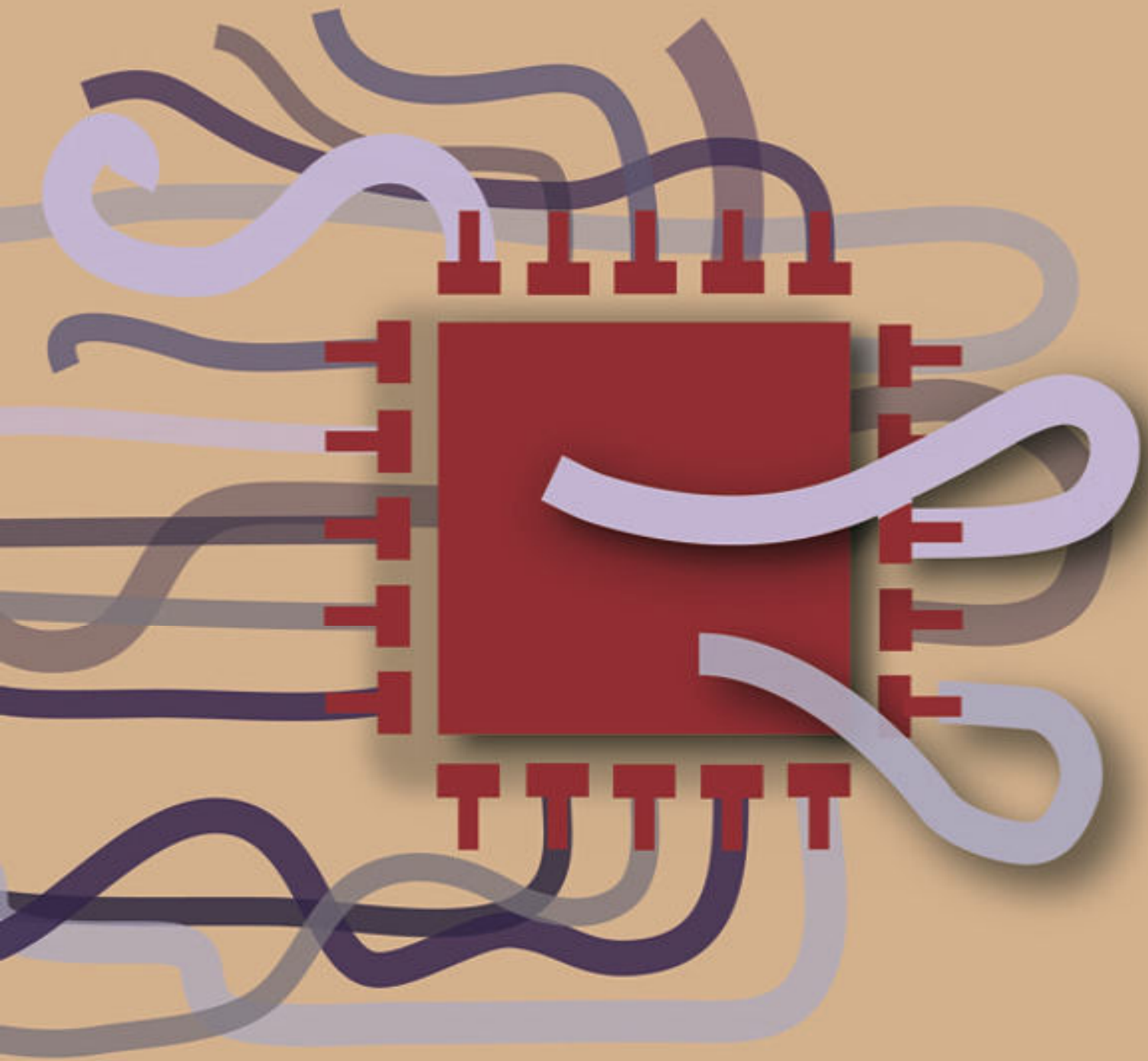
LOS MICROCONTROLADORES AVR DE ATMEL



Felipe Santiago Espinosa

Universidad Tecnológica de la Mixteca
Huajuapan de León, Oax.

Directorio

Dr. Modesto Seara Vázquez
Rector

M.C. Gerardo García Hernández
Vice-Rector Académico

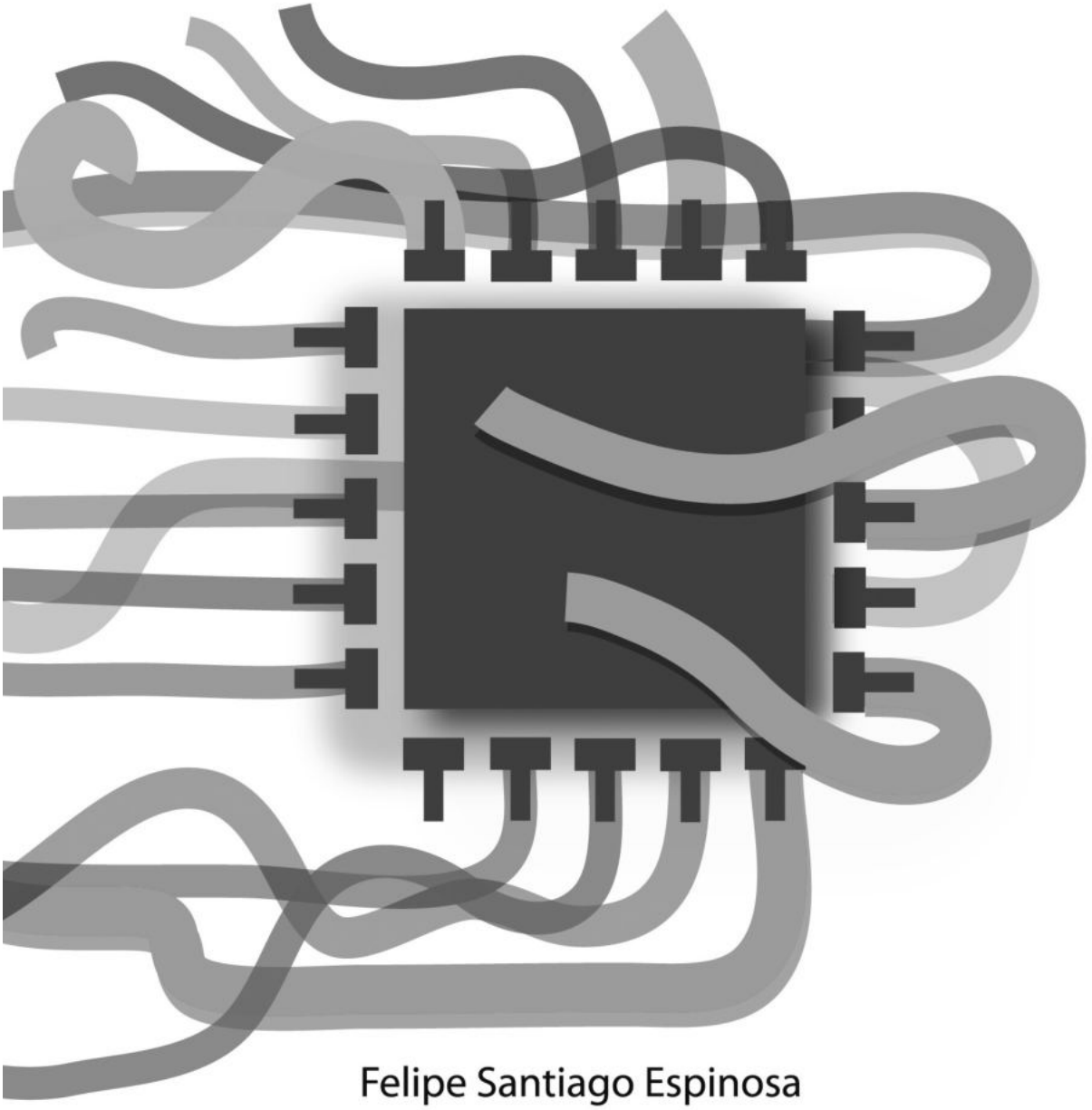
C.P. José Javier Ruiz Santiago
Vice-Rector Administrativo

Lic. María de los Ángeles Peralta Arias
Vice-Rectora de Relaciones y Recursos

LOS MICROCONTROLADORES AVR DE ATMEL

ATMega8 y ATMega16

Programación en Ensamblador y Lenguaje C



Felipe Santiago Espinosa

UNIVERSIDAD TECNOLÓGICA DE LA MIXTECA

Primera Edición; mayo 2012
ISBN: 978-607-95222-7-8

Carr. a Acatlima Km. 2.5
Huajuapan de León, Oaxaca.
C.P. 69000 Tel. 9535320214
www.utm.mx

Diseño: Alfonso Acosta Romero
Dir. Editorial:Reina Ortiz Escamilla

Impreso y hecho en México
Printed and made in México

Índice

Prólogo

Comencé a trabajar con microcontroladores en el año de 1994, precisamente en uno de mis últimos cursos de licenciatura. Un microcontrolador también suele ser referido como MCU (*Micro Controller Unit*), por lo que a lo largo del texto, indistintamente es tratado de una u otra manera.

El primer MCU que utilicé fue un 8031, un microcontrolador de 8 bits perteneciente a la familia MCS-51 de Intel. El 8031 requiere de todo un sistema de acondicionamiento para ser puesto en marcha. Posteriormente, otros microcontroladores llegaron a mis manos, adquirí experiencia trabajando con el DS5000T, una versión mejorada del 8031, con memoria de programa tipo NVRAM (RAM no volátil) y un reloj de tiempo real, pero manufacturado por Dallas Semiconductor. Luego, conocí a la familia de microcontroladores PIC de Microchip, tuve una ligera experiencia con el HC11 de Motorola y, en los últimos años, he trabajado con los microcontroladores AVR, de ATMEL.

Desde mi incorporación a la Universidad Tecnológica de la Mixteca, en 1998, año con año he impartido el curso de microcontroladores, utilizando uno u otro dispositivo, según la disponibilidad o requerimientos de las aplicaciones. Con la experiencia adquirida he observado que los microcontroladores AVR tienen más recursos en relación con sus equivalentes en costo de otras compañías, además de un rendimiento más alto.

Por ello, desde el año 2006 he enfocado mis cursos al manejo de los microcontroladores AVR, específicamente trabajando con el ATMega8 y el ATMega16. El primer paso para trabajar con estos dispositivos fue la búsqueda del libro de texto adecuado. Necesitaba un libro que detallara al hardware y lo vinculara con el software, que sentara las bases para el desarrollo de sistemas y permitiera a los estudiantes empezar desde cero en los microcontroladores, hasta adquirir ideas aplicables al desarrollo de sistemas complejos. Y que además, incluyera aspectos relacionados con su programación, tanto en Ensamblador, como en Lenguaje C. Al no encontrarlo, me di a la tarea de escribirlo.

En este libro pretendo reflejar la experiencia que he adquirido con estos dispositivos. Es un libro de texto básico, inicialmente para mis cursos y más adelante, quizás, también sea empleado en otras universidades o por profesionistas independientes interesados en este apasionante mundo de los microcontroladores.

Dado que el tema central son los microcontroladores, supongo que los lectores tienen fundamentos de electrónica digital, esto involucra un conocimiento de sistemas numéricos, compuertas lógicas, registros, memorias, máquinas de estados, etc., incluso algunos aspectos básicos de programación en ensamblador y en Lenguaje C, u otro lenguaje de alto nivel.

Por lo tanto, me enfoco en las características de los microcontroladores y, sólo si es necesario, profundizo en algún concepto en torno a ellos, pero sin desviarme del tema de interés.

A lo largo del texto, realizo una descripción del hardware y el software de los microcontroladores ATMega8 y ATMega16, mostrando cómo los diferentes recursos de hardware pueden ser manejados en Ensamblador o en Lenguaje C. Éste es un aspecto interesante, dado que pretendo mostrar las ventajas o inconvenientes de desarrollar aplicaciones en diferentes niveles de programación. Para todos los recursos internos, he documentado ejemplos completos, los cuales fueron previamente implementados como prácticas en la Universidad Tecnológica de la Mixteca.

Dispongo de un capítulo dedicado al manejo de dispositivos externos y concluyo con la propuesta de una metodología que se puede emplear para construir sistemas con más requerimientos, la cual ilustro con el desarrollo de dos sistemas relativamente complejos.

Agradezco a la Universidad Tecnológica de la Mixteca las facilidades para llevar a cabo la redacción de este libro, deseo sea de utilidad para las futuras generaciones de ésta y otras instituciones. También agradezco a todos los alumnos y profesores que, de una u otra manera, colaboraron en la realización y revisión de este texto.

Felipe. Santiago Espinosa
fsantiag@mixteco.utm.mx

1. Introducción a los Microcontroladores

En este capítulo se da una introducción al tema, exponiendo conceptos generales, es decir, conceptos que no están enfocados a un MCU particular. Se describen los alcances y limitaciones de estos dispositivos y se muestra una organización común a la mayoría de microcontroladores.

1.1 Sistemas Electrónicos

La electrónica ha evolucionado de manera sorprendente en los últimos años, tanto que actualmente no es posible concebir la vida sin los sistemas electrónicos. Los sistemas electrónicos son una parte fundamental en el trabajo de las personas, proporcionan entretenimiento y facilitan las actividades en los hogares.

Un sistema electrónico puede ser representado con el diagrama de la figura 1.1, sin importar la funcionalidad para la cual haya sido diseñado.

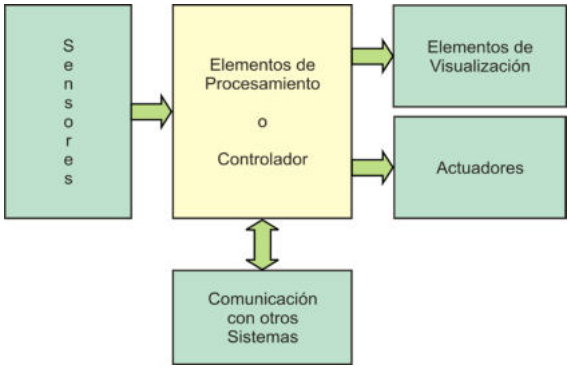


Figura 1.1 Abstracción de un sistema electrónico

El sistema recibe las peticiones de los usuarios o conoce lo que ocurre en su entorno por medio de los sensores. Los sensores son dispositivos electrónicos que se encargan de acondicionar diferentes tipos de información a un formato reconocido por los elementos de procesamiento. Un sensor puede ser tan simple como un botón o tan complejo como un reconocedor de huella digital, pero si los elementos de procesamiento son digitales, en ambos casos la salida va a estar codificada en 1's y 0's. Con los sensores se pueden monitorear diferentes parámetros, como: temperatura, humedad, velocidad, intensidad luminosa, etc.

Los elementos de visualización son dispositivos electrónicos que muestran el estado actual del sistema, notificando al usuario si debe tomar acciones. Los elementos de visualización típicos son: LEDs individuales o matrices de LEDs, displays de 7 segmentos o de cristal líquido.

Los actuadores son dispositivos electrónicos o electromecánicos que también forman parte de las salidas de un sistema, pero con la capacidad de modificar el entorno, es decir, van más allá de la visualización, algunos ejemplos son: motores, electroválvulas, relevadores, etc.

Los elementos de comunicación proporcionan a un sistema la capacidad de comunicarse con otros sistemas, son necesarios cuando una tarea compleja va a ser resuelta por diferentes sistemas. Entonces, un sistema complejo está compuesto por diferentes sistemas simples, cada uno con sus elementos de procesamiento, cada sistema simple o sub-sistema está orientado a resolver una etapa de la tarea compleja.

Los elementos de procesamiento son dispositivos electrónicos que determinan la funcionalidad del sistema, con el desarrollo de uno o varios procesos. Ocasionalmente a estos elementos de procesamiento se les refieren como la Tarjeta de Control de un sistema o simplemente el Controlador. El controlador recibe la información proveniente de los sensores y, considerando el estado actual que guarda el sistema, genera algunos resultados visuales, activa algún actuador o notifica sobre nuevas condiciones a otro sistema.

1.2 Controladores y Microcontroladores

El concepto de controlador ha permanecido invariable a través del tiempo, aunque su implementación física ha variado con los cambios tecnológicos. En principio, los controladores se construyeron con base en circuitos analógicos, las decisiones se tomaban con diferentes configuraciones de transistores o amplificadores operacionales. En los setentas se empleaba lógica discreta con circuitos digitales con baja o mediana escala de integración.

El primer microprocesador (4004 de Intel) fue puesto en operación en 1971, esto dio lugar al empleo de un microprocesador con sus elementos de soporte (memoria, entrada/salida, etc.) como tarjetas de control. A estas tarjetas también se les conoce como Computadoras en una Sola Tarjeta (SBC, *single board computer)*. Actualmente se han integrado todos estos elementos en un solo circuito integrado y a éste se le refiere como Unidad Micro Controladora (MCU, *Micro Controller Unit*) o simplemente microcontrolador, esta tendencia se ilustra en la figura 1.2.

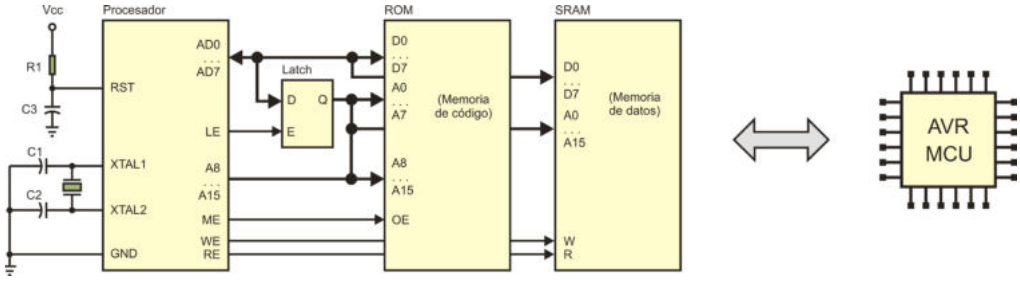


Figura 1.2 Los microcontroladores remplazan a tarjetas con varios CIs

Un microcontrolador es un Circuito Integrado con una escala de integración muy grande (VLSI[1], *very large scale integration*) que internamente contiene una Unidad Central de Procesamiento (CPU, *Cental Processing Unit*), memoria para código, memoria para datos, temporizadores, fuentes de interrupción y otros recursos necesarios para el desarrollo de aplicaciones, por lo general con un propósito específico.

Si bien, un MCU incluye prácticamente los elementos necesarios para ser considerado como una computadora en un circuito integrado, frecuentemente no es tratado como tal, ya que su uso típico consiste en el desempeño de funciones de "control" interactuando con el "mundo real" para monitorear condiciones (a través de sensores) y en respuesta a ello, encender o apagar dispositivos (por medio de actuadores).

1.3 Microprocesadores y Microcontroladores

Ocasionalmente estos dispositivos se tratan como iguales, sin embargo existen diferencias fundamentales a considerar.

Un microprocesador básicamente contiene una CPU, mientras que un microcontrolador además de la CPU contiene memoria, temporizadores, interrupciones y otros recursos útiles para el desarrollo de aplicaciones, todos estos elementos en un circuito integrado.

El microcontrolador tiene más recursos que el microprocesador, pero su CPU está limitada en términos de su capacidad de procesamiento. Las limitaciones principales son:

- **Velocidad de procesamiento:** Actualmente los microcontroladores trabajan a frecuencias máximas de 20 MHz, mientras que los microprocesadores están en el orden de GHz.

- **Capacidad de direccionamiento:** Un microcontrolador promedio dispone de 8 Kbyte para instrucciones y 1 Kbyte para datos, los microprocesadores modernos pueden direccionar hasta 1 Terabyte, espacio compartido para instrucciones y datos. Por lo que en su repertorio de instrucciones, los microprocesadores deben incluir modos de direccionamiento que les permitan este alcance.

- **Tamaño de los datos:** Los microcontroladores populares son de 8 bits y dentro de sus instrucciones incluyen algunas que permiten evaluar o modificar bits individuales. Los microprocesadores actuales trabajan con datos de 32 ó 64 bits. Sus instrucciones operan directamente sobre palabras de esta magnitud y generalmente no cuentan con instrucciones dedicadas a bits.

Estas notables diferencias entre microprocesadores y microcontroladores los enfocan a diferentes aplicaciones. Un microprocesador se utiliza como la CPU de una

1 Circuitos integrados con más de 10,000 transistores

computadora, una computadora es un **Sistema de Propósito General,** es decir, un sistema de procesamiento intensivo capaz de realizar cualquier tarea que se le solicite por programación.

Los microcontroladores están enfocados a **Sistemas de Propósito Específico,** sistemas que se crean con una funcionalidad única, la cual no va a cambiar durante su tiempo de vida útil. Por ejemplo: cajas registradoras, hornos de microondas, sistemas de control de tráfico, videojuegos, equipos de sonido, instrumentos musicales, máquinas de escribir, fotocopiadoras, etc.

Las limitaciones de los microcontroladores con respecto a los microprocesadores no son una restricción para este tipo de aplicaciones, si se consideran los siguientes aspectos:

- El tiempo de respuesta en una aplicación de propósito específico no es crítico, las operaciones para monitorear parámetros o actualizar resultados requieren de periodos en el orden de cientos de microsegundos o milisegundos, periodos que pueden conseguirse con un microcontrolador operando a unos cuantos mega hertzios.

- La aplicación es única, eso significa que la memoria de código no debe alojar otros programas que nada tengan que ver con la aplicación, como un cargador o un sistema operativo, lo cual es fundamental en un sistema de propósito general. Por lo tanto, la capacidad de memoria incluida en los microcontroladores llega a ser suficiente. Por otro lado, también hay microcontroladores con diferentes capacidades de memoria de código, que van desde 1 Kbyte hasta 256 Kbyte, el desarrollador de sistemas puede seleccionar el modelo que mejor se ajuste a sus requerimientos.

- Las aplicaciones por lo general utilizan pocas entradas, algunas son directamente de 1 bit y otras pueden ser agrupadas en un puerto de 8 bits, para su procesamiento es suficiente con una CPU que trabaje por bytes. De manera poco frecuente estas aplicaciones requieren datos de 16 bits, por ello, algunos microcontroladores incluyen instrucciones que operan directamente sobre 16 bits, o bien, puede buscarse un microcontrolador con una CPU de 16 bits. Para las salidas, es muy común que se requiera la manipulación directa de 1 bit. El encendido o apagado de un motor, un relevador, una lámpara, etc., no requiere más de 1 bit. Si fuera necesario algún tipo de variación en la intensidad de la salida, puede utilizarse modulación por ancho de pulso (PWM, *pulse width modulation*).

Puede observarse que un MCU efectivamente contiene los elementos suficientes para ser considerado como una computadora en un CI. Aunque sería una computadora con una capacidad de procesamiento limitada. No obstante, los recursos incluidos en un MCU son suficientes para aplicaciones de propósito específico, que no demanden un alto rendimiento y que no requieran manejar un conjunto masivo de datos. Aplicaciones como procesamiento de imágenes o video, están fuera del alcance de un microcontrolador.

1.4 FPGAs y Microcontroladores

Los FPGAs son dispositivos electrónicos programables que también pueden emplearse como elementos de procesamiento en sistemas electrónicos. La sigla FPGA (*Field Programmable Gate Array*) hace referencia a un Arreglo de Compuertas Programable en Campo. En la figura 1.3 se muestra la organización general de un FPGA, en donde puede notarse una disposición matricial de Bloques Lógicos Configurables (CLB, *Configurable Logic Block*) rodeados por Bloques de Entrada/ Salida (IOB, *Input/Output Block*), además de los recursos necesarios para la conexión de CLBs con IOBs o entre CLBs.

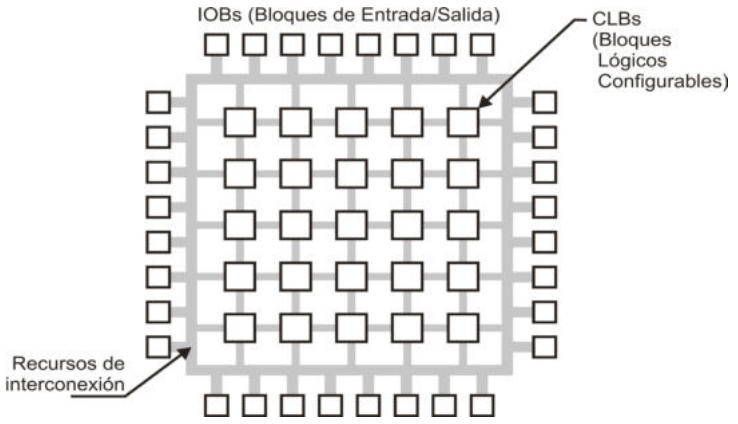


Figura 1.3 Organización típica de un FPGA

En los CLBs se pueden programar funciones lógicas combinacionales o secuenciales, con los recursos de interconexión es posible vincular diferentes bloques para construir funciones más complejas. Dependiendo del fabricante, un CLB puede contener una tabla de búsqueda (LUT, *look-up table*) o un arreglo de compuertas básicas más elementos de estado. Los IOBs proporcionan el mecanismo para que el FPGA se comunique con su entorno.

Los FPGAs se pueden programar por medio de diagramas esquemáticos, utilizando símbolos básicos y conexiones entre estos símbolos. No obstante, por la alta densidad de los dispositivos actuales, es mejor emplear un Lenguaje de Descripción de Hardware (HDL, *Hardware Description Language*). Existen diferentes HDLs, como VHDL, Verilog o ABEL.

Un aspecto común entre FPGAs y MCUs es que ambos son dispositivos configurables, con ambos se pueden construir sistemas flexibles, cuyo comportamiento se puede alterar al reprogramar al dispositivo. Con todo, debe distinguirse el papel del programa en cada caso, en un FPGA el programa determina cómo se van a conectar sus elementos internos, es decir, el programa define al hardware y de esta manera determina el comportamiento del sistema. En cambio, en un MCU el hardware es fijo y el programa establece la operación de ese hardware.

La organización de los FPGAs hace que el proceso de desarrollo de un sistema sea más complejo y tardado, con respecto al uso de microcontroladores. La ventaja de su uso es que la tecnología actual empleada en su fabricación y el hecho de trabajar directamente en hardware hacen que se alcance una velocidad de procesamiento muy alta (100 MHz o más) en relación a la velocidad de un MCU promedio.

Otra ventaja es que en un FPGA puede hacerse procesamiento concurrente real, si un sistema está organizado en forma modular, los módulos van a revisar sus entradas para generar sus salidas en forma concurrente. En un MCU el procesamiento es secuencial, aunque la inclusión de múltiples recursos facilita la realización simultánea de tareas, en el momento en que éstas generan un evento que requiere atención, la atención se realiza mediante líneas de código secuenciales.

En forma práctica, siempre debería intentarse emplear un MCU como el controlador de un sistema electrónico, si se requiere de más velocidad o capacidad de direccionamiento, la alternativa sería un microprocesador con sus elementos de soporte. Si se va a hacer un procesamiento aritmético intensivo, podría optarse por un procesador digital de señales (DSP, *Digital Signal Processor*), el cual es un circuito integrado que contiene un microprocesador más elementos de hardware enfocados a operaciones aritméticas, como sumadores y multiplicadores. Y sólo en aquellos casos donde se requiera de un hardware especializado, a la medida del sistema, que trabaje a altas velocidades y con módulos concurrentes, la mejor opción es el uso de un FPGA.

1.5 Organización de los Microcontroladores

Existe una gama muy amplia de fabricantes de microcontroladores y cada fabricante maneja diferentes familias con una variedad de modelos, a pesar de ello, hay bloques que son comunes a la mayoría de modelos, en la figura 1.4 se muestra la organización típica de un microcontrolador y en los siguientes apartados se describen sus bloques internos.

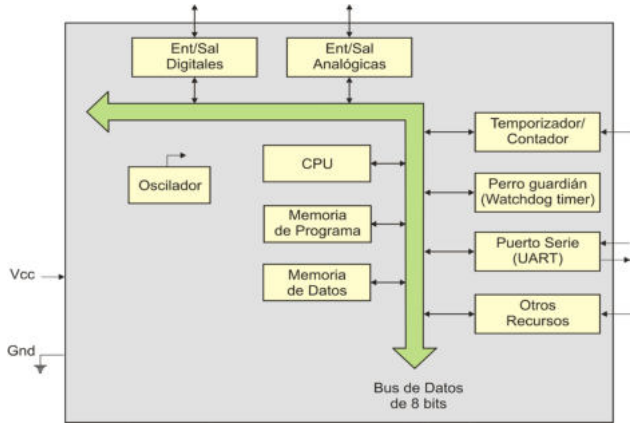


Figura 1.4 Organización típica de un Microcontrolador

1.5.1 La Unidad Central de Procesamiento (CPU)

Este bloque administra todas las actividades en el sistema y ejecuta todas las operaciones sobre los datos. Esto mediante la ejecución de las instrucciones ubicadas en la memoria de código, con las cuales se determina el comportamiento del sistema. Un **programa** se define como una serie de instrucciones, combinada para realizar algún trabajo específico. El grado en el cual los trabajos son realizados eficiente y correctamente depende muchas veces del software y no de qué tan sofisticada es la CPU.

El trabajo de la CPU puede resumirse en tres tareas fundamentales:

a) Captura de una Instrucción.

b) Decodificación de la misma.

c) Ejecución.

Trabajo que realiza a altas velocidades, por lo que el usuario observa el efecto de un programa completo y no de instrucciones individuales.

Cada procesador tiene su propio repertorio de instrucciones. Si un grupo de computadoras o microcontroladores comparten el mismo repertorio de instrucciones, pero los elementos del grupo difieren en recursos, costo y rendimiento, entonces este grupo forma una familia de computadoras.

Los repertorios de instrucciones difieren entre microcontroladores o microprocesadores, no obstante, existen algunos grupos de instrucciones que son comunes a la mayoría de este tipo de dispositivos. Estos grupos incluyen:

a) Aritméticas: suma, resta, producto, división, etc.

b) Lógicas: AND, OR, NOT, etc.

c) Transferencias de datos.

d) Bifurcaciones o saltos (condicionales o incondicionales).

Una computadora es un sistema originalmente planeado para procesamiento de datos, por lo que podría pensarse que las instrucciones de mayor uso son aritméticas o lógicas, sin embargo, actualmente las computadoras han ampliado tanto su campo de acción que las aplicaciones comunes hacen un uso exhaustivo de transferencias de datos. El ejemplo típico es el procesador de palabras, el cual transfiere datos del espacio disponible para entradas y salidas, a memoria principal y a memoria de video, cuando se respalda un documento, la información es transferida de memoria principal a memoria secundaria.

Una instrucción es una cadena de 1's y 0's que la computadora reconoce e interpreta, en esa cadena existen diferentes grupos de bits que se conocen como campos de la instrucción. Una instrucción incluye un campo para el **código de operación** (**opcode**), éste determina la operación a realizar, y típicamente uno o dos campos para los operandos, que corresponden a los datos sobre los cuales se aplica la operación.

Las CPUs se clasifican como CISC o RISC, esto de acuerdo con su organización interna. Con CISC se hace referencia a computadoras con un Repertorio de Instrucciones Complejo (CISC, *Complex Instruction Set Computers*) y RISC es para referir a computadoras con un Repertorio de Instrucciones Reducido (RISC, *Reduced Instruction Set Computers*).

Las primeras computadoras se construyeron con la filosofía CISC, en la cual se buscaba que el programador escribiera programas compactos, por lo tanto, cada instrucción requería de un hardware complejo. Esto afecta el rendimiento de las computadoras dado que se requiere de un ciclo de reloj duradero o de varios ciclos de reloj para la ejecución de una instrucción. La filosofía RISC es opuesta, busca que el hardware sea simple y que resuelva pocas instrucciones, con ello el hardware puede trabajar a frecuencias mayores.

Una arquitectura RISC tiene pocas instrucciones y generalmente éstas son del mismo tamaño; en la CISC hay demasiadas instrucciones con diferentes tamaños y formatos, que pueden ocupar varios bytes, uno para el opcode y los demás para los operandos. La tarea realizada por una instrucción CISC puede requerir de varias instrucciones RISC. En contraste, el hardware de un procesador RISC es tan simple, que se puede implementar en una fracción de la superficie ocupada en un circuito integrado por un procesador CISC.

La organización de los procesadores RISC hace que, aun con tecnologías de semiconductores comparables e igual frecuencia de reloj, su capacidad de procesamiento sea de dos a cuatro veces mayor que la de un CISC, esto porque permite la aplicación de técnicas como la segmentación, mediante la cual es posible solapar diferentes instrucciones en diferentes etapas del procesador, por ejemplo, mientras una instrucción se está ejecutando, otra puede estar en proceso de decodificación y la siguiente en la etapa de captura. El número de instrucciones que simultáneamente están en el procesador depende del número de etapas de segmentación incluidas.

1.5.1.1 Organización de una CPU

A pesar de que existe una diversidad de fabricantes de procesadores, hay elementos que son comunes a todos ellos. En la figura 1.5 se muestran los bloques típicos de una CPU, los cuales se pueden clasificar en dos grupos: el Camino de Datos y la Unidad de Decodificación y Control. El Camino de Datos involucra los elementos en donde puede fluir la información cuando se ejecuta una instrucción y la Unidad de Decodificación y Control determina qué elementos se activan dentro del Camino de los Datos para la correcta ejecución de una instrucción.

El Contador de Programa (PC, *Program Counter*), el Registro de Instrucción (IR, *Instruction Register*) y el Apuntador de Pila (SP, *Stack Pointer*), son registros con una función específica en una CPU.

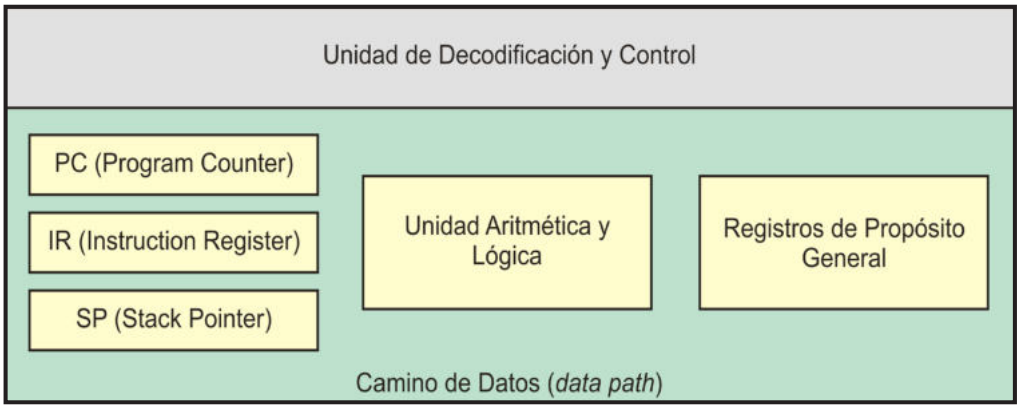


Figura 1.5 Elementos comunes en una CPU

El PC contiene la dirección de la instrucción que se va a ejecutar en un instante de tiempo determinado y mientras esa instrucción se ejecuta, el PC automáticamente actualiza su valor para apuntar a la siguiente instrucción a ejecutar.

El registro IR contiene la cadena de bits que conforman a la instrucción bajo ejecución, de esa cadena, la unidad de control considera el campo del opcode para determinar la activación de las señales en los demás elementos en la CPU.

El SP contiene la dirección del tope de la pila, que es un espacio de almacenamiento utilizado durante la invocación de rutinas. La llamada a una rutina requiere que el valor del PC sea respaldado en la pila, con ello, el SP se ajusta automáticamente al nuevo tope. Cuando la rutina termina, se extrae el valor del tope de la pila y con éste se reemplaza al PC, para que el programa continúe con la instrucción posterior al llamado de la rutina, esto también requiere un ajuste del SP. Además de las llamadas a rutinas, algunos procesadores incluyen instrucciones para hacer respaldos (*push*) y recuperaciones (*pop*) en forma explícita. Instrucciones que también producen cambios automáticos en el SP.

La Unidad Aritmética y Lógica es el bloque que se encarga de realizar las operaciones aritméticas y lógicas con los datos, no obstante, en ocasiones también opera sobre direcciones para calcular el destino de un salto o la ubicación de una localidad a la que se va a tener acceso para una transferencia de memoria a registro o viceversa.

Los registros de propósito general son los elementos más rápidos para el almacenamiento de variables. Dado que el número de registros en una CPU es limitado, si éste no es suficiente para todas las variables requeridas, debe utilizarse la memoria de datos para su almacenamiento.

1.5.1.2 Tareas de la CPU

Con cada instrucción, la CPU realiza tres tareas fundamentales: Captura, Decodificación y Ejecución.

La **Captura de una Instrucción** es una tarea que involucra los siguientes pasos:

a. El contenido del PC se coloca en el bus de direcciones.

b. La CPU genera una señal de control, para habilitar la lectura de memoria de código.

c. Una instrucción se lee de la memoria de código y se coloca en el bus de datos.

d. La instrucción se toma del bus de datos y se coloca en el IR.

e. El PC es preparado para la siguiente instrucción.

Una vez que la instrucción está en el IR, el procesador continúa con la **decodificación** de la instrucción. Decodificar una instrucción consiste en descifrar el opcode para generar las señales de control necesarias, dependiendo del tipo de instrucción.

Finalmente, la tercera de las tareas de la CPU es la **Ejecución**. Ejecutar una instrucción puede involucrar: habilitar a la ALU para que genere algún resultado, cargar un dato de memoria a un registro, almacenar el contenido de un registro en memoria o modificar el valor del PC, según las señales generadas por la unidad de decodificación y control.

1.5.2 Sistema de Memoria

Una computadora (y por lo tanto, también un microcontrolador) debe contar con espacios de memoria para almacenar los programas (código) y los datos. En relación a cómo se organizan estos espacios se tienen dos modelos de computadoras, un modelo en donde el código y datos comparten el mismo espacio de memoria y el otro en donde se tienen memorias separadas, una para código y otra para datos, éstos se ilustran en la figura 1.6.

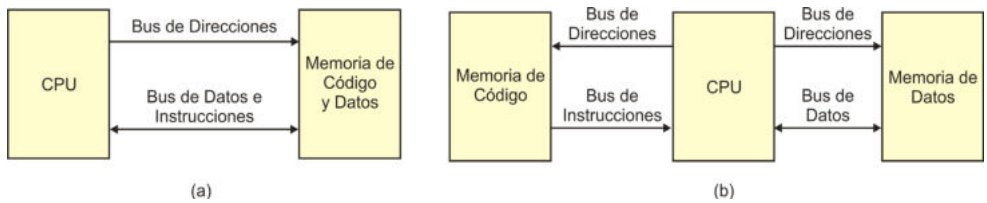


Figura 1.6 Modelos de computadoras respecto a la organización de la memoria (a) Arquitectura von Neumann y (b) Arquitectura Harvard

John von Neumann[2] propuso el concepto de programa almacenado, el cual establece que las instrucciones se lleven a memoria como si fueran datos, para que posteriormente se ejecuten sin tener que escribirlas nuevamente, por lo tanto, se requiere de un solo espacio de memoria para almacenar instrucciones y datos. Este concepto fue primeramente aplicado en la Computadora Automática Electrónica de Variable Discreta (EDVAC, *Electronic Discrete-Variable Automatic Computer*), desarrollada por Von Neumann, Eckert y Mauchly. Actualmente ha sido adoptado por los diseñadores de computadoras porque proporciona flexibilidad a los sistemas. Si una computadora se basa en este concepto, se dice que tiene una Arquitectura tipo Von Neumann.

Mientras la tendencia natural para los diseñadores de computadoras fue adoptar el concepto de programa almacenado, en la Universidad de Harvard desarrollaron la Mark I, la cual almacenaba instrucciones y datos en cintas perforadas, pero incluía interruptores rotatorios de 10 posiciones para el manejo de registros. Actualmente, si una computadora tiene un espacio para el almacenamiento de código físicamente separado del espacio de almacenamiento de datos, se dice que tiene una Arquitectura Harvard.

La mayoría de microcontroladores utilizan una Arquitectura Harvard. En la memoria de código se alojan las instrucciones que conforman el programa y algunas constantes. Algunos microcontroladores, además de su memoria interna, tienen la capacidad de direccionar memoria externa de código, para soportar programas con una cantidad grande de instrucciones.

Usualmente la memoria de programa es no volátil y suele ser del tipo EPROM, EEPROM, Flash, programable una sola vez (OTP, *one-time programmable*) o ROM enmascarable. Los primeros 3 tipos son adecuados durante las etapas de prototipado, la memoria OTP es conveniente si se va a hacer una producción de pocos volúmenes de un sistema y la ROM enmascarable es lo más acertado para una producción masiva.

Para la memoria de datos, los microcontroladores pueden contener RAM o EEPROM, la RAM se utiliza para almacenar algunas variables y contener una pila. La EEPROM es para almacenar aquellos datos que se quieran conservar aun en ausencia de energía. Todos los microcontroladores tienen memoria interna de datos, en diferentes magnitudes, algunos además cuentan con la capacidad de expansión usando una memoria externa.

1.5.3 Oscilador

La CPU va tomando las instrucciones de la memoria de programa para su posterior ejecución a cierta frecuencia. Esta frecuencia está determinada por el circuito de oscilación, el cual genera la frecuencia de trabajo a partir de elementos externos como un circuito RC, un resonador cerámico o un cristal de cuarzo, aunque algunos

2 **John von Neumann**, (28 de diciembre de 1903 - 8 de febrero de 1957) Matemático húngaro-estadounidense, doctorado por la Universidad de Budapest a los 23 años. Realizó contribuciones importantes en física cuántica, análisis funcional, teoría de conjuntos, informática, economía, análisis numérico, estadística y muchos otros campos.

microcontroladores ya incluyen un oscilador RC calibrado interno. Tan pronto como se suministra la alimentación eléctrica, el oscilador empieza con su operación.

1.5.4 Temporizador/Contador

El Temporizador/Contador (*timer/counter*) es un recurso con una doble función, como temporizador se utiliza para manejar intervalos de tiempo y como contador es la base para programar alguna tarea cada que ocurra una cantidad predeterminada de eventos externos al microcontrolador.

Se compone de un registro de n-bits que se incrementa en cada ciclo de reloj o cuando ocurra un evento externo, según el modo de operación. Cuando ocurre un desbordamiento del registro genera alguna señalización, poniendo en alto una bandera, para indicar a la CPU que ha pasado un intervalo de tiempo o que ha ocurrido un número esperado de eventos. El desbordamiento ocurre cuando el registro alcanza su valor máximo (todos los bits del registro en 1) y reinicia la cuenta (todos los bits en 0's). La organización básica de un temporizador se muestra en la figura 1.7.

El registro del temporizador tiene un comportamiento ascendente y puede ser precargado para reducir el número de eventos a contar. La CPU puede emplear su tiempo de procesamiento en otras tareas, dentro de las cuales debe reservar un espacio para monitorear la bandera, o bien, configurar al recurso para que genere una interrupción.

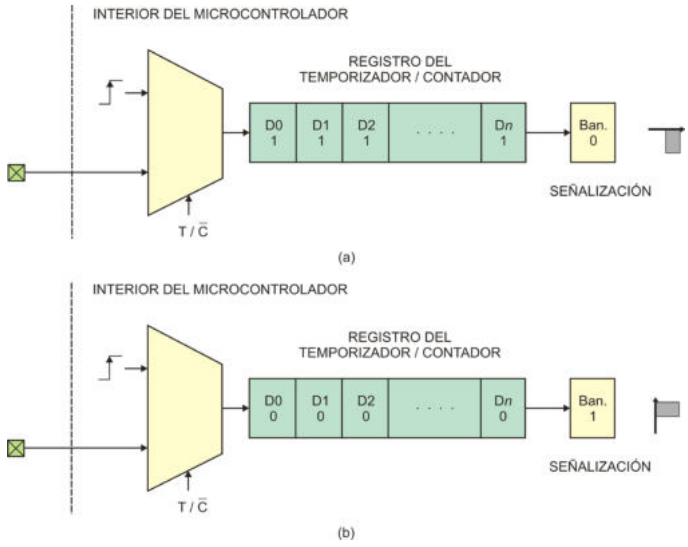


Figura 1.7 Organización básica de un Temporizador/Contador, en (a) el registro ha alcanzado su valor máximo y en (b) al reiniciar la cuenta se genera una señalización

En algunos microcontroladores la entrada del temporizador es precedida por un pre-escalador, el cual básicamente es un divisor de frecuencia configurable, con el que se puede contar un número más grande de eventos y por lo tanto, alcanzar intervalos de tiempo mayores.

En el caso de los microcontroladores AVR, además de los eventos de desbordamientos se pueden manejar eventos de coincidencias por comparación y de captura.

1.5.5 Perro Guardián (WDT, *watchdog timer*)

El WDT (*watchdog timer*) también es un temporizador y por lo tanto, también se compone de un registro de n-bits, sólo que cuando el WDT desborda ocasiona un reinicio del sistema (*reset*). El objetivo del WDT es evitar que el microcontrolador se cicle en estados no contemplados, lo cual llega a ser bastante útil en sistemas autónomos. Un microcontrolador puede ciclarse en estados no deseados ante situaciones inesperadas, como variaciones en la fuente de alimentación, desconexión repentina de un periférico, etc.

Algunos microcontroladores que poseen WDT requieren de su activación en el momento en que se programa al dispositivo, otros permiten activarlo o desactivarlo dentro del programa de aplicación, siempre que se siga alguna secuencia de seguridad para evitar activaciones no deseadas. Si se utiliza al WDT, en posiciones estratégicas del programa principal deben incluirse instrucciones que lo reinicien para evitar su desbordamiento.

1.5.6 Puerto Serie

La mayoría de microcontroladores cuentan con un receptor/transmisor universal asíncrono (UART, *Universal Asynchronous Receiver Transceiver*), para una comunicación serial con dispositivos o sistemas externos, bajo protocolos y razones de transmisiones estándares. La comunicación serial puede ser de dos tipos:

- Síncrona: Además de la línea de datos se utiliza una línea de reloj.
- Asíncrona: Sólo hay líneas para los datos, el transmisor y el receptor se deben configurar con la misma velocidad de transferencia (bits/segundo, *Baud Rate*), además de definir el mismo formato para cada trama.

La comunicación serial es bastante útil porque sólo requiere de un alambre o línea de conexión y tiene un alcance mucho mayor que una transmisión paralela (de varios bits). El hardware para la comunicación serial básicamente consiste en una conversión de paralelo a serie, durante una transmisión, o de serie a paralelo, cuando se hace una recepción. Puede realizarse entre un microcontrolador con una computadora, entre

microcontroladores o un microcontrolador con otros sistemas que incluyan un puerto de comunicación serial. En la figura 1.8 se muestra la diferencia entre una comunicación síncrona y asíncrona.

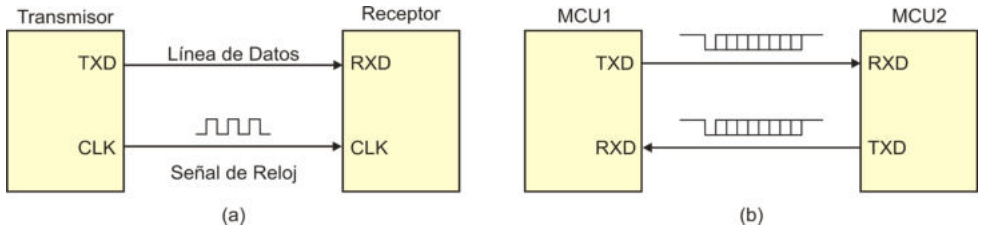


Figura 1.8 Comunicación Serial (a) Síncrona y (b) Asíncrona

1.5.7 Entradas/Salidas Digitales

Los microcontroladores incluyen puertos de Entradas/Salidas digitales para intercambiar datos con el mundo exterior. A diferencia de un puerto serie, en donde se transfiere un bit a la vez, en los puertos digitales es posible realizar un intercambio de bytes.

Todos los microcontroladores tienen puertos digitales, aunque el número de puertos o el número de bits por puerto puede variar entre dispositivos. Como entradas se utilizan para el monitoreo de dispositivos digitales como botones, interruptores, teclados, sensores con salida a relevador, etc., y como salidas para el manejo de LEDs, displays de 7 segmentos, activación de motores, LCDs, etc.

1.5.8 Entradas/Salidas Analógicas

Para entradas analógicas algunos microcontroladores incorporan Convertidores Analógico a Digital (ADC, *Analogic-Digital Converter*) o comparadores analógicos. Éstos son muy útiles porque sin elementos externos, permiten obtener información analógica del exterior, para monitorear parámetros como temperatura, velocidad, humedad, etc.

Para salidas analógicas podría pensarse en un Convertidor Digital a Analógico (DAC, *Digital-Analogic Converter*) pero no es común que se incluya en un microcontrolador. Para solventar esta carencia, algunos microcontroladores incluyen salidas con Modulación por Ancho de Pulso (PWM, *Pulse Width Modulation*), por medio de ellas, con pocos elementos externos es posible generar una señal analógica en una salida digital.

1.6 Clasificación de los Microcontroladores

Existen diferentes formas de clasificar a los microcontroladores y no son excluyentes unas de otras. A continuación se describen las formas típicas de clasificaciones.

Por la arquitectura de la CPU, los microcontroladores pueden clasificarse como **RISC** o **CISC**. Prácticamente todos los nuevos microcontroladores son **RISC**.

En relacion al tamaño de los datos, se tienen microcontroladores de **4**, **8**, **16** y hasta **32** bits. Por el tamaño de los datos debe entenderse el tamaño de los registros de trabajo y por lo tanto, corresponde con el número de bits de los operandos en la ALU, éste generalmente difiere del tamaño de las instrucciones que no es un parámetro para la clasificación.

Tomando como base la organización y el acceso a la memoria de código y datos, se tienen 2 modelos: Arquitectura **Von Neumann** y Arquitectura **Harvard**.

Considerando la memoria y sus capacidades de expansión, cuando un microcontrolador está acondicionado para tener acceso a memoria externa, se dice que tiene una arquitectura **abierta**, en caso contrario, su arquitectura es **cerrada**. Con una arquitectura **abierta**, además de manejar memoria externa, es posible manipular periféricos externos, mapeándolos en memoria de datos y reservándoles un espacio de direcciones para su manejo.

La última clasificación tiene que ver con la forma en que los datos son almacenados y manipulados internamente dentro de la CPU. Los microcontroladores manipulan datos por medio de un programa de usuario, en este esquema de clasificación se distingue a las arquitecturas de acuerdo a como la CPU ejecuta las instrucciones y tiene acceso a los datos que involucra cada instrucción. Bajo este esquema, se tienen los siguientes cuatro modelos: **Pila**, **Acumulador**, **Registro-Memoria** y **Registro-Registro**.

En una arquitectura tipo **Pila**, una pila es la base para el procesamiento, los datos a operar deben ingresarse en la pila, las operaciones se realizan sobre los últimos datos de la pila y dejan el resultado en el tope de la pila. Por ejemplo, para realizar la operación de alto nivel:

$$A = B - C$$

Suponiendo que A, B y C son variables almacenadas en memoria, se tendrían las siguientes instrucciones:

```
PUSH   B      ; Ingresa la variable B en la pila
PUSH   C      ; Ingresa la variable C en la pila
SUB           ; Resta los datos del tope de la pila
              ; el resultado queda en la pila
POP    A      ; Extrae el tope de la pila y lo almacena en A
```

Una arquitectura tipo **Acumulador** basa su operación en un registro con el mismo nombre. El Acumulador es el registro de trabajo, las instrucciones que únicamente

requieren un operando se aplican sobre el acumulador y en la ALU, necesariamente un operando debe ser el acumulador.

Si la misma operación de resta se va a realizar bajo una arquitectura tipo acumulador, las instrucciones resultantes son las siguientes (Acc representa al acumulador):

```
MOV    Acc, B  ; Transfiere la variable B al acumulador
SUB    Acc, C  ; Resta C del acumulador y ahí mismo queda el resultado
MOV    A, Acc  ; Transfiere el acumulador a la variable A
```

Una arquitectura del tipo **Registro-Memoria** implica que el procesador está acondicionado para que uno de los operandos de la ALU esté en memoria, mientras el otro debe estar en uno de los registros de trabajo. La operación bajo consideración se realizaría con las siguientes instrucciones:

```
LOAD   R1, B  ; Carga la variable B en el registro R1
SUB    R1, C  ; Resta C que está en memoria, del registro R1
MOV    A, R1  ; Almacena R1 en la variable A, que está en memoria
```

Finalmente, en una arquitectura del tipo **Registro-Registro,** los dos operandos que llegan a la ALU deben estar en los registros de propósito general. Las arquitecturas de este estilo también son conocidas como Arquitecturas tipo **Carga-Almacenamiento**, esto porque cuando se van a operar variables que están en memoria, primeramente deben ser cargadas en registros, el resultado queda en un registro y, por lo tanto, se requiere de un almacenamiento para llevarlo a una variable de memoria. Para el mismo ejemplo se tendrían las instrucciones siguientes:

```
LOAD   R1, B        ; Carga la variable B en un registro
LOAD   R2, C        ; Carga la variable C en otro registro
SUB    R1, R2       ; La ALU opera sobre registros
MOV    A, R1        ; Almacena el resultado en la variable A
```

Los microcontroladores bajo estudio son el ATMega8 y el ATMega16, estos microcontroladores son RISC, de 8 bits, con una Arquitectura tipo Harvard que es cerrada, y su operación es del tipo Registro-Registro.

1.7 Criterios para la Selección de los Elementos de Procesamiento

Existe una gama muy amplia de fabricantes de microprocesadores o microcontroladores, cada fabricante ha desarrollado diferentes familias y en cada familia se tiene un número variable de dispositivos, es por esto que resulta complejo determinar cuál sería el dispositivo adecuado para alguna aplicación. A continuación se listan algunos criterios que pueden tomarse en consideración.

La primera consideración son las **prestaciones** del dispositivo, las cuales se deben vincular con los requerimientos de procesamiento que debe realizar el sistema.

Considerando la capacidad de procesamiento, los dispositivos se pueden agrupar en 3 clases diferentes:

- **Gama baja**: Procesadores de 4, 8 y 16 bits. Dedicados fundamentalmente a tareas de control (electrodomésticos, cabinas telefónicas, tarjetas inteligentes, algunos periféricos de computadoras, etc.). Generalmente se emplean microcontroladores.

- **Gama media:** Dispositivos de 16 y 32 bits. Para tareas de control con cierto grado de procesamiento (control en automóvil, teléfonos móviles, PDA, etc.). En este caso puede utilizarse un microcontrolador o microprocesador, además de periféricos y memoria externa.

- **Gama alta:** 32, 64 y 128 bits. Fundamentalmente para procesamiento (computadoras, videoconsolas, etc.). Casi en su totalidad son microprocesadores más circuitería periférica y memoria.

Referente a la **tecnología** de fabricación, debe considerarse:

- El **consumo de energía**, algunos dispositivos cuentan con modos de ahorro de energía que les permiten un consumo de algunos micro-Watts, mientras que otros llegan a consumir algunas décimas de Watts.

- Otro aspecto es el **voltaje de alimentación**, algunos dispositivos puede operar con 5 V, 3.3 V, 2.5 V o 1.5 V, éste es fundamental si el sistema va a ser alimentado con baterías.

- La **frecuencia** de operación también es un factor bajo consideración, dado que los dispositivos pueden operar desde KHz a GHz, si un microcontrolador puede trabajar en un rango amplio de frecuencias, es conveniente operarlo en la frecuencia más baja que le permita un desempeño correcto en la aplicación, esto porque a menor velocidad de procesamiento, el consumo de energía es menor.

El siguiente criterio bajo consideración es el **costo**, este aspecto es esencial una vez que se ha comprobado que el dispositivo cumple con las prestaciones requeridas, es decir, después de un análisis del rendimiento del hardware y software, considerando el uso medio o el peor de los casos. El costo de un microcontrolador o microprocesador puede variar de 2 a 1000 dólares.

Un aspecto muy importante son las **herramientas de desarrollo**, debe considerarse su precio, complejidad y prestaciones. Actualmente muchos fabricantes de microcontroladores dejan disponible de manera gratuita alguna suite de desarrollo, buscando ponerse a la vanguardia entre los desarrolladores de sistemas.

Un factor importante es la **experiencia** del desarrollador, muchas veces se prefiere acondicionar un microcontrolador conocido para incluir un recurso externo, antes de aprender a manejar un nuevo dispositivo que ya tiene al recurso empotrado.

Una vez que se domina un microcontrolador no es complejo manejar uno diferente, por lo que este hecho puede deberse a la carencia de dispositivos o herramientas, o bien a

la ausencia de soporte técnico. Si no se tienen problemas de disponibilidad y soporte, la emigración a dispositivos con un mayor número de recursos es lo más adecuado, dado que con una sola compuerta externa que se ahorre en un sistema, puede representar grandes beneficios económicos si se considera una producción masiva.

La experiencia en el manejo de un dispositivo va a reflejarse en el **tiempo de desarrollo** de un producto, la rápida evolución de la tecnología requiere de tiempos de desarrollo cada vez más cortos para mantener competitividad. Por ejemplo, si se desea desarrollar un decodificador para un receptor satelital con base en un microprocesador y se invierte un tiempo aproximado de dos años, para cuando el producto sea puesto en el mercado, tal vez existan versiones de procesadores que trabajen al doble de velocidad y que consuman la mitad de la potencia, eso implicaría que el producto ya no sería competitivo. Por lo tanto, los retrasos de la puesta en el mercado de los nuevos productos pueden producir grandes pérdidas.

El último criterio bajo consideración es la **compatibilidad** entre los dispositivos de una misma familia, éste es importante cuando se proyecta el desarrollo de diferentes versiones de un producto. En una familia, todos los dispositivos manejan el mismo repertorio de instrucciones, pero se distinguen por los recursos de hardware incluidos en cada miembro, por lo que el desarrollador debe seleccionar el más adecuado ante estas diferentes versiones del producto, las cuales pueden ir desde dispositivos con características muy limitadas hasta las versiones altamente sofisticadas.

La **compatibilidad** implica que se requiera de pocos ajustes en hardware y software, para obtener una versión mejorada de un producto, empleando un microcontrolador con mayores prestaciones. Este factor es importante si se toma en cuenta que la vida media de los productos es cada vez más corta, actualmente se llega a considerar como obsoleto a un sistema después que ha trabajado un par de años. Esto resalta la conveniencia de utilizar microcontroladores que pertenecen a familias con una gama amplia de dispositivos.

1.8 Ejercicios

1. Explique la importancia de los sistemas electrónicos.
2. ¿Qué es un microcontrolador?
3. Exprese las diferencias principales entre un microcontrolador y un microprocesador.
4. Explique a qué tipo de aplicaciones se enfocan los microcontroladores y dé ejemplos de ellas.
5. Justifique en qué situaciones sería conveniente o necesario el uso de un FPGA en lugar de un MCU.
6. Muestre un diagrama con la organización típica de un microcontrolador.
7. Describa el papel de una CPU en un microcontrolador (o computadora) y explique las tareas que realiza con cada instrucción.

8. Indique el objetivo de los registros de propósito específico comúnmente encontrados en una CPU:
    a. Program Counter (PC)
    b. Instruction Register (IR)
    c. Stack Pointer (SP)
9. Liste los grupos de instrucciones típicos que maneja una CPU.
10. En qué difiere una arquitectura Harvard de una arquitectura basada en el modelo de Neumann.
11. Explique las diferencias entre una arquitectura RISC y una arquitectura CISC.
12. Indique los tipos de memoria utilizados por los microcontroladores para el almacenamiento de instrucciones y para el almacenamiento de datos.
13. Explique la función de los siguientes recursos en un microcontrolador:
    a. Oscilador Interno
    b. Temporizador (*timer*)
    c. Perro guardián (*watchdog timer*)
    d. Puerto Serie
    e. Entradas y salidas digitales
    f. Entradas y salidas analógicas
14. Muestre cómo se realizaría la suma: A = B + C + D, en una arquitectura:
    a. Tipo Pila
    b. Tipo Acumulador
    c. Tipo Memoria-Registro
    d. Tipo Registro-Registro (Carga-Almacenamiento)

    Suponiendo que A, B, C y D son variables ubicadas en memoria de datos.
15. En orden de consideración, explique tres criterios que tomaría en cuenta al seleccionar un microcontrolador para una aplicación.

2. Organización de los Microntroladores AVR de ATMEL

Los microcontroladores AVR incluyen un procesador **RISC** de **8 bits**, su arquitectura es del tipo **Harvard** y sus operaciones se realizan bajo un esquema **Registro-Registro**.

Este capítulo hace referencia al hardware de los microcontroladores AVR, específicamente del ATMega8 y ATMega16, se describe su organización interna y sus características de funcionamiento.

2.1 Características Generales

Los microcontroladores AVR se basan en un núcleo cuya arquitectura fue diseñada por Alf-Egil Bogen y Vegard Wollan, estudiantes del Instituto Noruego de Tecnología, arquitectura que posteriormente fue refinada y desarrollada por la firma Atmel. El término AVR no tiene un significado implícito, a veces se considera como un acrónimo en el que se involucra a los diseñadores del núcleo, es decir AVR puede corresponder con Alf-Vegard-RISC.

El núcleo es compartido por más de 50 miembros de la familia, proporcionando una amplia escalabilidad entre elementos con diferentes recursos. En la figura 2.1 se ilustra este hecho, los miembros con menos recursos caen en la gama Tiny, los miembros con más recursos pertenecen a la categoría Mega, además de que se cuenta con miembros orientados para aplicaciones específicas.

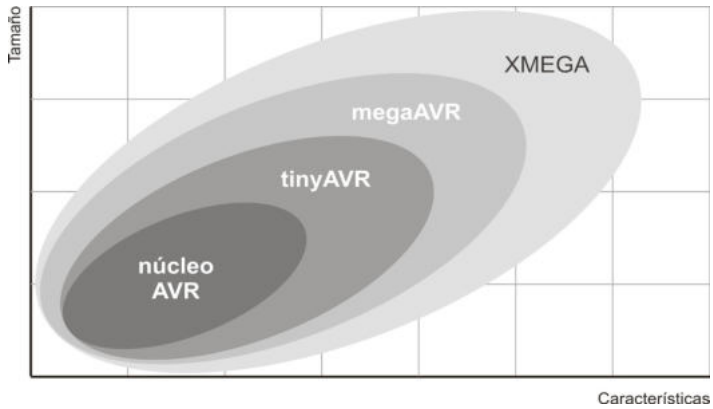


Figura 2.1 Escalabilidad entre dispositivos que comparten el núcleo

En concreto, este libro se enfoca a los dispositivos **ATMega8** y **ATMega16**, para ambos, sus principales características técnicas son:

- **Memoria de código:** 8 Kbyte (ATMega8) o 16 Kbyte (ATMega16) de memoria flash.
- **Memoria de datos:** 1 Kbyte de SRAM y 512 bytes de EEPROM.
- **Terminales para entrada/salida:** 23 (ATMega8) o 32 (ATMega16).
- **Frecuencia máxima de trabajo:** 16 MHz.
- **Voltaje de alimentación:** de 2.7 a 5.5 Volts.
- **Temporizadores:** 2 de 8 bits y 1 de 16 bits.
- **Canales PWM:** 3 (ATMega8) o 4 (ATMega16).
- **Fuentes de interrupción:** 19 (ATMega8) o 21 (ATMega16).
- **Interrupciones externas:** 2 (ATMega8) o 3 (ATMega16).
- **Canales de conversión Analógico/Digital:** 8 de 10 bits.
- Reloj de tiempo real.
- Interfaz SPI Maestro/Esclavo.
- Transmisor/Receptor Universal Síncrono/Asíncrono (USART).
- Interfaz serial de dos hilos.
- Programación "*In System*".
- Oscilador interno configurable.
- *Watchdog timer*.

Comercialmente el ATMega8 se encuentra disponible en encapsulados PDIP de 28 terminales o bien, encapsulados TQFP o MLF de 32 terminales. Para el ATMega16 se tiene una versión en PDIP de 40 terminales y otras con encapsulados TQFP, QFN o MLF de 44 terminales. Las versiones en PDIP son las más convenientes durante el desarrollo de prototipos por su compatibilidad con las tablillas de pruebas (*protoboard*). En la figura 2.2 se muestra el aspecto externo para ambos dispositivos, considerando un encapsulado PDIP.

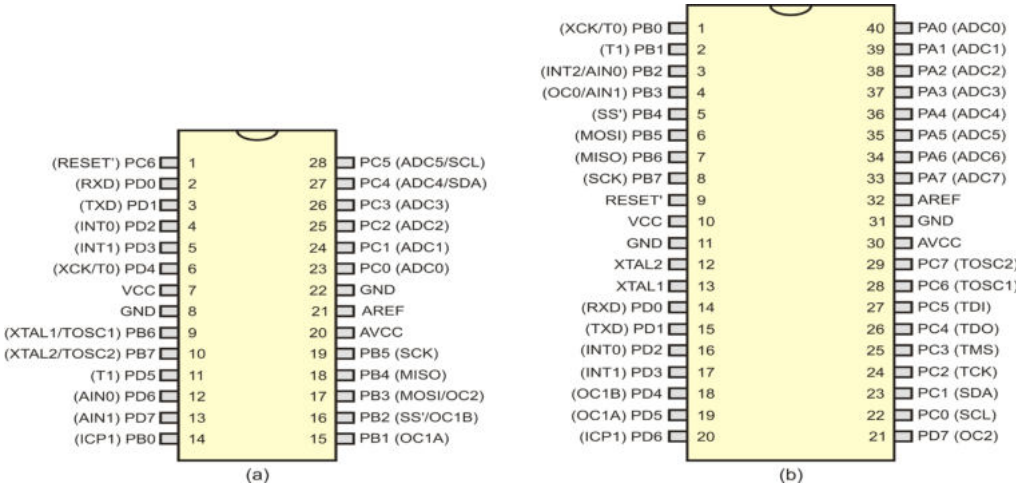


Figura 2.2 Aspecto externo de (a) un ATMega8 y (b) un ATMega16

El ATMega8 incluye 3 puertos, 2 de 8 bits y 1 de 7 bits; mientras que el ATMega16 contiene 4 puertos, todos de 8 bits. También se observa que todas las terminales incluyen una doble o triple función, esto significa que además de utilizarse como entrada o salida de propósito general, las terminales pueden emplearse con un propósito específico, relacionado con alguno de los recursos del microcontrolador.

2.2 El Núcleo AVR

La organización interna de los microcontroladores bajo estudio se fundamenta en el núcleo AVR, el núcleo es la unidad central de procesamiento (CPU), es decir, es el hardware encargado de la captura, decodificación y ejecución de instrucciones, su organización se muestra en la figura 2.3. En torno al núcleo se encuentra un bus de 8 bits al cual están conectados los diferentes recursos del microcontrolador, estos recursos pueden diferir entre dispositivos.

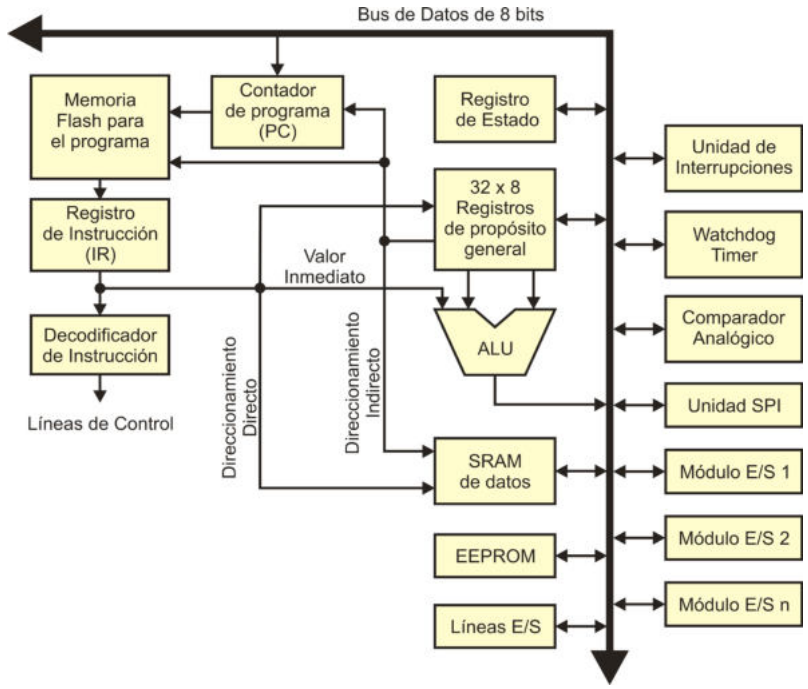


Figura 2.3 Diagrama a bloques del núcleo AVR

La principal función de la CPU es asegurar la correcta ejecución de programas. La CPU debe tener acceso a los datos, realizar cálculos, controlar periféricos y manejar interrupciones.

Para maximizar el rendimiento y paralelismo, el AVR usa una arquitectura Harvard con memorias y buses separados para el programa y los datos. Esto se observa en la figura 2.3, el programa se ubica en la memoria flash y los datos están en 3 espacios diferentes: En el archivo de registros (32 registros de 8 bits), en la SRAM y en la EEPROM.

De la memoria flash se obtiene cada instrucción del programa y se coloca en el registro de instrucción (IR) para su decodificación y ejecución. La memoria flash es direccionada por el contador de programa (PC) o bien, por uno de los registros de propósito general. El PC es en sí el registro que indica la ubicación de la instrucción a ejecutar, sin embargo, es posible que un registro de propósito general proporcione esta dirección a modo de que funcione como apuntador y se haga un acceso utilizando direccionamiento indirecto.

La ALU soporta operaciones aritméticas y lógicas entre los 32 registros de propósito general o entre un registro y una constante, para cualquier operación, al menos uno de los operandos es uno de estos registros. Los 32 registros son la base para el procesamiento de datos porque la arquitectura es del tipo registro-registro, esto implica que si un dato de SRAM o de EEPROM va a ser modificado, primero debe ser llevado a cualquiera de los 32 registros de 8 bits, dado que todos tienen la misma jerarquía.

El Registro de Estado principalmente contiene banderas que se actualizan después de una operación aritmética, para reflejar información relacionada con el resultado de la operación. Las banderas posteriormente pueden ser utilizadas por diversas instrucciones para tomar decisiones.

2.2.1 Ejecución de Instrucciones

El flujo del programa por naturaleza es secuencial, con incrementos automáticos del PC. Este flujo secuencial puede ser modificado con instrucciones de saltos condicionales o incondicionales y llamadas a rutinas, éstas son instrucciones que modifican directamente al PC, permitiendo abarcar completamente el espacio de direcciones.

La CPU va a capturar las instrucciones para después ejecutarlas, su organización hace posible que este proceso se segmente en dos etapas, solapando la captura con la ejecución de instrucciones. Es decir, mientras una instrucción está siendo ejecutada, la siguiente es capturada en IR. Con ello, aunque el tiempo de ejecución por instrucción es de dos ciclos de reloj, la productividad va a ser de una instrucción por ciclo de reloj, esto se muestra en la figura 2.4. Con lo cual el rendimiento de la CPU va a ser muy aproximado a 1 MIPS[3] por cada MHz de la frecuencia del oscilador.

3 MIPS, métrica para medir el rendimiento de procesadores, significa Millones de Instrucciones por Segundo.

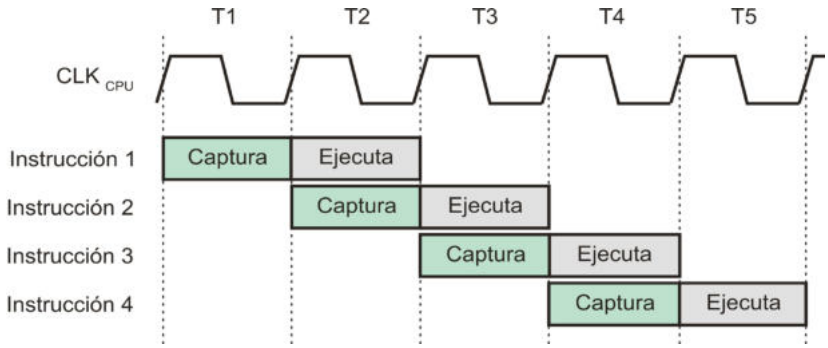


Figura 2.4 Segmentación a dos etapas realizado por el núcleo AVR

En los saltos y llamadas a rutinas no se puede anticipar la captura de la siguiente instrucción porque se ignora cuál es, por lo tanto, se pierde un ciclo de reloj. Algo similar ocurre con los accesos a memoria (cargas o almacenamientos), instrucciones que gastan un ciclo de reloj para la manipulación de direcciones, antes de hacer el acceso.

Para las instrucciones aritméticas y lógicas es suficiente con un ciclo de reloj para su ejecución (posterior a la captura), al comienzo del ciclo se capturan los operandos de los registros de propósito general, la ALU trabaja sincronizada con el flanco de bajada y prepara el resultado para que sea escrito en el siguiente flanco de subida, esto se muestra en la figura 2.5.

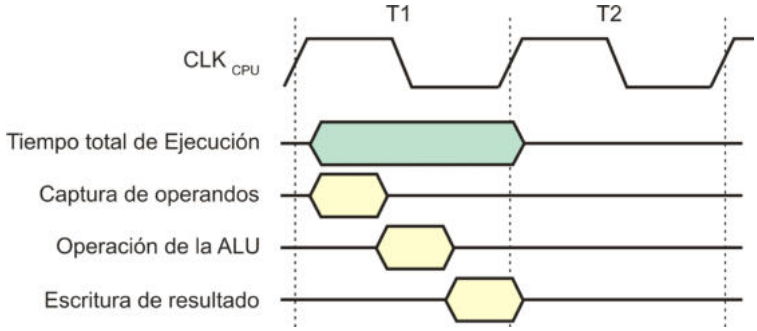


Figura 2.5 Temporización de la fase de ejecución

2.2.2 Archivo de Registros

El archivo de registros contiene 32 registros de 8 bits de propósito general, el núcleo AVR está acondicionado para tener un acceso rápido a ellos. La organización del archivo de registros se muestra en la figura 2.6, los registros se denominan R0, R1, R2, etc.; las instrucciones que operan sobre registros se ejecutan en 1 ciclo de reloj.

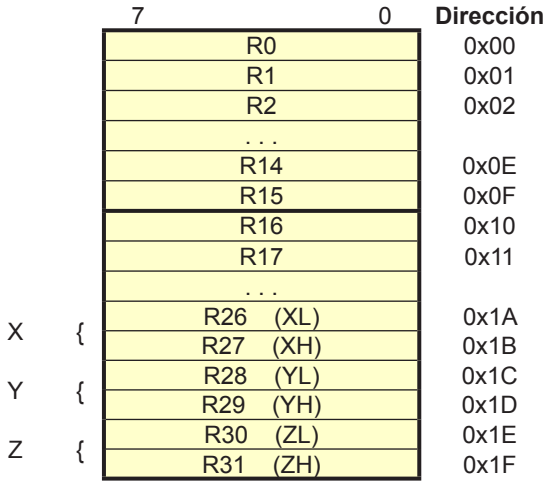


Figura 2.6 Archivo de Registros

Los últimos 6 registros se organizan por pares, formando 3 registros de 16 bits, de esta manera se pueden utilizar como apuntadores para direccionamiento indirecto en el espacio de datos, para ello, estos registros se denominan X, Y y Z. El registro Z también puede usarse como apuntador a la memoria de programa.

Al contar con registros que funcionan como apuntadores, acondicionados para realizar cálculos eficientes de direcciones, e incluir en su repertorio instrucciones de comparaciones con auto-incrementos o auto-decrementos, el núcleo AVR está optimizado para que ejecute código C compilado, ya que entre las características de este lenguaje se encuentra el uso extensivo de apuntadores y ciclos repetitivos con incrementos o decrementos.

2.3 Memoria de Programa

La memoria de programa es un espacio continuo de memoria Flash cuyo tamaño varía entre los miembros de la familia AVR, para el ATMega8 es de 8 KB y para el ATMega16 es de 16 KB. La memoria está organizada en palabras de 16 bits y la mayoría de las instrucciones utilizan una palabra, por lo tanto, el rango de direcciones es de 0x000 a 0xFFF en un ATMega8 y hasta 0x1FFF en un ATMega16. Esto se muestra en la figura 2.7, también se observa que la memoria puede ser particionada en una sección de aplicación y una sección de arranque. En la sección de arranque es posible manejar un cargador para auto programación, con esto, un sistema de manera autónoma puede revisar si existe una versión más actual de su aplicación, en la sección 7.2 se describe cómo tener acceso a la sección de arranque. Si la sección de arranque no es requerida, todo el espacio es dedicado a la aplicación.

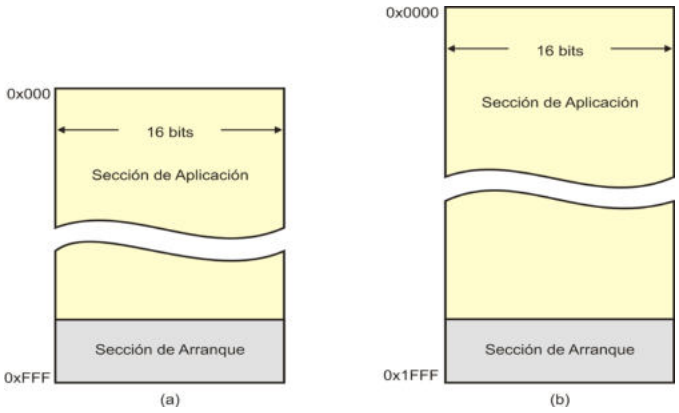


Figura 2.7 Memoria de Programa en (a) un ATMega8 y (b) un ATMega16

La memoria puede ser programada sin necesidad de retirar al MCU de un sistema (programación "*In System*") y soporta hasta 10,000 ciclos de escritura/borrado. En la memoria de programa se encuentran los vectores de interrupciones, es decir, direcciones que toma el PC para que el flujo del programa bifurque a las rutinas que atienden a los eventos que fueron configurados.

Se tiene 19 fuentes de interrupción en un ATMega8 y 21 en un ATMega16, incluyendo la interrupción por reinicio (*reset*). En las tablas 2.1 y 2.2 se describen los vectores de las interrupciones para el ATMega8 y el ATMega16, respectivamente.

Tabla 2.1 Vectores de las Interrupciones en un ATMega8

Vector	Dirección	Fuente	Definición de la Interrupción
1	0x000	RESET	Reinicio por terminal externa, encendido, voltaje bajo o por *watchdog timer*
2	0x001	INT0	Petición de interrupción externa 0
3	0x002	INT1	Petición de interrupción externa 1
4	0x003	TIMER2_COMP	Coincidencia por comparación en el temporizador 2
5	0x004	TIMER2_OVF	Desbordamiento del temporizador 2
6	0x005	TIMER1_CAPT	Captura del temporizador 1 ante un evento externo
7	0x006	TIMER1_COMPA	Coincidencia en la comparación A del temporizador 1
8	0x007	TIMER1_COMPB	Coincidencia en la comparación B del temporizador 1
9	0x008	TIMER1_OVF	Desbordamiento del temporizador 1
10	0x009	TIMER0_OVF	Desbordamiento del temporizador 0
11	0x00A	SPI_STC	Transferencia serial completa por SPI
12	0x00B	USART_RXC	Recepción serial completa en la USART
13	0x00C	USART_UDRE	Registro de datos de la USART vacío
14	0x00D	USATR_TXC	Transmisión serial completa con la USART

Vector	Dirección	Fuente	Definición de la Interrupción
15	0x00E	ADC	Conversión completa en el ADC
16	0x00F	EE_RDY	Concluye una escritura en la EEPROM
17	0x010	ANA_COMP	Comparador analógico
18	0x011	TWI	Interfaz serial de dos hilos
19	0x012	SPM_RDY	Almacenamiento en memoria de programa listo

Tabla 2.2 Vectores de las Interrupciones en un ATMega16

Vector	Dirección	Fuente	Definición de la Interrupción
1	0x000	RESET	Reinicio por terminal externa, encendido, voltaje bajo o por *watchdog timer*
2	0x002	INT0	Petición de interrupción externa 0
3	0x004	INT1	Petición de interrupción externa 1
4	0x006	TIMER2_COMP	Coincidencia por comparación en el temporizador 2
5	0x008	TIMER2_OVF	Desbordamiento del temporizador 2
6	0x00A	TIMER1_CAPT	Captura del temporizador 1 ante un evento externo
7	0x00C	TIMER1_COMPA	Coincidencia en la comparación A del temporizador 1
8	0x00E	TIMER1_COMPB	Coincidencia en la comparación B del temporizador 1
9	0x010	TIMER1_OVF	Desbordamiento del temporizador 1
10	0x012	TIMER0_OVF	Desbordamiento del temporizador 0
11	0x014	SPI_STC	Transferencia serial completa por SPI
12	0x016	USART_RXC	Recepción serial completa en la USART
13	0x018	USART_UDRE	Registro de datos de la USART vacío
14	0x01A	USATR_TXC	Transmisión serial completa con la USART
15	0x01C	ADC	Conversión completa en el ADC
16	0x01E	EE_RDY	Concluye una escritura en la EEPROM
17	0x020	ANA_COMP	Comparador analógico
18	0x022	TWI	Interfaz serial de dos hilos
19	0x024	INT2	Petición de interrupción externa 2
20	0x026	TIMER0_COMP	Coincidencia por comparación en el temporizador 0
21	0x028	SPM_RDY	Almacenamiento en memoria de programa listo

2.4 Memoria de Datos

Para el almacenamiento de datos, los microcontroladores incluyen dos espacios con tecnologías diferentes, como se muestra en la figura 2.8, un espacio de SRAM de 1120 bytes, para el almacenamiento de variables o datos volátiles, y un espacio de EEPROM de 512 bytes, para aquellos datos que se quieren preservar aun en ausencia de energía, como contraseñas, parámetros de configuración, etc. Esto tanto para el ATMega8 como para el ATMega16.

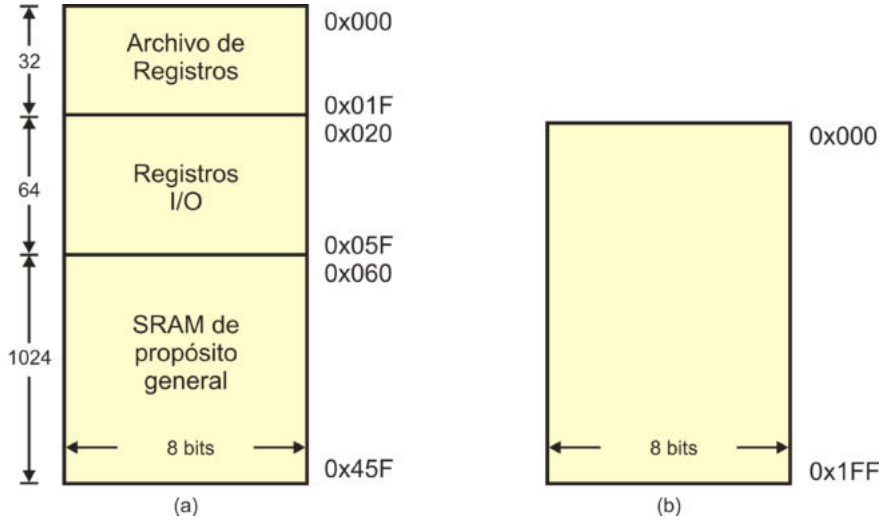


Figura 2.8 Memoria de Datos (a) SRAM y (b) EEPROM

2.4.1 Espacio de SRAM

En la SRAM se tienen tres espacios diferentes en un mapa con direccionamiento lineal, inicia en la dirección 0x000 y concluye en la 0x45F. Las primeras 32 localidades son del Archivo de Registros, luego siguen 64 localidades denominadas como Registros I/O, necesarios para el manejo de recursos, y finalmente se tienen 1024 localidades de SRAM de propósito general.

El núcleo AVR está optimizado para trabajar con los registros de propósito general (sección 2.2.2), las instrucciones los refieren como R0 a R31, o bien como apuntadores (X, Y o Z). No obstante, estos registros también pueden ser referidos como cualquier localidad de SRAM de propósito general, utilizando instrucciones de carga (**`LD`**) o almacenamiento (**`ST`**). Esto se muestra en la figura 2.9, en donde se observa que los registros tienen una dirección en el espacio de los datos.

2.4.1.1 Registros I/O

Los Registros I/O son necesarios para el manejo de los recursos internos de un microcontrolador, se tiene un espacio de direcciones para ubicar hasta 64 registros, las direcciones están en el rango de 0x00 a 0x3F. Aunque el número de registros realmente implementados puede variar entre dispositivos, dependiendo de los recursos internos incluidos.

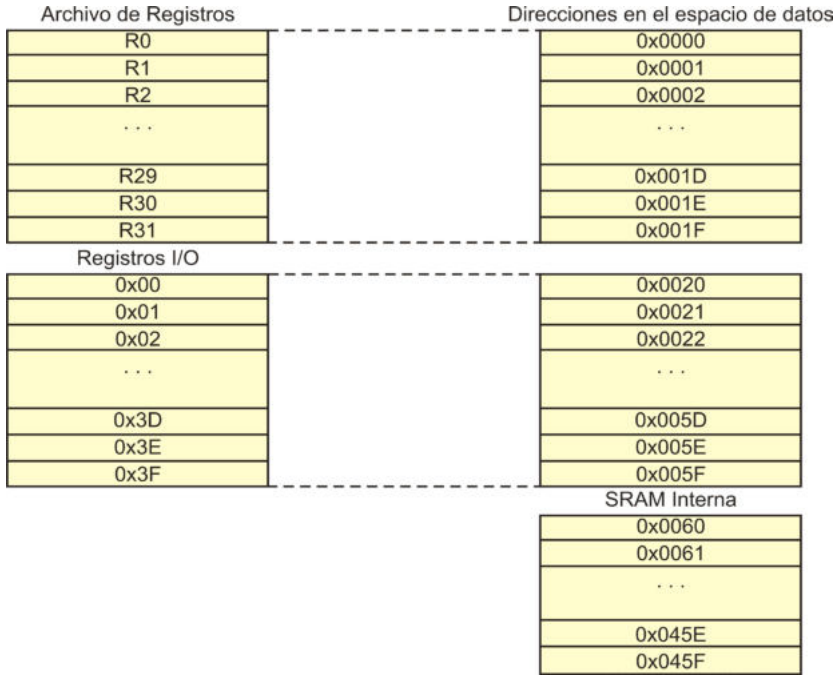


Figura 2.9 Memoria SRAM de Datos

Los Registros I/O se utilizan para definir la configuración, realizar el control o monitorear el estado de los recursos internos. En la figura 2.3 no se encuentra un espacio que explícitamente especifique la ubicación de los Registros I/O, dado que éstos son parte de los módulos con los recursos internos. Por ejemplo, para el manejo de cada uno de los puertos se requiere de 3 registros, uno para configurar al puerto como entrada o salida (configuración), otro para escribir en el puerto (control) y otro para leer del puerto (estado).

La arquitectura de los AVR incluye a las instrucciones **`IN`** y **`OUT`** con las que se tiene un acceso rápido a los Registros I/O, con **`IN`** se transfiere la información de un Registro I/O a un Registro de Propósito General y con **`OUT`** se realiza la operación complementaria, en ambos casos la ejecución se realiza en 1 ciclo de reloj.

De acuerdo con la figura 2.9, los Registros I/O también pueden ser referidos como cualquier localidad de SRAM de propósito general, utilizando instrucciones de carga (**`LD`**) o almacenamiento (**`ST`**), con direcciones en el rango de 0x20 a 0x5F. Tratar a los Registros de Propósito General o a los Registros I/O como SRAM de propósito general no es conveniente, porque las instrucciones de acceso a memoria se ejecutan en 2 ciclos de reloj.

En la tabla 2.3 se muestra una parte de los Registros I/O, el mapa completo se encuentra en el apéndice A. El objetivo de cada registro se describe conforme se van revisando los recursos a los que pertenece, los recursos internos son descritos en los capítulos 4, 5, 6 y 7.

Tabla 2.3 Parte del mapa de Registros I/O

Dirección (Espacio I/O)	Dirección (SRAM)	Nombre	Función
0x3F	0x5F	SREG	Registro de Estado y Control
0x3E	0x5E	SPH	Apuntador de Pila (byte alto)
0x3D	0x5D	SPL	Apuntador de Pila (byte bajo)
0x3C	0x5C	OCR0	Registro para comparación del Temporizador / Contador 0 (no disponible en ATMega8)
0x3B	0x5B	GICR	Registro General para el Control de la Interrupciones
0x3A	0x5A	GIFR	Registro General de banderas de Interrupciones
0x39	0x59	TIMSK	Registro para enmascarar las interrupciones por los Temporizadores/Contadores
0x38	0x58	TIFR	Registro de bandera de interrupciones por los Temporizadores/Contadores
...	...	...	...

Los Registros I/O que están en el rango de 0x00 a 0x1F pueden ser manipulados por sus bits individuales. Con las instrucciones **`SBI`** (*Set Bit in I/O Register*, ajusta un bit en un Registro I/O) y **`CBI`** (*Clear Bit in I/O Register*, limpia un bit en un Registro I/O) es posible cambiar el estado de un bit individual sin modificar al resto, y a través de las instrucciones **`SBIS`** (*Skip if Bit in I/O Register is Set,* brinca si el bit del Registro I/O está en alto) y **`SBIC`** (*Skip if Bit in I/O Register Cleared,* brinca si el bit del Registro I/O está en bajo) es posible evaluar el estado de un bit para determinar la realización de un brinco.

Registro de Estado

El Registro de Estado (**SREG**, *State Register*) es parte de los Registros I/O, por lo que su acceso puede hacerse con instrucciones **`IN`** y **`OUT`**. Este registro es importante debido a que refleja el estado de la CPU y no de algún recurso específico, por eso existen instrucciones especiales para modificar o evaluar a cada uno de sus bits individualmente. Se ubica en la dirección 0x3F (o 0x5F de SRAM), después de un reinicio, todos sus bits tienen el valor de 0. Los bits del Registro de Estado son:

	7	6	5	4	3	2	1	0	
0x3F	I	T	H	S	V	N	Z	C	**SREG**

- **Bit 7 – I: Habilitador Global de Interrupciones**

  Con un 1 lógico las interrupciones son habilitadas, sin embargo, cada interrupción también tiene su habilitador individual. Debe habilitarse por software. Cuando ocurre una interrupción, este bit es limpiado por hardware, para evitar que durante su atención ocurran otras interrupciones.

- **Bit 6 – T: Bit de Almacenamiento para copias**

  Es un espacio para el almacenamiento temporal de 1 bit, puede ser útil si se va a copiar un bit de un registro de propósito general a otro.

- **Bit 5 – H: Bandera de Acarreo en el nibble bajo (*Half Carry*)**

  Se pone en alto si después de una operación aritmética, existe un bit de acarreo del nibble bajo al nibble alto.

- **Bit 4 – S: Bit de Signo**

  Siempre mantiene una XOR entre la bandera de negativo (**N**) y la bandera de sobreflujo (**V**), ambas del registro de Estado.

- **Bit 3 – V: Bandera de Sobreflujo para operaciones en complemento-2**

  Indica que ocurrió un sobreflujo aritmético, es decir, que el resultado de una operación aritmética no alcanzó en la representación de números en complemento a 2 (positivos y negativos). Por ejemplo, al sumar el número 97 (0b01100001) con 42 (0b0101010), el resultado es 139 (0b10001011), con este resultado la bandera V se pone en alto, porque en una representación en complemento a 2 el número 0b10001011 representa al -117. El sobreflujo aritmético se puede presentar en sumas y restas, existen 4 situaciones en las que se presenta sobreflujo, las cuales se resumen en la tabla 2.4.

Tabla 2.4 Situaciones de sobreflujo

Operación	Operando A	Operando B	Resultado (Indicación de sobreflujo)
A + B	≥ 0	≥ 0	< 0
A + B	< 0	< 0	≥ 0
A – B	≥ 0	< 0	< 0
A – B	< 0	≥ 0	≥ 0

- **Bit 2 – N: Bandera de Negativo**

  Indica un resultado negativo en una operación aritmética o lógica, corresponde con el MSB del resultado.

- **Bit 1 – Z: Bandera de Cero**

  Indica que el resultado de una operación aritmética o lógica fue cero.

- **Bit 0 – C: Bandera de Acarreo**

  Indica que el resultado de una operación aritmética o lógica no alcanzó en 8 bits.

Ocasionalmente los bits de acarreo y sobreflujo suelen confundirse, aunque señalizan dos situaciones diferentes. En una operación aritmética, si los operandos son de 8 bits se espera que el resultado ocupe sólo 8 bits, si el resultado no alcanza en 8 bits se genera un bit de acarreo. El sobreflujo tiene que ver con representaciones en complemento a 2 y un sobreflujo no implica que el resultado tenga que requerir de un 9º bit.

El Apuntador de Pila (*Stack Pointer, SP*)

La Pila es un espacio para el almacenamiento temporal de variables, el cual está implementado dentro de la SRAM de propósito general. Una Pila es una estructura en la cual los datos son almacenados o recuperados en uno de sus extremos, denominado tope, de manera que el último dato que ingresa es el primero que es extraído. El Apuntador de Pila (**SP**) es un registro de 16 bits que forma parte de los Registros I/O y que contiene la dirección del tope de la pila, son necesarios 16 bits para poder direccionar todo el espacio de SRAM. El **SP** se compone de 2 registros de 8 bits, **SPH** para la parte alta (0x3E) y **SPL** para la parte baja (0x3D).

	7	6	5	4	3	2	1	0	
0x3E	SP15	SP14	SP13	SP12	SP11	SP10	SP9	SP8	**SPH**
0x3D	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0	**SPL**

La Pila tiene un crecimiento de las direcciones altas de SRAM hacia las direcciones bajas. Su acceso puede realizarse en forma explícita, mediante las instrucciones **`PUSH`** y **`POP`**. Con **`PUSH`** se inserta un dato y se disminuye al **SP**, y con **`POP`** se incrementa al **SP** y luego se extrae un dato. O bien de manera implícita, durante las llamadas y retornos de rutinas. En la llamada a una rutina, en la Pila se almacena la dirección de la instrucción que sigue a la llamada y con un retorno, el tope de la pila remplaza al PC, para continuar con el flujo anterior a la llamada.

Después de un reinicio, el **SP** tiene el valor de 0x0000, por lo tanto, los programas que incluyan rutinas o realicen accesos explícitos a la Pila, deben inicializar al **SP** con 0x045F (última localidad de SRAM, porque la pila crece hacia las direcciones bajas), para que los almacenamientos se realicen dentro de un espacio válido.

2.4.1.2 SRAM de Propósito General

El espacio de propósito general queda disponible para: variables simples que no alcanzan en los 32 registros, para variables compuestas, como arreglos o estructuras, o bien, para la pila de datos temporales. Pero al ser una arquitectura del tipo Registro a Registro, cualquier variable de SRAM que requiera una modificación debe ser llevada a un registro, para ello se realiza una carga (**`LD`**, *load*) y para respaldar un registro en SRAM se realiza un almacenamiento (**`ST`**, *store*), esto se representa en la figura 2.10.

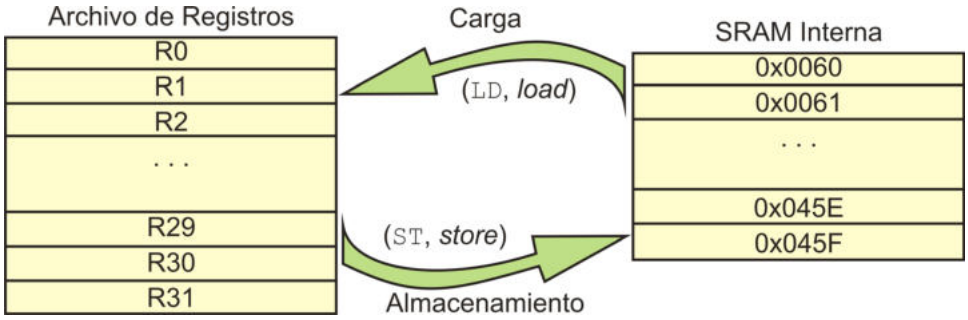


Figura 2.10 Accesos a la memoria SRAM de propósito general

Existe una variedad de instrucciones para el acceso a memoria, ya sean cargas o almacenamientos, utilizando modos de direccionamiento directo o indirecto (por apuntador), así como instrucciones que automáticamente modifican al apuntador, incrementándolo o disminuyéndolo. Todos los accesos a memoria requieren de 3 ciclos de reloj, uno para captura, otro para calcular la dirección de acceso y en el último se habilita la escritura o lectura de la SRAM, esto se muestra en la figura 2.11. Por la segmentación, en el tercer ciclo se captura la siguiente instrucción, aparentando que los accesos a memoria sólo requieren de 2 ciclos de reloj.

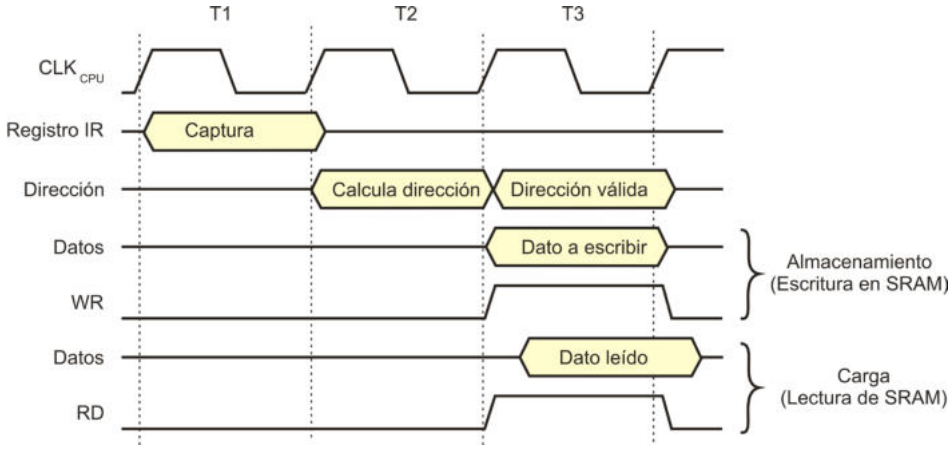


Figura 2.11 Temporización de los accesos a la SRAM de propósito general

2.4.2 Espacio de EEPROM

La EEPROM es un espacio no volátil para el almacenamiento de datos, cuyo tamaño varía en los diferentes integrantes de la familia AVR. Para los microcontroladores ATMega8 y ATMega16 este espacio es de 512 bytes. La memoria EEPROM es un espacio que el núcleo trata como un recurso interno, de manera que su acceso es por medio de los Registros I/O.

Una memoria de cualquier tipo requiere de 3 buses para su manejo: un bus de datos, un bus de direcciones y un bus de control. En el caso de la EEPROM, puesto que es interna al microcontrolador, los buses son manejados por medio de 3 Registros I/O. Para el manejo de las direcciones se tiene al registro **EEAR** (*EEPROM Address Register*), **EEAR** ocupa dos espacios en los Registros I/O para direccionar 512 bytes (9 bits de dirección), estos registros son:

	7	6	5	4	3	2	1	0	
0x1F	-	-	-	-	-	-	-	EEAR8	**EEARH**
0x1E	EEAR7	EEAR6	EEAR5	EEAR4	EEAR3	EEAR2	EEAR1	EEAR0	**EEARL**

Para el manejo de los datos se dispone del registro **EEDR** (*EEPROM Data Register*), el cual se ubica en la dirección 0x1D, dentro del mapa de Registros I/O. Si un dato va a ser escrito en la EEPROM, debe ser colocado en **EEDR**, antes de iniciar con un ciclo de escritura. Para lecturas de la EEPROM, después de un ciclo de lectura, el dato queda disponible en **EEDR**.

Las señales de control son manejadas con el registro **EECR** (*EEPROM Control Register*), en este registro se hacen las habilitaciones requeridas para iniciar los ciclos de lectura o escritura, únicamente los 4 bits menos significativos están implementados, éstos son:

	7	6	5	4	3	2	1	0	
0x1C	-	-	-	-	EERIE	EEMWE	EEWE	EERE	**EECR**

- **Bit 3 – EERIE: Habilitador de Interrupción por fin de Escritura en la EEPROM**

  Las escrituras en la EEPROM requieren varios ciclos de reloj, su final puede detectarse sondeando el estado del bit **EEWE** o por interrupción. Si el bit **EERIE** está en alto, se genera una interrupción cuando culmina una escritura en la EEPROM, siempre que el habilitador global de interrupciones (bit **I** de **SREG**) también esté puesto en alto.

- **Bit 2 – EEMWE: Habilitador Maestro para Escrituras en la EEPROM**

  La EEPROM está orientada para datos importantes en un sistema, como bits de configuración o contraseñas, por lo tanto, es importante evitar que su contenido se pierda por escrituras erróneas. El bit **EEMWE** es parte de un esquema de seguridad y protección del contenido de la EEPROM, al poner en alto a este bit se cuenta con 4 ciclos de reloj dentro de los cuales se puede iniciar con un ciclo de escritura. Pasados estos 4 ciclos el bit **EEMWE** es puesto automáticamente en un nivel bajo y ya no es posible escribir en la EEPROM.

- **Bit 1 – EEWE: Habilitador de Escritura en la EEPROM**

  Al poner en alto a este bit se inicia con un ciclo de escritura, siempre que el bit **EEMWE** haya sido puesto en alto en los 4 ciclos de reloj anteriores. No es posible iniciar con un ciclo de escritura si hay una escritura en proceso. Cuando concluye la escritura, el bit **EEWE** es automáticamente puesto en bajo, este evento puede ser detectado por sondeo o bien, si el bit **EERIE** está en alto, se genera una interrupción.

- **Bit 0 – EERE: Habilitador de Lectura en la EEPROM**

  Al poner en alto a este bit se inicia con un ciclo de lectura, para lo cual sólo se requiere que no haya una escritura en proceso. La lectura es inmediata, el dato leído puede manipularse con la siguiente instrucción.

Ejemplo 2.1: Muestre cómo se codificaría una rutina en lenguaje ensamblador para realizar una escritura en EEPROM.

Para esta rutina se asume que el dato a escribir está en **R16** y que la dirección a utilizar se describe en **R18:R17**, el código es:

```
EEPROM_write:
      SBIC   EECR, EEWE          ; Asegura que no hay escritura en proceso
      RJMP   EEPROM_write

      OUT    EEARH, R18          ; Establece la dirección
      OUT    EEARL, R17
      OUT    EEDR, R16           ; Coloca el dato a escribir

      SBI    EECR, EEMWE         ; Pone en alto al habilitador maestro
      SBI    EECR, EEWE          ; Inicia la escritura

      RET
```

Con la instrucción **SBIC** EECR, EEWE (brinca si el bit **EEWE** del registro **EECR** está en bajo) se está sondeando al bit **EEWE**. El brinco no se realiza si hay una escritura en proceso y se continúa con la siguiente instrucción, la cual reinicia la rutina, el ciclo se va a mantener mientras no concluya la escritura previa.

Ejemplo 2.2: Realice una rutina en lenguaje ensamblador para hacer una lectura en EEPROM.

La rutina lee de la dirección formada por **R18:R17** y el dato leído es colocado en **R16**.

```
EEPROM_read:
            SBIC EECR, EEWE; Asegura que no hay escritura en proceso RJMP
EEPROM_read
```

```
        OUT     EEARH, R18          ; Establece la dirección
        OUT     EEARL, R17
        SBI     EECR, EERE          ; Inicia la lectura

        IN      R16, EEDR           ; Coloca el dato leído
        RET
```

Ejemplo 2.3: Desarrolle 2 funciones en Lenguaje C, una para realizar una escritura en EEPROM y la otra para hacer una lectura.

En lenguaje C, las instrucciones deben realizar las mismas operaciones sobre los registros. La función para la escritura recibe como parámetros el dato a escribir y la dirección en donde va a ser escrito:

```
void  EEPROM_write (unsigned char dato, unsigned int direccion)
{
        while ( EECR & 1 << EEWE ) // Asegura que no hay escritura en proceso
              ;
        EEAR = direccion;           // Establece la dirección
        EEDR = dato;                // Coloca el dato a escribir

        EECR |= ( 1 << EEMWE );     // Pone en alto al habilitador maestro
        EECR |= ( 1 << EEWE );      // Inicia la escritura
}
```

La función para la lectura recibe la dirección a leer y regresa el dato leído:

```
unsigned char  EEPROM_read(unsigned int direccion)
{
        while ( EECR & 1 << EEWE ) // Asegura que no hay escritura en proceso
              ;
        EEAR = direccion;           // Establece la dirección
        EECR |= ( 1 << EERE );      // Inicia la lectura

        return EEDR;                // Regresa el dato leído
}
```

2.5 Puertos de Entrada/Salida

Los puertos de entrada/salida proporcionan el mecanismo por medio del cual un microcontrolador se comunica con su entorno. En la figura 2.2 se mostró el aspecto externo de los 2 dispositivos bajo estudio, puede notarse que el ATMega8 tiene tres puertos: Puerto B, C y D, donde el puerto C es de 7 bits, mientras que los puertos B y D son de 8 bits. Con respecto al ATMega16 se observan 4 puertos: Puerto A, B, C y D, todos de 8 bits.

Los puertos son de propósito general, es decir, el usuario puede utilizarlos para monitorear entradas o generar salidas como mejor le convenga, no obstante, todas las terminales tienen una función alterna, esto es necesario porque muchos de los recursos internos van a requerir información del exterior. Por ejemplo, el puerto serie necesita una terminal externa para recepción y otra para transmisión, ésta es la función alterna para PD0 y PD1, respectivamente. La función alterna de las terminales se revisa conforme se van describiendo los recursos internos del microcontrolador, los recursos internos se describen en los capítulos 4, 5, 6 y 7.

Para el manejo de cada puerto se requiere de 3 registros del espacio de Registros I/O, éstos son:

- **PORTx**: Registro para la escritura o lectura de un *latch* que está conectado a la terminal del puerto por medio de un buffer de 3 estados. El buffer está activo cuando el puerto es configurado como salida, bajo estas circunstancias, todo lo que se escribe en este registro se ve reflejado en las terminales de los puertos.

- **DDRx:** Registro para la escritura o lectura de un *latch* que está conectado a la terminal de activación del buffer de 3 estados, con este registro se define la dirección del puerto. Si se escribe un **1 lógico**, se activa al buffer haciendo que el puerto funcione como **salida**, al escribir un **0 lógico**, el buffer está inactivo y el puerto funciona como **entrada**. Cada terminal tiene sus propios recursos, de manera que la dirección de una terminal puede diferir de las demás, aun en el mismo puerto.

- **PINx:** Registro sólo de lectura, para hacer lecturas directas en las terminales de los puertos.

La **x** hace referencia a cada uno de los puertos, puede ser A, B, C o D. Para el ATMega8, como tiene 3 puertos requiere de 9 Registros I/O, éstos son:

	7	6	5	4	3	2	1	0	
0x18	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0	**PORTB**
0x17	DDRB7	DDRB6	DDRB5	DDRB4	DDRB3	DDRB2	DDRB1	DDRB0	**DDRB**
0x16	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0	**PINB**
0x15	-	PORTC6	PORTC5	PORTC4	PORTC3	PORTC2	PORTC1	PORTC0	**PORTC**
0x14	-	DDRC6	DDRC5	DDRC4	DDRC3	DDRC2	DDRC1	DDRC0	**DDRC**
0x13	-	PINC6	PINC5	PINC4	PINC3	PINC2	PINC1	PINC0	**PINC**
0x12	PORTD7	PORTD6	PORTD5	PORTD4	PORTD3	PORTD2	PORTD1	PORTD0	**PORTD**
0x11	DDRD7	DDRD6	DDRD5	DDRD4	DDRD3	DDRD2	DDRD1	DDRD0	**DDRD**
0x10	PIND7	PIND6	PIND5	PIND4	PIND3	PIND2	PIND1	PIND0	**PIND**

El bit 7 en los registros del puerto C no está implementado en un ATMega8.

El ATMega16 tiene 4 puertos, los registros de los puertos B, C y D tienen la misma dirección que en un ATMega8, difiriendo en que el bit 7 del puerto C si está implementado.

El ATMega16 también tiene al puerto A, el cual requiere de los siguientes 3 registros:

	7	6	5	4	3	2	1	0	
0x1B	PORTA7	PORTA6	PORTA5	PORTA4	PORTA3	PORTA2	PORTA1	PORTA0	**PORTA**
0x1A	DDRA7	DDRA6	DDRA5	DDRA4	DDRA3	DDRA2	DDRA1	DDRA0	**DDRA**
0x19	PINA7	PINA6	PINA5	PINA4	PINA3	PINA2	PINA1	PINA0	**PINA**

Si no se considera la función alterna en las terminales de los puertos, éstos incluyen el mismo hardware, el cual es mostrado en la figura 2.12, en donde se presenta la organización de la terminal **n** en el puerto **x.**

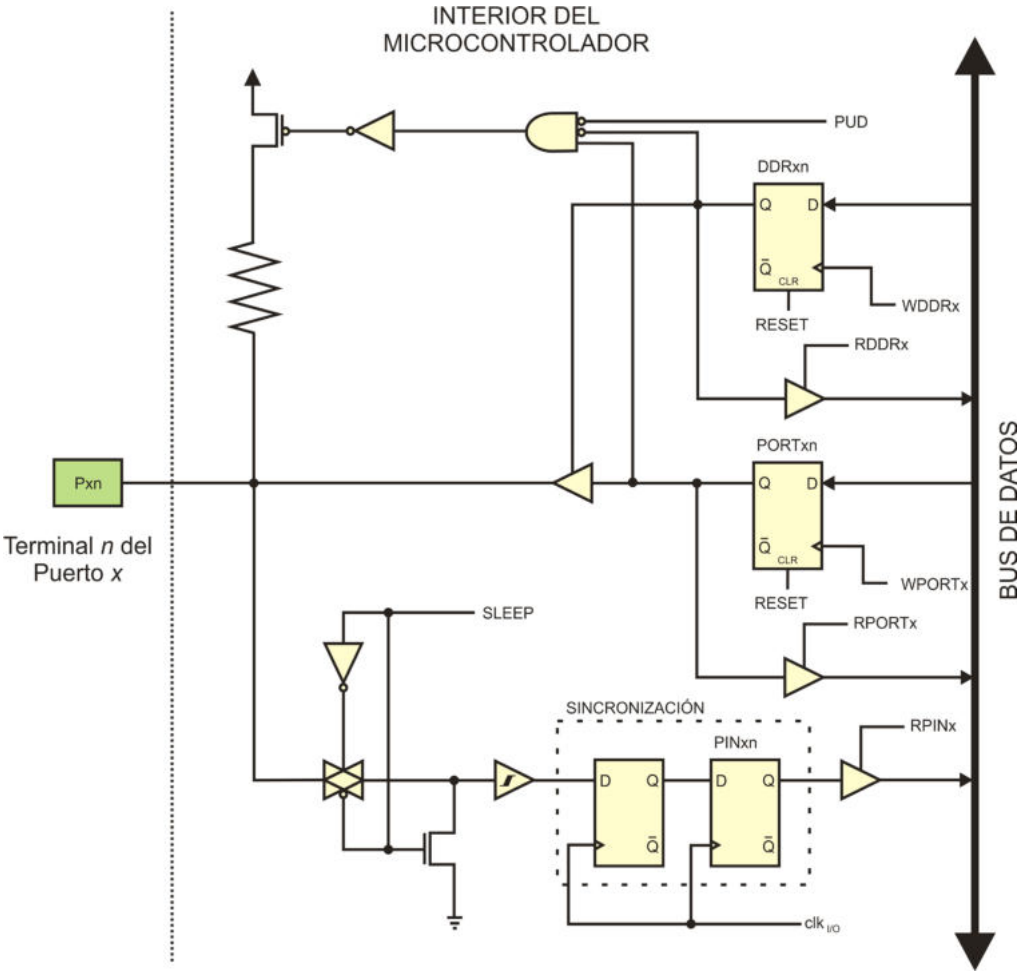


Figura 2.12 Organización del hardware de la terminal *n* del puerto *x*

En la figura 2.12 se observa que para cada terminal se tienen 3 rutas:

- Una ruta de salida que es manejada por el flip-flop tipo D denominado **PORTxn**, el cual puede ser leído o escrito. Este flip-flop se conecta a la terminal de salida por medio de un buffer de 3 estados. Una lectura en **PORTxn** no devuelve el valor de la terminal, sino lo último que fue escrito en el flip-flop.

- Una ruta de entrada que es manejada por el flip-flop tipo D denominado **PINxn**, en el cual continuamente se va a escribir el contenido de la terminal del puerto. Este flip-flop va precedido por otro similar, ambos manejados por la señal del reloj de entrada y salida. Esta conexión en cascada garantiza una señal estable al realizar las lecturas del puerto, con el inconveniente de que un cambio en la terminal se ve reflejado hasta el ciclo de reloj siguiente. También se observa un circuito para desconectar al puerto ante algún modo de reposo (SLEEP) del microcontrolador, así como un buffer con histéresis, para eliminar ruido. Si la terminal fue configurada como salida (buffer de 3 estados habilitado), con una lectura en **PINxn** se obtiene el dato escrito en **PORTxn**.

- Una ruta para el buffer de tres estados que es manejada por el flip-flop tipo D denominado **DDRxn**, el cual puede ser leído o escrito. Este flip-flop se conecta con el buffer de tres estados para determinar si el puerto va a estar configurado como entrada (se escribe un 0 lógico para desactivar al buffer) o como salida (se escribe un 1 lógico para activar al buffer).

También se observa una resistencia de fijación hacia un nivel alto de voltaje (*pull-up*), su habilitación depende de valor del flip-flop **PORTxn**, del flip-flop **DDRxn** y del bit **PUD** (*PUD: Pull-Up Disable*). El bit **PUD** se encuentra en el Registro I/O denominado **SFIOR** (*SFIOR: Special Function I/O Register,* Registro I/O de función especial). Con estos 3 bits se generan las combinaciones que se resumen en la tabla 2.5.

Tabla 2.5 Estado del Puerto ante las diferentes combinaciones entre DDRxn, PORTxn y PUD

DDRxn	PORTxn	PUD (en SFIOR)	E/S	Pull-Up	Comentario
0	0	X	Entrada	No	Entrada sin resistor de *Pull-Up*
0	1	0	Entrada	Si	Entrada con resistor de *Pull-Up*
0	1	1	Entrada	No	Entrada sin resistor de *Pull-Up*
1	0	X	Salida	No	Salida en bajo
1	1	X	Salida	No	Salida en alto

De las 3 combinaciones en que el puerto es entrada, sólo en la segunda queda habilitado el resistor de *Pull-Up*, esta combinación es adecuada para el manejo de botones o interruptores, porque garantiza un 1 lógico cuando el dispositivo está abierto. El otro extremo del botón o interruptor se conecta a tierra, de manera que cuando se cierra el circuito introduce un 0 lógico, el resistor de *Pull-Up* evita un corto entre el voltaje de alimentación y tierra.

Ejemplo 2.4: Muestre el código requerido para configurar la parte alta del puerto B como etradas y la parte baja como salidas, y habilite los resistores de *Pull-Up* de las 2 entradas más significativas.

La versión en lenguaje ensamblador:

```
LDI     R16, 0B00001111      ; Configuración de entradas y salidas
OUT     DDRB, R16

LDI     R17, 0B11000000      ; Resistencia de Pull-Up en los 2 MSB
OUT     PORTB, R17
```

En lenguaje C el código se simplifica:

```
DDRB  = 0B00001111; // Configuración de entradas y salidas
PORTB = 0B11000000; // Resistencia de Pull-Up en los 2 MSB
```

Ejemplo 2.5: Muestre la secuencia de código que configure al puerto A como entrada y al puerto B como salida, para luego transferir la información del puerto A al puerto B.

En la versión en ensamblador, se utiliza un registro interno para poder hacer la transferencia:

```
LDI     R16, 0x00            ; Configura el Puerto A como entrada
OUT     DDRA, R16
LDI     R16, 0xFF            ; Configura al Puerto B como salida
OUT     DDRB, R16

IN      R16, PINA            ; Las lecturas se hacen en el registro PIN
OUT     PORTB, R16    ; Las escrituras se hacen en el registro PORT
```

En lenguaje C no se requiere de una variable de manera explícita, para hacer la transferencia:

```
DDRA = 0x00;                 // Configura el Puerto A como entrada
DDRB = 0xFF;                 // Configura al Puerto B como salida

PORTB = PINA;                // Se lee en PIN y se escribe en PORT
```

Cabe aclarar que después de un reinicio los registros de los puertos tienen el valor de 0x00, por lo que por omisión, un puerto es configurado como entrada.

2.6 Sistema de Interrupciones

Una **interrupción** es la ocurrencia de un evento producido por algún recurso del microcontrolador, que ocasiona la suspensión temporal del programa principal. La CPU atiende al evento con una función conocida como rutina de servicio a la interrupción (*ISR*, *Interrupt Service Routine*). Una vez que la CPU concluye con las instrucciones de la ISR, continúa con la ejecución del programa principal, regresando al punto en donde fue suspendida su ejecución.

El núcleo AVR cuenta con la unidad de interrupciones, un módulo que va a determinar si se tienen las condiciones para que ocurra una interrupción. Son tres las condiciones necesarias para que un recurso produzca una interrupción: El habilitador global de interrupciones (bit **I** de **SREG**) debe estar activado, el habilitador individual de la interrupción del recurso también debe estar activado y en el recurso debe ocurrir el evento esperado.

Al configurar los recursos adecuadamente, por hardware se van a monitorear diferentes eventos, reduciendo la tarea que la CPU realiza por software, porque no necesita un sondeo continuo para determinar el estado de los recursos. Puesto que son muchos los eventos que pueden producir interrupciones, con el trabajo de la unidad de interrupciones se tiene la ilusión de que se están haciendo diferentes tareas en forma simultánea.

La ISR debe colocarse en una dirección preestablecida por Hardware, la cual corresponde con un vector de interrupciones.

Existen 2 esquemas para organizar a los vectores de interrupciones, en el primer esquema se maneja una dirección única (sólo un vector), de manera que cualquier evento ocasiona que la ejecución del programa bifurque a la misma dirección. Posterior a ello, por software se evalúa un conjunto de banderas para determinar la causa de la interrupción. En el segundo esquema se tienen diferentes direcciones (diversos vectores de interrupciones), una por cada evento, de manera que la dirección destino de la bifurcación depende de la causa de la interrupción.

El segundo esquema es el empleado por los microcontroladores AVR, requiere una estructura de Hardware más compleja que en el primer esquema, pero se simplifica el software, dado que no requiere de una revisión de banderas.

Un aspecto importante es que los eventos pueden ocurrir en cualquier momento, es decir, en forma asíncrona. Esto hace la diferencia entre una rutina normal y una ISR, dado que en el primer caso se puede calcular con certeza cuándo se va a realizar una llamada, podría decirse que el flujo de un programa sin interrupciones es conocido con antelación. Por el contrario, en un programa con interrupciones el flujo puede cambiar en cualquier momento, es como si el programa trabajara en dos niveles, el nivel del programa base y el nivel de atención a las interrupciones, por lo tanto, como parte de las ISRs debe incluirse el código necesario para proteger a las variables o registros cuyo contenido sea vital, al nivel del programa base. En la figura 2.13 se ilustra la idea.

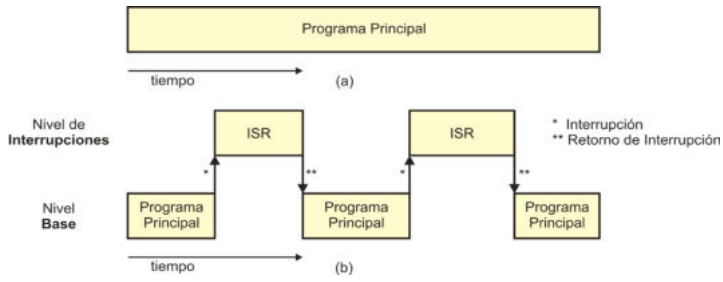


Figura 2.13 Representación de la ejecución de (a) un programa sin interrupciones y (b) un programa con interrupciones

En los microcontroladores AVR se tienen diferentes fuentes de Interrupción:

- Una interrupción por inicialización o *Reset*.
- Dos interrupciones externas (tres en el ATMega16).
- Siete interrupciones por los temporizadores (ocho en el ATMega16), pueden ser por comparación, captura o desbordamiento.
- Una interrupción al completar una transferencia serial (puerto SPI).
- Tres debidas el puerto serie (USART): por transmisión, por recepción y por buffer vacío.
- Una al finalizar una conversión analógica a digital.
- Una al finalizar una escritura en EEPROM.
- Una por el comparador analógico.
- Una por la interfaz serial a dos hilos (TWI).
- Una para la escritura en memoria de programa.

Se observa que son 2 interrupciones más en un ATMega16 que en un ATMega8, en otros miembros de la familia AVR el número de interrupciones puede variar, ya que depende de los recursos empotrados en el dispositivo.

Cuando el microcontrolador se enciende o reinicia, las interrupciones no están habilitadas, su habilitación requiere la puesta en alto del bit **I** de **SREG** y de los habilitadores individuales de los periféricos incorporados en el microcontrolador. En los capítulos 4, 5, 6 y 7 se describen los recursos internos, incluyendo una descripción de los bits para activar las interrupciones y las condiciones necesarias para que se generen.

Al generarse una interrupción, el PC es almacenado en la pila de datos y toma el valor de una entrada en el vector de interrupciones (según sea la interrupción). Además de desactivar al bit **I** para no aceptar más interrupciones. En la tabla 2.1 se mostraron las direcciones de los vectores de interrupciones en un ATMega8 y en la tabla 2.2 las de un ATMega16.

Una rutina de atención a interrupciones es finalizada con la instrucción **`RETI`**, con la cual el PC recupera el valor del tope de la pila y pone en alto nuevamente al bit **I**, para que la CPU pueda recibir más interrupciones.

El manejo de interrupciones necesariamente hace uso de la pila, por este motivo, cuando se programa en lenguaje ensamblador se debe inicializar al apuntador de pila en el programa principal. En lenguaje C esto no es necesario, dado que es parte del código que se agrega al hacer la traducción de alto nivel a código máquina.

Ejemplo 2.6: Muestre como se organizaría un programa en lenguaje ensamblador para utilizar interrupciones en un ATMega8.

```
; Un programa generalmente inicia en la dirección 0

       .ORG   0x000
       RJMP   Principal     ; Se evita el vector de interrupciones

       .ORG   0x001
       RJMP   Externa_0     ; Bifurca a su ISR externa 0

       .ORG   0x002
       RJMP   Externa_1     ; Bifurca a su ISR externa 1

; Si fuera necesario, aquí estarían otras bifurcaciones

       .ORG     0x013
Principal:                  ; Aquí se debe ubicar el código principal
         . . .              ; Debe activar las interrupciones
         . . .
; Posterior al código principal, deben situarse las ISRs

Externa_0:                  ; Respuesta a la interrupción externa 0
         . . .
       RETI                 ; Debe terminar con RETI

Externa_1:                  ; Respuesta a la interrupción externa 1
         . . .
       RETI                 ; Debe terminar con RETI
```

La directiva **.ORG** indica en donde se van a ubicar las instrucciones subsecuentes en la memoria de programa. Al comparar las tablas 2.1 y 2.2, se observa que en un ATMega8 sólo se dispone de una dirección para cada bifurcación, mientras que en un ATMega16 se dispone de 2 direcciones, esto porque el espacio completo de direcciones de un ATMega8 puede ser alcanzado con una instrucción **RJMP**, que sólo ocupa una palabra de 16 bits. Pero con esa instrucción no se puede cubrir el espacio completo de un ATMega16, para ello puede usarse la instrucción **JMP**, que tiene un alcance mayor por lo que ocupa 2 palabras de 16 bits. La diferencia en el alcance entre **RJMP** y **JMP** se debe a que manejan diferentes modos de direccionamiento, los cuales son descritos en la sección 3.2.

Por lo tanto, si el código del ejemplo 2.6 va a utilizarse en un ATMega16, únicamente deben cambiarse las direcciones de los vectores de interrupciones y en las bifurcaciones puede emplearse **RJMP** o **JMP**, dependiendo de la ubicación de la ISR.

En lenguaje C todas las funciones de atención a interrupciones se llaman ISR, difieren en que reciben argumentos diferentes, el argumento corresponde con una etiqueta proporcionada por el fabricante, seguida de la palabra **vect**. Las etiquetas se encuentran en la columna titulada como **fuente** en las tablas 2.1 y 2.2.

Ejemplo 2.7: Muestre cómo se organizaría un programa en lenguaje C para utilizar interrupciones en un AVR.

```
#include <avr/io.h>              // Definiciones de entradas y salidas
#include <avr/interrupt.h>       // Manejo de interrupciones

// Las ISRs se ubican antes del programa principal
ISR (INT0_vect)                  // Servicio a la interrupción externa 0
{
    . . . .
}

ISR (INT1_vect)                  // Servicio a la interrupción externa 1
{
   . . . .
}

int main(void)                   // Programa Principal
{
    . . . . .                    // Debe activar las interrupciones
}
```

El código descrito en el ejemplo 2.7 es independiente del dispositivo a utilizar, funciona para cualquier miembro de la familia AVR.

2.6.1 Manejo de Interrupciones

Si en un programa se utilizan las interrupciones, se requiere:

- Configurar el recurso o recursos para monitorear el evento o eventos.
- Habilitar a la interrupción o interrupciones (individual y global, en cada caso).
- Continuar con la ejecución normal de la aplicación.

En aplicaciones que utilizan interrupciones, es frecuente ciclar al programa principal en un lazo infinito sin realizar alguna actividad, con ello, el MCU permanece ocioso, dejando la funcionalidad del sistema a las ISRs.

Cuando ocurre una interrupción, el microcontrolador automáticamente realiza lo siguiente:

- Concluye con la instrucción bajo ejecución.
- Desactiva al habilitador global de interrupciones, para que no pueda recibir una nueva interrupción mientras atiende a la actual.
- Respalda en la pila al PC (previamente incrementado).
- Asigna al PC una dirección de los vectores de interrupciones, para dar paso a la ISR.
- Atiende al evento con la ISR.

Cuando una ISR termina (con la instrucción **`RETI`**, si se programó en ensamblador), en el MCU ocurre lo siguiente:

- Se limpia la bandera del evento que generó la interrupción.
- El habilitador global se activa.
- El PC toma el valor del tope de la pila, para que la ejecución continúe en el programa principal.

2.7 Inicialización del Sistema (*reset*)

La inicialización o *reset* de un microcontrolador es fundamental para su operación adecuada, porque garantiza que sus registros internos van a tener un valor inicial conocido. En un ATMega8 se tienen cuatro causas o fuentes de *reset*, en el ATMega16 se agrega una más, dado que tiene recursos adicionales para el manejo de la interfaz JTAG[4] . En la figura 2.14 se muestra la organización del hardware de *reset*, se observa cómo las diferentes causas pueden producir la señal denominada Reset Interno, que es en sí la que afecta a la CPU del microcontrolador. Las causas de inicialización son:

- **Reset de Encendido (*Power-on Reset*):** El MCU es inicializado cuando el voltaje de la fuente está por abajo del voltaje de umbral de encendido (V_{POT}), el cual tiene un valor típico de 2.3 V.
- **Reset Externo:** El MCU es inicializado cuando un nivel bajo está presente en la terminal RESET por un tiempo mayor a 1.5 uS, que es la longitud mínima requerida (t_{RST}).

4 **JTAG** es un acrónimo para **Joint Test Action Group** y es el nombre común de la norma IEEE 1149.1, originalmente para la evaluación de circuitos impresos. Actualmente **JTAG** hace referencia a una interfaz serial utilizada para la prueba de circuitos integrados y como medio para depurar sistemas empotrados.

- **Reset por *Watchdog*:** El MCU es inicializado cuando se ha habilitado al *Watchdog Timer* y éste se ha desbordado.

- **Reset por reducción de voltaje (*Brown out*).** Se inicializa al MCU cuando el detector de reducción de voltaje está habilitado y el voltaje de la fuente de alimentación está por debajo del umbral establecido (V_{BOT}). El valor de V_{BOT} es configurable a 2.7 V ó 4.0 V, y el tiempo mínimo necesario (t_{BOD}) para considerar una reducción de voltaje es de 2 uS.

- **Reset por JTAG:** El MCU es inicializado tan pronto como exista un 1 lógico en el Registro de Reset del Sistema JTAG.

Referente a la figura 2.14, los bits BODEN, BODLEVEL, CKSEL y SUT son fusibles que forman parte de los *Bits de Configuración y Seguridad*, su valor se define en el momento en que se programa al dispositivo y no pueden ser modificados en tiempo de ejecución.

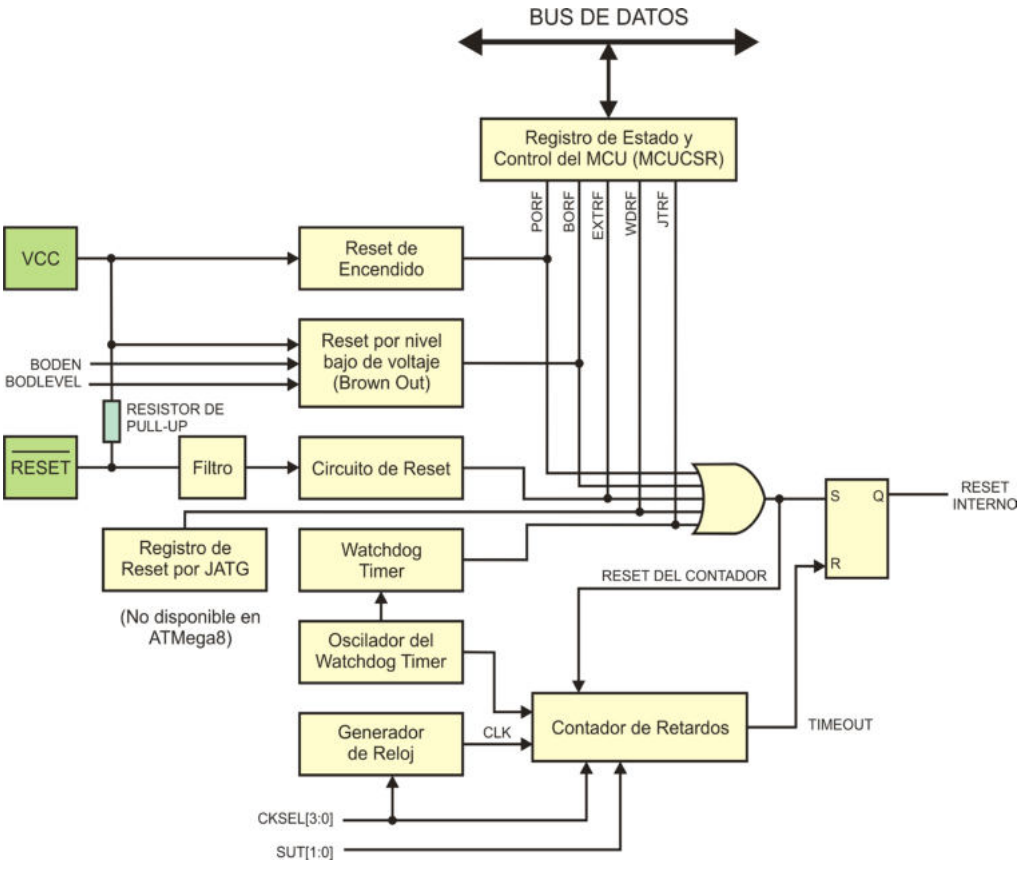


Figura 2.14 Organización del hardware de inicialización o *reset*

Puesto que hay diferentes causas de reinicio, los AVR incluyen al Registro de Estado y Control del MCU (**MCUCSR**, *MCU Control and Status Register*) en el cual queda indicada la causa de *reset* por medio de una bandera. Los bits del registro **MCUCSR** son:

	7	6	5	4	3	2	1	0	
0x34	JTD	ISC2	-	JTRF	WDRF	BORF	EXTRF	PORF	**MCUCSR**

- **Bits 7, 6 y 5:** No tienen relación con el *reset* del sistema, en el ATMega8 no están implementados.
- **Bit 4 – JTRF: Bandera de reinicio por JTAG**

  No está implementada en el ATMega8.
- **Bit 3 – WDRF: Bandera de reinicio por desbordamiento del *Watchdog timer***
- **Bit 2 – BORF: Bandera de reinicio por reducción de voltaje (*Brown out*)**
- **Bit 1 – EXTRF: Bandera de reinicio desde la terminal de *reset***
- **Bit 0 – PORF: Bandera de reinicio por encendido**

En la figura 2.15 se muestra la generación del Reset Interno inmediatamente después de que se ha alcanzado el voltaje de encendido (V_{POT}) en la terminal de V_{CC}. La terminal de RESET internamente está conectada a V_{CC}, a través de un resistor de *pull-up*, en la figura 2.15 se observa que su comportamiento es el mismo que el de la terminal V_{CC}.

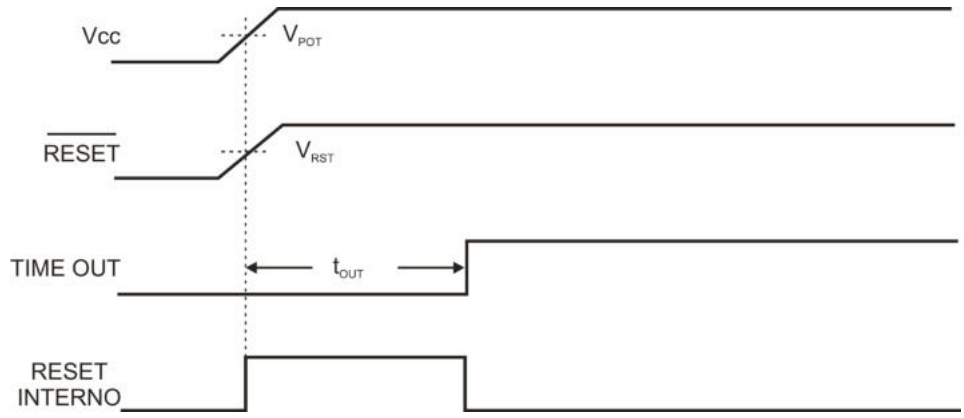


Figura 2.15 Reset de encendido

La duración de la señal Reset Interno es determinada por el módulo Contador de Retardos, que también puede verse en la figura 2.14.

El Reset Interno es generado mientras transcurre el tiempo de establecimiento (t_{OUT}), cuya duración depende de la frecuencia de trabajo y del valor de los fusibles CKSEL y SUT. Los MCUs ATMega8 y ATMega16 son comercializados con un tiempo de establecimiento ajustado a 65 mS.

En la figura 2.16 se muestra la generación del Reset Interno como una respuesta a la terminal RESET, en la cual debe conectarse un botón a tierra para su activación. En (a) el RESET externo está activo al alimentar al dispositivo, por lo que la señal de Reset Interno inicia en alto, el tiempo de establecimiento inicia desde el momento en que la terminal RESET alcanza el umbral de encendido (V_{RST}, valor máximo de 0.9 V) y al finalizar el tiempo de establecimiento la señal de Reset Interno se va a un nivel bajo. En la figura 2.16 (b) se muestra como en cualquier momento puede activarse la terminal externa de RESET, generando nuevamente la señal de Reset Interno.

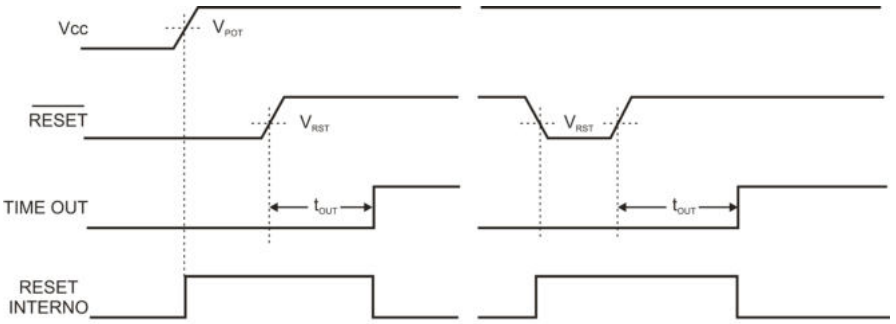


Figura 2.16 Reset externo (a) en el encendido y (b) en cualquier otro instante

En la figura 2.17 se muestra una condición de inicialización debida a una reducción de voltaje (*Brown out*) en la terminal de alimentación (V_{CC}). El Reset Interno se genera cuando V_{CC} cae por debajo del límite inferior de un voltaje de umbral (V_{BOT-}), el tiempo de establecimiento inicia cuando V_{CC} alcanza al límite superior del voltaje de umbral (V_{BOT+}). La señal de Reset Interno concluye al terminar el tiempo de establecimiento. La inicialización por reducción de voltaje se habilita con el fusible BODEN, que es parte de los *Bits de Configuración y Seguridad*, y con BODLEVEL se determina si el límite inferior del umbral es de 2.7 V o de 4.0 V; el voltaje de histéresis (V_{HYST}) es para evitar oscilaciones ante variaciones continuas y su valor es de 1.3 mV.

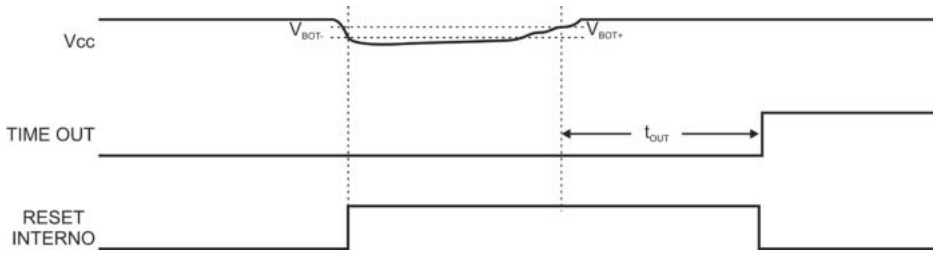


Figura 2.17 Inicialización por reducción de voltaje (*Brown out*)

Cuando el *Watchdog Timer* se desborda, genera un pulso en alto por un ciclo de reloj que es suficiente para provocar una inicialización del MCU, esto se muestra en la figura 2.18, donde sólo se incluyen las señales involucradas. El tiempo de establecimiento inicia inmediatamente después del desbordamiento.

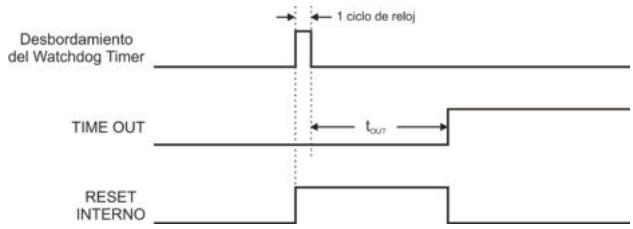


Figura 2.18 Inicialización por desbordamiento del *Watchdog Timer*

2.8 Reloj del Sistema

Para la sincronización de los microcontroladores ATMega8 y ATMega16 se tienen diferentes fuentes de reloj, la selección se realiza por medio de un multiplexor. La fuente de reloj seleccionada es distribuida por medio de un módulo de control del reloj hacia los diferentes módulos del sistema, esto se observa en la figura 2.19.

La selección de la fuente de reloj se realiza por medio de los fusibles CKSEL que son parte de los *Bits de Configuración y Seguridad*, otros fusibles involucrados son CKOPT y SUT. El fusible CKOPT proporciona más alternativas al utilizar un cristal o resonador externo o habilita la conexión de capacitores internos si se selecciona un reloj externo o un cristal de baja frecuencia; mientras que con los fusibles SUT es posible modificar el tiempo de establecimiento después de un reinicio.

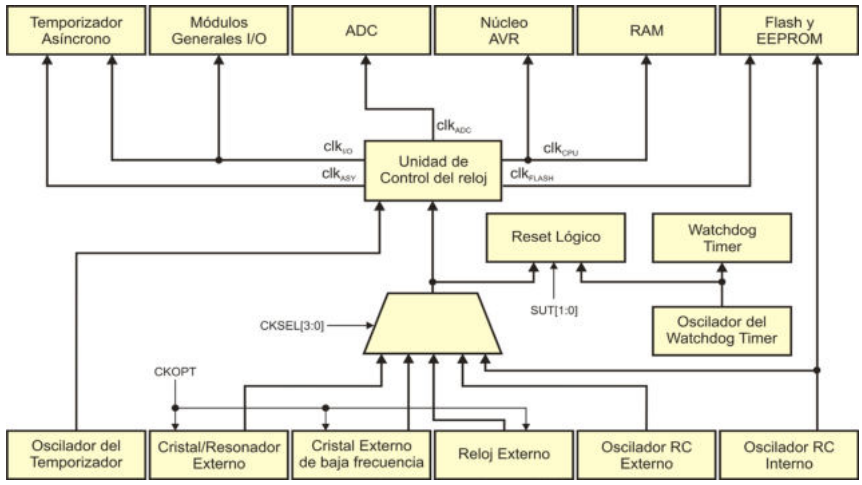


Figura 2.19 Generación y distribución de la señal de reloj

En la tabla 2.6 se indica cuál es la fuente de reloj en función del valor de los fusibles CKSEL, en algunos casos se tiene más de una combinación, porque se tienen diferentes alternativas.

Tabla 2.6 Selección de Reloj a partir de los fusibles CKSEL

Opción para el Reloj del Sistema	CKSEL[3:0]
Resonador Cerámico o Cristal Externo	1010 – 1111
Cristal de Baja Frecuencia Externo	1001
Oscilador RC Externo	0101 – 1000
Oscilador RC Calibrado Interno	0001 – 0100
Reloj Externo	0000

Además de las 5 posibles fuentes de reloj que ingresan al multiplexor, en la figura 2.19 se muestra el módulo del oscilador del temporizador. El módulo está optimizado para trabajar con un oscilador externo de 32.768 KHz, para generar una señal de reloj (**clk_{ASY}**) disponible para que con el temporizador 2 pueda manejarse un contador de tiempo real, con ello, es posible que el temporizador 2 trabaje a una frecuencia diferente a la del resto del sistema.

La unidad de control del reloj se encarga de generar diferentes señales de reloj y distribuirlas a los diferentes módulos, las señales de reloj son:

- **clk_{CPU}:** Ruteado al núcleo AVR, incluyendo al archivo de registros, registro de Estado, Memoria de datos, apuntador de pila, etc.

- **clk_{FLASH}:** Señal de reloj suministrada a las memorias FLASH y EEPROM.

- **clk_{ADC}:** Reloj dedicado al ADC, el ADC trabaja a una frecuencia menor que la CPU con el objetivo de reducir el ruido generado por interferencia digital y mejorar las conversiones.

- **$clk_{I/O}$:** Es la señal de reloj utilizada por los principales módulos de recursos: Temporizadores, interfaz SPI y USART. Además de ser requerido por el módulo de interrupciones externas.

- **clk_{ASY}:** Es una señal de reloj asíncrona, con respecto al resto del sistema, es empleada para sincronizar al temporizador 2, el módulo que genera esta señal está optimizado para ser manejado con un cristal externo de 32.768 KHz. Frecuencia que permite al temporizador funcionar como un contador de tiempo real, aun cuando el dispositivo está en reposo.

En los siguientes apartados se describen las fuentes de reloj y cómo seleccionarlas.

2.8.1 Resonador Cerámico o Cristal Externo

En la tabla 2.6 se presentaron 6 combinaciones de los fusibles CKSEL para esta opción de reloj, de 1010 a 1111. Sin embargo, sólo son 3 opciones porque el bit menos significativo está dedicado a la definición de los tiempos de establecimiento. Las opciones se amplían al utilizar al fusible CKOPT, en la tabla 2.7 se muestran las diferentes opciones con sus rangos de frecuencias. Para la primera opción (CKSEL = "101", CKOPT = '1'), por el rango de frecuencias no se deberían usar cristales, sólo resonadores cerámicos.

Tabla 2.7 Opciones para el empleo de un resonador cerámico o cristal externo

CKOPT	CKSEL [3:1]	Rango de Frecuencias (MHz)	Valores recomendados para C1 y C2 (pF)
1	101	0.4 – 0.9	-
1	110	0.9 – 3.0	12 – 22
1	111	3.0 – 8.0	12 – 22
0	101, 110, 111	1.0 ≤	12 – 22

La figura 2.20 muestra como se debe hacer la conexión del cristal o resonador cerámico junto con sus capacitores.

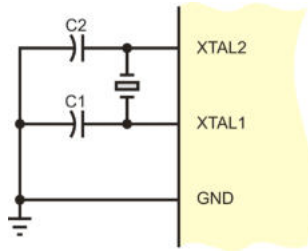


Figura 2.20 Resonador Cerámico o Cristal Externo

Con esta opción de reloj, se tienen los tiempos de establecimiento después de algún modo de bajo consumo y los retardos después de un reinicio, mostrados en la tabla 2.8, en donde ck significa: ciclos de reloj.

2.8.2 Cristal de Baja Frecuencia Externo

Con esta opción para el reloj, el hardware está acondicionado para ser manejado con un cristal de 32.768 KHz, esta frecuencia proporciona la base para un contador de tiempo real. Todo el sistema trabajaría a esta frecuencia, por lo que sólo es conveniente si la aplicación requiere ejecutar instrucciones en periodos que son fracciones de segundos.

Tabla 2.8 Tiempos de establecimiento y retardos al emplear un resonador cerámico o cristal externo

CKSEL0	SUT [1:0]	Establecimiento después de un bajo consumo	Retardo después de un *reset* (V_{cc} = 5 V)
0	00	258 ck	4.1 mS
0	01	258 ck	65 mS
0	10	1K ck	-
0	11	1K ck	4.1 mS
1	00	1K ck	65 mS
1	01	16K ck	-
1	10	16K ck	4.1 mS
1	11	16K ck	65 mS

Esta opción se selecciona con CKSEL = "1001". Si se programa al fusible CKOPT se habilitan dos capacitores internos de 36 pF, en caso contrario, deben conectarse dos capacitores externos con la configuración mostrada en la figura 2.20.

Con esta opción para el reloj, los tiempos de establecimiento después de algún modo de bajo consumo y los retardos después de un reinicio son mostrados en la tabla 2.9.

Tabla 2.9 Tiempos de establecimiento y retardos con un cristal de baja frecuencia externo

SUT [1:0]	Establecimiento después de un bajo consumo	Retardo después de un reset (V_{cc} = 5 V)
00	1 K ck	4.1 mS
01	1 K ck	65 mS
10	32 K ck	65 mS
11	Reservado	

2.8.3 Oscilador RC Externo

Esta es una alternativa de bajo costo si se requiere de una frecuencia diferente a las que maneja el oscilador interno y si esta frecuencia no es un factor determinante en el funcionamiento de un sistema, dado que el valor de R y C puede variar por sus tolerancias intrínsecas y en función de la temperatura. El circuito RC se conecta como se muestra en la figura 2.21, la frecuencia se determina como f = 1/(3RC) y el valor de C debe ser por lo menos de 22 pF.

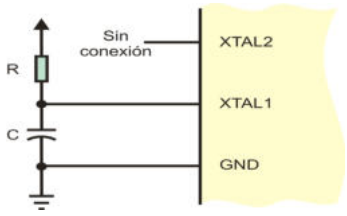


Figura 2.21 Oscilador RC Externo

Con este tipo de oscilador, se tienen 4 combinaciones de CKSEL, debe seleccionarse la que mejor se adapte a la frecuencia de trabajo. En la tabla 2.10 se muestra el rango de frecuencias para cada una de las combinaciones.

Tabla 2.10 Opciones para el empleo de un oscilador RC externo

CKSEL [3:0]	Rango de Frecuencias (MHz)
0101	0.1 – 0.9
0110	0.9 – 3.0
0111	3.0 – 8.0
1000	8.0 – 12.0

Para esta fuente de reloj, los tiempos de establecimiento después de algún modo de bajo consumo y los retardos después de un reinicio se muestran en la tabla 2.11.

Tabla 2.11 Tiempos de establecimiento y retardos con un oscilador RC externo

SUT [1:0]	Establecimiento después de un bajo consumo	Retardo después de un *reset* (V_{cc} = 5 V)
00	18 ck	-
01	18 ck	4.1 mS
10	18 ck	65 mS
11	6 ck	4.1 mS

2.8.4 Oscilador RC Calibrado Interno

Los microcontroladores ATMega8 y ATMega16 incluyen un oscilador interno que puede ser configurado para trabajar en frecuencias de 1, 2, 4 y 8 MHz. En la tabla 2.12 se muestran las combinaciones de los fusibles CKSEL para utilizar al oscilador interno. Se requiere que el dispositivo esté alimentado con 5 V y a una temperatura de 25 °C.

Tabla 2.12 Opciones para el empleo de un Oscilador RC interno

CKSEL [3:0]	Frecuencia Nominal (MHz)
0001	1.0
0010	2.0
0011	4.0
0100	8.0

Ésta es la mejor alternativa para la mayoría de aplicaciones, puesto que el oscilador ya está incluido en el MCU, con ello se minimiza el número de componentes externos y por lo tanto, el circuito impreso de la aplicación se reduce.

Con esta fuente de reloj, después de algún modo de bajo consumo se tiene un tiempo de establecimiento de 6 ciclos de reloj y después de un reinicio se tienen los retardos mostrados en la tabla 2.13.

Los dispositivos se comercializan con CKSEL = “0001” y SUT = “10”, por lo que inicialmente trabajan a una frecuencia de 1 MHz y con un retardo de 65 mS.

Tabla 2.13 Retardos con un oscilador RC calibrado interno y con un reloj externo

SUT [1:0]	Retardo después de un *reset* (V_{cc} = 5 V)
00	-
01	4.1 mS
10	65 mS
11	Reservado

Sin embargo, el voltaje de alimentación o la temperatura puede variar y esto va a modificar la frecuencia del oscilador interno, la cual también es afectada por las variaciones en los procesos de fabricación de cada dispositivo. Para compensar estas variaciones, en los AVR se ha incorporado un registro para la calibración del oscilador, el registro se denomina **OSCCAL** y sus bits son:

	7	6	5	4	3	2	1	0	
0x31	CAL7	CAL6	CAL5	CAL4	CAL3	CAL2	CAL1	CAL0	**OSCCAL**

Después de un *reset* en el registro **OSCCAL** se carga automáticamente el valor de calibración para 1 MHz, el cual está en el byte alto de la palabra ubicada en la dirección 0 de la firma del dispositivo. La firma del dispositivo es un conjunto de palabras que puede leerse desde un programador serial o paralelo, los bytes bajos contienen información que identifica al fabricante, el tamaño de memoria y al tipo de dispositivo. En los bytes altos se encuentran los valores de calibración para 1, 2, 4 y 8 MHz, en las direcciones 0, 1, 2 y 3, respectivamente.

Si un dispositivo opera con una frecuencia diferente a 1 MHz, debe cambiarse el valor de calibración. Para ello, se sugiere leer los valores de calibración de la firma del dispositivo e incluirlos como constantes en Flash o en EEPROM. Con estos valores se garantiza un error en la frecuencia menor al 3 %. En las notas de aplicación de Atmel se describen algunas técnicas para calibrar al oscilador en tiempo de ejecución, con las cuales es posible conseguir un error menor al 1 %, a cualquier voltaje y temperatura.

2.8.5 Reloj Externo

El MCU puede ser manejado por un generador de señales, con la combinación de “0000” para CKSEL. La salida del generador debe conectarse en la terminal XTAL1, dejando a la terminal XTAL2 sin conexión, como se muestra en la figura 2.22. Si se programa al fusible CKOPT, se habilita un capacitor de 36 pF entre XTAL1 y GND, con el propósito de eliminar ruido.

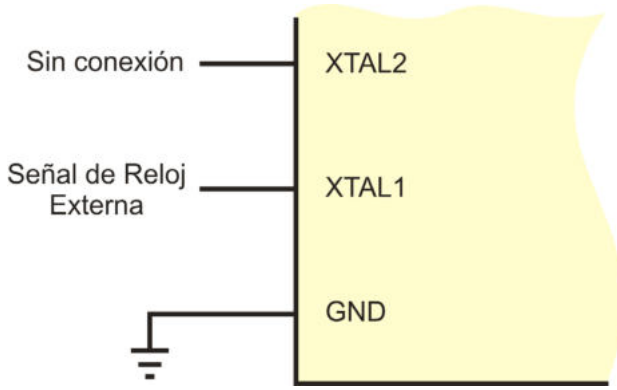


Figura 2.22 Reloj Externo

Con un reloj externo, el tiempo de establecimiento después de algún modo de bajo consumo es de 6 ciclos de reloj y los retardos después de un reinicio corresponden con los de la tabla 2.13.

2.9 Modos de Bajo Consumo de Energía

Los modos de bajo consumo permiten el ahorro de energía al "apagar" módulos de un MCU, cuyo uso no es requerido en un sistema. También se les conoce como modos de reposo y son adecuados si el sistema va a operar con baterías.

En la arquitectura AVR se simplifica el manejo de los modos de bajo consumo por la forma en que distribuye su señal de reloj, de manera que si un módulo no se va a requerir para alguna aplicación, simplemente se anula su señal de reloj, con lo cual el módulo no trabaja y por lo tanto, no consume energía.

Un ATMega8 tiene 5 modos de reposo y un ATMega16 tiene 6. En la tabla 2.14 se listan los modos de bajo consumo con los dominios de reloj que se mantienen activos, las fuentes presentes para el oscilador y los eventos que "despiertan" al MCU, haciendo que abandone el modo en que se encuentre. El último modo de la tabla 2.14 no está disponible en los ATMega8.

Tabla 2.14 Dominios de reloj activos en los diferentes modos de reposo y eventos que lo "despiertan"

Modo de Bajo Consumo de Energía	Reloj Activo					Oscilador		Eventos que despiertan al MCU					
	clk_{CPU}	clk_{FLASH}	clk_{IO}	clk_{ADC}	clk_{ASY}	Reloj Principal	Oscilador del temporizador	INT0, INT1, INT2	Interfaz de dos hilos (TWI)	Temporizador 2	EEPROM, Memoria de programa lista	ADC	Otros I/O
Modo ocioso (*idle*)			X	X	X	X	X	X	X	X	X	X	X
Reducción de ruido en el ADC				X	X	X	X	X	X	X	X	X	
Baja potencia								X	X				
Ahorro de potencia					X		X	X	X	X			
Modo de espera (*standby*)						X		X	X				
Modo de espera extendido					X	X	X	X	X	X			

Los diferentes modos de bajo consumo se describen a continuación:

- **Modo ocioso:** En este modo todos los recursos del MCU trabajan, pero la CPU no ejecuta instrucciones porque no tienen señal de reloj. El suministro del reloj principal y del oscilador del temporizador están activos. El hecho de mantener los recursos activos hace que, prácticamente cualquier evento de los diferentes recursos provoque una salida del modo de reposo.

- **Modo de reducción de ruido en el ADC:** En este modo únicamente trabaja el ADC y el oscilador asíncrono para el temporizador 2. Ambos suministros de reloj están activos, por ello, este modo es adecuado para aplicaciones que requieren el monitoreo de parámetros analógicos en periodos preestablecidos de tiempo. La salida del modo es posible por eventos en los recursos principales

- **Modo de baja potencia:** En este modo no hay reloj en los módulos de recursos y tampoco están activas las fuentes de oscilación, por lo tanto, es el modo con el menor consumo de energía. El MCU puede ser reactivado por eventos en la interfaz de dos hilos o por las interrupciones externas. Una posible aplicación para este modo es un control remoto, porque casi siempre está inactivo, sólo cuando se presiona una tecla del control remoto el MCU se despierta, emite el código seleccionado y regresa al modo de reposo.

- **Modo de ahorro de potencia:** En este modo sólo se tiene al reloj asíncrono, con su correspondiente oscilador. Por lo que prácticamente sólo se mantiene activo el temporizador 2, sincronizado con una fuente de reloj externa. Esto hace al modo ideal para aplicaciones que involucren un reloj de tiempo real. La salida del modo es posible con eventos en la interfaz de dos hilos, interrupciones externas o del temporizador 2.

- **Modo de espera:** Este modo es muy similar al modo de baja potencia, la única diferencia es que en este modo se mantiene el suministro del reloj principal. Con ello, el dispositivo despierta en 6 ciclos de reloj.

- **Modo de espera extendido:** Este modo es muy similar al modo de ahorro de potencia, la única diferencia es que en este modo también está activo el suministro del reloj principal. Este modo no está disponible en el ATMega8.

Para seleccionar uno de los modos debe configurarse al registro de Control del MCU (**MCUCR***, MCU Control Register*), los bits de este registro son:

	7	6	5	4	3	2	1	0	
0x35	SE	SM2	SM1	SM0	ISC11	ISC10	ISC01	ISC00	**MCUCR**

- **Bit 7 – SE: Habilitador del modo de reposo (*SE, Sleep Enable*)**

  Este bit se pone en alto después de ingresar a cualquiera de los modos de reposo, lo cual se consigue con la ejecución de la instrucción **`SLEEP`**, pero antes, debe seleccionarse el modo de reposo deseado.

- **Bits 6 al 4 – SM[2:0]: Bits de Selección del modo de reposo**

  En la tabla 2.15 se muestra la combinación de estos bits para seleccionar los diferentes modos de bajo consumo de energía.

- **Bits 3 al 0 – No están relacionados con los modos de reposo**

Los modos de espera sólo pueden ser empleados con resonadores cerámicos o cristales externos. La combinación "111" de la tabla 2.15 está sin uso en un ATMega8, porque no cuenta con el modo de espera extendido. En un ATMega16, la posición de los bits **SE** y **SM2** está intercambiada, **SM2** es el bit 7 y **SE** está en la posición 6.

Tabla 2.15 Selección del modo de reposo

SM2	SM1	SM0	Modo de reposo
0	0	0	Modo ocioso (*idle*)
0	0	1	Reducción de ruido en el ADC
0	1	0	Baja potencia
0	1	1	Ahorro de potencia
1	0	0	Reservado
1	0	1	Reservado
1	1	0	Modo de espera (*standby*)
1	1	1	Modo de espera extendido

2.10 Ejercicios

1. Complemente la tabla 2.16 con las características de los microcontroladores bajo estudio.

Tabla 2.16 Características de microcontroladores

	ATMega8	ATMega16
Cantidad de Memoria de Código:		
Cantidad de RAM (de propósito general):		
Cantidad de EEPROM:		
Registros de Propósito General:		
Registros I/O:		

2. Explique las ventajas de que el núcleo, en los AVR, trabaje en una segmentación de 2 etapas.
3. ¿Por qué se dice que los AVR están optimizados para trabajar con código C compilado?

4. Complemente la tabla 2.17 con el estado de las banderas, después de la ejecución de algunas instrucciones.

Tabla 2.17 Estado de las banderas

`LDI R16, 0x5A`
`LDI R17, 0x93`
`ADD R16, R17`
				`LDI R16, 0x4E`
				`LDI R17, 0xB2`
				`ADD R16, R17`
								`LDI R16, 0xA5`
								`LDI R17, 0x9B`
								`ADD R16, R17`			

Z =	H =	V =	C =	Z =	H =	V =	C =	Z =	H =	V =	C =

5. Explique para qué se emplean los Registros I/O en los microcontroladores AVR.
6. Indique qué Registros I/O se emplean para el acceso a la EEPROM y el objetivo de cada uno de ellos.
7. Muestre la organización de una terminal en un puerto y explique la funcionalidad de cada uno de los registros que se requieren para su manejo.
8. Defina qué es una interrupción y explique por qué es conveniente el uso de interrupciones.
9. Los microcontroladores AVR pueden ser operados por una gama muy amplia de osciladores ¿Por qué es conveniente incorporar esta flexibilidad y no únicamente manejarlos con un solo tipo de oscilador?
10. Describa tres fuentes de inicialización (o *reset*).
11. Explique el objetivo de los modos de bajo consumo de energía e indique cómo ingresar y salir de cualquiera de estos modos de operación.

3. Programación de los Microcontroladores

En este capítulo se describen los aspectos relacionados con la programación de los microcontroladores, se revisa el repertorio de instrucciones en ensamblador, los modos de direccionamiento, aspectos para programar en lenguaje C y la forma en que se puede vincular al lenguaje ensamblador con el lenguaje C.

3.1 Repertorio de Instrucciones

Aunque la arquitectura es tipo RISC, el repertorio de instrucciones es de un tamaño considerable, para el ATMega8 se tienen 128 instrucciones y para el ATMega16 son 131. Por su funcionalidad, las instrucciones están organizadas en 5 grupos:

- Instrucciones aritméticas y lógicas (28)
- Instrucciones para el control de flujo (Bifurcaciones) (34 en el ATMega8 y 36 en el ATMega16)
- Instrucciones de transferencia de datos (35)
- Instrucciones para el manejo de bits (28)
- Instrucciones especiales (3 en el ATMega8 y 4 en el ATMega16)

La mayoría de las instrucciones son de 16 bits. En las siguientes secciones se presentan algunas características propias de cada uno de los diferentes grupos de instrucciones, el apéndice B contiene una referencia rápida del repertorio completo.

3.1.1 Instrucciones Aritméticas y Lógicas

Este grupo incluye instrucciones para las operaciones básicas de suma y resta, con diferentes variantes, las cuales se muestran en la tabla 3.1.

Tabla 3.1 Instrucciones para las operaciones básicas de suma y resta

Instrucción		Descripción	Operación	Banderas
ADD	Rd, Rs	Suma sin acarreo	Rd = Rd + Rs	Z,C,N,V,H,S
ADC	Rd, Rs	Suma con acarreo	Rd = Rd + Rs + C	Z,C,N,V,H,S
ADIW	Rd, k	Suma constante a palabra	[Rd + 1:Rd] = [Rd + 1:Rd] + k	Z,C,N,V,S
SUB	Rd, Rs	Resta sin acarreo	Rd = Rd - Rs	Z,C,N,V,H,S
SUBI	Rd, k	Resta constante	Rd = Rd – k	Z,C,N,V,H,S
SBC	Rd, Rs	Resta con acarreo	Rd = Rd – Rs – C	Z,C,N,V,H,S
SBCI	Rd, k	Resta constante con acarreo	Rd = Rd – k – C	Z,C,N,V,H,S
SBIW	Rd, k	Resta constante a palabra	[Rd + 1:Rd] = [Rd + 1:Rd] - k	Z,C,N,V,S

Todas las instrucciones de suma y resta modifican las banderas de cero (**Z**), acarreo (**C**), negado(**N**), sobreflujo (**V**) y signo (**S**). Exceptuando a las instrucciones que operan

sobre palabras, el resto también modifica la bandera de acarreo del nibble menos significativo (**H**). Las instrucciones que operan sobre datos de 8 bits se ejecutan en 1 ciclo de reloj, las otras en 2 ciclos de reloj.

Las sumas y restas de 8 bits son entre registros, con o sin acarreo. No hay una suma de 8 bits con una constante, pero si se requiere, puede hacerse una resta con el negado de la constante que se desea sumar, si hay una resta con una constante de 8 bits, por ejemplo, para sumar 10 a R17 puede emplearse la instrucción **`SUBI R17, -10`**. Por el formato de las instrucciones, el cual se revisa en la siguiente sección, con los modos de direccionamiento, las instrucciones que involucran constantes sólo son aplicables a los registros R16 a R31.

Para la suma y resta de 16 bits, el segundo operando es una constante entre 0 y 63. La palabra de 16 bits se forma por 2 registros consecutivos entre R24 y R31, en la instrucción puede especificarse al registro menos significativo del par (**`ADIW R26, k`**), a ambos registros (**`ADIW R27:R26, k`**) o si es aplicable, el nombre de un registro de 16 bits (**`ADIW X, k`**).

El grupo también cuenta con instrucciones aritméticas más complejas, como multiplicaciones entre enteros y fracciones, éstas se listan en la tabla 3.2, todas se ejecutan en 2 ciclos de reloj.

Tabla 3.2 Instrucciones para las operaciones de multiplicación

Instrucción	Descripción	Operación	Banderas
MUL Rd, Rs	Multiplicación sin signo	R1:R0 = Rd * Rs	Z,C
MULS Rd, Rs	Multiplicación con signo	R1:R0 = Rd * Rs	Z,C
MULSU Rd, Rs	Multiplicación de un número con signo y otro sin signo	R1:R0 = Rd * Rs	Z,C
FMUL Rd, Rs	Multiplicación fraccional sin signo	R1:R0 = (Rd * Rs) << 1	Z,C
FMULS Rd, Rs	Multiplicación fraccional con signo	R1:R0 = (Rd * Rs) << 1	Z,C
FMULSU Rd, Rs	Multiplicación fraccional de un número con signo y otro sin signo	R1:R0 = (Rd * Rs) << 1	Z,C

Puesto que los operandos son de 8 bits, el resultado a lo más es de 16 bits. Todas las instrucciones dejan el resultado en R1 y R0 (en R1 la parte alta). Para las instrucciones con enteros, los operandos pueden ser interpretados como números sin signo o con signo. En la tabla 3.3 se muestran ejemplos que ilustran la diferencia entre instrucciones.

Tabla 3.3 Ejemplos de multiplicaciones con enteros

R16	R17	Instrucción	R1:R0	Valores en Decimal		
				R16	R17	R1:R0
0000 0010	1111 1111	MUL R16, R17	0000 0001 1111 1110	2	255	510
0000 0010	1111 1111	MULS R16, R17	1111 1111 1111 1110	2	-1	-2
0000 0010	1111 1111	MULSU R16, R17	0000 0001 1111 1110	2	255	510
1111 1111	1111 1111	MUL R16, R17	1111 1110 0000 0001	255	255	65, 025
1111 1111	1111 1111	MULS R16, R17	0000 0000 0000 0001	-1	-1	1
1111 1111	1111 1111	MULSU R16, R17	1111 1111 0000 0001	-1	255	-255

En la tabla 3.3 se observa como las instrucciones interpretan a los operandos de diferentes maneras. El resultado debe ser interpretado de acuerdo con el tipo de instrucción. Para la instrucción **MULSU** el primer operando corresponde con un número con signo, el segundo como un número sin signo y para que el resultado sea correcto, debe interpretarse como un número con signo.

La multiplicación fraccionaria utiliza una notación de punto fijo, bajo un esquema de 1.7 para los operandos y 1.15 para el resultado. En los operandos se tiene 1 dígito para la parte entera y 7 para la parte fraccionaria. Lo natural sería que 1.7 x 1.7 = 2.14, por eso es que en la tabla 3.2 aparece un desplazamiento a la izquierda, al desplazar el punto decimal el resultado queda en 1.15.

Para comprender la notación se tiene el siguiente ejemplo:

$$1110\ 0000_2 = 1.11_2 = 1x2^0 + 1x2^{-1} + 1x2^{-2} = 1 + 0.5 + 0.25 = 1.75$$

No obstante, si el número anterior es interpretado como un número con signo, el número es negativo porque su bit más significativo es 1, para obtener la magnitud se obtiene el complemento a 2, que corresponde con: 0001 1111 + 1 = 0010 0000. La magnitud es:

$$0010\ 0000_2 = 0.01_2 = 1x2^{-2} = 0.25$$

Por lo tanto, el número $1110\ 0000_2$ también puede corresponder con - 0.25, dependiendo de la instrucción que se utilice.

En la tabla 3.4 se tienen algunos ejemplos del uso de las instrucciones de multiplicación fraccional, considerando números sin signo y con signo, con sus interpretaciones en decimal.

Tabla 3.4 Ejemplos de multiplicaciones con fracciones en punto fijo

R16	R17	Instrucción	R1:R0	Valores en Decimal		
				R16	R17	R1:R0
0100 0000	1100 0000	FMUL R16, R17	0110 0000 0000 0000	0.5	1.5	0.75
0100 0000	1100 0000	FMULS R16, R17	1110 0000 0000 0000	0.5	-0.5	-0.25
0100 0000	1100 0000	FMULSU R16, R17	0110 0000 0000 0000	0.5	1.5	0.75
1100 0000	1100 0000	FMUL R16, R17	0010 0000 0000 0000[1]	1.5	1.5	2.25[1]
1100 0000	1100 0000	FMULS R16, R17	0010 0000 0000 0000	-0.5	-0.5	0.25
1100 0000	1100 0000	FMULSU R16, R17	1010 0000 0000 0000	-0.5	1.5	-0.75

En el 4º renglón de la tabla (nota[1]) el resultado en binario parece incorrecto, esto se debe a que el 2 no puede representarse con la notación 1.15, por lo que queda un 0 en la parte entera y se genera un bit de acarreo ($2 = 10_2$). Para la instrucción **FMULSU** ocurre algo similar que en la versión de enteros (**MULSU**) en cuanto a la interpretación de los números, el primer operando corresponde con un número con signo, el segundo como un número sin signo y el resultado es un número con signo.

En este grupo también se cuenta con las instrucciones lógicas binarias que se muestran en la tabla 3.5 (requieren 2 operandos). Estas instrucciones se aplican bit a bit, se ejecutan en 1 solo ciclo de reloj y modifican a las banderas **Z**, **N**, **V** y **S**.

La **EOR** (OR exclusiva) sólo puede aplicarse entre registros, las operaciones **AND** y **OR** además pueden aplicarse entre un registro con una constante.

Tabla 3.5 Instrucciones lógicas binarias

Instrucción		Descripción	Operación
AND	Rd, Rs	Operación lógica AND	Rd = Rd AND Rs
ANDI	Rd, k	Operación lógica AND con una constante	Rd = Rd AND k
OR	Rd, Rs	Operación lógica OR	Rd = Rd OR Rs
ORI	Rd, k	Operación lógica OR con una constante	Rd = Rd OR k
EOR	Rd, Rs	Operación lógica OR Exclusiva	Rd = Rd XOR Rs

Las operaciones lógicas por lo general se utilizan para enmascarar la información, una máscara consiste en modificar algunos bits de un byte, poniéndolos en alto o en bajo, sin modificar al resto. Las máscaras usualmente se realizan con operaciones **AND** y **OR**, pero por su extenso uso, el grupo incluye dos instrucciones dedicadas, las cuales se muestran en la tabla 3.6, en donde una constante especifica los bits a modificar en un registro. Estas instrucciones se ejecutan en 1 ciclo de reloj y modifican a las banderas **Z**, **C**, **N**, **V** y **S**; aunque por su enfoque, el valor de las banderas generalmente es ignorado.

Tabla 3.6 Instrucciones para enmascarar información

Instrucción		Descripción	Operación
SBR	Rd, k	Pone en alto los bits indicados en la constante	Rd = Rd OR k
CBR	Rd, k	Pone en bajo los bits indicados en la constante	Rd = Rd AND (0xFF – k)

El grupo se complementa con 7 instrucciones unarias (1 operando) con funciones diversas, las cuales se muestran en la tabla 3.7. Todas se ejecutan en 1 ciclo de reloj y las banderas que modifican difieren entre instrucciones.

Tabla 3.7 Instrucciones aritméticas o lógicas unarias

Instrucción		Descripción	Operación	Banderas
COM	Rd	Complemento a 1	Rd = 0xFF – Rd	Z,C,N,V,S
NEG	Rd	Negado o complemento a 2	Rd = 0x00 – Rd	Z,C,N,V,H,S
INC	Rd	Incrementa un registro	Rd = Rd + 1	Z,N,V,S
DEC	Rd	Disminuye un registro	Rd = Rd – 1	Z,N,V,S
TST	Rd	Evalúa un registro	Rd = Rd AND Rd	Z,C,N,V,S
CLR	Rd	Limpia un registro (pone en bajo)	Rd = 0x00	Z,C,N,V,S
SER	Rd	Ajusta un registro (pone en alto)	Rd = 0xFF	Ninguna

La instrucción **TST** no modifica al registro destino, puede usarse para evaluar si el registro es cero, evaluando la bandera **Z** después de su ejecución.

3.1.2 Instrucciones para el Control de Flujo

Durante la ejecución de un programa, el control de flujo se cambia al modificar el contenido del contador del programa (registro PC), para ello, este grupo cuenta con diversas instrucciones: saltos incondicionales, condicionales, llamadas y retornos de rutinas. En la tabla 3.8 se muestran las instrucciones de saltos incondicionales, las cuales no modifican banderas.

Tabla 3.8 Saltos incondicionales

Instrucción	Descripción	Operación
RJMP k	Salto relativo	PC = PC + k + 1
IJMP	Salto indirecto	PC = Z
JMP k	Salto absoluto	PC = k

Los saltos relativo e indirecto se ejecutan en 2 ciclos de reloj, el salto absoluto se ejecuta en 3 ciclos. El salto absoluto no está implementado en el ATMega8, no se requiere debido a que un salto relativo es suficiente para direccionar todo su espacio.

La instrucción **`RJMP`** es relativa con respecto a PC + 1, a partir de ahí se bifurca hacia adelante o atrás, utilizando una constante positiva o negativa. Los saltos relativos permiten crear código "reubicable", es decir, código que puede moverse a cualquier dirección, sin necesidad de hacer ajustes en la dirección destino, por lo tanto, las rutinas que sólo incluyen bifurcaciones relativas pueden ser compartidas como código máquina. Esto no se puede hacer con saltos absolutos, dado que hacen referencia a una dirección específica. Los saltos indirectos están enfocados a tablas de saltos, las cuales pueden resultar de codificar una estructura del tipo *switch-case*, en lenguaje C.

Relacionado con el manejo de rutinas, se tienen las instrucciones que se muestran en la tabla 3.9. Para las llamadas se tienen 3 instrucciones bajo el mismo esquema que los saltos incondicionales, incluyendo una llamada absoluta (**`CALL`**) que no está en el ATMega8.

Tabla 3.9 Instrucciones para el manejo de rutinas

Instrucción	Descripción	Operación
RCALL k	Llamada relativa a una rutina	PC = PC + k + 1, PILA ← PC + 1
ICALL	Llamada indirecta a una rutina	PC = Z, PILA ← PC + 1
CALL k	Llamada absoluta a una rutina	PC = k, PILA ← PC + 1
RET	Retorno de una rutina	PC ← PILA
RETI	Retorno de rutina de interrupción	PC ← PILA, I = 1

La llamada a una rutina también es un salto incondicional, con el agregado de que en la pila se almacena el valor de PC + 1, el cual corresponde con la dirección de la instrucción que sigue a la llamada, esta dirección se recupera de la pila y se escribe en el PC con la instrucción **`RET`** o **`RETI`**.

La instrucción **RETI** es el retorno de una ISR (en la sección 2.6 se describió el funcionamiento de las interrupciones), por lo tanto, con esta instrucción también se pone en alto al bit **I** del registro de estado (**SREG**), para habilitar nuevamente al sistema de interrupciones.

Las llamadas relativa e indirecta a rutinas se ejecutan en 3 ciclos de reloj, la llamada absoluta y los retornos se ejecutan en 4. Esta cantidad de ciclos de ejecución se debe a que requieren de un acceso a SRAM por los respaldos y recuperaciones en la pila de datos y un ajuste en el apuntador de pila (registro **SP**).

Dentro del grupo se encuentran 3 instrucciones para la comparación de datos, éstas se muestran en la tabla 3.10. Aunque no modifican el flujo de ejecución, modifican las banderas **Z**, **C**, **N**, **V**, **H** y **S**, en el registro estado, y el valor de estas banderas es la base para las instrucciones de brincos condicionales que se muestran en la tabla 3.11.

Tabla 3.10 Instrucciones para comparar datos

Instrucción	Descripción	Operación
CP Rd, Rs	Compara dos registros	Rd – Rs
CPC Rd, Rs	Compara registros con acarreo	Rd – Rs – C
CPI Rd, k	Compara un registro con una constante	Rd – k

Las instrucciones de la tabla 3.10 realizan la comparación con base en una resta, pero sin almacenar el resultado. Todas se ejecutan en 1 ciclo de reloj. En los programas es frecuente encontrar una instrucción de comparación seguida de un salto condicional (tabla 3.11).

Tabla 3.11 Instrucciones de brincos condicionales

Instrucción	Descripción	Operación
BRBS s, k	Brinca si el bit s del registro estado está en alto	si(SREG(s) == 1) PC = PC + k + 1
BRBC s, k	Brinca si el bit s del registro estado está en bajo	si (SREG(s) == 0) PC = PC + k + 1
BRIE k	Brinca si las interrupciones están habilitadas	si (I == 1) PC = PC + k + 1
BRID k	Brinca si las interrupciones están inhabilitadas	si (I == 0) PC = PC + k + 1
BRTS k	Brinca si el bit T está en alto	si (T == 1) PC = PC + k + 1
BRTC k	Brinca si el bit T está en bajo	si (T == 0) PC = PC + k + 1
BRHS k	Brinca si hubo acarreo del nibble bajo al alto	si (H == 1) PC = PC + k + 1
BRHC k	Brinca si no hubo acarreo del nibble bajo al alto	si (H == 0) PC = PC + k + 1
BRGE k	Brinca si es mayor o igual que (con signo)	si (S == 0) PC = PC + k + 1
BRLT k	Brinca si es menor que (con signo)	si (S == 1) PC = PC + k + 1
BRVS k	Brinca si hubo sobreflujo aritmético	si (V == 1) PC = PC + k + 1
BRVC k	Brinca si no hubo sobreflujo aritmético	si (V == 0) PC = PC + k + 1
BRMI k	Brinca si es negativo	si (N == 1) PC = PC + k + 1
BRPL k	Brinca si no es negativo	si (N == 0) PC = PC + k + 1
BREQ k	Brinca si los datos son iguales	si (Z == 1) PC = PC + k + 1
BRNE k	Brinca si los datos no son iguales	si (Z == 0) PC = PC + k + 1
BRSH k	Brinca si es mayor o igual	si (C == 0) PC = PC + k + 1
BRLO k	Brinca si es menor	si (C == 1) PC = PC + k + 1
BRCS k	Brinca si hubo acarreo	si (C == 1) PC = PC + k + 1
BRCC k	Brinca si no hubo acarreo	si (C == 0) PC = PC + k + 1

Los brincos condicionales pueden tardar 1 ó 2 ciclos de reloj en su ejecución, 1 ciclo si el brinco no se realizó, porque la siguiente instrucción ya está en la etapa de captura y 2 ciclos cuando el brinco se realiza, porque se debe anular a la instrucción que está en la etapa de captura y continuar con la siguiente después del brinco. Estos brincos son relativos al PC, es decir, la constante de la instrucción que se suma al PC + 1 puede ser positiva o negativa, para bifurcar hacia adelante o atrás de la instrucción del brinco.

En la tabla 3.11 hay 20 instrucciones y todas evalúan los bits del registro de Estado (**SREG**), sin embargo, sólo en las 2 primeras se especifica el número del bit a evaluar (bit s, entre 0 y 7), ya que en las siguientes 18 el número del bit queda implícito en la instrucción, podría decirse que son versiones particularizadas de las primeras 2 instrucciones. El nombre y descripción de cada instrucción está relacionado con la aplicación que se le puede dar a cada bandera, por eso se tienen 2 versiones para la bandera de acarreo. Estas 18 instrucciones se ordenaron de acuerdo con la disposición de los bits en el registro Estado (del bit más significativo al menos significativo).

Otras instrucciones para bifurcaciones condicionales se muestran en la tabla 3.12, la diferencia es que estas instrucciones son para saltos pequeños, o "saltitos", que sólo excluyen a la instrucción siguiente cuando la condición se cumple. Por lo general, estas instrucciones están seguidas por un salto incondicional, para complementar el control de flujo en un programa.

Tabla 3.12 Instrucciones para saltitos condicionales

Instrucción		Descripción	Operación
CPSE	Rd, Rs	Un saltito si los registros son iguales	si(Rd == Rs) PC = PC + 2 ó 3
SBRS	Rs, b	Un saltito si el bit b del registro Rs está en alto	si(Rs(b) == 1) PC = PC + 2 ó 3
SBRC	Rs, b	Un saltito si el bit b del registro Rs está en bajo	si(Rs(b) == 0) PC = PC + 2 ó 3
SBIS	P, b	Un saltito si el bit b del registro P está en alto, P es un Registro I/O	si(I/O(P, b) == 1) PC = PC + 2 ó 3
SBIC	P, b	Un saltito si el bit b del registro P está en bajo, P es un Registro I/O	si(I/O(P, b) == 0) PC = PC + 2 ó 3

Si la condición es verdadera, las instrucciones pueden incrementar al PC con 2 ó 3, esto porque se desconoce si la instrucción que se omite ocupa 1 ó 2 palabras de 16 bits. Por la misma razón, para su ejecución pueden requerirse 2 ó 3 ciclos de reloj, dado que se debe evaluar si ya es una nueva instrucción o si aún es parte de la que intenta omitirse. Si la condición no se cumple, se continúa con la siguiente instrucción (PC + 1) y por lo tanto, sólo se requiere de 1 ciclo de reloj.

Las últimas 2 instrucciones son frecuentemente utilizadas para monitorear los puertos o para evaluar el estado de otros recursos, dado que todos los recursos son manejados a través de los Registros I/O (sección 2.4.1.1).

3.1.3 Instrucciones de Transferencia de Datos

El grupo incluye instrucciones para diferentes tipos de transferencias. Puesto que la arquitectura es del tipo Registro-Registro, las transferencias están centradas en los 32 registros de propósito general. En la figura 3.1 se ilustra gráficamente este esquema, en donde se puede observar que, por ejemplo, si se quiere llevar una constante a memoria, primero se debe ubicar la constante en uno de los registros, para luego transferir de registro a memoria.

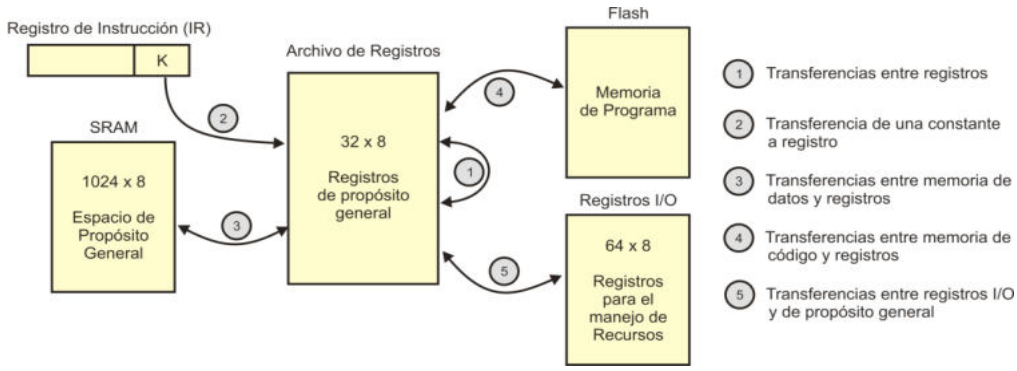


Figura 3.1 Transferencias de datos permitidas en la arquitectura AVR

Las transferencias de datos no modifican las banderas del registro Estado. Para transferencias entre registros sólo se tienen las 2 instrucciones mostradas en la tabla 3.13, una para mover un byte y la otra para mover una palabra de 16 bits. En las transferencias de palabras, para cada operando, se debe especificar un registro con número par, haciendo referencia a éste, como byte menos significativo, y al siguiente. Ambas instrucciones se ejecutan en 1 ciclo de reloj.

Tabla 3.13 Transferencias entre registros

Instrucción	Descripción	Operación
MOV Rd, Rs	Copia un registro	Rd = Rs
MOVW Rd, Rs	Copia un par de registros	Rd + 1: Rd = Rs + 1: Rs, Rd y Rs registros pares

Sólo se requiere de una instrucción para transferir una constante a un registro, la constante es parte de la instrucción por lo que se toma del registro IR. La instrucción se muestra en la tabla 3.14 y su ejecución requiere sólo de 1 ciclo de reloj.

Tabla 3.14 Transferencia de una constante a un registro

Instrucción	Descripción	Operación
LDI Rd, k	Copia la constante en el registro	Rd = k

Para transferencias entre la memoria de datos y los registros se tiene una gama amplia de instrucciones, a la transferencia de memoria a registro se le conoce como una carga y a la transferencia de registro a memoria se le conoce como un almacenamiento.

En la figura 2.10 se ilustraron estas transferencias. En la tabla 3.15 se muestran las instrucciones para cargas y almacenamientos, en ambos casos se tiene sólo 1 instrucción que utiliza direccionamiento directo, es decir, en la instrucción se especifica la dirección de la localidad a la que se tiene acceso, y 11 instrucciones con direccionamiento indirecto, empleando a uno de los registros de 16 bits (X, Y o Z) como apuntador, en la instrucción se especifica al apuntador.

La variedad de instrucciones se debe a que además de la transferencia, en la mayoría de los casos se modifica al apuntador. Todas las transferencias con SRAM, cargas o almacenamientos, directas o indirectas, requieren de 2 ciclos de reloj para su ejecución.

Tabla 3.15 Transferencias entre memoria de datos y registros (Cargas y Almacenamientos)

Instrucción		Descripción	Operación
LDS	Rd, k	Carga directa de memoria	Rd = Mem[k]
LD	Rd, X	Carga indirecta de memoria	Rd = Mem[X]
LD	Rd, X+	Carga indirecta con post-incremento	Rd = Mem[X], X = X + 1
LD	Rd, -X	Carga indirecta con pre-decremento	X = X - 1, Rd = Mem[X]
LD	Rd, Y	Carga indirecta de memoria	Rd = Mem[Y]
LD	Rd, Y+	Carga indirecta con post-incremento	Rd = Mem[Y], Y = Y + 1
LD	Rd, -Y	Carga indirecta con pre-decremento	Y = Y - 1, Rd = Mem[Y]
LDD	Rd, Y + q	Carga indirecta con desplazamiento	Rd = Mem[Y + q]
LD	Rd, Z	Carga indirecta de memoria	Rd = Mem[Z]
LD	Rd, Z+	Carga indirecta con post-incremento	Rd = Mem[Z], Z = Z + 1
LD	Rd, -Z	Carga indirecta con pre-decremento	Z = Z - 1, Rd = Mem[Z]
LDD	Rd, Z + q	Carga indirecta con desplazamiento	Rd = Mem[Z + q]
STS	k, Rs	Almacenamiento directo en memoria	Mem[k] = Rs
ST	X, Rs	Almacenamiento indirecto en memoria	Mem[X] = Rs
ST	X+, Rs	Almacenamiento indirecto con post-incremento	Mem[X] = Rs, X = X + 1
ST	-X, Rs	Almacenamiento indirecto con pre-decremento	X = X - 1, Mem[X] = Rs
ST	Y, Rs	Almacenamiento indirecto en memoria	Mem[Y] = Rs
ST	Y+, Rs	Almacenamiento indirecto con post-incremento	Mem[Y] = Rs, Y = Y + 1
ST	-Y, Rs	Almacenamiento indirecto con pre-decremento	Y = Y - 1, Mem[Y] = Rs
STD	Y + q, Rs	Almacenamiento indirecto con desplazamiento	Mem[Y + q] = Rs
ST	Z, Rs	Almacenamiento indirecto en memoria	Mem[Z] = Rs
ST	Z+, Rs	Almacenamiento indirecto con post-incremento	Mem[Z] = Rs, Z = Z + 1
ST	-Z, Rs	Almacenamiento indirecto con pre-decremento	Z = Z - 1, Mem[Z] = Rs
STD	Z + q, Rs	Almacenamiento indirecto con desplazamiento	Mem[Z + q] = Rs

En el grupo hay otras dos instrucciones que también hacen transferencias entre registros y SRAM, éstas no se incluyeron en la tabla 3.15 porque están enfocadas a trabajar en el espacio destinado para la Pila, para insertar o extraer un dato en el tope de la pila, el cual está referido por el registro **SP**, que es el apuntador de pila. En la tabla 3.16 se muestran las instrucciones de acceso a la Pila, las cuales también se ejecutan en 2 ciclos de reloj.

Tabla 3.16 Transferencias entre memoria de datos y registros (acceso a la pila)

Instrucción	Descripción	Operación
PUSH Rs	Inserta a Rs en la pila	Mem[SP] = Rs, SP = SP – 1
POP Rd	Extrae de la pila y coloca en Rd	SP = SP + 1, Rd = Mem[SP]

Las transferencias con la memoria de código incluyen tanto cargas como almacenamientos. Las cargas son muy frecuentes, dado que en una aplicación conviene utilizar a la memoria de código para todos aquellos datos constantes, liberando con ello espacio en la SRAM. Se tienen 3 instrucciones de carga, se ejecutan en 3 ciclos de reloj y todas usan al registro Z como apuntador (direccionamiento indirecto).

Los almacenamientos son poco usados en aplicaciones ordinarias, porque significa cambiar datos que en principio fueron considerados constantes, como los parámetros de un sistema de control, el contenido de un mensaje para un LCD, etc. Sólo se tiene 1 instrucción para almacenar datos en la memoria de código (**SPM**), no obstante, debe considerarse que una memoria Flash se modifica por páginas y cuenta con mecanismos de seguridad para proteger su contenido. El uso de la instrucción **SPM** se describe en la sección 7.2.3.

En la tabla 3.17 se muestran estas instrucciones, cuando se emplean, debe tenerse en cuenta que la memoria de programa está organizada en palabras de 16 bits.

Tabla 3.17 Transferencias entre memoria de código y registros

Instrucción	Descripción	Operación
LPM	Carga indirecta de memoria de programa en R0	R0 = Flash[Z]
LPM Rd, Z	Carga indirecta de memoria de programa en Rd	Rd = Flash[Z]
LPM Rd, Z+	Carga indirecta con post-incremento	Rd = Flash[Z], Z = Z + 1
SPM	Almacenamiento indirecto en memoria de programa	Flash[Z] = R1:R0

Por último, para las transferencias entre Registros I/O y registros de propósito general se tienen las instrucciones mostradas en la tabla 3.18, en donde la variable P hace referencia a cualquiera de los 64 Registros I/O, sin importar a qué recurso pertenecen. Estas instrucciones se ejecutan en 1 ciclo de reloj y no modifican las banderas del registro de Estado.

Tabla 3.18 Transferencias entre Registros I/O y registros de propósito general

Instrucción	Descripción	Operación
IN Rd, P	Lee de un Registro I/O, deja el resultado en Rd	Rd = P
OUT P, Rs	Escribe en un Registro I/O, el valor de Rs	P = Rs

3.1.4 Instrucciones para el Manejo de Bits

En este grupo se encuentran las instrucciones para desplazamientos y rotaciones que se muestran en la tabla 3.19. Todas se ejecutan en 1 ciclo de reloj y modifican las banderas **Z**, **C**, **N**, **V** y **S** del registro **SREG**. Con excepción del desplazamiento aritmético a la derecha, las demás instrucciones funcionan con apoyo de la bandera de acarreo.

Tabla 3.19 Instrucciones para desplazamientos y rotaciones

Instrucción		Descripción	Operación
LSL	Rd	Desplazamiento lógico a la izquierda	C = Rd(7), Rd(n+1) = Rd(n), Rd(0) = 0
LSR	Rd	Desplazamiento lógico a la derecha	C = Rd(0), Rd(n) = Rd(n + 1), Rd(7) = 0
ROL	Rd	Rotación a la izquierda	C = Rd(7), Rd(n + 1) = Rd(n), Rd(0) = C
ROR	Rd	Rotación a la derecha	C = Rd(0), Rd(n) = Rd(n + 1), Rd(7) = C
ASR	Rd	Desplazamiento aritmético a la derecha	Rd(n) = Rd(n + 1), n = 0 .. 6

En la figura 3.2 se ilustran de manera gráfica los diferentes tipos de desplazamientos y rotaciones.

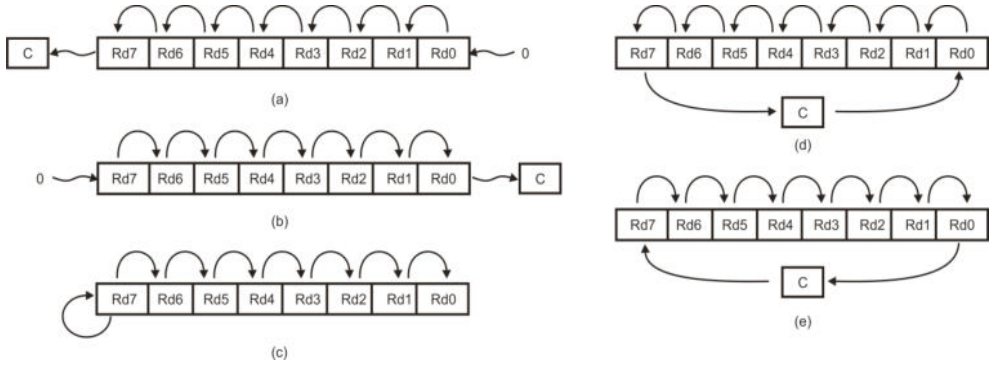


Figura 3.2 (a) Desplazamiento lógico a la izquierda, (b) a la derecha, (c) desplazamiento aritmético a la derecha, (d) rotación a la izquierda y (e) rotación a la derecha

Los desplazamientos lógicos insertan un 0 en el espacio generado, el desplazamiento a la izquierda equivale a una multiplicación por 2 y el desplazamiento a la derecha a una división entre 2. La diferencia entre un desplazamiento lógico y uno aritmético es que el último conserva el valor del bit en el espacio generado, con ello, el desplazamiento aritmético a la derecha equivale a una división entre 2 considerando números con signo, conservando el signo en el resultado. El desplazamiento aritmético a la izquierda no está implementado porque carece de significado.

Otra instrucción que es parte de este grupo es para intercambiar el nibble alto con el nibble bajo en un registro, ésta se muestra en la tabla 3.20. Se ejecuta en 1 ciclo de reloj y no modifica banderas.

Tabla 3.20 Instrucción para el intercambio de nibbles

Instrucción	Descripción	Operación
SWAP Rd	Intercambia nibbles en Rd	Rd(7..4) = Rd(3..0), Rd(3..0) = Rd(7..4)

También se incluyen instrucciones para modificar un bit en un Registro I/O, para ponerlo en alto o en bajo, siempre que esté permitido, dado que sólo los Registros I/O que están en el rango de 0x00 a 0x1F pueden ser manipulados por sus bits individuales. Estas instrucciones se muestran en la tabla 3.21, se ejecutan en 2 ciclos de reloj y tampoco modifican banderas.

Tabla 3.21 Instrucciones para modificar bits en los Registros I/O

Instrucción	Descripción	Operación
SBI P, b	Pone en alto al bit b del Registro P	P(b) = 1
CBI P, b	Pone en bajo al bit b del Registro P	P(b) = 0

El grupo incluye 2 instrucciones para transferencias de bits, una para cargar (*load*) del bit **T** a un bit de un registro de propósito general y otra para almacenar (*store*) un bit de un registro de propósito general en el bit **T**. El bit **T** está en el registro **SREG** y es empleado para respaldar el valor de un bit de un registro de propósito general. Estas instrucciones se muestran en la tabla 3.22, se ejecutan en 1 ciclo de reloj y es evidente que la instrucción de almacenamiento modifica al bit **T**.

Tabla 3.22 Instrucciones para transferencias de bits

Instrucción	Descripción	Operación
BTS Rs, b	Almacena al bit b de Rs en el bit T de SREG	T = Rs(b)
BTL Rd, b	Carga en el bit b de Rd desde el bit T de SREG	Rd(b) = T

El grupo se complementa con 18 instrucciones dedicadas a manipular los 8 bits del registro Estado. Éstas se muestran en la tabla 3.23, en las 2 primeras se especifica el número del bit a modificar, una es para poner al bit en alto y la otra para ponerlo en bajo. Las 16 restantes son una versión particular de las 2 primeras, son 8 pares y cada par está dedicado a uno de los bits del registro Estado, poniéndolo en alto o en bajo.

Todas se ejecutan en 1 ciclo de reloj y cada instrucción va a modificar la bandera del registro Estado que le corresponda. Las instrucciones en la tabla 3.23 están ordenadas de acuerdo con la ubicación de los bits en **SREG**, del bit más significativo al menos significativo.

Tabla 3.23 Instrucciones para manipular los bits del registro Estado

Instrucción	Descripción	Operación
BSET s	Pone en alto al bit s del registro Estado	SREG(s) = 1
BCLR s	Pone en bajo al bit s del registro Estado	SREG(s) = 0
SEI	Pone en alto al habilitador de interrupciones	I = 1

Instrucción	Descripción	Operación
CLI	Pone en bajo al habilitador de interrupciones	I = 0
SET	Pone en alto al bit de transferencias	T = 1
CLT	Pone en bajo al bit de transferencias	T = 0
SEH	Pone en alto a la bandera de acarreo del nibble bajo	H = 1
CLH	Pone en bajo a la bandera de acarreo del nibble bajo	H = 0
SES	Pone en alto a la bandera de signo	S = 1
CLS	Pone en bajo a la bandera de signo	S = 0
SEV	Pone en alto a la bandera de sobreflujo	V = 1
CLV	Pone en bajo a la bandera de sobreflujo	V = 0
SEN	Pone en alto a la bandera de negativo	N = 1
CLN	Pone en bajo a la bandera de negativo	N = 0
SEZ	Pone en alto a la bandera de cero	Z = 1
CLZ	Pone en bajo a la bandera de cero	Z = 0
SEC	Pone en alto a la bandera de acarreo	C = 1
CLC	Pone en bajo a la bandera de acarreo	C = 0

3.1.5 Instrucciones Especiales

Este grupo está integrado por 3 instrucciones para el ATMega8 y 4 para el ATMega16, son instrucciones que, por sus características, no pueden integrarse en los otros grupos. Estas instrucciones no modifican algún operando, su objetivo es auxiliar para la adecuada ejecución de un programa.

En la tabla 3.24 se muestran y describen estas instrucciones, la última de la lista no está en el ATMega8 porque no cuenta con el recurso JTAG.

Tabla 3.24 Instrucciones especiales

Instrucción	Descripción
NOP	No operación, empleada para forzar una espera de 1 ciclo de reloj
SLEEP	Introduce al MCU en el modo de bajo consumo previamente configurado
WDR	Reinicia al Watchdog Timer, para evitar reinicios por su desbordamiento
BREAK	Utilizada para depuración, desde el puerto JTAG

3.2 Modos de Direccionamiento

Los modos de direccionamiento son un aspecto fundamental cuando se diseña la arquitectura de un procesador, porque definen muchas de las características que se ven reflejadas cuando es puesto en marcha, características que involucran: la ubicación de los datos sobre los que operan las instrucciones, el tamaño de las constantes, los registros que se pueden considerar como operandos en una instrucción y el alcance de los saltos. En los microcontroladores AVR se observan 7 modos de direccionamiento:

1. Directo por registro
2. Directo a Registros I/O
3. Directo a memoria de datos
4. Indirecto a memoria de datos
5. Indirecto a memoria de código
6. Inmediato
7. Direccionamientos en bifurcaciones

En las siguientes secciones se describen e ilustran los diferentes modos de direccionamiento. Debe considerarse que para algunos modos se tienen diferentes variantes y que la mayoría de instrucciones sólo requieren de una palabra de 16 bits.

3.2.1 Direccionamiento Directo por Registro

En la instrucción se especifica el registro o registros que funcionan como operandos, ya que puede ser sólo uno o dos registros, este modo se ilustra en la figura 3.3, en donde se observa que se dispone de 5 bits para definir cada registro, por ello, bajo este modo de direccionamiento se puede utilizar a cualquiera de los 32 registros. Ejemplos de instrucciones que emplean sólo un registro son:

```
COM     R1
INC     R2
SER     R3
```

Ejemplos de instrucciones en donde se emplean dos registros son:

```
ADD     R0, R1
SUB     R2, R3
AND     R4, R5
MOV     R6, R7
```

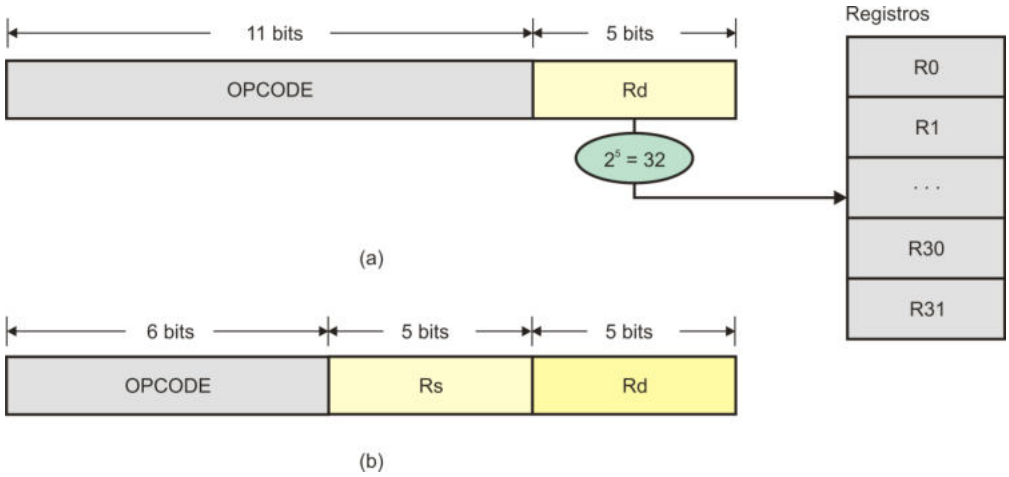


Figura 3.3 Direccionamiento directo empleando (a) un registro y (b) dos registros

3.2.2 Direccionamiento Directo a Registros I/O

Con este modo de direccionamiento se tiene acceso a los Registros I/O, que son la base para el manejo de los recursos en un MCU. En la figura 3.4 se ilustra el modo, se tienen 5 bits para el registro de propósito general (R), por lo que se puede utilizar cualquiera de ellos, y 6 bits para especificar al Registro I/O (P), también puede referirse a cualquiera de los 64 Registros I/O. Ejemplos de instrucciones con este modo de direccionamiento son:

```
OUT     PORTB, R13
IN      R15, PINA
```

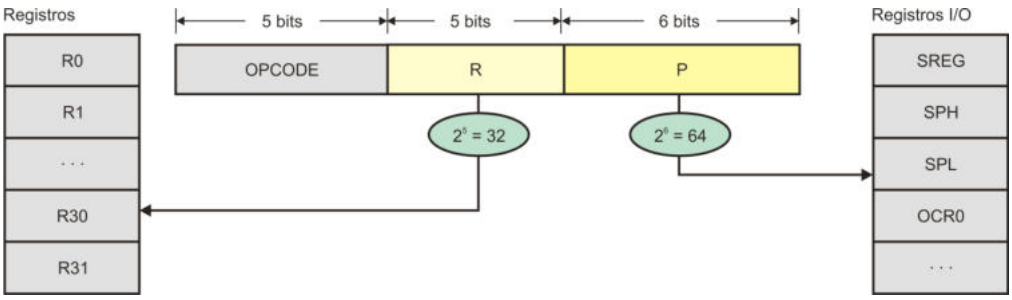


Figura 3.4 Direccionamiento directo a Registros I/O

3.2.3 Direccionamiento Directo a Memoria de Datos

La instrucción incluye la dirección de la localidad en memoria de datos a la que tiene acceso, las instrucciones con este modo requieren de 2 palabras de 16 bits, en la segunda palabra se indica la dirección del dato en memoria. Con 16 bits es posible direccionar hasta 64 K de datos, sin embargo, el límite real está determinado por el espacio disponible en cada dispositivo, para el ATMega8 y ATMega16 el límite superior es de 0x45F. En la figura 3.5 se ilustra este modo de direccionamiento. Ejemplos de instrucciones que lo utilizan son:

```
STS     0x0100, R5
LDS     R16, 0x0110
```

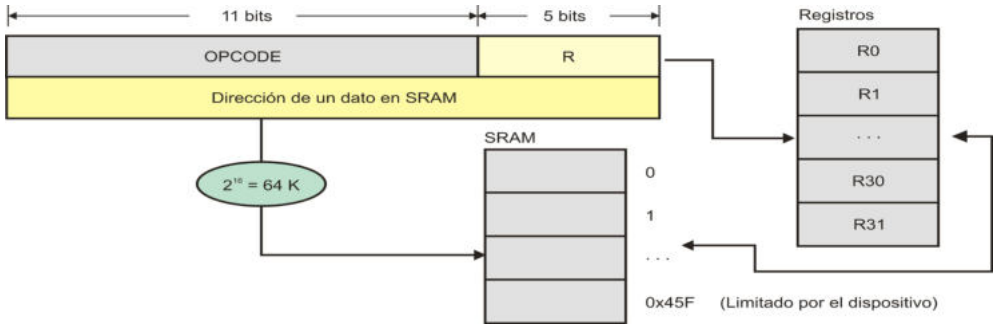


Figura 3.5 Direccionamiento directo a memoria de datos

3.2.4 Direccionamiento Indirecto a Memoria de Datos

Este modo de direccionamiento utiliza a los registros de 16 bits: X, Y o Z como apuntadores, el apuntador a usar queda implícito en el opcode, por lo que en la instrucción sólo se especifica al registro de propósito general con el que opera. El modo de direccionamiento se ilustra en la figura 3.6. Ejemplos de instrucciones que utilizan este modo son:

```
LD      R5, Y
ST      X, R11
```

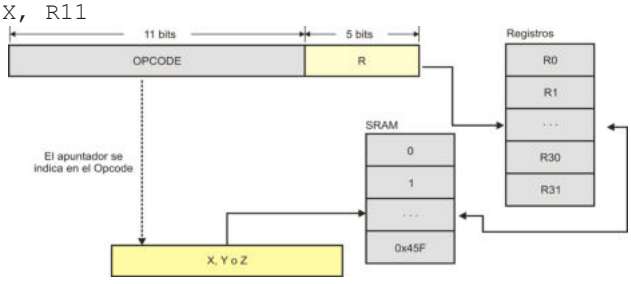


Figura 3.6 Direccionamiento indirecto a memoria de datos, X, Y o Z como apuntadores.

Existen dos variantes de este modo de direccionamiento en donde además se modifica al apuntador. La primera variante es con un post-incremento, donde primero obtiene el dato de memoria y después se incrementa al apuntador, y la segunda con un pre-decremento, en donde primero se disminuye al apuntador y luego se realiza el acceso a memoria. Esto también es aplicable a los 3 registros, X, Y o Z, estas ideas se muestran en la figura 3.7 y 3.8, respectivamente. Ejemplos de instrucciones con post-incrementos y pre-decrementos son:

```
LD      R5, Y+          ; Carga con post-incremento
ST      X+, R6          ; Almacenamiento con post-incremento
LD      R7, -X          ; Carga con pre-decremento
ST      -Z, R11         ; Almacenamiento con pre-decremento
```

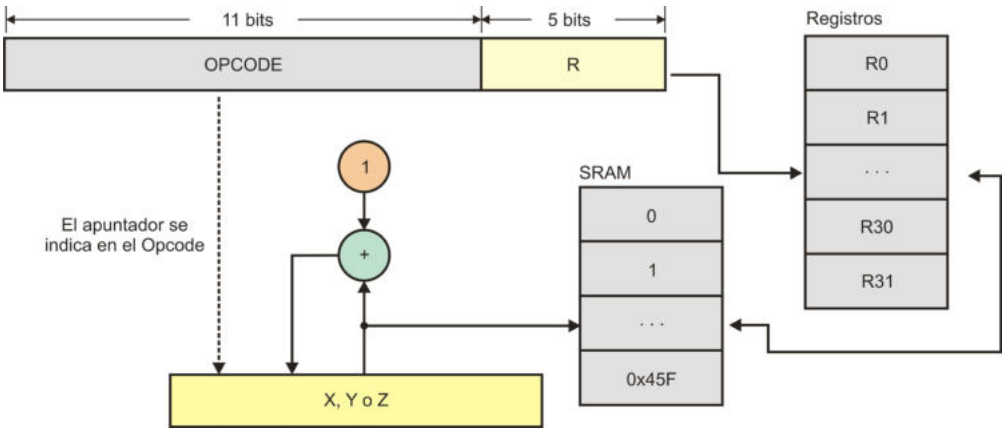


Figura 3.7 Direccionamiento indirecto con post-incremento

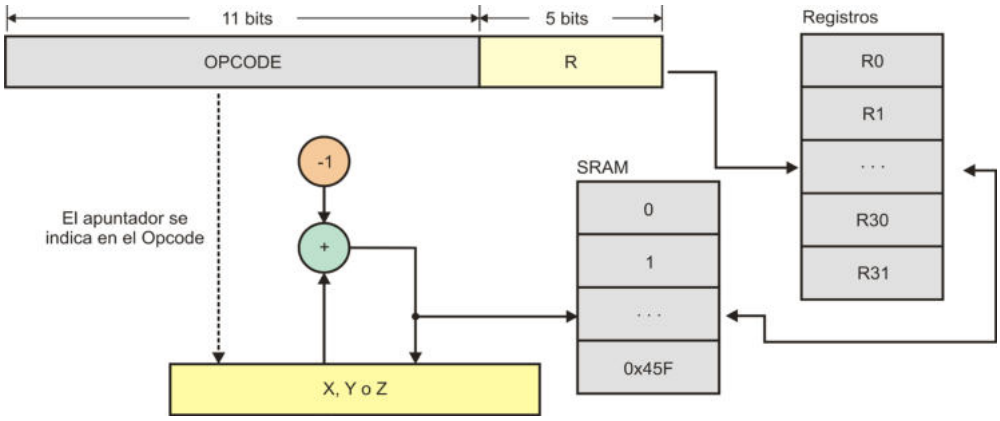


Figura 3.8 Direccionamiento indirecto con pre-decremento

Otra opción es el direccionamiento indirecto con desplazamiento, en este modo se tiene acceso a una dirección formada por la suma entre el apuntador y una constante, pero sin modificar al apuntador. Para la constante se dispone de 6 bits, por lo que debe estar en el rango de 0 a 63. Sólo es posible con los apuntadores Y o Z, esta idea se muestra en la figura 3.9. Ejemplos de instrucciones que usan este modo de direccionamiento son:

```
LDD     R5, Y+0x020
STD     Z+0x10, R11
```

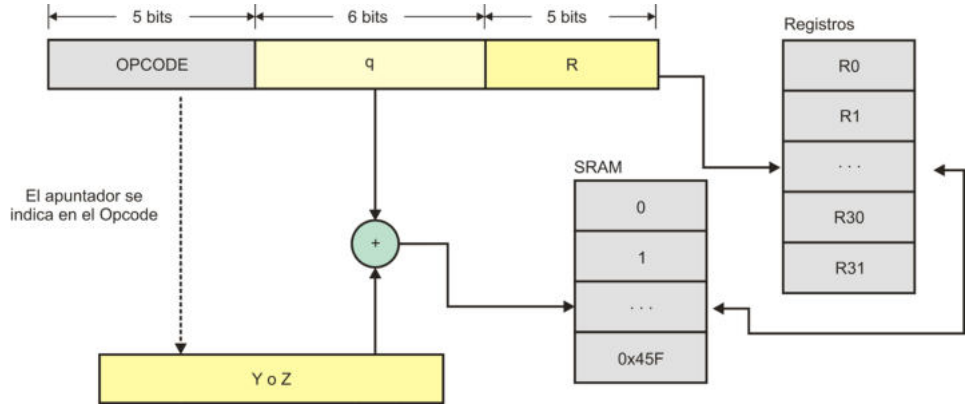


Figura 3.9 Direccionamiento indirecto con desplazamiento

3.2.5 Direccionamiento Indirecto a Memoria de Código

Este modo de direccionamiento utiliza al registro Z como apuntador, el cual queda implícito en el opcode. En la figura 3.10 se ilustra este modo de direccionamiento, ejemplos de instrucciones para acceso indirecto a memoria de código son:

```
LPM                     ; R0 y Z están implícitos en la instrucción
LPM     R3, Z
SPM                     ; R1:R0 y Z están implícitos
```

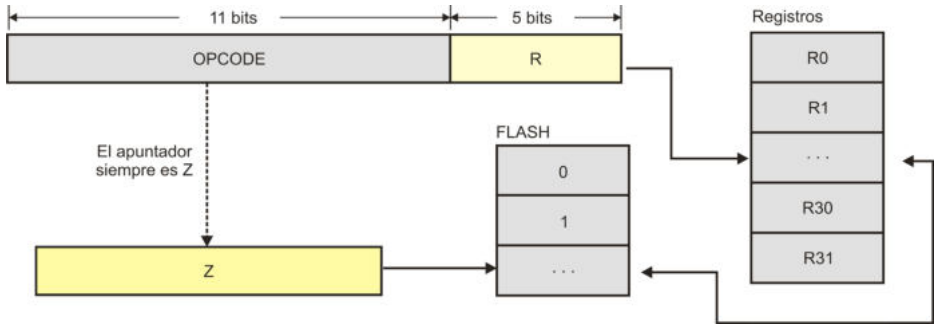


Figura 3.10 Direccionamiento indirecto a memoria de código

También existe una carga indirecta de memoria de código con post-incremento del apuntador Z, su comportamiento sería similar al mostrado en la figura 3.7.

3.2.6 Direccionamiento Inmediato

El direccionamiento inmediato es utilizado por aquellas instrucciones que incluyen una constante como uno de sus operandos. La constante es parte de la instrucción, debido a ello, uno de los operandos de la ALU se conoce de manera "inmediata", a esto se debe el nombre del modo de direccionamiento.

En la figura 3.11 se muestra cómo se organizan las instrucciones que utilizan este modo de direccionamiento. Se observa que al quedar disponibles únicamente 4 bits para definir el número de registro, sólo se puede referenciar a 16 de ellos, es por eso que las instrucciones con constantes sólo pueden incluir registros en el rango de R16 a R31.

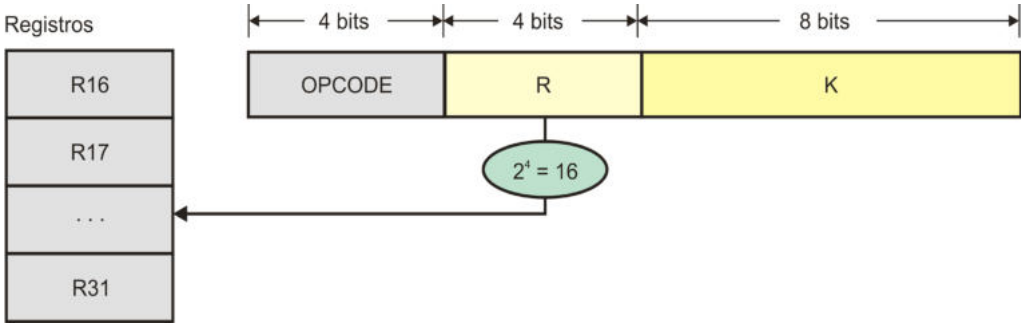


Figura 3.11 Direccionamiento inmediato

Ejemplos de instrucciones que utilizan direccionamiento inmediato son:

```
ANDI   R17, 0xF3
SUBI   R19, 0x12
ORI    R31, 0x03
LDI    R16, 0x25
```

Para la constante se dispone de 8 bits, puede estar en el rango de 0 a 255.

3.2.7 Direccionamientos en Bifurcaciones

Las bifurcaciones o saltos (condicionales o incondicionales) permiten cambiar el flujo secuencial durante la ejecución de un programa, esto se consigue modificando el valor del PC (contador del programa). Para estas instrucciones se tienen tres modos de direccionamiento: relativo, indirecto y absoluto.

3.2.7.1 Bifurcaciones con Direccionamiento Relativo

Las bifurcaciones con direccionamiento relativo incluyen una constante que se suma al PC, esta constante puede ser positiva o negativa, para bifurcar hacia adelante o atrás, con respecto a la ubicación de la instrucción de la bifurcación, en realidad son relativas a PC + 1, dado que el incremento del PC es automático. En la figura 3.12 se ilustra el funcionamiento de las bifurcaciones relativas.

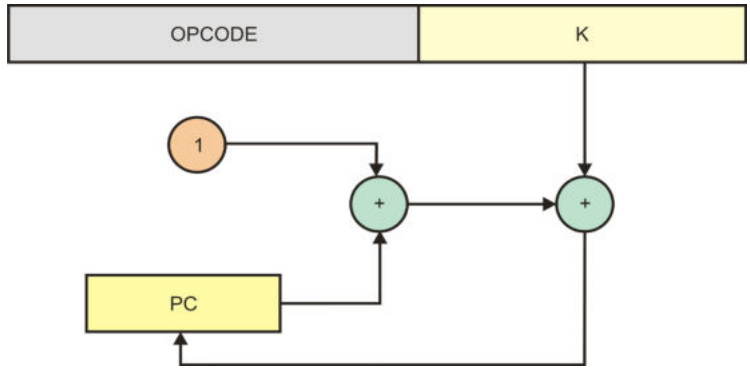


Figura 3.12 Bifurcaciones con direccionamiento relativo

El direccionamiento relativo se aplica en bifurcaciones incondicionales, en esos casos la constante K es de 12 bits. Ejemplos de estas instrucciones son:

```
RJMP   -20
RCALL  32
```

Las instrucciones para bifurcaciones condicionales sólo dejan disponibles 7 bits para la constante. Ejemplos de estas instrucciones son:

```
BREQ   15
BRNE   -10
BRGE   10
```

Cuando se escribe un programa se utilizan etiquetas, de acuerdo con su ubicación, el ensamblador calcula el valor de las constantes.

Existen bifurcaciones condicionales que sólo brincan la siguiente instrucción, estos "saltitos" también son relativos al PC, aunque el formato de la instrucción es más simple, dado que sólo se omite la ejecución de una instrucción.

3.2.7.2 Bifurcaciones con Direccionamiento Indirecto

El direccionamiento indirecto involucra el uso del registro Z como apuntador, un apuntador es una variable cuyo contenido es una dirección. En una bifurcación con este modo de direccionamiento, el contenido del apuntador Z se escribe en el PC, el cual también es un apuntador. Básicamente se realiza un reemplazo de direcciones, esto se ilustra en la figura 3.13.

Figura 3.13 Bifurcaciones con direccionamiento indirecto

Ejemplos de instrucciones que utilizan este modo de direccionamiento son:

```
IJMP
ICALL
```

El apuntador Z queda implícito en el opcode de la instrucción.

3.2.7.3 Bifurcaciones con Direccionamiento Absoluto

El direccionamiento es absoluto cuando se conoce la dirección destino de la bifurcación, la dirección forma parte de la misma instrucción. El formato que emplean las instrucciones con este modo de direccionamiento se muestra en la figura 3.14, en donde se observa que se dispone de 22 bits para el destino de la bifurcación, esta capacidad de direccionamiento queda sobrada para un ATMega16, no obstante, el formato de las instrucciones se dispone de esta manera para ser utilizado por otros miembros de la familia AVR que cuenten con un espacio mayor en su memoria de código.

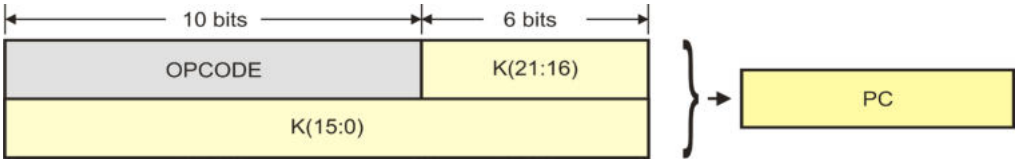


Figura 3.14 Bifurcaciones con direccionamiento absoluto

Un ATMega8 no incluye bifurcaciones que utilicen este modo de direccionamiento, dado que su espacio total puede cubrirse con bifurcaciones relativas o indirectas. Ejemplos de instrucciones que utilizan este modo de direccionamiento son:

```
JMP     0x001FFF
CALL    0x003000
```

3.3 Programación en Lenguaje Ensamblador

Un programa en lenguaje Ensamblador contiene:

- **Instrucciones:** Elementos del lenguaje que se traducen a código máquina, cada instrucción tiene su opcode y sus operandos. El procesador ejecuta las instrucciones para determinar el comportamiento de un sistema.

- **Directivas:** Elementos del lenguaje que ayudan en la organización de un programa, indicando diferentes aspectos como la ubicación del código, definiciones, etc. Las directivas no generan código máquina, son elementos propios de la herramienta empleada para ensamblar un programa.

En la sección 3.1 se mostraron las instrucciones que puede ejecutar un MCU AVR y en la sección 3.2, con los modos de direccionamiento, se revisaron los diferentes formatos de las instrucciones. En esta sección, para contar con los elementos suficientes para programar en Ensamblador, se revisan algunas de las directivas incluidas en el ensamblador desarrollado por Atmel (AVRASM) y distribuido con la herramienta AVR Studio[5] . No se revisan todas las directivas, sólo aquellas que son usadas con mayor frecuencia, mostrando ejemplos de su uso con los microcontroladores bajo estudio. Las directivas generalmente se preceden con un punto.

En la sección 3.5 se exponen 3 problemas simples con sus correspondientes soluciones. Las soluciones codificadas en ensamblador muestran la estructura general que deben seguir los programas escritos en este lenguaje e ilustran la forma en que se pueden traducir eficientemente las estructuras de control de flujo de alto nivel a bajo nivel, así como el uso de directivas.

3.3.1 Directiva INCLUDE

Esta directiva se utiliza para leer el código fuente de otro archivo, se coloca al inicio de un programa y es común utilizarla para incluir todas las definiciones relacionadas con un dispositivo particular. Por ejemplo, si se utiliza un ATMega8 se debe agregar:

```
.include <m8def.inc>          ; Incluye definiciones de un ATMega8
```

Al emplear un ATMega16 debe agregarse:

```
.include <m16def.inc>         ; Incluye definiciones de un ATMega16
```

3.3.2 Directivas CSEG, DSEG y ESEG

Estas directivas se utilizan para definir diferentes tipos de segmentos en un programa. Todo lo que se escribe, posterior a la directiva, es encausado al segmento correspondiente.

- **CSEG:** Marca el inicio de un segmento de código, es decir, un segmento en la memoria FLASH. Es el segmento por omisión, esto significa que la directiva puede omitirse en un programa si no se hace uso de otros segmentos. Un programa en ensamblador puede tener múltiples segmentos de código, los cuales son concatenados en el momento en que el programa es ensamblado. Cuando se ensambla un programa se genera un archivo con todos los segmentos de código definidos (*archivo.hex*).

5 AVR Studio es una suite para el desarrollo de aplicaciones con los microcontroladores AVR, proporciona las facilidades para la simulación y ensamblado de programas, es de distribución libre y puede descargarse del sitio de ATMEL (http://www.atmel.com).

- **DSEG:** Marca el inicio de un segmento para los datos, con ello, en un programa se pueden reservar localidades de SRAM para un propósito especial. De manera similar, puede haber múltiples segmentos de datos.
- **ESEG:** Indica que la siguiente información es guardada en EEPROM. También al ensamblar se crea un archivo a descargarse en la memoria EEPROM del MCU (*archivo.eep*), el cual sirve para definir el contenido inicial de variables en EEPROM. Para la lectura y modificación de estas variables puede utilizarse el código mostrado en la sección 2.4.2.

Un ejemplo del uso de estas directivas es el siguiente:

```
       .DSEG                ; Inicia un segmento de datos
var1:  .BYTE  1             ; Variable de 1 byte en SRAM

       .CSEG                ; Inicia un segmento de código

       ldi    R16, 0x25     ; Instrucciones para alguna tarea
       mov    R0,R16
       . . .

const: .DB 0x02             ; Constante con 0x02 en memoria de programa

       .ESEG                ; Inicia un segmento de EEPROM

eevar1:.DB    0x3f          ; Constante con 0x3f en memoria EEPROM
```

3.3.3 Directiva DB y DW

Con estas directivas se definen constantes o variables a utilizarse en memoria de programa o en EEPROM. **DB** es para datos que ocupan 1 byte (*Define Byte*) y **DW** para datos de 16 bits (*Define Word*). Si se requiere más de un dato, éstos deben organizarse en una lista de datos separados por comas. Ejemplos:

```
              .CSEG

; Constantes en memoria FLASH

const1:       .DB    0x33
consts1:      .DB    0, 255, 0b01010101, -128, 0xaa
consts2:      .DW    0, 0xffff, 0b1001110001010101, -32768, 65535
              .ESEG

; Constantes o variables en EEPROM

eevar1:       .DB    0x37
```

```
eelist1:      .DB    1,2,3,4
eelist2:      .DW    0,0xffff,10
```

La etiqueta con la definición puede utilizarse como referencia (dirección) para tener acceso a las constantes en un programa. Si es una lista, con la etiqueta se tiene acceso al inicio de la lista.

3.3.4 Directiva EQU

Se utiliza para definir constantes simbólicas, estas definiciones posteriormente pueden usarse en las instrucciones. Por ejemplo:

```
.EQU   io_offset = 0x23

.EQU   portA     = io_offset + 2

       .CSEG                ; Inicia el segmento de código
       CLR    R2            ; Limpia al registro 2
       OUT    portA, R2     ; Escribe a portA
```

El ensamblador hace referencia a los Registros I/O por su dirección y a sus bits por su posición, sin embargo, con la directiva **INCLUDE** se incluye una biblioteca con las definiciones de un dispositivo, por lo que al escribir un programa es posible utilizar sus nombres. Estas bibliotecas hacen un uso extensivo de la directiva **EQU**, asociando etiquetas a las direcciones.

3.3.5 Directiva ORG

Esta directiva se utiliza para ubicar la información subsecuente en cualquiera de los segmentos de memoria, indica la dirección a partir de la cual se colocan las variables en SRAM, constantes en EEPROM o en FLASH, o código en FLASH. Después de la directiva debe indicarse una dirección válida en el rango del segmento considerado.

Por ejemplo:

```
       .DSEG
       .ORG   0x120
var1:  .Byte  1             ; Variable ubicada en la dirección 0x120 del

                            ;segmento de datos
       .CSEG
       .ORG   0x000         ;Código ubicado en la dirección 0x00
       RJMP   inicio
       .ORG   0x010
inicio:                     ; Código ubicado en la dirección 0x10
       MOV    R1, R2
       . . . .
```

Se emplea principalmente para organizar a las instrucciones de un programa cuando se manejan interrupciones, porque hace posible la ubicación del código, de acuerdo con los vectores de interrupciones, como se mostró en la sección 2.6.

3.3.6 Directivas HIGH y LOW

Se utilizan para separar constantes de 16 bits, con **High** se hace referencia a la parte alta y con **Low** a la parte baja. Por ejemplo, para cargar la constante 578 en los registros R27 y R26:

```
        LDI     R27, HIGH(578)
        LDI     R26, LOW(578)
```

Se comentó al inicio que las directivas se preceden con un punto, esto no ocurre en **High** y **Low**, pero son consideradas como directivas porque no generan código máquina.

Un uso importante de estas directivas es la referencia a constantes que correspondan con direcciones de un programa, para después hacer accesos mediante direccionamiento indirecto. Por ejemplo, para posicionar al apuntador Z (R31:R30) al comienzo de una tabla de datos:

```
        LDI     R30, LOW(tabla)
        LDI     R31, HIGH(tabla)
        . . . .
tabla:  .DB     0x01,0x02,0x03,0x04
```

3.3.7 Directiva BYTE

Reserva uno o más bytes en SRAM o EEPROM para el manejo de variables, la cantidad de bytes debe especificarse. Este espacio no es inicializado, sólo queda reservado para que posteriormente sea referido con una etiqueta, por ejemplo:

```
        .DSEG

var1:   .BYTE   1               ; variable de 1 byte
var2:   .BYTE   10              ; variable de 10 bytes

        .CSEG

        STS     var1, R17       ;acceso a var1 por direccionamiento directo
        LDI     R30, LOW (var2) ;Z apunta a Var2
        LDI     R31, HIGH (var2)
        LD      R1, Z           ; acceso a var2 por el apuntador Z
```

3.4 Programación en Lenguaje C

Si bien, el entorno del AVR Studio únicamente incluye al programa ensamblador (AVRASM), proporciona las facilidades para enlazarse con compiladores de lenguaje C desarrollados por fuentes diferentes a Atmel. Instalando al compilador adecuado, desde el mismo entorno es posible la edición de programas, la invocación del compilador con exhibición de resultados, su simulación y depuración en lenguaje C.

Uno de estos compiladores es el avr-gcc, incluido en una suite conocida como WinAVR[6] , la cual es parte del proyecto GNU. Después de instalar a la suite, el compilador es llamado automáticamente desde el entorno del AVR Studio cada vez que se requiere, su uso queda transparente al programador. Además del compilador, la suite incluye un conjunto de bibliotecas con funciones enfocadas a los recursos de los AVR.

El compilador está orientado al estándar ANSI C, se pueden emplear a todos los elementos del lenguaje, como tipos de datos y estructuras de control de flujo. En esta sección se revisan algunas características del lenguaje, principalmente las consideraciones para trabajar con los microcontroladores AVR.

Los problemas de la sección 3.5 también se han resuelto en Lenguaje C, de manera que también muestran la estructura general que deben seguir los programas escritos en este lenguaje.

3.4.1 Tipos de Datos

Pueden usarse los tipos básicos con sus diferentes modificadores, en la tabla 3.25 se muestran los diferentes tipos de datos, con la cantidad de bits que requieren y el rango de combinaciones que cubren.

Tabla 3.25 Tipos de datos aceptados por el compilador

Tipo	Tamaño (bits)	Rango
char	8	-128 a 127
unsigned char	8	0 a 255
signed char	8	-128 a 127

6 WinAVR™ es una suite de herramientas de desarrollo para la serie de microcontroladores AVR de Atmel. Incluye al compilador AVR-GCC de GNU para C y C++. La suite contiene todas las herramientas necesarias para desarrollar aplicaciones con los AVR: Compilador, programador, depurador y más. Es un proyecto de código abierto, más información puede obtenerse de http://winavr.sourceforge.net/.

Tipo	Tamaño (bits)	Rango
int	16	-32, 768 a 32, 767
short int	16	-32, 768 a 32, 767
unsigned int	16	0 a 65, 535
signed int	16	-32, 768 a 32, 767
long int	32	-2, 147, 483, 648 a 2, 147, 483, 647
unsigned long int	32	0 a 4, 294, 967, 295
signed long int	32	-2, 147, 483, 648 a 2, 147, 483, 647
float	32	$+/- 1.175 \times 10^{-38}$ a $+/- 3.402 \times 10^{+38}$
double	32	$+/- 1.175 \times 10^{-38}$ a $+/- 3.402 \times 10^{+38}$

En la tabla 3.25 se muestran diferentes tipos de datos con el mismo comportamiento, por ejemplo *float* y *double* prácticamente hacen referencia al mismo tipo de datos, la inclusión de ambos debe ser para mantener compatibilidad con el estándar ANSI C. Las variables de los tipos *char, unsigned char* o *signed char* pueden asignarse directamente con los Registros I/O.

Al trabajar con un microcontrolador, debe considerarse el limitado espacio de memoria para no agotarlo con un mal manejo de los tipos de datos, por ejemplo, si en un ciclo repetitivo se realizan pocas iteraciones, es mejor usar un dato del tipo *unsigned char* en lugar de un *int*. Ya que no sólo es un byte extra, sino que con un *int* se ejecutan operaciones de 16 bits, requieriendo más instrucciones de bajo nivel.

En el entorno de desarrollo WinAVR se encuentran una serie de definiciones de tipos de datos que surgen a partir de los estándares mostrados en la tabla 3.25, ejemplos de estas definiciones de tipos son:

```
typedef signed char  int8_t
typedef unsigned char  uint8_t
typedef signed int  int16_t
typedef unsigned int  uint16_t
typedef signed long int  int32_t
typedef unsigned long int  uint32_t
```

Al programar se pueden utilizar los tipos de datos del ANSI C o las definiciones del WinAVR, esto no afecta el comportamiento de un programa. En el código desarrollado en este documento se utiliza al estándar y, sólo si es necesario, se emplea algún modificador propio del entorno de desarrollo.

3.4.2 Operadores Lógicos y para el Manejo de Bits

Aunque el compilador acepta todos los operadores del ANSI C (aritméticos, relacionales, de incremento y decremento), únicamente se describen los operadores lógicos y para el manejo de bits, porque son muy utilizados al trabajar con microcontroladores.

En la tabla 3.26 se muestran los operadores lógicos, sus operandos son expresiones lógicas (proporcionan falso o verdadero) y se emplean para crear expresiones lógicas de mayor complejidad, cualquier valor diferente de 0 es interpretado como verdadero.

Tabla 3.26 Operadores lógicos

Operador	Símbolo
AND	&&
OR	||

En la tabla 3.27 se muestran los operadores para el manejo de bits, éstos trabajan sobre datos de los tipos *char*, *int* o *long* (no trabajan sobre punto flotante) y afectan el resultado al nivel de bits. Son muy utilizados para enmascarar información, es decir, modificar únicamente algunos bits de un dato sin alterar a los demás, o bien para evaluar el estado de un bit.

Tabla 3.27 Operadores para el manejo de bits

Operador	Símbolo
Complemento a 1	~
Desplazamiento a la izquierda	<<
Desplazamiento a la derecha	>>
AND	&
OR	|
OR exclusivo	^

3.4.3 Tipos de Memoria

El microcontrolador tiene 3 espacios diferentes de memoria: SRAM (incluye a los registros de propósito general), FLASH y EEPROM, las aplicaciones pueden requerir que las variables y constantes se almacenen en diferentes tipos de memoria.

3.4.3.1 Datos en SRAM

Las variables son datos que van a ser leídos o escritos continuamente, por lo tanto, deben estar en SRAM, éste es el espacio de almacenamiento por default, por ejemplo, las siguientes declaraciones:

```
unsigned char   x, y;
unsigned int    a, b, c;
```

Colocan a las variables en SRAM. Si es posible, el compilador utiliza los registros de propósito general (R0 a R31) para algunas variables, pero como los registros están limitados en número, al agotarse se utiliza la SRAM de propósito general.

Los apuntadores son fundamentales al programar en lenguaje C y por lo tanto, no se han excluido de los microcontroladores AVR. Un apuntador también se maneja en SRAM porque es una variable, su contenido es una dirección que debe hacer referencia a algún objeto de SRAM. Por ejemplo:

```
char  cadena[]  = "hola mundo";
char  *pcad;
pcad = cadena;
```

3.4.3.2 Datos en FLASH

Las constantes son datos que no cambian durante la ejecución normal de un programa, los datos de este tipo pueden ser almacenados en la memoria FLASH. Al manejar las constantes en FLASH se liberan espacios significativos de SRAM, esto es fundamental para aplicaciones que incluyan cadenas de texto o tablas de búsqueda.

Para que el compilador dirija las constantes hacia la FLASH, éstas deben identificarse con la palabra reservada **const** e incluir al atributo **PROGMEM**, definido en la biblioteca **pgm space.h** que es parte del WinAVR. Ejemplos de declaraciones de constantes en FLASH son:

```
const  char cadena[] PROGMEM =  "Cadena con un mensaje constante";
const unsigned char tabla[] PROGMEM = { 0x24, 0x36, 0x48, 0x5A, 0x6C };
```

Las declaraciones anteriores tienen efecto cuando se realizan en un ámbito global, es decir, fuera de la función principal (**main**). Con ello, los datos quedan en FLASH y para su acceso deben utilizarse las funciones adecuadas, de lo contrario, al compilar se agrega una secuencia de código que realiza copias de FLASH a SRAM.

Las funciones para la lectura de constantes en FLASH están incluidas en la misma biblioteca (**pgmspace.h**), algunas de ellas son:

```
pgm_read_byte(address);       // Lee 8 bits
pgm_read_word(address);       // Lee 16 bits
pgm_read_dword(address);      // Lee 32 bits
```

Las funciones reciben como argumento la dirección del dato en FLASH, un ejemplo para la lectura del dato *i* de la tabla de constantes es:

```
x = pgm_read_byte(&tabla[i]);
```

Si se quiere emplear un apuntador a la memoria FLASH, debe declararse como **PGM_P**. La biblioteca **pgmspace.h** incluye funciones que trabajan con bloques completos de memoria FLASH, una de las más empleadas es **strcpy_P**, esta función sirve para leer una cadena de memoria FLASH y depositarla en SRAM, su uso se muestra en el siguiente código:

```
char   buf[32];             // Buffer destino, en SRAM
PGM_P  p;                   // Apuntador a memoria FLASH

    p = cadena;             // p apunta a la cadena en FLASH
    strcpy_P(buf, p);       // copia de FLASH a SRAM
```

En el ejemplo 3.2, descrito en la sección 3.5, se presenta una aplicación que hace uso de constantes en memoria FLASH.

3.4.3.3 Datos en EEPROM

Otra consideración debe tenerse para aquellas variables en las que se requiera conservar su contenido, aun en ausencia de energía, estas variables no pueden manejarse en SRAM porque es memoria volátil, deben almacenarse en la EEPROM.

Para ubicar una variable en la EEPROM, debe estar precedida con el atributo **EEMEM**, el cual está definido en la biblioteca **eeprom.h**, que también es parte del entorno de WinAVR. Este atributo hace que las declaraciones siguientes sean direccionadas a la EEPROM, éstas deben ser globales y realizarse antes de la declaración de constantes para la FLASH o variables para la SRAM. En el siguiente ejemplo se declaran variables para la EEPROM:

```
#include      <avr/eeprom.h>              //Biblioteca para la EEPROM

EEMEM  int  contador = 0x1234;            // Un entero requiere 2 bytes

EEMEM  unsigned char  clave[4] = { 1, 2, 3, 4}; // Arreglo de 4 bytes
```

La dirección para cada dato en EEPROM se define en el orden de su declaración, para el ejemplo anterior, la variable contador ocupa las direcciones 0 y 1, mientras que el arreglo utiliza de la localidad 2 a la 5. El contenido en la EEPROM con estas declaraciones es:

```
34 12 01 02 03 04 FF FF FF . . .
```

El acceso a los datos en la EEPROM se realiza por medio de los registros **EEAR**, **EEDR** y **EECR**, para ello pueden emplearse las funciones mostradas en la sección 2.4.2, o bien, desarrollar otras funciones en donde se involucre a la interrupción por fin de escritura en EEPROM. En la biblioteca **eeprom.h** se cuenta con diversas funciones para el acceso a la EEPROM, 2 de éstas son:

- **eeprom_read_byte**: Lee un byte de la EEPROM, en la dirección que recibe como argumento.
- **eeprom_write_byte:** Escribe un byte en la EEPROM, como argumentos recibe la dirección y el dato.

También se incluyen funciones para datos de 16 y 32 bits, así como para el manejo de bloques de memoria.

3.5 Programas de Ejemplo

En esta sección se muestran 3 ejercicios resueltos en Lenguaje C y en ensamblador, para familiarizarse con la programación de los microcontroladores AVR. Los ejercicios se probaron en un ATMega8, pero pueden funcionar en un ATMega16. En lenguaje C no se necesitarían cambios en el código y para ensamblador, sólo habría que remplazar la biblioteca de definiciones.

Cuando se inicia con un proyecto nuevo en el AVR Studio, se debe especificar el dispositivo destino. El apéndice C contiene un tutorial sobre el AVR Studio, mostrando los pasos a seguir durante la creación y simulación de un proyecto.

3.5.1 Parpadeo de un LED

Al iniciarse en algún lenguaje de programación, normalmente se codifica al típico programa "Hola Mundo", a pesar de que el programa no tiene algún propósito, sirve para mostrar la estructura de los programas futuros, éste es el objetivo del presente ejemplo.

Ejemplo 3.1 Realice un programa que haga parpadear un LED conectado en la terminal PB0 a una frecuencia aproximada de 1 Hz (periodo de 1 S), considerando un ciclo útil del 50 % (½ S encendido y ½ S apagado). En la figura 3.15 se muestra el hardware requerido y en la 3.16 un diagrama de flujo con el comportamiento esperado.

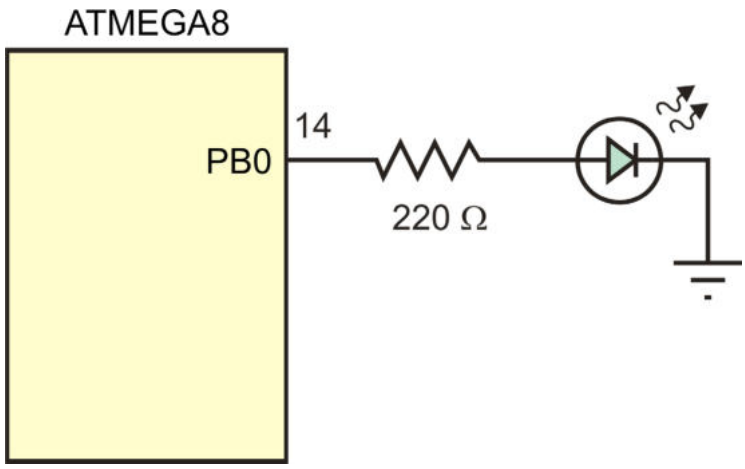


Fig. 3.15 Hardware para el problema del parpadeo de un LED

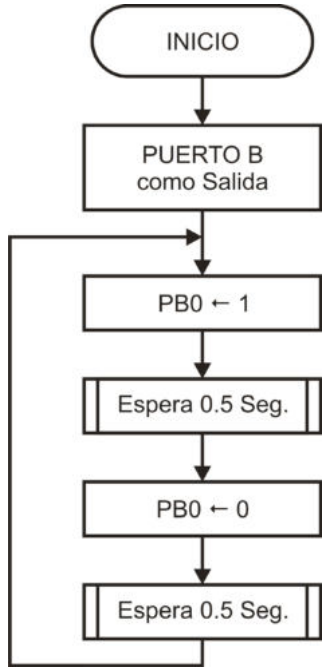


Fig. 3.16 Comportamiento deseado en el problema del parpadeo de un LED

En la figura 3.16 se observa que el programa nunca termina, permanece en un lazo infinito, esto es normal en sistemas basados en MCUs.

Para el programa en Ensamblador, se debe realizar una rutina para los retrasos con base en ciclos repetitivos, para ello, se asume un oscilador interno de 1 MHz, que corresponde con la configuración inicial de un ATMega8.

```
        .include  <m8def.inc>       ; Biblioteca con definiciones

        LDI     R16, 0xFF
        OUT     DDRB, R16           ; Puerto B como salida

        LDI     R16, 0x04           ; Ubica al apuntador de pila al final de SRAM
        OUT     SPH, R16            ; porque hay un llamado a una rutina
        LDI     R16, 0x5F
        OUT     SPL, R16
Lazo:                               ; Lazo infinito
        SBI     PORTB, 0            ; PB0 en alto
        RCALL   Espera_500mS
        CBI     PORTB, 0            ; PB0 en bajo
        RCALL   Espera_500mS
        RJMP    Lazo
;
; Una rutina de espera se debe revisar del lazo interno al externo
;       500 mS = 500 000 uS = 2 x ( 250 x (250 x 4 uS) )
;
```

```
Espera_500mS:
       LDI    R18, 2
et3:   LDI    R17, 250
et2:   LDI    R16, 250
et1:   NOP              ; Itera 250 veces, emplea 4 uS por iteración
       DEC    R16       ; 250 x 4 uS = 1000 uS = 1 mS
       BRNE   et1       ; La instrucción evalúa la bandera de cero
                        ; brinca si no hay bandera de cero
       DEC    R17
       BRNE   et2       ; 1 mS x 250 = 250 mS

       DEC    R18
       BRNE   et3       ; 250 mS x 2 = 500 mS
       RET
```

Para la versión en Lenguaje C, puesto que se desconoce cuantas instrucciones de bajo nivel corresponden a las instruc ciones de alto nivel, lo mejor para los retardos es emplear la función *_delay_ms(double _ms)* de la biblioteca *delay.h*, que es parte del entorno de desarrollo WinAVR. Esta función también se basa en iteraciones, el número de iteraciones depende del argumento recibido y de la frecuencia de la CPU (F_CPU).

La función *_delay_ms(double _ms)* recibe como argumento un número en punto flotante de doble precisión indicando la cantidad de milisegundos que se requieren de espera, el retraso máximo es de 262.14 mS/F_CPU, con la frecuencia en MHz. Si se intentan intervalos de espera mayores, no se producen errores de sintaxis, pero la función no proporciona el retardo esperado.

```
#define F_CPU 1000000UL          // Frecuencia de trabajo de 1 MHz

#include     <util/delay.h>     // Funciones para retrasos
#include     <avr/io.h>         // Definiciones de Registros I/O

int  main() {                    // La función es int, aunque no haya
                                 // retorno, si
                                 // se define como void, se obtiene una
                                 // precaución
DDRB = 0xFF;                     // Puerto B como salida

while(1) {                       // Lazo infinito
       PORTB = PORTB | 0x01;     // PB0 en alto (máscara con OR)
       _delay_ms(250);
       _delay_ms(250);
       PORTB = PORTB & 0xFE;     // PB0 en bajo (máscara con AND)
       _delay_ms(250);
       _delay_ms(250);

       }
}
```

Se observa que sin importar el lenguaje de programación, se debe conseguir el comportamiento mostrado en la figura 3.16. En lenguaje C, la inicialización del apuntador de pila no se realiza en el código fuente, estas instrucciones las agrega el compilador en el momento en que genera el código de bajo nivel.

La biblioteca *delay.h* también cuenta con la función *_delay_us(double _us)* para retardos en el orden de microsegundos, con un máximo de 768 us/F_CPU.

3.5.2 Decodificador de Binario a 7 Segmentos

En este ejemplo se resaltan 2 aspectos importantes al trabajar con MCUs, el primero es que los códigos de 7 segmentos son constantes, por lo tanto conviene depositarlos en memoria FLASH, y el segundo es que se debe tener el mismo tiempo de respuesta para todas las combinaciones, es decir, obtener el código de la F debe requerir el mismo tiempo que obtener el código del 0.

En otras palabras, en este ejemplo se ilustra el manejo de una búsqueda, en una tabla de constantes.

Ejemplo 3.2 Desarrolle un programa que lea los 4 bits menos significativos del Puerto D [PD3:PD0] y genere su código en 7 segmentos en el Puerto B. En la figura 3.17 se muestra el hardware requerido con el valor de las salidas para cada una de las entradas, la solución debe comportarse como el diagrama de flujo mostrado en la figura 3.18

Núm.	g	f	e	d	c	b	a	HEX
0	0	1	1	1	1	1	1	0x3F
1	0	0	0	0	1	1	0	0x06
2	1	0	1	1	0	1	1	0x5B
3	1	0	0	1	1	1	1	0x4F
4	1	1	0	0	1	1	0	0x66
5	1	1	0	1	1	0	1	0x6D
6	1	1	1	1	1	0	1	0x7D
7	0	0	0	0	1	1	1	0x07
8	1	1	1	1	1	1	1	0x7F
9	1	1	0	0	1	1	1	0x67
A	1	1	1	0	1	1	1	0x77
B	1	1	1	1	1	0	0	0x7C
C	0	1	1	1	0	0	1	0x39
D	1	0	1	1	1	1	0	0x5E
E	1	1	1	1	0	0	1	0x79
F	1	1	1	0	0	0	1	0x71

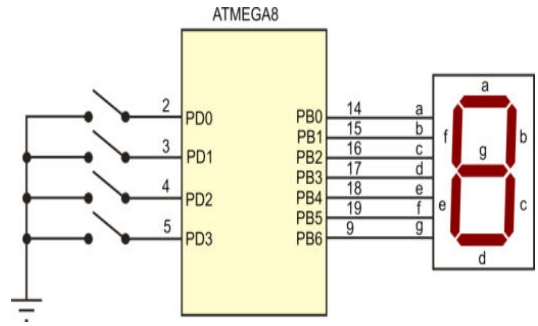


Figura 3.17 Decodificador de binario a 7 Segmentos

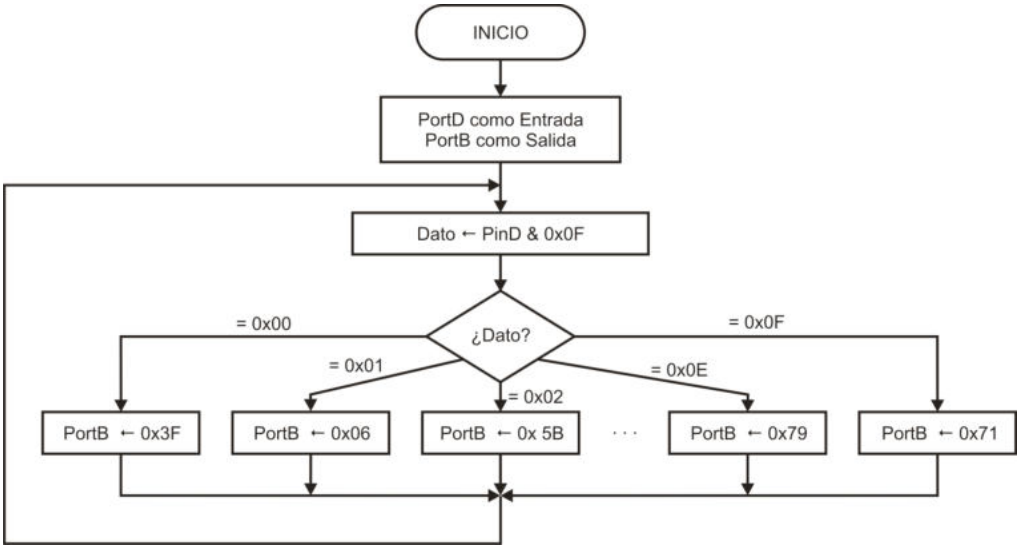


Figura 3.18 Comportamiento esperado en el decodificador de binario a 7 Segmentos

Una idea inmediata para solucionar este problema involucra la realización de 15 comparaciones, esta propuesta es ineficiente porque los tiempos de respuesta serían diferentes para cada caso.

Para la solución en lenguaje C, el diagrama de flujo sugiere el uso de una estructura *switch-case*, no obstante, con la variable *dato* se define la asignación de una constante de un conjunto de 16, por ello, una solución más simple involucra el uso de un arreglo de 16 constantes y el empleo de la variable *dato* como índice. La solución en lenguaje C es la siguiente:

```
#include      <avr/io.h>           // Definiciones de Registros I/O
#include      <avr/pgmspace.h>     // Acceso a memoria FLASH

// Tabla de constantes en memoria FLASH
const char  tabla[] PROGMEM = { 0x3F,0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07,
                    0x7F, 0x67, 0x77, 0x7C, 0x39, 0x5E, 0x79, 0x71 };

int  main() {

unsigned char  dato;
DDRD = 0x00;                              // Puerto D como entrada
PORTD = 0xFF;                             // Habilita resistores de Pull-Up
DDRB = 0xFF;                              // Puerto B como salida

while(1) {                          // Lazo infinito
                                    // Lee las entradas y anula los 4
       dato = PIND & 0x0F;          // bits más significativos
              PORTB = pgm_read_byte(&tabla[dato]);// Acceso a la FLASH
       }
}
```

En lenguaje ensamblador también se debe observar una tabla de constantes en memoria de código y un registro funcionando como apuntador, para hacer las lecturas empleando direccionamiento indirecto. El código es el siguiente:

```
        .include  <m8def.inc>

        LDI     R16, 0x00               ; Puerto D como entrada
        OUT     DDRD, R16
        LDI     R16, 0xFF               ; Resistores de Pull-Up
        OUT     PORTD, R16

        OUT     DDRB, R16               ; Puerto B como salida
LOOP:                                   ; Lazo infinito
        IN      R16, PIND               ; R16 se utiliza para la variable Dato
        ANDI    R16, 0x0F

        LDI     R31, HIGH (tabla << 1) ; Apuntador Z al inicio de la tabla
        LDI     R30, LOW (tabla << 1)
        ADD     R30, R16                ; Suma el índice a Z
        BRCC    s1
        INC     R31                     ; Si hay acarreo modifica la parte alta de Z
s1:
        LPM     R17, Z                  ; Carga de FLASH
        OUT     PORTB, R17              ; Coloca la salida
        RJMP    LOOP

; Tabla de constantes en memoria FLASH
tabla:  .DB  0x3F, 0x06, 0x5B, 0x4F, 0x66, 0x6D, 0x7D, 0x07
        .DB  0x7F, 0x67, 0x77, 0x7C, 0x39, 0x5E, 0x79, 0x71
```

Para la versión en ensamblador, con el posicionamiento del apuntador Z al comienzo de la tabla se realiza un desplazamiento a la izquierda de la constante, esto equivale a una multiplicación por 2 y se requiere porque la memoria está organizada en palabras de 16 bits, la etiqueta `tabla` hace referencia a una dirección de palabra y la instrucción **`LPM`** requiere una dirección de byte, cada palabra incluye 2 bytes. Por ello, si se utiliza un número impar de bytes como constantes, al ensamblar se genera una advertencia por la desalineación del código.

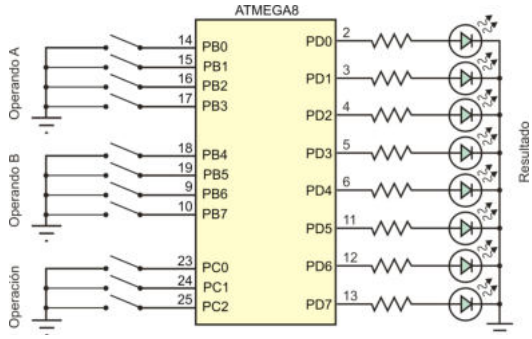
En este ejemplo también se mostró que existe una relación directa entre hardware y software, en aplicaciones con MCUs normalmente ocurre que es posible un ahorro de hardware si se escriben más líneas de código. O en caso contrario, si la memoria de código se ha agotado, algunas tareas de software podrían realizarse con hardware externo.

Respecto al ejemplo, al utilizar la AND para conservar sólo la parte baja del puerto D, hace innecesario tener que aterrizar vía hardware a la parte alta del mismo puerto, estas terminales pueden permanecer abiertas sin alterar el funcionamiento del sistema. También, los resistores de *Pull-Up* ya se encuentran dentro del AVR, sólo fue necesaria su habilitación para evitar el uso de resistores externos.

3.5.3 Diseño de una ALU de 4 Bits

En este ejemplo se muestra el uso de otra característica importante en la programación de MCUs, el uso de una tabla de saltos.

Ejemplo 3.3 Construya una ALU de 4 bits utilizando un ATMega8, en donde los operandos se lean del puerto B (nibble bajo para el operando A y nibble alto para el operando B), el resultado se genere en el puerto D y con los 3 bits menos s ignificativos del puerto C se defina la operación. En la figura 3.19 se muestran las entradas, salidas y las operaciones.



PortC[2:0]	Operación
000	R = A + B
001	R = A - B
010	R = A * B
011	R = A **AND** B
100	R = A **OR** B
101 a 111	R = 0

Figura 3.19 ALU de 4 bits con 5 operaciones

El comportamiento esperado para la ALU se muestra en el diagrama de flujo de la figura 3.20. Los 2 operandos se obtienen del mismo puerto, por lo que deben separarse utilizando operaciones lógicas.

Para el programa en lenguaje C, lo natural es el uso de una estructura *switch-case*, porque se tiene una operación diferente en cada caso, a diferencia del ejemplo anterior, en el cual únicamente se obtiene una constante.

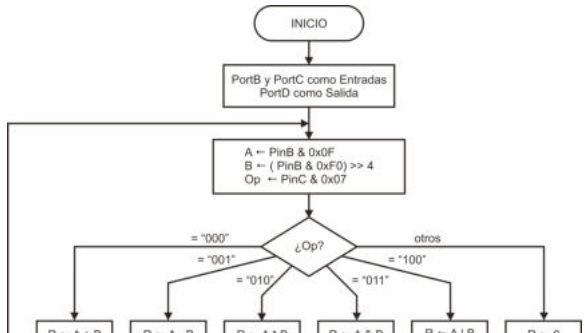


Figura 3.20 Comportamiento para la ALU

El programa en lenguaje C es:

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

int    main() {

unsigned  char   A, B, R, Op;     // Variables locales

       DDRB = 0x00;               // Configura los puertos de entrada
       DDRC = 0x00;
       PORTB = 0xFF;              // Resistores de Pull-Up
       PORTC = 0xFF;
       DDRD = 0xFF;               // Puerto D como salida

       while(1) {

              A = PINB & 0x0F;
              B = ( PINB & 0xF0 ) >> 4;
              Op = PINC & 0x07;

              switch( Op ) {
                     case 0:      R = A + B;          // Suma
                                  break;
                     case 1:      R = A - B;          // Resta
                                  break;
                     case 2:      R = A * B;          // Producto
                                  break;
                     case 3:      R = A & B;          // AND lógica
                                  break;
                     case 4:      R = A | B;          // OR lógica
                                  break;
                     default:     R = 0;
              }

              PORTD = R;          // Genera la salida
       }
}
```

No se considera la posibilidad de acarreos porque los datos son de 4 bits, para las 3 operaciones aritméticas el resultado alcanza perfectamente en 8 bits.

En una estructura *switch-case* se debe consumir el mismo tiempo para bifurcar a cada uno de los casos, ésta es la principal diferencia con un conjunto de estructuras *if-else* anidadas. Este comportamiento también debe observarse en ensamblador, para ello se utiliza una tabla de saltos, empleando bifurcaciones indirectas. El código en ensamblador correspondiente es:

```
       .include  <m8def.inc>

       LDI    R16, 0x00           ; Puertos B y C como entradas
       OUT    DDRB, R16
       OUT    DDRC, R16
```

```
        LDI     R16, 0xFF           ; Resistores de Pull-Up
        OUT     PORTB, R16
        OUT     PORTC, R16
        OUT     DDRD, R16           ; Puerto D como salida

loop:                               ; Lazo infinito
        IN      R20, PINB           ; Se emplea R20 para el operando A
        ANDI    R20, 0x0F
        IN      R21, PINB           ; Se emplea R21 para el operando B
        ANDI    R21, 0xF0
        SWAP    R21
        IN      R22, PINC           ; y R22 para la operación
        ANDI    R22, 0x07

        CPI     R22, 5              ; Observa si es un caso válido
        BRGE    no_valido

valido:
        LDI     R30, LOW(tabla); Z apunta al inicio de la tabla de saltos
        LDI     R31, HIGH(tabla)
        ADD     R30, R22            ; Suma el caso detectado
        BRNE    no_carry
        INC     R31                 ; Se considera el acarreo
no_carry:
        IJMP                        ;

no_valido:                          ; Todos los casos inválidos se etiquetan con 5
        LDI     R22, 5              ; con 5 el caso ya es válido (default)
        RJMP    valido

tabla:
        RJMP    case_0              ; Tabla de saltos
        RJMP    case_1
        RJMP    case_2
        RJMP    case_3
        RJMP    case_4
        RJMP    default
case_0:                             ; Suma, el resultado queda en R23
        MOV     R23, R20            ; para todos los casos
        ADD     R23, R21
        RJMP    salir

case_1:
        MOV     R23, R20            ; Resta
        SUB     R23, R21
        RJMP    salir
case_2:
        MUL     R21, R20            ; Producto
        MOV     R23, R0             ; ubica el resultado
        RJMP    salir
case_3:
        MOV     R23, R20            ; AND lógica
```

```
        AND     R23, R21
        RJMP    salir
case_4:
        MOV     R23, R20            ; OR lógica
        OR      R23, R21
        RJMP    salir
default:
        CLR     R23                 ; Operación no válida
salir:                              ; fin del switch-case
        OUT     PORTD, R23          ; Genera la salida
        RJMP    loop;
```

En el código en ensamblador se observa como antes de tener acceso a la tabla de saltos se debe garantizar un caso válido, a los casos inválidos se les asigna el valor de 5, que corresponde con el caso por default.

3.6 Relación entre Lenguaje C y Ensamblador

En los ejemplos anteriores se ha mostrado un aspecto importante en el desarrollo de sistemas con microcontroladores. Antes de determinar qué lenguaje se va a emplear, es necesario un análisis de la solución, buscando que ésta sea óptima.

En ocasiones se dice que es "mejor" programar en ensamblador porque tiene una relación directa con el código máquina que se genera. Lo cual es acertado, al programar en alto nivel siempre se produce código adicional, debido a los mecanismos que los compiladores utilizan para el respaldo de variables durante llamadas a funciones y a las políticas en el uso de registros, que conllevan a un uso exhaustivo de SRAM para variables.

Sin embargo, el aspecto más importante es la organización de la solución de un problema, un programa en ensamblador resultante de una mala o nula organización, puede ser más ineficiente que un programa en C, resultante de una propuesta de solución bien organizada.

La ventaja de emplear un lenguaje de alto nivel es que cuenta con estructuras de control de flujo que facilitan la codificación de soluciones estructuradas. Ante un problema con una complejidad de mediana a alta, la programación en alto nivel produce un código más compacto y menos confuso, reduciendo con ello la posibilidad de cometer errores.

Por otro lado, la velocidad de ejecución de las instrucciones y la cantidad de memoria con que actualmente cuentan los microcontroladores, hacen que el código adicional y el tiempo que se invierte en su ejecución no sea un factor determinante para no emplear un lenguaje de alto nivel en la mayoría de aplicaciones.

El lenguaje ensamblador sería necesario en las siguientes situaciones:

1. La aplicación requiere un control estricto en la temporización de algunas operaciones y los intervalos de tiempo requeridos no se puede conseguir con los temporizadores internos.

2. El tamaño de la memoria de código realmente es reducido, por ejemplo, algunos AVR de la gama Tiny incluyen 1 Kbyte en su memoria de programa.

3. La aplicación requiere una manipulación extensiva de bits, por ejemplo, para reducir el espacio utilizado para almacenar un conjunto de datos, en lenguaje ensamblador directamente puede hacerse un empaquetamiento de datos para comprimir la información.

Además, es posible ejecutar código ensamblador dentro de un programa en C utilizando la proposición `asm()`, la cual puede recibir hasta 4 argumentos con la siguiente sintaxis:

```
asm(código: operandos de salida: operandos de entrada [:restricciones]);
```

- **Código**: Cadena de texto entre comillas con la instrucción o instrucciones en ensamblador. Si es necesario pueden incluirse especificaciones de conversión con el carácter %.

- **Operandos de salida/entrada**: Lista de operandos que corresponden con las especificaciones de conversión indicadas en el código, pueden ser registros de propósito general o Registros I/O.

- **Restricciones**: Este argumento puede omitirse, se utiliza para indicar los registros que se están incluyendo en el código sin haber sido especificados como operandos de entrada o salida.

Los registros de propósito general son empleados por el compilador en el momento en que realiza la traducción a bajo nivel, por lo tanto, si se van a utilizar como operandos de salida o entrada, deben conocerse y respetarse las políticas que aplica el compilador para el manejo de registros.

Si se busca simplicidad en la escritura de código, sólo es recomendable incluir sentencias con ensamblador al realizar operaciones específicas con los Registros I/O. Algunos ejemplos, bajo este criterio de simplicidad, son:

```
asm("NOP");          // No operación, tarda 1 ciclo de reloj
asm("SBI 0x18, 0"); // Pone en alto al bit 0 de PORTB
asm("CBI 0x18, 0"); // Pone en bajo al bit 0 de PORTB
asm("SEI  \n"        // Habilita las interrupciones y
    "CLC");          // limpia la bandera de acarreo
```

La instrucción ejecutada en el primer ejemplo únicamente sirve para perder 1 ciclo de reloj. Por lo que no afecta a los recursos del microcontrolador.

En el segundo y tercer ejemplo se hace referencia a un Registro I/O por su dirección y no por su nombre (PORTB), no es posible utilizar el nombre porque, para el código en ensamblador, no se ha incluido la biblioteca con las definiciones.

En el último ejemplo se han incluido dos instrucciones en la misma proposición, se inserta al carácter de nueva línea porque la sintaxis del lenguaje ensamblador sólo permite incluir una instrucción por línea.

3.7 Ejercicios

Se presenta una serie de problemas que pueden resolverse utilizando lenguaje ensamblador o C. Para la implementación, es posible el uso de un ATMega8 o de un ATMega16.

1. Emule el circuito combinacional mostrado en la figura 3.21, utilice un AVR programado en lenguaje Ensamblador. Sugerencia: Mueva cada bit a la posición menos significativa de un registro y aplique las operaciones lógicas sobre los registros. El resultado debe quedar en el bit menos significativo.

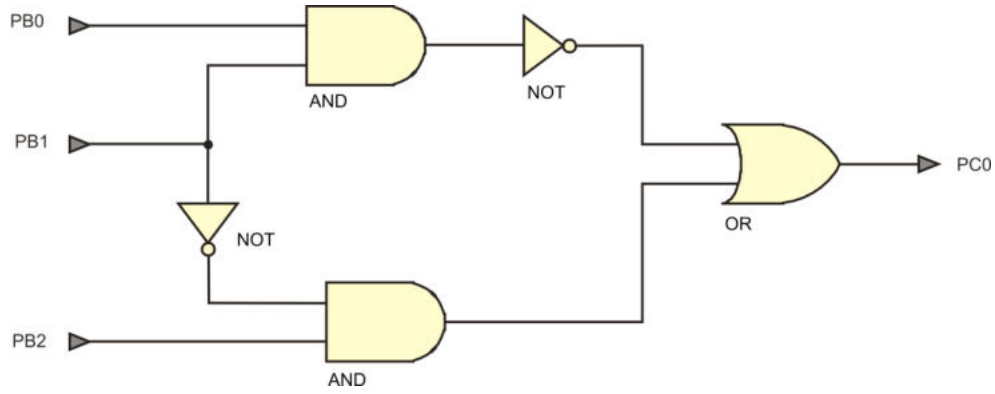


Figura 3.21 Circuito combinacional para el problema 1

   Ante cambios en las entradas ¿Cuál es el tiempo de respuesta si el MCU está operando a 1 MHz?

2. Construya un comparador de 2 datos (A y B) de 4 bits leídos en el puerto B del microcontrolador, A en PB[3:0] y B en PB[7:4]. El comparador debe generar 3 salidas en el puerto C para indicar si: A > B (PC0), A = B (PC1) o A < B (PC2).

3. Realice un contador de segundos con salida en un display de 7 segmentos conectado en el puerto B de un AVR. Inicialmente muestre al 0, un segundo después, incremente para mostrar al 1 y así sucesivamente (0, 1, 2, etc.). Al llegar a la F, inicie nuevamente con el 0.

4. Modifique el problema 3 agregando un botón conectado en la terminal PD0 como se muestra en la figura 3.22 (a), habilite al resistor de Pull-Up para tener un 1 lógico mientras el botón se mantiene abierto. Organice el programa para que la salida se incremente en 1 cada vez que se presione el botón.

   Al sondear un botón por software, debe considerarse que cuando el usuario lo presiona tarda un tiempo entre 150 y 300 mS, y como las operaciones del MCU están en el orden de microsegundos, la salida se incrementa en forma desmedida si no se inserta un retardo. Para ello, es conveniente utilizar un esquema como el mostrado en la figura 3.22 (b), con el cual también se va a eliminar ruido al sondear 2 veces al botón.

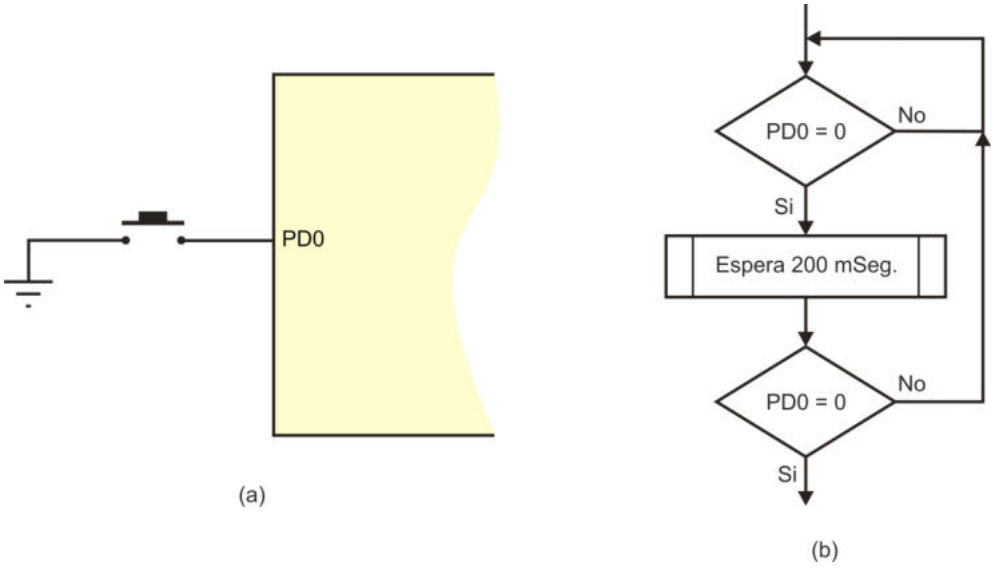


Figura 3.22 (a) Conexión de un botón y (b) espera a que el botón sea presionado

5. Implemente un sistema que maneje 2 semáforos con los 3 colores básicos (Rojo, Amarillo y Verde), siguiendo la secuencia de tiempos mostrada en la tabla 3.28.

Rojo1	Amarillo1	Verde1	Rojo2	Amarillo2	Verde2	Tiempo (Seg)
1	0	0	0	0	1	15
1	0	0	0	0	parpadeo	5
1	0	0	0	1	0	5
0	0	1	1	0	0	15
0	0	parpadeo	1	0	0	5
0	1	0	1	0	0	5

Tabla 3.28 Secuencia para los semáforos

Para el parpadeo en el color verde, considere medio segundo encendido y medio segundo apagado.

4. Interrupciones Externas, Temporizadores y PWM

Los microcontroladores AVR tienen una gama amplia de recursos internos, en este capítulo se describen dos recursos fundamentales: las interrupciones externas y los temporizadores. La generación de señales moduladas en ancho de pulso (PWM*, Pulse Width Modulation*) no se realiza con un recurso adicional, las señales PWM se generan con una de las diferentes formas de operación de los temporizadores, no obstante, se dedica la sección 4.3 a esta forma de operación debido al gran número de aplicaciones que pueden desarrollarse con base en PWM.

Debe recordarse que la configuración y uso de los recursos internos de un MCU se realiza por medio de los Registros I/O correspondientes. Por ello, un sistema basado en los recursos trabaja de manera adecuada si se colocan los valores correctos en sus registros de control.

4.1 Interrupciones Externas

Las interrupciones externas sirven para detectar un estado lógico o un cambio de estado en alguna de las terminales de entrada de un microcontrolador, con su uso se evita un sondeo continuo en la terminal de interés. Son útiles para monitorear interruptores, botones o sensores con salida a relevador. En la tabla 4 se describen las interrupciones externas existentes en los AVR bajo estudio, en el ATMega8 se tienen 2 fuentes y en el ATMega16 son 3.

Tabla 4.1 Interrupciones externas y su ubicación en MCUs con encapsulado PDIP

ATMega8			**ATMega16**		
Fuente	Ubicación	Terminal	Fuente	Ubicación	Terminal
INT0	PD2	4	INT0	PD2	16
INT1	PD3	5	INT1	PD3	17
			INT2	PB2	3

Las interrupciones externas pueden configurarse para detectar un nivel bajo de voltaje o una transición, ya sea por un flanco de subida o de bajada. Con excepción de INT2, que sólo puede activarse por flancos. Las interrupciones pueden generarse aun cuando sus respectivas terminales sean configuradas como salidas.

Las transiciones en INT0/INT1 requieren de la señal de reloj destinada a los módulos de los recursos ($clk_{I/O}$, sección 2.8) para producir una interrupción, esta señal de reloj es anulada en la mayoría de los modos de bajo consumo (sección 2.9). Por el contrario, un nivel bajo en INT0/INT1 y las transiciones en INT2 no requieren de una señal de reloj para producir una interrupción, puede decirse que son eventos asíncronos, por lo que éstos son adecuados para despertar al microcontrolador, sin importar el modo de reposo.

4.1.1 Configuración de las Interrupciones Externas

La configuración de INT0 e INT1 se define en el registro **MCUCR** (*MCU Control Register*), los 4 bits más significativos de este registro están relacionados con los modos de bajo consumo de energía y fueron descritos en la sección 2.9, los 4 bits menos significativos son:

	7	6	5	4	3	2	1	0	
0x35	SE	SM2	SM1	SM0	ISC11	ISC10	ISC01	ISC00	**MCUCR**

- **Bits 3 y 2 – ISC1[1:0]: Para configurar el sentido de INT1 (*ISC, Interrupt Sense Control*)**

  Definen el tipo de evento que genera la interrupción externa 1.

- **Bits 1 y 0 – ISC0[1:0]: Para configurar el sentido de INT0**

  Definen el tipo de evento que genera la interrupción externa 0.

En la tabla 4.2 se muestran los eventos que generan estas interrupciones, de acuerdo con el valor de los bits de configuración.

Tabla 4.2 Configuración del sentido de las interrupciones externas 0 y 1

ISCx1	ISCx0	Activación de la Interrupción
0	0	Por un nivel bajo de voltaje en INTx
0	1	Por cualquier cambio lógico en INTx
1	0	Por un flanco de bajada en INTx
1	1	Por un flanco de subida en INTx

x puede ser 0 ó 1

La configuración de INT2 se define con el bit **ISC2** ubicado en la posición 6 del registro **MCUCSR** (MCU Control and Status Register).

	7	6	5	4	3	2	1	0	
0x34	JTD	ISC2	-	JTRF	WDRF	BORF	EXTRF	PORF	**MCUCSR**

En la tabla 4.3 se muestran las transiciones que generan la interrupción externa 2, en función del bit **ISC2**.

Tabla 4.3 Configuración del sentido de la interrupción externa 2

ISC2	Activación de la Interrupción
0	Por un flanco de bajada en INT2
1	Por un flanco de subida en INT2

Dado que la interrupción 2 es asíncrona, se requiere que el pulso generador del evento tenga una duración mínima de 50 nS para que pueda ser detectado.

4.1.2 Habilitación y Estado de las Interrupciones Externas

Cualquier interrupción va a producirse sólo si se activó al habilitador global de interrupciones y al habilitador individual de la interrupción de interés. El habilitador global es el bit **I**, ubicado en la posición 7 del registro de Estado (**SREG**, sección 2.4.1.1).

Los habilitadores individuales de las interrupciones externas se encuentran en el registro general para el control de interrupciones (**GICR**, *General Interrupt Control Register*), correspondiendo con los 3 bits más significativos de **GICR**:

	7	6	5	4	3	2	1	0	
0x3B	INT1	INT0	INT2	-	-	-	IVSEL	IVCE	**GICR**

- **Bit 7 – INT1: Habilitador individual de la interrupción externa 1**
- **Bit 6 – INT0: Habilitador individual de la interrupción externa 0**
- **Bit 5 – INT2: Habilitador individual de la interrupción externa 2**

  No está disponible en un ATMega8.

- **Bits 4 al 2 – No están implementados**
- **Bits 1 y 0 – No están relacionados con las interrupciones externas**

El estado de las interrupciones externas se refleja en el registro general de banderas de interrupción (**GIFR**, *General Interrupt Flag Register*), el cual incluye una bandera por interrupción, estas banderas corresponden con los 3 bits más significativos de **GIFR**:

	7	6	5	4	3	2	1	0	
0x3A	INTF1	INTF0	INTF2	-	-	-	-	-	**GIFR**

- **Bit 7 – INTF1: Bandera de la interrupción externa 1**
- **Bit 6 – INTF0: Bandera de la interrupción externa 0**
- **Bit 5 – INTF2: Bandera de la interrupción externa 2**

  No está disponible en un ATMega8.

- **Bits 4 al 0 – No están implementados**

Las banderas se ponen en alto si el habilitador global y los habilitadores individuales están activados y ocurre el evento definido por los bits de configuración. La puesta en alto de una de estas banderas es lo que produce la interrupción, dando lugar a los

procedimientos descritos en la sección 2.6.1, la bandera se limpia automáticamente por hardware cuando se concluye con la atención a la interrupción. No es necesario evaluar las banderas por software, puesto que se tiene un vector diferente para cada evento.

4.1.3 Ejemplos de Uso de Interrupciones Externas

En esta sección se muestran dos ejemplos de uso de las interrupciones externas, documentando aspectos en la programación que deben ser considerados al momento de desarrollar otras aplicaciones.

Ejemplo 4.1: Realice un programa que conmute la salida menos significativa del puerto B (PB0) de un ATMega8 cada vez que es presionado un botón conectado en INT0, como se muestra en la figura 4.1.

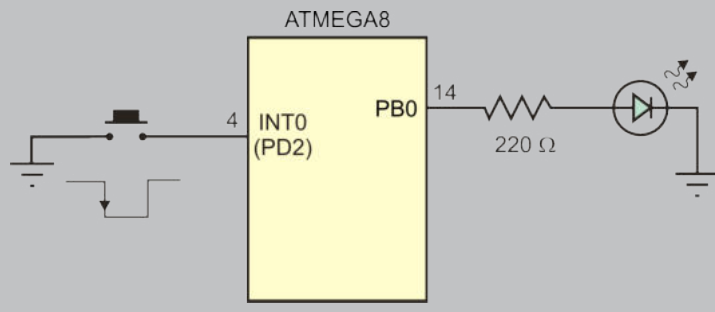


Figura 4.1 Manejo de un botón por interrupción

En el puerto D se habilita al resistor de *pull-up* para contar con un 1 lógico mientras no se presione el botón. El otro extremo del botón se conecta a tierra para insertar un 0 lógico al presionarlo, por lo tanto, la interrupción se configura por flanco de bajada.

La conmutación de la salida se realiza en la ISR, por lo que el programa principal queda ocioso.

Para la solución en ensamblador debe inicializarse al apuntador de pila, por la rutina de atención a la interrupción. La solución en ensamblador es:

```
        .include  <m8def.inc>       ; Biblioteca con definiciones

        .ORG    0x000               ; Vector de reset
        RJMP    Inicio


; La ISR se ubica en el vector de la interrupción, por ser la única a evaluar
        .ORG    0x001               ; Vector de la interrupción externa 0
        IN      R16, PORTB          ; Lee el último valor escrito
        EOR     R16, R17            ; Conmuta al LSB
        OUT     PORTB, R16          ; Escribe el resultado
```

```
        RETI

Inicio:                         ; Inicializaciones
        LDI     R16, 0x00
        OUT     DDRD, R16       ; Puerto D como entrada
        LDI     R16, 0xFF
        OUT     PORTD, R16      ; Resistor de Pull-Up en el puerto D

        OUT     DDRB, R16       ; Puerto B como salida

        LDI     R16, 0x04       ; Inicializa al apuntador de pila
        OUT     SPH, R16
        LDI     R16, 0x5F
        OUT     SPL, R16

        LDI     R16, 0B00000010 ; Configura INT0 por flanco de bajada
        OUT     MCUCR, R16
        LDI     R16, 0B01000000 ; Habilita la INT0
        OUT     GICR, R16

        CLR     R16             ; Estado inicial de la salida
        OUT     PORTB, R16

        LDI     R17, 0B00000001 ; Para conmutar con OR exclusiva

        SEI                     ; Habilitador global de interrupciones

Lazo:   RJMP    Lazo            ; Lazo infinito, permanece ocioso
```

Se observa que el trabajo del programa principal prácticamente es nulo, el recurso de la interrupción externa es el encargado de monitorear al botón y en su ISR se realiza la tarea deseada.

En lenguaje C es necesario incluir a la biblioteca **interrupt.h** para el manejo de las interrupciones, la solución en este lenguaje es:

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

ISR(INT0_vect) {                // ISR de la INT0

        PORTB = PORTB ^ 0x01;   // OR exclusiva para conmutar al LSB
}

int     main() {

        DDRD = 0x00;            // Puerto D como entrada
        PORTD = 0xFF;           // Resistor de Pull-Up en el puerto D

        DDRB = 0xFF;            // Puerto B como salida
```

```
        MCUCR = 0B00000010;          // Configura INT0 por flanco de bajada

        GICR = 0B01000000;           // Habilita la INT0

        PORTB = 0x00;                // Estado inicial de la salida

        sei();                       // Habilitador global de interrupciones

        while(1) {                   // Lazo infinito, permanece ocioso
            asm( "nop");
        }
}
```

La función `sei()` es para poner en alto al habilitador global de interrupciones.

La inclusión de la instrucción **`nop`** en el lazo infinito hace posible una simulación por pasos en el AVR Studio, si se omite, al no haber instrucciones dentro del **`while`**, no es posible simular la introducción de eventos en PD2 que produzcan la interrupción. Aunque el código máquina generado trabaja de manera correcta en el MCU.

El ejemplo siguiente muestra cómo una variable puede ser modificada por dos rutinas de atención a interrupciones diferentes.

Ejemplo 4.2: Realice un contador de eventos ascendente/descendente con salida en binario. Considere a las terminales INT0 e INT1 para el ingreso de eventos y muestre la salida en el puerto B, el hardware sugerido se muestra en la figura 4.2.

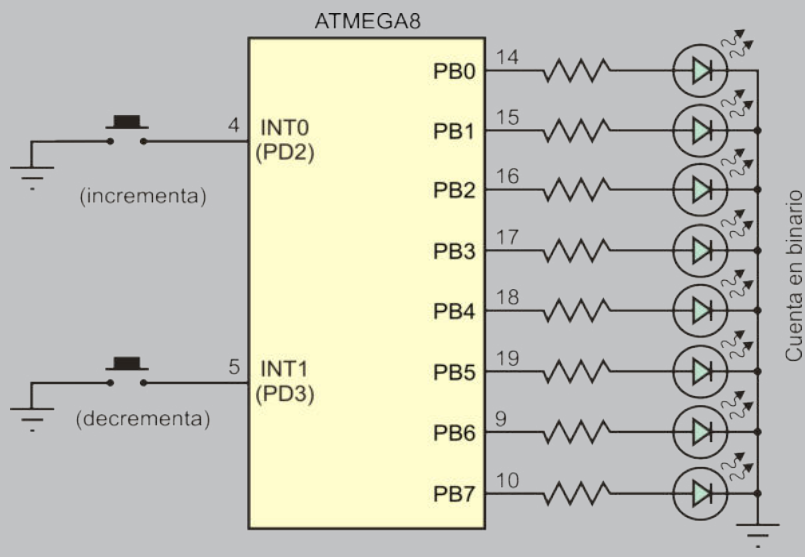


Figura 4.2 Manejo de dos interrupciones externas

Ambas interrupciones se configuran por flanco de bajada. En las ISRs se realiza la modificación del contador y la actualización de la salida, dejando ocioso al programa principal.

Para la solución en ensamblador, el valor del contador se lleva en el registro R17. La solución en ensamblador es:

```
       .include  <m8def.inc>      ; Biblioteca con definiciones

       .ORG   0x000               ; Vector de reset
       RJMP   Inicio

; Vectores de interrupciones
       .ORG   0x001               ; Vector de la interrupción externa 0
       RJMP   ISR_INT0

       .ORG   0x002               ; Vector de la interrupción externa 1
       RJMP   ISR_INT1

Inicio:                           ; Inicializaciones
       LDI    R16, 0x00
       OUT    DDRD, R16           ; Puerto D como entrada
       LDI    R16, 0xFF
       OUT    PORTD, R16          ; Resistor de Pull-Up en el puerto D

       OUT    DDRB, R16           ; Puerto B como salida

       LDI    R16, 0x04           ; Inicializa al apuntador de pila
       OUT    SPH, R16
       LDI    R16, 0x5F
       OUT    SPL, R16

       LDI    R16, 0B00001010     ; Configura INT1/INT0 por flanco de bajada
       OUT    MCUCR, R16
       LDI    R16, 0B11000000     ; Habilita INT1/INT0
       OUT    GICR, R16

       CLR    R17                 ; El contador inicia en 0
       OUT    PORTB, R17          ; Valor inicial de la salida
       SEI                        ; Habilitador global de interrupciones

Lazo:  RJMP   Lazo                ; Lazo infinito, permanece ocioso

ISR_INT0:                         ; Rutina de atención a la INT0
       INC    R17                 ; Incrementa al contador
       OUT    PORTB, R17          ; Actualiza la salida
       RETI

ISR_INT1:                         ; Rutina de atención a la INT1
       DEC    R17                 ; Reduce al contador
       OUT    PORTB, R17          ; Actualiza la salida
       RETI
```

En lenguaje C, debe manejarse una variable global con el valor del contador, para que pueda ser modificada por ambas rutinas de atención a las interrupciones. La solución en lenguaje C es:

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

unsigned  char cuenta;              // Variable global con el valor del contador

ISR(INT0_vect) {                    // ISR de la INT0
       cuenta++;                    // Incrementa al contador
       PORTB = cuenta;              // Actualiza la salida
}

ISR(INT1_vect) {                    // ISR de la INT1
       cuenta--;                    // Reduce al contador
       PORTB = cuenta;              // Actualiza la salida
}

int    main() {

       DDRD = 0x00;                 // Puerto D como entrada
       PORTD = 0xFF;                // Resistor de Pull-Up en el puerto D

       DDRB = 0xFF;                 // Puerto B como salida

       MCUCR = 0B00001010;          // Configura INT1/INT0 por flanco de bajada

       GICR = 0B11000000;           // Habilita INT1 e INT0

       cuenta = 0;                  // Valor inicial del contador
       PORTB = cuenta;              // Inicializa la salida

       sei();                       // Habilitador global de interrupciones

       while(1) {                   // Lazo infinito, permanece ocioso
           asm( "nop");
       }
}
```

En lenguaje C, el optimizador de código del compilador aplica algunas reglas que pueden generar un comportamiento erróneo, cuando se manejan variables globales que se modifican en las ISRs.

Si la solución al ejemplo anterior se organiza de la siguiente manera:

```
unsigned  char cuenta;

ISR(INT0_vect) {                    // ISR de la INT0
       cuenta++;                    // Incrementa al contador
}

ISR(INT1_vect) {                    // ISR de la INT1
       cuenta--;                    // Reduce al contador
}
```

```
int    main() {

       . . .

       while(1) {
           PORTB = cuenta;
       }
}
```

Se esperaría que las ISRs modificaran a la variable `cuenta` y que posteriormente ese valor fuera actualizado en el puerto B dentro del programa principal. Esto no ocurre, la variable se modifica pero el puerto B no se actualiza. Esto se debe a que al hacer una revisión del programa principal y no encontrar cambios en la variable `cuenta`, el optimizador de código realiza una asignación única para **PORTB**.

Para resolver este problema, a la variable `cuenta` se le debe agregar el modificador **volatile**, declarándola de la siguiente manera:

```
volatile unsigned  char cuenta;
```

Con ello, se indica que la variable es modificada en diferentes ISRs y por lo tanto, no debe ser considerada con una asignación única dentro del programa principal.

En ambos ejemplos no se agregaron diagramas de flujo porque las soluciones se basaron en el hardware de las interrupciones externas y en sus ISRs. En el programa principal únicamente se realizaron las siguientes tareas:

1. Ubicación del apuntador de pila (versión en ensamblador)
2. Configuración de Entradas y Salidas
3. Configuración de la interrupción o interrupciones
4. Habilitación de las interrupciones externas
5. Activación del habilitador global de interrupciones (bit **I** en **SREG**)
6. En el lazo infinito, permanece ocioso

El trabajo de las ISRs en estos ejemplos es muy simple, en sistemas con mayor complejidad es conveniente realizar un diagrama de flujo para cada ISR. A diferencia del programa principal, que entra en un lazo infinito y jamás termina, las ISRs si tienen un final bien definido.

4.2 Temporizadores

Una labor habitual de los controladores es la determinación y uso de intervalos de tiempo concretos. Esto se hace a través de un recurso denominado Temporizador (*Timer*), el cual básicamente es un registro de *n*-bits que se incrementa de manera automática en cada ciclo de reloj.

El recurso puede ser configurado para que el registro se incremente en respuesta a eventos externos, en esos casos suele ser referido como un Contador de eventos (*Counter*), no obstante, por simplicidad, en este libro siempre es tratado como Temporizador, independientemente de que la temporización esté dada por eventos internos o externos.

Tanto el ATMega8 como el ATMega16 incluyen 3 temporizadores, 2 son de 8 bits y 1 de 16 bits. En la tabla 4.4 se listan los Registros I/O para cada temporizador, como los Registros I/O son de 8 bits, el temporizador 1 utiliza 2 de ellos.

Tabla 4.4 Registros de los temporizadores en los AVR

Temporizador	Tamaño	Registros	Dirección
Timer 0	8 bits	**TCNT0**	0x32
Timer 1	16 bits	**TCNT1H** : **TCNT1L**	0x2D, 0x2C
Timer 2	8 bits	**TCNT2**	0x24

4.2.1 Eventos de los Temporizadores

En los microcontroladores AVR, los eventos que se pueden generar por medio de los temporizadores son: Desbordamientos, coincidencias por comparación y captura de entrada. La ocurrencia de alguno de estos eventos se ve reflejada en el registro de banderas de interrupción (**TIFR**, *Timer Interupt Flag Register*), cuyo contenido es descrito en la sección 4.2.5.

4.2.1.1 Desbordamientos

Este evento ocurre cuando alguno de los temporizadores (**TCNT*n***) alcanza su valor máximo (*MAXVAL*) y se reinicia con 0, es decir, ocurre con una transición de 1's a 0's, esta transición provoca que una bandera (**TOV*n***) sea puesta en alto. Una aplicación puede sondear en espera de un nivel alto en la bandera, o bien, se puede configurar al hardware para que el evento produzca una interrupción.

Esta señalización indica que ha transcurrido un intervalo de tiempo o un número específico de eventos, el conteo tiene flexibilidad porque es posible cargar al registro para que inicie desde un valor determinado. Si se utiliza una carga, ésta debe repetirse cada vez que el evento es producido, para generar intervalos regulares.

En la figura 4.3 se esquematiza al temporizador generando eventos por desbordamiento,

se muestra la posibilidad de una carga paralela y la generación de la bandera.

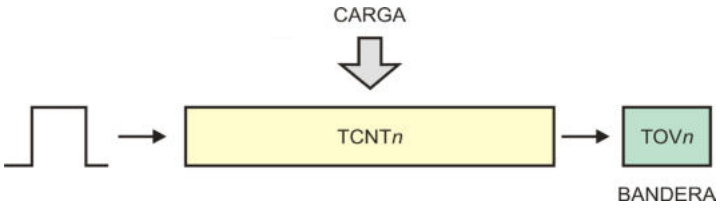


Figura 4.3 Desbordamientos en los temporizadores (*n = 0, 1, 2*)

Los desbordamientos pueden ser manejados por los 3 temporizadores (en la figura 4.3 *n* puede ser 0, 1 y 2). De hecho, los eventos por desbordamientos son un *de facto* en los temporizadores de los microcontroladores de diferentes fabricantes.

El valor máximo depende del número de bits del temporizador, queda determinado con la expresión:

$$MAXVAL = 2^{Tamaño(TCNTn)} - 1$$

4.2.1.2 Coincidencias por Comparación

En el hardware se dedica a un registro para comparaciones continuas (**OCRn**, *Output Compare Register*), en este registro se puede cargar un valor entre 0 y MAXVAL. En cada ciclo de reloj se compara al registro del temporizador con el registro de comparación, una coincidencia es un evento que se indica con la puesta en alto de una bandera (**OCF*n***, *Output Compare Flag*). La bandera puede sondearse por software o configurar al recurso para que genere una interrupción. En la figura 4.4 se esquematiza la generación de estos eventos.

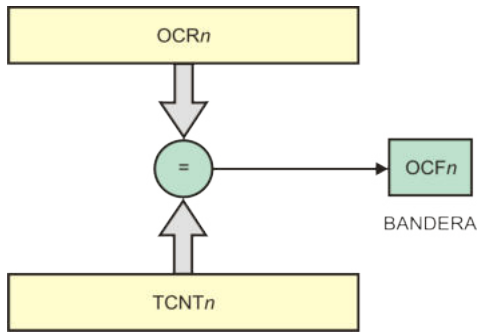


Figura 4.4 Coincidencias por comparación en los temporizadores

En un ATMega8 estos eventos pueden ser generados por los temporizadores 1 y 2, en un ATMega16 por los 3 temporizadores. En ambos microcontroladores, para el temporizador 1 se tienen 2 registros de comparación (**OCR1A** y **OCR1B**), por lo que con este temporizador se pueden manejar comparaciones con 2 valores diferentes.

Es posible configurar al temporizador para que se reinicie después de una coincidencia en la comparación y algunas terminales relacionadas pueden ajustarse, limpiarse o conmutarse automáticamente en cada coincidencia.

4.2.1.3 Captura de Entrada

Este tipo de eventos sólo es manejado por el temporizador 1, se tiene una terminal dedicada a capturar eventos externos (ICP, *Input Capture Pin*), un cambio en esta terminal provoca la lectura del temporizador y su almacenamiento en el registro de captura de entrada (**ICR**, *Input Capture Register*). El tipo de transición en ICP para generar las capturas es configurable, puede ser un flanco de subida o uno de bajada. En la figura 4.5 se muestra como se producen estos eventos.

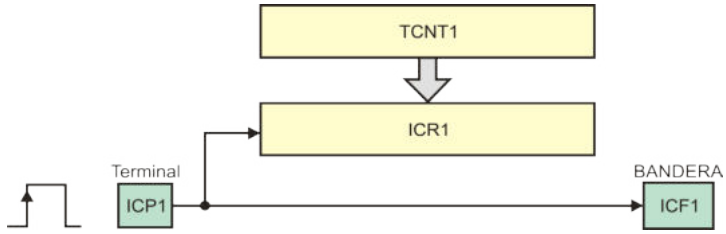


Figura 4.5 Captura de entrada

La bandera de captura de entrada (**ICF**, *Input Capture Flag*) es puesta en alto indicando la ocurrencia del evento, esta bandera puede sondearse por software o se puede configurar al recurso para que genere una interrupción. Este evento puede ser útil para medir el ancho de un pulso externo.

4.2.2 Respuesta a los Eventos

Existen 3 formas para detectar los eventos producidos por los temporizadores y actuar ante ellos.

4.2.2.1 Sondeo (*Polling*)

El programa trabaja a un solo nivel (de acuerdo con la figura 2.13), en el cual destina un conjunto de instrucciones para evaluar de manera frecuente el estado de las banderas. Este método es el menos eficiente porque requiere de instrucciones adicionales en el programa principal e implica tiempo de procesamiento.

Como ejemplo, en la tabla 4.5 se muestra una secuencia de instrucciones que espera un evento por desbordamiento del temporizador 0. Al utilizar sondeo, las banderas se deben limpiar por software después de que ocurra el evento esperado, para ello se les debe escribir un 1 lógico.

Tabla 4.5 Detección por sondeo de un desbordamiento del temporizador 0

Versión en ensamblador

```
LOOP:  IN     R16, TIFR     ; Lee banderas
       SBRS   R16, TOV0     ; Brinca si la bandera TOV0 está en alto
       RJMP   LOOP
                            ; Regresa a esperar la bandera
       OUT    TIFR, R16     ; Limpia la bandera
       . . .                ; Continúa
```

Versión en Lenguaje C

```
       while  ( ! ( TIFR & 1 << TOV0 ) )  // Espera a la bandera
                    ;
       TIFR |= 1 << TOV0;                 // Limpia la bandera
       . . .
```

En una aplicación real, se ejecutan otras instrucciones mientras no ocurre el evento, para no dejar al programa dedicado únicamente a esperar y atender al evento.

4.2.2.2 Uso de Interrupciones

Todos los eventos de los temporizadores pueden generar interrupciones, para ello, éstas se deben activar en el registro de enmascaramiento de interrupciones (**TIMSK**, *Timer Interrupt Mask Register*), además de activar al habilitador global de interrupciones (bit **I** en **SREG**).

Con ello, el programa principal se ejecuta de manera normal, cuando ocurre un evento se concluye con la instrucción en proceso para dar paso a la ISR correspondiente y al concluir con su ejecución, el programa continúa con la instrucción siguiente a aquella que se estaba ejecutando cuando ocurrió el evento.

Esta forma de dar atención a los eventos es de las más ampliamente usadas, es eficiente porque en el programa principal no se invierte tiempo de procesamiento para esperar la ocurrencia de eventos.

También es conveniente para atender eventos de otros recursos. Cuando se emplean interrupciones, las banderas son limpiadas automáticamente por hardware.

4.2.2.3 Respuesta Automática

Los temporizadores 1 y 2 (y el 0 en el ATMega16) incluyen un módulo de generación de formas de onda utilizado para reaccionar únicamente por hardware ante eventos de comparación. Esto significa que con una coincidencia, automáticamente se modifican las terminales de comparación de salida (OC, *Output Compare*) para ponerse en alto, en bajo o conmutarse.

Con ello, en el software sólo se incluye la configuración del recurso y la atención a eventos se realiza de manera paralela a la ejecución del programa principal, sin requerir de instrucciones adicionales.

4.2.3 Pre-escalador

Un pre-escalador básicamente es un divisor de frecuencia que se antepone a los registros de los temporizadores proporcionándoles la capacidad de alcanzar intervalos de tiempo mayores. Un pre-escalador incluye 2 componentes, un contador de *n*-bits y un multiplexor para seleccionar diferentes posiciones de bit en el contador. El contador se incrementa en cada ciclo de reloj y por lo tanto, con el multiplexor pueden seleccionarse diferentes frecuencias.

Los microcontroladores AVR incluyen 2 pre-escaladores, uno compartido por los temporizadores 0 y 1, y el otro sólo utilizado por el temporizador 2. En ambos, el contador es de 10 bits y el multiplexor es de 8 a 1, pero difieren en su organización.

En la figura 4.6 se muestra la organización del pre-escalador compartido por los temporizadores 0 y 1, puede notarse que en realidad sólo comparten al contador, porque el hardware de selección es independiente.

Los bits de selección (**CS*xn***) están en los registros de configuración de los temporizadores. Después de un reinicio no hay señal de reloj porque tienen el valor de 0. Con estos bits puede seleccionarse la señal de reloj sin división, la señal de reloj con un factor de división entre 4 diferentes y 1 señal externa o su complemento (flancos de subida o bajada).

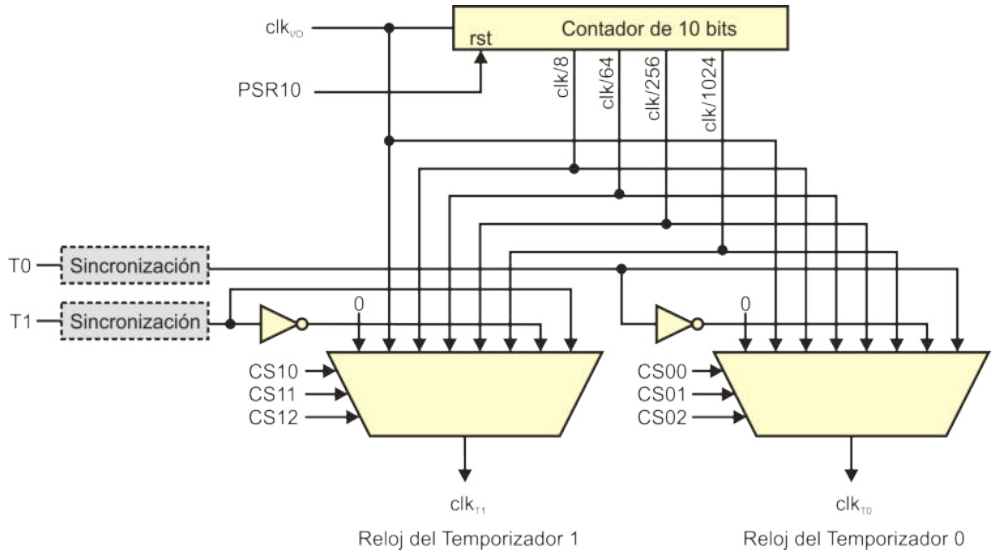


Figura 4.6 Pre-escalador compartido por los temporizadores 0 y 1

En la figura 4.7 se muestra la organización del pre-escalador utilizado por el temporizador 2, se observa que tiene 6 factores de división y que la señal externa es generada por un oscilador externo (TOSC1), permitiendo que el temporizador 2 trabaje a una frecuencia diferente al resto del sistema.

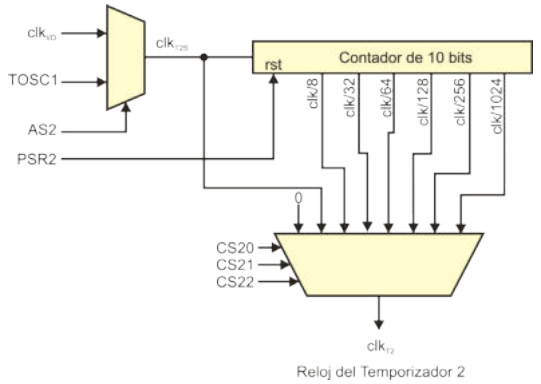


Figura 4.7 Pre-escalador del temporizador 2

Los contadores en los pre-escaladores tienen su señal de reinicio, la cual se activa con los bits **PSR10** y **PSR2**, estos bits corresponden con los 2 bits menos significativos del registro de función especial (**SFIOR**, *Special Function IO Register*), como se muestra a continuación:

	7	6	5	4	3	2	1	0	
0x30	ADTS2	ADTS1	ADTS0	-	ACME	PUD	PSR2	PSR10	**SFIOR**

Los bits se ponen en alto por software para reiniciar a los contadores, y después de ello, automáticamente son limpiados por hardware. Por lo que si se realiza una lectura, siempre se obtiene el valor de 0.

4.2.4 Temporización Externa

Los temporizadores 0 y 1 pueden ser manejados por señales externas, T0 y T1, respectivamente. En estos casos se les conoce como contadores de eventos. Esto se mostró en la figura 4.6, sin embargo, en la figura 4.8 se muestran detalles de cómo se realiza la sincronización y la detección del flanco (subida o bajada), la etapa de sincronización asegura una señal estable a la etapa de detección de flanco, la cual se encarga de generar un pulso limpio cuando ocurre el flanco seleccionado. La presencia de estos módulos hace necesario que la frecuencia de los eventos externos esté limitada por $f_{clk\_I/O}/2.5$.

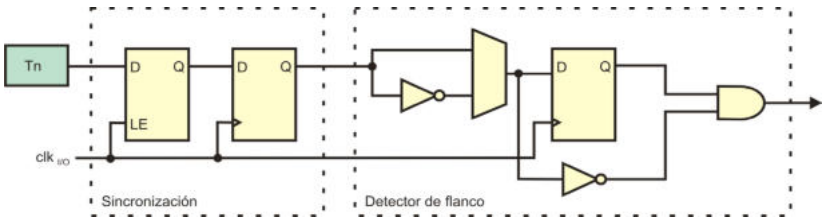


Figura 4.8 Acondicionamiento de las señales externas, para los temporizadores 0 y 1

El temporizador 2 también puede ser manejado por una señal externa, pero en este caso, se utiliza un oscilador externo que no se sincroniza con el oscilador interno (es asíncrono). El hardware se ha optimizado para ser manejado con un cristal de 32.768 KHz, esta frecuencia es adecuada para que, en combinación con el pre-escalador, se puedan generar fracciones o múltiplos de segundos. En la figura 4.9 se muestra cómo es posible seleccionar entre un oscilador externo o la señal de reloj interna.

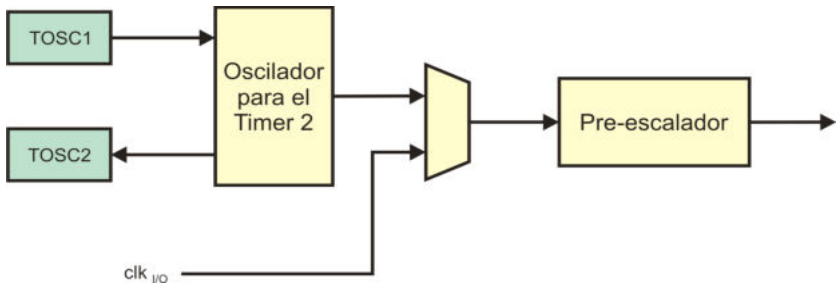


Figura 4.9 Uso de un oscilador externo para el temporizadores 2

La ventaja principal de un oscilador asíncrono para el temporizador 2 es que trabaja con una frecuencia independiente al resto del sistema, la cual es adecuada para manejar un reloj de tiempo real. Sin mucho esfuerzo, se puede realizar un reloj con displays de 7 segmentos y botones de configuración, de manera qué, mientras el temporizador 2 lleva el conteo de segundos, el resto del sistema está manejando a los elementos de visualización o sondeando los botones a una frecuencia mucho más alta.

4.2.5 Registros Compartidos por los Temporizadores

Como se describió en la sección 4.2.1, cuando ocurre un evento queda señalizado en el registro **TIFR**, el cual incluye los siguientes bits:

	7	6	5	4	3	2	1	0	
0x38	OCF2	TOV2	ICF1	OCF1A	OCF1B	TOV1	OCF0	TOV0	**TIFR**

- **Bit 7 – OCF2: Bandera de coincidencia por comparación en el temporizador 2**
- **Bit 6 – TOV2: Bandera de desbordamiento en el temporizador 2**
- **Bit 5 – ICF1: Bandera de captura de entrada en el temporizador 1**
- **Bit 4 – OCF1A: Bandera de coincidencia con el comparador A del temporizador 1**
- **Bit 3 – OCF1B: Bandera de coincidencia con el comparador B del temporizador 1**

El temporizador 1 puede ser comparado con 2 valores diferentes.

- **Bit 2 – TOV1: Bandera de desbordamiento en el temporizador 1**
- **Bit 1 – OCF0: Bandera de coincidencia por comparación en el temporizador 0**

  Este bit no está implementado en el ATMega8 porque su temporizador 0 no maneja los eventos de coincidencia por comparación.

- **Bits 0 – TOV0: Bandera de desbordamiento en el temporizador 0**

Otro registro compartido por los 3 temporizadores es el registro **TIMSK**, el cual, como se mencionó en la sección 4.2.2.2, es utilizado para habilitar las interrupciones por los diferentes eventos de los temporizadores. Los bits en el registro **TIMSK** tienen una organización similar a los bits del registro **TIFR**, éstos se describen a continuación:

	7	6	5	4	3	2	1	0	
0x39	OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	OCIE0	TOIE0	**TIMSK**

- **Bit 7 – OCIE2: Habilita la interrupción de coincidencia por comparación en el temporizador 2**
- **Bit 6 – TOIE2: Habilita la interrupción por desbordamiento del temporizador 2**
- **Bit 5 – TICIE1: Habilita la interrupción por captura de entrada en el temporizador 1**
- **Bit 4 – OCIE1A: Habilita la interrupción por coincidencia en el comparador A del temporizador 1**
- **Bit 3 – OCIE1B: Habilita la interrupción por coincidencia en el comparador B del temporizador 1**

  El temporizador 1 puede generar 2 interrupciones por comparación.

- **Bit 2 – TOIE1: Habilita la interrupción por desbordamiento del temporizador 1**
- **Bit 1 – OCIE0: Habilita la interrupción de coincidencia por comparación en el temporizador 0**

  En el ATMega8 este bit tampoco está implementado.

- **Bit 0 – TOIE0: Habilita la interrupción por desbordamiento del temporizador 0**

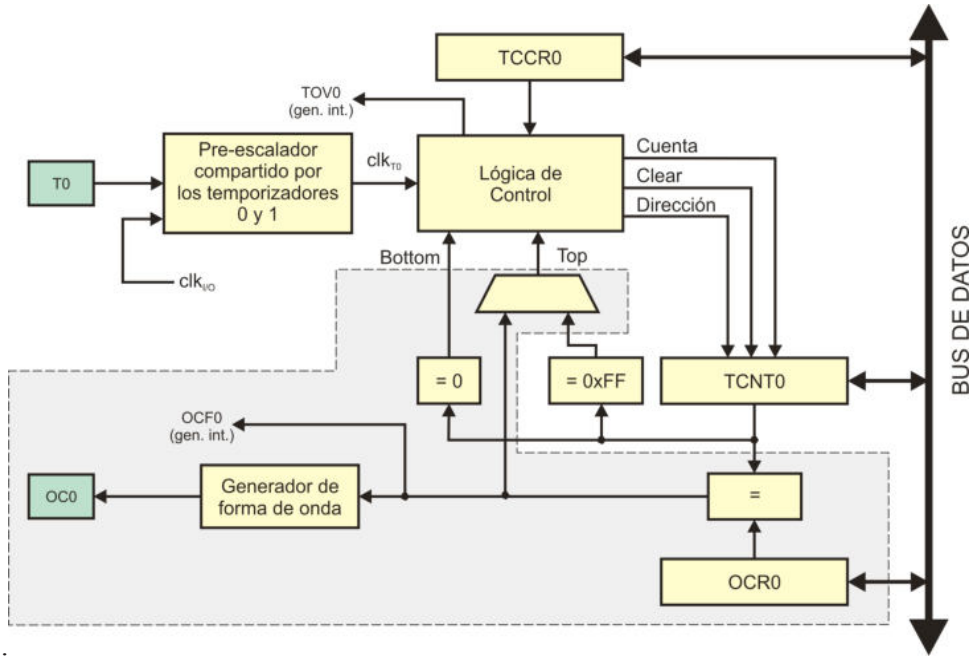
En un ATMega8 los temporizadores pueden generar 7 interrupciones diferentes y para un ATMega16 son 8, en la tabla 4.6 se muestran los vectores de las interrupciones relacionadas.

Tabla 4.6 Vectores de Interrupciones relacionadas con los temporizadores

Dirección		Evento	Descripción	Temporizador
ATMega8	ATMega16			
0x003	0x006	TIMER2_COMP	Coincidencia por comparación	2
0x004	0x008	TIMER2_OVF	Desbordamiento	2
0x005	0x00A	TIMER1_CAPT	Captura	1
0x006	0x00C	TIMER1_COMPA	Coincidencia en el comparador A	1
0x007	0x00E	TIMER1_COMPB	Coincidencia en el comparador B	1
0x008	0x010	TIMER1_OVF	Desbordamiento	1
0x009	0x012	TIMER0_OVF	Desbordamiento	0
	0x026	TIMER0_COMP	Coincidencia por comparación	0

4.2.6 Organización y Registros del Temporizador 0

El temporizador 0 es de 8 bits, su organización se muestra en la figura 4.10, en donde se observa que **TCNT0** (**0x32**) es el registro a incrementar y **OCR0** (**0x3C**, no implementado en el ATMega8) es el registro con el que se realiza la comparación en cada ciclo de reloj



.

Figura 4.10 Organización del temporizador 0, la línea interrumpida encierra los recursos no disponibles en el ATMega8

En el ATMega8 sólo se pueden generar eventos de desbordamientos. En el ATMega16 también se pueden generar eventos de coincidencias por comparación y dar respuesta automática en la terminal OC0, en la figura 4.10 se han encerrado los recursos que no están en el ATMega8.

El temporizador 0 es controlado por el registro **TCCR0** (*Timer/Counter Control Register 0*), cuyos bits son descritos a continuación, cabe señalar que los bits del 3 al 7 se relacionan con los recursos de comparación y no están implementados en el ATMega8.

	7	6	5	4	3	2	1	0	
0x33	FOC0	WGM00	COM01	COM00	WGM01	CS02	CS01	CS00	TCCR0

- **Bit 7 – FOC0: Obliga un evento de coincidencia por comparación (*Force Output Compare*)**

  La puesta en alto de este bit obliga a que ocurra un evento de coincidencia por comparación, incluyendo una respuesta automática en OC0, si es que fue configurada.

- **Bits 6 y 3 – WGM0[0:1]: Determinan el modo de operación del generador de formas de onda (*Waveform Generation Mode*)**

  Con estos 2 bits se puede seleccionar 1 de 4 modos de operación, éstos se describen en la sección 4.2.6.1

- **Bits 5 y 4 – COM0[1:0]: Configuran la respuesta automática en la terminal OC0 (*Compare Output Mode*)**

  Esta configuración se describe en la sección 4.2.6.2.

- **Bits 2, 1 y 0 – CS0[2:0]: Seleccionan la fuente de temporización (*Clock Select*)**

  Estos bits determinan la selección en el pre-escalador (sección 4.2.3) y se describen en la sección 4.2.6.3.

Con respecto a la figura 4.10, el bloque marcado como Lógica de Control determina el comportamiento del temporizador 0 a partir del valor de los bits en el registro **TCCR0**, después de un reinicio, el registro contiene ceros.

4.2.6.1 Generación de Formas de Onda con el Temporizador 0

En la tabla 4.7 se muestran los 4 posibles modos de generación de forma de onda, determinados por los bits **WGM0[1:0]**.

Tabla 4.7 Modos de generación de forma de onda

Modo	WGM01	WGM00	Descripción
0	0	0	Normal
1	0	1	PWM con fase correcta
2	1	0	CTC: Limpia al temporizador ante una coincidencia por comparación
3	1	1	PWM rápido

- **Modo 0:** Operación normal del temporizador, sólo se generan eventos de desbordamientos.
- **Modos 1 y 3:** Modos para la generación de PWM, en la sección 4.3 se describen todos los aspectos relacionados con PWM.
- **Modo 2:** Limpia al registro del temporizador (coloca 0's en **TCNT0**) tras una coincidencia en la comparación, se genera la bandera **OCF0** debido a que hubo una coincidencia.

4.2.6.2 Respuesta Automática en la Terminal OC0

En el modo normal o en el modo CTC, los bits **COM0[1:0]** definen el comportamiento descrito en la tabla 4.8 para la salida OC0, proporcionando una respuesta automática ante un evento de comparación. La respuesta en los modos de PWM se describe en la sección 4.3.

Tabla 4.8 Respuesta automática en OC0

COM01	COM00	Descripción
0	0	Operación Normal, terminal OC0 desconectada
0	1	Conmuta a OC0, tras una coincidencia por comparación
1	0	Limpia a OC0, tras una coincidencia por comparación
1	1	Pone en alto a OC0, tras una coincidencia por comparación

4.2.6.3 Selección del Reloj para el Temporizador 0

La selección de la señal de reloj está determinada por los bits **CS0[2:0]**, los cuales son descritos en la tabla 4.9. Estos bits se conectan directamente a los bits de selección del multiplexor del pre-escalador (sección 4.2.3). Después de un reinicio, el temporizador 0 está detenido porque no hay fuente de temporización.

Tabla 4.9 Bits para la selección del reloj en el temporizador 0

CS02	CS01	CS00	Descripción
0	0	0	Sin fuente de reloj (temporizador 0 detenido)
0	0	1	$clk_{I/O}$ (sin división)
0	1	0	$clk_{I/O}/8$ (del pre-escalador)
0	1	1	$clk_{I/O}/64$ (del pre-escalador)
1	0	0	$clk_{I/O}/256$ (del pre-escalador)
1	0	1	$clk_{I/O}/1024$ (del pre-escalador)
1	1	0	Fuente externa en T0, por flanco de bajada
1	1	1	Fuente externa en T0, por flanco de subida

4.2.7 Organización y Registros del Temporizador 1

La organización del temporizador 1 se muestra en la figura 4.11, este temporizador es el más completo porque es de 16 bits y puede manejar 3 tipos de eventos: por desbordamientos, por comparaciones con 2 valores diferentes y por captura de una entrada externa.

En la figura 4.11 se pueden ver 4 registros de 16 bits, cada uno requiere de 2 localidades en el espacio de Registros I/O: **TCNT1** (**0x2D:0x2C**) es el registro a incrementar, **OCR1A** (**0x2B:0x2A**) es uno de los registros con el que se realizan comparaciones, **OCR1B** (**0x29:0x28**) es el otro registro para las comparaciones e **ICR1** (**0x27:0x26**) es el registro para realizar capturas.

En lenguaje C se puede tener acceso directo a los 16 bits de cada uno de estos registros, en ensamblador se debe seguir un orden para las lecturas y escrituras, el cual se describe y justifica en la sección 4.2.7.4.

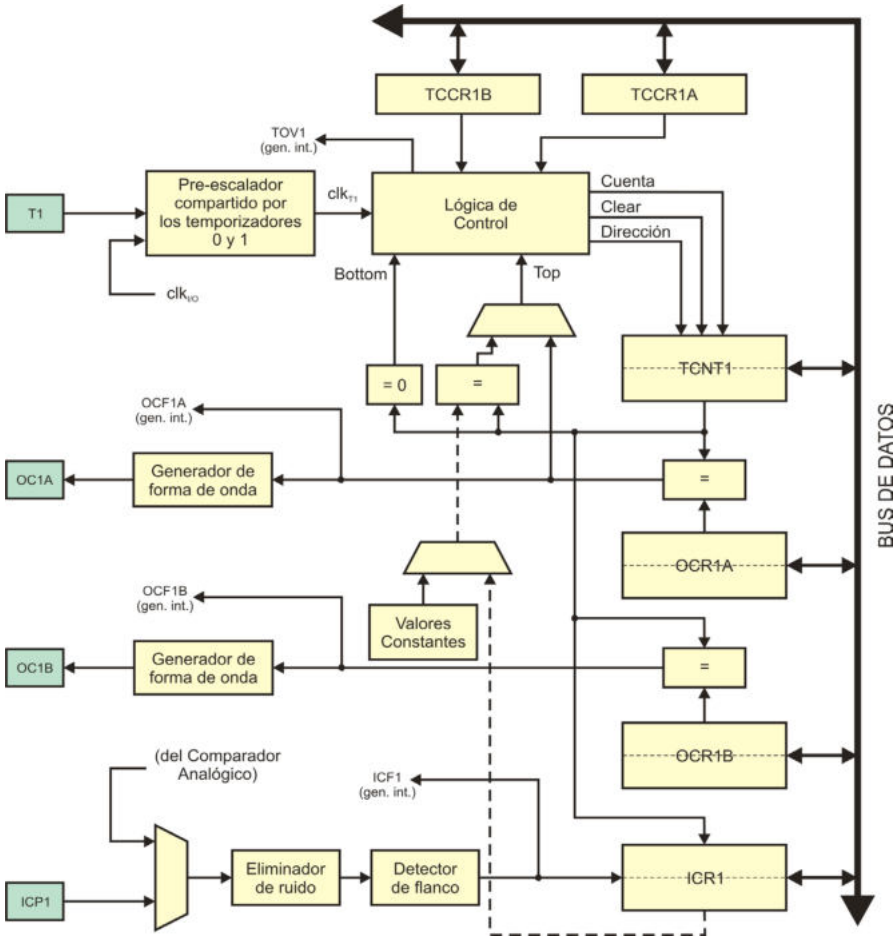


Figura 4.11 Organización del temporizador 1, es la misma para el ATMega8 y el ATMega16

El temporizador 1 es controlado por los registros **TCCR1A** y **TCCR1B** (*Timer/Counter Control Register 1A y 1B*), cuyos bits son descritos a continuación:

	7	6	5	4	3	2	1	0	
0x2F	COM1A1	COM1A0	COM1B1	COM1B0	FOC1A	FOC1B	WGM11	WGM10	**TCCR1A**

- **Bits 7 y 6 – COM1A[1:0]: Configuran la respuesta automática en la terminal OC1A**

  Para dar respuesta al comparador A, esta configuración se describe en la sección 4.2.7.2.

- **Bits 5 y 4 – COM1B[1:0]: Configuran la respuesta automática en la terminal OC1B**

  Para dar respuesta al comparador B, esta configuración se describe en la sección 4.2.7.2.

- **Bit 3 – FOC1A: Obliga un evento de coincidencia para el comparador A**

  La puesta en alto de este bit obliga a que ocurra un evento de coincidencia con el comparador A, incluyendo una respuesta automática en OC1A, si es que fue configurada.

- **Bit 2 – FOC1B: Obliga un evento de coincidencia para el comparador B**

  La puesta en alto de este bit obliga a que ocurra un evento de coincidencia con el comparador B, incluyendo una respuesta automática en OC1B, si es que fue configurada.

- **Bits 1 y 0 – WGM1[1:0]: Determinan el modo de operación del generador de formas de onda**

  Estos bits, en conjunción con otros 2 bits de **TCCR1B**, permiten seleccionar uno de sus modos de operación, éstos se describen en la sección 4.2.7.1.

	7	6	5	4	3	2	1	0	
0x2E	ICNC1	ICES1	-	WGM13	WGM12	CS12	CS11	CS10	**TCCR1B**

- **Bit 7 – ICNC1: Activa un eliminador de ruido para captura de entrada (*Input Capture Noise Canceler*)**

  Si el bit está activo, la entrada se filtra, de manera que, un cambio es válido si después de la transición, la señal se mantiene estable por cuatro muestras consecutivas. Esto retrasa un evento de captura por cuatro ciclos de reloj.

- **Bit 6 – ICES1: Selecciona el flanco de activación de la captura (*Input Capture Edge Select*)**

  Un evento externo en la terminal ICP1 hace que el temporizador 1 se copie en ICR1 y que la bandera ICF1 sea puesta en alto. Con este bit en bajo el evento es una transición de bajada y con el bit en alto es una transición de subida.

- **Bit 5 – No está implementado**

- **Bits 4 y 3 – WGM1[3:2]: Determinan el modo de operación del generador de formas de onda**

  Estos bits, en conjunción con otros 2 bits de **TCCR1A**, permiten seleccionar uno de sus modos de operación, éstos se describen en la sección 4.2.7.1

- **Bits 2, 1 y 0 – CS1[2:0]: Seleccionan la fuente de temporización (*Clock Select*)**

  Estos bits determinan la selección en el pre-escalador (sección 4.2.3) y se describen en la sección 4.2.7.3.

4.2.7.1 Generación de Formas de Onda con el Temporizador 1

El temporizador 1 incluye a los bits **WGM1[3:0]** para seleccionar entre 16 posibles modos de generación de forma de onda, no obstante, sólo 4 no están relacionados con PWM. En la tabla 4.10 se listan estos 4 modos y los 12 restantes se describen en la sección 4.3.

Tabla 4.10 Modos de generación de forma de onda que no están relacionados con PWM

Modo	WGM13	WGM12	WGM11	WGM10	Descripción
0	0	0	0	0	Normal
4	0	1	0	0	CTC: Limpia al temporizador ante una coincidencia con OCR1A
12	1	1	0	0	CTC: Limpia al temporizador ante una coincidencia con ICR1
13	1	1	0	1	Reservado

- **Modo 0:** Operación normal del temporizador, sólo se generan eventos de desbordamientos.

- **Modos 4 y 12:** Limpia al registro del temporizador (coloca 0's en **TCNT1**) tras una coincidencia en la comparación. En el modo 4 la comparación es con el registro **OCR1A** y en el modo 12 es con **ICR1**. En el modo 4, al ocurrir la coincidencia con **OCR1A** se genera la bandera **OCF1A**.

- **Modo 13:** Sin uso, en estas versiones de dispositivos.

En el modo 12, el registro **ICR1** contiene el valor máximo para el temporizador. Al asignarle una función alterna a este registro, la terminal ICP1 sólo funciona como entrada o salida general, de manera que no se pueden realizar tareas de captura.

4.1.2.1 Respuesta Automática en las Terminales OC1A y OC1B

En el modo normal y en los modos CTC, los bits **COM1A[1:0]** y **COM1B[1:0]** definen el comportamiento descrito en la tabla 4.11 para las salidas OC1A y OC1B, respectivamente, proporcionando una respuesta automática ante eventos de comparación.

Tabla 4.11 Respuesta automática en OC1A/OC1B

COM1A1 / COM1B1	COM1A0 / COM1B0	Descripción
0	0	Normal, terminales OC1A/OC1B desconectadas
0	1	Conmuta a OC1A/OC1B, tras una coincidencia por comparación
1	0	Limpia a OC1A/OC1B, tras una coincidencia por comparación
1	1	Pone en alto a OC1A/OC1B, tras una coincidencia por comparación

El comportamiento de las terminales OC1A/OC1B en los modos PWM se describe en la sección 4.3.

4.2.7.3 Selección del Reloj para el Temporizador 1

La selección de la señal de reloj está determinada por los bits **CS1[2:0]**, los cuales son descritos en la tabla 4.12. Estos bits se conectan directamente a los bits de selección del multiplexor del pre-escalador (sección 4.2.3). Después de un reinicio, el temporizador 1 está detenido porque no hay fuente de temporización.

Tabla 4.12 Bits para la selección del reloj en el temporizador 1

CS12	CS11	CS10	Descripción
0	0	0	Sin fuente de reloj (temporizador 1 detenido)
0	0	1	$clk_{I/O}$ (sin división)
0	1	0	$clk_{I/O}/8$ (del pre-escalador)
0	1	1	$clk_{I/O}/64$ (del pre-escalador)
1	0	0	$clk_{I/O}/256$ (del pre-escalador)
1	0	1	$clk_{I/O}/1024$ (del pre-escalador)
1	1	0	Fuente externa en T1, por flanco de bajada
1	1	1	Fuente externa en T1, por flanco de subida

4.2.7.4 Acceso a los Registros de 16 Bits del Temporizador 1

El temporizador 1 es de 16 bits y los Registros I/O son de 8 bits, por ello, el hardware incluye algunos mecanismos necesarios para su acceso, sin los cuales se presentaría un problema grave, porque el temporizador está cambiando en cada ciclo de reloj.

Para ilustrar este problema se asume que el hardware no incluye los citados mecanismos y que se requiere leer al temporizador 1, para dejar su contenido en los registros R17:R16. Si la lectura se realiza cuando el temporizador tiene

0x03FF, al mover la parte alta R17 queda con 0x03, en el siguiente ciclo de reloj el temporizador cambia a 0x0400, por lo que al leer la parte baja R16 queda con 0x00. Por lo tanto, el valor leído es 0x0300, que está muy lejos del valor real del temporizador.

Si la lectura se realiza en orden contrario, R16 queda con 0xFF y R17 con 0x04, el valor leído es de 0x04FF, que también muestra un error bastante significativo.

Para evitar estos conflictos y poder realizar escrituras y lecturas "al vuelo", es decir, sin detener al temporizador, se incorpora un registro temporal de 8 bits que no es visible al programador, el cual está conectado directamente con la parte alta del registro de 16 bits.

Cuando se escribe o lee la parte alta del temporizador, en realidad se hace la escritura o lectura en el registro temporal. Cuando se tiene acceso a la parte baja, se hacen lecturas o escrituras de 16 bits, tomando como parte alta al registro temporal. Esto significa que se debe tener un orden de acceso, una lectura debe iniciar con la parte baja y una escritura debe iniciar con la parte alta. El orden de las lecturas se ilustra con las instrucciones:

```
IN   R16, TCNT1L     ; lee 16 bits, el byte alto queda en el registro temporal
IN   R17, TCNT1H     ; lee del registro temporal
```

El orden para las escrituras se ilustra en la siguiente secuencia, con la que se escribe 1500 en el registro del temporizador 1:

```
LDI  R16, LOW(1500)    ; Escribe el byte bajo de la constante
LDI  R17, HIGH(1500)   ; Escribe el byte alto de la constante
OUT  TCNT1H, R17       ; Escribe en el registro temporal
OUT  TCNT1L, R16       ; Escritura de 16 bits
```

No es posible tener acceso sólo a un byte del temporizador.

En alto nivel no es necesario revisar el orden de acceso, puesto que se pueden emplear datos de 16 bits, pudiendo ser variables o registros.

4.1.3 Organización y Registros del Temporizador 2

El temporizador 2 es de 8 bits, difiere del temporizador 0 de un ATMega16 por su capacidad de sincronizarse con un oscilador externo, cuya salida posteriormente puede ser procesada por el pre-escalador. En la figura 4.12 se muestra su organización, donde se observa que el registro **TCNT2** (**0x24**) es el que se incrementa y el registro **OCR2** (**0x23**) es empleado para comparaciones.

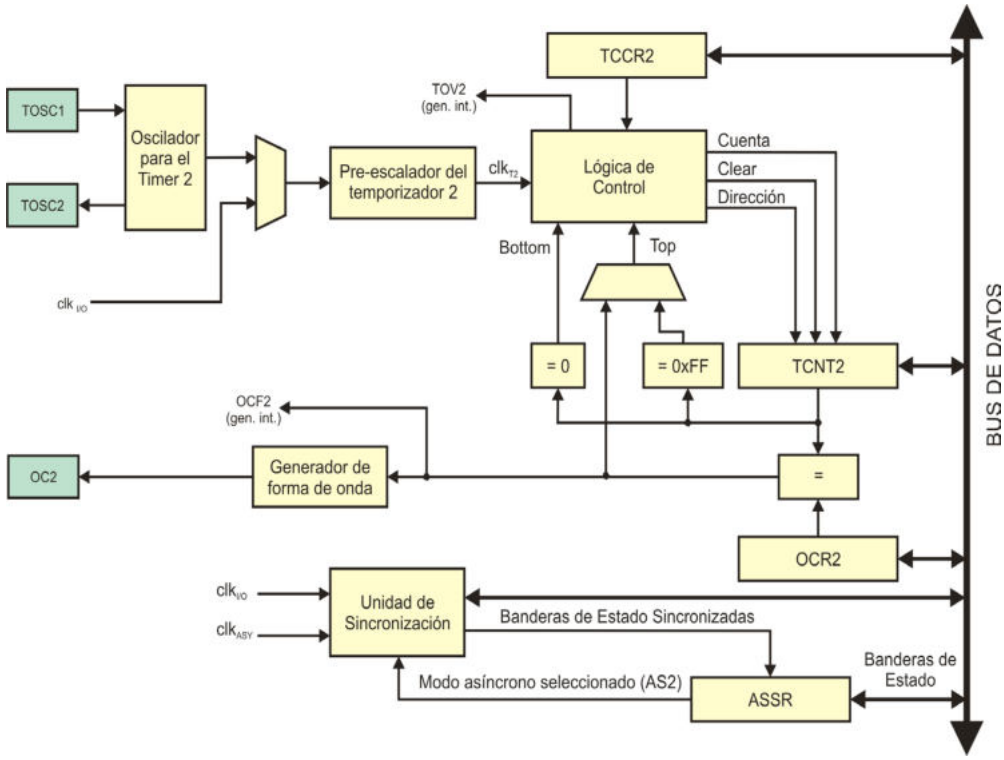


Figura 4.12 Organización del temporizador 2, es la misma para el ATMega8 y el ATMega16

El temporizador 2 es controlado por el registro **TCCR2** (*Timer/Counter Control Register 2*), cuyos bits se describen a continuación:

	7	6	5	4	3	2	1	0	
0x25	FOC2	WGM20	COM21	COM20	WGM21	CS22	CS21	CS20	**TCCR2**

- **Bit 7 – FOC2: Obliga un evento de coincidencia por comparación (*Force Output Compare*)**

  La puesta en alto de este bit obliga a que ocurra un evento de coincidencia por comparación, incluyendo una respuesta automática en OC2, si es que fue configurada.

- **Bits 6 y 3 – WGM2[0:1] Determinan el modo de operación del generador de formas de onda (*Waveform Generation Mode*)**

  Con estos 2 bits se puede seleccionar 1 de 4 modos de operación, éstos se describen en la sección 4.2.8.1

- **Bits 5 y 4 – COM2[1:0]: Configuran la respuesta automática en la terminal OC2 (*Compare Output Mode*)**

  Esta configuración se describe en la sección 4.2.8.2.

- **Bits 2, 1 y 0 – CS2[2:0] Seleccionan la fuente de temporización (*Clock Select*)**

  Estos bits determinan la selección en el pre-escalador (sección 4.2.3) y se describen en la sección 4.2.8.3.

En la figura 4.12 también se observa al registro de estado asíncrono (***ASSR***, *Asynchronous Status Register*), el cual sirve para habilitar la operación asíncrona del temporizador 2 y para conocer su estado, sus bits son:

	7	6	5	4	3	2	1	0	
0x22	-	-	-	-	AS2	TCN2UB	OCR2UB	TCR2UB	**ASSR**

- **Bits 7 al 4 – No están implementados en el registro**

- **Bit 3 – AS2: Habilita la operación asíncrona**

  La puesta en alto de este bit hace que el temporizador 2 sea manejado desde un oscilador externo conectado en TOSC1 y TOSC2. Si se mantiene en bajo, el temporizador es manejado con el reloj $CLK_{I/O}$.

- **Bit 2 – TCN2UB: Bandera para indicar que el registro TCNT2 está ocupado por actualización (*Update Busy*)**

- **Bit 1 – OCR2UB: Bandera para indicar que el registro OCR2 está ocupado por actualización**

- **Bit 0 – TCR2UB: Bandera para indicar que el registro TCCR2 está ocupado por actualización**

Los 3 bits menos significativos son banderas de estado para indicar si un registro está ocupado por actualización, son necesarias cuando se habilita al oscilador externo porque el temporizador 2 trabaja a una frecuencia diferente a la del resto del sistema, generalmente es más baja. Por ello, antes de hacer una lectura o escritura en uno de estos registros (**TCNT2**, **OCR2** y **TCCR2**) se debe verificar si no hay un cambio en proceso. Las banderas se ajustan o limpian de manera automática.

La activación del oscilador externo puede ocasionar que los valores de los registros **TCNT2**, **OCR2** y **TCCR2** se alteren con el cambio de la señal de reloj. Por ello, es recomendable que primero se ajuste al bit **AS2** y que luego se definan los valores correctos para los citados registros.

4.2.8.1 Generación de Formas de Onda con el Temporizador 2

En la tabla 4.13 se muestran los 4 posibles modos de generación de forma de onda, determinados por los bits **WGM2[1:0]**.

Tabla 4.13 Modos de generación de forma de onda

Modo	WGM21	WGM20	Descripción
0	0	0	Normal
1	0	1	PWM con fase correcta
2	1	0	CTC: Limpia al temporizador ante una coincidencia por comparación
3	1	1	PWM rápido

- **Modo 0:** Operación normal del temporizador, sólo se generan eventos de desbordamientos.
- **Modos 1 y 3:** Modos para la generación de PWM, se describen en la sección 4.3.
- **Modo 2:** Limpia al registro del temporizador (coloca 0's en **TCNT2**) tras una coincidencia en la comparación, se genera la bandera **OCF2** debido a que hubo una coincidencia.

4.2.8.2 Respuesta Automática en la Terminal OC2

En el modo normal o en el modo CTC, los bits **COM2[1:0]** definen el comportamiento descrito en la tabla 4.14 para la salida OC2, proporcionando una respuesta automática ante un evento de comparación. La respuesta en los modos de PWM se describe en la sección 4.3.

Tabla 4.14 Respuesta automática en OC2

COM21	COM20	Descripción
0	0	Operación Normal, terminal OC2 desconectada
0	1	Conmuta a OC2, tras una coincidencia por comparación
1	0	Limpia a OC2, tras una coincidencia por comparación
1	1	Pone en alto a OC2, tras una coincidencia por comparación

4.2.8.3 Selección del Reloj para el Temporizador 2

La selección de la señal de reloj está determinada por los bits **CS2[2:0]**, los cuales son descritos en la tabla 4.15. En este caso, las 8 combinaciones son aplicables para el oscilador interno o externo.

Tabla 4.15 Bits para la selección del reloj en el temporizador 2

CS22	CS21	CS20	Descripción
0	0	0	Sin fuente de reloj (temporizador 0 detenido)
0	0	1	clk_{T2S} (sin división)
0	1	0	$clk_{T2S}/8$ (del pre-escalador)
0	1	1	$clk_{T2S}/32$ (del pre-escalador)
1	0	0	$clk_{T2S}/64$ (del pre-escalador)
1	0	1	$clk_{T2S}/128$ (del pre-escalador)
1	1	0	$clk_{T2S}/256$ (del pre-escalador)
1	1	1	$clk_{T2S}/1024$ (del pre-escalador)

El hardware del oscilador externo está optimizado para trabajar a una frecuencia de 32.768 KHz, la cual es adecuada para aplicaciones que involucren un reloj de tiempo real. El periodo que le corresponde es de 30.517578125 µS. Como el temporizador es de 8 bits, si no utiliza al pre-escalador genera un desbordamiento cada 256 x 30.517578125 µS = 7.8125 mS = 1/128 S.

Al emplear al pre-escalador se generan desbordamientos en otras fracciones o múltiplos de segundos reales, en la tabla 4.16 se muestran las frecuencias de operación del temporizador y los periodos de desbordamiento con diferentes factores de pre-escala.

Tabla 4.16 Periodos de desbordamiento en el temporizador 2 con un oscilador externo de 32.768 KHz

Pre-escala	Frecuencia del temporizador 2 (Hz)	Periodo de desbordamiento (S)
1	32 768	1/128
8	4096	1/16
32	1024	1/4
64	512	1/2
128	256	1
256	128	2
1024	32	8

Un reloj de tiempo real con base en un oscilador externo de 32.768 KHz es preciso porque no requiere de instrucciones adicionales para recargar al registro del temporizador, instrucciones que introducirían retrasos de tiempo. Además, puesto que la base de tiempo no dependería del oscilador interno, no sería indispensable realizar su calibración, la cual se mencionó en la sección 2.8.4.

4.2.9 Ejemplos de Uso de los Temporizadores

En esta sección se muestran 3 ejemplos de uso de los temporizadores, estos recursos son muy versátiles, los ejemplos ilustran algunas características representativas de su funcionamiento.

Ejemplo 4.3 Con un ATMega16, suponiendo que está trabajando con el oscilador interno de 1 MHz, genere una señal con una frecuencia de 5 KHz y un ciclo de trabajo del 50 %, utilizando al temporizador 0. Desarrolle las soluciones utilizando:

a) Desbordamiento con sondeo

b) Desbordamiento con interrupciones

c) Coincidencia por comparación con interrupciones, y

d) Coincidencia por comparación con respuesta automática

Por la diversidad de soluciones, sólo se desarrollan en lenguaje C, las soluciones en lenguaje ensamblador deben contar con la misma estructura.

$$\text{Si } f = 5\ KHz \text{ entonces } T = 200\ \mu S.$$

Puesto que el ciclo útil es del 50 %, se tiene $T_{ALTO} = 100\ \mu S$ y $T_{BAJO} = 100\ \mu S$. Al emplear un oscilador de 1 MHz, sin pre-escalador, el temporizador se incrementa cada $1\ \mu S$, por lo tanto, el temporizador debe contar 100 eventos antes de conmutar la salida.

a) Desbordamiento con sondeo: El programa debe configurar al temporizador, esperar la bandera, conmutar la salida y recargar al temporizador, para nuevamente esperar la bandera. Puesto que no hay condiciones para la salida, ésta se genera en PB0, el código correspondiente es:

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

int    main() {

       DDRB = 0xFF;                // Puerto B como salida
       PORTB = 0x00;               // Inicializa la salida

       TCNT0 = -100;               // Para que desborde en el evento 100
       TCCR0 = 0x01;               // Reloj, sin pre-escalador

       while(1) {                  // Lazo infinito
              while( ! ( TIFR & 1 << TOV0 ) )   // Sondea, espera la bandera
                     ;
              TIFR = TIFR | 1 << TOV0;   // Limpia la bandera, le escribe 1
              TCNT0 = -100;              // Recarga al temporizador
              PORTB = PORTB ^ 0x01;      // Conmuta la salida
       }
}
```

Esta solución tiene las siguientes desventajas: Se invierte tiempo de procesamiento en el sondeo y en la limpieza de la bandera de desbordamiento. Como consecuencia, el periodo es incorrecto por las instrucciones agregadas.

Para corregir el periodo deberían contabilizarse los ciclos requeridos por estas actividades y considerarlos en el valor de recarga.

El temporizador se recarga con -100 porque cuenta en orden ascendente, la representación en hexadecimal de -100 es 0x9C, que corresponde con 156. Partiendo de este número, cuando ocurre la transición de 0xFF a 0x00 han transcurrido 100 eventos.

b) Desbordamiento con interrupciones: Ahora en el programa principal se configura al temporizador 0 para que genere una interrupción cuando haya un desbordamiento y en la rutina de atención a la interrupción se conmuta la salida y recarga al temporizador.

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

ISR(TIMER0_OVF_vect) {     // Atiende un evento de desbordamiento
       TCNT0 = -100;              // Recarga al temporizador
       PORTB = PORTB ^ 0x01;      // Conmuta la salida
}

int    main() {

       DDRB = 0xFF;         // Puerto B como salida
       PORTB = 0x00;        // Inicializa la salida

       TCNT0 = -100;        // Para que desborde en el evento 100
       TCCR0 = 0x01;        // Reloj, sin pre-escalador

       TIMSK = 0x01;  // Habilita interrupción por desbordamiento del
       sei();         // temporizador 0 y al habilitador global

       while(1) {           // Lazo infinito, permanece ocioso
           asm("nop");// Se agrega para hacer posible la simulación
       }
}
```

Aunque ya no se invierte tiempo en el sondeo y en la limpieza de la bandera de desbordamiento, el periodo aún es incorrecto por la recarga del temporizador. La bandera de desbordamiento se limpia automáticamente por hardware.

c) Coincidencia por comparación con interrupciones: Para esta solución se carga el registro de comparación (**OCR0**) con 99, la cuenta inicia con 0, de 0 a 99 son 100 eventos. Se configura al recurso para que cuando ocurra una coincidencia se reinicie al temporizador (modo CTC) y se generé una interrupción en cuya ISR se conmute la salida.

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

ISR(TIMER0_COMP_vect) { // Atiende evento de coincidencia por comparación
       PORTB = PORTB ^ 0x01;      // Conmuta la salida
}

int    main() {

       DDRB = 0xFF;         // Puerto B como salida
       PORTB = 0x00;        // Inicializa la salida

       OCR0  = 99;          // de 0 a 99 son 100 eventos
       TCCR0 = 0x09;        // Reloj sin pre-escalador y activa modo CTC

       TIMSK = 0x02;        // Habilita interrupción de coincidencia por compara-
       sei();               // ción en el temporizador 0 y al habilitador global

       while(1) {           // Lazo infinito, permanece ocioso
           asm("nop");      // Sólo para simulación
       }
}
```

Esta solución es eficiente, la señal se genera con la frecuencia solicitada porque no es necesario recargar al temporizador, éste automáticamente es reiniciado en 0 cuando su valor coincide con el del registro **OCR0**, sólo es necesario considerar la conmutación de la salida en la ISR.

d) Coincidencia por comparación con respuesta automática: En esta solución la salida es generada por hardware, por lo debe ser en la terminal OC0 (PB3), el código es:

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

int    main() {

       DDRB = 0xFF;         // Puerto B como salida
       PORTB = 0x00;        // Inicializa la salida

       OCR0  = 99;          // de 0 a 99 son 100 eventos
       TCCR0 = 0x19;        // Reloj sin pre-escalador, activa modo CTC
                            // y habilita conmutación automática
       while(1) {           // Lazo infinito, permanece ocioso
           asm("nop");      // Sólo para la simulación
       }
}
```

Ésta es la mejor solución porque hace uso de los recursos de hardware disponibles en el MCU.

Ejemplo 4.4 Empleando un ATMega16 genere una señal con una frecuencia de 200 Hz y un ciclo de trabajo del 50 %, suponga que está trabajando con el oscilador interno de 1 MHz, muestre la solución más simple en Ensamblador y en lenguaje C.

Si $f = 200\ Hz$ entonces $T = 5\ mS$ esto significa: $T_{ALTO} = 2500\ \mu S$ y $T_{BAJO} = 2500\ \mu S$.

Un conteo de 2500 eventos no es posible con un temporizador de 8 bits, a menos que se utilice al pre-escalador. Por simplicidad, se emplea al temporizador 1 (16 bits) sin pre-escalador y se configura para una limpieza ante una coincidencia por comparación (modo CTC) y con respuesta automática en OC1A (PD5).

El código en lenguaje C es el siguiente:

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

int    main() {

       DDRD = 0xFF;          // Puerto D como salida
       PORTD = 0x00;         // Inicializa la salida

       OCR1A  = 2499;        // para 2500 eventos

       TCCR1A = 0x40;        // Reloj sin pre-escalador, activa modo CTC
       TCCR1B = 0x09;        // y habilita conmutación automática de OC1A

       while(1) {            // Lazo infinito, permanece ocioso
           asm("nop");       // Sólo para la simulación
       }
}
```

El programa en ensamblador es:

```
       .include      "m16def.inc"

       LDI    R16, 0xFF           ; Puerto D como salida
       OUT    DDRD, R16
       CLR    R16                 ; Inicializa la salida
       OUT    PORTD, R16

       LDI    R16, LOW(2499)      ; Carga registro de comparación
       LDI    R17, HIGH(2499)     ; para 2500 eventos

       OUT    OCR1AH, R17
       OUT    OCR1AL, R16

       LDI    R16, 0x40           ; Reloj sin pre-escalador, activa modo CTC
       OUT    TCCR1A, R16         ; y habilita conmutación automática de OC1A
       LDI    R16, 0x09
       OUT    TCCR1B, R16
aqui:  RJMP   aqui                ; Lazo infinito, permanece ocioso
```

Con base en los últimos 2 ejercicios se observa que, la generación de una señal a determinada frecuencia, empleando respuesta automática, básicamente se reduce a configurar el recurso. Para modificar la frecuencia sólo hay que modificar el valor del registro de comparación. Si se requiriera generar las 2 señales en forma simultánea, basta con juntar el código de los 2 ejemplos anteriores en un programa, como las instrucciones hacen referencia a recursos diferentes, no interfiere la configuración del temporizador 0 con la configuración del temporizador 1.

En el siguiente ejemplo se combina el uso de una interrupción externa y 2 de los temporizadores. Un aspecto interesante es que los temporizadores no trabajan en todo momento, sino que el inicio de su operación lo determina la interrupción externa.

Ejemplo 4.5 Configure al ATMega8 para que funcione como un timbre, de manera que genere un tono de 400 Hz durante 3 segundos cuando se presione un botón. En la figura 4.13 se muestra el acondicionamiento del hardware con la salida esperada.

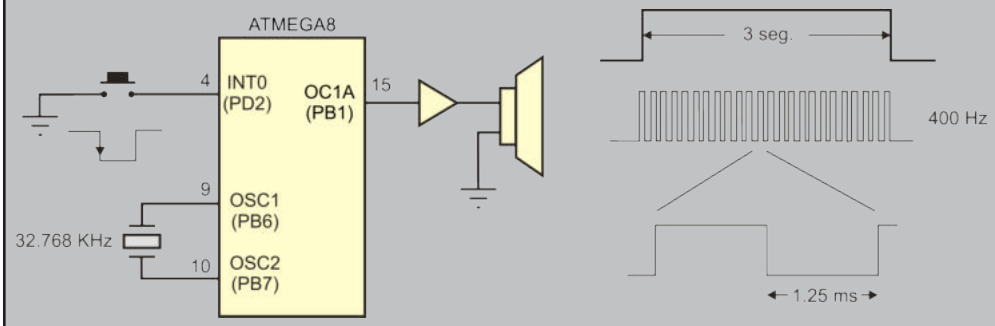


Figura 4.13 Hardware para el desarrollo del timbre

Cuando se manejan interrupciones de diferentes recursos, en ocasiones no es posible un planteamiento del software con un diagrama de flujo porque en el programa principal básicamente se realiza la configuración de los recursos y la funcionalidad del sistema queda determinada por las ISRs.

En esos casos, lo aconsejable es analizar la funcionalidad del sistema en su conjunto, determinando la tarea de cada recurso y definiendo cuándo debe ser activado o desactivado. En este ejemplo se utilizan 3 recursos:

1. La **interrupción externa 0** detecta si se ha sido presionado el botón, el recurso está activo desde que el programa inicia y se desactiva en su ISR, para evitar otras interrupciones externas mientras transcurre el intervalo de 3 segundos. El recurso es reactivado nuevamente en la ISR del temporizador 2, cuando terminan los 3 segundos.

2. El **temporizador 2** se emplea para el conteo de segundos, se configura para que trabaje con el oscilador externo y desborde cada segundo, generando una interrupción. Se activa en la ISR de la interrupción externa y se desactiva en su misma ISR, una vez que han concluido los 3 segundos.

3. El **temporizador 1** se emplea para generar el tono de 400 Hz, se configura para que trabaje con respuesta automática. Se activa en la ISR de la interrupción externa y se desactiva en la ISR del temporizador 2, una vez que ha finalizado el intervalo de 3 segundos.

El contador de segundos se inicializa en la ISR de la interrupción externa y se modifica en la ISR del temporizador 2, por lo tanto, debe manejarse con una variable global.

Con este análisis, se procede con la elaboración del programa, sólo se muestra la versión en lenguaje C.

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

volatile  unsigned char  segundos;

ISR(INT0_vect) {           // Atiende a la interrupción externa 0

       GICR = 0x00;               // Inhabilita la INT0
       // Complementa la configuración del temporizador 1
       TCNT1 = 0x0000;            // Garantiza al temporizador 1 en 0
       TCCR1B = 0x09;             // Activa al oscilador y al modo CTC
       // Complementa la configuración del temporizador 2
       segundos = 0;              // La cuenta de segundos inicia en 0
       TCNT2 = 0x00;              // Garantiza al temporizador 2 en 0
       TCCR2 = 0x05;              // Para que desborde cada segundo
       TIMSK = 0x40;              // Interrupción por desbordamiento
}

ISR(TIMER2_OVF_vect) {   // Interrupción por desbordamiento del timer 2

       segundos++;                // Ha pasado un segundo
       if( segundos == 3 ) {
              GICR = 0x40;        // Habilita la INT0
              TCCR1B = 0x00;// Detiene al temporizador 1 (quita el tono)
              TCCR2 = 0x00;       // Detiene al temporizador 2
              TIMSK = 0x00; //Desactiva su interrupción por desbordamiento
       }
}

int    main() {

       // Configuración de entradas y salidas
       DDRD = 0x00;               // Puerto D como entrada
       PORTD = 0xFF;              // Resistor de Pull-Up
       DDRB = 0xFF;               // Puerto B como salida

       // Configuración de la interrupción externa 0
       MCUCR = 0x02;              // Configura INT0 por flanco de bajada
       GICR = 0x40;               // Habilita la INT0

       // Configuración parcial del Temporizador 1
```

```
        OCR1A = 1249;               // Para 1.25 mS
        TCCR1A = 0x40;              // Configura para respuesta automática

        // Configuración parcial del temporizador 2
        ASSR = 0x08;                // Se usa al oscilador externo
        sei();                      // Habilitador global

        while(1) {                  // Lazo infinito, permanece ocioso
            asm("nop");             // Sólo para simulación
        }
}
```

4.3 Modulación por Ancho de Pulso (PWM)

La modulación por ancho de pulso es una técnica para generar "señales analógicas" en alguna salida de un sistema digital. Puede usarse para controlar la velocidad de un motor, la intensidad luminosa de una lámpara, etc. La base de PWM es la variación del ciclo de trabajo (*duty cycle*) de una señal cuadrada, en la figura 4.14 se muestra un periodo de una señal cuadrada con la definición del ciclo útil.

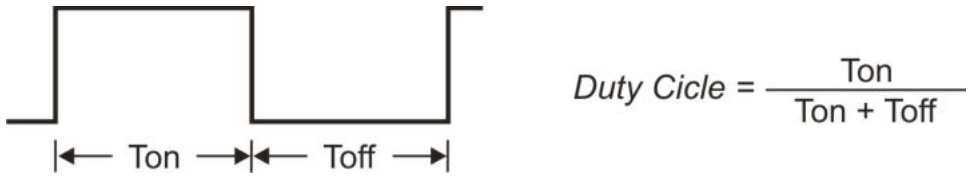


Figura 4.14 Definición del ciclo de trabajo

Al cambiar el ciclo de trabajo se modifica el voltaje promedio (V_{AVG}), el cual se obtiene con la ecuación:

$$V_{AVG} = \frac{1}{T}\int_0^T Vp\, dt = Vp\,\frac{Ton}{T}$$

Para un ciclo de trabajo del 50% el tiempo en alto es $T_{ON} = T/2$ y por lo tanto $V_{AVG} = Vp/2$.

En la mayoría de aplicaciones se conecta directamente la salida de PWM con el dispositivo a controlar, sin embargo, al conectar un filtro pasivo RC pasa bajas, como el mostrado en la figura 4.15, se genera una señal analógica.

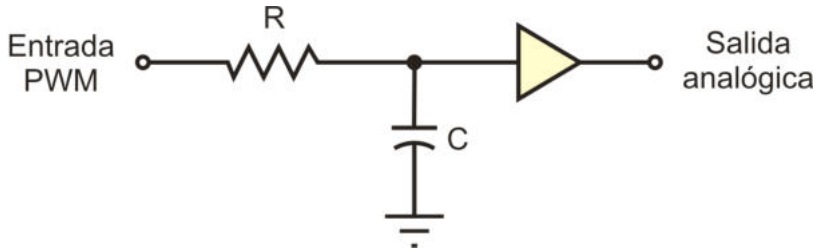


Figura 4.15 Filtro pasa bajas para generar una señal analógica

4.3.1 Generación de PWM con los Microcontroladores AVR

Los microcontroladores AVR pueden generar señales PWM en 3 modos diferentes:

- PWM rápido (*fast PWM*).
- PWM con fase correcta (*phase correct PWM*).
- PWM con fase y frecuencia correcta (*phase and frequency correct PWM*).

Las señales PWM se generan al configurar adecuadamente los temporizadores. Las salidas PWM quedan disponibles en las terminales OCx, en las figuras 4.10, 4.11 y 4.12 puede verse que estas salidas provienen de los módulos generadores de formas de onda en los diferentes temporizadores.

En la tabla 4.17 se indican los modos de PWM que se pueden generar con cada temporizador. Los diferentes modos se describen en las siguientes secciones.

Tabla 4.17 Modos de PWM para los diferentes temporizadores

AVR	Temporizador	PWM rápido	PWM con fase correcta	PWM con fase y frecuencia correcta
ATMega8	0			
	1	X	X	X
	2	X	X	
ATMega16	0	X	X	
	1	X	X	X
	2	X	X	

4.3.2 PWM Rápido

Es un modo para generar una señal PWM a una frecuencia alta, en este modo el temporizador cuenta de 0 a su valor máximo (MAX) y se reinicia, el conteo se realiza continuamente, de manera que, en algún instante de tiempo, el temporizador coincide con el contenido del registro de comparación (**OCRx**).

La señal PWM se genera de la siguiente manera: La salida OCx es puesta en alto cuando el registro del temporizador realiza una transición de MAX a 0 y es puesta en bajo cuando ocurre una coincidencia por comparación. Este modo se conoce como **No Invertido**. En el modo **Invertido** ocurre lo contrario para OCx, ambos modos se muestran en la figura 4.16, en donde puede notarse que el ancho del pulso está determinado por el valor del registro **OCRx**.

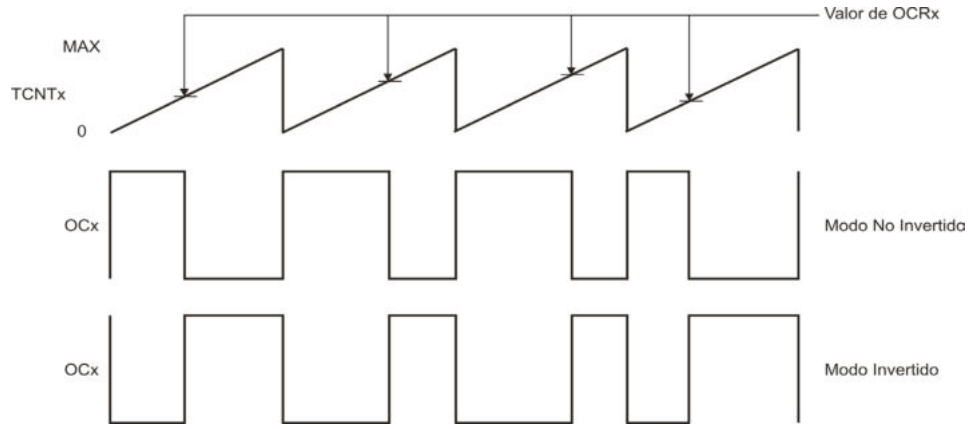


Figura 4.16 Señales de PWM rápido

Por el comportamiento del temporizador, al modo PWM rápido también se le conoce como un modo de pendiente única.

Puesto que el temporizador cuenta de 0 a MAX, la frecuencia de la señal de salida está dada por:

$$f_{PWM} = \frac{f_{clk}}{MAX + 1}$$

La frecuencia de salida se puede cambiar con el pre-escalador, con éste se modificaría el valor de f_{clk}.

Un problema se podría generar si se pretende escribir en el registro **OCRx** un valor menor al que en ese instante contiene el temporizador, esto se reflejaría en una variación de la frecuencia de la señal de salida. Para evitarlo, el acceso al registro **OCRx** se realiza por medio de un buffer doble. Cualquier instrucción que intente escribir en **OCRx** lo hace en un buffer intermedio, la escritura real en el registro **OCRx** se realiza en el momento que el temporizador pasa de MAX a 0, con ello, las señales siempre se generan con el mismo periodo.

4.3.3 PWM con Fase Correcta

En este modo, el temporizador cuenta en forma ascendente (de 0 a MAX), para después contar en forma descendente (de MAX a 0). La modificación de la terminal de salida OCx se realiza en las coincidencias por comparación con el registro **OCRx**.

En el modo **No Invertido** la salida OCx se pone en bajo tras una coincidencia mientras el temporizador se incrementa y en alto cuando ocurra la coincidencia durante el decremento. En el modo **Invertido** ocurre lo contrario para la salida OCx. El comportamiento de las señales de salida en este modo de PWM se muestra en la figura 4.17, en donde se observa que, para el modo no invertido, el pulso está centrado con el valor cero del temporizador (están en fase).

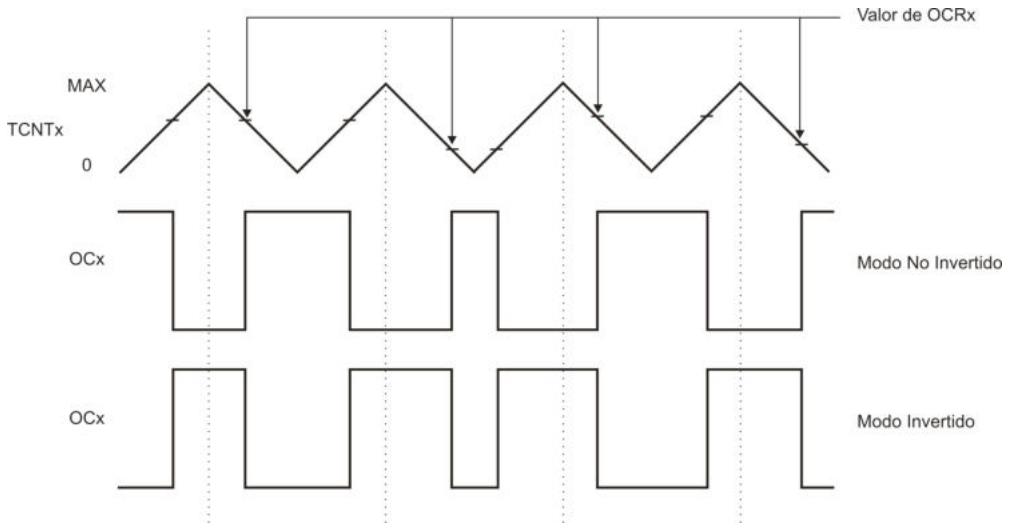


Figura 4.17 Señales de PWM con fase correcta

Por el comportamiento del temporizador, a este modo de PWM también se le conoce como un modo de pendiente doble.

La frecuencia de la señal generada está dada por:

$$f_{PWM} = \frac{f_{clk}}{2\,(MAX + 1)}$$

La cual también puede modificarse con el pre-escalador.

En este modo también se cuenta con un buffer doble para la escritura en el registro **OCRx**, las escrituras reales se realizan cuando el temporizador alcanza su valor máximo.

Cuando este modo es manejado por el temporizador 1 es posible modificar el valor del máximo durante la generación de las señales, con ello se cambia la frecuencia de la señal PWM, como se muestra en la figura 4.18. No obstante, la modificación del máximo en tiempo de ejecución da lugar a la generación de señales que no son simétricas.

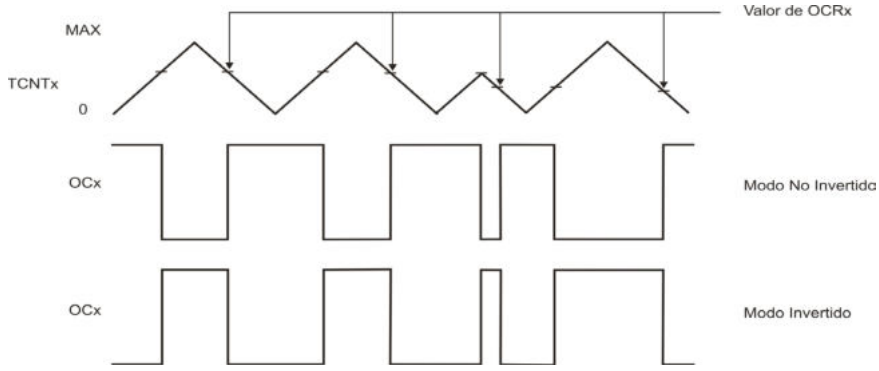


Figura 4.18 Señales de PWM con fase correcta y máximo variable

4.3.4 PWM con Fase y Frecuencia Correcta

Este modo es muy similar al modo de fase correcta, ambos modos son de pendiente doble, es decir, el temporizador cuenta de manera ascendente, alcanza su valor máximo, y luego cuenta en forma descendente.

Difieren en el momento de actualizar al registro **OCRx**, el modo de fase correcta actualiza al registro **OCRx** cuando **TCNTx** llega a su valor máximo, y el modo de fase

y frecuencia correcta lo hace cuando el registro **TCNTx** llega a cero. En la figura 4.19 se observan las señales generadas en el modo de PWM con fase y frecuencia correcta.

Si se utiliza un máximo constante, se tiene el mismo efecto al emplear uno u otro modo, pero si el máximo es variable, con el modo de fase correcta se pueden generar formas asimétricas, lo cual no ocurre con el modo de frecuencia y fase correcta. En la figura 4.19 puede notarse que la señal de salida es simétrica aun después de modificar el valor máximo del temporizador.

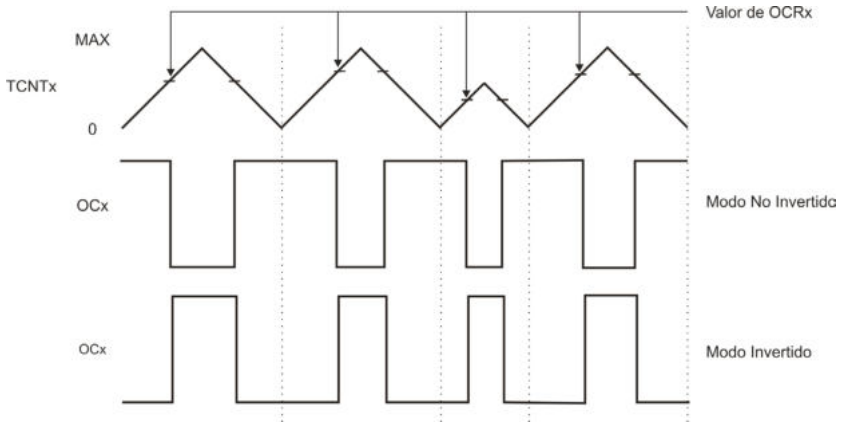


Figura 4.19 Señales de PWM con frecuencia y fase correcta

Este modo sólo puede ser generado por el temporizador 1, que es de 16 bits. Y únicamente está disponible con máximos variables, los cuales se definen en los registros **OCR1A** o **ICR1**.

4.3.5 El Temporizador 0 y la Generación de PWM

En un ATMega8 el temporizador 0 no puede generar señales PWM. En un ATMega16, con el temporizador 0 es posible generar los modos de PWM rápido y PWM con fase correcta.

El registro de control del temporizador 0 es el **TCCR0**, cuyos bits son:

	7	6	5	4	3	2	1	0	
0x33	FOC0	WGM00	COM01	COM00	WGM01	CS02	CS01	CS00	**TCCR0**

La función de estos bits fue descrita en la sección 4.2.6. Los bits **WGM0[1:0]** se listaron en la tabla 4.7, con ellos se definen los modos de generación de forma de onda, los 2 que determinan la generación de señales PWM se muestran en la tabla 4.18.

Tabla 4.18 Modos para la generación de PWM con el temporizador 0

Modo	WGM01	WGM00	Descripción
1	0	1	PWM con fase correcta
3	1	1	PWM rápido

Bajo ambos modos de PWM, los bits **COM0[1:0]** determinan el comportamiento de la salida OC0, este comportamiento es descrito en la tabla 4.19.

Tabla 4.19 Comportamiento de la salida OC0 en los modos de PWM

COM01	COM00	Descripción
0	0	Operación Normal, terminal OC0 desconectada
0	1	Reservado (sin uso)
1	0	Modo no invertido
1	1	Modo invertido

El temporizador 0 es de 8 bits, por ello, el máximo valor del temporizador es de 255.

4.3.6 El Temporizador 1 y la Generación de PWM

Para el ATMega8 y ATMega16, con el temporizador 1 se pueden generar 2 señales en cualquiera de los 3 modos de PWM descritos con anterioridad: PWM rápido, PWM con fase correcta y PWM con frecuencia y fase correcta. Las 2 señales se pueden generar en forma simultánea en las terminales OC1A y OC1B, porque el temporizador 1 cuenta con 2 registros para comparación: **OCR1A** y **OCR1B**. Con estos registros se determina el ancho de pulso en cada señal, aunque las señales van a tener la misma frecuencia.

Los registros de control del temporizador 1 son el **TCCR1A** y **TCCR1B**, los bits de estos registros son:

	7	6	5	4	3	2	1	0	
0x2F	COM1A1	COM1A0	COM1B1	COM1B0	FOC1A	FOC1B	WGM11	WGM10	**TCCR1A**
0x2E	ICNC1	ICES1	-	WGM13	WGM12	CS12	CS11	CS10	**TCCR1B**

La función de estos bits fue descrita en la sección 4.2.7. Los bits **WGM1[3:0]** definen los modos de generación de forma de onda, en la tabla 4.10 se listaron aquellos modos que no están relacionados con PWM, los modos que determinan la generación de señales PWM se muestran en la tabla 4.20.

Tabla 4.20 Modos para la generación de PWM con el temporizador 1

Modo	WGM13	WGM12	WGM11	WGM10	Descripción	Máximo
1	0	0	0	1	PWM con fase correcta de 8 bits	0x00FF
2	0	0	1	0	PWM con fase correcta de 9 bits	0x01FF
3	0	0	1	1	PWM con fase correcta de 10 bits	0x03FF
5	0	1	0	1	PWM rápido de 8 bits	0x00FF
6	0	1	1	0	PWM rápido de 9 bits	0x01FF
7	0	1	1	1	PWM rápido de 10 bits	0x03FF
8	1	0	0	0	PWM con frecuencia y fase correcta	ICR1
9	1	0	0	1	PWM con frecuencia y fase correcta	OCR1A
10	1	0	1	0	PWM con fase correcta	ICR1
11	1	0	1	1	PWM con fase correcta	OCR1A
14	1	1	1	0	PWM rápido	ICR1
15	1	1	1	1	PWM rápido	OCR1A

En la tabla 4.20 se observa el uso de 3 máximos fijos: 0x00FF, 0x01FF y 0x03FF, para que el temporizador trabaje con 8, 9 ó 10 bits, respectivamente. Los máximos fijos son utilizados en los modos de PWM rápido y PWM con fase correcta. No hay máximos fijos para el modo de PWM con frecuencia y fase correcta, ya que su comportamiento sería el mismo que el modo de fase correcta.

En los 3 modos de PWM se pueden emplear máximos variables, proporcionados por los registros **ICR1** u **OCR1A**. Sin embargo, si se destina a **OCR1A** para definir al máximo, únicamente se puede generar una señal PWM en la salida OC1B, utilizando a **OCR1B** como registro para modular el ancho del pulso.

Si alguna aplicación requiere la generación de 2 señales PWM a una frecuencia que no coincida con las proporcionadas por los valores fijos, aun considerando las variantes que proporciona el pre-escalador, debe emplearse al registro **ICR1** para definir el valor máximo del temporizador, anulando con ello la posibilidad de emplear los recursos de captura.

Los bits **COM1A[1:0]** y **COM1B[1:0]** determinan el comportamiento de las salidas OC1A y OC1B, respectivamente. Este comportamiento es descrito en la tabla 4.21.

Tabla 4.21 Comportamiento de la salidas OC1A/OC1B en los modos de PWM

COM1A1/ COM1B1	COM1A0/ COM1B0	Descripción
0	0	Operación Normal, terminal OC1A/OC1B desconectada
0	1	Reservado (sin uso)
1	0	Modo no invertido
1	1	Modo invertido

4.3.7 El Temporizador 2 y la Generación de PWM

Con el temporizador 2 es posible generar los modos de PWM rápido y PWM con fase correcta (ATMega8 y ATMega16). El registro de control del temporizador 2 es el **TCCR2**, cuyos bits son:

	7	6	5	4	3	2	1	0	
0x25	FOC2	WGM20	COM21	COM20	WGM21	CS22	CS21	CS20	**TCCR2**

La función de estos bits fue descrita en la sección 4.2.8. Los bits **WGM2[1:0]** se listaron en la tabla 4.13, con ellos se definen los modos de generación de forma de onda, los 2 que determinan la generación de señales PWM se muestran en la tabla 4.22.

Tabla 4.22 Modos para la generación de PWM con el temporizador 2

Modo	WGM21	WGM20	Descripción
1	0	1	PWM con fase correcta
3	1	1	PWM rápido

Bajo ambos modos de PWM, los bits **COM2[1:0]** determinan el comportamiento de la salida OC2, este comportamiento es descrito en la tabla 4.23.

Tabla 4.23 Comportamiento de la salida OC2 en los modos de PWM

COM21	COM20	Descripción
0	0	Operación Normal, terminal OC2 desconectada
0	1	Reservado (sin uso)
1	0	Modo no invertido
1	1	Modo invertido

El temporizador 2 es de 8 bits, por ello, el máximo valor del temporizador es de 255.

4.3.8 Ejemplos de Uso de las Señales PWM

En esta sección se muestran 2 ejemplos para ilustrar el uso de la generación de señales moduladas en ancho de pulso.

Ejemplo 4.6 Empleando un ATMega8, generar una señal PWM en modo no invertido, con el ancho de pulso determinado por el valor del puerto D.

Puesto que no se han establecido otras restricciones, se utiliza al temporizador 2 (de 8 bits) empleando el modo de PWM rápido, la salida queda disponible en la terminal OC2 (PB3). La solución en lenguaje C es:

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

int  main() {

       DDRB = 0xFF;          //  Puerto B como salida

       PORTD = 0xFF;         // Resistores de Pull-Up en el puerto D
                             // el Puerto D por default es entrada

       TCCR2 = 0x69;         // PWM rápido, en modo No Invertido y sin
                             // pre-escalador

       while(1) {
              OCR2 = PIND;  // El ancho de pulso lo determina el puerto D
       }
}
```

Como no se emplea al pre-escalador, suponiendo que el microcontrolador va a trabajar con su oscilador interno de 1 MHz, la frecuencia de la señal PWM generada es de:

$$f_{PWM} = \frac{1\,MHz}{256} = 3.90\,KHz$$

La solución en ensamblador sigue la misma estructura, el código es:

```
       .include <m8def.inc>

       LDI    R16, 0xFF
       OUT    DDRB, R16     ; Puerto B como salida

       OUT    PORTD, R16    ; Resistores de Pull-Up en el puerto D
                            ; El Puerto D por default es entrada

       LDI    R16, 0x69;PWM rápido, en modo No Invertido y sin pre-escalador
       OUT    TCCR2, R16
loop:
       IN     R16, PIND     ; El ancho de pulso lo determina el puerto D
       OUT    OCR2, R16
       RJMP   loop
```

Ejemplo 4.7 Para el manejo de un servomotor se requiere de una señal PWM con un periodo de 20 mS. Con este periodo, el servomotor se mantiene en su extremo izquierdo (0°) si el tiempo en alto es de 0.9 mS, en su posición central (90°) con un tiempo de 1.5 mS y en su extremo derecho (180°) con 2.1 mS. En la figura 4.20 se muestran las señales requeridas por el servomotor.

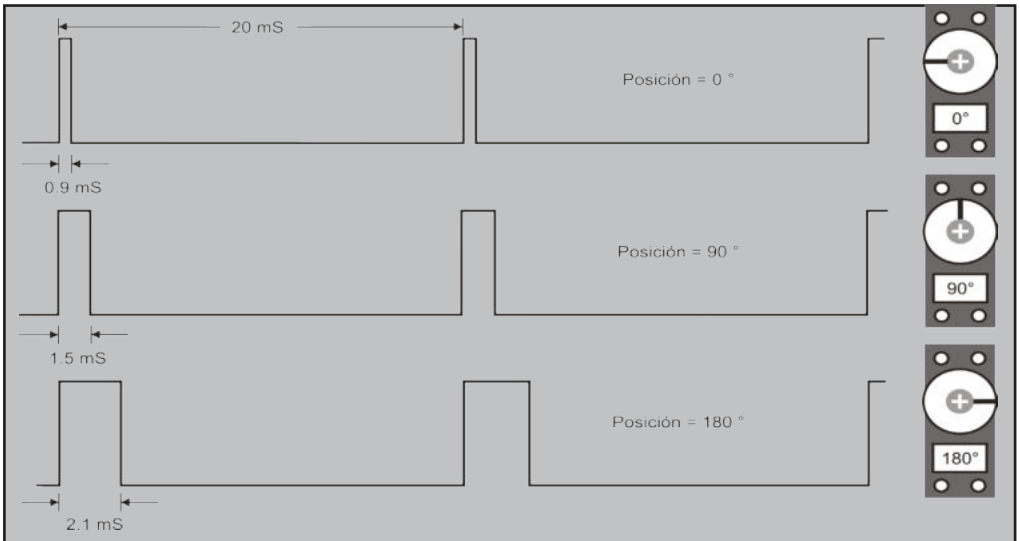


Figura 4.20 Señales para manejar un servomotor

Muestre la configuración del temporizador 1 para generar esta señal e indique los valores de los registros de comparación para la posición central y para cada uno de los extremos.

Se asume que el microcontrolador está operando con el oscilador interno de 1 MHz, por lo tanto, si no se utiliza al pre-escalador, el temporizador se incrementa cada 1 µS.

Para un periodo de 20 mS, utilizando PWM rápido, el temporizador 1 debe contar de 0 a 19 999, rango alcanzable con 16 bits.

La configuración requerida para generar la señal en OC1A es:

```
ICR1 = 19999;          // Valor máximo para el temporizador 1
TCCR1A = 0x82;         // COM1A[1:0] = "10" Modo no invertido,
                       // salida en OC1A
TCCR1B = 0x19;         // WGM1[3:0] ="1110",PWM rápido con máximo en ICR1
                       // Señal de reloj sin pre-escalador
```

Para que el servomotor esté en su extremo izquierdo (0 °): **OCR1A** = 899;

Para que el servomotor esté en su posición neutral (90 °): **OCR1A** = 1499;

Para que el servomotor esté en su extremo derecho (180 °): **OCR1A**= 2099;

Entre ambos extremos se tienen 1200 combinaciones posibles (2099 – 899), dado que con estos valores se consiguen posiciones intermedias, el servomotor puede manejarse con una resolución de 180 °/1200 = 0.15 °.

4.4 Ejercicios

Los problemas que a continuación se presentan están relacionados con los recursos que se han descrito en el presente capítulo, en algunos es necesario el uso de más de un recurso, haciendo necesario un análisis para determinar la función de cada recurso. Pueden resolverse con un ATMega8 o un ATMega16, empleando lenguaje ensamblador o lenguaje C.

1. Realice un sistema que simultáneamente genere 2 señales, la primera a 10 KHz y la segunda a 500 Hz, suponga que el microcontrolador va a trabajar con el oscilador interno de 1 MHz.

2. Con base en la figura 4.21, desarrolle el programa que controle el encendido de un horno para mantenerlo alrededor de una temperatura de referencia. El horno está acondicionado para generar 2 señales (*hot* y *cold*), debe encenderse si la temperatura está por debajo del nivel *cold* y apagarse si está por encima del nivel *hot*.

3. Empleando los recursos de captura del temporizador 1, desarrolle un programa que reciba y decodifique una secuencia serial de 8 bits de información modulada por el ancho de pulsos activos en bajo, como se muestra en la figura 4.22 (señales similares son manejadas por controles remoto comerciales). Después de detectar al bit de inicio, deben obtenerse los 8 bits de datos iniciando con el bit menos significativo, concluida la recepción, debe mostrarse el dato en cualquiera de los puertos libres (*sugerencia:* utilice 2 mS y 1 mS como referencias, si el ancho del pulso es mayor a 2 mS es un bit de inicio, menor a 2 mS pero mayor a 1 mS es un 1 lógico, menor a 1 mS, se trata de un 0 lógico).

4. En un supermercado se ha determinado premiar a cada cliente múltiplo de 200. Desarrolle un sistema basado en un AVR, el cual debe detectar al cliente número 200 y generar un tono de 440 Hz (aproximadamente), por 5 segundos, cuando eso ocurra. Los clientes deben presionar un botón para ser considerados. Se trata de una aplicación para los temporizadores, utilice al temporizador 0, manejado con eventos externos, para llevar el conteo de clientes. Al temporizador 1 para generar el tono de 440 Hz y al temporizador 2 para el conteo de segundos, empleando un oscilador externo de 32.768 KHz.

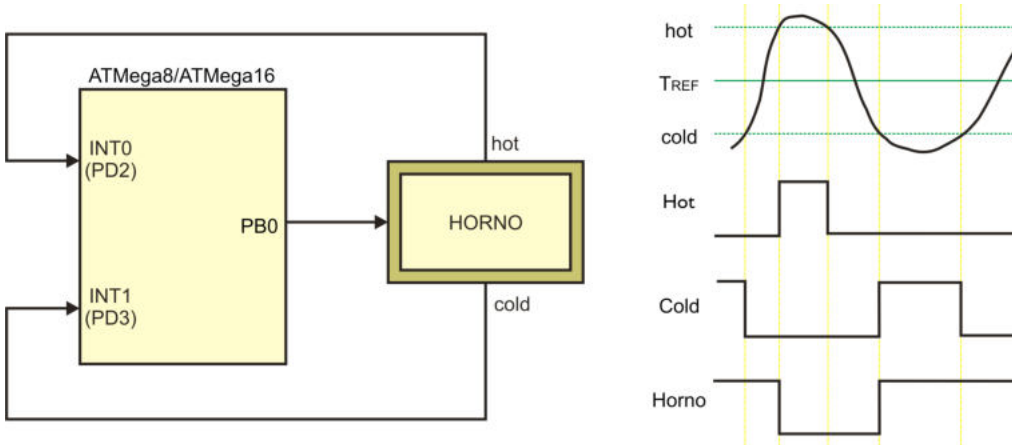


Figura 4.21 Control de un horno

5. Empleando una señal PWM, controle el encendido de un conjunto de LEDs ultra-brillantes, manejando 5 niveles de intensidad. El sistema debe contar con un botón para el cambio de intensidad, cada vez que el botón es presionado, la intensidad se debe incrementar el 20 % del valor máximo posible. Los LEDs deben estar apagados cuando se el sistema se enciende (0 % en el ancho de pulso). Si se alcanza la máxima intensidad (100 %) y se presiona nuevamente al botón, los LEDs deben apagarse.

   Configure para que la señal de salida tenga un periodo en el orden de décimas de milisegundos, para una operación adecuada de los LEDs.

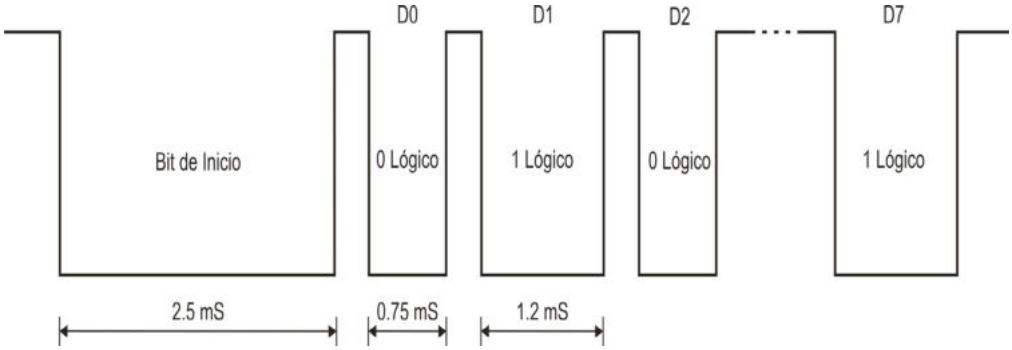


Figura 4.22 Señal modulada, la información está en los niveles bajos de voltaje

5. Recursos para el Manejo de Información Analógica

Aunque los microcontroladores AVR son elementos para el procesamiento de información digital, incluyen 2 recursos que les permiten obtener información generada por dispositivos analógicos. Se trata de un convertidor analógico a digital (ADC, *analog-to-digital converter*) y de un comparador analógico (AC, *analog comparator*), estos recursos se describen en el presente capítulo.

5.1 Convertidor Analógico a Digital

Un ADC recibe una muestra de una señal analógica, es decir, su valor en un instante de tiempo, a partir de la cual genera un número (valor digital). La función de un ADC es contraria a la que realiza un convertidor digital a analógico (DAC, *digital-to-analog converter*), un DAC genera un nivel de voltaje analógico a partir de un número o valor digital. En la figura 5.1 se ilustra la función de un ADC y un DAC, los microcontroladores AVR no incluyen un DAC, pero se revisa su funcionamiento porque el ADC embebido utiliza un DAC para poder operar.

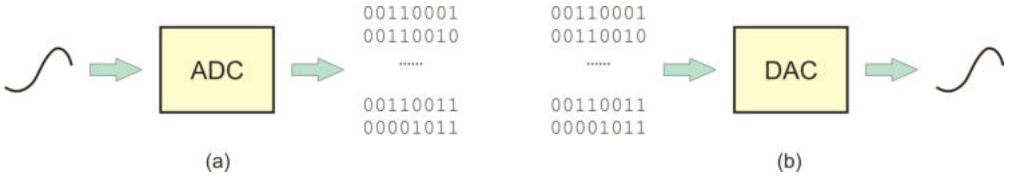


Figura 5.1 Funcionalidad de (a) un Convertidor Analógico a Digital (ADC), y de (b) un Convertidor Digital a Analógico (DAC)

5.1.1 Proceso de Conversión Analógico a Digital

Una señal analógica toma valores continuos a lo largo del tiempo, el proceso para convertirla a digital involucra 2 etapas: el muestreo y la cuantificación.

El muestreo básicamente consiste en tomar el valor de la señal analógica en un instante de tiempo, y mantenerlo, mientras el proceso concluye. Cada muestra se va tomando cuando ocurre un intervalo de tiempo predefinido, conocido como periodo de muestreo, cuyo inverso es la *frecuencia de muestreo.*

Para que la información contenida en la señal a digitalizar sea recuperada de manera correcta, se requiere que la frecuencia de muestreo sea por lo menos el doble de la frecuencia de la señal analógica. Por ejemplo, si se va a digitalizar una señal de audio acotada a 2 KHz, la frecuencia de muestreo por lo menos debe ser de 4 KHz.

La cuantificación consiste en la asociación de cada muestra con un número o valor digital, que pertenece a un conjunto finito de valores, el número de bits utilizado por el convertidor determina el total de valores. Cada muestra analógica se asocia con el valor digital más cercano. Por ejemplo, si el convertidor es de 8 bits, el conjunto tiene 2^8 = 256 valores diferentes (0 a 255).

El número de bits y el voltaje máximo a convertir (V_{MAX}) determinan la *resolución* del ADC. La resolución se define como el cambio requerido en la señal analógica para que la señal digital se incremente en un bit. Por ejemplo, para un ADC de 8 bits capaz de recibir un voltaje máximo de 5 V, su resolución es de:

$$Resolución = \frac{V_{MAX}}{2^8 - 1} = \frac{5\,V}{255} = 19.60\,mV$$

En la figura 5.2 se muestran las etapas involucradas en el proceso de conversión analógica a digital, se toman 10 muestras de una señal analógica y se utilizan 3 bits para su cuantificación.

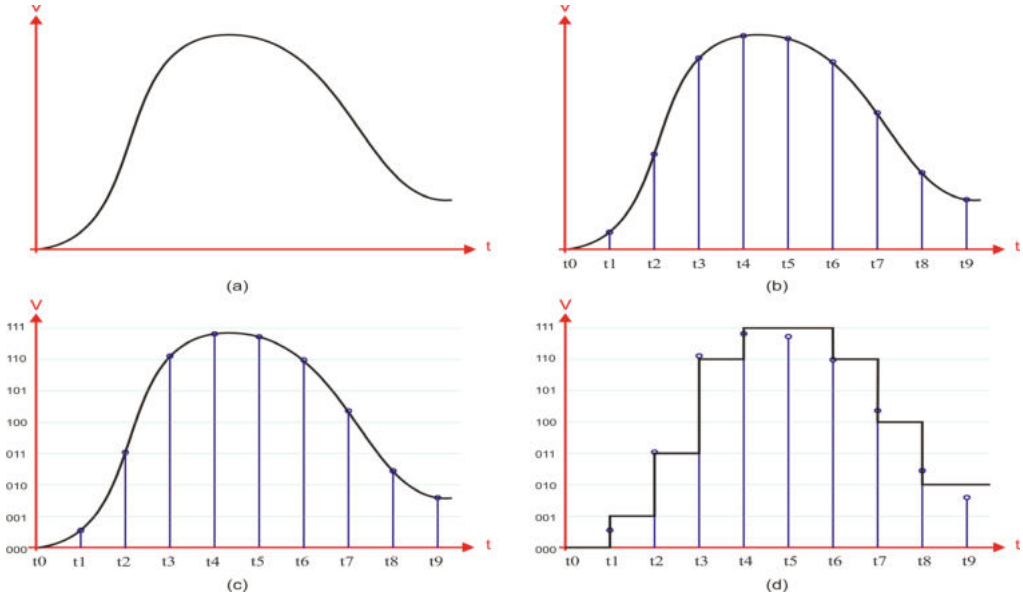


Figura 5.2 Proceso de conversión: (a) una señal analógica, (b) muestreo, (c) cuantificación y (d) señal digital

5.1.2 Hardware para la Conversión Digital a Analógico

El hardware para convertir una señal digital en su correspondiente valor analógico involucra una red resistiva R-2R, como se muestra en la figura 5.3. Por su estructura, la corriente se va dividiendo a la mitad en cada malla de la red. Por medio de interruptores digitales se define qué mallas contribuyen en la corriente de salida, los interruptores son controlados con el dato digital a convertir. Las corrientes se suman y se convierten a voltaje con la ayuda de un amplificador.

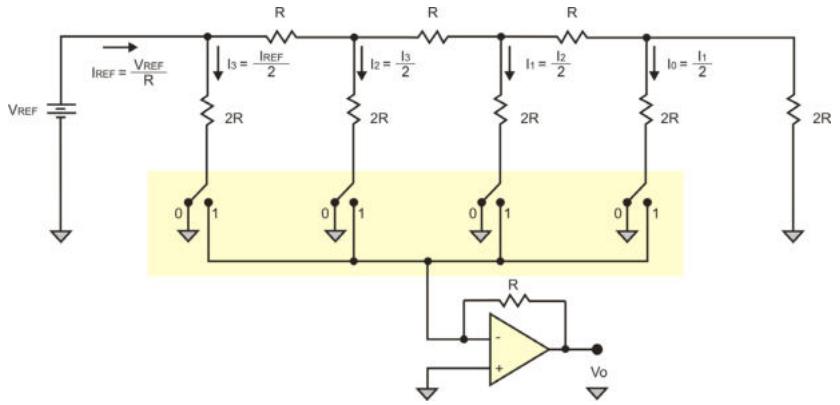


Figura 5.3 Red resistiva R-2R para una conversión digital a analógica

5.1.3 Hardware para la Conversión Analógico a Digital

Existen 3 configuraciones clásicas para estos convertidores, las cuales se caracterizan por su velocidad de conversión. Un ADC del tipo *integrador* es el más simple, en lo que a hardware se refiere, pero es demasiado lento, su tiempo de conversión está en el orden de milisegundos, por lo que sólo es aplicable si la señal analógica tiene variaciones mínimas a lo largo del tiempo. Otro tipo es el ADC de *aproximaciones sucesivas*, éste es más rápido, su tiempo de conversión está en el orden de microsegundos, puede emplearse para digitalizar señales de audio. Un convertidor de este tipo se encuentra empotrado en los microcontroladores AVR, por ello, en el siguiente apartado se describe su funcionamiento. El tercer tipo de ADC es un convertidor *paralelo* o *flash*, estos convertidores requieren de un hardware mucho más complejo, aunque por sus altas velocidades pueden ser empleados para digitalizar señales de video.

5.1.3.1 ADC de Aproximaciones Sucesivas

En la figura 5.4 se muestra la organización de un ADC de aproximaciones sucesivas de 4 bits. La señal analógica debe ubicarse en V_I y la salida digital resultante queda disponible en los bits *D3, D2, D1* y *D0*. La generación del dato digital no es inmediata, el proceso de conversión requiere de varios ciclos de reloj. Por ello, un ADC debe incluir señales de control (*inicio* y *fin*) para sincronizarse con otros dispositivos.

El bloque de muestreo y retención (*sample and hold*) tiene por objetivo mantener la muestra analógica durante el tiempo de conversión, es necesario porque la información analógica podría cambiar mientras se realiza la conversión y con ello, se generaría información incongruente.

La conversión inicia cuando la señal *inicio* es puesta en alto, la muestra es retenida y se realiza la primera aproximación poniendo en alto al bit más significativo del dato digital (*D3*). El DAC genera el valor analógico que corresponda a la combinación "1000", con el comparador se determina si este valor es mayor o menor que el valor analógico a convertir. Si el valor generado por el DAC es mayor, se remplaza el 1 de *D3* por un 0, en caso contrario el 1 se conserva. Hasta este momento se ha realizado la primera aproximación.

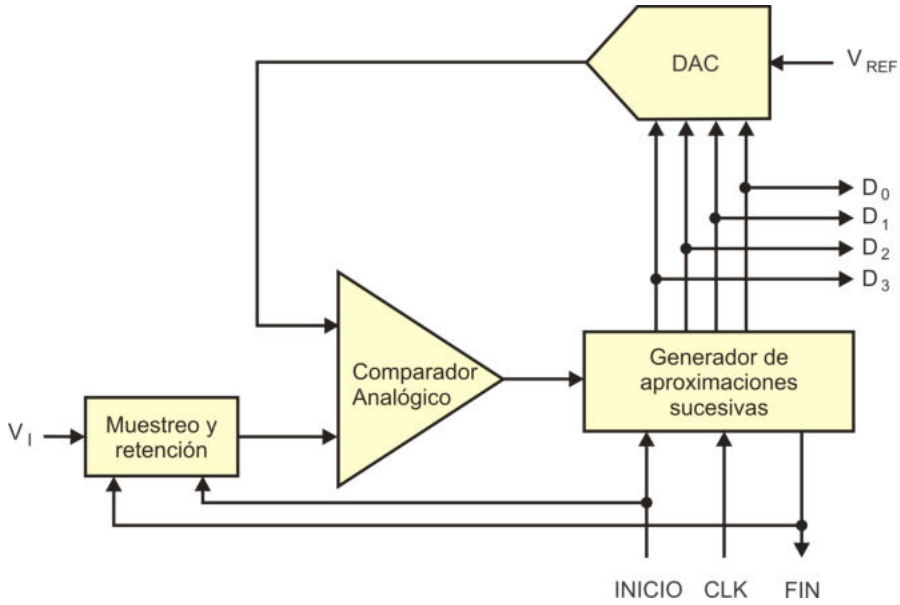


Figura 5.4 Organización de un ADC de aproximaciones sucesivas

El generador de aproximaciones sucesivas continúa colocando un 1 en *D2*, para nuevamente comparar el valor generado por el DAC con el valor a convertir y determinar si este 1 se mantiene o se remplaza por un 0. El mismo proceso debe realizarse para *D1* y *D0*. Una vez que se han generado todos los bits, se indica el fin de la conversión poniendo en alto a la señal *fin* al menos por 1 ciclo de reloj. El voltaje V_{REF} que se suministra al DAC determina el voltaje máximo que puede ser introducido en V_I.

Un comportamiento similar al de un ADC de aproximaciones sucesivas lo realizan las máquinas de cobro automático al momento de regresar el cambio después de recibir un pago (con la diferencia de que un billete o moneda puede figurar más de una vez). Por ejemplo, suponiendo que una máquina puede proporcionar billetes de $50.00 y $20.00 y monedas de $10.00, $5.00 y $1.00, si la máquina va a dar un cambio de $36.00, primero evalúa si alcanza un billete de a $50.00, puesto que no es así, lo descarta y pasa al siguiente. Prueba con uno de a $20.00 que si alcanza, lo proporciona y resta, quedando un residuo de $16.00. Otro de a $20.00 tampoco alcanza, por lo tanto proporciona una moneda de a $10.00. Queda un residuo de $6.00 que es cubierto con una moneda de $5.00 y otra de $1.00.

Algunos ADCs incluyen otras entradas de control como la habilitación del dispositivo (CE*, chip enable*) o la habilitación de la salida (OE*, output enable*) para que sean fácilmente conectados con microprocesadores o microcontroladores.

En ocasiones, la salida *fin* es conectada con la entrada *inicio*, de manera que cuando finaliza una conversión inmediatamente da inicio la siguiente, a este modo de operación se le conoce como “carrera libre”.

5.1.4 El ADC de un AVR

Los microcontroladores ATMega8 y ATMega16 incluyen un ADC de aproximaciones sucesivas de 10 bits, el cual está conectado a un multiplexor analógico que permite seleccionar 1 de 8 canales externos (ADC0, ADC1, ADC2, etc.), excluyendo al ATMega8 con encapsulado PDIP, éste sólo tiene 6 canales. En un ATMega16 se pueden emplear 3 canales para introducir una entrada diferencial (no referida a tierra) e incorporar un factor de ganancia configurable.

En la figura 5.5 se muestran los elementos para seleccionar el voltaje analógico en la entrada, la parte sombreada no está disponible en un ATMega8, para estos dispositivos, el voltaje analógico proviene directamente del multiplexor principal.

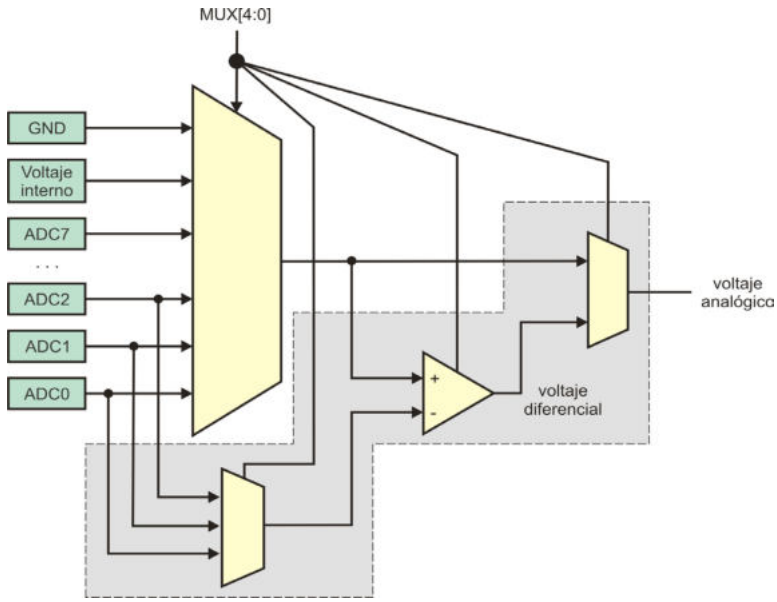


Figura 5.5 Etapa disponible para seleccionar el canal con la información analógica

Los bits **MUX[4:0]** son parte del registro **ADMUX**, éste es un Registro I/O disponible para el manejo del ADC y se describe en la siguiente sección. Como en un ATMega8 se tienen menos opciones, el bit **MUX(4)** no está implementado, en la tabla 5.1 se muestra la selección de la entrada para el ADC de un ATMega8 y en la tabla 5.2 para un ATMega16.

Tabla 5.1 Selección de la entrada para el ADC de un ATMega8

MUX[3:0]	Entrada
0000	ADC0
0001	ADC1
0010	ADC2
. . .	. . .
0111	ADC7
1000 - 1101	Sin uso
1110	Voltaje interno (1.23 V)
1111	0 V

Tabla 5.2 Selección de la entrada para el ADC de un ATMega16

MUX[4:0]	Entrada referida a tierra	Entrada diferencial (positiva)	Entrada diferencial (negativa)	Ganancia
00000	ADC0	No aplica		
00001	ADC1			
. . .	. . .			
00111	ADC7			

MUX[4:0]	Entrada referida a tierra	Entrada diferencial (positiva)	Entrada diferencial (negativa)	Ganancia
01000	No aplica	ADC0	ADC0	10x
01001		ADC1	ADC0	10x
01010		ADC0	ADC0	200x
01011		ADC1	ADC0	200x
01100		ADC2	ADC2	10x
01101		ADC3	ADC2	10x
01110		ADC2	ADC2	200x
01111		ADC3	ADC2	200x
10000		ADC0	ADC1	1x
10001		ADC1	ADC1	1x
10010		ADC2	ADC1	1x
10011		ADC3	ADC1	1x
10100		ADC4	ADC1	1x
10101		ADC5	ADC1	1x
10110		ADC6	ADC1	1x
10111		ADC7	ADC1	1x
11000		ADC0	ADC2	1x
11001		ADC1	ADC2	1x
11010		ADC2	ADC2	1x
11011		ADC3	ADC2	1x
11100		ADC4	ADC2	1x
11101		ADC5	ADC2	1x
11110	Voltaje interno (1.23 V)	No aplica		
11111	0 V			

Aunque el microcontrolador puede operar con osciladores en el orden de MHz, el ADC alcanza su máxima resolución si trabaja a una frecuencia entre 50 KHz y 200 KHz. Es posible emplear una frecuencia mayor con una resolución de 8 bits, pero esta frecuencia debería determinarse en forma práctica. Los AVR incluyen un pre-escalador de 7 bits para generar la frecuencia de trabajo del ADC a partir de la frecuencia del microcontrolador, éste se muestra en la figura 5.6.

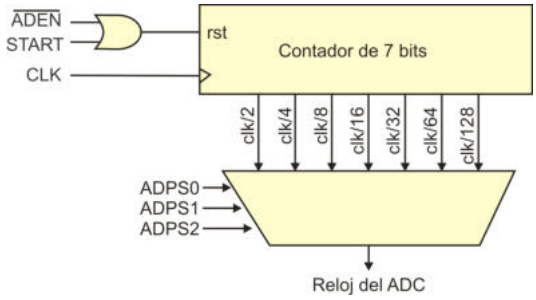


Figura 5.6 Pre-escalador del ADC

Las señales de control del pre-escalador son parte de los bits del Registro A de Control y Estado del ADC (**ADCSRA**, *ADC Control and Status Register A*), el cual se revisa en la siguiente sección. Los factores de división se seleccionan con los bits **ADPS[2:0]**, las diferentes opciones se muestran en la tabla 5.3.

Tabla 5.3 Selección del factor de división en el pre-escalador del ADC

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

Con la primera conversión se inicializa la circuitería analógica, por lo que requiere de 25 ciclos de reloj, para las conversiones siguientes sólo se emplean 13 ciclos.

El ADC y el multiplexor para la selección de la entrada analógica reciben su alimentación en la terminal AVcc. La terminal está disponible para que la circuitería analógica pueda alimentarse con un voltaje diferente al de la parte digital, proporcionando las facilidades para un posible aislamiento. Sólo debe considerarse que AVcc no debe diferir más de ±0.3 V de Vcc.

En la mayoría de aplicaciones es suficiente con conectar a AVcc directamente con Vcc. De hecho, es recomendable realizar esta conexión aun si no se va a emplear al ADC, para la adecuada operación del puerto C de un ATMega8 y del puerto A de un ATMega16, porque en estos puertos se encuentran los multiplexores para las entradas analógicas, como una función alternativa.

Para la conexión de AVcc con Vcc el fabricante recomienda el uso de un filtro pasa bajas, como el mostrado en la figura 5.7, con la finalidad de cancelar el ruido. El filtro es importante si la entrada analógica tiene un valor máximo pequeño, en relación al voltaje de alimentación, o bien, si se utiliza una entrada diferencial, en el caso de un ATMega16.

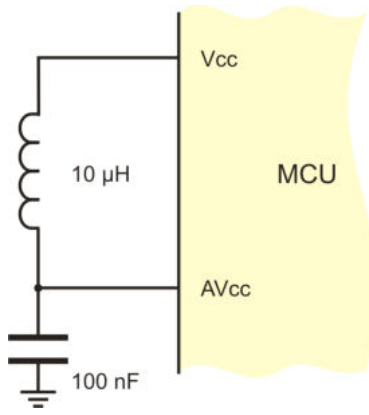


Figura 5.7 Filtro pasa bajas sugerido para conectar AVcc con Vcc

El ADC utiliza un DAC durante el proceso de conversión, por ser un ADC de aproximaciones sucesivas como el mostrado en la figura 5.4. El DAC requiere un voltaje de referencia (VREF), el cual puede ser proporcionado por diferentes fuentes, como se muestra en la figura 5.8, la selección de VREF se realiza con los bits **REFS1** y **REFS0**, del registro **ADMUX**. El voltaje de referencia determina el rango de conversión del ADC, si el voltaje analógico excede a VREF, es codificado como 0x3FF.

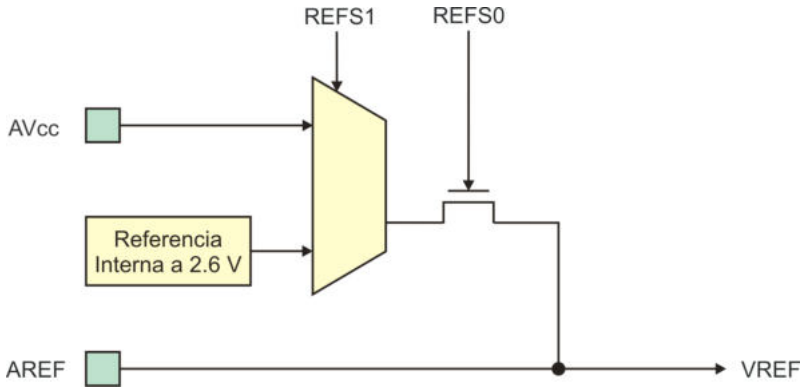


Figura 5.8 Hardware para el voltaje de referencia

El voltaje de referencia puede tomarse de la alimentación analógica (AVcc), de un voltaje interno o de la terminal AREF, en la tabla 5.4 se muestra la selección de estas fuentes. Si se utiliza a AVcc o al voltaje interno (opciones "01" y "11") es recomendable el uso de un capacitor de AREF a tierra, para que el voltaje de referencia tenga inmunidad al ruido.

Tabla 5.4 Alternativas para el voltaje de referencia

REFS1	REFS0	Voltaje de referencia
0	0	Voltaje externo en AREF, referencia interna apagada
0	1	Voltaje externo en AVcc
1	0	Reservado
1	1	Voltaje interno de 2.6 V

Una opción muy simple, desde el punto de vista práctico, consiste en la conexión de la terminal AREF con Vcc, empleando la combinación "00" para los bits **REFS1** y **REFS0**, esta opción también presenta inmunidad al ruido. Únicamente se requiere que la entrada analógica esté acondicionada para proporcionar un voltaje entre 0 y Vcc.

Un factor importante a determinar es la frecuencia máxima permitida en la señal analógica de entrada. Por ejemplo, si el microcontrolador está operando a 1 MHz, para alcanzar una resolución máxima el ADC debe operar con una frecuencia entre 50 y 200 KHz, con un factor de división de 8 se obtiene una frecuencia de 125 KHz. Si el ADC está dedicado sólo a un canal e ignorando el tiempo requerido por la primera muestra, los 13 ciclos por muestra conllevan a una razón de muestreo de 125 KHz/13 = 9.61 KHz. Por lo tanto, de acuerdo con el teorema del muestreo, la frecuencia máxima permisible para la señal de entrada es de 4.8 KHz.

Relacionando las señales de control de la figura 5.4 con el ADC de un AVR, se tiene que: el *inicio* de la conversión se realiza poniendo en alto al bit **ADSC** (*Start Conversion*), el *fin* se indica con la puesta en alto de la bandera **ADIF** (*Interrupt Flag*); ésta puede sondearse por software o bien, si se ajusta al bit **ADIE** (*Interrupt Enable*), va a producir una interrupción, estos bits están en el registro **ADCSRA**.

Un ATMega8 puede ser configurado para operar en un modo de carrera libre, de manera que al finalizar una conversión inicie con la siguiente, este modo se habilita con el bit **ADFR** (*ADC Free Running*). En el ATMega16, además del modo de carrera libre, se puede configurar al hardware para que el inicio de una conversión sea disparado por algún evento de otro recurso del microcontrolador.

5.1.5 Registros para el Manejo del ADC

El dato digital de 10 bits, resultante de la conversión, queda disponible en 2 Registros I/O: **ADCH** (para la parte alta) y **ADCL** (para la parte baja). En lenguaje C puede hacerse referencia a ambos tratándolos como **ADCW**. Los 10 bits de información están alineados a la derecha, de la siguiente manera:

	7	6	5	4	3	2	1	0	
0x05	-	-	-	-	-	-	ADC9	ADC8	**ADCH**
0x04	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADC1	ADC0	**ADCL**

La alineación puede cambiarse a la izquierda si se pone en alto al bit **ADLAR** (*ADC Left Adjust Result*) del registro **ADMUX**, con ello, la información se organiza como:

	7	6	5	4	3	2	1	0	
0x05	ADC9	ADC8	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	**ADCH**
0x04	ADC1	ADC0	-	-	-	-	-	-	**ADCL**

Esta organización es útil en aplicaciones donde resulte suficiente una resolución de 8 bits. En esos casos, sólo se emplearía al registro **ADCH** y ignorando el contenido del registro **ADHL**.

En el registro **ADMUX** (multiplexor del ADC) también se pueden seleccionar otros parámetros, sus bits son:

	7	6	5	4	3	2	1	0	
0x07	REFS1	REFS0	ADLAR	MUX4	MUX3	MUX2	MUX1	MUX0	**ADMUX**

- **Bits 7 y 6 – REFS[1:0]: Bits para seleccionar el voltaje de referencia del ADC**

  La tabla 5.4 muestra las alternativas para el voltaje de referencia, necesario para la conversión.

- **Bit 5 – ADLAR: Bit para alinear el resultado de la conversión a la izquierda**

  Al poner en alto este bit, los 10 bits resultantes de la conversión se alinean a la izquierda, dentro de los registros **ADCH** y **ADCL**.

- **Bits 4 al 0 – MUX[4:0]: Bits para seleccionar el canal de entrada analógico**

  En un ATMega8 el bit **MUX(4)** no está implementado, para estos dispositivos se tienen las opciones descritas en la tabla 5.1. En los ATMega16 las opciones para la entrada analógica se describen en la tabla 5.2, en donde se observa que es posible seleccionar una entrada diferencial con algunos factores de ganancia.

El control del ADC se realiza con el Registro A de Control y Estado del ADC (**ADCSRA**, *ADC Control and Status Register A*), los bits de este registro son:

	7	6	5	4	3	2	1	0	
0x06	ADEN	ADSC	ADFR/ ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0	**ADCSRA**

- **Bit 7 – ADEN: Habilitador del ADC**

  Con 0 el ADC está apagado y no es posible utilizarlo.

- **Bit 6 – ADSC: Inicio de conversión (*Start Conversion*)**

  La conversión inicia al escribirle 1, se limpia automáticamente por hardware. La primera conversión requiere de 25 ciclos de reloj, las siguientes de 13.

- **Bit 5 – ADFR ó ADATE: Activa el modo de carrera libre (*ADC Free Running*) o habilita un auto disparo (*ADC Auto Trigger Enable*)**

  La función de este bit depende del dispositivo, en un ATMega8 establece un modo de carrera libre, es decir, al terminar una conversión inicia con la siguiente. El bit **ADSC** debe ponerse en alto para dar paso a las conversiones.

  En un ATMega16 este bit habilita un auto disparo del ADC, es decir, el ADC inicia una conversión cuando ocurre un evento generado por otro recurso, el evento se configura con los bits **ADTS** en el registro **SFIOR**, con los cuales también puede elegirse el modo de carrera libre.

- **Bit 4 – ADIF: Bandera de fin de conversión**

  Se pone en alto indicando el fin de una conversión, puede generar una interrupción o sondearse vía software. La bandera se limpia automáticamente por hardware si se configura la interrupción. Al emplear sondeo, la bandera se limpia al escribirle nuevamente un 1 lógico.

- **Bit 3 – ADIE: Habilitador de interrupción**

  Habilita la interrupción por el fin de una conversión analógica a digital.

- **Bits 2 al 0 – ADPS[2:0]: Bits para seleccionar el factor de división del pre-escalador del ADC**

  El pre-escalador del ADC se mostró en la figura 5.6 y sus factores de división se describieron en la tabla 5.3.

En un ATMega16, además del modo de carrera libre, es posible iniciar automáticamente una conversión con un evento de otro recurso. El evento se define a través de los bits **ADTS[2:0]** (*ADC Auto Trigger Source*), estos bits son los más significativos del Registro I/O de Función Especial (**SFIOR**, *Special Function IO Register*), los bits de **SFIOR** son:

	7	6	5	4	3	2	1	0	
0x30	ADTS2	ADTS1	ADTS0	-	ACME	PUD	PSR2	PSR10	**SFIOR**

En la tabla 5.5 se muestra cómo seleccionar el modo de carrera libre y los eventos que automáticamente inician una conversión analógica a digital. Se observa que el modo de carrera libre corresponde con la combinación “000”, por lo que si el registro **SFIOR** conserva su valor de reinicio (0x00), la activación del modo de carrera libre se realiza de la misma manera, tanto en un ATMega16, como en un ATMega8.

Tabla 5.5 Selección del modo de carrera libre o de la fuente de disparo de una conversión

ADTS2	ADTS1	ADTS0	Fuente de disparo
0	0	0	Modo de carrera libre
0	0	1	Comparador analógico
0	1	0	Interrupción externa 0
0	1	1	Coincidencia por comparación en el temporizador 0
1	0	0	Desbordamiento del temporizador 0
1	0	1	Coincidencia con el comparador B del temporizador 1
1	1	0	Desbordamiento del temporizador 1
1	1	1	Captura en el temporizador 1

5.1.6 Ejemplos de Uso del Convertidor Analógico a Digital

En esta sección se muestran 2 ejemplos del uso de un ADC, en el primer ejemplo las soluciones se muestran en ensamblador y lenguaje C. En el segundo ejemplo se asume que existe una biblioteca de funciones para el manejo de un display de cristal líquido (LCD, *liquid crystal display*) y por lo tanto, sólo se presenta la solución en lenguaje C.

Ejemplo 5.1 Desarrolle un sistema con base en un ATMega8, que mantenga la temperatura de una habitación en el rango entre 18 y 23 °C, aplicando la siguiente idea: Si la temperatura es mayor a 23 °C, el sistema debe activar un ventilador, si es menor a 18 °C, el sistema debe activar un calefactor. Para temperaturas entre 18 y 23 °C no se debe activar alguna carga.

La solución inicia con la definición del hardware, para detectar la temperatura se emplea un sensor LM35, el cual entrega 10 mV/°C. Puesto que el sistema trabaja con la temperatura ambiente, el voltaje proporcionado por el sensor es amplificado por un factor de 10, de manera que para temperaturas entre 0 y 50 °C, el sensor entrega voltajes entre 0 y 5 V. Con ello, el voltaje de alimentación del ATMega8 (5 V) puede emplearse como referencia para el ADC.

También se asume que el ventilador y el calefactor están acondicionados para activarse con una señal digital, en la figura 5.9 se muestra el hardware propuesto.

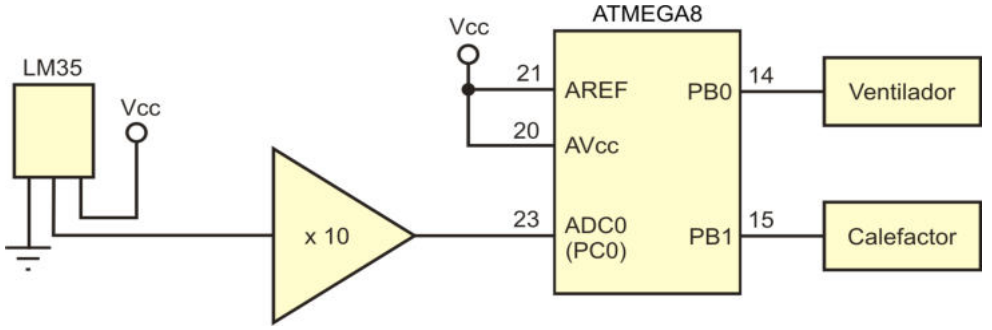


Figura 5.9 Hardware para el problema 5.1

En lo que refiere al software, puesto que el sistema está dedicado únicamente al control de temperatura, no es necesario un manejo de interrupciones. Dentro de un lazo infinito se inicia una conversión analógica a digital, se espera su culminación y el resultado es evaluado para determinar si se activa una salida.

Puesto que el ADC es de 10 bits, su resolución es de:

$$Resolución = \frac{V_{MAX}}{2^{10} - 1} = \frac{5\,V}{1023} = 4.8875\,mV$$

Con esta consideración, en la tabla 5.6 se tienen los valores digitales para las temperaturas de interés, el voltaje analógico a considerar es el de la salida del amplificador y el valor digital se ha redondeado para manipular números enteros.

Tabla 5.6 Valores digitales para las temperaturas de interés

Temperatura	Voltaje analógico (Van)	Valor digital (Van/Resolución)
18 °C	1.8 V	1.8 V/ 4.8875 mV = 368.28 ≈ 368
23 °C	2.3 V	2.3 V/ 4.8875 mV = 470.58 ≈ 471

Otro supuesto es que el ATMega8 trabaja a 1 MHz, para dividir esta frecuencia entre 8 y operar al ADC dentro del rango adecuado. El programa en lenguaje C es:

```
#define   inf   368          // Valor digital para 18 grados
#define   sup   471          // Valor digital para 23 grados

#include   <avr/io.h>

int   main(void) {
unsigned int   temp;

DDRB = 0xFF;                 // Puerto B como salida
PORTB = 0x00;                // Inicia con las salidas sin activar

ADMUX  = 0b00000000;         // Selecciona ADC0 y Vref en AREF
ADCSRA = 0b11000011;         // Habilita ADC, inicia conversión y divide entre 8
   while(1) {

     while( !(ADCSRA & 1 << ADIF ) );    // Espera fin de conversión
     ADCSRA |= 1 << ADIF;                // Limpia bandera
     temp = ADCW;                        // Obtiene la temperatura
     if( temp > sup )
        PORTB = 0x01;                    // Activa ventilador
     else if( temp < inf )
        PORTB = 0x02;                    // Activa calefactor
     else
        PORTB = 0x00;                    // Salidas sin activar

     ADCSRA |= 1 << ADSC;                // Inicia nueva conversión
   }
}
```

Para la versión en ensamblador, puesto que se comparan datos de 16 bits, primero se compara el byte menos significativo, si se genera un acarreo, éste debe ser considerado al comparar el byte más significativo, el código es el siguiente:

```
       .include      "m8def.inc"
       .EQU   inf = 368           ; Valor digital para 18 grados
       .EQU   sup  = 471          ; Valor digital para 23 grados

       CLR    R16
       OUT    ADMUX, R16          ; Selecciona ADC0 y Vref en AREF
       LDI    R16, 0b11000011
       OUT    ADCSRA, R16         ; Hab. ADC, inicia conversión y divide entre 8

       LDI    R16, 0xFF
       OUT    DDRB, R16           ; Puerto B como salida
       CLR    R16
       OUT    PORTB, R16          ; Inicia con las salidas sin activar

LOOP:  SBIS   ADCSRA, ADIF        ; Espera fin de conversión
       RJMP   LOOP
```

```
        SBI     ADCSRA, ADIF        ; Limpia bandera

; Se destinan R27 y R26 para el valor digital proporcionado por el ADC
        IN      R27, ADCH
        IN      R26, ADCL
                                    ; Comparación con el límite superior
        CPI     R26, LOW(sup)       ; Compara parte baja
        LDI     R16, HIGH(sup)
        CPC     R27, R16            ; Compara parte alta, considera acarreo
        BRLO    no_sup              ; brinca si temp < sup
        BREQ    no_sup              ; o si temp = sup
        LDI     R16, 0x01           ; Activa ventilador
        OUT     PORTB, R16          ; cuando temp > sup
        RJMP    fin_comp
no_sup:                             ; Comparación con el límite inferior
        CPI     R26, LOW(inf)       ; Compara parte baja
        LDI     R16, HIGH(inf)
        CPC     R27, R16            ; Compara parte alta, considera acarreo
        BRSH    no_inf              ; brinca si temp >= inf
        LDI     R16, 0x02           ; Activa calefactor
        OUT     PORTB, R16          ; cuando temp < inf
        RJMP    fin_comp
no_inf:
        LDI     R16, 0x00           ; salidas sin activar
        OUT     PORTB, R16          ; cuando temp < sup y temp > inf
fin_comp:
        SBI     ADCSRA, ADSC        ; Inicia nueva conversión
        RJMP    LOOP                ; Regresa al lazo infinito
```

El tiempo de ejecución del código puede mejorarse si se destinan algunos registros para el manejo de constantes y se asignan sus valores fuera del lazo infinito, con ello, se reduce el número de instrucciones a ejecutar en cada iteración.

Ejemplo 5.2 Realice un termómetro digital con un rango de 0 a 50 ^{0} C, con salida a un LCD y actualizando el valor de la temperatura cada medio segundo.

El LCD se conecta al puerto B de un ATMega8, como se muestra en la figura 5.10. Un LCD puede ser manipulado con una interfaz de 4 bits para que utilice sólo un puerto del microcontrolador. Para este ejemplo se asume que existe una biblioteca con las funciones necesarias para el manejo del LCD, la descripción de su funcionamiento es parte del capítulo 8.

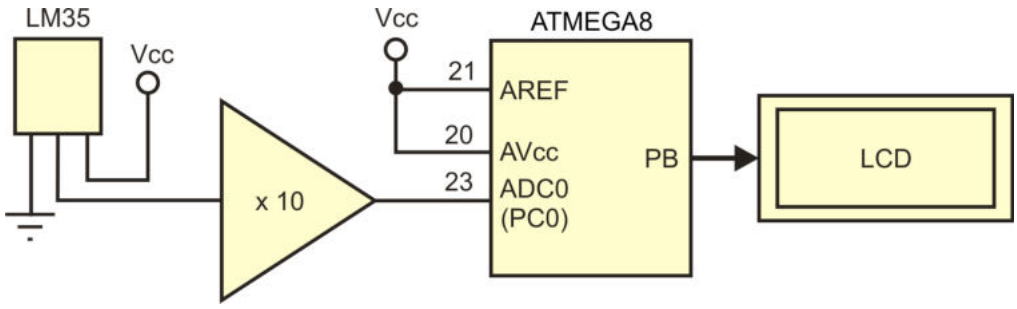


Figura 5.10 Hardware para el problema 5.2, referente al termómetro digital

Se emplea al temporizador 1 para obtener el intervalo de actualización de la temperatura. El temporizador genera un evento cada medio segundo y en su ISR inicia la conversión analógico a digital. Suponiendo el uso del oscilador interno de 1 MHz, el temporizador 1 debe interrumpir cada 500 000 ciclos, puesto que no es un valor que alcance en 16 bits se utiliza al pre-escalador con un factor de división de 8, dado que 500 000/8 = 62 500, este número si alcanza en 16 bits. Se utilizan eventos de comparación, configurando al modo CTC para limpieza automática del temporizador.

Al finalizar la conversión se produce otro evento, en cuya ISR se actualiza la temperatura en el LCD. El acondicionamiento del sensor es el mismo que en el ejercicio anterior.

El código en lenguaje C es:

```
#include  "LCD.h"                  // Biblioteca con las funciones para el LCD
#include <avr/io.h>
#include <avr/interrupt.h>

ISR (ADC_vect) {                   // ISR para el fin de conversión del ADC
float   temperatura;               // Variable con la temperatura
int     ent;                       // Variables auxiliares para convertir el valor
float   dec;                       // de la temperatura en una secuencia de
unsigned  char  d, u, f;           // caracteres

       temperatura = 0.0488758*ADCW;     // La temperatura en punto flotante
       ent = temperatura;                // Parte entera de la temperatura
       dec = temperatura - ent;          // Parte fraccionaria de la temperatura
       dec = dec * 10;                   // Para obtener sólo los decimales

// La información se convierte a caracteres
       d = ent/10;                 // Decenas de la parte entera de la temperatura
       u = ent - d*10;             // Unidades de la parte entera de la temperatura
       f = dec;                    // Sólo un dígito de la parte fraccionaria
// Despliegue de la información, byte por byte. Suma 0x30 para código ASCII
       LCD_cursor(0x08);                 // Ubica al cursor
       LCD_write_data(d + 0x30);         // Escribe decenas
       LCD_write_data(u + 0x30);         // Escribe unidades
```

```
        LCD_write_data('.');              // Escribe el punto decimal
        LCD_write_data(f + 0x30);         // Escribe decimales
}

ISR (TIMER1_COMPA_vect) {                 // ISR del temporizador 1

        ADCSRA = ADCSRA | 0x40;           // Cada medio segundo inicia una
}                                         // Conversión Analógico a Digital

int main(void)                      // Programa Principal
{
        DDRB = 0xFF;                // Puerto B como salida (para el LCD)
        LCD_reset();                // Inicializa al LCD

        ADMUX = 0x00;               // Entrada analógica en ADC0 y Vref en AREF
        ADCSRA = 0xCB;              // Habilita ADC, inicia conversión, divide por 8
                                    // y habilita interrupción por fin de conversión
        TIMSK = 0x10;               // Configura al temporizador 1 para que
        OCR1A = 62499;              // interrumpa cada medio segundo, con eventos
        TCCR1A = 0x00;              // de comparación, modo CTC y un factor de
        TCCR1B = 0x0A;              // pre-escala de 8
        LCD_write_cad(" TEMP = ", 8);     // Cadena constante de 8 caracteres

        sei();                      // Habilitador global de interrupciones

        while( 1 ) {                // En el lazo infinito permanece ocioso
                asm("nop");
    }
}
```

Al codificar en un lenguaje de alto nivel se simplifica el uso de datos en punto flotante.

5.2 Comparador Analógico

El comparador analógico es un recurso que indica la relación existente entre dos señales analógicas externas, es útil para aplicaciones en donde no precisa conocer el valor digital de una señal analógica, sino que es suficiente con determinar si ésta es mayor o menor que alguna referencia.

5.2.1 Organización del Comparador Analógico

En la figura 5.11 se muestra la organización del comparador, las entradas analógicas son tomadas de las terminales AIN0 y AIN1 (PD6 y PD7 en un ATMega8, y PB2 y PB3 en un ATMega16). Estas terminales se conectan a un amplificador operacional en lazo abierto, el cual se va a saturar cuando AIN0 sea mayor que AIN1, poniendo en alto al bit **ACO** (ACO, *Analog Comparator Output*).

Por medio de interruptores analógicos es posible remplazar la entrada AIN0 por una referencia interna cuyo valor típico es de 1.23 V, esto se consigue poniendo en alto al bit **ACBG** (*Analog Comparator Band Gap*). La entrada AIN1 puede remplazarse por la salida del multiplexor del ADC, de manera que es posible comparar más de una señal analógica con la misma referencia. Para ello se requiere que el ADC esté inhabilitado (**ADEN** = 0) y se habilite el uso del multiplexor por el comparador (**ACME** = 1).

Un cambio en el bit **ACO** puede generar una interrupción y además, producir un evento de captura en el temporizador 1, esto podría ser útil para determinar el tiempo durante el cual una señal superó a otra. La habilitación de la interrupción se realiza con el bit **ACIE** y la selección de la transición que genera la interrupción se realiza con los bits **ACIS[1:0]**, éstos y los demás bits para el manejo del comparador son parte del registro para el control y estado del comparador analógico (**ACSR**, *Analog Comparator Control and Status Register*).

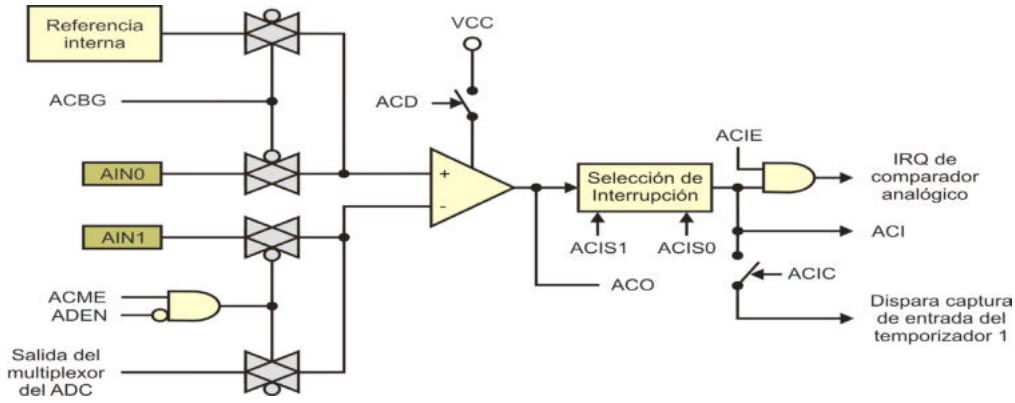


Figura 5.11 Organización del comparador analógico

5.2.2 Registros para el Manejo del AC

El registro **ACSR** es el más importante para el uso del comparador, los bits de este registro son:

	7	6	5	4	3	2	1	0	
0x08	ACD	ACBG	ACO	ACI	ACIE	ACIC	ACIS1	ACIS0	**ACSR**

- **Bit 7 – ACD: Inhabilita al AC**

  El AC puede ser inhabilitado para minimizar el consumo de energía (con 0 el AC está activo). Si el AC está inactivo y se va a activar nuevamente, se sugiere inhabilitar su interrupción porque podría generar un evento.

- **Bit 6 – ACBG: Selecciona el voltaje interno (V_{BG}, *Band Gap*)**

  Al ponerlo en alto, la entrada en la terminal positiva proviene de un voltaje de referencia interno.

- **Bit 5 – ACO: Salida del comparador analógico**

  La salida del AC se sincroniza y se muestra en este bit, la sincronización toma 1 ó 2 ciclos de reloj.

- **Bit 4 – ACI: Bandera de interrupción**

  Su puesta en alto va a generar una interrupción, siempre que la interrupción esté habilitada (**ACIE** en alto). Su valor depende de las entradas del comparador y de los bits **ACIS[1:0]**.

- **Bit 3 – ACIE: Habilitador de interrupción**

  Habilita la interrupción por una transición en **ACO**, definida en los bits **ACIS[1:0]**.

- **Bit 2 – ACIC: Habilita la captura de entrada**

  Con la puesta en alto de este bit se conecta la salida del comparador con el hardware del temporizador 1, para disparar su función de captura de entrada. Se debe configurar el flanco requerido y la cancelación de ruido, además de habilitar la interrupción por captura de entrada, poniendo en alto al bit **TICIE1** del registro **TIMSK**.

- **Bits 1 al 0 – ACIS[1:0]: Bits para seleccionar la transición para la interrupción**

  En estos bits se define la transición que produce la interrupción del comparador analógico, sus opciones se describen en la tabla 5.7.

Tabla 5.7 Selección de la transición que genera la interrupción del AC

ACIS1	ACIS0	Tipo de transición
0	0	Cualquier conmutación en ACO
0	1	Reservado
1	0	Flanco de bajada en ACO
1	1	Flanco de subida en ACO

El registro de función especial **SFIOR** también se involucra con el comparador analógico porque en la posición 3 de este registro se encuentra al bit **ACME** (*Analog Comparator Multiplexor Enable*), este bit habilita el uso del multiplexor del ADC para proporcionar la entrada negativa al comparador.

	7	6	5	4	3	2	1	0	
0x30	ADTS2	ADTS1	ADTS0	-	ACME	PUD	PSR2	PSR10	SFIOR

De acuerdo con el hardware de la figura 5.11, el empleo del multiplexor también requiere que el bit **ADEN** (*ADC enable*) tenga 0, inhabilitando así al **ADC**. El bit **ADEN** es parte del registro **ADCSRA**:

	7	6	5	4	3	2	1	0	
0x06	ADEN	ADSC	ADFR/ ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0	ADCSRA

Si el ADC se habilita, aunque el bit **ACME** tenga 1, la entrada negativa del comparador va a ser tomada de la terminal AIN1.

Cuando se conecta la salida del multiplexor con la entrada negativa del AC (**ACME** = 1 y **ADEN** = 0), el canal se selecciona con los bits **MUX[2:0]**, los cuales corresponden con los bits menos significativos del registro **ADMUX**, teniendo la posibilidad de emplear uno de 8 canales (6 en un ATMega8 con encapsulado PDIP).

	7	6	5	4	3	2	1	0	
0x07	REFS1	REFS0	ADLAR	MUX4	MUX3	MUX2	MUX1	MUX0	ADMUX

Los bits **MUX[4:3]** no están involucrados en la selección del canal porque para la entrada negativa del AC no es aplicable el uso de valores constantes o entradas diferenciales, si se tratase de un ATMega16.

5.2.3 Ejemplos de uso del Comparador Analógico

El comparador es un recurso muy simple en su uso, por ello, en esta sección se muestran 2 ejemplos con la solución codificada únicamente en lenguaje C. Las versiones en ensamblador pueden desarrollarse con la misma estructura.

Ejemplo 5.3 Realice un sistema que encienda un ventilador cuando la temperatura esté por encima de 20 °C, en caso contrario que el ventilador permanezca apagado.

En la entrada AIN0 del comparador se conecta un sensor de temperatura acondicionado para que entregue un voltaje de 0 a 5 V ante un rango de temperatura de 0 a 50 °C y en la entrada AIN1 un voltaje constante de 2.0 V, con ello, el estado del ventilador corresponde con la salida del comparador, reflejada en el bit **ACO**, en la figura 5.12 se muestra el acondicionamiento del hardware.

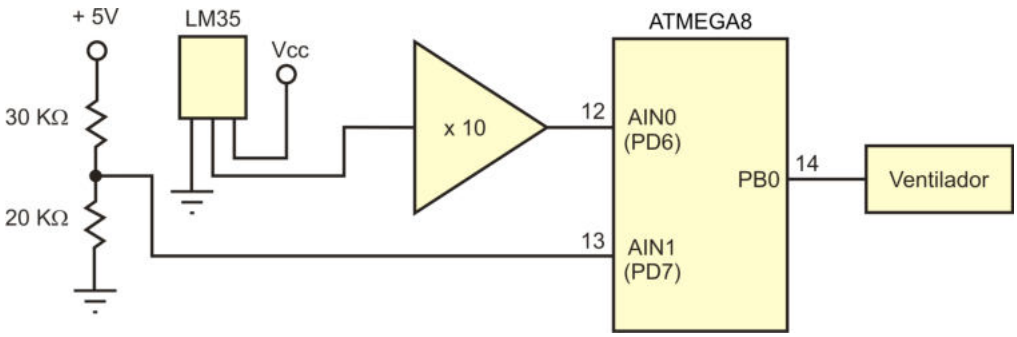


Figura 5.12 Hardware para mostrar el uso del comparador analógico

Con respecto al software, se emplea la interrupción por cambios en el AC, de manera que en la ISR únicamente se actualice la salida del ventilador de acuerdo con el bit **ACO**. El programa en lenguaje C es:

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

ISR (ANA_COMP_vect)                          // ISR para los cambios en el comparador
{
       if( ACSR & 1 << ACO )                 // Si la temperatura es alta
              PORTB = PORTB | 0x01;          // enciende el ventilado
       else
              PORTB = PORTB & 0xFE;          // Sino, lo apaga
}

int main(void) {                             // Programa Principal

       DDRB = 0xFF;          // Puerto B como salida
       PORTB = 0x00;         // Inicia con el ventilador apagado
       ACSR = 0x08;          // Configura al CA, el bit ACO inicia con 0,
                             // si la temperatura es alta produce la interrupción
       asm("sei");           // Habilitador global de interrupciones

       while (1)             // Lazo infinito, permanece ocioso
              asm("nop");
}
```

Ejemplo 5.4 Diseñe el control de un móvil seguidor de línea, compuesto por 2 llantas traseras manejadas por motores independientes y una llanta de libre movimiento al frente. El móvil debe seguir una línea blanca en una superficie con fondo negro, utilizando el comparador analógico y 2 canales del ADC de un ATMega8. Suponga que los sensores se han acondicionado de manera que, cuando se encuentran en una zona oscura presentan un voltaje mayor a 2.5 V y en una zona clara su voltaje está por debajo de ese nivel. Ubique los sensores al frente del móvil, uno en cada extremo, dejando la línea blanca al centro.

El estado de los sensores define el encendido de los motores. Si ambos detectan una zona oscura, los 2 motores deben estar encendidos (móvil en línea recta). Cuando un sensor detecta una zona clara (debido a una curva), se debe apagar al motor del mismo lado para provocar un giro en el móvil. Se espera que en ningún momento los 2 sensores detecten una zona clara, sin embargo, si eso sucede, también se deben encender los 2 motores, buscando que el móvil abandone esta situación no esperada.

En la figura 5.13 se hace un bosquejo del hardware, los sensores se conectan en 2 de los canales del ADC, para que alternadamente remplacen la entrada AIN1 del AC. En la entrada AIN0 se conecta una referencia constante de 2.5 V.

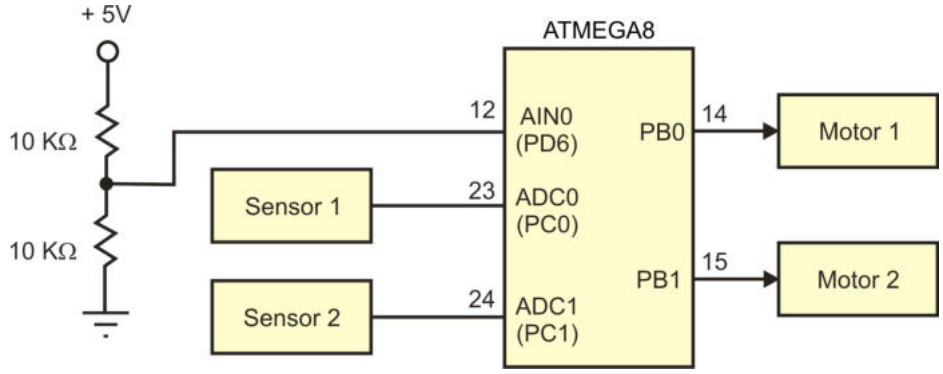


Figura 5.13 Hardware para el control de un móvil, seguidor de línea

Con respecto al software, en un lazo infinito se debe obtener el estado de los sensores para definir las salidas de los motores. No se utilizan interrupciones. El código en lenguaje C es:

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

int main(void)       {              // Programa Principal
unsigned  char       sens;          // Para el estado de los sensores

       DDRB = 0xFF;                 // Puerto B como salida
       PORTB = 0x03;                // Inicia con los 2 motores encendidos
                                    // El móvil inicia en una línea recta
       ADCSRA = 0x00;               // Asegura la desactivación del ADC
       SFIOR = 0x08;                // Conecta al multiplexor con el AC

       while(1) {
              sens = 0x00;
              ACSR = 0x80;                  // Desactiva al AC
              ADMUX = 0x00;                 // Selecciona el canal 0
              ACSR = 0x00;                  // Activa al AC
              asm("nop");                   // Espera el resultado del AC
              if(!(ACSR & 1 << ACO))        // Si ( Sensor 1 en zona oscura )
                     sens |= 0x01;          // motor 1 encendido

          ACSR = 0x80;                  // Desactiva al AC
```

```
            ADMUX = 0x01;               // Selecciona el canal 1
            ACSR = 0x00;                // Activa al AC
            asm("nop");                 // Espera el resultado del AC
            if(!(ACSR & 1 << ACO))      // Si ( Sensor 2 en zona oscura )
                    sens |= 0x02;       // motor 2 encendido
                                           // Si sólo un sensor está en
            if( sens == 0x01 || sens == 0x02 )  //  la zona oscura, enciende
                    PORTB = sens;          // el motor correspondiente
            else
                    PORTB = 0b00000011;    // Sino enciende los 2 motores
               }
}
```

En el código se muestra que es necesario desactivar al comparador analógico antes de definir el canal de entrada. Una vez activado el comparador, es conveniente esperar uno o dos ciclos de reloj, para que la salida del CA sea estable.

5.3 Ejercicios

Los siguientes ejercicios son para utilizar el ADC o el AC en combinación con otros recursos, pueden implementarse en un ATMega8 o en un ATMega16, programando con ensamblador o lenguaje C.

1. Genere una señal PWM de 10 bits en modo no invertido, en donde el ciclo de trabajo sea proporcional al voltaje proporcionado por un potenciómetro conectado al canal 0 del ADC, como se muestra en la figura 5.14.

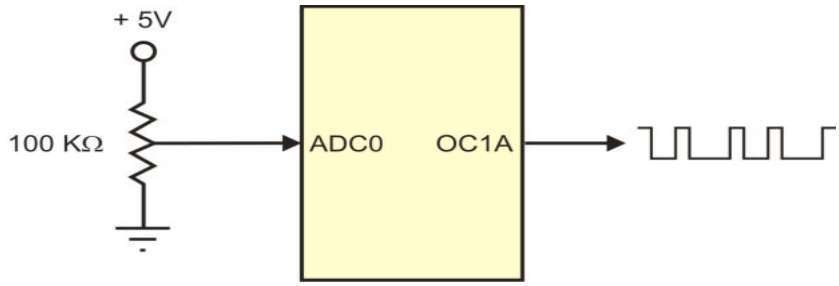


Figura 5.14 Generación de una señal PWM con un ciclo útil proporcional al voltaje en el ADC

2. Un *fotoresistor* iluminado con luz solar presenta una resistencia con un valor de 300 ohms, sin iluminación la resistencia está por encima de 20 Kohms. Desarrolle un circuito que encienda una lámpara si la resistencia en el sensor es mayor a 15 Kohms, y la apague cuando la resistencia esté por debajo de 2 Kohms.

   Se introduce una curva de histéresis para evitar oscilaciones. El *fotoresistor* debe acondicionarse para que entregue un voltaje proporcional a la resistencia. El problema puede resolverse de 2 formas diferentes:

a. El voltaje debido al *fotoresistor* se introduce al microcontrolador a través de su ADC y vía software se realizan las comparaciones.

b. Se utiliza al comparador, en una entrada se introduce el voltaje debido al sensor y en la otra se alternan los voltajes producidos con resistores de 2 y 15 Kohms.

3. Modifique el ejemplo 5.4 manejando los motores del móvil con señales PWM. En una línea recta, ambos motores deber tener un ciclo útil del 100 %. En una curva, el motor del sensor que detecte la zona clara debe trabajar con un ciclo útil del 20 % y el otro al 80 %, con la finalidad de que las curvas sean tomadas de una manera suave, para evitar un zigzagueo en el móvil.

4. Con respecto a la figura 5.15, las entradas Vin, Vref1 y Vref2 son señales analógicas y siempre ocurre que Vref1 < Vref2. De acuerdo con los resultados que proporcione el comparador analógico, utilizando el temporizador 1, genere una señal PWM a una frecuencia aproximada de 100 Hz, con un ciclo de trabajo de:

   - 10 %: si V_{in} es menor a V_{ref1}
   - 50 %: si V_{in} está entre V_{ref1} y V_{ref2}
   - 90 %: si V_{in} es mayor a V_{ref2}

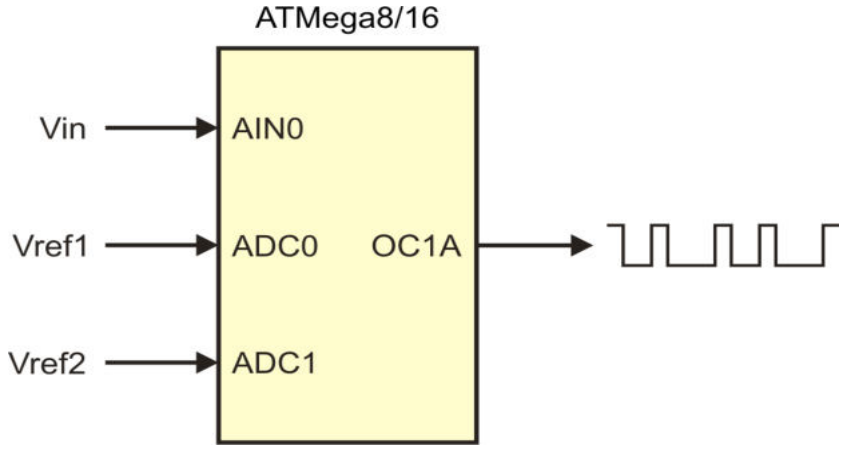


Figura 5.15 Generación de una señal PWM con un ciclo útil dependiente del AC

6. Interfaces para una Comunicación Serial

Los microcontroladores ATMega8 y ATMega16 incluyen 3 recursos para comunicarse de manera serial con dispositivos externos, empleando interfaces estándares. Estos recursos son: Un Transmisor-Receptor Serial Universal Síncrono y Asíncrono (USART, *Universal Synchronous and Asynchronous Serial Receiver and Transmitter*), una Interfaz para Periféricos Seriales (SPI, *Serial Peripheral Interface*) y una Interfaz Serial de Dos Hilos (TWI, *Two-Wire Serial Interface*). Las 3 interfaces son descritas en el presente capítulo.

6.1 Comunicación Serial a través de la USART

La USART incluye recursos independientes para transmisión y recepción, esto posibilita una operación *full-duplex*, es decir, es posible enviar y recibir datos en forma simultánea. La terminal destinada para la transmisión de datos es TXD y para la recepción es RXD. Estas terminales corresponden con PD1 (TXD) y PD0 (RXD), tanto en el ATMega8 como en el ATMega16.

La comunicación puede ser síncrona o asíncrona, la diferencia entre estos modos se ilustra en la figura 6.1. En una comunicación síncrona, el emisor envía la señal de reloj, además de los datos, de esta forma el receptor va obteniendo cada bit en un flanco de subida o bajada de la señal de reloj, según se haya configurado (la terminal dedicada para el envío o recepción de la señal de reloj es XCK). En una comunicación asíncrona el emisor únicamente envía los datos, no se envía señal de reloj.

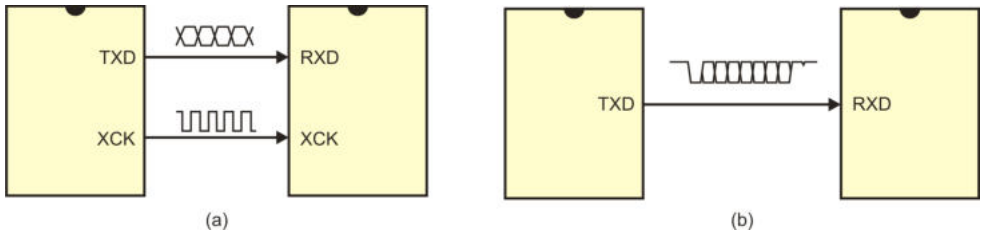


Figura 6.1 Comunicación serial (a) síncrona y (b) asíncrona

Por lo tanto, antes de cualquier transferencia de datos, para el transmisor y el receptor, se deben definir los siguientes parámetros:

- **Velocidad de transmisión (*Baudrate*):** Se refiere a la cantidad de bits que se transmiten en un segundo, se mide en *bits/segundo*, usualmente referido como *bauds* o simplemente *bps*.
- **Número de bits de datos**: Cada dato se integra por un número pre-definido de bits, pueden ser 5, 6, 7, 8 ó hasta 9 bits.

- **Bit de paridad:** Es un bit de seguridad que acompaña a cada dato, se ajusta automáticamente en alto o bajo para complementar un número par de 1's (paridad par) o un número impar de 1's (paridad impar). El emisor envía al bit de paridad después de enviar los bits de datos, el receptor lo calcula a partir de los datos recibidos y compara el bit de paridad generado con el bit de paridad recibido, una diferencia entre ellos indica que ocurrió un error en la transmisión. Por ello, es importante definir el uso del bit de paridad y configurar si va a ser par o impar.

- **Número de bits de paro:** Los bits de paro sirven para separar 2 datos consecutivos. En este caso, se debe definir si se utiliza 1 ó 2 bits de paro.

Una trama serial se compone de:

- 1 bit de inicio (siempre es un 0 lógico).
- 5, 6, 7, 8 o 9 bits de datos, iniciando con el menos significativo.
- Bit de paridad, par o impar, si se configuró su uso.
- 1 o 2 bits de paro (siempre son un 1 lógico).

En la figura 6.2 se ilustra una trama serial, considerando 8 bits de datos, paridad par y 1 bit de paro. Se observa que la línea de comunicación mantiene un nivel lógico alto mientras no se inicie con la transmisión de un dato.

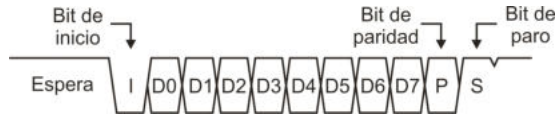


Figura 6.2 Trama serial de datos

6.1.1 Organización de la USART

La USART se compone de 3 bloques principales, éstos se observan en la figura 6.3. También pueden verse los registros **UCSRA**, **UCSRB** y **UCSRC**, con estos registros se realiza la configuración y se conoce el estado de la USART (**UCSR**, *USART Configuration and State Register*).

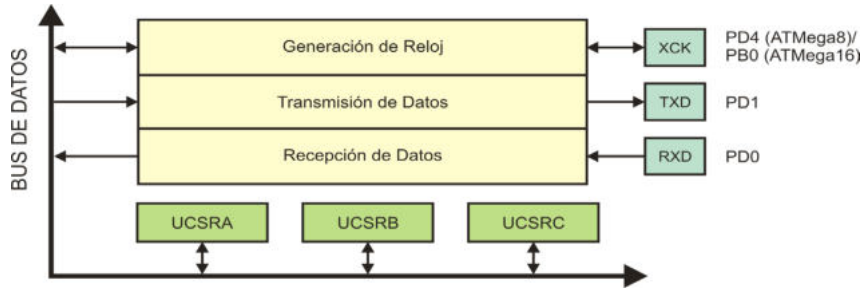


Figura 6.3 Organización de la USART

Los bloques de transmisión y recepción son independientes entre sí. El bloque generador de reloj proporciona las señales de sincronización a los otros 2 bloques.

6.1.1.1 Generación de Reloj y Modos de Operación

El bloque para la generación del reloj proporciona 2 señales de salida, como se muestra en la figura 6.4. La señal CLK1 es el reloj utilizado para transmisión/recepción asíncrona o bien para transmisión síncrona. La señal CLK2 es el reloj utilizado para recepción síncrona.

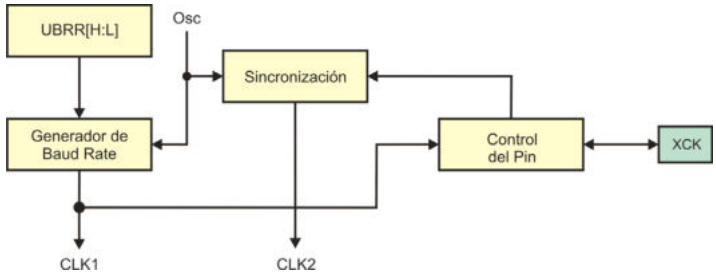


Figura 6.4 Bloque para la generación del reloj de la USART

El registro **UBRR** es la base para la generación de la señal CLK1 (**UBRR**, *USART Baud Rate Register*). Se compone de 2 registros de 8 bits, aunque sólo 12 de los 16 bits disponibles son utilizados.

El Generador de *Baud Rate* incluye un contador descendente cuyo valor máximo es tomado del registro UBRR, el contador recarga su valor máximo cuando llega a 0 o cuando se escribe en **UBRR[L]**. Una señal interna es conmutada cada vez que el contador alcanza un 0, por ello, la frecuencia base para CLK1 es la frecuencia del oscilador dividida entre **UBRR** + 1.

Dependiendo del modo de operación de la USART, esta señal interna es dividida entre 2, 8 ó 16. La USART soporta 4 modos de operación:

- Normal asíncrono: En el cual, la frecuencia de CLK1 es la frecuencia base entre 16.
- Asíncrono a doble velocidad: La frecuencia de CLK1 es la frecuencia base entre 8, el doble del modo normal.
- Síncrono como maestro: La frecuencia de CLK1 es la frecuencia base entre 2, se pueden alcanzar velocidades más altas al transmitir la señal de reloj.
- Síncrono como esclavo: En este caso, el microcontrolador está funcionando como un receptor síncrono, por lo que también recibe la señal de reloj en XCK. La señal externa debe sincronizarse con el oscilador interno. También se puede configurar el flanco en el que se obtienen los datos. La señal CLK2 queda disponible como el reloj para recepción síncrona.

La selección del modo de operación se realiza en los registros de configuración y estado de la USART. Específicamente, el bit **UMSEL** (bit 6 del registro **UCSRC**) se utiliza para seleccionar entre una operación síncrona (**UMSEL** = '1') y asíncrona (**UMSEL** = '0'). El bit **U2X** (bit 1 del registro **UCSRA**) sirve para seleccionar entre el modo normal asíncrono (**U2X** = '0') y el modo asíncrono a doble velocidad (**U2X** = '1').

En la tabla 6.1 se muestran las relaciones matemáticas para obtener la razón de transmisión (*baud rate*) a partir del valor del registro UBRR, no obstante, en la práctica, primero se determina a qué velocidad se va a transmitir, para después calcular el valor de UBRR. Estas relaciones también están en la tabla 6.1.

Tabla 6.1 Cálculo del Baud Rate

Modo	Baud Rate	Valor de UBRR
Normal asíncrono	$Baud\_Rate = \frac{fosc}{16(UBRR+1)}$	$UBRR = \frac{fosc}{16 \times Baud\_Rate} - 1$
Asíncrono a doble velocidad	$Baud\_Rate = \frac{fosc}{8(UBRR+1)}$	$UBRR = \frac{fosc}{8 \times Baud\_Rate} - 1$
Síncrono maestro	$Baud\_Rate = \frac{fosc}{2(UBRR+1)}$	$UBRR = \frac{fosc}{2 \times Baud\_Rate} - 1$

Debe tomarse en cuenta que al calcular el valor de **UBRR** se obtiene un número real, por ello, al redondear a entero se produce un error en el *Baud Rate*. El porcentaje de error se puede cuantificar con la expresión:

$$Error\ [\%] = \left(\frac{Baud\_Rate\_Real}{Baud\_Rate\_esperado} - 1\right) \times 100\ \%$$

En el ejemplo 6.1 se muestra cómo estimar el error en la razón de transmisión, un error en el rango de ± 5 % no afecta la comunicación.

Ejemplo 6.1 Se planea una transmisión asíncrona a 9600 bps con un oscilador de 1 MHz. Calcular el valor del registro **UBRR** y obtener el porcentaje de error, para una transmisión asíncrona normal y a doble velocidad.

En el modo normal asíncrono se tiene que:

$$UBRR = \frac{fosc}{16 \times Baud\_Rate} - 1 = \frac{1\ MHz}{16 \times 9600} - 1 = 5.51$$

Redondeando **UBRR** a 5, se calcula el *baud rate*:

$$Baud\_Rate = \frac{fosc}{16(UBRR+1)} = \frac{1\ MHz}{16\ (5+1)} = 10416.66\ bps$$

El porcentaje de error generado es de:

$$Error\ [\%] = \left(\frac{Baud\_Rate\_Real}{Baud\_Rate\_esperado} - 1\right) \times 100\ \% = \left(\frac{10416.66}{9600} - 1\right) \times 100\ \% = 8.5\ \%$$

Para el modo asíncrono a doble velocidad se tienen los siguientes resultados:

$$UBRR = \frac{1\ MHz}{8 \times 9600} - 1 = 12.02$$

Redondeando **UBRR** a 12, se tiene que:

$$Baud\_Rate = \frac{1\ MHz}{8\ (12 + 1)} = 9615.38\ bps$$

El porcentaje de error generado es de:

$$Error\ [\%] = \left(\frac{9615.38}{9600} - 1\right) \times 100\ \% = 0.16\ \%$$

El error se reduce drásticamente al usar el modo a doble velocidad.

En la tabla 6.2 se muestran algunos valores a emplear en el registro **UBRR** para razones de transmisión típicas, con diferentes osciladores y sus correspondientes porcentajes de error.

Tabla 6.2 Razones de transmisión típicas

Baud Rate (bps)	fosc = 1.0 MHz				fosc = 1.8432 MHz				fosc = 2.0 MHz			
	Normal		Doble Vel.		Normal		Doble Vel.		Normal		Doble Vel.	
	UBRR	Error	UBRR	Error	UBRR	Error	UBRR	Error	UBRR	Error	UBRR	Error
1200	51	0.2 %	103	0.2 %	94	0.0 %	191	0.0 %	103	0.2 %	207	0.2 %
2400	25	0.2 %	51	0.2 %	47	0.0 %	94	0.0 %	51	0.2 %	103	0.2 %
4800	12	0.2 %	25	0.2 %	23	0.0 %	47	0.0 %	25	0.2 %	51	0.2 %
9600	5	8.5 %	12	0.2 %	11	0.0 %	23	0.0 %	12	0.2 %	25	0.2 %
19200	2	8.5 %	5	8.5 %	5	0.0 %	11	0.0 %	5	8.5 %	12	0.2 %
115200	-	-	0	8.5 %	0	0.0 %	1	0.0 %	0	8.5 %	1	8.5 %

En el modo síncrono esclavo la USART recibe la señal de reloj en la terminal XCK, esta señal es sincronizada con el oscilador interno, requiriendo algunos ciclos de reloj para ello. Por esta razón, la frecuencia de la señal de reloj externa debe estar acotada por:

$$f_{XCK} = \frac{f_{osc}}{4}$$

En operaciones síncronas se debe definir el flanco de reloj en el cual son ajustados los datos (cuando se está transmitiendo) o en el que son muestreados (cuando se está recibiendo), esto se determina con el bit **UCPOL** (bit 0 del registro **UCSRC**). En la tabla 6.3 se muestran los flancos de activación (polaridad), dependiendo del valor del bit **UCPOL** (*Clock Polarity*).

Tabla 6.3 Polaridad en la transmisión/recepción síncrona

UCPOL	Ajuste TXD	Muestreo RXD
0	Flanco de subida	Flanco de bajada
1	Flanco de bajada	Flanco de subida

6.1.1.2 Transmisión de Datos

La figura 6.5 muestra la organización del bloque para la transmisión, donde se observa que los datos a ser transmitidos deben ser escritos en el Registro de Datos de la USART (**UDR**, *USART Data Register*). Con **UDR** realmente se hace referencia a dos registros ubicados en la misma dirección, un registro sólo de escritura para transmitir datos y un registro sólo de lectura para recibirlos.

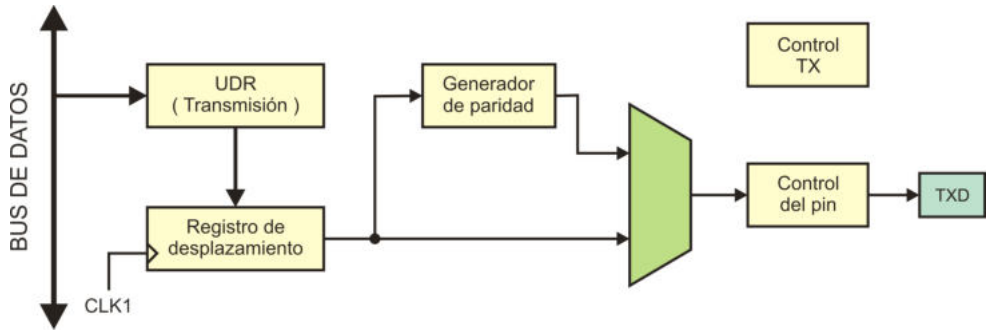


Figura 6.5 Bloque para la transmisión de datos

El bloque de transmisión de datos requiere de una habilitación, la cual se consigue poniendo en alto al bit **TXE** (bit 3 del registro **UCSRB**). La transmisión de un dato inicia cuando éste se escribe en el registro **UDR**, previamente se deben configurar los parámetros: velocidad, número de bits por dato, bit de paridad y número de bits de paro. Un registro de desplazamiento se encarga de la conversión de paralelo a serie.

El fin de la transmisión de un dato se indica con la puesta en alto de la bandera **TXC** (bit 6 del registro **UCSRA**), esta bandera puede sondearse por software o se puede configurar al recurso para que genere una interrupción. Si se utiliza sondeo, la bandera se limpia al escribirle un 1 lógico. No es posible escribir en el registro **UDR**

mientras hay una transmisión en proceso, por ello, además del uso de la bandera **TXC** es posible usar la bandera **UDRE** (bit 5 del registro **UCSRA**. **UDRE,** *USART Data Register Empty*). La bandera **UDRE** está en alto cuando el buffer transmisor (registro de desplazamiento) está vacío, por lo tanto, indica que es posible iniciar con una nueva transmisión. También puede sondearse por software o configurar el recurso para que genere una interrupción. El estado de la bandera **UDRE** cambia automáticamente, dependiendo de la actividad que ocurra en el registro **UDR**.

En la práctica únicamente se emplea una de las 2 banderas, **TXC** o **UDRE**. Usar ambas puede complicar la estructura de un programa, porque indican eventos similares cuando se transmite una sucesión de datos. Al finalizar con la transmisión de un dato, el buffer transmisor queda vacío.

El bit de paridad se genera por hardware conforme los datos se van transmitiendo. Éste queda disponible para ser transmitido después de los bits del dato.

6.1.1.3 Recepción de Datos

En la figura 6.6 se muestra la organización del bloque para la recepción de datos. También contiene un Registro de Datos de la USART (**UDR**), aunque en este caso es un registro sólo de lectura empleado para recibir datos seriales por medio de la USART. La recepción serial se basa en un registro de desplazamiento que realiza la conversión de serie a paralelo. El receptor debe habilitarse para que en cualquier momento pueda recibir un dato, la habilitación se realiza con la puesta en alto del bit **RXE** (bit 4 del registro **UCSRB**).

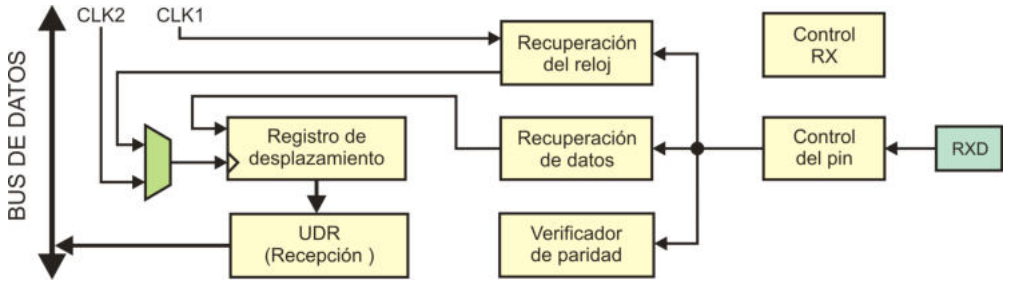


Figura 6.6 Bloque para la recepción de datos

El dato recibido es ubicado en el registro **UDR**, la indicación de que hay un dato disponible se realiza con la puesta en alto de la bandera **RXC** (Bit 7 del registro **UCSRA**). Esta bandera puede sondearse por software o configurar al recurso para que genere una interrupción. La lectura del registro **UDR** limpia a la bandera y libera al registro, quedando disponible para recibir un nuevo dato.

El cálculo del bit de paridad y su validación, comparándola con el valor del bit de paridad recibido, se realizan por hardware. Si existe un error, éste se indica con la puesta en alto de la bandera **PE** (bit 2 del registro **UCSRA**. **PE**, *Parity Error*).

El hardware es capaz de indicar la ocurrencia de otros 2 errores durante la recepción, con la puesta en alto de banderas. Un error de marco (*frame error*) se debe a la existencia de un bit de paro con un nivel bajo y se indica con la bandera **FE** (bit 4 del registro **UCSRA**). El otro error es debido a un exceso de datos por recepción, ocurre cuando el registro **UDR** tiene un dato listo para su lectura, el registro de desplazamiento contiene otro dato y hay un nuevo bit de inicio, este error se indica con la bandera **DOR** (bit 3 del registro **UCSRA**. **DOR**, *Data Over Run*).

6.1.2 Transmisión y Recepción de Datos de 9 Bits

Las transmisiones y recepciones se realizan por medio del registro **UDR**, el cual físicamente está compuesto por 2 registros de 8 bits, uno sólo de escritura para transmisión y otro sólo de lectura para recepción.

Sin embargo, la USART permite el envío y recepción de datos de 9 bits, para ello se emplean 2 bits del registro **UCSRB**. El bit **TXB8** (bit 0 del registro **UCSRB**) es el 9º en una transmisión de 9 bits y el bit **RXB8** (bit 1 del registro **UCSRB**) es el 9º durante una recepción de 9 bits. Para una transmisión primero debe definirse el valor de **TXB8**, porque la transmisión inicia cuando se escriba en el registro **UDR**. Y en una recepción primero debe leerse el valor de **RXB8** y luego al registro **UDR**. Tanto **TXB8** como **RXB8** corresponden con el bit más significativo en datos de 9 bits.

6.1.3 Comunicación entre Múltiples Microcontroladores

La USART de los AVR permite una comunicación entre más de 2 microcontroladores, bajo un esquema maestro-esclavos, pudiendo conectar hasta 256 esclavos. En la figura 6.7 se ilustra esta organización, los esclavos deben tener una dirección diferente, entre 0 y 255. Estas direcciones se pueden manejar como constantes en memoria *flash* o configurarse con interruptores en uno de los puertos del microcontrolador.

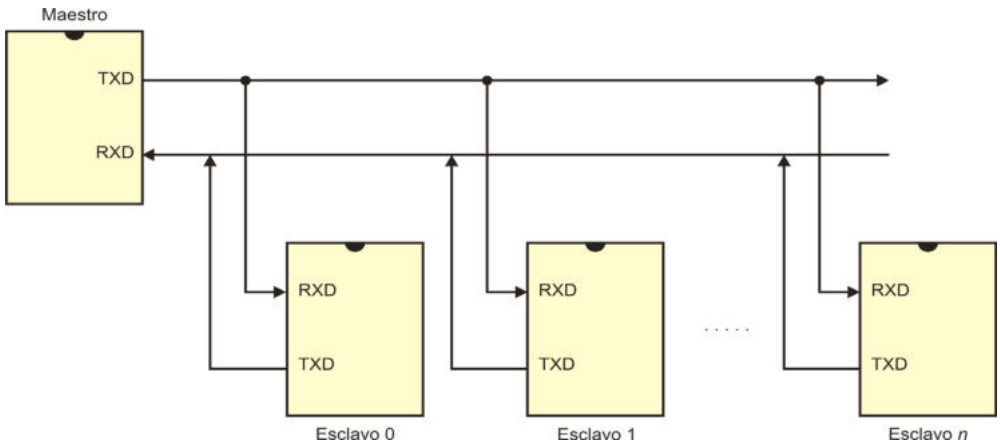


Figura 6.7 Organización maestro-esclavos para comunicar a múltiples microcontroladores

La comunicación utiliza un formato de 9 bits, donde el 9º bit (transmitido en **TXB8** o recibido en **RXB8**) sirve para distinguir entre dos tipos de información, con 0 se refiere a una trama de datos y con 1 a una trama de dirección.

La comunicación entre múltiples MCUs requiere el empleo del bit **MPCM** (bit 0 del registro **UCSRA**. **MPCM**, *Multiprocessor Comunication Mode*). Este bit es utilizado por los esclavos, si está en alto, únicamente pueden recibir tramas de dirección. Su puesta en bajo les permite recibir tramas de datos.

La comunicación se realiza de la siguiente manera:

1. Los esclavos habilitan el modo de comunicación entre multiprocesadores, con **MPCM** = 1. De manera que únicamente pueden recibir tramas de dirección (9º bit en 1).

2. El maestro envía la dirección del esclavo con el que va a interactuar, esta dirección es recibida por todos los esclavos.

3. Cada esclavo lee su registro **UDR** y lo compara con su dirección para determinar si ha sido seleccionado. El esclavo seleccionado limpia su bit **MPCM** en el registro **UCSRA**, quedando disponible para recibir datos.

4. El maestro y el esclavo seleccionado realizan el intercambio de datos (9º bit en 0), el cual pasa desapercibido por los otros esclavos, porque aún tienen su bit **MPCM** en alto.

5. Cuando el diálogo concluye, el esclavo seleccionado debe poner en alto su bit **MPCM**, quedando como los demás esclavos, en espera de que el maestro solicite su atención.

6.1.4 Registros para el Manejo de la USART

El registro **UDR** sirve para el manejo de datos en la USART. Físicamente se compone de 2 registros de 8 bits, uno sólo de lectura para la recepción de datos y otro sólo de escritura para transmisión, aunque ambos emplean la misma dirección.

	7	6	5	4	3	2	1	0	
0x0C	Byte recibido								**UDR** (Lectura)
	Byte a transmitir								**UDR** (Escritura)

Para definir la razón de transmisión (*baud rate*) se utiliza al registro **UBRR**, éste es un registro de 12 bits, por lo tanto, se integra por 2 registros de 8 bits: **UBRRL** para la parte baja y **UBRRH** para la parte alta:

	7	6	5	4	3	2	1	0	
0x20	URSEL	-	-	-	UBRR11	UBRR10	UBRR9	UBRR8	**UBRRH**
0x09	UBRR7	UBRR6	UBRR5	UBRR4	UBRR3	UBRR2	UBRR1	UBRR0	**UBRRL**

El registro **UBRRH** comparte su dirección (0x20) con el registro **UCSRC**, el cual es uno de los registros de control. Para ello, debe utilizarse al bit **URSEL** como un selector para escrituras. Con **URSEL** = 0 se escribe en **UBRRH** y con **URSEL** = 1 se escribe en **UCSRC**.

Para las lecturas, puesto que **UBRRH** y **UCSRC** hacen referencia a la misma dirección, una lectura aislada proporciona el valor de **UBRRH**. Para obtener el valor de **UCSRC** deben realizarse 2 lecturas consecutivas.

Los registros de control y estado de la USART son 3: **UCSRA**, **UCSRB** y **UCSRC**.

En el registro **UCSRA**, los 6 bits más significativos corresponden con las banderas de estado de la USART. Los bits de **UCSRA** son:

	7	6	5	4	3	2	1	0	
0x0B	RXC	TXC	UDRE	FE	DOR	PE	U2X	MPCM	**UCSRA**

- **Bit 7 – RXC: Bandera de recepción completa**

  Indica que hay un dato disponible por recepción. Puede generar una interrupción. Se limpia al leer al registro **UDR**.

- **Bit 6 – TXC: Bandera de transmisión completa**

  Indica que se ha concluido con la transmisión de un dato y por lo tanto, el buffer de salida está vacío. Puede generar una interrupción, con su atención se limpia automáticamente a la bandera. También puede limpiarse con la escritura de un 1.

- **Bit 5 – UDRE: Bandera de UDR vacío**

  Indica que el registro **UDR** de transmisión está vacío y por lo tanto, es posible transmitir un dato. También puede generar una interrupción. Su estado se ajusta automáticamente, de acuerdo con la actividad en el registro **UDR**.

- **Bit 4 – FE: Bandera de error de marco**

  Esta bandera se pone en alto cuando el bit de paro en el buffer receptor es 0. Se mantiene con ese valor hasta que se lee al registro **UDR**.

- **Bit 3 – DOR: Bandera de error por exceso de datos**

  Un exceso de datos significa que el registro receptor (**UDR**) tiene un dato disponible, el registro de desplazamiento contiene otro dato y se tiene un nuevo bit de inicio. Cuando la bandera es puesta en alto, mantiene su valor hasta que el registro **UDR** es leído.

- **Bit 2 – PE: Bandera de error por paridad**

  Se pone en alto cuando el bit de paridad generado no coincide con el bit de paridad recibido. Se mantiene con ese valor hasta que el registro **UDR** es leído.

- **Bit 1 – U2X: Dobla la velocidad de transmisión**

  En operaciones asíncronas, este bit debe ponerse en alto para cambiar del modo normal asíncrono, al modo asíncrono a doble velocidad.

- **Bit 0 – MPCM: Modo de comunicación entre multiprocesadores**

  Habilita un modo de comunicación entre multiprocesadores, bajo un esquema maestro-esclavos. El bit **MPCM** es empleado por los esclavos para que puedan recibir tramas de dirección o tramas de datos.

Los 3 bits más significativos de **UCSRB** son habilitadores para generar interrupciones, coinciden en orden con las banderas de **UCSRA**. Los bits del registro **UCSRB** son:

	7	6	5	4	3	2	1	0	
0x0A	RXCIE	TXCIE	UDRIE	RXEN	TXEN	UCSZ2	RXB8	TXB8	**UCSRB**

- **Bit 7 – RXCIE: Habilitador de interrupción por recepción completa**

  Si es puesto en alto, y el habilitador global de interrupciones también, se produce una interrupción cuando se reciba un dato (**RXC** en alto).

- **Bit 6 – TXCIE: Habilitador de interrupción por transmisión completa**

  Si es puesto en alto, y el habilitador global de interrupciones también, se produce una interrupción cuando finalice la transmisión de un dato (**TXC** en alto).

- **Bit 5 – UDRIE: Habilitador de interrupción por buffer de transmisión vacío**

  Si es puesto en alto, y el habilitador global de interrupciones también, se produce una interrupción cuando el buffer transmisor esté vacío (**UDRE** en alto).

- **Bit 4 – RXEN: Habilitador del receptor**

  Debe ser puesto en alto para poder recibir datos a través de la USART, con esto, la terminal RXD ya no se puede utilizar como entrada o salida de propósito general.

- **Bit 3 – TXEN: Habilitador del transmisor**

  Debe ser puesto en alto para poder transmitir datos a través de la USART, con esto, la terminal TXD ya no se puede utilizar como entrada o salida de propósito general.

- **Bit 2 – UCSZ2: Bit 2 para definir el tamaño de los datos**

  Junto con otros 2 bits del registro **UCSRC** permiten definir el tamaño de los datos a enviar o recibir. En la tabla 6.4 se definen los diferentes tamaños, de acuerdo con el valor de **UCSZ[2:0]**.

- **Bit 1 – RXB8: 9º bit recibido**

  Durante la recepción de datos de 9 bits, los 8 bits menos significativos se reciben en el registro **UDR**, en **RXB8** se recibe al bit más significativo.

- **Bit 0 – TXB8: 9º bit a ser transmitido**

  Para transmitir datos de 9 bits, los 8 bits menos significativos deben ubicarse en el registro **UDR**, en **TXB8** debe colocarse al bit más significativo.

El registro **UCSRC** comparte su dirección con **UBRRH** (0x20). El bit **URSEL** se utiliza para seleccionar el registro a escribir. Para leer a **UCSRC** deben realizarse 2 lecturas consecutivas, una lectura aislada proporciona el valor de **UBRRH**. Los bits del registro **UCSRC** son:

	7	6	5	4	3	2	1	0	
0x20	URSEL	UMSEL	UPM1	UPM0	USBS	UCSZ1	UCSZ0	UCPOL	**UCSRC**

- **Bit 7 – URSEL: Selector para escribir en UBRRH o UCSRC**

  Con **URSEL** = 0 se escribe en **UBRRH** y con **URSEL** = 1 se escribe en **UCSRC**.

- **Bit 6 – UMSEL: Selector del modo de la USART**

  Selecciona entre una operación síncrona o asíncrona. Con **UMSEL** = 0 la operación de la USART es asíncrona. Con **UMSEL** = 1 su operación es síncrona.

- **Bits 5 y 4 – UMP[1:0]: Bits para la configuración de la paridad**

  Con las combinaciones de estos bits se determina el uso del bit de paridad y se define su tipo. En la tabla 6.5 se muestran las diferentes opciones.

- **Bit 3 – USBS: Selector del número de bits de paro**

  Entre dato y dato puede incluirse sólo 1 bit de paro (**USBS** = 0) o pueden ser 2 bits de paro (**USBS** = 1).

- **Bits 2 y 1 – UCSZ[1:0]: Bits 1 y 0 para definir el tamaño de los datos**

  Junto con el bit **UCSZ2** del registro **UCSRB** definen el tamaño de los datos a enviar o recibir. En la tabla 6.4 se muestran los diferentes tamaños, de acuerdo con el valor de **UCSZ[2:0]**.

- **Bit 0 – UCPOL: Polaridad del reloj**

  En operaciones síncronas, **UCPOL** determina el flanco de reloj en el cual se ajustan los datos (cuando se está transmitiendo) o en el que son muestreados (cuando se está recibiendo). En la tabla 6.3 se mostraron los flancos de activación (polaridad), dependiendo del valor del bit **UCPOL**.

Tabla 6.4 Opciones para el tamaño de los datos

UCSZ2	UCSZ1	UCSZ0	Tamaño de los datos
0	0	0	5 bits
0	0	1	6 bits
0	1	0	7 bits
0	1	1	8 bits
1	0	0	Reservado
1	0	1	Reservado
1	1	0	Reservado
1	1	1	9 bits

Tabla 6.5 Activación del bit de paridad y sus diferentes tipos

UPM1	UPM0	Configuración
0	0	Sin bit de paridad
0	1	Reservado
1	0	Paridad Par
1	1	Paridad Impar

6.1.5 Ejemplos de Uso de la USART

Los siguientes ejemplos muestran cómo configurar a la USART y las rutinas básicas para transmisión y recepción, sondeando las banderas o utilizando interrupciones. Los primeros 4 ejemplos son realizados en ensamblador, el 5º es realizado en ensamblador y en lenguaje C, y el 6º sólo en lenguaje C.

Ejemplo 6.1 Escriba una rutina que configure a la USART para una comunicación asíncrona de 8 bits, sin paridad, a 9600 bps y sin manejo de interrupciones. Suponga que el microcontrolador opera a 1 MHz.

```
conf_USART:
       CLR    R16           ; Valor de recarga para una velocidad de 9600 bps
       OUT    UBRRH, R16
       LDI    R16, 12
       OUT    UBRRL, R16

       LDI    R16, 0x02     ; Modo asíncrono a doble velocidad
       OUT    UCSRA, R16

       LDI    R16, 0x18     ; Habilita receptor y transmisor
       OUT    UCSRB, R16    ; sin interrupciones

       LDI    R16, 0x86     ; 1 bit de paro, sin paridad y datos de 8 bits
       OUT    UCSRC, R16
       RET
```

Ejemplo 6.2 Realice una rutina que espere hasta recibir un dato por el puerto serie y lo coloque en R17.

```
Recibe:
       SBIS   UCSRA, RXC          ; Espera a que haya un dato disponible
       RJMP   Recibe              ; por recepción
       IN     R17, UDR            ; Lo deja en R17
       RET
```

Ejemplo 6.3 Escriba una rutina que transmita un carácter ubicado en R17, la rutina se debe asegurar que el registro UDR de transmisión está vacío.

```
Transmite:
       SBIS   UCSRA, UDRE         ; Garantiza que el buffer transmisor
       RJMP   Transmite           ; está vacío
       OUT    UDR, R17            ; Transmite el contenido de R17
       RET
```

Ejemplo 6.4 Apoyado en las rutinas de los 3 ejemplos anteriores, escriba un programa “eco” que transmita cada dato recibido.

```
        .include  <m8def.inc>

        LDI     R16, 0x04           ; Ubica al apuntador de pila
        LDI     R17, 0x5F
        OUT     SPH, R16
        OUT     SPL, R17

        RCALL   conf_USART          ; Configura la USART
loop:
        RCALl   Recibe              ; Espera a recibir un dato
        RCALL   Transmite           ; Transmite el dato recibido
        RJMP    loop
```

Ejemplo 6.5 Repita el ejemplo 6.4 usando interrupciones.

Se requiere de la interrupción por recepción para que en cualquier momento el MCU pueda recibir un dato, cuando esto sucede, se asegura que el buffer de transmisión está vacío para iniciar con el envío, el programa principal permanece ocioso. La solución en lenguaje ensamblador es:

```
        .include  <m8def.inc>

        .org    0x000
        RJMP    inicio

        .org    0x00B
        RJMP    ISR_USART_RXC

        .org    0x013               ; Después de los vectores de interrupciones
inicio:
        LDI     R16, 0x04           ; Ubica al apuntador de pila
        LDI     R17, 0x5F
        OUT     SPH, R16
        OUT     SPL, R17
        RCALL   conf_USART          ; Configura la USART
        SEI                         ; Habilitador global de interrupciones
loop:
        RJMP    loop                ; Ocioso, la interrupción recibe el dato

conf_USART:
        CLR     R16                 ; Para una velocidad de 9600 bps
        OUT     UBRRH, R16
        LDI     R16, 12
        OUT     UBRRL, R16
        LDI     R16, 0x02           ; Modo asíncrono a doble velocidad
        OUT     UCSRA, R16
        LDI     R16, 0x98           ; Habilita receptor y transmisor
        OUT     UCSRB, R16          ; con interrupción por recepción
        LDI     R16, 0x86           ; 1 bit de paro, sin paridad y datos de 8 bits
        OUT     UCSRC, R16
        RET
```

```
ISR_USART_RXC:
       IN     R17, UDR              ; Recibe un dato serial
Buffer_vacio:
       SBIS   UCSRA, UDRE           ; Garantiza que el buffer está vacío
       RJMP   Buffer_vacio
       OUT    UDR, R17              ; Transmite el contenido de R17
       RETI
```

La solución en lenguaje C es:

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

void   conf_USART();                // Función para configurar la USART

ISR(USART_RXC_vect) {
unsigned  char  dato;
       dato = UDR;
       while(!( UCSRA & 1 << UDRE ));     // Garantiza que el buffer está vacío
       UDR = dato;                        // Transmite el dato recibido
}

int    main() {
       conf_USART();                // Configura a la USART
       sei();                       // Habilitador global de interrupciones
       while(1)                     // En el lazo infinito permanece ocioso
              asm("nop");
}

void  conf_USART() {

       UBRRH = 0;                   // Para una velocidad de 9600 bps
       UBRRL = 12;
       UCSRA = 0x02;                // Modo asíncrono a doble velocidad
       UCSRB = 0x98;                // Habilita receptor y transmisor
                                    // con interrupción por recepción
       UCSRC = 0x86;                // 1 bit de paro, sin paridad y datos
                                    // de 8 bits
}
```

Las soluciones son muy similares, porque realmente es el recurso el que se encarga de la recepción y transmisión.

Ejemplo 6.6 Escriba un programa que envíe un carácter ASCII imprimible por el puerto serie, cada vez que se presione un botón. Los caracteres ASCII imprimibles están en el rango de 0x20 a 0x7E. Además, en cualquier momento puede arribar serialmente un dato y éste debe mostrarse en el puerto B. En la figura 6.8 se muestra el hardware requerido.

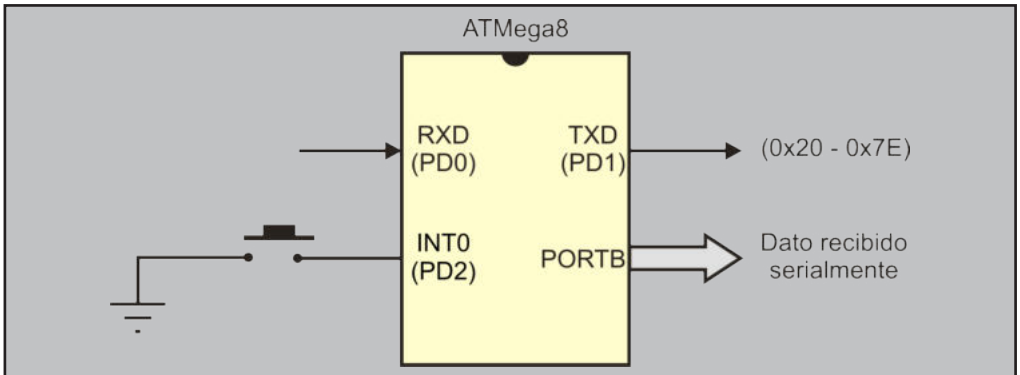


Figura 6.8 Hardware para el problema de comunicación serial

Se utilizan 2 interrupciones: por recepción, para ubicar el dato recibido en el puerto B y una interrupción externa, que prepara y envía el siguiente carácter imprimible, asegurando que no hay una transmisión en proceso. El código en lenguaje C es:

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

// Variables globales
unsigned  char dato = 0x20;       // El dato a enviar
void   conf_USART();              // Función de configuración de la
USART

ISR(USART_RXC_vect) {             // ISR de recepción serial
      PORTB = UDR;                // El dato recibido se muestra en el
puerto B
}

ISR(INT0_vect) {                          // ISR de la INT0
      while(!( UCSRA & 1 << UDRE ));      // Garantiza que el buffer
                                          // está vacío
      UDR = dato;                         // Transmite el dato actual
      dato++;                             // Prepara el dato siguiente
      if( dato == 0x7F )                  // Si llegó al límite
            dato = 0x20;                  // Inicia nuevamente
}

int    main() {

      conf_USART();               // Configura a la USART
      DDRD = 0b00000010;          // sólo TXD es salida en PORTD
      PORTD = 0b00000100;         // Resistor de Pull-Up en INT0
      DDRB = 0xFF;                // Puerto B como salida
      MCUCR = 0B00000010;         // Configura INT0 por flanco de bajada
      GICR = 0B01000000;          // Habilita la INT0
      sei();                      // Habilitador global de interrupciones
      while(1)                    // En el lazo infinito permanece ocioso
            asm("nop");
```

```
}

void  conf_USART() {

        UBRRH = 0;                  // Para una velocidad de 9600 bps
        UBRRL = 12;
        UCSRA = 0x02;               // Modo asíncrono a doble velocidad
        UCSRB = 0x98;               // Habilita receptor y transmisor
                                    // con interrupción por recepción
        UCSRC = 0x86;               // 1 bit de paro, sin paridad y datos de 8 bits
}
```

El ejemplo anterior muestra cómo los recursos de transmisión y recepción son independientes.

6.2 Comunicación Serial por SPI

La Interfaz Serial Periférica (SPI, *Serial Peripheral Interface*) establece un protocolo estándar de comunicaciones, usado para transferir paquetes de información de 8 bits entre circuitos integrados. El protocolo SPI permite una transferencia síncrona de datos a muy alta velocidad entre un microcontrolador y dispositivos periféricos o entre microcontroladores.

En la conexión de dos dispositivos por SPI, siempre ocurre que uno funciona como Maestro y otro como Esclavo, aunque es posible el manejo de un sistema con múltiples Esclavos. El Maestro es quien determina cuándo enviar los datos y quien genera la señal de reloj. El Esclavo no puede enviar datos por sí mismo y tampoco es capaz de generar la señal de reloj. Esto significa que el Maestro tiene que enviar datos al Esclavo para obtener información de él.

En la figura 6.9 se muestra una conexión entre dos dispositivos vía SPI, esta interfaz no es propia de los microcontroladores, también puede incluirse en memorias, sensores, ADCs, DACs, etc., estos dispositivos usualmente funcionan como Esclavos. Puede verse que la base del protocolo SPI son registros de desplazamiento con sus correspondientes circuitos de control.

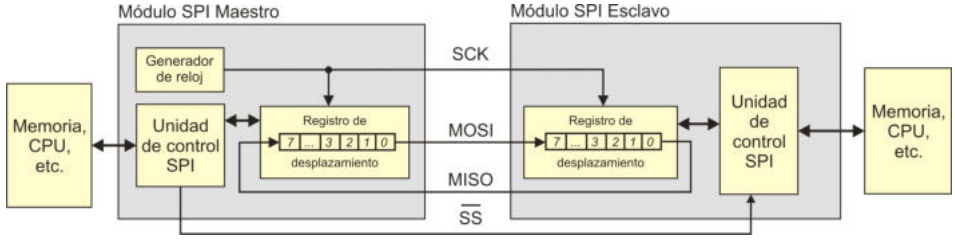


Figura 6.9 Conexión entre 2 dispositivos vía SPI

Las señales que intervienen en la comunicación son:

- **MOSI** (*Master Output, Slave* Input): Señal de salida del Maestro y entrada en el Esclavo. Típicamente el envío de información inicia con el bit menos significativo.

- **MISO** *(Master Input, Slave Output*): Señal de entrada al Maestro y salida en el Esclavo. Proporciona el mecanismo para que el Esclavo pueda dar respuesta al Maestro.

- **SCK** (*Shift Clock*): Es la señal de reloj para sincronizar la comunicación. Es generada por el Maestro.

- **SS** (*Slave Select*): Esta señal es útil para el manejo de múltiples Esclavos, para ello, deben realizarse los arreglos necesarios en el hardware, de manera que sólo se habilite un Esclavo en cada transmisión. Esta señal generalmente es activa en un nivel bajo.

El envío y recepción se realizan en forma simultánea, mientras el Maestro envía datos por MOSI recibe una respuesta del Esclavo por MISO. Sincronizando el envío y recepción con la señal de reloj, es decir, en el mismo ciclo de reloj se trasmite y recibe un bit. Debido a esto, los registros de desplazamiento de 8 bits pueden ser considerados como un registro de desplazamiento circular de 16 bits. Esto significa que después de 8 pulsos de reloj, el Maestro y el Esclavo han intercambiado un dato de 8 bits.

6.2.1 Organización de la Interfaz SPI en los AVR

Bajo el protocolo SPI, un microcontrolador AVR puede funcionar como Maestro o como Esclavo, de acuerdo con los requerimientos de la aplicación. En la figura 6.10 se muestra la organización del hardware para la interfaz SPI. El recurso se habilita con la puesta en alto del bit **SPE** del registro de control (**SPCR**, *SPI Control Register*), ninguna operación con la interfaz SPI es posible sin esta habilitación. La configuración como Maestro se realiza con la puesta en alto del bit **MSTR** en el registro **SPCR**, en caso contrario, el dispositivo funciona como Esclavo.

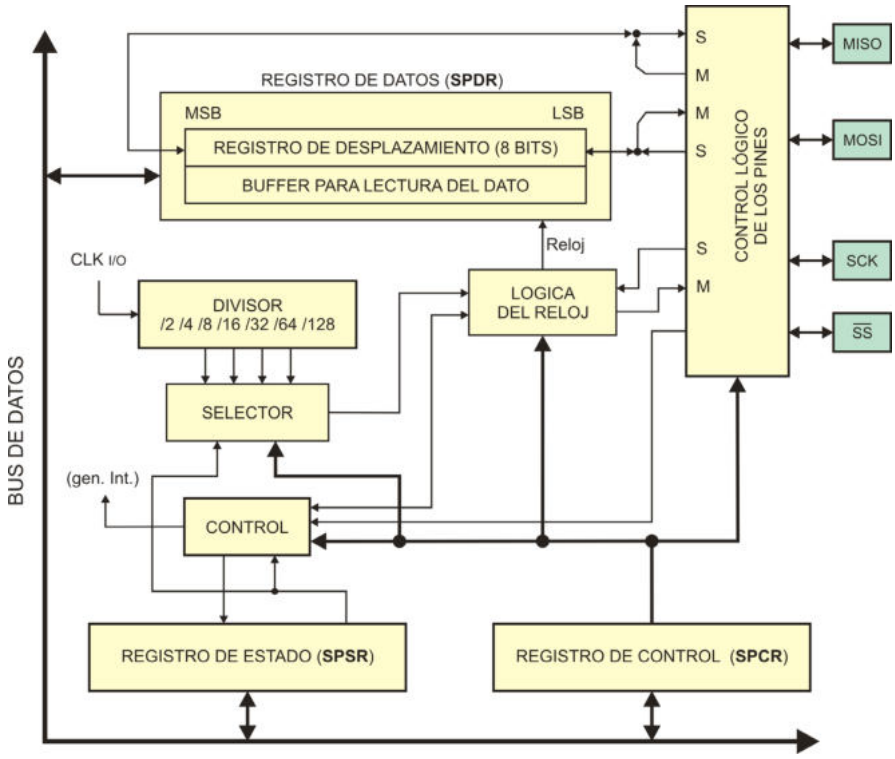


Figura 6.10 Organización de la interfaz SPI en los AVR

El modo en el que cada dispositivo funciona determina si las líneas: MISO, MOSI, SCK y SS, son señales de entrada o de salida, ésta es la tarea del módulo para el control lógico de los pines. El flujo de datos se establece de acuerdo con la tabla 6.6, donde se observa que sólo los pines de entrada se configuran automáticamente, los pines de salida tienen que ser configurados por software para evitar daños.

Tabla 6.6 Flujo de datos en la interfaz SPI

PIN	Maestro	Esclavo
MOSI	Definido por el usuario	Entrada
MISO	Entrada	Definido por el usuario
SCK	Definido por el usuario	Entrada
SS	Definido por el usuario	Entrada

La interfaz SPI incluye un buffer simple para la transmisión y un buffer doble para la recepción. El acceso a ambos se realiza por medio del registro de datos (**SPDR**, *SPI Data Register*).

La transmisión de un dato inicia después de escribir en el registro **SPDR**. No se debe escribir un dato nuevo mientras un ciclo de envío-recepción esté en progreso. En los AVR puede elegirse si se envia primero al bit más significativo o al menos significativo, esto se determina con el bit **DORD** (*data order*) del registro **SPCR**.

Los bits recibidos son colocados en el buffer de recepción inmediatamente después de que la transmisión se ha completado. El buffer de recepción tiene que ser leído antes de iniciar con la siguiente transmisión, de lo contrario, se va a perder el dato recibido. El dato del buffer de recepción se obtiene con la lectura del registro **SPDR**.

El fin de una transferencia se indica con la puesta en alto de la bandera **SPIF** en el registro de estado (**SPSR**, *SPI Status Register*). Este evento produce una interrupción si se habilitó la interrupción por transferencia serial completa vía SPI, la cual se habilita con la puesta en alto del bit **SPIE** en el registro **SPCR**, también se requiere la activación del habilitador global de interrupciones (bit **I** del registro **SREG**). Aunque la bandera **SPIF** también puede sondearse por software.

Una colisión de escritura ocurre si se realiza un acceso al registro **SPDR** mientras hay una transferencia en progreso, debido a que se podrían corromper los datos de la transferencia actual. Una colisión de escritura generalmente es el error de un Esclavo, porque desconoce si el Maestro inició con una transferencia. El Maestro sabe si hay una transferencia en progreso, por ello, no debería generar errores de colisión de escritura, no obstante, el hardware puede detectar estos errores tanto en modo Maestro como en modo Esclavo. Se indica con la puesta en alto de la bandera **WCOL**, en el registro **SPSR**. Las banderas **WCOL** y **SPIF** se limpian con la lectura del registro **SPSR** y el acceso al registro **SPDR**.

6.2.2 Modos de Transferencias SPI

Las transferencias son sincronizadas con la señal de reloj (SCK), un bit es transferido en cada ciclo. Para que la interfaz SPI sea compatible con diferentes dispositivos, la sincronización de los datos con el reloj es flexible, el usuario puede definir la polaridad de la señal de reloj y la fase del muestreo de datos. La polaridad se refiere al estado lógico de la señal de reloj mientras no hay transferencias. La fase define si el primer bit es muestreado en el primer flanco de reloj (en fase) o en el flanco siguiente, insertando un retraso al inicio para asegurar que la interfaz SPI receptora está lista.

En el registro **SPCR** se tienen 2 bits para configurar estos parámetros, el bit **CPOL** es para definir la polaridad y con el bit **CPHA** se determina la fase. Con los valores de **CPOL** y **CPHA** se tienen 4 combinaciones que corresponden con los modos de transferencia, éstos se definen en la tabla 6.7 y en la figura 6.11 se muestran los diagramas de tiempo para distinguirlos.

Tabla 6.7 Modos de transferencias SPI

CPOL	CPHA	Modo SPI	Descripción
0	0	0	Espera en bajo, muestrea en fase
0	1	1	Espera en bajo, muestrea con un retraso
1	0	2	Espera en alto, muestrea en fase
1	1	3	Espera en alto, muestrea con un retraso

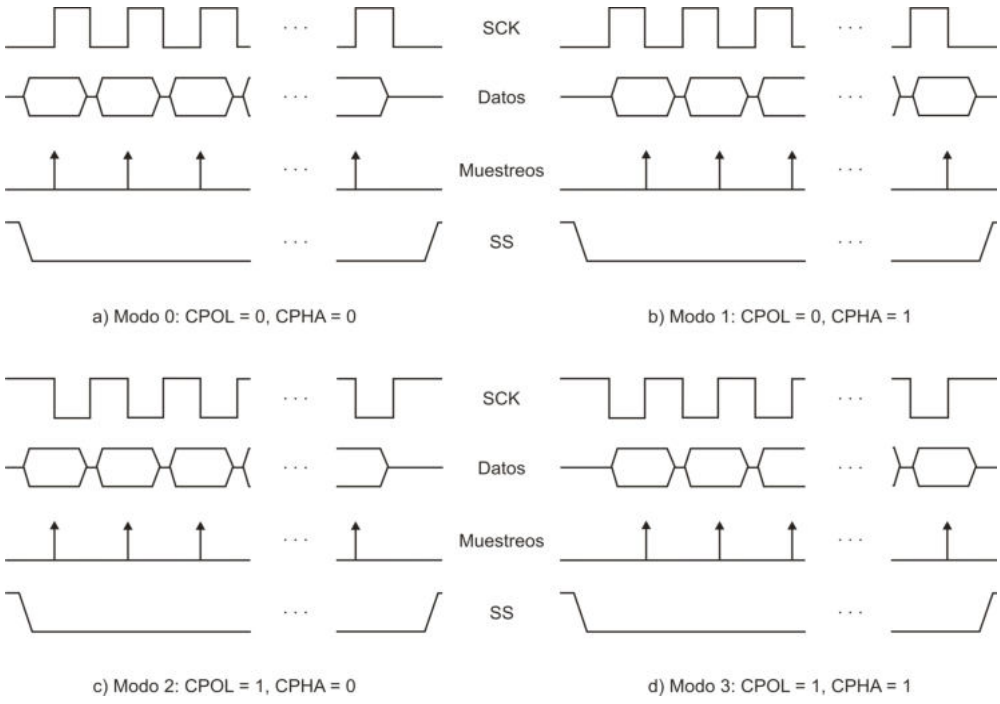


Figura 6.11 Modos de transferencias SPI

6.2.3 Funcionalidad de la Terminal SS

Si el AVR es configurado como Esclavo (bit **MSTR** del registro **SPCR** en bajo), la interfaz SPI se activa cuando existe un nivel bajo en SS. Cuando en la terminal SS se coloca un nivel alto, la interfaz SPI queda pasiva y no reconoce los datos de entrada, si hay datos parcialmente recibidos en el registro de desplazamiento, éstos van a ser desechados. Además, la terminal SS ayuda a mantener al Esclavo sincronizado con el reloj del Maestro, esto también puede notarse en la figura 6.11.

Si el AVR es configurado como Maestro (bit **MSTR** del registro **SPCR** en alto), el usuario debe determinar la dirección de la terminal SS. Como salida, SS es una salida general que no afecta a la interfaz SPI. Típicamente se debería conectar con la terminal SS de un Esclavo, activándolo o desactivándolo por software.

Si SS se configura como entrada, se le debe suministrar un nivel alto para asegurar su operación como Maestro. Si algún periférico introduce un nivel bajo, significa que otro Maestro está intentando seleccionar al MCU como Esclavo para enviarle datos. Para evitar conflictos en las transferencias, en la interfaz SPI automáticamente se realizan las siguientes acciones:

1. El bit **MSTR** del registro **SPCR** es limpiado para que el MCU sea tratado como Esclavo. Al ser un Esclavo, las terminales MOSI y SCK se vuelven entradas.

2. La bandera **SPIF** en el registro **SPSR** es puesta en alto, de manera que si el habilitador de la interrupción y el habilitador global están activos, se va a ejecutar la ISR correspondiente.

Por lo tanto, si la interfaz SPI siempre va a funcionar como Maestro, y existe la posibilidad de que la terminal SS sea llevada a un nivel bajo, en la ISR debe ajustarse el valor del bit **MSTR**, para mantenerlo operando en el modo correcto.

Si una aplicación requiere el manejo de un Maestro y varios Esclavos, otras terminales del Maestro deben funcionar como habilitadoras, conectándose con las terminales SS de cada uno de los Esclavos. Cuando el Maestro requiere intercambiar información con un Esclavo, primero debe habilitarlo por software. En la figura 6.12 se muestra la conexión de un Maestro y 3 Esclavos, el Maestro se basa en un ATMega16 y cada Esclavo es un ATMega8.

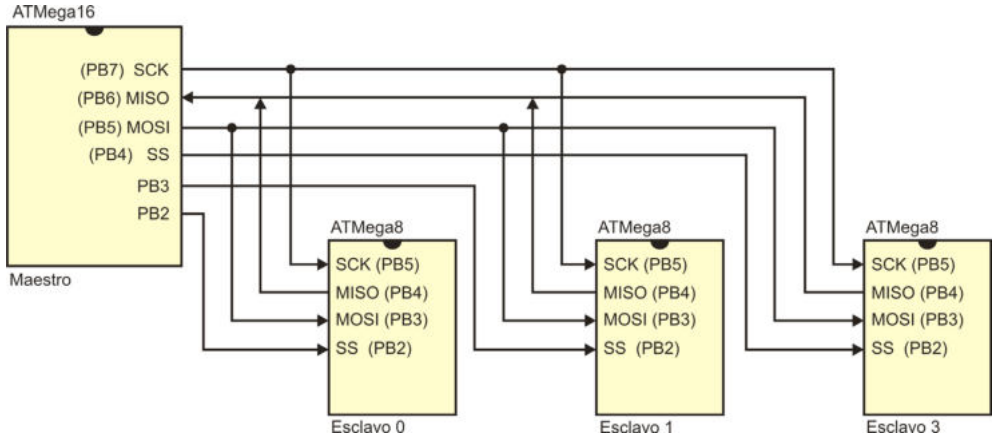
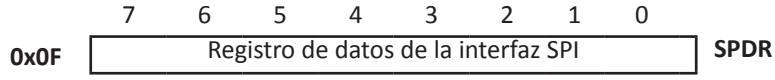


Figura 6.12 Conexión de un Maestro y 3 Esclavos por SPI

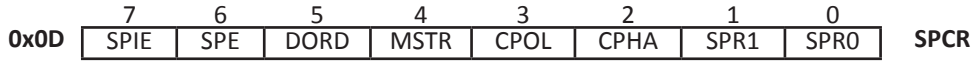
La figura 6.12 ilustra la forma de conectar 3 microcontroladores como Esclavos. En la práctica, los Esclavos suelen ser dispositivos con una tarea específica, como ADCs, DACs, memorias, etc.

6.2.4 Registros para el Manejo de la Interfaz SPI

El registro **SPDR** es el buffer para transmisión y recepción de la interfaz SPI, está conectado directamente con el registro de desplazamiento. Una escritura en **SPDR** da inicio a la transmisión de un dato. Una lectura en **SPDR** proporciona el dato recibido en el registro de desplazamiento.



El registro de control es el **SPCR**, sus bits son:



- **Bit 7 – SPIE: Habilitador de interrupción por SPI**

  Debe estar en alto, junto con el habilitador global, para que la bandera **SPIF** genere una interrupción por transferencia serial completa vía SPI.

- **Bit 6 – SPE: Habilitador de la interfaz SPI**

  Habilita a la interfaz SPI, debe estar en alto para realizar transferencias por esta interfaz.

- **Bit 5 – DORD: Orden de los datos**

  Con un 0 en **DORD**, se transfiere primero al bit más significativo (MSB). Con un 1 se transfiere primero al bit menos significativo (LSB).

- **Bit 4 – MSTR: Habilitador como Maestro**

  Un 1 en **MSTR** habilita a la interfaz como Maestro. Un 0 la deja como Esclavo.

- **Bit 3 – CPOL: Polaridad del reloj**

  Determina la polaridad del reloj (SCK) cuando la interfaz SPI está inactiva. En la figura 6.11 se mostró el efecto de este bit en la señal SCK.

- **Bit 2 – CPHA: Fase del reloj**

  Determina si los datos son muestreados en fase con el reloj o si se inserta un retardo inicial de medio ciclo de reloj. En la figura 6.11 se mostró el efecto de este bit en el muestreo de datos.

- **Bits 1 y 0 – SPR[1:0]**

  Determinan la frecuencia a la cual se genera la señal de reloj SCK. No tienen efecto si el MCU está configurado como Esclavo. En la tabla 6.8 se muestran las diferentes frecuencias obtenidas a partir del reloj del sistema. También se observa el efecto del bit **SPI2X**, con el que se duplica la frecuencia de la señal SCK.

Tabla 6.8 Frecuencia de la señal de reloj (SCK)

SPI2X	SPR1	SPR0	Frecuencia de SCK
0	0	0	fosc/4
0	0	1	fosc/16
0	1	0	fosc/64
0	1	1	fosc/128
1	0	0	fosc/2
1	0	1	fosc/8
1	1	0	fosc/32
1	1	1	fosc/64

El registro de estado es el **SPSR**, cuyos bits son:

	7	6	5	4	3	2	1	0	
0x0E	SPIF	WCOL	-	-	-	-	-	SPI2X	**SPSR**

- **Bit 7 – SPIF: Bandera de fin de transferencia SPI**

  Produce una interrupción, si se habilitó la interrupción por transferencia serial completa vía SPI y al habilitador global de interrupciones. Aunque también puede sondearse por software.

- **Bit 6 – WCOL: Bandera de colisión de escritura**

  Se pone en alto si se escribe en el registro **SPDR** mientras hay una transferencia en progreso. Las banderas **WCOL** y **SPIF** se limpian con la lectura del registro **SPSR** y el acceso al registro **SPDR.**

- **Bits 5 al 1 – No están implementados**

- **Bit 0 – SPI2X: Duplica la frecuencia de transmisión**

  En la tabla 6.8 se observa el efecto del bit **SPI2X**, duplicando la frecuencia de la señal SCK, definida por los bits **SPR1** y **SPR0**.

6.2.5 Ejemplos de Uso de la Interfaz SPI

En los siguientes ejemplos se ilustra cómo utilizar a la interfaz SPI, cuando el MCU funciona como Maestro o como Esclavo. Puesto que el papel del Maestro difiere del Esclavo, en cada problema se muestran 2 soluciones, codificándolas únicamente en lenguaje C.

Ejemplo 6.7 Sin emplear interrupciones, configure un ATMega8 como Maestro y otro como Esclavo. El Maestro debe enviar una cadena de caracteres terminada con el carácter nulo (0x00). El Esclavo debe colocar cada carácter recibido en su puerto D. Después de enviar al carácter nulo, el Maestro debe solicitar al Esclavo la longitud de la cadena y colocarla en su puerto D.

Suponga que los dispositivos están operando a 1 MHz y configure para que las transmisiones se realicen a 125 KHz.

Se asume que ambos dispositivos se energizan al mismo tiempo, aun con ello, el Maestro espera 100 mS para que el Esclavo esté listo. Después de enviar al carácter nulo, nuevamente el Maestro espera 100 mS antes de solicitar la longitud de la cadena, para evitar errores por colisión de escritura. Es posible realizar diagramas de flujo, puesto que no se emplean interrupciones. El comportamiento del Maestro se describe en el diagrama de la figura 6.13.

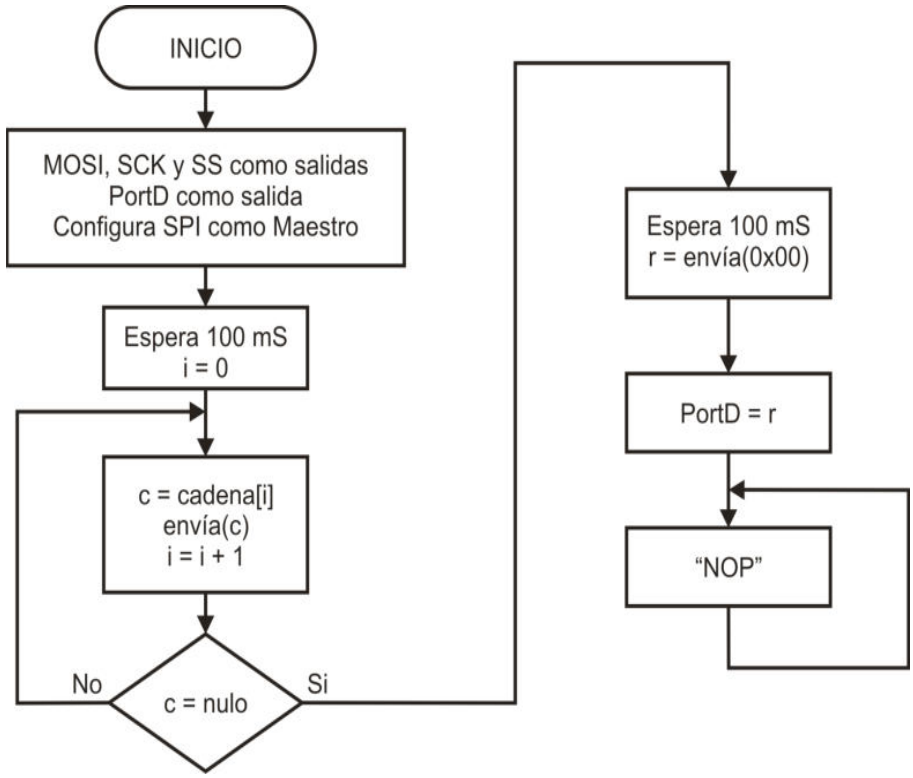


Figura 6.13 Comportamiento del Maestro

La función que envía un dato por SPI regresa el dato recibido, sin embargo, no es importante la respuesta mientras se envíen los caracteres, la respuesta se ignora en las llamadas realizadas dentro del ciclo repetitivo. Por el contrario, al solicitar el número de caracteres sólo la respuesta es importante, podría enviarse cualquier carácter.

El Esclavo tiene un comportamiento diferente, el cual se ilustra en la figura 6.14. Donde se observa que el Esclavo por sí mismo no da una respuesta al Maestro, su respuesta es colocada en el registro **SPDR** y espera a que el Maestro la solicite.

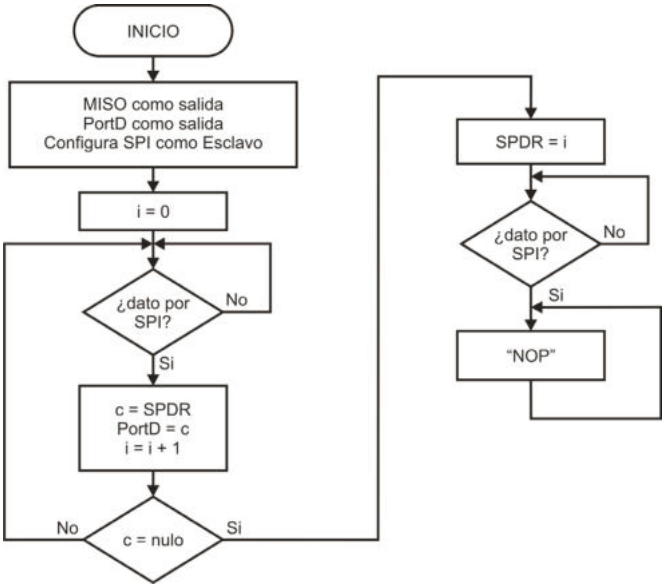


Figura 6.14 Comportamiento del Esclavo

El código en lenguaje C para el Maestro, considerando una cadena constante, es:

```
#define F_CPU 1000000UL              // Frecuencia de trabajo de 1 MHz
#include     <util/delay.h>          // Funciones para retrasos
#include     <avr/io.h>              // Definiciones de Registros I/O

unsigned char envia_SPI(unsigned char dato);   // Prototipo de la función

int  main() {                        // Programa principal
unsigned  char i, r;
char   c;
char   cadena[] = "cadena de prueba";

DDRB = 0b00101100;                   // MOSI, SCK y SS como salidas
DDRD = 0xFF;                         // Puerto D como salida
PORTB = 0x04;                        // SS en alto, Esclavo inhabilitado

SPCR = 0x51;                  // Habilita la interfaz SPI como Maestro
SPSR = 0x01;                         // Ajustando para 125 kHz

_delay_ms(100);                      // Espera a que el Esclavo esté listo
i = 0;                               // índice para el arreglo de datos

do {                                 // Envía caracteres hasta encontrar
       c = cadena[i];                // al carácter nulo
       envia_SPI(c);
       i = i + 1;
} while( c != 0x00);

_delay_ms(100);                      // Espera a que el Esclavo escriba
```

```
                                    // respuesta
r = envia_SPI(0x00);                // Solicita respuesta
PORTD = r;                          // Muestra respuesta

while(1)                            // Lazo infinito
      asm("nop");
}
/* Función para enviar un dato por SPI, debe considerarse la
habilitación e inhabilitación del Esclavo.                          */

unsigned  char  envia_SPI(unsigned char  dato) {
unsigned  char  resp;

PORTB &= 0xFB;                      // SS en bajo, Esclavo habilitado
SPDR = dato;                        // Dato a enviar

while(!(SPSR & 1 << SPIF));         // Espera fin de envío

resp = SPDR;                        // Lee la respuesta del Esclavo
PORTB |= 0x04;                      // SS en alto, Esclavo inhabilitado
return  resp;                       // Regresa la respuesta
}
```

El código para el Esclavo es:

```
#include     <avr/io.h>             // Definiciones de Registros I/O

int  main() {                       // Programa principal
unsigned  char i;
char   c;

DDRB = 0b00010000;                  // MISO como salida
DDRD = 0xFF;                        // Puerto D como salida
SPCR = 0x41;                        // Habilita la interfaz SPI como Esclavo
SPSR = 0x01;                        // Ajustando para 125 kHz
                                    // Aunque puede omitirse en el Esclavo
i = 0;                              // índice para contar los datos
do {
      while(!(SPSR & 1 << SPIF));       // Espera a recibir un dato
      c = SPDR;                         // Lee el dato
      PORTD = c;                        // Muestra el dato
      i = i + 1;
} while( c != 0x00);

SPDR = i;                           // Deja lista la respuesta
while(!(SPSR & 1 << SPIF));         // Espera una petición del Maestro
c = SPDR;                           // Lee, sólo para limpiar la bandera

while(1)                            // Lazo infinito
      asm("nop");}
```

En la solución del ejemplo anterior, se observa que si se utilizaran interrupciones, no se optimizaría al programa del Maestro, puesto que el Maestro define en qué momento

se realiza la transferencia de un dato. Por el contrario, el Esclavo se beneficiaría ampliamente porque puede estar desarrollando otras actividades en lugar de un sondeo continuo.

Ejemplo 6.8 Tomando como base el diagrama de la figura 6.12, al Maestro colóquele 2 arreglos de interruptores (con 8 y 2 interruptores) y un botón. Los 2 interruptores son para seleccionar un Esclavo y los 8 para introducir un dato. Con ello, cada vez que el botón es presionado, el Maestro debe enviar el dato al Esclavo seleccionado. Puesto que las direcciones de los Esclavos son 0, 1 y 2, si se selecciona el 3, el dato se debe mandar a todos los Esclavos (difusión). Cuando un Esclavo reciba un dato, lo debe mostrar en su puerto D.

El hardware, complementado con los interruptores y LEDs, se muestra en la figura 6.15.

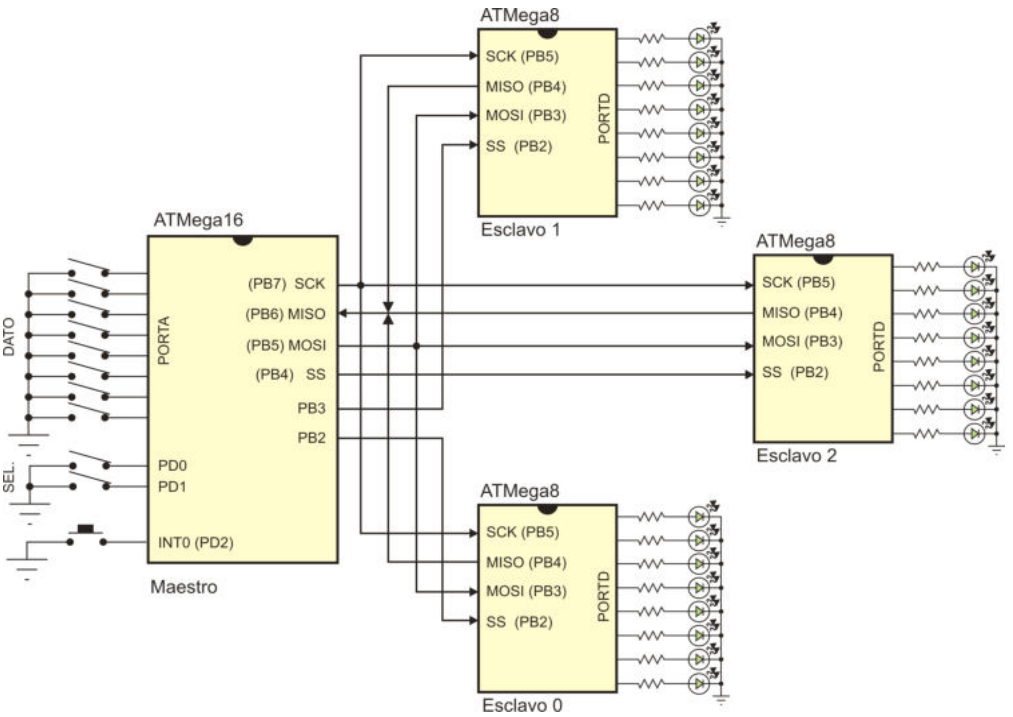


Figura 6.15 Envío de información de un Maestro a 3 Esclavos por SPI

En el Maestro se utiliza la interrupción externa para solicitar un envío. En su ISR se espera hasta que la transferencia concluya, para evitar errores de colisión de escritura. Podría ocurrir que un rebote en el botón solicite un envío mientras una transferencia está en progreso. El programa para el Maestro es:

```
#include      <avr/io.h>                  // Funciones de entrada/salida
#include      <avr/interrupt.h>           // Funciones de las interrupciones

unsigned char envia_SPI(unsigned char dato);   // Prototipo de la función
ISR(INT0_vect) {                          // Rutina que se ejecuta al presionar el botón
unsigned char  sel, dato;

sel = PIND & 0x03;                        // Obtiene el número de Esclavo seleccionado
dato = PINA;                              // Obtiene el dato a enviar
switch(sel) {
       case 0 : PORTB &= 0b11111011;      // Habilita al Esclavo 0
                envia_SPI(dato);          // envía el dato
                PORTB |= 0b00000100;      // Inhabilita al Esclavo 0
                break;
       case 1 : PORTB &= 0b11110111;      // Habilita al Esclavo 1
                envia_SPI(dato);          // envía el dato
                PORTB |= 0b00001000;      // Inhabilita al Esclavo 1
                break;
       case 2 : PORTB &= 0b11101111;      // Habilita al Esclavo 2
                envia_SPI(dato);          // envía el dato
                PORTB |= 0b00010000;      // Inhabilita al Esclavo 2
                break;
       default:                           // Difusión a los 3 Esclavos
                PORTB &= 0b11111011;      // Habilita al Esclavo 0
                envia_SPI(dato);          // envía el dato
                PORTB |= 0b00000100;      // Inhabilita al Esclavo 0
                PORTB &= 0b11110111;      // Habilita al Esclavo 1
                envia_SPI(dato);          // envía el dato
                PORTB |= 0b00001000;      // Inhabilita al Esclavo 1
                PORTB &= 0b11101111;      // Habilita al Esclavo 2
                envia_SPI(dato);          // envía el dato
                PORTB |= 0b00010000;      // Inhabilita al Esclavo 2
       }
}

int  main() {                             // Programa principal

DDRA = 0x00;                              // Puerto A como entrada
DDRB = 0b10111100;                        // MOSI, SCK y SS(s) como salidas
DDRD = 0x00;                              // Puerto D como entrada

PORTA = 0xFF;                             // Resistores de pull-up en el puerto A
PORTD = 0x07;                             // Resistores de pull-up en el puerto D

PORTB = 0b00011100;                       // Esclavos inhabilitados
SPCR = 0x51;                              // Habilita la interfaz SPI como Maestro
SPSR = 0x01;                              // Ajustando para 125 KHz

MCUCR = 0x02;                             // INT0 por flanco de bajada
GICR = 0x40;                              // Habilita la INT0
sei();                                    // Habilitador global de interrupciones
```

```
while(1)                              // Lazo infinito
       asm("nop");
}

unsigned  char  envia_SPI(unsigned char  dato) {
unsigned  char  resp;

SPDR = dato;                          // Dato a enviar
while(!(SPSR & 1 << SPIF));           // Espera fin de envío
resp = SPDR;                          // Lee la respuesta del Esclavo

return  resp;                         // Regresa la respuesta
}
```

En este ejemplo, la habilitación e inhabilitación del Esclavo se realiza en la ISR de la interrupción 0 y no en la función `envia_SPI`, esto porque la función es utilizada para que el Maestro envíe datos a 3 Esclavos diferentes.

Los Esclavos reciben los datos por interrupciones, en su lazo infinito permanecen ociosos. Los 3 Esclavos ejecutan el mismo programa, el cual es:

```
#include      <avr/io.h>                  // Funciones de entrada/salida
#include      <avr/interrupt.h>           // Funciones de las interrupciones

ISR(SPI_STC_vect){          // ISR por fin de transferencia por la interfaz SPI
       PORTD = SPDR;        // Lee y muestra el dato recibido
}

int  main() {               // Programa principal

DDRB = 0b00010000;          // MISO como salida

DDRD = 0xFF;                // Puerto D como salida

SPCR = 0xC1;  // Habilita la interfaz SPI, Esclavo con interrupciones

SPSR = 0x01;                // Ajustando para 125 KHz

sei();                      // Habilitador global de interrupciones

while(1)                    // Lazo infinito

       asm("nop");
}
```

El ejemplo anterior sirve como una referencia para expandir el número de puertos de un MCU, siempre que los periféricos a manejar en los nuevos puertos no requieran un tiempo de respuesta rápido, dado que la información fluye de manera serial.

6.3 Comunicación Serial por TWI

La Interfaz Serial de Dos Hilos (TWI, *Two Wire Serial Interface*) es un recurso disponible para que diferentes microcontroladores (u otros dispositivos) se comuniquen por medio de un bus bidireccional de 2 líneas, una para reloj (SCL) y otra para datos (SDA). En la figura 6.16 se muestra la forma en que los diferentes dispositivos se conectan, como hardware externo únicamente se requiere de 2 resistencias conectadas a Vcc (*pull-up*). El protocolo TWI permite al diseñador conectar hasta 128 dispositivos diferentes, cada uno con su propia dirección.

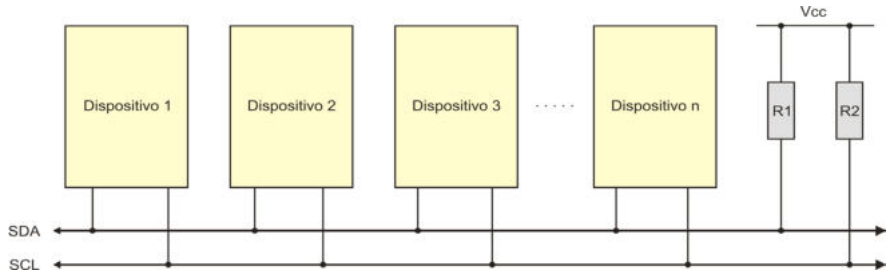


Figura 6.16 Bus de interconexión TWI

Esta interfaz es compatible en su operación con el bus I^2C, el cual es un estándar desarrollado por *Philips Semiconductor* (ahora *NXP Semiconductor*). Por lo tanto, puede utilizarse para el manejo de una gama muy amplia de dispositivos, como manejadores de LCDs y LEDs, puertos remotos de entrada/salida, RAMs, EEPROMs, relojes de tiempo real, ADCs, DACs, etc.

Los dispositivos deben contar con los mecanismos de hardware necesarios para cubrir con los requerimientos inherentes al protocolo TWI. Sus salidas deben manejar un tercer estado, siendo de colector o drenaje abierto. Las resistencias de *pull-up* imponen un nivel lógico alto en el bus cuando todas las salidas están en un tercer estado. Si en una o más salidas hay un nivel bajo, el bus va a reflejar ese nivel bajo. Con ello, se implementa una función AND alambrada, la cual es esencial para la operación del bus.

El protocolo TWI maneja un esquema Maestro-Esclavo, no obstante, cualquier dispositivo puede transmitir en el bus. Por ello, deben distinguirse los siguientes términos:

- **Maestro:** Dispositivo que inicia y termina una transmisión, también genera la señal de reloj (SCL).
- **Esclavo:** Dispositivo direccionado por un Maestro.
- **Transmisor:** Dispositivo que coloca los datos en el bus.
- **Receptor:** Dispositivo que lee los datos del bus.

6.3.1 Transferencias de Datos vía TWI

Cada bit transferido en la línea SDA debe acompañarse de un pulso en la línea SCL. El dato debe estar estable cuando la línea de reloj esté en alto, como se muestra en la figura 6.17(a). Las excepciones a esta regla se dan cuando se están generando las condiciones de INICIO y PARO.

El Maestro inicia y termina la trasmisión de datos, por lo tanto, es el Maestro quien genera las condiciones de INICIO y PARO, éstas son señalizadas cambiando el nivel en la línea SDA cuando SCL está en alto, se muestran en la figura 6.17 (b). Entre estas condiciones el bus se considera ocupado y ningún otro Maestro puede tomar control de él. Una situación especial se presenta si un nuevo INICIO es generado antes de un PARO, esta condición es referida como un INICIO REPETIDO y es utilizada por un Maestro cuando desea iniciar con una nueva transferencia, sin renunciar al control del bus. El INICIO REPETIDO se comporta como un INICIO y también es mostrado en la figura 6.17 (b).

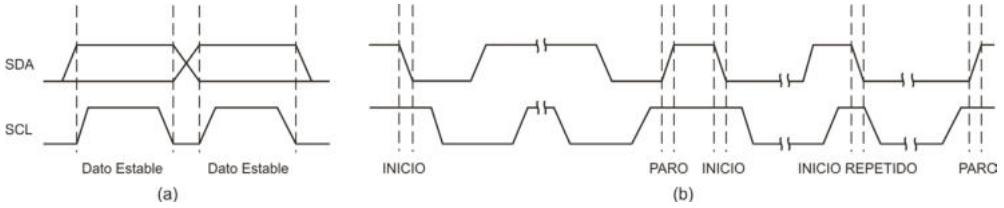


Figura 6.17 (a) Formato para datos válidos, y (b) Condiciones de INICIO, PARO e INICIO REPETIDO

6.3.1.1 Formato de los Paquetes de Dirección

Los paquetes de dirección, en el bus TWI, tienen una longitud de 9 bits, de los cuales, 7 bits son para la dirección de un Esclavo (iniciando con el MSB), 1 bit de control (R/W) y 1 bit de reconocimiento. El bit de control determina si se realiza una lectura (R = 1) o una escritura (W = 0). El bit de reconocimiento sirve para que el Esclavo direccionado dé respuesta al Maestro, colocando un 0 en la señal SDA durante el 9º ciclo de la señal SCL (respuesta referida como **ACK**). Si el Esclavo se encuentra ocupado o por alguna razón no da respuesta al Maestro, la señal SDA se va a mantener en alto (respuesta referida como **nACK**), entonces, el Maestro puede transmitir una condición de PARO o un INICIO REPETIDO, para iniciar nuevas transmisiones. A una trama de dirección con una petición de lectura se le refiere como SLA+R y con una petición de escritura es referida como SLA+W. En la figura 6.18 se muestra el formato de un paquete de dirección.

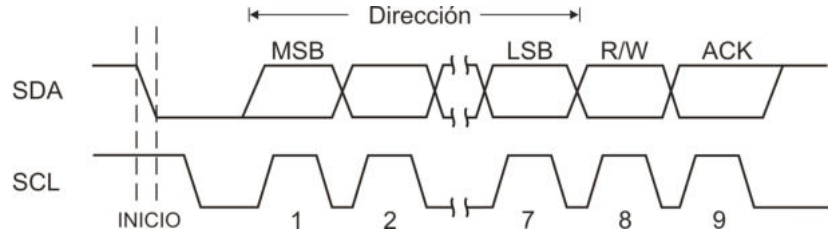


Figura 6.18 Formato de un paquete de dirección

Los Esclavos pueden tener cualquier dirección, excepto la 000 0000, la cual está reservada para una llamada general (GCA, *General Call Address*), es decir, del Maestro a todos los Esclavos. En una llamada general, el Maestro transmite el mismo mensaje a todos los Esclavos, los Esclavos deben responder con **ACK**. Las llamadas generales sólo son para peticiones de escritura (W = 0). Una petición de lectura en una llamada general provocaría una colisión en el bus, porque los Esclavos podrían transmitir datos diferentes.

6.3.1.2 Formato de los Paquetes de Datos

Los paquetes de datos también son de 9 bits, 8 bits para el dato (iniciando con el MSB) y un bit de reconocimiento. Durante las transferencias de datos, el Maestro genera la señal de reloj y las condiciones de INICIO y PARO. El receptor es el responsable de generar la señal de reconocimiento, poniendo en bajo a la señal SDA durante el 9º bit de SCL (**ACK**). En la figura 6.19 se muestra el formato de un paquete de datos. El Maestro no siempre es el transmisor, también un Esclavo puede serlo.

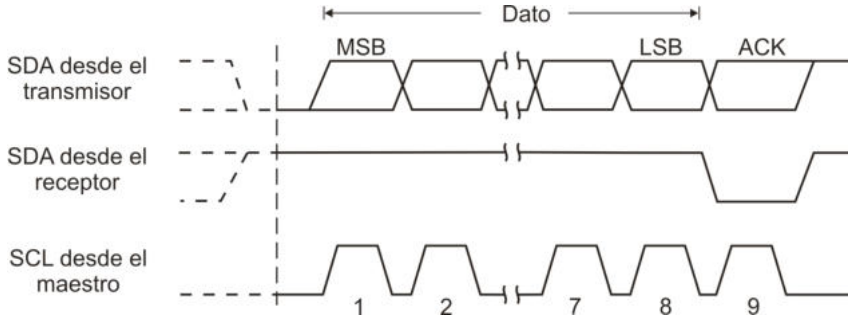


Figura 6.19 Formato de un paquete de datos

6.3.1.3 Transmisión Completa: Dirección y Datos

Un mensaje incluye una condición de INICIO, una SLA+R/W, uno o más paquetes de datos y una condición de PARO. Un mensaje vacío (INICIO seguido de un PARO) es ilegal y debe evitarse. La AND alambrada con las resistencias de *pull-up* sirve para coordinar la comunicación entre el Maestro y el Esclavo, si el Esclavo requiere un tiempo mayor para procesamiento entre bits, puede colocar un nivel bajo en SCL, retrasando con ello al bit siguiente. Esto no afecta el tiempo en alto de la señal SCL, el

cual es controlado por el Maestro. Por lo tanto, el Esclavo puede modificar la velocidad de transmisión de datos en el bus TWI.

En la figura 6.20 se ilustra la transmisión de un mensaje. Entre la SLA+R/W y la condición de PARO se pueden transferir muchos bytes de información, dependiendo del protocolo implementado por el software de la aplicación.

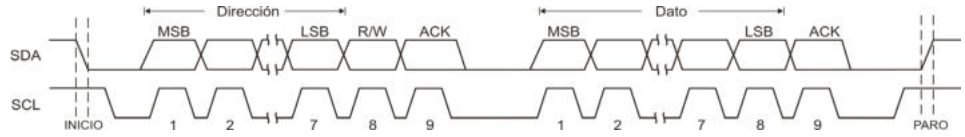


Figura 6.20 Transmisión típica de datos por TWI

6.3.2 Sistemas Multi-Maestros

El protocolo TWI permite el manejo de varios Maestros. Si dos o más Maestros intentan iniciar con una transmisión al mismo tiempo, pueden ocurrir 2 problemas que deben resolverse para que la comunicación proceda de manera normal. Estos problemas son:

- Sólo un Maestro puede concluir con la transmisión. Un proceso de selección, conocido como arbitración, define qué Maestro va a continuar con las transferencias.
- Los Maestros pueden generar señales de reloj que no están en fase. Se debe generar una señal de reloj sincronizada con el reloj de los diferentes Maestros, porque con ella va a realizarse el proceso de arbitración.

La AND alambrada es fundamental en la solución de estos problemas. En la figura 6.21 se muestra la señal de reloj en la línea SCL, resultante de la combinación de la señal de reloj de los Maestros A y B.

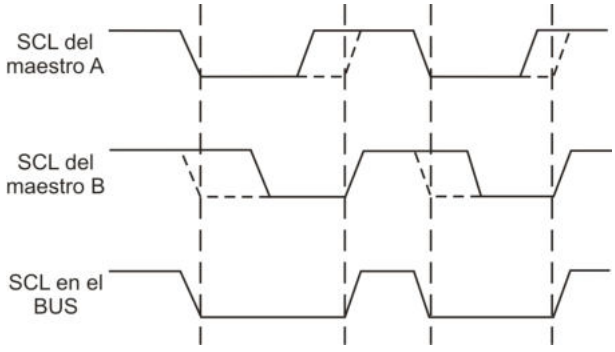


Figura 6.21 Sincronización de las señales de reloj de 2 Maestros

La AND ocasiona que en la señal de reloj resultante, el tiempo en alto sea más corto que en las señales de reloj de los Maestros. Esto porque corresponde con la intersección de las 2 señales. En consecuencia, el tiempo en bajo es más grande. Los Maestros que intenten transmitir en el bus se deben sincronizar con la señal de reloj resultante.

La arbitración la realizan los Maestros, monitoreando la línea SDA después de colocar un dato. Si el valor leído no coincide con el valor colocado por un Maestro, éste ha perdido la arbitración. Esto significa que un Maestro pierde la arbitración cuando coloca un 1 en el bus y en el mismo ciclo de reloj otro Maestro coloca un 0 (la AND hace que lea un 0), en la figura 6.22 se ilustra este proceso. Los Maestros que pierdan el proceso de arbitración inmediatamente deben conmutarse al modo de Esclavos, porque pueden ser direccionados por el Maestro ganador. El Maestro perdedor debe dejar la línea SDA en alto, pero puede continuar generando la señal de reloj hasta concluir con el paquete actual, de dirección o dato.

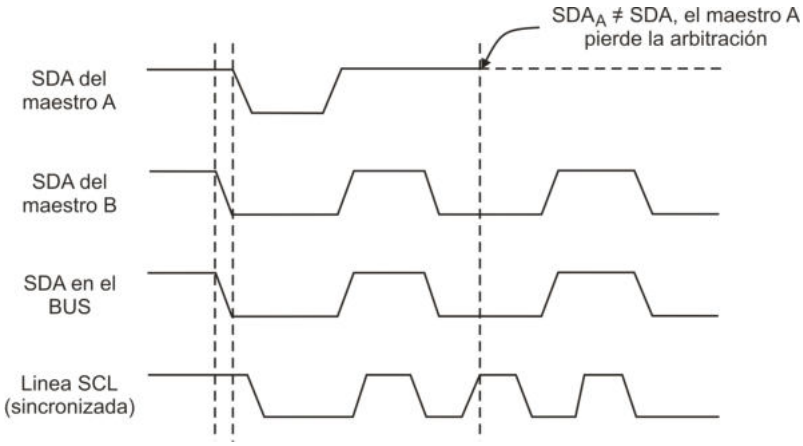


Figura 6.22 Proceso de arbitración ente 2 Maestros

La arbitración continúa hasta que sólo queda un Maestro y puede requerir de muchos bits. Si más de un Maestro direcciona al mismo Esclavo, la arbitración sigue en el paquete de datos. Existen algunas condiciones ilegales de arbitración, las cuales deben evitarse:

- Entre una condición de INICIO REPETIDO y el bit de un dato.
- Entre una condición de PARO y el bit de un dato.
- Entre una condición de INICIO REPETIDO y una condición de PARO.

Es responsabilidad del programador evitar que estas condiciones ocurran. Para ello, es conveniente que en un sistema multi-Maestros todas las transmisiones contengan el mismo número de paquetes de datos.

6.3.3 Organización de la Interfaz TWI

La interfaz TWI se compone de diferentes módulos, su organización se muestra en la figura 6.23. Todos los registros de la interfaz son accesibles por el núcleo AVR, desde el bus interno. Los registros se describen en la sección 6.3.4.

6.3.3.1 Terminales SCL y SDA

Son las terminales para la conexión con un bus TWI. Ambas incluyen un control de *slew-rate*, para cumplir con las especificaciones de la interfaz, y un filtro capaz de suprimir picos con una duración menor a 50 nS. El bus TWI requiere de 2 resistores externos de *pull-up*, es posible habilitar y usar los resistores de las terminales SCL y SDA, para no agregar hardware adicional.

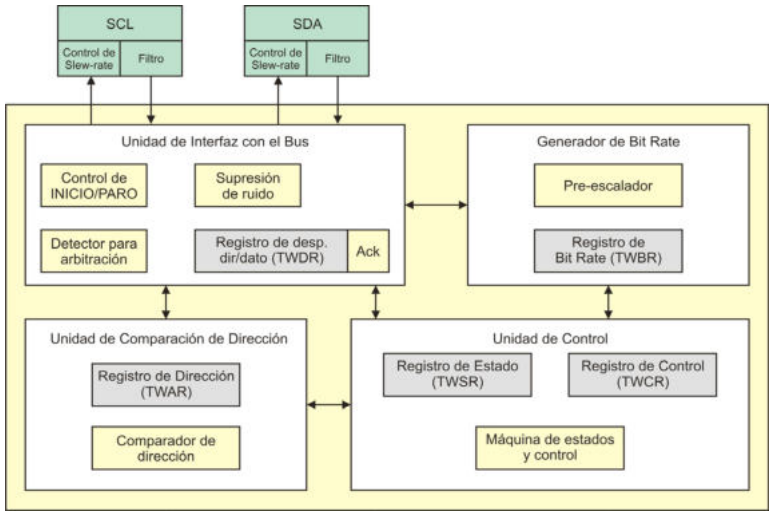


Figura 6.23 Organización de la Interfaz TWI

6.3.3.2 Generador de Bit Rate

Este módulo determina el periodo de la señal SCL, cuando el MCU está operando en modo Maestro. El periodo depende del valor del Registro de Bit Rate (**TWBR**) y de la configuración del pre-escalador, la cual se realiza con los bits **TWPS[1:0]** del Registro de Estado de la Interfaz TWI (**TWSR**). Considerando los valores del registro **TWBR** y de los bits **TWPS**, la frecuencia de SCL se genera de acuerdo con la ecuación:

$$frecuencia_{SCL} = \frac{frecuencia_{CPU}}{16 + 2(TWBR) \cdot 4^{TWPS}}$$

Si el MCU opera como Esclavo, no genera la señal SCL, sólo requiere operar a una frecuencia por lo menos de 16 veces la frecuencia de SCL.

Debe considerarse que un Esclavo puede prolongar el tiempo en bajo de la señal SCL, modificando el periodo promedio de la señal de reloj en el bus TWI.

6.3.3.3 Unidad de Interfaz con el Bus

Este módulo incluye al Registro de Desplazamiento para Datos y Direcciones (**TWDR**), en este registro se ubica la dirección o dato a ser transmitido, o la dirección o dato recibido. El bit Ack, adyacente al registro **TWDR** en la figura 6.23, es para el manejo del bit de reconocimiento. Este bit no es accesible por software, sin embargo, cuando se están recibiendo datos, puede ser puesto en alto o en bajo manipulando al Registro de Control de la interfaz TWI (**TWCR**). Cuando se está transmitiendo, el valor del bit AcK recibido se determina por los bits de estado en el registro **TWSR**.

El bloque para el Control de INICIO/PARO es responsable de generar y detectar las condiciones de INICIO, INICIO REPETIDO y PARO. Las condiciones de INICIO y PARO se detectan aun si el MCU está en alguno de los modos de reposo, "despertando" al MCU si fue direccionado por un Maestro.

El bloque Detector para arbitración incluye el hardware necesario para monitorear continuamente la actividad en el bus y determinar si una arbitración está en proceso, cuando la interfaz TWI ha iniciado una transmisión como Maestro. Si la interfaz TWI pierde el proceso de arbitración, debe informar a la Unidad de Control para que se muestre el estado y se realicen las acciones necesarias.

6.3.3.4 Unidad de Comparación de Dirección

El bloque Comparador de Dirección evalúa si la dirección recibida coincide con los 7 bits del Registro de Dirección de la Interfaz TWI (**TWAR**). Si el bit **TWGCE** del registro **TWAR** está en alto, el comparador está habilitado para reconocer llamadas generales en la interfaz TWI, por lo tanto, las direcciones entrantes también se deben comparar con la GCA. Ante una coincidencia, se informa a la Unidad de Control, para que realice las acciones correspondientes. La interfaz TWI puede o no reconocer su dirección, dependiendo de su configuración en el registro **TWCR**.

El comparador de dirección trabaja aun cuando el MCU se encuentra en modo de reposo, "despertando" al MCU si fue direccionado por un Maestro. Si ocurre otra interrupción mientras se realiza la comparación, la operación en la interfaz TWI es abortada y el recurso regresa a un estado ocioso. Por ello, antes de llevar al MCU a un modo de reposo, es conveniente únicamente activar la interrupción por la interfaz TWI.

6.3.3.5 Unidad de Control

La Unidad de Control monitorea los eventos que ocurren en el bus y genera respuestas de acuerdo con la configuración definida en el registro **TWCR**. Si un evento requiere atención, se pone en alto a la bandera **TWINT** y en el siguiente ciclo de reloj se actualiza al Registro de Estado (**TWSR**), mostrando el código que identifica al evento. El registro **TWSR** sólo tiene información relevante después de que la bandera **TWINT** es puesta en alto, en otras circunstancias, contiene un código de estado especial, indicando que

no hay información disponible. Tan pronto como la bandera **TWINT** es puesta en alto, la línea SCL es ajustada a un nivel bajo para permitir que la aplicación concluya con sus tareas por software, antes de continuar con las transmisiones TWI.

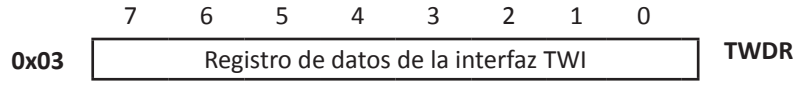
La bandera **TWINT** es puesta en alto ante las siguientes situaciones:

- Se transmitió una condición de INICIO o INICIO REPETIDO.
- Se transmitió una SLA+R/W.
- Se perdió una arbitración.
- La interfaz fue direccionada, con su dirección de Esclavo o por una GCA.
- Se recibió un dato.
- Recibió una condición de PARO o INICIO REPETIDO, mientras estaba direccionada como Esclavo.
- Ocurrió un error en el bus, debido a una condición ilegal de INICIO o PARO.

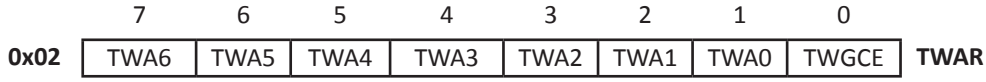
6.3.4 Registros para el Manejo de la Interfaz TWI

La interfaz TWI es manejada por 5 registros, para los datos (**TWDR**), para la dirección como esclavo (**TWAR**), para la velocidad de transmisión (**TWBR**), para el control (**TWCR**) y para el estado (**TWSR**). En esta sección se describe el papel de cada uno de estos registros.

El registro **TWDR** es el buffer para transmisión y recepción de datos. En modo transmisor, **TWDR** contiene el próximo dato a ser transmitido. En modo receptor, **TWDR** contiene el último dato recibido. Este registro sólo puede ser escrito después de la notificación de un evento, con la puesta en alto del bit **TWINT**, por ello, el registro no puede ser modificado por el usuario hasta después de que ocurra la primera interrupción. El contenido de **TWDR** permanece estable tan pronto como **TWINT** es puesto en alto.



La interfaz TWI debe contar con una dirección para que pueda funcionar como Esclavo, aunque también puede atender a llamadas generales (GCA). En el Registro de Dirección (**TWAR**) se definen ambos parámetros, los bits de este registro son:



- **Bits 7 al 1 – TWA[6:0] Bits de dirección (como Esclavo)**

  En estos 7 bits se define la dirección de la interfaz TWI como Esclavo.

- **Bit 0 – TWGCE: Habilitador para reconocer la dirección de llamadas generales**

  Si está en alto, la interfaz TWI reconoce y puede dar respuesta a una GCA.

El registro base para definir la razón de transmisión de datos es el Registro de Bit Rate (**TWBR**). Con el valor de **TWBR** y con la selección en el pre-escalador, se define la frecuencia de la señal SCL, cuando el MCU trabaja como Maestro.

	7	6	5	4	3	2	1	0	
0x00	Registro de Bit Rate de la interfaz TWI								**TWBR**

El Registro para el Control de la Interfaz TWI es el **TWCR**, con este registro es posible realizar diferentes tareas: Habilitar a la interfaz, aplicar una condición de INICIO, generar una condición de PARO y controlar las transferencias del bus al registro **TWDR**. Los bits del registro **TWCR** son:

	7	6	5	4	3	2	1	0	
0x36	TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	-	TWIE	**TWCR**

- **Bit 7 – TWINT: Bandera de interrupción por la interfaz TWI**

  Su puesta en alto indica que en el bus ocurrió un evento que requiere atención por software. Se produce una interrupción si el habilitador global (bit **I** en **SREG**) y el habilitador individual (bit **TWIE**) están activados. Este bit debe ser limpiado por software escribiéndole un 1 lógico, aun cuando se configure su interrupción. Esto es necesario porque la señal SCL mantiene un nivel bajo mientras el bit **TWINT** está en alto, en espera de que por software se conozca al estado de la interfaz y se le dé respuesta. Una vez que la bandera se ha limpiado, ya no se puede tener acceso a los registros **TWAR**, **TWSR** y **TWDR**.

- **Bit 6 – TWEA: Habilitador de la generación del bit de reconocimiento (Ack)**

  Cuando este bit está en alto, la interfaz TWI genera el bit de reconocimiento si ocurre alguna de las siguientes situaciones:

  1. Se ha recibido la dirección del Esclavo.
  2. Se ha recibido una llamada general, estando el bit **TWGCE** de **TWAR** en alto.
  3. Se ha recibido un dato, en el modo Maestro Receptor o Esclavo Receptor.

  Si el bit **TWEA** tiene un nivel bajo, la interfaz TWI está virtualmente desconectada del bus.

- **Bit 5 – TWSTA: Bit para una condición de INICIO en el bus TWI**

  Se escribe un 1 en este bit para que llegue a ser un Maestro en el bus TWI. Si el bus está libre genera una condición de INICIO. Si el bus no está libre, la interfaz espera hasta detectar una condición de PARO, para después generar una nueva condición de INICIO, reintentando el acceso al bus como Maestro. El bit **TWSTA** debe limpiarse por software después de que la condición de INICIO ha sido transmitida.

- **Bit 4 – TWSTO: Bit para una condición de PARO en el bus TWI**

  En el modo Maestro, un 1 en este bit genera una condición de PARO en el bus TWI. El bit se limpia automáticamente por hardware después de la condición de PARO. En modo Esclavo, un 1 en **TWSTO** puede recuperar a la interfaz de una condición de error. No se genera una condición de PARO, pero la interfaz regresa a un modo de Esclavo sin direccionar, llevando a las líneas SCL y SDA a un estado de alta impedancia.

- **Bit 3 – TWWC: Bandera de colisión de escritura**

  Esta bandera se pone en alto si se intenta escribir en el registro **TWDR** cuando el bit **TWINT** está en bajo. La bandera se limpia cuando se escribe en **TWDR** después de que el bit **TWINT** es puesto en alto.

- **Bit 2 – TWEN: Habilitador de la interfaz TWI**

  Con este bit en alto se habilita la operación de la interfaz TWI. La interfaz toma el control de las terminales SCL y SDA, habilitando al limitador de *slew-rate* y al filtro eliminador de ruido. Si en el bit **TWEN** se escribe un 0, la interfaz se apaga y todas las transmisiones terminan.

- **Bit 1 – No está implementado.**

- **Bit 0 – TWIE: Habilitador de interrupción por TWI**

  Si este bit está en alto, y el bit **I** de **SREG** también, se genera una interrupción cuando la bandera **TWINT** es puesta en alto.

El estado de la interfaz TWI se conoce por medio del registro **TWSR**. En este registro también se define el factor de pre-escala, los bits de **TWSR** son:

	7	6	5	4	3	2	1	0	
0x01	TWS7	TWS6	TWS5	TWS4	TWS3	-	TWPS1	TWPS0	**TWSR**

- **Bits 7 al 3 – TWS[7:3] Bits del estado de la interfaz TWI**

  En estos 5 bits se refleja el estado de la interfaz y del bus. Los diferentes códigos de estado se revisan en la siguiente sección, la cual describe los modos de transmisión

de la interfaz TWI. Por la presencia de los bits **TWPS[1:0]**, para la lectura correcta del estado se debe utilizar una máscara que únicamente conserve a los bits **TWS[7:3]**.

- **Bit 2 – No está implementado**
- **Bits 1 y 0 – TWPS[1:0]: Bits para la selección del factor de pre-escala en la interfaz TWI**

  En la tabla 6.9 se muestran los diferentes factores de pre-escala, con este factor y el valor del registro **TWBR**, se define la frecuencia a la que se va a generar la señal SCL cuando el MCU trabaja como Maestro.

Tabla 6.9 Factores de pre-escala, para definir la razón de transmisión por TWI

TWPS1	TWPS0	Factor de pre-escala
0	0	1
0	1	4
1	0	16
1	1	64

6.3.5 Modos de Transmisión y Códigos de Estado

La interfaz puede operar en 4 modos: Maestro Transmisor (MT), Esclavo Transmisor (ST), Maestro Receptor (MR) y Esclavo Receptor (SR). Una aplicación puede requerir más de un modo de operación. Por ejemplo, si un MCU va a manejar una memoria EEPROM vía TWI, con el modo MT puede escribir en la memoria y con el modo MR puede leer de ella. No obstante, si en la misma aplicación otro MCU direcciona al primero, éste también puede funcionar en los modos ST y SR. La aplicación decide cual es el modo más conveniente.

En los siguientes apartados se describen los modos de transmisión, listando los códigos de estado que se generan en cada transferencia. Un código de estado se genera cuando la bandera **TWINT** es puesta en alto, la actividad en el bus es detenida (señal SCL en bajo), por software debe leerse el estado de la interfaz y preparar la respuesta, en función de la aplicación. Con la respuesta lista debe limpiarse a la bandera **TWINT**, para continuar con las actividades del bus.

6.3.5.1 Modo Maestro Transmisor

Un MCU en el modo MT envía una cantidad de bytes a un MCU en el modo SR, esto se muestra en la figura 6.24. Un MCU entra al modo Maestro después de transmitir una condición de INICIO, posteriormente, el formato de la dirección determina si va a ser MT o MR. Para el modo MT se debe enviar una SLA+W (W = 0).

Una condición de inicio se genera escribiendo el siguiente valor en **TWCR:**

TWCR valor	TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	-	TWIE
	1	X	1	0	X	1	0	X

Con el cual, la interfaz TWI es habilitada (**TWEN** = 1), se da paso a la condición de INICIO (**TWSTA** = 1) y se limpia la bandera **TWINT** (escribiéndole un 1). Con ello, si el bus está disponible se transmite la condición de inicio, la bandera **TWINT** es puesta nuevamente en alto y en los bits de estado, del registro **TWSR**, se obtiene el código 0x08.

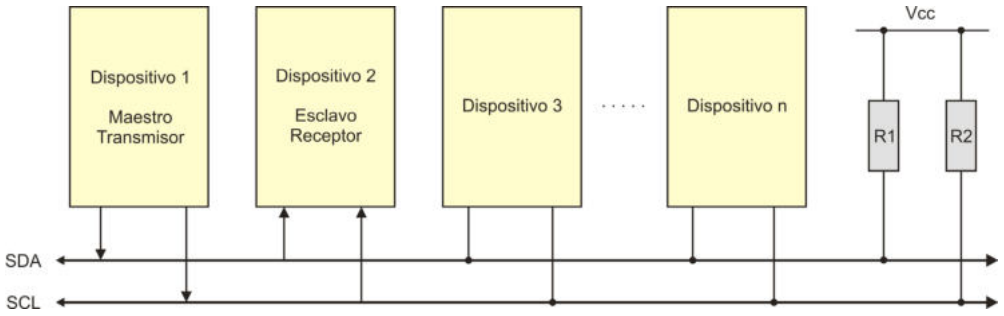


Figura 6.24 El dispositivo 1 transfiere datos en el modo Maestro Transmisor

En la tabla 6.10 se muestran los estados posibles en el modo MT, con una descripción del estado de la interfaz y las posibles acciones a seguir. Para todos los códigos se asume un enmascaramiento de los bits del pre-escalador (**TWPS**), de manera que se consideran como ceros.

Tabla 6.10 Estados posibles en el modo Maestro Transmisor

Código de Estado	Estado del bus y de la interfaz serial	Posibles acciones a realizar en la interfaz, en respuesta al estado
0x08	Una condición de INICIO ha sido transmitida	1. Transmitir SLA+W, recibir ACK o nACK
0x10	Una condición de INICIO REPETIDO ha sido transmitida	1. Transmitir SLA+W, recibir ACK o nACK 2. Transmitir SLA+R, conmutar la interfaz a MR
0x18	Se ha transmitido una SLA+W y recibido un ACK	1. Transmitir un byte de datos, recibir ACK o nACK 2. Transmitir un INICIO REPETIDO 3. Transmitir una condición de PARO 4. Transmitir una condición de PARO seguida de una condición de INICIO
0x20	Se ha transmitido una SLA+W y recibido un nACK	
0x28	Se ha transmitido un byte de datos y recibido un ACK	
0x30	Se ha transmitido un byte de datos y recibido un nACK	
0x38	Se ha perdido una arbitración al enviar una SLA o un byte de datos	1. Liberar al bus, únicamente limpiando a la bandera **TWINT** 2. Transmitir una condición de INICIO, cuando el bus esté libre

Para entrar al modo MT, el Maestro debe escribir una SLA+W en el registro **TWDR** para su transmisión. La transmisión de la SLA+W inicia cuando se limpia la bandera **TWINT**, para ello, en el registro **TWCR** debe escribirse el valor:

TWCR	TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	-	TWIE
Valor	1	X	0	0	X	1	0	X

Después de transmitir la SLA+W y recibir el bit de reconocimiento, la bandera **TWINT** es puesta en alto y en el registro **TWSR** se obtiene uno de los posibles estados: 0x18, 0x20 ó 0x38. Las acciones a realizar dependen del código de estado (tabla 6.10). Si la SLA+W se transmitió con éxito, el MCU Maestro está listo para enviar uno o varios datos. El dato a enviar debe colocarse en **TWDR**, mientras **TWINT** esté en alto. Después de escribir el dato, la bandera **TWINT** debe limpiarse escribiéndole un 1, en el registro **TWCR** debe escribirse el valor mostrado anteriormente. La actividad en el bus continúa y el dato escrito en **TWDR** es enviado. Este esquema se repite con cada uno de los datos. Cuando la bandera **TWINT** esté en alto, el dato debe ser escrito en **TWDR**, luego, debe limpiarse a la bandera para que el dato se envíe.

Una vez que se ha concluido con el envío de datos, el Maestro debe enviar una condición de PARO o una de INICIO REPETIDO. El valor del registro **TWCR** para una condición de PARO es:

TWCR	TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	-	TWIE
Valor	1	X	0	1	X	1	0	X

Un INICIO REPETIDO se solicita con el mismo valor que el de una condición de INICIO. Después de un INICIO REPETIDO (estado 0x10) la interfaz puede tener acceso al mismo o a otro Esclavo, sin transmitir una condición de PARO. El INICIO REPETIDO habilita a un Maestro a conmutar entre Esclavos o cambiar de Maestro Transmisor a Maestro Receptor, sin perder el control del bus.

6.3.1.1 Modo Maestro Receptor

Un MCU en el modo MR recibe una cantidad de bytes de un MCU en el modo ST, esto se muestra en la figura 6.25. Un MCU entra al modo Maestro después de transmitir una condición de INICIO, posteriormente, el formato de la dirección determina si va a ser MT o MR. Para el modo MR se debe enviar una SLA+R (R = 1).

Una condición de inicio se genera escribiendo el siguiente valor en **TWCR:**

TWCR	TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	-	TWIE
valor	1	X	1	0	X	1	0	X

Con el cual, la interfaz TWI es habilitada (**TWEN** = 1), se da paso a la condición de INICIO (**TWSTA** = 1) y se limpia la bandera **TWINT** (escribiéndole un 1). Con ello, si el bus está disponible se transmite la condición de inicio, la bandera **TWINT** es puesta nuevamente en alto y en los bits de estado, del registro **TWSR**, se obtiene el código 0x08.

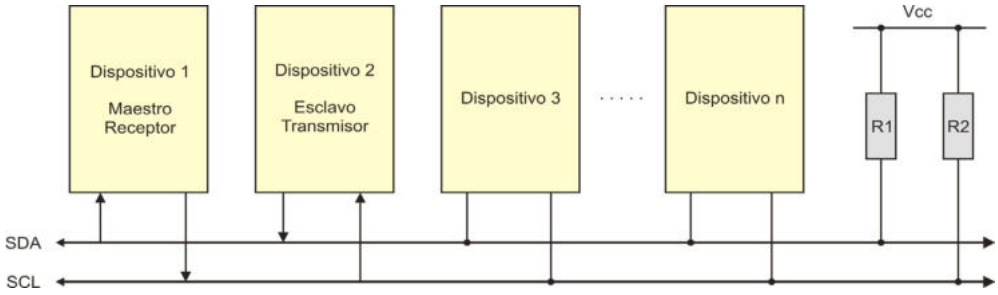


Figura 6.25 El dispositivo 1 recibe datos en el modo Maestro Receptor

En la tabla 6.11 se muestran los estados posibles en el modo MR, con una descripción del estado de la interfaz y las posibles acciones a seguir. Para todos los códigos se asume un enmascaramiento de los bits del pre-escalador (**TWPS**), de manera que se consideran como ceros.

Tabla 6.11 Estados posibles en el modo Maestro Receptor

Código de Estado	Estado del bus y de la interfaz serial	Posibles acciones a realizar en la interfaz, en respuesta al estado
0x08	Una condición de INICIO ha sido transmitida	1. Transmitir SLA+R, recibir ACK o nACK
0x10	Una condición de INICIO REPETIDO ha sido transmitida	1. Transmitir SLA+R, recibir ACK o nACK 2. Transmitir SLA+W, conmutar la interfaz a MT
0x38	Se perdió una arbitración al enviar una SLA+R o se envió un nACK	1. Liberar al bus, únicamente limpiando a la bandera **TWINT** 2. Transmitir una condición de INICIO, cuando el bus esté libre
0x40	Se ha transmitido una SLA+R y recibido un ACK	1. Recibir un byte de datos y dar respuesta con un ACK (**TWEA** = 1) 2. Recibir un byte de datos y dar respuesta con un nACK (**TWEA** = 0)
0x50	Se ha recibido un byte de datos y respondido con un ACK	
0x48	Se ha transmitido una SLA+R y recibido un nACK	1. Transmitir un INICIO REPETIDO 2. Transmitir una condición de PARO 3. Transmitir una condición de PARO seguida de una condición de INICIO
0x58	Se ha recibido un byte de datos y respondido con un nACK	

Para entrar al modo MR, el Maestro debe escribir una SLA+R en el registro **TWDR** para su transmisión. La transmisión de la SLA+R inicia cuando se limpia la bandera **TWINT**, para ello, en el registro **TWCR** debe escribirse el valor:

TWCR	TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	-	TWIE
Valor	1	X	0	0	X	1	0	X

Después de transmitir la SLA+R y recibir el bit de reconocimiento, la bandera **TWINT** es puesta en alto y en el registro **TWSR** se obtiene uno de los posibles estados: 0x38, 0x40 ó 0x48. Las acciones a tomar dependen del código de estado (tabla 6.11).

Si la SLA+R se transmitió con éxito, el MCU Maestro está listo para recibir uno o varios datos. Cada dato recibido se lee de **TWDR** cuando **TWINT** es puesta en alto, respondiendo con un bit ACK mientras se siguen recibiendo datos. La respuesta con reconocimiento y la limpieza de la bandera **TWINT** se realizan al escribir en el registro **TWCR** el siguiente valor:

TWCR	TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	-	TWIE
Valor	1	1	0	0	X	1	0	X

Una respuesta con un nACK le indica al Esclavo que ya no se van a recibir más datos. El valor a escribir en el registro **TWCR** es:

TWCR	TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	-	TWIE
Valor	1	0	0	0	X	1	0	X

Una vez que se ha concluido con la recepción de datos, el Maestro debe enviar una condición de PARO o una de INICIO REPETIDO. El valor del registro **TWCR**, para una condición de PARO, es:

TWCR	TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	-	TWIE
valor	1	X	0	1	X	1	0	X

Un INICIO REPETIDO se solicita con el mismo valor que el de una condición de INICIO. Después de un INICIO REPETIDO (estado 0x10) la interfaz puede tener acceso al mismo o a otro Esclavo, sin transmitir una condición de PARO. El INICIO REPETIDO habilita a un Maestro a conmutar entre Esclavos o cambiar de Maestro Receptor a Maestro Transmisor, sin perder el control del bus.

6.3.5.4 Modo Esclavo Receptor

Un MCU en el modo SR recibe una cantidad de bytes de un MCU en el modo MT, esto se muestra en la figura 6.26.

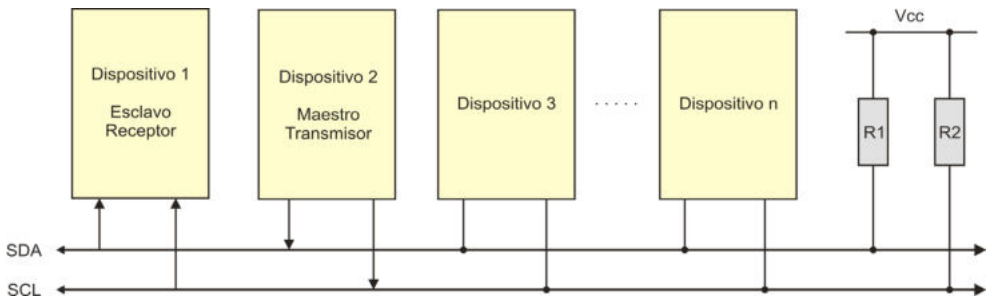


Figura 6.26 El dispositivo 1 recibe datos en el modo Esclavo Receptor

El MCU debe contar con una dirección a la cual debe responder como esclavo, esta dirección se define con los bits **TWA[6:0]**, los cuales corresponden con los 7 bits más significativos del registro **TWAR**. En el bit **TWGCE** (bit menos significativo de **TWAR**) se habilita al MCU para que también responda a una dirección de llamada general (GCA).

En el registro **TWCR** debe habilitarse a la interfaz TWI (**TWEN** = 1) y preparar una respuesta de reconocimiento (**TWEA** = 1), para ello, en este registro se debe escribir el valor:

TWCR	TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	-	TWIE
Valor	0	1	0	0	0	1	0	X

Con los registros **TWAR** y **TWCR** inicializados, la interfaz queda en espera de ser direccionada por su dirección de esclavo (o por una GCA, si fue habilitada) seguida por el bit de control del flujo de datos. La interfaz opera en el modo SR si el bit de control es 0 (*Write*), en caso contrario, entra al modo ST. Después de recibir su dirección, la bandera **TWINT** es puesta en alto y en los bits de estado del registro **TWSR** se refleja el código que determina las acciones a seguir por software. La interfaz también pudo ser llevada al modo SR si perdió una arbitración mientras estaba en modo Maestro.

En la tabla 6.12 se muestran los estados posibles en el modo SR, con una descripción del estado de la interfaz y las posibles acciones a seguir. Para todos los códigos se asume un enmascaramiento de los bits del pre-escalador (**TWPS**), de manera que se consideran como ceros.

Tabla 6.12 Estados posibles en el modo Esclavo Receptor

Código de Estado	Estado del bus y de la interfaz serial	Posibles acciones a realizar en la interfaz, en respuesta al estado
0x60	Se ha direccionado como Esclavo con una SLA+W y enviado un ACK	1. Recibir un byte de datos y regresar un ACK (**TWEA** = 1) 2. Recibir un byte de datos y regresar un nACK (**TWEA** = 0)
0x68	Se perdió una arbitración en una SLA+R/W como Maestro, se ha direccionado como esclavo con una SLA+W y enviado un ACK	
0x70	Se ha direccionado como Esclavo con una GCA y enviado un ACK	
0x78	Se perdió una arbitración en una SLA+R/W como Maestro, se ha direccionado como esclavo con una GCA y enviado un ACK	
0x80	Se ha recibido un byte de datos y respondido con un ACK, previamente se había direccionado con una SLA+W	
0x90	Se ha recibido un byte de datos y respondido con un ACK, previamente se había direccionado con una GCA	
0x88	Se ha recibido un byte de datos y respondido con un nACK, previamente se había direccionado con una SLA+W	1. Conmutar a un modo de Esclavo no direccionado, desactivando la interfaz para no reconocer su propia SLA o la GCA (**TWEA** = 0) 2. Conmutar a un modo de Esclavo no direccionado, capaz de reconocer su propia SLA o la GCA (**TWEA** = 1) 3. Conmutar a un modo de Esclavo no direccionado, desactivando la interfaz para no reconocer su propia SLA o la GCA y enviar un bit de inicio, cuando el bus esté disponible 4. Conmutar a un modo de Esclavo no direccionado, capaz de reconocer su propia SLA o la GCA y enviar un bit de inicio, cuando el bus esté disponible
0x98	Se ha recibido un byte de datos y respondido con un nACK, previamente se había direccionado con una GCA	
0xA0	Se ha recibido una condición de PARO o de INICIO REPETIDO, mientras estaba direccionado como Esclavo	

Si el bit **TWEA** es reiniciado durante una transferencia, la interfaz coloca un nACK en SDA después de recibir el próximo dato. Esto se puede hacer para que un Esclavo indique que no le es posible recibir más datos. Con el bit **TWEA** se puede aislar temporalmente a la interfaz del bus, con un 0 no reconoce su SDA o la GCA, pero el monitoreo continúa realizándose, de manera que puede ocurrir un reconocimiento tan pronto como **TWEA** es puesta en alto.

El reloj del sistema es omitido en algunos modos de reposo, sin embargo, si el bit **TWEA** está en alto, la interfaz TWI va a reconocer su SLA o a la GCA utilizando al reloj del bus (SCL) como mecanismo para su sincronización. Con ello, la interfaz "despierta" al MCU aunque la señal SCL se mantenga en bajo en espera de que la bandera **TWINT** sea limpiada. Esto hace que en el registro **TWDR** no se refleje el último byte presente en el bus. Una vez que el MCU está activo, las transferencias siguientes emplean al reloj del sistema. Si el MCU tiene un tiempo de ajuste largo, durante ese tiempo la señal SCL se mantiene en bajo y se detienen las transferencias en el bus.

6.3.5.4 Modo Esclavo Transmisor

Un MCU en el modo ST transmite una cantidad de bytes a un MCU en el modo MR, esto se muestra en la figura 6.27.

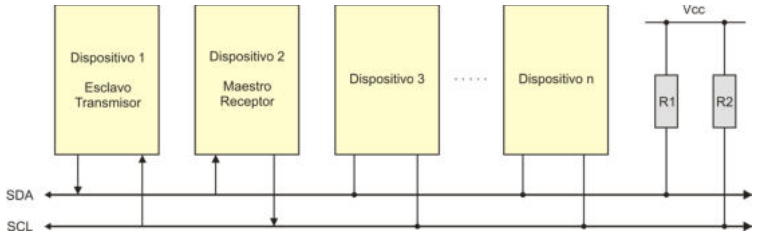


Figura 6.27 El dispositivo 1 envía datos en el modo Esclavo Transmisor

El MCU debe contar con una dirección a la cual va a responder como esclavo, esta dirección se define con los bits **TWA[6:0]**, los cuales corresponden con los 7 bits más significativos del registro **TWAR**. En el bit **TWGCE** (bit menos significativo de **TWAR**) se habilita al MCU para que también responda a una dirección de llamada general (GCA).

En el registro **TWCR** debe habilitarse a la interfaz TWI (**TWEN** = 1) y preparar una respuesta de reconocimiento (**TWEA** = 1), para ello, en este registro se debe escribir el valor:

TWCR	TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	-	TWIE
Valor	0	1	0	0	0	1	0	X

Con los registros **TWAR** y **TWCR** inicializados, la interfaz queda en espera de ser direccionada por su dirección de esclavo (o por una GCA, si fue habilitada, aunque en una llamada general no se solicita a los esclavos transmitir datos, porque se provocaría una colisión en el bus) seguida por el bit de control del flujo de datos. La interfaz opera en el modo ST si el bit de control es 1 (*Read*), en caso contrario, entra al modo SR. Después de recibir su dirección, la bandera **TWINT** es puesta en alto y en los bits de

estado del registro **TWSR** se refleja el código que determina las acciones a seguir por software. La interfaz también pudo ser llevada al modo ST si perdió una arbitración mientras estaba en modo Maestro.

En la tabla 6.13 se muestran los estados posibles en el modo ST, con una descripción del estado de la interfaz y las posibles acciones a seguir. Para todos los códigos se asume un enmascaramiento de los bits del pre-escalador (**TWPS**), de manera que se consideran como ceros.

Si el bit **TWEA** es limpiado, la interfaz transmite el último dato y el MCU conmuta a un esclavo sin direccionar. En los bits de estado se puede obtener 0xC0 ó 0xC8, dependiendo de si el Maestro Receptor transmitió un ACK o un nACK después de recibir el último dato. Si el Maestro intenta continuar con las transferencias, éstas van a ignorarse. El Maestro podría demandar más datos generando señales de ACK, en cuyo caso recibiría 1's en la línea de datos.

Tabla 6.13 Estados posibles en el modo Esclavo Transmisor

Código de Estado	Estado del bus y de la interfaz serial	Posibles acciones a realizar en la interfaz, en respuesta al estado
0xA8	Se ha direccionado como Esclavo con una SLA+R y enviado un ACK	1. Transmitir un byte de datos y recibir un ACK (**TWEA** = 1) 2. Transmitir un byte de datos y recibir un nACK (**TWEA** = 0)
0xB0	Se perdió una arbitración en una SLA+R/W como Maestro, se ha direccionado como esclavo con una SLA+R y enviado un ACK	
0xB8	Se transmitió un byte de datos en **TWDR** y se recibió un ACK	
0xC0	Se transmitió un byte de datos en **TWDR** y se recibió un nACK	1. Conmutar a un modo de Esclavo no direccionado, desactivando la interfaz para no reconocer su propia SLA o la GCA (**TWEA** = 0) 2. Conmutar a un modo de Esclavo no direccionado, capaz de reconocer su propia SLA o la GCA (**TWEA** = 1) 3. Conmutar a un modo de Esclavo no direccionado, desactivando la interfaz para no reconocer su propia SLA o la GCA y enviar un bit de inicio, cuando el bus esté disponible 4. Conmutar a un modo de Esclavo no direccionado, capaz de reconocer su propia SLA o la GCA y enviar un bit de inicio, cuando el bus esté disponible
0xC8	Se transmitió el último byte de datos en **TWDR** (TWEA = 0) y se recibió un ACK	

Con el bit **TWEA** se puede aislar temporalmente a la interfaz del bus, con un 0 no reconoce su SDA o la GCA, pero el monitoreo continúa realizándose, de manera que puede ocurrir un reconocimiento tan pronto como **TWEA** es puesta en alto.

6.3.5.5 Estados Misceláneos

La interfaz TWI incluye 2 códigos que no corresponden con alguno de los 4 modos de operación.

Estos estados se describen en la tabla 6.14, en donde se asume un enmascaramiento de los bits del pre-escalador (**TWPS**), de manera que se consideran como ceros.

Tabla 6.14 Estados misceláneos

Código de Estado	Estado del bus y de la interfaz serial	Posibles acciones a realizar en la interfaz, en respuesta al estado
0xF8	No hay información relevante disponible, **TWINT** = 0	1. Esperar o proceder con la siguiente transferencia
0x00	Error en el bus, debido a una condición ilegal de INICIO o PARO	1. Sólo el hardware interno es afectado. El bit **TWSTO** debe ser puesto en alto, pero no se envía una condición de PARO en el bus. El bus es liberado y el bit **TWSTO** es limpiado.

Un error en el bus puede deberse a una posición ilegal durante la transferencia de un byte de dirección, un byte de datos o un bit de reconocimiento. **TWINT** es puesta en alto cuando ocurre un error. Para recuperar al bus, la bandera **TWSTO** debe ser puesta en alto y la bandera **TWINT** en bajo. Con esto, la interfaz entra a un modo de Esclavo no direccionado y limpia a la bandera **TWSTO**. Las líneas del bus, SDA y SCL, son liberadas y no se transmite una condición de paro.

6.3.6 Ejemplos de Uso de la Interfaz TWI

Los modos de operación de la interfaz TWI le proporcionan a los AVR una capacidad extraordinaria para el desarrollo de sistemas "inteligentes" conectados como nodos en una red de 2 hilos, en donde cada MCU puede conmutarse, en tiempo de ejecución, entre los modos Maestro y Esclavo. En esta sección se muestran 2 ejemplos del uso de la interfaz TWI, codificados en lenguaje C. No obstante, las aplicaciones posibles van mucho más allá de los ejemplos acá descritos.

Ejemplo 6.9 Repita el ejemplo 6.8 utilizando la interfaz TWI. Se trata de un maestro y 3 esclavos, el Maestro tiene 2 arreglos de interruptores (uno con 8 y otro con 2 interruptores) y un botón. El arreglo de 2 interruptores es para seleccionar un Esclavo y el de 8 para introducir un dato. Con ello, cada vez que se presiona al botón, debe enviarse el dato al Esclavo seleccionado.

Las direcciones para los esclavos son 1, 2 y 3, y utilice la dirección 0 para una llamada general. Cuando un Esclavo reciba un dato, lo debe mostrar en su puerto D.

El hardware para este problema se muestra en la figura 6.28, el Maestro incluye los interruptores y el botón, cada Esclavo incluye su conjunto de LEDs. Además de incluir los resistores de *Pull-Up* con un valor de 2 Kohms.

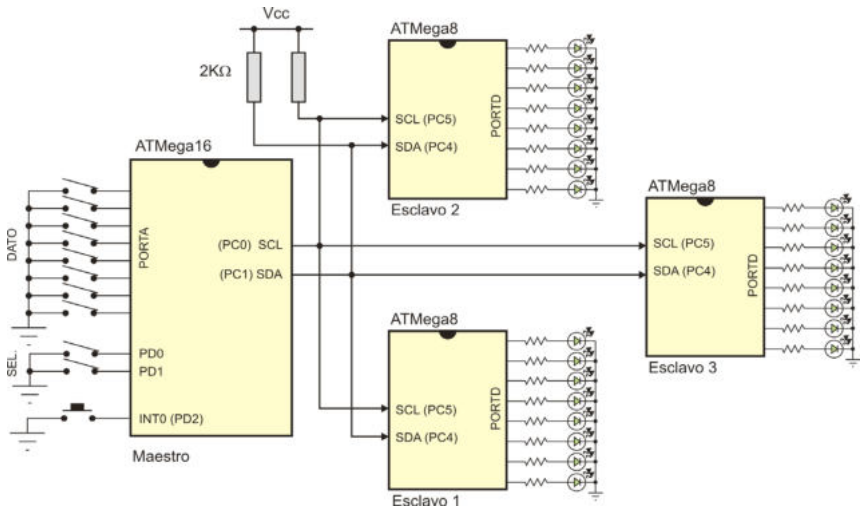


Figura 6.28 Envío de información de un Maestro a 3 Esclavos por TWI

En el Maestro se utiliza la interrupción externa (INT0), en la ISR de la INT0 se habilita a la interfaz TWI con su interrupción y se transmite una condición de inicio. Posteriormente, toda la actividad del bus es manejada en la ISR de la interfaz TWI, en donde se evalúan los bits de estado, para dar respuesta a sus diferentes valores.

El código en lenguaje C para el Maestro es:

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

ISR(INT0_vect) {

// Habilita a la interfaz TWI, su interrupción y transmite una condición
de inicio
       TWCR = (1 << TWINT) | (1 << TWSTA) | (1 << TWEN) | (1 << TWIE);
}

ISR(TWI_vect) {
unsigned  char  estado, esclavo, dato;

estado = TWSR & 0xFC;      // Obtiene el estado de la interfaz TWI

switch(estado) {                          // Actúa según el estado
   case  0x08:                            // Se transmitió la condición de
                                          // inicio
               esclavo = PIND & 0x03;     // Lee la dirección del esclavo
               esclavo = esclavo << 1;    // Ajusta a los 7 MSB y
                                          // compone la SLA+W
```

```
                TWDR = esclavo;             // Ubica la SLA+W
                TWCR = (1 << TWINT) | (1 << TWEN) | (1 << TWIE);
                break;
    case  0x18:                             // Transmitió la SLA+W con
                                            // reconocimiento
                dato = PINA;
                TWDR = dato;                // Prepara el dato a enviar
                TWCR = (1 << TWINT) | (1 << TWEN) | (1 << TWIE);
                break;
    default:   //No hubo ACK con la SLA+W o trasmitió el dato con un ACK o un nACK
               // Genera la condición de paro
        TWCR = (1 << TWINT) | (1 << TWSTO) | (1 << TWEN) | (1 << TWIE);
                break;
    }
}

int  main() {                       // Programa principal

DDRA = 0x00;                        // Puerto A como entrada
DDRD = 0x00;                        // Puerto D como entrada
PORTA = 0xFF;                       // Resistores de pull-up en el puerto A
PORTD = 0x07;                       // Resistores de pull-up en el puerto D

TWBR = 0x02;                        // Para definir la frecuencia de reloj
TWSR = 0x00;                        // Pre-escalador en 0 (factor de pre-escala = 1)
                                    // para una frecuencia de 31.25 KHz
                                    // (fosc=1 MHz)
MCUCR = 0x02;                       // INT0 por flanco de bajada
GICR = 0x40;                        // Habilita la INT0

sei();                              // Habilitador global de interrupciones

while(1)                            // Ocioso en el lazo infinito
       asm("nop");
}
```

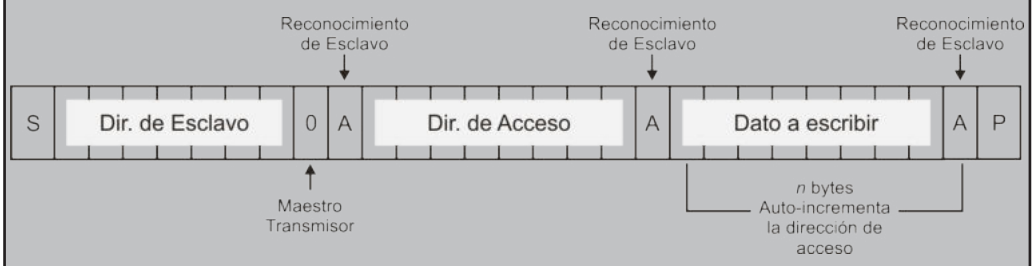
En los esclavos también se utiliza la interrupción por TWI, en su ISR se conoce el estado del bus, al cual se le da respuesta. Cada esclavo debe tener su propia dirección, el código siguiente es para un esclavo con dirección 0x01, para cambiar la dirección sólo se debe modificar la constante.

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

#define DIR_ESLVO   0x01

ISR(TWI_vect) {
unsigned  char  dato, estado;

estado = TWSR & 0xFC;               // Obtiene el estado de la interfaz TWI

switch(estado) {
```

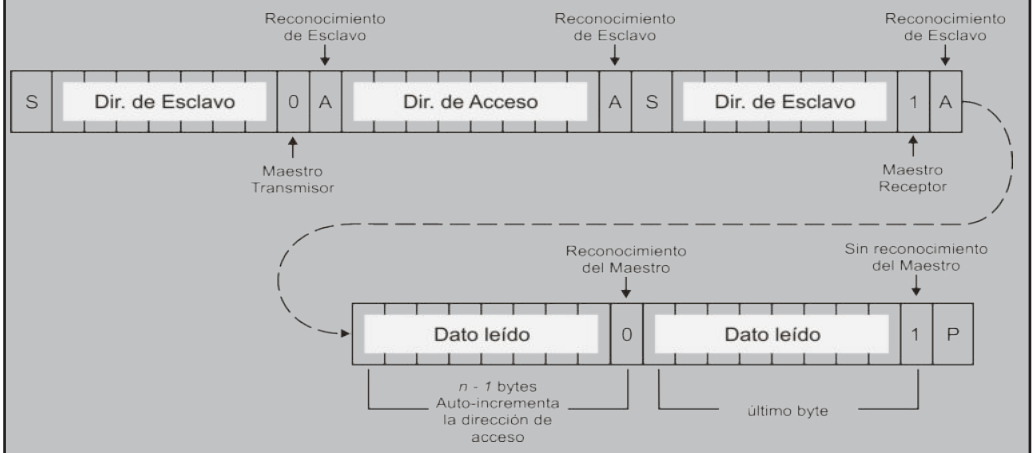
```
        case 0x60:                  // Esclavo direccionado con su SLA o la GCA
        case 0x70:    TWCR = (1 << TWINT) | (1 << TWEA) | (1 << TWEN) |
                      (1 << TWIE);
                      break;
        case 0x80:                  // Recibió un dato, previamente direccionado
        case 0x90:                  // con su SLA o  la GCA
                      dato = TWDR;  // Obtiene el dato recibido
                      PORTD = dato;
                      TWCR = (1 << TWINT) | (1 << TWEA) | (1 << TWEN) |
                      (1 << TWIE);
                      break;
        default:                    // Situación no esperada
                      TWCR = (1 << TWINT) | (1 << TWEA) | (1 << TWEN) |
                      (1 << TWIE);
                      break;
        }
}


int  main() {                       // Programa principal
unsigned  char  dir;


DDRD = 0xFF;                        // Puerto D como salida

dir = DIR_ESLVO << 1;               // Asigna la dirección del esclavo
dir = dir | 0x01;                   // habilitado para reconocer a la GCA
TWAR = dir;

// Habilita a la interfaz TWI, su respuesta de reconocimiento y su interrupción

TWCR = (1 << TWEA) | (1 << TWEN) | (1 << TWIE);

sei();                              // Habilitador global de interrupciones
while(1)                            // Ocioso en el lazo infinito
        asm("nop");
}
```

Se observa que la difusión hacia todos los esclavos es parte de la interfaz.

Ejemplo 6.10 El circuito PCF8570 es una memoria RAM de 256 x 8 bits, con interfaz I^2C y dirección configurable entre 0x50 y 0x57 (con 3 terminales). Esta memoria trabaja en los modos Esclavo Receptor (para escritura de datos) y Esclavo Transmisor (para lectura de datos). En la figura 6.29 (a) se muestra como escribir datos y en la 6.29 (b) puede verse como leerlos.



(a) El Maestro transmite y el esclavo recibe



b) El Maestro lee después de detectar la presencia del Esclavo

Figura 6.29 (a) Escritura de datos (b) Lectura de datos

Considerando una memoria PCF8570 con la dirección de Esclavo 0x50, conectada a la interfaz TWI de un AVR, realice una función que escriba *n* bytes en la memoria, recibiendo un arreglo con los datos, la cantidad de datos a escribir y la dirección donde deben iniciar las escrituras; y una función que lea *n* bytes de la memoria, recibiendo un arreglo para colocar los datos, la cantidad de datos a leer y la dirección donde deben iniciar las lecturas.

Ambas funciones regresan 0x01 si el acceso a la memoria se realizó con éxito y 0x00 si no hubo reconocimiento del esclavo o si ocurrió una falla durante el proceso. No se configura la frecuencia, para el programa completo debe tomarse en cuenta que la memoria puede operar con una frecuencia máxima de 100 KHz y tampoco se utilizan interrupciones.

El código de la función para la escritura de datos en la RAM es:

```
char  escribe_RAM(char datos[], unsigned char n, unsigned char dir) {
unsigned char i, estado;
```

```
// Habilita a la interfaz TWI y transmite una condición de INICIO
TWCR = (1 << TWINT) | (1 << TWSTA) | (1 << TWEN);
while(!(TWCR & (1 << TWINT)));          // Espera fin de condición de INICIO
estado = TWSR & 0xFC;      // Obtiene el estado de la interfaz TWI
if( estado != 0x08) {                   // Falló al intentar tomar el bus
      TWCR = (1 << TWINT) | (1 << TWEN);      // Limpia la bandera
      return  0x00;                     // Regresa sin éxito
}

TWDR = 0xA0;                            // Ubica la SLA+W (0x50 << 1 + 0)
TWCR = (1 << TWINT) | (1 << TWEN);      // Transmite la SLA+W
while(!(TWCR & (1 << TWINT)));
estado = TWSR & 0xFC;      // Obtiene el estado de la interfaz TWI
if( estado != 0x18) {                   // Falló al enviar la SLA+W
      TWCR = (1 << TWINT) | (1 << TWSTO) | (1 << TWEN);
                                        // Condición de PARO
      return  0x00;                     // Regresa sin éxito
}

TWDR = dir;                             // Transmite la dirección de acceso
TWCR = (1 << TWINT) | (1 << TWEN);
while(!(TWCR & (1 << TWINT)));          // Espera finalice el envío
estado = TWSR & 0xFC;     // Obtiene el estado de la  interfaz TWI
if( estado != 0x28) {                   // Falló al enviar un dato
      TWCR = (1 << TWINT) | (1 << TWSTO) | (1 << TWEN);
                                        // Condición de PARO
      return  0x00;                     // Regresa sin éxito
}

for( i = 0; i < n; i++ ) {                    // Transmite los datos
      TWDR = datos[i];                        // Ubica dato a transmitir
      TWCR = (1 << TWINT) | (1 << TWEN);      // Inicia el envío
      while(!(TWCR & (1 << TWINT)));          // Espera finalice el envío
      estado = TWSR & 0xFC; // Obtiene el estado de  la interfaz
      if( estado != 0x28) {                   // Falló al enviar un dato
            TWCR = (1 << TWINT) | (1 << TWSTO) | (1 << TWEN);// PARO
            return  0x00;                     // Regresa sin éxito
      }
}
TWCR = (1 << TWINT) | (1 << TWSTO) | (1 << TWEN);     // Condición de PARO
return  0x01;                           // Regresa con éxito
}
```

El código de la función para la lectura de datos en la RAM es:

```
char   lee_RAM(char datos[], unsigned char n, unsigned char dir) {
unsigned char i, estado;

// Habilita a la interfaz TWI y transmite una condición de INICIO
TWCR = (1 << TWINT) | (1 << TWSTA) | (1 << TWEN);
while(!(TWCR & (1 << TWINT)));    // Espera fin de condición de INICIO
estado = TWSR & 0xFC;      // Obtiene el estado de la interfaz TWI
if( estado != 0x08) {             // Falló al intentar tomar el bus
```

```
        TWCR = (1 << TWINT) | (1 << TWEN);// Limpia la bandera
        return  0x00;                     // Regresa sin éxito
}
TWDR = 0xA0;                        // Ubica la SLA+W (0x50 << 1 + 0)
TWCR = (1 << TWINT) | (1 << TWEN);      // Transmite la SLA+W
while(!(TWCR & (1 << TWINT)));
estado = TWSR & 0xFC;      // Obtiene el estado de la interfaz TWI
if( estado != 0x18) {                   // Falló al enviar la SLA+W
        TWCR = (1 << TWINT) | (1 << TWSTO) | (1 << TWEN);
                                        // Condición de PARO
        return  0x00;                   // Regresa sin éxito
}

TWDR = dir;                         // Transmite la dirección de acceso
TWCR = (1 << TWINT) | (1 << TWEN);
while(!(TWCR & (1 << TWINT)));          // Espera finalice el envío
estado = TWSR & 0xFC;      // Obtiene el estado de la interfaz TWI
if( estado != 0x28) {                   // Falló al enviar un dato
        TWCR = (1 << TWINT) | (1 << TWSTO) | (1 << TWEN);
                                        // Condición de PARO
        return  0x00;                   // Regresa sin éxito
}
// Transmite una condición de INICIO (INICIO REPETIDO)
TWCR = (1 << TWINT) | (1 << TWSTA) | (1 << TWEN);
while(!(TWCR & (1 << TWINT)));    // Espera fin de condición de INICIO
estado = TWSR & 0xFC;      // Obtiene el estado de la interfaz TWI
if( estado != 0x10) {             // Falló con el inicio repetido
        TWCR = (1 << TWINT) | (1 << TWEN);      // Limpia la bandera
        return  0x00;                   // Regresa sin éxito
}

TWDR = 0xA1;                                // Ubica la SLA+R (0x50 << 1 + 1)
TWCR = (1 << TWINT) | (1 << TWEN);      // Transmite la SLA+R
while(!(TWCR & (1 << TWINT)));
estado = TWSR & 0xFC;         // Obtiene el estado de la interfaz TWI
if( estado != 0x40) {                   // Falló al enviar la SLA+R
        TWCR = (1 << TWINT) | (1 << TWSTO) | (1 << TWEN);
                                       // Condición de PARO
        return  0x00;                  // Regresa sin éxito
}
for( i = 0; i < n - 1; i++) {           // Recibe n - 1 datos
        TWCR = (1 << TWINT) | (1 << TWEA)| (1 << TWEN);
                                        // Habilita respuesta
        while(!(TWCR & (1 << TWINT)));    // Espera dato del esclavo
        estado = TWSR & 0xFC;      // Obtiene el estado de la interfaz
        if( estado != 0x50) {             // Falló al recibir un dato
                TWCR = (1 << TWINT) | (1 << TWSTO) | (1 << TWEN);// PARO
                return  0x00;             // Regresa sin éxito
        }
        datos[i] = TWDR;                  // Dato recibido con éxito
}
```

```
TWCR = (1 << TWINT) | (1 << TWEN);       // Recibe el último dato
                                         // (sin el bit de reconocimiento)
while(!(TWCR & (1 << TWINT)));           // Espera dato del esclavo
estado = TWSR & 0xFC;              // Obtiene el estado de la interfaz
if( estado != 0x58) {                    // Falló al recibir un dato
      TWCR = (1 << TWINT) | (1 << TWSTO) | (1 << TWEN);
                                         // Condición de PARO
      return  0x00;                      // Regresa sin éxito
}
datos[i] = TWDR;                         // Último dato recibido con éxito

TWCR = (1 << TWINT) | (1 << TWSTO) | (1 << TWEN);// Condición de PARO
return  0x01;                                    // Regresa con éxito}
}
```

Un problema con las funciones del ejemplo 6.10 es que cuando no tienen éxito regresan con el valor 0x00, sin importar en que etapa del proceso se encontraban. Las funciones pueden mejorar si se establece un código para cada uno de los errores más frecuentes, de manera que por software se conozca el motivo del final de las transferencias.

6.4 Ejercicios

En los siguientes ejercicios se combina el uso de las interfaces seriales con los recursos revisados en los capítulos anteriores. Pueden ser resueltos en lenguaje C o ensamblador.

1. Haga una comparativa de las 3 interfaces seriales revisadas en el presente capítulo, mostrando las ventajas y desventajas que cada una de ellas tiene.

2. Realice un **sistema para el control de la temperatura ambiente**, generando una señal PWM de 10 bits para activar un ventilador de DC. La temperatura de referencia se debe recibir por el puerto serie (USART), como un número entero (1 byte). Si la diferencia de la temperatura actual (obtenida de un sensor LM35) con la temperatura de referencia es mayor o igual a 3 °C, la señal de activación del ventilador debe tener un ciclo útil del 100 %. Si la diferencia está entre 0 °C y 3 °C, al ancho de pulso debe ser proporcional a esta diferencia. Cuando la temperatura actual está por debajo de la referencia, el ventilador debe estar apagado. La temperatura de referencia puede arribar en cualquier momento.

3. Construya una **marquesina para exhibir mensajes, con 5 matrices de 5 x 7 LEDs**. Utilice una red SPI con 1 Maestro y 5 Esclavos, todos ellos con base en microcontroladores ATMega8. Que cada matriz sea manejada por un Esclavo, quienes deben recibir el

código ASCII del carácter a mostrar, generando los puntos en su respectiva matriz, con base en una tabla de constantes. El usuario va a enviar la cadena a exhibir y su tamaño desde una PC al Maestro, a través de la USART. Para un desplazamiento del mensaje de derecha a izquierda, el Maestro continuamente debe modificar el carácter de cada Esclavo. Un Esclavo básicamente va a funcionar como un driver para una matriz de 5 x 7 LEDs, con una interfaz SPI para recibir el carácter a exhibir.

- Proponer el hardware.
- Desarrollar el programa del maestro y de los esclavos (es el mismo para los 5 esclavos).

4. Repita el ejercicio anterior, pero utilice la interfaz TWI en lugar de emplear a la interfaz SPI. ¿Cuál de las dos interfaces (SPI y TWI) considera es más adecuada para este problema? Justifique su respuesta.

5. Se requiere de un **sistema para votaciones**, capaz de manejar hasta 100 votantes. Diseñe el sistema de manera que los votantes dispongan de un circuito con un indicador luminoso (LED) y 5 botones. Con el indicador luminoso encendido, los votantes deben elegir alguna opción, presionando uno de los botones.

   Organice el sistema con base en una red TWI, donde los circuitos de votación sean los Esclavos y que el Maestro sea el colector de los votos. Después de cada elección, el maestro debe enviar 5 pares ordenados a través de su USART. Cada par especifica cuantos votos se tuvieron en cada una de las opciones. Proponga el hardware y desarrolle el software, del maestro y de los esclavos.

6. El circuito DS1307 es un Reloj/Calendario de Tiempo Real, con interfaz I^2C y dirección de Esclavo 0x68 (1101000). El circuito trabaja en los modos Esclavo Receptor (para configurar la fecha y la hora) y Esclavo Transmisor (para leer la fecha y la hora), siguiendo secuencias similares a las mostradas en las figuras 6.29 (a) y en la 6.29 (b). En la tabla 6.15 se muestra la organización de los registros para la fecha y la hora, puede verse que la información está en BCD.

   Escriba las funciones para:

   - Ajustar la hora, recibiendo un arreglo de caracteres con los segundos, minutos y hora (esc_hora(**unsigned char** hora[])).
   - Leer la hora, colocando los datos en un arreglo (lee_hora(**unsigned char** hora[])).
   - Ajustar la fecha, recibiendo un arreglo de caracteres con el día de la semana, día del mes, mes y año (esc_fecha(**unsigned char** fecha[])).

- Leer la fecha, colocando los datos en un arreglo (lee_fecha(**unsigned char** fecha[])).

Tabla 6.15 Registros para mantener la fecha y hora

Dirección	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Función	Rango
00 h	CH	Decenas de segundos			Unidades de segundos				Segundos	00-59
01 h	0	Decenas de minutos			Unidades de minutos				Minutos	00-59
02 h	0	12	PM /AM	Dec. de hora	Unidades de horas				Horas	1-12 + AM/PM
		24	Dec. de hora		Unidades de horas					00-23
03 h	0	0	0	0	0	Día (semana)			Día (semana)	01-07
04 h	0	0	Dec. día (mes)		Unidades del día (mes)				Día (mes)	01-31
05 h	0	0	0	D. Mes	Unidades del mes				Mes	01-12
06 h	Decenas del año				Unidades del año				Año	00-99
07 h	OUT	0	0	SQWE	0	0	RS1	RS0	Control	-
08h-3Fh									RAM 56 x 8	00h-FFh

Respecto a la tabla 6.15, con el registro de control se configura al hardware para generar una señal cuadrada a diferentes frecuencias. En las funciones a desarrollar, ignore el registro y la memoria RAM disponible. El bit CH en la dirección 00h sirve para detener momentáneamente al reloj (*clock halt*), el valor de este bit no se debe modificar. Las funciones sólo se deben encargar del acceso al bus I^2C, las validaciones antes de una escritura o interpretaciones después de una lectura quedan fuera del contexto del problema, serían parte del programa principal..

7. Recursos Especiales

En este capítulo se describen 4 recursos de los microcontroladores AVR que pueden ser utilizados para complementar la funcionalidad de algunas aplicaciones, se trata del Perro Guardián (WDT, *watchdog timer*), la sección de arranque en la memoria de programa, los *Bits de Configuración y Seguridad*, y la interfaz JTAG, aunque ésta no se incluye en los ATMega8, sólo en los ATMega16.

7.1 Watchdog Timer de un AVR

El WDT es un temporizador que reinicia al MCU con su desbordamiento, es manejado por un oscilador interno, el cual tiene una frecuencia de 1 MHz cuando el MCU está alimentado con 5 V. En la figura 7.1 se muestra la organización del WDT, puede verse como a través de un pre-escalador es posible conseguir diferentes intervalos de tiempo, el intervalo se define con los bits **WDP[2:0]** del Registro de Control del WDT (**WDTCR**, *Watchdog Timer Control Register*). En la tabla 7.1 se muestran los intervalos de tiempo para las diferentes opciones del pre-escalador.

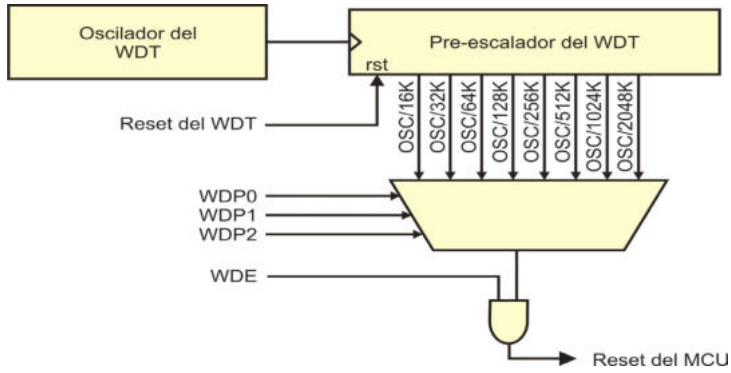


Figura 7.1 Organización del Watchdog Timer

Tabla 7.1 Factores de pre-escala del *Watchdog Timer*

WDP2	WDP1	WDP0	Ciclos	Tiempo (Vcc = 3.0 V)	Tiempo (Vcc = 5.0 V)
0	0	0	16K (16, 384)	17.1 mS	16.3 mS
0	0	1	32K (32, 768)	34.3 mS	32.5 mS
0	1	0	64K (65, 536)	68.5 mS	65 mS
0	1	1	128K (131, 072)	0.14 S	0.13 S
1	0	0	256K (262, 144)	0.27 S	0.26 S
1	0	1	512K (524, 288)	0.55 S	0.52 S
1	1	0	1, 024K	1.1 S	1.0 S
1	1	1	2, 048 K	2.2 S	2.1 S

En la tabla 7.1 se observa que el periodo es ligeramente menor cuando el voltaje de alimentación es de 5V, con respecto a un voltaje de 3.0 V. Si el periodo expira sin un reinicio al WDT, el MCU es reiniciado, ejecutando las instrucciones desde el vector de reset. La causa queda registrada en **MCUCSR.**

El WDT de un ATMega8 puede habilitarse en dos formas diferentes: Activando al fusible **WDTON**, que es parte de los *Bits de Configuración y Seguridad*, al momento de programar al dispositivo, o bien, modificando al bit WDE (*Watchdog Timer Enable)* del registro **WDTCR** durante la ejecución de una aplicación. Un ATMega16 no incluye al fusible **WDTON**, por lo tanto, la activación del WDT sólo se puede realizar con el bit **WDE** del registro **WDTCR**.

7.1.1 Registro para el Manejo del WDT

El registro de control del WDT es el **WDTCR**, cuyos bits son:

	7	6	5	4	3	2	1	0	
0x21	-	-	-	WDCE/ WDTOE	WDE	WDP2	WDP1	WDP0	**WDTCR**

- **Bits 7 al 5 – No están implementados**
- **Bit 4 – WDCE: Modifica la habilitación del WDT (ATMega8)/ WDTOE: Habilita la salida del WDT (ATMega16)**

  Este bit tiene un nombre diferente en los dispositivos bajo estudio, pero su funcionalidad es la misma. El bit es parte de un mecanismo de seguridad que evita la desactivación errónea del WDT, en un ATMega8 sólo si el WDT fue habilitado vía software. Para inhabilitar al WDT, este bit debe ponerse en alto y dentro de los siguientes 4 ciclos de reloj debe limpiarse al bit **WDE**. Pasado ese tiempo, el bit **WDCE/WDTOE** automáticamente es limpiado por hardware.

- **Bit 3 – WDE: Habilitador del WDT**

  Un 1 en este bit habilita al WDT y un 0 lo inhabilita. Sin embargo, la inhabilitación requiere que el bit **WDCE/WDTOE** esté en un nivel alto. La secuencia para inhabilitar al WDT es la siguiente:

  1. Con la misma operación, poner en alto a los bits **WDCE/WDTOE** y **WDE**. La escritura de **WDE** es parte de la secuencia, es necesaria aunque de antemano ya tenga un nivel alto.
  2. Dentro de los siguientes 4 ciclos de reloj escribir un 0 lógico en **WDE**, con ello el WDT ha quedado inhabilitado.

  Si en un ATMega8 el WDT fue habilitado activando al fusible **WDTON**, no se puede inhabilitar aun siguiendo la secuencia anteriormente descrita. En estos casos, la secuencia es útil para modificar el periodo de desbordamiento del WDT.

- **Bits 2 al 0 – WDP[2:0]: Bits para la selección del factor de pre-escala del WDT**

  Con estos bits se define el factor de pre-escala para alcanzar diferentes intervalos de tiempo, antes de un desbordamiento del WDT. En la tabla 7.1 se mostraron los intervalos de tiempo que se consiguen con las diferentes combinaciones de estos bis. La modificación de estos bits también requiere la secuencia de seguridad anteriormente descrita, excepto que en el paso 2 se involucra la modificación de los bits **WDP[2:0]**.

Ejemplo 7.1 Codifique una rutina o función para detener al WDT, en lenguaje ensamblador y en lenguaje C, enfocada a un ATMega8.

La rutina en lenguaje ensamblador es:

```
WDT_Off:
       WDR                                          ; Reinicia al WDT
       IN     R16, WDTCR
       ORI    R16, (1 << WDCE) | (1 << WDE)        ; 1 lógico en WDCE y en WDE
       OUT    WDTCR, R16
       CLR    R16
       OUT    WDTCR, R16                           ; Apaga al WDT
       RET
```

La función en lenguaje C es:

```
void WDT_Off(void) {
       asm("WDR");                                  // Reinicia al WDT

       WDTCR = WDTCR | (1 << WDCE) | (1 << WDE);// 1 lógico en
                                                    //  WDCE y en WDE

       WDTCR = 0x00;                                // Apaga al WDT
}
```

El WDT se reinicia para evitar que desborde mientras se ejecuta la rutina. Para un ATMega16 es suficiente con remplazar al bit **WDCE** con el bit **WDTOE**.

7.2 Sección de Arranque en la Memoria de Programa

La memoria Flash está particionada en dos secciones, una de aplicación y otra de arranque. En la mayoría de sistemas no se considera esta partición y se dedica todo el espacio a la sección de aplicación. Aunque en la sección de arranque podría ubicarse una secuencia de código que corresponda con una aplicación ordinaria, la sección está orientada para que el usuario pueda ubicar un cargador, es decir, un pequeño programa que facilite modificar una parte o toda la sección de aplicación. Esto significa que los microcontroladores AVR incluyen los mecanismos de hardware necesarios para hacer que una aplicación pueda ser actualizada por sí misma, realizando una autoprogramación.

7.2.1 Organización de la Memoria Flash

La memoria Flash está organizada en páginas, en un ATMega8 cada página es de 32 palabras de 16 bits y en un ATMega16 las páginas son de 64 palabras de 16 bits. Toda la memoria, considerando la sección de aplicación y la sección de arranque, está dividida en páginas. Ambos dispositivos, ATMega8 y ATMega16, incluyen 128 páginas en su memoria Flash. En la figura 7.2 se esquematiza esta organización. El acceso para el borrado y la escritura en la memoria Flash implica la modificación de una página completa. La lectura es diferente, ésta si puede realizarse con un acceso por bytes.

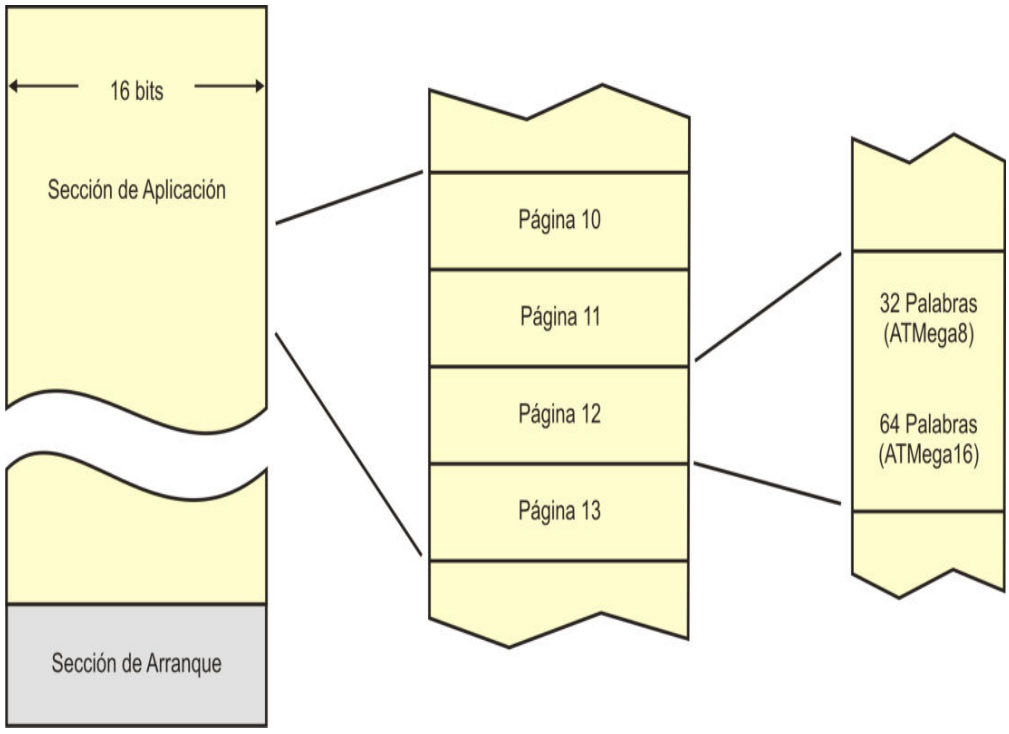


Figura 7.2 Organización de la memoria flash en páginas

La división de la memoria Flash se realiza con la programación de los fusibles **BOOTSZ1** y **BOOTSZ0**, los cuales son parte de los Bits de Configuración y Seguridad, con ellos se determina cuantas páginas son destinadas a la sección de arranque. En la tabla 7.2 se pueden ver las páginas destinadas para cada una de las secciones en un ATMega8 y en un ATMega16, en función del valor programado en estos fusibles. Un fusible tiene un 1 si está sin programar, los dispositivos son comercializados con **BOOTSZ[1:0]** = "00", es decir, los fusibles están programados para proporcionar el espacio máximo a la sección de arranque.

Tabla 7.2 División de la memoria en una sección de aplicación y una de arranque

BOOTSZ1	BOOTSZ0	Tamaño (Sección de arranque)	Páginas (ATMega8/ ATMega16)	Sección de Aplicación (ATMega8/ATMega16)	Sección de Arranque (ATMega8/ATMega16)
1	1	128 palabras	4 / 2	0x000 – 0xF7F / 0x0000 – 0x1F7F	0xF80 – 0xFFF/ 0x1F80 – 0x1FFF
1	0	256 palabras	8 / 4	0x000 – 0xEFF / 0x0000 – 0x1EFF	0xF00 – 0xFFF / 0x1F00 – 0x1FFF
0	1	512 palabras	16 / 8	0x000 – 0xDFF / 0x0000 – 0x1DFF	0xE00 – 0xFFF / 0x1E00 – 0x1FFF
0	0	1024 palabras	32 / 16	0x000 – 0xBFF / 0x0000 – 0x1BFF	0xC00 – 0xFFF / 0x1C00 – 0x1FFF

Una aplicación puede tener acceso al código escrito en la sección de arranque, por medio de llamadas a rutinas o saltos, pero si se programa al fusible **BOOTRST**, se consigue que después de un reset el CPU ejecute las instrucciones desde la sección de arranque y no desde la dirección 0.

En la memoria flash también se encuentran los vectores de las interrupciones, en las tablas 2.1 y 2.2 se mostraron las direcciones que les corresponden en un ATMega8 y en un ATMega16. No obstante, los vectores pueden moverse a la sección de arranque, esto se realiza con los bits **IVSEL** e **IVCE** del registro general para el control de interrupciones (**GICR**), correspondiendo con sus 2 bits menos significativos:

	7	6	5	4	3	2	1	0	
0x3B	INT1	INT0	INT2	-	-	-	IVSEL	IVCE	**GICR**

- **Bit 1 – IVSEL: Selecciona la ubicación de los vectores de las interrupciones**
  Cuando **IVSEL** tiene un 0, los vectores de las interrupciones se ubican al inicio de la memoria Flash. Si el bit **IVSEL** tiene un 1, los vectores son desplazados al inicio de la sección de arranque, cuya dirección depende de los fusibles **BOOTSZ1** y **BOOTSZ0**, como se mostró en la tabla 7.2.

- **Bit 0 – IVCE: Habilita el cambio en la ubicación de los vectores de las** interrupciones

  Para evitar cambios no deseados en la ubicación de los vectores de las interrupciones, cualquier ajuste requiere la puesta en alto del bit **IVCE** y dentro de los 4 ciclos de reloj siguientes se debe escribir el valor deseado en **IVSEL**, con la escritura de un 0 en **IVCE**.

La programación del fusible **BOOTRST** es independiente del estado del bit **IVSEL**, esto significa que aunque un programa reinicie en la dirección 0 de la memoria flash, sus vectores de interrupciones podrían estar colocados al inicio de la sección de arranque, o bien, un programa podría iniciar su ejecución en la sección de arranque, conservando los vectores de interrupciones al inicio de la memoria Flash.

Además, el fusible **BOOTRST** se programa durante la descarga del código en el dispositivo, mientras que el estado del bit **IVSEL** se define durante la ejecución del programa. Por lo tanto, independientemente de la dirección de inicio, los vectores de las interrupciones pueden moverse en tiempo de ejecución.

7.2.2 Acceso a la Sección de Arranque

En el ejemplo 7.2 se ilustra el acceso a la sección de arranque, sin considerar la posibilidad de un cargador para autoprogramación, se muestra cómo un programa inicia en la sección de arranque y, si se conmuta una entrada, cómo el programa pasa a ejecutar el código ubicado en la sección de aplicación.

Ejemplo 7.2 Realice un programa para un ATMega8 que en la sección de arranque haga parpadear un LED ubicado en la terminal PB0, y en la sección de aplicación haga parpadear a otro LED, ubicado en PB1. El programa debe abandonar la sección de arranque cuando se presione un botón conectado en INT0.

Para que el programa comience su ejecución en la sección de arranque, el fusible **BOOTRST** debe activarse durante la programación del dispositivo. Si los fusibles **BOOTSZ[1:0]** se mantienen con "00", quedan disponibles 32 páginas para la sección de arranque (iniciando en 0xC00).

La solución se desarrolla en lenguaje ensamblador, dado que en lenguaje C no es posible establecer la dirección para ubicar el código generado, el programa con la solución es:

```
       .org   0x000              ; Sección de aplicación
inicio:
       LDI    R16, 0x04          ; Ubica al apuntador de pila
       OUT    SPH, R16
       LDI    R16, 0x5F
       OUT    SPL, R16
       LDI    R16, 0xFF          ; Puerto B como salida
       OUT    DDRB, R16
Lazo:                            ; Lazo infinito
       SBI    PORTB, 1           ; Parpadeo en PB1
       RCALL  Espera_500mS
       CBI    PORTB, 1
       RCALL  Espera_500mS
       RJMP   Lazo

       .org   0xC00              ; Sección de arranque
       RJMP   ini_boot
       .org   0xC01              ; Vector de la INT0 desplazado
       RJMP   inicio             ; Brinca a la sección de aplicación
                                 ; Una instrucción RETI sería ignorada
ini_boot:
       LDI    R16, 0x01          ; Mueve los vectores de interrupciones
       OUT    GICR, R16          ; Para detectar la INT0 desde la
       LDI    R16, 0x02          ; sección de arranque
       OUT    GICR, R16

       CLR    R16                ; Puerto D como entrada
       OUT    DDRD, R16
       LDI    R16, 0xFF
       OUT    DDRB, R16          ; Puerto B como salida
```

```
        OUT     PORTD, R16          ; Pull-Up en el puerto D

        LDI     R16, 0x04           ; Ubica al apuntador de pila
        OUT     SPH, R16
        LDI     R16, 0x5F
        OUT     SPL, R16

        LDI     R16, 0x02           ; Configura la INT0 por flanco de bajada
        OUT     MCUCR, R16
        LDI     R16, 0x40           ; Habilita la INT0
        OUT     GICR, R16
        SEI                         ; Habilitador global de interrupciones
Lazo_boot:                          ; Lazo infinito
        SBI     PORTB, 0            ; Parpadeo en PB0
        RCALL   Espera_500mS
        CBI     PORTB, 0
        RCALL   Espera_500mS
        RJMP    Lazo_boot
;
; Rutina para esperar 500 mS
;
Espera_500mS:
        LDI     R18, 2
et3:    LDI     R17, 250
et2:    LDI     R16, 250
et1:    DEC     R16                 ; Itera 250 veces y requiere de 4 uS
        NOP                         ;
        BRNE    et1                 ;      250 x 4 uS = 1000 uS = 1 mS
        DEC     R17                 ;
        BRNE    et2                 ;            1 mS x 250 = 250 mS
        DEC     R18                 ;
        BRNE    et3                 ;            250 mS x 2 = 500 mS
        RET
```

Se observa que el código ubicado en la sección de aplicación está completo, es decir, incluye la configuración de los puertos y la inicialización del apuntador de pila. No se considera el estado resultante de la ejecución de la sección de arranque, porque debe funcionar adecuadamente aun sin ejecutar el código de la sección de arranque (sin programar al fusible **BOOTRST**). También puede verse que es posible invocar una rutina ubicada en la sección de arranque desde la sección de aplicación.

En el código también se mostró cómo desplazar los vectores de interrupciones, para que el evento del botón sea atendido en la sección de arranque. En la ISR de la INT0 básicamente se realiza la bifurcación a la sección de aplicación, no es posible insertar una instrucción **RETI** porque regresaría la ejecución al lazo principal de la sección de arranque. Sin embargo, en la sección de aplicación quedaron inhabilitadas las interrupciones porque el bit **I** de **SREG** quedó desactivado al no ejecutarse la instrucción **RETI.**

7.2.3 Cargador para Autoprogramación

Aunque el ejemplo 7.2 funciona correctamente, no tiene mucho sentido colocar una aplicación o parte de ella en la sección de arranque. Esta sección está orientada para que el usuario pueda ubicar un cargador, es decir, un programa que permita modificar una parte o toda la sección de aplicación.

El MCU iniciaría ejecutando al cargador (ubicado en la sección de arranque), el cual esperaría durante un tiempo determinado por si desde una PC u otro sistema se solicita la actualización del programa de aplicación, utilizando alguna de las interfaces incluidas en el MCU (USART, SPI o TWI). Si durante ese periodo se establece la comunicación, por esta interfaz se debe realizar la actualización del código, empleando algún protocolo que garantice la adecuada transferencia de datos; concluida la actualización se debe continuar con la ejecución del programa de aplicación. Si el periodo termina sin una petición para la comunicación con otro sistema, el MCU simplemente da paso a la ejecución de la aplicación.

Si el cargador requiere el uso de interrupciones, debe incluir instrucciones para ubicar a los vectores de interrupciones en la sección de arranque, por lo tanto, antes de bifurcar a la sección de aplicación, los vectores deben regresarse a la sección de aplicación, para que el código de la aplicación no bifurque erróneamente a la sección de arranque.

En la figura 7.3 se ilustra esta idea, la cual en apariencia es simple, no obstante, deben tomarse en cuenta algunos aspectos relevantes que son descritos en las siguientes subsecciones.

7.2.3.1 Restricciones de Acceso en la Memoria Flash

Como parte de los *Bits de Configuración y Seguridad,* los AVR incluyen 4 fusibles para definir el nivel de protección en cada una de las secciones de la memoria Flash. Con ellos se determina si:

- La memoria Flash queda protegida de una actualización.
- Sólo la sección de arranque queda protegida.
- Sólo la sección de aplicación queda protegida.
- Ambas secciones quedan sin protección, permitiendo un acceso total.

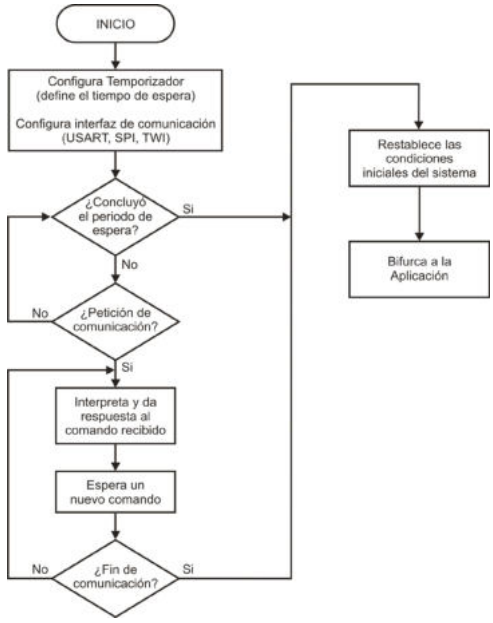


Figura 7.3 Flujo natural de un cargador para autoprogramación

Los fusibles se denominan Bits de Bloqueo de Arranque (**BLB**, *Boot Lock Bits*) y se organizan por pares, el par 0 está dedicado a la sección de aplicación y el par 1 está dedicado a la sección de arranque. En las tablas 7.3 y 7.4 se describen los diferentes niveles de protección, donde puede notarse que la seguridad se determina por las restricciones impuestas a las instrucciones **`LPM`** y **`SPM`**, orientadas a transferencias con memoria de código.

Los fusibles **BLB** tienen un 1 cuando están sin programar. La programación puede realizarse en el momento de configurar al dispositivo, aunque es posible una modificación vía software, en tiempo de ejecución.

Tabla 7.3 Bits de bloqueo para la sección de aplicación

Modo	BLB02	BLB01	Protección
1	1	1	Sin restricciones para las instrucciones **SPM** o **LPM** en la sección de aplicación.
2	1	0	La instrucción **SPM** no puede escribir en la sección de aplicación.
3	0	0	La instrucción **SPM** no puede escribir en la sección de aplicación y a **LPM**, ejecutada desde la sección de arranque, no se le permite leer en la sección de aplicación. Si los vectores de interrupciones son colocados en la sección de arranque, las interrupciones se inhabilitan mientras se ejecute desde la sección de aplicación.
4	0	1	A la instrucción **LPM,** ejecutada desde la sección de arranque, no se le permite leer en la sección de aplicación. Si los vectores de interrupciones son colocados en la sección de arranque, las interrupciones se inhabilitan mientras se ejecute desde la sección de aplicación.

Tabla 7.4 Bits de bloqueo para la sección de arranque

Modo	BLB12	BLB11	Protección
1	1	1	Sin restricciones para las instrucciones **SPM** o **LPM** en la sección de arranque.
2	1	0	La instrucción **SPM** no puede escribir en la sección de arranque.
3	0	0	La instrucción **SPM** no puede escribir en la sección de arranque y a **LPM**, ejecutada desde la sección de aplicación, no se le permite leer en la sección de arranque. Si los vectores de interrupciones son colocados en la sección de aplicación, las interrupciones se inhabilitan mientras se ejecute desde la sección de arranque.
4	0	1	A la instrucción **LPM,** ejecutada desde la sección de aplicación, no se le permite leer en la sección de arranque. Si los vectores de interrupciones son colocados en la sección de aplicación, las interrupciones se inhabilitan mientras se ejecute desde la sección de arranque.

7.2.3.2 Capacidades para Leer-Mientras-Escribe

Además de la división entre la sección de aplicación y la sección de arranque, la Flash también está dividida en dos secciones de tamaño fijo, la primera es una sección que posibilita la capacidad para Leer-Mientras-Escribe (RWW, *Read While Write***)** y la segunda sección restringe el acceso, de manera que No se puede Leer-Mientras-Escribe (NRWW, *No Read While* Write).

El tamaño de la sección NRWW ocupa el tamaño más grande de la sección de arranque, en la tabla 7.5 se muestran los tamaños de ambas secciones para un ATMega8 y para un ATMega16.

Tabla 7.5 Tamaño de las secciones RWW y NRWW

Dispositivo	Páginas RWW	Páginas NRWW	Sección RWW	Sección NRWW
ATMega8	96	32	0x000 – 0xBFF	0xC00 – 0xFFF
ATMega16	112	16	0x0000 – 0x1BFF	0x1C00 – 0x1FFF

Por lo tanto, de acuerdo al tamaño elegido para la sección de arranque, un cargador puede ocupar toda o parte de la sección NRWW, pero nunca va a poder utilizar una parte de la sección RWW. Mientras que una aplicación si puede llegar a ocupar ambas secciones de la memoria Flash.

La diferencia fundamental entre la sección RWW y la NRWW es que:

- Cuando se está borrando o escribiendo una página localizada en la sección RWW, la sección NRWW puede ser leída durante esta operación.
- Cuando se está borrando o escribiendo una página ubicada en la sección NRWW, la CPU se detiene hasta que concluya la operación.

La ubicación del cargador en la sección NRWW hace posible la lectura y ejecución de sus instrucciones consistentes en la modificación del código de la aplicación, situado en la sección RWW. Por lo tanto, Leer-Mientras-Escribe significa lecturas en la sección NRWW relacionadas con escrituras en RWW. Si un cargador intenta modificarse a sí mismo, la ejecución de la CPU queda detenida mientras se realice el borrado o la escritura de una página. En la figura 7.4 se ilustra este comportamiento, en donde puede verse que el apuntador Z es empleado para referenciar la ubicación de la página a modificar.

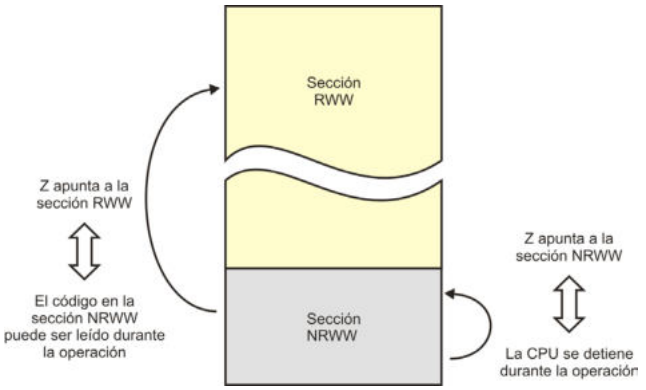


Figura 7.4 Lee de la sección NRWW mientras escribe en la sección RWW

En general, si una secuencia de código se va a mantener sin cambios y la aplicación requiere que incluya instrucciones para modificar un espacio variable en la memoria de código, es necesario que la secuencia fija se encuentre en la sección NRWW y el espacio variable en la sección RWW. La secuencia fija no forzosamente debe ser parte de un cargador.

7.2.3.3 Escritura y Borrado en la Memoria Flash

En la descripción de la instrucción **`SPM`** se indicó que con ella se escribe una palabra de 16 bits en la memoria de código, la palabra debe estar en los registros R1:R0 y la dirección de acceso en el registro Z. Sin embargo, la memoria Flash está organizada en páginas (figura 7.2) y no puede ser modificada por localidades individuales, la escritura o borrado se realiza en páginas completas.

Por lo tanto, el hardware incluye un buffer de memoria del tamaño de una página, en el que se hace un almacenamiento temporal antes de escribir en la memoria Flash. El espacio del buffer es independiente de la memoria SRAM, es sólo de escritura y debe ser llenado palabra por palabra.

El registro Z se utiliza para direccionar una palabra en el buffer, en un ATMega8 sólo se utilizan 5 bits (32 palabras) y en un ATMega16 se emplean 6 bits (64 palabras), en ambos casos, el bit menos significativo de Z es ignorado. En la figura 7.5 se ilustra el direccionamiento en un ATMega8.

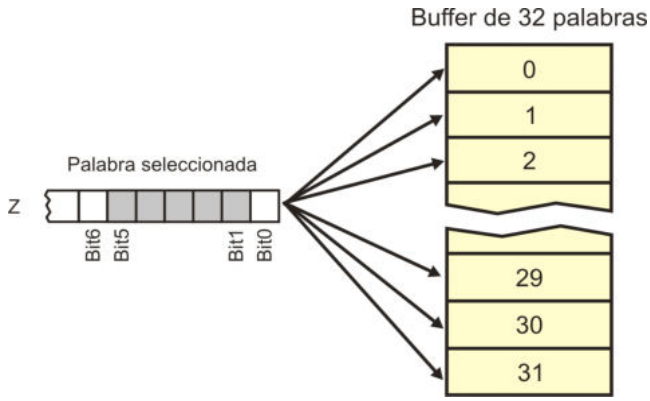


Figura 7.5 Direccionamiento en el buffer temporal

Para escribir una palabra en el buffer, ésta se coloca en R1:R0, con Z se apunta a la dirección deseada, se pone en alto el habilitador de escritura en memoria de programa (**SPMEN**, *Store Program Memory Enable*)**,** que corresponde con el bit menos significativo del Registro para el Control del Almacenamiento en Memoria de Programa (**SPMCR**, *Store Program Memory Control Register*)**,** y finalmente, se ejecuta la instrucción **`SPM`,** dentro de los 4 ciclos de reloj siguientes a la habilitación de escritura. No es posible la escritura de un byte individual.

Una vez que un buffer se ha llenado, su contenido puede ser copiado en la memoria Flash. Ésta y otras tareas se realizan por medio del registro **SPMCR**, los bits de este registro son:

	7	6	5	4	3	2	1	0	
0x37	SPMIE	RWWSB	-	RWWSRE	BLBSET	PGWRT	PGERS	SPMEN	**SPMCR**

- **Bit 7 – SPMIE: Habilita la interrupción por almacenamiento en la memoria de programa**

  Si este bit y el bit **I** de **SREG** están en alto, se produce una interrupción tan pronto como el bit **SPMEN** es puesto en bajo. El bit **SPMEN** se pone en bajo cuando concluye una operación sobre la memoria flash.

- **Bit 6 – RWWSB: Sección RWW ocupada**

  Bandera que se pone en alto cuando se está realizando una operación de autoprogramación (borrando o escribiendo una página) en la sección RWW y por lo tanto, no se puede realizar otro acceso a la sección RWW. La limpieza de la bandera se realiza con el apoyo del bit **RWWSRE**.

- **Bit 5 – No está implementado**

- **Bit 4 – RWWSRE: Habilita la lectura de la sección RWW**

  Si en la sección RWW se está realizando una operación de autoprogramación (borrando o escribiendo una página), la bandera **RWWSB** y el bit **SPMEN** van a tener un nivel alto. Al concluir con la operación, el bit **SPMEN** es puesto en bajo, pero la sección RWW aún permanece bloqueada. Para rehabilitar su acceso, los bits **RWWSRE** y **SPMEN** deben ser puestos en alto al mismo tiempo y durante los 4 ciclos de reloj siguientes se debe ejecutar la instrucción `SPM`, sin importar el contenido de los registros R1:R0 y del apuntador Z.

- **Bit 3 – BLBSET: Ajuste de los Bits de Bloqueo de Arranque**

  Con este bit se realiza la escritura y lectura de los fusibles de **BLB**, empleados para restringir el acceso a las diferentes secciones en la memoria Flash (ver la sección 7.2.2.1).

  Para una escritura, los bits **BLBSET** y **SPMEN** deben ser puestos en alto al mismo tiempo y durante los 4 ciclos de reloj siguientes se debe ejecutar la instrucción **`SPM`.** El valor a escribir debe estar en el registro R0, de acuerdo con la siguiente distribución:

1	1	BLB12	BLB11	BLB02	BLB01	1	1	**R0**

  Sin importar el valor de R1 y colocando 0x0001 en el apuntador Z. El bit **BLBSET** se pone en bajo al completar la escritura o si la instrucción **`SPM`** no se ejecutó dentro de los 4 ciclos de reloj (falló al intentar la escritura). Para una lectura, los bits **BLBSET** y **SPMEN** deben ser puestos en alto al mismo tiempo y durante los 3 ciclos de reloj siguientes se debe ejecutar la instrucción **`LPM`**. El valor leído queda disponible en el registro señalado como destino en la instrucción.

- **Bit 2 – PGWRT: Habilita la escritura de una página**

  Este bit hace posible que el contenido del buffer temporal sea escrito en la memoria Flash, en la página direccionada con la parte alta del apuntador Z. Para ello, los bits **PGWRT** y **SPMEN** deben ser puestos en alto al mismo tiempo y durante los 4 ciclos de reloj siguientes se debe ejecutar la instrucción **`SPM`**. El valor de los registros R1:R0 es ignorado. El bit se limpia automáticamente al concluir la escritura de la página o si la instrucción **`SPM`** no se ejecutó dentro de los 4 ciclos de reloj (falló al intentar la escritura). Si Z direcciona una página de la sección NRWW, la CPU se detiene mientras se realiza la escritura de la página.

- **Bit 1 – PGERS: Habilita el borrado de una página**

  Este bit hace posible el borrado de una página en la memoria Flash, la página debe ser direccionada con la parte alta del apuntador Z. Los bits **PGERS** y **SPMEN** deben ser puestos en alto al mismo tiempo y durante los 4 ciclos de reloj siguientes se debe ejecutar la instrucción **SPM**. El valor de los registros R1:R0 es ignorado. El bit se limpia automáticamente al concluir el borrado de la página o si la instrucción **SPM** no se ejecutó dentro de los 4 ciclos de reloj (falló al intentar el borrado). Si Z direcciona una página de la sección NRWW, la CPU se detiene mientras se realiza el borrado de la página.

- **Bit 0 – SPMEN: Habilitador de escritura en memoria de programa**

  Este bit debe ponerse en alto para modificar la memoria de programa o a los fusibles relacionados, solo o en combinación con otros bits del registro **SPMCR**, y dentro de los 4 ciclos de reloj siguientes se debe ejecutar la instrucción **SPM**.

  Si únicamente se puso en alto al bit **SPMEN**, la instrucción **SPM** escribe la palabra contenida en R1:R0 en el buffer de almacenamiento temporal, en la dirección apuntada por Z. Si **SPMEN** y **RWWSRE** se pusieron en alto, la instrucción SPM desbloquea el acceso a la sección RWW y limpia la bandera **RWWSB**. Si **SPMEN** y **BLBSET** se pusieron en alto, la instrucción **SPM** escribe en los fusibles de **BLB** el dato contenido en R0. Si **SPMEN** y **PGWRT** se pusieron en alto, la instrucción **SPM** escribe el contenido del buffer temporal en la memoria Flash, en la página direccionada con la parte alta del apuntador Z. Si **SPMEN** y **PGERS** se pusieron en alto, la instrucción **SPM** borra la página direccionada con la parte alta del apuntador Z.

  Además, si **SPMEN** y **BLBSET** se ponen en alto, y posteriormente se ejecuta a la instrucción **LPM**, se leen los fusibles de **BLB** y su valor queda en el registro indicado en la instrucción.

7.2.3.4 Direccionamiento de la Flash para Autoprogramación

El contador de programa (PC) es el registro encargado de direccionar la memoria de código. Los bits del PC se dividen en 2 partes, por la organización en páginas de la memoria Flash, con los bits más significativos se selecciona una página (PCPAGE) y con los bits menos significativos una palabra (PCWORD), dentro de la página. Las operaciones relacionadas con la instrucción **SPM** involucran el uso del apuntador Z, el cual toma la misma distribución de bits que el PC, pero sin incluir a su bit menos significativo. En la figura 7.6 se muestra la distribución de los bits para direccionar la memoria Flash.

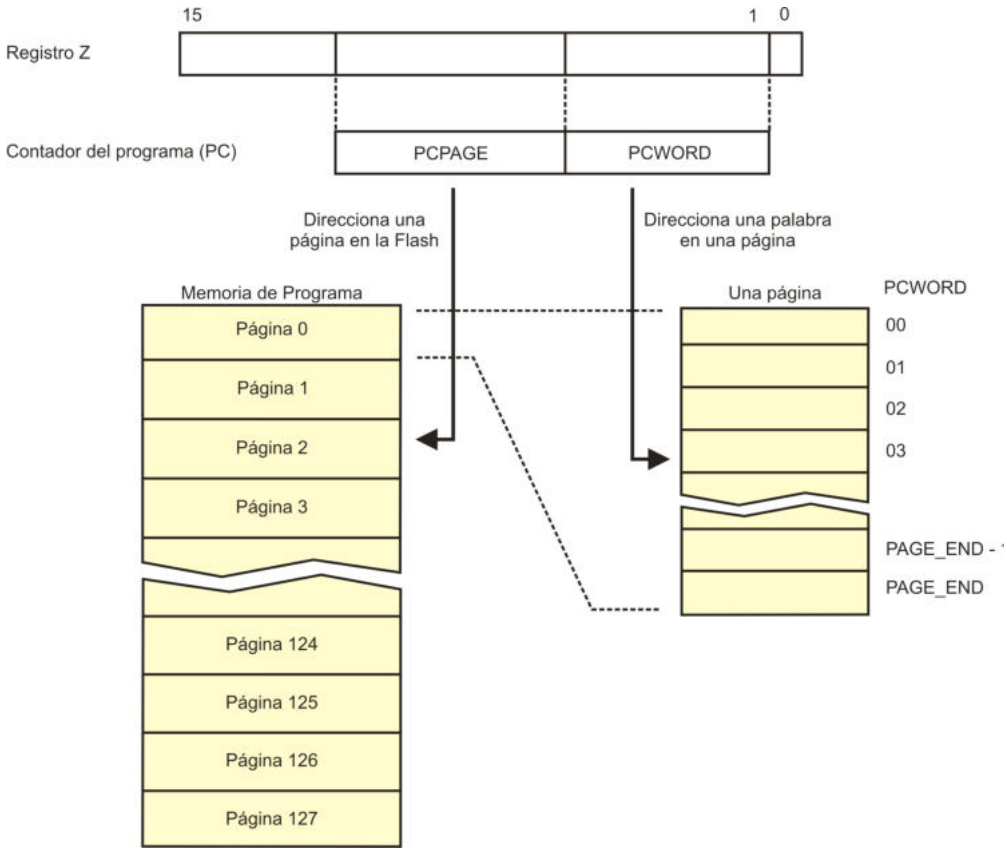


Figura 7.6 Distribución de bits para direccionar la memoria de código

En la tabla 7.6 se muestra el tamaño y ubicación de cada una de las partes del PC, para un ATMega8 y un ATMega16.

Tabla 7.6 Organización y acceso a la memoria de código

MCU	Páginas	Palabras por página	Tamaño de PCWORD	Tamaño de PCPAGE	Ubicación de PCWORD	Ubicación de PCPAGE
ATMega8	128	32	7 bits	5 bits	PC[11:5]	PC[4:0]
ATMega16	128	64	7 bits	6 bits	PC[12:6]	PC[5:0]

Al momento de escribir o borrar una página se debe considerar el tamaño de PCWORD y su ubicación en el PC, ya que de ello depende el valor que toma el apuntador Z. Por ejemplo, si para un ATMega8 se requiere que Z apunte a la página 5, se le debe escribir el valor de 0x0140, de acuerdo con la figura 7.7.

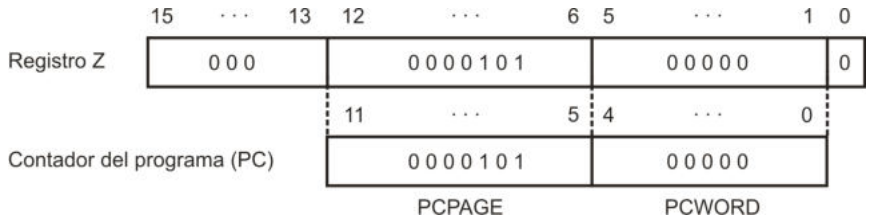


Figura 7.7 Ejemplo para definir el valor de Z, en un ATMega8

Las operaciones de escritura y borrado se realizan en varios ciclos de reloj, esto requiere la inserción de un latch intermedio en el que se captura la dirección proporcionada por el apuntador Z, de manera que, una vez iniciada la operación, el apuntador Z puede ser utilizado para otras tareas.

La única operación realizada con la instrucción **`SPM`** que no utiliza al apuntador Z, es para la escritura de los fusibles de **BLB**, en la cual, el contenido de Z es ignorado y no tiene efecto en la operación.

La instrucción **`LPM`** también hace uso del apuntador Z, sin embargo, el acceso para la carga de memoria de programa se realiza por bytes, por ello, esta instrucción si utiliza al bit menos significativo del registro Z.

7.2.3.5 Programación de la Flash

Es necesario borrar una página de la memoria Flash antes de escribirle nuevos datos, los cuales inicialmente deben estar en el buffer temporal. El proceso puede realizarse con dos secuencias diferentes. La primera secuencia consiste en: Llenar el buffer temporal, borrar la página y escribir el contenido del buffer temporal en la página borrada. La segunda secuencia implica: Borrar la página, llenar el buffer temporal y escribir el contenido del buffer temporal en la página borrada.

No es posible escribir sólo una fracción de una página, si esto es necesario, en el buffer deben incluirse ambas partes, la fracción a modificar y el contenido que va a permanecer sin cambios, para posteriormente escribir la página completa en la memoria Flash.

7.3 Bits de Configuración y Seguridad

Los microcontroladores AVR incluyen un conjunto de fusibles que pueden ser programados para proteger el contenido de un dispositivo o definir su comportamiento. Los fusibles se organizan en 3 bytes y tienen un 1 lógico cuando están sin programar.

El contenido del primer byte, para un ATMega8 y un ATMega16, se muestra en la tabla 7.7, en donde puede notarse que este byte está relacionado con la seguridad del dispositivo. En las posiciones 7 y 6 no hay fusibles implementados. Los que se ubican

en las posiciones de la 5 a la 2 son los Bits de Bloqueo de Arranque **(BLB),** están orientados a proteger el contenido de la memoria de código y se organizan por pares, el par 0 (**BLB02** y **BLB01**) está dedicado a la sección de aplicación y el par 1 (**BLB12** y **BLB11**) a la sección de arranque.

Tabla 7.7 Fusibles relacionados con la seguridad del MCU

No. Bit	Nombre	Descripción	Valor por default
7	-		1
6	-		1
5	BLB12	Bit de Bloqueo de Arranque.	1
4	BLB11	Bit de Bloqueo de Arranque.	1
3	BLB02	Bit de Bloqueo de Arranque.	1
2	BLB01	Bit de Bloqueo de Arranque.	1
1	LB2	Bit de seguridad.	1
0	LB1	Bit de seguridad.	1

En las tablas 7.3 y 7.4 se describieron los diferentes niveles de protección (tabla 7.3 para la sección de aplicación y tabla 7.4 para la de arranque), donde se mostró que la seguridad se determina por las restricciones impuestas a las instrucciones **`LPM`** y **`SPM`**, relacionadas con transferencias de memoria de código.

Los fusibles de las posiciones 1 y 0 protegen el contenido de las memorias Flash y EEPROM, ante programaciones futuras, se tienen 3 niveles de seguridad, los cuales se describen en la tabla 7.8.

Tabla 7.8 Modos de protección del contenido de la memoria Flash y EEPROM

Modo	LB2	LB1	Tipo de protección
1	1	1	Sin protección para las memorias Flash y EEPROM.
2	1	0	La programación futura de las memorias Flash y EEPROM es inhabilitada en los modos de programación Serial y Paralelo. Los fusibles también son bloqueados en los modos de programación Serial y Paralelo.
3	0	0	La programación y verificación futura de las memorias Flash y EEPROM es inhabilitada en los modos de programación Serial y Paralelo. Los fusibles también son bloqueados en los modos de programación Serial y Paralelo.

De acuerdo con la tabla 3.8, si se va a programar el modo 2 o el modo 3 con los fusibles **LB[2:1]**, primero se deben configurar los demás fusibles, antes de perder el acceso.

Los otros 2 bytes de fusibles son para configuración, en el segundo byte difieren los dos fusibles ubicados en las posiciones más significativas, de un ATMega8 con los de un ATMega16. En la tabla 7.9 se describe el segundo byte de fusibles y se resalta la diferencia entre ambos dispositivos.

Se observa que en un ATMega16 la interfaz JTAG está habilitada por default, esto hace que el puerto C no pueda emplearse por completo como entrada y salida de propósito general, si se requiere de su uso y no se va a emplear a la interfaz JTAG, ésta debe ser inhabilitada.

El tercer byte de fusibles se muestra en la tabla 7.10, estos fusibles están relacionados con el sistema de *reset* y el sistema de reloj, su aplicación fue descrita en las secciones 2.7 y 2.8.

Tabla 7.9 Segundo byte de fusibles y el primero relacionado con la configuración del MCU

No. Bit	Nombre	Descripción	Valor por default
7	RSTDISBL (ATMega8)	Determina si PC6 es una I/O genérica o terminal de *reset*.	1 (PC6 es *reset*)
7	OCDEN (ATMega16)	Habilita al hardware incluido para depuración (OCD, *On-Chip Debug*).	1 (ODC inhabilitado)
6	WDTON (ATMega8)	Define si el WDT va a estar siempre activado.	1 (Sin programar, el WDT se habilita con WDTRC)
6	JTAGEN (ATMega16)	Habilita a la interfaz JTAG.	0 (Interfaz JTAG habilitada)
5	SPIEN	Habilita la programación serial y descarga de datos vía SPI.	0 (Programación SPI habilitada)
4	CKOPT	Opciones del oscilador (Descrito en la sección 2.8).	1
3	EESAVE	El contenido de la EEPROM es preservado durante el borrado del dispositivo.	1 (La EEPROM no se preserva)
2	BOOTSZ1	Determina el tamaño de la sección de Arranque (tabla 7.2).	0
1	BOOTSZ0		0
0	BOOTRST	Selecciona la ubicación del vector de *reset* (sección 7.2).	1 (Vector de *reset* en 0x000)

Tabla 7.10 Fusibles relacionados con la seguridad del MCU

No. Bit	Nombre	Descripción	Valor por default
7	BODLEVEL	Define el nivel para el detector de un bajo voltaje, para un reinicio.	1
6	BODEN	Habilita al detector de bajo voltaje.	1
5	SUT1	Seleccionan el tiempo de establecimiento.	1
4	SUT0		0
3	CKSEL3	Seleccionan la fuente de reloj.	0
2	CKSEL2		0
1	CKSEL1		0
0	CKSEL0		1

Si se planea la modificación de los fusibles, es conveniente realizar su lectura antes de escribir los nuevos valores. Algunas herramientas de programación no realizan una lectura automática, de manera que si se modifican sólo los fusibles de interés sin conservar el valor de los restantes, podría conducir a que el microcontrolador no opere o que lo haga en forma incorrecta.

7.4 Interfaz JTAG

El acrónimo JTAG **(***Joint Test Action Group***)** hace referencia a la norma IEEE 1149.1, originalmente desarrollada para la evaluación de circuitos impresos. Actualmente con JTAG se describe una interfaz serial empleada para la programación y depuración de microcontroladores u otros dispositivos programables, como los FPGAs. Esta interfaz está incluida en los microcontroladores ATMega16 y cumple con ambos propósitos: programación y depuración.

7.4.1 Organización General de la Interfaz JTAG

En la figura 7.8 se muestra la organización general de la interfaz JTAG y su vinculación con un sistema digital. Desde alguna computadora u otro maestro se puede evaluar al sistema digital, para ello únicamente se emplean 4 señales:

- TDI **(***Test Data In***)** Entrada de datos de prueba.
- TDO **(***Test Data Out***)** Salida de datos de prueba.
- TMS **(***Test Mode Select***)** Selección del modo de prueba.
- TCK **(***Test Clock***)** Reloj.

La señal de *reset* no es parte del estándar, por lo que su disponibilidad depende del dispositivo en particular.

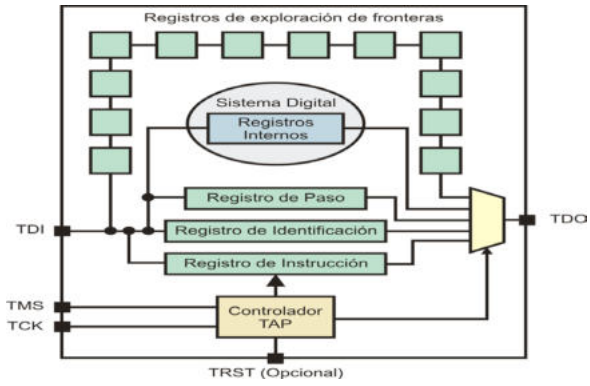


Figura 7.8 Organización de la Interfaz JTAG

La interfaz incluye un conjunto de registros que rodean al sistema digital bajo evaluación, todos funcionando bajo un esquema de desplazamiento serial. El registro de instrucción mantiene al comando recibido por parte del maestro mientras se ejecuta. El registro de identificación debe contener un código con el que se describe al dispositivo. Los registros de exploración de fronteras **(***boundary-scan***)** permiten al maestro conocer las actividades que ocurren en el entorno del dispositivo bajo exploración.

Es posible conectar varios dispositivos en serie, conectando la salida TDO de un dispositivo con la entrada TDI de otro. El registro de paso permite un flujo directo de información, omitiendo la evaluación de un dispositivo en particular, dentro de una cadena serial.

Con la interfaz JTAG también se puede conocer el estado del sistema, dado que es posible la lectura de sus registros internos.

El controlador del Puerto de Acceso para Pruebas (TAP, *Test Access Port***)** es el módulo que coordina a la interfaz JTAG. Se trata de una máquina de estados que es controlada por las señales TCK y TMS. La máquina es sincronizada por la señal de reloj recibida en TCK. Con la señal TMS se selecciona el modo de funcionamiento, es decir, su valor determina la secuencia de trabajo del controlador TAP.

7.4.2 La Interfaz JTAG y los Mecanismos para la Depuración en un AVR

Con la interfaz JTAG incluida en los ATMega16 se puede programar y depurar al dispositivo. En la figura 7.9 se muestra la organización de la interfaz y también pueden apreciarse todos los módulos agregados para hacer posible la depuración.

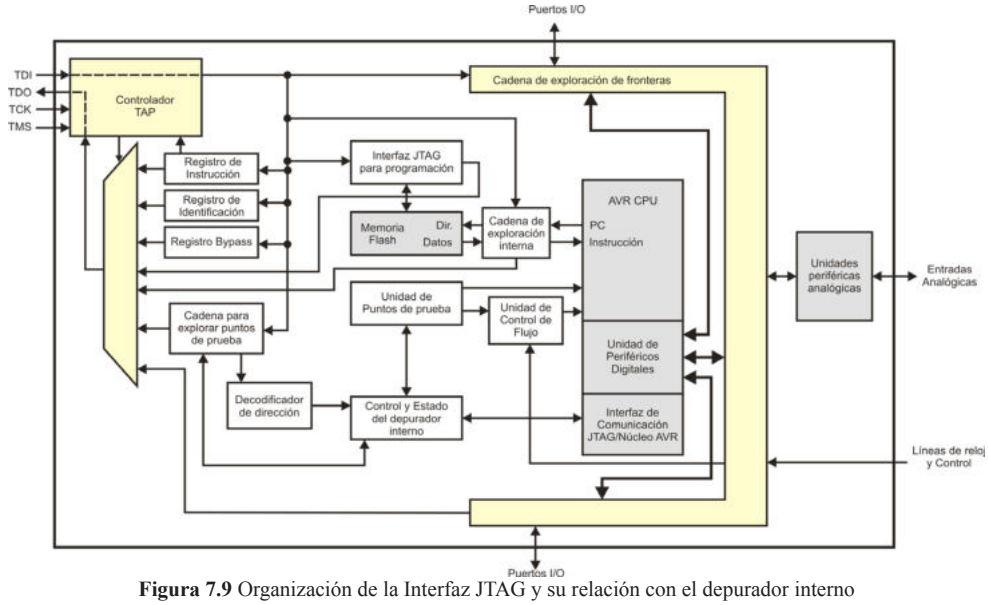


Figura 7.9 Organización de la Interfaz JTAG y su relación con el depurador interno

La relación de las señales de la interfaz JTAG con las terminales de los puertos se muestra en la tabla 7.8.

Tabla 7.8 Ubicación de las terminales de la interfaz JTAG en un ATMega16

Terminal JTAG	Ubicación	Terminal en un ATMega16
TCK	PC2	24
TMS	PC3	25
TDO	PC4	26
TDI	PC5	27

Los dispositivos ATMega16 son distribuidos con la interfaz JTAG habilitada, por lo que en principio el puerto C no está completamente disponible para entradas o salidas. Para el uso del puerto completo se debe modificar al fusible **JTAGEN**, el cual es parte de los *Bits de Configuración y Seguridad***,** (sección 7.3). No obstante, si alguna aplicación requiere la inhabilitación temporal de la interfaz JTAG, ésta se realiza con el bit **JTD** (*JTAG Disable*) ubicado en la posición 7 del registro **MCUCSR**.

	7	6	5	4	3	2	1	0	
0x34	JTD	ISC2	-	JTRF	WDRF	BORF	EXTRF	PORF	**MCUCSR**

El bit **JTD** tiene un 0 lógico después de un *reset***,** por lo tanto la interfaz JTAG está habilitada (siempre que el bit **JTAGEN** esté programado). La inhabilitación requiere la escritura de un 1 lógico. Pero para evitar habilitaciones o inhabilitaciones no intencionales, la escritura del bit **JTD** requiere de 2 escrituras con el mismo valor, dentro de un periodo de 4 ciclos de reloj.

El bit **JTRF** (bit 4 de **MCUCSR**) también está relacionado con la interfaz JTAG, es una bandera que se pone en alto cuando hay un reinicio del sistema debido a la instrucción AVR_RESET, de la interfaz JTAG (en la sección 2.7 se describieron las opciones de reinicio).

Aunque la interfaz JTAG y los recursos de depuración se muestran como un hardware complejo, en la práctica su uso es transparente, el software AVR Studio incluye los mecanismos para la programación y depuración de dispositivos. Empleando alguna plataforma de desarrollo, como la tarjeta *AVR Dragon*[7] .

La depuración hace uso del Registro del Depurador Interno (**OCDR**, *On-Chip Debug Register*)**,** el cual proporciona un canal de comunicación entre el programa ejecutándose en el núcleo AVR y el depurador. Los bits de este registro son:

	7	6	5	4	3	2	1	0	
0x31	IDRD	Registro del Depurador Interno							**OCDR**

7 AVR Dragon es una tarjeta de bajo costo (aproximadamente $60.00 USD) desarrollada por Atmel, con la cual es posible programar microcontroladores AVR por los puertos SPI y JTAG. Desde el puerto JTAG también es posible la depuración, si el microcontrolador tiene una memoria hasta de 32 Kbyte. La programación y depuración se realizan desde el AVR Studio, aun si el programa se desarrolló en Lenguaje C.

El bit **IDRD (***I/O Debug Register Dirty***)** es una bandera que se pone en alto indicando una escritura en el registro **OCDR**. El depurador limpia la bandera cuando ha leído la información.

En un ATMega16 el registro **OCDR** comparte la dirección con el registro **OSCCAL**, el cual es utilizado para calibrar al oscilador interno (descrito en la sección 2.8.4). Por lo tanto, se tiene acceso al registro **OCDR** sólo si el fusible **OCDEN** fue programado, en otro caso, con la dirección 0x31 se tiene acceso al registro **OSCCAL**. El fusible **OCDEN** es parte de los *Bits de Configuración y Seguridad***.**

7.5 Ejercicios

1. Para evaluar la funcionalidad del WDT:

   a) Desarrolle un contador binario de 16 bits, empleando 2 puertos de un microcontrolador AVR. Habilite al WDT pero en el código no incluya instrucciones para su reinicio. Observe hasta qué número alcanza el contador antes de reiniciar la cuenta y estime el periodo de desbordamiento del WDT.

   b) Modifique los bits **WDP[2:0]** del registro **WDTCR** y observe el cambio en el periodo de desbordamiento. Estime estos periodos con diferentes combinaciones de **WDP[2:0]**.

   c) Coloque la instrucción **`WDR`** dentro del lazo que incrementa al contador y observe cómo la cuenta ya no se reinicia.

2. Empleando la sección de arranque y la sección de aplicación, desarrolle un sistema con una doble funcionalidad, por ejemplo, podrían conectarse 4 displays de 7 segmentos con un bus de datos común y las aplicaciones posibles: Un contador ascendente/descendente y una marquesina de mensajes. Ubique la aplicación 1 en la sección de arranque y la 2 en la sección de aplicación. Agregue el hardware y software necesario para una comunicación serial y acondicione para que el sistema inicie con la aplicación 1 y tras recibir un comando serial, conmute a la aplicación 2.

3. Desarrolle un cargador para autoprogramación con base en el diagrama de flujo de la figura 7.3, utilice a la USART como medio de comunicación, de manera que la nueva aplicación provenga de una PC. Pruebe cargando diferentes aplicaciones.

4. En el ejemplo 3.1 se hizo parpadear a un LED conectado en la terminal PB0 a una frecuencia aproximada de 1 Hz, implemente este ejemplo y observe cómo cambia la frecuencia del parpadeo al modificar el valor de los fusibles **CKSEL[3:0]** *(Bits de Configuración y Seguridad)*, haciendo que el MCU opere a 2, 4 u 8 MHz (los ATMega8 y ATMega16 son distribuidos operando a una frecuencia de 1 MHz).

8. Interfaz y Manejo de Dispositivos Externos

En este capítulo se muestra el funcionamiento de dispositivos externos típicos y cómo manejarlos por medio de un microcontrolador, con algunas consideraciones para su programación.

8.1 Interruptores y Botones

Los interruptores y botones definen el estado lógico de una entrada en el microcontrolador. Difieren en que un interruptor deja un estado permanente, mientras que un botón introduce un estado temporal, únicamente mientras es presionado. En la práctica, los interruptores suelen emplearse para definir la configuración de un sistema y los botones para modificar el estado de variables internas.

Cuando se utiliza un interruptor o un botón, se requiere de un resistor de fijación a Vcc (*pull-up*) o un resistor de fijación a tierra (*pull-down*).

Con un resistor de *pull-up* se introduce un 1 lógico mientras los dispositivos se encuentran abiertos y un 0 lógico al cerrarse. El resistor de *pull-up* evita un corto circuito entre Vcc y tierra, un valor típico para este resistor es de 10 Kohms, con este valor, en el resistor de *pull-up* circula una corriente promedio de 0.5 mA cuando el dispositivo se cierra. En la figura 8.1 se muestra un interruptor y un botón con sus resistores de *pull-up*. Los microcontroladores AVR incluyen resistores de *pull-up* internos, únicamente debe habilitarse su conexión (ver sección 2.5). Al emplear los resistores de *pull-up* internos se reduce el hardware externo, aminorando el tamaño y costo del circuito impreso.

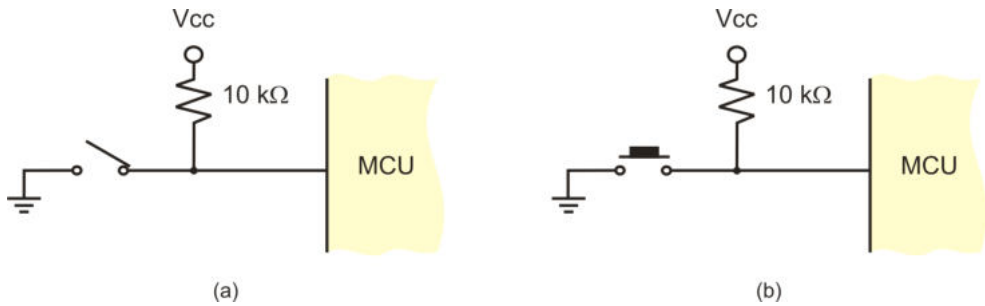


Figura 8.1 Conexión de (a) un interruptor y (b) un botón, con resistor de *pull-up*

La otra opción para la conexión de interruptores y botones involucra el uso de un resistor de *pull-down*, como se muestra en la figura 8.2. Con un resistor de *pull-down*, el dispositivo introduce un 0 lógico mientras el se encuentra abierto, al cerrarlo introduce un 1 lógico. En un AVR, el resistor de *pull-up* interno debe desconectarse si se utiliza un resistor de *pull-down* externo.

Figura 8.2 Conexión de (a) un interruptor y (b) un botón, con resistor de *pull-down*

8.2 Teclado Matricial

Cuando se requieren pocas entradas, los interruptores o botones pueden conectarse directamente al MCU, cada uno utilizando una terminal. Sin embargo, si se requiere de muchas entradas lo mejor es emplear un teclado matricial.

Un teclado matricial es un arreglo de botones, dispuestos de manera tal que reducen el número de terminales utilizadas en el MCU. Por ejemplo, para 16 botones es posible utilizar una matriz de 4 x 4, como se muestra en la figura 8.3, con ello, el manejo de 16 botones sólo requiere de 8 terminales. En general, para una matriz de *n* x *m* botones se requiere de *n* + *m* terminales en el MCU, *n* salidas y *m* entradas, lo cual sólo tiene sentido cuando *n* y *m* sean mayores a 2.

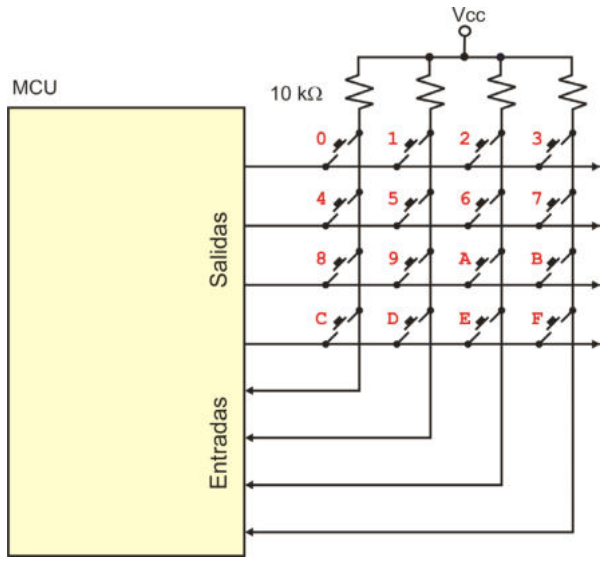


Figura 8.3 Teclado matricial de 4 x 4 botones

En la figura 8.3 se observa que mientras no se ha presionado alguna tecla, las entradas tienen un nivel alto de voltaje (0b1111) debido a los resistores de *pull-up*. El teclado se revisa renglón por renglón para detectar si hay una tecla presionada. En el renglón a revisar se coloca un 0 lógico, si hay una tecla presionada, al leer

las entradas se obtiene un valor diferente de 0b1111, con un 0 en la columna que corresponde con la tecla presionada.

La revisión completa implica colocar, secuencialmente, cada uno de los términos mostrados en la tabla 8.1.

Tabla 8.1 Secuencia de salidas para revisar un teclado matricial de 4 x 4

Salida	Acción
0b1110	Sondea el renglón 0
0b1101	Sondea el renglón 1
0b1011	Sondea el renglón 2
0b0111	Sondea el renglón 3

El valor de una tecla presionada depende del **renglón** bajo revisión y de la **columna** en la que se encontró un valor igual a 0. Con estos datos, el valor de la tecla está dado por:

tecla = 4 x **renglón** + **columna**

Por ejemplo, suponiendo que se presiona la tecla con el valor numérico de 6, al enviar a las salidas: 0b1110 (revisando el renglón 0), en las entradas se obtiene el valor de 0b1111, porque en el renglón 0 no hay tecla presionada. Al enviar a las salidas: 0b1101 (renglón 1), en las entradas se obtiene el valor de 0b1011, detectando un 0 en la columna 2. Por lo tanto:

tecla = 4 x **renglón** + **columna** = 4 x 1 + 2 = 6

En general, para un teclado de *n* x *m* botones, el valor de la tecla está determinado por:

tecla = *m* x **renglón** + **columna**

Ejemplo 8.1 Utilizando lenguaje C, realice una función que revise si hay una tecla presionada en un teclado matricial de 4 x 4 conectado en el puerto B de un ATMega16. La función debe regresar el valor de la tecla ó -1 (0xFF) si no hubo tecla presionada.

La conexión del teclado con el AVR se muestra en la figura 8.4. La parte baja del puerto se emplea para las salidas y la parte alta para las entradas. Para evitar el uso de resistores de *pull-up* externos, en el programa principal deben habilitarse los resistores de *pull-up* de las entradas.

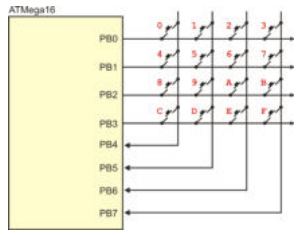


Figura 8.4 Conexión de un teclado de 4 x 4 con un ATMega16

El código en lenguaje C de la función es:

```
char  teclado() {
unsigned  char  secuencia[] = {0xFE, 0xFD, 0xFB, 0xF7 };
unsigned  char  i, renglon, dato;

for(renglon = 0, i = 0; i < 4; i++) {

      PORTB = secuencia[i];          // Ubica la salida
      asm("nop");                    // Espera que las señales se
                                     // estabilicen
      dato = PINB & 0xF0;            // Lee la entrada (anula la
                                     // parte baja)
      if( dato != 0xF0 ) {           // Si se presionó una tecla
            _delay_ms(200);          // Evita rebotes
            switch(dato) {           // Revisa las columnas

                  case  0xE0: return  renglon;
                  case  0xD0: return  renglon + 1;
                  case  0xB0: return  renglon + 2;
                  case  0x70: return  renglon + 3;
            }
      }
      renglon += 4;                  // Revisa el siguiente
                                     // renglón
}

return 0xFF;                         // No hubo tecla presionada
}
```

En la codificación de la función se utilizó una suma porque es más eficiente realizar una suma en cada iteración que un producto al detectar la tecla presionada. También, la ejecución de la función aumenta su rendimiento si la secuencia con las constantes se declara como global, dado que se evita el almacenamiento en SRAM cada vez que la función sea invocada. Otra alternativa podría ser el uso de la memoria de código, dado que la secuencia de salidas es constante.

El retardo que se introduce después de que se ha detectado una tecla presionada es necesario porque el tiempo durante el cual el usuario la presionó es mucho mayor al requerido para la revisión del teclado, sin este retardo, el programa procesaría como si la misma tecla se hubiera presionado muchas veces. Su duración puede ser menor, depende de las actividades a realizar entre cada tecla presionada.

8.2.1 Decodificadores Integrados para Teclados Matriciales

Existen circuitos integrados que funcionan como decodificadores de teclados matriciales. Por ejemplo, la firma Fairchild Semiconductor® manufactura dos modelos: El MM74C922 y el MM74C923. El primero es un decodificador de un teclado matricial de 4 x 4 (16 teclas) y el segundo decodifica un teclado de 5 x 4 (20 teclas). El circuito se encarga del barrido del teclado, generando las secuencias de salida y revisando las entradas. Presenta una interfaz para un MCU, que incluye dos señales de control (*dato disponible* y *habilitación de la salida*) y un bus de datos.

La señal *dato disponible* (DA, *data available*) le indica al MCU que el usuario ha presionado una tecla, esta señal puede conectarse con una interrupción externa del MCU, reduciendo el código requerido para la revisión del teclado. La señal *habilitación de la salida* (OE, *output enable*) activa unos buffers de 3 estados para que el valor de la tecla presionada esté disponible en el bus de datos. En la figura 8.5 se muestra la conexión de un MM74C922 con un ATMega8.

Para reducir el número de terminales empleadas por el microcontrolador, es posible conectar un inversor de la salida *DA* a la entrada *OE*, con ello, tan pronto se tiene un dato disponible, se habilita su salida en el bus de datos.

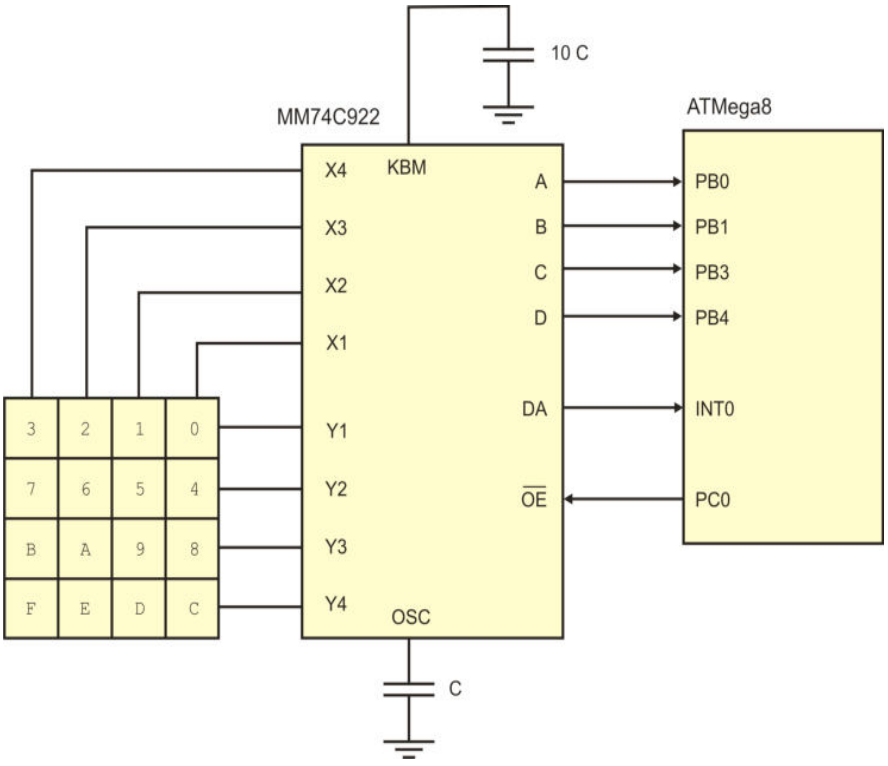


Figura 8.5 Conexión de un MM74C922 con un ATMega8

8.3 Interfaz con LEDs y Displays de 7 Segmentos

Un diodo emisor de luz (LED, *Light-Emitting Diode*) es un indicador visual utilizado para mostrar el estado de un sistema, para ello, únicamente se requiere que el LED sea polarizado directamente (voltaje positivo en el ánodo, con respecto al cátodo). En la figura 8.6 se muestran 2 formas de conectar un LED con un MCU, en (a) se utiliza lógica positiva, es decir, un 1 lógico en el puerto enciende al LED. En (b) se utiliza lógica negativa, esto significa que el LED se enciende con un 0 lógico.

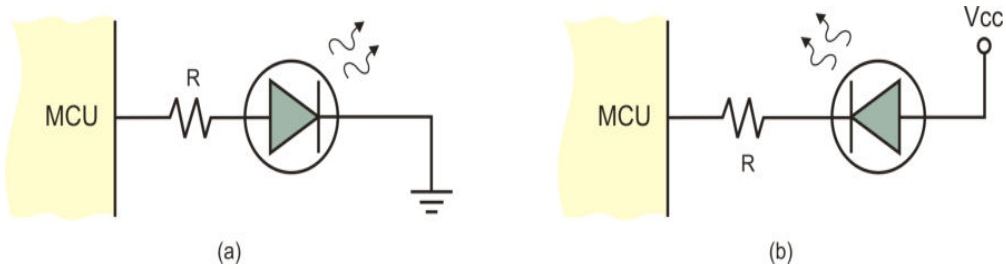


Figura 8.6 Conexión de un LED (a) con lógica positiva y (b) con lógica negativa

En otras palabras, en (a) el MCU proporciona la alimentación del LED y en (b) únicamente una referencia. Es preferible que el MCU proporcione referencias, de esta manera, la capacidad en el suministro de corriente por parte del MCU no es una limitante para el manejo de muchos LEDs.

La resistencia es para limitar la corriente, con esto se protege tanto al microcontrolador como al LED. Las terminales en los AVR pueden suministrar hasta 40 mA de corriente, lo cual puede resultar excesivo para algunos LEDs, el valor de R depende de la corriente que se desee hacer circular en el LED, se obtiene con la ley de Ohm:

$$i = \frac{Vcc - V_{LED}}{R} = \frac{Vcc - 0.7v}{R} \quad , \quad \text{por lo tanto:} \quad R = \frac{Vcc - V_{LED}}{i} = \frac{Vcc - 0.7v}{i}$$

Un display de 7 segmentos básicamente es un conjunto de 8 LEDs, dispuestos de manera tal que es posible mostrar un número y el punto decimal. Se tienen dos tipos de displays, de ánodo común o de cátodo común, en el primer tipo, en un nodo se conecta el ánodo de cada uno de los LEDs y en el segundo tipo, son los cátodos los que se conectan en el mismo nodo. En la figura 8.7 se muestran ambos tipos de displays y la forma en que se pueden conectar a uno de los puertos de un MCU.

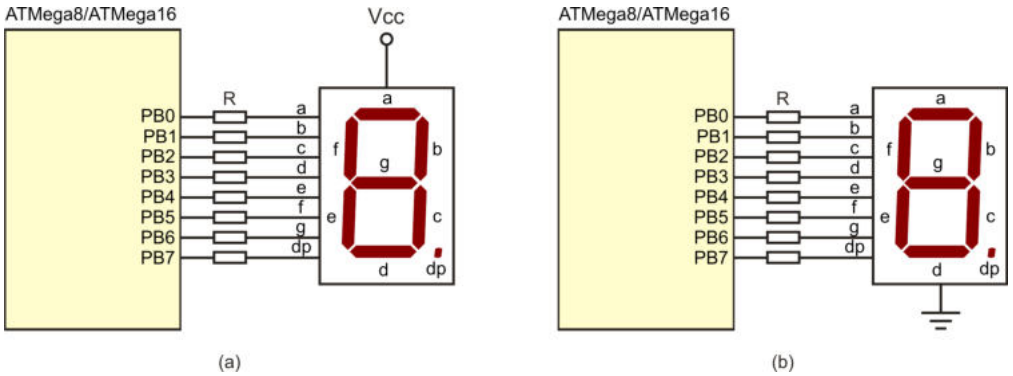


Figura 8.7 Conexión de un display de 7 segmentos (a) de ánodo común y (b) de cátodo común

Nuevamente se debe considerar el uso de resistores para limitar la corriente en los LEDs y el uso de lógica negada para que el MCU únicamente proporcione referencias y no suministre energía. En la figura 8.7 se ha considerado el uso de 1 resistor en cada segmento, no obstante, a veces se prefiere utilizar sólo 1 resistor en la terminal común, con ello se reduce el hardware, pero con la desventaja de que el encendido no es uniforme para todos los números.

En muchas aplicaciones de MCUs es común el manejo de más de un display de 7 segmentos, pueden ser 2, 3, 4 o más, dependiendo de la aplicación. En esos casos, no conviene dedicar un puerto para cada display, porque se agotarían los puertos y no se tendría disponibilidad para otras entradas o salidas. Una alternativa es conectar los segmentos de todos los displays en un bus común para los datos e ir habilitando a cada display en un instante de tiempo, mientras en el bus se coloca la información a mostrar. Si el barrido se realiza a una velocidad alta, aparenta que todos los displays están encendidos al mismo tiempo.

En la figura 8.8 se muestra la conexión de 4 displays de 7 segmentos con un ATMega8, esta conexión con un bus común hace que únicamente se requiera de un puerto para los datos y 4 terminales para los habilitadores. Se utilizan displays de ánodo común, de manera que el ATMega8 sólo proporcione referencias en el bus de datos. La habilitación de cada display se realiza por medio de un transistor bipolar PNP de propósito general (puede ser un BC558), esto para que la corriente que circule en los diferentes segmentos sea proporcionada por la fuente de alimentación y no por el microcontrolador.

Los transistores básicamente funcionan como interruptores, activándose con un 0 lógico para polarizar directamente la unión de emisor a base, permitiendo un flujo de corriente de emisor a colector. Las resistencias R1 y R2 permiten limitar este flujo de corriente, su valor depende de la luminosidad deseada en los displays. Podría omitirse a la resistencia R2 y limitar la corriente únicamente con R1.

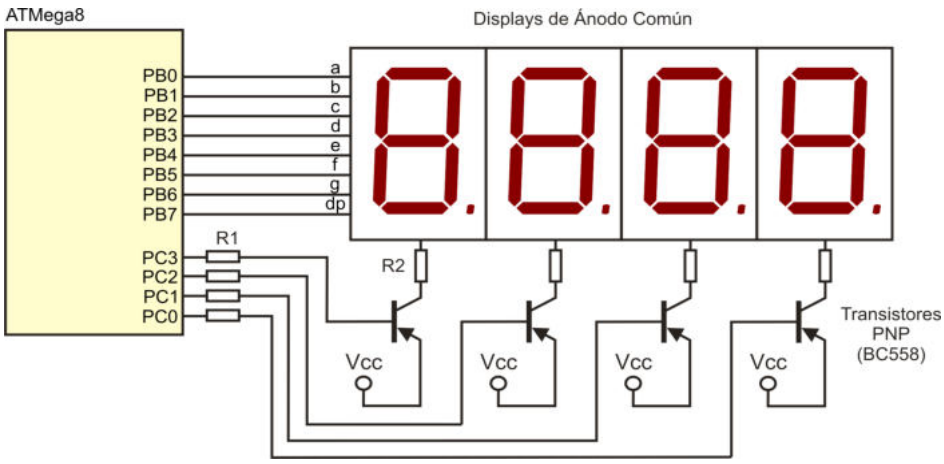


Figura 8.8 Conexión de 4 displays de 7 segmentos utilizando un bus común para los datos

Una vez activado un display, es necesario esperar el tiempo suficiente para que alcance su máxima luminosidad, antes de conmutar al siguiente. Un valor conveniente para este retardo es de 5 mS. Esta técnica para el manejo de varios displays con un bus común, es aplicable a las matrices comerciales de 7 x 5 LEDs, o bien, a matrices personalizadas de diferentes dimensiones.

Ejemplo 8.2 Considerando el hardware de la figura 8.8, realice una función en lenguaje C que haga un barrido en los displays para mostrar 4 datos recibidos en un arreglo. Suponga que los datos ya están codificados en 7 segmentos (con lógica negada por ser displays de ánodo común) y que la posición 0 del arreglo corresponde con el display que está ubicado en el extremo derecho.

```
// Constantes para los habilitadores (deben ubicarse en un ámbito global).
const char  habs[] PROGMEM = { 0x0E, 0x0D, 0x0B, 0x07 };

// Función para el despliegue de datos
void   mostrar(unsigned char     datos[]) {

unsigned char   i;

       for( i = 0; i < 4; i++) {
              PORTB = datos[i];                  // Envía el dato al puerto
              PORTC = pgm_read_byte(&habs[i]); // Habilita para su
                                                 // despliegue
              _delay_ms(5);                      // Espera se vea
                                                 // adecuadamente
              PORTC = 0x0F;                      // Inhabilita para no
                                                 // introducir
       }                                         // ruido en otro display
}
// La función concluye con los displays inhabilitados
```

8.4 Manejo de un Display de Cristal Líquido

Un display de cristal líquido (LCD, *liquid crystal display*) es un dispositivo de salida que permite mostrar más información que los LEDs o displays de 7 segmentos. La información a mostrar depende del tipo de LCD. Un LCD alfanumérico puede mostrar caracteres ASCII, japoneses o griegos, de un conjunto preestablecido, aunque también cuenta con un espacio para escribir 8 caracteres personalizados. En estos LCDs la información se organiza y presenta en columnas y renglones, sus tamaños típicos son 16 x 1, 16 x 2 y 20 x 4. Un LCD gráfico (GLCD) es capaz de presentar caracteres, símbolos especiales y gráficos. Un GLCD está organizado como una matriz de pixeles, por ejemplo de 128 x 64.

En esta sección únicamente se hace referencia a LCDs alfanuméricos, específicamente de 16 x 1 y de 16 x 2. En la figura 8.9 se observa que las terminales de un LCD de 16 x 1 son las mismas que en un LCD de 16 x 2, y en la tabla 8.2 se describe la funcionalidad de cada terminal.

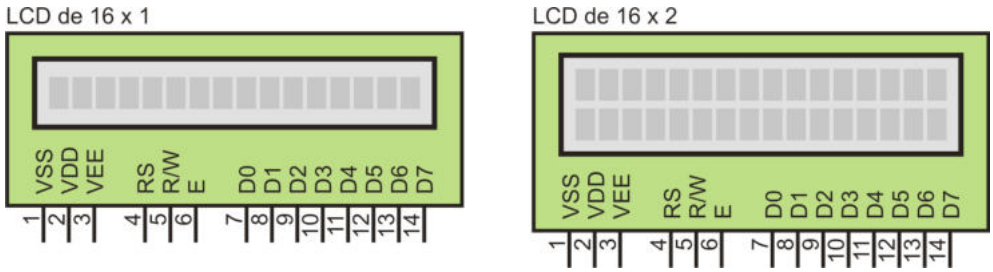


Figura 8.9 Disposición de terminales en un LCD de 16 x 1 y en uno de 16 x 2

Tabla 8.2 Función de las terminales en un LCD

Número	Nombre	Función
1	VSS	Tierra
2	VDD	Voltaje de alimentación
3	VEE	Contraste
4	RS	Selecciona entre un comando (RS = '0') o un dato (RS = '1')
5	R/W	Selecciona entre una escritura (R/W = '0') o una lectura (R/W = '1')
6	E	Habilitación del LCD
7 - 14	D0 – D7	Bus de datos

Un LCD incluye un circuito controlador, éste es el encargado de manejar a la pantalla de cristal líquido, generando los niveles de voltaje y realizando su refresco, para mostrar la información de un sistema electrónico. El controlador proporciona una interfaz con las terminales descritas en la tabla 8.2, para ser manejado por un MCU o un microprocesador, por medio de un conjunto de comandos que el controlador es capaz de reconocer y ejecutar. En la figura 8.10 se muestra la organización del LCD, separando la pantalla del controlador.

El controlador puede diferir de un fabricante a otro, la información de esta sección es compatible con los siguientes LCDs: el ST7066U de Sitronix, el S6A0069X de Samsung, el HD44780 de Hitachi, el SED1278 de SMOS y el TM161A de Tianma.

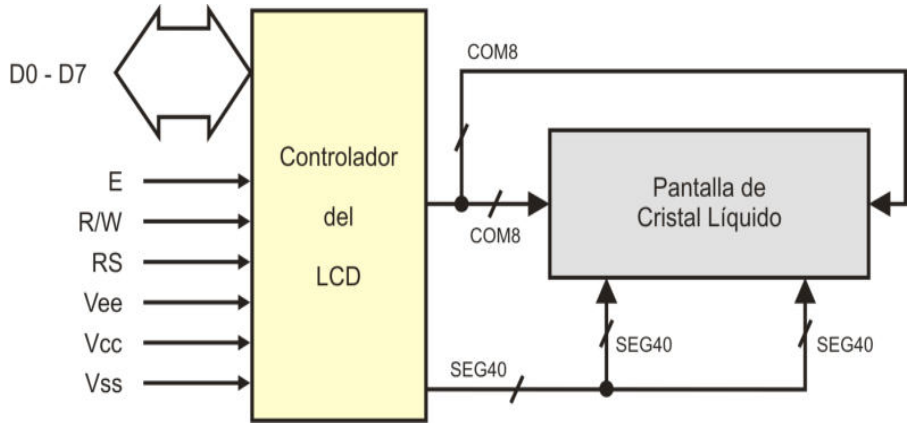


Figura 8.10 El controlador se encarga de manejar la pantalla de cristal líquido

8.4.1 Espacios de Memoria en el Controlador de un LCD

El controlador de un LCD tiene 3 espacios de memoria, cada uno con un propósito específico. Estos espacios son: una RAM para el despliegue de datos (DDRAM, *Display Data RAM*), una ROM generadora de caracteres (CGROM, *Character Generator ROM*) y una RAM generadora de caracteres (CGRAM, *Character Generator RAM*). Pero sólo tiene un contador de direcciones compartido por la DDRAM y la CGRAM. El acceso a estos espacios y al contador de direcciones únicamente es posible después de que se ha inicializado al LCD.

La DDRAM almacena referencias de los caracteres que se van a mostrar en la pantalla. Son referencias a mapas de bits almacenados en CGROM o a un conjunto de caracteres definido por el usuario, almacenados en CGRAM. La mayoría de aplicaciones sólo interactúan con DDRAM.

En un LCD de 16 x 2 la DDRAM contiene 80 localidades, 40 para cada línea. Las direcciones en la primera línea van de la 0x00 a la 0x27, en la segunda línea éstas van de la 0x40 a la 0x67. No obstante, sólo es posible mostrar 32 caracteres, por lo que en las localidades de la 0x10 a la 0x27 y de la 0x50 a la 0x67 se almacenan referencias de caracteres que no quedan visibles al usuario. Estos caracteres se van a visualizar cuando se utilicen los comandos de desplazamiento del display. En la figura 8.11 se observa cómo la pantalla aparenta una ventana que muestra 32 caracteres de 80 disponibles, para mostrar a los otros, se debe recorrer la ventana en este espacio.

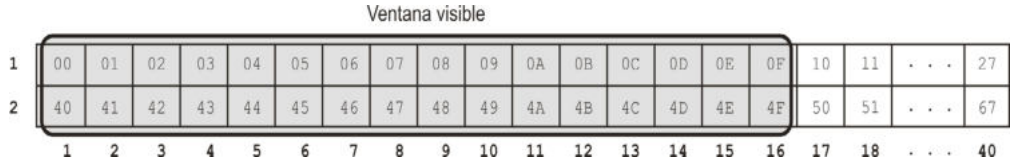


Figura 8.11 En la pantalla se visualizan 32 de los 80 caracteres disponibles

Se tienen diferentes comandos para el acceso a la DDRAM, con el comando *Configura Dirección en DDRAM* se inicializa al contador de dirección, se debe aplicar antes de leer o escribir. Para escribir en DDRAM se utiliza al comando *Escribe Dato* y para leer se tiene al comando *Lee Dato*. El contador de dirección se puede incrementar o disminuir automáticamente después de un acceso a DDRAM, o permanecer sin cambios, dependiendo de su configuración.

La CGROM es otro de los espacios de memoria en el controlador de un LCD, este espacio contiene los mapas de bits de los caracteres que pueden ser mostrados en la pantalla, en la figura 8.12 se muestra su contenido. Los mapas de bits codificados en la CGROM incluyen al conjunto básico de caracteres ASCII, caracteres japoneses y caracteres griegos, dispuestos en una matriz de 16 x 12 mapas de bits.

En las primeras 10 columnas los mapas de bits tienen una organización de 7 x 5 pixeles, mientras que en las últimas 2 su organización corresponde con 10 x 5 pixeles.

Figura 8.12 Contenido de la CGROM

La dirección de cada mapa de bits es de un byte, en el primer renglón de la figura 8.12 se muestra el nibble alto del byte de dirección y en la primera columna su nibble bajo. Puede notarse que para los caracteres ASCCI su dirección es la correcta, 0x20 para el espacio en blanco, 0x21 para '!', etc., hasta 0x7D que es la codificación de '}'.

La DDRAM y la CGROM están relacionadas, el código que se almacena en DDRAM hace referencia a una posición en la CGROM. El usuario escribe en DDRAM un conjunto de números y el controlador los toma como direcciones de acceso a la CGROM, para obtener los mapas de bits y mostrarlos en la pantalla. No existen comandos para leer o escribir directamente en CGROM.

El tercer espacio de memoria es la CGRAM, éste es un espacio para 8 mapas de bits personalizados, con un tamaño de 5 x 8 pixeles y direcciones de la 0x00 a la 0x07. Un mapa de bits de la CGRAM se obtiene de la misma manera que uno de la CGROM, es decir, si en la DDRAM se escribe un número entre el 0x00 y el 0x07, se hace referencia a CGRAM en vez de a CGROM.

Si es posible un acceso directo a la CGRAM. El comando *Configura Dirección en CGRAM* hace que el apuntador de dirección haga referencia a CGRAM. Las direcciones para la CGRAM son de 6 bits, de los cuales, los 3 bits más significativos determinan la ubicación del mapa de bits y los 3 bits menos significativos hacen referencia a cada uno de sus renglones.

Para escribir o leer en CGRAM se utilizan los comandos *Escribe Dato* y *Lee Dato*, respectivamente. Son los mismos con los que se tiene acceso a DDRAM, por ello, el acceso a uno u otro espacio se determina por el último comando utilizado para definir la dirección, pudiendo ser *Configura Dirección en DDRAM* o *Configura Dirección en CGRAM*. Por la organización de los mapas de bits en CGRAM, únicamente los 5 bits menos significativos en cada dato son válidos.

Por ejemplo, para colocar el mapa de bits con el dibujo de un rombo, como el de la figura 8.13, en la dirección 0x05 de la CGRAM, se deben escribir los datos mostrados en la tabla 8.3, en sus direcciones correspondientes.

8.4.2 Conexión de un LCD con un Microcontrolador

Aunque el bus de datos de un LCD es de 8 bits (ver figura 8.9), para conectarse con un MCU es posible utilizar una interfaz de 4 o de 8 bits. La interfaz de 4 bits es recomendable cuando el MCU tiene pocas terminales de entrada/salida. En la figura 8.14 (a) se muestra la conexión de un LCD con un ATMega8, utilizando una interfaz de 4 bits, se observa que con el puerto C (7 bits) es suficiente para su manejo. La interfaz de 8 bits requiere más terminales de entrada/salida, en la figura 8.14 (b) se muestra la conexión de un LCD con un ATMega16, al emplear una interfaz de 8 bits se requiere un puerto completo más 3 bits de otro puerto.

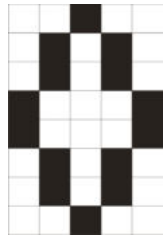


Figura 8.13 Mapa de bits personalizado a ubicar en la dirección 0x05 de CGRAM

Tabla 8.3 Datos que describen el mapa de bits de la figura 8.13

Dirección		Dato	
Binario	**Hexadecimal**	**Binario**	**Hexadecimal**
101 000	28	00100	04
101 001	29	01010	0A
101 010	2A	01010	0A
101 011	2B	10001	11
101 100	2C	10001	11
101 101	2D	01010	0A
101 110	2E	01010	0A
101 111	2F	00100	04

La interfaz de 4 bits tiene la ventaja de requerir menos terminales y la desventaja de un acceso más lento. Una interfaz de 4 bits requiere de 2 accesos al LCD para escribir o leer un dato, puesto que los datos son de 8 bits.

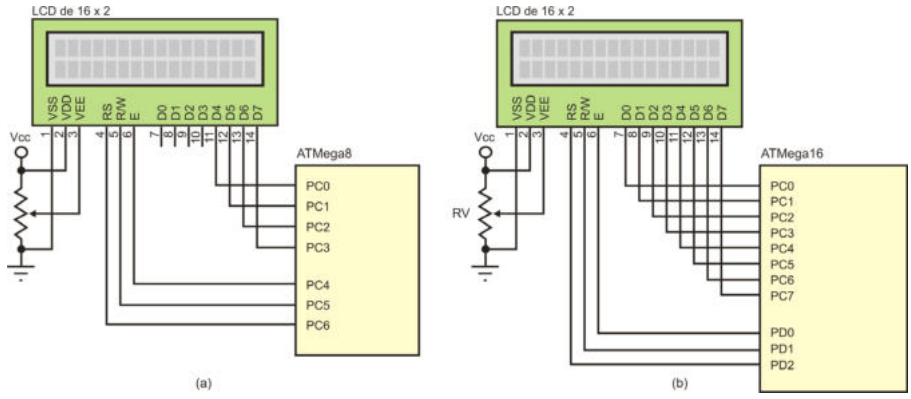


Figura 8.14 Conexión de un LCD con un MCU, con una interfaz de (a) 4 bits y (b) de 8 bits

Dos recomendaciones para simplificar el hardware son:

- Conectar la terminal VEE permanentemente con tierra, con ello, el contraste es constante pero la visibilidad es aceptable.

- Conectar la terminal RW permanentemente con tierra y sólo realizar escrituras. Las lecturas del LCD sirven para saber si está listo y obtener el valor del contador de dirección. El LCD está listo si después de ejecutar un comando el MCU se espera el tiempo que el comando requiere. La dirección actual se define desde el MCU, no debería ser necesario obtenerla del LCD. Por lo tanto, es posible manipular un LCD realizando únicamente escrituras.

8.4.3 Transferencias de Datos

En la figura 8.15 se muestra la temporización de las señales para un ciclo de escritura en el LCD. En la señal RS se observa que para un 0 lógico se requiere un voltaje menor a 0.6 V y para un 1 lógico un voltaje mayor a 2.2 V, estos niveles también se aplican en las otras señales.

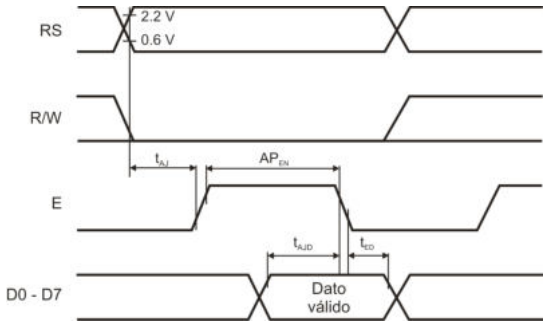


Figura 8.15 Ciclo de escritura en el LCD

Las señales de control RS y R/W deben permanecer estables por lo menos 140 nS antes del pulso de habilitación (t_{AJ}). La señal RS debe tener un 0 ó 1, según se requiera escribir un comando o un dato. R/W debe tener 0 para escrituras. La duración del pulso de habilitación debe ser al menos de 450 nS (AP_{EN}). Se observa en la figura que realmente la escritura se realiza durante el flanco de bajada de la señal E. Antes del flanco de bajada, el dato a escribir debe estar estable por un tiempo mínimo de 195 nS (t_{AJD}) y después del flanco, el dato aún debe continuar estable por un tiempo mínimo de 10 nS (t_{ED}).

Un ciclo de lectura es similar a uno de escritura, en niveles de voltaje y algunos tiempos, esto puede verse en la figura 8.16. En este caso, la señal R/W se mantiene en un nivel lógico alto. El dato leído está disponible después de un tiempo máximo de 350 nS (t_{DD}), a partir del flanco de subida, y se mantiene por un tiempo mínimo de 20 nS (t_{DH}) después del flanco de bajada.

Cuando se conecta el LCD con un ATMega8 o un ATMega16, es conveniente realizar una función o rutina que genere el pulso de habilitación. Si el MCU trabaja con el oscilador interno de 1 MHz, el ciclo de reloj es de 1 uS, de manera que es suficiente con poner un nivel alto en la terminal E en una instrucción y en la siguiente, ponerlo en bajo.

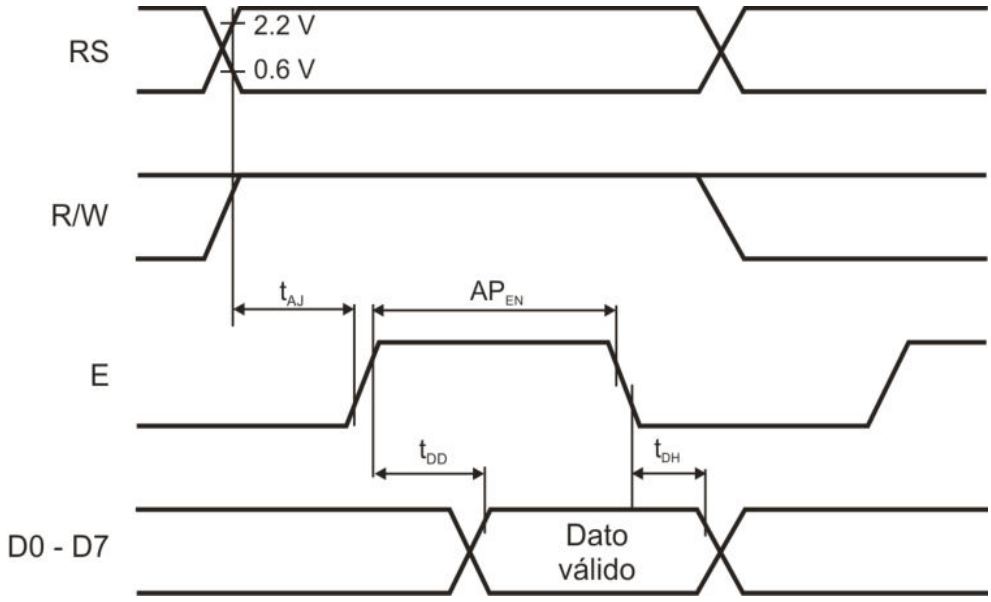


Figura 8.16 Ciclo de lectura en el LCD

El código para esta función, asumiendo que el LCD está conectado como se muestra en la figura 8.14 (a), es:

```
void  LCD_pulso_E( ) {
PORTC = PORTC | 0x10;    // Coloca un nivel alto en la terminal E
PORTC = PORTC & 0xEF;    // Coloca un nivel bajo en la terminal E
}
```

Con esta función, el procedimiento para una escritura es:

1. Asignar a RS el valor de 0 ó 1, según se requiera escribir un comando o un dato.
2. R/W = 0.

3. Colocar el dato a escribir en el bus de datos.

4. Llamar a la función LCD_pulso_E.

Para una lectura, el procedimiento a seguir es:

1. Asignar a RS el valor de 0 ó 1, según se requiera leer un comando o un dato.

2. R/W = 1.

3. Llamar a la función LCD_pulso_E.

4. El dato leído está disponible en el bus de datos.

En una interfaz de 4 bits, el tiempo mínimo necesario entre nibbles es de 1 µS, primero se debe escribir o leer al nibble más significativo e inmediatamente al menos significativo. El tiempo de 1 µS para el cambio de nibbles se consume en la ejecución de las instrucciones. En la figura 8.17 se comparan las transferencias de 4 y 8 bits, en una secuencia que escribe un comando, escribe un dato y luego lee un comando.

8.4.4 Comandos para el Acceso de un LCD

En la tabla 8.4 se muestran los comandos requeridos para el manejo de un LCD, incluyendo las opciones que se tienen para la escritura y lectura de datos. En las siguientes subsecciones se describe la funcionalidad de cada uno de ellos.

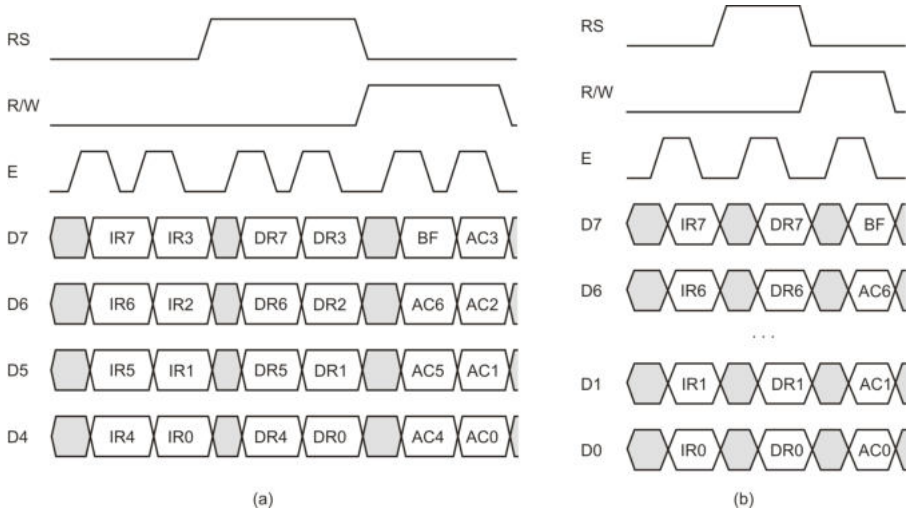


Figura 8.17 Transferencias de datos en una interfaz (a) de 4 bits y (b) de 8 bits

Tabla 8.4 Comandos del LCD

Comandos del LCD	RS	RW	Nibble Alto				Nibble Bajo			
			DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
Limpieza del Display	0	0	0	0	0	0	0	0	0	1
Regreso del Cursos al inicio	0	0	0	0	0	0	0	0	1	X
Ajuste de entrada de Datos	0	0	0	0	0	0	0	1	1/D	S
Encendido/Apagado del Display	0	0	0	0	0	0	1	D	C	B
Desplazamiento del Cursor y del Display	0	0	0	0	0	1	S/C	R/L	X	X
Configura la Función del Display	0	0	0	0	1	DL	N	F	X	X
Configura Dirección en CGRAM	0	0	0	1	A5	A4	A3	A2	A1	A0
Configura Dirección en DDRAM	0	0	1	A6	A5	A4	A3	A2	A1	A0
Lee la Bandera de Ocupado y la Dirección	0	1	BF	A6	A5	A4	A3	A2	A1	A0
Escribe Dato en CGRAM o en DDRAM	1	0	D7	D6	D5	D4	D3	D2	D1	D0
Lee Dato de CGRAM o de DDRAM	1	1	D7	D6	D5	D4	D3	D2	D1	D0

8.4.4.1 Limpieza del Display

Este comando se emplea para limpiar al display y regresar el cursor a la posición de inicio, la esquina superior izquierda. La limpieza del display consiste en colocar caracteres de espacio en blanco (0x20) en todas las localidades de DDRAM. El contador de direcciones se reinicia con 0, haciendo referencia a la DDRAM y queda configurado para que automáticamente se incremente después de la lectura o escritura de un dato. El comando requiere de un tiempo de ejecución entre 82 μS y 1.64 mS.

8.4.4.2 Regreso del Cursor al Inicio

Con este comando se regresa el cursor a la posición de inicio, esquina superior izquierda, pero sin modificar el contenido de la DDRAM. Si el display fue desplazado, también se regresa a su posición original. El contador de direcciones se reinicia con 0, haciendo referencia a la DDRAM. El cursor o el parpadeo también se mueven a la posición de inicio. Para la ejecución de este comando se requiere un tiempo entre 40 μS y 1.6 mS.

8.4.4.3 Ajuste de Entrada de Datos

Con este comando se configura la dirección de movimiento del cursor y se especifica si el display debe ser desplazado. Estas operaciones se realizan después de la lectura o escritura de un dato. El comando requiere de 40 μS para su ejecución y sus argumentos se definen en la tabla 8.5.

Tabla 8.5 Argumentos para el comando de Ajuste de Entrada de Datos

	Bit I/D	Bit S
0	El contador de dirección disminuye automáticamente	Desplazamiento inhabilitado
1	El contador de dirección se auto-incrementa	Desplaza al display completo en la dirección definida en el bit I/D, ante una escritura en DDRAM

8.4.4.4 Encendido/Apagado del Display

Comando empleado para encender o apagar al display, si el display está apagado los caracteres no se muestran, sin embargo, el contenido de la DDRAM no se pierde. También determina si se muestra al cursor o si se exhibe un parpadeo, el cual tiene un periodo aproximado de medio segundo. Requiere de 40 μS para su ejecución y sus argumentos se muestran en la tabla 8.6.

Tabla 8.6 Argumentos para el comando de Encendido/Apagado del Display

	Bit D	Bit C	Bit B
0	Display apagado	Sin cursor	Sin parpadeo
1	Display encendido	Con cursor	Con parpadeo

8.4.4.5 Desplazamiento del Cursor y del Display

Desplaza al cursor y al display sin tener acceso a DDRAM, es decir, sin realizar alguna escritura o lectura de datos. Requiere de 40 μS y las acciones ante los diferentes valores de los argumentos se muestran en la tabla 8.7.

Tabla 8.7 Opciones para el desplazamiento del cursor y del display

Bit S/C	Bit R/L	Función
0	0	Desplaza al cursor a la izquierda, el contador de direcciones disminuye en 1
0	1	Desplaza al cursor a la derecha, el contador de direcciones se incrementa en 1
1	0	Desplaza al display completo a la izquierda, el cursor sigue este desplazamiento. El contador de direcciones no cambia
1	1	Desplaza al display completo a la derecha, el cursor sigue este desplazamiento. El contador de direcciones no cambia

8.4.4.6 Configura la Función del Display

Con este comando se configuran 3 parámetros: El tamaño de la interfaz, el número de líneas del display y el tamaño de los mapas de bits. El comando requiere de 40 μS para su ejecución y sus argumentos se muestran en la tabla 8.8.

Tabla 8.8 Argumentos para la configuración del display

	Bit DL	Bit N	Bit F
0	4 bits	1 línea	5 x 7 puntos
1	8 bits	2 líneas	5 x 10 puntos

Algunos LCDs que físicamente se organizan como 16 caracteres en 1 línea (16 x 1), realmente funcionan como displays de 8 x 2. Requieren configurarse como LCDs de 2 líneas y para escribir en las posiciones de la 8 a la 16 es necesario hacer el cambio de línea.

8.4.4.7 Configura Dirección en CGRAM

Este comando es para configurar la dirección de acceso a CGRAM. El contador de direcciones queda referenciando a la CGRAM, de manera que las subsecuentes escrituras y lecturas de datos se hacen en CGRAM. El comando requiere de 40 μs para su ejecución.

8.4.4.8 Configura Dirección en DDRAM

Este comando es para configurar la dirección de acceso a DDRAM. El contador de direcciones queda referenciando a la DDRAM, de manera que las subsecuentes escrituras y lecturas de datos se hacen en DDRAM. El comando requiere de 40 μs para su ejecución.

8.4.4.9 Lee la Bandera de Ocupado y la Dirección

Este es el único comando de lectura, como se observa en la tabla 8.4, obtiene un byte en donde el bit más significativo es la bandera de ocupado (BF, *busy flag*) y los otros 7 bits contienen el valor del contador de direcciones.

La bandera BF está en alto si hay una operación en progreso, por lo que no es posible ejecutar otro comando en el LCD. El comando requiere de 1 μs para su ejecución, de manera que puede ser más eficiente evaluar la bandera en lugar de esperar el tiempo requerido por cada comando.

El comando también regresa el valor del contador de direcciones. Dado que el contador de direcciones es usado por DDRAM o CGRAM, la dirección que devuelve está en el contexto de la última configuración de la dirección.

8.4.4.10 Escribe Dato en CGRAM o en DDRAM

Con RS en alto y RW en bajo se realiza la escritura de un dato en DDRAM o CGRAM. El espacio al que se tiene acceso depende del último comando empleado para configurar la dirección. Si se empleó al comando *Configura Dirección en DDRAM*, el dato es escrito en DDRAM, pero si el comando fue *Configura Dirección en CGRAM*, el dato se escribe en CGRAM.

La escritura de un dato requiere de un tiempo de 40 μs, después de ésta, el contador de dirección se incrementa o reduce en 1, según como se haya definido con el comando *Ajuste de Entrada de Datos*. También se realiza el desplazamiento del display, si éste fue habilitado con el mismo comando.

8.4.4.11 Lee Dato de CGRAM o de DDRAM

Con RS y RW en alto se realiza la lectura de un dato en DDRAM o CGRAM. El espacio al que se tiene acceso depende del último comando empleado para configurar la dirección. Si se empleó al comando *Configura Dirección en DDRAM,* el dato es leído de DDRAM, pero si el comando fue *Configura Dirección en CGRAM*, el dato se lee de CGRAM.

La lectura de un dato requiere de un tiempo de 40 μs, después de ésta, el contador de dirección se incrementa o reduce en 1, según como se haya definido con el comando *Ajuste de Entrada de Datos*. No obstante, una lectura no realiza desplazamientos del display.

8.4.5 Inicialización del LCD

Un LCD debe inicializarse antes de que pueda ser utilizado. La inicialización consiste en el envío temporizado de una secuencia de comandos.

En la figura 8.18 se muestra la secuencia a seguir para inicializar al LCD con una interfaz de 4 bits. La secuencia es proporcionada por los fabricantes y los tiempos establecidos son fundamentales para la correcta inicialización del LCD. Aunque el bus de datos del LCD es de 8 bits, únicamente están conectados los 4 bits más significativos. Con estos bits es suficiente para realizar la inicialización.

En los pasos 1, 2 y 3 la interfaz aún es de 8 bits, en el nibble más significativo aparece el comando con el que se *Configura la Función del Display* y el bit DB4 está en alto para indicar una interfaz de 8 bits.

Después del paso 3, en la figura 8.18, no se indican los tiempos de espera debido a que ya es posible verificar la bandera de ocupado (*Busy Flag*). O en su defecto, esperar el tiempo necesario para la ejecución del comando. En el paso 4 se define la interfaz de 4 bits, siendo el último paso en el que sólo se mandan los 4 bits más significativos.

En los comandos siguientes se mandan los 8 bits, primero al nibble alto y después al nibble bajo. El tiempo de espera entre uno y otro nibble es de 1 μs, de manera que el envío del nibble bajo es inmediato al envío del nibble alto, después de ello, es necesario observar a la bandera BF antes de enviar otro comando, o bien, esperar el tiempo requerido para la ejecución del comando.

El paso 8 puede omitirse si la configuración obtenida con el comando *Limpieza del Display* es adecuada para la aplicación.

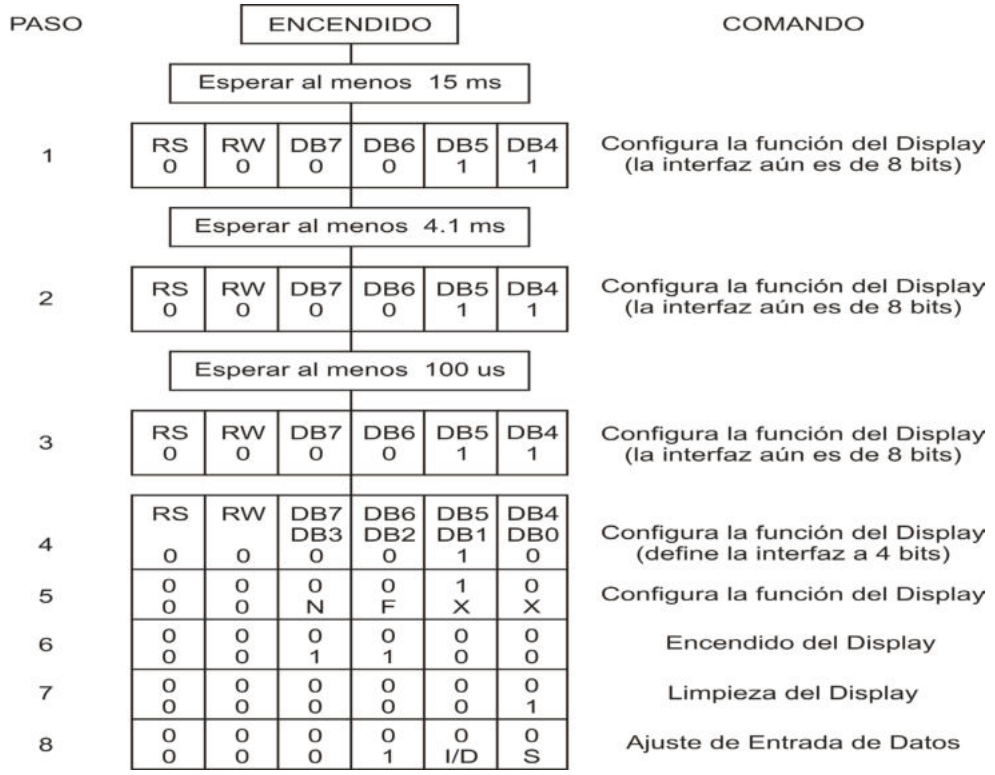


Figura 8.18 Secuencia para inicializar un LCD con una interfaz de 4 bits

La inicialización con una interfaz de 8 bits es similar a la de 4 bits, en la figura 8.19 se muestra la secuencia requerida para ello.

Nuevamente los tiempos de espera señalados en la secuencia se vuelven fundamentales. De manera similar, después del paso 3 no se indica el tiempo de espera porque ya es posible monitorear la bandera de ocupado (BF), o bien, esperar el tiempo requerido por el correspondiente comando. El paso 7 puede omitirse si la configuración obtenida con el comando *Limpieza del Display* es adecuada para la aplicación.

Una vez que se ha inicializado al LCD ya es posible escribir caracteres y ejecutar otros comandos, con la finalidad de mostrar el estado de un sistema. Debe tomarse en cuenta el orden de las señales requerido para las escrituras y lecturas, como se describió en la sección 8.4.3.

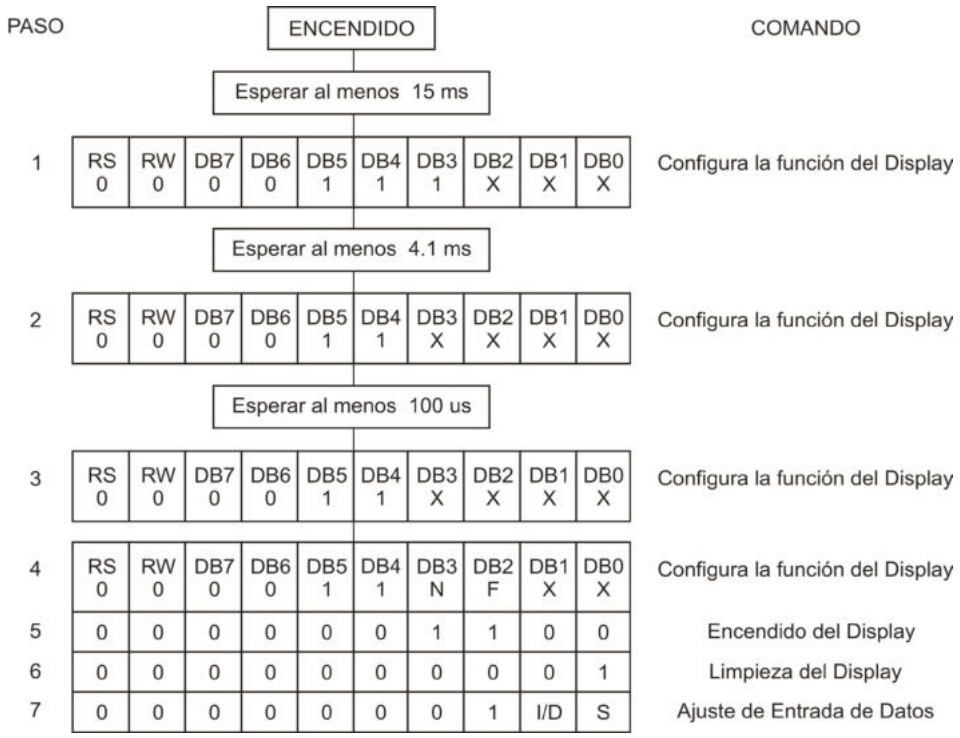


Figura 8.19 Secuencia para inicializar un LCD con una interfaz de 8 bits

8.5 Manejo de Motores

En esta sección se presentan algunas ideas para manejar motores de CD, motores paso a paso y servo motores. Estas ideas se ilustran empleando transistores bipolares, aunque también es posible el uso de MOSFETs, IGBTs o circuitos dedicados, como drivers de motores.

8.5.1 Motores de CD

Un motor de CD no debe conectarse directamente a una terminal de un MCU, porque la corriente que proporciona no es suficiente para su manejo. En la figura 8.20 se muestra una configuración básica con la que se puede activar o desactivar un motor de CD empleando un transistor NPN. Para modificar la velocidad, puede utilizarse una señal PWM en la activación del transistor.

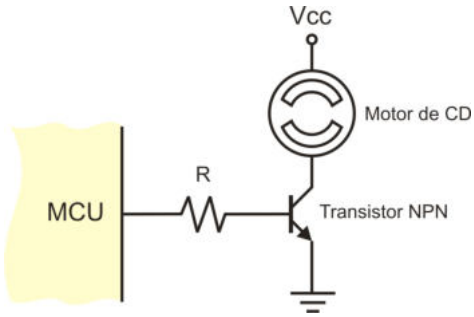


Figura 8.20 Conexión de un motor a una terminal de un MCU

Si el motor es de poca potencia, para su manejo es suficiente con un transistor de propósito general, como un BC548 o un 2N2222. Si requiere un poco más de corriente, puede emplearse un transistor TIP120 u otro transistor de potencia.

Para una corriente mayor, el transistor puede reemplazarse por un par Darlington, como el que se muestra en la figura 8.21, con esta configuración, la ganancia en corriente es el producto de las ganancias de los transistores individuales. Existen versiones integradas de configuraciones tipo Darlington, como la serie de circuitos ULN2001, ULN2002, ULN2003 and ULN2004, de la firma ST Microelectronics; aunque también pueden armarse con transistores discretos.

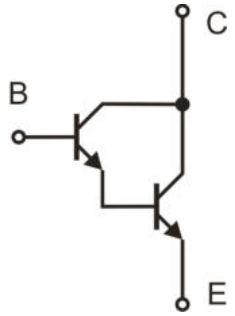


Figura 8.21 Un par Darlington puede remplazar a un transistor

Si se requiere el control de la dirección de movimiento de un motor, puede utilizarse una configuración de 4 transistores conocida como puente H. En la figura 8.22 se muestra un puente H compuesto de 2 transistores NPN y 2 transistores PNP, quedando disponibles 2 terminales de control (A y B), para el manejo de la dirección. Los 4 transistores operan como interruptores, sin embargo, los NPN se activan con un 1 lógico y los PNP con un 0 lógico. Las entradas de control pueden conectarse directamente a las salidas de un ATMega8 o de un ATMega16, o bien, puede colocarse un resistor de 1 Kohm o menor, para limitar la corriente. Los diodos en el puente evitan que el motor almacene energía, podrían omitirse para motores pequeños.

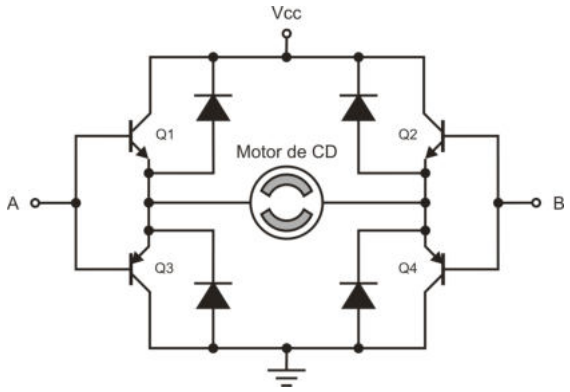


Figura 8.22 Organización de un puente H

Si en ambas terminales de control se colocan 0's, intentarían activarse los transistores Q3 y Q4. Esto no es posible porque no hay una diferencia de potencial en sus uniones base-emisor, por lo tanto, el motor está desenergizado y sin movimiento. Algo similar ocurre colocando 1's, en este caso intentarían activarse los transistores Q1 y Q2, dejando nuevamente al motor sin movimiento.

El motor se mueve en la dirección de las manecillas del reloj si A = 1 y B = 0, dado que quedan activos los transistores Q1 y Q4. Y se mueve en dirección contraria si A = 0 y B = 1, siendo Q2 y Q3 los transistores activos.

Para que adicionalmente se manipule la velocidad, una opción consiste en el agregado de una compuerta AND en cada una de las terminales de control, como se muestra en la figura 8.23. De esta manera, se tienen nuevas terminales A' y B' para determinar la dirección de movimiento, así como una terminal adicional para una señal PWM, con la que se determina la velocidad.

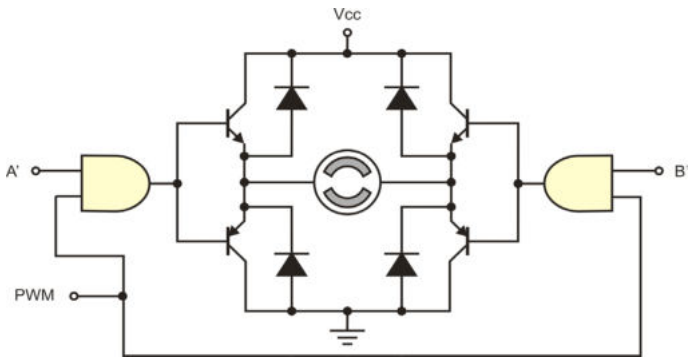


Figura 8.23 Acondicionamiento de un puente H para poder modificar la velocidad

Los circuitos integrados L293B, L293D y L293E de ST Microelectronics son drivers con los que se pueden manipular motores directamente o se pueden configurar como un puente H, el LMD18200 de National Semiconductor es un puente H integrado, capaz de manejar hasta 3 A de corriente.

8.5.2 Motores Paso a Paso

Los motores paso a paso (MPAP) forman parte de los llamados motores de conmutación electrónica. Son los más apropiados para la construcción de mecanismos en donde se requieren movimientos muy precisos. A diferencia de un motor de CD, un MPAP no tiene escobillas ni conmutador mecánico. En lugar de ello, la acción de conmutación necesaria para su funcionamiento se realiza con un controlador externo, el cual va a polarizar las bobinas del estator para imponer polos magnéticos. El rotor no tiene devanado de armadura, básicamente es una colección de imanes permanentes que se mueven para alinearse con el estator.

El controlador debe generar los pulsos para polarizar a las bobinas, moviendo al rotor un paso a la vez. El alcance de un paso es variable, depende de la construcción física del motor, puede ser sólo de 1.8° ó hasta de 90°. Para completar una vuelta, con 1.8° se requiere de 200 pasos y con 90° únicamente de 4. Esta precisión en el movimiento por paso hace a los MPAP ideales para sistemas de lazo abierto.

Existen 2 tipos de MPAP con rotor de imán permanente, el motor bipolar y el motor unipolar. En un motor bipolar, la corriente en las bobinas del estator va a fluir en ambas direcciones, por ello su control es ligeramente más complejo. En un motor unipolar, la corriente fluye en una sola dirección.

En la figura 8.24 se esquematizan ambos tipos de motores, en el MPAP bipolar se observan 4 cables y en el unipolar 6. Si se tuvieran 5 cables, sería un motor unipolar con una conexión interna de los puntos comunes.

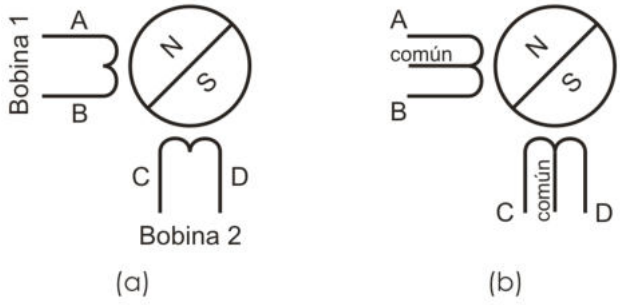


Figura 8.24 Motor paso a paso (a) bipolar y (b) unipolar

Con respecto a la figura 8.24 (a), para la bobina 1, la corriente va a fluir de A a B, o de B hacia A, en función de ello, en la bobina se impone un polo Norte o Sur magnético, ocasionando la alineación del rotor con los polos inducidos en el estator. La polarización de la bobina 2 refuerza la posición deseada en el rotor.

En un motor unipolar, el punto común generalmente es conectado a Vcc, de manera que por medio de transistores se lleva a los puntos A, B, C o D a tierra. Aunque es menos frecuente, el punto común también podría ser conectado a tierra. Al hacer circular la corriente en una o dos bobinas se van a inducir polos magnéticos en el estator, y con ello, el rotor va a moverse hasta quedar alineado con el estator.

8.5.2.1 Polarización y Operación de un Motor Bipolar

Para cada bobina de un MPAP bipolar debe emplearse un puente H, para permitir la polarización en ambos sentidos, éste puede construirse con elementos discretos, como se mostró en la figura 8.22, o bien, se pueden emplear drivers integrados en un chip.

Una opción es el L293B, de la firma ST Microelectronics, el circuito dispone de 4 drivers y cada uno es capaz de proporcionar hasta 1 A de corriente. En la figura 8.25 se muestra la conexión de un MPAP bipolar con un MCU, ATMega8 o ATMega16, utilizando un L293B, en donde se han configurado sus drivers como 2 puentes H.

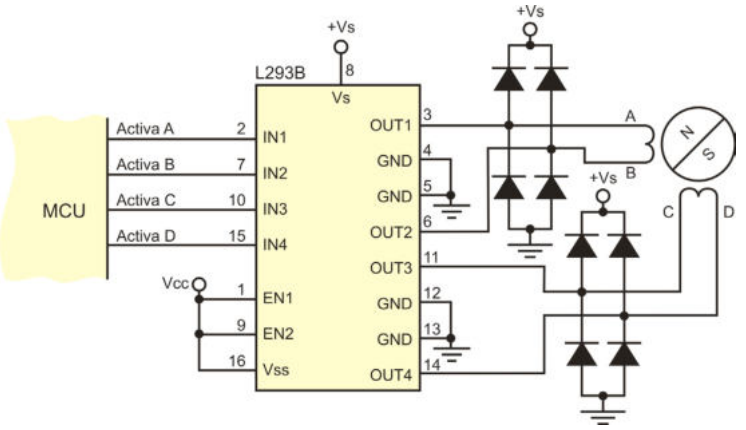


Figura 8.25 Conexión de un motor paso a paso bipolar, utilizando un L293B

El circuito L293B tiene 2 voltajes de alimentación, Vss es el voltaje lógico y puede conectarse a la fuente del MCU y Vs es el voltaje para el motor, pudiendo manejar hasta 36 V. Los diodos evitan que el motor almacene energía y pueden omitirse con motores pequeños.

En la figura 8.26 se muestra cómo se va a mover el rotor del motor al cambiar la polaridad de las bobinas, porque con ello se modifica la dirección del flujo magnético. El rotor tiende a alinearse con los polos magnéticos impuestos en el estator, realizando un movimiento de 90° en cada paso, en dirección de las manecillas del reloj.

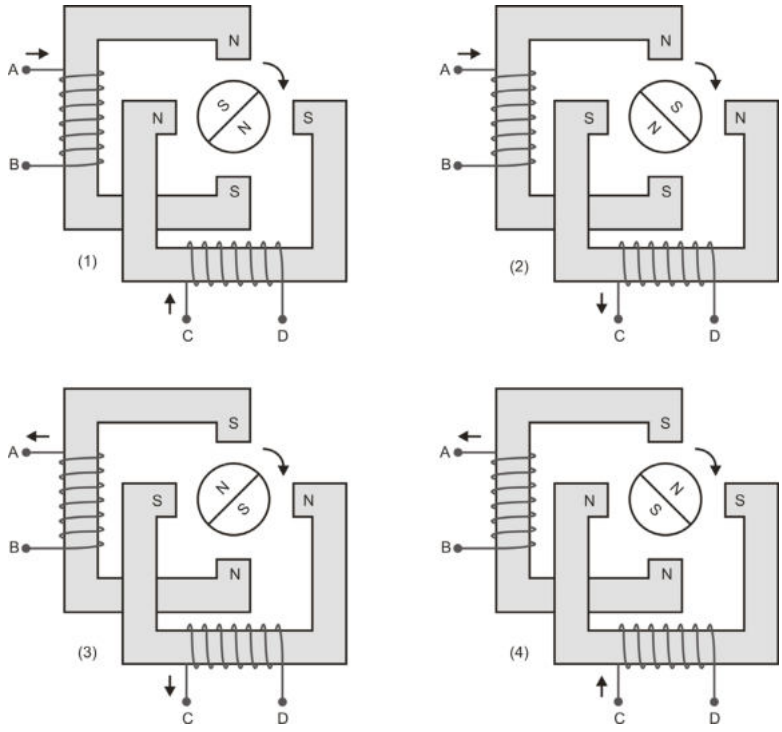


Figura 8.26 Operación de un motor paso a paso bipolar

En la tabla 8.9 se muestra el orden que se debe seguir en la polarización de las bobinas de un motor bipolar, para conseguir el movimiento mostrado en la figura 8.26, una vez que se ha llegado al paso 4, nuevamente debe iniciarse con el paso 1. Para un movimiento en contra de las manecillas del reloj, se debe seguir la secuencia en orden inverso. También se muestran las salidas que debe generar el MCU.

Tabla 8.9 Secuencia para controlar motores paso a paso bipolares

Paso	**Terminales (Bobina)**				**Salidas (MCU)**			
	A	B	C	D	A	B	C	D
1	+V	-V	+V	-V	1	0	1	0
2	+V	-V	-V	+V	1	0	0	1
3	-V	+V	-V	+V	0	1	0	1
4	-V	+V	+V	-V	0	1	1	0

En un motor real, el estator tiene una forma física más compleja, pero conserva la alternancia entre una y otra bobina, de manera que se tiene un mayor número de polos. El rotor tiene más polos magnéticos permanentes para conseguir pasos más pequeños.

8.5.2.2 Polarización y Operación de un Motor Unipolar

En un MPAP unipolar, el común en las bobinas es conectado a Vcc. El otro extremo es llevado a tierra a través de un transistor, como se muestra en la figura 8.27, donde el MCU controla el disparo de los transistores y con ello energiza a alguna de las bobinas, imponiendo un polo magnético Norte. En lugar de usar transistores discretos, para el manejo de un MPAP unipolar puede emplearse un ULN2803, el cual es un arreglo de 8 transistores tipo Darlington, capaces de suministrar hasta 500 mA, cuyas entradas pueden provenir directamente de un MCU.

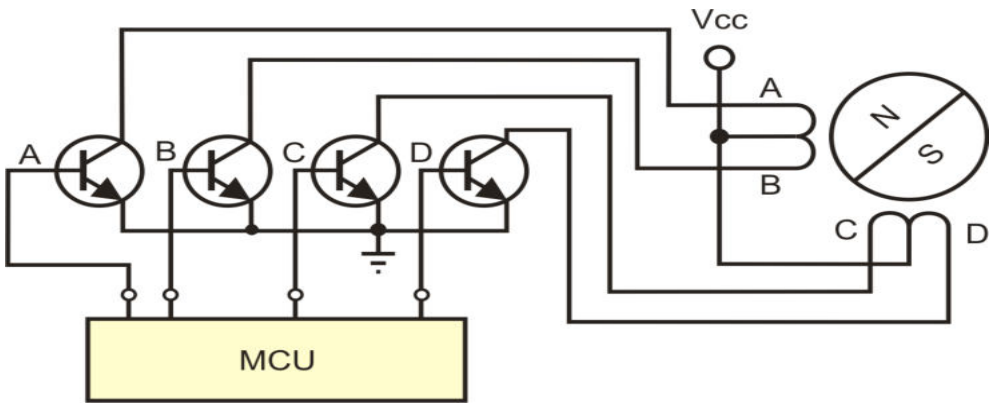


Figura 8.27 Polarización de un motor paso a paso unipolar

Para la operación de los motores unipolares se tienen 3 tipos de secuencias, las cuales se describen e ilustran en las tablas 8.10, 8.11 y 8.12. En todos los casos, una vez que se ha llegado al último paso, para continuar con el movimiento nuevamente debe repetirse el primero. La secuencia debe realizarse en orden inverso para invertir la dirección del movimiento.

La secuencia de la tabla 8.10 impone un mayor torque y retención, al activar a dos bobinas simultáneamente, por ello, esta secuencia es recomendada por los fabricantes.

Tabla 8.10 Secuencia para controlar motores unipolares, activando 2 bobinas

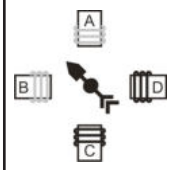
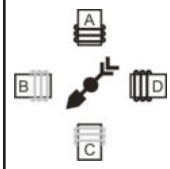
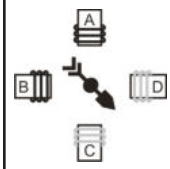
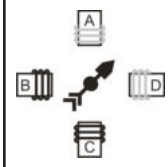
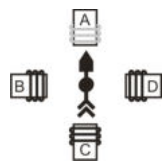
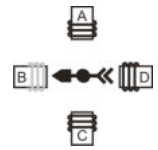
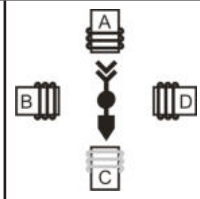
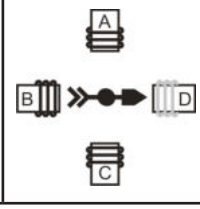
Paso	A	B	C	D	Efecto
1	1	1	0	0	
2	0	1	1	0	
3	0	0	1	1	
4	1	0	0	1	

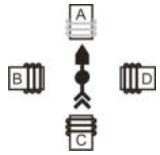
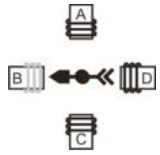
Tabla 8.11 Secuencia para controlar motores unipolares, activando 1 bobina

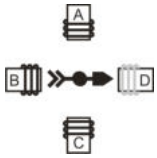
Paso	A	B	C	D	Efecto
1	1	0	0	0	
2	0	1	0	0	

Paso	A	B	C	D	Efecto
3	0	0	1	0	
4	0	0	0	1	

En la secuencia mostrada en la tabla 8.11 se reduce el torque y la retención al activar únicamente una bobina a la vez. La secuencia de la tabla 8.12 es una combinación de las dos secuencias anteriores, consiguiendo con ello pasos con menores dimensiones, conocidos como medios pasos.

Tabla 8.12 Secuencia para manejar medios pasos en un moto paso a paso unipolar

Paso	A	B	C	D	Efecto
1	1	0	0	0	
2	1	1	0	0	
3	0	1	0	0	
4	0	1	1	0	

Paso	A	B	C	D	Efecto
5	0	0	1	0	
6	0	0	1	1	
7	0	0	0	1	
8	1	0	0	1	

No debe alterarse el orden de la secuencia, por ejemplo, si un motor se mueve 5 medios pasos siguiendo la secuencia de la tabla 8.12, la última salida del MCU activa al transistor C. Después de ello, si se requiere otro medio paso en el mismo sentido, se deben activar los transistores C y D. Si el medio paso es en dirección contraria, los transistores a activar son B y C. Es decir, al cambiar el sentido del movimiento se debe considerar la última salida y no regresar a la posición inicial. En el ejemplo 8.3 se ilustra cómo conservar la secuencia.

Otro aspecto es el tiempo de establecimiento del rotor, esto es, una vez que se han polarizado las bobinas es necesario esperar por un intervalo de tiempo para que los polos magnéticos del rotor se alineen con los polos inducidos en el estator. Este tiempo se determina en forma práctica.

Ejemplo 8.3 Suponga que un MPAP unipolar está conectado en el nibble menos significativo del puerto B de un ATMega8 y realice una función en lenguaje C que lo mueva en medios pasos (tomando como base a la tabla 8.12), recibiendo como argumento el número de medios pasos y el sentido del movimiento (0 en dirección de las manecillas del reloj y 1 en dirección contraria). Indique las condiciones necesarias para la ejecución correcta de la función.

Como datos globales se requiere la secuencia de activación de los transistores y una variable que conserve el último paso realizado:

```
// Constantes con la secuencia de salida para los transistores

const char  sec_trans[] PROGMEM ={0x01,0x03,0x02,0x06,

                                  0x04,0x0C,0x08,0x09};

unsigned char  num_paso; // Variable para no perder la secuencia
```

En el programa principal debe haber un par de asignaciones, para establecer la condición inicial:

```
num_paso = 0;           // Inicia con el salida 0 de la secuencia

PORTB = 0x01;           // Primera salida en el puerto B
```

La función para mover a un MPAP unipolar:

```
void  motor_pasos(unsigned charpasos, unsigned char direccion ) {
unsigned char  i;

if( direccion == 0 )

      for( i = 0; i < pasos; i++) {

            num_paso++;

            if(num_paso == 8)

                  num_paso = 0;

            PORTB = pgm_read_byte(&sec_trans[num_paso]);

            _delay_ms(20);    // Espera se realice el movimiento

      }

else

      for( i = 0; i < pasos; i++) {

            num_paso--;

            if(num_paso == 0xFF)

                  num_paso = 7;

            PORTB = pgm_read_byte(&sec_trans[num_paso]);

            _delay_ms(20)     // Espera se realice el movimiento

      }

}
```

_____

En un motor real, se consiguen pasos más pequeños porque las bobinas del estator se organizan físicamente para presentar más de 4 polos. Además, el rotor tiene más polos magnéticos permanentes. El principio de operación es el mismo, aunque en algunos casos podría requerirse alterar la secuencia en los medios pasos, requiriendo polarizar una bobina y posteriormente a las dos frontales más cercanas, buscando que el polo realice un movimiento menor para alinearse.

8.5.3 Servomotores

Los servomotores son actuadores ampliamente utilizados en la construcción de robots u otros sistemas mecatrónicos, permitiendo un control preciso de posición. Su precisión es mayor que la de un motor paso a paso, porque realmente un servomotor es un sistema de control de posición comandado por una señal modulada en ancho de pulso (PWM). Con un rango de movimiento limitado a 180 °.

Un servomotor típicamente incluye un convertidor de voltaje, un amplificador (driver) para el manejo de un motor de CD, un potenciómetro y un conjunto de engranes, todos estos elementos forman un circuito retroalimentado para comandar posición y velocidad. En la figura 8.28 se muestra la organización de un servomotor.

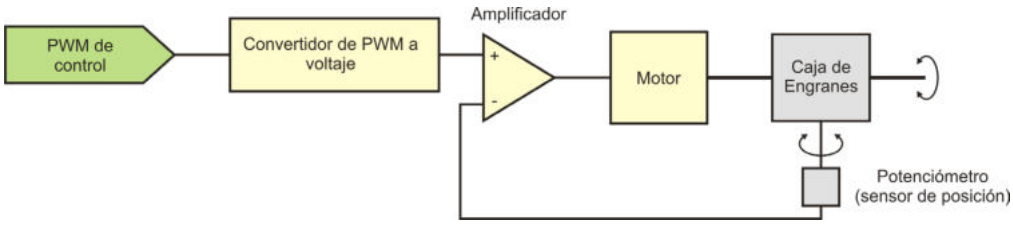


Figura 8.28 Organización de un servomotor

La frecuencia y el ancho de pulso pueden variar entre fabricantes, por ejemplo, los servomotores fabricados por Futaba y Hitech se manejan con una señal PWM de 50 Hz (20 mS). El servomotor está en su posición mínima (0 °) con un ancho del pulso de 0.9 mS, en su posición central (90 °) con 1.5 mS y en su posición máxima (180 °) con 2.1 mS. En la figura 8.29 se ilustran estas 3 posiciones principales, estableciendo un rango lineal para las posiciones intermedias.

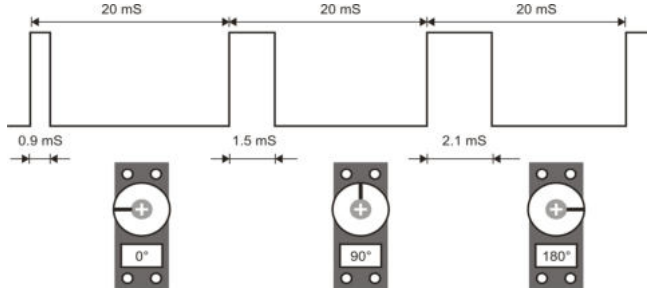


Figura 8.29 Operación de un servomotor

Con el ATMega8 y el ATMega16 se simplifica el manejo de servomotores, al contar con recursos de hardware para generar señales PWM. El temporizador 1 de estos dispositivos es de 16 bits, por lo que fácilmente se pueden conseguir periodos de 20 mS, como se mostró en el ejemplo 4.7.

8.6 Interfaz con Sensores

Dado que los microcontroladores ATMega8 y ATMega16 internamente contienen un convertidor analógico a digital (ADC) y un comparador analógico (AC), cuando se requiere obtener información de algún sensor, básicamente se debe acondicionar la señal que éste proporcione a un voltaje en el rango de la alimentación del microcontrolador, para que la información resultante sea recibida por el ADC o por el AC.

En este sentido, se sugiere el uso de los amplificadores operacionales LM358 o LM324, como elementos de acondicionamiento de señal. La característica principal de estos amplificadores es que pueden operar sólo con una fuente de alimentación, desde 3 hasta 32 Volts. Con ello, los elementos de acondicionamiento pueden alimentarse con la misma fuente del MCU, evitando fuentes adicionales.

El LM358 incluye 2 amplificadores independientes y el LM324 incluye 4. Podría decirse que un LM358 es la mitad de un LM324, en la figura 8.30 se muestran las terminales de ambos dispositivos.

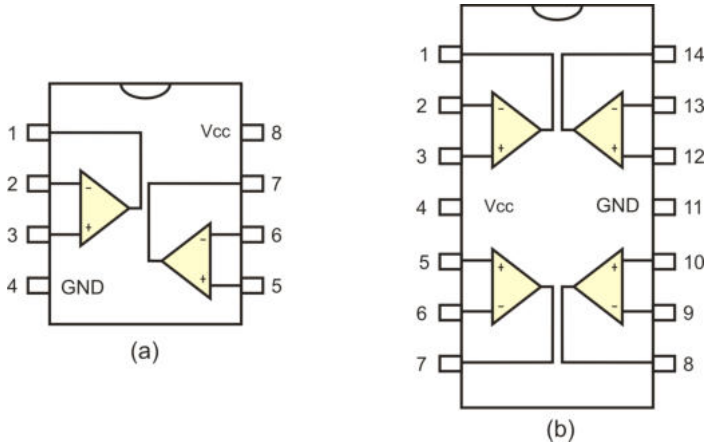


Figura 8.30 Terminales del (a) LM358 y del (b) LM324

Existen diferentes configuraciones típicas para acondicionar la información proporcionada por un sensor, entre las que se encuentran: amplificador inversor, amplificador no inversor, amplificador diferencial, etc. La configuración adecuada depende del tipo de sensor, aunque en algunos casos puede requerirse de un convertidor de corriente a voltaje o de un filtro para eliminar ruido.

Como un ejemplo, en la figura 8.31 se muestra la conexión de un LM35 acondicionado para detectar temperaturas entre 0 y 50 °C. El sensor entrega 10 mV/°C, el amplificador está configurado como no inversor, proporcionando una ganancia de 10 (ganancia =

1 + R2/R1). Por lo tanto, para el rango de 0 a 50 °C se tienen voltajes entre 0 y 5 V. En el programa interno se debe procesar la información, considerando que 0x0000 corresponde con 0 °C y 0x03FF con 50 °C.

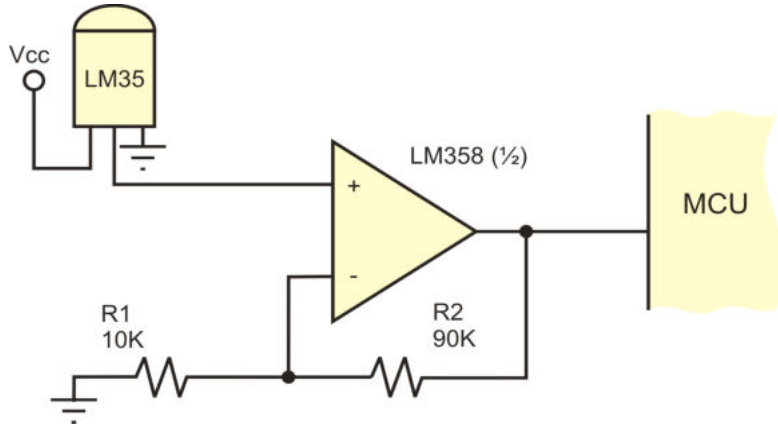


Figura 8.31 Acondicionamiento de un sensor de temperatura

En la práctica no se alcanzan los 5 V en la salida del amplificador operacional porque éste se satura antes de que eso ocurra. El fabricante garantiza un comportamiento lineal con un valor máximo para la salida de Vcc – 1.5V, por ello, para alcanzar un rango de operación más amplio, puede utilizarse un voltaje de alimentación mayor.

8.7 Interfaz con una Computadora Personal

Una PC proporciona 3 diferentes tipos de puertos, con los cuales se puede comunicar con un MCU: serie, paralelo y USB. Cabe aclarar que en los últimos años el uso de los puertos serie y paralelo es menos frecuente, incluyéndose únicamente en algunas computadoras de escritorio.

Esta comunicación, PC-MCU, es necesaria en diferentes aplicaciones. Por ejemplo, para un reloj checador electrónico, el sistema operaría en forma autónoma soportado por un MCU con memoria externa, no obstante, pasados varios días, requeriría la conexión con una PC para descargar la información almacenada y realizar los informes correspondientes.

Otro ejemplo es una marquesina para exhibir un mensaje en una matriz de LEDs, la cual puede ser manejada con uno o varios MCUs, funcionando de manera autónoma. Sin embargo, la conexión con una PC llega a ser necesaria cuando se requiere cambiar el mensaje.

En esta sección se muestran los requerimientos de hardware necesarios para la comunicación de los AVR con una PC y se dan algunas ideas para el software, pero sin revisar los elementos para la programación de la PC.

8.7.1 Puerto Serie

Los microcontroladores AVR incluyen una USART, por este medio pueden comunicarse con una PC, a través de su puerto serie, estableciendo una comunicación punto a punto. Debe considerarse que los AVR manejan niveles de voltaje TTL y una PC maneja niveles RS232. En la figura 8.32 se muestran los niveles de voltaje RS-232 y TTL.

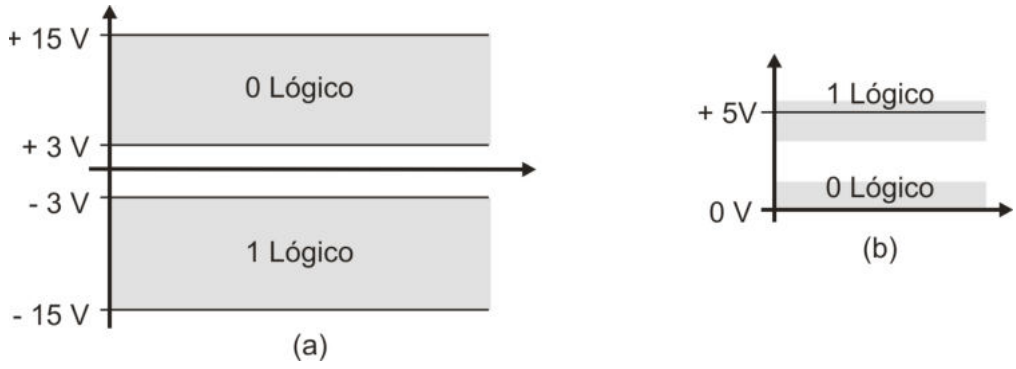


Figura 8.32 Comparación de (a) niveles de voltaje RS232 con (b) niveles de voltaje TTL

Los niveles RS-232 corresponden con rangos de voltaje para cada uno de los estados lógicos, en lugar de emplear un valor fijo, esto hace que la información continúe siendo válida aún después de atenuaciones debidas a la impedancia en la línea de transmisión. Con ello, una comunicación serial puede alcanzar distancias hasta de 15 metros, sin pérdida de información.

Para el acoplamiento de niveles de voltaje puede utilizarse un MAX232 o un MAX233, ambos circuitos de la firma MAXIM, básicamente son transmisores/receptores que convierten voltajes TTL/CMOS a RS-232 y viceversa, alimentándose con una fuente de 5 V. En la figura 8.33 se muestran las configuraciones de ambos CIs, el MAX232 requiere de capacitores externos para la generación del voltaje RS-232 y el MAX233 los tiene integrados. Las terminales mostradas corresponden con encapsulados del tipo DIP.

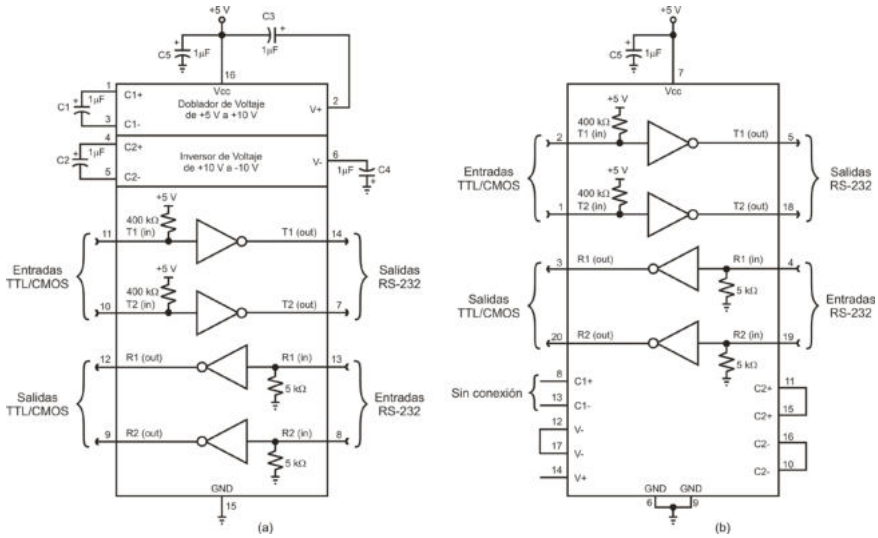


Figura 8.33 Circuitos para la conversión TTL/CMOS a RS-232: (a) MAX232 y (b) MAX233

El puerto serie en la PC se presenta a través de un conector DB9 (macho), éste se muestra en la figura 8.34 con los nombres de sus terminales. El puerto originalmente fue desarrollado para el manejo de un modem, es por eso que cuenta con terminales para un diálogo vía hardware (handshake). A pesar de ello, puede ser empleado como un puerto de propósito general para comunicarse con cualquier microcontrolador. Con un AVR es a través de su USART, utilizando únicamente a las terminales RXD (2), TXD (3) y GND (5). La configuración y uso de la USART se describió en la sección 6.1.

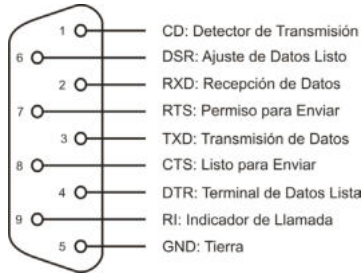


Figura 8.34 Conector DB9 de una PC para una comunicación serial

Dependiendo del software empleado en la PC, puede ser necesario anular el diálogo de hardware, conectando la terminal de Permiso para Enviar (RTS) con la de Listo para Enviar (CTS) y la Terminal de Datos Lista (DTR) con las terminales Detector de Transmisión (CD) y Ajuste de Datos Listo (DSR). En la figura 8.35 se muestra al conector DB9 con los puentes anteriormente descritos.

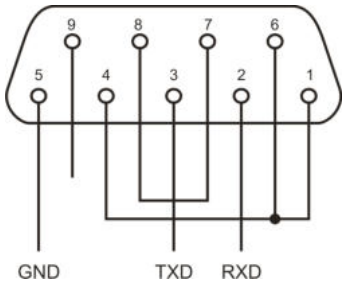


Figura 8.35 Puerto serie con puentes para anular las señales requeridas por un modem

Por lo tanto, para comunicar un AVR con una PC, a través de su puerto serie, básicamente se requiere de un conector DB9 hembra y un circuito MAX232 con 5 capacitores de 1μF o un circuito MAX233 con 1 capacitor de 1μF.

Para desarrollar el programa de la PC, debe utilizarse alguna biblioteca que incluya funciones para la configuración del puerto, el envío y la recepción de datos. Al puerto serie se le refiere como COM1, COM2, etc., dependiendo del número de puertos disponibles y del puerto al que se haga referencia.

8.7.2 Puerto Paralelo

Una PC puede incluir uno o más puertos con una interfaz paralela, los puertos paralelos originalmente fueron desarrollados para el manejo de impresoras, conocidos también como puertos Centronics, por una empresa desarrolladora de impresoras. No obstante, también pueden emplearse para comunicar una PC con otros sistemas electrónicos. Un puerto paralelo maneja señales TTL, esto significa que puede conectarse directamente con un MCU, pero es recomendable el uso de un buffer (circuito integrado 74LS244) para protección de la PC. El puerto paralelo en la PC se presenta a través de un conector DB25 (hembra), en la figura 8.36 se muestran sus terminales y en la tabla 8.13 la descripción de las mismas.

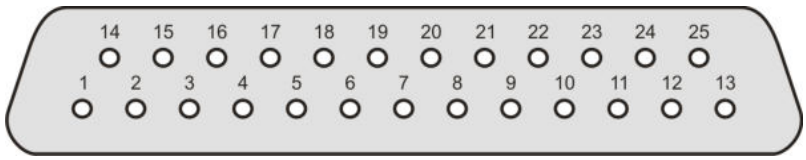


Figura 8.36 Puerto paralelo de una PC (también conocido como Centronics)

Tabla 8.13 Descripción de las terminales del puerto paralelo de una PC

Terminal	Nombre	Entrada/Salida	Descripción
1	STB	Salida	Estrobo, para informar que se han colocado datos paralelos.
2 – 9	D0 – D7	Salida	Líneas de datos (D0: terminal 2, D7: terminal 9).
10	ACK	Entrada	Línea de reconocimiento, activa cuando el sistema remoto toma datos.
11	Busy	Entrada	Si está activa, el sistema remoto no acepta datos.
12	PE	Entrada	Línea Falta de papel, se activa cuando falta papel en la impresora.
13	Slct In	Entrada	Se activa cuando se ha seleccionado a la impresora.
14	Auto FD	Salida	Se activa para que la impresora inserte una nueva línea por cada retorno de carro.
15	Error	Entrada	Indica que ocurrió un error en la impresora.
16	Init	Salida	Si se mantiene activa por al menos 50 µS, inicializa la impresora
17	Select	Salida	Obliga a la impresora a salir de línea.
18 – 25	GND	–	Tierra

En la tabla 8.13 se observa que las terminales tienen una función relacionada con el manejo de una impresora. Por esta funcionalidad, las señales de la interfaz Centronics se organizan en 3 puertos: uno de datos, uno de estado y uno de control. Cada puerto tiene una dirección relacionada con la dirección de la interfaz Centronics (LPTx). En la tabla 8.14 se muestran las direcciones de los diferentes puertos.

Tabla 8.14 Direcciones de los puertos debidos a una interfaz paralela

Impresora	Puerto de Datos	Puerto de Estado	Puerto de Control
LPT1	03BCh	03BDh	03BEh
LPT2	0378h	0379h	037Ah
LPT3	0278h	0279h	027Ah

Cuando se conecta un MCU u otro periférico, el programa de la PC debe utilizar las direcciones mostradas en la tabla 8.14 para establecer la comunicación. En la tabla 8.15 se muestra la ubicación de las terminales en los 3 puertos, debe notarse que no todos los puertos son de 8 bits y que algunos bits son activos en un nivel bajo de voltaje. En la figura 8.37 se distinguen los 3 puertos presentes en una interfaz paralela.

Tabla 8.15 Ubicación de terminales del puerto paralelo

Puerto	Bits								Ent/Sal
Datos	D7	D6	D5	D4	D3	D2	D1	D0	Salida
Estado	Busy’	ACK	PE	Slct In	Error	-	-	-	Entrada
Control	-	-	-	IRQEN	Select’	Init’	AutoFD	STB’	Salida

El bit 4 del registro de control (IRQEN) no corresponde con alguna terminal de la interfaz Centronics. Se trata de una bandera que habilita o prohíbe la generación de la interrupción IRQ7 cuando se activa la señal ACK.

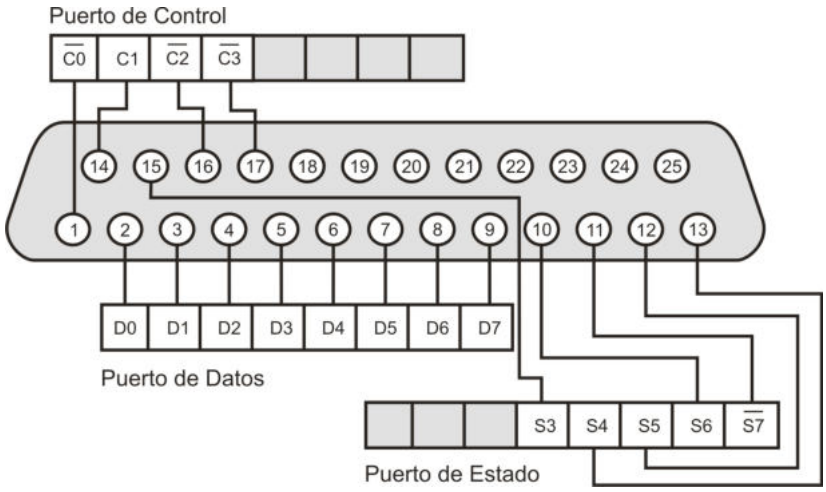


Figura 8.37 Puertos disponibles en una interfaz Centronics

Cabe aclarar que los nuevos sistemas operativos restringen el manejo del puerto paralelo, de manera que su acceso sólo es posible si se utiliza una librería de enlace dinámico (DLL), la cual debe incluir las funciones de entrada y salida de datos.

8.7.3 Puerto USB

El Bus Universal Serial (USB, *Universal Serial Bus*) es un puerto que sirve para conectar periféricos a una computadora. Actualmente la mayoría de periféricos se conectan por este puerto, evitando el uso del puerto serie o el paralelo, o incluso, la necesidad de alguna tarjeta de expansión manejada por los buses ISA (*Industry Standard Architecture*, Arquitectura Estándar Industrial) o PCI (*Peripheral Component Interconnect*, Interconexión de Componentes Periféricos) en una PC.

La velocidad en las transferencias vía USB es más alta que en los puertos serie y paralelo, pero más lenta que en los buses ISA o PCI. Sin embargo, se tiene la ventaja de que en un puerto USB es posible conectar o desconectar dispositivos sin la necesidad de reiniciar al sistema.

Cualquier dispositivo que se conecte vía USB requiere de un driver para ser reconocido por el procesador central. Algunos dispositivos requieren una potencia mínima, de manera que su alimentación puede tomarse del mismo puerto sin requerir alguna fuente de alimentación extra.

El USB casi ha reemplazado al puerto PS/2, para teclados y ratones, de manera que un amplio número de tarjetas madre modernas carecen del puerto PS/2. Las computadoras portátiles actuales sólo incluyen puertos USB como medio para la conexión de periféricos.

Por lo tanto, es importante considerar cómo conectar un ATMega8 o un ATMega16 a una computadora, a través del puerto USB. En esta sección se presentan diferentes opciones, debido a que los microcontroladores ATMega8 o ATMega16 no cuentan con los recursos necesarios para el manejo de la interfaz USB.

8.7.3.1 Adaptador de USB a RS-232

Una alternativa es el uso de un adaptador de USB a RS-232, la empresa Steren® manufactura un cable que se conecta a cualquiera de los puertos USB de una computadora. Una vez que se ha instalado el driver, proporcionado por la misma empresa, en la PC se presenta como un puerto COM virtual, de manera que es posible utilizar a la USART del MCU para establecer la comunicación con la PC. Puesto que con el driver se resuelve el reconocimiento del adaptador y la generación del puerto COM virtual, pareciera que el MCU se está conectando a un puerto serie real.

8.7.3.2 Circuitos Integrados Controladores

Otra alternativa consiste en emplear un circuito integrado controlador, que proporcione el mecanismo para la comunicación con una PC vía USB. Por ejemplo, la empresa FTDI (*Future Technology Devices International Ltd.*) ha manufacturado al FT232BM[8], el cual básicamente es un adaptador de USB a RS232. Las principales características de este circuito son:

- Compatible con USB 1.1 y USB 2.0
- Transferencias de datos desde 300 baudios hasta 3 Mbaudios a niveles TTL.
- Buffer receptor de 384 bytes y buffer transmisor de 128 bytes, para una productividad alta de datos.
- Interfaz UART que soporta datos de 7 u 8 bits, 1 o 2 bits de paro, paridad par/impar o sin paridad.

---

8 Para mas informacion, visite la pagina del fabricante: http://www.ftdichip.com/

- Alimentación única de 4.35 a 5.25 V.
- Regulador de 3.3 V integrado para la interfaz USB.
- Convertidor de nivel sobre la UART y señales de control, para conectarse con circuitos de 5 y 3.3 V.

Por lo tanto, en la tarjeta de un sistema basado en un microcontrolador se puede incluir al FT232BM con sus elementos de soporte. Para que el dispositivo sea reconocido por el sistema operativo de la computadora, se requiere la instalación de su driver. El fabricante proporciona 2 tipos de drivers, un driver VCP (*Virtual Com Port*) con el que se reconoce al FT232BM como un puerto COM virtual para comunicaciones seriales y un driver D2XX (*Direct Drivers*) el cual incluye una DLL para el desarrollo de una aplicación personalizada.

Existen otros circuitos controladores, por ejemplo el FT245BM proporciona una conversión de paralelo a USB y también es manufacturado por FTDI.

8.7.3.3 Módulos de Evaluación y Prototipado

Con base en el FT232BM, la empresa DLP Design ha manufacturado y comercializado al módulo DLP-USB232M-G. Con este módulo se tiene otra alternativa para comunicar una PC con un ATMega8 o un ATMega16 vía USB. En la figura 8.38 se puede ver que el módulo presenta una disposición tipo PDIP, facilitando el desarrollo de prototipos.

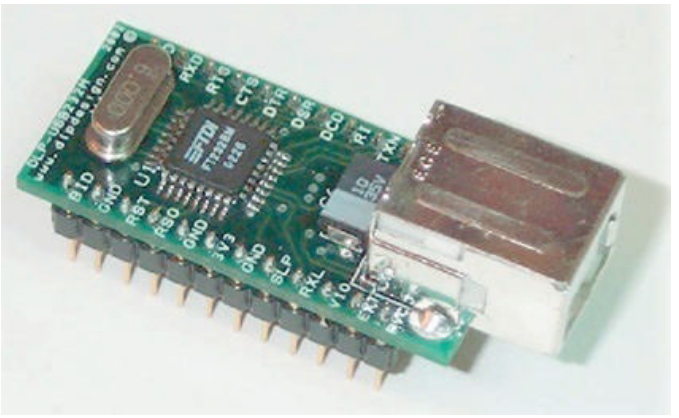


Figura 8.38 Módulo DLP-USB232M-G para una interfaz serial a USB

En la figura 8.39 se muestra como conectar al módulo DLP con un MCU, utilizando sólo una fuente de alimentación de 5 volts. Por lo tanto, al emplear este módulo, la comunicación USB se realiza por medio de la USART del MCU.

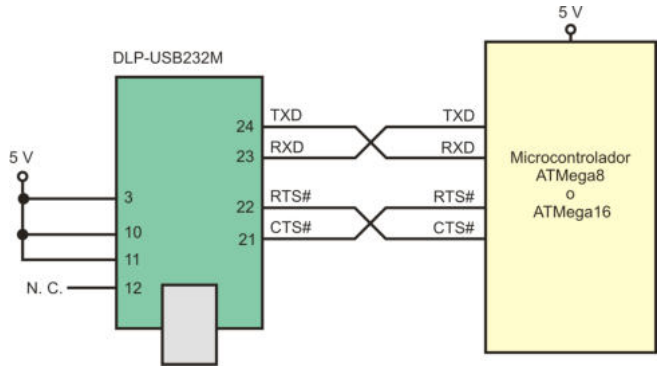


Figura 8.39 Conexión de un MCU con el módulo DLP-USB232M-G

Un módulo similar es el DLP-USB245M-G, también desarrollado por DLP Design, pero con base en el circuito FT245BM. Es un módulo que también presenta una disposición tipo PDIP, con una interfaz paralela para el microcontrolador. En la figura 8.40 se muestra al módulo y en la 8.41 un esquemático en el que se ilustra la forma de conectarlo con un microcontrolador.

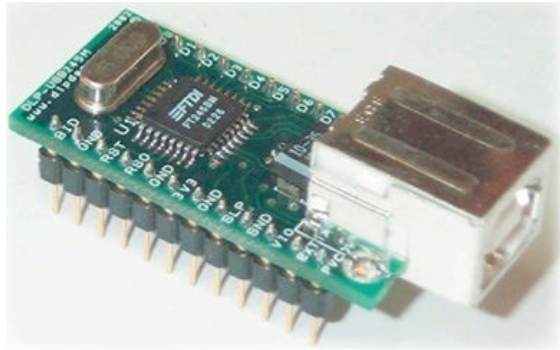


Figura 8.40 Módulo DLP-USB245M-G para una interfaz paralelo a USB

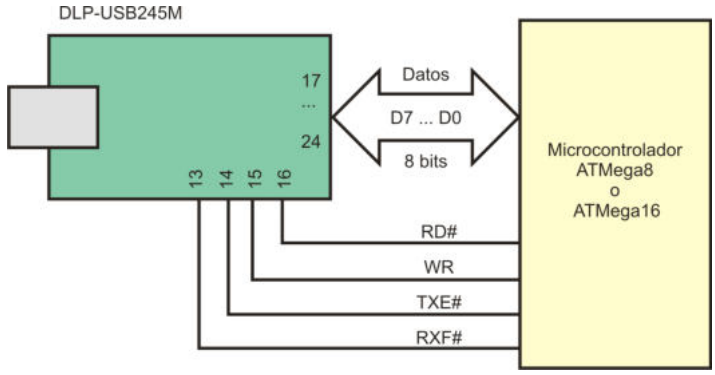


Figura 8.41 Conexión de un MCU con el módulo DLP-USB245M-G

Sin importar el módulo empleado para una comunicación USB con una PC, es necesario instalar su driver. Los módulos anteriormente descritos básicamente tienen por objetivo el acondicionamiento de los chips manufacturados por FTDI, por ello, los drivers a instalar son los mismos que los requeridos por los circuitos integrados controladores.

8.7.3.4 Uso de un AVR con Controlador USB Integrado

Uno de los criterios descritos en la sección 1.6 para seleccionar un MCU fue la compatibilidad entre dispositivos de una misma familia. Los miembros de una familia comparten el núcleo, por lo tanto manejan el mismo repertorio de instrucciones, aunque difieren en los recursos incluidos. Un aspecto importante, relacionado con esta compatibilidad, es que en la familia existan miembros con los recursos requeridos por diferentes aplicaciones.

Con respecto a la comunicación de un MCU con una computadora vía USB, dentro de la familia AVR se encuentran los dispositivos AT90USB82 y AT90USB162, los cuales, por su espacio en la memoria de código podrían verse como versiones expandidas del ATMega8 y ATMega16, con 8 y 16 Kbyte de memoria flash, respectivamente. En los que la empresa ATMEL ha incorporado un módulo controlador completamente compatible con USB 2.0, además de otras mejoras.

El controlador USB proporciona la interfaz para el flujo de datos almacenados en una memoria de doble puerto (DPRAM). Es una memoria independiente de 176 bytes, en donde se ubican los puntos terminales (*endpoints*) que sirven de base para las transferencias.

El controlador requiere una señal de reloj de 48 MHz ± 0.25 % para alcanzar una alta velocidad, la cual es generada con un lazo de seguimiento de fase (PLL, *Phase Locked Loop*) interno. El circuito PLL es manejado por una señal de reloj externa con una frecuencia menor, empleando un cristal o una señal de reloj externa suministrada en XTAL1.

El reloj de 48 MHz es utilizado para alcanzar transferencias a 12 Mbps, velocidad reflejada en los datos recibidos a través de la interfaz USB y empleada para las transmisiones. La recuperación del reloj en las líneas de los datos se realiza con un lazo de seguimiento de fase digital (DPLL, *Digital Phase Locked Loop*), el cual cumple con las especificaciones de la interfaz USB.

Para cumplir con las características eléctricas del puerto USB, el voltaje en las terminales D+ o D- debe estar en el rango de 3.0 a 3.6V. Dado que los microcontroladores AT90USB82 y AT90USB162 pueden alimentarse hasta con 5.5V, internamente incluyen un regulador para proporcionar el voltaje requerido por la interfaz USB. En la figura 8.42 se muestra la incorporación del controlador USB en el MCU, el regulador de voltaje y su vinculación con el núcleo AVR.

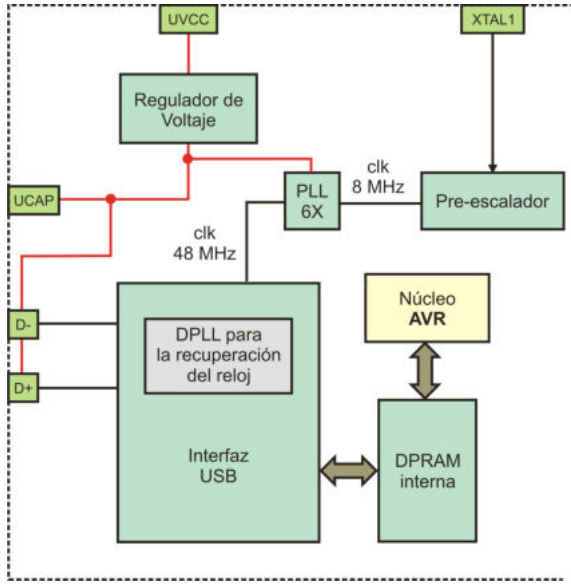


Figura 8.42 Microcontrolador AVR con controlador USB integrado

Sin embargo, la alternativa de emigrar a un AVR con un controlador USB integrado es más compleja que las otras opciones, dado que requiere más que un acondicionamiento del hardware externo, implica la modificación del software. Por ejemplo, las rutinas para el manejo de la USART deben remplazarse por el código para el manejo de la interfaz USB.

8.8 Ejercicios

Los ejercicios propuestos son útiles para experimentar con los dispositivos descritos en el presente capítulo. Si sus programas se organizan con rutinas o funciones, éstas pueden utilizarse nuevamente en aplicaciones reales.

1. Conecte un teclado matricial de 4x4 a un ATMega8 y muestre el valor de la tecla presionada en un display de 7 segmentos. En la figura 8.43 se muestra el hardware requerido.

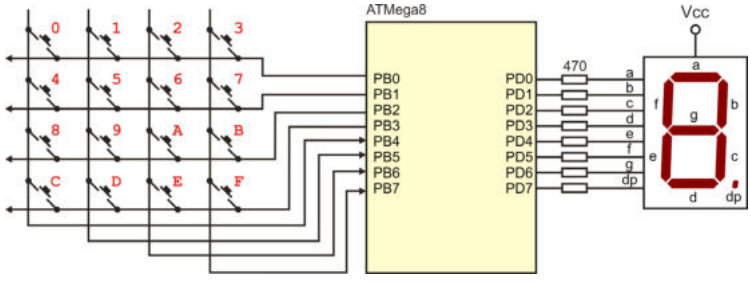


Figura 8.43 Circuito para evaluar un teclado matricial

2. Conecte 3 displays de 7 segmentos en un bus común e implemente un contador de eventos ascendente/descendente. En la figura 8.44 se muestra el hardware requerido.

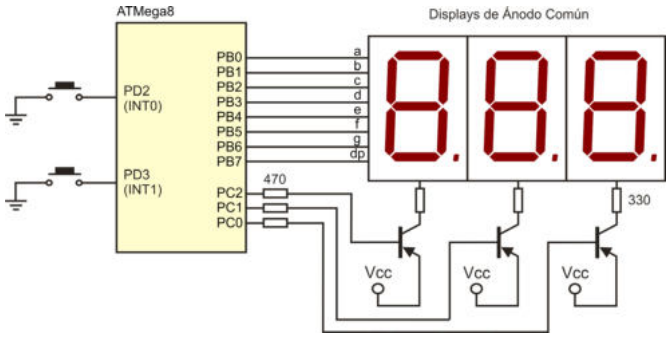


Figura 8.44 Circuito para evaluar el manejo de 3 displays en un bus común

3. Conecte un LCD con un ATMega8 empleando una interfaz de 4 bits, como se mostró en la figura 8.14 (a), e implemente las siguientes funciones:

- *LCD_pulso_E ()*: Para generar un pulso en la terminal E del LCD.
- *LCD_write_inst4(char inst)*: Escribe el nibble menos significativo del carácter recibido.
- *LCD_write_inst8(char inst_8b)*: Escribe una instrucción de 8 bits, iniciando con el nibble más significativo. Utilice la función anterior y considere un tiempo de espera de 40 µS (tiempo requerido por la mayoría de comandos).
- *LCD_write_data(char dat_8b)*: Escribe un dato de 8 bits, iniciando con el nibble más significativo. Considere un tiempo de espera 40 µS, tiempo requerido para la escritura de un dato.
- *LCD_reset( )*: Para realizar la secuencia mostrada en la figura 8.18, necesaria para inicializar al LCD con una interfaz de 4 bits.
- *LCD_clear( )*: Para limpiar al LCD no puede utilizarse la función *LCD_write_inst8* porque requiere un tiempo de ejecución mayor.
- *LCD_cursor(char pos)*: Ubica al cursor, debe considerarse si en el primero o en el segundo renglón.
- *LCD_write_cad(char cad[], unsigned char tam)*: Escribe una cadena, debe conocerse su tamaño. Utilice la función *LCD_write_data.*

Con las funciones desarrolladas, realice una aplicación que muestre una cadena constante en un LCD.

4. Conecte un motor paso a paso unipolar con un AVR y, mediante un par de botones, realice pasos en una u otra dirección, dando un paso cada vez que algún botón es presionado.

5. Implemente el termómetro digital descrito en el ejemplo 5.2, utilice las funciones realizadas en el ejercicio 3, considerando el acondicionamiento necesario para el sensor (Figura 8.31).

6. Tomando como base al problema 3, agregue un MAX232 o un MAX233 y modifique el programa de manera que el mensaje a mostrar provenga de una PC. Con la finalidad de probar la comunicación en ambos sentidos, también agregue un botón. Cuando el botón sea presionado, el sistema debe regresar el mensaje que ha recibido. En la figura 8.45 se muestra el hardware requerido. En la PC es posible emplear la terminal de Windows o algún programa comercial como el Serial Port Monitor.

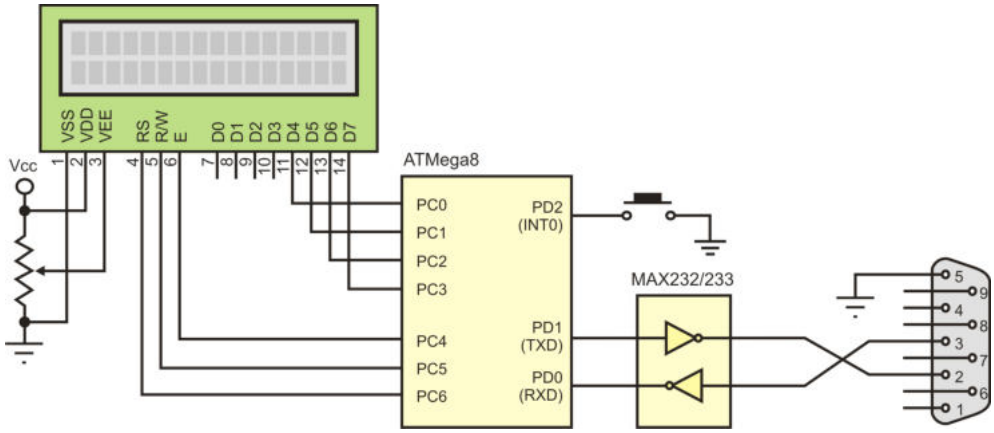


Figura 8.45 Circuito para evaluar la interfaz serial con una PC

7. Pruebe alguna de las alternativas propuestas para que el problema 6 sea resuelto mediante na comunicación USB.

9. Desarrollo de Sistemas

En este capítulo se muestra una metodología simplificada para desarrollar sistemas basados en microcontroladores y se ilustra su uso con 2 ejemplos de aplicación. La metodología está enfocada a aplicaciones soportadas por un solo microcontrolador y ha sido aplicada para el desarrollo de proyectos finales en cursos de microcontroladores.

Esta metodología difícilmente podría aplicarse directamente en sistemas de complejidad alta, cuya implementación requiera más de un MCU de 8 bits como parte de sus elementos de procesamiento. Una alternativa consistiría en tratar al sistema complejo como dos o más sistemas simples interactuando durante su ejecución, en donde cada sistema simple estaría basado en un microcontrolador AVR. De esta forma, la metodología podría ser empleada en cada subsistema o etapa del sistema complejo.

9.1 Metodología de Desarrollo

La metodología comprende 8 pasos, los cuales se describen a continuación:

1. **Planteamiento del problema:** Este aspecto es fundamental, no se puede iniciar con el desarrollo de un sistema mientras no se comprenda el comportamiento esperado. Consiste en una descripción detallada de las especificaciones, puede partirse de un dibujo ilustrando cómo va a ser el sistema cuando se haya concluido, mostrando sus entradas y salidas. También, se debe establecer el estado inicial de las salidas y entender cómo los cambios en las entradas afectan a las salidas, para ello, pueden realizarse descripciones textuales o diagramas de flujo simplificados. O bien, con diferentes diagramas se deben ilustrar las respuestas esperadas en el sistema ante las diferentes entradas.

   En el planteamiento del problema los estudiantes o desarrolladores deben interesarse en qué van a hacer y en cómo va a operar el sistema una vez que esté terminado. Es decir, proyectar una visión del resultado esperado, listando todas las tareas que va a realizar el sistema. En este momento aún no se le debe dar importancia a cómo se va a desarrollar el sistema.

2. **Requerimientos de hardware y software:** Una vez definidas las especificaciones del sistema, deben detectarse los requerimientos de hardware y software.

   La funcionalidad en el sistema y el número de entradas y salidas determinan qué microcontrolador debe usarse, si es suficiente con un ATMega8, si se requiere un ATMega16 o si es más conveniente utilizar otro miembro de la familia AVR.

En este punto se revisan las tareas que va a realizar el sistema, se hace una lista del hardware requerido y de los módulos o funciones de software que deben desarrollarse, o bien, se observa si existe alguna biblioteca con funciones que se puden reutilizar. Los requerimientos de hardware y software se complementan, emplear más hardware normalmente implica menos software o viceversa. Por ejemplo, si un sistema requiere el uso de un teclado matricial, se tiene la alternativa de emplear un decodificador de teclado externo o desarrollar la rutina para el manejo del teclado.

3. **Diseño del hardware:** Se debe realizar el diagrama electrónico del sistema, ya sea en papel o con el apoyo de alguna herramienta de software, en este punto se destinan los puertos del MCU para entradas o salidas y se define cómo conectar los diferentes elementos de hardware. Es importante conocer la organización del microcontrolador a utilizar, algunos recursos emplean terminales específicas y éstas no pueden ser comprometidas con otras tareas. Por ejemplo, si un sistema se basa en un ATMega8 y produce señales PWM para el manejo de un motor, no se puede disponer por completo del puerto B, esto significa que recursos como un teclado matricial de 4 x 4 o un LCD, deben conectarse en otro puerto porque van a utilizar todas sus terminales.

4. **Diseño del software:** En este paso se debe describir el comportamiento del sistema, mediante algoritmos o diagramas de flujo. Para el programa principal es conveniente un diagrama de flujo en el que se especifiquen las configuraciones de los recursos empleados y se realice el llamado a las funciones necesarias para resolver el problema. El programa principal de un sistema basado en un MCU generalmente entra en un lazo infinito, el cual no va a abandonar mientras el sistema siga energizado.

   Para las funciones y las rutinas de atención a interrupciones (ISRs) puede realizarse una descripción textual, una descripción algorítmica o un diagrama de flujo, dependiendo de la complejidad de la tarea a realizar. Las funciones y las ISRs si tienen un final bien definido.

   El diseño del software puede hacerse en forma paralela al diseño del hardware, dado que en el análisis de los requerimientos de hardware y software ya se tomaron las decisiones sobre qué actividades se van a realizar por hardware y cuáles por software.

   El diseño del software corresponde con una descripción estructurada del comportamiento global del sistema, sin considerar las conexiones de los periféricos con el MCU.

5. **Implementación del hardware:** Consiste en la realización física del hardware, en esta etapa debería hacerse en Protoboard, no es recomendable el desarrollo del circuito impreso hasta que se haya garantizado la funcionalidad del sistema bajo desarrollo.

   Concluida la implementación del hardware, debe revisarse que está correctamente conectado, realizando pruebas simples como la existencia de continuidad en las conexiones. No obstante, cuando la aplicación requiere de un número considerable de dispositivos externos, como un teclado matricial, un LCD, comunicación serial u otros recursos, es recomendable probar la integridad del hardware por medio de funciones o rutinas simples. Si se emplean elementos analógicos, también es necesario observar el buen desempeño de las etapas de acondicionamiento de señal.

   En ocasiones ocurre que se descarga la aplicación en el microcontrolador y el sistema no funciona correctamente, sin una comprobación previa del hardware es difícil determinar si el error está en el hardware o en el software. Al garantizar la integridad del hardware, se minimiza la posibilidad de errores en el resultado final.

6. **Implementación del software:** Consiste en codificar en lenguaje C o en ensamblador los algoritmos o diagramas de flujo desarrollados en la etapa de diseño del software. Para esta etapa ya se debe tener cubierto el diseño de hardware, es decir, deben conocerse todas las conexiones de los puertos del MCU con los periféricos externos. La implementación del software puede hacerse a la par con la implementación del hardware.

   Es conveniente complementar la codificación con simulaciones, para garantizar que el software cumple con la tarea planteada. Resultaría muy ilustrativo si se contase con alguna herramienta que permita una simulación visual.

7. **Integración y evaluación:** Consiste en la descarga del programa compilado en el MCU y la puesta en marcha del sistema, para evaluar su funcionamiento. Se tiene una garantía de éxito cercana a un 100 % si el hardware fue revisado y se comprobó su funcionamiento, y si el software fue simulado para garantizar que realizaba las tareas planeadas.

   El sistema debe exponerse ante las diferentes situaciones, modificando las entradas y evaluando las salidas, su comportamiento debe compararse con las especificaciones iniciales, realizadas durante el planteamiento del problema.

8. **Ajustes y correcciones:** Siempre existe la posibilidad de que el sistema requiera de algunos ajustes, las razones son diversas, por ejemplo, los rebotes que presentan los botones, el tiempo del barrido en un arreglo de displays, el tiempo de respuesta de algún actuador, etc. En todos esos casos, se debe detectar en donde puede hacerse un ajuste al sistema, modificar al programa y probar la nueva versión. Resulta bastante útil contar con un programador por SPI o JTAG, dado que los dispositivos permiten una programación en el sistema implementado (*in-system*), lo cual facilita los ajustes necesarios, sin tener que retirar al MCU del sistema, hasta alcanzar su funcionamiento adecuado.

   Los ajustes pueden deberse a que no se hizo una revisión completa de los requerimientos de hardware y software, por ejemplo, si se realizó una chapa electrónica y no se consideró el respaldo en memoria no volátil de una nueva clave, debería replantearse el software, agregando esta nueva característica al sistema. Para el mismo sistema, si al activar la salida a un electroimán se reinicia al microcontrolador por no tener un suministro suficiente de corriente, es necesario modificar al hardware, agregando un amplificador de corriente con un transistor o un par Darlington, o incorporar un opto-acoplador para aislar la etapa lógica de la etapa de potencia.

   El sistema debe mejorar después de realizar ajustes en el hardware o en el software. Como parte de los ajustes de hardware, puede considerarse el desarrollo del circuito impreso o la adecuación en un gabinete o chasis.

   Sin embargo, también puede ocurrir que una vez implementado el sistema se detecte que no satisface las necesidades del problema, esto porque durante su planteamiento no se consideraron todas las situaciones a las cuales el sistema final sería sometido, o bien, porque surgieron nuevas necesidades durante el proceso de desarrollo. Esto implica una revisión del planteamiento del problema. Al realizar estas correcciones, prácticamente se va a desarrollar una nueva versión del sistema.

La existencia de etapas que pueden realizarse en forma concurrente y la posibilidad de que existan ajustes o correcciones, hacen que la metodología no siga un flujo secuencial, esto se muestra en la figura 9.1. En el diagrama se ha separado la etapa de ajustes y correcciones, y se han puesto como condicionantes para las líneas de retroceso, después de la integración y evaluación se observa si se requieren ajustes o correcciones, en caso de no ser así, el sistema está listo para ser puesto en marcha.

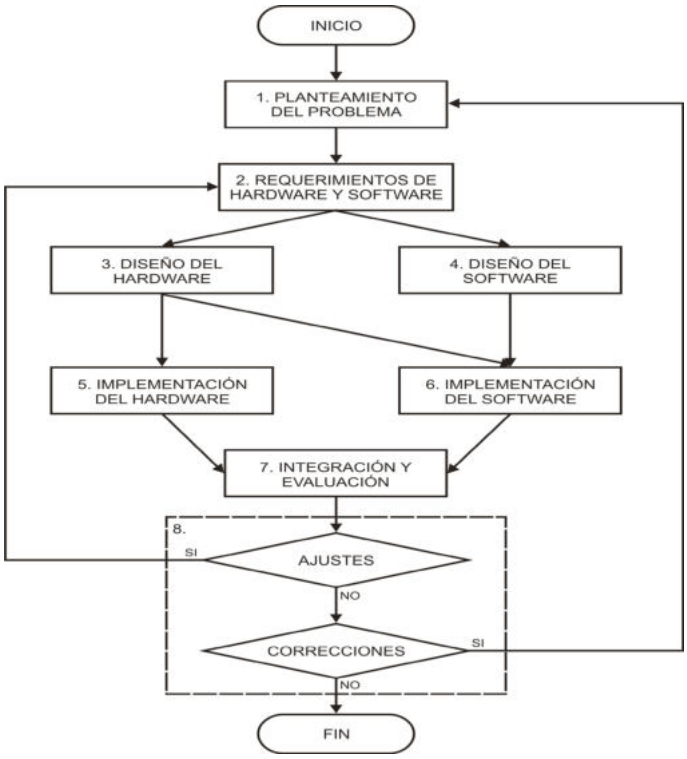


Figura 9.1 Metodología para el diseño de sistemas basados en microcontroladores

9.2 Ejemplos de Diseño

En esta sección se ilustran 2 ejemplos en los que se ha aplicado la metodología descrita en la sección anterior. En la figura 9.1 se observa como algunas etapas pueden realizarse en forma concurrente, esto es posible cuando el desarrollo de un sistema es realizado por un grupo de personas. No obstante, la descripción de los ejemplos sigue un flujo secuencial.

9.2.1 Reloj de Tiempo Real con Alarma

Un reloj es un sistema ampliamente usado, conocer la hora es una tarea común. Aunque existe una gama amplia de relojes comerciales, desarrollar un reloj con un MCU aún resulta interesante, porque el mismo diseño puede utilizarse para manejar displays grandes, muy útil para espacios públicos. Además, la alarma puede remplazarse con una salida de AC, con un relevador o un triac, para manejar cualquier aparato electrónico, como una lámpara, un radio, un televisor, etc. O bien, el reloj puede ser la base para un sistema que requiere temporización, como el control de un horno de microondas, de una lavadora, etc.

9.2.1.1 Planteamiento del Problema

El reloj debe mostrar su salida en 4 displays de 7 segmentos, el tercer display debe conectarse en forma invertida para proporcionar los 2 puntos intermedios. La configuración de la hora actual y de la hora para la alarma se va a realizar por medio de 3 botones. Un interruptor debe habilitar la alarma, la cual debe tener un zumbador en su salida. En la figura 9.2 se muestran las características esperadas en el sistema, ilustrando sus entradas y salidas.

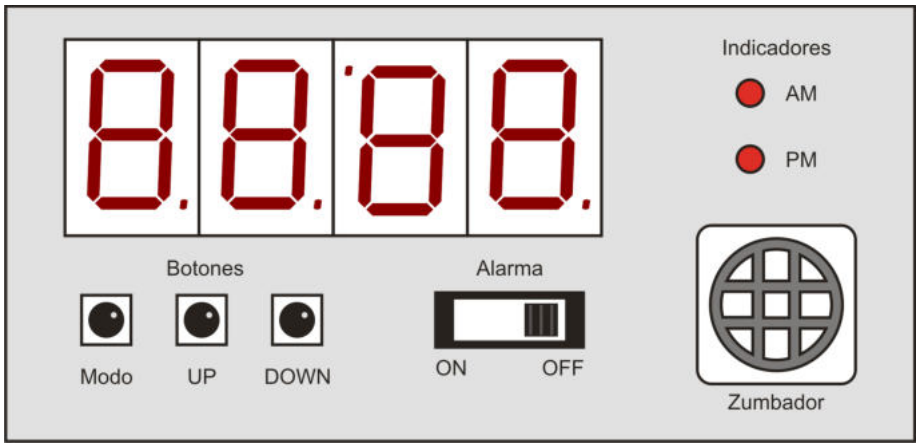


Figura 9.2 Reloj de tiempo real con alarma

Se espera que el sistema continuamente muestre la hora actual con los 2 puntos parpadeando cada medio segundo y además, dé atención al botón de Modo. La hora actual se va a mantener con el apoyo de uno de los temporizadores del MCU. Cuando la hora actual coincida con la hora de la alarma, se debe activar al zumbador durante un minuto o hasta que se conmute al habilitador de la alarma. Por lo tanto, en el lazo infinito del programa también se verifica si es necesario apagar la salida del zumbador.

El botón de Modo es para dar paso a los diferentes estados del sistema, dependiendo de cuantas veces ha sido presionado. Los modos para la operación del reloj deben ser:

1. Modo normal, en este modo, el sistema muestra la hora actual con los puntos intermedios parpadeando, es el modo inicial. Sólo se va a atender al botón Modo.

2. El reloj muestra los segundos, manteniendo aún a los botones UP y DOWN inhabilitados

3. Se muestra la hora actual, pero con la hora parpadeando y se habilitan los botones UP y DOWN para permitir su modificación.

4. También se muestra la hora actual, pero ahora con los minutos parpadeando y pudiendo modificarse con los botones UP y DOWN.

5. Se muestra la hora de la alarma, con la hora parpadeando para que pueda modificarse con los botones UP y DOWN.

6. También se muestra la hora de la alarma, pero ahora con los minutos parpadeando para que puedan modificarse con los botones UP y DOWN.

El sistema debe iniciar en el modo normal, cada vez que se presione al botón Modo, el reloj debe pasar por cada uno de los modos listados anteriormente, por lo que después de presionarlo 6 veces, nuevamente va a regresar al modo normal.

Los botones UP y DOWN sólo van a estar habilitados en los modos 3, 4, 5 o 6 (modos de configuración).

Entradas del sistema:

- Tres botones: El botón de Modo para cambiar entre los modos de operación, y los botones UP y DOWN, para configurar la hora o alarma. Estos últimos sólo son atendidos si se ha dado paso a algún modo de configuración (modos 3, 4, 5 y 6).

- Un interruptor, para habilitar la alarma. El zumbador va a activarse cuando coincida la hora actual con la hora de la alarma, siempre que haya sido habilitada.

Salidas del sistema:

- Cuatro displays de 7 segmentos, para mostrar la hora actual o la hora de la alarma. La información bajo exhibición depende del modo de operación.

- Dos indicadores basados en LEDs, la hora debe manejarse en un formato de 12 horas, por lo tanto, en estos LEDs se indica si corresponde a antes del medio día (AM) o si es posterior al medio día (PM).

- Un zumbador, para generar la alarma. Se activa cuando coincide la hora actual con la hora de la alarma, sólo si la alarma está habilitada.

Estado inicial:

El estado del sistema al ser energizado debe ser:

- **Datos internos:** Para la hora actual y la hora de la alarma se requieren variables internas. Cuando el sistema se energice, ambas horas deben tener las 12:00:00. Son necesarias 3 banderas, una para indicar que la hora inicial corresponde a antes del medio día (AM_F = 1), otra similar para la hora de la alarma (AM_F_A = 1) y otra para indicar que no hay alarma activada (ALARM = 0). También se requiere una variable para indicar el modo de operación (Modo = 1).

- **Entradas:** El sistema sondea únicamente al botón de modo, con el que se da paso a los diferentes modos de operación. Los cambios en los botones UP y DOWN son ignoradas.

- **Salidas:** El indicador de AM inicia encendido, el zumbador apagado y en los displays se muestra la hora inicial, aunque esto último se hace continuamente, por lo que es parte del lazo infinito.

9.2.1.2 Requerimientos de Hardware y Software

Lo primero es la selección del MCU, para ello debe considerarse el número de entradas y salidas requeridas. Considerando minimizar costos, la decodificación para los displays de 7 segmentos se realiza con una rutina de software (no se utiliza un CI decodificador de BCD a 7 segmentos), entonces, el número de terminales requeridas son:

Entradas: 3 para los botones y 1 para el interruptor, en total son 4 entradas.

Salidas: 7 para el bus de datos de los displays, 4 para la habilitación de cada display, 1 para los 2 puntos, 2 para los LEDs de estado y 1 para el zumbador, esto da un total de 15 salidas.

En total son 19 terminales y como la funcionalidad de un reloj no es tan compleja, con un ATMega8 es suficiente puesto que tiene 23 I/O distribuidas en 3 puertos.

Por el tipo de sistema, es conveniente utilizar un oscilador externo que maneje al temporizador 2, para tener una base de tiempo exacta y simplificar al software, esto implica el uso de 2 terminales más, específicamente PB6 y PB7 para la conexión de un cristal externo.

Además del ATMega8, los otros elementos de hardware requeridos son:

- 3 botones (Modo, UP y DOWN).
- 1 interruptor, para habilitar la alarma.
- 4 displays de ánodo común (bus de datos activo en bajo), con transistores BC558 (PNP) y resistores de 1 Kohm para acondicionar su manejo desde el MCU.
- 2 LEDs en color Rojo, con resistores de 330 ohms para limitar la corriente.
- 1 zumbador de 5 V con un transistor BC547 y un resistor de 330 ohms.
- Un cristal de 32.768 KHz, para la base de tiempo, empleando al temporizador 2.

En cuanto al software, de las funciones citadas a lo largo del texto sólo se reutiliza la función para el manejo de los displays, descrita en el ejemplo 8.2. El programa principal se centra en la exhibición de la hora actual, el sondeo del botón de Modo y la verificación de la desactivación de la alarma. Con el botón de Modo se da paso a los diferentes estados del sistema. Los botones UP y DOWN son para la configuración de la hora y son manejados por interrupciones externas, aunque inicialmente están desactivadas. Se activan después de que el botón de Modo ha sido presionado 2 veces y nuevamente se desactivan cuando se ha presionado 6 veces.

La variable Modo y las banderas determinan el estado del sistema y por lo tanto, el flujo del programa. También definen que se va a mostrar en los displays o que variables van a modificar las interrupciones externas.

El temporizador 2 se configura para manejar la base de tiempo, como es manejado por un cristal externo de 32.768 KHz, se configura para que cada medio segundo genere una interrupción, en la ISR correspondiente se actualiza la hora y se observa la posibilidad de una alarma. La base de tiempo es de medio segundo para que también sirva de base en el parpadeo.

9.2.1.3 Diseño del Hardware

Antes de ubicar los periféricos externos, deben observarse qué recursos internos se requieren, para respetar sus correspondientes terminales, los botones UP y DOWN son manejados por interrupciones, por lo tanto, utilizan las terminales PD2 y PD3. El temporizador 2 es manejado por un cristal externo, el cual es conectado en las terminales PB6 y PB7.

No habiendo otras terminales comprometidas, los periféricos externos se ubican en las terminales restantes. El hardware resultante se muestra en la figura 9.3, se debe inhabilitar el *reset* en el pin PC6 (con los *Bits de Configuración y Seguridad*), para que pueda utilizarse como salida. Pero debe considerarse que con esto se inhabilita la programación por SPI, éste debe ser el último paso si se cuenta con un programador que utilice esta interfaz.

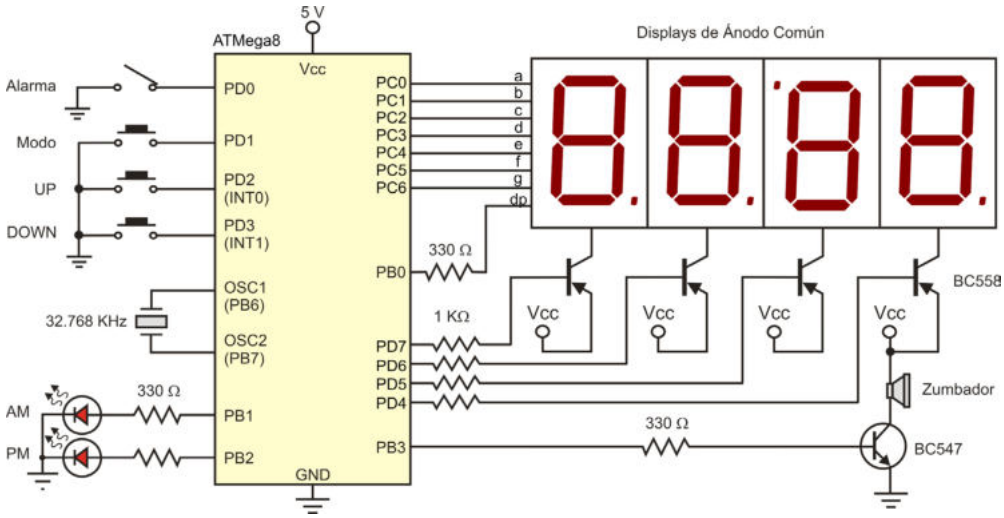


Figura 9.3 Diseño del hardware del reloj de tiempo real con alarma

9.2.1.4 Diseño del Software

El diseño del software inicia con el desarrollo de un diagrama de flujo en el que se muestre el comportamiento global del sistema, éste se muestra en la figura 9.4 y corresponde con el programa principal.

Todas las configuraciones iniciales son proposiciones secuenciales. Las entradas y salidas se definen de acuerdo con el diseño del hardware, habilitando los resistores de *pull-up* en las entradas. El temporizador 2 se configura y activa para que interrumpa cada medio segundo, a partir del cristal externo. Las interrupciones externas se configuran por flanco de bajada, pero no se habilitan.

La inicialización de variables, banderas y salidas también se realiza mediante proposiciones secuenciales. Hora_Actual realmente se maneja con 3 variables: h_act, m_act y s_act. Hora_Alarma con 2: h_alrm y m_alrm. La variable Modo indica el modo de operación inicial. AM_F, AM_F_A y ALARM son banderas, AM_F y AM_F_A indican que la hora actual y la de la alarma corresponden a antes del medio día, respectivamente. ALARM indica que no está activo el zumbador. El valor de cada salida corresponde con el estado inicial del sistema.

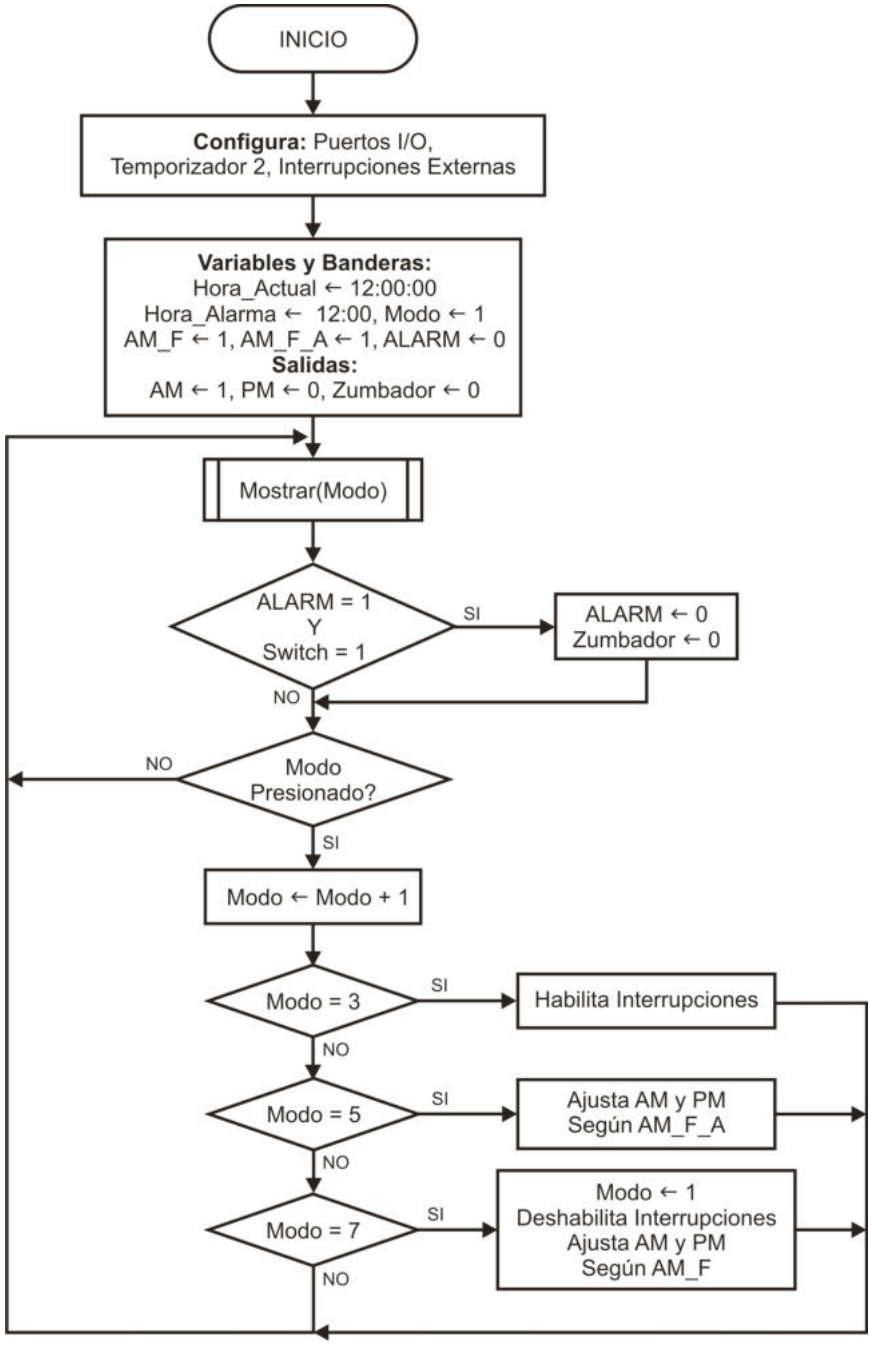


Figura 9.4 Diseño del software para el reloj de tiempo real con alarma

El programa se concentra en el despliegue de información, con el apoyo de la función Mostrar, además de sondear al botón de Modo y evaluar si se debe desactivar la alarma. Para el botón de Modo debe considerarse un retardo para evitar rebotes (por emplear sondeo). Al alcanzar los modos 1, 3 y 5 se deben realizar algunos ajustes. El modo 7 no existe, por lo que el sistema debe regresar al modo 1. El modo determina la información que el sistema está mostrando.

Se manejan 3 rutinas para dar servicio a las interrupciones externas y a la del temporizador 2, el comportamiento de las ISRs se revisa por separado. En el programa principal, la bandera ALARM sólo es evaluada, ésta es modificada en las ISRs. AM_F y AM_F_A son modificadas y evaluadas en las ISRs.

La función Mostrar es similar a la del ejercicio 8.2, sólo que ahora no recibe el arreglo con los datos, sino que los determina a partir de la variable Modo. La función Mostrar debe iniciar llamando a una función para que ubique cada dato en el arreglo. En la figura 9.5 se muestra el comportamiento de la función para la ubicación de los datos a mostrar.

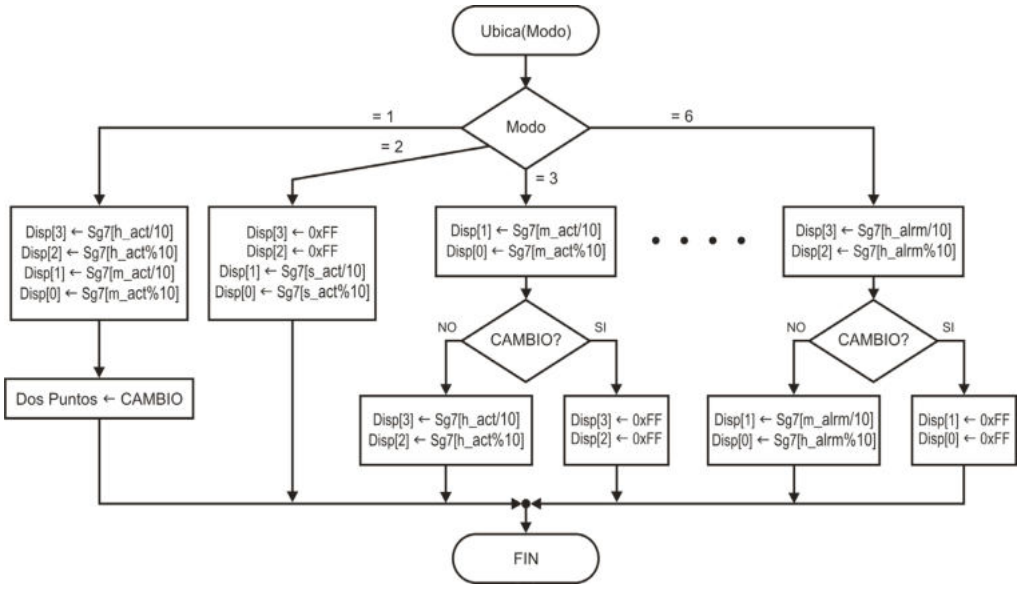


Figura 9.5 Ubica los datos para que se puedan mostrar en los displays

La bandera CAMBIO conmuta con cada interrupción del temporizador 2 (medio segundo), en el modo 1 determina si se encienden los puntos intermedios, en los modos 3, 4, 5 y 6 determina si se coloca información en los displays o si se dejan apagados, para producir el parpadeo.

En la figura 9.6 se muestra el comportamiento de la ISR debida al temporizador 2, la cual se encarga de actualizar las variables relacionadas con la hora actual, además de revisar si se debe activar la alarma, el interruptor de la alarma introduce un 0 lógico cuando se cierra.

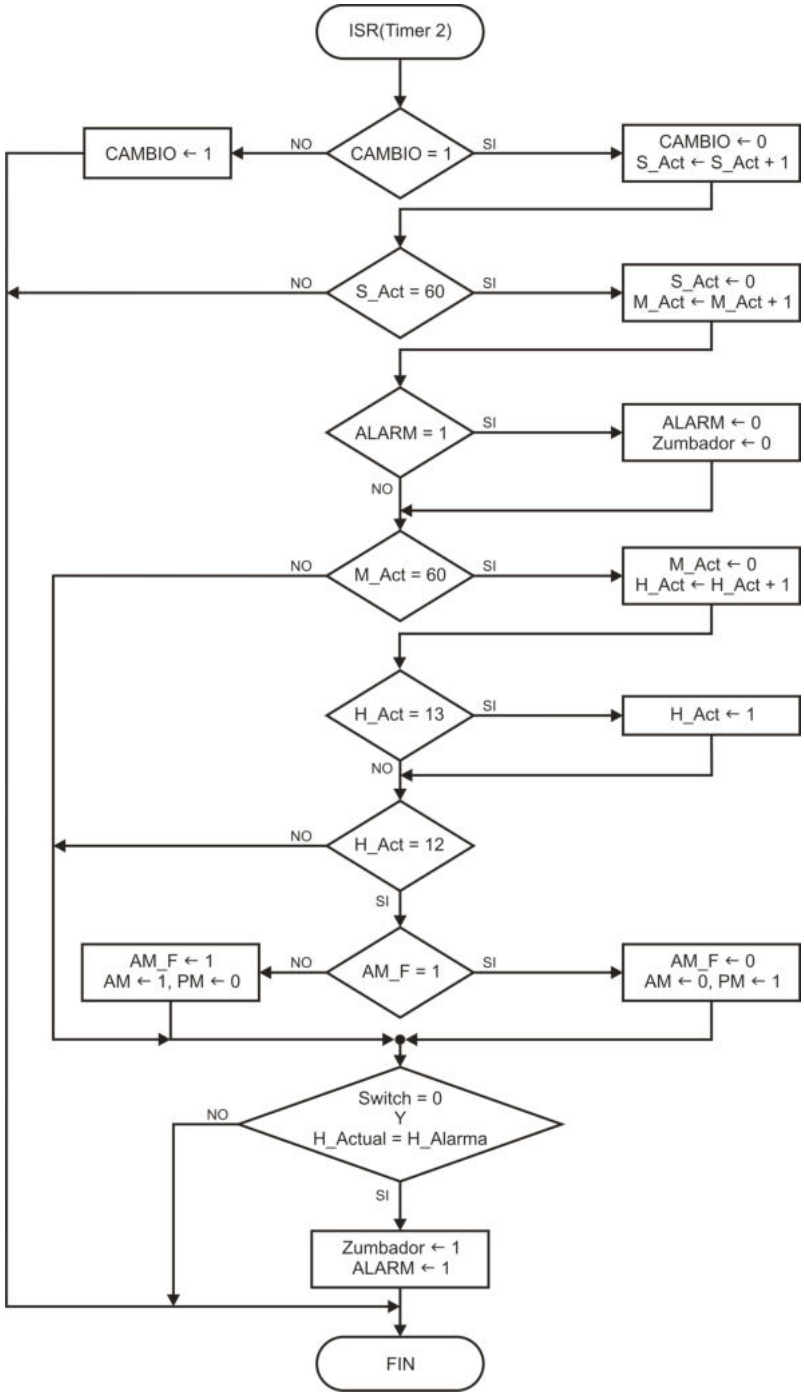


Figura 9.6 ISR del temporizador 2, se ejecuta cada medio segundo

La configuración de la hora actual y de la hora de la alarma se realiza con las interrupciones externas, con INTO se realizan los incrementos y con INT1 los decrementos, la variable a modificar con cada interrupción depende del modo de operación actual. En la figura 9.7 se muestra el comportamiento de la ISR de la interrupción externa 0.

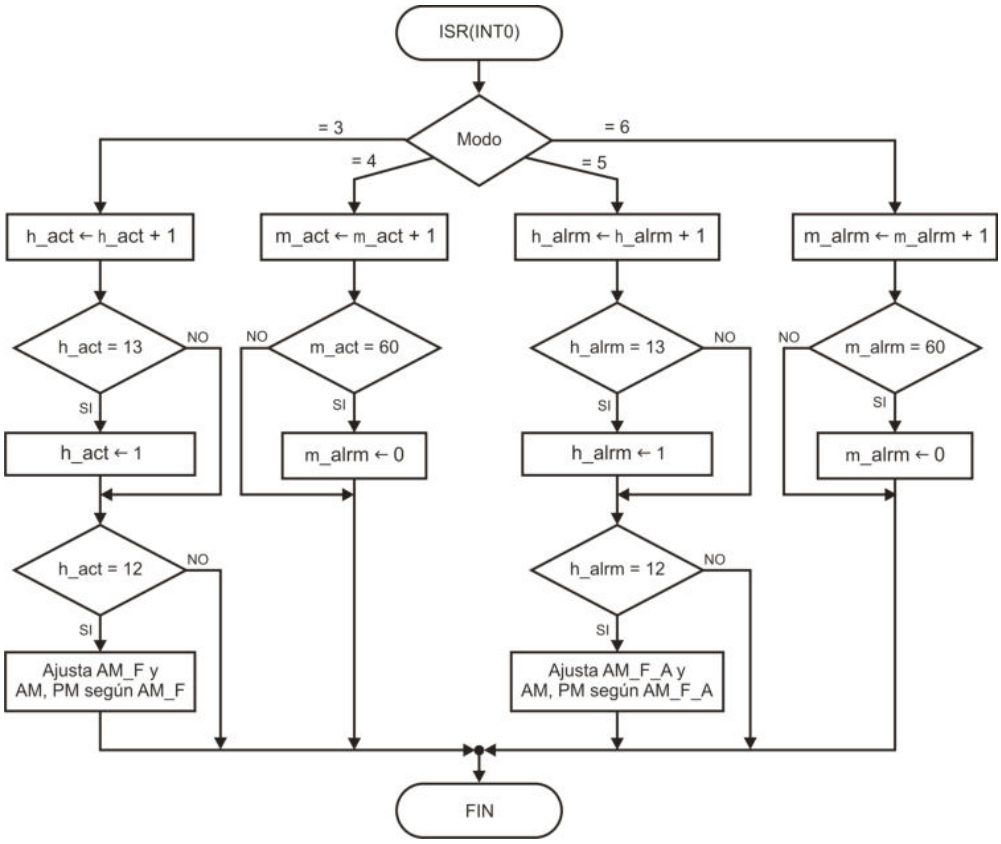


Figura 9.7 ISR de la interrupción externa 0, se ejecuta cada que el botón UP es presionado

La rutina de la interrupción externa 1 tiene un comportamiento similar, sólo que en lugar de realizar incrementos se realizan decrementos, revisando los límites para mostrar horas válidas y haciendo los ajustes de las banderas AM_F y AM_F_A, así como de las salidas AM y PM, cuando sean necesarios.

Aunque los diagramas de flujo muestran el comportamiento del sistema, no consideran las conexiones en el hardware, por ello, al desarrollar el código también debe considerarse el diseño del hardware. El diseño del software concluye cuando se tienen los elementos necesarios para su codificación.

9.2.1.5 Implementación del Hardware

La implementación del hardware básicamente consiste en el armado físico del circuito de la figura 9.3. En la figura 9.8 se muestran los resultados de la implementación en protoboard. Para garantizar que las conexiones se realizaron adecuadamente, se descargó un programa con una función que realizó el barrido en los 4 displays, mostrando datos constantes.

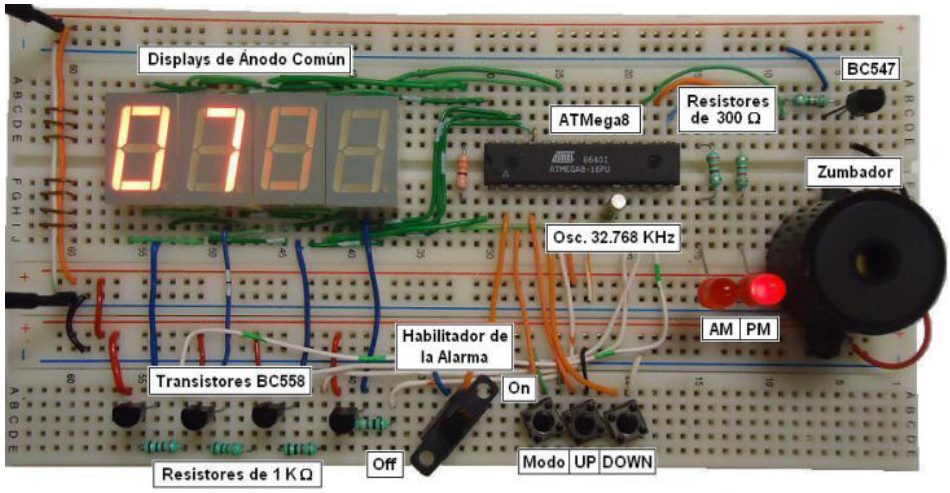


Figura 9.8 Implementación del hardware correspondiente al reloj con alarma

Con un barrido continuo en los displays, el ojo humano percibe que los 4 están encendidos al mismo tiempo, sin embargo, una fotografía captura sólo un instante de tiempo, por ello, este efecto no se alcanza a percibir en la figura 9.8, mostrando un display con una mayor intensidad y otro apagado.

9.2.1.6 Implementación del Software

El programa se implementó en Lenguaje C, en este apartado se muestran algunas de sus funciones, se omiten las declaraciones de variables globales y de bibliotecas de funciones.

El código de la función principal es:

```
int  main() {

// Configuración de entradas y salidas
DDRB = 0b00001111;          // Salidas: AM, PM, Zumbador y 2 puntos
DDRC = 0xFF;                // Salida a los 7 segmentos
```

```
DDRD = 0xF0;                // Botones e interruptor y salidas a
                            //transistores
PORTD = 0x0F;               // Pull-Up en las entradas

//Configuración del temporizador 2
TCCR2 = 0x04;               // Para que desborde cada medio segundo
TIMSK = 0x40;               // Interrupción por desbordamiento
ASSR = 0x08;                // Se usa al oscilador externo
sei();                      // Habilitador global de interrupciones

                            // Configura interrupciones externas
MCUCR = 0B00001010;         // INT1/INT0 por flanco de bajada (sin
                            // habilitar)

// Estado inicial del sistema: variables, banderas y salidas
h_act = 12; m_act = 0; s_act = 0;// La hora actual inicia con 12:00
h_alrm = 12; m_alrm = 0;         // La hora de la alarma es 12:00
modo = 1;                        // Modo normal, muestra la hora
AM_F = 1; AM_F_A = 1;            // Antes del medio día
ALARM = 0; CAMBIO = 0;           // Alarma apagada
PORTB = 0x02;                    // Sólo activa la salida AM

while(1) {                       // Lazo infinito
   mostrar();                    // Exhibe, según el modo
                                 // Modo es una variable global
   if( ALARM == 1 && (PIND & 0x01) ) {
       ALARM = 0;                // Apaga la alarma si está encendida e
       PORTB = PORTB & 0b11110111;// inhabilitada
       }
   if( modo_presionado() ) {
       modo++;
       if( modo == 3 )
          GICR = 0B11000000;     // Habilita INT1 e INT0
       else if( modo == 5 ) {
          if( AM_F_A == 1) {
              PORTB = PORTB & 0b11111011; // Ajusta AM y PM según la
              PORTB = PORTB | 0b00000010;// bandera AM_F_A de la alarma
          }
          else {
              PORTB = PORTB & 0b11111101;
              PORTB = PORTB | 0b00000100;
          }
       }
       else if( modo == 7 ) {
          modo = 1;
          GICR = 0x00;                  // Inhabilita INT1 e INT0
          if( AM_F == 1) {
              PORTB = PORTB & 0b11111011; // Ajusta AM y PM según la
              PORTB = PORTB | 0b00000010; // bandera AM_F de la hora actual
          }
          else {
              PORTB = PORTB & 0b11111101;
```

```
            PORTB = PORTB | 0b00000100;
         }
      }
   }   // Cierre del if de modo presionado
}      // Cierre del lazo infinito
}      // Fin de la función principal
```

Puede notarse que el código de la función principal sigue el comportamiento descrito en la figura 9.4. Algo similar ocurre con la función `ubica`, la cual debe seguir el diagrama de la figura 9.5. La función `ubica` complementa a la función `mostrar`, a continuación se muestran ambas funciones:

```
void   mostrar() {         // Función para el despliegue de datos
unsigned char  i;

ubica();                   // Coloca la información a mostrar
for( i = 0; i < 4; i++) {
   PORTC = Disp[i];        // Envía el dato al puerto
   PORTD = pgm_read_byte(&habs[i]);     // Habilita para su despliegue
   _delay_ms(5);                        // Espera se vea adecuadamente
   PORTD = 0xFF;           // Inhabilita para no introducir ruido
}                          // en otro display
}

void  ubica() {                  // Ubica los datos a mostrar en un
arreglo global

   switch(modo) {                // Según sea el modo
     case 1: Disp[3] = pgm_read_byte(&Sg7[h_act/10]);// Hora Actual
             Disp[2] = pgm_read_byte(&Sg7[h_act%10]);
             Disp[1] = pgm_read_byte(&Sg7[m_act/10]);
             Disp[0] = pgm_read_byte(&Sg7[m_act%10]);
             if( CAMBIO )        // Dos puntos parpadeando
                PORTB = PORTB | 0x01;
             else
                PORTB = PORTB & 0xFE;
             break;
     case 2: Disp[3] = 0xFF;                         // Segundos
             Disp[2] = 0xFF;
             Disp[1] = pgm_read_byte(&Sg7[s_act/10]);
             Disp[0] = pgm_read_byte(&Sg7[s_act%10]);

             break;
     case 3: if( CAMBIO ) {                          // Hora actual
                Disp[3] = pgm_read_byte(&Sg7[h_act/10]);// parpadeando
                Disp[2] = pgm_read_byte(&Sg7[h_act%10]);
             }
             else {
                Disp[3] = 0xFF;
                Disp[2] = 0xFF;
             }
             Disp[1] = pgm_read_byte(&Sg7[m_act/10]);// Minutos actuales
```

```
            Disp[0] = pgm_read_byte(&Sg7[m_act%10]); // normal
            break;
      case 4: Disp[3] = pgm_read_byte(&Sg7[h_act/10]); //Hora actual normal
            Disp[2] = pgm_read_byte(&Sg7[h_act%10]);
            if( CAMBIO ) {                    // Minutos actuales
               Disp[1] = pgm_read_byte(&Sg7[m_act/10]);// parpadeando
               Disp[0] = pgm_read_byte(&Sg7[m_act%10]);
            }
            else {
               Disp[1] = 0xFF;
               Disp[0] = 0xFF;
            }
            break;
      case 5: if( CAMBIO ) {                    // Hora de la alarma
               Disp[3] = pgm_read_byte(&Sg7[h_alrm/10]);//parpadeando
               Disp[2] = pgm_read_byte(&Sg7[h_alrm%10]);
            }
            else {
               Disp[3] = 0xFF;
               Disp[2] = 0xFF;
            }
            Disp[1] = pgm_read_byte(&Sg7[m_alrm/10]);// Minutos de la
            Disp[0] = pgm_read_byte(&Sg7[m_alrm%10]);// alarma normal

            break;
      case 6: Disp[3] = pgm_read_byte(&Sg7[h_alrm/10]);//Hora de la alarma
            Disp[2] = pgm_read_byte(&Sg7[h_alrm%10]);      // normal
            if( CAMBIO ) {
               Disp[1] = pgm_read_byte(&Sg7[m_alrm/10]);//Minutos de la
               Disp[0] = pgm_read_byte(&Sg7[m_alrm%10]);//alarma parpadeando
            }
            else {
               Disp[1] = 0xFF;
               Disp[0] = 0xFF;
            }
            break;

      }
}
```

La bandera CAMBIO es modificada cada medio segundo, en la ISR del temporizador. El efecto del parpadeo es posible porque la función `mostrar` se ejecuta varias veces durante este periodo.

La ISR del temporizador 2 sigue el comportamiento de la figura 9.6, su codificación es la siguiente:

```
ISR(TIMER2_OVF_vect) {            // Se ejecuta cada medio segundo

if( CAMBIO ) {
   CAMBIO = 0;
   s_act++;                       // Incrementa los segundos
```

```
   if( s_act == 60 ) {            // Si son 60 incrementa los minutos
       s_act = 0;
       m_act++;
       if( ALARM ) {
          ALARM = 0;              // Si la alarma está activa, la apaga
          PORTB = PORTB & 0b11110111;    // por ser otro minuto
       }
       if(m_act == 60) {          // Si son 60 minutos incrementa las horas
          m_act = 0;
          h_act++;
          if( h_act == 13)              // Con la hora 13 se regresa a 1
              h_act = 1;
          if( h_act == 12 )  {          // Con la hora 12 se ajusta la
bandera
              if( AM_F == 1 ) {         // AM_F y las salidas AM y FM
                 AM_F = 0;
                 PORTB = PORTB & 0b11111101;
                 PORTB = PORTB | 0b00000100;
              }
              else {
                 AM_F = 1;
                 PORTB = PORTB & 0b11111011;
                 PORTB = PORTB | 0b00000010;
              }
          }
       }
       // En cada nuevo minuto también revisa si se debe activar la alarma
       if( !(PIND&0x01) && AM_F==AM_F_A && h_act==h_alrm && m_act==m_alrm ) {
          ALARM = 1;
          PORTB = PORTB | 0b00001000;
       }
   }
}
else                              // Modifica la bandera si no es el
                                  // segundo
       CAMBIO = 1;                // completo
}
```

Con la ISR de la interrupción externa 0 se hacen los incrementos (botón UP), su comportamiento sigue el diagrama de la figura 9.7, el código se muestra a continuación, nuevamente se observa la importancia de la variable modo, en este caso determina el dato a configurar.

```
ISR(INT0_vect) {           // Atiende al botón UP

   switch( modo ) {
     case 3: h_act++;                   // Incrementa hora actual
             if( h_act == 13 )          // De 13 pasa a 1
                h_act = 1;
             if( h_act == 12 ) {        // En 12 ajusta bandera AM_F
                if( AM_F ) {            // y salidas AM y PM
                    AM_F = 0;
```

```
                    PORTB = PORTB & 0b11111101;
                    PORTB = PORTB | 0b00000100;
                }
                else {
                    AM_F = 1;
                    PORTB = PORTB & 0b11111011;
                    PORTB = PORTB | 0b00000010;
                }
            }
            break;
    case 4: m_act++;                    // Incrementa minutos actuales
            if( m_act == 60 )           // De 60 reinicia en 0
               m_act = 0;
            break;
    case 5: h_alrm++;                   // Incrementa la hora de la alarma
            if( h_alrm == 13 )          // De 13 pasa a 1
               h_alrm = 1;
            if( h_alrm == 12 ) {        // En 12 ajusta bandera AM_F_A
               if( AM_F_A ) {           // y salidas AM y PM
                   AM_F_A = 0;
                   PORTB = PORTB & 0b11111101;
                   PORTB = PORTB | 0b00000100;
               }
               else {
                   AM_F_A = 1;
                   PORTB = PORTB & 0b11111011;
                   PORTB = PORTB | 0b00000010;
               }
            }
            break;
    case 6: m_alrm++;                   // Incrementa minutos de la alarma
            if( m_alrm == 60 )          // De 60 reinicia en 0
               m_alrm = 0;
            break;
   }
}
```

No se incluye el código de la ISR de la interrupción externa 1 (botón DOWN) por ser muy similar al código anterior, sólo que en lugar de realizar incrementos se deben realizar decrementos. Revisando los límites adecuadamente para mostrar horas válidas y haciendo los ajustes de las banderas AM_F y AM_F_A, así como de las salidas AM y PM, cuando sean necesarios.

9.2.1.7 Integración y Evaluación

El programa completo se compiló con ayuda del AVR Studio, entorno de desarrollo de Atmel. Desde este entorno se realizó la programación del dispositivo, con el apoyo de la herramienta AVR Dragon, que es un programador y depurador de bajo costo, manufacturado por Atmel.

Puesto que la programación de los ATMega8 con el AVR Dragon es vía SPI, antes de inhabilitar la funcionalidad de *reset* en la terminal PC6, se observó el correcto

funcionamiento del sistema. Después de que PC6 es configurado como I/O genérico, el ATMega8 ya no puede ser programado por SPI.

9.2.1.8 Ajustes y Correcciones

Se requirió un ligero ajuste en el software, puesto que el botón Modo es revisado por sondeo, se requiere de un retardo para evitar rebotes. En la función `modo_presionado` inicialmente sólo se introdujo un retardo de 200 mS entre dos revisiones continuas de la terminal PD1, no obstante, el despliegue de datos se perdía momentáneamente cada vez que el botón se presionaba. Por lo tanto, se optó por realizar el retardo con 10 llamadas a la función `Muestra`, con ello, el despliegue de datos fue uniforme.

El código resultante es el siguiente:

```
unsigned  char  modo_presionado() {

if( !( PIND & 0x02 ) ) {          // Si modo está presionado
   for(int i = 0; i < 10; i++)    // Espera realizando llamadas a mostrar
       mostrar();
   if( !( PIND & 0x02 ) )         // Si aún sigue presionado regresa
       return  1;                 // verdadero
}

return 0;                                 // Sino, regresa falso
}
```

También se requirieron ajustes en el hardware, se dejó al sistema operando durante 3 días, observando un adelanto de 12 minutos en la hora actual. Puesto que la base de tiempo está dada por el oscilador externo, se conectaron 2 capacitores de 33 pF de cada uno de sus extremos con tierra, para proporcionarle estabilidad, con ello, el sistema operó por varios días mostrando un comportamiento adecuado.

El otro ajuste en el hardware consistió en el desarrollo del circuito impreso, en la figura 9.9 se muestra el prototipo obtenido. No se acomodó en un gabinete, se dejó en una placa con fines académicos.

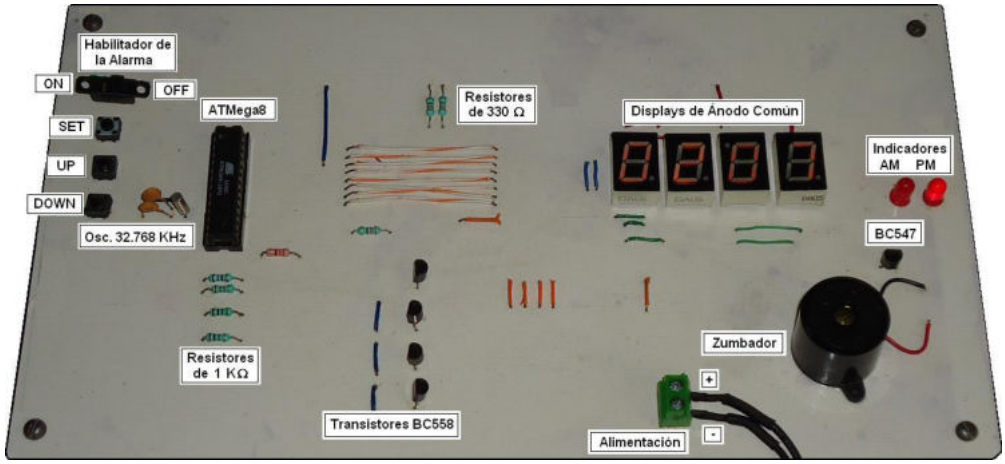


Figura 9.9 Implementación de un reloj de tiempo real, con base en un ATMega8

No fue necesaria alguna corrección en el sistema, cumplió con las expectativas planeadas.

9.2.2 Chapa Electrónica

Éste es el segundo sistema desarrollado y documentado bajo la metodología descrita al inicio del capítulo. Una chapa electrónica es un sistema utilizado en lugares con acceso restringido, como bancos, cajas de seguridad, etc. Con este sistema se controla la apertura de una puerta mediante la introducción de una clave, para la activación de un dispositivo electromecánico.

9.2.2.1 Planteamiento del Problema

La chapa electrónica debe contar con una clave de seguridad de 4 dígitos, el usuario debe introducirla por medio de un teclado y si ésta es correcta, el microcontrolador debe activar una salida conectada a un electroimán, realizando la apertura de una puerta. El estado del sistema va a conocerse por medio de un LCD. En la figura 9.10 se muestran las características esperadas en el sistema, ilustrando sus entradas y salidas.

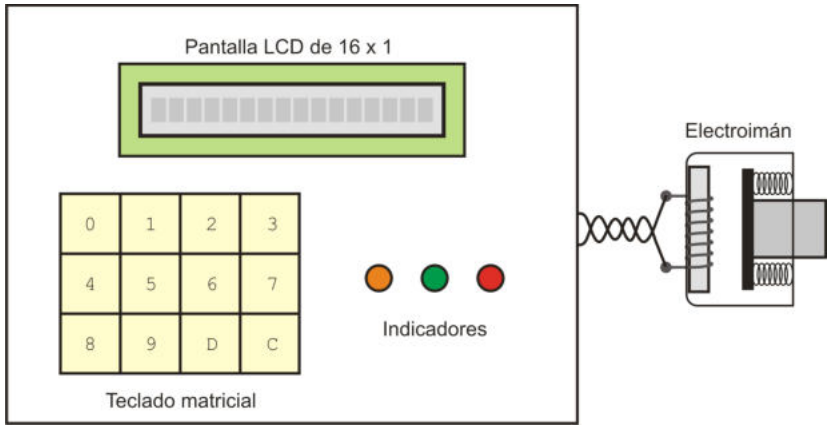


Figura 9.10 Chapa electrónica

El teclado debe disponer de 10 teclas numéricas y 2 teclas de función, la tecla D (*delete*), para borrar al último dígito introducido, dejando la posibilidad de corregir errores, y la tecla C (*change*), con la que se da paso al cambio de clave, después de haber introducido la clave correcta.

Al encender el sistema, en el LCD debe mostrarse el mensaje "Indique la Clave", quedando en espera de que el usuario presione una tecla numérica. Al presionar una tecla, el sistema inicia con el Modo de Apertura, capturando la clave para determinar si es correcta. Con la clave correcta introducida, se cuenta con un tiempo corto para atender a la tecla C, si ésta es presionada, el sistema pasa al Modo de Cambio de Clave. También debe limitarse temporalmente la introducción de información, después de algunos segundos sin actividad, el sistema debe regresar al estado de espera.

Los LEDs son indicadores visuales complementarios al LCD, inicialmente los 3 LEDs deben estar apagados. El LED naranja se debe encender después de que se ha presionado una tecla, indicando que hay actividad en progreso. El LED verde debe encenderse si la clave fue correcta y el LED rojo si la clave fue incorrecta.

Por lo tanto, además de un Estado Inicial, el sistema puede estar en uno de 2 modos de operación: el Modo de Apertura y el Modo de Cambio de Clave, el comportamiento del sistema en cada modo se describe a continuación.

En el Modo de Apertura el sistema debe tener el siguiente comportamiento:

- Con cada tecla numérica presionada en el LCD se debe mostrar un carácter de *, para que el usuario sepa cuantos números ha introducido y personas ajenas no puedan ver la clave de acceso. El LED Naranja se enciende para indicar que hay actividad en proceso.

- Se debe disponer de 10 segundos para concluir con la introducción de la clave, si el periodo termina sin haber introducido los 4 dígitos, el sistema regresar a su estado inicial.

- Durante la introducción de la clave, la tecla D puede ser utilizada para corregir los errores del usuario.

- Si la clave fue correcta, el LCD debe mostrar el mensaje "Bienvenido", se debe activar al electroimán para abrir la puerta y encender al indicador Verde. Esto durante un periodo de 3 segundos, después de los cuales el sistema debe regresar a su estado inicial.

- Mientras transcurre el periodo de 3 segundos debido a una clave correcta, el botón C puede presionarse para que el sistema pase al Modo de Cambio de Clave.

- Si la clave fue incorrecta, el LCD debe mostrar el mensaje "Clave Incorrecta" y se debe encender al indicador Rojo, también durante un periodo de 3 segundos, después de los cuales el sistema debe regresar al estado inicial.

En el Modo de Cambio de Clave el comportamiento del sistema debe ser el siguiente:

- Mientras no se presione alguna tecla, el LCD debe mostrar el mensaje "Cambio de Clave", se debe desactivar al electroimán y mantener encendido al LED Verde.

- Al presionar una tecla numérica, el LCD debe mostrar su valor para que el usuario sepa la clave que está introduciendo. El LED Naranja debe encenderse para indicar que hay actividad en proceso.

- Se debe disponer de 10 segundos para concluir con la introducción de la nueva clave, si el periodo termina sin haber introducido los 4 dígitos, el sistema debe regresar a su estado inicial, conservando la clave anterior. Este periodo debe iniciar desde que se entró al Modo de Cambio de Clave.

- Durante la introducción de la nueva clave, la tecla D puede ser utilizada para corregir los errores del usuario.

- Después de introducir los 4 dígitos, el LCD debe mostrar el mensaje "Clave Aceptada" y se debe encender al indicador Verde. Esto durante un periodo de 3 segundos, después de los cuales el sistema debe regresar al Estado Inicial.

- La nueva clave debe almacenarse en EEPROM, para que se conserve aun en ausencia de energía.

Debe notarse que del Estado Inicial sólo se puede pasar al Modo de Apertura y de éste al Modo de Cambio de Clave. De ambos modos se regresa al Estado Inicial, al concluir con la tarea planeada o al terminar el tiempo disponible para cada tarea.

Entradas del sistema:

- Un teclado matricial de 4 x 3, con 10 teclas numéricas y 2 teclas con funciones especiales: La tecla D (*delete*) para borrar al último número introducido y la tecla C (*change*) para pasar al Modo de Cambio de Clave, ésta es atendida sólo después de introducir la clave correcta.

Salidas del sistema:

- Una pantalla LCD de 16 x 1 caracteres para mostrar el estado del sistema, mediante mensajes de texto.
- Tres indicadores visuales basados en LEDs, en colores Naranja, Verde y Rojo.
- Una salida para activar al electroimán con el que se va a abrir la puerta.

Estado inicial:

El estado del sistema al ser energizado debe ser:

- **Datos internos:** Si es la primera vez que se energiza al sistema, su clave de acceso debe ser 1, 2, 3 y 4. La clave está almacenada en la memoria EEPROM del microcontrolador, pero se debe leer y dejar en SRAM para un fácil manejo.
- **Entradas:** El sistema debe sondear al teclado en espera de peticiones del usuario, sólo se deben atender las teclas numéricas, las teclas D y C inicialmente deben ignorarse.
- **Salidas:** El LCD debe mostrar un mensaje con la frase "Indique la Clave". Los 3 LEDs de estado deben estar apagados y el electroimán debe estar desactivado.

9.2.2.2 Requerimientos de Hardware y Software

Primero se debe seleccionar al MCU, de acuerdo con la figura 9.10 con 3 puertos resulta suficiente: El primero para el teclado, el segundo para un LCD manejado con una interfaz de 4 bits y el tercero para los LEDs de estado y para el electroimán. Además, observando que son pocos los requerimientos funcionales, se determina que con un ATMega8 es suficiente para el presente problema.

Los otros elementos de hardware requeridos son:

- 12 botones configurados como un teclado matricial de 4 x 3.
- 1 LCD de 16 x 1.
- 3 LEDs en colores Naranja, Verde y Rojo, con sus resistores de 330 ohms para limitar la corriente.

- 1 electroimán de 12 V con un transistor BC548 y un resistor de 330 ohms, como elementos de acondicionamiento.

En cuanto al software, las funciones que se han revisado a lo largo del texto y que pueden utilizarse son:

- Funciones para el manejo del LCD.
- Función para el sondeo del teclado.
- Funciones para la lectura y escritura en la EEPROM.

Al diseñar al software deben considerarse las 3 etapas presentes durante la operación del sistema: el Estado Inicial, el Modo de Apertura y el Modo de Cambio de Clave.

La clave de acceso va a residir en la memoria EEPROM, ésta se va a leer al iniciar con la ejecución del programa y se debe mantener en SRAM para una comparación rápida. Se debe escribir nuevamente en la EEPROM sólo cuando concluya adecuadamente el Modo de Cambio de Clave.

Por medio de variables globales se determina el flujo del programa, se requieren variables para contar los datos que introduce el usuario y para el almacenamiento de los mismos.

Para el manejo de los intervalos de tiempo se utiliza al temporizador 1, no se modifica su frecuencia de operación, va a trabajar a 1 MHz. Se utiliza una variable global para definir la duración del periodo y en la ISR del temporizador 1 se determina si ya se alcanzó el final del periodo para modificar una bandera que es consultada en el programa principal.

En el programa principal se define el periodo de tiempo requerido, se inicia la operación del temporizador 1 y continuamente se evalúa la bandera que indica el fin del periodo.

9.2.2.3 Diseño del Hardware

El diseño del hardware inicia con una revisión de los recursos internos que se van a emplear y si estos requieren de alguna de las terminales del MCU. Sólo se va a utilizar al temporizador 1, manejado por el oscilador interno y sin producir una respuesta automática (modificando alguna señal de salida), por lo tanto, ninguna de las terminales del ATMega8 está comprometida con alguna función, los periféricos externos pueden acomodarse en cualquiera de los puertos. En la figura 9.11 se muestra la organización del hardware requerido para la chapa electrónica.

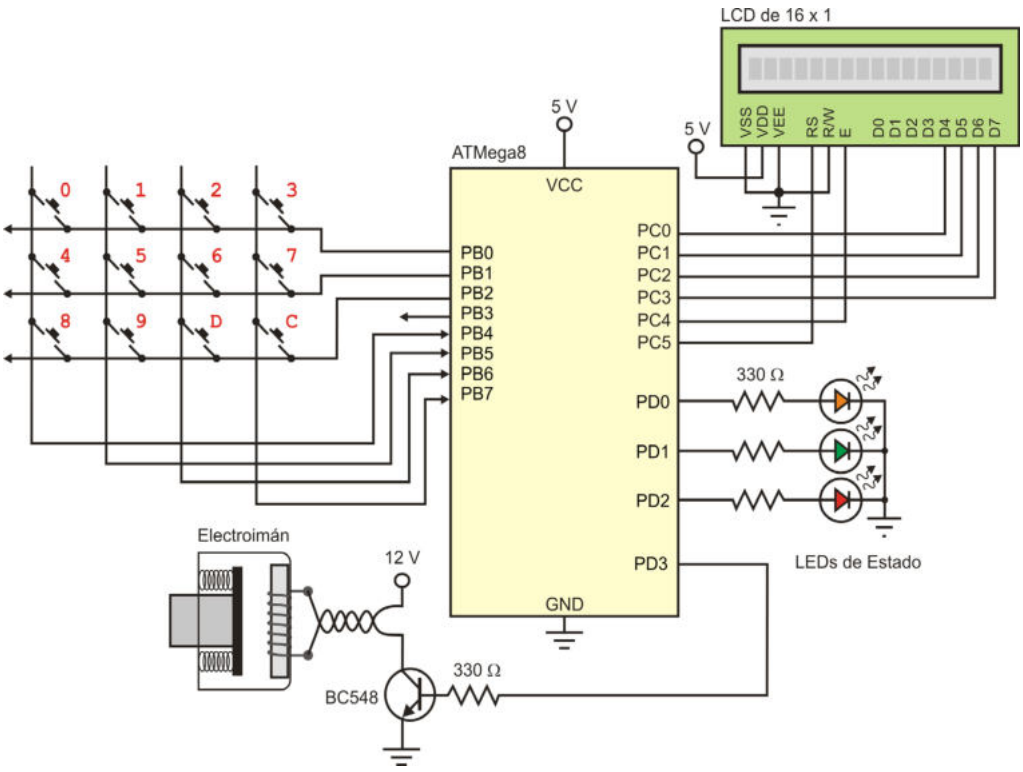


Figura 9.11 Diseño del hardware de la chapa electrónica

9.2.2.4 Diseño del Software

El diseño del software inicia con el desarrollo de un diagrama de flujo en el que se muestre el comportamiento global de sistema, éste se muestra en la figura 9.12. Por el tamaño de la figura, no es posible mostrar todos los detalles de la operación del sistema, el diagrama puede ampliarse en aquellas etapas que lo requieran, en donde no es necesario, se codifica directamente.

El diagrama de flujo muestra el comportamiento deseado para el sistema, sin embargo, se debe revisar cada etapa y determinar si es posible codificar directamente o si es necesario algún ajuste para una codificación estructurada.

La configuración de los recursos básicamente corresponde con proposiciones secuenciales que pueden codificarse directamente, lo mismo ocurre con el establecimiento del Estado Inicial, que se encuentra al inicio del lazo infinito.

La espera para que una tecla numérica sea presionada requiere de un ciclo repetitivo apoyado en la función del teclado descrita en la sección 8.2. El sistema queda en espera, mientras el valor regresado por la función no corresponda con el de una tecla numérica (0 – 9).

Luego, el flujo continúa con un ciclo repetitivo en el que se tienen 2 condiciones de salida. Estas dos posibles salidas dan claridad a la operación del sistema, aunque esta organización no es propia de la programación estructurada.

Esta parte debe reestructurarse con un ciclo repetitivo cuya terminación depende de 2 banderas, una para indicar el final del periodo (FIN_TIEMPO) y otra que la clave ha sido introducida (CLAVE_LISTA). Posteriormente, si el ciclo concluyó porque la bandera CLAVE_LISTA fue puesta en alto, se ejecuta el código subsecuente, sino, regresa al inicio del lazo infinito, justo como debe operar el sistema.

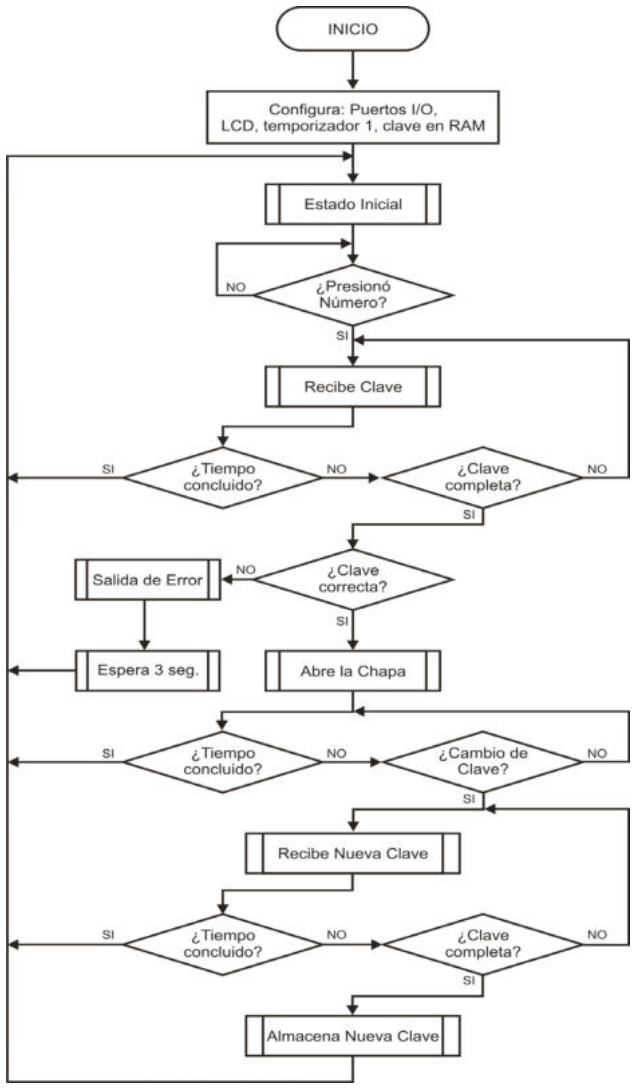


Figura 9.12 Diseño del software, comportamiento global de la chapa electrónica

En la figura 9.13 se muestra la nueva organización, en donde se puede ver un enfoque estructurado. De esta forma, la codificación básicamente requiere de un ciclo repetitivo por condición, seguido de una estructura de decisión simple.

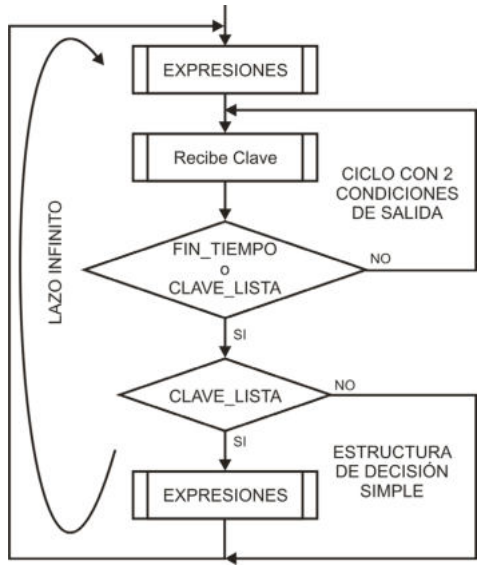


Figura 9.13 Reorganización para un enfoque estructurado

Continuando con la revisión del diagrama de flujo de la figura 9.12, las acciones para recibir la clave consisten en: Sondear el teclado, si se presionó una tecla numérica se debe conservar su valor en un arreglo, también se debe escribir un '*' en el LCD e incrementar un contador. Si se introdujeron 4 dígitos, la bandera CLAVE_LISTA debe ser puesta en alto. Si se presionó la tecla de borrado (D), se debe observar si hay datos en el arreglo (contador > 0), reducir al contador, ubicar al cursor en el LCD, escribir un espacio para borrar el último '*' y ubicar nuevamente al cursor para que la introducción de un nuevo dato se refleje adecuadamente en el LCD.

La determinación de los intervalos de tiempo se fundamenta en el uso del temporizador 1, para ello, este recurso se configura para que desborde cada medio segundo, como en el ejemplo 5.2, descrito en la sección 5.1.6. Antes de entrar al ciclo repetitivo, en una variable se almacena el doble del número de segundos requeridos (20 para 10 segundos) y se arranca al temporizador. La variable es reducida en 1 en la ISR ejecutada con cada desbordamiento y cuando alcanza el valor de 0, se pone en alto a la bandera FIN_TIEMPO.

De esta manera, en el programa principal se puede determinar si ha concluido el tiempo permitido para la introducción de la clave, únicamente sondeando a la bandera FIN_TIEMPO.

Evaluar si la Clave es correcta básicamente conlleva a la comparación de 2 arreglos de 4 datos, cuyo resultado determina el flujo en una estructura de decisión doble.

Si la clave es incorrecta, mediante expresiones secuenciales se indica el error y con el apoyo del temporizador 1 se consigue la espera de 3 segundos.

Si la clave es correcta, utilizando expresiones secuenciales se abre la chapa y nuevamente se tiene un ciclo con 2 condiciones de salida, el cual debe ser reestructurado para su codificación.

Esta situación se repite si se opta por cambiar la clave, también se tiene un tiempo de 10 segundos para concluir con la introducción de la clave. Es decir, otro ciclo repetitivo a reestructurar. La escritura de la nueva clave puede apoyarse en las funciones desarrolladas para la EEPROM o en la biblioteca *eeprom.h* del AVR Studio.

Con el apoyo de los diagramas de flujo, las descripciones textuales y el diseño del hardware, ya es posible codificar el programa en lenguaje ensamblador o en lenguaje C. El diseño del software culmina cuando ya se tienen los elementos necesarios para su codificación.

9.2..2.5 Implementación del Hardware

Consiste en el armado físico del circuito de la figura 9.11. En la figura 9.14 se muestran los resultados de la implementación en protoboard.

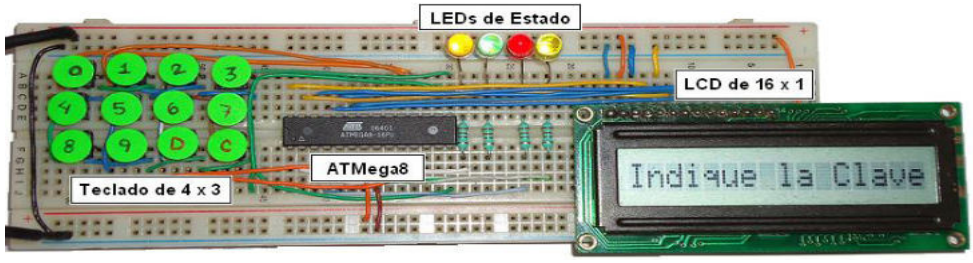


Figura 9.14 Implementación del hardware correspondiente a la chapa electrónica

No se incorporó al electroimán, en su lugar se agregó otro LED de estado. Como parte de la implementación del hardware se probaron las funciones del LCD y del teclado, para garantizar que las conexiones se realizaron de manera correcta.

9.2.2.6 Implementación del Software

El programa se implementó en Lenguaje C, dado que su comportamiento se describió con un diagrama de flujo estructurado. A continuación se muestra la codificación del programa, incluyendo las bibliotecas de funciones, las variables globales y la ISR del temporizador 1, sólo se omite la función del teclado.

El código resultante es:

```
#define F_CPU  1000000UL           // El MCU opera a 1 MHz

#include     "LCD.h"               // Funciones del LCD
#include     <avr/io.h>
#include     <util/delay.h>        // Para los retardos
#include     <avr/interrupt.h>     // Para el manejo de interrupciones
#include     <avr/eeprom.h>        // Para la EEPROM

EEMEM  unsigned char  clave_inicial[4] = { 1, 2, 3, 4};
                                   // Arreglo de 4 bytes

volatile unsigned char     tiempo;    // Para los intervalos de tiempo
volatile unsigned char     FIN_TIEMPO;// Bandera de fin de periodo

ISR (TIMER1_COMPA_vect) {             // ISR del temporizador 1
       tiempo --;
       if( tiempo == 0)
              FIN_TIEMPO = 1;
}

char  teclado();                      // Prototipo de la función del teclado

int main(void)                        // Programa Principal
{
unsigned char       tecla, i;         //valor de la tecla presionada y contador
unsigned char       clave[4];         // Clave de la chapa, en SRAM
unsigned char       clave_in[4];      // Clave de entrada
unsigned char       CLAVE_LISTA, CAMBIO_CLAVE;       // Banderas

                                      // Configuracion de los puertos

DDRB  = 0x0F;               // Entrada y salida, para el teclado
PORTB = 0xF0;               // Resistores de Pull-Up en las entradas
DDRC  = 0xFF;               // Salida para el LCD
DDRD  = 0x0F;               // Salidas para los LEDs de estado y el electroimán

                           // Inicializa al LCD
LCD_reset();

// Obtiene la clave de EEPROM y la ubica en SRAM
for( i = 0; i < 4; i++)
       clave[i] = eeprom_read_byte(i);

// Configuración parcial del temporizador 1

TIMSK = 0x10;               // El temporizador 1 interrumpe
OCR1A = 62499;              // cada medio segundo
TCCR1A = 0x00;              // pero aún no arranca (TCCR1B aún tiene 0x00)
sei();                      // Habilitador global de interrupciones
```

```
while( 1 ) {                        // Inicia el lazo infinito

                                    // Estado inicial

   TCCR1B = 0x00;                   // Temporizador 1 detenido
   PORTD = 0x00;                    // Salidas apagadas
   LCD_clear();
   LCD_write_CAD("Indique la Clave", 16);      // Mensaje inicial

                                    // Espera tecla numérica
   do {
       tecla = teclado();
   } while( tecla < 0 || tecla > 9 );

                                    // Se presionó una tecla numérica
   clave_in[0] = tecla;             // Primer dígito de la clave recibido
   i = 1;
   LCD_clear();
   LCD_cursor(0x04);                // Ubica al cursor e
   LCD_write_data('*');             // imprime un asterisco
   PORTD = 0x01;            // Enciende LED naranja (hay actividad)

                            // Recibe la clave
   TCNT1 = 0;               // Asegura que el temporizador 1 está en 0
   TCCR1B = 0x0A;           // Arranca al temporizador 1
   tiempo = 20;             // con un periodo de 10 segundos
   FIN_TIEMPO = 0;          // Banderas apagadas
   CLAVE_LISTA = 0;

   do {
       tecla = teclado();                        // Sondea al teclado
       if( tecla != 0xFF) {
          if( tecla >= 0 && tecla <= 9 ) {       // Tecla numérica
              clave_in[i] = tecla;   // guarda el valor de la tecla
              LCD_write_data('*');
              i++;
              if( i == 4 )
                 CLAVE_LISTA = 1; // Se han introducido 4 dígitos
          }
          else if(tecla == 0x0A) {               // Tecla de borrado
              if( i > 0 ) {
                 LCD_cursor(0x03 + i);           // Si hay datos, se
                                                 // borra un '*'
                 LCD_write_data(' ');
                 LCD_cursor(0x03 + i);
                 i--;
              }
          }
       }
   } while( ! ( FIN_TIEMPO || CLAVE_LISTA));

// El ciclo termina si la clave está completa o si terminó el tiempo disponible
```

```
TCCR1B = 0x00;          // Detiene al temporizador 1
if(CLAVE_LISTA) {       // Continúa si se introdujo la clave completa
                        // Compara la clave
    if( clave[0]==clave_in[0] && clave[1]==clave_in[1] &&
    clave[2]==clave_in[2] && clave[3]==clave_in[3] ){
       PORTD = 0x0A;           // Clave correcta: Abre la chapa
       LCD_clear();     // (LED verde y electroimán encendidos)
       LCD_write_CAD("<< Bienvenido >>", 16);
       TCNT1 = 0;       // Por un tiempo de 3 segundos muestra el
       TCCR1B = 0x0A;   // estado de abierto y sondea
       tiempo = 6;      // si se pide el cambio de clave
       FIN_TIEMPO = 0;
       CAMBIO_CLAVE = 0;
       do {
           tecla = teclado();
           if(tecla == 0x0B) { // Se solicitó el Cambio de clave
              PORTD = 0x02;    // LED verde encendido
              LCD_clear();
              LCD_write_CAD("Cambio de Clave", 15);
              CAMBIO_CLAVE = 1;
           }
       } while( ! ( FIN_TIEMPO || CAMBIO_CLAVE));

       TCCR1B = 0x00;          // Detiene al temporizador 1
       if( CAMBIO_CLAVE ) {
           i = 0;
           TCNT1 = 0;          // Se tienen 10 segundos para
           TCCR1B = 0x0A;      // introducir la nueva clave
           tiempo = 20;
           FIN_TIEMPO = 0;
           CLAVE_LISTA = 0;

           do {                // En este ciclo se lee la nueva
              tecla = teclado(); // clave o espera por 10 segundos
              if( tecla != 0xFF) {
                  if( tecla >= 0 && tecla <= 9 ) { // Tecla numérica
                     clave_in[i] = tecla;
                     if ( i == 0 ) {              // Con el primer
                                                  //dígito se ubica
                         LCD_clear();             // al cursor
                         LCD_cursor(0x04);
                         PORTD = 0x01;       // Enciende LED naranja
                     }                       // (hay actividad)
                     LCD_write_data(tecla + 0x30);  // Escribe al
                                                    // dígito
                     i++;                           // lo cuenta
                     if( i == 4 )// Con 4 dígitos, la clave está lista
                         CLAVE_LISTA = 1;
                  }
                  else if(tecla == 0x0A) {   // Tecla de borrado
                     if( i > 0 ) {           // Si hay datos, no considera
                         LCD_cursor(0x03 + i);// al último dígito
```

```
                        LCD_write_data(' ');
                        LCD_cursor(0x03 + i);
                        i--;
                    }
                }
            }
        } while( ! ( FIN_TIEMPO || CLAVE_LISTA));

        TCCR1B = 0x00;                    // Detiene al temporizador 1
        if(CLAVE_LISTA) {
            for( i = 0; i < 4; i++) {
                    // Respalda en EEPROM
                    eeprom_write_byte(i, clave_in[i]);
                clave[i] = clave_in[i];    // y la deja en RAM
            }                              // para su evaluación
            TCNT1 = 0;                     // Durante 3 segundos
                                           // se muestra
            TCCR1B = 0x0A;                 // el mensaje de que se
                                           // ha aceptado
            tiempo = 6;                    // la nueva clave
            FIN_TIEMPO = 0;
            PORTD = 0x02;                  // con el LED verde encendido
            LCD_clear();
            LCD_write_CAD(" CLAVE ACEPTADA", 16);
            while( !FIN_TIEMPO );
            TCCR1B = 0x00;
        }              // Fin if de nueva clave
    }                  // Fin if de cambio de clave
}                      // Fin if de clave correcta
else {
    TCNT1 = 0;         // Durante 3 segundos se muestra el mensaje
    TCCR1B = 0x0A;     // de que la clave es incorrecta
    tiempo = 6;        // con el LED rojo encendido
    FIN_TIEMPO = 0;
    PORTD = 0x04;
    LCD_clear();
    LCD_write_CAD("CLAVE INCORRECTA", 16);
    while( !FIN_TIEMPO );
    TCCR1B = 0x00;
}       // Fin else de clave incorrecta
}       // Fin if de introducción de clave completa
}       // Fin del lazo infinito

}       // Fin del programa principal
```

Puede notarse cómo el programa sigue el diagrama de flujo mostrado en la figura 9.12, con las re-estructuraciones mostradas en la figura 9.13.

9.2.2.7 Integración y Evaluación

El programa desarrollado se compiló con ayuda del entorno de desarrollo AVR Studio de Atmel. Desde ese mismo entorno se realizó la programación del dispositivo, con el apoyo de la herramienta AVR Dragon.

Se verificó el funcionamiento de la chapa electrónica, trabajó correctamente y cumplió con las especificaciones. El único detalle en su funcionalidad fue que no aceptaba los cambios de clave hasta después de que el sistema era reiniciado.

9.2.2.8 Ajustes y Correcciones

En cuanto al software, básicamente se agregó una línea más al código, para que después de escribir la clave en EEPROM, ésta también se actualizara en SRAM. El código mostrado ya incluye el ajuste requerido.

En cuanto al hardware, el único ajuste consistió en el desarrollo del circuito impreso, en la figura 9.15 se muestra el resultado obtenido.

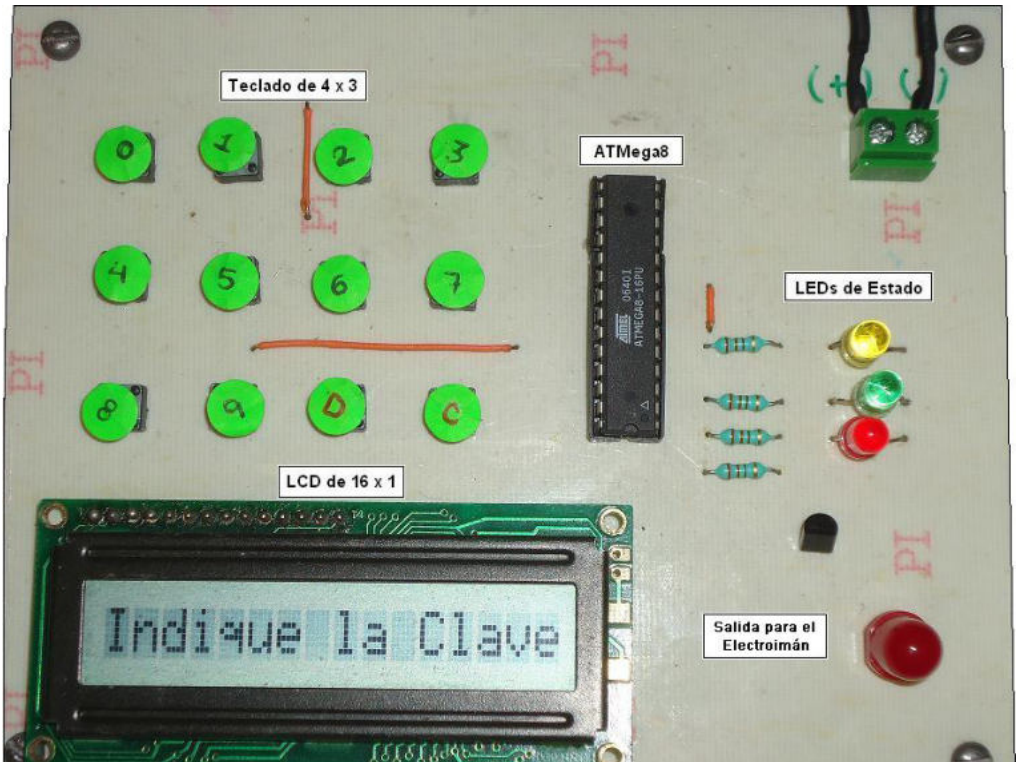


Figura 9.15 Implementación de una chapa electrónica

No fue necesaria alguna corrección al sistema, cumplió con todas las expectativas planeadas, aunque no se realizó la prueba del electroimán, en su lugar se utilizó un LED de mayores dimensiones.

9.3 Sistemas Propuestos

En la sección 9.1 se propuso una metodología y en la sección 9.2 se ilustró su aplicación con el desarrollo de 2 sistemas. Con ello, se ha dado una muestra de cómo combinar recursos internos o externos, para el desarrollo de sistemas basados en MCUs.

En esta sección se describen diversos sistemas, los cuales han sido desarrollados como proyectos finales en diferentes cursos de microcontroladores. Con esta lista se presentan otras alternativas para continuar practicando con los microcontroladores AVR de ATMel, tomando como base la metodología antes descrita. Cabe aclarar que para algunos sistemas es necesario utilizar más de un MCU.

1. **Visualización de señales analógicas en una matriz de LEDs:** El sistema requiere de una matriz por lo menos de 20 x 12 LEDs, en uno de los canales del ADC se introduce una señal analógica de baja frecuencia, la cual es mostrada en la matriz de LEDs.

2. **Grabadora y reproductora de un mensaje de voz:** El sistema cuenta con una SRAM y un DAC, dispositivos externos al MCU. Por medio de uno de los canales analógico, el sistema digitaliza una señal de voz y almacena la información en la SRAM, esto se realiza cuando se presióna un botón. Con otro botón se inicia la restauración de la señal, tomando la información de la SRAM y pasándola por el DAC para recuperar el mensaje de voz.

3. **Transmisor/receptor de mensajes utilizando tonos en código Morse:** El sistema requiere de 2 MCUs, uno para el transmisor y otro para el receptor. El código Morse describe los caracteres con puntos y rayas. En el sistema se usan tonos cortos y largos, donde un tono largo tarda 3 veces la duración de un tono corto. Un carácter es una combinación de tonos cortos y largos consecutivos. Entre caracteres debe haber un tiempo mayor al de un tono largo, para evitar confusiones. El transmisor genera los tonos, para ello cuenta con 2 botones, uno para cada tono, y una bocina. El receptor tiene un micrófono y un LCD, su función consiste en la captura de los tonos, decodificación de los caracteres y exhibición en el LCD.

4. **Visualizador de un archivo de texto en un LCD:** El sistema muestra un archivo de texto en un LCD, manejando renglón por renglón. Cuenta con 2 botones, para avanzar y retroceder entre renglones. Si un renglón supera la cantidad de caracteres visibles, se desplaza en el LCD. El sistema requiere de una memoria SRAM externa al MCU para almacenar al archivo, el cual se recibe serialmente desde una PC.

5. **Juego de memoria numérico:** El sistema cuenta con un LCD para mostrar números aleatorios. Son secuencias de 10 números que inicialmente sólo tienen 1 dígito. Se muestra un número aleatorio y el usuario debe introducirlo desde un

teclado matricial, luego otro y el usuario debe introducir los 2 números, otro y deben introducirse los 3 números (el sistema sólo muestra al último número), así sucesivamente hasta alcanzar los 10 números. Con los 10 números correctos se cubre un nivel y se pasa a un 2º nivel en donde los números son de 2 dígitos. En un 3er nivel los números son de 3 dígitos. El sistema debe dar un tiempo finito al usuario para introducir cada número, pudiendo ser 2 segundos por número. Con cualquier error el usuario pierde y debe iniciar nuevamente desde el nivel 1.

6. **Juego de memoria con una matriz de 16 LEDs**: El sistema tiene una matriz de 4 x 4 LEDs y un teclado matricial del mismo tamaño. El usuario debe memorizar las posiciones de los LEDs encendidos y reproducirlas con el teclado matricial. El sistema enciende un LED en una posición aleatoria y el usuario debe presionar el botón que le corresponde, luego el sistema enciende otro LED y el usuario debe presionar las 2 posiciones en los botones en el orden en que ocurrieron. Se enciende otro LED y el usuario debe introducir la secuencia de 3 posiciones, así sucesivamente, hasta memorizar 10 posiciones. De esta manera se desarrolla el primer nivel del juego.

   En un 2º nivel se limita el tiempo, por lo que el usuario debe mostrar una mayor habilidad al repetir las secuencias con el teclado. En todos los casos, cuando se presione una tecla, se debe reflejar su posición en la matriz de LEDs.

7. **Reloj checador electrónico:** Es un sistema con teclado y LCD en donde normalmente se muestra la hora, en tiempo real. Tiene la capacidad de administrar 10 usuarios, con un respaldo de información, al menos para 15 días, con 4 registros por día. Los usuarios se identifican con una clave de 3 dígitos, que deber introducir para registrar su acceso. La información se acumula en la memoria del sistema hasta que es solicitada serialmente desde una computadora.

8. **Teclado musical, con dos demos:** Es una aplicación interesante de los temporizadores, se trata de un sistema con un teclado matricial de 4 x 4 y una bocina, dedicando 14 de las 16 teclas para generar tonos naturales de la nota DO a la SI en 2 escalas diferentes. Se deben investigar las frecuencias que les corresponden. Las 2 teclas restantes son para demostraciones, con cada una se genera una melodía, una melodía básicamente es una combinación adecuada de frecuencias y periodos de tiempo. Además, el sistema debe contar con 2 botones adicionales, para grabar y reproducir los tonos del usuario, con un botón se inicia y termina la grabación, y con el otro se realiza su reproducción.

9. **Control de temperatura ambiental:** El sistema tiene un sensor de temperatura, un teclado, un LCD, un ventilador y una resistencia para generar calor. El usuario define un rango de temperatura (un valor mínimo y un valor máximo). El sistema conoce la temperatura actual por medio del sensor, si la temperatura ambiente está por encima del valor máximo, el sistema enciende al ventilador, activándolo con una de 3 velocidades diferentes, dependiendo de qué tan alta está la temperatura. Si la temperatura está por debajo del valor mínimo, se activa la resistencia y se enciende al ventilador con la velocidad más baja, para difundir el calor.

10. **Simulador de actividades en una casa**: Es un sistema que maneja 4 cargas de CA, cuenta con un LCD y un teclado matricial. El sistema normalmente lleva el registro de la hora, en tiempo real. El usuario debe configurar el horario de encendido y apagado de cada una de las cargas, con el apoyo del teclado, pudiendo ser diferente para cada día. Con esto, si una casa está deshabitada en algún periodo vacacional, con el sistema se tiene la apariencia de que hay alguien, si, por ejemplo, una lámpara se prende un rato por la noche, la TV en otro horario, un radio durante el día, etc. El sistema permite una programación por 3 días, consistente en el horario de encendido y apagado diario, para cada una de las cargas. La programación se repite si el sistema se deja operando por más tiempo.

11. **Dimmer con modo manual y automático:** El sistema controla el ángulo de disparo de un triac con el que se maneja una lámpara incandescente. En el modo manual, se tienen 5 ángulos de disparo diferentes y se avanza entre ellos con un botón. En el modo automático, el ángulo de disparo es inversamente proporcional a la cantidad de luz en el ambiente, para conocer este parámetro se utiliza un fotosensor.

12. **Decodificador de un control remoto comercial y activación de 5 cargas de CA:** El MCU decodifica las señales que entrega un control remoto comercial. Cuando se presionan las teclas 1, 2, 3, 4 y 5, dependiendo de la señal recibida, se enciende o apaga alguna carga de alterna. Entre un encendido y apagado se deja un periodo de tiempo cercano a 5 segundos, para evitar oscilaciones, dado que el control remoto envía la información en repetidas ocasiones.

13. **Móvil manipulado por un control remoto comercial:** Es necesario construir un móvil con 2 motores independientes, contando con la posibilidad de avanzar, retroceder, girar a la izquierda o a la derecha. El MCU decodifica las señales de un control remoto comercial e identifica 5 de ellas, para comandar al móvil y que éste realice alguno de los movimientos citados o se detenga. Es decir, si el móvil recibe el comando para avanzar, continúa avanzando mientras no reciba otro comando.

14. **Emulación del control de un horno de microondas:** Sistema con un teclado matricial y 4 displays de 7 segmentos, el cual realiza las funciones comunes de un horno, simulando la generación de microondas con un foco de CA. Cuenta con un botón para detectar la apertura de la puerta del horno.

15. **Animación en una matriz de 10 x 10 LEDs**: El sistema contiene 10 matrices binarias de información y las va mostrando una a una, en diversos instantes de tiempo. La animación se genera porque los contenidos de las matrices están muy relacionados. El sistema es versátil, dado que el contenido de las matrices puede modificarse serialmente desde una PC y con un par de botones puede aumentarse o reducirse el tiempo de exposición de cada matriz, es decir, se puede modificar la velocidad de la animación. La animación es continua, después de exhibir a la última matriz se inicia nuevamente con la primera.

16. **Marquesina de mensajes con 4 matrices comerciales de 7 x 5 LEDs**: El sistema recibe un mensaje por el puerto serie y lo almacena en la EEPROM del MCU. El

mensaje se recibe en código ASCII, en el sistema se incluyen constantes con los códigos de 7 x 5 pixeles para los diferentes caracteres alfa-numéricos.

17. **Alarma de 5 zonas con 4 claves de acceso**: Sistema de seguridad que inicialmente sólo puede ser operado con una clave de acceso de 4 dígitos. Esta clave es para el administrador y con ella se pueden dar de alta o baja a otras 3 claves (3 usuarios). Todas las claves deben conservarse en la EEPROM del MCU. Se tienen 3 tipos de zonas: entrada/salida, interior y 24 horas. El tipo de zona para cada sensor debe ser configurado por el administrador.

    Cuando la alarma está armada, una zona entrada/salida proporciona 15 segundos después de que su sensor fue irrumpido, para desarmar a la alarma antes de activar la sirena. Las zonas interiores no proporcionan este retraso de tiempo, si la alarma está armada, activan a la sirena inmediatamente después de que el sensor fue irrumpido. Las zonas del tipo 24 horas son para puertas que regularmente deben estar cerradas, activan a la sirena independientemente de que la alarma esté armada o no.

    Sin importar la causa de la activación de la sirena, ésta se apaga al introducir cualquier clave, del administrador o de los usuarios. Los usuarios sólo pueden armar o desarmar a la alarma, el administrador, además, puede configurar al sistema: definir el tipo para cada una de las 5 zonas, cambiar su clave y dar de alta o baja a los 3 usuarios.

18. **Tablero de fútbol**: Sistema con 11 displays de 7 segmentos para mostrar: el tiempo restante del juego (4 displays), el periodo actual (1 display), el marcador (2 displays para el equipo local y 2 para el visitante) y el número de faltas de cada equipo (2 displays, uno para el equipo local y otro para el visitante). Además, debe incluir un zumbador para indicar la conclusión de un periodo, así como un teclado para ajustes iniciales y control durante un partido. Por la cantidad de displays a manejar, puede ser conveniente emplear más de un MCU.

19. **Juego del gato o tres en raya:** Sistema con una matriz de 3 x 3 LEDs bicolor y un teclado, también de 3 x 3 para generar los tiros. Permite jugar a 2 usuarios o a 1 usuario contra el sistema. Incluye indicadores visuales para indicar al ganador o si el juego terminó empatado.

20. **Chapa electrónica con tarjetas telefónicas:** Sistema que decodifica la clave de una tarjeta telefónica, para activar un electroimán con el que se abre una puerta. Cuenta con una tarjeta maestra, con la que se pueden dar de alta o baja hasta 3 tarjetas diferentes. Incluye indicadores visuales que reflejan el estado del sistema.

21. **Manipulación de un brazo robótico basado en servomotores o motores de paso:** El sistema hace que el brazo automáticamente repita secuencias de movimiento introducidas manualmente. Con un MCU se manejan los motores del brazo, por cada motor se tienen 2 botones, para tener un movimiento en ambas direcciones. Además, son necesarios otros 2 botones, uno para iniciar y terminar de grabar la secuencia de movimientos y el otro para repetir la

secuencia grabada. El brazo se puede manipular libremente con los botones de los motores sin haber alguna secuencia bajo grabación, permitiendo al usuario ensayar trayectorias diferentes. Para mantener al brazo operando en un espacio válido, la secuencia se repite en el orden inverso a como fue registrada.

22. **Reloj de tiempo real y termómetro ambiental**: Sistema con 4 displays grandes en los que se muestra la hora y la temperatura ambiente, conmutando la información cada 3 segundos. Aunque el MCU lleva la hora, con cada reinicio la toma de un DS1307. El cual es un reloj de tiempo real con interfaz I^2C. El MCU se sincroniza con el DS1307 cada 24 horas, para mantener la hora exacta con menos esfuerzo. Es una variante del reloj desarrollado en la sección 9.2, pero ahora no se tiene alarma. Al configurar la hora, su nuevo valor debe respaldarse en el DS1307.

23. **Reloj de ajedrez:** Es una combinación de 2 relojes, cada uno basado en 4 displays de 7 segmentos (para minutos y segundos). El sistema tiene 3 botones de configuración, 1 botón de inicio, 2 botones grandes para cambio de turno y 1 zumbador. Al encender el sistema, debe configurarse la cantidad de minutos que va a durar una partida de ajedrez. Cuando se presiona el botón de inicio, el tiempo en el reloj del jugador 1 comienza a descender. Después de que el jugador 1 realiza su tirada debe presionar su botón de cambio de turno, su reloj se detiene y es el reloj del jugador 2 el que comienza a descender. De manera similar, después de que el jugador 2 realiza su tirada debe presionar su botón de cambio de turno, para detener a su reloj y sea el del jugador 1 el que descienda. Así se continúa durante el desarrollo del juego. El zumbador se activa cuando alguno de los relojes ha llegado a cero, indicando que ha concluido el tiempo para definir un ganador.

24. **Juego de cuatro en raya:** Es un juego que normalmente se desarrolla sobre un tablero de 6 x 7 casillas, el cual se encuentra vacío al comienzo de una partida. Se juega entre 2 oponentes, cada uno coloca una ficha de un color diferente en el tablero, comenzando en la parte inferior y sin dejar espacios vacíos. Gana quien consigue colocar 4 fichas en una línea continua, horizontal, vertical o diagonal.

    En este sistema el tablero de juego se implementa con una matriz de 6 x 7 LEDs bicolor. También es necesario un renglón de 7 LEDs bicolor y 3 botones para seleccionar y realizar las tiradas. Con 2 botones el usuario "desplaza" un LED encendido para elegir una columna, pudiendo realizar movimientos hacia la derecha o izquierda, y con el tercer botón realiza el tiro. Para complementar el sistema, contiene LEDs de estado que indiquen quien ganó el juego.

Apéndice A: Resumen de los Registros I/O

En la tabla A.1 se muestran todos los registros disponibles en un ATMega16, sombreando aquellos registros o bits que no están implementados en un ATMega8 (o en ambos). La dirección entre paréntesis sería empleada si el registro fuera tratado como SRAM.

Tabla A.1 Resumen de los Registros I/O

Dirección	Nombre	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
$3F ($5F)	SREG	I	T	H	S	V	N	Z	C
$3E ($5E)	SPH	-	-	-	-	-	SP10	SP9	SP8
$3D ($5D)	SPL	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0
$3C ($5C)	OCR0	Registro de comparación del Temporizador/Contador 0							
$3B ($5B)	GICR	INT1	INT0	INT2	-	-	-	IVSEL	IVCE
$3A ($5A)	GIFR	INTF1	INTF0	INTF2	-	-	-	-	-
$39 ($59)	TIMSK	OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	OCIE0	TOIE0
$38 ($58)	TIFR	OCF2	TOV2	ICF1	OCF1A	OCF1B	TOV1	OCF0	TOV0
$37 ($57)	SPMCR	SPMIE	RWWSB	-	RWWSRE	BLBSET	PGWRT	PGERS	SPMEN
$36 ($56)	TWCR	TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	-	TWIE
$35 ($55)	MCUCR	SM2 / SE	SE / SM2	SM1	SM0	ISC11	ISC10	ISC01	ISC00
$34 ($54)	MCUCSR	JTD	ISC2	-	JTRF	WDRF	BORF	EXTRF	PORF
$33 ($53)	TCCR0	FOC0	WGM00	COM01	COM00	WGM01	CS02	CS01	CS00
$32 ($52)	TCNT0	Temporizador/Contador 0							
$31($51)	OSCCAL	Registro de calibración del oscilador							
	OCDR	Registro del depurador interno							
$30 ($50)	SFIOR	ADTS2	ADTS1	ADTS0	-	ACME	PUD	PSR2	PSR10
$2F ($4F)	TCCR1A	COM1A1	COM1A0	COM1B1	COM1B0	FOC1A	FOC1B	WGM11	WGM10
$2E ($4E)	TCCR1B	ICNC1	ICES1	-	WGM13	WGM12	CS12	CS11	CS10
$2D ($4D)	TCNT1H	Byte alto del Temporizador/Contador 1							
$2C ($4C)	TCNT1L	Byte bajo del Temporizador/Contador 1							
$2B ($4B)	OCR1AH	Byte alto del registro A para la comparación del Temporizador/Contador 1							
$2A ($4A)	OCR1AL	Byte bajo del registro A para la comparación del Temporizador/Contador 1							
$29 ($49)	OCR1BH	Byte alto del registro B para la comparación del Temporizador/Contador 1							
$28 ($48)	OCR1BL	Byte bajo del registro B para la comparación del Temporizador/Contador 1							
$27 ($47)	ICR1H	Byte alto del registro de captura del Temporizador/Contador 1							
$26 ($46)	ICR1L	Byte alto del registro de captura del Temporizador/Contador 1							
$25 ($45)	TCCR2	FOC2	WGM20	COM21	COM20	WGM21	CS22	CS21	CS20
$24 ($44)	TCNT2	Temporizador/Contador 2							
$23 ($43)	OCR2	Registro de comparación del Temporizador/Contador 2							
$22 ($42)	ASSR	-	-	-	-	AS2	TCN2UB	OCR2UB	TCR2UB
$21 ($41)	WDTCR	-	-	-	WDCE / WDTOE	WDE	WDP2	WDP1	WDPO
$20 ($40)	UBRRH	URSEL	-	-	-	UBRR[11:8]			
	UCSRC	URSEL	UMSEL	UPM1	UPM0	USBS	UCSZ1	UCSZ0	UCPOL
$1F ($3F)	EEARH	-	-	-	-	-	-	-	EEAR8

Dirección	Nombre	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
$1E ($3E)	EEARL	EEAR7	EEAR6	EEAR5	EEAR4	EEAR3	EEAR2	EEAR1	EEAR0
$1D ($3D)	EEDR	Registro de datos de la EEPROM							
$1C ($3C)	EECR	-	-	-	-	EERIE	EEMWE	EEWE	EERE
$1B ($3B)	PORTA	PORTA7	PORTA6	PORTA5	PORTA4	PORTA3	PORTA2	PORTA1	PORTA0
$1A ($3A)	DDRA	DDA7	DDA6	DDA5	DDA4	DDA3	DDA2	DDA1	DDA0
$19 ($39)	PINA	PINA7	PINA6	PINA5	PINA4	PINA3	PINA2	PINA1	PINA0
$18 ($38)	PORTB	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0
$17 ($37)	DDRB	DDB7	DDB6	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0
$16 ($36)	PINB	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0
$15 ($35)	PORTC	PORTC7	PORTC6	PORTC5	PORTC4	PORTC3	PORTC2	PORTC1	PORTC0
$14 ($34)	DDRC	DDC7	DDC6	DDC5	DDC4	DDC3	DDC2	DDC1	DDCO
$13 ($33)	PINC	PINC7	PINC6	PINC5	PINC4	PINC3	PINC2	PINC1	PINC0
$12 ($32)	PORTD	PORTD7	PORTD6	PORTD5	PORTD4	PORTD3	PORTD2	PORTD1	PORTD0
$11 ($31)	DDRD	DDD7	DDD6	DDD5	DDD4	DDD3	DDD2	DDD1	DDD0
$10 ($30)	PIND	PIND7	PIND6	PIND5	PIND4	PIND3	PIND2	PIND1	PIND0
$0F ($2F)	SPDR	Registro de datos de la interfaz SPI							
$0E ($2E)	SPSR	SPIF	WCOL	-	-	-	-	-	SPI2X
$0D ($2D)	SPCR	SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0
$0C ($2C)	UDR	Registro de datos de la USART							
$0B ($2B)	UCSRA	RXC	TXC	UDRE	FE	DOR	PE	U2X	MPCM
$0A ($2A)	UCSRB	RXCIE	TXCIE	UDRIE	RXEN	TXEN	UCSZ2	RXB8	TXB8
$09 ($29)	UBRRL	Registro de la taza de baudios de la USART Byte bajo							
$08 ($28)	ACSR	ACD	ACBG	ACO	ACI	ACIE	ACIC	ACIS1	ACIS0
$07 ($27)	ADMUX	REFS1	REFS0	ADLAR	MUX4	MUX3	MUX2	MUX1	MUX0
$06 ($26)	ADCSRA	ADEN	ADSC	ADFR ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0
$05 ($25)	ADCH	Registro de datos del ADC Byte alto							
$04 ($24)	ADCL	Registro de datos del ADC Byte bajo							
$03 ($23)	TWDR	Registro de datos de la interfaz serial de dos hilos (TWI)							
$02 ($22)	TWAR	TWA6	TWA5	TWA4	TWA3	TWA2	TWA1	TWA0	TWGCE
$01 ($21)	TWSR	TWS7	TWS6	TWS5	TWS4	TWS3	-	TWPS1	TWPS0
$00 ($20)	TWBR	Registro de Bit Rate de la interfaz TWI							

El signo $ también es reconocido por el AVR Studio para denotar números en hexadecimal, equivale a 0x.

En el registro **MCUCR** (0x35), los bits **SM2** y **SE** están intercambiados, refiriendo a un ATMega8 y a un ATMega16, en la tabla A.1 se sombreó la ubicación en el ATMega16.

En el registro **WDTCR** (0x21), el bit ubicado en la posición 4 tiene un nombre diferente para cada MCU, en el ATMega8 se denomina **WDCE** y en el ATMega16 su nombre es **WDTOE** (sombreado en la tabla A.1), aunque su funcionalidad es la misma.

En el registro **ADCSRA** (0x06), el bit ubicado en la posición 5 se llama **ADFR** para un ATMega8 y habilita un modo de carrera libre. En un ATMega16 el bit se llama **ADATE** (sombreado en la tabla A.1) y además del modo de carrera libre habilita diferentes opciones para el auto disparo del ADC por otros recursos de hardware.

Apéndice B: Resumen del Repertorio de Instrucciones

El repertorio completo se muestra en la tabla B.1, indicando las instrucciones que no están disponibles en un ATMega8.

Tabla B.1 Repertorio de instrucciones de un ATMega8 y un ATMega16

Instrucción		Descripción	Operación	Banderas
Instrucciones Aritméticas y Lógicas				
ADD	Rd, Rs	Suma sin acarreo	Rd = Rd + Rs	Z,C,N,V,H,S
ADC	Rd, Rs	Suma con acarreo	Rd = Rd + Rs + C	Z,C,N,V,H,S
ADIW	Rd, k	Suma constante a palabra	[Rd + 1:Rd] = [Rd + 1:Rd] + k	Z,C,N,V,S
SUB	Rd, Rs	Resta sin acarreo	Rd = Rd – Rs	Z,C,N,V,H,S
SUBI	Rd, k	Resta constante	Rd = Rd – k	Z,C,N,V,H,S
SBC	Rd, Rs	Resta con acarreo	Rd = Rd – Rs – C	Z,C,N,V,H,S
SBCI	Rd, k	Resta constante con acarreo	Rd = Rd – k – C	Z,C,N,V,H,S
SBIW	Rd, k	Resta constante a palabra	[Rd + 1:Rd] = [Rd + 1:Rd] - k	Z,C,N,V,S
MUL	Rd, Rs	Multiplicación sin signo	R1:R0 = Rd * Rs	Z,C
MULS	Rd, Rs	Multiplicación con signo	R1:R0 = Rd * Rs	Z,C
MULSU	Rd, Rs	Multiplicación de un número con signo y otro sin signo	R1:R0 = Rd * Rs	Z,C
FMUL	Rd, Rs	Multiplicación fraccional sin signo	R1:R0 = (Rd * Rs) << 1	Z,C
FMULS	Rd, Rs	Multiplicación fraccional con signo	R1:R0 = (Rd * Rs) << 1	Z,C
FMULSU	Rd, Rs	Multiplicación fraccional de un número con signo y otro sin signo	R1:R0 = (Rd * Rs) << 1	Z,C
AND	Rd, Rs	Operación lógica AND	Rd = Rd AND Rs	Z,N,V,S
ANDI	Rd, k	Operación lógica AND con una constante	Rd = Rd AND k	Z,N,V,S
OR	Rd, Rs	Operación lógica OR	Rd = Rd OR Rs	Z,N,V,S
ORI	Rd, k	Operación lógica OR con una constante	Rd = Rd OR k	Z,N,V,S
EOR	Rd, Rs	Operación lógica OR Exclusiva	Rd = Rd XOR Rs	Z,N,V,S
SBR	Rd, k	Pone en alto los bits indicados en la constante	Rd = Rd OR k	Z,C,N,V,S
CBR	Rd, k	Pone en bajo los bits indicados en la constante	Rd = Rd AND (0xFF – k)	Z,C,N,V,S
COM	Rd	Complemento a 1	Rd = 0xFF – Rd	Z,C,N,V,S
NEG	Rd	Negado o complemento a 2	Rd = 0x00 – Rd	Z,C,N,V,H,S
INC	Rd	Incrementa un registro	Rd = Rd + 1	Z,N,V,S
DEC	Rd	Disminuye un registro	Rd = Rd – 1	Z,N,V,S
TST	Rd	Evalúa un registro	Rd = Rd AND Rd	Z,C,N,V,S
CLR	Rd	Limpia un registro (pone en bajo)	Rd = 0x00	Z,C,N,V,S
SER	Rd	Ajusta un registro (pone en alto)	Rd = 0xFF	-
Instrucciones para el Control de Flujo				
RJMP	k	Salto relativo	PC = PC + k + 1	-
IJMP		Salto indirecto	PC = Z	-
JMP	k(1)	Salto absoluto	PC = k	-
RCALL	k	Llamada relativa a una rutina	PC = PC + k + 1, PILA ← PC + 1	-
ICALL		Llamada indirecta a una rutina	PC = Z, PILA ← PC + 1	-
CALL	k(1)	Llamada absoluta a una rutina	PC = k, PILA ← PC + 1	-
RET		Retorno de una rutina	PC ← PILA	-
RETI		Retorno de rutina de interrupción	PC ← PILA, I = 1	-
CP	Rd, Rs	Compara dos registros	Rd – Rs	Z,C,N,V,H,S
CPC	Rd, Rs	Compara registros con acarreo	Rd – Rs – C	Z,C,N,V,H,S
CPI	Rd, k	Compara un registro con una constante	Rd – k	Z,C,N,V,H,S
BRBS	s, k	Brinca si el bit s del registro estado está en alto	si(SREG(s) == 1) PC = PC + k + 1	-
BRBC	s, k	Brinca si el bit s del registro estado está en bajo	si (SREG(s) == 0) PC = PC + k + 1	-
BRIE	k	Brinca si las interrupciones están habilitadas	si (I == 1) PC = PC + k + 1	-
BRID	k	Brinca si las interrupciones están inhabilitadas	si (I == 0) PC = PC + k + 1	-
BRTS	k	Brinca si el bit T está en alto	si (T == 1) PC = PC + k + 1	-
BRTC	k	Brinca si el bit T está en bajo	si (T == 0) PC = PC + k + 1	-
BRHS	k	Brinca si hubo acarreo del nibble bajo al alto	si (H == 1) PC = PC + k + 1	-
BRHC	k	Brinca si no hubo acarreo del nibble bajo al alto	si (H == 0) PC = PC + k + 1	-
BRGE	k	Brinca si es mayor o igual que (con signo)	si (S == 0) PC = PC + k + 1	-
BRLT	k	Brinca si es menor que (con signo)	si (S == 1) PC = PC + k + 1	-

Instrucción		Descripción	Operación	Banderas
BRVS	k	Brinca si hubo sobreflujo aritmético	si (V == 1) PC = PC + k + 1	-
BRVC	k	Brinca si no hubo sobreflujo aritmético	si (V == 0) PC = PC + k + 1	-
BRMI	k	Brinca si es negativo	si (N == 1) PC = PC + k + 1	-
BRPL	k	Brinca si no es negativo	si (N == 0) PC = PC + k + 1	-
BREQ	k	Brinca si los datos son iguales	si (Z == 1) PC = PC + k + 1	-
BRNE	k	Brinca si los datos no son iguales	si (Z == 0) PC = PC + k + 1	-
BRSH	k	Brinca si es mayor o igual	si (C == 0) PC = PC + k + 1	-
BRLO	k	Brinca si es menor	si (C == 1) PC = PC + k + 1	-
BRCS	k	Brinca si hubo acarreo	si (C == 1) PC = PC + k + 1	-
BRCC	k	Brinca si no hubo acarreo	si (C == 0) PC = PC + k + 1	-
CPSE	Rd, Rs	Un saltito si los registros son iguales	si(Rd == Rs) PC = PC + 2 o 3	-
SBRS	Rs, b	Un saltito si el bit b del registro Rs está en alto	si(Rs(b) == 1) PC = PC + 2 o 3	-
SBRC	Rs, b	Un saltito si el bit b del registro Rs está en bajo	si(Rs(b) == 0) PC = PC + 2 o 3	-
SBIS	P, b	Un saltito si el bit b del registro P está en alto, P es un Registro I/O	si(I/O(P, b) == 1) PC = PC + 2 o 3	-
SBIC	P, b	Un saltito si el bit b del registro P está en bajo, P es un Registro I/O	si(I/O(P, b) == 0) PC = PC + 2 o 3	-
Instrucciones para Transferencia de Datos				
MOV	Rd, Rs	Copia un registro	Rd = Rs	-
MOVW	Rd, Rs	Copia un par de registros	Rd + 1: Rd = Rs + 1: Rs, Rd y Rs registros pares	-
LDI	Rd, k	Copia la constante en el registro	Rd = k	-
LDS	Rd, k	Carga directa de memoria	Rd = Mem[k]	-
LD	Rd, X	Carga indirecta de memoria	Rd = Mem[X]	-
LD	Rd, X+	Carga indirecta con post-incremento	Rd = Mem[X], X = X + 1	-
LD	Rd, -X	Carga indirecta con pre-decremento	X = X - 1, Rd = Mem[X]	-
LD	Rd, Y	Carga indirecta de memoria	Rd = Mem[Y]	-
LD	Rd, Y+	Carga indirecta con post-incremento	Rd = Mem[Y], Y = Y + 1	-
LD	Rd, -Y	Carga indirecta con pre-decremento	Y = Y - 1, Rd = Mem[Y]	-
LDD	Rd, Y + q	Carga indirecta con desplazamiento	Rd = Mem[Y + q]	-
LD	Rd, Z	Carga indirecta de memoria	Rd = Mem[Z]	-
LD	Rd, Z+	Carga indirecta con post-incremento	Rd = Mem[Z], Z = Z + 1	-
LD	Rd, -Z	Carga indirecta con pre-decremento	Z = Z - 1, Rd = Mem[Z]	-
LDD	Rd, Z + q	Carga indirecta con desplazamiento	Rd = Mem[Z + q]	-
STS	k, Rs	Almacenamiento directo en memoria	Mem[k] = Rs	-
ST	X, Rs	Almacenamiento indirecto en memoria	Mem[X] = Rs	-
ST	X+, Rs	Almacenamiento indirecto con post-incremento	Mem[X] = Rs, X = X + 1	-
ST	-X, Rs	Almacenamiento indirecto con pre-decremento	X = X - 1, Mem[X] = Rs	-
ST	Y, Rs	Almacenamiento indirecto en memoria	Mem[Y] = Rs	-
ST	Y+, Rs	Almacenamiento indirecto con post-incremento	Mem[Y] = Rs, Y = Y + 1	-
ST	-Y, Rs	Almacenamiento indirecto con pre-decremento	Y = Y - 1, Mem[Y] = Rs	-
STD	Y + q, Rs	Almacenamiento indirecto con desplazamiento	Mem[Y + q] = Rs	-
ST	Z, Rs	Almacenamiento indirecto en memoria	Mem[Z] = Rs	-
ST	Z+, Rs	Almacenamiento indirecto con post-incremento	Mem[Z] = Rs, Z = Z + 1	-
ST	-Z, Rs	Almacenamiento indirecto con pre-decremento	Z = Z - 1, Mem[Z] = Rs	-
STD	Z + q, Rs	Almacenamiento indirecto con desplazamiento	Mem[Z + q] = Rs	-
PUSH	Rs	Inserta a Rs en la pila	Mem[SP] = Rs, SP = SP – 1	-
POP	Rd	Extrae de la pila y coloca en Rd	SP = SP + 1, Rd = Mem[SP]	-
LPM		Carga indirecta de memoria de programa en R0	R0 = Flash[Z]	-
LPM	Rd, Z	Carga indirecta de memoria de programa en Rd	Rd = Flash[Z]	-
LPM	Rd, Z+	Carga indirecta con post-incremento	Rd = Flash[Z], Z = Z + 1	-
SPM		Almacenamiento indirecto en memoria de programa	Flash[Z] = R1:R0	-
IN	Rd, P	Lee de un Registro I/O, deja el resultado en Rd	Rd = P	-

Instrucción		Descripción	Operación	Banderas
OUT	P, Rs	Escribe en un Registro I/O, el valor de Rs	P = Rs	-
Instrucciones para el Manejo de Bits				
LSL	Rd	Desplazamiento lógico a la izquierda	C=Rd(7),Rd(n+1)=Rd(n),Rd(0)=0	Z,C,N,V,S
LSR	Rd	Desplazamiento lógico a la derecha	C=Rd(0),Rd(n)=Rd(n+1),Rd(7)=0	Z,C,N,V,S
ROL	Rd	Rotación a la izquierda	C=Rd(7),Rd(n+1)=Rd(n),Rd(0)=C	Z,C,N,V,S
ROR	Rd	Rotación a la derecha	C=Rd(0),Rd(n)=Rd(n+1),Rd(7)=C	Z,C,N,V,S
ASR	Rd	Desplazamiento aritmético a la derecha	Rd(n) = Rd(n + 1), n = 0 .. 6	Z,C,N,V,S
SWAP	Rd	Intercambia nibbles en Rd	Rd(7..4) = Rd(3..0) Rd(3..0) = Rd(7..4)	-
SBI	P, b	Pone en alto al bit b del Registro P	P(b) = 1	-
CBI	P, b	Pone en bajo al bit b del Registro P	P(b) = 0	-
BTS	Rs, b	Almacena al bit b de Rs en el bit T de SREG	T = Rs(b)	T
BTL	Rd, b	Carga en el bit b de Rd desde el bit T de SREG	Rd(b) = T	-
BSET	s	Pone en alto al bit s del registro Estado	SREG(s) = 1	SREG(s)
BCLR	s	Pone en bajo al bit s del registro Estado	SREG(s) = 0	SREG(s)
SEI		Pone en alto al habilitador de interrupciones	I = 1	I
CLI		Pone en bajo al habilitador de interrupciones	I = 0	I
SET		Pone en alto al bit de transferencias	T = 1	T
CLT		Pone en bajo al bit de transferencias	T = 0	T
SEH		Pone en alto a la bandera de acarreo del nibble bajo	H = 1	H
CLH		Pone en bajo a la bandera de acarreo del nibble bajo	H = 0	H
SES		Pone en alto a la bandera de signo	S = 1	S
CLS		Pone en bajo a la bandera de signo	S = 0	S
SEV		Pone en alto a la bandera de sobreflujo	V = 1	V
CLV		Pone en bajo a la bandera de sobreflujo	V = 0	V
SEN		Pone en alto a la bandera de negativo	N = 1	N
CLN		Pone en bajo a la bandera de negativo	N = 0	N
SEZ		Pone en alto a la bandera de cero	Z = 1	Z
CLZ		Pone en bajo a la bandera de cero	Z = 0	Z
SEC		Pone en alto a la bandera de acarreo	C = 1	C
CLC		Pone en bajo a la bandera de acarreo	C = 0	C
Instrucciones Especiales				
NOP		No operación, empleada para forzar una espera de 1 ciclo de reloj	-	-
SLEEP		Introduce al MCU al modo de bajo consumo previamente seleccionado	MCU en bajo consumo	-
WDR		Reinicia al Watchdog Timer, para evitar reinicios por su desbordamiento	WDT ← 0	-
BREAK[1]		Utilizada para depuración, desde el puerto JTAG	-	-

Nota[1]: Instrucciones no disponibles en un ATMega8.

Apéndice C: Uso del AVR Studio

El entorno del AVR Studio únicamente incluye al programa ensamblador (AVRASM), no obstante, proporciona las facilidades para enlazarse con compiladores de lenguaje C desarrollados por alguna fuente diferente a Atmel. Instalando al compilador adecuado, desde el mismo entorno es posible la edición de programas, la invocación del compilador con exhibición de resultados, su simulación y depuración en lenguaje C.

Uno de estos compiladores es el avr-gcc, incluido en una suite conocida como WinAVR. La suite se puede descargar de manera gratuita del sitio http://winavr.sourceforge.net/. Este compilador fue utilizado en los diferentes ejemplos a lo largo del texto y también se emplea en este apéndice. Por lo tanto, para el desarrollo de aplicaciones con los microcontroladores AVR empleando lenguaje C, debe estar instalado el WinAVR y el AVR Studio.

A continuación se muestra el proceso a seguir para desarrollar y simular programas, suponiendo que ambas herramientas están instaladas. El proceso se ilustra con la solución del ejemplo 3.1 en lenguaje C.

1.- Al ejecutar el programa AVR Studio se abre la ventana mostrada en la figura C.1, en donde se da inicio a un nuevo proyecto presionando al botón *New Project.* Desde esta ventana también puede continuarse con un proyecto previo, la ventana muestra una lista con los proyectos recientes, basta seleccionar al proyecto y presionar al botón *Open*.

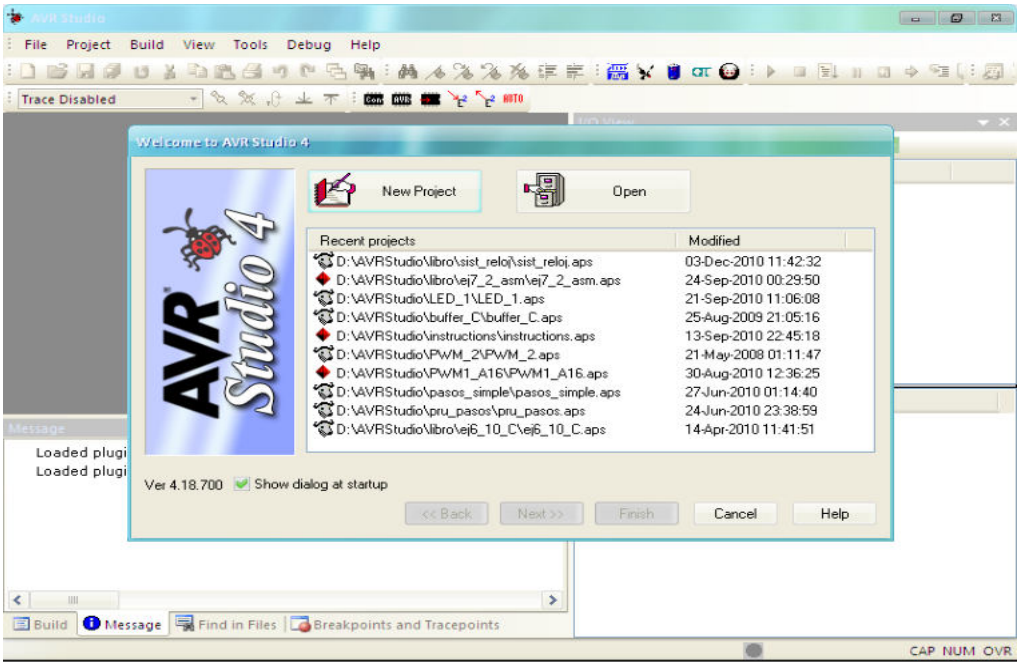


Figura C.1 Creación de un proyecto nuevo

2.- Después de presionar al boton *New Project* se abre la ventana de la figura C.2 en la que se define el nombre y el tipo del proyecto, el tipo depende del lenguaje a utilizar, es decir, determina si se va a programar en lenguaje C o en ensamblador. También debe definirse si se crea un archivo inicial y la carpeta del proyecto. Una vez que se han hecho estas definiciones debe presionarse el botón *Next*.

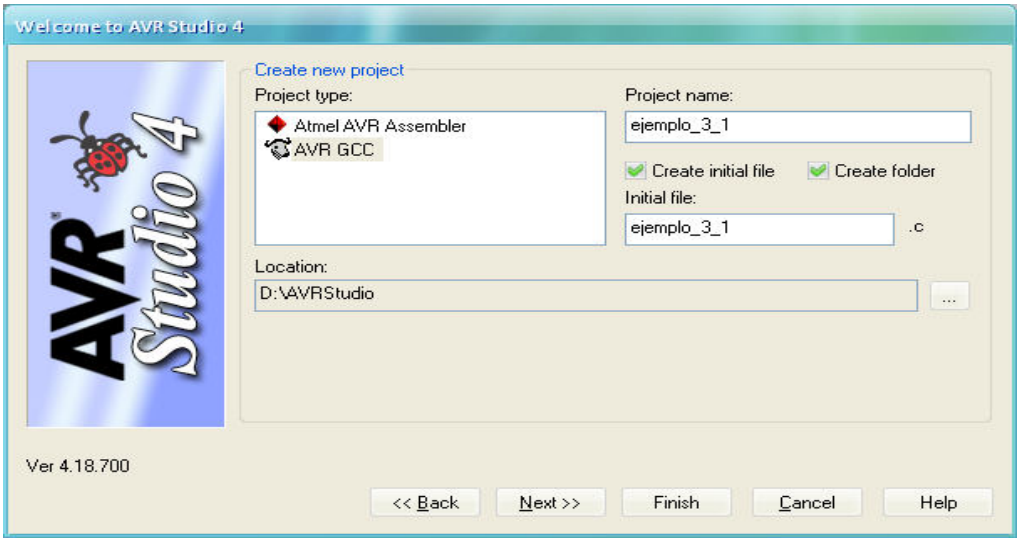


Figura C.2 Definición del nombre y tipo de proyecto

3.- Lo siguiente es la selección de la plataforma de depuración y el dispositivo al que se enfoca el proyecto. Si no se cuenta con alguna herramienta de hardware debe seleccionarse al simulador. En la figura C.3 se muestra esta selección y la del dispositivo ATMega8, en el cual se va a implementar el ejemplo 3.1.

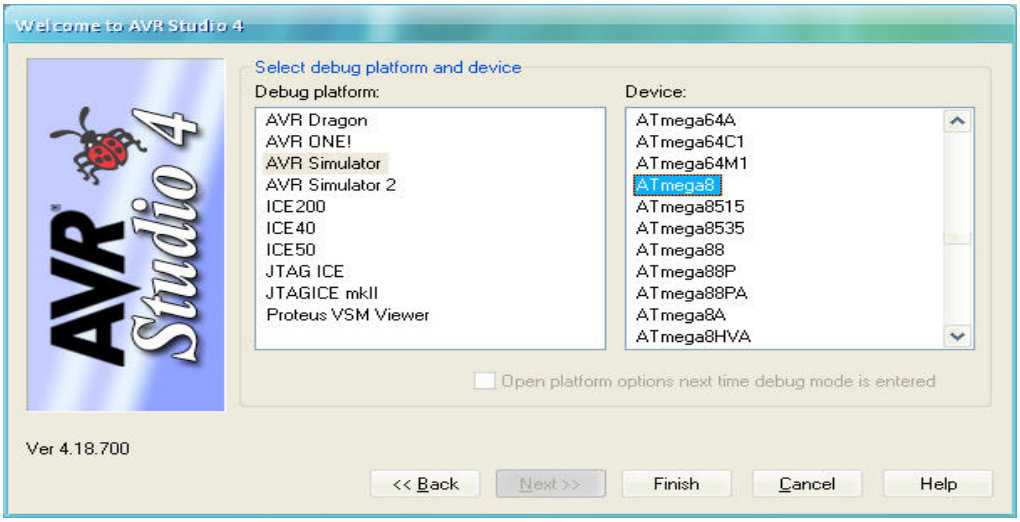


Figura C.3 Selección de la plataforma de depuración y del dispositivo

4.- Después de seleccionar estos parámetros debe presionarse al botón *Finish*, con ello, se cierra la ventana flotante de configuraciones, dejando sólo a la ventana principal del AVR Studio, con el aspecto mostrado en la figura C.4.

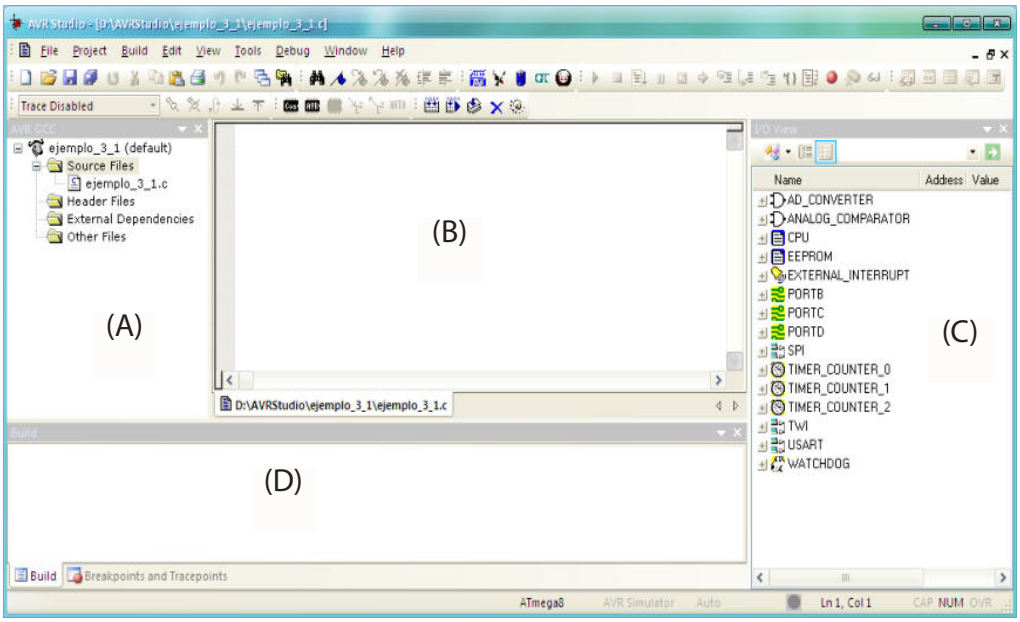


Figura C.4 Se distinguen 4 areas diferentes

A) Archivos del proyecto: Aquí se muestran los archivos de cabecera y el código fuente creado. Si ya se cuenta con alguna biblioteca de funciones y quiere agregarse al proyecto, debe darse un clic derecho en la carpeta de Archivos Fuente (*Source Files*) y en el menú contextual debe seleccionarse Agregar Arhivo(s) Fuente Existente(s) (*Add Existing Source File(s)*).

B) Espacio de trabajo: En esta área se escriben los programas y se puede observar la instrucción bajo ejecución, cuando se realiza una simulación o depuración.

C) Zona de recursos: En esta área se exhiben los recursos internos del MCU elegido, expandiendo cada recurso pueden observarse sus Registros I/O, para conocer su estado o realizar cambios durante la simulación de aplicaciones.

D) Notificación de mensajes: En esta área se muestran diferentes mensajes, si la compilación fue exitosa o si se generaron precauciones o errores. Otros mensajes están relacionados con el espacio utilizado después de la construcción del código binario e información sobre los puntos de evaluación (*breakpoints*) que se coloquen en el archivo, así como información sobre alguna búsqueda en los archivos.

5.- En el espacio de trabajo queda abierto el archivo fuente, para dar paso a la codificación del programa. En la figura C.5 se observa el código en lenguaje C del ejemplo 3.1.

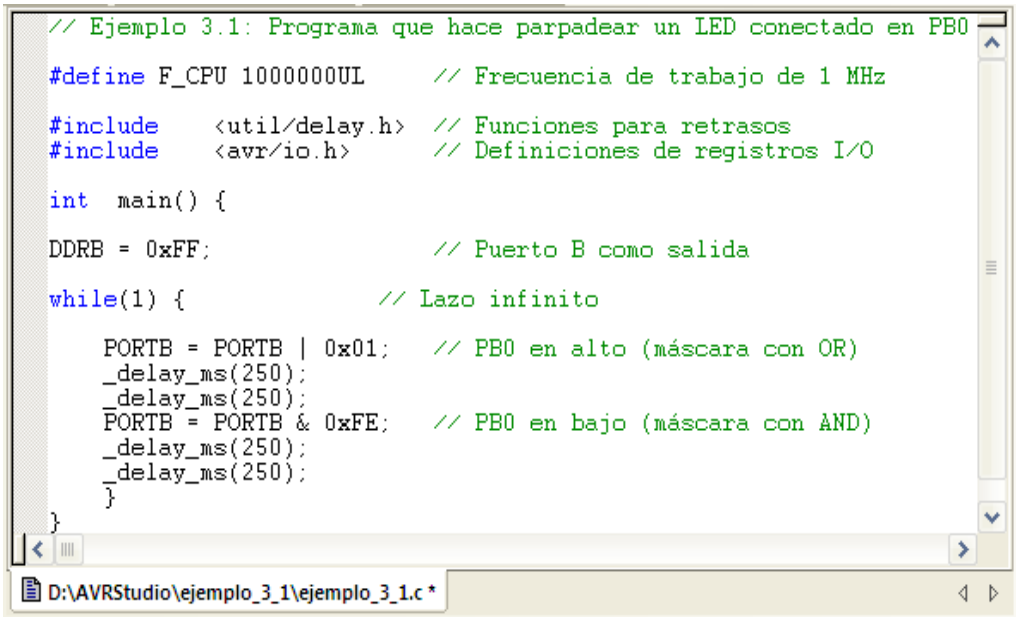


Figura C.5 Desarrollo del código fuente en el espacio de trabajo

6.- Una vez que se ha codificado el programa se procede con la construcción del archivobinario (archivo con extensión HEX), para ello se tiene la opción de Construir (*Build*) en el menú con el mismo nombre o con el botón de acceso rápido. Ambas alternativas se muestran en la figura C.6, el botón se distingue porque se ha encerrado con un círculo.

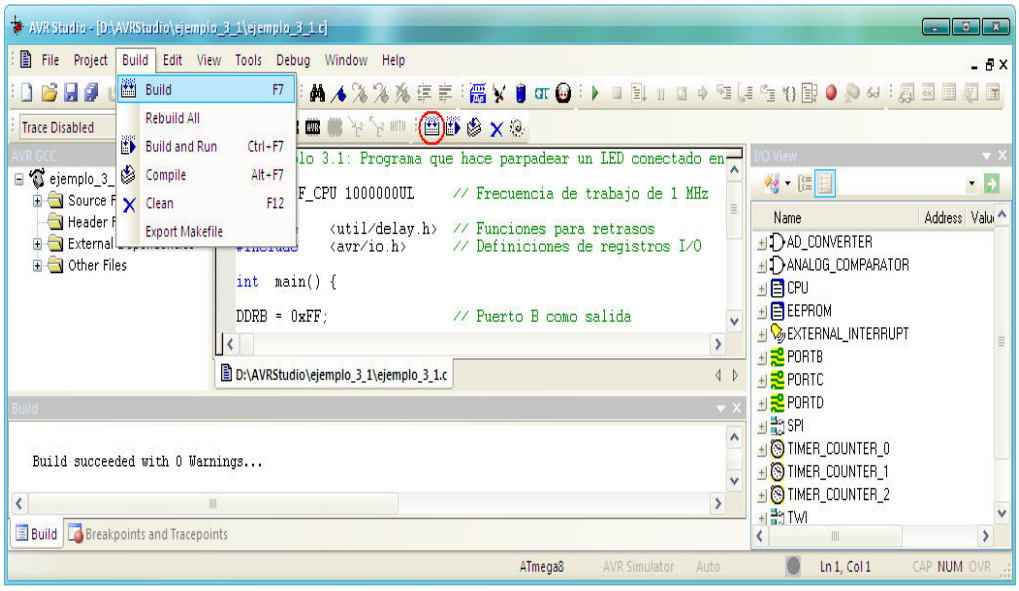


Figura C.6 Construcción del archivo binario

Se observa en la figura C.6 que el menú *Build* también tiene la opción para compilar un archivo (*Compile*). Ambas opciones generan el mismo resultado si el proyecto sólo tiene un archivo fuente. Si se incluyen varios archivos, con la opción *Compile* únicamente se compila al archivo activo y no a todos los incluidos en el proyecto.

En el área de notificación de mensajes se observa que el archivo binario fue construido con éxito, también se generan mensajes indicando la cantidad de memoria utilizada en Flash y en SRAM (no se alcanza a ver en la figura C.6). Esto significa que ya se tiene disponible al archivo con extensión HEX para ser descargado en el microcontrolador o para que sea empleado en algún otro simulador.

Si existen errores de sintaxis o algún otro impedimento para crear al archivo HEX, se muestran los mensajes correspondientes en el espacio de Notificaciones. Para los errores sintácticos se indica el número de renglón en donde se generó, un ejemplo de ello se muestra en la figura C.7, resultado de quitar un ";" en el código fuente.

```
../ejemplo_3_1.c:15: error: expected ';' before '_delay_ms'
make: *** [ejemplo_3_1.o] Error 1
Build failed with 1 errors and 0 warnings...
```

Figura C.7 Error de sintaxis

Con un doble clic en el renglón del error se señaliza en donde fue detectado, dentro del código fuente.

7.- Antes de descargar el archivo HEX en el microcontrolador es conveniente simular la aplicación. Para ello, se debe seleccionar la opción Iniciar Depuración (*Start Debugging*) en el menú Depurar (*Debug*). En la figura C.8 se ilustra esta opción y el acceso rápido con el botón encerrado en un círculo.

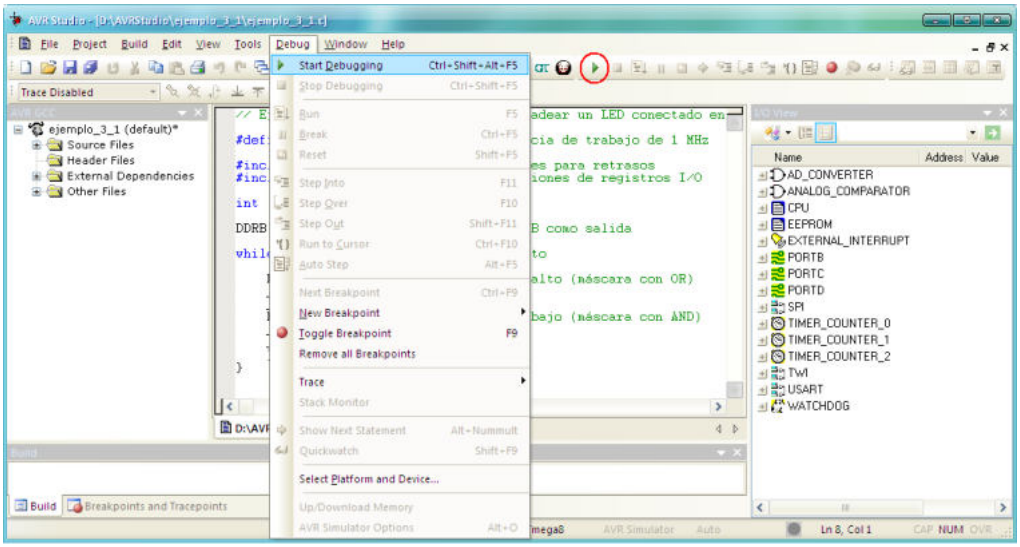


Figura C.8 Inicio de la simulación

8.- Una vez que la simulación ha iniciado, en el área Archivos del proyecto se abre una ceja denominada Procesador (*Processor*) en donde se pueden ver los registros fundamentales en la CPU del MCU: el contador del programa (*Program Counter*), el apuntador de la pila (*Stack Pointer*), los apuntadores de 16 bits, el registro de Estado (SREG) y los registros de propósito general (R0 a R31). El contenido de estos registros se actualiza conforme la simulación avanza. Esto se observa en la figura C.9, también puede verse como en el espacio de trabajo se ha señalizado la instrucción que está por ejecutarse.

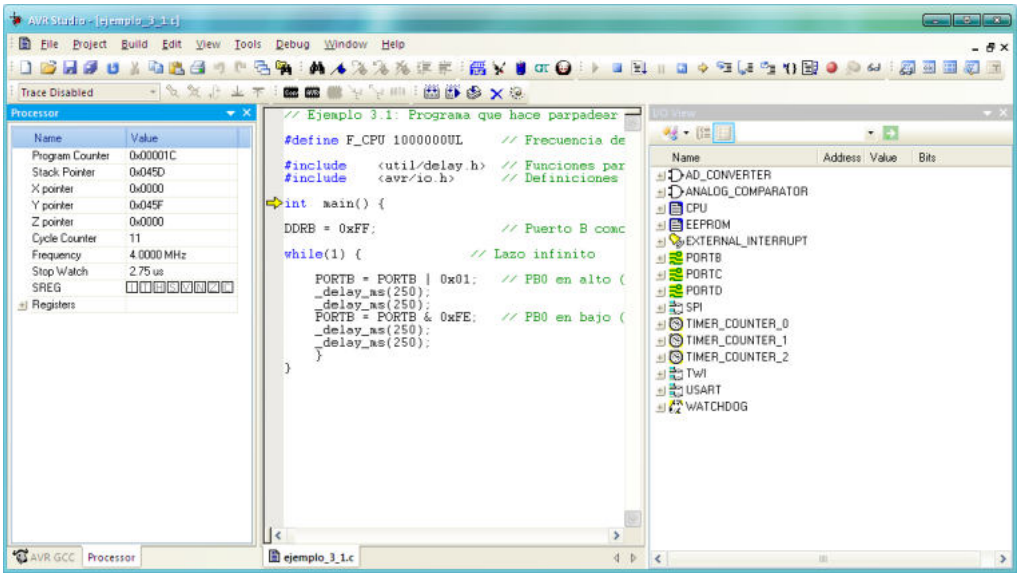


Figura C.9 Simulación puesta en marcha

En la figura C.10 se ha desplegado al menú *Debug* para mostrar las opciones disponibles durante la simulación. Con F5 la simulación se ejecuta (*Run*) hasta que se alcanza un punto de evaluación (*Breakpoint*) o mientras no se realice una pausa (*Break*) con Shift-F5. Con F10 y F11 se avanza una instrucción en la simulación, difieren en que con F11 se ingresa a las funciones y con F10 se evita el ingreso. Con Shift-F11 se realiza un paso hacia atrás en la simulación, pero no funciona adecuadamente en lenguaje C, sólo en ensamblador. Además de otras opciones, se puede ver que la simulación se detiene con Ctrl+Shift+F5 (*Stop Debugging*). En todos los casos se tienen botones de acceso rápido, a la izquierda de cada opción se ilustra el botón que le corresponde.

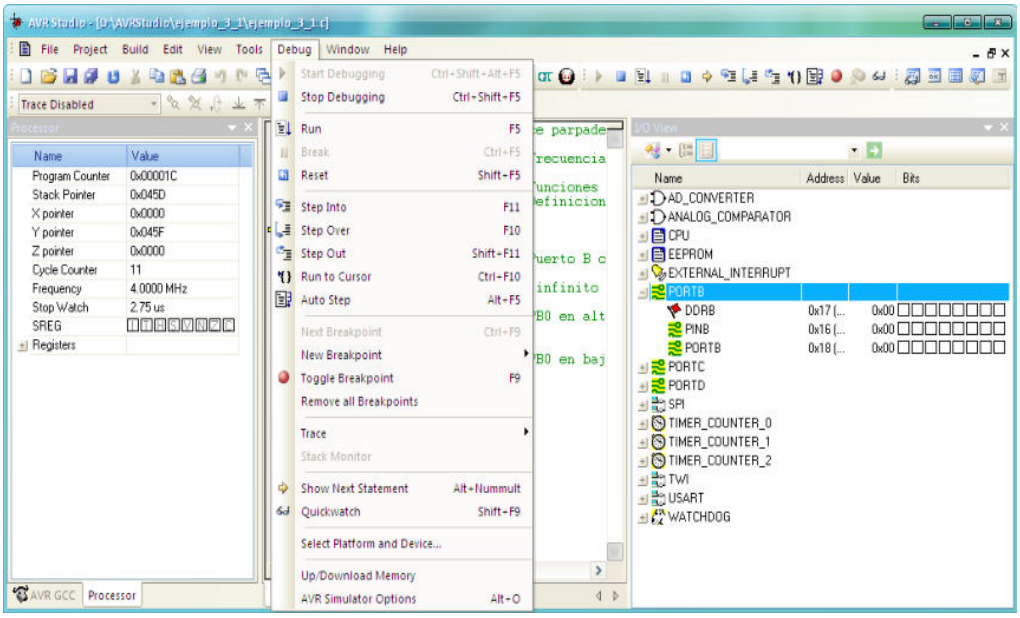


Figura C.10 Opciones para desarrollar la simulación

9.- Mientras la simulación está en proceso, en el espacio de Archivos del proyecto se muestran los cambios en el procesador, en el Espacio de trabajo se va avanzando en la señalización de las instrucciones y en la zona de recursos se actualizan los Registros I/O. Los Registros I/O también pueden modificarse manualmente, para la entrada de datos o para obligar a que ocurran diferentes situaciones. En la figura C.11 puede verse como ha avanzado la simulación.

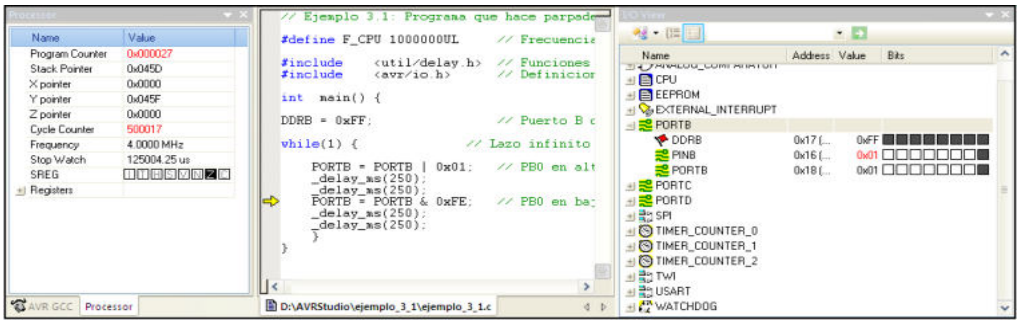


Figura C.11 Simulación en proceso

Si las herramientas se instalaron correctamente y el programa fue codificado respetando la sintaxis y las características del dispositivo, la compilación y simulación va a realizarse adecuadamente. Sin embargo, si es necesario modificar algún parámetro en el proyecto, se debe abrir la ventana con las opciones de configuración (*Configuration Option*) en el menú Proyecto (*Project*). En la figura C.12 puede verse la capacidad que presenta la herramienta para configurar diferentes parámetros del proyecto bajo desarrollo.

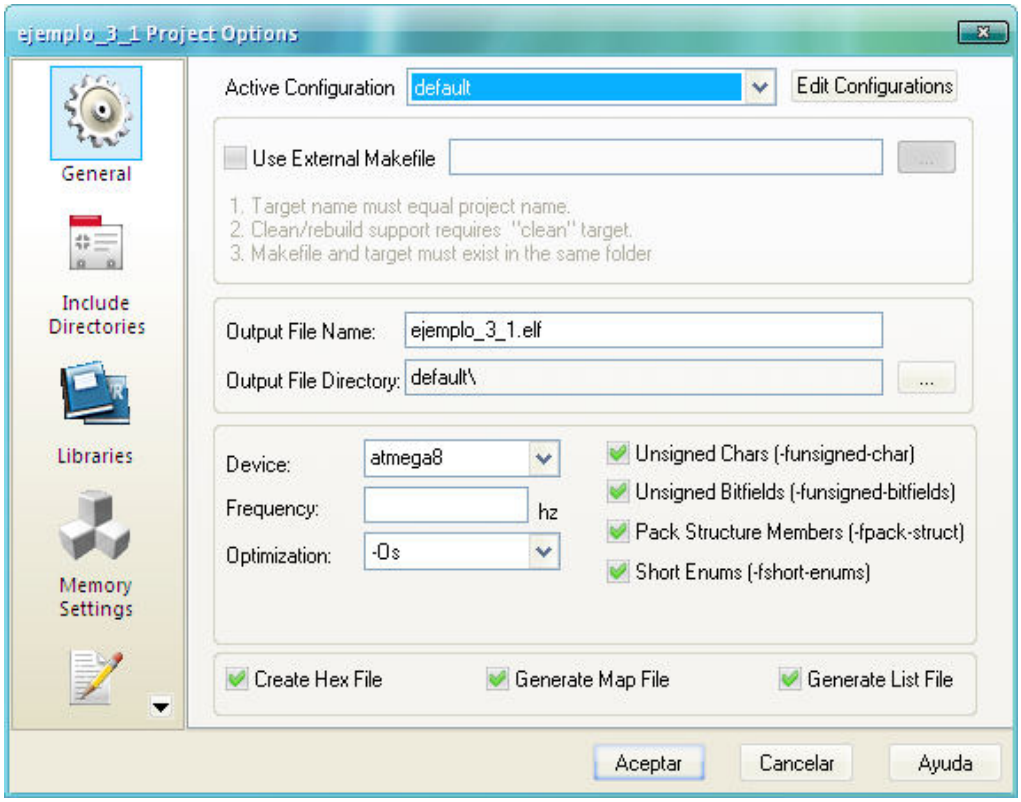


Figura C.12 Opciones de configuración en un proyecto bajo desarrollo

Todas las opciones de simulación también fueron probadas con el AVR Dragon, el cual es un depurador de bajo costo manufacturado y distribuido por Atmel. Aunque la depuración se realizó en un ATMega16, no es posible depurar un programa en un ATMega8 porque no cuenta con la interfaz JTAG o algún otro recurso que permita este proceso. La depuración implica ejecutar paso a paso el programa con el sistema conectado, para ir observando los resultados tanto en hardware como en software.

Apéndice D: Sitios Web de Referencia

En este apéndice se describen diferentes sitios de internet que contienen documentos complementarios para el manejo de los microcontroladores AVR, así como información sobre herramientas de hardware y software.

1.- http://www.atmel.com – Sitio web de la compañía Atmel, fabricante de los microcontroladores AVR. Del sitio pueden descargarse hojas de datos y notas de aplicación, documentos que fueron ampliamente consultados durante el desarrollo del texto. De este sitio también se obtuvo al software de desarrollo AVR Studio, el cual está disponible en forma gratuita.

En el sitio también hay información sobre diferentes herramientas para programación y depuración, como el STK500 o el AVR Dragon, éstas son las 2 herramientas de más bajo costo manufacturadas por Atmel. El STK500 es un kit de inicialización y desarrollo de sistemas con microcontroladores AVR. El AVR Dragon es un programador y depurador de MCUs de 8 y 32 bits, los MCUs deben incluir un depurador interno y su memoria de programa no debe exceder 32 Kbyte. Ambas herramientas son compatibles con el AVR Studio.

2.- http://winavr.sourceforge.net/ – Sitio web en donde se puede descargar al software WinAVR, una suite de herramientas de desarrollo basadas en código abierto para la serie de microcontroladores AVR de Atmel, enfocada a la plataforma Windows. Incluye al compilador GCC del proyecto GNU para C y C++.

3.- http://www.lancos.com/prog.html – Sitio de Claudio Lanconelli, programador experimentado de PCs y sistemas embebidos. Quien desarrolló al software **PonyProg,** un programador serial poderoso que permite descargar archivos binarios (código máquina) en memorias EEPROM y en una gama amplia de microcontroladores. Funciona satisfactoriamente con los MCUs ATMega8 y ATMega16, para ello debe desarrollarse un hardware simple. En las 2 referencias siguientes se encuentran versiones diferentes para este hardware, una con interfaz paralela y la otra con interfaz serial. El PonyProg está disponible para descargarse libremente del sitio.

4.- http://chaokhun.kmitl.ac.th/~kswichit/avr/avr.htm – Sitio de Wichit Sirichote en donde muestra un cable muy simple para descargar programas de una PC a un AVR. Este cable se conecta a una PC por medio de su puerto paralelo y al AVR con su interfaz SPI. Funciona satisfactoriamente con el programa PonyProg. Empleando este cable, con el AVR Studio se crea al programa HEX, se conecta al cable y se utiliza al PonyProg para descargar la aplicación.

5.- http://www.mecatronika.com/2009/04/programador-serial-para-avr/ – Publicación del Ing. Oscar Eduardo Enriquez Viveros en el sitio Mecatronika.com, en esta publicación describe un cable serial para descargar programas de una PC a un AVR. Utiliza el puerto serie de una PC y la interfaz SPI en los AVR. También funciona satisfactoriamente con el programa PonyProg. Empleando este cable, con el AVR Studio se crea al programa HEX, se conecta al cable y se utiliza al PonyProg para descargar la aplicación.

6.- http://www.fischl.de/usbasp/ – Sitio web en donde se describe cómo implementar al sistema **USBasp**, el cual es un programador USB para microcontroladores Atmel. Es un programador compacto soportado por un ATMega48 o un ATMega8 y algunos componentes pasivos, el firmware evita la necesidad de un CI controlador USB. En el sitio puede revisarse la organización del programador y descargarse el firmware para el ATMega48 o para el ATMega8, también está el driver para que el programador sea reconocido desde la PC.
Aunque el driver puede funcionar en diferentes plataformas (Linux, Mac OS X y Windows), sólo se ha probado bajo Windows. No funciona con PonyProg, se requiere de otros programas para realizar las descargas, sus referencias se describen en seguida.

7.- http://khazama.com/project/programmer/– Sitio del que puede obtenerse al software *Khazama AVR Programmer*, un programa simple, pequeño y rápido para realizar descargas en los microcontroladores AVR utilizando al programador **USBasp**. El programa presenta una GUI de fácil manejo y funciona satisfactoriamente bajo un entorno Windows.

8.- http://extremeelectronics.co.in/avr-tutorials/gui-software- for-usbasp-based-usb-avr-programmers/ – Sitio del que puede obtenerse al software *eXtreme Burner-AVR***,** otro programa compacto para realizar descargas en los microcontroladores AVR utilizando al programador **USBasp**. Su GUI es muy simple y de fácil manejo, funciona satisfactoriamente bajo un entorno Windows y también hay una versión para Linux.

Índice Temático

PUBLICACIONES DEL SUNEO

Universidad Tecnológica de la Mixteca

Las Rutas de la Tierra del Sol.
Ortiz Escamilla, Reina (compiladora)
2012. 343 págs.

Los Microcontroladores AVR de ATMEL.
Santiago Espinosa, Felipe
1ª Edición,
2012. 385 págs.

Diccionario del Idioma Mixteco
Caballero Morales, Gabriel
2ª. Edición,
2011. 899 págs.

Miradas al Mundo Mixteco
Ortiz Escamilla, Reina (compiladora)
2011. 245 págs.

La investigación científica en el Sistema de Universidades Estatales de Oaxaca
Seara Vázquez, Modesto (director de la obra)
2010. 230 págs.

A new Model of University
Seara Vázquez, Modesto
2010. 280 págs.

Un nuevo Modelo de Universidad
Seara Vázquez, Modesto
2ª. Edición,
2010. 216 págs.

Tres mixtecas. Una sola alma
Ortiz Escamilla, Reina (compiladora)
2010. 199 págs.

El significado de los sueños y otros Temas Mixtecos
Ortiz Escamilla, Reina (editor)
2009. 190 págs.

Caminos de la Historia Mixteca
Ortiz Escamilla, Reina (editor)
2008. 190 págs.

Agua el Líquido de la Vida
Cuaderno de divulgación técnica y científica No. 2
Alvarez Olguín, Gabriela et al.
2008. 54 págs.

El Secreto del Espectro
Cuaderno de divulgación técnica y científica No. 1
Vázquez de la Cerda, Alberto Mariano (editor)
2008. 62 págs.

Presencias de la Cultura Mixteca
López García, Ubaldo et al.
3ª. Impresión,
2008, 111 págs.

Raíces Mixtecas
Gallegos Ruiz, Roberto et al.
2006. 321 págs.

Ñuu Savi. La Patria Mixteca
Ruiz Ortiz, Víctor Hugo et al.
2006. 227 págs.

Pasado y Presente de la Cultura Mixteca
Ojeda Díaz, Ma. de los Ángeles et al.
2005. 321 págs.

Personajes e Instituciones del Pueblo Mixteco
Rivera Guzmán, Angel Iván et al.
2004. 357 págs.

A new charter for the United Nations
Seara Vázquez, Modesto
2003. 357 págs.

La Tierra del Sol y de la Lluvia
Galindo Trejo, Jesús et al.
2002. 211 págs.

Aplicación de un modelo de Balances Hídricos en la Cuenca del Río Mixteco
Blanco Andray, Alfredo y Martínez Ramírez, Saúl
2001. 250 págs.

La Vivienda Tradicional en la Mixteca Oaxaqueña
Fuentes Ibarra, Luis Guillermo
2000. 95 págs.

El Agua Recurso Vital
Arias Chávez, José et al.
1993. 147 págs.

Una Nueva Carta de las Naciones Unidas
Seara Vázquez, Modesto
1993. 196 págs.

Cuaderno Estadístico Municipal
INEGI-UTM
1993. 113 págs.

Inteligencia Artificial
Galindo Soria, Fernando et al.
1992. 178 págs.

Electrónica y Computación en México.
Gil Mendieta, Jorge (comp.)
1992. 118 págs.

Revista: Temas de Ciencia y TecnologÌa
Publicación cuatrimestral. 1997 a la fecha.
36 números. 88 págs. Cada una

Universidad del Mar

La Sociedad Internacional Amorfa
Seara Vázquez, Modesto (coordinador)
2011, 654 págs.

La iguana negra
Fundamentos de reproducción y su manejo en cautiverio
Arcos García, José Luis y López Pozos, Roberto
2009, 70 págs.

Diagnóstico de los Recursos Naturales de la Bahía y Micro-cuencas de Cacaluta
Domínguez Licona, Juan Manuel (editor)
2008. 453 págs.

Rusia hacia la cuenca del Pacífico
Roldán, Eduardo (editor)
2008. 355 págs.

Estudio de Ordenamiento Ecológico para la Zona Costera del Istmo de Tehuantepec
Serrano Guzmán, Saúl J.
2004. 159 págs.

Mujeres Empresarias y Turismo en la Costa Oaxaqueña Informe Diagnóstico y Directorio
Fernández Aldecua, María et al.
2001. 81 págs.

Biología y Aprovechamiento del Camarón Duende
Castrejón Ocampo, Laura et al.
1993. 72 págs.

Diagramas Prácticos para la Acuacultura
Porras Díaz, Demetrio y Castrejón Ocampo, Laura
1993. 111 págs.

Revista: Ciencia y Mar
Publicación cuatrimestral. 1997 a la fecha
34 números. 86 págs. Cada una.

Universidad del Istmo

La Cultura Zapoteca. Una cultura viva
Acevedo Conde, María Luisa et al.
2009. 248 págs.

Secretos del Mundo Zapoteca
Méndez Martínez, Enrique et al.
2008. 321 págs.

Un recorrido por el Istmo
Ramírez Gasga, Eva (editor)
2006. 224 págs.

Etnobiología Zapoteca
Smith Stark, Tomas C. et al.
2005. 293 págs.

Palabras de luz, palabras floridas
Winter, Marcus et al.
2004. 139 págs.

Venta en las librerías del SUNEO y principales librerías del país.

Informes:

Vicerectoría de Relaciones y Recursos
Pino Suarez No. 509. Centro. C.P. 68000, Oaxaca, Oax.
Tel. 01 951 13 269 58 y 13 253 30

Sacramento No. 347
Col. Del Valle. C.P. 3100, México. D.F.
01 55 55 75 13 65
01 55 55 75 15 89

Universidad Tecnológica de la Mixteca
www.utm.mx.

Universidad del Mar
www.umar.mx

Universidad del Istmo
www.unistmo.edu.mx

Universidad del Papaloapan
www.unpa.edu.mx

Universidad de la Sierra Sur
www.unsis.edu.mx

Universidad de la Sierra Juárez
www.unsij.edu.mx

Universidad de la Cañada
www.unca.edu.mx

Nova Universitas
www.novauniversitas.edu.mx

Los Microcontroladores AVR de ATMEL
Se terminó de imprimir
Mayo de 2012 en los talleres

1000 ejemplares