un circuito integrado.

**PXn.**

Siglas utilizadas para representar un pin cualquiera “n” de un pórtico cualquiera “X”

**RF.**

Término asociado a una señal de radiofrecuencia utilizada como forma de comunicación, ya que esta describe a una señal electromagnética inalámbrica.

**Rx.**

Término que hace referencia al pin de recepción de datos de un microcontrolador.

**TOOGLE.**

Término relacionado con el cambio de estado de un pin respecto de su estado anterior.

**Tx.**

Término que hace referencia al pin de transmisión de datos de un microcontrolador.

**USB-TTL.**

Término que hace referencia a la gama de cables convertidores USB a serie, los cuales proporcionan conectividad entre interfaces USB y UART.

**VIN.**

Término que hace referencia al voltaje de entrada de una placa al momento en el que se está utilizando una fuente de alimentación externa.

**While.**

Término que hace referencia a la sentencia de “Mientras” en programación.

## **Trayectorias profesionales**

Esta es una obra que surge de la colaboración intensa de tres autores.



**Ing. David  
Fernando Pozo  
Espín, MSc.**

participado en concursos de divulgación científica, innovación y robótica a nivel nacional e internacional. Actualmente, se encuentra trabajando en proyectos de investigación y ha producido varias publicaciones científicas en las áreas de robótica, visión por computador, control y *machine learning*.

*Docente Carrera de Ingeniería en Electrónica y Automatización*

*Universidad de Las Américas*

David Pozo inició su formación profesional en la Escuela Politécnica Nacional (EPN) en el área de Ingeniería en Electrónica y Control. Una vez culminados sus estudios, se desempeñó como asistente de cátedra en el Departamento de Automatización y Control Industrial, en la Facultad de Ingeniería Eléctrica y Electrónica de la EPN. Con el fin de continuar su preparación académica, realizó estudios de posgrado en la Universidad Politécnica de Catalunya, Barcelona-España, gracias a una beca otorgada por la SENESCYT (Ecuador), y obtuvo el grado de Máster en Automática y Robótica (2012). A su regreso, y con ánimos de aplicar sus conocimientos, retornó a la EPN como docente ocasional a tiempo completo. Siguiendo con su carrera profesional, se vinculó a la Universidad de Las Américas, donde desempeña el cargo de docente a tiempo completo. Ha formado parte del equipo que creó la Unidad de Innovación Tecnológica (UITEC) y ha



**Ing. Jorge Luis  
Rosero Beltrán,  
MSc.**

*Coordinador Carrera de Ingeniería en Electrónica y Automatización*  
*Universidad de Las Américas*

Jorge Luis Rosero se graduó como Ingeniero en Electrónica y Control, en la Escuela Politécnica Nacional, en el año 2008. Tiene experiencia en la implementación de sistemas automatizados de generación termoeléctrica, en campos petroleros y embarcaciones atuneras. Tiene una Maestría en Ciencias con Especialidad en Automatización, en el Tecnológico de Monterrey, con mención honorífica de excelencia académica. Realizó una estancia de investigación en la Universidad Politécnica de Catalunya, en Barcelona, España, en el año 2012. Ambos logros académicos, a través de becas de los gobiernos ecuatoriano y mexicano. Se desempeñó como docente a tiempo completo en la Escuela Politécnica Nacional y, actualmente, es coordinador académico de la carrera de Electrónica y Automatización, de cuya creación fue parte, en la Universidad de Las Américas. Ha publicado artículos científicos en áreas de robótica, energía, internet de las cosas y comunicaciones, en varios congresos y revistas con indexación LatinIndex, IEEE y SCOPUS.



**Ing. Santiago  
Leonardo  
Solórzano, MSc.**

*Consultor técnico en proyectos de investigación*

Santiago Leonardo Solórzano Lescano es Máster en Inteligencia Artificial. Cuenta con una ingeniería en Redes y Telecomunicaciones y una tecnología en Electrónica Instrumentación & Aviónica. Actualmente (2020), es consultor para el PNUD (Proyecto de las Naciones Unidas Para el Desarrollo) y docente a tiempo parcial de la Universidad de las Américas, UDLA. Su formación le ha permitido manejar y desarrollar proyectos con tecnología de punta e impulsar soluciones de múltiples ingenierías. Parte de su experiencia profesional se centra en la Unidad de Innovación Tecnológica, UITEC, de la UDLA, en donde ha participado en concursos nacionales e internacionales como: Laureate Award for Excellence in Robotic Engineering, organizado por la Universidad Europea de Madrid, con el proyecto SAMAYBOT. En dicho certamen logró el tercer lugar a nivel mundial. FINAL CONTINENTAL Proyecto Multimedia INFORMATRÍX impulsado por la Universidad Autónoma de Guadalajara, en el cual obtuvo la medalla de oro y el galardón al mejor proyecto. Ha recibido varios reconocimientos en diferentes categorías de robótica (robots seguidores de línea, drones, robots humanoides, categoría innovación). Cuenta con varias publicaciones científicas y una patente por invención de un biorreactor de inmersión temporal.

# Índice de figuras

11	<b>Figura 1.1.</b>	Formas de representación de números con signo	23	<b>Figura 2.1.</b>	Arquitectura chip Intel 4004
14	<b>Figura 1.2.</b>	Símbolo lógico y tabla de verdad de la compuerta NOT	24	<b>Figura 2.2.</b>	Esquema básico de un microcontrolador
14	<b>Figura 1.3.</b>	Operación de la compuerta lógica NOT	26	<b>Figura 2.3.</b>	Gamas de microcontroladores tipo PIC
14	<b>Figura 1.4.</b>	Símbolo lógico y tabla de verdad de la compuerta AND	26	<b>Figura 2.4.</b>	Gama microcontroladores AVR
15	<b>Figura 1.5.</b>	Operación de la compuerta lógica AND.	26	<b>Figura 2.5.</b>	Aplicación microcontroladores
15	<b>Figura 1.6.</b>	Símbolo lógico y tabla de verdad de la compuerta OR	36	<b>Figura 2.6.</b>	Esquema de conexión led parpadeante
15	<b>Figura 1.7.</b>	Operación de la compuerta lógica OR	39	<b>Figura 3.1.</b>	Niveles de tensión para E/S
16	<b>Figura 1.8.</b>	Símbolo lógico y tabla de verdad de la compuerta NAND	40	<b>Figura 3.2.</b>	Asignación de bits para un registro DDRx
16	<b>Figura 1.9.</b>	Operación de la compuerta lógica NAND.	40	<b>Figura 3.3.</b>	Asignación de bits para un registro PORTx
16	<b>Figura 1.10.</b>	Símbolo lógico y tabla de verdad de la compuerta NOR	41	<b>Figura 3.4.</b>	Registro de control
16	<b>Figura 1.11.</b>	Operación de la compuerta lógica NOR	41	<b>Figura 3.5.</b>	Configuraciones de <i>pull-up</i> y estados lógicos asociados
17	<b>Figura 1.12.</b>	Símbolo lógico y tabla de verdad de la compuerta XOR	42	<b>Figura 3.6.</b>	Asignación de bits para un registro PINx
17	<b>Figura 1.13.</b>	Operación de la compuerta lógica XOR	42	<b>Figura 3.7.</b>	Configuración de registros. Ejercicio 3.1
18	<b>Figura 1.14.</b>	Transformación binaria a Gray	42	<b>Figura 3.8.</b>	Esquema de conexión y estado de salidas. Ejercicio 3.1
18	<b>Figura 1.15.</b>	Transformación Gray a binario	43	<b>Figura 3.9.</b>	Configuración de registros. Ejercicio 3.2
			43	<b>Figura 3.10.</b>	Esquema de conexión y estados entrada. Ejercicio 3.2

43	<b>Figura 3.11.</b>	Representación de la información en registro PINx. Ejercicio 3.2	59	<b>Figura 4.4.</b>	Bits de configuración del registro EICRB - EICRA
44	<b>Figura 3.12.</b>	Configuración de registros. Ejercicio 3.3	59	<b>Figura 4.5.</b>	Bits de configuración en el registro EIMSK
44	<b>Figura 3.13.</b>	Esquema de conexión y estados entrada. Ejercicio 3.3	59	<b>Figura 4.6.</b>	Bit 7 encargado de la habilitación global de interrupciones en el registro SREG
45	<b>Figura 3.14.</b>	Enmascaramiento tipo AND para modificaciones de bit a 0L en el registro DDRA. Ejercicio 3.4	59	<b>Figura 4.7.</b>	Banderas de INT en el registro EIFR
45	<b>Figura 3.15.</b>	Enmascaramiento tipo AND para rescate de información en bits específicos. Ejercicio 3.5	60	<b>Figura 4.8.</b>	Interpretación de cambios de estado en el tiempo de un grupo de bits
46	<b>Figura 3.16.</b>	Enmascaramiento tipo OR para modificaciones de bit a 1L en el registro DDRAEjercicio 3.6	60	<b>Figura 4.9.</b>	Registros de habilitación para interrupciones tipo PCINT
46	<b>Figura 3.17.</b>	Esquema de conexiones para encendido y apagado de ledes ( <i>blinking</i> ). Ejercicio 3.7	60	<b>Figura 4.10.</b>	Representación de grupos de pines con función PCINT asociados a los vectores de interrupción.
46	<b>Figura 3.18.</b>	Enmascaramiento tipo XOR para operaciones bit a bit. Ejercicio 3.7.	61	<b>Figura 4.11.</b>	Registro PCICR y bits asociados con la habilitación de los grupos para los vectores de interrupción PCINT0, PCINT1 y PCIT2
47	<b>Figura 3.19.</b>	Esquema de conexiones con dip-switch y ledes. Ejercicio 3.8	61	<b>Figura 4.12.</b>	Registro PCMSK0 y bits para la habilitación individual de pines por cambio de estado.
48	<b>Figura 3.20.</b>	Esquema de conexiones para luces inteligentes. Ejercicio 3.9	61	<b>Figura 4.13.</b>	Registro PCMSK1 y bits para la habilitación individual de pines por cambio de estado
49	<b>Figura 3.21.</b>	Esquema de conexiones para display ánodo común. Ejercicio 3.10	61	<b>Figura 4.14.</b>	Registro PCMSK2 y bits para la habilitación individual de pines por cambio de estado
50	<b>Figura 3.22.</b>	Esquema de tiempos para barrido de displays. Ejercicio 3.11	61	<b>Figura 4.15.</b>	Esquema de conexiones para display ánodo común. Ejercicio 4.1
50	<b>Figura 3.23.</b>	Esquema para conexión barrido 4 displays tipo ánodo común. Ejercicio 3.11	62	<b>Figura 4.16.</b>	Configuración de registros E/S. Ejercicio 4.1
52	<b>Figura 3.24.</b>	Esquema para conexión teclado 4x4 Ejercicio 3.12	62	<b>Figura 4.17.</b>	Configuración de registros interrupciones. Ejercicio 4.1
58	<b>Figura 4.1.</b>	Eventos configurables para una interrupción externa tipo INT	63	<b>Figura 4.18.</b>	Esquema de conexiones. Ejercicio 4.2
58	<b>Figura 4.2.</b>	Bits de configuración del registro EICRA	63	<b>Figura 4.19.</b>	Configuración de registros E/S. Ejercicio 4.2
58	<b>Figura 4.3.</b>	Bits de configuración del registro EICRB	64	<b>Figura 4.20.</b>	Configuración de registros interrupciones. Ejercicio 4.2
			64	<b>Figura 4.21.</b>	Esquema de conexiones. Ejercicio 4.3

65	<b>Figura 4.22.</b>	Configuración de registros E/S. Ejercicio 4.3	80	<b>Figura 5.11.</b>	Bits de configuración del registro UCSRnC
65	<b>Figura 4.23.</b>	Configuración de registros interrupción. Ejercicio 4.3	81	<b>Figura 5.12.</b>	Bits de configuración registro UDRn
66	<b>Figura 4.24.</b>	Esquema de conexiones. Ejercicio 4.4	82	<b>Figura 5.13.</b>	Configuración velocidad normal registro UCSR0A. Ejercicio 5.1
66	<b>Figura 4.25.</b>	Configuración de registros E/S. Ejercicio 4.4	82	<b>Figura 5.14.</b>	Configuración registro UCSR0B sin interrupción por TX y RX. Ejercicio 5.1
66	<b>Figura 4.26.</b>	Configuración de registro PCICR. Ejercicio 4.4	82	<b>Figura 5.15.</b>	Configuración bit UDRIEn. Ejercicio 5.1
67	<b>Figura 4.27.</b>	Configuración de registro PCMSK0. Ejercicio 4.4	82	<b>Figura 5.16.</b>	Habilitar TX y RX en el registro UCSR0B. Ejercicio 5.1
68	<b>Figura 4.28.</b>	Esquema de conexiones. Ejercicio 4.5	82	<b>Figura 5.17.</b>	Configuración bit UCSZn2. Ejercicio 5.1
68	<b>Figura 4.29.</b>	Configuración de registros E/S. Ejercicio 4.5	82	<b>Figura 5.18.</b>	Configuración bit RXB8n y TXB8n. Ejercicio 5.1
68	<b>Figura 4.30.</b>	Configuración de registro PCICR. Ejercicio 4.5	83	<b>Figura 5.19.</b>	Configuración modo asíncrono registro UCSRnC. Ejercicio 5.1
69	<b>Figura 4.31.</b>	Configuración de registro PCMSK1. Ejercicio 4.5	83	<b>Figura 5.20.</b>	Configuración sin paridad registro UCSRnC. Ejercicio 5.1
70	<b>Figura 4.32.</b>	Esquema de conexiones. Ejercicio 4.6	83	<b>Figura 5.21.</b>	Configuración 1 bit de parada registro UCSRnC. Ejercicio 5.1
70	<b>Figura 4.33.</b>	Configuración de registros E/S. Ejercicio 4.6	83	<b>Figura 5.22.</b>	Configuración 8 bits de datos registro UCSRnC. Ejercicio 5.1
70	<b>Figura 4.34.</b>	Configuración de registros interrupciones. Ejercicio 4.6	83	<b>Figura 5.23.</b>	Configuración bit UCPOLn registro UCSRnC. Ejercicio 5.1
73	<b>Figura 5.1.</b>	Transmisión serial	84	<b>Figura 5.24.</b>	Bit 7 registro UCSRN A. Ejercicio 5.2
73	<b>Figura 5.2.</b>	Transmisión paralela	85	<b>Figura 5.25.</b>	Bit 5 del registro UCSRN A. Ejercicio 5.3
74	<b>Figura 5.3.</b>	Transferencia asíncrona de datos	86	<b>Figura 5.26.</b>	Esquema de conexión. Ejercicio 5.4
74	<b>Figura 5.4.</b>	Transferencia síncrona de datos	87	<b>Figura 5.27.</b>	Esquema de conexión. Ejercicio 5.5
74	<b>Figura 5.5.</b>	Modos de comunicación	87	<b>Figura 5.28.</b>	Configuración inicial registro UCSRN B. Ejercicio 5.5
76	<b>Figura 5.6.</b>	Tipos de conexiones puerto serial	89	<b>Figura 5.29.</b>	Esquema de conexión. Ejercicio 5.7
78	<b>Figura 5.7.</b>	Diagrama de bloques USART	92	<b>Figura 6.1.</b>	Diagrama de bloques del contador del timer 0
78	<b>Figura 5.8.</b>	Bits de configuración del registro UBRR	92	<b>Figura 6.2.</b>	Diagrama de bloques del contador del timer 2
79	<b>Figura 5.9.</b>	Bits de configuración registro UCSRN A			
79	<b>Figura 5.10.</b>	Bits de configuración registro UCSRN B			

93	<b>Figura 6.3.</b>	Diagrama de bloques de los contadores de 16 bits	110	<b>Figura 7.2.</b>	Bits de configuración del registro ADCSRB
94	<b>Figura 6.4.</b>	Registro de control A del TCO	110	<b>Figura 7.3.</b>	Resultado de una conversión con justificación a derecha
94	<b>Figura 6.5.</b>	Registro de control B del TCO	110	<b>Figura 7.4.</b>	Resultado de una conversión con justificación a izquierda
94	<b>Figura 6.6.</b>	Registro de control A del TC2	111	<b>Figura 7.5.</b>	Ejemplo de ciclos para una conversión AD
95	<b>Figura 6.7.</b>	Registro de control B del TC2	112	<b>Figura 7.6.</b>	Bits de configuración del registro ADCSRA para selección de prescaler
95	<b>Figura 6.8.</b>	Registro de control A del TCn	113	<b>Figura 7.7.</b>	Bits de configuración del registro ADCSRB para la selección de fuentes de disparo
95	<b>Figura 6.9.</b>	Registro de control B del TCn	113	<b>Figura 7.8.</b>	Bits de configuración del registro ADCSRA para modos de conversión
95	<b>Figura 6.10.</b>	Representación gráfica del funcionamiento del TC de 8 bits.	114	<b>Figura 7.9.</b>	Bits de configuración del registro DIDR0 para deshabilitar buffer entrada digital (0-7)
95	<b>Figura 6.11.</b>	Representación gráfica del funcionamiento del TC de 16 bits.	114	<b>Figura 7.10.</b>	Bits de configuración del registro DIDR2 para deshabilitar buffer entrada digital (8-15)
96	<b>Figura 6.12.</b>	Registro de enmascaramiento de interrupciones de los TC de 8 bits. Ejercicio 6.1	114	<b>Figura 7.11.</b>	Esquema de conexiones para visualizar una conversión A/D mediante barrido de display. Ejercicio 7.1
96	<b>Figura 6.13.</b>	Registro de enmascaramiento de interrupciones de los TC de 16 bits. Ejercicio 6.1	115	<b>Figura 7.12.</b>	Configuración de registros ADMUX, ADCSRB. Ejercicio 7.1
97	<b>Figura 6.14.</b>	Diagrama de bloques del TCO.	115	<b>Figura 7.13.</b>	Configuración de registros ADCSRA. Ejercicio 7.1
97	<b>Figura 6.15.</b>	Diagrama de bloques de los TC de 16 bits.	117	<b>Figura 7.14.</b>	Configuración de registros ADCSRA. Ejercicio 7.2
99	<b>Figura 6.16.</b>	Generación de ondas en OC0A y OC0B del TCO. Ejercicio 6.2	118	<b>Figura 7.15.</b>	Esquema de conexiones para visualizar una conversión A/D mediante monitor serial. Ejercicio 7.3
100	<b>Figura 6.17.</b>	Generación de ondas en OC0A y OC0B del TCO	118	<b>Figura 7.16.</b>	Configuración de registros ADMUX, ADCSRB. Ejercicio 7.3
103	<b>Figura 6.18.</b>	Diagrama de tiempos del modo PWM Phase Correct	119	<b>Figura 7.17.</b>	Configuración de registros ADCSRA. Ejercicio 7.3
104	<b>Figura 6.19.</b>	Diagrama de tiempos del modo PWM Phase and Frequency Correct	120	<b>Figura 7.18.</b>	Configuración de registros ADMUX, ADCSRB. Ejercicio 7.4
104	<b>Figura 6.20.</b>	Diagrama de bloques de la unidad de captura.			
105	<b>Figura 6.21.</b>	Diagrama de bloques del sensor conectado al microcontrolador. Ejercicio 6.8			
105	<b>Figura 6.22.</b>	Diagrama de tiempos del sensor HC-SR04. Ejercicio 6.8			
110	<b>Figura 7.1.</b>	Bits de configuración del registro ADMUX			

- 120 Figura 7.19.** Configuración de registros ADCSRA.  
Ejercicio 7.4
- 122 Figura 7.20.** Esquema de conexiones para  
visualizar varias conversiones  
A/D mediante monitor serial.  
Ejercicio 7.5
- 123 Figura 7.21.** Configuración de registros ADMUX,  
ADCSRB. Ejercicio 7.5
- 123 Figura 7.22.** Configuración de registros ADCSRA.  
Ejercicio 7.5
- 123 Figura 7.23.** Esquema para la conversión de varios  
canales. Ejercicio 7.5

# Índice de tablas

7	<b>Tabla 1.1.</b>	Sistemas de numeración
8	<b>Tabla 1.2.</b>	Operación de suma
12	<b>Tabla 1.3.</b>	Representación de números con signo
17	<b>Tabla 1.4.</b>	Representación de números binarios en BCD
18	<b>Tabla 1.5.</b>	Representación de números en código Gray
19	<b>Tabla 1.6.</b>	Caracteres ASCII
31	<b>Tabla 2.1</b>	Especificaciones técnicas Arduino Mega 2560
32	<b>Tabla 2.2</b>	Descripción general de los pines
41	<b>Tabla 3.1.</b>	Comportamiento de los pines E/S en función de los registros asociados PORTx, DDRx y MCUCR
47	<b>Tabla 3.2.</b>	Operaciones abreviadas para Atmel Studio
55	<b>Tabla 4.1.</b>	Vectores de reset e interrupción
57	<b>Tabla 4.2.</b>	Interrupciones y pines asociados
59	<b>Tabla 4.3.</b>	Configuración de eventos para generación de interrupciones
73	<b>Tabla 5.1.</b>	Diferencias entre serial y paralelo
80	<b>Tabla 5.2.</b>	Bits de configuración de tipo de transmisión
80	<b>Tabla 5.3.</b>	Bits de configuración de paridad
80	<b>Tabla 5.4.</b>	Configuración bits de parada
81	<b>Tabla 5.5.</b>	Cantidad de bits de envío y recepción
81	<b>Tabla 5.6.</b>	Configuración del bit UCPOLn
93	<b>Tabla 6.1.</b>	Modos de operación de los TCO y TC2
93	<b>Tabla 6.2.</b>	Modos de operación de los TC1, TC3, TC4 y TC5
94	<b>Tabla 6.3.</b>	Descripción de los bits de selección de reloj del TCn (excepto TC2)
95	<b>Tabla 6.4.</b>	Descripción de los bits de selección de reloj del TC2
96	<b>Tabla 6.5.</b>	Posibles bases de tiempo usando el modo normal. Ejercicio 6.1
98	<b>Tabla 6.6.</b>	Descripción de los bits COM de los TCn en el modo CTC
100	<b>Tabla 6.7.</b>	Descripción de los bits COM de los TCn en el modo Fast PWM
101	<b>Tabla 6.8.</b>	Valores de TOP en función del prescaler. Ejercicio 6.5
101	<b>Tabla 6.9.</b>	Valores de frecuencia y errores según valores de la tabla 6.4
103	<b>Tabla 6.10.</b>	Valores de TOP en función del prescaler. Ejercicio 6.7
108	<b>Tabla 7.1.</b>	Canales de entrada simple
108	<b>Tabla 7.2.</b>	Canales diferenciales y ganancias
110	<b>Tabla 7.3.</b>	Canales de entrada simple
111	<b>Tabla 7.4.</b>	Ciclos de muestra/retención y tiempo de conversión
112	<b>Tabla 7.5.</b>	Configuración de bits para selección de prescaler
112	<b>Tabla 7.6.</b>	Configuración de bits para seleccionar las fuentes de disparo

# Índice de imágenes

- |   |  |
|---|--|
| 25 <b>Imagen 2.1.</b> Clasificación de microcontroladores | 37 <b>Imagen 2.23.</b> Construcción de la solución   |
| 25 <b>Imagen 2.2.</b> Familias de microcontroladores      | 37 <b>Imagen 2.24.</b> Archivos generados por la compilación   |
| 27 <b>Imagen 2.3.</b> Samsung Artika 710                  | 40 <b>Imagen 3.1.</b> Esquema de pines para el<br>microcontrolador ATmega2560                          |
| 27 <b>Imagen 2.4.</b> Placa SensorTile                    | 77 <b>Imagen 5.1.</b> Convertidores USB-TTL  |
| 27 <b>Imagen 2.5.</b> Módulo Adalm-Pluto                  | 77 <b>Imagen 5.2.</b> Puertos USART mega 2560  |
| 28 <b>Imagen 2.6.</b> Placa AudioSmart                    | 92 <b>Imagen 6.1.</b> Distribución de terminales   |
| 28 <b>Imagen 2.7.</b> Placa Thunderboard Sense            | 97 <b>Imagen 6.2.</b> Ondas generadas usando el modo<br>normal de los TC0, TC2 y TC4. Ejercicio<br>6.1 |
| 29 <b>Imagen 2.8.</b> Placa Flora AdaFruit                | 99 <b>Imagen 6.3.</b> Ondas de salida por OC0A y OC0B del<br>TC0. Ejercicio 6.2                        |
| 29 <b>Imagen 2.9.</b> Placa LaunchPad                     | 100 <b>Imagen 6.4.</b> Ondas de salida por OC1A, OC1B y<br>OC1B del TC1. Ejercicio 6.3                 |
| 29 <b>Imagen 2.10.</b> Placa PICAXE                       | 101 <b>Imagen 6.5.</b> Ondas PWM por OC0A y OC0B.<br>Ejercicio 6.4                                     |
| 30 <b>Imagen 2.11.</b> Placa Wiring S.                    | 102 <b>Imagen 6.6.</b> Onda PWM por OC0B con valor<br>“exacto”. Ejercicio 6.5                          |
| 30 <b>Imagen 2.12.</b> Placa Netduino                     | 102 <b>Imagen 6.7.</b> Onda PWM por OC1A, OC1B y OC1C<br>con valor “exacto”. Ejercicio 6.6             |
| 31 <b>Imagen 2.13.</b> Placas comerciales Arduino         | 104 <b>Imagen 6.8.</b> Ondas PWM Phase Correct por OC3A,<br>OC3B y OC3C. Ejercicio 6.7                 |
| 31 <b>Imagen 2.14.</b> Arduino Mega 2560                  |  |
| 32 <b>Imagen 2.15.</b> Mega Pinout                        |  |
| 32 <b>Imagen 2.16.</b> Arduino Mega pinout                |  |
| 34 <b>Imagen 2.17.</b> Software Atmel StudioImagen 2.18   |  |
| 35 <b>Imagen 2.18</b> Ventana principal Atmel Studio 7    |  |
| 35 <b>Imagen 2.19.</b> Configuración nuevo proyecto       |  |
| 35 <b>Imagen 2.20.</b> Selección de microcontrolador      |  |
| 36 <b>Imagen 2.21.</b> Área de trabajo Atmel Studio 7     |  |
| 36 <b>Imagen 2.22.</b> Código en Atmel Studio 7           |  |

## BIBLIOGRAFÍA

- Adafruit.** (2019). FLORA-Wearable electronic platform: Arduino-compatible-v3. Recuperado de <https://www.adafruit.com/product/659>.
- Analog Devices.** (2019). ADALM - PLUTO. Recuperado de <https://www.analog.com/en/design-center/evaluation-hardware-and-software/evaluation-boards-kits/adalm-pluto.html#eb-overview>.
- Arduino.** (2018). Arduino. Recuperado de i58from <https://www.arduino.cc/>.
- Arduino.** (2020a). ARDUINO NANO. Recuperado de <https://store.arduino.cc/usa/arduino-nano>
- Arduino.** (2020b). Paquete de Fundamentos de Arduino. Recuperado de <https://store.arduino.cc/usa/>.
- Atmel.** (2014). *Atmel ATmega640/V-1280/V-1281/V-2560/V-2561/V*. San José: Atmel.
- Barragán, H.** (2019). Hardware Wiring. Recuperado de <http://wiring.org.co/hardware/es/>.
- Britannica, T. E. of E.** (2019). *ASCII*. Encyclopædia Britannica.
- Cass, S.** (2018). Chip Hall of Fame: Intel 4004 Microprocessor. Recuperado de <https://spectrum.ieee.org/tech-history/silicon-revolution/chip-hall-of-fame-intel-4004-microprocessor>.
- Digi-Key Electronics.** (2019). Kits de desarrollador y módulos ARTIK 710. Recuperado de <https://www.digikey.com/es/products/highlights/samsung-led/artik-710-modules-and-developer-kits>.
- directindustry.** (2020). Microcontrolador 32 bits STM32 series. Recuperado de <https://www.directindustry.es/prod/stmicroelectronics/product-33699-558379.html>.
- Hall, B. E.** (s.f.). *Serial Interfaces: Demystified*. Recuperado de <http://www.ece.ubc.ca/~edc/464/lectures/lec14.pdf>
- HETPRO.** (2017). Puerto Serial – protocolo y su teoría. Recuperado de <https://hetpro-store.com/TUTORIALES/puerto-serial/>
- Huang, H.-W.** (1996). *An Introduction: Software and Hardware Interfacing*. West Publishing Company.
- Madruga, B.** (2016). Conceptos Introductorios a Los Microcontroladores. Recuperado de <https://www.scribd.com/document/299965499/1-Conceptos-Introductorios-a-Los-Microcontroladores>.
- Martin, L.** (2012). Placa de desarrollo con PICAXE. Recuperado de <http://www.automatismos-mdq.com.ar/blog/2012/12/placa-de-desarrollo-con-picaxe.html>
- Maxembedded.** (2014). Serial Communication – Introduction. Recuperado de <http://maxembedded.com/2013/09/serial-communication-introduction/>

- Mazidi, M. A., Naimi, S., & Sepehr Naimi.** (2011). *The avr microcontroller and embedded system using assembly and C*. New Jersey: Prentice Hall.
- Microchip Technology, I.** (2020a). AT32UC3A1512. Recuperado de <https://www.microchip.com/wwwproducts/en/AT32UC3A1512>
- Microchip Technology, I.** (2020b). Atmel Studio 7. Recuperado de <https://www.microchip.com/mplab/avr-support/atmel-studio-7>
- Microchip Technology, I.** (2020c). Configuración de megaAVR USART. Recuperado de <https://microchipdeveloper.com/8avr:uart-mega-configuration>.
- Microchip Technology, I.** (2020d). *La cartera de microcontroladores (MCU) de 32 bits más amplia e innovadora de la industria*. Recuperado de <https://www.microchip.com/design-centers/32-bit>
- Netduino.foundation.** (2019). NETDUINO. Recuperado de <https://www.wildernesslabs.co/netduino>
- Richard, B., Cox, S., & O'Cull, L.** (2007). *Embedded C Programming and the Atmel AVR* (2.<sup>a</sup> edición). New York: THOMSON.
- Synaptics Incorporated.** (2019). AudioSmart DS20921. Recuperado de <https://www.synaptics.com/partners/amazon/ds20921>
- Texas Instruments Incorporated.** (2017). *LM35 Precision Centigrade Temperature Sensors*. Recuperado de <https://www.ti.com/lit/ds/symlink/lm35.pdf>
- Texas Instruments Incorporated.** (2019). MSP430G2 LaunchPad Development kit. Recuperado de <http://www.ti.com/tool/MSP-EXP430G2>
- Villamil, H.** (2009). Microprocesadores y microcontroladores (p. 295). Recuperado de [https://repository.unad.edu.co/bitstream/handle/10596/6933/M\\_309696\\_Microp %26 Microc\\_Ing\\_Electronica.pdf?sequence=1&isAllowed=y](https://repository.unad.edu.co/bitstream/handle/10596/6933/M_309696_Microp %26 Microc_Ing_Electronica.pdf?sequence=1&isAllowed=y)
- Velasco, J.** (2011). Historia de la Tecnología: 40 años del Intel 4004. Recuperado de <https://hipertextual.com/2011/11/40-aniversario-intel-4004>
- Williams, E.** (2014). *Make: AVR Programming* (1.<sup>a</sup> ed.). San Francisco: Maker Media.
- Zephyr Project.** (2019). ST SensorTile.box. Recuperado de [https://docs.zephyrproject.org/latest/boards/arm/sensortile\\_box/doc/index.html](https://docs.zephyrproject.org/latest/boards/arm/sensortile_box/doc/index.html)



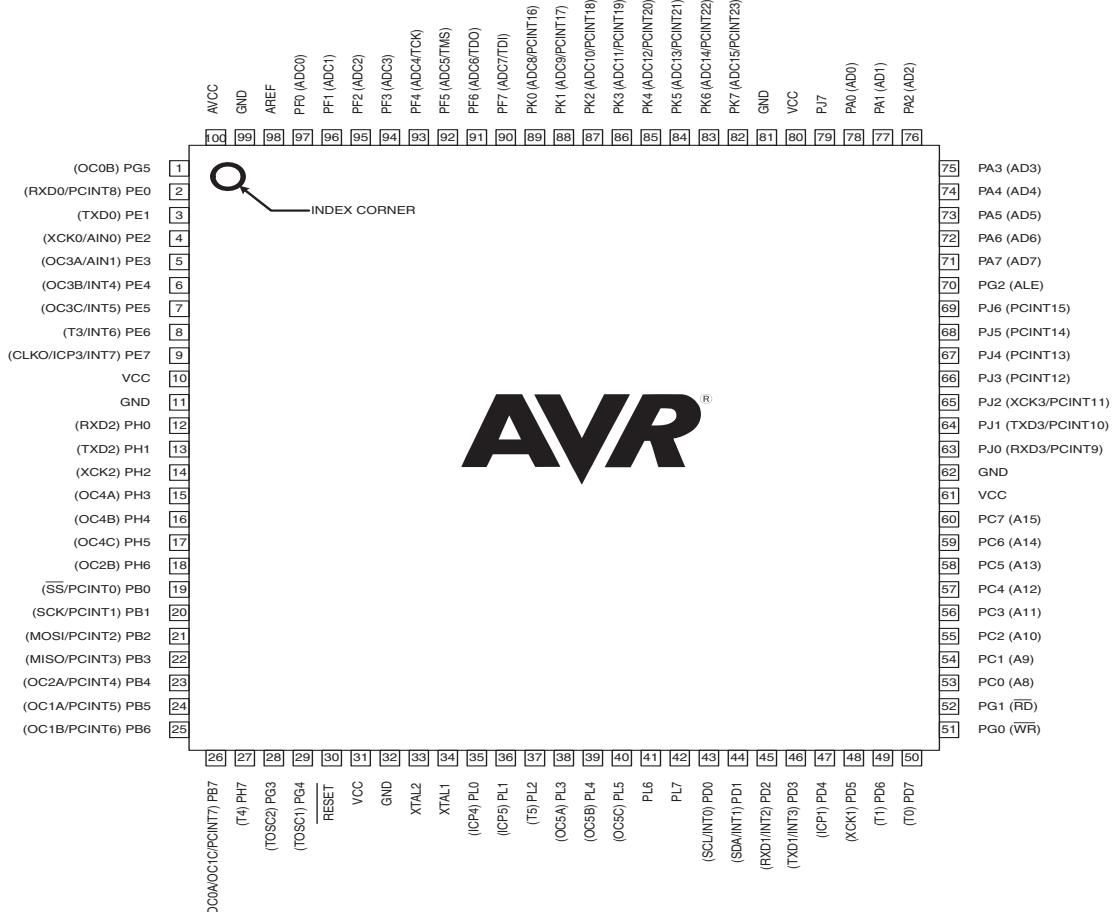
# ANEXOS

IMAGEN 2.15

**CLICK PARA REGRESAR**

## 1. Pin Configurations

Figure 1-1. TQFP-pinout ATmega640/1280/2560



# MEGA PINOUT

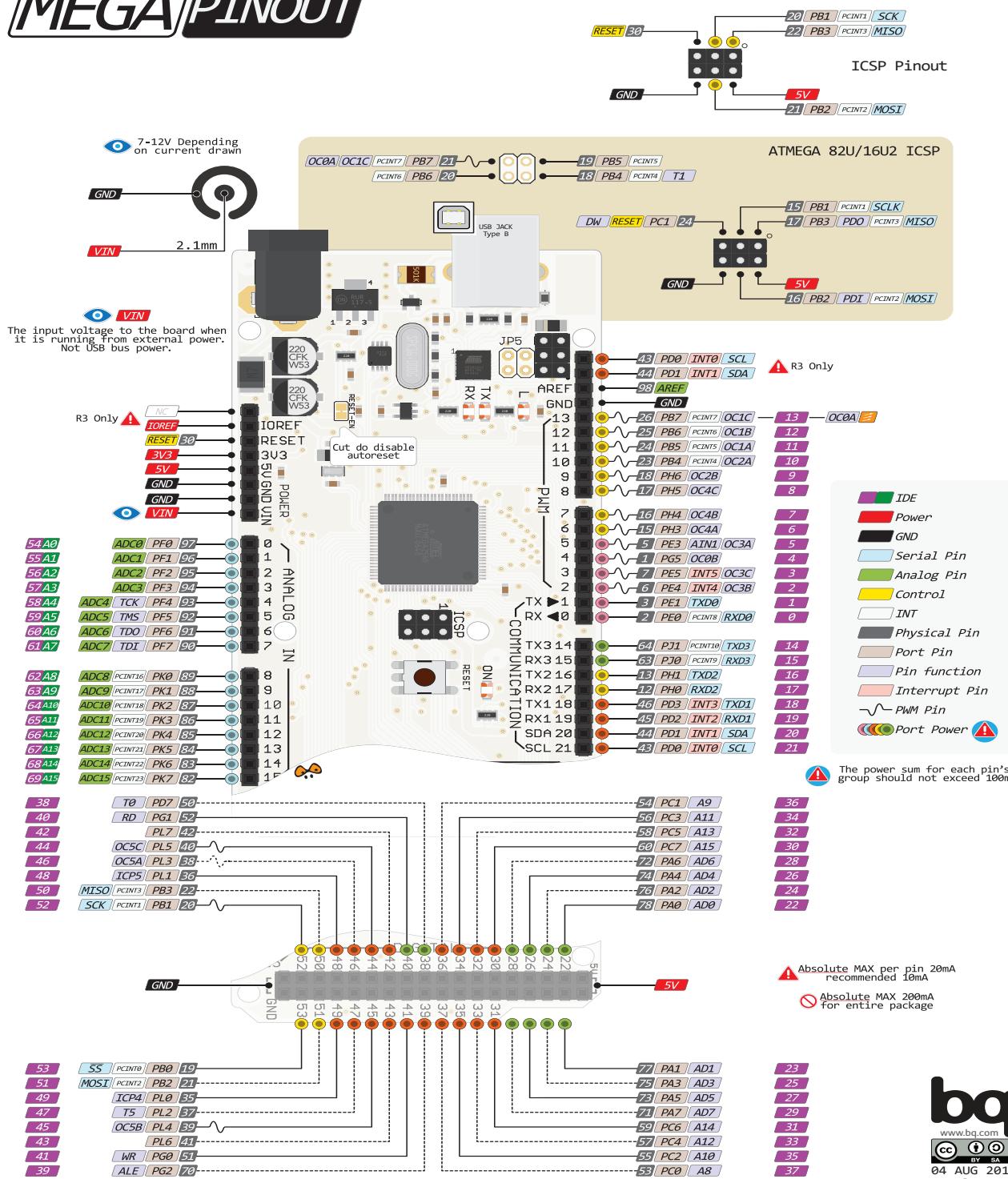


IMAGEN 2.17

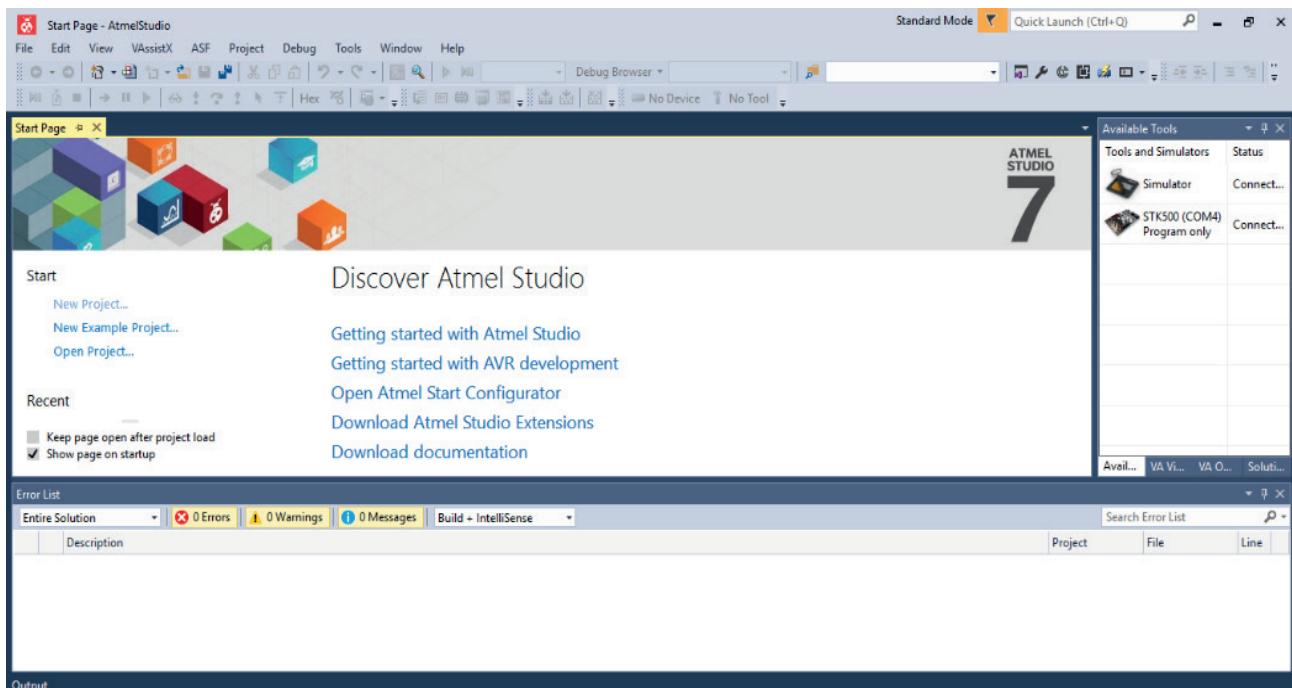


IMAGEN 2.18

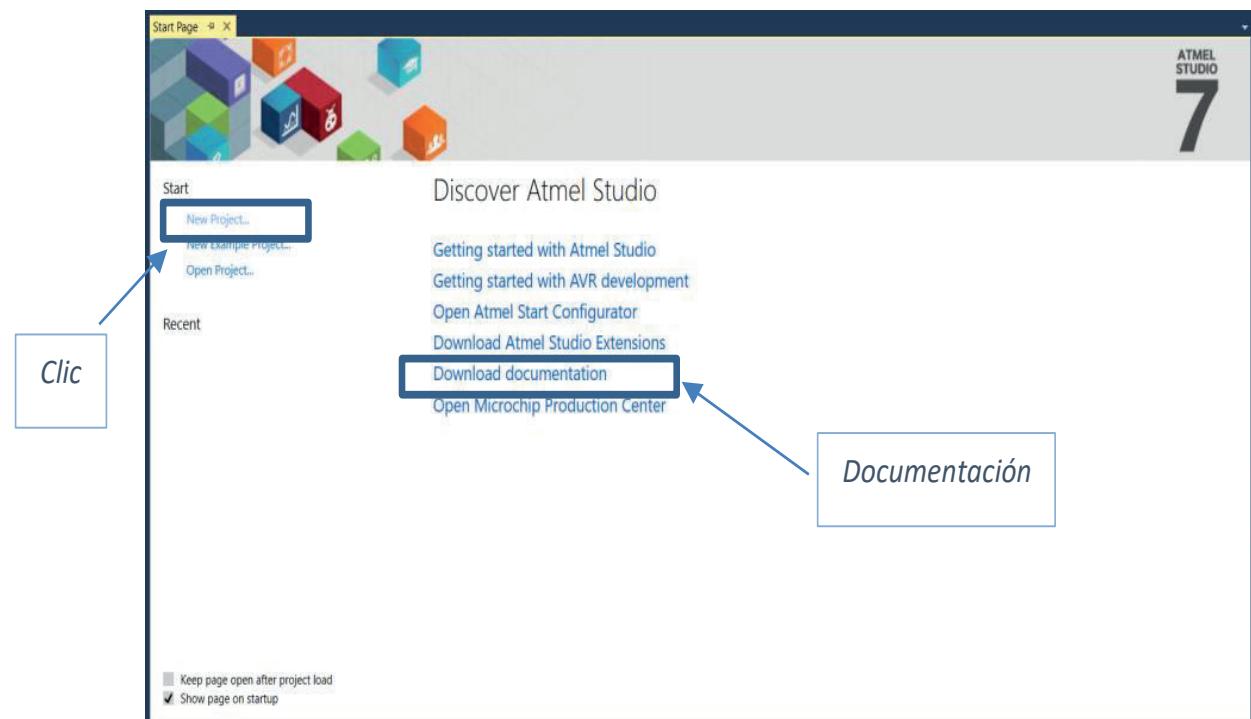


IMAGEN 2.19

CLICK PARA REGRESAR

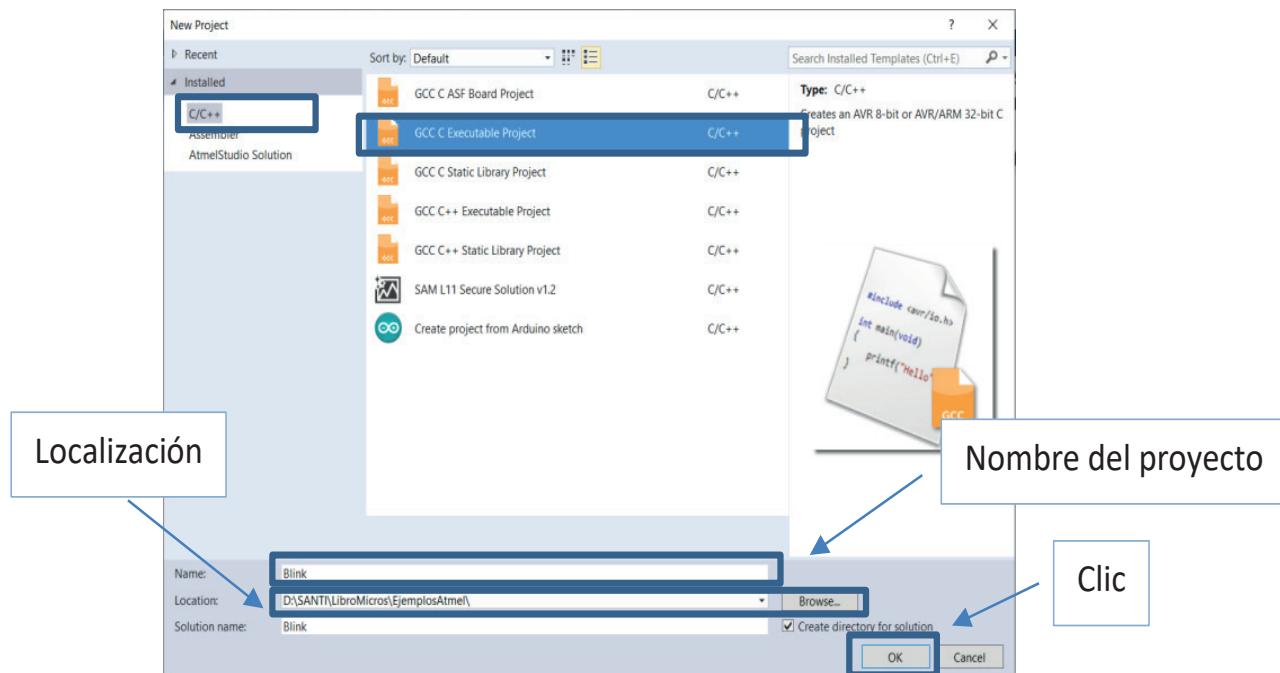


IMAGEN 2.20

CLICK PARA REGRESAR

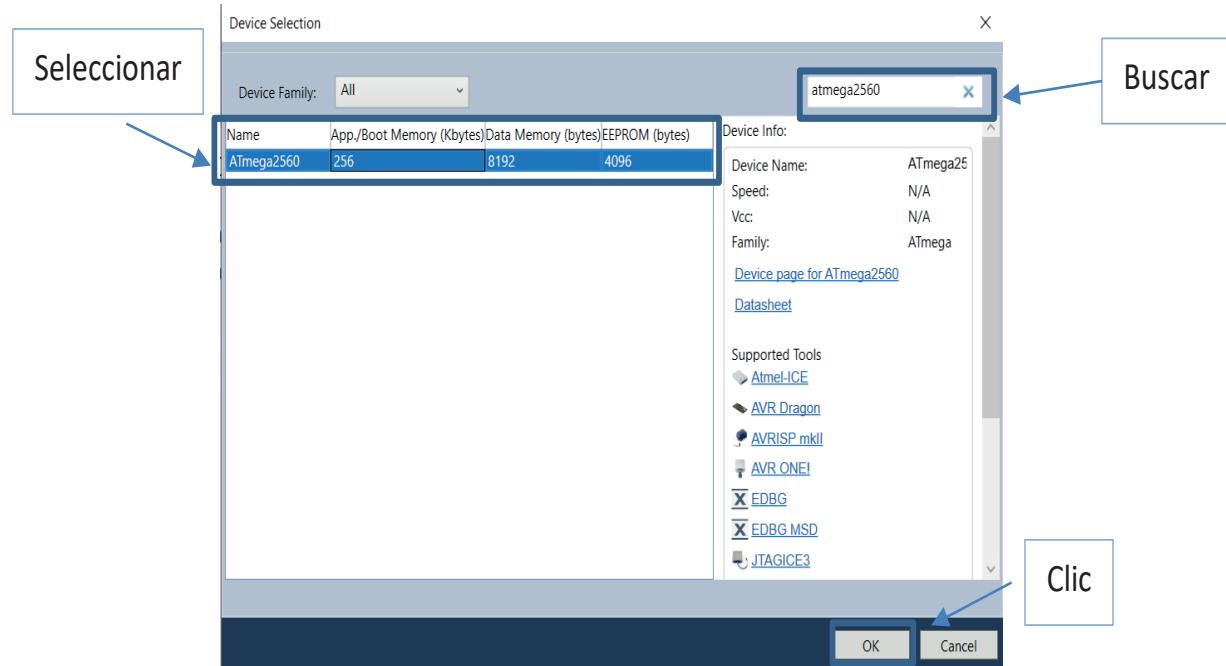


IMAGEN 2.21 | CLICK PARA REGRESAR <|>

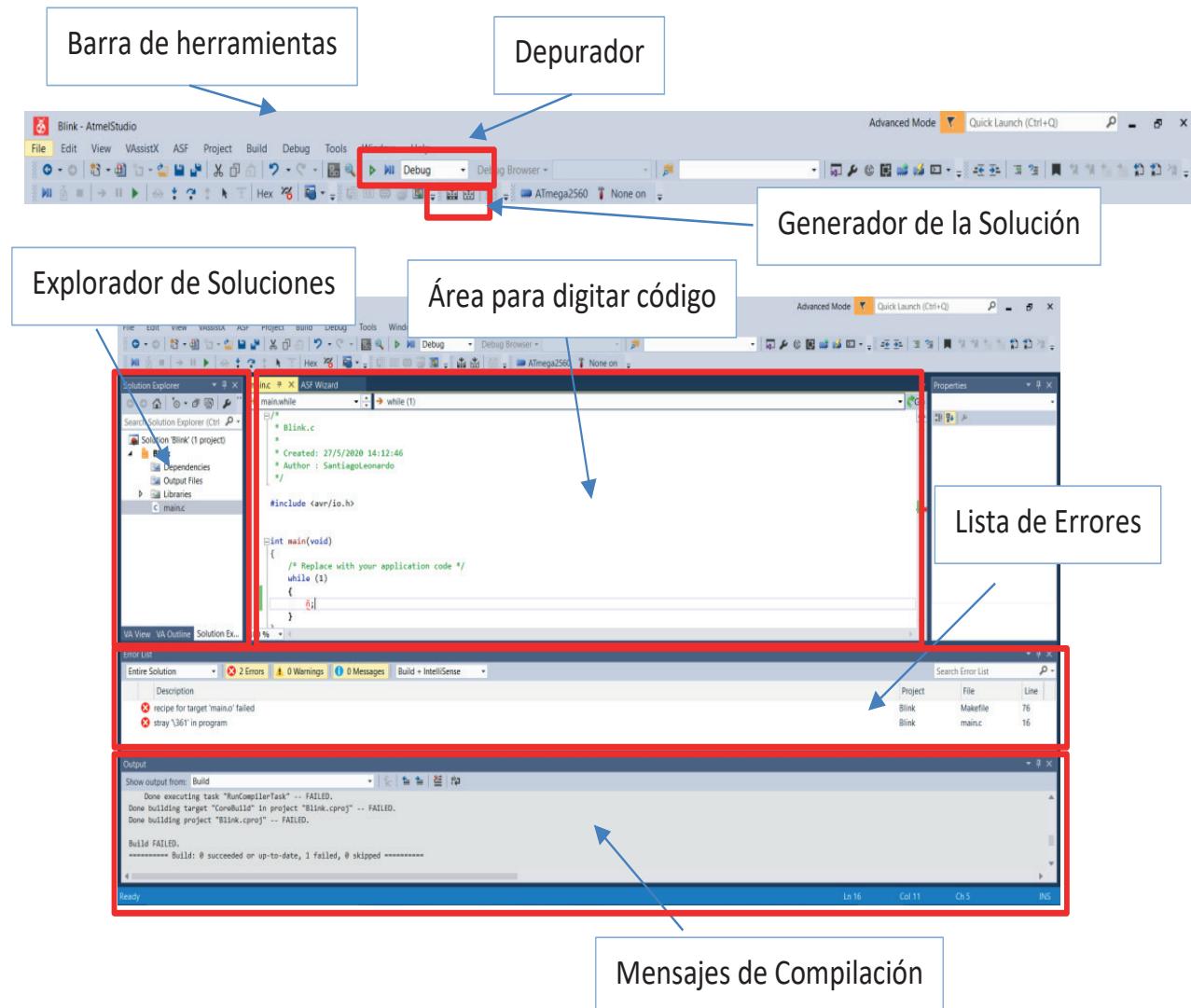


IMAGEN 2.22 | CLICK PARA REGRESAR | <

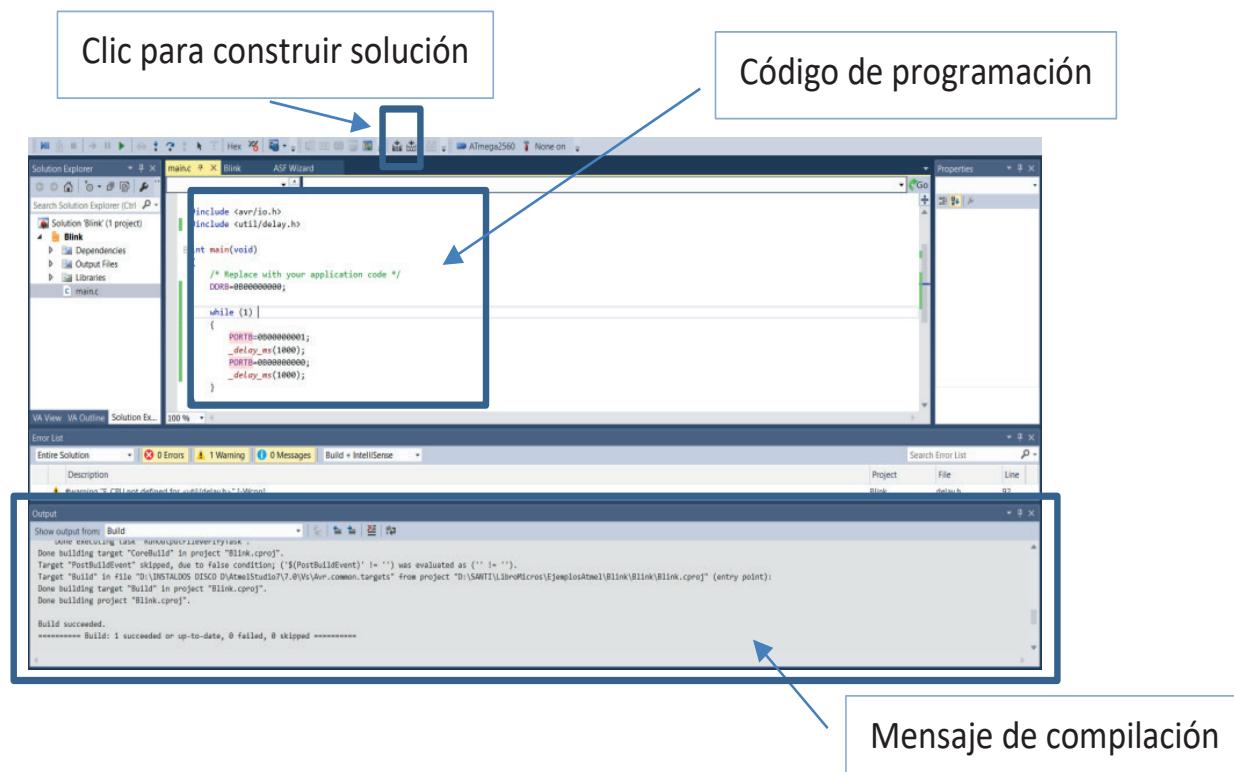
The screenshot shows the AVR Studio 7 interface with the main.c file open in the code editor. The code is a simple blink program for an ATmega2560. It includes includes for avr/io.h and util/delay.h, defines DDRB=0B00000000, and a main loop where PORTB toggles between 0B00000001 and 0B00000000 every 1000ms using \_delay\_ms(1000). The code editor has syntax highlighting and a status bar showing 100%.

```
#include <avr/io.h>
#include <util/delay.h>

int main(void)
{
    /* Replace with your application code */
    DDRB=0B00000000;

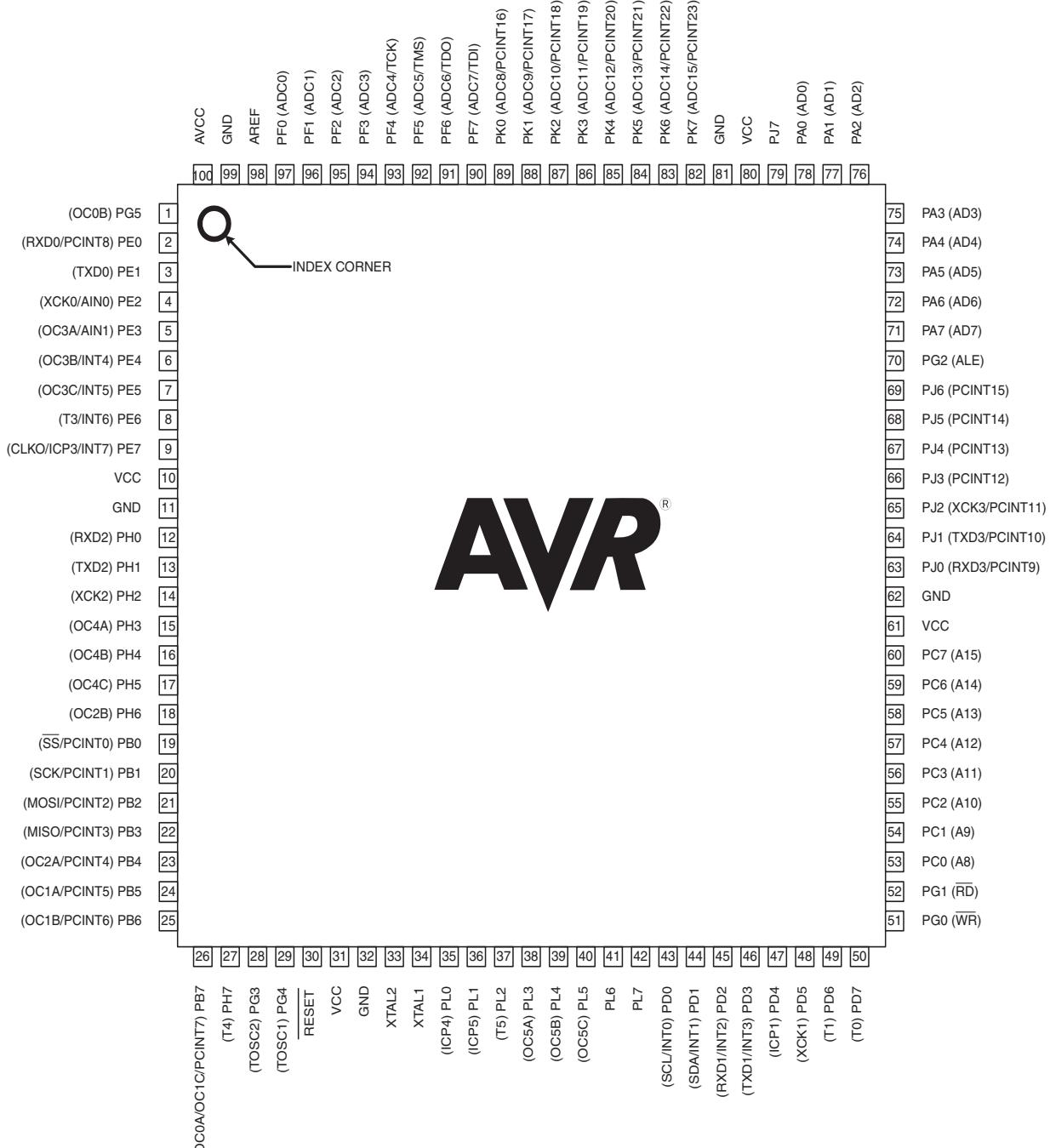
    while (1)
    {
        PORTB=0B00000001;
        _delay_ms(1000);
        PORTB=0B00000000;
        _delay_ms(1000);
    }
}
```

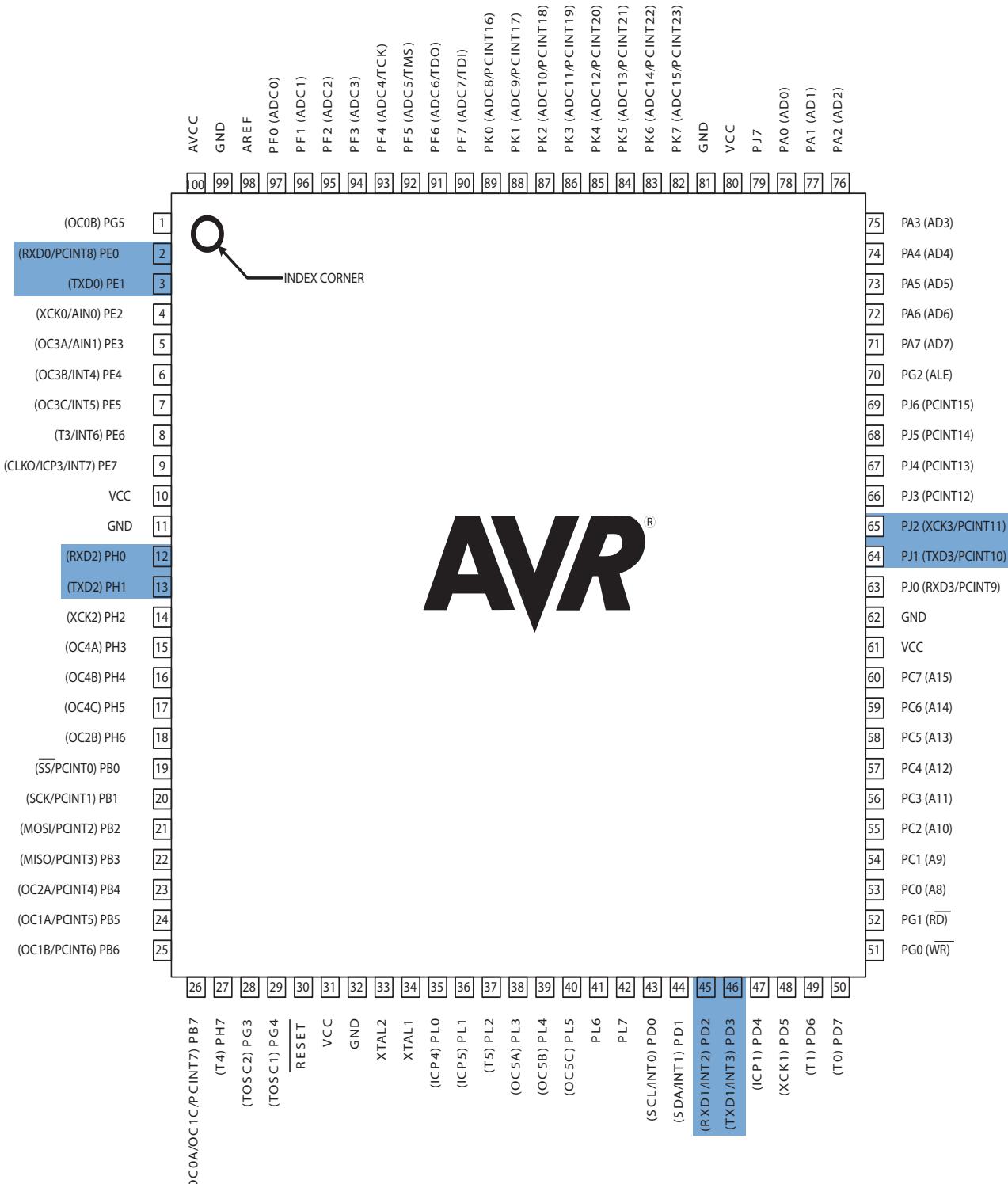
IMAGEN 2.23 | CLICK PARA REGRESAR | <



## 1. Pin Configurations

**Figure 1-1.** TQFP-pinout ATmega640/1280/2560





Estimado/a lector/a:

DESCARGAR



En este enlace (<https://www.udlaediciones.com.ec/wp-content/uploads/2022/05/CODIGO-MODIFICABLE-DEL-LIBRO.docx>) se encuentra un archivo con el que podrán descargar todos los códigos utilizados en esta publicación, para su uso estudiantil pertinente.

**ATmega2560:**  
**Teoría y aplicaciones usando Atmel Studio**  
se terminó de editar,  
en Quito, Ecuador,  
el mes de abril de 2022  
bajo la marca



siendo rector el Dr. Carlos Larreátegui Nardi

ISBN: 978-9-9427-7951-9



9 789942 779519

Este libro exhibe ideas fundamentales sobre el manejo de microcontroladores en general. Y, a la vez, aterriza su conceptualización en uno en particular: el Atmega2560. Aquí se presentan varios ejemplos de aplicación junto con ilustraciones y gráficos que facilitan su comprensión y permiten al lector aplicar los conocimientos adquiridos de manera práctica. Los algoritmos de programación son presentados y explicados detalladamente, de tal manera que puedan ser fácilmente replicados usando el *software* y *hardware* propuesto en el texto. Esta obra es el resultado de varios años de enseñanza en la materia de Microcontroladores, dictada desde la Facultad de Ingeniería y Ciencias Aplicadas de la Universidad de Las Américas por parte de los autores.