

Libros de **Cátedra**

# Sistemas digitales basados en microcontroladores

Descripción, funcionalidades y aplicaciones  
de los microcontroladores basadas en el CPU HCS08

Jorge R. Osio, Walter J. Aróztegui y José A. Rapallini

FACULTAD DE  
INGENIERÍA

**e**  
exactas

  
EDITORIAL DE LA UNLP



UNIVERSIDAD  
NACIONAL  
DE LA PLATA

# **SISTEMAS DIGITALES BASADOS EN MICROCONTROLADORES**

DESCRIPCIÓN, FUNCIONALIDADES Y APLICACIONES  
DE LOS MICROCONTROLADORES BASADAS  
EN EL CPU HCS08

Jorge R. Osio  
Walter J. Aróztegui  
José A. Rapallini

Facultad de Ingeniería



UNIVERSIDAD  
NACIONAL  
DE LA PLATA

  
**EduLP**  
EDITORIAL DE LA UNLP

Dedicado a todas las personas que nos acompañan  
en la vida cotidiana y nos dan fuerza  
con sus palabras de aliento

# Agradecimientos

Un especial agradecimiento a David Domínguez por su gran aporte en el diseño de los kits basados en el procesador hcs08.

Se agradece especialmente a todos los docentes y auxiliares alumnos que han brindado su experiencia a lo largo de los años enriqueciendo el contenido de estas páginas.

A los Docentes que nos dieron la oportunidad de iniciarnos en esta profesión tan noble, en especial a Nilda Garcia y Mirta Salerno.

Al fundador del CeTAD Antonio Quijano, por su apoyo y ejemplo de trabajo diario.

A la UIDET CeTAD, que nos brindó el espacio físico y los recursos materiales necesarios para el trabajo diario.

Muy especialmente a nuestros compañeros de trabajo, becarios y alumnos de Trabajo Final, sin cuyo aporte y colaboración este trabajo no se hubiera consumado.

A nuestras Familias por todo su apoyo incondicional.

*Esta producción se logró mediante un esfuerzo conjunto entre los autores y la UNLP con el objeto de mejorar la calidad de la enseñanza.*

*La obra intenta introducir al lector en el diseño e implementación de sistemas digitales mediante la descripción de los principales conceptos y la aplicación de técnicas de diseño eficientes. Se debe resaltar que aunque las descripciones se basan en una tecnología específica, el contenido del libro se puede aplicar de igual forma sobre todas las tecnologías actuales.*

JORGE R. OSIO, Sistemas digitales basados en microcontroladores

# Índice

**Introducción** \_\_\_\_\_ 9

*Jorge R. Osio*

## **PRIMERA PARTE**

### **Descripción y configuración del microprocesador**

#### **Capítulo 1**

Microprocesadores y Microcontroladores \_\_\_\_\_ 11

*Jorge R. Osio y Walter J. Aróztegui*

#### **Capítulo 2**

Descripción del CPU08 \_\_\_\_\_ 29

*Jorge R. Osio, Walter J. Aróztegui y Jose A. Rapallini*

#### **Capítulo 3**

Modos de Direccionamiento \_\_\_\_\_ 39

*Jorge R. Osio y Walter J. Aróztegui*

#### **Capítulo 4**

Memoria \_\_\_\_\_ 57

*Jorge R. Osio*

#### **Capítulo 5**

Interrupciones \_\_\_\_\_ 60

*Jorge R. Osio y Walter J. Aróztegui*

#### **Capítulo 6**

Módulo Oscilador (OSC) \_\_\_\_\_ 65

*Jorge R. Osio y Walter J. Aróztegui*

## **Capítulo 7**

Programación a bajo nivel (Assembler) \_\_\_\_\_ 70

*Jorge R. Osio*

## **Capítulo 8**

Características de la programación en alto nivel \_\_\_\_\_ 79

*Jorge R. Osio*

## **SEGUNDA PARTE**

### **Descripción y Configuración de los Módulos**

## **Capítulo 9**

Módulos disponibles en las diferentes líneas de Microcontroladores \_\_\_\_\_ 94

*Jorge R. Osio y Walter J. Aróztegui*

## **Capítulo 10**

Puertos de entrada/salida \_\_\_\_\_ 104

*Jorge R. Osio y Walter J. Aróztegui*

## **Capítulo 11**

Módulo de Interrupción externa (IRQ) \_\_\_\_\_ 110

*Jorge R. Osio y Walter J. Aróztegui*

## **Capítulo 12**

Módulos de temporización \_\_\_\_\_ 118

*Jorge R. Osio y Walter J. Aróztegui*

## **Capítulo 13**

Módulo de interrupción por teclado (KBI) \_\_\_\_\_ 137

*Jorge R. Osio y Walter J. Aróztegui*

## **Capítulo 14**

Módulo conversor analógico/digital \_\_\_\_\_ 143

*Jorge R. Osio y Walter J. Aróztegui*

## **Capítulo 15**

Interfaz de comunicación serie (SPI) \_\_\_\_\_ 150

*Jorge R. Osio y Walter J. Aróztegui*

**Apéndice A**

Sistemas de numeración y código \_\_\_\_\_ 157

*Jorge R. Osio*

**Apéndice B**

Set de instrucciones \_\_\_\_\_ 164

*Jorge R. Osio*

**Referencias** \_\_\_\_\_ 170

**Los autores** \_\_\_\_\_ 171



# Introducción

Con el avance de la tecnología, los sistemas embebidos han adquirido un papel fundamental en la mayoría de los desarrollos de electrónica digital, es por eso que los diseños de sistemas digitales vienen comúnmente acompañados de un núcleo procesador que se encarga de controlar el sistema y de manejar los periféricos.

Este libro se divide en 2 secciones bien definidas, en donde se describe cómo está formado un microprocesador, su diferencia con un microcontrolador y la descripción completa de este último, teniendo como caso de estudio el HC908QY4 de freescale. Para entender su funcionamiento se hace necesario explicar en qué consiste la programación a bajo nivel y como simular el código de forma eficiente. Por otro lado, se explica la programación en lenguaje de alto nivel, más específicamente en C. Esto último, permite programar sistemas digitales de forma eficiente, mediante código reutilizable y portable.

En la segunda sección del libro se describen los módulos más importantes que poseen los microcontroladores actuales, los cuales permiten implementar una amplia variedad de interfaces con dispositivos externos. Adicionalmente, se desarrollan algunas configuraciones básicas de los mismos teniendo como caso de estudio la familia HCS08 de freescale. También se describen ejemplos de aplicación que permiten entender cuál es la manera más eficiente de realizar un proyecto con características portables y reusables.

*Jorge R. Osio*

## **PRIMERA PARTE**

---

### Descripción y configuración del microprocesador

Cátedra de Circuitos Digitales y Microprocesadores

# CAPÍTULO 1

## Microprocesadores y Microcontroladores

*Jorge R. Osio y Walter J. Aróztegui*

### 1.1. Características generales de un microprocesador

En un sistema microprocesador básico, (basado en la arquitectura Von Newman), como el que muestra la figura siguiente se distinguen; la CPU, el bloque de memoria y los módulos de Entrada/Salida.

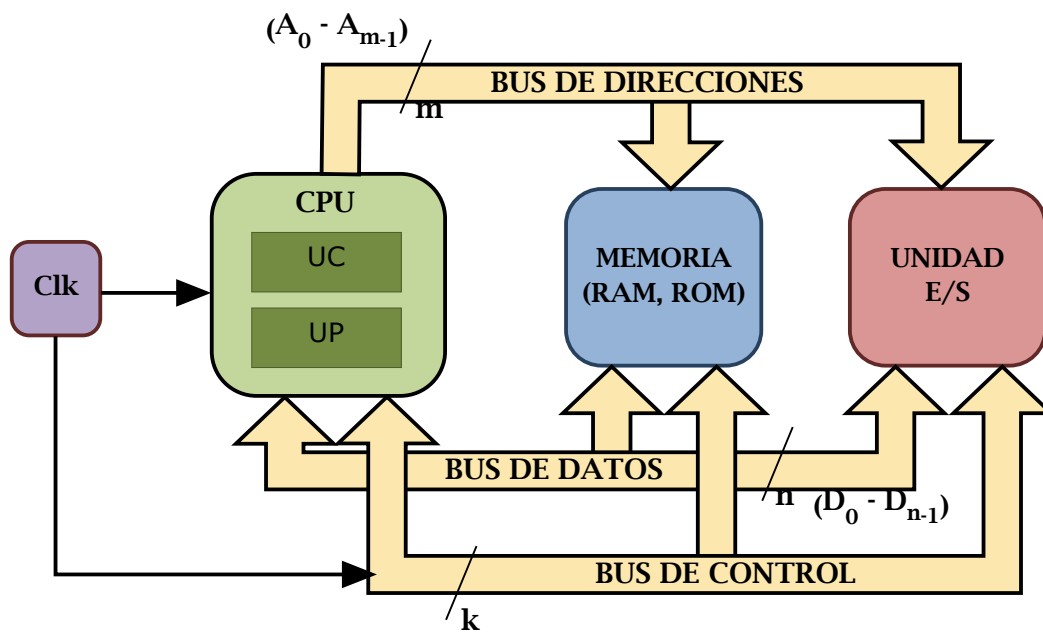


Figura 1. Diagrama de un Microprocesador básico.

La **CPU** es la Unidad Central de Procesos y está formada por la Unidad de Control, Registros y la Unidad de Procesos.

La **memoria** se divide en memoria de programa y memoria de datos, en donde se almacenan las instrucciones de programa y los datos respectivamente.

Los **módulos de entrada/salida** permiten intercambiar información con el mundo exterior. Todo el sistema debe estar sincronizado para funcionar correctamente, es por eso que se utiliza un clock de sincronismo, el cual posibilita que todos los módulos interactúen entre sí de

manera ordenada y coordinada. Los tres bloques principales deben estar vinculados mediante buses de comunicación, (bus de datos, bus de direcciones y bus de control), por donde viajan los datos digitales.

### **1.1.1. Descripción de los buses de comunicación**

#### **1.1.1.1. Bus de Datos**

El bus de datos permite al microprocesador intercambiar datos con los bloques de memoria y los Módulos de E/S. El bus de datos tiene la particularidad de ser bidireccional permitiendo el envío y recepción de datos. Este bus tiene una cantidad de líneas que representa el tamaño de palabra (cantidad de bits de los datos) con la que opera el microprocesador.

Las líneas del bus de datos deben ser tri-state, por lo que pueden estar en bajo (0 lógico), en alto (1 lógico) o en alta impedancia (HZ). Entonces, cuando un dispositivo se encuentra conectado al bus de datos, sus líneas podrán adquirir los estados L y H, luego los demás dispositivos estarán con sus líneas desconectadas en alta impedancia.

#### **1.1.1.2. Bus de direcciones**

Mediante este bus el microprocesador selecciona la dirección de memoria donde se desea leer o escribir un dato. Por este motivo dicho bus es unidireccional y la cantidad de líneas que lo constituyen determinan la cantidad de bits que se utiliza para representar una dirección. Si la cantidad de líneas es  $n$ , entonces la CPU podrá direccionar  $2^n$  direcciones de memoria diferentes (máximo direccionamiento). Si el bus tiene 16 líneas, esto quiere decir que la CPU podrá direccionar  $2^{16}$  posiciones de memoria (65536 posiciones).

#### **1.1.1.3. Bus de Control**

Por estas líneas circulan las señales de control y sincronización del sistema, entre las más comunes se encuentran:

- Señal de clock
- Señal de reset
- Señal de lectura/escritura de memoria
- Señal de Interrupción

## 1.1.2. Descripción del Bloque de Memoria

La memoria se puede separar en dos bloques bien definidos, el de memoria de datos y el de instrucciones.

### 1.1.2.1. Memoria de datos

La memoria de datos se utiliza para almacenar los datos y variables del programa. Estos datos son normalmente de 8 bits (byte), 16 bits (word) o 32 bits (long Word). Para el almacenamiento de datos y variables se utiliza memoria RAM, la cual es de lectura/escritura y se caracteriza por ser volátil.

### 1.1.2.2. Memoria de instrucciones

La memoria de instrucciones contiene el programa que ejecuta el microprocesador, el cual está formado por instrucciones que se ejecutan de manera secuencial. En dicha memoria las instrucciones se codifican mediante un Código de Operación (OP), y están formadas por uno o varios bytes. Para el almacenamiento del programa se utiliza memoria Flash (no volátil), la cual es de solo lectura y permite almacenar el programa a ejecutar por el microprocesador y valores constantes a ser utilizados por dicho programa.

## 1.1.3. Descripción de la CPU

La unidad central de procesos está formada por registros internos, la unidad de control y la unidad de procesos.

### 1.1.3.1. Registros internos

Estos registros son bloques biestables que permiten almacenar los datos básicos con los cuales va a trabajar la CPU para ejecutar las instrucciones que forman el programa.

La figura 2 muestra la estructura interna de un registro, en donde el control de entrada es gobernado por una señal de sincronismo y el valor del registro quedará disponible a la salida cuando el control de salida CS este en '0 lógico', cuando CS esté en '1' la salida estará en alta impedancia.

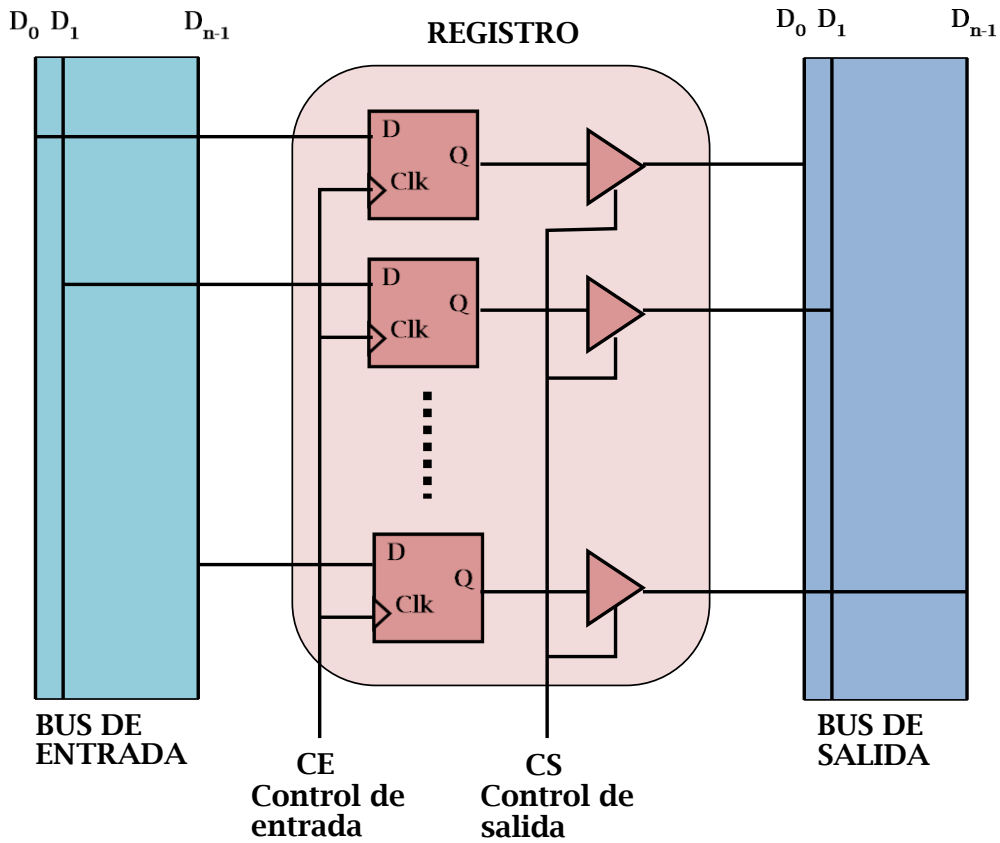


Figura 2. Estructura interna de un registro.

En la Figura 3 se ejemplifica el funcionamiento de los registros mediante el envío de datos de un registro a otro, en donde se observa que para enviar un dato desde el registro A hacia el B, se debe colocar el chip select "CS1" del registro A en bajo, esperar a que el Bus 2 esté disponible, luego poner en alto el Chip Enable "CE2" del registro B y por último poner CE1 en bajo.

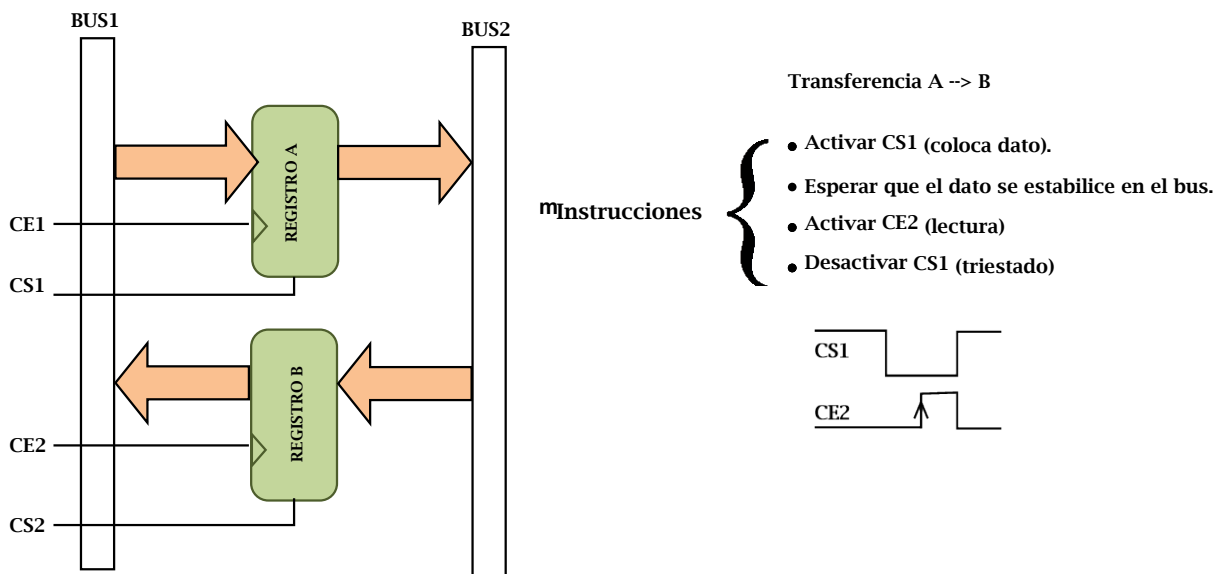


Figura 3. Esquema de una Transferencia de Datos entre Registros.

Es importante destacar que los datos almacenados en los Registros de la CPU tienen un tiempo de acceso mucho menor al de los datos almacenados en memoria externa.

### 1.1.3.2. Unidad de control

La unidad de control de la CPU se encarga de decodificar las instrucciones almacenadas en Memoria y ejecutar las **micro-operaciones** que la conforman.

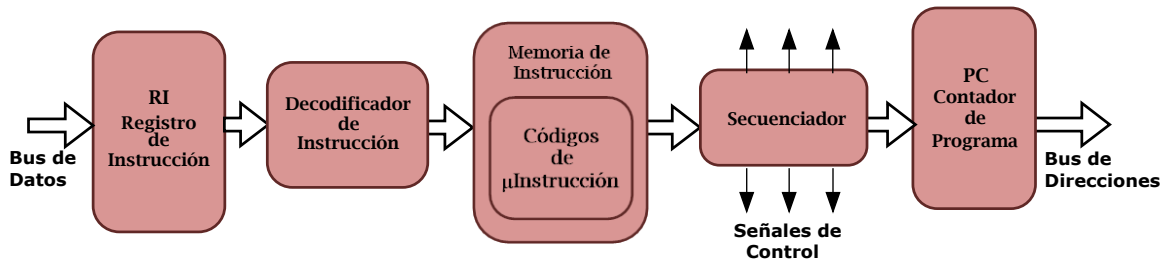


Figura 4. Estructura interna de la Unidad de Control

En la Figura 4 se muestra un diagrama de la **Unidad de control**, en donde se recibe el código binario de la instrucción a ejecutar por el bus de datos y se almacena en el **registro de instrucciones**. Luego, el **decodificador de instrucciones** selecciona las micro-instrucciones, (relativas a la instrucción a ejecutar), en una Memoria ROM de la CPU llamada “**Memoria de Instrucciones**” o “**Memoria de Microprograma**”. Dependiendo de las micro-instrucciones, el bloque **secuenciador** se encargará de activar secuencialmente un conjunto de señales que se corresponden con dichas micro-instrucciones y le indican a los diferentes componentes de la arquitectura la operación a realizar.

La dirección de una instrucción de salto en la entrada del secuenciador puede venir de tres lugares diferentes: del registro de instrucción, de la unidad de control de interrupciones, ó de la memoria de Microprograma. El tamaño de la palabra de las instrucciones codificadas es de 8 bits. Este código es cargado en el registro de instrucciones y extendido de 8 a 12 bits mediante la adición de cuatro ceros a la derecha. Este nuevo valor es la dirección en la memoria de microprograma en donde comienzan las micro-operaciones que ejecuta esta instrucción.

El **Contador de Programa (PC)** es un registro que contiene la dirección de memoria donde está la siguiente instrucción del programa a ejecutar. Una de las primeras acciones del secuenciador antes de ejecutar una instrucción es incrementar el PC para que apunte a la instrucción siguiente.

### 1.1.3.3. Unidad de procesos

El bloque principal es la **ALU o Unidad Lógico-Aritmética (ver figura 5)**, que permite realizar las operaciones aritméticas y lógicas básicas como sumas, restas, y, o, o exclusiva, negación, etc.

El secuenciador de la unidad de control activa las líneas de selección de la ALU para realizar la operación indicada por la instrucción en curso.

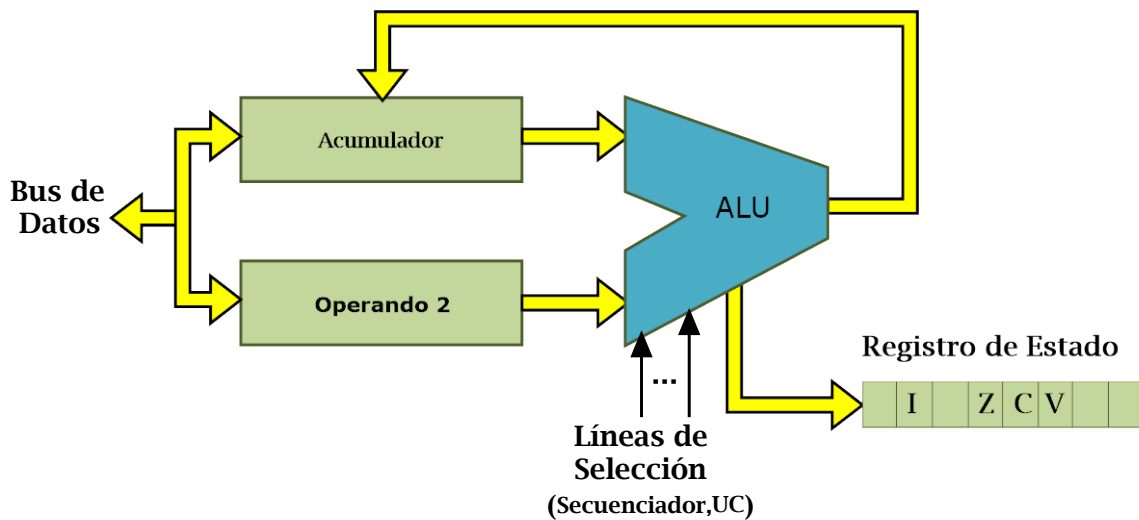


Figura 5. Estructura interna de la Unidad de Procesos.

Los operandos se suministran por medio de dos registros cargados desde el bus de datos, uno de los registros es el Acumulador, que se utiliza para realizar todas las operaciones y en donde se guarda el resultado de las mismas, el otro registro puede ser un registro cualquiera o una dirección de memoria, aquí se lo denomina **Registro 2º Operando**.

- **Registro Acumulador:** Contiene siempre el resultado de la última operación realizada en la ALU.
- **Registro 2º Operando:** Es el 2º operando necesario para realizar la operación que se corresponde con la instrucción a ejecutar, el código de operación de la instrucción a ejecutar indica que registro representará este operando según los diferentes modos de direccionamiento.

Esta forma de realizar las operaciones permite simplificar la ejecución de las instrucciones ya que cada instrucción sólo tiene que suministrar un operando, el otro se encuentra cargado previamente en el acumulador.

El **Registro de Estado** está formado por bits denominados banderas (flags) que se ponen a 1 ó 0 de acuerdo con el resultado de la instrucción ejecutada. Los flags más comunes son:

- **Z**, bit *zero*, se pone en 1 si el resultado es cero.
- **C**, bit *carry*, se pone en 1 si hubo acarreo de orden superior
- **V**, bit *overflow*, se pone en 1 si hubo desborde
- **I**, bit de *interrupción*; Este bit es independiente del resultado. Poniendo este flag en 1 se pueden inhibir las interrupciones enmascarables.



### 1.1.4. Descripción de la unidad de Entrada/Salida

Permite la comunicación del sistema Microprocesador con otros dispositivos. Los dispositivos de E/S se denominan habitualmente **periféricos**, por ejemplo, teclados, displays, unidades de Memoria, etc.

Cualquier periférico necesita un módulo adicional que permite realizar la conexión del mismo con los buses del sistema Microprocesador. En los Microcontroladores estos módulos se encuentran en el mismo chip que el procesador.

Existen varios métodos para manejar los dispositivos de E/S:

- 1.- Mediante **Acceso Directo a Memoria (DMA)**. La CPU pone los buses de direcciones y de datos en tri-state. Un dispositivo controlador de DMA toma el control de los buses y pasa los datos directamente entre el dispositivo E/S y la memoria. Cabe aclarar que este método se encuentra disponible en los Microcontroladores más modernos, en el HC908 no está disponible.
- 2.- Mediante **Técnicas de Interrupción**. El periférico activa las líneas de interrupción de la CPU, la cual detiene el programa en ejecución y carga el contador de programa con la dirección de inicio de la "subrutina de interrupción", creada especialmente para atender al periférico que solicita la interrupción.
- 3.- Mediante el **tratamiento de las E/S como posiciones de memoria**. Permite el empleo de las mismas instrucciones para acceso a memoria que para E/S. Una zona del mapa de memoria es reservada para los registros de los dispositivos de E/S (módulos del microcontrolador). Estas posiciones se denominan Puertos de E/S.

### 1.1.5. Ejemplo de ejecución en un Microprocesador de 8 Bits

La ejecución de una instrucción se lleva a cabo en dos fases:

#### Fase de Búsqueda:

Se inicia en el contador de programa (PC), que contiene la dirección de memoria donde se encuentra el código binario de la instrucción. Esta dirección se coloca en el registro de direcciones de la CPU y de ahí a la memoria a través del bus de direcciones como se muestra en la figura 6 (con color rojo). Una vez decodificada la dirección en la memoria, su contenido se traslada al bus de datos hacia el registro de Instrucciones de la Unidad de Control, como se muestra, (con color Azul), también en la figura 6.

## Fase de Ejecución:

En esta fase se decodifica la instrucción dentro de la unidad de control. Se busca su código de micro-Instrucciones en la memoria interna de la CPU y se activan las señales correspondientes del secuenciador para ejecutar la instrucción completa.

Las instrucciones que constituyen el programa se almacenan en memoria en paquetes de 8 bits (Bytes). El primer Byte indica el Código de operación (**COP ó OPCODE**), que representa la operación que realiza la instrucción y su función.

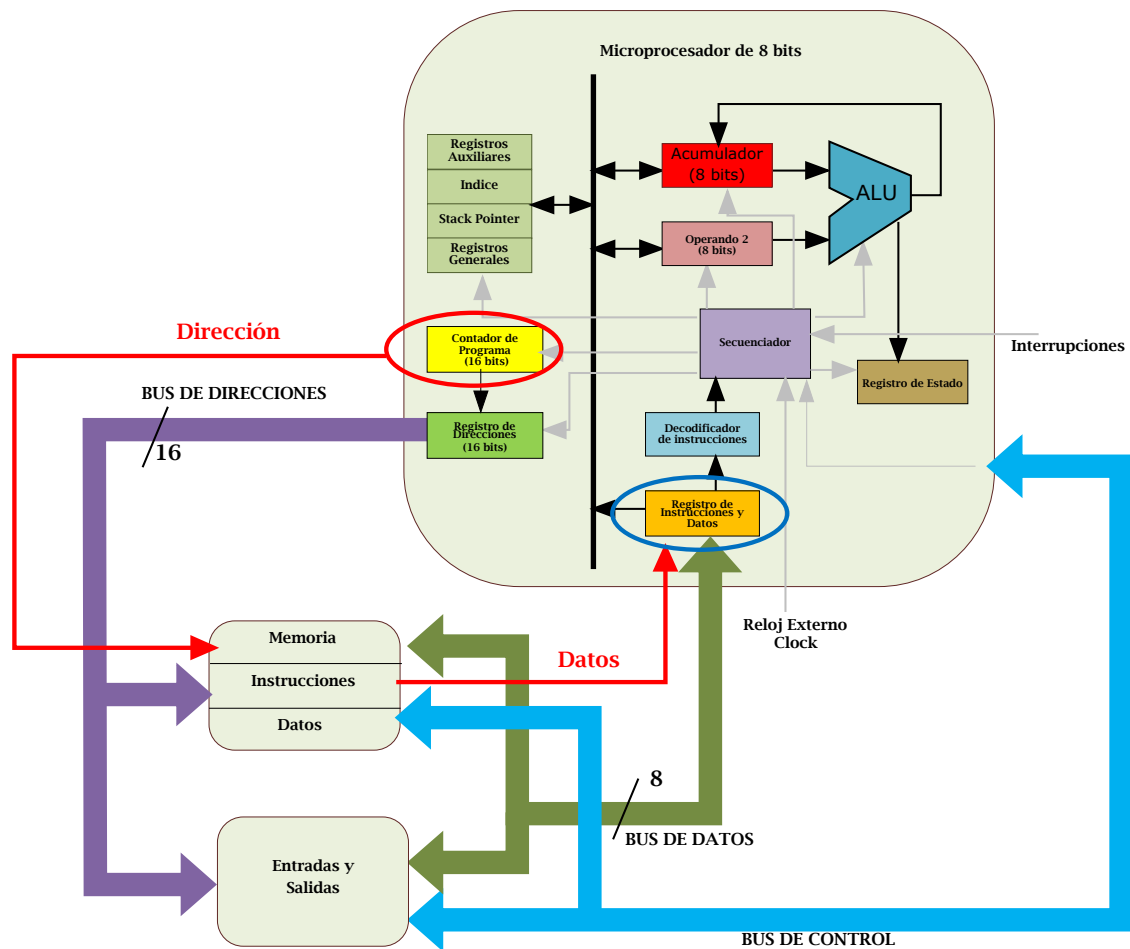


Figura 6. Diagrama General de un Microprocesador de 8 bits.

El o los siguientes Bytes normalmente proporcionan la información necesaria para acceder al dato sobre el que va a operar la instrucción. Estos bytes pueden ser el propio dato, la dirección de memoria donde se encuentra el dato, etc. Las diferentes posibilidades para acceder a ese dato se denominan **modos de direccionamiento** del Microprocesador.

## 1.2. Descripción de Arquitecturas Clásicas

### 1.2.1. Arquitectura Harvard

La arquitectura Harvard almacena instrucciones y datos en memorias separadas como muestra el diagrama de la figura 7. Muchas arquitecturas de microcontroladores contienen la estructura Harvard.

La instrucción se trae a la CPU en un solo acceso a la memoria de programa, mientras tanto el bus de datos está libre y puede accederse a través de él a los datos que se necesitan para ejecutar la instrucción anterior a la que se está trayendo de la memoria de programa en ese momento.

Al tener 2 buses separados, el bus de instrucciones es más ancho que el bus de datos. Esto permite que las instrucciones se codifiquen en palabras de más de 8 bits. Concretamente, la codificación se realiza acorde a los requisitos de la arquitectura.

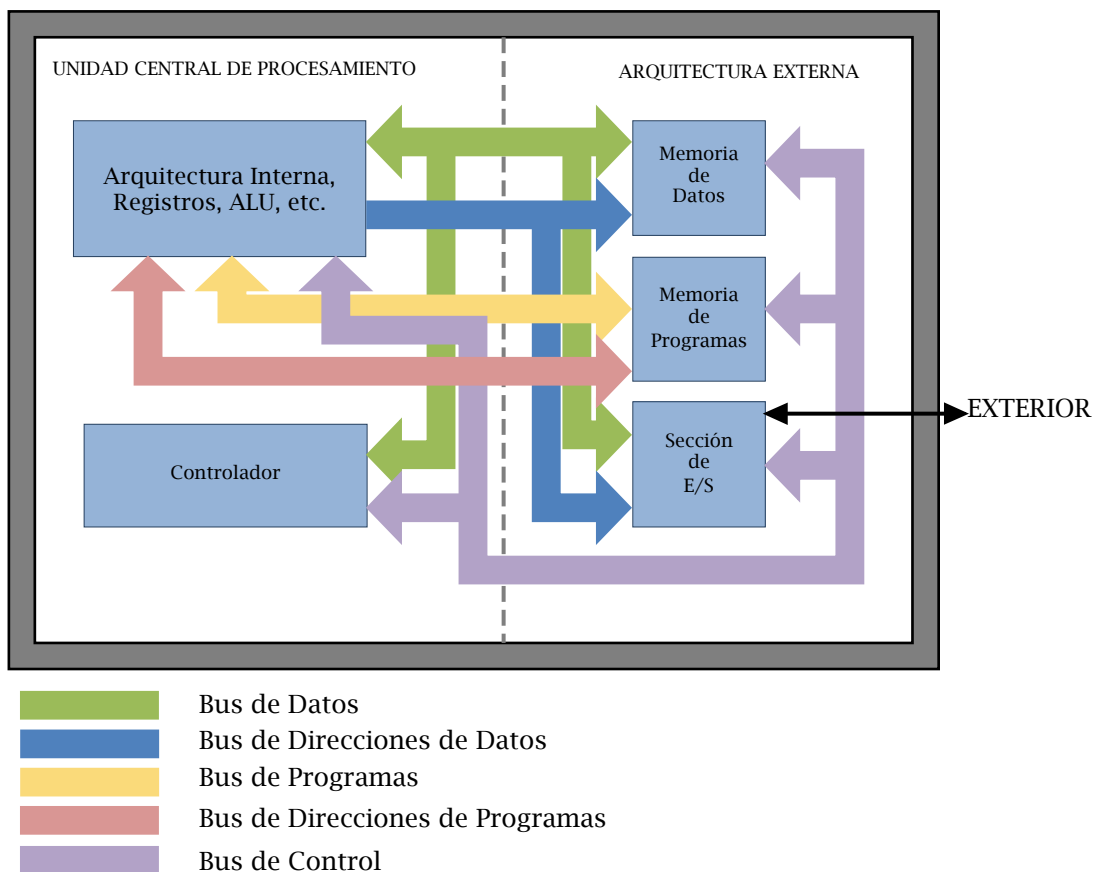


Figura 7. Diagrama en bloques de la Arquitectura Harvard

A su vez, al codificarse en una sola palabra, cada instrucción se trae a la CPU en un único ciclo de instrucción (equivalente a 4 ciclos de reloj).

Al disponer de un bus de memoria de programa, la instrucción se trae a la CPU en un solo ciclo de instrucción. Esta instrucción contiene toda la información requerida y se ejecuta en un solo ciclo.

Durante el ciclo de ejecución de la instrucción el proceso es el siguiente:

La instrucción traída durante el ciclo de instrucción anterior se almacena en el registro de instrucciones (IR) durante el ciclo Q1. La instrucción es decodificada y ejecutada durante los ciclos Q2, Q3 y Q4. Si la instrucción conlleva un acceso a la memoria de datos para lectura, este acceso se realiza durante el ciclo Q2. Si la instrucción conlleva un acceso a la memoria de datos para escritura, este acceso se realiza durante el ciclo Q4.

Esta arquitectura suele utilizarse en DSPs para procesamiento de audio y video.

### 1.2.2. Arquitectura Von Neumann

La característica más destacada de la arquitectura Von Neumann es que almacena datos e instrucciones en una misma memoria como muestra la Figura 8.

Requiere acceso (o varios accesos) a memoria para traer la instrucción. Si esta instrucción maneja datos de memoria, se debe(n) realizar otro(s) acceso(s) para traer, operar y volver a almacenar los datos. El bus se congestiona con tanto acceso. En la arquitectura von Neumann se necesitan habitualmente varios paquetes de 8 bits para codificar una instrucción. Así, por ejemplo, un microcontrolador con 4 Kbytes de memoria de programa podría almacenar 2K instrucciones aproximadamente (a una media de 2 bytes por instrucción, aunque depende de la aplicación).

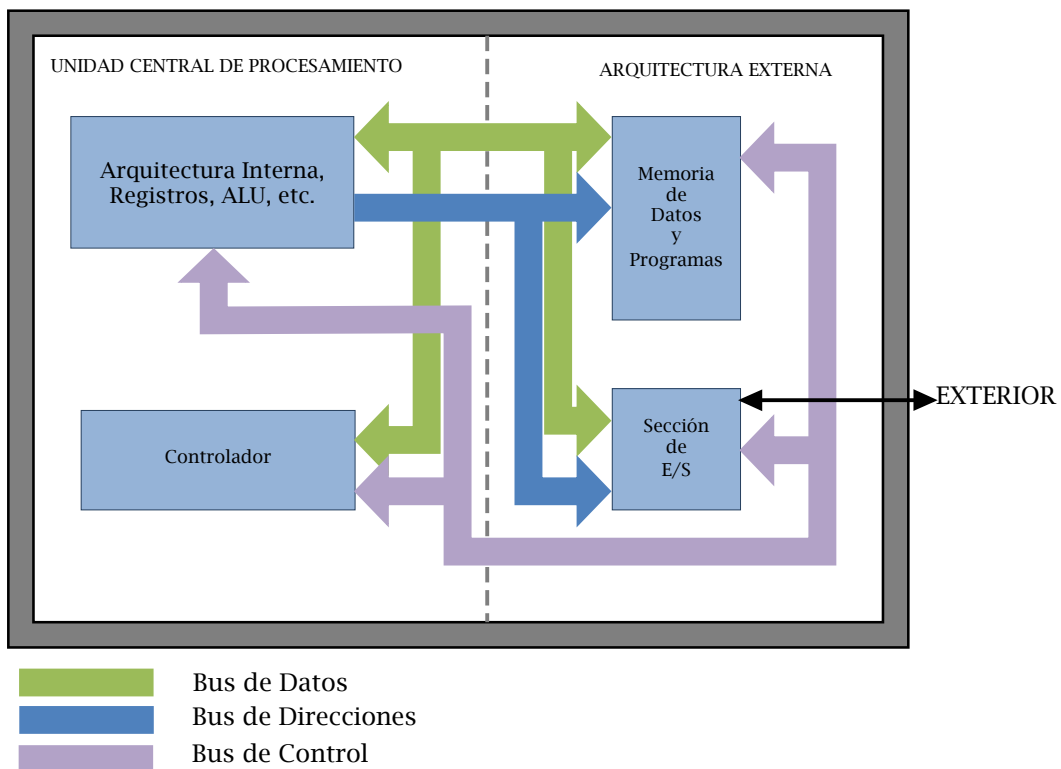


Figura 8. Diagrama en bloques de la Arquitectura Von Neumann.

El canal de transmisión de los datos entre CPU y memoria genera un cuello de botella para el rendimiento del procesador. En la mayoría de las computadoras modernas, la velocidad de comunicación entre la memoria y la CPU es más baja que la velocidad a la que puede trabajar esta última, reduciendo el rendimiento del procesador y limitando seriamente la velocidad de proceso eficaz, sobre todo cuando se necesitan procesar grandes cantidades de datos. La CPU se ve forzada a esperar continuamente a que lleguen los datos necesarios desde o hacia la memoria.

La velocidad de procesamiento y la cantidad de memoria han aumentado mucho más rápidamente que el rendimiento de transferencia entre ellos, lo que ha agravado el problema del cuello de botella.

### 1.2.3. Arquitectura Harvard Vs Arquitectura Von Neumann

Von Neumann:

- Permite almacenar datos e instrucciones en el mismo módulo de memoria
- Más flexible y fácil de implementar
- Adecuado para muchos procesadores de propósito General

Harvard:

- Usa Módulos de memoria separados para almacenar instrucciones y datos
- Es fácilmente implementable el pipeline
- Alto rendimiento de memoria
- Ideal para DSP (Procesadores Digitales de Señales)
- El tiempo de acceso se mejora respecto a la arquitectura von Neumann donde programa y datos se traen a la CPU usando el mismo bus.

### 1.2.4. Arquitectura ARM

#### 1.2.4.1. Introducción

La arquitectura ARM (Advanced RISC Machine) fue creada en 1985 por el Acorn Computer Group, como el primer procesador RISC con gran impacto comercial en el mundo.

La filosofía RISC (Reduced Instruction Set Computer) es buscar la eficiencia tratando de realizar las operaciones de manera más simple. Gracias a su diseño sencillo, el ARM tiene relativamente pocos componentes en el chip, por lo que no alcanza altas temperaturas y tiene bajos requerimientos de energía. Esto último, lo ha hecho el candidato perfecto para el mercado de aplicaciones embebidas (embedded applications).

ACORN se dio cuenta del potencial de esta arquitectura y junto con un grupo de socios-capital, creó una compañía independiente llamada ARM en 1990.

Desde entonces, la arquitectura ARM ha crecido hasta convertirse en la arquitectura más popular del planeta.

En 1987, la arquitectura ARM tuvo su primera aparición en productos comerciales con los asistentes digitales personales Newton de Apple.

En 1995, Digital Semiconductor y ARM, Ltd. crean el StrongARM, que básicamente es un core que utiliza el conjunto de instrucciones de la arquitectura ARM, pero es implementado con la tecnología de la serie Alpha de Digital Semiconductor. De aquí nace el StrongARM de 200 Mhz.

#### 1.2.4.2. Arquitecturas ARM para microcontroladores

La arquitectura ARM ha sido utilizada para el diseño de Microcontroladores estándar de 32 bits. En la actualidad, todos los principales fabricantes de semiconductores usan el robusto núcleo ARM como la base para su línea de microcontroladores. Las principales características de los núcleos ARM son:

- Se basa en los principios de RISC.
- 37 registros de 32 bits (16 disponibles)
- Memoria caché (dependiendo de la aplicación)
- Estructura del bus tipo Von Neuman (ARM7), tipo Harvard ( ARM9)
- Tipos de datos de 8/16/32 bits
- 6 modos de operación: usr y sys, fiq, irq, svc, abt, sys, und.
- Estructura simple = excelente velocidad / bajo consumo de potencia
- Todas las familias de procesadores ARM comparten el mismo conjunto de instrucciones.
- Instrucciones conceptualmente simples.
- Transferencias Memoria/Registros exclusivamente LOAD/STORES.
- Las operaciones aritméticas son entre registros.
- Tamaño de instrucciones uniformes.
- Pocos formatos para las instrucciones.
- Conjunto de instrucciones ortogonal: poco o ningún traslape en la funcionalidad de las instrucciones.
- Pocos modos de direccionamiento.

#### 1.2.5. CISC VS RISC

Hay otras formas de diferenciar Arquitecturas de Computadores. Los Modelos de Arquitecturas son conocidos como RISC (computadora con set de instrucciones reducido) y

CISC (Computadora con set de instrucciones complejas). En la Tabla siguiente se indican las diferencias entre estos dos modelos.

**Tabla 1. Modelo CISC VS Modelo RISC**

Característica	RISC	CISC
Tamaño de instrucción	Una palabra	1 a 54 bytes
Tiempo de ejecución	1 ciclo de clock	De 1 a 100 ciclos
Modos de direccionamiento	pocos	muchos
Tamaño del set instrucciones	Pequeño	Grande

Durante los años 80, se desató una gran controversia sobre las ventajas y desventajas de cada estilo de arquitectura. Como resultado, RISC salió vencedor. De cualquier manera la arquitectura CISC a sobrevivido, en la familia de Intel x86, aunque esta ha adoptado muchas ideas desde el campo del estilo RISC para mantener el buen desempeño.

### 1.3. Características generales de un microcontrolador

Un **Microcontrolador** puede definirse como un sistema integrado por un CPU, memoria, reloj oscilador y módulos I/O en el mismo circuito integrado. Cuando carece de alguno de estos elementos, ya sea I/O o memoria, el circuito integrado lleva el nombre de **Microprocesador** ([10]-[12]). En la Figura 9 se puede observar la Arquitectura del Microcontrolador HC908. Para una mejor comprensión de la forma de manipulación de datos, interacción entre los módulos y los lenguajes de comunicación más utilizados en estos Sistemas se recomienda leer el *apartado 1 del Apéndice*.

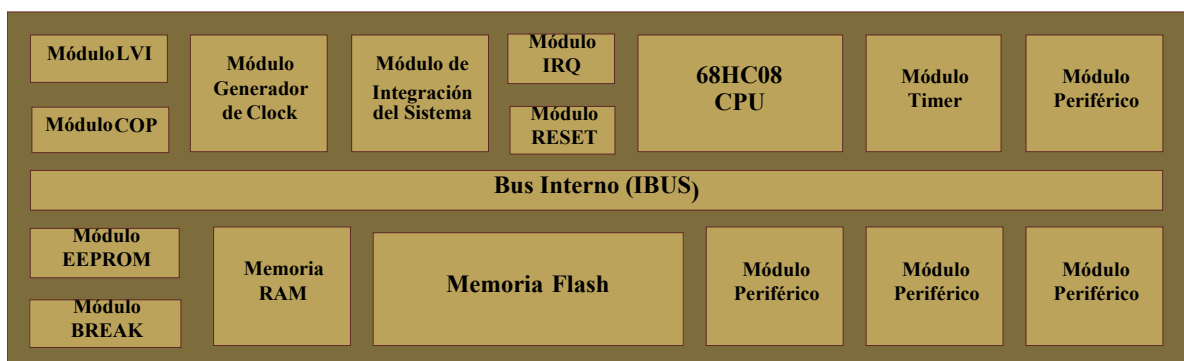


Figura 9. Arquitectura del HC908.

El MC68HC908 es una unidad Microcontroladora (MCUs) de bajo costo, alto desempeño de la familia M68HC08 de 8-bit. Esta familia tiene un conjunto de instrucciones complejas (CISC) con una Arquitectura Von Neumann. Todos los MCUs de la familia M68HC08 usan una unidad Microprocesadora llamada “CPU08” que se describe en el capítulo 2, junto con una amplia variedad de Módulos y un espacio de memoria RAM y Flash que permiten realizar múltiples aplicaciones.

### 1.3.1. Generalidades del microcontrolador HC908

- Posee un poderoso **CPU08** con **más de 100 instrucciones y 16 modos de direccionamiento**.
- Velocidad **Máxima de Bus de 8Mhz (fBus)**
- Memoria de Programa del tipo **FLASH** que permite **programación y reprogramación “En-Circuito”** y uso de esta como **“EEPROM”** para el almacenamiento “no-volátil” de datos temporales.
- **Conversores A/D y TIMERS** flexibles y poderosos en todos los distintos dispositivos de la familia HC908.
- **LVI** (Low Voltage Inhibit) (supervisor de baja tensión) incorporado en todos los Microcontroladores de la familia HC908.
- Los microcontroladores de aplicación específica también poseen múltiples y prácticos periféricos como generadores de **PWM**, **Módulos analógicos**, **sensores de temperatura internos**, **I<sup>2</sup>C**, **SPI**, **UART**, etc.
- Un **amplio conjunto de herramientas de desarrollo**, ya sea para programar en lenguaje C o en Assembler o para programar o simular en circuito el código desarrollado.

### 1.3.2. Características de los microcontroladores HC908QT/QY

En la Figura 10 se muestra el diagrama en bloques de la línea de microcontroladores HC908QT/QY, cuyas características se describen a continuación [3]:

- Core M68HC08 CPU de alta performance
- Voltajes de operación de 5-V y 3-V (VDD)
- Bus interno de 8-MHz a 5 V, 4-MHz a 3 V
- Oscilador interno ajustable <sup>1</sup>
  - Operación de bus interno a 3.2 MHz
  - 8-bit de ajuste (trimmer) que permite 0.4% exactitud
  - $\pm 25\%$  sin ajustar

<sup>1</sup> La frecuencia del oscilador está garantizada a  $\pm 5\%$  a un rango de temperatura y voltaje luego del ajuste.

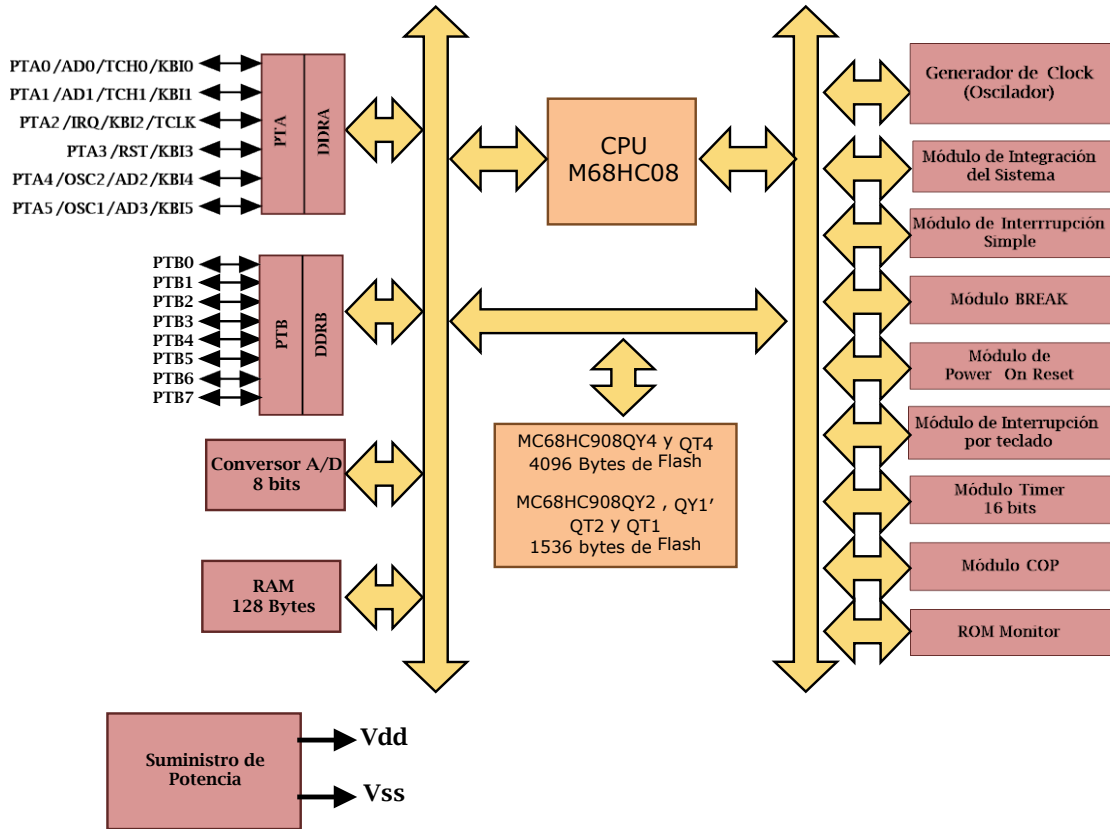


- Auto wake-up desde STOP
- Registro de Configuración (CONFIG) para opciones de configuración del MCU, incluyendo:
  - Inhibición por baja tensión (LVI)
- Programación en sistema de FLASH
- Seguridad de FLASH<sup>2</sup>
- Memoria Flash programable On-chip en la aplicación
  - MC68HC908QY4 y MC68HC908QT4 tiene 4096 bytes de flash
- 128 bytes de memoria RAM on-chip
- 2-canales, Módulo de interfaz temporizador de 16-bit (TIM)
- 4-canales, Conversor Analógico Digital de 8-bit (ADC) sobre MC68HC908QY2, MC68HC908QY4
- 5 o 13 líneas de entradas/salidas bidireccionales (I/O) y una entrada:
  - 6 compartidas con funciones de interrupción por teclado y ADC
  - 2 compartidas con canales de timer
  - Una compartida con interrupción externa (IRQ)
  - 8 líneas con funcionalidad I/O únicamente, sobre encapsulado de 16 pines
  - Capacidad de alta corriente de drain/source en todos los pines del puerto
  - Pull-ups seleccionables sobre todos los puertos.
  - Posibilidad de asignación de 3- estados sobre todos los pines de los puertos
- Interrupción por teclado de 6-bit con característica de wake-up (KBI)
- Módulo de Inhibición por bajo voltaje (LVI), consiste en la supervisión de baja tensión.
- Características de protección del Sistema:
  - Módulo de Operación correcta del Computador (COP) mediante el uso de watchdog
  - Detección de Baja Tensión con reset
  - Detección de Código de Operación ilegal con reset
  - Detección de dirección ilegal con reset
- Pines de Interrupción externa asincrónica (IRQ) con pull-up internos
- Pin de Reset asincrónico maestro (RST) compartido con pines de entrada/salida (I/O)
- Power-on reset
- Pull-ups internos sobre IRQ y RST para reducir la cantidad de componentes externos
- Ahorro de energía en modo wait y stop
- 16 modos de direccionamiento
- Registro índice y puntero de pila de 16 bits
- Transferencia de datos memoria a memoria
- Instrucción rápida de multiplicación de tamaño de datos de  $8 \times 8$
- Instrucción rápida de división de tamaño de datos de 16/8
- Instrucciones de Código Binario codificado a Decimal (BCD)
- Soporta lenguaje C eficiente

---

<sup>2</sup> La característica de seguridad es inviolable. Esto se hace para evitar copias de terceros.

Este tipo de Microcontroladores viene en varios tipos de encapsulado, PDIP, SOIC, TSSOP y DFN, el encapsulado y la distribución de pines del integrado se muestra en la Figura 11. La tabla 2 muestra la descripción completa de los pines de entrada salida de la línea de Microcontroladores HC908QT4/QY4.



RST, IRQ: Tienen pull up internos (30K Ohms)  
 PTA[0:5]: Capacidad de alto suministro de corriente  
 PTA[0:5]: Tienen interrupción por teclado programables y pullup

Figura 10. Diagrama en Bloques específico de los Microcontroladores HC908QTQY.

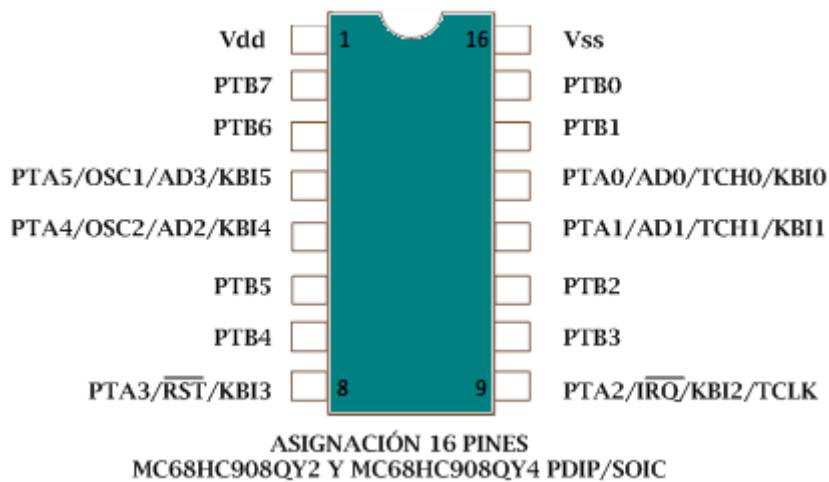


Figura 11. Encapsulado y distribución de pines.

**Tabla 2. Descripción de los pines de entrada/salida de la Línea HC908QT/QY4**

Nombre del Pin	Descripción	Entrada / Salida
VDD	Suministro de Potencia	Power
VSS	Tierra	Power
PTA0	PTA0 — puerto de entrada/ salida de propósito general	Entrada/salida
	AD0 — A/D canal 0 – entrada	entrada
	TCH0 — Timer Canal 0 I/O	Entrada/salida
	KBI0 — interrupción por teclado entrada 0	entrada
PTA1	PTA1 — puerto de entrada/ salida de propósito general	Entrada/salida
	AD1 — A/D canal 1 - entrada	entrada
	TCH1 — Timer Canal 1 I/O	Entrada/salida
	KBI1 — interrupción por teclado - entrada 1	entrada
PTA2	PTA2 — puerto de entrada de propósito general	entrada
	IRQ — interrupción externa con pullup programable y entrada de Schmitt trigger	entrada
	KBI2 — interrupción por teclado - entrada 2	entrada
	TCLK — Timer clock - entrada	entrada
PTA3	PTA3 — puerto de entrada/ salida de propósito general	Entrada/salida
	RST — entrada de reset, activo en bajo con pullup interno y Schmitt trigger	entrada
	KBI3 — interrupción por teclado entrada 3	Entrada
PTA4	PTA4 — puerto de entrada/ salida de propósito general	Entrada/salida
	OSC2 —salida de oscilador XTAL (solo para XTAL) RC o salida de oscilador interno (OSC2EN = 1 en registro PTAPUE)	salida
	AD2 — A/D canal 2 – entrada	entrada
	KBI4 — interrupción por teclado entrada 4	entrada
PTA5	PTA5 — puerto de entrada/ salida de propósito general	Entrada/salida
	OSC1 — XTAL, RC, o entrada de oscilador externo	entrada
	AD3 — A/D canal 3 – entrada	entrada
	KBI5 — interrupción por teclado entrada 5	entrada
PTB[0:7]	8 puerto de entrada/ salida de propósito general	Entrada/salida

### 1.3.3. Ventajas adicionales

- Costos de programación reducidos en cuanto a tiempo.
  - 100 veces más rápidos que los de otros MCU's con FLASH u OTP's (2mseg. Para 64 Bytes Vs múltiples mseg. / byte).
- Puede utilizarse como memoria de almacenamiento de Datos temporales.
  - 10.000 ciclos de escritura / borrado en la peor condición de Temp.
  - + de 100.000 ciclos a temperatura ambiente (+ 20 °C a + 30 ° C).
- Programación garantizada a lo largo de un amplio rango de tensiones.
- Bloque de protección y seguridad flexible
  - Seguridad contra lecturas no autorizadas por PASSWORD.
  - Protección anti-grabación por bloques flexibles y seguros.

### 1.3.4. Módulos disponibles en la familia HC908

Si bien en la Cátedra se programa una sola línea de microcontroladores, es importante conocer todas las líneas de microcontroladores disponibles dentro de una Familia y en qué casos conviene utilizar cada línea. La Familia HC908 de Freescale (ver [1] y [2]), posee microcontroladores de aplicación específica para control, comunicaciones y procesamiento digital con una gran variedad de módulos adicionales que los hacen aptos para dichos casos, esto está indicado implícitamente en la línea a la que pertenece cada MCU.

- Módulo de 2, 3, y 4 canales de Timers Programables.
- PWM dedicado de 8 bits y 16 bits (Modulación por ancho de pulso) (línea MR).
- Conversores A/D (8-bits y 10-bits) (línea MR,SR,AP,QL).
- Sensores de Corriente y Temperatura (línea SR).
- Comparadores (línea SR).
- EEPROM (línea AB32).
- I2C (línea SR, AP).
- 2 Módulos UART (SCI), 1 con modulador / demodulador de INFRARROJO (línea AP).
- Control de DISPLAY LCD NO INTELIGENTE (línea LD).
- Osciladores internos (línea QT / QY / QL).

# CAPÍTULO 2

## Descripción del CPU08

*Jorge R. Osio, Walter J. Aróztegui y José A. Rapallini*

El CPU 08 [5] (Figura 12) es el “Core” de los MCU’s de la flia. HC908. Dentro de la estructura interna de un HC908, el módulo del CPU se vincula con el resto de los módulos del MCU por medio de un bus de datos interno de 8 bits, y un bus de direcciones de 16 bits, que le permite direccionar código de hasta 64K bytes. Este Bus es denominado IBUS (Bus Interno). La frecuencia máxima del BUS interno es de 8 MHZ reales a 5 Volts de alimentación y 4 MHZ a 3 Volts. Esta frecuencia de Bus implica que cada ciclo de Clock (reloj) del Bus es de 125 nS.

El HC908 es un MCU del tipo “cerrado” (single chip) y por lo tanto no se tiene acceso a los BUSES internos del mismo.

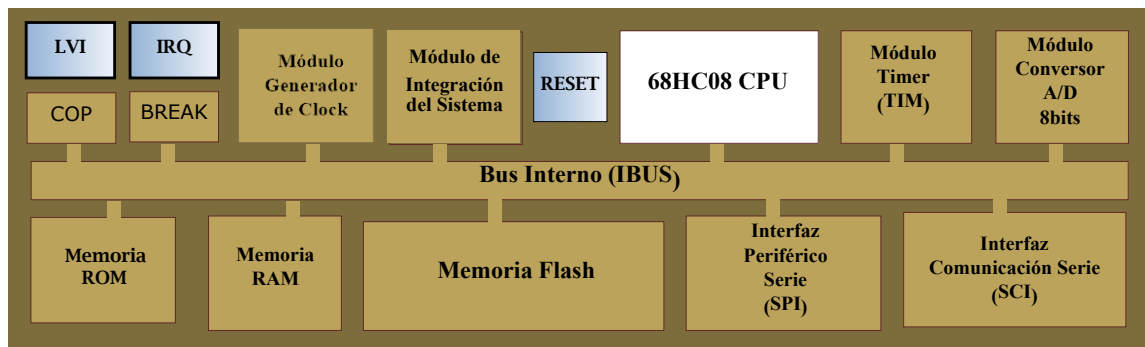


Figura 12. Esquema del CPU 08.

### 2.1. Arquitectura de ejecución del CPU 08

El CPU 08 está dividido en dos bloques; la “**Unidad de Control** (Control Unit)” y la “**Unidad de Ejecución** (Execution Unit)”, ver Figura 13.

El primer bloque contiene una Máquina de Estados Finitos, unidades de Control y temporización, para manejar la “unidad de Ejecución”. Además, dos señales de la Unidad de Control manejan la pre búsqueda “prefetch” y la carga de instrucciones, una es “Opcode Lookahead” (Señal para la operación de pre búsqueda) y la otra es Lastbox (Señal para el último ciclo de la instrucción en curso).

El segundo bloque contiene la ALU (Unidad Aritmético Lógica, encargada de todas las operaciones lógicas binarias y aritméticas), Registros internos de CPU (Acumulador, Puntero de pila, Contador de Programa, Registro índice y Registro de Código de Condiciones CCR) y la interfaz con el bus interno.

El CPU 08 pertenece a la arquitectura del tipo “Von Neumann” clásica ([10] - [12]), característica de la familia 68xx de Motorola y ampliamente utilizada en el mundo, en la sección 1.2 se describieron las arquitecturas clásicas más utilizadas en el diseño de Microcontroladores. En este tipo de arquitectura, existe un solo Bus de datos, tanto para memoria de programas como para memoria de datos, lo que da origen a un mapa “lineal” de acceso a memoria y por consiguiente no existen instrucciones especiales y diferentes para trabajar con “DATOS” o con “código de programa”. De esta forma, todas las instrucciones son aplicables en cualquier parte del mapa de memoria sin importar si se trabaja con datos o código. Por lo que no es raro encontrar aplicaciones cuyos programas se ejecutan en la RAM como si estuvieran en Flash. Se debe destacar que esto no podría hacerlo una arquitectura Harvard clásica.

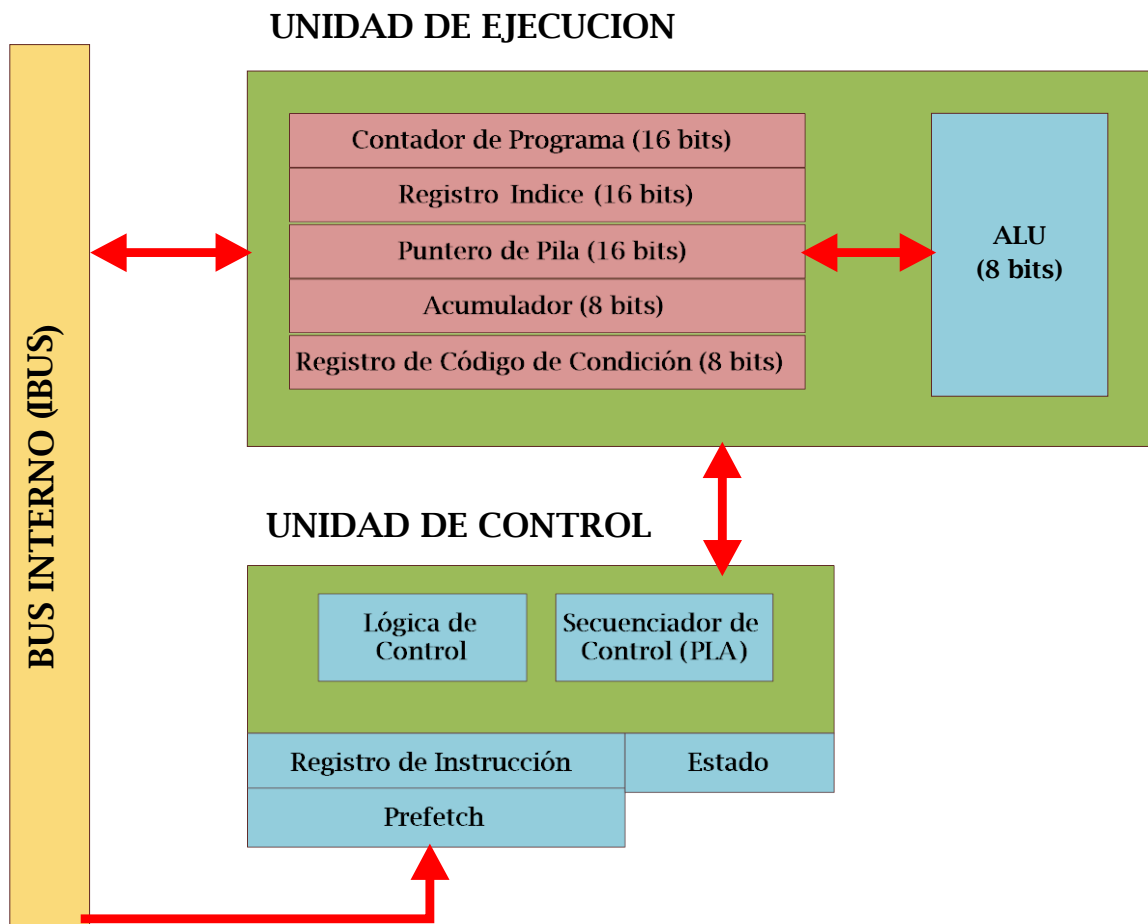


Figura 13. CPU 08 - Arquitectura de Ejecución.

## 2.2. Temporización Interna

El clock del CPU08 necesita de un clock de 4 fases, (identificadas como T1, T2, T3, T4 en la Figura 14), para ejecutar un ciclo de máquina. Por ello, en el HC908 se necesita un XTAL (Oscilador Externo) 4 veces superior a la frecuencia de Bus deseada. Por ejemplo, para trabajar con una frecuencia de bus de 8Mhz, se necesitará un cristal de 32Mhz. El ciclo de Bus del CPU consiste de 1 pulso de clock desde cada fase, es decir, cuatro pulsos de clock del cristal. Para simplificar, en la figura siguiente el periodo T de clock se ha unificado en una señal llamada Clock del CPU. El inicio de un ciclo de CPU se define como el flanco principal de T1 y la dirección de memoria asociada con este ciclo se introduce al bus de direcciones recién en el flanco ascendente del tercer pulso de clock T3. Se puede observar que la nueva dirección de memoria accedida adelanta al dato asociado en medio ciclo de Bus.

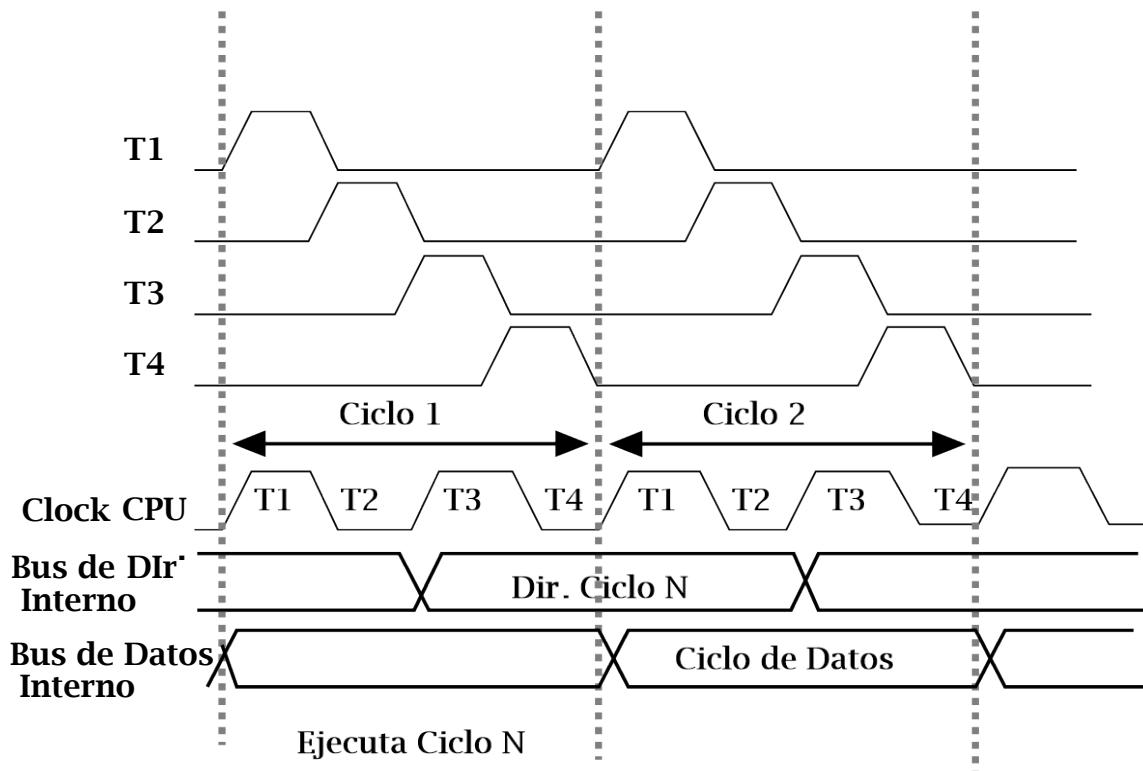


Figura 14. Temporización Interna.

Por ejemplo, el dato leído, asociado con la nueva dirección contenida en el registro contador de programa, generado en T1/T2 del ciclo 1, no debería ser leído en la unidad de control hasta T2 del próximo ciclo.

### 2.3. El “Prefetch” en el CPU08

El procesador CPU08 contiene un “Código de Operación” con un mecanismo de Pre búsqueda (Prefetch) hacia adelante. Esto incrementa el desempeño eliminando tantos “ciclos muertos” de Bus como sea posible, de esta forma se obtienen instrucciones con menor número de ciclos de reloj, mejorando la velocidad real de ejecución de código.

El flujo de ejecución de instrucciones del CPU08 fue desarrollado para ser tan eficiente como sea posible en una estructura del tipo “pipeline”, y gracias a esta estructura es que se consume la menor cantidad posible de ciclos de máquina en la ejecución de las distintas instrucciones del CPU08. Es por esta razón, que la mejora total de desempeño en los HC908 es 5 (cinco) veces superior a la de las familias anteriores.

### 2.4. Unidad de control

La unidad de control está formada por un secuenciador, un bloque de control de almacenamiento y lógica de control aleatorio.

Estos bloques forman una máquina finita de estados que genera las señales de control que se envían hacia la unidad de ejecución.

El secuenciador provee el siguiente estado de la máquina en función de la instrucción contenida en el registro de instrucción (IR) y del estado actual de la máquina. El control de almacenamiento será “strobed” (habilitado) cuando la entrada del próximo estado esté estable, produciendo una salida que representa el siguiente estado decodificado, el cual es enviado hacia la unidad de ejecución (EU). Este resultado, con la ayuda de lógica aleatoria, se usa para implementar la unidad de control que configura la unidad de ejecución. La lógica aleatoria selecciona la señal apropiada y agrega tiempos a la salida del control de almacenamiento. La unidad de control emite una señal por ciclo de bus, pero utiliza casi un ciclo completo antes de la unidad de ejecución para decodificar y generar todos los controles para la ejecución de la próxima instrucción. La secuencia natural de la máquina de estados se muestra en la Figura 15.

El secuenciador también contiene y controla el registro opcode lookahead, que es usado para pre cargar la próxima instrucción a ejecutar.



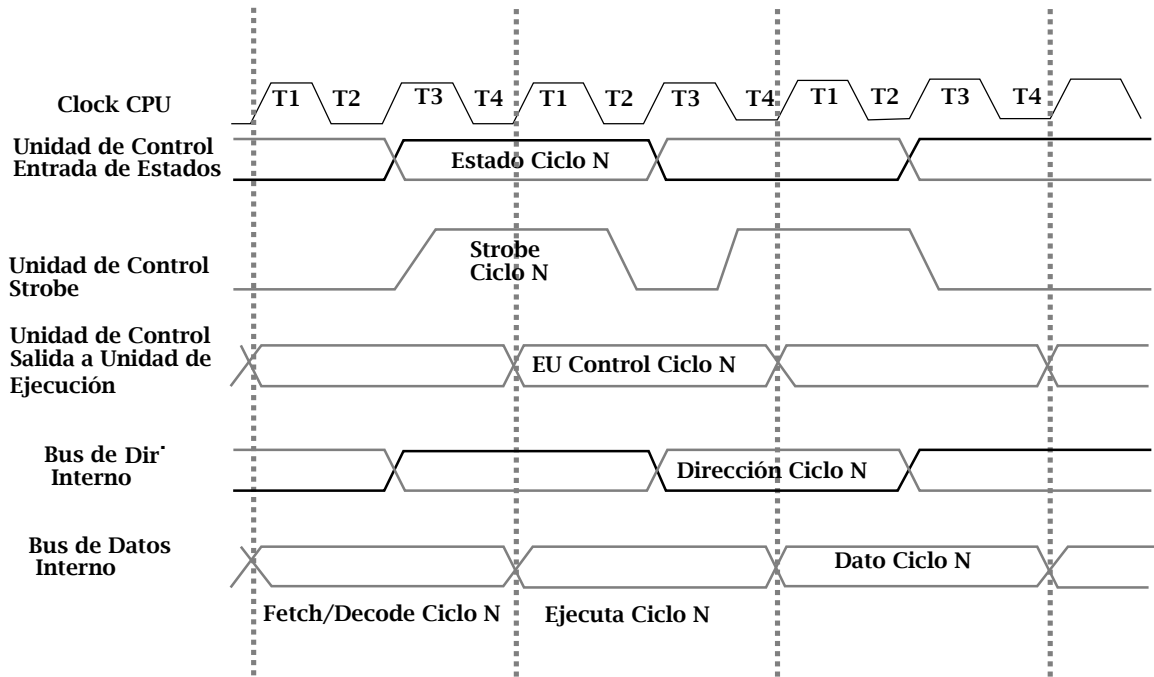


Figura 15. Temporización en la Unidad de Control

## 2.5. Unidad de Ejecución

La unidad de ejecución (UE) contiene todos los registros, la unidad aritmético lógica (ALU), y el bus de interfaz.

Por cada ciclo de bus se calcula una nueva dirección pasando valores por el registro seleccionado a lo largo de los buses de direcciones internos hasta los buffers de direcciones. La unidad de ejecución también contiene algunas funciones lógicas especiales para instrucciones más complejas como DAA, multiplicación sin signo (MUL), y división (DIV).

## 2.6. Ejecución de Instrucciones

Los pasos de ejecución de una instrucción comienzan en el registro contador de programa, donde se encuentra la dirección de la siguiente instrucción a ejecutar. Dicha dirección se traslada al registro de direcciones, quien la deposita en el bus de direcciones para indicar a la memoria que se quiere leer el contenido de dicha dirección. El contenido de la dirección \$00 corresponde a la instrucción LDA (OP CODE \$A6), el cual es depositado en el bus de datos y trasladado al registro de instrucción, como se muestra en la figura 16. A partir de este registro comienza el proceso de ejecución de la instrucción, este código es interpretado por el bloque decodificador de instrucciones y transformado en las señales de control, las cuales serán emitidas por el secuenciador. Las señales de control generadas serán las siguientes:

- Se genera una señal que depende del modo de direccionamiento de la instrucción LDA, en este caso el modo de direccionamiento es inmediato, lo que significa que el operando se encuentra en la dirección \$01

- Luego se genera la señal que obtiene la dirección del operando del contador de programa y la traslada al bus para solicitar el contenido de esa dirección a la memoria
- Por último se deposita el operando, (número \$7), en el registro de instrucciones, para luego ser trasladado al registro acumulador según la función de la instrucción LDA.

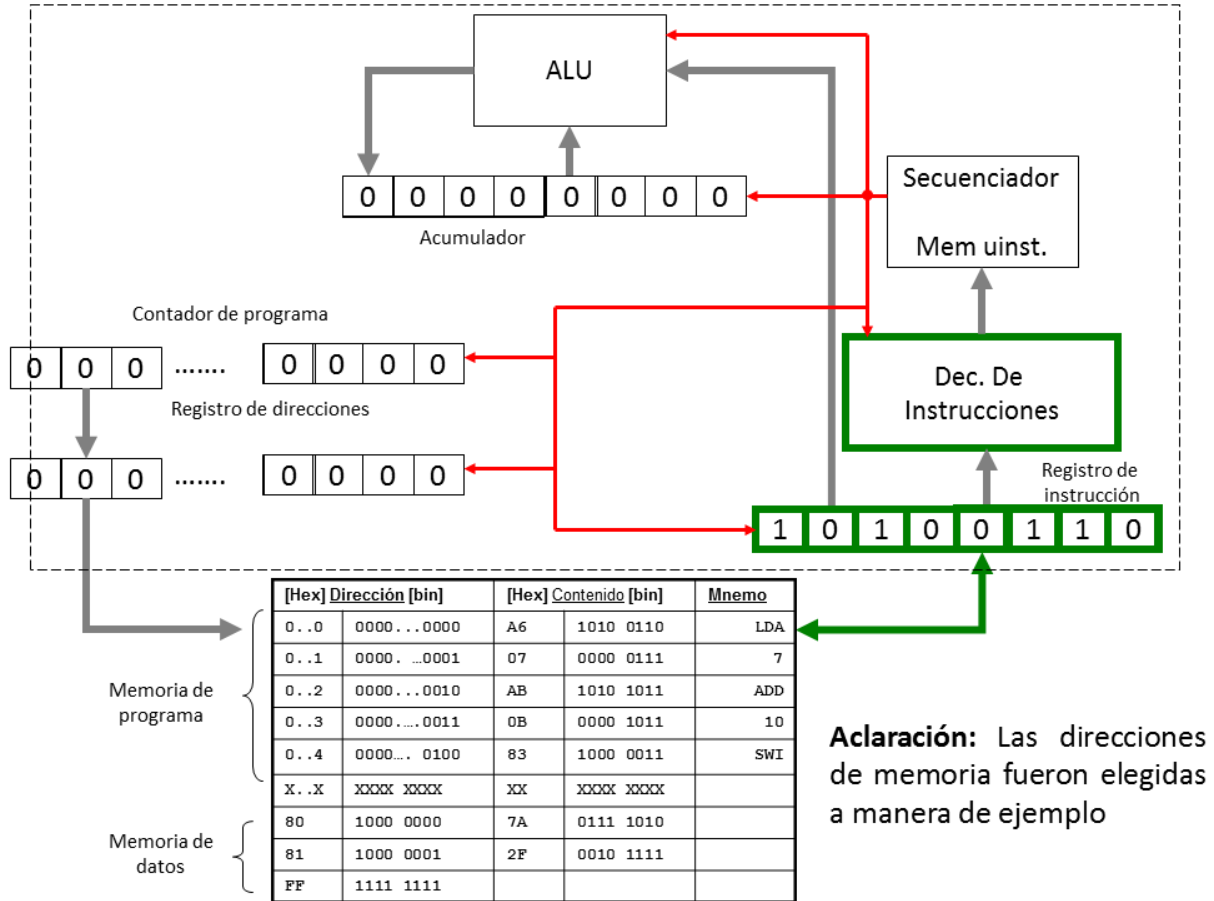
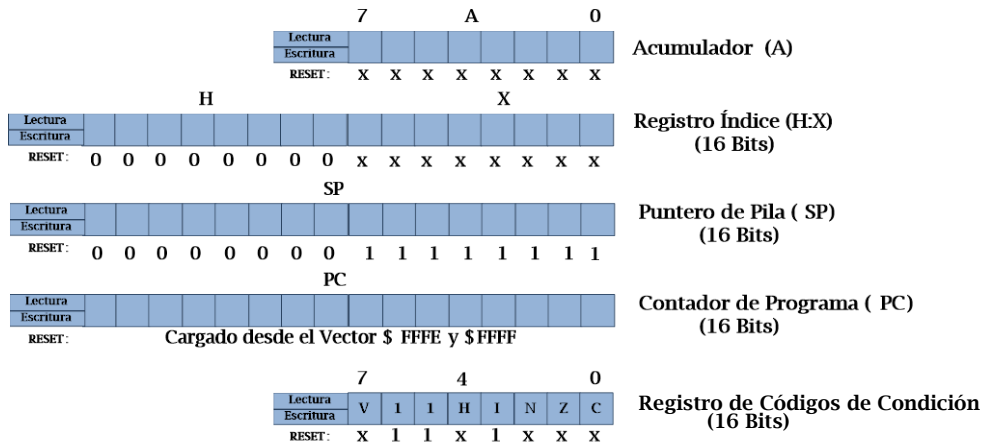


Figura 16. Ejecución de un conjunto de instrucciones

Al igual que en la ejecución de la instrucción LDA, la ejecución de la instrucción ADD, ubicada en la dirección \$02, será similar solo que en este caso el secuenciador emitirá señales de control hacia la ALU para indicar que se deberá realizar la operación de suma.

## 2.7. Registros de la CPU (Modelo de Programación)

La CPU [5] contiene cinco registros tal como se lo presenta en la Figura 17. Los registros de la CPU son registros de memoria que se alojan dentro del microprocesador (no son parte del mapa de memoria).



X = Indeterminado

Figura 17. Registros del CPU08.

### 2.7.1. Acumulador (A)

El acumulador (Figura 18) es un registro de propósitos generales de 8 bits usado para almacenar operandos, resultados de cálculos aritméticos, y de manipulación de datos. Además, es directamente accesible desde la CPU para operaciones no aritméticas. El acumulador es usado durante la ejecución de un programa, cuando el contenido de alguna posición de memoria es cargado en el acumulador. También, la instrucción almacenar (sta) causa que el contenido del acumulador sea almacenado en alguna posición de memoria preestablecida.

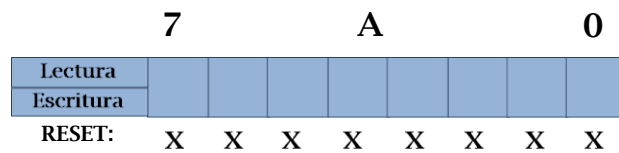


Figura 18. Registro Acumulador.

### 2.7.2. Registro Índice (H:X)

El registro índice (Figura 19) es el registro utilizado para todas las operaciones indexadas (con y sin offset) que posee el CPU. Tiene 16 bits de longitud, formado por una parte "baja" (el byte de menor peso) denominado "X" y una parte alta (el byte de mayor peso) denominado "H". Estos registros se encuentran concatenados para formar un único registro H:X. Esto permite direccionamientos indexados de **hasta 64 Kbytes** de espacio de memoria.

En el registro Índice puede utilizarse la parte baja ("X"), en los distintos modos de direccionamiento. Solo se debe tener en cuenta que cuando en una instrucción con direccionamiento indexado, se menciona el registro "X", en realidad se está haciendo mención

al registro concatenado H:X de 16 bits de largo, por lo que deberá ponerse a cero (forzar el valor \$00) la parte superior del registro índice, o sea "H. De esta forma, cuando se utilice el registro índice, el contenido del mismo será \$00xx, donde "xx" contendrá el valor del registro "X" propiamente dicho.

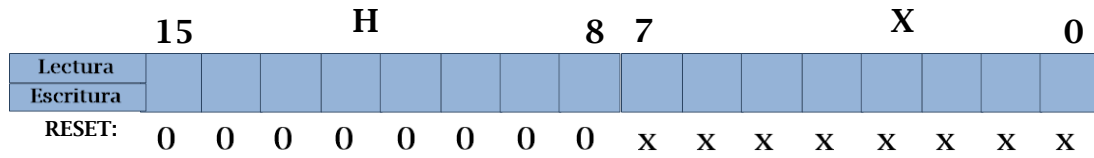


Figura 19. Registro Índice.

### 2.7.3. Puntero de Pila (SP)

El puntero de pila (SP) como se observa en la Figura 20, es un registro de 16 bits que contiene la dirección del próximo lugar en la pila (Stack). Es el registro utilizado por el CPU para mantener en "orden" (guardar y rescatar los datos en RAM) los registros principales del CPU ante una excepción en la secuencia del programa, como lo son los saltos a sub-rutinas y los distintos pedidos de interrupción.

Durante un Reset, el puntero de pila, es pre seteado a \$00FF. La instrucción Reset Stack Pointer (RSP), cambia el byte menos significativo a \$FF y no afecta al byte más significativo.

El puntero de pila es decrementado cuando un dato es almacenado (Push) dentro de la pila e incrementado cuando un dato es recuperado (Pull) desde la pila. La localización de la pila es arbitraria, y puede ser "re-ubicada" en cualquier parte de la memoria RAM. Moviendo el puntero fuera de la página cero o página directa (\$0000 a \$00FF) se libera el espacio del direccionamiento directo. Para una operación correcta, el puntero de pila debe apuntar solamente a posiciones de RAM, aunque por su longitud, pueda "barrer" todo el espacio de memoria del MCU. Gracias a esta característica, en los modos de direccionamiento con el SP (Stack Pointer) con 8 bits de offset y 16 bits de offset, el puntero de pila (SP) puede funcionar como un segundo registro índice de 16 bits o bien para acceder a datos en la pila. El uso del SP como un segundo registro índice, es muy utilizado en los compiladores de lenguaje de alto nivel como los compiladores "C" y otros.

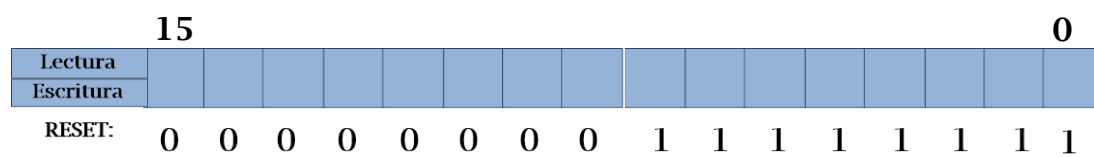


Figura 20. Puntero de pila.

### 2.7.4. Contador de Programa (PC)

El Contador de Programa (PC), es el registro utilizado por el CPU para mantener el control de las direcciones de las próximas instrucciones a ser ejecutadas. Este registro tiene una longitud de 16 bits, y tiene la ventaja de poder moverse entre \$0000 y \$FFFF. De esta forma, el PC puede moverse por los 64 Kbytes de espacio de memoria (salvo en los Microcontroladores que tienen memoria menor a 64 Kbytes).

Durante el Reset, el contador de programa (PC) se carga con la dirección contenida en el "vector de reset" que para el MC68HC908 se encuentra en las posiciones \$FFFE y \$FFFF. La dirección contenida en el vector, es la dirección de la primera instrucción a ser ejecutada después de salir del estado de RESET.



Figura 21. Contador de Programa.

### 2.7.5. Registro de Código de Condición (CCR)

El registro de código de condición contiene una máscara de interrupción y cinco indicadores de estado que reflejan el resultado de operaciones aritméticas y de otro tipo, efectuadas por el CPU con anterioridad.

Los cinco flags (banderas) son, Desborde "overflow" (V), semi-acarreo (H), máscara de interrupción (I), negativo (N), cero (Z) y acarreo / préstamo (C).

Luego de un reset los flags tomarán los siguientes valores: V = x; H = x; I = 1; N = x; Z = x; C = x; donde x es un valor indeterminado.

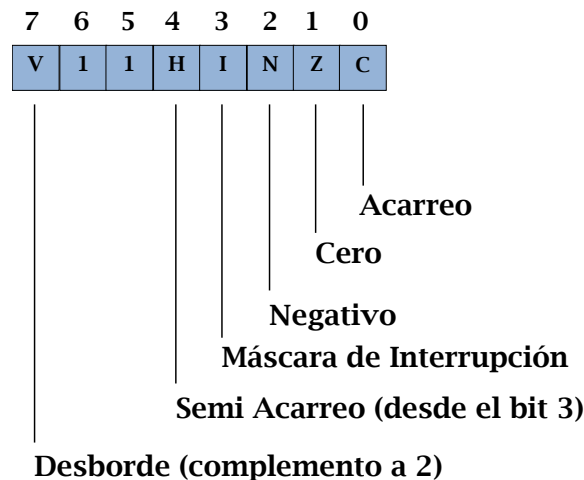


Figura 22. Registro de código de condición.

#### **2.7.5.1. Flag V (Bit de Desborde)**

Es utilizado en “chequeos” de operaciones aritméticas signadas. El CPU pone en 1 el bit de desborde cuando ocurre un desborde por complemento a Dos en una operación aritmética con signo. Las instrucciones de salto condicionales signados como BGT, BGE, BLE, y BLT usan este bit de desborde.

#### **2.7.5.2. Flag H (semi-acarreo)**

El bit es “Seteado” si un carry (acarreo) ocurre desde el bit 3 al bit 4, se utiliza en operaciones aritméticas BCD.

#### **2.7.5.3. Flag I (Mascara Global de Interrupciones)**

Cuando está seteado, deshabilita las interrupciones del CPU.

#### **2.7.5.4. Flag N (Negativo)**

Es seteado, si el bit 7 está seteado en el Acumulador.

#### **2.7.5.5. Flag Z (cero)**

Este bit es seteado si todos los bits en el Acumulador son ceros.

#### **2.7.5.6. Flag C (acarreo / préstamo)**

Seteado si se produce un carry o borrow durante una operación.

# CAPÍTULO 3

## Modos de Direccionamiento

*Jorge R. Osio y Walter J. Aróztegui*

Los modos de direccionamiento de la CPU proveen la capacidad de acceder a los operandos que necesitan las instrucciones por diferentes caminos. Los **modos de direccionamiento** [4], [11] indican la manera en que una instrucción obtendrá el dato requerido para su ejecución.

La CPU del MC68HC908 [3] usa siete modos de direccionamiento para acceder a memoria, estos son: inherente, inmediato, extendido, directo, indexado (sin desplazamiento, con desplazamiento de 8 bits, con desplazamiento de 16 bits, con Stack pointer y 8 bit de desplazamiento, con stack pointer y 16 bits de desplazamiento, sin desplazamiento con post incremento, con desplazamiento de 8 bit y post incremento ), relativo y de memoria a memoria (inmediato a directo, directo a directo, indexado con post incremento a directo, directo a indexado con post incremento) .

En los pequeños Microcontroladores MC68HC908, todas las variables del programa y los registros de I/O se encuentran en el área de memoria que va de \$0000 a \$00FF donde el modo de direccionamiento más comúnmente usado es el direccionamiento directo.

En los siguientes párrafos se provee una descripción general y ejemplos de los distintos modos de direccionamiento. Se debe aclarar que el término **dirección efectiva** es usado para indicar la dirección de la posición de memoria donde el argumento para una **instrucción** es buscado o almacenado. En el apéndice B se muestran ejemplos de uso de los distintos tipos de instrucciones en los distintos modos de direccionamiento.

Para cada modo de direccionamiento se explica en detalle una instrucción. Esta explicación describe qué sucede en la CPU durante cada ciclo de reloj del procesador. En estos ejemplos, los números de secuencia entre paréntesis hacen referencia a un ciclo de reloj específico.

### 3.1. Modo de Direccionamiento Inmediato

Especifica el valor del operando directamente a continuación del código de operación de la instrucción. Para indicar un valor en una instrucción se antepone el símbolo # al operando en cuestión. Este modo de direccionamiento tiene un solo operando que está contenido en el byte o los bytes seguidos inmediatamente al código de operación. Este modo es usado cuando hay

un valor o constante conocido al momento de escribir el programa que no cambiará durante la ejecución del programa. Esta es una instrucción de dos bytes, uno para el código de operación y otro para el byte que representa al operando.

**Ejemplo:**

**Código de operación:** A6

**Operando:** FF

**Formato de instrucción en Assembler:** LDA #\$FF; cargar el Acumulador con el valor inmediato FF. En la Figura 23 se representa el espacio de memoria en la parte izquierda de la figura y el contenido del registro A (acumulador) dentro de la CPU.

**Secuencia de Ejecución:**

Dirección	Contenido
\$EE00	\$A6 (1)
\$EE00	\$FF (2)

**Explicación:**

- (1) La CPU lee el código de operación \$A6 - Carga el acumulador con el valor en la posición de memoria siguiente al código de operación.
- (2) La CPU lee el dato inmediato \$FF de la posición de memoria \$EE01 y lo carga en el acumulador.

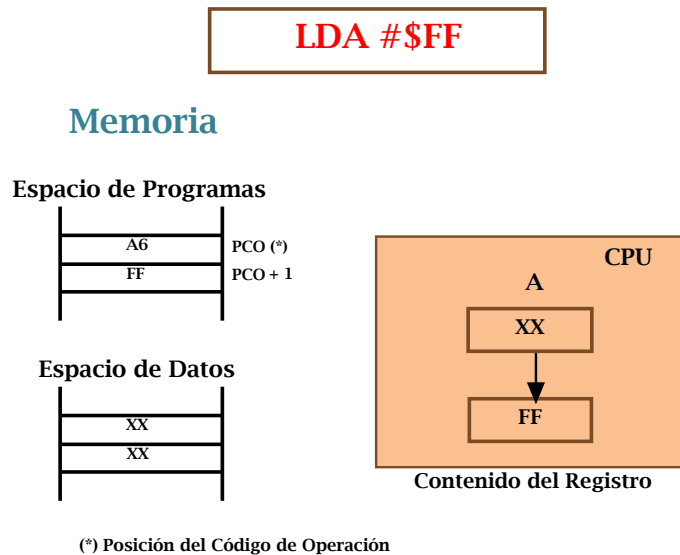


Figura 23. Ejemplo de direccionamiento inmediato.



### 3.2. Modo de Direccionamiento Inherente

En el modo de direccionamiento **inherente**, toda la información requerida para la operación ya es implícitamente conocida por la CPU y no es necesario recuperar un operando exterior desde la memoria. Los operandos (si los hay) son sólo los registros de la CPU o bien valores de datos almacenados en la pila. Esta es una instrucción de un solo byte (el código de operación).

**Ejemplo:**

**Código de operación:** 4F

**Formato de instrucción en Assembler:** CLRA; limpia el Acumulador

**Secuencia de Ejecución:**

**Dirección Contenido**

\$EE00      \$4F (1) y (2)

**Explicación:**

- (1) La CPU lee el código de operación \$4F – borra el acumulador.
- (2) La CPU almacena el valor 00 en el acumulador.

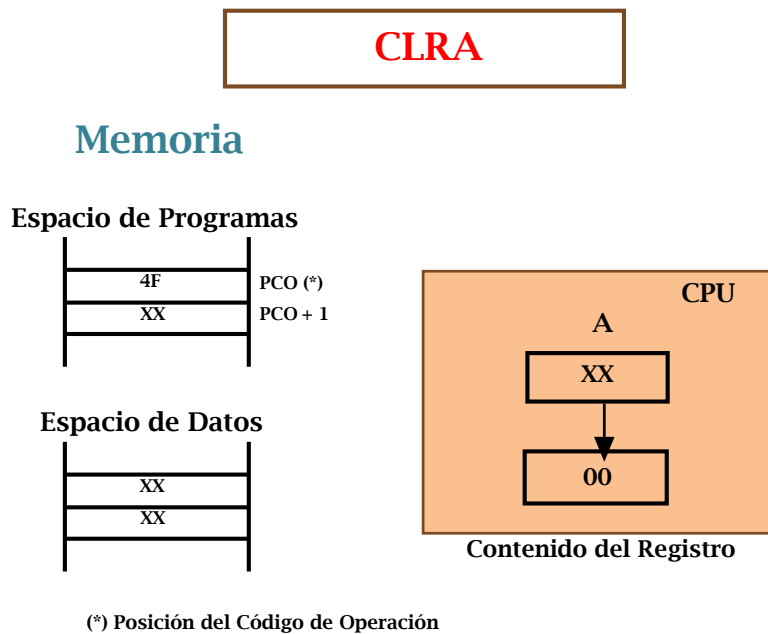


Figura 24. Ejemplo de direccionamiento inherente.

### 3.3. Modo de Direccionamiento Directo

En el modo de direccionamiento **directo**, la dirección del operando se encuentra a continuación del código de operación de la instrucción. Se llama direccionamiento directo porque la dirección donde se encuentra el operando está en la parte baja de la memoria entre \$00 y \$FF y por este motivo ocupa un solo byte en la instrucción. El motivo por el cual esta dirección ocupa un solo byte es porque en el direccionamiento directo, el procesador asume que el byte más significativo de la dirección es cero cuando no se indica explícitamente.

Este modo es usado para acceder a los primeros 256 bytes de memoria, cuya área se denomina **página directa** e incluye a los registros de RAM e I/O del interior del chip. Este modo es eficiente tanto en economía de espacio de memoria como en tiempo de ejecución.

Esta es una instrucción de dos bytes, uno para el código de operación y otro para el byte de menor peso de la dirección del operando.

#### Ejemplo:

**Código de operación:** B6

**Formato de instrucción en Assembler:** LDA \$50; carga el acumulador desde una dirección de página directa (la dirección \$50).

#### Secuencia de Ejecución:

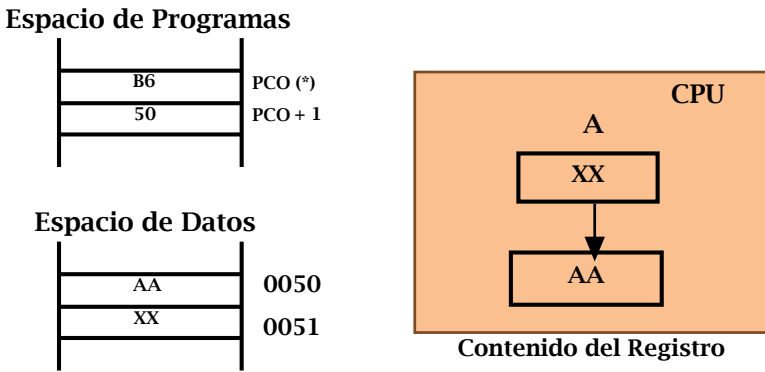
Dirección	Contenido
\$EE00	\$B6 (1)
\$EE01	\$50 (2) y (3)

#### Explicación:

- (1) la CPU lee el código de operación \$B6 - Cargar del acumulador usando el modo de direccionamiento directo.
- (2) La CPU lee la dirección \$50 contenida en la posición de memoria \$EE01. Este \$50 es interpretado como la dirección de página directa.
- (3) La CPU arma la dirección directa completa \$0050 asumiendo el valor del byte de mayor peso en \$00, con el valor previamente leído de la instrucción. Esta dirección es colocada en el bus de direcciones y la CPU lee el valor del dato contenido en la posición de memoria \$0050 y lo carga en el acumulador (ver figura 25).

LDA \$50

Memoria



(\*) Posición del Código de Operación

Figura 25. Ejemplo del modo de direccionamiento directo.

**3.4. Modo de Direccionamiento Extendido**

En el modo de **direccionamiento extendido**, la dirección del operando está contenida en los dos bytes siguientes al código de operación. Este modo es usado para acceder a cualquier posición de memoria mayor a \$00FF, incluyendo espacio de RAM mayor a \$00FF, ROM o FLASH. Esta es una instrucción de tres bytes, uno para el código de operación y otros dos para la dirección del operando.

**Ejemplo:**

**Código de operación:** C6

**Formato de instrucción en Assembler:** LDA \$0400; Carga el acumulador desde la dirección extendida \$0400.

**Secuencia de Ejecución:**

**Dirección Contenido**

\$EE00	\$C6 (1)
\$EE01	\$04 (2)
\$EE02	\$00 (3) y (4)

**Explicación:**

(1) La CPU lee el código de operación \$C6 - Carga del acumulador usando el modo de direccionamiento extendido.

- (2) La CPU lee \$04 de la posición de memoria \$EE01. Este \$04 es interpretado como la mitad de mayor peso de una dirección.
- (3) La CPU lee \$00 de la posición de memoria \$EE02. Este \$00 es interpretado como la mitad de menor peso de una dirección.
- (4) La CPU arma la dirección extendida completa \$0400 con los dos valores previamente leídos. Esta dirección es colocada en el bus de direcciones para que la CPU lea el valor del dato contenido en la posición de memoria \$0400 y lo cargue en el acumulador.

**LDA \$0400**

### Memoria

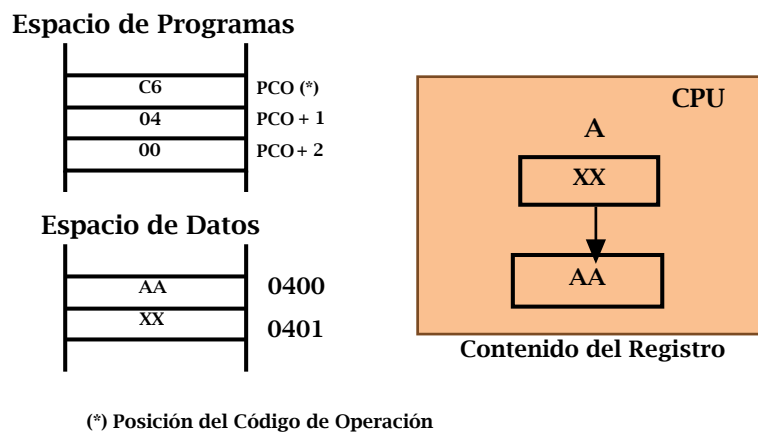


Figura 26. Ejemplo de direccionamiento extendido.

### 3.5. Modo de Direccionamiento Indexado

En el modo de direccionamiento **indexado**, la dirección efectiva del operando es variable y depende de dos factores:

- 1) el contenido actual del registro índice (X).
- 2) el desplazamiento contenido en el o los bytes siguientes al código de operación.

La CPU HC08 soporta tres tipos de direccionamientos indexados: sin desplazamiento, con desplazamiento de 8 bits y con desplazamiento de 16 bits. Un código ensamblador bien programado usará el modo de direccionamiento indexado que requiera el menor número de bytes para expresar el desplazamiento.

### 3.5.1. Direccionamiento indexado “sin desplazamiento (Offset)”

Se especifica el contenido del Registro índice H:X como dirección del operando. En el HC908 el registro índice es de 16 Bits, por lo que el compilador assembler interpretará “H:X” cuando solo vea “X” en la instrucción con direccionamiento indexado.

En el modo de direccionamiento indexado sin desplazamiento, la dirección efectiva del operando para la instrucción está contenida en los 16 bits del registro índice.

**Ejemplo:**

**Código de operación:** 7F

**Formato de instrucción en Assembler:** CLR ,X; borra el contenido de la dirección apuntada por X.

**Secuencia de Ejecución:**

Dirección	Contenido
\$EE00	\$7F (1), (2) y (3)

**Explicación:**

- (1) La CPU lee el código de operación \$7F – borrar el contenido de la dirección apuntada por el registro índice.
- (2) la CPU arma la dirección completa sumando \$0000 al contenido del registro índice de 16 bits (X:H).
- (3) La CPU borra el valor del dato contenido en esa posición de memoria, que para este ejemplo es la dirección \$0400.

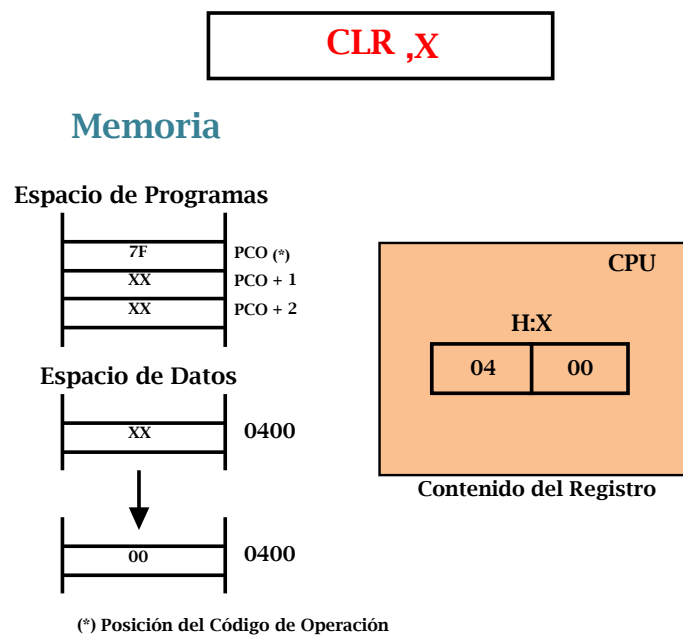


Figura 27. Ejemplo del modo de direccionamiento indexado sin desplazamiento.

### 3.5.2. Direccionamiento indexado con “desplazamiento (offset) de 8 bits”

En el modo de direccionamiento indexado con desplazamiento de 8 bits, la dirección efectiva es la suma del contenido del registro índice de 16 bits y el byte de desplazamiento siguiente al código de operación. El byte de desplazamiento suministrado en la instrucción es un número entero no signado de 8 bits.

Esta es una instrucción de dos bytes, uno para el código de operación y otro para el byte de desplazamiento. El contenido del registro índice no es alterado.

#### Ejemplo:

**Código de operación:** 6F

**Formato de instrucción en Assembler:** CLR 10,X ; Limpia la dirección indicada por el décimo primer ítem de la tabla apuntada por X.

#### Secuencia de Ejecución:

Dirección	Contenido
\$EE00	\$6F (1)
\$EE01	\$0A (2), (3) y (4)

#### Explicación:

(1) La CPU lee el código de operación \$6F – borrado del contenido de la dirección apuntada por el registro índice, usando el modo de direccionamiento indexado con desplazamiento de 8 bits.

(2) La CPU lee un \$0A de la posición de memoria \$EE01. Este \$0A es interpretado como un desplazamiento de 8 bits.

(3) La CPU arma la dirección completa sumando el valor antes leído (\$0A) al contenido del registro índice de 16 bits (X:H).

[4] la CPU borra el valor del dato contenido en la posición de memoria que se encuentra en la dirección obtenida como la suma del contenido del registro índice y el desplazamiento. En este ejemplo es \$0400 + \$0A= \$040A. Lo que significa que se borrará el contenido de la dirección de memoria \$040A

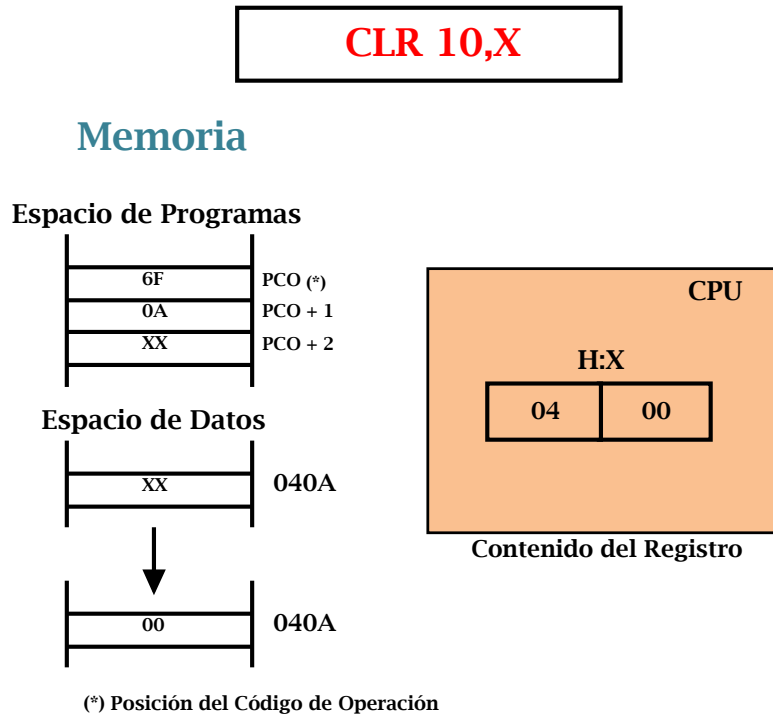


Figura 28. Ejemplo de direccionamiento indexado con 8 bits de offset.

### 3.5.3. Direccionamiento indexado con “desplazamiento (offset) de 16 bits”

En el modo de direccionamiento indexado con desplazamiento de 16 bits, la dirección efectiva es la suma del contenido del registro índice de 16 bits y los dos bytes de desplazamiento siguientes al código de operación. El byte de desplazamiento suministrado en la instrucción es un número entero no signado de 16 bits. Esta es una instrucción de tres bytes, uno para el código de operación y otros dos para los bytes de desplazamiento.

**Ejemplo:**

**Código de operación:** D7

**Formato de instrucción en Assembler:** `STA $0100,X` ; almacena en el contenido de X + \$0100 el valor del acumulador.

**Secuencia de Ejecución:**

Dirección	Contenido
\$EE00	\$D7 (1)
\$EE01	\$01 (2)
\$EE02	\$00 (3), (4) y (5)

**Explicación:**

- (1) La CPU lee el código de operación \$D7 – almacena en memoria el dato del acumulador usando el modo de direccionamiento indexado con desplazamiento de 16 bits.
- (2) La CPU lee \$03 de la posición de memoria \$EE01. Este \$01 es interpretado como la mitad de mayor peso de una dirección base.
- (3) La CPU lee \$00 de la posición de memoria \$EE02. Este \$00 es interpretado como la mitad de menor peso de una dirección base.
- (4) La CPU arma la dirección completa sumando la dirección base de 16 bits antes leída (\$0100) al contenido del registro índice de 16 bits (X:H).
- (5) Esta dirección es colocada en el bus de direcciones y la CPU almacena el valor del dato contenido en el acumulador en dicha posición de memoria. Para el ejemplo de la Figura 29 se almacena el número 55 en la Dirección \$0150

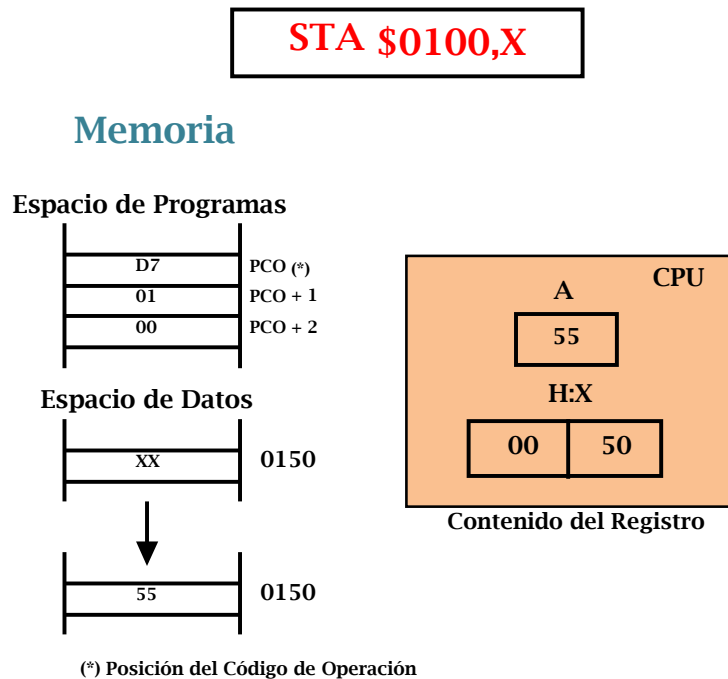


Figura 29. Ejemplo de direccionamiento indexado con 16 bits de offset.

**3.5.4. Direccionamiento Indexado “8 & 16 Bit de desplazamiento (Offset)”**

Comúnmente usado para acceder a elementos de estructura de datos. El Offset es la dirección base de la estructura. El registro índice contiene el desplazamiento del nuevo elemento.

En la figura 30 se esquematiza el funcionamiento de este direccionamiento. En el caso que la tabla se encuentre en los primeros 256 bytes de memoria, la mayoría de los ensambladores usarán instrucciones de 8 bit de offset.



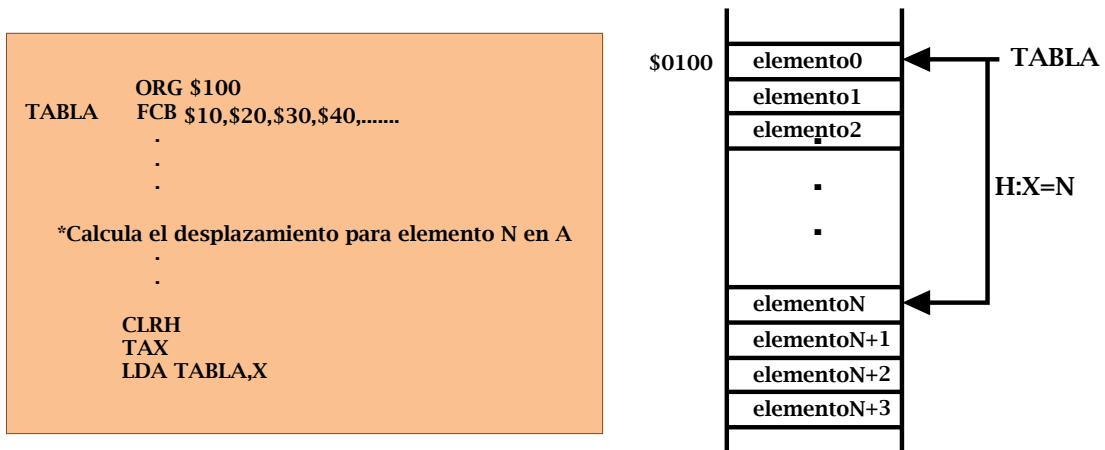


Figura 30. Direccionamiento indexado de 8 y 16 bits de desplazamiento.

### 3.5.5. Direccionamiento Indexado Usando el “Stack pointer” y “8-Bit de Offset”

La localización de memoria se obtiene de hacer: 8 bit offset no signado + Registro SP no signado. El byte inmediatamente seguido al byte del código de operación es el correspondiente a los 8 bit de offset. En el ejemplo de la Figura 31 se almacena el contenido del acumulador en la dirección \$D5 que se obtiene como suma del contenido del stack y el desplazamiento dado por el 5.

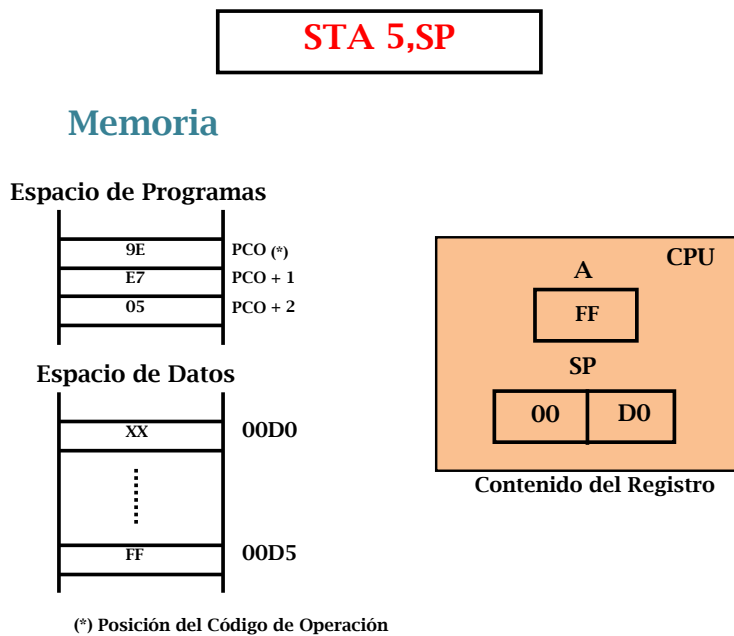


Figura 31. Ejemplo de direccionamiento indexado usando SP y 8 bits de offset.

### 3.5.6. Direccinamiento Indexado usando “Stack pointer” y “16-Bit de Offset”

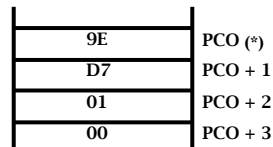
La localización de memoria se obtiene de hacer: 16 bit offset no signado + registro SP no signado. El Registro SP no es afectado. Los bytes inmediatamente seguidos al byte del código de operación, son los correspondientes a los 16 bit de offset.

Se debe notar que si las interrupciones están deshabilitadas, el SP puede ser usado como un registro índice adicional (menos eficiente por el pre-byte).

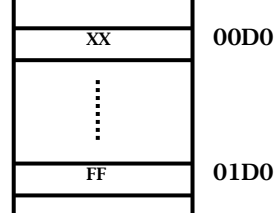
STA \$100,SP

#### Memoria

##### Espacio de Programas



##### Espacio de Datos



(\*) Posición del Código de Operación

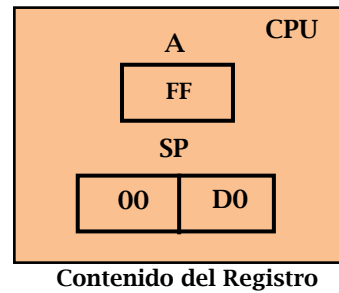


Figura 32. Ejemplo de direccionamiento indexado usando SP y 16 bits de offset.

### 3.5.7. Stack Pointer “8 bit de offset”

Soporta Lenguajes de Alto Nivel. Los Compiladores a menudo colocan parámetros para procedimientos y almacenamiento temporal en el stack, una manera eficiente de acceder a esa información es direccionando el stack pointer.

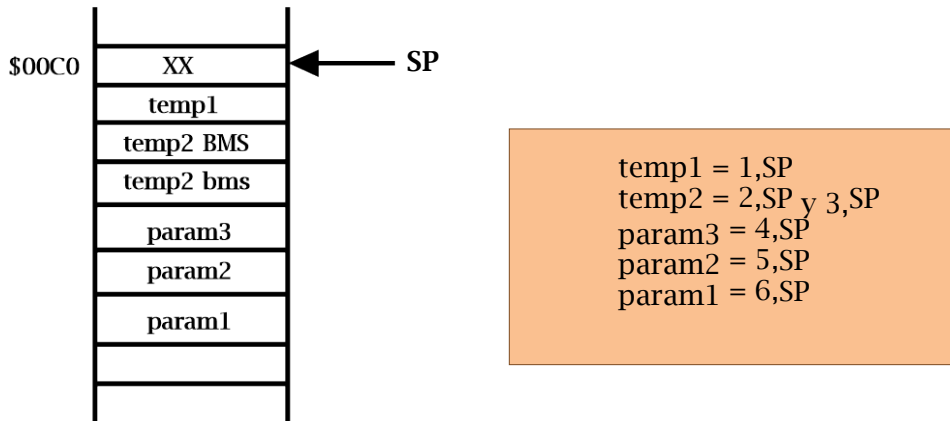


Figura 33. Stack Pointer con 8 bit de offset.

### 3.5.8. Direccionamiento Indexado “sin offset con Post Incremento”

El Registro índice H:X contiene la dirección del operando. Después que se calcula la dirección del operando, el contenido del registro H:X se incrementa en 1.

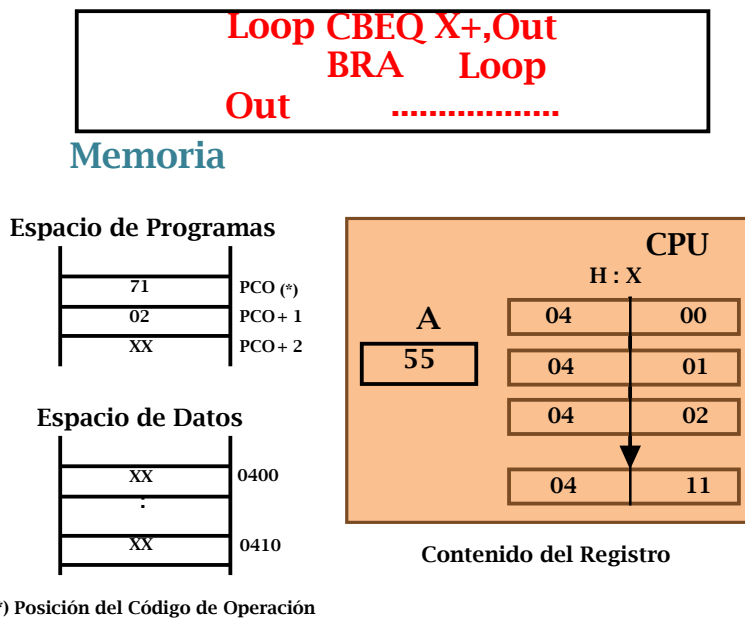


Figura 34. Ejemplo de direccionamiento indexado sin offset con post incremento.

### 3.5.9. Direccionamiento Indexado “con 8 bit de offset y Post Incremento”

Igual al direccionamiento indexado con 8 bit offset, más post incremento. Después que se calcula la dirección del operando, el contenido del registro H:X se incrementa en 1.

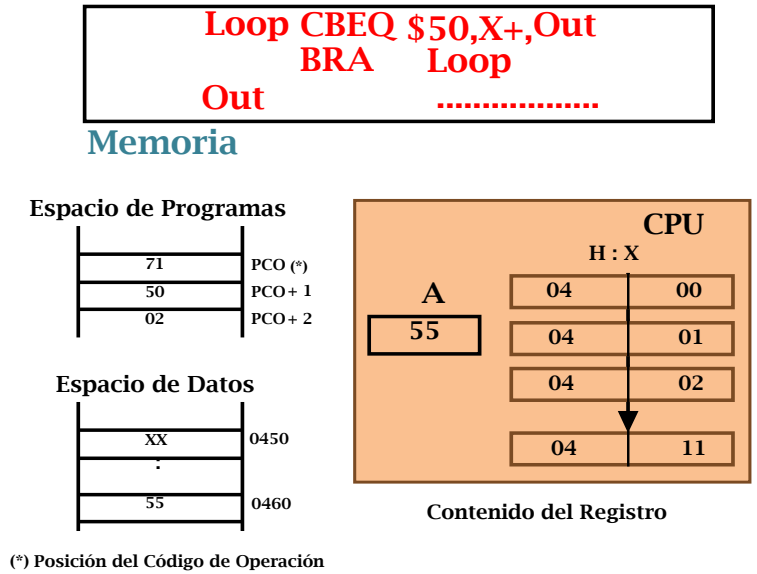


Figura 35. Direccionamiento indexado con 8 bit de desplazamiento y post incremento.

### 3.6. Modo de Direccionamiento Relativo

El modo de **direccionamiento relativo** es usado solamente por las instrucciones de bifurcación (saltos condicionales). Las instrucciones de bifurcación, salvo las bifurcaciones en su versión de manipulación de bits generan dos bytes de código de máquina: uno para el código de operación y otro para el desplazamiento relativo. Ya que es deseable bifurcar en cualquier sentido, el byte de desplazamiento es un número signado según el método de representación en “complemento a dos” con un rango que va desde -128 hasta +127 bytes (el salto se realiza respecto a la dirección de la instrucción inmediata posterior a la instrucción de bifurcación). Si la condición de salto es verdadera, el contenido de los 8 bits, del byte con signo, siguiente al código de operación es sumado al contenido del contador de programa para formar la dirección de salto efectiva, es decir, Contador de Programa = Contador de Programa + 8 bit offset signado; de otro modo, la ejecución continúa bajo la instrucción inmediata posterior a la instrucción de salto condicional (en este caso el Contador de Programa no es afectado). Un programador especifica el destino de una bifurcación como una dirección absoluta (o etiqueta que hace referencia a una dirección absoluta). **El compilador calcula el desplazamiento relativo de 8 bits con signo**, que es colocado en memoria luego del código de operación de la bifurcación.

En la figura 36 se puede observar el esquema de este tipo de direccionamiento, el cual tiene básicamente las siguientes características:

- Solo para instrucciones de salto (branch)
- El PC es incrementado en 2 desde OCL (debido a la precarga)
- 8 bit de offset. El rango es de -128 a + 127 desde el registro PC

- Dirección efectiva (EA) = PC + desplazamiento (8 bits de offset)
- El desplazamiento calculado por el compilador de Assembler es:  $desp. = EA - PC$

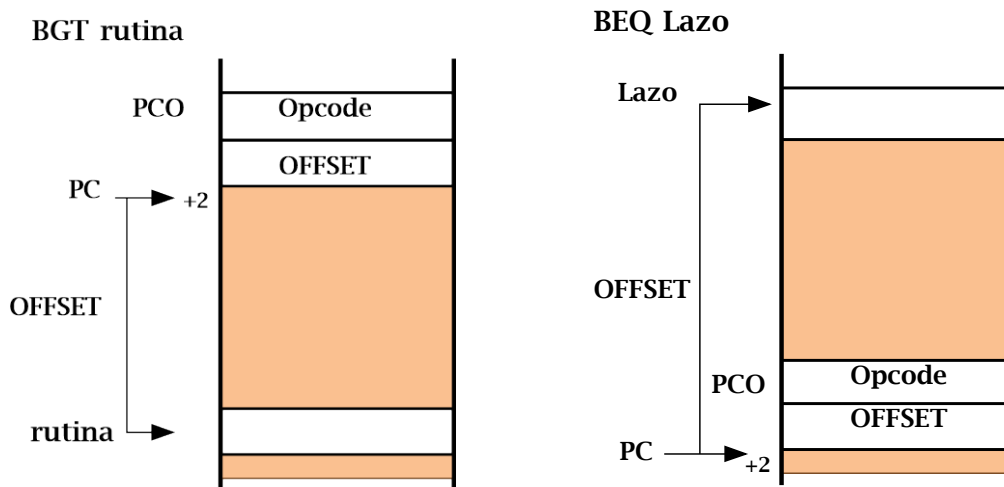


Figura 36. Modo de direccionamiento relativo.

**Ejemplo:**

**Código de operación:** 27

**Formato de instrucción en Assembler:** **BEQ \$F100**; salta a \$F100 si Z = 1 (si el resultado de la operación anterior es igual a cero).

**Secuencia de Ejecución:**

**Dirección Contenido**

- \$F150 \$27 (1)
- \$F151 \$AE (2) y (3)

**Explicación:**

- (1) La CPU lee el código de operación \$27 (saltar si Z = 1). El bit Z del registro de código de condición será uno si el resultado de la operación aritmética o lógica previa fue cero.
- (2) La CPU lee \$AE de la posición de memoria \$E151. Este \$AE es interpretado como el valor de desplazamiento relativo. Después de este ciclo el contador de programa apunta al primer byte de la próxima instrucción (\$F152).
- (3) Si el bit Z está en cero, no sucede nada en este ciclo y el programa debe continuar con la próxima instrucción. Si el bit Z está en uno, la CPU armará la dirección completa sumando el desplazamiento signado leído previamente (\$AE) al contenido del registro contador de programa para obtener la dirección destino de la bifurcación. Esto provoca que la ejecución del programa continúe desde una nueva dirección (\$F100).

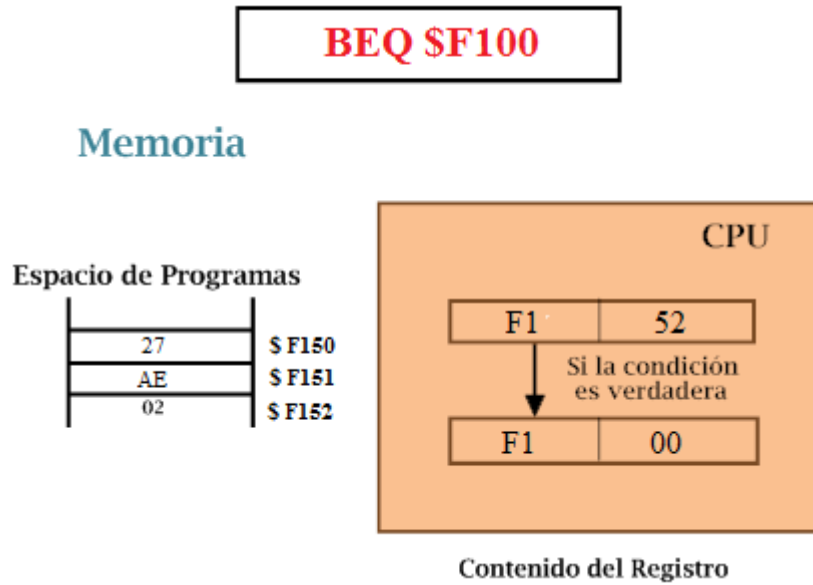


Figura 37. Ejemplo de direccionamiento relativo.

### 3.7. Direccionamiento Memoria a Memoria

Usado para mover información desde una locación de memoria a otra. No usa ni afecta registros del CPU, excepto cuando se usa direccionamiento indexado con post incremento.

Es más eficiente que la combinación Load/Store, pero puede utilizarse con instrucciones MOV solamente. La expresión debe comprender: MOV Dirección Fuente, Dirección Destino.

Hay cuatro variantes de direccionamiento memoria a memoria: Inmediato a Directo, Directo a Directo, Indexado a Directo con Post Incremento, Directo a Indexado con Post Incremento.

#### 3.7.1. Direccionamiento Memoria a Memoria “Inmediato a Directo “

La Fuente es un byte “valor inmediato” y el Destino debe estar en los primeros 256 bytes de memoria. Se usa comúnmente en inicialización de variables o registros en RAM.

MOV # $\$AA$ , $\$F0$

## Memoria

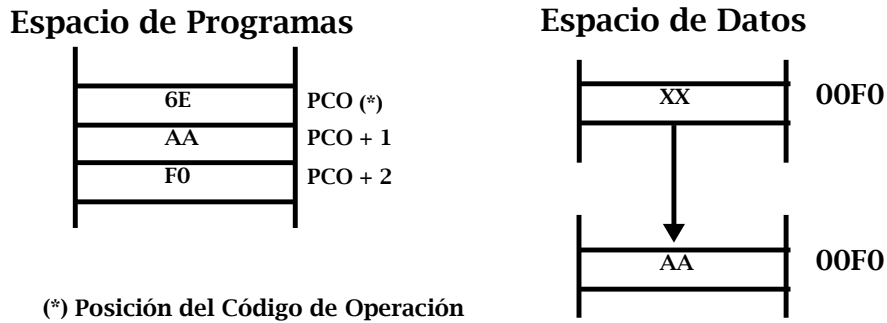


Figura 38. Direccionamiento memoria a memoria, "Inmediato a Directo".

### 3.7.2. Direccionamiento Memoria a Memoria "Directo a Directo"

La Fuente y el destino deben estar en los primeros 256 bytes de memoria. Comúnmente se usa para movimiento de datos desde una página cero a otro lugar dentro de la misma página (mover datos dentro de la misma RAM).

MOV  $\$00$ , $\$F0$

## Memoria

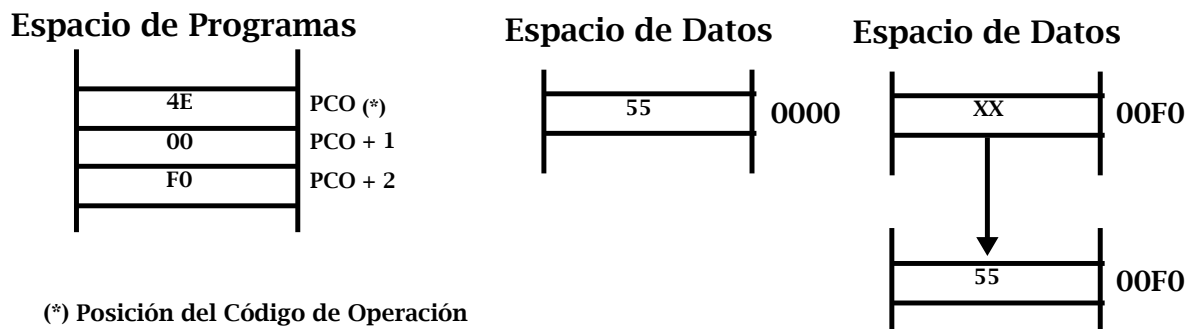


Figura 39. Direccionamiento memoria a memoria, "directo a directo".

### 3.7.3. Direccionamiento Memoria a Memoria “Indexado con Post Incremento a Directo”

La Fuente puede ser cualquier lugar en el mapa de memoria y el Destino debe estar en los primeros 256 bytes de memoria. Comúnmente se usa para escribir datos a un dispositivo de comunicación desde un buffer en RAM o Flash.

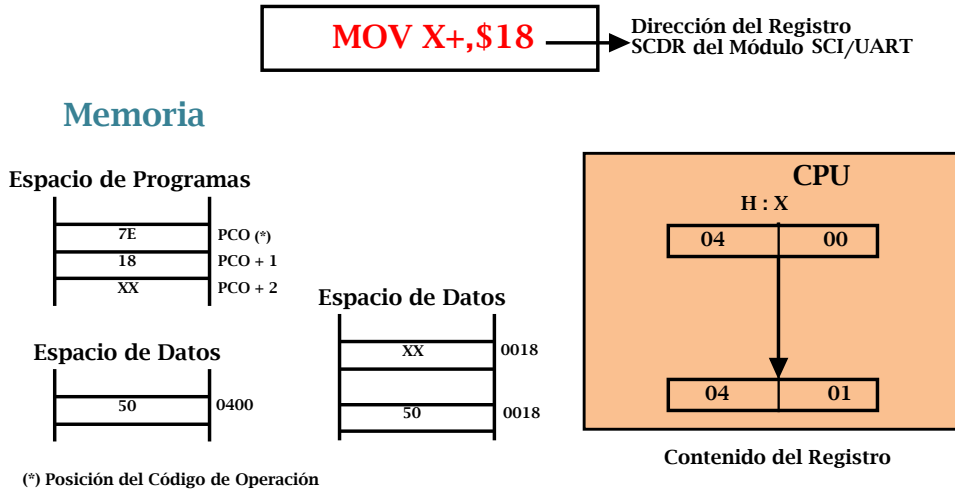


Figura 40. Direccionamiento memoria a memoria “indexado con post incremento a Directo”

### 3.7.4. Direccionamiento Memoria a Memoria “Directo a Indexado con Post Incremento”

La Fuente debe estar en los primeros 256 bytes de memoria y el Destino puede ser cualquier lugar en el mapa de memoria. Comúnmente se usa para escribir datos desde un dispositivo de comunicación a un buffer en RAM o Flash.

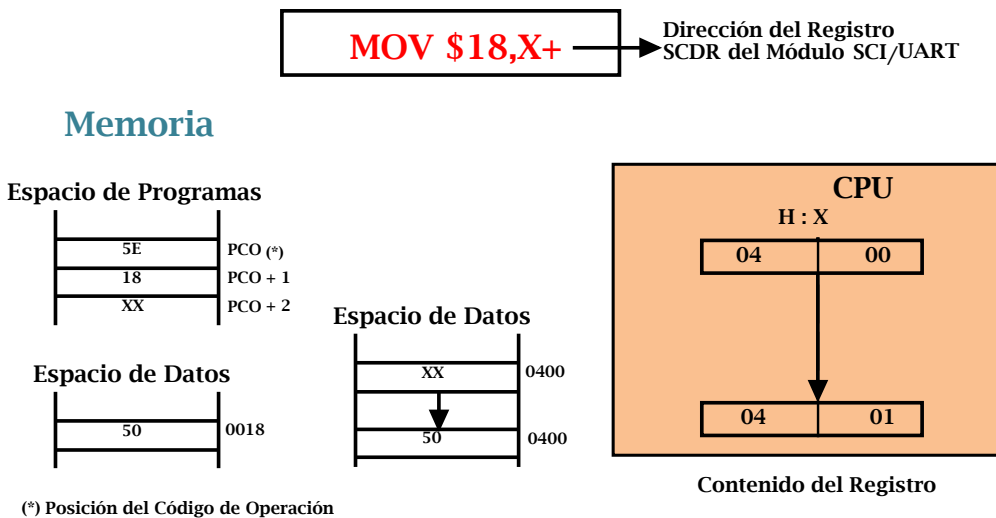


Figura 41. Direccionamiento memoria a memoria “Directo a Indexado con post -Incremento”.



# CAPÍTULO 4

## Memoria

*Jorge R. Osio*

### 4.1. Introducción

La CPU08 puede direccionar 64 Kbytes de espacio de memoria [3]. El mapa de memoria que se encuentra en la figura 42 muestra:

- 4096 bytes de FLASH de usuario para los MCU MC68HC908QT4 y MC68HC908QY4
- 1536 bytes de FLASH de usuario para los MCU MC68HC908QT2, MC68HC908QT1, MC68HC908QY2, y MC68HC908QY1
- 128 bytes de RAM
- 48 bytes de vectores definidos por el usuario, ubicados en FLASH
- 416 bytes de memoria de solo lectura de monitor (ROM)
- 1536 bytes de FLASH de programa y de rutinas de borrado, ubicadas en ROM

### 4.2. Ubicaciones de memoria no implementada

Acceder a ubicaciones no implementadas puede tener respuestas impredecibles en la operación del MCU.

### 4.3. Ubicaciones de memoria reservada

Acceder a ubicaciones de memoria reservadas puede tener efectos impredecibles sobre la operación del MCU. En la Figura 42 la memoria reservada está indicada con la palabra reservada o con la letra R.

### 4.4. Sección entrada/salida (I/O)

Las direcciones \$0000–\$003F, contiene registros de control, estado y datos. Los demás registros de entrada/salida tienen las siguientes direcciones:

- \$FE00 — Registro de estado Break, BSR
- \$FE01 — Registro de estado Reset, SRSR
- \$FE02 — Registro auxiliar Break, BRKAR
- \$FE03 — Registro de control Break flag, BFCR
- \$FE04 — Registro de estado Interrupt 1, INT1
- \$FE05 — Registro de estado Interrupt 2, INT2
- \$FE06 — Registro de estado Interrupt 3, INT3
- \$FE07 — Reserved
- \$FE08 — FLASH control register, FLCR
- \$FE09 — Registro de dirección Break parte alta, BRKH
- \$FE0A — Registro de dirección Break parte baja, BRKL
- \$FE0B — Registro de estado y control Break, BRKSCR
- \$FE0C — Registro de estado LVI, LVISR
- \$FE0D — Reservado
- \$FFBE — Registro de protección de bloque FLASH, FLBPR
- \$FFC0 — Valor de trim del OSC interno — Opcional
- \$FFFF — Registro de control COP, COPCTL



Figura 42. Mapa de memoria de la línea HC908QT/QY4

## 4.5. Memoria de Acceso Aleatorio (RAM)

Las direcciones \$0080–\$00FF corresponden a la ubicación de la memoria RAM [3], [11] y [12]. La ubicación de la Pila de RAM es programable. El puntero de pila de 16-bit permite a la pila estar en cualquier espacio de memoria de los 64-Kbyte. Es importante comprender que para una operación correcta el puntero de pila debe apuntar solo a las direcciones de RAM. Antes de procesar una interrupción, la CPU usa 5 bytes de la pila para guardar el contenido de los registros de la CPU.

Durante una llamada a subrutina, la CPU usa 2 bytes de la pila para almacenar la dirección de retorno de la subrutina. El puntero de Pila se decrementa durante el ingreso de datos y se incrementa al extraer datos de la pila.

Se debe tener cuidado al usar subrutinas anidadas. La CPU puede sobre escribir los datos en la RAM durante una subrutina o durante la operación de apilado en una interrupción.

## 4.6. Memoria FLASH (FLASH)

A continuación se describe el funcionamiento de la memoria FLASH. La memoria FLASH puede ser leída, programada y borrada externamente. La operación de programado y borrado se habilita a través del uso de un pulso de carga.

La memoria flash consiste de un arreglo de 4096 o 1536 bytes con 48 bytes adicionales para vectores de usuario. El tamaño mínimo de memoria flash que se puede borrar es de 64 bytes; y el máximo tamaño de memoria FLASH que puede ser programado en un ciclo de programa es de 32 bytes (una columna). Las operaciones de programado y borrado se facilitan a través de bits de control en el registro de control de FLASH (FLCR).

Los rangos de direcciones para la memoria de usuario y vectores son:

- \$EE00 – \$FDFF; memoria de usuario, 4096 bytes: MC68HC908QY4 y MC68HC908QT4
- \$F800 – \$FDFF; memoria de usuario, 1536 bytes: MC68HC908QY2, MC68HC908QT2, MC68HC908QY1 y MC68HC908QT1
- \$FFD0 – \$FFFF; vectores de interrupción del usuario, 48 bytes.

# CAPÍTULO 5

## Interrupciones

*Jorge R. Osio y Walter J. Aróztegui*

Los resets e interrupciones son excepciones del CPU (al proceso normal), o sea “rompen” con la secuencialidad normal de la ejecución de un programa. Determinando que tipo de manejo se requiere, se llama al proceso de excepción, el cual contiene el código a ser ejecutado durante la interrupción.

### 5.1. Introducción

El Módulo de operación correcta del computador (COP) [3] contiene contadores ejecutándose libremente que generan un reset si se produce un desborde. El módulo COP ayuda a recuperar la línea de ejecución de código del software. Se puede prevenir un reset de COP borrando el contador periódicamente. El módulo COP se puede deshabilitar a través del bit COPD en el registro de configuración 1 (CONFIG1).

Las interrupciones se pueden clasificar en:

- Enmascarables
- No enmascarables
- De software

Las **interrupciones enmascarables**, se pueden enmascarar mediante el flag (indicador) I del CCR. Mientras el flag I se encuentre en alto (1 lógico), el procesador no atenderá las interrupciones enmascarables. Esto quiere decir que dichas interrupciones surtirán efecto cuando el flag esté en bajo (cero lógico). Esto último se consigue mediante las instrucción cli (clear I)

Las **no enmascarables** no se pueden inhibir y son las correspondientes a un fallo: una instrucción ilegal o un RESET.

Las **interrupciones por software** son producidas por el programador, esto se logra al agregar la instrucción SWI dentro del código de programa. Este tipo de interrupción es no enmascarable.

Adicionalmente, entre los tipos de interrupciones se pueden distinguir por un lado las **internas** del microcontrolador, que se utilizan para monitorear el buen funcionamiento del MCU e independizar el uso de los módulos. Por otro lado las **externas**, correspondientes a las producidas por circuitos externos al microcontrolador.

Interrupciones externas:

- IRQ (Línea de interrupción externa)
- KBI (interrupciones externas mediante puertos de entrada salida)

Interrupciones internas:

- SCI (interrupciones de comunicación serie asincrónica)
- SPI (interrupción de comunicación serie sincrónica)
- ADC (interrupción del conversor analógico digital)
- I2C (interrupción de comunicación serie sincrónica)
- USB (interrupción desde un host USB)
- TPM (interrupción por desborde del contador de tiempo e interrupción por un evento en el canal para el modo comparador de salida o captura de entrada)
- RTC (interrupción de contador de tiempo real)
- SWI (Interrupción por Software)
- MCG (Interrupción de PLL para generación del reloj del sistema)

## 5.2. Secuencia de Reset

El Reset puede ser usado por un evento “power on reset” (POR), en condiciones internas como el watchdog (reinicia el sistema cuando detecta inactividad) del COP (computer operating properly), o por un pin de reset externo activo en bajo. Cuando ocurre un reset, el CPU detiene inmediatamente la ejecución de la aplicación. Al finalizar el evento de reset, el CPU requiere 6-ciclos para buscar el contenido del vector de reset en las direcciones 0xFFFFE y 0xFFFF y cargar la primer instrucción que se encuentra en la dirección contenida en dichos vectores.

Los tipos de reset pueden ser:

- Power on
- COP
- Generado por el Pin de reset externamente
- LVI
- Dirección ilegal
- Código de operación ilegal

El contador del COP es un contador de 6 bits que se ejecuta libremente precedido por el del Módulo de Integración del sistema (SIM) de 12 bits (ver figura 43). Si no se borra por

software, el contador del COP desbordará y generará un reset asincrónico después de 262128 o 8176 ciclos de BUSCLKX4; dependiendo del estado del bit de selección de la tasa de ejecución del COP, COPRS, en el registro de configuración 1. Con una opción de desborde de 262128 ciclos de BUSCLKX4, el oscilador interno de 12.8-MHz da un periodo de time out de COP de 20,48 ms. Escribiendo cualquier valor en la dirección \$FFFF, antes de la ocurrencia de un desborde (overflow), se previene un reset por COP haciendo que se reinicie el contador del COP.

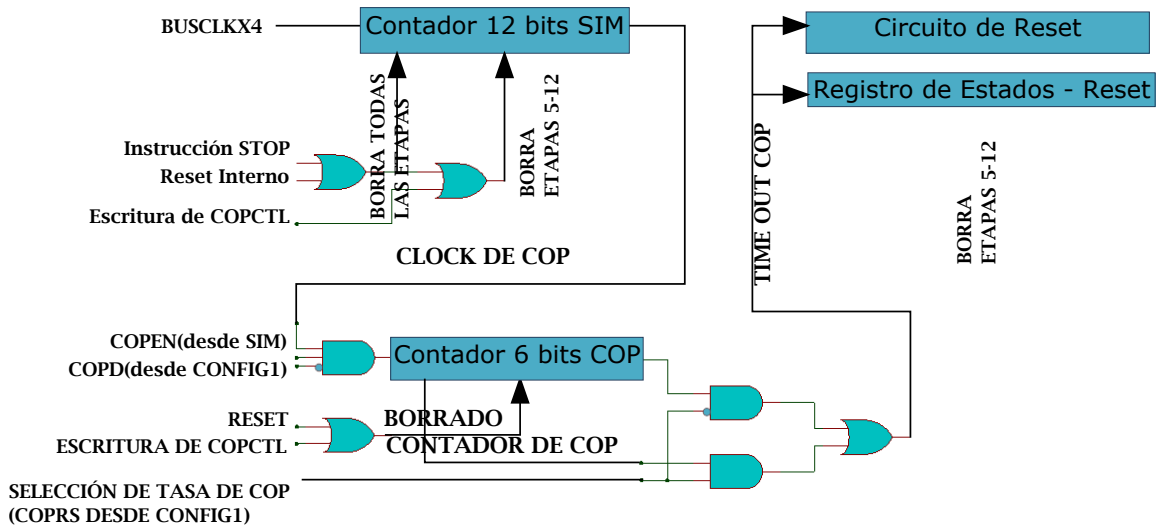


Figura 43. Diagrama en bloques del Módulo COP

Modificando el COP inmediatamente después de un reset y antes de entrar o después de excitar el modo stop, se garantiza el máximo tiempo antes del primer desborde en el contador del COP.

Un reset causado por el COP pone el pin RST en bajo (si el bit RSTEN es seteado en el registro CONFIG1) durante 32 x BUSCLKX4 ciclos y setea el bit de COP en el registro de estado del reset (RSR), esto último se ha detallado en la sección del Módulo SIM.

Se debe colocar la instrucción de borrado de COP en el programa principal y no en una subrutina de interrupción. Ya que la subrutina podría mantener el COP generando un reset, haciendo que el programa principal no funcione correctamente.

### 5.2.1. Registro de Control de COP

El registro de control del COP (COPCTL) [3] se encuentra ubicado en la dirección \$FFFF y se superpone con el vector de reset. Escribiendo algún valor en la dirección \$FFFF se borra el contador del COP e inicia un nuevo periodo de time out. Leyendo la dirección \$FFFF retorna el byte menos significativo del vector de reset.

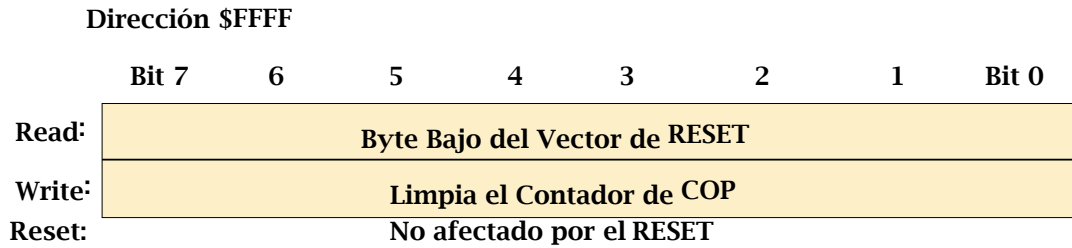


Figura 44. Registro de Control (COPCTL)

### 5.3. Secuencia de Interrupción

Cuando se genera un pedido de interrupción, el CPU finaliza la ejecución de la instrucción en curso antes de atender dicha interrupción. En este punto, el contador de programa queda apuntando a la dirección de la siguiente instrucción, que es a la instrucción que el procesador debe retornar luego del servicio de interrupción. El CPU ejecuta la interrupción de la misma manera que al ejecutar una interrupción por software (SWI).

La secuencia del CPU para atender una interrupción se muestra en la figura 45 y consta de los siguientes pasos:

1. Almacena el contenido de los registros PCL, PCH, X, A, y CCR en la pila, en ese orden.
2. Setea el bit I en el CCR.
3. Busca el contenido de la parte alta del vector de interrupciones.
4. Busca el contenido de la parte baja del vector de interrupciones.
5. Espera un ciclo de bus.
6. Busca tres bytes del programa de interrupción situado a partir de la dirección indicada en el vector de reset, para llenar la cola de ejecución de las instrucciones de la interrupción.

Luego el contenido del CCR se pone sobre la pila, el bit I en el CCR es seteado para prevenir otras interrupciones mientras se ejecuta la rutina del servicio de interrupciones. Al finalizar la ejecución de la rutina de interrupción se deberá borrar el flag I para poder atender otras interrupciones.

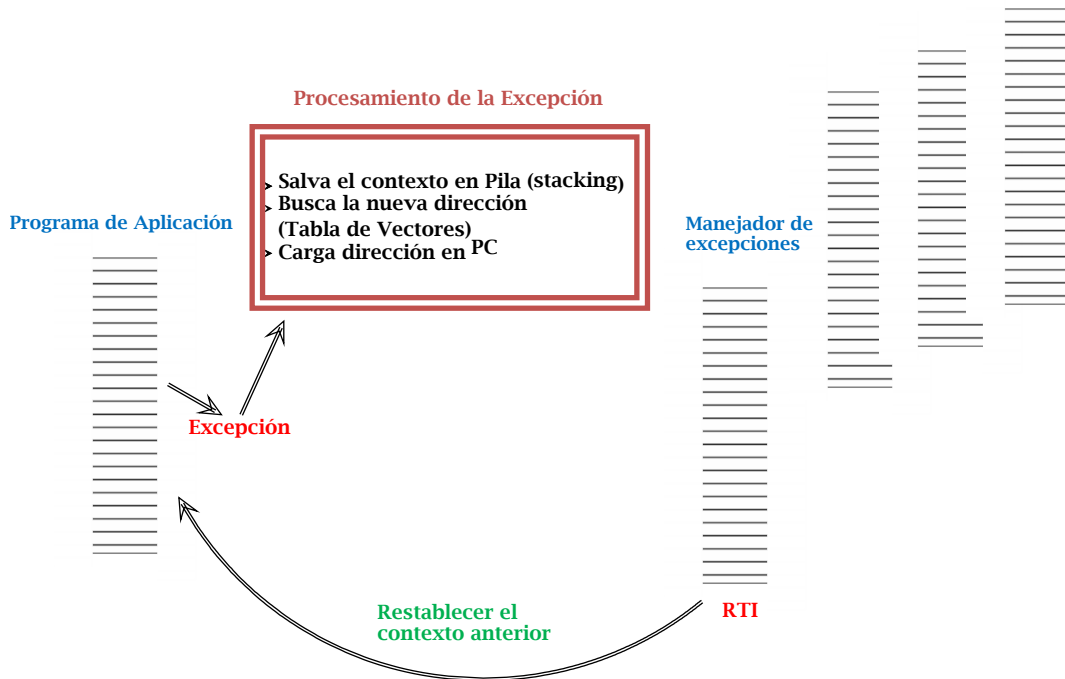


Figura 45. Ejecución de interrupciones

## 5.4. Interrupciones externas al procesador

La siguiente tabla describe las señales de la figura anterior:

**Tabla 3. Señales de Entrada al módulo COP**

Señales	Descripción
BUSCLKX4	BUSCLKX4 es la señal de salida del oscilador. La frecuencia de BUSCLKX4 es igual a la del oscilador interno, la del cristal, o la del oscilador-RC.
Instrucción STOP	La instrucción STOP borra el contador del SIM.
Escritura de COPCTL	Escribiendo un valor al registro de control del COP (COPCTL), Se borra el contador del COP. Leyendo el registro de control de COP se obtiene el byte menos significativo del vector de reset.
Power-On Reset	El power on reset (POR) borra el contador del SIM $4096 \times$ BUSCLKX4 ciclos después del power up.
Reset Interno	Un reset interno borra el contador del SIM y el del COP.
COPD (Deshabilita COP)	La señal COPD refleja el estado del bit que deshabilita el COP en el registro CONFIG1.
COPRS (selección de tasa de COP)	La señal COPRS muestra el estado de la tasa COP seleccionada en el bit (COPRS) en el Registro de configuración 1 (CONFIG1).



# CAPÍTULO 6

## Módulo Oscilador (OSC)

*Jorge R. Osio y Walter J. Aróztegui*

### 6.1. Introducción

El Clock del MCU se puede generar de varias maneras [3]. Por defecto el Microcontrolador utiliza un oscilador interno para generar el reloj de la CPU. Este oscilador tiene una frecuencia de bus de 3,2 MHz para el procesador HC08 (que puede variar  $\pm 25$ ). Para ajustar este valor a un valor de frecuencia conocido, se debe usar una frecuencia de referencia que permita encontrar el valor de ajuste (Trimmer).

Si se desea mayor frecuencia de bus se debe usar un **oscilador externo a cristal [6]**, cuya frecuencia máxima de bus puede llegar a los 8 MHz para un cristal de 32 MHz (se debe aclarar que estos valores dependen de la familia de procesadores, en este caso los valores corresponden al HC08). Este tipo de generación de clock utiliza una disposición como la que se muestra en la figura 46 utilizando un cristal, 2 capacitores y una resistencia externa. Otro tipo de **oscilador externo** es el **RC**, que utiliza solo dos componentes, una resistencia y un capacitor.

Por último, se puede utilizar un **oscilador externo**, de acuerdo a la configuración del registro Config. 2 que se muestra en la tabla 4.

En el caso del procesador HCS08, se dispone de un módulo generador de clock multipropósito (MCG) que provee varias fuentes de reloj seleccionables por el MCU. Este módulo posee un frequency-locked loop (FLL) y un phase-locked loop (PLL). El módulo MCG permite seleccionar clocks FLL o PLL, o ya sea un clock de referencia interno o externo como fuente de reloj del sistema Microcontrolador. Cualquier fuente de reloj seleccionada, pasará por un divisor de reloj que permite una menor frecuencia de señal a ser utilizada en otras partes del sistema. El MCG también controla un oscilador externo (XOSC) para el uso de un cristal como oscilador de referencia.

## 6.2. Inicialización del Oscilador externo en el HC08

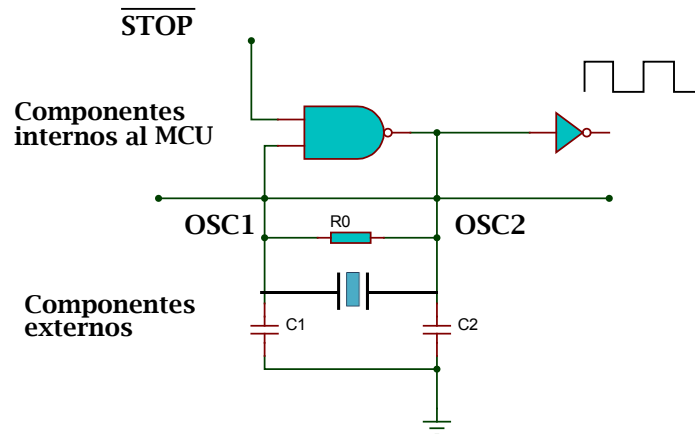


Figura 46. Oscilador externo.

La figura 46 muestra la conexión del oscilador estándar usado en MCUs para frecuencias en el rango de 1MHz a 32 MHz. Los pines del oscilador son llamados OSC1 y OSC2.

### 6.2.1. Pasos a seguir para cambiar del Clock interno al externo

Cuando se desea usar la fuente de reloj externa [7] (OSC externo, RC, o XTAL), el usuario debe seguir los siguientes pasos:

1. Para uso de cristal externo se debe setear, OSCOPT [1:0] = 1:1 (Ver Figura 47). Esto ayuda a arrancar un oscilador de cristal externo. Se debe setear PTA4 (OSC2) como salida y ponerlo en alto durante varios ciclos. Esto ayuda a iniciar el circuito del cristal más robustamente.

Tabla 4. Configuración de los bits OSCOPT

OSCOPT <sub>1</sub>	OSCOPT <sub>2</sub>	Modos del Oscilador
0	0	Oscilador Interno
0	1	Oscilador Externo
1	0	RC externo
1	1	Cristal Externo

**Dirección \$001E Registro de Configuración 2**

	Bit 7	6	5	4	3	2	1	Bit 0
Read:	IRQPUD	IRQEN	R	OSCOPT1	OSCOPT0	R	R	RSTEN
Write:								
Reset:	0	0	0	0	0	0	0	0

Figura 47. Registro config 2.

**Nota:** El registro Config 2 se puede escribir solo una vez después de cada reset. Por lo tanto, se deben setear todos los bits a la vez. El bit RSTEN se resetea a cero solo mediante un power on reset (POR)

- Se deben setear los bits [OSCOPT 1:0] de CONFIG2 de acuerdo a la lógica de control del módulo oscilador, entonces se colocará OSC1 como una entrada externa de reloj y si la opción externa del cristal se selecciona, OSC2 estará seteado como la salida del reloj.
- Se debe crear un retardo de software para esperar la estabilización del oscilador interno (clock por defecto) necesaria para la selección de la fuente de reloj externo (el cristal, el resonador, RC), esto está recomendado por el fabricante del componente. Una buena regla general para osciladores de cristal es esperar 4096 ciclos de la frecuencia de cristal, por ejemplo, Para un cristal 4-MHz, se debe esperar aproximadamente un 1 mseg. Las líneas de código siguientes permiten crear retardos donde la cantidad de milisegundos está dada por el valor guardado en el acumulador.

```

*****
* retardo por X mS
*
* El bucle interior atrasa 1mS. El lazo exterior cuenta un número de ms
* pasado por el acumulador.
*****
DelaymS2_Body:      ; JSR EXT trae el valor      6
DelaymS2010  ldx  #$F8      ; carga el retardo en X      2-\
DelaymS2020  decx      ; decrementa el retardo      3-\|
              nop          ; espera 2 ciclos de bus      2 ||
              bne  DelaymS2020  ; salta si no terminó      3-|/
              stx  COPR      ; atiende el WDOG          5 |
              ; Note que X será
              ; siempre cero aquí
              brn  *          ; espera 3 ciclos de bus      3 |
              deca         ; decrementa el # de mS        3 |
              bne  DelaymS2010  ; salta si no terminó      3-|/
DelaymS2030  rts          ; retorno de subrutina          6
    
```

- Luego que el retardo recomendado haya transcurrido, el bit ECGON en el registro de estado de OSC (OSCSTAT) debe ser seteado por el software del usuario.

**Dirección \$0036**

	Bit 7	6	5	4	3	2	1	Bit 0
Read:								ECGST
Write:	R	R	R	R	R	R	ECGON	
Reset:	0	0	0	0	0	0	0	0

= No Implementado     
 R = Reservado

Figura 48. Registro de estado del oscilador.

El registro de estado del oscilador (OSCSTAT) contiene el bit que hace el cambio desde el reloj interno al reloj externo.

**ECGON — Bit Generador de reloj externo:** Este bit de lectura/escritura habilita al generador externo de reloj, a fin de que se inicie el proceso de cambio. Este bit es puesto a cero forzado, durante el reset. Este bit se ignora en el modo monitor, con el oscilador interno, modo PTM o CTM.

- 1 = generador de reloj externo habilitado
- 0 = reloj externo deshabilitado

**ECGST — bit de estado del reloj externo:** Este bit, de solo lectura, sólo indica si hay una fuente de reloj externa manejando el reloj del sistema o no.

- 1 = Una fuente de reloj externo se acopló
- 0 = Una fuente de reloj externo se desacopló

5. Después que el valor de ECGON es detectado, el módulo OSC chequea la actividad del oscilador, esperando dos flancos ascendentes de reloj externo.
6. El módulo OSC entonces cambia al reloj externo. La lógica provee un glitch de transición.
7. El módulo OSC primero setea el bit ECGST en el registro OSCSTAT y entonces detiene el oscilador interno.

**NOTA:** Una vez que la transición al reloj externo termina, el oscilador interno sólo será reactivado con reset. Ningún cambio posterior del reloj monitor es implementado (el reloj no vuelve a cambiar a interno si el reloj externo muere).

**6.3. Inicialización del oscilador externo en el HCS08**

Típicamente se utiliza un cristal externo de 12Mhz y capacitores cerámicos de 20 pF. Para la configuración del clock del sistema se puede utilizar la función `init_clock` escrita en lenguaje C, la cual permitirá generar un reloj del sistema de 48Mhz mediante el PLL del sistema. De otra forma, al usar el *device initialization*, el clock del bus se configura a 8Mhz.

```

static void init_clock(void)
{
    /* se asume un cristal externo de 12MHz */

    /* Para usar el módulo USB se debe entrar en modo PEE y setear el MCGOUT a 48 MHz. luego del reset el MCG
    estará en modo FEI mode. */

    /**** cambiando desde el modo FEI (FLL engaged internal) a PEE (PLL engaged external). */

    /* Pasaje de FEI a FBE (FLL bypassed external) */

    /* Habilita fuente de reloj externa */

    MCGC2 = MCGC2_HGO_MASK    /* oscilador en modo de alta ganancia */
    | MCGC2_EREFS_MASK /* porque se está inicializando el uso del cristal */
    | MCGC2_RANGE_MASK /* 12 MHz está en rango de alta frecuencia */
    | MCGC2_ERCLKEN_MASK; /* active el reloj de referencia externo */

    while (MCGSC_OSCINIT == 0) ;

    /* Selecciona modo reloj */

    MCGC1 = (2<<6)    /* CLKS = 10 -> reloj de referencia externo. */
    | (3<<3); /* RDIV = 3 -> 12MHz/8=1.5 MHz */

    /* Espera a que termine el cambio de modo */

    while (MCGSC_IREFST != 0) ;

    while (MCGSC_CLKST != 2) ;

    /* cambia de modo FBE a PBE (PLL bypassed internal) */

    MCGC3=MCGC3_PLLS_MASK | (8<<0);

    /* VDIV=6 -> multiplica por 32 -> 1.5MHz * 32 = 48MHz */

    while(MCGSC_PLLST != 1) ;

    while(MCGSC_LOCK != 1) ;

    /* finalice el cambio de PBE a PEE (PLL enabled external mode) */

    MCGC1 = (0<<6) | (3<<3); /* CLKS = 0 -> PLL or FLL output clock. */

    /* RDIV = 3 -> 12MHz/8=1.5 MHz */

    while(MCGSC_CLKST!=3) ;

    /* Ahora el clock del microcontrolador es MCGOUT=48MHz, y el clock del bus de BUS_CLOCK=24MHz */
}

```

# CAPÍTULO 7

## Programación a bajo nivel (Assembler)

*Jorge R. Osio*

### 7.1. Introducción

El lenguaje de bajo nivel (assembler o ensamblador) se considera extremadamente importante dentro de la enseñanza de los sistemas de procesamiento, principalmente para la comprensión del funcionamiento interno del procesador en cuanto al movimiento de datos, ejecución de instrucciones, estructura de registros, etc.

Si bien actualmente es mucho más eficiente programar en Lenguajes de alto nivel, la programación en assembler es necesaria cuando se requiere precisión en la temporización de determinada rutina. También se utiliza en los sistemas operativos, por la necesidad de interacción entre el HW y el SO. Esto significa que es necesario comprender la programación y la ejecución de instrucciones para comprender el funcionamiento de los sistemas operativos.

El objetivo no es formar expertos programadores en assembler, pero es necesario comprender el funcionamiento de un programa y aprender a programar operaciones básicas como movimiento de datos, comparación entre datos y operaciones aritmético-lógicas.

En las siguientes subsecciones se describirá lo más importante de la programación en assembler, tal como la descripción de instrucciones, el simulador y la programación paso a paso de una aplicación en lenguajes de bajo nivel.

### 7.2. El lenguaje de bajo nivel

El lenguaje assembler es propio de cada procesador, esto significa que cada procesador tendrá su conjunto de instrucciones y la descripción de cómo utilizarlas. Para entender las características de la programación a bajo nivel es necesario conocer los registros que posee el procesador a utilizar y su función. A diferencia de los lenguajes de alto nivel, con estos lenguajes se pueden almacenar operandos y direcciones en los registros accesibles por el programador. En el caso del HC08, se dispone del Acumulador que se utiliza para guardar operandos y para guardar el resultado de la mayoría de las instrucciones que así lo requieren. Por ejemplo, para realizar una suma se requiere cargar uno de los operandos en el acumulador y luego mediante la instrucción de suma indicar la ubicación del segundo operando.

LDA #5A ; Carga el número 5A en el acumulador

ADD #5 ; suma el número 5 con el contenido del acumulador, el resultado se guarda en el

; acumulador

Otro registro utilizado para direcciones y datos es el registro índice, este registro es de 16 bits y tiene parte alta H y parte baja X. Cuando se almacena una dirección se puede utilizar por ejemplo para recorrer el contenido de una tabla de datos.

LDHX #80 ; esta instrucción almacena el valor 80 que es interpretado como una dirección

LDX \$95 ; esta instrucción almacena el contenido de la dirección 95 en el registro índice

Por último se dispone del Registro puntero de pila SP, que permite almacenar datos temporalmente en la pila. Se debe aclarar que la pila es utilizada por el procesador para guardar el contexto, (dirección de retorno al programa principal y contenido de los registros para el caso de la interrupción), cuando se produce una interrupción o se produce un salto a subrutina. Es por eso que se recomienda tener especial cuidado con el uso de la pila, al entrar a una subrutina. Para guardar un dato en la pila se dispone de la instrucción psh (guarda un dato en la pila) y para extraer un datos se utiliza pul.

PSHA ; guarda el contenido del acumulador en la pila

PULA ; extrae el último elemento guardado en la pila y lo almacena en el acumulador

### 7.3. Software de Programación y simulación

Para la simulación se utiliza el software winIDE, que es un entorno de desarrollo integrado por varios programas, de los cuales se usa el compilador y el simulador.

La pantalla principal del winIDE se muestra en la figura siguiente, donde se puede observar un programa ejemplo escrito en assembler. En este ejemplo se distinguen las etiquetas inicio, siguiente y fin. Dichas etiquetas se deben escribir a partir de la primera posición de la hoja en el lado izquierdo, este espacio está reservado para etiquetas y no se deben escribir instrucciones para evitar errores de compilación.

El código de programa se debe escribir dejando una o dos tabulaciones a la derecha, tal cual se observa en la Figura 49.

Para escribir comentarios se utiliza el carácter “;”. Esto es importante para dejar bien documentado cada programa. También se debe tener en cuenta que los comentarios no deben ser muy extensos, de otra forma el compilador emitirá un error. Si es necesario un comentario extenso, lo mejor es continuar en el siguiente renglón.

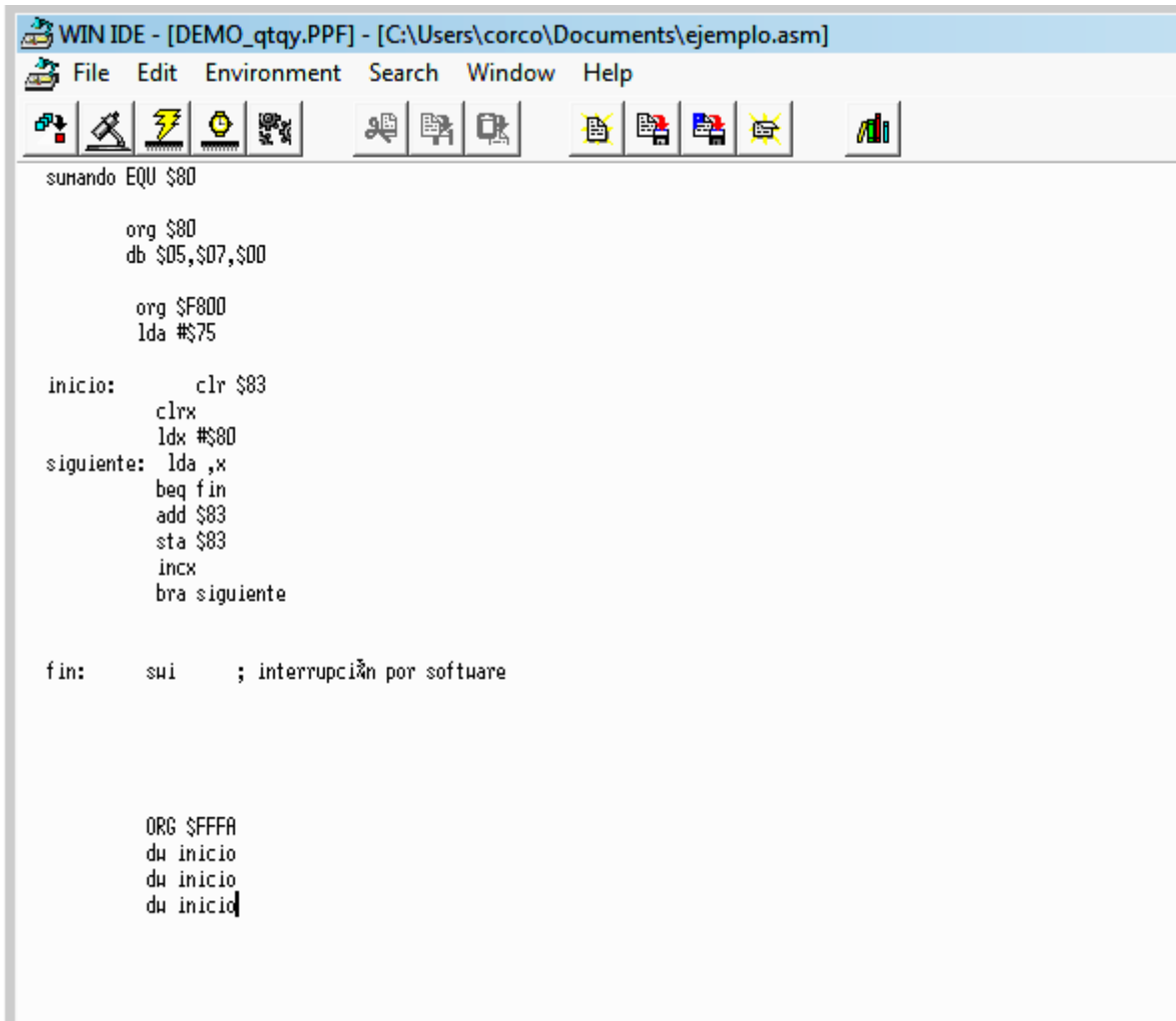


Figura 49. Entorno de Programación WinIDE

En la Figura 50 se muestran las diferentes herramientas de Software del entorno WinIDE, para verificar que el código no tiene errores se utilizará el compilador, el cual se ubica en la primera posición del menú que se muestra en dicha figura.

Una vez que se obtiene una compilación exitosa se puede pasar a la simulación presionando el quinto botón de la figura 50, que permite realizar simulación pura en la PC.

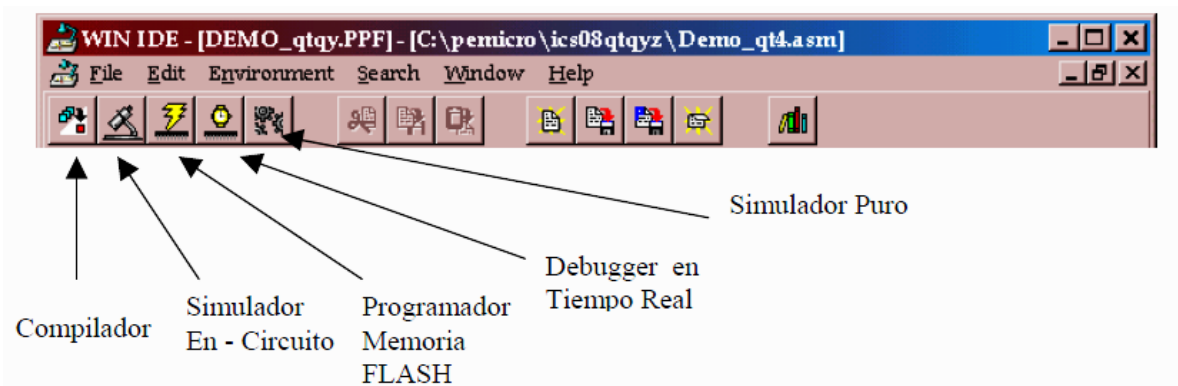


Figura 50. Herramientas de Software del entorno WinIDE.



Al terminar de escribir el programa en assembler se presiona entonces la opción de compilador y si el código no tiene errores de sintaxis se generará una ventana como la que se muestra en la Figura 51, donde en Status aparecerá la frase “Successful assembly” y aparecerá un tilde de color verde a la izquierda de ok. Al realizar la compilación se generará un archivo de código binario con extensión “s.19”, donde se almacena el programa en formato binario listo para ser almacenado en la memoria del sistema de procesamiento.

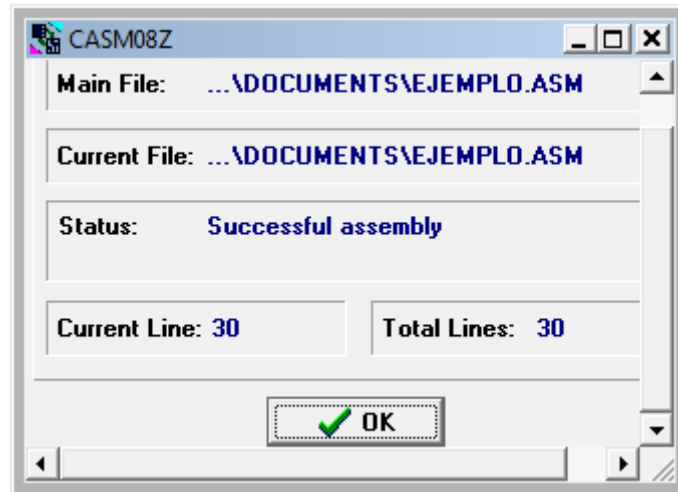


Figura 51. Compilación exitosa.

En esta instancia se sabe que el código no tiene errores de sintaxis, pero nos interesa saber si el programa hace lo que planificamos, para lo cual se usa el simulador. Al presionar la opción de simulador puro aparece la ventana que se muestra en la figura 52, aquí solo se debe presionar la opción “simulation only”. Al realizar la simulación por primera vez no aparecerá una ventana que nos permite elegir sobre que microcontrolador se desea ejecutar el programa, se recomienda seleccionar el HC809qy4 por ser el que más espacio de memoria posee.

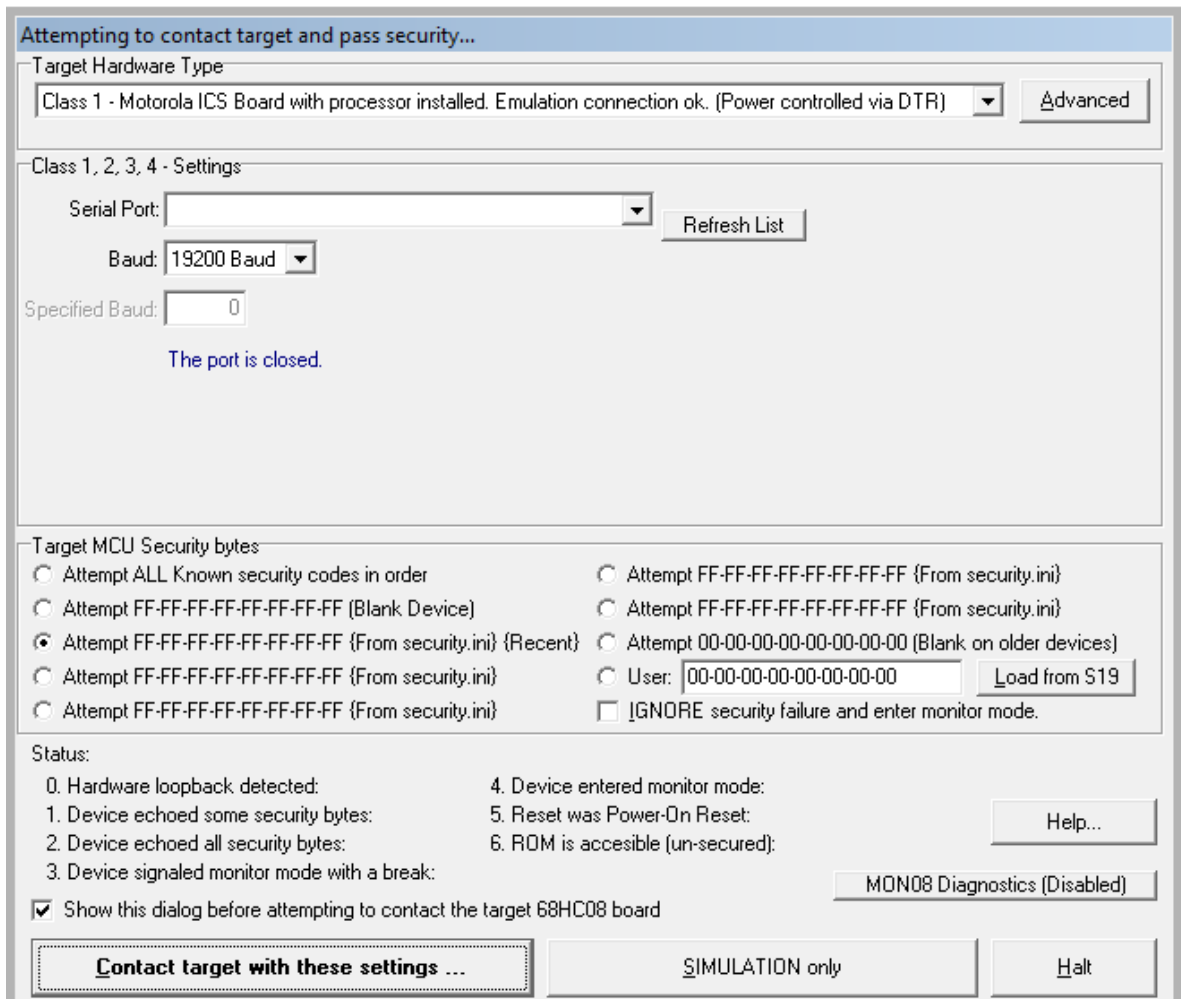


Figura 52. Ventana de selección de las características de la simulación.

Al abrirse la ventana de simulación se observarán las características que se muestran en la Figura 53. En la parte superior izquierda se mostrarán el estado de los registros del procesador en cada instancia de la simulación. En la parte inferior derecha se muestra una ventana que contiene tres columnas; la primera indica la dirección de memoria, la segunda el programa en hexadecimal y la tercera el programa en assembler. La flecha azul que se observa en esta ventana indica cual es la instrucción que se ejecutará en cada paso de la simulación.

Es muy importante en la ejecución de un programa poder conocer el contenido de la memoria de datos, para esto se tiene en la parte superior derecha una ventana que muestra el contenido de la memoria a partir de la dirección \$80 que es donde se encuentran los datos utilizados por el programa.

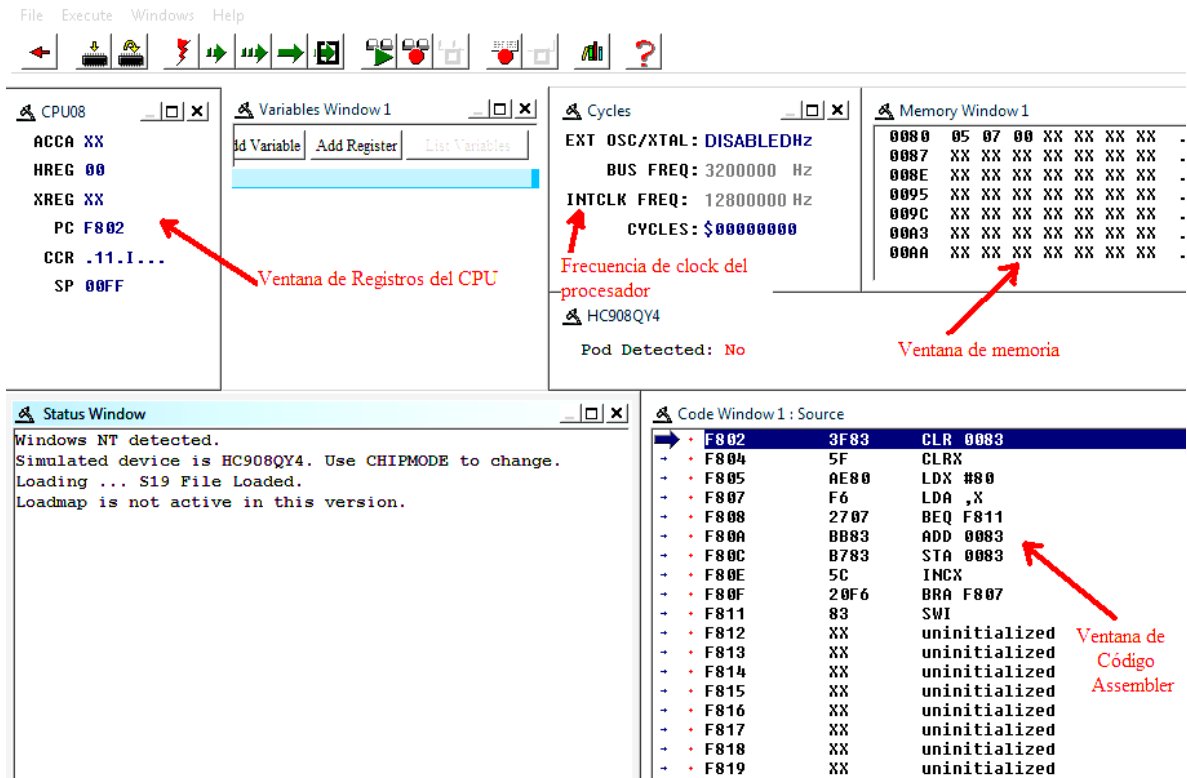



Figura 53. Ventana de simulación.

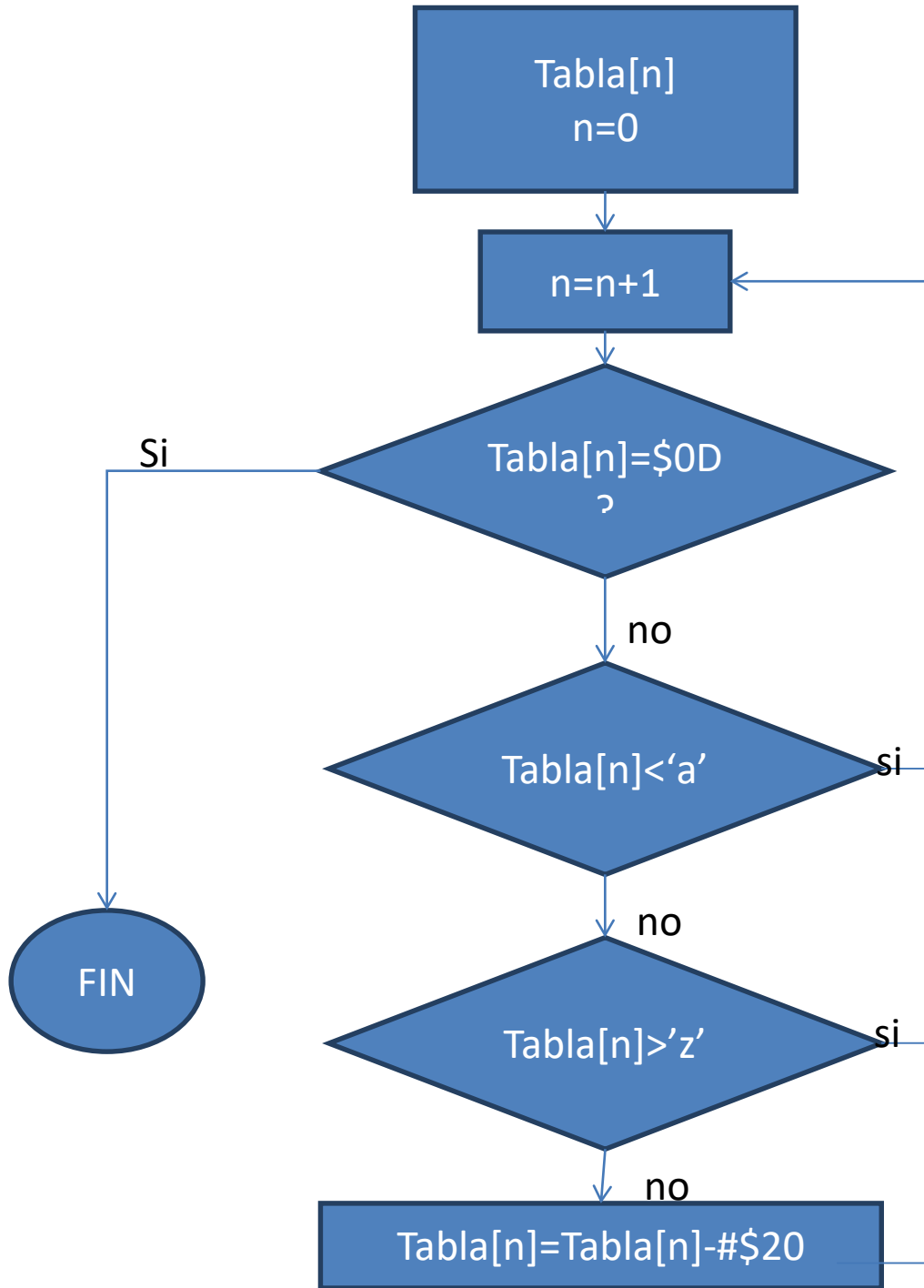
Para realizar la simulación paso a paso, se deberá presionar el botón , cada vez que se presione se ejecutará una nueva instrucción y se podrán visualizar los cambios en los registros y en la memoria de datos.

## 7.4. Planificación y programación de una aplicación en assembler

Para realizar una aplicación en assembler en primer lugar se debe planificar la misma mediante la descripción ordenada de cada una de sus partes. Esto se realiza mediante el diagrama de flujo, el cual no debe contener nombres de registros ni direcciones para que la descripción sea bien genérica. Luego se procederá a la programación siguiendo el orden indicado en el diagrama.

A continuación se realizará la planificación y programación de una aplicación que lee caracteres ascii desde una tabla en memoria que comienza en la dirección \$90 y termina con el carácter \$0D. La aplicación deberá detectar las letras minúsculas y reemplazarlas por mayúsculas en la misma posición en que fueron leídas. Cuando se detecte el carácter \$0D el programa finalizará.

### 7.4.1. Diagrama de flujo



## 7.4.2. Programación de la aplicación

```

    org $90 ; inicio de la memoria de datos (RAM)
    db 'A','f','B','c',$0D ; tabla de caracteres ascii a partir de
;la dirección $90
    ORG $EE00 ; inicio de la memoria de programa
    ldx #$8F
ciclo:  ;
    incx
    lda ,x
    CBEQA #$0D,fin
    cmp #$60
    ble ciclo
    cmp #$7B
    bhe ciclo
    sbc #$20
    sta ,x
    bra ciclo
fin:swi ; esta instrucción se pone para indicar el fin del programa

org $ffa ; dirección de inicio de los vectores de reset
dw $ee00 ;SWI
dw $ee00 ;IRQ
dw $ee00 ;RESET

```

## 7.4.3. Descripción del código

A partir de la dirección 90 se almacenan los datos de la tabla de forma consecutiva

```

    org $90
    db $6D,$41,$72,$43,$65,$0D ; tabla de caracteres ascii

```

A partir de la dirección EE00 se almacena el programa de la aplicación

```

ORG $EE00

```

Con la instrucción LDX cargo la dirección \$8F en el registro índice. Se debe aclarar que el registro índice es ideal para usar como puntero para recorrer una tabla de datos.

```

ldx #$8F

```

La etiqueta ciclo representa un bucle que se va a ejecutar hasta encontrar el carácter 0D en la tabla de caracteres

ciclo:

Se incrementa el puntero para apuntar a la dirección del siguiente carácter en la tabla

```

incx

```

Se guarda el caracter apuntado por x en el acumulador

```

lda ,x

```

Se compara el carácter del acumulador con el \$0D para saber si se llegó al final de la tabla. CBEQA es un salto condicional y en caso en que el contenido del acumulador sea igual a 0D, entonces se producirá un salto al final del programa. Si el contenido del acumulador es distinto de 0D, entonces se sigue ejecutando la siguiente instrucción.

```

CBEQA #$0D,fin

```

La siguiente instrucción compara el contenido del acumulador con el carácter a minúscula.

**cmp #'a'**

blo analiza si la comparación anterior dio que el contenido del acumulador es menor que 'a'. Si es menor significa que no es una letra minúscula y entonces se produce un salto a la etiqueta ciclo. En el caso de que el contenido del acumulador sea igual o mayor que 'a', entonces se ejecutará la siguiente instrucción

**blo ciclo**

Ahora se realiza una comparación entre el contenido del acumulador y la z minúscula

**cmp #'z'**

bhi analiza si el contenido del acumulador es mayor a la letra 'z'. En el caso de que sea mayor significa que el carácter del acumulador está no es una letra minúscula y se produce un salto a ciclo para analizar el siguiente carácter. Si el contenido del acumulador es menor a 'z', significa que la letra es una minúscula y se ejecutará la siguiente instrucción

**bhi ciclo**

La siguiente instrucción le resta 20 al contenido del acumulador, esto significa que se está convirtiendo la minúscula que está en el acumulado en mayúscula, debido a que las mayúsculas se encuentran 20 posiciones debajo de las minúsculas. El resultado de la resta se guarda en el acumulador.

**sbc #\$20**

La siguiente instrucción guarda el contenido del acumulador en la posición de memoria a la que apunta x. En otras palabras se está reemplazando el carácter de la tabla por una mayúscula.

**sta ,x**

La instrucción bra realiza un salto obligado a la etiqueta ciclo para analizar el siguiente carácter de la tabla.

**bra ciclo**

La etiqueta fin indica que cuando termine de ejecutarse la aplicación se producirá una interrupción por software que reiniciará el programa.

**fin: swi**

# CAPÍTULO 8

## Características de la programación en alto nivel

*Jorge R. Osio*

### 8.1. Introducción

La programación en C de alto nivel permite al programador abstraerse de las características del procesador que se está utilizando. Esto quiere decir que el programador no necesita conocer los registros internos del procesador, no requiere conocer el tamaño de datos con el que trabaja el procesador ni la arquitectura interna del mismo.

En cuanto a las características de código, programar en alto nivel posibilita escribir programas más compactos y con mucho menos cantidad de líneas. Cada sentencia en C equivale a varias en assembler. En esta sección se describirán las bondades de programar en alto nivel, las características de una programación eficiente y los pasos para lograr un código modularizado y reutilizable.

### 8.2. Programación en C eficiente para sistemas embebidos

Para producir un código de máquina más eficiente, el programador debe construir el programa cuidadosamente. En este contexto “Código Eficiente” significa tamaño de código compacto y rápido tiempo de ejecución. A continuación se detallan algunas sugerencias y consejos para ayudar al programador a escribir código C de manera que un compilador pueda convertirlo en código de máquina eficiente.

#### 8.2.1. Tipos de datos Básicos

Los tipos de datos básicos son los de la tabla siguiente.

**Tabla 5. Tipos de datos básicos**

Tipos de datos Básicos	Tamaño	Rango sin signo	Rango con signo
Char	8 bits	0 a 255	-128 a 127
Short int	16 bits	0 a 65535	-32768 a 32767
Int	16 bits	0 a 65535	-32768 a 32767
Long int	32 bits	0 a 4294967295	-2147483648 a 2147483647

La mejor performance en tamaño de código y tiempo de ejecución se puede lograr definiendo el tipo de datos más apropiado para cada variable. Esto es apropiado para microcontroladores de 8 bits donde el tamaño de datos interno natural es de 8 bits. El tipo de datos preferido en C es el "int". El estándar ANSI no define precisamente el tamaño de los tipos de datos naturales, pero los compiladores para microcontroladores de 8 bits implementan "int" como un valor de 16 bits con signo. Como los microcontroladores de 8 bits pueden procesar tipos de datos de 8 bits más eficientemente que de 16 bits. Los tipos de datos "int" y "long int" deben ser usados solo donde se requiere en función del tamaño de datos a ser representado. Las operaciones de doble precisión y punto flotante son ineficientes y deben ser evitadas donde la eficiencia es importante. Esto quizás parece obvio, pero frecuentemente se pasa por alto y tiene un enorme impacto sobre el tamaño de código en tiempo de ejecución.

Como se sabe la magnitud del tipo de datos requeridos y el signo deben ser especificados. El estándar ANSI C especifica 'int' con signo por defecto, pero el 'char' no está definido y puede variar entre compiladores. Por lo tanto para crear código portable, el tipo de datos 'char' no debe ser usado. Y para el tipo 'char' el signo debe ser definido explícitamente: 'unsigned char' o 'signed char'.

Es buena práctica crear definiciones de tipos para estos tipos de datos en un archivo encabezado (con extensión .h) que se incluye en todos los otros archivos. También vale la pena crear definiciones de tipos para todos los otros tipos de datos usados, por consistencia, y para permitir portabilidad entre compiladores. Se puede usar algo así:

```
typedef unsigned char UINT8;
typedef signed char SINT8;
typedef unsigned int UINT16;
typedef int SINT16;
typedef unsigned long int UINT32;
typedef long int SINT32;
```

Una variable se usa en más de una expresión, pero algunas de estas expresiones no requieren el tamaño de datos completo o con signo de la variable. En este caso el ahorro se



puede hacer definiendo la variable o parte de la expresión que contiene la variable con el tipo de datos más apropiado.

### 8.2.2. Variables locales vs variables globales

Las variables pueden estar clasificadas por su alcance. Las variables globales son accesibles por cualquier parte del programa y son almacenadas permanentemente en RAM. Las variables locales son accesibles sólo por la función dentro de la cual son declaradas y son almacenadas en la pila.

Las variables locales por consiguiente sólo ocupan RAM mientras se ejecuta la función a la cual pertenecen. Su dirección absoluta no se puede determinar cuando el código es compilado y mapeado, así es que son ubicadas en memoria relativa al puntero de pila. Para acceder a las variables locales el compilador puede usar el modo de direccionamiento del puntero de pila. Este modo de direccionamiento requiere un byte adicional y un ciclo adicional para acceder a una variable, comparado a la misma instrucción en el modo de direccionamiento indexado. Si el código requiere varios accesos consecutivos a las variables locales, entonces el compilador transferirá el puntero de pila al registro índice de 16 bits y usará direccionamiento indexado en lugar de este. El acceso 'estático' modificado puede ser usado con variables locales. Esto causa que la variable local sea almacenada permanentemente en la memoria, como una variable global, así es que el valor de la variable es preservado entre llamados a funciones. Sin embargo la variable local estática es sólo accesible por la función dentro de la cual está declarada.

Las variables globales son almacenadas permanentemente en memoria en una dirección absoluta determinada cuando el código es compilado. La memoria ocupada por una variable global no puede ser reusada por cualquier otra variable. De cualquier manera las variables globales no están protegidas, ya que cualquier parte del programa puede tener acceso a una variable global en cualquier momento. Esto da origen al asunto de consistencia de datos para las variables globales de más de un simple byte de tamaño. Esto quiere decir que los datos variables podrían ser corrompidos si una parte de la variable proviene de un valor y el resto de la variable proviene de otro valor. Los datos inconsistentes aparecen cuando una variable global es accedida (leída o escrita) por una parte del programa y antes de que cada byte de la variable haya sido accedido el programa se interrumpe. Esto se puede deber a una interrupción de hardware por ejemplo, o un sistema operativo, si se usa uno. Si la variable global es entonces accedida por la rutina de interrupción entonces pueden aparecer los datos inconsistentes. Esto se debe evitar si se desea la ejecución confiable del programa y se logra frecuentemente deshabilitando las interrupciones al acceder a las variables globales.

El acceso 'estático' modificado también puede ser usado con variables globales. Esto da algún grado de protección a las variables como restringir el acceso a la variable para esas funciones en el archivo en el cual la variable ha sido declarada.

El compilador generalmente usará el modo de direccionamiento extendido para acceder a las variables globales o el modo de direccionamiento indexado si son accedidas a través de un puntero. El uso de variables globales generalmente no resulta significativamente más eficiente en código que las variables locales. Hay algunas excepciones limitadas para esta generalización, una es cuando la variable global está ubicada en la página directa.

Sin embargo, el uso de variables globales impide que una función sea recursiva o entrante, y frecuentemente no hace el uso más eficiente de RAM, lo cual es un recurso limitado en la mayoría de microcontroladores. El programador por consiguiente debe hacer una elección meticulosa en decidir que variables definir como globales en el ámbito. Las ganancias importantes en eficiencia algunas veces pueden obtenerse definiendo justamente unas pocas de las variables, más intensivamente usadas, como globales, particularmente si estas variables están ubicadas en la página directa.

### 8.2.3. Variables de Página directa

El rango de direcciones \$ 0000 a \$ 00FF es llamado la página directa, la página base o la página cero. En los microcontroladores M68HC08, la parte inferior de la página directa siempre contiene los registros I/O y de control y la parte superior de la página directa siempre contiene RAM. Después de un reset, el puntero de pila siempre contiene la dirección \$ 00FF. La página directa es importante porque la mayoría de las instrucciones del CPU08 tienen un modo de direccionamiento directo por medio del que pueden acceder a los operandos en la página directa en un ciclo de reloj menos que en el modo de direccionamiento extendido. Además la instrucción de modo de direccionamiento directo requiere un byte menos de código.

Un compilador no puede sacar ventaja del modo de direccionamiento directo eficiente a menos que las variables sean explícitamente declaradas para estar en la página directa. No hay en estándar ANSI modo de hacer esto y los compiladores generalmente ofrecen soluciones diferentes.

La cantidad de RAM en la página directa es siempre limitada, por eso solo las variables más intensivamente usadas deberían estar ubicadas en la página directa.

Muchos puertos de entrada/salida y registros de control están ubicados en la página directa y ellos deberán estar declarados como tal, así el compilador puede usar el modo de direccionamiento directo donde sea posible. Una forma de definir el registro es:

Se define el nombre del registro del puerto A y puerto B vinculándolo con su dirección en memoria como se muestra a continuación.

```
#define PortA (*((volatile UINT8 *)0x0000))
#define PortB (*((volatile UINT8 *)0x0001))
```

Cabe aclarar que la dirección 0x0000 corresponde al registro A y la 0x0001 al registro B.

#### 8.2.4. Bucles

Si un bucle debe ser ejecutado menos de 255 veces, se usa 'unsigned char' para el tipo bucle contador. Si el bucle debe ser ejecutado más que 255 veces, se usa 'unsigned int' para el bucle contador. Esto es porque la aritmética de 8 bits es más eficiente que la aritmética de 16 bits y la aritmética sin signo es más eficiente que con signo.

Si el valor del bucle contador es irrelevante, entonces es más eficiente el contador para el decremento y comparar con cero que para incrementar y comparar con un valor distinto de cero. Esta optimización no es efectiva si el bucle debe ser ejecutado con el bucle contador igual a cero, como cuando el bucle contador se usa para indexar un arreglo de elementos y el primer elemento debe ser accedido.

Si se usa el valor del bucle contador en expresiones dentro del bucle, entonces se debe mantener el tipo de datos más apropiado cada vez que se usa.

Cuando un bucle se ejecuta un número fijo de veces y aquel número es pequeño, como tres o cuatro, es frecuentemente más eficiente no tener un bucle. En lugar de eso, se escribe el código explícitamente tantas veces según lo solicitado. Esto dará como resultado más líneas de código C pero a cambio generará menos código ensamblador y puede ejecutarse mucho más rápido que un bucle. Los ahorros reales variarán, dependiendo del código a ser ejecutado.

#### 8.2.5. Estructuras de datos

Al programar en C es fácil crear estructuras de datos complejas, por ejemplo un arreglo de estructuras, donde cada estructura contiene un número de tipos de datos diferentes. Esto producirá código complejo y lento en un microcontrolador de 8 bits que tiene un número limitado de registros de CPU para usar por indexado. Cada nivel de referencia resultará en una multiplicación del número de elementos por el tamaño del elemento, con el resultado probablemente puesto encima de la pila en orden, para hacer el siguiente cálculo. Las estructuras deberán ser evitadas donde sea posible y las estructuras de datos mantenerse simples. Esto puede hacerse organizando datos en simples arreglos unidimensionales de un tipo de datos simple. Esto dará como resultado un gran número de arreglos, pero el programa podrá acceder a los datos mucho más rápidamente. Si las estructuras son inevitables, entonces no deberán hacerse pasar como un argumento de función o un valor de retorno de función, en lugar de esto deberán pasarse por referencia.

### 8.2.6. Ejemplos de código C eficiente y su código assembler equivalente

Los siguientes ejemplos están basados en las siguientes definiciones de tipos:

```
typedef unsigned char UINT8; //entero sin signo de 8 bits
typedef signed char SINT8; //entero con signo de 8 bits
typedef unsigned int UINT16; //entero sin signo de 16 bits
typedef int SINT16; //entero con signo de 16 bits
```

#### 8.2.6.1. Uso de Registros

Este ejemplo muestra manipulación de bits para un registro en paginado directo (port A) y para uno fuera de paginado directo (CMCR0). El Seteado o borrado de uno o más bits en una dirección que no es registro en el paginado directo requiere 7 bytes de rom y 9 ciclos de CPU. El Seteado o borrado de un simple bit en un registro en el paginado directo requiere 2 bytes de rom y 4 ciclos de CPU.

**Tabla 6. Consumo de bytes y ciclos de reloj en la asignación de valores a registros**

Código C	Código Ensamblador	Bytes	Ciclos
#define PORTA (*(volatile UINT8 *) (0x0000))			
#define CMCR0 (*(volatile UINT8 *) (0x0500))			
void register1(void)	LDHX		
{	#0x0500	3	3
CMCR0 &= ~0x01; /* clr bit1 */	LDA ,X	1	2
PORTA  = 0x03; /* set b1,2 */	AND #0xFE	2	2
PORTA &= ~0x02; /* clr bit2 */	STA ,X	1	2
}	LDA 0x00	2	3
	ORA #0x03	2	2
	STA 0x00	2	3
	BSET 0,0x00	2	3
	RTS	2	4
		1	4

#### 8.2.6.2. Copia de datos 1

Este es un ejemplo del uso inapropiado de 'int' para la variable 'i' que se usa en el bucle contador y en el arreglo índice. El compilador es forzado a usar 16 bit con signo aritméticos para calcular la dirección de cada elemento de dataPtr[ ]. Esta rutina requiere 50 bytes de ROM y con 4 iteraciones del bucle, ejecutados en 283 ciclos de CPU.

**Tabla 7. Consumo de bytes y ciclos de reloj en un for con índice int y contador ascendente**

Código C	Código Ensamblador	Bytes	Ciclos
UINT8 buffer[4];	PSHA	1	2
	PSHX	1	2
void	AIS #2	2	2
datacopy1(UINT8 * dataPtr)	TSX	1	2
{	CLR 1,X	2	3
int i;	CLR ,X	1	2
for (i = 0; i < 4; i++)	TSX	1	2
{	LDA 3,X	2	3
buffer[i] = dataPtr[i];	ADD 1,X	2	3
}	PSHA	1	2
}	LDA ,X	1	2
	ADC 2,X	2	3
	PSHA	1	2
	PULH	1	2
	PULX	1	2
	LDA ,X	1	2
	TSX	1	2
	LDX ,X	1	2
	PSHX	1	2
	LDX 3,SP	3	4
	PULH	1	2
	STA buffer,X	3	4
	TSX	1	2
	INC 1,X	2	4
	BNE *1	2	3
	INC ,X	1	3
	LDA ,X	1	2
	PSHA	1	2
	LDX 1,X	2	3
	PULH	1	2
	CPHX #0x0004	3	3
	BLT *-39	2	3
	AIS #4	2	2
	RTS	1	4

### 8.2.6.3. Copia de datos 2

En este ejemplo, el bucle contador y la variable son optimizados para un 'unsigned char'. Esta rutina ahora requiere 33 bytes de ROM, 17 bytes menos que en copia de datos 1. Con cuatro iteraciones, la copia de datos 2 ejecuta en 180 ciclos de CPU, 103 ciclos menos que en copia de datos 1. En este ejemplo el valor del bucle contador es importante; El bucle debe ejecutarse con i = 0, 1, 2 y 3. No se obtendrá ninguna mejora significativa, en este caso, por decrementar el bucle contador en lugar de incrementarlo. Tampoco se obtendrá ninguna mejora significativa, en este caso, si la variable 'buffer' se ubica en la página directa: La

instrucción 'STA buffer, X' usa direccionamiento directo en lugar de extendido, ahorrando un byte de código y un ciclo de CPU por iteración.

**Tabla 8. Consumo de bytes y ciclos de reloj en un for con índice de ocho bits**

Código C	Código Ensamblador	bytes	Ciclos
UINT8 buffer[4];	PSHA	1	2
	PSHX	1	2
void	PSHH	1	2
datacopy2(UINT8 * dataPtr)	TSX	1	2
{	CLR ,X	1	2
UINT8 i;	LDA ,X	1	2
for (i = 0; i < 4; i++)	ADD 2,X	2	3
{	PSHA	1	2
buffer[i] = dataPtr[i];	CLRA	1	1
}	ADC 1,X	2	3
}	PSHA	1	2
	PULH	1	2
	PULX	1	2
	LDX ,X	1	2
	TXA	1	1
	TSX	1	2
	LDX ,X	1	2
	CLRH	1	1
	STA buffer,X	3	4
	TSX	1	2
	INC ,X	1	3
	LDA ,X	1	2
	CMP #0x04	2	2
	BCS *-25	2	3
	AIS #3	2	2
	RTS	1	4

### 8.2.6.4. Copia de datos 3

En este ejemplo, los datos son copiados sin usar un bucle. Esta rutina requiere 23 bytes de ROM y se ejecuta en 36 ciclos de CPU. Ésta usa 10 bytes menos de ROM y 144 ciclos menos de CPU que en copia de datos 2. Si se copiaran 8 bytes, entonces este método requeriría 10 bytes más de ROM que copia de datos 2, pero se ejecutaría en 280 ciclos menos de CPU. Aunque hay ahorros potenciales que se dan si la variable 'buffer' está ubicada en la página directa, el compilador no saca mucha ventaja de ellos en este caso.

**Tabla 9. Consumo de bytes y ciclos de reloj en cuatro asignaciones que reemplazan al for**

Código C	Código Ensamblador	Bytes	Ciclos
UINT8 buffer[4];		1	2
void		1	2
datacopy3(UINT8 * dataPtr)	PSHX	1	1
{	PULH	1	2
buffer[0] = dataPtr[0];	TAX	3	4
buffer[1] = dataPtr[1];	LDA ,X	2	3
buffer[2] = dataPtr[2];	STA buffer	3	4
buffer[3] = dataPtr[3];	LDA 1,X	2	3
}	STA buffer:0x1	3	4
	LDA 2,X	2	3
	STA buffer:0x2	3	4
	LDA 3,X	1	4
	STA buffer:0x3		
	RTS		

### 8.2.6.5. Uso de un bucle en decremento

Si sólo importa el número de iteraciones y no el valor del bucle contador, es más eficiente el decremento del bucle contador y comparar la variable con cero. En este ejemplo la declaración 'for' requiere 7 bytes de ROM, y el incremento y test del bucle contador en dirección opuesta toman 6 ciclos de CPU por cada iteración. Esto ahorra 2 bytes de ROM y 9 ciclos de CPU por iteración comparada a la declaración 'for' en copia de datos 2. De cualquier forma esta optimización no puede ser aplicada a copia de datos 2 ya que el código dentro de este bucle es ejecutado con i= 4, 3, 2 y 1.

**Tabla 10. Consumo de bytes y ciclos de reloj en un for con índice de ocho bits en decremento**

Código C	Código Ensamblador	Bytes	Ciclos
Void			
loop1(void)			
{			
UINT8 i;	PSHH		
for (i=4; i!=0; i--)	LDA #0x04	1	2
{	TSX	2	2
/* code */	STA ,X	1	2
}		1	2
}	TSX	1	2
	DBNZ ,X,*-offset	2	4
	PULH	1	2
	RTS	1	4

### 8.3. Ventajas y Objetivos de la Programación en C

#### Ventajas

Posee las ventajas del *diseño "Top-Down"*:

- Enfoque orientado a la problemática (No es necesario que el desarrollador conozca el funcionamiento interno del sistema de HW).
- Tiempos de desarrollo reducidos.
- Se simplifica la detección de errores.
- Facilita el seguimiento del desarrollo

#### Objetivos:

Se busca que el programa desarrollado sea:

- Fácil de entender
- Fácil de depurar
- Fácil de verificar
- Fácil de mantener

Regla de Oro: **“Escribe software para otros como te gustaría que lo escriban para vos.”**

### 8.4. Características de código reusable

Para que un código sea fácilmente reutilizable debe cumplir con las siguientes características:

- Código Documentado
- Abstracción
- Modularidad
- Software en capas

#### 8.4.1. Código Documentado

Comentar el código sirve para entender rápidamente programas escritos con mucha anterioridad o programas escritos por otras personas. Normalmente, se intenta entender un programa escrito por otra persona cuando se requiere hacer una modificación en la aplicación.

Adicionalmente, si los comentarios se realizan de manera correcta y siguiendo ciertas reglas, mediante determinado software se puede generar documentación automáticamente.



Este no es un detalle menor, debido a que el programador detesta realizar informes sobre lo desarrollado.

El Mantenimiento de software se realiza por algunos de los siguientes motivos:

- Corrección de errores,
- Programación de nuevas características,
- Optimizaciones en la velocidad de ejecución o uso de memoria,
- Migración hacia otro hardware,
- Adaptación a distintas situaciones.

Para comprender de qué manera se debe comentar un programa correctamente se analiza el siguiente fragmento de código:

```
•X = X + 4;    /* sumo 4 a X */
•Flag = 0;    /* pongo flag en cero */
```

Como se observa en los comentarios anteriores no se está agregando ninguna información a lo que se puede interpretar de línea de código. En otras palabras, los comentarios agregados son una obviedad de la línea de programa. Además, comentar cada línea del programa impide leer la función real que cumple el código, es por eso que se recomienda hacer comentarios sobre un fragmento del código y no sobre cada instrucción. Un comentario útil sobre las líneas de código de ejemplo sería:

```
•X = X + 4;    /* Se suman 4 (mV) para corregir el offset del transductor */
•Flag = 0;    /* Significa que no se presionó ningún pulsador*/
```

### 8.4.2. Modularidad

La modularidad consiste en dividir un programa en bloques funcionales. Quiere decir que cada función cumplirá una tarea específica e indivisible. Esto es importante debido a que permite dividir un problema complejo en pequeños problemas. De esta manera, varios programadores podrían trabajar en el mismo problema a la vez y se aceleraría el tiempo de desarrollo de un sistema.

Las razones de la modularidad son:

- Abstracción funcional*: Permite reutilizar módulos de software desde múltiples lugares y aplicaciones.
- Abstracción en complejidad*: Dividir un sistema complejo en componentes pequeños y simples permite que varios usuarios puedan trabajar sobre el mismo en paralelo y facilita la comprensión funcional.
- Portabilidad*: Permite utilizar un mismo código en diferentes plataformas de HW.

Supongamos que tenemos que diseñar una estación meteorológica que lee los datos de temperatura, humedad y presión desde 3 sensores diferentes y los almacena en una memoria externa. Además, supongamos que la estación debe transmitir los datos leídos desde los sensores a un servidor mediante un puerto serie asíncrono.

Para la programación de este sistema deberíamos dividir la aplicación en pequeños módulos según las siguientes funciones:

- Lectura de temperatura
- Lectura de humedad
- Lectura de presión
- Almacenamiento en memoria
- Menú de comandos
- Transmisión y recepción de comandos

Hasta aquí se logró dividir el problema en pequeños módulos. Estos módulos serán reutilizables, ya que si se requiere incorporar un sensor de temperatura a una aplicación, simplemente se debe llamar a la porción de código que lee el sensor de temperatura. Ahora, para poder reutilizar esos módulos en una plataforma diferente se deben seguir una serie de pasos.

1. En primer lugar se deberá crear un archivo con extensión `.c` por cada módulo definido. En estos archivos se escribirán las funciones que cumplan con la tarea definida por el módulo. Es indispensable no utilizar nombres de registros o direcciones que hagan alusión a una tecnología de microcontrolador para que los archivos `.c` sean completamente reutilizables en otras plataformas. Para el ejemplo los archivos serán los siguientes:

- a. `Temperatura.c`
  - `Lectura_temp ()`
- b. `Humedad.c`
  - `Lectura_humedad ()`
- c. `Presion.c`
  - `Lectura_presion ()`
- d. `Memoria.c`
  - `Escritura_memoria(datos, direccion)`
  - `Lectura_memoria(datos, direccion)`
  - `Borrar_memoria(sector)`
- e. `Menú.c`
  - `Almacenar_sensor(sensor, destino)`
  - `Leer_sensor(sensor)`
  - `Transmitir_sensor(sensor)`
- f. `Comunicación_serie.c`
  - `Inicializar_modulo_serie()`

- Recibir\_dato\_serie()
- Transmitir\_dato\_serie()
- Dato\_disponible()

2. Los archivos .c van acompañados de un archivo .h con el mismo nombre. El objetivo de los archivos .h es lograr la portabilidad del código, es decir, que el archivo .h contenga nombres de registros o características que lo vinculen a una tecnología de microcontrolador y el .c solo código puro. En cada .h se definirán:

- a. Mediante *#define* se definen nombres que representan la funcionalidad y utilizan algún registro.
  - i. Ejemplo, *#define* temperatura ADR  
`#define temp_disponible ADSCR_COCO`
- b. También se pueden definir constantes como por ejemplo
  - i. *#define* lectura\_humedad 1  
`#define lectura_presion 2`  
`#define lectura_temperatura 3`
- c. Por último se deben declarar todas las funciones que se definirán en el archivo .c

### 8.4.3. Niveles de abstracción

Para facilitar la programación se deben definir dos niveles de abstracción.

- En primer lugar el nivel de abstracción de HW, que tiene que ver con la parte del programa que interactúa con el HW, por ejemplo configuración de puerto como entrada o salida, uso de ADC, protocolos de comunicación UART, SPI, I2C, etc.
- Luego, el nivel de usuario que se será todo lo visible por el usuario del dispositivo. Por ejemplo, menú de opciones, valores devueltos por los sensores, opciones de teclado.



Figura 54. Niveles de abstracción.

#### 8.4.4. Programación en Capas

La programación en capas permite independizar totalmente el HW de la aplicación. A nivel de usuario no se tendrá información del HW sobre el cual se está ejecutando la aplicación. A nivel de Driver se configurará el HW para que responda como lo requiere la aplicación. Por último la API será la etapa de adaptación entre el HW y el usuario y es la que permite aislar totalmente el HW del usuario.

##### Reglas:

- Un módulo puede **hacer una llamada a otro módulo en la misma capa**
- Un módulo puede **llamar a un módulo de una capa inferior solo utilizando la API**
- Un módulo **no puede acceder directamente a ninguna función o variable en otra capa** (sin utilizar la API)
- Un módulo **no puede llamar a una rutina de mayor nivel.**



Figura 55. Programación en capas.

## **SEGUNDA PARTE**

---

Descripción y Configuración de los Módulos  
(Basado en los procesadores  
Freescale HC08 y HCS08)

Cátedra de Circuitos digitales y Microprocesadores

# CAPÍTULO 9

## Módulos disponibles en las diferentes líneas de microcontroladores

*Jorge R. Osio*

La gran diferencia entre los Microprocesadores [4] y [10] y los microcontroladores es que estos últimos tienen espacio de memoria y una variedad de módulos periféricos que le permiten interactuar directamente con el mundo Exterior de diferentes maneras.

La memoria se puede dividir en memoria Flash (para el almacenamiento del programa), Memoria RAM (en donde se crean las variables) y Registros de control y estados.

Dentro de los módulos Periféricos se pueden distinguir dos grupos, aquellos que se encargan de contribuir, asegurar y monitorear el buen funcionamiento del MCU y los que permiten comunicar al MCU con dispositivos externos de diferentes maneras y mediante diferentes protocolos.

En esta Sección se desarrollarán los módulos que incluye la Familia HC908QT/QY y la Familia HCS08JM60 / JM32 ([1], [15] y [16]). Por un lado, se describirán todos aquellos módulos que le permiten al microcontrolador interactuar con el exterior, considerados como módulos básicos por estar presentes en todos los microcontroladores de 8 bits. Por otro lado, se describirán los módulos de comunicaciones que le permiten interactuar al microcontrolador con dispositivos externos como, memorias, sensores, actuadores, computadoras, etc. Es importante destacar que el objetivo no es solo realizar una descripción de los módulos, sino también presentar aplicaciones y configuraciones sobre diferentes circuitos digitales típicos que se conectan a estos módulos.

Entre los módulos básicos que se encuentran en la mayoría de los microcontroladores se tienen los puertos de entrada/salida digitales que pueden interpretar unos y ceros; módulo de interrupción por teclado, que permite detectar cuando se presionan pulsadores por medio de interrupciones; el módulo ADC que permite digitalizar entradas analógicas; módulos de temporización que permiten generar retardos en diferentes unidades de tiempo, realizar captura de entrada y generar salidas PWM (modulación por ancho de pulso) y por último el módulo de interrupción externa IRQ de alta prioridad.

Los módulos que permiten implementar, mediante la configuración de registros, los protocolos de comunicaciones más comunes en el diseño digital se encuentran en microcontroladores más robustos como el HCS08JM60, que aunque no deja de ser un microcontrolador de 8 bits, viene provisto de interfaces muy potentes implementadas en HW,

permitiendo de esta manera utilizar todos los recursos del microprocesador HCS08 para procesamiento y dejar la labor de dichos protocolos a los módulos periféricos I/O.

En esta sección se incluyen ejemplos de aplicación que permiten ilustrar el funcionamiento y configuración de las distintas interfaces. La programación de los ejemplos fue realizada en lenguaje de bajo nivel para los módulos básicos y en C para los módulos más complejos debido a que sus protocolos requieren una gran cantidad de líneas de configuración si se realizan en lenguaje de bajo nivel. Además, la realidad indica que en la actualidad la mayoría de las aplicaciones complejas se implementan mediante lenguajes de alto nivel debido a la rápida configuración, la portabilidad del código y la gran cantidad de librerías disponibles para la utilización de los mismos. A continuación se realiza una pequeña introducción a los contenidos de esta sección.

El objetivo de este capítulo es comentar que cambios y que novedades se van incorporando en los Microcontroladores a medida que avanza la tecnología.

En esta sección se detallarán los módulos principales que puede contener hoy en día un MCU, pero conjuntamente con estos Módulos de altísimas prestaciones se han incorporado otros de aplicaciones específicas que permiten a dichos dispositivos interactuar directamente con cualquier otro periférico ([15] y [12]).

Entre las interfaces más comunes disponibles en los nuevos MCUs se encuentran la Ethernet, CAN, USB, I2C, SPI (serial síncrono), SCI (serial asíncrono) [15], interfaces inalámbricas, etc.

Por otro lado, el avance de la tecnología ha logrado que estos dispositivos alcancen frecuencias de funcionamiento arriba de los 100 MHz e incorporen módulos para procesamiento digital de señales, facilitando la realización de operaciones como la correlación, la implementación de transformadas como la FFT y de filtros FIR.

Hoy en día es muy común encontrar microcontroladores con módulos controladores de memoria SD, de display TFT y todo tipo de periférico que aporte autonomía al MCU.

Se ha llegado a un punto en donde algunos microcontroladores permiten implementar un Sistema Operativo Embebido dentro de ellos, facilitando de esta manera la configuración y el uso de cualquiera de las interfaces disponibles en ellos.

## 9.1. Módulos básicos

Los módulos básicos disponibles en la familia HCS08JM32/60 ([1] y [16]) son:

- **Módulos I/O digitales:** Este dispositivo posee 7 puertos digitales (A-G) para comunicarse con cualquier tipo de dispositivo. Estos puertos comparten funcionalidades con los demás módulos.
- **Módulo IRQ:** Es un módulo que permite al MCU recibir interrupciones externas mediante un puerto de entrada específico.

- **Módulo MCG:** Generador de clock multipropósito. El cual permite obtener la señal de clock de diferentes formas y permite generar señales de clock multipropósito.
- **Módulo COP:** Ayuda a recuperar la línea de ejecución de código del usuario.
- **Módulo LVD:** Realiza la supervisión de la fuente de alimentación.
- **Módulo SVR:** sistema regulador de voltaje.
- **Módulo TIM o TPM:** Provee Diferentes modos de temporización mediante contadores, divisores de la frecuencia de bus, captura de entrada, comparación de salida y hasta modulación por ancho de pulso (PWM).
- **Módulo RTC:** Reloj de tiempo real, es un módulo que permite contar en diferentes unidades de tiempo, uS, mS, seg., etc, útil en la temporización de aplicaciones aunque con mayor tasa de error que los módulos TPM y TIM.
- **Módulo KBI:** Permite generar interrupciones externamente utilizadas para detectar pulsadores.
- **Módulo ADC:** Permite Digitalizar señales analógicas con una resolución de entre 255 niveles para conversiones de 8 bits hasta 1024 niveles para conversiones de 10 bits.

## 9.2. Módulos de comunicaciones disponibles en microcontroladores

En la actualidad la mayoría de los Microcontroladores contienen módulos para la implementación de interfaces de comunicación tales como, SPI, SCI, I2C y USB. Todos estos módulos vienen provistos de librerías específicas que facilitan la configuración del módulo.

Entre los módulos de comunicación más comunes en la familia de microcontroladores HCS08 se enumeran, el SCI (serial communication interface), el SPI (serial peripheral interface), el I2C o IIC (Inter-Integrated Circuit) y el USB (Universal Serial Bus). En esta sección se desarrollarán los módulos más importantes de la familia HCS08 [15].

### 9.2.1. SCI – Serial Communication Interface

La interfaz de comunicación serie es una interfaz asincrónica que permite enviar datos entre dispositivos en forma serie sobre una única línea de transmisión y una única línea de recepción. Su característica principal es que la línea se encuentra normalmente en alto y cuando se desea enviar información se debe enviar un bit de inicio “un cero lógico” y seguido de este los bits de datos.



### 9.2.2. SPI – Serial Peripheral Interface

Esta interfaz permite enviar datos en forma serie, pero a diferencia de la anterior, requiere de sincronismo para el envío de datos. Por lo que se utiliza una línea de clock para indicar cuándo se enviará un dato, obviamente sincronizado con el clock.

Este protocolo permite comunicar un dispositivo Master con varios Slaves. Lo que quiere decir es que el Master puede decidir con que dispositivo intercambiar datos, poniendo en cero la línea de selección (CS – chip select) correspondiente al esclavo seleccionado.

### 9.2.3. USB – Universal Serial Bus

Esta interfaz es mucho más compleja que las anteriores. Es la interfaz por excelencia de las computadoras actuales y de todos los dispositivos de última generación. Este protocolo es muy eficiente para el envío de datos y requiere de un dispositivo Host que maneja la comunicación y de uno o varios dispositivos Slaves que se conectan al host para iniciar una transferencia de datos. Este protocolo permite la conexión de varios Slaves hacia el host mediante la utilización de un Hub USB.

Una de las grandes ventajas de este protocolo es que cada dispositivo conectado se enumera automáticamente indicando al Host sus características principales y permitiendo que este instale los drives para permitir el correcto funcionamiento del dispositivo USB conectado.

Este protocolo tiene varios modos de transferencia, dependiendo del tipo de dispositivo que se esté conectando y de su funcionamiento.

También se debe destacar que el protocolo soporta varias velocidades de comunicación y que las velocidades más altas requieren un diseño de HW específico.

Para la comunicación requiere solo dos líneas de datos diferenciales, pero los paquetes de comunicación son mucho más complejos que cualquiera de los protocolos anteriores.

## 9.3. Kit de Desarrollo

Para poder implementar los ejemplos de aplicación que se desarrollarán en esta sección se dispone de un kit de desarrollo que fue diseñado con el objeto principal de entender e iniciarse en el uso y programación de los sistemas microcontrolados. Además, poder sacarle todo el provecho al MC9S08JM60 de Freescale que es un potente MCU de 8bits con amplias prestaciones.

El kit Presenta la facilidad de poder programar el MCU por medio de un programador llamado USBDM, que se conecta al puerto nombrado sobre la placa base como “PROG.”, pudiendo así programar el JM60 vía el software CodeWarrior sin ningún problema de compatibilidad o de librerías.

La placa tiene a disposición todos los pines del MCU por medio de 2 headers dispuestos a los laterales con las nomenclaturas correspondientes a sus funciones principales para una fácil identificación. En el perímetro del zócalo del MCU se encuentran dispuestos leds, pulsadores, switches y un potenciómetro para poder lograr una aplicación de forma directa y práctica (ver figura 56).

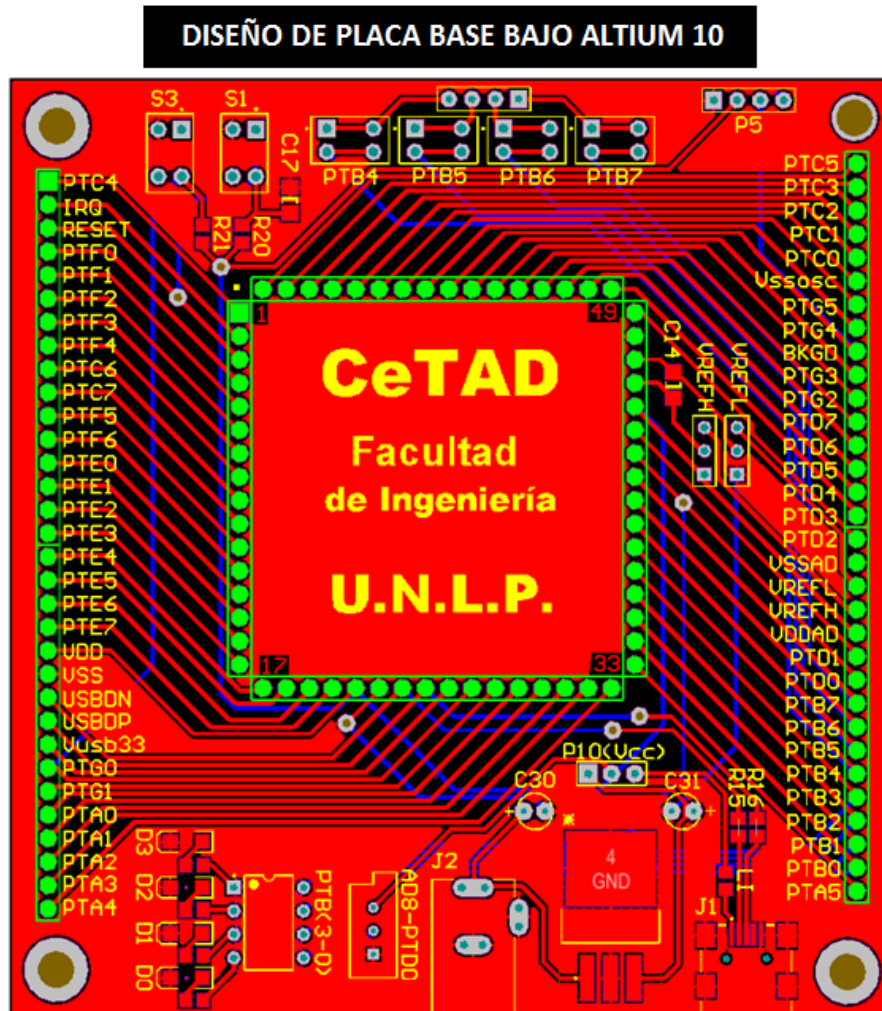


Figura 56. Circuito impreso de la placa base

Para el diseño se tuvo en cuenta la posibilidad de cambio de MCU (64 o 44 pines) por cualquiera de la familia hcs. Además, se buscó un fácil acceso a los pines por medio de conectores laterales. Se dejó espacio reservado para periféricos que faciliten la implementación de programas de aplicación.

Adicionalmente, la placa cuenta con múltiples opciones de alimentación y programación. Se puede alimentar directamente desde el programador, desde el conector USB o de una fuente externa mediante J2, para la selección de la fuente de alimentación se debe usar el jumper P10.

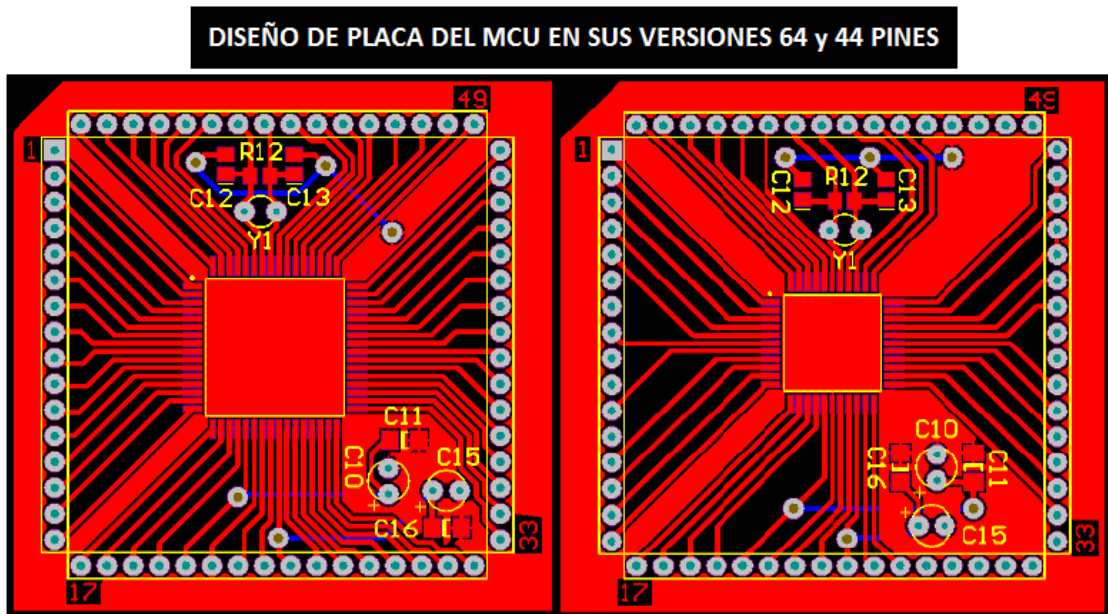


Figura 57. Placas de Microcontrolador

El circuito del MCU contiene todo lo que necesita el microcontrolador para funcionar, un cristal externo para el clock y capacitores de desacople entre VCC y GND (ver figura 57).

### 9.3.1. Características de la Placa Base

#### 9.3.1.1. Puertos I/O (entradas/salidas) disponibles

Por medio de las borneras se accede a todo el MCU, la bornera ubicada en la izq. facilita los pines del 1 al 32 y la de la derecha del 33 al 64. En primer lugar se puede resaltar que se tienen 7 puertos I/O para propósitos generales, PTA, PTB, PTC, PTD, PTE, PTF y PTG, los cuales pueden compartir sus pines con módulos particulares del MCU (microcontrolador). A continuación se los describirá con más detalle (*Entre “{}” se representan los pines en cuestión de la placa base*):

##### **PTA {28:33}:**

No comparte ninguna otra función con el MCU por lo tanto se puede disponer de cualquiera de sus 6 pines (**PTA [0:5]**) como entrada/salida sin ningún problema.

##### **PTB {34:41}:**

Este puerto comparte funciones con el **SERIAL PERIPHERAL INTERFACE MODULE 2** (SPI2) por **PTB [0:3]**, con el cual se puede comunicar al MCU con otro procesador, conversores, shift registers, etc. Tiene 2 de los 6 canales del **KEYBOARD INTERRUPT MODULE** por **PTB [4,5]** y 8 de los 12 canales del **ADC** por **PTB [0:7]**.

**PTC {1,9,60,61,62,63,64}:**

El mismo comparte **PTC[0,1]** con el módulo **IIC** que tiene el propósito de establecer comunicación con otros circuitos integrados, por medio de un pin SCL (Línea de Clock Serie) y uno SDA (Línea de Datos Serie). Además, por medio de **PTC[3,5]** se tiene una de las 2 interfaces serie de comunicación también conocida como **UART**.

**PTD {42,43,48,49,50,51,52,53}:**

A través de **PTC[2,3]** se puede acceder a 2 canales más del **KBI** del MCU; **PTD[0,1,3,4]** completan los 12 canales del **ADC**, y por medio de **PTD[0,1,2]** se puede utilizar el **Comparador Analógico o ACMP**.

**PTE {13:20}:**

Desde PTE[0,1] se accede a la segunda interfaz serie de comunicación (**UART**) y por **PTE[4:7]** a la **SERIAL PERIPHERAL INTERFACE MODULE 1** (SPI1). Además, comparte los pines **PTE[2,3]** 2 de los 6 canales del **TIMER/PWM MODULE 1** (TPM1).

**PTF {4:10}:**

En este puerto se encuentra los 4 restantes canales **TIMER/PWM MODULE 1** (TPM1), **PTF[0:3]**, así como también el **TIMER/PWM MODULE 2** (TPM2) que consta de 2 canales, **PTF[8,11]**.

**PTG {26,27,54,55,57,58}:**

Finalizando, **PTG[0,1,2,3]** completan el módulo **KBI** y **PTG[4,5]** son los terminales para un **crystal** externo.

Por último se pueden diferenciar los pines de alimentación **VCC{21}** y tierra **GND{20}** que se distribuyen por toda la placa base a partir de la elección de alimentación de la misma, y los pines de tensión de referencia del **ADC**, **VREFH{45}** y **VREFL{46}**, que pueden ser administrados de forma externa.

**9.3.1.2. Periféricos Disponibles**

En los alrededores del sócalo del MCU se encuentran diferentes periféricos listos para ser utilizados dispuestos e identificados de forma tal que sea simple y didáctico su uso. En la parte superior se encuentran los Pulsadores (ver figura 58), que se encuentran conectados al conector superior. Se debe aclarar que dicho conector se debe conectar a GND para poder usar los pulsadores configurando los puertos PTB4-PTB7 como entrada y con pull-up.

Sobre la derecha la selección de la tensión de referencia VREF del ADC, que permite mediante jumpers seleccionar como tensión de referencia la tensión de alimentación del MCU o un valor de tensión externa (figura 59a). Adicionalmente, en la figura 5a se muestra el potenciómetro que se encuentra conectado a la entrada AD8 (canal 8 del ADC). En la parte

inferior se dispone de 4 Leds seleccionables mediante un conjunto de Switchs como se muestra en la figura 59b.

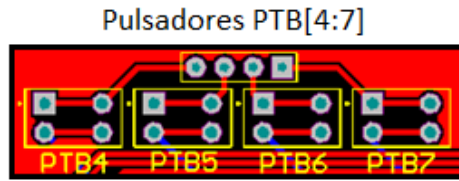


Figura 58. Disposición de los 4 pulsadores

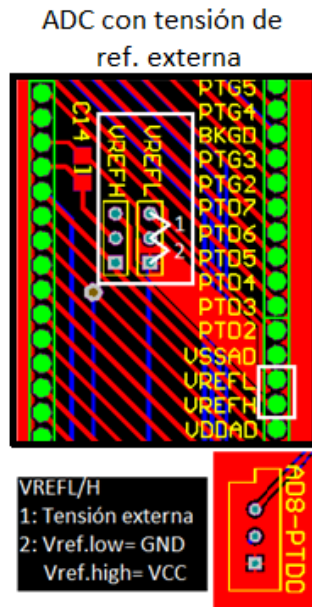


Figura 59a. Tensión de referencia ADC y potenciómetro

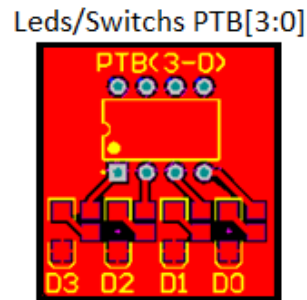


Figura 59b. Leds seleccionables

Figura 59. Periféricos ubicados en la parte inferior de la placa

### 9.3.1.3. Alimentación

El Kit tiene 3 formas de ser alimentado, por una entrada Regulada de 5V mediante el Jack P2, por una interfaz USB o por el Programador USBDM. Como se puede ver en la siguiente figura, la selección entre la entrada USB o la Regulada de 5V se realiza por medio del jumper P10, el cual en posición 2 permite alimentar por el primer medio y en la posición 1 mediante un cable USB. Es importante remarcar que la opción mediante el programador se habilita por software al configurar el modo de programación en la ventana de conexión del codewarrior. En este caso el jumper P10 deberá estar en la posición 2.

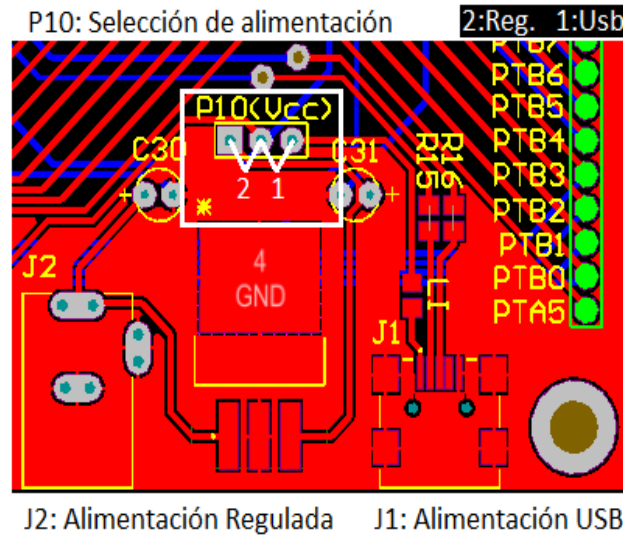


Figura 60. Conectores de alimentación

### 9.3.2. Opciones de Programación

El programador es un diseño de hardware/software/firmware de código libre denominado USBDM capaz de proporcionar programación y debug a las familias HCS12, HCS08, Coldfire V1 y RS08, de Freescale. Posee comunicación USB FullSpeed, de tamaño reducido y que además permite alimentar a la placa base. Compatible con CodeWarrior con la única necesidad de instalación de un driver que permite comunicar al programador vía USB. Con el driver instalado, y las DLL del programador ubicadas en la capeta Sistem32 de Windows, solo hace falta ingresar al CodeWarrior y seleccionar el debugger correspondiente al MCU. En la siguiente figura se resalta el conector de programación USBDM del programador.

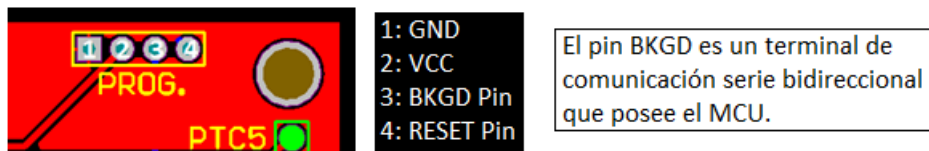


Figura 61. Conector de Programación de la placa base

El programador USBDM permite programar el microcontrolador y hacer debugging en circuito, lo que significa que se puede controlar la ejecución del código programado desde el microcontrolador. En la Figura 62 se muestra el circuito del programador, donde en el extremo izquierdo se encuentra el conector USB para conexión con la PC y en el derecho el conector BDM para conexión con la placa base.

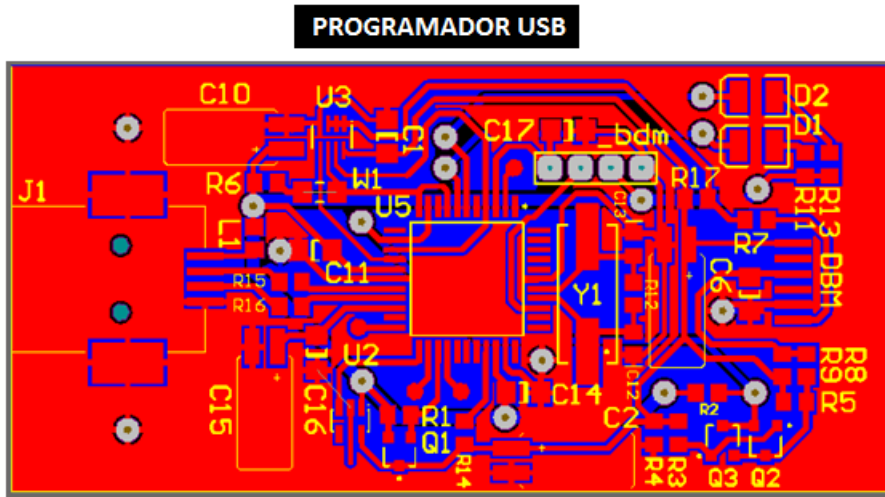


Figura 62. Progamador USBDM

# CAPÍTULO 10

## Puertos de entrada/salida

*Jorge R. Osio y Walter J. Aróztegui*

Los puertos de entrada/salida son puertos digitales que permiten recibir o transmitir valores lógicos (unos y ceros) desde el microcontrolador. Estos puertos se pueden utilizar para leer el estado de un pulsador, para el encendido y apagado de un led o para la recepción y transmisión de datos digitales en la comunicación con dispositivos externos.

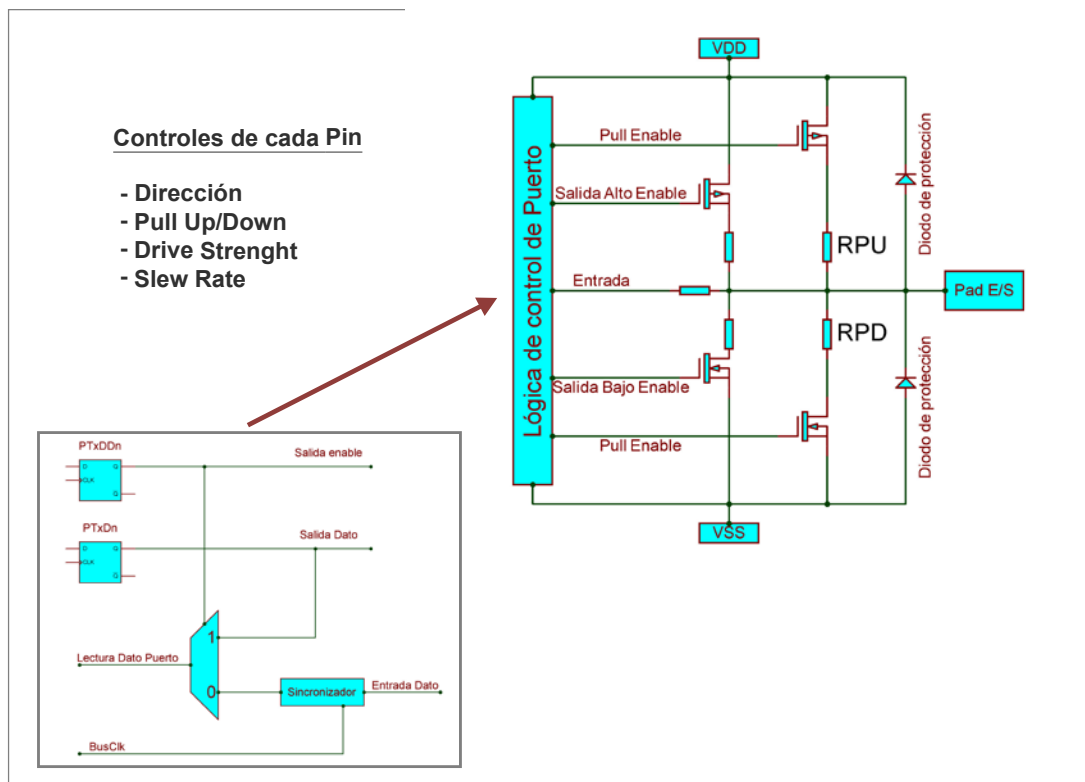


Figura 63. Esquema interno de los puertos de E/S

Entre las características eléctricas se debe tener en cuenta que cada pin del microcontrolador soporta como máximo 5,3 V y puede entregar hasta 5,8V.

Puede entregar por pin hasta 25 mA de corriente y en total sumando el consumo de todos los puertos puede entregar 100mA cuando el MCU se alimenta a 5V.



**Tabla 11. Registros de los puertos de entrada/salida**

Registros E/S paralelo	datos	PTxD	Valor del pin (R/W)
	Dirección de datos	PTxDD	Setea el pin como entrada o salida
Registros de control del pin	Habilitación de Pull-up	PTxPE	Habilita pull-up interno
	Habilitación de Slew	PTxSE	Habilita slew rate
	Drive Strength	PTxDS	Setea drive strength

## 10.1. Registro de datos

El registro de datos es el que contiene el valor del pin que se está utilizando en la aplicación. Cada puerto tiene un registro de datos de 8 bits, donde cada bit se corresponde con un pin de entrada/ salida del microcontrolador. El programador puede leer o escribir el puerto completo, o simplemente un pin del microcontrolador. El registro que contiene el valor de un puerto se llama PTxD, donde x representa el puerto correspondiente.

Como ejemplo, supongamos que se quiere asignar un 1 lógico al pin cero del puerto B que se encuentra configurado como salida. Una forma rápida de hacerlo, sin modificar el valor de los demás pines, es mediante una máscara como se muestra a continuación.

```
PTBD = PTBD | 0x01; //realiza la or bit a bit entre el contenido del puerto B y número 00000001
// esto modificará solo el bit menos significativo del puerto B
```

Para el caso en que se desee leer el estado de un pin del puerto B configurado como entrada, el procedimiento consiste aplicar una máscara, pero para el caso de la lectura se aplica mediante el operador lógico and.

```
char pinB7;
pinB7 = PTBD & 0x80 ; //la operación AND con la máscara 0x80 mantiene inalterado el bit 7 y
//pone a cero el resto de los bits de la variable pinB7
pinB7 >> 7 ; // desplaza a derecha siete posiciones el contenido de la variable pinB7
```

## 10.2. Registro de dirección

El registro de dirección es el PTxDD y permite configurar la dirección de los bits individuales de un puerto. Si se desea encender y apagar un led mediante el pin cero del puerto B se debe configurar como salida poniendo el bit correspondiente del registro PTBDD a uno lógico. Si se desea utilizar el pin 7 del puerto B como entrada para determinar el estado de un pulsador, se debe configurar el bit correspondiente del registro de dirección a cero lógico.

```
PTBDD = PTBDD | 0x01; //pone en 1 el bit 0 del puerto B y el resto de los bits no cambian
PTBDD = PTBDD & 0x7F; //pone el bit 7 en cero y el resto de los bits se mantienen inalterados
```

### 10.3. Registro de Pull-Up

Este registro permite configurar resistencias de pull-up en los pines que se configuran como entrada. Cuando un pin se configura como entrada, este actúa como una antena receptora de ruido ambiente, el cual es interpretado por el procesador como una sucesión de unos y ceros. Para configurar un pin como entrada y fijar un nivel conocido en dicha línea se utilizan los pull-ups.

Un pull-up es una resistencia de 10 Kohm conectada entre el pin y VCC (tensión de alimentación del MCU)

El registro que permite habilitar los pullups en un puerto es el PTxPE, para el caso del puerto B el registro es el PTBPE. Para configurar el pin 7 del puerto B como entrada y habilitar el pull-up para tener un estado lógico conocido (1 lógico) en la línea, se debe agrega la siguiente línea.

```
PTBPE = PTBPE | 0x80; // pone un 1 en el bit 7 del registro de pull-up del puerto B
```

### 10.4. Slow rate y Drive Strength

#### 10.4.1. Drive Strength

El drive strength (manejo de corriente) permite mejorar la capacidad de manejo de corriente de un pin sin deterioros apreciables en la tensión de unos o ceros lógicos. Se configura mediante el registro PTxDSn, donde x es el puerto que se quiere configurar, por ejemplo para el puerto B se configura el registro PTBDSn.

- Si el registro se pone en 0 (PTBDSn=0, manejo en bajo) para una tensión de 5V se tendrá una corriente de carga de 4mA
- Si el registro se pone en 1 (PTBDSn=1, manejo en alto) para una tensión de 5V se podrá manejar una corriente de carga de hasta 15mA con una mínima caída de tensión en el pin.

Con un mayor manejo de corriente en cada pin, se deberá tener cuidado con los límites totales de corrientes de los pines en conjunto. Además, se producirán mayores picos en las transiciones, generando en mayor grado perturbaciones electromagnéticas.

La figura 64 muestra un deterioro en el 0 lógico para una misma carga en el pin con Low Drive Strenght con respecto a High Drive Strenght.

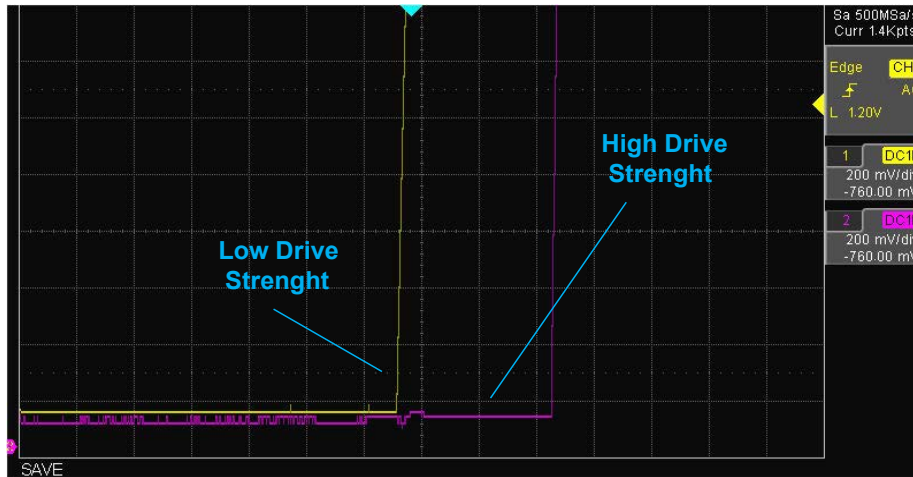


Figura 64. Efecto de Drive Strength

### 10.4.2. Slew rate

Slew rate (control de pendiente) consiste en suavizar las transiciones en los pines del microcontrolador, de esta manera se minimizan los picos durante los flancos ascendentes o descendentes, disminuyendo de esta manera las perturbaciones electromagnéticas propias de transiciones abruptas. Para la configuración se utiliza el registro PTxSEn, donde x representa el puerto a configurar.

Cuando PTxSEn=0 el slew rate se deshabilita y el cambio de cero a uno será brusco. Para el caso de PTxSEn=1 el cambio de cero a uno será más suave como se muestra en la figura siguiente.

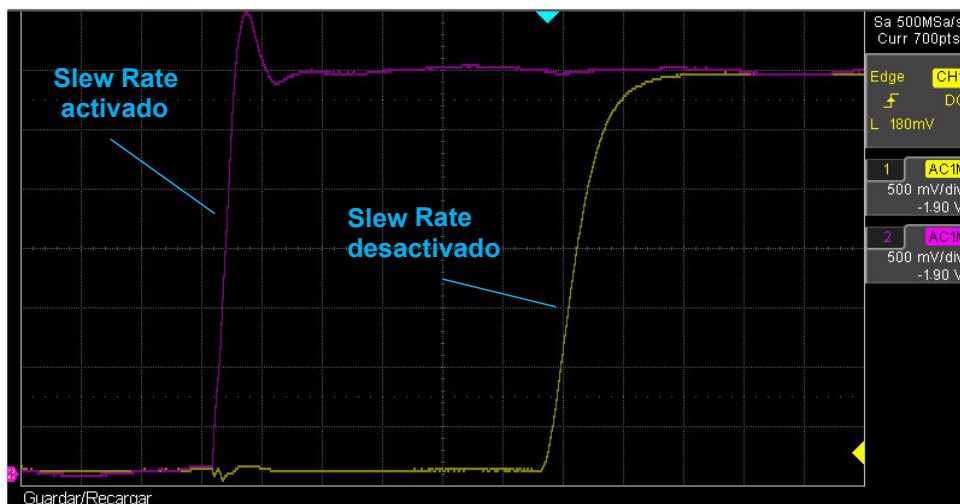


Figura 65. Efecto de Slew rate

## 10.5. Ejemplo de Aplicación

El ejemplo de aplicación consiste en leer el estado de un pulsador configurado como entrada para cambiar el estado del led que se encuentra configurado como salida. A continuación se describe el hardware y el software utilizado para generar esta aplicación.

### 10.5.1. Descripción de Hardware

La aplicación que aquí se describe tiene dos componentes principales:

- La aplicación lee el estado del pin 7 del puerto B que se encuentra configurado como entrada y con pullup, donde está conectado el pulsador.
- Si se lee un cero en la entrada PTB7, se cambiará el estado del led conectado al pin 0 del puerto B.

En la Figura 66 se observa que el pulsador se conecta al pin 7 del puerto B. Por otro lado, el led está conectado a tierra mediante una resistencia que limita la corriente máxima que se le exigirá al pin. Este se enciende cuando el pin PTB0 se encuentra configurado como salida y en nivel alto (1 lógico). Los otros pines requeridos para la aplicación son VDD y VSS. Se usa un capacitor de desacople entre VDD y tierra para reducir el nivel de ruido en la alimentación del microcontrolador, es decir, para soportar las caídas de tensión producidas por los cambios en el consumo. En el programa de la aplicación, los pines no utilizados se configuran como entradas y se habilitan las resistencias de pull-up internas, esto reduce la necesidad de hardware externo en los pines no utilizados.

**NOTA:** La conexión de cualquier pin no utilizado a tierra, con la resistencia de pull-up interno habilitada, podría causar un consumo mayor a 100  $\mu$ A por pin.

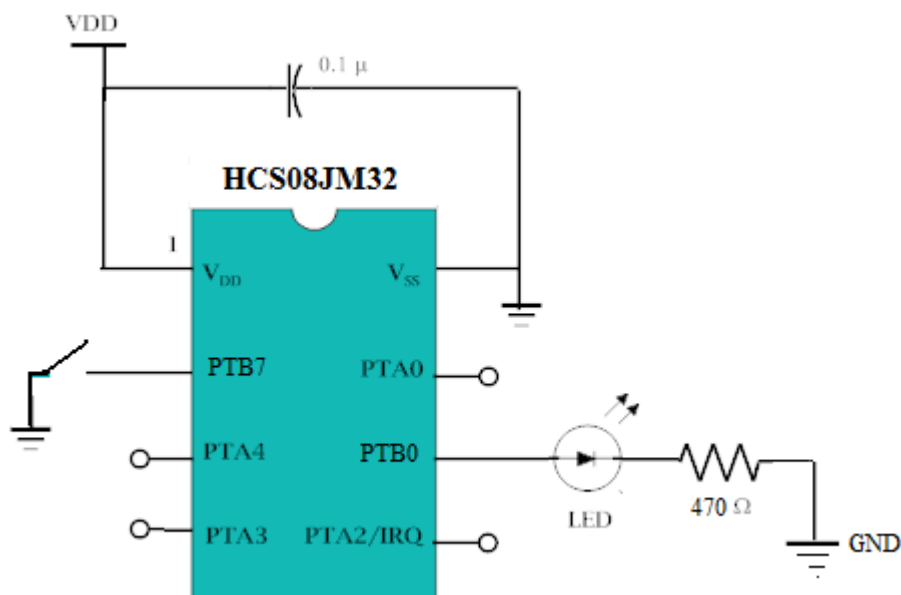


Figura 66. Circuito Eléctrico con pulsador y led

## 10.5.2. Código de la aplicación

```
#include <MC9S08JM60.h>
void main(void)
{
char pulsador;
PTBDD = PTBDD | 0x01; //pone en 1 el bit 0 del puerto B como salida
PTBDD = PTBDD & 0x7F; //pone el bit 7 del puerto B en cero como entrada
PTBPE = PTBPE | 0x80; // pone un 1 en el bit 7 del registro de pullup (pullup habilitado)
While (1)
{
pulsador = PTBD & 0x80 ; //se lee el estado del pulsador conectado al bit 7 del puerto B
  pulsador >> 7 ;          // se desplaza el valor leído a derecha siete posiciones hasta ubicarlo en
                          //el bit menos significativo

If(pulsador==0)
PTBD = PRBD^0x01; //cambia el estado del led aplicando una or exclusiva con un 1 lógico

} //fin bucle infinito
} //fin main
```

# CAPÍTULO 11

## Módulo de Interrupción Externa (IRQ)

*Jorge R. Osio y Walter J. Aróztegui*

Este módulo se desarrollará en función del microcontrolador HC908QY4, debido a que su funcionamiento no cambia de un microcontrolador a otro. El Pin /IRQ. (interrupción externa) [3], que se comparte con PTA2 (entrada de propósito general) en el HC908QY4, provee una interrupción enmascarable

### 11.1. Características

Entre las características del IRQ se incluyen:

- Pin de interrupción externa /IRQ.
- Bits de control de interrupción IRQ
- Detección de Interrupción programable por flanco o por flanco y nivel.
- Detección de interrupción automática
- Pull-up interno seleccionable.

### 11.2. Descripción Funcional

La funcionalidad del pin /IRQ se activa mediante el registro de configuración 2 (CONFIG2), específicamente el bit IRQEN, donde un cero desactiva la función IRQ y el pin PTA2 funcionará como pin de entrada / salida, un uno habilita la función IRQ.

Un nivel bajo aplicado al pin de requerimiento de interrupción externa (IRQ) puede generar un requerimiento de interrupción del CPU. La Figura 67 muestra la estructura del módulo IRQ. Las señales de Interrupción sobre el pin /IRQ se conservan en el latch IRQ (cerrojo IRQ). Dicho latch mantiene su estado hasta que se produce una de las siguientes acciones:

- Vector de carga IRQ (IRQ vector fetch) — Un vector de carga IRQ genera automáticamente una señal de acuse de interrupción que borra el latch IRQ.
- Borrado por Software, el Software puede borrar el latch IRQ escribiendo un 1 en el bit ACK del “registro de control y estado de interrupción” (INTSCR).
- Reset — Al resetear se borra automáticamente el latch IRQ.

El pin de interrupción externa IRQ se dispara con un flanco descendente fuera de un estado de reset y es configurable por software para ser detectado por flanco descendente o por flanco descendente y nivel bajo. El bit MODE en el registro INTSCR controla la sensibilidad de disparo del pin /IRQ.

Cuando se setea, el bit IMASK en la máscara del registro INTSCR, no se almacena el requerimiento de interrupción en la lógica de prioridad de interrupción. Esto solo sucederá cuando el IMASK sea borrado.

Se debe tener en cuenta que la máscara de interrupción (I) en el registro de código de condición (CCR) enmascara todos los requerimientos de interrupciones, incluyendo el requerimiento de interrupción IRQ, por lo tanto para habilitar las interrupciones de los distintos módulos se debe borrar el flag I del CCR.

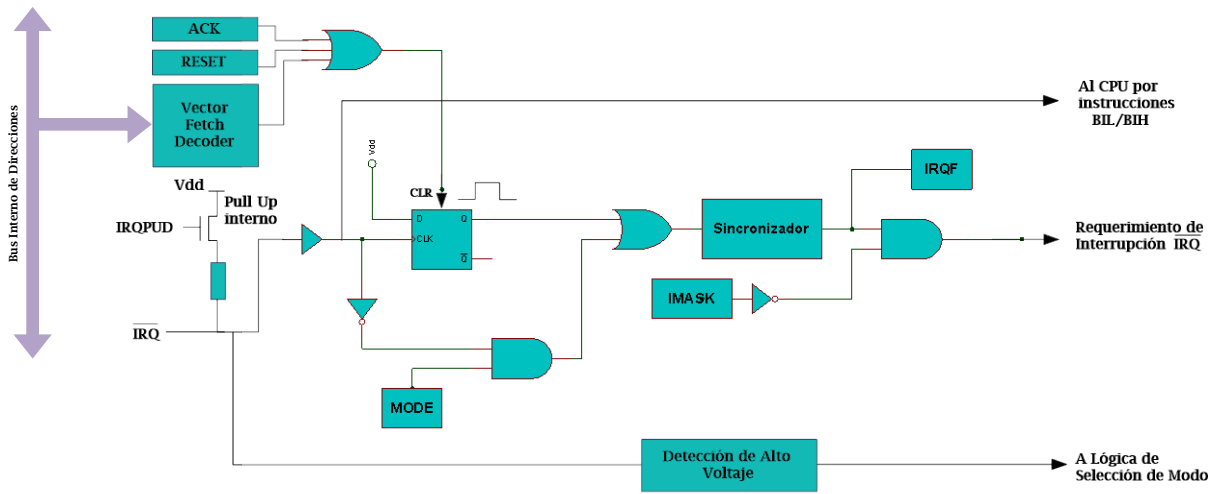


Figura 67. Diagrama en bloques del Módulo IRQ

### 11.2.1. MODE = 1

Si el bit Mode está seteado, el pin /IRQ tendrá sensibilidad por flanco descendente y por nivel bajo. Con el Mode seteado, se deben realizar las siguientes acciones para borrar el requerimiento de interrupción IRQ:

- Retorno del pin /IRQ a un nivel alto, ya que mientras el pin /IRQ esté en bajo, el requerimiento de IRQ seguirá activo.
- Borrado del vector de carga IRQ (vector Fetch IRQ) o el borrado por software. El vector fetch IRQ genera una señal de reconocimiento de interrupción para borrar el latch IRQ. Por Software se genera la señal de reconocimiento de interrupción escribiendo un 1 en el bit ACK del registro INTSCR. El bit ACK es útil en aplicaciones que consultan el pin IRQ y requieren borrar el latch IRQ por software. Escribiendo el ACK antes de abandonar una rutina de servicio de interrupción también se pueden prevenir interrupciones indeseadas causadas por ruido. Con el bit ACK en alto, las subsiguientes transiciones en el pin IRQ no

generarán requerimiento de interrupción. Un flanco descendente que ocurre después de escribir el ACK guarda otro requerimiento de interrupción. Si el bit IMASK, está en nivel bajo, la CPU [4] carga el contador de programa con la dirección que contiene el vector IRQ, para ejecutar la rutina de interrupción.

El vector fetch IRQ o el borrado por software y el regreso del pin /IRQ a un nivel alto, se pueden ejecutar en cualquier orden.

La solicitud de interrupción sigue pendiente mientras el pin /IRQ está en nivel bajo. Un reset borrará el latch IRQ y el bit de control MODE, borrando así la interrupción incluso si el pin permanece en nivel bajo. Se debe utilizar la instrucción BIH o BIL para leer el nivel lógico sobre el pin /IRQ.

### 11.2.2. MODE = 0

Si el bit MODE está en cero, el pin /IRQ solo se activará con un flanco descendente. Con el MODE borrado, el borrado del vector fetch IRQ o un borrado por software borra inmediatamente el latch IRQ.

El bit IRQF en el registro INTSCR se puede leer para chequear las interrupciones pendientes. El bit IRQF no es afectado por IMASK, lo que hace que sea útil en aplicaciones donde se prefiere hacer polling (lectura continua del estado del pin).

## 11.3. Interrupciones

La siguiente fuente de IRQ puede generar requerimientos de interrupción:

- flag de interrupción (IRQF) — el bit IRQF es seteado cuando el pin /IRQ se encuentra configurado en el modo IRQ. El bit de la máscara de interrupción IMASK se utiliza para habilitar o deshabilitar los requerimientos de interrupción por IRQ.

## 11.4. Modos de bajo Consumo

Las instrucciones WAIT y STOP ponen al MCU en modo de espera de bajo consumo. El módulo IRQ permanece activo en modo wait. Borrando el IMASK en el registro INTSCR se habilita el requerimiento de interrupción IRQ para sacar al MCU fuera del modo wait durante una interrupción. En el modo STOP sucede lo mismo que en modo wait.



## 11.5. Módulo IRQ durante el requerimiento de Interrupción

El módulo de integración del Sistema (SIM) controla que los bits de estado en otros módulos puedan borrarse durante el estado de interrupción (break). El bit BCFE en el flag de break del registro de control (BFCR) habilita al software a borrar el bit de estado durante el estado de break.

Para permitir el borrado por software del bit de estado durante un requerimiento de interrupción, se debe poner en 1 el bit BCFE. Si el bit de estado se borra durante el estado de interrupción, se mantendrá borrado cuando el MCU salga de dicho estado.

Para proteger el bit de estado durante el estado de interrupción, se debe escribir un cero en BCFE. Con BCFE en cero (estado predeterminado), el software puede leer y escribir registros durante el estado de interrupción sin afectar los bits de estado. Algunos bits de estado de lectura/escritura tienen un procedimiento de dos pasos de borrado. Si el software realiza el primer paso sobre uno de esos bits antes del requerimiento de interrupción, el bit no podrá cambiar durante el estado de interrupción hasta tanto BCFE sea borrado. Después de la interrupción, realizando el segundo paso se borra el bit de estado.

## 11.6. Señales de Entrada/Salida

El módulo IRQ comparte el pin con el módulo de interrupción por teclado, el de puertos de entrada/salida, y el TIM.

Cuando el IRQ está activado en el registro CONFIG2, las instrucciones BIH y BIL pueden ser utilizadas para leer el nivel lógico sobre el pin IRQ. Si la función IRQ está desactivada, estas instrucciones se comportan como si el pin IRQ tuviera un 1 lógico, independientemente del estado real del pin. Por el contrario, cuando el IRQ está activado, el bit 2 del registro de datos del puerto A siempre se leerá como un 0.

Cuando se utiliza el disparo de interrupción sensible al nivel, se evitan las falsas interrupciones por enmascaramiento del requerimiento de interrupción en la rutina de interrupción. Para desconectar el pull-up interno a VDD del pin IRQ se debe setear el bit IRQPUD en el registro CONFIG2 (\$001E).

## 11.7. Registros

El registro de control y estado IRQ (INTSCR) controla y monitorea las operaciones del módulo y tiene las siguientes funciones:

- Muestra el estado del flag IRQ
- Borra el latch IRQ
- Enmascara el requerimiento de interrupción IRQ
- Controla la sensibilidad de disparo del pin de interrupciones IRQ

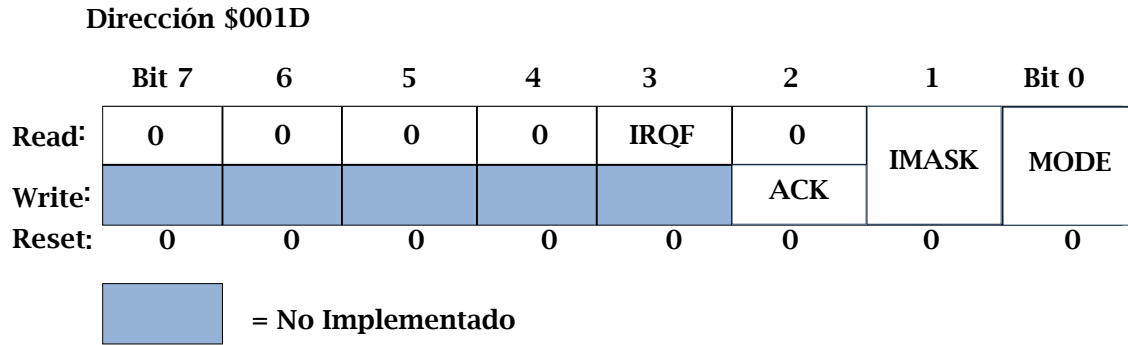


Figura 68. Registro de Control y Estado (INTSCR)

**Flag IRQF:** Este bit de estados de solo lectura es seteado cuando hay una interrupción IRQ pendiente.

- 1 = Interrupción pendiente IRQ
- 0 = No hay ninguna interrupción IRQ pendiente

**Bit de acuse de requerimiento de Interrupción ACK:** Escribiendo un 1 a este bit de solo escritura se borra el latch IRQ. ACK siempre se lee como 0.

**Bit de mascara de interrupción IMASK:** Escribiendo un 1 a este bit de lectura/escritura se deshabilita el requerimiento de interrupción IRQ.

- 1 = Requerimiento de interrupción IRQ deshabilitado
- 0 = Requerimiento de interrupción IRQ habilitado

**Bit de selección por nivel/flanco MODE:** Este bit de lectura/escritura controla la sensibilidad de disparo del pin IRQ.

- 1 = Requerimiento de Interrupción IRQ sobre flanco descendente y niveles bajos
- 0 = Requerimiento de Interrupción IRQ sobre flanco descendente únicamente

## 11.8. Ejemplo de aplicación

El siguiente ejemplo contiene la configuración indispensable para interactuar con la fuente más simple de interrupciones por IRQ en el microcontrolador HC908QY4.

### 11.8.1. Descripción funcional

Este programa realiza la conmutación del estado de encendido / apagado de un led conectado al puerto B, cada vez que se presiona diez veces consecutivas un pulsador conectado al pin PTA2/IRQ

Al ejecutarse el programa, el led conectado al puerto B se apaga, y el programa entra en un ciclo de conteo de las veces que ha sido presionado el pulsador, pero verifica el estado del botón con una variable definida en memoria RAM denominada “cuenta”.

Esta variable se utiliza en el bucle de consulta del contador y se actualiza en la rutina de atención a la interrupción en donde se chequea el estado del pulsador, ya que sólo se ejecuta dicha rutina cuando el botón ha sido presionado.

### 11.8.2. Descripción de hardware

La figura 69 muestra el HW necesario para que funcione la aplicación. Se observa un led conectado al pin PB0 mediante una resistencia que limita la corriente entregada por el microcontrolador a unos 11 mA ( $3V/270\text{ohm}$ ), de las especificaciones eléctricas se desprende que cada pin del puerto B puede entregar hasta 15mA, por lo que el valor de resistencia seleccionado asegura un consumo por debajo del límite especificado. En esta configuración de salida, el microcontrolador es quien entrega alimentación al led, existe otro modo de configuración en donde el led se alimenta externamente y el pin del MCU actúa drenando la corriente cuando se pone como entrada y en estado bajo, dicho modo se aplicará cuando no se desee consumir corriente del MCU. Siguiendo con el HW, se muestra un pulsador conectado al pin PTA2/IRQ mediante una resistencia de pull-up que le asegura un estado “alto” firme a la entrada IRQ, se debe aclarar que los microcontroladores de la familia HC908 contienen pull-ups internos configurables que facilitan el diseño del sistema y permiten reducir el HW necesario [17].

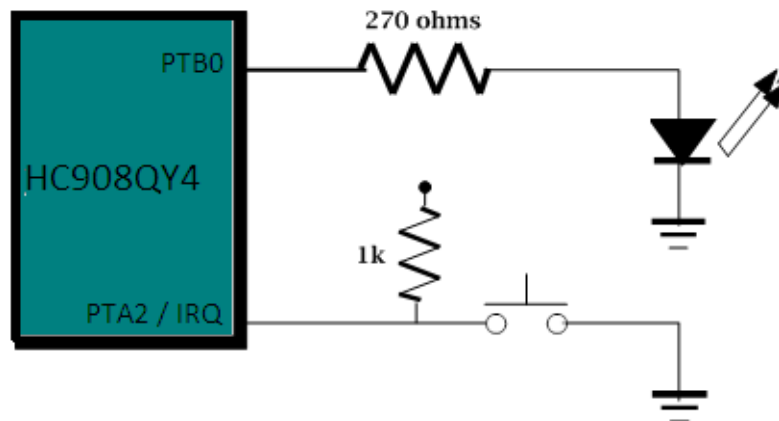


Figura 69. Circuito eléctrico del ejemplo

### 11.8.3. Descripción del código de la aplicación

Este programa conmuta el estado de un LED conectado a PB0 cada vez que un pulsador conectado a la línea IRQ sea presionado (y soltado) 10 veces consecutivas (el accionamiento del botón debe ser "rápido").

```

*****
;
;           ASIGNACIÓN DE REGISTROS           *
;
*****

```

;Se incluye el archivo con las definición de los registros, es ;importante que este archivo "qtqy\_registers.inc" se ubique en el mismo ;directorio que el archivo de código.

```

$include 'qtqy_registers.inc'
VECIRQ EQU $FFFC    ; Definición del vector IRQ
durac EQU $8000    ; Definición del valor a contar en el retardo de
                    ; software

```

```

*****
;inicio de RAM
    ORG $80
contador RMB 1

```

```

*****
; inicio de memoria de Programa
    ORG $F800
Inicio sei          ; deshabilita interrupciones
    Mov #$01, DDRB  ; configura el pin 0 del portb como salida
    clr contador   ; inicia contador
    mov #$40, CONFIG2 ; habilita la función IRQ
    mov #$01, CONFIG1 ; deshabilita el watchdog
    mov #$01, INTSCR ; habilita requerimiento de interrupción,
                    ; detección de interrupción por flanco y nivel
    cli            ; habilita interrupciones
lee   lda contador ; consulta contador
    cmp #$0A      ; se verifica si llegó a 10
    blt lee       ; si no llegó a 10, sigue contando
    lda Portb     ; si llegó invierte el estado del led
    eor #1
    sta Portb     ; invierte estado del LED
    bra inicio

```

```

*****
*           Rutina de atención a la interrupción           *
*           se ejecutara siempre que IRQ=0                 *
*
*****

```

```

rutina   bset 1,INTSCR
        jsr retardo ;pausa de 0.1 seg
        inc contador
        bset 2,INTSCR
        bclr 1,INTSCR
        rti

```

```

retardo   ldx durac
repite    decx

```

```
    bne repite
    rts
*****
;Vector de Reset de IRQ
    ORG VECIRQ
dw rutina    ;Salto a la rutina de atención a IRQ
*****
```

# CAPÍTULO 12

## Módulos de temporización

*Jorge R. Osio y Walter J. Aróztegui*

Esta sección describe los Módulos de Temporización, conocido como módulo de interfaz temporizador (TIM) en la familia de MCUs HC908 y como modulador por ancho de pulso/temporizador (TPM) en la familia HCS08, también se desarrollará el módulo RTC de la familia HCS08. El módulo de temporización es un temporizador de varios canales, que proporciona referencia de tiempo con **captura de entrada (input capture)**, **comparador de salida (output compare)**, y funciones de **modulación por ancho de pulso (PWM)**. La Figura 70 muestra un diagrama en bloques del TPM que posee el HCS08, y se describirá en este capítulo.

Este módulo, además de permitir la temporización para infinidad de aplicaciones, también permite controlar fuentes conmutadas, control de motores, control de elementos termoelectrónicos, choppers para sensores en ambientes ruidosos, generación de sonido, mediante la generación de PWM. Por otro lado, la captura de entrada permite leer pulsos de un encoder para determinar la velocidad y el sentido de giro de un motor de corriente continua entre otras aplicaciones.

### 12.1. Características

El TPM posee las siguientes características:

- 8 canales:
  - Cada canal puede ser input capture, output compare, o PWM
  - flanco ascendente, flanco descendente, o captura por disparo con flanco de entrada
  - Polaridad seleccionable de las salidas PWM
- El módulo puede ser configurado con buffer, modulación por ancho de pulso alineado en el centro (CPWM) sobre todos los canales
- Fuente de clock seleccionable del clock de bus con ple-escalador, clock del sistema fijo o pin de clock externo
  - Pre-escalador divisible por 1, 2, 4, 8, 16, 32, 64, o 128
  - Pin de clock externo, puede ser compartido por cualquier canal del temporizador o por un pin común mediante interrupciones.
- Operación de cuenta por incremento o decremento de 16 bits
- Una interrupción por canal

## 12.2. Modos de operación

Los canales del TPM se pueden configurar de manera independiente para funcionar en modo captura de entrada, comparación de salida o PWM alineado al borde. Un bit de control permite que todos los TPM (todos los canales) conmuten al modo PWM alineado al centro. Cuando se selecciona el modo PWM alineado al centro, los modos captura de entrada, comparación de salida y PWM alineado al borde no están disponibles sobre ninguno de los canales del módulo.

- **Modo *Captura de Entrada*:** Cuando ocurre un evento sobre el pin seleccionado del MCU, el valor actual del contador del timer de 16-bit se captura en el registro del valor del canal y se setea el bit del flag de interrupción.
- **Modo *Comparación de Salida*:** Cuando el valor en el registro contador del timer coincide con el valor del registro del canal, se setea el bit del flag de interrupción y se fuerza una acción sobre la salida seleccionada del MCU.
- **Modo *PWM alineado por flanco*:** El valor del registro de módulo de 16-bit setea el periodo de la señal de salida del PWM. El valor del registro del canal setea el ciclo de trabajo de la señal PWM. Este tipo de señal PWM es llamado alineado por flanco porque los flancos principales de todas las señales PWM están alineados con el comienzo del periodo, que es el mismo para todos los canales en un módulo TPM.
- **Modo *PWM alineado en el centro*:** El valor del registro de módulo de 16-bit setea el periodo de la señal de salida del PWM. El valor del registro del canal setea el ciclo de trabajo medio de la señal PWM. El contador del timer cuenta hasta que alcanza el valor del registro módulo y luego cuenta en decremento hasta llegar a cero. A medida que el valor del registro del canal coincide con la cuenta mientras se realiza en decremento, la salida PWM se mantiene activa. Cuando la cuenta coincide con el valor del registro del canal mientras el contador se incrementa, la salida PWM se mantiene inactiva. El nombre *alineado en el centro* se debe a que los centros de los periodos del ciclo de trabajo activo están alineados al valor de la cuenta cero. Este tipo de PWM se utiliza en motores utilizados en pequeños electrodomésticos.

## 12.3. Descripción funcional

La figura 70 muestra la estructura del TPM. El componente central es el contador de 16-bit que puede operar como un contador de ejecución libre o como un módulo contador con valor de cuenta preseleccionado. El contador del TPM provee el tiempo de referencia para las funciones de **captura de entradas** y de **comparación de salidas**. Los registros de Módulo, TPMxMODH : TPMxMODL, contienen el valor de referencia de cuenta, indicando mediante el mismo, hasta que valor deberá incrementarse el contador. Por software se puede leer el valor del contador en cualquier momento sin afectar la secuencia de conteo.

### 12.3.1. Pre-escalador del contador TPM

La fuente de clock del TPM es una de las siete salidas del pre-escalador o el pin de clock del TPM, CLKS. El pre-escalador genera siete tasas de clock desde el bus de clock interno. Los bits de selección de pre-escalador, PS[2:0], en el registro de control y estado del TPM (TPMxSC –registro de control y estados) permiten seleccionar la fuente de clock del módulo.

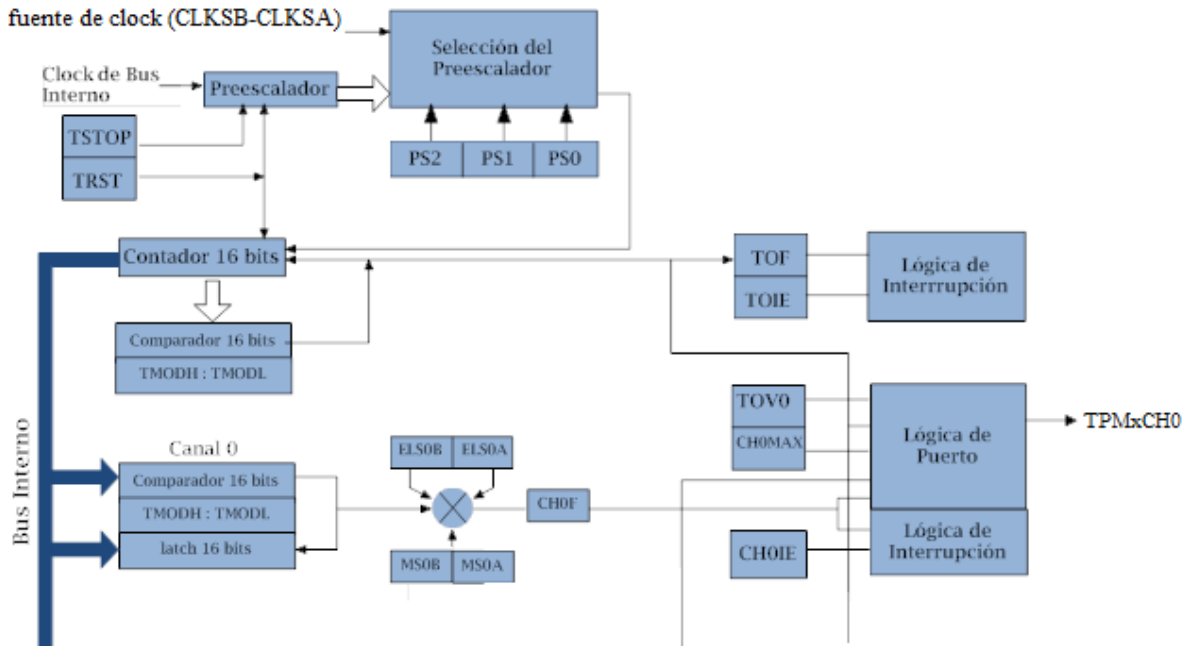


Figura 70. Diagrama en bloques del TPM

### 12.3.2. Captura de entrada

En este modo el MCU medirá eventos temporales externos, aplicados a uno de los pines de los canales del TPM. Estos eventos pueden ser tanto, la medida del ancho de un pulso o la frecuencia de una señal, la figura 71 muestra un esquema del funcionamiento de este modo.

Una aplicación posible puede ser la lectura de pulsos de un encoder incremental para medir la velocidad de giro de un motor de corriente continua, otra aplicación puede ser la medida del ancho de pulso de un paquete de datos enviado por otro dispositivo, para determinar a qué frecuencia está transmitiendo y así poder recibir los datos a la misma frecuencia.

Con la función de captura de entrada, el TPM puede capturar el momento en que ocurre un evento externo. Cuando ocurre un flanco activo sobre el pin de un canal de captura de entrada, el módulo de temporización almacena el contenido del contador en los registros de valor del canal, TPMxCnVH : TPMxCnVL. La polaridad del flanco activo es programable. La captura de entrada del TPM puede generar requerimientos de interrupción de la unidad central de procesos (CPU).



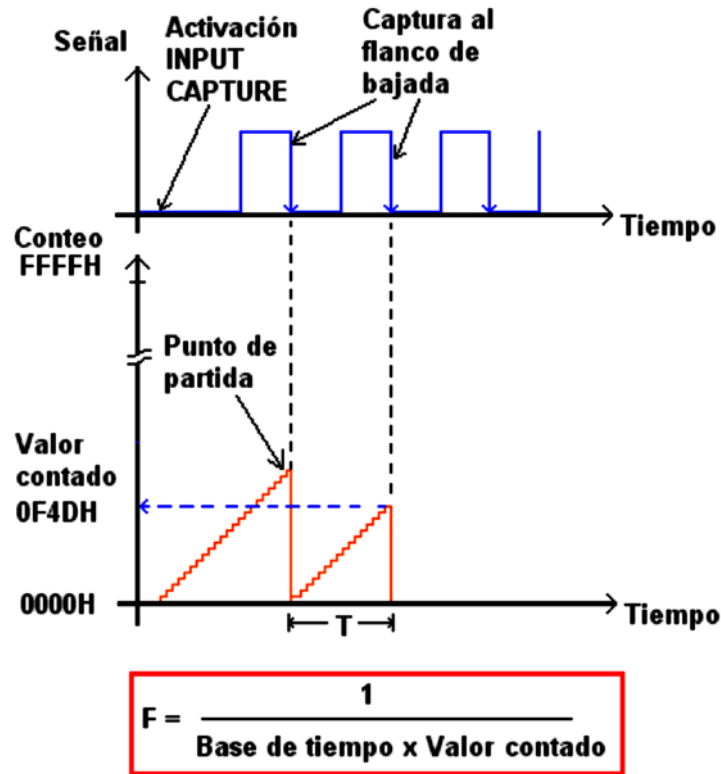


Figura 71. Esquema de funcionamiento de la captura de entrada

### 12.3.3. Comparación de salida

Con la función de comparación de salida, el TPM puede generar un pulso periódico con polaridad, duración y frecuencia programable. En la figura 72 se muestra un esquema del funcionamiento de este Modo, cuando el contador alcanza el valor en los registros de valor del canal comparador de salida, el TPM puede fijar, borrar, o cambiar el estado del pin del canal. El comparador de salida setea el bit de flag CHnF, que opcionalmente puede generar requerimientos de interrupción de la CPU.

En este modo el Módulo puede generar una señal cuadrada con un ciclo de trabajo del 50% (este es un modo particular del PWM).

En este modo los valores son transferidos a los correspondientes registros del canal temporizador solo después de que se hayan escrito ambas partes del registro de 16 bits y de acuerdo a los valores de los bits CLKSb:CLKSA, que pueden ser:

- Para CLKSb:CLKSA = 0:0, el registro se actualiza cuando se escriba el segundo byte.
- Para CLKSb:CLKSA distinto de 0:0, los registros se actualizan en el siguiente cambio del contador TPM, luego de que se escriba el segundo byte.

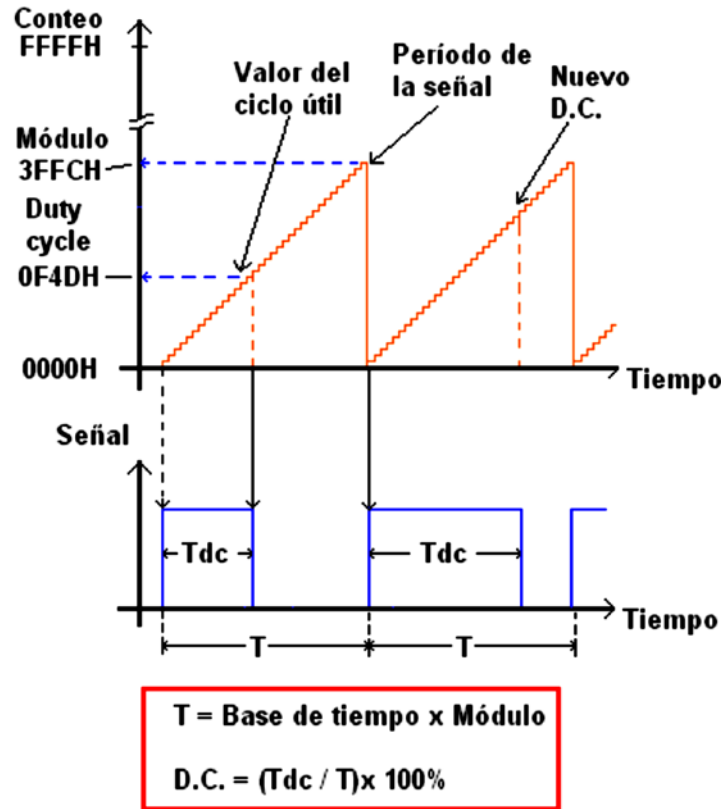


Figura 72. Esquema de Funcionamiento de Comparador de Salida / PWM

### 12.3.4. Modulación por Ancho de Pulso alineada por flanco (PWM)

En este modo el Módulo puede generar una señal cuadrada, aplicada a un pin del MCU. A la señal generada se le puede controlar el ciclo de trabajo (Duty Cycle), convirtiéndose en una señal PWM (Pulse Width Modulation).

El ciclo de trabajo de la señal (D.C. Duty Cycle) es programado como un porcentaje entre el 0 y el 100% del período de la señal. Generalmente este ciclo se controla desde un canal del módulo temporizador y deberá ser un valor menor que el período de la señal.

Usando la característica de cambio de estado por desborde con un canal comparador de salida, el TPM puede generar una señal PWM [15]. El valor en los registros TMODH:TMODL del módulo, determina el periodo de la señal PWM. El pin del canal cambia su estado cuando el contador alcanza el valor de los registros de módulo. El tiempo entre desbordes es el período de la señal PWM.

Como muestra la figura 73, el valor del "comparador de salida" en los registros del canal del TPM determina el ancho de pulso de la señal PWM. El tiempo entre el desborde del contador del temporizador y el desborde del registro comparador de salida es el ancho de pulso del PWM.

La polaridad de la señal PWM está determinada por la configuración en el bit de control ELSnA, Es posible configurar el ciclo de trabajo entre 0% y 100%. Si ELSnA = 0, el desborde del contador fuerza a la señal PWM a un "1 lógico", y la comparación de salida fuerza a la señal

PWM a un “nivel bajo”. Si  $ELSnA = 1$ , el desborde del contador fuerza a la señal PWM a un “nivel bajo”, y la comparación de salida fuerza a la señal PWM a un “1 lógico”.

La frecuencia de una señal PWM de 8-bit puede variar en un rango de 256, si se escribe \$00FF (255) en los registros del módulo contador (TPMxMODH:TPMxMODL), se produce un periodo de PWM de 256 veces el periodo de clock del bus interno, siempre y cuando se haya seteado el pre escalador con el valor 000. El valor en los registros del canal del TPM determina el ancho de pulso de la salida PWM. Si se escribe el valor \$0080 (128) en el registro del canal, se produce un ciclo de 128/256 o 50%.

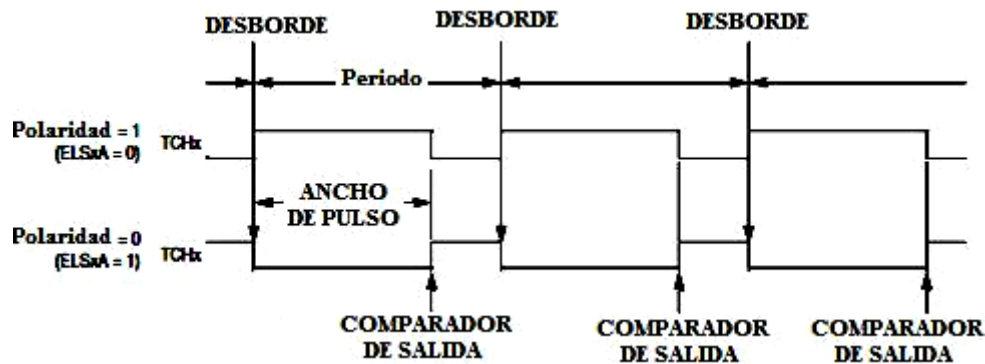


Figura 73. Periodo y ancho de pulso de PWM

### 12.3.5. Generación de señal PWM alineada en el centro

Este tipo de salida PWM usa el modo de conteo ascendente / descendente del contador del temporizador (CPWMS = 1). El valor de comparación de salida en TPMxCnVH:TPMxCnVL determina el ancho de pulso (ciclo de trabajo) de la señal PWM, mientras que el período está determinado por el valor en TPMxMODH:TPMxMODL. TPMxMODH: TPMxMODL puede adquirir valores en el rango de 0x0001 a 0x7FFF porque los valores fuera de este rango pueden producir resultados ambiguos. El bit ELSnA determinará la polaridad de la salida de CPWM (PWM alineado al centro).

- Ancho de pulso = 2 x (TPMxCnVH:TPMxCnVL)
- Periodo = 2 x (TPMxMODH:TPMxMODL);
- TPMxMODH:TPMxMODL=0x0001-0x7FFF

Si el registro de valor de canal (TPMxCnVH:TPMxCnVL) es cero o negativo (bit 15 configurado), el ciclo de trabajo será 0%. Si tiene un valor positivo (bit 15 borrado) y es mayor que el valor del registro de módulo, el ciclo de trabajo será del 100% porque la comparación del ciclo de trabajo nunca ocurrirá. Esto implica que el rango utilizable de períodos establecidos por el registro de módulo es de 0x0001 a 0x7FFE (0x7FFF si no necesita generar un ciclo de trabajo del 100%). Esta no es una limitación significativa, el período resultante sería mucho más largo que el requerido para las aplicaciones típicas.

TPMxMODH: TPMxMODL = 0x0000 es un caso especial que no se debe usar con el modo PWM alineado en el centro. Cuando CPWMS = 0, este caso corresponde al contador que corre libre de 0x0000 a 0xFFFF, pero cuando CPWMS = 1, el contador necesita una coincidencia válida con el registro de módulo en otro lugar que no sea en 0x0000 para poder cambiar las instrucciones de la cuenta ascendente a descendente.

El valor de comparación de salida en los registros de canal TPM determina el ancho de pulso (ciclo de trabajo) de la señal CPWM (Figura 74). Si ELSnA = 0, se produce una comparación mientras el contador ascendente pone a cero la señal de salida del CPWM y se produce una comparación, mientras el contador descendente pone a nivel alto la señal de salida. El contador cuenta regresivamente desde el valor asignado al registro de módulo en TPMxMODH: TPMxMODL hasta que llega a cero, luego cuenta progresivamente nuevamente hasta el valor asignado al módulo. Esto setea el período igual a dos veces el valor de TPMxMODH:TPMxMODL.

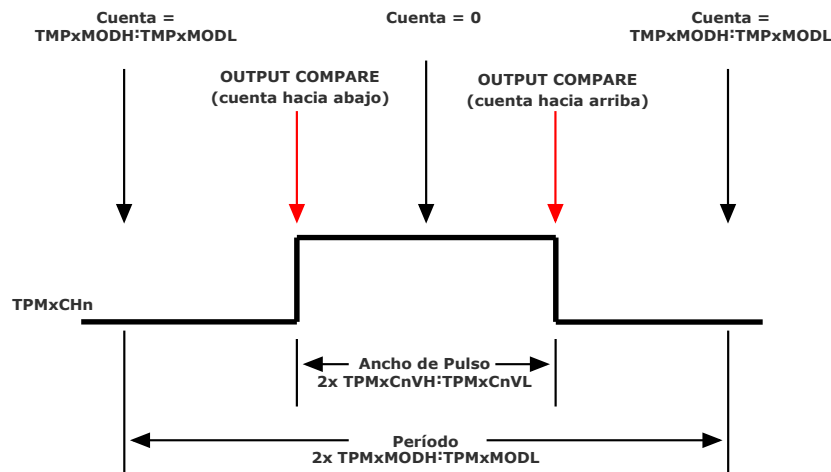


Figura 74. PWM alineado al centro

### 12.3.6. Inicialización del PWM

Para asegurar una correcta operación en la generación de señales PWM se debe seguir el siguiente procedimiento de inicialización:

1. En el Registro de control y estado:
  - a. Detener el contador seteando el bit de stop.
  - b. Resetear el contador y el pre-escalador seteando el bit de reset del TPM
2. En los registros de módulo, se debe escribir el valor para el periodo de PWM deseado.
3. En los registros del canal, se debe escribir el valor del ancho de pulso deseado.
4. En el registro de control y estado del canal:
  - a. Se debe seleccionar el modo de uso del TPM, en este caso PWM.
  - b. Se debe escribir 1 al bit de "conmutado sobre desborde".

- c. Se debe escribir 1:0 (polaridad 1 — para borrar la salida en la comparación) o 1:1 (polaridad 0 — para setear la salida en la comparación) para seleccionar el flanco/nivel en los bits, ELSxB:ELSxA.
5. En el registro de control y estado se debe borrar el bit de stop.

## 12.4. Interrupciones

Las siguientes Fuentes de TPM pueden generar requerimientos de interrupción:

- **Flag de Desborde del TPM (TOF)** — El bit TOF es seteado cuando el contador alcanza el valor programado en los registros del Módulo contador. El bit de habilitación de interrupción por desborde del TPM, TOIE, habilita el requerimiento de interrupción del CPU por desborde del contador. Los flags TOF y TOIE se encuentran en el Registro de control y estado del TPM.
- **Flags de canal del TPM (CHxF)** — El bit CHxF es seteado cuando ocurre una captura de entrada o una comparación de salida sobre el canal x. El requerimiento de Interrupción del CPU en el canal x del TPM es controlado por el bit de habilitación de interrupción del canal x, CHxIE. El requerimiento de interrupción del CPU en el canal x del TPM se habilita cuando CHxIE =1. CHxF y CHxIE se encuentran en el registro de control y estado del canal x del TPM.

## 12.5. Ejemplo de Aplicación

Con el Módulo TPM se pueden implementar infinidad de aplicaciones, ya sea para temporización, captura de entrada, comparación de salida, PWM, etc. A continuación se muestra un ejemplo simple de temporización usando el TPM y un ejemplo implementando PWM, el cual contiene utilidades del comparador de salida.

### 12.5.1. Uso del TPM con interrupciones

#### 12.5.1.1. Descripción funcional

Este ejemplo permite encender un led (PTB3) y generar una intermitencia de 2hz una vez que se oprime el pulsador (PTA1). Es un ejemplo muy sencillo en donde se utiliza el módulo de temporización para generar el retardo de 1/2 de segundo.

### 12.5.1.2. Descripción de hardware

Básicamente se requiere un circuito muy similar al de la aplicación que usa el IRQ, en este caso se configura el pull-up interno en PTB4 y se conecta el otro extremo del pulsador a tierra, por otro lado el led conectado a PTB3 se activa poniendo en alto la salida de dicho pin, como muestra la Figura 75.

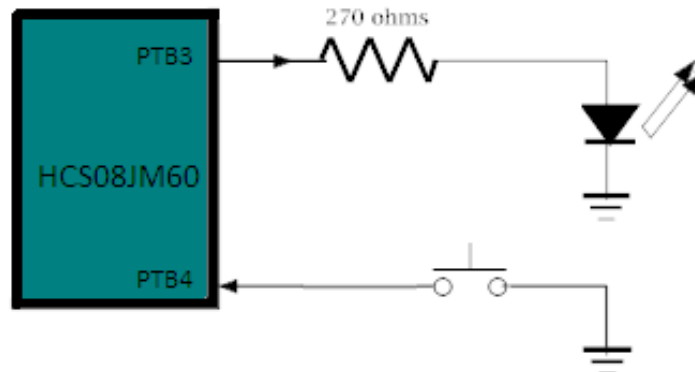


Figura 75. Configuración de HW del ejemplo de temporización

### 12.5.1.3. Descripción del software de la aplicación

El software tiene un bucle principal en donde se está continuamente leyendo el estado del pin 4 del puerto B, esperando que se presione el pulsador. Para la detección del pulsador se utiliza un anti rebote por software (un retardo), en donde se verifica dos veces que se presionó el pulsador para tomar el estado como válido. Una vez detectado el pulsador, se ejecuta el temporizador del TPM, para que en cada desborde de la cuenta (0,5 segundo) se cambie el estado del led.

El retardo de 0,5 seg se hace mediante el módulo TPM por interrupción mediante la configuración del registro de módulo para contar hasta 0x79E0 (31200), selección del clock del bus (CLKSB=0,CLKSA=1) y pre-escalador a 128 (PS2=1,PS1=1,PS0=1). Entonces,  $31200/8\text{mhz} \cdot 128 = 0,5 \text{ seg}$ .

A continuación se presenta el código de la aplicación:

```
#include <hidef.h> /* macro para habilitar interrupciones */
#include "derivative.h" /* declaración de periféricos */
#define pulsador PTBD_PTBD4
int i,j;
int contador=0;
void MCU_init(void); /* inicialización de los módulos */
void init_tpm1(void); //declaración de la función de configuración de tpm
void init_gpio(void);

interrupt 15 void isrVtpm1ovf(); //declaración función de interrupción tpm
```

```

void main(void) //programa principal
{
    MCU_init();    /* llamado a la inicialización del dispositivo */
    init_gpio();
    init_tpm1(void);
    for(;;)        //bucle infinito
    {
        if(pulsador==0)
        {for(i=0;i<15000;i++);        //retardo de anti rebote
        if(pulsador==0)
        TPM1SC_TOIE=!TPM1SC_TOIE ;    //resetea el valor del flag de interrupción TPM1 para indicar que
                                        //la misma fue atendida
        }
    }
} //fin bucle infinito
} //fin main

/*
** =====
** Inicialización el MCUs : MCU_init
** Descripción:
** Configuración de registros del CPU
** Parámetros: No
** Retorna: nada
** =====
*/

void MCU_init(void)
{
    /* Inicialización de registros del cpu que se escriben una vez */
    /* SOPT1: COPT=0,STOPE=0 */
    SOPT1 = 0x13;
    /* SPMSC1: LVWF=0,LVWACK=0,LVWIE=0,LVDRE=1,LVDSE=1,LVDE=1,BGBE=0 */
    SPMSC1 = 0x1C;
    /* SPMSC2: LVDV=0,LVWV=0,PPDF=0,PPDACK=0,PPDC=0 */
    SPMSC2 = 0x00;

    /* Inicialización del clock */
    if (*(unsigned char*)0xFFAF != 0xFF) { /* verifica si el valor del trimer está almacenado en la dirección especificada
    */
        MCGTRM = *(unsigned char*)0xFFAF; /* inicializa el registro MCGTRM */
        MCGSC = *(unsigned char*)0xFFAE; /* inicializa el registro MCGSC */
    }
    /* MCGC2: BDIV=1,RANGE=0,HGO=0,LP=0,EREFS=0,ERCLKEN=0,EREFSTEN=0 */
    MCGC2 = 0x40;          /* setea el registro MCGC2 */
    /* MCGC1: CLKS=0,RDIV=0,IREFS=1,IRCLKEN=0,IREFSTEN=0 */
    MCGC1 = 0x04;          /* setea el registro MCGC1 */
    /* MCGC3: LOLIE=0,PLLS=0,CME=0,VDIV=1 */
    MCGC3 = 0x01;          /* setea el registro MCGC3 */
    while(!MCGSC_LOCK) {    /* espera hasta que el FLL se estabilice */
    }

    /* Inicialización de los registros comunes del CPU */

```

```

/* PTASE: PTASE5=1,PTASE4=1,PTASE3=1,PTASE2=1,PTASE1=1,PTASE0=1 */
PTASE |= (unsigned char)0x3F;
/*PTBSE:PTBSE7=1,PTBSE6=1,PTBSE5=1,PTBSE4=1,PTBSE3=1,PTBSE2=1,PTBSE1=1,PTBSE0=1 */
PTBSE = 0xFF;
/* PTCSE:PTCSE6=1,PTCSE5=1,PTCSE4=1,PTCSE3=1,PTCSE2=1,PTCSE1=1,PTCSE0=1
*/
PTCSE |= (unsigned char)0x7F;
/*PTDSE:PTDSE7=1,PTDSE6=1,PTDSE5=1,PTDSE4=1,PTDSE3=1,PTDSE2=1,PTDSE1=1,PTDSE0=1 */
PTDSE = 0xFF;
/*PTESE:PTESE7=1,PTESE6=1,PTESE5=1,PTESE4=1,PTESE3=1,PTESE2=1,PTESE1=1,PTESE0=1 */
PTESE = 0xFF;
/*PTFSE:PTFSE7=1,PTFSE6=1,PTFSE5=1,PTFSE4=1,PTFSE3=1,PTFSE2=1,PTFSE1=1,PTFSE0=1 */
PTFSE = 0xFF;
/* PTGSE: PTGSE5=1,PTGSE4=1,PTGSE3=1,PTGSE2=1,PTGSE1=1,PTGSE0=1 */
PTGSE |= (unsigned char)0x3F;
/* PTADS: PTADS5=0,PTADS4=0,PTADS3=0,PTADS2=0,PTADS1=0,PTADS0=0 */
PTADS = 0x00;
/*PTBDS:PTBDS7=0,PTBDS6=0,PTBDS5=0,PTBDS4=0,PTBDS3=0,PTBDS2=0,PTBDS1=0,PTBDS0=0 */
PTBDS = 0x00;
/*PTCDS:PTCDS6=0,PTCDS5=0,PTCDS4=0,PTCDS3=0,PTCDS2=0,PTCDS1=0,PTCDS0=0*/
PTCDS = 0x00;
/*PTDDS:PTDDS7=0,PTDDS6=0,PTDDS5=0,PTDDS4=0,PTDDS3=0,PTDDS2=0,PTDDS1=0,PTDDS0=0 */
PTDDS = 0x00;
/*PTEDS:PTEDS7=0,PTEDS6=0,PTEDS5=0,PTEDS4=0,PTEDS3=0,PTEDS2=0,PTEDS1=0,PTEDS0=0 */
PTEDS = 0x00;
/*PTFDS:PTFDS7=0,PTFDS6=0,PTFDS5=0,PTFDS4=0,PTFDS3=0,PTFDS2=0,PTFDS1=0,PTFDS0=0*/
PTFDS = 0x00;
/* PTGDS: PTGDS5=0,PTGDS4=0,PTGDS3=0,PTGDS2=0,PTGDS1=0,PTGDS0=0 */
PTGDS = 0x00;
} /*fin MCU_init*/

```

*/\* Inicialización de puertos de entrada / salida\*/*

```
void init_gpio(void);
```

```
{
```

```
//Entrada del MCU
```

```
PTBDD_PTBD4=0; //pin 4 del Puerto B al pulsador
```

```
//pull-up interno
```

```
PTBPE_PTBE4=1;
```

```
// Salidas del MCU
```

```
PTBDD_PTBD3=1; //pin 3 del puerto B al led
```

*/\* ### Código de inicialización de TPM1 \*/*

```
void init_tpm1(void)
```

```
{
```

```
/* TPM1SC: TOF=0,TOIE=0,CPWMS=0,CLKSB=0,CLKSA=0,PS2=0,PS1=0,PS0=0 */
```

```
TPM1SC = 0x00; /* detiene y resetea el contador */
```

```
TPM1MOD = 0x79E0U; /* seteo del valor del período */
```

```
(void)(TPM1SC == 0); /* Overflow y borrado de flag (primera parte) */
```

```
/* TPM1SC: TOF=0,TOIE=0,CPWMS=0,CLKSB=0,CLKSA=1,PS2=1,PS1=1,PS0=1 */
```

```
TPM1SC = 0x0F; /* borrado del flag (2da parte) y seteo del registro  
de control del temporizador */
```



```

asm CLI;          /* habilita interrupciones */
}

/*
** =====
** Manejo de interrupción: isrVtpm1ovf
**
** Descripción:
** Rutina de servicio de interrupción del TPM1.
** Parámetros: no
** Retorna : nada
** =====
*/
interrupt 15 void isrVtpm1ovf(void)
{
  /* Código que se ejecuta durante la interrupción */
  PTBD_PTBD3=!PTBD_PTBD3; //cambia el estado del led
  TPM1SC_TOF=0; //resetea el tof
}
/* end of isrVtpm1ovf */

```

## 12.5.2. Generación de señal PWM

### 12.5.2.1. Descripción funcional

Esta aplicación permite realizar el control de un motor de corriente continua mediante una señal PWM que varía la velocidad del motor al cambiar el ancho de pulso y una señal que varía el sentido de giro, mediante el estado de un pin del MCU. Para modificar el ancho de pulso se utiliza un potenciómetro conectado a un canal del Conversor AD. Entonces, cuando el valor de la conversión está entre 0x00 y 0x7F el motor gira en un sentido y cuando está entre 0x81 y 0xff gira en el sentido contrario. La frecuencia de la señal PWM debe tener un periodo de 20kHz aprox. y el ciclo de trabajo, (que representa el ancho de pulso), se corresponde con el valor del conversor.

Para la interfaz entre el motor y el MCU se utiliza un puente H como se muestra en la figura 76-a, en donde la Tensión V1 representa la señal PWM que ingresa al puente H y la tensión V2 representa la señal que modifica el sentido de giro (generada por un pin de entrada salida).

### 12.5.2.2. Descripción de hardware

El Hardware necesario para la aplicación es un potenciómetro conectado a un canal del conversor mediante el pin 0 del puerto D (AD8), el cual permite modificar la velocidad asignando el valor leído a la parte alta del registro del canal TPM (este valor fija el ancho de pulso). Adicionalmente, se requiere un puente H como el que se muestra en la figura 76-a que se alimenta con 5V y tierra por los extremos superior e inferior para el funcionamiento del

motor. También recibe la entrada PWM por el extremo izquierdo V1, la que modifica la velocidad dependiendo del ancho de pulso y la señal que determina el sentido de giro por el extremo derecho V2 [17].

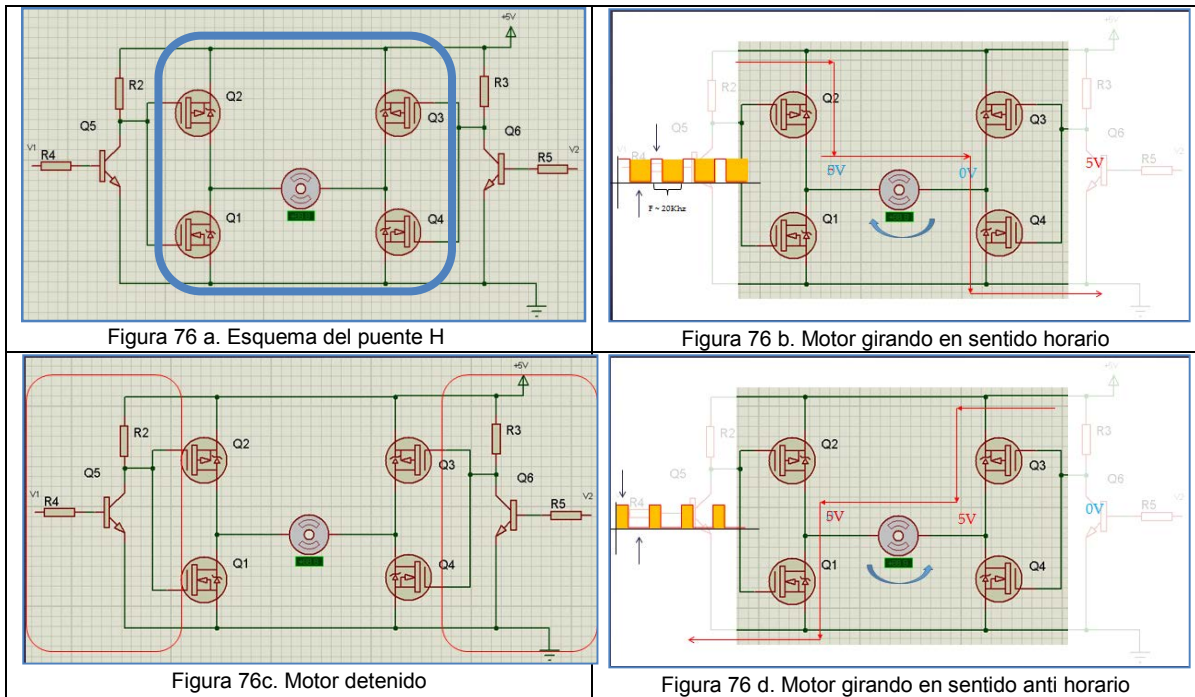


Figura 76. Esquema circuital del puente H para control de motor de CC.

### 12.5.2.3. Descripción del software de la aplicación

A continuación se describen las características que deberá tener el software de la aplicación, pero solo se detallará el código que permite generar la señal PWM, el resto de la aplicación queda como tarea para el lector. El software básicamente deberá leer el valor del canal del conversor que se encuentra en el pin 0 del puerto D y deberá sobre escribir la parte alta del registro del canal PWM. Esto significa que el valor leído del ADC se usará para fijar el ancho de pulso PWM. Las salida del PWM se realizará por el pin 1 del puerto F que se envía a una de las entradas del puente H, para controlar la velocidad del motor. La salida del pin 4 del puerto B se usará para controlar el sentido de giro del motor y también se conecta al puente H.

A continuación se detallan los parámetros del funcionamiento del sistema:

- ADR  $\rightarrow$  0 V = 0x00  
5 V = 0xFF
- Frec. PWM  $\rightarrow$  20 khz = 1 ciclo / 50 useg. (frecuencia compatible con la inercia del motor)
- con Tmod = 0x00FF habrá un desborde cada 79.7 useg para una frecuencia de bus de 3,2Mhz aprox.

El funcionamiento en sentido horario se produce poniendo la línea de sentido de giro en alto (pin 4 de puerto B), en este caso, cuando el ciclo de trabajo sea mayor, menor será la velocidad del motor. Este sentido de giro se produce cuando el valor de la conversión es negativo, (MSB en alto, entre \$80 y FF), y se toman los 7 bits menos significativos de la conversión como valor de ciclo de trabajo PWM, para esto se guardan en la parte alta del registro TCH1, es decir TCH1H. Analizando la figura 76-b se pueden observar el sentido de circulación de corriente sobre el puente H cuando el motor gira en este sentido.

Para que el motor gire en sentido anti horario el valor de la conversión debe ser positivo (entre 0 y 0x7F), por lo que, en este caso se pone la línea de sentido de giro a nivel bajo, lo que producirá mayor velocidad de giro del motor cuando mayor sea el ciclo de trabajo, tal cual lo ilustra la figura 76-d. Como en el caso anterior, el valor de la conversión se utiliza para determinar el ciclo de trabajo y por ende la velocidad de giro del motor.

A continuación se muestra el código que permite generar una señal PWM con diferentes ciclos de trabajo en función de la combinación de pulsadores conectados a los pines 5,6 y 7 del puerto B:

```
#include <hidef.h> /* macro para habilitar interrupciones */
#include "derivative.h" /* declaración de periféricos */
#define TPM_INPUT_CLK 15000 //1.5MHz
void Mcu_Init(void);
void Init_GPIO(void)
void Get_Button_Status(char *State);
void TPM_Init(void);
char TPM_Config(int Freq, char Duty);
void TPM_Stop(void);
void TPM_Start(void);
void main(void)
{
    Mcu_Init();
    Init_GPIO();
    TPM_Init();
    EnableInterrupts;          /* habilita interrupciones */
    char pulsador=0xff;

    for(;;) //bucle infinito
    {
        Get_Button_Status(&pulsador)
        __RESET_WATCHDOG();    /* resetea el watchdog */

        switch(pulsador)
        {
            case 0:             /*inicia PWM*/
                TPM_Config(int 20, 10)
                TPM_Start();
                break;
            case 1:             /*inicia PWM*/
                TPM_Config(int 20, 20)
```

```

        TPM_Start();
        break;
    case 2:          /*inicia PWM*/
        TPM_Config(int 20, 30)
        TPM_Start();
        break;
    case 3:          /*iniciaPWM*/
        TPM_Config(int 20, 40)
        TPM_Start();
        break;
    case 4:          /*inicia PWM*/
        TPM_Config(int 20, 50)
        TPM_Start();
        break;
    case 5:          /*inicia PWM*/
        TPM_Config(int 20, 60)
        TPM_Start();
        break;
    case 6:          /*inicia PWM*/
        TPM_Config(int 20, 70)
        TPM_Start();
        break;
    case 7:          /*inicia PWM*/
        TPM_Stop();
        break;
    default:
        break;
    }
}
}
//inicializa el módulo TPM
void TPM_Init(void)
{
    TPM1SC = 0x00;
    TPM2SC = 0x00;
    return;
}

/*****
* Funcion   MCU_Init
* resumen:  inicializa el MCU.
*           Deshabilita STOP y COP
*           setea la frecuencia del clock del bus
*****/
void Mcu_Init()
{
    SOPT1 = 0xF3;          /*habilita el COP, y habilita el modo stop del MCU */
    /* 0b00000010 bit5: STOP Mode enable; bit6: COP configuration; bit7: COP configuration */
    SOPT2 = 0x00;
    SPMSC1 = 0x41;
    /*0b01000001 ; bit0: BGBE Bandgap Buffer enable; bit6: LWWACK Low-Voltage

```

```

Warning Acknowledge*/
SPMSC2 = 0x00;
    // MCG clock initialization, fBus=24MHz
MCGC2 = 0x36;
while(!(MCGSC & 0x02));    //espera a que el OSC se estabilice
MCGC1 = 0x1B;
MCGC3 = 0x48;
while ((MCGSC & 0x48) != 0x48); //espera a que el PLL se bloquee
}

//inicializa puertos de entrada salida
void Init_GPIO(void)
{
PTBD = 0x00;
PTBDD = 0x1F;    /*PTB5-PTB7 : pulsadores*/
PTBDS = 0x00;
PTBPE = 0xE0;
return;
}

//devuelve el estado de los pulsadores
void Get_Button_Status(char *State)
{ char temp;
temp= (~ PTBD ) & 0xE0;
*State = temp>> 5;
return;
}

/*****
* Parámetros:   Freq: (cientos de Hz) Range: 1-1000
*               Duty: 0-100%
* Frecuencia = Freq * 100 (100Hz-100KHz)
* retorna:      0: falla config; 1: config exitosa
*****/
char TPM_Config (int Freq, char Duty)
{
unsigned int Mod = TPM_INPUT_CLK;
unsigned long Cnt = 0x00;
if(Freq > 1000)
return 0;
else
Mod /= Freq;
TPM2MOD = Mod;
TPM2C0SC = 0x24;
Cnt = (unsigned long)(Mod * Duty);
Cnt /= 100;
TPM2C0V = (int)Cnt;
return 1;
}

//inicia el tpm 1

```

```

void TPM_Start(void)
{
    TPM2CNT = 0;
    TPM2SC |= 0x0C;    //bus de clock, dividido 8

    return;
}
//detiene el TPM 1
void TPM_Stop(void)
{ char Dummy;
    Dummy = TPM2SC;
    TPM2SC &= 0x67;    //bus de clock, dividido 8
    return;
}
    
```

### 12.6. Contador de tiempo real (RTC)

El contador de tiempo real (RTC) consta de un contador de 8 bits, un comparador de 8 bits, varios bits para realizar un pre escalado (división de frecuencia), dos fuentes de reloj y una interrupción periódica programable como se muestra en el diagrama de la figura 77. Este módulo se puede usar para indicar la hora, el calendario o cualquier función del planificador de tareas. También puede servir como un activador cíclico, si se configura el MCU en el modo de bajo consumo.

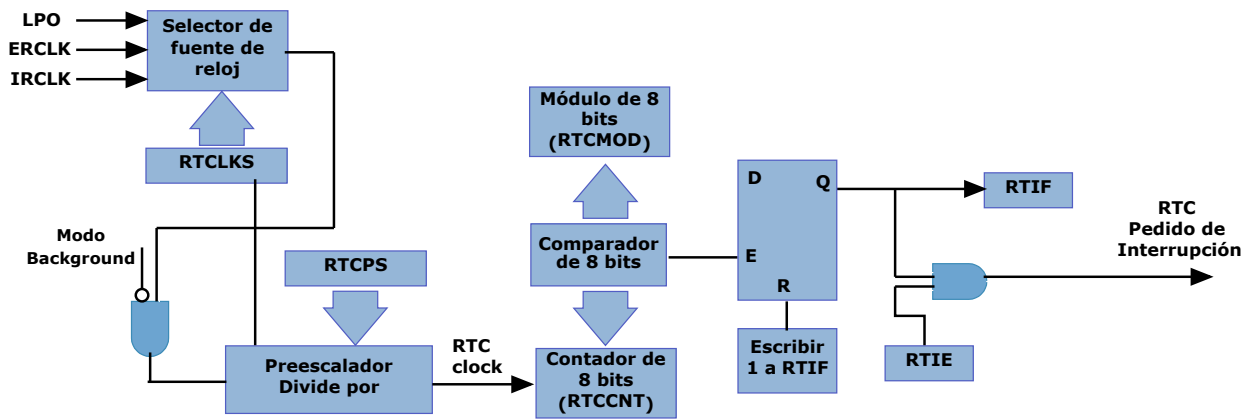


Figura 77. Diagrama en bloques RTC

Después de cualquier reinicio de la MCU, el contador se detiene y se restablece a 0x00, el registro de módulo se configura a 0x00 y el pre-escalador está en off (0000). El reloj del oscilador interno de 1 kHz se selecciona como la fuente de reloj predeterminada. Para iniciar el pre-escalador, se debe escribir un valor distinto de cero en los bits de selección del pre-escalador (RTCPS). En la tabla 12 se indican las bases de tiempo definidas para los distintos valores que se pueden asignar al RTCPS y las distintas fuentes de clock.

Tres fuentes de reloj son seleccionables por software: el reloj del oscilador de baja potencia (LPO), el reloj externo (ERCLK) y el reloj interno (IRCLK). Los bits de selección de reloj RTC (RTCLKS) seleccionan la fuente de reloj deseada. Si se escribe un valor diferente en RTCLKS, el pre-escalador y los contadores RTCCNT se restablecen a 0x00.

**Tabla 12. Pre-escalador del RTC**

RTCPS	Clock interno de 1khz (rtclks=00)	Clock externo de 1 mhz (rtclks=01)	Clock interno de 32khz (rtclks=10)	Clock externo de 32khz (rtclks=11)
0000	Off	off	off	Off
0001	8ms	1.024ms	250us	32ms
0010	32ms	2.048ms	1ms	64ms
0011	64ms	4.096ms	2ms	128ms
0100	128ms	8.192ms	4ms	256ms
0101	256ms	16.4ms	8ms	512ms
0110	512ms	32.8ms	16ms	1.024s
0111	1.024 s	65.5ms	32ms	2.048s
1000	1ms	1ms	31.25us	31.25ms
1001	2ms	2ms	62.5us	62.5ms
1010	4ms	5ms	125us	156.25ms
1011	10ms	10ms	312.5us	312.5ms
1100	16ms	20ms	0,5ms	0.625s
1101	0.1s	50ms	3.125ms	1.5625s
1110	0.5s	0.1s	15.625 ms	3.125s
1111	1s	0.2s	31.25ms	6.25s

### 12.6.1. Configuración del módulo de temporización con RTC

Se puede implementar la aplicación realizada en la sección 12.5.1, de forma rápida usando el RTC mediante el reemplazo de las funciones del TPM1 por las siguientes funciones.

```
//la siguiente sentencia se debe ubicar en la parte del programa donde se desee iniciar la cuenta //del RTC
RTCSC_RTIE=1;
//Función de inicialización del RTC para generar un retardo de 124 x 2ms = 248ms o 0,25 s
```

```
Void rtc_init(void)
{
/* RTCMOD: RTCMOD=0x7C */
```

```

RTCSC = 0x7C;          /* seteo del registro módulo con el valor de cuenta 0x7C o 124 en decimal */

/* RTCSC: RTIF=1,RTCLKS=00,RTIE=0,RTCPS=1001 (2ms) */
RTCSC = 0x89;          /* Configuración del RTC */
}

/*
** =====
** Manejo de interrupción: isrVrtc
**
** Descripción:
** Rutina de servicio de interrupción del RTC.
** Parámetros: no
** Retorna : nada
** =====
interrupt 29 void isrVrtc(void)
{ RTCSC_RTIE=0; //deshabilito las interrupciones
  /* código que se ejecuta durante la interrupción */
  PTBD_PTBD3=!PTBD_PTBD3; //cambia el estado del led
  RTCSC_RTIF=1; //seteo el flag para indicar que la interrupción fue atendida
  RTCSC_RTIE=1; //habilito las interrupciones de RTC
}

```



# CAPÍTULO 13

## Módulo de interrupción por teclado (KBI)

*Jorge R. Osio y Walter J. Aróztegui*

El módulo de interrupción por teclado permite disponer de varias interrupciones externas independientes.

### 13.1. Características

Este módulo incluye:

- Pines de interrupción por teclado con bits de habilitación separados y uno de enmascaramiento de interrupción.
- Dispositivo de pull-up configurable por software en el caso de que el pin del puerto sea configurado como entrada.
- Sensibilidad de la interrupción programable a flanco solo o flanco y nivel.
- Salida desde modos de bajo consumo.

### 13.2. Descripción funcional

El módulo de interrupción por teclado controla la habilitación y deshabilitación de las funciones de interrupción sobre varios pines de los puertos de forma independiente uno del otro.

#### 13.2.1. Operación de teclado

Escribiendo los bits KBIE0-KBIE7 en el registro de habilitación de interrupción por teclado, se habilita o deshabilita, independientemente, cada pin del puerto como un pin de interrupción por teclado, esto activa automáticamente el respectivo dispositivo interno de pull-up desde el bit del registro de habilitación de entradas del puerto.

Una interrupción por teclado se produce cuando una o más de éstas entradas se pone en estado bajo "0 lógico" después de estar todas en estado alto (1 lógico).

El bit KBMOD del registro de control y estado de teclado, controla el modo de disparo de la interrupción por teclado:

- Si la interrupción por teclado es solo sensible a flanco, un flanco de bajada sobre el pin no almacena un pedido de interrupción si otro pin ya está en nivel bajo. Para prevenir la pérdida de un pedido de interrupción, el software puede deshabilitar la última entrada mientras está en nivel bajo.
- Si la interrupción por teclado es sensible al flanco descendente y nivel bajo, se producirá un requerimiento de interrupción cuando cualquier entrada de interrupción esté en nivel bajo.

Si el bit KBMOD está seteado, las entradas de interrupción por teclado son sensibles al “flanco descendente” y “nivel bajo” y para borrar un pedido de interrupción se deben cumplir las siguientes acciones:

- Borrado por software: Por software se puede generar esta señal de reconocimiento, escribiendo 1 en el bit de KBACK en el registro de control y estado de teclado (KBISC); tal bit es útil en aplicaciones que chequean continuamente las entradas de interrupción por teclado (poll) y necesitan que el software borre los pedidos. Escribiendo el bit KBACK antes de dejar la rutina del servicio de interrupción se pueden prevenir las interrupciones espurias debidas al ruido. Setear este bit no afecta las subsecuentes transiciones sobre las entradas de interrupción por teclado, un flanco de bajada que ocurre después de escribir este bit almacena otro pedido de interrupción.
- Retorno de todas las entradas de interrupción por teclado a 1 lógico (mientras cualquiera de los pines de entrada de interrupción por teclado esté en 0, la interrupción permanece seteada).

Si el bit KBMOD está borrado, el pin de interrupción por teclado es sensible sólo a flanco descendente. Con este bit en cero, el borrado por software limpia inmediatamente los pedidos de interrupción.

El bit de flag de teclado KBF en el registro de control y estado de teclado puede ser usado para ver si queda pendiente un pedido de interrupción. Este bit no se ve afectado por el bit de enmascaramiento de interrupción (flag I del CCR), lo que lo hace útil en aplicaciones donde se prefiere la encuesta (polling).

Para determinar el nivel lógico sobre un pin de interrupción por teclado, se debe leer el registro de datos correspondiente a ese pin.

En este caso, configurando un pin para interrupción por teclado se fuerza a dicho pin a ser una entrada, sin necesidad de configurar el “registro de dirección de datos DDR.

### 13.2.2. Inicialización de teclado

Cuando un pin de interrupción por teclado se habilita, lleva un tiempo para que el pull-up interno se ponga en 1 lógico, tiempo durante el cual puede ocurrir una falsa interrupción.

Para prevenir esta falsa interrupción se debe inicializar con:

1. Enmascarar la interrupción por teclado seteando el bit KBIE en el registro de control y estado del módulo KBI.
2. Habilitar los pines KBI seteando los bits de KBIPEX apropiados en el registro de habilitación de interrupción por teclado KBIPE.
3. Escribir el bit KBACK en el registro de control y estado del KBI para limpiar cualquier falsa interrupción.
4. Borrar el flag KBIE.

### 13.3. Ejemplo de aplicación

#### 13.3.1. Ejemplo en lenguaje C

Como en los ejemplos anteriores, se usará la sentencia interrupt específica de CodeWarrior que permite simplificar el proceso de definición de una interrupción mediante el número que especifica la entrada en la tabla de vectores de interrupción. Este ejemplo enciende un led diferente en función de la tecla de KBI presionada. Se utilizaron los pines de 0 a 3 del puerto B para los leds, luego para las interrupciones por teclado se configuraron los pines 4,5 del puerto B y 2,3 del puerto G.

```
#include <hides.h> /* macro para habilitar interrupciones */
#include "derivative.h" /* declaración de periféricos */
//declaración de funciones
Void MCU_init(void);
Void gpio_init(void);
Void kbi_init(void);
interrupt 25 void isrVkeyboard(void);

void main (void)
{
MCU_init();
gpio_init();
kbi_init();

for (;;) { //bucle infinito

} //fin bucle infinito
} //fin main
```

```

/*
** =====
**  Inicialización del CPU : MCU_init
**
**  Descripción:
**  Configuración de registros del CPU
**  Parámetros : No
**  Retorna   : nada
** =====
*/
void MCU_init(void)
{
    /* Inicialización de registros del cpu que se escriben una vez */
    /* SOPT1: COPT=0,STOPE=0 */
    SOPT1 = 0x13;
    /* SPMSC1: LVWF=0,LVWACK=0,LVWIE=0,LVDRE=1,LVDSE=1,LVDE=1,BGBE=0 */
    SPMSC1 = 0x1C;
    /* SPMSC2: LVDV=0,LVWV=0,PPDF=0,PPDACK=0,PPDC=0 */
    SPMSC2 = 0x00;

    /* Inicialización del clock */
    if (*(unsigned char*)0xFFAF != 0xFF) { /* verifica si el valor del trimer está almacenado en la dirección especificada */
    /*
        MCGTRM = *(unsigned char*)0xFFAF; /* inicializa el registro MCGTRM */
        MCGSC = *(unsigned char*)0xFFAE; /* inicializa el registro MCGSC */
    }
    /* MCGC2: BDIV=1,RANGE=0,HGO=0,LP=0,EREFS=0,ERCLKEN=0,EREFSTEN=0 */
    MCGC2 = 0x40;          /* setea el registro MCGC2 */
    /* MCGC1: CLKS=0,RDIV=0,IREFS=1,IRCLKEN=0,IREFSTEN=0 */
    MCGC1 = 0x04;          /* setea el registro MCGC1 */
    /* MCGC3: LOLIE=0,PLLS=0,CME=0,VDIV=1 */
    MCGC3 = 0x01;          /* setea el registro MCGC3 */
    while(!MCGSC_LOCK) { /* espera hasta que el FLL se estabilice */
    }

    /* Inicialización de los registros comunes del CPU */
    /* PTASE: PTASE5=1,PTASE4=1,PTASE3=1,PTASE2=1,PTASE1=1,PTASE0=1 */
    PTASE |= (unsigned char)0x3F;
    /*PTBSE:PTBSE7=1,PTBSE6=1,PTBSE5=1,PTBSE4=1,PTBSE3=1,PTBSE2=1,PTBSE1=1,PTBSE0=1 */
    PTBSE = 0xFF;
    /* PTCSE:PTCSE6=1,PTCSE5=1,PTCSE4=1,PTCSE3=1,PTCSE2=1,PTCSE1=1,PTCSE0=1 */
    /*
    PTCSE |= (unsigned char)0x7F;
    /*PTDSE:PTDSE7=1,PTDSE6=1,PTDSE5=1,PTDSE4=1,PTDSE3=1,PTDSE2=1,PTDSE1=1,PTDSE0=1 */
    PTDSE = 0xFF;
    /*PTESE:PTESE7=1,PTESE6=1,PTESE5=1,PTESE4=1,PTESE3=1,PTESE2=1,PTESE1=1,PTESE0=1 */
    PTESE = 0xFF;
    /*PTFSE:PTFSE7=1,PTFSE6=1,PTFSE5=1,PTFSE4=1,PTFSE3=1,PTFSE2=1,PTFSE1=1,PTFSE0=1 */
    PTFSE = 0xFF;
    /* PTGSE: PTGSE5=1,PTGSE4=1,PTGSE3=1,PTGSE2=1,PTGSE1=1,PTGSE0=1 */
    PTGSE |= (unsigned char)0x3F;

```

```

/* PTADS: PTADS5=0,PTADS4=0,PTADS3=0,PTADS2=0,PTADS1=0,PTADS0=0 */
PTADS = 0x00;
/*PTBDS:PTBDS7=0,PTBDS6=0,PTBDS5=0,PTBDS4=0,PTBDS3=0,PTBDS2=0,PTBDS1=0,PTBDS0=0 */
PTBDS = 0x00;
/*PTCDS:PTCDS6=0,PTCDS5=0,PTCDS4=0,PTCDS3=0,PTCDS2=0,PTCDS1=0,PTCDS0=0*/
PTCDS = 0x00;
/*PTDDS:PTDDS7=0,PTDDS6=0,PTDDS5=0,PTDDS4=0,PTDDS3=0,PTDDS2=0,PTDDS1=0,PTDDS0=0 */
PTDDS = 0x00;
/*PTEDS:PTEDS7=0,PTEDS6=0,PTEDS5=0,PTEDS4=0,PTEDS3=0,PTEDS2=0,PTEDS1=0,PTEDS0=0 */
PTEDS = 0x00;
/*PTFDS:PTFDS7=0,PTFDS6=0,PTFDS5=0,PTFDS4=0,PTFDS3=0,PTFDS2=0,PTFDS1=0,PTFDS0=0*/
PTFDS = 0x00;
/* PTGDS: PTGDS5=0,PTGDS4=0,PTGDS3=0,PTGDS2=0,PTGDS1=0,PTGDS0=0 */
PTGDS = 0x00;
} /*fin MCU_init*/

//función de configuración de leds como salida
void gpio_init(void)
{
  /* ### configuración de puertos de entrada salida GPIO */
  /* PTBD: PTBD0=0 */
  PTBD &= (unsigned char)~0x01;
  /* se configuran los 4 leds como salida
PTBDD: PTBDD3=1,PTBDD2=1,PTBDD1=1,PTBDD0=1 */
  PTBDD |= (unsigned char)0x0F;
}

//función de configuración kbi
Void kbi_init(void)
{
  /* KBISC: KBIE=0 */
  KBISC &= (unsigned char)~0x02;
  /* KBIES: KBEDG7=0,KBEDG6=0,KBEDG5=0,KBEDG4=0,KBEDG3=0,KBEDG2=0,KBEDG1=0,KBEDG0=0 */
  KBIES = 0x00;
  /* KBISC: KBMOD=0 */
  KBISC &= (unsigned char)~0x01;
  /* se configuran los 4 bits más significativos del kbi
KBIPE:KBIPE7=1,KBIPE6=1,KBIPE5=1,KBIPE4=1,KBIPE3=0,KBIPE2=0,KBIPE1=0, KBIPE0=0 */
  KBIPE = 0xF0;
  /* KBISC: KBACK=1 */
  KBISC |= (unsigned char)0x04;
  /* KBISC: KBIE=1 */
  KBISC |= (unsigned char)0x02;
}

//función de interrupción kbi
interrupt 25 void isrVkeyboard(void)
{
  KBISC_KBIE=0; //se Deshabilitan las interrupciones para evitar el
                //rebote del pulsador
  if(PTBD_PTBD4==0)
  {

```

```
    PTBD_PTBD0=1;
    PTBD_PTBD1=0;
    PTBD_PTBD2=0;
    PTBD_PTBD3=0;
}
else if(PTBD_PTBD5==0)
{
    PTBD_PTBD0=0;
    PTBD_PTBD1=1;
    PTBD_PTBD2=0;
    PTBD_PTBD3=0;
}
else if(PTGD_PTGD2==0)
{
    PTBD_PTBD0=0;
    PTBD_PTBD1=0;
    PTBD_PTBD2=1;
    PTBD_PTBD3=0;
}
else if(PTGD_PTGD3==0)
{
    PTBD_PTBD0=0;
    PTBD_PTBD1=0;
    PTBD_PTBD2=0;
    PTBD_PTBD3=1;
}
KBISC_KBIE=1;    //se habilitan las interrupciones
KBISC_KBACK=1;  //sale de la interrupción
}
/* fin de isrVkeyboard */
```

# CAPÍTULO 14

## Módulo conversor analógico/digital

*Jorge R. Osio y Walter J. Aróztegui*

### 14.1. Introducción

Antes de la descripción del módulo ADC del microcontrolador se analizarán las características y la utilidad de los conversores AD.

Como se describió anteriormente, los dispositivos microcontroladores son puramente digitales, lo que no quita que puedan procesar señales analógicas. Para el procesamiento de señales analógicas en los MCUs se utilizan los conversores AD, lo cuales digitalizan las señales analógicas para su posterior procesamiento dentro del microprocesador.

El proceso de convertir introduce una inevitable pérdida de información. Esta pérdida es inherente al proceso de discretizar las señales análogas y continuas, que finalmente serán llevadas a cantidades binarias.

La Figura 78 representa una señal analógica continua entre los puntos  $t_0$  y  $t_1$ , que desde el punto de vista de la magnitud será muestreada a un número discreto de valores binarios y que la cantidad de valores se conoce con el nombre de Resolución del sistema [5].

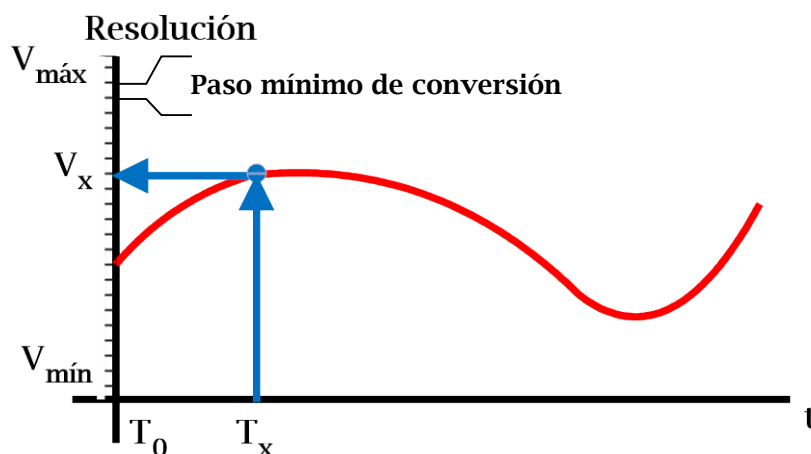


Figura 78. Eje de resolución en la conversión AD

La resolución de un conversor analógico a digital es la cantidad de valores, discretos, en los cuales se interpreta la señal a digitalizar. Por ejemplo, para un procesador con un conversor A/D que tiene una resolución de 12 bits el número de valores discretos en los cuales se puede

valorar a una señal, sería de  $2^{12} = 4096$ . El valor ideal para esos valores sería un número infinito, pero tecnológicamente es imposible.

Esos valores deben estar comprendidos dentro de dos límites, que forman la ventana de conversión o valores de referencia ( $V_{min}$ ,  $V_{max}$ ). Para la Figura 78, el punto P tiene una interpretación en el mundo de lo discreto y es de  $V_x$ .

Se recomienda que el usuario aproveche al máximo la resolución del sistema, adecuando la señal analógica para que excursione de la manera más completa en la ventana de conversión. Por ejemplo, una señal con un valor máximo de 100mV deberá ser amplificada por un factor de 30, para una ventana de conversión de 3V y de ésta manera aprovechar la resolución del sistema.

Para calcular el paso mínimo de conversión y por otro lado conocer el intervalo de pérdida de información, supóngase que se tiene una señal sometida a un conversor de 12 bits de resolución, un  $V_{min} = 0V$  y un  $V_{max} = 5V$ . El paso mínimo de conversión está dado por:

$$\text{Paso mínimo} = (V_{max} - V_{min}) / \text{Resolución}$$

$$\text{Paso mínimo} = (5V - 0V) / 2^{12}$$

$$\text{Paso mínimo} = 1.22mV$$

El cálculo anterior indica que la diferencia en magnitud entre el resultado de una conversión y la inmediatamente superior (o inferior) es de 1.22mV. Todo valor que no sea múltiplo entero de un paso mínimo, se deberá aproximar al valor más cercano y es allí donde un conversor A/D trunca información del mundo analógico.

El **muestreo** (sample) es otra característica importante de un conversor A/D y se refiere a la cantidad de muestras tomadas por unidad de tiempo que se pueden procesar y convertir a cantidades discretas.

La Figura 79 presenta una señal analógica continua entre los puntos  $t_0$  y  $t_1$ , que desde el punto de vista del muestreo es sometida a un número finito de muestras y que cada muestra es tomada a un intervalo constante  $T$ , llamado período de muestreo [18].

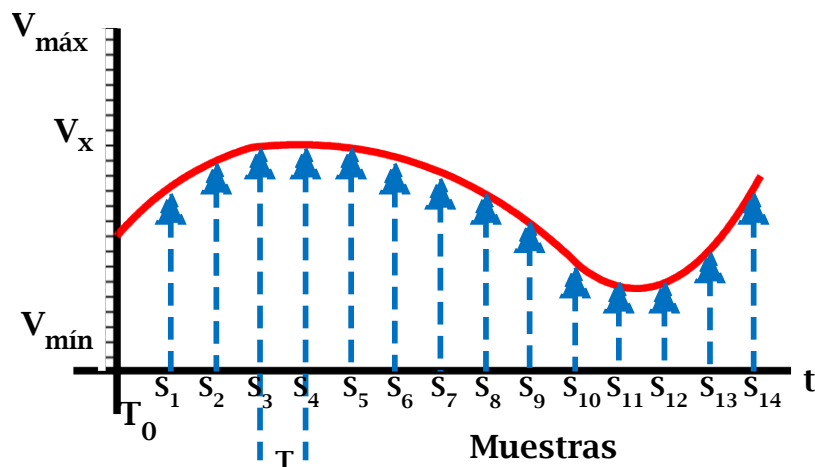


Figura 79. Muestreo en la conversión AD.



Al igual que en la resolución el muestreo introduce pérdida de información, debido a los valores que no son muestreados entre dos intervalos de muestreo contiguos. Idealmente la tasa de muestreo debería ser infinita, pero existen restricciones tecnológicas. Entonces, mientras menor sea la separación entre los  $S_i$  ( $T$  pequeño), mostrados en la Figura 79, más fiel será la señal digitalizada con respecto a la señal analógica original.

## 14.2. Descripción

El ADC en el microcontrolador MC9S08JM60 /32 es un conversor unipolar, de aproximaciones sucesivas disponible para una resolución de 8 bits a 12-bits [5]. Los microcontroladores de Freescale incluyen módulos ADC con una gran variedad de opciones para el usuario, permitiendo de este modo diferentes aplicaciones [18].

### Algunas de las características del módulo ADC son:

- De 4 a 27 canales ADC con una gran variedad de opciones de por medio.
- Dos opciones para la resolución: están disponibles los módulos ADC de 8 a 12-bits, dependiendo de la familia del microcontrolador.
- El tipo de conversión es adaptable a cada aplicación: Permite *conversión de una muestra o continua*, con la opción de *conversión auto-scan* en algunos microcontroladores.
- Soporta ambos métodos de programación: Incluye una *conversión completa por flag* y una *conversión completa por interrupciones*, permitiendo al usuario elegir un método por llamada selectiva o uno basado en interrupciones.
- Frecuencia seleccionable del reloj ADC: Incluye un pre-escalador de reloj, y algunos microcontroladores incluyen la opción de elegir entre el reloj de bus o el cristal externo como reloj de entrada.
- Flexibilidad: El ADC 8-/12-bits contiene cuatro opciones para justificar el resultado de la conversión (12 bits justificados a izquierda, 12 bits justificados a derecha, 12 bits justificados a izquierda con signo, y el modo de truncado de 8 bits).

### Los principales registros del módulo ADC son:

#### 1. El registro de control y estado 1 del ADC (ADSC1)

- Flags (banderas) que indica una conversión ADC completa (el bit 7, COCO)
- habilitación de interrupciones ADC (el bit 6, AIEN)
- Selección de conversiones continuas o conversión única (el bit 5, ADCO)
- Selección de uno de los canales ADC para ser escaneado (bit 4:0, ADCH4:ADCH0)

#### 2. El registro de control y estado 2 del ADC (ADSC2)

- Flags (banderas) que indica que una conversión está en progreso cuando está en 1 (el

bit 7, ADACT)

- selección de disparo de la conversión, si vale 1 la conversión se activa por HW, en caso contrario por SW (el bit 6, ADTRG)
- Si está en 1 habilita la función de comparación (el bit 5, ACFE)
- configura la función de comparación para disparar la conversión cuando el resultado de la conversión sea mayor o igual que el valor comparado (bit 4, ACFGT)

### 3. El registro de configuración (ADCCFG)

- Si el bit 7 vale 1 se configurará para funcionar a bajo consumo (bit 7, ADLPC)
- Selección de divisor del clock de entrada, se puede dividir por 2, 4 y 8 (el bit 6:5, ADIV)
- Si el bit 4 vale 1 permite seleccionar un tiempo de conversión largo (conversión más lenta) (bit 4, ADLSMP)
- Permite seleccionar una conversión de 8, 10 o 12 bits (bit 3:2, MODE1:MODE0)
- Permite seleccionar la fuente de clock; puede ser el clock del bus, el clock del bus /2, clock alternativo o clock asincrónico (bit 1:0, ADICLK)

### 4. Registro de conversión parte alta (ADCRH)

- si se configura el ADC para conversiones de 12 bits, este registro contendrá los 4 bits más significativos de la conversión

### 5. Registro de conversión parte baja (ADCRL)

- contendrá los 8 bits menos significativos de la conversión. En el caso de conversión de 8 bits solo se deberá leer este registro

6. Registro de valor de comparación parte alta (ADCCVH), contendrá la parte alta del valor de comparación en el caso de usar el disparo por HW.

7. Registro de valor de comparación parte baja (ADCCVI), contendrá la parte baja del valor de comparación en el caso de usar el disparo por HW.

8. Por último, están los 3 registros de control de pin, que permiten deshabilitar los pines de entrada/salida para que funcionen como entradas analógicas.

## 14.3. Voltaje de conversión

VREFH es la tensión de referencia del conversor AD y puede estar conectado al mismo potencial que la tensión de alimentación del conversor (VDDAD), o puede ser obtenida de una fuente externa cuya tensión no puede superar la de VDDAD. Cuando la tensión VREFH provenga de una fuente diferente a la de VDDAD, el pin VREFL debe estar conectado al mismo potencial de voltaje que VSSAD (potencial de tierra).

## 14.4. Tiempo de conversión

### Inicio de la conversión

- Después de escribir en ADCSC1 (con los bits ADCH no todos en 1) cuando se selecciona la conversión disparada por software
- Después del disparo por hardware (ADHWT) si se selecciona la operación de disparo por hardware.
- Luego de la transferencia del resultado a los registros de datos, cuando se selecciona conversión continua

Si las conversiones continuas están habilitadas, una nueva conversión se inicia automáticamente después de finalizada la conversión actual. En la conversión disparada por software, las conversiones continuas comienzan después de que se escribe ADCSC1 y continúan hasta que se cancela. En la operación desencadenada por hardware, las conversiones continuas comienzan después de un evento de disparo por HW y continúan hasta que se cancelan.

$$\text{tiempo de conversión} = \frac{16 \text{ ciclos de reloj del ADC}}{\text{frecuencia de reloj del ADC}}$$

**Nota:** una conversión se cancela cuando se escribe el ADCS1

## 14.5. Ejemplo de Aplicación

### 14.5.1. Descripción funcional

La descripción funcional del ejemplo consiste en fijar un valor de tensión entre 0 y 5V a la entrada del canal 8 del conversor y visualizar el valor digitalizado de la conversión en los 8 leds conectados al puerto B del MCU [19].

### 14.5.2. Descripción de HW

El HW a utilizar consiste en un potenciómetro conectado al canal AD8 del conversor y 8 leds conectados por el ánodo a los pines del puerto B como muestra la figura 80 [17].

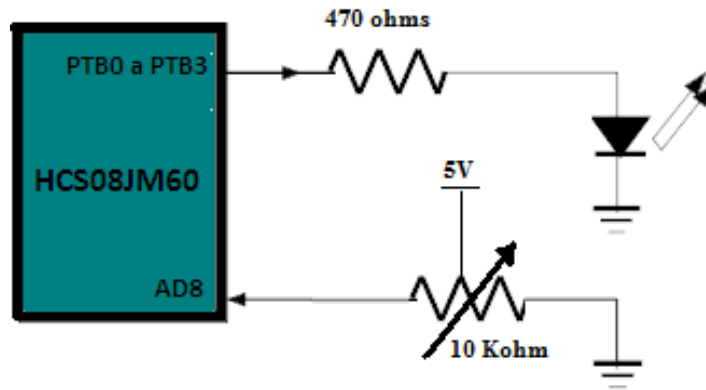


Figura 80. Conexión de la aplicación ADC

### 14.5.3. Descripción del código de la aplicación

Esta aplicación configura el canal 8 del convertidor en modo conversión continua de 8 bits. Luego asigna los 8 bits de la conversión a todo el puerto B que está configurado como salida (leds). El canal 8 del convertidor se encuentra conectado a un potenciómetro que permite digitalizar un valor de tensión entre 0 y 5V, de esta manera a medida que se gire el potenciómetro irá cambiando el estado de los leds.

```
#include <hief.h>           /* macro para habilitar interrupciones */
#include "derivative.h"     /* incluye las declaraciones de los módulos periféricos */

void init_ADC(int);
int valor_convertor;       //variable que almacena la conversión

void main(void)           //programa principal
{
    PTBDD=0xFF;           //configura el Puerto B como salida
    EnableInterrupts;     /* habilita interrupciones */
    init_ADC();           //Inicializa el convertor analógico digital

    for(;;)               //bucle infinito
    {
        ADC();             //lectura de la conversión
        PTED=valor_convertor; //se asigna el valor de la conversión al puerto E
    }                       /* fin de bucle infinito */
}                           //fin de programa principal

//configuración del convertor AD

void init_ADC (void)
{
    ADCSC1 = 0x00;        //Borra Registro
    ADCSC1_ADSC = 1;     //Conversiones Continuas
}
```

```
APCTL1_ADPC1 = 1;    //Deshabilita como I/O
ADCCFG_MODE0 = 0;    //Setea registro de 8 bits
ADCCFG_MODE1 = 0;
ADCSC1_ADCH = 01000; //configura el canal 8 del ADC
```

```
void ADC(int &valor_convertor) //obtención del valor de conversión.
{
    valor_convertor =ADCRL;     //se lee la parte baja del registro de conversión (configuración de 8 bits)
}
```

# CAPÍTULO 15

## Interfaz de comunicación serie (SPI)

*Jorge R. Osio y Walter J. Aróztegui*

### 15.1. Descripción del protocolo SPI

La interfaz periférica serie es una interfaz sincrónica master-slave que se basa en un registro de desplazamiento de 8 bits. El master SPI genera una señal de clock usada por todos los dispositivos SPI para coordinar la transferencia de datos. El dispositivo periférico conectado al SPI también incluye un registro de desplazamiento. Juntos, los 2 registros de desplazamiento de 8 bits son conectados desde un registro de rotación a izquierda de 16 bits. Una transferencia de datos consiste de un desplazamiento de 8 bits, el cual resulta en una transferencia de datos entre el dispositivo master y el slave. Muchos dispositivos, tales como los conversores AD, conversores DA, sensores y chips de memoria Flash, SRAM, FRAM, SD, entre otros, poseen interfaz SPI.

#### 15.1.1. Breve repaso de las comunicaciones seriales sincrónicas

Una comunicación sincrónica es aquella en donde los datos se envían sincronizados con una señal de clock, ya sea como una línea independiente o embebida dentro de la misma.

Generalmente existe un dispositivo maestro, que es el generador de la sincronía de la comunicación. De tal manera que el reloj es generado en una línea independiente del sistema y es el maestro quien lo inserta en el canal de comunicación.

Los demás dispositivos del sistema actúan como esclavos de la comunicación y la señal de reloj entra por un pin a cada uno de ellos, estableciendo el sincronismo de los bits de información que llegan o salen.

Otros sistemas utilizan la misma señal de datos, para generar el sincronismo de los bits. La Figura 81 ilustra el protocolo Manchester, del cual se puede extraer el reloj del sistema.

Basta con tomar cualquier flanco de la señal de clock para lograr una relación entre, (utilizando el flanco) el reloj y los datos.

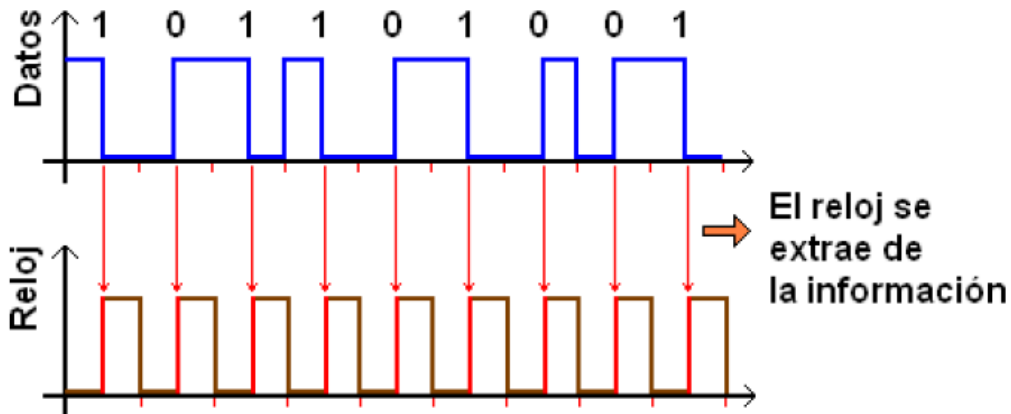


Figura 81. Sincronismo entre señal de clock y datos.

En los sistemas seriales sincrónicos para la interconexión de periféricos, se manejan distancias cortas y modos de un solo maestro y múltiples esclavos.

Un maestro podría perder su función y convertirse en un esclavo, mediante un mecanismo de arbitraje de bus. De esta manera cualquier dispositivo podría ser maestro en un momento determinado.

Las tramas de comunicación son de variada presentación y algunas sólo involucran la carga útil, sin adicionar bits que permitan el handshaking (coordinación) entre esclavos y maestro. Para estos protocolos deberá existir una línea tanto para el dato saliente (TxD) como para el dato entrante (RxD), así como la posibilidad de hacer comunicación simultánea y bidireccional (Full Duplex) (Ejemplo: SPI).

Con  $COPH=0$  (fase= $0^\circ$ ) se logra una configuración del módulo, en donde el dato a enviar se mantiene estable durante el primer flanco del período de clock y el siguiente bit se envía en el primer flanco del segundo período de clock, tenga este cualquiera de las 2 polaridades posibles (línea normalmente en alto o en bajo).

Con  $COPH=1$  (fase= $180^\circ$ ) se logra una configuración del módulo, en donde cada dato se mantiene estable durante el segundo flanco del período de clock.

## 15.2. Características del módulo SPI

El módulo SPi incluye las siguientes características:

- Operación en modo master o slave
- Modo bidireccional en Full-duplex o simple línea
- Tasa de bit de transmisión programable
- Registro de datos de transmisión y recepción de Doble buffer
- Opciones de polaridad y fase del clock
- Salida de selección de slave
- Flag de modo de error por falla con posibilidad de interrupción de CPU

- Control de operación de SPI durante el modo wait
- Desplazamiento de datos primero MSB o primero LSB seleccionable
- Longitud de datos de transmisión programable a 8 o 16 bits

### 15.3. Descripción funcional del módulo SPI

El módulo se habilita seteando el bit “SPI enable” (SPE) en el Registro “SPI Control Register 1”. Mientras el bit SPE esté seteado, los cuatro pines asociados al módulo SPI cumplirán las funciones de:

- Selector de esclavo (SS)
- Clock serial (SPSCK)
- Master out/slave in (MOSI)
- Master in/slave out (MISO)

Una transferencia SPI se inicia en el dispositivo master leyendo el registro de estados SPI (SPIxS) cuando SPTEF = 1 y escribiendo el dato en el buffer de transmisión de datos (escribiendo SPIxDH:SPIxDL). Cuando se completa una transferencia, el dato recibido se almacena en el buffer de datos recibidos. Los Registros SPIxDH:SPIxDL actúan como el buffer de datos de recepción de datos SPI cuando es leído y como el buffer de transmisión de datos SPI cuando se escribe.

El bit de control de fase de clock (CPHA) y el bit de control de polaridad de clock (CPOL) ubicados en el registro de control 1 SPI (SPIxC1) permiten seleccionar uno de los cuatro posibles formatos de clock a ser usados por el sistema SPI. El bit CPOL selecciona una inversión o no de polaridad en el clock. El bit CPHA se usa para coordinar dos protocolos para el muestreo de datos sobre los flancos impares llamados SPSCK o sobre los flancos pares llamados SPSCK.

El SPI puede configurarse como master o como slave. Cuando se setea el bit MSTR del registro de control 1, se habilita como master, cuando se borra el bit MSTR, se selecciona el modo slave.

#### 15.3.1. Modo master

El Módulo SPI opera en modo master cuando el bit MSTR se encuentra seteado. Solo en modo master se pueden iniciar transmisiones. Una transmisión se inicia leyendo el registro SPIxS mientras el SPTEF = 1 y escribiendo el registro de datos SPI. Si el registro de desplazamientos está vacío, el byte se transfiere inmediatamente a este registro para ser enviado.



El dato comienza a desplazarse hacia el pin de salida MOSI bajo la referencia del clock de control serial.

- SPSCCK: Los bits de selección de baud rate SPR2, SPR1, y SPR0 conjuntamente con los bits de preselección de baud rate SPPR2, SPPR1, y SPPR0 en el registro de control generan un baud rate y determinan la velocidad de transmisión. El pin SPSCCK es la salida de clock SPI. A través de este pin el generador de baud rate del master controla el registro de desplazamiento del periférico slave.

- Los pines MOSI, MISO en modo master, tienen la función de pin de datos de salida (MOSI) y pin de datos de entrada (MISO), esto está definido por los bits de control SPC0 y BIDIROE.

- El pin SS cuando los bits MODFEN y SSOE están seteados, se configura como selector de slave de salida. El pin SS de salida se pone en bajo para una transmisión y en alto cuando la línea está en estado desocupado "idle".

Si el bit MODFEN está seteado y el SSOE borrado, el pin SS se configura como entrada para detectar el modo de error por falla en la transmisión.

Si la entrada SS está en bajo, indica un error por falla cuando otro master trata de manejar las líneas de MOSI y SPSCCK. En este caso, el SPI cambia a modo slave, borrando el bit MSTR y deshabilitando el buffer de salida slave MISO.

### 15.3.2. Modo slave

El SPI funciona en modo slave cuando el bits MSTR en el registro de control 1 está borrado.

- SPSCCK: En modo slave, SPSCCK es el clock entrada que llega desde el master.
- Pines MISO, MOSI: En modo slave, el pin (MISO) tiene la función de salida de datos y el pin (MOSI) tiene la función de entrada de datos, esto se determina por los bits SPC0 y BIDIROE en el registro de control 2.

- Pin SS: El pin SS es la entrada de selección del slave. Antes de una transmisión de datos, el pin SS del slave debe estar en bajo y debe mantenerse en este estado hasta que finalice dicha transmisión. Si el pin SS está en alto, el SPI se fuerza a estado idle.

La entrada SS también controla el pin de salida de datos seriales, Si el SS está en alto (no detectado), el pin de salida de datos estará en alta impedancia, y, si SS está en bajo el primer bit en el registro de datos SPI se enviará al pin de salida de datos seriales. También si el SS del slave está en alto, se ignora la señal de clock enviada por el master.

Con el sistema de varios slaves, es posible implementar una comunicación serial enviando a varios slaves la misma transmisión desde el master, aunque el master no recibirá información de todos los slaves a la vez.

Si el bit CPHA en el registro de control 1 está borrado, un número impar de flancos sobre la entrada SPSCCK causará que el dato se mantenga en el pin de entrada de datos. Un número par de flancos causará que el valor previamente almacenado desde el pin de entrada de datos

seriales se desplace en el LSB o MSB del registro de desplazamientos del SPI, dependiendo del bit LSBFE.

Si se setea el bit CPHA, un número par de flancos en la entrada del SPSCCK causará que el dato se mantenga en el pin de entrada de datos serie. Un número impar de flancos causará que el valor previamente almacenado se desplace desde el pin de entrada serie al LSB o MSB del registro de desplazamientos, dependiendo del bit LSBFE.

Cuando el CPHA es seteado, el primer flanco se usa para enviar el primer bit de datos al pin de salida serie. Cuando el CPHA es borrado y la entrada SS está en bajo (slave seleccionado), el primer bit de datos del SPI se envía por el pin de salida de datos. Después del octavo (SPIMODE = 0) o el dieciseisavo (SPIMODE = 1) desplazamiento, la transferencia se considera finalizada y el dato recibido es transferido al registro de datos del SPI. Para indicar que la transferencia se completó, se setea el flag SPRF en el Registro de estados del SPI.

## 15.4. Registros de configuración

### 15.4.1. Registro de control 1 (SPIxC1)

SPIE (bit 7): Habilitación de interrupción (para SPRF y MODF)

SPE (bit 6): Habilitación de sistema

SPTIE (bit 5): Habilitación de interrupción por transmisión

MSTR (bit 4): Selección de modo Master/Slave

CPOL (bit 3): polaridad de Clock

CPHA (bit 2): Fase de clock

SSOE (bit 1): habilitación de salida de selección de slave

LSBFE (bit 0): primer bit LSB (dirección de desplazamiento)

**Tabla 13. Bits de selección de modo Master o Slave**

MODFEN	SSOE	Modo Master	Modo Slave
0	0	I/O de propósito general	Entrada de selección de slave
0	1	I/O de propósito general	Entrada de selección de slave
1	0	Entrada SS para modo falla	Entrada de selección de slave
1	1	Salida SS automática	Entrada de selección de slave

### 15.4.2. Registro de control 2 (SPIxC2)

SPMIE (bit 7): Habilitación de interrupción por coincidencia

SPIMODE (bit 6): Modo 8 o 16 bits

MODFEN (bit 4): Habilita función en modo falla como master

BIDIROE (bit 3): habilita salida en modo bidireccional  
 SPISWAI (bit 1): Stop en modo wait  
 SPC0 (bit 0): Control de pin 0

**Tabla 14. Modos de Operación**

Modo de Pin	SPC0	BIDIROE	MISO	MOSI
<b>Operación de Modo Master</b>				
<b>NORMAL</b>	0	X	Entrada Master	Salida Master
<b>BIDIRECCIONAL</b>	1	0	MISO no usado por SPI	Entrada Master
		1		Master E/S
<b>Operación de Modo slave</b>				
<b>NORMAL</b>	0	X	Salida Slave	Slave
<b>BIDIRECCIONAL</b>	1	0	Entrada Slave	MOSI no usado por SPI
		1	Slave E/S	

### 15.4.3. Registro de baud rate (SPIxBR)

Bits 6:4 del registro SPIxBR (SPPR[2:0]): Pre-escalador Divisor de Baud Rate

Bits 2:0 del registro SPIxBR (SPR[2:0]): Divisor de Baud Rate

**Tabla 15. Divisor pre-escalador**

SPR2:SPR1:SPR0	Divisor de tasa
0:0:0	<b>2</b>
0:0:1	<b>4</b>
0:1:0	<b>8</b>
0:1:1	<b>16</b>
1:0:0	<b>32</b>
1:0:1	<b>64</b>
1:1:0	<b>128</b>
1:1:1	<b>256</b>

**Tabla 16. Divisor de Baud Rate**

SPPR2:SPPR1:SPPR0	Pre-escalador Divisor
0:0:0	1
0:0:1	2
0:1:0	3
0:1:1	4
1:0:0	5
1:0:1	6
1:1:0	7
1:1:1	8

$$\text{Baud rate} = \text{CGMOUT} / (2 \times \text{BD})$$

#### 15.4.4. Registro de estados (SPIxS)

SPRF (bit 7): Flag de buffer de lectura lleno

SPMF (bit 6): Flag de Match. Cuando el valor en el buffer de recepción coincide con el valor en SPIMH:SPIML

SPTEF (bit 5): Flag de buffer de transmisión vacío

MODF (bit 4): Flag modo master por falla

#### 15.4.5. Registros de datos SPI (SPIxDH:SPIxDL)

En modo 8-bit, solo está disponible SPIxDL. La lectura de SPIxDH retornará todos ceros. La escritura en SPIxDH será ignorada.

En modo 16-bit, leyendo cualquier byte (SPIxDH o SPIxDL) se guarda el contenido de ambos bytes en el buffer en donde se mantiene hasta que se lea otro byte. Escribiendo cualquier byte (SPIxDH o SPIxDL) se mantiene el valor en el buffer. Cuando ambos bytes han sido escritos, se transferirá como un valor de 16 bits en el buffer de transmisión de datos.

#### 15.4.6. Registros de coincidencia (SPIxMH:SPIxML)

En modo 8-bit, solo está disponible SPIxML, la lectura de SPIxMH retornará ceros y la escritura a SPIxMH será ignorada.

En modo 16-bit, leyendo cualquier byte (SPIxMH o SPIxML) se mantiene el contenido de ambos bytes en un buffer, en donde se conservará hasta la lectura de otro byte. Escribiendo cualquier byte (SPIxMH o SPIxML) se transfiere el valor a un buffer. Cuando ambos buffer hayan sido escritos, se transferirá el contenido como un valor a los registros de coincidencia del SPI.

# APENDICE A

## Sistemas de numeración y código

*Jorge R. Osio*

Las CPUs trabajan adecuadamente con información en un formato diferente al que la gente está acostumbrada a manejar a diario. Típicamente se trabaja en el sistema de numeración de base 10 (decimal, con niveles de 0 a 9). Las computadoras digitales binarias trabajan con el sistema de numeración en base 2 (binario, 2 niveles), puesto que éste permite representar cualquier información mediante un conjunto de dígitos, que solo serán “ceros” (0) ó “Unos” (1).

Un **uno** o un **ceros** puede representar la presencia o ausencia de un nivel lógico de tensión sobre una línea de señal, o bien el estado de **encendido** o **apagado** de una simple llave.

En este apartado se explican los sistemas de numeración más comúnmente utilizados por las computadoras, es decir, **binario**, **hexadecimal**, **octal** y **binario codificado en decimal (BCD) [9] y [12]**.

Las CPUs además, usan códigos especiales para representar información del alfabeto o sus instrucciones. La comprensión de estos códigos ayudará a entender cómo una computadora se las ingenia para entender cadenas de dígitos que sólo pueden ser unos o ceros.

### A.1. Números binarios y hexadecimales

Para un número decimal (base 10) el peso (valor) de un dígito es diez veces mayor que el que se encuentra a su derecha. El dígito del extremo derecho de un número decimal entero, es el de las unidades, el que está a su izquierda es el de las decenas, el que le sigue es el de las centenas, y así sucesivamente.

Para un número binario (base 2), el peso de un dígito es dos veces mayor que el que se encuentra a su derecha. El dígito del extremo derecho de un número binario entero, es el de la unidad, el que está a su izquierda es el de los duplos, el que le sigue es el de los cuádruplos, el que le sigue es el de los óctuplos, y así sucesivamente. Para algunas computadoras es habitual trabajar con números de 8, 16 o bien 32 dígitos binarios.

El sistema de numeración de **base 16 (hexadecimal)** resulta ser una práctica solución de compromiso. Con un dígito hexadecimal se puede representar exactamente igual, a cuatro dígitos binarios, de este modo un número binario de 8 dígitos se puede expresar mediante dos dígitos hexadecimales. La sencilla correspondencia que existe entre las representaciones de un dígito hexadecimal y la de cuatro dígitos binarios, permite realizar mentalmente la conversión entre

ambos. Para un número **hexadecimal (base 16)**, el peso de un dígito es de dieciséis, respecto al que se encuentra a su derecha. El dígito del extremo derecho de un número hexadecimal entero, es de las unidades, el que está a su izquierda es el de los décimo séxtuplos, y así sucesivamente. La tabla 17 muestra la relación existente, en la representación de valores, en decimal, binario y hexadecimal. Estos tres diferentes sistemas de numeración resultan ser tres modos diferentes de representar físicamente las mismas cantidades. Se utilizan las letras de la A a la F para representar los valores hexadecimales correspondientes que van del 10 al 15, ya que cada dígito hexadecimal puede representar 16 cantidades distintas. Si se considera que el sistema de representación sólo incluye diez símbolos (del 0 al 9); por lo tanto, se debe recurrir a algún otro símbolo de un solo dígito, para de esta manera, representar los valores hexadecimales que van del 10 al 15. Para evitar confusiones acerca de si un número es hexadecimal o decimal, se antepone un símbolo \$ a cada cantidad hexadecimal. Por ejemplo, 64 es el decimal “sesenta y cuatro”; entonces \$64 es hexadecimal “seis-cuatro”, que es equivalente al decimal 100.

**Tabla 17. Equivalentes entre decimal binario y hexadecimal.**

Base 10 Decimal	Base 2 Binaria	Base 16 Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F
16	0001 0000	10
17	0001 0001	11
100	0110 0100	64
255	1111 1111	FF
1024	0100 0000 0000	400
65535	1111 1111 1111 1111	FFFF

El hexadecimal es el modo adecuado tanto para expresar como para discutir acerca de información numérica procesada por una computadora, ya que a la gente le resulta fácil realizar la conversión entre un dígito hexadecimal y sus 4 bits equivalentes. La notación hexadecimal es mucho más compacta que la binaria mientras se mantienen las connotaciones binarias.

## A.2. Código ASCII

Las CPUs deben manejar otros tipos de información además de los números. Tanto los textos (caracteres alfanuméricos) como las instrucciones deben codificarse de tal modo que la CPU interprete esta información. El código más común para la información tipo texto es el **American Standard Code for Information Interchange (ASCII)** [9]. El código ASCII es una correlación ampliamente aceptada entre caracteres alfanuméricos y valores binarios específicos. En este código, el número \$41 corresponde a una letra A mayúscula, el \$20 al carácter espacio, etc. El código ASCII traduce un carácter a un código binario de 7 bits, aunque en la práctica la mayoría de las veces la información es transportada en caracteres de 8 bits con el bit más significativo en cero. Este estándar hace posible las comunicaciones entre equipos hechos por diversos fabricantes, puesto que todas las máquinas utilizan el mismo código. La Tabla 18 muestra la relación existente entre los caracteres ASCII y los valores hexadecimales.

**Tabla. 18. Conversión de ASCII a Hexadecimal**

ASCII	Hex	Símbolo	ASCII	Hex	Símbolo	ASCII	Hex	Símbolo	ASCII	Hex	Símbolo
0	0	NUL	32	20	(espacio)	64	40	@	96	60	`
1	1	SOH	33	21	!	65	41	A	97	61	a
2	2	STX	34	22	"	66	42	B	98	62	b
3	3	ETX	35	23	#	67	43	C	99	63	c
4	4	EOT	36	24	\$	68	44	D	100	64	d
5	5	ENQ	37	25	%	69	45	E	101	65	e
6	6	ACK	38	26	&	70	46	F	102	66	f
7	7	BEL	39	27	'	71	47	G	103	67	g
8	8	BS	40	28	(	72	48	H	104	68	h
9	9	TAB	41	29	)	73	49	I	105	69	i
10	A	LF	42	2A	*	74	4A	J	106	6A	j
11	B	VT	43	2B	+	75	4B	K	107	6B	k
12	C	FF	44	2C	,	76	4C	L	108	6C	l
13	D	CR	45	2D	-	77	4D	M	109	6D	m
14	E	SO	46	2E	.	78	4E	N	110	6E	n
15	F	SI	47	2F	/	79	4F	O	111	6F	o
16	10	DLE	48	30	0	80	50	P	112	70	p
17	11	DC1	49	31	1	81	51	Q	113	71	q
18	12	DC2	50	32	2	82	52	R	114	72	r
19	13	DC3	51	33	3	83	53	S	115	73	s
20	14	DC4	52	34	4	84	54	T	116	74	t
21	15	NAK	53	35	5	85	55	U	117	75	u
22	16	SYN	54	36	6	86	56	V	118	76	v
23	17	ETB	55	37	7	87	57	W	119	77	w
24	18	CAN	56	38	8	88	58	X	120	78	x
25	19	EM	57	39	9	89	59	Y	121	79	y
26	1A	SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	FS	60	3C	<	92	5C	\	124	7C	
29	1D	GS	61	3D	=	93	5D	]	125	7D	}
30	1E	RS	62	3E	>	94	5E	^	126	7E	~
31	1F	US	63	3F	?	95	5F	_	127	7F	□

### A.3. Códigos de operación

Las CPUs utilizan otros códigos para darle instrucciones a la CPU. Este código se denomina **código de operación (opcode)**. Cada código de operación instruye a la CPU en la ejecución de una muy específica secuencia de etapas que debe seguirse para cumplir con la operación propuesta. Las computadoras de distintos fabricantes usan diferentes repertorios de códigos de operación, previstos en la lógica cableada de la CPU. El **repertorio (set) de instrucciones** para una CPU es el conjunto de instrucciones que ésta es capaz de realizar. Los códigos de operación son una representación del set de instrucciones y los mnemónicos son otra. Aún cuando difieren de una CPU a otra, todas las CPUs digitales binarias realizan el mismo tipo de tareas básicas de modo similar. La CPU en el MCU MC68HC05 puede entender 140 instrucciones básicas. Algunas de éstas presentan mínimas variaciones, cada una de las cuales requiere su propio código de operación. El set de instrucciones del MC68HC08 incluye 288 opcodes distintos.

### A.4. Mnemónicos y ensambladores

Un opcode tal como \$4C es interpretado por la CPU, pero no es fácilmente manejable por una persona. Para resolver este problema, se usa un sistema de **mnemónicos** de instrucción equivalentes. El opcode \$4C corresponde al mnemónico **INCA**, que se lee "incrementar el acumulador". La relación entre el mnemónico de cada instrucción y el opcode que la representa, es rara vez utilizada por el programador puesto que el proceso de transducción lo realiza automáticamente un programa de computadora específico denominado **ensamblador**. Este programa es el que convierte los mnemónicos de las instrucciones de un programa en una lista de **códigos de máquina** (códigos de operación e información adicional), para que puedan ser utilizados por la CPU.

Un ingeniero desarrolla un grupo de instrucciones para una CPU en la forma de mnemónicos y luego utiliza un ensamblador para trasladar estas instrucciones a los opcodes que la CPU pueda entender.

### A.5. Octal

Antes de comenzar la discusión sobre los sistemas de numeración y códigos, se describieron dos códigos más muy nombrados. La notación **octal (base 8) [9]** fue usada para trabajar con algunas computadoras primitivas, pero rara vez es utilizada en la actualidad. Esta usa los números que van del 0 al 7 para representar un conjunto de tres dígitos binarios de un modo análogo a la hexadecimal, en donde se recurre a un conjunto de cuatro dígitos binarios. El sistema octal tiene la ventaja de no necesitar símbolos no numéricos (como los símbolos hexadecimales ya vistos, que van de la "A" a la "F").



**Tabla 19. Conversión de Decimal, Hexadecimal y Binario a Octal.**

Dec.	Hex.	Oct.	Bin.	Dec.	Hex.	Oct.	Bin.
0	0	000	00000000	32	20	040	00100000
1	1	001	00000001	33	21	041	00100001
2	2	002	00000010	34	22	042	00100010
3	3	003	00000011	35	23	043	00100011
4	4	004	00000100	36	24	044	00100100
5	5	005	00000101	37	25	045	00100101
6	6	006	00000110	38	26	046	00100110
7	7	007	00000111	39	27	047	00100111
8	8	010	00001000	40	28	050	00101000
9	9	011	00001001	41	29	051	00101001
10	A	012	00001010	42	2A	052	00101010
11	B	013	00001011	43	2B	053	00101011
12	C	014	00001100	44	2C	054	00101100
13	D	015	00001101	45	2D	055	00101101
14	E	016	00001110	46	2E	056	00101110
15	F	017	00001111	47	2F	057	00101111
16	10	020	00010000	48	30	060	00110000
17	11	021	00010001	49	31	061	00110001
18	12	022	00010010	50	32	062	00110010
19	13	023	00010011	51	33	063	00110011
20	14	024	00010100	52	34	064	00110100
21	15	025	00010101	53	35	065	00110101
22	16	026	00010110	54	36	066	00110110
23	17	027	00010111	55	37	067	00110111
24	18	030	00011000	56	38	070	00111000
25	19	031	00011001	57	39	071	00111001
26	1A	032	00011010	58	3A	072	00111010
27	1B	033	00011011	59	3B	073	00111011
28	1C	034	00011100	60	3C	074	00111100
29	1D	035	00011101	61	3D	075	00111101
30	1E	036	00011110	62	3E	076	00111110
31	1F	037	00011111	63	3F	077	00111111

Cambiar la notación hexadecimal usada hoy en día por la octal acarrea dos desventajas. La primera es que las CPUs utilizan "words" (palabras) de 4, 8, 16 ó 32 bits, estas "palabras" no resultan fácilmente fraccionables en grupos de tres bits (algunas CPUs muy antiguas usaban palabras de 12 bits siendo divisibles en cuatro grupos de tres bits). La segunda, es el hecho de carecer de compactibilidad con la hexadecimal, en cuanto a cantidad de bits necesarios para la representación. Por ejemplo, el valor ASCII de la letra "A" mayúscula es **1000001(base 2)**, **41(base 16)** en hexadecimal y **101 (base 8)** en octal. Cuando se menciona el valor ASCII para la "A", es más fácil decir "**cuarto - uno**" que "**uno - cero - uno**".

La tabla 19 presenta las correlaciones entre octal y binario. La columna "binario directo" muestra dígito por dígito el pasaje de los dígitos octales a grupos de 3 bits. Cada grupo de 4 bits se convierte directamente en un dígito hexadecimal.

Cuando mentalmente se convierten valores octales a valores binarios de un byte, el valor octal resultante se representa mediante 3 dígitos octales. Cada dígito octal representa a su vez 3 bits con lo que resulta un bit extra (3 dígitos x 3 bits = 9 bits). Típicamente se opera de

izquierda a derecha, lo que hace fácil olvidarse del tratamiento que debe recibir el bit extra del extremo izquierdo (noveno) de un octal. Cuando se convierte de hexadecimal a binario, resulta más fácil puesto que cada dígito hexadecimal se transforma exactamente en cuatro bits. Dos dígitos hexadecimales coinciden exactamente con los ocho bits de un byte.

## A.6. Binario Codificado en Decimal

El sistema **Binario Codificado en Decimal (BCD)** es una notación híbrida usada para expresar valores decimales en forma binaria. Un BCD utiliza cuatro bits para representar cada dígito decimal.

De esta manera cuatro dígitos binarios pueden expresar 16 diferentes cantidades físicas, habiendo seis combinaciones consideradas no válidas (específicamente, los valores hexadecimales de la “A” a la “F”). Los valores BCD se representan con el signo “\$”, pues ellos son números hexadecimales que representan cantidades decimales.

Cuando la CPU hace una operación de suma BCD, ella realiza una suma binaria y luego realiza un ajuste que genera un resultado BCD. Como un simple ejemplo, se realiza la siguiente suma BCD.

$9 + 1 = 10$ ; en decimal.

La computadora hace la siguiente suma:

$0000\ 1001 + 0000\ 0001 = 0000\ 1010$ ; en binario.

Pero 1010 en binario es equivalente a “A en Hexadecimal” que es un código BCD no válido. Cuando la CPU termina el cálculo, realiza un chequeo para ver si el resultado es un código BCD válido. Si hubo un “acarreo” (un desborde) de un dígito BCD a otro o si Hubiese algún código no válido, se desencadena una secuencia de etapas para corregir el resultado y llevarlo al formato BCD apropiado. El número 0000 1010 (en binario) es corregido y se transforma en 0001 0000(en binario) o 10(en BCD). Esta corrección consiste en analizar el resultado y determinar si en un dígito BCD (4 bits) se superó el valor 1001, de ser así, se deberá suma 6 o 0110 en binario para lograr el desborde y poder representar un dígito decimal en 4 bits.

En la mayoría de los casos es ineficiente utilizar la notación BCD para los cálculos de la CPU. Es mejor convertir la información de decimal a binario en el momento de su ingreso, realizar todos los cálculos en binario y convertirlos nuevamente a BCD o decimal sólo si es necesario presentarlos en pantalla.

No todos los microcontroladores son capaces de realizar cálculos en BCD ya que se debe tener la indicación del acarreo dígito a dígito que no está presente en todas las CPU (los MCUs de Freescale tienen este indicador de semi acarreo). Forzar a una computadora a comportarse como nosotros resulta menos eficiente que permitirle trabajar en su sistema de numeración natural.

## A.7. Símbolos usados en assembler

En esta sección se detallan los símbolos utilizados para representar los distintos sistemas de numeración en assembler [4].

- El símbolo `!` indica que el número es decimal. Este número será trasladado a un valor binario antes de ser almacenado en memoria para ser usado por la CPU.
- El símbolo `$` precediendo a un número indica que el número es hexadecimal; por ejemplo `$24` es 24 (base 16) en hexadecimal o el equivalente de 36 (base 10).
- El símbolo `#` indica un operando y el número es buscado en la posición de memoria siguiente a la del código de operación. Se puede usar una variedad de símbolos y expresiones siguiendo al carácter `#`.

**Nota:** Ya que no todos los ensambladores usan las mismas reglas de sintaxis ni los mismos caracteres especiales, es necesario referirse a la documentación del ensamblador que se deseé usar, en la tabla 20 se muestran los símbolos usados en los microcontroladores de Freescale.

**Tabla 20. Símbolos en assembler**

Prefijo	Significado
<code>!</code>	Decimal
<code>\$</code>	Hexadecimal
<code>@</code>	Octal
<code>%</code>	Binario
<code>'</code> (apóstrofe)	Carácter ASCII

# APENDICE B

## Set de instrucciones

*Jorge R. Osio*

El set de instrucciones de la familia HC908 [4], es una versión muy mejorada y ampliada del set de instrucciones de la flia. anterior. Esta característica hace que los programas realizados para familias anteriores puedan utilizarse en el HC908. El conjunto de instrucciones del HC908 se puede clasificar en las siguientes categorías [5]:

- Movimiento de Datos
- Aritméticas
- Lógicas
- Manipulación de Datos
- Manipulación de Bits
- Control del Programa
- Operaciones BCD
- Especiales

**Nota:** En los ejemplos que siguen a continuación los comentarios se escriben usando el símbolo “;”

### **B.1. Movimiento de Datos**

Las instrucciones de Movimiento de datos se pueden dividir a su vez en:

- Instrucciones de carga de Registros del CPU
- Almacenamiento de registros del CPU
- Operaciones con La Pila
- Movimiento de Datos Registro a Registro
- Movimiento de datos Memoria a Memoria

El agregado de instrucciones que involucran al nuevo registro concatenado H:X de 16 bits como ser LDHX, STHX, otorgan gran flexibilidad en el manejo de tablas y rutinas de acceso indexado, ahorrando código y aumentando la velocidad de ejecución de las mismas.

Además, se puede apreciar que por cada tipo de instrucción, se agrega un nuevo modo de direccionamiento, basado en el uso del Stack Pointer “SP” (puntero de pila) como “segundo registro índice”, lo que facilita el uso de lenguajes de alto nivel como el “C” y otros.

Las instrucciones PUSH y PULL permiten resguardar y rescatar el contenido del Acumulador (ACC) y del puntero índice H:X en memoria RAM, ante sub-rutinas e interrupciones al programa (externas / internas), en forma más rápida y transparente.

Las instrucciones “MOV” en sus diferentes versiones, facilitan el movimiento de datos sin afectar los registros del CPU, de esta forma se consiguen operaciones más rápidas y algoritmos más sencillos. Estas instrucciones son útiles en rutinas de transmisión y recepción de datos en las comunicaciones series asincrónicas SCI (UART) de los distintos MCUs de la familia, o bien en movimientos de datos de una tabla a otra.

## Ejemplo:

```
LDA #A5 ; guarda el número A5 en el acumulador
LDX $80 ; guarda el contenido de la dirección 80 en X
LDA ,X ; carga en el acumulador el contenido de la dirección almacenada en X
STA ,X ; guarda el contenido del acumulador en la dirección a la que apunta X
STA $80 ; guarda el contenido del acumulador en la dirección 80
PSHA ; guarda el contenido del acumulador en la pila
PULA ; extrae el último elemento de la pila y lo guarda en el acumulador
TAP ; guarda el contenido del acumulador en el CCR
TPA ; guarda el contenido del CCR en el A
MOV $80,$85 ; movimiento de memoria a memoria
Mov #5,$80 ; movimiento del valor 5 a la dirección 80
```

## B.2. Aritméticas

Las Instrucciones Aritméticas se pueden clasificar en:

- Instrucciones de Adición
- Instrucciones de sustracción
- Instrucciones de Multiplicación
- Instrucciones de División
- Instrucciones de Complemento y negación
- Instrucciones de comparación
- Otras Instrucciones:
  - Clear

- Chequeo de cero o negativo
- Reserva de espacio en pila
- Reserva de espacio en registro índice

La familia HC908 contiene instrucciones de Multiplicación y de División. La instrucción de multiplicación en el CPU08, es del tipo No signado (sin signo) de 8 x 8 bits y se ejecuta en 5 ciclos de Clock. En los registros "A" y "X" se cargan los valores a multiplicar, el resultado de la operación, se obtiene en los mismos registros "A" y "X", en "A" se encontrará la parte menos "significativa" del resultado, mientras que en "X" se encontrará la parte más significativa del resultado.

La instrucción División en el CPU08 es del tipo No signado (sin signo) de 16 / 8 bits. En el registro "H" se carga la parte más significativa del valor a dividir, en el registro "A" se carga la parte menos significativa de dicho valor, mientras el divisor se carga en el registro "X".

El resultado de la operación, se carga en el registro "A", mientras que el resto o remanente se carga en el "H". Si el resultado de la operación es mayor que "\$ FF", entonces se activará el flag de "CARRY" (C) en el CCR y el valor en el registro "H" será indeterminado.

ADD #06 ; suma el contenido del acumulador y el número 6

ADD \$80 ; suma el contenido del acumulador con el contenido de la dirección 80

ADC \$81 ; suma el contenido del acumulador con el contenido de la dirección 81 y el carry

SUB #\$0A ; resta el contenido del acumulador y el número A

MUL ; producto entre X y A, el resultado se guarda parte alta en X y parte baja en A

DIV; cociente entre H:A y X el resultado se guarda en A y el resto en H

#### MUL

- X contendrá el MSB del producto
- A contendrá el LSB del producto

#### DIV

- H es el MSB del dividendo
- A es el LSB del dividendo
- X no es afectado

COM \$80 ; aplica el complemento a 1

NEG \$80 ; aplica el complemento a 2

### B.3. Lógicas

LDA %10101111

AND #%01010000 ; hace una and bit a bit entre el acumulador y el número 01010000. El

resultado (que en este ejemplo es 00000000) se guarda en el acumulador

ORA #%01010000 ; hace una or bit a bit entre el acumulador y el número 01010000. El

resultado (en este ejemplo es 11111111) se guarda en el acumulador

EOR #%01010000 ; hace una or exclusiva bit a bit entre el acumulador y el número

01010000. El resultado (en este ejemplo es 11111111) se guarda en A

## B.4. Comparación

LDA #09

Cmp #08 ; compara el contenido del acumulador con el número 8

; la comparación se hace mediante una resta. En este ejemplo el contenido

; del acumulador es  $9 - 8 = 1$ . De aquí se concluye que el contenido del

; acumulador es mayor que 8, debido a que la resta dio un número mayor ;a cero.

Mov #5, \$80 ; almacena el número 5 en la dirección 80

LDX #05

CPX \$80 ; compara el registro índice con el contenido de la dirección 80.

; La comparación se hace mediante una resta  $X - (\$80) = 5 - 5 = 0$ . Como la

; resta da cero significa que los números son iguales

## B.5. Manipulación de datos

ROL \$80 ; hace un desplazamiento a izquierda introduciendo lo que estaba en el carry por el

; bit menos significativo y lo que estaba en el bit más significativo va a parar al carry

ROLA ; lo mismo pero la operación de desplazamiento se aplica sobre el acumulador

ROR \$80 ; hace un desplazamiento a derecha del contenido de la dirección 80, introduciendo

; lo que estaba en el carry por el bit más significativo y lo que estaba en el bit menos

; Significativo va a parar al carry

## B.6. Manipulación de bits

Bit \$80 ; testea los bits haciendo una and entre el acumulador y el contenido de la dirección

;80

CLC ; borra el bit de carry

SEC ; setea el bit de carry

Bset n,\$80 ; setea el bit n de la dirección 80

Bclr n, \$80 ; borra el bit n de la dirección 80

## B.7. Control de flujo de ejecución de programa

### B.7.1. Sentencias de saltos

BRA \$EE10 ; salta a la dirección \$EE10

BRCLR n,opr,rel ; salta si el bit n de la dirección opr está en cero

**CBEQ** combina las instrucciones CMP y BEQ, realiza operaciones más rápidas de búsqueda / acceso a tablas.

CBEQ opr,rel ; compara el contenido de opr con A y salta si son iguales

**DBNZ** combina las instrucciones DEC y BNE. Implementa bucles más rápidos y eficientes

DBNZ opr,rel ; decrementa el contenido de opr y salta si es distinto de cero

JMP opr ; permite un salto largo de más de 256 posiciones de memoria  
 JSR opr ; salta a la subrutina con dirección de salto opr  
 BSR rel ;salto a subrutina que se encuentra a menos de 255 posiciones de memoria  
 RTS ; retorno de subrutina

## 2.7.2. Interrupciones

SWI ; interrupción por software  
 RTI; retorno desde una interrupción

## B.8. Operaciones en BCD

DAA ; hace un ajuste decimal en el acumulador lo que está en hexa lo pasa a decimal  
 NSA ; intercambia nibble bajo por nibble alto

## B.9. Especiales

NOP ; no hace nada  
 RSP ; guarda la dirección FF en el stack pointer

### WAIT:

- El CPU08 detiene el procesamiento de instrucciones
- Espera por una interrupción
- No se detiene el oscilador

### STOP:

- El CPU08 detiene el procesamiento de instrucciones
- Detiene el circuito del oscilador
- Pone al MCU en estado “low power”
- Espera por una interrupción

## B.10. Otras operaciones

CLRA ; borra el contenido del acumulador  
 TST \$80 ; testea si es negativo o cero

## Instrucción AIS

**AIS** puede usarse para un rápido alojamiento o desalojo de espacio de la Pila.

- Variables Temporales
- Procesos en “trama”



Ejemplo:

```

SUB1      AIS  #6      ; Aloja 6 bytes en la pila
          AIS  #6      ; Desaloja 6 bytes de la pila
          RTS
    
```

## Instrucción AIX

**AIX** puede usarse:

- Eficiente incremento o decremento del registro **H:X**
  - La instrucción INCX / DECX solo afecta al registro X
  - La instrucción INCX / DECX afecta el CCR, AIX No lo afecta
- Bucles (loops) alrededor de un bloque de memoria
  - Direccionamiento indexado con post incremento solo para incrementos
- Solamente disponible para instrucciones MOV y CBEQ

La instrucción AIX permite “adicionar” en forma inmediata un número signado (positivo o negativo) al registro índice H:X, de esta forma pueden lograrse manejos de tablas más eficientes, búsquedas ascendentes o descendentes a partir de un punto, “saltos” discretos mayores a “1” en una tabla, tanto positivos como negativos. La instrucción AIX no afecta el CCR (registro de código de condiciones), y permite incrementar / decrementar al registro H:X en forma amplia (16bits), y no reducida como las instrucciones INCX / DECX que solo afectan al registro “X”.

Ejemplo: En este ejemplo se calculan 8 bit de checksum para una tabla de 512 bytes.

```

          ORG $0080
TABLA    RMB  512      ; Tabla de datos
          ORG $EE00
          LDHX #511    ; Inicialización del contador de byte
          CLRA         ; Inicializa el Checksum
LASOSUMA ADD  TABLA, X ; Calcula el checksum
          AIX  #-1     ; Decrementa el contador de byte
                          ; Con DECX no hay “carry” desde X a través de H.
                          ; Con AIX si.
          CPHX #0     ; si es cero terminó.
                          ; CPHX setea bits del CCR
          BPL LASOSUMA ; Sigue el lazo si no terminó
    
```

# Referencias

- [1] Web Site "Freescale Semiconductors", [www.freescale.com](http://www.freescale.com) , 2007.
- [2] "Electrocomponentes S.A.", [www.electrocomponentes.com.ar](http://www.electrocomponentes.com.ar), 14/11/2005
- [3] Data sheet: MC68HC908QY4 Microcontrolers, Rev. 5, 07/2005
- [4] Daniel Di Lella, Curso de Microcontroladores HC908, EDU Devices, 2018
- [5] Reference Manual: CPU Central Processor Unit Microprocesador, CPU08RM, Rev. 4, 02/2006.
- [6] Stuart Robb y David Brook, East Kilbride, Scotland Andreas Rusznyak, "Determining MCU Oscillator Start-up Parameters", Nota de Aplicación, Motorola, Geneva, Switzerland. Rev 1.0, December 1998.
- [7] Cathy Cox y Clay Merritt, "Microcontroller Oscillator Circuit Design Considerations", Motorola, 1997.
- [8] Scout Pape, "MC68HC908QY4 Internal Oscilador Usage Notes", Nota de Aplicación, Motorola, julio de 2002.
- [9] James W. Bignell y Robert L. Donovan, "Electrónica Digital" . México: Editorial Continental, 2005.
- [10] H. Taub: "Circuitos Digitales y Microprocesadores", Mc Graw-Hill, 1989.
- [11] Fernando I. Szklanny, Hoarcio Martinez Del Pezzo: "Introducción a los Microprocesadores", Editorial Arbó, 1979.
- [12] A. C. Downton: "Computadores y Microprocesadores: Componentes y Sistemas", Versión en español de Ernesto morales Peake, Addison-Wesley, 1993.
- [13] John D. Carpinelli, "Computer Systems Organization & Architecture", ISBN: 0-201-61253-4
- [14] Parra Miranda Mauro, "Un vistazo a la Arquitectura ARM", Julio 25, 2001
- [15] Douglas H. Summerville, "Embedded Systems Interfacing for Engineers using the Freescale HCS08 Microcontroller II: Digital and Analog Hardware Interfacing", State University of New York at Binghamton, Morgan y Claypool Publishers, 2009.
- [16] Data sheet: MC9S08JM60 Microcontrolers, Rev. 5, 07/2010
- [17] James W. Bignell y Robert L. Donovan, "Electrónica Digital". México: Editorial Continental, 2005.
- [18] J. Feddeler, Lucas Bill, "ADC Definitions and Specifications", División de sistemas de Ingeniería, Austin, Texas, febrero de 2003.
- [19] Luis Reynoso Covarrubias, "Programming the Analog-to-Digital Converter on MC9s08 Microcontrollers", Freescale semiconductor, Nota de aplicación, Julio de 2010.
- [20] Jan Axelson, "Serial Port Complete", tercera edición, Madison, Lakeview research, 2007.

## Los autores

### **Osio, Jorge Rafael**

Recibió el título de Ingeniero en Electrónica en la Facultad de Ingeniería de la Universidad Nacional de La Plata (UNLP), Argentina, en 2007. Se desempeña como docente Universitario en la Facultad de Ingeniería y de Ciencias Exactas de la UNLP. Es Profesor Colaborador del Curso de Posgrado “Introducción al Procesamiento Digital de Imágenes. Realiza tareas de investigación desde el año 2005 en el CENTRO DE TÉCNICAS ANALÓGICAS Y DIGITALES (CeTAD), Facultad de Ingeniería, UNLP. Posee categoría V del programa de incentivos y Categoría III de la Facultad de Ingeniería. Su experiencia de investigación está relacionada con sistemas embebidos y sistemas reconfigurables aplicados a cómputo paralelo, procesamiento digital de imágenes y soluciones tecnológicas. El estudio sobre estos temas, le ha permitido la publicación de numerosos trabajos en revistas especializadas, la presentación en eventos científicos nacionales e internacionales y la formación de recursos humanos.

### **Aróztegui, Walter José**

Recibió el título de Ingeniero en Electrónica en la Facultad de Ingeniería de la Universidad Nacional de La Plata (UNLP), Argentina, en 2005. Se desempeña como docente Universitario en la Facultad de Ingeniería de la UNLP. Realiza tareas de investigación desde el año 2005 en el CENTRO DE TÉCNICAS ANALÓGICAS Y DIGITALES (CeTAD), Facultad de Ingeniería, UNLP. Posee categoría V del programa de incentivos y Categoría IV de la Facultad de Ingeniería. Su experiencia de investigación está relacionada con Sistemas Digitales, Cómputo Paralelo Clústers, Microelectrónica y MEMS. El estudio sobre estos temas, le ha permitido la publicación de numerosos trabajos en revistas especializadas, la presentación en eventos científicos nacionales e internacionales y la formación de recursos humanos.

### **Rapallini, José Antonio**

Es Ingeniero en Telecomunicaciones. Se desempeña como Profesor Titular en la Facultad de Ingeniería de la UNLP. Es Coordinador del CENTRO DE TÉCNICAS ANALÓGICAS Y DIGITALES (CeTAD), Facultad de Ingeniería, UNLP. Posee categoría II del programa de incentivos. Su experiencia de investigación está relacionada con Sistemas Digitales y codiseño HW/SW. El estudio sobre estos temas, le ha permitido la publicación de numerosos trabajos en revistas especializadas, la presentación en eventos científicos nacionales e internacionales y la formación de recursos humanos.

Osio, Jorge Rafael

Sistemas digitales basados en microcontroladores : descripción, funcionalidades y aplicaciones de los microcontroladores basados en el CPU HCS08 / Jorge Rafael Osio ; Walter José Aróztegui ; José Antonio Rapallini. - 1a ed. - La Plata : Universidad Nacional de La Plata ; La Plata : EDULP, 2020.

Libro digital, PDF - (Libros de cátedra)

Archivo Digital: descarga

ISBN 978-950-34-1873-4

1. Ingeniería Electrónica. 2. Microcontroladores. I. Aróztegui, Walter José II. Rapallini, José Antonio III. Título  
CDD 537.5

Diseño de tapa: Dirección de Comunicación Visual de la UNLP

Universidad Nacional de La Plata – Editorial de la Universidad de La Plata  
48 N.º 551-599 / La Plata B1900AMX / Buenos Aires, Argentina  
+54 221 644 7150  
edulp.editorial@gmail.com  
www.editorial.unlp.edu.ar

Edulp integra la Red de Editoriales Universitarias Nacionales (REUN)

Primera edición, 2020  
ISBN 978-950-34-1873-4  
© 2020 - Edulp

**e**  
**exactas**

**Edulp**  
EDITORIAL DE LA UNLP



UNIVERSIDAD  
NACIONAL  
DE LA PLATA