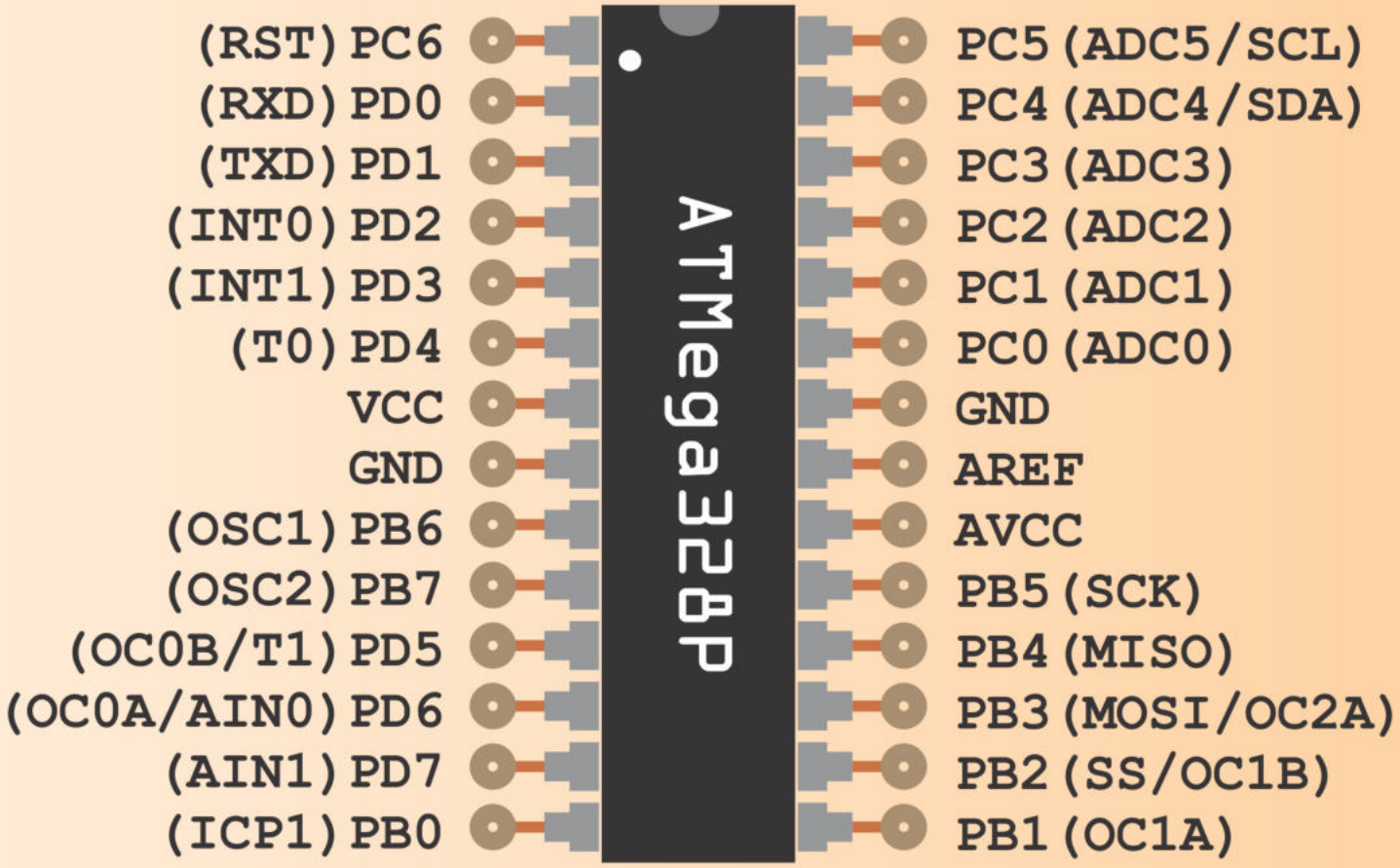
El Microcontrolador ATMega328P de Microchip:

Programación en Ensamblador, Lenguaje C y un enlace con Arduino



Felipe Santiago Espinosa

UNIVERSIDAD TECNOLÓGICA DE LA MIXTECA

El Microcontrolador ATMega328P de Microchip:

Programación en Ensamblador, Lenguaje C y un enlace con Arduino

Felipe Santiago Espinosa

ISBN: 978-607-98020-9-7

Universidad Tecnológica de la Mixteca

El Microcontrolador ATMega328P de Microchip:
Programación en Ensamblador, Lenguaje C y un enlace con Arduino
Felipe Santiago Espinosa

Universidad Tecnológica de la Mixteca
Km. 2.5 Carretera a Acatlima
Huajuapan de León, Oaxaca
México

1^a Edición, 19 de noviembre de 2021
ISBN: 978-607-98020-9-7

Índice general

Prólogo

En el año 2012 la Universidad Tecnológica de la Mixteca me publicó el libro **Los Microcontroladores AVR de ATMEL**, inicio el presente libro compartiendo su prólogo porque describe la motivación que me llevó a su escritura y este aspecto no ha cambiado. Posteriormente complemento el prólogo haciendo referencia a la nueva versión, enfocada al Microcontrolador **ATMega328P**.

Prólogo: Los Microcontroladores AVR de ATMEL

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 lo trato 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 como los diferentes recursos del MCU pueden ser manejados en Ensamblador o en Lenguaje C. Este 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 esta 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.

Prólogo a la versión ATMega328P

Actuamente la empresa ATMEL es parte de Microchip, otro fabricante de microcontroladores y dispositivos electrónicos, quien clasifica su gama de microcontroladores de 8 bits en dos familias: Los PIC y los AVR.

Un par de años posteriores a la publicación del primer libro se popularizó el uso de la tarjeta Arduino, una plataforma que facilita el diseño de sistemas con el mínimo de

conocimientos sobre los microcontroladores. La tarjeta Arduino inició soportada por un ATMega8, pero pasó poco tiempo para que migrara al ATMega328P, por tener más memoria y recursos adicionales que aumentan su capacidad de procesamiento.

El ATMega328P tiene la misma disposición de pines que el ATMega8, las diferencias principales entre ellos son: incluye 32 kbytes para memoria de código, todas las terminales pueden generar interrupciones y los tres temporizadores tienen dos registros de comparación, pudiendo generar respuesta automática en seis terminales diferentes, lo que significa que un ATMega328P puede generar hasta seis señales PWM en forma simultánea.

Por ello, desde el año 2016 utilizó al ATMega328P para mis cursos de Microcontroladores. El incremento en recursos hizo necesaria la inclusión de nuevos registros o el cambio en la estructura de los existentes, por lo tanto, un programa realizado para un ATMega8 puede no funcionar en un ATMega328P. De manera que fue necesario actualizar el libro de texto y el resultado ahora ha llegado a tus manos.

Si bien es cierto que la tarjeta Arduino facilita el desarrollo de aplicaciones, en la medida en que los desarrolladores avanzan sus diseños tienen requerimientos especiales, como el uso simultáneo de recursos atendidos por interrupciones o configuraciones distintas a las establecidas por el entorno, por ejemplo, la generación de una señal PWM con una frecuencia variable. Para esos casos, se debe entender la organización del microcontrolador y hacer un manejo directo de los registros de configuración, sin la envoltura que proporciona Arduino.

La buena noticia es que el código C desarrollado para un ATMega328P puede incluirse dentro de un *Sketch* de Arduino y se ejecutará sin inconveniente. Para algunos ejemplos del texto se muestra cómo se puede ralizar este híbrido, introduciendo código C para simplificar o personalizar el manejo de recursos.

Nuevamente agradezco a la Universidad Tecnológica de la Mixteca porque sigo perteneciendo a esta gran institución y a todas las personas que se tomaron un minuto para felicitarme por la escritura del primer libro, con mucho agrado y satisfacción vi que se utilizó en diferentes instituciones de México y que llegó a paises hermanos, como Perú, Chile, Colombia, Cuba y República Dominicana. Esa fue la motivación para la adaptación del libro al microcontrolador ATMega328P, fue satisfactorio revisar el correo electrónico y encontrarme con la noticia de que hubo alguien en algún lugar distante a quien le sirvió el libro, espero ocurra lo mismo con este nuevo material y que sigan llegando los correos.

Felipe Santiago Espinosa
fsantiag@mixteco.utm.mx

Capítulo 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 fue diseñado. 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, esta información le permite al usuario tomar decisiones. Algunos elementos de visualización son: LEDs individuales o matrices de LEDs, *displays* de 7 segmentos o de cristal líquido.

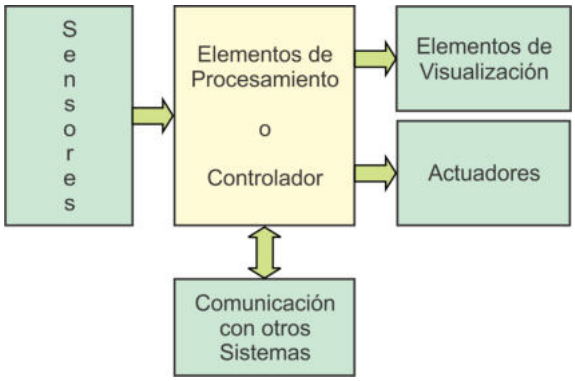


Figura 1.1: Abstracción de un sistema electrónico

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 enlazarse con otros, son necesarios cuando varios sistemas deben coordinarse para realizar una tarea compleja. Cada sistema tiene sus elementos de procesamiento y está orientado a resolver una etapa específica, es a través de protocolos predefinidos como se establece la comunicación entre sistemas.

Los elementos de procesamiento son un conjunto de dispositivos electrónicos que determinan la funcionalidad del sistema con el desarrollo de uno o varios procesos, también se les refiere como el **Controlador** o la **Tarjeta de Control** del sistema. El **Controlador** recibe la información proveniente de los sensores, evalúa el estado actual del sistema y genera 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 cambiado con los avances tecnológicos. Los controladores de los primeros sistemas electrónicos 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 de baja o mediana escala de integración.

El primer microprocesador (4004 de Intel) fue puesto en operación en 1971, esto permitió el desarrollo de las tarjetas de control con un microprocesador y sus elementos de soporte (memoria, entrada/salida, etc.). A estas tarjetas también se les conoce como **Computadoras en una Sola Tarjeta** (SBC, *single board computer*).

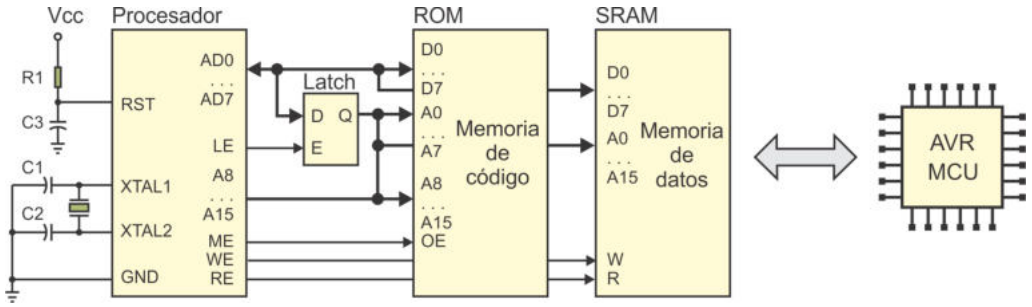


Figura 1.2: Los microcontroladores remplazan tarjetas con varios CI

Actualmente se han integrado todos estos elementos en un circuito integrado único y a este se le refiere como **Unidad Micro Controladora** (MCU, *Micro Controller Unit*) o simplemente como **Microcontrolador**, esta integración se ilustra en la Figura 1.2.

Un **Microcontrolador** es un circuito con una escala de integración muy grande (VLSI[1], *very large scale integration*) que internamente contiene una Unidad Central de Procesamiento (CPU, *Central 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.

A pesar de que un MCU incluye los elementos necesarios para ser considerado como una computadora en un circuito integrado, frecuentemente no es tratado como tal porque 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, manejador de 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 de gama baja trabajan a frecuencias cercanas a 20 MHz, mientras que los microprocesadores están trabajando en el orden de GHz.

[1]Circuitos integrados con más de 10,000 transistores

- **Capacidad de direccionamiento:** Un microcontrolador promedio dispone de 8 Kbytes para instrucciones y 1 Kbyte para datos, en contraste con el espacio que pueden direccionar los microprocesadores modernos, este está en el orden de Terabytes y es compartido por instrucciones y datos. Por ello, el repertorio de instrucciones de los microprocesadores debe incluir modos de direccionamiento que les permitan este alcance.

- **Tamaño de los datos:** Los microcontroladores populares son de 8 bits y dentro de su repertorio incluyen instrucciones para evaluar o modificar bits individuales. Los microprocesadores actuales trabajan con datos de 32 o 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 generalmente se utiliza como la CPU de una computadora, que es un **Sistema de Propósito General** capaz de realizar cualquier tarea que se le solicite por programación. Su capacidad de procesamiento le posibilita una operación multitarea, es decir, una computadora puede mantener varios procesos en ejecución al mismo tiempo. Los microprocesadores también se pueden encontrar en otros sistemas de procesamiento intensivo como consolas de video juegos o teléfonos inteligentes.

Los microcontroladores están enfocados a **Sistemas de Propósito Específico**, también conocidos como **Sistemas Embebidos**, son sistemas que se crean con una funcionalidad única que no va a cambiar durante su tiempo de vida útil. Por ejemplo: cajas registradoras, hornos de microondas, sistemas de control de tráfico, 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 de tiempo en el orden de milisegundos, periodos que pueden conseguirse con un microcontrolador operando a unos cuantos megahercios.

- Solo realizan una tarea, esto 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, que son fundamentales en sistemas de propósito general. Por lo tanto, la cantidad 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 kbytes, 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 las aplicaciones utilizan datos de 16 bits, por ello, algunos microcontroladores incluyen instrucciones que operan directamente sobre datos de 16 bits, o en caso de ser necesario, puede elegirse 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. Las aplicaciones con salidas analógicas también son realizables porque los microcontroladores pueden generar señales moduladas 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. Los recursos incluidos en un MCU son suficientes para aplicaciones que no demanden un alto rendimiento y que no requieran manejar un conjunto masivo de datos, esto significa que aplicaciones como el procesamiento de video o imágenes, quedan fuera del alcance de un microcontrolador.

1.4. Microcontroladores y FPGAs

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 las conexiones entre CLBs o de CLBs con IOBs.

Los CLBs sirven para programar funciones lógicas combinacionales o secuenciales, con los recursos de interconexión se pueden enlazar diferentes bloques para construir funciones complejas. Dependiendo del fabricante, un CLB va a contener una o más tablas de búsqueda (LUT, *look-up table*), elementos de estado y hardware para acarreo rápido. La LUT es una pequeña memoria para almacenar una función lógica booleana de 4 o 6 entradas y una salida. Los IOBs proporcionan el mecanismo para que el FPGA se comunique con su entorno, las terminales se configuran por medio de los IOBs como entradas, salidas o bidireccionales.

Los FPGAs se pueden programar con diagramas esquemáticos, utilizando símbolos básicos y conexiones entre ellos. 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.

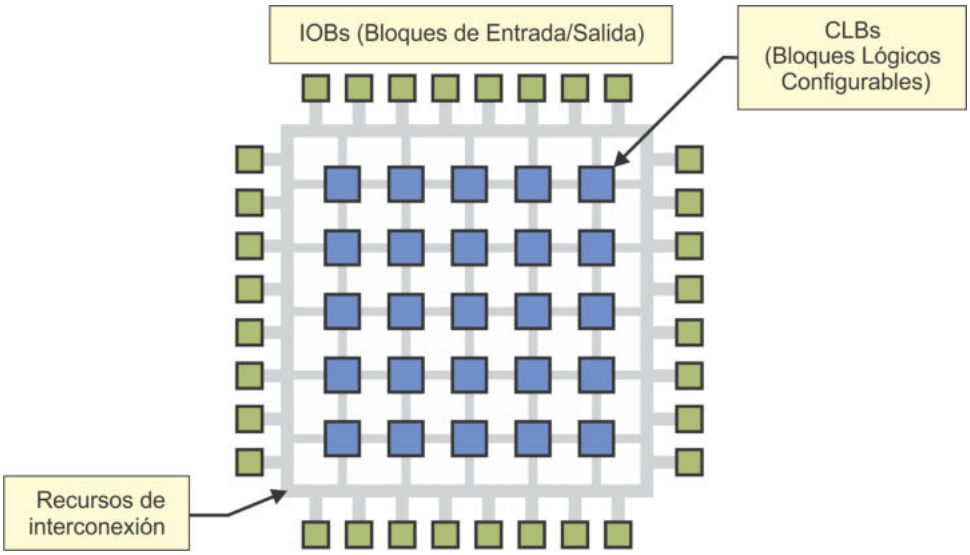


Figura 1.3: Organización típica de un FPGA

Un aspecto común entre un FPGA y un MCU es que ambos son dispositivos configurables, con ambos se construyen sistemas flexibles porque la reprogramación del dispositivo modifica el comportamiento del sistema. Pero el programa tiene un objetivo diferente en cada caso, en un FPGA el programa determina qué funciones se van a generar en las LUTs y cómo se van a conectar sus elementos internos, es decir, el programa define al hardware y de esta manera determina el comportamiento de un sistema. En cambio, en un MCU el hardware es fijo y el programa sólo establece el comportamiento 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 un recurso genera un evento que requiere atención de la CPU, la atención se realiza mediante líneas de código secuenciales.

En forma práctica, primero se debería considerar emplear un MCU como el controlador de un sistema electrónico, si se requiere de más velocidad o capacidad de direccionamiento, las alternativas son: un MCU de gama alta o un microprocesador con sus elementos de soporte. Si el sistema 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 similar a un MCU porque contiene una CPU, memorias y recursos adicionales, la diferencia entre un MCU y un DSP es que el DSP incorpora elementos de hardware enfocados a operaciones aritméticas, como sumadores y multiplicadores, generalmente de punto flotante. Un FPGA es la última opción y es la mejor para aquellos casos en donde se requiere de un hardware especializado, a la medida del sistema, que trabaje a altas velocidades y con módulos concurrentes.

1.5. Organización de los Microcontroladores

Existe un número grande de fabricantes de microcontroladores y todos manejan diferentes familias con una variedad de modelos, a pesar de ello, hay bloques que son comunes a la mayoría de dispositivos, en la Figura 1.4 se muestra la organización típica de un microcontrolador y en los siguientes apartados se describen sus bloques internos.

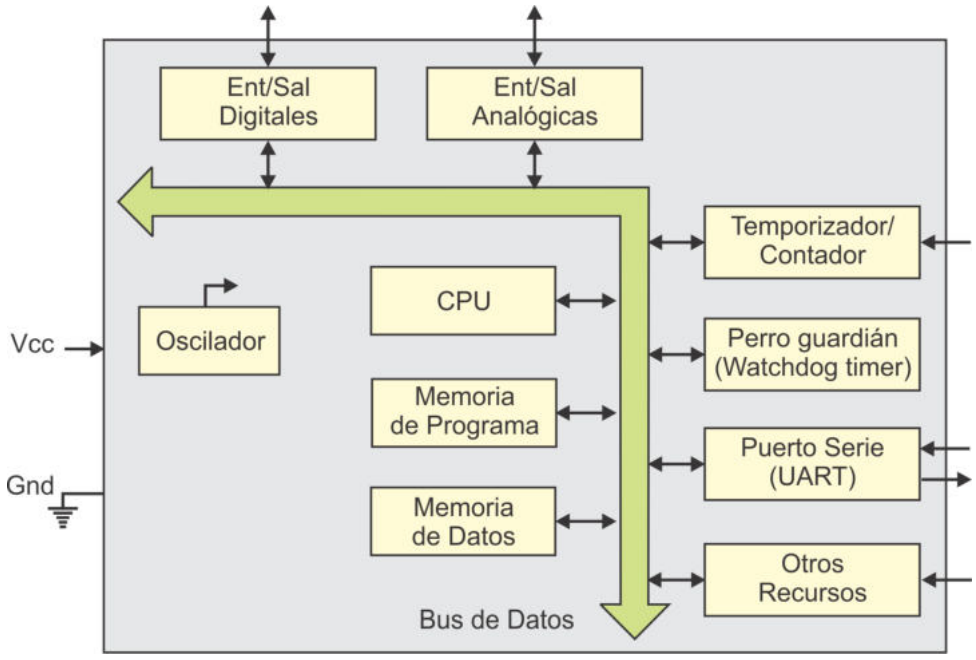


Figura 1.4: Organización típica de un microcontrolador

1.5.1. La Unidad Central de Procesamiento (CPU)

La CPU es el bloque principal del microcontrolador, administra todas las actividades que ocurren en el chip y realiza las operaciones sobre los datos, esto mediante la ejecución del programa ubicado en la memoria de código, con ello se determina el comportamiento del sistema controlado por el MCU.

Un programa es una serie de instrucciones combinada de manera lógica para realizar un trabajo específico. Aunque el hardware y el software se complementan, el grado en que un sistema trabaja de manera correcta y eficiente depende muchas veces de la estructura del programa y no de qué tan sofisticada es la CPU.

Para cada instrucción de un programa, el trabajo de la CPU se organiza en tres etapas fundamentales:

1. Captura de la Instrucción.
2. Decodificación de la misma.
3. 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 pero sus elementos difieren en recursos, costo y rendimiento, entonces este grupo forma una familia de computadoras o de microcontroladores.

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

1. **Aritméticas:** suma, resta, producto, división, etc.
2. **Lógicas:** AND, OR, XOR, NOT, etc.
3. **Transferencias de datos:** directas o indirectas.
4. **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 programa 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, puede notarse que básicamente son transferencias de datos con un nulo procesamiento aritmético o lógico.

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. El **código de operación** u **opcode** es un campo presente en todas las instrucciones porque determina la operación a realizar. Las instrucciones también

deben incluir campos para los operandos, estos indican la ubicación de los datos a los que se aplica la operación.

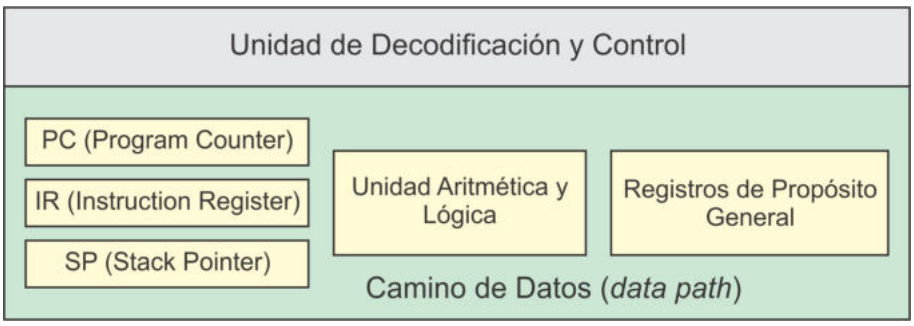


Figura 1.5: Elementos comunes en una CPU

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 por donde va a fluir la información cuando se ejecute una instrucción y la **Unidad de Decodificación y Control** determina qué elementos se activan dentro del **Camino de Datos** para la correcta ejecución de cada instrucción.

Como parte del camino de datos, en la Figura 1.5 se observan tres registros con una función específica:

- El **Contador de Programa** (PC, *Program Counter*) 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 de Instrucción** (IR, *Instruction Register*) contiene la cadena de bits que conforman a la instrucción bajo ejecución, de esa cadena, la unidad de control analiza al campo del opcode para determinar la activación de las señales en los demás elementos en la CPU.
- El **Apuntador de Pila** (SP, *Stack Pointer*) 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 se remplaza 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**.

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 no es suficiente para todas las variables requeridas por un programa, debe utilizarse la memoria de datos para su almacenamiento.

La Unidad Aritmética y Lógica es el bloque que se encarga de realizar las operaciones aritméticas y lógicas con los datos guardados en los registros de propósito general, 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.

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:

1. El contenido del **PC** se coloca en el bus de direcciones.
2. La CPU genera una señal de control para habilitar la lectura de memoria de código.
3. Una instrucción se obtiene de la memoria de código y se coloca en el **IR**.
4. El **PC** es preparado para la siguiente instrucción.

Una vez que la instrucción está en el **IR** el procesador continúa con su decodificación. Decodificar consiste en descifrar el opcode para generar las señales de control adecuadas para el tipo de instrucción.

Finalmente, la tercer tarea 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 y llevarlo a un registro, almacenar el contenido de un registro en memoria o modificar el valor del **PC** para un salto. La unidad de decodificación y control genera las señales requeridas por cada instrucción.

Tipos de CPU

De acuerdo con su organización interna una CPU puede ser **CISC** o **RISC**. 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*).

En el diseño de las primeras computadoras se buscó que el programador escribiera programas compactos, para ello, las instrucciones deben realizar tareas complejas y, por lo tanto, requieren de un hardware extenso para su ejecución. Esto afecta el rendimiento de las computadoras porque significa que el ciclo de reloj debe ser largo o que se van a necesitar varios ciclos de reloj para la ejecución de cada instrucción. La complejidad en el hardware para ejecutar este tipo de instrucciones hace que estas computadoras correspondan a la categoría **CISC**.

La filosofía **RISC** es opuesta, para aumentar el rendimiento los diseños se orientaron a instrucciones con tareas muy simples, esto hace posible que el hardware trabaje a frecuencias mucho más grandes.

Una arquitectura **RISC** tiene pocas instrucciones y generalmente son del mismo tamaño; en una arquitectura **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** generalmente requiere de varias instrucciones **RISC**, sin embargo, la simplicidad en las tareas de las instrucciones **RISC** permite que en la organización del procesador se apliquen técnicas como la segmentación[2], que consiste en el traslape de diferentes instrucciones en cada una de las 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.

Además, el hardware de un procesador **RISC** es tan simple que actualmente es frecuente encontrar varios núcleos en un circuito integrado.

Es por ello que desde los 90's prácticamente todos los nuevos procesadores son **RISC** y los que originalmente fueron **CISC** se han convertido en un híbrido que acepta instrucciones complejas e internamente las separa en microinstrucciones o instrucciones simples para ser procesadas en flujos de ejecución del tipo **RISC**.

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, estos se ilustran en la Figura 1.6.

[2]La segmentación proporciona un paralelismo al nivel de instrucciones que queda transparente al programador.

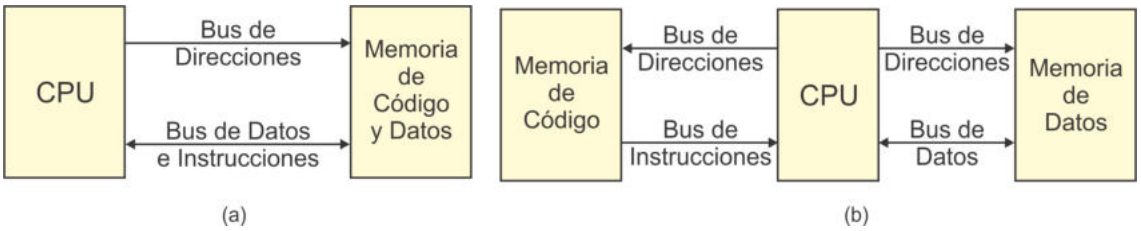


Figura 1.6: Modelos de computadoras respecto a la organización de la memoria (a) Arquitectura von Neumann y (b) Arquitectura Harvard

John von Neumann[3] 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 para los 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.

La memoria de programa generalmente es no volátil y puede 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 pocas unidades de un sistema y la ROM enmascarable es la más acertada para una producción masiva.

[3]**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.

Con respecto a la memoria de datos, los microcontroladores contienen un espacio de SRAM y algunos también incluyen EEPROM. La SRAM se utiliza para el almacenamiento de variables y el manejo de una pila para organizar las llamadas a rutinas. La EEPROM sirve para almacenar aquellos datos que se deben conservar aun en ausencia de energía. Todos los microcontroladores tienen una SRAM interna de diferentes magnitudes, algunos además cuentan con la capacidad de expansión por medio de 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 microcontroladores incluyen un oscilador RC calibrado interno para reducir el tamaño del sistema final. Tan pronto como se suministra la alimentación eléctrica a un MCU, 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 coordinar acciones periódicas y como contador sirve para que una tarea se realice cuando ocurre una cantidad predeterminada de eventos externos.

Se compone de un registro de n-bits que se incrementa en cada ciclo de reloj o cuando ocurre un evento externo, según el modo de operación. Cuando el registro desborda se genera una señalización, es decir, se pone 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 la cuenta se reinicia (todos los bits en 0). 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. Mientras el temporizador trabaja, 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.

En algunos microcontroladores, la entrada del temporizador es precedida por un preescalador, 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.

Los temporizadores en los MCUs modernos tienen funciones adicionales, incluyen

registros de comparación para que ante coincidencias con el temporizador se puedan producir respuestas automáticas en algunas de sus terminales, de esta forma se pueden generar tonos a determinadas frecuencias o señales moduladas en ancho de pulso (PWM, *pulse width modulation*). PWM es una técnica para manipular dispositivos analógicos desde salidas digitales, al modificar el ciclo de trabajo de una señal periódica.

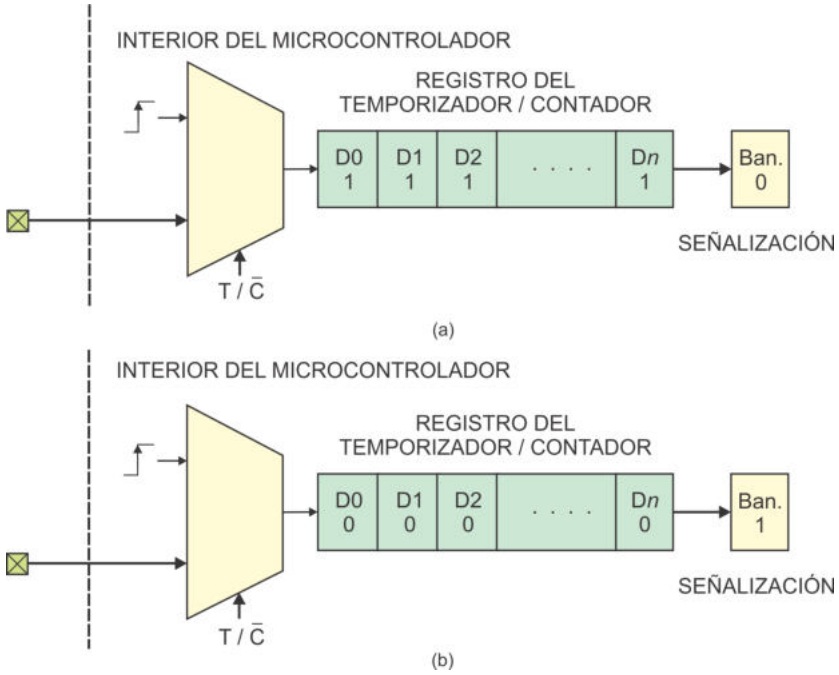


Figura 1.7: Organización básica de un Temporizador/Contador, en (a) el registro ha alcanzado su valor máximo y en (b) al reiniciar la cuenta se genera una señalización

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, solo que cuando 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 muy útil en sistemas autónomos, el microcontrolador se puede ciclar ante situaciones inesperadas, como variaciones en la fuente de alimentación o por la desconexión repentina de un periférico.

En algunos microcontroladores el WDT se debe activar 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. En cualquier caso, si se utiliza al WDT, en posiciones estratégicas del programa principal deben incluirse instrucciones para su reinicio y con ello se evita su desbordamiento.

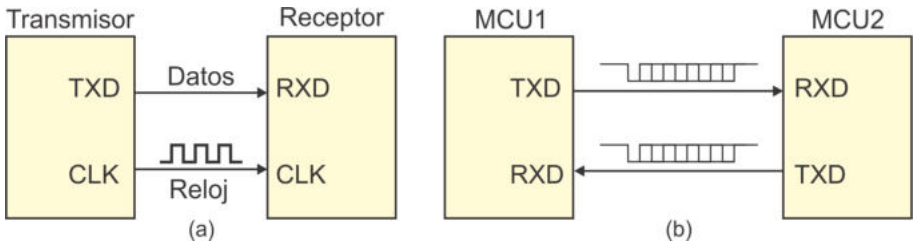


Figura 1.8: Comunicación serial (a) síncrona y (b) asíncrona

1.5.6. Puerto Serie

La mayoría de microcontroladores cuentan con un receptor/transmisor universal asíncrono (UART, *Universal Asynchronous Receiver and Transmitter*), 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 para la señal de reloj.
- **Asíncrona:** Solo hay líneas para los datos, la comunicación se realiza con éxito si el transmisor y el receptor se configuran con la misma velocidad de transferencia (bits/segundo, *Baud Rate*) y el mismo formato para los datos.

La comunicación serial es muy útil porque solo 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 para el transmisor o de serie a paralelo para la 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 ilustra la diferencia entre la comunicación síncrona y asíncrona.

1.5.7. Entradas y Salidas Digitales

Los microcontroladores incluyen puertos digitales que funcionan como entradas o salidas para intercambiar datos con el 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 pueden 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 sirven para el manejo de LEDs, *displays* de 7 segmentos, activación de motores, LCD, etc.

1.5.8. Entradas y Salidas Analógicas

Para entradas, algunos microcontroladores incorporan un convertidor analógico a digital (ADC, *Analogic-Digital Converter*) o un comparador analógico, ambos son muy útiles porque sin elementos externos es posible monitorear parámetros analógicos como temperatura, velocidad, humedad, etc.

Para salidas, podría considerarse el uso de un convertidor digital a analógico (DAC, *Digital-Analogic Converter*) pero no es común que un microcontrolador contenga un DAC, en lugar de ello, los microcontroladores incluyen salidas PWM para manejar cargas analógicas desde terminales digitales, sin elementos externos se puede manipular la intesidad de unos LEDs o la velocidad de un motor.

1.6. Clasificación de los Microcontroladores

Existen diferentes formas de clasificar a los microcontroladores y no son excluyentes unas de otras. En esta sección se describen las formas típicas de clasificación.

Por la arquitectura de la CPU, los microcontroladores pueden ser **RISC**, si el dispositivo tiene un hardware simple o **CISC** si sus instrucciones requieren de un hardware complejo para su ejecución. Prácticamente los nuevos microcontroladores son **RISC**.

Con respecto al tamaño de los datos, se tienen microcontroladores de **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, este parámetro generalmente difiere del tamaño de las instrucciones, el cual no es empleado para una 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 el manejo de periféricos externos mapeándolos en el espacio de la memoria de datos, es decir, algunas direcciones hacen referencia a un periférico y no a datos de la memoria.

La última clasificación tiene que ver con la forma en que los datos son almacenados y manipulados por la CPU. En este esquema se distingue a las arquitecturas de acuerdo a cómo la ALU tiene acceso a los operandos involucrados en una instrucción aritmética o lógica. Bajo este criterio se distinguen cuatro tipos de arquitectura: **Pila**, **Acumulador**, **Registro-Memoria** y **Registro-Registro**.

En una arquitectura tipo **Pila**, como su nombre lo indica, los datos deben ingresar

a una pila para ser procesados, las operaciones se realizan con los últimos datos ingresados y el resultado se deja en el tope de la misma. Por ejemplo, para realizar la operación de alto nivel:

A = B - C

suponiendo que A, B y C son variables almacenadas en memoria, se tiene la secuencia de instrucciones:

```
PUSH B      ; La variable B ingresa en la pila
PUSH C      ; La variable C ingresa en la pila
SUB         ; La resta se hace con los datos de la pila
POP  A      ; Extrae el resultado 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 por lo que debe ser uno de los operandos de la ALU y el resultado también queda en el acumulador. Las instrucciones que únicamente requieren de un operando se aplican sobre el acumulador.

Si la misma operación de resta se realiza en 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 de Acc (en Acc queda el resultado)
MOV  A, Acc    ; Transfiere el Acumulador a la variable A
```

En una arquitectura del tipo **Registro-Memoria** 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 propósito general. La operación bajo consideración se realiza con las siguientes instrucciones:

```
LD   R1, B     ; Carga la variable B en el registro R1
SUB  R1, C     ; Resta la variable C de R1
ST   A, R1     ; Almacena R1 en la variable A (en memoria)
```

Finalmente, en una arquitectura del tipo **Registro-Registro** los dos operandos que llegan a la ALU deben estar en 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 primero 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 tienen las instrucciones siguientes:

```
LD   R1, B     ; Carga la variable B en R1
LD   R2, C     ; Carga la variable C en R2
SUB  R1, R2    ; La ALU opera con los registros R1 y R2
ST   A, R1     ; Almacena el resultado en la variable A
```

Este libro está enfocado al microcontrolador **ATMega328P**, un microcontrolador **RISC** de **8 bits**, con una Arquitectura **Harvard** que es **cerrada** y sus operaciones son del tipo **Registro-Registro**.

1.7. La Tarjeta Arduino

La tarjeta Arduino es una plataforma electrónica originalmente enfocada a personas con poca experiencia en programación y conocimientos básicos de electrónica, el objetivo de sus creadores fue dotar de herramientas tecnológicas a diseñadores y artistas para complementar sus invenciones, sin embargo, al rededor de Arduino se fue creando una comunidad de desarrolladores que creció sustancialmente y actualmente en el mercado existe una cantidad enorme de tarjetas de desarrollo y módulos para expandir sus capacidades. Este aspecto resultó muy atractivo para los desarrolladores de sistemas basados en microcontroladores, porque con Arduino se simplificó la evaluación de sensores, actuadores o interfaces de comunicación.

Con Arduino se involucran tres elementos principales:

- El **IDE** o **entorno** en el que se desarrollan los programas, el IDE incluye un lenguaje de programación de código abierto soportado por un gran número de bibliotecas de funciones enfocadas al manejo de recursos internos y periféricos externos, con ejemplos de cómo utilizarlos.
- Las **tarjetas de desarrollo**, que varían en precio y características, pero que están soportadas por microcontroladores AVR de Microchip. Por ejemplo, el ATMega328P es empleado en los modelos Arduino UNO, Arduino Mini y Arduino LilyPad, entre otros. Una tarjeta con mayores prestaciones es la Arduino Mega, manejada por un ATMega2560. Los diagramas de las tarjeras Arduino son públicos bajo una licencia de *Creative Commons*.
- El tercer elemento son los módulos de expansión, referidos como *shields*. Lo interesante de los *shields* es que la comunidad de Arduino no solo ha diseñado el hardware, también ha desarrollado las bibliotecas de funciones que pueden integrarse al IDE para su evaluación rápida.

Al emplear tarjetas Arduino ya se está trabajando con microcontroladores, sin embargo, las ventajas para el desarrollo rápido de sistemas a través del entorno dejan un vacío de conocimientos sobre estos dispositivos, lo que complica la optimización de los sistemas que se puede conseguir con un manejo directo de los registros.

Por ello, Arduino es una excelente opción para iniciar con el uso de microcontroladores, después de las primeras experiencias es conveniente entender su organización para estar en condiciones de desarrollar bibliotecas de funciones propias, así como la posibilidad de crear un hardware a la medida de las especificaciones de un sistema. El hardware y software deben adecuarse a los requerimientos de un sistema y no debe ocurrir que los sistemas se restrinjan por las características de una tarjeta de desarrollo.

La flexibilidad de Arduino permite expandir el entorno agregando bibliotecas de C/C++. También es posible incorporar código AVR-C directamente en los *sketchs* o programas de Arduino, en algunos de los ejemplos descritos en el texto se hace

este ejercicio, sobre todo en donde el código C simplifica un *sketch* en contraste a emplear únicamente código Arduino.

1.8. Sistemas Embebidos e Internet de las Cosas

Los **sistemas embebidos** o empotrados son sistemas de propósito específico basados en microcontroladores, FPGAs o tarjetas de desarrollo como Arduino, la característica principal es que el usuario no detecta la presencia del dispositivo programable y solo aprovecha sus beneficios.

El avance en la tecnología ha llevado a una miniaturización en los componententes electrónicos, lo que permite empotrar sistemas completos de procesamiento en artículos electrónicos convencionales o subsistemas electrónicos que son parte de un sistema complejo, el nombre de **Sistema Embebido** surgió de manera natural con esta integración. Un sistema embebido cumple con la funcionalidad original para la cual fue creado pero está dotado de la inteligencia que le proporciona el procesador empotrado, el cual también le da un soporte para ampliar sus capacidades.

Los microcontroladores son la base para los sistemas embebidos, también se llegan a emplear FPGAs, aunque es menos frecuente. El MCU es un control computarizado dentro del sistema, al cual le brinda inteligencia y le agrega un valor substancial al producto final. A pesar de su extenso uso, la mayoría de personas no es consciente de que tiene un contacto continuo con varios sistemas embebidos, en el auto, el hogar y la oficina.

Por ejemplo, una cafetera es un electrodoméstico simple que calienta el agua de un depósito por medio de una resistencia eléctrica, el agua se bombea automáticamente cuando alcanza su punto de ebullición y de esta manera llega al compartimiento del café, el usuario determina el encendido y apagado de la cafetera. La cafetera se convierte en un sistema embebido si un microcontrolador interno maneja un reloj de tiempo real y a través de periféricos como un *display* y botones, le permite al usuario configurar tareas como el encendido a una hora predeterminada, el apagado después de un tiempo de operación o el apagado urgente en caso de detectar un exceso de corriente debido a alguna falla, entre otras posibles funciones.

Por otra parte, la frase **Internet de las Cosas** (IoT, *Internet of Things*) indica que actualmente no solo las computadoras se pueden conectar a Internet, sino que también algunos sistemas embebidos cuentan con esta conexión, ya sea por cable o de manera inalámbrica. Un sistema embebido se convierte en un dispositivo del Internet de las cosas cuando puede ser monitoreado o manipulado a distancia, no todos los sistemas embebidos tienen esta propiedad, en algunos casos resulta innecesario. Para el ejemplo de la cafetera, esta sería un dispositivo IoT si desde una página web se pudiera monitorear su estado o realizar su encendido o apagado.

El Internet de las Cosas es algo mucho más amplio que dotar de conexiones a los

sistemas embebidos, también involucra el manejo de protocolos de red, gestión y almacenamiento de datos, seguridad e interfaces. Esto hace necesario el uso de microcontroladores de gama alta para el desarrollo de sistemas IoT, un ATMega328P queda limitado para ello, a menos que se complemente con módulos que apoyen esa funcionalidad.

1.9. Ejercicios

1. ¿Qué es un microcontrolador?
2. Describa qué es un sistema embebido y por qué es conveniente el uso de microcontroladores en sistemas embebidos.
3. Exprese las diferencias entre microcontroladores y microprocesadores, considere sus características de hardware y aplicaciones.
4. Explique en qué situaciones es conveniente o necesario el uso de un FPGA en lugar de un MCU.
5. Realice un diagrama con la organización típica de un microcontrolador.
6. Describa el papel de una CPU en un microcontrolador (o computadora) y explique las tareas que realiza con cada instrucción.
7. 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)
8. Liste los grupos de instrucciones típicos que maneja una CPU.
9. Explique en qué difiere una arquitectura Harvard de una arquitectura basada en el modelo de von Neumann.
10. Explique las diferencias entre una arquitectura RISC y una arquitectura CISC.
11. Indique los tipos de memoria utilizados por los microcontroladores para el almacenamiento de instrucciones y para el almacenamiento de datos.
12. 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

13. Para la operación en alto nivel A = B + C + D, suponiendo que A, B, C y D son variables ubicadas en memoria de datos, muestre cómo se traduce a ensamblador para una arquitectura:

    a) Tipo Pila

    b) Tipo Acumulador

    c) Tipo Registro-Memoria

    d) Tipo Registro-Registro (Carga-Almacenamiento)

    Utilice la instrucción `ADD` para la suma.

Capítulo 2

Los Microcontroladores AVR

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 ATMega328P, se describe su organización interna y sus características de funcionamiento.

2.1. Características Generales

Los microcontroladores AVR se basan en una arquitectura diseñada en 1992 por Alf-Egil Bogen y Vegard Wollan, graduados del Instituto Noruego de Tecnología. La arquitectura posteriormente fue refinada y comercializada por la firma Atmel, la cual en el año 2016 fue adquirida por la empresa Microchip, otro fabricante de microcontroladores. El término AVR no tiene un significado implícito, a veces es considerado como un acrónimo en el que se involucra a los diseñadores del núcleo, es decir, AVR se hace corresponder con **A**lf-**V**egard-**R**ISC *Microcontroller*, aunque también suele interpretarse como *Advanced Virtual RISC* (RISC virtual avanzado).

El núcleo es compartido por más de 50 miembros de la familia, proporcionando una escalabilidad amplia entre elementos con diferentes recursos. En la Figura 2.1 se ilustra este hecho, hay tres categorías de microcontroladores en la familia AVR, los miembros con menos recursos caen en la gama Tiny y los miembros con más recursos pertenecen a la categoría XMega. La gama Mega es una versión intermedia y a esta categoría pertenece el ATMega328P, que es el caso de estudio.

Este libro se enfoca al dispositivo ATMega328P, sus principales características técnicas son:

- **Memoria de código:** 32 kbytes de memoria Flash.

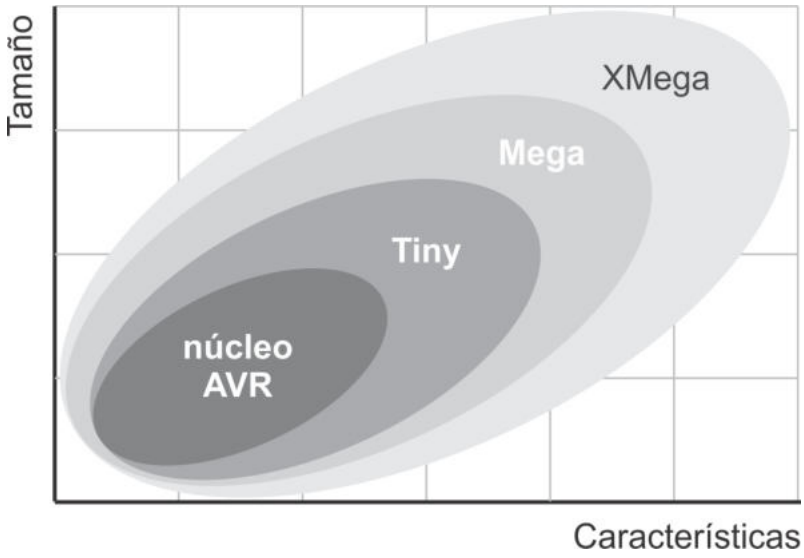


Figura 2.1: Categorías de microcontroladores en la familia AVR

- **Memoria de datos:** 2 kbytes de SRAM y 1 kbyte de EEPROM.
- **Terminales para entrada/salida:** 23.
- **Frecuencia máxima de trabajo:** 20 MHz.
- **Voltaje de alimentación:** de 1.8 a 5.5 Volts.
- **Temporizadores:** 2 de 8 bits y 1 de 16 bits.
- **Canales PWM:** 6.
- **Fuentes de interrupción:** 26.
- **Interrupciones externas:** 2 individuales y 3 por puerto.
- **Canales de conversión analógico/digital:** 6 de 10 bits en encapsulado PDIP, 8 en encapsulados TQFP o MLF.
- Facilidades para un **Reloj de Tiempo Real**.
- Interfaz **SPI** como Maestro o Esclavo.
- Transmisor/Receptor Universal Síncrono/Asíncrono (**USART**).
- Interfaz serial de dos hilos (**TWI**) compatible con I^2C.
- Programación *In System*.
- **Oscilador interno** configurable.
- *Watchdog timer*.

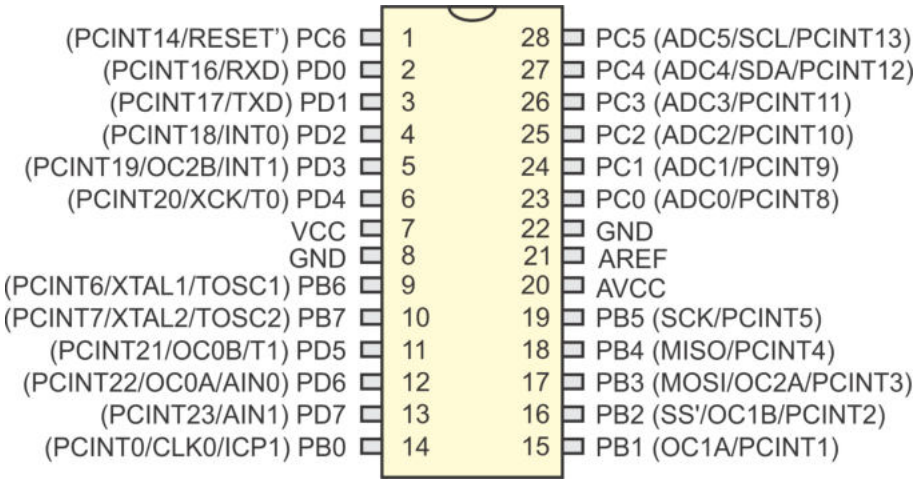


Figura 2.2: Aspecto externo de un ATMega328P

Comercialmente el ATMega328P se encuentra disponible en encapsulados PDIP con 28 terminales o encapsulados TQFP y MLF con 32 terminales. La versión en PDIP es la más conveniente durante el desarrollo de prototipos por su compatibilidad con las tablillas de pruebas (*protoboard*). En la Figura 2.2 se muestra el aspecto externo del ATMega328P con encapsulado PDIP, el MCU tiene 3 puertos, 2 de 8 bits y 1 de 7 bits. Se observa que todas sus terminales tienen una doble o triple función, esto significa que además de ser entradas o salidas de propósito general, las terminales pueden emplearse con un propósito específico, relacionado con los recursos del microcontrolador.

2.2. El Núcleo AVR

La organización interna del ATMega328P 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. El núcleo contiene un bus de 8 bits al que se conectan los diferentes recursos del microcontrolador. El núcleo es el mismo para toda la familia AVR pero los recursos pueden diferir entre dispositivos.

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 permitir el paralelismo y maximizar el rendimiento, los AVR usan 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.

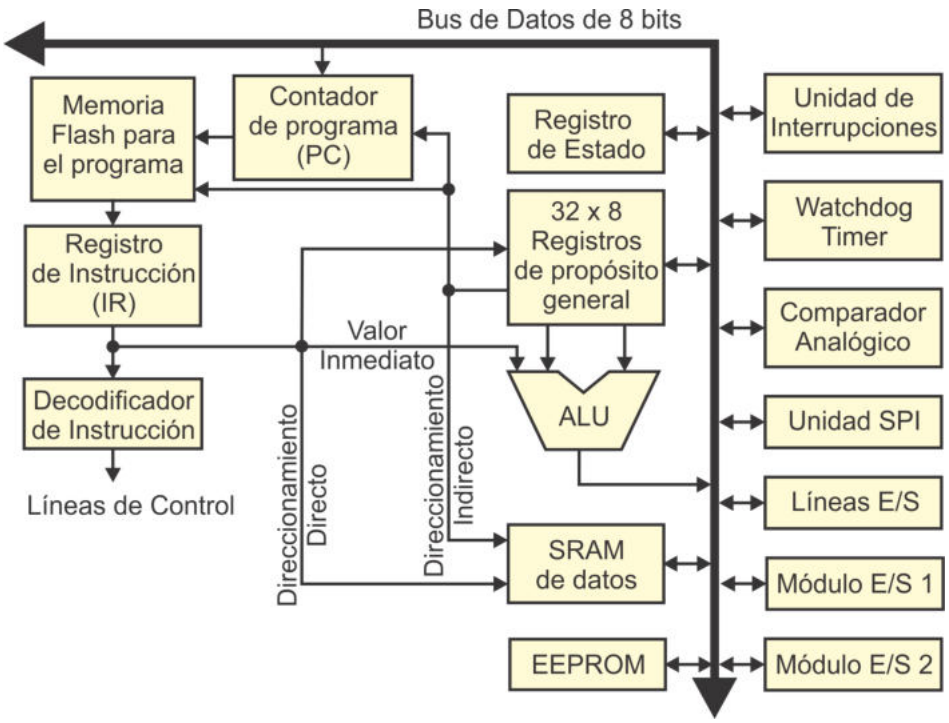


Figura 2.3: Diagrama a bloques del núcleo AVR

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 quien contiene la dirección de la instrucción a ejecutar, sin embargo, es posible que un registro de propósito general proporcione esta dirección, el registro se vuelve un apuntador y el acceso se realiza mediante 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 contiene principalmente 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.

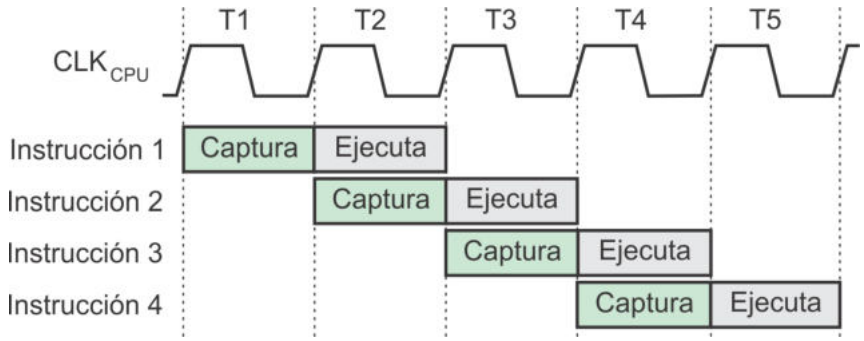


Figura 2.4: Segmentación a dos etapas realizada por el núcleo AVR

2.2.1. Ejecución de Instrucciones

El flujo de ejecución de un programa por naturaleza es secuencial, con incrementos automáticos del PC. El flujo se modifica con instrucciones de saltos y llamadas a rutinas, las cuales escriben directamente al PC y pueden abarcar el espacio completo de direcciones.

La CPU va a capturar las instrucciones para después ejecutarlas, su organización hace posible que el proceso se segmente en dos etapas, solapando la captura con la ejecución. Es decir, mientras una instrucción es ejecutada se captura la siguiente en el IR. Con ello, aunque cada instrucción requiere de dos ciclos de reloj, la productividad del procesador es de una instrucción por ciclo. En la Figura 2.4 se representa la segmentación a dos etapas empleada en los AVR, con la cual su rendimiento va a ser muy cercano a 1 MIPS[1] por cada MHz de la frecuencia del oscilador.

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

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 leen los operandos de los registros de propósito general, la ALU inicia su operación 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.

2.2.2. El Archivo de Registros

El archivo contiene 32 registros de propósito general de 8 bits, el núcleo AVR está acondicionado para tener un acceso rápido a ellos. La organización del archivo de

[1]MIPS es una métrica para medir el rendimiento de procesadores, significa Millones de Instrucciones por Segundo.

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.

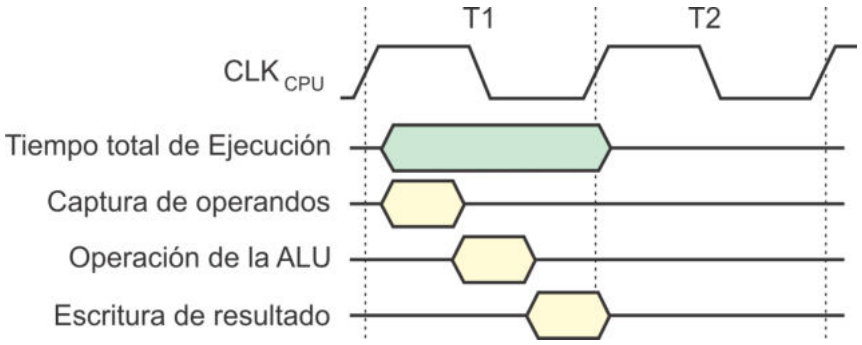


Figura 2.5: Temporización de la fase de ejecución

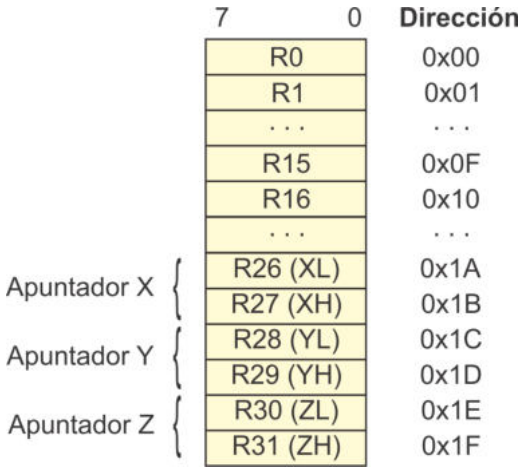


Figura 2.6: Archivo de registros

Los últimos 6 registros se organizan por pares, formando 3 registros de 16 bits que 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 con direcciones e incluir en su repertorio instrucciones de comparaciones con autoincrementos o autodecrementos, el núcleo AVR está optimizado para ejecutar 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.

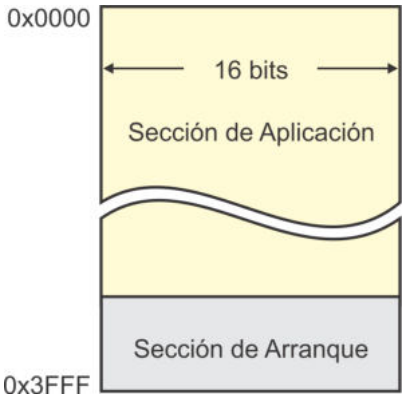


Figura 2.7: Memoria de programa en un ATMega328P

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 ATMega328P es de 32 kbytes y está organizada en palabras de 16 bits, por lo que sus direcciones van de la 0x0000 a la 0x3FFF, la mayoría de instrucciones ocupan una palabra de 16 bits, por lo que se pueden almacenar programas relativamente grandes.

La memoria se puede particionar en dos secciones, una de arranque y otra de aplicación, como se puede ver en la Figura 2.7. La sección de arranque sirve para hospedar un cargador, es decir, un programa que facilite la reprogramación del dispositivo sin requerir elementos adicionales de hardware o software. El acceso a la sección de arranque puede o no habilitarse, en caso afirmativo, después de un reinicio se ejecuta el cargador y el sistema de manera autónoma revisa si existe una nueva versión para la aplicación, de ser así, la sección de aplicación se escribe y posteriormente, con o sin actualización, la ejecución bifurca a la sección de aplicación, este aspecto de la memoria en los AVR es la base para el funcionamiento de Arduino. En el capítulo 7 se describe el manejo de la sección de arranque, si esta no es requerida, todo el espacio es dedicado a la sección de aplicación.

La memoria puede ser programada sin necesidad de retirar al MCU de un sistema (programación "*In System*") y soporta hasta 10,000 ciclos de borrado y escritura. 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 26 fuentes de interrupción en un ATMega328P, incluyendo la interrupción por reinicio (*reset*). En la Tabla 2.1 se listan los vectores de interrupciones existentes en un ATMega328P y en la Sección 2.6 se describe al sistema de interrupciones.

Tabla 2.1: Vectores de interrupción de un ATMega328P

No.	**Dir.**	**Etiqueta**	**Descripción**
1	0x000	RESET	Reinicio por encendido, terminal externa, bajo voltaje o *watchdog timer*
2	0x002	INT0	Interrupción externa 0
3	0x004	INT1	Interrupción externa 1
4	0x006	PCINT0	Interrupción por cambios en PORTB
5	0x008	PCINT1	Interrupción por cambios en PORTC
6	0x00A	PCINT2	Interrupción por cambios en PORTD
7	0x00C	WDT	Desbordamiento del *watchdog timer*
8	0x00E	TIMER2_COMPA	Coincidencia del temporizador 2 con su comparador A
9	0x010	TIMER2_COMPB	Coincidencia del temporizador 2 con su comparador B
10	0x012	TIMER2_OVF	Desbordamiento del temporizador 2
11	0x014	TIMER1_CAPT	Captura de entrada con el temporizador 1
12	0x016	TIMER1_COMA	Coincidencia del temporizador 1 con su comparador A
13	0x018	TIMER1_COMB	Coincidencia del temporizador 1 con su comparador B
14	0x01A	TIMER1_OVF	Desbordamiento del temporizador 1
15	0x01C	TIMER0_COMA	Coincidencia del temporizador 0 con su comparador A
16	0x01E	TIMER0_COMB	Coincidencia del temporizador 0 con su comparador B
17	0x020	TIMER0_OVF	Desbordamiento del temporizador 0
18	0x022	SPI_STC	Transferencia serial completa por SPI
19	0x024	USART_RX	Recepción de un dato por la USART
20	0x026	USART_UDRE	Registro de datos de la USART vacío
21	0x028	USART_TX	Transmisión de un dato por la USART
22	0x02A	ADC	Fin de conversión analógico a digital
23	0x02C	EE_READY	Fin de escritura en la EEPROM
24	0x02E	ANALOG_COMP	Comparador analógico
25	0x030	TWI	Transferencia serial por TWI
26	0x032	SPM_READY	Fin de escritura en la Flash

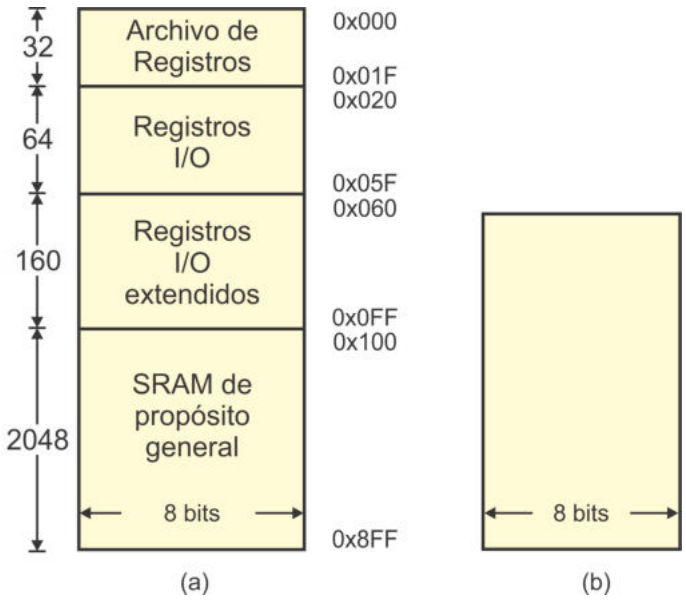


Figura 2.8: Memoria para datos: (a) SRAM y (b) EEPROM

2.4. Memoria de Datos

Los microcontroladores AVR incluyen dos espacios con tecnologías diferentes para el almacenamiento de datos: SRAM y EEPROM. La SRAM sirve para el almacenamiento de variables o datos volátiles y para el manejo de la pila de la CPU. La EEPROM es adecuada para aquellos datos que se quieren preservar aun en ausencia de energía, como contraseñas, parámetros de configuración, etc. El espacio en ambos tipos de memoria puede variar entre dispositivos, un ATMega328P cuenta con 2304 bytes de SRAM y 1024 bytes de EEPROM, esto se muestra en la Figura 2.8.

2.4.1. Espacio de SRAM

La SRAM incluye tres espacios diferentes en un mapa con direccionamiento lineal, inicia en la dirección 0x000 y concluye en la 0x8FF (ver Figura 2.8). Las primeras 32 localidades corresponden con el **Archivo de Registros**, luego siguen 64 localidades conocidos como **Registros I/O**, después está un espacio de 160 bytes denominado **Registros I/O Extendidos** y finalmente se tienen 2048 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 tratados 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.

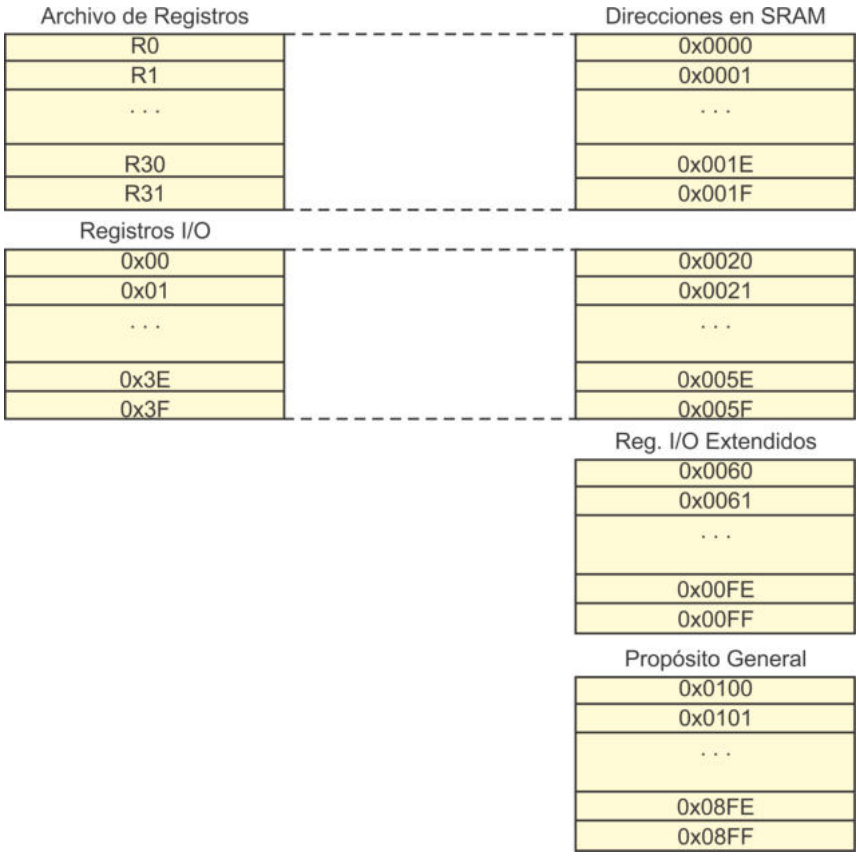


Figura 2.9: Organización de la memoria SRAM

Registros I/O

Los Registros I/O son necesarios para el manejo de los recursos internos de un microcontrolador. El ATMega328P tiene un espacio para ubicar hasta 64 Registros I/O, con direcciones que van de la 0x00 a la 0x3F, aunque no todas las localidades están ocupadas. En un AVR, el número de Registros I/O implementados depende de los recursos internos incluidos.

Los Registros I/O se utilizan para definir la configuración, realizar el control o monitorear el estado de los recursos internos. Si se revisa al núcleo AVR en la Figura 2.3, no se va a encontrar un espacio que especifique la ubicación de estos registros porque son parte de los bloques 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 AVR incluye a las instrucciones `IN` y `OUT` con las que se realiza 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. Para estas instrucciones los Registros I/O tienen las direcciones en el rango de 0x00 a 0x3F.

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`), para estos casos las direcciones que se utilizan están en el rango de 0x20 a 0x5F. En la Figura 2.9 también se observa que el Archivo de Registros está mapeado como SRAM, pero tratar a los registros de propósito general o a los Registros I/O como SRAM no es conveniente porque las instrucciones de acceso a memoria se ejecutan en 2 ciclos de reloj (adicionales a la captura).

En los capítulos 4, 5, 6, 7 y 8 se describen los recursos internos del MCU bajo estudio y para entender su operación será necesario revisar los Registros I/O relacionados con cada recurso. Solo dos Registros I/O son globales al núcleo AVR, se trata del Registro de Estado y el Apuntador de Pila, estos se describen en apartados posteriores.

Registros I/O Extendidos

Los Registros I/O Extendidos básicamente son Registros I/O mapeados en SRAM, por lo tanto, ambos grupos tienen la misma función, es decir, se emplean para la configuración y el control de recursos internos, así como para conocer su estado. Algunos recursos son manejados con los Registros I/O y otros con los Registros I/O Extendidos. Como un ejemplo, los registros para el manejo del Temporizador 1 de un ATMega328P son parte de los Registros I/O Extendidos.

El espacio para los Registros I/O Extendidos es de 160 localidades, aunque la mayoría están libres en un ATMega328P. El acceso a los Registros I/O Extendidos solo se puede realizar con instrucciones de carga (`LD`) y almacenamiento (`ST`), utilizando las direcciones 0x060 a 0x0FF, como se observa en la la Figura 2.9.

Los Registros I/O Extendidos no están presentes en los miembros de la familia AVR que tienen pocos recursos porque para ellos es suficiente con el espacio destinado para los Registros I/O, cuyo acceso es más eficiente con instrucciones `IN` y `OUT`.

Registro de Estado

El Registro de Estado (`SREG`, *State Register*) es parte de los Registros I/O, ubicado en la dirección 0x3F (o 0x5F de SRAM). Este registro es importante porque refleja el estado de la CPU después de la ejecución de instrucciones aritméticas y lógicas. Aunque su acceso puede hacerse con instrucciones `IN` y `OUT`, hay instrucciones especiales para modificar o evaluar a cada uno de sus bits individualmente. Los bits del Registro de Estado son:

REG.	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**DIR.**
SREG	I	T	H	S	V	N	Z	C	0x3F (0x5F)

- **Bit 7 - I: Habilitador Global de Interrupciones**
  Con un 1 lógico las interrupciones son habilitadas, sin embargo, cada interrupción también tiene su habilitador individual. Cuando ocurre una interrupción este bit es limpiado automáticamente para evitar que durante su atención ocurran otras interrupciones.

- **Bit 6 - T: Para transferencia de bits**
  Es un espacio para el almacenamiento temporal de 1 bit, es empleado cuando se va a copiar un bit de un registro de propósito general a otro.

- **Bit 5 - H: Bandera de acarreo del nibble bajo (*Half Carry*)**
  Se pone en alto si existe un bit de acarreo del nibble bajo al nibble alto, después de una operación aritmética.

- **Bit 4 - S: Bit de Signo**
  Mantiene la XOR entre la bandera de negativo (`N`) y la bandera de sobreflujo (`V`), ambas también en el registro de estado.

- **Bit 3 - V: Bandera de Sobreflujo**
  Indica que ocurrió un sobreflujo aritmético, es decir, el resultado de una operación aritmética no alcanzó en 8 bits, considerando una representación de números en complemento a 2 (positivos y negativos).

- **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 solo 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 no implica que el resultado requiere de un bit adicional.

Por ejemplo, al sumar el número 97 (0b01100001) con 42 (0b00101010), el resultado es 139 (0b10001011), con este resultado no hay acarreo porque alcanzó en 8 bits, sin embargo, hay sobreflujo 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 ocurre y se resumen en la Tabla 2.2.

Tabla 2.2: Situaciones de sobreflujo

Operación	**Operando A**	**Operando B**	**Resultado**
$A+B$	≥ 0	≥ 0	< 0
$A+B$	< 0	< 0	≥ 0
$A-B$	≥ 0	< 0	< 0
$A-B$	< 0	≥ 0	≥ 0

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

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. Esta característica hace que la pila sea un elemento fundamental para que un procesador organice las llamadas a rutinas. En los microcontroladores AVR la pila se implementa en la SRAM de propósito general y como parte de los Registros I/O se tiene al apuntador de pila (`SP`, *stack pointer*) para almacenar la dirección del tope.

El registro `SP` es de 12 bits para que pueda direccionar todo el espacio de la SRAM. Puesto que los Registros I/O son de 8 bits, el registro `SP` se compone de dos partes: `SPL` para la parte baja (registro ubicado en la dirección 0x3D o 0x5D) y `SPH` para la parte alta (ubicado en la dirección 0x3E o 0x5E como SRAM), aunque solo los 4 bits menos significativos de SPH son empleados, como se muestra a continuación:

REG.	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**DIR.**
SPH	-	-	-	-	SP11	SP10	SP9	SP8	0x3E (0x5E)
SPL	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0	0x3D (0x5D)

La pila tiene un crecimiento de las direcciones altas de SRAM hacia las direcciones bajas, su acceso se puede realizar en forma explícita con las instrucciones `PUSH` y `POP`. Con `PUSH Rx`, el valor de `Rx` se copia en la SRAM en la dirección tomada del registro `SP` y el regristro `SP` decrementa su valor. Con `POP Rx` se incrementa al `SP` y después se lee el dato de la SRAM ubicado en la dirección dada por el `SP`, el dato leído es copiado en `Rx`. La pila también es empleada de manera implícita durante las llamadas y retornos de rutinas. En la llamada a una rutina, en la pila se almacenan dos bytes con la dirección de la instrucción que sigue a la llamada y con un retorno, del tope de la pila se extraen dos bytes para remplazar al PC y continuar con el flujo anterior a la llamada.

Después de un reinicio el `SP` tiene el valor de 0x08FF (última localidad de la SRAM) de manera que la pila se sitúa al final de la SRAM de propósito general, sin embargo, la pila se puede mover a cualquier dirección modificando el valor del `SP`.

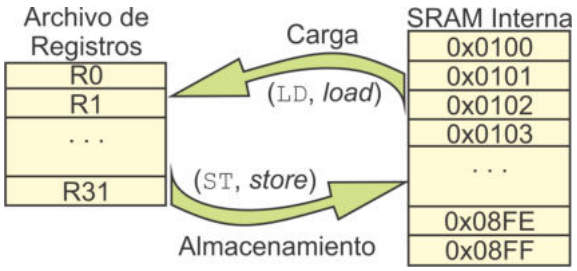


Figura 2.10: Acceso a la memoria SRAM de propósito general

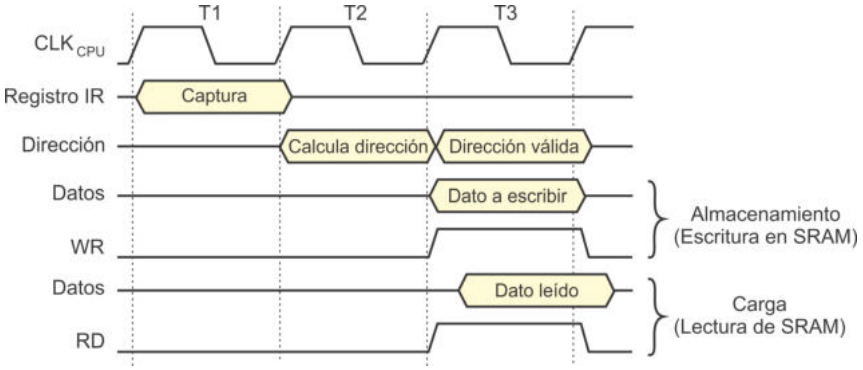


Figura 2.11: Temporización de los accesos a la SRAM de propósito general

SRAM de Propósito General

El espacio de propósito general queda disponible para variables simples que no alcancen en los 32 registros y variables compuestas como arreglos o estructuras, además de ser el lugar en donde se ubica la pila de datos temporales.

Puesto que la arquitectura AVR es del tipo Registro-Registro, cualquier variable de SRAM que requiera ser evaluada debe ser llevada a uno de los 32 registros de propósito general, esta es una operación de carga (`LD`, *load*), y para respaldar el resultado en SRAM se realiza un almacenamiento (`ST`, *store*), las operaciones de carga y almacenamiento se representan en la Figura 2.10.

Existe una variedad de instrucciones para el acceso a memoria, ya sean cargas o almacenamientos, utilizando modos de direccionamiento directo o indirecto (con los registros X, Y o Z como apuntadores), algunas de estas instrucciones modifican al apuntador automáticamente, incrementándolo o disminuyéndolo. Todos los accesos a memoria requieren de 3 ciclos de reloj, uno para la captura de la instrucción, el segundo para calcular la dirección de acceso y el último para realizar la escritura o lectura de la SRAM, este proceso 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 solo requieren de 2 ciclos de reloj.

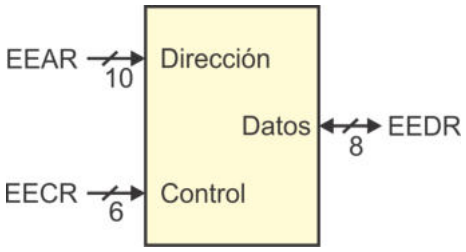


Figura 2.12: Memoria EEPROM

2.4.2. Espacio de EEPROM

La EEPROM es un espacio no volátil para el almacenamiento de datos cuyo tamaño también varía entre los integrantes de la familia AVR, en el ATMega328P este espacio es de 1024 bytes. Una memoria convencional requiere de 3 buses para su manejo: un bus de datos, un bus de direcciones y un bus de control. Pero como la EEPROM es interna al microcontrolador, los buses son manejados por medio de Registros I/O. En la Figura 2.12 se muestran los buses con su correspondiente tamaño y registros asociados.

El bus de direcciones se maneja con el registro `EEAR` (*EEPROM Address Register*), `EEAR` ocupa dos espacios en el mapa de Registros I/O para poder direccionar 1024 localidades (se requieren 10 bits de dirección), estos registros son: `EEARH` para la parte alta (ubicado en la dirección 0x22 o 0x42 como SRAM) y `EEARL` para la parte baja (ubicado en la dirección 0x21 o 0x41). Por el tamaño del bus de direcciones, en el registro `EEARH` únicamente se utilizan los 2 bits menos significativos.

Para el manejo de los datos se dispone del registro `EEDR` (*EEPROM Data Register*), el cual se ubica en la dirección 0x20, dentro del mapa de Registros I/O (o 0x40 si es tratado como SRAM). 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, el dato queda disponible en `EEDR` después de un ciclo de lectura.

Las señales de control son manejadas con el registro `EECR` (*EEPROM Control Register*), el registro se ubica en la dirección 0x1F (o 0x3F si es tratado como SRAM). En este registro se configura el modo de acceso y se hacen las habilitaciones requeridas para iniciar los ciclos de lectura o escritura, únicamente los 6 bits menos significativos están implementados, estos son:

REG.	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**DIR.**
EECR	-	-	EEPM1	EEPM0	EERIE	EEMPE	EEPE	EERE	0x1F (0x3F)

- **Bits[5:4] - EEPM[1:0]: Modos de Programación de la EEPROM**
  Definen cómo se realizará el acceso a la EEPROM, los modos se describen en la Tabla 2.3, es posible borrar el valor actual y escribir el nuevo en una operación.

Tabla 2.3: Modos de programación de la EEPROM

EEMP1	EEMP0	Tiempo	Operación
0	0	3.4 mS	Borra y escribe en una operación
0	1	1.8 mS	Únicamente borra
1	0	1.8 mS	Únicamente escribe
1	1	-	Reservado

- **Bit 3 - EERIE: Habilitador de Interrupción de la EEPROM**
  Con este bit en alto se genera una interrupción cuando concluye una escritura en la EEPROM, la duración del ciclo de escritura depende del modo de programación.

- **Bit 2 - EEMPE: Habilitador Maestro para la escritura de la EEPROM**
  Sirve como medio de seguridad para evitar escrituras no deseadas en la EEPROM, después de poner en alto a este bit se cuenta con 4 ciclos de reloj para iniciar un ciclo de escritura, se limpia automáticamente.

- **Bit 1 - EEPE: Habilitador de Escritura de la EEPROM**
  Su puesta en alto da inicio a un ciclo de escritura y se limpia automáticamente cuando el ciclo finaliza. Previamente se debió haber puesto en alto al habilitador Maestro (`EEMPE`). Para saber si hay un ciclo de escritura en progreso, un programa de usuario debe monitorear el estado de este bit o si se pone en alto al bit `EERIE` se producirá una interrupción cuando la escritura concluya.

- **Bit 0 - EERE: Habilitador de Lectura de la EEPROM**
  Se pone en alto para realizar una lectura y se limpia automáticamente por Hardware. El dato leído está disponible de manera inmediata en el registro `EEDR`. No es posible realizar una lectura si hay un ciclo de escritura en proceso.

Ejemplo 2.1: *Escriba una rutina en lenguaje ensamblador para realizar una escritura en la EEPROM.*

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

```
Esc_EEPROM:
    SBIC    EECR, EEPE      ; Revisa si hay una escritura en proceso
    RJMP    Esc_EEPROM

    OUT     EEARH, R18      ; Establece la dirección
    OUT     EEARL, R17
    OUT     EEDR, R16       ; Dato a escribir
    SBI     EECR, EEMPE     ; Habilitador Maestro en alto
```

```
    SBI     EECR, EEPE      ; Inicia la escritura
    RET
```

Con la instrucción `SBIC EECR, EEPE` (brinca si el bit `EEPE` del registro `EECR` está en bajo) se está sondeando al bit `EEPE`. El brinco no se realiza si hay una escritura en proceso y se continúa con la siguiente instrucción que 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 la EEPROM.*

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

```
Lee_EEPROM:
    SBIC    EECR, EEPE      ; Asegura que no hay una escritura
    RJMP    Lee_EEPROM

    OUT     EEARH, R18      ; Establece la dirección
    OUT     EEARL, R17
    SBI     EECR, EERE      ; Inicia la lectura
    IN      R16, EEDR       ; Obtiene el dato de la EEPROM
    RET
```

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

La función para la escritura recibe como parámetros el dato a escribir y la dirección en donde va a ser escrito:

```
void Esc_EEPROM(uint8_t dato, uint16_t dir ) {
  // Asegura que no hay escritura en proceso
  while ( EECR & 1 << EEPE )
    ;
  EEAR = dir;                 // Establece la dirección
  EEDR = dato;                // Coloca el dato
  EECR |= ( 1 << EEMPE );     // Habilitador maestro
  EECR |= ( 1 << EEPE );      // Inicia la escritura
}
```

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

```
uint8_t Lee_EEPROM(uint16_t dir ) {
  // Asegura que no hay escritura en proceso
  while ( EECR & 1 << EEPE )
    ;
  EEAR = dir;                 // Establece la dirección
  EECR |= ( 1 << EERE );      // Inicia la lectura

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

En lenguaje C se puede emplear al registro `EEAR` completo, no es necesario separarlo en sus partes alta y baja porque se pueden manejar variables de 16 bits.

2.5. Puertos de Entrada/Salida

Los puertos de entrada/salida son el medio para que un microcontrolador se comunique con su entorno. En la Figura 2.2 se mostró el aspecto externo del ATMega328P, este tiene tres puertos: Puerto B, C y D, los Puertos B y D son de 8 bits y el Puerto C es de 7 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 dos o tres funciones alternas, esto es necesario porque muchos de los recursos internos requieren información del exterior. Por ejemplo, el puerto serie necesita una terminal externa para recepción y otra para transmisión, esta es una función alterna para PD0 y PD1, respectivamente, otra función alterna en ambos casos es la interrupción por cambios en las terminales. Las funciones alternas se revisan conforme se van describiendo los recursos internos del microcontrolador, en los capítulos 4, 5, 6, 7 y 8.

El manejo de cada puerto requiere de 3 registros del espacio de Registros I/O, en la Figura 2.13 se muestra cómo se relacionan estos registros en una terminal para comprender su operación. Los registros son:

- **DDRx:** Registro que determina la dirección de un puerto. El registro controla un *buffer* de 3 estados, con un 1 lógico el *buffer* se activa y el puerto funciona como salida, con un 0 lógico el *buffer* está inactivo y el puerto es entrada (las terminales son entradas por defecto). Cada terminal tiene sus propios recursos, de manera que la dirección de una terminal puede diferir de las demás, aun en el mismo puerto.

- **PORTx:** Registro que está conectado a la terminal del puerto a través del *buffer* de 3 estados. Cuando el puerto es salida, el *buffer* está activo y todo lo que se escriba en `PORTx` se verá reflejado en las terminales externas. Cuando el puerto es entrada, el registro `PORTx` sirve para habilitar un resistor de *pull-up*.

- **PINx:** Si un puerto es entrada, este registro sirve para hacer lecturas directas en las terminales. Cuando el puerto es salida, al escribir un 1 en este registro se conmuta el valor almacenado en el registro `PORTx`, si hay un 1 lógico cambia a un 0 lógico y viceversa.

La x hace referencia a cada uno de los puertos, puede ser B, C o D. Por los 3 puertos se requiere de 9 Registros I/O, todos tienen dos direcciones, una como parte de los Registros I/O y otra para ser tratados como SRAM, estos son:

REG.	7	6	5	. . .	1	0	DIR.
PINB	PINB7	PINB6	PINB5	. . .	PINB1	PINB0	0x03(0x23)
DDRB	DDRB7	DDRB6	DDRB5	. . .	DDRB1	DDRB0	0x04 (0x24)
PORTB	PORTB7	PORTB6	PORTB5	. . .	PORTB1	PORTB0	0x05 (0x25)
PINC	-	PINC6	PINC5	. . .	PINC1	PINC0	0x06 (0x26)
DDRC	-	DDRC6	DDRC5	. . .	DDRC1	DDRC0	0x07 (0x27)
PORTC	-	PORTC6	PORTC5	. . .	PORTC1	PORTC0	0x08 (0x28)
PIND	PIND7	PIND6	PIND5	. . .	PIND1	PIND0	0x09 (0x29)
DDRD	DDRD7	DDRD6	DDRD5	. . .	DDRD1	DDRD0	0x0A (0x2A)
PORTD	PORTD7	PORTD6	PORTD5	. . .	PORTC1	PORTC0	0x0B (0x2B)

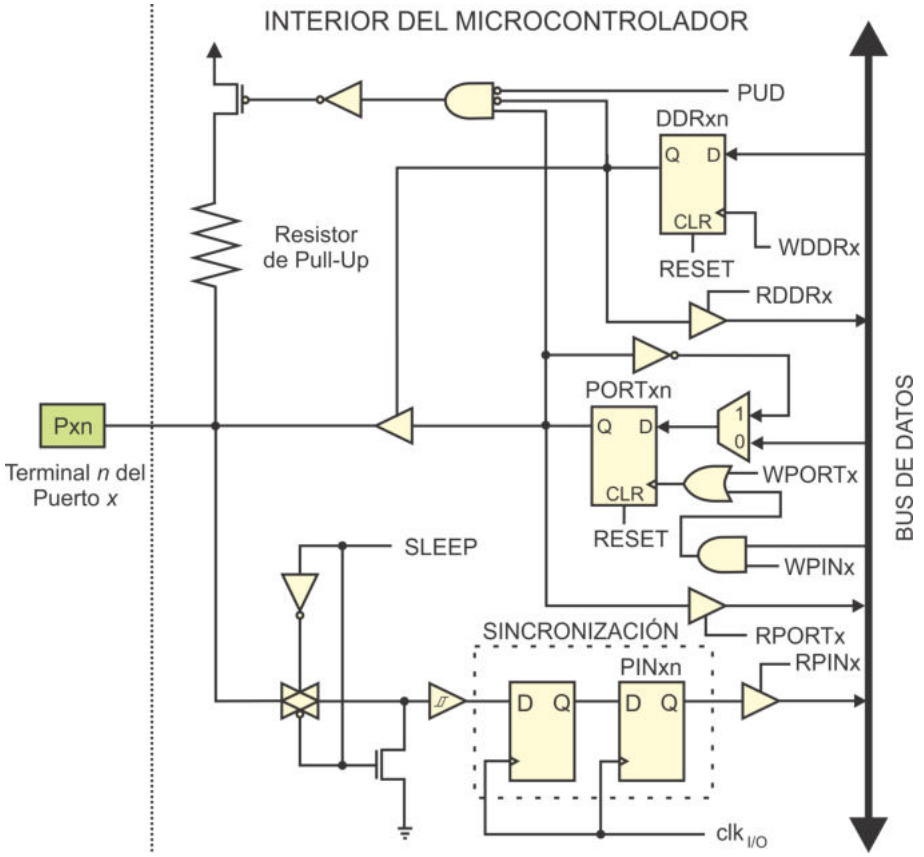


Figura 2.13: Hardware de la terminal n del puerto x

En la Figura 2.13 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 el último valor escrito en el *flip-flop*.
- Una ruta de entrada manejada por el *flip-flop* tipo D denominado `PINxn`, este *flip-flop* está precedido por otro similar, ambos sincronizados con el reloj del sistema. Esta conexión en cascada garantiza una señal estable al realizar las

lecturas en la terminal del puerto, con el inconveniente de que un cambio se retarda por un ciclo de reloj. También se observa un circuito para desconectar al puerto en los modos de bajo consumo 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` y la escritura de un 1 lógico conmuta el valor de `PORTxn`.

- Una ruta para el *buffer* de tres estados que es manejada por el *flip-flop* tipo D denominado `DDRxn`, el cual también puede ser leído o escrito. Este *flip-flop* se conecta con el *buffer* de tres estados, se debe escribir un 0 lógico para desactivar al *buffer* si la terminal va a ser entrada o un 1 lógico para activarlo cuando la terminal será salida.

También se observa un resistor de fijación hacia el nivel alto de voltaje (*pull-up*), su habilitación depende de valor del *flip-flop* `PORTxn`, del *flip-flop* `DDRxn` y del bit `PUD` (*Pull-Up Disable*). El bit `PUD` se encuentra en el Registro I/O denominado `MCUCR` (*MCU Control Register*). Con estos 3 bits se generan las combinaciones que se resumen en la Tabla 2.4.

Tabla 2.4: Estado del puerto en función de `DDRxn`, `PORTxn` y `PUD`

DDRxn	PORTxn	PUD	E/S	Pull-Up	Comentario
0	0	X	Entrada	No	Entrada sin *Pull-Up*
0	1	0	Entrada	Si	Entrada con *Pull-Up*
0	1	1	Entrada	No	Entrada sin *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, solo 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 debe conectarse a tierra para introducir un 0 lógico cuando se cierre el circuito, 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 entradas 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    ; 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; // Pull-up en los 2 MSB
```

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

En ensamblador se utiliza un registro interno para hacer la transferencia:

```
LDI   R16, 0x00
OUT   DDRB, R16    ; Puerto B como entrada
LDI   R16, 0xFF
OUT   DDRD, R16    ; Puerto D como salida

IN    R17, PINB    ; Lecturas en PINB
OUT   PORTD, R17   ; Escrituras en PORTD
```

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

```
DDRB  = 0x00; // Puerto B como entrada
DDRD  = 0xFF; // Puerto D como salida
PORTD = PINB; // Lee en PINB y escribe en PORTD
```

Cabe aclarar que después de un reinicio todos los Registros I/O relacionados con los puertos tienen el valor de 0x00, por lo que por omisión, un puerto es configurado como entrada y los resistores de *pull-up* no están habilitados.

2.6. Sistema de Interrupciones

Una interrupción es la ocurrencia de un evento producido por un recurso del microcontrolador que ocasiona la suspensión temporal del programa principal. La CPU atiende al evento con una rutina de servicio a la interrupción (ISR, *Interrupt Service Routine*). Cuando concluye la ejecución de la ISR, la CPU regresa al punto en donde fue suspendido el programa principal para continuar con su ejecución. La ISR debe colocarse en una dirección preestablecida por Hardware, la cual corresponde con un vector de interrupciones.

El núcleo AVR cuenta con la unidad de interrupciones, un módulo que determina 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 debe ocurrir el evento en el recurso.

Hay dos ventajas importantes con el uso de interrupciones:

1. Se reduce el trabajo de la CPU en el programa principal porque el monitoreo de eventos se realiza por hardware, la CPU no necesita realizar un sondeo continuo para determinar el estado de los recursos.

2. Se realizan diferentes tareas en forma simultánea porque los recursos se encargan de monitorear los eventos y la unidad de interrupciones coordina su atención, aunque esta última etapa se realiza de manera secuencial.

La atención de las interrupciones se puede organizar en dos esquemas diferentes, el primero consiste en el uso de una dirección única, es decir, cualquier evento hace que la ejecución del programa bifurque a la misma dirección. Posterior a ello, por software se evalúa un conjunto de banderas para determinar qué recurso causó la interrupción y darle atención. En el segundo esquema se tienen diferentes direcciones o vectores de interrupción, una dirección por cada evento, de manera que la dirección destino de la bifurcación depende del recurso que causó la interrupción. El segundo esquema requiere de una estructura de hardware más compleja que el primero, pero el software se simplifica porque no se necesita una revisión de banderas. La arquitectura AVR utiliza vectores de interrupción y las direcciones están predefinidas como parte de la memoria de código (ver Tabla 2.1).

Un aspecto importante es que los eventos pueden ocurrir en cualquier momento, es decir, en forma asíncrona. Esta es la diferencia entre una rutina normal y una ISR, para una rutina normal se puede determinar con certeza cuándo va a ser ejecutada, de manera 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, la idea se ilustra en la Figura 2.14, por ello, una ISR debe incluir el código necesario para proteger variables o registros cuyo contenido es fundamental al nivel del programa base.

El número de interrupciones puede variar entre los miembros de la familia AVR, ya que depende de los recursos empotrados en el dispositivo.

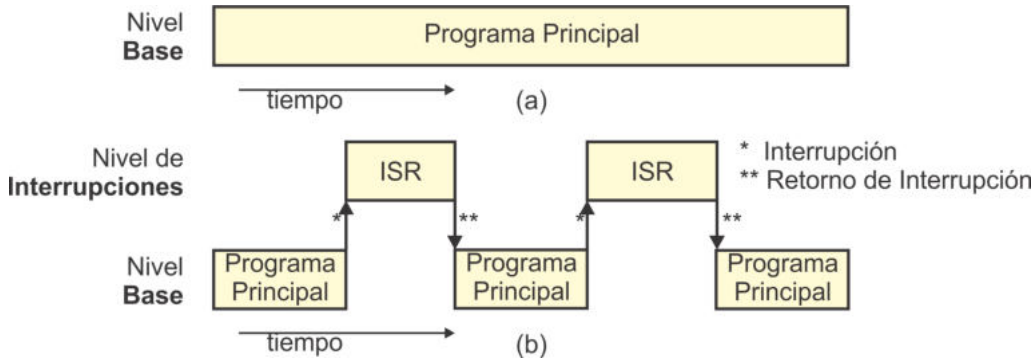


Figura 2.14: Ejecución de un programa (a) sin interrupciones y (b) con interrupciones

En el ATMega328P se tienen 26 fuentes de interrupción:

- Una por inicialización o *reset*.
- Dos interrupciones externas.
- Tres por cambios en las terminales de los puertos.
- Una por el perro guardián (*watchdog timer*).
- Diez debidas a los temporizadores: Seis por comparación, tres por desbordamiento y una por captura de entrada.
- Una por concluir una transferencia serial vía SPI.
- Tres debidas al puerto serie (USART): por fin de transmisión, por recepción y por *buffer* vacío.
- Una al finalizar una conversión analógico a digital.
- Una al finalizar la escritura en EEPROM.
- Una por el comparador analógico.
- Una por la interfaz serial a dos hilos (TWI).
- Una al concluir la escritura en memoria de programa.

En los programas que utilizan interrupciones se deben configurar los recursos que van a monitorear los eventos de interés, como parte de la configuración se debe incluir la habilitación de la interrupción propia de cada recurso. También se requiere la puesta en alto del bit `I` del registro `SREG` que es el habilitador global de interrupciones. Posteriormente el programa continúa con su ejecución normal, es frecuente que el programa principal ingrese en un lazo infinito realizando alguna actividad mínima o nula, dejando las tareas importantes a las ISRs.

Cuando ocurre un evento, la bandera que produce la interrupción se pone en alto y el microcontrolador automáticamente realiza lo siguiente:

- Concluye con la instrucción bajo ejecución.
- Desactiva al habilitador global de interrupciones para que no se genere otra interrupción mientras atiende a la actual.
- Respalda en el tope de la pila al PC (previamente incrementado).
- El SP se decrementa para apuntar a la última localidad vacía.
- Asigna al PC una dirección de los vectores de interrupción, que corresponda con el evento que la generó para dar paso a su ISR.
- Atiende al evento con la ISR.

Si se programa en lenguaje ensamblador, al inicio de la ISR se deben respaldar en la pila los registros que van a ser empleados en el código que atenderá la interrupción porque se desconoce si tienen información que posteriormente será utilizada en el programa principal y su valor debe ser recuperado antes de concluir con la ISR (con la instrucción `PUSH` se respalda un registro y con la instrucción `POP` se recupera). En lenguaje C estas acciones no son necesarias porque el respaldo y recuperación de la pila es parte del código que se agrega al hacer la traducción de alto nivel a código máquina.

Cuando una ISR termina (con la instrucción `RETI` si se programa en ensamblador), también de manera automática ocurre lo siguiente:

- Se limpia la bandera del evento que generó la interrupción.
- El habilitador global se activa.
- El SP se incrementa.
- El PC toma el valor del tope de la pila.
- La ejecución continúa en el programa principal

El manejo de interrupciones hace un uso extensivo de la pila, de manera implícita para el respaldo y recuperación del PC, y de forma explícita durante el respaldo y recuperación de los registros empleados en la ISR.

Ejemplo 2.6: *Muestre cómo se debe organizar un programa en lenguaje ensamblador para utilizar interrupciones en un ATMega328P.*

```
  .ORG    0x000
   JMP    Principal      ; Salta los vectores de interrupción
   JMP    ISR_INT0       ; Bifurca a la ISR de la INT0
   JMP    ISR_INT1       ; Bifurca a la ISR de la INT1
   JMP    ISR_PCINT0     ; Bifurca a la ISR de PCINT0
    . . .                ; Salto a las ISRs que se vayan a emplear

; El código principal es posterior a los vectores de interrupción
  .ORG    0x034
Principal:
    . . .                ; Todas las inicializaciones
    . . .                ; Se deben activar las interrupciones
    SEI                  ; Habilitador global de interrupciones
LOOP:                    ; Lazo infinito
    . . .                ; Código repetitivo
    RJMP   LOOP

; Posterior al código principal, deben situarse las ISRs
ISR_INT0:                ; Respuesta a la interrupción externa 0
    . . .
    RETI                 ; Debe terminar con RETI
```

```
ISR_INT1:               ; Respuesta a la interrupción externa 1
    . . .
    RETI

; Otras ISRs
```

La directiva `.ORG` indica la dirección para ubicar las instrucciones subsecuentes en la memoria de programa. En la Tablas 2.1 se observa que cada vector de interrupción ocupa 2 localidades y por ello se utiliza la instrucción `JMP` en las diferentes bifurcaciones, esta ocupa 2 palabras de 16 bits y tiene un alcance completo en el espacio de la memoria de código. En un programa real no es necesario poner la tabla completa del vector de interrupciones, solo los saltos a las ISRs de los recursos empleados, con el apoyo de la directiva `.ORG` para la ubicación en las posiciones correctas.

En lenguaje C todas las funciones de atención a interrupciones se llaman ISR, se distinguen porque reciben argumentos diferentes, el argumento corresponde con una etiqueta proporcionada por el fabricante para cada evento, seguida de la palabra `vect`. Las etiquetas se encuentran en la tercera columna de la Tabla 2.1.

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

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

// Las ISRs se pueden ubicar antes o después del main
ISR (INT0_vect) {            // Servicio a la interrupción externa 0
    . . . .
}

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

int main(void){              // Programa Principal
    . . .                    // Todas las inicializaciones
    . . .                    // Se deben activar las interrupciones
    sei();                   // Habilitador global de interrupciones
    while(1) {               // Lazo infinito
       . . .                 // Código repetitivo
    }
}
```

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

2.7. Inicialización del Sistema (*reset*)

La inicialización o *reset* es fundamental para la operación adecuada de un MCU porque garantiza que sus registros internos van a tener un valor inicial conocido. En el ATMega328P se tienen cuatro causas o fuentes de *reset*:

- **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 1.4 V.
- **Reset Externo:** El MCU es inicializado cuando un nivel bajo está presente en la terminal RESET (PC6) por un tiempo mayor a 2.5 us, que es el mínimo requerido (t_{RST}).
- **Reset por Watchdog:** El MCU es inicializado cuando se ha habilitado al *Watchdog Timer* y este 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 con valores típicos de 1.8 V, 2.7 V o 4.3 V y el tiempo mínimo necesario (t_{BOD}) para considerar una reducción de voltaje es de 2 us.

En la Figura 2.15 se muestra la organización del hardware de *reset*, se observa como las diferentes causas pueden producir una señal denominada Reset Interno, que es la que afecta a la CPU del microcontrolador. Esta señal se mantiene en alto por un tiempo (t_{OUT}) determinado por el bloque Contador de Retardos, el cual es configurado por los bits CKSEL y SUT. Los bits BODLEVEL, RSTDISBL, 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.

Puesto que hay diferentes causas de reinicio, los AVR incluyen al Registro de Estado del MCU (`MCUSR`, *MCU Status Register*) en el cual queda indicada la causa de *reset* por medio de una bandera. El registro `MCUSR` está ubicado en la dirección 0x34 del mapa de Registros I/O (o 0x54 de SRAM) y sus bits son:

REG.	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**DIR.**
MCUSR	-	-	-	-	WDRF	BORF	EXTRF	PORF	0x34 (0x54)

- **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 externa.
- **Bit 0 - PORF:** Bandera de reinicio por encendido.

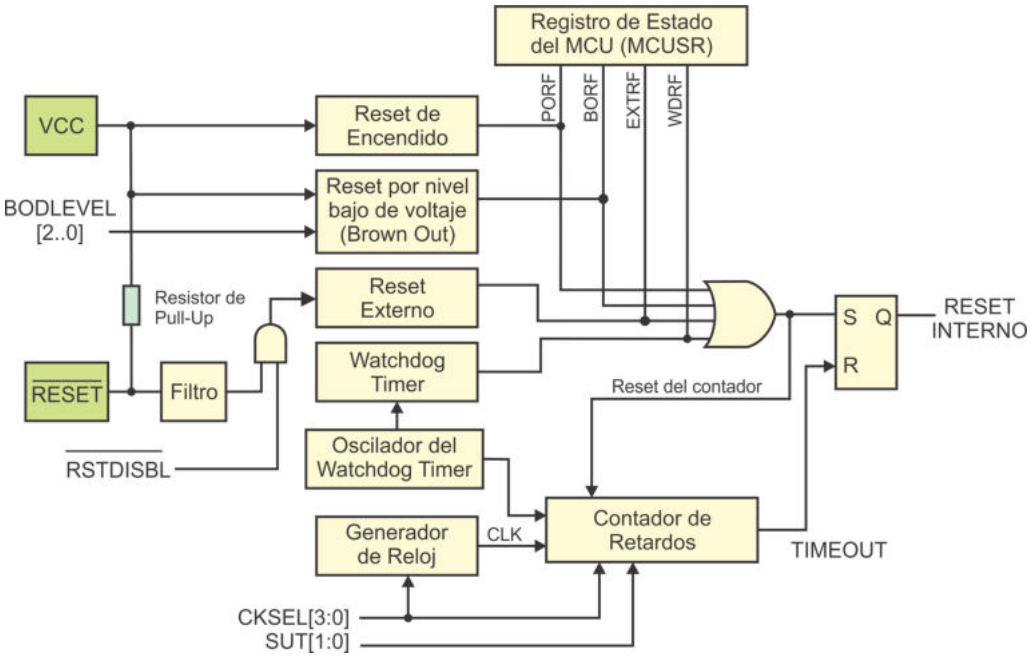


Figura 2.15: Organización del hardware de inicialización o *reset*

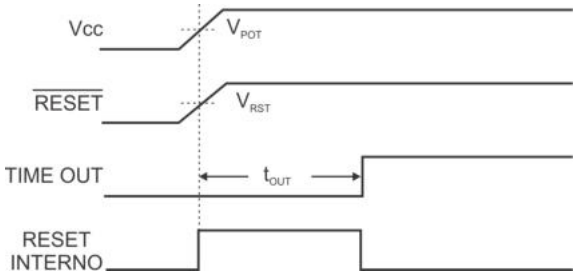


Figura 2.16: Reset de encendido

Para comprender el **Reset de Encendido**, en la Figura 2.16 se muestra la puesta en alto de la señal 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* (ver Figura 2.15) y por ello tienen el mismo comportamiento.

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 microcontroladores ATMega328P son comercializados con un tiempo de establecimiento ajustado a 65 ms.

Para generar el **Reset Externo**, el ATMega328P cuenta con la terminal RESET ubicada en el pin PC6, en esta terminal se puede conectar un circuito que mantenga un nivel bajo temporal cuando se alimenta al dispositivo o bien, un botón que

provoque el reinicio del MCU cuando ya está en operación. El comportamiento de ambos casos se muestra en la Figura 2.17, la señal Reset Interno se genera en respuesta a la terminal RESET, en (a) inicia en alto por el *reset* de encendido pero el tiempo de establecimiento inicia cuando 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 Reset Interno se va a un nivel bajo. En (b) se observa que en cualquier momento se puede introducir un nivel bajo en la terminal RESET, debido al cierre de un botón, activando nuevamente a la señal Reset Interno.

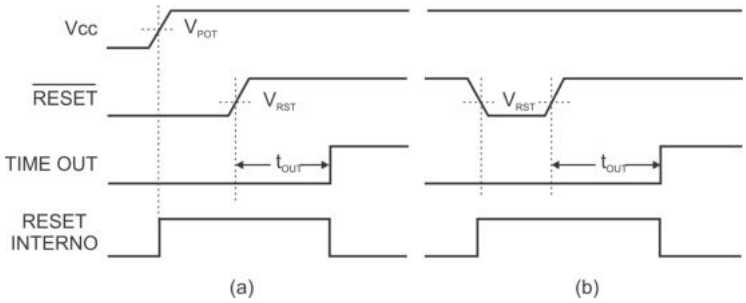


Figura 2.17: Reset externo

El **Reset por Reducción de Voltaje o Brown out** se describe en la Figura 2.18, la señal Reset Interno se pone en alto cuando la terminal de alimentación (V_{CC}) está por debajo de un umbral inferior (V_{BOT-}), el tiempo de establecimiento inicia cuando V_{CC} alcanza al umbral superior del voltaje(V_{BOT+}) y la señal Reset Interno se desactiva al terminar el tiempo de establecimiento. Esta condición de inicialización solo se puede presentar si fue habilitada, la habilitación y selección del valor para el voltaje de umbral (V_{BOT}) se realiza con los fusibles BODLEVEL (que son parte de los *Bits de Configuración y Seguridad*), en la Tabla 2.5 se muestran las diferentes opciones de configuración. Los umbrales inferior y superior se obtienen a partir del valor de (V_{BOT}) y considerando un nivel de histéresis, de manera que $V_{BOT+} = V_{BOT} + V_{HYST}/2$ y $V_{BOT-} = V_{BOT} - V_{HYST}/2$, el valor típico para V_{HYST} es de 50 mV. La histéresis evita oscilaciones en la generación del Reset Interno ante pequeñas variaciones de V_{CC}.

Esta condición de reinicio debe emplearse cuando el MCU va a trabajar con dispositivos que requieren un voltaje de alimentación estable, porque aunque el MCU puede operar aún si V_{CC} desciende hasta 1.8V, algunos periféricos probablemente no tendrán un comportamiento correcto.

La última causa de reinicio es el **Reset por Watchdog Timer**, cuando el WDT se desborda genera un pulso en alto por un ciclo de reloj que es suficiente para provocar una inicialización del MCU, en la Figura 2.19 se muestra el comportamiento de las señales involucradas, la señal Reset Interno se pone en alto justo en el momento en que ocurre el desbordamiento e inicia el tiempo de establecimiento, transcurrido este periodo de tiempo la señal Reset Interno regresa a un nivel bajo.

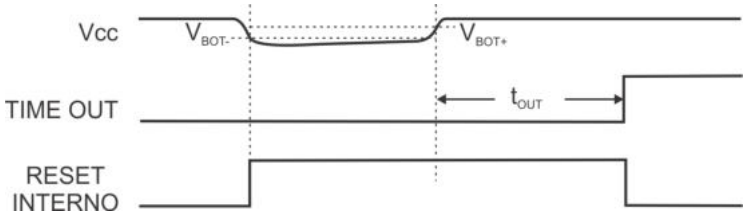


Figura 2.18: Reset por reducción de voltaje

Tabla 2.5: Selección del voltaje de umbral (V_{BOT})

BODLEVEL	**MIN**	**TIP**	**MAX**
111	Detector de bajo voltaje inactivo		
110	1.7 V	1.8 V	2.0 V
110	2.5 V	2.7 V	2.9 V
100	4.1 V	4.3 V	4.5 V
otros	Reservado		

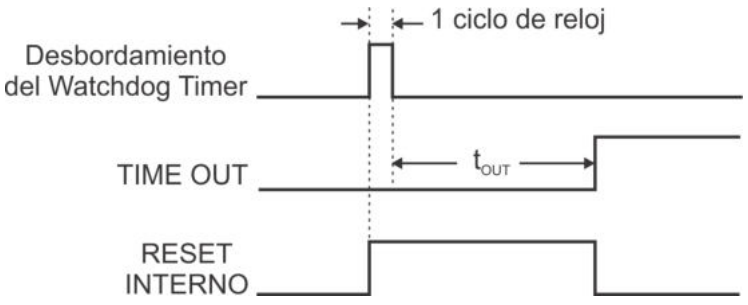


Figura 2.19: Reset por desbordamiento del *Watchdog Timer*

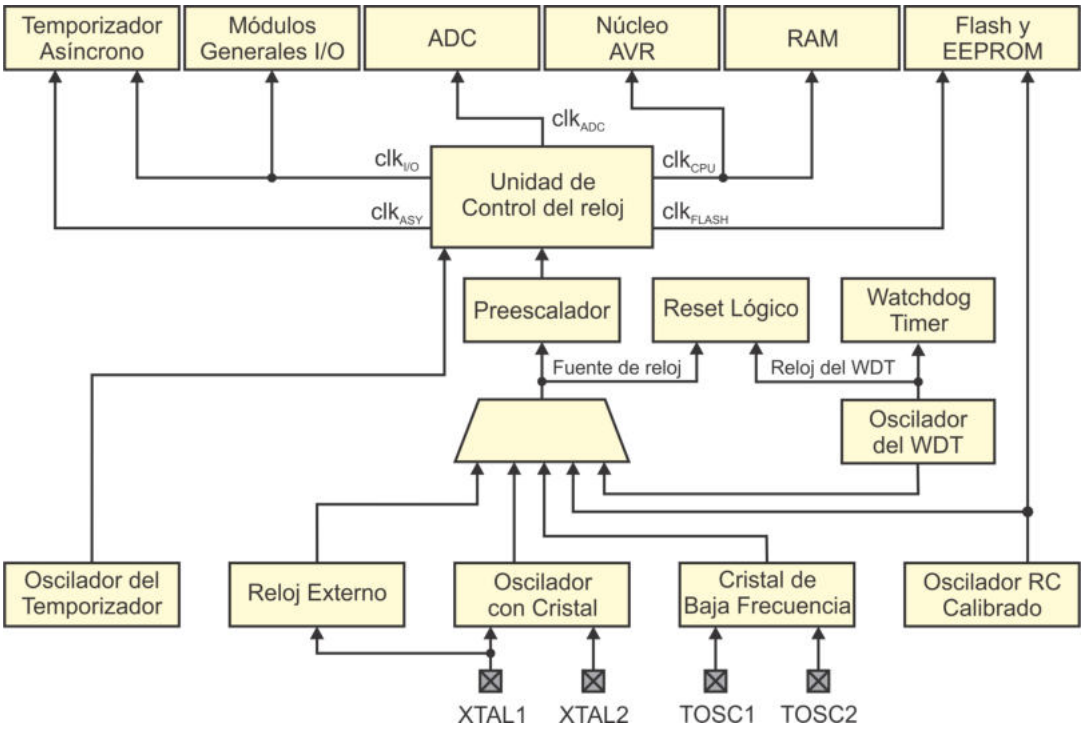


Figura 2.20: Generación y distribución de la señal de reloj

2.8. Reloj del Sistema

La sincronización de un MCU requiere de una señal de reloj principal, sin embargo, la frecuencia de operación no es la misma para todas las aplicaciones por lo que los MCUs deben contar con el hardware necesario para seleccionar entre diferentes fuentes de reloj.

En un ATMega328P se puede seleccionar entre cinco opciones para la generación del reloj del sistema. En la Figura 2.20 se observa que por medio de un multiplexor se selecciona la fuente que llega a la Unidad de Control de Reloj, esta se encarga de generar y distribuir diferentes señales de reloj a todos los módulos del MCU. También se observa que hay un oscilador que evita el multiplexor y llega directamente al Control de Reloj, este oscilador está destinado al temporizador 2 del MCU, por lo que puede ocurrir que en alguna aplicación se tengan dos relojes activos, uno para el temporizador 2 y otro para el resto del sistema.

Las señales que genera y distribuye la unidad de control del 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}: Es una señal de reloj dedicada al ADC, trabaja a una frecuencia

menor que la CPU para mejorar las conversiones al reducir el ruido generado por interferencia digital.

- $clk_{I/O}$: Es la señal de reloj utilizada por los principales recursos del MCU: Temporizadores, interfaz SPI, USART e interrupciones externas.
- clk_{ASY}: Es una señal de reloj 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.

La fuente para el reloj se selecciona con los fusibles CKSEL (por *clock selection*), que son parte de los *Bits de Configuración y Seguridad*. En la Tabla 2.6 se muestran las opciones disponibles y en los siguientes apartados se describe cada una de ellas. Otros fusibles relacionados son los SUT (por *Start-Up Time*), con ellos es posible modificar el tiempo de establecimiento después de que el MCU sale de un modo de bajo consumo o después de un reinicio, las combinaciones de los fusibles SUT también se describen en los siguientes apartados.

Tabla 2.6: Selección de reloj a partir de los fusibles CKSEL

Opción para el Reloj del Sistema	**Bits CKSEL[3:0]**
Cristal de baja potencia	1111 - 1000
Cristal de rango completo	0111 - 0110
Cristal de baja frecuencia	0101 - 0100
Oscilador RC interno de 128 kHz	0011
Oscilador RC calibrado interno	0010
Reloj externo	0000
Reservado	0001

Cristal de Baja Potencia

Con esta opción se puede usar un cristal o resonador cerámico, las terminales XTAL1 y XTAL2 son entrada y salida, respectivamente. Un oscilador de baja potencia genera una oscilación pequeña en la salida XTAL2, por lo que es muy susceptible a ambientes ruidosos, si se quiere una mayor inmunidad al ruido debe emplearse un cristal de rango completo, pero se tendrá un ligero aumento en el consumo de corriente. En la Figura 2.21 se muestra la conexión del oscilador con dos capacitores para estabilización. En la Tabla 2.7 se muestra el rango de frecuencias de trabajo para los diferentes valores de CKSEL[3:1], se observa que los capacitores se pueden omitir en la opción de más baja frecuencia y que su intervalo de valores es el mismo en las otras opciones.

Con el fusible CKSEL0 junto con los fusibles SUT[1:0] se puede configurar el tiempo de establecimiento después de que el MCU deja un modo de bajo consumo así como

el retardo después de un reinicio. En la Tabla 2.8 se muestran las diferentes opciones de configuración, con el MCU operando a 5 V.

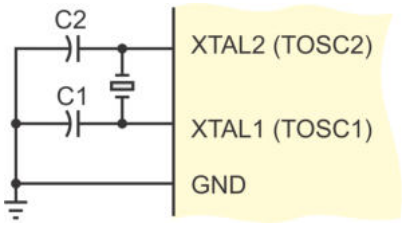


Figura 2.21: Conexión de un cristal o resonador cerámico

Tabla 2.7: Modos de operación para un oscilador de baja potencia

Frecuencia (MHz)	**Valores para C1 y C2 (pF)**	**CKSEL[3:1]**
0.4 - 0.9	-	100
0.9 - 3.0	12 - 22	101
3.0 - 8.0	12 - 22	110
8.0 - 16.0	12 - 22	111

Cristal de Rango Completo

La conexión de un cristal de rango completo es la misma que para un cristal de baja potencia y se puede ver en la Figura 2.21, las terminales XTAL1 y XTAL2 son entrada y salida, respectivamente. La diferencia es que en la salida XTAL2 se tiene una oscilación completa, por lo que al emplear este tipo de cristales se tiene una mayor inmunidad al ruido y un consumo de corriente más alto. Los cristales de rango completo pueden operar con voltajes entre 2.7 y 5.5 V.

En los fusibles CKSEL[3:1] debe emplearse la combinación “011” para cristales con un rango de frecuencias entre 0.4 y 20 MHz, los capacitores C1 y C2 deben tener valores entre 12 y 22 pF.

Similar a un cristal de baja potencia, con el fusible CKSEL0 y los fusibles SUT[1:0] se configura el tiempo de establecimiento después de que el MCU deja un modo de bajo consumo y el retardo después de un reinicio. Las combinaciones con estos bits generan los mismos retardos a los mostrados en la Tabla 2.8.

Cristal de Baja Frecuencia

Al seleccionar esta opción para el reloj, el hardware se acondiciona para ser manejado con un cristal de 32.768 kHz, esta frecuencia proporciona la base para un contador de tiempo real; cuando el MCU trabaja a esta frecuencia el consumo de potencia es muy bajo.

Tabla 2.8: Configuración de retardos: (1) al abandonar un modo de bajo consumo y (2) después de un reset

Oscilador / Condiciones de Potencia	**Retardo 1**	**Retardo 2**	**CKSEL0**	**SUT[1:0]**
Resonador Cerámico, arranque rápido	258 CK	14 CK + 4.1 ms	0	00
Resonador Cerámico, arranque lento	258 CK	14 CK + 65 ms	0	01
Resonador Cerámico, BOD habilitado	1 K CK	14 CK	0	10
Resonador Cerámico, arranque rápido	1 K CK	14 CK + 4.1 ms	0	11
Resonador Cerámico, arranque lento	1 K CK	14 CK + 65 ms	1	00
Cristal, BOD habilitado	16 K CK	14 CK	1	01
Cristal, arranque rápido	16 K CK	14 CK + 4.1 ms	1	10
Cristal, arranque lento	16 K CK	14 CK + 65 ms	1	11

Tabla 2.9: Retardo adicional al tiempo de establecimiento

SUT[1:0]	**Retardo posterior a un reset**	**Uso recomendado**
00	4 CK	Arranque rápido o BOD habilitado
01	4 CK + 4.2 ms	Arranque lento
10	4 CK + 65 ms	Frecuencia estable al inicio
11	Reservado	

El cristal se conecta como se mostró en la Figura 2.21, sin embargo, para el valor de los capacitores debe considerarse que el módulo para el manejo del oscilador tiene una capacitancia interna de 18 pF en XTAL1 y 8 pF en XTAL2 y que el cristal tiene una capacitancia parásita que generalmente no es especificada.

Para seleccionar un cristal de baja frecuencia como fuente de reloj en los bits CKSEL se debe usar una de las combinaciones “0101” o “0100” (ver Tabla 2.6), la diferencia es que con la primera se tiene un tiempo de establecimiento posterior a un modo de bajo consumo de 1K CK y con la segunda es de 32K CK. Con los bits SUT se define el retardo posterior a un *reset*, en la Tabla 2.9 se muestran las diferentes combinaciones para los bits SUT.

Oscilador RC Interno de 128 kHz

El MCU trabaja con un oscilador interno de 128 kHz si en los fusibles CKSEL se tiene la combinación “0011”. Esta es la frecuencia nominal si el MCU está operando a 3 V y 25°C.

El tiempo de establecimiento al salir de un modo de bajo consumo es de 6 CK y para el retardo posterior a un *reset* también se aplican las combinaciones de los fusibles SUT mostradas en la Tabla 2.9.

Oscilador RC Calibrado Interno

El MCU trabaja con un oscilador interno de 8 MHz si en los fusibles CKSEL se tiene la combinación "0010". Esta es la frecuencia nominal si el MCU está operando a 3 V y 25°C, en otras condiciones la frecuencia puede estar en el intervalo de 7.3 a 8.1 MHz. Con esta combinación se ponen a la venta los microcontroladores, sin embargo, el MCU incluye un preescalador que divide la frecuencia de trabajo entre 8 y es activado por el fusible CKVID8. El fusible está programado de fábrica, de manera que los microcontroladores ATMega328P inician su operación con una frecuencia de 1 MHz.

La frecuencia del oscilador interno es afectada por el voltaje de alimentación y la temperatura, así como por variaciones en los procesos de fabricación de cada dispositivo. En los AVR se incorpora un registro para la calibración del oscilador con lo que se compensan estas diferencias, el registro se denomina `OSCCAL` y se ubica en la dirección 0x66 dentro de los Registros I/O Extendidos. Después de un *reset* en el registro `OSCCAL` se carga automáticamente el valor de calibración para 8 MHz, el cual es una constante que se determina durante el proceso de fabricación. El valor del registro puede ser modificado por software para mantener la frecuencia de operación deseada por el usurario, para ello se deben revisar las notas de aplicación desarrolladas por el fabricante.

Con el uso de este oscilador, el tiempo de establecimiento posterior a un modo de bajo consumo y el retardo que sigue a un *reset* tienen el mismo comportamiento que con un oscilador RC interno de 128 kHz. El ATMega328P es comercializado con el valor de "10" en los fusibles SUT[1:0], de acuerdo con la Tabla 2.9, su tiempo de establecimiento después de un reinicio es de 65 ms.

Reloj Externo

El MCU puede ser manejado por un generador de señales con la combinación "0000" para los fusibles CKSEL. La salida del generador debe conectarse en la terminal XTAL1 y la terminal XTAL2 debe quedar sin conexión (ver Figura 2.22).

Con un reloj externo, el tiempo de establecimiento después de algún modo de bajo consumo también es de 6 CK y los retardos después de un reinicio también corresponden con los de la Tabla 2.9. Con esta selección se deben evitar cambios repentinos en la frecuencia de reloj para asegurar una operación estable del MCU. Una variación en la frecuencia mayor al 2 % de un ciclo de reloj al siguiente puede llevar a un comportamiento impredecible. Si es necesario un cambio mayor al 2 % se debe mantener un *reset* activo durante la transición.

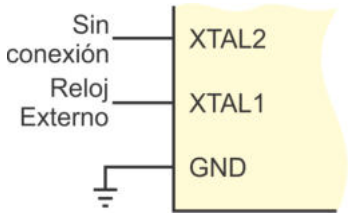


Figura 2.22: Oscilador externo

Preescalador del sistema de reloj

En la Figura 2.20 se observa un bloque preescalador que divide la frecuencia de la señal de reloj seleccionada, además de conseguir frecuencias de trabajo diferentes se puede reducir el consumo de potencia (a menor frecuencia el consumo también se reduce). La señal resultante afecta a la CPU y a los periféricos síncronos, las señales $clk_{I/O}$, clk_{ADC}, clk_{CPU} y clk_{FLASH} son divididas por el mismo factor, definido en el registro `CLKPR` (*Clock Prescaler Register*), el cual es un Registro I/O Extendido con dirección 0x61, cuyos bits son:

REG.	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**DIR.**
CLKPR	CLKPCE	-	-	-	CLKPS3	CLKPS2	CLKPS1	CLKPS0	(0x61)

- **Bit 7 - CLKPCE: Habilitador para cambiar el preescalador del reloj**
  Cuando este bit se pone en alto se cuenta con 4 ciclos de reloj para la escritura de los bits `CLKPS[3:0]`, con los que se determina el factor de preescala. El bit `CLKPCE` se limpia automáticamente transcurridos los 4 ciclos o después de que se hizo la escritura. Es conveniente desactivar las interrupciones para asegurar que la secuencia se cumpla.

- **Bits [3:0] - CLKPS[3:0]: Selectores de preescala para el reloj**
  Con estos bits se selecciona el factor de división para el reloj del sistema, en la Tabla 2.10 se muestran los factores permitidos. El preescalador debe ser empleado si la frecuencia de la fuente de reloj es mayor a la requerida por las condiciones de operación del dispositivo.

El fusible CKDIV8 determina el valor inicial de los bits `CLKPS`, si el fusible está sin programar, el valor inicial para `CLKPS` es de "0000" por lo que la frecuencia no se divide, con el fusible CKDIV8 programado los bits `CLKPS` inician con "0011", correspondiendo a un factor de división entre 8. Los bits `CLKPS` se pueden modificar en tiempo de ejecución sin importar el estado del fusible CKDIV8, el cual está programado cuando los dispositivos son puestos a la venta.

Tabla 2.10: Selección del factor de preescala para el reloj del sistema

CLKPS3	CLKPS2	CLKPS1	CLKPS0	Factor de División
0	0	0	0	1
0	0	0	1	2
0	0	1	0	4
0	0	1	1	8
0	1	0	0	16
0	1	0	1	32
0	1	1	0	64
0	1	1	1	128
1	0	0	0	256
Otras combinaciones				Reservado

Emisión del Reloj del Sistema

Sin importar la fuente de reloj, el MCU puede emitir el reloj del sistema en la terminal CLKO (función alterna para PB0). El fusible CKOUT (parte de los *Bits de Configuración y Seguridad*) debe ser programado para habilitar la salida del reloj y la terminal deja de ser de propósito general.

El reloj se emite para que otros circuitos puedan operar a la misma frecuencia, no importa si se está empleando un oscilador RC interno. Si se utiliza un factor de preescala, CLKO se emite a la frecuencia que resulte de la división.

Tabla 2.11: Modos de bajo consumo de energía

	Reloj Activo					Oscilador		El MCU despierta por:							
Modo	clk_{CPU}	clk_{FLASH}	$clk_{I/O}$	clk_{ADC}	clk_{ASY}	Reloj Principal	Reloj del temporizador 2	Eventos en los puertos	Interfaz TWI	Temporizador 2	Fin de escritura en EEPROM o Flash	ADC	WDT	Otros I/O	Desactivación del BOD
Ocioso (*Idle*)			X	X	X	X	X	X	X	X	X	X	X	X	
ADC con Bajo Ruido				X	X	X	X	X	X	X	X	X	X		
Potencia Baja								X	X				X		X
Ahorro de Energía					X		X	X	X	X			X		X
Espera (*standby*)						X		X	X				X		X
Espera Extendido					X	X	X	X	X	X			X		X

2.9. Modos de Reposo o Bajo Consumo

Los modos de reposo o bajo consumo permiten el ahorro de energía al "apagar" módulos del MCU, cuyo uso no es requerido en un sistema. Es conveniente llevar al MCU a un modo de bajo consumo cuando un sistema va a operar con baterías, en una operación normal un ATMega328P consume 0.2 mA (Trabajando a 1 MHz, 1.8 V y 25°C) y en el modo de Potencia Baja el consumo se reduce hasta 0.1 uA.

En la arquitectura AVR se simplifica el manejo de los modos de bajo consumo por la forma en que distribuye la señal de reloj, cuando un módulo no se requiere para alguna aplicación básicamente se cancela el reloj que le corresponde, con ello el módulo no trabaja y por lo tanto, no consume energía.

El ATMega328P tiene 6 modos de bajo consumo, estos se listan en la Tabla 2.11 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. Los diferentes modos de bajo consumo son:

- **Modo ocioso:** Todos los recursos del MCU trabajan pero la CPU no ejecuta instrucciones porque no tiene señal de reloj. El suministro del reloj principal y del oscilador del temporizador están activos. Se mantienen así porque prácticamente cualquier evento de los recursos puede provocar una salida de este modo de reposo.
- **Modo ADC con bajo ruido:** 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 de tiempo preestablecidos. La salida del modo es posible por eventos en los recursos principales.
- **Modo de potencia baja:** 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 aplicación muy adecuada para este modo consiste en un control remoto, porque es un sistema que generalmente está inactivo y solo reacciona cuando se presiona una de sus teclas, emite el código que le corresponde y regresa al modo de reposo.
- **Modo de ahorro de energía:** Está presente el reloj asíncrono con su correspondiente oscilador, por lo que prácticamente solo se mantiene activo al 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:** Es muy similar al modo de potencia baja, la única diferencia es que se mantiene activo el suministro de reloj principal. Con ello, el

dispositivo despierta sólo en 6 ciclos de reloj.

- **Modo de espera extendido:** Es muy similar al modo de ahorro de energía, la única diferencia es que se mantiene activo el suministro de reloj principal. El dispositivo también despierta en 6 ciclos de reloj.

Cuando el detector de reducción de voltaje (BOD, *Brown-out Detector*) es habilitado con los fusibles BODLEVEL (ver Tabla 2.5), el BOD va a monitorear continuamente el voltaje de alimentación aún si el MCU entró a un modo de bajo consumo. En algunos modos se puede desactivar al BOD por software para ahorrar energía como si los fusibles BODLEVEL no se hubiesen programado. En la última columna de la Tabla 2.11 se muestra en qué modos se puede deshabilitar al BOD y este efecto se consigue al poner en alto al bit `BODS` (*BOD Sleep*) del registro `MCUCR` (*MCU Control Register*). El monitoreo del nivel de voltaje continúa de manera automática en el momento en que el MCU deja el modo de bajo consumo.

El bit `BODS` trabaja de manera conjunta con el bit `BODSE` (*BOD Sleep Enable*), también parte del registro `MCUCR`, primero, ambos bits deben ser puestos en alto y dentro de los siguientes 4 ciclos de reloj en el bit `BODS` se debe escribir un 1 y el bit `BODSE` se debe escribir un 0. El bit `BODS` se mantiene en alto durante los 3 ciclos de reloj posteriores y dentro de ese periodo se debe ingresar al modo de bajo consumo porque de lo contrario, el bit `BODS` automáticamente se pondrá en bajo.

La selección y habilitación para el ingreso en alguno de los modos de bajo consumo se realiza en el Registro I/O `SMCR` (*Sleep Mode Control Register*), ubicado en la dirección 0x33 (o 0x53 de SRAM), cuyos bits son:

REG.	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**DIR.**
SMCR	-	-	-	-	SM2	SM1	SM0	SE	0x33 (0x53)

- **Bits [3:1] - SM[2:0]: Selectores del modo de bajo consumo**
  En la Tabla 2.12 se muestran las combinaciones de los bits SM[2:0] para elegir uno de los modos de bajo consumo.

- **Bit 0 - SE: Habilitador del modo de reposo (SE, Sleep Enable)**
  Este bit debe ponerse en alto para habilitar el ingreso a cualquiera de los modos de bajo consumo, lo cual se consigue con la ejecución de la instrucción `SLEEP`, pero antes se debe seleccionar el modo de reposo deseado.

El ATMega328P cuenta con la posibilidad de apagar algunos recursos del dispositivo (omitiendo su señal de reloj) sin que el MCU sea llevado a algún modo de ahorro de energía, esto a través del Registro para la Reducción de Potencia (`PRR`, *Power Reduction Register*), `PRR` es un Registro I/O Extendido con dirección 0x64, sus bits son:

REG.	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
PRR	PRTWI	PRTIM2	PRTIM0	-	PRTIM1	PRSPI	PRUSART0	PRADC

Tabla 2.12: Selección del modo de bajo consumo

SM2	SM1	SM0	Modo de Bajo Consumo
0	0	0	Modo ocioso
0	0	1	ADC con bajo ruido
0	1	0	Potencia baja
0	1	1	Ahorro de energía
1	0	0	Reservado
1	0	1	Reservado
1	1	0	Modo de espera
1	1	1	Modo de espera extendido

- **Bit 7 - PRTWI: Reducción de potencia en la interfaz TWI**
  Este bit en alto apaga la interfaz TWI deteniendo el reloj del módulo, el recurso se activa al limpiar el bit pero, la interfaz TWI debe inicializarse nuevamente para garantizar su operación.
- **Bit 6 - PRTIM2: Reducción de potencia en el temporizador 2**
  Este bit en alto detiene las operaciones síncronas del temporizador 2, el recurso regresa a su operación anterior al limpiar el bit.
- **Bit 5 - PRTIM0: Reducción de potencia en el temporizador 0**
  Este bit en alto detiene al temporizador 0, el recurso regresa a su operación anterior al limpiar el bit.
- **Bit 3 - PRTIM1: Reducción de potencia en el temporizador 1**
  Este bit en alto detiene al temporizador 1, el recurso regresa a su operación anterior al limpiar el bit.
- **Bit 2 - PRSPI: Reducción de potencia en la interfaz SPI**
  Este bit en alto apaga la interfaz SPI, el recurso se activa al limpiar el bit pero, la interfaz SPI debe inicializarse nuevamente para garantizar su operación. Si el MCU va a ser depurado a través del recurso *debugWire*, la interfaz SPI no debe ser apagada.
- **Bit 1 - PRUSART0: Reducción de potencia en la USART**
  Este bit en alto apaga a la USART, el recurso se activa al limpiar el bit pero, la USART debe inicializarse nuevamente para garantizar su operación.
- **Bit 0 - PRADC: Reducción de potencia en el ADC**
  Este bit en alto apaga al ADC, el cual no debe estar habilitado para que se pueda apagar. El comparador analógico no puede usar el MUX del ADC si el bit está en alto. El recurso regresa a su operación anterior al limpiar el bit.

Cuando un recurso se apaga sus registros quedan congelados, por lo que no pueden ser leídos o escritos, por ello, para no perder el estado de los recursos es conveniente

desactivarlos antes de quitarles su señal de reloj y configurarlos nuevamente después de activarlos. El registro `PRR` se puede usar cuando el MCU está en modo activo o para complementar algunos de los modos de bajo consumo.

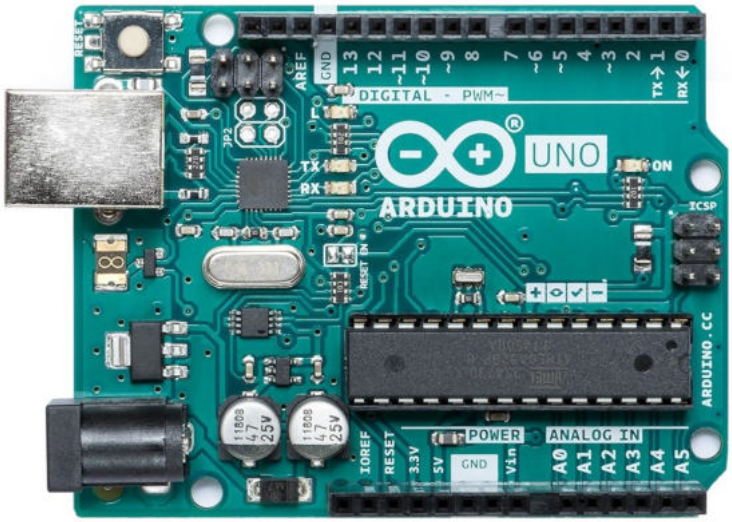


Figura 2.23: Tarjeta Arduino UNO

2.10. El ATMega328P en la Tarjeta Arduino UNO

El entorno de desarrollo de Arduino, con sus bibliotecas y funciones, ha hecho que el programador no necesite conocer la organización del MCU que indirectamente está utilizando, esto incluye al número y tamaño de los puertos. Para simplificar el trabajo de los desarrolladores, Arduino renombra las terminales con números de acuerdo a su posición o función en la tarjeta, por ejemplo, en la Figura 2.23 se muestra una Arduino UNO [2] soportada por un ATMega328P, puede verse que las entradas/salidas digitales están marcadas con los números del 0 al 13, y las entradas analógicas son referidas como A0 a A5. Para poder incluir código AVR-C en un *sketch* de Arduino, es conveniente conocer la relación entre las terminales de los puertos del MCU ATMega328P con el nombre que Arduino le otorga, esta relación se presenta en la Tabla 2.13.

Para la configuración y acceso a las terminales digitales en Arduino se utilizan las funciones:

```
pinMode()       // Para configurar una terminal como entrada o salida
digitalRead()   // Para leer una entrada de 1 bit
digitalWrite()  // Para escribir en una salida de 1 bit
```

[2]Imagen descargada del sitio https://www.arduino.cc

Tabla 2.13: Conexiones del ATMega328P en la Arduino Uno

Arduino	**MCU**	**Arduino**	**MCU**	**Arduino**	**MCU**
0	PD0	8	PB0	A0	PC0
1	PD1	9	PB1	A1	PC1
2	PD2	10	PB2	A2	PC2
3	PD3	11	PB3	A3	PC3
4	PD4	12	PB4	A4	PC4
5	PD5	13	PB5	A5	PC5
6	PD6				
7	PD7				

En las funciones se debe especificar el número o nombre de terminal a la que se tendrá acceso. Sin embargo, un *sketch* de Arduino también puede incluir instrucciones que modifiquen a los Registros I/O para la configuración y acceso de los puertos. Los registros involucrados son: `DDRx` para la configuración, `PINx` en la lectura y `PORTx` durante la escritura de cada puerto, teniendo la ventaja de poder manipular más de un bit por asignación (Sección 2.5).

En la Tabla 2.13 se observa que en la tarjeta Arduino UNO no están disponibles las terminales que corresponden con PB6 y PB7, esto es así porque el MCU trabaja con un oscilador externo de 16 MHz y este se conecta en estas terminales. Cabe señalar que todas las funciones de Arduino relacionadas con el tiempo utilizan esta frecuencia como base para sus operaciones, incluyendo la comunicación serial, por lo que si se cambia el oscilador por uno interno para reducir la frecuencia de trabajo, ya no se podrán realizar descargas desde el IDE de Arduino, no obstante, como parte del código se puede configurar al preescalador del sistema de reloj cuando una aplicación requiera una frecuencia menor. La terminal PC6 tampoco está disponible porque corresponde con la entrada para el reset externo y en la tarjeta Arduino esta terminal se conecta a tierra a través de un botón.

2.11. Ejercicios

1. Complemente la Tabla 2.14 con las características de un ATMega328P.

Tabla 2.14: Características de un ATMega328P

Recurso	**Cantidad**
Memoria Flash de código	
Memoria SRAM para datos	
Memoria EEPROM	
Registros de Propósito General	
Puertos de Entrada/Salida	
Terminales de Entrada/Salida	

2. Explique las ventajas de que el núcleo AVR trabaje en una segmentación de 2 etapas.

3. ¿Por qué se considera que los AVR están optimizados para trabajar con código C compilado?

4. En la Tabla 2.15 escriba el valor de las banderas del registro `SREG` después de la ejecución de las instrucciones especificadas.

Tabla 2.15: Estado de las banderas del registro SREG

`LDI R16, 0x73`
`LDI R17, 0x65`
`ADD R16, R17`
				`LDI R16, 0xFF`
				`LDI R17, 0xFE`
				`ADD R16, R17`			

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

`LDI R16, 0x66`
`LDI R17, 0x9A`
`ADD R16, R17`
				`LDI R16, 0xB4`
				`LDI R17, 0xC6`
				`ADD R16, R17`			

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

5. Explique para qué se emplean los Registros I/O en los microcontroladores AVR y en qué difieren de los Registros I/O Extendidos.

6. Indique qué Registros I/O se emplean para el acceso a la EEPROM y el objetivo de cada uno de ellos.

7. Explique la funcionalidad de los Registros I/O empleados para el manejo de los puertos.

8. ¿Qué es una interrupción y qué ventajas hay en el uso de interrupciones?

9. ¿Por qué es conveniente que los microcontroladores AVR puedan ser operados con una diversidad de osciladores?

10. Describa tres fuentes de inicialización (o *reset*).

11. ¿Para qué sirven los modos de bajo consumo de energía? ¿Cómo ingresa y sale un MCU de un modo de bajo consumo?

Capítulo 3

Programación de los Microcontroladores

En este capítulo se describen los aspectos relacionados con la programación del microcontrolador ATMega328P, se revisa su repertorio de instrucciones, sus modos de direccionamiento, programación en ensamblador y en lenguaje C, así como la forma en que se pueden vincular ambos estilos de programación.

3.1. Repertorio de Instrucciones

Aunque la arquitectura es tipo RISC, el repertorio de instrucciones es de un tamaño considerable, el ATMega328P tiene 131 instrucciones y por su funcionalidad, están organizadas en 5 grupos:

1. Instrucciones aritméticas y lógicas (28).
2. Instrucciones para el control de flujo (Bifurcaciones) (36).
3. Instrucciones de transferencia de datos (35).
4. Instrucciones para el manejo de bits (28).
5. Instrucciones especiales (4).

En las siguientes secciones se presentan algunas características propias de cada uno de los grupos y en el Apéndice B se incluye una referencia rápida del repertorio completo.

3.1.1. Instrucciones Aritméticas y Lógicas

El 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

Tabla 3.2: Instrucciones para las operaciones de multiplicación

Instrucción	Descripción	Operación	Banderas
`MUL Rd, Rs`	Multiplicación sin signo	$R1:R0 = Rd \times Rs$	Z,C
`MULS Rd, Rs`	Multiplicación con signo	$R1:R0 = Rd \times Rs$	Z,C
`MULSU Rd, Rs`	Rd con signo y Rs sin signo	$R1:R0 = Rd \times Rs$	Z,C
`FMUL Rd, Rs`	Multiplicación fraccional sin signo	$R1:R0 = (Rd \times Rs) \ll 1$	Z,C
`FMULS Rd, Rs`	Multiplicación fraccional con signo	$R1:R0 = (Rd \times Rs) \ll 1$	Z,C
`FMULSU Rd, Rs`	Rd con signo y Rs sin signo	$R1:R0 = (Rd \times Rs) \ll 1$	Z,C

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.

Se observa en la Tabla 3.1 que no hay una suma de un registro con una constante, esta operación se puede hacer restando el negado de la constante al registro, por ejemplo, para sumar 10 a R17 se puede emplear la instrucción `SUBI R17, -10`. Por los modos de direccionamiento, que se revisan en la siguiente sección, las instrucciones que involucran constantes solo son aplicables a los registros R16 a R31 y se dispone de 8 bits para una constante con signo, es decir, en el intervalo de $(-128, 127)$.

Las sumas y restas de 16 bits utilizan 2 registros consecutivos entre R24 y R31, además de una constante sin signo de 6 bits (de 0 a 63). En la instrucción se puede especificar un registro con número par que corresponde al byte menos significativo (`ADIW R26, k`), 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, estas se listan en la Tabla 3.2 y todas se ejecutan en 2 ciclos de reloj.

El resultado de multiplicar 2 números de 8 bits ocupa 16 bits, por ello, todas las multiplicaciones dejan el resultado en los registros R1 y R0 (en R1 la parte alta). Los operandos pueden ser interpretados como números sin signo o con signo, de acuerdo

con la instrucción que se esté empleando. En la Tabla 3.3 se muestran ejemplos que ilustran este hecho, los resultados difieren aunque los operandos tengan el mismo valor.

Tabla 3.3: Ejemplos de multiplicaciones con enteros

				Valores en Decimal		
R16	**R17**	**Instrucción**	**R1:R0**	**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

Puesto que 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 a un número con signo y el segundo es un número sin signo, el resultado será correcto si se interpreta como un número con signo (Ver Tabla 3.3).

La multiplicación fraccionaria utiliza una notación de punto fijo, bajo un esquema de 1.7 para los operandos y 1.15 para el resultado. Es decir, en los operandos se tiene 1 dígito para la parte entera y 7 para la parte fraccionaria. Lo natural es que 1.7 x 1.7 = 2.14, pero en la Tabla 3.2 aparece un desplazamiento a la izquierda, esto hace que el resultado quede en un esquema 1.15.

Para comprender la notación se tiene el siguiente ejemplo:

$$11100000_2 = 1.11_2 = 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} = 1 + 0.5 + 0.75 = 1.75$$

No obstante, si el número 11100000_2 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:

$$00100000_2 = 0.01_2 = 1 \times 2^{-2} = 0.25$$

Por lo tanto, dependiendo de la instrucción, el número 11100000_2 puede interpretarse como 1.75 o -0.25. En la Tabla 3.4 se muestran ejemplos del uso de las instrucciones de multiplicación fraccional, considerando números sin signo y con signo, con sus interpretaciones en decimal. Con la instrucción `FMULSU` ocurre algo similar que en la versión para 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.

El grupo incluye las 5 instrucciones lógicas binarias que se muestran en la Tabla 3.5, son binarias porque requieren dos operandos y el operador lógico se aplica bit

a bit. La OR exclusiva (`EOR`) solo se puede aplicar entre registros mientras que las operaciones AND y OR también se pueden aplicar con una constante. Todas se ejecutan en 1 ciclo de reloj y modifican a las banderas Z, N, V y S.

Tabla 3.4: Ejemplos de multiplicaciones con fracciones en punto fijo

				Valores en Decimal		
R16	**R17**	**Instrucción**	**R1:R0**	**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[a]	1.5	1.5	2.25
1100 0000	1100 0000	`FMULS R16, R17`	0010 0000 0000 0000[b]	-0.5	-0.5	0.25
1100 0000	1100 0000	`FMULSU R16, R17`	1010 0000 0000 0000	-0.5	1.5	-0.75

[a]La bandera de acarreo se pone en alto para representar el 2 del resultado.
[b]La bandera de acarreo se mantiene en bajo.

Tabla 3.5: Instrucciones lógicas binarias

Instrucción	**Descripción**	**Operación**	**Banderas**
`AND Rd, Rs`	AND entre registros	Rd = Rd AND Rs	Z,N,V,S
`ANDI Rd, k`	AND de un registro con una constante	Rd = Rd AND k	Z,N,V,S
`OR Rd, Rs`	OR entre registros	Rd = Rd OR Rs	Z,N,V,S
`ORI Rd, k`	OR de un registro con una constante	Rd = Rd OR k	Z,N,V,S
`EOR Rd, Rs`	OR exclusiva entre registros	Rd = Rd XOR Rs	Z,N,V,S

Las operaciones lógicas por lo general se utilizan para enmascarar información, una máscara hace referencia a un conjunto de bits de un byte y a través de operaciones lógicas AND y OR es posible modificar los bits especificados en la máscara, poniéndolos en alto o en bajo, sin alterar a los demás. Por el extenso uso de las máscaras, el grupo incluye dos instrucciones con ese propósito, estas 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 las banderas Z, C, N, V y S; aunque por su enfoque, el valor de las banderas generalmente es ignorado.

El grupo se complementa con 7 instrucciones unarias 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.

La instrucción `TST` no modifica al operando, puede usarse para evaluar si un registro tiene el valor de cero, revisando la bandera Z después de su ejecución.

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)

Tabla 3.7: Instrucciones ariméticas 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`	Decrementa un registro	Rd = Rd - 1	Z,N,V,S
`TST Rd`	Evalúa un registro	Rd = Rd AND Rd	Z,N,V,S
`CLR Rd`	Limpia un registro (pone en bajo)	Rd = 0x00	Z
`SER Rd`	Ajusta un registro (pone en alto)	Rd = 0xFF	Ninguna

Tabla 3.8: Saltos incondicionales

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

3.1.2. Instrucciones para el Control de Flujo

Para cambiar el flujo de ejecución de un programa se debe modificar el contenido del registro PC (contador del programa). En este grupo se cuenta con diversas instrucciones que hacen esta tarea, se tienen saltos incondicionales y condicionales, además de llamadas y retornos de rutinas. En la Tabla 3.8 se muestran las instrucciones de saltos incondicionales, las cuales no modifican banderas.

Los saltos relativo e indirecto se ejecutan en 2 ciclos de reloj mientras que el absoluto se ejecuta en 3. El salto indirecto únicamente se puede realizar con el apuntador Z (implícito en la instrucción).

La instrucción `RJMP` es relativa con respecto a PC + 1, a partir de ahí se bifurca hacia adelante o atrás, sumando una constante positiva o negativa. Los saltos relativos generan código "reubicable", es decir, código que se puede mover a cualquier dirección sin necesidad de hacer ajustes en el destino de los saltos, por lo tanto, las rutinas con saltos relativos se pueden compartir sin problema como código compilado, protegiendo el código fuente. Esto no se puede hacer con saltos absolutos porque hacen referencia a una dirección específica y van a provocar errores en el flujo de ejecución cuando se ubiquen en una posición diferente a la original.

Los saltos indirectos están enfocados a tablas de saltos, las cuales pueden resultar de traducir una estructura del tipo *switch-case* de alto nivel.

El repertorio incluye 3 instrucciones para llamadas a rutinas y 2 para los retornos, las 5 se describen en la Tabla 3.9, estas también son saltos incondicionales, con el agregado de que realizan accesos a la pila, las llamadas almacenan el valor de PC + 1 en la pila y los retornos recuperan el tope de la pila y con su valor remplazan el contenido del PC.

Tabla 3.9: Instrucciones para el manejo de rutinas

Instrucción	**Descripción**	**Operación**
`RCALL k`	Llamada relativa a una rutina	$Pila \leftarrow PC + 1, PC = PC + k + 1$
`ICALL`	Llamada indirecta a una rutina	$Pila \leftarrow PC + 1, PC = Z$
`CALL k`	Llamada absoluta a una rutina	$Pila \leftarrow PC + 1, PC = k$
`RET`	Retorno de una rutina	$PC \leftarrow Pila$
`RETI`	Retorno de una ISR	$PC \leftarrow Pila, I = 1$

Tabla 3.10: Instrucciones para comparar datos

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

En las llamadas a rutinas también se tienen los tres esquemas para la modificación del PC: llamadas relativas, indirectas y absolutas, similar a los saltos incondicionales. Sin embargo, las llamadas requieren de un ciclo de reloj adicional por la escritura en la pila y el ajuste del SP, es decir, las llamadas relativas e indirectas se ejecutan en 3 ciclos de reloj y las absolutas en 4 ciclos.

Las 2 instrucciones de retorno difieren ligeramente, `RET` es un retorno de rutina y `RETI` es el retorno de una ISR (rutina de servicio a interrupción). Con `RETI` además de extraer el tope de la pila para remplazar al PC se pone en alto al bit `I` del registro de estado (`SREG`), para mantener habilitadas las interrupciones. Ambos retornos tardan 4 ciclos de reloj por la carga de la pila, además del ajuste de los registros PC y SP.

Dentro del grupo se encuentran 3 instrucciones para la comparación de datos, estas se muestran en la Tabla 3.10. Una comparación no cambia el flujo de ejecución de un programa pero modifica el valor de las banderas Z, C, N, V, H y S, en el registro `SREG`. El valor de las banderas es determinante en la ejecución de los brincos condicionales que se muestran en la Tabla 3.11. Las comparaciones se realizan con base en una resta pero sin almacenar el resultado para no alterar el contenido de los operandos, de manera que solo tardan 1 ciclo de reloj.

En los programas es frecuente encontrar una instrucción de comparación (Tabla 3.10) seguida de un salto condicional (Tabla 3.11). Los brincos condicionales pueden tardar 1 o 2 ciclos de reloj en su ejecución, 1 ciclo si el brinco no se realizó porque la instrucción siguiente 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, la constante de la instrucción puede ser positiva o negativa y se suma a PC + 1 para bifurcar hacia adelante o atrás de la instrucción del brinco.

Tabla 3.11: Instrucciones para brincos condicionales

Instrucción	Condición para el brinco	Operación
`BRBS s, k`	El bit s de SREG está en alto	si(SREG(s)==1) PC = PC + 1 + k
`BRBC s, k`	El bit s de SREG está en bajo	si(SREG(s)==0) PC = PC + 1 + k
`BRIE k`	Interrupciones habilitadas	si(I==1) PC = PC + 1 + k
`BRID k`	Interrupciones deshabilitadas	si(I==0) PC = PC + 1 + k
`BRTS k`	El bit T está en alto	si(T==1) PC = PC + 1 + k
`BRTC k`	El bit T está en bajo	si(T==0) PC = PC + 1 + k
`BRHS k`	Hubo acarreo del nibble bajo	si(H==1) PC = PC + 1 + k
`BRHC k`	No hubo acarreo del nibble bajo	si(H==0) PC = PC + 1 + k
`BRGE k`	Es mayor o igual que (con signo)	si(S==0) PC = PC + 1 + k
`BRLT k`	Es menor que (con signo)	si(S==1) PC = PC + 1 + k
`BRVS k`	Hubo sobreflujo aritmético	si(V==1) PC = PC + 1 + k
`BRVC k`	No hubo sobreflujo aritmético	si(V==0) PC = PC + 1 + k
`BRMI k`	Es negativo	si(N==1) PC = PC + 1 + k
`BRPL k`	No es negativo	si(N==0) PC = PC + 1 + k
`BREQ k`	Los datos son iguales	si(Z==1) PC = PC + 1 + k
`BRNE k`	Los datos no son iguales	si(Z==0) PC = PC + 1 + k
`BRSH k`	Es mayor o igual que	si(C==0) PC = PC + 1 + k
`BRLO k`	Es menor que	si(C==1) PC = PC + 1 + k
`BRCS k`	Brinca si hubo acarreo	si(C==1) PC = PC + 1 + k
`BRCC k`	Brinca si no hubo acarreo	si(C==0) PC = PC + 1 + k

Tabla 3.12: Instrucciones para pequeños saltos

Instrucción	Condición para el "saltito"	Operación
`CPSE Rd, Rs`	Los registros son iguales	si(Rd==Rs) PC = PC + 2 o 3
`SBRS Rs, b`	El bit b del registro Rs está en alto	si(Rs(b)==1) PC = PC + 2 o 3
`SBRC Rs, b`	El bit b del registro Rs está en bajo	si(Rs(b)==0) PC = PC + 2 o 3
`SBIS P, b`	El bit b del Registro I/O P está en alto	si(P(b)==1) PC = PC + 2 o 3
`SBIC P, b`	El bit b del Registro I/O P está en bajo	si(P(b)==0) PC = PC + 2 o 3

En la Tabla 3.11 hay 20 instrucciones y todas evalúan los bits del registro de estado (`SREG`), sin embargo, solo 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 bit a evaluar queda implícito en la instrucción, puede notarse que las últimas 18 instrucciones son versiones particulares de las 2 primeras y que con esas dos sería suficiente para la evaluación de cualquier bit. 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. Las instrucciones se ordenaron en la Tabla 3.11 de acuerdo con la disposición de los bits en el registro de estado (del bit más significativo al menos significativo).

El grupo se complementa con 5 bifurcaciones condicionales de menor alcance, estas se muestran en la Tabla 3.12. La diferencia con la Tabla 3.11 es que la Tabla 3.12 hace referencia a pequeños saltos, o "saltitos", que solo excluyen la instrucción siguiente cuando la condición se cumple. Estas instrucciones por lo general están seguidas por un salto incondicional para completar el control de flujo en un programa.

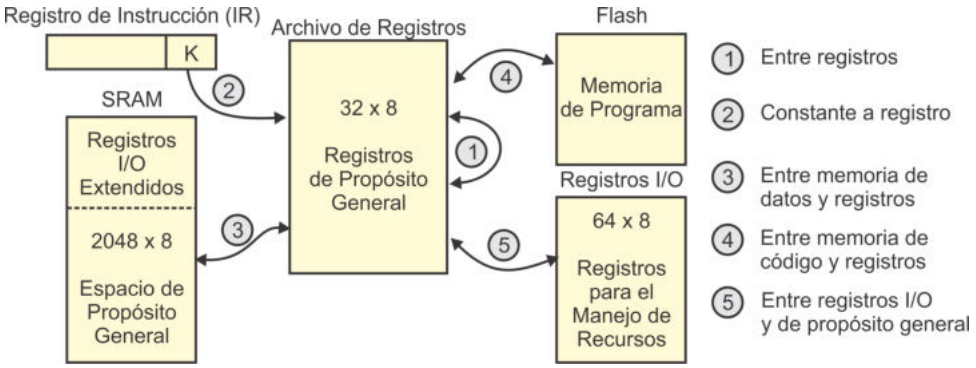


Figura 3.1: Tipos de transferencias de datos

Las instrucciones de la Tabla 3.12 pueden incrementar al PC con 2 o 3 cuando la condición es verdadera, porque se desconoce si la instrucción que se omite ocupa 1 o 2 palabras de 16 bits. Por la misma razón, para su ejecución pueden requerirse 1, 2 o 3 ciclos de reloj, si la condición no se cumple con 1 ciclo de reloj es suficiente. Con la condición verdadera se debe evaluar si la palabra ubicada en PC + 2 es una nueva instrucción (se invierten 2 ciclos) o si es parte de la instrucción ubicada en PC + 1, en cuyo caso se ajusta nuevamente el valor del PC y se requiere de otro ciclo de reloj.

Las últimas 2 instrucciones de la Tabla 3.12 hacen referencia a Registros I/O y por ello son utilizadas para evaluar el estado de los recursos, por ejemplo, se puede revisar si un botón conectado a la terminal de un puerto ha sido presionado o si existe una operación de escritura de la EEPROM en proceso.

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 esta relación, 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 de propósito general para luego transferir de registro a memoria. Las transferencias de datos no modifican las banderas del registro de estado.

Para transferencias entre registros solo 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. Para los operandos en las transferencias de palabras se puede especificar: un registro con número par que corresponde al byte menos significativo de cada palabra (`MOVW R26, R10`), ambos registros (`MOVW R27:R26, R11:R10`) o si es aplicable, el nombre de un registro de 16 bits (`MOVW X, R11:R10`). Ambas instrucciones se ejecutan en 1 ciclo de reloj.

Tabla 3.13: Instrucciones para 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 deben ser registros pares

Tabla 3.14: Instrucción para transferir una constante a un registro

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

El grupo cuenta con una instrucción para transferir una constante a un registro, esta se muestra en la Tabla 3.14 y su ejecución requiere de 1 ciclo de reloj. La constante es de 8 bits y al ser parte de la instrucción, su valor se obtiene del registro IR. Como en todas las instrucciones que utilizan constantes, solo se pueden emplear los registros del R16 al R31.

Para transferencias entre la memoria de datos y los registros se tiene un número grande de instrucciones, una carga es la transferencia de memoria a un registro y un almacenamiento es la transferencia de registro a memoria, 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 1 instrucción con direccionamiento directo (en la instrucción se especifica la dirección de acceso) y 11 con direccionamiento indirecto, utilizando a los registros X, Y o Z como apuntadores (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.

Dado que los Registros I/O Extendidos están mapeados en memoria SRAM, su lectura y ajustes se realizan con cargas y almancenamientos, generalmente se emplean instrucciones con direccionamiento directo porque no cambia la ubicación de los Registros I/O Extendidos en la SRAM.

En la Tabla 3.15 se observa que los accesos a memoria con desplazamiento se pueden realizar con los registros `Y` y `Z`, y que el apuntador no se modifica, es decir, se ignora el resultado de sumar la constante `q` con el apuntador.

El grupo incluye otras dos transferencias entre registros y SRAM, estas no son parte de la Tabla 3.15 porque están enfocadas al espacio destinado para la pila, el apuntador de pila (registro SP) contiene la dirección para insertar o extraer datos. En la Tabla 3.16 se muestran las instrucciones de acceso a la pila, se puede ver el comportamiento del registro SP para ubicarse en el nuevo tope después de cualquier

acceso. Estas instrucciones se ejecutan en 2 ciclos de reloj, incluyendo el acceso a la SRAM y el ajuste del tope de la pila.

Tabla 3.15: Instrucciones para transferencias entre SRAM y registros

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 postincremento	Rd=Mem[X], X=X+1
`LD Rd, -X`	Carga indirecta con predecremento	X=X-1, Rd=Mem[X]
`LD Rd, Y`	Carga indirecta de memoria	Rd=Mem[Y]
`LD Rd, Y+`	Carga indirecta con postincremento	Rd=Mem[Y], Y=Y+1
`LD Rd, -Y`	Carga indirecta con predecremento	Y=Y-1, Rd=Mem[Y]
`LD 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 postincremento	Rd=Mem[Z], Z=Z+1
`LD Rd, -Z`	Carga indirecta con predecremento	Z=Z-1, Rd=Mem[Z]
`LD 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 postincremento	Mem[X]=Rs, X=X+1
`ST -X, Rs`	Almacenamiento indirecto con predecremento	X=X-1, Mem[X] = Rs
`ST Y, Rs`	Almacenamiento indirecto en memoria	Mem[Y]=Rs
`ST Y+, Rs`	Almacenamiento indirecto con postincremento	Mem[Y]=Rs, Y=Y+1
`ST -Y, Rs`	Almacenamiento indirecto con predecremento	Y=Y-1, Mem[Y] = Rs
`ST 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 postincremento	Mem[Z]=Rs, Z=Z+1
`ST -Z, Rs`	Almacenamiento indirecto con predecremento	Z=Z-1, Mem[Z] = Rs
`ST Z+q, Rs`	Almacenamiento indirecto con desplazamiento	Mem[Z+q]=Rs

Tabla 3.16: Instrucciones para el 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]

Otras transferencias se realizan entre los registros de propósito general y la memoria de código, también hay cargas y almacenamientos. Las cargas son muy frecuentes porque en una aplicación conviene usar la memoria de código para almacenar datos constantes, liberando 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). En la Tabla 3.17 se pueden ver los diferentes tipos de cargas.

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. Solo se tiene 1 instrucción para almacenar datos en la memoria de código (`SPM`), no obstante, su uso es complejo porque una memoria Flash se modifica por páginas e incluye mecanismos de seguridad para proteger su contenido, el uso de la instrucción `SPM` se describe en el capítulo 9. Se observa en la Tabla 3.17 que cuando se realiza una carga se lee un byte de la memoria Flash, pero para un almacenamiento se escribe una palabra de 16 bits (con el mecanismo de seguridad y procedimiento requerido).

Tabla 3.17: Instrucciones para el acceso a memoria de código

Instrucción	**Descripción**	**Operación**
`LPM`	Carga indirecta de memoria de programa a R0	R0=Flash[Z]
`LPM Rd, Z`	Carga indirecta de memoria de programa a Rd	Rd=Flash[Z]
`LPM Rd, Z+`	Carga indirecta con postincremento	Rd=Flash[Z], Z=Z+1
`SPM`	Almacenamiento indirecto a memoria de programa	Flash[Z]=R1:R0

Por último, el grupo contiene dos instrucciones para transferencias entre Registros I/O y registros de propósito general, estas se muestran 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 pertenece, estas instrucciones se ejecutan en 1 ciclo de reloj.

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

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

3.1.4. Instrucciones para el Manejo de Bits

Como parte de las instrucciones para el manejo de bits se tienen los 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
`ASR Rd`	Desplazamiento aritmético a la derecha	Rd(n)=Rd(n+1), n=0..6
`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

En la Figura 3.2 se ilustran de manera gráfica los diferentes tipos de desplazamientos y rotaciones. Los desplazamientos lógicos insertan un 0 en el espacio generado y el

aritmético conserva el valor del bit en ese espacio. Un desplazamiento a la izquierda equivale a una multiplicación por 2 y a la derecha corresponde a una división entre 2. Si la división es para un número con signo, se debe realizar un desplazamiento aritmético en lugar de uno lógico para conservar el signo en el resultado.

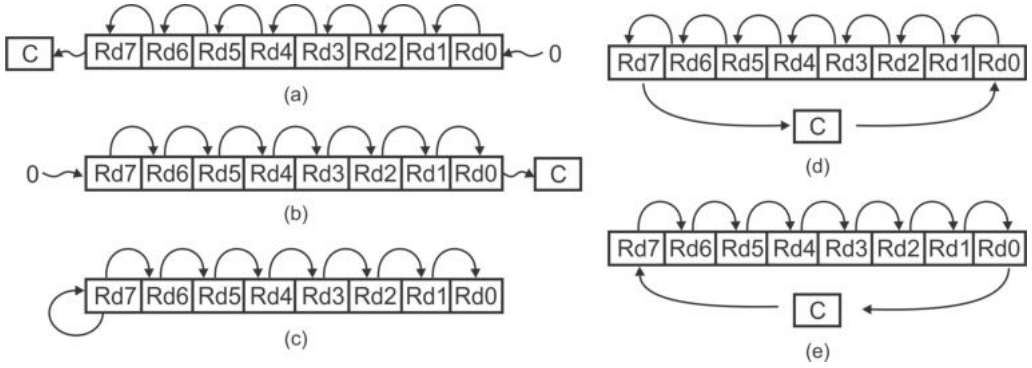


Figura 3.2: (a) Desplazamiento lógico a la izquierda, (b) a la derecha, (c) desplazamiento aritmético a la derecha, (d) rotación a la izquierda y (e) rotación a la derecha

Como parte de este grupo se cuenta con una instrucción para intercambiar el nibble alto con el nibble bajo de un registro, esta se muestra en la Tabla 3.20, se ejecuta en 1 ciclo de reloj y no modifica las banderas de `SREG`.

Tabla 3.20: Instrucción para el intercambio de nibbles

Instrucción	**Descripción**	**Operación**
`SWAP Rd`	Intercambia nibbles en Rd	Rd[7:4]=Rd[3:0] y Rd[3:0]=Rd[7:4]

El grupo también incluye instrucciones para modificar un bit en un Registro I/O, para ponerlo en alto o en bajo, siempre que esté permitido porque solo 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 no modifican las banderas de `SREG`.

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 regristro P	P(b) = 1
`CBI P, b`	Pone en bajo al bit b del regristro P	P(b) = 0

El registro `SREG` tiene al bit `T` en la posición 6, este es un espacio para el almacenamiento temporal de un bit, se emplea para mover un bit de un registro a otra posición del mismo o de otro registro. Para ello, en este grupo se cuenta con 2 instrucciones para transferencias de bits, una para almacenar (*store*) un bit de un registro de propósito general en el bit `T` y otra para cargar (*load*) del bit `T` a 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 el bit b de Rs en el bit T de SREG	SREG(T) = Rs(b)
`BTL Rd, b`	Carga al bit b de Rd desde el bit T de SREG	Rd(b) = SREG(T)

El grupo se complementa con 18 instrucciones dedicadas a manipular los 8 bits del registro de estado (`SREG`), estas se muestran en la Tabla 3.23, en las 2 primeras se indica el bit a modificar, una sirve para ponerlo en alto y la otra para ponerlo en bajo. Las 16 restantes son una versión particular de las 2 primeras, son 8 pares, un par por cada uno de los bits de `SREG`, con una instrucción el bit se pone en alto y con la otra se ponen en bajo.

Todas se ejecutan en 1 ciclo de reloj y cada instrucción va a modificar la bandera de `SREG` 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 de estado (SREG)

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

Puede verse en la Tabla 3.23 que con las 2 primeras instrucciones es suficiente para realizar el trabajo de las 16 restantes.

3.1.5. Instrucciones Especiales

Este grupo está integrado por 4 instrucciones que, por sus características, no pueden ser parte de los otros grupos. Las instrucciones especiales se describen en la Tabla 3.24, se observa que no tienen operandos y cada una tiene un propósito muy específico que no se relaciona con el de las otras.

Tabla 3.24: Instrucciones especiales

Instrucción	**Descripción**
`NOP`	No operación, obliga a una espera de 1 ciclo de reloj
`SLEEP`	Introduce al MCU al modo de bajo consumo previamente configurado
`WDR`	Reinicia al *Watchdog Timer*
`BREAK`	Para depuración con la interfaz *Debug Wire*

3.2. Modos de Direccionamiento

Los modos de direccionamiento son un aspecto fundamental en el diseño de un procesador porque definen algunas características que se ven reflejadas cuando está en operación, características que involucran: la ubicación de los datos sobre los que operan las instrucciones, el tamaño de las constantes, los registros que se pueden emplear como operandos en una instrucción y el alcance de los saltos. Los microcontroladores AVR tienen 9 modos de direccionamiento, de los cuales, 3 están relacionados con las bifurcaciones, los modos de direccionamiento son:

1. Directo por registros.
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:
   a) Relativo.
   b) Indirecto.
   c) Absoluto.

Direccionamiento Directo por Registros

En la instrucción se especifica el registro o registros que funcionan como operandos, ya que puede ser 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 solo 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
```

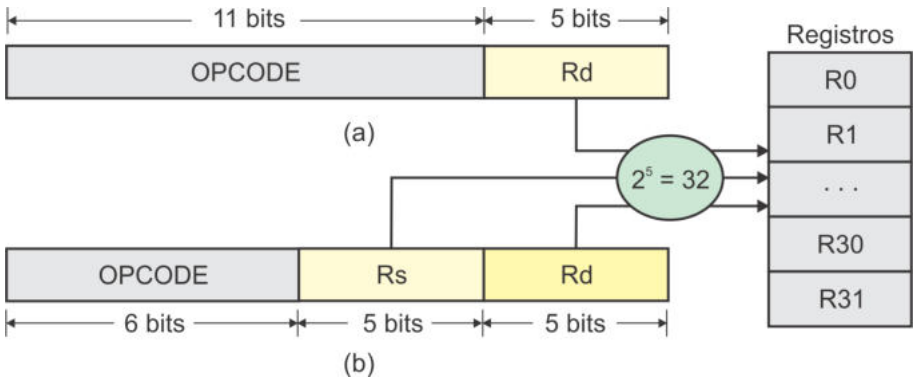


Figura 3.3: Direccionamiento directo empleando (a) un registro y (b) dos registros

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 Registros I/O disponibles. Ejemplos de instrucciones con este modo de direccionamiento son:

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

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 ATMega328P el límite superior es de 0x08FF. Los Registros I/O extendidos generalmente se manejan con este modo de direccionamiento, el cual es ilustrado en la Figura 3.5. Ejemplos de instrucciones que lo utilizan son:

```
STS   0x0100, R5
LDS   R16, 0x0110
STS   TCNT1L, R17      ; Escribe en un Registro I/O Extendido
```

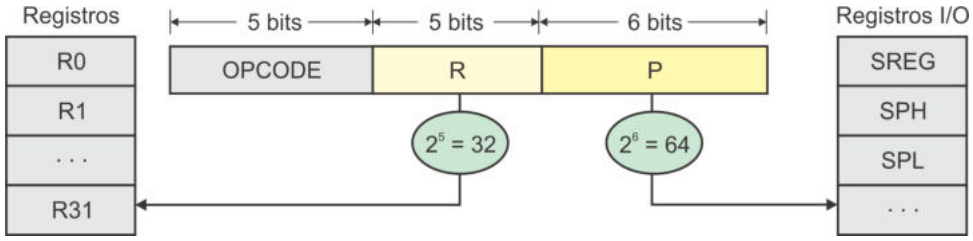


Figura 3.4: Direccionamiento directo a Registros I/O

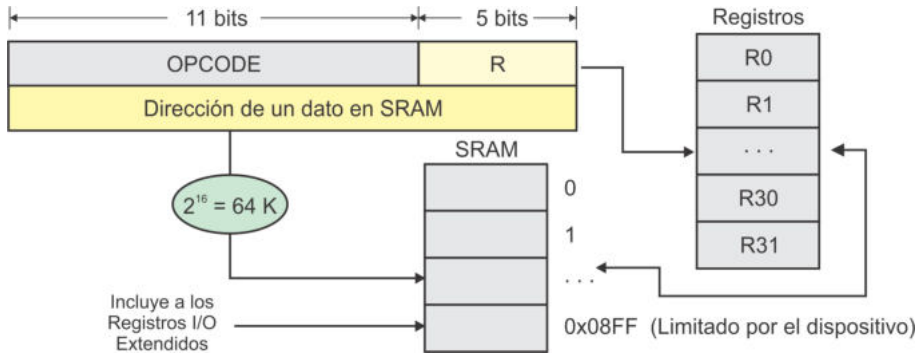


Figura 3.5: Direccionamiento directo a memoria de datos

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 solo 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
```

Existen dos variantes de este modo de direccionamiento en donde además del acceso a memoria se modifica al apuntador. La primera incluye un postincremento, es decir, primero obtiene el dato de memoria y después se incrementa al apuntador, y la segunda variante tiene un predecremento, en este caso primero se decrementa al

apuntador y luego se realiza el acceso a memoria. Estas variantes pueden utilizar a cualquiera de los registros apuntadores: X, Y o Z, en las Figuras 3.7 y 3.8 se ilustran los ajustes requeridos en el hardware para implementar estas ideas. Ejemplos de instrucciones con postincrementos y predecrementos son:

```
LD    R5, Y+      ; Carga con postincremento
ST    Z+, R6      ; Almacenamiento con postincremento
LD    R7, –X      ; Carga con predecremento
ST    –Y, R11     ; Almacenamiento con predecremento
```

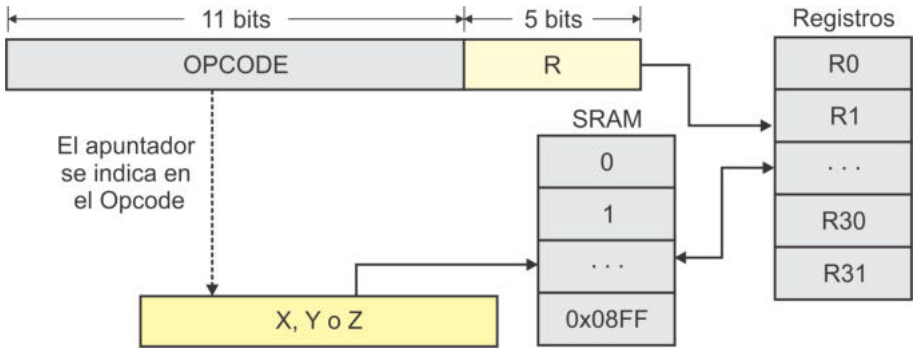


Figura 3.6: Direccionamiento indirecto a memoria de datos

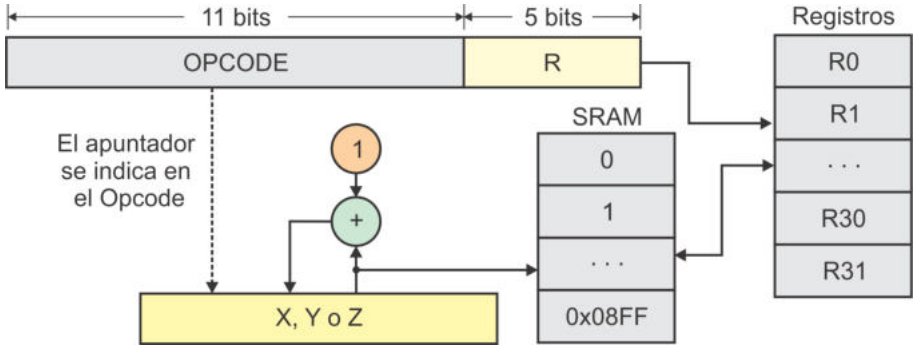


Figura 3.7: Direccionamiento indirecto con postincremento

Otra versión es el direccionamiento indirecto con desplazamiento, en este modo se tiene acceso a la dirección que resulta de la suma del apuntador con una constante, pero sin modificar al apuntador. Para la constante se dispone de 6 bits, por lo que es un número entre 0 y 63. El direccionamiento con desplazamiento se puede realizar con los apuntadores Y o Z, el hardware que incluye los desplazamientos se muestra en la Figura 3.9. Ejemplos de instrucciones con este modo de direccionamiento son:

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

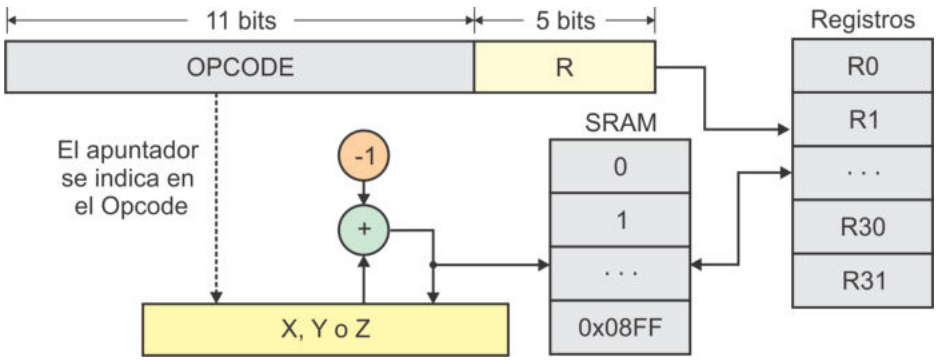


Figura 3.8: Direccionamiento indirecto con predecremento

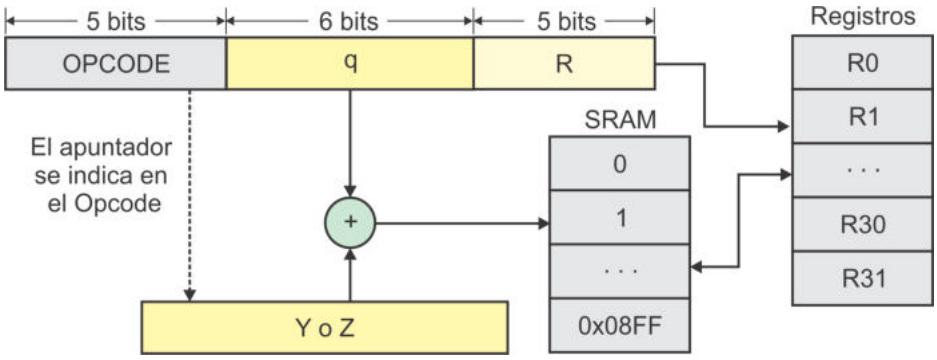


Figura 3.9: Direccionamiento indirecto con desplazamiento

Direccionamiento Indirecto a Memoria de Código

Este modo de direccionamiento solo puede utilizar al apuntador Z, el cual queda implícito en el opcode. En la Figura 3.10 se ilustra el modo, ejemplos de instrucciones para acceso indirecto a memoria de código son:

```
LPM             ; R0 como registro a cargar
LPM   R3, Z
SPM             ; R1:R0 como registros a almacenar
```

También existe una carga indirecta de memoria de código con postincremento del apuntador Z, su comportamiento es similar al mostrado en la Figura 3.7, pero haciendo referencia a la memoria Flash.

Direccionamiento Inmediato

El direccionamiento inmediato es utilizado por 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.

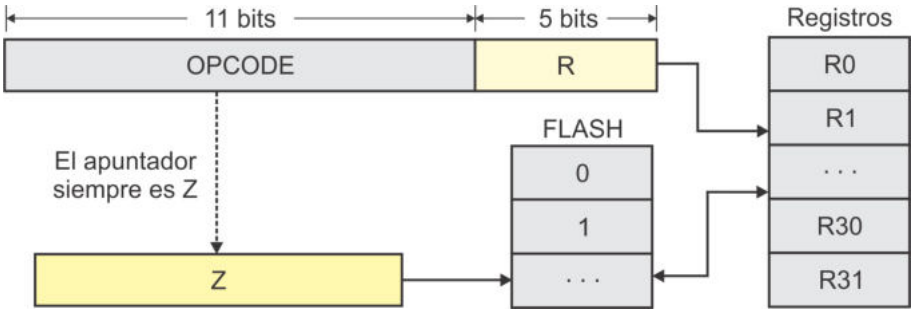


Figura 3.10: Direccionamiento indirecto a memoria de código

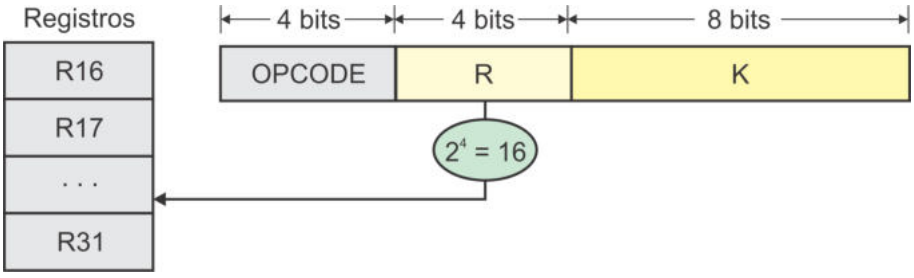


Figura 3.11: Direccionamiento inmediato

En la Figura 3.11 se muestra la organización de las instrucciones que utilizan este modo de direccionamiento. Se observa que al quedar disponibles únicamente 4 bits para definir el número de registro, se puede referenciar a 16 de ellos, es por eso que las instrucciones con constantes solo pueden emplear registros en el intervalo de R16 a R31. Para la constante se dispone de 8 bits, pueden utilizarse enteros sin signo, en un intervalo de 0 a 255, o números con signo, en un intervalo de -128 a 127. Ejemplos de instrucciones que utilizan direccionamiento inmediato son:

```
ANDI    R17, 0xF3
SUBI    R19, 10
LDI     R16, -5
```

Bifurcaciones con Direccionamiento Relativo

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.

Las bifurcaciones con direccionamiento relativo incluyen una constante que se suma al PC, la constante puede ser positiva o negativa para bifurcar hacia adelante o atrás de la instrucción actual, en realidad son relativas a PC + 1 porque el PC en forma automática se incrementa en 1. En la Figura 3.12 se ilustra el funcionamiento de las bifurcaciones relativas.

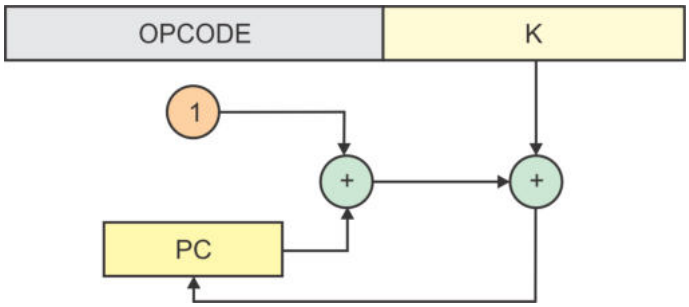


Figura 3.12: Bifurcaciones con direccionamiento relativo

Figura 3.13: Bifurcaciones con direccionamiento indirecto

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 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, el ensamblador calcula el valor de las constantes de acuerdo con la ubicación de las etiquetas. Existen bifurcaciones condicionales que pueden evitar la siguiente instrucción, estos “saltitos” también son relativos al PC, aunque el formato de la instrucción es más simple porque solo se omite la ejecución de una instrucción.

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. Dado que el PC también es un apuntador, en una bifurcación con este modo de direccionamiento básicamente se realiza un remplazo de direcciones, el contenido del apuntador Z se escribe en el PC, esto se ilustra en la Figura 3.13. Ejemplos de instrucciones que utilizan este modo de direccionamiento son:

```
IJMP
ICALL
```

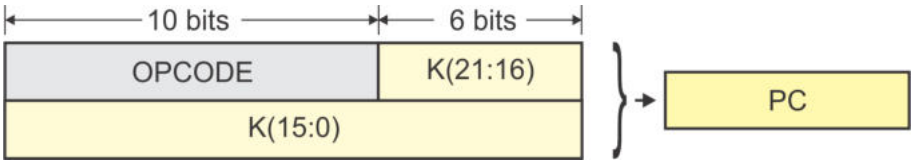


Figura 3.14: Bifurcaciones con direccionamiento absoluto

Bifurcaciones con Direccionamiento Absoluto

El direccionamiento es absoluto cuando se conoce la dirección destino de la bifurcación y esta se incluye en la 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 ATMega328P, no obstante, el formato de las instrucciones se dispone de esta manera para ser utilizado por otros miembros de la familia AVR que cuentan con un espacio mayor en su memoria de código. 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.

Es conveniente señalar que los programas escritos en lenguaje ensamblador no distinguen entre minúsculas y mayúsculas, el programador puede escribir código con su propio estilo. En cuanto a los comentarios, en el lenguaje solo se tienen comentarios de línea y estos inician con un punto y coma, no hay comentarios de bloque.

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.

Para contar con los elementos suficientes para desarrollar programas, en esta sección se revisan algunas de las directivas incluidas en el ensamblador AVR de la herramienta Microchip Studio, así como dos funciones muy utilizadas en los programas.

Las directivas se identifican por estar precedidas por un punto, mientras que una función se caracteriza por recibir un argumento.

En la Sección 3.5 se exponen 4 problemas 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 y funciones del ensamblador AVR.

Directiva INCLUDE

Esta directiva se utiliza para leer el código fuente de otro archivo, se coloca al inicio de un programa y generalmente es utilizada para incluir todas las definiciones relacionadas con un dispositivo particular. En los programas desarrollados para un ATMega328P se debe agregar:

```
.include <m328pdef.inc>    ; Definiciones para un ATMega328P
```

Con ello, todos los Registros I/O (normales y extendidos) pueden ser tratados por su nombre y no por su dirección, aunque en la versión más reciente de Microchip Studio no es necesario incluir algún archivo para que los nombres de los registros sean reconocidos.

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 y está orientada al uso de la memoria Flash. Es el segmento por omisión, no es necesario especificarla cuando un programa no va a utilizar 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 establecidos (archivo con extensión *hex*).
- **DSEG:** Señala el inicio de un segmento para los datos en la SRAM, es útil para reservar algunas localidades para un propósito especial. También puede haber múltiples segmentos de datos.
- **ESEG:** Indica que la siguiente información será guardada en memoria EEPROM, memoria destinada a variables que se deben conservar en ausencia de energía. Al ensamblar un programa con un segmento de EEPROM, se crea un archivo con extensión *eep*, este debe ser descargado al MCU para establecer

el contenido inicial de variables en EEPROM. La lectura y ajuste de estas variables se puede realizar con las rutinas descritas en la Sección 2.4.2.

Ejemplos del uso de estas directivas son:

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

        .CSEG                   ; Inicia un segmento de código
         LDI    R16, 0x25       ; R16 = 0x25
         STS    var1,R16        ; escribe en un espacio de SRAM
          . . .

const: .DB 0x02                 ; Constante en memoria Flash

        .ESEG                   ; Inicia un segmento de EEPROM
eevar1:.DB 0x3f                 ; Variable en memoria EEPROM
```

Las directivas `.BYTE` y `.DB` se describen en apartados posteriores.

Directiva DB y DW

Las directivas DB y DW sirven para definir constantes o variables en los segmentos de código y de 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, estos deben organizarse en una lista con los elementos separados por comas, de manera indistinta se pueden utilizar números en decimal, binario o hexadecimal. 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 ser empleada como referencia por las instrucciones de un programa. Si es una lista, con la etiqueta se tiene acceso al comienzo de la lista.

Directiva EQU

La directiva EQU Se utiliza para realizar definiciones que posteriormente se pueden usar en las instrucciones. Por ejemplo:

```
        .EQU  PortA = 0x25

        .CSEG                           ; Inicia el segmento de código
```

```
        CLR   R2                ; Limpia al registro 2
        OUT   PortA, R2         ; Escribe PortA
```

La biblioteca `m328pdef.inc` hace un uso extensivo de la directiva EQU, al hacer corresponder los nombres de los Registros I/O y sus bits con sus direcciones y posiciones. Al agregar la biblioteca con la directiva INCLUDE, en los programas se utilizan nombres en lugar de números.

Directiva ORG

Esta directiva se utiliza para ubicar la información subsecuente en cualquiera de los segmentos de memoria. En la SRAM o EEPROM indica la dirección a partir de la cual se colocan las variables y en la Flash puede ubicar constantes o código. 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   0x000           ; Código ubicado en la dirección 0x00
        RJMP    inicio

        .ORG    0x010
inicio:                         ; Código ubicado en la dirección 0x10
        MOV     R1, R2
         . . . .
```

La directiva ORG se emplea principalmente para organizar a las instrucciones de un programa cuando se manejan interrupciones, porque ubica al código en la dirección que corresponda con su vector de interrupción, como se describió en la Sección 2.6.

Funciones HIGH y LOW

Las funciones HIGH y LOW se utilizan para separar constantes de 16 bits, con HIGH se obtiene la parte alta y con LOW la parte baja. Por ejemplo, para cargar la constante 578 en los registros R27 y R26:

```
        LDI   R27, HIGH(578)    ; El valor de la constante se identifica
        LDI   R26, LOW(578)     ; claramente
```

Las funciones HIGH y LOW se traducen en el momento en que se ensambla un programa, sin generar código adicional. Un uso importante de estas funciones es la referencia a direcciones que corresponden con etiquetas de un programa, para después hacer accesos mediante direccionamiento indirecto. Por ejemplo, para posicionar al apuntador Z (R31:R30) al comienzo de una tabla y obtener una constante, se tiene el código:

```
        LDI   R31, HIGH(tabla)
        LDI   R30, LOW(tabla)
        LPM   R16, Z+
        . . . .
tabla: .DB    0x01,0x02,0x03,0x04
```

Cuando un programa es escrito se utilizan etiquetas para simplificar el proceso, su valor para el código máquina se obtiene en el momento en que el programa se ensambla. Con el uso de las funciones HIGH y LOW automáticamente se pueden obtener los valores de las etiquetas para posicionar a los apuntadores de manera correcta.

Directiva BYTE

La directiva BYTE sirve para reservar uno o más bytes en SRAM o EEPROM para el manejo de variables, la cantidad de bytes debe especificarse y el espacio no es inicializado, solo 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 directo a var1
        LDI   R30, LOW(var2)   ; Z apunta a var2
        LDI   R31, HIGH(var2)
        ST    Z, R1            ; Acceso a var2 con el apuntador Z
```

Directiva DEF

La directiva DEF establece un nombre simbólico a un registro, para dar claridad a los programas. El nombre puede ser empleado en cualquier parte de un programa y un registro puede tener más de un nombre. En el siguiente ejemplo se muestra parte de un contador, el uso de la directiva ayuda a comprender el papel de cada registro:

```
.DEF   limite = R16
.DEF   conta = R20
        . . .
        LDI    limite, 10
        CLR    conta
loop:   CP     conta, limite
        BRSH   fin
        OUT    PORTB, conta
        INC    conta
        RJMP   loop
fin:    . . .
```

3.4. Programación en Lenguaje C

El entorno más adecuado para el desarrollo de programas depende de la versión del sistema operativo, para Windows 7 o versiones anteriores es conveniente emplear un IDE conocido como **AVR Studio** porque ocupa muy pocos recursos. **AVR Studio** únicamente incluye al programa ensamblador (AVRASM), pero proporciona las facilidades para enlazarse con compiladores de lenguaje C desarrollados por fuentes diferentes a Microchip. Con el compilador instalado, desde el mismo entorno es posible la edición de programas, la invocación del compilador con exhibición de resultados, así como la simulación o depuración. Uno de estos compiladores es el avr-gcc, incluido en una suite conocida como WinAVR64, la cual es parte del proyecto GNU. Después de instalar la suite, el compilador es llamado automáticamente desde el AVR Studio cada vez que se requiere y su uso queda transparente al programador.

En versiones posteriores de Windows, como Windows 8 o 10, la herramienta de desarrollo más adecuada es el **Microchip Studio**, una suite que permite trabajar con microcontroladores AVR de 8 bits y con otros dispositivos de la empresa Microchip. El IDE de Microchip Studio ya incluye al compilador para C/C++ y es totalmente compatible con el estándar ANSI C, se pueden emplear 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, las soluciones muestran la estructura general que deben seguir los programas escritos en este lenguaje.

3.4.1. Tipos de Datos

Para la programación de los microcontroladores AVR se pueden usar 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. Se observa que diferentes tipos de datos tienen el mismo comportamiento, por ejemplo `float` y `double` prácticamente hacen referencia al mismo tipo de datos, la inclusión de ambos es para mantener compatibilidad con el estándar ANSI C. Por otro lado, las variables de los tipos `char`, `unsigned char` o `signed char` pueden asignarse directamente con los Registros I/O.

Al trabajar con un microcontrolador se debe considerar el limitado espacio de la memoria y evitar agotarlo con un mal manejo de los tipos de datos, por ejemplo, si un ciclo repetitivo tiene pocas iteraciones, es mejor usar una variable del tipo `unsigned char` en lugar de un `int`, ya que no solo es un byte extra, sino que con un `int` se ejecutan operaciones de 16 bits que requieren más instrucciones de bajo nivel.

El entorno de Microchip Studio cuenta con una serie de redefiniciones para los tipos

de datos que se utilizan con mayor frecuencia, estas son:

```
typedef signed char int8_t          // 8 bits
typedef unsigned char uint8_t
typedef signed int int16_t          // 16 bits
typedef unsigned int uint16_t
typedef signed long int int32_t     // 32 bits
typedef unsigned long int uint32_t
```

Tabla 3.25: Tipos de datos

Tipo	Tamaño (bits)	Rango
char	8	$[-128, 127]$
unsigned char	8	$[0, 255]$
signed char	8	$[-128, 127]$
int	16	$[-32768, 32767]$
short int	16	$[-32768, 32767]$
unsigned int	16	$[0, 65535]$
signed int	16	$[-32768, 32767]$
long int	32	$[-2147483648, 2147483647]$
unsigned long int	32	$[0, 4294867295]$
signed long int	32	$[-2147483648, 2147483647]$
float	32	$[\pm 1.175 \times 10^{-38}, \pm 3.402 \times 10^{+38}]$
double	32	$[\pm 1.175 \times 10^{-38}, \pm 3.402 \times 10^{+38}]$

Al programar se pueden utilizar los tipos de datos del estándar de C o las redefiniciones de Microchip Studio, esto no afecta el comportamiento de un programa. La ventaja de las redefiniciones es que utilizan nombres más cortos.

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), en este libro 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 y como resultado proporcionan un valor `falso` o `verdadero`. Estos operadores se utilizan para crear expresiones complejas al combinar expresiones lógicas simples. En lenguaje C cualquier valor diferente de 0 es interpretado como `verdadero` y 0 corresponde con `falso`.

En la Tabla 3.27 se muestran los operadores para el manejo de bits, estos trabajan sobre datos de los tipos `char`, `int` o `long` 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 3.26: Operadores lógicos para expresiones

Operador	**Símbolo**
AND	&&
OR	||

Tabla 3.27: Operadores para el manejo de bits

Operador	**Símbolo**
Complemento a 1	$\sim$
Desplazamiento a la izquierda	$\ll$
Desplazamiento a la derecha	$\gg$
AND	&
OR	|
OR Exclusivo	$\wedge$

En el siguiente segmento de código se ilustra el uso de los operadores para el manejo de bits, al revisar el estado de un botón conectado en PB2 (con el botón presionado se introduce un 0 lógico):

```
if( !(PINB & 1 << 2) ) {
    . . .           // Acciones para cuando el botón fue presionado
}
```

3.4.3. Tipos de Memoria

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

Datos en SRAM

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

```
uint8_t  x, y;
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 el almacenamiento de 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 y su contenido debe hacer referencia a algún objeto de SRAM. Por ejemplo:

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

pcad = cadena;                    // pcad apunta a la cadena
```

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 incluyen cadenas de texto o tablas de búsqueda.

Para que el compilador dirija las constantes hacia la Flash, estas deben identificarse con la palabra reservada `const` e incluir al atributo `PROGMEM`, definido en la biblioteca `pgmspace.h`, que es parte de Microchip Studio. Ejemplos de declaraciones de constantes en Flash son:

```
const char     cadena[] PROGMEM = "Cadena con un mensaje constante";
const uint8_t  tabla[]  PROGMEM = { 0x24, 0x36, 0x48, 0x5A, 0x6C };
```

La lectura de los datos de la memoria Flash requiere de funciones dedicadas, de lo contrario, al compilar se agrega una secuencia de código que realiza copias de Flash a SRAM y se pierde el ahorro del espacio es SRAM. Las funciones para la lectura de constantes en Flash también están incluidas en la biblioteca `pgmspace.h`, algunas de ellas son:

```
pgm_read_byte(address);     // Lee 1 byte
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, que se obtiene con el operador &, por ejemplo, la lectura del i-ésimo dato de la tabla de constantes se realiza con la expresión:

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

Un apuntador a la memoria Flash debe declararse como `PGM_P`, este tipo de datos está definido en la biblioteca `pgmspace.h`, la cual 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
```

El Ejemplo 3.3 de la Sección 3.5 muestra cómo declarar y leer datos constantes de la memoria Flash.

Datos en EEPROM

Otra consideración se debe tener para aquellas variables cuyo contenido se requiera conservar aun en ausencia de energía, no se pueden manejar en SRAM porque es memoria volátil, el espacio adecuado es la EEPROM.

Para ubicar una variable en la EEPROM, su declaración debe estar precedida con el atributo `EEMEM`, definido en la biblioteca `eeprom.h`, que también es parte del entorno de Microchip Studio. Este atributo provoca que las declaraciones siguientes sean direccionadas a la EEPROM, las declaraciones deben ser globales y es conveniente situarlas antes de las 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 uint16_t contador = 0x1234;           // Un dato de 16 bits
EEMEM uint8_t 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 . . .
```

Al compilar un programa con declaraciones para la EEPROM, se crea un archivo con extensión *eep*, este debe ser descargado al MCU para establecer el contenido inicial de estas variables.

El acceso a los datos en la EEPROM se realiza por medio de los registros EEAR, EEDR y EECR, para ello se pueden emplear las funciones descritas en la Sección 2.4.2 o acondicionarlas para involucrar la interrupción por fin de escritura en EEPROM. Además, la biblioteca `eeprom.h` cuenta con diversas funciones para el acceso a la EEPROM, algunas de ellas son:

- **eeprom_read_byte:** Lee un byte de la EEPROM, en la dirección que recibe como argumento.
- **eeprom_read_word:** Lee una palabra de 16 bits 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.
- **eeprom_write_word:** Escribe una palabra de 16 bits en la EEPROM, como argumentos recibe la dirección y el dato.

En la siguiente secuencia de código se lee la variable contador de la EEPROM (de 16 bits), se incrementa en 1 y se vuelve a escribir:

```
uint16_t  aux;

   aux = eeprom_read_word(&contador);     // Lee de la EEPROM
   aux = aux + 1;                         // Incrementa en 1
   eeprom_write_word(&contador, aux);     // Escribe en la EEPROM
```

La biblioteca también contiene funciones para el acceso a datos de 32 bits, así como para el manejo de bloques de memoria.

3.5. Programas de Ejemplo

En esta sección se muestran 4 ejemplos de pequeños sistemas para familiarizarse con la programación de los microcontroladores AVR, tanto en ensamblador como en lenguaje C. Los ejemplos se probaron en el ATMega328P pero pueden funcionar en otros miembros de la familia AVR, siempre que tenga los puertos requeridos en cada caso. Cuando se crea un proyecto nuevo en Microchip Studio, se debe especificar el lenguaje a emplear y el dispositivo destino. El Apéndice C contiene un tutorial que muestra los pasos a seguir para 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", un programa sin algún propósito práctico pero sirve para mostrar la estructura general de los programas, este es el objetivo del primer ejemplo.

Ejemplo 3.1 - Parpadeo de un LED
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 % (0.5 S encendido y 0.5 S apagado). En la Figura 3.15(a) se muestra el hardware requerido y en la Figura 3.15(b) un diagrama de flujo con el comportamiento esperado.

En la Figura 3.15(b) se observa que el programa nunca termina, después de algunas configuraciones iniciales permanece en un lazo infinito, esto es normal en sistemas basados en microcontroladores, por ello, Arduino utiliza la estructura `Setup-Loop`, para separar las configuraciones iniciales del lazo infinito.

En la solución con lenguaje C se requiere de una función para el retardo porque se desconoce la duración de cada instrucción, Microchip Studio proporciona la biblioteca `delay.h` que contiene la función `_delay_ms(double _ms)`. La función recibe como argumento un número en punto flotante que indica la cantidad de milisegundos de espera. Puesto que la función `_delay_ms(double _ms)` se basa en iteraciones, el com-

pilador debe conocer la frecuencia a la que está operando la CPU, en el programa se establece este valor con una definición inicial.

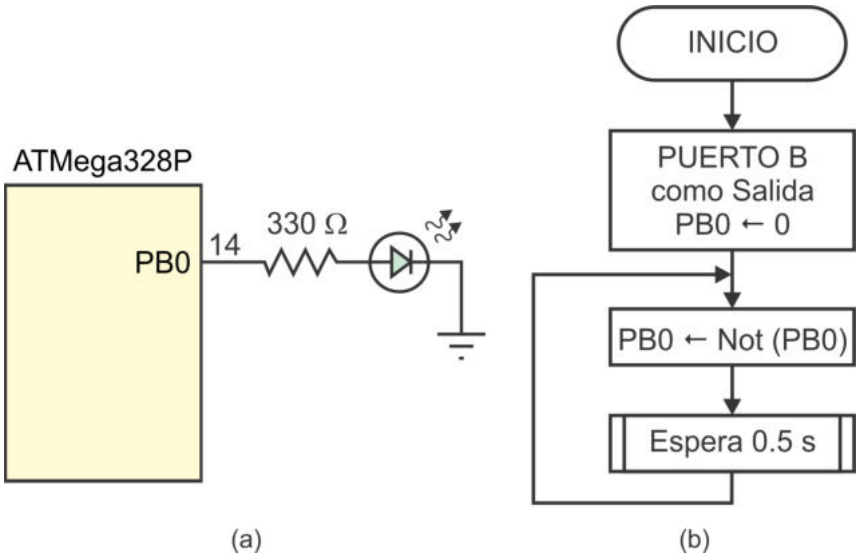


Figura 3.15: Parpadeo de un LED (a) hardware y (b) comportamiento esperado

El código C con la solución del ejercicio es:

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

#include <util/delay.h>      // Biblioteca para retardos
#include <avr/io.h>          // Definición de Registros I/O

int main() {

  DDRB = 0xFF;               // Puerto B como salida
  PORTB = 0x00;              // Valor inicial en la salida
  while(1) {                 // Lazo infinito
     PINB = 0x01;            // Conmuta PB0 sin modificar otros bits
     _delay_ms(500);         // Retardo de 500 ms
  }
}
```

El estándar de C establece que la función `main` debe ser del tipo `int` a pesar de que no haya valores de retorno.

Para la conmutación de la salida se aprovecha la estructura del hardware de las terminales (ver Sección 2.5), cuando un puerto es salida sus terminales se conmutan si se escribe un 1 lógico en su registro `PIN`.

En la solución con ensamblador se debe realizar una rutina para el retardo, para ello se considera que el ATMega328P trabaja con un oscilador interno de 1 MHz, se calcula el número de iteraciones requeridas y se organizan ciclos repetitivos.

En ensamblador se conoce con precisión la cantidad de ciclos de reloj que tarda cada instrucción, pero por el tamaño de los registros es necesario anidar diferentes ciclos para conseguir un retardo de 500 ms, que corresponde con 500 000 ciclos de reloj de 1 us.

El código en ensamblador con la solución del ejercicio es:

```
      .include <m328pdef.inc>  ; Biblioteca con definiciones

       LDI    R16, 0xFF
       OUT    DDRB, R16         ; Puerto B como salida
       CLR    R16
       OUT    PORTB, R16        ; Valor inicial para la salida

Lazo:                           ; Lazo infinito
       SBI    PINB, 0           ; Conmuta PB0 sin modificar otros bits
       RCALL  Espera_500mS
       RJMP   Lazo

;
; Una rutina de espera se revisa del lazo interno al externo
; 500 ms = 500 000 us = 2*( 250*( 250*4 us ) )
;

Espera_500mS:
       PUSH   R18
       PUSH   R17
       PUSH   R16
       LDI    R18, 2
et3:   LDI    R17, 250
et2:   LDI    R16, 250
et1:   NOP                      ; Itera 250 veces, 4 us por iteración
       DEC    R16               ; 250*4 us = 1000 us = 1 ms
       BRNE   et1               ; Evalúa la bandera de cero
                                ; brinca si R16 no ha llegado a cero
       DEC    R17
       BRNE   et2               ; 1 ms*250 = 250 ms
       DEC    R18
       BRNE   et3               ; 250 ms*2 = 500 ms
       POP    R16
       POP    R17
       POP    R18
       RET
```

En la rutina de espera se respaldan los registros R16, R17 y R18 en la pila, para que no pierdan su contenido cuando la rutina termina. Estas acciones no son necesarias en este ejemplo porque los registros no tienen otra tarea en el programa principal, sin embargo, es una buena práctica de programación realizar este tipo de respaldos para poder reutilizar las rutinas en otros programas.

3.5.2. ALU de 4 Bits

Una ALU realiza diferentes operaciones ariméticas o lógicas con dos operandos, por medio de algunos bits de selección se determina la operación. La solución en lenguaje C de este ejemplo utiliza una esctructura `switch-case` para garantizar que todos los casos se atienden con el mismo tiempo de respuesta. El mismo comportamiento se debe tener en ensamblador y para conseguirlo, se utiliza una tabla de saltos.

Ejemplo 3.2 - ALU de 4 Bits
Construya una ALU de 4 bits utilizando un ATMega328P, los operandos se deben leer del Puerto B (nibble bajo para el operando A y nibble alto para el operando B), la operación se determina con los 3 bits menos significativos del Puerto C y el resultado se debe generar en el Puerto D. En la Figura 3.16 se muestran las entradas y salidas de la ALU, y en la Tabla 3.28 están las combinaciones para seleccionar la operación.

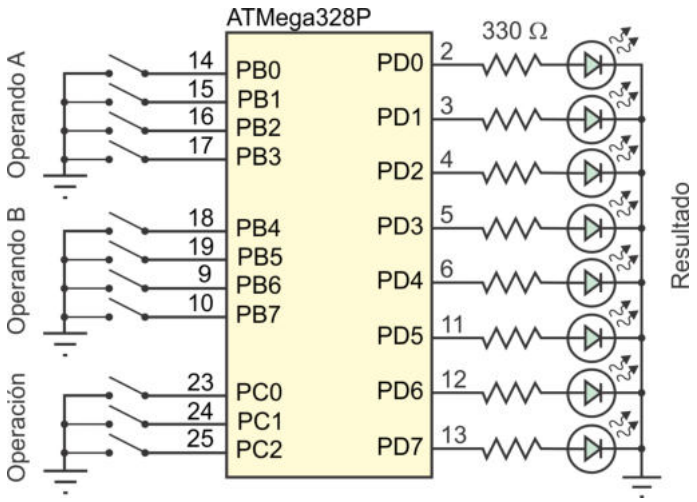


Figura 3.16: ALU de 4 bits

Tabla 3.28: Comportamiento de la ALU

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

El comportamiento esperado para la ALU se muestra en el diagrama de flujo de la Figura 3.17. Los 2 operandos se obtienen del mismo puerto por lo que deben separarse en variables utilizando operaciones lógicas.

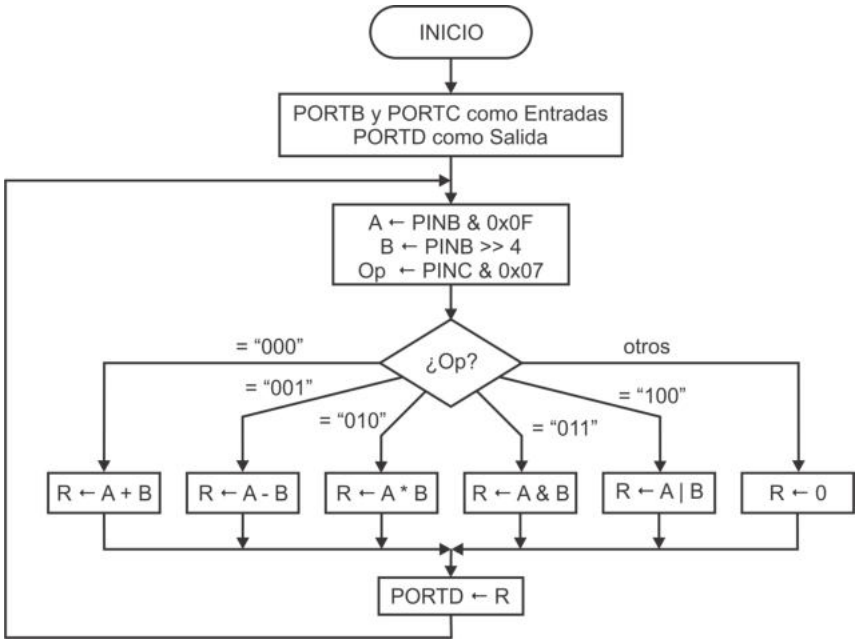


Figura 3.17: Comportamiento de la ALU de 4 bits

Para el programa en lenguaje C lo natural es el uso de una estructura `switch-case`, donde la selección de cada caso depende de los 3 bits menos significativos del Puerto C, se utiliza una máscara para que no influyan los bits no utilizados. La solución en lenguaje C es:

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

int main() {
uint8_t  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;          // Lee el operando A
     B = PINB >> 4;            // Lee el operando B
     Op = PINC & 0x07;         // Lee la operación

     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
    }
}
```

El programa no considera la posibilidad de acarreos porque los datos son de 4 bits, el resultado sin problema alcanza en 8 bits en las 3 operaciones aritméticas.

En la solución con ensamblador se utiliza una tabla de saltos para implementar la estructura `switch-case`, una vez definida la tabla, con direccionamiento indirecto se accede al caso que corresponde y la tabla redirecciona para que se ejecute el código correcto. El programa en ensamblador es:

```
  .include <m328pdef.inc>
  .def  A = R20
  .def  B = R21
  .def  Op = R22
  .def  R = R23

   CLR   R16
   OUT   DDRB, R16        ; Puertos B y C como entradas
   OUT   DDRC, R16
   SER   R16
   OUT   PORTB, R16       ; Resistores de pull-up
   OUT   PORTC, R16
   OUT   DDRD, R16        ; Puerto D como salida

loop:                     ; Lazo infinito
   IN    A, PINB          ; Lee el operando A
   ANDI  A, 0x0F
   IN    B, PINB          ; Lee el operando B
   ANDI  B, 0xF0
   SWAP  B
   IN    Op, PINC         ; Obtiene la operación
   ANDI  Op, 0x07         ; Sólo 3 bits son válidos
   CPI   Op, 5            ; Observa si es un caso válido
   BRLO  valido
   LDI   Op, 5            ; con 5 el caso ya es válido (default)
valido:
   LDI   R30, LOW(tabla) ; Z apunta al inicio de la tabla de saltos
   LDI   R31, HIGH(tabla)
   ADD   R30, Op          ; Suma el caso detectado
   BRNE  no_carry
   INC   R31              ; Se considera un posible acarreo
no_carry:
   IJMP                   ; Salta a la tabla + caso
```

```
tabla:
   RJMP  case_0          ; Tabla de saltos
   RJMP  case_1
   RJMP  case_2
   RJMP  case_3
   RJMP  case_4
   RJMP  default

case_0:                  ; Suma
   MOV   R, A            ; R = A
   ADD   R, B            ; R = R + B = A + B
   RJMP  salir
case_1:                  ; Resta
   MOV   R, A            ; R = A
   SUB   R, B            ; R = A - B
   RJMP  salir
case_2:                  ; Producto
   MUL   A, B            ; El resultado queda en R1:R0
   MOV   R, R0           ; ubica el resultado
   RJMP  salir
case_3:                  ; AND lógica
   MOV   R, A            ; R = A
   AND   R, B            ; R = A & B
   RJMP  salir
case_4:                  ; OR lógica
   MOV   R, A            ; R = A
   OR    R, B            ; R = A | B
   RJMP  salir
default:
   CLR   R               ; Por default R = 0
salir:                   ; fin del switch-case
   OUT   PORTD, R        ; Genera la salida
   RJMP  loop;
```

En el código en ensamblador se observa que 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, dejando este valor para el caso por defecto.

La instrucción `RJMP salir` corresponde al `break` de lenguaje C, si se omite, después de ejecutar el código que corresponde con el caso solicitado se ejecutará el código de los casos posteriores, hasta encontrar un `RJMP salir`, este comportamiento también es el esperado para una estructura `switch-case` de alto nivel.

3.5.3. Contador con Salida en 7 Segmentos

En este ejemplo se muestra como sondear si se ha presionado un botón para modificar el valor de un contador y como manejar un arreglo de datos constantes para los códigos de un *display* de 7 segmentos. El arreglo de constantes se ubica en memoria

Flash y se tiene el mismo tiempo de acceso para todas las combinaciones, es decir, obtener el código de la F requiere el mismo tiempo que obtener el código del 0. En lenguaje C el manejo de arreglos es directo y sin complicaciones, pero en ensamblador se establece una tabla de datos constantes y su acceso se realiza de manera indirecta.

Ejemplo 3.3 - Contador con Salida en 7 Segmentos
Desarrolle un programa para un contador de 0 a 15 con salida a un display de 7 segmentos, conectado en el Puerto D de un ATMega328P. La cuenta se debe incrementar al presionar un botón conectado en PB0. En la Figura 3.18 se muestra el hardware requerido.

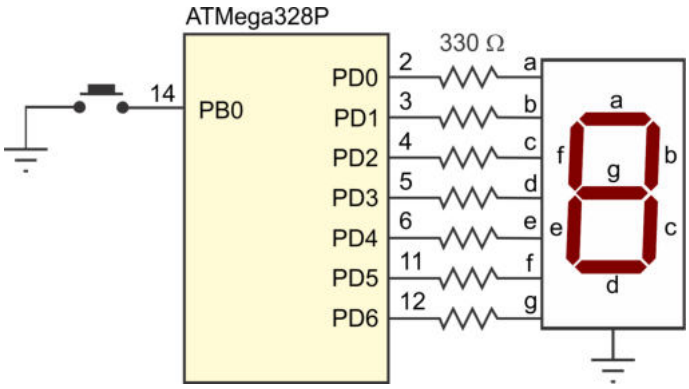


Figura 3.18: Contador con salida en 7 segmentos

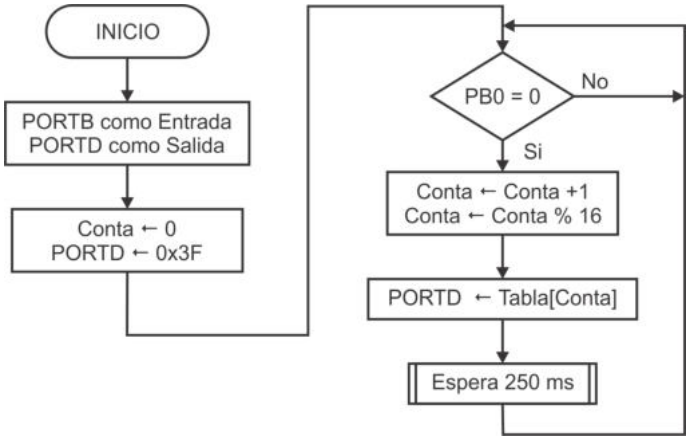


Figura 3.19: Comportamiento esperado en el contador de eventos

En la Figura 3.19 se muestra el comportamiento esperado, algunos aspectos a considerar son:

- El botón introduce un 0 lógico al ser presionado, con el resistor de *pull-up* habilitado se garantiza un 1 lógico cuando el botón está abierto.

- El arreglo `Tabla` debe contener los códigos de 7 segmentos, considerando un display de cátodo común.
- Se inserta un retardo después de detectar que el botón fue presionado para evitar que el contador realice incrementos en exceso porque las operaciones del MCU están en el orden de microsegundos.
- El contador debe limitarse a números entre 0 y 15, para no solicitar a la tabla un dato que no existe.

El programa en lenguaje C con la solución del ejemplo es:

```
#define   F_CPU   1000000UL

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

const uint8_t tabla[] PROGMEM = { 0x3F, 0x06, 0x5B, 0x4F, 0x66, 0x6D,
                                  0x7D, 0x07, 0x7F, 0x67, 0x77, 0x7C,
                                  0x39, 0x5E, 0x79, 0x71          };

int main() {
uint8_t conta;

  DDRB = 0x00;                          // Entrada para el botón
  PORTB = 0xFF;                         // Pull-up
  DDRD = 0xFF;                          // Salida para el display

  conta = 0;                            // El contador inicia en 0
  PORTD = 0x3F;

  while(1) {
    if(!(PINB & 0x01) ) {               // Botón presionado
      conta = conta + 1;                // Incrementa el contador
      conta = conta % 16;               // Asegura que está entre 0 y 15
      PORTD = pgm_read_byte(&tabla[conta]);
      _delay_ms(250);
    }
  }
}
```

En lenguaje ensamblador, la tabla de constantes se incorpora en la memoria de código con la directiva `DB` y el acceso a la misma se realiza con el apuntador `Z`, usando direccionamiento indirecto, el código es el siguiente:

```
        .include <m328pdef.inc>
        .def   Conta = R20

        CLR    R16
        OUT    DDRB, R16              ; Entrada
        SER    R17
```

```
        OUT   PORTB, R17              ; Pull-up
        OUT   DDRD, R17               ; Salida
        CLR   Conta
        LDI   R16, 0x3F
        OUT   PORTD, R16

LOOP:   SBIC  PINB, 0
        RJMP  LOOP
        INC   Conta
        ANDI  Conta, 0x0F

        LDI   R30, LOW(tabla << 1)    ; Z apunta al inicio de la tabla
        LDI   R31, HIGH(tabla << 1)
        ADD   R30, Conta              ; Z = Z + Conta
        BRCC  s1
        INC   R31

s1:     LPM   R17, Z                  ; R17 = tabla[Z]
        OUT   PORTD, R17
        RCALL delay_250ms
        RJMP  LOOP

; Para el retardo se deben anidar dos contadores, por la relación:
;       250 mS = 250 000uS = 250*( 250*4 uS )
delay_250mS:
        PUSH  R16           ; Respaldo en la pila
        PUSH  R17
        LDI   R17, 250
et2:    LDI   R16, 250
et1:    DEC   R16           ; 250 veces con 4 uS por iteración
        NOP                 ; 250*4 uS = 1000 uS = 1 mS
        BRNE  et1           ; Evalúa la bandera de cero
                            ; brinca si no hay bandera de cero
        DEC   R17
        BRNE  et2           ; 1 mS*250 = 250 mS
        POP   R17           ; Recupera de la pila
        POP   R16
        RET

tabla: .db 0x3F, 0x06, 0x5B, 0x4F, 0x66, 0x6D, 0x7D, 0x07
       .db 0x7F, 0x67, 0x77, 0x7C, 0x39, 0x5E, 0x79, 0x71
```

En el código se observa que la constante `tabla` tiene un desplazamiento a la izquierda cuando se posiciona al apuntador `Z` y equivale a una multiplicación por 2. La memoria está organizada en palabras de 16 bits por lo que la etiqueta `tabla` hace referencia a la dirección de una palabra, sin embargo, la instrucción `LPM` requiere la dirección de un byte para realizar la transferencia correcta, esta es la causa por la que la multiplicación por 2 es necesaria. En general, cuando se manejan tablas con constantes de 1 byte, el número de elementos debe ser par para no desalinear el código posterior.

En el Ejemplo 3.2 no fue necesaria la multiplicación por 2 porque Z apunta a una tabla de saltos y en ese caso, para el comienzo de la tabla, no se requiere de una dirección por bytes.

3.5.4. Máquina de Estados

En el diseño de sistemas digitales, para algunos problemas la solución inmediata se puede describir a través de una máquina de estados finitos (MEF). La traducción de una MEF a lenguaje C suele complicarse porque difiere significativamente de un diagrama de flujo, sin embargo, si se considera que el lazo infinito se ejecuta en forma periódica, se puede tratar a cada iteración como un ciclo de ejecución de la MEF. En cada ciclo de ejecución de una MEF se deben generar las salidas según el estado actual y revisar las entradas para determinar el estado siguiente. Este es el objetivo del presente ejemplo, evaluar cómo implementar una MEF en un MCU, la solución solo se presenta en lenguaje C.

Ejemplo 3.4 - Máquina de Estados
Desarrolle un programa en lenguaje C para un ATMega328P que realice el control para una puerta automática, suponga que el sistema tiene 3 sensores, un sensor de personas (SP), un detector para indicar que la puerta está completamente abierta (TA) y otro que indica si la puerta está totalmente cerrada (TC). A través de dos salidas el MCU debe manejar un motor con el apoyo de un Puente H para poder abrir o cerrar la puerta. En la Figura 3.20(a) se muestra la conexión de los sensores y el actuador con el MCU, y en la Figura 3.20(b) se puede ver el comportamiento esperado, expresado mediante una MEF.

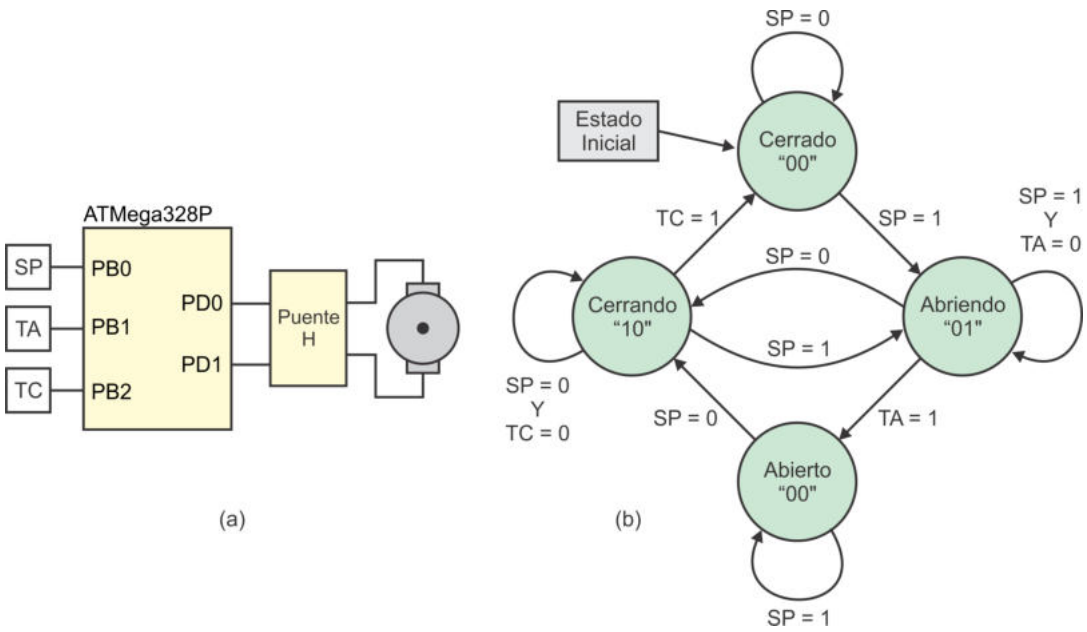


Figura 3.20: Circuito y comportamiento del control para una puerta automática

El programa en lenguage C que describe el comportamiento de la MEF de la Figura 3.21 es:

```
#define  F_CPU  1000000UL      // Frecuencia de trabajo
#define  SP     0x01           // Ubicación de los sensores
#define  TA     0x02
#define  TC     0x04

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

int  main() {
enum { cerrado, abriendo, abierto, cerrando } estado;

  DDRB = 0x00;                 // Entradas para los sensores
  DDRD = 0xFF;                 // Salidas para el motor
  estado = cerrado;            // Estado inicial
  PORTD = 0x00;                // Sin movimiento

  while(1) {
    switch(estado) {
      case cerrado : PORTD = 0x00;
                     if(PINB & SP)              // SP = 1
                       estado = abriendo;
                     break;
      case abriendo: PORTD = 0x01;
                     if(!(PINB & SP))           // SP = 0
                       estado = cerrando;
                     else if(PINB & TA)         // TA = 1
                       estado = abierto;
                     break;
      case abierto : PORTD = 0x00;
                     if(!(PINB & SP))           // SP = 0
                       estado = cerrando;
                     break;
      case cerrando: PORTD = 0x02;
                     if(PINB & SP)              // SP = 0
                       estado = abriendo;
                     else if(PINB & TC)         // TC = 1
                       estado = cerrado;
                     break;
      }
    _delay_ms(5);
  }
}
```

En el código se utiliza una enumeración para dar claridad a la máquina de estados, se declara la variable `estado` con el tipo de la enumeración y su valor se ajusta en cada iteración. En cada estado se genera la salida que le corresponde y se revisan los sensores para determinar el estado siguiente, se deja un retardo de 5 ms para mantener estables las salidas.

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 el lenguaje a emplear es conveniente hacer un análisis buscando la mejor solución.

En ocasiones se dice que es mejor programar en ensamblador porque tiene una relación directa con el código máquina que se genera, el argumento es acertado porque al programar en alto nivel 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, un programa más compacto no siempre tendrá el mejor desempeño, 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. En un problema con una complejidad mediana o alta, la programación en alto nivel produce un código fuente más compacto y menos confuso, reduciendo con ello la posibilidad de cometer errores. También ante la necesidad de caracteísticas específicas, como operaciones o funciones con números en punto flotante, es mejor dejar que el compilador genere el código máquina que desarrollar rutinas propias, además de invertir mucho tiempo en el desarrollo no se garantiza un resultado óptimo.

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 evitar el uso de lenguaje C en la mayoría de aplicaciones.

El lenguaje ensamblador es necesario en las siguientes situaciones:

1. 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.

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

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 se facilita el empaquetamiento de datos para comprimir la información.

Si el tamaño de la memoria de código de un MCU es considerable (8 kbytes o más), no es necesario desarrollar una aplicación completa en ensamblador. En lenguaje C

es posible insertar código ensamblador en las secciones donde es requerido, utilizando la función `asm()`, la cual puede recibir hasta 4 argumentos, con la siguiente sintaxis:

```
asm(código :[operandos_salida :operandos_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:** En este argumento se indican los registros que se utilizarán en el código sin ser operandos de entrada o salida.

En los siguientes ejemplos se muestran llamadas a la función `asm` sin operandos de entrada o salida:

```
asm("NOP");              // No operación, tarda 1 ciclo de reloj
asm("SBI  0x05,0");      // Pone en alto al bit 0 de PORTB
asm("CBI  0x05,0");      // Pone en bajo al bit 0 de PORTB
asm("SEI  \n\t"          // Habilita las interrupciones y
    "CLC");              // limpia la bandera de acarreo
```

La instrucción del primer ejemplo solo sirve para perder 1 ciclo de reloj y no afecta a los recursos del MCU. En el segundo y tercer ejemplo se hace referencia al Registro `PORTB` por su dirección, se requiere de una especificación de conversión para utilizar su nombre. En el último ejemplo se tienen dos instrucciones en la misma proposición, se inserta al carácter de nueva línea y al tabulador porque en ensamblador se escribe una instrucción por línea.

Cada operando de entrada o salida es descrito por una cadena de restricción seguida por una expresión de lenguaje C entre paréntesis. El compilador AVR-GCC reconoce 23 caracteres de restricción, de los cuales, 3 son de los más empleados:

- **d** - Para referir a un registro superior, es decir, de R16 a R31, solo con estos registros se hacen operaciones con constantes.
- **r** - Hace referencia a cualquier registro (R0 a R31).
- **I** - Indica que se empleará una constante positiva de 6 bits, para obtener la dirección de un Registro I/O se puede emplear la función `_SFR_IO_ADDR()`.

En la función que se describe a continuación se muestra como emplear los operandos de entrada o salida:

```
void seguidor(void) {
uint8_t  aux;
```

```
  asm volatile(
      "in    %0,%1    \n\t"
      "out   %2,%0    \n\t"
      :"=r"(aux)
      :"I"(_SFR_IO_ADDR(PIND)), "I"(_SFR_IO_ADDR(PORTB))
  );
}
```

La función básicamente lee un dato del Puerto D y lo copia al Puerto B, con %0 se hace referencia a "=r"`(aux)`, %1 es para "I"`(_SFR_IO_ADDR(PIND))` y %2 se relaciona con "I"`(_SFR_IO_ADDR(PORTB))`. La variable `aux` de alto nivel se puede asociar con cualquier registro porque en el código ensamblador no se realizan operaciones con constantes. Si después de la lectura en PIND se incluye la instrucción `andi %0, 0x0F`, se debe garantizar que %0 haga referencia a un registro superior y para ello se debe utilizar "=d"`(aux)`.

Los compiladores utilizan diferentes estrategias de optimización, una de ellas consiste en remover expresiones que no tienen efecto en el código posterior, en ese sentido, el compilador no puede determinar el efecto de la llamada a la función `asm()` y la puede anular de la compilación. Para evitar esta situación se utiliza al atributo *volatile*, su inclusión evita que el compilador optimice al argumento de la función `asm()` y el código ensamblador se ejecuta sin problema dentro del código C.

3.7. Manejo de Registros I/O con Arduino

Las funciones de Arduino para la configuración y acceso a los puertos hacen referencia a terminales individuales, cuando se requiere el manejo de varios bits o de un puerto completo se presenta un problema porque las lecturas realizadas con `digitalRead()` deben agruparse para que se puedan interpretar en conjunto y los datos de salida se deben separar por bits para mostrarlos con `digitalWrite()`. En esos casos, lo mejor es utilizar los Registros I/O para el manejo de los puertos dentro de un *sketch* de Arduino.

El *sketch* que se muestra a continuación resuelve el Ejemplo 3.3, en donde se desarrolló un contador de eventos con salida en 7 segmentos. El problema ahora se resuelve con un *sketch* para una tarjeta Arduino, pero el acceso al Puerto D se hace a través de sus Registros I/O, el código resultante es:

```
const uint8_t tabla[] PROGMEM = { 0x3F, 0x06, 0x5B, 0x4F, 0x66, 0x6D,
                                  0x7D, 0x07, 0x7F, 0x67, 0x77, 0x7C,
                                  0x39, 0x5E, 0x79, 0x71          };
uint8_t  conta;                 // El contador es una variable global

void setup() {
  pinMode(8, INPUT_PULLUP);     // Entrada para el botón
  DDRD = 0xFF;                  // Salida para el display
```

```
  conta = 0;                          // El contador inicia en 0
  PORTD = 0x3F;
}

void loop() {
  if(digitalRead(8) == LOW) {         // Botón presionado
    conta = conta + 1;                // Incrementa el contador
    conta = conta % 16;               // Asegura que está entre 0 y 15
    PORTD = pgm_read_byte(&tabla[conta]);
    delay(250);
  }
}
```

Si el programador de Arduino conoce la funcionalidad de los diferentes Registros I/O, tiene la libertad de emplearlos en donde sea conveniente. En el programa anterior, por ejemplo, se observa que para el manejo del botón es más simple emplear las funciones de Arduino porque se evalúa solo un bit.

Otro aspecto que se observa en el *sketch* anterior es que en Arduino también es posible colocar las constantes en la memoria Flash, el entorno reconoce al atributo `PROGMEM` y a la función de lectura en Flash sin requerir la inclusión explícita de alguna biblioteca.

En la Sección 2.10 se mostró la ubicación del ATMega328P en una tarjeta Arduino UNO, se puede ver que el Puerto D ocupa desde la terminal 0 hasta la 7, ahí se debe conectar al visualizador de 7 segmentos y el botón debe conectarse en la terminal 8, que corresponde con PB0. Las terminales 0 y 1 (PD0 y PD1) tienen la función alterna de un puerto serie (RXD y TXD, ver Figura 2.2). El MCU de Arduino tiene un cargador en su sección de arranque que permite actualizar la sección de aplicación, recibiendo el nuevo código de manera serial (ver la Sección 2.3, Memoria de Programa), así que estas terminales están conectadas al convertidor USB-TTL con el que Arduino se comunica con la PC. Por ello, es conveniente que las terminales 0 y 1 estén desconectadas cuando la tarjeta se programa para evitar cualquier efecto de carga, si los pines son salidas digitales no hay problema cuando se tiene una resistencia de carga relativamente alta.

3.8. Ejercicios

En esta sección se presenta una serie de problemas para practicar con el ATMega328P, algunos fueron planeados para resolverse con ensamblador, en los demás se puede emplear lenguaje C.

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.

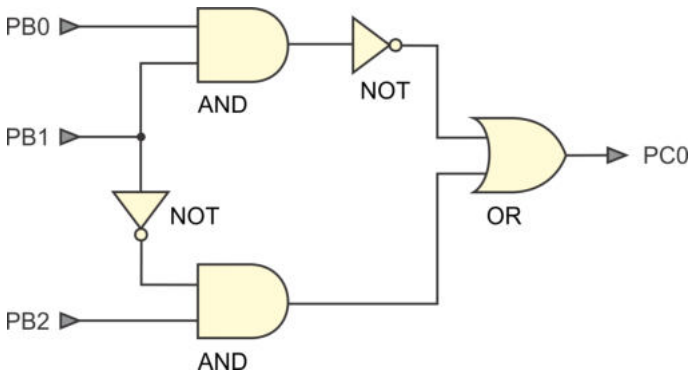


Figura 3.21: Circuito combinacional para el problema 1

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. Modifique el hardware del Ejemplo 3.3 agregando un botón en PB1 y agregue el código necesario al programa para que el contador sea ascendente-descendente.

4. 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 3.29. Para el parpadeo en el color verde considere una frecuencia de 1 Hz.

Tabla 3.29: Secuencia para un sistema con dos semáforos

Rojo1	**Amarillo1**	**verde1**	**Rojo2**	**Amarillo2**	**verde2**	Tiempo(Seg)
1	0	0	0	0	1	10
1	0	0	0	0	parpadeo	5
1	0	0	0	1	0	5
0	0	1	1	0	0	10
0	0	parpadeo	1	0	0	5
0	1	0	1	0	0	5

5. Realice un programa en ensamblador que lea dos números de 4 bits del Puerto B de un ATMega328P (sean A y B, A del nibble alto y B del nibble bajo), para luego realizar el producto de A con B mediante sumas sucesivas, dejando el resultado en el Puerto D.

$$PuertoD = A + A + ... + A \text{ (B veces, pero B puede tener 0)}$$

6. Con un ATMega328P genere una señal PWM en su terminal PB0, la frecuencia de la salida no es importante, pero el ciclo de trabajo debe estar determinado por el Puerto D. Se sugiere manejar un contador de 8 bits (0 a 255) que se incremente en cada iteración y organizar el programa de manera que si el contador es menor o igual a la entrada del Puerto D, la salida PB0 debe estar en alto, en caso contrario, en la salida PB0 se tendrá un nivel bajo.

7. Empleando un ATMega328P y programando en lenguaje ensamblador, implemente un decodificador de 3 a 8 con salidas activas en bajo. Utilice el Puerto C como entrada y al Puerto B como salida.

8. Se tiene una barra con 8 LEDs conectada en el Puerto D de un ATMega328P, realice un programa que inicialmente encienda al LED de PD0, con la ayuda de un botón conectado en PB0 "desplace" el LED encendido de PD0 a PD7 y con otro botón conectado en PB1 el "desplazamiento" será de PD7 a PD0. Los botones se podrán presionar en cualquier orden y llegando a un extremo se mantendrá al LED encendido hasta que se presione el botón contrario al que llevó al LED encendido a ese extremo.

9. Para el hardware de la Figura 3.18 (AMTega328P con un botón y un *display* de 7 segmentos), codifique su nombre o apellido en 7 segmentos (mínimo 8 caracteres) y al encender el circuito muestre el primer carácter, al presionar el botón avance de carácter en carácter hasta llegar al final, alcanzado el final apague todos los segmentos (un espacio) y después vuelva a iniciar.

10. Mediante un ATMega328P y codificando en lenguaje ensamblador, realice un circuito detector de números primos en datos de 4 bits, lea el número en el nibble menos significativo del Puerto B e indique si es primo en PC0.

11. Diseñe una máquina de estados finitos para el encendido automático de una bomba de agua, posteriormente implemente la MEF en un ATMega328P. El sistema debe contar con dos sensores para un tinaco (Sen_Inf y Sen_Sup) y un sensor para una cisterna (Sen_Cist). Los sensores se conectarán en el Puerto B del MCU y proporcionarán un 1 lógico si hay agua y un 0 lógico en caso contrario.

    La bomba se manejará en el bit PC0 e inicialmente estará apagada, se encenderá cuando Sen_Inf no detecte agua pero solo si Sen_Cist indica que hay agua en la cisterna. Una vez que la bomba esté encendida, se apagará cuando el agua alcance al sensor Sen_Sup.

12. Para mover un motor a pasos unipolar se debe generar la secuencia que se muestra en la Tabla 3.30. Realice un programa que genere la secuencia en los bits [PB3:PB0], considere la inclusión de dos interruptores en el Puerto C, uno para seleccionar la dirección (PC0) y otro para introducir una pausa (PC1). Cuando el interruptor de PC0 está cerrado, la secuencia se debe generar en

orden creciente y en caso contrario, el orden de la generación será el inverso. Cuando el botón de pausa se cierre se detendrá la secuencia, la cual continuará desde ese punto al abrirse el botón, según la dirección seleccionada. Tenga en cuenta que los interruptores introducen un 0 cuando están cerrados.

Tabla 3.30: Secuencia para un motor a pasos

PB3	**PB2**	**PB1**	**PB0**	**PC0 = 0**	**PC0 = 1**
0	0	0	1	↓	↑
0	0	1	1	↓	↑
0	0	1	0	↓	↑
0	1	1	0	↓	↑
0	1	0	0	↓	↑
1	1	0	0	↓	↑
1	0	0	0	↓	↑
1	0	0	1	↓	↑

Considere un retardo de 100 ms entre cada cambio en la salida y cuando se alcance el último valor la secuencia se debe iniciar con el primero, según la dirección seleccionada.

Capítulo 4

Interrupciones en los Puertos

Los microcontroladores AVR tienen una gama amplia de recursos internos, en este capítulo se describen dos tipos de interrupciones que se pueden presentar en los puertos: las interrupciones externas y las interrupciones por cambios en las terminales.

Debe recordarse que la configuración y uso de los recursos internos de un MCU AVR se realiza por medio de los Registros I/O correspondientes. Por ello, un sistema que utiliza los recursos internos requiere que se coloquen los valores correctos en sus registros de control para una operación adecuada.

También se debe tomar en cuenta que los recursos generalmente son atendidos por interrupciones, cuando ocurre un evento, el contador del programa (registro PC) toma un valor del vector de interrupciones para dar paso a la ejecución de la rutina que dará servicio a la interrupción (ISR). De manera que, para el manejo adecuado de cada uno de los recursos internos se deben conocer las condiciones para la generación de las interrupciones y los vectores de interrupción involucrados.

4.1. Interrupciones Externas

Las interrupciones externas sirven para detectar un estado lógico o un cambio de estado en terminales predefinidas de un microcontrolador. Son útiles para monitorear interruptores, botones o sensores con salida a relevador, con su uso se evita un sondeo continuo por programación. El ATMega328P tiene dos interrupciones externas, las cuales se describen en la Tabla 4.1 con su nombre y dirección, la dirección se utiliza al programar en ensamblador y el nombre cuando se programa en lenguaje C.

Las interrupciones externas se pueden configurar para detectar un nivel bajo de voltaje o una transición, ya sea por un flanco de subida o de bajada y se pueden generar aun cuando sus respectivas terminales estén configuradas como salidas.

Tabla 4.1: Interrupciones externas

Nombre	Dirección	Ubicación	Descripción
INT0	0x002	PD2	Interrupción externa 0
INT1	0x004	PD3	Interrupción externa 1

Para detectar las transiciones se requiere que esté activa la señal de reloj $clk_{I/O}$, esta señal está destinada a los módulos de los recursos y es anulada en la mayoría de los modos de bajo consumo (Sección 2.9). Por el contrario, la detección de un nivel bajo no requiere de una señal de reloj para producir una interrupción, por lo que esta opción es la más adecuada para despertar al microcontrolador con una interrupción externa, sin importar el modo de reposo.

4.1.1. Configuración

La configuración de la INT0 y la INT1 se realiza en el registro `EICRA` (*External Interrupt Control Register A*, Registro A para el Control de las Interrupciones Externas). El registro `EICRA` es un Registro I/O Extendido, sus bits son:

REG.	7	6	5	4	3	2	1	0	DIR.
EICRA	-	-	-	-	ISC11	ISC10	ISC01	ISC00	(0x69)

- **Bits [3:2] - ISC1[1:0]:** Para configurar el sentido de la INT1 (ISC, *Interrupt Sense Control*). Definen el tipo de evento que va a generar la interrupción externa 1.
- **Bits [1:0] - ISC0[1:0]:** Para configurar el sentido de la INT0. Definen el tipo de evento que va a generar la interrupción externa 0.

En la Tabla 4.2 se muestran las combinaciones de los bits `ISCx[1:0]` (x puede ser 1 o 0), para identificar los eventos que dan lugar a las interrupciones externas.

Tabla 4.2: Configuración de las interrupciones externas (x puede ser 1 o 0)

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

4.1.2. Habilitación y Estado

Cualquier interrupción va a producirse solo 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).

Los habilitadores individuales de la INT0 y la INT1 se encuentran en el registro con la máscara de las interrupciones externas (`EIMSK`, *External Interrupt Mask Register*), correspondiendo con los 2 bits menos significativos:

REG.	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**DIR.**
EIMSK	-	-	-	-	-	-	INT1	INT0	0x1D (0x3D)

- **Bit 1 - INT1:** Habilitador de la interrupción externa 1.
- **Bit 0 - INT0:** Habilitador de la interrupción externa 0.

El estado se refleja en el registro de banderas de las interrupciones externas (`EIFR`, *External Interrupt Flag Register*), el cual incluye una bandera por interrupción y también corresponden con los 2 bits menos significativos:

REG.	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**DIR.**
EIFR	-	-	-	-	-	-	INTF1	INTF0	0x1C (0x3C)

- **Bit 1 - INTF1:** Bandera de la interrupción externa 1.
- **Bit 0 - INTF0:** Bandera de la interrupción externa 0.

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 (Tabla 4.2). La puesta en alto de una de estas banderas es lo que produce la interrupción, dando paso a la ejecución de la ISR correspondiente y con ello la bandera se limpia automáticamente por hardware, aunque también se puede limpiar por software reescribiendo un 1 lógico. En la ISR no es necesario evaluar las banderas porque cada interrupción externa tiene su propio vector.

4.1.3. Ejemplos de Uso

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. En ambos ejemplos no se utilizan diagramas de flujo para describir su comportamiento porque en el programa principal solo se realizan las configuraciones necesarias y las tareas de las ISRs son muy simples.

Ejemplo 4.1 - Conmutación por Interrupción
Realice un programa que conmute la salida menos significativa del Puerto B (PB0) de un ATMega328P cada vez que es presionado un botón conectado en PD2 (INT0), en la Figura 4.1 se muestra el hardware correspondiente.

En el Puerto D se habilita al resistor de *pull-up* para que tenga un 1 lógico mientras el botón no se presione. El otro extremo del botón se conecta a tierra para insertar un 0 lógico al presionarlo, por lo tanto, la interrupción se configura para detectar un flanco de bajada. La conmutación de la salida se realiza en la ISR, por lo que el programa principal queda ocioso.

Figura 4.1: Manejo de un botón por interrupción

En lenguaje C es necesario incluir a la biblioteca `interrupt.h` para el manejo de las interrupciones, la biblioteca contiene a la función `sei()` para poner en alto al habilitador global de interrupciones.

La solución en lenguaje C es:

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

ISR(INT0_vect) {            // ISR de la INT0
    PINB = 0x01;            // Conmuta PB0 sin modificar otros bits
}

int main() {
    DDRD = 0x00;            // Puerto D como entrada
    PORTD = 0xFF;           // Resistores de pull-up en el Puerto D
    DDRB = 0xFF;            // Puerto B como salida
    EICRA = 0x02;           // Configura la INT0 por flanco de bajada
    EIMSK = 0x01;           // Habilita la INT0
    PORTB = 0x00;           // Estado inicial de la salida
    sei();                  // Habilitador global de interrupciones
    while(1) {              // Lazo infinito, permanece ocioso
        asm("NOP");
    }
}
```

La instrucción `NOP` en el lazo infinito solo es para mostrar que el programa principal permanece sin realizar alguna tarea significativa.

Para la solución en ensamblador, puesto que en este ejemplo solo se maneja la interrupción externa 0, el código para atenderla se puede ubicar directamente en la dirección 0x002 y realizar un salto al programa principal en la dirección 0x000, este salto se ejecutará en cada reinicio del MCU. El programa con la solución es:

```
        .include <m328pdef.inc>  ; Biblioteca con definiciones

        .ORG    0x000            ; Vector de reset
         JMP    Inicio
```

```
        .ORG    0x002           ; Interrupción externa 0
         SBI    PINB, 0         ; Conmuta PB0 sin modificar otros bits
         RETI
Inicio:                         ; Inicializaciones
         LDI    R16, 0x00
         OUT    DDRD, R16       ; Puerto D como entrada
         LDI    R16, 0xFF
         OUT    PORTD, R16      ; Resistores de pull-up en el Puerto D
         OUT    DDRB, R16       ; Puerto B como salida
         LDI    R16, 0x02       ; INT0 por flanco de bajada
         STS    EICRA, R16
         LDI    R16, 0x01       ; Habilita la INT0
         OUT    EIMSK, R16
         CLR    R16             ; Estado inicial de la salida
         OUT    PORTB, R16
         SEI                    ; Habilitador global de interrupciones

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

La inclusión de la biblioteca `m328pdef.inc` no se requiere cuando se utiliza el entorno de Microchip Studio, se ha dejado en este programa y posteriores, para que exista compatibilidad con Atmel Studio, entorno de desarrollo previo a la adquisición de la empresa Atmel por la empresa Microchip.

La directiva `ORG` se utilizó para dar claridad al programa, no es necesaria porque por omisión un programa inicia en la dirección 0x000 y la instrucción `JMP` utiliza dos palabras, de manera que la siguiente instrucción se va a ubicar en la dirección 0x002. En esta versión también se observa que el trabajo del programa principal prácticamente es nulo, la interrupción externa es la encargada de monitorear al botón y en su ISR se realiza la tarea deseada.

Otro aspecto a señalar es que la escritura del registro `EIMSK` se hace con la instrucción `OUT` a diferencia de la escritura del registro `EICRA` que utiliza la instrucción `STS`, esto es así porque `EIMSK` es un Registro I/O y `EICRA` es un Registro I/O Extendido, y por eso requiere de una instrucción de almacenamiento directo en SRAM.

En el siguiente ejemplo se utilizan las dos interrupciones externas y se muestra cómo una variable puede ser modificada por dos ISRs diferentes.

Ejemplo 4.2 - Contador de Eventos
Realice un contador de eventos ascendente/descendente de 0 a 9 con salida en un visualizador de 7 segmentos. Utilice las terminales INT0 e INT1 para el ingreso de los eventos y al Puerto B para generar la salida. La INT0 será para los incrementos y cuando el contador alcance el valor de 9, con otro incremento debe pasar a 0. La INT1 será para los decrementos y cuando el contador alcance el valor de 0, otro decremento lo debe llevar a 9. En la Figura 4.2 se muestra el hardware requerido.

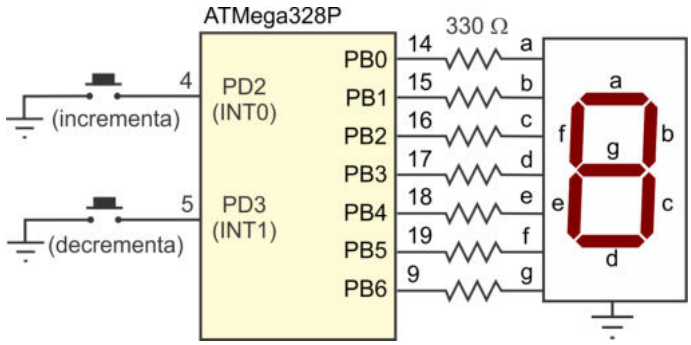


Figura 4.2: Contador con interrupciones

Dado que se van a manejar dos botones, las dos interrupciones externas se configuran por flanco de bajada. En las ISRs se modifica al contador y se actualiza la salida, dejando ocioso al programa principal.

En el programa en C se utiliza una variable global para el manejo del contador, las ISRs evalúan y modifican esta variable, el código con la solución en lenguaje C es:

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

const uint8_t tabla[] PROGMEM = { 0x3F, 0x06, 0x5B, 0x4F, 0x66,
                                  0x6D, 0x7D, 0x07, 0x7F, 0x67  };
uint8_t Conta = 0;

ISR(INT0_vect) {                                  // ISR de la INT0
   Conta = (Conta < 9)? Conta + 1: 0;             // Incrementa al contador
   PORTB = pgm_read_byte(&tabla[Conta]);          // Actualiza la salida
}

ISR(INT1_vect) {                                  // ISR de la INT1
   Conta = (Conta > 0)? Conta - 1: 9;             // Decrementa al contador
   PORTB = pgm_read_byte(&tabla[Conta]);          // Actualiza la salida
}

int main() {
   DDRD = 0x00;              // Puerto D como entrada
   PORTD = 0xFF;             // Resistores de pull-up en el Puerto D
   DDRB = 0xFF;              // Puerto B como salida
   EICRA = 0x0A;             // INT0 e INT1 por flanco de bajada
   EIMSK = 0x03;             // Habilita la INT0 e INT1
   PORTB = 0x3F;             // Estado inicial de la salida
   sei();                    // Habilitador global de interrupciones

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

Para la solución en ensamblador se destina al registro `R20` para el contador y la tabla con las constantes se ubica después de las rutinas. El comportamiento es el mismo que en alto nivel, en las ISRs se debe evaluar al contador para determinar su nuevo valor y se llama a una rutina para actualizar la salida. El código con la solución en ensamblador es:

```
        .include <m328pdef.inc>
        .def    Conta = R20

        JMP     inicio                  ; Vector de reset
        JMP     ISR_Ext0                ; Interrupción externa 0
        JMP     ISR_Ext1                ; Interrupción externa 1
inicio:
        CLR     R16
        OUT     DDRD, R16               ; Entrada para los botones
        SER     R17
        OUT     PORTD, R17              ; Pull-up para las entradas
        OUT     DDRB, R17               ; Salida para el display

        LDI     R16, 0x0A               ; Configura INT0/INT1
        STS     EICRA, R16              ; por flanco de bajada
        LDI     R16, 0x03               ; Habilita a INT0 e INT1
        OUT     EIMSK, R16
        CLR     Conta                   ; Estado inicial del contador
        LDI     R16, 0x3F               ; y de la salida
        OUT     PORTB, R16
        SEI                             ; Habilitador global
Loop:   NOP                             ; Lazo infinito,
        RJMP    Loop                    ; permanece ocioso

ISR_Ext0:                               ; Rutina de atención a la INT0
        INC     Conta
        CPI     Conta, 10               ; Si el contador llega a 10
        BRNE    s_01
        CLR     Conta                   ; se inicia con 0
s_01:   RCALL   lee_tabla               ; Obtiene el código 7 segmentos
        RETI

ISR_Ext1:                               ; Rutina de atención a la INT1
        CPI     Conta, 0
        BREQ    s_11                    ; Salta si es cero
        DEC     Conta                   ; si no es 0 decrementa
        RJMP    s_12
s_11:   LDI     Conta, 9
s_12:   RCALL   lee_tabla               ; Obtiene el código 7 segmentos
        RETI

lee_tabla:
        LDI     R30, LOW(tabla << 1)    ; Z apunta a la tabla
        LDI     R31, HIGH(tabla << 1)
        ADD     R30, Conta              ; Suma al contador
```

```
        BRCC   s_21                    ; Brinca si no hay acarreo
        INC    R31                     ; Incrementa debido al acarreo
s_21:   LPM    R17, Z                  ; Obtiene el dato de la tabla
        OUT    PORTB, R17              ; lo manda al Puerto D
        RET

tabla: .db 0x3F, 0x06, 0x5B, 0x4F, 0x66, 0x6D, 0x7D, 0x07, 0x7F, 0x67
```

Eliminación de Rebotes

Para que una interrupción se genere deben ocurrir 3 situaciones: el habilitador global debe estar en alto (bit `I` de `SREG`), las interrupciones deben estar habilitadas (en el registro `EIMSK`) y debe ocurrir el evento configurado en el registro `EICRA`. Las dos primeras son acciones que se realizan por software y la tercera cuando se presione un botón, esta acción puede ocurrir en cualquier momento y queda reflejada en el registro `EIFR`.

El registro `EIFR` generalmente no se evalúa por software porque la bandera que generó la interrupción se limpia automáticamente cuando se da paso a la ejecución de la ISR. Sin embargo, es frecuente que al emplear botones se produzca un rebote una vez transferido el control a la ISR y la bandera se ponga en alto nuevamente, lo que provocará otra ejecución de la ISR. Para evitar esta situación es conveniente esperar un pequeño intervalo de tiempo, después evaluar la bandera y si está en alto, limpiarla para evitar una ejecución inesperada de la ISR, la bandera se limpia por software al reescribirle un 1 lógico. Con estas consideraciones, el código para la interrupción externa 1 del ejemplo 4.2 queda de la siguiente manera:

```
ISR(INT1_vect) {                          // ISR de la INT0

   Conta = (Conta > 0)? Conta - 1: 9;     // Decrementa al contador
   PORTB = pgm_read_byte(&tabla[Conta]); // Actualiza la salida

   _delay_ms(100);                 // Para que el botón se estabilice
   if(EIFR & 1 << INTF1)           // Si la bandera está en alto
      EIFR |= 1 << INTF1;          // se limpia con un 1 lógico
}
```

El inconveniente con el código anterior es que el MCU no puede hacer otras tareas durante el retardo y esto afecta el desempeño de algunos sistemas, por lo que otra alternativa para la eliminación de rebotes se basa en la inserción de un capacitor en paralelo con el botón (de la terminal de entrada a tierra). El capacitor se descarga cuando el botón es presionado y absorbe los rebotes mientras se carga nuevamente, la carga se realiza por medio de la resistencia de *pull-up*. Un capacitor cerámico de 100 nF trabaja adecuadamente con este propósito.

Tabla 4.3: Interrupciones por cambios en las terminales

Nombre	Dirección	Descripción
PCINT0	0x006	Interrupción por cambios en el Puerto B
PCINT1	0x008	Interrupción por cambios en el Puerto C
PCINT2	0x00A	Interrupción por cambios en el Puerto D

4.2. Cambios en las Terminales

Las interrupciones externas son muy útiles porque pueden detectar el flanco deseado o un nivel bajo de voltaje, sin embargo, el ATMega328P tiene la desventaja de contar únicamente con dos eventos de ese tipo. Para aminorar esa desventaja el MCU también incluye las interrupciones por cambios en las terminales (*Pin Change Interrupt*), se trata de 3 eventos que pueden ser generados por las terminales de los puertos, en la Tabla 4.3 se describen estas interrupciones. Todas las terminales del mismo puerto van a producir la misma interrupción, de manera que por software se debe evaluar el estado de las terminales de interés para identificar cual cambió, conocer su nuevo valor y realizar las acciones requeridas por el sistema.

Una característica de estas interrupciones es que son asíncronas, esto significa que no requieren de una señal de reloj activa para generarse por lo que se pueden emplear para liberar al MCU de cualquier modo de bajo consumo.

En la Figura 4.3 se observa que cada terminal tiene un identificador debido a las interrupciones por cambios en las terminales y la forma en que se agrupan en tres vectores de interrupción. En los registros `PCMSK0`, `PCMSK1` y `PCMSK2` se deben establecer las máscaras que determinan las terminales que serán monitoreadas, sus bits corresponden con los identificadores de la misma figura, estos son:

REG.	7	6	5	. . .	1	0
PCMSK0	PCINT7	PCINT6	PCINT5	. . .	PCINT1	PCINT0
PCMSK1	-	PCINT14	PCINT13	. . .	PCINT9	PCINT8
PCMSK2	PCINT23	PCINT22	PCINT21	. . .	PCINT17	PCINT16

Los registros `PCMSK0`, `PCMSK1` y `PCMSK2` son Registros I/O Extendidos ubicados en las direcciones 0x6B, 0x6C y 0x6D, respectivamente. Con un 1 en la posición deseada el hardware va a monitorear la ocurrencia de cambios en las terminales correspondientes, por ejemplo, si un sistema contiene 4 botones conectados en las terminales PB3, PB2, PB1 y PB0, así como un sensor de presencia en PD3, el programa debe incluir las siguientes asignaciones:

```
PCMSK0 = 0x0F;    // Para detectar cambios en los botones
PCMSK2 = 0x08;    // Para monitorear el sensor de presencia
```

Puesto que los cambios en las terminales de PB3 a PB0 van a provocar la misma interrupción (PCINT0), por software se debe evaluar el estado de las 4 terminales

para determinar las acciones a realizar, debe tomarse en cuenta que los cambios se producen cuando un botón se presiona o se libera. En el caso del sensor de presencia ubicado en PD3 es un solo evento el que produce la interrupción (PCINT2), aún con ello, se debe evaluar el estado de la terminal para saber si el cambio se produjo porque alguien se acercó o se alejó.

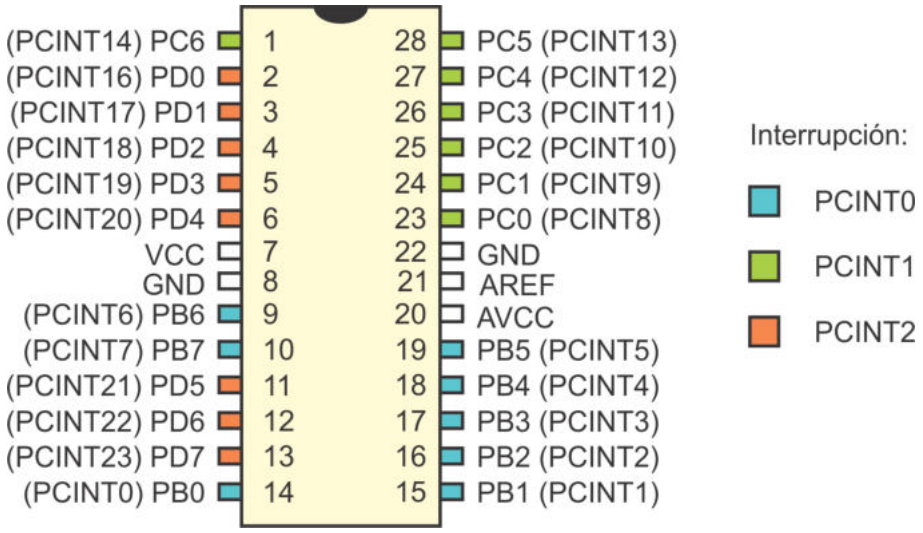


Figura 4.3: Identificadores para las interrupciones por cambios en las terminales

4.2.1. Habilitación y Estado

Los habilitadores de las interrupciones por cambios en las terminales se encuentran en el Registro I/O Extendido denominado `PCICR` (*Pin Change Interrupt Control Register*), cuyos bits son:

REG.	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**DIR.**
PCICR	-	-	-	-	-	PCIE2	PCIE1	PCIE0	(0x68)

- **Bit 2 - PCIE2:** Habilita las interrupciones por cambios en el Puerto D.
- **Bit 1 - PCIE1:** Habilita las interrupciones por cambios en el Puerto C.
- **Bit 0 - PCIE0:** Habilita las interrupciones por cambios en el Puerto B.

Para que las interrupciones se generen también se requiere que el habilitador global esté en alto (bit `I` de `SREG`).

El estado de las interrupciones se refleja en su registro de banderas denominado `PCIF` (*Pin Change Interrupt Flag Register*). Es un Registro I/O que incluye una bandera por cada interrupción, estas corresponden con los 3 bits menos significativos:

REG.	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**DIR.**
PCIFR	-	-	-	-	-	PCIF2	PCIF1	PCIF0	0x1B (0x3B)

- **Bit 2 - PCIF2:** Se pone en alto cuando ocurre un cambio en las terminales enmascaradas del Puerto D.
- **Bit 1 - PCIF1:** Se pone en alto cuando ocurre un cambio en las terminales enmascaradas del Puerto C.
- **Bit 0 - PCIF0:** Se pone en alto cuando ocurre un cambio en las terminales enmascaradas del Puerto B.

Las banderas se ponen en alto si el habilitador global e individual están habilitados y ocurre un cambio en las terminales incluidas en las máscaras (registros `PCMSK0`, `PCMSK1` y `PCMSK2`). La puesta en alto de una de estas banderas es lo que produce la interrupción, dando paso a la ejecución de la ISR correspondiente y con ello la bandera se limpia automáticamente por hardware, aunque también se puede limpiar por software reescribiendo un 1 lógico, en la ISR se debe evaluar el estado de las terminales involucradas para identificar cuál de ellas cambió y realizar las acciones correspondientes. Las banderas se pueden limpiar por software para eliminar rebotes, en caso de conectar botones.

4.2.2. Ejemplos de Uso

En esta sección se presentan dos ejemplos de uso de las interrupciones por cambios en las terminales, solo se muestran los programas en lenguaje C porque el comportamiento del microcontrolador queda determinado por la configuración de los Registros I/O y no por el lenguaje empleado en la codificación, como se pudo ver en los ejemplos con interrupciones externas.

Ejemplo 4.3 - Revisión de Botones
En la figura 4.4 se muestra un ATMega328P al que se le ha conectado una barra de 8 LEDs en el Puerto D y 2 botones en el Puerto C (botón 1 en PC0 y botón 2 en PC1). Realice un programa por medio del cual debe iniciar encendido el LED conectado en PD0 y con el botón 1 el LED encendido debe desplazarse en la barra de PD0 a PD7, mientras que con el botón 2 el desplazamiento debe ser de PD7 a PD0. Al llegar a un extremo la salida permanecerá sin cambios hasta que se presione el botón contrario al que llevó al LED encendido a ese extremo.

En el programa principal se hará la configuración de las entradas para los botones, las salidas para los LEDs y las interrupciones por cambios en las terminales, habilitando a la PCINT1 para evaluar al Puerto C. Después de ello, el programa principal permanecerá ocioso porque el estado de los botones se revisará en la ISR correspondiente. Los dos botones dan lugar a la misma interrupción, se produce un evento cuando un botón se presiona y otro cuando se libera, por lo tanto, en la ISR se debe evaluar el estado de las terminales para identificar al evento. Debido a los resistores de *pull-up*, se tendrá un nivel lógio bajo en PC0 o PC1 cuando su respectivo botón es presionado.

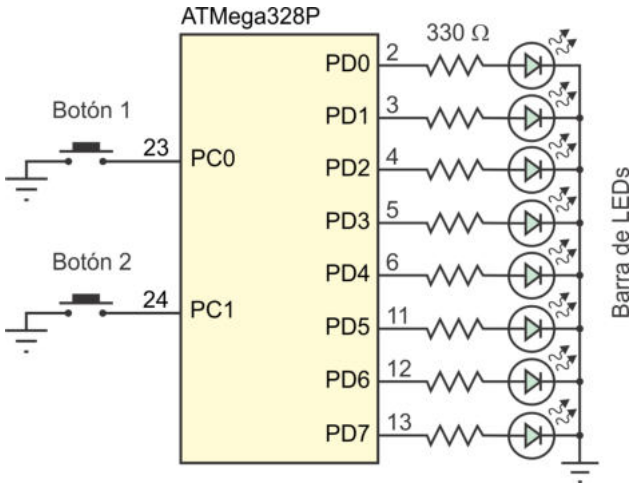


Figura 4.4: Revisión de botones con interrupciones por cambios en las terminales

La solución se basa en un contador ascendente-descendente de 0 a 7, los eventos en los botones modifican su valor y la salida se ajusta con una tabla de valores constantes situada en memoria Flash, el programa con la solución del problema es:

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

const uint8_t LED[] PROGMEM = { 0x01, 0x02, 0x04, 0x08,
                                0x10, 0x20, 0x40, 0x80  };
uint8_t sal = 0;

ISR(PCINT1_vect) {                              // Cambios en el Puerto C

   if(!(PINC & 0x01)) {                         // Botón 1 presionado
      if(sal < 7) {
         sal++;                                 // Incrementa al contador
         PORTD = pgm_read_byte(&LED[sal]);      // Actualiza la salida
      }
   }
   else if(!(PINC & 0x02)) {                    // Botón 2 presionado
      if(sal > 0) {
         sal--;                                 // Decrementa al contador
         PORTD = pgm_read_byte(&LED[sal]);      // Actualiza la salida
       }
    }
}

int main() {

   DDRC = 0x00;          // Entrada para los botones
   PORTC = 0x03;         // Pull-up
   DDRD = 0xFF;          // Salida para la barra de LEDs
```

```
    PCMSK1 = 0x03;          // Máscara para los dos botones
    PCICR = 0x02;           // Habilita la interrupción

    PORTD = 0x01;           // Estado inicial de la salida
    sei();                  // Habilitador global de interrupciones
    while(1)
        asm("NOP");         // Lazo infinito, permanece ocioso
}
```

En el siguiente ejemplo se evalúa el estado de 6 botones, incluyendo la revisión de un grupo de 4 botones que dan lugar a la misma salida, sin importar cual fue el botón que produjo el evento.

Ejemplo 4.4 - Luces de un Automóvil
En las 4 puertas, cajuela y cofre de un automóvil se han puesto botones para detectar si están abiertas o cerradas, en la Figura 4.5 se muestra la conexión con el microcontrolador, los botones están presionados con las puertas cerradas y la indicación de una puerta abierta se realiza mediante una luz ubicada en el área que corresponde, es decir, si alguna de las 4 puertas está abierta debe encenderse la luz interior; si la cajuela está abierta, además de la luz interior debe encenderse la luz de la cajuela y si el cofre está abierto, también debe encenderse la luz interior así como la luz del motor. Desarrolle el programa en lenguaje C para el microcontrolador.

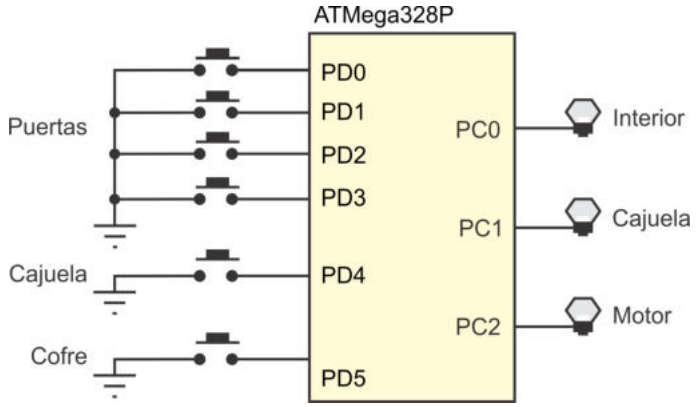


Figura 4.5: Puertas de un automóvil

Para la solución se asume que todas las puertas están cerradas cuando el sistema se energiza y, por lo tanto, las 3 salidas inician apagadas. En el programa principal se debe configurar la interrupción por cambios en el Puerto D, para que en la ISR se identifique el estado de las puertas y se genere la salida que corresponda. Considerando que se puede producir un evento cuando una puerta se abre o cierra, en la ISR se deben revisar todas las combinaciones posibles en las entradas para determinar el estado de las salidas. En la Tabla 4.4 se resumen las diferentes situaciones y durante la codificación se inicia con la combinación que genera más salidas en alto.

Tabla 4.4: Consideración para las puertas de un automóvil

PD5	PD4	PD3 - PD0	PC2	PC1	PC0
1	1	X	1	1	1
0	1	X	0	1	1
1	0	X	1	0	1
0	0	!= 0	0	0	1
0	0	0	0	0	0

El programa con la solución es:

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

ISR(PCINT2_vect) {

  if( (PIND & 0x10) && (PIND & 0x20) )  // Cajuela y cofre abiertos
     PORTC = 0x07;
  else if(PIND & 0x10)                  // Sólo cajuela abierta
     PORTC = 0x03;
  else if(PIND & 0x20)                  // Sólo motor
     PORTC = 0x05;
  else if( PIND & 0x0F )                // Alguna puerta abierta
     PORTC = 0x01;
  else
     PORTC = 0x00;                      // Todo cerrado
}

int main() {

  DDRD = 0x00;          // Entradas de puertas
  PORTD = 0xFF;         // Pull-up
  DDRC = 0xFF;          // Salida para las lámparas

  PCMSK2 = 0x3F;        // Máscara de interrupciones
  PCICR = 0x04;         // Habilita PCINT2 (Puerto D)

  PORTC = 0x00;         // Todo cerrado (condición inicial)

  sei();                // Habilitador global

  while(1)
     asm("NOP");
}
```

4.3. El atributo *volatile*

Con el manejo de interrupciones es común el uso de variables globales que serán modificadas por diferentes ISRs, esta es la forma en que las ISRs se pueden comunicar y compartir información. Las variables también se pueden modificar en el programa principal aunque es más frecuente que solo sean evaluadas en el lazo infinito, para determinar las acciones a realizar como una respuesta a los eventos del sistema que son atendidos por interrupción, en esos casos, si solo se analiza el ciclo infinito aparentará que las variables se mantienen sin cambios.

Los compiladores suelen emplear una estrategia de optimización que consiste en sacar de las estructuras repetitivas el código que no cambia dentro del ciclo, para evitar la ejecución de código que en cada iteración tendrá el mismo efecto. Sin embargo, en muchos casos el compilador no puede determinar si una variable será modificada en una ISR y, por lo tanto, no debería ser considerada en esta optimización. Para estas situaciones se puede emplear al atributo `volatile` en la declaración de variables, esto le indica al compilador que la variable podría ser modificada en una ISR y no se debe considerar en la optimización. El atributo se antepone a la declaración de la siguiente manera: `volatile uint8_t Contador;`.

Para los microcontroladores AVR y específicamente con las pruebas realizadas en un ATMega328P, se observó que el uso del atributo `volatile` depende de la herramienta empleada y del recurso utilizado, con AVR Studio es necesario emplear el atributo para todas aquellas variables que son modificadas en alguna ISR y su valor es consultado en el lazo infinito del programa principal, sin embargo, con Microchip Studio no se requiere el uso del atributo para las interrupciones externas, pero si para otros recursos como el convertidor analógico a digital, puerto serie o interfaz de dos hilos.

4.4. Manejo de Interrupciones con Arduino

La función `attachInterrupt(interrupt, ISR, mode)` está incluida en el IDE de Arduino para atender las interrupciones externas. Donde: `interrupt` es el número de la interrupción (0 o 1), en `ISR` se indica el nombre de la función que le dará servicio y `mode` debe corresponder con una las constantes disponibles (FALLING, RISING, CHANGE y LOW). En el Ejemplo 4.2 se describió un contador de eventos atendidos por interrupciones, si se realiza una versión para Arduino en la sección de `setup` se deben incluir las declaraciones:

```
attachInterrupt(0, incrementa, FALLING);
attachInterrupt(1, decrementa, FALLING);
```

Posterior a la sección de `loop` debe estar el cuerpo de las funciones `incrementa()` y `decrementa()`.

En la Sección 3.7 se mostró que es posible modificar a los Registros I/O directamente en un *skecth* de Arduino, esto incluye a los registros de configuración y habilitación de las interrupciones en los puertos, tanto externas como por cambios en las terminales. En Arduino se reconoce a la función ISR con los diferentes argumentos, para poder atender a cualquier interrupción, esto le da la flexibilidad de poder monitorear eventos en todas las terminales.

En el siguiente *sketch* se muestra un contador ascendente/descendente con salida en binario. Se utilizan las terminales 8 y 9 de la tarjeta Arduino para los botones (corresponden con PB0 y PB1) y al Puerto D como salida. Los eventos se detectan en la PCINT0, se revisa el estado de los botones para asegurarse que han sido presionados, el código es:

```
uint8_t conta = 0;

void setup() {
  pinMode(8, INPUT_PULLUP);       // Botón UP en PB0
  pinMode(9, INPUT_PULLUP);       // Botón DOWN en PB1
  PCMSK0 = 0x03;                  // Interrupciones por cambios
  PCICR = 0x01;                   // en PB0 y PB1
  DDRD = 0xFF;                    // Puerto D como salida
}

void loop() {                     // Permanece ocioso en
  asm("NOP");                     // el lazo infinito
}

ISR(PCINT0_vect) {
   if(digitalRead(8) == false) {         // PB0 presionado
     if(conta < 255) {
        conta++;
        PORTD = conta;
     }
   } else if(digitalRead(9) == false) { // PB1 presionado
      if(conta > 0) {
         conta--;
         PORTD = conta;
      }
   }
}
```

En el *sketch* anterior se observa que el habilitador global de interrupciones está en alto por defecto, no es necesario incluir la llamada a la función `sei()` para activarlo.

4.5. Ejercicios

Los ejercicios de este capítulo son para resolverse en lenguaje C, aunque también pueden ser resueltos en ensamblador, lo más importante en la solución es el uso de

las interrupciones en los puertos de un ATMega328P.

1. En la Figura 4.6 se muestra la conexión de un horno eléctrico a un ATMega328P y el comportamiento de dos señales `hot` y `cold` que el horno genera en función de la temperatura. Realice el programa para que el MCU encienda o apague al horno con la finalidad de mantener su temperatura alrededor de una referencia. Observe el flanco de activación para las interrupciones externas y antes de iniciar el lazo infinito, revise el estado de las entradas para determinar si el horno va a iniciar encendido o apagado.

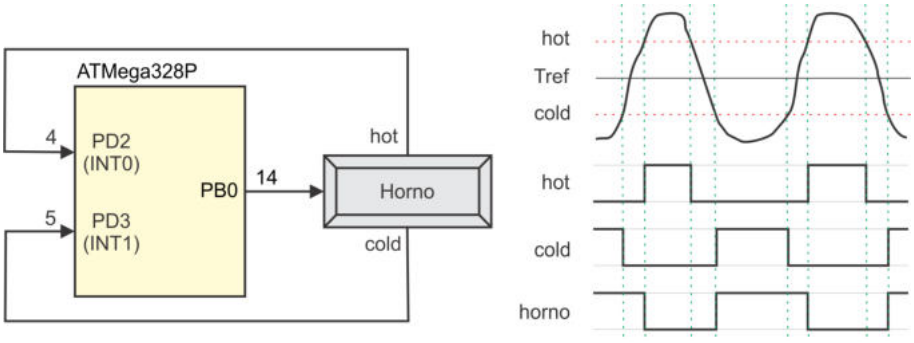


Figura 4.6: Control simple de temperatura

2. Realice el programa para el contador de eventos mostrado en la Figura 4.7, el circuito incluye dos visualizadores de 7 segmentos, por lo que será de 0 a 99. En una variable global se llevará el valor del contador y el código del lazo infinito se enfocará en mostrarlo en los visualizadores. Dos botones serán atendidos por interrupciones externas y dos por cambios en las terminales. El botón *reset* pondrá 0 en la variable de la cuenta y el botón *set* pondrá 99.

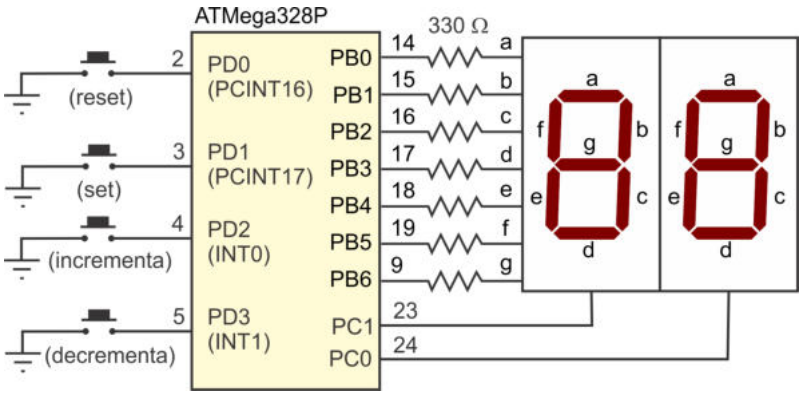


Figura 4.7: Contador atendido por interrupciones

3. Resuelva el problema anterior empleando solo interrupciones por cambios en las terminales.

4. Realice el programa para que el circuito de la Figura 4.8 detecte la secuencia en código Gray mostrada en la misma figura. El LED conectado en PB0 debe encenderse cuando la secuencia ha sido introducida en orden correcto. Al emplear las interrupciones por cambios en las terminales se producirá un evento cuando uno de los interruptores se abra o cierre, por lo que en la ISR se debe revisar si la entrada actual es parte de la secuencia y si está en el orden esperado.

Secuencia	Código
0	0000
1	0001
2	0011
3	0111
4	0110
5	1110
6	1111

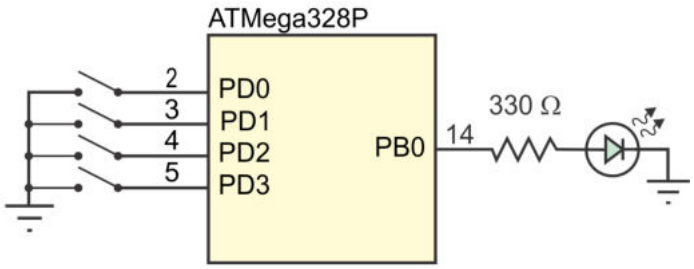


Figura 4.8: Detector de secuencia en código Gray

5. Diseñe el programa para un contador ascendente con múltiples fuentes, el circuito se muestra en la Figura 4.9. Las salidas en los Puertos B y D se mostrarán en binario, en el Puerto B se indicará el botón presionado y en el Puerto D el valor del contador.

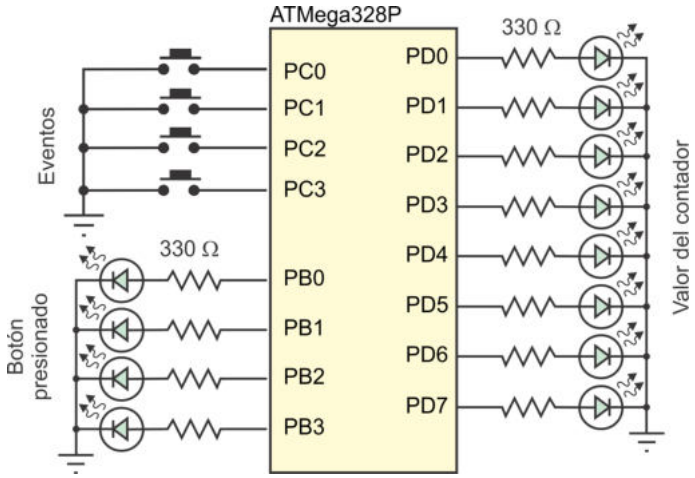


Figura 4.9: Contador con múltiples fuentes

6. Un enconder rotativo genera dos señales cuadradas denominadas A y B, las cuales se encuentran desfasadas por 90°, como se muestra en la Figura 4.10, este comportamiento permite detectar la dirección de giro del encoder. Realice

un programa para el ATMega328P de la Figura 4.10 que encienda un LED de acuerdo con la dirección de giro detectada con ayuda de las interrupciones externas.

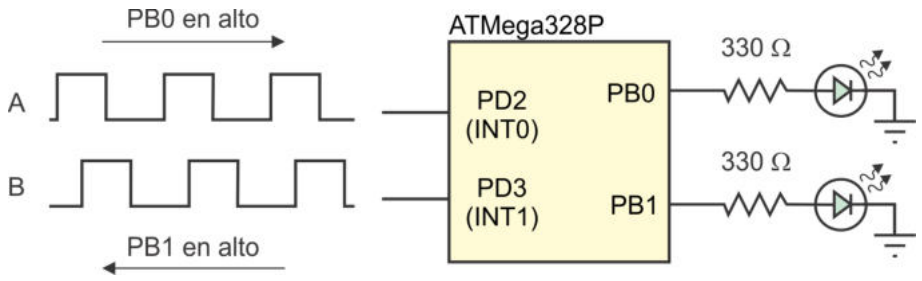


Figura 4.10: Detección de la dirección de giro de un encoder rotativo

Capítulo 5

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 solo es tratado como temporizador, independientemente de que sea manejado por una señal de reloj interna o por eventos externos.

El ATMega328P tiene 3 temporizadores, los temporizadores 0 y 2 son de 8 bits, y el temporizador 1 es de 16 bits. En la Tabla 5.1 se listan los Registros I/O de cada temporizador, el temporizador 1 utiliza 2 de ellos porque son registros de 8 bits. Por la dirección, se observa que el temporizador 0 utiliza un Registro I/O mientras que los temporizadores 1 y 2 utilizan Registros I/O Extendidos.

5.1. Eventos de los Temporizadores

Los eventos que se pueden generar en un ATMega328P con los temporizadores son: desbordamiento, coincidencia por comparación y captura de entrada.

Tabla 5.1: Registros de los temporizadores en el ATMega328P

Temporizador	**Tamaño**	**Registros**	**Dirección**
Timer 0	8 bits	TCNT0	0x26 (0x46)
Timer 1	16 bits	TCNT1H:TCNT1L	(0x85), (0x84)
Timer 2	8 bits	TCNT2	(0xB2)

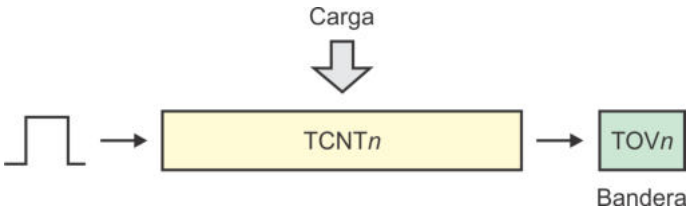


Figura 5.1: Desbordamientos en los temporizadores (*n = 0, 1 o 2*)

Cada temporizador tiene su registro de banderas (`TIFR`, *Timer Interrupt Flag Register*), en donde se señaliza la ocurrencia de los eventos, la distribución de eventos por temporizador es la siguiente:

- **Temporizador 0**: 1 evento de desbordamiento y 2 de coincidencia por comparación. Indicados en el registro `TIFR0`.
- **Temporizador 1**: 1 evento de desbordamiento, 2 de coincidencia por comparación y 1 de captura de entrada. Indicados en el registro `TIFR1`.
- **Temporizador 2**: 1 evento de desbordamiento y 2 de coincidencia por comparación. Indicados en el registro `TIFR2`.

5.1.1. Desbordamiento

Este evento ocurre cuando un temporizador (`TCNTn`) alcanza su valor máximo (MAXVAL) y se reinicia con 0, es decir, hay una transición de 1's a 0's, esto provoca que una bandera (`TOVn`, *Timer/Counter Overflow Flag*) 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.

La 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, esta debe repetirse cada que el evento es producido, para generar intervalos regulares. En la Figura 5.1 se esquematiza al temporizador generando eventos por desbordamiento, se muestra la posibilidad de una carga paralela y la generación de la bandera.

Los desbordamientos pueden ser manejados por los 3 temporizadores (en la Figura 5.1, n puede ser 0, 1 o 2). Desde los primeros microcontroladores, los eventos por desbordamientos en los temporizadores han sido un *de facto* para los diferentes fabricantes.

El valor máximo (MAXVAL) depende del tamaño o número de bits del temporizador y queda determinado con la expresión:

$$MAXVAL = 2^{Tamaño(TCNTn)} - 1$$

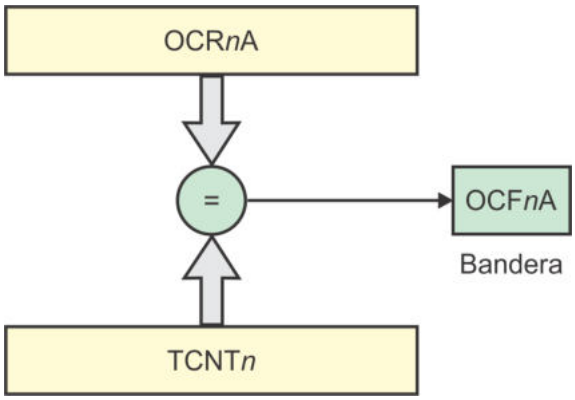


Figura 5.2: Coincidencias por comparación en los temporizadores

5.1.2. Coincidencia por Comparación

El MCU incluye dos registros para comparaciones continuas por cada temporizador (`OCRnA` y `OCRnB`, *Output Compare Register*), en estos registros se pueden cargar valores entre 0 y MAXVAL. En cada ciclo de reloj se compara al temporizador con los registros de comparación, una coincidencia es un evento que se indica con la puesta en alto de la bandera `OCFnA` u `OCFnB` (*Output Compare Flag*). Las banderas se pueden sondear por software o configurar al recurso para que genere una interrupción ante una coincidencia. En la Figura 5.2 se esquematiza la generación de estos eventos considerando solo al registro A para la comparación, se puede referir a cualquiera de los 3 temporizadores.

Es posible configurar a los temporizadores para que se reinicien después de una coincidencia por comparación y algunas terminales relacionadas con el recurso pueden ajustarse, limpiarse o conmutarse automáticamente. Estas características de los temporizadores son la base para la generación de tonos o señales PWM.

5.1.3. Captura de Entrada

Este tipo de eventos solo es manejado por el temporizador 1, el MCU destina una terminal para capturar eventos externos (ICP1, *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 ICP1 para generar las capturas es configurable, puede ser un flanco de subida o uno de bajada. En la Figura 5.3 se muestra como se producen estos eventos.

La bandera de captura de entrada (`ICF1`, *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.

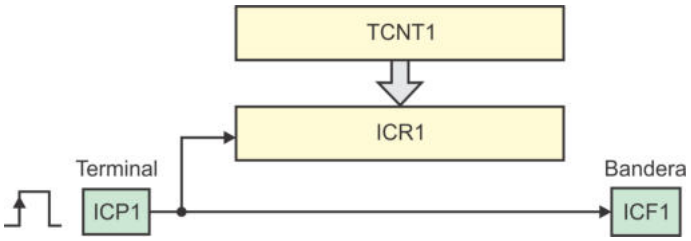


Figura 5.3: Captura de entrada

5.2. Respuesta a los Eventos

Existen 3 formas para detectar los eventos producidos por los temporizadores y actuar ante ellos: sondeo, interrupciones y respuesta automática.

5.2.1. Sondeo (*Polling*)

En este método se destina un conjunto de instrucciones en el lazo infinito para evaluar el estado de las banderas, es el menos eficiente porque el sondeo requiere tiempo de procesamiento de la CPU además de que las banderas deben ser limpiadas por software, reescribiéndoles un 1 lógico. Como un ejemplo, en la Tabla 5.2 se muestra una secuencia de instrucciones que espera un evento por desbordamiento del temporizador 0, la bandera se limpia después de que el evento es atendido.

Tabla 5.2: Sondeo de un desbordamiento del temporizador 0

Versión en ensamblador

```
Lazo:   SBIC   TIFR0, TOV0     ; Salta si la bandera está en bajo
        RJMP   Tiempo
        . . .                  ; Código no temporizado
        RJMP   Lazo
Tiempo:
        . . .                  ; Código temporizado
        SBI    TIFR0, TOV0     ; Limpia la bandera
        RJMP   Lazo
```

Versión en lenguaje C

```
while(1) {                          // Lazo infinito
    if(TIFR0 & 1 << TOV0) {
        . . .                       // Código temporizado
        TIFR0 |= 1 << TOV0;         // Limpia la bandera
    }
    . . .                           // Código no temporizado
}
```

En la Tabla 5.2 se observa que un programa no queda dedicado a esperar y atender un evento, se pueden ejecutar otras instrucciones mientras no ocurra.

5.2.2. Uso de Interrupciones

Todos los eventos de los temporizadores pueden generar interrupciones, para ello, estas se deben activar en su respectivo registro de enmascaramiento de interrupciones (`TIMSK`, *Timer Interrupt Mask Register*), son tres los registros involucrados: `TIMSK0`, `TIMSK1` y `TIMSK2`, uno por temporizador. Además, se debe activar al habilitador global de interrupciones (bit `I` en `SREG`).

Al utilizar interrupciones, el lazo infinito de un programa se ejecuta de manera continua, cuando ocurre un evento, la CPU concluye con la instrucción en proceso para dar paso a la ISR correspondiente y al terminar su ejecución, la CPU procede con la instrucción siguiente a la que se estaba ejecutando cuando ocurrió el evento.

Esta forma de dar atención a los eventos es la más usada, es eficiente porque en el programa principal no se invierte tiempo de procesamiento para esperar la ocurrencia de eventos. Además, cuando se emplean interrupciones, las banderas son limpiadas automáticamente por hardware.

5.2.3. Respuesta Automática

Los temporizadores 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 solo se incluye la configuración de los recursos y la respuesta a los eventos se realiza de manera paralela a la ejecución del programa, sin requerir de instrucciones adicionales. Dado que el MCU incluye dos registros de comparación por cada temporizador, hay 6 terminales en donde se pueden generar respuestas automáticas, a esto se debe que con un ATMega328P se puedan generar 6 señales PWM en forma simultánea.

5.3. Preescalador en los Temporizadores

Un preescalador es un circuito divisor de frecuencia que se antepone a los temporizadores para proporcionarles la capacidad de alcanzar intervalos de tiempo mayores. El circuito tiene 2 componentes principales: un contador de n-bits y un multiplexor, el contador se incrementa en cada ciclo de reloj y con el multiplexor se pueden seleccionar diferentes bits del contador, dando lugar a diferentes frecuencias.

El ATMega328P contiene 2 preescaladores, uno compartido por los temporizadores 0 y 1, y otro solo 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.

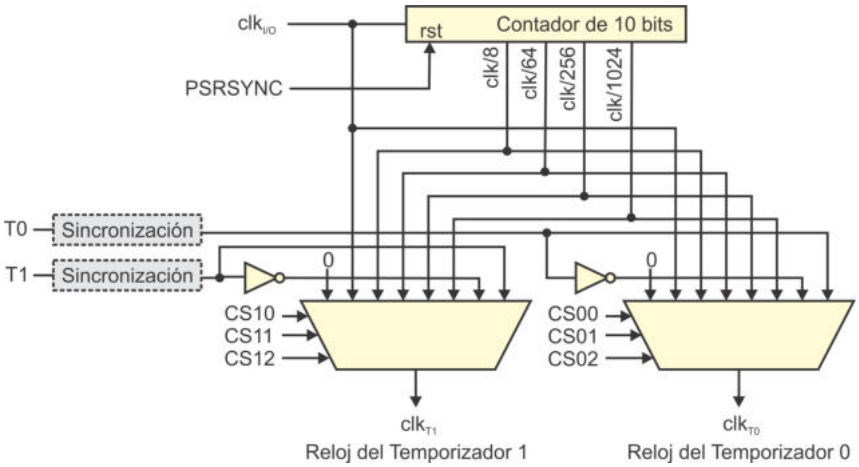


Figura 5.4: Preescalador compartido por los temporizadores 0 y 1

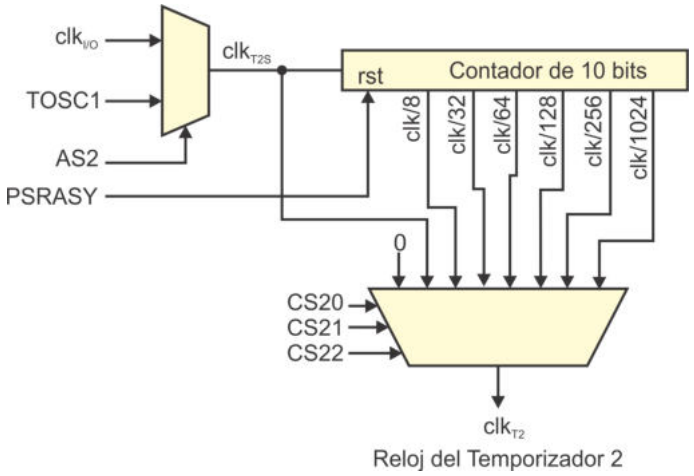


Figura 5.5: Preescalador del temporizador 2

En la Figura 5.4 se muestra la organización del preescalador compartido por los temporizadores 0 y 1, puede notarse que en realidad solo comparten al contador, porque el hardware de selección es independiente.

Los bits de selección (`CSnx`) están en los registros de configuración de los temporizadores, con estos bits se puede seleccionar la señal de reloj sin división, la señal de reloj con un factor de división entre 4 diferentes opciones y una señal externa o su complemento (para considerar flancos de subida o de bajada). Después de un reinicio los bits de selección tienen el valor de 0 y, por lo tanto, no hay señal de reloj.

En la Figura 5.5 se muestra la organización del preescalador utilizado por el temporizador 2, se observa que tiene 6 factores de división y que puede recibir una señal de reloj generada por un oscilador externo (en la entrada TOSC1), permitiendo que el temporizador 2 trabaje a una frecuencia diferente al resto del sistema.

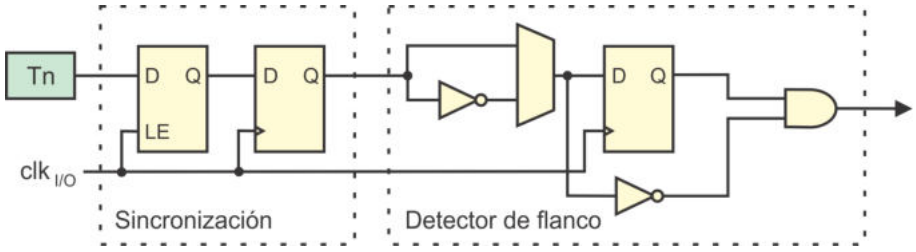


Figura 5.6: Acondicionamiento de las señales externas en los temporizadores 0 y 1

Los contadores de ambos preescaladores tienen una señal de reinicio que se activa con los bits `PSRSYNC` y `PSRASY`, los cuales corresponden con los bits menos significativos del registro de control general de los temporizadores/contadores (`GTCCR`, *General Timer/Counter Control Register*), cuyos bits se muestran a continuación:

REG.	7	6	5	4	3	2	1	0	DIR.
GTCCR	TSM	-	-	-	-	-	PSRASY	PSRSYNC	0x23 (0x43)

- **Bit 7 - TSM:** Modo síncrono en los temporizadores.
  Un nivel alto en este bit hace que se mantenga el valor escrito en `PSRASY` y `PSRSYNC`, ayuda a sincronizar la operación de los temporizadores.
- **Bit 1 - PSRASY:** Reinicio para el preescalador del temporizador 2.
  Un nivel alto en este bit reinicia al contador del preescalador del temporizador 2, el bit se limpia automáticamente por hardware a menos que el bit `TSM` esté en alto.
- **Bit 0 - PSRSYNC:** Reinicio del preescalador de los temporizadores 0 y 1.
  Un nivel alto en este bit reinicia al contador del preescalador de los temporizadores 0 y 1, el bit se limpia automáticamente por hardware a menos que el bit `TSM` esté en alto.

5.4. Temporización Externa

En la Figura 5.4 se muestra que los temporizadores 0 y 1 pueden ser manejados por las señales externas T0 y T1, respectivamente; en estos casos se les conoce como contadores de eventos. En la Figura 5.6 se presentan 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 $f_{clk\_I/O}/2.5$.

El temporizador 2 también puede ser manejado por una señal externa, pero en este caso se utiliza un oscilador que no se sincroniza con el reloj interno (es asíncrono).

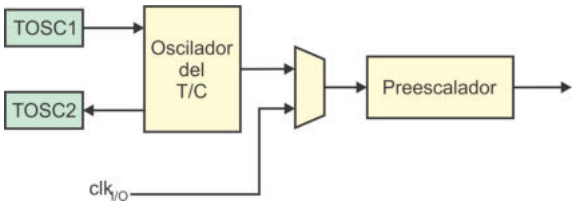


Figura 5.7: Uso de un oscilador externo para el temporizador 2

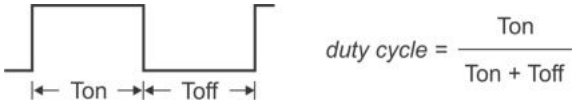


Figura 5.8: Definición del ciclo de trabajo

El hardware está optimizado para ser manejado con un cristal de 32.768 KHz, esta frecuencia es adecuada para que, en combinación con el preescalador, se puedan generar periodos con fracciones o múltiplos de segundos. En la Figura 5.7 se muestra como se puede seleccionar entre un oscilador externo o la señal de reloj interna.

La ventaja principal de usar un oscilador asíncrono para el temporizador 2 es que trabajaría a una frecuencia independiente al resto del sistema, la cual es adecuada para manejar un reloj de tiempo real. Con poco esfuerzo se puede realizar un reloj con visualizadores de 7 segmentos y botones de configuración, así, mientras el temporizador 2 lleva el conteo de segundos, otros recursos manejan a los elementos de visualización y sondean los botones a una frecuencia mucho más alta.

5.5. Modulación por Ancho de Pulso (PWM)

La modulación por ancho de pulso es una técnica para generar "señales analógicas" en salidas de sistemas digitales. 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 5.8 se muestra un periodo de una señal cuadrada con la 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:

$$V_{AVG} = \frac{1}{T}\int_0^T V_p dt = V_p \frac{T_{on}}{T}$$

Para un ciclo de trabajo del 50 % el tiempo en alto es $T_{ON} = \frac{T}{2}$ y por lo tanto $V_{AVG} = \frac{V_p}{2}$. En la mayoría de aplicaciones se conecta directamente el dispositivo a controlar en la salida PWM, pero si es necesario, se puede generar una señal analógica de baja frecuencia al conectar un filtro pasivo RC pasa bajas, como el mostrado en la Figura 5.9.

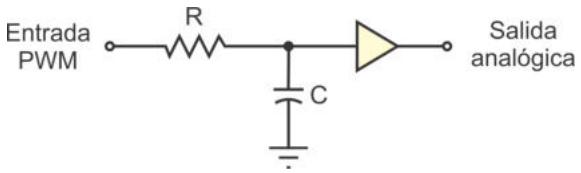


Figura 5.9: Filtro pasa bajas para generar una señal analógica

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 se generan con los temporizadores y sus registros de comparación, puesto que cada temporizador tiene dos comparadores, podrá generar dos señales PWM. En secciones posteriores se describe cómo se deben configurar los temporizadores para proporcionar las señales PWM requeridas por las aplicaciones.

5.5.1. PWM Rápido

Es un modo para generar una señal PWM a una frecuencia alta, el temporizador cuenta de 0 a su valor máximo (MAXVAL) y reinicia, el conteo se realiza continuamente, de manera que, en algún instante de tiempo el temporizador coincide con uno de los registros de comparación `OCRnx` (temporizador n y registro de comparación x).

La señal PWM se genera de la siguiente manera: la salida OCnx es puesta en alto cuando el temporizador realiza una transición de MAXVAL a 0 y se pone en bajo cuando ocurre una coincidencia por comparación. Este modo se conoce como No Invertido. En el modo Invertido ocurre lo contrario para OCnx, ambos modos se muestran en la Figura 5.10, en donde puede notarse que el ancho del pulso está determinado por el valor del registro `OCRnx`.

Por el comportamiento del temporizador, al modo PWM rápido también se le conoce como un modo de pendiente única. La frecuencia de la señal de salida está dada por:

$$f_{PWM} = \frac{f_{clk}}{MAXVAL+1}$$

Hay dos alternativas para cambiar la frecuencia de salida: una consiste en modificar el valor de MAXVAL y la otra en utilizar al preescalador para modificar el valor de f_{clk}. La modificación de MAXVAL está permitida en algunos de los modos PWM.

Un problema se podría generar si se pretende escribir en el registro de comparación un valor menor al que en ese instante contiene el temporizador, en un ciclo de reloj no habría coincidencia y se reflejaría en una variación de la frecuencia de la

señal de salida. Para evitarlo, el acceso al registro `OCRnx` se realiza por medio de un *buffer* doble. Cualquier instrucción que intente escribir en `OCRnx` lo hace en un *buffer* intermedio y la escritura en el registro `OCRnx` se realiza en el momento que el temporizador pasa de MAXVAL a 0, con ello, la frecuencia de la señal de salida se mantiene estable.

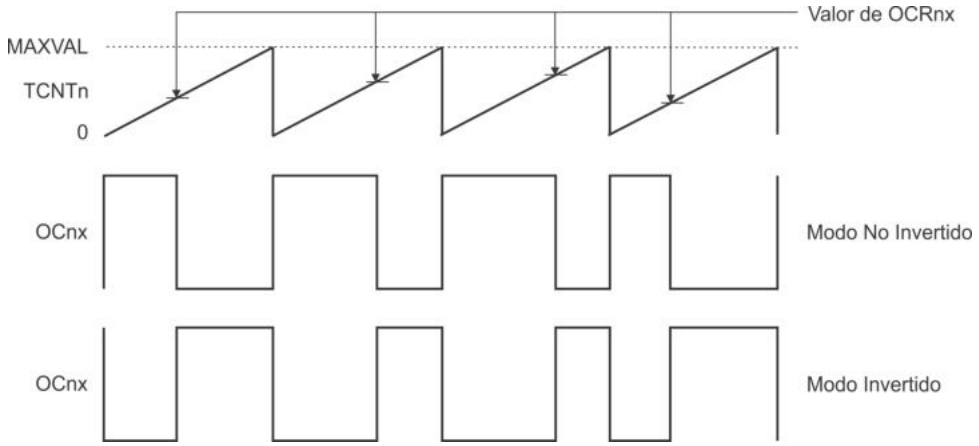


Figura 5.10: Señales de PWM rápido

5.5.2. PWM con Fase Correcta

En este modo, el temporizador cuenta en forma ascendente (de 0 a MAXVAL), para después contar en forma descendente (de MAXVAL a 0). La modificación de la terminal de salida OCnx se realiza en las coincidencias por comparación con el registro `OCRnx`.

En el modo No Invertido, la salida OCnx 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 OCnx. El comportamiento de la señal de salida en este modo de PWM se muestra en la Figura 5.11, en donde se observa que, para el modo no invertido, el pulso está centrado con el valor cero del temporizador (están en fase).

Por el comportamiento del temporizador, a este modo de PWM también se le conoce como un modo de pendiente doble y la frecuencia de la señal generada está dada por:

$$f_{PWM} = \frac{f_{clk}}{2(MAXVAL+1)}$$

En este modo también se cuenta con un *buffer* doble para la escritura en el registro `OCRnx`, las escrituras reales se realizan cuando el temporizador alcanza su valor máximo. La frecuencia de la señal de salida también se puede modificar con cambios en MAXVAL o con el uso del preescalador. No obstante, la modificación del máximo

en tiempo de ejecución da lugar a la generación de señales que no son simétricas, este efecto se muestra en la Figura 5.12.

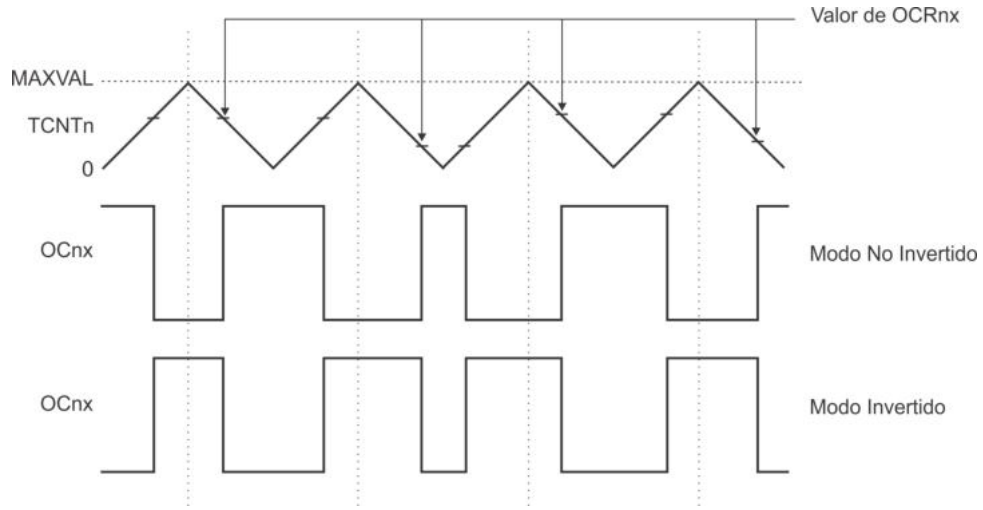


Figura 5.11: Señales de PWM con fase correcta

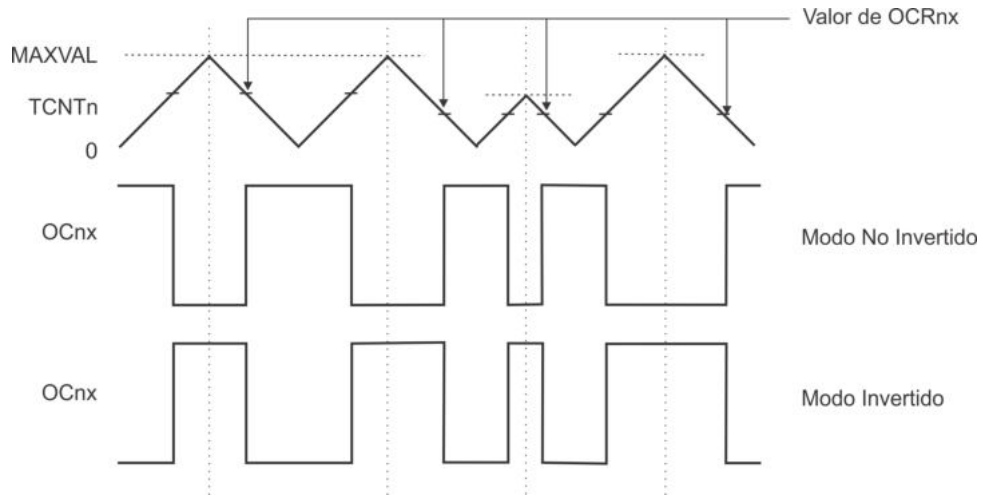


Figura 5.12: Señales de PWM con fase correcta y máximo variable

5.5.3. PWM con Fase y Frecuencia Correcta

Este modo es muy similar al modo de fase correcta, ambos 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 `OCRnx`, el modo de fase correcta actualiza al registro `OCRnx` cuando `TCNTn` llega a su valor máximo, y el modo de fase y frecuencia correcta lo hace cuando el registro `TCNTn` llega a cero.

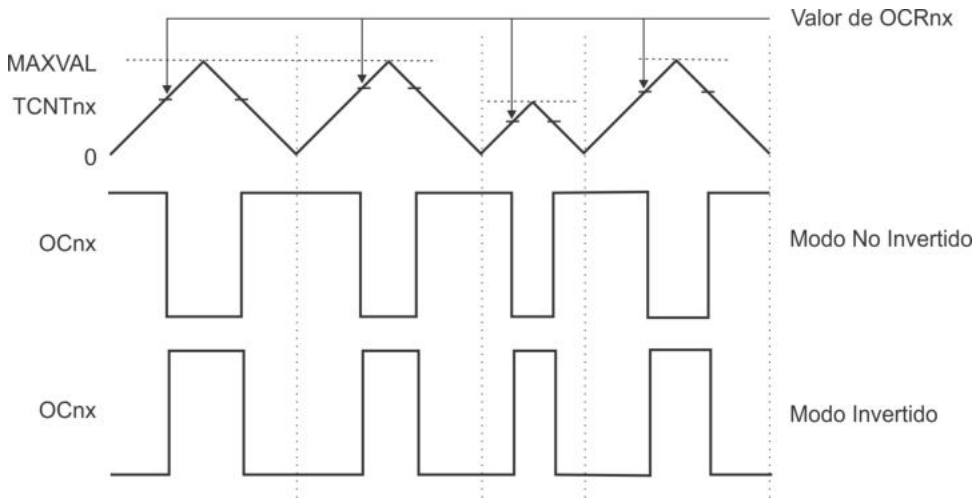


Figura 5.13: Señales de PWM con fase y frecuencia correcta

Si se utiliza un valor constante para el máximo, se tiene el mismo efecto al emplear uno u otro modo, pero si el valor del máximo se va a modificar en tiempo de ejecución, con el modo de fase correcta se pueden generar formas asimétricas, lo cual no ocurre con el modo de fase y frecuencia correcta. En la Figura 5.13 puede notarse que la señal de salida es simétrica aun después de modificar el valor máximo del temporizador.

Este modo solo puede ser generado por el temporizador 1 con máximos variables, el valor para MAXVAL se puede definir en los registros `OCR1A` o `ICR1`, según el modo de operación.

5.6. El Temporizador 0

En la Figura 5.14 se muestra la organización del temporizador 0 y en la Tabla 5.3 se listan los Registros I/O utilizados por el recurso, todos son de 8 bits. Por su dirección, se observa que solo `TIMSK0` es un Registro I/O Extendido.

Tabla 5.3: Registros del temporizador 0

Registro	Dirección	Operación
TCNT0	0x26 (0x46)	Registro del temporizador 0
OCR0A	0x27 (0x47)	Comparador A del temporizador 0
OCR0B	0x28 (0x48)	Comparador B del temporizador 0
TCCR0A	0x24 (0x44)	Registro A para la configuración y control
TCCR0B	0x25 (0x45)	Registro B para la configuración y control
TIFR0	0x15 (0x35)	Registro de banderas
TIMSK0	(0x6E)	Máscara para activar las interrupciones

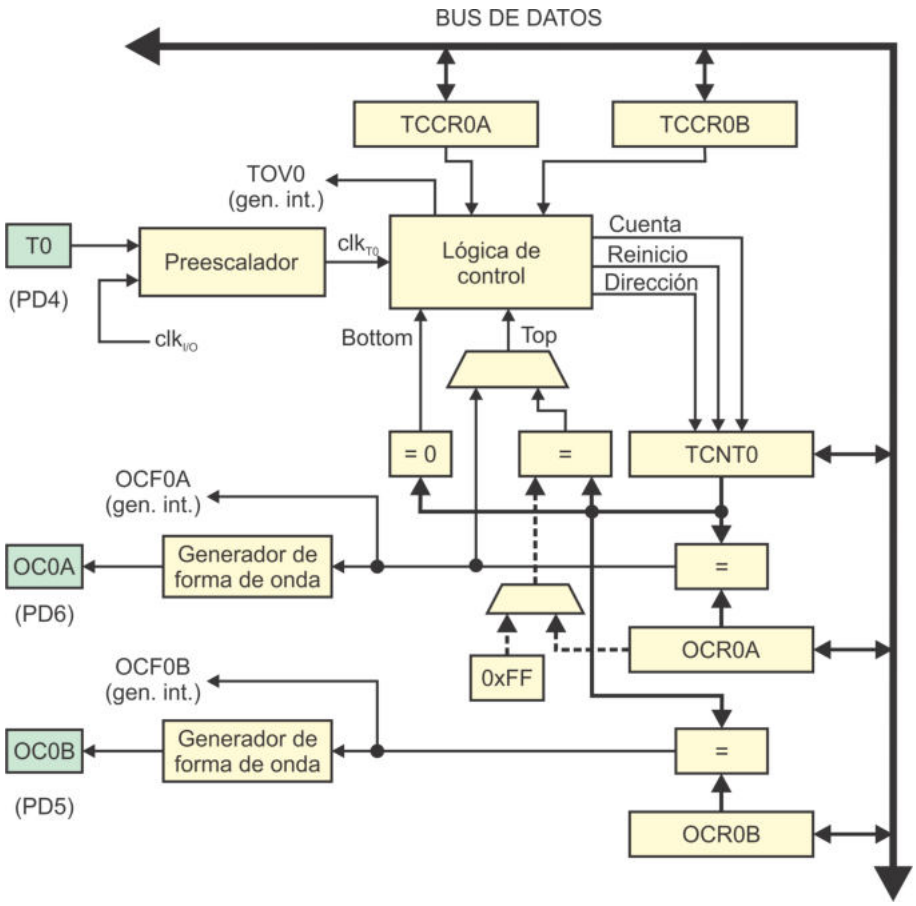


Figura 5.14: Organización del temporizador 0

5.6.1. Configuración y Control del Temporizador 0

El temporizador 0 es controlado por los registros `TCCR0A` y `TCCR0B` (*Timer/Counter Control Register*), cuyos bits son descritos a continuación:

REG.	7	6	5	4	3	2	1	0
TCCR0A	COM0A1	COM0A0	COM0B1	COM0B0	-	-	WGM01	WGM00
TCCR0B	FOC0A	FOC0B	-	-	WGM02	CS02	CS01	CS00

La operación de los bits se revisa por grupos:

- **Bits WGM0[2:0]:** Definen el modo para la generación de una forma de onda (*Waveform Generation Mode*), en otras palabras, en estos bits se configura el modo de operación del temporizador 0.
- **Bits COM0x[1:0]:** En estos bits se configura una respuesta automática ante coincidencias por comparación (*Compare Output Mode*). La x puede ser A o B, debido a que el recurso tiene dos comparadores.
- **Bits FOC0x:** Forzan u obligan un evento de comparación, si se configuró una

respuesta automática, esta también va a realizarse. La x puede ser A o B por los dos comparadores.

- **Bits CS0[2:0]:** Bits de selección de la fuente de reloj (*Clock Select*).

Modos de Operación

En los bits `WGM0[2:0]` se configura el modo de operación del temporizador 0, en la Tabla 5.4 se muestran los diferentes modos, los cuales son:

- **Modo 0:** El temporizador 0 solo genera eventos de desbordamiento.
- **Modos 1 y 5:** Modos de PWM con fase correcta, el temporizador automáticamente se incrementa de cero a su valor máximo y decrementa del valor máximo a cero, al coincidir con los registros de comparación conmutará una salida según la configuración de los bits `COM0x[1:0]`.
- **Modo 2:** Es un modo CTC (*clear timer on compare match*), en donde el temporizador 0 se limpia automáticamente después de una coincidencia con el registro `OCR0A`.
- **Modos 3 y 7:** Modos de PWM rápido, el temporizador automáticamente se incrementa de cero a su valor máximo, al coincidir con los registros de comparación conmutará una salida según la configuración de los bits `COM0x[1:0]`.

Tabla 5.4: Modos de operación del temporizador 0

Modo	WGM02	WGM01	WGM00	Descripción	MAXVAL
0	0	0	0	Normal	0xFF
1	0	0	1	PWM con fase correcta	0xFF
2	0	1	0	CTC, el temporizador se limpia ante una coincidencia	OCR0A
3	0	1	1	PWM rápido	0xFF
4	1	0	0	Reservado	-
5	1	0	1	PWM con fase correcta	OCR0A
6	1	1	0	Reservado	-
7	1	1	1	PWM rápido	OCR0A

Para los modos PWM, si el valor máximo para el temporizador es 0xFF se pueden generar dos señales en las salidas OC0A y OC0B, pero cuando el valor máximo se toma del registro `OCR0A` solo se puede generar una señal en la terminal OC0B.

Respuesta automática

En los bits `COM0x[1:0]` se configura la respuesta automática para las salidas OC0x ante coincidencias por comparación, la x puede ser A o B porque el recurso tiene dos comparadores. La configuración de estos bits es diferente para los modos normal y CTC, con respecto a los modos PWM, como se muestra en la Tabla 5.5.

Tabla 5.5: Configuración de la respuesta automática (x puede ser A o B)

COM0x1	COM0x0	Descripción
Modos Normal y CTC		
0	0	OC0x desconectado
0	1	Conmuta OC0x
1	0	OC0x es puesto en bajo
1	1	OC0x es puesto en alto
Modos PWM		
0	0	OC0x desconectado
0	1	Reservado (sin uso)
1	0	Modo no invertido
1	1	Modo invertido

Tabla 5.6: 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	$CLK_{I/O}/8$
0	1	1	$CLK_{I/O}/64$
1	0	0	$CLK_{I/O}/256$
1	0	1	$CLK_{I/O}/1024$
1	1	0	Fuente externa en T0, por flanco de bajada
1	1	1	Fuente externa en T0, por flanco de subida

Selección de Reloj

La selección de la señal de reloj está determinada por los bits `CS0[2:0]`, los cuales son descritos en la Tabla 5.6. Estos bits se conectan directamente a los bits de selección del multiplexor del preescalador (Sección 5.3). Después de un reinicio, el temporizador 0 está detenido porque no tiene una señal de reloj que lo active.

5.6.2. Interrupciones debidas al Temporizador 0

El temporizador 0 puede interrumpir a la CPU por 3 eventos: un evento de desbordamiento y dos de coincidencia por comparación. Los eventos quedan registrados en `TIFR0` (*Time/Counter Interrupt Flag Register*), cuyos bits son:

REG.	7	6	5	4	3	2	1	0
TIFR0	-	-	-	-	-	OCF0B	OCF0A	TOV0

- **Bit 2 - OCF0B:** (*Output Compare Match Flag*) Indica una coincidencia del temporizador 0 con el registro `OCR0B`.
- **Bit 1 - OCF0A:** La coincidencia del temporizador 0 es con el registro `OCR0A`.

- **Bit 0 - TOV0:** (*Timer/Counter Overflow*) Indica un desbordamiento del temporizador 0.

Las banderas se limpian automáticamente si se emplean interrupciones o reescribiendo un 1 lógico si se usa sondeo. Para que los eventos provoquen una interrupción a la CPU, se deben poner en alto sus habilitadores individuales, además del habilitador global, en el registro `TIMSK0` (*Timer/Counter Interrupt Mask Register*) se realizan estas habilitaciones, sus bits son:

REG.	7	6	5	4	3	2	1	0
TIMSK0	-	-	-	-	-	OCIE0B	OCIE0A	TOIE0

- **Bit 2 - OCIE0B:** (*Output Compare Match Interrupt Enable*) Habilita la interrupción por una coincidencia con el registro `OCR0B`.
- **Bit 1 - OCIE0A:** Habilita la interrupción por una coincidencia con el registro `OCR0A`.
- **Bit 0 - TOIE0:** (*Timer/Counter Interrupt Enable*) Habilita la interrupción por desbordamiento del temporizador 0.

En la Tabla 5.7 se muestran los vectores de interrupción para los eventos del temporizador 0, la dirección se utiliza al programar en ensamblador y el nombre para lenguaje C.

Tabla 5.7: Vectores de interrupción para el temporizador 0

Dirección	Evento	Descripción
0x01C	TIMER0_COMPA	El temporizador 0 coincide con el registro OCR0A
0x01E	TIMER0_COMPB	El temporizador 0 coincide con el registro OCR0B
0x020	TIMER0_OVF	Desbordamiento del temporizador 0

5.6.3. Ejemplos con el Temporizador 0

Ejemplo 5.1 - Generación de una Señal Periódica
Escriba un programa que genere una señal de 5 kHz, considerando que un ATMega328P estará operando a 1 MHz. Utilice: desbordamientos atendidos por interrupción (con salida en PD0) y respuesta automática ante coincidencias por comparación y modo CTC (la salida será en OC0A, que está en PD6).

Si se va a generar una señal con una frecuencia f = 5 kHz , entonces, su periodo debe ser T = 200 us. Si se considera un ciclo de trabajo del 50 %, se tiene $T_{ALTO} = 100us$ y $T_{BAJO} = 100us$. En la Figura 5.15 se muestra la generación de la señal, primero será en PD0 y después en PD6 (OC0A).

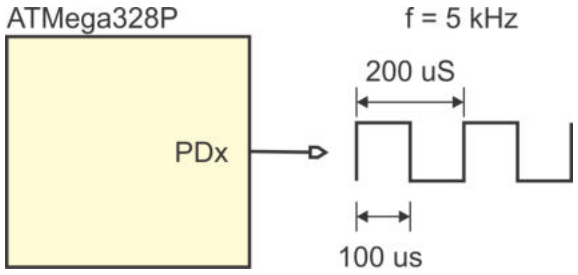


Figura 5.15: Generación de una señal a 5 kHz

Al emplear un oscilador de 1 MHz, sin preescalador, el temporizador se va a incrementar cada microsegundo, por lo tanto, el temporizador debe contar 100 eventos antes de conmutar la salida.

Desbordamientos atendidos por interrupción

Para que el temporizador 0 desborde transcurridos 100 ciclos de reloj, debe iniciar con el valor de 256 - 100 = 156 = 0x9C. En la sección de configuración del programa se debe: establecer el valor inicial del temporizador, configurar para que tenga una fuente de reloj y habilitar su interrupción por desbordamiento; en el lazo infinito el MCU permanece ocioso. Transcurridos 100 ciclos de reloj, el temporizador interrumpe a la CPU, en su ISR conmuta la salida y recarga su valor para que los desbordamientos ocurran con el mismo periodo.

La solución en lenguaje C para este problema es:

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

ISR(TIMER0_OVF_vect) {

   TCNT0 = 156;            // Valor de recarga
   PIND = 0x01;            // Conmuta la salida
}

int  main() {

   DDRD = 0xFF;            // Salida en PD0
   PORTD =0x00;            // Inicia en 0

   // Configuración del temporizador 0

   TCNT0 = 156;
   TCCR0A = 0x00;
   TCCR0B = 0x01;          // Reloj sin preescala
   TIMSK0 = 0x01;          // Interrupción por desbordamiento

   sei();                  // Habilitador global de interrupciones
```

```
    while(1)
        asm("NOP");          // Ocioso en el lazo infinito
}
```

La bandera de desbordamiento se limpia en forma automática al emplear interrupciones. La solución es funcional y en la ISR se puede hacer otra tarea periódica, sin embargo, el periodo no es exacto porque se pierden algunos ciclos de reloj en lo que se da paso a la ISR y se realiza la recarga.

Modo CTC con respuesta automática

Puesto que el temporizador inicia en 0, en el registro de comparación `OCR0A` se debe colocar el número 99 para que la coincidencia sea a los 100 eventos. Además, en la sección de configuración del programa se debe: seleccionar el modo CTC (`WGM0[2:0]` = "010") y configurar la conmutación automática de OC0A (`COM0A[1:0]` = "01"); en el lazo infinito el MCU permanece ocioso. El reinicio del temporizador y la generación de la salida se realizan por hardware y, por lo tanto, no se requiere el uso de interrupciones.

El código en lenguaje C para esta versión de la solución del problema es:

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

int main() {

    DDRD = 0xFF;            // Salida en OC0A
    PORTD =0x00;

    // Configuración del temporizador 0

    OCR0A = 99;             // Valor para la comparación
    TCCR0A = 0x42;          // Respuesta automática y modo CTC
    TCCR0B = 0x01;          // Reloj sin preescala

    while(1)
        asm("NOP");         // Ocioso en el lazo infinito
}
```

El periodo es muy exacto porque todo se realiza por hardware, para hacer otras tareas periódicas se debe habilitar la interrupción por coincidencia en la comparación con `OCR0A`.

Para generar señales a otras frecuencias se debe adecuar el valor del registro `OCR0A`, en caso de que la frecuencia sea baja y se requiera un valor que no alcance en 8 bits, se puede emplear el factor de preescala que mejor convenga.

Otra labor habitual en los temporizadores es la generación de señales PWM, en el Ejemplo 5.2 se ilustra esta tarea.

Ejemplo 5.2 - Generación de una Señal PWM
Empleando el temporizador 0 de un ATMega328P, genere una señal PWM en modo no invertido en la terminal PD6, que corresponde con OC0A, con el ciclo de trabajo determinado por el valor del Puerto B, el hardware requerido se muestra en la Figura 5.16.

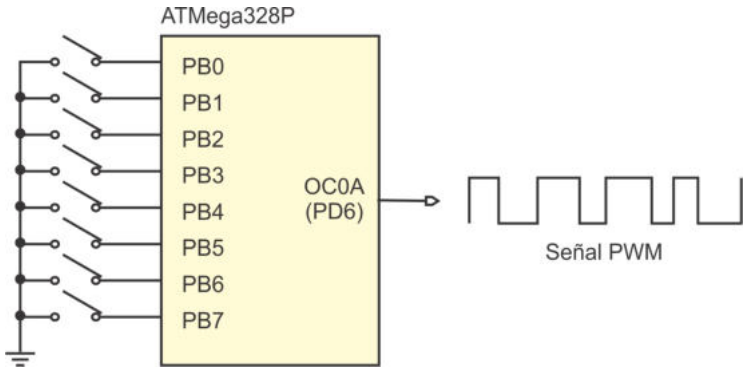


Figura 5.16: Generación de una señal PWM a partir del valor del Puerto B

Considerando que no hay restricción para la frecuencia, se utilizará el modo 3 del temporizador 0 (`WGM0[2:0]` = "011"), que corresponde con un modo de PWM rápido. Los bits `COM0A[1:0]` deben tener "10" para un modo no invertido.

En el lazo infinito se modificará al registro de comparación (`OCR0A`) cada que haya un cambio en el Puerto B. La solución en lenguaje C es:

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

int main(){

    DDRB = 0x00;          // Entrada para definir el ancho de pulso
    PORTB = 0xFF;         // Pull-up
    DDRD = 0xFF;          // Salida en OC0A (PD6)

    TCCR0A = 0x83;        // PWM rápido
                          // COM0A[1:0] = "10", WGM0[2:0] = "011"
    TCCR0B = 0x01;        // Reloj sin preescala

    while(1) {
        OCR0A = PINB;
    }
}
```

En el código previo se observa que el temporizador 0 no utiliza al preescalador, por ello, si el microcontrolador trabaja con su oscilador interno de 1 MHz, la frecuencia de la señal PWM generada será de:

$$f_{PWM} = \frac{1MHz}{256} = 3.90kHz$$

Tabla 5.8: Registros del temporizador 1

Registro	Dirección	Operación
TCNT1L	(0x84)	Registro del temporizador 1 (parte baja)
TCNT1H	(0x85)	Registro del temporizador 1 (parte alta)
ICR1L	(0x86)	Registro de captura de entrada (parte baja)
ICR1H	(0x87)	Registro de captura de entrada (parte alta)
OCR1AL	(0x88)	Comparador A del temporizador 1 (parte baja)
OCR1AH	(0x89)	Comparador A del temporizador 1 (parte alta)
OCR1BL	(0x8A)	Comparador B del temporizador 1 (parte baja)
OCR1BH	(0x8B)	Comparador B del temporizador 1 (parte alta)
TCCR1A	(0x80)	Registro A para la configuración y control
TCCR1B	(0x81)	Registro B para la configuración y control
TCCR1C	(0x82)	Registro C para la configuración y control
TIFR1	0x16 (0x36)	Registro de banderas
TIMSK1	(0x6F)	Máscara para activar las interrupciones

5.7. El Temporizador 1

En la Figura 5.17 se muestra la organización del temporizador 1 y en la Tabla 5.8 se listan los Registros I/O utilizados por el recurso. Dado que el temporizador 1 es de 16 bits, muchos de sus registros se forman por dos partes, como se puede ver en la misma tabla. Por su dirección, se observa que solo el registro de banderas `TIFR1` es un Registro I/O, todos los demás son Registros I/O Extendidos.

Cuando se programa en ensamblador, las instrucciones deben utilizar los registros de 8 bits que se describen en la Tabla 5.8, sin embargo, en lenguaje C cada par se puede tratar como un registro de 16 bits, empleando los nombres: `TCNT1`, `ICR1`, `OCR1A` y `OCR1B`.

5.7.1. Configuración y Control del Temporizador 1

El temporizador 1 es controlado por los Registros I/O Extendidos: `TCCR1A`, `TCCR1B` y `TCCR1C` (*Timer/Counter Control Register*), cuyos bits son descritos a continuación:

REG.	7	6	5	4	3	2	1	0
TCCR1A	COM1A1	COM1A0	COM1B1	COM1B0	-	-	WGM11	WGM10
TCCR1B	ICNC1	ICES1	-	WGM13	WGM12	CS12	CS11	CS10
TCCR1C	FOC1A	FOC1B	-	-	-	-	-	-

La operación de los bits se revisa por grupos:

- **Bits WGM1[3:0]:** Definen el modo para la generación de una forma de onda (*Waveform Generation Mode*), en otras palabras, en estos bits se configura el modo de operación del temporizador 1.
- **Bits COM1x[1:0]:** Configuran una respuesta automática ante coincidencias por comparación (*Compare Output Mode*). La x puede ser A o B.

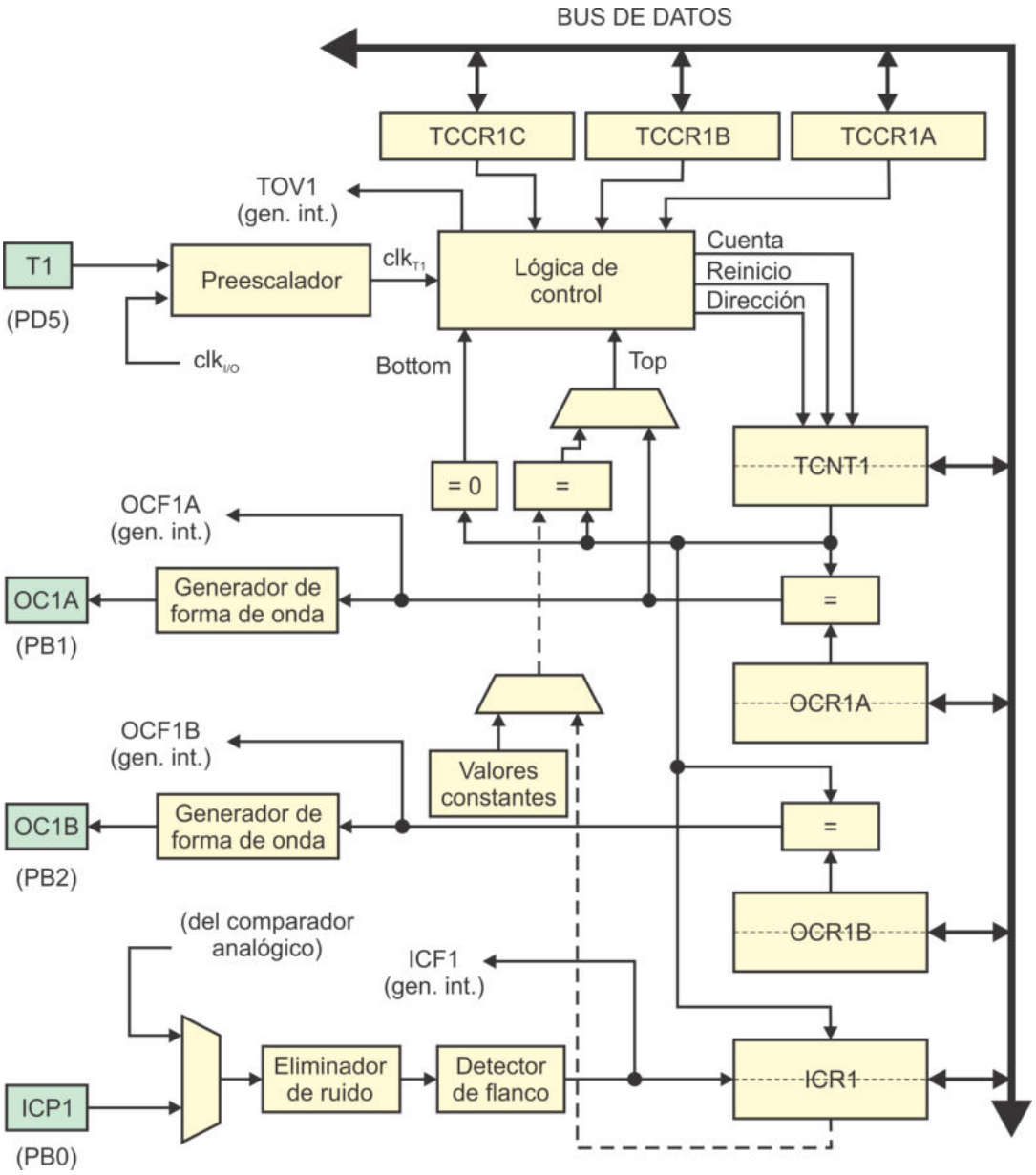


Figura 5.17: Organización del temporizador 1

- **Bits FOC1x:** Forzan u obligan un evento de comparación, si se configuró una respuesta automática, esta también va a realizarse. La x puede ser A o B por los dos comparadores.
- **Bits CS1[2:0]:** Bits de selección de la fuente de reloj (*Clock Select*).
- **Bits ICNC1 e ICES1:** A través de estos bits se configura el recurso de captura de entrada.

Modos de Operación

En los bits `WGM1[3:0]` se configura el modo de operación del temporizador 1, en la Tabla 5.9 se muestran los diferentes modos, los cuales son:

- **Modo 0:** El temporizador 0 solo genera eventos de desbordamiento.
- **Modos 1, 2, 3, 10 y 11:** Modos de PWM con fase correcta, el temporizador automáticamente se incrementa de cero a su valor máximo y decrementa del valor máximo a cero, al coincidir con los registros de comparación conmutará una salida según la configuración de los bits `COM1x[1:0]`.
- **Modos 4 y 12:** Son modos CTC (*clear timer on compare match*), en donde el temporizador 1 se limpia automáticamente después de una coincidencia con el registro `OCR1A` o `ICR1`, según el modo.
- **Modos 5, 6, 7, 14 y 15:** Modos de PWM rápido, el temporizador automáticamente se incrementa de cero a su valor máximo, al coincidir con los registros de comparación conmutará una salida según la configuración de los bits `COM1x[1:0]`.
- **Modos 8 y 9:** Modos de PWM con fase y frecuencia correcta, son similares a los modos de PWM con fase correcta, cambia el momento en que se actualiza el registro de comparación, el modo de fase correcta actualiza al registro `OCR1x` cuando `TCNT1` llega a su valor máximo, y el modo de fase y frecuencia correcta lo hace cuando el registro `TCNT1` llega a cero.

En el registro `ICR1` se realiza la captura del temporizador 1, sin embargo, este registro es empleado en los modos 8, 10, 12 y 14 para establecer el valor máximo del temporizador. Al asignarle una función alterna a este registro, la terminal ICP1 será una entrada o salida general y no se podrán realizar tareas de captura.

En los modos PWM donde MAXVAL es constante o se establece en el registro `ICR1`, se pueden generar dos señales en las salidas OC1A y OC1B, pero cuando el valor máximo se toma del registro `OCR1A` solo se puede generar una señal en la terminal OC1B.

Tabla 5.9: Modos de operación del temporizador 1

Modo	WGM13	WGM12	WGM11	WGM10	Descripción	MAXVAL
0	0	0	0	0	Normal	0xFFFF
1	0	0	0	1	PWM con fase correcta (8 bits)	0x00FF
2	0	0	1	0	PWM con fase correcta (9 bits)	0x01FF
3	0	0	1	1	PWM con fase correcta (10 bits)	0x03FF
4	0	1	0	0	CTC, el temporizador se limpia ante una coincidencia	OCR1A
5	0	1	0	1	PWM rápido (8 bits)	0x00FF
6	0	1	1	0	PWM rápido (9 bits)	0x01FF
7	0	1	1	1	PWM rápido (10 bits)	0x03FF
8	1	0	0	0	PWM con fase y frecuencia correcta	ICR1
9	1	0	0	1	PWM con fase y frecuencia correcta	OCR1A
10	1	0	1	0	PWM con fase correcta	ICR1
11	1	0	1	1	PWM con fase correcta	OCR1A
12	1	1	0	0	CTC, el temporizador se limpia ante una coincidencia	ICR1
13	1	1	0	1	Reservado	-
14	1	1	1	0	PWM rápido	ICR1
15	1	1	1	1	PWM rápido	OCR1A

Tabla 5.10: Configuración de la respuesta automática (x puede ser A o B)

COM1x1	COM1x0	Descripción
Modos Normal y CTC		
0	0	OC1x desconectado
0	1	Conmuta OC1x
1	0	OC1x es puesto en bajo
1	1	OC1x es puesto en alto
Modos PWM		
0	0	OC1x desconectado
0	1	Reservado (sin uso)
1	0	Modo no invertido
1	1	Modo invertido

Respuesta automática

En los bits `COM1x[1:0]` se configura la respuesta automática para las salidas OC1x ante coincidencias por comparación, la x puede ser A o B porque el recurso tiene dos comparadores. La configuración de estos bits es diferente para los modos normal y CTC, con respecto a los modos PWM, como se muestra en la Tabla 5.10.

Selección de Reloj

La selección de la señal de reloj está determinada por los bits `CS1[2:0]`, los cuales son descritos en la Tabla 5.11. Estos bits se conectan directamente a los bits de selección del multiplexor del preescalador (Sección 5.3). Después de un reinicio, el temporizador 1 está detenido porque no tiene una señal de reloj que lo active.

Tabla 5.11: 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	$CLK_{I/O}/8$
0	1	1	$CLK_{I/O}/64$
1	0	0	$CLK_{I/O}/256$
1	0	1	$CLK_{I/O}/1024$
1	1	0	Fuente externa en T1, por flanco de bajada
1	1	1	Fuente externa en T1, por flanco de subida

Captura de Entrada

Un cambio en la terminal ICP1 hace que el valor de `TCNT1` se copie en el registro `ICR1`. El registro `TCCR1B` incluye a los bits `ICNC1` e `ICES1` para el manejo de este recurso.

El bit `ICNC1` (*Input Capture Noise Canceler*) activa un eliminador de ruido, haciendo que la señal de entrada se filtre por cuatro muestras sucesivas, de manera que una captura se realiza solo si el cambio en ICP1 es estable.

Con el bit `ICES1` (*Input Capture Edge Select*) se selecciona el flanco de activación de la captura, un 0 en `ICES1` hace que la captura se realice con un flanco de bajada en ICP1 y con un 1 el flanco de activación es de subida.

Debe notarse que el recurso de captura generalmente está activo, sin embargo, mientras la bandera `ICF1` no se evalúe o no se habilite su interrupción, el valor del registro `ICR1` será ignorado. Las capturas se omiten en los modos 8, 10, 12 y 14, en donde el registro `ICR1` es usado para establecer el valor máximo del temporizador 1.

5.7.2. Interrupciones debidas al Temporizador 1

El temporizador 1 puede interrumpir a la CPU por 4 eventos: un evento de desbordamiento, dos de coincidencia por comparación y uno por captura de entrada. Los eventos quedan registrados en `TIFR1` (*Time/Counter Interrupt Flag Register*), cuyos bits son:

REG.	7	6	5	4	3	2	1	0
TIFR1	-	-	ICF1	-	-	OCF1B	OCF1A	TOV1

- **Bit 5 - ICF1:** (*Input Capture Flag*) Indica un evento de captura de entrada.
- **Bit 2 - OCF1B:** (*Output Compare Match Flag*) Indica una coincidencia del temporizador 1 con el registro `OCR1B`.
- **Bit 1 - OCF1A:** La coincidencia del temporizador 1 es con el registro `OCR1A`.

- **Bit 0 - TOV1:** (*Timer/Counter Overflow*) Indica un desbordamiento del temporizador 1.

Las banderas se limpian automáticamente si se emplean interrupciones o reescribiendo un 1 lógico si se usa sondeo. Para que los eventos provoquen una interrupción a la CPU, se deben poner en alto sus habilitadores individuales, además del habilitador global, en el registro `TIMSK1` (*Timer/Counter Interrupt Mask Register*) se realizan estas habilitaciones, sus bits son:

REG.	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
TIMSK1	-	-	ICIE1	-	-	OCIE1B	OCIE1A	TOIE1

- **Bit 5 - ICIE1:** (*Input Capture Interrupt Enable*) Habilita la interrupción por eventos de captura de entrada.
- **Bit 2 - OCIE1B:** (*Output Compare Match Interrupt Enable*) Habilita la interrupción por una coincidencia con el registro `OCR1B`.
- **Bit 1 - OCIE1A:** Habilita la interrupción por una coincidencia con el registro `OCR1A`.
- **Bit 0 - TOIE1:** (*Timer/Counter Interrupt Enable*) Habilita la interrupción por desbordamiento del temporizador 1.

En la Tabla 5.12 se muestran los vectores de interrupción para los eventos del temporizador 1, la dirección se utiliza al programar en ensamblador y el nombre para lenguaje C.

Tabla 5.12: Vectores de interrupción para el temporizador 1

Dirección	**Evento**	**Descripción**
0x014	TIMER1_CAPT	Captura de entrada con el temporizador 1
0x016	TIMER1_COMPA	El temporizador 1 coincide con el registro OCR1A
0x018	TIMER1_COMPB	El temporizador 1 coincide con el registro OCR1B
0x01A	TIMER1_OVF	Desbordamiento del temporizador 1

5.7.3. 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 muy significativo.

Para evitar estos conflictos y poder realizar escrituras y lecturas "al vuelo", es decir, sin detener al temporizador 1, el hardware 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:

```
LDS  R16, TCNT1L     ; Lee 16 bits, el byte alto queda en
                     ; el registro temporal
LDS  R17, TCNT1H     ; Lee el 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)   ; Carga el byte bajo de la constante
LDI R17, HIGH(1500)  ; Carga el byte alto de la constante
STS TCNT1H, R17      ; Escribe en el registro temporal
STS TCNT1L, R16      ; Escritura de 16 bits
```

De manera que no es posible tener acceso solo a un byte del temporizador. En alto nivel se pueden emplear datos de 16 bits, pudiendo ser variables o registros, el orden de acceso se genera cuando un programa se traduce a bajo nivel.

5.7.4. Ejemplos con el Temporizador 1

Un aspecto interesante en sistemas basados en microcontroladores es el uso simultáneo de recursos, en el siguiente ejemplo se combina el uso de la interrupción externa 0 con el temporizador 1.

Ejemplo 5.3 - Generación de un Tono
Configure al ATMega328P para que genere un tono con una frecuencia aproximada de 440 Hz, mientras se mantiene presionado un botón conectado en PD2, que corresponde a la INT0. En la Figura 5.18 se muestra la conexión de los elementos de hardware.

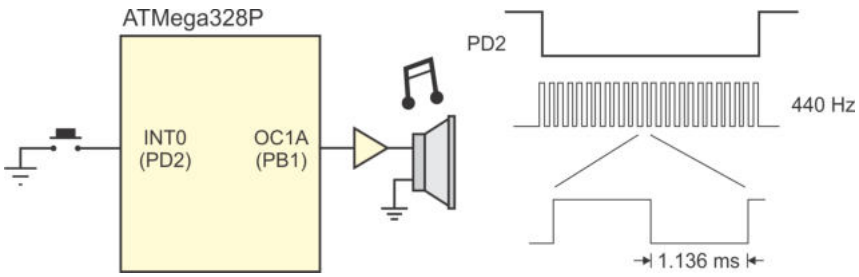


Figura 5.18: Generación de un tono al presionar un botón

Dado que la frecuencia del tono será de f = 440 Hz , entonces, su periodo debe ser T = 2272.72 us, lo que significa T_{ALTO} = 1136.36 us y T_{BAJO} = 1136.36 us. Si el ATMega328P trabaja con un oscilador de 1 MHz, el temporizador se va a incrementar cada microsegundo.

Lo más conveniente es generar el tono configurando al temporizador en un modo CTC con respuesta automática, redondeando el valor para el registro `OCR1A` a 1135 (puesto que inicia en 0), sin embargo, el tono solo se debe generar con el botón presionado, de manera que en la ISR de la interrupción externa se va a activar o quitar el reloj del temporizador.

La interrupción externa se debe generar con cualquier flanco en la señal de entrada y por software se debe evaluar la terminal PD2, si tiene 0 es que el botón fue presionado y si tiene 1 significa que el botón ha sido liberado. El código en lenguaje C para la solución del problema es:

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

ISR(INT0_vect) {

    if(!(PIND & 0x04)) {      // Botón presionado
        TCCR1B = 0x09;        // Modo 4 (CTC), reloj sin preescala
    }
    else {                    // Botón liberado
        PORTB = 0x00;         // Sin carga
        TCCR1B = 0x00;        // Sin reloj
    }
}

int main() {

    DDRB = 0xFF;              // Salida para OC1A
    DDRD = 0x00;              // Entrada para el botón
    PORTD = 0xFF;

// Configuración de la interrupción externa 0
    EICRA = 0x01;             // Cualquier cambio en PD2
```

```
    EIMSK = 0x01;

// Configuración del temporizador 1
    OCR1A = 1135;              // Valor de recarga
    TCCR1A = 0x40;             // Respuesta automática
    TCCR1B = 0x00;             // Inicia sin reloj
    TCCR1C = 0x00;

    sei();                     // Habilitador global de interrupciones

    while(1)
        asm("NOP");
}
```

En el código anterior solo se tiene una ISR porque el tono se genera con una respuesta automática en la terminal OC1A, utilizando el modo 4 del temporizador 1, que corresponde a un modo CTC con valor máximo en el registro `OCR1A`.

Un servomotor es un dispositivo electromecánico que integra un motor, un sistema de engranes y un control de posición en lazo cerrado. El eje del motor tiene un intervalo de movimiento entre 0° y 180°, para posicionarlo, en su entrada de control se debe introducir una señal PWM con una frecuencia de 50 Hz y el ancho de pulso determina la ubicación del eje. En la Figura 5.19 se muestra el ancho requerido para 3 posiciones estratégicas: 0°, 90° y 180°. En el siguiente ejemplo se muestra como el manejo de un servomotor con un ATMega328P se simplifica al utilizar al temporizador 1.

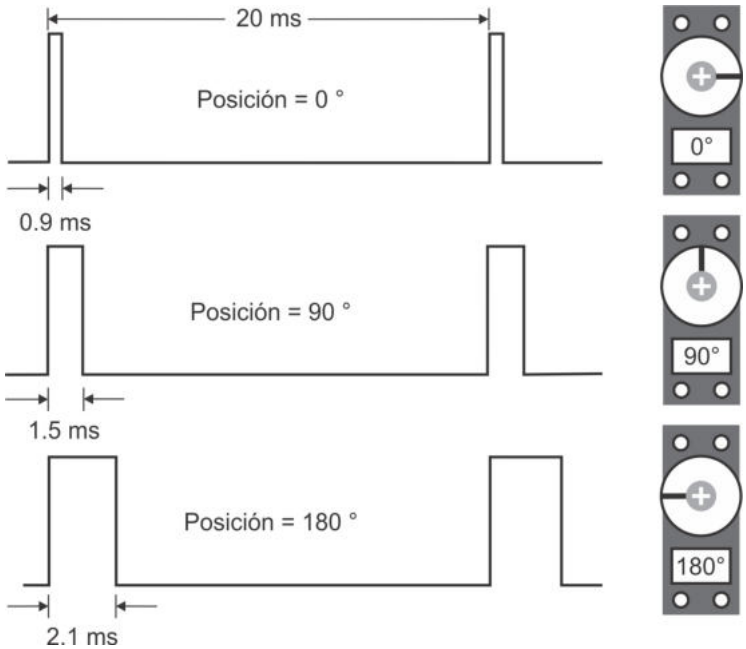


Figura 5.19: Señales para el manejo de un servomotor

Ejemplo 5.4 - Manejo de un Servomotor
Realice un programa que mueva un servomotor de 0° *a* 90°*, de* 90° *a* 180°*, de* 180° *a* 90° *y de* 90° *a* 0°*, con un periodo de 200 ms entre cada cambio. Repitiendo la secuencia de movimientos de manera continua.*

Se asume que el microcontrolador está operando a 1 MHz, por lo tanto, el temporizador se incrementa cada 1 us. Para un periodo de 20 ms, utilizando PWM rápido, el temporizador 1 debe contar de 0 a 19999, rango alcanzable con 16 bits. Si se utiliza el modo 14, el valor máximo se debe ubicar en el registro `ICR1` y con el registro `OCR1A` se determinará el ancho de pulso, el código C con la solución del problema es:

```
#define  F_CPU  1000000UL

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

int  main() {

   DDRB = 0xFF;          // Salida en OC1A (PB1)

   ICR1 = 19999;         // Valor máximo, T = 20 ms
   TCCR1A = 0x82;        // PWM rápido con modo no invertido
   TCCR1B = 0x19;        // Reloj sin preescalador
   TCCR1C = 0x00;

   while(1) {
      OCR1A = 899;       // 0°
      _delay_ms(200);
      OCR1A = 1499;      // 90°
      _delay_ms(200);
      OCR1A = 2099;      // 180°
      _delay_ms(200);
      OCR1A = 1499;      // 90°
      _delay_ms(200);
   }
}
```

El espacio de movimiento hace que para el registro `OCR1A` se tenga un intervalo de 899 a 2099 valores posibles, para un total de 1200 combinaciones. Por lo tanto, la precisión que se puede conseguir en la posición del eje de un servomotor es de $180°/1200 = 0.15°$.

5.8. El Temporizador 2

En la Figura 5.20 se muestra la organización del temporizador 2 y en la Tabla 5.13 se listan los Registros I/O utilizados por el recurso, todos son de 8 bits. Por

su dirección, se observa que solo `TIFR2` es un Registro I/O, todos los demás son Registros I/O Extendidos.

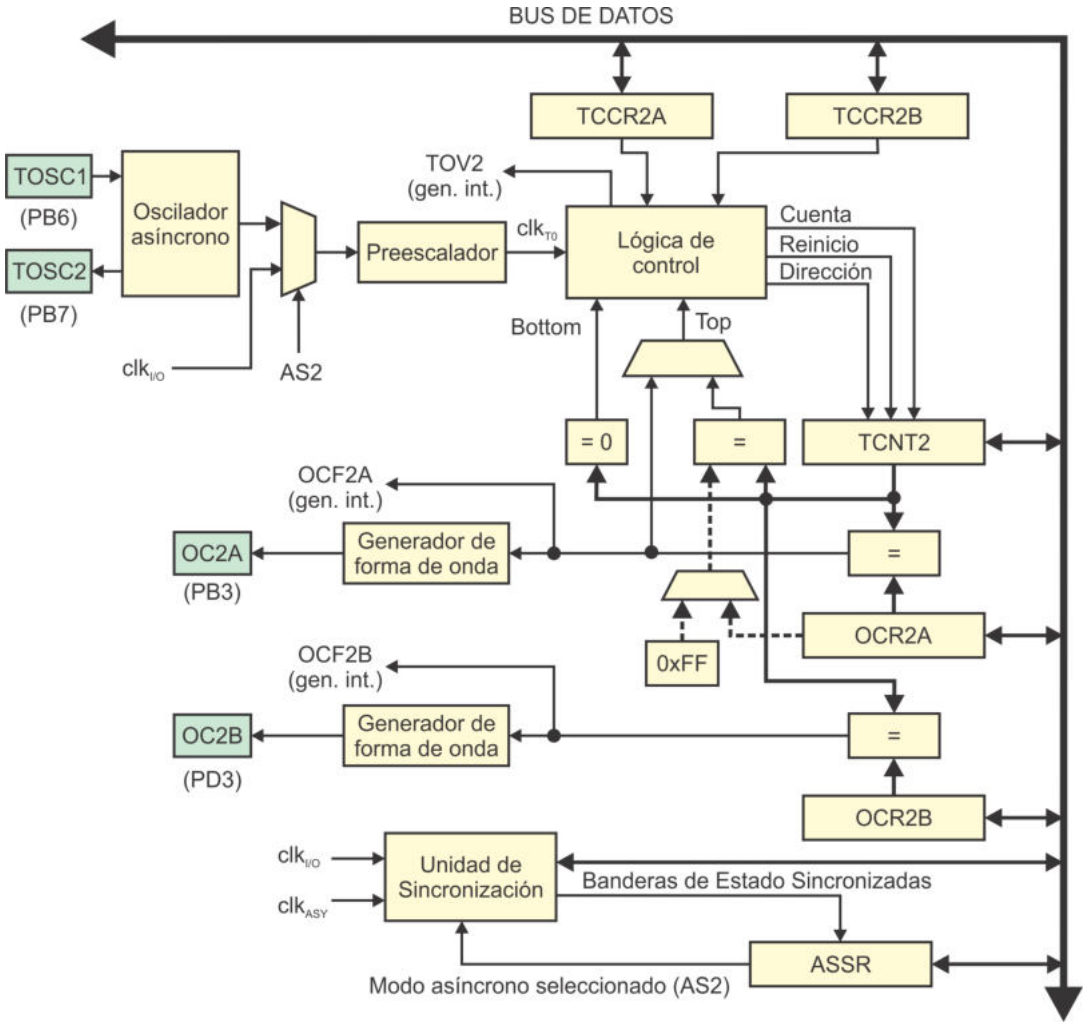


Figura 5.20: Organización del temporizador 2

Tabla 5.13: Registros del temporizador 2

Registro	**Dirección**	**Operación**
TCNT2	(0xB2)	Registro del temporizador 2
OCR2A	(0xB4)	Comparador A del temporizador 2
OCR2B	(0xB3)	Comparador B del temporizador 2
TCCR2A	(0xB0)	Registro A para la configuración y control
TCCR2B	(0xB1)	Registro B para la configuración y control
TIFR2	0x17 (0x37)	Registro de banderas
TIMSK2	(0x70)	Máscara para activar las interrupciones
ASSR	(0xB6)	Registro para la operación asíncrona

5.8.1. Configuración y Control del Temporizador 2

El temporizador 2 es controlado por los registros `TCCR2A` y `TCCR2B` (*Timer/Counter Control Register*), cuyos bits son descritos a continuación:

REG.	7	6	5	4	3	2	1	0
TCCR2A	COM2A1	COM2A0	COM2B1	COM2B0	-	-	WGM21	WGM20
TCCR2B	FOC2A	FOC2B	-	-	WGM22	CS22	CS21	CS20

La operación de los bits se revisa por grupos:

- **Bits WGM2[2:0]:** Definen el modo para la generación de una forma de onda (*Waveform Generation Mode*), en otras palabras, en estos bits se configura el modo de operación del temporizador 2.
- **Bits COM2x[1:0]:** En estos bits se configura una respuesta automática ante coincidencias por comparación (*Compare Output Mode*). La x puede ser A o B, debido a que el recurso tiene dos comparadores.
- **Bits FOC2x:** Forzan u obligan un evento de comparación, si se configuró una respuesta automática, esta también va a realizarse. La x puede ser A o B por los dos comparadores.
- **Bits CS2[2:0]:** Bits de selección de la fuente de reloj (*Clock Select*).

Tabla 5.14: Modos de operación del temporizador 2

Modo	WGM22	WGM21	WGM20	Descripción	MAXVAL
0	0	0	0	Normal	0xFF
1	0	0	1	PWM con fase correcta	0xFF
2	0	1	0	CTC, el temporizador se limpia ante una coincidencia	OCR2A
3	0	1	1	PWM rápido	0xFF
4	1	0	0	Reservado	-
5	1	0	1	PWM con fase correcta	OCR2A
6	1	1	0	Reservado	-
7	1	1	1	PWM rápido	OCR2A

Modos de Operación

En los bits `WGM2[2:0]` se configura el modo de operación del temporizador 2, en la Tabla 5.14 se muestran los diferentes modos, los cuales son:

- **Modo 0:** El temporizador 2 solo genera eventos de desbordamiento.
- **Modos 1 y 5:** Modos de PWM con fase correcta, el temporizador automáticamente se incrementa de cero a su valor máximo y decrementa del valor máximo a cero, al coincidir con los registros de comparación conmutará una salida según la configuración de los bits `COM2x[1:0]`.
- **Modo 2:** Es un modo CTC (*clear timer on compare match*), en donde el temporizador 2 se limpia automáticamente después de una coincidencia con el registro `OCR2A`.

- **Modos 3 y 7:** Modos de PWM rápido, el temporizador automáticamente se incrementa de cero a su valor máximo, al coincidir con los registros de comparación conmutará una salida según la configuración de los bits `COM2x[1:0]`.

Para los modos PWM, si el valor máximo para el temporizador es 0xFF se pueden generar dos señales en las salidas OC2A y OC2B, pero cuando el valor máximo se toma del registro `OCR2A` solo se puede generar una señal en la terminal OC2B.

Respuesta automática

En los bits `COM2x[1:0]` se configura la respuesta automática para las salidas OC2x ante coincidencias por comparación, la x puede ser A o B porque el recurso tiene dos comparadores. La configuración de estos bits es diferente para los modos normal y CTC, con respecto a los modos PWM, como se muestra en la Tabla 5.15.

Tabla 5.15: Configuración de la respuesta automática (x puede ser A o B)

COM2x1	COM2x0	Descripción
Modos Normal y CTC		
0	0	OC2x desconectado
0	1	Conmuta OC2x
1	0	OC2x es puesto en bajo
1	1	OC2x es puesto en alto
Modos PWM		
0	0	OC2x desconectado
0	1	Reservado (sin uso)
1	0	Modo no invertido
1	1	Modo invertido

Selección de Reloj

La selección de la señal de reloj está determinada por los bits `CS2[2:0]`, los cuales son descritos en la Tabla 5.16. Estos bits se conectan directamente a los bits de selección del multiplexor del preescalador (Sección 5.3), en este caso, todos los factores de división son aplicables para el oscilador interno o externo. Después de un reinicio, el temporizador 2 está detenido porque no tiene una señal de reloj que lo active.

5.8.2. Operación Asíncrona del Temporizador 2

El temporizador 2 se puede manejar con un oscilador externo, esta es una operación asíncrona porque sus incrementos se realizarán a una velocidad diferente al resto del sistema. El hardware 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 us. Como el temporizador es de 8

bits, si no utiliza al preescalador genera un desbordamiento cada 256 x 30.517578125 us = 7.8125 ms = 1/128 s.

Tabla 5.16: Bits para la selección del reloj en el temporizador 0

CS22	**CS21**	**CS20**	**Descripción**
0	0	0	Sin fuente de reloj, temporizador 2 detenido
0	0	1	CLK_{T2S} (sin división)
0	1	0	$CLK_{T2S}/8$
0	1	1	$CLK_{T2S}/32$
1	0	0	$CLK_{T2S}/64$
1	0	1	$CLK_{T2S}/128$
1	1	0	$CLK_{T2S}/256$
1	1	1	$CLK_{T2S}/1024$

Con el uso del preescalador se generan desbordamientos en otras fracciones o múltiplos de segundos reales, en la Tabla 5.17 se muestran las frecuencias de operación del temporizador 2 y los periodos de desbordamiento con diferentes factores de preescala.

La operación asíncrona se habilita en el registro `ASSR` (*Asynchronous Status Register*), el cual aparece en la parte inferior de la Figura 5.20 y tiene los siguientes bits:

REG.	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
ASSR	-	EXCLK	AS2	TCN2UB	OCR2AUB	OCR2BUB	TCR2AUB	TCR2BUB

- **Bit 6 - EXCLK:** (*Enable External Clock Input*) Habilita la entrada de reloj externa, es decir, el temporizador 2 se podrá sincronizar con un reloj externo conectado en la terminal TOSC1.
- **Bit 5 - AS2:** (*Asynchronous Timer/Counter*) Habilita la operación asíncrona, si el bit `EXCLK` tiene 0, el temporizador 2 trabajará con un cristal externo de 32.768 kHz, en caso contrario (`EXCLK` = 1), la señal de reloj debe suministrarse en la terminal TOSC1.

Tabla 5.17: Desbordamientos del temporizador 2 con un oscilador de 32.768 kHz

Preescala	**Frecuencia del temporizador 2 (Hz)**	**Periodo de desbordamiento (s)**
1	32 768	1/128
8	4 096	1/16
32	1 024	1/4
64	512	1/2
128	256	1
256	128	2
1024	32	8

- **Bit 4 - TCN2UB:** Bandera para indicar que el registro `TCNT2` está ocupado por actualización (UB, *Update Busy*).
- **Bit 3 - OCR2AUB:** Bandera para indicar que el registro `OCR2A` está ocupado por actualización.
- **Bit 2 - OCR2BUB:** Bandera para indicar que el registro `OCR2B` está ocupado por actualización.
- **Bit 1 - TCR2AUB:** Bandera para indicar que el registro `TCCR2A` está ocupado por actualización.
- **Bit 0 - TCR2BUB:** Bandera para indicar que el registro `TCCR2B` está ocupado por actualización.

Los 5 bits menos significativos son banderas que indican si un registro está ocupado por actualización, son necesarias cuando se habilita al oscilador externo porque el temporizador 2 trabajará a una frecuencia menor que el resto del sistema, por ello, antes de hacer una lectura o escritura en uno de los registros del recurso, se debe verificar que no hay un cambio en proceso. Las banderas se ajustan o limpian de manera automática.

Un cambio en la fuente de reloj para un sistema secuencial puede provocar que los registros internos alteren su contenido, de manera que, es recomendable que primero se ajuste al bit `AS2` y luego se definan los valores correctos para los registros: `TCNT2`, `OCR2A`, `OCR2B`, `TCCR2A` y `TCCR2B`.

5.8.3. Interrupciones debidas al Temporizador 2

El temporizador 2 puede interrumpir a la CPU por 3 eventos: un evento de desbordamiento y dos de coincidencia por comparación, independientemente de que sea manejado por el oscilador interno o un cristal externo. Los eventos quedan registrados en `TIFR2` (*Time/Counter Interrupt Flag Register*), cuyos bits son:

REG.	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
TIFR2	-	-	-	-	-	OCF2B	OCF2A	TOV2

- **Bit 2 - OCF2B:** (*Output Compare Match Flag*) Indica una coincidencia del temporizador 2 con el registro `OCR2B`.
- **Bit 1 - OCF2A:** Indica una coincidencia del temporizador 2 con el registro `OCR2A`.
- **Bit 0 - TOV2:** (*Timer/Counter Overflow*) Indica un desbordamiento del temporizador 2.

Las banderas se limpian automáticamente si se emplean interrupciones o reescribiendo un 1 lógico si se usa sondeo. Para que los eventos provoquen una interrupción a la

CPU, se deben poner en alto sus habilitadores individuales, además del habilitador global, en el registro `TIMSK2` (*Timer/Counter Interrupt Mask Register*) se realizan estas habilitaciones, sus bits son:

REG.	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
TIMSK2	-	-	-	-	-	OCIE2B	OCIE2A	TOIE2

- **Bit 2 - OCIE2B:** (*Output Compare Match Interrupt Enable*) Habilita la interrupción por una coincidencia con el registro `OCR2B`.

- **Bit 1 - OCIE2A:** Habilita la interrupción por una coincidencia con el registro `OCR2A`.

- **Bit 0 - TOIE2:** (*Timer/Counter Interrupt Enable*) Habilita la interrupción por desbordamiento del temporizador 2.

En la Tabla 5.18 se muestran los vectores de interrupción para los eventos del temporizador 2, la dirección se utiliza al programar en ensamblador y el nombre para lenguaje C.

Tabla 5.18: Vectores de interrupción para el temporizador 2

Dirección	**Evento**	**Descripción**
0x00E	TIMER2_COMPA	El temporizador 2 coincide con el registro OCR2A
0x010	TIMER2_COMPB	El temporizador 2 coincide con el registro OCR2B
0x012	TIMER2_OVF	Desbordamiento del temporizador 2

5.8.4. Ejemplos con el Temporizador 2

La diferencia principal entre el temporizador 0 y el 2 es que, el temporizador 2 se puede manejar con un oscilador externo optimizado para tiempo real, en los siguientes ejemplos se utiliza esta característica, combinando su operación con otros recursos.

Ejemplo 5.5 - Generación de un Tono Variable
Modifique el Ejemplo 5.3 para que el tono generado combine las frecuencias de 440 y 880 Hz, cambiando la frecuencia cada medio segundo, mientras se mantiene presionado un botón conectado en PD2. En la Figura 5.21 se muestra la conexión de los elementos de hardware.

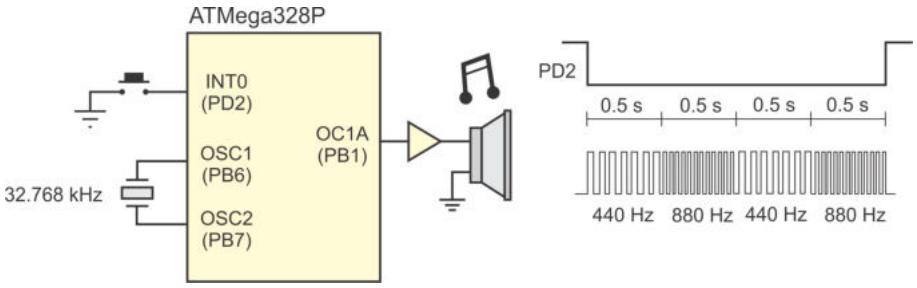


Figura 5.21: Generación de un tono variable

En este ejemplo nuevamente se utilizará la interrupción externa para monitorear al botón y al temporizador 1 para generar el tono, se incorpora el uso del temporizador 2 manejado por un cristal externo de 32.768 kHz para que sincronice los cambios de tono. El temporizador 2 debe configurarse para que desborde cada medio segundo (una vez que esté activo) y en su ISR se asignará el valor adecuado al registro `OCR1A`. El periodo de medio segundo se consigue dividiendo la frecuencia del cristal externo entre 64 (Tabla 5.17).

Para f = 440 Hz se tiene que T = 2 272.72 us, en el Ejemplo 5.3 se utilizó un valor máximo de 1 135 para el registro `OCR1A` del temporizador 1 operando en un modo CTC. Para f = 880 Hz ahora T = 1 136.36 us, dividiendo entre 2 y redondeando, el valor máximo para este tono debe ser de 567.

Con una variable global se indicará el tono a generar, su valor será ajustado en cada interrupción del temporizador 2. El código C para este ejemplo es:

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

uint8_t tono = 0;           // 0 para 440 Hz y 1 para 880 Hz

ISR(INT0_vect) {
    if(!(PIND & 0x04)) {    // Botón presionado
        TCCR1B = 0x09;      // Genera el tono
        TCCR2B = 0x04;      // Inicia el periodo de 0.5 s
    } else {                  // Botón liberado
        PORTB = 0x00;       // Sin carga
        TCCR1B = 0x00;      // Sin reloj para el tono
        TCCR2B = 0x00;      // y para el periodo
        TCNT1 = 0;          // Reinicia los dos temporizadores
        TCNT2 = 0;
    }
}
ISR(TIMER2_OVF_vect) {
    if(tono == 0) {
        TCNT1 = 0;          // Por si tiene un valor mayor
        OCR1A = 567;
        tono = 1;
```

```
    } else {
        OCR1A = 1135;
        tono = 0;
    }
}

int  main() {

    DDRB = 0x02;                // Salida para OC1A
    DDRD = 0x00;                // Entrada para el botón
    PORTD = 0xFF;

// Configuración de la interrupción externa 0
    EICRA = 0x01;               // Cualquier cambio en PD2
    EIMSK = 0x01;

// Configuración del temporizador 1
    OCR1A = 1135;               // Iniciará con 440 Hz
    TCCR1A = 0x40;              // Respuesta automática
    TCCR1B = 0x00;              // Inicia sin reloj
    TCCR1C = 0x00;

// Configuración del temporizador 2
    ASSR = 0x20;                // Cristal externo de 32.768 kHz
    TCCR2A = 0x00;
    TCCR2B = 0x00;              // Inicia sin reloj
    TIMSK2 = 0x01;              // Interrupción por desbordamiento

    sei();                      // Habilitador global de interrupciones
    while(1)
        asm("NOP");
}
```

Se observa que cuando el botón se presiona se habilitan los temporizadores para que se empiece a generar el tono y el conteo de medio segundo, y cuando el botón se libera los temporizadores son detenidos. Además, cuando el temporizador 2 indica que ha transcurrido medio segundo, se cambia valor del registro de comparación del temporizador 1 para modificar la frecuencia del tono.

En el siguiente ejemplo se combina el uso de los 3 temporizadores: el temporizador 0 como contador de eventos, el temporizador 1 para la generación de un tono y el temporizador 2 para determinar un periodo de tiempo real.

Ejemplo 5.6 - Sistema para la Detección de Ganadores
En un supermercado se ha determinado premiar a cada cliente múltiplo de 100, para ello, los clientes van a presionar un botón después de ser atendidos. Desarrolle un sistema basado en un ATMega328P que genere un tono de 400 Hz por 3 segundos cuando detecte al cliente número 100. En la Figura 5.22 se muestra el hardware propuesto.

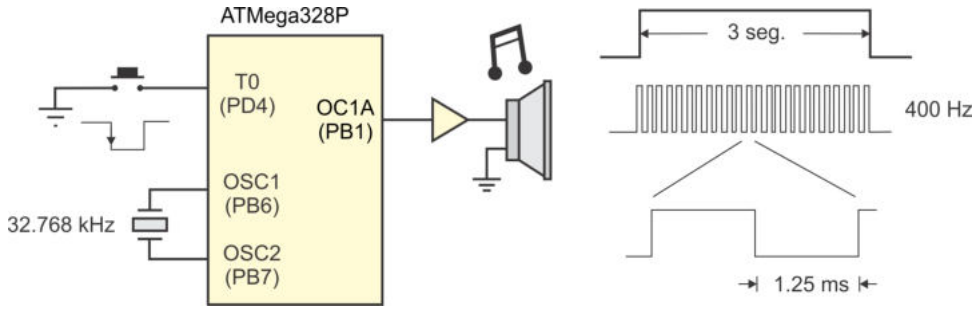


Figura 5.22: Sistema para la detección de ganadores

El temporizador 0 debe ser manejado por eventos externos, se aprecia en la Figura 5.22 que debe incrementarse con flancos de bajada y generar una interrupción en el evento número 100. En el registro `OCR0A` se pondrá el número 99 porque el temporizador inicia en 0. Con un modo CTC el temporizador será reiniciado para repetir el conteo de eventos. En su ISR se habilitarán los otros dos temporizadores para iniciar con la generación del tono y el conteo de segundos.

El temporizador 1 generará el tono con respuesta automática, similar a los Ejemplos 5.3 y 5.5, solo que ahora el valor para el registro de comparación será de 1249, porque el tono debe ser de 400 Hz.

El temporizador 2 se debe configurar para que desborde cada segundo y con el apoyo de una variable global podrá realizar el conteo de segundos. Transcurridos los 3 segundos, en la misma ISR se desactivarán ambos temporizadores para que el sistema deje de generar el tono y de contar segundos.

En el programa se utilizan definiciones para cambiar el número de clientes o el número de segundos de una forma simple, el programa con la solución en lenguaje C es el siguiente:

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

#define clientes 100          // Número de clientes
#define segundos 3            // Número de segundos

uint8_t seg;                  // Contador de segundos

ISR(TIMER0_COMPA_vect){       // Se alcanzó el número de clientes
   TCCR1B = 0x09;             // Habilita la generación del tono
   seg = 0;
   TCCR2B = 0x05;             // Habilita el contador de segundos
}

ISR(TIMER2_OVF_vect) {
   seg++;
```

```
    if(seg == segundos) {      // Pasados 3 segundos
        TCCR1B=0x00;           // Apaga el tono
        TCCR2B=0x00;           // Deja de contar segundos
    }
}

int  main() {

    DDRB = 0x02;               // Salida para OC1A
    DDRD = 0x00;               // Entrada en T0 (PD4)
    PORTD = 0xFF;

// Temporizador 0: contador de eventos en modo CTC
    OCR0A = clientes - 1;      // Número de clientes
    TCCR0A = 0x02;             // Modo CTC
    TCCR0B =0x06;              // Incrementa por flanco de bajada
    TCNT0 = 0;
    TIMSK0 = 0x02;             // Interrupción por comparación

// Temporizador 1: generación del tono
    OCR1A = 1249;              // Tono de 400 Hz
    TCCR1A = 0x40;             // Respuesta automática
    TCCR1B = 0x00;             // Inicia sin reloj
    TCCR1C = 0x00;

// Temporizador 2: contador de segundos
    ASSR = 0x20;               // Cristal externo de 32.768 kHz
    TCCR2A = 0x00;
    TCCR2B = 0x00;             // Inicia sin reloj
    TIMSK2 = 0x01;             // Interrupción por desbordamiento

    sei();                     // Habilitador global de interrupciones

    while(1)
        asm("NOP");
}
```

5.9. Los Temporizadores en Arduino

En el entorno de Arduino hay dos funciones nativas que utilizan los temporizadores:

- **tone**: Es una función que genera un tono o señal cuadrada en la terminal y frecuencia recibidas como argumentos, de manera opcional, la función también puede recibir la duración del tono. Aunque la función utiliza uno de los temporizadores, sus desventajas son que no se pueden generar varias señales en forma simultánea y que al generar una secuencia de tonos, se absorben los recursos del sistema dificultando la ejecución de otras tareas. La función para dejar de generar la señal se llama **noTone(Pin)**.

- **analogWrite**: Genera una señal PWM en la terminal indicada como argumento, la terminal debe corresponder a aquellas donde los temporizadores pueden generar una respuesta automática. La frecuencia de la señal es de 490 Hz si es generada por los temporizadores 0 y 2, y de 980 Hz para el temporizador 1. Además de la terminal, la función recibe un número entre 0 y 255 que corresponde al ciclo de trabajo. Las desventajas de esta función son que no se puede generar una señal a otra frecuencia y tampoco cambiar la resolución en el ciclo de trabajo permitido, mucho menos cambiar la frecuencia de la señal PWM en tiempo de ejecución.

La biblioteca **Servo.h** también es parte del IDE, esta biblioteca define la clase **Servo**, en donde se tienen dos métodos principales: `attach()` para asociar un objeto tipo **Servo** con una de las terminales de los temporizadores y `write()` que recibe un número entre 0 y 180 para posicionar el eje del servomotor.

El temporizador 1 es el más flexible por ser de 16 bits y contar con 15 modos de operación diferentes, para aminorar las limitaciones de las funciones nativas de Arduino, la comunidad ha desarrollado diferentes bibliotecas para ampliar su funcionalidad, una de las más populares es **TimerOne.h**, la biblioteca contiene una colección de rutinas para configurar el temporizador 1, la biblioteca comenzó con la necesidad de establecer el periodo o la frecuencia de una señal PWM, pero creció en características para incluir el manejo de interrupciones por desbordamiento del temporizador.

A pesar del desarrollo de nuevas bibliotecas, hay características del hardware que van quedando sin inclusión, como el recurso de captura de entrada, que es propio del temporizador 1 pero no es considerado en la biblioteca **TimerOne**. Si el desarrollador quiere aprovechar todos los recursos del MCU, lo conveniente es que comprenda la organización del hardware y el manejo de Registros I/O, aunque surjan nuevas bibliotecas, se deberá comprender al hardware para poder aprovecharlas.

Además, el hardware de Arduino impone otras restricciones, por ejemplo, la Arduino Uno es manejada por un cristal externo de 16 MHz, el cual está conectado en las terminales PB6 y PB7, esto limita la operación asíncrona del temporizador 2 porque no hay forma de conectar un cristal externo de 32.768 kHz para su manejo.

En el siguiente *sketch* se muestra la solución para Arduino del Ejemplo 5.6, los principales ajustes son debidos a la frecuencia de operación de la tarjeta, para el tono de 400 Hz se utilizó un valor de 1250 para el registro `OCR1A`, pero como el MCU ahora opera a 16 MHz, este valor se multiplica por 16, quedando un valor de 20000 para el registro de comparación, número que alcanza sin problema en un registro de 16 bits.

El temporizador 2 ahora debe ser manejado con el cristal de 16 MHz, con el factor de preescala de 1024, que es el más alto, su frecuencia de desbordamiento es de 16 MHz/(1024*256) = 61.03 Hz, esto significa que se deben contar 61 desbordamientos para determinar que ha transcurrido un segundo.

El código del *sketch* es:

```
#define clientes 100                // Número de clientes
#define segundos 3                  // Número de segundos

uint8_t banderas;                   // Contador de banderas

ISR(TIMER0_COMPA_vect){             // Se alcanzó el número de clientes
   TCCR1B = 0x09;                   // Habilita la generación del tono
   banderas = 0;
   TCCR2B = 0x07;                   // Habilita el contador de segundos
}

ISR(TIMER2_OVF_vect){
   banderas++;
   if(banderas == segundos*61) {    // Pasados los 3 segundos
      TCCR1B = 0x00;                // Apaga el tono
      TCCR2B = 0x00;                // Deja de contar segundos
  }
}

void setup() {
   pinMode(9, OUTPUT);              // Salida en OC1A
   pinMode(4, INPUT_PULLUP);        // Entrada en T0

// Temporizador 0: contador de eventos en modo CTC
   OCR0A = clientes - 1;            // Número de clientes
   TCCR0A = 0x02;                   // Modo CTC
   TCCR0B =0x06;                    // Incrementa por flanco de bajada
   TCNT0 = 0;
   TIMSK0 = 0x02;                   // Interrupción por comparación

// Temporizador 1: generación del tono
   OCR1A = 19999;                   // Tono de 400 Hz
   TCCR1A = 0x40;                   // Respuesta automática
   TCCR1B = 0x00;                   // Inicia sin reloj
   TCCR1C = 0x00;

// Temporizador 2: contador de segundos
   TCCR2A = 0x00;
   TCCR2B = 0x00;                   // Inicia sin reloj
   TIMSK2 = 0x01;                   // Interrupción por desbordamiento
}

void loop() {

  asm("NOP");
}
```

Aparentemente el uso del temporizador 1 se puede remplazar con las llamadas a las funciones **tone()** y **noTone()**, sin embargo, al hacerlo de esa manera la ejecución

se pierde en el momento en que se da paso a la ISR del temporizador 0, la función **tone()** se llama adecuadamente pero se pierde el flujo de ejecución porque nunca se realiza la llamada a la función **noTone()**.

En conclusión, no es necesario emplear bibliotecas adicionales para el manejo de los temporizadores, lo mejor es comprender su organización y configurarlos directamente a través de sus Registros I/O, estas configuraciones se pueden colocar en un programa en C o dentro de un *sketch* de Arduino.

5.10. Ejercicios

Los siguientes ejercicios involucran el uso de los recursos hasta el momento revisados: interrupciones en los puertos y temporizadores, para todos se debe suponer que el ATMega328P estará operando a 1 MHz, por ser la frecuencia con la que son comercializados, y los ejercicios pueden ser resueltos en lenguaje C.

1. Escriba un programa en C que genere un número aleatorio de 8 bits en el Puerto B cada que se presione un botón conectado en PD2 (INT0). Observe que si se utiliza un temporizador de 8 bits incrementando cada microsegundo, lo "aleatorio" está en el momento en que el usuario presiona el botón.

2. Utilizando los temporizadores 0 y 1 de un ATMega328P, realice un programa que simultáneamente genere 2 señales, una de 10 kHz y otra de 500 Hz, en las terminales donde los temporizadores pueden dar una respuesta automática.

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 5.23 (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).

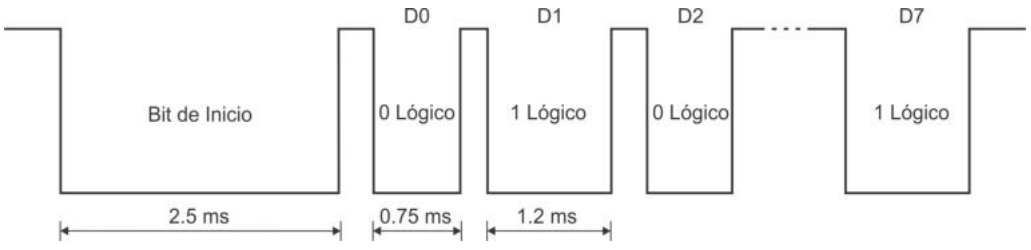
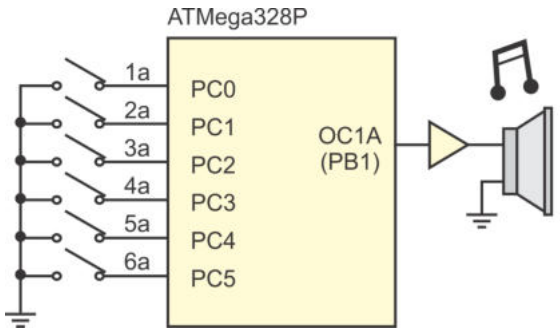


Figura 5.23: Señal modulada, la información está en los niveles bajos de voltaje

4. Realice un sistema que controle la intensidad de un conjunto de LEDs ultrabrillantes, manejando 5 niveles por medio de una señal PWM. 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 el sistema se enciende (0 % del ciclo de trabajo). Si se alcanza la máxima intensidad (100 %) y se presiona nuevamente al botón, los LEDs deben apagarse. Configure para que la señal PWM tenga una frecuencia cercana a 100 Hz, para una operación adecuada de los LEDs.

5. Construya un Afinador de Guitarra, en la Figura 5.24 se muestra el hardware con la frecuencia de los tonos que se deben generar. Cuando se cierre uno de los interruptores, el circuito debe producir el tono que le corresponda, mientras el interruptor permanece cerrado. No se debe generar algún tono cuando se han cerrado dos o más interruptores.



Cuerda	Nota	Frecuencia
1a	MI	1318.5 Hz
2a	SI	987.77 Hz
3a	SOL	783.99 Hz
4a	RE	587.33 Hz
5a	LA	440 Hz
6a	MI	329.63 Hz

Figura 5.24: Afinador de guitarra

6. Con un ATMega328P genere 3 señales PWM con una frecuencia aproximada de 100 Hz, estas servirán para el manejo de un LED RGB, las señales iniciarán con un ciclo de trabajo del 0 % y este se irá incrementando en factores del 10 % con el apoyo de 3 botones. Utilice una interrupción por cambios en las terminales para detectar la actividad en los botones.

7. Apoyados en la Figura 5.25, realice un medidor de frecuencia con salida a 2 visualizadores de 7 segmentos. La frecuencia de una señal se define como el número de ciclos por segundo, por lo que se puede medir utilizando 2 temporizadores, el temporizador 0 para el conteo de ciclos y el temporizador 2 para el periodo de 1 segundo. De esta forma, la frecuencia medida se actualizará cada segundo, por interrupción, mientras que en el lazo infinito continuamente se mostrará la última medición.

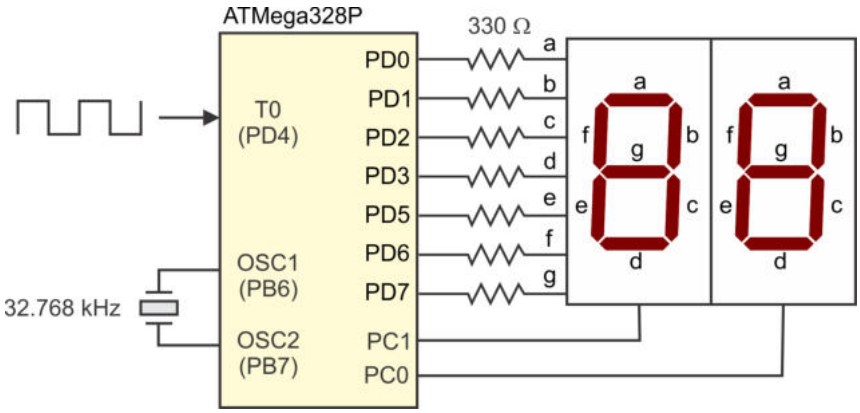


Figura 5.25: Medidor de frecuencia

8. En la Figura 5.26 se muestra un servomotor conectado a un ATMega328P, realice un programa que inicialmente coloque el eje del servomotor en su posición central (90°) y con la ayuda de 2 botones mueva el eje a la izquierda o derecha, según el botón presionado, con un movimiento aproximado de 15° por evento. Asegúrese que los límites de cada extremo no sean rebasados.

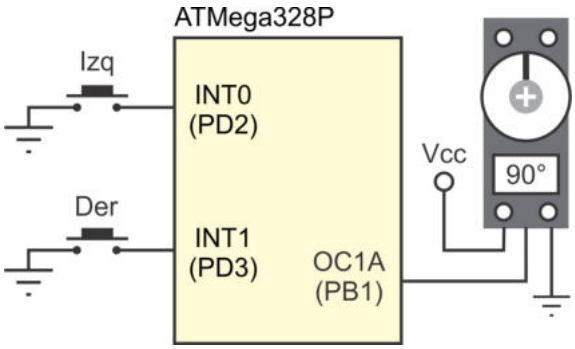


Figura 5.26: Manejo de un servomotor

9. En la Figura 5.27 se muestran las notas de la melodía denominada "La pequeña Marian", identificando cada nota con su nombre y su duración por tiempos musicales. En la Tabla 5.19 se indica la frecuencia para cada nota y la duración en segundos. Utilizando esta información, realice un programa para el ATMega328P mediante el que se reproduzca esta melodía cada que se presione un botón conectado en su terminal PD2, utilice el temporizador 1 para la generción de los tonos y el temporizador 2 para el manejo de los tiempos.

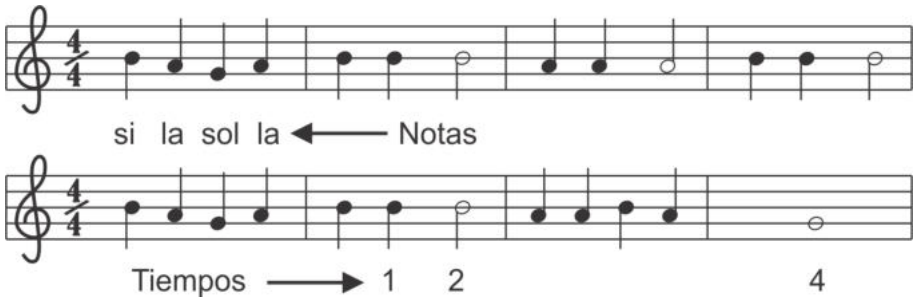


Figura 5.27: Melodía “La pequeña Marian”

Tabla 5.19: Frecuencias y tiempos para la melodía “La pequeña Marian”

Nota	**Frecuencia**	**Tiempo**	**Duración**
si	493.883 Hz	1	0.5 s
la	440 Hz	2	1 s
sol	391.995 Hz	4	2 s

10. Con el temporizador 1 genere una rampa con 10 niveles diferentes y un periodo aproximado de 1 s (0.1 s en cada nivel), en la Figura 5.28 se muestra la idea. La base para mantener un nivel “analógico” debe ser una señal PWM de 1 kHz, se sugiere emplear al registro `OCR1A` como valor máximo en la señal PWM y a `OCR1B` para definir el ancho de pulso, y configurar para que las coincidencias con `OCR1A` produzcan interrupciones, así, en su ISR se podrá llevar un contador de ciclos con el que se determine si es necesario cambiar el ancho de pulso.

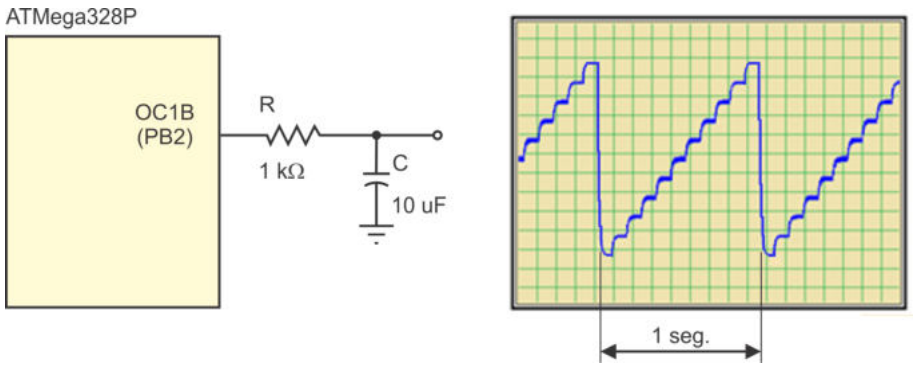


Figura 5.28: Generación de una rampa analógica

Capítulo 6

Manejo de Información Analógica

Aunque los microcontroladores son dispositivos para el procesamiento de información digital, incluyen recursos que les permiten obtener información generada por dispositivos analógicos. El ATMega328P cuenta con un convertidor analógico a digital (ADC, *analog-to-digital converter*) y un comparador analógico (AC, *analog comparator*), estos recursos se describen en el presente capítulo.

6.1. Convertidor Analógico a Digital

Un ADC recibe una muestra de una señal analógica, es decir, el valor de la señal en un instante de tiempo, a partir de la cual genera un número o 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 6.1 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 como una etapa en el proceso de conversión.

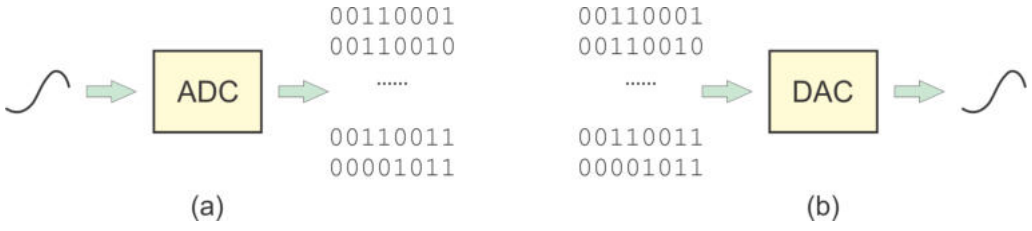


Figura 6.1: Funcionalidad (a) de un ADC y (b) de un DAC

6.1.1. 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 consiste en tomar el valor de la señal analógica en un instante de tiempo, y mantenerlo, mientras el proceso de conversión concluye. Cada muestra se va tomando en 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 salida 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:

$$resolución = \frac{V_{MAX}}{2^8 - 1} = \frac{5V}{255} = 19.607\,mV$$

En la Figura 6.2 se muestran las etapas involucradas en el proceso de conversión analógico a digital, se toman 10 muestras de una señal analógica y se utilizan 3 bits para su cuantificación.

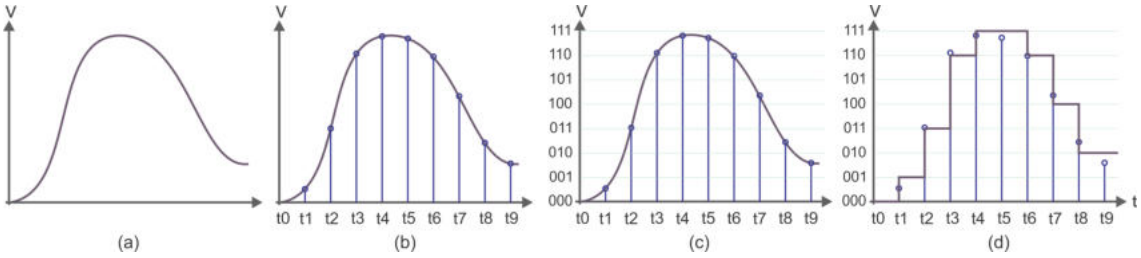


Figura 6.2: Proceso de conversión: (a) una señal analógica, (b) muestreo, (c) cuantificación y (d) señal digital

Hardware para la Conversión Digital a Analógico

El hardware básico para convertir una señal digital en su correspondiente valor analógico involucra una red resistiva R-2R, como se muestra en la Figura 6.3. 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 el valor resultante se convierte a voltaje con la ayuda de un amplificador operacional.

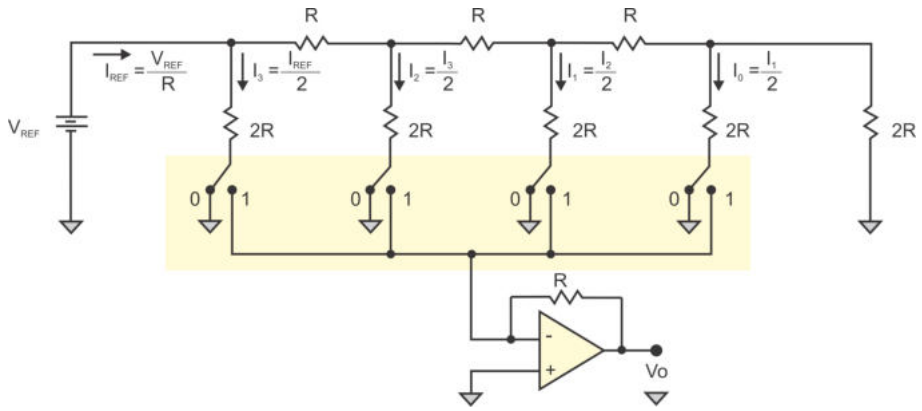


Figura 6.3: Red resistiva R-2R para una conversión digital a analógica

Hardware para la Conversión Analógico a Digital

Hay tres configuraciones clásicas para los convertidores analógico a digital:

- Un ADC del **tipo integrador** es el más simple, económico y lento, su hardware incluye un circuito RC, un contador, un DAC y un comparador, puede convertir señales con frecuencias menores a 1 kHz, por lo que solo es aplicable si la señal analógica tiene variaciones mínimas a lo largo del tiempo.
- Un ADC de **aproximaciones sucesivas** está basado en un comparador, un DAC y un generador de aproximaciones sucesivas, puede trabajar con señales en el rango de frecuencias de 1 kHz a 1 MHz, por lo que una de sus aplicaciones es la digitalización de señales de audio.
- Un ADC **paralelo** o *flash* tiene un hardware complejo, para generar un dato digital de n bits el ADC requiere de 2^n comparadores, lo que eleva mucho su costo, sin embargo, se puede emplear para digitalizar señales con frecuencias mayores a 1 MHz, como las señales de video.

Los ADCs de aproximaciones sucesivas son los que comúnmente están incluidos en los microcontroladores, su estructura general se muestra en la Figura 6.4, considerando únicamente 4 bits de resolución. 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.

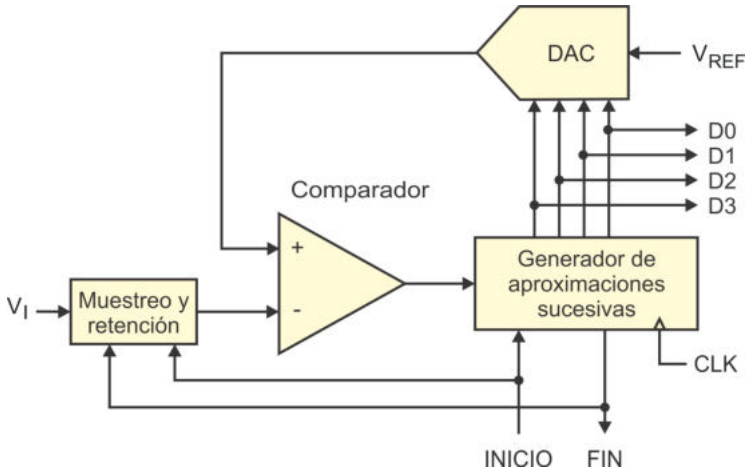


Figura 6.4: Organización de un ADC de aproximaciones sucesivas

El bloque de muestreo y retención (*sample and hold*) tiene por objetivo mantener la muestra analógica durante el proceso de conversión, es necesario porque la información analógica podría cambiar mientras se realiza el proceso y con ello, el resultado sería incongruente.

La conversión da comienzo 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, el generador de aproximaciones sucesivas debe continuar con las siguientes aproximaciones, en la Figura 6.5 se muestra un diagrama de flujo para comprender este proceso.

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.

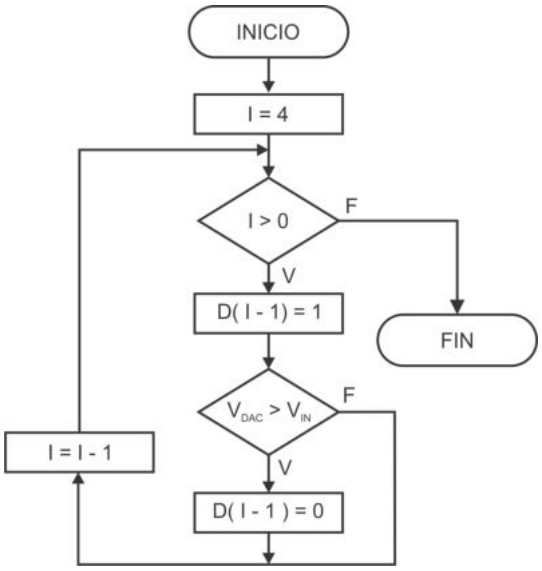


Figura 6.5: Comportamiento del generador de aproximaciones sucesivas

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".

6.1.2. El ADC del ATMega328P

El ATMega328P incluye un ADC de aproximaciones sucesivas de 10 bits, en la Figura 6.6 se puede ver que la entrada analógica proviene de un multiplexor que permite seleccionar 1 de 8 canales externos (ADC0 a ADC7, aunque los dispositivos con encapsulado PDIP solo tienen 6 canales), además de dos valores constantes y un sensor de temperatura de baja resolución. La selección del voltaje a convertir se realiza con los bits `MUX[3:0]`, que son parte del registro `ADMUX`, en la Tabla 6.1 se muestran las opciones para elegir la entrada para el ADC.

Tabla 6.1: Selección de la entrada para el ADC

MUX[3:0]	**Entrada**
0000	ADC0
0001	ADC1
0010	ADC2
...	...
0111	ADC7
1000	Sensor de temperatura
1001 - 1101	Sin uso
1110	1.1 V
1111	0 V (GND)

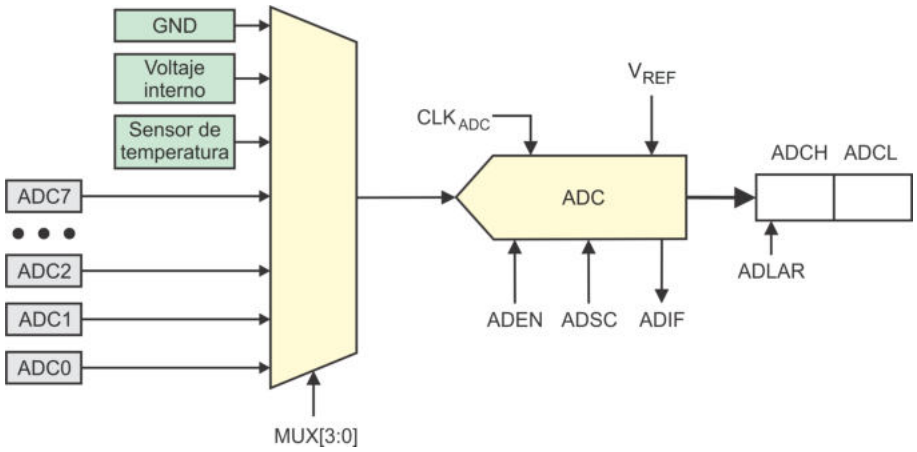


Figura 6.6: El ADC del ATMega328P

El registro `ADCSRA` (*ADC Control and Status Register A*) incluye 3 bits fundamentales para la operación del ADC, el bit `ADEN` para habilitarlo, el bit `ADSC` para iniciar una conversión y la bandera `ADIF` que se pone en alto al concluir una conversión. La bandera se puede sondear por software o se puede configurar al recurso para que produzca una interrupción por fin de conversión. El registro también incluye al bit `ADATE` con el que se da paso al modo de carrera libre o a inicios automáticos de una conversión.

El resultado de la conversión queda disponible en los registros `ADCH` y `ADCL`, aunque de `ADCH` solo se utilizan los bits menos significativos. En lenguaje C es posible leer el resultado completo mediante un registro de 16 bits denominado `ADCW`. El bit `ADLAR` que se aprecia en la Figura 6.6, también es parte del registro `ADMUX` y hace que el resultado se desplace a la izquierda, es decir, los 8 bits más significativos resultantes de la conversión quedan en `ADCH` y los 2 bits faltantes se ubican en la parte alta de `ADCL`. Esta alineación es útil en aplicaciones donde con 8 bits es suficiente para el manejo de la información analógica, al poner en alto al bit `ADLAR`, solo se utilizará al registro `ADCH`.

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 debería determinarse en forma práctica. Los AVR incluyen un preescalador de 7 bits para generar la frecuencia de trabajo del ADC a partir de la frecuencia del microcontrolador, en la Figura 6.7 se muestra la estructura del preescalador, los factores de división se seleccionan con los bits `ADPS[2:0]`, que también son parte del registro `ADCSRA`, y en la Tabla 6.2 se pueden ver las opciones disponibles para dividir la frecuencia del reloj principal.

Con la primera conversión se inicializa la circuitería analógica, por lo que requiere de 25 ciclos de reloj, para las conversiones siguientes solo se emplean 13 ciclos, excepto

cuando la conversión inicia con un disparo automático, hay una sincronización previa por lo que son necesarios 13.5 ciclos de reloj.

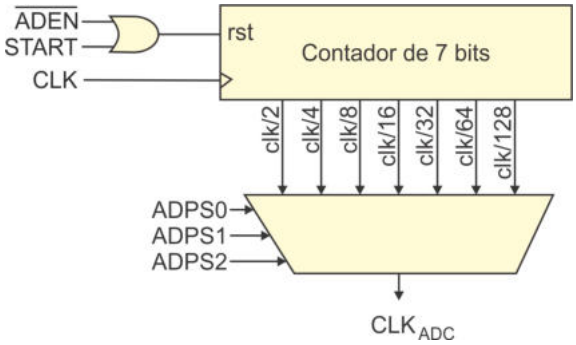


Figura 6.7: Preescalador del ADC

Tabla 6.2: Selección del factor de división en el preescalador 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

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. Solo debe considerarse que AVcc no debe diferir más de $\pm 0.3\,V$ de Vcc. En la mayoría de aplicaciones es suficiente si se conecta a AVcc directamente con Vcc, sin embargo, cuando la entrada analógica tiene un valor máximo pequeño, en relación al voltaje de alimentación, el fabricante recomienda el uso de un filtro LC pasa bajas, como el mostrado en la Figura 6.8, con la finalidad de proporcionar inmunidad al ruido.

El ADC utiliza un DAC durante el proceso de conversión, por ser un ADC de aproximaciones sucesivas como el descrito en la Figura 6.4. El DAC requiere un voltaje de referencia (V_{REF}), que determina el valor máximo para la entrada analógica. En la Figura 6.9 se observa que V_{REF} puede ser proporcionado por diferentes fuentes, la selección se realiza con los bits `REFS[1:0]` que son parte del registro `ADMUX`. Si el valor de la señal analógica de entrada excede a V_{REF}, es codificado como 0x3FF.

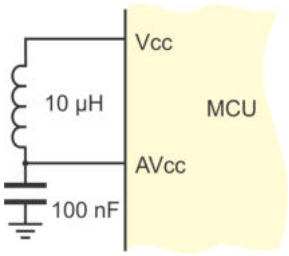


Figura 6.8: Filtro pasa bajas sugerido para conectar AVcc con Vcc

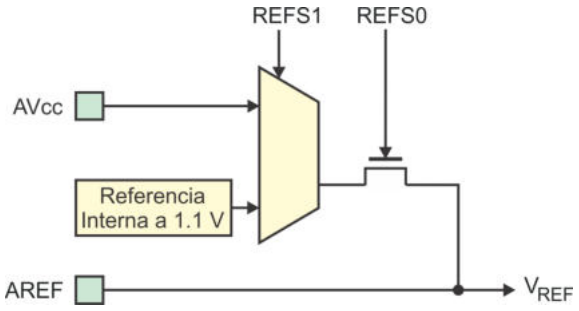


Figura 6.9: Hardware para el voltaje de referencia

El voltaje de referencia puede tomarse de la terminal AREF, de la alimentación analógica (AVcc) o de un voltaje interno, en la Tabla 6.3 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.

Una opción muy simple, desde el punto de vista práctico, consiste en la conexión de las terminales AREF y AVcc 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.

El valor del registro `ADCW` resultante de la conversión, en términos de V_{REF}, queda expresado de la siguiente manera (V_{IN} es el voltaje a digitalizar):

$$ADCW = \frac{V_{IN} * 1023}{V_{REF}}$$

Tabla 6.3: 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 1.1 V (valor típico)

Tabla 6.4: Registros para el manejo del ADC

Registro	Dirección	Operación
ADCL	(0x78)	Parte baja del resultado de la conversión
ADCH	(0x79)	Parte alta del resultado de la conversión
ADCSRA	(0x7A)	Registro A para la configuración y control
ADCSRB	(0x7B)	Registro B para la configuración y control
ADMUX	(0x7C)	Selector para el multiplexor del ADC
DIDR0	(0x7E)	Registro 0 para deshabilitar un *buffer* digital

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 trabajar 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 solo a un canal e ignorando el tiempo requerido por la primera conversión, las siguientes requieren de 13 ciclos de reloj, lo que conlleva 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 analógica de entrada es de 4.8 kHz.

6.1.3. Registros para el Manejo del ADC

En la Tabla 6.4 se listan todos los registros involucrados en el manejo del ADC de un ATMega328P, por su dirección, se observa que todos son Registros I/O Extendidos.

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 se puede hacer referencia a ambos registros tratándolos como `ADCW` y su valor se puede asignar directamente a una variable de 16 bits.

Los 10 bits de información están alineados a la derecha, de la siguiente manera:

REG.	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
ADCH	-	-	-	-	-	-	ADC9	ADC8
ADCL	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADC1	ADC0

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:

REG.	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
ADCH	ADC9	ADC8	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2
ADCL	ADC1	ADC0	-	-	-	-	-	-

Esta organización es útil en aplicaciones donde resulte suficiente una resolución de 8 bits. En esos casos, solo se emplearía al registro `ADCH`, ignorando el contenido del registro `ADCL`.

El control y estado del ADC se maneja con los registros `ADCSRA` y `ADCSRB` (*ADC Control and Status Register*). Siendo el registro `ADCSRA` el principal para la operación del ADC, sus bits son:

REG.	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
ADCSRA	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0

- **Bit 7 - ADEN:** (*ADC Enable*) Es el habilitador del ADC, debe ser puesto en alto para utilizar al ADC.
- **Bit 6 - ADSC:** (*ADC Start Conversion*) La conversión inicia al escribir un 1 en este bit, el cual se limpia automáticamente cuado concluye. La primera conversión requiere de 25 ciclos de reloj, las siguientes solo de 13.
- **Bit 5 - ADATE:** (*ADC Auto Trigger Enable*) Habilita el modo de carrera libre o un auto disparo, es decir, el ADC inicia una conversión cuando ocurre un evento en otro recurso. Cuando este bit es puesto en alto, el hardware evalúa la configuración del registro `ADCSRB`, en donde se debe configurar la forma de operación del ADC. Con este bit en bajo, el contenido del registro `ADCSRB` es ignorado.
- **Bit 4 - ADIF:** (*ADC Interrupt Flag*) Es la bandera de fin de conversión, puede generar una interrupción o sondearse vía software. La bandera se limpia automáticamente por hardware si se configura la interrupción. Al emplear sondeo, se debe reescribir un 1 lógico para limpiar la bandera.
- **Bit 3 - ADIE:** (*ADC Interrupt Enable*) Habilita la interrupción por el fin de una conversión analógica a digital, el habilitador global de interrupciones debe estar activo para que se produzca la interrupción. En la Tabla 6.5 se muestra el vector de la interrupción del ADC.
- **Bits [2:0] - ADPS[2:0]:** (*ADC Prescaler Select Bits*) Bits para seleccionar el factor de división del preescalador del ADC. El preescalador del ADC se mostró en la Figura 6.7 y sus factores de división se describieron en la Tabla 6.2.

Tabla 6.5: Vector de interrupción para el ADC

Dirección	**Evento**	**Descripción**
0x02A	ADC	Fin de conversión analógico a digital

En el registro `ADCSRB` se configura el modo de carrera libre o un auto disparo, sus bits son:

REG.	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
ADCSRB	-	ACME	-	-	-	ADTS2	ADTS1	ADTS0

- **Bit 6 - ACME:** Este bit hace que el multiplexor sea utilizado por el comparador analógico y no por el ADC, debe estar en bajo para una operación adecuada del ADC.
- **Bits [2:0] - ADTS[2:0]:** (*ADC Auto Trigger Source*) Si el bit `ADATE` del registro `ADCSRA` tiene un nivel lógico alto, con estos bits se selecciona el modo de carrera libre o la fuente de autodisparo, en la Tabla 6.6 se muestran las diferentes opciones.

Tabla 6.6: Selección de la fuente de autodisparo para el ADC

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 del temporizador 0 con su comparador A
1	0	0	Desbordamiento del temporizador 0
1	0	1	Coincidencia del temporizador 1 con su comparador B
1	1	0	Desbordamiento del temporizador 1
1	1	1	Captura de entrada debida al temporizador 1

El registro `ADMUX` es el selector para el multiplexor del ADC, aunque también permite seleccionar otros parámetros, sus bits son:

REG.	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
ADMUX	REFS1	REFS0	ADLAR	-	MUX3	MUX2	MUX1	MUX0

- **Bits [7:6] - REFS[1:0]:** (*Reference Selection Bits*) Bits para seleccionar el voltaje de referencia del ADC, en la Tabla 6.3 se muestran las diferente alternativas.
- **Bit 5 - ADLAR:** (*ADC Left Adjust Result*) 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 [3:0] - MUX[3:0]:** Bits para seleccionar el canal de entrada analógico, se puede elegir entre 8 canales externos, dos valores constantes y un sensor de temperatura, en la Tabla 6.1 se muestran las diferentes opciones.

En aplicaciones que utilicen más de un canal analógico, el valor del registro `ADMUX` debe establecerse antes de iniciar una conversión.

El registro `DIDR0` (*Digital Input Disable Register*) sirve para desactivar un *buffer* digital en cada una de las entradas que serán usadas para señales analógicas, sus bits son:

REG.	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
DIDR0	-	-	ADC5D	ADC4D	ADC3D	ADC2D	ADC1D	ADC0D

- **Bits [5:0] - ADC[5:0]D:** (*Digital Input Disable*) Las terminales digitales incluyen un *buffer* de entrada, un 1 en cada uno de estos bits deshabilita a su *buffer* digital y con ello se reduce el consumo de potencia.

Una vez que se ha deshabilitado el *buffer* digital, si se realiza una lectura al registro `PINC` se obtendrán 0's. En los dispositivos que cuentan con 8 canales analógicos externos, las entradas ADC7 y ADC6 están dedicadas a señales analógicas, por lo que no tienen un *buffer* digital.

6.1.4. El Sensor de Temperatura

El ATMega328P tiene un sensor interno referido a tierra, acoplado al canal 8 del ADC, para su lectura, en los bits `MUX[3:0]` del registro `ADMUX` debe escribirse el valor "1000" y con los bits `REFS[1:0]` se debe seleccionar el voltaje interno de 1.1 V como referencia para el convertidor.

La sensibilidad del sensor es de $1\,mV/°C$ y la temperatura medida tiene una aproximación de $\pm 10°C$, la segunda característica hace que sea muy difícil utilizarlo en aplicaciones que requieran precisión en la medición del parámetro, además de que mide la temperatura interna del microcontrolador y no la temperatura ambiente, sin embargo, el sensor puede emplearse para evaluar la integridad del dispositivo o del sistema que lo contiene.

El voltaje medido tiene una relación casi lineal con la temperatura, algunos valores típicos proporcionados por el fabricante son:

Temperatura $[°C]$	$-45°C$	$+25°C$	$+85°C$
Voltaje [mV]	242 mV	314 mV	380 mV

Haciendo una aproximación lineal a los 3 puntos, se obtiene la siguiente expresión para la temperatura, en función del voltaje interno (en mV):

$$T = 0.94249[\tfrac{°C}{mV}]V_{int} - 272.39[°C]$$

Ahora, si se considera que el voltaje de referencia es de 1.1 V, se puede obtener una expresión para la temperatura en función del valor proporcionado por el ADC, quedando de la siguiente manera:

$$T = (1.01343 * ADCW)[°C] - 272.39[°C]$$

Sin embargo, esta expresión se basa en valores típicos y no se puede aplicar de manera general porque, como resultado de los procesos de fabricación, la salida de voltaje para una misma temperatura puede variar entre dispositivos.

Por ello, el fabricante proporciona una expresión matemática en donde compensa por software las diferencias, la expresión es:

$$T = \tfrac{ADCW - TOS}{k}[°C]$$

Las constantes TOS y k se obtienen en las pruebas de fabricación y deberían colocarse en la EEPROM de cada dispositivo.

Lectura del Sensor con Arduino

Uno de los aspectos más importantes de Arduino es que facilita la evaluación rápida de recursos, sin embargo, el sensor de temperatura no está considerado en las bibliotecas porque Arduino inició con microcontroladores que no lo incluyen. Por ello, si se desea leer la temperatura con las ecuaciones anteriores, deben utilizarse los registros del ADC.

En el siguiente *sketch* se lee el voltaje del sensor cada segundo, el programa calcula la temperatura y la envía por el puerto serie para que se pueda ver en el monitor serial del entorno Arduino.

Para la temporización y comunicación serial se utilizan las funciones de Arduino, es necesario incluir la biblioteca `TimerOne` para emplear la interrupción del temporizador. La lectura del sensor se realiza por medio de los Registros I/O, utilizando la interrupción por fin de conversión.

```
#include <TimerOne.h>                  // Biblioteca del temporizador 1

ISR(ADC_vect) {                        // ISR por fin de conversión
float  temp;

   temp = 1.01343*ADCW - 272.39; // Obtiene la temperatura
   Serial.println(temp);         // Manda la temperatura
}

void timerIsr() {                      // Transcurrido un segundo
   ADCSRA |= 1 << ADSC;                // Inicia una conversión
}

void setup() {

   Serial.begin(9600);                 // Inicializa la comunicación serial
   ADMUX = 0xC8;                       // Referencia interna y canal 8
   ADCSRA = 0x8F;                      // Habilita al ADC por interrupción y
                                       // ajusta su frecuencia a 125 KHz
   Timer1.initialize(1000000);            // Interrumpirá cada segundo
   Timer1.attachInterrupt( timerIsr ); // ISR del temporizador 1
}

void loop() {
   asm("NOP");
}
```

6.1.5. Ejemplos con el Convertidor Analógico a Digital

En esta sección se muestran 3 ejemplos con el ADC del ATMega328P, en el primero se presentan dos soluciones para observar la diferencia entre esperar el fin de conversión por sondeo y por interrupción. En el segundo ejemplo se combina el uso del ADC con el temporizador 1, para generar una señal PWM con un ciclo de trabajo controlado desde un potenciómetro. En el tercero se muestra cómo manejar al ADC con una resolución de 8 bits y la sincronización de las conversiones con el apoyo del temporizador 1. Los programas con las soluciones solo se han desarrollado en lenguaje C.

Ejemplo 6.1 - Control Simple de Temperatura
Desarrolle un sistema con base en un ATMega328P, 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. Se asume que el ventilador y el calefactor están acondicionados para activarse con una salida del MCU, en la Figura 6.10 se muestra el hardware propuesto, se observa que para detectar la temperatura se empleará el sensor LM35.*

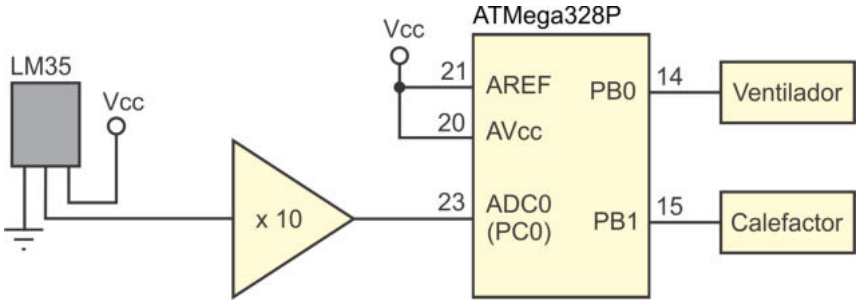


Figura 6.10: Control simple de temperatura

El sensor de temperatura LM35 entrega $10\,mV/°C$. Puesto que el sistema trabajará con la temperatura ambiente, es conveniente multiplicar el voltaje del sensor por un factor de 10 para obtener niveles más manejables, si el ATMega328P se alimenta con 5 V y este valor se utiliza como referencia para el ADC, para temperaturas entre 0 y 50°C, el sensor entregará voltajes entre 0 y 5 V. A las temperaturas de interés, 18 y 23°C, les corresponden los niveles de voltaje de 1.8 y 2.3 V, respectivamente.

Para obtener los valores digitales de las temperaturas de interés se considera la resolución del ADC:

$$resolución = \frac{5V}{2^{10}-1} = \frac{5V}{1023} = 4.8875\,mV$$

Con esta información, se tiene la relación:

Temperatura	**Voltaje Analógico**	**Valor digital**
$18°C$	$1.8V$	$1.8V/4.8875\,mV = 368.23 \approx 368$
$23°C$	$2.3V$	$2.3V/4.8875\,mV = 470.58 \approx 471$

Otro aspecto a considerar es la frecuencia de operación del ATMega328P, si trabaja a 1 MHz, la frecuencia se debe dividir entre 8 para que el ADC trabaje en el rango adecuado.

En el programa con sondeo se espera el fin de conversión para comparar la temperatura actual con los valores de referencia, el código es:

```
#define   inf   368   // Valor digital para 18 grados
#define   sup   471   // Valor digital para 23 grados

#include <avr/io.h>

int main(void) {
uint16_t   temp;

   DDRC = 0x00;              // Entrada para el sensor
   DDRB = 0xFF;              // Salida para los actuadores
   PORTB = 0x00;             // Inicia con las salidas sin activar

   ADMUX = 0x00;             // Selecciona ADC0 y Vref en AREF
   ADCSRA = 0xC3;            // Habilita ADC, inicia conversión
                             // y divide entre 8
   DIDR0 = 0x01;             // Anula el buffer digital en ADC0

   while(1) {
      while( !(ADCSRA & 1 << ADIF ) );    // Espera fin de conversión
      ADCSRA |= 1 << ADIF;                // Limpia la bandera ADIF
      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
   }
}
```

Con interrupciones no es necesario esperar a que la bandera se ponga en alto y su limpieza es automática, y si se activa el modo de carrera libre, tampoco será necesario reiniciar las conversiones, el código con estas características es el siguiente:

```
#define   inf   368         // Valor digital para 18 grados
#define   sup   471         // Valor digital para 23 grados

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

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

ISR(ADC_vect) {
uint16_t    temp;

    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
}

int main(void) {

    DDRC = 0x00;                // Entrada para el sensor
    DDRB = 0xFF;                // Salida para los actuadores
    PORTB = 0x00;               // Inicia con las salidas sin activar

    ADMUX = 0x00;               // Selecciona ADC0 y Vref en AREF
    ADCSRA = 0xEB;              // Habilita ADC con interrupción, inicia
                                // conversión y divide entre 8
    ADCSRB = 0x00;              // El ADC tendrá carrera libre
    DIDR0 = 0x01;               // Anula el buffer digital en ADC0

    sei();                      // Habilitador global de interrupciones

    while(1)                    // Ocioso en el lazo infinito
       asm("NOP");
}
```

Ejemplo 6.2 - Generación de una Señal PWM
Empleando el temporizador 1 de un ATMega328P, genere una señal PWM en modo no invertido en la salida OC1A (PB1), con una frecuencia de 100 Hz (periodo de 10 ms) y un ciclo de trabajo determinado por un voltaje analógico recibido en el canal ADC0 (PC0). En la Figura 6.11 se muestran los elementos de hardware y la relación entre el valor leído en el ADC con el ciclo de trabajo de la señal de salida.

En la Figura 6.11 se observa que hay una relación lineal entre la entrada y la salida, de manera que el valor para el registro `OCR1A` se obtiene de la expresión:

$$OCR1A = \frac{9999 * ADCW}{1023} = 9.774 * ADCW$$

En el código se utiliza una variable auxiliar de punto flotante para hacer el producto. Adicionalmente, el temporizador 1 se configura en el modo 14, que es un modo PWM rápido en donde el valor máximo (9999) se escribe en el registro `ICR1`. El ADC se maneja por interrupción, en cada fin de conversión se calcula el valor para el registro `OCR1A` y se inicia la nueva conversión.

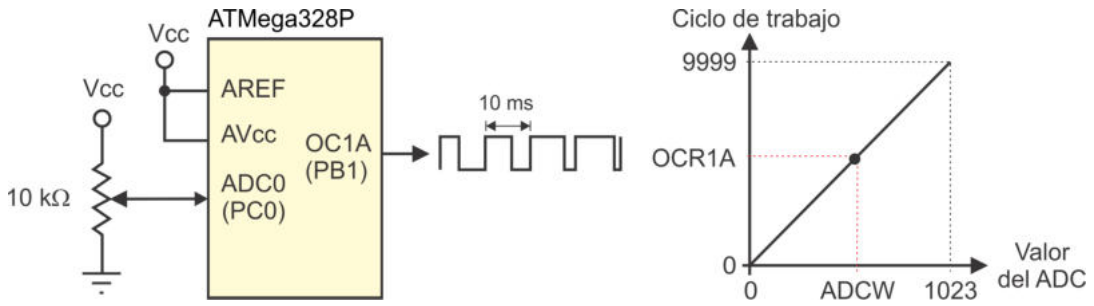


Figura 6.11: Generación de una señal PWM a partir de una entrada analógica

No se utiliza el modo de carrera libre porque la ISR del ADC involucra una operación en punto flotante y se ignora la cantidad de ciclos de reloj que va a requerir.

El código con la solución es el siguiente:

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

ISR(ADC_vect) {
float  aux;

    aux = 9.774*ADCW;          // Calcula el ciclo de trabajo
    OCR1A = aux;
    ADCSRA |= 1 << ADSC;       // Inicia una nueva conversión
}

int  main() {

    DDRB = 0xFF;               // Salida en OC1A (PB1)

    ADMUX = 0x00;              // Selecciona ADC0 y Vref en AREF
    ADCSRA = 0xCB;             // Habilita ADC con interrupción, inicia
                               // conversión y divide entre 8
    DIDR0 = 0x01;              // Anula el buffer digital en ADC0

    ICR1 = 9999;               // Valor máximo para 10 ms
    TCCR1A = 0x82;             // PWM rápido, modo no invertido
    TCCR1B = 0x19;             // Reloj sin preescala

    sei();                     // Habilitador global de interrupciones

    while(1)
        asm("NOP");
}
```

Ejemplo 6.3 - Digitalización de un parámetro analógico
Apoyados en el hardware de la Figura 6.12, realice un programa que digitalice un parámetro analógico leído en el canal ADC0 (PC0) y muestre su valor en 2 visualiza-

dores de 7 segmentos. Sincronizando con el temporizador 1, actualice la información de la salida cada 300 ms.

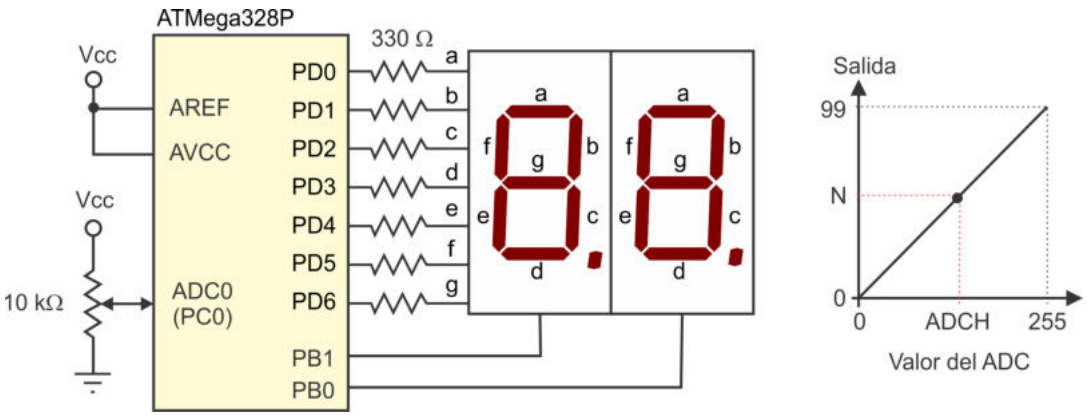


Figura 6.12: Digitalización de un parámetro analógico

Puesto que la salida será un número entre 0 y 99, es suficiente con una resolución de 8 bits para el ADC, de manera que en su configuración se utiliza una alineación a la izquierda de la salida y el resultado de la conversión queda en el registro `ADCH`. En la Figura 6.12 también se puede ver la relación entre el valor obtenido del ADC con el dato de salida (N). El valor para la variable N se obtiene de la expresión:

$$N = \frac{99 * ADCH}{255}$$

El temporizador 1 es utilizado para actualizar la información, por lo que se configura para que interrumpa cada 300 ms, considerando que el MCU va a operar a 1 MHz, se requiere de 300 000 ciclos para este periodo, con un preescalador de 8 se consigue un valor de 37 500, que sin problema alcanza en el registro de comparación (`OCR1A`).

En la ISR del temporizador 1 se habilita el inicio de una conversión y en la ISR de fin de conversión se actualiza el valor de N, de esta manera, el lazo infinito está dedicado a mostrar la información en los visualizadores de 7 segmentos. El código con la solución a este ejemplo es el siguiente:

```
#define  F_CPU    1000000UL

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

uint8_t  N = 0;                  // Define el ciclo de trabajo
const uint8_t codigos[] PROGMEM = { 0x3F, 0x06, 0x5B, 0x4F, 0x66,
                                    0x6D, 0x7D, 0x07, 0x7F, 0x67 };

void mostrar();                  // Manda el valor de N a los displays
```

```
ISR(TIMER1_COMPA_vect) {    // Cada 300 ms
   ADCSRA |= 1 << ADSC;     // Inicia una conversión
}

ISR(ADC_vect) {             // Al terminar la conversión
uint16_t  aux;

   aux = (99*ADCH)/255;
   N = aux;                 // Obtiene el valor de N
}

int  main() {

   DDRD = 0xFF;             // Salida para los visualizadores
   DDRB = 0xFF;             // y para sus habilitaciones
   DDRC = 0x00;             // Entrada para el ADC
   PORTB = 0x03;            // Visualizadores deshabilitados

   // Configuración del ADC
   ADMUX = 0x20;            // Alineación a la izquierda
   ADCSRA = 0xCB;           // Habilita al ADC con interrupciones
                            // e inicia conversión
   DIDR0 = 0x01;            // Anula al buffer digital

   // Configuración del temporizador 1
   OCR1A = 37499;           // Valor para la comparación
   TCCR1A = 0x00;           // Modo 4, CTC con máximo en OCR1A
   TCCR1B = 0x0A;           // El preescalador divide entre 8
   TIMSK1 = 0x02;           // Habilita la interrupción por comparación
   sei();                   // Habilitador global de interrupciones

   while(1)                 // El lazo infinito está dedicado a mostrar
      mostrar();
}

void  mostrar() {
uint8_t  uni, dec;

   uni = N % 10;
   dec = N / 10;

   PORTD = pgm_read_byte(&codigos[uni]);  // Manda unidades
   PORTB = 0x02;                          // Habilita unidades
   _delay_ms(5);
   PORTB = 0x03;                          // Deshabilita

   PORTD = pgm_read_byte(&codigos[dec]);  // Manda decenas
   PORTB = 0x01;                          // Habilita decenas
   _delay_ms(5);
   PORTB = 0x03;                          // Deshabilita
}
```

Se observa en la función `mostrar` del código anterior que los datos se multiplexan para dar la apariencia de que los dos visualizadores están encendidos al mismo tiempo. Además, se utiliza una tabla de constantes en memoria Flash con los códigos de 7 segmentos, para reducir el espacio utilizado en SRAM.

6.2. 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 es mayor o menor que alguna referencia. En la práctica y a pesar de su simpleza, este recurso es poco usado porque las aplicaciones que lo podrían utilizar generalmente son implementadas con el ADC.

6.2.1. Organización del Comparador Analógico

En la Figura 6.13 se muestra la organización del comparador, las entradas analógicas son tomadas de las terminales AIN0 y AIN1 (PD6 y PD7). 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` (*Analog Comparator Output*).

Por medio de interruptores analógicos es posible remplazar la entrada AIN0 por una referencia interna cuyo valor típico es de 1.1 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 deshabilitado (`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 la va a generar se hace con los bits `ACIS[1:0]`, estos 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*).

6.2.2. Registros para el Manejo del AC

El registro `ACSR` es el más importante para el uso del comparador, es un Registro I/O con dirección 0x30 (o 0x50 como SRAM) y sus bits son:

REG.	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
ACSR	ACD	ACBG	ACO	ACI	ACIE	ACIC	ACIS1	ACIS0

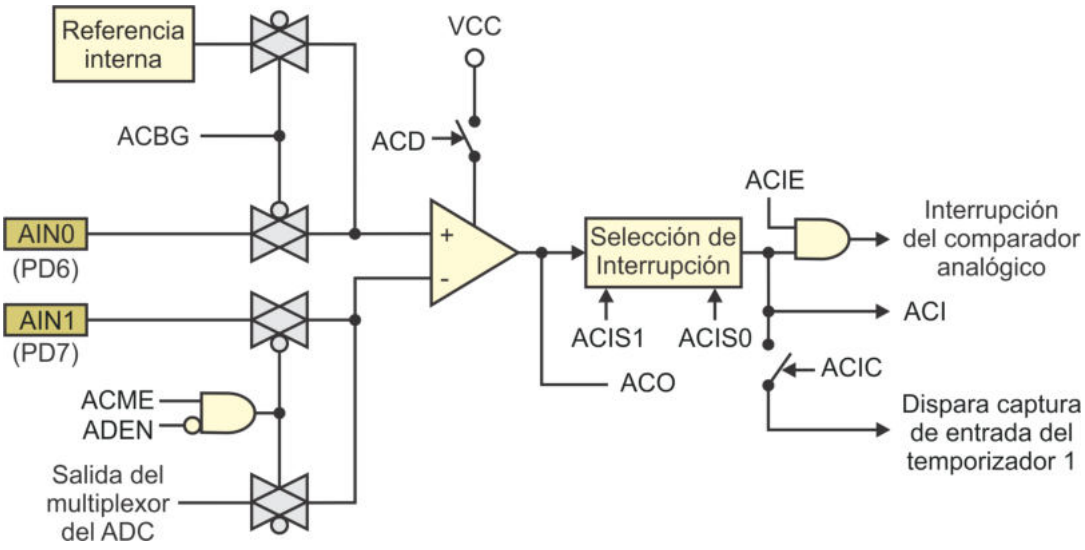


Figura 6.13: Organización del comparador analógico

- **Bit 7 - ACD:** (*Analog Comparator Disable*) Deshabilita al AC 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 deshabilitar su interrupción porque podría generar un evento.
- **Bit 6 - ACBG:** (*Analog Comparator Bandgap Select*) Al ponerlo en alto, la entrada en la terminal positiva proviene de un voltaje de referencia interno y no de la entrada externa AIN0.
- **Bit 5 - ACO:** (*Analog Comparator Output*) La salida del comparador analógico se sincroniza con este bit para su lectura, la sincronización toma 1 o 2 ciclos de reloj.
- **Bit 4 - ACI:** (*Analog Comparator Interrupt Flag*) Es una bandera que indica que ocurrió un evento en el AC, el evento que detecta depende de los bits `ACIS[1:0]` y va a generar una interrupción si el bit `ACIE` fue puesto en alto.
- **Bit 3 - ACIE:** (*Analog Comparator Interrupt Enable*) Es el habilitador de la interrupción por un evento del comparador analógico, el evento se define en los bits `ACIS[1:0]`. En la Tabla 6.7 se muestra el vector de interrupción del AC.
- **Bit 2 - ACIC:** (*Analog Comparator Input Capture Enable*) Con la puesta en alto de este bit se conecta la salida del comparador con el hardware del temporizador 1, para realizar la función de captura de entrada con el evento definido en los bits `ACIS[1:0]`. Se debe habilitar la interrupción por captura de entrada para atender al evento, poniendo en alto al bit `ICIE1` del registro `TIMSK1`.

- **Bits [1:0] - ACIS[1:0]:** (*Analog Comparator Interrupt Mode Select*) En estos bits se define el evento que produce la interrupción del comparador analógico, en la Tabla 6.8 se muestran los posibles eventos.

Tabla 6.7: Vector de interrupción para el AC

Dirección	Evento	Descripción
0x02E	ANALOG_COMP	Evento en el comparador analógico

Tabla 6.8: Selección del evento que genera la interrupción del AC

ACIS1	ACIS0	Tipo de evento
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 `DIDR1` (*Digital Input Disable Register*) sirve para desactivar un *buffer* digital en las entradas del comparador analógico, es un Registro I/O Extendido con dirección (0x7F), sus bits son:

REG.	7	6	5	4	3	2	1	0
DIDR1	-	-	-	-	-	-	AIN1D	AIN0D

- **Bits [1:0] - AIN[1:0]D:** (*Digital Input Disable*) Las terminales digitales incluyen un *buffer* de entrada, un 1 en cada uno de estos bits deshabilita a su *buffer* digital y con ello se reduce el consumo de potencia.

El registro `DIDR1` del comparador analógico tiene el mismo uso que el registro `DIDR0` en el convertidor analógico a digital. Ambos recursos manejan información analógica en sus terminales involucradas y es conveniente que desactiven sus *buffers* digitales.

Para que la entrada AIN1 del comparador analógico sea remplazada por la salida del multiplexor del convertidor analógico a digital, es necesario que en 3 de los registros del ADC se realicen las configuraciones adecuadas, en los bits que corresponden, estos son:

- El bit `ADEN` del registro `ADCSRA` debe tener un 0 para que el ADC esté deshabilitado.
- El bit `ACME` del registro `ADCSRB` debe tener un 1 para que el multiplexor se conecte con la entrada negativa del comparador.
- En los bits `MUX[2:0]` del registro `ADMUX` se debe seleccionar uno de los 8 canales externos (6 en dispositivos con encapsulado PDIP). El bit `MUX[3]` no está involucrado en la selección del canal porque para la entrada negativa del AC no es aplicable el uso de valores constantes o del sensor de temperatura.

6.2.3. Ejemplos con el Comparador Analógico

En esta sección se muestran 2 ejemplos del uso del comparador, las soluciones se codifican en lenguaje C.

Ejemplo 6.4 - Encendido Automático de un Ventilador
Realice un sistema que encienda un ventilador cuando la temperatura del ambiente esté por encima de $20°C$*, en caso contrario que el ventilador permanezca apagado.*

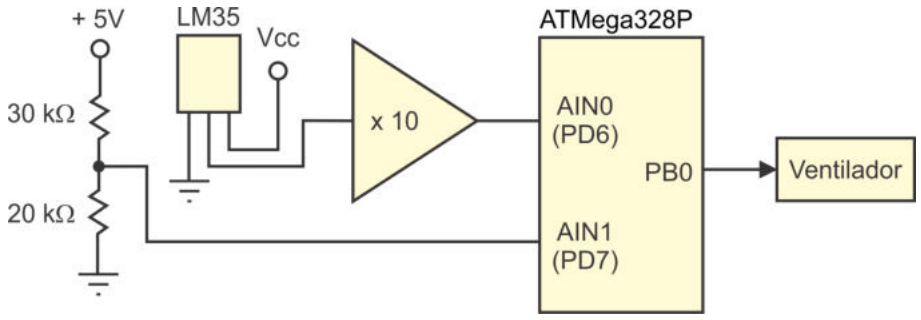


Figura 6.14: Hardware para mostrar el uso del comparador analógico

En la Figura 6.14 se muestra el acondicionamiento del hardware. 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 debe corresponder con la salida del comparador, reflejada en el bit `ACO`.

Con respecto al software, se emplea la interrupción por cambios en el comparador analógico, de manera que en la ISR únicamente se actualiza la salida de acuerdo con el bit `ACO`.

El programa en lenguaje C es:

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

ISR(ANALOG_COMP_vect){          // Hay un cambio en el comparador
    if( ACSR & 1 << ACO )       // Si la temperatura es alta
        PORTB |= 0x01;          // enciende el ventilador
    else
        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;                // El comparador interrumpirá con cualquier
                                // cambio en ACO
```

```
    DIDR1 = 0x03;              // Deshabilita los buffers digitales
    sei();                     // Habilitador global de interrupciones
    while (1){                 // Lazo infinito, permanece ocioso
        asm("nop");
    }
}
```

El siguiente ejemplo tiene por objetivo mostrar cómo emplear el multiplexor del ADC con el AC, para comparar dos entradas analógicas con una referencia.

Ejemplo 6.5 - Control Simple para un Móvil Seguidor de Línea
Desarrolle el programa para un móvil seguidor de línea, compuesto por 2 llantas traseras manejadas por motores independientes y una llanta de libre movimiento al frente. En la Figura 6.15 se muestra un bosquejo del hardware. El móvil debe seguir una línea blanca en una superficie con fondo negro, para detectar esas zonas se utilizará el comparador analógico y 2 canales del ADC. En la entrada AIN0 se ha puesto un potenciómetro para definir una referencia (V_{REF}) y se considera que los sensores proporcionan un voltaje mayor a V_{REF} cuando están en una zona clara y su voltaje está por debajo de V_{REF} en las zonas oscuras.
Los sensores se conectan en 2 de los canales del ADC para que alternadamente remplacen la entrada AIN1 del AC. El potenciómetro le da flexibilidad al móvil, porque V_{REF} se puede ajustar dependiendo de las condiciones de iluminación.

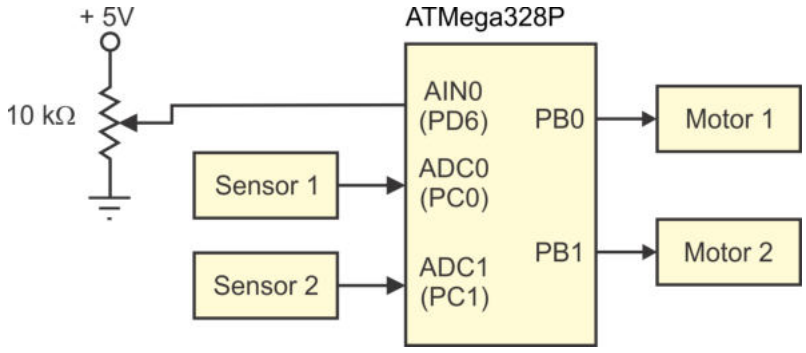


Figura 6.15: Hardware para el control de un móvil seguidor de línea

Los sensores estarán al frente del móvil, uno en cada extremo, dejando la línea blanca al centro y su estado 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. En ningún momento los 2 sensores deberían detectar una zona clara, sin embargo, si eso sucede, también se deben encender los 2 motores, buscando que el móvil abandone esta situación inesperada.

El programa se realiza sin el uso de interrupciones, en el lazo infinito se obtiene el estado de los sensores para definir las salidas de los motores.

El código en lenguaje C es:

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

int main(void) {                    // Programa Principal
uint8_t sens;                       // Estado de los sensores

   DDRB = 0xFF;                     // Puerto B como salida
   PORTB = 0x03;                    // Inicia con los 2 motores encendidos
                                    // El móvil inicia en una recta
   ADCSRA = 0x00;                   // ADC desactivado
   ADCSRB = 0x40;                   // Conecta al multiplexor con el AC
   DIDR0 = 0x03;                    // Deshabilita los buffers digitales
   DIDR1 = 0x01;

   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

      if(ACSR & 1 << ACO)           // Sensor 1 en zona oscura?
         sens |= 0x01;              // Guarda resultado
      ACSR = 0x80;                  // Desactiva al AC
      ADMUX = 0x01;                 // Selecciona el canal 1
      ACSR = 0x00;                  // Activa al AC
      asm("nop");                   // Espera el resultado

      if(ACSR & 1 << ACO)           // Sensor 2 en zona oscura?
         sens |= 0x02;              // Guarda resultado
      if( sens == 0x01 || sens == 0x02 )  // Un sensor en zona oscura?
         PORTB = sens;                    // Enciende su motor
      else
         PORTB = 0x03;                    // Enciende los 2 motores
   }
}
```

Es necesario desactivar al comparador analógico antes de definir el canal de entrada. Después de activarlo es conveniente esperar por lo menos un ciclo de reloj, para que la salida del AC sea estable.

6.3. Manejo del ADC y AC desde Arduino

El entorno Arduino incluye un par de funciones para el manejo del ADC, estas son:

- **analogReference**: Para configurar el voltaje de referencia que determina el valor máximo para la entrada analógica. La función puede recibir como argumento una de las siguientes constantes: DEFAULT, VEXTERNAL o INTERNAL, cuya descripción corresponde con la Tabla 6.3.

- **analogRead**: La función recibe como argumento el número de entrada o canal analógico en donde se hará la lectura, puede ser de A0 a A5, y regresa como resultado un número entre 0 y 1023, debido a que utiliza los 10 bits del convertidor. Dentro de la función se define el canal, configurando al registro `ADMUX`, se inicia la conversión y espera su culminación, con el apoyo del registro `ADCSRA`. Durante la conversión no es posible realizar otra tarea porque la función se basa en sondeo y no en interrupciones.

La ventaja de emplear las funciones de Arduino es que se simplifica el manejo del ADC, la desventaja es que no se puede solapar la operación del ADC con otras tareas, la función `analogRead` espera hasta tener el resultado de la conversión.

En el entorno de Arduino no hay funciones para el manejo del comparador analógico, de manera que solo se puede utilizar a través de los Registros I/O. Para mostrar como emplear al comparador analógico en una tarjeta Arduino, en el siguiente *sketch* se repite la solución del Ejemplo 6.4.

```
ISR(ANALOG_COMP_vect){          // Hay un cambio en el comparador
    if( ACSR & 1 << ACO )       // Si la temperatura es alta
        digitalWrite(8, HIGH);  // enciende el ventilador
    else
        digitalWrite(8, LOW);   // Sino, lo apaga
}

void setup() {
    pinMode(8, OUTPUT);       // Salida en PB1 (8 en Arduino)
    digitalWrite(8, LOW);     // Inicia con el ventilador apagado
    ACSR = 0x08;              // Interrupción por cambios en ACO
    DIDR1 = 0x03;             // Deshabilita los buffers digitales
}

void loop() {
    asm("nop");
}
```

6.4. Ejercicios

Los ejercicios principalmente están enfocados al manejo del ADC, ya que es mucho más versátil que el AC. Todos pueden resolverse en lenguaje C y es opcional el manejo de interrupciones.

1. Configure el temporizador 1 en su modo 7 para generar una señal PWM de 10 bits, la salida debe estar en modo no invertido y el ciclo de trabajo debe ser proporcional a un voltaje analógico recibido en el canal 0 del ADC.

2. Para el movimiento de un servomotor se requiere de una señal PWM con un periodo de 20 ms, el servomotor se mantiene en 0° (extremo izquierdo) con un

ancho de pulso de 0.9 ms y en 180° (extremo derecho) si el ancho es de 2.1 ms. Considerando que un servomotor tiene un comportamiento lineal, realice el programa para el circuito de la Figura 6.16 que lo haga girar de un extremo a otro, con un potenciómetro conectado a una entrada analógica. En la figura también se presenta la relación entre el valor digital del ADC y el ancho de pulso para el manejo del servomotor.

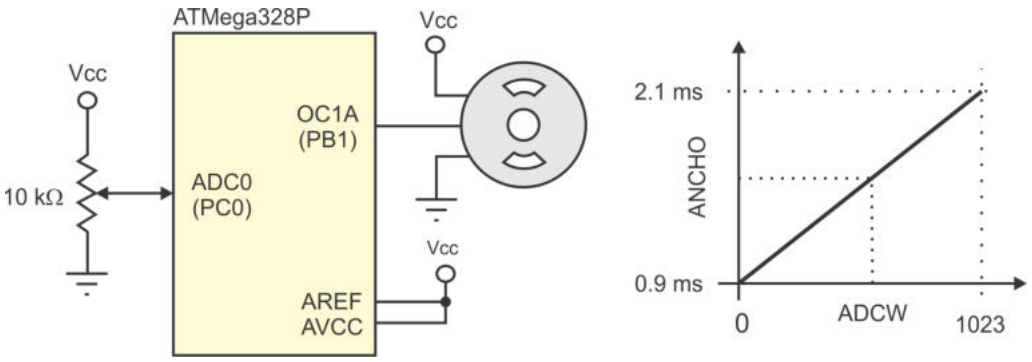


Figura 6.16: Manejo de un servomotor con un potenciómetro

3. Extienda el circuito del problema anterior y modifique el programa para manejar dos servomotores, uno en la salida OC1A y otro en OC1B, controlados por dos potenciómetros, uno conectado en la entrada ADC0 y el otro en ADC1.

4. Realice un termómetro digital empleando el hardware de la Figura 6.17, el sensor de temperatura entrega $10\,mV/°C$ pero se ha acondicionado para que el MCU reciba $0.1\,V/°C$. El termómetro debe mostrar la temperatura ambiente en dos dígitos para la parte entera y uno para la parte decimal. Puesto que el voltaje de referencia se ha establecido en 5 V, la temperatura máxima a mostrar será de $50°C$. La temperatura es un parámetro que cambia muy lentamente, por lo que la lectura del sensor se debe realizar cada medio segundo, utilizando al temporizador 1 como mecanismo para obtener este periodo sin afectar la exhibición en los visualizadores de 7 segmentos.

   Se sugiere manipular los recursos por interrupciones, en la ISR del temporizador 1 marcar el inicio de una conversión y en la ISR de fin de conversión actualizar los datos, de esta manera, el lazo infinito solo se dedica a mostrar la temperatura en los visualizadores de 7 segmentos.

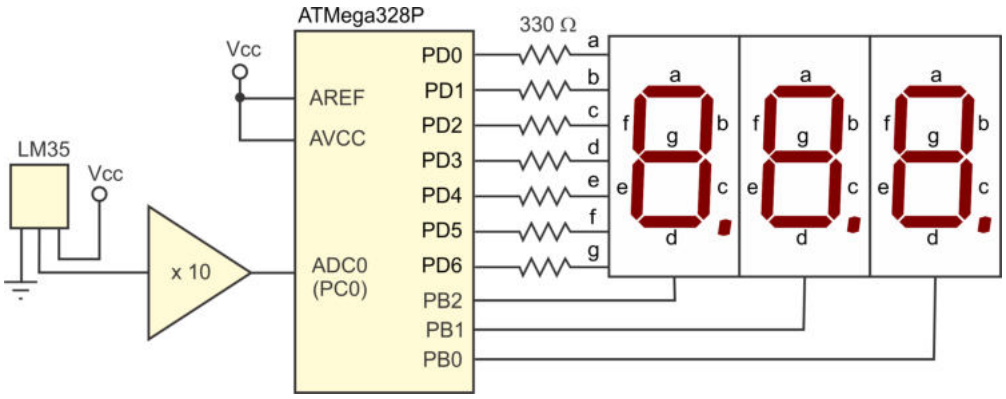


Figura 6.17: Termómetro digital para la temperatura ambiente

5. Un fotoresistor iluminado con luz solar presenta una resistencia con un valor de 300 Ω, sin iluminación la resistencia está por encima de 20 $K\Omega$. Desarrolle un circuito que encienda una lámpara si la resistencia en el sensor es mayor a 15 $K\Omega$ y la apague cuando la resistencia esté por debajo de 1 $K\Omega$.

   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 analógico con el multiplexor, en la terminal AIN0 se introduce el voltaje debido al sensor y en dos canales del multiplexor se colocan los voltajes producidos con resistores de 15 $K\Omega$ y 1 $K\Omega$, para cambiar la entrada de comparación, de acuerdo con el estado de la salida.

6. Modifique el Ejemplo 6.5 manejando los motores del móvil con señales PWM. En una línea recta, ambos motores deben tener un ciclo de trabajo del 100 %. En una curva, el motor del sensor que detecte la zona clara debe trabajar con un ciclo 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.

7. Con respecto a la Figura 6.18, 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 Vin es menor a Vref1.
   - 50 %: si Vin está entre Vref1 y Vref2.
   - 90 %: si Vin es mayor a Vref2.

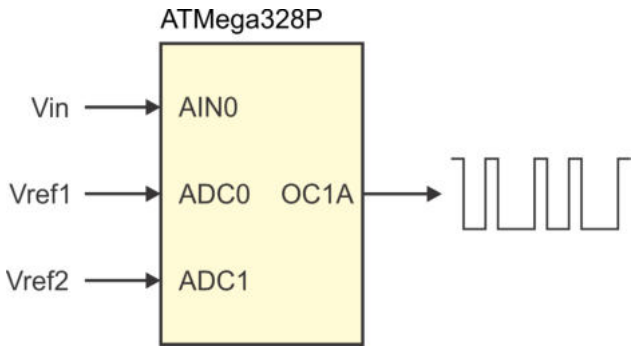


Figura 6.18: Uso del AC en la generación de una señal PWM

8. Con un microcontrolador ATMega328P digitalice 2 señales analógicas (V1 y V2) manteniendo la resolución de 10 bits. Si $V1 > V2$, genere una señal PWM en OC1A; en caso de que $V1 < V2$, la señal PWM se generará en OC1B. Cuando $V1 = V2$, tanto en OC1A como en OC1B se tendrá una salida de 0 V (o PWM con 0 %).

   La señal PWM debe tener una frecuencia de 50 Hz y un ancho de pulso proporcional a la diferencia entre V1 y V2. Note que la diferencia máxima se obtiene cuando V1 = 5 V y V2 = 0, o viceversa, y en ese caso la señal PWM tendrá un ciclo de trabajo del 100 %.

9. Realice un control proporcional de temperatura empleando 2 canales del ADC de un ATMega328P. En ADC0 se conectará un sensor de temperatura, acondicionado para proporcionar un intervalo entre 0 y 50°C (Treal). En ADC1 se conectará un potenciómetro para generar un voltaje analógico (Tref) escalado a un número entre 0 y 50 (mantenga una representación en punto flotante). El sistema debe generar una señal PWM para el manejo de un ventilador de CD, bajo el siguiente esquema:

   - Si $Treal <= Tref$ el ventilador estará apagado.
   - Si $Treal > Tref$ y $Treal <= Tref + 3.0$, la señal PWM será proporcional a $Treal - Tref$. Considerando un comportamiento lineal para un ciclo de trabajo entre 0 y 100 %.
   - Si $Treal > Tref + 3.0$ el ventilador estará encendido (señal PWM al 100 %).

   En la Figura 6.19 se muestra el comportamiento esperado en el sistema. La señal PWM debe tener una frecuencia aproximada de 500 Hz y las operaciones del MCU para definir su ciclo de trabajo se deben realizar cada 300 ms.

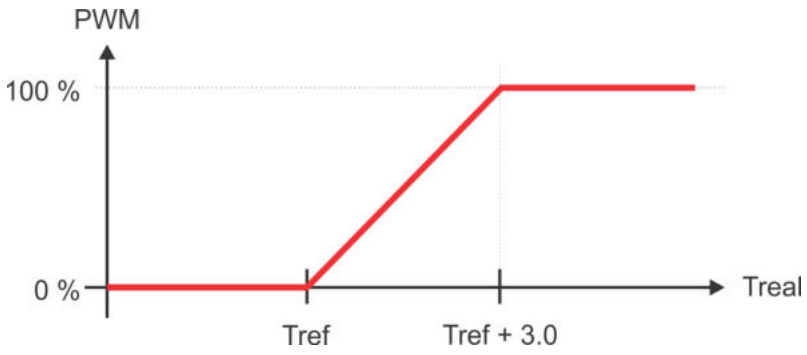


Figura 6.19: Control proporcional de temperatura

10. Con un arreglo de resistores se pueden sondear varios botones en una entrada analógica, como se muestra en la Figura 6.20. Realice un programa que evalúe la entrada ADC0 e indique si hay un botón presionado escribiendo un número entre 1 y 5 en el visualizador de 7 segmentos, si no hay botón presionado debe escribir un 0. Puesto que los resistores tienen tolerancias, difícilmente se obtendrán los valores teóricos esperados, por lo que es conveniente considerar la comparación del valor digitalizado por intervalos.

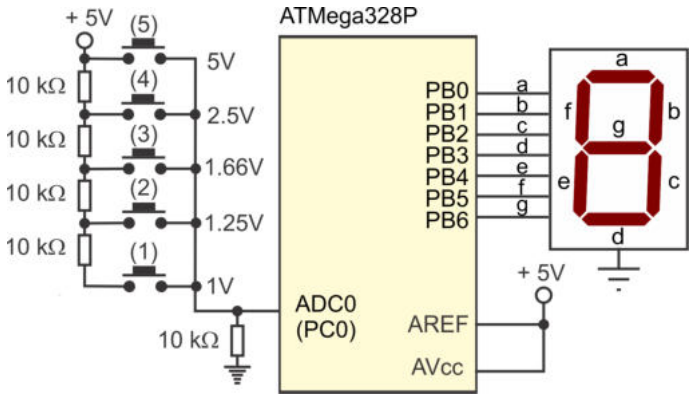


Figura 6.20: Arreglo para sondear 5 botones en una entrada analógica

Capítulo 7

Comunicación Serial (Parte I)

El microcontrolador ATMega328P incluye 3 recursos para comunicarse de manera serial con dispositivos externos, empleando interfaces estándares. Estos recursos son:

- La USART (*Universal Synchronous and Asynchronous Serial Receiver and Transmitter*), transmisor-receptor universal síncrono y asíncrono.
- La interfaz SPI (*Serial Peripheral Interface*), interfaz estándar para periféricos seriales.
- La interfaz TWI (*Two-Wire Serial Interface*), interfaz serial de dos hilos, completamente compatible con el bus estándar I^2C (*inter-integrated circuits*).

Los primeros dos recursos (USART y SPI) son descritos en el presente capítulo, incluyendo el modo de operación de la USART como interfaz SPI. La interfaz TWI es tratada en el siguiente capítulo.

7.1. Comunicación Serial a través de la USART

La USART (*Universal Synchronous and Asynchronous Serial Receiver and Transmitter*) 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), en el ATMega328P.

La comunicación puede ser síncrona o asíncrona, la diferencia entre estos modos se ilustra en la Figura 7.1. 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 y se ubica en PD4.

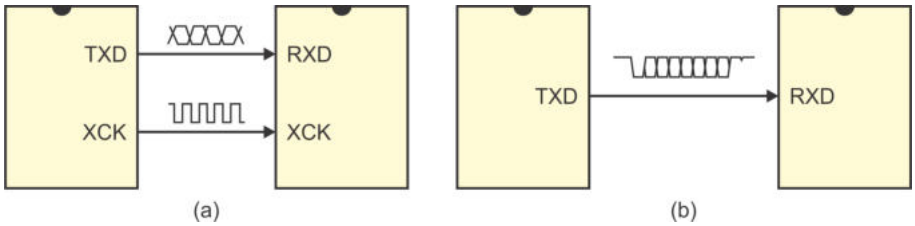


Figura 7.1: Comunicación serial (a) síncrona y (b) asíncrona

En una comunicación asíncrona, el emisor únicamente envía los datos y no hay envío de la señal de reloj. Esta forma de comunicación es más utilizada en forma práctica por el ahorro del alambre para el reloj, pero requiere que antes de cualquier transferencia de datos, en el transmisor y el receptor se definan 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 predefinido de bits, la interfaz permite manejar desde 5 hasta 9 bits, aunque es más frecuente utilizar datos de 8 bits.
- **Bit de paridad:** Es un bit de seguridad que opcionalmente acompaña a cada dato, automáticamente se pone en alto o en bajo para complementar un número par o impar de 1's (se define como paridad par o impar). El emisor envía al bit de paridad después de enviar los bits del dato, el receptor lo calcula a partir del dato recibido 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 si se hará 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 o 2 bits de paro.

Por cada dato que se envía se debe integrar una trama serial, esta contiene:

- 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 contienen un 1 lógico).

En la Figura 7.2 se ilustra una trama serial, considerando un dato de 8 bits, bit de paridad y 1 bit de paro. Se observa que la línea de comunicación mantiene un nivel lógico alto mientras no inicie la transmisión de un dato.

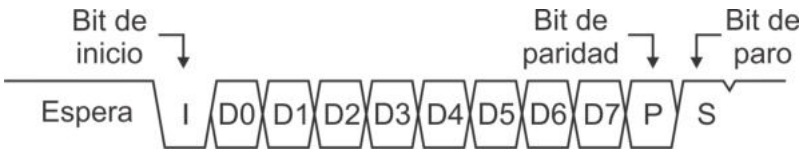


Figura 7.2: Trama serial de datos

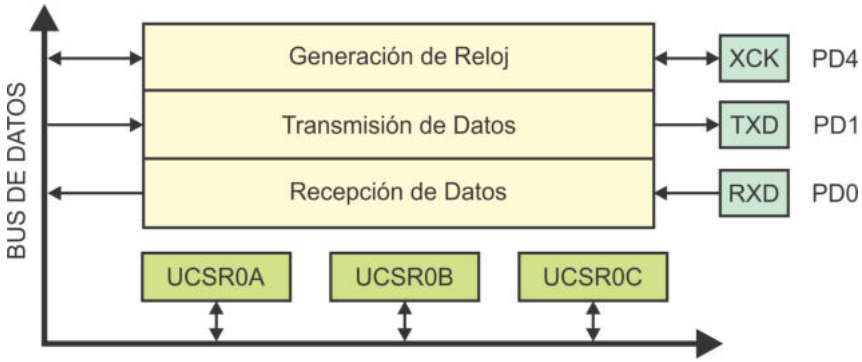


Figura 7.3: Organización de la USART

7.1.1. Organización de la USART

El ATMega328P contiene solo un módulo USART, se denomina USART0 y el 0 es para mantener compatible el nombre con versiones que contienen más módulos de este tipo, por ejemplo, el ATMega2560 contiene cuatro módulos USART y se denominan USART0, USART1, USART2 y USART3. Los módulos prácticamente son iguales, para tener acceso a los otros módulos USART de un ATMega2560 habrá que cambiar el 0 en los nombres de los registros y en sus respectivos bits, se hace referencia al ATMega2560 porque está incluido en la tarjeta Arduino Mega.

El módulo USART0 se compone de 3 bloques principales, estos se observan en la Figura 7.3, el generador de reloj proporciona las señales de sincronización a los otros 2 bloques, los bloques de transmisión y recepción trabajan en forma independiente. En la Figura 7.3 también pueden verse los registros `UCSR0A`, `UCSR0B` y `UCSR0C`, con estos registros se realiza la configuración y se conoce el estado de la USART (`UCSR`, *USART Configuration and Status Register*).

Generación de Reloj y Modos de Operación

El bloque para la generación de reloj tiene 2 señales de salida: CLK1 y CLK2, como se muestra en la Figura 7.4. La señal CLK1 es el reloj utilizado para la transmisión o recepción asíncrona, así como para la transmisión síncrona, la frecuencia de CLK1 se define por medio del registro `UBRR0`. La señal CLK2 es el reloj utilizado para la recepción síncrona, es una señal externa que se recibe y sincroniza, su frecuencia la determina el transmisor.

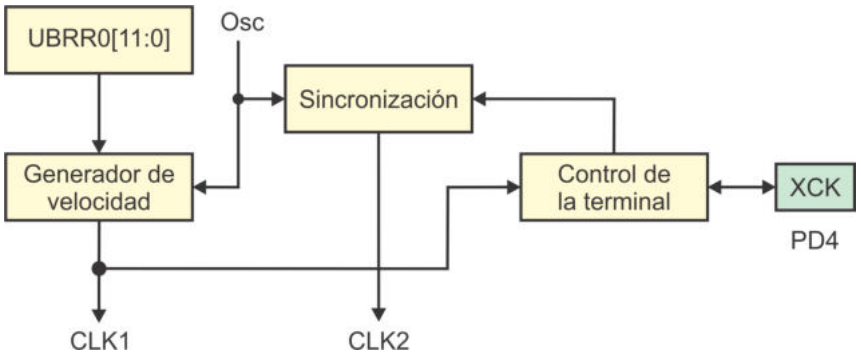


Figura 7.4: Bloque para la generación del reloj en la USART0

El registro `UBRR0` proporciona la base para la generación de la señal CLK1 (`UBRR`, *USART Baud Rate Register*). Se compone de 2 Registros I/O Extendidos: `UBRR0H` y `UBRR0L`, aunque solo 12 de los 16 bits disponibles son utilizados.

El bloque **generador de velocidad** incluye un contador descendente cuyo valor máximo es tomado del registro `UBRR0`, el contador recarga su valor máximo cuando llega a 0 o cuando se escribe en `UBRR0L`. 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 `UBRR0` + 1.

La señal interna pasa por tres divisores de frecuencia conectados en cascada, de manera que la frecuencia base es dividida entre 2, 8 o 16. El factor de división depende del modo de operación seleccionado para la USART, la cual soporta 4 modos diferentes:

- **Asíncrono normal:** En este modo la frecuencia de CLK1 es la frecuencia base entre 16.
- **Asíncrono a doble velocidad:** Utiliza el factor de división de 8 para que la frecuencia de CLK1 sea el doble que el modo asíncrono 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 modo el MCU está funcionando como un receptor síncrono, por lo que también recibe la señal de reloj en XCK. La señal recibida 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 registro `UCSR0C` en sus posiciones 7 y 6 contiene a los bits `UMSEL0[1:0]` para definir el modo, la combinación “00” es para un modo asíncrono y “01” para un modo síncrono. Si se elige el modo asíncrono, con

el bit `U2X0` (bit 1 del registro **UCSR0A**) se debe seleccionar entre una operación normal (`U2X0` = '0') y una operación a doble velocidad (`U2X0` = '1').

En la Tabla 7.1 se muestran las relaciones matemáticas para obtener la velocidad de transmisión (*baud rate*) a partir del valor del registro `UBRR0`, no obstante, en la práctica primero se define a qué velocidad se va a transmitir, para después calcular el valor de `UBRR0`, estas relaciones también están en la Tabla 7.1.

Tabla 7.1: Cálculo de la velocidad de transmisión y del valor para UBRR0

Modo	**Velocidad de transmisión**	**Valor para UBRR0**
Asíncrono normal	$Baud\_Rate = \frac{f_{OSC}}{16(UBRR0+1)}$	$UBRR0 = \frac{f_{OSC}}{16 \times Baud\_Rate} - 1$
Asíncrono a doble velocidad	$Baud\_Rate = \frac{f_{OSC}}{8(UBRR0+1)}$	$UBRR0 = \frac{f_{OSC}}{8 \times Baud\_Rate} - 1$
Síncrono como maestro	$Baud\_Rate = \frac{f_{OSC}}{2(UBRR0+1)}$	$UBRR0 = \frac{f_{OSC}}{2 \times Baud\_Rate} - 1$

Debe tomarse en cuenta que al calcular el valor para el registro `UBRR0` se obtiene un número real, por ello, al redondear a entero se produce un error en la velocidad de transmisión. El porcentaje de error se puede cuantificar con la expresión:

$$Error[\,\%] = (\frac{Baud\_Rate(real)}{Baud\_Rate(esperado)} - 1) \times 100\,\%$$

Como un ejemplo para observar el comportamiento del error, se considera una comunicación asíncrona a 9600 bps con un oscilador de 1 MHz. Primero se va a calcular el valor del registro `UBRR0` y después se obtendrá el porcentaje de error para un modo asíncrono normal:

$$UBRR0 = \frac{f_{OSC}}{16 \times Baud\_Rate} - 1 = \frac{1MHz}{16 \times 9600} - 1 = 5.51$$

Redondeando el valor obtenido para `UBRR0` a 6, se calcula la velocidad de transmisión real:

$$Baud\_Rate = \frac{f_{OSC}}{16(UBRR0+1)} = \frac{1MHz}{16(6+1)} = 8928.57\ bps$$

El porcentaje de error generado es de:

$$Error[\,\%] = (\frac{Baud\_Rate(real)}{Baud\_Rate(esperado)} - 1) \times 100\,\% = (\frac{8928.57}{9600} - 1) \times 100\,\% = -6.99\,\%$$

El error es muy significativo, si el valor para el registro `UBBR0` se redondea a 5, se obtendrá una velocidad de transmisión real de 10416.66 bps, la que producirá un error del 8.5 %, que también es muy grande pero con signo contrario.

Ahora se van a repetir los cálculos pero considerando el modo asíncrono a doble velocidad, primero se calcula el valor del registro `UBRR0` y después se obtiene el porcentaje de error generado:

$$UBRR0 = \frac{f_{OSC}}{8 \times Baud\_Rate} - 1 = \frac{1MHz}{8 \times 9600} - 1 = 12.02$$

Redondeando el valor obtenido para `UBRR0` a 12, se calcula la velocidad de transmisión real:

$$Baud\_Rate = \frac{f_{OSC}}{8(UBRR0+1)} = \frac{1MHz}{8(12+1)} = 9615.38\ bps$$

El porcentaje de error generado es de:

$$Error[\,\%] = (\frac{9615.38}{9600} - 1) \times 100\,\% = 0.16\,\%$$

El error se reduce drásticamente al usar el modo a doble velocidad.

Un error positivo significa que la velocidad de transmisión real es mayor que la velocidad esperada y un error negativo es por la causa contraria, es decir, la velocidad de transmisión real es menor que la velocidad esperada. Para la sincronización entre transmisor y receptor, el transmisor debe colocar los datos en la línea de comunicación y el receptor, una vez que detecte al bit de inicio empezará a tomar muestras lo más cercano del punto medio de cada bit, para hacer lecturas cuando el bit está estable.

Si el transmisor tiene una velocidad más alta, el receptor leerá en un punto posterior al medio y en cada bit el retraso será más significativo, con un error mayor al 5 % podría ser que en lugar de leer al bit D7, el receptor lea al bit de paridad o al bit de paro, si no se usó bit de paridad. Con errores negativos el receptor va a leer un poco antes del punto medio, si la magnitud del error es grande se podría estar leyendo el valor de D7 cuando se esperaba al bit de paridad o al bit de paro. En cualquier caso y por la cantidad de bits que hay en una trama, los errores en el rango de ± 5 % aun son aceptables.

En la Tabla 7.2 se muestran algunos valores a emplear en el registro `UBRR0` para velocidades de transmisión típicas en el modo de comunicación asíncrona, se consideran frecuencias de operación de 1, 8 y 16 MHz para el ATMega328P y se muestran los porcentajes de error en cada caso. Las frecuencias de 1 y 8 MHz se pueden obtener con el oscilador interno del MCU y en caso de emplear una tarjeta Arduino, esta tendrá un cristal de 16 MHz.

En el modo de comunicación síncrona no existe un error en la velocidad porque la señal de reloj es enviada o recibida en la terminal XCK. De acuerdo con la Tabla 7.1, la frecuencia máxima de transmisión se obtiene cuando el registro `UBRR0` tiene 0 y corresponde con *fosc/2*. Sin embargo, la recepción síncrona requiere que la señal recibida en XCK se sincronice con el oscilador interno y esto consume algunos ciclos de reloj, por ello, el límite de la frecuencia para la recepción síncrona es de *fosc/4*.

Otro aspecto que se debe definir en operaciones síncronas es la polaridad de los datos con respecto a la señal de reloj, el transmisor coloca los datos en un flanco y el receptor hace las lecturas en el flanco contrario. El bit `UCPOL0` (bit 0 del registro `UCSR0C`) determina el flanco en que los datos son escritos por el transmisor o leídos por el receptor, en la Tabla 7.3 se muestran las opciones.

Tabla 7.2: Velocidades de transmisión típicas

Velocidad (bps)	**Normal**		**Doble Velocidad**	
	UBRR0	**Error [%]**	**UBRR0**	**Error [%]**
Fosc = 1 MHz				
1 200	51	0.16	103	0.16
2 400	25	0.16	51	0.16
4 800	12	0.16	25	0.16
9 600	6	-6.99	12	0.16
19 200	2	8.51	6	-6.99
57 600	0	8.51	1	8.51
115 200	0	-45.74	0	8.51
Fosc = 8 MHz				
1 200	416	-0.08	832	0.04
2 400	207	0.16	416	-0.08
4 800	103	0.16	207	0.16
9 600	51	0.16	103	0.16
19 200	25	0.16	51	0.16
57 600	8	-3.54	16	2.12
115 200	3	8.51	8	-3.55
Fosc = 16 MHz				
1 200	832	0.04	1 666	-0.02
2 400	416	-0.08	832	0.04
4 800	207	0.16	416	-0.08
9 600	103	0.16	207	0.16
19 200	51	0.16	103	0.16
57 600	16	2.12	34	-0.79
115 200	8	-3.55	16	2.12

Tabla 7.3: Polaridad en la comunicación síncrona

UCPOL0	**Escritura en TXD**	**Lectura en RXD**
0	Flanco de subida	Flanco de bajada
1	Flanco de bajada	Flanco de subida

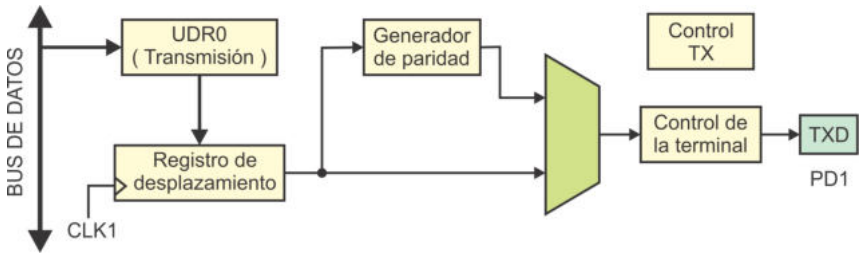


Figura 7.5: Bloque para la transmisión de datos

Transmisión de Datos

La Figura 7.5 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 o simplemente registro `UDR0` (*USART0 I/O Data Register*). Con el nombre `UDR0` realmente se hace referencia a dos registros ubicados en la misma dirección, un registro solo de escritura para transmitir datos y un registro solo de lectura para recibirlos.

El bloque de transmisión de datos requiere de una habilitación para que inicie su operación, el bit `TXEN0` (bit 3 del registro `UCSR0B`) es el habilitador del recurso. La transmisión inicia cuando un dato se escribe en el registro `UDR0`, previamente se deben configurar los parámetros: velocidad, número de bits por dato, paridad y número de bits de paro. Un registro de desplazamiento se encarga de la conversión de paralelo a serie, sincronizado con la señal CLK1.

El fin de la transmisión de un dato se indica con la puesta en alto de la bandera `TXC0` (bit 6 del registro `UCSR0A`), 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 reescribirle un 1 lógico. No es posible escribir en el registro `UDR0` mientras hay una transmisión en proceso, por ello, además del uso de la bandera `TXC0` es posible usar la bandera `UDRE0` (bit 5 del registro `UCSR0A`). La bandera `UDRE0` se pone en alto automáticamente 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 `UDRE0` depende de la actividad que ocurre en el registro `UDR0`.

En la práctica únicamente se emplea una de las 2 banderas, `TXC0` o `UDRE0`. 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, `TXC0` se pone en alto y, además, el *buffer* transmisor queda vacío por lo que `UDRE0` también se pone en alto.

El bit de paridad se genera por hardware conforme los datos se van transmitiendo y queda disponible para ser transmitido después de los bits del dato. Si se configura

el uso del bit de paridad, su generación y colocación en la trama de cada dato se realiza de manera automática.

Recepción de Datos

En la Figura 7.6 se muestra la organización del bloque para la recepción de datos. También contiene un registro de datos de la USART (`UDR0`), aunque en este caso es un registro solo 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 recurso debe habilitarse para que en cualquier momento pueda recibir un dato, la habilitación se realiza con la puesta en alto del bit `RXEN0` (bit 4 del registro `UCSR0B`).

El dato recibido es ubicado en el registro `UDR0`, la indicación de que hay un dato disponible se realiza con la puesta en alto de la bandera `RXC0` (Bit 7 del registro `UCSR0A`). Esta bandera puede sondearse por software o configurar al recurso para que genere una interrupción. La lectura del registro `UDR0` limpia la bandera y libera al registro, quedando disponible para recibir un nuevo dato. En la práctica lo más conveniente es habilitar la interrupción por recepción para evitar el sondeo dentro del lazo infinito.

El cálculo del bit de paridad y su validación, comparándolo con el valor del bit de paridad recibido, se realizan automáticamente por hardware. Si existe un error, este se indica con la puesta en alto de la bandera `PE0` (bit 2 del registro `UCSR0A`; `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 `FE0` (bit 4 del registro `UCSR0A`). El otro error es debido a un exceso de datos por recepción, ocurre cuando el registro `UDR0` 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 `DOR0` (bit 3 del registro `UCSR0A`; `DOR`, *Data OverRun*).

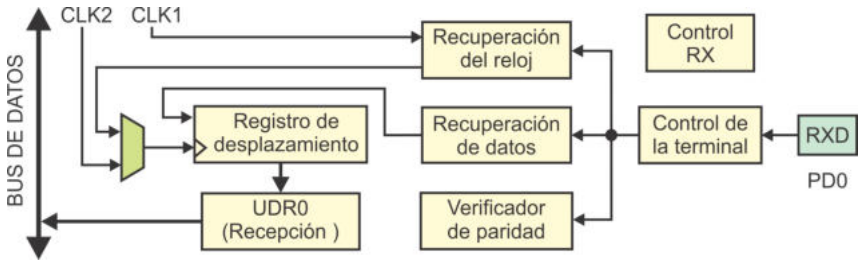


Figura 7.6: Bloque para la recepción de datos

Tabla 7.4: Registros para el manejo de la USART0

Registro	Dirección	Operación
UDR0	(0xC6)	Registro de datos de la USART
UBRR0H	(0xC5)	Byte alto del registro para establecer la velocidad
UBRR0L	(0xC4)	Byte bajo del registro para establecer la velocidad
UCSR0A	(0xC0)	Registro A para el control y estado de la USART
UCSR0B	(0xC1)	Registro B para el control y estado de la USART
UCSR0C	(0xC2)	Registro C para el control y estado de la USART

Transmisión y Recepción de Datos de 9 Bits

La transmisión y recepción de datos se realiza por medio del registro `UDR0`, el cual físicamente está compuesto por 2 registros de 8 bits, uno solo de escritura para transmisión y otro solo 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 `UCSR0B`. El bit `TXB80` (bit 0 del registro `UCSR0B`) es el noveno en una transmisión de 9 bits y el bit `RXB80` (bit 1 del registro `UCSR0B`) es el noveno durante una recepción de 9 bits.

Para una transmisión primero debe definirse el valor de `TXB80`, porque la transmisión inicia cuando se escribe en el registro `UDR0`. En una recepción primero debe leerse el valor del bit `RXB80` y luego al registro `UDR0`. Tanto `TXB80` como `RXB80` corresponden con el bit más significativo en datos de 9 bits.

7.1.2. Registros para el manejo de la USART0

En la Tabla 7.4 se muestran todos los registros involucrados en el manejo de la USART0, por su dirección se observa que todos son Registros I/O Extendidos.

El registro `UDR0` sirve para el manejo de datos en la USART0. Físicamente se compone de 2 registros de 8 bits con la misma dirección, uno de escritura para la transmisión de datos y otro de lectura para la recepción. La trasmisión inicia automáticamente al escribir en `UDR0` (transmisor) y cuando se recibe un dato, este queda disponible en `UDR0` (receptor). Los recursos de transmisión y recepción son independientes, solo el nombre de `URDR0` es compartido.

Para definir la velocidad de transmisión (*baud rate*) se utiliza al registro `UBRR0`, este es un registro de 12 bits, por lo tanto, se integra por 2 registros de 8 bits: `UBRR0L` para la parte baja y `UBRR0H` para la parte alta, aunque de `UBRR0H` solo los 4 bits menos significativos son utilizados. En lenguaje C se puede emplear el nombre de `UBRR0` como un registro de 16 bits.

La USART0 tiene 3 registros de control y estado, estos son: `UCSR0A`, `UCSR0B` y `UCSR0C`. En el registro `UCSR0A`, los 6 bits más significativos corresponden con las

banderas de estado de la USART0 y con los 2 menos significativos se pueden configurar algunos parámetros. Los bits del registro `UCSR0A` son:

REG.	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
UCSR0A	RXC0	TXC0	UDRE0	FE0	DOR0	PE0	U2X0	MPCM0

- **Bit 7 - RXC0:** (*USART Receive Complete*) Indica que hay un dato disponible por recepción. Puede generar una interrupción. Se limpia al leer al registro `UDR0`.
- **Bit 6 - TXC0:** (*USART Transmit Complete*) 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 si se reescribe un 1 lógico en `TXC0`.
- **Bit 5 - UDRE0:** (*USART Data Register Empty*) Indica que el registro `UDR0` 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 `UDR0`.
- **Bit 4 - FE0:** (*Frame Error*) Esta bandera se pone en alto cuando el bit de paro en el *buffer* receptor es 0, indicando un error de marco. Se mantiene con ese valor hasta que se haga la lectura del registro `UDR0`.
- **Bit 3 - DOR0:** (*Data OverRun*) Indica un exceso de datos por recepción, esto significa que el registro receptor (`UDR0`) 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 se haga la lectura del registro `UDR0`.
- **Bit 2 - PE0:** (*Parity Error*) 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 se haga la lectura del registro `UDR0`.
- **Bit 1 - U2X0:** (*Double the USART Transmission Speed*) En operaciones asíncronas, este bit debe ponerse en alto para cambiar del modo asíncrono normal al modo asíncrono a doble velocidad.
- **Bit 0 - MPCM0:** (*Multi-processor Communication Mode*) Habilita un modo de comunicación entre multiprocesadores, bajo un esquema maestro-esclavos y datos de 9 bits. Este modo se describe en un apartado posterior.

El registro `UCSR0B` incluye los habilitadores para la generación de interrupciones y los habilitadores de recursos, además de otros bits de interés. Los bits del registro `UCSR0B` son:

REG.	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
UCSR0B	RXCIE0	TXCIE0	UDRIE0	RXEN0	TXEN0	UCSZ02	RXB80	TXB80

- **Bit 7 - RXCIE0:** (*RX Complete Interrupt Enable*) Si es puesto en alto, y el habilitador global de interrupciones también, se produce una interrupción cuando se reciba un dato (`RXC0` en alto).
- **Bit 6 - TXCIE0:** (*TX Complete Interrupt Enable*) Si es puesto en alto, y el habilitador global de interrupciones también, se produce una interrupción cuando finalice la transmisión de un dato (`TXC0` en alto).
- **Bit 5 - UDRIE0:** (*USART Data Register Empty Interrupt Enable*) Si es puesto en alto, y el habilitador global de interrupciones también, se produce una interrupción cuando el *buffer* transmisor está vacío (`UDRE0` en alto).
- **Bit 4 - RXEN0:** (*Receiver Enable*) 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 - TXEN0:** (*Transmitter Enable*) 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 - UCSZ02:** (*Character Size*) Junto con los bits `UCSZ0[1:0]` del registro `UCSR0C` permiten definir el tamaño de los datos a enviar o recibir. En la Tabla 7.7 se muestran los diferentes tamaños, de acuerdo con el valor de los bits `UCSZ0[2:0]`.
- **Bit 1 - RXB80:** (*Receive Data Bit 8*) Durante la recepción de datos de 9 bits, los 8 bits menos significativos se reciben en el registro `UDR0`, en `RXB80` se recibe al bit más significativo.
- **Bit 0 - TXB80:** (*Transmit Data Bit 8*) Para transmitir datos de 9 bits, los 8 bits menos significativos deben ubicarse en el registro `UDR0`, en `TXB80` debe colocarse al bit más significativo.

El último registro para la configuración y estado de la USART es el `UCSR0C`, en este registro se definen las características que tendrá la comunicación, sus bits son:

REG.	7	6	5	4	3	2	1	0
UCSR0C	UMSEL01	UMSEL00	UPM01	UPM00	USBS0	UCSZ01	UCSZ00	UCPOL0

- **Bits [7:6] - UMSEL0[1:0]:** (*USART Mode Select*) Con estos bits se selecciona el modo de operación de la USART0, en la Tabla 7.5 se muestran los diferentes modos, de acuerdo con el valor de los bits `UMSEL0[1:0]`.
- **Bits [5:4] - UPM0[1:0]:** (*Parity Mode*) Con las combinaciones de estos bits se determina el uso del bit de paridad y se define su tipo. En la Tabla 7.6 se muestran las diferentes opciones.
- **Bit 3 - USBS0:** (*Stop Bit Select*) Una trama para un dato concluye con el bit de paro (siempre tiene un 1 lógico), si el bit `USBS0` se pone en alto, en lugar de ser uno serán dos bits de paro.

Tabla 7.5: Modos de operación de la USART0

UMSEL01	UMSEL00	Modo de la USART0
0	0	Operación asíncrona
0	1	Operación síncrona
1	0	Reservado
1	1	Modo Maestro SPI (MSPIM)

Tabla 7.6: Activación del bit de paridad y sus diferentes tipos

UPM01	UPM00	Configuración
0	0	Sin bit de paridad
0	1	Reservado
1	0	Paridad par
1	1	Paridad impar

- **Bits [2:1] - UCSZ0[1:0]:** (*Character Size*) Junto con el bit `UCSZ02` del registro `UCSR0B` definen el tamaño de los datos a enviar o recibir. En la Tabla 7.7 se muestran los diferentes tamaños, de acuerdo con el valor de los bits `UCSZ0[2:0]`.

- **Bit 0 - UCPOL0:** (*Clock Polarity*) En operaciones síncronas, el bit `UCPOL0` determina el flanco de reloj en el cual se escriben los datos (cuando se está transmitiendo) o en el que son leídos (cuando se está recibiendo). En la Tabla 7.3 se mostraron los flancos de activación o polaridad, dependiendo del valor del bit `UCPOL0`.

Es importante señalar que si se utiliza el modo Maestro SPI (MSPIM), algunos bits en los registros de configuración y control dejan de tener uso, mientras que otros cambian su funcionalidad. El modo MSPIM se describe en la Sección 7.3.

Tabla 7.7: Opciones para el tamaño de los datos

UCSZ02	UCSZ01	UCSZ00	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 7.8: Vectores de interrupción para la USART0

Dirección	Evento	Descripción
0x024	USART_RX	Recepción de un dato por la USART0
0x026	USART_UDRE	Registro de datos de la USART0 vacío
0x028	USART_TX	Transmisión de un dato por la USART0

7.1.3. Interrupciones debidas a la USART0

El módulo USART0 puede interrumpir a la CPU por 3 eventos: recepción completa, transmisión completa y *buffer* transmisor vacío. Las banderas que generan estos eventos son: `RXC0`, `TXC0` y `UDRE0`, respectivamente y corresponden con los bits más significativos del registro `UCSR0A`.

Las interrupciones se deben habilitar, cada una tiene su habilitación individual, correspondiendo con los bits: `RXCIE0`, `TXCIE0` y `UDRIE0`, estos bits corresponden con los más significativos del registro `UCSR0B`. Una interrupción se va a realizar solo si está en alto el habilitador global de interrupciones, que es el bit `I` del registro `SREG`. En la Tabla 7.8 se muestran los vectores de las interrupciones debidas a la USART0.

7.1.4. Comunicación entre Múltiples Microcontroladores

El transmisor-receptor universal asíncrono es un recurso presente en microcontroladores y otros dispositivos, por ejemplo, algunos módulos Wi-Fi o módulos Bluetooth utilizan esta interfaz para su manejo. La comunicación generalmente es punto a punto, es decir, solo existe un transmisor y un receptor. Sin embargo, la USART de los AVR permite una comunicación entre más de 2 microcontroladores, bajo un esquema maestro-esclavos, de esta manera, un MCU-Maestro se puede comunicar hasta con 256 esclavos.

En la Figura 7.7 se ilustra esta organización, los esclavos deben tener una dirección diferente a la que darán respuesta, la dirección es un número 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.

La comunicación utiliza un formato de 9 bits, donde el noveno bit (transmitido en `TXB80` o recibido en `RXB80`) sirve para distinguir entre dos tipos de información, un 0 indica que es una trama de datos y un 1 que es una trama de dirección.

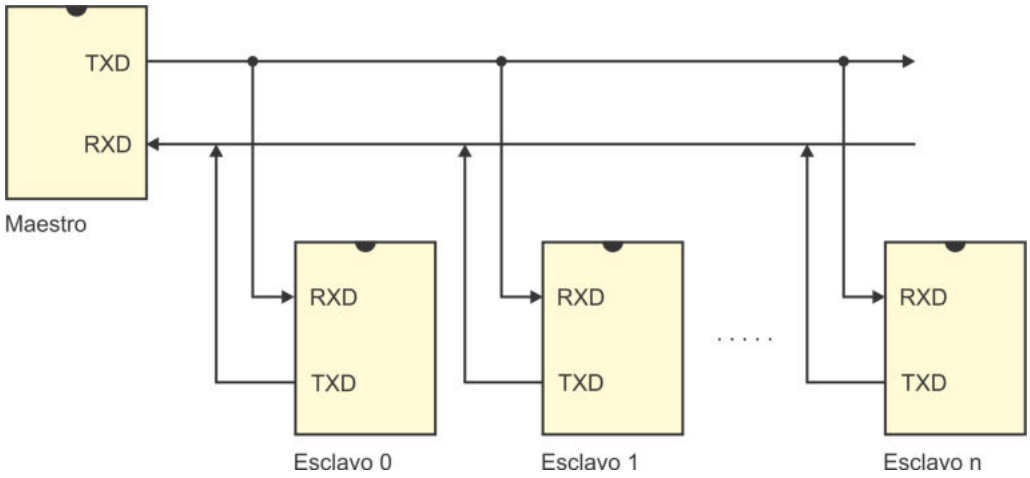


Figura 7.7: Organización maestro-esclavos para múltiples microcontroladores

La comunicación entre múltiples MCUs requiere el uso del bit `MPCM0` (bit 0 del registro `UCSR0A`). Este bit es utilizado por los esclavos, si está en alto, únicamente pueden recibir tramas de dirección e ignoran las tramas de datos. 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 `MPCM0` = 1. De manera que únicamente pueden recibir tramas de dirección (9° 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 `UDR0` y lo compara con su dirección para determinar si ha sido seleccionado. El esclavo seleccionado limpia su bit `MPCM0` de su registro `UCSR0A`, quedando disponible solo para recibir datos.

4. El maestro realiza el intercambio de datos con el esclavo seleccionado (9° bit en 0), el cual pasa desapercibido por los otros esclavos, porque aún tienen su bit `MPCM0` en alto.

5. Cuando el diálogo concluye, el esclavo seleccionado debe poner en alto su bit `MPCM0`, quedando como los demás esclavos, en espera de que el maestro solicite su atención.

Si no se utilizan todas las direcciones, las no empleadas pueden funcionar como comandos genéricos, para que el maestro envíe peticiones a todos los esclavos, como el reinicio de registros, activación o desactivación de salidas, etc.

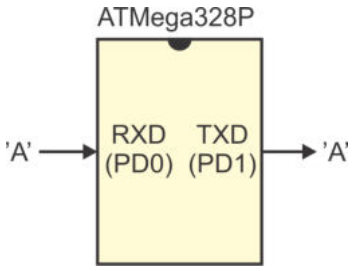


Figura 7.8: Sistema eco

7.1.5. Ejemplos de Uso del Módulo USART0

Se muestran dos ejemplos para ilustrar como utilizar al módulo USART0, suponiendo que el ATMega328P está trabajando con una frecuencia de 1 MHz. El primero se resuelve de dos maneras, por sondeo y por interrupción, mientras que en el segundo se combina el uso de la USART0 con la interrupción externa 0. Todos los programas se realizan en lenguaje C.

Ejemplo 7.1 - Sistema Eco
Escriba un programa "eco" que utilice al módulo USART0 de un ATMega328P para transmitir cada dato recibido. Configure para que la comunicación sea asíncrona a 9600 bps, con datos de 8 bits, sin paridad y con 1 bit de paro.

En la Figura 7.8 se muestra al ATMega328P para ubicar las terminales de recepción y transmisión, si el MCU se va a conectar a una computadora, será necesario emplear un adaptador USB-TTL.

En cuanto al software, en la primera versión de la solución se utilizará sondeo, después de configurar a la USART0, en el lazo infinito el programa quedará en espera del arribo de un dato en forma serial, cuando esto suceda lo transmitirá por el mismo medio. Antes de escribir un dato en el registro `UDR0`, el programa se asegura que el *buffer* serial está vacío. El programa con la solución es:

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

int main() {
uint8_t aux;

   DDRD = 0x02;          // RXD es entrada y TXD es salida

   // Configuración del módulo USART0
   UBRR0 = 12;           // Para 9600 bps
   UCSR0A = 0x02;        // Doble velocidad
   UCSR0B = 0x18;        // Habilita transmisor y receptor
   UCSR0C = 0x06;        // Datos de 8 bits, sin paridad, 1 bit de paro

   while(1) {
      while(!(UCSR0A & 1 << RXC0 ));        // Espera un dato serial
```

```
        aux = UDR0;                                  // Lee el dato
        while(!(UCSR0A & 1 << UDRE0 ));              // Asegura buffer vacío
        UDR0 = aux;                                  // Envía el dato
    }
}
```

Para habilitar la interrupción por recepción se debe poner en alto al bit `RXCIE0` del registro `UCSR0B` más la activación del habilitador global de interrupciones. Con ello, el programa principal queda ocioso ya que los datos serán recibidos por interrupción, el código con la solución es:

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

ISR(USART_RX_vect) {
uint8_t aux;

    aux = UDR0;                                  // Lee el dato
    while(!(UCSR0A & 1 << UDRE0 ));              // Asegura buffer vacío
    UDR0 = aux;                                  // Envía el dato
}

int  main() {

    DDRD = 0x02;          // RXD es entrada y TXD es salida

    UBRR0 = 12;           // Para 9600 bps
    UCSR0A = 0x02;        // Doble velocidad
    UCSR0B = 0x98;        // Habilita transmisor y receptor
                          // con interrupción por recepción
    UCSR0C = 0x06;        // Datos de 8 bits, sin paridad, 1 bit de paro

    sei();                // Habilitador global de interrupciones

    while(1)
        asm("NOP");
}
```

De las tres interrupciones debidas a la USART0, de manera práctica solo la interrupción por recepción llega a ser empleada, porque se desconoce en qué momento llegará un dato de manera serial. Una transmisión es una tarea agendada como parte de un programa, así que no es muy común emplear interrupciones por fin de transmisión o por registro de datos vacío, pero si se debe garantizar que el *buffer* transmisor está vacío antes de iniciar una nueva transmisión para no crear conflictos en el hardware.

Ejemplo 7.2 - Transmisión y Recepción Simultánea
Escriba un programa que envíe un carácter ASCII imprimible por el puerto serie, cada vez que se presiona un botón. Los caracteres ASCII imprimibles están en el rango de 0x20 a 0x7E. Además, en cualquier momento puede arribar serialmente un

dato y este debe mostrarse en el Puerto B. Configure para que la comunicación sea asíncrona a 4800 bps, con datos de 8 bits, sin paridad y con 1 bit de paro. En la Figura 7.9 se muestra el hardware requerido por el sistema.

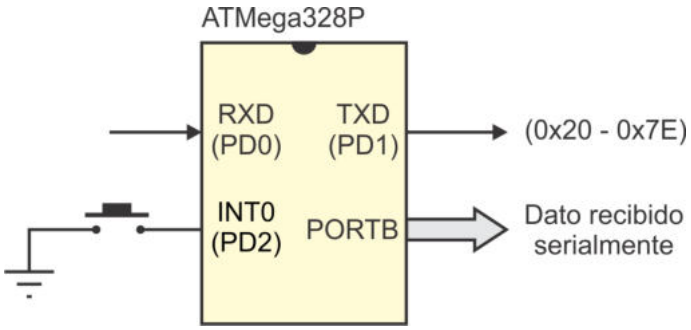


Figura 7.9: Sistema de transmisión y recepción serial

En el código con la solución del ejercicio se utilizan 2 interrupciones: la interrupción por recepción para ubicar el dato recibido en el Puerto B y la interrupción externa para enviar un carácter ASCII imprimible y preparar el siguiente. De acuerdo con la Tabla 7.2, para la velocidad de 4800 bps con el MCU operando a 1 MHz, se puede emplear el modo asíncrono normal o bien, asíncrono a doble velocidad, se utiliza la primera opción. El código con la solución en lenguaje C es:

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

uint8_t  dato = 0x20;                          // Primer dato a enviar

ISR(INT0_vect) {                               // Botón presionado

   while(!( UCSR0A & 1 << UDRE0 ));            // Asegura buffer vacío
   UDR0 = dato;                                // Envía el dato

   dato = (dato == 0x7E)? 0x20: dato+1;        // Prepara el siguiente
}

ISR(USART_RX_vect) {                           // Recibe un carácter

   PORTB = UDR0;                               // Lo ubica en el Puerto B
}

int main (){

   DDRD = 0x02;          // RXD e INT0 son entradas, TXD es salida
   PORTD = 0x04;         // Pull-up en INT0
   DDRB = 0xFF;          // Puerto B como salida

   EICRA = 0x02;         // INT0 por flanco de bajada
```

```
    EIMSK = 0x01;          // Habilita la INT0

    UBRR0  = 12;           // Para 4800
    UCSR0A = 0x00;         // Modo asíncrono normal
    UCSR0B = 0x98;         // Habilita transmisor y receptor
                           // con interrupción por recepción
    UCSR0C = 0x06;         // Datos de 8 bits, sin paridad, 1 bit de paro

    sei();                 // Habilitador global de interrupciones
    while(1)
        asm("NOP");
}
```

7.1.6. Envío y Recepción de Datos con Formato

Cuando se intercambia información de un MCU con una computadora, en la computadora generalmente se emplea un programa conocido como terminal o monitor serial. Existen diferentes aplicaciones con este propósito, como el programa Hercules, el programa Serial Port Monitor e incluso, el entorno de Arduino incluye una herramienta con esa función. Generalmente estas herramientas manejan la información en código ASCII, aunque algunas también permiten escribir los datos en binario o hexadecimal.

En esta sección se analizan los aspectos a considerar cuando la terminal serial recibe y transmite información en código ASCII, sobre todo si se trata de información numérica. Primero se considera el caso en que el ATMega328P transmite y la terminal recibe, si desde el MCU se va a enviar una variable y se utiliza la función:

```
void  envia_dato(uint8_t  x ) {      // Función que envía un dato

    while(!( UCSR0A & 1 << UDRE0 )); // Asegura buffer vacío
    UDR0 = x;                        // Envía la variable x
}
```

Aunque la función es correcta, en la terminal serial no se mostrará el valor numérico de la variable `x`, porque recibe un byte y no la secuencia ASCII que describe al número. Para obtener la secuencia ASCII de la variable `x`, esta debe separarse en unidades, decenas y centenas, y a cada dígito se le debe sumar 0x30 para obtener su código ASCII. Además, para una adecuada visualización en la terminal, primero se deben enviar las centenas, después las decenas y finalmente las unidades.

El operador módulo de la variable `x` con el número 10 devuelve las unidades, haciendo una división entre 10 y volviendo a aplicar el operador módulo se obtienen las decenas. Se aprecia un proceso iterativo que implica módulos y divisiones entre 10, pero los resultados parciales se deben almacenar en algún lugar para que, posteriormente, el envío se realice en el orden correcto.

Si se pudiese observar el valor del número, inmediatamente se podría determinar el dígito a enviar, es decir, en automático se desechan los dígitos menos significativos y se ubica al más significativo. Después de enviar al dígito más significativo, el proceso se repite con los dígitos restantes. Este es un razonamiento recursivo que se traduce a código en la función:

```
void envia_numero(uint16_t  num) {
uint8_t aux;

   if(num >= 10)
      envia_numero(num/10);          // Llamada recursiva

   aux = num % 10;
   while(!(UCSR0A & 1 << UDRE0));    // Asegura buffer vacío
   UDR0 = aux + 0x030;
}
```

Las llamadas recursivas se realizan hasta que se obtiene el dígito más significativo, luego se hace su envío. El operador módulo es necesario porque la variable `num` conserva su valor al regresar de las llamadas recursivas y, si ya se enviaron los dígitos más significativos, falta enviar a lo que en ese momento representa al menos significativo. La suma de 0x30 es para obtener el código ASCII de cada digito, de esta forma, el número se está enviando como una secuencia de caracteres ASCII. Posterior al envío del número se puede enviar un separador, como un retorno de carro (0x0D). Esto se ilustra en el siguiente código, en donde el número obtenido del Puerto B se envía a través de la USART:

```
uint8_t  x;                          // Se declara una variable

   x = PINB;                         // Se asigna un valor a x
   envia_numero(x);                  // Envía la secuencia ASCII de x
   while(!(UCSR0A & 1 << UDRE0));    // Asegura buffer vacío
   UDR0 = 0x0D;                      // Envía un retorno de carro
```

La otra situación se presenta cuando se transmite un número desde una terminal serial y se recibe en el ATMega328P, con el número representado como una secuencia de caracteres ASCII, terminada con cualquier carácter no numérico. El valor ASCII para los caracteres numéricos está entre los valores de 0x30 y 0x39.

Cuando un MCU hace una recepción serial, lo conveniente es que esta se configure por interrupción, tomando en cuenta que se va a generar un evento por cada byte o carácter recibido. El valor del número recibido se debe almacenar en una variable global, la variable debe iniciar con 0 y su valor se debe actualizar en cada evento de recepción. Si se recibe un carácter numérico, el valor actual de la variable debe multiplicarse por 10, porque el nuevo dígito recibido es el menos significativo, y al resultado del producto se le suma el valor numérico del carácter recibido, el cual se obtiene al restarle 0x30.

Cuando se recibe un carácter no numérico, significa que el número ya está disponible y, en la ISR de recepción se puede realizar su procesamiento o se puede activar una bandera para que el procesamiento se realice dentro del lazo infinito.

En la siguiente secuencia de código se muestra cómo se realiza la recepción de caracteres numéricos, para dejar su valor en una variable de 16 bits:

```
uint16_t  num = 0;
uint8_t   ban = 0;

ISR(USART_RX_vect){                          // Recepción serial
uint8_t  dato;

   dato = UDR0;
   if(dato >= 0x30 && dato <= 0x39)          // Carácter numérico
      num = (num*10) + (dato-0x30);
   else
      ban = 1;                               // Número listo
}
```

En los segmentos de código mostrados como ejemplos, se utilizaron variables de 16 bits, tanto en el envío como en la recepción de datos numéricos. El tamaño puede reducirse a 8 o ampliarse a 32 bits, dependiendo de la aplicación. Si se quieren transferir datos en punto flotante, como secuencias de caracteres ASCII, el proceso de transferencias de datos sería similar, aplicándolo primero a la parte entera y posteriormente a la parte fraccionaria.

7.1.7. El Módulo USART0 en Arduino

La USART0 es fundamental para las tarjetas Arduino porque es a través de este módulo por donde se reciben los nuevos programas de aplicación. En cuanto al hardware, además del microcontrolador principal, las tarjetas Arduino incluyen otro MCU que se encarga de hacer la conversión de USB a TTL, en el caso de la Arduino UNO revisión 3, se trata de un ATMega16U2, otro miembro de la familia AVR que contiene un controlador USB.

El IDE de Arduino cuenta con dos herramientas que facilitan la interacción con esta interfaz serial, la primera es un monitor serial desde donde se pueden enviar y recibir datos como secuencias de caracteres ASCII y la segunda es un graficador serial, el cual recibe números, también en formato ASCII, separados por un caracter de nueva línea. Si se requiere de dos o más gráficas en forma simultánea, los datos para cada gráfica deben separarse por comas, y con un carácter de nueva línea se pasa a la nueva secuencia de datos. El monitor y el graficador serial son herramientas que se pueden emplear de manera general, es decir, su uso no es exclusivo para las tarjetas Arduino, solo debe indicarse el puerto de comunicación y considerar el formato de los datos.

Para el manejo del módulo USART0, Arduino utiliza un objeto predefinido denominado `Serial`, el cual cuenta con 21 métodos que facilitan el envío y recepción de datos, los más empleados son:

- **begin():** Configura al puerto de comunicación serial. Puede recibir uno o dos argumentos: `speed` para establecer la velocidad de transmisión, utilizando velocidades típicas desde 300 hasta 115 200 bps; y `config` para definir el formato de la trama, incluyendo el tamaño de los datos, bit de paridad y bits de paro, hay constantes definidas para estos parámetros, por defecto se utiliza `SERIAL_8N1`, que significa datos de 8 bits, sin paridad y un bit de paro, las constantes se adecuan de acuerdo con los valores permitidos para cada parámetro.

- **print():** Envía el argumento recibido por medio del puerto serie como una secuencia de caracteres ASCII. El método identifica el tipo de argumento para hacer las conversiones necesarias, porque puede recibir un número entero, un número en punto flotante, caracteres individuales y cadenas. Para números enteros, el método puede recibir un segundo argumento con una de las constantes: BIN, OCT, DEC y HEX, para que la tarjeta Arduino envíe la secuencia ASCII con el valor en binario, octal, decimal y hexadecimal del primer argumento. Para números en punto flotante la función por defecto solo envía dos dígitos decimales, sin embargo, con un segundo argumento se puede cambiar la cantidad de decimales a enviar.

- **println():** Este método llama a `print()` con cualquiera de los posibles argumentos y después envía los caracteres de retorno de carro y de nueva línea (‘\r’ y ‘\n’).

- **available():** Con la inicialización del puerto serie también se habilita al receptor por interrupción, conforme se reciben los datos, se alojan en un *buffer* que tiene una capacidad de 128 bytes. Con el método `available()` se obtiene el número de bytes disponibles en el *buffer*, no recibe argumentos y regresa un entero, la lectura del *buffer* solo se puede realizar si hay datos disponibles.

- **read():** Lee un byte del *buffer* serial y lo regresa como un entero o regresa -1 si no hay datos en el *buffer*.

- **readBytes():** Lee una secuencia de bytes del *buffer* serial, como argumentos recibe un arreglo para ubicar los datos leídos y un entero con la cantidad de bytes por leer. El método termina cuando se han alcanzado los bytes requeridos o si se superó el tiempo de espera.

- **setTimeout():** Ante una petición de lectura y sin datos en el *buffer* serial, el software espera un periodo de tiempo inicialmente establecido en 1000 ms, con este método se ajusta el tiempo de espera a la cantidad de milisegundos recibida como argumento.

- **parseInt():** Busca y regresa un entero válido del *buffer* serial. Los caracteres que no son dígitos son ignorados.
- **parseFloat():** Busca y regresa el primer número en punto flotante válido del *buffer* serial. También ignora caracteres no relacionados.
- **write():** Transmite datos binarios sin formato por el puerto serie. Opcionalmente se puede utilizar un segundo argumento para especificar la cantidad de bytes a enviar.
- **flush():** Limpia el *buffer* de entrada de datos. Si hay un dato en proceso de transmisión espera a que concluya antes de realizar la limpieza.
- **end():** Deshabilita la comunicación serial, las terminales RXD y TXD vuelven a ser de propósito general.

En tarjetas como la Arduino Mega, que tiene 4 puertos seriales, además del objeto `Serial`, se han predefinido los objetos: `Serial1`, `Serial2` y `Serial3`, todos de la misma clase.

El amplio repertorio de métodos incluidos en el objeto `Serial` facilita el manejo del módulo USART0, como consecuencia, a un usuario de Arduino le resulta complicado implementar la comunicación serial utilizando sólo registros.

En el Ejemplo 7.3 se contrastan ambos estilos de comunicación, para el mismo problema se presenta una solución con las funciones de Arduino y otra que realiza un acceso directo a los registros. En la segunda solución se ejemplifica el envío y recepción de datos con formato, presentado en la sección anterior.

Ejemplo 7.3 - Contraste con Arduino
Suponiendo que el microcontrolador ATMega328P se va a comunicar con una computadora, a través de una terminal serial que maneja números como secuencias de caracteres ASCII, realice un programa que reciba dos enteros y regrese el resultado de la suma de los mismos. Suponga que el MCU está operando a 16 MHz y realice la comunicación a 9600 bps. Desarrolle el programa empleando las funciones de Arduino y con los registros del MCU.

Sin importar cómo se realizará la solución, debe contar con variables para ubicar los datos recibidos y una bandera que determine si se está recibiendo el primero o el segundo dato. La solución mediante un *sketch* de Arduino es:

```
int A, B, R;
bool ban = false;       // Bandera para saber qué dato se recibe

void setup() {
   Serial.begin(9600);
}

void loop() {
```

```
   if( Serial.available() ) {
      if( ban == false ) {
         A = Serial.parseInt();
         ban = true;
      }
      else {
         B = Serial.parseInt();
         Serial.print(A);
         Serial.print(" + ");
         Serial.print(B);
         Serial.print(" = ");
         R = A + B;
         Serial.println(R);
         ban = false;
      }
   }
}
```

Para la solución basada en el manejo de registros se debe tomar en cuenta que se va a generar una interrupción con cada carácter recibido, así que los números A y B se van a integrar conforme llegan los caracteres. Si se recibe un carácter que no es un dígito, significa que el envío del número ha concluido, una bandera ayuda a pasar de recibir un número a recibir el siguiente.

El código en C con el mismo comportamiento, pero sin el uso de funciones de Arduino es:

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

uint16_t  A = 0, B = 0, R;
uint8_t   ban = 0;

void envia_numero(uint16_t num);
void envia_cadena(char cad[]);
int main(void){

   DDRD = 0x02;          // RXD es entrada y TXD es salida
   UBRR0 = 103;          // Para 9600 bps con el MCU
                         // trabajando a 16 MHz
   UCSR0A = 0x00;        // Asíncrono normal
   UCSR0B = 0x98;        // Habilita transmisor y receptor
   UCSR0C = 0x06;        // Datos de 8 bits, sin paridad y
                         // 1 bit de paro
   sei();                // Habilitador global de interrupciones
   while (1)
      asm("NOP");
}

ISR(USART_RX_vect){
uint8_t  dato;
```

```
    dato = UDR0;
    if(ban == 0) {                          // Primer número
        if(dato >= 0x30 && dato <= 0x39)
            A = (A*10) + (dato-0x30);
        else
            ban = 1;
    } else {                                // Segundo número
        if(dato >= 0x30 && dato <= 0x39)
            B = (B*10) + (dato-0x30);
        else {
            R = A + B;                      // Suma los números
            envia_numero(A);
            envia_cadena(" + ");
            envia_numero(B);
            envia_cadena(" = ");
            envia_numero(R);
            while(!(UCSR0A & 1<<UDRE0));
            UDR0 = '\n';                    // Nueva línea
            while(!(UCSR0A & 1<<UDRE0));
            UDR0 = '\r';                    // Comienzo de línea
            A = B = ban = 0;
        }
    }
}

void envia_numero(uint16_t  num) {
uint8_t  aux;

    if(num >= 10)
        envia_numero(num/10);           // Llamada recursiva
    aux = num % 10;
    while(!(UCSR0A & 1 << UDRE0));      // Asegura buffer vacío
    UDR0 = aux + 0x030;
}

void envia_cadena(char cad[]) {
uint8_t  i = 0;

    while(cad[i] != '\0') {
        while(!(UCSR0A & 1 << UDRE0)); // Asegura buffer vacío
        UDR0 = cad[i];                  // Envía un carácter a la vez
        i++;
    }
}
```

Además de la función `envia_numero()`, se utilizó la función `envia_cadena()` para simplificar la estructura del programa, se trata de una función que envía una secuencia de caracteres hasta encontrar el carácter nulo. El envío de los caracteres de nueva línea y de comienzo de línea son necesarios al emplear la terminal serial de Arduino.

Comparando ambos programas, se observa la simplicidad que se tiene con el *sketch* de Arduino en contraste con la solución basada en registros. Sin embargo, Arduino introduce código y variables adicionales, y esto incrementa los requerimientos de memoria, en la Tabla 7.9 se compara la cantidad de memoria utilizada por ambas soluciones, se observa que el espacio utilizado por la solución basada en registros es mucha más pequeño que la solución con Arduino.

Tabla 7.9: Espacio de memoria utilizado por dos soluciones diferentes

	Flash	**SRAM**
Uso de Arduino	2346 bytes (7 %)	201 bytes (9 %)
Uso directo de registros	726 bytes (2.2 %)	15 bytes (0.7 %)

Es evidente que la ejecución de un programa con más instrucciones requiere de un periodo de tiempo mayor en comparación con un programa con menos instrucciones. Al ejecutar ambas soluciones en forma práctica se observó que la solución basada en registros tiene un mejor desempeño que la solución basada en Arduino, la mejora se aprecia, aunque es complicado cuantificarla.

7.2. 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 microcontroladores o entre un microcontrolador y dispositivos periféricos.

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 genera la señal de reloj y determina cuándo enviar los datos. 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 7.10 se muestra una conexión entre dos dispositivos vía SPI, esta interfaz no es propia de los microcontroladores, también puede encontrarse en memorias, sensores, ADCs, DACs, etc., estos dispositivos usualmente funcionan como esclavos. Puede verse que la base para la operación del protocolo SPI son dos registros de desplazamiento de 8 bits con sus correspondientes circuitos de control, conectados de manera tal que integran un registro circular de 16 bits.

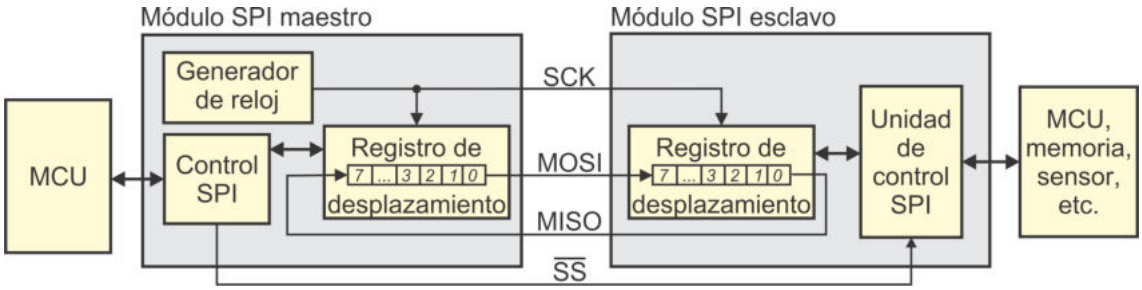


Figura 7.10: Conexión entre 2 dispositivos vía SPI

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 y es generada por el maestro.
- **SS** (*Slave Select*): Señal de selección del esclavo, el maestro debe habilitar al esclavo para realizar las transferencias de datos, la selección generalmente es activa en un nivel bajo. Esta señal es fundamental para el manejo de múltiples esclavos, la selección del maestro determina con qué esclavo realizará las transferencias.

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. Es evidente que el esclavo desconoce qué información le va a solicitar el maestro, por ello, en muchos sistemas el maestro primero envía una petición y como una respuesta inmediata recibe basura, luego, debe enviar cualquier otro carácter para poder obtener la respuesta esperada.

7.2.1. Organización de la Interfaz SPI en los AVR

Bajo el protocolo SPI, un microcontrolador AVR puede funcionar como maestro o esclavo, de acuerdo con los requerimientos de la aplicación. En la Figura 7.11 se muestra la organización del hardware para la interfaz SPI.

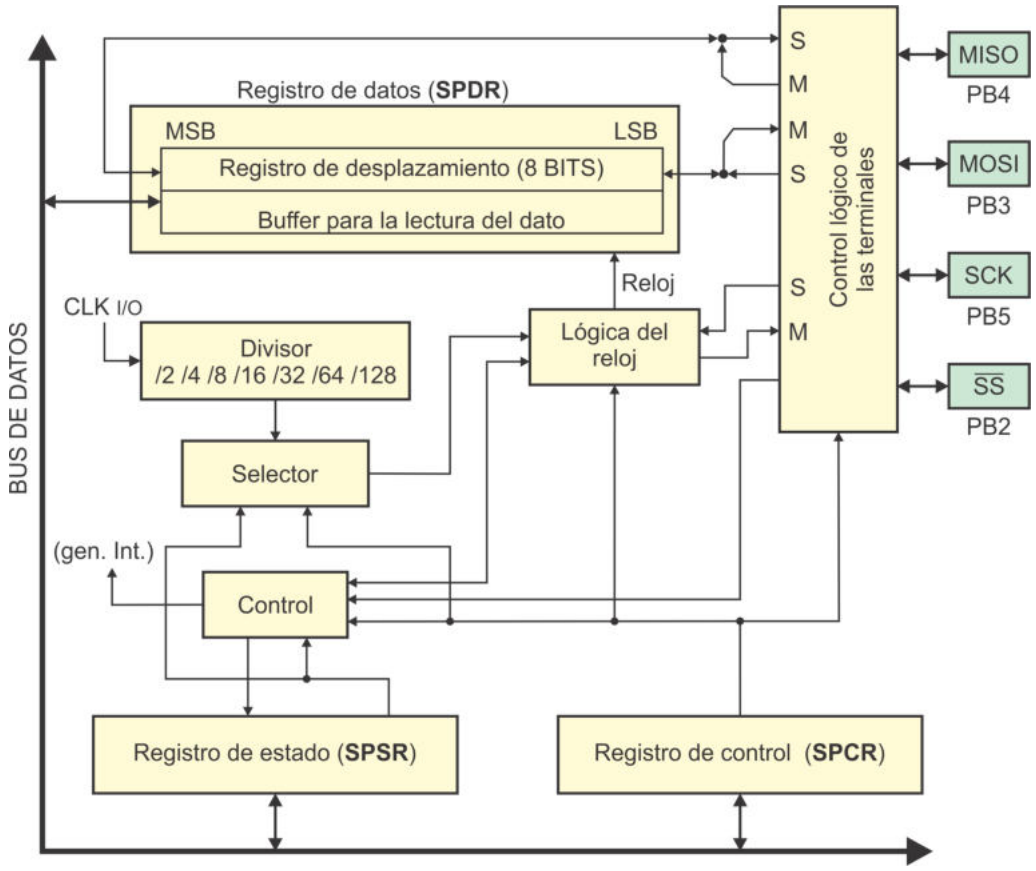


Figura 7.11: Organización de la interfaz SPI en los AVR

El recurso se habilita con la puesta en alto del bit `SPE` del registro `SPCR` (*SPI Control Register*), sin esta habilitación no es posible realizar alguna operación. El dispositivo por defecto va a operar como esclavo, la configuración como maestro se realiza con el bit `MSTR`, también en el registro `SPCR`.

Dependiendo del modo en el que un dispositivo va a funcionar, las líneas: MISO, MOSI, SCK y SS, se deben configurar como entradas o salidas, según corresponda, de esa forma, el módulo para el control lógico de las terminales podrá manejar el flujo requerido. Para cada modo se tienen las consideraciones siguientes:

- **Modo maestro:** la terminal MISO debe ser entrada y las terminales MOSI y SCK deben ser salidas. La terminal SS es de propósito general, típicamente debería ser salida y emplearse para habilitar a un esclavo, pero esta función no es exclusiva porque puede haber más de un esclavo y deberán utilizarse otras terminales como habilitadores. Si la terminal SS no es empleada por el maestro, la habilitación de un esclavo se debe hacer desde otra terminal.
- **Modo esclavo:** las terminales MOSI, SCK y SS deben ser entradas, mientras que la terminal MISO debe ser salida.

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 nuevo dato mientras un ciclo de envío-recepción está en progreso. En los AVR puede elegirse si se envía 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 puede producir la interrupción por transferencia serial completa vía SPI, la cual va a ocurrir si se habilitó 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 van a corromper los bits de la transferencia actual. La colisión generalmente es provocada por un esclavo, porque desconoce si el maestro inició una transferencia. El maestro es quien inicia las transferencias, por ello, no debería generar errores de colisión de escritura, no obstante, el hardware puede detectar estos errores tanto en el modo maestro como en el modo esclavo. El error 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`.

7.2.2. Modos de Transferencia por 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, estos se definen en la Tabla 7.10 y en la Figura 7.12 se muestran los diagramas de tiempo para distinguirlos.

Tabla 7.10: Modos de Transferencia por SPI

Modo	CPOL	CPHA	Descripción
0	0	0	Espera en bajo, muestrea en fase
1	0	1	Espera en bajo, muestrea con un retraso
2	1	0	Espera en alto, muestrea en fase
3	1	1	Espera en alto, muestrea con un retraso

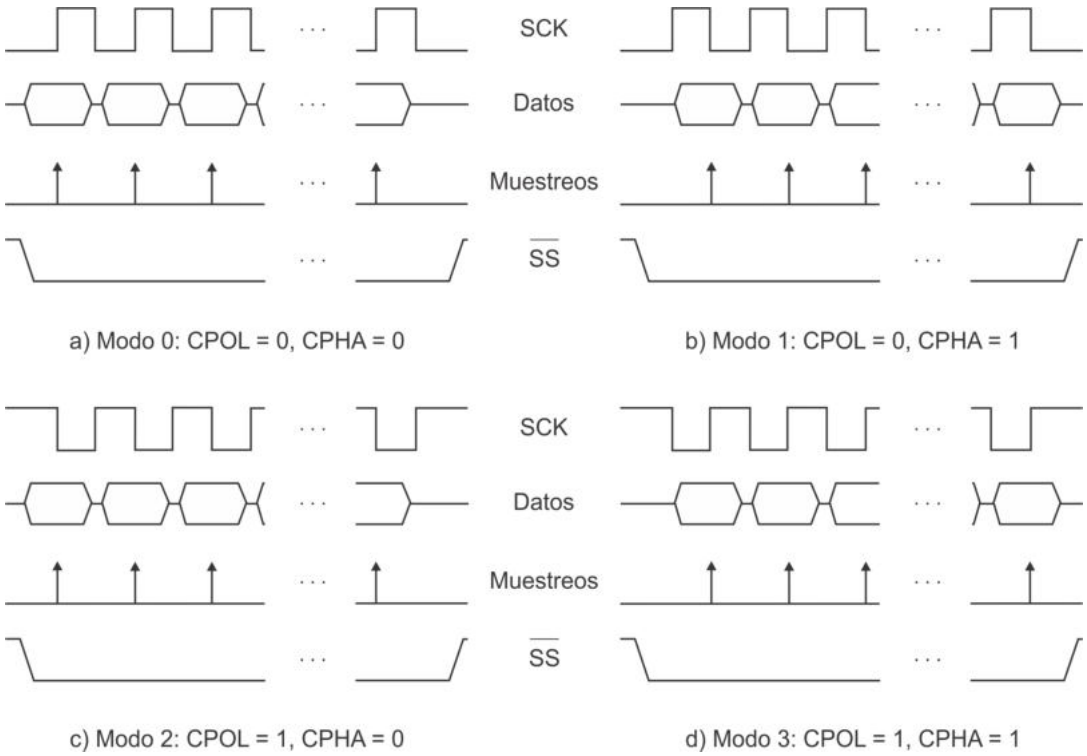


Figura 7.12: Modos de Transferencia por SPI

7.2.3. Funcionalidad de la Terminal SS

Si un microcontrolador AVR es configurado como Esclavo (bit `MSTR` del registro `SPCR` en bajo), la interfaz SPI se activa al existir un nivel bajo en la terminal SS. Cuando en la terminal se coloca un nivel alto, la interfaz SPI queda pasiva y no reconoce los datos de entrada. Si el nivel lógico de la terminal cambia y hay datos parcialmente recibidos en el registro de desplazamiento, estos van a ser desechados. Además, en la Figura 7.12 se puede ver como la terminal SS ayuda a mantener al esclavo sincronizado con el reloj del maestro.

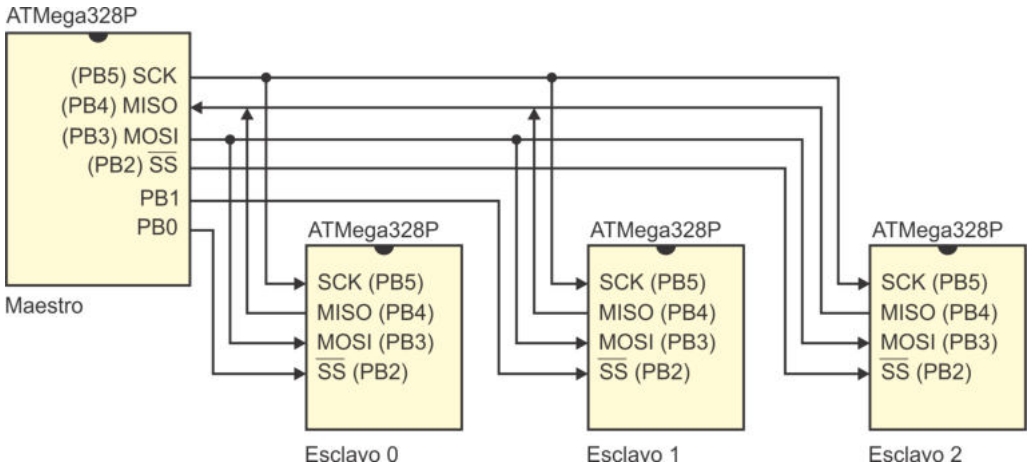


Figura 7.13: Conexión de un Maestro y 3 Esclavos por SPI

Si el AVR es configurado como maestro (bit `MSTR` del registro `SPCR` en alto), la dirección de la terminal SS es configurada por el usuario. Si la terminal es salida, será 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 la terminal SS se configura como entrada, un conflicto puede ocurrir por estar habilitada la interfaz SPI, la entrada debe recibir un nivel alto para asegurar la operación del MCU 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 por SPI 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, lo más simple es que la terminal SS se configure como salida, o en caso de ser entrada y exista la posibilidad de que la terminal SS sea llevada a un nivel bajo, en la ISR debe ajustarse el valor del bit `MSTR`, para mantener a la interfaz 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 7.13 se muestra la conexión de 4 microcontroladores ATMega328P, uno como maestro y 3 como esclavos.

Tabla 7.11: Registros para el manejo de la interfaz SPI

Registro	Dirección	Operación
SPDR	0x2E (0x4E)	Registro de datos
SPSR	0x2D (0x4D)	Registro de estado
SPCR	0x2C (0x4C)	Registro de control

Aunque los microcontroladores también pueden ser esclavos (ver Figura 7.13), en la práctica, un microcontrolador es el maestro y los esclavos suelen ser dispositivos con una tarea específica, como ADCs, DACs, memorias, etc.

7.2.4. Registros para el Manejo de la Interfaz SPI

La interfaz SPI es relativamente simple, solo son 3 los Registros I/O involucrados en su manejo, estos se describen en la Tabla 7.11, con su respectiva dirección.

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 para el control de la interfaz SPI es el `SPCR`, este registro es fundamental en el manejo de la interfaz, sus bits son:

REG.	7	6	5	4	3	2	1	0
SPCR	SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0

- **Bit 7 - SPIE:** (*SPI Interrupt Enable*) Es el 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, en la Tabla 7.12 se muestra el vector de la interrupción.
- **Bit 6 - SPE:** (*SPI Enable*) Habilita a la interfaz SPI, debe estar en alto para realizar transferencias por esta interfaz.
- **Bit 5 - DORD:** (*Data Order*) Determina el 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:** (*Master/Slave Select*) Un 1 en `MSTR` habilita a la interfaz SPI como maestro, un 0 la deja como esclavo.
- **Bit 3 - CPOL:** (*Clock Polarity*) Determina la polaridad del reloj (SCK) cuando la interfaz SPI está inactiva. En la Figura 7.12 se mostró el efecto de este bit en la señal SCK y en la Tabla 7.10 se puede ver como este bit, junto con el bit `CPHA`, definen los modos de transferencia por SPI.

- **Bit 2 - CPHA:** (*Clock Phase*) 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 7.12 se ilustró el efecto de este bit en el muestreo de datos y en la Tabla 7.10 se puede ver como este bit, junto con el bit `CPOL`, definen los modos de transferencia por SPI.
- **Bits [1:0] - SPR[1:0]:** (*SPI Clock Rate Select*) 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 7.13 se muestran las diferentes frecuencias obtenidas a partir del reloj del sistema. También se observa el efecto del bit `SPI2X` (ubicado en el registro `SPSR`), con el que se duplica la frecuencia de la señal SCK.

Tabla 7.12: Vector de interrupción para la interfaz SPI

Dirección	Evento	Descripción
0x022	SPI_STC	Transferencia serial completa por SPI

Tabla 7.13: Frecuencia de la señal de reloj (SCK)

SPI2X	SPR1	SPR0	Frecuencia de SCK
0	0	0	$CLK_{I/O}/4$
0	0	1	$CLK_{I/O}/16$
0	1	0	$CLK_{I/O}/64$
0	1	1	$CLK_{I/O}/128$
1	0	0	$CLK_{I/O}/2$
1	0	1	$CLK_{I/O}/8$
1	1	0	$CLK_{I/O}/32$
1	1	1	$CLK_{I/O}/64$

El registro de estado es el `SPSR`, solo 3 de los 8 bits están implementados, estos son:

REG.	7	6	5	4	3	2	1	0
SPSR	SPIF	WCOL	-	-	-	-	-	SPI2X

- **Bit 7 - SPIF:** (*SPI Interrupt Flag*) Es la bandera de fin de transferencia por 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:** (*Write COLlision Flag*) Es la 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 [5:1]:** No están implementados.

- **Bit 0 - SPI2X:** (*Double SPI Speed Bit*) Duplica la velocidad de transmisión, en la Tabla 7.13 se observa su efecto, duplicando la frecuencia de la señal SCK, definida por los bits `SPR1` y `SPR0`.

7.2.5. Ejemplos de Uso de la Interfaz SPI

El uso de la interfaz SPI se ilustra a través de 3 ejemplos, el primero tiene como propósito mostrar cómo enlazar dos microcontroladores por SPI, uno como maestro y otro como esclavo. En el segundo se expone un sistema compuesto por 4 microcontroladores, uno es el maestro y los otros tres son esclavos. En el último ejemplo se exhibe cómo manejar un módulo SPI, el MCU es maestro y un DAC funciona como esclavo, utilizando la misma lógica se puede tener acceso a otros módulos SPI. Solo se presentan las soluciones en lenguaje C.

Ejemplo 7.4 - Comunicación por SPI
Sin emplear interrupciones, configure un ATMega328P 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, configure para que las transmisiones se realicen a 125 kHz y utilice el modo de transferencia 1 para una adecuada sincronización entre ambos microcontroladores.

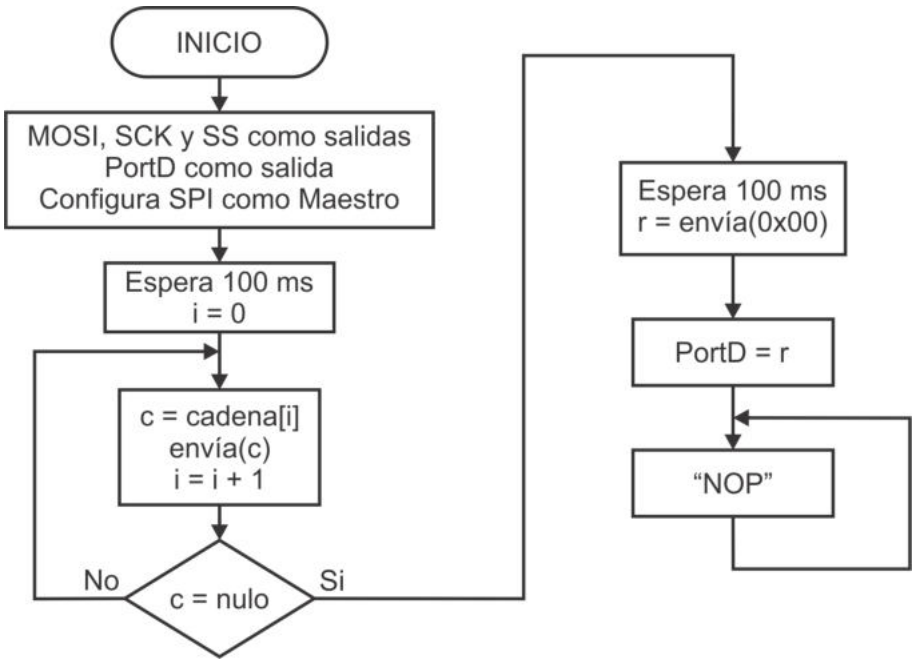


Figura 7.14: Comportamiento del maestro

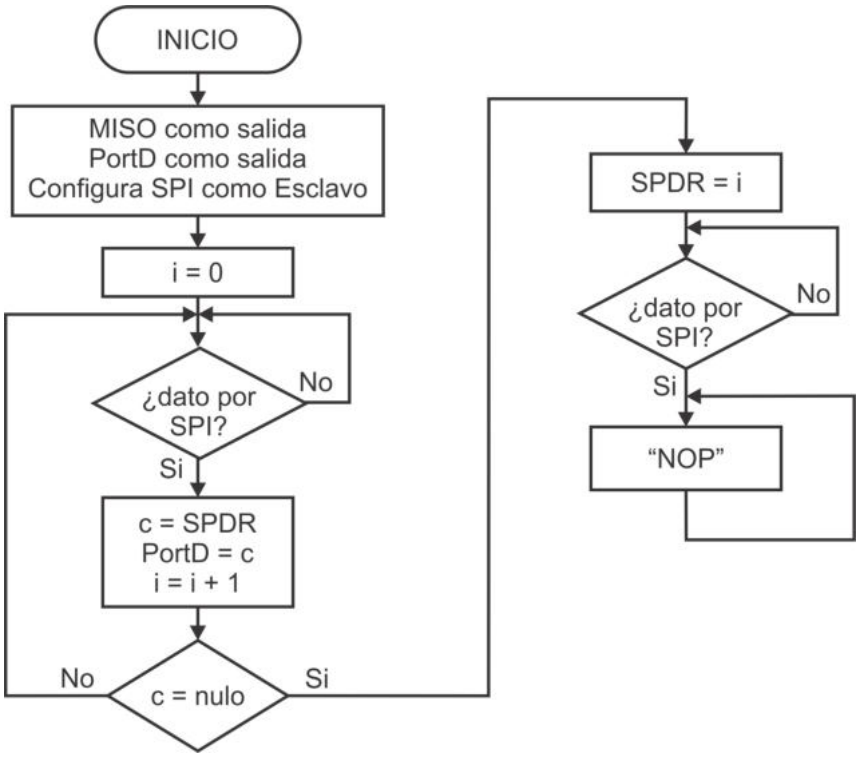


Figura 7.15: Comportamiento del esclavo

Se asume que ambos dispositivos se energizan al mismo tiempo, aun con ello, el maestro espera 100 ms para asegurar que el esclavo está listo para recibir los caracteres de la cadena. 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. Puesto que no se emplean interrupciones, el comportamiento del maestro se puede describir con el diagrama de flujo mostrado en la Figura 7.14.

La función que envía un dato por SPI regresa el dato recibido, sin embargo, la respuesta no es importante mientras se envían los caracteres, por lo que esta se ignora en las llamadas realizadas dentro del ciclo repetitivo. Por el contrario, al solicitar el número de caracteres solo la respuesta es importante, en esta última etapa del programa podría enviarse cualquier carácter.

El esclavo tiene un comportamiento diferente, el cual se ilustra en la Figura 7.15. 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.

El código en lenguaje C para el maestro, considerando una cadena constante, es:

```
#define   F_CPU 1000000UL          // El MCU trabaja a 1 MHz
#include  <util/delay.h>           // Biblioteca para los retardos
#include  <avr/io.h>               // Definiciones de Registros I/O
```

```
uint8_t envia_SPI(uint8_t dato);  // Prototipo de la función

int main() {                      // Programa principal
uint8_t i, r;
char c, cadena[] = "cadena de prueba";

   DDRB = 0b00101100;             // MOSI, SCK y SS como salidas
   DDRD = 0xFF;                   // Puerto D como salida
   PORTB = 0x04;                  // SS en alto, esclavo inactivo
   SPCR = 0x55;                   // SPI como maestro en modo 1
   SPSR = 0x01;                   // A 125 kHz

   _delay_ms(100);                // Espera que el esclavo esté listo
   i = 0;

   do {                           // Envía caracteres hasta encontrar
      c = cadena[i];              // al carácter nulo
      envia_SPI(c);
      i = i + 1;
   }while( c != 0x00);

   _delay_ms(100);                // Espera respuesta del esclavo
   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 y deshabilitación del esclavo. */

uint8_t envia_SPI(uint8_t dato) {
uint8_t 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 deshabilitado

   return resp;                   // Regresa la respuesta
}
```

El código para el esclavo es:

```
#include <avr/io.h>               // Definiciones de Registros I/O

int main() {                      // Programa principal
uint8_t  i;
char  c;

   DDRB = 0b00010000;             // MISO como salida
```

```
    DDRD = 0xFF;                        // Puerto D como salida
    SPCR = 0x44;                        // SPI como esclavo en modo 1
    SPSR = 0x00;                        // No se configura la velocidad
    i = 0;
    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 para limpiar la bandera
    while(1)                            // Lazo infinito
        asm("nop");
}
```

En la solución del ejemplo anterior, se observa que si se utilizan interrupciones no se optimiza el programa del maestro, porque el maestro define en qué momento se realiza la transferencia de un dato. Por el contrario, el esclavo se beneficiaría porque puede estar desarrollando otras actividades en lugar de realizar un sondeo continuo.

Ejemplo 7.5 - Sistema Maestro-Esclavos
Tomando como base el diagrama de la Figura 7.13, al maestro colóquele 2 arreglos de interruptores (con 8 y 2 interruptores) y un botón, y a los esclavos conécteles 8 LEDs. El objetivo es expandor las salidas de un sistema, al enviar datos de 8 bits del maestro a los diferentes esclavos para su exhibición. El arreglo de 8 interruptores es para introducir el dato y el de 2 para seleccionar a uno de los esclavos. Con ello, cada vez que el botón es presionado, el maestro debe enviar el dato al esclavo seleccionado. Cuando un esclavo reciba un dato, lo debe mostrar en los LEDs. De las 4 combinaciones que se consiguen con los 2 interruptores, utilice la combinación 3 para mandar el dato a todos los esclavos, en una especie de difusión.

El hardware completo se muestra en la Figura 7.16, puesto que la interfaz SPI utiliza las terminales del Puerto B, en el maestro se ha utilizado el Puerto D para recibir el dato y, el Puerto C para la selección del esclavo y el botón para enviar. En los esclavos se ha utilizado al Puerto D para conectar a los LEDs.

En cuanto al software para el maestro, el botón será monitoreado con la interrupción por cambios en PC2, en su ISR se obtendrá el número de esclavo y el dato a enviar. La función `envia_SPI()` debe acondicionarse para que, ademaás de recibir el dato, también reciba el número de esclavo a quien será enviado. Se deberán hacer 3 envíos cuando se seleccione al esclavo 3 para una difusión, porque el *buffer* circular de 16 bits imposibilita que un maestro envíe un dato a más de un esclavo. Por último, dado que no se han establecido condiciones para la transferencia, se utilizará el modo 0 y la velocidad máxima permitida.

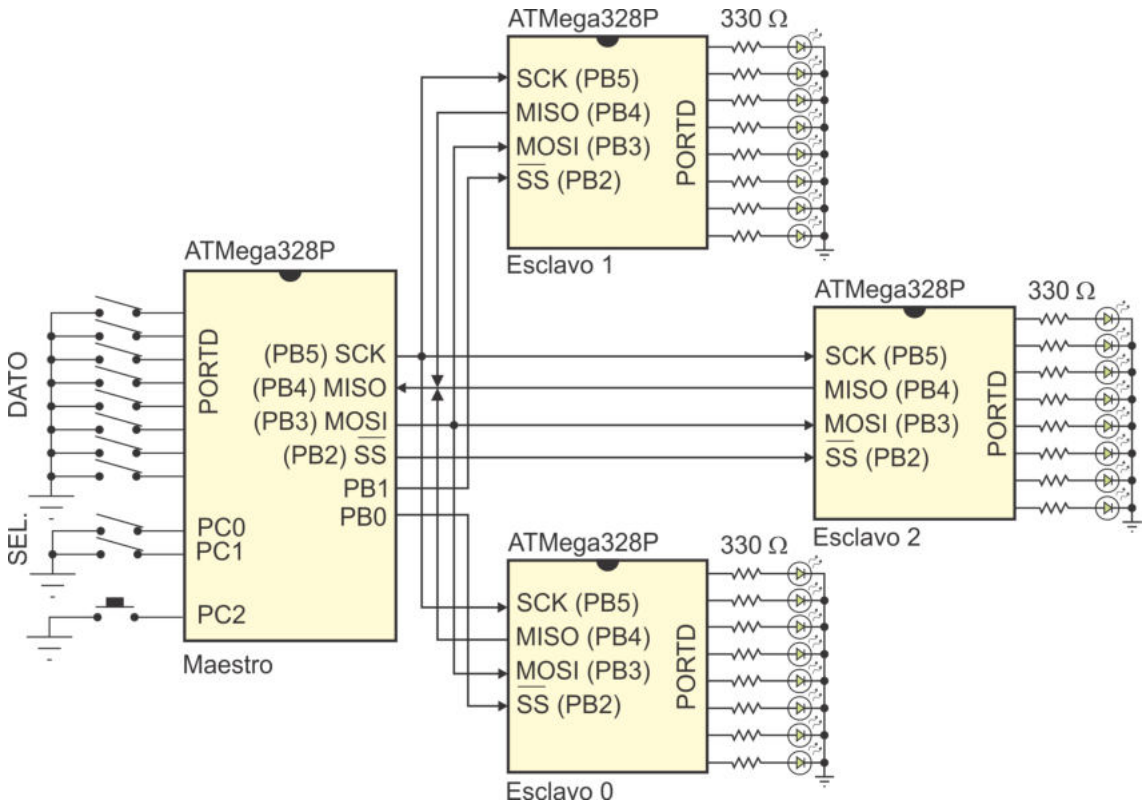


Figura 7.16: Envío de información de un maestro a 3 esclavos por SPI

El programa en lenguaje C para el maestro es:

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

uint8_t envia_SPI(uint8_t dato, uint8_t esvo);

ISR(PCINT1_vect) {
uint8_t  sel, dato;

   if( !(PINC & 0x04) ) {          // Botón presionado
      sel = PINC & 0x03;
      dato = PIND;

      if(sel < 3)                  // Esclavos 0, 1 o 2
         envia_SPI(dato, sel);     // Envía el dato
      else {                       // Difusión
         envia_SPI(dato, 0);       // Esclavo 0
         envia_SPI(dato, 1);       // Esclavo 1
         envia_SPI(dato, 2);       // Esclavo 2
         }
   }
}
```

```
int  main() {            // Programa principal

   DDRD = 0x00;          // Entrada de datos
   DDRB = 0x2F;          // MOSI, SCK y SS(s) como salidas
   DDRC = 0x00;          // Puerto C como entrada
   PORTD = 0xFF;         // Pull-up
   PORTC = 0x07;
   PORTB = 0x07;         // Esclavos deshabilitados

   SPCR = 0x50;          // Habilita la interfaz SPI como maestro
   SPSR = 0x01;          // Ajustando para 500 KHz
   PCMSK1 = 0x04;        // Interrupción por cambios en PC2
   PCICR  = 0x02;        // Habilita la interrupción

   sei();                // Habilitador global de interrupciones
   while(1)              // Lazo infinito
      asm("nop");
}

uint8_t envia_SPI(uint8_t dato, uint8_t esvo) {
uint8_t aux;

   aux = 1 << esvo;
   PORTB &= ~aux;                  // Habilita al esclavo
   SPDR = dato;                    // Inicia el envío
   while(!(SPSR & 1 << SPIF));     // Espera fin de envío
   PORTB |= aux;                   // Deshabilita al esclavo

   return  SPDR;                   // Regresa la respuesta
}
```

En la función `envia_SPI()` se realiza la habilitación y deshabilitación del esclavo, esta acción se basa en desplazamientos y trabaja sin problema para los esclavos 0, 1 y 2; aunque también podría funcionar en esclavos habilitados con las terminales PB6 y PB7, puesto que estas terminales quedan libres en el Puerto B. Otro aspecto es que la función regresa el valor leído en el registro `SPDR`, el valor de retorno no se utiliza en este problema pero debe dejarse porque la bandera `SPIF` se limpia con el acceso al registro `SPDR`.

Para el esclavo es conveniente esperar la recepción del dato por interrupción en lugar de sondear para saber si hay actividad en la interfaz y no es necesario configurar la velocidad de operación porque la determina el maestro. Basta con habilitar a la interfaz con su respectiva interrupción, el programa para el esclavo es:

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

ISR(SPI_STC_vect){     // Fin de transferencia por SPI
   PORTD = SPDR;       // Lee el dato recibido
}
```

```
int  main() {

   DDRB = 0x10;          // MISO como salida
   DDRD = 0xFF;          // Puerto D como salida

   SPCR = 0xC0;          // Habilita la interfaz SPI con interrupción
   SPSR = 0x00;

   sei();                // Habilitador global de interrupciones
   while(1){             // Lazo infinito
      asm("nop");
   }
}
```

El circuito integrado MCP4821 es un convertidor digital-analógico de 12 bits que es manejado por una interfaz SPI con una velocidad de hasta 20 MHz, su voltaje de operación está en el rango de 2.7 V a 5.5 V, por lo que se puede conectar sin problema con un microcontrolador ATMega328P. El DAC tiene una referencia de voltaje interna de 2.048 V y por medio de un bit de configuración puede duplicar el valor del voltaje de salida. La interfaz SPI del MCP4821 trabaja en el modo 0 y recibe los bits iniciando con el más significativo.

El MCP4821 recibe una trama compuesta por dos bytes, primero se le debe enviar al byte alto e inmediatamente después al byte bajo. La trama contiene el valor digital a convertir y dos bits de configuración, su formato es:

0	-	GA'	SHDN'	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
bit 15								bit 0							

El bit `GA'` es para definir una ganancia, es activo en bajo, un 1 lógico en `GA'` genera un voltaje máximo en la salida de 2.048 V (voltaje de referencia) y un 0 lógico duplica este valor, siendo el valor máximo para la salida de 4.096 V.

El bit `SHDN'` es para apagar al DAC, también es activo en bajo, por lo que un 1 lógico en `SHDN'` lo mantendrá en operación. En el siguiente ejemplo se muestra como manipular a un CI MCP4821 desde un MCU ATMega328P.

Ejemplo 7.6 - Manejo de un DAC por SPI
En la Figura 7.17 se muestra la conexión de un ATMega328P con un DAC MCP4821, así como un potenciómetro y un interruptor. Para evaluar el manejo del DAC, realice un programa que digitalice la entrada analógica y utilice el valor digital para enviarlo al DAC a través de la interfaz SPI. El estado del interruptor debe determinar el valor del bit GA' para establecer la ganancia.

En el programa con la solución se utilizará sondeo para esperar el fin de conversión del ADC, de esta manera, una vez que se obtiene el valor digital de la entrada

analógica, se procede con las transferencias via SPI. Además, el ADC es de 10 bits y el DAC es de 12 bits, por ello, en las 2 posiciones menos significativas de la trama a enviar, siempre se colocarán 1's para alcanzar el valor máximo de la salida.

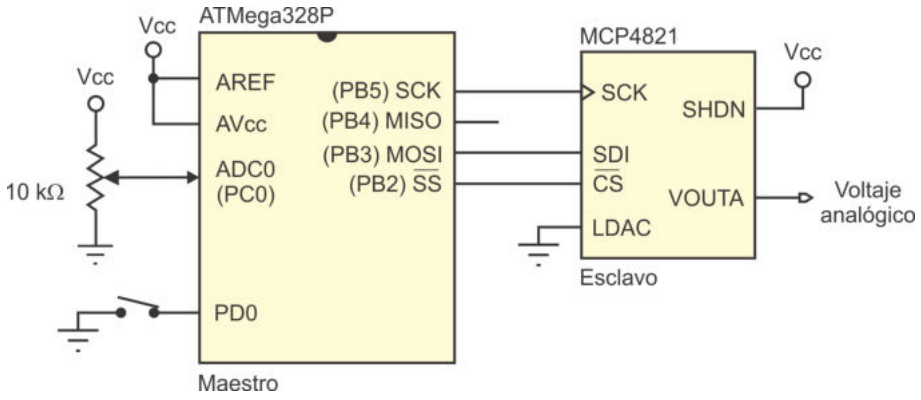


Figura 7.17: Conexión del ATMega328P con el DAC MCP4821

El programa con la solución del ejercicio es:

```
#define F_CPU 1000000UL

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

int main(){
uint16_t  aux;
uint8_t   high, low, extra;

   DDRD = 0x00;          // Entrada para el interruptor
   PORTD = 0x01;         // Pull-up para el interruptor
   DDRC = 0x00;          // Entrada para el ADC
   DDRB  = 0x2C;         // MOSI, SCK y SS como salidas
   PORTB = 0x04;         // SS en alto, esclavo deshabilitado

   SPCR = 0x50;          // Habilita la interfaz SPI como Maestro
   SPSR = 0x01;          // Ajustando para 500 KHz

   DIDR0 = 0x01;         // Anula el buffer digital en ADC0
   ADMUX = 0x00;         // Selecciona ADC0 y Vref en AREF
   ADCSRA = 0xC3;        // Habilita al ADC, inicia conversión
                         // y divide entre 8
   while(1) {
      while(!(ADCSRA & 1 << ADIF ));  // Espera fin de conversión
      ADCSRA |= 1 << ADIF;            // Limpia la bandera ADIF
      aux = ADCW;                     // Obtiene el valor analógico
      aux = aux << 2;                 // Deja los dos bits menos
      aux = aux | 0x0003;             // significativos en alto
      low = aux;
```

```
        high  = aux >> 8;
        high |= 0x10;                    // DAC encendido
        if(PIND & 0x01)
            high |= 0x20;                // Bit de ganancia en alto

        PORTB &= 0xFB;                   // Habilita al esclavo
        SPDR = high;                     // Envía el byte alto
        while(!(SPSR & 1 << SPIF));      // Espera fin de envío
        extra = SPDR;                    // Para limpiar la bandera SPIF
        SPDR = low;                      // Envía el byte bajo
        while(!(SPSR & 1 << SPIF));      // Espera fin de envío
        extra = SPDR;                    // Para limpiar la bandera SPIF
        PORTB |= 0x04;                   // Deshabilita al esclavo
        _delay_ms(100);
        ADCSRA |= 1 << ADSC;             // Inicia nueva conversión
    }
}
```

En este ejemplo no se utiliza la función `envia_SPI()` porque el DAC debe mantenerse habilitado (con un nivel bajo) mientras se realizan las dos transferencias. El uso de la variable `extra` para la lectura del registro `SPDR` es necesario para limpiar la bandera `SPIF` que indica el fin de la transferencia.

7.2.6. La Interfaz SPI en Arduino

Para utilizar la interfaz SPI en un *sketch* de Arduino se debe incluir la biblioteca `SPI.h`, la biblioteca incluye un objeto predefinido denominado `SPI`, con el que se manipula a la interfaz, así como una clase con el nombre `SPISettings` para establecer los parámetros de la comunicación. Con la clase `SPISettings` se define un objeto que es utilizado por el método `SPI.beginTransaction` antes de iniciar cualquier operación.

Por ejemplo, con la sentencia: `SPISettings parametros(16000000, MSBFIRST, SPI_MODE0);` se crea el objeto `parametros`, en el que se indica que la velocidad máxima del MCU es de 16 MHz, que se va a transferir primero al bit más significativo y que se utilizará el modo de transferencia 0. Este objeto sirve de argumento para la llamada del método `SPI.beginTransaction`.

El objeto `SPI` tiene 9 métodos, de los cuales, los más importantes son:

- **begin():** Configura las terminales MOSI, SCK y SS como salidas.
- **beginTransaction():** Establece los parámetros de la comunicación, como argumento recibe un objeto del tipo `SPISettings`.
- **transfer():** Para realizar la transferencia, como argumento recibe el byte a enviar y regresa la respuesta del esclavo.

- **endTransaction():** Detiene las transferencias por SPI, es necesario para que otras funciones utilicen la interfaz o si se van a modificar los parámetros de las transferencias.
- **end():** Deshabilita la interfaz SPI, sin cambios en el modo de las terminales.

En el siguiente *sketch* se muestra la solución al Ejemplo 7.6 (manejo de un DAC), para ilustrar el uso de las funciones:

```
#include <SPI.h>

uint16_t  aux;
uint8_t   high, low, extra;

const int boton = 0;                  // El botón se ubica en PD0
const int ss = 10;                    // La terminal SS está en PB2

SPISettings parametros(16000000, MSBFIRST, SPI_MODE0);

void setup() {

  pinMode(boton, INPUT_PULLUP);
  pinMode(ss, OUTPUT);
  SPI.begin();
  SPI.beginTransaction(parametros);
}

void loop() {
   aux = analogRead(A0);
   aux = aux << 2;                    // Deja los dos bits menos
   aux = aux | 0x0003;                // significativos en alto
   low = aux;
   high  = aux >> 8;
   high |= 0x10;                      // DAC encendido

   if(digitalRead(boton))
      high |= 0x20;                   // Bit de ganancia en alto
   digitalWrite(ss, LOW);             // Habilita al esclavo
   extra = SPI.transfer(high);        // Envía el byte alto
   extra = SPI.transfer(low);         // Envía el byte bajo
   digitalWrite(ss, HIGH);            // Deshabilita al esclavo
}
```

Se observa en el *sketch* que Arduino no ofrece muchas ventajas al manipular la interfaz con la biblioteca `SPI.h`, el *sketch* sigue los mismos pasos que el código C basado en registros. A diferencia de otros recursos, el programador debe comprender a la interfaz para poder utilizarla; además, las funciones están enfocadas a la operación del MCU como maestro, sin considerar el modo esclavo.

Sin embargo, algunos desarrolladores de *shields* o tarjetas de expansión para Arduino muchas veces también comparten una biblioteca de funciones para su manejo, de

esta forma, el uso de los *shields* se simplifica y los mecanismos para el acceso a la interfaz SPI pasan desapercibidos por el usuario. Algunos ejemplos son:

- **Matriz de LED de 8x8:** este *shield* es operado por el chip MAX7219, un controlador compacto con interfaz serial compatible con SPI y 16 salidas para manejar hasta 8 displays de 7 segmentos o 64 LEDs individuales. El MAX7219 contiene una RAM estática de 8x8 para almacenar la información a mostrar. La biblioteca `LedControlMS.h` facilita el manejo del *shield*, incluyendo funciones básicas para el encendido de LEDs individuales, renglones o columnas, así como la impresión de caracteres alfanuméricos. Estos módulos se pueden conectar en cascada para crear matrices con dimensiones mayores y con las funciones de la biblioteca se especifica a cual de ellos se enviará la información.
- **Sensor BMP183:** es un sensor ambiental de temperatura y presión barométrica con interfaz SPI, es una solución de bajo costo para medir la presión barométrica con una precisión absoluta de $\pm$ 1 hPa y la temperatura con una precisión de $\pm$ 1.0 $^{\circ}C$. Debido a que la presión cambia con la altitud, también se puede usar como un altímetro con una precisión de $\pm$ 1 m. La biblioteca `Adafruit_BMP183.h` incluye funciones mediante las cuales se lee la presión, la temperatura y la altitud.
- **Memoria SD o micro SD:** estos *shields* proporcionan un espacio de almacenamiento no volátil casi ilimitado para la mayoría de proyectos basados en Arduino. La biblioteca `SD.h` incluye las funciones básicas para leer y escribir archivos, crear o eliminar directorios, así como buscar información en archivos. Su desventaja es que impone una importante carga de trabajo para Arduino, el programa ocupará cerca del 40 % de la memoria Flash y casi el 50 % de la memoria SRAM en una tarjeta Arduino UNO.

Estos ejemplos demuestran la versatilidad de la interfaz SPI, entender el funcionamiento al nivel de registros ayuda a comprender las bibliotecas desarrolladas por terceros y a utilizar de manera eficiente las funciones que se requieran.

7.3. La USART en Modo SPI Maestro

La USART se puede configurar para operar como una interfaz SPI maestro (MSPIM, *Master SPI Mode*), este modo se consigue con la combinación "11" en los bits `UMSEL0[1:0]` del registro `UCSR0C`. Los recursos de la USART0 que se emplean son:

- *Buffers* y registros de desplazamiento del transmisor y receptor.
- Generador de velocidad de transmisión.
- Lógica de reloj.

La lógica de transmisión y recepción de la USART es remplazada por una lógica

de transferencia SPI. La funcionalidad de las terminales y de algunos bits en los registros de control se ajustan al entrar en el MSPIM. La correspondencia en las terminales se muestra en la Tabla 7.14, la terminal SS no se hace corresponder con una terminal específica porque en el modo maestro para la interfaz SPI, cualquier terminal puede ser empleada para habilitar un esclavo.

La señal de reloj se genera en la terminal XCK y la frecuencia de transmisión se determina con el apoyo del registro `UBRR0`, con la misma relación empleada en el modo síncrono maestro de la USART. En la Tabla 7.15 se muestra la expresión para obtener la velocidad de transmisión a partir del registro `UBRR0`, así como la forma de obtener el valor para el registro `UBRR0` en función de la velocidad de transmisión. Con `UBRR0` en 0 se obtiene la máxima velocidad de transmisión para el MSPIM, que es la mitad de la frecuencia del oscilador.

Tabla 7.14: Terminales de la USART en el modo MSPIM

USART MSPIM	**Interfaz SPI**	**Descripción**
TXD	MOSI	Salida del maestro
RXD	MISO	Entrada para el maestro
XCK	SCK	Señal de reloj generada por el maestro
-	SS	No aplica

Tabla 7.15: Cálculo de la velocidad de transmisión y del valor para UBRR0

Modo	**Velocidad de transmisión**	**Valor para UBRR0**
Síncrono como maestro	$Baud\_Rate = \frac{f_{OSC}}{2(UBRR0+1)}$	$UBRR0 = \frac{f_{OSC}}{2 \times Baud\_Rate} - 1$

El registro `UDR0` es el *buffer* para la transmisión y recepción de datos, solo que al emplear los recursos de la USART, la interfaz MSPIM tiene espacios independientes para estas operaciones, a diferencia de la interfaz SPI normal, donde un registro en el maestro se enlaza con un registro del esclavo, para crear un registro circular de 16 bits.

7.3.1. Adaptación de los Registros de Control

Cuando la USART0 opera como una interfaz MSPIM, algunos bits en los registros de control tienen una funcionalidad diferente y otros quedan sin uso.

En el registro `UCSR0A` solo se utilizan los 3 bits más significativos, estos son:

REG.	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
UCSR0A	RXC0	TXC0	UDRE0	-	-	-	-	-

- **Bit 7 - RXC0:** (*USART Receive Complete*) Se pone en alto si hay un dato disponible en el *buffer* receptor. Puede generar una interrupción, se limpia al leer al *buffer* de entrada (`UDR0`). Si la recepción no se habilita los datos entrantes son eliminados.

- **Bit 6 - TXC0:** (*USART Transmit Complete*) Indica que el dato a enviar ha sido completamente desplazado y, por lo tanto, el *buffer* de salida está vacío. Puede generar una interrupción, con su atención se limpia automáticamente. También puede limpiarse si se reescribe un 1 lógico en `TXC0`.
- **Bit 5 - UDRE0:** (*USART Data Register Empty*) Indica que el registro `UDR0` 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 `UDR0`.
- **Bits [4:0]:** No son utilizados en el modo MSPIM.

En cuanto al registro `UCSR0B`, los 5 bits que se utilizan son:

REG.	7	6	5	4	3	2	1	0
UCSR0B	RXCIE0	TXCIE0	UDRIE0	RXEN0	TXEN0	-	-	-

- **Bit 7 - RXCIE0:** (*RX Complete Interrupt Enable*) Habilita la interrupción por recepción completa.
- **Bit 6 - TXCIE0:** (*TX Complete Interrupt Enable*) Habilita la interrupción por transmisión completa.
- **Bit 5 - UDRIE0:** (*USART Data Register Empty Interrupt Enable*) Habilita la interrupción por *buffer* vacío.
- **Bit 4 - RXEN0:** (*Receiver Enable*) Habilita la recepción de datos.
- **Bit 3 - TXEN0:** (*Transmitter Enable*) Habilita la transmisión de datos.
- **Bits [2:0]:** No son utilizados en el modo MSPIM.

La USART0 puede generar 3 interrupciones, sin embargo, en la interfaz SPI los 3 eventos son simultáneos, cuando el maestro concluye la transmisión recibe la respuesta del esclavo y el *buffer* de transmisión queda vacío. De manera que solo debería habilitarse una de las interrupciones. Con respecto a la activación de los recursos, el transmisor siempre debe ser habilitado y el receptor es opcional, debe habilitarse únicamente si el maestro espera respuesta del esclavo.

Finalmente, para el registro `UCSR0C`, con los 2 bits más significativos se determina el MSPIM y los 3 menos significativos ajustan su comportamiento al modo de operación, sus bits quedan de la siguiente manera:

REG.	7	6	5	4	3	2	1	0
UCSR0C	UMSEL01	UMSEL00	-	-	-	UDORD0	UCPHA0	UCPOL0

- **Bits [7:6] - UMSEL0[1:0]:** (*USART Mode Select*) Con estos bits se selecciona el modo de operación de la USART0, deben tener “11” para el modo SPI maestro (MPSIM).

- **Bits [5:3]:** No son utilizados en el modo MSPIM.
- **Bit 2 - UDORD0:** (*Data Order*) Determina el orden de los datos, con un 0 en `UDORD0`, se transfiere primero al bit más significativo (MSB). Con un 1 se transfiere primero al bit menos significativo (LSB).
- **Bit 1 - UCPHA0:** (*Clock Phase*) Determina si los datos son muestreados en fase con el reloj o si se inserta un retardo inicial de medio ciclo de reloj. Junto con el bit `UCPOL0` definen los modos de transferencia por SPI, descritos en la Tabla 7.10. El efecto de este bit en el muestreo de datos es el mismo que el del bit `CPHA` del registro `SPCR`, mostrado en la Figura 7.12.
- **Bit 0 - UCPOL0:** (*Clock Polarity*) Determina la polaridad del reloj (XCK) cuando la interfaz SPI está inactiva. Junto con el bit `UCPHA0` definen los modos de transferencia por SPI, descritos en la Tabla 7.10. El efecto de este bit en la señal XCK es el mismo que el del bit `CPOL` del registro `SPCR`, mostrado en la Figura 7.12, para la salida SCK.

7.3.2. Uso de la USART en el Modo SPI Maestro

En el siguiente ejemplo se manipula nuevamente al DAC MCP4821, dispositivo utilizado en el Ejemplo 7.6, pero ahora el DAC es manejado con la USART operando en el modo SPI Maestro.

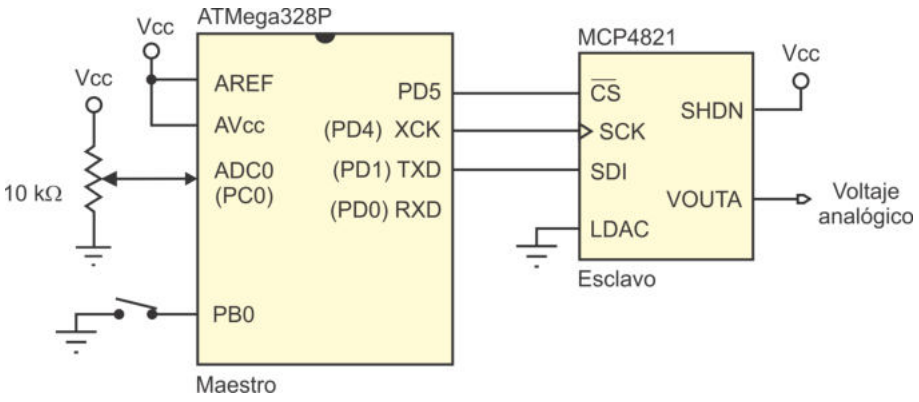


Figura 7.18: Conexión del ATMega328P con el DAC MCP4821 a través de la USART

Ejemplo 7.7 - Manejo de un DAC con la USART en MSPIM
El hardware de la Figura 7.17 se ha adecuado para ubicar al DAC MCP4821 en el puerto de la USART, porque va a trabajar en el modo SPI maestro, el resultado se muestra en la Figura 7.18. La interfaz SPI ahora está en el Puerto D, empleando las terminales como se indica en la Tabla 7.14 y utilizando a PD5 como el habilitador del esclavo. El interruptor que determina el valor del bit GA' debe moverse porque

PD0 ahora tiene la funcionalidad de MISO, aunque en este ejemplo el MCU no va a recibir respuesta, la terminal ya no es de propósito general, el interruptor se mueve a la terminal PB0.

Se mantiene el estilo de la solución mostrada para el Ejemplo 7.6 (utilizando sondeo para esperar el fin de conversión del ADC), prácticamente solo se realiza la adecuación para las transferencias vía USART en modo SPI maestro.

El programa con la solución del ejercicio es:

```
#define F_CPU 1000000UL

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

int main(){
uint16_t  aux;
uint8_t   high,  low;

   DDRB  = 0x00;          // Entrada para el interruptor
   PORTB = 0x01;          // Pull-up para el interruptor
   DDRD  = 0x32;          // PD1 (TXD), PD4 (XCK) y PD5 (SS) como salidas
   PORTD = 0x20;          // SS en alto, esclavo deshabilitado
   DDRC = 0x00;           // Entrada para el ADC

   UCSR0C = 0xC0;         // MSPIM, en modo 0
   UCSR0B = 0x08;         // Solo habilita al transmisor
   UBRR0 = 0x00;          // Para 500 KHz

   DIDR0 = 0x01;          // Anula el buffer digital en ADC0
   ADMUX = 0x00;          // Selecciona ADC0 y Vref en AREF
   ADCSRA = 0xC3;         // Habilita al ADC, inicia conversión
                          // y divide entre 8

   while(1) {
      while( !(ADCSRA & 1 << ADIF ) );  // Espera fin de conversión
      ADCSRA |= 1 << ADIF;              // Limpia la bandera ADIF
      aux = ADCW;                       // Obtiene el valor analógico
      aux = aux << 2;                   // Deja los dos bits menos
      aux = aux | 0x0003;               // significativos en alto
      low = aux;
      high  = aux >> 8;

      high |= 0x10;                     // DAC encendido
      if(PINB & 0x01)
         high |= 0x20;                  // Bit de ganancia en alto

      PORTD &= 0xDF;                    // Habilita al esclavo
      UDR0 = high;                      // Envía el byte alto
      while(!(UCSR0A & 1 << TXC0));     // Espera fin de envío
      UCSR0A |= 1 << TXC0;              // Limpia bandera TXC0
      UDR0 = low;                       // Envía el byte bajo
```

```
        while(!(UCSR0A & 1 << TXC0));        // Espera fin de envío
        UCSR0A |= 1 << TXC0;                 // Limpia bandera TXC0
        PORTD |= 0x20;                       // Deshabilita al esclavo

        _delay_ms(100);
        ADCSRA |= 1 << ADSC;                 // Inicia nueva conversión
    }
}
```

Una vez que se comprende el funcionamiento de la interfaz SPI, se observa que no es complicado emplear a la USART con ese propósito. De esta forma, el ATMega328P puede ser utilizado con dos interfaces SPI, lo cual no tiene mucha ventaja porque la interfaz SPI por sí misma puede manejar un esquema maestro-esclavos. Esta característica de contar con 2 interfaces SPI solo es conveniente si se van a manipular varios módulos externos y uno de ellos requiere una atención continua por lo que no puede compartir el bus con los demás.

7.4. Ejercicios

En los siguientes ejercicios suponga que el MCU estará operando a 1 MHZ, todos pueden resolverse en lenguaje C.

1. Realice un programa para el ATMega328P que, a través de su puerto serie, reciba una cadena de caracteres terminada con el carácter '$', cuente los caracteres antes del carácter '$' y regrese la longitud de la cadena como una secuencia de caracteres ASCII a una velocidad de 4800 bps.

2. Realice un programa para el ATMega328P que, a través de su puerto serie, reciba una cadena con el siguiente formato: "Num1 Num2 Op", donde Num1 y Num2 son enteros y Op puede ser +, - y *. El programa hará la operación artimética indicada y enviará el resultado por el mismo medio, por ejemplo, ante la cadena "25 12 -", el programa enviará "13" como respuesta.

3. Desarrolle un programa para el ATMega328P que cada segundo lea un parámetro analógico (en la entrada ADC0) y lo envíe por el puerto serial. Cuando el parámetro analógico es digitalizado genera un número entre 0 y 1023, este debe ser enviado como una cadena de caracteres ASCII finalizada con un retorno de carro (carácter 0x0D) a una velocidad de 9600 bps. Se sugiere utilizar al temporizador 1 para manejar el periodo de un segundo y coordinar los envíos. En la Figura 7.19 se muestra el hardware requerido para este problema, el LED ubicado en PB0 debe encenderse desde que inicia una conversión y apagarse cuando finaliza el envío, este elemento indica que hay actividad en el sistema y puede ayudar a medir el tiempo requerido por el MCU para la digitalización y transmisión de datos.

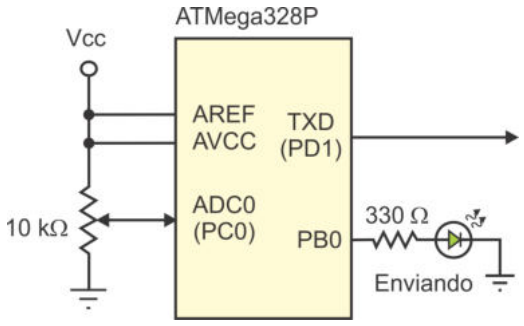


Figura 7.19: Envío de un parámetro analógico a través de la USART

4. Modifique el programa del ejercicio anterior pero ahora se hará uso de la recepción serial, de manera que el MCU ya no va a estar leyendo el parámetro analógico cada segundo, sino que debe iniciar la lectura cuando reciba al carácter 'D'. Al finalizar la conversión, el resultado se enviará en el mismo formato y condiciones de transmisión que en el ejercicio anterior. El LED ubicado en PB0 debe encenderse desde que se recibe el carácter 'D' y apagarse cuando finaliza el envío.

5. Genere una señal PWM con una frecuencia de 25 Hz y un ciclo de trabajo del 50 %. El ciclo de trabajo se modificará cuando se reciba una cadena de uno a tres caracteres ASCII finalizada con el carácter de retorno de carro (0x0D), indicando su nuevo porcentaje. Por ejemplo, con "0" + 0x0D se solicita el 0 %, "50" + 0x0D para el 50 % y "100" + 0x0D para 100 %. Si la cadena representa un número mayor a 100 deberá ignorarse. Configure la USART para una comunicación a 4800 Baudios, datos de 8 bits, un bit de paro y sin paridad.

6. Realice un programa para manipular al DAC MCP4821, utilizado en el Ejemplo 7.6, pero ahora desde la USART se obtendrá el valor digital. El DAC es de 12 bits, sin embargo, puede considerarse de 13 al disponer de un bit para duplicar el voltaje analógico resultante. Por ello, desde la USART se podrá recibir un número entre 0 y 8 191 ($2^{13} - 1$), como una secuencia de caracteres ASCII terminada con el carácter de retorno de carro (0x0D). Recibido el número y ubicado en una variable de 16 bits, deberá evaluarse su magnitud para determinar el valor del bit para la ganancia (GA') y organizar los dos bytes que serán enviados al DAC vía SPI. El bit de ganancia no se debe activar (se mantiene en alto) si el número es menor a 4 095 ($2^{12} - 1$). El número será ignorado si tiene una magnitud mayor que 8 191.

7. Diseñe un sistema para apoyar un estudio de la temperatura ambiente, acondicionando un sensor para que tome muestras con una resolución de 8 bits. Cada 10 minutos se almacenará una muestra en un arreglo con una capacidad de 600 datos. El sistema deberá tener un puerto serie configurado para una

comunicación asíncrona de 8 bits, sin paridad, a 9 600 bps y con 1 bit de paro. Por medio de la USART podrán recibir comandos en cualquier momento, a los cuales deberá dar respuesta. Los comandos son:

- Con 'N' el sistema enviará el número de datos almacenados en el arreglo, por ser una variable de 16 bits, iniciará con el byte menos significativo.
- Con 'D' el sistema enviará los datos, uno a uno.
- Con 'C' el sistema eliminará los datos, limpiando el arreglo y el contador de datos.
- Con 'R' el sistema reiniciará la cuenta de los 10 minutos, antes de tomar el siguiente dato.
- Ante comandos no identificados, el sistema responderá con 0xFF.

8. Se requiere manipular un servomotor a distancia a través de la USART, en la Figura 7.20 se muestra el hardware para ello, el MCU (1) debe obtener el valor digital de la entrada analógica cada 300 ms y enviar la información por el puerto serie, puesto que el resultado de la conversión ocupa 2 bytes, se enviará primero al menos significativo. El MCU (2) va a recibir la información por interrupción, después de recibir el segundo byte integrará al entero (entre 0 y 1023) y lo acondicionará para generar la salida PWM para el manejo del servomotor, debe tener un periodo de 20 ms y un ancho de pulso entre 0.9 ms y 2.1 ms, para posiciones entre 0° y 180°.

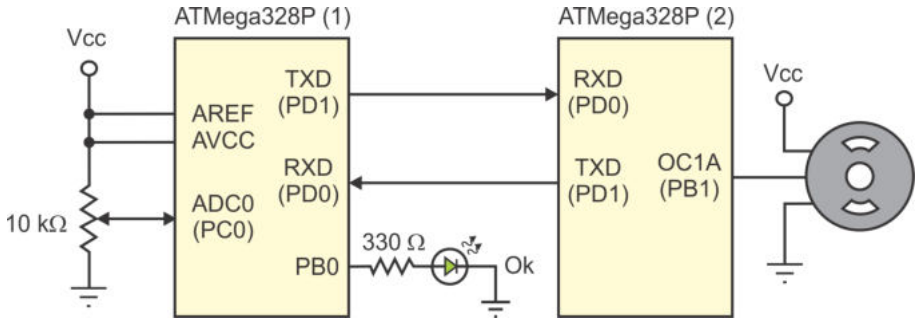


Figura 7.20: Manejo de un servomotor a distancia a través de la USART

Ante cada par de bytes recibidos, el MCU (2) responderá con 0x55 para que el MCU (1) sepa que el receptor está presente. El LED conectado en PB0 debe encenderse mientras el MCU (1) obtenga una respuesta del MCU (2), en caso contrario, debe apagarse. Se sugiere emplear el periodo de 300 ms como tiempo de respuesta, si llegado el momento de enviar se detecta que no hubo respuesta ante el envío previo, el LED se apaga para conocimiento del usurario, pero los envíos deben seguir porque el receptor se podrá reconectar en cualquier momento.

9. Repita el ejercicio anterior pero empleando la interfaz SPI, en la Figura 7.21 se muestra el acondicionamiento del hardware, la interfaz de comunicación ahora ocupa el Puerto B y el LED se ha movido a PD0. Por el registro circular de 16 bits que se forma entre el maestro y el esclavo, después de enviar el primer byte deje un intervalo de 5 ms, para que el esclavo pueda colocar 0x55 y responder cuando el maestro envíe el segundo byte. Utilice el modo de transferencia 1 para una adecuada sincronización entre ambos microcontroladores.

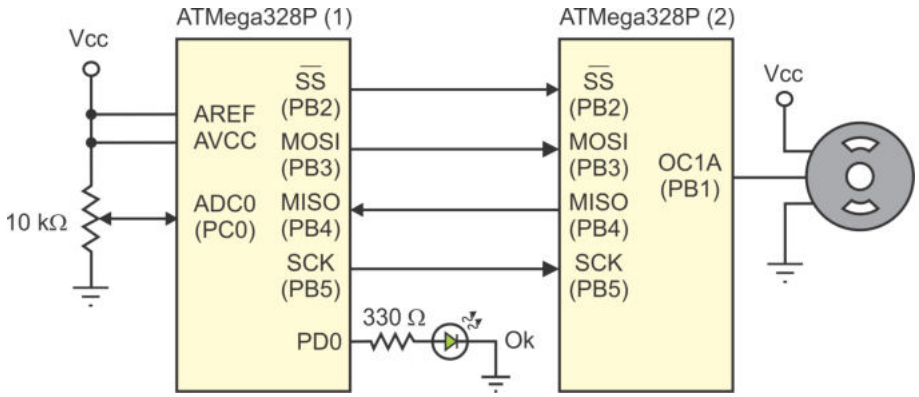


Figura 7.21: Manejo de un servomotor a distancia por la interfaz SPI

10. El circuito integrado MCP6S21 es un amplificador con ganancia programable (PGA, *Programmable Gain Amplifier*) manejado a través de la interfaz SPI. El circuito puede operar en los modos 0 y 3 a una frecuencia máxima de 10 MHz. En la Figura 7.22 se muestra la conexión del MCP6S21 con el ATMega328P. Realice un programa para el MCU que envíe la cadena "0100 0000 0000 0PPP" cada que haya un cambio en las terminales PD[2:0] (de estas terminales se obtiene el valor para los bits "PPP"). El primer byte es la instrucción para el PGA y el segundo indica el factor de ganancia, que de acuerdo con la combinación puede ser de 1, 2, 4, 5, 8, 10, 16 y 32, respectivamente. Configure la interfaz SPI en el modo 0 trabajando a 500 kHz.

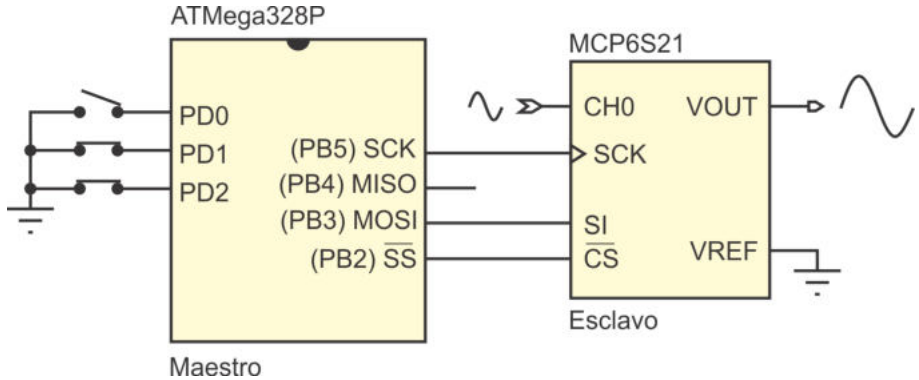


Figura 7.22: Manejo de un amplificador con ganancia programable vía SPI

Capítulo 8

Comunicación Serial (Parte II)

8.1. Introducción

Este capítulo está dedicado al estudio de la interfaz TWI (*Two-Wire Serial Interface*), es una interfaz serial disponible en microcontroladores y algunos periféricos, para que se comuniquen a través de un bus bidireccional de 2 líneas, una para reloj (SCL) y otra para datos (SDA). En la Figura 8.1 se muestra la forma en que los diferentes dispositivos se conectan, como hardware externo únicamente se requiere de 2 resistores conectados a Vcc (*pull-up*). La interfaz TWI permite a los desarrolladores de sistemas embebidos conectar hasta 127 dispositivos diferentes, cada uno con su propia dirección.

Esta interfaz es compatible en su operación con el bus I^2C (*inter-integrated circuits*), el cual es un estándar desarrollado por *Philips Semiconductors* (ahora *NXP Semiconductors*). 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.

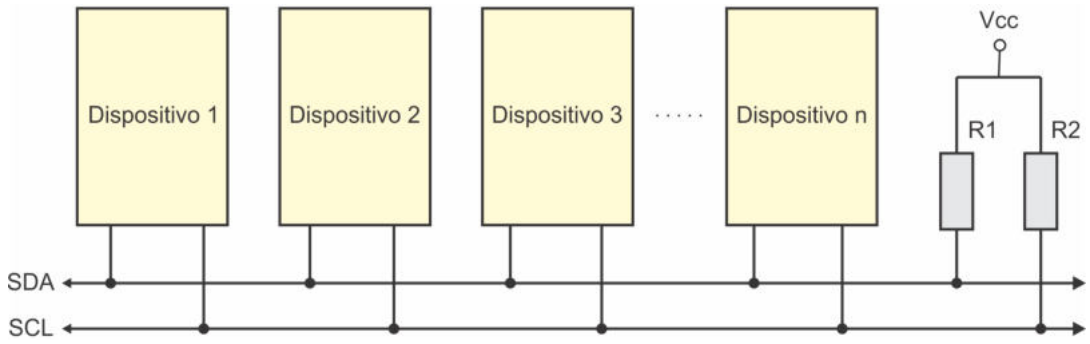


Figura 8.1: Interconexión de dispositivos a través de la interfaz serial de dos hilos

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. Los resistores 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-Esclavos, 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 en la línea SCL.
- **Esclavo:** Dispositivo direccionado por un maestro.
- **Transmisor:** Dispositivo que coloca los datos en el bus.
- **Receptor:** Dispositivo que lee los datos del bus.

La transmisión de datos no es exclusiva para el maestro, de hecho, los términos se pueden combinar en: maestro transmisor, maestro receptor, esclavo transmisor o esclavo receptor. Si en una aplicación se tiene un maestro transmisor es porque existe un esclavo receptor, o bien, si hay un maestro receptor se debe a la presencia de un esclavo transmisor.

Una característica muy interesante de este protocolo es que se pueden tener sistemas con múltiples maestros y, en tiempo de ejecución, un maestro puede ser direccionado por otro para pasar a ser un esclavo. Aunque en un instante de tiempo, solo un maestro puede coordinar las transferencias de datos, hasta que ese maestro libere al bus, otro maestro podrá utilizarlo para enviar o recibir datos.

Los mecanismos para las transferencias de datos, sincronización, organización y manejo de la interfaz, se describen a detalle en las siguientes secciones.

8.2. 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 está en alto, como se muestra en la Figura 8.2(a). Las excepciones a esta regla se dan cuando se están generando las condiciones de INICIO y PARO.

El maestro inicia y termina la trasmisiíon de datos, por lo tanto, es el maestro quien genera las condiciones de INICIO y PARO, estas condiciones se señalizan cambiando el nivel en la línea SDA cuando SCL está en alto, como se puede ver en la Figura 8.2(b). Una condición de INICIO se especifica pasando la señal SDA de un nivel

alto a un nivel bajo y una condición de PARO ocurre cuando SDA pasa de un nivel bajo a un nivel alto. 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 8.2(b).

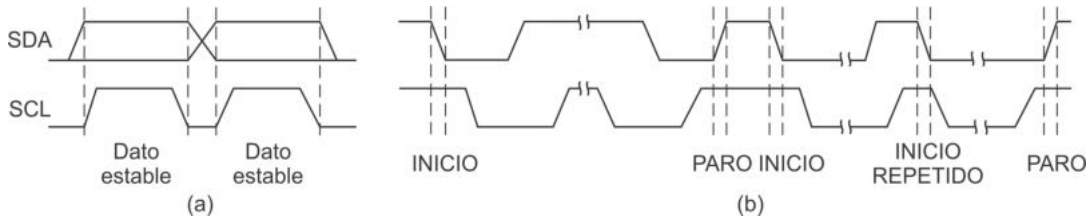


Figura 8.2: (a) Formato para datos válidos, y (b) condiciones de INICIO, PARO e INICIO REPETIDO

8.2.1. Formato de los Paquetes de Dirección

Cualquier dispositivo que se conecte en un bus TWI debe contar con una dirección de 7 bits para que pueda ser direccionado como esclavo por algún maestro, las direcciones deben ser diferentes entre sí y no puede ser "000 0000", esa dirección se reserva para llamadas generales y se abrevia como GCA (*General Call Address*), por eso es que en el bus pueden coexistir hasta 127 dispositivos.

Los paquetes de dirección tienen una longitud de 9 bits, de los cuales, 7 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 el maestro realizará una lectura (R/W' = 1) o una escritura (R/W' = 0). El bit de reconocimiento es generado por el esclavo para indicar su presencia al maestro, colocando un 0 en la señal SDA durante el noveno ciclo de la señal SCL, esta respuesta es 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 y a esta respuesta se le refiere como nACK. Los resistores de *pull-up* son fundamentales para la operación del bus, hacen que el maestro obtenga un nACK cuando intente direccionar a un esclavo que no está presente. Ante un nACK, el maestro puede transmitir una condición de PARO o un INICIO REPETIDO, para intentar direccionar al mismo esclavo o a uno diferente. En la Figura 8.3 se muestra el formato de un paquete de dirección.

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. Si una SLA+R se realiza con reconocimiento (ACK), el maestro está listo para recibir datos del esclavo, y si fue una SLA+W, el maestro podrá enviar datos al esclavo. Las llamadas generales

(dirección "000 0000" o GCA) solo pueden ser peticiones de escritura (SLA+W), para que el maestro envíe el mismo mensaje a todos los esclavos. Una petición de lectura en una llamada general provocaría una colisión en el bus, porque los esclavos podrían transmitir datos diferentes.

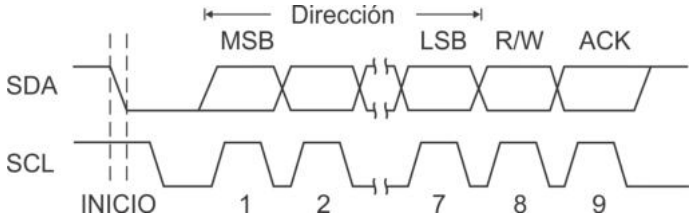


Figura 8.3: Formato de un paquete de dirección

8.2.2. 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 pero no siempre va a colocar los datos, si previamente hubo una SLA+W, será el maestro quien coloque los datos, pero si fue una SLA+R le tocará al esclavo colocar los datos. En cualquier caso, el transmisor coloca los datos, sincronizando bit a bit con el reloj del maestro y, en el noveno ciclo, el receptor pone un nivel bajo en SDA para indicar un reconocimiento. En la Figura 8.4 se muestra el formato de un paquete de datos.

Después de una SLA+W, el maestro es el transmisor e irá colocando datos en el bus mientras reciba reconocimiento del esclavo. Cuando el maestro va a terminar con las transferencias, simplemente establece una condición de PARO.

Después de una SLA+R, el esclavo es el transmisor e irá colocando datos en el bus, el maestro le dará reconocimiento mientras quiera recibir datos. Cuando el maestro va a terminar con las transferencias, recibe al último dato sin reconocimiento y posteriormente debe establecer una condición de PARO. Con el nACK, el esclavo interpreta que no debe continuar enviando datos.

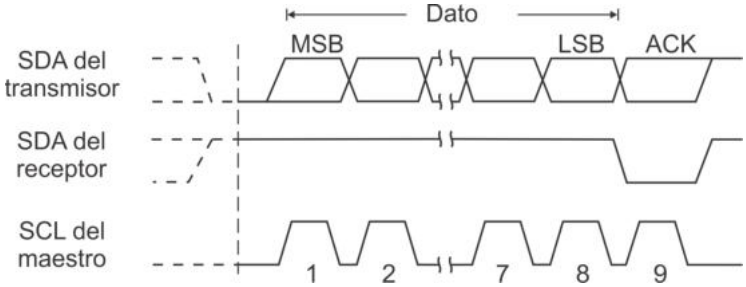


Figura 8.4: Formato de un paquete de datos

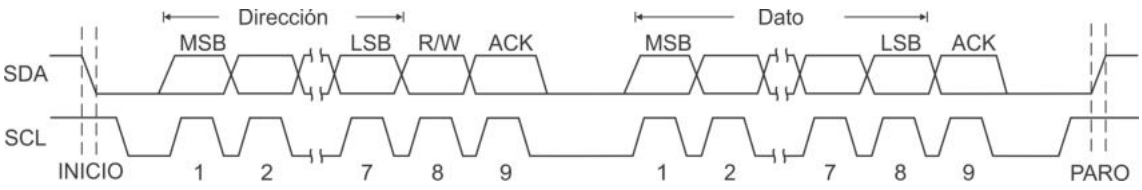


Figura 8.5: Mensaje con un dato a través de la interfaz TWI

8.2.3. 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 los resistores de *pull-up* sirve para coordinar la comunicación entre el maestro y el esclavo, si el esclavo requiere tiempo para el procesamiento de un dato, puede colocar un nivel bajo en SCL, retrasando con ello al siguiente paquete de datos. 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 8.5 se ilustra la transmisión de un mensaje con un solo dato. 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.

8.3. Sistemas Multi-Maestros

La interfaz TWI permite el manejo de varios maestros, pero solo uno toma el control del bus en un instante de tiempo. El bus podrá ser utilizado por otro maestro hasta que sea liberado con una condición de PARO. Con el bus disponible dos o más maestros pueden intentar iniciar una transmisión al mismo tiempo, esta situación va a dar lugar a 2 problemas que deben resolverse para que la comunicación proceda de manera normal. Estos problemas son:

- Los maestros van generar señales de reloj que no están completamente en fase, estas deben combinarse para que los maestros puedan sincronizar sus operaciones con la señal de reloj resultante.
- Puesto que solo un maestro va a realizar las transferencias, se desarrolla un proceso de selección, conocido como arbitración, para definir qué maestro va a tomar el control del bus.

La AND alambrada es fundamental en la solución del primer problema, la AND ocasiona que la señal de reloj resultante corresponda con la intersección de las señales de reloj emitidas por los maestros.

En la Figura 8.6 se muestra la señal de reloj en la línea SCL, resultante de la intersección de las señales de reloj de dos maestros, marcados como A y B. Puede

verse que el tiempo en alto de la señal de reloj resultante es más corto que el de las señales de reloj de los maestros A y B.

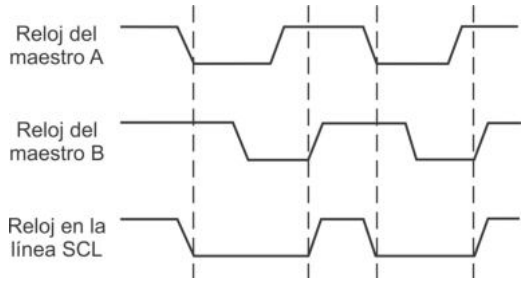


Figura 8.6: Sincronización de las señales de reloj de 2 maestros

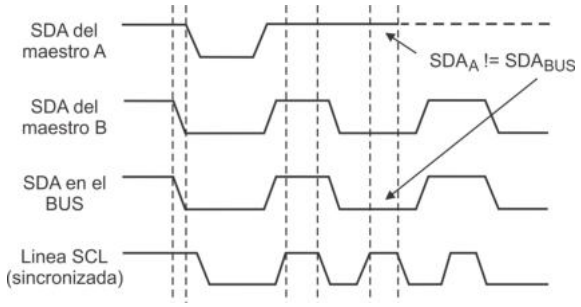


Figura 8.7: Proceso de arbitración ente 2 maestros

El proceso de arbitración se realiza de la siguiente manera, después de que un maestro coloca un bit en la línea SDA, debe monitorear para determinar si su valor coincide con el valor presente en el bus. El maestro pierde la arbitración cuando el valor leído no coincide con el valor colocado. En la Figura 8.7 se ilustra este proceso, después de la condición de inicio, en el primer ciclo de reloj ambos maestros colocaron un 1 lógico en el bus y es el valor que ambos leen. En el segundo ciclo el maestro A coloca un 1 y el maestro B coloca un 0, como un resultado debido a la AND alambrada, en la línea SDA se tiene un 0, por lo tanto, el maestro A ha perdido la arbitración.

Los maestros que pierden un proceso de arbitración 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.

Por la forma en que se realiza el proceso, si dos maestros intentan direccionar al mismo esclavo, pero uno para escritura (SLA+W) y otro para lectura (SLA+R), será la escritura la que se lleve a cabo. Si el direccionamiento por ambos maestros al mismo esclavo es para escritura (SLA+W), la arbitración va a continuar en el paquete de datos. Un direccionamiento de dos maestros al mismo esclavo para lectura no va a

ocurrir porque las operaciones de lectura son posteriores a operaciones de escritura. También debe notarse que una llamada general es prioritaria al direccionamiento de cualquier esclavo. En cualquier situación, la arbitración termina cuando queda solo un maestro y puede requerir de muchos bits.

Existen algunas condiciones ilegales de arbitración que deben evitarse, estas son:

- 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.

Para que estas condiciones no ocurran, en sistemas multi-maestros todas las transmisiones deben utilizar paquetes que contengan el mismo número de bytes, esta tarea es responsabilidad del protocolo de comunicación empleado.

8.4. Organización de la Interfaz TWI en los AVR

La interfaz TWI se compone de cuatro módulos principales, su organización se muestra en la Figura 8.8 en donde se distinguen 6 Registros I/O para su manejo, los módulos son descritos en los apartados de la presente sección. Los registros de la interfaz son accesibles por el núcleo AVR, desde el bus interno, y se describen en la sección siguiente.

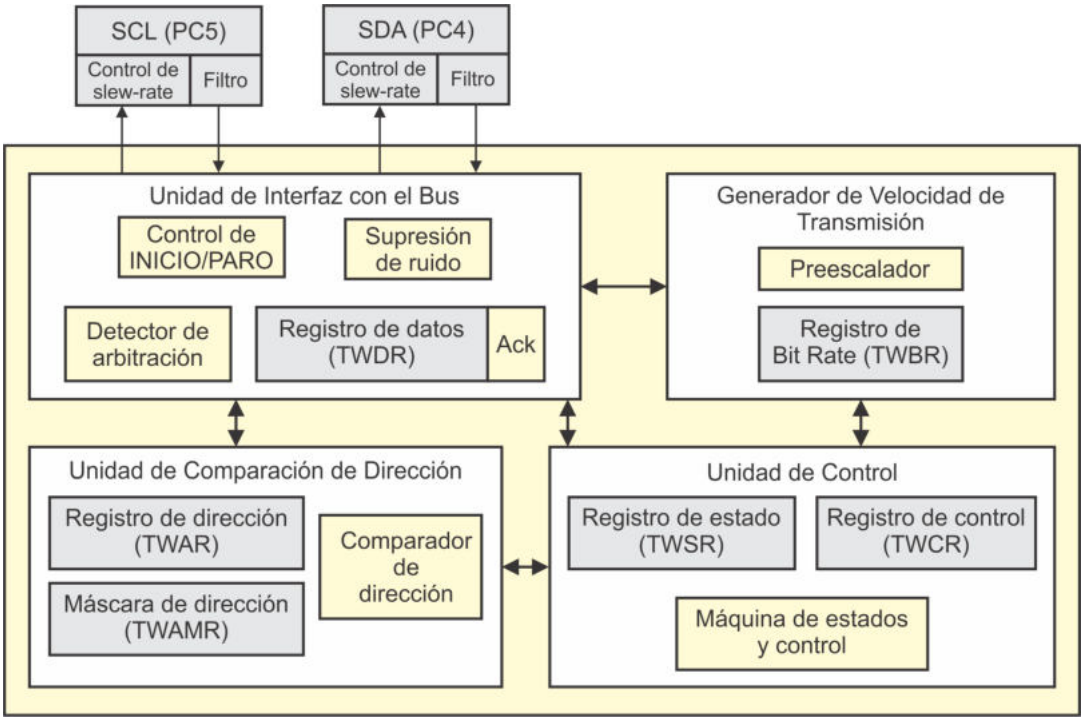


Figura 8.8: Organización de la interfaz TWI

Tabla 8.1: Selección del factor de preescala para la velocidad de transmisión

TWPS1	TWPS0	Factor de preescala
0	0	1
0	1	4
1	0	16
1	1	64

8.4.1. Terminales SCL y SDA

Cuando la interfaz serial de 2 hilos es habilitada, las terminales PC5 y PC4 se desconectan del puerto digital para convertirse en la línea de reloj (SCL) y la línea de datos (SDA), respectivamente, de la interfaz TWI. En este modo, se activa un filtro supresor que anula picos con una duración menor a 50 ns en las entradas, y las terminales son manejadas por un controlador con drenaje abierto, el cual cuenta con un limitador de *slew rate* para cumplir con las especificaciones de la interfaz. El bus TWI requiere de 2 resistores externos de *pull-up*, es posible habilitar y usar los resistores de las terminales PC5 y PC4, para no agregar hardware adicional.

8.4.2. Generador de Velocidad de Transmisión

Este módulo determina la velocidad de transmisión en el bus, es decir, la frecuencia de la señal SCL. La velocidad debe establecerse cuando el MCU está operando en modo maestro, en modo esclavo se puede omitir la configuración porque es el maestro quien genera la señal de reloj. La frecuencia de SCL depende del valor del Registro de Bit Rate (`TWBR`) y del factor de preescala seleccionado con los bits `TWPS[1:0]` del Registro de Estado de la Interfaz TWI (`TWSR`). En la Tabla 8.1 se muestran los diferentes factores de preescala. Considerando el valor del registro `TWBR` y el factor de preescala, la frecuencia de SCL se genera de acuerdo con la ecuación:

$$Frecuencia_{SCL} = \frac{Frecuencia_{CPU}}{16+2(TWBR)(factor\_de\_preescala)}$$

Se observa en la ecuación anterior que la velocidad de transmisión máxima es de $Frecuencia_{CPU}/16$, este resultado establece una restricción para el esclavo, el esclavo no genera la señal SCL pero debe operar a una frecuencia por lo menos de 16 veces la frecuencia de SCL, para una adecuada sincronización.

8.4.3. Unidad de Interfaz con el Bus

Este módulo incluye al Registro de Datos de la Interfaz (`TWDR`), este es un registro de desplazamiento en donde se debe ubicar la dirección o dato para su transmisión, o donde va a quedar una dirección o dato después de su recepción. El bit ACK, adyacente al registro `TWDR` en la Figura 8.8, 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 bit `TWEA` (habilitador de recono-

cimiento) del registro de control de la interfaz TWI (`TWCR`). Cuando la interfaz está transmitiendo, el valor del bit ACK se puede conocer a partir de los bits de estado en el registro `TWSR`.

El bloque denominado Control de INICIO/PARO es el responsable de generar y detectar las condiciones de INICIO, INICIO REPETIDO y PARO. Las condiciones de INICIO 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 de 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.

8.4.4. Unidad de Comparación de Dirección

Cuando un MCU es parte de un sistema con interconexión por la interfaz TWI, debe contar con una dirección a la que dará respuesta como esclavo, esta dirección debe escribirse en los 7 bits más significativos del registro `TWAR`. En la posición menos significativa del registro `TWAR` se encuentra el bit `TWGCE`, que es el habilitador de llamadas generales, con este bit en alto el MCU también va a atender las llamadas generales. El Comparador de dirección determina si la dirección recibida coincide con la establecida en el registro `TWAR` o con la GCA, si se puso en alto al bit `TWGCE`, ante una coincidencia, se informa a la Unidad de Control, para que realice las acciones correspondientes.

En el registro `TWAMR` se puede establecer una máscara para la dirección del dispositivo, la disposición de los bits en el registro `TWAMR` es la misma que en el registro `TWAR`, los bits que se pongan en alto en `TWAMR` serán ignorados durante la comparación de direcciones, de esta manera, un dispositivo esclavo puede dar respuesta a más de una dirección.

Por ejemplo, si en un MCU su registro `TWAR` tiene el valor 0x37, significa que la dirección como esclavo del MCU es 0x1B y que dará respuesta a llamadas generales, pero si en el registro `TWAMR` se escribe 0x02, va a responder a las direcciones 0x1A y 0x1B. En un caso hipotético donde los 7 bits más significativos del registro `TWAMR` se pongan en alto, hará que el esclavo dé respuesta a cualquier dirección.

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, si se espera su reactivación por este recurso.

Tabla 8.2: Registros para el manejo de la interfaz TWI

Registro	Dirección	Operación
TWDR	(0xBB)	Registro de datos de la interfaz TWI
TWBR	(0xB8)	Registro para establecer la velocidad de las transferencias
TWAR	(0xBA)	Registro de dirección de la interfaz TWI
TWAMR	(0xBD)	Máscara para la dirección de la interfaz TWI
TWCR	(0xBC)	Registro de control de la interfaz TWI
TWSR	(0xB9)	Registro de estado de la interfaz TWI

8.4.5. 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`, ubicada en la posición más significativa de `TWCR`, y en el siguiente ciclo de reloj se actualiza el registro de estado (`TWSR`), mostrando el código que identifica al evento.

El registro `TWSR` solo tiene información relevante después de que la bandera `TWINT` es puesta en alto, en otras circunstancias, contiene el código de estado 0xF8, 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.

8.5. Registros para el manejo de la Interfaz TWI

En la Tabla 8.2 se muestran todos los registros involucrados en el manejo de la Interfaz TWI, por su dirección se observa que todos son Registros I/O Extendidos.

El registro `TWDR` (*TWI Data Register*) es el *buffer* para transmisión y recepción de datos. En modo transmisor, en `TWDR` se coloca la dirección o dato para su envío.

En modo receptor, en `TWDR` se ubica al último dato recibido para su lectura. Este registro solo puede ser escrito o leído después de la puesta en alto del bit `TWINT`, por ello, el registro no puede ser modificado por el usuario hasta que ocurra un evento, esto significa que, un maestro no puede escribir la dirección de un esclavo si antes no ha enviado una condición de INICIO. La bandera `TWINT` debe limpiarse después de escribir o leer al registro `TWDR`, reescribiéndole un 1 lógico, para continuar con las operaciones en el bus.

El registro `TWBR` (*TWI Bit Rate Register*) determina la velocidad de transmisión de la interfaz cuando el MCU trabaja como maestro, reflejada en la frecuencia de la señal de reloj SCL, establecida por la ecuación:

$$Frecuencia_{SCL} = \frac{Frecuencia_{CPU}}{16+2(TWBR)(factor\,de\,preescala)}$$

Donde el factor de preescala se elige con los bits `TWPS[1:0]` del registro `TWSR`, cuyas opciones se mostraron en la Tabla 8.1.

El registro `TWAR` (*TWI (Slave) Address Register*) sirve para establecer la dirección a la que responderá el MCU en modo esclavo, así como para habilitar la atención a las llamadas generales. Sus bits son:

REG.	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
TWAR	TWA6	TWA5	TWA4	TWA3	TWA2	TWA1	TWA0	TWGCE

- **Bits [7:1] - TWA[6:0]:** (*TWI Slave Address*) Estos bits constituyen la dirección a la que responderá el MCU como esclavo.
- **Bit 0 - TWGCE:** (*TWI General Call Recognition Enable Bit*) Este bit debe ponerse en alto para que el MCU atienda las llamadas generales, las cuales utilizan la dirección "000 0000".

El registro `TWAMR` (*TWI (Slave) Address Mask Register*) puede emplearse para establecer una máscara para la dirección del MCU como esclavo, los bits indicados en la máscara no son considerados en la comparación, por lo que el MCU como esclavo puede dar respuesta a más de una dirección. Sus bits son:

REG.	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
TWAMR	TWAM6	TWAM5	TWAM4	TWAM3	TWAM2	TWAM1	TWAM0	-

- **Bits [7:1] - TWAM[6:0]:** (*TWI Address Mask*) Bits para establecer la máscara de dirección.
- **Bit 0:** No está implementado.

El registro para el control de la interfaz es el `TWCR` (*TWI Control Register*), 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 con el registro `TWDR`. Los bits del registro `TWCR` son:

REG.	7	6	5	4	3	2	1	0
TWCR	TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	-	TWIE

- **Bit 7 - TWINT:** (*TWI Interrupt Flag*) Es la bandera de interrupción por la interfaz TWI, su puesta en alto indica que ocurrió un evento en la interfaz 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 reescribiéndole un 1 lógico, aun si se configura su interrupción. Esto porque mientras el bit `TWINT` está en alto, la señal SCL mantiene un nivel bajo en espera de que por software se obtenga el estado de la interfaz y se le dé respuesta. El acceso a los registros `TWAR`, `TWSR` y `TWDR` solo se puede realizar cuando la bandera `TWINT` está en alto, una vez que se ha limpiado, ya no es posible el acceso. Solo cuando se establece una condición de INICIO se hace la escritura explícita de un 1 lógico en el bit `TWINT`, las escrituras posteriores son para limpiar la bandera.
- **Bit 6 - TWEA:** (*TWI Enable Acknowledge Bit*) Habilita la generación del bit de reconocimiento (ACK), cuando el bit está en alto, la interfaz TWI genera el bit de reconocimiento si ocurre una de las siguientes situaciones:

  1. El MCU ha recibido su dirección de esclavo.
  2. El MCU ha recibido una llamada general, estando el bit `TWGCE` del registro `TWAR` en alto.
  3. El MCU ha recibido un dato, siendo maestro receptor o esclavo receptor.

  Si el bit `TWEA` tiene un nivel bajo, la interfaz TWI está virtualmente desconectada del bus.
- **Bit 5 - TWSTA:** (*TWI START Condition Bit*) Si el bus está libre, un nivel alto en este bit genera una condición de INICIO y el MCU pasa a ser maestro. Si el bus no está libre, la interfaz espera hasta detectar una condición de PARO y después genera 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:** (*TWI STOP Condition Bit*) Si el MCU es maestro, un 1 en este bit genera una condición de PARO y posteriormente, el bit se limpia automáticamente. Si el MCU es 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:** (*TWI Write Collision Flag*) Esta bandera se pone en alto si se intenta escribir en el registro `TWDR` cuando el bit `TWINT` está en bajo, indicando una colisión en la escritura. La bandera se limpia cuando se escribe en `TWDR` después de que el bit `TWINT` es puesto en alto.

Tabla 8.3: Vector de interrupción para la interfaz TWI

Dirección	Evento	Descripción
0x030	TWI	Transferencia serial por TWI

- **Bit 2 - TWEN:** (*TWI Enable Bit*) Este bit es el habilitador de la interfaz TWI, con un 1 lógico en `TWEN`, la interfaz toma el control de las terminales SCL y SDA, habilitando al controlador del *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:** (*TWI Interrupt Enable*) Es el 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. Este recurso solo genera una interrupción, cuyo vector se muestra en la Tabla 8.3.

El último registro para el manejo de la interfaz TWI es el `TWSR` (*TWI Status Register*), este registro contiene los bits que indican el estado de la interfaz, así como los bits para definir el factor de preescala. Los bits de `TWSR` son:

REG.	7	6	5	4	3	2	1	0
TWSR	TWS7	TWS6	TWS5	TWS4	TWS3	-	TWPS1	TWPS0

- **Bits [7:3] - TWS[7:3]:** (*TWI Status*) En estos 5 bits se refleja el estado de la interfaz y del bus. Los códigos de estado dependen del modo de transmisión y se revisan en la siguiente sección. Puesto que los bits de estado ocupan las 5 posiciones más significativas del registro, para su lectura e interpretación correcta se debe utilizar una máscara AND con el byte 0xF8, para anular los bits no relacionados con el estado.
- **Bit 2:** No está implementado.
- **Bits [1:0] - TWPS[1:0]:** (*TWI Prescaler Bits*) Bits para la selección del factor de preescala en la interfaz TWI. En la Tabla 8.1 se mostraron los diferentes factores de preescala, 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.

8.6. Modos de Transmisión y Códigos de Estado

La interfaz puede operar en 4 modos:

- Maestro Transmisor (MT).
- Maestro Receptor (MR).

- Esclavo Transmisor (ST).
- Esclavo Receptor (SR).

Si se analizan los registros para el manejo de la interfaz TWI, se observará que no existen bits para definir si un MCU va a ser maestro o esclavo, el modo se establece en el contexto de la aplicación. Un MCU es maestro si consigue ganar el bus estableciendo una condición de INICIO, para posteriormente direccionar a un esclavo. El MCU será esclavo si fue seleccionado con su dirección establecida en el registro `TWAR` o a través de una llamada general.

Una aplicación puede requerir más de un modo de operación. Por ejemplo, si un MCU va a manejar una memoria SRAM vía TWI, durante la escritura el MCU debe ser MT. Sin embargo, en la lectura, puesto que la SRAM tiene diferentes localidades, el MCU inicia siendo MT para indicar la dirección de lectura, posteriormente, el MCU se debe volver MR y así podrá obtener los datos de la memoria. Si en la misma aplicación otro MCU direcciona al primero, este también puede funcionar en los modos ST y SR. La aplicación determina el modo que será empleado en cada fase de operación.

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 señal SCL se pone en bajo y la actividad en el bus se detiene. Por software debe leerse el estado de la interfaz y preparar la respuesta, en función de la aplicación. Con la respuesta lista, la bandera `TWINT` debe limpiarse para continuar con las actividades del bus.

8.6.1. Modo Maestro Transmisor (MT)

Un MCU en el modo MT envía una cantidad de bytes a un MCU u otro dispositivo en el modo SR, este flujo se muestra en la Figura 8.9. 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).

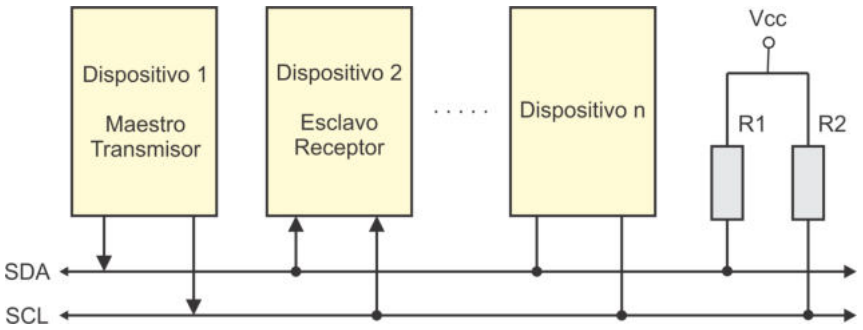


Figura 8.9: El dispositivo 1 transfiere datos en el modo Maestro Transmisor

Una condición de INICIO se genera escribiendo el siguiente valor en el registro `TWCR`:

TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	-	TWIE
1	X	1	0	X	1	X	X

Con esta combinación se habilita la interfaz TWI (`TWEN` = 1), se da paso a la condición de INICIO (`TWSTA` = 1) y se escribe en la bandera `TWINT`, el hardware limpia la bandera al iniciar con la operación. Si el bus está disponible, se transmite la condición de inicio, la bandera `TWINT` se pone en alto y en los bits de estado, del registro `TWSR`, se leerá el código 0x08.

En la Tabla 8.4 se muestran los estados posibles en el modo MT, con una descripción del estado de la interfaz y las diferentes acciones a seguir. En todos los códigos se asume un enmascaramiento para solo obtener los bits de estado.

Para entrar al modo MT, el maestro debe escribir una `SLA+W` en el registro `TWDR` y realizar 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:

TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	-	TWIE
1	X	0	0	X	1	X	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 o 0x38. Las acciones a realizar dependen del código de estado (Tabla 8.4).

Tabla 8.4: 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	Estableció una condición de INICIO	1. Transmitir una SLA+W
0x10	Estableció una condición de INICIO REPETIDO	1. Transmitir una SLA+W 2. Transmitir una SLA+R para conmutar la interfaz a MR
0x18	Transmitió una SLA+W y obtuvo ACK	1. Transmitir un byte de datos
0x20	Transmitió una SLA+W y obtuvo nACK	2. Transmitir un INICIO REPETIDO
0x28	Transmitió un byte de datos y obtuvo ACK	3. Transmitir una condición de PARO
0x30	Transmitió un byte de datos y obtuvo nACK	4. Transmitir una condición de PARO seguida de una condición de INICIO
0x38	Perdió una arbitración al enviar la SLA+W o un byte de datos	1. Liberar al bus, limpiando la bandera TWINT 2. Transmitir una condición de INICIO, cuando el bus esté libre

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 la bandera `TWINT` está

en alto. Después de escribir el dato, la bandera `TWINT` debe limpiarse escribiéndole un 1, en el registro `TWCR` debe escribirse la misma combinación:

TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	-	TWIE
1	X	0	0	X	1	X	X

Este proceso se repite por cada dato que se envía, el dato se escribe en `TWDR` cuando la bandera `TWINT` está en alto y posteriormente, el dato se envía al limpiar la bandera `TWINT`.

Después de enviar todos los datos requeridos por la aplicación, el maestro debe enviar una condición de PARO o un INICIO REPETIDO. El valor del registro `TWCR` para una condición de PARO es:

TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	-	TWIE
1	X	0	1	X	1	X	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.

8.6.2. Modo Maestro Receptor (MR)

Un MCU en el modo MR recibe una cantidad de bytes de un MCU u otro dispositivo en el modo ST, este flujo se muestra en la Figura 8.10. 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).

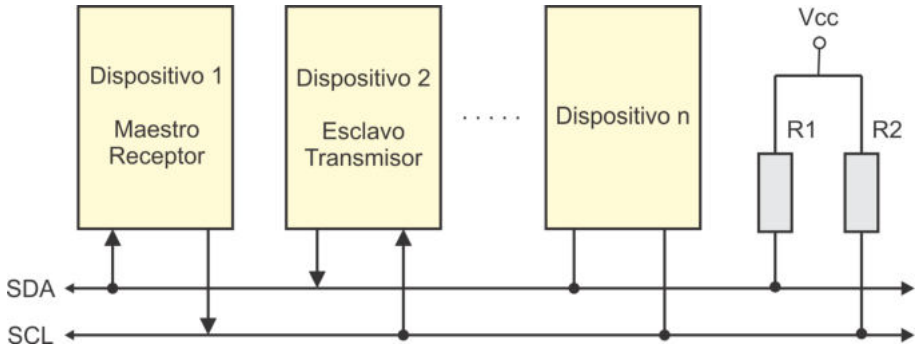


Figura 8.10: El dispositivo 1 recibe datos en el modo Maestro Receptor

Una condición de INICIO se genera escribiendo el siguiente valor en el registro `TWCR`:

TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	-	TWIE
1	X	1	0	X	1	X	X

Con esta combinación se habilita la interfaz TWI (`TWEN` = 1), se da paso a la condición de INICIO (`TWSTA` = 1) y se escribe en la bandera `TWINT`, el hardware limpia la bandera al iniciar con la operación. Si el bus está disponible, se transmite la condición de inicio, la bandera `TWINT` se pone en alto y en los bits de estado, del registro `TWSR`, se leerá el código 0x08.

En la Tabla 8.5 se muestran los estados posibles en el modo MR, con una descripción del estado de la interfaz y las diferentes acciones a seguir. En todos los códigos se asume un enmascaramiento para solo obtener los bits de estado.

Para entrar al modo MR, el maestro debe escribir una `SLA+R` en el registro `TWDR` y realizar 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:

TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	-	TWIE
1	X	0	0	X	1	X	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 o 0x48. Las acciones a realizar dependen del código de estado (Tabla 8.5).

Tabla 8.5: 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	Estableció una condición de INICIO	1. Transmitir una SLA+R
0x10	Estableció una condición de INICIO REPETIDO	1. Transmitir una SLA+R 2. Transmitir una SLA+W para conmutar la interfaz a MT
0x38	Perdió una arbitración al enviar la SLA+R	1. Liberar al bus, limpiando la bandera TWINT 2. Transmitir una condición de INICIO, cuando el bus esté libre
0x40	Transmitió una SLA+R y obtuvo ACK	1. Recibir un byte de datos y dar respuesta con ACK (TWEA = 1) 2. Recibir un byte de datos y dar respuesta con nACK (TWEA = 0)
0x50	Recibió un byte de datos y respondió con ACK	
0x48	Transmitió una SLA+R y obtuvo 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	Recibió un byte de datos y respondió con nACK	

Si la SLA+R se transmitió con éxito, el MCU maestro está listo para recibir uno o varios datos. Cada dato recibido se puede leer de `TWDR` cuando la bandera `TWINT` sea puesta en alto, el MCU debe responder con ACK mientras continúe recibiendo datos.

La respuesta con reconocimiento y la limpieza de la bandera `TWINT` se realizan al escribir en el registro `TWCR` el siguiente valor:

TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	-	TWIE
1	1	0	0	X	1	X	X

Una respuesta sin reconocimiento (nACK) le indica al esclavo que ya no se van a recibir más datos. Para ello, el valor a escribir en el registro `TWCR` es:

TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	-	TWIE
1	0	0	0	X	1	X	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:

TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	-	TWIE
1	X	0	1	X	1	X	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.

8.6.3. Modo Esclavo Receptor (SR)

Un MCU en el modo SR recibe una cantidad de bytes de un MCU u otro dispositivo en el modo MT, este flujo se muestra en la Figura 8.11.

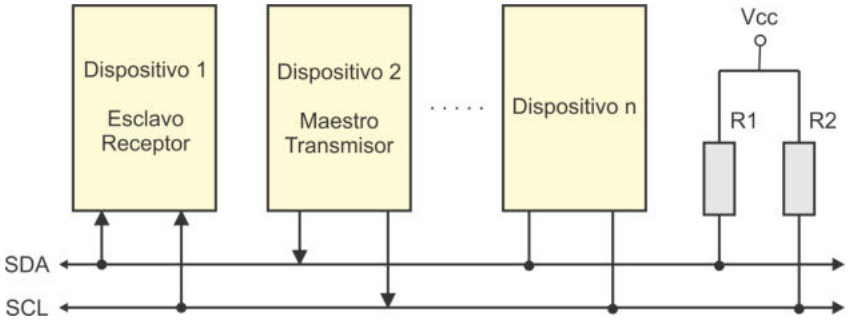


Figura 8.11: El dispositivo 1 recibe datos en el modo Esclavo Receptor

El MCU debe contar con una dirección a la que responderá 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 la dirección de llamada general (GCA).

En el registro `TWCR` debe habilitarse la interfaz TWI (`TWEN` = 1) y preparar una respuesta de reconocimiento (`TWEA` = 1), para ello, en este registro se debe escribir el valor:

TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	-	TWIE
0	1	0	0	X	1	X	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 que controla el 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 puede ser llevada al modo SR si perdió una arbitración mientras estaba en modo maestro.

En la Tabla 8.6 se muestran los estados posibles en el modo SR, con una descripción del estado de la interfaz y las diferentes acciones a seguir. En todos los códigos se asume un enmascaramiento para solo obtener los bits de estado.

Tabla 8.6: 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	El MCU fue direccionado como esclavo con una SLA+W y respondió con ACK	
0x68	El MCU perdió una arbitración como maestro, fue direccionado como esclavo con una SLA+W y respondió con ACK	
0x70	El MCU fue direccionado como esclavo con una GCA y respondió con ACK	1. Recibir un byte de datos y regresar un ACK (TWEA = 1)
0x78	El MCU perdió una arbitración como maestro, fue direccionado como esclavo con una GCA y respondió con ACK	2. Recibir un byte de datos y regresar un nACK (TWEA = 0)
0x80	El MCU recibió un byte de datos y respondió con ACK, previamente se había direccionado con una SLA+W	
0x90	El MCU recibió un byte de datos y respondió con ACK, previamente se había direccionado con una GCA	
0x88	El MCU recibió un byte de datos y respondió con 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 SLA o la GCA (TWEA = 0)
0x98	El MCU recibió un byte de datos y respondió con nACK, previamente se había direccionado con una GCA	
0xA0	El MCU detectó una condición de PARO o de INICIO REPETIDO, mientras estaba direccionado como esclavo	2. Conmutar a un modo de esclavo no direccionado, capaz de reconocer su SLA o la GCA (TWEA = 1)

Si el bit `TWEA` es puesto en bajo durante una transferencia, la interfaz coloca un nACK en la línea SDA después de recibir el próximo dato. Esto se puede hacer para que el receptor 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 dirección o la GCA, pero el monitoreo continúa realizándose, de manera que puede ocurrir un reconocimiento tan pronto como el bit `TWEA` es puesto en alto.

8.6.4. Modo Esclavo Transmisor (ST)

Un MCU en el modo ST transmite una cantidad de bytes a un MCU u otro dispositivo en el modo MR, este flujo se muestra en la Figura 8.12.

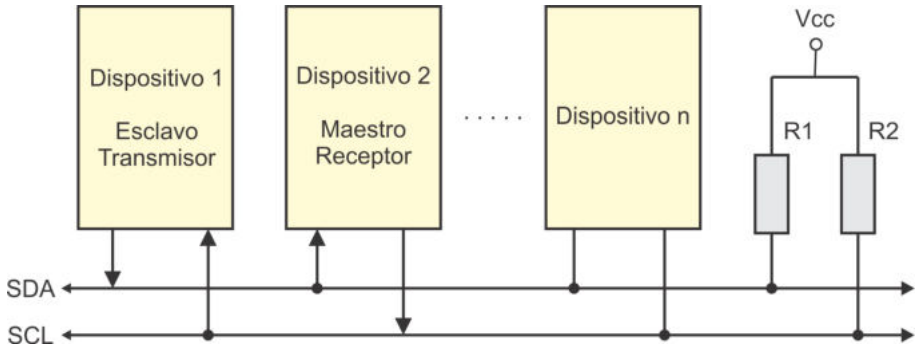


Figura 8.12: El dispositivo 1 envía datos en el modo Esclavo Transmisor

El MCU debe contar con una dirección a la que responderá 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 la dirección de llamada general (GCA), aunque en las llamadas generales no se solicita a los esclavos transmitir datos, porque se provocaría una colisión en el bus.

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:

TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	-	TWIE
0	1	0	0	X	1	X	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 que controla el 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 puede ser llevada al modo ST si perdió una arbitración mientras estaba en modo maestro.

En la Tabla 8.7 se muestran los estados posibles en el modo ST, con una descripción del estado de la interfaz y las diferentes acciones a seguir. En todos los códigos se asume un enmascaramiento para solo obtener los bits de estado.

Tabla 8.7: 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	El MCU fue direccionado como esclavo con una SLA+R y respondió con ACK	1. Transmitir un byte de datos y recibir ACK 2. Transmitir un byte de datos y recibir nACK
0xB0	El MCU perdió una arbitración como maestro, fue direccionado como esclavo con una SLA+R y respondió con ACK	
0xB8	El MCU transmitió un byte de datos y recibió ACK	
0xC0	El MCU transmitió un byte de datos y recibió nACK	1. Conmutar a un modo de esclavo no direccionado, desactivando la interfaz para no reconocer su SLA o la GCA (TWEA = 0) 2. Conmutar a un modo de esclavo no direccionado, capaz de reconocer su SLA o la GCA (TWEA = 1)
0xC8	El MCU transmitió el último byte de datos, al colocar TWEA = 0 y recibió ACK	

Las transferencias pueden terminar por dos situaciones diferentes, una ocurre cuando el ST no obtiene reconocimiento después de transmitir un byte, de esta forma el MR le indica que no va a recibir más datos (el SR observa el código de estado 0xC0) y la otra situación se presenta cuando el ST no tiene más datos por enviar, para indicarlo debe desactivar al bit `TWEA` y con ello, el MCU conmuta a un esclavo sin direccionar (se presenta el código de estado 0xC8), si el MR demanda más datos generando reconocimiento (`ACK`), va a recibir 1's en la línea de datos.

8.6.5. Estados Misceláneos

La interfaz TWI incluye 2 códigos de estado independientes de los 4 modos de operación, estos se muestran en la Tabla 8.8 con un enmascaramiento para solo conservar los bits de estado. El estado 0xF8 se presenta cuando se realiza un acceso a los registros de la interfaz sin que la bandera `TWINT` se haya puesto en alto y el estado 0x00 indica un error en el bus.

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. La bandera `TWINT` es puesta en alto cuando ocurre un error. Para recuperar al bus, el bit `TWSTO` debe ser

puesto en alto y la bandera `TWINT` debe limpiarse reescribiéndole un 1. Con esto, la interfaz entra a un modo de esclavo no direccionado, no se transmite una condición de paro y las líneas del bus, SDA y SCL, son liberadas. Después de realizar estas acciones, el bit `TWSTO` se limpia automáticamente.

Tabla 8.8: 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 continuar con las actividades del MCU
0x00	Ocurrió un error en el bus, debido a una condición ilegal de INICIO o PARO	1. El bit TWSTO debe ser puesto en alto para liberar al bus de la condición. No se genera una condición de PARO y el bit TWSTO es limpiado automáticamente.

8.7. Operación en Modo Maestro

El uso de la interfaz TWI ha crecido significativamente por la simpleza del hardware para conectar diferentes dispositivos, por ejemplo, en el caso de los sensores, los módulos TWI o I^2C comerciales se presentan con una interfaz digital que ahorra las etapas de acondicionamiento de señal, minimizando el tamaño del circuito y por lo tanto, el consumo de potencia. Para el manejo de la enorme cantidad de módulos que están emergiendo, el MCU va a trabajar en modo maestro (transmisor o receptor) y los módulos serán esclavos, por lo que es conveniente contar con una biblioteca de funciones que facilite este trabajo.

El manejo de la interfaz TWI se reduce a 5 funciones, que se describen en esta sección, las funciones no utilizan interrupciones porque una vez que se inicia con una transacción por TWI, esta debe continuar hasta concluir, sin poder pausar para dar paso a otras tareas. El prototipo de las funciones se debe integrar en un archivo denominado `TWI.h` y su cuerpo debe estar en el archivo TWI.c, de esta manera, las funciones podrán reutilizarse al incluir la biblioteca en los nuevos proyectos.

Configuración de la Interfaz

La función `TWI_Config()` configura la velocidad y habilita la interfaz. La función no recibe y tampoco regresa argumentos, su código es:

```
void TWI_Config() {

   TWBR = 0x02;          // Frecuencia del MCU entre 20
   TWSR = 0x00;          // Factor de preescala en 1
   TWCR = 1 << TWEN;     // Habilita a la interfaz
}
```

Con esta función, la frecuencia de la señal SCL es la frecuencia del MCU dividida entre 20, porque 20 es el resultado de la operación $16 + 2(TWBR)(factor\,de\,preescala) = 16 + 2(2)(1)$.

Si el MCU está operando a 1 MHz, la interfaz TWI va a trabajar a 50 kHz. La frecuencia se puede ajustar en función de la velocidad permitida por el módulo esclavo.

Condición de Inicio

La función `TWI_Inicio()` establece una condición de INICIO o INICIO REPETIDO, esta función debe ser la primera en ser llamada en una transacción por TWI. La función no recibe argumentos, regresa 0x01 si cualquiera de las dos condiciones se establecieron con éxito o el código de estado si ocurrió otra situación. El código de la función es:

```
uint8_t TWI_Inicio() {
uint8_t edo;

   TWCR = (1<<TWINT)|(1<<TWEN)|(1<<TWSTA); // Condición de inicio
   while(!(TWCR & (1<<TWINT)));            // Espera la bandera TWINT
   edo = TWSR & 0xF8;                      // Obtiene el estado
                                           // 2 posibilidades de éxito:
   if( edo == 0x08 ||                      // 1. Condición de INICIO
       edo == 0x10 )                       // 2. INICIO REPETIDO
          return 0x01;

   return edo;                             // Situación no esperada
}
```

Debe notarse que cuando la función termina, la bandera `TWINT` sigue en alto, por lo que la función que continúe en la secuencia de llamadas, podrá tener acceso al registro `TWDR`.

Escritura de un Byte

La función `TWI_EscByte()` se encarga de escribir un byte en el bus, el byte es recibido como argumento y puede ser una SLA+W, SLA+R o un dato. La función regresa 0x01 si el byte fue colocado con éxito en el bus o el código de estado si ocurrió otra situación. El código de la función es:

```
uint8_t TWI_EscByte(uint8_t dato) {
uint8_t  edo;

   TWDR = dato;                    // Carga el dato
   TWCR = (1<<TWEN)|(1<<TWINT);    // Inicia el envío
   while(!(TWCR & (1<<TWINT)));    // Espera la bandera TWINT
```

```
    edo = TWSR & 0xF8;                  // Obtiene el estado
                                        // 3 posibilidades de éxito:
    if( edo == 0x18 ||                  // Transmitió una SLA + W con ACK
        edo == 0x28 ||                  // Transmitió una SLA + R con ACK
        edo == 0x40 )                   // Transmitió un dato con ACK
            return 0x01;

    return edo;                         // Situación no esperada
}
```

Esta función también termina con la bandera **TWINT** en alto.

Lectura de un Byte

La función `TWI_LeeByte()` se encarga de leer un byte en el bus, la función recibe dos argumentos, el primero es un apuntador a la variable en donde se ubicará al byte leído y el segundo es la indicación para hacer la lectura con o sin reconocimiento, una lectura sin reconocimiento de un MR le indica a un ST que ya no debe enviar más datos. La función regresa 0x01 si el byte se leyó con éxito del bus o el código de estado si ocurrió otra situación. El código de la función es:

```
uint8_t TWI_LeeByte(uint8_t *dato, uint8_t ack){
uint8_t  edo;

    if(ack)
       TWCR |= (1<<TWEA);               // Lectura con ACK
    else
       TWCR &= (~(1<<TWEA));            // Lectura con nACK

   TWCR |= (1<<TWINT);                  // Inicia la lectura
    while(!(TWCR & (1<<TWINT)));        // Espera la bandera TWINT
    edo = TWSR & 0xF8;                  // Obtiene el estado
                                        // 2 posibilidades de éxito
    if( edo == 0x58 ||                  // 1. Dato leído con ACK
        edo == 0x50 ) {                 // 2. Dato leído con nACK
          *dato = TWDR;                 // Ubica el dato leído
          return 0x01;
    }

    return edo;                         // Situación no esperada
}
```

La función `TWI_LeeByte()` también termina con la bandera `TWINT` en alto.

Condición de Paro

La función `TWI_Paro()` establece una condición de PARO, esta función debe ser la última en ser llamada en una transacción por TWI. La función no recibe y tampoco regresa argumentos, su código es:

```
void TWI_Paro() {

   TWCR=(1<<TWINT)|(1<<TWEN)|(1<<TWSTO);   // Condición de paro
   while(TWCR & 1<<TWSTO);                 // El bit se limpia por HW
}
```

La función espera a que el bit `TWSTO` sea puesto en bajo, la interfaz continúa habilitada y la bandera `TWINT` también queda en bajo.

Secuencias de Escritura y Lectura

Al emplear las funciones previamente descritas, el manejo de la interfaz TWI para un MCU en modo maestro se simplifica significativamente, como un ejemplo, en la siguiente secuencia de instrucciones se envía la constando 0x32 a un esclavo con dirección 0x08:

```
   TWI_Inicio();              // Condición de INICIO
   TWI_EscByte(0x08 << 1);    // SLA+W
   TWI_EscByte(0x32);         // Escribe un dato
   TWI_Paro();                // Condición de PARO
```

Al hacer el desplazamiento de la dirección del esclavo se inserta un 0 y con ello se crea la SLA+W.

Una secuencia para leer un dato con reconocimiento, de un esclavo con dirección 0x08, dejando el dato leído en una variable `aux`, es la siguiente:

```
   TWI_Inicio();              // Condición de INICIO
   TWI_EscByte(0x08<<1 + 1);  // SLA+R
   TWI_LeeByte(&aux, 1);      // Lectura con ACK
   TWI_Paro();                // Condición de PARO
```

El desplazamiento de la dirección del esclavo inserta un 0 en su posición menos significativa, al sumar un 1 se crea la SLA+R. A la función `TWI_LeeByte()` se le pasa como primer argumento la dirección de la variable `aux`, porque la función espera un apuntador.

En la práctica, una secuencia de lectura no es aislada, generalmente es posterior a una secuencia de escritura porque cuando un MCU maestro va a obtener datos de un dispositivo funcionando como esclavo, el maestro primero debe indicar qué datos requiere, el INICIO REPETIDO se utiliza para que el MCU pase de MT a MR.

En las secuencias de escritura y lectura por TWI se está ignorando el retorno de las funciones de la biblioteca, este retorno debe evaluarse para dar robustez a un sistema y si el resultado es diferente de 0x01, se deben codificar las acciones necesarias para notificar al usuario.

8.8. Ejemplos de Uso de la Interfaz TWI

En esta sección se muestran tres ejemplos para ilustrar el uso de la interfaz TWI, en el primero se construye un sistema maestro-esclavos similar al sistema del Ejemplo 7.5, en donde se evaluó la interfaz SPI. En el segundo ejemplo se desarrollan funciones para el acceso a una memoria con interfaz TWI y en el tercero se utilizan estas funciones con el apoyo de la USART.

Ejemplo 8.1 - Sistema Maestro-Esclavos
En la Figura 8.13 se muestra un sistema con un maestro y tres esclavos, el maestro tiene 2 arreglos de interruptores (con 8 y 2 interruptores) y un botón, y a los esclavos se les han conectado 8 LEDs. El objetivo es enviar datos de 8 bits del maestro a los diferentes esclavos, los 2 interruptores son para seleccionar a uno de los esclavos y los 8 para introducir el dato. Con ello, cada vez que el botón es presionado, el maestro debe enviar el dato al esclavo seleccionado. Cuando un esclavo reciba un dato, lo debe mostrar en los LEDs. Los esclavos obtienen su dirección de dos interruptores, en donde deben configurarse las combinaciones 1, 2 y 3, porque la dirección 0 es para una llamada general, mediante la cual, el maestro enviará el mismo dato a todos los esclavos.

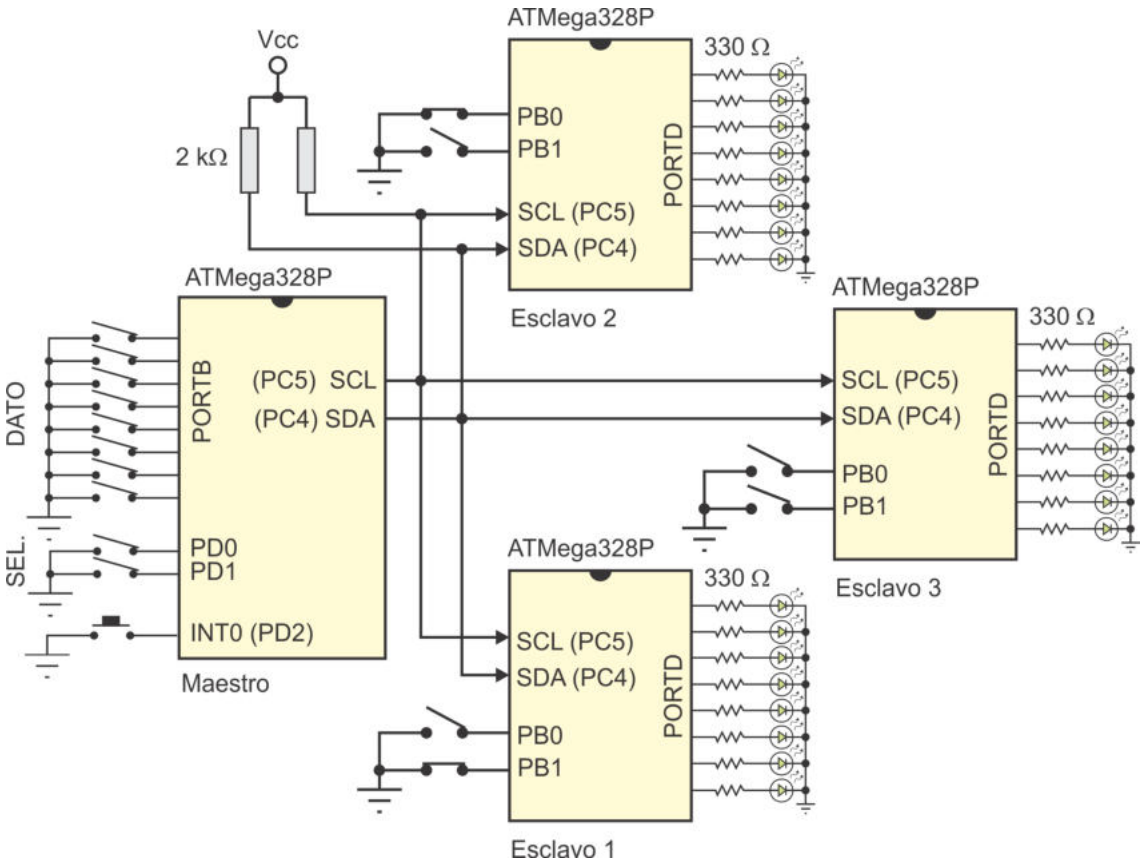


Figura 8.13: Envío de información de un maestro a 3 esclavos por TWI

El programa del maestro incluye la biblioteca `TWI.h` con las funciones descritas en la sección previa. El botón está conectado en la terminal PD2, por lo que utiliza la interrupción externa 0, en su ISR se obtiene el número de esclavo, el dato a enviar y se realiza la secuencia de escritura.

La difusión a todos los esclavos se simplifica porque la dirección corresponde con una llamada general (dirección 0x00). Por último, dado que no se han establecido condiciones para la velocidad de las transferencias, se utilizará la función de configuración incluida en la biblioteca, la cual establece la frecuencia en 50 kHz, asumiendo que el MCU está operando a 1 MHz. El programa en lenguaje C para el maestro es:

```
#include <avr/io.h>
#include <avr/interrupt.h>
#include "TWI.h"                    // Para el manejo de la interfaz TWI

ISR(INT0_vect) {                    // Botón presionado
uint8_t esclavo, dato, aux;

   esclavo = PIND & 0x03;           // Dirección del esclavo
   esclavo = esclavo << 1;          // Compone la SLA+W
   dato = PINB;

   aux = TWI_Inicio();              // Condición de INICIO
   if(aux != 0x01) {                // Si no se consigue
      TWCR |= (1<<TWINT);           // Limpia la bandera y
      return;                       // no continúa
   }
   aux = TWI_EscByte(esclavo);      // Direcciona con la SLA+W
   if(aux != 0x01) {                // Ocurre algo inesperado
      TWI_Paro();                   // No continúa, libera al bus
      return;
   }
   TWI_EscByte(dato);               // Envía el dato
   TWI_Paro();
}

int  main() {

   DDRB = 0x00;                     // Entrada para el dato
   DDRD = 0x00;                     // Para la dirección e interrupción
   PORTB = 0xFF;                    // Pull-up
   PORTD = 0x07;

   TWI_Config();                    // Configura la interfaz TWI
   EICRA = 0x02;                    // INT0 por flanco de bajada
   EIMSK = 0x01;                    // Habilita la INT0

   sei();                           // Habilitador global de interrupciones
   while(1)                         // Ocioso en el lazo infinito
      asm("nop");
}
```

Se observa que el uso de la biblioteca simplifica el manejo de la interfaz TWI para el maestro, las funciones regresan el valor de 0x01 en todas las situaciones correctas y el código de estado en caso de ocurrir una situación no esperada, por ello, en la ISR se evalúa su retorno, si ocurre una situación no esperada se limpia la bandera `TWINT` o se establece una condición de PARO para liberar al bus. De esta manera, si un esclavo se desconecta, el sistema puede funcionar con los dos restantes.

Para el esclavo, en el registro `TWAR` se debe escribir la dirección a la que dará respuesta, leída del Puerto B, y también habilitar la atención a las llamadas generales. Además, en el registro `TWCR` se debe activar al recurso con reconocimiento, es conveniente hacer uso de las interrupciones para no tener que sondear en espera de que ocurra alguna actividad en la interfaz. Cuando ocurra una interrupción, en la ISR se debe evaluar el estado de la interfaz para determinar las acciones a realizar.

No se puede utilizar la biblioteca porque fue diseñada para cuando el MCU opera en modo maestro y tampoco es necesario configurar la velocidad de operación porque la determina el maestro. El programa para el MCU esclavo es:

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

ISR(TWI_vect) {
uint8_t  dato, estado;

   estado = TWSR & 0xFC;              // Lee el estado de la interfaz
   switch(estado) {
      case 0x60:                      // Direccionado con su SLA
      case 0x70: TWCR |= 1 << TWINT; // o con la GCA, limpia la bandera
                 break;
      case 0x80:                      // Recibió un dato, con su SLA
      case 0x90: dato = TWDR;         // o con la GCA, lo lee
                 PORTD = dato;        // y ubica en el Puerto D
                 TWCR |= 1 << TWINT; // Limpia la bandera TWINT
                 break;
// Situación inesperada: Libera al bus de cualquier error
        default: TWCR |= (1 << TWINT) | (1 << TWSTO);
   }
}

int  main() {              // Programa principal
uint8_t   dir;

   DDRB = 0x00;            // Dirección del esclavo
   DDRD = 0xFF;            // Salida para el dato
   PORTB = 0xFF;           // Pull-Up

   dir = PINB & 0x03;      // Lee la dirección del esclavo
   dir <<= 1;              // Ubica la dirección y
   dir |= 0x01;            // habilita para reconocer la GCA
   TWAR = dir;
```

```
// Habilita la interfaz, con reconocimiento e interrupción
   TWCR = (1 << TWEA) | (1 << TWEN) | (1 << TWIE);

   sei();                    // Habilitador global de interrupciones

   while(1)                  // Ocioso en el lazo infinito
      asm("nop");
}
```

En la ISR se observa que en el caso de *default* el esclavo aparentemente establece una condición de PARO, lo cual es incongruente porque al maestro le corresponde establecer estas condiciones. Sin embargo, si se revisan los códigos de estado de la Tabla 8.6, se notará que en esta aplicación no debe ocurrir un estado diferente a los que ya se están comparando, por lo tanto, un estado diferente solo puede ser consecuencia de un error en el bus y un esclavo puede salir de esta situación colocando al bit `TWSTO` en alto.

El circuito PCF8570 es una memoria RAM con interfaz I^2C de 256 x 8 bits. La memoria puede funcionar en los modos esclavo receptor o esclavo transmisor, con una dirección configurable entre 0x50 y 0x57, esta se define con 3 terminales externas, denominadas: A0, A1 y A2. En la Figura 8.14 se muestra la secuencia requerida por la interfaz para escribir n datos y en la Figura 8.15 se puede ver la secuencia requerida para leer n datos. En ambas operaciones, después de establecer la dirección de acceso, esta se incrementa automáticamente para la lectura o escritura de varios bytes. En el siguiente ejemplo se implementan un par de funciones para facilitar el acceso a una memoria PCF8570.

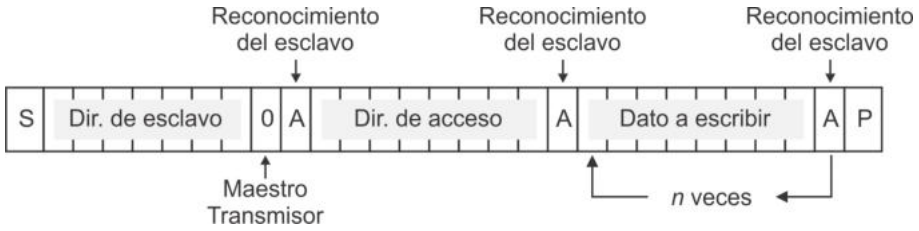


Figura 8.14: Secuencia de escritura en la memoria PCF8570

Ejemplo 8.2 - Funciones para el acceso a la memoria PCF8570
Realice dos funciones, la primera para escribir n bytes en una memoria PCF8570 y la segunda para leer n bytes de la misma memoria. La primera recibirá como argumentos: un arreglo con los datos a escribir, la cantidad de datos y la dirección donde van a iniciar las escrituras. Los argumentos para la segunda función son: un arreglo para colocar los datos, la cantidad de datos a leer y la dirección donde van a iniciar las lecturas. Para dar versatilidad a las funciones, suponga que la dirección de la memoria como esclavo se ha definido con la constante DIR_ESVO.

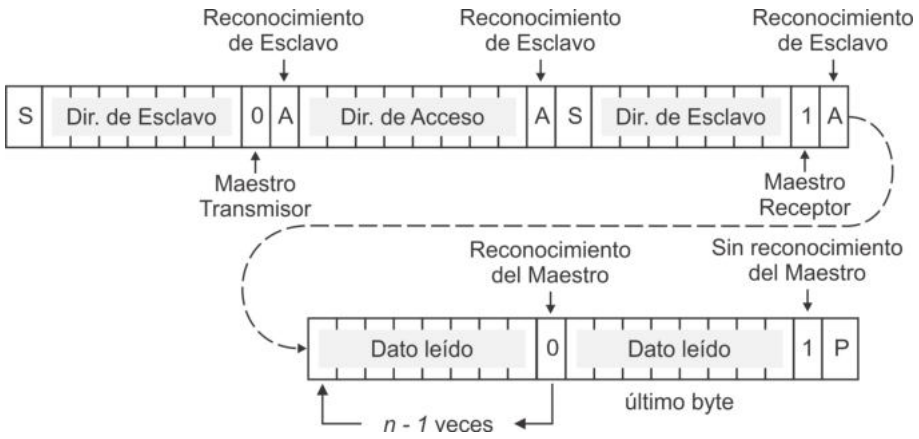


Figura 8.15: Secuencia de lectura en la memoria PCF8570

El microcontrolador va a ser maestro, por ello, nuevamente se utiliza la biblioteca `TWI.h` para simplificar la estructura de las funciones. Las funciones van a regresar 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. En las funciones no se configura la frecuencia, sería parte del programa principal y debe tomarse en cuenta que la memoria puede operar con una frecuencia máxima de 100 kHz.

La función para la escritura de datos en la RAM sigue la secuencia mostrada en la Figura 8.14, su código es:

```
uint8_t escribe_RAM(uint8_t datos[], uint8_t n, uint8_t  dir) {
uint8_t i, edo;

   edo = TWI_Inicio();                  // Condición de inicio
   if(edo != 0x01)  {                   // Si no se consigue
      TWCR |= (1<<TWINT);               // Limpia la bandera
      return 0x00;
   }
   edo = TWI_EscByte(DIR_ESVO << 1);    // Transmite la SLA+W
   if(edo != 0x01) {                    // Ocurre algo inesperado
      TWI_Paro();                       // Libera al bus
      return 0x00;
   }
   edo = TWI_EscByte(dir);              // Transmite la dirección
   if(edo != 0x01) {                    // Ocurre algo inesperado
      TWI_Paro();                       // Termina las transferencias
      return 0x00;
   }
   for( i = 0; i < n; i++ ) {           // Transmite los datos
      edo = TWI_EscByte(datos[i]);
      if(edo != 0x01) {                 // Ocurre algo inesperado
         TWI_Paro();                    // Termina las transferencias
         return 0x00;
      }
```

```
    }
    TWI_Paro();

    return 0x01;                          // Regresa con éxito
}
```

Se observa que si no se consigue la condición de INICIO, solo se debe limpiar la bandera **TWINT**, no se puede establecer una condición de PARO porque el MCU no llegó a ser maestro. En las llamadas subsecuentes el MCU ya es maestro, de manera que si ocurre un estado inesperado, debe establecer una condición de PARO y regresar con 0x00 indicando que la escritura en la RAM no pudo concluir con éxito. Cuando el procedimiento termina de manera normal, se establece la condición de PARO y la función regresa 0x01.

Antes de la llamada a la función `escribe_RAM()`, en el programa principal se debe validar que no se supere el espacio disponible, es decir, se debe garantizar que `dir + n` es menor que 256.

La función de lectura de datos en la RAM debe seguir el comportamiento de la Figura 8.15, su código es:

```
uint8_t lee_RAM(uint8_t datos[], uint8_t n, uint8_t dir) {
uint8_t i, edo;

    edo = TWI_Inicio();                   // Condición de inicio
    if(edo != 0x01) {                     // Si no se consigue
        TWCR |= (1<<TWINT);               // Limpia la bandera
        return 0x00;
    }
    edo = TWI_EscByte(DIR_ESVO << 1);     // Transmite la SLA+W
    if(edo != 0x01) {                     // Ocurre algo inesperado
        TWI_Paro();                       // Libera al bus
        return 0x00;
    }
    edo = TWI_EscByte(dir);               // Transmite la dirección
    if(edo != 0x01) {                     // Ocurre algo inesperado
        TWI_Paro();                       // Termina las transferencias
        return 0x00;
    }
    edo = TWI_Inicio();                   // Inicio repetido
    if(edo != 0x01) {                     // Si no se consigue
        TWCR |= (1<<TWINT);               // Limpia la bandera
        return 0x00;
    }
    edo = TWI_EscByte((DIR_ESVO << 1)+1); // Transmite la SLA+R
    if(edo != 0x01) {                     // Ocurre algo inesperado
        TWI_Paro();                       // Libera al bus
        return 0x00;
    }
```

```
    for( i = 0; i < n - 1; i++ ) {          // Recibe n - 1 datos
        edo = TWI_LeeByte(&datos[i], 1);    // Lee con reconocimiento
        if(edo != 0x01) {                   // Ocurre algo inesperado
            TWI_Paro();                     // Termina las transferencias
            return 0x00;
        }
    }
    TWI_LeeByte(&datos[i], 0);              // Lee el último dato
                                            // sin reconocimiento
    TWI_Paro();                             // Condición de paro

    return 0x01;                            // Regresa con éxito
}
```

También se incluyen las condiciones de seguridad, se limpia la bandera `TWINT` si no se consigue la condición de INICIO y se establece una condición de PARO cuando ocurre un estado inesperado. De manera similar, antes de la llamada a la función `lee_RAM()`, en el programa principal se debe validar que la lectura se hará en un espacio válido.

Ejemplo 8.3 - Manejo de la memoria PCF8570 desde la USART
En la Figura 8.16 se muestra la conexión de una memoria PCF8570 con un ATMega328P, también se observa la disponibilidad de un puerto de comunicación serial a través de la USART. Realice un programa que facilite leer algunos datos de la memoria PCF8570 y enviarlos por la USART, así como recibir una secuencia de caracteres desde la USART para escribirlos en la memoria PCF8570.

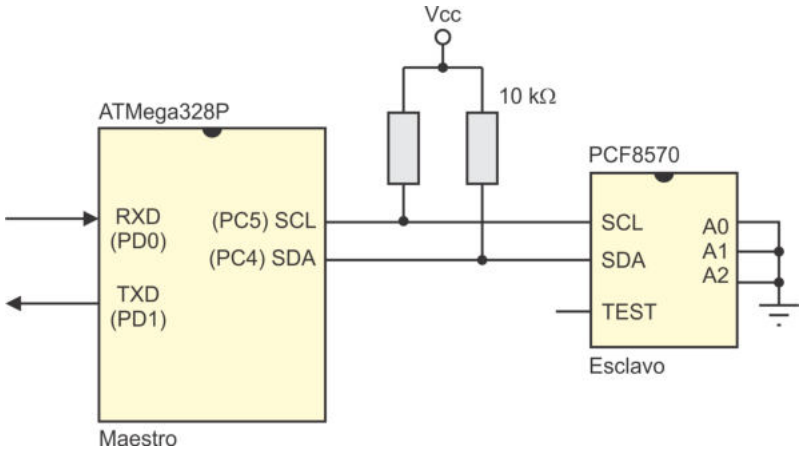


Figura 8.16: Conexión de una memoria PCF8570 con un ATmega328P

La memoria puede responder como esclavo a una dirección configurable entre 0x50 y 0x57, pero como se puede ver en la Figura 8.16, sus terminales A0, A1 y A2 se han conectado a tierra, de manera que su dirección como esclavo es la 0x50.

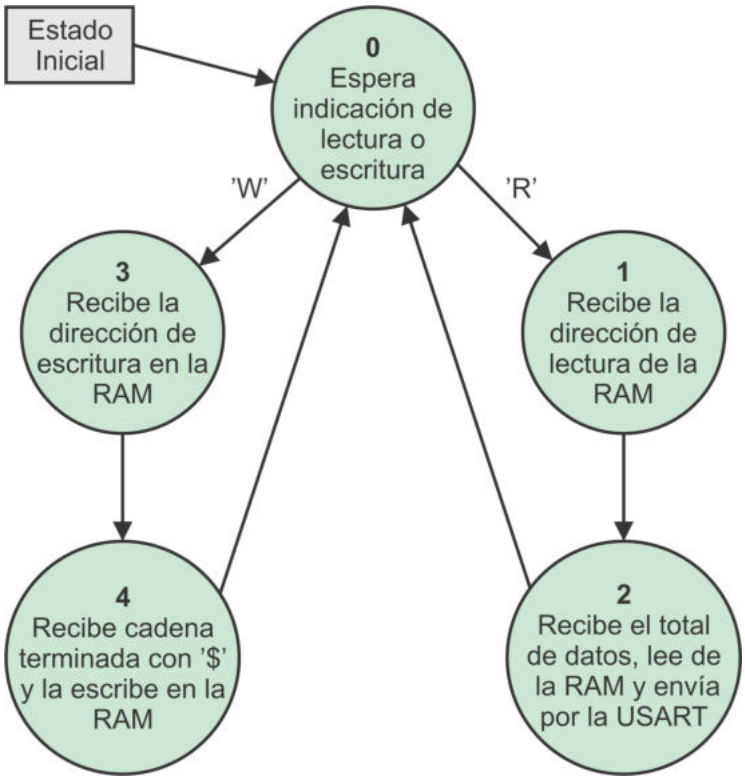


Figura 8.17: Comportamiento del MCU al recibir caracteres a través de la USART

Dado que el manejo de la memoria se hará desde la USART, el MCU va a configurarse para que sea interrumpido cada que llega un carácter serial, el cual debe interpretarse adecuadamente de acuerdo con la operación en progreso. El comportamiento de la ISR de recepción se puede describir con un diagrama de estados, en la Figura 8.17 se muestra de manera general este comportamiento. Para simplificar el programa se van a utilizar las funciones `escribe_RAM()` y `lee_RAM()` del ejercicio anterior.

El estado de la recepción debe manejarse por medio de una variable global, esta debe iniciar en 0. Con la llegada de un nuevo carácter, a través de la USART, este se evalúa según el estado actual y dependiendo de su contenido, se podrá complementar la información requerida en ese estado o se podrá cambiar de estado. En los estados 2 y 4 se realizan los accesos a la RAM, antes de ello, el programa se debe garantizar que se tendrá acceso a un espacio válido.

Es necesario el uso de otras variables globales, una para recibir la dirección de acceso en la RAM, otra para el total de caracteres a leer o escribir, y un arreglo para colocar la cadena de caracteres que se recibe de la USART para escribirse en la RAM o que se lee de la RAM para enviarse por la USART.

La comunicación con el ATMega328P se hará desde una terminal serial, por lo que es conveniente que desde el programa se envíen mensajes para que el usuario conozca

el estado del sistema. El código con la solución al ejercicio es:

```
#define  DIR_ESVO  0x50

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

void envia_cadena(char cad[]);
uint8_t escribe_RAM(uint8_t datos[], uint8_t n, uint8_t dir);
uint8_t lee_RAM(uint8_t datos[], uint8_t n, uint8_t dir);

uint8_t estado = 0;
// 0 - Sin actividad, espera comando
// 1 - Recibe dirección de lectura
// 2 - Recibe el total de datos, lee la RAM y envía datos
// 3 - Recibe dirección de escritura
// 4 - Recibe cadena terminada con '$' y escribe en la RAM

uint16_t dir, total;
uint8_t  datos[64], i;

ISR(USART_RX_vect) {
uint8_t dato;

  dato = UDR0;
  switch( estado ) {
   case 0: dir = 0;                                  // Espera indicación
           if(dato == 'R' || dato == 'r')
              estado = 1;                            // Lectura
           else if(dato == 'W' || dato == 'w')
              estado = 3;                            // Escritura
           else
              envia_cadena("Comando no identificado");
           break;
   case 1: if(dato >= 0x30 && dato <= 0x39)          // Carácter numérico
              dir = (dir*10) + (dato-0x30);          // Forma dirección
           else {                                    // Otro carácter
              if(dir > 255) {
                 envia_cadena("Direccion fuera de rango");
                 estado = 0;
              }
              else {
                 total = 0;
                 estado = 2;                         // Dirección lista
              }
           }
           break;
   case 2: if(dato >= 0x30 && dato <= 0x39)          // Carácter numérico
              total = (total*10) + (dato-0x30);      // Forma el total
           else {                                    // Otro carácter
              if(dir + total > 255)
```

```
                envia_cadena("Lectura fuera de rango");
            else {
                if(lee_RAM(datos, total, dir)){ // Lectura por TWI
                    datos[total] = 0;           // Fin de cadena
                    envia_cadena(datos);        // Lectura con éxito
                }
                else
                    envia_cadena("Falla en la lectura");

            }
            estado = 0;
        }
        break;
  case 3: if(dato >= 0x30 && dato <= 0x39)      // Carácter numérico
            dir = (dir*10) + (dato-0x30);       // Forma dirección
        else {                                  // Otro carácter
            if(dir > 255) {
                envia_cadena("Direccion fuera de rango");
                estado = 0;
            }
            else {
                i = 0;                          // Indice de repepción
                estado = 4;                     // Dirección lista
            }
        }
        break;
  case 4: if( dato != '$') {                    // Lee caracteres
            datos[i] = dato;                    // hasta el fin de
            i++;                                // la cadena
        }
        else {
            if((dir + i) > 255)
                envia_cadena("Escritura fuera de rango");
            else {
                if(escribe_RAM(datos, i, dir))  // Escritura por TWI
                    envia_cadena("Exito en la escritura");
                else
                    envia_cadena("Falla en la escritura");
            }
            estado = 0;
        }
        break;
  default: estado = 0;
        envia_cadena("Estado inesperado");
  }
}

int main() {

  DDRD = 0x02;          // RXD es entrada y TXD es salida
  TWI_Config();
```

```
    // Configuración de la USART
    UBRR0  = 0X0C;          // 9600 bps
    UCSR0A = 0X02;          // Asíncrono a doble velocidad
    UCSR0B = 0X98;          // Interrupción por recepción
    UCSR0C = 0X06;          // Datos de 8 bits

    sei();

    while(1)
        asm("NOP");
}

void envia_cadena(char cad[]) {
uint8_t  j = 0;

    while(cad[j] != '\0') {
        while(!(UCSR0A & 1 << UDRE0));    // Asegura buffer vacío
        UDR0 = cad[j];
        j++;
    }
    // Envía el retorno de carro
    while(!(UCSR0A & 1 << UDRE0));        // Asegura buffer vacío
    UDR0 = 0x0D;
}
```

Si se compara el código con la Figura 8.17 se podrán relacionar las acciones que se planean en el diagrama de flujo con la forma en que se ejecutan. La función `envia_cadena()` le da mucha versatilidad al sistema al notificar al usuario sobre las situaciones que se presentan durante la ejecución, esta función fue utilizada en el Ejemplo 7.3, solo se le agregó el envío de un retorno de carro para mejorar la presentación en la terminal serial receptora.

El MCU envía mensajes indicando si hubo fallas en la lectura o escritura de la RAM, pero no se describe la falla, esta se puede conocer con precisión analizando los códigos de estado, pero significa modificar las funciones de acceso a la RAM y evaluar cada uno de los diferentes valores de retorno.

Comprender el manejo de una RAM es fundamental para manipular otros dispositivos con interfaz I^2C, porque generalmente estos dispositivos incluyen registros de configuración y su acceso es similar a la escritura y lectura de datos en una memoria RAM.

8.9. La Interfaz TWI en Arduino

Arduino cuenta con la biblioteca `Wire.h` para manejar dispositivos desde un *sketch* a través de la interfaz TWI, la biblioteca incluye un objeto predefinido denominado `Wire`.

El objeto `wire` facilita la operación del MCU a través de 10 métodos, los cuales son:

- **begin():** Habilita al recurso y configura la velocidad de la interfaz a 100 kHz, opcionalmente puede recibir un argumento que corresponde con su dirección como esclavo. El argumento se omite cuando el MCU solo será maestro.
- **setClock():** Con este método se ajusta la velocidad de la interfaz TWI al argumento recibido en Hz.
- **beginTransmission():** Establece la condición de inicio y direcciona a un esclavo para escritura (SLA+W), como argumento recibe la dirección del esclavo.
- **write():** Envía uno o varios bytes a través de la interfaz TWI, esta función se puede usar cuando el MCU es maestro o esclavo (modos MT o ST).
- **endTransmission():** Establece una condicion de paro cuando el dispositivo está en modo maestro transmisor (MT).
- **requestFrom():** Si el MCU va a trabajar como maestro receptor (MR), la petición de datos al esclavo la realiza con este método. El método recibe dos argumentos obligatorios y uno opcional, el primero es la dirección del esclavo, el segundo corresponde con la cantidad de bytes a leer y el tecero es una variable booleana que determina si habrá o no condición de paro. El método realiza todo el proceso: establece la condición de inicio, envía la SLA+R, lee los bytes que se han solicitado y si el tercer argumento se omite o es `True`, establece la condición de paro. Cuando el tercer argumento es `False` el bus no se libera, esto para que el MCU pueda cambiar al modo maestro transmisor o direccionar a otro esclavo. El resultado de la lectura queda en un *buffer* de memoria al que se tiene acceso por medio de otros dos métodos.
- **available():** Indica la cantidad de bytes disponibles en el *buffer* TWI, los datos llegan al *buffer* después de la llamada al método `requestFrom()`.
- **read():** Obtiene un byte del *buffer* TWI.
- **onReceive():** Con este método se registra una función que será invocada cuando un MCU como esclavo receptor reciba algunos bytes de un maestro transmisor. En la función se deben emplear los métodos `available()` y `read()` para obtener los bytes leídos.
- **onRequest():** Con este método se registra una función que será invocada cuando a un MCU como esclavo trasmisor le soliciten algunos bytes desde un maestro receptor. En la función se debe emplear el método `write()` para enviar los bytes solicitados.

Los últimos dos métodos son empleados cuando un MCU va a operar en modo esclavo, se registran las funciones porque un esclavo no sabe cuando va recibir una petición de un maestro, de esta manera, la atención se da por interrupción.

Para ilustrar el uso de las funciones de Arduino, se reescribe la solución del Ejemplo 8.1, en donde un MCU es maestro transmisor y cuando el usuario presiona un botón, envía un dato a uno de tres esclavos (están en modo esclavo receptor), el esclavo se selecciona por medio de dos interruptores. En la Figura 8.13 se mostró el diagrama original, sin embargo, en Arduino no se puede emplear el Puerto B completo porque utiliza un oscilador externo que se ubica en las terminales PB6 y PB7.

Por ello, para el maestro se ha cambiado el uso de los puertos, del Puerto D se lee el dato a enviar y, del Puerto B se obtiene el número de esclavo y el estado del botón, el cual se sondea en el lazo infinito.

El *sketch* para el MCU maestro es:

```
#include <Wire.h>

void setup() {
  DDRD = 0x00;      // Dato a enviar
  PORTD = 0xFF;     // Pull-up
  DDRB = 0x00;      // Dirección y habilitación
  PORTB = 0x07;     // Pull-up

  Wire.begin();     // Habilita como maestro
}

void loop() {
uint8_t  esclavo, dato;

  if( digitalRead(10) == LOW) {       // Botón presionado
    esclavo = PINB & 0x03;            // Dirección del esclavo
    dato = PIND;
    Wire.beginTransmission(esclavo); // Inicio y SLA+W
    Wire.write(dato);                 // Dato
    Wire.endTransmission();           // Paro
    delay(300);
  }
}
```

En el esclavo no hay cambios en el hardware porque del Puerto B solo se utilizan dos terminales para definir la dirección del MCU como esclavo, quedando libres las terminales para el oscilador externo. El *sketch* para el MCU esclavo es:

```
#include <Wire.h>

void setup() {
uint8_t   dir;

  DDRB = 0x00;                  // Entrada para la dirección
  PORTB = 0x03;                 // Pull-Up
  DDRD = 0xFF;                  // Salida para el dato

  dir = PINB & 0x03;            // Lee dirección del esclavo
```

```
  Wire.begin(dir);              // Habilita la interfaz
  Wire.onReceive(Rec_TWI);      // Registra la función para recepción
}

void loop() {
  asm("NOP");                   // Ocioso en el lazo infinito
}

void Rec_TWI() {
  if(Wire.available())          // Si hay un dato disponible
    PORTD = Wire.read();        // lo lee y envía al Puerto D
}
```

En el programa del maestro se observa una secuencia similar a la que se utiliza en la solución original, por lo que es necesario comprender la operación de la interfaz para utilizar la biblioteca `Wire.h`. Sin embargo, el programa del esclavo es muy diferente al original, con las funciones no es necesario revisar los códigos de estado de la interfaz.

Una limitante para el esclavo es que no se habilita la atención a las llamadas generales, con el sistema funcionando el maestro no obtiene un reconocimiento al enviar un dato a la dirección 0. Para resolver esta limitante, la inicialización de la interfaz se debe realizar de la siguiente manera:

```
  Wire.begin(dir);              // Habilita la interfaz
  TWAR |= 0x01;                 // Habilita las llamadas generales
  Wire.onReceive(Rec_TWI);      // Registra la función para recepción
```

Lo que significa que el programador debe conocer la organización de los Registros I/O relacionados con la interfaz TWI.

La ventaja de Arduino es que, similar a lo que ocurre con la interfaz SPI, algunos desarrolladores de *shields* con interfaz I^2C también comparten su biblioteca de funciones, de esta forma, el uso de los *shields* se simplifica y los mecanismos para su acceso pasan desapercibidos por el usuario, quien puede desconocer completamente la operación de la interfaz TWI. Algunos ejemplos son:

- **Relojes de tiempo real:** Comercialmente existen módulos para el manejo de un reloj-calendario, basados en los chips DS1307, DS3231 o PCF8523. Además del chip RTC, los módulos incluyen los elementos requieridos para su operación, así como una batería para mantener la información actualizada aún en ausencia de energía. La biblioteca `RTCLib.h` desarrollada por la empresa Adafruit incluye las funciones necesarias para leer y ajustar todos los parámetros de la fecha y hora en los tres diferentes chips.
- **Display de Cristal Líquido por I2C:** El manejo de un LCD requiere de 4 u 8 terminales para los datos y 3 para las señales de control. El número de terminales se reduce significativamente con el uso de un adaptador de LCD a I^2C,

estos módulos se basan en el chip PCF8574, que es un expansor de entradas y salidas digitales. Aunque el hardware tiene una funcionalidad simple, se han desarrollado bibliotecas como la `LiquidCrystal_I2C.h`, que incluye funciones para la inicialización y escritura de caracteres en un LCD, así como para la activación de características especiales como el cursor o parpadeo. De esta forma, un LCD se conecta al MCU solo con dos terminales y su manipulación por medio de la biblioteca se simplifica tanto que el desarrollador no requiere conocimientos de la interfaz TWI ni de las funciones de un LCD.

- **Sensores de temperatura y humedad:** Existe una versatilidad de sensores de temperatura y humedad compatibles con la interfaz I^2C que pueden cubrir con los requerimientos de un sistema, dos ejemplos de ello son: el módulo AM2315 y el módulo Si7021. El primero tiene un rango de humedad relativa de 0-100 % con una precisión del 2 % y un rango de temperatura de -20 a 80°C de con una precisión de 0.1°C. En el segundo, su rango de humedad relativa es de 0-80 % con una precisión del 3 % y su rango de temperatura es de -10 a 85°C con una precisión de 0.4°C. Ambos módulos cuentan con bibliotecas para su manejo desde el entorno Arduino, estas son `Adafruit_AM2315.h` y `Adafruit_Si7021.h`, respectivamente, sus funciones permiten la lectura de cualquiera de los parámetros, de manera que en ningún momento el usuario se involucra con los registros de la interfaz TWI.

Estos ejemplos demuestran el extenso uso de la interfaz TWI, Arduino y sus bibliotecas pueden ser un buen punto de partida para evaluar el funcionamiento de diversos módulos con esta interfaz. Pero para comprender las bibliotecas desarrolladas por terceros es necesario entender el funcionamiento de la interfaz al nivel de registros, de esta forma, se podrán implementar las funciones necesarias para cubrir con los requerimientos de un sistema.

8.10. Ejercicios

En los siguientes ejercicios suponga que el MCU estará operando a 1 MHZ, todos pueden resolverse en lenguaje C y puede emplearse la biblioteca `TWI.h` si el MCU va a operar como maestro.

1. 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 basado en un LED y 5 botones. Con el LED 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 esclavos y el maestro sea el colector de los votos. Después de una decisión, el maestro debe enviar 5 pares de datos a través de su USART, cada par indica cuantos votos se tuvieron en cada una de las opciones. Proponga el hardware y desarrolle el software, del maestro y de los esclavos.

2. En el Ejemplo 8.3 se manejó una memoria RAM PCF8570 de 256 x 8. Modifique el programa considerando que se han conectado 4 memorias para crear una memoria de 1k x 8, como se muestra en la Figura 8.18. Observe que es necesario modificar las funciones de Escritura y Lectura de la memoria, y que solo se va a tener acceso a 2 memorias consecutivas porque el límite se ha establecido en 64 caracteres.

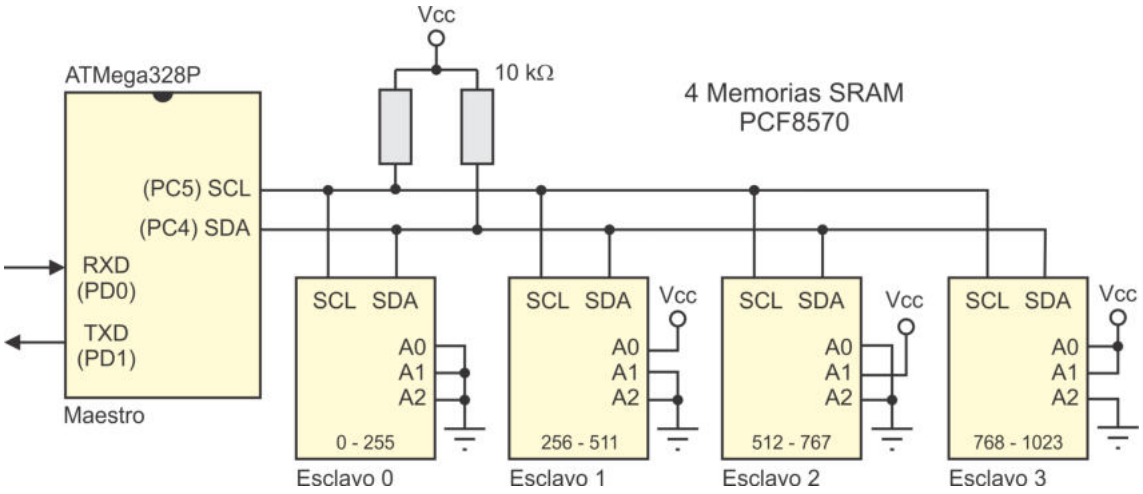


Figura 8.18: Manejo de 4 memorias PCF8570 desde un ATMega328P

3. Se quiere construir un sistema para el manejo de 3 persianas empleando motores unipolares de pasos. El sistema debe incuir un maestro y 3 esclavos, comunicándose bajo la interfaz TWI, como se muestra en la Figura 8.19.

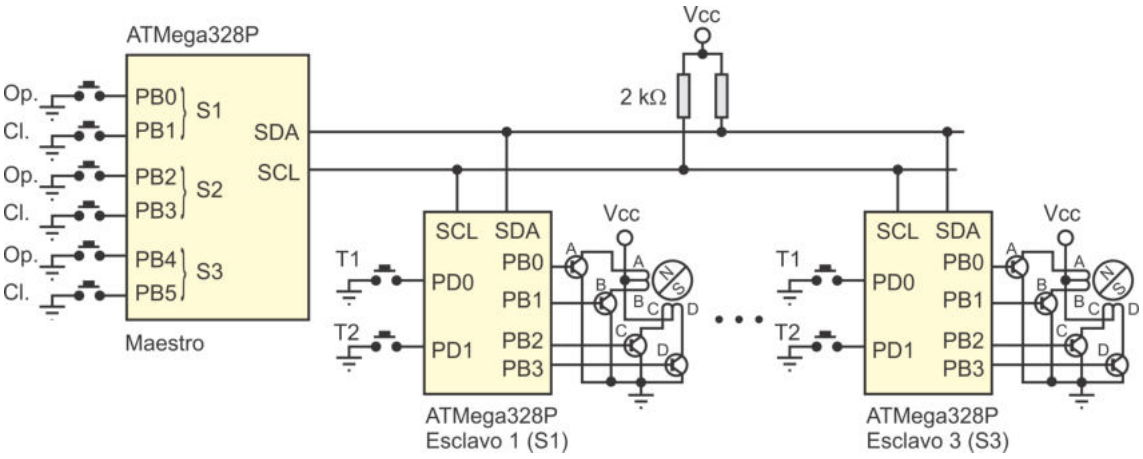


Figura 8.19: Sistema para el manejo de 3 persianas

El maestro sondeará los botones cada 300 ms y en caso de que alguno esté presionado, debe enviar un comando al esclavo que corresponda (S1, S2 o S3).

Con 0x33 le indicará que abra y con 0xCC que cierre. No se enviará comando alguno cuando se detecten 2 o más botones presionados. Los esclavos van a abrir o cerrar las persianas con motores de pasos, con detectores de tope (T1 y T2) sabrán si pueden seguir girando. Un esclavo generará 5 medios pasos en orden creciente cuando reciba 0x33 y T1 está abierto, y 5 medios pasos en orden decreciente si recibe 0xCC y T2 está abierto.

Realice el programa para el maestro y para el esclavo 1 e indique qué cambios debe haber para los esclavos 2 y 3. Designe las direcciones a los esclavos que considere convenientes.

4. El circuito DS1307 es un Reloj/Calendario de Tiempo Real (RTC), con interfaz I2C 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 8.14 y 8.15. En la Tabla 8.9 se muestran sus registros internos, se puede ver que la hora y fecha están en BCD.

Tabla 8.9: Registros del circuito DS1307

Dir.	**Bit 7**	**Bit 6**	**Bit 5**	**Bit 4**	**Bit 3**	**Bit 2**	**Bit 1**	**Bit 0**	**Función**	**Rango**
00h	CH	Decenas de segundos			Unidades de segundos				Segundos	00-59
01h	0	Decenas de minutos			Unidades de minutos				Minutos	00-59
02h	0	12	PM/AM	Dec. hora	Unidades de hora				Horas	1-12
		24	Dec. de hora							00-23
03h	0	0	0	0	0	Día (semana)			Día	01-07
04h	0	0	Dec. día (mes)		Unidades del día (mes)				Día (mes)	01-31
05h	0	0	0	Dec. mes	Unidades del mes				Mes	1-12
06h	Decenas del año				Unidades del año				Año	00-99
07h	OUT	0	0	SQWE	0	0	RS1	RS0	Control	-
08h-3Fh	56 localidades de 8 bits								RAM	00h-FFh

Apoyados en la biblioteca `TWI.h`, desarrolle las siguientes funciones:

- **lee_hora(uint8_t hora[ ]):** para leer la hora, incluyendo hora, minutos y segundos, colocando la información en un arreglo.
- **esc_hora(uint8_t hora[ ]):** para escribir la hora, recibiendo un arreglo de caracteres con la hora, minutos y segundos.
- **lee_fecha(uint8_t fecha[ ]):** para leer la fecha, incluyendo día, mes y año, colocando la información en un arreglo.
- **esc_fecha(uint8_t fecha[ ]):** para escribir la fecha, recibiendo un arreglo de caracteres con el día, mes y año.

En la Tabla 8.9 se puede ver que el circuito DS1307 incluye una memoria RAM de 56 localidades y un registro de control, en el que se configura al hardware para generar una señal cuadrada a diferentes frecuencias. En las funciones a desarrollar se va a ignorar la RAM y el registro de control. El bit CH de la

dirección 00h sirve para detener al reloj (*clock halt*), este se debe mantener en bajo al escribir la nueva hora.

Los arreglos que reciben las funciones como argumentos contendrán secuencias de caracteres ASCII, como parte de la función se debe realizar la conversión de ASCII a BCD en las escrituras y de BCD a ASCII en las lecturas.

5. En la Figura 8.20 se muestra la conexión de un circuito DS1307 con un ATMega328P, también se observa la disponibilidad de un puerto de comunicación serial a través de la USART. Utilizando las funciones desarrolladas en el ejercicio anterior, realice un programa para que el MCU pueda dar respuesta a diferentes comandos recibidos desde la USART, los comandos son:

   - **H:** el MCU debe enviar la hora completa (hh:mm:ss), obtenida del RTC.
   - **F:** el MCU debe enviar la fecha (dd/mm/aa) obtenida del RTC.
   - **T:** el MCU debe ajustar la hora en el RTC, después de recibir **T** debe esperar una secuencia de 6 caracteres con una hora válida.
   - **D:** el MCU debe ajustar la fecha en el RTC, después de recibir **D** el sistema espera una secuencia de 6 caracteres con una fecha válida.

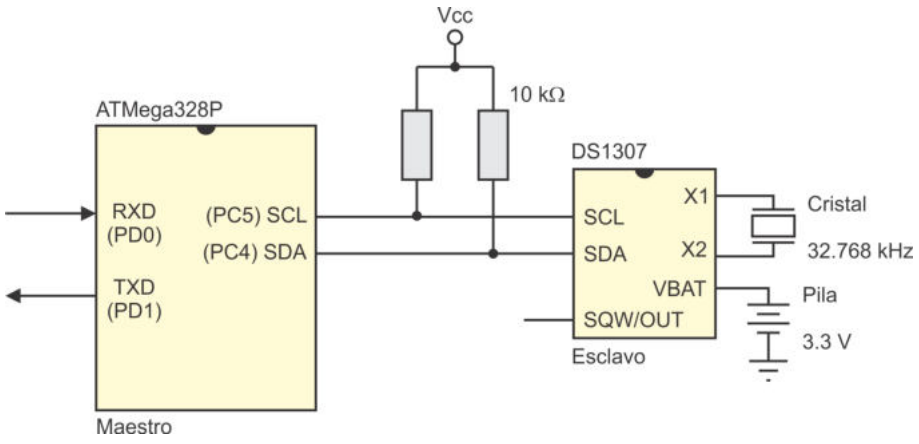


Figura 8.20: Conexión de un circuito DS1307 con un ATMega328P

6. Construya una marquesina para exhibir mensajes, con 8 matrices de 5 x 7 LEDs. Utilice una red TWI con 1 maestro y 8 esclavos, todos ellos con base en microcontroladores ATMega328P. Un esclavo será un controlador de una matriz de LEDs y recibirá el código ASCII del carácter a mostrar, encendiendo los LEDs a partir de una tabla de constantes. El usuario va a enviar la cadena a exhibir y su tamaño desde una PC al maestro, quien la recibe a través de la USART. Para un desplazamiento del mensaje de derecha a izquierda, el maestro continuamente debe modificar el carácter de cada esclavo.

Capítulo 9

Recursos Especiales

En este capítulo se explica como utilizar 5 recursos especiales incluidos en el microcontrolador ATMega328P. Con estos recursos se puede complementar la funcionalidad de algunas aplicaciones, se consideran:

- Los modos de reposo o bajo consumo.
- El perro guardián (WDT, *watchdog timer*).
- La sección de arranque en la memoria Flash.
- Los Bits de Configuración y Seguridad.
- La interfaz *debugWire*.

En el Capítulo 2 ya se hizo referencia a algunos de estos recursos, al describir la organización de los microcontroladores AVR, sin embargo, no se mostró su manejo porque aún no se revisaban los aspectos de programación.

9.1. Modos de bajo consumo

En la Sección 2.9 se describieron los diferentes modos de reposo o bajo consumo. Un ATMega328P puede ser llevado a cualquiera de estos modos si su aplicación utiliza interrupciones, porque es a través de estos eventos cuando el microcontrolador va a despertar para ejecutar una tarea, posterior a la cual podrá nuevamente ser ingresado al modo de bajo consumo. Para que el MCU ingrese a alguno de los modos se debe:

1. Configurar al Registro I/O `SMCR` (*Sleep Mode Control Register*), seleccionando el modo requerido y poniendo en alto al habilitador del modo de reposo.
2. Ejecutar la instrucción `SLEEP`, esta es la última instrucción que el MCU va a ejecutar antes de ingresar al modo de reposo elegido.

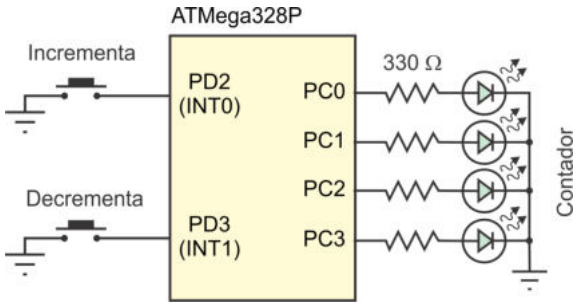


Figura 9.1: Contador para evaluar los modos de bajo consumo

Cuando ocurra una interrupción, el MCU va a "despertar" y esperar los ciclos de reloj que comprenda su tiempo de establecimiento (en la Sección 2.8 se indican los tiempos de establecimiento para las diferentes fuentes de reloj). Concluido el tiempo de establecimiento se ejecutará el código incluido en la ISR. Posterior a ello, el MCU regresará al programa principal, en donde va a continuar su ejecución con la instrucción posterior a la instrucción `SLEEP`. Esto significa que el MCU ya no está en el modo de reposo, si se quiere regresar al modo previamente seleccionado, nuevamente deberá ejecutarse la instrucción `SLEEP`.

Ejemplo 9.1 - Evaluación de los modos de reposo
Realice un contador de eventos de 0 a 15, con salida en binario, configurando para que el MCU esté en un modo de reposo mientras no ocurra un evento. Ubique la salida del contador en el Puerto C y atienda los eventos por interrupciones externas, una para incrementar el contador y otra para decrementarlo. En la Figura 9.1 se muestra el hardware requerido.

Antes del ciclo infinito se debe configurar el modo de reposo, al cual se va a ingresar con la primera instrucción del ciclo, para que después de la ejecución de las ISRs, el MCU ingrese nuevamente a un estado de reposo en la siguiente iteración del lazo infinito.

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

uint8_t   conta = 0;

ISR(INT0_vect) {
   conta = (conta < 15)?conta + 1:0;
   PORTC = conta;
}

ISR(INT1_vect) {
   conta = (conta > 0)?conta - 1:15;
   PORTC = conta;
}
```

```
int main(void) {
   DDRD = 0x00;          // Entradas
   PORTD = 0x0C;         // Pull-Up en INT0 e INT1
   DDRC = 0xFF;

   EICRA = 0x0A;         // INT0 e INT por flanco de bajada
   EIMSK = 0x03;         // Habilita INT0 e INT1

   PORTC = 0;            // Valor inicial de la salida

   SMCR = 0x01;          // Modo ocioso
// SMCR = 0x03;          // ADC con bajo ruido
// SMCR = 0x05;          // Potencia baja
// SMCR = 0x07;          // Ahorro de Energía
// SMCR = 0x0D;          // Espera
// SMCR = 0x0F;          // Espera extendido

   sei();                // Habilitador global de interrupciones
   while (1) {
      asm("SLEEP");      // Ingresa al modo de bajo consumo
      asm("NOP");
   }
}
```

En el código se muestra la selección del modo ocioso, no obstante, mediante comentarios se indican los valores requeridos en el registro `SMCR` para los otros modos.

En la Tabla 9.1 se presentan resultados obtenidos de forma práctica sobre el consumo de corriente, en los diferentes modos de reposo. Es posible evaluar los seis modos disponibles porque las interrupciones externas son eventos que van a despertar al MCU sin importar el modo elegido.

Tabla 9.1: Comparación de los modos de bajo consumo

Modo	**Conta = 0**	**Conta = 15**
Normal (sin bajo consumo)	6.34 mA	45.51 mA
Modo ocioso	4.28 mA	44.68 mA
ADC con bajo ruido	4.20 mA	44.63 mA
Potencia baja	0.4 uA	40.61 mA
Ahorro de energía	0.4 uA	41.13 mA
Modo de espera	96.3 uA	40.85 mA
Modo de espera extendido	96.3 uA	40.20 mA

Para propósitos de comparación, en la Tabla 9.1 se incluyen resultados cuando el ATMega328P está operando solo, porque el contador vale 0 y todos los LEDs están apagados, así como cuando el contador tiene 15 y, por lo tanto, los 4 LEDs están encendidos.

Puesto que el consumo en corriente es debido a los LEDs, si se quiere minimizar, en las ISRs se podría mostrar el valor del contador sólo durante un breve instante de tiempo para después apagar los LEDs. El valor del contador no se pierde porque el contenido de las variables y los registros se conserva cuando un MCU ingresa a cualquier modo de reposo.

Los resultados prácticos demuestran el gran ahorro de energía que se tiene al utilizar los modos de reposo, sobre todo en los modos de Potencia baja y de Ahorro de energía.

9.2. El perro guardián (WDT, *watchdog timer*)

El perro guardián o WDT (por *watchdog timer*) es un temporizador manejado por un oscilador interno independiente con una frecuencia de 128 kHz, cuando el MCU está alimentado con 5 V. El WDT de un ATMega328P tiene dos modos de operación, el clásico modo de reinicio y un modo de interrupción, es decir, cuando el WDT desborda puede provocar el reinicio del MCU y también puede provocar una interrupción que dará paso a la ejecución de una ISR, para atender al evento. En la Tabla 2.1 se puede ver que el vector número 7 corresponde con el evento de desbordamiento del WDT, el vector está ubicado en la dirección 0x00C.

En la Figura 9.2 se muestra la organización del WDT, a través de un preescalador es posible conseguir diferentes periodos de desbordamiento, su duración se establece con los bits `WDP[3:0]` del Registro de Control del WDT (`WDTCSR`, *Watchdog Timer Control Register*). En la Tabla 9.2 se muestran los periodos de desbordamiento para las diferentes opciones del preescalador. La bandera del WDT (`WDF`), el habilitador del recurso (`WDE`) y el habilitador de su interrupcón (`WDIE`), presentes en la Figura 9.2, también son parte del registro `WDTCSR`.

Tabla 9.2: Factores de preescala del *watchdog timer*

WDP3	**WDP2**	**WDP1**	**WDP0**	**No. de ciclos**	**Periodo**
0	0	0	0	2k (2048)	16 ms
0	0	0	1	4k (4096)	32 ms
0	0	1	0	8k (8192)	64 ms
0	0	1	1	16k (16384)	128 ms
0	1	0	0	32k (32768)	256 ms
0	1	0	1	64k (65536)	512 ms
0	1	1	0	128k (131072)	1.024 s
0	1	1	1	256k (262144)	2.048 s
1	0	0	0	512k (524288)	4.096 s
1	0	0	1	1024k (1048576)	8.192 s
Otras combinaciones				Reservadas	

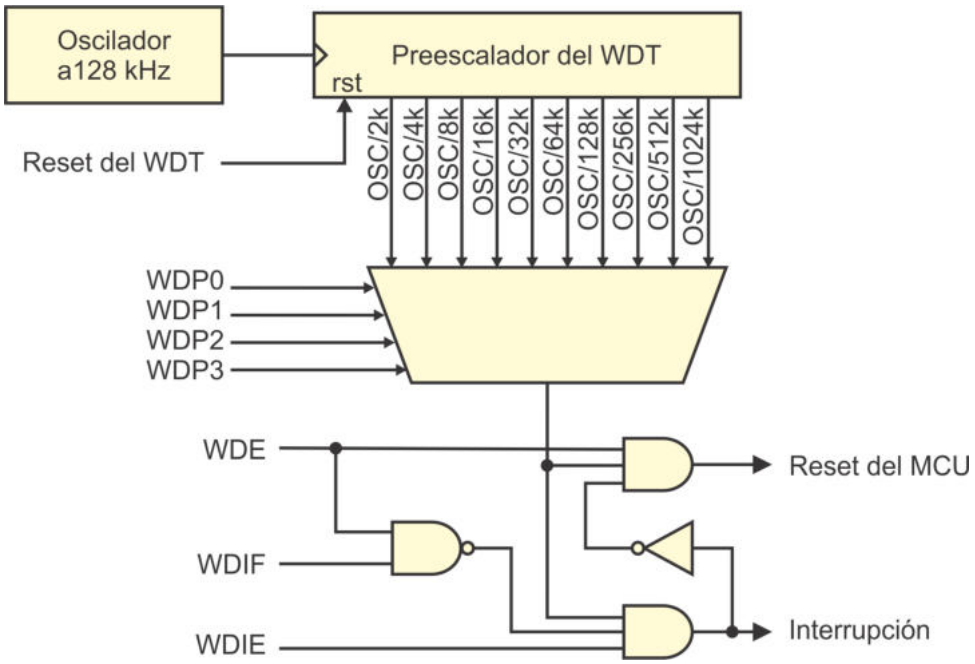


Figura 9.2: Organización del *watchdog timer*

El modo de reinicio es el típico modo en donde el MCU se va a reiniciar si el WDT desborda, para que esta situación no ocurra, en posiciones estratégicas del programa se debe colocar la instrucción `WDR` para aplicar un *reset* al WDT. Este modo es empleado para evitar que el MCU se quede ciclado en estados no deseados, por ejemplo, cuando un sistema queda en espera de un dato proporcionado por el usuario o proveniente de un sensor y el dato no llega.

El modo de interrupción hace que el WDT se pueda emplear como un temporizador de propósito general, agendando tareas que se deseen realizar de manera periódica. Algunos usos de este modo son: el WDT se puede emplear para liberar periódicamente al MCU de algún modo de bajo consumo, o bien, con el WDT se puede limitar el tiempo de respuesta de algún periférico.

Los dos modos se pueden combinar, de manera que, cuando el WDT desborda genera una interrupción y después de su atención se reinicia al MCU. El modo de operación del WDT se selecciona en el registro `WDTCSR`.

El WDT de un ATMega328P se puede habilitar en dos formas diferentes: Activando al fusible `WDTON` al momento de programar al dispositivo (el fusible es parte de los Bits de Configuración y Seguridad), o bien, modificando los bits del registro `WDTCSR` durante la ejecución de una aplicación. Sin embargo, con la activación del fusible, solo queda activo el modo de reinicio y no se puede emplear el modo de interrupción.

9.2.1. Registro para el Manejo del WDT

El registro de control del WDT es el `WDTCSR`, es un Registro I/O Extendido ubicado en la dirección 0x60, cuyos bits son:

REG.	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
WDTCSR	WDIF	WDIE	WDP3	WDCE	WDE	WDP2	WDP1	WDP0

- **Bit 7 - WDIF:** (*Watchdog Interrupt Flag*) Es una bandera que se pone en alto cuando el WDT desborda estando configurado en el modo de interrupción. La bandera se limpia automáticamente cuando se transfiere la ejecución a la ISR o se puede limpiar por software al reescribirle un 1.
- **Bit 6 - WDIE:** (*Watchdog Interrupt Enable*) Este bit habilita el modo interrupción. Si además, el bit `WDE` está en alto, se combinan los modos de operación, cuando el WDT desborda, primero se produce la interrupción y posteriormente se reinicia al MCU. En la Tabla 9.3 se muestran los distintos modos de operación.
- **Bits 5 y [2:0] - WDP[3:0]:** (*Watchdog Timer Prescaler*) Estos bits determinan el factor de preescala para el reloj del WDT, para modificar su periodo de desbordamiento. En la Tabla 9.2 se mostraron las combinaciones de estos bits para seleccionar el periodo de desbordamiento deseado.
- **Bit 4 - WDCE:** (*Watchdog Change Enable*) Es un habilitador de cambios en la configuración del WDT. A través de una secuencia de seguridad se evitan cambios inesperados, la secuencia requiere que los bits `WDCE` y `WDE` se pongan en alto en una instrucción y a partir de entonces se cuenta con 4 ciclos de reloj para modificar el factor de preescala con los bits `WDP[3:0]` y definir el estado del bit `WDE`.
- **Bit 3 - WDE:** (*Watchdog System Reset Enable*) Con este bit en alto se habilita el modo de reinicio del WDT. Este bit da seguimiento a la bandera `WDRF` del registro `MCUSR`, la bandera se pone en alto cuando el MCU se reinicia por un desbordamiento del WDT. El seguimiento implica que si se quiere limpiar al bit `WDE`, primero se debe limpiar la bandera `WDRF`.

9.2.2. Configuración del WDT

En la Tabla 9.3 se resumen los modos de operación del WDT, los cuales dependen del fusible `WDTON` y de los bits `WDE` y `WDIE`, que son parte del registro `WDTCSR`. El fusible tiene el valor de 1 cuando está sin programar y 0 cuando ha sido programado.

Los microcontroladores ATMega328P se comercializan con el fusible sin programar y cuando el fusible se programa, el bit `WDE` y `WDIE` quedan bloqueados con los valores de 1 y 0, respectivamente, por lo que el modo de reinicio queda programado de manera permanente, sin que se pueda anular por software.

Tabla 9.3: Configuración del *watchdog timer*

WDTON	WDE	WDIE	Modo
1	0	0	WDT detenido
1	0	1	Modo interrupción
1	1	0	Modo reinicio
1	1	1	Modo interrupción seguido de un reinicio
0	X	X	Modo reinicio

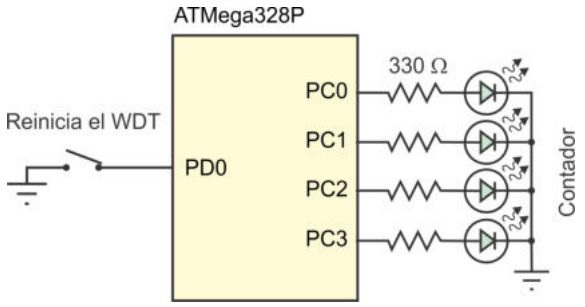


Figura 9.3: Contador para evaluar al *watch dog timer*

Ejemplo 9.2 - El WDT en modo de reinicio
Realice un contador de 0 a 15 que incremente automáticamente cada segundo y configure al WDT con un periodo de desbordamiento de 2.048 segundos, ubique la salida del contador en el Puerto C. Mediante un interruptor conectado en PD0 condicione el reset del WDT, para que, dependiendo de su estado, el MCU pueda o no reinicarse cuando desborda el WDT. Suponga que el fusible `WDTON` *está programado, por lo que siempre estará activo el modo de reinicio del WDT. En la Figura 9.3 se muestra la organización del hardware.*

Es necesario el acceso al registro `WDTCSR` para ajustar el tiempo de desbordamiento a 2.048 segundos. Se considera que cuando el interruptor está cerrado, se dará un reinicio al WDT en cada iteración del lazo infinito y el contador avanzará sin problema, en caso contrario, el MCU se va a reiniciar después de 2.048 segundos, por desbordamiento del WDT.

El código con la solución al problema es:

```
#define  F_CPU  1000000UL

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

int main(void) {
uint8_t  conta = 0;

  DDRC = 0xFF;                          // Salida del contador
  DDRD = 0x00;                          // Entrada para el interruptor
```

```
  PORTD = 0x01;                          // Pull-up en PD0

  WDTCSR |= (1<<WDCE) | (1<<WDE);        // Acceso a la configuración
  WDTCSR = 0x07;                         // Desborde a 2.048 segundos

  while (1) {
     PORTC = conta;                      // Salida del contador
     conta = (conta < 15)?conta+1:0;
     _delay_ms(1000);
     if(!(PIND & 0x01))                  // Interruptor cerrado
        asm("WDR");                      // Reset al WDT
  }
}
```

En caso de que el fusible WDTON no esté programado, en el momento en que se define el factor de preescala también se debe habilitar el modo de reinicio, la configuración requerida para el registro `WDTCSR` es:

```
WDTCSR |= (1<<WDCE) | (1<<WDE); // Acceso a la configuración
WDTCSR = 0x0F;                 // Desborde a 2.048 s y activa al WDT
```

Con el fusible sin programar, el WDT se puede activar o desactivar por software, sin embargo, para poner al bit `WDE` en bajo, primero se debe garantizar que la bandera `WDRF` también está en bajo, en caso contrario, el MCU va a quedar inestable. En la siguiente secuencia de código se muestra como desactivar al WDT:

```
  MCUSR &= ~(1 << WDRF);                 // Asegura bandera en bajo
  WDTCSR |= (1<<WDCE) | (1<<WDE);        // Acceso a la configuración
  WDTCSR = 0x00;                         // Desactiva al WDT
```

Ejemplo 9.3 - Modo interrupción del WDT
Repita el diseño del contador de 0 a 15 con incrementos automáticos pero configurando al WDT para que temporice los incrementos, con un periodo de desbordamiento de 1.024 s.

El modo interrupción requiere que el fusible `WDTON` se mantenga sin programar, se debe configurar al WDT en ese modo con el factor de preescala requerido y poner en alto al habilitador global de interrupciones. La bandera `WDIF` se limpia automáticamente cuando se da paso paso a la ISR. El hardware es similar al de la Figura 9.3, se omite el interruptor porque no se requiere. La solución del problema es:

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

uint8_t  conta = 0;                  // El contador es global

ISR(WDT_vect) {
  conta = (conta < 15)?conta+1:0;
```

```
  PORTC = conta;
}

int main(void) {
  DDRC = 0xFF;                          // Salida para el contador
  PORTC = 0;                            // Inicia en 0

  WDTCSR |= (1<<WDCE) | (1<<WDE);       // Acceso a la configuración
  WDTCSR = 0x46;                        // Modo interrupción con
                                        // desborde a 1.024 segundos
  sei();                                // Habilita las interrupciones
  while (1)
     asm("NOP");
}
```

El modo de interrupción seguido por un reinicio se consigue poniendo en alto a los bits `WDE` y `WDIE`, de la siguiente manera:

```
  WDTCSR |= (1<<WDCE) | (1<<WDE); // Acceso a la configuración
  WDTCSR = 0x4E;                  // Modo interrupción más reinicio
```

Si esta configuración se utiliza para el contador anterior, únicamente va a avanzar de 0 a 1 y luego se va a reiniciar.

El modo interrupción con reinicio se puede emplear en aplicaciones donde el MCU debe notificar a otro sistema que se va a reinicar porque el WDT ha desbordado, para ello, en la ISR del WDT se puede generar un pulso o se puede enviar un dato por una de las interfaces seriales.

9.3. Sección de Arranque en la Memoria Flash

La memoria Flash está particionada en dos secciones, una de aplicación y otra de arranque (ver Sección 2.3). 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 puede ubicarse una secuencia de código que corresponda con una aplicación ordinaria, esta sección está orientada para 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 se pueda actualizar por sí misma, realizando una autoprogramación. Esta organización es fundamental para la operación del software de Arduino, en la sección de arranque se ubica al *firmware* y este se encarga de actualizar la sección de aplicación.

La sección de aplicación inicia en la dirección 0 por lo que al energizar al MCU, o despueés de un *reset*, la ejecución comienza con la sección de aplicación. Sin embargo, con el fusible `BOOTRST` se consigue que la ejecución inicie en la sección de arranque

y no en la dirección 0. La ubicación de la sección de arranque es configurable, como se describe en el apartado siguiente.

9.3.1. Organización de la Memoria Flash

La memoria flash del ATMega328P es de 16k de palabras de 16 bits, se estructura en páginas de 64 palabras teniendo un total de 256 páginas por las dos secciones. En la Figura 9.4 se esquematiza esta organización. El acceso para el borrado y escritura implica la modificación de una página completa. La lectura es diferente, esta se puede realizar con un acceso por bytes.

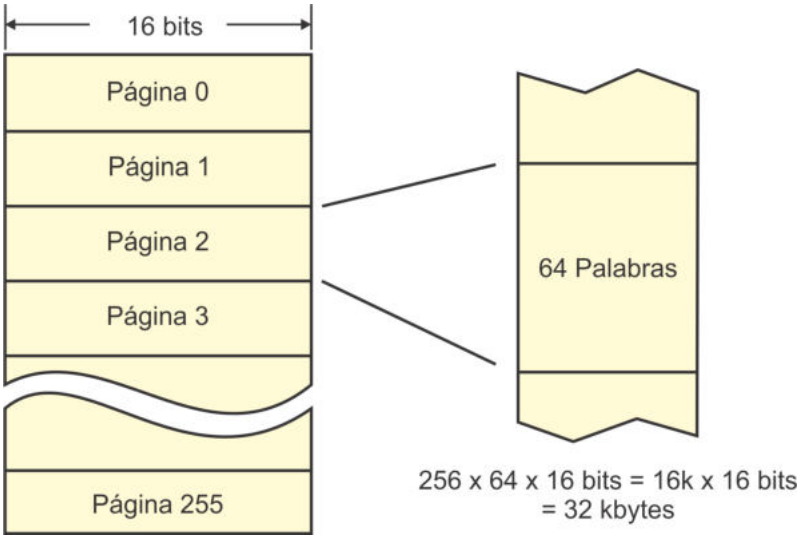


Figura 9.4: Organización de la memoria Flash en páginas

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 cuántas páginas son destinadas a la sección de arranque.

En la Tabla 9.4 se pueden ver las páginas destinadas para cada una de las secciones en un ATMega328P, en función del valor de 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 permitido para la sección de arranque.

Tabla 9.4: División de la memoria en secciones de aplicación y de arranque

BOOTSZ1	**BOOTSZ0**	**Aplicación**	**Arranque**	**Tamaño (Aplicación)**
1	1	0x0000-0x3EFF	0x3F00-0x3FFF	4 páginas (256 palabras)
1	0	0x0000-0x3DFF	0x3E00-0x3FFF	8 páginas (512 palabras)
0	1	0x0000-0x3BFF	0x3C00-0x3FFF	16 páginas (1024 palabras)
0	0	0x0000-0x37FF	0x3800-0x3FFF	32 páginas (2048 palabras)

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*, la 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 la Tabla 2.1 se mostraron las direcciones que les corresponden a los diferentes eventos. No obstante, los vectores pueden moverse a la sección de arranque, esto se realiza con los bits `IVSEL` e `IVCE` del registro de control del MCU (`MCUCR`, *MCU Control Register*). El registro `MCUCR` es un Registro I/O ubicado en la dirección 0x35 (0x55 de SRAM), cuyos bits son:

REG.	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
MCUCR	-	BODS	BODSE	PUD	-	-	IVSEL	IVCE

- **Bits 6 y 5 - BODS y BODSE:** Relacionados con la habilitación del detector de potencia baja (BOD) en un modo de reposo (ver Sección 2.9).
- **Bit 4 - PUD:** Para deshabilitar los resistores de *Pull-Up* (ver Sección 2.5).
- **Bit 1 -IVSEL:** (*Interrupt Vector Select*) Con un 1 en este bit, los vectores de interrupción se desplazan al inicio de la sección de arranque, cuya dirección depende de los fusibles `BOOTSZ` (ver Tabla 9.4). Un 0 en `IVSEL` mantiene los vectores de interrupción en la dirección 0.
- **Bit 0 -IVCE:** (*Interrupt Vector Change Enable*) Es un habilitador que evita cambios no deseados en la ubicación de los vectores de interrupción. Cualquier ajuste requiere la puesta en alto de `IVCE` y dentro de los 4 ciclos de reloj siguientes se debe escribir el valor deseado en `IVSEL`.

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.

9.3.2. Acceso a la Sección de Arranque

En el siguiente ejemplo se ilustra el acceso a las dos secciones de memoria, sin considerar un cargador para autoprogramación. El MCU inicia ejecutanto el código ubicado en la sección de arranque y posteriormente, cuando se presiona un botón, pasa a ejecutar el código ubicado en la sección de aplicación.

Ejemplo 9.4 - Acceso a la sección de arranque
Realice un programa con dos tareas, la primera será el parpadeo de un LED ubicado en la terminal PC0, el LED conmutará automáticamente cada 500 ms, aproximadamente. La segunda tarea será el parpadeo automático de otro LED ubicado en PC1 y la conmutación de un tercer LED ubicado en PC2, como una respuesta a un evento en la terminal PD3 (INT1). El código de la primera tarea se debe colocar en la sección de arranque y el de la segunda en la sección de aplicación. El programa debe abandonar la sección de arranque cuando se presione un botón conectado en PD2 (INT0), dejando los registros en su estado inicial para una adecuada ejecución del código ubicado en la sección de aplicación. En la Figura 9.5 se muestra el hardware requerido.

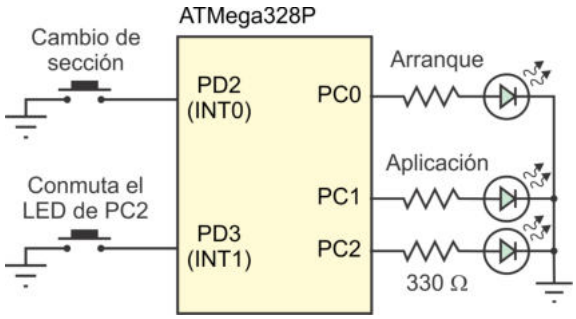


Figura 9.5: Contador para evaluar al *watch dog timer*

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 la dirección 0x3800.

Los botones se van a atender por interrupciones, por ello, inicialmente los vectores de interrupción deben moverse a la sección de arranque. Cuando ocurra un evento en la INT0 se deben regresar los vectores de interrupción a la sección de aplicación y los registros que se han modificado por software deben retomar su valor original, de esta manera, el cambio de sección básicamente se realiza con un salto a la dirección 0x0000 y, así, el programa ubicado en la sección de aplicación podrá ejecutarse sin inconvenientes.

La solución se desarrolla en lenguaje ensamblador, porque en lenguaje C no es posible establecer la dirección para ubicar el código generado. El programa es:

```
;-----------------------------------------------------------------
;---                    Sección de aplicación                  ---
;-----------------------------------------------------------------
    .org 0x0000
    JMP  inicio
```

```
        .org 0x0004          ; Vector de la INT1
        SBI  PINC, 2         ; Conmuta PC2
        RETI

inicio:
        CLR  R16             ; Puerto D como entrada
        OUT  DDRD, R16
        LDI  R16, 0xFF
        OUT  DDRC, R16       ; Puerto C como salida
        OUT  PORTD, R16      ; Pull-Up en el puerto D

        LDI  R16, 0x08       ; Configura la INT1 por flanco de bajada
        STS  EICRA, R16
        LDI  R16, 0x02       ; Habilita la INT1
        OUT  EIMSK, R16

        SEI                  ; Habilitador global de interrupciones

Lazo:                        ; Lazo infinito
        SBI  PINC, 1         ; Conmuta PC1 cada 500 ms
        CALL Espera_500ms
        RJMP Lazo

;-------------------------------------------------------------------
;---                  Sección de arranque                        ---
;-------------------------------------------------------------------
        .org 0x3800          ; Dirección de inicio
        JMP  ini_boot        ; Salta al código principal

        .org 0x3802          ; Vector de la INT0 desplazado
        LDI  R16, 0x01       ; Regresa los vectores de interrupciones
        OUT  MCUCR, R16      ; al inicio de la sección de aplicación
        LDI  R16, 0x00
        OUT  MCUCR, R16
        LDI  R16, 0xFF       ; Restablece el apuntador de la pila
        OUT  SPL, R16
        CLR  R16
        OUT  PORTC, R16      ; Recupera el estado inicial de los
        OUT  DDRC, R16       ; registros modificados, no se pueden
        OUT  PORTD, R16      ; reiniciar por hardware
        STS  EICRA, R16
        OUT  EIMSK, R16
        OUT  EIFR, R16
        OUT  SREG, R16       ; Limpia cualquier bandera modificada
        JMP  inicio          ; Salta a la sección de aplicación

ini_boot:
        LDI  R16, 0x01       ; Mueve los vectores de interrupciones
        OUT  MCUCR, R16      ; Para detectar la INT0 desde la
        LDI  R16, 0x02       ; sección de arranque
        OUT  MCUCR, R16
```

```
      CLR  R16              ; Puerto D como entrada
      OUT  DDRD, R16
      LDI  R16, 0xFF
      OUT  DDRC, R16        ; Puerto C como salida
      OUT  PORTD, R16       ; Pull-Up en el puerto D

      LDI  R16, 0x02        ; Configura la INT0 por flanco de bajada
      STS  EICRA, R16
      LDI  R16, 0x01        ; Habilita la INT0
      OUT  EIMSK, R16

      SEI                   ; Habilitador global de interrupciones

Lazo_boot:                  ; Lazo infinito
      SBI  PINC, 0          ; Parpadeo en PC0
      CALL Espera_500ms
      RJMP Lazo_boot
;
; Rutina para esperar 500 mS
;
Espera_500ms:
      PUSH R18              ; Respaldo de registros en la pila
      PUSH R17
      PUSH R16
      LDI  R18, 2
et3:  LDI  R17, 250
et2:  LDI  R16, 250
et1:  DEC  R16              ; Itera 250 veces, 4 us por iteración
      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
      POP  R16              ; Recuperación de registros de la pila
      POP  R17
      POP  R18
      RET
```

El código en la sección de aplicación debe estar completo, es decir, deben configurarse todos los recursos empleados para no depender de la sección de arranque. De esta forma, el código de la sección de aplicación podrá funcionar adecuadamente aún si el fusible `BOOTRST` no fue programado.

El paso de la sección de arranque a la sección de aplicación se realiza con un simple salto, pero antes de hacerlo, se modifican los registros para regresar al MCU a su estado inicial, se consideran 3 aspectos:

- Los vectores de interrupción se regresan a la sección de aplicación, en el ejemplo se valida su funcionalidad al emplear la interrupción externa 1.

- El apuntador de pila se regresa a su posición inicial, solo se restaura el valor del registro `SPL` porque se modificó con la llamada a la rutina `Espera_500ms` y con el paso a la ISR de la INT0. El registro `SPH` se mantuvo sin cambios.

- Los Registros I/O adicionales que se modificaron también recuperan su valor inicial, incluyendo los registros `EIFR` y `SREG`, para evitar una interrupción no deseada en la sección de aplicación.

La ISR de la INT0 en la sección de arranque no tiene una instrucción `RETI`, no es necesaria porque después de recuperar el estado inicial del MCU, es suficiente con un salto absoluto a la dirección cero, la instrucción `JMP` no modifica las banderas. Puede decirse que es un reinicio por software y no por hardware.

En el Ejemplo 9.4 también se observa que desde la sección de aplicación se tiene acceso a una rutina ubicada en la sección de arranque, aunque podría ocurrir lo contrario. Esta característica hace que se pueda omitir la escritura de rutinas en la sección de aplicación, si se sabe de su presencia en la sección de arranque.

9.3.3. Cargador para Autoprogramación

Aunque el Ejemplo 9.4 funciona correctamente, no tiene mucho sentido separar una aplicación en las dos secciones de la memoria Flash. La sección de arranque 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 mientras evalúa si desde una computadora, u otro sistema, va a llegar una actualización de la aplicación, a través de una de las interfaces incluidas en el MCU (USART, SPI o TWI).

Si durante ese periodo se establece la comunicación, por esa interfaz se va a recibir la nueva versión , 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.

En la Figura 9.6 se ilustra esta idea, la cual en apariencia es simple, no obstante, deben tomarse en cuenta algunos aspectos relevantes para la escritura de la memoria Flash, aspectos que son descritos en los siguientes apartados.

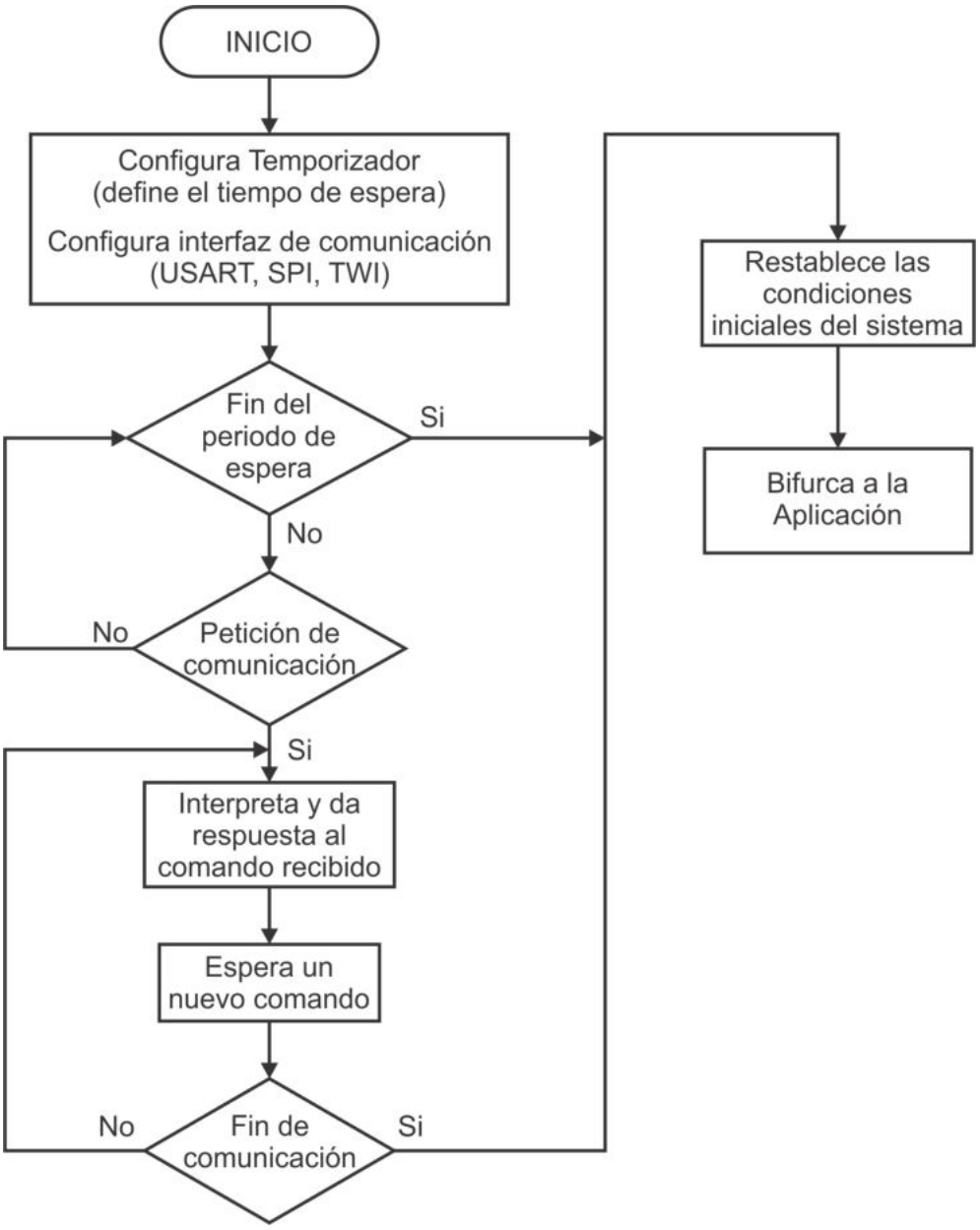


Figura 9.6: Flujo natural de un cargador para autoprogramación

El proceso mostrado en la Figura 9.6 siguen los sistemas Arduino, sus tarjetas incluyen dos microcontroladores, el MCU principal tiene al cargador en su sección de arranque y el MCU secundario es el adaptador de USB a TTL, pero además, maneja la señal de *reset* del principal, así, cuando desde una computadora se va a enviar un nuevo programa, el MCU secundario aplica una señal de *reset* al principal para que ejecute al cargador y escriba el nuevo programa en la sección de aplicación. Es por ello que, cuando se presiona el botón de *reset* en una tarjeta Arduino, hay

un periodo de tiempo antes de que inicie la ejecución, debido a que el cargador está en espera de una nueva aplicación.

Si el cargador requiere el uso de interrupciones, debe incluir instrucciones para ubicar a los vectores de interrupciones en la sección de arranque y, 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.

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.
- Solo la sección de arranque queda protegida.
- Solo la sección de aplicación queda protegida.
- Ambas secciones quedan sin protección, permitiendo un acceso total.

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 es para la sección de arranque. En la Tabla 9.5 se describen los diferentes niveles de protección, los cuales dependen de las restricciones impuestas a las instrucciones `LPM` y `SPM`, enfocadas a transferencias con la memoria de código.

Los fusibles `BLB` tienen un 1 cuando están sin programar, por lo que un MCU ATMega328P se pone a la venta sin restricciones para las instrucciones `SMP` y `LMP`. La programación de los fusibles se puede realizar en el momento de configurar al dispositivo.

Capacidades para Leer Mientras Escribe

Además de la división entre la sección de aplicación y la sección de arranque, la memoria Flash también se divide en dos secciones de tamaño fijo, la primera es una sección con capacidad de 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*). En la Tabla 9.6 se muestra el tamaño de cada sección en un ATMega328P, debe notarse que la sección NRWW abarca el espacio que originalmente tiene la sección de arranque.

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.

Tabla 9.5: Bits de bloqueo para las diferentes secciones de memoria

Modo	**BLB02**	**BLB01**	**Protección (sección de aplicación)**
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	1	La instrucción `SPM` no puede escribir en la sección de aplicación y `LPM`, ejecutada desde la sección de arranque, no puede leer en la sección de aplicación. Si los vectores de interrupción son colocados en la sección de arranque, las interrupciones se deshabilitan mientras se ejecute desde la sección de aplicación.
4	0	0	La instrucción `LPM`, ejecutada desde la sección de arranque, no puede leer en la sección de aplicación. Si los vectores de interrupción son colocados en la sección de arranque, las interrupciones se deshabilitan mientras se ejecute desde la sección de aplicación.
Modo	**BLB12**	**BLB11**	**Protección (sección de arranque)**
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	1	La instrucción `SPM` no puede escribir en la sección de arranque y `LPM`, ejecutada desde la sección de aplicación, no puede leer en la sección de arranque. Si los vectores de interrupción son colocados en la sección de aplicación, las interrupciones se deshabilitan mientras se ejecute desde la sección de arranque.
4	0	0	La instrucción `LPM`, ejecutada desde la sección de aplicación, no puede leer en la sección de arranque. Si los vectores de interrupción son colocados en la sección de aplicación, las interrupciones se deshabilitan mientras se ejecute desde la sección de arranque.

Tabla 9.6: Tamaño de las secciones RWW y NRWW

Sección	**Páginas**	**Dirección**
Leer mientras escribe (RWW)	224	0x0000 - 0x37FF
No leer mientras escribe (NRWW)	32	0x3800 - 0x3FFF

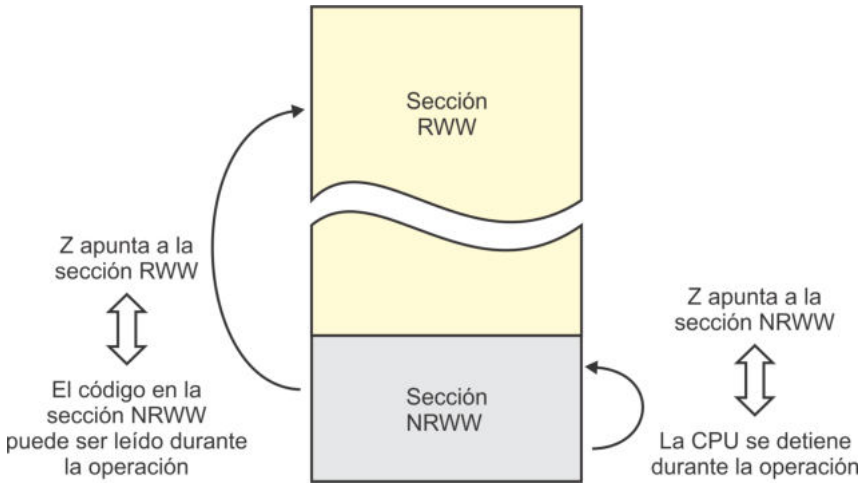


Figura 9.7: Lee de la sección NRWW mientras escribe en la sección RWW

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 9.7 se ilustra este comportamiento, en donde puede verse que el apuntador Z es empleado para referenciar la ubicación de la página a modificar.

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 Flash, 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.

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 9.4) y no se puede modificar por localidades individuales, la escritura o borrado se realiza en páginas completas.

Para lograr ese acceso, 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 *buffer* es independiente de la memoria SRAM, es sólo de escritura y debe ser llenado por palabras, no es posible la escritura de un byte individual.

El registro Z se utiliza para direccionar una palabra en el *buffer*, puesto que una página tiene 64 palabras, se utilizan 6 bits del registro para indicar el número de palabra, en la Figura 9.8 se ilustra el direccionamiento, el bit menos significativo de Z es ignorado.

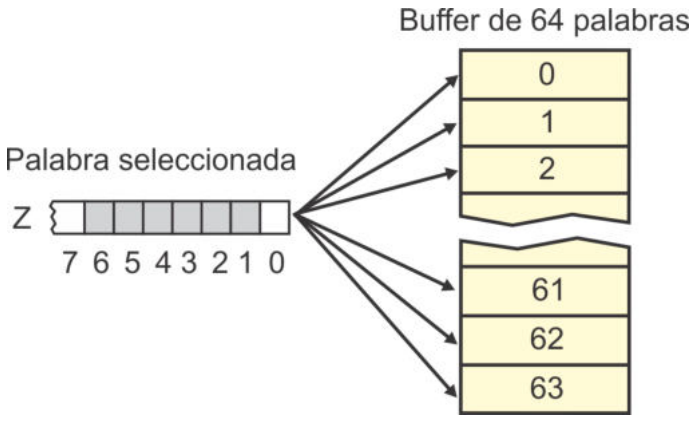


Figura 9.8: Direccionamiento en el buffer temporal

La escritura en el *buffer* se auxilia del registro `SPMCSR` (*Store Program Memory Control and Status Register*), el bit menos significativo de este registro es el habilitador de escritura (`SELFPRGEN`, *Self Programming Enable*). El procedimiento para escribir una palabra en el *buffer* es:

1. Escribir la palabra en R1:R0.
2. Colocar la dirección donde se escribirá la palabra en el registro Z.
3. Poner en alto al bit `SELFPRGEN` del registro `SPMCSR`.
4. Dentro de los 4 ciclos de reloj siguientes, ejecutar la instrucción `SPM`.

Es importante precisar que si el bit `SELFPRGEN` se pone en alto junto con algún otro bit del registro `SPMCSR`, se tendrán 4 ciclos de reloj para realizar una tarea diferente a la escritura de una palabra en el *buffer*.

Una vez que el *buffer* se ha llenado, su contenido puede ser copiado en la memoria Flash. Nuevamente se requiere del apoyo del registro `SPMCSR`, este es un Registro

I/O con dirección 0x37 (0x57 si es tratado como SRAM). Los bits de este registro son:

7	6	5	4	3	2	1	0
SPMIE	RWWSB	-	RWWSRE	BLBSET	PGWRT	PGERS	SELFPRGEN

- **Bit 7 - SPMIE:** (*SPM Interrupt Enable*) Si este bit y el bit `I` de `SREG` están en alto, se produce una interrupción tan pronto como el bit `SELFPRGEN` es puesto en bajo. El bit `SELFPRGEN` se pone en bajo cuando concluye una operación sobre la memoria Flash, puede verse en la Tabla 2.1 que el vector de interrupción tiene la etiqueta SPM_READY y se ubica en la dirección 0x032.

- **Bit 6 - RWWSB:** (*Read-While-Write Section Busy*) Es una bandera que se pone en alto cuando se está realizando una operación de autoprogramación en la sección RWW, puede ser el borrado o escritura de una página y, por lo tanto, no se puede realizar otro acceso a la sección RWW. La bandera se limpia cuando se escribe un 1 en el bit `RWWSRE` después de que concluye la operación de autoprogramación.

- **Bit 4 - RWWSRE:** (*Read-While-Write Section Read Enable*) Es un habilitador de 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 `SELFPRGEN` van a tener un nivel alto. Al concluir con la operación, el bit `SELFPRGEN` se pone en bajo pero la sección RWW aún permanece bloqueada. Para rehabilitar su acceso, los bits `RWWSRE` y `SELFPRGEN` 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. Es necesario esperar que el bit `SELFPRGEN` sea puesto en bajo antes de escribir en el bit `RWWSRE`.

- **Bit 3 - BLBSET:** (*Boot Lock Bit Set*) Con este bit se habilita el acceso a los diferentes fusibles o Bits de Configuración y Seguridad, el acceso puede ser para una escritura o lectura.

  Para una escritura, los bits `BLBSET` y `SELFPRGEN` 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:

7	6	5	4	3	2	1	0
1	1	BLB12	BLB11	BLB02	BLB01	LB2	LB1

  Se observa que se pueden esribir los bits BLB, empleados para restringir el acceso a las diferentes secciones en la memoria Flash (ver Tabla 9.5), y los bits LB, que definen una protección para la EEPROM y la Flash (se describen en la siguiente sección).

  En una escritura no importa el valor del registro R1 y tampoco del apuntador

Z, aunque el fabricante sugiere que se utilice 0x0001 para una compatibilidad futura. 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, después de que los bits `BLBSET` y `SELFPRGEN` se ponen en alto, se cuenta con 3 ciclos de reloj para ejecutar la instrucción `LPM`. En el apuntador Z se debe especificar el grupo de fusibles a leer y el resultado de la lectura queda disponible en el registro señalado como destino en la instrucción. En la siguiente sección se describen los grupos de fusibles y el valor que debe tener el apuntador Z para la lectura de cada grupo.

- **Bit 2 - PGWRT:** (*Page Write*) 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 `SELFPRGEN` 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. En el siguiente apartado se indica cómo hacer el direccionamiento con el apuntador Z.

- **Bit 1 - PGERS:** (*Page Erase*) Este bit hace posible el borrado de una página en la memoria Flash, la página debe estar direccionada con la parte alta del apuntador Z. Los bits `PGERS` y `SELFPRGEN` 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 - SELFPRGEN:** (*Self Programming Enable*) Este bit habilita el uso de la instrucción `SMP` durante los siguientes 4 ciclos de reloj con diferentes consecuencias. Si únicamente se puso en alto al bit `SELFPRGEN`, la instrucción `SPM` escribe la palabra contenida en R1:R0 en el *buffer* de almacenamiento temporal, en la dirección apuntada por Z. Si `SELFPRGEN` y `RWWSRE` se pusieron en alto, la instrucción `SPM` desbloquea el acceso a la sección RWW y limpia la bandera `RWWSB`. Si `SELFPRGEN` y `BLBSET` se pusieron en alto, la instrucción SPM escribe en los fusibles de BLB y LB, el dato contenido en R0. Si `SELFPRGEN` 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 `SELFPRGEN` y `PGERS` se pusieron en alto, la instrucción `SPM` borra la página direccionada con la parte alta del apuntador Z.

Además, si `SELFPRGEN` y `BLBSET` se ponen en alto, se cuenta con 3 ciclos de reloj para ejecutar la instrucción `LPM` y leer un grupo de fusibles especificado en el apuntador Z. El valor leído queda en el registro indicado en la instrucción.

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. PCPAGE requiere de 8 bits porque un ATMega328P tiene 256 páginas y PCWORD es de 6 bits dado que una página tiene 64 palabras.

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 con un desplazamiento de una posición para no incluir a su bit menos significativo. En la Figura 9.9 se muestra la distribución de los bits para direccionar la memoria Flash desde el PC y su correspondencia con el apuntador Z.

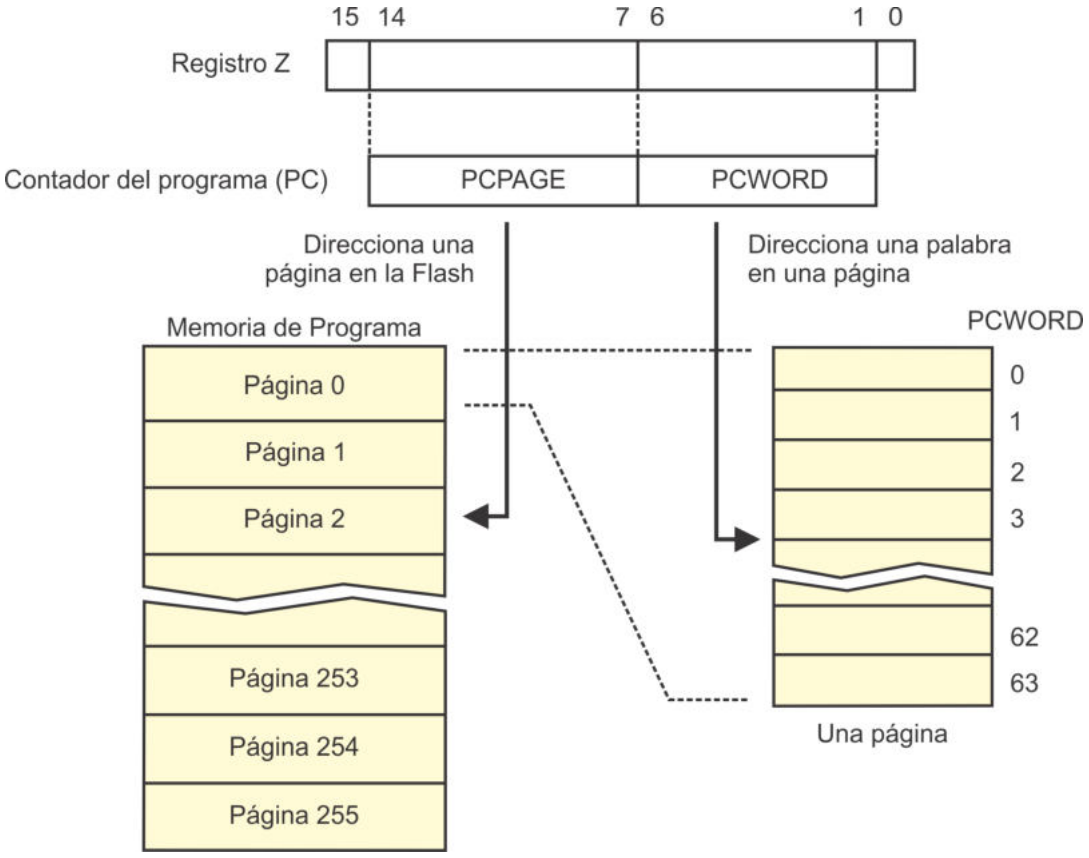


Figura 9.9: Distribución de bits para direccionar la memoria de código

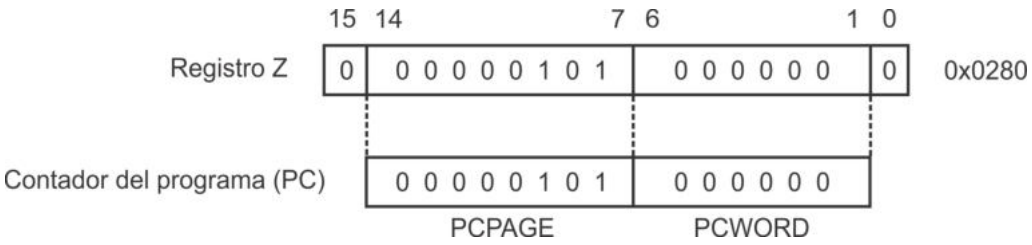


Figura 9.10: Ejemplo para definir el valor de Z

Para comprender esta distribución de bits, en la Figura 9.10 se muestra que el apuntador Z debe tener el valor de 0x0280 para hacer referencia a la página 5 de la memoria Flash (5 desplazado a la izquierda por 7 posiciones).

Una vez que se ha colocado el valor correcto en el apuntador Z y se han ajustado los bits requeridos en el registro `SPMCSR`, se puede ejecutar la instrucción `SMP`, para iniciar con la escritura o borrado de una página en la memoria Flash. Dado que estas operaciones requierien de varios ciclos de reloj, hay 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 instrucción `LPM` también hace uso del apuntador Z, sin embargo, el acceso para una carga desde la memoria de programa se realiza por bytes y no por páginas, por ello, para esta instrucción es necesario ubicar los 15 bits menos significativos del registro Z.

Consideraciones para la Escritura de una Página

Es necesario borrar una página de la memoria Flash antes de escribir nuevos datos, el nuevo contenido inicialmente debe estar en el *buffer* temporal. El procedimiento a seguir es: se borra la página de la Flash, se llena el buffer temporal y se escribe el contenido del *buffer* en la memoria Flash, aunque se puede invertir el orden de los primeros dos pasos.

Se debe escribir una página completa, no es posible modificar solo una fracción de la misma. Si una aplicación requiere la modificación de una pequeña parte de una página, primero se debería llenar el *buffer* termporal con el contenido de la página, con la información en el *buffer*, se puede modificar la fracción deseada, se borra la página de la Flash y, finalmente, se escribe la página completa.

9.4. Bits de Configuración y Seguridad

Los microcontroladores AVR tienen un conjunto de fusibles que se pueden programar para proteger el contenido de un dispositivo o definir su comportamiento. Los fusibles se organizan en 4 bytes, siendo 1 byte de seguridad y 3 de configuracón. Todos los

fusibles tienen un 1 lógico cuando están sin programar y su programación se realiza de manera similar a la descarga de aplicaciones en la memoria Flash.

El byte de seguridad solo tiene 6 bits válidos (*Lock Bits*) y su contenido se muestra en la Tabla 9.7. Los bits de las posiciones 7 y 6 no están implementados. Los bits 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 9.7: Fusibles relacionados con la seguridad del MCU

No. Bit	Nombre	Descripción	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 la Tabla 9.5 se describieron los diferentes niveles de protección para las dos secciones de la memoria Flash, en donde se puede ver que la seguridad se determina por las restricciones impuestas a las instrucciones `LPM` y `SPM`.

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 9.8. Se puede ver que un MCU ya no se puede reprogramar después de elegir los modos 2 o 3, esto incluye los fusibles, por lo que tampoco se podría cambiar la seguridad.

Tabla 9.8: 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.

Los 3 bytes de configuración están organizados como: un byte bajo (*Low Fuse Bits*), un byte alto (*High Fuse Bits*) y un byte extendido (*Extend Fuse Bits*). Los fusibles del byte bajo están relacionados con el sistema de reloj del MCU, el cual se describió en la Sección 2.8, en la Tabla 9.9 se muestran sus bits con los valores por *default*, recordando que un 1 significa que no ha sido programado.

Tabla 9.9: Byte bajo de los fusibles de configuración

No. Bit	Nombre	Descripción	Default
7	CKDIV8	Divide el reloj por 8	0
6	CKOUT	Reloj de salida	1
5	SUT1	Selecciona tiempo de establecimiento	1
4	SUT0	Selecciona tiempo de establecimiento	0
3	CLKSEL3	Selecciona la fuente de reloj	0
2	CLKSEL2	Selecciona la fuente de reloj	0
1	CLKSEL1	Selecciona la fuente de reloj	1
0	CLKSEL0	Selecciona la fuente de reloj	0

Los ATMega328P son puestos a la venta con `CKSEL[3:0]` = "0010", seleccionando un oscilador interno de 8 MHz, pero como `CKDIV8` inicia con 0 (programado), se habilita un divisor entre 8, haciendo que el MCU opere a 1 MHz. Además, el fusible `CKOUT` inicia en 1 (sin programar), de manera que no se emite el reloj del sistema en la terminal PB0. Finalmente, los fusibles `SUT[1:0]` inician con "10", ajustando el tiempo de establecimiento después de un reinicio a 65 ms.

Con los fusibles del byte alto se pueden configurar diferentes parámetros, en la Tabla 9.10 se muestran sus bits con los valores por *default*, cabe señalar que si se cambia el valor a los fusibles `RSTDISBL`, `DWEN` o `SPIEN`, ya no se podrá programar al MCU por medio de un programador serial.

Tabla 9.10: Byte alto de los fusibles de configuración

No. Bit	Nombre	Descripción	Default
7	RSTDISBL	Deshabilita el *reset* externo (PC6)	1
6	DWEN	Habilita la interfaz debugWire	1
5	SPIEN	Habilita la interfaz SPI	0
4	WDTON	El WDT estará siempre activo	1
3	EESAVE	Protege la EEPROM en el borrado del MCU	1
2	BOOTSZ1	Define el tamaño de las secciones de la Flash	0
1	BOOTSZ0	Define el tamaño de las secciones de la Flash	0
0	BOOTRST	Ubica los vectores de interrupciones	1

Los bits 0, 1 y 2 están relacionados con la sección de arranque de la memoria Flash y su operación fue descrita en la Sección 9.3. El bit `EESAVE` protege a la EEPROM,

haciendo que su contenido se preserve durante el borrado de la memoria Flash, inicialmente no está programado. El bit `WDTON` activa de manera permanente el modo de reinicio del WDT (Sección 9.2), inicialmente no está activo.

El bit `SPIEN` inicia programado, si se desprograma no se podrán transferir programas y datos por medio de la interfaz SPI. El bit `DWEN` habilita la interfaz *debugWire*, una interfaz de depuración que se describe en la sección posterior. Finalmente, el bit `RSTDISBL` deshabilita al *reset* externo, si se programa, la terminal PC6 funcionará como una terminal de propósito general.

El byte extendido de fusibles se muestra en la Tabla 9.11, solo 3 de los 8 bits son utilizados y se trata de los fusibles `BODLEVEL`, con los cuales se habilita el *reset* del MCU por bajo voltaje (*Brown out*) y se selecciona el valor para el voltaje de umbral. En la Sección 2.7 se describió el comportmiento de este tipo de reinicio y la configuración de los fusibles relacionados.

En la Tabla 9.12 se muestra un resumen de los 4 bytes de fusibles con sus valores iniciales, es decir, valores con los que se ponen a la venta los ATMega328P. Un 1 significa que el fusible está sin programar. No se recomienda modificarlos sin anticipar el resultado esperado, por ejemplo, si el MCU está trabajando con el oscilador interno y se decide cambiar los fusibles para emplear un oscilador externo, el MCU dejará de operar si no está conectado el cristal o resonador externo.

Tabla 9.11: Byte extendido de los fusibles de configuración

No. Bit	**Nombre**	**Descripción**	**Default**
7	-		1
6	-		1
5	-		1
4	-		1
3	-		1
2	BODLEVEL2	Configura el *reset* por bajo voltaje	1
1	BODLEVEL1	Configura el *reset* por bajo voltaje	1
0	BODLEVEL0	Configura el *reset* por bajo voltaje	1

Tabla 9.12: Bits de Configuración y Seguridad

Lock Byte		**Low Byte**		**High Byte**		**Extended Byte**	
-	1	CKDIV8	0	RSTDISBL	1	-	1
-	1	CKOUT	1	DWEN	1	-	1
BLB12	1	SUT1	1	SPIEN	0	-	1
BLB11	1	SUT0	0	WDTON	1	-	1
BLB02	1	CKSEL3	0	EESAVE	1	-	1
BLB01	1	CKSEL2	0	BOOTSZ1	0	BODLEVEL2	1
LB2	1	CKSEL1	1	BOOTSZ0	0	BODLEVEL1	1
LB1	1	CKSEL0	0	BOOTRST	1	BODLEVEL0	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, en esos casos se debe revisar la configuración completa porque si solo se modifican los fusibles de interés, sin conservar el valor de los restantes, podría ocasionar que el microcontrolador no opere o que lo haga en forma incorrecta.

9.4.1. Acceso a los Fusibles desde una Aplicación

En la sección anterior se describió el acceso a la memoria Flash para operaciones de borrado y escritura. El acceso se realiza con el apoyo del registro `SPMCSR` (*Store Program Memory Control and Status Register*). En la posición 3 de este registro se encuentra el bit `BLBSET` y en la posición 0 está el bit `SELFPRGEN`, combinando ambos bits se puede realizar la escritura del byte de seguridad y la lectura de los 4 grupos de fusibles: el byte de seguridad y los 3 bytes de configuración.

Para la escritura del byte de seguridad, el valor a escribir se debe ubicar en el registro R0, de acuerdo con la Tabla 9.7. Posteriormente, los bits `BLBSET` y `SELFPRGEN` se deben poner en alto al mismo tiempo y, a partir de ese momento, se tienen 4 ciclos de reloj para ejecutar la instrucción `SPM`. No importa el valor del registro R1 y tampoco del apuntador Z, aunque el fabricante sugiere que se utilice 0x0001 para una compatibilidad futura. 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.

La operación de lectura se puede aplicar en los 4 grupos de fusibles, en el apuntador Z se debe especificar el grupo a leer, con 0x0001 se hace referencia al byte de seguridad, con 0x0000 la lectura es del byte bajo de configuración, el valor de 0x0003 es para el byte alto y por último, con 0x0002 se hará referencia al byte extendido.

Elegido el byte a leer, se deben poner en alto los bits `BLBSET` y `SELFPRGEN`, despues de ello, se cuenta con 3 ciclos de reloj para ejecutar la instrucción `LPM`. El resultado de la lectura queda disponible en el registro señalado como destino en la instrucción. Los fusibles que están programados se leerán como cero y los que están sin programar van a tener uno.

9.5. La interfaz *debugWire*

Los microcontroladores ATMega328P se pueden depurar empleando la interfaz *debugWIRE*, estos MCUs cuentan con un sistema interno de depuración (*On-Chip Debug System*) que utiliza un solo alambre, en modo bidireccional, para controlar la ejecución de instrucciones de un programa. La terminal destinada para la depuración es la terminal de *Reset*, la cual toma esta funcionalidad cuando se habilita el módulo de depuración. Es evidente que el MCU debe energizarse para que se pueda depurar, en la Figura 9.11 se muestran las terminales que utiliza la interfaz *debugWire*.

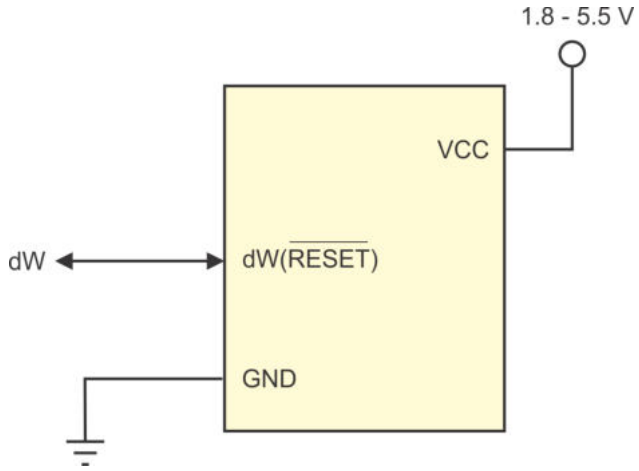


Figura 9.11: Interfaz *debugWire*

El modo de depuración se activa cuando se programa el fusible `DWEN`, que es parte de los Bits de Configuración y Seguridad. El fusible puede modificarse con cualquier programador, del tipo paralelo o serial (a través de la interfaz SPI); sin embargo, se debe considerar que si el MCU ingresa al modo de depuración, el fusible `DWEN` no se podrá modificar con cualquier programador que utilice la interfaz SPI porque la función de *reset* no está disponible. Por ello, es necesario emplear una herramienta especializada para la programacón y depuración del ATMega328P. La programación permitirá la descarga del código binario y la modificación del fusible `DWEN`, mientras que con la depuración se podrá realizar la ejecución paso a paso, mostrando los resultados en el hardware real, así como en el entrono de Microchip Studio. La depuración requiere que los fusibles del byte de seguridad no estén programados, para permitir el acceso a las memorias no volátiles.

Existen herramientas para la programación y depuración, como ATATMEL-ICE o AVR Dragon. La última es una versión de bajo costo que dejó de manufacturarse, su demanda bajó con el incremento de la popularidad de Arduino, las tarjetas Arduino tienen un costo más bajo y la puesta en marcha de una nueva aplicación se ha simplificado considerablemente.

Algunas características y consideraciones del sistema de depuración manejado por la interfaz *debugWire*, son:

- Es posible depurar todas las funciones del ATMega328P, tanto analógicas como digitales, con excepción de la terminal de *Reset*.
- La depuración se realiza desde el entorno de Microchip Studio, el programa del MCU puede escribirse en Ensamblador o en Lenguaje C.
- El recurso de depuración no es intrusivo, es decir, no afecta la ejecución en tiempo real de las instrucciones.

- El número de puntos de ruptura (*Break Points*) es ilimitado.
- La depuración no afecta al sistema de reloj, se mantiene la fuente de reloj seleccionada con los fusibles `CKSEL`.
- Es preferible que la terminal de *Reset* se mantenga libre, sin hardware adicional, solo se puede colocar un resistor de pull-up con un valor de 10 kΩ o superior.

9.5.1. Puntos de Ruptura

El repertorio de intrucciones de los microcontroladores AVR incluye la instrucción `BREAK`, con esta instrucción se detiene la ejecución de un programa en modo de depuración, en espera de que el programador solicite el avance en el entorno de Microchip Studio, esto se puede realizar aún cuando se ha seleccionado al simulador, como la herramienta de depuración del entorno.

Además, si durante la depuración se agrega un punto de ruptura (*break point*), significa que se debe incluir una instrucción `BREAK` en el programa, esto implica que la memoria Flash se debe reprogramar para una adecuada depuración, estas acciones se realizan de manera automática en Microchip Studio.

9.5.2. Registro de Depuración

El sistema de depuración interno incluye el registro `DWDR` (*debugWire Data Register*), este registro proporciona un canal de comunicación entre el MCU y el depurador. El acceso al registro solo se puede realizar por medio de la interfaz *debugWire*, por lo que las instrucciones normales no lo pueden utilizar como un Registro I/O.

9.5.3. Limitaciones de la Interfaz *debugWire*

La comunicación con el sistema interno de depuración se realiza por medio de la terminal de *reset*, la cual se conoce como dW cuando el fusible `DWEN` se ha programado, esto significa que el reset externo no se puede emplear mientras se encuentre activo el modo de depuración.

Con la programación del fusible DWEN se habilitan algunas partes del sistema de reloj que se mantienen activas en todos los modos de bajo consumo, como una consecuencia, el consumo de potencia en bajo consumo se incrementa.

Por lo tanto, el fusible `DWEN` debe mantenerse desprogramado siempre que la interfaz *debugWire* no sea utilizada.

9.6. Ejercicios

1. Para evaluar la funcionalidad del WDT:

   a) Desarrolle un contador binario de 16 bits que se incremente cada milisegundo, mostrando la salida en 16 LEDs conectados en los puertos B y D de un ATMega328P. Habilite al WDT, con el fusible `WDTON` o con el registro `WDTCSR`, asegurando que los bits `WDP[3:0]` se quedan con 0. En el código no incluya instrucciones para el reinicio del WDT y estime el valor máximo que alcanza el contador, antes de reiniciar la cuenta, observando los LEDs encendidos.

   b) Modifique los bits `WDP[3:0]` del registro `WDTCSR` y observe el valor máximo que alcanza el contador antes de su reinicio, pruebe con diferentes combinaciones de los bits `WDP[3:0]`.

   c) Coloque la instrucción `WDR` dentro del lazo que incrementa al contador y observe que la cuenta ya no se reinicia.

2. Empleando el modo temporizador del WDT, realice un sistema que envíe un carácter ASCII imprimible cada 8.192 s, periodo máximo que alcanza el WDT en modo temporizador. Los caracteres ASCII imprimibles están en el rango de 0x20 a 0x7E. El envío se hará por medio de la USART, configurando para que la comunicación sea asíncrona a 9600 bps, con datos de 8 bits, sin paridad y con 1 bit de paro.

3. Considerando que la memoria Flash tiene dos secciones, desarrolle un sistema con una doble funcionalidad, el sistema debe contar con 4 displays de 7 segmentos conectados por medio de un bus común de datos y dos botones. En la sección de arranque coloque el código para una marquesina de mensajes y en la sección de aplicación, el código para un contador ascendente/descendente. Agregue el hardware y software necesario para una comunicación serial y acondicione para que el sistema pase de la ejecución de la sección de arranque a la sección de aplicación cuando reciba un comando serial.

4. Diseñe un sistema que realice animaciones gráficas en una matriz de 8x8 LEDs, la animación a mostrar se basará en un conjunto de pequeñas imágenes situadas en la memoria Flash. Complemente el sistema con un cargador para autoprogramación, con base en el diagrama de flujo de la Figura 9.6, para que se puedan cambiar las imágenes que conforman la animación. Utilice la USART como medio de comunicación para actualizar la información, probando con diferentes animaciones.

5. 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 velocidad del parpadeo al modificar el valor del fusible `CKDIV8`.

6. Utilizando una tarjeta Arduino, realice un programa que lea los 4 grupos de fusibles y los envíe por el puerto serie. Reciba los 4 bytes en la terminal serial de Arduino y evalúe su contenido, explique las diferencias que encuentre con respecto a la Tabla 9.12.

Capítulo 10

Manejo de Dispositivos Externos

En este capítulo se muestra el funcionamiento de dispositivos externos típicos y cómo manejarlos por medio de un microcontrolador, específicamente con el ATMega328P, con algunas consideraciones para su programación.

10.1. 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 y 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 kΩ, con este valor, en el resistor de *pull-up* circula una corriente de 0.5 mA cuando el dispositivo se cierra, si el MCU está alimentado con 5 V.

En la Figura 10.1 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.

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 10.2. Con un resistor de *pull-down*, el dispositivo introduce un 0 lógico mientras se encuentra abierto, al cerrarlo

introduce un 1 lógico. En un AVR, el resistor de *pull-up* interno no debe conectarse si se utiliza un resistor de *pull-down* externo.

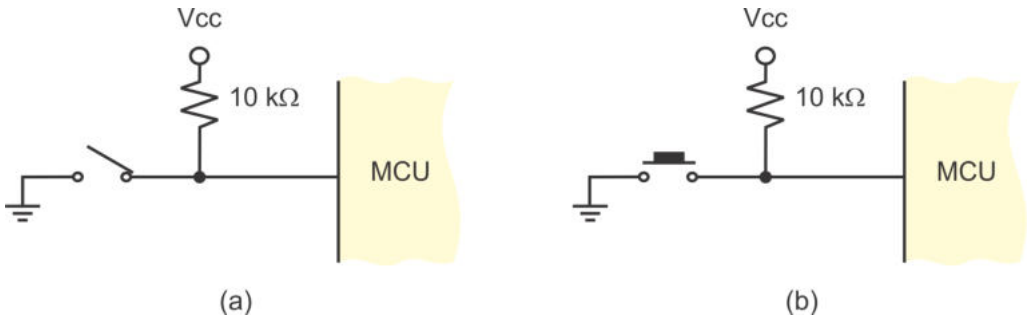


Figura 10.1: Conexión de (a) un interruptor y (b) un botón, con resistor de *pull-up*

Figura 10.2: (a) Un interruptor y (b) un botón, con resistor de *pull-down*

10.2. Teclado Matricial

Cuando un sistema maneja pocas entradas, los interruptores o botones se pueden conectar directamente al MCU, utilizando una terminal por botón o interruptor. Sin embargo, si el sistema requiere de muchas entradas lo mejor es emplear un teclado matricial.

Un teclado matricial consiste en 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 10.3, con ello, el manejo de 16 botones solo 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 tiene sentido únicamente si n y m son mayores a 2.

En la Figura 10.3 se observa que mientras no se ha presionado alguna tecla, las entradas tienen un nivel lógico alto (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 corresponda con la tecla presionada. La revisión completa implica colocar, secuencialmente, cada uno de los términos mostrados en la Tabla 10.1.

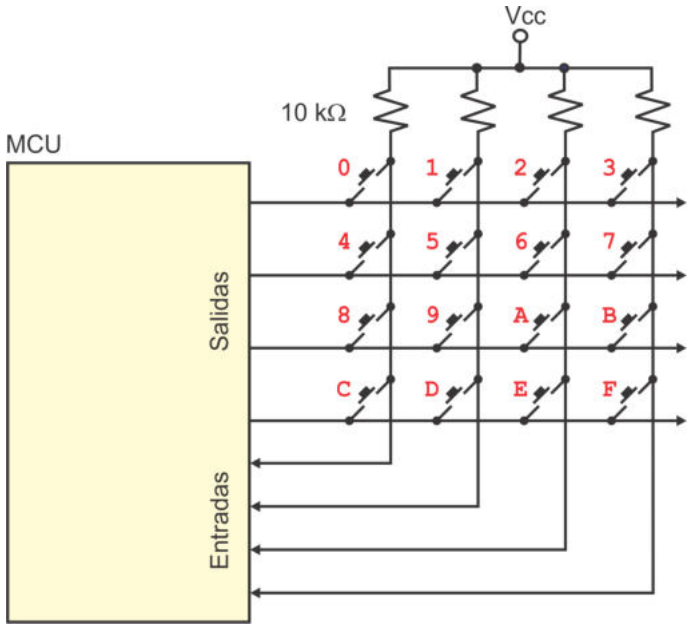


Figura 10.3: Teclado matricial de 4 x 4 botones

Tabla 10.1: Secuencia de salidas para revisar un teclado matricial de 4 x 4

Salida	**Acción**
0b1110	Sondea el rengón 0
0b1101	Sondea el rengón 1
0b1011	Sondea el rengón 2
0b0111	Sondea el rengón 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:

$$tecla = 4 \times renglón + columna$$

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:

$$tecla = 4 \times renglón + columna = 4 \times 1 + 2 = 6$$

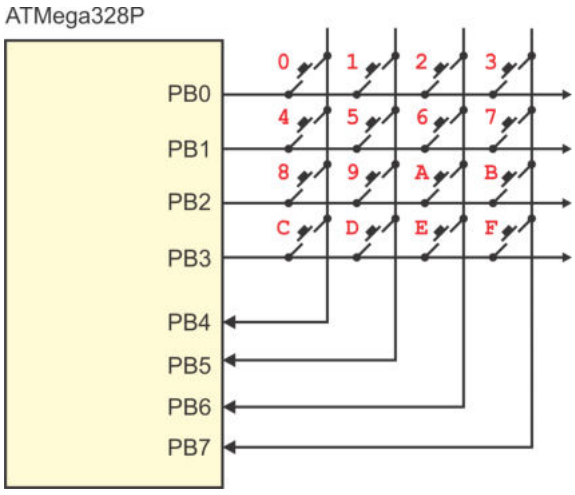


Figura 10.4: Conexión de un teclado de 4 x 4 con un ATMega328P

En general, para un teclado de n x m botones, el valor de la tecla está determinado por la ecuación:

$$tecla = m \times renglón + columna$$

Ejemplo 10.1 - Función para un teclado de 4×4
Utilizando lenguaje C, realice una función que revise si hay una tecla presionada en un teclado matricial de 4×4 *conectado en el Puerto B de un ATMega328P. La función debe regresar el valor de la tecla o 0xFF (-1) si no hubo tecla presionada.*

La conexión del teclado con el MCU se muestra en la Figura 10.4. 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.

El código en lenguaje C de la función es:

```
uint8_t  teclado( ) {
uint8_t  secuencia[] = {0xFE, 0xFD, 0xFB, 0xF7 };
uint8_t  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
       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 y se ubica en la memoria de código, dado que se evita el almacenamiento en SRAM cada que la función es invocada.

Opciones para el Sondeo de un Teclado

La función que sondea un teclado se puede invocar dentro del lazo infinito de una aplicación, en esos casos, es conveniente introducir un retardo entre cada llamada porque cuando se presiona una tecla, el lapso de tiempo que está presionada es mucho mayor al requerido por el sondeo del teclado, sin ese retardo, un programa podría interpretar que la misma tecla se ha presionado muchas veces. La duración del retardo depende de las actividades a realizar cada vez que se presiona una tecla.

Una segunda alternativa es el uso de un temporizador, configurado para que interrumpa a la CPU cada 200 o 300 ms, en su ISR se realizaría la llamada a la función que sondea el teclado y, en caso de detectar una tecla presionada, se le puede dar atención en la misma ISR o bien, su valor se almacena en una variable global y se modifica una bandera que se revisaría como parte del lazo infinito, con ello, se tiene la notificación de una tecla presionada y el valor de la misma.

La última opción consiste en el uso de las interrupciones por cambios en las terminales, las 4 entradas podrán generar un evento si en las 4 salidas se deja un nivel bajo, el evento se producirá con cualquier tecla presionada, luego, en la ISR se deberá llamar a la función que hace el sondeo para obtener su valor. Puesto que la ISR se va a ejecutar solo cuando ha ocurrido un evento, es conveniente que contenga el código que dé respuesta a la tecla presionada, aunque, se puede modificar una bandera que se sondearía como parte del lazo infinito.

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 10.5 se muestra la conexión de un MM74C922 con un ATMega328P.

Para reducir el número de terminales empleadas por el microcontrolador, es posible conectar una compuerta NOT 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.

Los decodificadores integrados son un buen complemento para simplificar el software de un sistema y ahorrar terminales de un ATMega328P. No obstante, su precio es comparable con el precio de un microcontrolador, por lo tanto, se podría dedicar a un ATMega328P secundario a realizar el sondeo de un teclado y que interrumpa a un MCU principal, notificando que se ha presionado una tecla. Al emplear otro MCU se tiene la ventaja de que la notificación no solo se puede realizar con una interrupción externa, se puede utilizar cualquiera de las interfaces seriales disponibles en el ATMega328P.

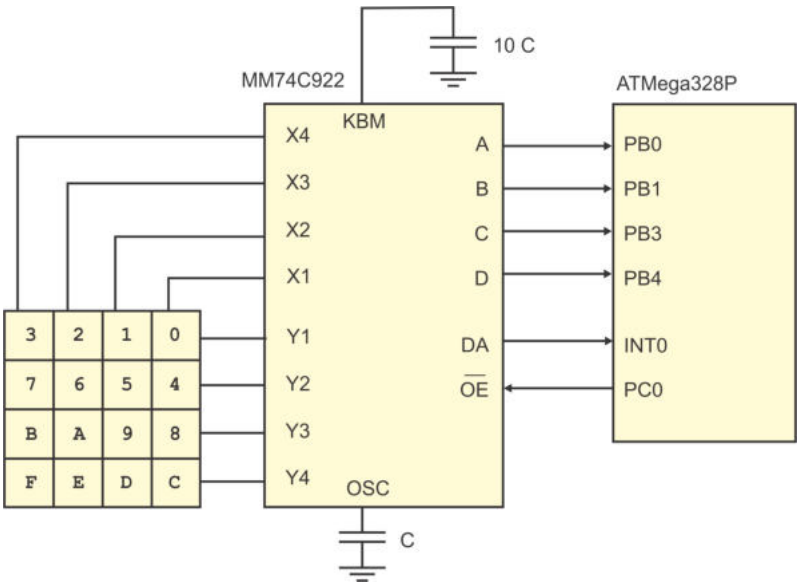


Figura 10.5: Conexión de un MM74C922 con un ATMega328P

10.3. 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).

Figura 10.6: Conexión de un LED (a) con lógica positiva y (b) con lógica negativa

En la Figura 10.6 se muestran 2 formas de conectar un LED con un MCU, en (a) se utiliza lógica positiva, es decir, un 1 lógico en la terminal del puerto enciende al LED. En (b) se utiliza lógica negativa, el LED se enciende con un 0 lógico.

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.

El resistor R es para limitar la corriente, con esto se protege tanto al microcontrolador como al LED. Cada terminal del ATMega328P puede suministrar hasta 20 mA de corriente y este valor puede resultar excesivo para algunos LEDs pequeños, además, el fabricante recomienda que la corriente total en las terminales no exceda a 100 mA por puerto. El valor de R depende de la intensidad de la corriente que se desee hacer circular en el LED, se obtiene con la ley de Ohm:

$$i = \frac{V_{CC} - V_{LED}}{R}, \text{ o bien, } R = \frac{V_{CC} - V_{LED}}{i}$$

El voltaje en el LED depende de su color, puede estar entre 1.5 y 2.5 V, y para que ilumine con una intensidad media, la corriente que debe circular debe estar entre 5 y 10 mA. Por ello, un valor típico para R es de $330\,\Omega$, con este valor, la corriente en el LED estará en el rango de 7.5 a 10.6 mA, adecuado para la mayoría de LEDs, si el voltaje de alimentación del circuito es de 5 V.

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 todos los LEDs y en el segundo tipo, son los cátodos los que se conectan en el mismo nodo. En la Figura 10.7 se muestran ambos tipos de displays y la forma en que se pueden conectar a uno de los puertos de un MCU.

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 10.7 se ha considerado el uso de 1 resistor en cada segmento, no obstante, a veces se prefiere utilizar solo 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.

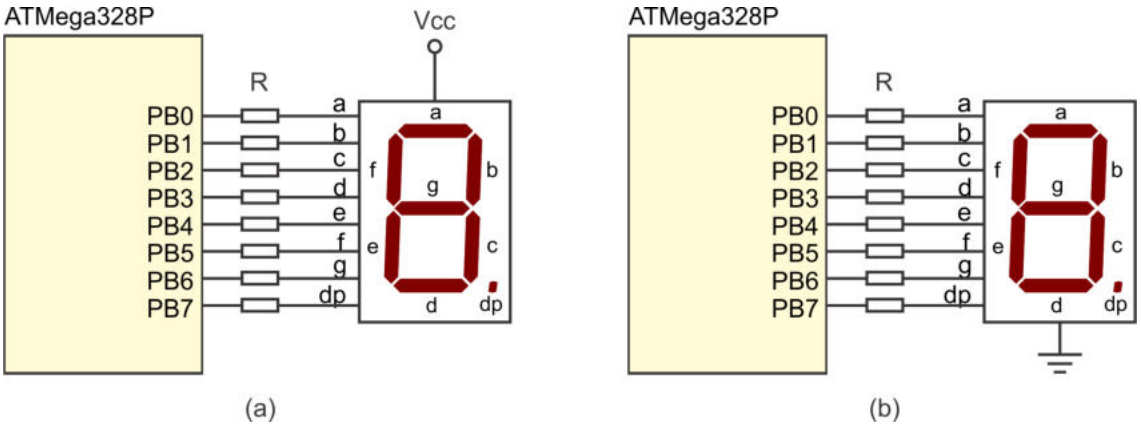


Figura 10.7: Conexión de un display de (a) ánodo común y (b) cátodo común

En muchas aplicaciones de MCUs es frecuente el manejo de más de un display de 7 segmentos, pueden ser 2, 3, 4 o más. En esos casos, no conviene dedicar un puerto para cada display porque el ATMega328P solo tiene 3 y pueden ser insuficientes, además, no se tendría disponibilidad de terminales 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, se tiene la apariencia de que todos los displays están encendidos al mismo tiempo.

En la Figura 10.8 se muestra la conexión de 4 displays de 7 segmentos con un ATMega328P, esta conexión a través de 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 para que el ATMega328P solo 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. Los resistores R1 y R2 permiten limitar este flujo de corriente, su valor depende de la luminosidad deseada en los displays. El resistor R2 podría omitirse y limitar la corriente únicamente con R1.

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.

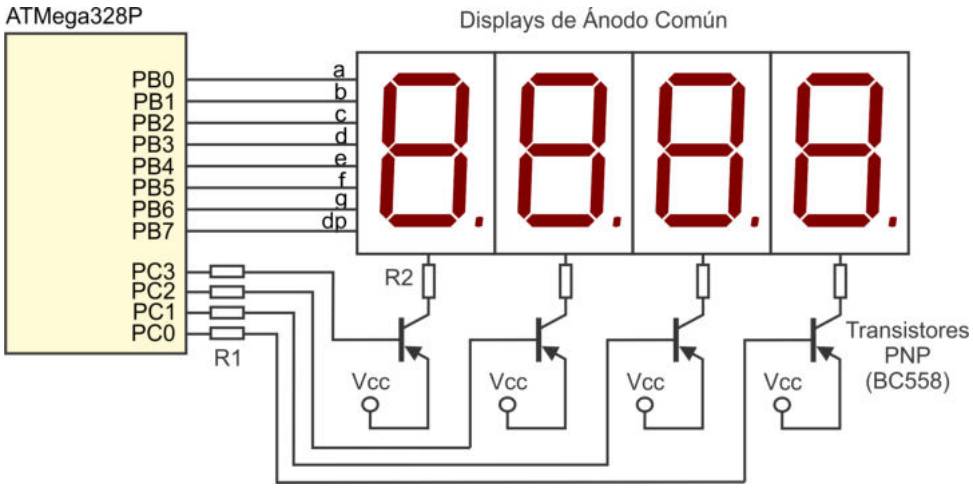


Figura 10.8: Conexión de 4 displays de 7 segmentos con un bus común para los datos

Ejemplo 10.2 - Función para el manejo de 4 displays
Considerando el hardware de la Figura 10.8, 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 y que la posición 0 del arreglo corresponde con el display que está ubicado en el extremo derecho.

La función debe recibir el arreglo con la información a mostrar, codificada con lógica negada por ser displays de ánodo común, su código es:

```
// Constantes a ubicarse en un ámbito global
const uint8_t habs[] PROGMEM = { 0x0E, 0x0D, 0x0B, 0x07 };

// Función para la exhibición de datos
void mostrar(uint8_t datos[]) {
uint8_t  i;

 for( i = 0; i < 4; i++) {
   PORTB = datos[i];                  // Envía el dato
   PORTC = pgm_read_byte(&habs[i]); // Habilita el display
    _delay_ms(5);                     // Espera que se vea adecuadamente
   PORTC = 0x0F;                      // Deshabilita para no introducir
 }                                    // ruido al siguiente display
}
```

Las constantes para los habilitadores deben ubicarse en un ámbito global, antes de la función principal, se han configurado para que se ubiquen en la memoria de código.

Después de la configuración de los puertos se debe asignar PORTC = 0x0F para que los displays inicien deshabilitados, estos quedan de la misma manera cuando la función concluye, ya que esta función sería invocada continuamente, como parte del lazo infinito de un programa, para mantener visible el contenido de los displays.

Opciones para usar menos terminales de un ATMega328P

Es natural que se podría dedicar un ATMega328P para el manejo de un conjunto de displays de 7 segmentos, este recibiría la información a mostrar a través de una de las interfaces seriales disponibles, la cual sería enviada desde otro microcontrolador. No obstante, con la ayuda de circuitos integrados auxiliares se puede reducir el número de terminales utilizadas en el ATMega328P, en esta sección se revisan dos opciones.

El CI MC74HC595A es un registro de desplazamiento de 8 bits que se puede emplear para el manejo de displays de 7 segmentos. El MC74HC595A internamente contiene un registro de desplazamiento, un registro de almacenamiento y un *buffer* de 3 estados, los registros tienen relojes independientes, con ello, los resultados parciales del registro de desplazamiento quedan ocultos hasta que se habilita el registro de almacemanmiento. Además, el circuito contiene una salida serial de datos para su conexión en cascada con circuitos similares.

Para el manejo de 4 displyas de 7 segmentos se pueden emplear 2 circuitos del tipo MC74HC595A, en la Figura 10.9 se muestra una forma para conectarlos, en el ATMega328P solo se utilizan 3 terminales: PB0 para enviar los datos en forma serial (DS), PB1 es el reloj para el registro de desplazamiento (SHCP) y PB2 es el reloj para el registro de almacemanmiento (STCP).

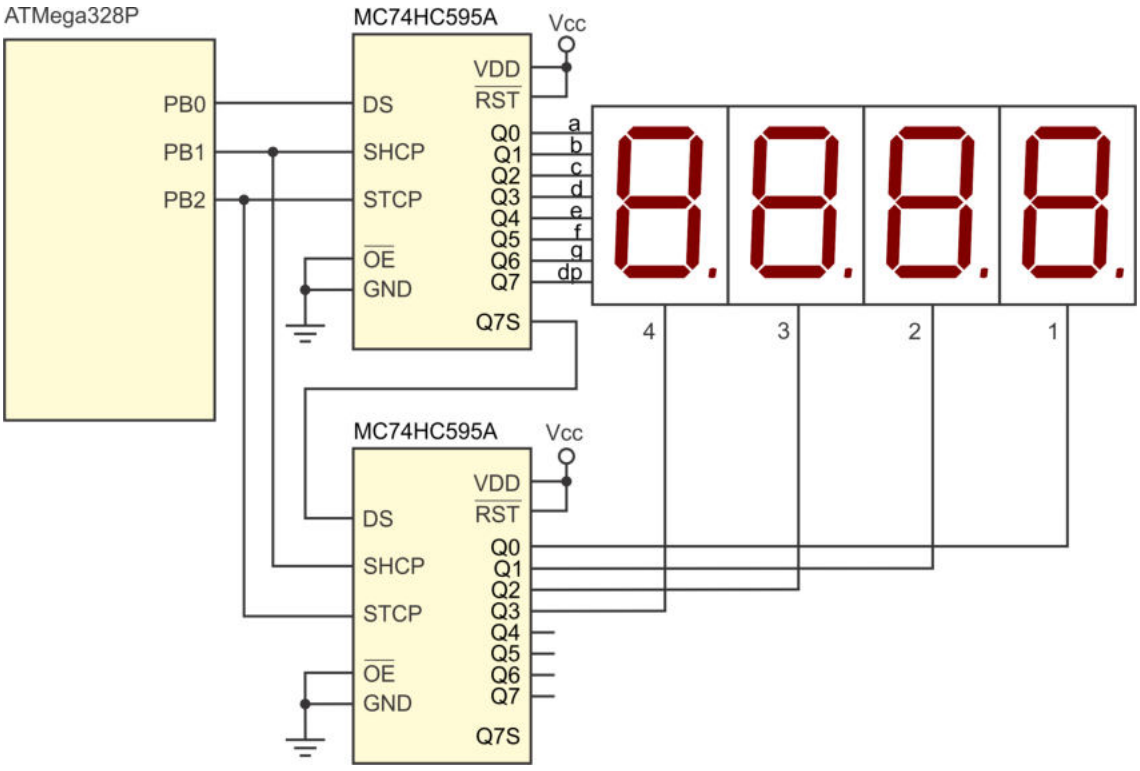


Figura 10.9: Uso del CI MC74HC595A para el manejo de 4 displays con 3 terminales

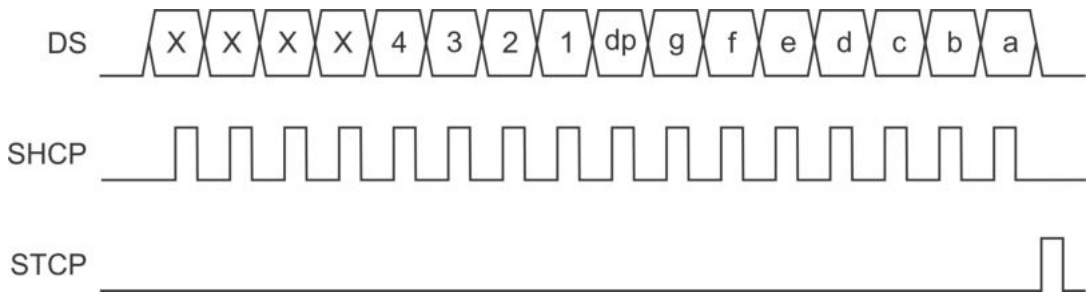


Figura 10.10: Señales para el manejo de 4 displays con el apoyo de registros de desplazamiento

Para exhibir un dato se debe enviar una trama de 16 bits, sincronizando con el reloj de desplazamiento y, al final de la trama, el dato quedará capturado en la salida de los registros cuando se coloque un pulso en el reloj de almacenamiento.

El orden en que se colocan los datos en la terminal DS depende de la forma en que se realizan las conexiones, en la Figura 10.10 se muestran las tramas en las señales considerando las conexiones de la Figura 10.9, se debe enviar una trama por cada display, haciendo un recorrido continuo en los 4 displays para que aparente que todos están encendidos al mismo tiempo, este recorrido sería parte del ciclo infinito de una aplicación y es conveniente dejar un retardo de 5 ms entre cada trama de 16 bits, para que los displays alcancen una intensidad adecuada.

Con el hardware de la Figura 10.9 se pueden manejar displays de ánodo o cátodo común, dependiendo del tipo elegido, se determina el nivel lógico de activación de cada segmento (a, b, .., dp) y el nivel lógico para activar un display (1, 2, 3 o 4), dejando inactivos a los otros.

Analizando la Figura 10.9, se observa que el hardware se puede complementar para manejar hasta 8 displays de 7 segmentos, en las señales marcadas como No Importa (X) de la línea de datos (Figura 10.10), se haría la activación de los displays adicionales. El recorrido continuo ahora sería sobre el total de displays conectados.

Con el uso de los registros MC74HC595A también se puede manipular una matriz de 8 x 8 LEDs, un registro se emplearía para enviar la información a todos los renglones y el otro para habilitar solo una de las columnas.

Otra opción para el manejo de varios displays de 7 segmentos con pocas terminales, consiste en el uso del CI MAX7219, este es un controlador con el que se pueden manejar hasta 8 displays de cátodo común, el circuito es manipulado por medio de la interfaz SPI. Dentro del entorno de Arduino se cuenta con una biblioteca para su manejo, sin embargo, no es complejo manejarlo directamente desde el ATMega328P. En la Figura 10.11 se muestra la conexión del circuito con 4 displays, solo se requiere un resistor externo para determinar la corriente máxima que se manejará para cada segmento.

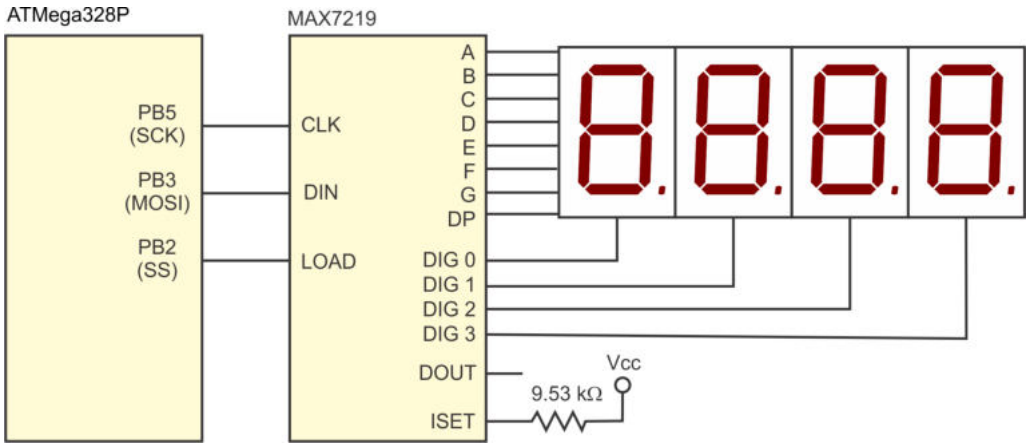


Figura 10.11: Uso del CI MAX7219 para el manejo de 4 displays vía SPI

El MAX7219 se maneja mediante tramas SPI de 16 bits, dado que esta interfaz es de 8 bits en el ATMega328P, se deben hacer dos transferencias para el envío de información. El circuito tiene 14 registros para su configuración, en la Tabla 10.2 se muestran los registros, con una breve descripción. Una trama de 16 bits se compone del número de registro a modificar y el valor a escribir en ese registro.

El MCU no recibe respuesta del MAX7219, la terminal DOUT del MAX no es para conectarse con la terminal MISO del ATMega328P, esta terminal es para una conexión en cascada de más de un MAX7219. La salida DOUT se conecta con la entrada DIN del siguiente MAX7219, en esos casos, desde el MCU se deben enviar tramas con una longitud múltiplo de 16 bits, así, un dispositivo se queda con 16 bits y deja pasar los siguientes. El registro No-Op es empleado para que un CI deje pasar algunos bits sin algún efecto.

El MAX7219 se encarga de hacer el refresco en los displays conectados, pudiendo ser hasta 8, en el registro 0x0B se indica este número. El dato a mostrar en cada display se escribe en su registro correspondiente (de 0x01 a 0x08), es posible colocar el valor numérico si se ha habilitado el decodificador en el registro 0x09, en caso contrario, se debe colocar el código en 7 segmentos. La elección del uso del decodificador de 7 segmentos es independiente en cada display, en el registro 0x09 se deben colocar 1's en las posiciones de bits que corresponden con los displays en donde si será utilizado.

10.4. 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 cuentan con un espacio para escribir caracteres personalizados.

Tabla 10.2: Mapa de registros del MAX7219

Número	Nombre	Descripción
0x00	No-Op	Empleado para una operación en cascada
0x01	Digito 0	Valor a mostrar en el display 0
0x02	Digito 1	Valor a mostrar en el display 1
0x03	Digito 2	Valor a mostrar en el display 2
0x04	Digito 3	Valor a mostrar en el display 3
0x05	Digito 4	Valor a mostrar en el display 4
0x06	Digito 5	Valor a mostrar en el display 5
0x07	Digito 6	Valor a mostrar en el display 6
0x08	Digito 7	Valor a mostrar en el display 7
0x09	Decodificador	Habilita un decodificador de 7 segmenos
0x0A	Intensidad	Establece la intensidad entre 16 niveles posibles
0x0B	Límite	Define el número de displays para una exhibicón continua
0x0C	Apagado	Se apagan los displays
0x0F	Prueba	Se encienden todos los segmentos

En los LCDs alfanuméricos, 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, elementos geométricos como líneas, rectángulos, círculos, etc., así como imágenes pequeñas. Un GLCD está organizado como una matriz de pixeles, por ejemplo de 128 x 64. El manejo de un GLCD generalmente se realiza con el apoyo de una biblioteca proporcionada por su fabricante.

En esta sección únicamente se hace referencia a LCDs alfanuméricos, específicamente de 16 x 1 y de 16 x 2. En la Figura 10.12 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 10.3 se describe la funcionalidad de cada terminal.

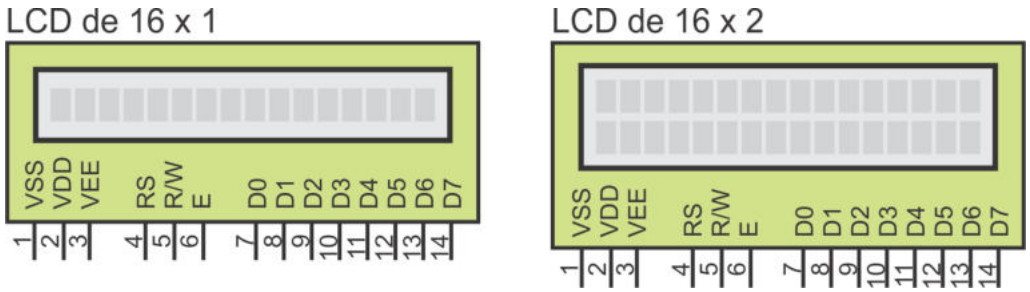


Figura 10.12: Disposición de terminales en un LCD de 16 x 1 y en uno de 16 x 2

Cabe señalar que los LCDs más modernos incluyen otras dos terminales (15 y 16), denominadas A y K (por ánodo y cátodo), para suministrar energía a un LED que ilumina la pantalla.

Tabla 10.3: Función de las terminales en un LCD

Número	Nombre	Función
1	VSS	Tierra
2	VDD	Voltaje de alimentación
3	VEE	Contraste
4	RS	Selecciona un comando (RS = 0) o un dato (RS = 1)
5	R/W	Selecciona una escritura (R/W = 0) o una lectura (R/W = 1)
6	E	Habilitación del LCD
7 - 14	D0 - D7	Bus de datos

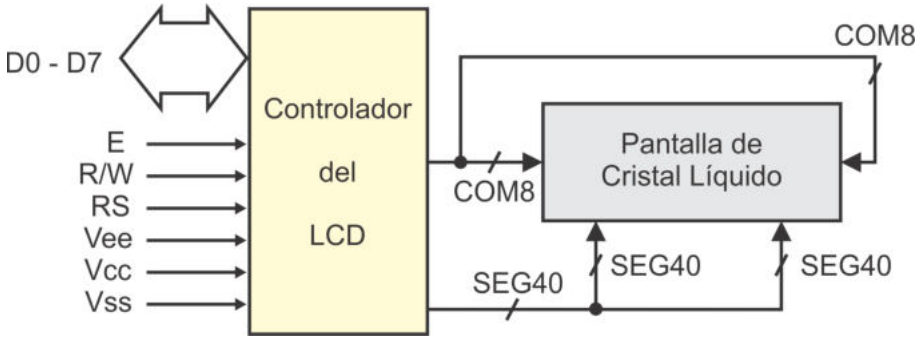


Figura 10.13: El controlador se encarga de manejar la pantalla de cristal líquido

Un LCD incluye un circuito controlador, este 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 10.3, 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 10.13 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.

10.4.1. 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 solo 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.

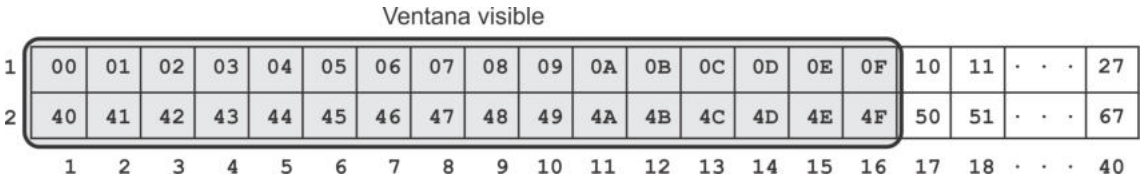


Figura 10.14: En la pantalla se visualizan 32 de los 80 caracteres disponibles

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 solo 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 son de la 0x00 a la 0x27 y en la segunda línea 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 10.14 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 ese espacio.

Para el acceso a la DDRAM se tienen diferentes comandos, con el comando *Configura Dirección en DDRAM* se ubica al contador de dirección, se debe aplicar antes de una escritura o lectura. 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 decrementar 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 10.15 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.

La dirección de cada mapa de bits es de un byte, en el primer renglón de la Figura 10.15 se muestra el nibble alto del byte de dirección y en la primera columna su nibble bajo. Puede notarse que para los caracteres ASCII imprimibles su dirección es la correcta, 0x20 para el espacio en blanco, 0x21 para '!', etc., hasta 0x7D que es la codificación de '}'.

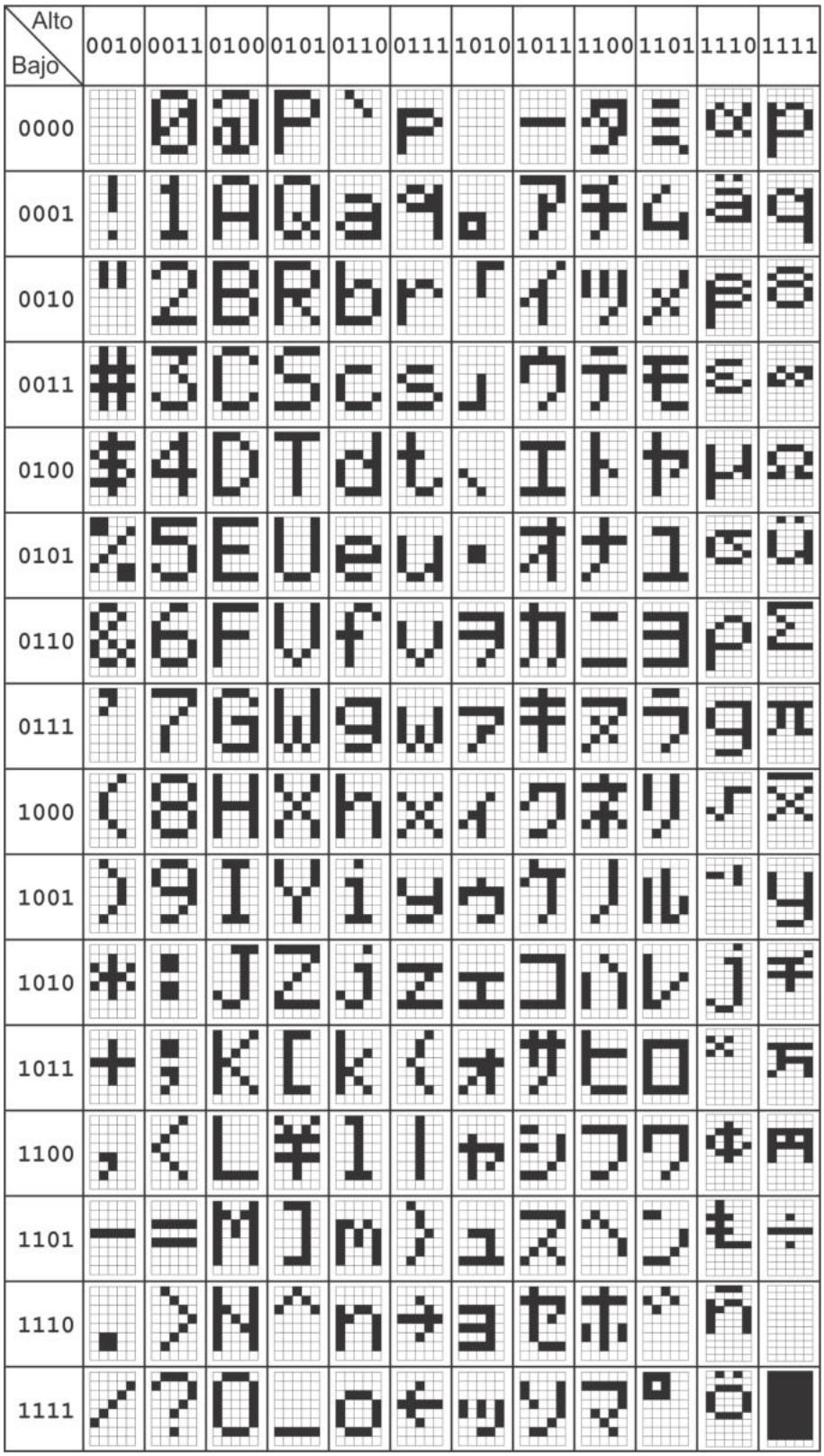


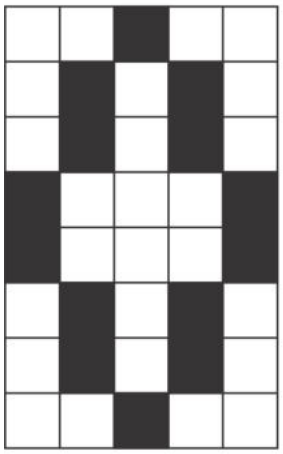
Figura 10.15: Contenido de la CGROM

La DDRAM y la CGROM están directamente 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, este 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. La impresión de un mapa de bits de la CGRAM en la pantalla LCD se realiza de la misma forma en que se imprime 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.

La CGRAM, a diferencia de la CGROM, si puede ser leída o escrita por el usuario, pero como el contador de direcciones es compartido por la DDRAM y la CGRAM, el acceso a uno u otro espacio se determina por el último comando utilizado para definir la dirección de acceso, pudiendo ser *Configura Dirección en DDRAM* o *Configura Dirección en CGRAM*. Si se utiliza la segunda opción, las llamadas a los comandos *Lee Dato* y *Escribe Dato* harán referencia a CGRAM y no a DDRAM.

Las direcciones para la CGRAM son de 6 bits, de los cuales, los 3 bits más significativos determinan la ubicación del mapa de bits (son 8 mapas) y los 3 bits menos significativos hacen referencia a cada uno de sus renglones. Por ser mapas de 5 x 8 pixeles, únicamente los 5 bits menos significativos en cada dato son válidos. Como un ejemplo, en la Figura 10.16 se muestra el dibujo de un rombo con la información a colocar en la CGRAM, en sus respectivas direcciones, suponiendo que será ubicado en el mapa de bits con dirección 0x05.



Dirección						Dato				
1	0	1	0	0	0	0	0	1	0	0
1	0	1	0	0	1	0	1	0	1	0
1	0	1	0	1	0	0	1	0	1	0
1	0	1	0	1	1	1	0	0	0	1
1	0	1	1	0	0	1	0	0	0	1
1	0	1	1	0	1	0	1	0	1	0
1	0	1	1	1	0	0	1	0	1	0
1	0	1	1	1	1	0	0	1	0	0

Mapa Renglón

Figura 10.16: Mapa de bits personalizado para la dirección 0x05 de CGRAM

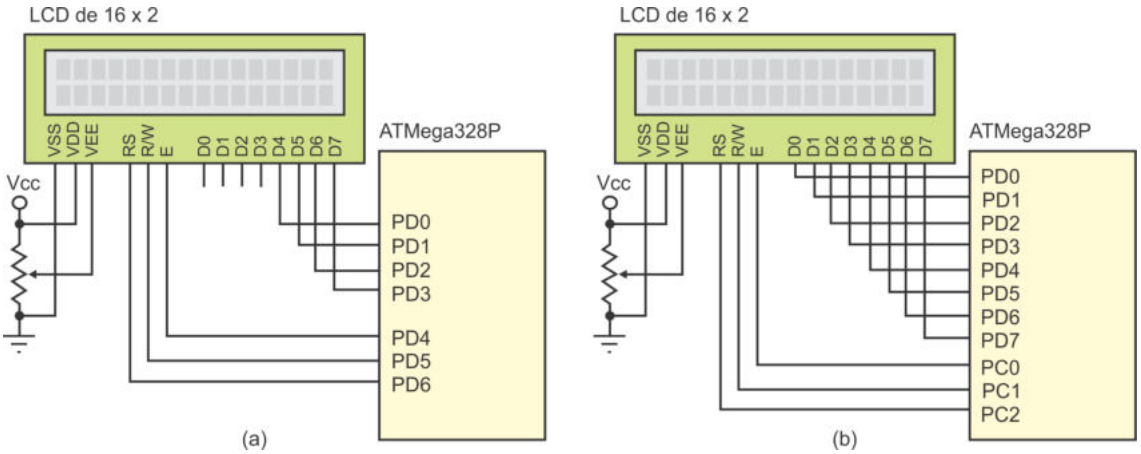


Figura 10.17: Conexión de un LCD: (a) interfaz de 4 bits y (b) interfaz de 8 bits

10.4.2. Conexión de un LCD con un Microcontrolador

Aunque el bus de datos de un LCD es de 8 bits (ver Figura 10.12), para conectarse con un MCU es posible utilizar una interfaz de 4 o de 8 bits. En la Figura 10.17(a) se muestra la conexión de un LCD con un ATMega328P por medio de una interfaz de 4 bits, del bus de datos del LCD se utilizan los 4 bits más significativos. La interfaz de 8 bits requiere más terminales, es necesario emplear un puerto completo más 3 bits de otro puerto, como se puede ver en la Figura 10.17(b). La interfaz de 4 bits requiere menos terminales pero es más lenta, porque los datos son de 8 bits. Una interfaz de 4 bits realiza 2 accesos al LCD para escribir o leer un dato.

Para simplificar el hardware se puede conectar la terminal R/W permanentemente con tierra y solo realizar escrituras. Las lecturas del LCD sirven para saber si está listo y para obtener el valor del contador de dirección. La lectura de la bandera de listo se puede omitir si el MCU espera el tiempo requerido por cada comando. En cuanto al contador de dirección, su valor se define desde el MCU, de manera que no debería ser necesario obtenerlo del LCD. Por lo tanto, es posible manipular un LCD realizando únicamente escrituras.

10.4.3. Ciclos de Escritura y Lectura en un LCD

Las señales de control deben sincronizarse durante los ciclos de escritura y lectura. En la Figura 10.18 se muestra la temporización y niveles de voltaje de las señales para un ciclo de escritura. 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.

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 0 cuando se va a escribir un comando y 1 si la escritura es de un dato. R/W debe tener 0 en los ciclos de

escritura. La duración del pulso de habilitación debe ser al menos de 450 ns (AP_{EN}). Se observa en la figura que la escritura se sincroniza con 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}).

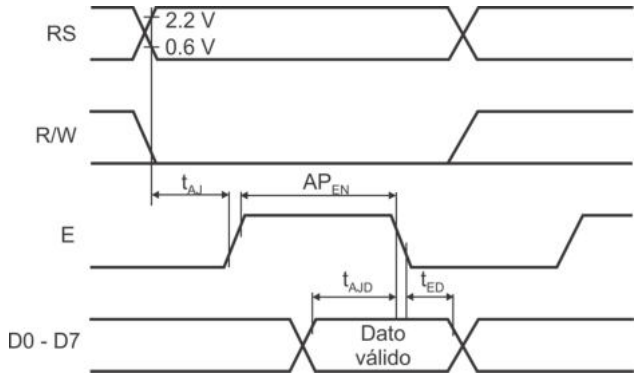


Figura 10.18: Ciclo de escritura en un LCD

Un ciclo de lectura es similar a uno de escritura, en niveles de voltaje y algunos tiempos, esto puede verse en la Figura 10.19. 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.

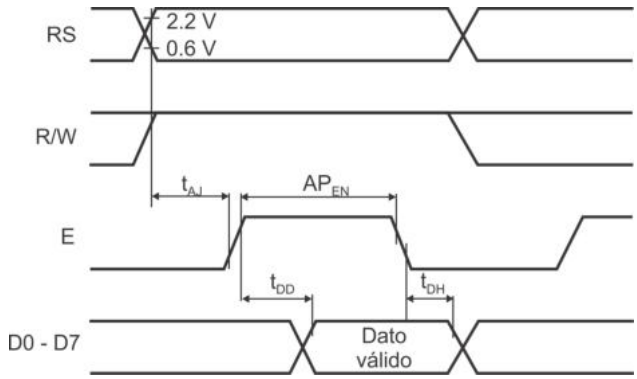


Figura 10.19: Ciclo de lectura en un LCD

Cuando se conecta el LCD con un ATMega328P u otro MCU, 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.

La función para generar el pulso en la terminal E, asumiendo que el LCD está conectado como se muestra en la Figura 10.17(a), es:

```
void LCD_pulso_E( ) {
   PORTD = PORTD | 0x10;      // Nivel alto en la terminal E
   PORTD = PORTD & 0xEF;      // Nivel bajo en la terminal E
}
```

Con esta función, el algoritmo para un ciclo de escritura debe seguir los pasos:

1. Asignar a RS el valor de 0 o 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 un ciclo de lectura, los pasos del algoritmo son:

1. Asignar a RS el valor de 0 o 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 us, primero se debe escribir o leer al nibble más significativo e inmediatamente al menos significativo. El tiempo de 1 us para el cambio de nibbles se consume en la ejecución de las instrucciones. En la Figura 10.20 se comparan las transferencias de 4 y 8 bits, en una secuencia que escribe un comando, luego escribe un dato y finalmente lee un comando.

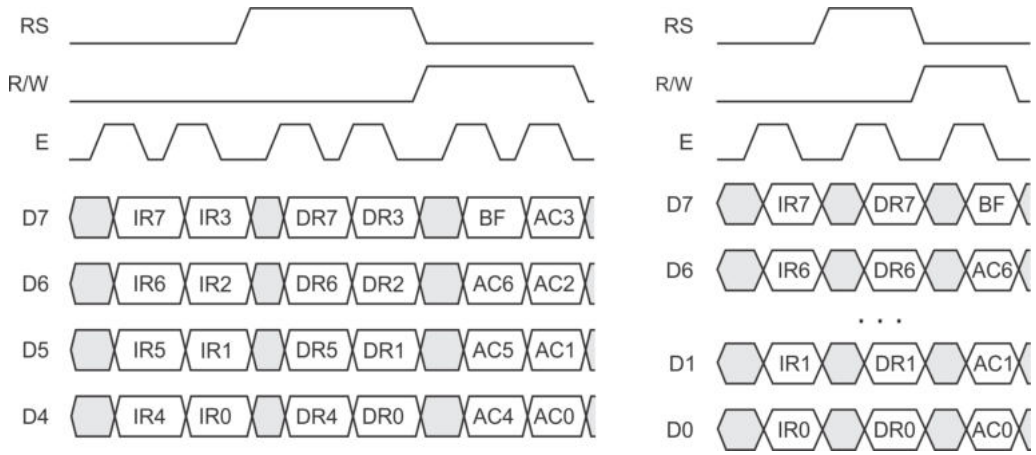


Figura 10.20: Transferencias de datos en una interfaz (a) de 4 bits y (b) de 8 bits

10.4.4. Comandos para el Manejo de un LCD

El manejo de un LCD se realiza a través de comandos, por medio de ellos es posible configurarlo, apagarlo, desplazar la ventana visible, ubicar y mover el cursor, etc. En la Tabla 10.4 se muestran los comandos disponibles, incluyendo las opciones que se tienen para la escritura y lectura de datos. En los siguientes apartados se describe la funcionalidad de cada uno de ellos.

Tabla 10.4: Comandos del LCD

			Nibble alto				**Nibble bajo**			
Comando	**RS**	**RW**	**DB7**	**DB6**	**DB5**	**DB4**	**DB3**	**DB2**	**DB1**	**DB0**
Limpieza del display	0	0	0	0	0	0	0	0	0	1
Regreso del cursor 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	I/D	S
Encendido/apagado del display	0	0	0	0	0	0	1	D	C	B
Desplaza el cursor y el 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 direccón y bandera de ocupado	0	1	BF	A6	A5	A4	A3	A2	A1	A0
Escribe dato en CGRAM o DDRAM	1	0	D7	D6	D5	D4	D3	D2	D1	D0
Lee dato en CGRAM o DDRAM	1	1	D7	D6	D5	D4	D3	D2	D1	D0

Limpieza del display

Este comando se emplea para limpiar al display y regresar el cursor a la posición de inicio, que corresponde con 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 us y 1.64 ms.

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 us y 1.6 ms.

Tabla 10.5: Argumentos del comando Ajuste de entrada de datos

	Bit I/D	**Bit S**
0	El contador de dirección se auto-decrementa	Desplazamiento deshabilitado
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

Tabla 10.6: Argumentos del comando Encendido/apagado del display

	Bit D	**Bit C**	**Bit B**
0	Display apagado	Sin cursor	Sin parpadeo
1	Display encendido	Con cursor	Con parpadeo

Ajuste de entrada de datos

Con este comando se configura la dirección de movimiento del cursor y se especifica si el display va a ser desplazado. Estas acciones se realizan después de la lectura o escritura de un dato. El comando requiere de 40 us para su ejecución y tiene 2 argumentos, los cuales se describen en la Tabla 10.5.

Encendido/apagado del display

Este comando es 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 us para su ejecución y sus argumentos se describen en la Tabla 10.6.

Desplaza el cursor y el display

Desplaza al cursor y al display sin tener acceso a la DDRAM, es decir, sin realizar alguna escritura o lectura de datos. El comando requiere de 40 us para su ejecución y contiene dos argumentos, los cuales combinan sus valores para diferentes acciones, como se describe en la Tabla 10.7.

Configura la función del display

Con este comando se configuran 3 parámetros determinantes para la operación del LCD: el tamaño de la interfaz, el número de líneas y el tamaño de los mapas de bits. El comando requiere de 40 us para su ejecución y sus argumentos se describen en la Tabla 10.8. Algunos LCDs que se ven de 16 caracteres en 1 línea (16 x 1), funcionan como displays de 8 x 2. Deben 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.

Tabla 10.7: 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 se decrementa
0	1	Desplaza al cursor a la derecha, el contador de direcciones se incrementa
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

Tabla 10.8: 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

Configura dirección en CGRAM

Este comando es para configurar la dirección de acceso a la CGRAM, se dispone de 6 bits para colocar la dirección. El contador de direcciones queda referenciando a la CGRAM, de manera que las subsecuentes escrituras y lecturas de datos se harán en CGRAM. El comando requiere de 40 us para su ejecución.

Configura dirección en DDRAM

Este comando es para configurar la dirección de acceso a la DDRAM, se dispone de 7 bits para colocar la dirección. El contador de direcciones queda referenciando a la DDRAM, de manera que las subsecuentes escrituras y lecturas de datos se harán en DDRAM. El comando requiere de 40 us para su ejecución.

Lee dirección y bandera de ocupado

Este es el único comando de lectura, como se observa en la Tabla 10.4, 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 sería posible ejecutar otro comando en el LCD. El comando requiere de 1 us 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.

Escribe dato en CGRAM o 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 utilizado 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 us, después de esta, el contador de dirección se incrementa o decrementa, según como se haya definido con el comando *Ajuste de entrada de datos*. También se realiza el desplazamiento del display, si fue habilitado con el mismo comando.

Lee dato en CGRAM o 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 utilizado 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 us, después de esta, el contador de dirección se incrementa o decrementa, según como se haya definido con el comando *Ajuste de entrada de datos*. En contraste con la escritura, una lectura no realiza desplazamientos del display.

10.4.5. Inicialización del LCD

Un LCD debe inicializarse para que pueda mostrar información en la pantalla. La inicialización consiste en el envío temporizado de una secuencia de comandos, mediante esta secuencia se establece el tamaño de la interfaz y diferentes características para la exhibición de los datos. La secuencia es proporcionada por los fabricantes y los tiempos establecidos son fundamentales para la correcta inicialización del LCD.

En la Figura 10.21 se muestra la secuencia a seguir para inicializar al LCD con una interfaz de 4 bits. Aunque el bus de datos del LCD es de 8 bits, únicamente se conectan los 4 bits más significativos (ver Figura 10.21). 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 se envía el comando *Configura la función del display* con el bit DL (DB4) en alto para indicar una interfaz de 8 bits. Es en el paso 4 en donde la interfaz cambia a 4 bits, siendo el último paso en el que solo se mandan los 4 bits más significativos.

Después del paso 3, en la Figura 10.21 no se indican los tiempos de espera debido a que ya es posible verificar la bandera de ocupado (*Busy Flag*). Si no se desean hacer

lecturas, una vez establecido el comando, el MCU debe esperar el tiempo necesario para su ejecución.

Se observa en la secuencia que en los primeros 3 pasos solo se enviaron los 4 bits más significativos y a partir del paso 4 se mandan 8 bits, primero al nibble alto e inmediatamente después al nibble bajo, dado que la espera entre nibbles solo es de 1 us. El paso 8 puede omitirse si la configuración obtenida con el comando *Limpieza del display* es adecuada para la aplicación.

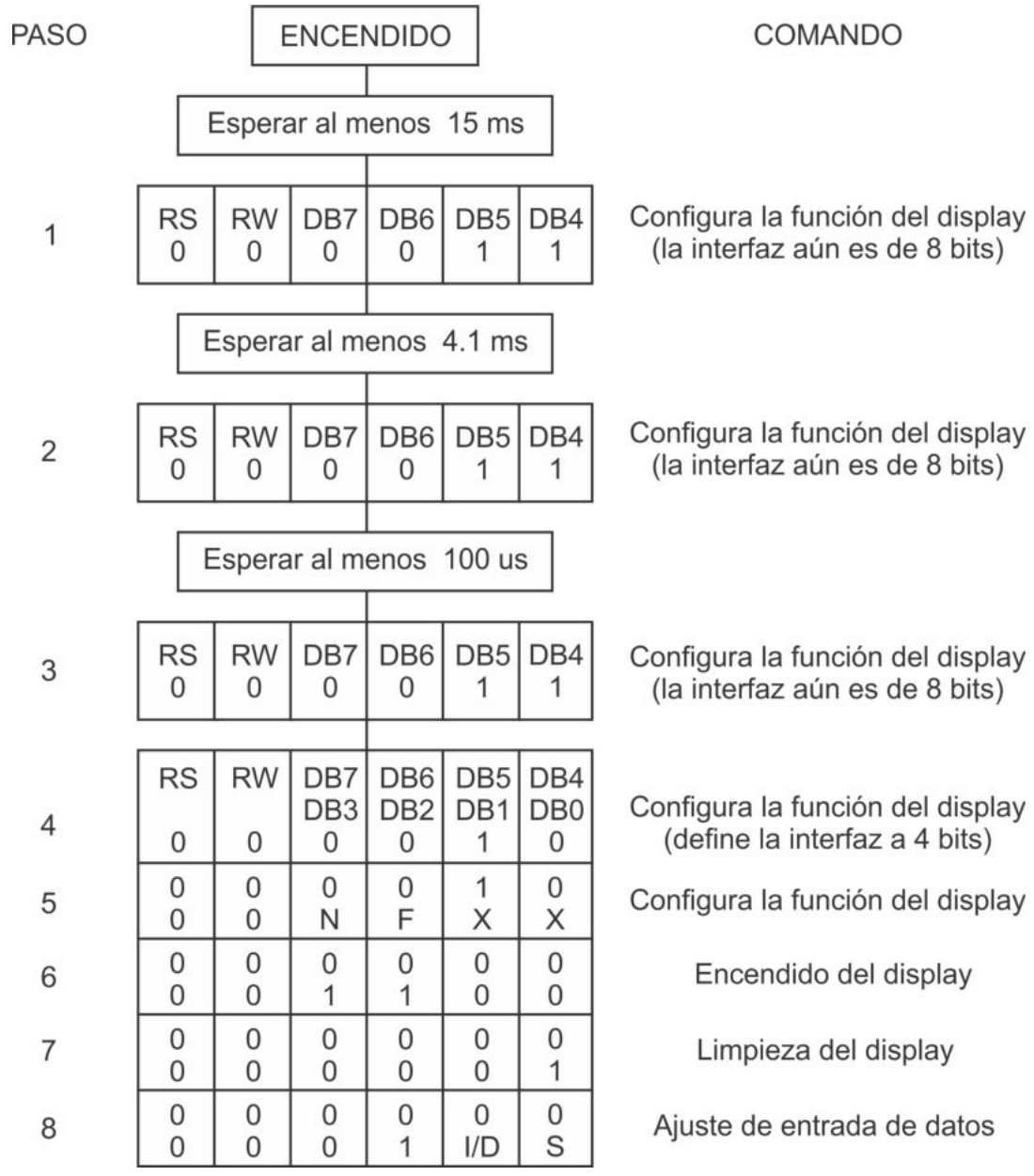


Figura 10.21: Secuencia para inicializar un LCD con una interfaz de 4 bits

La inicialización del LCD con una interfaz de 8 bits es similar a la de 4 bits, en la Figura 10.22 se muestra la secuencia requerida para ello. Tampoco se indica el tiempo de espera después del paso 3 porque ya es posible monitorear la bandera de ocupado (BF), o bien, esperar el tiempo requerido por el correspondiente comando.

La secuencia tiene un paso menos porque cuando se confirma la interfaz de 8 bits, también se definen las características del LCD. 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 10.4.3.

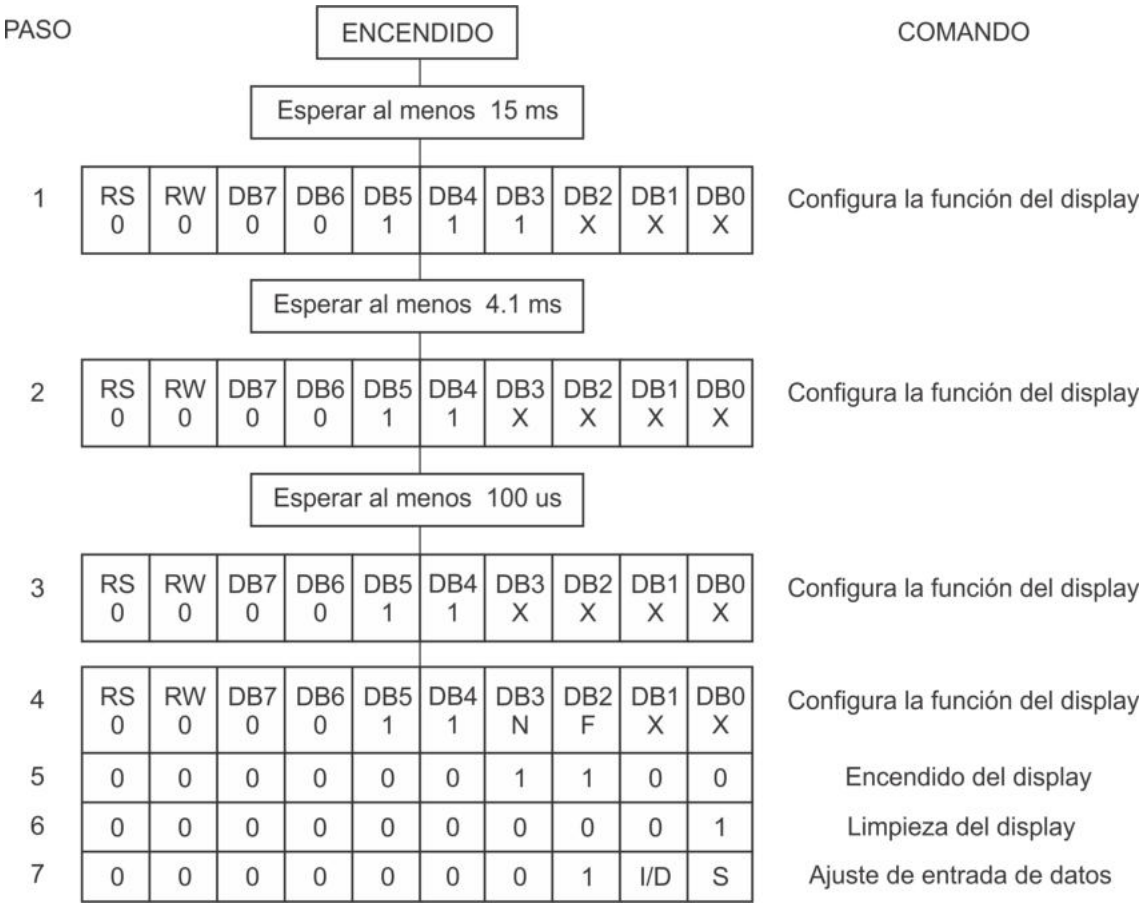


Figura 10.22: Secuencia para inicializar un LCD con una interfaz de 8 bits

10.4.6. Biblioteca para el Manejo de un LCD

En esta sección se describen las funciones de una biblioteca para el manejo de un LCD con un ATMega328P, dado que este MCU solo tiene 3 puertos, lo más con-

veniente es el uso de una interfaz de 4 bits. También se considera la omisión de lecturas, para dejar la terminal R/W conectada a tierra y reducir a 6 las terminales requeridas por el MCU. De esta forma, el LCD también puede ser manejado desde el Puerto C del ATMega328P sin comprometer la terminal de *reset* ubicada en PC6.

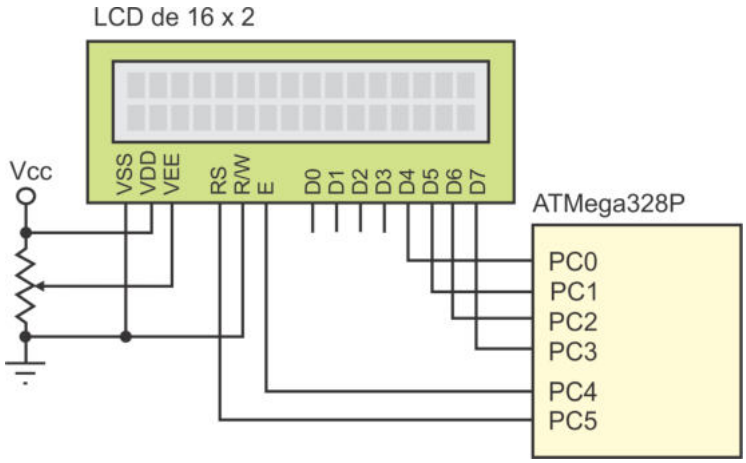


Figura 10.23: Conexón de un LCD con un ATMega328P empleando solo 6 terminales

En la Figura 10.23 se muestra la conexión del LCD en el Puerto C del ATMega328P, el orden en la ubicación de las terminales es empleado en las funciones de la biblioteca, pero el puerto se establece con la definición:

```
#define  P_LCD  PORTC
```

Para cambiar de puerto es suficiente con indicar cual será el nuevo puerto para la conexión del LCD. Con la definición establecida y considerando que el MCU va a estar operando a 1 MHz, la función que genera el pulso de habilitacón es:

```
void LCD_pulso_E( ) {
   P_LCD = P_LCD | 0x10;     // Nivel alto en la terminal E
   P_LCD = P_LCD & 0xEF;     // Nivel bajo en la terminal E
}
```

Por el tipo de interfaz, es necesario contar con una función que escriba una instrucción de 4 bits, la cual va a recibir la información en 8 bits para descartar los 4 bits más significativos, el código de esta función es:

```
void LCD_write_inst4(uint8_t  inst) {
   P_LCD = inst & 0x0F;          // Rs =  0
   LCD_pulso_E();
}
```

Con el apoyo de la función `LCD_write_inst4()` se puede desarrollar una función que escriba una instrucción o comando de 8 bits, invocándola 2 veces con la información adecuada, esta función es:

```
void LCD_write_inst8(uint8_t  inst) {
uint8_t   temp;

   temp = (inst & 0xF0) >> 4;
   LCD_write_inst4(temp);        // Primero el nibble alto
   temp = inst & 0x0F;
   LCD_write_inst4(temp);        // Después el nibble bajo

   _delay_us(40);
}
```

La función `LCD_write_inst8()` incluye un retardo de 40 us porque es el tiempo requerido por la mayoría de comandos, sin embargo, el comando *Limpieza del display* necesita un tiempo mayor, por lo que debe tener una función especial, esta es:

```
void LCD_clear() {
   LCD_write_inst8(0X01);
   _delay_ms(2);
}
```

El tiempo de espera es importante porque las funciones no están haciendo la lectura de la bandera de ocupado.

En cuando a la escritura de datos, todos los datos son de 8 bits, primero se envía la parte alta y luego la parte baja, como se muestra en la función:

```
void LCD_write_data(uint8_t  dat_8b) {
uint8_t temp;

   temp = (dat_8b & 0xF0) >> 4;     // PARTE ALTA
   P_LCD = temp | 0x20;             // RS = 1
   LCD_pulso_E();

   temp = dat_8b & 0x0F;            // PARTE BAJA
   P_LCD = temp | 0x20;             // RS = 1
   LCD_pulso_E();

   _delay_us(40);
}
```

La inicialización del LCD debe seguir la secuencia mostrada en la Figura 10.21, personalizando algunos parámetros. En la siguiente función se muestra la secuencia de inicialización, considerando un LCD de dos líneas y mapas de bits de 5x7 puntos, el código es:

```
void LCD_reset() {

  _delay_ms(15);
  LCD_write_inst4(0x03);       // Configura la función del display
                               // La interfaz aún es de 8 bits
```

```
  _delay_ms(4.1);
  LCD_write_inst4(0x03);        // Configura la función del display
                                // La interfaz aún es de 8 bits
  _delay_us(100);
  LCD_write_inst4(0x03);        // Configura la función del display
  _delay_us(40);                // La interfaz aún es de 8 bits

  LCD_write_inst4(0x02);        // Configura la función del display
  _delay_us(40);                // Define la interfaz de 4 bits

  LCD_write_inst8(0x28);        // Configura la función del Display
                                // Dos líneas y 5x7 puntos

  LCD_write_inst8(0x0C);        // Encendido del display

  LCD_clear();                  // Limpieza del display

  LCD_write_inst8(0x06);        // Ajuste de entrada de datos
                                // Autoincremento del contador
                                // y sin desplazamiento
}
```

La biblioteca incluye una función para ubicar el cursor, puesto que se considera un LCD de 16 x 2, la función recibe un byte del cual se toma el nibble alto para definir el renglón (0 o 1) y el nibble bajo para establecer la columna. La función se basa en el comando *Configura dirección en DDRAM*, por lo que debe tomarse en cuenta que el primer renglón inicia en la dirección 0x00 y el segundo en la 0x40.

El código de la función `LCD_cursor()` es:

```
void LCD_cursor(uint8_t  pos) {
uint8_t  col;

   col = pos & 0x0F;
   if((pos & 0xF0) == 0) {
      col = col | 0x80;          // DB7 = 1
      LCD_write_inst8(col);
   }
   else {
      col = col | 0xC0;          // DB7 = 1 y dirección = 0x40
      LCD_write_inst8(col);
   }
}
```

La última función de la biblioteca es para imprimir una cadena en la pantalla, solo se considera el área visible de manera que la lóngitud de la cadena no puede ser mayor que 32. La función limpia al LCD por lo que cualquier información previa va a perderse. El cambio de rengón es automático después de escribir 16 caracteres.

El código de la función `LCD_write_cad()` es:

```
void LCD_write_cad(char cad[], uint8_t tam) {
uint8_t i;

    LCD_clear();
    for(i=0; i<tam; i++) {
        LCD_write_data((uint8_t) cad[i]);

        if(i==15)
            LCD_cursor(0X10);
    }
}
```

En el siguiente ejemplo se ilustra el uso de la biblioteca `LCD.h`, se presenta un contador de eventos con la finalidad de mostrar información que cambia en tiempo de ejecución.

Ejemplo 10.3 - Contador de eventos con salida en un LCD
En la Figura 10.24 se muestra un ATMega328P con dos botones conectados a sus interrupciones externas y un LCD como elemento de salida. Realice un programa para que este sistema funcione como un contador de eventos en un intervalo entre 0 y 999.

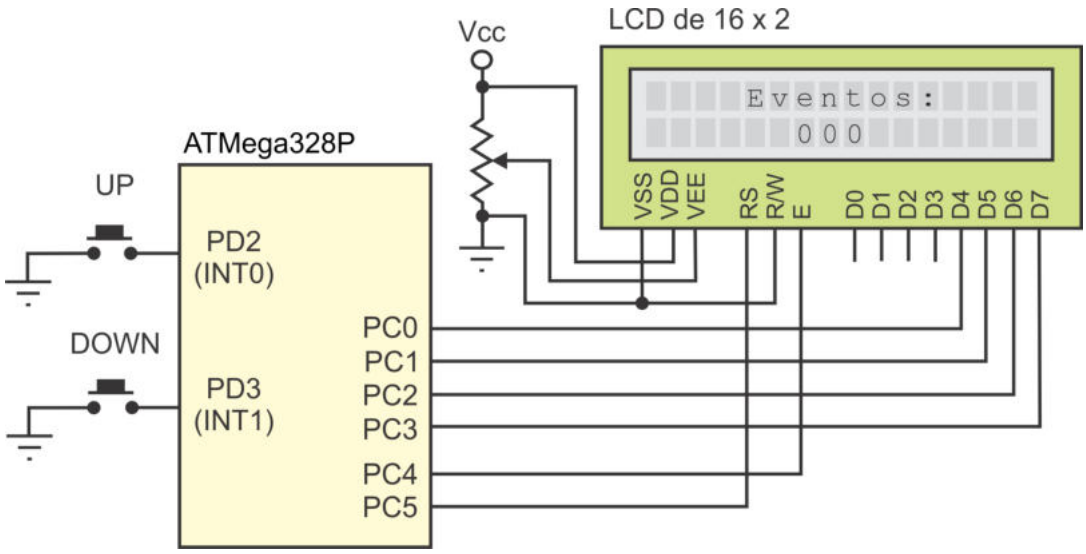


Figura 10.24: Contador de eventos

En la solución de este ejemplo se maneja una variable global de 16 bits para que cubra el intervalo requerido. En las ISR de las interrupciones externas se modifica el contador y se actualiza su valor en el LCD, para esta última tarea se utiliza una función que se encarga de ubicar al cursor y preparar el envío de cada dígito al LCD.

El código con la solución del ejercicio es:

```
#include  <avr/io.h>
#include  <avr/interrupt.h>
#include  "LCD.h"              // Biblioteca para el LCD

uint16_t conta = 0;
void  imprime_LCD();

ISR(INT0_vect){
   conta = (conta < 999)?conta+1:0;
   imprime_LCD();
}

ISR(INT1_vect){
   conta = (conta > 0)?conta-1:999;
   imprime_LCD();
}

int main() {

   DDRC = 0xFF;                // Salida para el LCD
   DDRD = 0x00;                // Entrada para los botones
   PORTD = 0xFF;               // Pull-up

   EICRA = 0x0A;               // Configura interrupciones
   EIMSK = 0x03;               // Habilita interrupciones externas

   LCD_reset();                // Inicializa al LCD
   LCD_write_cad("____Eventos:__________000", 25);

   sei();
   while (1) {
      asm("NOP");
   }
}

void  imprime_LCD() {
uint8_t  u, d, c;

   u = conta % 10;
   d = (conta/10)%10;
   c = conta/100;

   LCD_cursor(0x16);
   LCD_write_data(c + 0x30);
   LCD_write_data(d + 0x30);
   LCD_write_data(u + 0x30);
}
```

La función `imprime_LCD` solo escribe el número en la pantalla, es decir, no se reescribe la parte que no cambia, por eso la importancia de ubicar al cursor en la posición

correcta. A cada dígito se le suma 0x30 para obtener su código ASCII, que es lo que espera el controlador del LCD.

Otra alternativa para escribir números en el LCD consiste en emplear la función `sprintf` de la biblioteca `stdio.h` del estándar de C. Primero, mediante la función `sprintf` se copia la cadena en un arreglo de caracteres y, posteriormente, el arreglo se evía al LCD con la función `LCD_write_cad` de la biblioteca `LCD.h`, se debe considerar que el LCD será borrado, por lo que hay que escribir en mensaje completo.

10.5. Manejo de Motores

Para construir robots u otros sistemas mecatrónicos, es necesario el manejo de actuadores que le otorguen movimiento al sistema. Dependiendo de la aplicación, el desarrollador puede optar por el uso de motores de corriente directa, motores paso a paso o servo motores. En esta sección se describen algunas ideas básicas para manipular estos dispositivos desde un microcontrolador ATMega328P.

10.5.1. Motores de Corriente Directa

Un motor de corriente directa (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 10.25 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.

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, el cual también se muestra en la Figura 10.25, con este dispositivo, la ganancia en corriente es el producto de las ganancias de los transistores individuales.

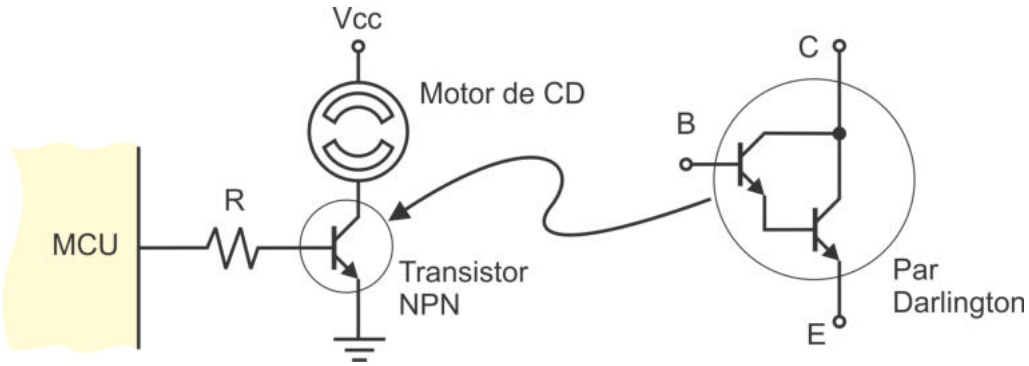


Figura 10.25: Conexión de un motor a una terminal de un MCU

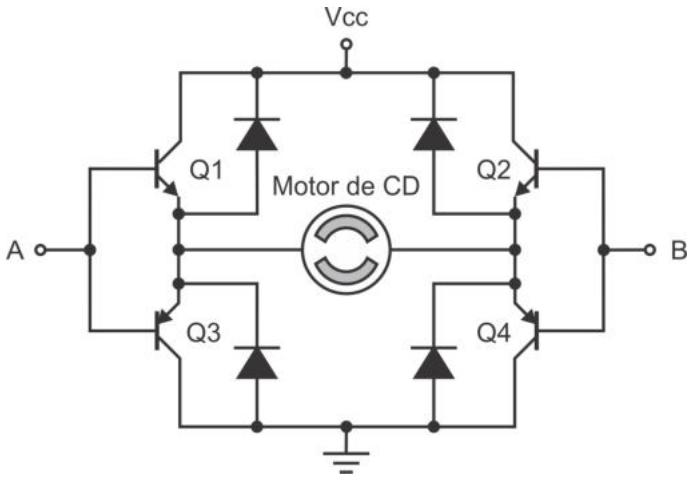


Figura 10.26: Organización de un puente H

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.

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 10.26 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 ATMega328P, o bien, puede colocarse un resistor de 1 kΩ o menor, para limitar la corriente. Los diodos en el puente evitan que el motor almacene energía, pueden omitirse en motores pequeños.

Si en ambas terminales de control se colocan 0's, intentarán 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 intentarán 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; el movimiento es en dirección contraria si A = 0 y B = 1, siendo Q2 y Q3 los transistores activos.

Para que adicionalmente se manipule la velocidad, en lugar de un encendido constante en las terminales A y B, se puede enviar una señal PWM en una de ellas mientras la otra se mantiene con un nivel lógico bajo, de acuerdo con la dirección deseada para el movimiento. Los 3 temporizadores de un ATMega328P son capaces de generar 2 señales PWM, de manera que sin problema se podrá controlar la velocidad y dirección en un motor de CD.

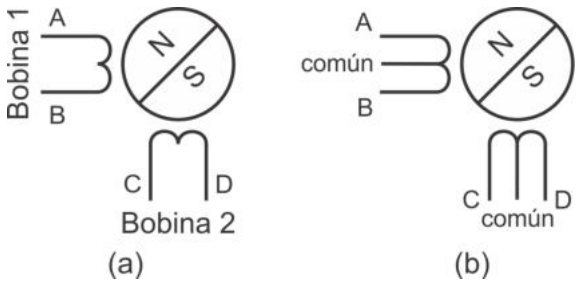


Figura 10.27: Motor paso a paso (a) bipolar y (b) unipolar

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 suministrar hasta 3 A de corriente directa.

10.5.2. 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 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 solo de 1.8° o hasta de 90°. Para completar una vuelta, se requiere de 200 pasos de 1.8° y únicamente 4 de 90°. Esta precisión en el movimiento por paso hace a los MPAP ideales para sistemas en 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 10.27 se esquematizan ambos tipos de motores, en el MPAP bipolar se observan 4 cables y en el unipolar 6, aunque en algunos modelos se tiene una conexión interna entre los puntos comunes por lo que de manera externa solo se aprecian 5 cables.

Con respecto a la Figura 10.27(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 (Figura 10.27(b)), 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 se puede conectar a tierra. Al hacer circular la corriente en una o dos bobinas continuas se van a inducir polos magnéticos en el estator, y con ello, el rotor va a moverse hasta quedar alineado.

Polarización y Operación de un Motor Bipolar

Cada bobina de un MPAP bipolar se debe manejar con un puente H, para permitir la polarización en ambos sentidos, este puede construirse con elementos discretos, como se mostró en la Figura 10.26, 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 10.28 se muestra la conexión de un MPAP bipolar con un MCU, puede ser el ATMega328P, utilizando un L293B, en donde se han configurado sus *drivers* como 2 puentes H.

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 soportar hasta 36 V. Los diodos evitan que el motor almacene energía y pueden omitirse con motores pequeños.

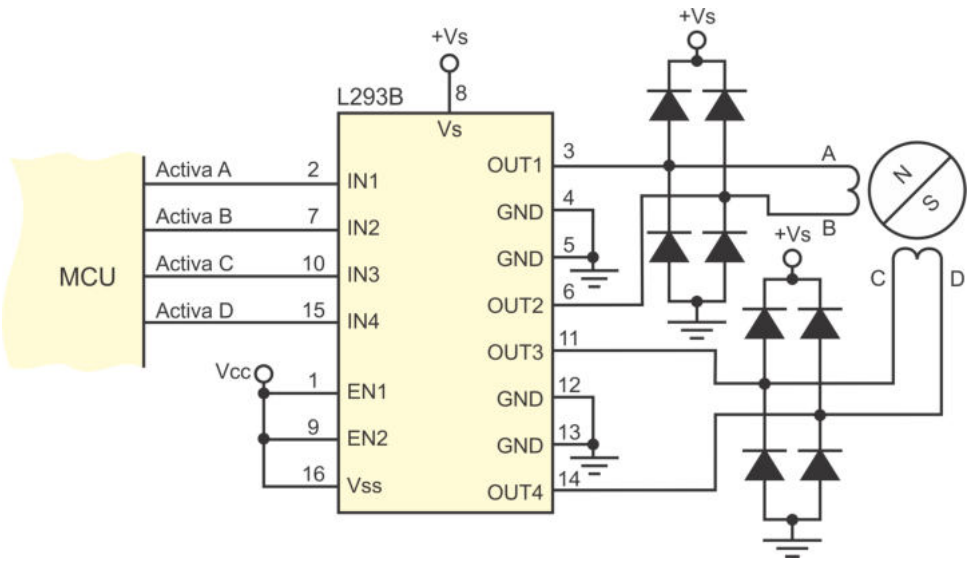


Figura 10.28: Manejo de un MPAP bipolar a través de un L293B

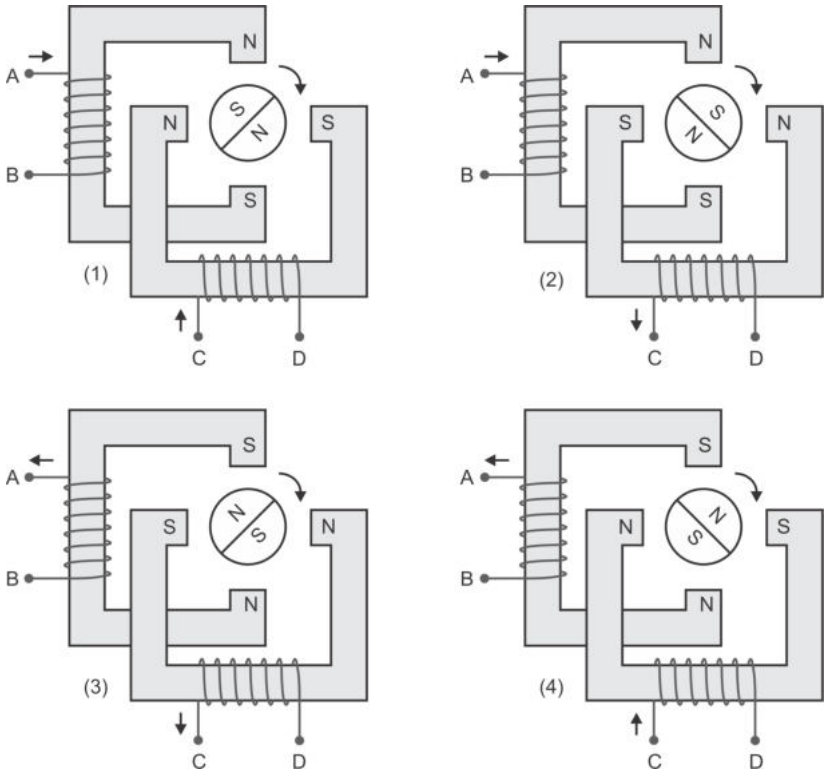


Figura 10.29: Operación de un motor paso a paso bipolar

En la Figura 10.29 se exhibe el movimiento del rotor de un motor bipolar al cambiar la polaridad de las bobinas, modificando 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.

En la Tabla 10.9 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 10.29, 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. En la tabla también se muestran las salidas que debe generar el MCU.

Tabla 10.9: Secuencia para controlar motores paso a paso bipolares

	Terminales (Bobinas)				**Salidas (MCU)**			
Paso	**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 van a inducir más polos magnéticos. El rotor también tiene un mayor número de polos magnéticos permanentes para que coincida con el estator. Con esta organización se pueden conseguir pasos más pequeños pero no cambia la secuencia en las salidas del MCU para realizar el movimiento.

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 10.30, donde el MCU controla el disparo de los transistores y con ello energiza una 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.

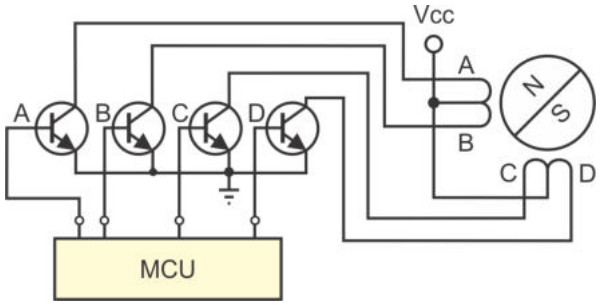


Figura 10.30: Polarización de un MPAP unipolar

La operación de los motores unipolares se puede realizar con 3 secuencias diferentes, en la Tabla 10.10 se muestra la primera secuencia en la que se polariza solo una de las bobinas y el rotor se alinea con el campo magnético impuesto.

Tabla 10.10: Secuencia para controlar motores unipolares, activando 1 bobina

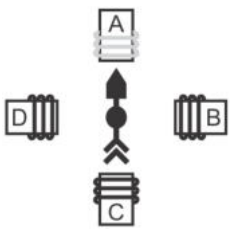
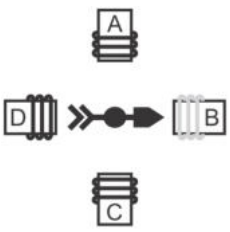
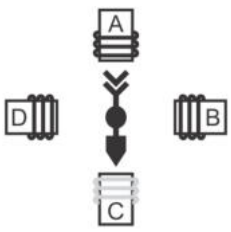
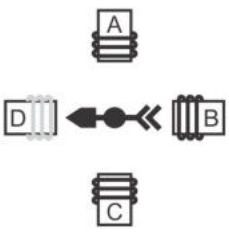
Paso 1				**Paso 2**				**Paso 3**				**Paso 4**			
A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1
															

Tabla 10.11: Secuencia para controlar motores unipolares, activando 2 bobinas

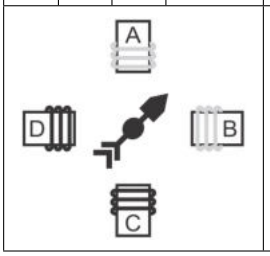
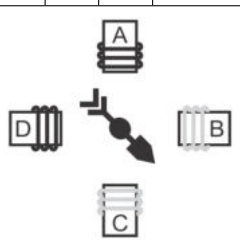
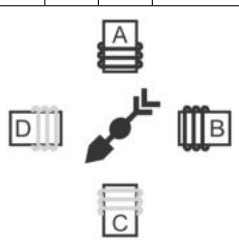
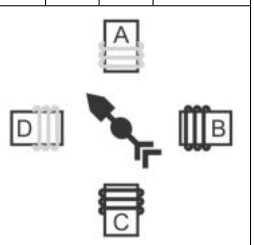
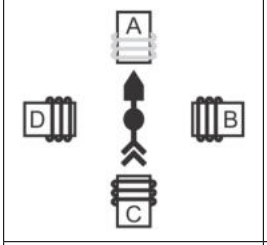
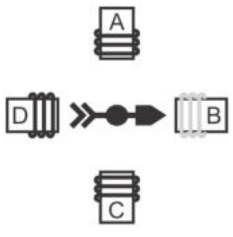
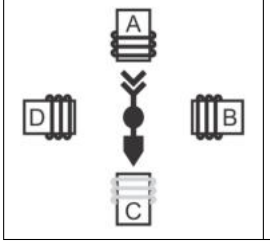
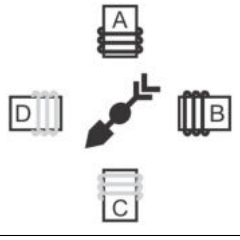
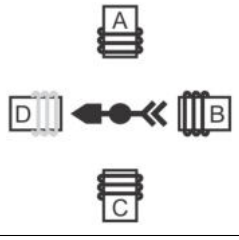
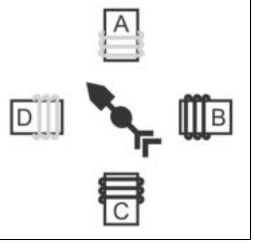
Paso 1				**Paso 2**				**Paso 3**				**Paso 4**			
A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
1	1	0	0	0	1	1	0	0	0	1	1	1	0	0	1
															

Tabla 10.12: Secuencia para manejar medios pasos en un MPAP unipolar

Paso 1				**Paso 2**				**Paso 3**				**Paso 4**			
A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
1	0	0	0	1	1	0	0	0	1	0	0	0	1	1	0
															
Paso 5				**Paso 6**				**Paso 7**				**Paso 8**			
A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
0	0	1	0	0	0	1	1	0	0	0	1	1	0	0	1
															

En la Tabla 10.11 se muestra la segunda secuencia en donde se polarizan dos bobinas en forma simultánea, de manera que el rotor se alinea al centro del campo magnético impuesto por ambas bobinas, esta secuencia ofrece un mayor torque al aumentar la intensidad del campo magnético inducido.

La tercera secuencia es una combinación de las 2 primeras, es decir, se polariza una bobina y posteriormente dos, consiguiendo avances de medios pasos, esta secuencia se muestra en la Tabla 10.12. Sin importar la secuencia elegida, no debe alterarse su orden, por ejemplo, si un motor se mueve 5 medios pasos siguiendo la secuencia

de la Tabla 10.12, 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 10.4 se ilustra cómo conservar una secuencia.

Otro aspecto importante es el tiempo requerido para el establecimiento del rotor, esto es, una vez que se han polarizado las bobinas es necesario esperar un periodo de tiempo mínimo para que los polos magnéticos del rotor se alineen con los polos inducidos en el estator. El fabricante debe proporcionar la velocidad máxima de movimiento, conociendo este parámetro y el número de pasos, se puede calcular el tiempo mínimo requerido, otra opción consiste en estimar este tiempo en forma práctica.

En un motor unipolar real ocurre lo mismo que en un motor bipolar, su estructura interna es compleja para conseguir pasos pequeños aunque el principio de operación es el mismo, por lo que las secuencias ilustradas en las Tablas 10.10, 10.11 y 10.12, siguen siendo válidas.

Ejemplo 10.4 - Manejo de un MPAP unipolar
Suponga que un MPAP unipolar está conectado en el nibble menos significativo del Puerto B de un ATMega328P y realice una función en lenguaje C que lo mueva en medios pasos (ver Tabla 10.12). La función recibirá como argumentos el número de medios pasos y el sentido del movimiento (0 en dirección de las manecillas del reloj y 1 en dirección contraria).

Como datos globales se requiere la secuencia de activación de los transistores y una variable que conserve el último paso realizado. La secuencia se ubica en memoria de código por ser constante, el código para ello es:

```
// Constantes para activar los transistores
const uint8_t sec_trans[] PROGMEM ={0x01,0x03,0x02,0x06,
                                    0x04,0x0C,0x08,0x09};

uint8_t  num_paso;      // Variable para no perder la secuencia
```

En el programa principal debe haber un par de asignaciones, para establecer la condición inicial de la salida:

```
  num_paso = 0;      // Inicia con el salida 0 de la secuencia
  PORTB = 0x01;      // Primera salida en el puerto B
```

Bajo estas condiciones, la función para mover a un MPAP unipolar es:

```
void motor_pasos(uint8_t pasos, uint8_t direccion ) {
uint8_t  i;
```

```
    if( direccion == 0 ) {
        for( i = 0; i < pasos; i++) {
            num_paso = (num_paso < 7) ? num_paso + 1: 0;
            PORTB = pgm_read_byte(&sec_trans[num_paso]);
            _delay_ms(10);          // Tiempo de establecimiento
        }
    }
    else {
        for( i = 0; i < pasos; i++) {
            num_paso = (num_paso > 0) ? num_paso - 1: 7;
            PORTB = pgm_read_byte(&sec_trans[num_paso]);
            _delay_ms(10);          // Tiempo de establecimiento
        }
    }
}
```

10.5.3. 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 en realidad un servomotor es un sistema de control de posición comandado por una señal modulada en ancho de pulso (PWM). El rango de movimiento de un servomotor está limitado a 180° pero es suficiente para dotar de movimiento a un robot.

Un servomotor incluye un convertidor de voltaje, un amplificador (o *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 10.31 se muestra la organización típica de un servomotor, en donde se aprecian los diferentes elementos.

La frecuencia y el ciclo de trabajo para el manejo de un servomotor puede variar entre fabricantes, por ejemplo, los servomotores fabricados por Futaba y Hitech se manejan con una señal PWM de 50 Hz (periodo de 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.

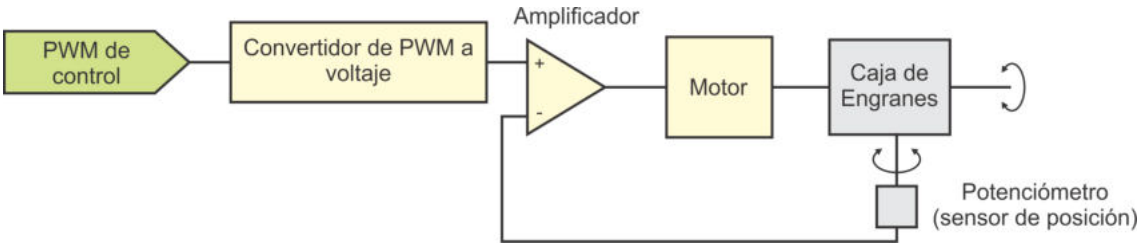


Figura 10.31: Organización de un servomotor

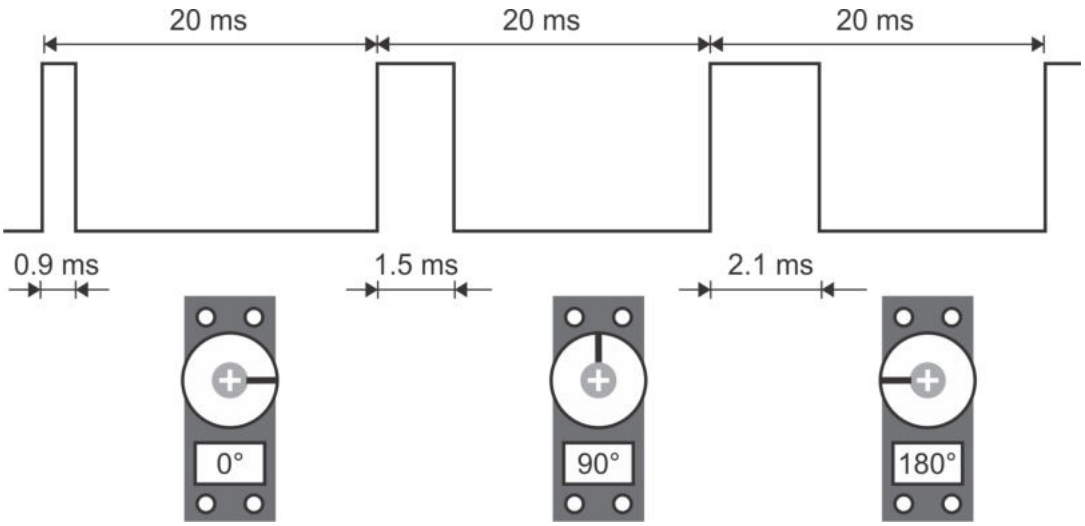


Figura 10.32: Operación de un servomotor

En la Figura 10.32 se ilustran las 3 posiciones principales de un servomotor, su comportamiento es lineal para las posiciones intermedias. Con el ATMega328P 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 5.4. Con los temporizadores 0 y 2 se debe ajustar el factor de preescala para aproximar el periodo y ciclo de trabajo requeridos por un servomotor.

10.6. Interfaz con Sensores

Dado que los microcontroladores ATMega328P internamente contienen un convertidor analógico a digital (ADC) y un comparador analógico (AC), cuando un sistema debe obtener información de algún sensor analógico, básicamente se debe acondicionar la señal que proporciona para que quede en el rango de trabajo del microcontrolador y, de esta forma, la información resultante pueda ser recibida por el ADC o por el AC.

Aunque el acondicionamiento de señal se puede realizar con amplificadores operacionales cásicos como el LM741 o el TL081, se sugiere el uso de los amplificadores LM358 o LM324, porque pueden operar con una fuente de alimentación única, desde 3 V hasta 32 V. Con ello, los elementos de acondicionamiento se pueden alimentar 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 10.33 se muestran la estructura interna de ambos dispositivos.

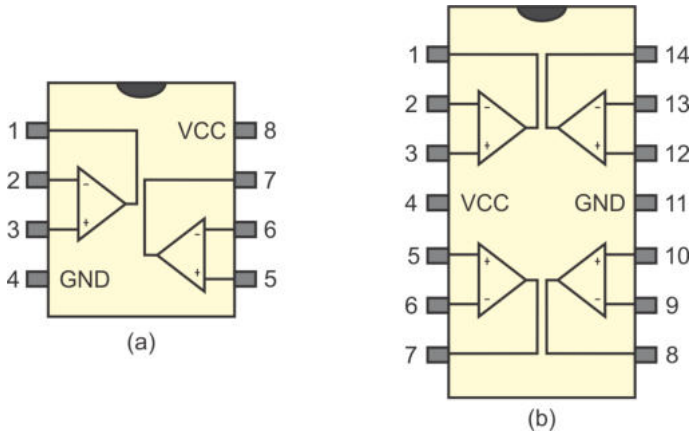


Figura 10.33: Terminales del (a) LM358 y del (b) LM324

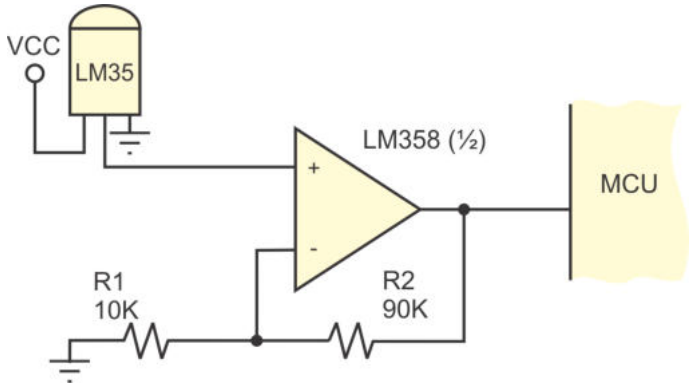


Figura 10.34: Acondicionamiento de un sensor de temperatura

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 10.34 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 se debe procesar la información, si se emplea al ADC con una resolición de 10 bits, el valor de 0x0000 va a corresponder con 0°C y 0x03FF con 50°C.

En la práctica, si el circuito está alimentado con 5 V, no se va a alcanzar ese valor en la salida del amplificador operacional porque se satura antes de que eso ocurra. El fabricante garantiza un comportamiento lineal con un valor máximo para la salida

de Vcc - 1.5 V, por ello, para alcanzar un rango de operación más amplio, debe emplearse un voltaje de alimentación mayor o reducir la ganacia del amplificador no inversor y compensar por software para estimar el valor real de la temperatura.

Un aspecto importante, referente al manejo de sensores desde un microcontrolador, es que actualmente existen muchos módulos comercialmente disponibles, que contienen sensores acondicionados para proporcionar información a través de las interfaces digitales SPI o I2C, estos módulos son muy convenientes porque no se requiere hardware para el acondicionamiento de señal o fuentes de alimentación adicionales para su manejo.

Basta enviar a internet la consulta de "sensores SPI" o "sensores I2C" para obtener información sobre sensores de temperatura, humedad, presión barométrica, sensores de distancia, acelerómetros, giroscopios, etc. Por eso resulta fundamental el manejo de estas interfaces disponibles en un ATMega328P, descritas en los capítulos 7 y 8 del presente libro, para su aplicación en el manejo de sensores.

10.7. Interfaz con una Computadora Personal

Muchos sistemas basados en microcontroladores requieren de una comunicación con una computadora personal (PC, *personal computer*) para concentrar y manipular información que van recolectando a lo largo del tiempo, así como para proporcionar una interfaz desde donde el usuario puede configurar diferentes parámetros del sistema.

Por ejemplo, un sistema embebido para el monitoreo de variables ambientales, como temperatura y humedad, puede estar colectando datos periódicamente, posteriormente, el usuario debe conectar el sistema embebido con una PC para hacer la descarga de información. Desde la misma aplicación en la PC, el usuario puede definir el periodo de muestro o ajustar algún otro parámetro como la fecha y hora en el sistema embebido.

Otro ejemplo puede ser un reloj checador electrónico, el sistema debe operar en forma autónoma soportado por un MCU con memoria externa, no obstante, pasados algunos días va a requerir la conexión con una PC para descargar la información almacenada y realizar los informes correspondientes.

Para establecer esta comunicación PC-MCU, anteriormente las computadoras contaban con 3 puertos diferentes: serie, paralelo y USB. Sin embargo, en los últimos años el uso de los puertos serie y paralelo se ha perdido de manera que solo el puerto USB se mantiene como mecanismo para que una computadora pueda interactuar con su entorno.

El ATMega328P no cuenta con un módulo para el manejo de la interfaz USB, la comunicación con una computadora solo se puede realizar por medio de la USART.

Se requiere de hardware externo para la conversión de las señales TTL de la USART del MCU a las señales de la interfaz USB que maneja la PC y para ello se tienen diferentes alternativas, que se describen en los siguientes apartados.

10.7.1. Cable Adaptador de USB a RS-232

La empresa Steren® comercializa un cable adaptador de USB a RS-232 que se puede conectar 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 disponible para la conexión de un MCU u otro dispositivo compatible. Comercialmente se pueden encontrar cables adaptadores similares de otros fabricantes con mejores precios, el problema en cualquier caso es que los niveles de voltaje de una interfaz RS-232 no son totalmente compatibles con los niveles 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. En la Figura 10.35 se comparan los niveles de voltaje RS-232 con los niveles de voltaje TTL.

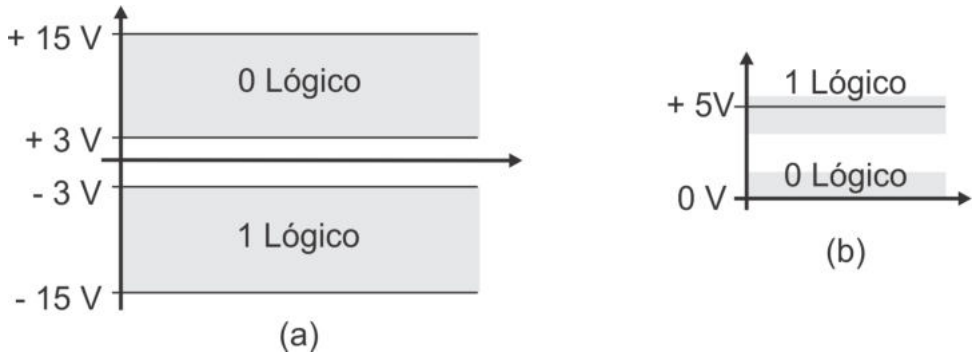


Figura 10.35: Comparación de (a) niveles de voltaje RS-232 con (b) TTL

Para el acoplamiento de niveles de voltaje puede utilizarse un circuito integrado 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 10.36 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 PDIP.

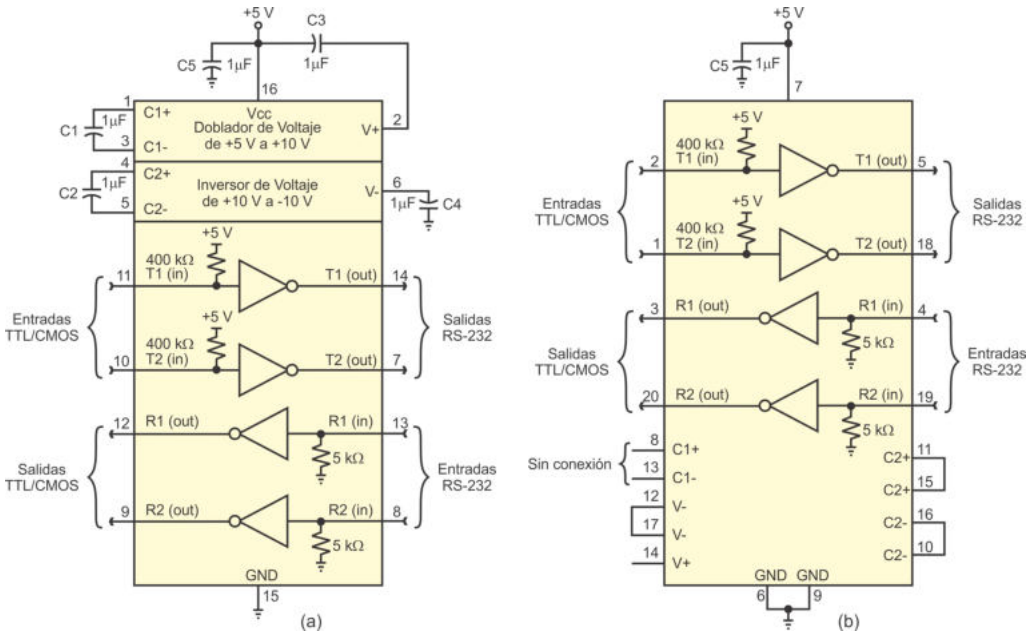


Figura 10.36: Circuitos para la conversión TTL/CMOS a RS-232: (a) MAX232 y (b) MAX233

10.7.2. Circuitos Integrados Controladores

Otra alternativa para establecer la comunicación de un MCU con una PC, vía USB, consiste en emplear un circuito integrado controlador. Un CI que se ha mantenido funcional por muchos años es el FT232BM[1], manufacturado por la empresa FTDI (*Future Technology Devices International Ltd.*), 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 en las transferencias de datos.
- Interfaz serial que soporta datos de 7 u 8 bits con 1 o 2 bits de paro, el bit de paridad también es configurable: par, impar o sin paridad.
- 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.

[1]Para más información visite la página del fabricante: http://www.ftdichip.com/

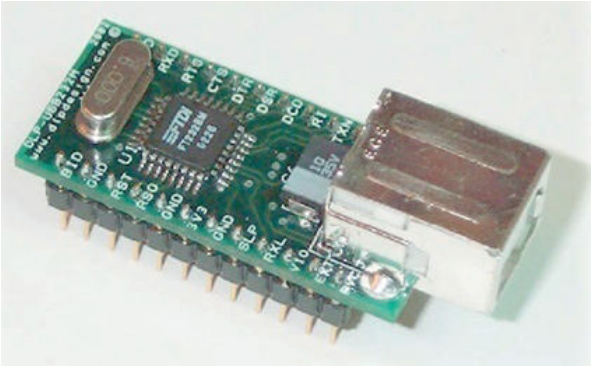


Figura 10.37: Módulo DLP-USB232M-G para una interfaz serial a USB

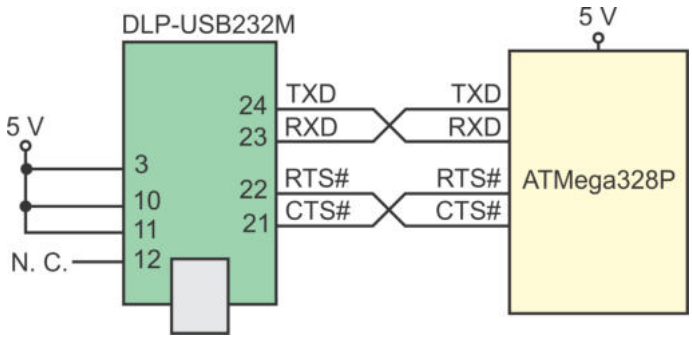


Figura 10.38: Conexión de un MCU con el módulo DLP-USB232M-G

Por lo tanto, en la tarjeta de un sistema basado en un MCU 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.

Con base en el FT232BM, la empresa DLP Design manufactura y comercializa al módulo DLP-USB232M-G. Con este módulo se puede conectar un sistema basado en un ATMega328P con una PC vía USB. En la Figura 10.37 se puede ver al módulo con una disposición tipo PDIP para facilitar el desarrollo de prototipos.

En la Figura 10.38 se muestra cómo conectar al módulo DLP con un ATMega328P, se requiere solo de una fuente de alimentación de 5 volts, en el MCU se utiliza a la USART como el medio de comunicación, las líneas de conexión adicionales pueden ser empleadas para sincronización.

La empresa FTDI también comercializa al circuito integrado FT245BM, el cual, a diferencia del FT232BM, proporciona una conversión de paralelo a USB. Empleando

al circuito FT245BM, la empresa DLP Design manufactura y comercializa al módulo DLP-USB245M-G. El módulo también presenta una disposición tipo PDIP, como se puede ver en la Figura 10.39. En la Figura 10.40 se muestra un esquemático en el que se ilustra la forma de conectarlo con un microcontrolador, por utilizar una interfaz paralela, se requiere más de un puerto del ATMega328P, lo cual no es conveniente por el reducido número de puertos disponibles en el ATMega328P.

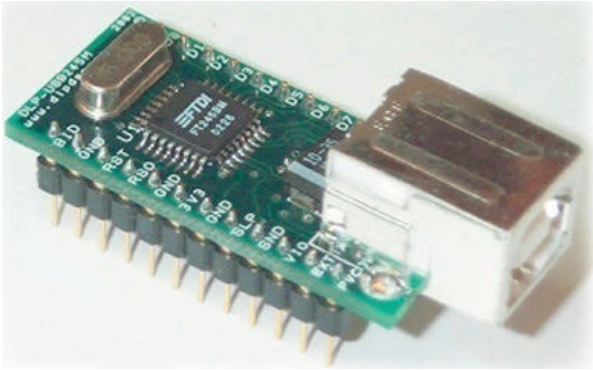


Figura 10.39: Módulo DLP-USB245M-G para una interfaz paralela a USB

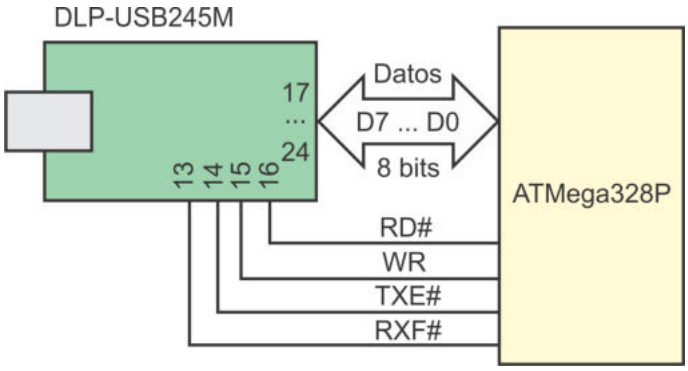


Figura 10.40: Conexión de un MCU con el módulo DLP-USB245M-G

10.7.3. Adaptadores de USB a TTL

Debido al creciente uso del puerto USB, se han diseñado otros circuitos integrados que realizan la conversión de USB a TTL, dos de ellos son: el CH340G manufacturado por la empresa china WCH[2] y el PL-2303HX de la empresa taiwanesa Prolific Technology Inc[3]. Con estos circuitos se han desarrollado módulos adaptadores USB a TTL, los cuales se muestran en la Figura 10.41. La globalización permite la adquisición de estos módulos desde sus paises de origen, a una fracción del costo de un módulo DLP.

[2]http://www.wch.cn/
[3]http://www.prolific.com.tw/

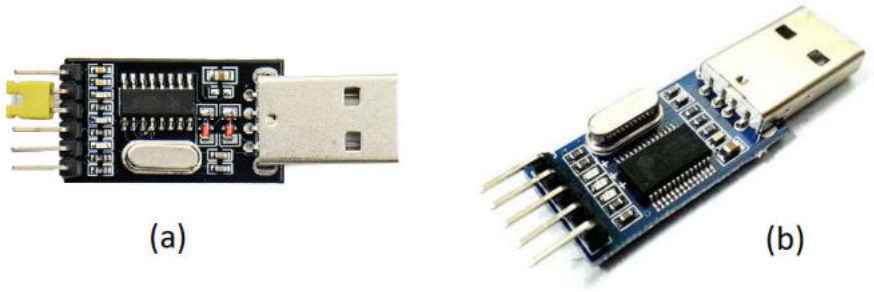


Figura 10.41: Adaptadores USB-TTL: (a) CH340G y (b) PL-2303HX

Funcionalmente, los módulos CH340G y PL-2303HX son muy similares, con el controlador adecuado ambos proporcionan una interfaz serial mediante la cual se puede comunicar una computadora personal con un microcontrolador. La única diferencia significativa es que el PL-2303HX maneja niveles de voltaje de 3.3 V para las salidas Tx y Rx, mientras que en el CH340G se puede seleccionar el nivel con un conector externo, entre 3.3 y 5 V.

10.7.4. AVR con Controlador USB Integrado

Una ventaja en la familia de microcontroladores AVR es que incluye muchos miembros con una variedad en los recursos incluidos. Por ello, se puede buscar el más adecuado para la aplicación que se pretende realizar.

Con respecto a la comunicación de un MCU con una computadora vía USB, dentro de la familia AVR se encuentran los dispositivos ATMega8U2, ATMega16U2 y ATMega32U2, los cuales tienen 8, 16 y 32 kbyte de memoria Flash, respectivamente. Estos dispositivos cuentan con un módulo controlador completamente compatible con USB 2.0, además de otros recursos similares a los del ATMega328P.

El controlador requiere una señal de reloj de 48 MHz $\pm 0.25\,\%$ 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 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 3.6 V. Dado que los microcontroladores AVR pueden alimentarse hasta con 5.5 V, internamente incluyen un

regulador para proporcionar el voltaje requerido por la interfaz. En la Figura 10.42 se muestra la incorporación del controlador USB en el MCU, el regulador de voltaje y su vinculación con el núcleo AVR.

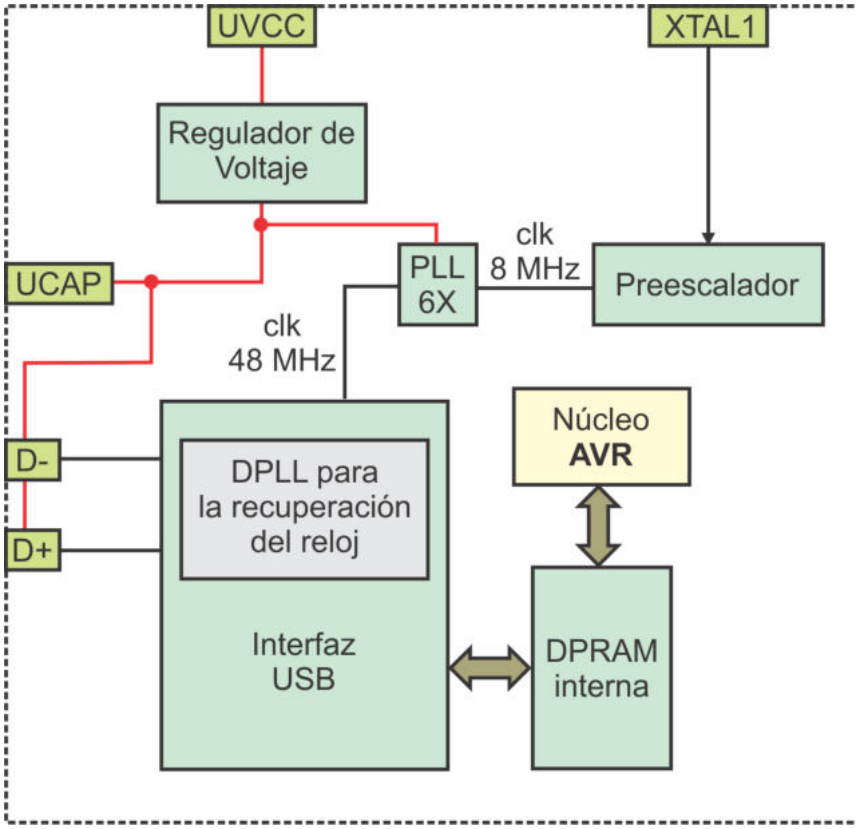


Figura 10.42: Controlador USB integrado en un microcontrolador AVR

El controlador USB solo puede operar como periférico dentro de un bus e incluye una memoria de doble puerto (DPRAM) de 176 bytes, independiente al núcleo AVR, en donde se ubican los elementos lógicos denominados *endpoints*, que son la base para la comunicación con el anfitrión. Cada *endpoint* tiene asociado un conjunto de características que van a regir el intercambio de información, entre las cuales se destacan: ancho de banda, frecuencia de acceso al bus, tipo de transferencia, orientación en la que se transmiten los datos, entre otras.

Una desventaja de los microcontroladores ATMega8U2, ATMega16U2 y ATMega32U2 es que solo se encuentran disponibles en encapsulados QFP o QFN, por lo que resulta complejo desarrollar prototipos con estos dispositivos. Sin embargo, las nuevas versiones de Arduino están utilizando un ATMega16U2 como el medio para programar al ATMega328P (en una Arduino UNO) o a un ATMega2560 (en una Arduino Mega) desde una PC, por medio de la interfaz USB.

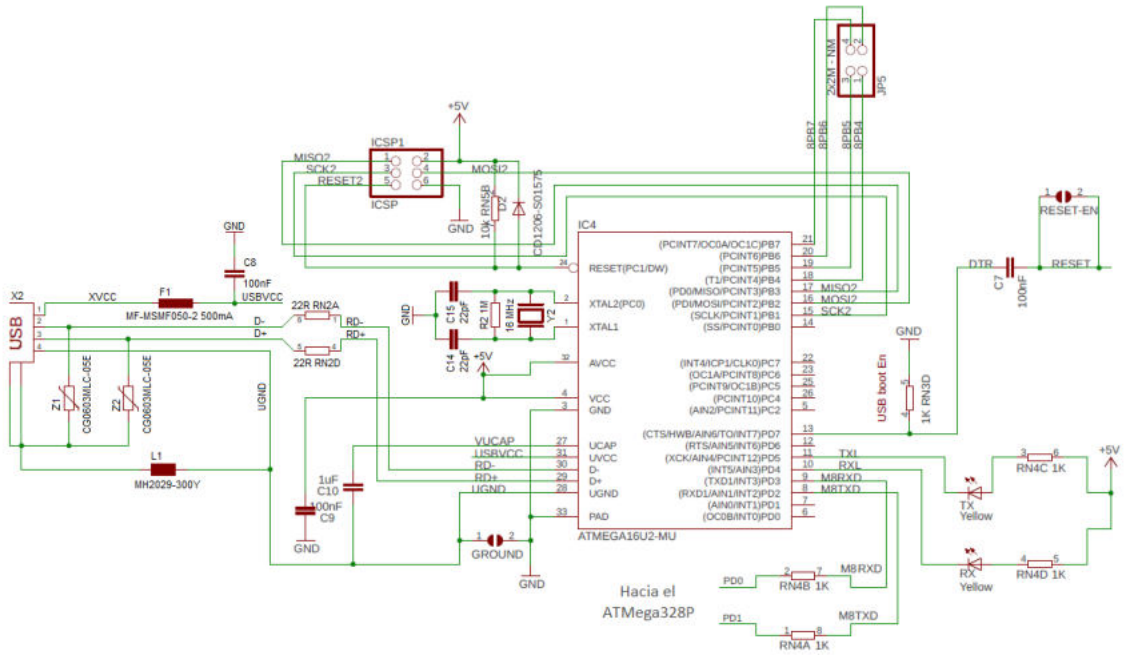


Figura 10.43: Interfaz USB a TTL con base en un ATMega16U2

En la Figura 10.43 se muestra la parte de la tarjeta que corresponde con el ATMega16U2[4], ilustrando todos los elementos de acondicionamiento, quedando disponibles las dos terminales para conectarse con un ATMega328P, en caso de tratarse de una Arduino UNO.

El circuito ilustrado en la Figura 10.43 básicamente es un adaptador de USB a TTL, se puede emplear como una pasarela y conectar un ATMega328P externo a la tarjeta Arduino. El microcontrolador propio de la tarjeta Arduino debe estar sin programar o debe contener un *sketch* vacío para que no interfiera en este enlace. Como un ejemplo, para una Arduino UNO, se debe concectar su terminal 0 con RXD (PD0) del ATMega328P externo y su terminal 1 con TXD (PD1), no se hace el cruce entre transmisor y receptor porque la comunicación es por medio del ATMega16U2.

10.8. 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, estas pueden utilizarse nuevamente en aplicaciones reales.

1. Conecte un teclado matricial de 4x4 a un ATMega328P y muestre el valor de la tecla presionada en un display de 7 segmentos. En la Figura 10.44 se muestra el hardware requerido.

[4]Imagen obtenida del sitio https://www.arduino.cc/

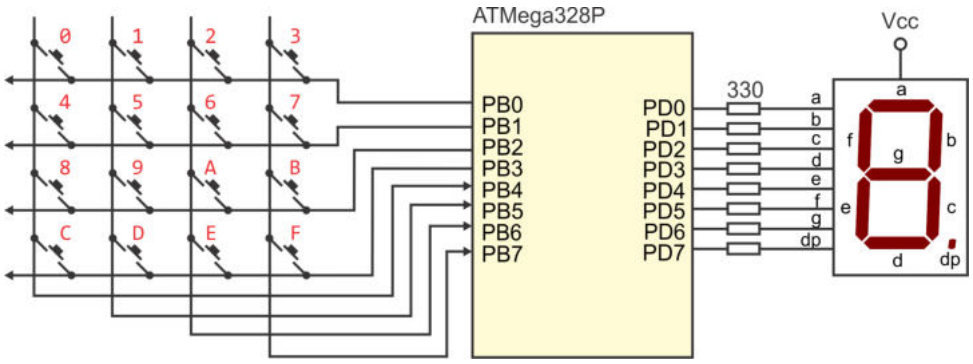


Figura 10.44: Circuito para evaluar un teclado matricial

2. Conecte una matriz de LEDs y un teclado matricial a un ATMega328P (uno en cada puerto), como se muestra en la Figura 10.45. Desarrolle un programa mediante el que se conmute el estado del LED ubicado en la posición que corresponda a una tecla presionada, si el LED está apagado debe encenderse y si está encendido, debe apagarse. Inicialmente todos los LEDs deben estar apagados. Note que el manejo de la matriz de LEDs requiere de un refresco continuo por renglones, habilitando el rengón a encender con un 0 lógico. Dedique una mayor cantidad de tiempo al refresco de la matriz que al sondeo del teclado, con esto también se evitarán múltiples consideraciones para una tecla presionada.

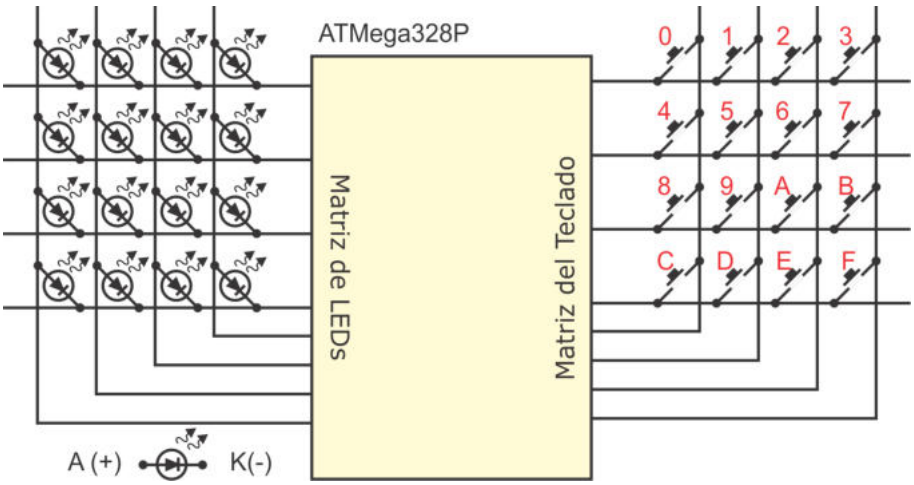


Figura 10.45: Matriz de LEDs manejada con un teclado matricial

3. Conecte 3 displays de 7 segmentos en un bus común e implemente un contador de eventos ascendente/descendente. En la Figura 10.46 se muestra el hardware requerido.

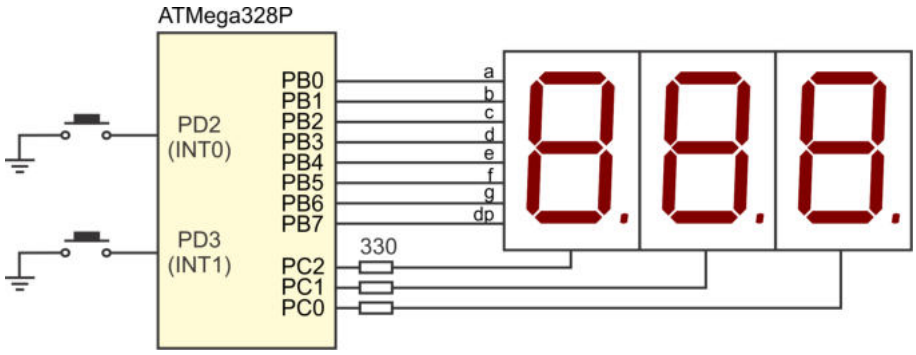


Figura 10.46: Manejo de 3 displays en un bus común

4. Realice un termómetro digital con un rango de 0 a 50°C, con salida en un LCD y actualizando el valor de la temperatura cada medio segundo. En la Figura 10.47 se puede ver que se utiliza al sensor LM35, su salida se amplifica por 10 para que el MCU reciba 0.1 V por grado centígrado. Utilice el temporizador 1 para sincronizar las conversiones y, por lo tanto, para actualizar la salida en el LCD.

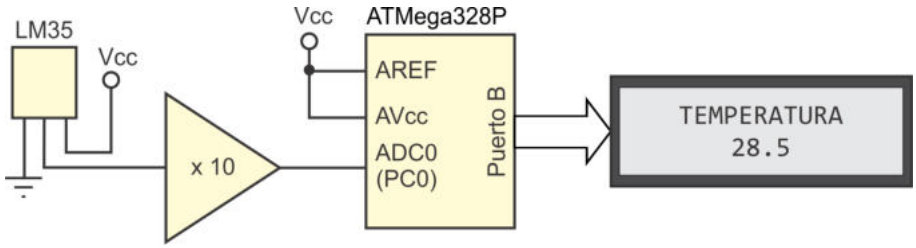


Figura 10.47: Termómetro digital con el sensor LM35

5. Conecte un motor paso a paso unipolar con un ATMega328P y, mediante un par de botones, realice pasos en una u otra dirección, dando un paso cada vez que algún botón es presionado. Si uno de los botones se mantiene presionado, modifique la secuencia cada 300 ms.

6. Implemente el circuito mostrado en la Figura 10.48 para evaluar la comunicación serial y el manejo de un LCD. El circuito podrá recibir un mensaje desde una PC con una longitud máxima de 64 caracteres, el cual debe mostrarse en el LCD. El mensaje es una cadena de caracteres ASCII terminada con el carácter '$'. Si la longitud de la cadena es mayor a 32, debe realizar un desplazamiento continuo para que se pueda leer el mensaje. El botón de Retorno ayudará a evaluar el flujo de información en ambos sentidos, el mensaje se enviará una vez cuando el botón se presiona.

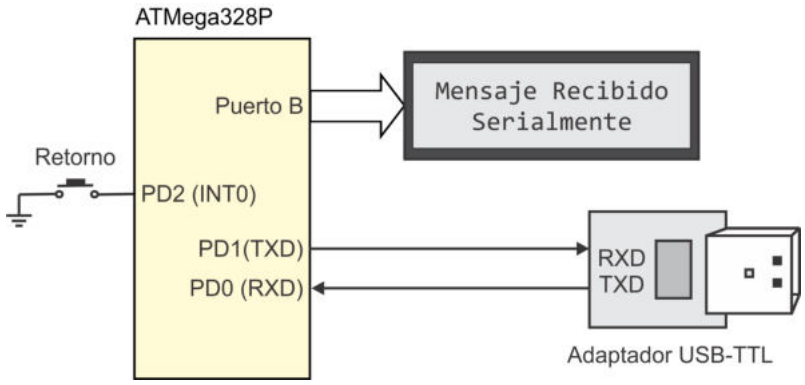


Figura 10.48: Comunicación serial

7. En los ejercicios 4 y 5 del capítulo 8 se solicitó manipular al circuito DS1307, que es un Reloj/Calendario de Tiempo Real (RTC), con interfaz I2C. Al resultado obtenido, agregue un LCD para mostrar la hora y fecha, actualizando la información en la pantalla cada medio segundo.

8. Realice un sistema que genere una señal PWM con una frecuencia de 100 Hz. El ciclo de trabajo será recibido a través de la USART, como una secuencia de caracteres ASCII terminada con cualquier carácter no numérico, y su valor se deberá mostrar en una pantalla LCD. En la Figura 10.49 se muestra la organización del sistema, inicie con un ciclo de trabajo del 50 %.

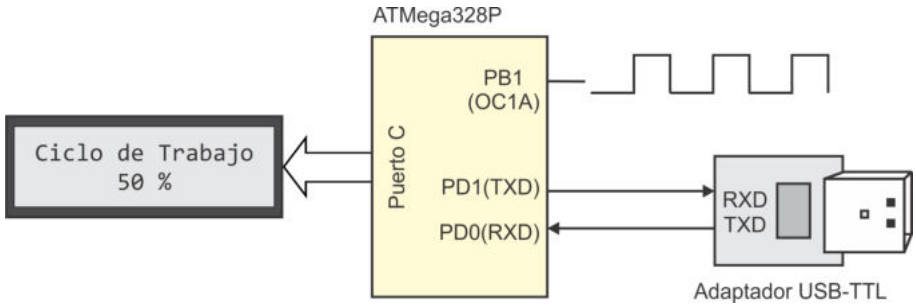


Figura 10.49: PWM con un ciclo de trabajo recibido de manera serial

9. El circuito de la Figura 10.50 digitaliza 4 entradas analógicas para mostrar su valor en un LCD. Desarrolle el programa para el ATMega328P, realizando un monitoreo continuo de las entradas, pero solo actualizando el valor de aquella en donde se haya encontrado un cambio con respecto a la revisión anterior, para evitar parpadeo en el LCD. Considerando que el LCD tiene 16 caracteres visibles por línea, debe emplear una resolución de 8 bits en el ADC para poder exhibir la información completa.

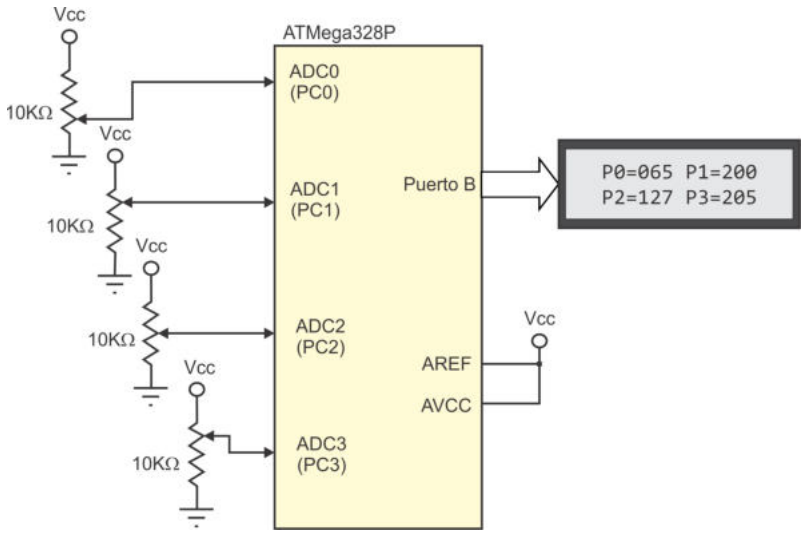


Figura 10.50: Digitalización de 4 señales analógicas.

10. El circuito integrado TMP100 es un sensor de temperatura con interfaz digital I2C. El circuito tiene una resolución de $0.0625°C$ y su rango de temperatura es de $-55°C$ a $+125°C$. Empleando este circuito, implemente un termómetro digital con salida en una pantalla LCD, actualizando la información cada medio segundo. Este ejercicio es similar al 4, pero ahora se debe utilizar un sensor con salida digital en lugar de uno con salida analógica.

Capítulo 11

Desarrollo de Sistemas

En este capítulo se muestra una metodología para desarrollar sistemas embebidos basados en microcontroladores y se ilustra su uso. La metodología está enfocada a aplicaciones soportadas por un solo microcontrolador y ha sido empleada en 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 consiste 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 se puede emplear en cada subsistema o etapa del sistema complejo.

En sistemas con una complejidad media, es decir, sistemas que requieren de un solo microcontrolador pero tienen múltiples funciones y utilizan muchos de sus recursos, la metodología de desarrollo se puede complementar con el diseño incremental, una estrategía que trabaja con diferentes versiones. Desde la primera versión se debe plantear la creación de un sistema funcional con características básicas y, para las siguientes, se deberán agregar nuevas características, enfocando hacia el sistema final. Puede haber tantas versiones como sean necesarias, cada una se considera como un incremento en el proceso de diseño, el último incremento debe cumplir con todos los requerimientos del sistema.

El diseño incremental es una estrategia originalmente orientada al desarrollo de software, que se puede enfocar al diseño de sistemas embebidos, pero como estos combinan el hardware con el software, cada incremento debe probarse implementado físicamente, una simulación no es suficiente por las diferencias naturales que hay con el hardware real.

En este capítulo se describen dos ejemplos, en el primero se aplica la metodología para el desarrollo de una chapa electrónica y, en el segundo, la metodología se complementa con el diseño incremental para el desarrollo de un sistema embebido con las

funciones de reloj de tiempo real, calendario, alarma, temporizador y termómetro, con salida en 4 displays de 7 segmentos.

11.1. 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 ATMega328P o si es mejor emplear 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 estas no se pueden comprometer con otras tareas. Por ejemplo, si un sistema basado en un ATMega328P va a producir señales PWM de 16 bits

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 para que utilicen 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 emplearse un 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 sensores analógicos, es necesario observar el buen desempeño de las etapas de acondicionamiento de señal y asegurar que su voltaje de salida no es superior al esperado en las entradadas analógicas.

   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. Resulta muy ilustrativo si se cuenta 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 asegurar que realiza las tareas planeadas.

   El sistema debe exponerse ante 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 un 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 muy útil contar con un programador serial, dado que los dispositivos permiten una programación en el sistema implementado (*in-system*), se pueden realizar 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 sería sometido el sistema final, 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 11.1. 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.

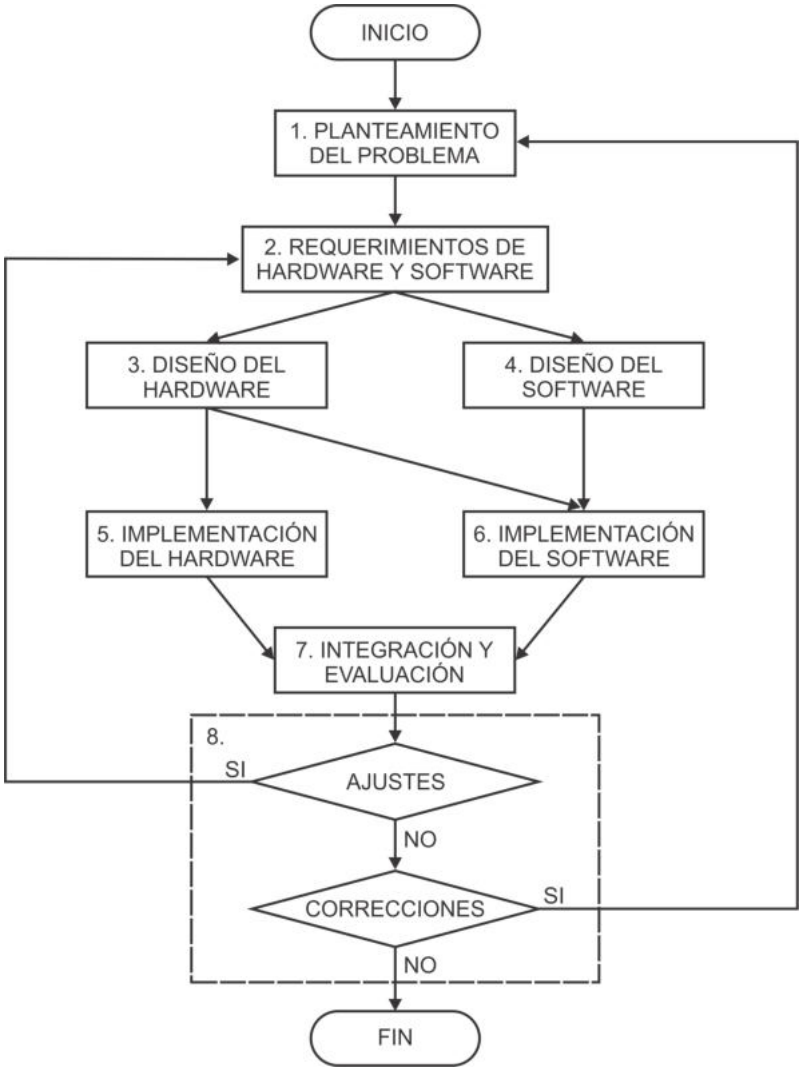


Figura 11.1: Metodología para el diseño de sistemas basados en microcontroladores

En las siguientes secciones se ilustra el uso de la metodología en 2 ejemplos diferentes. En la Figura 11.1 se observa que 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.

11.2. Chapa Electrónica

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.

11.2.1. 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 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 11.2 se muestran las características esperadas en el sistema, ilustrando sus entradas y salidas.

El teclado debe disponer de 10 teclas numéricas y 2 teclas de función: la tecla D (*delete*) para borrar el 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.

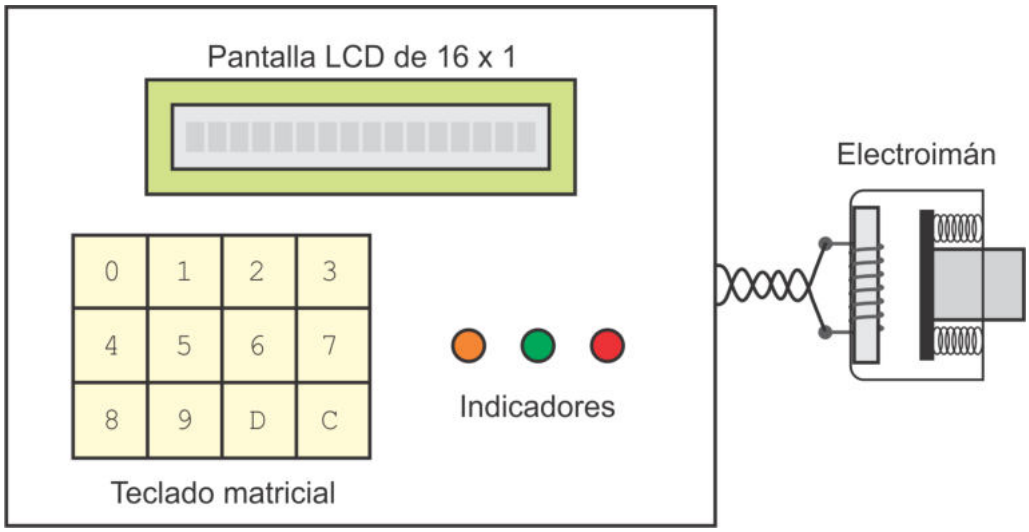


Figura 11.2: Chapa electrónica

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 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 esta es presionada, el sistema pasa al Modo 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 Apertura y el Modo Cambio de Clave, el comportamiento del sistema en cada modo se describe a continuación.

En el Modo Apertura, el sistema debe tener el siguiente comportamiento:

- Con cada tecla numérica presionada en el LCD se debe mostrar el carácter de *, para que el usuario sepa cuántos 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 debe 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 el indicador Verde durante 3 segundos, después, 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 Cambio de Clave.
- Si la clave fue incorrecta, el LCD debe mostrar el mensaje "Clave Incorrecta" y se debe encender el indicador Rojo, también durante un periodo de 3 segundos, después de los cuales el sistema debe regresar al Estado Inicial.

En el Modo 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 vea 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 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 el 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.

A partir del Estado Inicial, el sistema solo puede pasar al Modo Apertura y de este, al Modo Cambio de Clave. De ambos modos va a regresar 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 el último dígito introducido y la tecla C (*change*) para pasar al Modo Cambio de Clave, esta es atendida solo 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 estará 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 van a 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.

11.2.2. Requerimientos de Hardware y Software

Un ATMega328P resulta adecuado para este proyecto, de acuerdo con la Figura 11.2, el MCU a emplear debe disponer de 3 puertos: uno para el teclado, otro para un LCD manejado con una interfaz de 4 bits y un tercero para los LEDs de estado y para el electroimán. Además, un ATMega328P tiene 32 kbytes de memoria Flash, espacio suficiente para cubrir todos los requerimientos funcionales, y 1 kbyte de EEPROM, excediendo por mucho lo requerido, ya que solo se almacenará la clave, ocupando 4 bytes.

Los otros elementos de hardware requeridos son:

- 12 botones configurados como un teclado matricial de 3 x 4.
- 1 LCD de 16 x 1.
- 3 LEDs en colores Naranja, Verde y Rojo, con sus resistores de 330 Ω para limitar la corriente.
- 1 electroimán de 12 V con un transistor BC548 y un resistor de 330 Ω, como elementos de acondicionamiento.

En cuanto al software, las funciones que se han revisado a lo largo del texto y que pueden utilizarse son:

- Biblioteca de funciones para el manejo del LCD.
- Función para el sondeo del teclado, aunque será llamada por interrupción.
- Funciones para la lectura y escritura en la EEPROM.

Los recursos internos que se emplearán son:

- La memoria EEPROM para mantener la clave de acceso, aunque debe copiarse en la RAM para una comparación rápida. La escritura de la clave en EEPROM solo se hará al concluir adecuadamente el Modo Cambio de Clave.
- Las interrupciones por cambios en los puertos, para detectar una tecla presionada, aunque el estado del teclado y el valor de la tecla presionada se deben manejar como variables globales, que serán consultadas en el programa principal.

- El temporizador 1 para el manejo de los intervalos de tiempo, en cada interrupción se revisará si ya se alcanzó el periodo establecido para modificar una bandera que también será consultada en el programa principal.

11.2.3. Diseño del Hardware

El diseño del hardware inicia con una revisión de los recursos internos que se van a emplear, para determinar si hay terminales comprometidas. En este caso, la EEPROM es interna, las interrupciones por cambios en los puertos se aplican en cualquier terminal y el temporizador 1 no tendrá una respuesta automática, por lo tanto, los periféricos externos se pueden acomodar en cualquiera de los puertos. Se opta por ubicar al teclado en el Puerto B, al LCD en el Puerto C y queda disponible el Puerto D para los LEDs de estado y el actuador. En la Figura 11.3 se muestra el hardware resultante para la chapa electrónica.

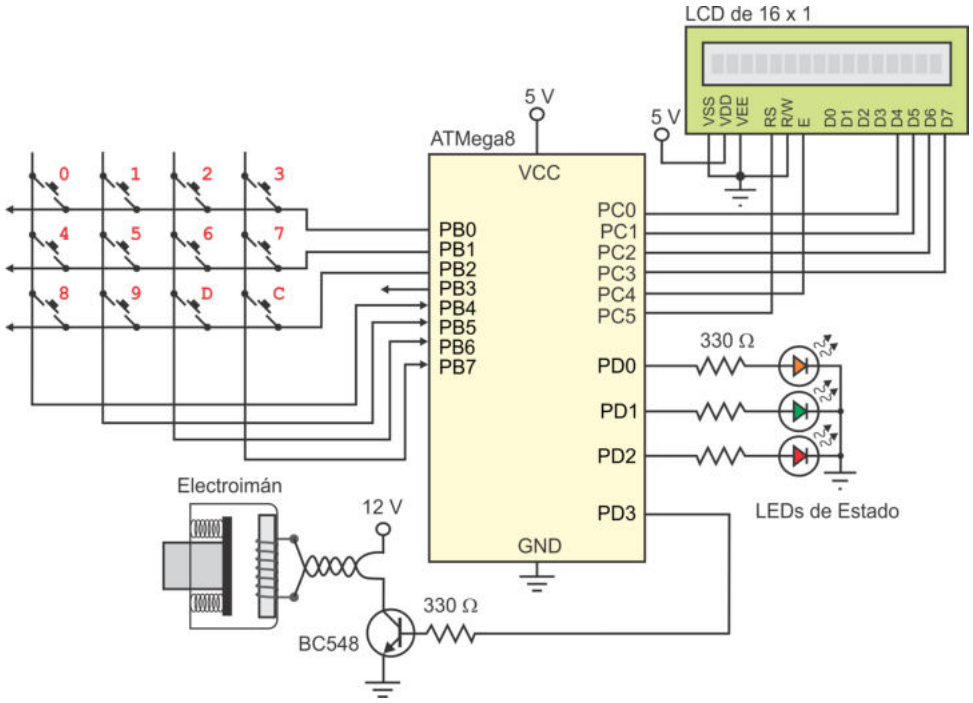


Figura 11.3: Diseño del hardware de la chapa electrónica

11.2.4. Diseño del Software

Para el diseño del software se debe realizar un diagrama de flujo en el que se muestre el comportamiento global de sistema, considerando que podrá estar en tres diferentes estados, el Estado Inicial, el Modo Apertura y el Modo Cambio de Clave. El cambio de estado puede deberse a dos situaciones diferentes: el usuario presionó una tecla o terminó el tiempo de espera. Estas situaciones serán detectadas por interrupción y

en sus rutinas de atención se modificarán banderas para su revisión en el programa principal.

El diagrama de flujo resultante debe proporcionar una idea clara de cómo se escribirá el programa principal. El sistema tiene una complejidad media y esto dificulta realizar un diagrama con todos los detalles para una codificación directa. Es importante que el diagrama de flujo tenga un estilo estructurado, es decir, que permita emplear las estructuras de control de lenguaje C. Por ello, será necesario revisar la disyunción de diferentes banderas para determinar el cambio de estado del sistema y, posterior a ello, se debe identificar la causa y hacer que el sistema le dé respuesta. Con estas consideraciones, en la Figura 11.4 se muestra el comportamiento del programa principal para la chapa electrónica.

El sistema inicia con la configuración de los recursos, incluyendo al LCD, los puertos de entrada/salida y el temporizador 1, los últimos con sus respectivas interrupciones y el temporizador 1 aún sin reloj. También se debe leer la clave de la EEPROM y llevarla a la RAM, así como limpiar todas las banderas. Todas estas acciones se realizan mediante proposiciones secuenciales.

Se observa en la Figura 11.4 que se tienen 4 banderas: `Tecla_Pres` indica que se presionó una tecla, `Tiempo` señaliza que ha concluido un periodo de tiempo predefinido, `Clave_Lista` para indicar que se han introducido de manera correcta 4 caracteres numéricos y `Cambio_Clave` que se pondrá en alto cuando el usuario presiona la tecla C, después de haber introducido la clave correcta, para pasar al Modo Cambio de Clave.

El lazo infinito empieza con el Estado Inicial, donde el sistema básicamente escribe el mensaje "Indique la Clave" y luego se queda en espera de que se presione una tecla. Con una tecla numérica el sistema pasa al Modo Apertura, el valor de la tecla se debe guardar en un arreglo y un contador de dígitos debe iniciar con 1.

En el Modo Apertura se limpia la pantalla LCD y se escribe un '$*$' por el carácter ya introducido, se activa al temporizador 1 porque el usuario dispone de 10 segundos para indicar la clave. El sistema debe recibir los 4 dígitos para poner en alto a la bandera `Clave_Lista`, escribiendo en el LCD un '$*$' por cada dígito tecleado, también debe respaldar su valor en el arreglo e incrementar el contador de dígitos.

Si se presiona la tecla de borrado (D), se debe observar si hay datos en el arreglo (contador $>$ 0), reducir al contador, mover el cursor a la izquierda 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.

Dado que hay dos condiciones posibles para terminar el ciclo de lectura de la clave, al salir del mismo se revisa la bandera `Clave_Lista`, si está en alto se compara la clave para saber si es correcta. Con la clave correcta, el sistema espera el tiempo necesario para saber si el usuario solicita el paso al Modo Cambio de Clave, si la clave está

incorrecta, solo se envía el mensaje de "Clave Incorrecta", durante 3 segundos y el sistema regresa al Estado Inicial. El sistema también va a regresar al Estado Inicial si la culminación del Modo Apertura no fue porque la clave estuviera completa, ya que en ese caso, se debió a que se agotó el tiempo de espera.

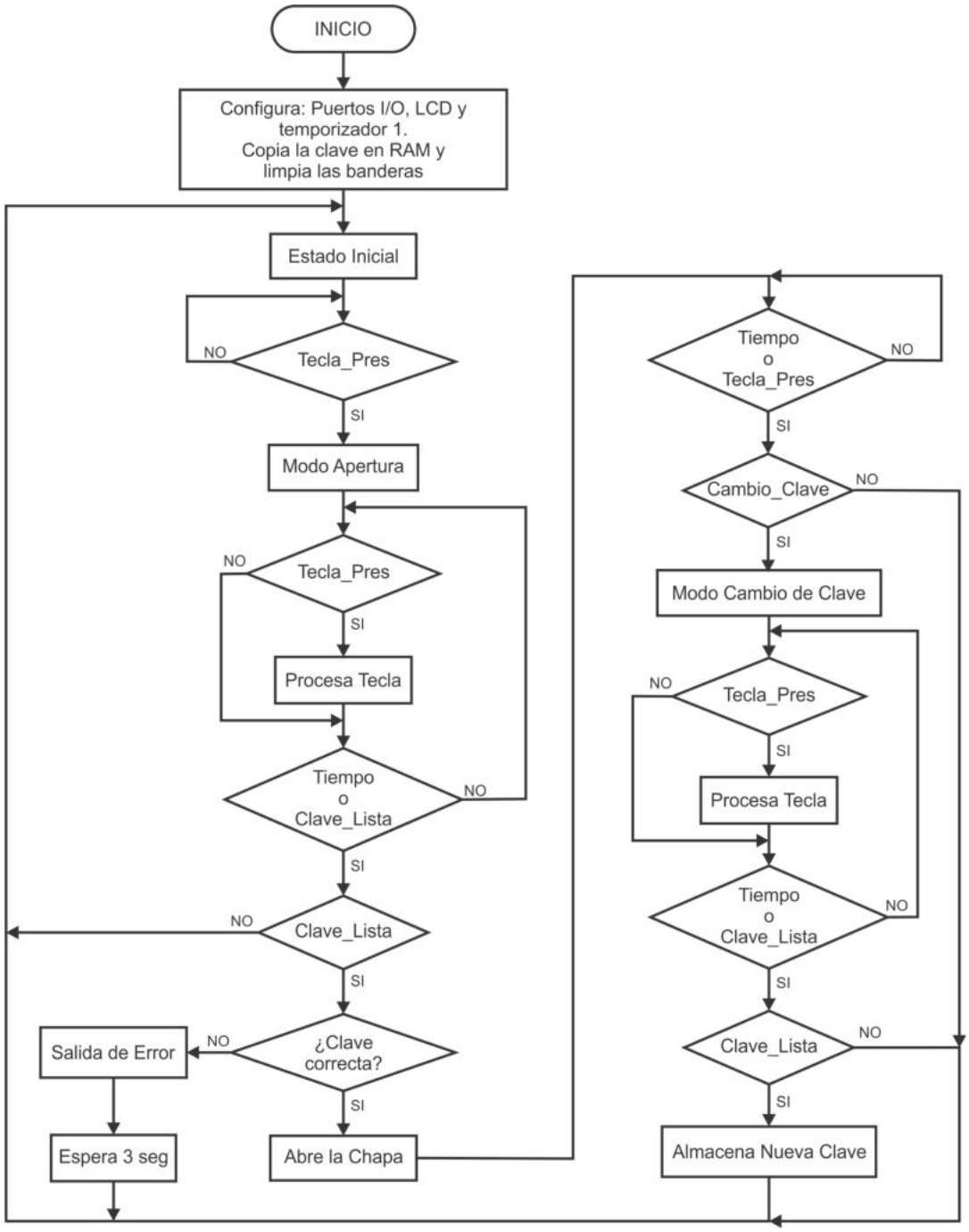


Figura 11.4: Diseño del software de la chapa electrónica

Se puede ver en la Figura 11.4 que el comportamiento del sistema en el Modo Cambio de Clave es similar al Modo Apertura, es decir, el sistema espera que el usuario introduzca los 4 dígitos de la nueva clave o que concluya el tiempo disponible para ello. La diferencia fundamental es que en este modo si se van a mostrar los dígitos en el LCD, para que el usuario vea la clave que está introduciendo.

Concluida la introducción de la nueva clave, esta se almacenará en EEPROM y se enviará al usuario el mensaje "Clave Aceptada". La escritura de la nueva clave puede apoyarse en las funciones desarrolladas para la EEPROM o en la biblioteca `eeprom.h` de Microchip Studio. Si el tiempo concluye sin que el usuario termine de insertar los 4 dígitos, la clave se mantiene sin cambios y el sistema regresa al Estado Inicial.

En el sistema se van a manejar diferentes periodos de tiempo, como se especificó en la fase de Planteamiento del Problema, esta tarea se hará con el apoyo del temporizador 1, para que de manera concurrente se pueda atender al teclado. El temporizador 1 debe configurarse para que interrumpa cada segundo y, antes de iniciar con un periodo de tiempo, en una variable se debe indicar la cantidad de segundos requerida. En la rutina de atención a la interrupción del temporizador, el contador de segundos debe decrementarse en 1 y cuando alcance el valor de 0, se debe poner en alto a la bandera `Tiempo`, que es monitoreada en el programa principal.

Con respecto al teclado, a través de las interrupciones por cambios en los puertos se va a detectar si se presionó una tecla, para que esto sea posible, en las salidas PB0, PB1 y PB2 deberán colocarse niveles bajos de voltaje, así, sin importar qué tecla se presione, se va a generar un evento en una de las entradas (PB7 a PB4). En la rutina de atención a la interrupción se debe llamar a la función que sondea al teclado, similar a la descrita en la Sección 10.2, con los ajustes necesarios porque ahora el teclado es de 3 x 4.

Con el diagrama de flujo del programa principal, las descripciones textuales para las rutinas de servicio a interrupciones y considerando el diseño del hardware, ya se tienen todos los elementos para codificar el programa en lenguaje C.

11.2.5. Implementación del Hardware

Consiste en el armado físico del circuito de la Figura 11.3. En la Figura 11.5 se muestra el hardware resultante, después de implementar el circuito impreso, aunque las primeras pruebas se hicieron en protoboard. No se incorporó al electroimán, en su lugar se agregó otro LED de estado. Como parte de la implementación del hardware, se evaluaron las funciones del LCD y del teclado, para garantizar que las conexiones se realizaron de manera correcta.

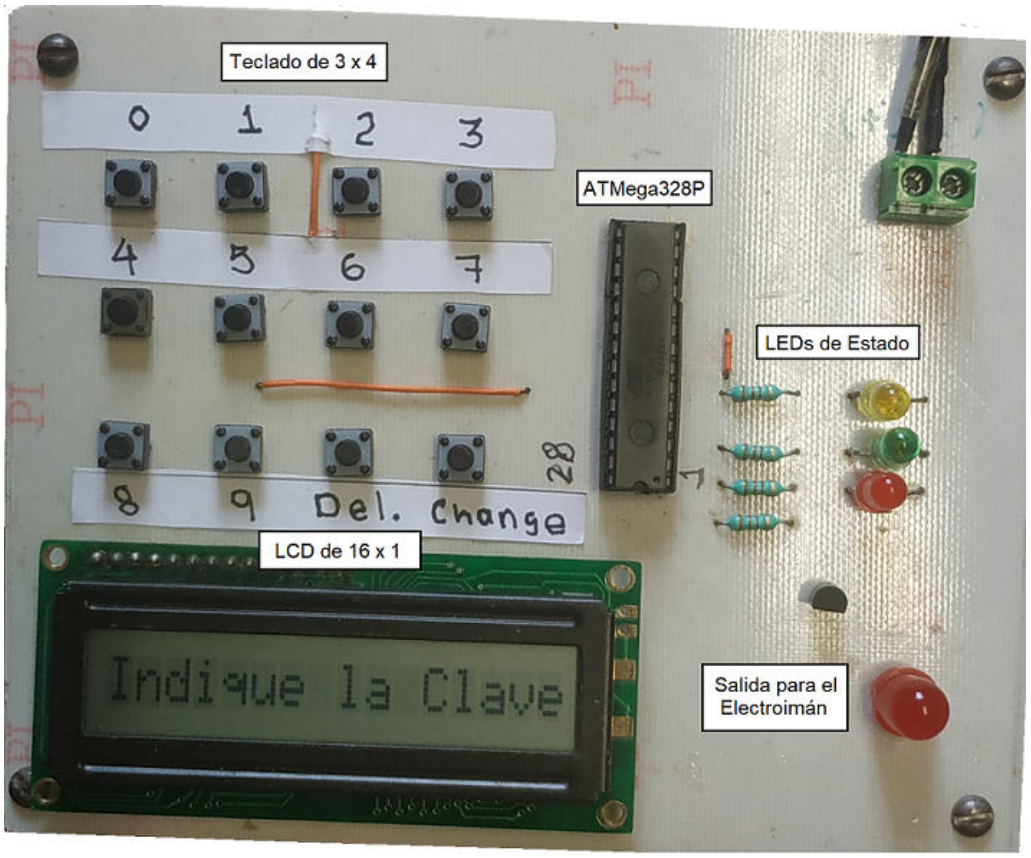


Figura 11.5: Implementación del hardware correspondiente a la chapa electrónica

11.2.6. Implementación del Software

El programa se implementó en lenguaje C, en la primera parte se incluyen las bibliotecas, se definen las constantes y las variables globales. Deben ser globales aquellas variables que serán modificadas en las rutinas que dan servicio a las interrupciones. El código de está primera parte es el siguiente:

```
#define     F_CPU   1000000UL       // El MCU opera a 1 MHz

#include    <avr/io.h>              // Entradas y salidas
#include    <util/delay.h>          // Para los retardos
#include    <avr/interrupt.h>       // Para las interrupciones
#include    <avr/eeprom.h>          // Para la EEPROM
#include    <avr/pgmspace.h>        // Para las constantes en Flash
#include    "LCD.h"                 // Biblioteca del LCD

EEMEM uint8_t   clave_inicial[4] = { 1, 2, 3, 4};       // EEPROM
const uint8_t   sec[] PROGMEM = { 0xFE, 0xFD, 0xFB};    // Flash

uint8_t   Tiempo, conta_seg;                    // Periodos de tiempo
volatile uint8_t   Tecla_Pres, val_tecla;       // Sondeo del teclado
```

Se observa en el código que en las variables empleadas para el sondeo del teclado se utiliza al atributo `volatile` y que no es necesario en las variables utilizadas por el temporizador. En la Sección 4.3 se explica el uso de este atributo y se aclara que su uso se puede omitir en algunos recursos.

La siguiente secuencia de código corresponde con las rutinas que dan servicio a las interrupciones, son necesarias dos ISRs, una para que el temporizador 1 maneje el periodo de 1 segundo y la otra para que las interrupciones por cambios en los puertos hagan un sondeo en el teclado. El código de las ISRs es:

```
ISR(TIMER1_COMPA_vect) {                    // ISR del temporizador 1
  conta_seg--;                              // Interrumpe cada segundo
  if(conta_seg == 0)
    Tiempo = 1;                             // Periodo concluido
}

ISR(PCINT0_vect) {                          // Cambios en PB4 a PB7
uint8_t i, aux;

  for(i = 0; i < 3; i++) {                  // Sondea los renglones
    PORTB = pgm_read_byte(&sec[i]);
    asm("NOP");
    aux = PINB & 0xF0;
    if(aux != 0xF0) {
      switch(aux) {
        case 0xE0: val_tecla = i*4;     break;
        case 0xD0: val_tecla = i*4 + 1; break;
        case 0xB0: val_tecla = i*4 + 2; break;
        case 0x70: val_tecla = i*4 + 3;
      }
      Tecla_Pres = 1;                       // Tecla presionada
      break;                                // No revisa más
    }
  }
  PORTB = 0xF0;                             // Salidas en bajo
  _delay_ms(150);
  if(PCIFR & 0x01)                          // Ocurrió un rebote
    PCIFR |= 0x01;                          // Limpia la bandera
}
```

En la última parte de la ISR de la `PCINT0` se elimina el rebote del teclado, el retardo y la limpieza condicionada de la bandera se colocaron como ajustes, los rebotes de los botones se presentaron en el sistema funcionando, no en la simulación.

La última parte del código corresponde con el programa principal, debe incluir la configuración de los recursos utilizados y tener el comportamiento mostrado en la Figura 11.4, el código es:

```
int main(void) {                    // Programa Principal
uint8_t  clave[4];                  // Clave actual
uint8_t  clave_in[4];               // Clave de entrada
```

```
uint8_t  i = 0;                  // Contador de datos
uint8_t  Cambio_Clave;           // Bandera para el cambio de la clave
uint8_t  Clave_Lista;            // Bandera para la clave lista

  DDRB = 0x0F;                   // Entrada y salida, para el teclado
  PORTB = 0xF0;                  // Pull-Up en las entradas
  DDRC = 0xFF;                   // Salida para el LCD
  DDRD = 0x0F;                   // LEDs de estado y electroimán

  LCD_reset();                   // Inicializa al LCD

  for( i = 0; i < 4; i++)        // Obtiene la clave de la EEPROM
    clave[i] = eeprom_read_byte(&clave_inicial[i]);

  // Configuración parcial del temporizador 1
  TIMSK1 = 0x02;                 // El temporizador 1 interrumpirá
  OCR1A = 15624;                 // cada segundo (modo CTC)
  TCCR1A = 0x00;                 // pero aún no arranca
  TCCR1B = 0x08;                 // TCCR1B = 0x0B, preescala de 64

  // Configuración de las interrupciones
  PCMSK0 = 0xF0;                 // Cambios en PB7:PB4
  PCICR = 0x01;                  // Habilita la interrupción

  sei();                         // Habilitador global de interrupciones

  while(1) {

    // Establece el Estado Inicial
    PORTD = 0x00;                // Salidas apagadas
    LCD_write_cad("Indique la Clave", 16);
    do {
      Tecla_Pres = 0;            // Limpia bandera
      while(!Tecla_Pres);        // Espera una tecla numérica
    } while( val_tecla < 0 || val_tecla > 9 );

    Tecla_Pres = 0;              // Limpia bandera

    // Inicia el Modo Apertura con una tecla presionada
    clave_in[0] = val_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 al LED naranja
                                 // porque hay actividad
    Tiempo = 0;                  // Limpia las banderas
    Clave_Lista = 0;
    conta_seg = 10;              // Periodo de 10 segundos
    TCNT1 = 0;
    TCCR1B = 0x0B;               // Activa al temporizador 1
    do {                         // Ciclo repetitivo para leer la clave
```

```
    if( Tecla_Pres) {
      if( val_tecla >= 0 && val_tecla <= 9 ) { // Tecla numérica
        clave_in[i] = val_tecla;               // guarda su valor
        LCD_write_data('*');
        i++;
        if( i == 4 )                           // Revisa si ya son
          Clave_Lista = 1;                     // 4 dígitos
      }
      else if(val_tecla == 0x0A) {             // Tecla de borrado
        if( i > 0 ) {                          // Hay datos?
          LCD_cursor(0x03 + i);                // borra el último
          LCD_write_data(' ');
          LCD_cursor(0x03 + i);                // Ubica el cursor
          i--;                                 // Ajusta el contador
        }
      }
      Tecla_Pres = 0;                          // Tecla procesada
    }
  } while( ! ( Tiempo || Clave_Lista));

  TCCR1B = 0x08;                               // Detiene al timer 1
  if(Clave_Lista) {                            // Clave completa
    Clave_Lista = 0;
    Tiempo = 0;
    conta_seg = 3;                             // Periodo de 3 seg
    TCCR1B = 0x0B;                             // Activa al timer 1
    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
      LCD_write_cad("<< Bienvenido >>", 16);   // Abre la chapa
      Cambio_Clave = 0;
      do{                                      // Espera petición de
        if(Tecla_Pres) {                       // cambio de clave
          if(val_tecla == 0x0B)
            Cambio_Clave = 1;
            Tecla_Pres = 0;
          }
      } while( !Tiempo && !Cambio_Clave);
      TCCR1B = 0x08;

      // Revisa la bandera que da paso al Modo Cambio de Clave
      if(Cambio_Clave) {
        LCD_write_cad("Cambio  de Clave", 16);
        i = 0;
        TCNT1 = 0;                             // 10 segundos para
        TCCR1B = 0x0B;                         // introducir la clave
        conta_seg = 10;
        Tiempo = 0;                            // Prepara banderas
        Tecla_Pres = 0;
        Clave_Lista = 0;
        do {                                   // Ciclo para leee la
          if( Tecla_Pres) {                    // nueva clave
```

```
            if( val_tecla >= 0 && val_tecla <= 9 ) {
          clave_in[i] = val_tecla;          // Tecla numérica
          if ( i == 0 ) {                   // Primer dígito?
            LCD_clear();                    // ubica al cursor
            LCD_cursor(0x04);
            PORTD = 0x01;                   // LED naranja
          }                                 // (hay actividad)
          LCD_write_data(val_tecla + 0x30); // Escribe el dígito
          i++;                              // lo cuenta
          if( i == 4 )                      // Con 4 dígitos,
            Clave_Lista = 1;                // la clave está lista
        }
        else if(val_tecla == 0x0A) {        // Tecla de borrado
          if( i > 0 ) {                     // Si hay datos,
            LCD_cursor(0x03 + i);           // elimina al último
            LCD_write_data(' ');            // dígito
            LCD_cursor(0x03 + i);
            i--;
          }
        }
        Tecla_Pres = 0;                     // Tecla procesada
      }
    } while( ! ( Tiempo || Clave_Lista));

    TCCR1B = 0x08;                          // Detiene al timer 1
    if(Clave_Lista) {
      for( i = 0; i < 4; i++) {             // Respalda en EEPROM
        eeprom_write_byte(&clave_inicial[i], clave_in[i]);
        clave[i] = clave_in[i];             // y copia en RAM
      }
      TCNT1 = 0;                            // Muestra mensaje
      TCCR1B = 0x0B;                        // por 2 segundos
      conta_seg = 2;
      Tiempo = 0;
      PORTD = 0x02;                         // LED verde encendido
      LCD_write_cad(" CLAVE ACEPTADA ", 16);
      while( !Tiempo );
      conta_seg = 2;                        // También muestra la
      Tiempo = 0;                           // nueva clave por
      LCD_write_cad("RECIBIO ->", 10);      // 2 segundos
      LCD_cursor(0x13);
      for( i = 0; i < 4; i++)
        LCD_write_data(clave[i] + 0x30);
      while( !Tiempo );
      TCCR1B = 0x08;
    }                                       // Fin: nueva clave
  }                                        // Fin: Cambio de Clave
}                                         // Fin: clave correcta
else {
  PORTD = 0x04;                           // LED rojo
  LCD_write_cad("CLAVE INCORRECTA", 16);  // Clave Incorrecta
```

```
            while( !Tiempo );                    // Muestra por
            TCCR1B = 0x08;                       // 3 segundos
        }
    }                                          // Fin de Clave Lista
  }                                          // Cierra ciclo infinito
}                                          // Cierra el main
```

11.2.7. Integración y Evaluación

El programa desarrollado se compiló con ayuda del entorno de Microchip Studio, posteriormente se realizó la programación del microcontrolador ATMega328P con el apoyo del programador USB-Asp, un programador serial de bajo costo.

Se evaluó el funcionamiento de la chapa electrónica y se observó que de manera frecuente una tecla era considerada en dos más ocasiones, esto debido a los rebotes de los botones, por lo que fue necesario hacer un ajuste en el software para resolver el problema.

11.2.8. Ajustes y Correcciones

En cuanto al software, se agregó la secuencia para la eliminación de los rebotes del teclado, esta incluye un retardo y la limpieza condicionada de la bandera que indica la ocurrencia de un nuevo evento. El código mostrado ya incluye el ajuste requerido.

En cuanto al hardware, es importante aclarar que la etapa de implementación fue trabajada en protoboard y como un ajuste se hizo el desarrollo del circuito impreso, que se mostró en la Figura 11.5.

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 con mayores dimensiones.

11.3. Sistema Embebido Multifunción

En esta sección se describe el desarrollo de un sistema embebido con las funciones de reloj de tiempo real, calendario, alarma, temporizador y termómetro, con salida en 4 displays de 7 segmentos.

El sistema se diseña bajo un enfoque incremental, en cada versión o incremento se agrega una de las funciones que se han listado y se aplica la metodología de 8 pasos descrita en la Sección 11.1. Además de ilustar las técnicas de diseño, en este sistema se utilizan las interfaces SPI y TWI, así como el ADC, por lo que es un ejemplo muy completo para mostrar el uso de recursos del ATMega328P.

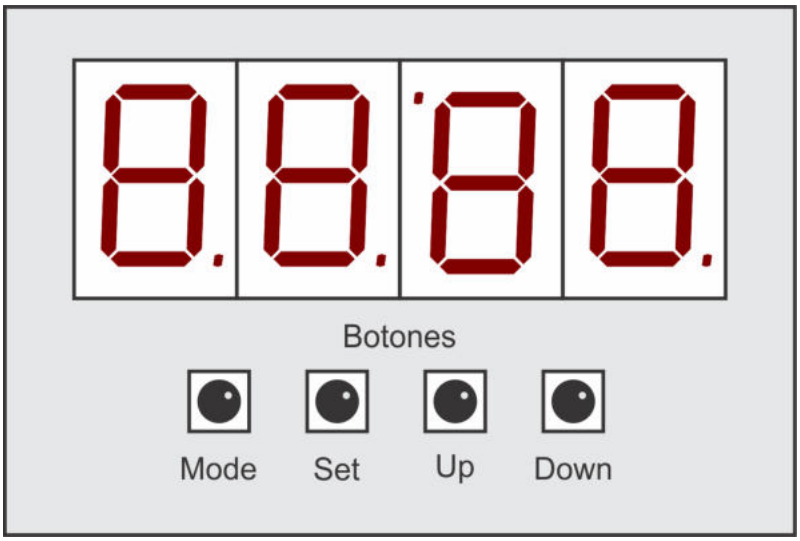


Figura 11.6: Reloj de tiempo real

11.3.1. Incremento 1: Reloj de Tiempo Real

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 se puede utilizar para manejar displays grandes, muy útil para espacios públicos. Un reloj puede ser la base para un sistema que requiere temporización, como el control de un horno de microondas, de una lavadora, etc., en este caso, el primer incremento en el sistema embebido multifunción será un reloj de tiempo real.

Planteamiento del Problema

El reloj debe mostrar su salida en 4 displays de 7 segmentos, el tercer display se debe conectar en forma invertida para proporcionar los 2 puntos intermedios. En este primer incremento se mostrará la hora y minutos, o los segundos. El reloj manejará un formato de 24 horas, para evitar un indicador adicional de AM y PM. Los 3 parámetros: hora, minutos y segundos deben ser configurables, por lo tanto, se considera conveniente que el sistema tenga 4 botones, además de los displays, en la Figura 11.6 se muestran las características esperadas en el primer incremento del sistema, se utilizan nombres en inglés para los botones por ser más cortos.

La funcionalidad para los botones será la siguiente:

- **Mode:** Determina el modo del sistema, inicialmente se mostrará la hora y minutos, con los dos puntos intermedios parpadeando (modo 0). Cuando se presione el botón **Mode** pasará a mostrar los segundos (modo 1). Al presionar nuevamente el mismo botón regresará al modo 0.

- **Set:** Es el botón de configuración, en el modo 0 permite configurar a la hora y los minutos, y en el modo 1 se podrán configurar los segundos. Con el mismo botón se termina la configuración.
- **Up:** Cuando la configuración esté activa, con este botón se podrá incrementar el parámetro bajo configuración. Se ignora si no hay una configuración activa.
- **Down:** Similar al botón **Up**, pero es para decrementar el parámetro bajo configuración.

Estado inicial

El sistema iniciará en el modo 0, mostrando la hora y minutos con los dos puntos parpadeando. Se utilizará un módulo RTC externo al ATMega328P para conservar la hora, de manera que se hará una lectura inicial para obtener la información, la cual se actualizará cada medio segundo.

En cuanto a los botones, en el estado inicial solo se atenderán los botones **Mode** y **Set**, los otros dos serán ignorados mientras no inicie una configuración.

Requerimientos de Hardware y Software

Con respecto al hardware, el sistema será soportado por un ATMega328P y, un aspecto importante del proyecto es que se utilicen las interfaces del MCU, por ello, en este sistema se utilizarán dos dispositivos externos:

- El CI MAX7219 para manejar los 4 displays de 7 segmentos por medio de la interfaz SPI.
- El módulo RTC basado en el CI DS1307, con su batería integrada, para conservar la hora en ausencia de energía.

Además del MCU y los dos dispositivos externos, en este primer incremento se requiere de:

- 4 botones, nombrados como **Mode**, **Set**, **Up** y **Down**.
- 4 displays de 7 segmentos, de cátodo común porque así los requiere el MAX7219.
- 1 resistor de 50 kΩ para limitar la corriente en el MAX7219.
- 2 capacitores para eliminar ruido en el MAX7219, uno de 100 nF y otro de 10 mF.

En cuanto al software, a partir de la biblioteca `TWI.h` se van a desarrollar funciones para el acceso al módulo RTC. Además de acondicionar las funciones vistas en el Capítulo 8 para el manejo del MAX7219, a través de la interfaz SPI.

Los recursos internos adicionales que se emplearán son:

- Las interrupciones de los puertos para atender a los botones, **Mode** y **Set** por cambios en las terminales, **Up** y **Down** por interrupciones externas.
- El temporizador 1 para actualizar la información que se está exhibiendo, dependiendo del modo de operación y del estado de la configuración.

Diseño del Hardware

La mayoria de los dispositivos de hardware a emplear ya tienen un lugar para su conexión en el ATMega328P, por los recursos internos. El MAX7219 utiliza la interfaz SPI, por lo que se conecta en el Puerto B, utilizando las terminales PB2 (SS), PB3 (MOSI) y PB5 (SCK). El módulo RTC utiliza la interfaz TWI, este debe conectarse en el Puerto C, en las terminales PC5 (SCL) y PC4 (SDA). Los botones **Up** y **Down** serán atendidos por interrupciones externas, deben conectarse en PD2 (INT0) y PD3 (INT1).

Finalmente, están los botones **Mode** y **Set**, dado que serán atendidos por interrupciones debidas a cambios en las terminales, se pueden conectar en cualquier terminal libre, por conveniencia se dejan junto a los otros botones, ocupando las terminales PD0 y PD1. El hardware resultante se muestra en la Figura 11.7.

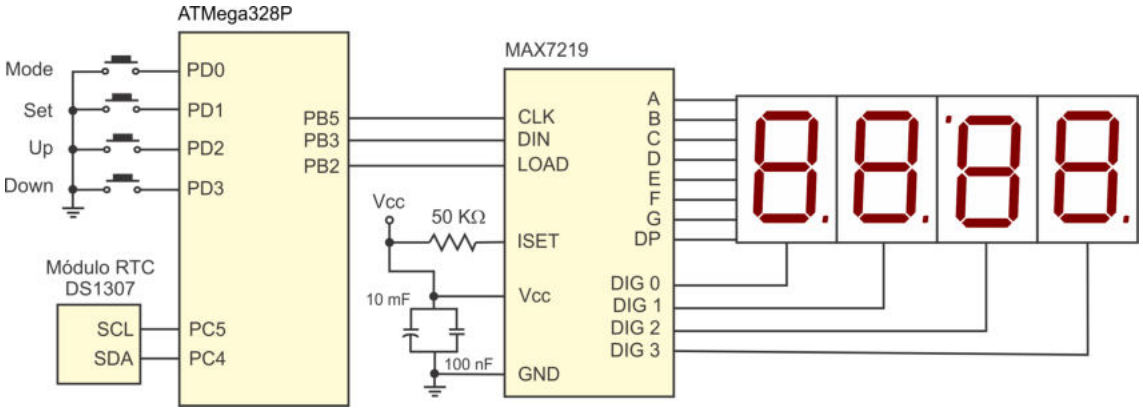


Figura 11.7: Diseño del hardware de un reloj de tiempo real

Diseño del Software

Para el diseño del software se debe separar entre las tareas que se realizarán en el programa principal y las tareas que harán las ISRs. En total se tienen 4 ISRs, dos debidas a las interrupciones externas, una debida a los cambios en PD0 y PD1, y la última debida al temporizador 1, con el que se establece la base de tiempo para la exhibición de la información.

En el programa principal básicamente se van a configurar los recursos y establecer el estado inicial del sistema, a partir de ello, la CPU queda en modo ocioso, en espera de que sean las ISRs quienes determinen el comportamiento del sistema. En el programa principal se realizan los siguientes pasos:

1. Configura entradas y salidas.
2. Define el estado inicial: `mode = 0`, `set = 0`, `dp = 0`.
3. Configura las interrupciones externas, para que se activen con flancos de bajada.
4. Configura las interrupciones por cambios en PD0 y PD1.
5. Configura la interfaz TWI.
6. Lee la hora actual (hora, minutos y segundos).
7. Configura la interfaz SPI.
8. Inicializa al MAX7219.
9. Muestra la hora actual en los displays, con puntos intermedios apagados (porque `dp` tiene 0).
10. Configura al temporizador 1, para que interrumpa cada medio segundo.
11. Activa al habilitador global de interrupciones.
12. En el lazo infinito, la CPU se mantiene ociosa (sin ejecutar instrucciones).

Hay 3 variables en el algoritmo mediante las cuales se establece el estado inicial, su funcionalidad es la siguiente:

- La variable `mode` solo tomará los valores de 0 y 1, para mostrar hora y minutos (`mode = 0`) o segundos (`mode = 1`).
- La variable `set` inicia con 0 indicando que no hay una configuración en proceso, si la variable `mode` está en 0, la variable `set` podrá tomar los valores 0, 1 y 2, porque se puede configurar la hora (`set = 1`) y los minutos (`set = 2`); pero si la variable `mode` está en 1, `set` solo puede tomar los valores de 0 o 1, para poder configurar los segundos.
- La variable `dp` inicia con 0 para indicar que los dos puntos van a estar apagados, se va a conmutar automáticamente cada medio segundo, con el apoyo del temporizador 1, para que se observe el parpadeo, aunque esto solo se hará en el modo 0. Cuando esté activa la configuración de algún parámetro, la variable `dp` ayudará a realizar un parpadeo para un efecto visual.

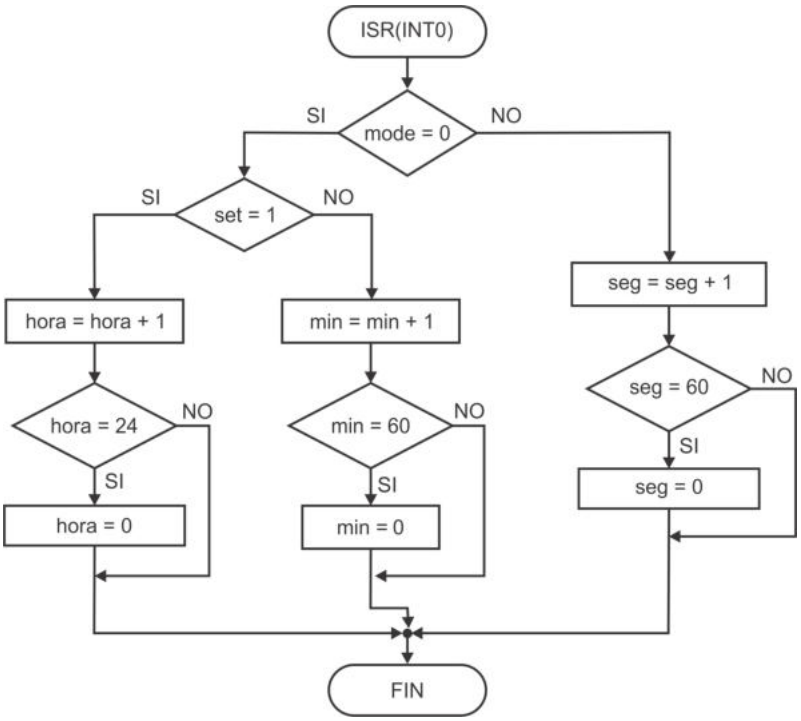


Figura 11.8: Comportamiento de la INT0, para incrementar un parámetro del reloj

Interrupciones externas

Las interrupciones externas no están activas en todo momento, se activan cuando se ha dado paso a la configuración de un parámetro. Con el botón **Up** se genera la INT0 para incrementar el parámetro y con el botón **Down** se decrementa al parámetro, mediante la INT1. El parámetro a modificar y su límite, depende de las variables `mode` y `set`, en la Figura 11.8 se muestra el comportamiento de la ISR de la INT0. La INT1 tiene un comportamiento similar, solo que involucra decrementos, por ello, no se realizó su diagrama de flujo.

Interrupciones por cambios en las terminales

En el programa principal se debe configurar para que las terminales PD0 y PD1 generen una interrupción. En esas terminales se conectarán los botones **Mode** y **Set**, y por software se debe identificar al botón que ha sido presionado. Con ello, se cambia el valor de las variables `mode` y `set`, en la Figura 11.9 se muestra el comportamiento de la ISR que atiende estos eventos. Básicamente se condicionan los cambios en las variables, considerando que si la variable `mode` está en 0, la variable `set` podrá tomar los valores 0, 1 y 2, porque se puede configurar la hora (`set = 1`) y los minutos (`set = 2`); pero si la variable `mode` está en 1, `set` solo puede tomar los valores de 0 o 1, para poder configurar los segundos.

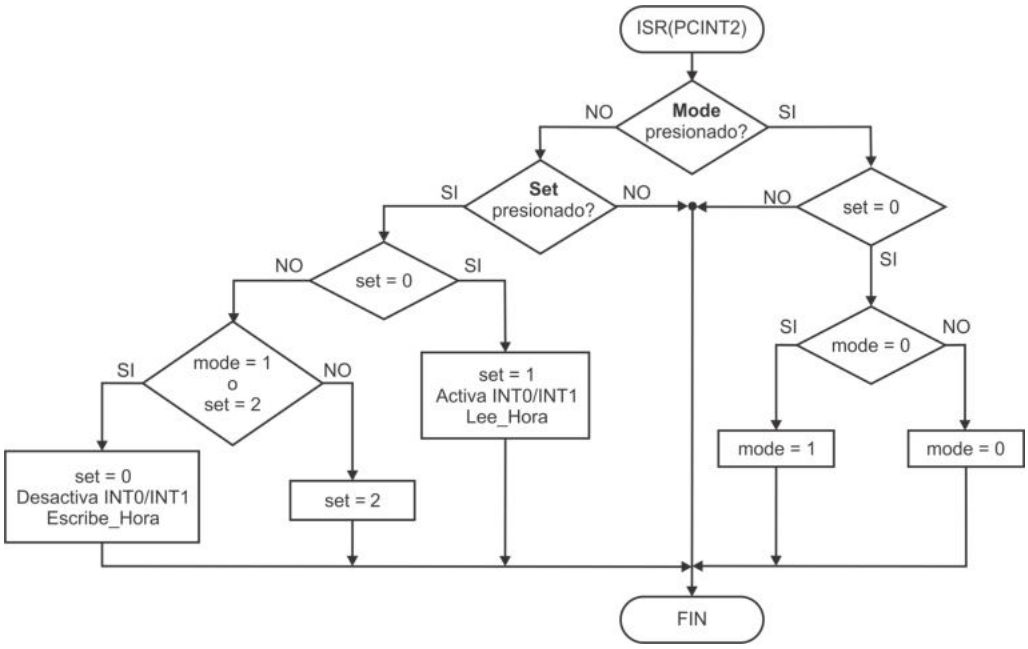


Figura 11.9: Comportamiento de la PCINT2, define el modo o el tipo de ajuste

Las interrupciones se producen cuando un botón se presiona o cuando se libera, por ello, en el diagrama de flujo se observa que puede ocurrir el evento y que ninguno de los botones esté presionado.

También se aprecia en la Figura 11.9 que se hace la lectura del módulo RTC cuando el sistema entra al modo de configuración, para contar con la información actualizada, y la información se respalda en el RTC al concluir con la configuración del sistema.

Interrupción debida al temporizador 1

El temporizador 1 se debe configurar para que produzca un evento de coincidencia por comparación cada medio segundo, en su rutina de interrupción se evaluarán las variables `mode` y `set`, para determinar la salida que se mostrará en los displays de 7 segmentos. La variable `dp` se va a conmutar en cada interrupción, su valor se utiliza en el parpadeo de los dos puntos o de un parámetro bajo configuración. El comportamiento general de la ISR se muestra en la Figura 11.10.

Con el algoritmo del programa principal y los diagramas de flujo de las ISRs se determina el comportamiento que tendrá el sistema. Los diagramas de flujo solo incluyen estructuras de decisión, se podrían catalogar como simples, sin embargo, el sistema en su conjunto responde a la combinación de los diferentes eventos, las variables globales ayudan al trabajo colaborativo de las ISRs.

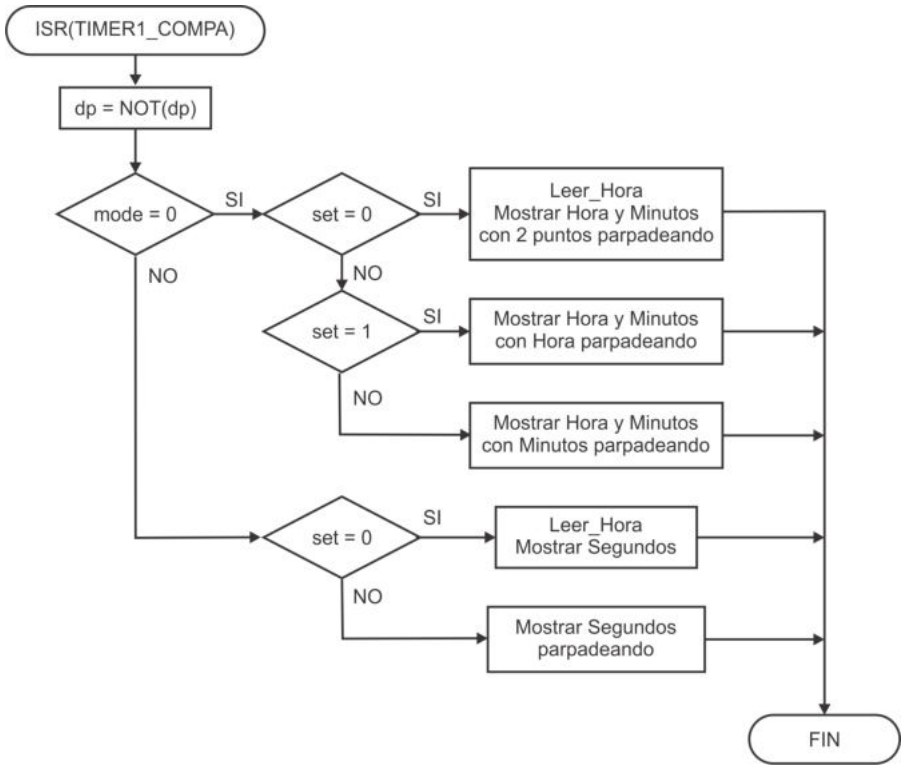


Figura 11.10: Comportamiento de la interrupción del temporizador 1

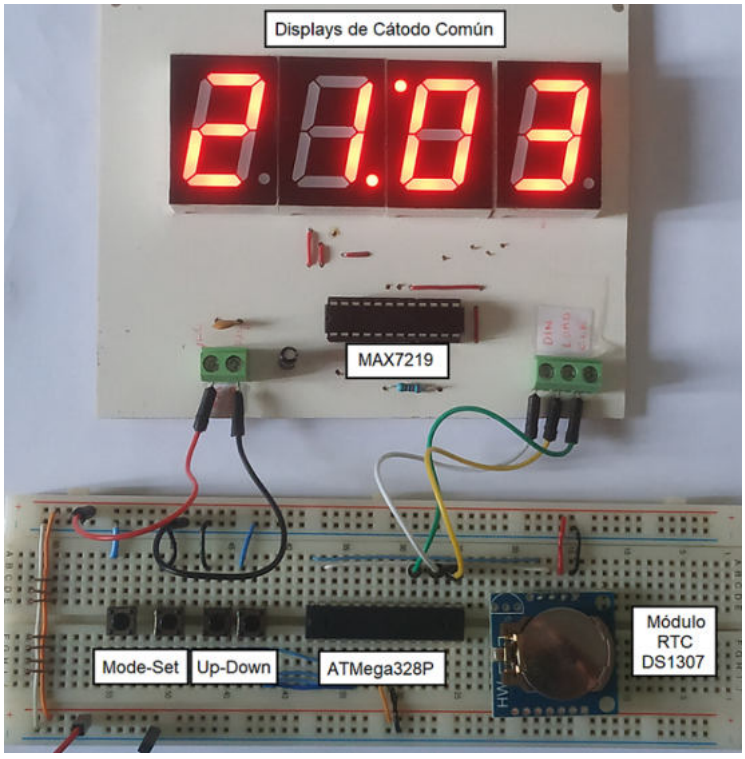


Figura 11.11: Implementación del Hardware del reloj de tiempo real

Implementación del Hardware

Consiste en el armado físico del circuito mostrado en la Figura 11.7. La conexión de los displays de 7 segmentos con el MAX7219 se complica por la decisión de colocar al tercer display en forma invertida para juntar los 2 puntos. Por ello, se optó por realizar un circuito impreso para el MAX7219 y los displays, los dispositivos restantes se manejaron en protoboard hasta que se alcanzó el último incremento. En la Figura 11.11 se muestra la implementación del hardware para el reloj de tiempo real.

Implementación del Software

Similar a la etapa de diseño, se separa el programa principal de las rutinas de atención a las interrupciones, pero primero se debe considerar la inclusión de bibliotecas, los prototipos de las funciones y la definición de constantes y variables globales. El código de esta primera parte es el siguiente:

```
#define  F_CPU       1000000UL
#define  DIR_SLVO    0x68          // Dirección del RTC

#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>
#include <avr/pgmspace.h>
#include  "TWI.h"

const uint8_t Tabla[] PROGMEM = {0x7E, 0x06, 0x6D, 0x4F, 0x17,
                                 0x5B, 0x7B, 0x0E, 0x7F, 0x1F}; // 0-9
uint8_t  mode, set, dp;
uint8_t  hora_act[6];             // Almacena la hora actual
uint8_t  displays[4];             // Información para los displays

void hab_RTC(), lee_hora(), esc_hora(); // Funciones para el RTC

void MAX7219_init(), MAX7219_Datos();   // Funciones para el MAX7219
void MAX7219_escDato(uint8_t dir, uint8_t dato);
```

El funcionamiento del MAX7219 se describió en la Sección 10.3, como una alternativa para reducir el número de terminales al utilizar displays de 7 segmentos. El chip incluye un decodificador de 7 segmentos que opcionalmente se puede habilitar, en este sistema, el decodificador se utiliza para los dígitos 0, 1 y 3, en el dígito 2 no se puede emplear porque el display se coloca de manera invertida para juntar los dos puntos, para ese display se establece la tabla de constantes en memoria Flash que se observa en el código previo.

En el código se observan dos arreglos porque la información a mostrar depende de las variables `mode` y `set`, por ello, del arreglo llamado `hora_act[]` se obtiene la información que será colocada en `displays[]`.

El módulo RTC funciona con la interfaz TWI, en la sección de ejercicios del Capítulo 8 se describe al módulo y se deja como un ejercicio el desarrollo de las funciones para su acceso. La función `hab_RTC()` es para asegurar que el bit CH (*clock halt*) del módulo RTC está en bajo, sin modificar la hora. Las funciones `lee_hora()` y `esc_hora()` son para realizar la lectura y escritura de la hora, minutos y segundos contenidos en el módulo RTC, las funciones utilizan al arreglo global `hora_act[]` para dejar la información leída u obtener la información a escribir, las funciones se pueden desarrollar a partir del contenido del Capítulo 8.

Para el acceso al MAX7219 se utilizan 3 funciones, estas básicamente realizan transferencias SPI de 16 bits, que se pueden desarrollar a partir del contenido del Capítulo 7, solo se deben comprender los registros del MAX7219, que se describen en la Sección 10.3. La función `MAX7219_init()` establece la configuración inicial del chip, la función `MAX7219_escDato()` escribe en uno de los registros internos, recibe la dirección del registro a escribir y el dato que será escrito, y por último, la función `MAX7219_Datos()` lee el arreglo `displays[]` para escribir su contenido en los registros que corresponden con la salida a los 4 displays de 7 segmentos.

En el programa principal se realiza la configuración de los diferentes recursos, como se describió en el algoritmo descrito en el diseño del software. Para realizar el código se deben revisar los Registros I/O de los recursos involucrados y establecer su valor correcto. El código de la función `main` es:

```
int main(void) {

   DDRB = 0b00101100;        // SCK, MOSI y SS como salidas
   DDRD = 0x00;              // Entradas para los  botones
   PORTD = 0x0F;             // Pull-up en los botones
   PORTB = 0x04;             // SS inicia inactivo

   mode = set = dp = 0;      // Estado inicial del sistema

   EICRA = 0x0A;             // Interrupciones Externas
   EIMSK = 0x00;             // Inician inactivas

   PCMSK2 = 0x03;            // Cambios en PD0 y PD1
   PCICR = 0x04;

   TWI_Config();             // Configura la interfaz TWI
   hab_RTC();                // Asegura que el RTC no esté detenido
   lee_hora();               // Lee la hora actual

   SPCR = 0x50;              // SPI como maestro
   SPSR = 0X00;              // a 250 kHz
   MAX7219_init();           // Inicializa al MAX7219

   displays[0] = hora_act[0];
   displays[1] = hora_act[1];
   displays[2] = pgm_read_byte(&Tabla[hora_act[2]]);
```

```
    displays[3] = hora_act[3];
    MAX7219_Datos();          // Muestra la hora actual

    OCR1A = 62499;
    TCCR1A = 0x00;            // Modo CTC, preescala de 8
    TCCR1B = 0x0A;            // para medio segundo
    TIMSK1 = 0x02;            // Interrupción por comparación

    sei();                    // Habilitador global de interrupciones

    while (1)
        asm("NOP");
}
```

En los comentarios de cada secuencia de código se especifica el recurso que se está configurando.

Las interrupciones externas se utilizan para incrementar o decrementar los parámetros en el reloj de tiempo real, por medio de los botones **Up** y **Down**. En el diagrama de flujo de la Figura 11.8 se muestra el comportamiento para la INT0, su traducción a código C es la siguiente:

```
ISR(INT0_vect) {
uint8_t  temp;

    if(mode == 0){
        if(set == 1){
            temp = hora_act[0]*10 + hora_act[1];
            temp = (temp < 23)? temp + 1:0;
            hora_act[0] = temp/10;
            hora_act[1] = temp%10;
        } else {
            temp = hora_act[2]*10 + hora_act[3];
            temp = (temp < 59)? temp + 1:0;
            hora_act[2] = temp/10;
            hora_act[3] = temp%10;
            }
    }
    else {
        temp = hora_act[4]*10 + hora_act[5];
        temp = (temp < 59)? temp + 1:0;
        hora_act[4] = temp/10;
        hora_act[5] = temp%10;
    }
}
```

En el código anterior se observa que el parámetro a modificar depende de las variables `mode` y `set`. El código para la ISR de la INT1 es muy similar, solo que debe decrementar algún parámetro en lugar de incrementarlo.

Las interrupciones por cambios en los puertos determinan qué información se va a mostrar y qué parámetro se va a modificar, ajustando los valores para las variables `mode` y `set`. En el diagrama de flujo de la Figura 11.9 se muestra el comportamiento para la PCINT2, su traducción a código C es la siguiente:

```
ISR(PCINT2_vect) {

   if(!(PIND & 0x01)) {              // Botón Mode
      if(set == 0)
         mode = (mode == 0)? 1: 0;
   }
   else if(!(PIND & 0x02)) {         // Botón Set
      if(set == 0) {
         set = 1;
         EIMSK = 0x03;
         lee_hora();
      }
      else if(mode == 1 || set == 2) {
         set = 0;
         EIMSK = 0x00;
         esc_hora();
      } else
         set = 2;
   }
}
```

En el código se puede ver que en el momento en que se pasará a la configuración de un parámetro (`set = 1`), se activan las interrupciones externas y se lee al módulo RTC. Por el contrario, cuando termina la configuración, las interrupciones externas son deshabilitadas y se escribe en el módulo RTC, para conservar los cambios hechos por el usuario.

El temporizador 1 establece una base de tiempo a medio segundo, actualizando la información que se mostrará en los displays. La información a exhibir depende de las variables `mode` y `set`. En la Figura 11.10 se mostró el comportamiento de la ISR del temporizador 1, el código C para ese diagrama de flujo es el siguiente:

```
ISR(TIMER1_COMPA_vect) {

   dp = (dp == 0)? 1: 0;
   if(mode == 0) {
      switch(set) {
         case 0:
            lee_hora();
            displays[0] = hora_act[0];
            displays[1] = hora_act[1];
            displays[2] = pgm_read_byte(&Tabla[hora_act[2]]);
            displays[3] = hora_act[3];
            if(dp == 1) {
               displays[1] |= 0x80;          // 2 puntos
```

```
                displays[2] |= 0x80;
            }
            break;
        case 1:
            displays[2] = pgm_read_byte(&Tabla[hora_act[2]]);
            displays[3] = hora_act[3];
            if(dp == 1) {
                displays[0] = hora_act[0];
                displays[1] = hora_act[1];
            } else {
                displays[0] = 0x0F;
                displays[1] = 0x0F;
            }
            break;
        case 2:
            displays[0] = hora_act[0];
            displays[1] = hora_act[1];
            if(dp == 1) {
                displays[2] = pgm_read_byte(&Tabla[hora_act[2]]);
                displays[3] = hora_act[3];
            } else {
                displays[2] = 0x00;
                displays[3] = 0x0F;
            }
            break;
    }
  }
  else {                                    // mode = 1
    switch(set) {
        case 0:
            lee_hora();
            displays[0] = 0x0F;
            displays[1] = 0x0F;
            displays[2] = pgm_read_byte(&Tabla[hora_act[4]]);
            displays[3] = hora_act[5];
            break;
        case 1:
            displays[0] = 0x0F;
            displays[1] = 0x0F;
            if(dp == 1) {
                displays[2] = pgm_read_byte(&Tabla[hora_act[4]]);
                displays[3] = hora_act[5];
            } else {
                displays[2] = 0x00;
                displays[3] = 0x0F;
            }
            break;
    }
  }
  MAX7219_Datos();                  // Escribe en los displays
}
```

La primer tarea en la ISR es el ajuste de la variable `dp`, se puede ver en el código que su valor determina la forma en que se muestra la información y con ello se establece el comportamiento del sistema.

La implementación del software se complementa con los cuerpos de las funciones de apoyo, no se incluyen en el texto porque son parte de los ejercicios que se han dejado en capítulos previos.

Integración y Evaluación

El programa desarrollado se compiló para crear el archivo HEX, este se descargó en el microcontrolador ATMega328P. Con el MCU programado se evaluó el funcionamiento del hardware y este resultó correcto, el sistema muestra la hora actual (hora y minutos) y atiende a los botones **Mode** y **Set**.

Ajustes y Correcciones

El sistema no necesito de ajustes y correcciones, las pruebas se hicieron en protoboard y cumplió con las expectativas planeadas, los botones no generaron rebotes.

11.3.2. Incremento 2: Reloj/Calendario de Tiempo Real

Además de exhibir la hora, un reloj se puede complementar para mostrar la fecha, de esta forma se tendrá un sistema más versátil, que se puede emplear como punto de partida para construir sistemas que permitan agendar tareas en algunos días a determinadas horas.

Planteamiento del Problema

En la Figura 11.6 se puede ver el planteamiento establecido para el sistema en el incremento 1, este se mantiene para el incremento 2, ya que las características físicas se van a mantener y solo se incrementará la funcionalidad. Un sistema embebido puede crecer en hardware, en software o en ambos aspectos, en este caso, el incremento solo se hará en software.

Al sistema se le agregarán 2 modos de operación: el modo 2 para mostrar el día y el mes actual, y el modo 3 para exhibir el año actual. Con el botón **Mode** se harán los cambios, después del modo 3 el sistema regresará al modo 0.

En los modos 2 y 3 también se podrá presionar al botón **Set**, con el que se dará paso a la configuración. La variable `set` inicia con 0, indicando que no hay una configuración activa. En el modo 2 la variable `set` podrá tomar los valores 1 y 2, correspondiendo con la configuración del día y mes. En el modo 3, la variable `set` solo podrá tomar el valor de 1, para la configuración del año.

El estado inicial del sistema se mantiene, iniciará en el modo 0, mostrando la hora y minutos con los dos puntos parpadeando. El módulo RTC externo también tiene la capacidad de mantener la fecha actual, incluyendo el día de la semana, día del mes, mes y año, el día de la semana no será considerado.

Requerimientos de Hardware y Software

No hay cambios en los requerimientos de hardware para el incremento 2, se mantiene el mismo hardware que en el incremento 1.

En cuanto al software, se deben considerar dos nuevas funciones para el acceso al módulo RTC, una para la lectura de la fecha y otra para su escritura.

Se siguen empleando los mismos recursos internos, las interrupciones externas para atender los botones **Up** y **Down**, y la interrupción por cambios en PD0 y PD1, para monitorear los botones **Mode** y **Set**. Así como el temporizador 1, para actualizar la información que se está exhibiendo, dependiendo de las variables `mode` y `set`.

Diseño del Hardware

El hardware no cambió, la Figura 11.7 también corresponde al hardware para el incremento 2.

Diseño del Software

Se conserva la estructura general del software, se tiene al programa principal y las 4 ISRs. El estado inicial del sistema es el mismo, por ello, el programa principal tiene el mismo comportamiento que el exhibido en el incremento 1, solo se debe agregar un paso con la lectura de la fecha actual (dia, mes y año). También es necesario modificar las ISRs, para que cubran los nuevos requerimientos funcionales. Las variables `mode` y `set` se siguen empleando para definir el estado del sistema, en la Tabla 11.1 se muestran los valores que pueden tomar y el comportamiento del sistema en cada combinación.

Tabla 11.1: Estados posibles del reloj calendario

Mode	**Set**	**Descripción**
0	0	Muestra hora y minutos
0	1	Configura hora
0	2	Configura minutos
1	0	Muestra segundos
1	1	Configura segundos
2	0	Muestra mes y día
2	1	Configura día
2	2	Configura mes
3	0	Muestra año
3	1	Configura año

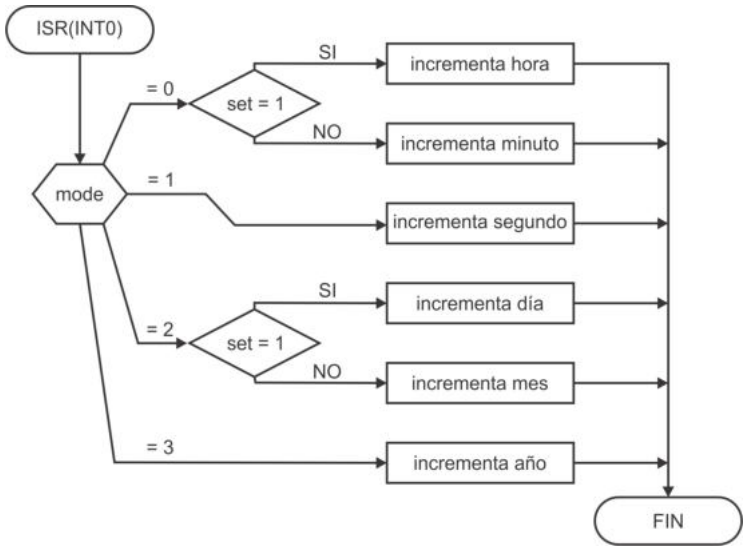


Figura 11.12: Comportamiento de la INT0, para incrementar un parámetro

Interrupciones externas

La funcionalidad de las interrupciones externas se mantiene, para atender a los botones **Up** y **Down**, con los que se harán ajustes a los parámtetros: hora, minutos, segundos, día, mes y año, en función de las variables `mode` y `set`. Con la INT0 se harán incrementos y con la INT1 los decrementos. En la Figura 11.12 se muestra el comportamiento de la ISR de la INT0, la estructura condicional doble de la Figura 11.8 se cambió por una condicional múltiple porque los modos de operación se han incrementado. El comportamiento para la ISR de la INT1 será similar, solo que con decrementos en vez de incrementos.

En la Figura 11.12 no se muestran detalles de la evaluación del límite en cada uno de los parámetros, pero es similar a la Figura 11.8. Sin embargo, debe tomarse en cuenta que el número máximo de días no es el mismo para todos los meses, por lo que es conveniente establecer una tabla de constantes con esos valores, además, un caso especial se presenta para febrero porque puede tener 28 o 29 días. Puesto que el módulo RTC solo maneja dos dígitos para el año, es suficiente con saber si es múltiplo de 4 para determinar que es bisiesto y sumar un día a febrero.

Interrupciones por cambios en las terminales

Los botones **Mode** y **Set** se están atendiendo con interrupciones por cambios en las terminales PD0 y PD1. Analizando la Figura 11.9, se puede notar que se requieren algunos ajustes porque la variable `mode` ahora puede llegar hasta 3, en los nuevos modos, se debe considerar el alcance de la variable `set`, y por último, ya no solo se lee y escribe la hora, también se lee y escribe la fecha, como resultado de estos ajustes se obtiene la Figura 11.13

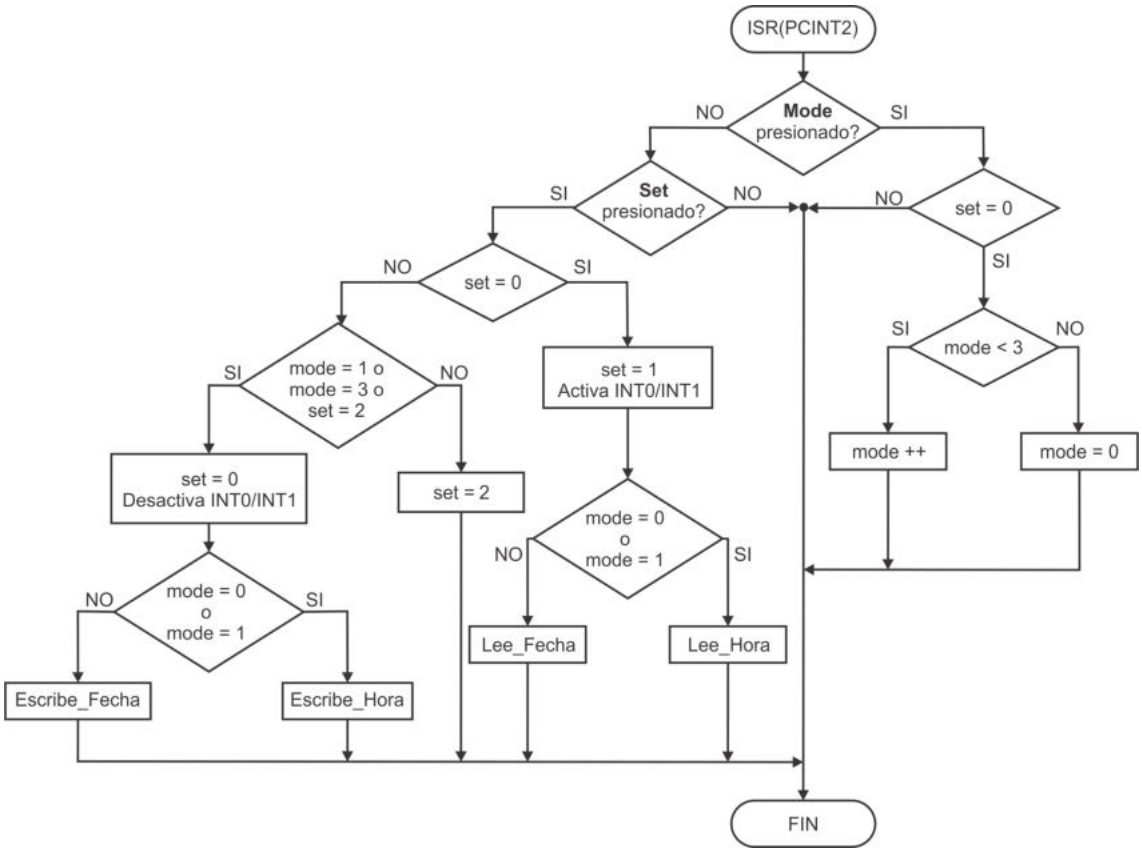


Figura 11.13: Comportamiento de la PCINT2, define el modo o el tipo de ajuste

Interrupción debida al temporizador 1

En la ISR del temporizador 1 se debe realizar un ajuste similar al de la interrupción externa 0, por tener más modos de operación. En el diagrama de flujo de la Figura 11.14, se debe remplazar la estructura condicional doble que evalúa la variable `modo`, por una estructura condicional múltiple, considerando que la variable `mode` ahora puede tomar valores del 0 al 3. En la Figura 11.14 se muestran los resultados de este ajuste, indicando qué se va a mostrar en cada caso.

Implementación del Hardware

En el incremento 2 del sistema se mantiene el hardware del incremento 1, por ello, la Figura 11.11 sigue correspondiendo con la implementación del hardware para el reloj/calendario de tiempo real.

Implementación del Software

Partiendo del software del incremento 1 del sistema, se revisa cada una de las partes y se describen los cambios que se hicieron para alcanzar el incremento 2.

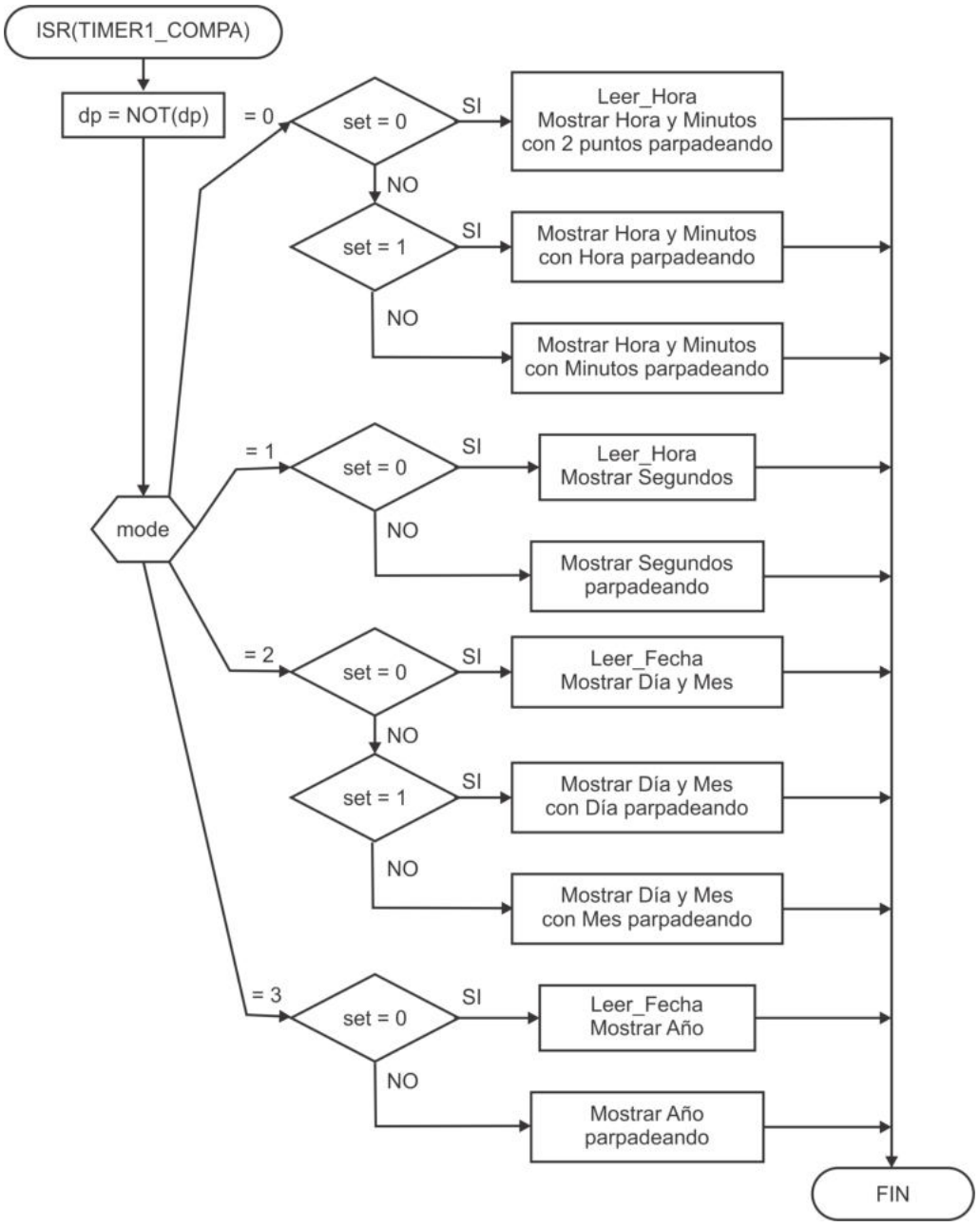


Figura 11.14: Comportamiento de la interrupción del temporizador 1

En la sección de definiciones, constantes y variables globales, se agregan los siguientes elementos, para el manejo del calendario:

```
const uint8_t Meses[] PROGMEM = { 31, 28, 31, 30, 31, 30,
                                  31, 31, 30, 31, 30, 31 };
uint8_t  fecha_act[6];
void     lee_fecha(), esc_fecha();
```

En el programa principal (función `main`) solo se agrega la llamada a la función `lee_fecha()` para actualizar el arreglo `fecha_act[]`, el estado inicial del sistema se mantiene y se configuran los mismos recursos, sin cambios en su operación.

Las interrupciones externas se utilizan para modificar los parámetros del sistema, con respecto al incremento 1, se agregaron 2 nuevas opciones para la variable `mode` y ahora se tienen más parámetros por modificar. En la Figura 11.12 se mostró el comportamiento de la ISR de la INT0 por medio de un `switch-case`, el código de esa estructura y las nuevas opciones para la variable `mode` es:

```
ISR(INT0_vect) {
uint8_t  temp, mes, max, anio;

   switch(mode) {
      case 0:
                  . . .       // Incrementa hora o minutos
      case 1:
                  . . .       // Incrementa segundos
      case 2:
         if(set == 1) {                                  // Incrementa día
            temp = fecha_act[0]*10 + fecha_act[1];
            mes = fecha_act[2]*10 + fecha_act[3];
            max = pgm_read_byte(&Meses[mes - 1]);
            if(mes == 2) {                               // febrero?
               anio = fecha_act[4]*10 + fecha_act[5];
               if(anio % 4 == 0)                         // año bisiesto?
                  max = max + 1;                         // suma 1
            }
            temp = (temp < max)? temp + 1:1;
            fecha_act[0] = temp/10;
            fecha_act[1] = temp%10;
         } else {                                        // Incrementa mes
            temp = fecha_act[2]*10 + fecha_act[3];
            temp = (temp < 12)? temp + 1:1;
            fecha_act[2] = temp/10;
            fecha_act[3] = temp%10;
         }
          break;
      case 3:
         temp = fecha_act[4]*10 + fecha_act[5];          // Incrementa año
         temp = (temp < 99)? temp + 1:0;
         fecha_act[4] = temp/10;
         fecha_act[5] = temp%10;
      break;
   }
}
```

Puede notarse en el código anterior que para conocer el número de días de cada mes se requieren instrucciones adicionales, porque no es el mismo en todos los meses, así como la consideración especial para los años bisiestos.

En la Figura 11.13 se muestra el comportamiento de las interrupciones por cambios en PD0 y PD1, el código C de esta ISR, considerando los cambios que se realizaron para este incremento, queda de la siguiente manera:

```
ISR(PCINT2_vect) {
   if(!(PIND & 0x01)) {              // Botón Mode
      if(set == 0)
         mode = (mode < 3)? mode + 1: 0;
   }
   else if(!(PIND & 0x02)) {         // Botón Set
      if(set == 0) {
         EIMSK = 0x03;
         set = 1;
         if(mode == 0 || mode == 1)
            lee_hora();
         else
            lee_fecha();
      }
      else if(mode == 1 || mode == 3 || set == 2) {
         EIMSK = 0x00;
         set = 0;
         if(mode == 0 || mode == 1)
            esc_hora();
         else
            esc_fecha();
      }
      else
         set = 2;
   }
}
```

La ISR del temporizador 1 también cambia su estructura al emplear un `switch-case` por el incremento en el número de modos de operación, pero su objetivo sigue siendo el mismo, determina qué se va a mostrar en los displays a partir de las variables `mode` y `set`. El código C con esta nueva estructura y las nuevas opciones para la variable `mode` es:

```
ISR(TIMER1_COMPA_vect) {
   dp = (dp == 0)? 1: 0;
   switch(mode) {
      case 0:
            . . .      // Muestra hora y minutos
      case 1:
            . . .      // Muestra segundos
      case 2:
         switch(set) {
            case 0:
               lee_fecha();
               displays[0] = fecha_act[0];
               displays[1] = fecha_act[1];
               displays[2] = pgm_read_byte(&Tabla[fecha_act[2]]);
```

```
                displays[3] = fecha_act[3];
            break;
            case 1:
                displays[2] = pgm_read_byte(&Tabla[fecha_act[2]]);
                displays[3] = fecha_act[3];
                if(dp == 1) {
                    displays[0] = fecha_act[0];
                    displays[1] = fecha_act[1];
                } else {
                    displays[0] = 0x0F;
                    displays[1] = 0x0F;
                }
            break;
            case 2:
                displays[0] = fecha_act[0];
                displays[1] = fecha_act[1];
                if(dp == 1) {
                    displays[2] = pgm_read_byte(&Tabla[fecha_act[2]]);
                    displays[3] = fecha_act[3];
                } else {
                    displays[2] = 0x00;
                    displays[3] = 0x0F;
                }
            break;
        }
    break;
    case 3:
        switch(set) {
            case 0:
                lee_fecha();
                displays[0] = 0x0F;
                displays[1] = 0x0F;
                displays[2] = pgm_read_byte(&Tabla[fecha_act[4]]);
                displays[3] = fecha_act[5];
            break;
            case 1:
                displays[0] = 0x0F;
                displays[1] = 0x0F;
                if(dp == 1) {
                    displays[2] = pgm_read_byte(&Tabla[fecha_act[4]]);
                    displays[3] = fecha_act[5];
                } else {
                        displays[2] = 0x00;
                    displays[3] = 0x0F;
                }
            break;
        }
    break;
  }
  MAX7219_Datos();
}
```

Un cambio significativo que se observa en los nuevos casos es que, al mostrar el día y el mes, no se hacen parpadear los 2 puntos intermedios.

Integración y Evaluación

El programa se compiló y el archivo HEX se descargó en el microcontrolador ATMega328P. Con el MCU programado se evaluó el desempeño del hardware y se observó un funcionamiento correcto, se pueden cambiar los modos de operación y realizar ajustes en cada caso.

Ajustes y Correcciones

Un ajuste importante que se hizo en el sistema fue la selección de la información a leer o escribir en el módulo RTC, apartir de la variable `mode`. En principio no se hacia esta distinción y las operaciones de lectura y escritura se realizaban con los dos arreglos (`hora_act` o `fecha_act`). El problema es que mientras el sistema se está configurando, un arreglo está siendo modificado por el usuario y el otro no se está actualizando porque no se están haciendo lecturas en el módulo RTC. Por ello, si se escriben ambos arreglos, uno puede tener información desactualizada. Este aspecto es más perceptible cuando se configura la fecha y el arreglo de la hora queda sin cambios. El diagrama de flujo de la Figura 11.13 y su código ya incluyen este ajuste.

11.3.3. Incremento 3: Reloj/Calendario con Alarma

En esta tercera versión, al sistema reloj/calendario de tiempo real se le agregará una alarma, este en un incremento que requiere ajustes en hardware y software.

Planteamiento del Problema

La alarma consistirá en un zumbador que se activará cuando un interrptor esté cerrado y exista una coincidencia entre la hora actual (hora y minutos) y la hora de la alarma. De esta forma, el zumbador estará activo durante un minuto o cuando el usuario abra el interruptor de activación. En la Figura 11.15 se muestra el nuevo aspecto que tendrá el sistema.

El estado inicial nuevamente se conserva, el sistema iniciará en el modo 0, mostrando la hora y minutos con los dos puntos parpadeando. La alarma se activará por hardware (interruptor cerrado), por lo que depende del usuario si decide mantenerla activa, la hora de la alarma se almacenará en un arreglo de 4 caracteres e iniciará con el valor de las 12:00 horas.

Se agrega el modo 4, en el que el usuario podrá ver y modificar la hora de la alarma, siguiendo el mismo procedimiento que en los otros modos, con el botón **Mode** se conmuta hasta que se alcance el modo 4 y con el botón **Set** se habilitan los ajustes,

que se pueden realizar con los botones **Up** y **Down**. Después del modo 4, el sistema regresa al modo 0.

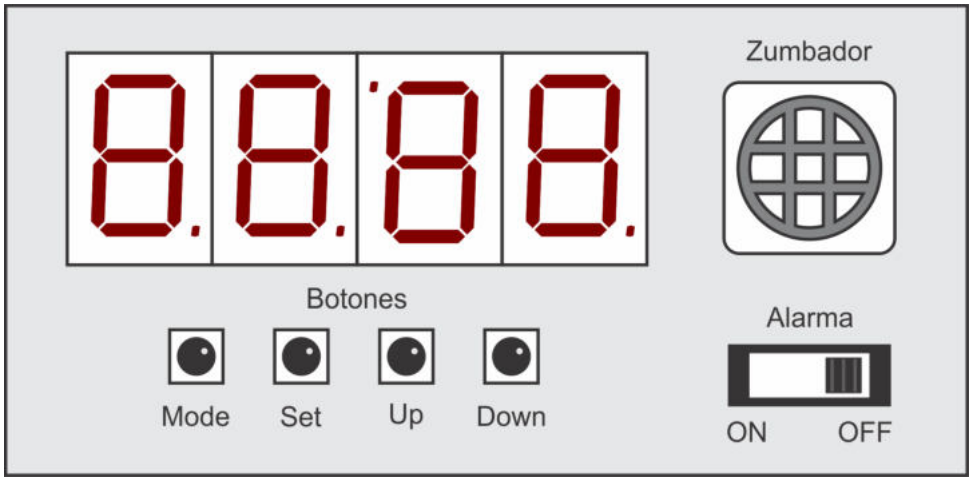


Figura 11.15: Reloj/Calendario con alarma

Requerimientos de Hardware y Software

Con respecto al hardware, se deben agregar los siguientes elementos:

- 1 interruptor, para activar la alarma.
- 1 zumbador con un transistor BC547 y un resistor de 330 Ω, para habilitar su disparo.

No se requiere de nuevas bibliotecas de software y se siguen empleando los mismos recursos internos: interrupciones externas, interrupción por cambios en PD0 y PD1, y el temporizador 1.

Diseño del Hardware

El interruptor para activar la alarma se ubicará en PD4 y el zumbador se conectará en PD5, el hardware resultante se muestra en la Figura 11.16.

Diseño del Software

Al incorporar una alarma al sistema se está agregando un nuevo modo de operación, pero se mantiene la estructura del programa principal y las 4 ISRs.

El programa debe incluir un arreglo global de 4 bytes para almacenar la hora de la alarma (hora y minutos), esta se iniciará con las 12:00 horas y es el único elemento a incorporar en la etarpa de difiniciones y variables globales.

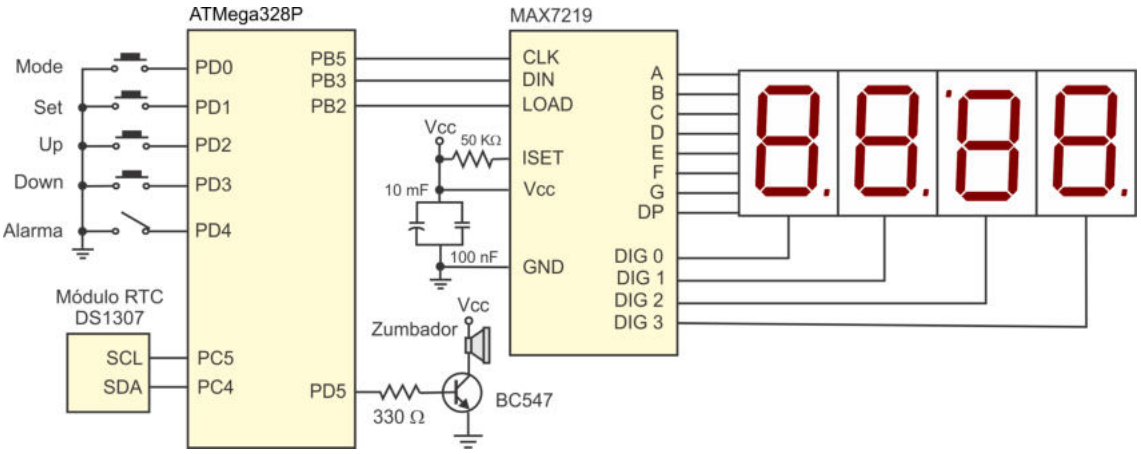


Figura 11.16: Deseño del hardware de un reloj/calendario con alarma

En el programa principal solo se debe agregar una instrucción para asegurar que el zumbador iniciará apagado, no hay otros cambios con este incremento.

Las interrupciones externas son el medio para ajustar la hora de la alarma, tomando como base el diagrama de la Figura 11.12, se debe considerar un nuevo caso, cuando la variable `mode` tiene 4, si `set = 1` se incrementará la hora y si `set = 2` se incrementarán los minutos. La Figura 11.12 no se actualiza porque su estructura se conserva.

Con respecto a la interrupción por cambios en PD0 y PD1, que atiende a los botones **Mode** y **Set**, partiendo de la Figura 11.13, se debe tener en cuenta que cuando **Mode** se presiona, el nuevo valor máximo para la variable `mode` será 4. Si es **Set** el botón que se presiona, el comportamiento es muy parecido, solo se debe condicionar la lectura y escritura de la fecha, debe ser exclusiva para los modos 2 y 3, porque ahora también existe el modo 4, en el que no se tiene acceso al módulo RTC. Tampoco es necesario actualizar la Figura 11.13, los cambios requeridos son mínimos y de fácil comprensión.

En cuanto al temporizador 1, si se observa la Figura 11.14, se notará que básicamente se selecciona la información a mostrar, dependiendo de las variables `mode` y `set`, esta parte se puede expandir para incluir el nuevo modo.

Lo que faltaría agregar al sistema es la revisión de la alarma, hay dos condiciones para que se active el zumbador, estas se van a revisar periódicamente, por lo que también se hará en la ISR del temporizador 1. El estado del zumbador se establece después de que se determine qué información se va a mostrar, como se puede ver en la Figura 11.17.

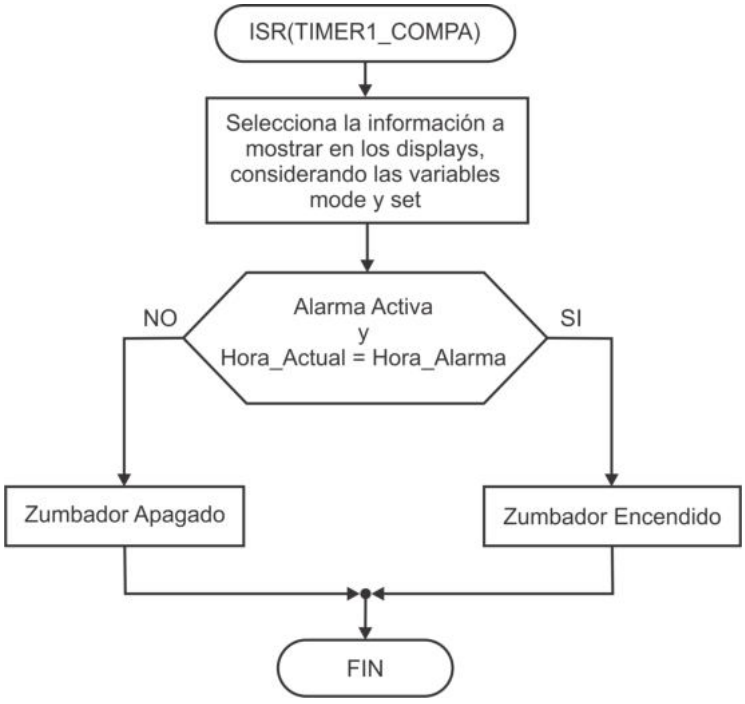


Figura 11.17: Manejo de la alarma con el temporizador 1

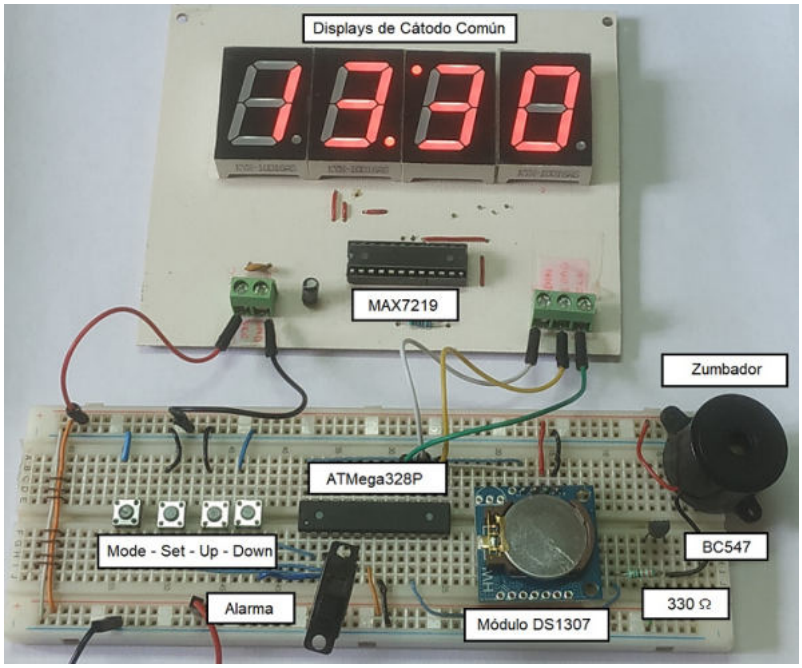


Figura 11.18: Implementación del hardware del incremento 3

Implementación del Hardware

En este incremento se agregaron algunos elementos de hardware, con respecto al incremento 2. En la Figura 11.16 se muestra el diseño del hardware y la realización física o implementación se presenta en la Figura 11.18.

Implementación del Software

La visualización y configuración de la alarma se realiza de manera similar a la de los otros parámetros, solo que en este caso, la variable `mode` debe tener 4 y la variable `set` debe estar en 1 o 2, para configurar la hora y los minutos, respectivamente. No es necesario repetir el código que se requiere para ello, lo que hace diferente a este incremento es la revisión y activación de la alarma, esto se hace dentro de la ISR del temporizador 1, al final de la misma, por ello, la ISR queda estructurada de la siguiente manera:

```
ISR(TIMER1_COMPA_vect) {
   dp = (dp == 0)? 1: 0;
   switch(mode) {
      case 0:
               . . .      // Muestra hora y minutos
      case 1:
               . . .      // Muestra segundos
      case 2:
               . . .      // Muestra día y mes
      case 3:
               . . .      // Muestra año
      case 4:
                . . .     // Muestra hora y minutos de la alarma
   }
   MAX7219_Datos();
   if(!(PIND & 0x10) &&((hora_act[0] == alarma[0])    // Revisa
                     && (hora_act[1] == alarma[1])    // la alarma
                     && (hora_act[2] == alarma[2])
                     && (hora_act[3] == alarma[3])))
      PORTD |= 0x20;          // Zumbador encendido
   else
      PORTD &= 0xDF;          // Zumbador apagado
}
```

Integración y Evaluación

El programa se compiló y el archivo HEX se descargó en el microcontrolador ATMega328P. Se probó el sistema en el hardware actualizado y funcionó como se esperaba, sin embargo, se apreciaron 2 detalles de funcionalidad que pueden mejorarse, estos se describen en la sección de ajustes y correcciones.

Ajustes y Correcciones

Se realizaron 2 ajustes al sistema para mejorar su funcionalidad, el primero fue con respecto a la activación de la alarma. Puesto que solo se realizaba el encendido del zumbador, se generaba un tono continuo que podía tardar hasta 1 minuto. Un tono continuo puede resultar estresante, por lo que se cambió por un tono interrumpido, buscando que el zumbador trabaje durante un segundo y esté apagado por medio segundo. Puesto que el temporizador 1 interrumpe cada medio segundo, básicamente se agregó un contador de la alarma como variable global (`cnt_alm`), así, cuando la alarma debe estar activa, esta se enciende solo si el contador es menor que 2. El ajuste en el código queda de la siguiente manera:

```
ISR(TIMER1_COMPA_vect) {

   . . .        // Selecciona y muesta los datos que corresponden
                // con las variables mode y set

   if(!(PIND & 0x10) &&((hora_act[0] == alarma[0])
                     && (hora_act[1] == alarma[1])
                     && (hora_act[2] == alarma[2])
                     && (hora_act[3] == alarma[3])) && cnt_alm < 2) {
      cnt_alm++;
      PORTD |= 0x20;          // Zumbador encendido
  }
  else {
      PORTD &= 0xDF;          // Zumbador apagado
          cnt_alm = 0;
  }
}
```

El otro ajuste fue en el espacio de almacenamiento de la hora de la alarma, originalmente solo estaba en RAM, por lo que su valor se perdía en ausencia de energía, esto contrasta con la hora y fecha, que se conservan porque el RTC cuenta con una pila de respaldo. Por ello, se optó por almacenar la hora de la alarma en EEPROM, los cambios que se agregan al código son:

```
// Debe incluirse la biblioteca de la EEPROM
#include <avr/eeprom.h>
 . . .

// Arreglo con la alarma inicial
EEMEM uint8_t  alarma_EEPROM[4] = { 1, 2, 0, 0};
 . . .

// En el main se lee la hora de la alarma
   for(uint8_t i = 0; i < 4; i++)
      alarma[i] = eeprom_read_byte(&alarma_EEPROM[i]);
 . . .
```

```
// En la ISR de la PCINT2 se actualiza la nueva alarma en EEPROM
// cuando el botón Set es presionado
   else if( . . .  || set == 2) {
      EIMSK = 0x00;           // Desactiva interrupciones externas
      set = 0;                // set tendrá 0 nuevamente
      . . .
          else if(mode == 4)      // Respalda en EEPROM
         for(uint8_t  i = 0; i < 4; i++)
            eeprom_write_byte(&alarma_EEPROM[i], alarma[i]);
   }
 . . .
```

En la metodología de 8 pasos se contemplan las correcciones, sin embargo, dado que se está trabajando con un modelo incremental de diseño, puede decirse que cada nuevo incremento es una corrección de la versión previa.

11.3.4. Incremento 4: Reloj/Calendario con Alarma y Temporizador

En el incremento 4 se le va a agregar al sistema la funcionalidad de un temporizador, un modo de operación que permitirá al usuario establecer un periodo de tiempo para una determinada tarea. El sistema debe mostrar los minutos y segundos restantes, y señalizar el final del periodo mediante una alarma audible.

Planteamiento del Problema

Los temporizadores son ampliamente usados en la vida ordinaria, en tareas que requieren el manejo de intervalor de tiempo precisos, como gastronomía, hornos industriales, revelado fotográfico, etc. Es factible agregar esta característica al sistema porque tiene los elementos necesarios para ello: una base de tiempo proporcionada por el módulo RTC, displays para mostrar el tiempo restante, botones de configuración y un zumbador para notificar el fin del periodo.

Puesto que el sistema tiene 4 displays y por la importancia de observar el tiempo restante, el temporizador mostrará los minutos y segundos, y manejará un periodo máximo de una hora, cuando inicie con los 4 dígitos en 0.

El temporizador se incluirá como el modo de operación 5, el sistema con el temporizador mantendrá el aspecto mostrado en la Figura 11.15. Con el botón **Mode** se llevará al sistema a este modo y con el botón **Set** se podrán configurar los minutos y segundos, empleando los botones **Up** y **Down**. Su configuración seguirá el procedimiento utilizado con los parámetros de los otros modos de operación. Sin embargo, una vez que se han establecido los minutos y segundos requeridos por el usuario, se pretende utilizar al botón **Up** para dar inicio o pausar al temporizador, este será el único modo en que se habilita al botón **Up** estando la variable `set` con 0, porque de esta forma se evita agregar un botón adicional al sistema.

Requerimientos de Hardware y Software

El hardware que se ha alcanzado hasta el incremento 3 es suficiente para agregar el modo temporizador, por lo que no hay requerimientos de hardware adicionales.

Con respecto al software, tampoco es necesario agregar nuevas bibliotecas, de hecho, se mantiene la estructura hasta el momento empleada, consistente en un programa principal y 4 ISRs, para las interrupciones externas, la interrupción por cambios en PD0 y PD1, y coincidencia por comparación en el temporizador 1.

Diseño del Hardware

No hay cambios en el diseño del hardware, el diagrama mostrado en la Figura 11.16, elaborado para el incremento 3, se mantiene en este incremento 4.

Diseño del Software

En este 4^o incremento se le va a agregar al sistema la función de temporizador, se ha observado que el hardware se mantiene y que solo se requiere de adecuaciones en software. De manera general, el código a agregar se puede separar en dos tareas, la configuración del temporizador y la operación del mismo. Es decir, la primera tarea consiste en establecer los minutos y segundos requeridos por el usuario y, la segunda, en esperar a que el usuario presione el botón de arranque, se lleve el decremento con exhibición en los displays y se señalice el final del periodo con el apoyo del zumbador. Puesto que la primera tarea es muy parecida a la configuración de los parámetros en otros modos de operación, esta sección de diseño del software se va a centrar en la segunda tarea.

No es posible hacer un diagrama de flujo con las acciones para la operación del temporizador, porque se reparten en las diferentes ISRs, en el programa principal no se realizan acciones relacionadas con este modo de operación.

Es conveniente incluir 2 banderas para definir el estado del temporizador: `tempo_act` para indicar que el temporizador está activo y `tempo_fin` con la que se señaliza que el tiempo del temporizador ha concluido. Ambas deben iniciar con 0 y se van a modificar o evaluar en las ISRs.

Además, el avance del temporizador se debe exhibir en forma descendente. Considerando que la base de tiempo es un RTC, para determinar que ha transcurrido un segundo, se debe comparar el segundo actual con el segundo anterior, el cual se va a almacenar en una variable global, a la que se denomina `seg_ant`.

La operación del temporizador se puede separar en 3 etapas, que se sincronizan con el apoyo de las variables globales, estas etapas son:

- **Arranque:** Cuando en la ISR de la PCINT2 se detecte que el sistema está en el modo 5 y que la variable `set` tiene 0, se debe habilitar la INT0 para que

sea posible detectar el evento para el arranque del temporizador. En la ISR de la INT0, para `mode = 5`, se agrega el caso de `set = 0`, si se detecta esta situación, se pone en alto a la bandera `tempo_act`. Si ocurre el evento pero `tempo_act` ya estaba en alto, significa que el temporizador está activo y se está solicitando una pausa, esta se consigue poniendo la bandera `tempo_act` en bajo. Cuando la bandera `tempo_act` se pone en alto también se almacena el valor del segundo actual en la variable `seg_ant`.

- **Avance:** El avance del temporizador se debe mostrar periódicamente, para ello, el código necesario se va a agregar en la ISR del temporizador 1, cuando `mode = 5` y `set = 0`, se debe evaluar la bandera `tempo_act`, si está en alto, significa que el temporizador está activo y se debe observar si hay diferencia entre el segundo actual y el anterior (`seg_ant`), en caso de que sean diferentes, debe haber un decremento en el arreglo con el valor del temporizador y actualizar el segundo anterior, también se debe observar si el temporizador ha llegado a cero, para limpiar la bandera `tempo_act` y poner en alto a `tempo_fin`. El segundo actual y el anterior pueden ser iguales porque el temporizador 1 desborda cada medio segundo.

  Durante el avance, es conveniente ignorar los botones **Mode** y **Set**, para no cambiar el modo o pasar a la configuración mientras el temporizador esté activo. Para ello, se deben agregar condiciones en la ISR de la PCINT2.

- **Terminación:** Se detecta cuando la bandera `tempo_fin` está en alto, esta bandera se va a sondear periódicamente, al final de la ISR del temporizador 1, el zumbador se pondrá en alto pero no por tiempo indefinido, se utilizará un contador de interrupciones que llegará hasta 15. El zumbador se apagará si el contador es múltiplo de 3, de esta forma, se van a generar 5 pulsos con una duración de 1 segundo y el tono no será monótono. Después de ello, se limpia la bandera `tempo_fin`, concluyendo la operación del sistema en modo temporizador. Para dar un efecto visual, durante la generación del tono final, los displays se pueden apagar cuando el zumbador está apagado.

Puede verse que los recursos del MCU empleados hasta el incremento 3 están involucrados en el modo temporizador, el incremento 4 solo requiere que se agreguen instrucciones en lugares muy precisos.

Implementación del Hardware

El hardware para el incremento 4 es el mismo que el del incremento 3, el cual se mostró en la Figura 11.18.

Implementación del Software

Durante el diseño del software se hizo la separación en dos tareas para el modo temporizador: configuración y operación. El diseño del software se centró en la se-

gunda tarea, la cual se separó en 3 estapas, en cada una de ellas se especificaron las acciones que se deben realizar.

En esta sección se implementará el código, se muestra la estructura de las ISRs pero solo se incluyen las instrucciones relacionadas con la operación del temporizador. Los datos globales que se incorporan en este incremento 4 son:

```
uint8_t  temporizador[4] = {0};          // Almacenamiento
uint8_t  tempo_act = 0, tempo_fin = 0; // Banderas de estado
uint8_t  seg_ant, cnt_tmp = 0;           // Variables de apoyo
void  dec_tempo();                       // Decrementa al temporizador
```

La función `dec_tempo()` decrementa la información del arreglo `temporizador[]` y evalúa si se obtuvo el valor de 0 en los 4 elementos, de ser así, pondrá en alto a la bandera `tempo_fin`.

Puesto que las banderas y variables globales se inicializan durante su declaración, no es necesario incorporar código adicional a la función `main()`, se mantiene sin cambios, con respecto al incremento 3.

En la ISR de la INT0 se realiza el arranque del temporizador, el código relacionado con esa tarea es:

```
ISR(INT0_vect) {
. . .

   switch(mode) {
      case 0:
      . . .          // Código para configurar otros parámetros
      case 5:
         if(set == 0) {
            if(tempo_act == 0) {          // No activo
               lee_hora();
               seg_ant = hora_act[5];     // Respalda el segundo
               tempo_act = 1;             // anterior y se activa
            } else
               tempo_act = 0;             // Pausa
         } else if(set == 1) {
            . . .   // Configura los minutos del temporizador

         } else {
            . . .   // Configura los segundos del temporizador
         }
      break;
   }
}
```

La INT1 se utiliza durante la configuración del valor del temporizador (decrementando minutos y segundos), el código no se incluye porque es muy similar a la configuración de otros parámetros.

En la ISR de la PCINT2 se agregaron diferentes acciones, dependiendo del botón presionado y del estado de las variables `mode` y `set`. El código completo de esta ISR es:

```
ISR(PCINT2_vect) {

   if(!(PIND & 0x01)) {                // Botón Mode
      if(set == 0 && tempo_act == 0) {
         mode = (mode < 5)? mode + 1: 0;
         if(mode == 5)
            EIMSK = 0x01;
      }
   }
   else if(!(PIND & 0x02)) {           // Botón Set
      if(set == 0) {
         if(mode != 5 || tempo_act == 0) {
            EIMSK = 0x03;
            set = 1;
            if(mode == 0 || mode == 1)
               lee_hora();
            else if(mode == 2 || mode == 3)
               lee_fecha();
         }
      }
      else if(mode == 1 || mode == 3 || set == 2) {
         EIMSK = 0x00;
         set = 0;
         if(mode == 0 || mode == 1)
            esc_hora();
         else if(mode == 2 || mode == 3)
            esc_fecha();
         else if(mode == 4)
            for(uint8_t  i = 0; i < 4; i++)
               eeprom_write_byte(&alarma_EEPROM[i], alarma[i]);
         else if(mode == 5)
            EIMSK = 0x01;
      } else
         set = 2;
   }
}
```

Analizando el código, con respecto al botón **Mode** se observa que, no se puede cambiar de modo si el temporizador está avanzando y que se activa la INT0 cuando se ha alcanzado el modo 5.

Para el botón **Set** se puede ver que no se puede hacer el paso a la configuración si el temporizador está avanzando y que si la variable `set` nuevamente regresó al valor de 0, se va a activar la INT0 si el sistema está en el modo 5.

En la ISR del temporizador 1 se realizan las etapas de avance y terminación del sistema en el modo temporizador, el código relacionado con estas etapas es:

```
ISR(TIMER1_COMPA_vect) {

   dp = (dp == 0)? 1: 0;
   switch(mode) {
      case 0:
         . . .          // Ubica la información a mostrar
                        // según las variables mode y set

      case 5:           // El modo 5 es el modo temporizador
         switch(set) {
            case 0:
               if(tempo_act == 1) {      // Temporizador activo
                  lee_hora();
                  if(seg_ant != hora_act[5]) {
                     dec_tempo();        // Avanza
                     seg_ant = hora_act[5];
                  }
               }
               displays[0] = temporizador[0];
               displays[1] = temporizador[1];
               displays[2] = pgm_read_byte(&Tabla[temporizador[2]]);
               displays[3] = temporizador[3];
            break;
            . . .       // Otros casos de set para mostrar la
                        // configuración del temporizador
         }
      break;
   }

   . . .                            // Código para activar la alarma

   if(tempo_fin == 1) {             // Fin del periodo
      PORTD |= 0x20;                // Zumbador encendido
      cnt_tmp++;
      if(cnt_tmp % 3 == 0) {
         PORTD &= 0xDF;             // Zumbador apagado
         displays[0] = 0x0F;
         displays[1] = 0x0F;
         displays[2] = 0x00;
         displays[3] = 0x0F;
      }
      if(cnt_tmp == 15)
         tempo_fin = 0;
   }
   MAX7219_Datos();
}
```

En la función `dec_tempo()` se activará la bandera `tempo_fin`, cuando el temporizador llegue a cero, también se limpia la bandera `tempo_act` para que el temporizador ya no siga decrementando.

La revisión de la bandera `tempo_fin` se hace al final de la ISR, ahí se emplea al contador `cnt_tmp` para limitar el tiempo de la generación del tono, hacer que el tono sea interrumpido y exhibir un parpadeo en los displays. El contador `cnt_tmp` se reinicia en la función `dec_tempo()`, cuando el temporizador llega a cero. La llamada a la función `MAX7219_Datos()` es la última instrucción en la ISR, para que el sistema muestre la información más reciente del arreglo `displays[]`.

Integración y Evaluación

Después de compilar el programa y programar al ATMega328P, se evalúo el sistema observando un desempeño favorable, el sistema cumplió con los requerimientos establecidos en el incremento 4.

Ajustes y Correcciones

Dado que el sistema funcionó como se esperaba, no fue necesario realizar ajustes o correcciones.

11.3.5. Incremento 5: Reloj/Calendario con Alarma, Temporizador y Termómetro

Este es el último incremento, se le incorporará al sistema la capacidad de mostrar la temperatura ambiente en grados centígrados, con ello, además de las funciones digitales hasta el momento conseguidas, el sistema estará mostrando un parámtro analógico.

Planteamiento del Problema

La exhibición de la temperuatura es un complemento que se le va a agregar al sistema embebido. El resultado puede tomarse como punto de partida para la realización de un sistema que mantenga la temperatura ambiente en un rango establecido, con el apoyo de actuadores.

El sistema requiere de un sensor de temperatura, con su etapa de acondicionamiento, se pretende emplear al LM35, el cual entrega 10 mV/^oC. Es un sensor muy popular y de bajo costo.

En la Figura 11.15 se mostró el aspecto del sistema en el incremento 4, este se mantiene sin cambios porque el hardware que se agregará para el incremento 5 es interno. Dado que la temperatura es un parámetro de exhibición, no es necesario agregar un nuevo modo de operación, en el modo 0 el sistema deberá mostrar alternadamente la hora actual y la temperatura, con un periodo de 3 segundos para conmutar entre una u otra información. Esto siempre que no esté activa una configuración, es decir, `mode = 0` y `set = 0`.

Requerimientos de Hardware y Software

Los nuevos elementos de hardware que se deben agregar al sistema son:

- 1 sensor de temperatura LM35.
- 1 amplificador operacional LM358P, porque solo requiere de una fuente.
- 2 mini-potenciómetros de precisión, para ajustar sus valores, de acuerdo con la ganancia requerida.

Además de los recursos internos del ATMega328P, hasta el momento empleados, se utilizará su ADC, con un voltaje de referencia de 5 V y no se requiere de nuevas bibliotecas de software.

Diseño del Hardware

En la Figura 11.19 se muestra el sistema resultante, después de agregar al sensor de temperatura con sus etapa de acondicionamiento.

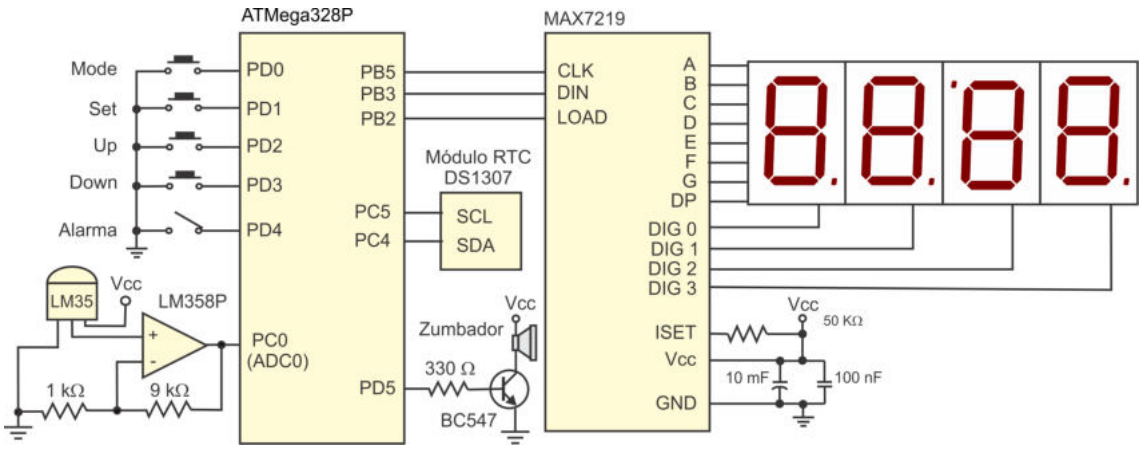


Figura 11.19: Diseño del hardware del incremento 5

Diseño del Software

La función de termómetro se agrega como un complemento al modo 0 y es una función de exhibición, es decir, no hay nada por configurar mientras se muestra la temperatura. Por lo que la estructura del programa se mantiene, solo hay que agregar algunos elementos en diferentes secciones, estas son:

- **Varaiables globales:** Se requiere de un arreglo de 2 bytes para almacenar la temperatura actual y de un contador para determinar si se exhibe la hora o la temperatura.
- **Programa principal:** En el *main* se debe configurar al ADC, iniciando una nueva conversión y habilitando la interrupción por fin de conversión.

- **ISR del ADC:** Cuando termina una conversión, el resultado queda en el registro `ADCW`, a partir de este valor se debe obtener la temperatura y colocar su valor entero en el arreglo global de 2 bytes.
- **ISR del temporizador 1:** El temporizador 1 también proporciona la base de tiempo en este incremento, en cada interrupción se inicia una nueva conversión para mantener la información actualizada y se determina si se mostrará la hora actual o la temperatura, considerando que cada 3 segundos se cambiará el parámetro. El contador global es utilizado con este propósito, si tiene un valor menor a 6, se muestra la hora, en caso contrario se muestra la temperatura. El contador se incrementa en cada interrupción y se reinicia cuando llega a 11, son 6 eventos por parámetro, considerando que se generan cada medio segundo.

La descripción textual de cada una de las tareas es suficiente para el desarrollo del software, no es necesario realizar diagramas de flujo.

Implementación del Hardware

Tomando como base el diseño del hardware (Figura 11.19), se implementó una tarjeta PCB con el microcontrolador y todos los demás dispositivos: el zumbador, el módulo RTC y el sensor con su etapa de acondicionamiento. En la Figura 11.20 se muestra esta tarjeta, en la que se pueden ver los conectores para los botones, el interruptor de la alarma y hacia la tarjeta con los displays y el MAX7219.

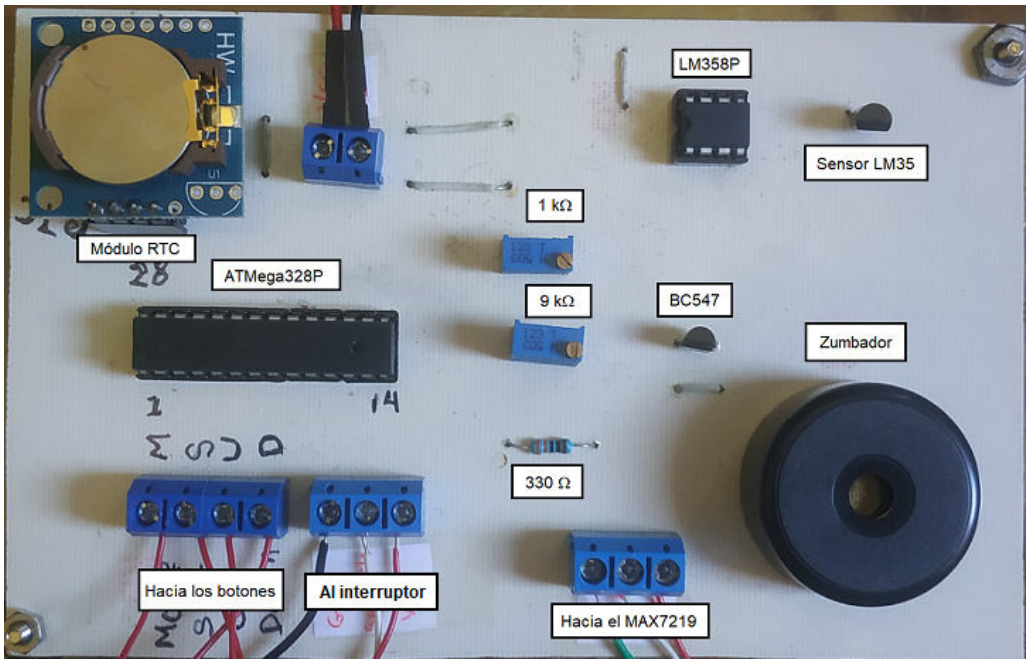


Figura 11.20: Implementación del hardware del incremento 5

Implementación del Software

En el diseño del software se indicaron 3 secciones en donde se debe agregar código para incluir el termómetro en el sistema. En esta sección se muestra el código en cada una de esas secciones.

Los datos globales que se incorporan en este incremento 5 son:

```
const uint8_t Tabla3[] PROGMEM ={0x7E, 0x30, 0x6D, 0x79, 0x33,
                                 0x5B, 0x5F, 0x70, 0X7F, 0X73}; // 0-9

uint8_t  temp_act[2];     // Almacena la temperatura actual
uint8_t  cnt_temp = 0;    // Contador para la exhibición
```

Los displays 0, 1 y 3 hasta el momento solo manejaban información numérica, por ello, utilizaban el decodificador de 7 segmentos incluido en el MAX7219. En este incremento, en el display 3 se va a mostrar la letra C, para indicar que la temperatura está en grados centigrados, esto significa que el display 3 dejará de usar al decodificador y sus códigos en 7 segmentos deben manejarse por software, es por eso que en la parte de datos globales se incorpora una nueva tabla de constantes.

En el programa principal se configura al ADC con el siguiente código:

```
int main(void) {
   . . .              // Otras configuraciones

   ADMUX=0x00;        // Se utilizará el canal 0
   ADCSRA = 0xCB;     // Habilita ADC con interrupción e inicia
                      // una conversión, preescala de 8
   DIDR0=0x01;        // Anula el buffer digital en ADC0

   . . .              // Código adicional
}
```

Para la ISR del ADC se debe considerar que la temperatura máxima será de 50 ^oC, cuando esta se produzca, en el registro `ADCW` se tendrá el valor de 1023, por ello, se obtiene un factor de escalamiento de 0.04882, el código de esta ISR es:

```
ISR(ADC_vect){
float    temp;
uint8_t  temp_ent;

   temp=ADCW*0.04882;            // Calcula la temperatura
   temp_ent = temp;              // Parte entera de la temperatura
   temp_act[0] = temp_ent/10;    // Ubica en el arreglo para mostrar
   temp_act[1] = temp_ent%10;    // en los displays
}
```

El código que se agrega a la ISR del temporizador 1 para cubrir con los requerimientos establecidos en la etapa de diseño es:

```
ISR(TIMER1_COMPA_vect) {
   dp = (dp == 0)? 1: 0;
   ADCSRA |= 1<<ADSC;        //inicia nueva convesión de temperatura
   switch(mode) {
      case 0:
         switch(set) {
            case 0:
               if(cnt_temp < 6) {
                  lee_hora();
                  displays[0] = hora_act[0];
                  displays[1] = hora_act[1];
                  displays[2] = pgm_read_byte(&Tabla2[hora_act[2]]);
                  displays[3] = pgm_read_byte(&Tabla3[hora_act[3]]);
                  if(dp == 1) {
                     displays[1] |= 0x80;
                     displays[2] |= 0x80;
                  }
               } else {
                  displays[0] = temp_act[0];
                  displays[1] = temp_act[1];
                  displays[2] = 0x1D;      // Símbolo de grados
                  displays[3] = 0x4E;      // Letra C
               }
               cnt_temp = (cnt_temp < 11)? cnt_temp + 1: 0;
            break;

            . . .          // Código para mostrar la configuración
                           // en el modo 0
         }
         . . .             // Código para mostrar otros parámetros
        }

   . . .                  // Otras acciones antes de exhibir los datos
   MAX7219_Datos();       // Coloca los datos en los displays
}
```

Se observa en el código que se está utilizando la `Tabla3[]` para el display 3 y la `Tabla2[]` para el display 2, la última ya se usaba, por la posición invertida del display 2, solo se renombró para distinguirla de la nueva, asociándola con su respectivo display. El renombre y las asignaciones para el display 3 deben actualizarse en los otros parámetros.

Integración y Evaluación

El programa compilado se descargó en el microcontrolador ATMega328P, se unieron las dos placas PCB desarrolladas, con los botones e interruptor y se evalúo el sistema completo, el cual se muestra en la Figura 11.21. El sistema exhibió un desempeño favorable, cumpliendo con las especificaciones de cada uno de los incrementos.

Figura 11.21: Sistema embebido multifunción

Las tarjetas se colocaron en una base de acrílico transparente para que se puedan apreciar los diferentes componentes del sistema, aunque es un sistema funcional, su implementación queda disponible para explicar el uso de recursos en los cursos de microcontroladores.

Ajustes y Correcciones

El sistema funcionó como se esperaba, sin embargo, una vez que se colocó en la base de acrílico, al emplear botones diferentes a los utilizados durante las pruebas, se observaron muchos rebotes en los botones, por ello, se agregaron sentencias de código en las ISRs de los botones para eliminar los rebotes y hubo una reducción significativa, aunque no al 100 %.

Otro ajuste que se realizó fue la incorporación de la posibilidad de modificar el brillo de los displays, el MAX7219 facilita esta operación, en su registro de configuración 0x0A se debe escribir un número entre 0x00 y 0x0F, que define el nivel de brillo, ajustando el ciclo de trabajo de una señal PWM. Por lo tanto, se agregó la variable `brillo` para almacenar su valor, el cual inició con 0x07, también el código necesario para que el brillo se pudiera modificar con las interrupciones externas (botones **Up** y **Down**), solo cuando el sistema esté en el modo 0 y sin configuración (`mode = 0` y `set = 0`). Este ajuste implica que las interrupciones externas deben iniciar habilitadas y así se deben mantener cuando las variables `mode` y `set` tengan 0. Los cambios se hicieron en pasos de ±2, según la interrupción.

Además de los ajustes realizados, no fue necesario realizar alguna corrección, el sistema embebido multifunción trabaja adecuadamente, cumpliendo con las funcionalidades de reloj, calendario, alarma, temporizador y termómetro.

11.4. Sistemas Propuestos

En la Sección 11.1 se propuso una metodología y en las Secciones 11.2 y 11.3 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 el microcontrolador ATMega328P, 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 analógicos, el sistema digitaliza una señal de voz y almacena la información en la SRAM, esto se realiza cuando se presiona 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 tiene un teclado matricial y un LCD, el usuario escribe una cadena y el MCU genera los tonos en una bocina cuando se presiona un botón para enviar. El receptor tiene un micrófono y un LCD, su función consiste en la captura de los tonos, decodificación de los caracteres y exhibición del mensaje recibido 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 solo 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 presenta al nuevo número y envía un mensaje para que el usuario repita la secuencia completa, haciendo eco en el LCD), así sucesivamente hasta alcanzar los 10 números. Con los 10 números correctos se cubre un nivel y se pasa a un 2^o nivel en donde los números son de 2 dígitos. Es posible considerar un 3^{er} nivel con números 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 activo. Los números aleatorios se pueden generar con uno de los temporizadores de 8 bits, avanzando en carrera libre y leyendo su valor cuando el usuario presione una tecla, el número es aleatorio porque el temporizador avanza a una velocidad muy alta y se desconoce el momento en que el usuario presionará una tecla.

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 2^o nivel, en cada paso se van a encender 2 LEDs cuyas posiciones debe repetir el usario con el teclado. Cuando se presione una tecla, se debe reflejar su posición en la matriz de LEDs. Pueden incluirse LEDs adicionales para indicar el estado del juego, o si prefiere, un LCD para dar instrucciones al usuario.

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 2 registros por día (entrada y salida). Los usuarios se identifican con una clave de 3 dígitos, que deben introducir para registrar su acceso. La información se acumula en la memoria del sistema hasta que es solicitada serialmente desde una computadora. Las claves de los usuarios se deberán definir de manera seial, estas deben almacenarse en EEPROM para que se conserven en ausencia de energía. La hora y fecha del sistema tambíen se podrán configurar desde el puerto serie.

8. **Teclado musical, con dos demos:** 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, que básicamente es una combinación de frecuencias y periodos de tiempo. Además, el sistema debe contar con 2 botones para grabar y reproducir tonos generados por el usuario, con un botón se inicia y termina la grabación, y con el otro se realiza su reproducción. Es conveniente incluir LEDs para reflejar el estado del sistema.

9. **Control de temperatura ambiental:** En estre proyecto se va a acondicionar un ventilador comercial de 3 velocidades, para transformarlo en un sistema embebido con un ATMega328P, un sensor de temperatura, un teclado, y un LCD, los interruptores de las velocidades se remplazan con relevadores. El sistema muestra en el LCD la temperatura actual y el usuario introduce, a través del teclado, la temperatura que desea para el ambiente. Si la temperatura actual está por encima de lo deseado, el sistema enciende al ventilador, con la velocidad en función de la diferencia entre estas temperaturas.

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 enciende 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 3 segundos, para evitar oscilaciones, dado que el control remoto envía la información en repetidas ocasiones.

13. **Móvil manipulado en forma inalámbrica:** Es necesario construir un móvil con 2 motores independientes, contando con la posibilidad de avanzar, retroceder, girar a la izquierda o a la derecha. Se puede comandar desde un control remoto comercial o desde un teléfono inteligente. En el primer caso, el MCU decodifica las señales del control remoto e identifica 5 de ellas, para comandar al móvil y que este 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. Con la segunda opción, al móvil se le debe colocar un módulo receptor por *bluetooth* y se debe desarrollar una aplicación para el teléfono inteligente o buscar alguna versión disponible en la *AppStore*. La recepción no se complica porque los módulos *bluetooth* presentan una interfaz compatible con la USART de los microcontroladores, para el MCU equivale a recibir datos seriales.

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 8 x 8 LEDs:** El sistema cuenta con una matriz de 8 x 8 LEDs, para su conexión con el MCU se pueden emplear dos de sus puertos, o bien, registros de desplazamiento como el MC74HC595A o manejadores de displays como el MAX7219, para reducir el número de terminales. En el ATMega328P se tendrán 10 matrices binarias de información y se irán mostrando una a una, en diversos instantes de tiempo. Para que la animación se genere, los contenidos de las matrices deben estar relacionados entre sí. El sistema es versátil, dado que el contenido de las matrices puede modificarse serialmente desde una PC y con un par de botones se puede incrementar o reducir 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 8 x 8 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 binarios para los diferentes caracteres alfanuméricos. Cada matriz se manejará con un CI MAX7219 y se hará una conexión en cascada para las 4 matrices. La información a mostrar se puede colocar en un arreglo de 32 bytes, en el que se harán desplazamientos para que aparente que el mensaje se desplaza.

17. **Alarma de 5 zonas con 4 claves de acceso:** Sistema de seguridad que inicialmente solo puede ser operado con una clave de acceso de 4 dígitos. Esta clave es para el administrador y con ella puede dar de alta o baja a otras 3 claves (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 un 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, esta se apaga al introducir cualquier clave, del administrador o de los usuarios. Los usuarios solo 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 8 displays de 7 segmentos para mostrar: el tiempo restante del juego (4 displays), el marcador (1 display por equipo) y el número de faltas de cada equipo (1 display por equipo). Además, debe incluir 4 LEDs para mostrar el periodo activo y un zumbador para indicar la conclusión de un periodo, así como un teclado para ajustes iniciales y para el control durante un partido. Por la cantidad de displays a manejar, es conveniente emplear el CI MAX7219 o dos microcontroladores ATMega328P.

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 LEDs de estado para indicar al ganador o si el juego terminó empatado.

20. **Chapa electrónica con tarjetas telefónicas:** El sistema 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 pasos:** 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 una secuencia de movimientos y el otro para repetir la secuencia grabada. El brazo se puede manipular libremente con los botones de los motores si no hay una grabación en progreso, 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 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.

23. **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 contiene un display de 7 segmentos y dos botones, uno para seleccionar la columna a tirar, con ayuda del display, y el otro para realizar el tiro. Para complementar el sistema, se incluyen LEDs de estado que indican quien ganó el juego.

24. **Lámpara RGB manejada en forma inalámbrica:** La lámpara puede estar basada en un conjunto de LEDs RGB tradicionales o en un arreglo de LEDs de la serie WS2812, los cuales incluyen un controlador para que se puedan manejar con solo una terminal. El ATMega328P debe contar con un módulo receptor por *bluetooth* para recibir comandos. Se debe desarrollar una aplicación para un teléfono inteligente o descargar alguna disponible en la *AppStore*. Desde la aplicación se podrá solicitar el cambio de intensidad de cada uno de los colores RGB en forma independiente, o bien, establecer un color predeterminado.

25. **Exhibición de imágenes pequeñas en una matriz RGB de 16 x 16:** El sistema requiere de una matriz de 16 x 16 LEDs RGB WS2812, los cuales utilizan 3 bytes de información por LED. Una imagen ocupa un total de 768 bytes. En la memoria de código del ATMega328P se almacenarán 4 imágenes previamente procesadas para que se escriban como constantes. En el sistema se mostrará una imagen a la vez, cambiando cada que el usuario presiona un botón.

Apéndice A

Registros I/O

En este apéndice se muestran los Registros I/O, normales o extendidos, para una consulta rápida.

Registros de la CPU

- **SREG:** Registro de estado, contiene al habilitador global de interrupciones (`I`), un bit para transferencias (`T`) y banderas que se actualizan después de alguna operación aritmética o lógica.

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

- **SP:** Apuntador de Pila, es un registro de 12 bits, 4 en la parte alta (`SPH`) y 8 en la parte baja (`SPL`). El registro inicia con el valor 0x08FF, para apuntar a la última localidad de la SRAM.

- **MCUSR:** Registro de estado del MCU, contiene las banderas que indican la causa de un reinicio.

7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
-	-	-	-	WDRF	BORF	EXTRF	PORF

- **CLKPR:** Registro para configurar el preescalador del reloj del sistema y hacer que el MCU trabaje a una frecuencia diferente.

7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
CLKPCE	-	-	-	CLKPS3	CLKPS2	CLKPS1	CLKPS0

- **SMCR:** Registro para configurar y llevar al MCU a un modo de bajo consumo.

7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
-	-	-	-	SM2	SM1	SM0	SE

- **OSCCAL:** Registro que contiene el valor para la calibración del oscilador interno.

- **PRR:** Registro para reducir el consumo de potencia, apagando algunos recursos, sin entrar a algún modo de bajo consumo.

7	6	5	4	3	2	1	0
PRTWI	PRTIM2	PRTIM0	-	PRTIM1	PRSPI	PRUSART0	PRADC

- **WDTCSR:** Registro de control del *Watchdog Timer*, define el modo del WDT y su periodo de desbordamiento.

7	6	5	4	3	2	1	0
WDIF	WDIE	WDP3	WDCE	WDE	WDP2	WDP1	WDP0

- **SPMCSR:** Registro de control y estado del almacenamiento en la memoria de programa. Este registro ayuda a las escrituras en la memoria Flash, con el apoyo de un *buffer* de datos intermedio.

7	6	5	4	3	2	1	0
SPMIE	RWWSB	-	RWWSRE	BLBSET	PGWRT	PGERS	SELFPRGEN

Registros para el acceso a la EEPROM

- **EEDR:** Registro para el manejo de los datos.
- **EEAR:** Registro para el manejo de las direcciones, es un registro de 10 bits, 2 en la parte alta (`EEARH`) y 8 en la parte baja (`EEARL`).
- **EECR:** Registro para el manejo de las señales de control, define el modo de acceso y sincroniza las operaciones de lectura y escritura.

7	6	5	4	3	2	1	0
-	-	EEPM1	EEPM0	EERIE	EEMPE	EEPE	EERE

Registros para el manejo de los Puertos

- **DDRB:** Registro para configurar al Puerto B, con 0's se define qué terminales serán entradas y con 1's se definen las salidas.
- **PORTB:** Registro para escribir en el Puerto B cuando es salida y para configurar sus resistores de *pull-up* si el puerto es entrada.
- **PINB:** Registro para leer de las terminales del Puerto B si es entrada o para conmutar su valor si es salida.
- **DDRC:** Registro para configurar al Puerto C, con 0's se define qué terminales serán entradas y con 1's se definen las salidas.
- **PORTC:** Registro para escribir en el Puerto C cuando es salida y para configurar sus resistores de *pull-up* si el puerto es entrada.
- **PINC:** Registro para leer de las terminales del Puerto C si es entrada o para conmutar su valor si es salida.

- **DDRD:** Registro para configurar al Puerto D, con 0's se define qué terminales serán entradas y con 1's se definen las salidas.
- **PORTD:** Registro para escribir en el Puerto D cuando es salida y para configurar sus resistores de *pull-up* si el puerto es entrada.
- **PIND:** Registro para leer de las terminales del Puerto D si es entrada o para conmutar su valor si es salida.

Registros de las Interrupciones Externas

- **EICRA:** Registro de control, en este registro se configura el evento que va a generar una interrupción externa.

7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
-	-	-	-	ISC11	ISC10	ISC01	ISC00

- **EIMSK:** Registro para habilitar las interrupciones externas.

7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
-	-	-	-	-	-	INT1	INT0

- **EIFR:** Registro con las banderas que indican la ocurrencia de una interrupción externa.

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

Interrupciones por Cambios en las Terminales

- **PCMSK0:** Registro para indicar qué terminales del Puerto B van a generar una interrupción, colocando 1's en las terminales a monitorear.
- **PCMSK1:** Registro para indicar qué terminales del Puerto C van a generar una interrupción, colocando 1's en las terminales a monitorear.
- **PCMSK2:** Registro para indicar qué terminales del Puerto D van a generar una interrupción, colocando 1's en las terminales a monitorear.
- **PCICR:** Registro para habilitar las interrupciones por cambios en las terminales.

7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
-	-	-	-	-	PCIE2	PCIE1	PCIE0

- **PCIFR:** Registro con las banderas que indican la ocurrencia de una interrupción por cambios en las terminales.

7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
-	-	-	-	-	PCIF2	PCIF1	PCIF0

Registro de los Temporizadores

- **GTCCR:** Registro de control general, contiene los bits para reiniciar los preescaladores que son empleados por los temporizadores.

7	6	5	4	3	2	1	0
TSM	-	-	-	-	-	PSRASY	PSRSYNC

Registros del Temporizador 0

- **TCNT0:** Registro del temporizador 0, se incrementa con la señal de reloj o con eventos externos en T0.
- **OCR0A:** Registro A para la comparación del temporizador 0.
- **OCR0B:** Registro B para la comparación del temporizador 0.
- **TCCR0A:** Registro A para la configuración y control del temporizador 0.

7	6	5	4	3	2	1	0
COM0A1	COM0A0	COM0B1	COM0B0	-	-	WGM01	WGM00

- **TCCR0B:** Registro B para la configuración y control del temporizador 0.

7	6	5	4	3	2	1	0
FOC0A	FOC0B	-	-	WGM02	CS02	CS01	CS00

- **TIFR0:** Registro con las banderas que indican la ocurrencia de un evento del temporizador 0.

7	6	5	4	3	2	1	0
-	-	-	-	-	OCF0B	OCF0A	TOV0

- **TIMSK0:** Registro para activar las interrupciones debidas al temporizador 0.

7	6	5	4	3	2	1	0
-	-	-	-	-	OCIE0B	OCIE0A	TOIE0

Registros del Temporizador 1

- **TCNT1:** Registro del temporizador 1, se incrementa con la señal de reloj o con eventos externos en T1. Es un registro de 16 bits, compuesto por `TCNT1H` y `TCNT1L`.
- **ICR1:** Registro para capturar el valor del registro `TCNT1` ante un evento externo. Es un registro de 16 bits, compuesto por `ICR1H` y `ICR1L`.
- **OCR1A:** Registro A para la comparación del temporizador 1. Es un registro de 16 bits, compuesto por `OCR1AH` y `OCR1AL`.
- **OCR1B:** Registro B para la comparación del temporizador 1. Es un registro de 16 bits, compuesto por `OCR1BH` y `OCR1BL`.

- **TCCR1A:** Registro A para la configuración y control del temporizador 1.

7	6	5	4	3	2	1	0
COM1A1	COM1A0	COM1B1	COM1B0	-	-	WGM11	WGM10

- **TCCR1B:** Registro B para la configuración y control del temporizador 1.

7	6	5	4	3	2	1	0
ICNC1	ICES1	-	WGM13	WGM12	CS12	CS11	CS10

- **TCCR1C:** Registro C para la configuración y control del temporizador 1.

7	6	5	4	3	2	1	0
FOC1A	FOC1B	-	-	-	-	-	-

- **TIFR1:** Registro con las banderas que indican la ocurrencia de un evento del temporizador 1.

7	6	5	4	3	2	1	0
-	-	ICF1	-	-	OCF1B	OCF1A	TOV1

- **TIMSK1:** Registro para activar las interrupciones debidas al temporizador 1.

7	6	5	4	3	2	1	0
-	-	ICIE1	-	-	OCIE1B	OCIE1A	TOIE1

Registros del Temporizador 2

- **TCNT2:** Registro del temporizador 2, se incrementa con la señal de reloj interna o externa.
- **OCR2A:** Registro A para la comparación del temporizador 2.
- **OCR2B:** Registro B para la comparación del temporizador 2.
- **TCCR2A:** Registro A para la configuración y control del temporizador 2.

7	6	5	4	3	2	1	0
COM2A1	COM2A0	COM2B1	COM2B0	-	-	WGM21	WGM20

- **TCCR2B:** Registro B para la configuración y control del temporizador 2.

7	6	5	4	3	2	1	0
FOC2A	FOC2B	-	-	WGM22	CS22	CS21	CS20

- **TIFR2:** Registro con las banderas que indican la ocurrencia de un evento del temporizador 2.

7	6	5	4	3	2	1	0
-	-	-	-	-	OCF2B	OCF2A	TOV2

- **TIMSK2:** Registro para activar las interrupciones debidas al temporizador 2.

7	6	5	4	3	2	1	0
-	-	-	-	-	OCIE2B	OCIE2A	TOIE2

- **ASSR:** Registro para habilitar la operación asíncrona del temporizador 2.

7	6	5	4	3	2	1	0
-	EXCLK	AS2	TCN2UB	OCR2AUB	OCR2BUB	TCR2AUB	TCR2BUB

Registros del Convertidor Analógico a Digital

- **ADCW:** Registro de 10 bits en donde queda el resultado de una conversión anaógico a digital. La parte alta queda en `ADCH` y solo se utilizan 2 bits, y la parte baja queda en `ADCL`. Aunque se puede alinear el resultado a la izquierda para dejar los dos bits menos significativos en `ADCL`.
- **ADMUX:** En este registro se define el voltaje de referencia, la alineación del resultado de la conversión y la procedencia del voltaje analógico a convertir.

7	6	5	4	3	2	1	0
REFS1	REFS0	ADLAR	-	MUX3	MUX2	MUX1	MUX0

- **ADCSRA:** Es el registro A para configurar al ADC y conocer su estado.

7	6	5	4	3	2	1	0
ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0

- **ADCSRB:** Es el registro B para configurar al ADC y conocer su estado. Principalmente se define el modo de carrera libre o si habrá un auto disparo.

7	6	5	4	3	2	1	0
-	ACME	-	-	-	ADTS2	ADTS1	ADTS0

- **DIDR0:** Registro para deshabilitar los *buffers* digitales cuando se utilizan entradas analógicas, en las terminales ADC0 a ADC5 (PC0 a PC5).

Registros del Comparador Analógico

- **ACSR:** Registro para configurar al AC y conocer su estado.

7	6	5	4	3	2	1	0
ACD	ACBG	ACO	ACI	ACIE	ACIC	ACIS1	ACIS0

- **DIDR1:** Registro para deshabilitar los *buffers* digitales cuando se utilizan entradas analógicas, en las terminales AIN0 y AIN1 (PD6 y PD7).

Registros de la USART0

- **UDR0:** Registro de datos de la USART0, es el *buffer* para la transmisión y recepción de datos.
- **UBRR0:** Registro para establecer la velocidad de transmisión. Es un registro de 12 bits, los 4 bits más significativos están en `UBRR0H` y los 8 menos significativos en `UBRR0L`.
- **UCSR0A:** Registro A para el control y estado de la USART0. Principalmente incluye banderas, además de un bit para duplicar la velocidad de transmisión

en modo asíncrono y un bit para el modo de comunicación entre multiprocesadores.

7	6	5	4	3	2	1	0
RXC0	TXC0	UDRE0	FE0	DOR0	PE0	U2X0	MPCM0

- **UCSR0B:** Registro B para el control y estado de la USART0. Incluye los habilitadores de recursos y de interrupciones, y otros bits para transmisiones de 9 bits.

7	6	5	4	3	2	1	0
RXCIE0	TXCIE0	UDRIE0	RXEN0	TXEN0	UCSZ02	RXB80	TXB80

- **UCSR0C:** Registro C para el control y estado de la USART0. En este registro se configura el modo de transmisión, la paridad, el tamaño de los datos y la polaridad en una comunicación síncrona.

7	6	5	4	3	2	1	0
UMSEL01	UMSEL00	UPM01	UPM00	USBS0	UCSZ01	UCSZ00	UCPOL0

Algunos bits de los registros de control tienen un comportamiento diferente cuando la USART0 está en modo SPI Maestro.

Registros para la Comunicación Serial por SPI

- **SPDR:** Registro de datos de la interfaz SPI, es el *buffer* para la transmisión y recepción de datos.
- **SPSR:** Registro de estado, contiene dos banderas y un bit para duplicar la velocidad de transmisión.

7	6	5	4	3	2	1	0
SPIF	WCOL	-	-	-	-	-	SPI2X

- **SPCR:** Registro de control, habilita el recurso y establece las condiciones de las transferencias SPI.

7	6	5	4	3	2	1	0
SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0

Registros de la Interfaz TWI

- **TWDR:** Registro de datos de la interfaz TWI, es el *buffer* para la transmisión y recepción de datos.
- **TWBR:** Registro para establecer la velocidad de las transferencias.
- **TWAR:** Registro de dirección, contiene 7 bits para definir la dirección a la cual responderá el MCU como esclavo y al habilitador de llamadas generales.

7	6	5	4	3	2	1	0
TWA6	TWA5	TWA4	TWA3	TWA2	TWA1	TWA0	TWGCE

- **TWAMR:** Máscara para la dirección, los bits que se ponen en alto se ignoran durante la comparación de la dirección.

7	6	5	4	3	2	1	0
TWAM6	TWAM5	TWAM4	TWAM3	TWAM2	TWAM1	TWAM0	-

- **TWCR:** Registro de control, a través de este registro se habilita la interfaz TWI y se coordinan las transferencias.

7	6	5	4	3	2	1	0
TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	-	TWIE

- **TWSR:** Registro de estado, contiene 5 bits con el estado de la interfaz y 2 bits que definen un factor de preescala para la velocidad de las transferencias.

7	6	5	4	3	2	1	0
TWS7	TWS6	TWS5	TWS4	TWS3	-	TWPS1	TWPS0

Apéndice B

Repertorio de Instrucciones

El repertorio de instrucciones del ATMega328P contiene 131 instrucciones organizadas en 5 grupos:

1. Instrucciones aritméticas y lógicas (28).
2. Instrucciones para el control de flujo (Bifurcaciones) (36).
3. Instrucciones de transferencia de datos (35).
4. Instrucciones para el manejo de bits (28).
5. Instrucciones especiales (4).

En las siguientes tablas se resumen las instrucciones.

Tabla B.1: Instrucciones Aritméticas y Lógicas

Instrucción	**Descripción**	**Operación**
`ADD Rd, Rs`	Suma sin acarreo	$Rd = Rd + Rs$
`ADC Rd, Rs`	Suma con acarreo	$Rd = Rd + Rs + C$
`ADIW Rd, k`	Suma constante a palabra	$[Rd+1 : Rd] = [Rd+1 : Rd] + k$
`SUB Rd, Rs`	Resta sin acarreo	$Rd = Rd - Rs$
`SUBI Rd, k`	Resta constante	$Rd = Rd - k$
`SBC Rd, Rs`	Resta con acarreo	$Rd = Rd - Rs - C$
`SBCI Rd, k`	Resta constante con acarreo	$Rd = Rd - k - C$
`SBIW Rd, k`	Resta constante a palabra	$[Rd+1 : Rd] = [Rd+1 : Rd] - k$
`MUL Rd, Rs`	Multiplicación sin signo	$R1 : R0 = Rd \times Rs$
`MULS Rd, Rs`	Multiplicación con signo	$R1 : R0 = Rd \times Rs$
`MULSU Rd, Rs`	Rd con signo y Rs sin signo	$R1 : R0 = Rd \times Rs$
`FMUL Rd, Rs`	Multiplicación fraccional sin signo	$R1 : R0 = (Rd \times Rs) \ll 1$
`FMULS Rd, Rs`	Multiplicación fraccional con signo	$R1 : R0 = (Rd \times Rs) \ll 1$
`FMULSU Rd, Rs`	Rd con signo y Rs sin signo	$R1 : R0 = (Rd \times Rs) \ll 1$
`AND Rd, Rs`	AND entre registros	Rd = Rd AND Rs
`ANDI Rd, k`	AND de un registro con una constante	Rd = Rd AND k
`OR Rd, Rs`	OR entre registros	Rd = Rd OR Rs
`ORI Rd, k`	OR de un registro con una constante	Rd = Rd OR k
`EOR Rd, Rs`	OR exclusiva entre registros	Rd = Rd XOR Rs
`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)
`COM Rd`	Complemento a 1	Rd = 0xFF - Rd
`NEG Rd`	Negado o complemento a 2	Rd = 0x00 - Rd
`INC Rd`	Incrementa un registro	Rd = Rd + 1
`DEC Rd`	Decrementa un registro	Rd = Rd - 1
`TST Rd`	Evalúa un registro	Rd = Rd AND Rd
`CLR Rd`	Limpia un registro (pone en bajo)	Rd = 0x00
`SER Rd`	Ajusta un registro (pone en alto)	Rd = 0xFF

Tabla B.2: Instrucciones para el Control de Flujo

Instrucción	**Descripción**	**Operación**
`RJMP k`	Salto relativo	PC = PC + k
`IJMP`	Salto indirecto	PC = Z
`JMP k`	Salto absoluto	PC = k
`RCALL k`	Llamada relativa a una rutina	$Pila \leftarrow PC + 1, PC = PC + k + 1$
`ICALL`	Llamada indirecta a una rutina	$Pila \leftarrow PC + 1, PC = Z$
`CALL k`	Llamada absoluta a una rutina	$Pila \leftarrow PC + 1, PC = k$
`RET`	Retorno de una rutina	$PC \leftarrow Pila$
`RETI`	Retorno de una ISR	$PC \leftarrow Pila, I = 1$
`CP Rd, Rs`	Compara dos registros	Rd - Rs
`CPC Rd, Rs`	Comparación con acarreo	Rd - Rs - C
`CPI Rd, k`	Compara un registro con una constante	Rd - k
`BRBS s, k`	Brinca si el bit s de SREG está en alto	si(SREG(s)==1) PC = PC + 1 + k
`BRBC s, k`	Brinca si el bit s de SREG está en bajo	si(SREG(s)==0) PC = PC + 1 + k
`BRIE k`	Brinca si las interrupciones están habilitadas	si(I==1) PC = PC + 1 + k
`BRID k`	Brinca si interrupciones están deshabilitadas	si(I==0) PC = PC + 1 + k
`BRTS k`	Brinca si el bit T está en alto	si(T==1) PC = PC + 1 + k
`BRTC k`	Brinca si el bit T está en bajo	si(T==0) PC = PC + 1 + k
`BRHS k`	Brinca si hubo acarreo del nibble bajo	si(H==1) PC = PC + 1 + k
`BRHC k`	Brinca si no hubo acarreo del nibble bajo	si(H==0) PC = PC + 1 + k
`BRGE k`	Brinca si es mayor o igual que (con signo)	si(S==0) PC = PC + 1 + k
`BRLT k`	Brinca si es menor que (con signo)	si(S==1) PC = PC + 1 + k
`BRVS k`	Brinca si hubo sobreflujo aritmético	si(V==1) PC = PC + 1 + k
`BRVC k`	Brinca si no hubo sobreflujo aritmético	si(V==0) PC = PC + 1 + k
`BRMI k`	Brinca si es negativo	si(N==1) PC = PC + 1 + k
`BRPL k`	Brinca si no es negativo	si(N==0) PC = PC + 1 + k
`BREQ k`	Brinca si los datos son iguales	si(Z==1) PC = PC + 1 + k
`BRNE k`	Brinca si los datos no son iguales	si(Z==0) PC = PC + 1 + k
`BRSH k`	Brinca si es mayor o igual que	si(C==0) PC = PC + 1 + k
`BRLO k`	Brinca si es menor que	si(C==1) PC = PC + 1 + k
`BRCS k`	Brinca si hubo acarreo	si(C==1) PC = PC + 1 + k
`BRCC k`	Brinca si no hubo acarreo	si(C==0) PC = PC + 1 + k
`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 I/O P está en alto	si(P(b)==1) PC = PC + 2 o 3
`SBIC P, b`	Un saltito si el bit b del Registro I/O P está en bajo	si(P(b)==0) PC = PC + 2 o 3

Tabla B.3: Instrucciones de transferencia de datos

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]
`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 postincremento	Rd=Mem[X], X=X+1
`LD Rd, -X`	Carga indirecta con predecremento	X=X-1, Rd=Mem[X]
`LD Rd, Y`	Carga indirecta de memoria	Rd=Mem[Y]
`LD Rd, Y+`	Carga indirecta con postincremento	Rd=Mem[Y], Y=Y+1
`LD Rd, -Y`	Carga indirecta con predecremento	Y=Y-1, Rd=Mem[Y]
`LD 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 postincremento	Rd=Mem[Z], Z=Z+1
`LD Rd, -Z`	Carga indirecta con predecremento	Z=Z-1, Rd=Mem[Z]
`LD 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 postincremento	Mem[X]=Rs, X=X+1
`ST -X, Rs`	Almacenamiento indirecto con predecremento	X=X-1, Mem[X] = Rs
`ST Y, Rs`	Almacenamiento indirecto en memoria	Mem[Y]=Rs
`ST Y+, Rs`	Almacenamiento indirecto con postincremento	Mem[Y]=Rs, Y=Y+1
`ST -Y, Rs`	Almacenamiento indirecto con predecremento	Y=Y-1, Mem[Y] = Rs
`ST 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 postincremento	Mem[Z]=Rs, Z=Z+1
`ST -Z, Rs`	Almacenamiento indirecto con predecremento	Z=Z-1, Mem[Z] = Rs
`ST 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 a R0	R0=Flash[Z]
`LPM Rd, Z`	Carga indirecta de memoria de programa a Rd	Rd=Flash[Z]
`LPM Rd, Z+`	Carga indirecta con postincremento	Rd=Flash[Z], Z=Z+1
`SPM`	Almacenamiento indirecto a memoria de programa	Flash[Z]=R1:R0
`IN Rd, P`	Lee un Registro I/O, deja el resultado en Rd	Rd=Reg_I/O[P]
`OUT P, Rs`	Escribe el valor de Rs en un Registro I/O	Reg_I/O[P]=Rs

Tabla B.4: Instrucciones para el manejo de bits

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
`ASR Rd`	Desplazamiento aritmético a la derecha	Rd(n)=Rd(n+1), n=0..6
`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
`SWAP Rd`	Intercambia nibbles en Rd	Rd[7:4]=Rd[3:0] y Rd[3:0]=Rd[7:4]
`SBI P, b`	Pone en alto al bit b del regristro P	P(b) = 1
`CBI P, b`	Pone en bajo al bit b del regristro P	P(b) = 0
`BTS Rs, b`	Almacena el bit b de Rs en el bit T de SREG	SREG(T) = Rs(b)
`BTL Rd, b`	Carga al bit b de Rd desde el bit T de SREG	Rd(b) = SREG(T)
`BSET s`	Pone en alto al bit s de SREG	SREG(s) = 1
`BCLR s`	Pone en bajo al bit s de SREG	SREG(s) = 0
`SEI`	Pone en alto al habilitador de interrupciones	SREG(I) = 1
`CLI`	Pone en bajo al habilitador de interrupciones	SREG(I) = 0
`SET`	Pone en alto al bit de transferencias	SREG(T) = 1
`CLT`	Pone en bajo al bit de transferencias	SREG(T) = 0
`SEH`	Pone en alto el acarreo del nibble bajo	SREG(H) = 1
`CLH`	Pone en bajo el acarreo del nibble bajo	SREG(H) = 0
`SES`	Pone en alto a la bandera de signo	SREG(S) = 1
`CLS`	Pone en bajo a la bandera de signo	SREG(S) = 0
`SEV`	Pone en alto a la bandera de sobreflujo	SREG(V) = 1
`CLV`	Pone en bajo a la bandera de sobreflujo	SREG(V) = 0
`SEN`	Pone en alto a la bandera de negativo	SREG(N) = 1
`CLN`	Pone en bajo a la bandera de negativo	SREG(N) = 0
`SEZ`	Pone en alto a la bandera de cero	SREG(Z) = 1
`CLZ`	Pone en bajo a la bandera de cero	SREG(Z) = 0
`SEC`	Pone en alto a la bandera de acarreo	SREG(C) = 1
`CLC`	Pone en bajo a la bandera de acarreo	SREG(C) = 0

Tabla B.5: Instrucciones especiales

Instrucción	**Descripción**
`NOP`	No operación, obliga a una espera de 1 ciclo de reloj
`SLEEP`	Introduce al MCU al modo de bajo consumo previamente configurado
`WDR`	Reinicia al *Watchdog Timer*
`BREAK`	Para depuración con la interfaz *Debug Wire*

Apéndice C

Uso de Microchip Studio

Microchip Studio es una suite que permite trabajar con microcontroladores AVR de 8 bits y con otros dispositivos de la empresa Microchip. El IDE incluye al compilador para C/C++ y es totalmente compatible con el estándar ANSI C.

C.1. Construcción de un Proyecto

En esta sección se describe el proceso a seguir para crear y construir un proyecto. El proceso se ilustra con la solución del Ejemplo 3.1, en lenguaje C.

1. Al ejecutar el programa Microchip Studio se abre la ventana mostrada en la Figura C.1.

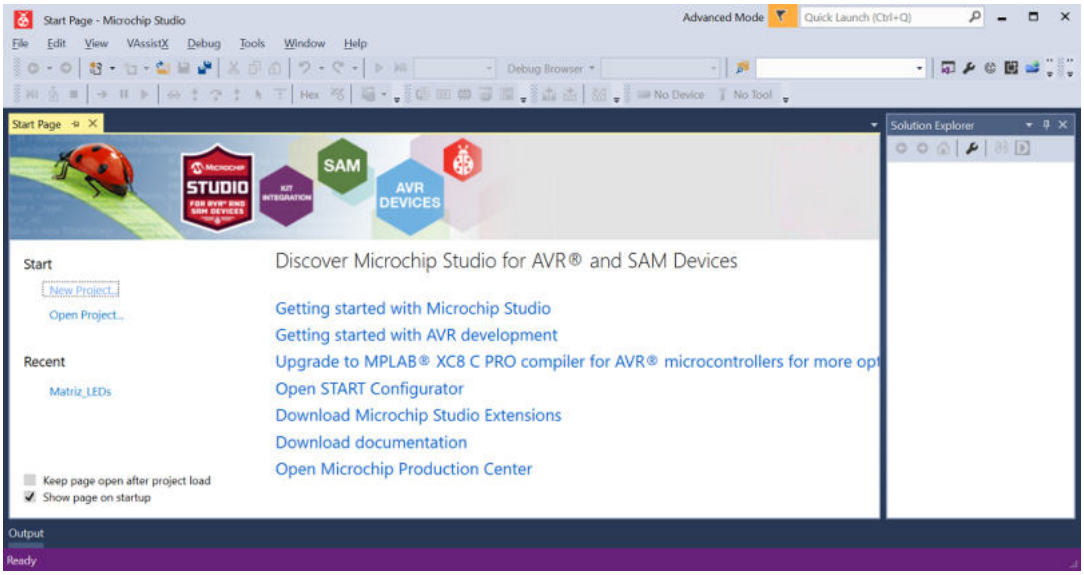


Figura C.1: Ventana inicial de Microchip Studio

2. Para iniciar un nuevo proyecto se debe acceder al menú **File**, seleccionar la opción **New** y en el sub-menú seleccionar **Project**, como se muestra Figura C.2.

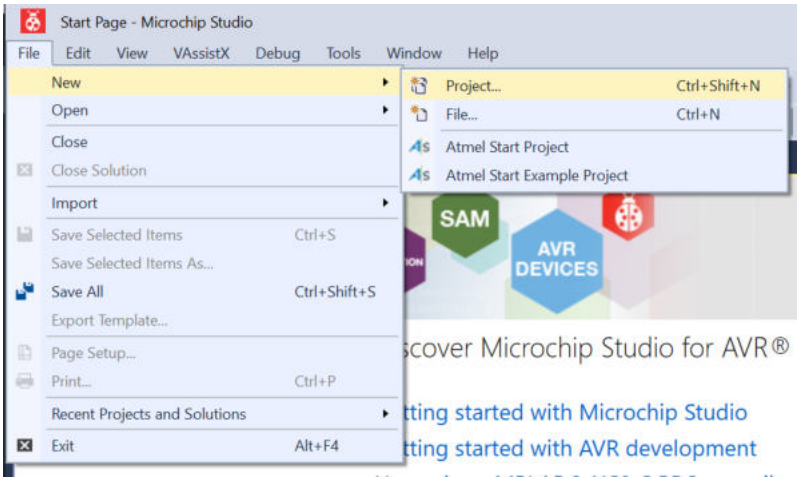


Figura C.2: Creación de un nuevo proyecto

3. A la izquierda de la ventana del nuevo proyecto se indica si se empleará C/C++ o ensemblador. En este caso se elige C/C++ y con ello salen diferentes opciones, de las cuales debe elegirse **GCC C Executable Project**, como se puede ver en la Figura C.3. En la parte inferior se debe indicar el nombre del proyecto y su ubicación, también se tiene una casilla para crear una carpeta en donde será ubicado el proyecto, no es necesario marcarla porque ya se crea una por defecto, si se marca, se creará una carpeta adicional. Después de indicar el tipo de proyecto, definir su nombre y ubicación, se debe presionar **OK**.

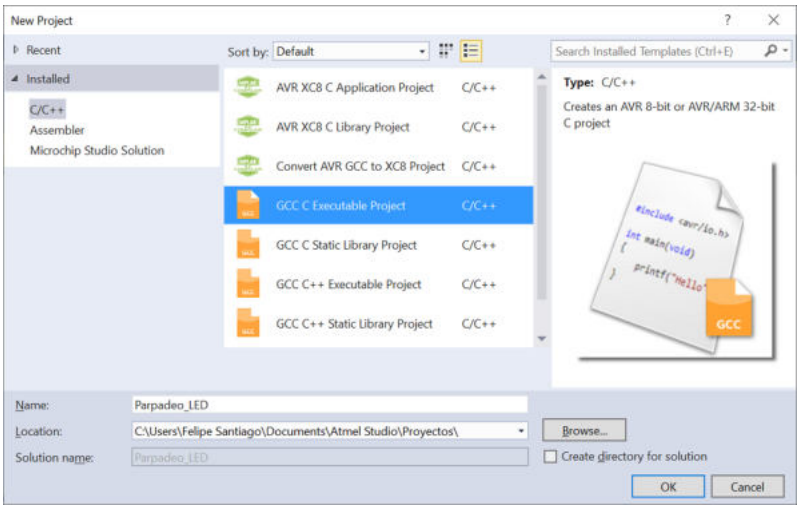


Figura C.3: Selección del tipo de proyecto

4. Posteriormente se debe seleccionar el MCU en una lista de los dispositivos disponibles, en este caso se elige al ATMega328P, como se puede ver en la Figura C.4. En la esquina superior derecha se observa un filtro de búsqueda, si se escribe 328 se reduce la lista.

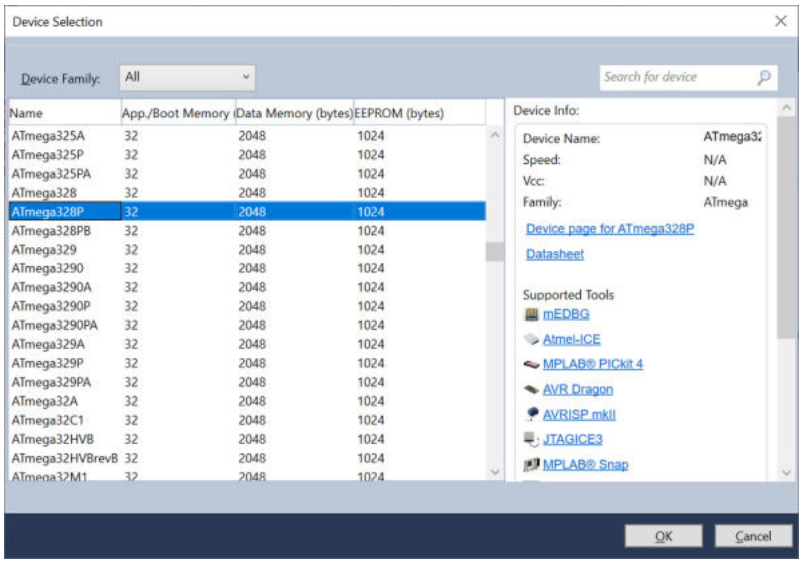


Figura C.4: Selección del dispositivo a emplear

5. Elegido el dispositivo se debe presionar **OK** para continuar. Con ello, el entorno está listo para la captura del programa, en la Figura C.5 se puede ver que en la ventana de edición se crea una plantilla que contiene al `main`, lista para que el programador la personalice.

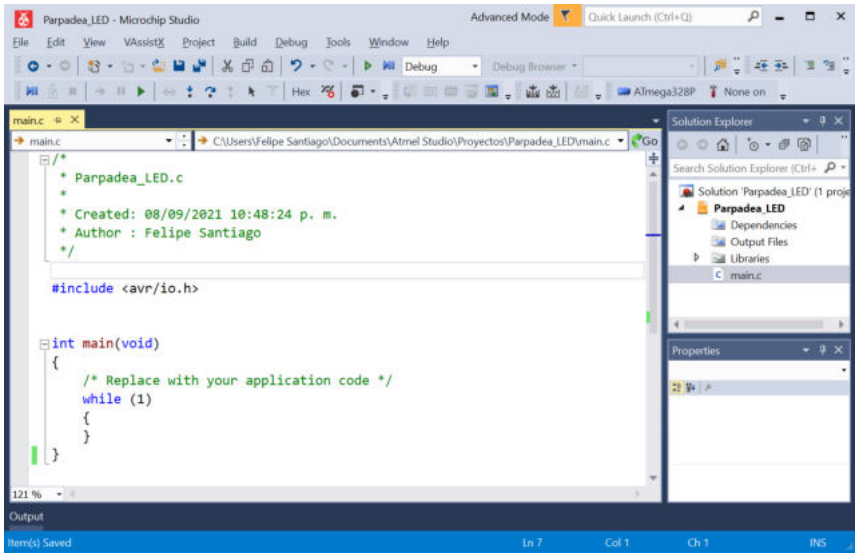


Figura C.5: Entorno listo para la edición del programa

A la derecha de la pantalla está el explorador de la solución, ahí se puede ver resaltado el nombre del archivo `main.c`, eventualmente, cuando un proyecto se cierra, no se abre automáticamente el archivo `main.c` en el momento en que el proyecto se vuelve a abrir, el archivo se debe abrir con un doble click en su nombre desde el explorador de la solución, para que la edición quede en el contexto del proyecto.

6. El proyecto está listo para la edición del programa, en este caso se escribirá el código C del Ejemplo 3.1, que corresponde con el parpadeo de un LED. En la Figura C.6 se muestra el programa capturado.

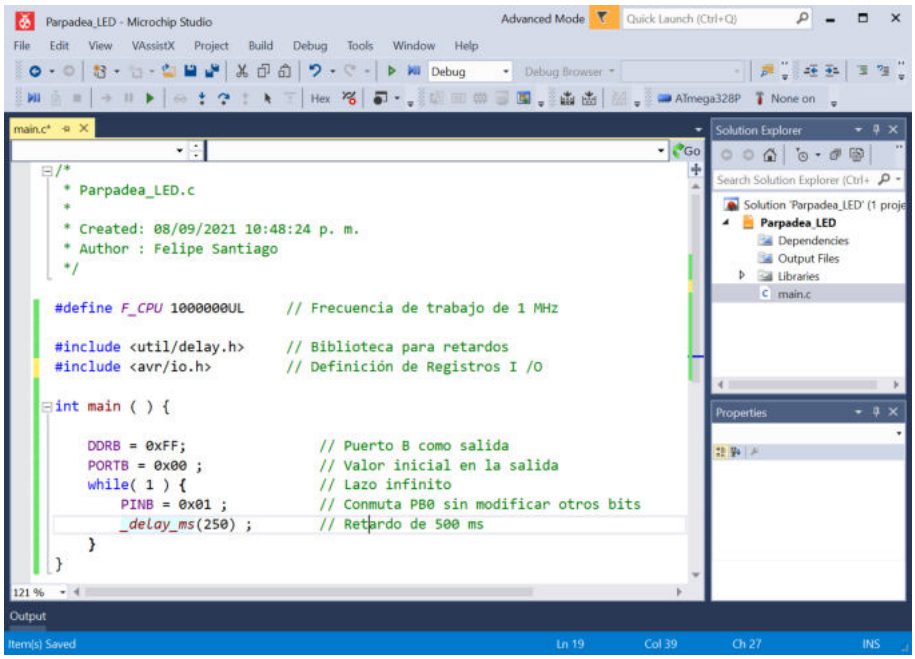


Figura C.6: Programa capturado

7. Una vez que el programa ha sido capturado, se debe construir el archivo HEX que será empleado para programar al MCU. Para ello, desde el menú **Build** se debe seleccionar la opción **Buid Solution**, resaltada en color amarillo en la Figura C.7, también se puede emplear F7 o el botón de acceso rápido que también está marcado con color amarillo.

   Otra alternativa para este proyecto consiste en el uso de la opción **Build Parpadea_LED**, del mismo menú, marcada en verde en la Figura C.7 o bien, se puede usar su botón de acceso rápido, señalizado con el mismo color.

   En este proyecto no hay diferencia en el uso de una u otra opción, la diferencia se presenta cuando un proyecto tiene diferentes archivos fuente, porque el archivo HEX se puede construir a partir de un archivo específico o con todos los arvhivos fuente del proyecto.

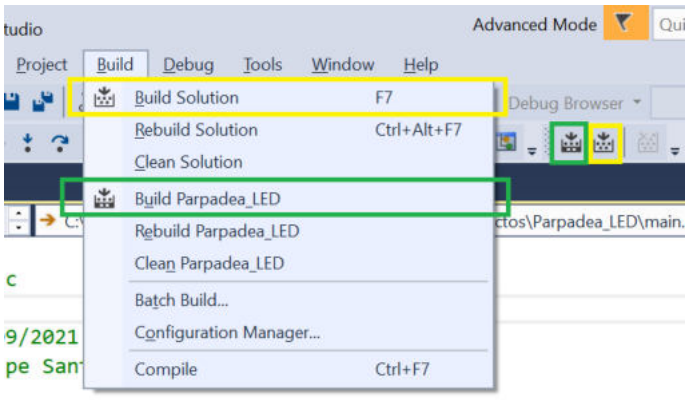


Figura C.7: Opciones para la construcción del archivo HEX

8. Después de realizar la construcción del proyecto, en la parte inferior del entorno emerge una ventana de salida, en donde se indica si hubo errores en la construcción o si se realizó con éxito. En la Figura C.7 se observa que la construcción fue exitosa. También se describen los pasos que se realizaron durante el proceso y se muestra la cantidad de memoria utilizada, en este caso, se puede ver que el proyecto usa 156 bytes de Flash y no utiliza RAM.

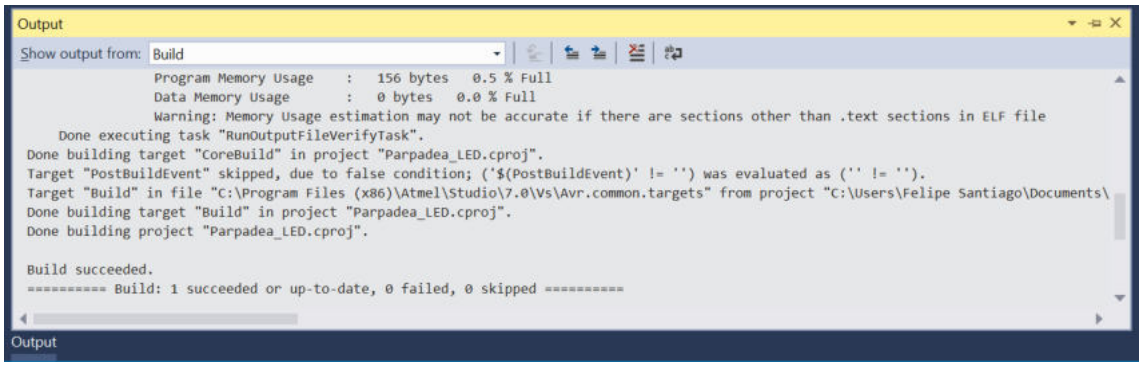


Figura C.8: Resultados de la construcción del proyecto

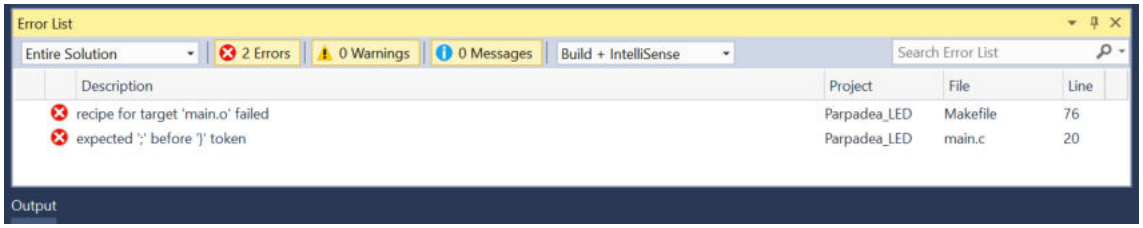


Figura C.9: Error en la construcción del proyecto

9. Al código fuente se le quitó el punto y coma posterior al retardo, dentro del lazo infinito, para ilustrar la generación de un error. En la Figura C.9 se muestra el mensaje: `expect ';' before '}' token`, con doble click en el mensaje, el cursor se ubicará cerca del error, en la ventana de edición del código fuente.

10. Finalmente, como resultados de la construcción de un proyecto se obtienen los archivos con extensión HEX y EEP, que se pueden descargar en el ATMega328P o utilizarlos para la simulación. En la Figura C.10(a) se observa la carpeta del proyecto, con el archivo fuente (`main.c`) y la carpeta debug, que es donde están los archivos generados. El contenido de la carpeta debug se puede ver en la Figura C.10(b).

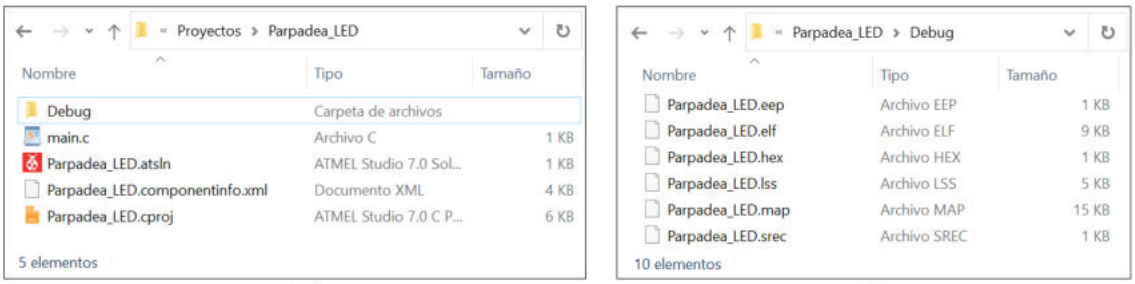


Figura C.10: Carpetas del proyecto

C.2. Simulación de un Proyecto

Comercialmente existen herramientas para una simulación visual de sistemas electrónicos, como Proteus® de Labcenters Electronics, sin embargo, desde el entorno de Microchip Studio también se pueden realizar simulaciones. Las simulaciones en Microchip Studio tienen la ventaja de que muestran el comportamiento de los recursos internos con la desventaja de que no permiten agregar componentes externos al MCU. En esta sección se expone el procedimiento para realizar estas simulaciones.

1. El primer paso es la selección del simulador como herramienta de depuración, para ello, se debe dar un click al botón que se resalta en la Figura C.11, el cual originalmente indica que no hay una herramienta activa.

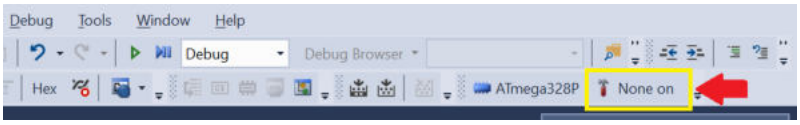


Figura C.11: Botón para elegir la herramienta de depuración

2. En el espacio de trabajo (junto al archivo `main.c`) se genera una pestaña para seleccionar al simulador como herramienta de programación o depuración. En

la Figura C.12 se puede ver una lista en donde se va a seleccionar al simulador, se agregarán otros elementos si se conectan herramientas de depuración propias de Microchip. Después de seleccionar al simulador se debe salvar al proyecto

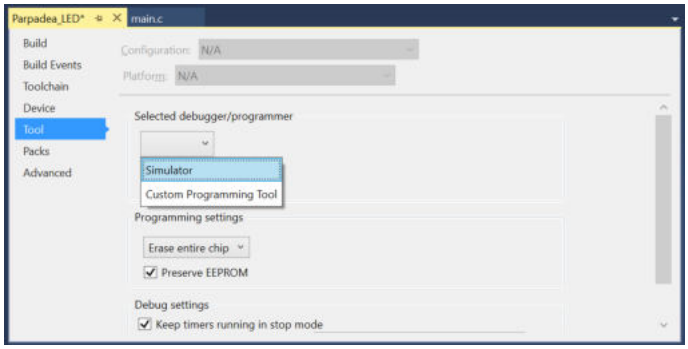


Figura C.12: Selección del simulador como heramienta de depuración

3. Para iniciar la simulación se debe dar un click en la opción **Start Debugging and Break** en en el menú **Debug**. En la Figura C.13 se ilustra esta opción y el botón de acceso rápido encerrado en un círculo.

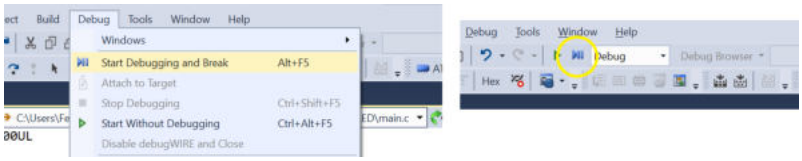


Figura C.13: Opciones para iniciar la simulación

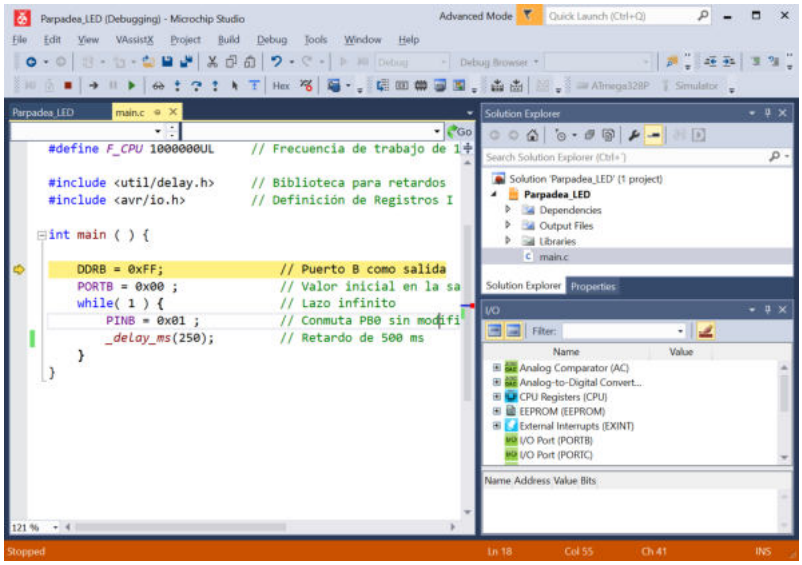


Figura C.14: Simulación iniciada en el entorno de Microchip

En la Figura C.14 se muestra el espacio de trabajo con la simulación iniciada, se observa a la izquierda una flecha amarilla que indica la siguiente instrucción a simular y a la derecha se pueden ver los Registros I/O.

Es posible observar otros recursos del MCU, como el estado de la CPU o el contenido de la memoria, estos se seleccionan con los botones de la herramienta de depuración, los cuales se presentan en la Figura C.15, indicando el recurso que se muestra con cada botón. Estos botones solo están activos cuando la depuración ya inició.

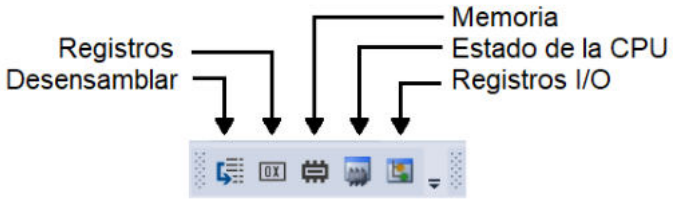


Figura C.15: Botones para mostrar recursos del MCU

4. La simulación se realiza con las opciones presentes en el menú **debug**, que se muestran en la Figura C.16, puede ser continua o paso a paso. En la figura también están los botones de acceso rápido con una breve descripción de los más utilizados.

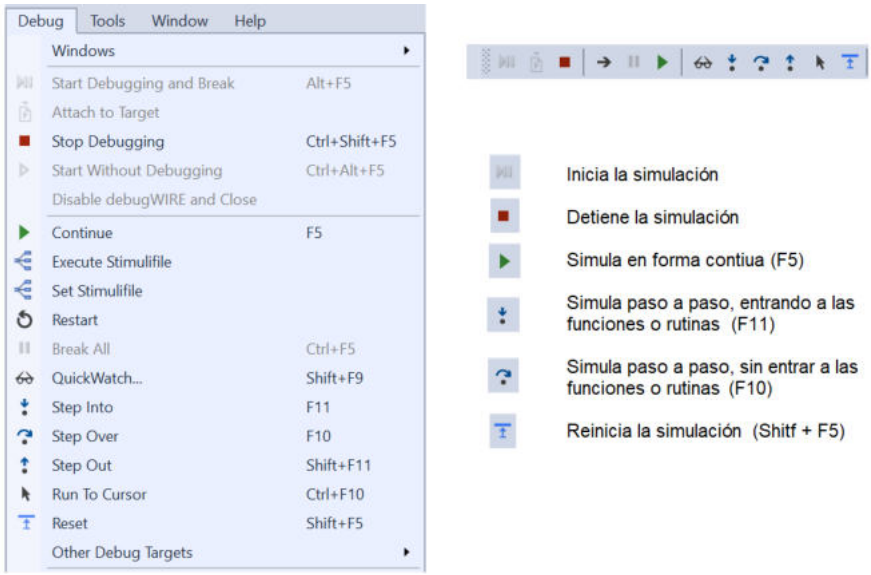


Figura C.16: Opciones y botones para realizar la simulación

5. Si se avanza paso a paso en la simulación, con **F10** para la opción **Step Over**, en el código fuente se verá el avance de la flecha amarilla y a la derecha, en la

ventana de recursos, se mostrarán los cambios en los Registros I/O involucrados. En la Figura C.17 se puede ver el avance de la simulación en el Ejemplo 3.1, específicamente después de que se ha conmutado la salida en PB0. Se agregó una instrucción `NOP` para que la simulación se detenga antes de entrar a la rutina del retardo.

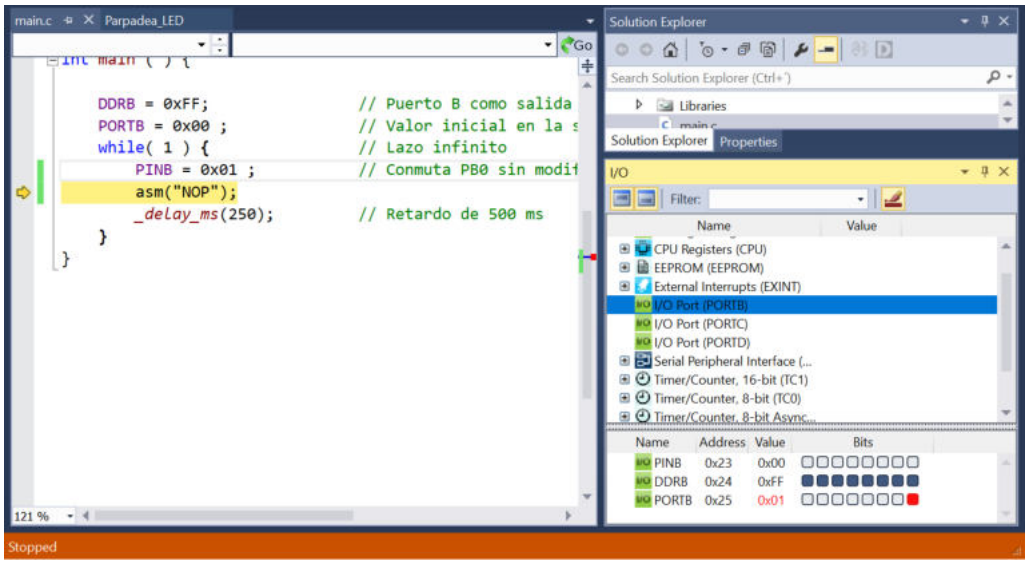


Figura C.17: Simulación avanzada paso a paso

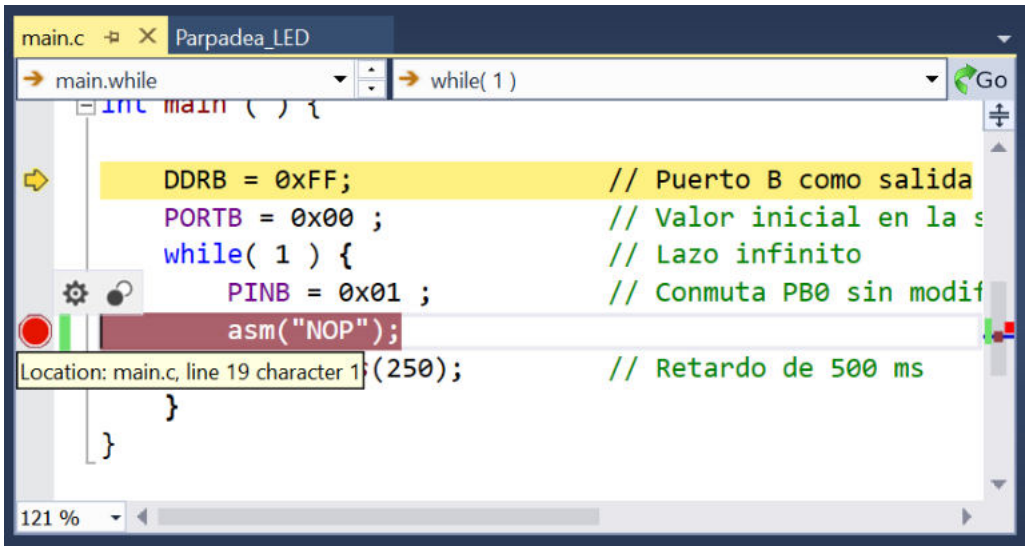


Figura C.18: Insersión de puntos de ruptura (breakpoints)

6. Para hacer la simulación más ágil, se pueden agregar algunos puntos de ruptura (*breakpoints*) en el código fuente, estos se consiguen dando doble click a la

izquiera de la instrucción de interés. En la Figura C.18 se muestra un circulo en color rojo, indicando la posición del punto de ruptura. Con los puntos de ruptura insertados, es conveniente realizar una simulación continua (con F5), el programa se detendrá cuando encuentre un punto de ruptura, hecho esto, se puede dar un paso para observar el efecto de la intrucción de interés, se podrán ver los cambios en los Registros I/O, o bien, en otro tipo de aplicaciones, para modificar alguna entrada. Posteriormente se puede repetir la ejecución continua.

El simulador del entorno de Microchip Studio tiene la ventaja de mostrar el estado de la CPU y de los recursos internos, además, se puede estimar con precisión cuantos ciclos de reloj tarda en ejecutarse una secuencia de código. Los *breakpoints* son muy útiles porque los Registros I/O o el estado de la CPU se actualizan hasta que la simulación se suspende.

C.3. Inserción de una Biblioteca de Funciones

En el Capítulo 8 se desarrolló la biblioteca `TWI.h` para utilizar la interfaz de dos hilos y en el Capítulo 10 se repitió el proceso para la biblioteca `LCD.h`, con las funciones para el manejo de un LCD[1], en general, es posible agrupar un conjunto de funciones en una biblioteca para facilitar la reutilización de código.

En esta sección se muestra como incluir una biblioteca de funciones en un proyecto, dentro del entorno de Microchip Studio. Específicamente, se hará la integración de la biblioteca `LCD.h` en un proyecto que muestra una cadena constante en un LCD.

La biblioteca se compone de dos archivos: el archivo `LCD.h` con los prototipos de las funciones y, el arhivo `LCD.c`, con los cuerpos de las funciones.

El procedimiento para insertar la biblioteca es el siguiente:

1. En Microchip Studio se crea el proyecto, como cuaquier otro, en la Figura C.19 se muestra el proyecto con el código de interés, la biblioteca `LCD.h` se incluye con comillas porque se debe ubicar en la carpeta del proyecto. Se observa en la Figura C.19 que se resaltan las funciones porque no son reconocidas por el entorno.

2. Los archivos `LCD.h` y `LCD.c` deben copiarse en la carpeta del proyecto, en el lugar en donde está ubicado el archivo main.c, en la Figura C.20 se muestra la carpeta, después de que se ha realizado la copia.

[1]En el sitio web del autor están disponibles ambas bibliotecas de funciones.

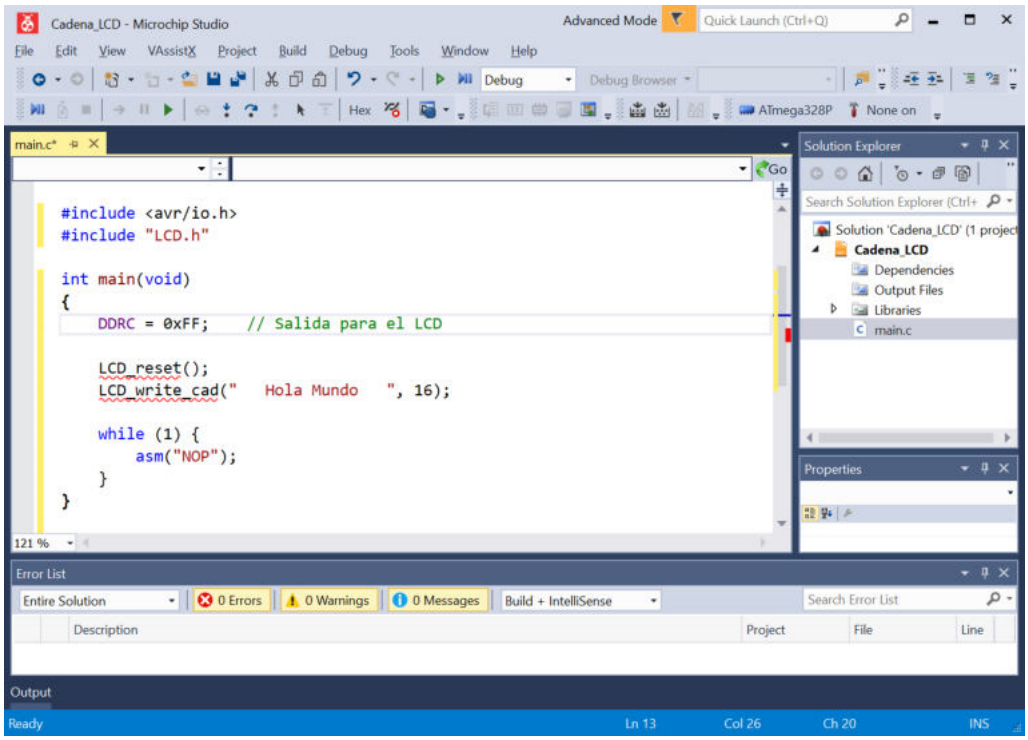


Figura C.19: Proyecto con una biblioteca externa

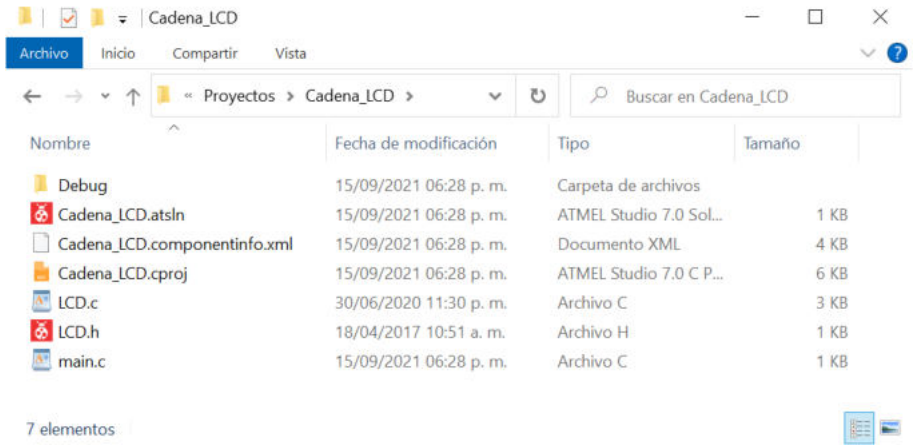


Figura C.20: Copia de la biblioteca en la carpeta del proyecto

3. No es suficiente con la copia de los archivos, si se presiona el botón para construir el proyecto, se obtendrán los mensajes de error: *undefined reference to 'LCD_reset'* y *undefined reference to 'LCD_write_cad'*.

4. Para agregar la biblioteca al proyecto, se debe dar un click derecho en el explorador de la solución, especificamente en el nombre del proyecto y mover

el cursor del ratón a la opción **Add**, en donde se presenta un sub-menú, del cual se debe elegir la opción **Existing Item**. En la Figura C.21 se pueden ver las diferentes opciones.

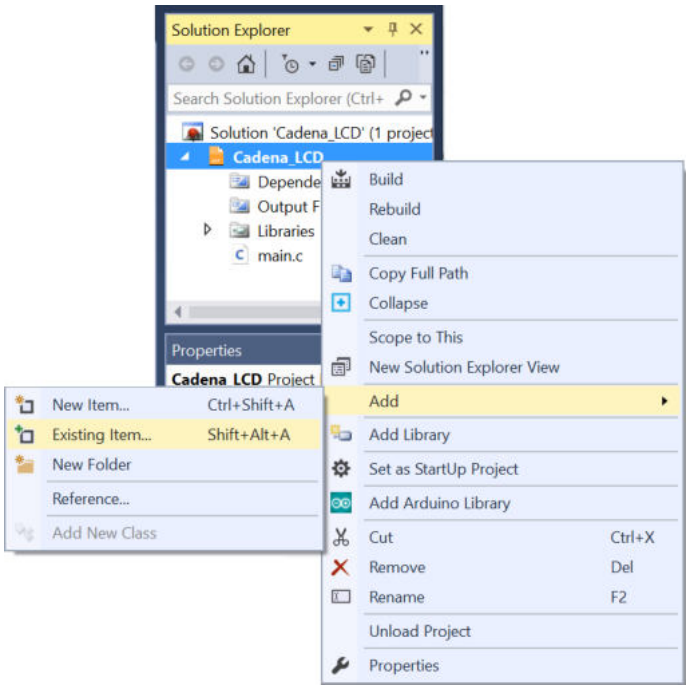


Figura C.21: Menú para agregar archivos a un proyecto

5. Después de dar un click en **Existing Item** se obtiene la ventana emergente de la Figura C.22, en donde únicamente se debe seleccionar al archivo `LCD.c` y dar click en el botón **Add**.

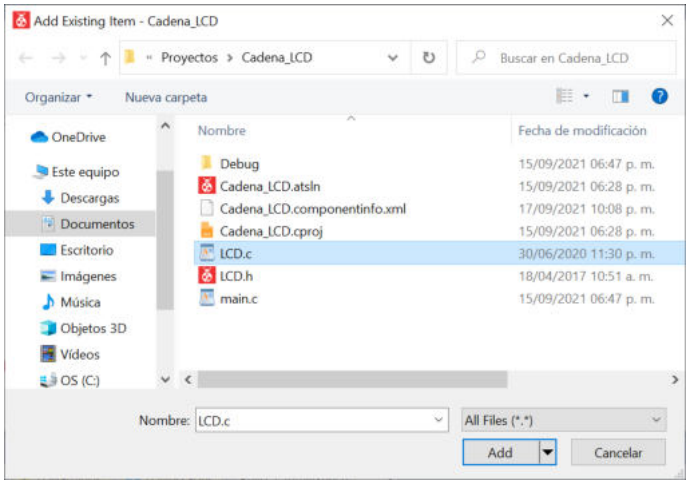


Figura C.22: Selección de un archivo para agregarlo en el proyecto

6. En el explorador de la solución se puede ver que el archivo `LCD.c` ya es parte del proyecto (Figura C.23). El archivo `LCD.h` no se agrega de esta manera porque se está incluyendo con la directiva `include`.

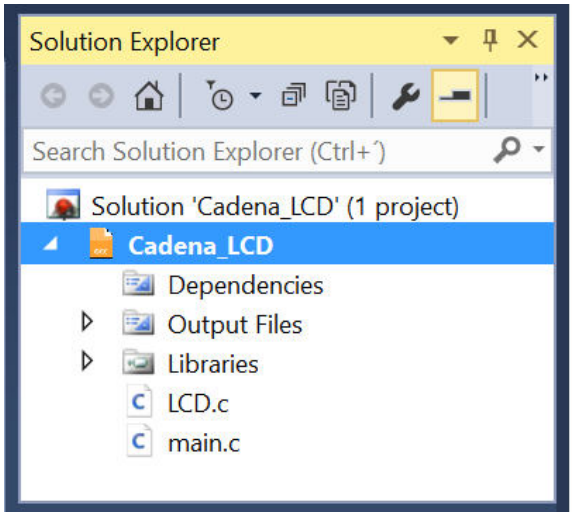


Figura C.23: Archivo `LCD.c` incluido en el proyecto

7. Con el archivo incluido en el proyecto, si se da click en el botón para su construcción, ya no se obtendrán errores y se generará el archivo HEX para la simulación o programación del ATMega328P.

Un proyecto puede incluir tantas bibliotecas de usuario como sean necesarias.